PRIMA TECH'S
GAME DEVELOPMENT SERIES

CD INCLUDED

MULTIPLAYER GAME PROGRAMMING

Todd Barron

Series Editor
André LaMothe
CEO Xtreme Games LLC

PRIMA TECH

Multiplayer Game Programming

PRIMA TECH'S

GAME DEVELOPMENT

Check the Web for Updates

To check for updates or corrections relevant to this book and/or CD-ROM, visit our updates page on the Web at http://www.prima-tech.com/support.

Send Us Your Comments

To comment on this book or any other PRIMA TECH title, visit our reader response page on the Web at http://www.prima-tech.com/comments.

How to Order

For information on quantity discounts, contact the publisher: Prima Publishing, P.O. Box 1260BK, Rocklin, CA 95677-1260; (916) 787-7000. On your letterhead, include information concerning the intended use of the books and the number of books you want to purchase.

Multiplayer Game Programming

Todd Barron

PRIMA TECH'S GAME DEVELOPMENT

PRIMA TECH

A DIVISION OF PRIMA PUBLISHING

A Division of Prima Publishing

Prima Publishing and colophon are registered trademarks of Prima Communications, Inc. PRIMA TECH is a registered trademark of Prima Communications, Inc., Roseville, California 95661.

Publisher: Stacy L. Hiquet
Associate Marketing Manager: Jennifer Breece
Managing Editor: Sandy Doell
Acquisitions Editor: Emi Smith
Project Editor: Cathleen D. Snyder
Technical Reviewer: Greg Perry
Copy Editor: Kris Simmons
Interior Layout: Marian Hartsough Associates
Cover Design: Prima Design Team
CD-ROM Producer: Arlie Hartman
Indexer: Katherine Stimson
Proofreader: Erin Hartshorn

ISBN: 0-7615-3298-6
Library of Congress Catalog Card Number: 00-111679
Printed in the United States of America

00 01 02 03 04 DD 10 9 8 7 6 5 4 3 2 1

For the person who has always given me
support, guidance, and love:
My father

Acknowledgments

First and foremost, I would like to thank my wife Jennie for taking on most of the household duties while I worked on this book. Without her support and encouragement, I never would have been able to accomplish the project.

I would also like to give a big thanks to Prima Publishing for giving me the opportunity to put my thoughts on paper. Emi Smith and Cathleen Snyder have been a joy to work with. I really appreciate the fact that they have always been prompt to answer my many questions and provide valuable assistance. Tech Editor Greg Perry and Copy Editor Kris Simmons also deserve many thanks for their help in editing the book.

Next, I would like to thank Bob Miller for the in-book cartoons and the 3D robot model in SpacePirates. Matt Darr receives my next kudos for helping me debug video initialization routines and providing general testing. Of course, I cannot go without thanking everyone on GameDev.net for their help with testing each revision of the in-book games.

Last, I would like to thank André LaMothe for writing so many books that have contributed to my game programming skills.

Oh yes, I almost forgot. Thanks to my mother for buying me my first computer, a Commodore Vic-20. My father also deserves a lot of credit for my love of computers. You see, when I was writing my first programs at age ten, he told me to quit wasting my time with computers, because they were a passing fad. Thanks Dad—my inherited rebellious nature paid off.

About the Author

Todd Barron started writing computer games at the age of ten, using a friend's Timex Sinclair computer. In 1990, he deviated from the path of a programmer and started what was to end up as Vector X Studios, a company specializing in commercial 3D animation. In 1996, he sold his share of the company and pursued his dream of being a video game programmer. During his time in Philadelphia as Game Development Manager at Merit Industries, he wrote multiple video games and worked on their custom networking solution. In 1999, Todd and his wife moved back to Arkansas, where he accepted a job at Acxiom, a leading information company. Today, Todd works as a Lead Developer at Acxiom, architecting and implementing network solutions for Acxiom's online products. In order to satisfy his game development needs, he operates Lost Logic Inc., a company specializing in multiplayer game development. You can reach Todd at webmaster@lostlogic.com.

Contents at a Glance

Contents

Foreword

Why should you read a book on making MMPs? Because MMPs have the potential to become *the* method of interaction across all borders for this decade. This technology represents an opportunity that has never existed before—game developers can now enact social change in a positive way. I became a game designer at Turbine Entertainment Software, the creators of *Asheron's Call*, because I fervidly want to influence that change.

MMPs provide a fascinating, addictive new way to play games. Some of us make games because we dislike corporate culture, love games, or refuse to "grow up." Some of us make games because we want to change the world. MMPs provide game developers with a way to give players a new forum in which to play, meet people from afar, work together, become friends, and fall in love.

We've read *Snow Crash*; we know where technology, and therefore society, is headed. Creating MMPs allows us to shape the future of human interaction. Who wouldn't want to be a part of that kind of movement?

As a Lead Designer at Turbine, I work with people who have vastly different backgrounds. Anthropologists, architects, philosophers, writers, and a whole host of other people get together at Turbine to make socially significant games. So, go on, read Todd's book. Then go out and make a great game. The world is waiting for us to change it.

Patricia Pizer
Turbine Entertainment Software

Letter from the Series Editor

If there's one thing that many game programmers have no idea how to do, it's networking! It's not that writing network code is hard, or that creating multiplayer games is impossible, it's just not something that many people can learn, simply due to hardware constraints. Let's face it, how many of us have four to eight computers on which to run little networks and game simulations in our homes? Therefore, the people who have knowledge about networking and multiplayer game programming usually come from an industry where they have the hardware and opportunity to learn and develop these types of games. Now, luckily for us, the man who wrote this book, Todd Barron, is just such a person. He is a professional game programmer and an expert in networking and multiplayer game programming. The hard part was getting him to write this book, but with the proper motivation (how's that arm, Todd?), he did it! I'm kidding—Todd wanted to write this book and we are all glad. *Multiplayer Game Programming* contains information that you will not find anywhere else—period! There are no books, no journals, nothing, nada, zilch about this kind of game programming. Sure, you might think that you have seen books on the topic, but they are all extremely simple or they cover general concepts in a broad scope. *Multiplayer Game Programming* is a pedal-to-the-metal, real-world doctrine on one of the most complex areas of game programming, about which very little has been published.

Now at Prima, one of our goals with the Game Development series is to have all of the books cover game programming concepts and teach the main theme of the book, and *Multiplayer Game Programming* is no exception. Todd starts you off with basic and online gaming background, and then covers introductory networking concepts and programming techniques. Then, he moves on to explaining network sockets and creating a networking class. Once you have that under your belt, it's time for a crash course in Windows programming—just enough so that if you're a C/C++ programmer with a heartbeat, you should be able to write a minimal Windows application to support your game's infrastructure. At this point, the book gets freaky. It covers simply everything you could ever want to know about networking and game programming. Networking models such as peer-to-peer, client server, and massive online RPG systems are explained. Synchronization techniques are discussed, compression issues are brought up, and full client and server designs are built and implemented. And did I mention that the book covers just enough Direct3D to allow you to create a 3D game? But don't take my word for it—look at the Table of Contents. This book has it all! This is the book that everyone has been waiting for on the subject of networked and multiplayer game programming. And if that wasn't enough, Todd Barron understands that a picture is worth a thousand words, and he has beautifully annotated this book with figures and other visual aids to help you understand the complex interrelationships of many networking and multiplayer concepts. Put simply, this book rocks!

In today's fast-moving world of game programming technology, you don't have the time or the hardware resources to figure this stuff out from scratch. You need a guide from an expert in the field, who will show you the technology from square one, so you can jumpstart yourself in the fastest growing segment of gaming in the world. And this book is the key. . . .

Sincerely,

[illegible]

André LaMothe
Series Editor

Introduction

Hello and welcome to the first comprehensive book on writing games that play across the Internet or networks. It has been a long time coming, but now you finally have access to a book whose focus is on multiplayer game programming! I remember back in the early '90s, I was feverishly searching for a book on network programming for games. It didn't take very long for me to realize that I would not find such a book, considering the fact that multiplayer games were not popular at the time. It was at that moment that I vowed to write a book about multiplayer game programming once I learned how to do it myself. Well, here you go, my personal vow is now fulfilled, as promised!

What Lies Ahead

When I first started writing the outline for this book, I had a hard time deciding what information I could pack into a single volume. As it turns out, I decided to include information about everything from basic Windows programming to Direct3D with DirectX 8. I think that you will be very pleased with the results. In the pages to come, you will learn about:

- The background of online gaming
- How online games tick
- Basic networking
- Making programs communicate with sockets
- Basic C++ programming
- Windows programming
- DirectPlay without dialog boxes
- DirectPlay peer-to-peer programming
- DirectPlay client-server programming
- Direct3D programming
- DirectInput programming
- Massively multiplayer game design

Hopefully, you will be as excited as I was when I ran my first program that sent a message between two computers. I had written a small sockets program to send a hello message to another PC. It was such a glorious moment of relief the first time it ran, because I realized I had finally figured out network programming! Not that it was a bad thing, but the usual sinking feeling followed, because I realized how much I still had to learn.

Who Can Understand This Stuff Anyway?

One question you might have is, how much do you need to know to use this book? There is no clear answer on that one, but I will give you my two cents worth. Although knowing how to write in C++ is a big benefit, if you know how to write proficiently in at least one computer language, you should be okay. It all really depends on how well you pick up new things. To make this book an easy read, I recommend that you at least know C programming, and you have a basic understanding of how games work.

I also recommend that you have access to the Internet or to a network. Not to worry, though, if you don't have multiple computers lying around your house—all of the programs will work on a single PC.

As far as software goes, you should at least have Windows 95, Windows 98, or Windows 2000 as your operating system. For a compiler, I use Microsoft's Visual C++ 6.0 compiler for all of the source projects, but feel free to use whichever compatible C++ compiler you like.

Working with the CD-ROM

The CD that comes with this book contains some very important software. Be sure to install all of the code to your local machine, as you will need it for the sample applications. Another important piece is the DirectX 8 Software Development Kit. It is required for the programs to work properly. The SDK is included on the CD, and you can refer to Appendix B, "Installing the Software," to learn how to install it.

NOTE

All of the DirectX-related code in this book is based on version 8 of the DirectX SDK.

The Style of the Book

One of the first things you might notice about my writing style is that I follow a pretty loose form of prose. My code follows a basic style, because I like to keep things as simple as possible. I do not use the most complicated code here, because I prefer to keep it clean and easy to read for demonstration purposes.

One of the promises I do make is to help you if you have any problems with the text or code. Please e-mail me at webmaster@lostlogic.com if you have any problems or questions. You can also visit my Web site at http://www.lostlogic.com for updates and articles about game programming.

Enjoy!

Part One

Understanding the Technology

CHAPTER 1

Sup, Wait! It's the History of Online Gaming!

Imagine yourself traveling into another dimension. A dimension of not only sight and sound, but also of the Internet. A journey into a wondrous land whose boundaries are only limited by coders. There's a prompt up ahead—get ready—it's your turn to click OK!

Doo-de-doo-doo, doo-de-doo-doo. All right, I must apologize to ol' Serling for that little abomination; let's move on. First, I want to welcome you to Chapter 1 of this book. As with any good beginning, I am going to start with the foundation of online games—you know they always say history is the best teacher. So sit down for a bit and I'll tell you a little about the history of online gaming.

Before I can discuss the history of online gaming, I must first establish what an online game really is. An online game is nothing more than a game played over some means of communication. The most common form of communication employed by games today is the Internet. I hope you have heard of the Internet by now; if you haven't, wake up! Okay, maybe that's a little harsh; here you go: The Internet is a series of computers connected to a central network.

In this chapter, you will learn the history of the following game types:

- Multiuser dungeons
- Modem games
- Real-time strategy games
- Massively multiplayer online role-playing games

Your Name Will Be MUD

The big question is, how did games evolve out of the confines of single-playerdom? The answer is not all that easy to give because it is a little fuzzy, but I'll take a stab at it. In a land not too terribly far away, massively multiplayer games started appearing on university computer systems. These games were text-based and simple in nature, but massive numbers of students played them for hours on end. The common term used to describe these games is MUD (*Multiuser Dungeon*).

Multiuser dungeons are large-scale games that usually take place in a fantasy-type setting. When you log on to play one of these games, you first create a character to represent yourself in the game world. Usually, you are given a set of points to spend on different attributes, which lets you customize your character. For example, take a look at a set of fictitious character attributes in Table 1.1.

Using these four attributes, you could create a character to your liking. Imagine that you want to create a rugged fighter to smash your enemies. To create a fighter, you assign many points to strength and endurance. This gives your character the ability to inflict massive damage while being able to sustain hits. If you want to create a master of magic, assign most of your points to intelligence and wisdom. When you do so, your character becomes a wise and powerful wielder of spells. My fighter in *Asheron's Call* is named Sakei, and he can be seen in Figure 1.1.

After you create a character, you then enter the game world. In many scenarios, *newbies* start out in a tavern or meeting place.

The term *newbie* commonly refers to a new player to a game. Most players help out newbies, but some predators just wait to scam them.

Once your newbie character is in the game world, you can then begin adventuring around the game. Because most MUD games are text-based, you have to navigate the game with text commands. If you are old enough to remember text adventure games such as the ones by Scott Adams (not the *Dilbert* guy), you should be familiar with the command sets. Most of the commands consist of two words, a verb and a noun. For a list of sample commands, take a peek at Table 1.2.

Table 1.1 Sample MUD Character Attributes

Attribute	Effect
Strength	How strong your character is. A higher strength results in more damage dealt with melee attacks.
Intelligence	How smart your character is. Characters with higher intelligence do better with the schools of magic.
Endurance	A character's ability to sustain movement or activity. Characters with a high endurance can withstand more physical damage.
Wisdom	How wise and knowledgeable a character is. Having a higher wisdom allows a character to understand more complex magic.

Figure 1.1

Sakei, my fighter in Asheron's Call

Table 1.2 Sample MUD Commands

Command	Effect
Move North	Causes your character to move north
Look Target	Gives you a detailed description of the target person, place, or thing
Attack Target	Causes your character to engage the target in combat
Show Inventory	Displays a list of items your character is carrying
Take Item	Causes your character to pick up a certain item

This should give you the idea of the commands used in MUDs. When you enter a command, the game server processes it and sends you a response. For instance, if you move north, you get a description of the new area into which you move. The entire game is played out this way, and the majority of the action takes place in your imagination. This is one of the reasons why people enjoy these types of games—they spur their imaginations.

The cool part about MUDs is that many players play them simultaneously. When you wander around the game world, you run into other players in the game. Everything that your character does can have an effect on other players. For example, if you drop a pile of gold on the ground, another player can pick it up. This type of interaction spawns all sorts of social situations. Once you are no longer playing in a closed game, you have to weigh the consequences of your actions. This brings you to the topic of persistence.

Persistent Worlds

Multiuser dungeons provide a persistent-world experience for players. When an online game is *persistent*, the game world is always present, even when players are offline. You can think of it as work or school. When you go on your lunch break, other people are still busy with their work. The world does not come to a standstill just because you are away; it persists. So unlike *The Sims*, a game you might know of, persistent-world games don't freeze when your computer is shut down.

Being persistent, these types of games result in dramatically different play styles. The main difference for players is the amount of time invested. Persistent-world games require vast amounts of personal time to master. Whereas you might be able to beat the latest iteration of *Resident Evil* in 45 hours, persistent-world games are never truly won.

Reel 'Em In, They're Hooked!

So what makes people play MUDs for hours on end? I like to think that the explanation has three aspects: competition, ownership, and society. Sure, there is a lot more to the reasoning than just three aspects, but they form the roots.

Character Advancement and Competition

The first thing that keeps players going in these worlds is competition. Most people love to compete, and these worlds offer endless opportunities for it. In the previous MUD game example, you looked at some possible character attributes. Not only do characters gain possessions during their lifetimes, but they also get stronger and better at what they do. Some games use skill systems to measure advancement, while others use levels.

When a character starts out in a MUD for the first time, he normally starts at what is called *level one*. A level-one character is fresh to the world and arrives with beginning skills and attributes. As you play characters, they gain experience with their skills. Take, for instance, the example of a fighter character. When the fighter character kills creatures, he gets better at using his sword. To give players a sense of accomplishment, most games use an experience point system. Figure 1.2 illustrates the gap between veteran and newbie characters.

Experience points are generally awarded to characters when they do something that requires use of their skills. So, the fighter gets experience points when he bashes some evil creature. Or, the magic user might get experience points from casting some difficult spell. Once characters have amassed a certain number of experience points, they gain a level. Normally when a character gains a level, he gets to increase his skills or attributes. For example, if a fighter goes from level one to level two, he probably gains some strength and endurance. This process goes a long way to setting higher-level characters apart from lower-level characters. Higher-level characters have much higher attributes and statistics. To get an idea of how levels correspond to experience points, look at Table 1.3.

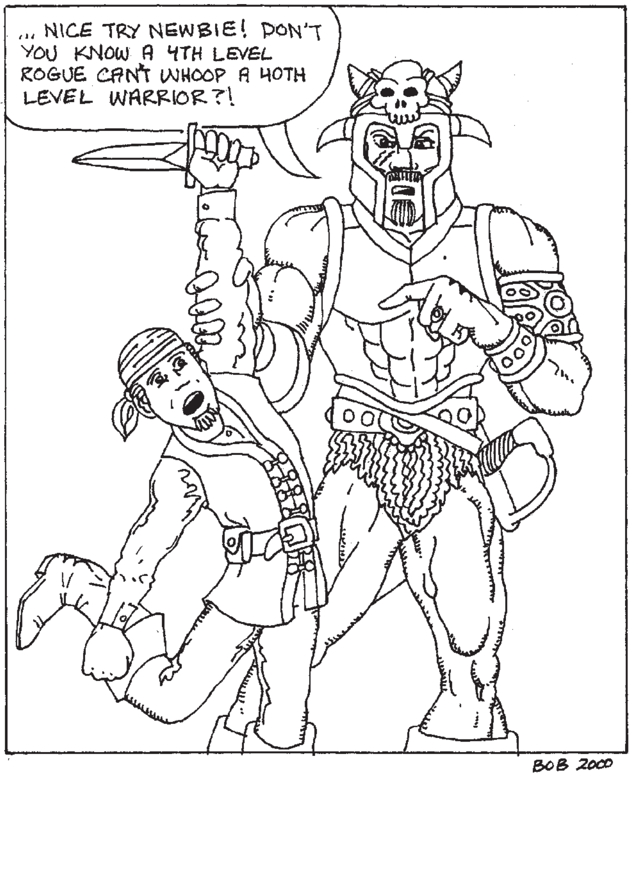

... NICE TRY NEWBIE! DON'T YOU KNOW A 4TH LEVEL ROGUE CAN'T WHOOP A 40TH LEVEL WARRIOR?!
BOB 2000

Figure 1.2

Can you spot the newbie in this picture?

Table 1.3 Experience Points for Levels

Level	Experience Points
1st	0–1,500
2nd	1,501–3,000
3rd	3,001–10,000
4th	10,001–25,000
5th	25,001–75,000

As you can see by looking at Table 1.3, it gets harder to advance in levels as your character rises through the ranks. This keeps people from rocketing to the top of the food chain in a short amount of time. With an advancement curve, you also keep new players from feeling too far out of reach of the high players. Now, here is the good part for you accountants out there: Because it gets difficult for players to advance at later stages, they have to spend vast amounts of time playing the game. Because they have to spend vast amounts of time playing the game, they also have to spend money monthly to maintain their subscriptions. The natural drive of competition is generally enough to keep most players coming back for more. Because other players are constantly online, they are gaining experience points and levels. And because they are gaining levels, they are either catching up to you or passing you. We can't let that happen, now can we?

Many people might argue with me about the significance of competition. Some players out there couldn't care less about competition, but believe me, they are the minority.

Ownership

Ah, the concept of ownership is my personal favorite. Ownership is what I have seen keep the majority of online gamers coming back for more. In many multiplayer online games, such as MUDs, players acquire possessions as they play. Take, for instance, the magic-wielding character example; he probably gains all sorts of arcane magic in his travels. The fighter character might find special weapons and unique armor. All of these items are completely virtual, but to the player, they represent a time investment. And you know what they always say, time is money. So as players get better at these games, they acquire better items. As the players acquire better items, they normally hunger for even better goods.

Take, for instance, the following scenario involving the fighter-character example. Suppose that at level one, he starts with a short sword. While adventuring around, he gets to level five and comes across a long sword. The new long sword does a lot more damage, so now he can fight some tougher creatures. Now that he can fight tougher creatures, he gets experience points faster and moves up in levels. The fighter is now level ten, but the creatures he encounters are very tough, so he wishes for a better weapon. He finds then a magic long sword and can dispatch his current foes with ease. This scenario continues to pan out until the player reaches the limits of the game he is playing. Most online game developers continually add content to keep players from ever reaching the top. One of life's little lessons is illustrated in Figure 1.3.

Okay, here is how this turns into long-term commitment for a game player. For players to get very high-level characters in most of these games, they must invest

Figure 1.3

Greed, the root of all evil

days of their time and effort. That's right—days, weeks, and even months of time. Believe it or not, many people play these games for years. Heck, I've had an *Ultima Online* account for more than three years! Anyway, once players acquire tons of items and power, they are usually reluctant to give it all up. This is how ownership keeps them coming back for more. I also refer to it as the *investment factor*. Either way, the bottom line is that people don't like to give up something for which they have worked hard, virtual or not.

Society

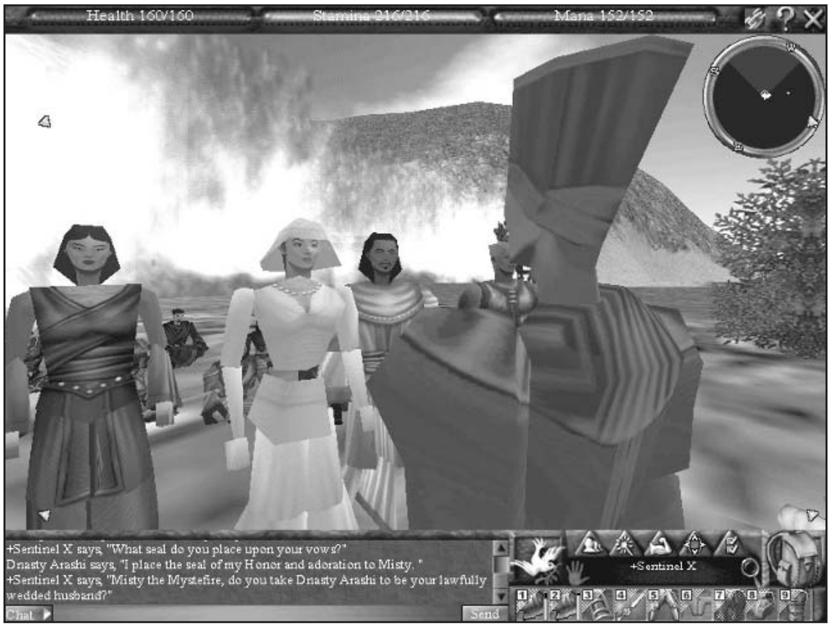
Society, the last hook I'll discuss, is one of the most complicated. You probably could write a short book on the subject. The easiest way I can explain it is that people are social creatures, so they like to socialize even when gaming. Most people who play MUDs, or any other persistent-world game, spend large amounts of time meeting other gamers online. I know that when I first beta-tested *EverQuest*, I spent way too much time socializing and never got a high enough level to test out many areas. Of course, I didn't know what the heck I was doing either, so that might have kept my levels down, too!

If you've ever played a persistent online adventure game, you have probably heard of online weddings. In fact, I was playing *Asheron's Call* the other day, and our guild monarch had a virtual wedding online. So you see, people can form strong friendships and ties in online games. This is why I mention society as a hook. Many people keep playing these games just to keep friendships going strong. As a prime example of in-game friendships, Figure 1.4 shows a virtual wedding I attended in *Asheron's Call*.

That's about it for MUDs and the like. As I said before, many of these old university games are still around, and a few of them even have Java interfaces for the Internet!

Let There Be Modems!

This is the part where I reminisce about my very first 300 baud modem. Some time back in the 1800s, modems became popular on personal computers. Okay, that might not be entirely true, but it seems like a long time ago. Anyway, modems started coming out for PCs, and people bought them up like hotcakes. The first popular modems were insanely slow by today's standards, but they were a blast to play with. Besides chatting with other people and sharing files, you could also play games with them. The games you could play weren't like *EverQuest*, but they were fun in their own right.



+Sentinel X says, "What seal do you place upon your vows?"
Dnasty Arashi says, "I place the seal of my Honor and adoration to Misty. "
+Sentinel X says, "Misty the Mystefire, do you take Dnasty Arashi to be your lawfully wedded husband?"

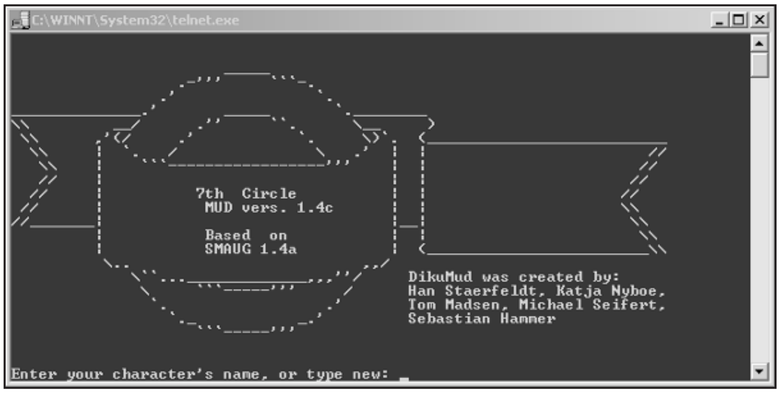
Figure 1.4

Dnasty Arashi's wedding in Asheron's Call

BBS Games

The games you could play on these early modems took place on BBSs (*Bulletin Board Systems*). Although the Internet has all but wiped them off the face of the Earth, BBSs are still around today. The games you can play on them are similar to those played on university networks. All of the games are text-based and usually consist of simple commands and rule sets.

One of my all-time personal favorite BBS games was called *Empire*. In *Empire*, you were allowed to play only so many turns per day. On most of the systems where I played this game, you could only make one turn per day. The basic objective of the game was to build your empire through a larger population base and then use that population base to amass a huge army. Once you had a large army, you could defeat your neighbors and one day be supreme leader of the game. Most of the games lasted about a month, and several ran simultaneously. Playing the game never had a long-term goal; you usually just got your name somewhere on the BBS as the winner. I poked around the Internet and came across an example BBS game called *7th Circle*. Figure 1.5 shows the *7th Circle* screen.



```
7th  Circle
 MUD vers. 1.4c

 Based  on
 SMAUG 1.4a

                    DikuMud was created by:
                    Han Staerfeldt, Katja Nyboe,
                    Tom Madsen, Michael Seifert,
                    Sebastian Hammer

Enter your character's name, or type new: _
```

Figure 1.5

Output from the 7th Circle *BBS game*

So what made BBS games so much fun to play? Well, for one thing, most of the games had the same hooks I mentioned in MUDs. Your empire would expand and get better with play, you had to invest time to beat it, and you were able to socialize with your opponents. To better illustrate this, take a look at a common set of turns in *Empire*.

1. Join a new game and start with 200 gold, 200 citizens, and 500 food.
2. Buy 50 farms for 100 gold, leaving you with 100 gold.
3. Buy 50 houses for 50 gold, leaving you with 50 gold.
4. End turn.
5. Your farms generate 50 food, leaving you with 550 food.
6. 50 people move into the 50 new houses; you now have 250 citizens.
7. The new population of 250 eats 250 food, leaving you with 300 food.
8. Your population generates one gold piece for each citizen, which nets you 250 gold. You now have 300 gold in your coffers.
9. At this point, your farms cannot sustain your population, so you buy another 150 farms for 300 gold. This leaves you with 0 gold and 200 farms.
10. End turn.
11. Your 200 farms generate 200 food, leaving you with 500 food.
12. The population of 250 eats 250 food, leaving you with 250 food.
13. Your population generates one gold piece for each citizen, which nets you 250 gold. You now have 250 gold in your coffers.
14. Your farms still cannot sustain the population, so you buy another 100 farms for 200 gold. This leaves you with 50 gold and 300 farms.

15. End turn.
16. Your 300 farms generate 300 food, leaving you with 550 food.
17. The population of 250 eats 250 food, leaving you with 300 food.
18. Your population generates one gold piece for each citizen, which nets you 250 gold. You now have 300 gold in your coffers.
19. The farms now sustain your population and even leave you with some extra at the end. Now it's time to save for an army.
20. End turn.

Even from this little example, you might start to get the idea of why the games were, and still are, a lot of fun. They have no spectacular graphics or sounds, but the strategy involved in winning the games can be quite in-depth. The entire game of *Empire* wasn't completely based on numbers either; random events occurred for good measure. I remember that in one instance, I started a turn and a random event let rats eat most of my grain in storage. This caused many of my people to starve. As a result, I had to rethink my strategy in midstream and do some serious damage control. So you see, games can be fun without all of the hype and glamour. This leads me to something I have always tried to reiterate, "Games are about gameplay." Guess it doesn't take a genius to see that, eh?

Modem-to-Modem Games

Back in the late 1980s, a designer named Dani Bunten Berry created one of the first multiplayer games for the Commodore 64. *Modem Wars* could be played by two players connected via modems. A small publisher named Electronic Arts published the game. Of course, Electronic Arts is now a huge publishing company that owns many game-development houses. Anyway, I always think it's kind of interesting to see how quickly things evolve.

Modem Wars was similar in basic design to popular real-time strategy games of today. In the game, you basically selected units and then clicked where you wanted them to move. The units attacked enemies if they came within range along the way. I once read somewhere that Berry used four-byte packets to transmit all game data. The program used deltas to figure out each player's game state. One of the coolest aspects of the game was that everything happened in real time. You took no turns, just all-out fun. This game was significant because no other game of its type had been published up to this point. Everything was designed with solo play in mind. One of the main reasons for this was that modems weren't popular yet. However, by the time modems became popular, *Modem Wars* had been discontinued. It was a fun game that followed the KISS standard, otherwise known as "Keep it simple, stupid." Berry was a fierce believer in keeping games simple and fun. Most of her games followed this belief and proved to be popular successes.

Not too many years after games such as *Modem Wars* were released, other games started following suit. In most multiplayer games of the early 1990s, your only option was to connect via a direct modem-to-modem connection. In a direct modem-to-modem game, you basically either put your modem into accept mode or dial into your friend's modem, which is in accept mode. Because the modems had to connect to each other directly, only two computers could connect at once. Most of these games also gave you the ability to connect via a *null-modem cable*, which was nothing more than a special serial cable. To see how these types of games connect, refer to Figure 1.6.

You will find that DirectPlay and some other networking libraries still support serial cables and modem-to-modem connections. I personally do not bother with them because almost everyone uses TCP/IP via the Internet now. The main drawback to modem-to-modem connections is that they support only two players at a time. This setup is fine for certain types of games, but most games out today support more than two players.

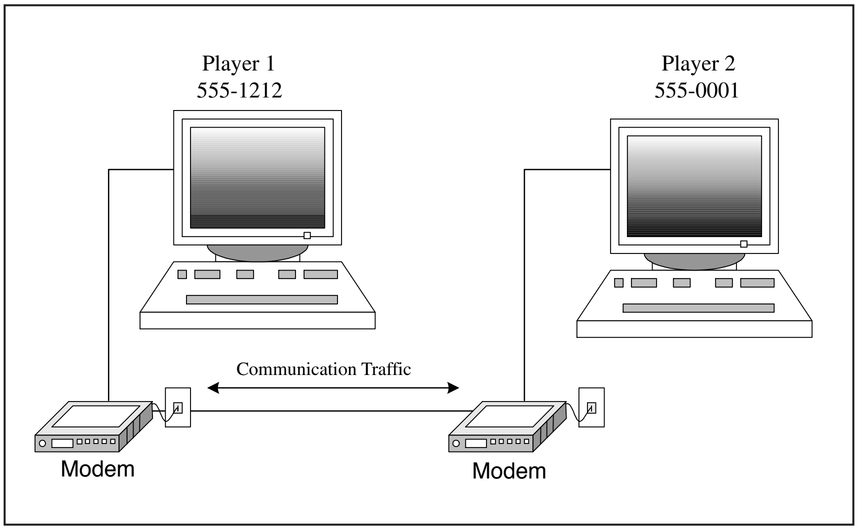


Figure 1.6
Modem-to-modem connection architecture

Real-Time Strategy Games

For a long time, relatively few multiplayer games on the scene worked with more than two players at a time. The game I remember changing all of this was *Command & Conquer* by Westwood. Now, *C&C* was not the first game out to offer four players online at the same time, but it was one of the most popular. It also happens to be one of my favorite games of all time. Games such as *Command & Conquer* used a communication protocol called IPX, which was mainly intended for communication across closed networks, or LANs. Because *C&C* did not have any built-in support for connecting to the Internet, you had to use special software. You can see an in-game scene of *Command & Conquer* in Figure 1.7.

Enter stage left, Kali Inc. Kali Inc. released a piece of software that allowed you to create and join games that used IPX across the Internet. You connected to the Internet with your modem and then launched the Kali software. Once the software was loaded, you could then set a game to play with it. Kali handled all of the connecting of players and then launched the game when ready. The software was really great to use at the time and allowed distant friends to play together. By today's standards, though, it was a pain in the rear to use. The main problems stemmed from

Figure 1.7

Command & Conquer (©1999 Electronic Arts. All rights reserved.)

the fact that it was difficult to find others who used Kali. You could find other players, but you never knew with whom you were dealing. I can't begin to count the number of games I started only to have the opponent drop the connection within a few minutes.

Command & Conquer

Anyway, enough about the difficulties of IPX; let's move on. *Command & Conquer* is a great game in which you build an infrastructure to support an army to destroy your enemies. If you ever played *Starcraft*, *Zeus*, *Warcraft*, or *Total Annihilation*, you should have a good idea of what I am trying to describe. The goal of *C&C* is to destroy all other players who are not on your team.

To destroy other players, you must build an infrastructure. How do you do this? Well, in *C&C*, you harvest a mineral called tiberium. Tiberium only appears around special tiberium-generating columns. You start the game with an MCV (*Mobile Construction Vehicle*). You must deploy the MCV to work, so the best idea is to move it to a clearing and then deploy it. Once you deploy the MCV, it becomes your base of operations and lets you build additional buildings. The first order of business is to build a refinery that releases a harvester. The harvester proceeds to seek out tiberium to harvest. Once the tiberium is harvested, you bring it back to refine. You can think of tiberium as money. Everything you build in *C&C* costs tiberium.

The game proceeds in this way, building more buildings and such, until you raise an army. Once armies are in place, the players can fight it out and try to destroy each other's bases. The game is over for a player when his last MCV is destroyed.

The Appeal of RTS Games

As with any other game type, RTS (*Real-Time Strategy*) games do not appeal to everyone. But they have been a dominant force in the game industry and deserve notice. At the heart of any successful RTS game are three basic elements: an intuitive interface, good AI, and good network support. Take a brief look at each of these aspects.

The Interface: Man's Last Great Hope

Okay, so maybe interfaces have nothing to do with mankind, but they do make or break games. Almost every RTS game I have seen that failed in the sales category did so due to a lousy interface. Players don't want to have to read a manual to play your game. If a game can't be figured out without a manual, then it is no longer a game; it's a test. And I can't think of many people who like to pay to play a test.

When designing an interface, let some of your friends try it. Don't give them any hints on how to play the game; just see how they do without guidance. If your testers can't figure it out, then you need to do a redesign.

TIP

A good strategy to use is to record the number of mouse clicks generated during the course of your game. After you have the number of clicks generated, see what you can do to reduce them. The less a player has to click on the mouse, the better.

Artificial Intelligence

Some games can get away with bad AI (*Artificial Intelligence*). Others cannot, and believe me; RTS games cannot get away with bad AI. Some RTS games use scripted AI. When AI is scripted, the computer-controlled units follow a set script, regardless of what the player does. This works as long as players do what you expect them to do. But guess what? Players never do what you expect them to do.

Some bad RTS games use just plain stupid AI. Either the enemy units get bunched up on obstacles, or they get lost. Poor path finding will doom a real-time strategy game. Bad path finding makes the enemy units do stupid things, and the player-controlled units follow the same behavior. If you have ever played a game where your units couldn't figure out how to get somewhere, then you know the frustration. Players will not put up with having to baby-sit their armies when they are walking around the obstacles.

Good Network Support, Duh!

Not many people like to play games that have bad multiplayer support. And I sure hope you don't like them much either! Maybe I don't need to talk much about this concept because you are reading this book, but some RTS games have poor network support. In fact, one of the most anticipated games, *Tiberian Sun,* came out with IPX. IPX was fine back when the first *Command & Conquer* game arrived, but TCP/IP has since replaced it. Not only did *TS* utilize IPX, but you also had to register and play through Westwood's game service. You can't just connect directly with a friend and play the game; Westwood makes you connect through its service to reduce game piracy. This is a fine concept if utilized properly, but the game becomes extremely difficult to get up and running. The main problem is that some people lose the jewel cases in which the game comes. Without the code on the jewel case, you cannot play on the servers. I can understand how this process is required for something such as *EverQuest*, where your characters are stored online, but not for an RTS.

Even though these three elements make up a good RTS game, they are also important in most other games. So, keep them in mind for your next game project. Okay, I'm parched; time for a Mountain Dew . . . the lifeblood of any good programmer.

Massively Multiplayer Online Role-Playing Games

Wow, that's a mouthful, eh? MMORPGs (*Massively Multiplayer Online Role-Playing Games*) came onto the game scene not too long ago. Basically, they are graphical MUDs with much more functionality. Just like MUDs, MMORPGs are persistent. Some of the more common games of this type out today are *EverQuest*, *Asheron's Call*, and *Ultima Online*. The main difference between MUDs and MMORPGs is that MMORPGs utilize pay-for-play models. Take, for instance, *Ultima Online*, where you must pay $10 a month to play. If you don't pay, you can't play. Take a peek at one of the first games of this type.

Meridian 59

The first MMORPG I ever played was called *Meridian 59*. It was created and maintained by 3DO. It was based in a medieval setting, much like other games of its type. I don't know why so many games are based on this time period, but they are definitely the most prevalent. Anyway, *M59* cost $10 a month for unlimited usage. You could operate up to five characters on a single account.

You could mold characters in *M59* however you wanted because it utilized a skill system. You didn't have to pick a specific character class; you just chose your skills and stats. The game used a decent magic system that contained about five schools of magic. One of the magic schools taught offensive spells; another school taught healing and defensive spells. With many schools of magic available, the game kept players from getting good at all the spells.

As with most other persistent-world games, *M59* had no particular goals. Most people just played to raise their characters in level. The game had some quests, but not much beyond killing the next big creature. *Meridian 59* did have one cool feature I wish were present in other games like it, though. Once every so often in the game, the world entered a sort of chaos mode. During this time, you could attack and fight any other character you wanted with impunity. Once the chaos period ended, the server reverted back to the state it had before the event. This gave players a chance to try their skills on other characters. Because nothing could be lost or gained, it was a fun time. One time during a chaos period, I actually attacked a game counselor. Of course, he blasted me to the netherworld, but it was a lot of fun.

Even though *Meridian 59* had a 3D user interface, its graphics were extremely basic. The game did not look or even sound good, but it had the right elements of a persistent-world game to keep you hooked. And that's what really matters if you are looking to make a game for profit. By the way, you might have noticed that I keep talking about *Meridian 59* in the past tense. Well, 3DO quit supporting the game servers a while back. I have heard rumors that a German company is going to reopen it, but I'm not holding my breath.

Ultima Online

Not long after I cancelled my *Meridian 59* account, another MMORPG game called *Ultima Online* went into beta. *Ultima Online*, or *UO*, as most people call it, is the online version of Origin's Ultima series. For those of you not familiar with Origin, it was founded by Richard Garriot, also known as Lord British. Lord British released *Ultima I* about a million years ago. It was a top-down tile-based game that had no multiplayer support. Many more Ultimas were released with great success, which put Origin on the map.

Great Expectations

Around 1997, Origin decided to release an online version of its successful Ultima series. It was to be the greatest multiplayer experiment ever conducted. Not only would it allow thousands of players to play at once, but it would also be a huge evolving world.

Ultima Online does allow a few thousand players to be online in the world simultaneously. The other great feat *UO* pulled off is that the world is seamless. In other games, such as *Meridian 59* and *EverQuest*, the worlds consist of distinct zones. In *UO*, most players never even realize when they cross server boundaries. When the game first came out, the *lag* was horrible. You couldn't move more than a few squares without pausing. The good thing is that it didn't take long for Origin to fix most of the lag problems. *UO* now has what is probably the best network code around. The game rarely crashes, and even when dozens of people occupy the same area, the frame rate does not drop.

Lag **refers to the delay between your computer and the computer to which you are connecting. If you have ever played an online game and noticed delays or pauses, you have experienced lag.**

Before *UO* was actually released, Origin promised that the world would have tons of interaction. For instance, it said that if players killed off all of a dragon's food supply, the dragon might extend its range and attack a human village for food. Now, this might not sound like much, but basically, it discussed an entire ecosystem within the game. Just ask the Biosphere people how hard it is to maintain an ecosystem! Although Origin had good intentions, no such functionality ever made it into the final game. In fact, nothing greater than random creature spawns made it in.

Before I discuss the pros and cons of the game, take a look at how you play it.

Creating a UO Character

After you sign up for an account, the first thing you do in *Ultima Online* is select a server on which to play. Because the game world can only support about 3,000 players simultaneously, a special arrangement had to be made to let the rest of the people play. All Origin did was duplicate the world on multiple servers. That way, 3,000 people times *n* number of servers can play at once. The only downside is that a character created and played on one server cannot transfer to another server. You can think of the different servers as parallel worlds that never meet. Here is a brief list of some of the servers available.

- Lake Superior
- Great Lakes
- Atlantic
- Baja
- Sonoma
- Europa
- Oceania
- Yamato
- Izumo

You might have noticed that the last two servers have Japanese-sounding names. Well, *UO* is so popular that Origin opened some servers just for Japan. In fact, *UO* has servers available to players all over the world. Oh, yeah, I almost forgot; servers are called *shards* in *UO*. Take a look at the select-a-shard screen for *UO* in Figure 1.8.

The number of shards should give you an idea of the scale of the game. After you select which shard you want to play in, you get to create a new character or load an existing one. The character creation screen lets you choose an adventurer, a merchant, or an advanced character (see Figure 1.9). For the purposes of this book, I chose the adventurer class of character.

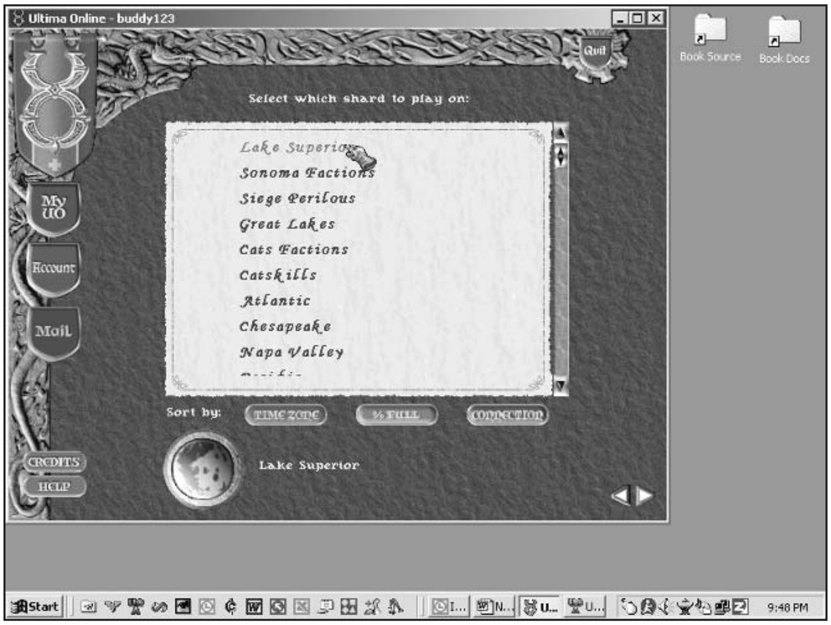


Figure 1.8

The shard selection screen in Ultima Online (©2000 Electronic Arts. All rights reserved.)

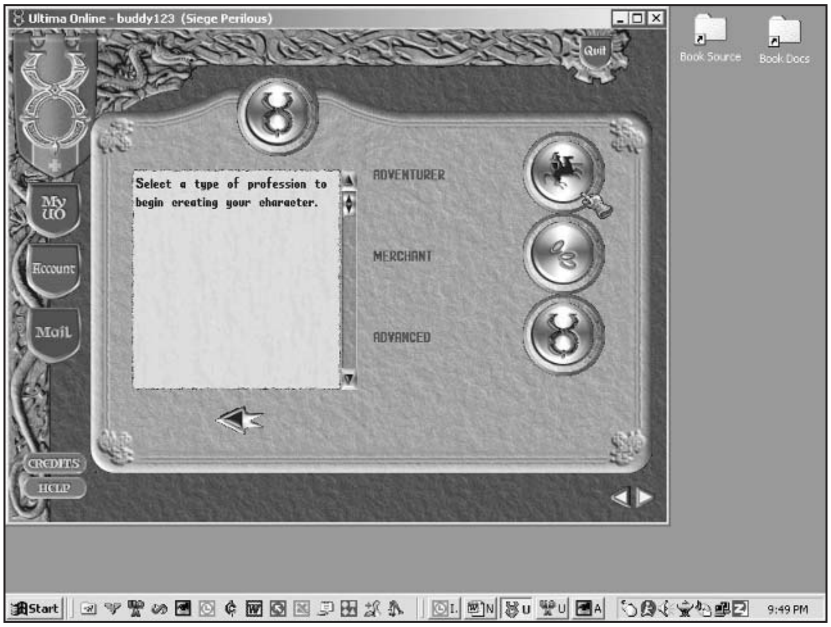

Ultima Online - buddy123 (Siege Perilous)
Quit
Select a type of profession to begin creating your character.
ADVENTURER
MERCHANT
ADVANCED
My UO
Account
Mail
CREDITS
HELP
Book Source
Book Docs
Start
9:49 PM

Figure 1.9

The character creation screen (©2000 Electronic Arts. All rights reserved.)

After you choose your character's main class, you then see a screen requesting a subclass. For the adventurer class, the following subclasses are available.

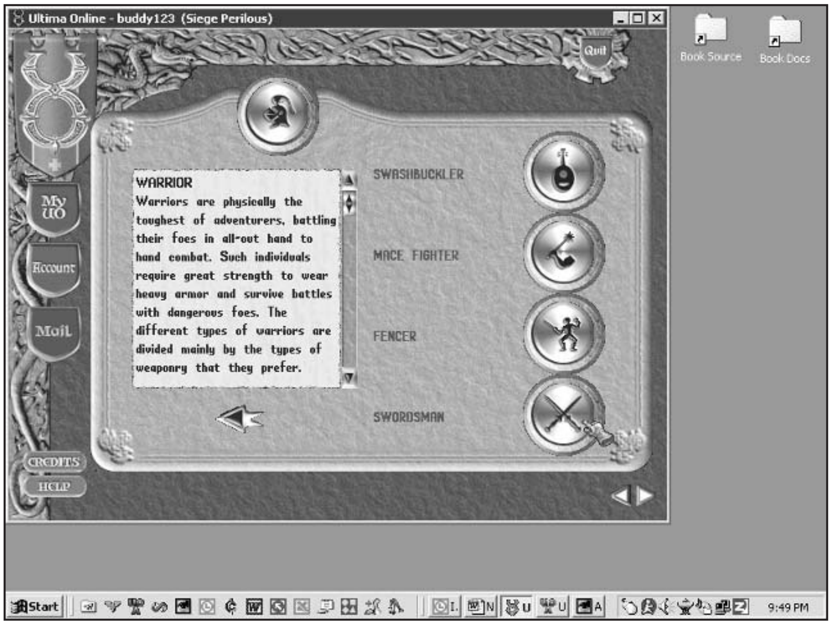
- Archer
- Magician
- Warrior

Because fighter types are usually the easiest to operate in these games, I'll choose the warrior class.

The next screen you see lets you pick your warrior's specialization. The following four specializations are available (see Figure 1.10).

- Swashbuckler
- Mace Fighter
- Fencer
- Swordsman

Swordsmen are great because they get to wear super-heavy armor and wield the mightiest of weapons. So, I'll choose swordsman as my specialty.



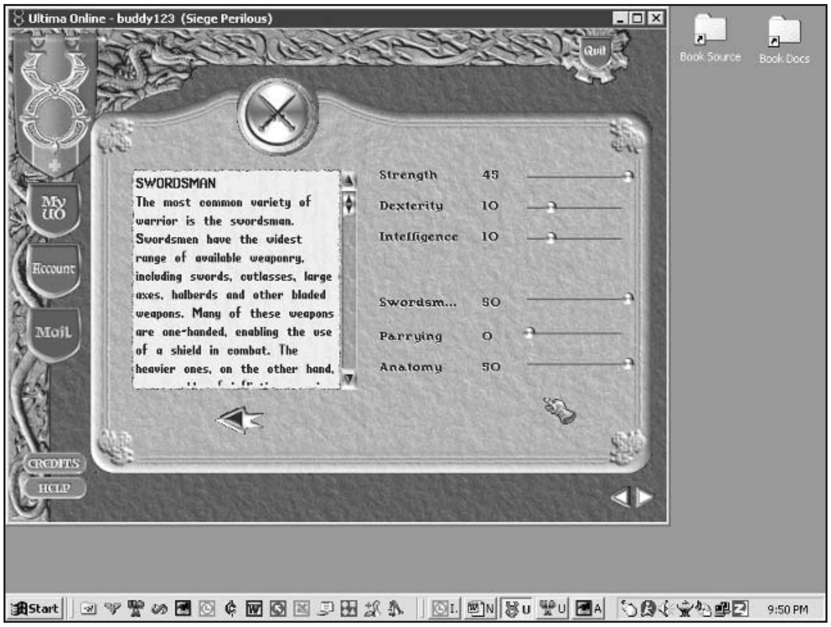
WARRIOR
Warriors are physically the toughest of adventurers, battling their foes in all-out hand to hand combat. Such individuals require great strength to wear heavy armor and survive battles with dangerous foes. The different types of warriors are divided mainly by the types of weaponry that they prefer.

Figure 1.10
The specialization screen
(©2000 Electronic Arts. All rights reserved.)

All right, now that you have a character type, you'll see a screen to pick your stat and skill points. As in the MUD games, you have a certain number of points to allocate for stats and skills. When you move a slider to advance a stat, the other stats decrease. This functionality lets you fine-tune your character to your liking. Refer to Figure 1.11 to see the point allotment screen.

Okay, now that the character is defined, it is time to choose what he looks like. I know that this might sound a little strange, but when you are playing a game with thousands of others, it helps to stand out. In the old days of single-player games, your character could look like whatever the designers wanted and you never got lost in a crowd. Why? Well, there was no crowd in which to get lost! *UO* lets you choose your hairstyle, skin color, gender, and general appearance. A sample screen appears in Figure 1.12.

Once you design your character's look, you get to pick your starting town. Of the many towns in *Ultima Online*, any of the available ones are fine for a newbie. Because the town of Britannia is the default, I chose it. Once you select a starting town, you enter into the game with your brand new character. I got a pleasant surprise when I created the character for this chapter; a player wanting to help me greeted me. She even asked whether I needed anything to get me started in the



SWORDSMAN
The most common variety of warrior is the swordsman. Swordsmen have the widest range of available weaponry, including swords, cutlasses, large axes, halberds and other bladed weapons. Many of these weapons are one-handed, enabling the use of a shield in combat. The heavier ones, on the other hand,

Strength	45
Dexterity	10
Intelligence	10
Swordsm...	50
Parrying	0
Anatomy	50

Figure 1.11

The skill and stat point adjustment screen

(©2000 Electronic Arts. All rights reserved.)

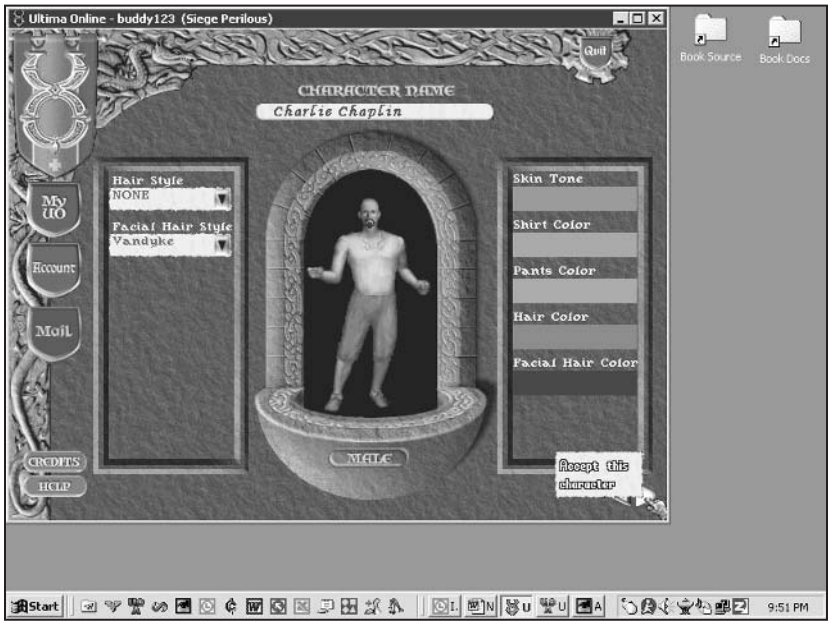


Figure 1.12

Designing your character's appearance (©2000 Electronic Arts. All rights reserved.)

world. (But alas, I declined.) If you want to see what the game looked like to the new swordsman, take a look at Figure 1.13.

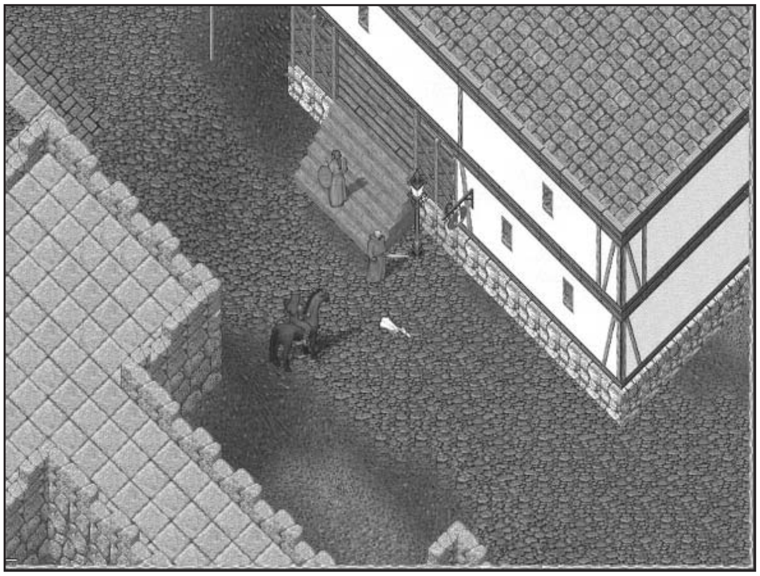


Figure 1.13

Entering the world for the first time in UO (©2000 Electronic Arts. All rights reserved.)

Gameplay in Ultima Online

Well, now that I have walked through how you get started in the game, take a minute to learn about gameplay. Because *UO* is a skill-based game, your character only gets better at what he does by actually using his skills. So, the swordsman only gets better at using his weapon if he goes out and fights enemies. In this particular game, it is as easy as walking up to a creature, entering attack mode, and then selecting the creature to attack. Your character continues to swing his sword until the creature is dead, the character is dead, or you decide to exit combat mode. For every successful hit on the creature, your skills have a chance to increase. *Ultima Online* does use a difficulty scale, though. As you get better with your sword, you have to fight more difficult creatures to get even better. This keeps a player from becoming a grandmaster swordsman by just killing rabbits. Refer to Figure 1.14 to see the curve concept. In the chart, skills cost a greater number of experience points each time the skill level goes up.

Skills

To give players a sense of accomplishment, *UO* grants titles to characters based on their best skills. For instance, a beginning swordsman might have the title of apprentice, but an accomplished swordsman can have the title of master. The highest level possible is grandmaster, which represents a skill of 100 in something. To keep players from becoming masters of everything, the game limits any one character to 700 total skill points. This keeps any character from becoming grandmaster in more than seven skills at once. Most characters in the game aren't grandmasters

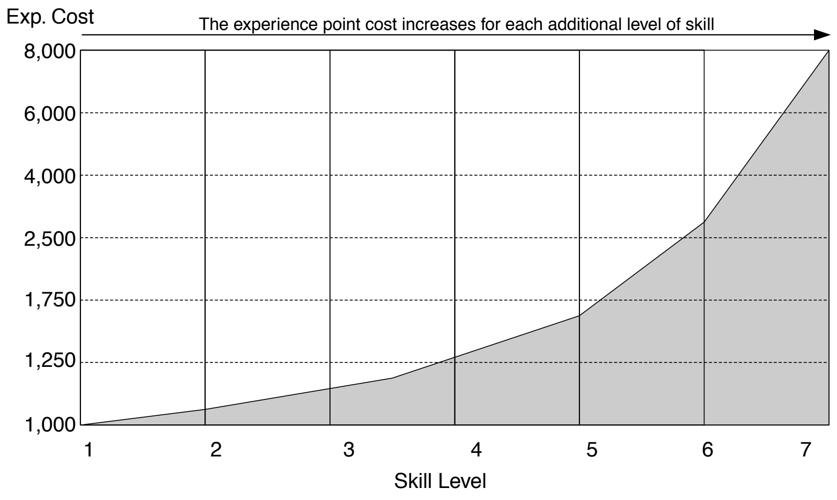


Figure 1.14

Skill difficulty versus gain

in seven skills; they usually are just really good in a couple of skills and mediocre in many others. There really is not much else to character development than getting better at whatever skills you see fit. Even though this might not sound like much, many players spend months getting their characters built up.

Trade Skills

You might be asking, what makes *UO* so special that it has survived for more than three years? Well, *UO* is unique in that it offers many more ways to play with a character than just killing creatures. The game engine has included what players commonly refer to as trade skills. An example of a trade skill is blacksmithing. Blacksmithing allows players to create weapons and armor in the game.

The following scenario in *UO* demonstrates the complexity involved in this. Imagine that you created a character with blacksmithing and mining skills, called Miner49er. For Miner49er to create some armor, he needs some metal ore with which to work. So you take Miner49er to a nearby mountain chain and begin using your mining skill to pick away at the rocks. Every once in a while, you get lucky and obtain some metal ore from the mountain. Once you have enough metal ore, you head back to the forge in town. At the forge, Miner49er once again uses his mining skill, this time to smelt the ore into ingots. Now that he has some workable ingots, Miner49er uses his blacksmithing skill to hammer the ingots into some armor. If the whole process works, you end up with some new armor. If something goes wrong, you must start over. To review, take another look at the steps involved in making some armor in the game.

1. Utilize your mining skill by using a pickax on a mountain chain.
2. Keep mining for ore until the ore is obtained from the mountain.
3. Once enough the ore has been mined, smelt it into ingots at a forge.
4. Utilize the blacksmithing skill by using a smith's hammer on the ingots.
5. If the blacksmithing skill succeeds, you end up with brand new armor.

That's a pretty in-depth process for making armor. Many factors are involved. If your mining skill is too low, it might take a long time to get the ingots required to make any armor. If your blacksmithing skill is too low, you probably won't be able to make the armor you want. These types of intricacies keep players in *UO* coming back for more.

Housing

Another special aspect of *Ultima Online* is home ownership. Your characters can actually own homes in the game world. Some homes are not very expensive, running about 40,000 gold, but some are very expensive, running around 500,000 gold. The main thing that keeps everyone in the game from owning a home is real estate. Because the game world is only so large, it contains only so many places to build a house. Most places that can have a home built on them are already taken. This capacity has led to a huge sellers' market in the game. Many players who own homes in the game can even sell them for real money on auction sites such as eBay. At this time, I have one account with a full-sized tower and another account with a small house. I could easily sell the tower on eBay. I have always found it strange that people actually pay for virtual items, but hey, stranger things have happened!

Houses are important to players in the game because not only can you decorate them with cool items, but you can also store your belongings in them. The larger the house, the more items you are allowed to store, which explains the appeal for larger homes. In Figure 1.15, you can see my bard character standing on top of my large tower.

Player Killing

So far, the game seems pretty harmless, with players fighting creatures, using their trade skills, and vying for home ownership. Well, all of this comes to a crashing halt when you witness player killing. Player killing, more commonly known as PKing, is the best part of *UO* to some players and the worst part of *UO* to others.

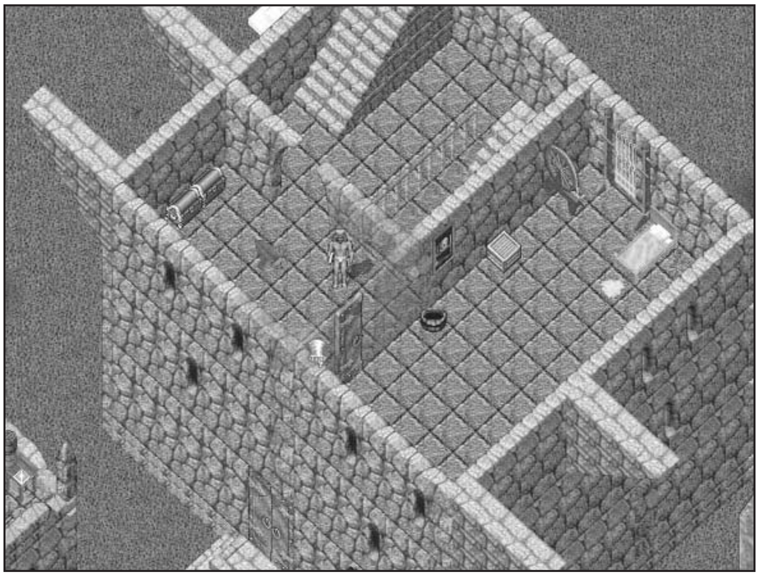


Figure 1.15

A large tower house in Ultima Online (©2000 Electronic Arts. All rights reserved.)

In the early days of *Ultima Online*, few rules concerned the killing of other players. You could attack, kill, and then loot at will. Of course, when you die in the game, you aren't out of it for good. You merely run to a healer to get resurrected. This might not seem like a big penalty, but more than likely, you lose all of the possessions you were carrying, which is a big pain in the rear if you happen to be carrying something prized.

The problem *UO* had with PKs, or player killers, was that they were everywhere. You couldn't walk 20 meters outside of a safe zone before being attacked by a PK. This is fine and good for players who want to fight one another, but it is not good for players wanting to adventure in safety. *UO* players have debated rampant PKing for a long time. The PKs want to be able to fight other players, but the non-PKs want to be left alone. Ultimately, no pun intended, Origin created a duplicate world for all of the shards in which player killing is not allowed, so players who want to avoid PKs can play in absolute safety. Most PKs affectionately refer to the new safe lands as care-bear lands.

The unfortunate side effect to having such a violent world is that it tends to bring out the worst in players. *UO* has more rude and annoying players than any other online game I have ever played. I mostly attribute this to the competitive nature of the game. So far, I have been able to narrow down my theory to one simple rule:

> The more gain that players can get from others' misfortunes, the more anti-social behavior will be present.

This might not sound exactly right at first, but all you have to do is examine other games and the amount of anti-social behavior in them. Take, for example, the game *Asheron's Call*. In the non-PK worlds of *AC*, people are generally very nice to one another, and I have yet to witness any real jerks playing the game. I attribute this to the fact that there is no way to gain from another's detriment in the game. You can't kill another player, you can't steal from another player, and you can't really do much of anything to affect another player in a negative way. So with no gain to be had from hurting another player, why do it? People in *AC* can only gain from helping out others. There is no incentive to be anti-social. Now, go and play *AC* on one of the PK servers, and you will see immediate anti-social behavior. Not only will you probably be attacked immediately, but you will also find that people lie to you and try to trick you into all sorts of things. It is like night and day between the two types of servers. Luckily, AC gives you the choice: You can play on the normal servers with nice people, or you can play on the PK server with seemingly rude players. Understand that the PKs are not really rude or jerks in real life; they are just playing to compete with and beat you. That is fine if that is your cup of tea.

Anyway, the approach I took to *UO* goes along with the old adage, "If you can't beat them, join them." All my guild really ever did in *Ultima Online* was figure out how to PK other players. We weren't content with fighting characters in normal ways, though; we always had to think up some new clever way of doing it. So go grab a cold one, sit back, relax, and read about some of my guild's little adventures in PKing.

Exploding Chests, Oh My!

One of the skills we enjoyed the most in the game was the tinkering skill. The tinkering skill mainly allowed you to create such boring things as clocks and keys, but it also allowed you to set traps on treasure chests. The traps could either explode or set off a poison dart. We preferred the exploding ones.

Anyway, our favorite thing was playing off other players' greed. Hmm, maybe there is a moral in here somewhere, but I doubt it. What we did was load a treasure chest with all sorts of garbage items and then put a trap on it. We then set the chest on the ground beside some busy road. Inevitably, some doofus would come along, see the chest just sitting there, give into greed, and open it. *BOOM!* The chest would explode, thereby killing the curious player. After he died, we would loot his body. To avoid too much suspicion, we then moved our little chest down the road and reset the trap on it. You would be amazed how many times we pulled this off in a small area. I guess it is true about curiosity killing the cat. You can see the results from one of our adventures in Figure 1.16.

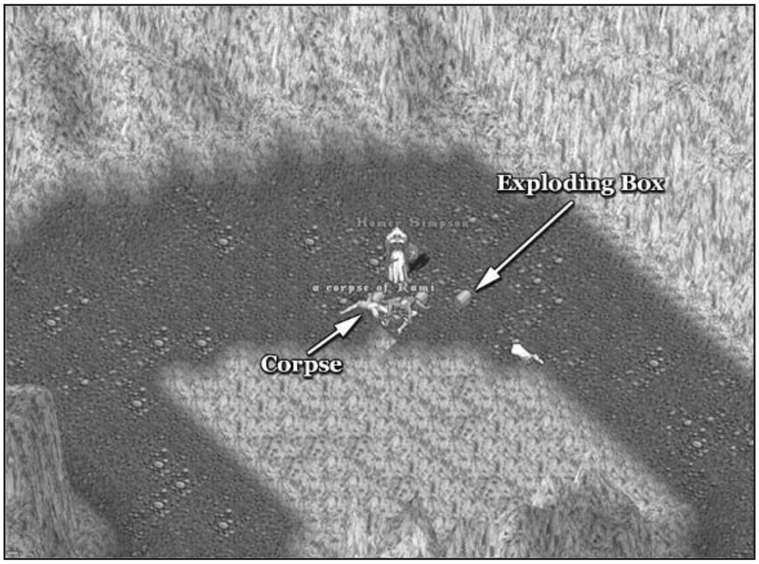


Figure 1.16

The exploding-box trick at work

(©2000 Electronic Arts. All rights reserved.)

I will admit that this wasn't the most ingenious or clever way to PK people, but it really was funny to watch. In fact, it led us to another method that proved to be even more entertaining.

Guards, Come Get This Scum!

Before you can fully understand this PK method, you must first understand how safe zones work in *Ultima Online*. In *UO*, towns are guarded by computer-controlled guards. When you do something illegal in a town, guards can be called on you. The guards in *UO* are unforgiving, kind of like the ones at some southern prisons, and they kill you for any illegal action. This deterrent keeps most sane players from doing anything illegal in town. All that has to happen is for someone to yell the word "Guards!" and you are as dead as unagi at a sushi bar. But once again, greed often overrules common sense.

For us to do our trick, we had a few things in place, including

- A newbie character wearing nice-looking armor
- A bombed chest being carried by the newbie character
- A hidden accomplice

As you can see, we didn't need many components to pull off this particular trick. We created the last item, a hidden accomplice, with the hiding skill. The hiding skill in *UO* enables your character to literally disappear off the screen when standing still. This is a great skill to have and a must for any serious PK. Okay, on with the trick.

The first thing we did was to scout out a location beside a busy road leading out of town. We made sure the location was still within the guarded zone of the town, but almost outside of it. The accomplice then hid beside the road. Our newbie character, or bait, as we liked to call him, stood in the middle of the road, looking well off with his bright shiny armor. As soon as a player walked by, our newbie character said something like, "Oh, wow, I wonder what this is?" and opened the bombed chest. When he opened it, the bombed chest exploded, thereby killing our newbie character. This left the player all alone with the dead bait on the ground. Of course, the passerby was overcome with desire to loot the body of all the shiny armor. Because he was almost out of the guard zone, the passerby figured he could loot the body and run for the border, so to speak. Well, the second he looted a single item off the body, the accomplice came out of hiding and yelled, "Guards!" Immediately, the computer-controlled town guards popped in and killed the offender. Naturally, the accomplice just went over and looted the newly-dead greedy player. Ta da!

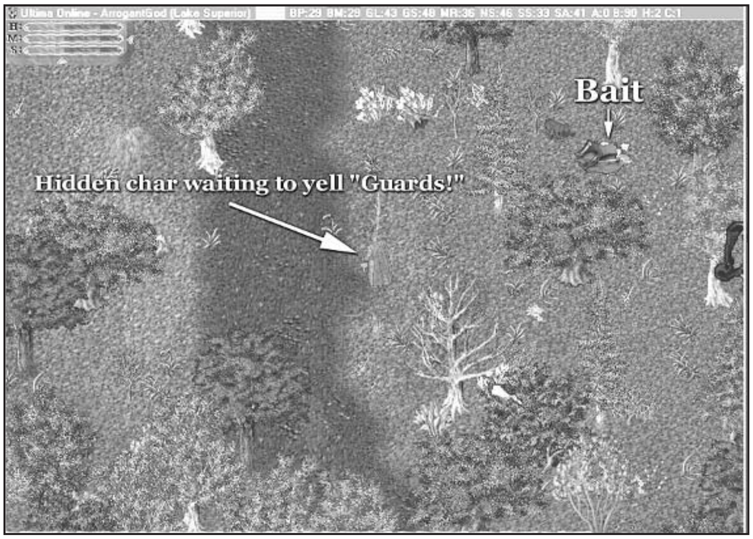


Figure 1.17

The dead-body trick (©2000 Electronic Arts. All rights reserved.)

That's it; we killed more than a hundred greedy players this way. Check out this plan at work in Figure 1.17.

I guess in most states this could be called entrapment, but I have seen worse on *Cops*. Anyway, most of this story might not seem that funny if you haven't played *Ultima Online*. But at the very least, it should give you an idea of the variety of things possible in the game.

What Makes Ultima Online a Great Game?

Even though *UO* has some flaws with player killers and such, it really has some shining attributes. First of all, you can customize your character's appearance to your heart's content. This feature might not sound that exciting, but I have seen players spend hours playing around with different colors and styles of clothing! It really does give players a sense of personality within the game.

Second, *Ultima Online* has the greatest sense of ownership I have ever seen in a game. The mere fact that you can own a house in the game keeps many players active. No other game comes close to keeping players hooked to this degree.

And last, *UO* has some of the best network code around. Sure, sometimes lag is bad, but the network code really is a technological marvel.

I can say other good things about the game, but if you want to find out, go play it for yourself. It's worth a short peek.

Ultima Online, the Ugly Side

All right, with all of these good things to say about *UO*, what are the bad things? Well, *Ultima Online* suffers from the worst case of stagnation I have ever seen in an online game. Heck, even the game *Subspace* was updated more often, and it was never really sold! I guess that the inherent design of *UO* has made it difficult for the designers to add new content. Only once in three years have they added completely new creatures, and even then they made you pay for it. With other online games updating monthly, *UO* just doesn't cut it anymore. It does provide bug fix updates and some functional updates, but it never adds new areas, items, or creatures. The only thing it ever does to design new creatures is colorize an existing one and give it a new name. I tend to get very bored with fighting the same creature in 20 different colors. *UO* never evolves so it never feels anything like a living experience.

Another gripe is Origin's love of *nerfing*. Nothing is more irritating to players than to get very good at a skill only to have it nerfed. In the beginning of *UO*, magic users could become very powerful. Even third-level spells could inflict severe damage on other players. Well, guess what? *UO* reduced the amount of damage inflicted by third-level spells to almost nothing. You can literally stand and cast the lightning bolt spell on someone all day long and he will never die. This really annoyed me because I had spent months getting my magery high in level. Origin also did the same thing with weapons. It deemed some of the weapons in the game as too powerful, so it made them ultra-weak. Most players have resorted to joking about what Origin will ruin for them next. This is not a good thing.

***Nerfing* refers to making something less powerful. When Origin nerfed magery, it made the spells do less damage.**

I don't want to delve into the negative too much, so I'll leave it alone for now. I don't want you to get the wrong idea, though. I really did like *UO*—but its time has come and gone.

EverQuest

Well, wouldn't you know it; there I was still playing *Ultima Online*, and another online MMORPG came out. The new game that caught my attention was called *EverQuest*. It was created by a bunch of guys at 989 Studios, who later formed Verant. Sony publishes the game and maintains the game worlds. *EverQuest* kind of resembles *Meridian 59* in that it has a 3D game client. That's about the only way it resembles *Meridian 59*, though, because *EverQuest* is the best-looking MMORPG out today. Almost every zone in the game has a different look and feel. You really have to play the game to appreciate its beauty.

Unlike *Ultima Online*, the world of *EverQuest* consists of multiple zones or areas. This helps keep network lag to a minimum. Because no more than 100 people or so gather in a zone at a time, the servers don't have to keep track of too much. I talk about how zones work in the next chapter, so for now let's move on.

Character Races in EverQuest

EverQuest was unique when it came out in that you could chose from multiple races. You could be a troll, ogre, dwarf, elf, or human, for example. This selection allows for a great range of possibilities. Not only can you pick your race, but you can also pick what type of character you want to play, such as a monk, cleric, or fighter. Each race has pros and cons. For instance, dwarfs have night vision but cannot be monks. Humans can be any type of skilled character but have mediocre stats. There are just too many combinations to list here. Another interesting thing about races is that racism actually plays into the game. I don't mean that racism is good by any means, but it adds a dynamic to the game. For example, elves are not welcome in troll villages. So if you are an elf and you wander into a troll village, the guards might attack you. One way to make peace with the trolls is to kill their enemies. The old "enemy of my enemy is my friend" saying applies here. This minor part of the game does go a long ways toward adding depth to the game; keep that in mind when developing your own MMORPGs.

The Difficulty Factor

I played *EQ* for about four months. Then, all of the sudden, I quit. Why? *EQ* is just too darn hard and time-intensive for my lifestyle. I personally don't have time to invest four hours a day in a game and work full-time, write a book, and raise kids! The problem I had with *EQ* was that when you die, you lose hard-gained experience points. You can actually lose enough experience points to move down a level in the game. There I was one day, having a fun time playing the game, and then whammo, I got killed during a bout of network lag. Because the lag was so bad, I had been disconnected from the game. When I logged back in, I found that I had indeed died and that I had also lost a ton of experience points. I actually lost two weeks' worth of experience points. At that time, I did a quick calculation of salary versus time and decided it was costing me too much to keep up. I have better things to do with my life than play catch-up in a game. There is a lesson to be learned here: Make games somewhat difficult, but don't punish players with real-life penalties. You should make players want to avoid in-game death, but don't inflict insane penalties on them. Just remember, people play games to have fun, not to become so frustrated that they want to throw their computers out of windows.

Starsiege Tribes

Because I am never one to live without at least one game addiction, after *EverQuest* I immediately picked up a game by Dynamix called *Starsiege Tribes*. *Tribes* is the type of game commonly called a *shooter*. These types of games earn the title of shooter because the main objective is to shoot other players. Some other games that fit within this category are *Quake III Arena*, *Half-Life*, *Goldeneye 007*, and *Unreal Tournament*.

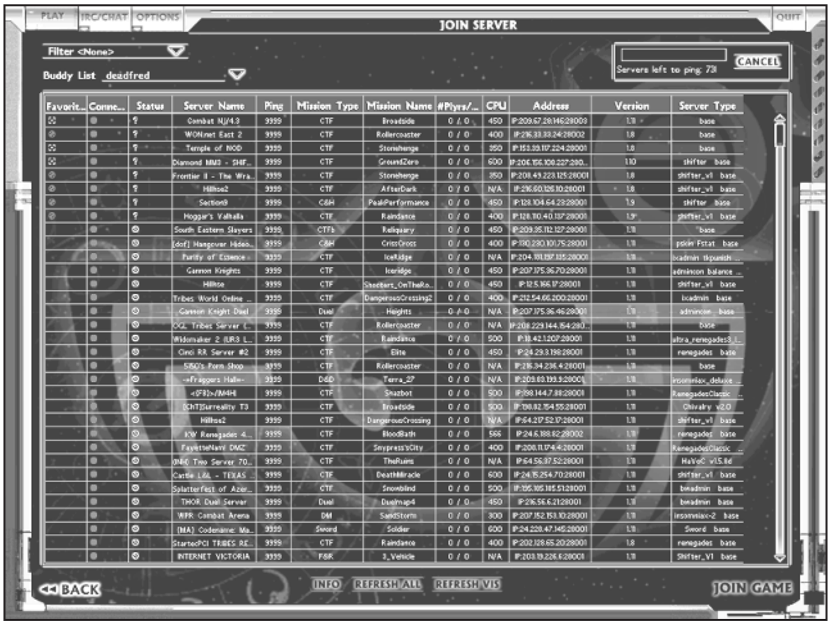
Because the only single-player aspect of *Tribes* is a training mission, the game is only played online. This aspect was considered a downfall when the game first came out but was finally recognized for its forethought. The game supports up to 32 players simultaneously, depending on the server connection speed and CPU. This capacity is quite an accomplishment considering the fast-paced nature of the game.

After you set up your name and look, you see a list of game servers. Because anyone with a decent connection can be a server, you have literally hundreds from which to choose. Refer to Figure 1.18 to see the server selection screen.

Once all of the servers are listed, you can choose the one on which you want to play. Some of the available game types appear in Table 1.4.

The key to most of the game types is teamwork. *Tribes* is unique as far as shooters go because it builds in all sorts of teamwork. Lone wolves need not apply in *Tribes*. Teams only do well when players work together. The number one reason I enjoy playing this game so much is that team play is much more fun than playing solo.

Another cool aspect of *Tribes* is that you can play defensively if you want. You can gear up in engineering armor and place defenses. One example is to place turrets and force fields around your flag. This helps keep enemy players from capturing it. The good part is that you get points for setting up defenses, so you have other types of gameplay besides shooting everything that moves. I guess that should be the lesson for this game: Give players optional play styles. Now that I am finished with the *Tribes* speech, you can see what the game looks like in Figure 1.19.



Filter <None>

Buddy List deadfred

Servers left to ping: 731

Favorit...	Conne...	Status	Server Name	Ping	Mission Type	Mission Name	#Plyrs/...	CPU	Address	Version	Server Type
⊠	●	?	Combat NJ/4.3	9999	CTF	Broadside	0 / 0	450	IP:209.67.28.146:28003	1.11	base
⊘	●	?	WONnet East 2	9999	CTF	Rollercoaster	0 / 0	400	IP:216.33.33.24:28002	1.8	base
⊠	●	?	Temple of NOD	9999	CTF	Stonehenge	0 / 0	350	IP:153.33.117.224:28001	1.8	base
⊠	●	?	Diamond MM3 - SHIF...	9999	CTF	GroundZero	0 / 0	600	IP:206.156.108.227:280...	1.10	shifter base
⊘	●	?	Frontier II - The Wra...	9999	CTF	Stonehenge	0 / 0	350	IP:208.49.223.125:28001	1.8	shifter_v1 base
⊘	●	?	Hillhoe2	9999	CTF	AfterDark	0 / 0	N/A	IP:216.60.126.10:28001	1.8	shifter_v1 base
⊘	●	?	Section9	9999	C&H	PeakPerformance	0 / 0	450	IP:128.104.64.23:28001	1.9	shifter base
⊘	●	?	Hoggar's Valhalla	9999	CTF	Raindance	0 / 0	400	IP:128.110.40.137:28001	1.9	shifter_v1 base
	●	◎	South Eastern Slayers	9999	CTFb	Reliquary	0 / 0	450	IP:209.35.112.127:28001	1.11	base
	●	◎	[dof] Hangover Hideo...	9999	C&H	CrissCross	0 / 0	400	IP:130.230.101.75:28001	1.11	pskin Fstat base
	●	◎	Purity of Essence	9999	CTF	IceRidge	0 / 0	N/A	IP:204.181.197.135:28001	1.11	bcadmin tkpunish ...
	●	◎	Gannon Knights	9999	CTF	Iceridge	0 / 0	450	IP:207.175.36.70:28001	1.11	admincon balance ...
	●	◎	Hillhoe	9999	CTF	Shooters_OnTheRo...	0 / 0	450	IP:12.5.166.17:28001	1.11	shifter_v1 base
	●	◎	Tribes World Online ...	9999	CTF	DangerousCrossing2	0 / 0	400	IP:212.54.66.200:28001	1.11	bcadmin base
	●	◎	Gannon Knight Duel	9999	Duel	Heights	0 / 0	N/A	IP:207.175.36.46:28001	1.11	admincon base
	●	◎	OGL Tribes Server (...	9999	CTF	Rollercoaster	0 / 0	N/A	IP:208.229.144.154:280...	1.11	base
	●	◎	Widomaker 2 (UR3 L...	9999	CTF	Raindance	0 / 0	500	IP:18.42.1.207:28001	1.11	ultra_renegades3_l...
	●	◎	Cinci RR Server #2	9999	CTF	Elite	0 / 0	450	IP:24.29.3.198:28001	1.11	renegades base
	●	◎	5150's Porn Shop	9999	CTF	Rollercoaster	0 / 0	N/A	IP:216.34.236.4:28001	1.11	base
	●	◎	-=Fraggers Hall=-	9999	D&D	Terra_27	0 / 0	N/A	IP:209.83.199.9:28001	1.11	insomniax_deluxe ...
	●	◎	<{FB}>-/M4H	9999	CTF	Snazbot	0 / 0	500	IP:198.144.7.88:28001	1.11	RenegadesClassic ...
	●	◎	[ChT]Surreality T3	9999	CTF	Broadside	0 / 0	500	IP:198.82.154.55:28001	1.11	Chivalry v2.0
	●	◎	Hillhoe2	9999	CTF	DangerousCrossing	0 / 0	N/A	IP:64.217.52.17:28001	1.11	shifter_v1 base
	●	◎	KW Renegades 4...	9999	CTF	BloodBath	0 / 0	566	IP:24.6.188.82:28002	1.11	renegades base
	●	◎	FayetteNam DMZ	9999	CTF	Smypress'sCity	0 / 0	400	IP:208.11.174.4:28001	1.11	RenegadesClassic ...
	●	◎	(NH) Two Server 70...	9999	CTF	TheRuins	0 / 0	N/A	IP:64.56.37.52:28001	1.11	HaVoC v1.5.lld
	●	◎	Castle L&L - TEXAS ...	9999	CTF	DeathMiracle	0 / 0	600	IP:24.15.254.70:28001	1.11	shifter_v1 base
	●	◎	Splatterfest of Azer...	9999	CTF	Snowblind	0 / 0	500	IP:156.105.185.51:28001	1.11	bwadmin base
	●	◎	THOR Duel Server	9999	Duel	Duelmap4	0 / 0	450	IP:216.56.6.21:28001	1.11	bwadmin base
	●	◎	WPR Combat Arena	9999	DM	SandStorm	0 / 0	300	IP:207.152.153.10:28001	1.11	insomniax-2 base
	●	◎	[MA] Codename: Ma...	9999	Sword	Soldier	0 / 0	600	IP:24.228.47.145:28001	1.11	Sword base
	●	◎	StartecPCI TRIBES RE...	9999	CTF	Raindance	0 / 0	400	IP:202.128.65.20:28001	1.8	renegades base
	●	◎	INTERNET VICTORIA	9999	F&R	3_Vehicle	0 / 0	N/A	IP:203.19.226.6:28001	1.11	Shifter_V1 base

Figure 1.18

The server selection screen in Tribes

Table 1.4 Game Types in Starsiege Tribes

Game Type	Summary
CTF	A capture-the-flag game. In this type of game, each team has a flag to defend. The objective of the game is to capture an enemy's flag and bring it back to your own. Once you have done this, your team is awarded one point.
C&H	A capture-and-hold game. In this type of game, you have objectives to defend. The object of the game is to capture an objective and then defend it from the enemy. The longer you hold an objective, the more points your team gets.
DM	The deathmatch game for Tribes. In deathmatch mode, it is every man for himself. In this type of game, points are awarded for every kill you get.

```
Emergency Force Shields Exausted
Emergency Force Shields Activated
ArrogantGod blasts AXIS* Lt Hydro with flying death.
```

```
Score +2 = 23 Total Score .
```

Figure 1.19

An in-game view from a player in Starsiege Tribes

Asheron's Call

Well, I still play *Tribes*, but I also have found another game that I enjoy quite a bit. *Asheron's Call* is published by Microsoft. The game company called Turbine actually wrote the software; Microsoft just publishes the game and maintains the servers.

Asheron's is yet another example of an MMORPG-style game. It has a 3D client interface and supports thousands of simultaneous players. I started playing it sometime in the year 2000. It is not the best-looking game on the market—*EverQuest* looks much better—but it is the most dynamic. The number-one reason I have stuck with *AC* is that Turbine actually updates the game once per month. The good part is that when Turbine updates the game, it does so in grand fashion. The game updates are usually very significant and they progress the in-game stories. The game world is truly dynamic because of this feature. New towns, creatures, and weapons are just some of the possible items introduced with each patch.

The Present and the Future

I have covered just a small sampling of the many online games available. The great part is that most games coming out today have online support. Even console systems such as Dreamcast and Xbox are getting into the online market.

The greatest hope that is becoming more available for online gaming is broadband communications. Technologies such as cable modems, DSL, ISDN, and others are giving home users super high-speed connections to the Internet. Not only are the methods of communication getting faster, but they are also getting cheaper.

I hope in the not-too-distant future that almost everyone has broadband access at an affordable rate. This will enable developers such as you to write games that are not limited by communication bandwidth. Truly immersive worlds where thousands of people participate in real time will be possible.

Review

Okay, here you are at the end of Chapter 1. I don't want you to have to dig back through the chapter for any tidbits of knowledge, so let's recap.

I'm not trying to preach here or anything, but I do believe the following lessons are key to good games. If you can think of any additional good ones, please e-mail them to me at webmaster@lostlogic.com and I will share them on my Web site.

- Lesson 1: Keep players challenged but not frustrated.
- Lesson 2: Give players optional play styles. Not everyone likes to play a game the same way.
- Lesson 3: A good online game has the following elements: an intuitive interface, good AI, and good network support.
- Lesson 4: The more gain that players can get from others' misfortunes, the more anti-social behavior will be present in a game.

My time is done for now. I hope you enjoyed this little trip down memory lane with me. Expect much more to come, though, because in the next chapter I actually tell you how these games work!

TIP

In case you are wondering, this chapter gets its title from a common phrase PKs use in *Ultima Online* to get your attention. PKs say something like "Sup, Wait!" to get you to stop, and then they kill you.

CHAPTER 2

Understanding the Magic

Before you can start creating your own games, you need to understand how games operate. I'm not talking about the rules of gameplay or anything like that; I'm talking about how games actually work. If you are an experienced game developer, then you probably know how most games work, but read on and you just might find something useful.

Most of this chapter is written from what I like to call the 15,000-foot level. When I talk about complex subjects, I commonly think about them as somewhere between the 1,000-foot level and the 50,000-foot level. I use the lowest altitude when talking to fellow engineers and I reserve the highest altitude for talking to my family. So this puts us cruising just outside of Vegas. When we land, be sure to stay away from the Keno games; they're a rip-off.

I just called 1-900-Lie-ToMe, and the tarot card reader told me to tell you what topics are covered in this chapter, so here they are.

- Loops
- Graphics
- Sound
- Artificial intelligence

Game Loops

I use the term *game loop* to refer to the architecture behind a game's process. To give an analogy, think of the loop a car engine uses.

1. Fuel is injected into the cylinders.
2. The spark plug ignites the fuel.
3. The resulting explosions move the pistons.
4. The pistons turn the gears.
5. Repeat.

There's not too much to it really, just your basic combustion routine. Game loops also work in a simple manner when you get right down to it. When you think about how a game works, what is the first thing that comes to mind? Graphics, sound, input? Most people generally think about the graphics involved in a game, but

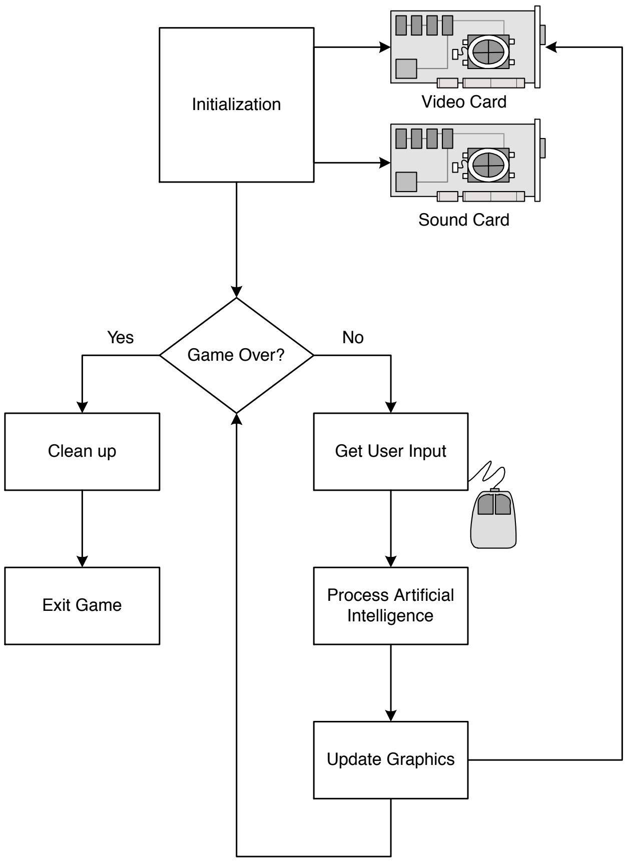


Figure 2.1

Game loop architecture

graphics are really just fluff for games. The real work happens behind the scenes in the logic loop. Take a look at Figure 2.1 to see an illustration of a basic game loop.

Initialization

The first thing that happens in the game loop is initialization. This process refers to setting up the environment in which the game will be played. You must handle many things here, including graphics, sound, input, and network initialization.

Graphics Initialization

DirectX requires that you complete a series of steps before displaying graphics. For instance, you must create a window in which to display the graphics. This is also the case for full-screen applications. Even though the game is full-screen, that doesn't mean there isn't a window. In fact, there is a window—you just don't see it most of the time.

In addition to creating the display window, another step is initializing any video hardware the user might have present. Usually, this means setting up a 3D video card. Because most mainstream gamers have 3D video cards, you must activate them. The same goes for sounds cards and input devices. If you want to play sound effects in your game, you need to set up the audio environment.

Screen Resolution

Generally, starting up a game for the first time means a pause while it loads. Games must preload most of the information players need. For example, the title screen and background music are loaded when the game starts for the first time. Many files must be in memory before the game can start. All of this processing and data loading makes up the initialization phase.

Now drop down to a lower level, and take a look at what must happen to initialize a graphics display. You must

1. Determine screen resolution.
2. Determine color depth.
3. Determine which adapter to use.
4. Activate the adapter.
5. Set the video mode.

The first item required is the screen resolution. Screen displays consist of tiny little dots called *pixels*. The number of pixels displayed at once represents your screen resolution. So if you have 800 vertical pixels and 600 horizontal pixels, your display is said to be in 800 × 600 mode. The first number of a resolution reference is always the number of pixels across the display. The second number always represents the number of pixels down the screen.

***Pixels* are the dots on computer monitors that make up images. Higher-resolution images have more pixels in them. The word *pixel* actually comes from the words *picture element*.**

Graphics and displays always look better when your resolution is set higher. This is why HDTV, or high-definition television, looks so good—it has a higher resolution. Here are some common video resolutions.

- 640 × 480
- 800 × 600
- 1024 × 768
- 1024 × 1280
- 1600 × 1200

The first resolution, 640 × 480, used to be the standard size most games used. Given today's hardware, most games operate at much higher resolutions now. I find that the 1024 × 768 mode gives me the most bang for my buck. Take a look at Figure 2.2 to see an illustration of pixels.

Color Depth

Now that I have covered screen resolution, it's time to talk a little about *color depth*. Color depth is always referenced in bits. Older computer video cards had only 8 bits of color to utilize, but most cards out today can support 32 bits of color.

***Color depth* refers to the number of colors in bits that an application uses.**

So what does this really mean? Well, the number of bits actually refers to the number of colors available—the more bits, the more colors. Refer to Table 2.1 to see some common color depths.

All of the modes look pretty normal except for 32-bit mode. Basically, it is true-color with an alpha channel. Don't worry—I'll explain alpha channels a little later.

Video Adapters

Even though most people have only one video card, many people out there have multiple cards. This was especially true back when 3D accelerator cards were sold separately. Some people like to have a good card for 2D and then a good card for 3D. This can result in a multicard system. Another case for this situation is when you have a computer with built-in video and then decide to upgrade to a new card. In this case, you have two video adapters, the built-in one and the new one.

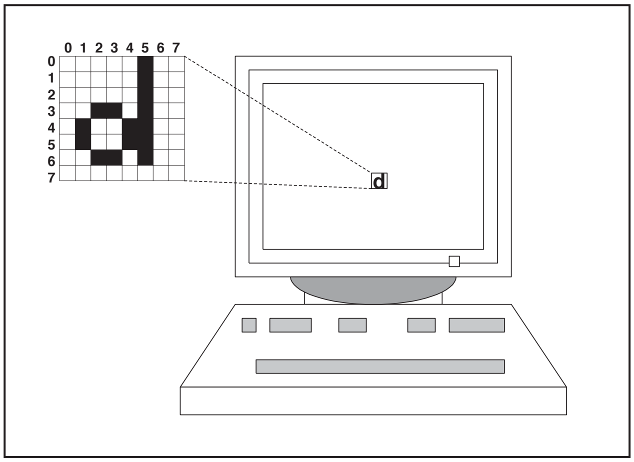


Figure 2.2

The letter d can be viewed as a series of pixels if you zoom in on it.

Table 2.1 Color Depth

Depth	Meaning
8 bits	256 colors. This is referred to as low-color mode. Each pixel takes up one byte of memory.
16 bits	65,536 colors. This is referred to as high-color mode. Each pixel takes up 2 bytes of memory.
24 bits	16.8 million colors. This is referred to as true-color mode. Each pixel takes up 4 bytes of memory.
32 bits	16.8 million colors + 256 grayscale alpha channel. This has the same number of colors as true-color mode, but it also has an alpha channel. Each pixel takes up 5 bytes of memory.

Because you can have multiple cards, DirectX requires that you specify the display adapter before proceeding. I cover how to do this later, but for now just remember that you must select the video card to be used. Most of the time, you just select the *primary display adapter*.

The *primary display adapter* is the one currently in use by Windows. It commonly refers to the most preferred adapter in a system. The video adapter with the most features and power is usually the most preferred.

Video adapters that are not primary are usually called secondary adapters. After you decide which adapter to use, you must initialize it. You do this with DirectX, and the process really isn't a big deal.

Setting the Video Mode

When initializing the display adapter, you normally pick the screen resolution and color depth. You choose them once at the beginning of your application. Sometimes, it is necessary to change the resolution in the middle of a game, but not normally. Another choice you must make when setting the video mode is whether your game will be a *full-screen* application.

***Full-screen* applications take up the entire display and leave no Windows controls visible. When an application does not run in full-screen mode, it is said to be *windowed*.**

Once you choose the video mode and whether it is windowed or full-screen, then the application switches the user's desktop to that mode. Just be careful not to switch the video display to something invalid.

Another concern about the video mode is the amount of memory required to achieve it. You can calculate the amount of memory required to use a specific display using the following formula.

CAUTION

Some older 3D display adapters require that you be in full-screen mode to realize their acceleration capabilities.

Memory required = Screen width × Screen height × Color depth

For example, an 800 × 600 screen in 24-bit mode takes up 800 × 600 × 4 bytes of memory. That's 1,920,000 bytes of video memory! Now just think about how some video cards out there have only 1MB of video memory. They wouldn't be able to display such a screen. Keep this in mind when setting video displays.

Sound Initialization

Sound initialization is much more straightforward than video initialization. The main reason for this is that you can play multiple-resolution sounds without having to switch sound modes. When I mention *sound resolution*, I am talking about the quality of the sound.

Sounds with a higher resolution have a higher sample rate. Sample rates determine how often a sound has been sampled when recording it. The more samples taken from a sound source, the more accurately the played-back sound represents the source.

Imagine that you are copying a mural that is 100 feet wide by 20 feet tall. If you copied only one brush stroke every two inches, then you would get a low-quality sample of the original painting. You would have holes in your copy, and it would not look as good as the original. But if you copied a brush stroke every two millimeters, then you would get a high-quality sample, and it would look very good. Sound works in much the same way; if you sample a sound at a low rating, then it does not sound very good. To see an illustration of sampling, take a peek at Figure 2.3.

Some sound samples can handle low sample rates without much notice from humans. You can usually sample music at lower ratings because it has so many changes. The sound just changes too quickly for your ears to pick up on the problems. Sounds that have problems being sampled at low ratings are voices and such. People tend to pick up on low-quality voices more quickly than low-quality sound effects or music. Higher-quality sounds also take up much more memory. Voices alone consume quite a bit of storage in a game.

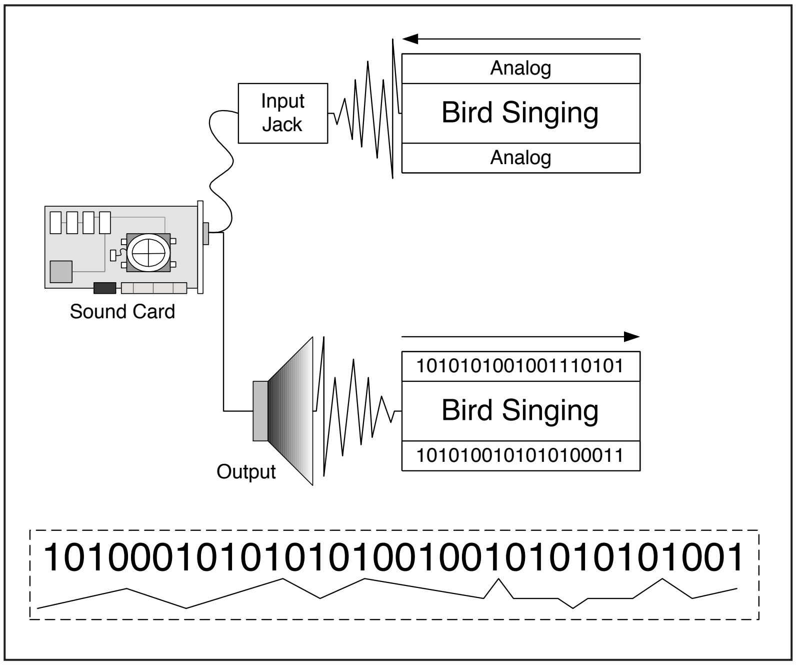


Figure 2.3

When an analog sound is converted to digital data, the sampling rate determines the quality of the sound.

As with video adapters, you must choose the sound adapter to use. Most people have only one sound card, but there are exceptions. Once you choose the sound card to use, you just choose whether you want to use 3D sound or 2D sound. You make these choices with the DirectX *API*.

***API* stands for application programming interface. The term refers to a set of grouped commands or functions used in programming.**

Input Initialization

Luckily for us, setting up input devices doesn't require much work. Most of the commonly used devices are already set up by Windows. In fact, if your application uses only a keyboard and a mouse, then you don't have to do anything to get started.

Most programmers initialize input devices when their program supports other peripherals such as joysticks. Because not all joysticks operate in the same manner, you must initialize the system to recognize the one the user has. You accomplish this by querying the system for all available input devices and then picking the one the user wants to use. Some possible input devices appear in Figure 2.4.

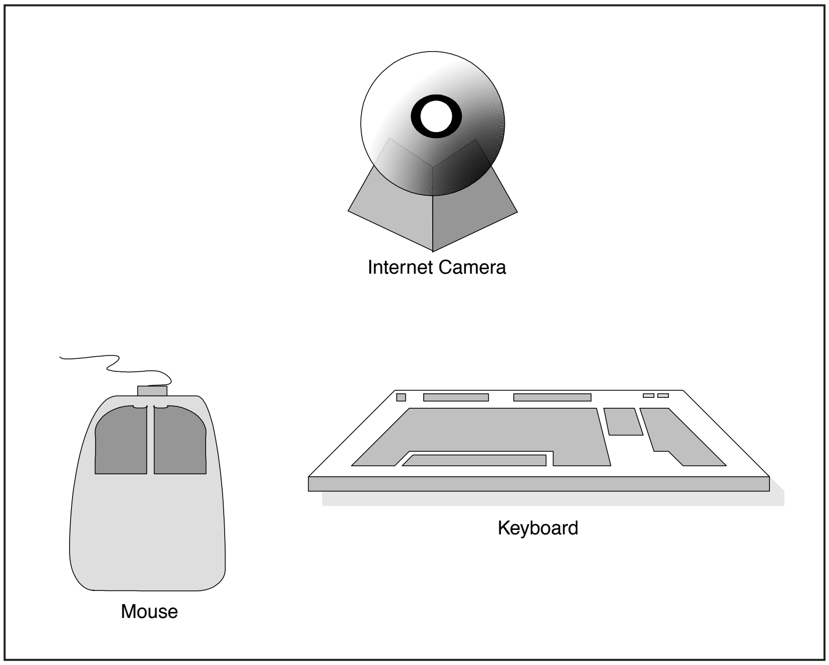


Figure 2.4

Possible input devices

Good ol' Microsoft provides an API to use input devices: DirectInput. DirectInput is a part of the *DirectX SDK* and gives you the freedom to use most input devices out there. It even supports force-feedback devices, which are really cool.

SDK stands for software development kit. An SDK provides one or more APIs for programmers to use. Microsoft's premier SDK for game programming is the _DirectX SDK_. It is also used in developing games for Microsoft's game console, the Xbox.

Network Initialization

Oh, boy, my personal favorite! Nothing beats a good network initialization. Because computers don't just get up and start talking to one another, you must work to establish such communications.

Before the computer game can begin to communicate with others, you must teach it how. Selecting which *communication protocol* to use accomplishes this. You can think of a communication protocol as a method in which to speak.

Communication protocols are methods computers use to speak to one another. Without protocols, computers do not know how to talk to each other.

Once you select how your computer will communicate, you can do such fun things as getting it online and then connecting to another computer. If you just can't wait and you want to see what protocols are, you can jump ahead to the next chapter. Just be sure to come back!

Content, Content, Content!

Do not be intimidated by all the initialization stuff; it's not all that bad once you have done it a few times. The great part about initialization routines is that they change very little, so you can reuse them in other projects.

Now that the game hardware is initialized, you must get the content for the game. What is content? Game content refers to the images, sounds, and data used by a game. Without content, a game is nothing more than a logic loop. Just think about how little fun *Final Fantasy* would be without graphics!

Most advanced games load content dynamically when needed. For our purposes, I'm just going to talk about games that load most of the data at the beginning. Because content is data, it takes up memory and must be monitored closely. Maybe you have wondered why games don't use more graphics and sounds; the reason is that generally, there is not enough memory to handle it. Because memory capacity is ever-increasing and the cost is decreasing, using graphics and sounds is becoming less of an obstacle every day.

Graphic Files

The graphics used in games are stored in certain *graphic formats.* A graphic format describes how an image is stored and how to read it.

Graphic formats are file formats that describe a graphical image. Common graphic formats include BMP, JPEG, and TARGA.

Information such as the graphic's dimensions and color depth are stored in the file's header. The most important information stored in a graphic file is the image data. Without image data, the graphic doesn't exist. Each and every pixel of an image is stored in the file. Refer to Figure 2.5 to see a *file header.*

A _file header_ refers to the bytes stored at the beginning of a file. Headers usually describe the contents of the file.

You can count on graphics taking up the most memory and physical storage in your games. Once your game has loaded all of the graphics, it can then proceed to load the sound content.

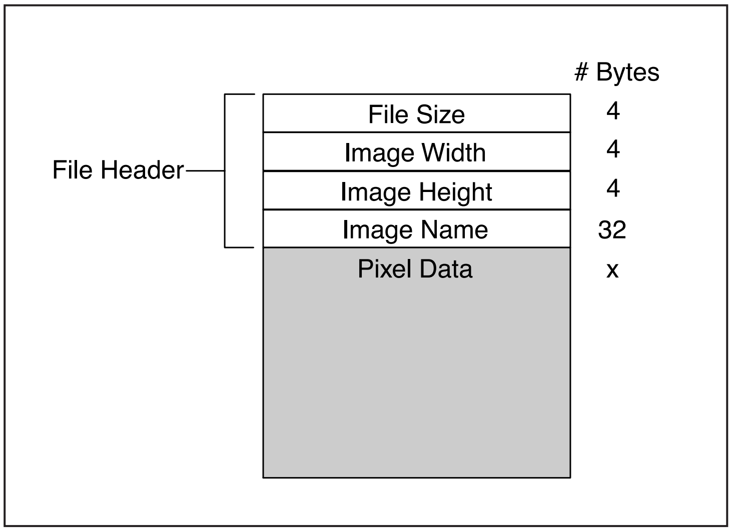


Figure 2.5

Example structure of an imaginary file header

Sound Files

Sound files are just like graphic files except that sound files store sound information. Sound files also have a header, but sound-file headers describe the sample rate and other information. Take a peek at Table 2.2 to see some common sound file types.

Table 2.2 Sound File Types

Type	Meaning
Wave	This is the most common type of sound file used in games and most other applications. Almost every sound effect used in a game is stored in this format.
MP3	This is a highly-compressed format used to store music. It can save a lot of storage space. This format is also popular in replacing CDs.
MIDI	MIDI files were once primarily used for game music. Because computer systems nowadays have much more power, this format is antiquated. It doesn't actually store recorded sound. It stores information necessary to play the music via instruments.

I use wave files for all of my in-game sounds and music because they allow for easy editing. Almost every sound-editing package out there today supports wave file manipulation. Music from CDs can also be *streamed* in. This allows for super high-quality music that doesn't take up too many resources.

When data is *streamed*, it is loaded in pieces. When a piece is used, it is then unloaded to make room for the next piece. This process lets you use large pieces of data, such as music files, without storing them all in memory at once.

Data Content

Obviously, the graphics and sounds are data, but what I refer to here is all of the data other than media. Games need such interesting things as in-game text, maps, geometry, and a myriad of other data.

Inside the Game Loop

Well, now that initialization is completely out of the way, the user can play the game. The first thing your game can check for is user input. After all, it is helpful to know what the player wants to do.

Checking User Input

This check is a fairly straightforward process because all you have to do is poll the input hardware and see what has been altered. For example, you can check whether the left mouse button was pressed. Whenever an input event transpires, you should store the information for later retrieval, which lets other processes check the state of the hardware without having to poll for it again.

Once input is checked, you then alter the game based on that input. For example, if the player presses the left cursor key, you might want to move the spaceship to the left. If the player presses the fire button, you probably create a missile or bullet.

You probably can relate to these types of causes and effects if you have ever played a video game. In fact, most applications must do this checking of user input. Without it, there is no interactivity.

Processing Artificial Intelligence

After you check for user input, it is time to move the computer-controlled players. Artificial intelligence, or AI, is a complicated subject because there are countless methods for implementing it.

The AI in your game might do all sorts of things, such as moving, attacking, or dying. Because the player does not control these things, you have to. Even though I use the term AI, don't think that this applies to *intelligent* objects only. This part of the game loop must also process all in-game objects such as bullets.

Updating the Display

Now is the fun part, where you get to update the display for the player. The only way for players to notice something has changed in the game is to update their displays. You do this by drawing images on-screen with the video adapter.

Take, for instance, the game of *Asteroids*. In *Asteroids*, you draw the asteroids, the player's bullets, and the player's ship. If the player moves his ship, you draw it in a new location. If an asteroid blows up, you might display the image of an explosion.

Because every game is different, I cannot cover all possibilities. Luckily, graphic output is the most obvious effect in video games. Just go look at one of your favorite games, and pick out all of the graphical elements present. I'll bet you find hundreds of things.

Cleaning Up

Remember all of that content you loaded into your game upon initialization? Well, guess what? You must remove all those elements to free up memory. When your game exits, you need to unload any graphics, sound, or data that you loaded at the beginning. If you do not do this, the system will eventually run out of free memory.

TIP

Be sure to remove everything that you allocate; there is nothing more critical to game development than cleaning up.

Inside Online Games

Since this book mainly concerns multiplayer games, I guess I'd better start talking about how they work. The main element of an online game is communication. Without communication, the game wouldn't even be online, so as you can see, it is a very important element.

Client/Server Games

Let's jump right into the thick of things, and talk about how *Ultima Online* works. *Ultima Online* is a client/server game that supports thousands of players. Okay, so what is a client/server game? In a client/server game, the players' computers talk only to a central computer and not to one another. The computer they talk to is called the server, and the players' computers are called clients. All communication must go through the server. This requires that the server has a lot of bandwidth. If the server runs out of communication bandwidth, then the players notice lag.

Clients and Servers

Clients in *UO* send requests to the central servers. The central servers process the requests and send back responses. If a player wants to send a chat message to another player, it is first sent to the server, and then the server sends it to the target player. This kind of communication allows for increased security and also allows for large numbers of players. Because servers must validate every move a client makes, they can deny certain requests. To see the client/server architecture in action, refer to Figure 2.6.

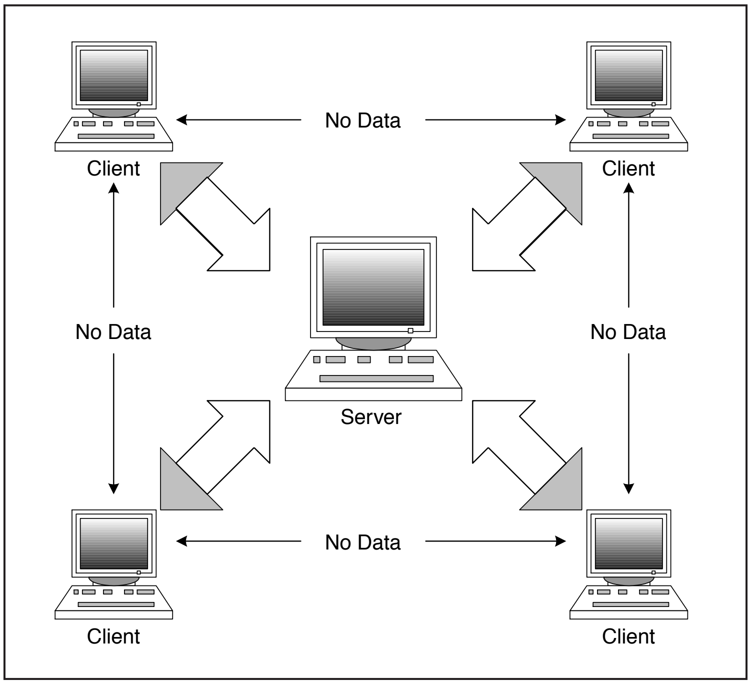


Figure 2.6

Client/server architecture

Requests and Responses

A command sent to the server by a client is called a request. It is called this because the client is requesting a response. Responses range from simple acknowledgments to complex commands. Examine a scenario in which a player wants to move his character forward one square.

1. The player presses the up-arrow key to move his character forward one square.
2. The client application sends a move request to the game server.
3. The game server receives the request.
4. The game server checks the location into which the player wants to move. If the location is blocked, it sends an invalid move response. If the location is not blocked, the server sends a valid move response.
5. The player happens to be moving into a non-blocked area, so he receives a valid move response.
6. The player's character is moved forward one square.

Notice how the server validates the player's request. This step ensures that the player is moving into a valid location, which helps keep cheating to a minimum. How is this, you might ask. Well, imagine that the player had hacked his game to remove all obstacles, such as walls. He could then move his player around the game world without obstruction. This means he could walk through oceans, walls, or anything else that gets in his way. He could wreak much havoc in such a situation.

I know that hacking a game client seems a little extreme, but believe me, many people hack online games. In fact, I witnessed several hacks in *Ultima Online*. (I'll talk about those in a minute.) Anyway, take a look at the request and response architecture in Figure 2.7.

In the request and response architecture diagram, you can see the method I discuss. An opportunity for hacking does still exist, though. Even though the opportunity allows only limited cheating, it can still be effective. One example I witnessed in *Ultima Online* used a little program called UOExtreme. This program was released by some unknown player and allowed the user to do certain things not normally allowed in *UO*. It accomplished the cheats by intercepting responses from the server. It would intercept a response, examine the contents, and then modify the response accordingly.

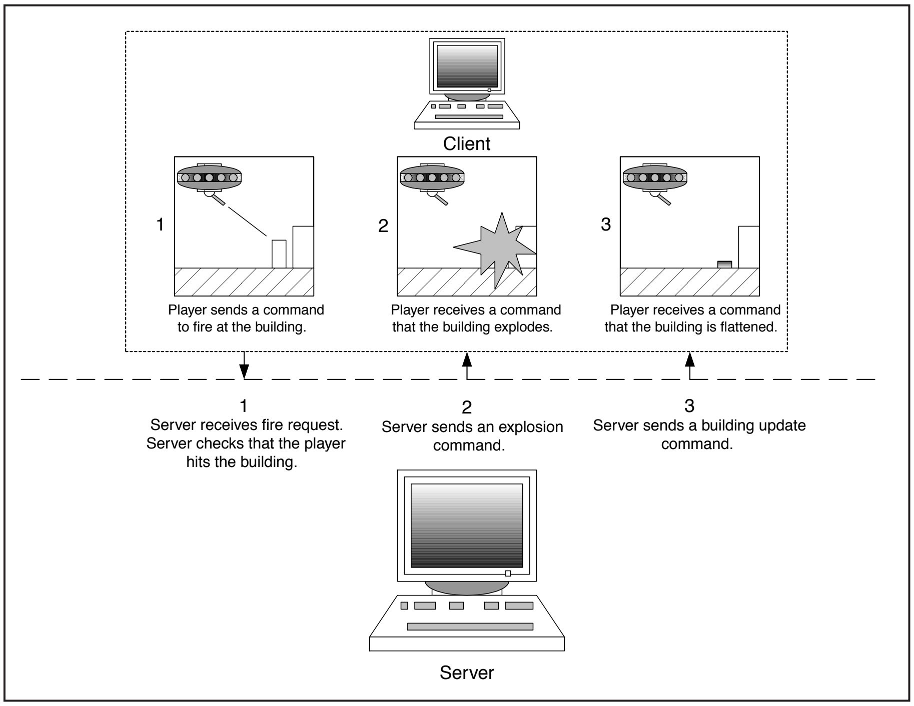

Client
1
Player sends a command to fire at the building.
2
Player receives a command that the building explodes.
3
Player receives a command that the building is flattened.
1
Server receives fire request. Server checks that the player hits the building.
2
Server sends an explosion command.
3
Server sends a building update command.
Server

Figure 2.7

Request and response architecture

Hacking Ultima Online

Let me give you some background about the game mechanics of *Ultima Online* so I can better explain some of the hacks used. In *UO*, time passes at an accelerated rate. The only way to visibly note the time is to see how dark it is. At night, the entire game looks darker; during the day, it is lighter. No big surprises here. Oh, yeah, in a dungeon, it is normally dark. Because it can get dark, players have to either carry torches or cast light spells to see well.

To set the time of day, the servers send time commands to clients. The time commands contain a light level. When it is nighttime, the servers send out a low-light setting command. The clients receive the light settings and adjust their displays accordingly. UOExtreme would intercept these light packets and always change the light value to the highest setting possible. This allowed the users of UOExtreme to see daylight no matter what time it was! This might not seem like a big hack, but it sure did make life easier for users of the hack utility. Take a look at Figure 2.8 to see command interception illustrated.

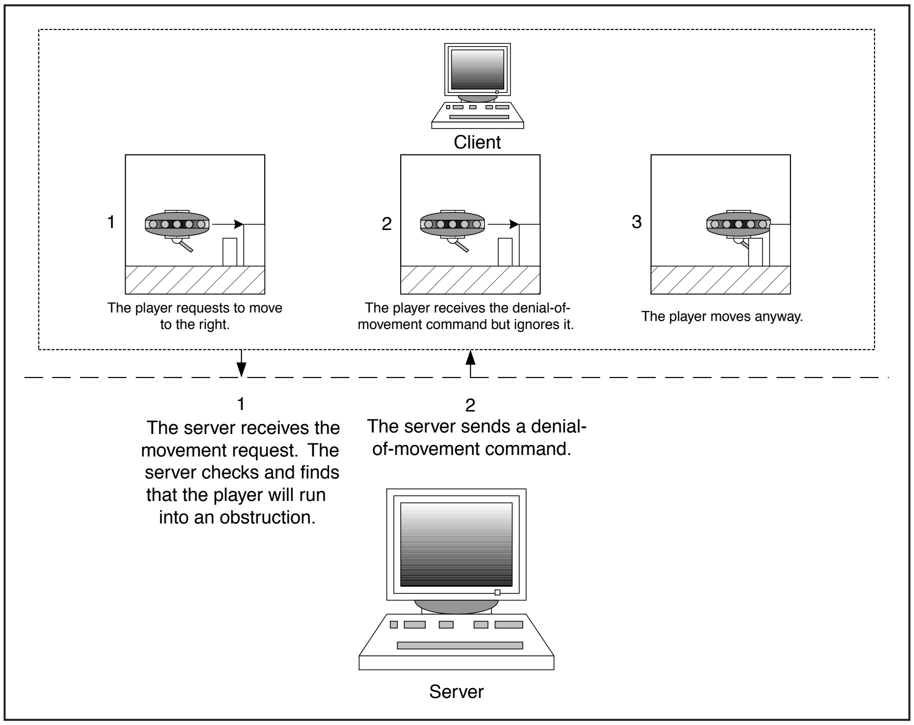

Client
1
2
3
The player requests to move to the right.
The player receives the denial-of-movement command but ignores it.
The player moves anyway.
1
The server receives the movement request. The server checks and finds that the player will run into an obstruction.
2
The server sends a denial-of-movement command.
Server

Figure 2.8

Intercepting server commands

Another more devious use of UOExtreme, or UOE, was to let players see hidden people. *Ultima Online* offers a *hiding* skill that lets players become invisible. If a hidden player moves, he is revealed, but if he stays still, no one can see him. This skill has always been useful for laying ambushes and such. Behind the scenes, the hiding skill works by sending an invisible command to other players. Other players still receive a command telling them you are on-screen, but it tells their client programs to hide you from view. Well, this process gives UOE a great opportunity. UOE merely intercepts the hidden packets and keeps the client program from receiving them. This lets all users of UOE see hidden people! Not only can players with this program see the hidden people, but they can also attack them. It basically renders hiding useless. I would have to say that this particular hack is what gained the attention of Origin, the makers of *Ultima Online*.

Maybe you are asking how *Ultima Online* ever solved this little problem. The programmers at Origin originally created several updates to the game clients that would help them detect when UOE was running. This idea was never very successful because the developers of UOE just changed their code with every client update.

Eventually, Origin came up with a neat way to detect cheaters. *UO* has monsters that rarely ever get far into towns because the town guards kill them on sight. With this system, you almost never see a creature in the middle of town. Game administrators used this to their advantage to catch UOE characters. First, the game admins entered the game as a creature. Not only did they assume the forms of creatures, but they also changed their colors to something like purple. So you had game admins walking around in purple troll bodies. The admins even went one step further and made themselves invisible. Do you remember how UOE let people see invisible players? Well, people running UOE saw a purple troll walking through town and said something like, "Hey, there's a purple troll in town!" The game admins then knew exactly who was a cheater. Non-cheaters couldn't see the troll in the first place because it was invisible. All the admins had to do was ban people who mentioned the purple trolls. Everyone else had no idea what the cheaters were talking about anyway because they couldn't see the invisible administrators. It's kind of funny when you think about it; the cheaters got themselves caught.

Eventually, Origin changed the code stream enough that UOE stopped working. If the authors of UOE had kept updating the program, it would probably still work today. But the developers got bored with UOE and quit working on it. Even though this one hacking utility quit working, it does not mean hackers quit messing with the game.

Interfaces

One thing that normally separates most clients from servers is the interface. The client application is responsible for graphical updates. All the servers do is handle requests and responses. Because servers don't have to worry about displaying graphics, they can devote all of their power to processing requests.

If you try running a *Quake* server, you'll notice that the actual server program runs in a separate console window because it uses no graphics. A good side effect of clients and servers not being tied to each other graphically is that you can update clients without changing the servers. Because the servers don't really care about the client graphics, you can change the client graphics all day long without affecting the server code.

Peer-to-Peer Games

The main difference between peer-to-peer games and client/server games is that peer-to-peer games have no dedicated server. In the peer-to-peer world, each player is responsible for sending requests and responses to all of the other players. This

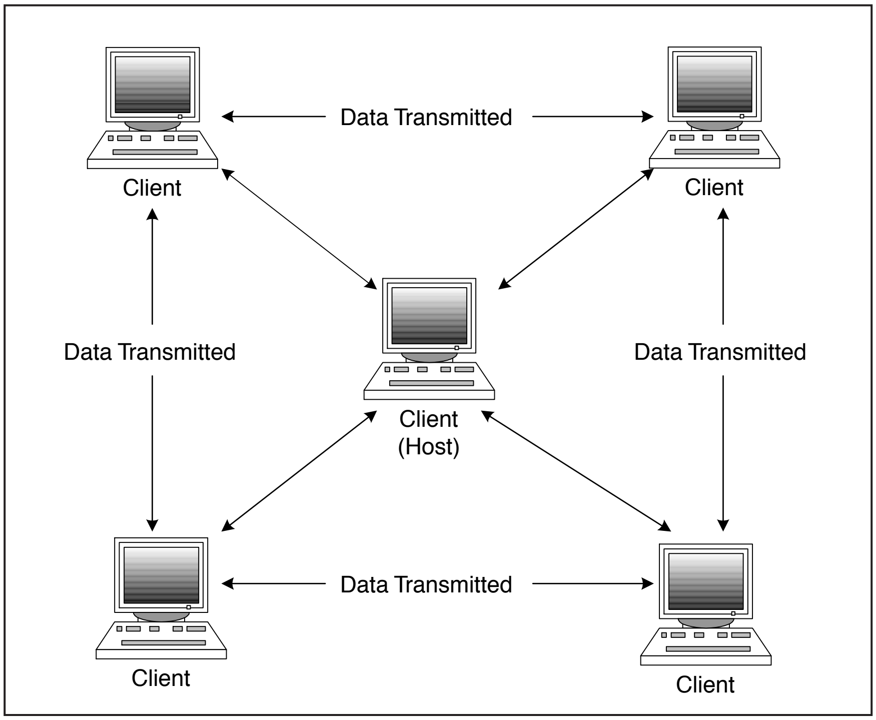


Figure 2.9

Communication traffic in peer-to-peer games

requirement alone limits the maximum number of players possible in this type of game. Take a look at the traffic flow in Figure 2.9.

Figure 2.9 shows a lot of communication traffic in peer-to-peer games. Basically, whenever a player wants to do something such as move forward, he must transmit his action to every player present. One thing that sets this process apart from that of a client/server game is that the player does not wait for permission to complete the action. Take a look at the following steps to see this concept in action.

1. Player A presses the joystick up to move forward.
2. Player A sends the movement command to Player B.
3. Player A continues playing.
4. Player B receives the movement command and updates Player A's position in his game.

Notice how Player A continues playing after sending the command to move. Because the game has no server, Player A couldn't care less what the other players think. No security or validity checks take place in this scenario. This process can lead to a lot of hacking because players assume commands coming from other players are valid. In fact, a group of us used this weakness to hack *Diablo* and *Diablo II*.

In *Diablo II*, by Blizzard, character data is stored in simple-to-read data files on the hard drive. All you have to do is open the character file with a hex editor and change the values. Within a few minutes of owning the game, we edited our characters to have millions of hit points. Not only were our characters invincible, but they also had the best offensive spells and capabilities possible. In no time, we were roaming around Battle.net, killing people left and right in PK games. Every once in a while, we ran across another hacker and things got interesting. It just goes to show you that people will not play games the way you expect.

Graphics, Graphics, Where Are the Graphics?

I did talk a little earlier about graphics, but I didn't really cover in detail how they work. Before you can write a video game, the one thing you must fully understand is graphics. Without them, you wouldn't have much of a game to play.

Here is a brief list of topics I have already covered relating to graphics.

- Video adapter initialization
- Screen resolution
- Pixels
- Color depth

So you now know that video adapters are responsible for manipulating the display. You also should know that the screen consists of pixels. With this knowledge, you are not far from understanding game graphics. The next thing to understand is *image composition*. By image composition, I mean the process of arranging images on the screen to create a complete picture. This topic might not sound like much, but it is one of the most complicated aspects of game programming.

Bitmaps

The first element of a composition is an image. Images in game development are called *bitmaps*. An example of a bitmap is the spaceship in the game of *Asteroids*.

A *bitmap* is a graphical image primarily displayed on-screen.

All bitmaps share common characteristics. Table 2.3 describes some common attributes of bitmaps.

Table 2.3 Common Bitmap Attributes

Attribute	Meaning
Width	The number of pixels in the image horizontally.
Height	The number of pixels in the image vertically.
Color depth	The number of colors available in the image. The depth is represented in bits just as screen resolution is.
Color key	The color to be treated as transparent.

Bitmap Size

The width and height attributes are fairly straightforward because they represent in pixels the size of the image. Most bitmaps follow sizes evenly divisible by four. For example, a 32 × 32 image is common, whereas a 32 × 31 image is not. The reason is that many systems can deal with four bytes at a time just as fast as one byte. So if you keep the bytes divisible by four, you get the best performance. Other considerations for size include hardware limitations. Some 3D accelerator cards on the market can only support an image up to 256 × 256 pixels in size.

Bitmap Color Depth

The color depth of a bitmap works just like the screen display does. So a 24-bit bitmap has 16.8 million colors, and an 8-bit bitmap has 256 possible colors. Color depth also carries memory considerations along with it. An image that is 100 × 100 at 24 bits takes up 100 × 100 × 4 bytes of memory. Roughly 40KB might not sound like much, but when you consider that games have thousands of images in them, the numbers add up quickly. For a quick look at a bitmap, take a peek at Figure 2.10.

Bitmap Color Key

The color key for a bitmap represents the color in the image that does not appear when the bitmap is displayed. This setting is useful when you want to see through parts of an image. When you use a color key with a bitmap, you are using

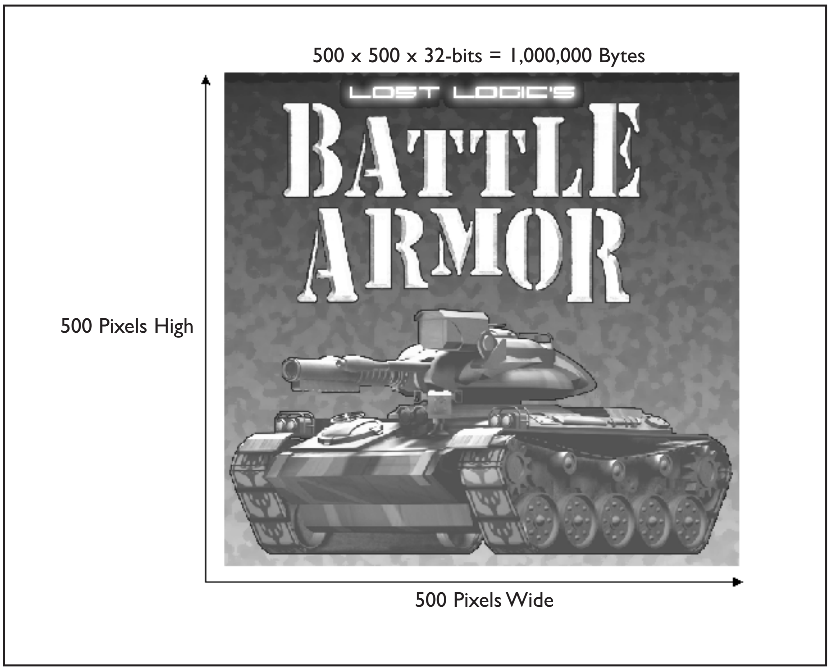


Figure 2.10

A simple bitmap

transparency. If you did not use a color key, then every bitmap would be square and block out any background. Take a look at Figure 2.11 to see the difference between a bitmap that uses transparency and one that does not.

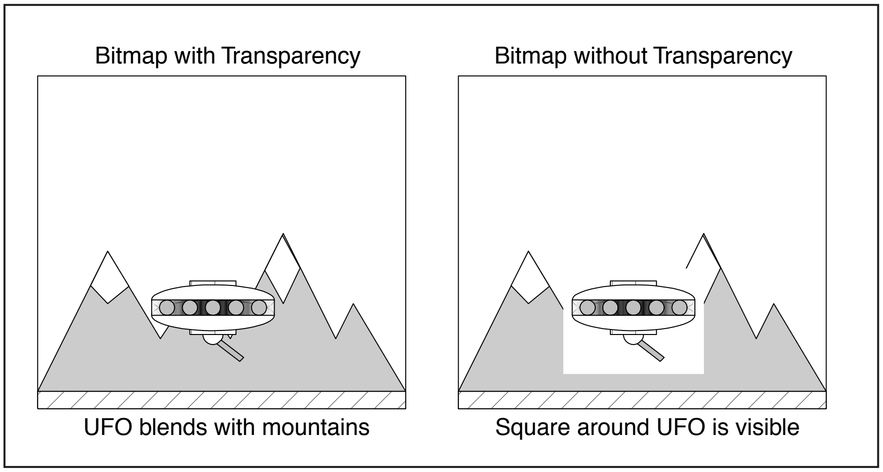

Bitmap with Transparency
Bitmap without Transparency
UFO blends with mountains
Square around UFO is visible

Figure 2.11

A bitmap with and without transparency

Backgrounds and Layering

The idea of color keying brings me to the next subject, backgrounds. Backgrounds in video games are really just large bitmaps. Sometimes, backgrounds even consist of several small bitmaps. Take, for instance, a side-scrolling shooter-type game like the old R-Type series. The background for that game is a star field or an underground cavern. Backgrounds get their name because they are in back of all the other objects in the game. Anything else displayed in-game appears in front of the background image.

When you display your spaceship bitmap on screen, it is pasted on top of a background image. This is the first rule of composition in game development. First, you display your background, and then you display foreground objects such as the player's object.

Whenever you display something on screen in a video game, it layers the graphic onto the display. Bitmaps that are drawn first appear below images that are drawn last. This is an important fact to remember when designing your screen layouts. Careful planning must ensure that everything turns out the way you want it to. Let me show you a simple example of layering.

Imagine that you are designing a game like the old SNES version of *Zelda*. In that game, you had background, middle-ground, and foreground images. Look at the background image in Figure 2.12.

The background image doesn't offer much to see, so it is time to add some middle-ground pieces. For the middle, I added a player. Check out the resulting composite image in Figure 2.13.

Well, so far this doesn't look too exciting. One thing that would spice it up is a graphic that partially covers the hero. Now I am going to paste down a box in front

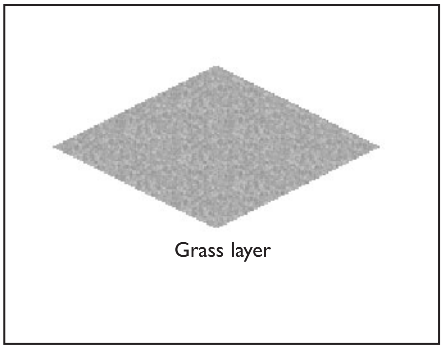


Figure 2.12

A background image in a simple RPG

An *RPG* is a role-playing game. In these types of games, people play the role of a character by acting out what they want to do in the game.

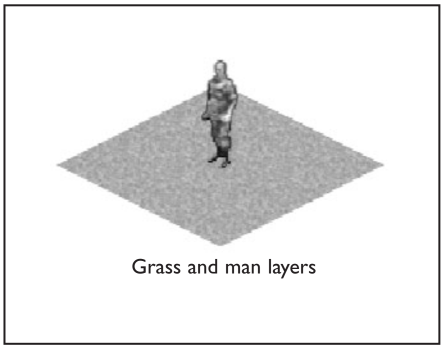


Figure 2.13

A background image with a character layered on top of it

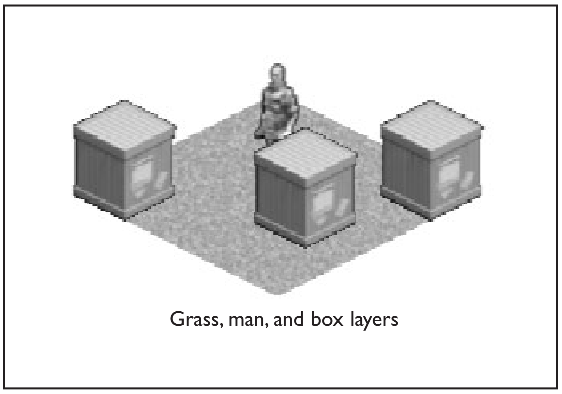


Figure 2.14

A three-layer composite image

of him, which gives the illusion of another dimension within the image. Look at Figure 2.14 to see the composite this creates.

Notice how your character is partially covered by one of the boxes? This is not possible if the boxes are drawn in the background image. If they were, you would see the character appear above the boxes. By creating a foreground layer, I add to the depth of the game. Without the foreground layer, everything would look too flat.

Considering that this is such a simple example, you can probably get an idea of how complicated graphics can get with advanced games. Don't let this be a deterrent to you, though. Everyone has to start somewhere!

Animation and Frames

When you see the word *animation*, you might think of a cartoon or movie. However, in the video-game world, an animation refers to a multi-image bitmap. To provide the illusion of animation in game programming, you use multi-image bitmaps. This means that you have different images for the same game object. Take, for instance, one of the units in *Starcraft*. When you move a unit, it might rotate. When it rotates, you are actually seeing different bitmaps in succession to give the illusion of rotation.

Each image in an animation is called a *frame*. Most fluff animations in video games consist of eight or fewer frames. They use this small number because of memory considerations. Remember the formula for bitmap memory usage? Well, for an animated bitmap, it gets even worse. Take, for example, a 24-bit 64 × 64 animated bitmap of an explosion with 8 frames. It takes up 64 × 64 × 4 × 8 bytes of memory. That's more than 131,000 bytes! And that is just one explosion.

In Figure 2.15, you can see how several images can give the illusion of movement or animation.

TIP

Before you write a game, be sure to calculate approximately how much memory the graphics will require. A good way to figure out what is acceptable is to look at some popular game requirements. If your game will require more memory than current titles in production, you are limiting your audience. Many more players have low-powered machines than monster computers. The optimal situation is to work on a closed system such as the Xbox. Then, you know how much memory you have to work with.

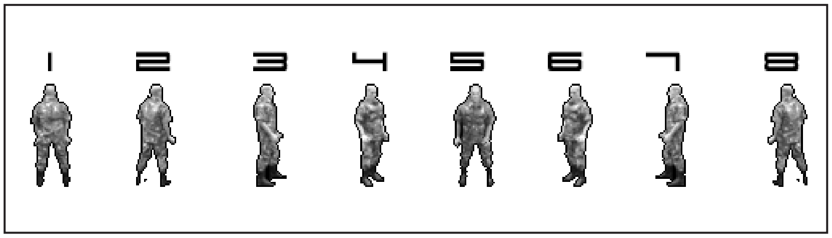


Figure 2.15

Frames used to create an animation

Blitting

Wow, that's a strange word, eh? *Blitting* is what occurs when you copy a bitmap to the display or to another bitmap. Most of the time, this term only applies to 2D graphics. So if you are working with 3D graphics, don't expect to do any blitting. Microsoft's DirectX 8 has even gone so far as to remove 2D hardware blitting from its SDK. This brings me to another point, hardware versus software.

Hardware Versus Software Blitting

Most video cards today support hardware-level operations. What this means is that the video card itself contains one or many processors. These processors can speed up operations such as blitting. Newer cards have 3D accelerators in addition to 2D accelerators. These cards can accelerate all sorts of operations such as lighting, alpha blending, and transformation. Don't worry if you don't know what all of this means yet; you soon will.

Some blitting functions take advantage of the hardware on the video adapter. If you use video hardware, your blit operations run much faster. Whenever you have to resort to software blitting, you'll notice a huge decrease in speed. So, always use hardware acceleration wherever possible.

Common Blitting Parameters

Most SDKs that support blitting give you some common options when doing so. Here are some common parameters you might expect to see.

- Target
- Source X
- Source Y
- Destination X
- Destination Y
- Width
- Height

The first parameter, the target, expects the object to which you are trying to blit. This can be either the background or another bitmap. In many instances, you create new bitmaps out of several other ones. This is why the target can change from blit to blit. You are not always blitting to the background.

The second parameter, the source X, is asking for the starting horizontal point within your bitmap from which to blit. This lets you blit portions of a bitmap to another one. The third parameter, the source Y, is asking for the starting vertical point within your bitmap from which to blit. The destination X and Y parameters determine where within the destination you want to blit to. This allows you to paste your bitmap anywhere within the target area. The width and height parameters let you specify how much of the source bitmap you want to copy. Perhaps you want to copy only a portion of it, or maybe you want to copy the entire thing—it is up to you.

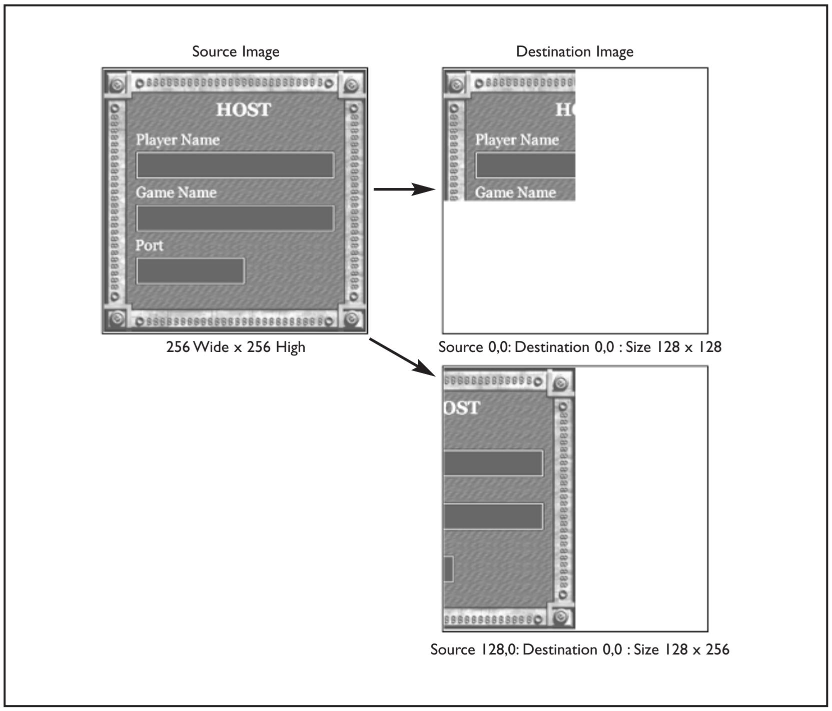

Source Image
Destination Image
HOST
Player Name
Game Name
Port
256 Wide x 256 High
Source 0,0: Destination 0,0 : Size 128 x 128
Source 128,0: Destination 0,0 : Size 128 x 256

Figure 2.16

Different blitting parameters and their effects

These are just a few of the common parameters you can run across. To see some options in action, take a look at Figure 2.16.

Collision Detection

Generally, when your airbag goes off, you know you have collided with something. To detect collisions in video games, you use the `iCheckAirBag()` function. Okay, seriously, you use collision detection in video games to determine when two objects have hit one another. This information is useful when you're trying to figure out whether a projectile struck the player or the player blasted the bejeezus out of something else.

A common question from most new game developers concerns how you detect collisions between bitmaps. Well, it's not all that difficult. All you have to do is think of bitmaps as rectangles, which they really are. Once you realize that bitmaps are just rectangles, all you have to do is process a rectangle intersection algorithm. Ooh, big words, eh? To put it in layman's terms, you check whether the rectangles overlap. Take a look at Figure 2.17 to see how the bitmaps overlap one another.

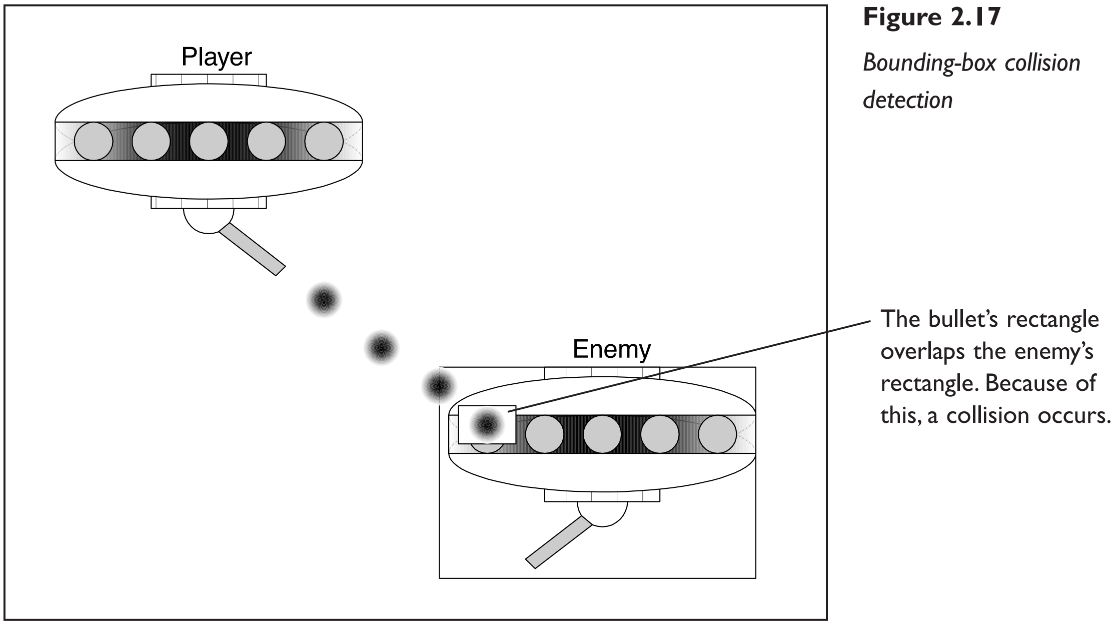

Player
Enemy
The bullet's rectangle overlaps the enemy's rectangle. Because of this, a collision occurs.

Figure 2.17

Bounding-box collision detection

Notice that the player's bullet is overlapping the enemy. Because the bullet is just a rectangle, you can check whether it overlaps by running the following formula:

```
If absolute_value( bullet_center - enemy_center )
  <= enemy_size then COLLISION!
```

This formula checks whether the center of the bullet is within range of the enemy's center point. This type of collision detection is called the bounding-box method. Basically, you are putting a box around every object in the game and checking whether something enters the box. This isn't the most accurate form of collision detection, but it works for most situations.

What about the Third Dimension?

Hey, I can't very well leave out 3D graphics now, can I? Most games on the market today use 3D engines. The third dimension adds all sorts of facets to game development. No longer can you get by with mediocre math skills; now you must understand such things as linear algebra. Okay, maybe it's not that bad—you really only have to understand a few things to get by.

What Is 3D?

2D games have a two-coordinate system, the x-axis and the y-axis. 3D games have a three-coordinate system, the x-axis, the y-axis, and the z-axis. Take a look at these two systems compared in Figure 2.18.

As you can see in Figure 2.18, the 3D system appears to be going into the page. The x-axis refers to left and right movement, the y-axis refers to up and down movement, and the z-axis refers to in and out movement.

Projection

We all know that most computer displays really have only two dimensions. Because this is the case, you are really just simulating the third dimension in 3D games. To simulate 3D, you must project a 3D image onto a 2D surface. This allows you to create a flat image that appears to have depth. You see this effect every day with what is called *perspective*. Take a peek at Figure 2.19 to see an example of image projection.

Vertices

Objects in 3D consist of several points. Each point has a coordinate value for the x-axis, y-axis, and z-axis. These points are collectively called *vertices*, with each point being called a *vertex*. You can think of vertices as points in 3D space. When two vertices connect to one another, they form an edge, or line. When three vertices connect, they form a triangle or a *polygon*. Figure 2.20 shows three vertices connecting to form a polygon.

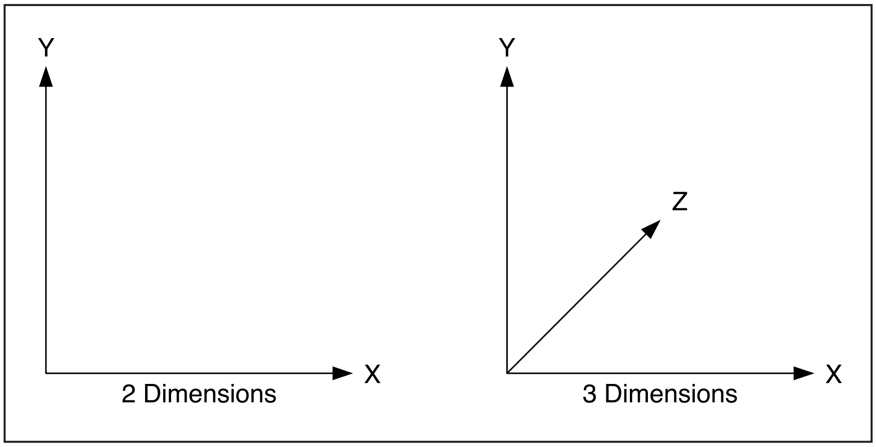


Figure 2.18

2D coordinates versus 3D coordinates

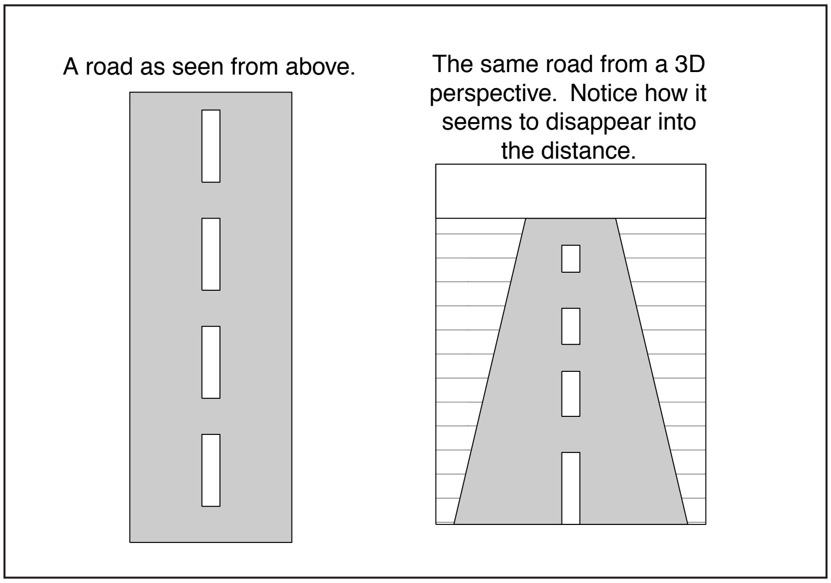

A road as seen from above.
The same road from a 3D perspective. Notice how it seems to disappear into the distance.

Figure 2.19

A 3D scene projected onto a 2D surface

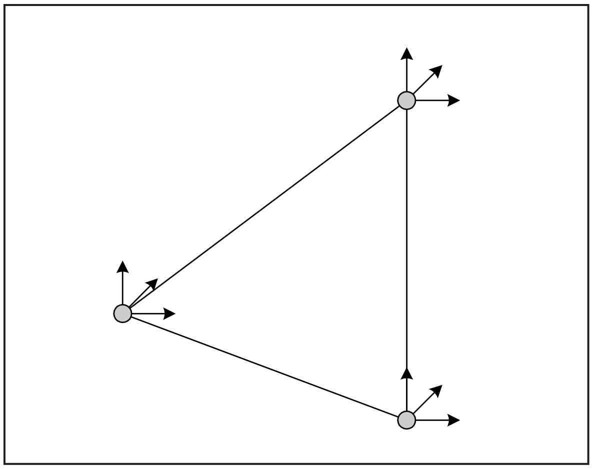


Figure 2.20

Three vertices connecting to form a polygon

Polygons

The polygons that are formed when vertices connect allow you to view the object in *solid* form. If the object had no polygons, you would see only a cloud of points. Now you know why a game system or video card indicates how many polygons it

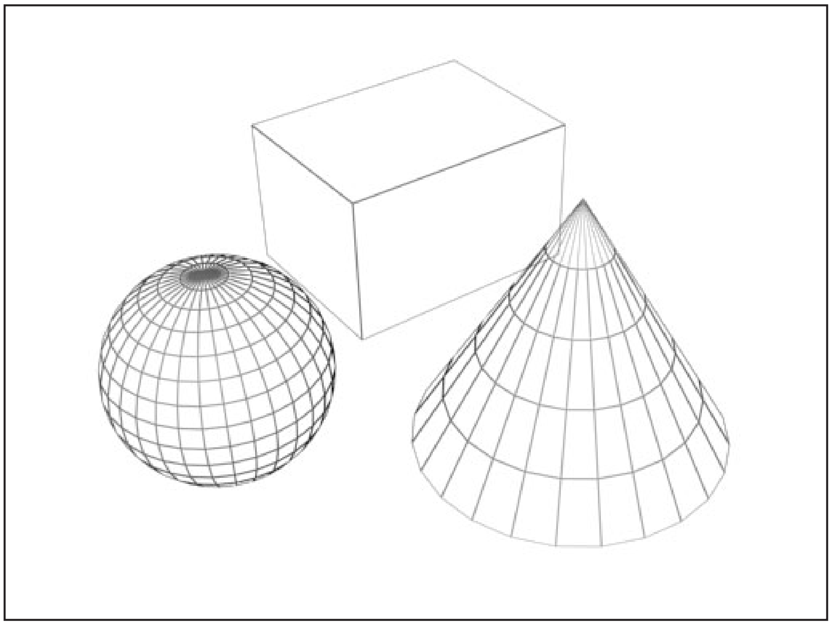


Figure 2.21

Different objects created with triangles

can handle. The great thing about using triangles to make polygons is that you can create anything with this method. Check out Figure 2.21 to see how triangles can form several different objects.

Rendering

Displaying a 3D object on the user's screen is called *rendering*. All this means is that you are converting the polygons into pixels or *texels*. Rendering does involve a lot of steps and operations, but you don't need to worry about that right now. Just remember that rendering means creating images of your 3D objects.

Texels **are textured pixels. Some video cards refer to how many texels they can render at once, which means how many textured pixels they can display. Nvidia, the popular video card maker, coined this term.**

Solid Objects

Now it's time for some solid advice, pun intended. Okay, that was a bad joke, but hey, it's late and I'm tired! Excuses aside, solid objects can only be rendered when

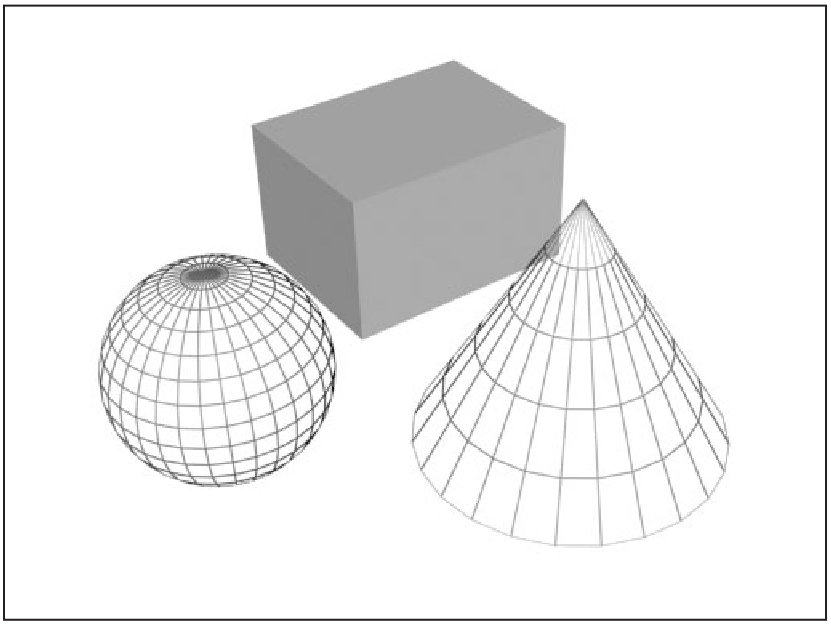


Figure 2.22

Solid and wire-frame rendering

they have *normals*. What is a normal? Well, a normal tells a *face* how to be shaded. *Faces* are just three-sided polygons, or triangles.

To keep things simple, I am going to say that normals define rendering surfaces for polygons. With normals, you can view your objects as solid entities. If you don't render your objects in solid mode, they appear as wire-frame objects. The images in Figure 2.22 demonstrate the difference between solid and wire-frame rendering.

Textures

3D objects would look pretty boring without textures. Textures are bitmaps that are wrapped around 3D objects. For example, texture-mapped cubes could represent the keys on your keyboard. Each cube's texture map would be a bitmap of a different key. Take a look at Figure 2.23 to see a texture-mapped object.

That's it for the nickel tour of 3D graphics. Later, I'll show you how to create an entire 3D game. When the time comes, you will learn more about 3D graphics than you ever wanted to know.

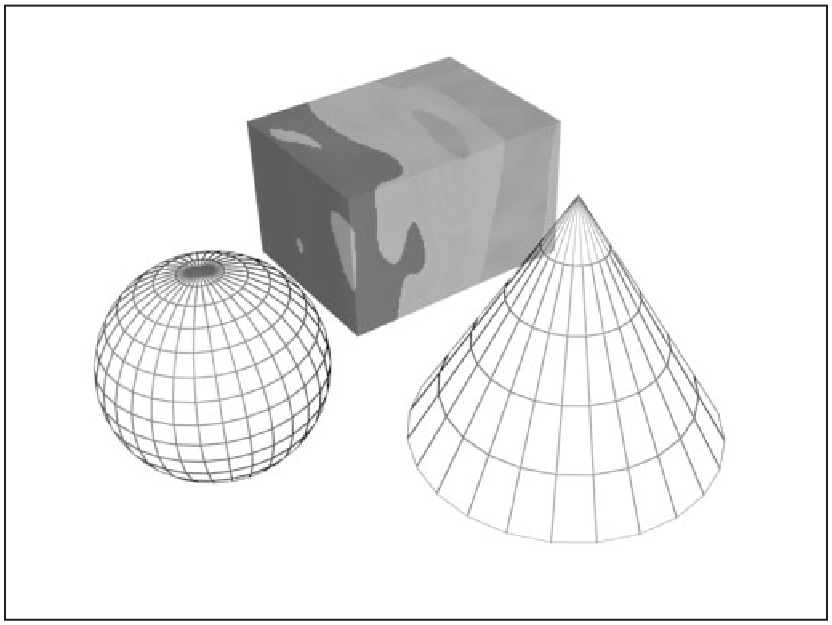


Figure 2.23

Texture mapping

Review

Well, here we are at the end of Chapter 2 already. I covered most of the important aspects of video game architecture, so all that is left is actual implementation. But before I take you to the next chapter, which covers networking, take a minute to review some key points.

- Game loops are logical loops in your program where reoccurring events happen. Tasks such as input gathering and rendering are handled within game loops.
- When you initialize a game, you must set up the video, sound, input, and networking interfaces for it.
- When initializing the player's video display, you must take into consideration the screen resolution and color depth requirements for the game.
- You can create your own game sounds by sampling real world recordings. Always be sure to use an appropriate sample rate so that the digitized sounds keep their clarity.

- There are two main types of multiplayer games, peer-to-peer and client/server. The peer-to-peer method is good for a small number of players, while the client/server model is good for massively multiplayer games.
- 3D games use polygons to create their graphics. The final renderings are made up of pixels and texels.

CHAPTER 3

Introduction to Network Communication

There is no better way to start learning network game programming than to learn the basics of how computers communicate with one another. In this chapter, I'll give you a mid-level view of how networks are built and how the computers within them communicate. I'll cover the following topics.

- Network theory
- Game network protocols
- Ports and firewalls
- Types of game connections
- Hardware setup
- Data packets

Network Theory

Well, here you are, standing at the gate of multiplayer game development. Before you can enter, you need to better understand what you are dealing with. You probably already know what a network is, but the information you need to know is how a network affects game development. Following are some definitions of a network (*The American Heritage® Dictionary of the English Language*, 3d ed., s.v. "network.").

- An openwork fabric or structure in which cords, threads, or wires cross at regular intervals
- A system of lines or channels that cross or interconnect: *a network of railroads*
- A complex, interconnected group or system: *an espionage network*
- An extended group of people with similar interests or concerns who interact and remain in informal contact for mutual assistance or support
- A chain of radio or television broadcasting stations linked by wire or microwave relay
- A group or system of electric components and connecting circuitry designed to function in a specific manner
- A system of computers interconnected by telephone wires or other means to share information

For our purposes, you get to use the last definition. So, there you have it; a network is nothing more than computers connected by some means to share information. Okay, what does that mean? Well, for network game development, you must figure out how to make your games talk to one another across some connection. The problem in the past was the lack of a standard way for computers to communicate. Can you imagine having to write communication code for several varieties of networks? If you are a veteran to the industry, you might know that games had no standard way to communicate until recently.

Standard Methods for Game Communication

The most widely used method for game communication is through the use of sockets. Sockets have been around for a while and have been accepted by the game-development community as a fitting method. A more recent addition to the game-development community is DirectPlay. For now, I'll talk about sockets first, and then I'll go into DirectPlay.

Sockets

Sockets are commonly referred to as connections through a network, be it a LAN or the Internet. The most common sockets library you hear about when dealing with Windows is Winsock. Winsock is nothing more than a library provided by Microsoft to allow programmers to use sockets within their programs. All of the sockets-based sample programs covered in this book use the Winsock library. When using sockets, you follow a certain protocol depending on what you are trying to achieve. For game-development purposes, I prefer to use TCP/IP (*Transmission Control Protocol/Internet Protocol*). TCP/IP is a sockets protocol that guarantees packet delivery. When computers have the TCP/IP protocol installed, they also have an IP address.

Inside IP Addresses

IP addresses identify individual computers over a LAN or the Internet. An IP address consists of four numbers separated by periods. The format looks like this:

```
100.100.100.100
```

Each of the four numbers ranges in value from 0 to 255, so there can be more than 256^4 unique addresses in the world. That is more than four billion addresses—

4,294,967,296 to be exact! The four billion or so unique addresses only limit computers that have an address exposed to the Internet. This does not even include addresses of computers within closed networks. Every private network is allowed to have this many addresses as well. So, as you can see, there will be no shortage of IP addresses to go around for a long time. Refer to Figure 3.1 for a diagram of private and public IP addresses.

NOTE

The range of possible IP addresses is actually being expanded to handle more than the current limit of 256^4 addresses.

The interesting part is that a computer in a private network can have the same address as a computer in another private network. Take a look at Figure 3.1, and you'll notice that the computers in the private network have the same addresses as those in the Internet. You can think of them as apartment complexes. An apartment complex has a main address for the main office, but the individual apartments have addresses such as A-1 or C-2. Well, another apartment in a

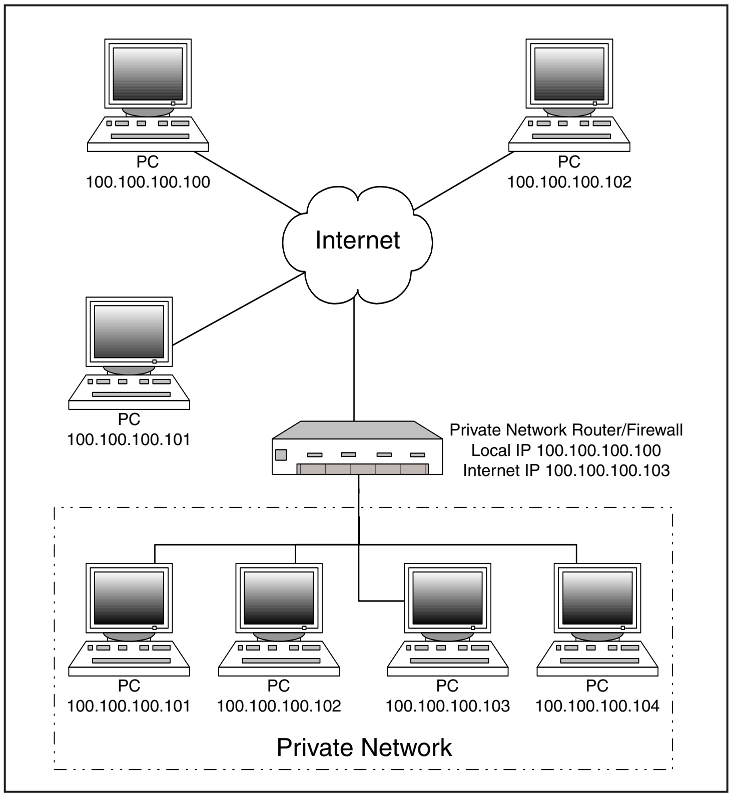


Figure 3.1

Diagram of public and private IP addresses totally different complex can also have an address of A-1 or C-2. In this analogy, you can refer to the apartments' main offices as the routers connected to the firewalls.

Firewall Primer

Firewalls provide a layer of protection for computers on private networks that want Internet access. Most of the time when you operate a network of computers, you do not want just anyone to have access to your systems. You can restrict access by using firewall hardware or software. Most firewalls are hardware-based and resemble network hubs. A firewall works by hiding the IP addresses of computers attached to it in the private network. Computers in the private network can see out into the Internet through the firewall, but computers on the Internet cannot see behind the firewall. This provides a two-way mirror approach to security. One problem this presents for developers is that servers might not work if you are writing a standard peer-to-peer game. For example, take a look at Figure 3.2.

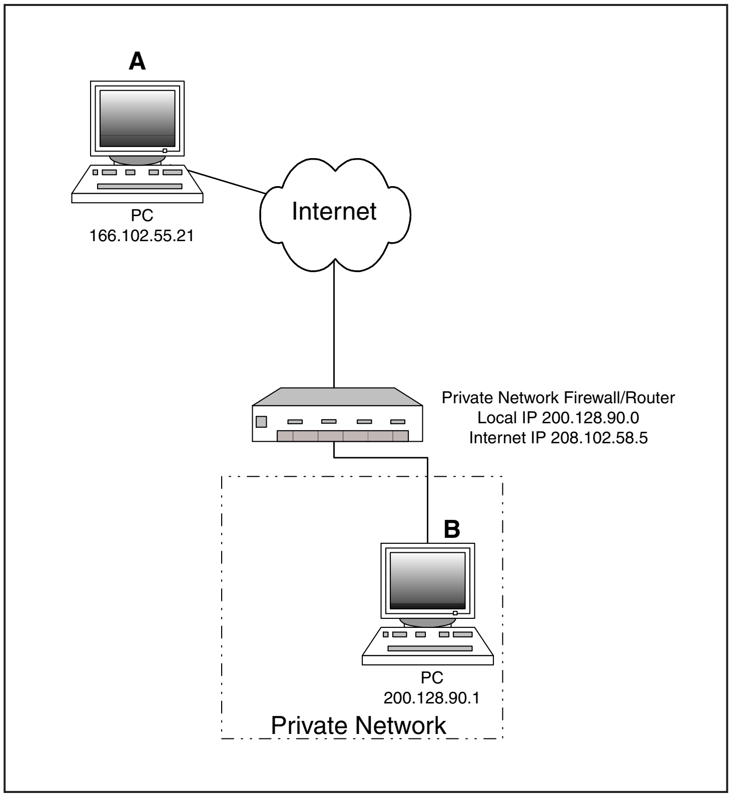


Figure 3.2

Firewall access

In this example, Computer B can see Computer A, but Computer A cannot see Computer B. The only thing Computer A can see is the firewall in front of B. For a client to connect to a server, it must be able to see the server's IP address. In the example, Computer B's IP address is masked, so that means Computer B cannot be a server. On the other hand, Computer A could be the server and Computer B would be allowed to connect to it. A server does not necessarily have to know the client's IP address.

CAUTION

Some game services, such as Microsoft's Gaming Zone, require that you are not entirely blocked behind a firewall. If you are blocked behind a firewall, you might find that you cannot play the games. To alleviate this, you must have a range of required ports opened.

There is light at the end of the tunnel, though. You can have a server behind a firewall if you open some holes in the wall. Holes in a firewall allow traffic to come in through them. These holes are commonly referred to as ports.

Ports and More about Firewalls

Ports are just channels in which network traffic is delivered within an IP address. The usual ranges of ports to use are from 1,000 to 10,000. You don't use those numbered 1 to 1,000 because they are used by standard protocols such as FTP and HTTP. For example, HTTP requests come in on port 40. Most firewalls allow you to open individual ports or ranges of ports. If you want to send a transmission to a port within an IP address from your Web browser, you type in the IP address followed by a colon and then the port number. For example, to send a request to IP address 100.100.100.100 through port 40, you would use the following address:

```
100.100.100.100:40
```

By specifying a port number, you drill down into the IP address where you want to send the packet. By doing this, you can go through the hole opened by the firewall. Refer to Figure 3.3 for an example.

In this figure, you see Computer A connecting to Computer B through the firewall. Specifying the open port number 6000 achieves this. When the firewall receives the request from Computer A, it translates the request to an internal IP address, a process commonly referred to as NAT (*Name Address Translation*). The firewall maintains a list of ports and to what IP address the ports are translated. To open a port in your firewall, you make an entry into the NAT table, specifying the port

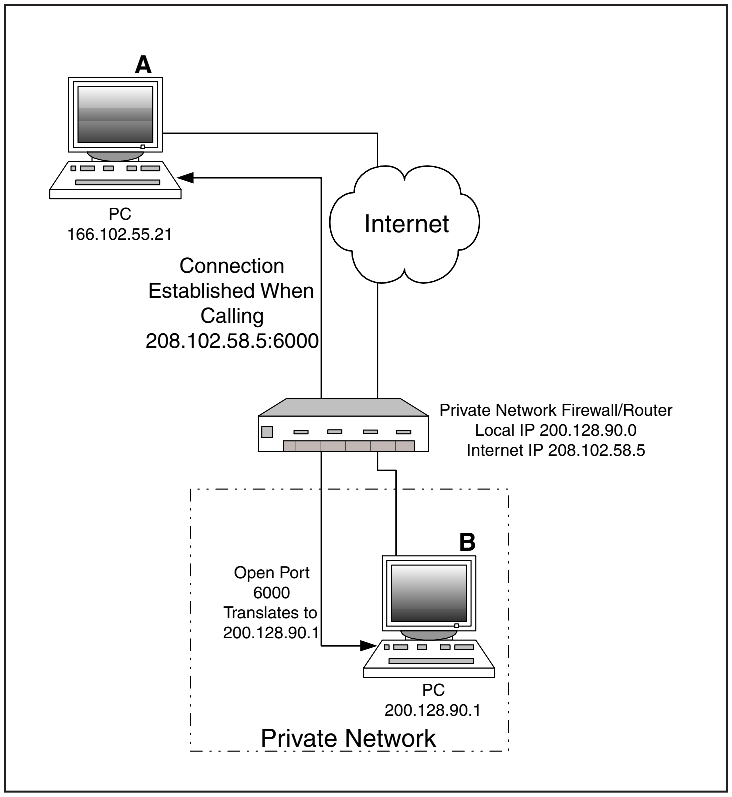


Figure 3.3

Firewall access through open ports

number and IP address. When requests come into the firewall, it checks whether the specified port is open. If the port is open, the traffic is routed to the internal IP address in the table.

Regardless of whether you are behind a firewall, you have ports on your system. When you are running a server, it must specify a port to accept connection requests.

Some games allow you to specify the server's port number, but others do not. For an example of this, you can look at Red Storm Entertainment's game, *Rogue Spear.* The connection menu in *Rogue Spear* asks you which port number the server is listening on. When a game does not allow you to specify a port, normally the reason is that the program hard-codes specific ports. The best practice when writing your own games is to let the client and server programs specify port numbers. This practice keeps users from having to change their configurations to fit your programs.

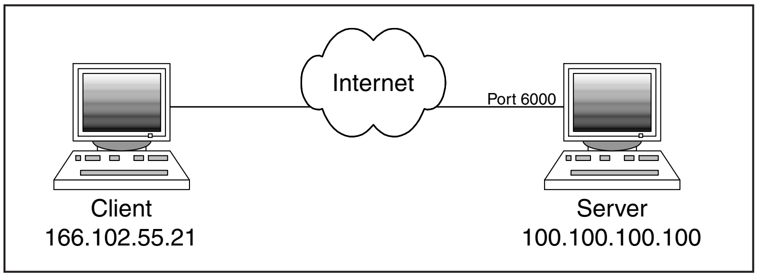


Figure 3.4

Connecting computers that are not behind firewalls

The less amount of pain the customer has to endure, the better. Refer to Figure 3.4 to see how two computers not behind firewalls would connect.

The illustration in Figure 3.4 looks a bit simpler than firewall examples because there is no hardware between the server and the client. This is most often the case in games where both the client and server are using modem connections. If you dial into the Internet via an ISP (*Internet Service Provider*), then you most likely have no protection between you and the Internet. In this case, you can run a server without having to open ports to incoming traffic because the ports are already open. Okay, enough about ports; let's move on to something else.

Overview of DirectPlay

DirectPlay, a part of DirectX, is produced by Microsoft and provides a layer of abstraction away from connections. With DirectPlay, your program doesn't have to be aware of the connection present, whether it is modem to modem, Internet, serial cable, or something completely new. DirectPlay is a good generic API in this manner, but it does have some shortcomings because it abstracts too much in some cases.

For instance, if you are developing an MMORPG, you might not use DirectPlay. The reason is that such a large-scale project requires multiple servers, and the project requires absolute low-level control of the communication streams. DirectPlay hides a lot of the functionality and might not be suitable. Another point to mention is that some MMORPG servers are UNIX-based, and DirectPlay works only in Windows environments. Lucky for you, though, Microsoft just rewrote the core of DirectPlay to better handle MMORPGs; time will tell whether it was fixed enough. The sample programs you will work with expect DirectPlay to use the TCP/IP protocol. For information about TCP/IP and sockets, refer back to the "Sockets" section earlier in this chapter.

Types of Game Connections

The two basic types of game connections used primarily in game development are peer-to-peer and client/server, both of which are supported by DirectPlay. Both types have good and bad aspects, so you should weigh which method is best for the game you are writing.

Peer-to-Peer Game Connections

The earliest method used is peer-to-peer game connections. This type of connection puts the burden of keeping game state on each client participating in the game session. *Game state* refers to the complete status of a player's game environment at any given time. *Game state changes* are things that alter a player's environment, such as movement and special events. For example, if Player 1's tank were moving forward and then turned left, the game state would change. When a player's game state changes in peer-to-peer games, all other players must be informed of the change. So from the previous example, when the player turned left, every player in the game would have to be informed that it was turning.

Peer-to-peer games start up with all the players in the game knowledgeable about all the other players. One advantage to this is that if a player gets disconnected, the game continues. Because there is no reliance on a central server, the game is safe unless everyone is disconnected. Refer to Figure 3.5 to see the architecture of a peer-to-peer network.

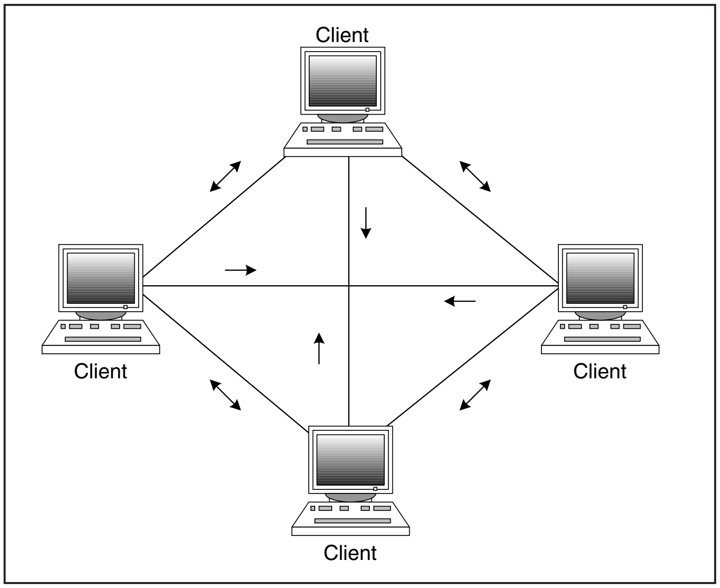


Figure 3.5

Peer-to-peer network architecture

One thing you might notice from Figure 3.5 is a lot of cross-traffic between clients. This is due to every client sending state changes to every other client. Eventually, this becomes cumbersome when you have many players in a game. In fact, games such as *EverQuest* and *Asheron's Call* do not work with this method. The process of thousands of players communicating with each other is impossible given today's technology. Most games that require many players (more than four) do not use the peer-to-peer method for game communication. One of the nice things about peer-to-peer games, though, is that you do not need a server with a fast Internet connection to play the game.

Even though it is not called client/server communication, peer-to-peer games do have a server. Instead of a traditional server, the server in the peer-to-peer method transfers only player connection information. When a new player joins a peer game, he requests information about all the players connected from the server. The game server then responds with the IP addresses and other information about all players in the game. Here is a sample sequence for requesting player information.

1. Player A starts a multiplayer game and takes an ID number of 0.
2. Player B joins A's game and requests a new ID number and information about players.
3. Player A sends B the new number of 1. Player A sends information about himself and nothing else because no other players are present.
4. Player C joins A's game and requests a new ID number.
5. Player A sends C the new number of 2. Player A sends information about himself and Player B.
6. Player C loops through the list of players and sends information about himself. In doing so, Player B is informed that Player C is now in the game.

If the designated server in a peer-to-peer game is disconnected or leaves the game session, then normally the next player in "line" becomes the new server. So from the previous example, if Player A disconnects, then Player B becomes the new server. Player B then functions as the repository of player information for new connections. Even though some of this explanation might look complicated, client/server games are usually much more difficult to design.

Take a look at the pros and cons of peer-to-peer games in Table 3.1. One item that I mention is the security risk. In peer-to-peer games, players are dependent on other players for accurate information. This presents a major game security, or *integrity*, risk because clients have to trust other players with their information. Many games have fallen prey to this risk, such as Blizzard's Diablo game series.

Table 3.1 Pros and Cons of Peer-to-Peer Game Design

Pros	Cons
Fault tolerant	Limited maximum number of players
Easiest to implement	Unnecessary network traffic
Fast server not needed	High security risk

When I show you how to develop a peer-to-peer game later, I'll show you some tips on securing these types of games. The main thing to remember is that you need to evaluate all methods of network design on a per-game basis.

Well, that's about it for peer-to-peer theory; let's move on to the client/server model.

Client/Server Game Connections

The client/server model of game development is becoming more prevalent as a standard for game communication. Not only does the client/server model work well for games, but it is also used by many commercial industries as a replacement for mainframe architecture. One good thing about this is that the skills you learn while developing client/server games can carry over to most industries. It never hurts to have a good item on the ol' resume!

The client/server model differs from the peer-to-peer model in that a single server is responsible for keeping game state. The server maintains all communication between clients and keeps clients informed about the entire game state. Clients in this network model send data only to the server. Refer to Figure 3.6 to see the architecture of a client/server environment.

Because servers maintain state for all clients, it is best if they are well powered and have high-speed Internet connections such as ISDN, DSL, T-1, or better. Here is a sample server setup for a first-person "shooter" style game, such as Dynamic's *Tribes 2.*

- Windows 2000 OS
- Pentium III, 1 gigahertz (GHz) or faster
- 256MB of RAM
- T-1 Internet connection

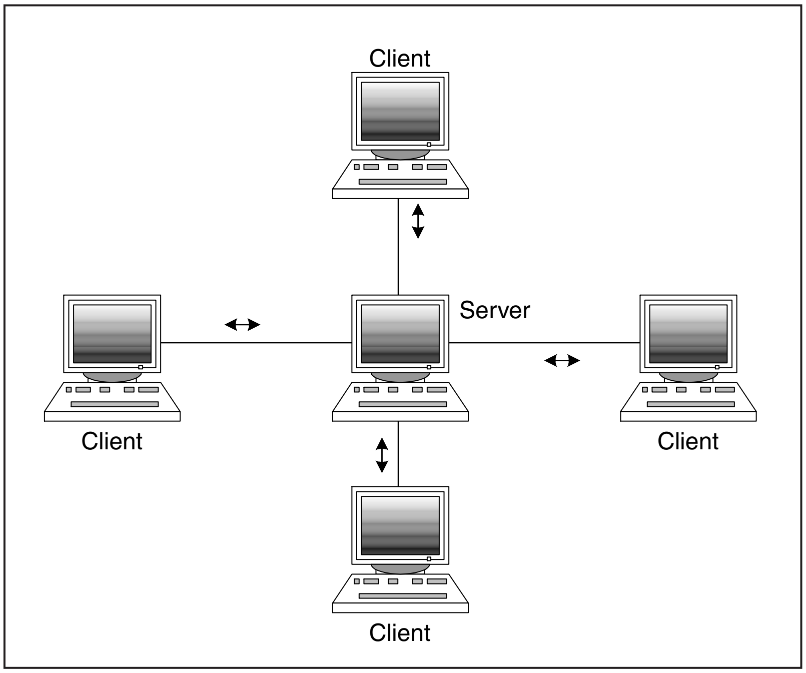


Figure 3.6

Client/server network architecture

The default hard-drive storage required is usually plenty for games of this type. Because hosted games such as Dynamic's *Tribes 2* are not persistent world servers, they do not require a lot of storage space. For supporting MMORPG-style games, the requirements for servers change drastically. MMORPG servers require immense amounts of hardware and bandwidth. In fact, many games of this genre incorporate multiple servers. Origin's *Ultima Online* game utilizes a cluster of servers that divide the game world into boundaries. When players cross server boundaries, they are handed off to a different server. This helps with load balancing. Here is a sample server setup for a generic MMORPG-style game.

- Windows 2000 Server or UNIX OS
- Quad Pentium IIIs, 1 GHz or faster
- 2GB of RAM
- T-3 Internet connection or faster

One item to note is the operating system. Many massive online games use UNIX servers because they are generally better suited for the task. You will hear many different opinions about this. Some people say Windows is better; others say UNIX or

Linux is better. It really depends on the task at hand, and the best thing to do is try different OS solutions until you find the optimal one for your game. The corporate solutions I design use a mix of UNIX and Windows. There are pros and cons to each.

Now I want to talk about how game state is handled in the client/server environment. Because servers are responsible for the game state, they process quite a bit of data. Refer to Figure 3.7 for a sample communication between a client and a server in a shooter-style game. In Figure 3.7, you can see a single client (Client A) and some other unnamed clients. For this example, Client A is a tank moving north. Client A then decides to turn right. Because this action changes the overall game state, the server must be notified. Here are the events that transpire with this action.

1. Client A sends an action packet to the server stating that the client's tank is turning right.
2. The server receives the action packet and acknowledges the reception to Client A.
3. The server updates its game state with the new information.
4. The server sends a packet to all other connected clients, informing them that Client A is turning right.
5. The clients other than A update their games with the information that Client A's tank is turning right.

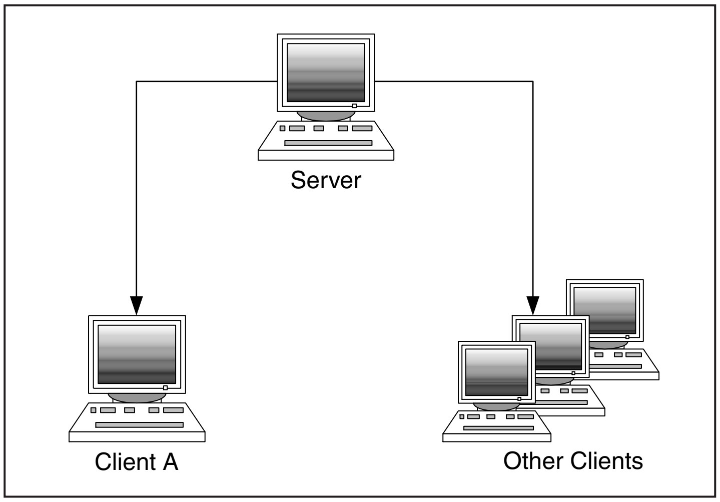


Figure 3.7

Sample communication between a client and a server

In this example, you see that the server must relay game information to all players involved. This is why the server requires a high-speed Internet connection. This type of communication does make life a little simpler for synchronization, though, because the server is aware at all times what state each client is in. If a client starts lagging behind, the server can slow down the entire game until the slow player catches up. If the slow player never catches up, the server might disconnect it.

When you are playing a game over the Internet, abnormal delays that cause jerky movement or missed actions are commonly referred to as *lagging*. A *lag* denotes the delay between a client and a server. If you are experiencing lag, you are getting a slow response from the server, or you are transmitting slowly to the server.

The main difference between MMORPG servers and other game servers is that MMORPG servers do more than just transmit data between clients. These servers also approve client actions and store all data for the game. Because these types of servers have to approve client actions, the games are generally much more secure than traditional servers or peer-to-peer games. The MMORPG server checks the client requests and rejects the requests if the actions are invalid. Now take a look at the pros and cons for utilizing the client/server model in Table 3.2.

Personally, I prefer the client/server model, but that might be due to the fact that I work with client/server applications daily. If you want to design MMORPG-style games, then the client/server model is the way to go; otherwise, try out the methods and choose for yourself.

Table 3.2 Pros and Cons of Client/Server Game Design

Pros	Cons
Improved security	Expensive hardware
Easier to synchronize game state	High-speed Internet connection required
Lower network traffic	Not fault tolerant
Can support many players	

Game Communication Hardware

Now that I have covered the methods of game communication, I want to talk about what it means from the hardware level to get set up for network games. First I'm going to cover how some network hardware works. Then, I'll discuss how to set up your environment to develop network-ready games. Last, I'll talk a little about the hardware design of game servers.

Connecting to the Internet

One method common to multiplayer gaming is playing through the Internet. The majority of game players use modems to connect to the Internet. When you are designing your network game, be sure to keep this in mind. If you design your game to work well only under high-speed connections, then you'll leave most gamers out in the cold.

Modem Basics

Most modulators/demodulators, or *modems*, are internal and included with packaged computers. A computer's connection to the Internet through a modem is commonly referred to as a *dial-up connection*. Because the modem must make a connection before the user can go online, you must program your games to handle this event. Some current games on the market crash if you try to play over the Internet when you are not already connected. I have found that a good way to be prepared for this is to perform a connection test before your game really starts. If the computer is not connected to the Internet, then inform the user and ask whether he still wants to continue. This keeps the player from having to switch back to the desktop to connect to the Internet while in the middle of your program.

The standard speed modem you should develop for is the 56.6k modem. Many new technologies are faster, but they are not mainstream yet. The 56.6k modem has been around for a while, and even Sega's Dreamcast console gaming unit has one built in. The 56.6k refers to the transfer speed in bits per second. A 56.6k modem is capable of transferring 56,600 bits per seconds, or around 7,075 bytes per second. This might sound like enough, but imagine that most games today play at around 60 frames per second. If you take 7,075 bytes per second and divide that by 60, you find that only around 118 bytes can be transmitted per frame. You need a total bandwidth of 118 bytes per frame for both receiving and transmitting;

therefore, you really only have 118 divided by 2, or 59 bytes, available to send and receive per frame.

One common problem with modems is that the connection speed is never guaranteed. It is not uncommon for a modem user with a 56.6k modem to get a 28.8k connection. It really depends on the service the player uses and the current load on the Internet. Keep these restrictions in mind when you develop your network-ready games.

High-Speed Bandwidth from Home

Many home users have faster options. One of the oldest is ISDN, which comes in two flavors: 64k and 128k. Basically, ISDN connects just like a standard phone line. The phone company installs the line in your house, and then you connect to it with special hardware. ISDN lines are not compatible with standard phone equipment, so do not expect to use your phone on the same line. However, you can install a router for use with your ISDN line, which lets you connect a phone to it. ISDN can support up to two channels, each with 64k of bandwidth. You can use the two lines in tandem for a grand total of 128k. One nice advantage to an ISDN connection is that the 64k or 128k connection is guaranteed to be the full bandwidth. Unlike dial-up modems, the speed does not vary based on service providers or system load. The reason for the constant speed is that ISDN lines are digital, not analog. The main downfall of ISDN lines is the cost. They are typically much more expensive than other solutions. Refer to Figure 3.8 to see a typical setup for ISDN.

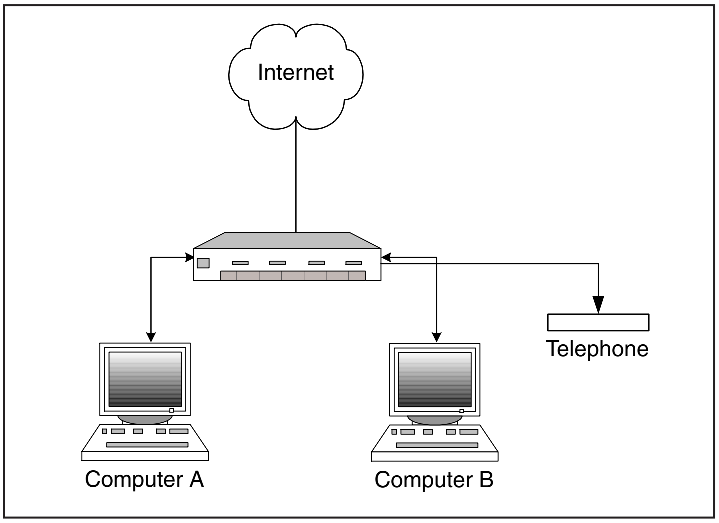


Figure 3.8

ISDN configured for two computers with a router

Another high-bandwidth option for home users is DSL (*Digital Subscriber Line*). DSL works on existing phone lines and only requires a special modem to utilize it. DSL can be very fast at a reasonable cost. The main downfall to DSL is that it currently works only for customers within limited range from phone switches or centers. This problem has kept DSL from the homes of many users. If you have access to DSL, I suggest you go with it if you are currently using something slower. DSL can handle the throughput to operate a server for most games other than MMORPGs.

One of the most controversial connection methods around is cable. Some cable companies now offer Internet services to their customers. Cable promises massive bandwidth, rivaling T-1 connections, for a reasonable price. Sometimes these promises are fulfilled, but other times they fall short. Cable connections use a special cable modem and a standard digital cable line. One problem with cable connections is that when you are connected to cable, your computer is visible to all other cable modem subscribers within the service. When you check your Network Neighborhood (or the equivalent), you actually see every computer that has a cable modem installed from your cable provider. If you are concerned about network security and do not feel comfortable battling hackers, then I suggest you stay away from cable modems. On the other hand, if you are familiar with firewalls and understand how to use them, cable modems are a good solution. Refer to Figure 3.9 for an illustrated view of a cable modem network.

Probably the most real problem with cable Internet is that the more users a system has registered, the slower your connection. The entire cable modem network shares a single resource pool. If many people take a lot of bandwidth, then others have less to share. Most cable companies regulate bandwidth in one fashion or another. If you can get cable modem access and are considering it, be sure to check what bandwidth restrictions are in place.

One of the most extreme solutions, mainly for people who do not have any other means of high-speed access, is satellite. A few companies offer satellite Internet access from your home. Hughes Satellite offers it through a system named DirecPC. To use this system, you must install a special satellite dish and a receiver card in your computer. You must have a modem on your computer with access to the Internet as well. Current satellite systems are only fast when receiving data, and all the data you want to transmit is handled over your modem connection. Most Internet users download the majority of their data, so this is not a problem. However, this is a problem for game enthusiasts because whenever you want to receive data, you have to transmit to a relay point before you ever get the data. This process takes quite a bit of time as far as games are concerned, and makes most

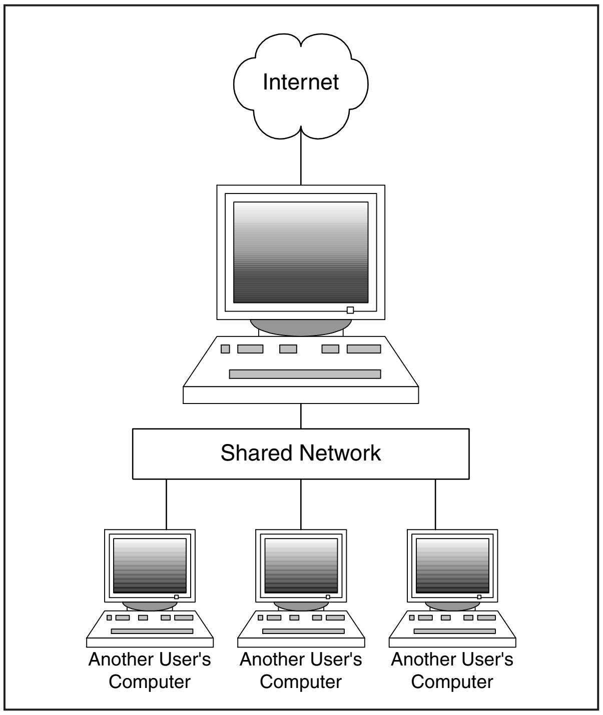


Figure 3.9

A sample cable modem network

games unplayable. This reason alone makes the satellite solution a poor one for game players. I have used the satellite solution before, and it was great for downloading (approximately 1MB of data received every 10 seconds), but it did not work at all for game playing. Bottom line: It's not for us gamers.

NOTE

The Hughes DirecPC satellite provider also limits the amount of bandwidth you can use based on how much data you have transmitted in a 24-hour period. Satellite systems also suffer the same fate as cable modems in that more users translates into less bandwidth for everyone.

Internet Service Providers

For all of these methods of connecting to the Internet, you need an ISP (*Internet Service Provider*). ISPs handle your connection to the Internet through their hardware. The majority of ISPs rent lines from the phone company to connect to the Internet's backbone. When you hear about the backbone of the Internet, the reference is to the super-fast lines used by phone companies for Internet traffic. Most people in the United States have access to an ISP, and the best place to find one is through your local phone book. Unless you do not have a mailbox, you have probably received a CD or disk from AOL (*America Online*). AOL provides Internet access to most of the country and has reasonable rates. If you are very new to the Internet, you might want to use it as an option. Personally, I always go through a more direct ISP for speed reasons. AOL can be slow sometimes due to its vast number of subscribers, a problem encountered by ISPs when they get too many subscribers. When an ISP has too many users, the equipment gets bogged down, and all users experience slow connections and bad lag. You might have to try many ISPs before you find one with consistently good connections. I would avoid any long-term contracts with an ISP unless you have tried it out for several months. Even then, I would question why a contract is necessary. Your savings might not be that great in the long run if your connection goes bad.

Dynamic and Fixed IP Addresses

When you establish a connection with your ISP, you are either given a new IP address or you are given the same one time and time again. If you receive a new IP address every time you connect, then your IP address is considered *dynamic*. If you always have the same IP address, then it is a *fixed* address.

CAUTION

To operate a proper persistent-world game, you need a fixed IP address; otherwise, players never know where to connect. Fixed addresses are generally much more expensive than dynamic ones, so keep this in mind when designing an MMORPG.

Local Area Networks

Let's talk a bit about LANs. A LAN refers to a group of computers that are linked together to form a network. If you use a computer at work, you probably are connected to a LAN. Most medium-to-large companies have networks of computers to share e-mail, files, video, and so on.

Each computer in a network has an NIC (*Network Interface Card*) installed. The more common name for this interface card is an Ethernet card. Ethernet cards allow computers to communicate across a LAN to other computers. The cards communicate with one another through a protocol, or language. The language used by networks varies from network to network. Microsoft and Novell are two of the largest network-software makers. Microsoft is the most prevalent, considering that its networking software is built right into its Windows line of products. Novell is the maker of NetWare, which is an alternative to Microsoft's software. I use Microsoft's networking software in all of my network setups because it comes free with Windows and also because it is the most commonly used. If I have learned anything about technology, it is that the best does not always win; the most popular does.

When your computer is connected to a network, you normally have an IP address specific to your system. This IP address refers to your identity within the network. If you have access to the Internet through your LAN, you have a separate IP address to the outside world. This is because computers within a network have local addresses that have nothing to do with the Internet. Refer to the "Inside IP Addresses" section earlier in this chapter if you are unclear about this concept.

Setting Up Your Own LAN

To build a network at home, you need multiple computers with Ethernet cards in them and a router or hub. You can buy complete network kits from most electronics stores. Most of these kits contain multiple Ethernet cards and a hub. Basically, you end up installing a network card in each machine and then plugging the computers into a central hub. The hub acts as a virtual traffic cop for all network traffic. You'll most likely install some network services to make the computers visible to each other.

If you are planning to develop network games, and I hope you are, then the most ideal setup is multiple computers on a LAN. With multiple systems connected, you can test your software without having to distribute updates to friends over the Internet. This greatly reduces your development time. If you do not have access to multiple systems, don't worry; most of the programs can run on the same computer. Refer to Figure 3.10 to see a sample home-network setup.

CAUTION

If you are going to develop on a LAN, make sure you install TCP/IP to use the programs in this book. You can install TCP/IP from your Network Settings dialog box.

Something you might notice in Figure 3.10 is the network addresses. The hub in this example does not have an IP address because hubs do not require one. However, if you use a router on your system to connect to the Internet, the router does require its own IP address.

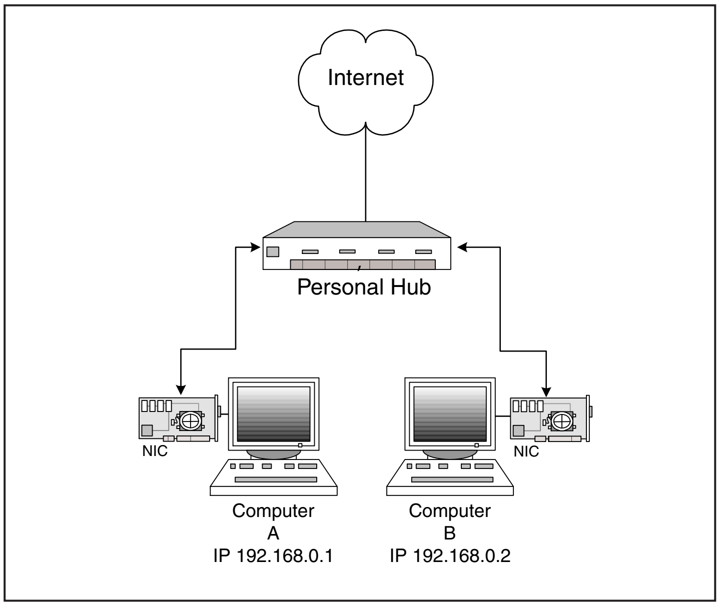


Figure 3.10

A sample home-based network

Server Architecture

If you plan to develop a service such as Blizzard's Battle.net, or if you are going to write an MMORPG, you have to design a dedicated game server. The design put into these types of dedicated servers is very time-consuming and important to your project. There is no perfect setup, so you should expect to experiment quite a bit until you get it right.

One of the most popular game server design methods is a zone-based server system. This system allows large numbers of simultaneous users.

Zone-Based Servers

When I talk about zone-based servers, I am not referring to Microsoft's Gaming Zone, but rather to server-based games that consist of zones. A couple of MMORPG games out now, such as *Ultima Online* and *EverQuest*, use zones to handle the massive number of players. *EverQuest* is the most noticeable because whenever your character ventures far within an area, you are transported into a new zone after loading some data. By breaking the world into zones, the game can span multiple hardware servers. This results in multiple servers sharing the load instead of a single machine doing all the work.

To set up a zone-based server system, you configure multiple computers with like settings. When all of the machines have similar setups, you avoid the situation where any one zone is faster than another. In some cases, this is desirable, though, if you designate a single large computer to handle popular zones.

Connection Servers

For players to connect to your game world, you set up a connection server. The connection server is only responsible for handling requests from players to join your game. With a designated connection server, you reduce traffic to your back-end servers.

Figure 3.11 shows a connection server, which manages all incoming connection requests. When a player starts his program, he connects to the connection server and requests a list of available game worlds. Once the player chooses a game world, he is routed to the appropriate pool of back-end game servers. A nice advantage to a game connection server is that it can make maintenance easier. Instead of having to shut down all of your servers to perform maintenance, you can just turn off traf-

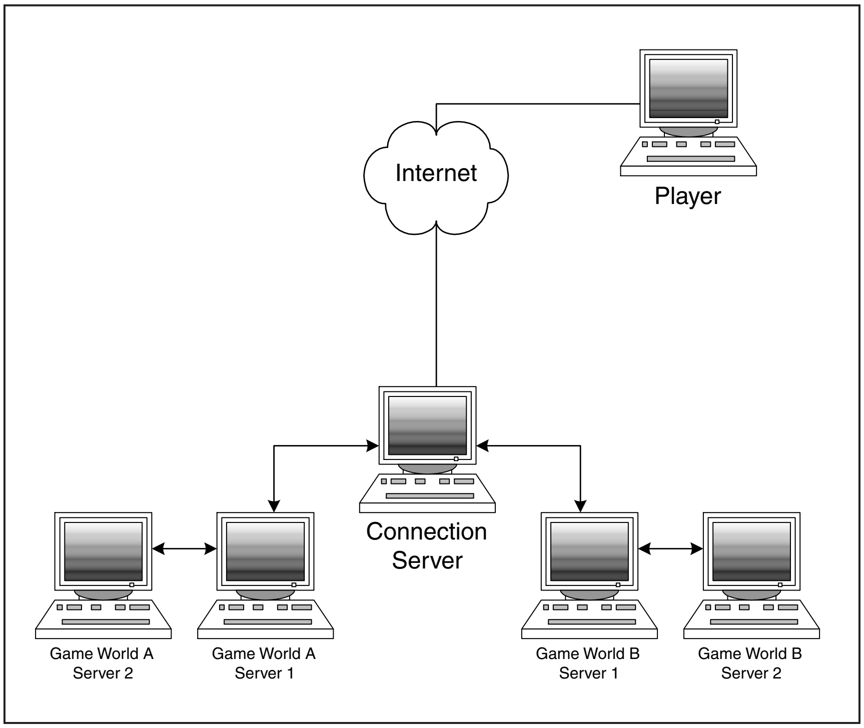
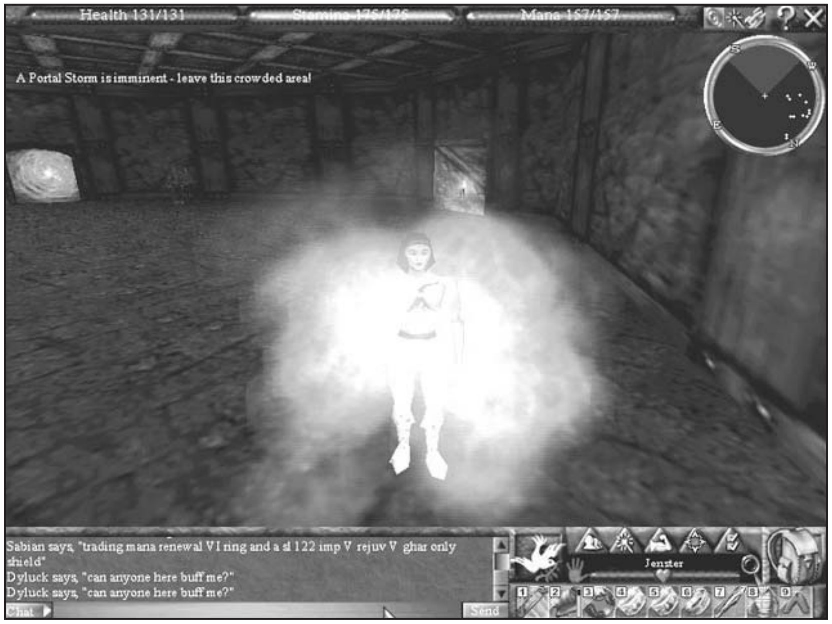


Figure 3.11

A zone-based server setup with a connection server fic to the connection server. You can also use this method to manage the number of simultaneous users in a game world. If your game world gets too full, you just deny access to the world until it has room.

Load Balancing and Fail-Over

Zone-based servers inherently utilize load balancing. The term *load balancing* describes a network that accommodates excess traffic by balancing the communication load across servers. Because the zone-based server method distributes players across servers, it is performing load balancing. Problems can still arise, though, when too many players are present in a single zone. In these instances, you must take action, or players can experience extreme lag, or even worse, the server might crash. You must design a software solution to handle this important issue. One method used in Microsoft's *Asheron's Call* game is *portal storming*. When too many players in this game congregate in one area, people are teleported randomly away from the high traffic zone. This system works on a FIFO, or first-in first-out, basis. This means that the first people into the zone are the first people to get teleported out. In this manner, people who just showed up to the party are not teleported out before others who have been there for a while. Figure 3.12 shows an example of what the player sees when he is about to be portal stormed while playing *Asheron's Call.*



A Portal Storm is imminent - leave this crowded area!

Sabian says, "trading mana renewal VI ring and a sl 122 imp V rejuv V ghar only shield"
Dyluck says, "can anyone here buff me?"
Dyluck says, "can anyone here buff me?"

Figure 3.12

An example of portal storming in Microsoft's Asheron's Call

In Figure 3.12, the player is receiving an on-screen message stating that a portal storm is brewing and that he must leave the area. If the situation gets worse, the player might be teleported to another nearby location. Even though this can be downright annoying to players, it's better than having a game server crash!

A big issue with game servers is *fail-over*, which refers to the actions that take place when a server goes down. If your game has a fail-over solution, your game world does not go down when a server crashes. A fail-over solution is difficult at best to implement because data synchronization is problematic. One way to create a fail-safe system is to have duplicate game servers share a common file system. When a server goes down in this system, its duplicate takes over. The servers all share data, so there are no problems with synchronization. The only problem with this solution occurs when the shared file system crashes. If this happens, the entire world goes offline. Even though there really is no way to keep your game world up all of the time, it is better to put in place some fail-over than none at all. Take a look at a sample fail-over system in Figure 3.13.

You are done with server architecture for now. I suggest that you play some games such as Origin's *Ultima Online* and Microsoft's *Asheron's Call* to see whether you can tell how the servers work.

Packet, Packet, Who's Got the Packet?

Ahhhh, nothing beats a good packet. So what is a packet anyway? A packet is a block of data sent or received. Basically, a packet defines a data block that you either send to another computer or receive from another computer. When you send data to another computer, it is best to have a set of defined data structures. This lets the program know what data to look for when it processes a packet. Suppose you want to send a speech packet to another player in a game. When the other player receives the packet, how does he even know what to do with it if you don't have predefined structures? That's where packets come in.

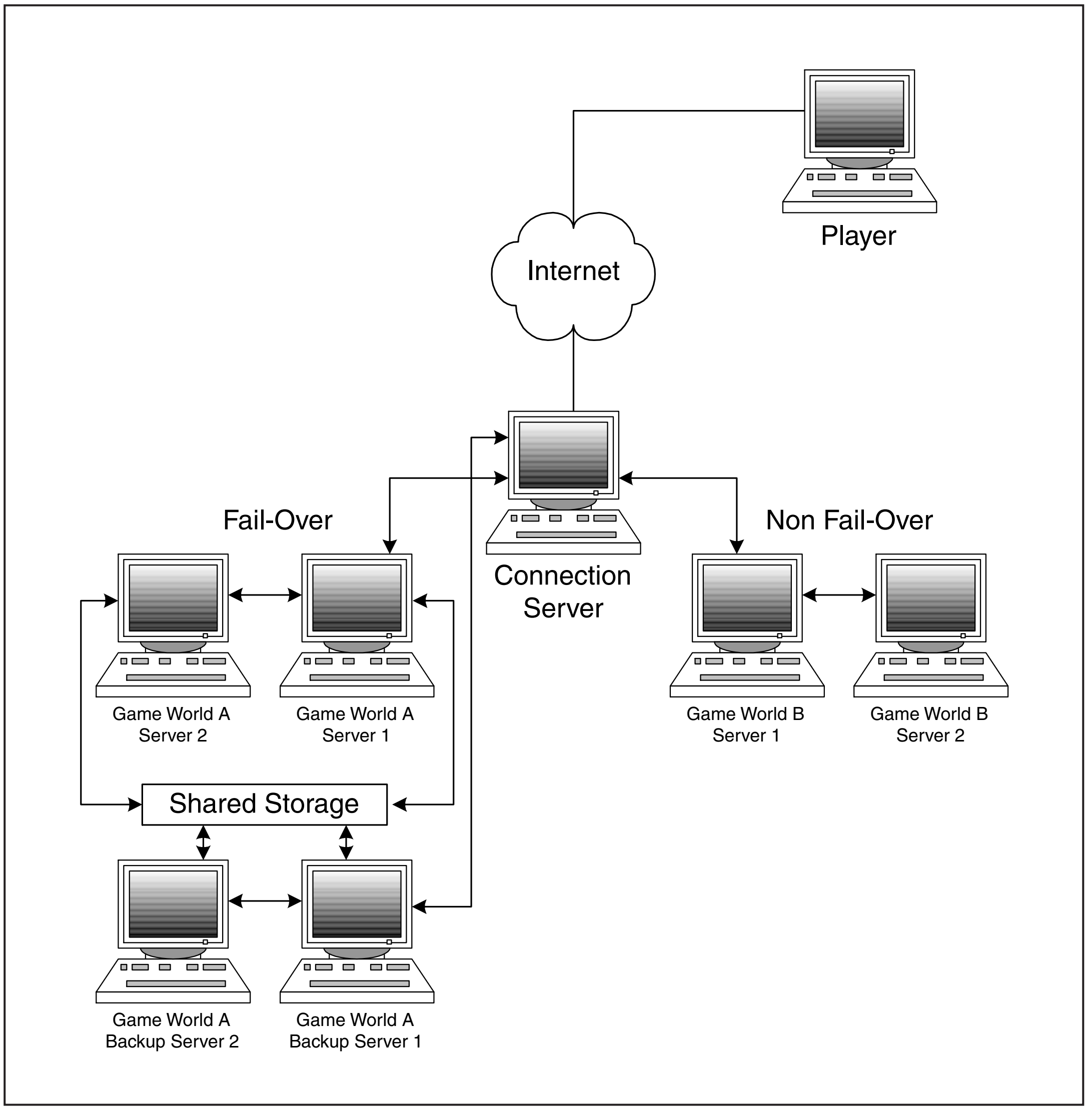


Figure 3.13 *Fail-over server architecture*

The Tic-Tac-Toe Example

Take a moment to examine a simple game and how it would structure its game packets. Suppose the game is Tic-Tac-Toe, and the players can chat back and forth. The game requires the following actions.

- **Move action**. This action tells the other player that a move has been made. After receiving the move action, the other player then takes his turn.
- **Talk action**. This action talks to the other player. When you type something into the chat window, you want to send it to the other player so he can read it.

Okay, now that you have some actions to deal with, I am going to talk about how you structure them in packets. The first action, Move, is fairly simple. All you have to send is the desired location for the X or O. The packet definition might look something like this:

```
#define PACKET_TYPE_MOVE      1

// Standard header packet
struct pacStandardHeader
{
    int     iPacketType;
    int     iPacketLength;
};
// Move packet
struct pacMove
{
    pacStandardHeader      pacHeader;
    int                    iHorizontalPosition;
    int                    iVerticalPosition;
};
```

The `pacMove` structure contains some header information and information regarding the move taken. The header information is contained in a common structure called `pacHeader`. This data structure gives the identity of the packet along with the amount of data to be expected from the packet. The actual data stream for this packet might look like the following.

```
1
16
2
2
```

The first item listed is the number 1. When the program receives the packet, it first looks at the header information. The identity of the packet is number 1, which is defined as PACKET_TYPE_MOVE. This tells the program what to expect from the packet. The next number, 16, tells the program the size of the packet. Because I am using a 32-bit operating system, integers are four bytes each. Because the packet has four integers in it, the total size is 16 bytes. When you include a packet length, the program knows when the packet ends and the next one begins. The last two numbers read are the horizontal and vertical positions for the move. The Tic-Tac-Toe grid has three rows across and three rows down. This means the upper-left coordinate is 0,0 and the lower-right coordinate is 2,2. In the preceding example, the move is made in the lower-right corner.

Wow, you just processed a Move packet. Now suppose you want to send a chat message to your friend. To send the message, you transmit a Talk packet. The Talk packet might look like the following:

```
#define PACKET_TYPE_TALK      2
struct pacTalk
{
     pacStandardHeader       pacHeader;
     char                    szMessage[128];
};
```

The Talk packet contains the usual header structure along with a character array to store the message. In this case, you allow up to 128 characters to be sent in the chat message. Suppose that you want to send "Hello World" to your friend. The data stream from the packet looks like the following:

```
2
136
Hello World\0
```

The first number read is the usual packet type identifier. Because this is packet type 2, the program knows it is a Talk packet. The next line read is the usual packet length. In this case, the length is 136 because you have two integers, four bytes each, and 128 bytes worth of character data. The last line read is the actual contents of szMessage. The string terminator, \0, appears here. The string terminator tells you where the message ends. If a terminator was not in place, you might see garbage text after the actual message. There you have it—packet design for Tic-Tac-Toe! I know it's not very glamorous, but I want to keep it simple until you get the hang of it.

Ideal Packet Size

"Ideal packet size" is somewhat of a controversial topic name. There really is no ideal packet size. It depends greatly on what you are trying to accomplish. When designing the packet size for your game, think about what the players have to communicate. Your largest packet should fit perfectly the largest action a player can send. There is no sense in wasting space. First, you have to make a choice on whether to send many small packets or a few large packets. The advantage to smaller packets is that you can send a lot of them in a short amount of time. The advantage to larger packets is that you cut down on transmission overhead because you are sending fewer packets. Every time you transmit data across the wire, an amount of overhead goes with it. This is especially prevalent in guaranteed packet delivery protocols such as TCP/IP.

Some action games, such as first-person shooters, use a packet size of 512 bytes. This lets you send quite a bit of information at a time, but is small enough to be quick over most connections. On the other hand, one of the first network game systems I worked on used 11-byte packets for delivery. I have to say that 11 bytes is a very tiny packet size, and I do not recommend it. If you just can't decide on the perfect packet size, you can always use dynamic-length packets.

Dynamic-Length Packets

Dynamic-length packets are packets whose sizes vary. They allow games to handle multiple packets of different lengths. The Tic-Tac-Toe example shows you how to use packets with dynamic lengths. The only real downside to packets of this type is that your program never really knows how much data to pull from the receive buffer to get the next complete packet. Refer to Figure 3.14 to see an example of fixed-length and dynamic-length buffers.

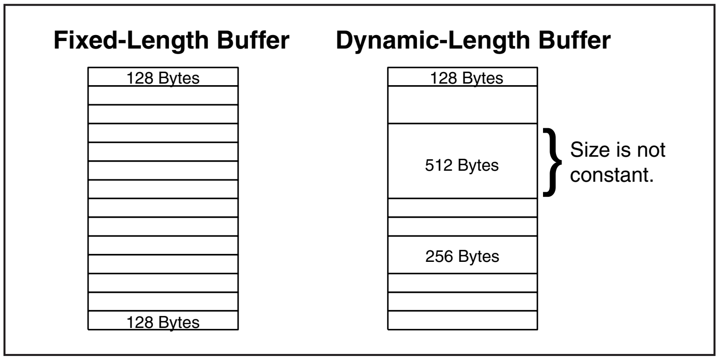


Figure 3.14

Fixed-length and dynamic-length packet buffers

For most of the games I develop, I use dynamic-length packets. I prefer the flexibility to send large or small packets depending on what I am trying to do.

Review

Here you are, at the end of another chapter already! Now that you have had a crash course on everything from network protocols to server design, take a moment to do a little review.

- A network is a system of computers interconnected by telephone wires or other means to share information.
- Network games fall into one of the two types, peer-to-peer or client/server. The peer-to-peer method is used mostly for games that have four or less players. The client/server method is used mostly for games that have more than four players or MMORPG-type games.
- When an IP address is static, it never changes. Static IP addresses are mostly used by fixed servers rather than by game players. If your IP address changes every time you log onto the Internet, then you have a dynamic IP address. Dynamic IP addresses are common for dial-up connections.
- A packet defines data used for sending and receiving.

This page intentionally left blank

Part Two

Sockets-Based Network Programming

CHAPTER 4

Writing Your First Network Programs

Hey, guess what? It's time to write a sockets program! Confetti falls from the ceiling, balloons rise into the air, and Bill Gates and Steve Jobs dance until dawn. Okay, seriously though, it's time to finally write some communication code.

The programs in this chapter use a C++ class object for the socket interface. After this chapter is over, you will know how to program for sockets. What you will not know is how to do the low-level code for sockets. I address this issue in Chapter 5, "Creating a Sockets Class Object," by showing you how to write a socket object class from the ground up. In this chapter, I'll cover three complete sockets programs.

The first program, ConnectionTest, demonstrates a client connecting to a server. This program shows the absolute minimal method for a client and server to connect. It provides the foundation for all of the other programs.

The second program, SendDataTest, demonstrates a client sending data to a server. In this sample program, you'll learn how the server listens for data and how the client sends it.

The third program, SimpleChat, demonstrates two-way communication between both client and server. This fully functional program allows the server and client to interactively talk to each other.

After you finish this chapter, you will have learned several things, including

- How the `SocketObject` class is structured
- How to establish a connection
- How to send and receive data
- How to write a chat program

I'll finish up with some summary information and suggested exercises for skill development. Let the games begin!

Overview of the SocketObject Class

Now that all of the theory is out of the way, it's time to start writing some code. To make life easier for you, I have written a class that handles all of the socket communication. The `SocketObject` class provides a layer of abstraction away from the low-level Windows Sockets API. By using the class, the sample programs are easier to read and follow. Okay, now it's time to take a look at the member functions available in the class.

SocketObject Class Member Functions

The `SocketObject` class has a handful of member functions at its disposal. Some of the member functions are specific to the server, some are specific to the client, and both servers and clients share a few.

The functions in Table 4.1 are specific to programs acting as servers. Server programs manage all connections and keep game state.

The `Connect` function is specific to programs acting as clients. The client uses `Connect` to establish a connection with a server at the specified IP address and port number. Clients connect to servers to play in the game world.

Both servers and clients share the functions in Table 4.2.

NOTE

Keep in mind that clients can also be servers and vice versa. This allows for a type of two-way communication that helps reduce packet collisions.

Table 4.1 Server-Specific Functions

Function	Purpose
`Bind`	Establishes a port on which to listen
`Listen`	Checks for new client connections
`Accept`	Accepts an attempted client connection

Table 4.2 Functions Shared by Both Servers and Clients

Function	Purpose
`Recv`	Receives data from a specified connection
`Send`	Sends data to a specified connection
`Disconnect`	Closes an open connection

SocketObject Class Sample Code

Now that you have seen an overview of the member functions, here is some sample code to go along with them.

CAUTION

The code snippets shown are not complete stand-alone programs, so don't try to compile them by themselves.

Server Function Sample Code

All right, now that you have had the nickel tour, it's time to learn a little about the server functions. The following code snippet shows some of the functionality available to server objects. The code demonstrates all of the steps necessary to establish a connection with a client. To establish a connection, the server takes the following steps.

1. Establish a port to accept connection requests.
2. Listen for a connection.
3. Accept the client connection.
4. Disconnect from the client connection.

Okay, here is the code that completes the steps.

```
// The following code snippet shows how a server listens for, accepts,
and then disconnects a client
SocketObject     ServerSocketObject;
SocketObject     ClientSocketObject;
// Server listens on port 6000
ServerSocketObject.Bind( 6000 );
// Server waits for a connection request
ServerSocketObject.Listen();
// Server has received a connection request; now it accepts it
ServerSocketObject.Accept( ClientSocketObject );
// Server disconnects the client
ClientSocketObject.Disconnect();
```

The first member function shown is the `Bind()` function. The `ServerSocketObject` binds to port 6000. By *binding*, the server sets the port number from which clients are allowed to request connections. The next member function shown is the `Listen()` function. Nothing must be passed to this function, so read on.

With the `Accept()` function, the server accepts the connection request and assigns the `ClientSocketObject` to the connection. Notice that two `SocketObjects` are created. By creating a new `SocketObject` for each connection, the server can manage connections more easily.

To disconnect from the client, you use the `Disconnect()` member function of the `ClientSocketObject`. Refer to Figure 4.1 to see the structure of this code sample.

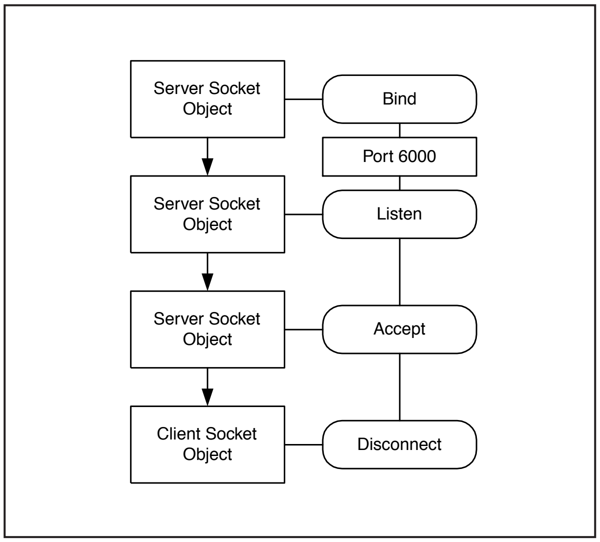


Figure 4.1

Process flow for the server code example

That's really about all there is to establishing a connection with a client. It's time to jump to the other side of the fence and see what the client has to accomplish.

Client Function Sample Code

The first thing a client must do in a multiplayer game is connect with the game server. Here is the code to do just that.

```
// Attempt to connect to a server with an IP address of 100.100.100.100
on port 6000
if( ClientSocketObject.Connect( "100.100.100.100", 6000 )
{
    // Disconnect from the server
    ClientSocketObject.Disconnect();
}
```

Because the client doesn't have to worry about much, it only has to call one function to establish a connection. And that function is . . . drum roll please . . . the `Connect()` function!

In the example, the `Connect()` member function establishes a connection to a server with an IP address of 100.100.100.100. The connection is tried on the server's port number 6000. If the connection is successful, the `ClientSocketObject` then calls the `Disconnect()` member function to end the communication.

Ta da! That is it for establishing a connection with a server. Okay, the basics are out of the way now, so it's time to look at some complete code samples.

Writing the ConnectionTest Program

Now that you have a basic understanding of the functions available in the `SocketObject` class, you finally get to write a full program. I've always thought that starting with the basics is the best approach because it helps weed out any unnecessary information, so I am going to start you off with something simple to test. The first program demonstrates a server listening for a client connection and a client connecting to the server. Once the connection is established, both the client and server disconnect. You need to run two instances of the program from the command prompt—one for the server and one for the client.

Components Needed to Build the Program

Before you take a look at the code, I want to show you which files you need to compile and link the program. Table 4.3 details the files used in the project. Refer to Figure 4.2 for a hierarchical view of the file structure.

Table 4.3 Components for the ConnectionTest Program

Component	Purpose
CONNECTIONTEST.CPP	The main body of the program that handles both the client and server code. This is the only code that you will examine for this example.
SOCKETOBJECT.CPP	The class file for the sockets functions, which holds all of the member functions in the `SocketObject` class.
SOCKETOBJECT.H	The class file for the sockets functions, which holds all of the header information for the `SocketObject` class.
WS2_32.LIB	The Windows library that holds all of the low-level sockets functions.
WINSOCK.H	The header to the Windows ws2_32.lib library.

Getting Started with the Compiler

I say it's time to create a program, so here goes nothing. Because this is your first program to write from this book, I want to walk you through the process of creating the project from scratch.

1. Open the Visual C++ program and select File, New.
2. Under the Projects tab, select Win32 Console Application.

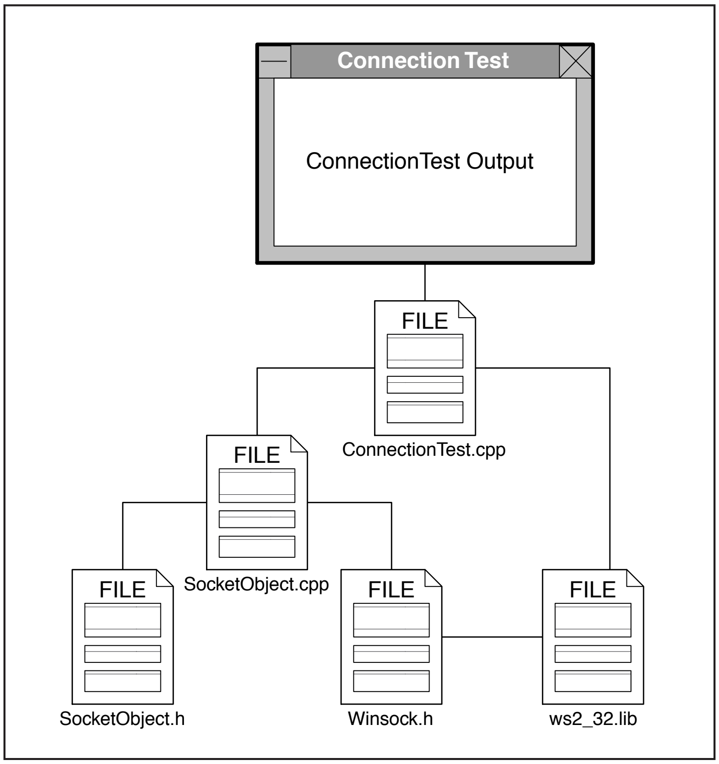


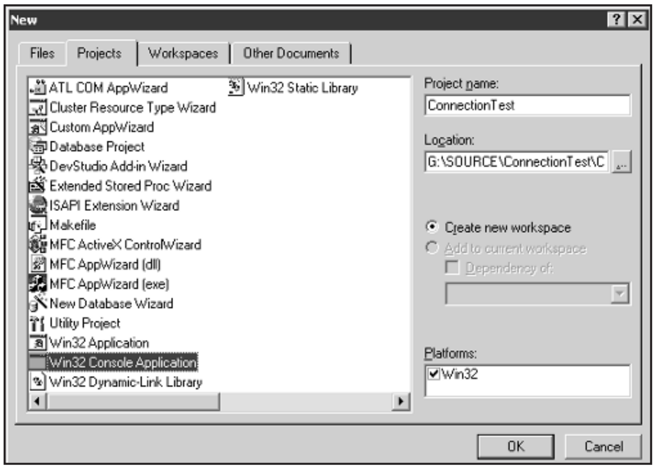
Figure 4.2

A hierarchical view of the files used in the program

3. Enter the project name, ConnectionTest, in the Project Name text box.
4. Select where you want to save the project in the Location box. Your screen should resemble the image in Figure 4.3.
5. Select OK.

You are now prompted with two dialog boxes; first select Finish and then select OK. Now that you have a project started, it is time to add the required files to make it work. First, add the ws2_32.lib library to the project. To add the library, follow these steps.

1. Choose Project, Settings in your Visual C++ compiler, and select the Link tab.
2. Under the Object/Library Modules text box, add ws2_32.lib to the list of libraries already there. This library is the Windows socket library that you use in all sockets-based programs, so be sure to repeat this step whenever creating a program that uses sockets.
3. Select OK.



Project name: ConnectionTest

Location: G:\SOURCE\ConnectionTest\C

(●) Create new workspace

Platforms: ☑ Win32

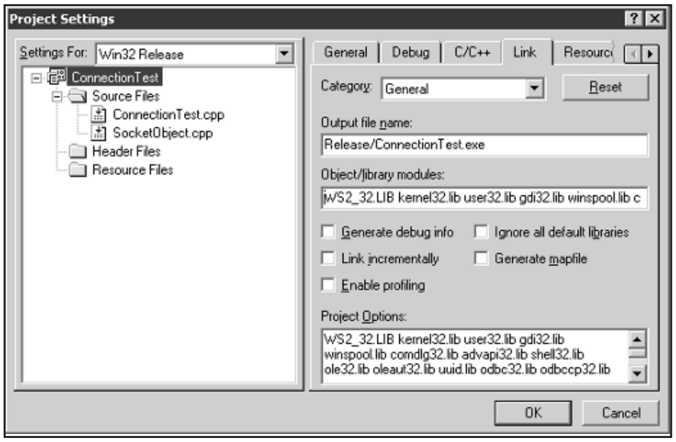
Figure 4.3

Creating a console application in Visual C++

When you have the project window open, it should look like the image in Figure 4.4.

It is time to add the `SocketObject` class files to the project. The `SocketObject` files consist of a .cpp source file and an .h header file. Both of these files are used in all of the upcoming sockets projects. Follow these steps to add the files.

1. Select Project, Add To Project, and then select Files from the Visual C++ menu. You are prompted with a browse window.
2. Browse to the folder named SocketObject wherever you installed the CD that came with the book, and select SocketObject.cpp.
3. Select OK.



Settings For: Win32 Release

ConnectionTest
- Source Files
  - ConnectionTest.cpp
  - SocketObject.cpp
- Header Files
- Resource Files

Category: General

Output file name: Release/ConnectionTest.exe

Object/library modules: WS2_32.LIB kernel32.lib user32.lib gdi32.lib winspool.lib c

Project Options:

WS2_32.LIB kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib

Figure 4.4

Modifying your project settings in Visual C++

Now you need an empty source file in which to enter the code for the project. Follow these steps to create the file.

1. Select Project, Add To Project, New from the Visual C++ menu. You are prompted with several file type options.
2. From the Files tab, select C++ Source File.
3. In the File Name text box, enter the name **ConnectionTest.cpp**.
4. Select OK.

Writing and Compiling the Code

A new empty source file window is just waiting for you to add some code to it, so get to it! Type in the following source code or copy it from the CD that came with the book.

```
//###########################################################################
// Standard includes
#include <iostream.h>
#include "..\\socketobject\\SocketObject.h"
void vServerConnection( int iListenPort );
void vClientConnection( char *szServerIP, int iServerListenPort );

// ----> Main program function (REQUIRED)
int main( int argc, char *argv[] )
{
    if( argc < 3 ) {
        cout << "---------------------------------------" << endl;
        cout << "                ConnectionTest Help        " << endl;
        cout << "---------------------------------------" << endl;
        cout << "Usage: ConnectionTest [client/server]
         [ip,port/port,port]" << endl;
        cout << "" << endl;
        cout << "Example: ConnectionTest client 198.168.0.1 6000" << endl;
        cout << "" << endl;
        cout << "Example: ConnectionTest server 6000" << endl;
        cout << "" << endl;
        return( 0 );
    }
    //
    // If user selected server, listen on the given port
    // and assign connection to the second port number
    //
    if( !stricmp( argv[1], "server" ) ) {
        vServerConnection( atoi( argv[2] ) );
    }
    //
    // User selected client; connect to given port and IP address
    //
    else {
        vClientConnection( argv[2], atoi( argv[3] ) );
    }
    return( 1 );
}

// Function for server
void vServerConnection( int iListenPort )
{
    SocketObject     ServerSocketObject;
    SocketObject     ClientSocketObject;

    cout << "<Server> Attempting to listen on Port " << iListenPort << endl;
    // Attempt to start the server on port 6000
    if ( ServerSocketObject.Bind( iListenPort ) ) {
        cout << "<Server> Listening" << endl;
        // Listen for connection on the listen port,
        ServerSocketObject.Listen( );
        // Accept the connection
        ServerSocketObject.Accept( ClientSocketObject );
        cout << "<Server> Client Connected to Port " << iListenPort << endl;
        // Disconnect the client
        ClientSocketObject.Disconnect();
        cout << "<Server> Client Disconnected" << endl;
    }
    else {
        cout << "<Server> Failed to Listen" << endl;
    }
}

// Function for client
void vClientConnection( char *szServerIP, int iServerListenPort )
{
    SocketObject     ClientSocketObject;
    cout << "<Client> Connecting to " << szServerIP << ", Port " <<
iServerListenPort << endl;
    // Connect to the IP and port
    if( ClientSocketObject.Connect( szServerIP, iServerListenPort ) ) {
        cout << "<Client> Connected" << endl;

        // Disconnect from the server
        ClientSocketObject.Disconnect();

        cout << "<Client> Disconnected From Server" << endl;
    }
    else {
        cout << "<Client> Failed to Connect" << endl;
    }
}
//###############################################################################
```

Now that you have the code in the ConnectionTest.cpp file, select File, Save from the Visual C++ menu. To compile the program, select Build, Build ConnectionTest.exe. You can also use F7 as a hotkey to perform this function. If everything went well, you'll see "ConnectionTest.exe - 0 error(s), 0 warning(s)" in the output window. If you see an error, go back and double-check all of the steps you completed.

Running the Server and Client Programs

Even though it might look like a lot of code for a simple connection and disconnection, notice that most of it is the cout command. If you remove the remarks and cout commands, not much is left. Now, test out your new program! Open a command prompt (DOS window), change to the directory where you saved your project, and then go to the debug folder.

Run the program first with the following parameters.

```
ConnectionTest server 6000
```

Now, open another command prompt, go to the debug folder again, and run the program with the following parameters. (Replace the 100.100.100.100 with your IP address.)

```
ConnectionTest client 100.100.100.100 6000
```

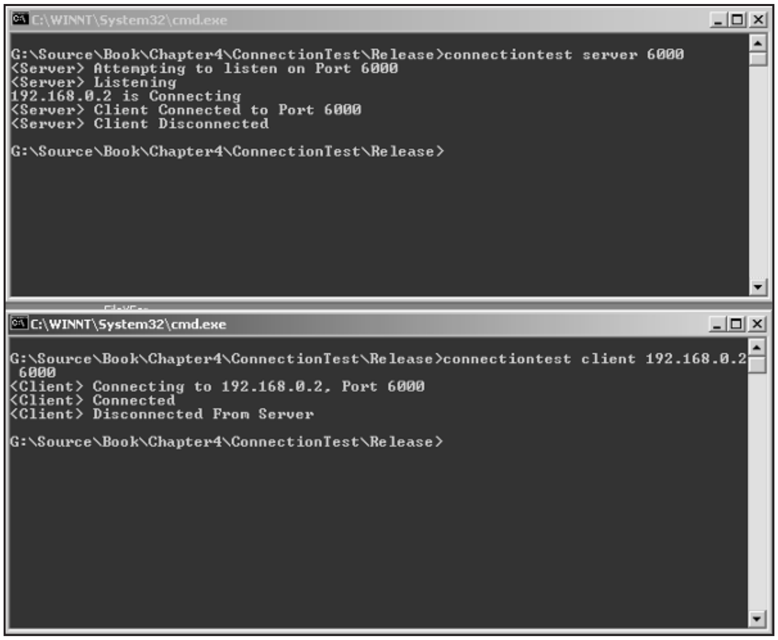
If everything worked properly, you should see two output screens, as shown in Figure 4.5.

Pretty cool, huh? If you take a look at the output, you'll see that the following events happened.

1. The server bound to port 6000 to listen for connections.
2. The client attempted to connect to port 6000 on the server.
3. The server accepted the request.
4. The client disconnected.
5. The server disconnected.

TIP

To find your computer's IP address in Windows 98, run the winipcfg program. To find your computer's IP address in Windows NT or Windows 2000, run the ipconfig program.



```
C:\WINNT\System32\cmd.exe

G:\Source\Book\Chapter4\ConnectionTest\Release>connectiontest server 6000
<Server> Attempting to listen on Port 6000
<Server> Listening
192.168.0.2 is Connecting
<Server> Client Connected to Port 6000
<Server> Client Disconnected

G:\Source\Book\Chapter4\ConnectionTest\Release>
```

```
C:\WINNT\System32\cmd.exe

G:\Source\Book\Chapter4\ConnectionTest\Release>connectiontest client 192.168.0.2
 6000
<Client> Connecting to 192.168.0.2, Port 6000
<Client> Connected
<Client> Disconnected From Server

G:\Source\Book\Chapter4\ConnectionTest\Release>
```

Figure 4.5

Output from the server and client programs

Examining the Code

The first major hurdle is out of the way, so give yourself a pat on the back for all that hard work. Now I'll walk you through the code I just showed you.

Header Information

The first few lines of the code handle all of the header information needed for the program. The header files let your program know how to use the functions you want. Most header files have a matching library or source file to go along with them.

```
#include <iostream.h>
#include "..\\socketobject\\SocketObject.h"
```

The iostream.h header file is included to give your program basic IO functionality. Without this header file, your program would not be able to input or output anything to the command prompt. The second header file, SocketObject.h, is included so the program can access the functionality of the `SocketObject` class. This header file is included in all of the sockets-based programs. The next few lines of code define the prototypes for the client and server functions.

CAUTION

You might need to alter the path for the SocketObject.h header file to resemble your file system. If you have copied over the CD structure verbatim then you should not have any problems. This rule applies to all of the source code in this book.

```
void vServerConnection( int iServerListenPort );
void vClientConnection( char *szServerIP, int iServerListenPort );
```

The `vServerConnection()` function takes one parameter. The first and only parameter, named `iServerListenPort`, is the port number where all incoming connection requests are received. It is a good practice to pick a number between 1000 and 9999 to assign as the listening port.

The `vClientConnection()` function takes two parameters. The first parameter, named `szServerIP`, is a character pointer to the IP address of the server where the connection request is sent. The second parameter, named `iServerListenPort`, is the port number on the server that accepts the connection request.

The main() Function

All console applications require a `main()` function. This function is the first one called when the program executes. The sample `main` function has two parameters with it, `argc` and `argv`.

```
int main( int argc, char *argv[] )
```

These two parameters handle user input at the command line. A real-world example concerns the `copy` command. To use the `copy` command, you must pass at least two parameters along with it, the source file name and the destination file name. The first parameter in your `main()` function is `argc`, which holds the number of arguments the user passed into the program. The next parameter is `argv`, which points to the array of parameters the user passed.

The first part of the `main()` function checks the number of parameters passed. If the number of parameters passed is less than what is minimally required, then the program displays some help text.

The ConnectionTest program requires at least two parameters, plus the name of the program, which gives it three parameters total. So it checks for three parameters minimally.

```
if( argc < 3 )  {
    cout << "----------------------------------------" << endl;
    cout << "                ConnectionTest Help         " << endl;
    cout << "----------------------------------------" << endl;
    cout << "Usage: ConnectionTest [client/server] [ip,port/port,port]" << endl;
    cout << "" << endl;
    cout << "Example: ConnectionTest client 198.168.0.1 6000" << endl;
    cout << "" << endl;
    cout << "Example: ConnectionTest server 6000" << endl;
    cout << "" << endl;
    return( 0 );
}
```

It is a good practice to output help text from console applications when the user enters incorrect parameters. If the user passed at least two parameters, go on to check whether he is running the server or client version of the ConnectionTest program.

```
if( !stricmp( argv[1], "server" ) ) {
    vServerConnection( atoi( argv[2] ) );
}
else  {
    vClientConnection( argv[2], atoi( argv[3] ) );
}
```

You make the check for server or client using the `stricmp()` function. This function compares two strings and returns a `0` if they are identical regardless of case. If you want to be case sensitive, you use the `strcmp()` function with the same parameters.

If the user chose to launch the server version of the program, then you call the `vServerConnection()` function by passing the listening port number to it. The listening port is the one the program binds to and listens for connections on.

If the user chose not to run the server version of the program, the program defaults to the client version. Two parameters are passed to the `vClientConnection()` function, the server IP address and the server listening port.

CAUTION

If you are following along and converting the programs to UNIX, the `stricmp()` function probably will not work for you. It is not an ANSI standard function. I recommend that you write your own version of the function for use in the future.

Inside the vServerConnection() Function

Once the `vServerConnection()` function is invoked from the `main()` function, it creates two `SocketObject` objects, the `ServerSocketObject` and the `ClientSocketObject`.

```
SocketObject    ServerSocketObject;
SocketObject    ClientSocketObject;
```

After the two objects are created, the server then tries to open up a listening port with the `Bind()` function. The function tries to bind to the port number passed to it. In the examples, I use 6000 as the port number. If the bind fails, the program displays an error message and exits out of the function. Bindings can fail if the port is already open by another application.

```
if ( ServerSocketObject.Bind( iListenPort ) )  {
    ...
}
```

Once the bind succeeds, the program starts listening for connections with the `Listen()` function. When it is combined with the `Accept()` function, the program waits for connections.

The `Accept()` function is called to accept attempted connections. It is passed a client `SocketObject` to use for the connection. Refer to Figure 4.6 for a visual of the server's port usage.

```
ServerSocketObject.Listen( );
ServerSocketObject.Accept( ClientSocketObject );
```

Normally, the server does something else, such as send data to the client or wait for more connections. Because I want to keep it simple, the server just disconnects the client with the `Disconnect()` function. No parameters are needed, but you should note that it uses the `ClientSocketObject`'s member function to issue the disconnection. You use this function because you are not disconnecting the server from a host; you are disconnecting the client from the server.

```
ClientSocketObject.Disconnect();
```

That's it for the server function. It accomplishes what it set out to do as a server. To recap what was covered, here are the steps the function followed.

1. Bind to port 6000 to listen for connections with the `Bind()` function.
2. Listen on the bound port, and assign the new connection to it with the `Listen()` function.
3. Accept the new connection, and assign a `SocketObject` to it with the `Accept()` function.
4. The client disconnected from the server using the `Disconnect()` function.

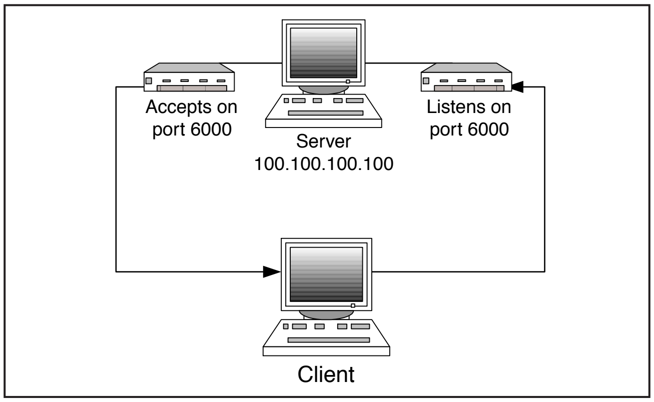


Figure 4.6

Server port usage

Inside the vClientConnection() Function

The `vClientConnection()` function must create only one local `SocketObject` object. The first thing the function does is create this object upon execution:

```
SocketObject      ClientSocketObject;
```

Once the object is created, all that is left to do is attempt a connection to the server with the `Connect()` function. The `Connect()` function is passed the server's IP address and listening port number. If the connection attempt is successful, a `1` is returned. Check against this return code to determine whether to proceed with disconnection.

```
if( ClientSocketObject.Connect( szServerIP, iServerListenPort ) ) {
     cout << "<Client> Connected" << endl;
     // Disconnect from the server
     ClientSocketObject.Disconnect();
     cout << "<Client> Disconnected From Server" << endl;
}
else {
     cout << "<Client> Failed to Connect" << endl;
}
```

If the connection attempt returns a `0`, output an error message and exit the function; otherwise, output a message and then disconnect from the server.

The client disconnects by calling the `Disconnect()` member function of the client's `SocketObject`. Because you are disconnecting yourself from the server, it differs from when a server must disconnect a client, as shown in the "Inside the vServerConnection() Function" section. Because I am keeping things simple for now, this is it for the client function. To recap what was covered, here are the steps the function followed.

1. Connect to server IP address and port number using the `Connect()` function.
2. Disconnect from the server using the `Disconnect()` function.

You have now covered the entire first program. That wasn't so bad, was it? Now move on to something a bit more complicated.

Writing a Program to Send Data

The first program was quite simple in its result, but you did accomplish quite a bit. Now I want to show you something a little more exciting—a program that actually sends data!

The SendDataTest program is identical to the ConnectionTest program except for two functions, `Send()` and `Recv()`.

The client in this example uses the `Send()` function to send some data to the server. The data to send is stored in a standard character array, 128 bytes in length. The amount you can send is not limited to 128 bytes; I just chose that amount for efficiency. The return code for the `Send()` function tells you how many bytes were transmitted. If nothing is transmitted, a `0` is returned.

The `Recv()` function waits for data to come in from a connection, and then it stores that data in a character buffer. The `Recv()` function waits indefinitely for data to come in. Once data is received, the function returns the number of bytes received. Even though you tell the `Recv()` function how many bytes to retrieve, there is no guarantee you will get all of the data at once. The network protocol can break transmissions into many packets. This is why it is important to check how many bytes are received. If you do not get all of the data expected, you should call `Recv()` again until you get the number of bytes expected.

Components Needed to Build the Program

This project is almost identical to the ConnectionTest program, so I will be brief about the contents. A list of the files used in the project can be seen in Table 4.4.

Are you experiencing déjà vu yet? If you haven't slept too much since the last section, you'll notice that all of the files are the same as those for the ConnectionTest program, except the name of the main source file. ConnectionTest.cpp has been replaced with SendDataTest.cpp. Follow the necessary steps outlined in "Getting Started with the Compiler" and create a new project named SendDataTest.

Table 4.4 Components for the SendDataTest Program

Component	Purpose
SENDDATATEST.CPP	The main body of the program that handles both the client and server code. This is the only code that I examine for this example.
SOCKETOBJECT.CPP	The class file for the sockets functions, which holds all of the member functions in the `SocketObject` class.
SOCKETOBJECT.H	The class file for the sockets functions, which holds all of the header information for the `SocketObject` class.
WS2_32.LIB	The Windows library that holds all of the low-level sockets functions.
WINSOCK.H	The header to the Windows ws2_32.lib library.

Writing the SendDataTest Program

You can either type in the following code or copy it from the CD that came with the book. I recommend that you type in most of the code samples, to help you become more familiar with the functions.

```
//###################################################################################

// Standard Includes
#include <iostream.h>
#include "..\\SocketObject\\SocketObject.h"

void vServerConnection( int iListenPort );
void vClientConnection( char *szServerIP, int iServerListenPort );

// ----> Main Program Function (REQUIRED)
int main( int argc, char *argv[] )
{
     if( argc < 3 ) {
         cout << "----------------------------------------" << endl;
         cout << "                SendDataTest Help           " << endl;
         cout << "----------------------------------------" << endl;
         cout << "Usage: SendDataTest [client/server]
          [ip,port/port,port]" << endl;
         cout << "" << endl;
         cout << "Example: SendDataTest client 198.168.0.1 6000" << endl;
         cout << "" << endl;
         cout << "Example: SendDataTest server 6000" << endl;
         cout << "" << endl;
         return( 0 );
     }
     // If user selected server, listen on the given port
     // and assign connection to the second port number
     //
     if( !stricmp( argv[1], "server" ) ) {
         vServerConnection( atoi( argv[2] ) );
     }
     // User selected client; connect to given port and IP address
     else {
         vClientConnection( argv[2], atoi( argv[3] ) );
     }
     return( 1 );
}

// Function for server
void vServerConnection( int iListenPort  )
{
     SocketObject     ServerSocketObject;
     SocketObject     ClientSocketObject;
     char             DataPacket[128];          // Data packet to receive
     int              iBytesReceived = 0;       // # of bytes received

     cout << "<Server> Attempting to listen on Port "
      << iListenPort << endl;
     // Attempt to start the server
     if ( ServerSocketObject.Bind( iListenPort ) ) {
         cout << "<Server> Listening" << endl;
         // Listen for connection on the listen port
```

```
            ServerSocketObject.Listen( );
            // Accept the connection
            ServerSocketObject.Accept( ClientSocketObject );
                cout << "<Server> Client Connected to Port "
                  << iListenPort << endl;
              // Receive data
              iBytesReceived = ClientSocketObject.Recv(&DataPacket, 128, 0);
              cout << "<Server> Received " << iBytesReceived <<
                " Bytes" << endl;
              cout << "<Server> Data Received = " << DataPacket << endl;
              // Disconnect the client
              ClientSocketObject.Disconnect();
              cout << "<Server> Client Disconnected" << endl;
         }
         else {
              cout << "<Server> Failed to Listen" << endl;
         }
}

// Function for client
void vClientConnection( char *szServerIP, int iServerListenPort )
{
    SocketObject     ClientSocketObject;
    char             DataPacket[128];          // Data packet to transmit
    int              iBytesSent = 0;           // # of bytes sent

    cout << "<Client> Connecting to " << szServerIP << ", Port "
     << iServerListenPort << endl;
    // Connect to the IP and port
    if( ClientSocketObject.Connect( szServerIP, iServerListenPort ) ) {
        cout << "<Client> Connected" << endl;
        // Populate the data packet
        strcpy(DataPacket, "TestData from Client");
        // Send data
        iBytesSent = ClientSocketObject.Send(DataPacket, 128, 0);
        cout << "<Client> Transmitted " << iBytesSent <<
          " Bytes" << endl;
        // Disconnect from the server
        ClientSocketObject.Disconnect();
        cout << "<Client> Disconnected From Server" << endl;
    }
    else {
        cout << "<Client> Failed to Connect" << endl;
    }
}
//################################################################################
```

Save the code and press F7 to build it. If you have errors, review your code and try it again. Okay, let's run this puppy.

Running the SendDataTest Program

I know the following is a repeat, but the process must be burned into your memory. Open a command prompt (DOS window), change to the directory where you saved your project, and then go to the debug folder.

Run the program first with the following parameters.

```
SendDataTest server 6000
```

Now, open another command prompt, go to the debug folder again, and run the program with the following parameters. (Replace the 100.100.100.100 with your IP address.)

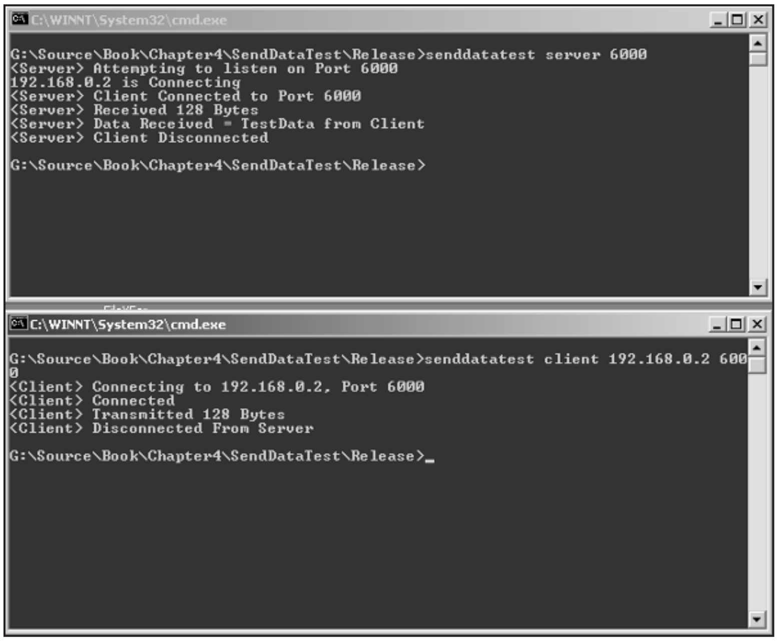
```
SendDataTest client 100.100.100.100 6000
```

If everything worked properly, you should see two output screens, as shown in Figure 4.7.

Check it out! Isn't that the coolest thing you have ever seen? I don't know about you, but I was very excited the first time I wrote a network program that was able to connect and send data over the Internet. You have just written a program that involves the entire purpose of network programming—transmitting and receiving data. Heck, I don't see any reason for me to discuss network programming any more now that you know how to send data. I've always wanted to write about the migratory habits of African swallows, so here goes. . . . On second thought, I haven't covered about 600 pages worth of information. I'd better continue.

Put your train of thought back on track, and take a look at the output from the programs. You should notice that the following events transpired.

1. The server bound to port 6000 to listen for connections.



```
C:\WINNT\System32\cmd.exe

G:\Source\Book\Chapter4\SendDataTest\Release>senddatatest server 6000
<Server> Attempting to listen on Port 6000
192.168.0.2 is Connecting
<Server> Client Connected to Port 6000
<Server> Received 128 Bytes
<Server> Data Received = TestData from Client
<Server> Client Disconnected

G:\Source\Book\Chapter4\SendDataTest\Release>
```

```
C:\WINNT\System32\cmd.exe

G:\Source\Book\Chapter4\SendDataTest\Release>senddatatest client 192.168.0.2 600
0
<Client> Connecting to 192.168.0.2, Port 6000
<Client> Connected
<Client> Transmitted 128 Bytes
<Client> Disconnected From Server

G:\Source\Book\Chapter4\SendDataTest\Release>_
```

Figure 4.7

Output from the server and client programs

2. The server listened for a connection to use port 6000.
3. The client attempted to connect to port 6000 on the server.
4. The server accepted the request.
5. The client sent a packet of 128 bytes to the server.
6. The server received a data packet consisting of 128 bytes from the client.
7. The client disconnected.
8. The server disconnected.

Now that you have seen the program, it is time to walk through the code that made this magic possible.

Walking through the SendDataTest Code

This program operates in the same way as the ConnectionTest program, as far as the `main()` function is concerned, so I'll skip talking about it. Here is the skinny on the `vServerConnection()` and `vClientConnection()` functions, which did change.

Inside SendDataTest's vServerConnection() Function

When the `vServerConnection()` function is invoked from the `main()` function, it starts off by creating two `SocketObject` objects, the `ServerSocketObject` and the `ClientSocketObject`. This part has not changed from the ConnectionTest program. Two new variables are introduced, though, `DataPacket` and `iBytesReceived`.

The variable `DataPacket` is nothing more than a character array to hold received data. The second variable, `iBytesReceived`, stores how many bytes are pulled from the `receive` command.

```
SocketObject     ServerSocketObject;
SocketObject     ClientSocketObject;
char             DataPacket[128];          // Data packet to receive
int              iBytesReceived = 0;       // # of bytes received
```

The next chunk of code is identical to that of the ConnectionTest program. Basically, the server binds to a listening port, listens for a connection, and then assigns the connection to the port.

```
if ( ServerSocketObject.Bind( iListenPort ) ) {
     cout << "<Server> Listening" << endl;
     // Listen for connection on the Listen port
     ServerSocketObject.Listen( );
     // Accept the connection
     ServerSocketObject.Accept( ClientSocketObject );
     cout << "<Server> Client Connected to Port " << iListenPort << endl;
...
```

You should be most interested in the member function `Recv()`, listed next. The `Recv()` member function is invoked using the `ClientSocketObject`. The function prototype follows.

```
int Recv(void* pBuf, int iBufLen, int iFlags);
```

The first parameter, `void* pBuf`, requires a pointer to a receive buffer. For the receive buffer, you use the `DataPacket` variable. The second parameter asks for the length of the buffer being used to receive data. Because you are allocating 128 bytes for the receive buffer, you pass a 128 to this parameter. The last parameter, `iFlags`, is used for various socket flag options. You are not using any special flags this time, so leave it at `0`.

```
// Receive data
iBytesReceived = ClientSocketObject.Recv(&DataPacket, 128, 0);
```

The Recv() function waits on data to be received. If the client does not transmit any data, the server is hung on this call infinitely. This is commonly referred to as a *blocking* call. I'll cover how you avoid this scenario in the next program. Refer to Figure 4.8 to see some blocking calls.

Once data is received, the function returns the number of bytes received. For this instance, the program assumes that the full 128 bytes it is expecting comes in one delivery. However, this is not the way to do it for real-world applications because you may or may not get the total amount of data requested. Okay, now that you have data, you should output it.

```
cout << "<Server> Received " << iBytesReceived << " Bytes" << endl;
cout << "<Server> Data Received = " << DataPacket << endl;
```

Because the DataPacket variable is nothing more than a character array, you can output it as any normal string. The number of bytes received is also displayed. After displaying the packet contents, the program disconnects the client connection and exits out of the program.

```
// Disconnect the client
ClientSocketObject.Disconnect();
cout << "<Server> Client Disconnected" << endl;
```

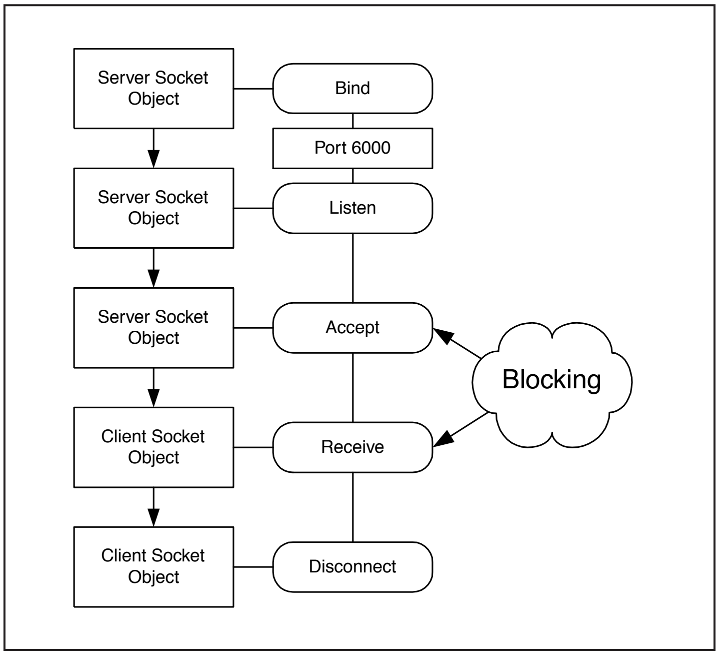


Figure 4.8

Blocking calls used so far

Basically, the only new function used is Recv(). Just remember that Recv() waits to get data, so be careful when using it or your program might hang forever. To recap, here is the list of functions used by the server program so far.

- Bind()
- Listen()
- Accept()
- Recv()
- Disconnect()

Inside SendDataTest's vClientConnection() Function

The vClientConnection() function still must create only one local SocketObject object, but now two new variables are defined. The two new variables are DataPacket and iBytesSent.

If you just read the "Inside SendDataTest's vServerConnection() Function" section, you'll notice that DataPacket is the same. It is a character buffer to hold data you want to transmit. The second variable, iBytesSent, is the inverse of the iBytesReceived used in the server function. iBytesSent stores how many bytes are successfully transmitted.

```
char     DataPacket[128];        // Data packet to transmit
int      iBytesSent = 0          // # of bytes sent
```

Please note that the DataPacket is not limited to 128 bytes; I am only using that size for efficient transmissions and consistency in the code examples. The next block of code demonstrates the familiar Connect() function used to connect to the server. It specifies the server's IP address and listening port number.

```
cout << "<Client> Connecting to " << szServerIP << ", Port "
  << iServerListenPort << endl;
// Connect to the IP and port
if( ClientSocketObject.Connect( szServerIP, iServerListenPort ) )
{
    cout << "<Client> Connected" << endl;
    ...
```

The first thing you do after you are connected to the server is specify what data you want to send with the standard strcpy() function. In this example, you are sending a small message to the server, “TestData from Client.” Feel free to play around with changing this text to something more interesting; just don’t exceed the 128-byte buffer size.

```
// Populate the data packet
strcpy(DataPacket, "TestData from Client");
```

Now that you have this wonderful and exciting message to send to the server, it is time to send it. You can accomplish this using the, take a guess here, `Send()` function!

```
// Send data
iBytesSent = ClientSocketObject.Send(DataPacket, 128, 0);
```

The `Send()` member function takes three parameters, just as the `Recv()` function does. The function prototype follows.

```
int Send(void* pBuf, int iBufLen, int iFlags);
```

The first parameter, `void* pBuf`, requires a pointer to a receive buffer. For the send buffer, you use the `DataPacket` variable. The second parameter asks for the length of the buffer being used to send the data. Because you are allocating 128 bytes for the send buffer, you pass 128 to this parameter. The last parameter, `iFlags`, is used for various socket flag options. You are not using any special flags this time, so leave it at zero.

Geez, didn’t that function sound familiar? However, unlike the `Recv()` function, the `Send()` function is not a blocking call. So, the program continues right after sending the data. Refer to Figure 4.9 for a diagram of the `SendDataTest` server function.

```
cout << "<Client> Transmitted " << iBytesSent << " Bytes" << endl;
// Disconnect from the server
ClientSocketObject.Disconnect();
cout << "<Client> Disconnected From Server" << endl;
```

Last, you output how many bytes were sent and then disconnect from the server. Here is a list of the functions used by the client program so far.

- `Connect`
- `Send`
- `Disconnect`

Wahoo! That program is out of the way, and you can now work on something a little more interactive. If you haven’t taken a break yet, you might want to do so now; the next program gets more involved.

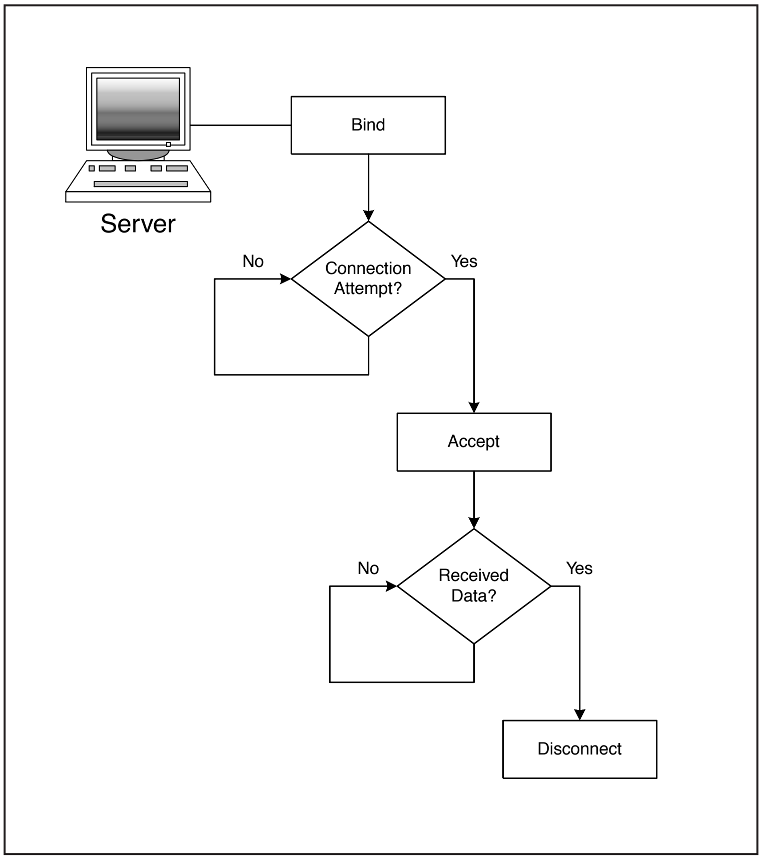


Figure 4.9

The `SendDataTest` *server function*

Writing a Simple Chat Program

Now that we have covered those two basic programs, I think it is time to write something more entertaining. I am going to show you how to write a chat program that lets a client and server send data back and forth. This is commonly referred to as *two-way communication*. Actually, the first two programs communicated back and forth, just not in a manner you could see. The chat program is still a console application, so it's not very fancy. It follows the usual idea of the simpler, the better. This program uses a new function from the `SocketObject` arsenal, `vGetPacket()`. This member function retrieves any packet data in the receive buffer. It allows you to receive data without waiting for it. So `vGetPacket()` is a non-blocking version of

Recv(). Cool, eh? Another new item you will see is the _kbhit() function, which checks the console for keyboard input. It does not wait for input, so it comes in handy when checking for incoming data. If that sounds a bit confusing, don't worry; you'll see what I am talking about in a little while.

NOTE

You might have noticed that the _kbhit() function has an underscore in front of its name. With Microsoft's Visual C++ compiler, all functions with underscores in front of them are specific to the compiler. The repercussion is that these functions might not work in other compilers or operating systems. If you decide to use other compilers or operating systems, you need to research equivalent commands.

Components Needed for the Chat Program

This project is almost identical to the ConnectionTest and SendDataTest programs, so I will be brief about the contents. The components needed for the chat program can be seen in Table 4.5.

Follow the necessary steps outlined in the "Getting Started with the Compiler" section and create a new project named SimpleChat.

Table 4.5 Components for the Chat Program

Component	Purpose
SIMPLECHAT.CPP	The main body of the program that handles both the client and server code. This is the only code that I examine for this example.
SOCKETOBJECT.CPP	The class file for the sockets functions, which holds all of the member functions in the SocketObject class.
SOCKETOBJECT.H	The class file for the sockets functions, which holds all of the header information for the SocketObject class.
WS2_32.LIB	The Windows library that holds all of the low-level sockets functions.
WINSOCK.H	The header to the Windows ws2_32.lib library.

Writing the SimpleChat Program

You can either type in the following code or copy it from the CD that came with the book. I still recommend that you type in most of the code samples; it will help you become more familiar with the functions.

```
// Standard includes
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include "..\\SocketObject\\SocketObject.h"
// Packet for sending a chat message
struct stChatPacket
{
    stPacketHeader     stHeader;
    char               szChatMessage[128];
};
// Function prototypes
void vServerConnection( int iListenPort );
void vClientConnection( char *szServerIP, int iServerListenPort );
// ----> Main Program Function (REQUIRED)
int main( int argc, char *argv[] )
{
    if( argc < 3 ) {
        cout << "----------------------------------------" << endl;
        cout << "                SimpleChat Help             " << endl;
        cout << "----------------------------------------" << endl;
        cout << "Usage: SimpleChat Help [client/server]  [ip,port/port,port]" << endl;
        cout << "" << endl;
        cout << "Example: SimpleChat Help client 198.168.0.1 6000" << endl;
        cout << "" << endl;
        cout << "Example: SimpleChat Help server 6000" << endl;
        cout << "" << endl;
        return( 0 );
    }
    // If user selected server, listen on the given port
    if( !stricmp( argv[1], "server" ) ) {
        vServerConnection( atoi( argv[2] ) );
    }
    // User selected client; connect to given port and IP address
    else {
        vClientConnection( argv[2], atoi( argv[3] ) );
    }
    return( 1 );
}

// Function for server
void vServerConnection( int iListenPort )
{
    SocketObject       ServerSocketObject;      // Server socket object
    SocketObject       ClientSocketObject;      // Client socket object
    char                DataPacket[128];        // Data packet to transmit
    int                iBytesReceived = 0;      // # of bytes received
    int                iBytesSent = 0;          // # of bytes sent
    char                szCommand[128];         // Chat buffer
    char                szTempChar;
    int                i;
    stChatPacket       ChatPacket;
    char               szPacketBuffer[32768];

    cout << "<Server> Attempting to listen on Port " << iListenPort << endl;
    // Attempt to start the server
    if ( ServerSocketObject.Bind( iListenPort ) ) {
    // Listen for connection on the Listen port
        ServerSocketObject.Listen( );
        // Accept the connection
        ServerSocketObject.Accept( ClientSocketObject );
        cout << "<Server> Client Connected to Port " << iListenPort << endl;
        // Loop forever or until a break is issued
        while( 1 ) {
            printf("\n<Server> ");
            // Loop through getting up to 127 characters to transmit,
            // and leave room for \n code
            for( i = 0; i < 127; i++ ) {
                // Loop until a key is pressed
                while( !_kbhit() ) {
                    // Check for incoming data
                    iBytesReceived = ClientSocketObject.vGetPacket(szPacketBuffer);
                    if( iBytesReceived > 0 ) {
                        // Copy the received data into
                        // the chat packet
                        memcpy(&ChatPacket,szPacketBuffer,sizeof(ChatPacket));
                        strcpy(DataPacket,ChatPacket.szChatMessage);
                        if( !stricmp( DataPacket, "/exit" ) ) {
                            break;
                        }
                        else {
                            cout << endl << "<Client> "<< DataPacket << endl;
                            printf("<Server> ");
                        }
                    }
                }
                // Quit chatting if /exit command given
                if( !stricmp( DataPacket, "/exit" ) ) {
                    break;
                }
                // Get next key from keyboard input buffer
                szTempChar = getche();
                // Add the keypress to our command string
                szCommand[i] = szTempChar;
                // Check for Enter key
                if( szTempChar == 13 ) {
                    break;
                }
            }
            // Check for user-initiated exit code
            if( !stricmp( DataPacket, "/exit" ) ) {
                break;
            }
            // Null terminate the command
            szCommand[i] = '\0';
            // Populate the packet
            strcpy(DataPacket,szCommand);
            // Quit chatting if /exit command given
            if( !stricmp( szCommand, "/exit" ) ) {
                break;
            }
            else {
                // Load the packet data
                ChatPacket.stHeader.iID = 0;
                ChatPacket.stHeader.iType = 0;
                ChatPacket.stHeader.iLength = 128;
```

```
                    ChatPacket.stHeader.iSender = 0;
                    strcpy(ChatPacket.szChatMessage,DataPacket);
                    ChatPacket.stHeader.iCheckSum = 128;
                    // Load the packet buffer
                    memcpy(szPacketBuffer,&ChatPacket,sizeof(stChatPacket));
                    // Send the data
                    iBytesSent = ClientSocketObject.Send(szPacketBuffer,sizeof
(stPacketHeader)+128,0);
                    // Check if client has disconnected
                    if( iBytesSent == -1 ) {
                        cout << "<Server> Client No Longer Online"<< endl;
                        break;
                    }
                }
            }
            // Disconnect the client
            ClientSocketObject.Disconnect();
            cout << "<Server> Client Disconnected" << endl;
        }
        else {
            cout << "<Server> Failed to Listen" << endl;
        }
}

// Function for client
void vClientConnection( char *szServerIP, int iServerListenPort )
{
    SocketObject       ClientSocketObject;         // Client socket object
    char               DataPacket[128];            // Data packet to transmit
    int                iBytesReceived = 0;         // # of bytes received
    int                iBytesSent = 0;             // # of bytes sent
    char               szCommand[128];             // Chat buffer
    char               szTempChar;
    int                i;
    char               szPacketBuffer[32768];
    stChatPacket       ChatPacket;

    cout << "<Client> Connecting to " << szServerIP << ", Port "
     << iServerListenPort << endl;
    // Connect to the IP and port
    if( ClientSocketObject.Connect( szServerIP, iServerListenPort ) ) {
```

```
cout << "<Client> Connected" << endl;
// Keep chatting until a /exit is given
while( 1 ) {
    printf("\n<Client> ");
    // Loop through getting up to 127 characters to transmit,
    // and leave room for \n code
    for( i = 0; i < 127; i++ ) {
        // Loop until a key is pressed
        while( !_kbhit() ) {
            // Check for incoming data
            iBytesReceived =
               ClientSocketObject.vGetPacket(szPacketBuffer);
            if( iBytesReceived ) {
                // Copy the received data
                // into the chat packet
                memcpy(&ChatPacket,szPacketBuffer,
                 sizeof(ChatPacket));
                strcpy(DataPacket,ChatPacket.szChatMessage);
                cout << endl << "<Server> "
                 << DataPacket << endl;
                printf("<Client> ");
            }
        }
        // Get next key from keyboard input buffer
        szTempChar = getche();
        // Add the keypress to our command string
        szCommand[i] = szTempChar;
        // Check for Enter key
        if( szTempChar == 13 )
            break;
    }
    // Null terminate the command
    szCommand[i] = '\0';
    // Populate the packet
    strcpy(DataPacket,szCommand);
    // Quit chatting if /exit command given
    if( !stricmp( szCommand, "/exit" ) ) {
        break;
    }
    else {
        // Load the packet data
```

```
                ChatPacket.stHeader.iID = 0;
                ChatPacket.stHeader.iType = 0;
                ChatPacket.stHeader.iLength = 128;
                ChatPacket.stHeader.iSender = 0;
                strcpy(ChatPacket.szChatMessage,DataPacket);
                ChatPacket.stHeader.iCheckSum = 128;
                // Load the packet buffer
                memcpy(szPacketBuffer,&ChatPacket,sizeof(stChatPacket));
                // Send the data
                iBytesSent = ClientSocketObject.Send(szPacketBuffer,sizeof
(stPacketHeader)+128,0);
                // Check if client has disconnected
                if( iBytesSent == -1 ) {
                    cout << "<Server> Client No Longer Online" << endl;
                    break;
                }
                // Check for server disconnection
                if( iBytesSent == -1 ) {
                    cout << "<Client> Server No Longer Online" << endl;
                    break;
                }
                // Quit chatting if /exit command given
                if( !stricmp( szCommand, "/exit" ) ) {
                    break;
                }
            }
        }
        // Disconnect from the server
        ClientSocketObject.Disconnect();
        cout << "<Client> Disconnected From Server" << endl;
    }
    else {
        cout << "<Client> Failed to Connect" << endl;
    }
}
```

Save the code and press F7 to build it. If you have errors, review your code and try again. Now that everything has been compiled, take the program out for a spin.

Running the SimpleChat Program

Is this all starting to sound familiar yet? What's that? You forgot the steps? Here, I'll show you again. Open a command prompt (DOS window), change to the directory where you saved your project, and then go to the debug folder. Run the program first with the following parameters.

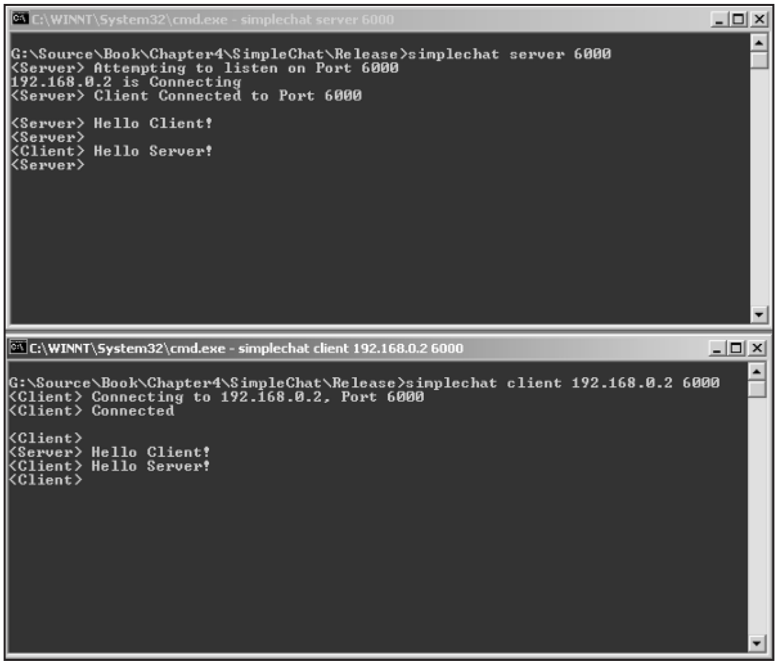
```
SimpleChat server 6000
```

Now, open another command prompt, go to the debug folder again, and run the program with the following parameters. (Replace the 100.100.100.100 with your IP address.)

```
SimpleChat client 100.100.100.100 6000
```

If everything went well, you should see two output screens, as shown in Figure 4.10.

One thing you might notice is a blinking cursor in the client program's window. You see it because you can type into that window and send the text to the server. Go ahead—type **Hello Server!** and press Enter. If everything went according to plan, you should see your message pop onto the server program's output. Wasn't



```
C:\WINNT\System32\cmd.exe - simplechat server 6000

G:\Source\Book\Chapter4\SimpleChat\Release>simplechat server 6000
<Server> Attempting to listen on Port 6000
192.168.0.2 is Connecting
<Server> Client Connected to Port 6000

<Server> Hello Client!
<Server>
<Client> Hello Server!
<Server>
```

```
C:\WINNT\System32\cmd.exe - simplechat client 192.168.0.2 6000

G:\Source\Book\Chapter4\SimpleChat\Release>simplechat client 192.168.0.2 6000
<Client> Connecting to 192.168.0.2, Port 6000
<Client> Connected

<Client>
<Server> Hello Client!
<Client> Hello Server!
<Client>
```

Figure 4.10

Output from the SimpleChat program

that cool? You just interactively sent a message to the server! Okay, now switch to the server program's window, and you'll see the familiar blinking cursor. Go ahead and type **Hello Client!** into the window and press Enter. Even more impressive, eh? Now you have communicated back to the client. Play around with typing back and forth from the client to the server.

The program is lacking in the area of looks, and you might notice that messages being typed can get interrupted. Also, the Backspace key does not move the cursor. These are minor issues, but by not accounting for them, you have been able to keep the code clear. If you are feeling adventurous, alter the program to account for the Backspace key and other minor issues.

This is a fun program to try with friends online. Send the program to a buddy and play around with chatting back and forth over the Internet. I found it very exciting the first time I sent this program to a friend and established successful two-way communication. You actually have the beginnings of an IRC (*Internet Relay Chat*) program. But before you go off editing the program to do all sorts of strange things, I'd better show you how it works.

Walking through the SimpleChat Code

The SimpleChat program is similar to the first two programs as far as structure goes. But I added some new functions, so with that comes new header information.

Header Information

For the program to use the new functions, you have to include a couple additional header files. The conio.h file contains console input/output functions. You need this file for the `_kbhit()` and `getche()` functions. The second new header file you use is stdio.h, which is necessary for you to use the `printf()` function in the program.

```
// Standard includes
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include "..\\SocketObject\\SocketObject.h"
struct stChatPacket
{
     stPacketHeader       stHeader;
     char                 szChatMessage[128];
};
```

SimpleChat works the same way as the first two programs in this chapter as far as the `main()` function is concerned, so I'll skip talking about it. But there is one thing new here—the data structure called `stChatPacket`.

The Chat Packet Structure

The `stChatPacket` data structure is used by the chat program to send a chat message across the wire. If you look closely at the structure, you'll see that the first data member is an `stPacketHeader`. This is defined in the SocketObject.h file that I provided as part of the project. The `stPacketHeader` object has the following layout.

```
struct stPacketHeader
{
    int          iType;
    int          iLength;
    int          iID;
    int          iCheckSum;
    int          iSender;
};
```

The first member is an integer that can hold the type of the packet being sent. The second member specifies the length of the packet, which allows for variable-length packets. The third parameter is the ID of the packet. This is optional, and you can set it to zero if you want. The fourth parameter is the checksum. The SocketObject.cpp file expects the checksum to equal the length plus the type plus the ID. If the checksum does not equal this, then the packet is discarded. The last parameter holds the ID of the sender.

This data structure both sends and receives data in the chat program. Without further ado, I present the `vServerConnection()` and `vClientConnection()` functions.

Inside SimpleChat's vServerConnection() Function

The `vServerConnection()` function now has a few new variables declared upon its execution. The first thing you might notice is that the server now has variables for both sent and received bytes, `iBytesSent` and `iBytesReceived`. In the SendDataTest program, the server only had to track `iBytesReceived` because it was only listening for data. Now that you have two-way communication, the server must keep track of both.

```
SocketObject     ServerSocketObject;         // Server socket object
SocketObject     ClientSocketObject;         // Client socket object
char             DataPacket[128];            // Data packet to transmit
```

```
int             iBytesReceived = 0;       // # of bytes received
int             iBytesSent = 0;            // # of bytes sent
char            szCommand[128];           // Chat buffer
char            szTempChar;
int             i;
stChatPacket    ChatPacket;
char            szPacketBuffer[32768];
```

The next new variable is `szCommand`. This character buffer keeps track of what the user on the server program is typing. Every character typed is stored in `szCommand` until it is ready for transmission. `szTempChar` is a temporary placeholder for information returned from the keyboard. The `ChatPacket` variable converts data saved into or read from the `szPacketBuffer`. The `i` variable is used for looping.

The next portion of code should look familiar to you. Nothing has changed in the way you connect to or disconnect from the client. The server binds to a port, listens for a new connection on it, and then assigns the connection to the port.

```
if ( ServerSocketObject.Bind( iListenPort ) ) {
     // Listen for connection on the listen port
     ServerSocketObject.Listen( );
     // Accept the connection
     ServerSocketObject.Accept( ClientSocketObject );
     ...
}
```

What happens next should be new to you. The next lines of code put the program into a possible never-ending loop using a while loop.

```
// Loop forever or until a break is issued
while( 1 ) {
     ...
}
```

Next, you start off a loop that runs for 127 iterations. This loop lets the user enter up to 127 characters to send to the client. You use 127 because the data packet is only 128 bytes, and you need to leave room for a null terminator. The user does not have to enter all 127 characters, though; this can be interrupted with a break command.

CAUTION

Be very careful when using a never-ending loop command such as `while(1)`. It can cause your program to hang indefinitely if you do not make sure to include a break within it.

```
// Loop through getting up to 127 characters to transmit,
// and leave room for \n code
for( i = 0; i < 127; i++ ) {
...
}
```

During the stay in the for loop, you should do something useful, such as wait for a keypress. Normally, you might think you could simply ask for input from the keyboard, wait for a string to come in, and then send it across the wire. This procedure does not work in this case because you need to check whether the client sent data while you were typing. The result wouldn't be pretty if you only saw text from the client after you sent something. So you have to use a non-blocking keyboard function that does not stop the program while waiting for a keypress. You use the `_kbhit()` function I mentioned earlier. Refer to Figure 4.11 to see the flow of the server function and then look at the next piece of code.

```
// Loop until a key is pressed
while( !_kbhit() ) {
    // Check for incoming data
    iBytesReceived = ClientSocketObject.vGetPacket(szPacketBuffer);
    if( iBytesReceived > 0 ) {
        // Copy the received data into the chat packet
        memcpy(&ChatPacket,szPacketBuffer,sizeof(ChatPacket));
        strcpy(DataPacket,ChatPacket.szChatMessage);
        if( !stricmp( DataPacket, "/exit" ) ) {
            break;
        }
        else {
            cout << endl << "<Client> " << DataPacket << endl;
            printf("<Server> ");
        }
    }
}
```

The program hangs out in a while loop until a key is pressed. Within the loop, it checks for a received packet.

If the return code from `vGetPacket()` is greater than zero, the program knows something was transmitted, so it outputs the data by first copying the raw message into the `stChatPacket` format. If the program used more than one type of packet, you might have checked the type first. But because the program uses only the `stChatPacket` type,

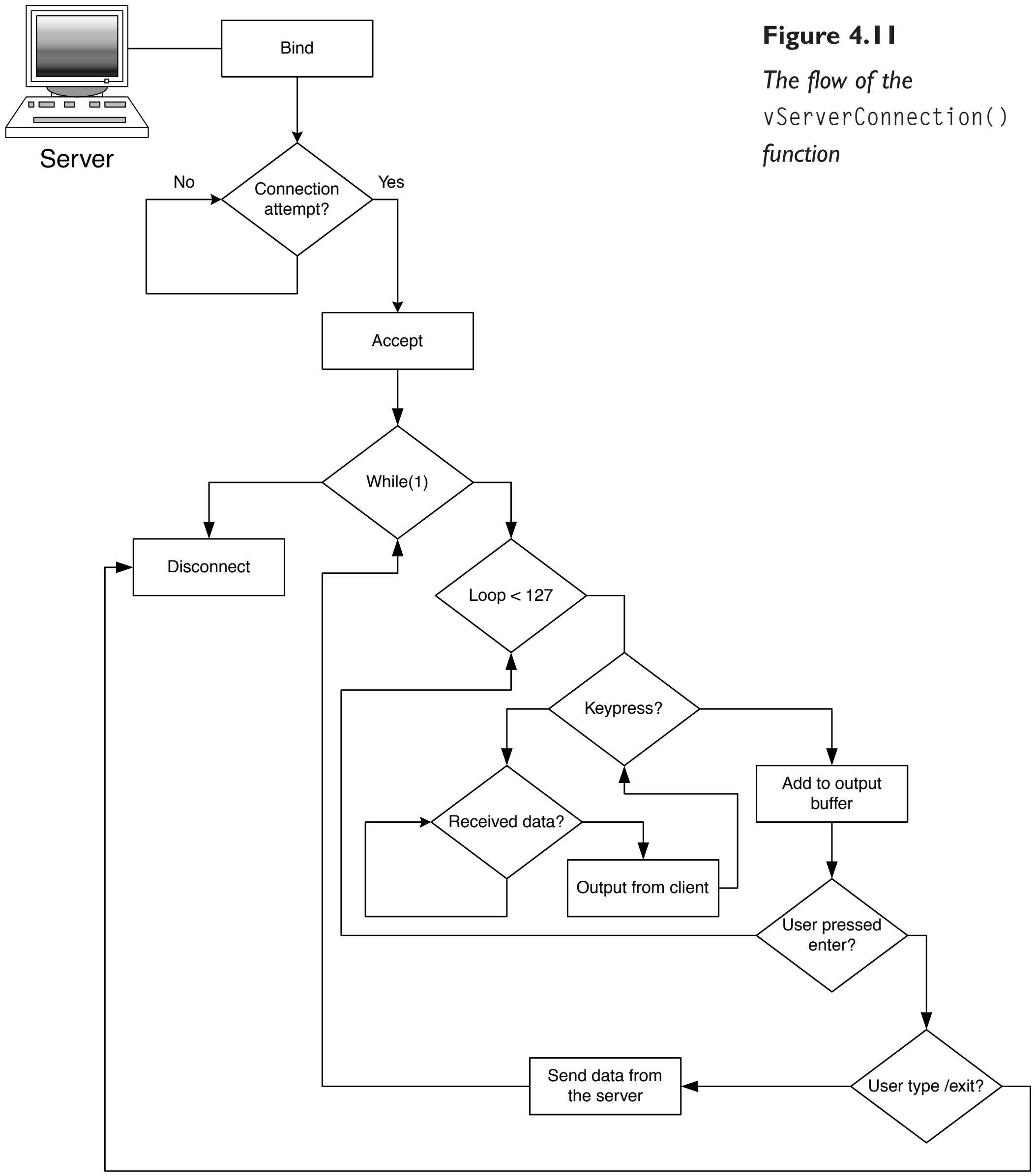


Figure 4.11

The flow of the `vServerConnection()` *function* you can just go ahead and load it in. To output the chat message, the code goes on to copy the `szChatMessage` member of the packet to the `DataPacket` array.

There you have it: Your program just allowed the server's user to enter data while accepting incoming information. If the user presses Enter, the program exits out of the for loop and sends the message to the client, using the `Send()` member function of the `ClientSocketObject`. If the returned code from the send command is `-1`, you know that the client is no longer connected. Because the client is no longer connected, you end the communication using the `Disconnect()` function.

```
    ...
    // Load the packet data
    ChatPacket.stHeader.iID = 0;
    ChatPacket.stHeader.iType = 0;
    ChatPacket.stHeader.iLength = 128;
    ChatPacket.stHeader.iSender = 0;
    strcpy(ChatPacket.szChatMessage,DataPacket);
    ChatPacket.stHeader.iCheckSum = 128;
    // Load the packet buffer
    memcpy(szPacketBuffer,&ChatPacket,sizeof(stChatPacket));
    // Send the data
    iBytesSent = ClientSocketObject.Send(szPacketBuffer,
     sizeof(stPacketHeader)+128,0);
    // Check if client has disconnected
    if( iBytesSent == -1 ) {
        cout << "<Server> Client No Longer Online" << endl;
        break;
    }
}
// Disconnect the client
ClientSocketObject.Disconnect();
```

Take note that sending is just the opposite of receiving as far as packet initialization goes. To send a packet, you first set up the header information. Then you load the meat of the packet—the chat message, in this case. After those two steps, you copy the packet into a sending buffer and send it on its way.

There you have it! That's all there is to the server's function call. Now I'll review the steps the function follows.

1. Bind to a port.
2. Listen on that port for new connection.

3. Accept the new connection.
4. Loop until the server or client exits.
5. Check for messages from the client.
6. Send messages to the client.
7. Disconnect from the client.

Inside SimpleChat's vClientConnection() Function

Now I want talk about the client function. The client function is almost identical to the server function; the main difference is that the client must only make a call to the `Connect()` function to establish the connection.

```
// Connect to the IP and port
if( ClientSocketObject.Connect( szServerIP, iServerListenPort ) ) {
    ...
}
```

Everything else that happens in the function works as it does for the server, so you are done here. I bet you didn't expect it to be that quick, did you?

Review

In this chapter, I covered the basics for using the `SocketObject` class. Even though you now know how to connect, send, receive, and disconnect, you still have much to learn. The `SocketObject` class is designed to give you a good foundation for sockets programming, and I think you have gained that from this chapter. I suggest that you now try writing server and client programs from the ground up. If you can recreate all that was covered here, then you are off to a great start. Add some of these features to the sample programs:

- Support for more than two persons chatting
- An enhanced user interface
- Enhanced error checking

I'm off to turn my house key in *Ultima Online* for now, so see you in the next chapter!

CHAPTER 5

Creating a Sockets Class Object

I never have liked game development books that wrapped all of the functionality into classes without showing you how to do the actual work. To save this book from that failure, I am going to show you how to create an entire sockets class from start to finish. No stone will be left unturned. In the preceding chapter, I covered how to use the class you are about to write. I know this might seem a little strange or backward, but I wanted you to understand the class before you wrote it.

To begin with, I'll cover how to create functionality using standard C without classes. Then, after the functionality has been created, I'll cover how to add it to your class object. Think of it as getting the best of both worlds. All in all, you will learn about:

- Constructors
- Advantages to C++
- Accepting client connections
- Connecting to a server
- Disconnecting from a session
- Sending data
- Receiving data

C++ Classes in All Their Glory

Maybe you never have used classes or you just don't use them often in your code. I hope I can persuade you to use them in the future. C++ is a great improvement over straight C in many ways. The only thing that kept me from using C++ several years ago was that I could never find out what it really meant. I bought all sorts of books on object-oriented theory and programming, but all of them completely confused me. After all, what the heck are objects, and why do you need to use them?

An Overview of C++ Smaller than a Fly Burp

The most basic definition I have ever used for C++ is that C++ is nothing more than C style structures with functions. That's it: *C++ is just data structures with functions*! I realize this does not jibe with several object prophets, but hey, it gets the point across. I could spend countless pages describing to you the theory and methods behind classes, but this is not a book about C++; it is a book about game programming! So what do I mean by my definition of C++? Well, I've always said the best way to teach something is with an example, so here goes.

Imagine a game where you have to keep track of several monsters. In this game, you have to store their positions, hit points, statuses, and types. In old-school C, you create a data structure to hold this information, like this example.

```
struct MonsterData
{
    int x,y;            // Position
    int hitpoints;
    int type;
    int status;         // 0 = dead, 1 = alive
};
```

After you create the structure, you then create an array to hold all of the monsters in the game.

```
MonsterData     Monsters[100];     // 100 monsters in game
```

Now, just imagine a situation in the game where you have to deduct from the monster's hit points. Suppose that this situation occurs when a player strikes the monster. Not only do you have to deduct the hit points, but you also have to check whether the creature dies as a result of the injury. To accomplish this in C, you might create a function to do something like this:

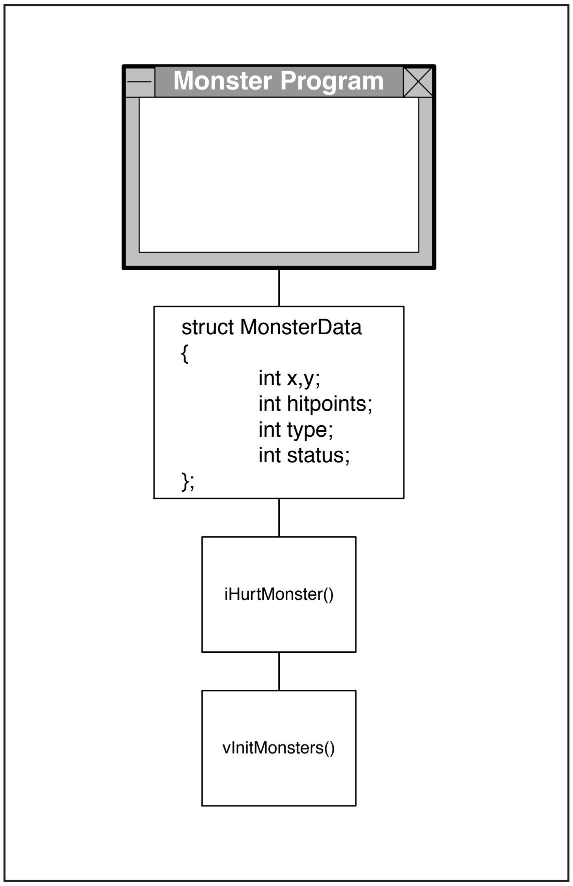
```
int iHurtMonster(int monster, int damage)
{
    Monsters[monster].hitpoints -= damage;
    // Check whether monster died
    if(Monsters[monster].hitpoints <= 0 ) {
        Monsters[monster].status = 0;
        return(1);
    }
    return(0);
}
```

Oops, I almost forgot to initialize the monsters. To initialize the monsters, you create some sort of initialization routine like this:

```
void vInitMonsters(void)
```

```
{
     int iLoop;
     for(iLoop = 0; iLoop < 100; iLoop++ ) {
          Monsters[iLoop].status = 1;
          Monsters[iLoop].x = Monsters[iLoop].y = 0;
          Monsters[iLoop].type = 0;
          Monsters[iLoop].hitpoints = 250;
     }
}
```

Alrighty, now you have what you need to handle your game logic with a structure and two functions. Look at Figure 5.1 to see the structure of the program so far.



```
struct MonsterData
{
          int x,y;
          int hitpoints;
          int type;
          int status;
};
```

Figure 5.1

The Monster Program layout

The code so far looks something like this:

```
#include <stdio.h>
struct MonsterData
{
    int x,y;          // Position
    int hitpoints;
    int type;
    int status;       // 0 = dead, 1 = alive
};
// Global data
MonsterData     Monsters[100];     // 100 monsters in game
// Prototypes
int iHurtMonster(int monster, int damage);
void vInitMonsters(void);

// Main function
void main(void)
{
    vInitMonsters();
    // Do some player logic here; detect a monster getting hit
    iHurtMonster(monsterid,damageinflicted);
}
int iHurtMonster(int monster, int damage)
{
    Monsters[monster].hitpoints -= damage;
    // Check whether monster died
    if(Monsters[monster].hitpoints <= 0 ) {
        Monsters[monster].status = 0;
        return(1);
    }
    return(0);
}
void vInitMonsters(void)
{
    int iLoop;
    for(iLoop = 0; iLoop < 100; iLoop++ ) {
        Monsters[iLoop].status = 1;
        Monsters[iLoop].x = Monsters[iLoop].y = 0;
        Monsters[iLoop].type = 0;
```

```
            Monsters[iLoop].hitpoints = 250;
        }
    }
```

Okay, now that the C code is out of the way, take a look at how it is done in C++. First, you create a class object to handle monster functionality. Once again, you need to store the position, hit points, status, and monster type. You also need functions to handle when monsters are damaged and when you initialize them. Here is what the class looks like before adding the functions.

```
class MonsterObject
{
    public:
        int     iXPos,iYPos;
        int     iHitPoints;
        int     iStatus;
        int     iType;
        MonsterObject();                // Constructor
        // Function for monster to take damage
        int iInflictDamage(int damage);
};
```

You might notice that the only differences between the preceding class and the C data structure are the functions. As I said before, the main thing about C++ is that it allows you to have functions within structures. So no big deal so far, right? Just as you did in the C version of the code, you also declare an array of 100 monsters.

```
MonsterObject     Monsters[100];
```

Now you have to create the function for when monsters take damage. Instead of creating a global function for use by everyone, just create a function local to the class object. Here is what the function looks like.

```
int MonsterObject :: iInflictDamage(int damage)
{
    iHitPoints -= damage;
    // Check whether monster died
    if(iHitPoints <= 0 ) {
        iStatus = 0;
        return(1);
    }
    return(0);
}
```

The main thing to note is that the variables are local to the class object. This is why you do not need to pass the monster's ID to the function. Here is how you call the function.

```
Monsters[monsterid].iInflictDamage(damageinflicted);
```

You are actually calling the function that is owned by the monster object instead of calling a global function owned by everyone. Now that you have created the damage function, it is time to create the initialization function for your class object.

```
MonsterObject::MonsterObject(void)
{
    iStatus = 1;
    iXPos = IYPos = 0;
    iHitPoints = 250;
    iType = 0;
}
```

Where's the function name, right? This is not a function per se, but rather a class *constructor*. Because it is a constructor, it has no name to call it. A class constructor is called when a class object is declared and a *destructor* is called when a class object leaves scope. So in the example, the `MonsterObject::MonsterObject()` function is called as soon as you declare the global `Monsters` array. Using the class method, you don't have to write a loop to initialize the objects; they get initialized as soon as they are constructed!

A class *constructor* is called when a class object is created. A class *destructor* is called when a class is destroyed.

Take a peek at how the code looks using C++.

```
#include <stdio.h>
class MonsterObject
{
    public:
        int     iXPos,iYPos;
        int     iHitPoints;
        int     iStatus;
        int     iType;
        MonsterObject();                // Constructor
        // Function for monster to take damage
        int iInflictDamage(int damage);
};
MonsterObject::MonsterObject(void)
```

```
{
    iStatus = 1;
    iXPos = IYPos = 0;
    iHitPoints = 250;
    iType = 0;
}
int MonsterObject :: iInflictDamage(int damage)
{
    iHitPoints -= damage;
    // Check whether monster died
    if(iHitPoints <= 0 ) {
        iStatus = 0;
        return(1);
    }
    return(0);
}
MonsterObject  Monsters[100];
// Main function
void main(void)
{
    // Do some player logic here; detect a monster getting hit
    Monsters[monsterid].iHurtMonster(damageinflicted);
}
```

Get to the Point Already!

I hear ya: Get to the point already, right? Hey, that reminds me of the chat code in *Age of Empires II* that says something like “Start the game already!” Well, I like to program using classes because it can result in much cleaner code. Not only does the code normally look cleaner, but you also gain advantages from a smaller code base, code reusability, and code isolation.

Because classes allow you to write less redundant code, your programs usually end up smaller in size. Unless you work for one of the companies that pay programmers by the number of lines, you’ll like your code to be as small as possible. Now don’t get me wrong here: Any goof can write millions of lines of code, no matter what style he uses.

Ah, the holy hand-grenade of programmers—reusable code. C++ reeks of reusability. Not only can you layer classes upon classes, but you can also create virtual classes that apply to all sorts of data types. I can’t stress enough how well C++ gets reused. Because classes are nice tight little packages, you can move them around from code base to code base without fear of losing pieces—just like in the MonsterObject example. It is much easier to figure out what to reuse in the C++ version of the code than it is in the C version.

Because functions for classes are local to the classes themselves, you don't have tons of dangling functions in your program. This also points to cleaner code. It is much easier to tell what a class function belongs to because you can just look at its scope. The `MonsterObject` function was easy to pick out; all you had to notice was the `MonsterObject::` in front of it. Figure 5.2 shows the architectural differences between C and C++.

Oh, yes, code cleanliness. I'm not trying to win a sanitation award here, but I like clean and concise code. Clean and concise code is easy to read and easy to understand when you return to it a year later. Table 5.1 outlines some of the advantages I see with C++.

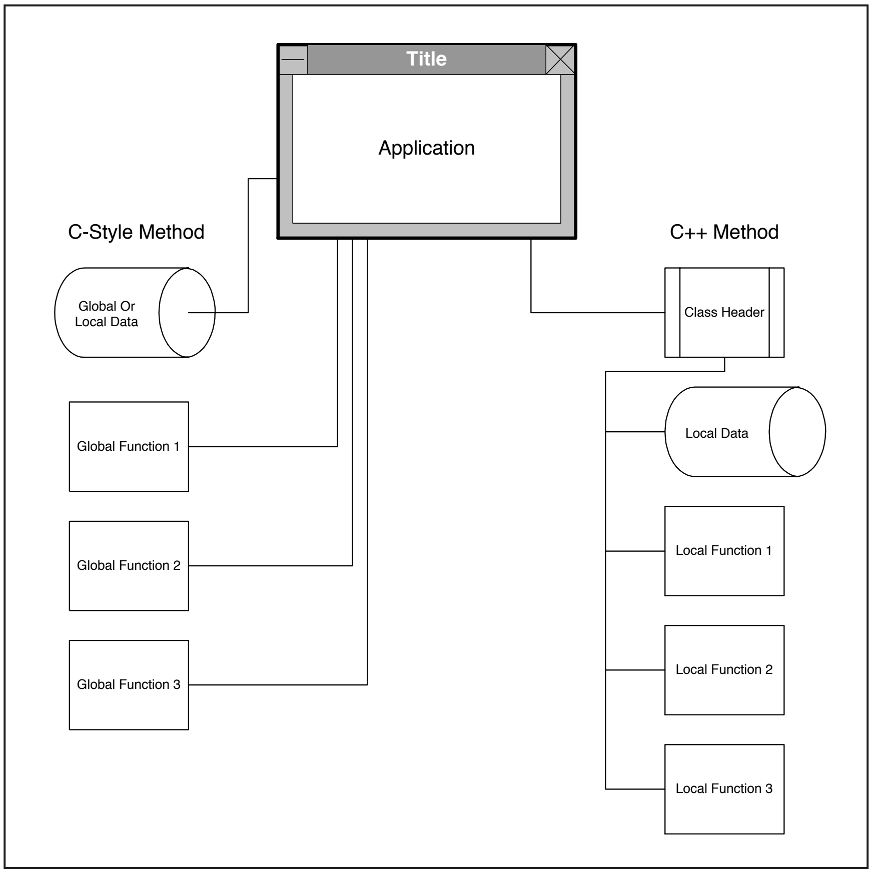


Figure 5.2

C-style structure versus C++ structure

Table 5.1 Results and Advantages of Using C++

Result	Advantage
Cleaner code	Easy for others to read and understand. Can be understood more easily when revisited.
Smaller code base	Less compile time, easier to follow, and with smaller executables.
Code reusability	Saves time on future projects because you can reuse previous code efforts.
Code isolation	Code is easy to remove or edit. Can more easily be transported to other projects.

Wow, what a quick trip down the yellow-brick road of C++. I know that I only briefly covered the topic, and believe me; I left out tons of information about C++. If you are hungry for more information, I suggest you find a highly rated book about C++ on the Internet or at a good bookstore. I do remember a very good book I once read called *Moving from C to C++* (Sams, 1992). It's a few years old, but buy it if you can find it. Another good resource is *1001 Visual C++ Programming Tips* (Prima, 2001).

Designing the SocketObject Class

Now that the virtues of C++ are out of the way, it's time to talk about the class I am going to show you how to create. The purpose of the class is to wrap all of the functionality for sockets into a nice, neat package. For all of the Dilbert fans out there, here is our *vision* statement.

> To create a C++ class that will provide all of the functionality used in sockets for both clients and servers.

All right, now that I have dispensed with the corporate mumbo-jumbo, take a look at what this actually means. Because you use sockets to facilitate communication between computers, the class needs to accomplish this, at the very least. The class also needs to provide the following functions, at a minimum, to be useful.

- Accept connections (server)
- Connect to a server (client)
- Disconnect (server and client)
- Send data (server and client)
- Receive data (server and client)

You will want to add other functions and features, but none of them are absolutely necessary to facilitate communication. Refer to Figure 5.3 to see the layout of the proposed socket class.

TIP

You can find all of the working source code listed in this chapter on the CD that comes with the book. Look in the Chapter5 Zip file on the CD. Each project consists of the source files along with the ws2_32.lib library file. You need the library file for socket functionality; always include this library in your socket projects.

The SOCKET Data Type

The first thing you have to do when creating a class object is design the class header information. First, all of the class functions deal with sockets; therefore, you need a `SOCKET` data type in the class for all of the socket operations.

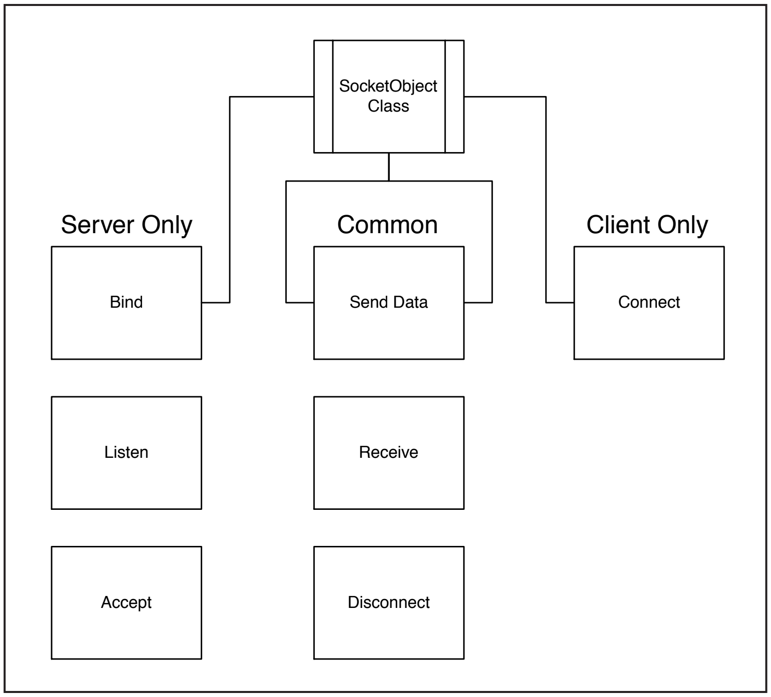


Figure 5.3

Layout of the C++ socket class

The SOCKET data type is really just like the Windows HWND data type. It is nothing more than a handle to a socket. Because it is a handle, it is really just an integer used as a reference number. You will only ever deal with two types of sockets, stream-based and datagram-based.

- *Stream* sockets provide data in a constant stream without breakpoints. Stream data is always guaranteed to be delivered in order. When you want guaranteed packet delivery, this is generally the method to use.
- *Datagram* sockets send data in blocks and do not guarantee the order or the delivery. Unless you plan to create your own guaranteed delivery mechanism, this is not the way to go. Datagrams are useful, though, for sending non-critical game data. Because the packets are not guaranteed, they take less transmission time.

Now that you know what the SOCKET data type is, here is how you implement it into the class.

```
class SocketObject
{
    private:
    public:
        SOCKET     skSocket;
}
```

From now on, my examples will use the custom Hungarian notation of sk for all SOCKET data types. Not too complicated yet, right? Okay, now that you have a handle for the socket, you need a function to use it. I say it's time to start with the bind functionality.

The Bind Functionality

I know I demonstrated using the bind() function in the last chapter, but let me talk about it a bit more here. Basically, the bind() function associates a socket with an address. When you first create a socket, it is not associated with anything. That is why you must use the bind() function; otherwise, the system would never know what you are trying to communicate with. For you to use the connect() or listen() function later, you must first call the bind() function. Here is what the prototype for bind() looks like.

```
int bind ( SOCKET s, const struct sockaddr FAR*  name, int namelen );
```

You should be familiar with the first parameter data type by now because it only requires an unbound socket handle. *Unbound* means that the socket has not already been bound with the `bind()` function.

The SOCKADDR Data Type

The second parameter looks a bit strange but really isn't all that bad. It just wants an address to store the information about the bind. All you do here is pass a temporary `sockaddr_in` data structure. `Sockaddr_in` is just a derivative of the `sockaddr` data structure used in specifying Internet addresses. Here is what the `sockaddr_in` data structure looks like.

```
struct sockaddr_in {
    short               sin_family;
    unsigned short      sin_port;
    struct in_addr      sin_addr;
    char                sin_zero[8];
};
```

This data structure holds all the information that is necessary for defining the address of an Internet connection. To do this, it must store important information such as the IP address and port. Table 5.2 shows a breakdown of the member variables.

Table 5.2 Member Variables

Variable	Purpose
`sin_family`	This is always set to `AF_INET`. Don't you love stuff like this? I think the men in black like to make us type a lot.
`sin_port`	This is the IP port to which you will be binding. Normal ranges are from 0 to 10000.
`sin_addr`	This is the IP address to which you will be binding or connecting. Actually, this uses a data structure of type `in_addr`. I'll cover this after I have finished with the `sockaddr_in` type.
`sin_zero`	This is nothing more than padding used to make `sockaddr_in` the same size as `sockaddr` for type-casting purposes.

As promised, I'll now talk about the in_addr data structure. It is defined in winsock.h and looks like this.

```
struct in_addr {
    union {
        struct {
            unsigned  char   s_b1,
                s_b2,
                s_b3,
                s_b4;
        }  S_un_b;
        struct {
            unsigned  short  s_w1,
                s_w2;
        }  S_un_w;
        unsigned long  S_addr;
    } S_un;
};
```

My first impression of this data type was. . . . Well, I don't think I can say that here. I don't want to muddy the waters, so I'm only going to discuss the part of this you will use, the S_un_b structure. This structure holds four character values. Each of these values represents a part of the IP address. Because IP addresses consist of four numbers between 0 and 255, this fits perfectly. If your IP address were 192.168.0.2, then the values in the structure would look like this:

```
s_b1 = 192;
s_b2 = 168;
s_b3 = 0;
s_b4 = 2;
```

Now it makes sense, right? Well, even if it doesn't, I'll cover this more later. Where was I, anyway? Oh, yes, I was talking about the parameters used in the bind() function. I covered the first two parameters, so that leaves me with the last parameter, which is the length of the sockaddr data structure. This is a very easy one to do; you pass sizeof(sockaddr) to make it work.

The last thing you need to worry about is what the bind() function returns. It returns a simple integer to be checked. The function returns a 0 if successful or a SOCKET_ERROR on failure. Just check for this value when calling to determine whether it failed.

Now that the bind() function is out of the way, I need to talk about the all-important socket() function.

The Socket() Function

Before you can even call the old bind() function, you must create the socket you want to use with the ambiguously named socket() function. (Just kidding about the ambiguous part!) Here is the prototype for this function.

```
SOCKET socket ( int af, int type, int protocol )
```

Ah, that looks easy enough—only three integers for parameters. The first parameter of type integer specifies the address family. What does this mean to you? Not much; just use the AF_INET definition for this.

The second parameter, also of type integer, specifies the type of socket you want to create. For our purposes, there are two types of sockets, stream sockets and datagram sockets. Déjà vu yet? This is where that lesson about streams and datagrams comes in handy. To specify a stream socket, use the SOCK_STREAM define. To specify a datagram socket, use the SOCK_DGRAM define.

The last parameter specifies the protocol to be used with the socket. Just pass a 0 to this for our purposes.

If the function succeeds as planned, then it returns the handle to a brand new socket. If it fails, it returns an INVALID_SOCKET code. Be sure to check for this return code when calling the socket() function.

The C Version of Binding

Are we there yet? Are we there yet? If this is starting to feel like a road trip with kids, then just hang in there. I am finally going to cover writing the bind() functionality. Using what you have learned already, you need to create the socket with the socket() function and then bind to an address using the bind() function. To help you better follow what is required of a server, take a look at Figure 5.4.

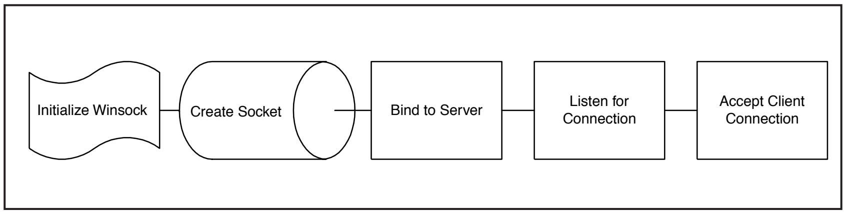


Figure 5.4

Steps required by a server to accept a client connection

Here is how you do it without using a class.

```
// Standard includes
#include <iostream.h>
#include <winsock.h>
#include <stdio.h>

// ----> Main Program Function (REQUIRED)
void main(void)
{
    SOCKET              skSocket;
    sockaddr_in         saServerAddress;
    int                 iPort = 6000,iStatus;
    WSADATA             wsaData;
    WORD                wVersionRequested;

    // Initialize the socket handle
    skSocket = INVALID_SOCKET;
    // Tell WinSock we want version 2
    wVersionRequested = MAKEWORD( 2, 0 );
    // Start up WinSock
    iStatus = WSAStartup( wVersionRequested, &wsaData );
    // Create the socket
    skSocket = socket( AF_INET, SOCK_STREAM, 0 );
    // Check whether there was an error
    if( skSocket == INVALID_SOCKET ) {
        cout << "**ERROR** Could Not Create Socket" << endl;
        // Clean up WinSock
        WSACleanup();
        exit(1);
    }
    cout << "<-- SOCKET CREATED -->" << endl;
    // Clear out the socket address structure
    memset(&saServerAddress, 0, sizeof(sockaddr_in));
    // Initialize the socket address structure
    saServerAddress.sin_family = AF_INET;
    saServerAddress.sin_addr.s_addr = htonl( INADDR_ANY );
    saServerAddress.sin_port = htons( iPort );
    // Attempt to bind
    if( bind( skSocket, (sockaddr*) &saServerAddress,
     sizeof(sockaddr) ) == SOCKET_ERROR ) {
        cout << "**ERROR** Could Not Bind" << endl;
```

```
            // Clean up WinSock
            WSACleanup();
            exit(1);
        }
        // Close the socket
        closesocket(skSocket);
        // Clean up WinSock
        WSACleanup();
        cout << "<-- SOCKET BOUND -->" << endl;
}
```

The first thing I did was include some standard libraries such as iostream.h and stdio.h. The most important include, though, is winsock.h. Winsock.h is required for all sockets programming you do. I then went on to create the `main()` function, which is standard to all console applications.

NOTE

All of the programs in this book that you'll write using sockets have the ws2_32.lib Winsock library linked in with them. Be sure to include this in your projects.

The variables you use for `main()` allow you to do all of the operations required, and are listed in Table 5.3.

Table 5.3 Variable Usage in the Main() Function

Variable	Purpose
`skSocket`	The `SOCKET` handle that points to the socket you are going to create.
`saServerAddress`	The `SOCKADDR_IN` to hold the socket address information.
`iPort`	An integer to hold the port number to which you are going to bind.
`iStatus`	An integer to hold the status of the call to `WSAStartup()`.
`wsaData`	A `WSAData` structure to hold information about the Winsock implementation.
`wVersionRequested`	A `WORD` to hold the version number of Winsock you want to use.

The next piece of code is quite simple; you merely set `skSocket` to equal `INVALID_SOCKET`. This sets up the socket handle to be used in the future. What happens next has not been covered so far.

Initializing Winsock with WSAStartup()

Before you can call any Winsock functions, you must first initialize Winsock by loading the ws2_32.dll. To load this dynamic link library, Windows provides you with the `WSAStartup()` function. Here is the prototype for the function.

```
int WSAStartup ( WORD wVersionRequested, LPWSADATA lpWSAData )
```

This function is really quite basic in that it only requires the version number of Winsock you want to use and a pointer to hold information about the Winsock implementation. So for the first parameter, pass in a `WORD` variable set with the `MAKEWORD()` function. For the second parameter, pass in the address of a `WSAData` structure. You don't ever really use the data structure, but here are the member variables for it anyway.

```
typedef struct WSAData {
      WORD                      wVersion;
      WORD                      wHighVersion;
      char                      szDescription[ WSADESCRIPTION_LEN+1 ];
      char                      szSystemStatus[ WSASYS_STATUS_LEN+1 ];
      unsigned short            iMaxSockets;
      unsigned short            iMaxUdpDg;
      char FAR                  *lpVendorInfo;
} WSADATA, FAR *LPWSADATA;
```

After Winsock is initialized, the program goes on to create the socket with the ever familiar `socket()` function I discussed in the section, "The Socket() Function." After you call the `socket()` function, you can go on to make sure it succeeded. Checking whether the socket equals `INVALID_SOCKET` tells you whether the socket call resulted in an error. If an error occurs, you need to clean up Winsock with the `WSACleanup()` function and then exit the program.

TIP

You can ignore the `iMaxSockets`, `iMaxUdpDg`, and `*lpVendorInfo` members of the `WSAData` structure. They exist only for backward compatibility with versions of Winsock prior to 2.0.

TIP

Always be sure to call the `WSACleanup()` function before exiting any Winsock-based programs.

Initializing the SOCKADDR_IN Data Structure

The next block of code initializes the `saSocketAddress` data structure. The call to `memset()` just sets all of the data in the structure to 0 to clean the slate.

```
// Clear out the socket address structure
memset(&saServerAddress, 0, sizeof(sockaddr_in));
// Initialize the socket address structure
saServerAddress.sin_family = AF_INET;
saServerAddress.sin_addr.s_addr = htonl( INADDR_ANY );
saServerAddress.sin_port = htons( iPort );
```

The htons() Function

The next three calls set up the data structure members. As I discussed before, the `sin_family` data member is always set to `AF_INET`. You then set up `sin_addr` for an IP address using the `htonl()` function. Here is the prototype for that function.

```
u_long htonl ( u_long hostlong );
```

Basically, this function takes a 32-bit integer and converts it to the proper byte order used by TCP/IP. This order is also referred to as *big-endian*. For your purposes, just pass `INADDR_ANY` to this function.

The last member you need initialize is `sin_port`. You can do this with a call to the `htons()` function. Here is the prototype for it.

```
u_short htons ( u_short hostshort )
```

This function is just like `htonl()` except it converts shorts to TCP/IP byte order. All you have to pass to this function is the port to which the server wants to bind. Figure 5.5 illustrates the TCP/IP dot order.

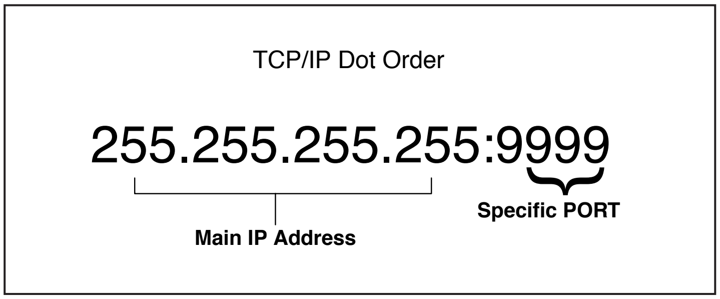


Figure 5.5

The TCP/IP dot order layout

I don't know about you, but I feel like playing a little *Rogue Spear* right about now. Nothing brings a little stress relief better than a friendly game of *Rogue Spear*. Of course, my compadres and I affectionately refer to the game as "Shot in the Head." We call it this because we really stink at the game and spend most of our time reloading the missions. Or, I could stay up into the a.m. and play some *Starsiege Tribes*. Now there's an addictive game. Well, I'd better keep writing; deadlines to meet.

Binding, Finally

Now that all of the initialization is out of the way, you can finally bind the socket to a port. You do this with the `bind()` function I discussed in the section, "The Bind Functionality." All you pass to it is your socket handle, the address of your socket address information, and the byte size of the address information. Here is the call that is made.

```
if( bind( skSocket, (sockaddr*) &saServerAddress,
 sizeof(sockaddr) ) == SOCKET_ERROR ) {
    cout << "**ERROR** Could Not Bind" << endl;
    // Clean up WinSock
    WSACleanup();
    exit(1);
}
```

Because the `bind()` function returns a value, just check to see whether it equals `SOCKET_ERROR`. If it does, clean up Winsock with the `WSACleanup()` function and then exit the program.

Cleaning Up Sockets

The last thing to do in your program is clean up your socket with the `closesocket()` and `WSACleanup()` functions. The `closesocket()` function closes an existing socket and returns the resource to the system. The `closesocket()` function takes only one parameter, the socket handle to close. Call the function like this:

```
closesocket(skSocket);
```

Once everything is cleaned up, the program exits. That's all there is to binding a socket with C!

The C++ Version of Binding

As promised, I'll now show you how to create the `bind` functionality for the C++ `SocketObject` class. Do you remember the start of the class header in the section, "The SOCKET Data Type?" Don't worry; I'll show it to you now, but with the addition of the constructor, the destructor, and the bind function prototype.

```
class SocketObject
{
    private:
    public:
        SOCKET     skSocket;
        SocketObject();      // Constructor
        ~SocketObject();     // Destructor
        int Bind(int iPort);
}
```

Because the only unique things you need to know to call the bind() function are a socket handle and a port number, you have just one parameter for the function. Here is the prototype.

```
int Bind( int iPort );
```

Nothing too complicated here, just the port number to which you want to bind the socket. You get the socket handle from the skSocket variable already defined in the class. Now that I've covered the header information for bind, take a look at the actual member function.

```
int SocketObject::Bind( int iPort )
{
    sockaddr_in saServerAddress;

    skSocket = socket(AF_INET, SOCK_STREAM, 0);
    if(skSocket == INVALID_SOCKET) {
        return false;
    }
    memset(&saServerAddress, 0, sizeof(sockaddr_in));
    saServerAddress.sin_family = AF_INET;
    saServerAddress.sin_addr.s_addr = htonl(INADDR_ANY);
    saServerAddress.sin_port = htons(iPort);
    if( bind(skSocket, (sockaddr*) &saServerAddress,
      sizeof(sockaddr)) == SOCKET_ERROR) {
        Disconnect();
        return false;
    }
    else
        return true;
}
```

All of this should look strikingly familiar to you by now. All of the functionality of the C code is in the class function here. One notable difference, though, is the exclusion of the Winsock initialization and cleanup routines. These are actually called from functions in the constructor and destructor members. Here are those member functions.

```
// Constructor
SocketObject::SocketObject()
{
    WSADATA wsaData;
    WORD     wVersionRequested;

    wVersionRequested = MAKEWORD( 2, 0 );
    skSocket = INVALID_SOCKET;
    iStatus = WSAStartup(wVersionRequested,&wsaData);
}
// Destructor
SocketObject::~SocketObject()
{
    Disconnect();
}
```

Because you call the constructor when you create a `SocketObject` class object, you just put the initialization routines into the constructor. It includes all of the functionality done in the C program for initializing Winsock.

The destructor is called when the class object is released or leaves program scope. All the destructor does is call another function called `Disconnect()`. Here is what the `Disconnect()` member function looks like.

```
void SocketObject::Disconnect()
{
    if(skSocket != INVALID_SOCKET) {
        closesocket(skSocket);
        skSocket = INVALID_SOCKET;
    }
}
```

There is not much different here from the C program except that now you check to make sure the socket handle is valid before attempting to close it. Most C++ classes include all sorts of error handling to help out the programmer. You perform the check by making sure the socket handle does not equal `INVALID_SOCKET`. One thing you might notice missing is the `WSACleanup()` function. I intentionally didn't include this function because you do not want to unload the ws2_32.dll every time you delete a `SocketObject`. The class leaves this up to the programmer to clean up.

Because I threw in the `Disconnect()` member function without showing it in the class header, here is the class header you have created so far.

```
class SocketObject
{
    private:
    public:
        SOCKET     skSocket;
        SocketObject();       // Constructor
        ~SocketObject();      // Destructor
        int Bind(int iPort);
        void Disconnect();
}
```

Now that the class infrastructure is out of the way, it is time to see the class in action! Here is the code to use the class functions so far.

```
// Standard includes
#include <iostream.h>
#include "..\\socketobject\\SocketObject.h"

// ----> Main Program Function (REQUIRED)
void main( void )
{
    SocketObject      ServerSocketObject;
    int               iPort = 6000;

    // Attempt to bind
    if ( ServerSocketObject.Bind( iPort ) ) {
        cout << "<-- SOCKET BOUND -->" << endl;
        ServerSocketObject.Disconnect();
    }
    else {
        cout << "**ERROR** Could Not Bind" << endl;
    }
    WSACleanup();
}
```

Now, I don't know what you think, but the preceding code looks much simpler

```
"G:\Source\Book\Chapter5\Bind_SocketObject\Release\Bind_SocketObject.exe"
<-- SOCKET BOUND -->
Press any key to continue_
```

Figure 5.6

The output from the Bind program

than the C version. Granted, you had to create some infrastructure, but once that infrastructure is in place, the actual implementation goes smoothly.

All of the initialization code takes place as soon as you declare the `ServerSocketObject` object. You handle all of the cleanup code by calling the `Disconnect()` member function and then the `WSACleanup()` function. Figure 5.6 shows the output from the Bind program.

The Listen Functionality

With `bind` out of the way, it is time to cover the `listen` functionality. Don't forget that you only call the `listen` function when you are using a server socket to listen for new client connections.

The only thing the `listen()` function does is wait for connections. Luckily for you, this function is simple to implement now that the basics are out of the way. Here is the updated C code that includes the new `listen` functionality.

```
// Standard includes
#include <iostream.h>
#include <winsock.h>
#include <stdio.h>

// ----> Main Program Function (REQUIRED)
void main(void)
{
```

```
SOCKET              skSocket;
sockaddr_in         saServerAddress;
int                 iPort = 6000;
int                 iStatus;
WSADATA             wsaData;
WORD                wVersionRequested;

// Tell WinSock we want version 2
wVersionRequested = MAKEWORD( 2, 0 );
// Initialize the socket handle
skSocket = INVALID_SOCKET;
// Start up WinSock
iStatus = WSAStartup( wVersionRequested, &wsaData );
// Create the socket
skSocket = socket( AF_INET, SOCK_STREAM, 0 );
// Check whether there was an error
if( skSocket == INVALID_SOCKET ) {
    cout << "**ERROR** Could Not Create Socket" << endl;
    // Clean up WinSock
    WSACleanup();
    exit(1);
}
cout << "<-- SOCKET CREATED -->" << endl;
// Clear out the socket address structure
memset(&saServerAddress, 0, sizeof(sockaddr_in));
// Initialize the socket address structure
saServerAddress.sin_family = AF_INET;
saServerAddress.sin_addr.s_addr = htonl( INADDR_ANY );
saServerAddress.sin_port = htons( iPort );
// Attempt to bind
if( bind( skSocket, (sockaddr*) &saServerAddress,
  sizeof(sockaddr) ) == SOCKET_ERROR ) {
    cout << "**ERROR** Could Not Bind" << endl;
    // Clean up WinSock
    WSACleanup();
    exit(1);
}
cout << "<-- SOCKET BOUND -->" << endl;
// Listen for a connection on the bound port
iStatus = listen( skSocket, 32 );
```

```
    if( iStatus == SOCKET_ERROR ) {
        cout << "**ERROR** Error Listening" << endl;
        // Close the socket
        closesocket(skSocket);
        // Clean up WinSock
        WSACleanup();
        exit(1);
    }
    cout << "<-- SOCKET LISTENING -->" << endl;
    // Close the socket
    closesocket(skSocket);
    // Clean up WinSock
    WSACleanup();
}
```

I put the relevant code changes in bold text to make them easier to pick out.

The Listen() Function

Because the first notable thing I added is a call to the `listen()` function, I'd better show you the prototype for it.

```
int listen ( SOCKET s, int backlog );
```

The first parameter just wants the handle to the socket with which you want to listen. For this, you should pass in the tried and true `skSocket` variable. The second parameter, `int backlog`, really threw me for a loop the first time I ever saw it. I mean, what the heck is a backlog? I have never claimed to give the most technical explanations for things, so the backlog is just how many pending connections you will allow to queue up. Normally, I just use the number 32 here.

The `listen()` function does return an integer, so you need to check what you get back from it. If everything goes okay, the function returns a `0`. If something goes wrong, then the function returns a `SOCKET_ERROR` code. In the example, I close the socket, clean up Winsock, and exit the program if something goes wrong.

Because the `listen()` function is non-blocking, the program calls it and then immediately continues on to clean up things and shut down. Refer to Figure 5.7 to see the `listen()` function illustrated.

The C++ Version of Listening

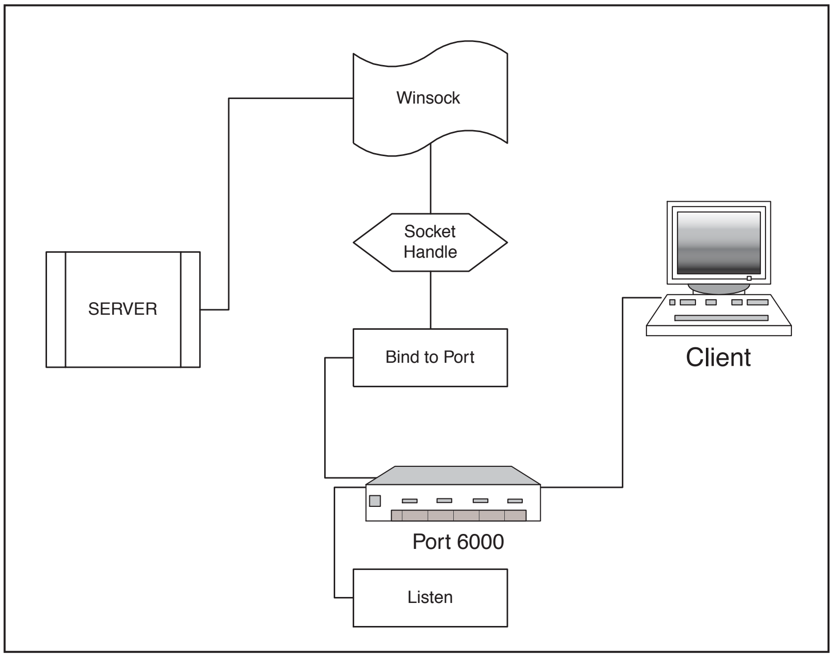


Figure 5.7
The `listen()` *function*

Because the C version of listening was so short and simple, the C++ version is going to be even shorter! To begin, you must add the `listen()` function to the class header for the `SocketObject` class. The class already has the socket handle present, so all you need to pass to the function is the port to which you want to assign the next connection. Here is the updated class header.

```
class SocketObject
{
    private:
    public:
        SOCKET     skSocket;
        SocketObject();       // Constructor
        ~SocketObject();      // Destructor
        int Bind(int iPort);
        void Disconnect();
        int Listen( void );
}
```

Once again, the relevant change appears in bold-faced text. All you had to add was the `Listen()` function. Now that the header is out of the way, peek at the member function implementation.

```
int SocketObject::Listen( void )
{
    return listen( skSocket, 32 );
}
```

Wow, isn't that a ton of code! Not. This one is pretty easy because all you have to do is call the `listen()` function provided by Winsock. You pass it the socket handle and the number of queued connections to allow.

The program returns the results from the `listen()` function, which is `0` for success or `SOCKET_ERROR` for failure. The infrastructure for listening is in place; take a look and see how it is implemented in C++ style.

```
// Standard includes
#include <iostream.h>
#include "..\\socketobject\\SocketObject.h"

// ----> Main Program Function (REQUIRED)
void main( void )
{
    SocketObject      ServerSocketObject;
    int               iPort = 6000;

    // Attempt to bind
    if ( ServerSocketObject.Bind( iPort ) ) {
        cout << "<-- SOCKET BOUND -->" << endl;
        if( ServerSocketObject.Listen( ) != SOCKET_ERROR ) {
            cout << "<-- SOCKET LISTENING -->" << endl;
        }
        else {
            cout << "**ERROR** Could Not Listen" << endl;
        }
        ServerSocketObject.Disconnect();
    }
    else {
        cout << "**ERROR** Could Not Bind" << endl;
    }
    WSACleanup();
}
```

Take a look at the code in bold text; it shows the differences between this program and the bind C++ program.

You can then check the call to the `listen()` function and see whether it returns the `SOCKET_ERROR` code. If so, display an error message and proceed to disconnect.

TIP

When you set up your server to listen for connections, use a listen port number that is greater than 1000. You avoid bumping into reserved port numbers such as port 80, which is reserved for HTTP traffic.

TIP

The backlog number limits itself to the service provider used if you pass a number that is too high for it. There is no provision for finding the limitation, but don't worry because the system changes it to the closest valid number if you exceed the maximum setting.

The Accept Functionality

Man, this information can be dryer than Rocky Mountain air. Tell you what; here is a tidbit of knowledge to spice things up. Once upon a time, there was a program called WinNuke. This program was very simple in appearance because all it displayed was a small window asking for an IP address. After you entered an IP address, you could then press the Nuke button, which sent a packet to the target machine. The packet caused a network stack overflow on the target machine and crashed it. The only thing the victim saw on his computer was the big nasty blue screen. Of course, I in no way condone using such a nasty piece of programming, but it *might* have been a great way to get rid of rude gamers. The reason I refer to this program in the past tense is because after Windows 95, Microsoft fixed this bug. If you still own Windows 95, you can download a patch to prevent this program from successfully attacking your machine. Now, back to our regularly scheduled program.

As you learned in the last chapter, `accept` is used by servers to accept new client connections. To get to the point of accepting connections, the server must first complete the following steps.

1. Initialize Winsock.
2. Create a socket.
3. Bind the socket to a port.
4. Listen on the bound port to accept the client connection.

NOTE

The `accept()` **function applies only to sockets of type** `SOCK_STREAM`**. Datagram sockets do not use this function.**

The Accept() Function

Once all the criteria are met for accepting a connection, you can call the `accept()` function. The most important part of this function is that it returns a socket handle to the new client socket connection. Here is the prototype.

```
SOCKET accept ( SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen );
```

For the first parameter, you pass the handle to the server socket. This tells the `accept` function on which server socket to accept. The socket passed to this parameter must be in the *listening state* by being called with the `listen()` function.

The second parameter expects the address of a `SOCKADDR_IN` data structure to hold all of the connection information. From this data structure, you will be able to pull the connecting client's IP address. The third and final parameter is a pointer to an integer containing the size of the `SOCKADDR_IN` data structure.

When the `accept()` function returns, it returns a socket handle. If the call was not successful, a value of `INVALID_SOCKET` is returned.

CAUTION

The `bind()` **function is a blocking call. The program sits and waits until a client connects. Keep this in mind when calling the** `bind()` **function.**

Okay, now that you understand the `accept` function, here it is in plain old C.

```
// Standard includes
#include <iostream.h>
#include <winsock.h>
#include <stdio.h>

// ----> Main Program Function (REQUIRED)
void main(void)
{
    SOCKET                skSocket;
```

```
sockaddr_in          saServerAddress;
SOCKET               skClientSocket;
sockaddr_in          saClientAddress;
int                  iClientSize = sizeof(sockaddr_in);
int                  iPort = 6000;
int                  iStatus;
WSADATA              wsaData;
WORD                 wVersionRequested;

// Tell WinSock we want version 2
wVersionRequested = MAKEWORD( 2, 0 );
// Initialize the socket handle
skSocket = INVALID_SOCKET;
// Start up WinSock
iStatus = WSAStartup( wVersionRequested, &wsaData );
// Create the socket
skSocket = socket( AF_INET, SOCK_STREAM, 0 );
// Check whether there was an error
if( skSocket == INVALID_SOCKET ) {
     cout << "**ERROR** Could Not Create Socket" << endl;
     // Clean up WinSock
     WSACleanup();
     exit(1);
}
cout << "<-- SOCKET CREATED -->" << endl;
// Clear out the socket address structure
memset(&saServerAddress, 0, sizeof(sockaddr_in));
// Initialize the socket address structure
saServerAddress.sin_family = AF_INET;
saServerAddress.sin_addr.s_addr = htonl( INADDR_ANY );
saServerAddress.sin_port = htons( iPort );
// Attempt to bind
if( bind( skSocket, (sockaddr*) &saServerAddress, sizeof(sockaddr) )
 == SOCKET_ERROR ) {
     cout << "**ERROR** Could Not Bind" << endl;
     // Clean up WinSock
     WSACleanup();
     exit(1);
}
cout << "<-- SOCKET BOUND -->" << endl;
// Listen for a connection
```

```
    iStatus = listen( skSocket );
    if( iStatus == SOCKET_ERROR ) {
        cout << "**ERROR** Error Listening" << endl;
        // Close the socket
        closesocket(skSocket);
        // Clean up WinSock
        WSACleanup();
        exit(1);
    }
    cout << "<-- SOCKET LISTENING -->" << endl;
    skClientSocket = accept( skSocket,
     (struct sockaddr*)&saClientAddress, &iClientSize );
    if( skClientSocket == INVALID_SOCKET ) {
        cout << "**ERROR** Error Accepting" << endl;
        // Close the socket
        closesocket(skSocket);
        // Clean up WinSock
        WSACleanup();
        exit(1);
    }
    cout << "<-- CLIENT ACCEPTED -->" << endl;
    // Close the sockets
    closesocket(skSocket);
    closesocket(skClientSocket);
    // Clean up WinSock
    WSACleanup();
}
```

Hey, would you look at that; the relevant changes are shown in bold text! Sometimes, I amaze myself. Anyway, take a look-see at the code changes in bold and see if you can follow them.

One of the more notable things I had to do was add a new socket handle for the client connection. Because the `accept()` function returns a socket handle, I had to create a holder for it. I also had to add a client address and an integer holding the client address size.

After the new variables are out of the way and the server socket is in listening mode, the program makes a call to the `accept()` function. The usual parameters are passed to this function, and the `skClientSocket` variable receives the results. To make sure the `accept()` function succeeded, you should compare the client socket handle to `INVALID_SOCKET`. If the client socket handle equals this value, you then close the server socket and clean up Winsock before exiting the program.

The last thing I had to add was a call to close the client socket. Because the new client socket exists, you must make this call to free the resource. You don't want to hold it hostage now, do you? Take a look at Figure 5.8 to see the `accept()` function illustrated.

That's all for the C version of the server accepting a client connection. See, no hidden wrappers or classes, just straight C code. Now, for all of the object-oriented

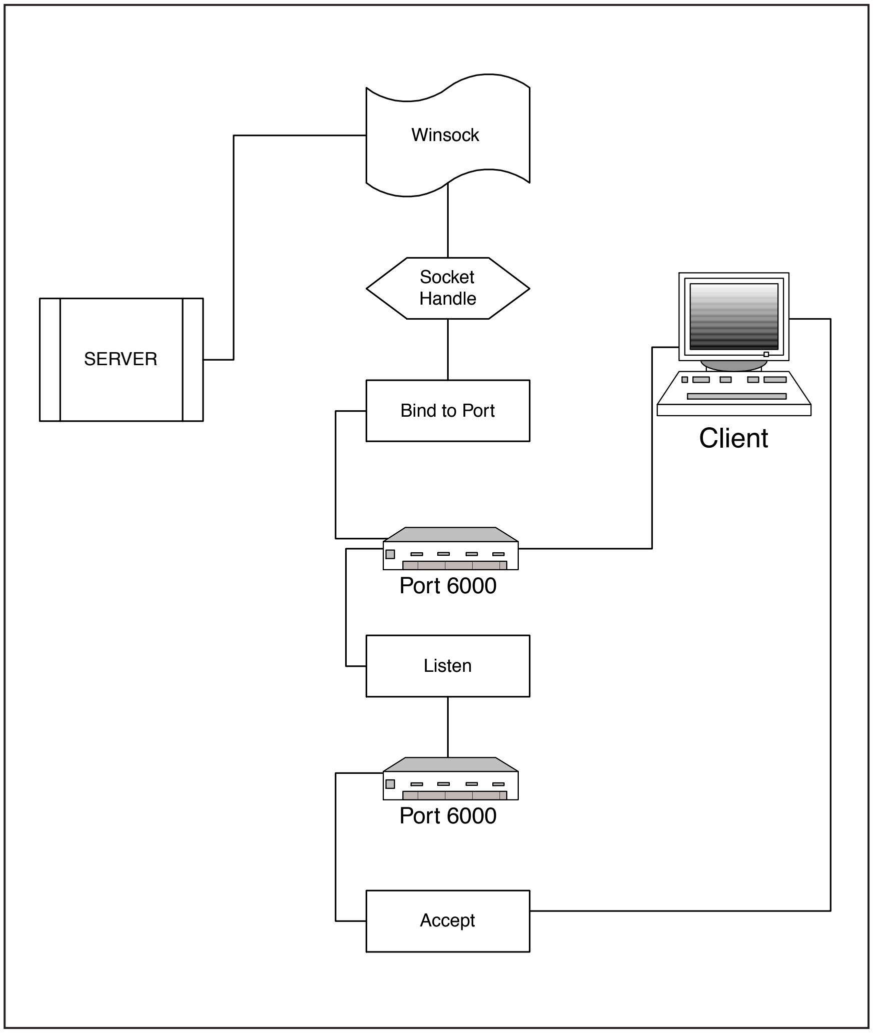


Figure 5.8

The `accept()` *function*

folks out there, take a look at the C++ version of the code.

The C++ Version of Accepting a Client Connection

Because examining the past is the best way to predict the future, use your past practices as a guide. First of all, you must add a new function prototype to the class header. Here is the class header with the new `accept` functionality.

```
class SocketObject
{
    private:
    public:
        SOCKET     skSocket;
        SocketObject();      // Constructor
        ~SocketObject();     // Destructor
        int Bind(int iPort);
        void Disconnect();
        int Listen( void );
        bool Accept(SocketObject& skAcceptSocket);
}
```

The only thing you had to add to the class was a single `Accept()` function. The `Accept()` function expects one parameter of type `SocketObject` address. Basically, all you have to do is pass the address of the client's `SocketObject` to the server's `Accept()` function. To better illustrate, take a look at the latest our C++ client has to offer.

```
// Standard includes
#include <iostream.h>
#include "..\\socketobject\\SocketObject.h"

// ----> Main Program Function (REQUIRED)
void main( void )
{
    SocketObject     ServerSocketObject;
    SocketObject     ClientSocketObject;
    int              iPort = 6000;

    // Attempt to bind
    if ( ServerSocketObject.Bind( iPort ) ) {
        cout << "<-- SOCKET BOUND -->" << endl;
        if( ServerSocketObject.Listen( ) != SOCKET_ERROR ) {
            cout << "<-- SOCKET LISTENING -->" << endl;
            if( ServerSocketObject.Accept( ClientSocketObject ) ) {
                cout << "<-- CLIENT ACCEPTED -->" << endl;
            }
            else {
                cout << "**ERROR** Could Not Accept" << endl;
            }
        }
        else {
            cout << "**ERROR** Could Not Listen" << endl;
        }
        ServerSocketObject.Disconnect();
    }
    else {
        cout << "**ERROR** Could Not Bind" << endl;
    }
    WSACleanup();
}
```

Luckily for you, there isn't a whole lot of difference between this client and the last version you worked on. First, you add a client SocketObject to hold the new socket connection. Next, you call the Accept() member function and pass the address of the client socket object. If the Accept() member function is successful, a 1 is returned. If a problem occurs, a 0 is returned. So all you have to do is check whether the function returned a 1, and if it didn't, output an error message.

Now that you have seen the prototype and the implementation, take a look at the Accept() member function.

```
bool SocketObject::Accept( SocketObject &skAcceptSocket )
{
    sockaddr_in      saClientAddress;
    int              iClientSize = sizeof(sockaddr_in);

    skAcceptSocket.skSocket = accept( skSocket,
     (struct sockaddr*)&saClientAddress, &iClientSize );
    if( skAcceptSocket.skSocket == INVALID_SOCKET ) {
        return false;
    }
    else {
        return true;
    }
}
```

As I briefly mentioned before, the `Accept()` member function expects the address of the `SocketObject` to accept the new connection. If everything goes okay, the function returns a `1`. If something fails, the function returns a `0`.

The first bits of code create the necessary local variables to call the `accept()` socket function. After the required variables are out of the way, you get down to business with the `accept()` function. When calling the `accept()` function, you pass in the local `SocketObject`'s socket handle for the server socket parameter. You then hold the return value in the `skAcceptSocket` socket handle. Next, you check the value of the accepting socket handle to see whether it equals `INVALID_SOCKET`. If it does, then the call failed and you return a `0`. If everything went okay, you return a `1`, as expected.

Because all you are doing is wrapping functionality, most of this code should look very similar to the C counterpart. That's the beauty of C++! One problem you will have when running the accept programs is that they just sit there waiting for a client connection. And guess what: You don't have a client program example in this chapter yet! I'll help you fix this problem by teaching you how to use the client-side `connect` functionality.

The Connection Functionality

Now, you finally get to play with clients. One thing to remember is that the connection functionality is used strictly by clients to connect to servers. Before the client application can make a connection to a server, it must complete the following steps.

1. Initialize Winsock.
2. Create a socket.

Consider this small number of steps my holiday gift to you. No, I don't know which holiday; just pick one. Or make one up, for that matter. Do you recall the *Seinfeld* episode where George's father created his own holiday? During the holiday he created, named Festivus, the family would get together for a meal and then proceed to air all the grievances they had against one another. I think most of us call this *Thanksgiving*.

Anyway, all jokes aside, you should already be familiar with the two prerequisite

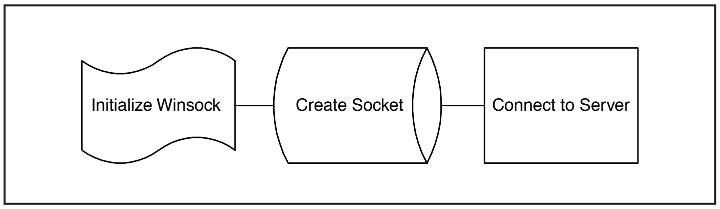


Figure 5.9
Client steps to connect to a server

steps I mentioned. To establish connections with servers, you use the `connect()` function provided in Winsock. Refer to Figure 5.9 to see the `connect()` function illustrated.

The Connect() Function

The `connect()` function, provided by Winsock and most other socket APIs, is fairly straightforward. Here is the prototype for the function.

```
int connect ( SOCKET s, const struct sockaddr FAR*  name, int namelen );
```

The function returns an integer when it is done. The return value is `0` if everything went all right. If something fails, then a value of `SOCKET_ERROR` is returned.

The first parameter passed to the `connect()` function is a socket handle. This `SOCKET` variable ends up holding the handle to the newly connected socket. The second parameter expects a `SOCKADDR` structure address holding the name of the server to which you want to connect. The third and last parameter is nothing more than the size of the server address. This one is fairly simple because all you have to pass to it is `sizeof(sockaddr)`.

The `connect()` function does just what its name implies; it establishes a connection to a server. Once a connection is established, you can send and receive data from the server. Here is the C version of the `connect` code.

```
// Standard includes
#include <iostream.h>
#include <winsock.h>
#include <stdio.h>

// ----> Main Program Function (REQUIRED)
void main(void)
{
    SOCKET            skSocket;
```

```
sockaddr_in         saServerAddress;
int                 iPort = 6000,iStatus;
WSADATA             wsaData;
WORD                wVersionRequested;
LPHOSTENT           lpHost;
char                szHost[128];

// Init the host value; change this IP to whatever valid IP you wish
sprintf(szHost,"192.168.0.2");
// Tell WinSock we want version 2
wVersionRequested = MAKEWORD( 2, 0 );
// Initialize the socket handle
skSocket = INVALID_SOCKET;
// Start up WinSock
iStatus = WSAStartup( wVersionRequested, &wsaData );
// Create the socket
skSocket = socket( AF_INET, SOCK_STREAM, 0 );
// Check whether there was an error
if( skSocket == INVALID_SOCKET ) {
     cout << "**ERROR** Could Not Create Socket" << endl;
     // Clean up WinSock
     WSACleanup();
     exit(1);
}
cout << "<-- SOCKET CREATED -->" << endl;
// Initialize the server address data structure
memset(&saServerAddress,0,sizeof(sockaddr_in));
// Set this by default
saServerAddress.sin_family = AF_INET;
// Load the IP address
saServerAddress.sin_addr.s_addr = inet_addr(szHost);
// If the host specified is not an IP address,
// we must look up the value
if( saServerAddress.sin_addr.s_addr == INADDR_NONE )
{
     cout << "<-- LOOKING UP HOST IP -->" << endl;
     // Get the hostname
     lpHost = gethostbyname(szHost);
     // Check whether we got something back
     if (lpHost != NULL) {
        // Load the server address with the host information
        saServerAddress.sin_addr.s_addr =
         ((LPIN_ADDR)lpHost->h_addr)->s_addr;
        }
        else {
            cout << "**ERROR** Could Not Locate Host" << endl;
            // Clean up WinSock
            WSACleanup();
            exit(1);
        }
    }
    // Set the server port
    saServerAddress.sin_port = htons(iPort);
    // Attempt to connect to the server
    iStatus = connect(skSocket,
     (struct sockaddr*)&saServerAddress,sizeof(sockaddr));
    // Check whether there was an error
    if( iStatus == SOCKET_ERROR ) {
        cout << "**ERROR** Could Not Connect To Server" << endl;
        // Clean up WinSock
        WSACleanup();
        exit(1);
    }
    cout << "<-- CONNECTED TO SERVER -->" << endl;
    // Close the socket
    closesocket(skSocket);
    // Clean up WinSock
    WSACleanup();
}
```

I added a couple of interesting variables at the beginning of the program. The `lpHost` variable is type `HOSTENT` and holds the host's official name. The `szHost` variable is a *character buffer* you use to hold the name of the server to which you want to connect.

Oops, I don't want to leave you guessing what a `HOSTENT` structure is, so here you go.

```
struct hostent {
    char FAR            *h_name;
    char FAR *FAR       *h_aliases;
    short               h_addrtype;
    short               h_length;
    char FAR *FAR       *h_addr_list;
};
```

You only use this data structure when you have to look up a host IP address. For instance, if you want to connect to lostlogic.com, you have to look up the IP address before connecting.

In the beginning of the program, you complete the usual steps you have followed so far, such as initializing Winsock and creating the socket. The first notable thing you do in this program is set up the server's location with the following call.

```
saServerAddress.sin_addr.s_addr = inet_addr(szHost);
```

The inet_addr() Function

With the last piece of code discussed in the previous section, you call the `inet_addr()` function. You pass the hostname that you have loaded in `szHost`. Here is the prototype for the `inet_addr()` function.

```
unsigned long inet_addr ( const char FAR *cp );
```

The first and only parameter to this function is a string containing the Internet address in dot format. For example, a dot-formatted IP address is 192.168.0.2. You can also pass the IP address in octal, hexadecimal, or a mixture. All of my examples use the decimal system.

If the call is successful, then an integer containing the binary equivalent of the address is returned. If the call fails, then a value of `INADDR_NONE` is returned.

This brings you to the next piece of code that checks the return value. If the address equals `INADDR_NONE`, then the program goes on to do a host lookup. This situation happens if you pass an Internet name such as yahoo.com or lostlogic.com to the `inet_addr()` function. Here is the code you use to do a host lookup.

```
lpHost = gethostbyname(szHost);
```

The gethostbyname() Function

Notice that you perform a host lookup using the `gethostbyname()` function. You pass the function the same Internet address that you passed to the `inet_addr()` function. Here is the prototype.

```
struct hostent FAR *gethostbyname ( const char FAR *name )
```

As I just mentioned, the only parameter you pass to this is an Internet address,

CAUTION

Never try to modify the member variables in the `HOSTENT` data structure. Windows handles all of the memory allocation for this data structure. Also, each thread is limited to one of the structures at a time. The best practice is to copy the contents of the structure to local variables before using the data.

such as lostlogic.com. In this example, you need to pass the `szHost` string to it. On a successful call to this function, a structure of type `HOSTENT` is returned. If Winsock is unable to locate the host, then a `NULL` is returned.

If the `gethostbyname()` function returns a non-`NULL` value, you go on to set your `saServerAddress` variable with the located Internet address.

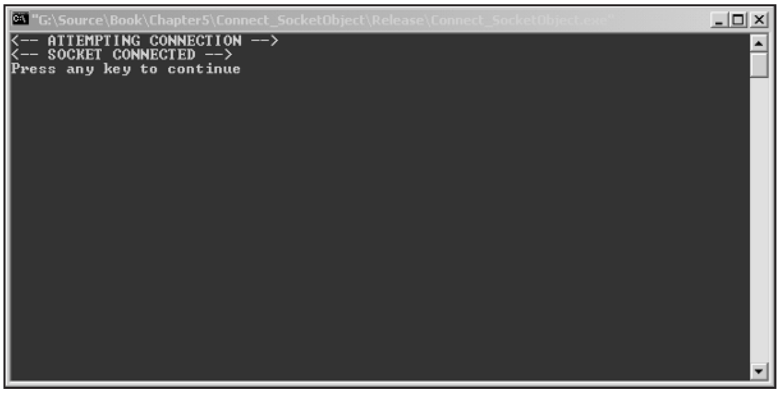
```
if (lpHost != NULL) {
    // Load the server address with the host information
    saServerAddress.sin_addr.s_addr =((LPIN_ADDR)lpHost->h_addr)->s_addr;
}
```

Now that the host lookup is out of the way, you can go on to set the server port using the familiar `htons()` function. After the port is set, you finally get to call the `connect()` function.

```
saServerAddress.sin_port = htons(iPort);
// Attempt to connect to the server
iStatus = connect(skSocket,
 (struct sockaddr*)&saServerAddress,sizeof(sockaddr));
```

You pass to the `connect()` function your socket, server address, and the size of the server address. The `iStatus` variable holds the return value. You might remember that if the call to connect fails, a value of `SOCKET_ERROR` is returned. To check for this, all you do is compare `iStatus` to `SOCKET_ERROR` and see whether they are equal. If the call did indeed fail, then close down sockets and exit.

Darn those games of *Age of Empires II*. A friend of mine called, and we decided to play a "quick" game of *AOEII*. Well, guess what: There are no quick games of *AOEII*. Anyway, at least I got to blow off some steam with my elite janissaries. We normally play against the computer, and this time we beat three computer opponents on the



```
"G:\Source\Book\Chapter5\Connect_SocketObject\Release\Connect_SocketObject.exe"
<-- ATTEMPTING CONNECTION -->
<-- SOCKET CONNECTED -->
Press any key to continue
```

Figure 5.10

Output from the connection program

hardest level. Maybe we're good enough to play on The Zone? Naaaahhh. For some reason, I always seem to run into people who advance to the castle age within five minutes. Something seems fishy about that. Here's an idea: Why don't I talk about how to prevent cheating in a later chapter? I think it's a good idea, so count on seeing it. Okay, back to business. Before I move on, take a look at the output from the connection program in Figure 5.10.

Now that the C version is out of the way, it's time for the good old C++ version.

The C++ Version of Establishing a Connection

History is going to repeat itself here because you first add the `connect()` function prototype to your class header. Here is what the latest header needs to look like.

```
class SocketObject
{
    private:
    public:
        SOCKET     skSocket;
        SocketObject();      // Constructor
        ~SocketObject();     // Destructor
        int Bind(int iPort);
        void Disconnect();
        int Listen( void );
        bool Connect(char* szServerAddress, int iPort);
}
```

The only part you have to add is in bold text. The new `Connect()` function accepts two parameters. The first parameter expects the server's address to which you are trying to connect. The address can be either dot format or a hostname.

The second parameter expects the port number on the server to which you want to connect. The port number uses the usual range of numbers between 1 and 10000 that you have been using so far. If the connection is a success, then a `True` or `1` is returned. If something goes wrong, then a `0` is returned.

Now that the prototype is out of the way, take a look at the actual member function. Without further ado, here it is.

```
bool SocketObject::Connect(char* szServerAddress, int iPort)
{
    struct sockaddr_in      serv_addr;
    LPHOSTENT               lpHost;
    int                     err;

    // Open the socket
    skSocket = socket(AF_INET, SOCK_STREAM, 0);
    if(skSocket == INVALID_SOCKET) {
        return false;
    }
    memset(&serv_addr,0,sizeof(sockaddr_in));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(szServerAddress);
    if (serv_addr.sin_addr.s_addr == INADDR_NONE) {
        lpHost = gethostbyname(szServerAddress);
        if (lpHost != NULL) {
            serv_addr.sin_addr.s_addr =
             ((LPIN_ADDR)lpHost->h_addr)->s_addr;
        }
        else {
            return FALSE;
        }
    }
    // Assign the port
    serv_addr.sin_port = htons(iPort);
    // Establish the connection
    err = connect(skSocket,
     (struct sockaddr*)&serv_addr,sizeof(sockaddr));
    if(err == SOCKET_ERROR) {
        Disconnect();
        return false;
    }
    return true;
}
```

There really isn't much here that you didn't already do in the C version of the code. But, go over the steps that take place as a refresher.

1. Create the socket using the `socket()` function.
2. Initialize the server address structure using the `inet_addr()`, `gethostbyname()`, and `htons()` functions.
3. Establish the connection using the `connect()` function.

Now, I want you to take a look back at the C version of the client application quickly. Why do I want you to do this? I think you will be pleasantly surprised at how small the C++ version is! So, here it is, the C++ version of the client application.

```
// Standard includes
#include <iostream.h>
#include "..\\socketobject\\SocketObject.h"

// ----> Main Program Function (REQUIRED)
void main( void )
{
    SocketObject     ClientSocketObject;
    int              iPort = 80;     // HTTP port

    cout << "<-- ATTEMPTING CONNECTION -->" << endl;
    // Attempt to connect
    if ( ClientSocketObject.Connect( "www.lostlogic.com", iPort ) ) {
        cout << "<-- SOCKET CONNECTED -->" << endl;
    }
    else {
        cout << "**ERROR** Could Not Connect" << endl;
    }
    WSACleanup();
}
```

I hope the first words out of your mouth are something like “Wow, look how short it is!” Yeah, yeah, I know; you already saw this in Chapter 4, “Writing Your First Network Programs.” But hey, now maybe you understand it better. The first thing I do is create a `SocketObject` object and a variable to hold the port number to which I want to connect. I then call the `Connect()` function and pass to it the Internet address and port with which I want to connect. All I do then is check whether the call was successful; if it wasn’t, I throw an error message.

Maybe you noticed that I changed the usual port from 6000 to 80. I did this to demonstrate how browsers connect to Web sites. They just establish a connection with the standard HTTP port number, which is port number 80. So to establish a connection with lostlogic.com, all you have to do is set lostlogic.com as the hostname and port 80 as the port number. Voilà! You now have the beginnings of an Internet browser program.

Sending Data

The previous programs have given you a good foundation to work with, but they are lacking in some areas. Like, how about sending data? You might be thinking that this is going to be the hard part, but actually, the `send()` command is one of the easiest to master. Uh oh, I let the cat out of the bag. So now that you know, you are going to be using the `send()` function.

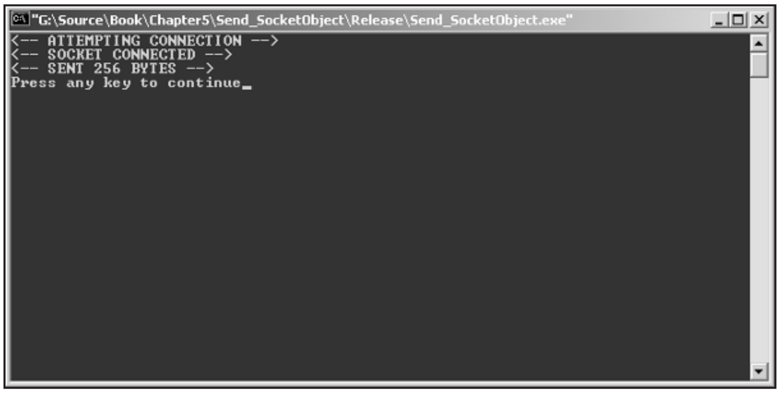
When you send data using sockets, you send a stream of bytes across the communication channel. You don’t really have complete control over when and how the data is transmitted; you just tell Winsock to send it. Even though there are no real limits to the amount of data you can send, you should keep it to a minimum. If you need to send a large stream of data, such as a file, break it up into multiple sends.

The C Version of Sending Data

You might remember the C++ version of sending data from when you wrote the SendDataTest program in Chapter 4, “Writing Your First Network Programs.” You are going to cover the same type of thing here, and as a bonus; you are going to learn how to do it in C. Here is the code for a client application that wants to send data to a server.

```
// Standard includes
#include <iostream.h>
```

```
#include <winsock.h>
#include <stdio.h>

// ----> Main Program Function (REQUIRED)
void main(void)
{
    SOCKET              skSocket;
    sockaddr_in         saServerAddress;
    int                 iPort = 80,iStatus;
    WSADATA             wsaData;
    WORD                wVersionRequested;
    LPHOSTENT           lpHost;
    char                szHost[128];
    char                szSendBuffer[256];
    int                 iBytesSent;

    // Init the host value; change this IP to whatever valid IP you wish
    sprintf(szHost,"www.lostlogic.com");
    // Tell WinSock we want version 2
    wVersionRequested = MAKEWORD( 2, 0 );
    // Initialize the socket handle
    skSocket = INVALID_SOCKET;
    // Start up WinSock
    iStatus = WSAStartup( wVersionRequested, &wsaData );
    // Create the socket
    skSocket = socket( AF_INET, SOCK_STREAM, 0 );
    // Check whether there was an error
    if( skSocket == INVALID_SOCKET ) {
        cout << "**ERROR** Could Not Create Socket" << endl;
        // Clean up WinSock
        WSACleanup();
        exit(1);
    }
    cout << "<-- SOCKET CREATED -->" << endl;
    // Initialize the server address data structure
    memset(&saServerAddress,0,sizeof(sockaddr_in));
    // Set this by default
    saServerAddress.sin_family = AF_INET;
    // Load the IP address
    saServerAddress.sin_addr.s_addr = inet_addr(szHost);
    // If the host specified is not an IP address,
    // we must look up the value
    if( saServerAddress.sin_addr.s_addr == INADDR_NONE ) {
        cout << "<-- LOOKING UP HOST IP -->" << endl;
        // Get the hostname
        lpHost = gethostbyname(szHost);
        // Check whether we got something back
        if (lpHost != NULL) {
            // Load the server address with the host information
               saServerAddress.sin_addr.s_addr =
                ((LPIN_ADDR)lpHost->h_addr)->s_addr;
          }
          else {
               cout << "**ERROR** Could Not Locate Host" << endl;
               // Clean up WinSock
               WSACleanup();
               exit(1);
          }
     }
     // Set the server port
     saServerAddress.sin_port = htons(iPort);
     // Attempt to connect to the server
     iStatus = connect(skSocket,
      (struct sockaddr*)&saServerAddress,sizeof(sockaddr));
     // Check whether there was an error
     if( iStatus == SOCKET_ERROR ) {
          cout << "**ERROR** Could Not Connect To Server" << endl;
          // Clean up WinSock
          WSACleanup();
          exit(1);
     }
     cout << "<-- CONNECTED TO SERVER -->" << endl;
     sprintf(szSendBuffer,"Test Data");
     iBytesSent = send(skSocket,szSendBuffer,256,0);
     cout << "<-- SENT " << iBytesSent << " BYTES -->" << endl;
     // Close the socket
     closesocket(skSocket);
     // Clean up WinSock
     WSACleanup();
}
```



```
"G:\Source\Book\Chapter5\Send_SocketObject\Release\Send_SocketObject.exe"
<-- ATTEMPTING CONNECTION -->
<-- SOCKET CONNECTED -->
<-- SENT 256 BYTES -->
Press any key to continue_
```

Figure 5.11

Output from the send data program

The more notable events are once again bolded for you. The beginning of the program has a couple of new variables to handle the data you are about to send. First, you create a character buffer to hold the data you want to send. Then, you create an integer to hold how many bytes you successfully sent. After all of that is out of the way, you can perform the usual steps to establish a connection—and then you get to the good part. Refer to Figure 5.11 to see the output from the send data program.

The Send() Function

The `send()` function is the heart and soul of sending data with sockets. Even though it provides a great function, it really is easy to use. Here is the prototype.

```
int send ( SOCKET s, const char FAR *buf, int len, int flags );
```

When you call the `send()` function, it returns the number of bytes successfully transmitted. If for some reason the transmission has errors, then the value of `SOCKET_ERROR` is returned.

The first parameter passed to the function is the socket handle through which you send the data. In the example, you pass the socket variable, `skSocket`, for this parameter. The second parameter expects a character buffer containing the data to transmit. As long as the data is in a character buffer, it really doesn't matter what you send. The `send()` function does ignore null terminators and line feeds.

The third parameter tells the `send()` function how much data you want to transmit. In the current example, you can just go ahead and transmit the full character buffer. You can also use the string length of the buffer for this parameter to save transmission bytes.

The last parameter of the `send()` function holds flags for altering the way the message is transmitted. Just pass a `0` or `NULL` to this parameter because it is not of much use to you. If you really want to pass something to it, you can pass it `MSG_DONTROUTE`. This flag specifies that the transmission should not get routed. Chances are that Winsock will ignore this, though.

In the client application, you call the `send` function like this:

```
sprintf(szSendBuffer,"Test Data");
iBytesSent = send(skSocket,szSendBuffer,256,0);
```

First you load the send buffer with `"Test Data"` and then you call the `send` function. You pass it the socket handle, the send buffer, the size of the send buffer, and 0 for flags. The `iBytesSent` variable catches how many bytes you sent, and that's all there is to it!

The C++ Version of Sending Data

Ahh, the sweet scent of C++. You know the drill; it's time to modify the class header. Here is the new function prototype you are going to add.

```
int Send(char *szBuffer, int iBufLen, int iFlags);
```

The function returns the number of bytes sent. The first parameter expects a character array containing the data to transmit. The second parameter expects the number of bytes to transmit. The last parameter allows you to pass flags determining how the data is transmitted. All of these parameters are identical to the C version except that you do not have to pass the socket handle. Here is what the function looks like in the header.

```
class SocketObject
{
    private:
    public:
        SOCKET      skSocket;
        SocketObject();     // Constructor
        ~SocketObject();     // Destructor
        int Bind(int iPort);
        void Disconnect();
        int Listen( void );
        bool Connect(char* szServerAddress, int iPort);
        int Send(char *szBuffer, int iBufLen, int iFlags);
}
```

All right, now that the header is out of the way, take a look at the member function.

```
int SocketObject::Send(char *szBuffer, int iBufLen, int iFlags)
{
    return send(skSocket,szBuffer,iBufLen,iFlags);
}
```

Wow, isn't that the most complicated thing you have ever seen? All you are actually doing is calling the Winsock send() function with the given parameters. Because you have a SOCKET handle in the class object, you pass it as the first parameter. The rest of the parameters are passed to the function. That's really about it as far as the member function is concerned. Take a look at the C++ client application code now.

```
// Standard includes
#include <iostream.h>
#include "..\\socketobject\\SocketObject.h"

// ----> Main Program Function (REQUIRED)
void main( void )
{
    SocketObject        ClientSocketObject;
    int                 iPort = 80;     // HTTP port
    char                szSendBuffer[256];
    int                 iBytesSent;

    cout << "<-- ATTEMPTING CONNECTION -->" << endl;
    // Attempt to connect
    if ( ClientSocketObject.Connect( "www.lostlogic.com", iPort ) ) {
        cout << "<-- SOCKET CONNECTED -->" << endl;
        sprintf(szSendBuffer,"TestData");
        iBytesSent = ClientSocketObject.Send(szSendBuffer,256,0);
        cout << "<-- SENT " << iBytesSent << " BYTES -->" << endl;
    }
    else {
        cout << "**ERROR** Could Not Connect" << endl;
    }
    WSACleanup();
}
```

This program is not much different from the `connect()` function example. Two new variables handle the data transmission. The `szSendBuffer` character array holds the data to send. The `iBytesSent` tells you how many bytes were successfully transmitted.

You might want to run the `Accept_SocketObject` program and then connect and send data to it with this program. Just change the connection target in the Send_SocketObject program from lostlogic.com to the IP address of the server.

Receiving Data

Now that you can send data, you need to receive some. Luckily, receiving is just as easy as sending. You accomplish receiving data using Winsock's `recv()` function. When you make a call to the `recv()` function, your program waits until data is obtained. If no data is received, then the program just sits and waits. Because this call is a blocking call, you must be careful when calling it. You don't want your programs to sit and wait forever, do you? When you develop your first socket-based game, I'll go into more detail on how to deal with blocking calls. For now, let's just see how it works.

The Recv() function

The `recv()` function bears a striking resemblance to the `send()` function. Here is the prototype.

```
int recv ( SOCKET s, char FAR *buf, int len, int flags );
```

The first parameter requests a handle to the socket connection from which you want to receive data. You must have established the connection with either the `accept()` function for servers or the `connect()` function for clients.

The second parameter passes the pointer to the buffer to receive the data. All data is received and sent using character buffers. Once you receive the data, you can

Table 5.4 Flags for the Recv() Function

Flag	Purpose
MSG_PEEK	Allows you to peek at data in the receive buffer. Data is copied to the receive character buffer, but it is not removed from the socket buffer. This flag is useful if you want to make sure data is ready to be received. It can keep you from blocking your program to wait for data.
MSG_OOB	Allows you to process out-of-band data. None of the code examples I write use this option.

then copy it into whatever data structure you want.

The third parameter tells the function how large the receive buffer is. You can pass a smaller number here than the actual size of your receive buffer, but do not pass a larger number.

The fourth and final parameter accepts flag settings to determine the method the call uses. Some of the flags can be seen in Table 5.4.

The C Version of Receiving Data

Okay, now that you have seen the prototype, take a look at it in action with C.

```
// Standard includes
#include <iostream.h>
#include <winsock.h>
#include <stdio.h>

// ----> Main Program Function (REQUIRED)
void main(void)
{
    SOCKET           skSocket;
    sockaddr_in      saServerAddress;
    int              iPort = 80,iStatus;
    WSADATA          wsaData;
```

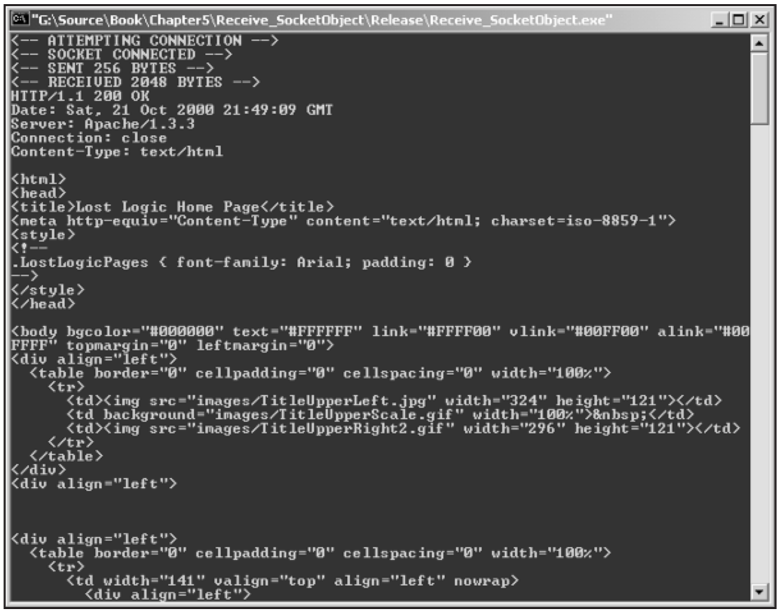
```
WORD                wVersionRequested;
LPHOSTENT           lpHost;
char                szHost[128];
char                szSendBuffer[256];
char                szRecvBuffer[32768];
int                 iBytesSent;
int                 iBytesReceived;

// Init the host value; change this IP to whatever valid IP you wish
sprintf(szHost,"www.lostlogic.com");
// Tell WinSock we want version 2
wVersionRequested = MAKEWORD( 2, 0 );
// Initialize the socket handle
skSocket = INVALID_SOCKET;
// Start up WinSock
iStatus = WSAStartup( wVersionRequested, &wsaData );
// Create the socket
skSocket = socket( AF_INET, SOCK_STREAM, 0 );
// Check whether there was an error
if( skSocket == INVALID_SOCKET ) {
    cout << "**ERROR** Could Not Create Socket" << endl;
    // Clean up WinSock
    WSACleanup();
    exit(1);
}
cout << "<-- SOCKET CREATED -->" << endl;
// Initialize the server address data structure
memset(&saServerAddress,0,sizeof(sockaddr_in));
// Set this by default
saServerAddress.sin_family = AF_INET;
// Load the IP address
saServerAddress.sin_addr.s_addr = inet_addr(szHost);
// If the host specified is not an IP address,
// we must look up the value
if( saServerAddress.sin_addr.s_addr == INADDR_NONE ) {
    cout << "<-- LOOKING UP HOST IP -->" << endl;
    // Get the hostname
    lpHost = gethostbyname(szHost);
```

```
        // Check whether we got something back
        if (lpHost != NULL) {
            // Load the server address with the host information
            saServerAddress.sin_addr.s_addr =
             ((LPIN_ADDR)lpHost->h_addr)->s_addr;
        }
        else {
            cout << "**ERROR** Could Not Locate Host" << endl;
            // Clean up WinSock
            WSACleanup();
            exit(1);
        }
    }
    // Set the server port
    saServerAddress.sin_port = htons(iPort);
    // Attempt to connect to the server
    iStatus = connect(skSocket,
     (struct sockaddr*)&saServerAddress,sizeof(sockaddr));
    // Check whether there was an error
    if( iStatus == SOCKET_ERROR ) {
        cout << "**ERROR** Could Not Connect To Server" << endl;
        // Clean up WinSock
        WSACleanup();
        exit(1);
    }
    cout << "<-- CONNECTED TO SERVER -->" << endl;
    // Load the data to send
    sprintf(szSendBuffer,"GET / HTTP/1.0\n\n");
    // Send the HTTP request
    iBytesSent = send(skSocket,szSendBuffer,256,0);
    cout << "<-- SENT " << iBytesSent << " BYTES -->" << endl;
    // Wait for incoming data
    iBytesReceived = recv(skSocket,szRecvBuffer,32768,0);
    cout << "<-- RECEIVED " << iBytesReceived << " BYTES -->" << endl;
    // Output the data received
    cout << szRecvBuffer << endl;
    // Close the socket
    closesocket(skSocket);
    // Clean up WinSock
    WSACleanup();
}
```

At the start of the program, I added a character buffer to hold the received data and an integer to tell me how many bytes I have received. The next noticeably different code, though, has resulted in a change from the last program. Instead of sending the sample string of "Test Data," this time I am sending an HTTP request message.

***HTTP (Hypertext Transfer Protocol)* is just a method of handling communication over the Internet. When a browser program goes to a Web site and downloads the Web page, all it is doing is passing commands via a socket connection.**

To request the information stored on the main page of a Web site, I first connect to the HTTP port number, which is 80. After I connect, I send a message containing the request for the Web page contents. The message I send contains the text "GET /HTTP/1.0\n\n". This message tells the Web server to send back the source code that makes up the Web page. After I submit the request message, the program goes into a receive loop and waits for the response. I then output the response. If everything goes well, you see HTML codes on your screen. Pretty cool, eh?



```
"G:\Source\Book\Chapter5\Receive_SocketObject\Release\Receive_SocketObject.exe"
<-- ATTEMPTING CONNECTION -->
<-- SOCKET CONNECTED -->
<-- SENT 256 BYTES -->
<-- RECEIVED 2048 BYTES -->
HTTP/1.1 200 OK
Date: Sat, 21 Oct 2000 21:49:09 GMT
Server: Apache/1.3.3
Connection: close
Content-Type: text/html

<html>
<head>
<title>Lost Logic Home Page</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<style>
<!--
.LostLogicPages { font-family: Arial; padding: 0 }
-->
</style>
</head>

<body bgcolor="#000000" text="#FFFFFF" link="#FFFF00" vlink="#00FF00" alink="#00
FFFF" topmargin="0" leftmargin="0">
<div align="left">
  <table border="0" cellpadding="0" cellspacing="0" width="100%">
    <tr>
      <td><img src="images/TitleUpperLeft.jpg" width="324" height="121"></td>
      <td background="images/TitleUpperScale.gif" width="100%">&nbsp;</td>
      <td><img src="images/TitleUpperRight2.gif" width="296" height="121"></td>
    </tr>
  </table>
</div>
<div align="left">


<div align="left">
  <table border="0" cellpadding="0" cellspacing="0" width="100%">
    <tr>
      <td width="141" valign="top" align="left" nowrap>
        <div align="left">
```

Figure 5.12

Output from lostlogic.com's Web server using the receive program

You call the recv() function by passing to it your socket handle, receive buffer, length of bytes to receive, and 0 for flags. Once you are done receiving, you output the contents of the receive buffer to the screen. That's it in a nutshell. Before looking at the C++ version of the code, take a look at the output from the receive program so far, as shown in Figure 5.12.

Receiving Data with C++

The C version of receiving data was really the hardest part. Adding it to the SocketObject is a walk in the park. Here is what the new class header looks like with the new Recv() function.

```
class SocketObject
{
    private:
    public:
        SOCKET     skSocket;
        SocketObject();      // Constructor
        ~SocketObject();     // Destructor
        int Bind(int iPort);
        void Disconnect();
        int Listen( void );
        bool Connect(char* szServerAddress, int iPort);
        int Send(char *szBuffer, int iBufLen, int iFlags);
        int Recv(char *szBuffer, int iBufLen, int iFlags);
}
```

As you can see, the Recv() function is just like the Send() function. They both take the same parameters, but they do have different valid values for the iFlags parameter. Let's move straight into the member function implementation.

```
int SocketObject::Recv( char *szBuffer, int iBufLen, int iFlags )
{
    return recv(skSocket, szBuffer, iBufLen, iFlags);
}
```

Not much new here to report. The function returns the number of bytes received. The first parameter of type character pointer holds the receive buffer. The second parameter, an integer, holds the size of the receive buffer. The third and last parameter, also type integer, sets how the receive function is called.

Okay, now that is out of the way, so take a look at the client implementation of the new C++ member function.

```
// Standard includes
#include <iostream.h>
#include "..\\socketobject\\SocketObject.h"

// ----> Main Program Function (REQUIRED)
void main( void )
{
    SocketObject        ClientSocketObject;
    int                 iPort = 80;     // HTTP Port
    char                szSendBuffer[256];
    int                 iBytesSent;
    char                szRecvBuffer[32768];
    int                 iBytesReceived;

    cout << "<-- ATTEMPTING CONNECTION -->" << endl;
    // Attempt to connect
    if ( ClientSocketObject.Connect( "www.lostlogic.com", iPort ) ) {
        cout << "<-- SOCKET CONNECTED -->" << endl;
        // Load the request buffer
        sprintf(szSendBuffer,"GET / HTTP/1.0\n\n");
        // Send the request
        iBytesSent = ClientSocketObject.Send(szSendBuffer,256,0);
        cout << "<-- SENT " << iBytesSent << " BYTES -->" << endl;
        // Receive the response
        iBytesReceived = ClientSocketObject.Recv(szRecvBuffer,32768,0);
        cout << "<-- RECEIVED " << iBytesReceived
         << " BYTES-->" << endl;
        // Output what was received
        cout << szRecvBuffer << endl;
    }
    else {
        cout << "**ERROR** Could Not Connect" << endl;
    }
    WSACleanup();
}
```

Here, you are using the HTTP request sample format again. You first send an HTTP request message and then retrieve the data from the response. The last step you follow is to output the data received.

One item of interest is that the data received is not a complete Web page. This is because TCP/IP does not guarantee how much data is transmitted in a single "burst." TCP/IP does guarantee delivery, though . . . just not when. Play around with looping through the `Recv()` function until the entire Web page is retrieved. Here's a hint: You have to use the `MSG_PEEK` flag to do it properly.

Review

It's that time again—time for a quick review of some key ideas learned in the chapter.

- C++ class objects automatically call their constructors when they are allocated memory. They also call their destructors when they are deleted or when they go out of program scope.
- You must make at least one call to `WSAStartup()` before using sockets. The function is used to load the socket dll.
- You should unload the sockets dll at program exit. This is accomplished by using the `WSACleanup()` function.
- The server must bind to a port number before it can listen for client connections.
- Always accept connections on port numbers greater than 1000. Many port numbers in the lower ranges are reserved for system ports.
- Try to limit the amount of data you send at a time. Many systems will end up breaking your packets into smaller pieces if they are too large. This can cause inefficiency in your communications.
- When you receive data, it is not always in one complete piece. You must make multiple calls to the receive function in these cases.

I hope you now have a good understanding of how to create a C++ socket class because you are finished with it for the time being. Look back over the source code and see whether there are items you didn't quite understand. Everything covered here is important to understand for the upcoming chapters, so re-read anything you need to. But as far as this chapter is concerned, it's time to use Private Hudson's saying in *Aliens*, "Game over, man. Game over!"

CHAPTER 6

Introduction to Windows Programming

When you think about learning Windows programming, the first phrase that comes to your mind might be, "You will be assimilated." Even though many refer to Microsoft as the Borg, its programming environment actually makes sense once you get the hang of it. If you are new to developing in Windows, then this chapter should be quite helpful to you. I'll cover everything from the message handler to multithreading. By the time this chapter is over, you will know how to write complete Windows programs with ease. Now, don't get me wrong; you won't be a Windows master or anything, but you will know enough to write games. And because games are all you care about knowing how to develop, this should be very fitting. Here are some of the topics I will cover.

- The history of Windows
- Windows architecture
- MFC
- Creating a window
- The message handler
- Multithreading

A Brief History of Time According to Gates

I am sure there is nothing you want to know more than how Windows evolved, right? Okay, you Linux folks can stop choking now. Really, it will be okay; just read on.

DOS

Unless you have been hiding in a bomb shelter since the Cuban Missile Crisis, you know that Bill Gates is the head founder of Microsoft. Bill Gates created Microsoft with some partners and released DOS (*Disk Operating System*), which was the original operating system for Microsoft PCs and still has ties to recent incarnations of Windows such as Windows 95 and Windows 98.

Most of us veteran game developers have used DOS for quite some time, and some still use it. It's not a great operating system, but it is extremely easy to develop for. With the use of interrupts and other tricks, you can even fake multitasking! But, alas, DOS is not a true multithreaded environment, nor is it a true 32-bit environment. Seemingly forever, DOS was limited to 640KB of program space. After all, who would ever need more than 640KB, right? It really makes you wonder sometimes how we got this far. Well, along came DOS extenders that allowed developers to access a lot more than 640KB. One of the most popular extenders used by developers was DOS/4GW. I personally used this extender because it came free with my Watcom C++ compiler. It's hard to beat free; ask Netscape if you have any doubts. If you have some old DOS games lying around, try loading some of them and I bet you will see DOS/4GW show up on the screen right before the game loads.

As they say, "All good things must come to an end." I bet a pessimist wrote that phrase; what do you think? DOS now looks to be on its way out permanently. Windows 2000 and Windows NT do not support it, and the next version of Windows might not either. Luckily, most of us game developers gave up on DOS a while back, but there are still a few hangers-on. If you haven't made the switch yet yourself, now you have the perfect opportunity.

Windows 1.0–2.0

Ack! I didn't even like Windows 3.1, much less 1.0. There is not much to say about Windows 1.0 other than it was horribly slow, it had a laughable interface, and it had no real-world use. Windows 1.0 was based on DOS and really didn't get a lot of market penetration. Windows 2.0 came out next but was still plagued with many of the original problems. I am not knocking the development teams here; I'm sure they did their best. The main problem was the architecture of the PC. Basically, multimedia hardware did not exist yet for the PC. Processors and video cards were just too slow to handle the load. As I said, not much to say here. I didn't even develop for this operating system; I stuck with DOS.

Windows 3.1–3.11

Oh, yes, here it is: Microsoft's answer to Apple's Macintosh operating system. Microsoft released Windows 3.1, which finally broke some of the ties to DOS. Although the operating system endured some major changes, it still lacked in usability. Windows 3.1 had a long way to go to reach the functionality of Apple Macintoshes or Commodore Amigas. I was still using Amiga computers at this time. The Amiga computer made by Commodore was seriously ahead of its time and even supported multitasking from the beginning. The Amiga died a horrible death some time later mainly due to bad management at Commodore. If nothing else, I learned never to put all of your eggs in one basket when it comes to technology. At least this school of thought has kept me from evangelizing one operating system over another. When it comes down to it, it really doesn't matter: Just use what is best for a given task. Okay, enough ranting already! I almost forgot Windows 3.11. Windows 3.11 added networking and some other features to Windows 3.1. Windows 3.11 was able to get a foothold on the market and finally forced Apple off its perch.

Windows 95

Windows 95 came onto the scene and gave new life to Windows. It featured a slick new interface that I am sure you know by now. Most of the features were similar to those used by Apple, but others were new. For instance, the taskbar is located at the bottom instead of the top. And the trashcan is now called the recycle bin. Amazing differences, no? Jokes aside, Windows 95 really was a great improvement over Windows 3.11. When Windows 95 came out, I finally came to the conclusion that DOS's days were numbered. After this epiphany, I bought a book on Windows programming and learned how to display a window. But there was something missing from my game programs—speed! Regardless of the countless promises, Windows 95 was slower running games than DOS.

A year later or so, Microsoft released the GDK (*Game Development Kit*). The GDK was a new development kit meant purely for games. It was kind of awkward and slow, but it worked. By version 5.0, the GDK was called DirectX and started showing great promise. This is where I picked up the ball and started learning how to do DirectX game programming in Windows. For DOS, it was time to leave; the tribe had spoken.

Windows 98

Windows 98 wasn't a huge change from Windows 95. It smoothed some rough edges Windows 95 had and added some nice touches. I've always thought of Windows 98 as a great big bug fix patch for Windows 95. I switched to it, and many hardware and software problems went away. All in all, I would say Windows 98 provided a stable platform for game development and entertainment. The only disappointment I had was that it added too few new features. Oh, well, you can't please everyone.

Windows 2000 and Others

I failed to mention that Windows NT came out back there somewhere. It is not really important to you in the context of this book because Windows NT does not support the latest versions of DirectX. I use Windows NT for corporate work but not game development. I suggest that you do the same and keep games on platforms intended for them. So what do I use? I use the glorious Windows 2000! Windows 2000, which was released fairly recently, uses a very tight system kernel and has the security features of Windows NT and the user interface of Windows 98. Also, Windows 2000 supports multiple processors. This is a great thing for those of us with dual-processor systems. If you can afford it and have not done so already, I suggest you build or buy a dual-processor machine. Computers run so much better with multiple CPUs.

TIP

If you want to build or buy a multiple CPU computer, be sure to purchase a copy of Windows 2000 or Windows NT for it. These are currently the only two operating systems from Microsoft that support multiple processors. If you have a choice, I recommend Windows 2000.

Windows 2000 supports the latest versions of DirectX and handles most games without any trouble. The only games I have seen that do not work with it are very old and are based on DOS.

The Architecture of Windows Programs

If your background is in developing DOS applications, then you are in for a bit of a shock. In fact, if you have been developing for anything other than Windows, a lot of the following information is going to seem strange at first. But it won't be that bad, I promise.

The first thing I need to cover about developing for Windows is sharing. In the good old DOS days, you didn't have to share anything. When your program executed, no other programs required attention. Your program was king of the world while it ran. This is not so in Windows.

Multitasking

Windows is a multitasking environment—multiple tasks can happen simultaneously. Now, the term *simultaneously* can be misleading here. In reality, nothing happens simultaneously in software. There is always an order of events. But multitasking gives the perception of simultaneous action with something I like to call time-slicing. *Time-slicing* means just what it says. Each process, or program, running gets a little slice of time from the operating system. Because each process gets a little slice, everything seems to run simultaneously over time. Think of it as a relay race where the baton is handed over infinitely. Each runner runs a few steps and then hands the baton to the next runner. Eventually, all of the runners get to run a few steps before the process starts all over again. Multitasking works in this way. Each program runs for a bit and then the next program gets to go. Because all of this happens so quickly, the user cannot perceive it. Refer to Figure 6.1 to see how time-slicing works on a single-processor system.

Figure 6.1 shows four programs running on a single-processor system. The item of interest here is the execution order list. In this list, you can see that each program

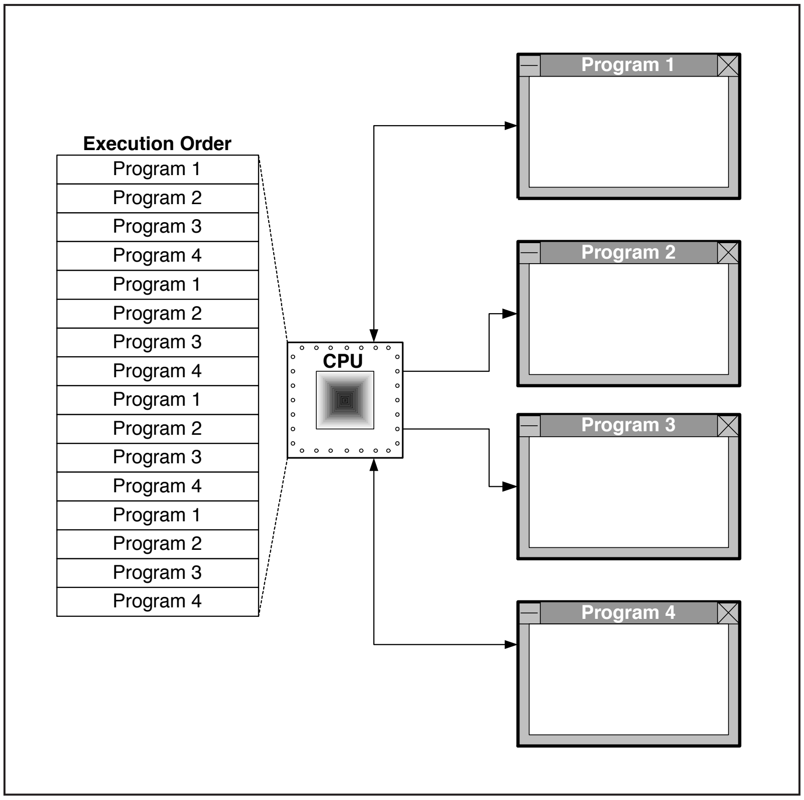


Figure 6.1

Time-slicing for a single-processor system gets a slice of time in which to run. Program 1 goes first, followed by 2, 3, and 4. Once all of the programs have a chance to run for a brief amount of time, the whole order starts over again. As you might have noticed, each program must wait for three others to go before it can run again. This is why your system slows down when running many applications; each one of them must wait its turn. The bright side of this is that Windows NT and Windows 2000 support multiple processors, as I mentioned before. Take a look at how two processors helps, shown in Figure 6.2.

In Figure 6.2, you see the same scenario as in Figure 6.1, but with two processors. Once again, the item of interest here is the execution order. Notice that now there are two execution order lists because there are two processors. The cool thing about having two lists is that the programs can be distributed. In this example, programs 1 and 2 run on the first processor, and programs 3 and 4 run on the second processor. This results in each program only having to wait on one other program

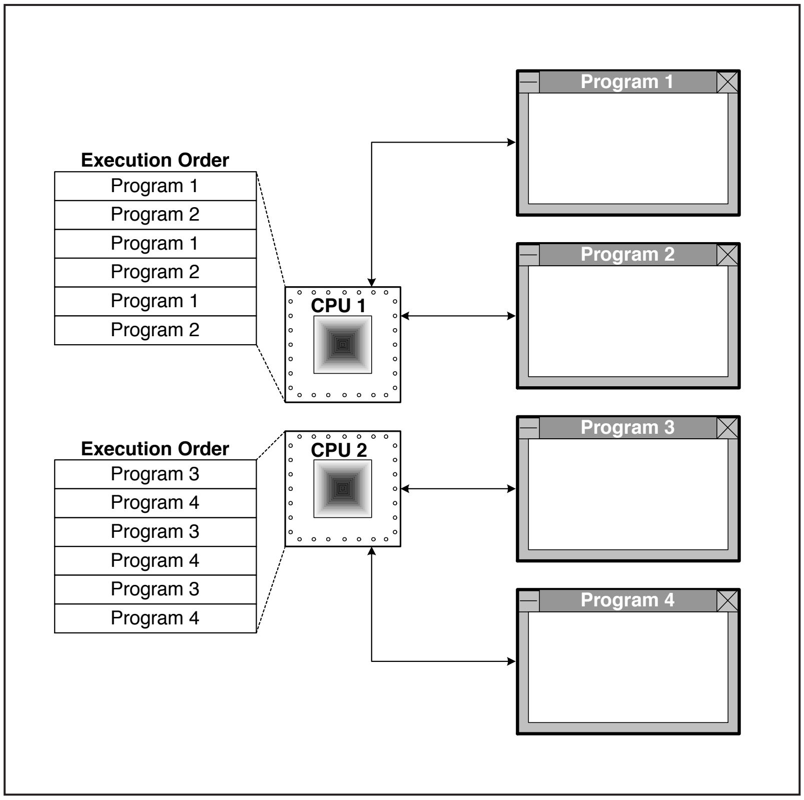


Figure 6.2

Time-slicing for a dual-processor system before it gets another slice of time! We have just moved from waiting on three programs to waiting on one. This is the beauty of using *nth-way* computers. Now imagine how much faster things would run on a four-processor machine or even an eight-processor machine. Most advanced servers on the market use at least four processors. Keep this in mind if you plan to develop an MMORPG.

Most server-oriented people refer to multiple processor machines as *nth-way* systems. For example, a two-processor machine is referred to as a two-way system. An eight-processor machine is an eight-way system.

Maybe you are asking, how does the system manage the processors? Actually, the operating system is responsible for all of it. If multiple processors were handled at the hardware level, then operating systems such as DOS would take advantage of them. But this is not the case. Windows handles the entire overhead for using multiple processors. This is a great thing considering what a pain it would be for the programmer to do it! This management does come at a price. Because Windows must keep track of what is going on for all of the processors, it does incur some overhead. For this reason, a dual-processor machine is not truly two times as fast as a single-processor computer. If you upgrade to a dual-processor machine, don't expect to see a 100 percent improvement. If fact, to see a huge speed increase, you have to run software that takes advantage of multiple processors. Software that takes advantage of multiple processors is multithreaded. Because I don't want to get too far off track here, I'll cover multithreading at the end of this chapter.

What Is a Window Anyway?

Now that you have been inundated with multitasking concepts, it's time to move on. So what is a window anyway? It is normally a transparent surface with supporting brackets. (I bet you saw that one coming.) Actually, a window is really an operating-system object provided as a front end for an application. For programmers, this means that a window provides standard methods for applications to send and receive data from users. Because the Windows operating system handles all of the generic functions, you can concentrate on writing actual functionality instead of having to worry about other things.

Take a gander at Figure 6.3. In this figure, you see a standard Windows program. It does nothing more than output the words "Hello World!" No big deal, right? Wrong. Break out the microscope and examine this program. First of all, you see a window on the screen. If you had to write this from scratch, you would have to display a background image that looks like a window. The next item of note is a button in the upper-right corner. When it is clicked, the program exits. If you had to program this by hand, you would have to add a hotspot to click, detect the mouse click, and then exit the program. Not to mention that you would have to show a little button-depressing animation, too. Ah, and what else do I see here? Scrollbars! If you drag the scrollbars around, the text moves in relation to them. Wow, what a pain that would be to program by hand. Not only would you have to update the scrollbars, but you would also have to implement scrolling of the text area.

Okay, I bet you have seen the light by now and can appreciate how helpful the windows layer is. The windows layer does countless other things for you, and I'll cover some of those things later. For now, it is enough to understand that the actual display, input, and output for a window are handled by the operating system.

Events That Don't Even Require a Ticket

Something a little different about Windows programming is the notion of *events*. Most programs you write for other operating systems are structured in a linear

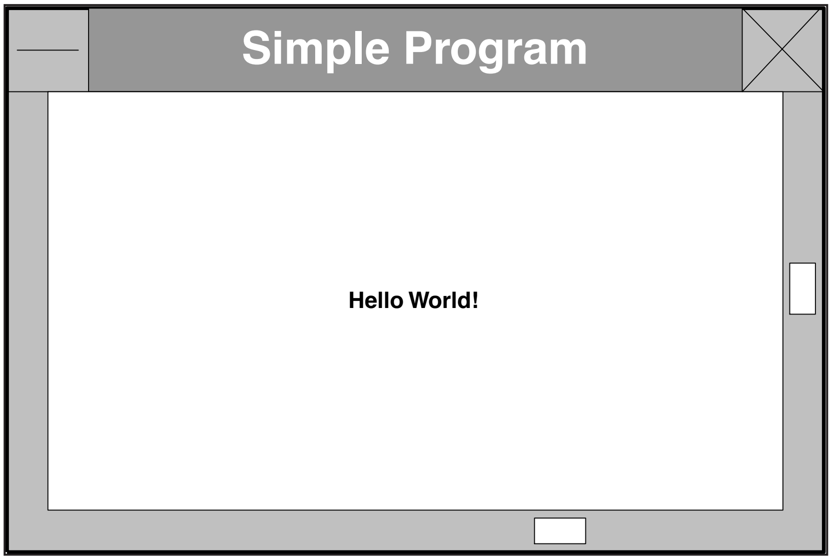


Figure 6.3

Output from a simple "Hello World" Windows program fashion. Normally, you do some processing, check for user input, and repeat. For example, take a peek at the following DOS program pseudo-code.

```
Begin Program
    Begin Main Loop
        Check for User Input
            User Hit "Fire"; Fire Laser
        Check If Player Died
            Player Died; Take Away One Life
        Process Game Objects
    Update Game Screen
    End Main Loop
End Program
```

That's a pretty standard game loop. You check for user input, check whether the player died, and then update the game screen. All of this happens in a tightly closed loop. In this example, the program is totally oblivious to anything else happening except for what is in the main loop. This, of course, is fine for a DOS program, but not so for a Windows program.

In Windows programs, everything happens via events. There are no closed loops where you wait for user input. Programs are structured with one loop where you check for events. When events happen, they are sent to an event-processing function that does whatever is needed.

To handle events, all Windows programs you write will use a *callback function.* Callback functions are triggered when specified criteria are met. For example, you could set up a callback function to be triggered when a certain amount of time has passed. For our use, I have a callback function that processes the Windows events or messages. So when the operating system sends an event to the Windows program, it is received and then dispatched to the callback function. Now that you have the absolute basics down, take a look at the Windows equivalent for the DOS pseudo-code.

```
Begin Program
    Create Window
    Begin Windows Loop
        Check for Events
            Got an Event; Send It to the Callback Function
        Process Game Objects
        Update Game Screen
    End Windows Loop

    Begin Callback Function
        Check Events
            Is This a "Fire" Event?
                Fire Laser
                Return to Windows Loop
            Is This a "Player Killed" Event?
                Take Away One Life
                Return to Windows Loop
    End Callback Function
End Program
```

In this example, you still process game objects and update the screen in the main loop. The only notable difference with this pseudo-code over the DOS version is the inclusion of a secondary function to process events. This might seem very similar; in fact, you could just create a secondary function in DOS to process game events. But with Windows, you must process messages and events from the main loop only. You cannot go off into a secondary function and wait for messages there. To better illustrate this, take a look at the following DOS pseudo-code.

```
Begin Program
    Begin Main Loop
        Check for User Input
            User Hit "Help"; Call Help Screen Function
        Process Game Objects
        Update Game Screen
    End Main Loop

    Begin Help Screen Function
        Display Help Graphic
        Begin Wait for Keypress Loop
            Wait for Keypress
        End Wait for Keypress Loop
        Erase Help Graphic
    End Help Screen Function
End Program
```

In this example, a help function is called when the user presses the help button. Once inside this function, the program displays the help screen and then waits for the user to press any key. When the user presses a key, the help screen is erased and the program's control returns to the main game loop. This is pretty standard game programming so far. Now, take a look at the difference with Windows.

```
Begin Program
    Create Window
    Set Current Menu to MENU_MAINGAMELOOP
    Begin Windows Loop
        Check for Events
            Got an Event; Send It to the Callback Function
        Which Menu Is the User Currently in?
            Is User in Help Menu (MENU_HELPSCREEN)?
                Call Help Screen Function
            Is User in the Main Game Loop (MENU_MAINGAMELOOP)?
                Process Game Objects
                Update Game Screen
    End Windows Loop

    Begin Callback Function
        Check Events
            Is This a "Help" Event?
                Set Menu to MENU_HELPSCREEN
                Return to Windows Loop
            Is This a "Keypress" Event?
                Set Keypress Equals to True
    End Callback Function

    Begin Help Screen Function
        Have We Displayed Help Graphic Yet?
            No, Then Display Help Graphic
        Does Keypress Equal True?
            Yes; Erase Help Graphic; Set Current Menu to MENU_MAINGAMELOOP
    End Help Screen Function
End Program
```

The Windows version might be a little difficult to follow at first. For one thing, you always come back to the Windows loop no matter what you might be doing. So when the user decided to see the help screen, you had to leave the main loop for a second and display the help screen with the help screen function. The program then returned to the main loop and checked for a keypress along with its usual chores. Once a key was pressed, a flag was set. The program then checked which menu to display. Because the current menu was set to `MENU_HELPSCREEN`, the program called the help screen function again. Once the keypress flag was set, the help screen function knew it was time to exit. The help screen function then cleared the help screen and set the current menu to `MENU_MAINGAMELOOP` so that the main loop would know not to come back.

Refer to Figure 6.4 to see a flowchart of the pseudo-code I just discussed. Nothing to really talk about figure-wise, so I'll just leave it alone. Okay, enough about events already!

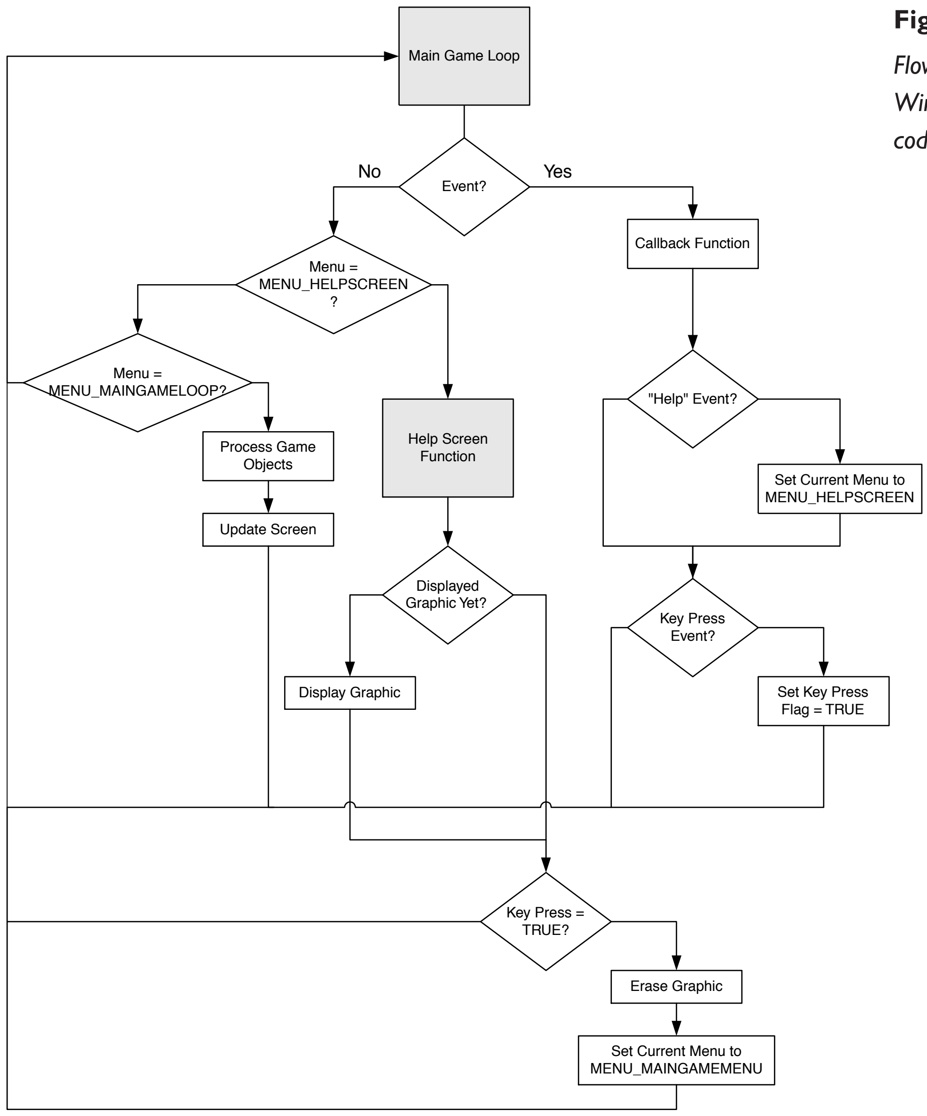


Figure 6.4

Flowchart for the Windows pseudo-code example

How to Make Sense of Windows Code

I know that the first time I ever looked at Windows code, I was completely confused. For one thing, all of the variables had strange prefixes such as `sz`, `lptr`, and `dw`. The prefixes were used all over the place and only seemed to clutter up the code. Much to my delight, I later figured out that unlike Microsoft Foundation Classes (MFC), there is a method to the madness. You see, at that time I was just a lone-wolf programmer and didn't have to share or look at anyone else's code. There was no real need, or so I thought, to have code standards just for myself. Well, this method doesn't work for large corporations or programming groups where multitudes of programmers work on the same code-set.

Hungarian Notation

Bear with me here while I expound upon the virtues of standardized coding practices. If you currently work or have ever worked at a large corporation with a group of programmers, you probably know why standardized coding practices help a great deal. When you have to read someone else's code, it can be a real pain in the rear to follow without standards. Imagine how hard it would be to read a book if there were no rules for writing English.

Enter stage left, standardized coding guy, Charles Simonyi. Charles Simonyi, of Hungarian origin, came up with a set of naming standards for variables, functions, and a few other things. Originally, these standards were part of his doctoral thesis. Simonyi's coding standards eventually became known as *Hungarian notation*. Hungarian notation has become standard at Microsoft—hence, the presence of it in all of Microsoft's code. Because the standards help developers read other developers' code, they are a great time- and frustration-saver. The main benefit is that everyone who understands the standards can look at most variables and functions and know the data types. Table 6.1 lets you peek at some of the notation.

Note that all of the prefixes shown here are not necessarily official Hungarian notation. Many of them are used due to popularity.

Maybe you want to see some of this in action, eh? Here is a sample of code I'll show you later in the book.

```
int     iBytesReceived = 0;       // # of bytes received
int     iBytesSent = 0;           // # of bytes sent
char    szCommand[128];           // Chat buffer
```

Table 6.1 Some Popular Hungarian Notation Prefixes

Prefix	Translation
sz	String with a null terminator. Basically a character array.
s	String without a null terminator.
c	Character, char.
b	Boolean. Used when the value is either 1 or 0.
i	Integer. Usually 32 bits.
l	Long. 'Nuff said.
ul	Unsigned long.
dw	A double word or DWORD.
lp, lptr	Long pointer.
g	Global variable. Sometimes used with an underscore as in g_iMyGlobal.
fn	Function pointer. Used when referencing a function's address.
h	Handle. Used for many Windows calls and multithreading.
v	Void.

The first two variables declared are integers, so they have i as a prefix. The next variable declared is a character buffer, so it has a prefix of sz. You might be wondering why this is such a big deal because you can just read the declaration to see the variable type anyway. Well, imagine you were using a library with no knowledge of how it was written. Maybe you want to use a function but are unsure of the return type. Look at how the following code handles this sort of situation.

```
vServerConnection( 6000, 6001 );
```

Well, the first thing you can tell right away from the function name is that it has a v for its prefix. Because the v stands for void, you can tell that it does not even return data. If the function name had no prefix, you would have to either guess or look at the prototype in the header file. I strongly suggest you adopt this style of coding if you have not already done so. It can be a huge time-saver, and it makes for readable code.

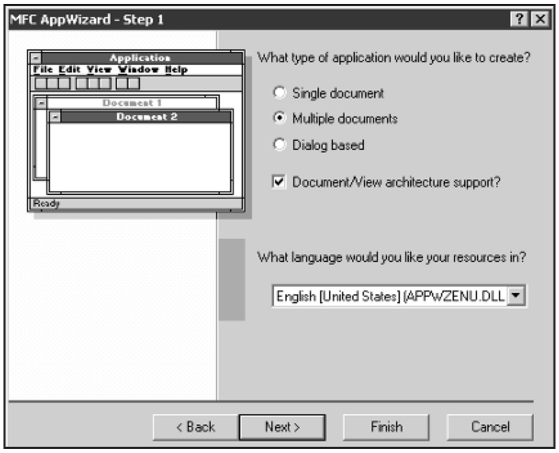
MFC, Insert Favorite Acronym Here

When you purchase most flavors of Microsoft's Visual C++ compiler, you also get the Microsoft Foundation Classes. *MFC* is just what the name implies, a set of foundation classes used when programming for Microsoft Windows. They are called *foundation classes* because most of the classes are built upon other classes and so forth. If you aren't up to speed on C++, I suggest you buy a book on it if you really want to understand Windows programming. *Kris Jamsa's Starting with Microsoft Visual C++* (Prima, 2001) is a good reference.

Why It's a Good Thing

First of all, MFC has many, many, many functions. Did I mention it has a lot of functions? The MFC libraries include a huge wealth of information and functions. With MFC, you can do everything from Internet programming to file management. In fact, there really isn't much you can't do with MFC. To use it, all you have to do is include some library files, and you are off and running. Because MFC builds upon layers of code, you can use some or all of the functions provided.

To facilitate the use of MFC, Microsoft offers a wizard to build applications in Visual C++. You simply create a new project and select MFC AppWizard, and you can create a new application. Figure 6.5 shows a step from the process.


MFC AppWizard - Step 1

What type of application would you like to create?

- Single document
- [✓ Multiple documents]
- Dialog based

[✓] Document/View architecture support?

What language would you like your resources in?

English [United States] (APPWZENU.DLL

Figure 6.5

Creating an MFC application using the AppWizard

I'm not going to cover how to use the application wizard, and you'll find out why in the "Why It's a Bad Thing" section that's coming up next. Basically, the wizard is great for quick prototyping or creating small GUI applications. Most serious programmers create their Windows code from scratch and skip using MFC altogether. Read on to find out why.

Why It's a Bad Thing

So what does MFC mean to us game programmers? Well, it means that Microsoft has written tons of nifty classes to make life easier for us. The problem is that they have good intentions with this method, but the execution of the plan is flawed in multiple ways. My main gripe about MFC is that if you ever try to link with libraries not written by Microsoft, you usually get hundreds of errors. Their libraries are great until you try to do something with them other than call MFC classes.

Another big problem is that programs written with MFC are generally huge in size compared to non-MFC solutions. Take, for instance, a program a friend of mine was recently writing. When he originally used MFC, the program was more than 1MB in size. When he switched to coding everything by hand, the program had the same functionality but was around 38KB in size! Now, this isn't the case if you link the MFC DLLs dynamically, but that is another area of trouble.

When you create an MFC application, you can link the libraries statically or dynamically. When you link using the static method, your program includes all of the MFC information, which results in a large file size. When you link dynamically, your program looks for the MFC DLLs in the Windows directory to read the information from there. This is the problem, though: Because there are many different versions of MFC out there, you have no idea which DLLs your program might end up calling. Even worse, sometimes when you install a program that uses MFC, it overwrites your DLLs and hoses some of your software!

A Little Compromise

I don't want this information to come across the wrong way. MFC is a great tool and it is useful for many applications. It is just not well suited for game development. Because game programmers are always looking to eke out the most speed from their applications as possible, MFC is just not suitable. But if you are prototyping a simple GUI program, or you aren't writing a game, then MFC could very well suit your task.

Creating a Window from Start to Finish

Hey, wow, we are finally to the good stuff! I think it's a fine time to show you how to open a window. The simplest Windows programming skeleton I use consists of two functions. The first function is responsible for defining and creating the window. The second function processes all events and messages. Oh, yeah, the first function also loops through getting messages and accepting user input until the user exits the program.

The All Important WinMain() Function

As I briefly mentioned a second ago, the first function defines and creates the window. The function is named `WinMain()`. You can think of it as the Windows version of `main()`.

Because you'll use the `WinMain()` function in all of your programs in this book, you can rely on the prototype to be consistent. Here is the prototype for `WinMain()`.

```
int WINAPI WinMain (HINSTANCE, HINSTANCE, PSTR, int )
```

TIP

Every Windows program you'll write requires the `WinMain()` function to be present.

The first parameter is of type `HINSTANCE`. This refers to the handle for the instance of the current application. Each time you run your program, an instance is created. If you run multiple copies of your program, then multiple instances are created.

The second parameter is also of type `HINSTANCE` and refers to the previous instance of the program. This is `NULL` for 32-bit operations, and you really don't need to worry about it as far as game programming is concerned.

The third parameter is of type `PSTR`, which is a pointer to the command-line string passed to your program. You can use this string to accept command-line-level directives from users. It is much like `argc` and `argv` in DOS. You don't use any directives from the command line, so I wouldn't worry about it too much either.

The last parameter is of type `int` and tells the window how it will first appear. Table 6.2 contains a few of the values available for this parameter.

Table 6.2 Parameter List for How a Window Will Be Shown

Value	Purpose
SW_SHOWMAXIMIZED	Displays the window maximized.
SW_SHOWMINIMIZED	Displays the window minimized.
SW_SHOWNORMAL	Displays the window in its default created state.

Coding the WinMain() Function

Okay, now that you have seen the prototype for WinMain(), take a look at a complete code listing for it.

```
//###############################################################################
// Standard windows include
#include <windows.h>

// Message loop callback function prototype
// ( REQUIRED FOR ALL WINDOWS PROGRAMS )
LRESULT CALLBACK fnMessageProcessor (HWND, UINT, WPARAM, LPARAM);

//
// Function to create the window and display it
// ( REQUIRED FOR ALL WINDOWS PROGRAMS )
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
 PSTR szCmdLine, int iCmdShow)
{
    HWND          hWnd;
    MSG           msg;
    WNDCLASSEX    wndclass;

    // Set up window attributes
```

```
wndclass.cbSize           = sizeof(wndclass);
wndclass.style            = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc      = fnMessageProcessor;
wndclass.cbClsExtra       = 0;
wndclass.cbWndExtra       = 0;
wndclass.hInstance        = hInstance;
wndclass.hIcon            = LoadIcon( NULL, IDI_APPLICATION );
wndclass.hCursor          = LoadCursor( NULL, IDC_ARROW );
wndclass.hbrBackground    = (HBRUSH) GetStockObject (WHITE_BRUSH);
wndclass.lpszMenuName     = NULL;
wndclass.lpszClassName    = "Hello World";        // Application name
wndclass.hIconSm          = LoadIcon( NULL, IDI_APPLICATION );

if( ( RegisterClassEx( &wndclass ) == 0 ) {
    // Do error logic here
    exit(1);
}

// Create the window
hWnd = CreateWindow(     "Hello World",      // Application name
             "Hello World Program",  // Name displayed on title bar
             WS_OVERLAPPEDWINDOW,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             NULL,
             NULL,
             hInstance,
             NULL );

// Display the window
ShowWindow( hWnd, iCmdShow );
UpdateWindow( hWnd );

// Process messages until the program is terminated
while( GetMessage ( &msg, NULL, 0, 0 ) )
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}
```

```
    return ( msg.wParam );
}
```

See, that wasn't so bad. The code listing here shows the complete `WinMain()` function. Don't try to create a new Windows32 application space in Visual C++ just yet because you are still missing a major portion of the code. Take a peek at what the output would look like from this program if it were complete, as shown in Figure 6.6.

As you can see, the program does not do very much except sit there and look boring. But hey, it really does do quite a bit compared to what it would take to create it without Windows! The first item of interest is the content of the `WinMain()` function. We talked plenty about the prototype but never discussed what actually goes into the function. The most important variable declared is `wndclass`.

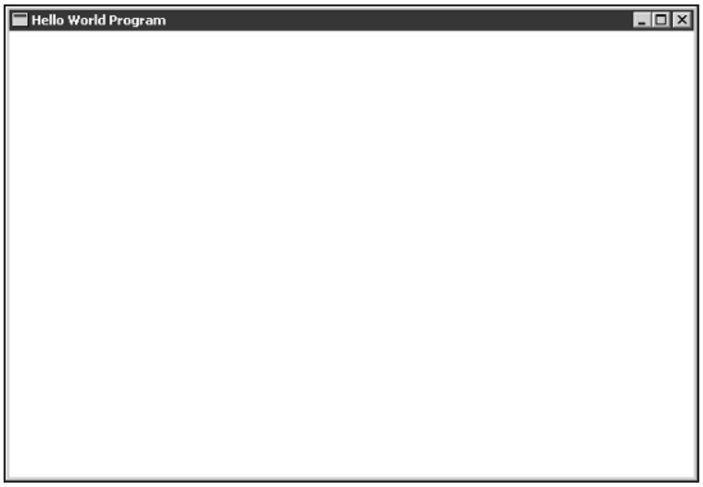


Figure 6.6

Output from your first Windows program

Setting Up the Window's Attributes

`wndclass` is of type `WNDCLASSEX`. `WNDCLASSEX` is really just the definition of a data structure containing information about a window. So `wndclass` is the data structure containing information about your future window. The first code you see besides the variable declarations is attribute initialization code. You specify how the window will look by setting various parameters in the `wndclass` data structure.

```
// Set up window attributes
wndclass.cbSize          = sizeof(wndclass);
wndclass.style           = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc     = fnMessageProcessor;
wndclass.cbClsExtra      = 0;
wndclass.cbWndExtra      = 0;
wndclass.hInstance       = hInstance;
wndclass.hIcon           = LoadIcon( NULL, IDI_APPLICATION );
wndclass.hCursor         = LoadCursor( NULL, IDC_ARROW );
wndclass.hbrBackground   = (HBRUSH) GetStockObject (WHITE_BRUSH);
wndclass.lpszMenuName    = NULL;
wndclass.lpszClassName   = "Hello World";           // Class name
wndclass.hIconSm         = LoadIcon( NULL, IDI_APPLICATION );
```

You must set several members in the `wndclass` data structure. Table 6.3 provides a breakdown of what each member means for you.

TIP

A very good source of information is Microsoft's MSDN Library for Visual C++. It contains a great wealth of information about C++'s functions and Windows programming.

Registering the Class

Now that you have set up all of the parameters for your class, you must tell Windows that it exists. You do this by *registering* the class with the `RegisterClassEx()` function.

The `RegisterClassEx()` function tells Windows about your class so that you can use it later when displaying a window with it. When a class is registered, you are allowed to use it by its name. In our working example, the class is named "Hello World." Take a quick look at the function prototype.

```
ATOM RegisterClassEx ( CONST WNDCLASSEX *lpWndClass )
```

The first and only parameter you pass to the function is a pointer to your class object. In your case, this is the `wndclass` object. The only other item of note is the return type of `ATOM`. Personally, I never have bothered with the return type when it succeeds; I only check whether it fails. If the function fails, a `0` is returned. So all you have to do is make sure the function doesn't return a `0`! Here is the code you used from the previous example.

```
If( ( RegisterClassEx( &wndclass ) == 0 ) {
    // Do error logic here
    exit(1);
}
```

All you do is check whether the class returns a `0`. If it does, you call the `exit()` function, which causes your program to abruptly end. This is not a recommended way to

Table 6.3 Member Variables for WNDCLASSEX

Value	Purpose
`cbSize`	Size of the `WNDCLASSEX` data structure.
`style`	Specifies the style of the window class. A sample value is `CS_NOCLOSE`, which disables the close window button. These values can be combined using the `OR (|)` operator.
`lpfnWndProc`	This one is very important—it is a pointer to the message-processing function. You must populate this with a function name if you want to process Windows messages and events. The pointer goes to a callback function that processes all the messages that come to this window.
`cbClsExtra`	Number of bytes allocated following the class structure. You will always use 0 for this.
`cbWndExtra`	Number of bytes allocated following the Windows instance. You will always use 0 for this.
`hInstance`	Handle to the instance of the window where the class will be running. This normally is populated with the `hInstance` variable passed to `WinMain()`.
`hIcon`	Handle to the icon the application will use. If it is set to `NULL`, then the application is responsible for drawing its own icon.
`hCursor`	Handle to the cursor the application will use. If it is set to `NULL`, then the application is responsible for setting the shape of the mouse when it is present within the window.
`hbrBackground`	Handle to the background brush used in the window. In my example, I set it to white.
`lpszMenuName`	Pointer to a string containing the name of the menu resource to be used by the program. Because most of your games are full-screen, keep this at `NULL`.
`lpszClassName`	A string pointer that contains the name of the class or application.
`hIconSm`	Handle to an icon used when the program is represented by a small icon.

leave your program, but it demonstrates the point. So what does registering do for you anyway? Well, it lets you use the `CreateWindow()` function to actually create your window. All this work wouldn't be very useful if you couldn't create a window!

Creating the Window

Ah, now you have created a window class and registered it, so it's time for the good stuff. Creating your window is probably one of the easiest parts because it requires only a single function, `CreateWindow()`. Here is the prototype for the `CreateWindow()` function.

```
HWND CreateWindow (
     LPCTSTR       lpClassName,     // Pointer to registered class name
     LPCTSTR       lpWindowName,    // Text displayed on the title bar
     DWORD         dwStyle,         // Window style
     int           x,               // Horizontal position of window
     int           y,               // Vertical position of window
     int           nWidth,          // Window width
     int           nHeight,         // Window height
     HWND          hWndParent,      // Handle to parent or owner window
     HMENU         hMenu,           // Handle to menu or child-window identifier
     HANDLE        hInstance,       // Handle to the instance of this program
     LPVOID        lpParam          // Pointer to window-creation data
)
```

The first parameter, `lpClassName`, is a pointer to the name of your class. Because you named the class "Hello World," you use "Hello World" for the parameter here. As long as you keep your class names unique, it really doesn't matter what you call them.

The next parameter, `lpWindowName`, is a pointer to the text that will be displayed on the title bar. In your example, you are using "Hello World Program" as your title bar text. Please note that not every `wndclass` you create for Windows is used to display a window. If you had created a button class, then the text specified here would appear on the button.

`dwStyle`, which comes next, denotes the style of the window you are creating. The styles available can be combined with standard `OR (|)` logic. In Table 6.4, you can see a condensed list of available styles.

The next four parameters, `x`, `y`, `nWidth`, and `nHeight`, are pretty straightforward. They set the size and position of the window. Because you just rely on Windows to display the default size and position, you set these parameters to `CW_USEDEFAULT`. Of course, you can alter the size and position if you want to; you just don't need to for this example. Oh, yeah, I almost forgot: The values represent pixel coordinates. So if your screen is set to 1024 × 768 resolution, then you have 1,024 pixels on the x-coordinate and 768 pixels on the y-coordinate.

Table 6.4 Window Styles

Value	Purpose
`WS_BORDER`	The window is created with a thin border.
`WS_CHILD`	The window is a child window.
`WS_DISABLED`	The window is displayed inactive. No input can be received from the user.
`WS_DLGFRAME`	The window is dialog style, which means it has no title bar.
`WS_HSCROLL`	The window has a horizontal scrollbar.
`WS_VSCROLL`	The window has a vertical scrollbar.
`WS_THICKFRAME`	The window has a sizing bar.
`WS_MINIMIZE`	The window starts out minimized.
`WS_MAXIMIZE`	The window starts out maximized.

You are almost done with the function; hang in there! The next parameter, `hWndParent`, is of type `HWND` and takes a handle to the parent window. Because you have no parent window, this is set to `NULL`. The next parameter, `hMenu`, is also a handle, but of type `HMENU`. This parameter points to a menu or a child-window ID, depending on the class type. If it points to a menu object, then the menu is used with the window.

The next to last parameter, `hInstance`, which is type `HINSTANCE`, is a handle to the instance of your application. You just use the `HINSTANCE` that was passed to `WinMain()`. The final parameter, `lpParam`, of type `LPVOID`, points to some Windows-specific creation data. You don't use this value, so just set it to `NULL`. Feel free to find a Windows programming book and read more on this subject if you are interested in this parameter. Personally, I have never had to use it for my games.

Now that you have gone over the `CreateWindow()` function, take a look at the code for it one last time.

```
hWnd = CreateWindow(     "Hello World",          // Application name
                         "Hello World Program",  // Name displayed on title bar
                         WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          NULL,
                          NULL,
                          hInstance,
                          NULL );
```

Figure 6.7 shows a diagram of what you have done so far. You have created a Windows class object, registered it, and then created a window. Now, it's time to display it.

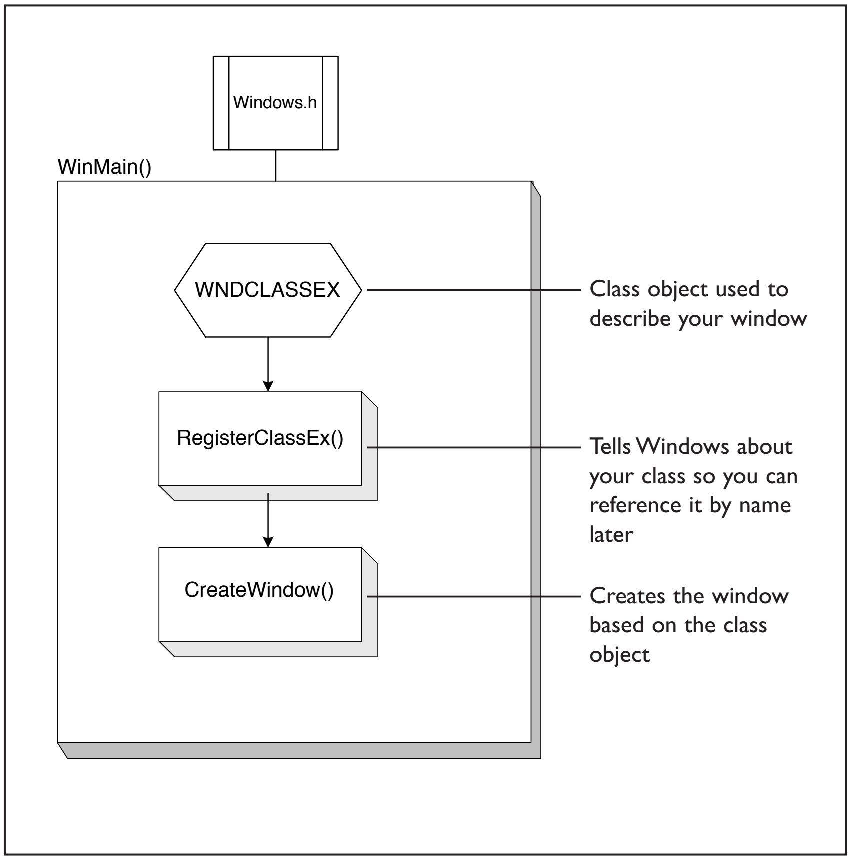

Windows.h
WinMain()
WNDCLASSEX
RegisterClassEx()
CreateWindow()
Class object used to describe your window
Tells Windows about your class so you can reference it by name later
Creates the window based on the class object

Figure 6.7

`WinMain()` *before* `ShowWindow()`

Displaying the Window Using ShowWindow()

You display the window using the `ShowWindow()` function. This function does more than just show a window, though. It actually sets the display state of the current window. Table 6.5 contains some of the display states you can specify.

Oops, I almost forgot to show you the function prototype, so here it is.

```
BOOL ShowWindow( HWND hWnd, int nCmdShow )
```

The first parameter the function expects is `hWnd`. `hWnd` is nothing more than a handle to the window you want to show. The second and final parameter is an integer named `nCmdShow`. The `nCmdShow` parameter expects display state values like the ones shown in Table 6.5. You just use the value passed to `WinMain()`. Here is the code used to call this function.

```
ShowWindow( hWnd, iCmdShow );
```

See, that wasn't very complicated!

Table 6.5 Window Display States with ShowWindow()

Value	Purpose
`SW_SHOW`	Activates and displays the window in its current state.
`SW_HIDE`	Hides the window and activates another window.
`SW_SHOWNORMAL`	Activates and displays the window at its default state. You should use this when calling the function for the first time.
`SW_SHOWNA`	Displays the window in its current state but leaves the currently active window active.
`SW_RESTORE`	Restores and displays the window. You can use this to restore the window to its original state if it was minimized or maximized.
`SW_MINIMIZE`	Displays the window in a minimized state and activates the next window in line.
`SW_MAXIMIZE`	Displays the window in a maximized state.
`SW_SHOWMAXIMIZED`	Activates the window and displays it maximized.
`SW_SHOWMINIMIZED`	Activates the window and displays it minimized.

Updating the Window Using UpdateWindow()

Now that you have set the display state for the window, it is time to update it. Here's the prototype to do so.

```
BOOL UpdateWindow( HWND hWnd )
```

Once again you see the `hWnd` parameter here. It expects a handle to the window you want to update, just as `ShowWindow()` does. The update function actually sends a `WM_PAINT` message to the event function. This message tells the window to update its client area. Whenever you have changed the display in a window, you can force it to update by calling this function.

The Message-Processing Loop

Now that you have created and displayed a window with `WinMain()`, it is time to accept messages from the application. You accomplish message processing for your sample application using three functions: `GetMessage()`, `TranslateMessage()`, and `DispatchMessage()`. Here is a refresher of the code you are using.

```
// Process messages until the program is terminated
while( GetMessage ( &msg, NULL, 0, 0 ) )
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}
```

Inside the GetMessage() Function

The `GetMessage()` function retrieves all messages from a calling thread's message queue for the specified window. I know this sounds complicated, but all it really means is that the function retrieves the next message waiting for the window. Here is the prototype.

```
BOOL GetMessage( LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin,
 UINT wMsgFilterMax )
```

The first parameter expects a pointer to an `MSG` structure. The `MSG` structure contains all of the information about the retrieved message. Here are the inner workings of the `MSG` structure.

```
MSG {
    HWND          hwnd;
    UINT            message;
    WPARAM          wParam;
    LPARAM          lParam;
    DWORD           time;
    POINT           pt;
}
```

hwnd points to the window that receives the message. The message field specifies the message number. wParam holds additional information about the data depending on the value of the message number. lParam is just like wParam; it holds additional data about the message. The time variable does just what it implies—it stores the time the message was generated. The last field, pt, stores the screen coordinates of the cursor when the message was generated.

The next parameter, hWnd, should look familiar by now. All it expects is a handle to the window from which to get messages. The last two parameters, wMsgFilterMin and wMsgFilterMax, specify the lowest and highest value messages to retrieve. Because you don't want to limit the message range, just set these both to 0.

The function returns a non-zero if successful. The function returns a 0 if a WM_QUIT message is present. This is why you have the while loop around the call; you just stay in the loop until the user exits the program.

CAUTION

The GetMessage() function can return a -1 if the call fails. You should code for this accordingly.

Inside the TranslateMessage() Function

The TranslateMessage() function is much simpler than the GetMessage() function. Here is the prototype.

```
BOOL TranslateMessage( CONST MSG *lpMsg )
```

The only parameter you have to worry about for this function is lpMsg. lpMsg is just a pointer to an MSG structure that you previously got from GetMessage(). If you want to, you can populate an MSG structure manually and pass it to this function.

This function takes the MSG data sent to it and translates any virtual-key messages it finds into character messages. When it finds these messages, it adds them to the queue to be read by the next GetMessage() function call. So really, all this function does is check for keyboard events and then post them to the queue. Oh, yeah, I almost forgot: This function does not alter the MSG data passed to it.

Inside the DispatchMessage() Function

The last of the three functions you use for handling messages is `DispatchMessage()`. This function works just like `TranslateMessage()` in that it expects a pointer to an `MSG` structure. Here is the prototype.

```
LONG DispatchMessage( CONST MSG *lpMsg )
```

The `MSG` passed to this function is sent to the event procedure specified for the current window. In this case, it goes to the `fnMessageProcessor()` function. The `DispatchMessage()` function is what allows your program to process the messages it receives. If you never call this function, messages are never processed.

All right, time for review. To handle messages for your program, do the following.

1. Get the next message.
2. Translate the message for virtual-key events.
3. Dispatch the message to the message processor.
4. Repeat the process.

That's it for `WinMain()`!

Handling Window Messages and Events

I promised you that there would be only two functions involved in our sample application, and I am here to deliver. The second function, `fnMessageProcessor()`, handles all of the messages generated for your window.

As you learned just a minute ago, all messages passed to the `DispatchMessage()` function are sent to the window's message-processing function. This processing function is declared in the class object describing the window. Recall that the code looked something like this:

```
wndclass.lpfnWndProc      = fnMessageProcessor;
```

The value of `fnMessageProcessor` tells you that all messages dispatched should be sent to the `fnMessageProcessor()` function. Okay, take a look at the code for this function.

```
//
// Message loop callback function
// ( REQUIRED FOR ALL WINDOWS PROGRAMS )
//
```

```
LRESULT CALLBACK fnMessageProcessor ( HWND hWnd, UINT iMsg,
 WPARAM wParam, LPARAM lParam )
{
     switch( iMsg )
     {
          // Called when window is first created
          case WM_CREATE:
               return( 0 );
          // Called when the window is refreshed
          case WM_PAINT:
               return( 0 );
          // Called when the user closes the window
          // or terminates the application
          case WM_DESTROY:
               PostQuitMessage( 0 );
               return( 0 );
     }

     return DefWindowProc( hWnd, iMsg, wParam, lParam );
}
```

The Message Processor Function

Well, the first thing you can examine is the function prototype used for your message processor.

```
LRESULT CALLBACK fnMessageProcessor (
                           HWND hWnd,
                           UINT iMsg,
                           WPARAM wParam,
                           LPARAM lParam
                           )
```

The first parameter, `hWnd`, contains the handle to the window for which you are processing messages. The next parameter, `iMsg`, contains the message identifier. The identifier can be one of many different Windows message types. Some of the more valuable IDs are shown in Table 6.6.

The last two parameters passed to the function, `wParam` and `lParam`, contain extra information about the message. For example, when you receive a `WM_KEYDOWN` message, the virtual-key code is stored in the `wParam`. I'll talk about more uses for these values later. Now that you have the prototype down, it's time to see what the function actually does.

Table 6.6 Valuable Windows Message IDs

Value	Purpose
WM_CREATE	The window is created using CreateWindow().
WM_PAINT	The window needs to be redrawn.
WM_QUIT	The user has closed the window.
WM_TIMER	A timer expires—you can use this for special events.
WM_KEYDOWN	The user has pressed a key.
WM_KEYUP	The user has released a key.
WM_MOUSEMOVE	The user has moved the mouse.
WM_ACTIVATE	The window is activated or deactivated.
WM_SIZE	The window has been sized.
WM_LBUTTONDOWN	The user has the left mouse button pressed.
WM_LBUTTONUP	The user has the left mouse button released.
WM_RBUTTONDOWN	The user has the right mouse button pressed.
WM_RBUTTONUP	The user has the right mouse button released.
WM_SETFOCUS	The window has keyboard focus.
WM_DESTROY	It is time to close the window.

Once the program is inside the message processor function, all it has to do is check for each type of message and act accordingly. In the example I am using, I only check for WM_CREATE, WM_PAINT, and WM_DESTROY. By checking these messages, you can handle it when the window is created, updated, or closed.

The Default Message Processor

The last thing you do before leaving the function is return the results of the DefWindowProc() function. This function is the default message processor provided by Windows. It processes any messages you left behind. Because it is never a good idea to let messages pile up, this function is handy for cleaning up. This function takes the same parameters as the other message processor, so I think it's safe to move on.

The Complete Code Listing

I never like leaving people with incomplete code examples, so here is the complete listing for the Windows program.

```
//###########################################################################
// Standard Windows include
#include <windows.h>

// Message loop callback function prototype
// ( REQUIRED FOR ALL WINDOWS PROGRAMS )
LRESULT CALLBACK fnMessageProcessor (HWND, UINT, WPARAM, LPARAM);

//
// Function to create the window and display it
// ( REQUIRED FOR ALL WINDOWS PROGRAMS )
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
 PSTR szCmdLine, int iCmdShow)
{
    HWND          hWnd;
    MSG           msg;
    WNDCLASSEX    wndclass;

    // Set up window attributes
    wndclass.cbSize           = sizeof(wndclass);
    wndclass.style            = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc      = fnMessageProcessor;
    wndclass.cbClsExtra       = 0;
    wndclass.cbWndExtra       = 0;
    wndclass.hInstance        = hInstance;
    wndclass.hIcon            = LoadIcon( NULL, IDI_APPLICATION );
    wndclass.hCursor          = LoadCursor( NULL, IDC_ARROW );
    wndclass.hbrBackground    = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wndclass.lpszMenuName     = NULL;
    wndclass.lpszClassName    = "Hello World";     // Application name
    wndclass.hIconSm          = LoadIcon( NULL, IDI_APPLICATION );

    if( ( RegisterClassEx( &wndclass ) == 0 ) {
        // Do error logic here
        exit(1);
    }

    // Create the window
    hWnd = CreateWindow(     "Hello World",     // Application name
                 "Hello World Program",  // Name displayed on title bar
                 WS_OVERLAPPEDWINDOW,
                 CW_USEDEFAULT,
                 CW_USEDEFAULT,
                 CW_USEDEFAULT,
                 CW_USEDEFAULT,
                 NULL,
                 NULL,
                 hInstance,
                 NULL );

    // Display the window
    ShowWindow( hWnd, iCmdShow );
    UpdateWindow( hWnd );

    // Process messages until the program is terminated
    while( GetMessage ( &msg, NULL, 0, 0 ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }

    return ( msg.wParam );
}
//
// Message loop callback function ( REQUIRED FOR ALL WINDOWS PROGRAMS )
//
LRESULT CALLBACK fnMessageProcessor ( HWND hWnd, UINT iMsg,
 WPARAM wParam, LPARAM lParam )
{
    switch( iMsg )
    {
        // Called when window is first created
        case WM_CREATE:
             return( 0 );
        // Called when the window is refreshed
        case WM_PAINT:
            return( 0 );
        // Called when the user closes the window
        // or terminates the application
        case WM_DESTROY:
            PostQuitMessage( 0 );
             return( 0 );
    }

    return DefWindowProc( hWnd, iMsg, wParam, lParam );
}
//################################################################################
```

Oh, sheesh, I forgot to talk about something—the include file! The first lines of code include the most important header file you use for Windows programming. It is aptly named windows.h. So be sure to include it whenever you write a Windows program.

Basic Theory of Multithreading

Now do an about-face and switch back to theory mode. The last topic I want to cover is *multithreading*. Multithreading is one of those things that if you don't bite the bullet and start using it, you probably will never understand it fully.

The only thing multithreading does is take advantage of time-slicing. If you skipped the beginning of this chapter, I suggest you go back and read the "Multitasking" section. If you recall, time-slicing just gives different programs slices of time to use the CPU.

Programs that do not take advantage of multitasking run serially. When a program runs *serially*, it executes a command, waits on the command to finish, and then executes the next command. This is fine for most programs, but it is not the best method for some games. Take, for instance, an arcade game. What would happen if someone put a quarter in the machine while you were in some huge logic loop? You would probably miss the quarter-drop signal. If your program had a process listening for the quarter drop while you were doing your logic loop, you would be fine. You would accomplish this by launching a thread to listen for the quarter drop.

Stack Space

Threads are nothing more than small pieces of executing code. They are functions that operate on their own once launched. Because threads act like little self-contained programs, they take up stack space. *Stack space* is required to hold static memory within the thread function. If you plan to hold a lot of static memory within your threads, be sure to create them with plenty of stack space, or your program might crash. Take a look at how the quarter-drop scenario is handled with a thread in Figure 6.8.

From the diagram, you can see that the main program launches a thread to handle quarter processing. Because the thread is operating on its own, no quarters are missed while the main program is running its logic loop. In this example, you would have to make sure to tell the thread to shut down when the main program exits.

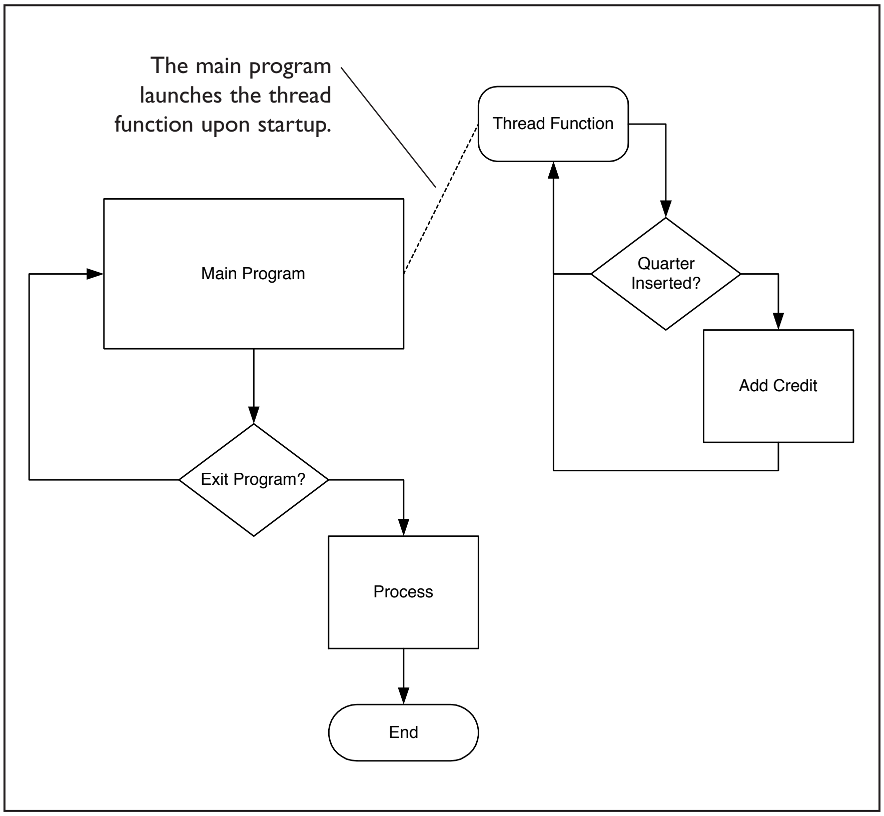


Figure 6.8

Detecting an inserted quarter

Communicating with Threads

The simplest way to communicate with a thread is to change a variable that is shared with it. When you create a thread, you can specify the address of a memory block you want to share with it. Commonly, you send the address of a data structure. By passing the address of a data structure, you can share all sorts of variables within the structure.

Because the memory is shared, you can change a variable in the main program and the thread will see it. This is how you can communicate to the thread. Imagine passing a quit variable to the thread in the quarter-drop function. The thread checks the quit variable until it equals 1. If the quit variable equals 1, then the thread exits. All the main program has to do is set quit to 1 before closing down. The thread can communicate to the main program in a very similar way. You just allocate a variable in the shared data structure for it. Figure 6.9 illustrates how the quarter-drop program can communicate with its thread.

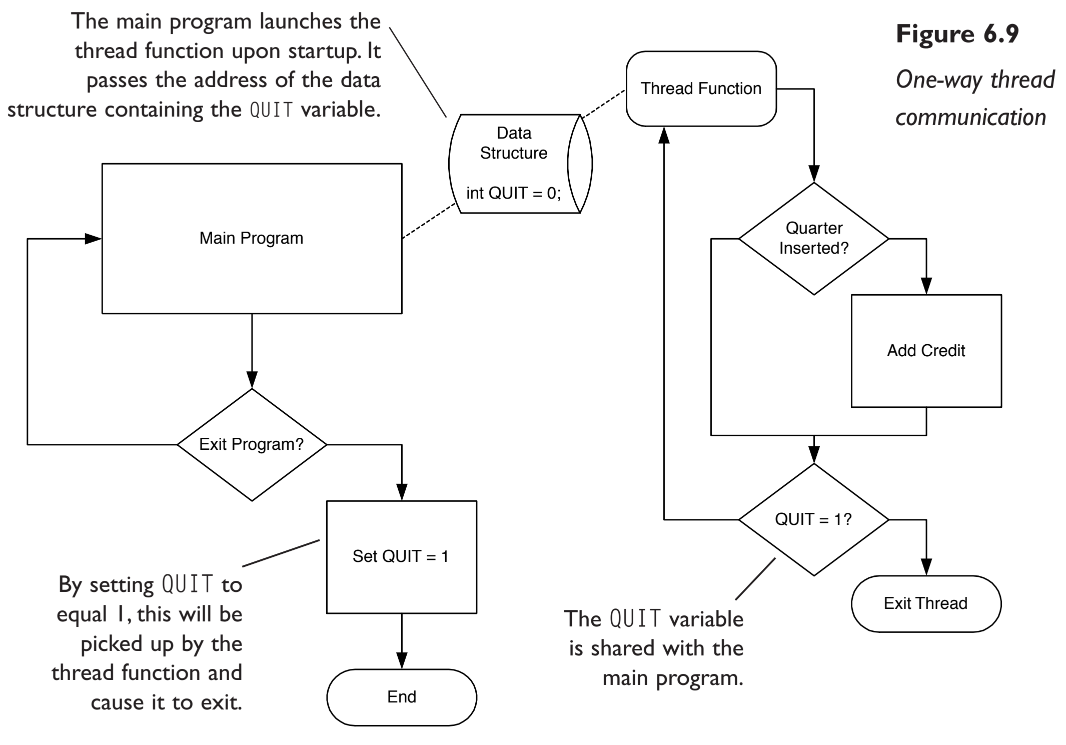

The main program launches the thread function upon startup. It passes the address of the data structure containing the QUIT variable.
Data Structure
int QUIT = 0;
Thread Function
Main Program
Quarter Inserted?
Add Credit
Exit Program?
QUIT = 1?
Set QUIT = 1
Exit Thread
End
By setting QUIT to equal 1, this will be picked up by the thread function and cause it to exit.
The QUIT variable is shared with the main program.

Figure 6.9
One-way thread communication

The main thing to remember about sharing memory with threads is that you do not try to write to the same memory from both the thread and the main program. This can cause serious problems.

Thread Synchronization with Mutexes

If you just had to have memory that both the main program and threads could write to, then you could arrange it—with a little item called a mutex. *Mutex* stands for mutual exclusion. Basically, a mutex allows you to lock and unlock shared memory between threads. Because your main program is really just one big thread, you can use a mutex with it, too.

To try a mutex, add some new functionality to the arcade game. Now that you have a thread listening for and adding up credits, you also want the main program to be able to deduct credits. How else do you make any money, right? So you need to add a variable to your shared memory to hold the number of credits the player has left. The thread function needs to add credits, and the main program needs to deduct credits. To do this, you create a credit-lock mutex and put it around the calls that affect the credits (see Figure 6.10).

Okay, this might look a little complicated, but it's really not too bad. All you had to do out of the ordinary was put mutex calls before and after any operations writing to the credits variable. Before you write to the credits variable, you call a function to wait for the mutex to become available. If no other processes are using the mutex, you are returned ownership of it. If another process is using the mutex, you just wait for it to become available. After you have ownership, you can then write to the credits variable. As soon as you are done with the credits variable, release the mutex so the next process can use it. If you never release the mutex, then no other process is ever able to use it.

Just remember that you are not locking the credits variable; you are locking whatever happens within the code between the wait and release functions of the mutex. Be careful not to lock a mutex, and never release it by accident. This can be the cause of much frustration when trying to debug the problem.

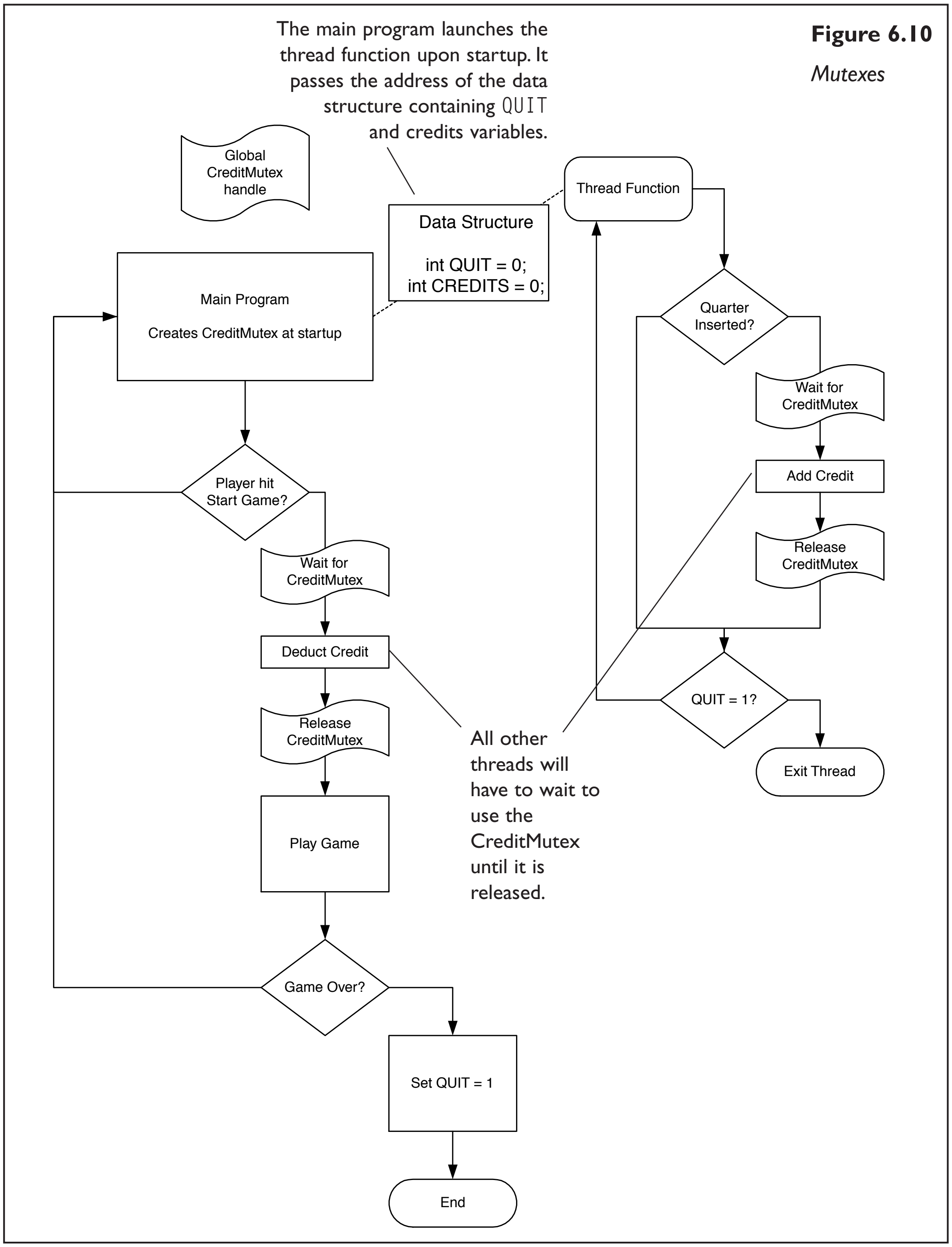

The main program launches the thread function upon startup. It passes the address of the data structure containing QUIT and credits variables.
Global CreditMutex handle
Data Structure
int QUIT = 0;
int CREDITS = 0;
Thread Function
Main Program
Creates CreditMutex at startup
Quarter Inserted?
Wait for CreditMutex
Add Credit
Release CreditMutex
Player hit Start Game?
Wait for CreditMutex
Deduct Credit
Release CreditMutex
Play Game
All other threads will have to wait to use the CreditMutex until it is released.
QUIT = 1?
Exit Thread
Game Over?
Set QUIT = 1
End

Figure 6.10

Mutexes

Creating a Thread Using CreateThread()

I bet you wouldn't be very happy if all I did was talk about the theory of threads and never showed you any actual code. Okay, so here it is: Some actual code for creating a simple thread like the one used in the arcade simulator.

```
// Memory shared with thread
struct SharedMemory
{
    int      iQuit;
    int      iCredits;
};
SharedMemory g_SharedMem;

// Thread variables
DWORD           dwCreditsID;
HANDLE          hCreditsThreadHandle;

// Create the CreditThread here
hCreditsThreadHandle = CreateThread(
            NULL,                 // Security, default is okay
            NULL,                 // Initial stack size, default okay
            (LPTHREAD_START_ROUTINE ) &thrCreditsThread,
            // The thread function
            &g_SharedMem,         // Data to pass to the function
            NULL,                 // Creation flags, keep NULL
            &dwCreditsID          // Identifier (we don't use this)
            );
```

Even though this listing shows only some of the code for creating the thread, the main part that interests us is the `CreateThread()` function. The `CreateThread()` function takes six parameters.

The first parameter sets the security attributes for the thread. Because you don't need to mess with the defaults provided, just set this to `NULL`. The second parameter is the stack size you are going to allocate for the thread. The default size varies depending on your environment but normally is set to around 1MB. If you need more than 1MB of static memory for your thread, go ahead and specify, in number of bytes, the requested stack size.

The third parameter is a pointer to the thread function. Because your thread must have a function, you need to create one for it to call. Just pass the address of your function here for this to work. The fourth parameter is a pointer to data you want to share or pass to the thread. In your case, you want to send a data structure, so you pass the address of it here. The fifth parameter tells the thread how to act when it is created. You only need the defaults here, so set it to `NULL`. The sixth and last parameter is a pointer to a `DWORD`, which will hold the thread's identifier. You just create a `DWORD` variable to hold this value and pass the address of it.

Wow, that's it! That's really all there is to creating a thread.

Closing a Thread Properly

Once you create a thread, you can close it using the `CloseHandle()` function. This function takes only one parameter, which is the handle to close. Here is the implementation of this call:

```
// Free the credits thread
CloseHandle( hCreditsThreadHandle );
```

Once you call this function, the handle to the thread is terminated. The only problem is that you must make sure the thread is finished before you close it. You wait for the thread to finish by using the `WaitForSingleObject()` function.

The `WaitForSingleObject()` function takes only two parameters. The first parameter expects the thread handle on which you want to wait. The second parameter is the amount of time you want to wait for it in milliseconds. Here is how you use it.

```
// Wait on the thread to finish (wait forever if necessary)
WaitForSingleObject( hCreditsThreadHandle, INFINITE );
```

You pass your thread handle for the first parameter and `INFINITE` for the second parameter. The `INFINITE` word, defined by Windows, means that you will wait forever for the thread to terminate.

TIP

If you do not wait for threads to terminate before closing them, your program might crash.

A complete program named CreateThread appears on this book's CD. This program demonstrates the thread concepts I have been discussing. Try it and see whether you can follow the use of threads and mutexes.

Review

Hey, guess what; we are finished with this chapter. Take a moment now to review some of the key concepts covered in this chapter, and then it's on to the next chapter!

- The Windows operating system supports multitasking. This allows your programs to perform multiple operations simultaneously.
- Including events in your programs can be useful for triggering code sections based on timed logic.
- Hungarian notation is used to help differentiate variable types.
- All Windows programs need a `WinMain()` function. This is the equivalent of the `main()` function seen in other operating systems.
- The `CreateWindow()` function is used to create the window for your Windows application. You can also use the `CreateWindowEx()` function for extended style windows.
- The window's message loop is used to process and translate all messages sent by the window or operating system.
- Multithreading can be used to increase your program's performance on computers with multiple processors.

Now that the theory is out of the way, it's time for some complete Windows programming. See you in the next chapter!

CHAPTER 7

Practical Windows Programming

Now that you know all about Windows and how the programs operate, it is time to actually write some useful code. I know you just got done reading a whole chapter on mostly theory, so I am going to keep things straight and to the point for this one. After reading this chapter, you should have a thorough understanding of how to implement the following items in Windows.

- Static text
- Buttons
- Sound
- List boxes
- Edit fields

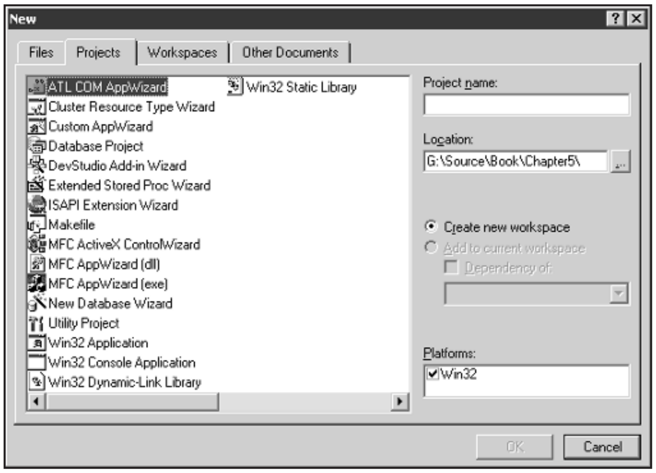
By the end of this chapter, you will have laid the groundwork to create complete Windows programs. Prepare to be assimilated.

Setting Up the Visual C++ Project

Before I dive into the code, I think it is best to cover how to create a Windows project in Visual C++. Okay, first things first: Launch the Visual C++ main program. It's the one with the multicolored Möbius strip for an icon. Well, maybe it's not a Möbius strip, but doesn't it look cool? All right, now that you have the program launched, choose File, New. You should see a dialog box that looks like the one in Figure 7.1.

Under the Projects tab, select Win32 Application. In the Project Name text box, type **HelloWorld**. You might want to change the default location at this time, but it is up to you. When you are finished, click OK. The next dialog box you'll see looks like the one in Figure 7.2.

At this prompt, you are asked whether you want an empty project, a simple Win32 application, or a typical "Hello World!" application. Even though you could make things easier on yourself and cheat using the Hello World option, pick the default empty project setting. Go ahead and click Finish. You'll see a dialog box that looks like the one in Figure 7.3.



Project name:

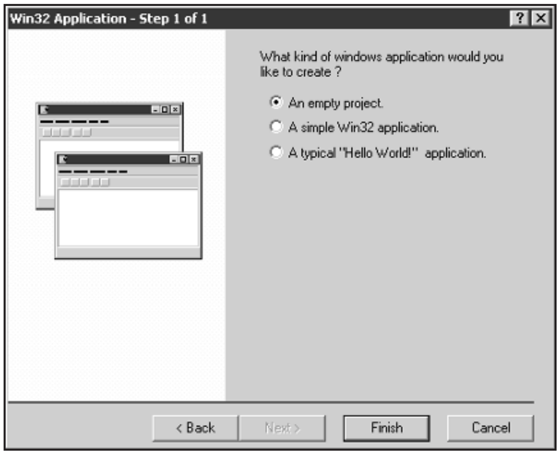
Location: G:\Source\Book\Chapter5\

[✓ Create new workspace] Add to current workspace

Platforms: [✓] Win32

Figure 7.1

Visual C++'s New dialog box



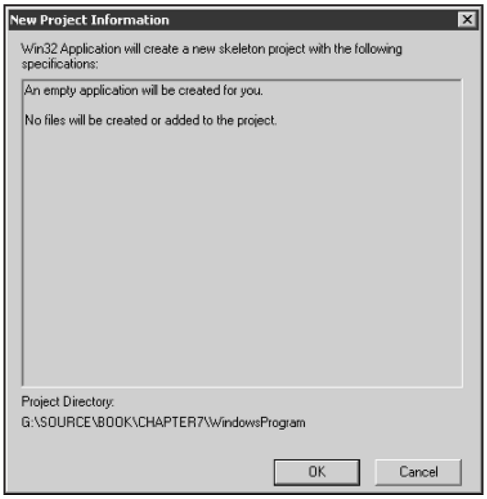
What kind of windows application would you like to create ?

[✓ An empty project.]
A simple Win32 application.
A typical "Hello World!" application.

Figure 7.2

Creating a Windows application, Step 1 of 1

This window tells you what files and such are created with your project. Because you chose to add nothing special, you have no new files included with the project. Go ahead and click OK. Yeah, yeah, yeah, I know this sounds like the steps in Chapter 4, "Writing Your First Network Programs," but you created a console application there, not a Win32 application. Once you have gone down the path of no return and clicked the OK button, you should see something like the display in Figure 7.4. Your display might vary based on your settings. For one thing, I have clicked on the FileView tab in Figure 7.4.


New Project Information

Win32 Application will create a new skeleton project with the following specifications:

An empty application will be created for you.

No files will be created or added to the project.

Project Directory:
G:\SOURCE\BOOK\CHAPTER7\WindowsProgram

Figure 7.3

The New Project Information dialog box

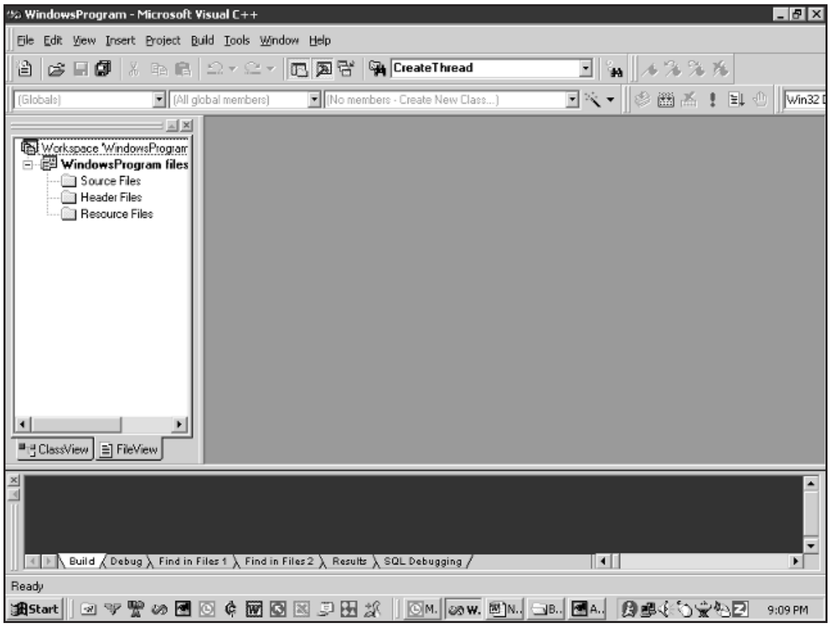


Figure 7.4

An empty Visual C++ Win32 project

Hey, that's a lot of figures for just the beginning, isn't it? Well, I'm trying to make up for Chapter 5, "Creating a Sockets Class Object." The rest of the steps you follow are just about the same as those in previous chapters. First, you need a main C++ file into which you can write code. Go up to the menu and select Project, Add to

Project, New. On the menu that appears, select C++ Source File and then type whatever you want to call it. For this first program, use the working name of HelloWorld.

Creating the Hello World Program

NOTE

On the CD that comes with this book, you will find source code for almost every example shown in the book. You will also find already-created projects for many of the programs. In this case, the Hello World program can be found under the HelloWorld project. If you are not using Visual C++, you should be able to create the projects yourself by including the appropriate source files.

Now that you have that pretty blank window to enter code into, type the following code.

```
// Standard window defines
#include <windows.h>
#include <stdio.h>

LPCTSTR         lpszApplicationName     = "HelloWorld";
LPCTSTR         lpszTitle               = "Hello World Program";

// Message Loop CallBack Function prototype ( REQUIRED FOR ALL WINDOWS PROGRAMS )
LRESULT CALLBACK fnMessageProcessor (HWND, UINT, WPARAM, LPARAM);

// Function to create the window and display it ( REQUIRED FOR ALL WINDOWS PROGRAMS )
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
 int nCmdShow)
{
    MSG             msg;
    HWND            hWnd;
    WNDCLASSEX      wndclass;

    // Set up window attributes
    wndclass.cbSize                     = sizeof(wndclass);
    wndclass.style                      = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc                = fnMessageProcessor;
    wndclass.cbClsExtra                 = 0;
    wndclass.cbWndExtra                 = 0;
    wndclass.hInstance                  = hInstance;
    wndclass.hIcon                      = LoadIcon( NULL, IDI_APPLICATION );
    wndclass.hCursor                    = LoadCursor( NULL, IDC_ARROW );
    wndclass.hbrBackground              = (HBRUSH)(COLOR_WINDOW);
    wndclass.lpszMenuName               = NULL;
    wndclass.lpszClassName              = lpszApplicationName;
    // Registered class name
    wndclass.hIconSm                    = LoadIcon( NULL, IDI_APPLICATION );

    if( RegisterClassEx( &wndclass ) == 0 ) {
        // Do error logic here
        exit(1);
    }
    // Create the window
    hWnd = CreateWindow(     lpszApplicationName,     lpszTitle,
                 WS_OVERLAPPEDWINDOW,
                 100,     100,     400,
                 340,     NULL,
                 NULL,
                 hInstance,
                 NULL );
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    // Process messages until the program is terminated
    while( TRUE ) {
        // Check for windows events & messages
        if (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE)) {
            if (!GetMessage(&msg, NULL, 0, 0))
                break;
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return(msg.wParam);
}
// Callback function to handle window messages
LRESULT CALLBACK fnMessageProcessor ( HWND hWnd, UINT iMsg, WPARAM wParam,
 LPARAM lParam )
{
    switch (iMsg) {
        case WM_COMMAND:
            break;
        case WM_CREATE:
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return(DefWindowProc(hWnd, iMsg, wParam, lParam));
    }
    return(0L);
}
```

If you don't want to type this in, you can always copy it off the CD, but where's the fun in that? Go ahead and press the F7 key to build the executable. Once you have done that, press Ctrl+F5 to execute it. If everything went okay, you should see the window in Figure 7.5 appear on your desktop. Don't be alarmed if your window colors appear a bit different; they can vary from system to system.

If the program doesn't run, check to make sure you had no compile errors and try again. If it still doesn't execute, compare your code to the working code on the CD. If the code on the CD doesn't work, buy a new computer. No, seriously, if it doesn't work from the CD, send me an e-mail at webmaster@lostlogic.com. Okay, now let's briefly walk through your new creation. Here are the steps you followed to get the window on the screen.

1. Create and define a window class of type `WNDCLASSEX`.
2. Register the class using the `RegisterClassEx()` function.
3. Create the window using the `CreateWindow()` function.
4. Display the window using the `ShowWindow()` function.
5. Update the window using the `UpdateWindow()` function.

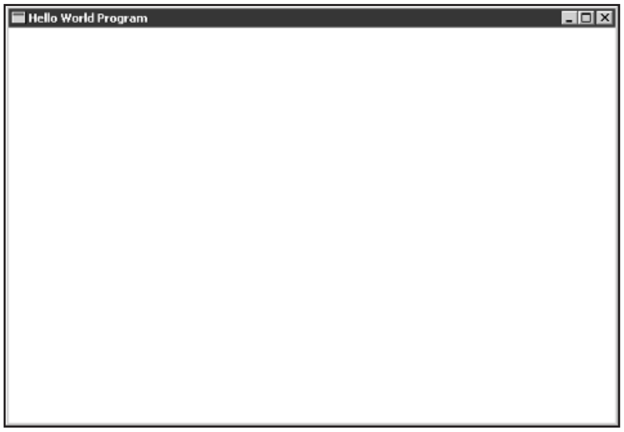


Figure 7.5

Output from the Hello World program

If any of these steps or functions look strange, you can go back and read over Chapter 6, "Introduction to Windows Programming," for a quick review. The next piece of code you run into, after creating and displaying the window, is the message loop.

```
while( TRUE ) {
    // Check for windows events & messages
    if (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE)) {
        if (!GetMessage(&msg, NULL, 0, 0))
            break;
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

The question is, do you want to take the blue pill and stay forever in this dream, or do you want to take the red pill and awake to a whole new reality? Tell you what, stress is a bad thing, so I think I will just take the blue pill and hang out where I am. Okay, now that we have satiated Morpheus, look a little more closely at the message loop.

The PeekMessage() Function

You use a new feature for this message loop, and the feature comes from the `PeekMessage()` function. This function lets you check for pending messages before trying to process them. If you don't use `PeekMessage()`, the program just sits there until the next Windows message comes along. This isn't very helpful if you want to do game updates on a consistent basis. Here is the prototype for the `PeekMessage()` function.

```
BOOL PeekMessage(     LPMSG lpMsg,
            HWND hWnd,
            UINT wMsgFilterMin,
            UINT wMsgFilterMax,
            UINT wRemoveMsg );
```

I don't normally like to spread out prototypes like this one, but it's just too darn long to fit on one line. Anyway, the first parameter is the pointer to a `MSG` data structure to receive the message information. The second parameter expects the handle to the window you want to check. Because you can have multiple windows operating within a single program, you have to specify which one you want to check.

The third parameter lets you set the starting range of message numbers to look at. You always set this to `0` for game programs. The fourth parameter lets you set the ending range of message numbers to look at. This is also set to `0` for our purposes.

The fifth parameter specifies what to do with the messages once they are peeked at. If you pass a `PM_NOREMOVE` to it, then the message stays in the queue. If you pass a `PM_REMOVE`, the message is removed. Because you just want to peek at the message and not actually process it yet, you use the `PM_NOREMOVE` flag.

The `PeekMessage()` function returns a `0` if no messages are present. If messages are present, then the function returns a non-zero number. So you check whether the return is true, and if so, you go ahead and call the `GetMessage()` function. Here are the usual steps you follow after you have found a message in the queue.

1. Call `GetMessage()` to load the `MSG` structure with the message information.
2. Call `TranslateMessage()` to translate virtual key messages into usable character messages.
3. Call `DispatchMessage()` to send the retrieved message to the message handler function.

That about covers it for the message loop. Now that you have the all-exciting Hello World program out of the way, add some text to the window.

Adding Static Text to a Window

If you look at almost any Windows program out there, you see some static text. You can place static text anywhere in a window that is not occupied by another object. You normally see static text used as headers for edit boxes and the like. Go ahead and load up the StaticText project from the CD-ROM to see the following code.

NOTE

If you are not using Visual C++, you can create the project yourself by creating a new Windows project and including the StaticText.cpp file in it. There should be no other dependencies. Consult your compiler's documentation for how to create an empty Windows project.

```
#include <windows.h>

#include <stdio.h>
// ID defines for resources
```

```
#define             IDC_hBU_StaticText       40001

// Create some global handles for our child window components
HWND                hBU_StaticText          = NULL;
LPCTSTR             lpszApplicationName      = "StaticText";
LPCTSTR             lpszTitle               = "Static Text Example";

// Message Loop CallBack Function prototype ( REQUIRED FOR ALL
 WINDOWS PROGRAMS )
LRESULT CALLBACK fnMessageProcessor (HWND, UINT, WPARAM, LPARAM);

// Function to create the window and display it ( REQUIRED FOR ALL
 WINDOWS PROGRAMS )
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    MSG             msg;
    HWND            hWnd;
    WNDCLASSEX      wndclass;

    // Set up window attributes
    wndclass.cbSize                 = sizeof(wndclass);
    wndclass.style                  = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc            = fnMessageProcessor;
    wndclass.cbClsExtra             = 0;
    wndclass.cbWndExtra             = 0;
    wndclass.hInstance              = hInstance;
    wndclass.hIcon                  = LoadIcon( NULL, IDI_APPLICATION );
    wndclass.hCursor                = LoadCursor( NULL, IDC_ARROW );
    wndclass.hbrBackground          = (HBRUSH)(COLOR_WINDOW);
    wndclass.lpszMenuName           = NULL;
    wndclass.lpszClassName          = lpszApplicationName;
    // Registered class name
    wndclass.hIconSm                = LoadIcon( NULL, IDI_APPLICATION );

    if( RegisterClassEx( &wndclass ) == 0 ) {
        // Do error logic here
        exit(1);
    }
    // Create the window
    hWnd = CreateWindow(     lpszApplicationName,     lpszTitle,
                  WS_OVERLAPPEDWINDOW,
                  100,
                  100,
                  400,
                  340,
                  NULL,
                  NULL,
                  hInstance,
                  NULL );
    // Create the static text
    hBU_StaticText = CreateWindow(
            "static","Static Text",
            WS_CHILD | SS_CENTER | WS_VISIBLE,
            120,
            130,
            135,
            28,
            hWnd,(HMENU)IDC_hBU_StaticText,hInstance,NULL);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    // Process messages until the program is terminated
    while( TRUE ) {
        // Check for windows events & messages
        if (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE)) {
            if (!GetMessage(&msg, NULL, 0, 0))
            break;
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return(msg.wParam);
}

// Callback function to handle window messages
LRESULT CALLBACK fnMessageProcessor ( HWND hWnd, UINT iMsg, WPARAM wParam,
 LPARAM lParam )
{
    switch (iMsg) {
        case WM_COMMAND:
            break;
        case WM_KEYDOWN:
            break;
        case WM_CREATE:
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return(DefWindowProc(hWnd, iMsg, wParam, lParam));
    }
    return(0L);
}
```

The code that has changed from the Hello World program to this one is in bold text. The rest of the code is identical to that in the Hello World program. Execute the program, and you should see some output similar to that shown in Figure 7.6.

Resource Identification

The first noticeably different code is the define at the beginning.

```
#define          IDC_hBU_StaticText     40001
```

This define is setting up an ID number for the static text resource object. What do I mean by a *static text resource object*? Well, every item in a window's area is really its own object. Each and every object in a window takes up a *resource*. So the objects in a window are called *resources*. Because the text you want to display operates on its

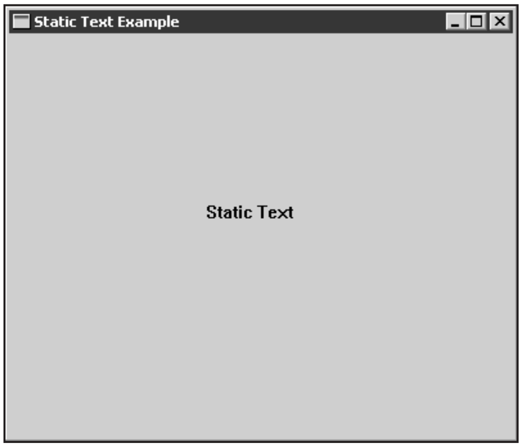


Figure 7.6

Output from the Static Text program

own, you must give Windows an *identity number* to associate with it. That way, when you want to move, remove, or alter the text, you have a reference to go by. I just chose an arbitrary number of 40001 for the static text object. You can choose anything you want here as long as it does not conflict with any other resource.

Creating a Child Window

The next piece of code that stands out is the creation of a variable to hold the handle to that static text object.

```
HWND                hBU_StaticText          = NULL;
```

Because window objects such as static text are really child windows, you create a handle of type HWND. Now that you have an ID for the object and a handle, it is time to create the child window. You do this using the all-too-familiar CreateWindow() function.

```
hBU_StaticText = CreateWindow(
                 "static",
                 "Static Text",
                 WS_CHILD | SS_CENTER | WS_VISIBLE,
                 120,
                 130,
                 135,
                 28,
                 hWnd,
                 (HMENU)IDC_hBU_StaticText,
                 hInstance,
                 NULL);
```

Now take a moment to recap what the CreateWindow() function takes as parameters. The parameters can be seen in Table 7.1.

You tell Windows what type of child object to create by using a predefined registered class name. Because you want a static text object, you tell Windows to create this object by passing the value of "static" for the first parameter. Table 7.2 contains a list of class types predefined by Windows.

CAUTION

When you create a child window using the CreateWindow() function, be sure to include WS_CHILD as one of the style flags. This lets Windows know that it is a child object.

Table 7.1 Parameters for the CreateWindow() Function

Parameter	Description
`LPCTSTR`	Registered class name for the window.
`LPCTSTR`	The name of the window; for child windows it can denote text associated with the object.
`DWORD`	Style flags that determine how the window looks.
`int`	Starting X, or horizontal, position.
`int`	Starting Y, or vertical, position.
`int`	Starting width of the window or child object.
`int`	Starting height of the window or child object.
`HWND`	Handle to the parent window.
`HMENU`	Handle to the menu or child window identifier.
`HANDLE`	Handle to the instance of the application.
`LPVOID`	Window creation data.

For the static text example, all you had to do was create a new child window of type `STATIC` and specify the main window as the parent. The X and Y positions determine where the text is displayed. The width and height parameters set up how much of the text can be displayed. Try moving the X and Y coordinates around to see the effects. Remember that two-dimensional screen coordinates are laid out like those shown in Figure 7.7.

When an object moves to the right of the screen, its X coordinate increases; when it moves to the left, the X coordinate decreases. When an object moves down the screen, its Y coordinate increases; when it moves up, the Y coordinate decreases. The position values you give to your static object are local to the parent window. *Local* coordinates base their origins on their parents. So if you set your static text at position 120,130, it really is offset 120 pixels to the right and 130 pixels down from where the parent window starts.

Table 7.2 Preregistered Windows Class Types

Class	Function
`STATIC`	A text field or box. Used mainly for labeling other objects. This class type accepts no input or output.
`BUTTON`	Used to display a rectangular depressible button. The button can optionally have text. Pressing the button can trigger a Windows message event.
`SCROLLBAR`	Creates a scrollbar with arrows at either end. Normally used in conjunction with a text box to scroll text. Clicking on the scrollbar creates a Windows message event.
`COMBOBOX`	Creates a list box and a selection area. Used mainly when creating a drop-down box containing a list of items. If the contents of the box are highlighted and the user types any text, the next item that starts with the letter typed is selected.
`EDIT`	Used to create text entry boxes.
`LISTBOX`	Creates a box containing rows of text. The text can be selected.
`RICHEDIT_CLASS`	Used to create a rich edit text window for the user.

Would you believe that I just now bought *Quake III Arena*? I purchased *Unreal Tournament* first because I liked its graphic style better. You know, I can only crawl around in the sewer-like hell that id Software creates for so long. Anyway, I have to say that *Quake III Arena* is one superb piece of work. Not only is the engine beautiful, but also the network code is phenomenal. Too bad they stuck with the sewer levels again, but that is my only complaint about it. Anyway, you'd better watch out; I'm getting better at playing it! Oh, yeah, if you are wondering how I wrote about *Quake III* in earlier chapters, it is because people normally write

TIP

When firing the rocket launcher at someone in *Quake III*, point at his feet. You miss much less often and still do a great deal of residual damage.

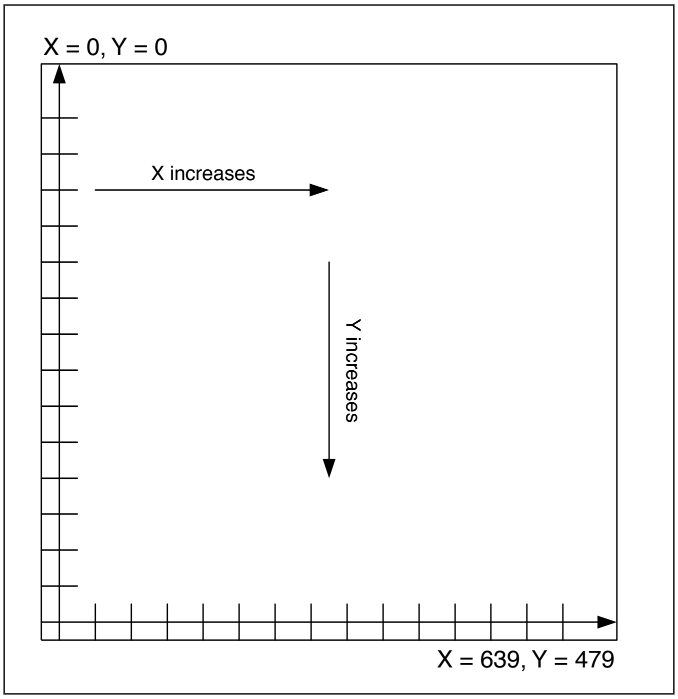


Figure 7.7

A two-dimensional coordinate system

chapters for books out of order. So this chapter was written before the introduction chapter. How's that for insider information? Oh well, back to the grind.

Having Fun with Buttons in Windows

Now that you have created static text in a window, let's do something more interactive, like create a button. On the CD, there is a project named Button. Load it up now to follow along. The following files are needed to compile the program.

- Button.cpp, the main source file
- Winmm.lib, the Windows multimedia file for the `PlaySound()` function

CAUTION

The `PlaySound()` function used in the Button project requires the inclusion of the winmm.lib library file in your project. You will get linking errors if you don't include it.

NOTE

If you are not using Visual C++, you can create the project yourself by creating a new Windows project and including the Button.cpp file and the winmm.lib library file. If you get linking errors about the `PlaySound()` function or if you do not have the winmm.lib library file, remove the reference to the `PlaySound()` function in the `fnMessageProcessor()` function in the code below.

The following code is a complete Windows program to create a depressible button on the screen.

```
#include <windows.h>
#include <stdio.h>
#define                IDC_hBU_Join        40001

// Create some global handles for our child window components
HWND                hBU_Join                = NULL;

LPCTSTR           lpszApplicationName       = "ButtonProg";
LPCTSTR           lpszTitle                 = "Button Example";

// Message Loop CallBack Function prototype ( REQUIRED FOR ALL
 WINDOWS PROGRAMS )
LRESULT CALLBACK fnMessageProcessor (HWND, UINT, WPARAM, LPARAM);
// Function to create the window and display it ( REQUIRED FOR ALL
 WINDOWS PROGRAMS )
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    MSG                msg;
    HWND               hWnd;
    WNDCLASSEX         wndclass;

    // Set up window attributes
    wndclass.cbSize                = sizeof(wndclass);
    wndclass.style                 = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc           = fnMessageProcessor;
    wndclass.cbClsExtra            = 0;
    wndclass.cbWndExtra            = 0;
    wndclass.hInstance             = hInstance;
    wndclass.hIcon                 = LoadIcon( NULL, IDI_APPLICATION );
    wndclass.hCursor               = LoadCursor( NULL, IDC_ARROW );
    wndclass.hbrBackground         = (HBRUSH)(COLOR_WINDOW);
    wndclass.lpszMenuName          = NULL;
    wndclass.lpszClassName         = lpszApplicationName;
    // Registered CLASS NAME
    wndclass.hIconSm               = LoadIcon( NULL, IDI_APPLICATION );
    if( RegisterClassEx( &wndclass ) == 0 ) {
        // Do error logic here
        exit(1);
    }
    // Create the window
    hWnd = CreateWindow(     lpszApplicationName, lpszTitle,
                  WS_OVERLAPPEDWINDOW,
                  100, 100, 400, 340, NULL, NULL, hInstance, NULL );
    // Create the JOIN button
    hBU_Join = CreateWindow(
        "BUTTON",
        "Join",
        WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
        285,
        280,
        100,
        28,
        hWnd,(HMENU)IDC_hBU_Join,hInstance,NULL);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    // Process messages until the program is terminated
    while( TRUE ) {
        // Check for windows events & messages
        if (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE)) {
             if (!GetMessage(&msg, NULL, 0, 0))
                 break;
             TranslateMessage(&msg);
             DispatchMessage(&msg);
        }
    }
    return(msg.wParam);
}

// Callback function to handle window messages
LRESULT CALLBACK fnMessageProcessor ( HWND hWnd, UINT iMsg,
 WPARAM wParam, LPARAM lParam )
{
    switch (iMsg)
    {
        case WM_COMMAND:
            // Check for child window messages
            switch(LOWORD(wParam)) {
                // Check whether the user clicked the button
                case IDC_hBU_Join:
                    PlaySound("join.wav",NULL,SND_FILENAME);
                break;
            }
            break;
        case WM_CREATE:
            break;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return(DefWindowProc(hWnd, iMsg, wParam, lParam));
    }
    return(0L);
}
```

The Button Class

For this program example, you changed the name of your object identifier and the name of your window handle. There's not much to that really; it was just changed to better reflect the functionality of the new object. The next thing you do is call the `CreateWindow()` function with the parameters needed for a button object. Instead of using `"static"` for the value of the first parameter, you now pass in `"button"`. This

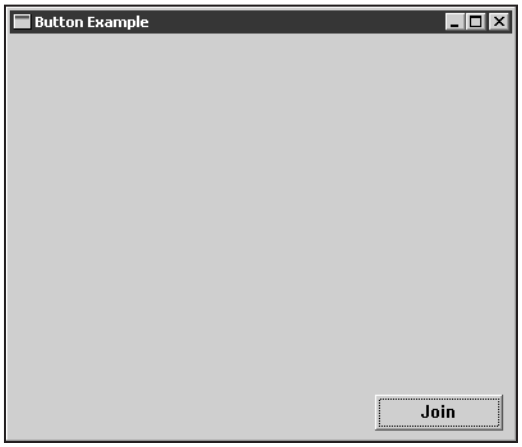


Figure 7.8

Output from the Button program

tells Windows to use the preregistered class type for a button control. Oh, yeah, the output of the program should look like that in Figure 7.8.

The next parameter, with a value of "Join", tells your program what to display on the button. The rest of the parameters tell you how large the button is and where to put it.

Now comes the interesting part. You have added a new logic branch to your message handler function.

```
case WM_COMMAND:
    // Check for child window messages
    switch(LOWORD(wParam)) {
        // Check whether the user clicked the button
        case IDC_hBU_Join:
        PlaySound("join.wav",NULL,SND_FILENAME);
        break;
    }
    break;
```

In the previous Windows programs, you did nothing under the WM_COMMAND switch statement. Now, you are checking the low-order bytes in the wParam field. By doing this, you can check for messages set off by the child window objects. The identifiers you have been using for the child objects represent the messages that get passed here. So when the Join button is pressed, a message with the parameter of IDC_hBU_Join is sent to the message handler. This is why you include a unique identifier in the call to CreateWindow(). The message parameter sent is the number used in your definition of the identifier. For instance, the code you used to identify `IDC_hBU_Join` is the following:

```
#define          IDC_hBU_Join          40001
```

When a message is passed to the message handler for the Join button, the number 40001 is actually used as the low-order value of the `wParam`. Given this logic, the switch statement could have looked like this:

```
case WM_COMMAND:
    // Check for child window messages
    switch(LOWORD(wParam)) {
        // Check whether the user clicked the button
        case 40001:     // Our identifier
        PlaySound("join.wav",NULL,SND_FILENAME);
        break;
    }
    break;
```

Notice the 40001 in place of the defined word. Not that you would want to code this way, but at least you now know why we do it.

The PlaySound() Function

I bet you thought I was going to skip over how to use the `PlaySound()` function. Right? Well, I don't want to leave you out in the cold, so here is the prototype.

```
BOOL PlaySound( LPCSTR pszSound, HMODULE hmod, DWORD fdwSound )
```

The function returns a value of `TRUE` if it succeeds or a value of `FALSE` if it fails. The first parameter expects the name of the sound to play. The flags you set in the third parameter determine from where the sound is loaded.

The second parameter tells the function the executable that contains the resource from which to load the file. Because you just load your sounds, leave this parameter at `NULL`.

The third and final parameter sets flags for playing the sound. Table 7.3 contains a list of some of the more useful flags.

You use the `PlaySound()` function to tell you that Windows has received your button press message. So when you press the Join button, you should hear a sound play. Try playing other wave files for when the button is pressed.

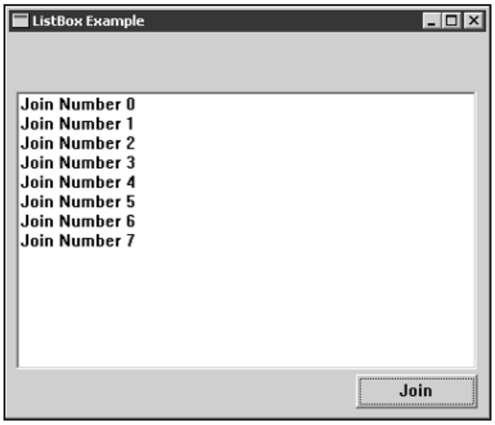
Well, that's about it for creating buttons within a Windows program. Now it's time to work on a list box.

Table 7.3 Flags Used in the PlaySound() Function

Flag	Purpose
`SND_FILENAME`	The sound is loaded from a file.
`SND_RESOURCE`	The sound is already loaded as a resource.
`SND_LOOP`	The sound is played continuously until a `NULL` is passed as the sound name in the `PlaySound()` function.
`SND_ASYNC`	The `PlaySound()` function does not wait for the sound to finish playing before continuing. Without this flag, the function waits until the sound is done.
`SND_NOSTOP`	If another sound is playing, it is not interrupted with the new sound call.

The ListBox Child Class Object

The `ListBox` class type lets you create a box within a Windows application to hold text. Each line of text is considered an additional entry into the list. For a change of pace, take a look at the output of the program before you see the source code (see Figure 7.9).



```
Join Number 0
Join Number 1
Join Number 2
Join Number 3
Join Number 4
Join Number 5
Join Number 6
Join Number 7
```

Figure 7.9

Output from the ListBox program

TIP

The project for this program can be found on the CD in the Chapter7 Zip file. Load it up now if you want to follow along.

CAUTION

The ListBox project uses the ListBox.cpp file and the winmm.lib library file to work. You will need to include both of these into your project if you decide to create it from scratch.

Now that you have seen the output, take a look at the source code for this little gem.

```
#include <windows.h>
#include <stdio.h>
#define          IDC_hBU_Join            40001
#define          IDC_hLB_Output          40002
// Create some global handles for our child window components
HWND             hBU_Join                = NULL;
HWND             hLB_Output              = NULL;
LPCTSTR          lpszApplicationName     = "ListBoxProg";
LPCTSTR          lpszTitle               = "ListBox Example";
// Message Loop CallBack Function prototype ( REQUIRED FOR ALL
 WINDOWS PROGRAMS )
LRESULT CALLBACK fnMessageProcessor (HWND, UINT, WPARAM, LPARAM);
// Function to display text into a listbox
void vShowText(HWND    hChildHandle, char *szText);
// Function to create the window and display it ( REQUIRED FOR ALL
 WINDOWS PROGRAMS )
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    MSG                 msg;
    HWND                hWnd;
    WNDCLASSEX          wndclass;

    // Set up window attributes
    wndclass.cbSize             = sizeof(wndclass);
    wndclass.style              = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc        = fnMessageProcessor;
    wndclass.cbClsExtra         = 0;
    wndclass.cbWndExtra         = 0;
    wndclass.hInstance          = hInstance;
    wndclass.hIcon              = LoadIcon( NULL, IDI_APPLICATION );
    wndclass.hCursor            = LoadCursor( NULL, IDC_ARROW );
    wndclass.hbrBackground      = (HBRUSH)(COLOR_WINDOW);
    wndclass.lpszMenuName       = NULL;
    wndclass.lpszClassName      = lpszApplicationName;
    // Registered class name
    wndclass.hIconSm            = LoadIcon( NULL, IDI_APPLICATION );

    if( RegisterClassEx( &wndclass ) == 0 ) {
        // Do error logic here
        exit(1);
    }
    // Create the window
    hWnd = CreateWindow(        lpszApplicationName, lpszTitle, WS_OVERLAPPEDWINDOW,
                  100, 100, 400, 340, NULL, NULL, hInstance, NULL );
    // hBU_Join
    hBU_Join = CreateWindow(
        "BUTTON","Join",        WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
        285, 280, 100, 28,
        hWnd,(HMENU)IDC_hBU_Join,hInstance,NULL);

    // hLB_Output
    hLB_Output = CreateWindowEx(
        WS_EX_CLIENTEDGE,
        "LISTBOX",
        NULL,
        WS_CHILD | WS_VISIBLE | LBS_NOTIFY | WS_VSCROLL | WS_BORDER,
        5,
        47,
        380,
        230,
        hWnd,(HMENU)IDC_hLB_Output,hInstance,NULL);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    // Process messages until the program is terminated
    while( TRUE ) {
        // Check for windows events & messages
        if (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE)) {
            if (!GetMessage(&msg, NULL, 0, 0))
                break;
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return(msg.wParam);
}

// Callback function to handle window messages
LRESULT CALLBACK fnMessageProcessor ( HWND hWnd, UINT iMsg,
 WPARAM wParam, LPARAM lParam )
{
    char     szText[256];
    static   count = 0;

    switch (iMsg) {
        case WM_COMMAND:
            // Check for child window messages
            switch(LOWORD(wParam)) {
                // Check whether the user clicked the button
                case IDC_hBU_Join:
                    sprintf(szText,"Join Number %d",count++);
                    PlaySound("join.wav",NULL,SND_FILENAME);
                    vShowText(hLB_Output,szText);
                    break;
                }
            break;
        case WM_CREATE:
            break;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return(DefWindowProc(hWnd, iMsg, wParam, lParam));
    }
    return(0L);
```

```
}

// Function to display text into the edit window
void vShowText(HWND    hChildHandle, char *szText)
{
    int Line;

    // Add string to the listbox
    SendMessage(hChildHandle,LB_ADDSTRING,0,(LPARAM) szText);

    // Determine number of items in listbox
    Line = SendMessage(hChildHandle,LB_GETCOUNT,0,0);

    // Flag last item as the selected item to scroll listbox down
    SendMessage(hChildHandle,LB_SETCURSEL,Line-1,0);

    // Unflag all items to eliminate negative highlight
    SendMessage(hChildHandle,LB_SETCURSEL,-1,0);
}
```

This program builds upon the button program you wrote earlier. To have a new child object, you had to create a unique identifier for it. First, you added a new define and then you added a new handle.

```
#define        IDC_hLB_Output        40002
HWND           hLB_Output            = NULL;
```

The notation of `hLB` denotes a ListBox handle. Previously, you used `hBU` to denote buttons. Anyway, now that you have the handles and such out of the way, you can add the creation routine to your main program. Here you should notice a difference from the previous child objects. To create the list box, make a call to the `CreateWindowEx()` function.

The CreateWindowEx() Function

The `CreateWindowEx()` function allows you to create windows with extended styles. Other than that, it is no different than the `CreateWindow()` function. The `CreateWindowEx()` function has an additional parameter at the beginning of the function call. This parameter expects flags denoting the extended style of the window. Here is the prototype for the function.

```
HWND CreateWindowEx(
    DWORD dwExStyle,
    LPCTSTR lpClassName,
    LPCTSTR lpWindowName,
     DWORD dwStyle,
     int x,
     int y,
     int nWidth,
     int nHeight,
     HWND hWndParent,
     HMENU hMenu,
     HINSTANCE hInstance,
     LPVOID lpParam );
```

If you compare the parameters to those in `CreateWindow()`, you'll see that this function does indeed have only one additional parameter. Table 7.4 shows some of the values for the `dwExStyle` parameter.

For your list box, you chose to use the `WS_EX_CLIENTEDGE` extended style. You use this so that your output window has a sunken 3D look to it. That's about it as far as extended windows are concerned.

Table 7.4 Extended Window Styles

Flag	Meaning
`WS_EX_CLIENTEDGE`	The window has a sunken edge. This gives it a 3D look.
`WS_EX_TRANSPARENT`	Child windows underneath the window are visible.
`WS_EX_TOPMOST`	The window always stays on top, even when it is not active.
`WS_EX_CONTROLPARENT`	The user can use the Tab key to switch to other child windows.
`WS_EX_DLGMODALFRAME`	The window has a double border; it can also have a name if you use `WS_CAPTION`.

When running the program, you might notice that when you press the Join button, text appears in the list box. If you haven't noticed this, go ahead and run the program to see it in action. Okay, so how did you output text to the listbox? Well, here's your answer: By using the `SendMessage()` function.

The SendMessage() Function

The `SendMessage()` function sends a message directly to a window. When you call this function, it then calls the target window's message procedure. The `SendMessage()` function does not return until the sent message is processed. Here is the prototype for the function.

```
LRESULT SendMessage(
     HWND hWnd,          // Destination window handle
     UINT Msg,           // Message to send
     WPARAM wParam,      // Parameter one
     LPARAM lParam       // Parameter two
);
```

The first parameter expects a handle to the window to which you want to send the message. The handle can also point to a child window; it does not have to be the main window. The second parameter holds the message to send to the target window. The third and fourth parameters allow you to specify additional information about the message.

In the `vShowText()` function, you use quite a few different variations of the `SendMessage()` function. Here is the first call you make to it.

```
SendMessage(hChildHandle,LB_ADDSTRING,0,(LPARAM) szText);
```

The first thing you pass to the function is the handle to the list box. You do this because you want to send text to it. The second parameter you pass, `LB_ADDSTRING`, is a message type that tells the list box you want to add some text to it.

List Box Messages

The `LB_ADDSTRING` message is associated with list boxes. You can generally tell what a message is associated with by its prefix. In this case, the prefix is `LB`—hence, a list box message.

This type of message expects two additional parameters. The first parameter must be set to `0`. The second parameter expects the text you want to send to the list box. For your purposes, pass a `0` and then the text for the extended parameters.

Now that you have added text to your list box, you need to do some other operations to ensure it appears the way you want. By default, when you add text to a list box using the LB_ADDSTRING message, it puts the text at the bottom of the list box but does not scroll the window. This causes problems because you want to see the last item of text added to the list box without having to manually scroll the control. If you want to see the effects of the default behavior, comment out the rest of the code in the vShowText() function after the first call to SendMessage(). You can also see an illustration of this effect in Figure 7.10.

Okay, now that you see what can happen if you don't take care of this little scrolling problem, take a look at how to fix it. To display the last item, you have to do a little trickery. The first thing you have to do is figure out how many lines of text are in the list box. This is accomplished with another call to the SendMessage() function.

```
Line = SendMessage( hChildHandle, LB_GETCOUNT, 0, 0 );
```

Well, would you look at that? You can use messages to retrieve information as well as send it. In this case, you use the LB_GETCOUNT message to get the number of text lines in the list box. You need no special parameters, just a call to SendMessage() with the message type of LB_GETCOUNT. Now that you know the number of lines in the list box, you want to select the last line. By selecting the last line, you force the list box to scroll down to the bottom. Here is the call you make to select the last item in the list box.

```
SendMessage( hChildHandle, LB_SETCURSEL, Line-1, 0 );
```

Basically, all you do is use the LB_SETCURSEL message to set the currently selected line item in the list box. The first parameter you pass to the message tells it which

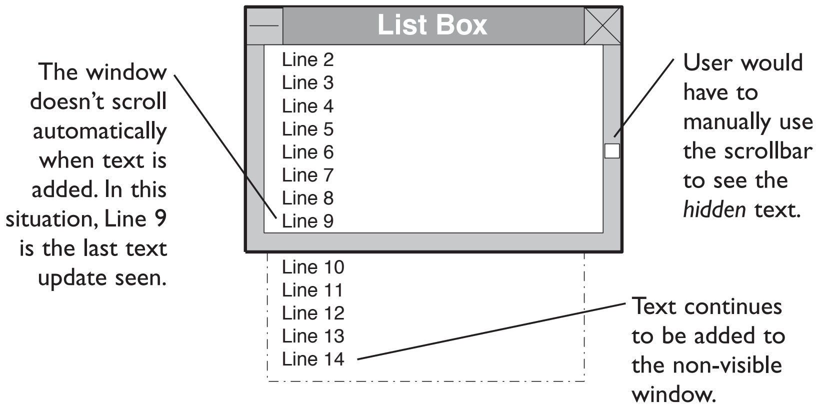

List Box
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
Line 13
Line 14
The window doesn't scroll automatically when text is added. In this situation, Line 9 is the last text update seen.
User would have to manually use the scrollbar to see the *hidden* text.
Text continues to be added to the non-visible window.

Figure 7.10

Effects of the LB_ADDSTRING *message*

item to select. By activating an item in the list box, you force the control to scroll to it. Now that the item is visible, you want to deselect it; otherwise, the item has a big nasty highlight on it. To deselect the text, you send another `LB_SETCURSEL` message, but with a `-1` as the first parameter. By sending a `-1` to the message, you are telling it to deselect all items. Play around with the parameters of the calls I've just covered to get a feel for them.

Accepting Text Input with Edit Fields

So far you have an application that lets you display some text by pressing a button. Don't you think it would be nice if you could enter some information into it? I do, so take a look at edit fields. *Edit fields* let you accept text from the user and then use it in your application. Because Windows handles all of the functionality within the edit field, all you have to do is create an edit field in your window and then pull information from it. I have updated our little application with edit field functionality. Here is the source code for the updated program.

```
#include <windows.h>
#include <stdio.h>

#define         IDC_hBU_Join          40001
#define         IDC_hLB_Output        40002
#define         IDC_hEB_Name          40003

// Create some global handles for our child window components
HWND          hBU_Join             = NULL;
HWND          hLB_Output           = NULL;
HWND          hEB_Name             = NULL;

LPCTSTR     lpszApplicationName     = "EditFieldProg";
LPCTSTR     lpszTitle               = "EditField Example";

// Message Loop CallBack Function prototype ( REQUIRED FOR ALL
 WINDOWS PROGRAMS )
LRESULT CALLBACK fnMessageProcessor (HWND, UINT, WPARAM, LPARAM);
void vShowText(HWND    hChildHandle, char *szText);
```

```
// Function to create the window and display it ( REQUIRED FOR ALL
 WINDOWS PROGRAMS )
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nCmdShow)

{
     MSG                msg;
     HWND               hWnd;
     WNDCLASSEX         wndclass;

    // Set up window attributes
    wndclass.cbSize                = sizeof(wndclass);
    wndclass.style                 = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc           = fnMessageProcessor;
    wndclass.cbClsExtra            = 0;
    wndclass.cbWndExtra            = 0;
    wndclass.hInstance             = hInstance;
    wndclass.hIcon                 = LoadIcon( NULL, IDI_APPLICATION );
    wndclass.hCursor               = LoadCursor( NULL, IDC_ARROW );
    wndclass.hbrBackground         = (HBRUSH)(COLOR_WINDOW);
    wndclass.lpszMenuName          = NULL;
    wndclass.lpszClassName         = lpszApplicationName;
    // Registered class name
    wndclass.hIconSm               = LoadIcon( NULL, IDI_APPLICATION );

    if( RegisterClassEx( &wndclass ) == 0 ) {
        // Do error logic here
        exit(1);
    }
    // Create the window
    hWnd = CreateWindow(      lpszApplicationName, lpszTitle, WS_OVERLAPPEDWINDOW,
                100, 100, 400, 340, NULL, NULL, hInstance, NULL );
    // hBU_Join
    hBU_Join = CreateWindow(
        "BUTTON","Join",      WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
        285, 280, 100, 28,
        hWnd,(HMENU)IDC_hBU_Join,hInstance,NULL);

    // hLB_Output
    hLB_Output = CreateWindowEx(
        WS_EX_CLIENTEDGE,"LISTBOX",     NULL,
        WS_CHILD | WS_VISIBLE | LBS_NOTIFY | WS_VSCROLL | WS_BORDER,
        5,47,380,230,hWnd,(HMENU)IDC_hLB_Output,hInstance,NULL);

    // Name
    hEB_Name = CreateWindowEx(
        WS_EX_CLIENTEDGE,
        "EDIT","Your Name",
        WS_CHILD | WS_VISIBLE | WS_BORDER | ES_LEFT,
        250,
        20,
        135,
        28,
        hWnd,(HMENU)IDC_hEB_Name,hInstance,NULL);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    // Process messages until the program is terminated
    while( TRUE ) {
        // Check for windows events & messages
        if (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE)) {
            if (!GetMessage(&msg, NULL, 0, 0))
                break;
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return(msg.wParam);
}

// Callback function to handle window messages
LRESULT CALLBACK fnMessageProcessor ( HWND hWnd, UINT iMsg,
 WPARAM wParam, LPARAM lParam )
{
    char        szText[256];
    static      count = 0;
    char        szName[256];
    switch (iMsg) {
        case WM_COMMAND:
            // Check for child window messages
            switch(LOWORD(wParam)) {
            // Check if the user clicked the button
                case IDC_hBU_Join:
                    // Get the name from the name edit window
                    GetWindowText(hEB_Name,szName,256);
                    sprintf(szText,"<%s> Join Number %d",szName,count++);
                    PlaySound("join.wav",NULL,SND_FILENAME|SND_ASYNC);
                    vShowText(hLB_Output,szText);
                    break;
            }
            break;
        case WM_CREATE:
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return(DefWindowProc(hWnd, iMsg, wParam, lParam));
    }
    return(0L);
}

// Function to display text into the listbox
void vShowText(HWND    hChildHandle, char *szText)
{
    int Line;

    // Add string to the listbox
    SendMessage(hChildHandle,LB_ADDSTRING,0,(LPARAM)szText);
    // Determine number of items in listbox
    Line = SendMessage(hChildHandle,LB_GETCOUNT,0,0);
    // Flag last item as the selected item to scroll listbox down
    SendMessage(hChildHandle,LB_SETCURSEL,Line-1,0);
    // Unflag all items to eliminate negative highlight
    SendMessage(hChildHandle,LB_SETCURSEL,-1,0);
}
```

Please note the changes that appear in bold text. They are the only relevant code changes you need to notice. I hope most of the changes look familiar to you by now. First, you assign a new unique identifier with a define statement. Then, you create a new window handle for your edit field. After that is out of the way, you make a call to `CreateWindowEx()` to create the edit field. If you haven't already, go ahead and launch the program. The output should look like that shown in Figure 7.11.

Okay, now for the good stuff. Change the name in the edit field, and then click the Join button. What you have done is pull the information entered into the edit field and then put it into the list box. The question should be, how do you get data from an edit field into your application? The answer is the `GetWindowText()` function.

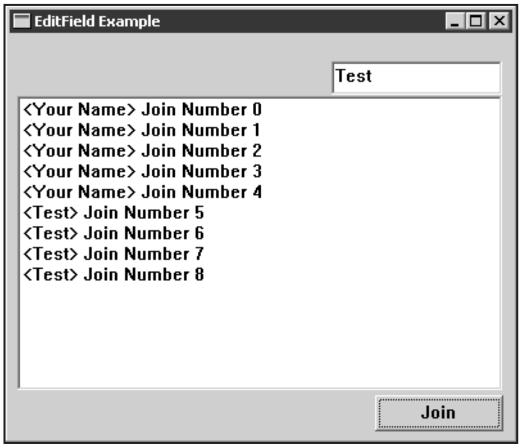
The GetWindowText() Function

The `GetWindowText()` function copies the text from a window control into a specified buffer. You can also use the function to copy the text from a window's title bar, but I don't use that method here. Here is the prototype for the function.

```
int GetWindowText( HWND hWnd, LPTSTR lpString, int nMaxCount )
```

The function returns the number of characters retrieved if it was successful. If something went wrong, or there was no text to retrieve, then a `0` is returned.

The first parameter expects the handle of the window or child window from which you want to retrieve text. In this application, pass the handle to the edit field here.



```
<Your Name> Join Number 0
<Your Name> Join Number 1
<Your Name> Join Number 2
<Your Name> Join Number 3
<Your Name> Join Number 4
<Test> Join Number 5
<Test> Join Number 6
<Test> Join Number 7
<Test> Join Number 8
```

Figure 7.11

Output from the EditField program

The second parameter expects a character buffer to hold the text to retrieve. The final parameter tells the function the maximum number of characters to retrieve. Make sure that you have enough space in your buffer to match this number.

As a recap, here is how you called the `GetWindowText()` function.

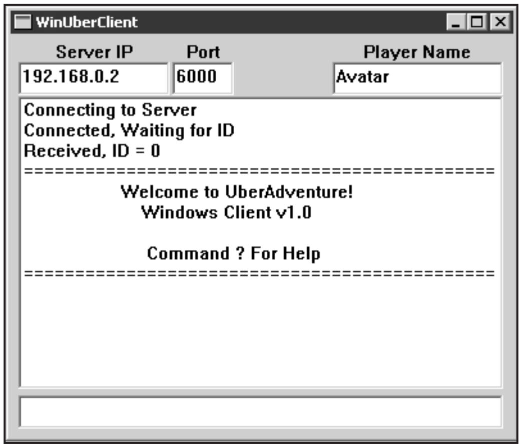
```
GetWindowText(hEB_Name,szName,256);
```

Not much to it really—just a few easy parameters. Remember that you must call this function each time you want to use text from a window control object. The contents of the character buffer are not automatically updated whenever the user changes the edit field.

The WinUberClient Program

As a little bonus, I went ahead and converted the client program for UberAdventure to a Windows application. The new program operates in the same way as the console client, but with a graphical interface. The source code is too long to list here, but I want you to load the project and take a look at the source code. Figure 7.12 shows the output of the program.

You need to launch the UberServer program from Chapter 4 to have something to which to connect. Once you've done that, launch the new client application and try it out.



Server IP: 192.168.0.2
Port: 6000
Player Name: Avatar

```
Connecting to Server
Connected, Waiting for ID
Received, ID = 0
==========================================
          Welcome to UberAdventure!
            Windows Client v1.0

             Command ? For Help
==========================================
```

Figure 7.12

Output from the WinUberClient program

Review

In this chapter, you learned how to create more interesting windows, complete with various gizmos. Here are some key points to remember.

- All objects within the main window are called child windows.
- Static text child windows are useful for displaying text that cannot be edited in a window.
- Child windows require a `HWND` handle and a unique identifier to properly use them.
- List boxes are useful for displaying lists of text in an output window.
- Button child windows are useful for providing an interactive interface for the user.
- Edit fields are great for retrieving input from the user.

Guess what? It's that time again—time to close another chapter. Even though I didn't cover some Windows programming issues, such as GDI programming, you now have the knowledge you need to write the complete games covered in this book. Well, I'm off to frag some people in *Quake III*; see you on the flip side!

Part Three

It's All About DirectX, Baby!

CHAPTER 8

What the Heck Is DirectPlay Anyway?

Welcome to Part Three of this little adventure. I'm glad you've made it this far because some of the best stuff is right around the corner. Oh, and by the way, that shadowy figure around the corner happens to be Microsoft's DirectPlay API. Because I have covered everything from network theory to Windows programming, I figure it's time to jump headfirst into the world of DirectX programming. I'll discuss the following topics in this chapter.

- DirectPlay
- DirectPlay addresses
- The DirectPlay message handler
- Peer-to-peer sessions

For the first assault, I am going to cover DirectPlay, the networking aspect of DirectX. That's what you're really here for anyway, right? As you would to build any good foundation, I am going to start with the theory and structure behind DPlay. Once I build a good foundation of knowledge, I'll put it to use. But that's for the next chapter. Kirk here; two to beam up.

DirectPlay, the Equalizer

If you are old enough, you might remember the campy TV series, *The Equalizer*. If you do, don't worry; DirectPlay has nothing to do with it! Anyway, here goes; it's time for the dry information part.

DirectPlay is the multiplayer component of Microsoft's DirectX SDK. Although other DirectX components, such as DirectDraw and Direct3D, get most of the attention, DirectPlay is the most important for your purposes. The main problem, before now, has been that there is little or no documentation for the API. In fact, the CD containing the DirectX SDK doesn't even have a Word file for DirectPlay!

The main idea behind DirectPlay is to hide all of the low-level communication worries from the developer. DirectPlay does this by abstracting the communication methods. Ooh, abstracting—big word, eh? When a program utilizes DirectPlay for its networking infrastructure, the application does not have to worry about the method used. For instance, your program can run using TCP/IP,

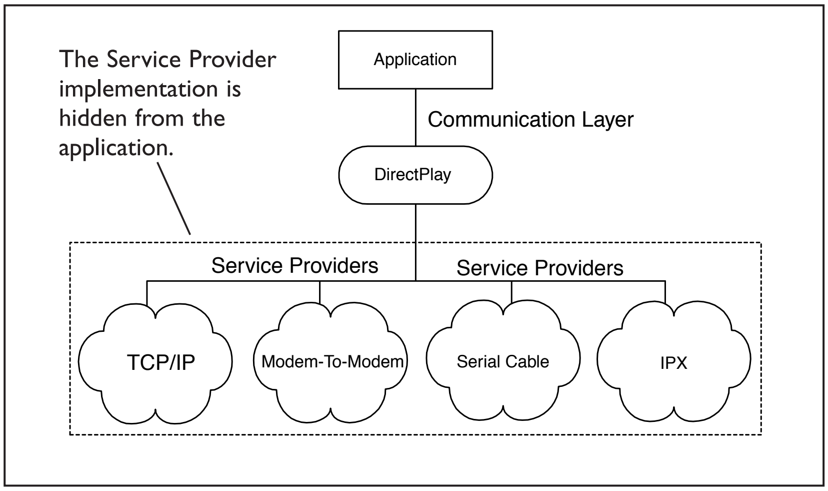


Figure 8.1

DirectPlay architecture

modem-to-modem, or serial cable, and you have to write only one code-set. To see this architecture in action, refer to Figure 8.1.

From the diagram, you can see that the application layer is removed from the communication layer. The theory behind DPlay is that you can add future communication methods without making many code changes.

Service Providers

The communication methods available in DPlay, such as TCP/IP, are commonly referred to as *service providers*. Even though you do not have to write entirely new code-sets to take advantage of different service providers, you do have to check for their capabilities. What this means is that different providers use different functionalities. For instance, TCP/IP can use guaranteed packets at this time, but IPX cannot. You have to keep these rules in mind when developing your application.

Enumeration of Service Providers

How do you know which service providers are available? You find out what is available by enumerating them. When you enumerate service providers with DPlay, you cycle through all of the available ones and create a list. The application can then check this list for whichever provider the user, developer, or player wants to use. If you want to keep things simple, you can attempt to select a certain service provider and check the return code. If the call returns without error, then you can use the service provider.

For most of the examples I provide, I try to set TCP/IP as service provider. If it fails, then I exit out of the program. I do this because TCP/IP is by far the most popular protocol for developing games, and it is also well suited for the Internet. Anyway, on with the show.

Game Sessions

Before I get too ahead of myself, I want to stop and talk about game sessions. A *game session* is an instance of a multiplayer game running simultaneously on two or more computers. A real-world example of this is when two people get together and play *Quake III* against one another. Each of the players in this scenario is playing in the same game session. You can also have two or more other players playing the same game but in different sessions.

NOTE

Only when people are playing together are they in the same game session.

To keep from being confused, take a look at Figure 8.2. It illustrates the session theory. You can see multiple sessions with multiple players in each one. Even though some of the sessions are running the same software, they are not running the same session.

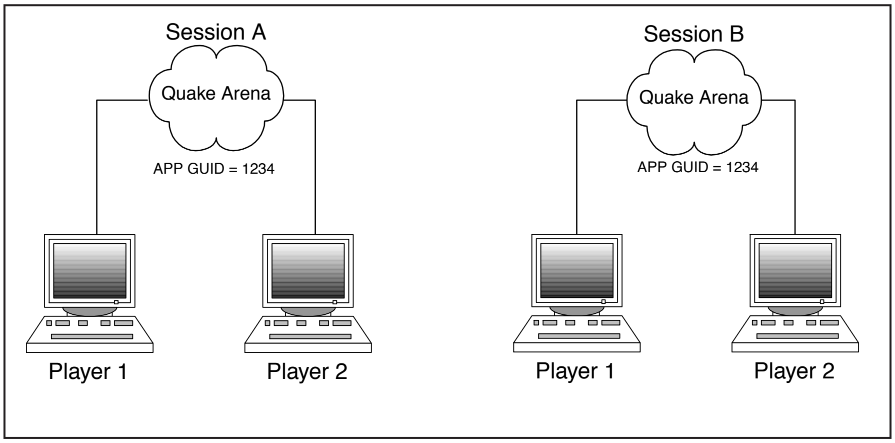


Figure 8.2

DirectPlay game sessions

Types of Game Sessions

The two basic types of game sessions available are peer-to-peer and client/server. I have already covered these two types of sessions, but I think they warrant a re-cap in relation to DPlay. One difference with DPlay is that you work with entirely different objects depending on which type of game session you utilize. This requirement makes it very important to plan which type of game session your game will use.

Peer-to-Peer Game Sessions

Peer-to-peer games in DirectPlay operate just like most other peer-to-peer systems. Basically, players in a peer-to-peer session are responsible for communicating their actions to all other players. Check out Figure 8.3 to see the architecture of six players participating in this type of session.

One thing obvious in the diagram is the high amount of network traffic. With each new player added to the session, the amount of traffic increases dramatically. This is the main reason why most games don't use the peer-to-peer method anymore.

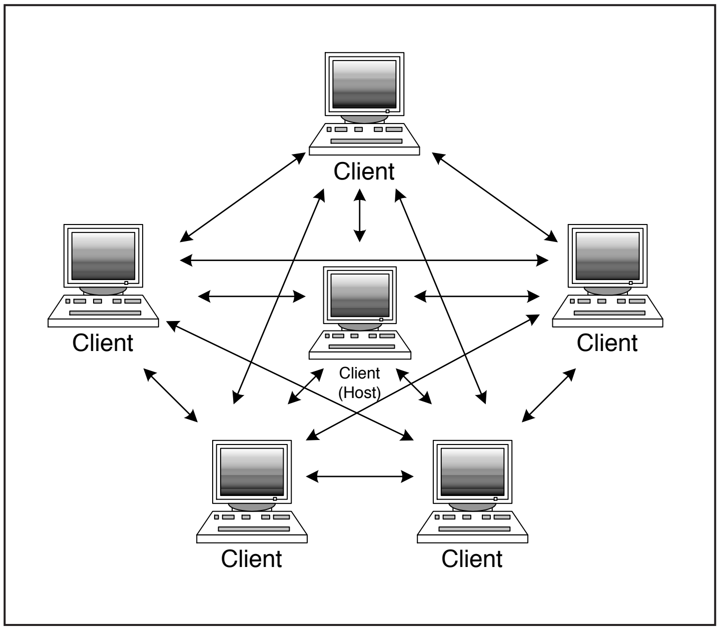


Figure 8.3

Peer-to-peer game architecture

The Peer Communication Object

Before you can start using peer-to-peer sessions, you must first create a peer communication object of type `CLSID_DirectPlay8Peer`. This is a COM object that is initialized with the `CoCreateInstance()` function call. The following code snippet shows you how to create just such an object.

```
// Create an IDirectPlay8Peer Object
IDirectPlay8Peer          *g_pDP;
CoCreateInstance(       CLSID_DirectPlay8Peer,
                                   NULL,
                                   CLSCTX_INPROC_SERVER,
                                   IID_IDirectPlay8Peer,
                                   (LPVOID*) &g_pDP );
```

You can see that the code uses an `IDirectPlay8Peer` interface. This is just one interface of many supported by DirectPlay, but it is the main interface for peer-to-peer communication.

Peer Hosting

Even though peer-to-peer games are responsible for communicating with one another, they still use a server of sorts. The server, called a *host,* is responsible for accepting new players to the game. Hosts are required because they provide the central place to which to connect.

TIP

Make sure you call the `CoInitialize()` function before using COM objects. It initializes the COM system, and it is a prerequisite to using all COM libraries.

The host can also manage some other aspects of game play, but the main point of hosting is to have a central location of connection. For a real-world example, you can look to Microsoft's Age of Empires series. In those games, you can pick a host to which to connect. Once everyone has joined the host's game and the game is started, everyone is responsible for communication.

Host Migration Unlike birds, most multiplayer games go south when things get bad and don't come back. In many games if the host leaves, the game is over. However, some games allow for host migration. What this means is that if the host quits the game, another player is automatically chosen to take over hosting. This process is called *host migration.* The cool part is that DirectPlay can take care of it for you automatically. All your programs have to do is check for a migrate-host message and continue accordingly. To see this concept illustrated, take a look at Figure 8.4.

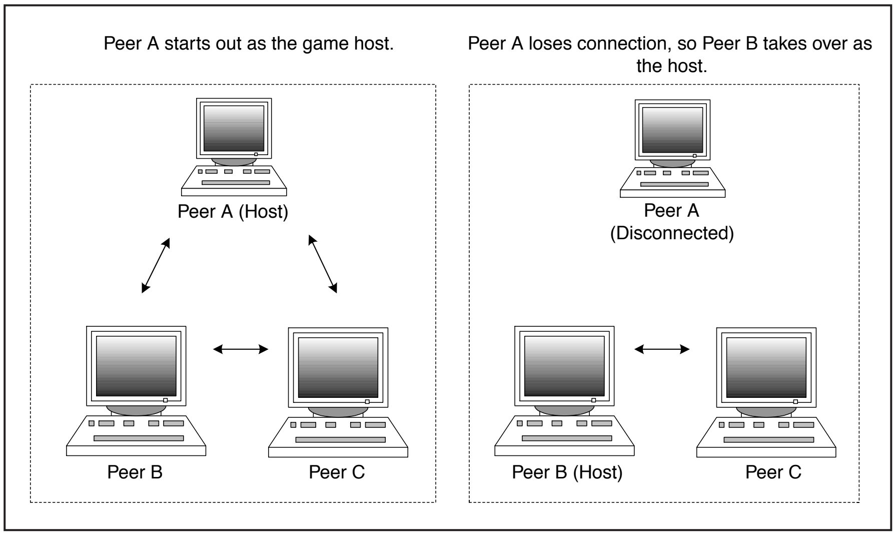

Peer A starts out as the game host.
Peer A (Host)
Peer B
Peer C
Peer A loses connection, so Peer B takes over as the host.
Peer A (Disconnected)
Peer B (Host)
Peer C

Figure 8.4
Host migration

You don't have to use host migration if you don't want to. But if you do want to use it, all you have to do is set a flag when creating the application description.

Host Responsibilities In addition to accepting new players to the game, the host is also responsible for a few other things, including

- Keeping track of who is in the game
- Informing everyone when a new player joins the game
- Informing everyone when a player leaves the game
- Accepting or denying a new player to the game

Luckily, all you have to deal with here are a few management items.

Connecting to a Peer Game

Unless you are the host, you have to establish a connection to participate in a peer-to-peer game. Not much is required other than specifying the host to which you want to connect. If you're using TCP/IP, then you specify an IP address and a port number to which to connect.

Client/Server Sessions

Like peer games, client/server games work just like any other architecture. Basically, a server is responsible for all game communication. Take a look at Figure 8.5 to see the architecture of four players participating in this type of session.

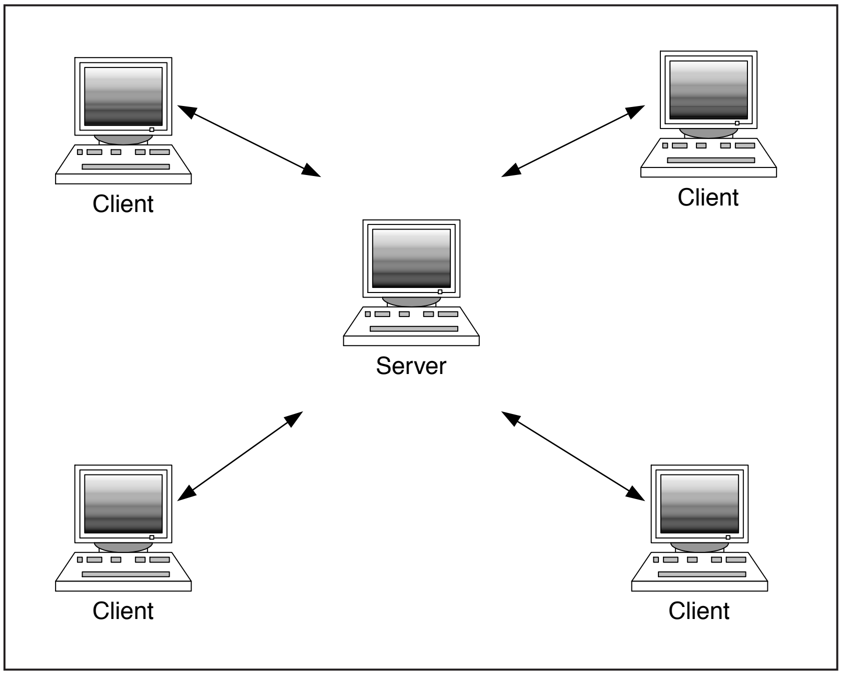


Figure 8.5

Client/server game architecture

As you can see in the image, the clients talk to the server but not to the other clients. This is because the server manages everything. The best advantage to this model is scalability. With client/server applications, you can have literally thousands of players online at the same time. This requires a very powerful server, but at least it is possible.

These types of games are actually simpler to operate in many regards. Because the server is responsible for everything, all the clients have to worry about are themselves.

The Server Communication Object

Because there isn't much of a game without a server, you must have a server game object somewhere, so you create a COM object of type `CLSID_DirectPlay8Server`. The interface that accompanies it is the `IID_IDirectPlay8Server` interface. You don't really need to worry about these objects now; just keep them in mind for when I show you how to use them later.

Server Hosting

In client/server games, the server is the game host. In many games, the server runs as a stand-alone console application. An example of this is a *Tribes* or *Quake* server.

Because the server program has no graphics to deal with, it can concentrate on network code. This keeps complexity down and speed up. Another benefit is that you can run the server on a graphically-challenged machine without problems. Many people don't have multiple computers lying around, so they just launch the server application and then launch a client application on the same machine to connect to it. This lets people who are hosting the game play it at the same time. Take a look at Figure 8.6 to see this concept illustrated.

When you host a game, you set up a port to which players can connect. If you are behind a firewall, you need to open up some ports for these player connections. Once you have decided which port to host

TIP

One thing to note in the client/server world is that no host migration is available. When the host goes away, the game is over.

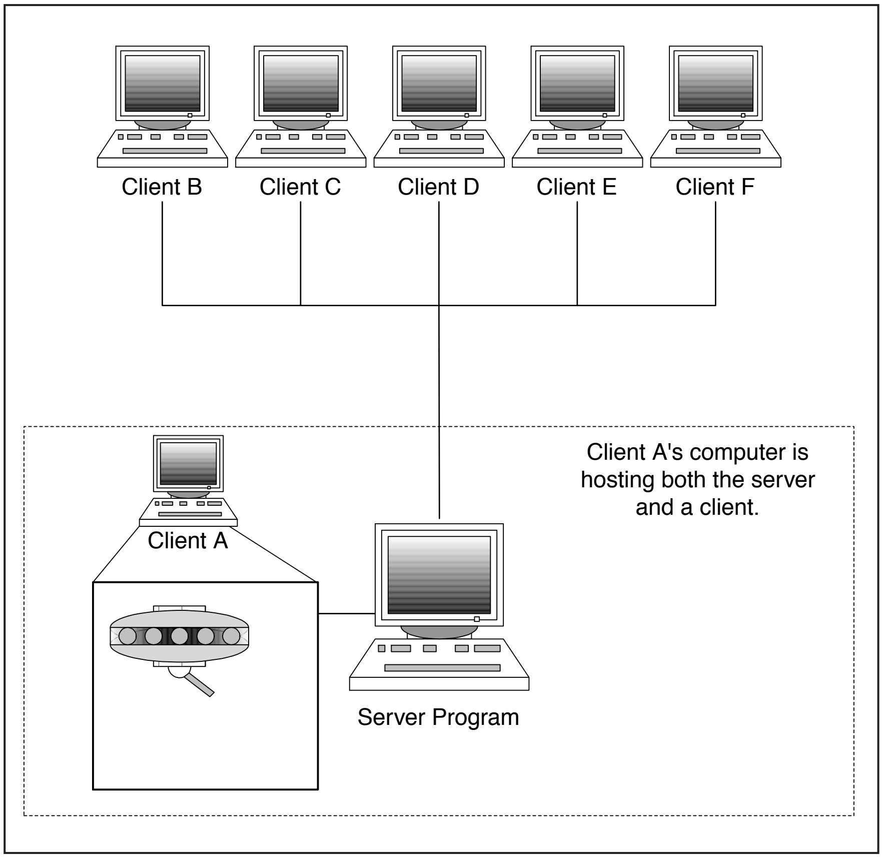


Figure 8.6

Separate server and client programs on the same machine on, all you have to do is use the host function provided by DPlay. After that is done, the server program can sit back with a cold one and wait for clients to connect.

The Clients

Now that you have a server out there, you need clients to connect to it. Clients in a client/server environment work mainly as peers do in a peer-to-peer environment. Basically, you locate a server to connect to and then establish the connection. The main difference is that clients in the client/server world only have to worry about sending packets to the server. Because clients can't send or receive packets from other players, they only have to listen for packets from the server.

TIP

Every DirectPlay address must be unique for multiplayer games to work properly.

DirectPlay Addresses

The first main concept that is unique to DirectPlay is addresses. To establish a connection with another computer, you must have an address to which to connect. With socket connections, you use the URL and port number. With DPlay, you must create a special address that can hold all sorts of information. Because DPlay can do more than TCP/IP, the address can hold such things as baud rate for modem connections or the IP address for Internet connections.

The Address Object

To create an address, you must use the `IDirectPlay8Address` interface. You do this by creating a COM object of type `CLSID_DirectPlayAddress`. Here is a brief example of creating a DPlay address object.

```
hReturn = CoCreateInstance( CLSID_DirectPlay8Address,
                                          NULL, CLSCTX_INPROC_SERVER,
                                          IID_IDirectPlay8Address,
                                          (LPVOID*) &g_pHostAddress );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Failed to Create Host Address()",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
```

You will be creating two types of address objects: one for the computer you are running the application from and one for the computer to which you are connecting. The address object that represents your machine is called the *device address*. The address object that represents the machine you are connecting to is called the *host address*.

Setting the Address Object's Service Provider

Although you have created the address object using the `CoCreateInstance()` function, the object is still useless. So far all you have done is create an empty address. To prepare the address for the type of data it needs to hold, you must set which service provider to use.

```
// Set the host address to TCP/IP
if( FAILED( hReturn = g_pHostAddress->SetSP( &CLSID_DP8SP_TCPIP ) ) ) {
    MessageBox( hWindow, "Failed to SetSP for Host Address",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
```

The code here sets up the host address to be of type `CLSID_DP8SP_TCPIP`. This type sets up the address to use TCP/IP. By specifying the service provider, you are setting the service provider *GUID* for the address object.

TCP/IP is not the only service provider available; take a look at Table 8.1 to see some more available providers.

A *GUID* is a unique identifier. Many things can have a GUID, such as the application, a connection type, or anything else that you want to have a unique number. GUIDs are so large they virtually ensure uniqueness.

Table 8.1 Service Provider Definitions

#Define	Use
`CLSID_DP8SP_IPX`	IPX-based address objects
`CLSID_DP8SP_MODEM`	Modem-to-modem connections
`CLSID_DP8SP_SERIAL`	Null-modem cable connections
`CLSID_DP8SP_TCPIP`	TCP/IP communication

The Contents of a DirectPlay Address

So far, all I have shown you is how to create a DPlay address and then how to set the service provider. But what does that really mean? Well, here I'll show you what a DPlay address really looks like.

The address is just a string that contains value pairs. The first part of the address always contains the following text.

```
x-directplay:/
```

When you create a DPlay address, it starts out containing this text, which does nothing more than let DirectPlay know it is a real address string. As you learned earlier, the address must specify a service provider to work properly. Once you call the `SetSP()` function, it fills in the address string with the provider's GUID. Here is what the address looks like once you have set the service provider.

```
x-directplay:/provider=%7BEBFE7BA0-628D-11D2-AE0F-006097B01411%7D;
device=%7BIP ADAPTER GUID%7D;
```

Wow, that looks messy, doesn't it? If you look closely, you can pick out a few components that have been added to the address. The first component is the *provider*, where the service provider's GUID is stored. The second parameter is the *device*, which stores the GUID for the device on your computer the connection will use. If you pick out these components separately, you can visualize them in the following list.

```
x-directplay:/
provider=%7BEBFE7BA0-628D-11D2-AE0F-006097B01411%7D;
device=%7BIP ADAPTER GUID%7D;
```

Every string in the address must have a matching value. Table 8.2 contains a list of available address strings. From the table, you can piece together any type of connection address information you need. For example, to use TCP/IP and connect to IP 192.168.0.2 on port 6000, you would use the following elements.

```
x-directplay:/
provider=%7BEBFE7BA0-628D-11D2-AE0F-006097B01411%7D;
device=%7BIP ADAPTER GUID%7D;
hostname=192.168.0.2;
port=6000;
```

All of these elements put together into one string represent a network address. Take a look at Figure 8.7 to see this concept illustrated.

Table 8.2 Address Strings

Name	Description
`Provider`	The GUID for the service provider to use
`Device`	The device to use for connecting
`HostName`	The name of the host computer; used mainly for TCP/IP
`Port`	The port on the host computer; used in TCP/IP or IPX
`ApplicationInstance`	The GUID of the game to which you want to connect
`Baud`	The baud rate for modem or serial connections
`StopBits`	The stop bits for modem or serial communications
`Parity`	The parity of modem or serial connections
`PhoneNumber`	The phone number to call for modem connections
`FlowControl`	The flow control for modem or serial connections
`Program`	An optional program GUID

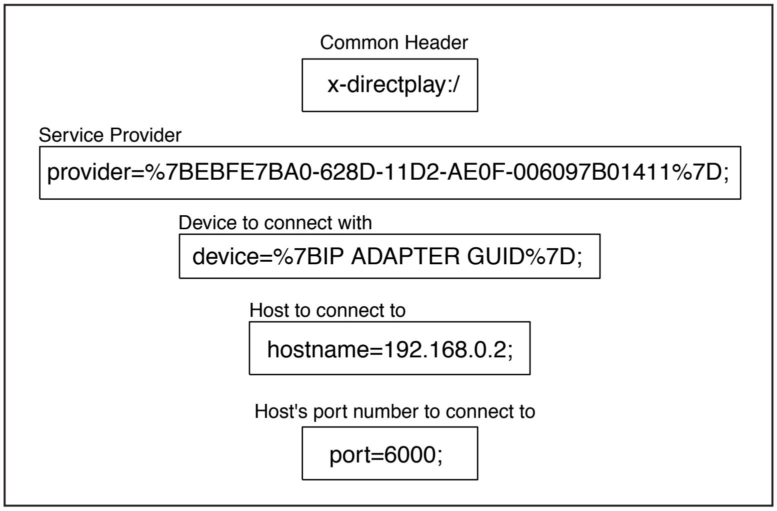


Figure 8.7

String elements combined to create a DirectPlay address

You can also specify user data by putting a # at the end. After the #, you can put whatever application-specific strings you want.

TIP

DirectPlay addresses use the :/ separator with only one slash to avoid confusion with standard Internet addresses. Be sure to never use // as the separator with DPlay, or it will not work.

IDirectPlay8Address Functions

The `IDirectPlay8Address` interface lets you work with it in two different ways. First, you can access the address string information directly and handle it yourself. Second, you can use many of the functions provided by the interface to modify the string. Here I am going to show you some of the various functions provided by this interface.

Because all of the available functions use the address string in some fashion, you will notice that most of them are rather simplistic. Take a look at Table 8.3 to see the functions and brief descriptions of them.

The only strange one in the group is the `BuildFromDPADDRESS()` function. You use this member function to make DirectX 8 games compatible with DirectPlay 4 interfaces.

One nice part about DPlay addresses is that you don't have to use any of the functions if you don't want to. You can just create the entire string by hand and then use the `BuildFromURLA()` function to load it. The main drawback to this method is that your code might be difficult to update when new releases of DirectPlay come out.

Components of an Address

Each of the elements in an address, such as a hostname, is called a component. When you create an address, you specify different components to add to it. For most situations, you use functions to modify addresses, and the most common one is `AddComponent()`. The `AddComponent()` member function's prototype looks like the following code.

```
HRESULT AddComponent(
                                          const WCHAR *const pwszName,
                                          const void *const lpvData,
                                          const DWORD dwDataSize,
                                          const DWORD dwDataType
);
```

Table 8.3 IDirectPlay8Address Functions

Name	Description
`BuildFromURLW`	Builds a DPlay address from a wide string
`BuildFromURLA`	Builds a DPlay address from a string
`Duplicate`	Duplicates the DPlay address
`SetEqual`	Sets the address to be equal to another address
`IsEqual`	Checks whether two addresses are equal
`Clear`	Clears out the address string
`GetURLW`	Retrieves the address in wide string format
`GetURLA`	Retrieves the address in string format
`GetSP`	Retrieves the service provider GUID from the address
`GetUserData`	Retrieves any user-specified data
`SetSP`	Sets the service provider GUID in the address
`SetUserData`	Sets user data in the address
`GetNumComponents`	Retrieves the number of components in the address
`GetComponentByName`	Retrieves the contents of a component by name
`GetComponentByIndex`	Retrieves the contents of a component by position
`AddComponent`	Adds a component to the address
`GetDevice`	Retrieves the device in the address
`SetDevice`	Sets the device in the address
`BuildFromDPADDRESS`	Sets the address to be equal to a specified DPlay4 address

This function lets you add a different component to the address without accessing the string directly. The first parameter expects the name of the component to add. This parameter has several defines set up for you to use, which you can see in Table 8.4.

Each component definition in Table 8.4 corresponds to a string definition. Some people find it easier to remember the definitions, while others find it easier to remember the string names. You can use either.

Table 8.4 IDirectPlay8Address Component Definitions

Name	Description
`DPNA_KEY_APPLICATION_INSTANCE`	The application instance
`DPNA_KEY_BAUD`	Modem baud rate
`DPNA_KEY_DEVICE`	Connection device
`DPNA_KEY_FLOWCONTROL`	Modem flow control
`DPNA_KEY_HOSTNAME`	IP or hostname
`DPNA_KEY_PARITY`	Modem parity
`DPNA_KEY_PHONENUMBER`	Modem phone number
`DPNA_KEY_PORT`	IP port
`DPNA_KEY_PROGRAM`	Program GUID
`DPNA_KEY_PROVIDER`	Service provider
`DPNA_KEY_STOPBITS`	Modem stop bits

The second parameter in the `AddComponent()` function expects the data to be used, and the third parameter specifies the size of the data in the second parameter. The three possible sizes you can use follow:

```
sizeof(DWORD)
(wcslen(wszYourStringHere)+1)*sizeof(WCHAR)
sizeof(GUID)
```

You use the `wcslen()` function you see in the example to get the length of a wide character string. You add 1 to the size to account for the `NULL` terminator. Finally, the size is multiplied by `sizeof(WCHAR)` to get the total string size. Just be sure to keep this in mind when adding wide character components.

The final parameter expects the type of data of which your component consists. The following types are allowed.

- `DPNA_DATATYPE_STRING`
- `DPNA_DATATYPE_DWORD`

- DPNA_DATATYPE_GUID
- DPNA_DATATYPE_BINARY

Okay, now that I have shown you all of this nifty theory about components, I think it's time to see it in action, so here you go.

```
hReturn = g_pHostAddress->AddComponent(
                              DPNA_KEY_HOSTNAME,
                              wszHostName,
                              (wcslen(wszHostName)+1)*sizeof(WCHAR),
                              DPNA_DATATYPE_STRING);
if( hReturn != S_OK ) {
    MessageBox( hWnd, "Failed to AddComponent()", "hrJoinGame()",
     MB_ICONERROR );
    return -1;
}
```

Well, there it is in all its glory—the `AddComponent()` function. In the example, I am adding the hostname to the address. If the function returns anything but an `S_OK`, then something bad happened. Always be sure to check the return code when using this function. That's about it for addresses; they're really not all that hard to use once you get the hang of the definitions.

The DirectPlay Message Handler

Now announcing the greatest thing since sliced bread—message handlers! Yes, you can own one too for just $19.95, shipping extra. It slices, it dices, it pumps more messages through than Linux on steroids! Okay, get to the point already, right?

While reading the past several chapters, I hope you have learned that multiplayer games operate by sending and receiving messages. DirectPlay uses what is called a message handler to process all communication. The message handler in DirectPlay works a lot like the main message pump used in every Windows program. When messages are received, they are sent to the message handler. The handler then looks at the message and determines what to do with it.

It's been a little while since I showed you an illustration, so take a gander at Figure 8.8 to see the architecture of the message handler. As you can see in the illustration, all messages received are sent into the message handler for processing.

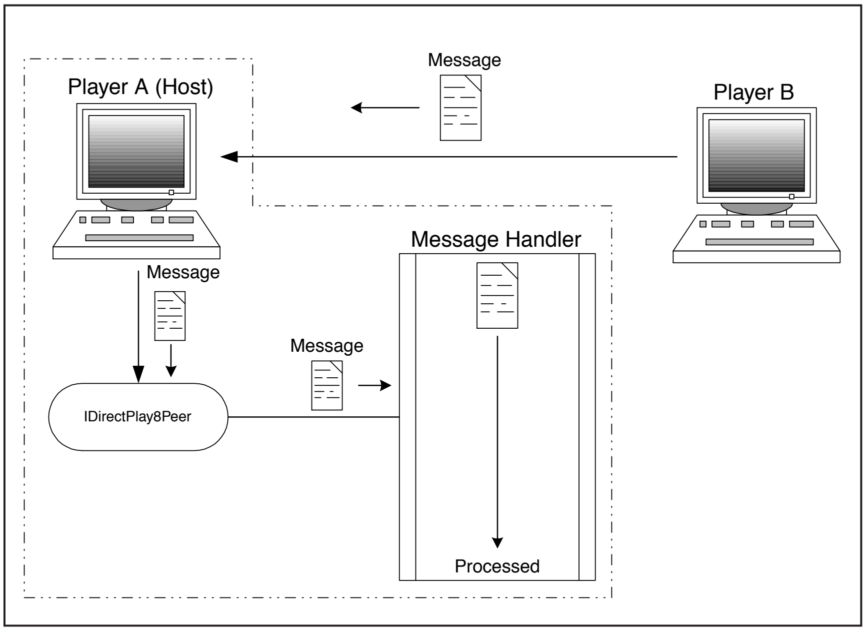


Figure 8.8

The DirectPlay message handler in a peer-to-peer session

Creating the Message Handler

Before you can start using a message handler, you must create it. This process is fairly straightforward no matter which game session type you are using. Whether you have created an `IDirectPlay8Peer`, an `IDirectPlay8Server`, or an `IDirectPlay8Client` interface, you use the same method to set up the handler.

The Initialize() Member Function

All of the interfaces you deal with use the `Initialize()` member function to set up the message handler. For simplicity's sake, I'll show you how to set up the handler in a peer-to-peer game session.

```
// Init IDirectPlay8Peer Message Handler
if( FAILED( hReturn = g_pDP->Initialize( NULL,
 DirectPlayMessageHandler, 0 ) ) ) {
    MessageBox( hWindow, "Failed to Message Handler", "DirectPlay Error",
     MB_ICONERROR );
    return -1;
}
```

In this example, I use an `IDirectPlay8Peer` interface variable called `g_pDP`. I use its member function, called `Initialize()`, to set up the handler. Here is the prototype for the `Initialize()` member function.

```
HRESULT Initialize(
                    PVOID const pvUserContext,
                    const PFNDPNMESSAGEHANDLER pfn,
                    const DWORD dwFlags
);
```

The first parameter points to what is called user-provided context. What this means is that it contains information about the person who sent the message. Important things such as the sender's ID are stored here.

The second parameter expects the name of the message handler function to receive DirectPlay messages. The last parameter is normally set to `NULL`. At this time, you use no flags for this parameter.

The Message Handler Function

To set a function to be the message handler, you must first create one. The function prototype for a DirectPlay message handler looks like this:

```
HRESULT WINAPI MessageHandler( PVOID pvUserContext,
 DWORD dwMessageId, PVOID pMsgBuffer );
```

The first parameter contains the user's context that sent the message. The second parameter contains an identifier to the type of message passed. Different interfaces receive different message types. You can find the total list of message types available in Table 8.5. Just keep in mind that not all the message types shown in Table 8.5 are available to every interface.

The final parameter points to void data that contains the actual message contents.

Processing Messages

Once you create the message-handler skeleton, you need to add logic to process the messages. Here is a brief excerpt from a handler.

```
switch( dwMessageId )
{
    case DPN_MSGID_HOST_MIGRATE:
    {
```

Table 8.5 DirectPlay Message Definitions

#define
DPN_MSGID_ADD_PLAYER_TO_GROUP
DPN_MSGID_ASYNC_OP_COMPLETE
DPN_MSGID_CLIENT_INFO
DPN_MSGID_CONNECT_COMPLETE
DPN_MSGID_CREATE_GROUP
DPN_MSGID_CREATE_PLAYER
DPN_MSGID_DESTROY_GROUP
DPN_MSGID_DESTROY_PLAYER
DPN_MSGID_ENUM_HOSTS_QUERY
DPN_MSGID_ENUM_HOSTS_RESPONSE
DPN_MSGID_GROUP_INFO
DPN_MSGID_HOST_MIGRATE
DPN_MSGID_INDICATE_CONNECT
DPN_MSGID_INDICATED_CONNECT_ABORTED
DPN_MSGID_PEER_INFO
DPN_MSGID_RECEIVE
DPN_MSGID_REMOVE_PLAYER_FROM_GROUP
DPN_MSGID_RETURN_BUFFER
DPN_MSGID_SEND_COMPLETE
DPN_MSGID_SERVER_INFO
DPN_MSGID_TERMINATE_SESSION
DPL_MSGID_CONNECT
DPL_MSGID_CONNECTION_SETTINGS
DPL_MSGID_DISCONNECT
DPL_MSGID_RECEIVE
DPL_MSGID_SESSION_STATUS

```
        vShowText(hLB_Output,"Migrate Host");
        PDPNMSG_HOST_MIGRATE pHostMigrateMsg;
        pHostMigrateMsg = (PDPNMSG_HOST_MIGRATE)pMsgBuffer;
        // Check to see if we are the new host
        if( pHostMigrateMsg->dpnidNewHost == g_dpnidLocalPlayer ) {
                vShowText(hLB_Output,"(HOSTING)");
            }
            break;
    }
}
```

Basically, all I am doing here is using `switch` to check the message ID. Once a match is found, I then create a message structure pointer of the type found. In this case, I create a `PDPNMSG_HOST_MIGRATE` message structure pointer. All message structures begin with `PDPNMSG` and end with their corresponding message IDs.

```
PDPNMSG_HOST_MIGRATE pHostMigrateMsg;
```

Once I have the message structure pointer, I then set it equal to the message buffer passed by the function. To set the pointer, I have to cast the message contents to the pointer type.

```
pHostMigrateMsg = (PDPNMSG_HOST_MIGRATE)pMsgBuffer;
```

Once the message pointer is set, the code can then properly utilize the contents of the message. Here is another case statement example of processing a chat packet.

```
case DPN_MSGID_RECEIVE:
{
    PDPNMSG_RECEIVE pReceiveMsg;
    pReceiveMsg = (PDPNMSG_RECEIVE)pMsgBuffer;
    vShowText(hLB_Output,(char*)pReceiveMsg->pReceiveData);
    break;
}
```

In this example, I am processing a custom game packet that defaults to type `DPN_MSGID_RECEIVE`. Because I know that the packet contains a chat message, I just cast a pointer of its type and then pass the data to the display.

CAUTION

The message handler must be thread-safe because it can be called from multiple locations. Be sure to implement critical sections around areas of shared memory.

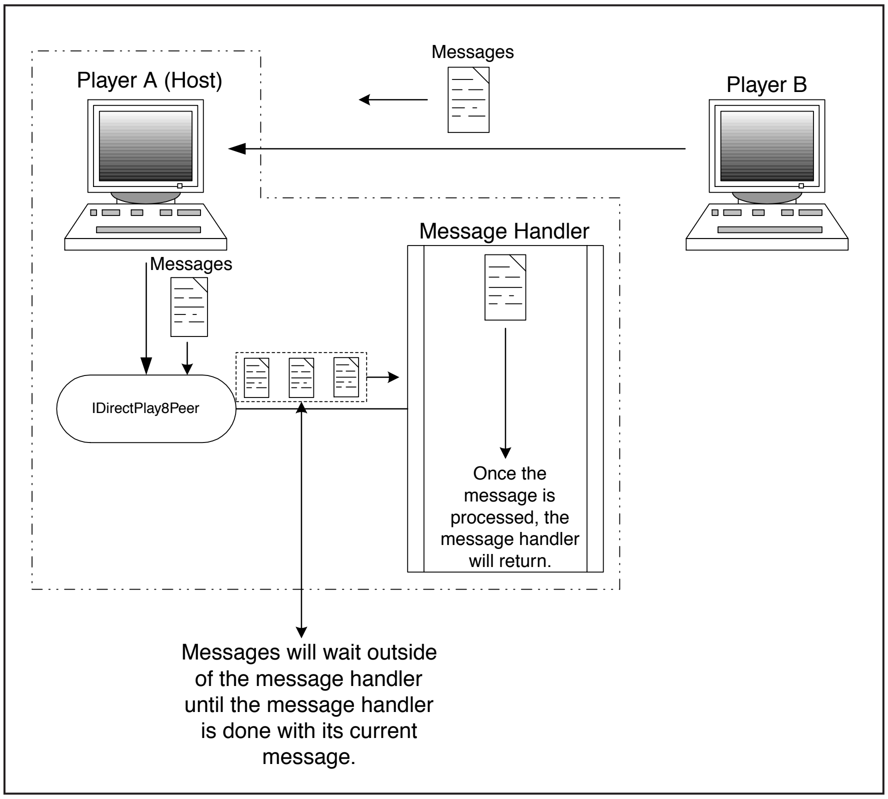

Messages
Player A (Host)
Player B
Message Handler
Messages
IDirectPlay8Peer
Once the message is processed, the message handler will return.
Messages will wait outside of the message handler until the message handler is done with its current message.

Figure 8.9

DirectPlay message traffic

Because messages get queued up for the message handler, you need to process messages quickly. Other messages are not processed until you have returned from the message handler. To see this queuing in action, take a look at Figure 8.9.

All right, now it's time for a little *Asheron's Call* action. I'll see you back here in 30 minutes.

Peer-to-Peer Sessions

Wow, that 30-minute break turned into a two-day ordeal. The state I live in was hit by a major ice storm. More than 200,000 homes, including mine, were without power! Well, power is back where I live, so it's time to continue. I did talk a little theory earlier about peer-to-peer games, but I didn't really show you the underlying functionality available. What I am going to do now is walk you through creating a peer-to-peer game session. I'll cover both hosting and joining.

Initializing a Peer-to-Peer Session

Regardless of whether you are joining or hosting a peer-to-peer game, you follow a few common steps. Here are the common steps, in order.

1. Initialize COM.
2. Create an `IDirectPlay8Peer` interface.
3. Initialize the message handler.
4. Create a device address.
5. Set the service provider.
6. Create a host address.

Initializing COM

That doesn't seem so bad, right? The first step is to initialize COM, which must be done within threads that use the DPlay objects. Because the main application is nothing more than one giant thread, you must initialize COM. Here is the code for doing so.

```
// Initialize COM
hReturn = CoInitialize( NULL );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Error Initializing COM", "DirectPlay Error",
     MB_ICONERROR );
    return hReturn;
}
```

The function you should notice here is `CoInitialize()`. This function initializes the COM system, which has to happen only once per thread. Remember to follow up each call to this function with `CoUninitialize()` before you exit your game. It cleans things up for you. That's it for COM; just remember to do it, or nothing else in DirectPlay will work.

Creating an IDirectPlay8Peer Interface

Before you can start using peer-to-peer game functions, you must first create an interface to a peer-to-peer game object. You only have to create one of these interfaces for each connection your game will manage. Most games only manage one connection, their own. Here is the code to initialize the peer-to-peer interface.

```
// Our global peer Interface
IDirectPlay8Peer     g_pDP;
```

```
// Create IDirectPlay8Peer Object
if( FAILED( hReturn = CoCreateInstance( CLSID_DirectPlay8Peer,
                NULL,
                CLSCTX_INPROC_SERVER,
                IID_IDirectPlay8Peer,
                (LPVOID*) &g_pDP ) ) ) {
      MessageBox( hWindow, "Can't Create DPlayPeer", "DirectPlay Error",
       MB_ICONERROR );
}
```

No big surprises here either; you are using one function only to create the interface, and the function is called CoCreateInstance(). Here is the prototype for CoCreateInstance().

```
STDAPI CoCreateInstance(     REFCLSID rclsid,
                                            LPUNKNOWN pUnkOuter,
                                            DWORD dwClsContext,
                                            REFIID riid,
                                            LPVOID * ppv
                                      );
```

If that doesn't look a little confusing to you, then you belong with the CHUDs. (Just kidding.) This function can still be a little daunting to non-veterans. Because COM is a strange beast to begin with, it results in strange-looking functions. You use the CoCreateInstance() function when you want to create a local copy of a COM object. DirectX is a COM-based system, so whenever you want to use one of its objects, you have to get a local copy of it.

The first parameter in this function expects the *CLSID* that matches the object you are trying to retrieve. For this parameter, I am passing CLSID_DirectPlay8Peer, which is a define for the GUID that corresponds to a peer interface.

A *CLSID* is a globally unique identifier, or GUID, associated with a class object. To have a CLSID, a class must be registered in the system.

The second parameter for CoCreateInstance() tells whether the object is part of an aggregate. For our use, just leave this at NULL.

The third parameter tells how the object's code will be processed. By passing a CLSCTX_INPROC_SERVER to it, I am telling the system that the object will run in the same process from which it is called. For all of the examples in this book, this setting remains the same.

TIP

You can find all of the CLSIDs for DirectPlay in the dplay8.h header file that comes with DirectX.

The fourth parameter contains the reference to the interface of the object. For this parameter, I am passing an IID_IDirectPlay8Peer, which tells the object to use the peer interface. These types of references are commonly referred to as IIDs.

TIP

You can find all of the IIDs for DirectPlay in the dplay8.h header file that comes with DirectX.

The last parameter holds the address of the variable to receive the interface pointer. In this example, I am passing in a variable of type IDirectPlay8Peer.

If everything goes well, by this time you have created a peer-to-peer interface. If CoCreateInstance() returns anything other than S_OK, then something went wrong. Well, that's it for creating the COM interface; now it's time to show you how to initialize the message handler.

Initializing the Message Handler

I have already talked about this quite a bit in the "Creating the Message Handler" section earlier in this chapter, but here I'll show you step-by-step how to initialize one. Here is the code to initialize the handler.

```
// Init IDirectPlay8Peer Message Handler
if( FAILED( hReturn = g_pDP->Initialize( NULL, DirectPlayMessageHandler, 0 ) ) ) {
    MessageBox( hWindow, "Failed to Message Handler", "DirectPlay Error",
     MB_ICONERROR );
    return -1;
}
```

What I have done here is call the Initialize() member function of the IDirectPlay8Peer interface. If you missed it earlier, you can read about this function in the "The Initialize() Member Function" section earlier in this chapter.

To pass in DirectPlayMessageHandler for my message handler, I had to have first created such a function. Here is the code for the message handler.

```
HRESULT WINAPI DirectPlayMessageHandler(
                                  PVOID pvUserContext,
                                  DWORD dwMessageId,
                                  PVOID pMsgBuffer )
{
    HRESULT     hReturn = S_OK;
    switch( dwMessageId ) {
                  // Process messages here
       }
       return hReturn;
}
```

Trust me; there is more to this function than I demonstrate here, but it is enough to get a peer program up and running. The message handler has been initialized by this time, so now it's off to create the device address.

Creating the Device Address

Because you need a device address to represent your machine's method for communication, one must be created. For a quick refresher on why this is necessary, read the "DirectPlay Addresses" section earlier in this chapter. Here is the code for creating the device address.

```
// Create a device address
hReturn = CoCreateInstance(CLSID_DirectPlay8Address,
                                          NULL,
                                          CLSCTX_INPROC_SERVER,
                                          IID_IDirectPlay8Address,
                                          (LPVOID*) &g_pDeviceAddress );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Failed to Create Device",
     "CoCreateInstance()", MB_ICONERROR );
    return -1;
}
```

It's the ol' CoCreateInstance() function again. This time, I use the class ID of CLSID_DirectPlay8Address to get an interface to IID_IDirectPlay8Address that is loaded into my IDirectPlay8Address variable called g_pDeviceAddress. Cool, I got to use a lot of neat buzzwords in one sentence! Okay, now you have a device address; it's time to set the service provider.

Setting the Service Provider

Now that you have an address shell, it is time to set its service provider. To do this, I use the SetSP() member function available to objects of type IDirectPlay8Address. Here is the code.

```
// Set our service provider to TCP/IP
if( FAILED( hReturn = g_pDeviceAddress->SetSP( &CLSID_DP8SP_TCPIP ) ) ) {
    MessageBox( hWindow, "Failed to SetSP() for Device Addr",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
```

For this instance, I passed the GUID for a TCP/IP connection. You can use whichever service provider GUID you want here, but I like TCP/IP.

You are done with the device address, so it's time to create the host address.

TIP

You can find all the service provider GUIDs for DirectPlay in the dplay8.h header file that comes with DirectX.

Creating the Host Address

You follow the same steps for creating the host address as you did for creating the device address. The only difference is that I use `g_pHostAddress` as the `IDirectPlay8Address` variable to hold the interface pointer. I don't want to leave you hanging, so here is the code.

```
// Create a host address
hReturn = CoCreateInstance( CLSID_DirectPlay8Address,
 NULL,CLSCTX_INPROC_SERVER,  IID_IDirectPlay8Address,
 (LPVOID*) &g_pHostAddress );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Failed to Create Host Address()",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
// Set the host address to TCP/IP
if( FAILED( hReturn = g_pHostAddress->SetSP( &CLSID_DP8SP_TCPIP ) ) ) {
    MessageBox( hWindow, "Failed to SetSP() for Host Addr",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
```

Take a peek at Figure 8.10 to see a graphical outlook of what has been accomplished so far.

Guess what? That's it for the common code for peer-to-peer sessions! Now that you have that out of the way, I can show you how to join and host peer games.

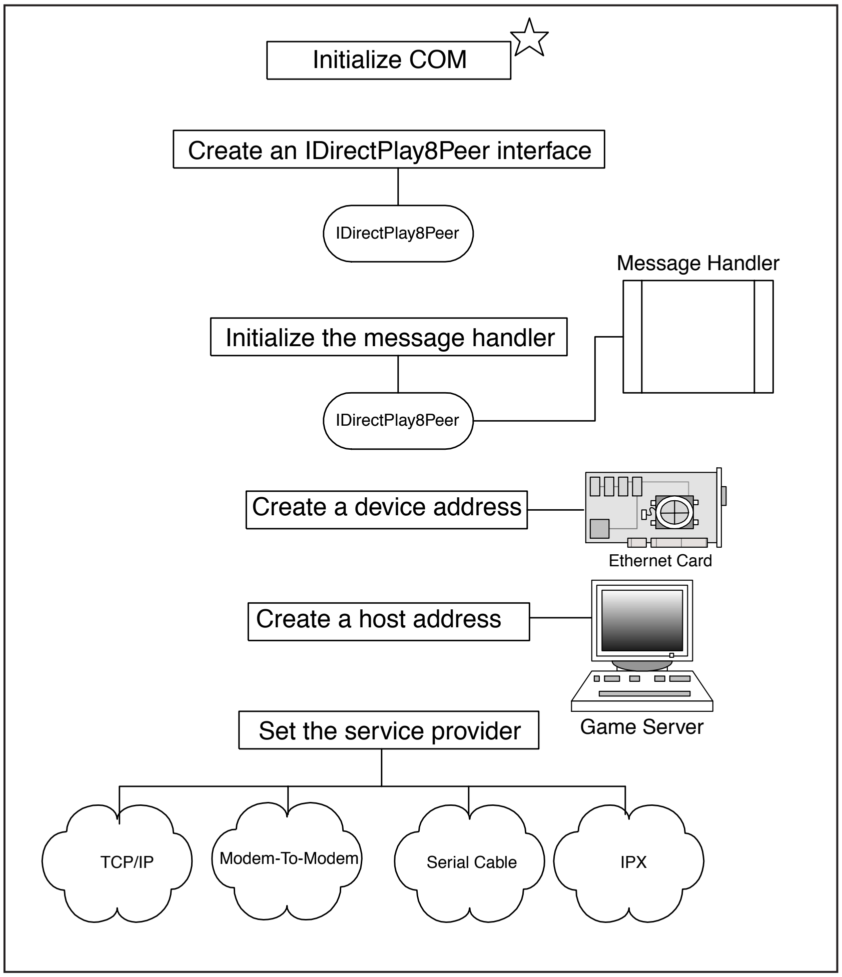


Figure 8.10

Common operations among peer-to-peer sessions

Hosting a Peer-to-Peer Session

After all of this reading, it is finally time to see how to host a peer-to-peer session—drum roll, please. Tada! That's it. You are now hosting. What's that? The mind-meld didn't work? Fine, then; here is the code for hosting.

```
// Host the game
hReturn = g_pDP->Host(   &dnAppDesc,
                                        &g_pDeviceAddress,
                                        1,
                                        NULL,
                                        NULL,
                                        NULL,
                                        NULL );
```

For this call, I use the `Host()` member function available to the `IDirectPlay8Peer` interface. The first parameter expects the description of the application you are hosting. I'll talk about that in a minute.

The second parameter expects an `IDirectPlay8Address` that corresponds to your device address. I use the `g_pDeviceAddress` variable that was created earlier. The device address now becomes the host device. The third parameter tells the function how many addresses are contained within the second parameter's object. I set this to `1` because only one address is stored in the device address.

The fourth and fifth parameters are reserved for future functionality and should be set to `NULL`. The sixth parameter can hold optional player context information. For our purposes here, I chose not to use this information and I set it to `NULL`.

The last parameter lets you specify different flags. One flag of interest is `DPNHOST_OKTOQUERYFORADDRESSING`. If you set this flag, then you are allowing DirectPlay to display a pop-up window asking for additional connection information if it needs it. This step usually is not a desirable one to use because most games run in full-screen mode. But if you want DPlay to handle this sort of thing, feel free to use it. Just make sure you have a valid window for use.

If everything goes okay and `hReturn` does not fail, then you will be up and hosting a session! Oh, yeah; I almost forgot about application descriptions from the first parameter.

Application Descriptions

An application description in DirectPlay is a structure of type `DPN_APPLICATION_DESC` and does just what it says—it describes the application. To better understand what this means, check out the definition of the structure.

```
typedef struct _DPN_APPLICATION_DESC {
    DWORD  dwSize;
    DWORD  dwFlags;
    GUID   guidInstance;
    GUID   guidApplication;
    DWORD  dwMaxPlayers;
    DWORD  dwCurrentPlayers;
    WCHAR* pwszSessionName;
    WCHAR* pwszPassword;
    PVOID  pvReservedData;
    DWORD  dwReservedDataSize;
    PVOID  pvApplicationReservedData;
    DWORD  dwApplicationReservedDataSize;
} DPN_APPLICATION_DESC, *PDPN_APPLICATION_DESC;
```

Luckily, most of the members are more or less self-descriptive. Before you can begin using an application description, you should clear it. Here is the code to clear the structure you use in hosting.

```
// The application description
DPN_APPLICATION_DESC     dnAppDesc;
// Clear out the application description
ZeroMemory( &dnAppDesc, sizeof(DPN_APPLICATION_DESC) );
```

As you can see, I used the `ZeroMemory()` function to clear the structure. Not much to this, just your standard memory-clearing stuff.

The first member, `dwSize`, holds the size of the `DPN_APPLICATION_DESC` structure. For this—you use the following code.

```
dnAppDesc.dwSize     = sizeof(DPN_APPLICATION_DESC);
```

You need to make this call every time you create a description, so get used to using it a lot.

You use the second member to store flags that determine the application's behavior. Here are the flags available to you.

- `DPNSESSION_CLIENT_SERVER`
- `DPNSESSION_MIGRATE_HOST`
- `DPNSESSION_NODPNSVR`
- `DPNSESSION_REQUIREPASSWORD`

Unless you want the session to allow migration, you can leave flags set to `NULL` for peer sessions. Otherwise, you use the following code.

```
dnAppDesc.dwFlags     = DPNSESSION_MIGRATE_HOST;
```

If you use the `DPNSESSION_REQUIREPASSWORD` flag, then the `pwszPassword` member variable must be populated with the session's password. You can also mix flags with the `OR (|)` operator.

The third member stores the GUID that DPlay creates for the instance of the application. There is no need to do anything with this member for hosting. The fourth member stores the GUID of the game. Be sure to populate it with the GUID for the application. Normally, this is set in a define somewhere. The fifth member is a DWORD that sets the maximum number of players that can join the game session. If you set this to 0, then infinite players can join.

The sixth member is only filled when you use the GetApplicationDescription() function to retrieve an application's description. It contains the number of players currently in a session. The seventh member holds a Unicode string representing the session name. It has no bearing on the actual game and cannot be used by clients for connecting. The eighth member holds a Unicode password for the session. This optional member requires that the DPNSESSION_REQUIREPASSWORD flag be present in the dwFlags member.

The ninth and tenth members are reserved and should always be left NULL. The eleventh member allows you to point to reserved data specific to the application. This is optional, and I do not use it for my examples. The last member is a DWORD that tells the size of the reserved data in the eleventh member.

Setting Peer Information

One task that you must handle before calling the Host() member function for the IDirectPlay8Peer interface is setting your player information. You do this so that other players can have access to your attributes. To set your player or peer information, you use the DPN_PLAYER_INFO data structure. You'd better buckle-up; I hear another structure storm brewing.

```
typedef struct _DPN_PLAYER_INFO{
    DWORD  dwSize;
    DWORD  dwInfoFlags;
    PWSTR  pwszName;
    PVOID  pvData;
    DWORD  dwDataSize;
    DWORD  dwPlayerFlags;
} DPN_PLAYER_INFO, *PDPN_PLAYER_INFO;
```

This structure holds everything about players in the game. Keep in mind that this information is static; therefore, it does not change. The first member might look familiar because it is the size of the DPN_PLAYER_INFO structure.

The second member stores flags that tell what type of information is stored in the structure. When you are setting player information, the only flag needed is DPNINFO_NAME. If you were using the GetPlayerInfo() function, then the flags would be set to both DPNINFO_NAME and DPNINFO_DATA because data would be returned.

The third member points to a Unicode string that holds the player's name. The fourth member points to data that describes the player. The fifth member tells the size of the data from the fourth member.

The last member can contain either the DPNPLAYER_LOCAL flag, with data for the local player, or the DPNPLAYER_HOST flag, with data for the host machine.

That's it for the structure definition; here is the code I use for the host example.

```
// Create local player Information
DPN_PLAYER_INFO                      dpPlayerInfo;
// Clear out the structure
ZeroMemory( &dpPlayerInfo, sizeof(DPN_PLAYER_INFO) );
// Set the size
dpPlayerInfo.dwSize = sizeof(DPN_PLAYER_INFO);
// This structure contains the player name
dpPlayerInfo.dwInfoFlags = DPNINFO_NAME;
// Set the name
dpPlayerInfo.pwszName = wszPeerName;
```

Notice that I used the ZeroMemory() function again. You will see this used for just about every data structure I cover. It is always good practice to clear out memory before you use it.

The SetPeerInfo() Member Function

Now that the player information structure is created, you must set the peer information in the g_pDP variable. You can do this using the SetPeerInfo() member function. Here is the code.

```
// Set us up to be non-asynchronous
if( FAILED( hReturn = g_pDP->SetPeerInfo( &dpPlayerInfo, NULL, NULL,
 DPNSETPEERINFO_SYNC ) ) ) {
    MessageBox( hWindow, "Failed to SetPeerInfo()", "DirectPlay Error",
     MB_ICONERROR );
    return -1;
}
```

Here is the prototype for SetPeerInfo().

```
HRESULT SetPeerInfo(
                          const DPN_PLAYER_INFO *const pdpnPlayerInfo,
                          PVOID const pvAsyncContext,
                          DPNHANDLE *const phAsyncHandle,
                          const DWORD dwFlags
                              );
```

From the previous code, you can see that the first parameter expects the local player information in the form of a DPN_PLAYER_INFO structure. For this, I pass in the address of the dpPlayerInfo variable.

The second parameter contains user-specified context. I don't use this, so I keep it set to NULL. The third parameter passes back an asynchronous handle. It can't be used, though, so I don't bother passing a DPNHANDLE to it. The fourth parameter lets you specify which flags you want to use with the call. The only flag you can use is DPNSETPEERINFO_SYNC. This flag tells the operation to perform synchronously.

Before continuing, make sure you always check the return code of this function. As long as it returns an S_OK, you are fine.

Hosting Review

I have now shown you all there is to hosting, but I want to review the steps. Here is the complete hosting code, taken from my chat example in the next chapter.

```
HRESULT     hrHostGame( HWND hWindow)
{
    HRESULT                                 hReturn;
    char                                    szPeerName[256];
    char                                    szSessionName[256];
    WCHAR                                   wszPeerName[256];
    WCHAR                                   wszSessionName[256];
    DPN_APPLICATION_DESC                    dnAppDesc;
    char                                    szPort[6];
    DWORD                                   dwLength = 256;
    DPN_PLAYER_INFO                         dpPlayerInfo;
    DWORD                                   dwPort = 9000;

    // Set up our player information
    GetWindowText(hEB_InputName,szPeerName,36);
    // Get name from Window Edit Box
    DXUtil_ConvertGenericStringToWide( wszPeerName, szPeerName );
    ZeroMemory( &dpPlayerInfo, sizeof(DPN_PLAYER_INFO) );
    dpPlayerInfo.dwSize = sizeof(DPN_PLAYER_INFO);
    dpPlayerInfo.dwInfoFlags = DPNINFO_NAME;
    dpPlayerInfo.pwszName = wszPeerName;

    // Set us up to be non-asynchronous
```

```
if( FAILED( hReturn = g_pDP->SetPeerInfo( &dpPlayerInfo, NULL, NULL,
 DPNSETPEERINFO_SYNC ) ) ) {
    MessageBox( hWindow, "Failed to SetPeerInfo()",
     "DirectPlay Error", MB_ICONERROR );
    return -1;
}

// Set up the application description
sprintf(szSessionName,"%s's Game",szPeerName);
DXUtil_ConvertGenericStringToWide( wszSessionName, szSessionName );
ZeroMemory( &dnAppDesc, sizeof(DPN_APPLICATION_DESC) );
dnAppDesc.dwSize            = sizeof(DPN_APPLICATION_DESC);
dnAppDesc.guidApplication   = DP_CHAT;
dnAppDesc.pwszSessionName   = wszSessionName;
dnAppDesc.dwMaxPlayers      = 4;
dnAppDesc.dwFlags           = 0;

// Get port from edit box
GetWindowText(hEB_InputServerPort,szPort,6);
// Convert the port string to a DWORD
dwPort = atol(szPort);
// Add port number to address
hReturn = g_pDeviceAddress->AddComponent(
 DPNA_KEY_PORT,&dwPort,sizeof(DWORD),DPNA_DATATYPE_DWORD);
if( hReturn != S_OK ) {
    MessageBox( hWindow, "Failed to AddComponent()", "hrHost()",
     MB_ICONERROR );
    return -1;
}
// Host the game
hReturn = g_pDP->Host(     &dnAppDesc, &g_pDeviceAddress, 1, NULL,
 NULL, NULL, NULL );
if( FAILED( hReturn ) ) {
    MessageBox( hWindow, "Failed to Host()", "DirectPlay Error",
     MB_ICONERROR );
    return -1;
}
// Let us know we are the host
bHost = 1;
```

```
    vShowText(hLB_Output,"<- Hosting ->");
    return hReturn;
}
```

From this code, you can see that I completed the following steps to host the session.

1. Set up peer information using the `SetPeerInfo()` function and the `DPN_PLAYER_INFO` data structure.
2. Set up the application description with the `DPN_APPLICATION_DESC` data structure.
3. Optionally, add the port number to the device address.
4. Host the game with `IDirectPlay8Peer` interface's `Host()` member function.

Take a breather. You are done with hosting for now. An illustration of the steps I just discussed appears in Figure 8.11. Just wait until the next chapter, where I show you how to use it in a full application. Until then, here is how you join a session.

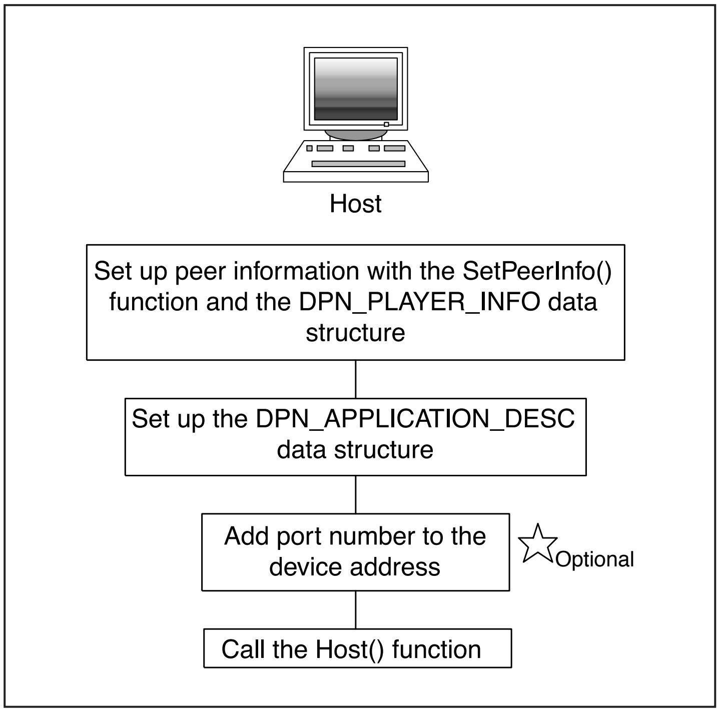

Host
Set up peer information with the SetPeerInfo() function and the DPN_PLAYER_INFO data structure
Set up the DPN_APPLICATION_DESC data structure
Add port number to the device address
Optional
Call the Host() function

Figure 8.11

Hosting a peer session

Joining a Peer-to-Peer Session

I've discussed most of the hard work already. To join a peer-to-peer session, you follow mostly the same steps as when hosting. The main difference is that you have to specify the host's name and use the `Connect()` function instead of the `Host()` function. Here are the steps you follow to join a session.

1. Set up the peer information.
2. Set up the application description.
3. Set the host's name and optional port number.
4. Connect to the host using the `Connect()` function.

Setting the Peer Information When Joining

Well, what can I say? This part is identical to setting the peer information when hosting. Just use a `DPN_PLAYER_INFO` data structure and call the `SetPeerInfo()` member function. Wasn't that painless?

Setting the Application Description When Joining

Setting the application description when joining is a little shorter than setting the application description when hosting. All you have to do is create the description and load it with the size and GUID for the application. Here is the code to accomplish this.

```
// Prepare the application description
ZeroMemory( &dpnAppDesc, sizeof( DPN_APPLICATION_DESC ) );
dpnAppDesc.dwSize = sizeof( DPN_APPLICATION_DESC );
dpnAppDesc.guidApplication = DP_CHAT;
```

Because the host is responsible for setting the maximum number of players and all of that other data, all the joiners have to worry about is setting the application GUID.

Setting the Host's Name and Optional Port Number

To set up the host's name and optional port number, I use the `AddComponent()` function mentioned earlier in this chapter in the "Components of an Address" section. Here is the code for doing so.

```
// Add host name to address
hReturn = g_pHostAddress->AddComponent(DPNA_KEY_HOSTNAME,wszHostName,(
wcslen(wszHostName)+1)*sizeof(WCHAR),DPNA_DATATYPE_STRING);
if( hReturn != S_OK ) {
    MessageBox( hWnd, "Failed to AddComponent()", "hrJoinGame()",
     MB_ICONERROR );
    return -1;
}
// Add port number to address
hReturn = g_pHostAddress->AddComponent(
DPNA_KEY_PORT,&dwPort,sizeof(DWORD),DPNA_DATATYPE_DWORD);
if( hReturn != S_OK ) {
    MessageBox( hWnd, "Failed to AddComponent()", "hrJoinGame()",
     MB_ICONERROR );
    return -1;
}
```

What I did here was call `AddComponent()` twice. I called it the first time to add the host's IP address. The second time, I called it to add the port number.

Connecting with the Connect() Function

Ah, here we are, the meat of connecting. Once all of the previous steps are out of the way, you are ready to establish the actual connection. Here is the prototype for the `IDirectPlay8Peer::Connect()` function.

```
HRESULT Connect(
                        const DPN_APPLICATION_DESC *const pdnAppDesc,
                        IDirectPlay8Address *const pHostAddr,
                        IDirectPlay8Address *const pDeviceInfo,
                        const DPN_SECURITY_DESC *const pdnSecurity,
                        const DPN_SECURITY_CREDENTIALS *const pdnCredentials,
                        const void *const pvUserConnectData,
                        const DWORD dwUserConnectDataSize,
                        void *const pvPlayerContext,
                        void *const pvAsyncContext,
                        DPNHANDLE *const phAsyncHandle,
                        const DWORD dwFlags
                             );
```

Don't fret too much; it's really not too bad. The first parameter expects the application description for your game. The second parameter expects the address of the host. The third parameter expects the address of the connection device. The fourth and fifth parameters are always set to NULL. The sixth parameter points to application-specific data. This is optional, and I normally leave it at NULL.

The seventh parameter sets the size of the data in the previous parameter. Because I don't use any application data, I set this to 0. The eighth parameter points to user context data and can optionally be set to NULL. The ninth parameter points to user-supplied context data that is returned when a DPN_MSGID_CONNECT_COMPLETE system message is received. The tenth parameter points to a DPNHANDLE handle, which allows you to cancel the asynchronous nature of connecting with the IDirectPlay8Peer::CancelAsyncOperation() function. This must be set to NULL if you specify synchronous connecting.

The last parameter lets you set certain flags for the connection. The first flag, DPNCONNECT_OKTOQUERYFORADDRESSING, tells DirectPlay that it is okay to query for any additional connection information. Normally, you would not do this because most games operate in full-screen mode. The second flag possible is DPNCONNECT_SYNC, and it tells the connection to operate synchronously.

Once you call the Connect() function, the application continues running. When the host accepts the connection, you receive a DPN_MSGID_CONNECT_COMPLETE message. Once the message is received, the connection is complete.

Here is the code showing the connection function.

```
// Connect to the session
hReturn = g_pDP->Connect(     &dpnAppDesc,
                                    g_pHostAddress,
                                    g_pDeviceAddress,
                                    NULL,
                                    NULL,
                                    NULL,
                                    0,
                                    NULL,
                                    NULL,
                                    &g_hConnectAsyncOp,
                                    NULL); // flags
```

First, I pass in the application description. It is followed by the host address and then the device address. Several `NULL`s follow before the asynchronous handle, and then for flags, I use `NULL` again. That's really about it for connecting. In the next chapter, I'll give you the full skinny on setting up a chat program. Before you leave, be sure to check out Figure 8.12 to see the steps for joining a peer session.

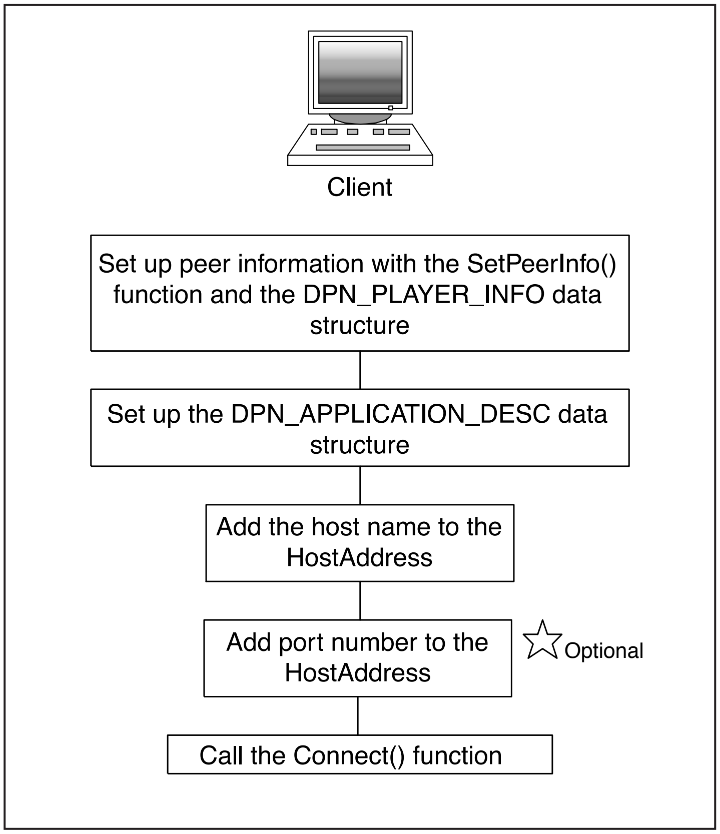

Client
Set up peer information with the SetPeerInfo() function and the DPN_PLAYER_INFO data structure
Set up the DPN_APPLICATION_DESC data structure
Add the host name to the HostAddress
Add port number to the HostAddress
Optional
Call the Connect() function

Figure 8.12

Joining a peer game

Review

In this chapter you have learned all about DirectPlay and what its place is in the DirectX SDK. Here is a quick review of some of the key points discussed.

- DirectPlay is the networking portion of DirectX. It handles all communication methods, ranging from modem-to-modem to LAN connections.
- The different types of connections available in DirectPlay are referred to as service providers. This book only deals with TCP/IP.
- Sessions define individual games.
- The DirectPlay address is used to store connection information such as a player's URL.
- The DirectPlay message handler is used to receive all game communication.
- DirectPlay is based on COM; therefore, you must initialize COM before using it.

CHAPTER 9

Chatting with DirectPlay

Howdy, partner! What a lame introduction, eh? Yeah, I thought so, too. Well, here you are at the gates to paradise, just waiting to get in to see how to write a complete DirectPlay application. Here you go, a season pass for one; the show starts in 10 seconds.

In this chapter, I'll show you how to write a complete chat program using DirectPlay. The program uses standard Windows dialogs for graphics to keep things as simple as possible. Currently, the application allows up to four people to chat at once, but you can increase this number if you so desire. I'll cover the following topics in this chapter.

- How to set up the project space for DPChat
- The components of the header file
- How to initialize DirectPlay
- How to join the game
- How to host the game
- How to send chat messages

Quiet, please; it's curtain time!

Creating the Project Space

Before you can start looking at code, you need to have a project space in which to work. If you want to, you can just load the project I created for you. But that wouldn't be as much fun, would it? Setting up Visual C++ for DirectPlay isn't too painful; just follow along with me here.

1. Make sure you have the DirectX SDK installed. It's on the CD that came with the book. You should also make sure that your compiler has paths to the directx8/include, directx8/lib, directx8/samples/multimedia/common/src, and directx8/samples/multimedia/common/include folders. If you installed the DirectX SDK folder into a folder other than directx8, replace instances of directx8 with your base folder name.
2. Create a new Win32 application project in Visual C++.
3. Under the Project Settings menu, select the Link tab.
4. In the Object/Libraries edit box, enter the winmm.lib and dplay.lib libraries.

5. Select the C/C++ tab.
6. From the Category drop-down menu, select Code Generation.
7. In the Use Run-Time Library drop-down menu, select Multithreaded.
8. Select OK.
9. Under the Project menu, select Add To Project, and then select Files. Browse to where you installed DirectX 8. Under directx8/samples/multimedia/common/src/ you will find the dxutil.cpp file. Select this file, and then select OK. Repeat this step again, but this time add the dxutil.h file from the directx8/samples/multimedia/common/include directory to the project.

If you followed the steps properly, you should now have an empty project in which to create DirectPlay applications. To hold the main program, you need to create a .cpp file called DPChat.cpp. Go ahead and do this now.

To keep things clean, go ahead and add a new header file to the project and call it DPChat.h.

Okay, your project architecture should look like that shown in Figure 9.1.

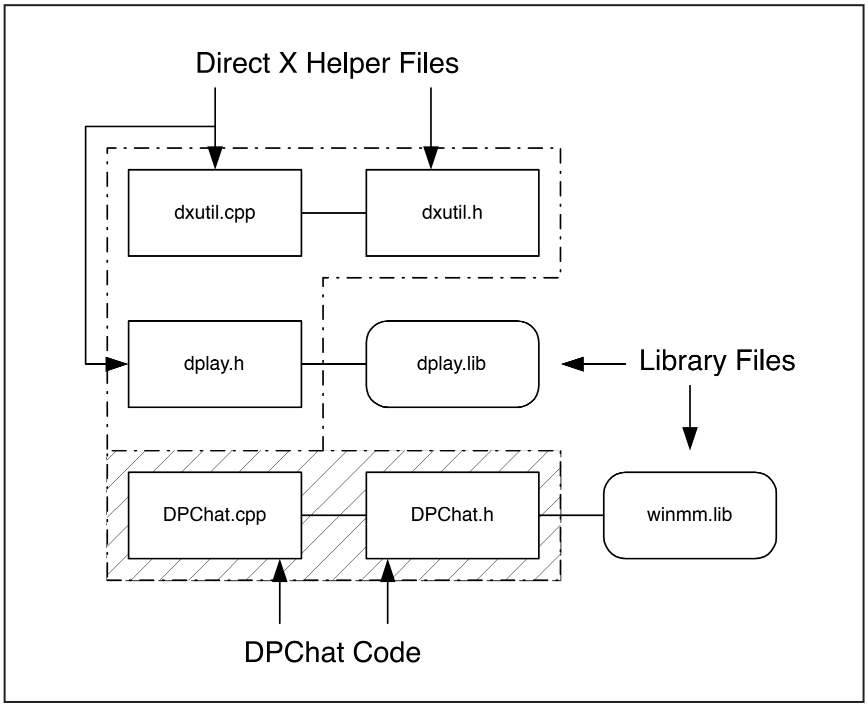


Figure 9.1

DirectPlay project architecture

TIP

The dxutil.cpp file requires a dxutil.h to work properly. You might find that it is easier to copy these two files to your project's folder than it is to keep track of where they are. If you decide not to copy the files, you should make sure your paths in Visual C++ point to the samples/multimedia/common/src and samples/multimedia/common/include folders. This will allow the compiler to find them in the future.

The DPChat Header File

The DPChat header file will contain all of your global variable declarations, function prototypes, and preprocessor definitions. By separating the project into two files, you can keep the clutter down. Here is the code that you need to copy into the DPChat header file. Or, you can just look at the one I wrote.

```
#define INITGUID
#include <windows.h>
#include <dplay8.h>
#include <stdio.h>
#include "DXUtil.h"
#define IDI_ICON1                           101    // Icon Resource
// IDs for child windows
#define    IDC_hBU_Join                     40001
#define    IDC_hLB_Output                   40002
#define    IDC_hBU_Host                     40004
#define    IDC_hEB_InputServerIP            40005
#define    IDC_hEB_InputServerPort          40006
#define    IDC_hEB_InputName                40007
#define    IDC_hST_TextServerIP             40008
#define    IDC_hST_TextServerPort           40009
#define    IDC_hST_TextName                 40010
#define    IDC_hEB_InputField               40011
// Handles to child windows
HWND        hBU_Join                = NULL;
HWND        hBU_Host                = NULL;
HWND        hLB_Output              = NULL;
HWND        hEB_InputServerIP        = NULL;
HWND        hEB_InputServerPort      = NULL;
HWND        hEB_InputName            = NULL;
HWND        hEB_InputField           = NULL;
HWND        hST_TextServerIP         = NULL;
HWND        hST_TextServerPort       = NULL;
HWND        hST_TextName             = NULL;
#define TIMERID_CONNECT_COMPLETE    1    // Timer ID
// GUID, needed to identify application
GUID DP_CHAT = { 0x2ae835d, 0x9179, 0x485f, { 0x83, 0x43, 0x90, 0x1d,
 0x32, 0x7c, 0xe7, 0x94 } };
// Class name and app name
LPCTSTR                    lpszApplicationName        = "DPChat";
LPCTSTR                    lpszTitle                  = "DPChat";
IDirectPlay8Peer*          g_pDP;
IDirectPlay8Address*       g_pDeviceAddress;
IDirectPlay8Address*       g_pHostAddress;
DPNHANDLE                  g_hConnectAsyncOp;
DPNID                      g_dpnidLocalPlayer = 0;
#define MAX_PLAYERS        4
struct PLAYER_INFORMATION
{
            bool       bActive;
            DPNID    dpnidPlayer;
            char       szPlayerName[32];
};
PLAYER_INFORMATION       PlayerInfo[MAX_PLAYERS];
HANDLE                   g_hConnectCompleteEvent;
HRESULT                  g_hrConnectComplete;
CRITICAL_SECTION         g_csModifyPlayer;
LONG                     g_lNumberOfActivePlayers = 0;
BOOL                     bHost = 0;
struct PACKET_CHAT {
            char       szText[256];
};
// Message Loop CallBack Function prototype ( REQUIRED FOR ALL WINDOWS PROGRAMS )
LRESULT CALLBACK fnMessageProcessor (HWND, UINT, WPARAM, LPARAM);
// Function to display text in the output box
void vShowText(HWND     hChildHandle, char *szText);
```

```
// Function to set up display of GUI
void vCreateInterface(HWND hWnd,HINSTANCE hInstance);
// Function called when program exits, cleans up allocated objects
void vCleanup(void);
// Function to initialize DirectPlay system
HRESULT hrInitializeDirectPlay( HWND hWindow );
// Function to send chat message
HRESULT            hrSendChatMessage(int player, char         *szmessage);
// Function to handle all incoming DPlay messages
HRESULT WINAPI DirectPlayMessageHandler( PVOID pvUserContext,
 DWORD dwMessageId, PVOID pMsgBuffer );
HRESULT        hrHostGame( HWND hwnd );        // Function to host a game
HRESULT        hrJoinGame( HWND hwnd );        // Function to join a game
// Function called when a player joins the game
HRESULT        hrCreatePlayer( PVOID pvUserContext, PVOID pMsgBuffer );
// Function called when a player leaves the game
HRESULT        hrDestroyPlayer( PVOID pvUserContext, PVOID pMsgBuffer );
```

Wow! Doesn't that look like a lot of information to digest? If you cut out all of the comments, then it shrink a bit, and if you remove all of the Windows GUI data, it shrinks another 20 lines or so. You will notice that I did cut out quite a few of the comments from the version on the book's CD, but that is to keep paper usage down. Be sure to save the header file once you have typed in everything.

INITGUID

The first thing accomplished in the header file is the definition of `INITGUID`. This definition sets up the environment to utilize GUIDs that are needed later. Be sure to specify this definition at the beginning of all DirectX applications.

DirectPlay Include Files

Once `INITGUID` is defined, I then go on to add the necessary header files to the application. Here they are as listed.

```
#include <windows.h>
#include <dplay8.h>
#include "DXUtil.h"
#include <stdio.h>
```

The first one, windows.h, sets up the application to use Windows macros and such. This is standard for all Windows applications I demonstrate. The second include file, dplay8.h, is required to use DirectPlay. None of the API's functions work without it. The third include file, dxutil.h, is the header file for the dxutil.cpp file I added in the project space. The dxutil files provide several helper functions to aid in common or mundane tasks. Why reinvent the wheel, right? The last include file is for standard I/O functions, which you should already recognize.

Child Windows

The next block of code in the header file defines the child window identifiers. Table 9.1 shows you the IDs for the controls required by the GUI.

The next block of code in the header file allocates variables for the child window handles.

Table 9.1 Child Window IDs Used in DPChat

Control	Description
`IDC_hBU_Join`	Button used to join the session
`IDC_hLB_Output`	List box to output text
`IDC_hBU_Host`	Button used to host the session
`IDC_hEB_InputServerIP`	Edit box to input host's name
`IDC_hEB_InputServerPort`	Edit box to input host's port
`IDC_hEB_InputName`	Edit box to input player name
`IDC_hST_TextServerIP`	Static text
`IDC_hST_TextServerPort`	Static text
`IDC_hST_TextName`	Static text
`IDC_hEB_InputField`	Edit box to accept chat text

The Timer ID

The next piece of code sets the value for a Windows timer you need when establishing a connection.

```
#define TIMERID_CONNECT_COMPLETE    1
```

You need a unique number for every timer you create in your application. Later, I'll show you how to use the `SetTimer()` function that uses this value. For now, just keep it in the back of your noggin.

The Ominous GUID

The next piece of code you see is probably one of the most important pieces to DirectPlay architecture. Without it, you are not able to communicate with other peers.

```
// GUID, needed to identify application
GUID DP_CHAT = { 0x2ae835d, 0x9179, 0x485f, { 0x83, 0x43, 0x90,
 0x1d, 0x32, 0x7c, 0xe7, 0x94 } };
```

What this little baby does is give the application its own unique identifier. For the program to find other instances of it, it must know which GUID to look for. This keeps games from cross talking.

NOTE

The guidgen program in Windows is used to create the unique GUIDs. You can run it from the command prompt.

The Class and Application Names

The next couple of lines of code create two global strings that hold the application name and the class name. If you put them here, you can change the name and title of the program without having to search all over the place.

The DirectPlay Interface Globals

Here come the important things—the interface globals. The code, please, maestro.

```
IDirectPlay8Peer*         g_pDP;
IDirectPlay8Address*      g_pDeviceAddress;
IDirectPlay8Address*      g_pHostAddress;
```

The first global listed here, g_pDP, is of type IDirectPlay8Peer and provides you with your connection to the DirectPlay world. This interface is used in both joining and hosting.

The second and third global interfaces allocated are of type IDirectPlay8Address and are your placeholders for the host and device address interfaces. These are used to establish connections.

The Player Structure and Other Miscellaneous Information

The rest of the header file contains function prototype and other information that you will find out about in the main .cpp file. The only noteworthy information at this time is the player data structure.

```
struct PLAYER_INFORMATION
{
            bool        bActive;
            DPNID       dpnidPlayer;
            char        szPlayerName[32];
};
```

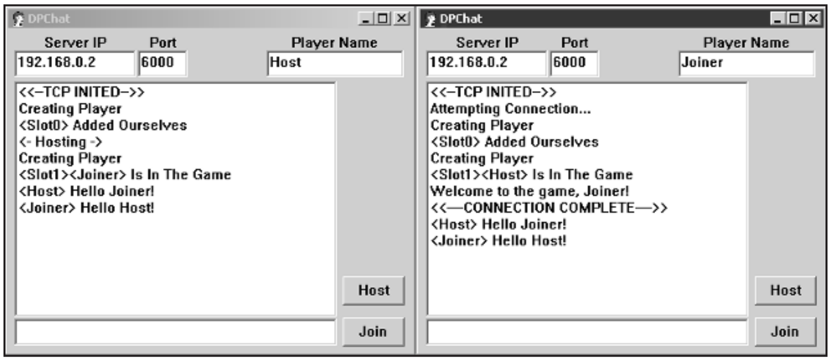
This structure defines the player information for each person who connects to the chat session. The first member sets whether the slot is being used. The second member holds the player's unique DirectPlay identifier. The third member holds the player's name. Right after the structure definition, I create holders for the data like so:

```
PLAYER_INFORMATION      PlayerInfo[MAX_PLAYERS];
```

That's it for the header file; on with the show!

The Main Project File

The main file you deal with is the DPChat.cpp file. This file contains the majority of the code used in the application. If you are creating your own project from scratch, then you need to copy the contents of my DPChat.cpp file to your own. There is just way too much code to list here. What I do is list it piece-by-piece and explain the functions in the order they appear. Before I get ahead of myself, take a look at Figure 9.2 to see the output of the program.



```
192.168.0.2   6000   Host
<<–TCP INITED–>>
Creating Player
<Slot0> Added Ourselves
<- Hosting ->
Creating Player
<Slot1><Joiner> Is In The Game
<Host> Hello Joiner!
<Joiner> Hello Host!
```

```
192.168.0.2   6000   Joiner
<<–TCP INITED–>>
Attempting Connection...
Creating Player
<Slot0> Added Ourselves
Creating Player
<Slot1><Host> Is In The Game
Welcome to the game, Joiner!
<<—CONNECTION COMPLETE—>>
<Host> Hello Joiner!
<Joiner> Hello Host!
```

Figure 9.2

Output from DPChat.exe

Window Initialization

The first line of code you run across is used to include the DPChat.h header file. This gives you access to everything defined in it. Once that is out of the way, then the `WinMain()` function follows. Here it is in its entirety.

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
 int nCmdShow)
{
    MSG                 msg;
    WNDCLASSEX          wndclass;
    HWND                hWnd;
    HRESULT             hReturn;

    // Set up window attributes
    wndclass.cbSize                 = sizeof(wndclass);
    wndclass.style                  = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc            = fnMessageProcessor;
    wndclass.cbClsExtra             = 0;
    wndclass.cbWndExtra             = 0;
    wndclass.hInstance              = hInstance;
    wndclass.hIcon                  = LoadIcon( hInstance,
                                             MAKEINTRESOURCE(IDI_ICON1) );
    wndclass.hCursor                = LoadCursor( NULL, IDC_ARROW );
    wndclass.hbrBackground          = (HBRUSH)(COLOR_WINDOW);
    wndclass.lpszMenuName           = NULL;
    wndclass.lpszClassName          = lpszApplicationName;
                                             // Registered Class Name
    wndclass.hIconSm                = LoadIcon( hInstance,
                                                      MAKEINTRESOURCE(IDI_ICON1) );

    if( RegisterClassEx( &wndclass ) == 0 ) {
            // Do error logic here
            exit(1);
    }

    // Create the main window
    hWnd = CreateWindow(
                        lpszApplicationName,  // Application name
                        lpszTitle,       // Name displayed on title bar
                        WS_OVERLAPPEDWINDOW,
                        100,100,400,340,NULL,NULL,hInstance,NULL );
    // Create child objects
    vCreateInterface(hWnd,hInstance);
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    // Initialize DirectPlay
    hReturn = hrInitializeDirectPlay( hWnd );
    if( FAILED(hReturn) ) {
            vCleanup();
            exit(1);
    }
    // Process messages until the program is terminated
    while( GetMessage ( &msg, NULL, 0, 0 ) )
    {
            TranslateMessage( &msg );
            DispatchMessage( &msg );
    }
    return(msg.wParam);
}
```

There should be no real surprises here. Just about everything in the function resembles the code written in Chapter 7, "Practical Windows Programming." The first thing I do is create the window class object and its attributes. The class is then registered. After that is done, I create the main window using the `CreateWindow()` function.

One item of note is that I am not using `PeekMessage()` here to loop through messages. Instead, the program is sitting in the while loop until a Windows message happens along. This keeps the program from taking up too much processing power during idle times. Refer to Figure 9.3 to see this concept illustrated.

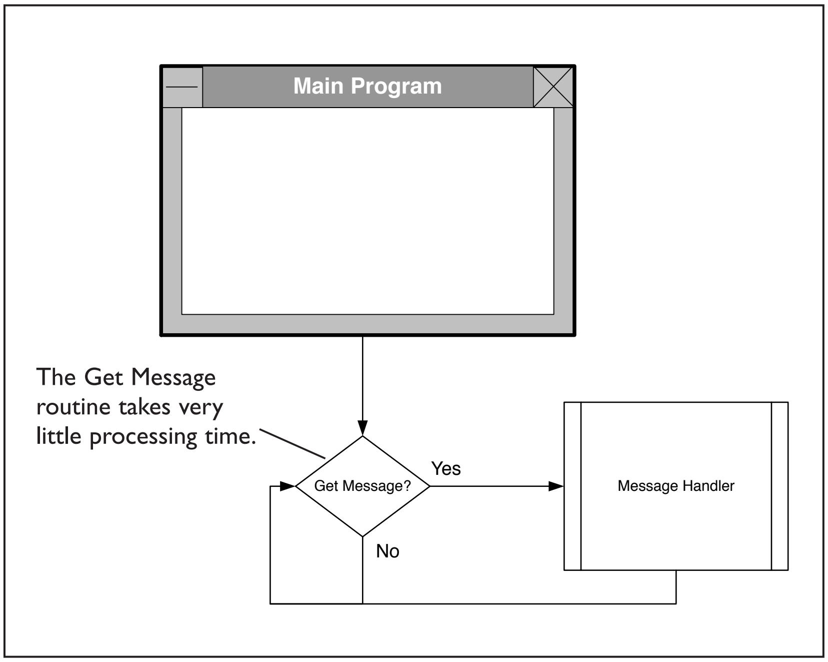


Figure 9.3

The message loop

Initializing Child Windows

The next function called is `vCreateInterface()`. Its sole purpose is to create the child window objects in the main window. Things such as the edit boxes and static text objects are created there. Here is the code listing for it.

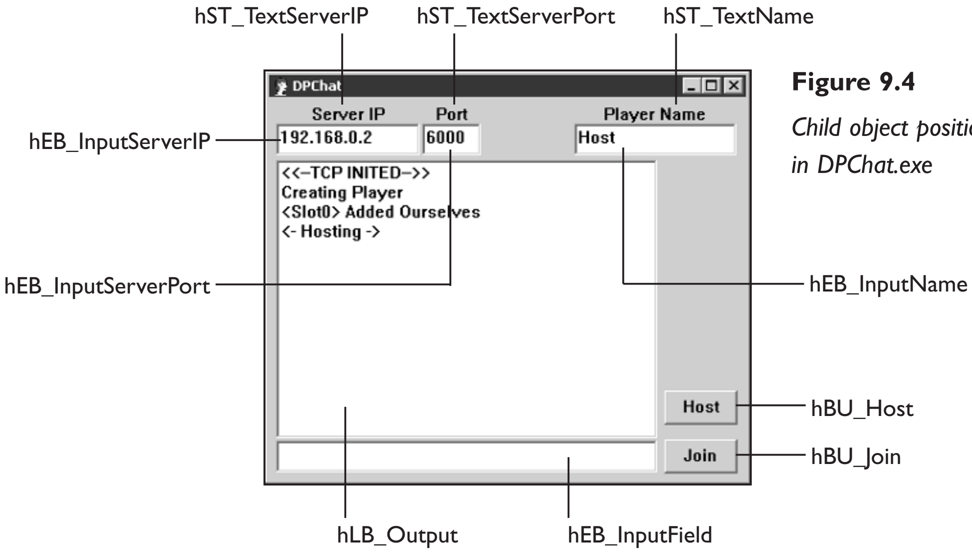
```
void vCreateInterface(HWND hWnd,HINSTANCE hInstance)
{
     // Server IP text
     hST_TextServerIP = CreateWindow(
               "static","Server IP",WS_CHILD | SS_CENTER |
                WS_VISIBLE,5,5,120,28,
               hWnd,(HMENU)IDC_hST_TextServerIP,hInstance,NULL);
     // Port Text
     hST_TextServerPort = CreateWindow(
               "static","Port",WS_CHILD | SS_CENTER |
                WS_VISIBLE,125,5,50,28,
               hWnd,(HMENU)IDC_hST_TextServerPort,hInstance,NULL);
     // Name text
     hST_TextName = CreateWindow(
                    "static","Player Name",WS_CHILD | SS_CENTER |
                     WS_VISIBLE,250,5,135,28,
                    hWnd,(HMENU)IDC_hST_TextName,hInstance,NULL);
        // Join button
        hBU_Join = CreateWindow(
                    "BUTTON","Join",WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
                    325,280,60,28,
                    hWnd,(HMENU)IDC_hBU_Join,hInstance,NULL);
        // Host button
        hBU_Join = CreateWindow(
                    "BUTTON","Host",WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
                    325,240,60,28,hWnd,(HMENU)IDC_hBU_Host,hInstance,NULL);
        // Output window
        hLB_Output = CreateWindowEx(
                    WS_EX_CLIENTEDGE,"LISTBOX",NULL,
                    WS_CHILD | WS_VISIBLE | LBS_NOTIFY | WS_VSCROLL | WS_BORDER,
                    5,50,315,230,hWnd,(HMENU)IDC_hLB_Output,hInstance,NULL);
        // Name
        hEB_InputName = CreateWindowEx(
                    WS_EX_CLIENTEDGE,"EDIT","Unnamed",
                    WS_CHILD | WS_VISIBLE | WS_BORDER | ES_LEFT,
                    250,20,135,28,hWnd,(HMENU)IDC_hEB_InputName,hInstance,NULL);
        // Server IP
        hEB_InputServerIP = CreateWindowEx(
                    WS_EX_CLIENTEDGE,"EDIT","192.168.0.2",
                    WS_CHILD | WS_VISIBLE | WS_BORDER | ES_LEFT,
                    5,20,120,28,hWnd,(
                     HMENU)IDC_hEB_InputServerIP,hInstance,NULL);
        // Server port
        hEB_InputServerPort = CreateWindowEx(
                    WS_EX_CLIENTEDGE,"EDIT","6000",
                    WS_CHILD | WS_VISIBLE | WS_BORDER | ES_LEFT,
                    125,20,50,28,hWnd,(
                     HMENU)IDC_hEB_InputServerPort,hInstance,NULL);
        // Edit field to enter chat messages into
        hEB_InputField = CreateWindowEx(
                    WS_EX_CLIENTEDGE,"EDIT",NULL,
                    WS_CHILD | WS_VISIBLE | WS_BORDER | ES_LEFT |
                     ES_AUTOVSCROLL | ES_MULTILINE,
                    5,280,315,28,hWnd,(HMENU)IDC_hEB_InputField,hInstance,NULL);
    }
```

Each of the child objects shown creates the GUI for the chat application. The one you will see mentioned many times is hLB_Output. This handle is used quite often when outputting text to the main window. Take a look at Figure 9.4 to see where each of the child objects belong.

Outputting Text to the Window

The function used to output text to the hLB_Output list box is called vShowText(). This function accepts two parameters. The first parameter accepts the child handle to output the text to, and the second parameter accepts the text to output. Because the application has only one list box, vShowText() is always passed hLB_Output for the child handle. Here is the code for this function.

```
void vShowText(HWND     hChildHandle, char *szText)
{
    int Line;
    // Add string to the listbox
    SendMessage(hChildHandle,LB_ADDSTRING,0,(LPARAM)szText);
    // Determine number of items in listbox
    Line = SendMessage(hChildHandle,LB_GETCOUNT,0,0);
    // Flag last item as the selected item, to scroll listbox down
    SendMessage(hChildHandle,LB_SETCURSEL,Line-1,0);
    // Unflag all items to eliminate negative highlight
    SendMessage(hChildHandle,LB_SETCURSEL,-1,0);
}
```



```
<<–TCP INITED–>>
Creating Player
<Slot0> Added Ourselves
<- Hosting ->
```

Figure 9.4

Child object positions in DPChat.exe

The Window Message Handler

Back in WinMain(), I set the function named fnMessageProcessor() to be the window class's message handler. This handler is needed to process all messages to the window, such as key presses. Here is the code to the message handler.

```
LRESULT CALLBACK fnMessageProcessor ( HWND hWnd, UINT iMsg, WPARAM wParam,
 LPARAM lParam )
{
    char          szMessage[256];
    char          szCompMessage[256];
    HRESULT       hReturn;
    switch (iMsg) {
        case WM_COMMAND:
        {
            // Check for child window messages
            switch(LOWORD(wParam))
                // Check whether the user clicked the button
                case IDC_hBU_Join:
                    hrJoinGame( hWnd );
                    break;
                case IDC_hBU_Host:
                    hrHostGame( hWnd );
                    break;
            }
            switch(HIWORD(wParam)) {
                case EN_UPDATE:
                    // Get the text from the edit box
                    GetWindowText(hEB_InputField,szMessage,256);
                    // Check whether she pressed Enter
                    if( szMessage[strlen(szMessage)-1] == 10 ) {
                        // Get rid of trailing garbage
                        szMessage[strlen(szMessage)-2] = '\0';
                        sprintf(szCompMessage,"<%s> %s",
                        PlayerInfo[0].szPlayerName, szMessage);
                        hReturn = hrSendChatMessage(-1,szCompMessage);
                        // Clear input field
                        SetWindowText(hEB_InputField,"");
                }
                }
                break;
            }
        case WM_DESTROY:
            // Clean up system
            vCleanup();
            // Exit out of Windows
            PostQuitMessage(0);
            break;
        case WM_TIMER:
            if( wParam == TIMERID_CONNECT_COMPLETE ) {
                // Check whether the message is telling us
                   our connection is complete
                if( WAIT_OBJECT_0 == WaitForSingleObject(
                g_hConnectCompleteEvent, 0 ) ) {
                    if( FAILED( g_hrConnectComplete ) ) {
                        vShowText(hLB_Output,"<<----CONNECTION IN-COMPLETE---->>");
                    }
                    else {
                        vShowText(hLB_Output,"<<----CONNECTION COMPLETE---->>");
                    }
                    KillTimer( hWnd, TIMERID_CONNECT_COMPLETE );
                }
            }
            break;
        default:
            return(DefWindowProc(hWnd, iMsg, wParam, lParam));
    }
    return(0L);
}
```

The main functions of interest here are `hrJoinGame()`, `hrHostGame()`, and `hrSendChatMessage()`. These three functions get called when their corresponding child controls are activated. The join and host functions are activated by their corresponding buttons. The chat function is activated when the user presses Enter after typing text into the edit box.

Another important item in this function is the `WM_TIMER` check. That is used later to determine when a connection has been made. Check out Figure 9.5 to see the architecture of what has been completed so far.

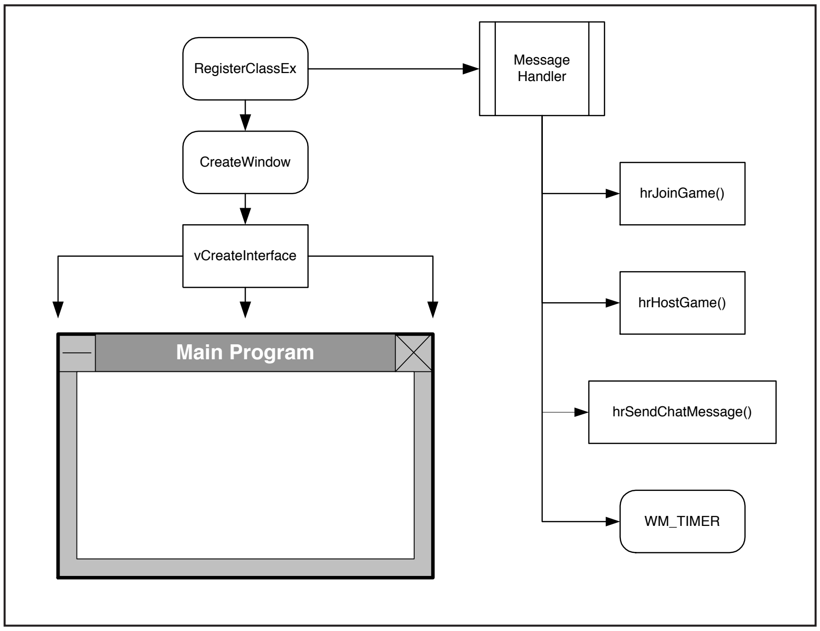


Figure 9.5

The program architecture so far

The hrInitializeDirectPlay() Function

Up to now, all I have done is create Windows GUI code. Now comes the good part, DirectPlay goodies. In the `WinMain()` function, I make a call to `hrInitializeDirectPlay()`. If the call fails, then the program exits. Here is the code for the function.

```
HRESULT hrInitializeDirectPlay( HWND hWindow )
{
    HRESULT      hReturn;
    int          i;

    // Initialize COM
    hReturn = CoInitialize( NULL );
    if( FAILED(hReturn) ) {
        MessageBox( hWindow, "Error Initializing COM",
         "DirectPlay Error", MB_ICONERROR );
        return hReturn;
    }
```

```
// Initialize critical sections for multithreading
InitializeCriticalSection( &g_csCreatePlayer );
InitializeCriticalSection( &g_csDestroyPlayer );

// Create IDirectPlay8Peer object
if( FAILED( hReturn = CoCreateInstance( CLSID_DirectPlay8Peer,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_IDirectPlay8Peer,
    (LPVOID*) &g_pDP ) ) )
    MessageBox( hWindow, "Can't Create DPlayPeer",
     "DirectPlay Error", MB_ICONERROR );

// Init IDirectPlay8Peer message handler
if( FAILED( hReturn = g_pDP->Initialize( NULL,
 DirectPlayMessageHandler, 0 ) ) ) {
    MessageBox( hWindow, "Failed to Message Handler",
     "DirectPlay Error", MB_ICONERROR );
    return -1;
}

// Create a device address
hReturn = CoCreateInstance( CLSID_DirectPlay8Address, NULL,CLSCTX_INPROC_SERVER,
 IID_IDirectPlay8Address, (LPVOID*) &g_pDeviceAddress );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Failed to Create Device",
     "CoCreateInstance()", MB_ICONERROR );
    return -1;
}

// Set our service provider to TCP/IP
if( FAILED( hReturn = g_pDeviceAddress->SetSP(
  &CLSID_DP8SP_TCPIP ) ) ) {
    MessageBox( hWindow, "Failed to SetSP() for Device Address",
     "Invalid Param", MB_ICONERROR );
    return -1;
}

// Create a host address
hReturn = CoCreateInstance( CLSID_DirectPlay8Address,
 NULL,CLSCTX_INPROC_SERVER, IID_IDirectPlay8Address,
     (LPVOID*) &g_pHostAddress );
    if( FAILED(hReturn) ) {
        MessageBox( hWindow, "Failed to Create Host Address()",
         "Invalid Param", MB_ICONERROR );
        return -1;
    }
    // Set the host address to TCP/IP
    if( FAILED( hReturn = g_pHostAddress->SetSP(
     &CLSID_DP8SP_TCPIP ) ) ) {
        MessageBox( hWindow, "Failed to SetSP() for Host Address",
         "Invalid Param", MB_ICONERROR );
        return -1;
    }

    // Create connection complete event for later use
    g_hConnectCompleteEvent = CreateEvent( NULL, FALSE, FALSE, NULL );

    vShowText(hLB_Output,"<<--TCP INITED-->>");

    // Init miscellaneous variables
    for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
        PlayerInfo[i].bActive = 0;
    }

    return S_OK;
}
```

That's a bit of a doozy, eh? I hope some of it looks familiar from Chapter 8, "What the Heck Is DirectPlay Anyway?" This function is responsible for initializing all of the common code used in the chat application. Regardless of whether the instance of the program you are running is the host, it calls this code.

Thou Shalt Initialize COM

It's not a commandment, but it is required for DirectPlay to work. The first thing I do is initialize COM so that I can use the COM interfaces provided by DirectX. This is accomplished using the infamous `CoInitialize()` function.

```
hReturn = CoInitialize( NULL );
```

As long as it returns okay, the program can proceed.

Critical Sections

The next two lines of code should look a little foreign to you if you are not a multithreading junkie like me.

```
// Initialize critical sections for multithreading
InitializeCriticalSection( &g_csModifyPlayer );
```

Do you remember mutexes back at the end of Chapter 6, "Introduction to Windows Programming?" If not, please return to that chapter and review them. Basically, a critical section is just like a mutex.

A critical section is a synchronization object that allows only one thread at a time to access a block of data. It is used to keep two or more threads from corrupting shared memory. There are many examples of where this is useful, one of which is the coin counter scenario.

Imagine that your application ran a worker thread that did nothing other than check the coin slot for coin drops. Every time a coin dropped through, the worker thread would add one credit to the global counter, called `g_credits`. The main program that accepts user input would wait for the user to select the Player One button. When this happened, the main program would deduct a credit from the `g_credits` counter.

The problem here is, what happens if a credit is deposited at the same time the user is selecting the Player One button? You could have a possible conflict of operations because both threads would hit the same global variable. Well, if you introduce a critical section around adding and subtracting the credit variable, you solve the problem. The critical section would restrict access to modifying `g_credits` to only one thread at a time. If the critical section were in use by the worker thread when the user selected the Player One button, then the main thread would have to wait to deduct a coin until the worker thread released the critical section. No matter how many times I read this, it still sounds confusing to me, so here's an illustration to help. Take a look at Figure 9.6 to see critical sections illustrated.

Now that you understand critical sections, I can go ahead and explain the critical section I am initializing. The chat program keeps track of a list of active participants. When a player joins the chat room, all players, including the joiner, receive a message. When a new player message is received, the chat program calls the `hrCreatePlayer()` function. Within this function, a global list is modified to add the player. Because different threads can call this function at the same time, you must lock the globally shared data when modifying it, using the `g_csModifyPlayer` critical section.

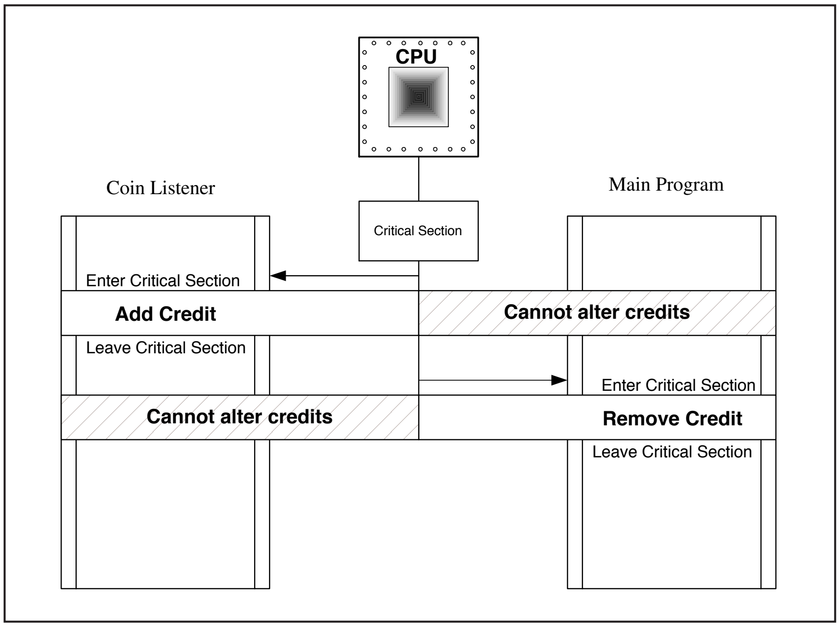


Figure 9.6

A critical section used in coin counting

The same logic applies when players leave the game. When this happens, the `hrDestroyPlayer()` function is called, and it also uses the `g_csModifyPlayer` critical section to lock the global data when removing a player from the list. This keeps `hrCreatePlayer()` from interfering with `hrDestroyPlayer()`. You will see how to use the critical section later.

The IDirectPlay8Peer Object

The next block of code that comes along gets an interface to the DirectPlay peer-to-peer functionality.

```
// Create IDirectPlay8Peer Object
if( FAILED( hReturn = CoCreateInstance( CLSID_DirectPlay8Peer,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_IDirectPlay8Peer,
    (LPVOID*) &g_pDP ) ) )
MessageBox( hWindow, "Can't Create DPlayPeer", "DirectPlay Error", MB_ICONERROR );
```

Here I make a call to `CoCreateInstance()` with the class `CLSID_DirectPlay8Peer`. This loads my pointer called `g_pDP` with the `IDirectPlay8Peer` interface desired. Once this is accomplished, I can go on to make peer interface calls.

The DirectPlay Message Handler

Now that the interface is created, it is time to assign a message handler to hold all of the DirectPlay messages the application will receive. Here is the code to initialize the message handler.

```
// Init IDirectPlay8Peer Message Handler
if( FAILED( hReturn = g_pDP->Initialize( NULL, DirectPlayMessageHandler, 0 ) ) ) {
    MessageBox( hWindow, "Failed to Message Handler", "DirectPlay Error",
     MB_ICONERROR );
    return -1;
}
```

For the initialization, I use the `IDirectPlay8Peer::Initialize()` function. In the second parameter, I pass it the address for the message handler function to use. In this case, it will use the function named `DirectPlayMessageHandler()`. Keep in mind that all messages received from now on with the `g_pDP` interface are routed to this function. Here is the code for the message handler function.

```
HRESULT WINAPI DirectPlayMessageHandler( PVOID pvUserContext,
 DWORD dwMessageId, PVOID pMsgBuffer )
{
    HRESULT     hReturn = S_OK;
    switch( dwMessageId ) {
        case DPN_MSGID_CREATE_PLAYER:
        {
            hrCreatePlayer(pvUserContext,pMsgBuffer);
            break;
        }

        case DPN_MSGID_DESTROY_PLAYER:
        {
            hrDestroyPlayer(pvUserContext,pMsgBuffer);
            break;
        }

        case DPN_MSGID_HOST_MIGRATE:
        {
            vShowText(hLB_Output,"Migrate Host");
            PDPNMSG_HOST_MIGRATE pHostMigrateMsg;
            pHostMigrateMsg = (PDPNMSG_HOST_MIGRATE)pMsgBuffer;

            // Check to see whether we are the new host
```

```
                if( pHostMigrateMsg->dpnidNewHost == g_dpnidLocalPlayer ) {
                    vShowText(hLB_Output,"(HOSTING)");
                }
                break;
            }

            case DPN_MSGID_TERMINATE_SESSION:
            {
                vShowText(hLB_Output,"<-Terminating Session->");
                PDPNMSG_TERMINATE_SESSION pTerminateSessionMsg;
                pTerminateSessionMsg =
                (PDPNMSG_TERMINATE_SESSION)pMsgBuffer;
                break;
            }

            case DPN_MSGID_RECEIVE:
            {
                PDPNMSG_RECEIVE pReceiveMsg;
                pReceiveMsg = (PDPNMSG_RECEIVE)pMsgBuffer;

                vShowText(hLB_Output,(char*)pReceiveMsg->pReceiveData);

                break;
            }

            case DPN_MSGID_CONNECT_COMPLETE:
            {
                PDPNMSG_CONNECT_COMPLETE pConnectCompleteMsg;
                pConnectCompleteMsg = (PDPNMSG_CONNECT_COMPLETE)pMsgBuffer;

                g_hrConnectComplete = pConnectCompleteMsg->hResultCode;
                SetEvent( g_hConnectCompleteEvent );
                break;
            }
        }
        return hReturn;
}
```

All of the messages the application handles appear in this function. I already briefly mentioned DPN_MSGID_CREATE_PLAYER and DPN_MSGID_DESTROY_PLAYER when I talked about critical sections.

The DPN_MSGID_HOST_MIGRATE message is received by whichever peer is picked to take over when a host ends the session. This is only valid if the session has migration turned on. The DPN_MSGID_TERMINATE_SESSION message is received if migration is not active and the host terminates the session. In this case, everyone is disconnected. The DPN_MSGID_RECEIVE message is received whenever a chat message is sent. This message type is the one used by all games to send custom messages. The DPN_MSGID_CONNECT_COMPLETE message signals that a connection has been completed. Any player who joins a session successfully receives this message.

Take a look at Figure 9.7 to see which messages trigger which functions in DPChat.exe.

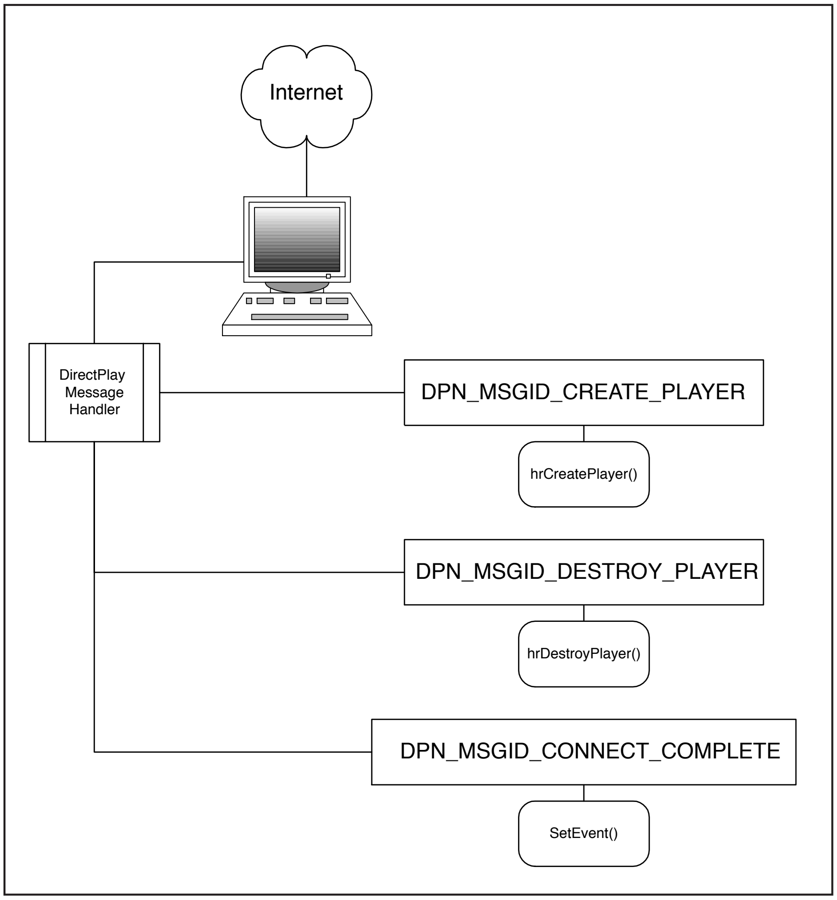


Figure 9.7

Messages and their corresponding functions

The Host and Device Addresses

Because every peer game requires both a host and a device address, I create those next, using the `CLSID_DirectPlay8Address` COM object, as shown here.

```
// Create a device address
hReturn = CoCreateInstance( CLSID_DirectPlay8Address,
 NULL,CLSCTX_INPROC_SERVER, IID_IDirectPlay8Address,
 (LPVOID*) &g_pDeviceAddress );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Failed to Create Device", "CoCreateInstance()",
     MB_ICONERROR );
    return -1;
}
// Set our service provider to TCP/IP
if( FAILED( hReturn = g_pDeviceAddress->SetSP( &CLSID_DP8SP_TCPIP ) ) ) {
    MessageBox( hWindow, "Failed to SetSP() for Address",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
// Create a host address
hReturn = CoCreateInstance( CLSID_DirectPlay8Address,
 NULL,CLSCTX_INPROC_SERVER, IID_IDirectPlay8Address,
 (LPVOID*) &g_pHostAddress );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Failed to Create Host Address()",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
// Set the host address to TCP/IP
if( FAILED( hReturn = g_pHostAddress->SetSP( &CLSID_DP8SP_TCPIP ) ) ) {
    MessageBox( hWindow, "Failed to SetSP() for Address",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
```

The `CoCreateInstance()` function creates both of the addresses, and then the `IDirectPlay8Address::SetSP()` function sets their service providers to use TCP/IP. This operation does not involve a whole lot of fanfare; it's just another gear in the machine, so to speak.

The Completion Event

Yippee! Here is something new for you, a Windows event. Check out this piece of code.

```
// Create connection complete event for later use
g_hConnectCompleteEvent = CreateEvent( NULL, FALSE, FALSE, NULL );
```

Here I am triggering a Windows event to use later with the `CreateEvent()` function. Here is the prototype.

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes,
    BOOL bManualReset,
    BOOL bInitialState,
    LPCTSTR lpName
    );
```

The first parameter can contain security attributes if the event can be inherited by child processes. Because this application doesn't need this functionality, I set this to `NULL`. The second parameter accepts a `TRUE` or `FALSE`, which tells the event whether it must be reset using the `ResetEvent()` function. I want an auto-resetting event, so I set this to `FALSE`. The third parameter also accepts a `TRUE` or `FALSE` and tells the event whether to start out being signaled or nonsignaled. I set it to `FALSE` so that it starts out turned off.

TIP

Signaled can be considered *on*, and nonsignaled can be considered *off*.

The fourth parameter lets you name the event with a string name. I don't need to name it, so I set this to `NULL`.

The `g_hConnectCompleteEvent` handle is used later to signal the application that it has successfully established a connection to the host. Because the program is multithreaded, I have to use a thread-safe method for signaling. This is where the event comes into play. You will see shortly how it is used for joining.

The rest of the code left in `hrInitializeDirectPlay()` initializes the player information data. All I am doing here is setting each player slot to be inactive. As the application receives create player messages, the slots will be activated.

```
vShowText(hLB_Output,"<<--TCP INITED-->>");
// Init miscellaneous variables
for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
    PlayerInfo[i].bActive = 0;
}
return S_OK;
```

Last, the function returns an `S_OK`. Once this occurs, the `WinMain()` function goes on to enter its message loop to wait for messages. Look at Figure 9.8 to see the program's flow so far.

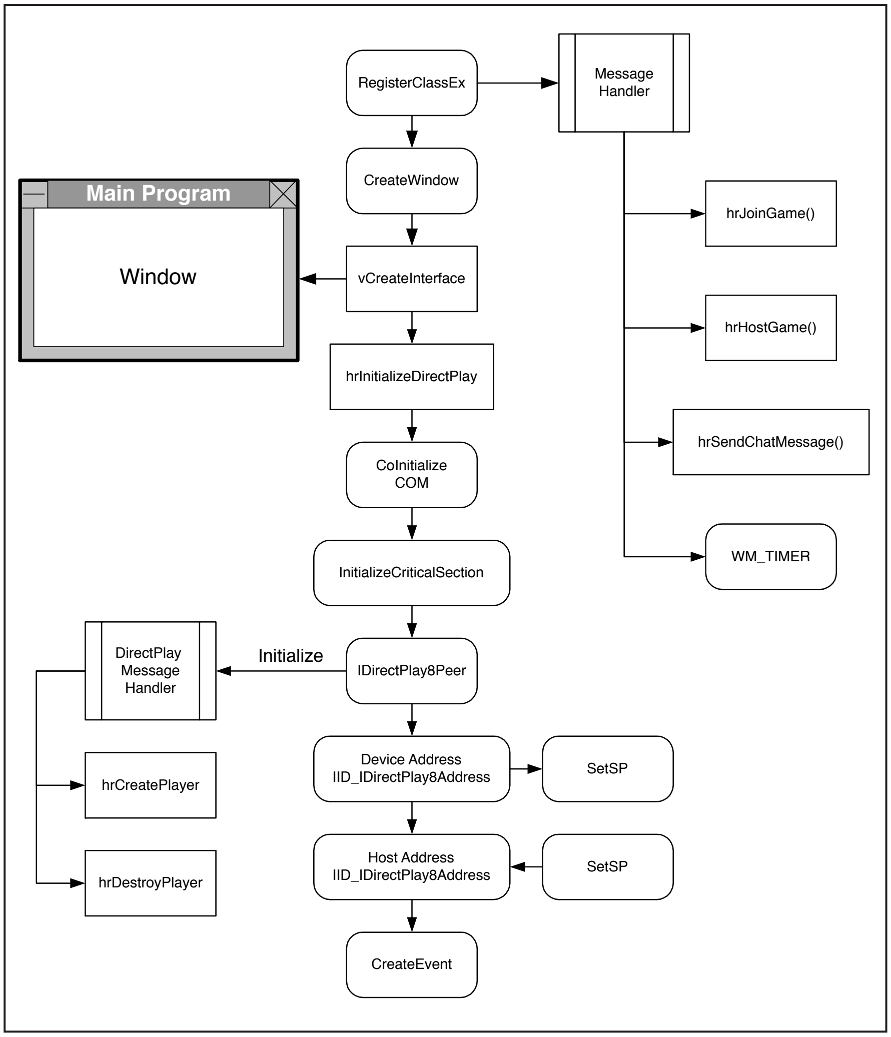


Figure 9.8

Program flow, including the initialization of DirectPlay

Now that the window is displayed and DirectPlay is active, it is time to do something. Because you need a host before anyone can join, go ahead and take a look at the host logic.

The hrHostGame() Function

When the user clicks the Host button, a `WM_COMMAND` message is sent to the Windows message handler. When the application checks the `LOWORD` of the parameter, it finds an `IDC_hBU_Host` value. This triggers a call to the `hrHostGame()` function. This logic appears in the already mentioned `fnMessageProcessor()` function. Check out Figure 9.9 to see this process in action.

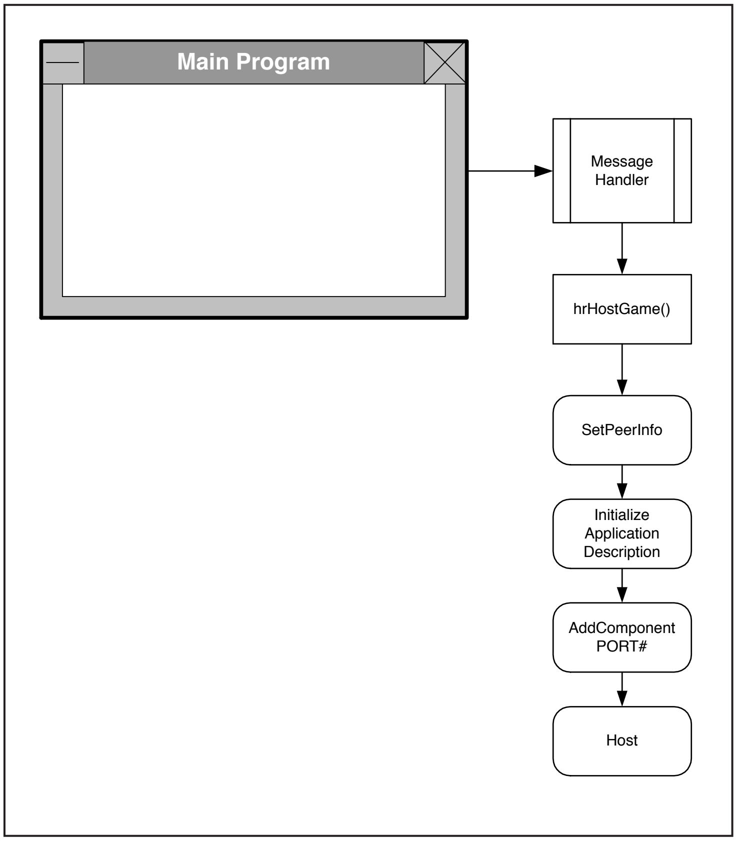


Figure 9.9

Program flow when the user clicks the Host button

Now that the hrHost() function is called, here is what the code looks like.

```
HRESULT     hrHostGame( HWND hWindow)
{
    HRESULT                                     hReturn;
    char                                        szPeerName[256];
    char                                        szSessionName[256];
    WCHAR                                       wszPeerName[256];
    WCHAR                                       wszSessionName[256];
    DPN_APPLICATION_DESC                        dnAppDesc;
    char                                        szPort[6];
    DWORD                                       dwLength = 256;
    DPN_PLAYER_INFO                             dpPlayerInfo;
    DWORD                                       dwPort = 9000;

    // Set up our player information
    GetWindowText(hEB_InputName,szPeerName,36);
    // Get name from window edit box
    DXUtil_ConvertGenericStringToWide( wszPeerName, szPeerName );
    ZeroMemory( &dpPlayerInfo, sizeof(DPN_PLAYER_INFO) );
    dpPlayerInfo.dwSize = sizeof(DPN_PLAYER_INFO);
    dpPlayerInfo.dwInfoFlags = DPNINFO_NAME;
    dpPlayerInfo.pwszName = wszPeerName;
    // Set us up to be non-asynchronous
    if( FAILED( hReturn = g_pDP->SetPeerInfo( &dpPlayerInfo, NULL, NULL,
     DPNSETPEERINFO_SYNC ) ) ) {
        MessageBox( hWindow, "Failed to SetPeerInfo()",
         "DirectPlay Error", MB_ICONERROR );
        return -1;
    }
    // Set up the application description
    sprintf(szSessionName,"%s's Game",szPeerName);
    DXUtil_ConvertGenericStringToWide( wszSessionName, szSessionName );
    ZeroMemory( &dnAppDesc, sizeof(DPN_APPLICATION_DESC) );
    dnAppDesc.dwSize                = sizeof(DPN_APPLICATION_DESC);
    dnAppDesc.guidApplication       = DP_CHAT;
    dnAppDesc.pwszSessionName       = wszSessionName;
    dnAppDesc.dwMaxPlayers          = MAX_PLAYERS;
    dnAppDesc.dwFlags               = DPNSESSION_MIGRATE_HOST;
    // Get port from edit box
    GetWindowText(hEB_InputServerPort,szPort,6);
    // Convert the port string to a DWORD
    dwPort = atol(szPort);
    // Add port number to address
    hReturn = g_pDeviceAddress-
    >AddComponent(DPNA_KEY_PORT,&dwPort,sizeof(
    DWORD),DPNA_DATATYPE_DWORD);
    if( hReturn != S_OK ) {
        MessageBox( hWindow, "Failed to AddComponent()", "hrHostGame()",
         MB_ICONERROR );
        return -1;
    }
    // Host the game
    hReturn = g_pDP->Host(      &dnAppDesc,
        &g_pDeviceAddress,
        1,
        NULL,
        NULL,
        NULL,
        NULL );
    if( FAILED( hReturn ) ) {
        MessageBox( hWindow, "Failed to Host()", "DirectPlay Error",
        MB_ICONERROR );
        return -1;
    }
    // Let us know we are the host
    bHost = 1;
    vShowText(hLB_Output,"<- Hosting ->");
    return hReturn;
}
```

Setting Up the Player Information

The first thing you have to do when you enter the host function is set up the player information for the host. Even though this player is hosting the game, it is still a player and must have information about itself. You do this by setting up the `DPN_PLAYER_INFO` data structure. If you are a bit fuzzy about this data structure, you can read about it in Chapter 8, "What the Heck Is DirectPlay Anyway?," in the "Setting Peer Information" section.

The main thing to note here is that when setting up player information in a peer-to-peer game, all you have to do is set the name of the player. Specifying `DPNINFO_NAME` as the value for the `dwInfoFlags` member variable does this. This tells DirectPlay to look for the player's name in the `pwszName` field.

Once the data structure is loaded, you set the peer information using the SetPeerInfo() member function. By passing the DPNSETPEERINFO_SYNC flag to this function, you specify that it operates synchronously. That's it for setting up the player information; now it's time to define the application.

TIP

Whenever you see the pwsz **Hungarian notation, it is referring to a wide-character string. You can use the DirectX helper function** DXUtil_ConvertGenericStringToWide() **to assist in conversions.**

Setting Up the Application Description

By setting up the application description, you are telling the program all about the application and what it does. For a refresher of this concept, refer to the section titled "Application Descriptions" in Chapter 8. The main thing to remember is that to set up the description, you use the DPN_APPLICATION_DESC data structure.

One item of note in the code I have shown you is the DPNSESSION_MIGRATE_HOST flag set under the dwFlags member variable. This tells the program that the host can be migrated when shut down. If you want to test out the application and not support migration, you can just set dwFlags to NULL.

Setting Up the Device Address

Because the host should specify a port number to accept connections, you must add the port number to the device address. You do this using my personal favorite, the AddComponent() member function. To add the port component, I passed the DPNA_KEY_PORT define to the function. This tells AddComponent() which component to add.

Calling the Host() Function

Once the address is defined, you can finally call the Host() member function of the IDirectPlay8Peer interface. For this wonderful function, I have passed the parameters listed in Table 9.2.

That's about all you have to do to start hosting. The program then sits and waits for connections once the Host()

TIP

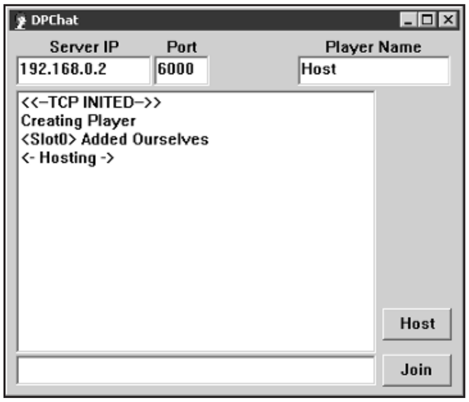
The defines for which components you can add to an address appear in the dpaddr.h include file that comes with the DirectX SDK. Make sure you have the directx8/include and directx8/lib paths set up in your compiler. If you do not, you might get File Not Found errors when compiling.

Table 9.2 Parameters Passed to the Host() Member Function

Parameter	Description
`&dnAppDesc`	The address of the application description structure
`&g_pDeviceAddress`	The host address that defines where other peers can connect
`1`	The number of host addresses contained in `&g_pDeviceAddress`
`NULL`	Reserved for later versions of DirectPlay
`NULL`	Reserved for later versions of DirectPlay
`NULL`	Optional player context information
`NULL`	Optional flags

function executes. Oh, yeah; the program sets the `bHost` variable to `True` so that it can remember it is hosting.

Okay, cool, the application now knows how to host. If you run the program and click the Host button, you should see a successful message. If it fails for some reason, be sure to check your network settings and try again. The output from the DPChat program in Figure 9.10 shows what the screen looks like when someone has hosted a game.



```
192.168.0.2   6000   Host
<<–TCP INITED–>>
Creating Player
<Slot0> Added Ourselves
<- Hosting ->
```

Figure 9.10

Output from DPChat once the player has clicked the Host button

The hrCreatePlayer() Function

As soon as the Host() function completes, you'll notice that the program receives a create player message. This message tells the program to create the host player in the game. Even though the player is the host, he is still a player. The message type received is DPN_MSGID_CREATE_PLAYER.

Take a look at the DirectPlayMessageHandler() function, and take note of where the switch statement looks for this message type. This is where the hrCreatePlayer() function is called from. Here is the code for the function.

```
HRESULT     hrCreatePlayer( PVOID pvUserContext, PVOID pMsgBuffer )
{
    HRESULT                  hReturn = S_OK;
    PDPNMSG_CREATE_PLAYER    pCreatePlayerMsg;
    char                     strName[256];
    char                     szOutput[256];
    DWORD                    dwSize = 0;
    DPN_PLAYER_INFO          *pdpPlayerInfo = NULL;
    int                      i;

    // Get a Create Message pointer to the buffer
    pCreatePlayerMsg = (PDPNMSG_CREATE_PLAYER)pMsgBuffer;

    // Get the peer info and extract its name
    hReturn = g_pDP->GetPeerInfo( pCreatePlayerMsg->dpnidPlayer,
     pdpPlayerInfo, &dwSize, 0 );
    if( FAILED(hReturn) && hReturn != DPNERR_BUFFERTOOSMALL ) {
        hReturn = -1;
    }
    else {
        pdpPlayerInfo = (DPN_PLAYER_INFO*) new BYTE[ dwSize ];
        ZeroMemory( pdpPlayerInfo, dwSize );
        pdpPlayerInfo->dwSize = sizeof(DPN_PLAYER_INFO);
        hReturn = g_pDP->GetPeerInfo( pCreatePlayerMsg->dpnidPlayer,
         pdpPlayerInfo, &dwSize, 0 );
        if( FAILED(hReturn) ) {
            hReturn = -1;
        }
        else {
            EnterCriticalSection( &g_csModifyPlayer );
            // Convert player name to ANSI
            DXUtil_ConvertWideStringToGeneric( strName,
             pdpPlayerInfo->pwszName );
            // Add player to list
            for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
                if( !PlayerInfo[i].bActive ) {
                    PlayerInfo[i].bActive = 1;
                    PlayerInfo[i].dpnidPlayer =
                     pCreatePlayerMsg->dpnidPlayer;
                    strcpy(PlayerInfo[i].szPlayerName,strName);
                    break;
                }
            }
            // Check whether no free slot found
            if( i == MAX_PLAYERS ) {
                vShowText(hLB_Output,"No free slots in game!");
            }
            // Check whether we are adding ourselves
            else if( pdpPlayerInfo->dwPlayerFlags & DPNPLAYER_LOCAL ) {
                g_dpnidLocalPlayer = pCreatePlayerMsg->dpnidPlayer;
                sprintf(szOutput,"<Slot%d> Added Ourselves",i);
                vShowText(hLB_Output,szOutput);
            }
            else {
                sprintf(szOutput,"<Slot%d><%s> Is In The Game",i,strName);
                vShowText(hLB_Output,szOutput);
                // Send a welcoming message if we are the host
                sprintf(szOutput,"Welcome to the game, %s!",strName);
                if( bHost ) {
                    hrSendChatMessage(i,szOutput);
                }
            }
            SAFE_DELETE_ARRAY( pdpPlayerInfo );
            // Update the number of active players in a thread-safe way
            InterlockedIncrement( &g_lNumberOfActivePlayers );
            LeaveCriticalSection( &g_csModifyPlayer );
        }
    }
    return hReturn;
}
```

The `hrCreatePlayer()` function takes two parameters. The first one passes the optional player context. I don't do anything with it for this example, so you can ignore it for now. The second parameter holds the message that was received in the message handler.

To do much of anything with a DirectPlay message, you must first cast it to the data type with which you want to work. Because you know that you want to work with a create player message in this function, it is safe to cast the message to type PDPNMSG_CREATE_PLAYER. This converts the raw message data into a usable structure for your needs.

TIP

The definitions for the different data types appear in the dplay8.h header file that comes with the DirectX SDK.

Because the message type begins with the letter P, you can tell it is a pointer to the requested data type. It is useful to keep this in mind when searching the DirectX help file. The index won't pick up on PDPNMSG_CREATE_PLAYER, but it will pick up on DPNMSG_CREATE_PLAYER.

Okay, so what is in the DPNMSG_CREATE_PLAYER message? Well, here you go.

```
typedef struct     _DPNMSG_CREATE_PLAYER
{
    DWORD       dwSize;               // Size of this structure
    DPNID       dpnidPlayer;          // DPNID of new player
    PVOID       pvPlayerContext;      // Player context value
} DPNMSG_CREATE_PLAYER, *PDPNMSG_CREATE_PLAYER;
```

The only member variable in it that you should care about for now is the dpnidPlayer variable. It is of type DPNID and it holds the identification number for the player who sent the message. This fact is very important because you need to know who the player is that you are creating.

Now that the pCreatePlayerMsg variable points to proper data, it is time to use the GetPeerInfo() member function to get the information about the peer who is being created.

The IDirectPlay8Peer::GetPeerInfo() Function

You use the GetPeerInfo() function primarily to get information about a peer in the game session. Here is the prototype.

```
HRESULT GetPeerInfo(
    const DPNID dpnid,
    DPN_PLAYER_INFO *const pdpnPlayerInfo,
    DWORD *const pdwSize,
    const DWORD dwFlags
);
```

The first parameter expects the DPNID, or identification number, of the peer about which you want to get information. For this example, the program passes the DPNID contained in the pCreatePlayerMsg pointer for the first parameter.

The second parameter is used to pass the DPN_PLAYER_INFO data structure that will hold the peer information once it is retrieved. For this parameter, you pass a pointer of type DPN_PLAYER_INFO called pdpPlayerInfo.

The third parameter holds the size of the returned data. This one is easy because all you have to do is pass in the DWORD variable called dwSize. The fourth parameter is reserved data and should always be set to zero. 'Nuff said.

Because you have not allocated any memory for pdpPlayerInfo yet, you'll probably get a DPNERR_BUFFERTOOSMALL message back from the GetPeerInfo() function. Not to worry, though, because this is expected. Player information can vary in size, so it is necessary to go through these steps. The first call told you the size of the player data. Now that you know the size, you can allocate memory for it with the following line of code.

```
pdpPlayerInfo = (DPN_PLAYER_INFO*) new BYTE[ dwSize ];
```

All this does is allocate as many bytes as the previous GetPeerInfo() function said was necessary to hold the data. Now that the data is allocated, you can call GetPeerInfo() again and not get the DPNERR_BUFFERTOOSMALL error. This process might seem a little confusing, so take a look at Figure 9.11 to see it in an illustration.

Storing the Player Information

The program didn't get an error on the second GetPeerInfo() call, so now it should be in the block of code that actually adds the new player data to the global PlayerInfo data structure. Before anything else happens, the game enters into the critical section for modifying player data. This section keeps multiple threads from attempting to modify the data at once.

The next piece of code loops through the maximum number of players and looks for ones who are not active. It is looking for an empty player slot. Once an empty slot is found, the program then sets the slot as active to keep it from being written over by another thread. The next thing that happens is that the application stores the DPNID of the player being created. Last, the application stores the name of the player found in the pdpPlayerInfo data structure.

The DPNPLAYER_LOCAL Flag

One of the most important things to check is whether the player you are adding is yourself. This is important to know because you don't yet have your own DPNID

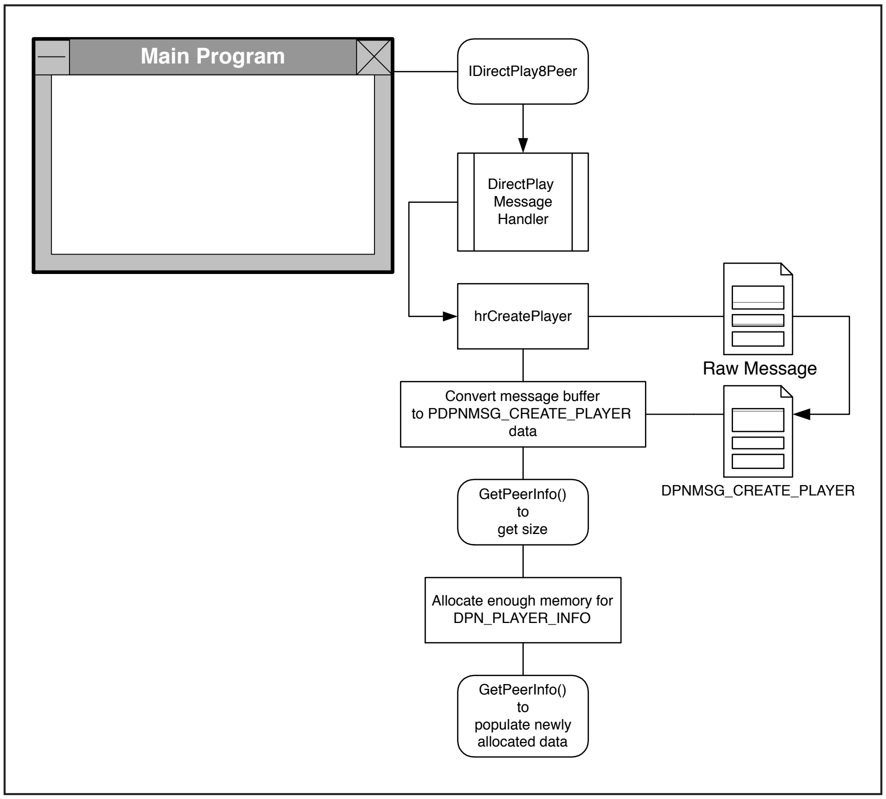

Main Program
IDirectPlay8Peer
DirectPlay Message Handler
hrCreatePlayer
Raw Message
Convert message buffer to PDPNMSG_CREATE_PLAYER data
DPNMSG_CREATE_PLAYER
GetPeerInfo() to get size
Allocate enough memory for DPN_PLAYER_INFO
GetPeerInfo() to populate newly allocated data

Figure 9.11

Retrieving peer information

stored. To check whether this player is yourself, you can check the `dwPlayerFlags` in the `pdpPlayerInfo` data structure. If the `DPNPLAYER_LOCAL` bit is set, then you know you are adding yourself.

If the code runs and it is not adding itself, it then checks whether it is the host. If it is the host, it sends a welcome message to the joining player by using the `hrSendChatMessage()` function, which I'll talk about in a minute.

Now that all the hoopla is out of the way, it is time to clean up allocated memory. Do you remember allocating memory for the `pdpPlayerInfo` structure? Well, guess what: You need to delete it now that you are done with it. To do so, you use the `SAFE_DELETE_ARRAY` macro provided by the `DXUtil` toolset.

The InterlockedIncrement() Function

Wow, doesn't the next piece of code look weird? What the heck is an interlocked increment anyway? Well, it is a means to increment a global variable in a thread-safe way. Basically, it keeps two threads from trying to use the same variable simultaneously. The function returns a `LONG` that is the resulting value of the variable you pass into it.

Okay, memory is cleaned up, and you have incremented the number of active players in the game. Now you have to leave the critical section you entered earlier. If you don't leave the section, the program will never be able to enter it again. This results in a lockup of the code.

Before I discuss joining a game, I want to leave you with a visual diagram of creating a player, so take a look at Figure 9.12.

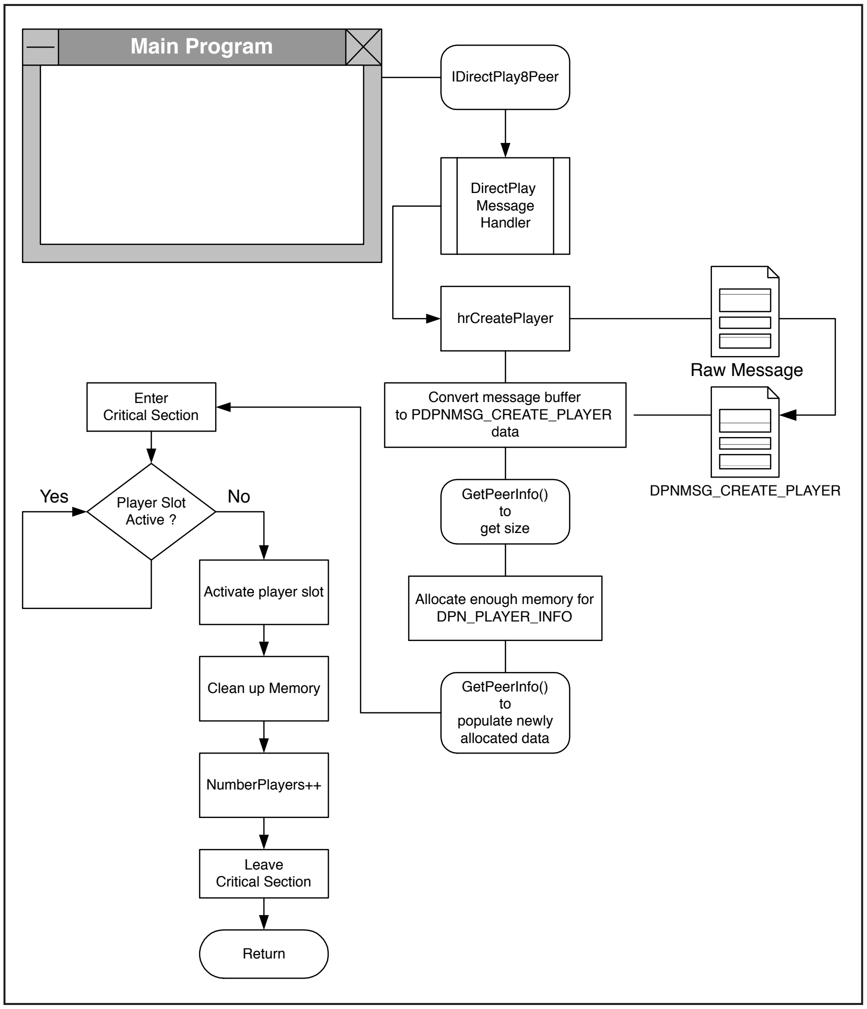


Figure 9.12

Creating a player

The hrJoinGame() Function

If you haven't taken a break in a while, now might be a good time to do so. Joining a session isn't all that problematic, and you might even notice some familiar functions along the way. Here is the code for the `hrJoinGame()` function.

```
HRESULT hrJoinGame( HWND hWnd )
{
    HRESULT                      hReturn = S_OK;
    WCHAR                        wszHostName[256];
    WCHAR                        wszPeerName[256];
    char                         szPeerName[256];
    char                         szIP[256];
    char                         szPort[256];
    DWORD                        dwPort;
    DWORD                        dwLength = 256;
    DPN_APPLICATION_DESC         dpnAppDesc;
    DPN_PLAYER_INFO              dpPlayerInfo;

    vShowText(hLB_Output,"Attempting Connection...");
    // Set the peer info
    GetWindowText(hEB_InputName,szPeerName,36);
    // Get name from window edit box
    DXUtil_ConvertGenericStringToWide( wszPeerName, szPeerName );
    ZeroMemory( &dpPlayerInfo, sizeof(DPN_PLAYER_INFO) );
    dpPlayerInfo.dwSize = sizeof(DPN_PLAYER_INFO);
    dpPlayerInfo.dwInfoFlags = DPNINFO_NAME;
    dpPlayerInfo.pwszName = wszPeerName;
    // Make this a synchronous call
    if( FAILED( hReturn = g_pDP->SetPeerInfo( &dpPlayerInfo, NULL, NULL,
     DPNSETPEERINFO_SYNC ) ) ) {
        vShowText(hLB_Output,"Failed to set peer info");
        return -1;
    }
    // Prepare the application description
    ZeroMemory( &dpnAppDesc, sizeof( DPN_APPLICATION_DESC ) );
    dpnAppDesc.dwSize = sizeof( DPN_APPLICATION_DESC );
    dpnAppDesc.guidApplication = DP_CHAT;
    // Get IP from edit box
    GetWindowText(hEB_InputServerIP,szIP,32);
    // Get port from edit box
    GetWindowText(hEB_InputServerPort,szPort,6);
    // Convert the IP to a wide string
    DXUtil_ConvertGenericStringToWide( wszHostName, szIP );
    // Convert the port string to a DWORD
    dwPort = atol(szPort);
    // Add hostname to address
    hReturn = g_pHostAddress-
    >AddComponent(DPNA_KEY_HOSTNAME,wszHostName,(
    wcslen(wszHostName)+1)*sizeof(WCHAR),DPNA_DATATYPE_STRING);
    if( hReturn != S_OK ) {
        MessageBox( hWnd, "Failed to AddComponent()", "hrJoinGame()",
         MB_ICONERROR );
        return -1;
    }
    // Add port number to address
    hReturn = g_pHostAddress-
    >AddComponent(DPNA_KEY_PORT,&dwPort,sizeof(
    DWORD),DPNA_DATATYPE_DWORD);
    if( hReturn != S_OK ) {
        MessageBox( hWnd, "Failed to AddComponent()", "hrJoinGame()",
         MB_ICONERROR );
        return -1;
    }
    // Connect to the session
    hReturn = g_pDP->Connect(     &dpnAppDesc,
                    g_pHostAddress,
                    g_pDeviceAddress,
                    NULL,
                    NULL,
                    NULL,
                    0,
                    NULL,
                    NULL,
                    &g_hConnectAsyncOp,
                    NULL); // flags
    if( hReturn != E_PENDING && FAILED(hReturn) ) {
        vShowText(hLB_Output,"Failed to Connect");
        return -1;
    }
    SetTimer( hWnd, TIMERID_CONNECT_COMPLETE, 100, NULL );
    return(hReturn);
}
```

This function is activated whenever the player clicks the Join button. There really isn't much to pass to the function except the current window handle.

Setting the Joiner's Peer Information

The first piece of code that sets up the `DPN_PLAYER_INFO` data structure is just like the code used in the `hrHostGame()` function I covered earlier. What this accomplishes is setting up the joiner's player information, such as the name. The name is grabbed from the window's edit box using the good ol' `GetWindowText()` function. All this does is pull the text from the window and load it into a variable you can use. Don't forget that most DirectPlay strings use wide characters, so you must convert most strings with the `DXUtil_ConvertGenericStringToWide()` helper function.

Hey, cool; I just put together a quest item in *Asheron's Call*. It has a special sword hilt in the game that, when combined with a rapier, allows it to swing twice per attack. I tried it out on Drudge Stalker and wailed on it. The rapier I converted turned into a Bandit Rapier once I applied the new hilt. It sure is nice to have quests such as this in MMORPGs. Thanks go out to Thrillseeker for helping me get the required items for the quest.

Setting the Joiner's Application Description

After you have set up the peer information, you can set up the application description. This works just as it did when you were hosting, but this time you don't have to tell it to migrate the host.

Setting Up the Host Address

Now that the application description is out of the way, you get to set up the host address. You do this with the `AddComponent()` function, which you should recognize by now. The first component to add is the port number, with the `DPNA_KEY_PORT` component type. The second thing to add to the address is the host's name, with the `DPNA_KEY_HOSTNAME` component type.

The Connect() Function

Ah, the glorious `Connect()` function. Don't forget that this function belongs to the `IDirectPlay8Peer` interface. To call this function, you use the parameters listed in Table 9.3.

If all goes well, then the code proceeds to set up a timer that goes off every 100 milliseconds. This timer is set up to signal the program to check for a connection complete indicator. Because it can take a little while to connect, you don't want to sit in

Table 9.3 Connection Parameters

Parameter	Description
`&dpnAppDesc`	The application description for the game
`g_pHostAddress`	The address of the host with which to join up
`g_pDeviceAddress`	Your address
`NULL`	Reserved
`NULL`	Reserved
`NULL`	Optional user connect data, which is not used here
`0`	Optional user connect data size
`NULL`	Optional player context
`NULL`	Optional user-supplied context
`&g_hConnectAsyncOp`	Asynchronous handle
`NULL`	Flags

an infinite loop waiting to do so. Using a timer, the program can continue on its merry way and wait for the connection to complete.

The DPN_MSGID_CONNECT_COMPLETE Message

The big question you might have is, how do you detect when the connection is complete? Well, when you get a `DPN_MSGID_CONNECT_COMPLETE` message, you know the connection is complete. If you take a look at the message handler, you'll notice the following chunk of code.

```
case DPN_MSGID_CONNECT_COMPLETE:
{
      PDPNMSG_CONNECT_COMPLETE pConnectCompleteMsg;
      pConnectCompleteMsg = (PDPNMSG_CONNECT_COMPLETE)pMsgBuffer;
      g_hrConnectComplete = pConnectCompleteMsg->hResultCode;
      SetEvent( g_hConnectCompleteEvent );
      break;
}
```

When the connection finishes, this message is received. The first thing you do with it is cast the raw message data into the PDPNMSG_CONNECT_COMPLETE message type. This lets you pull the pertinent data out of the message.

Once the data is cast, you then set the g_hrConnectComplete variable to equal the result of the connection. Now that the variable is set, it's time to set up the completion event. You do this by setting the g_hConnectCompleteEvent event with the SetEvent() function. By calling this function, you put the event into a signaled state.

Reading the Connection Timer

Do you remember the timer that was just set up? Well, every time the timer goes off, it sends a WM_TIMER message to the main Windows message loop. Here is the code that gets called when the timer goes off.

```
case WM_TIMER:
    if( wParam == TIMERID_CONNECT_COMPLETE ) {
        // Check whether the message is telling us
            our connection is complete
        if( WAIT_OBJECT_0 == WaitForSingleObject(
         g_hConnectCompleteEvent, 0 ) ) {
            if( FAILED( g_hrConnectComplete ) ) {
                vShowText(hLB_Output,"<<----CONNECTION
                 IN-COMPLETE---->>");
            }
            else {
                vShowText(hLB_Output,"<<----CONNECTION
                 COMPLETE---->>");
            }
            KillTimer( hWnd, TIMERID_CONNECT_COMPLETE );
        }
    }
    break;
```

The wParam of the timer message contains the timer ID that triggered the message. What you do is check for the TIMERID_CONNECT_COMPLETE timer ID that you set up earlier. After that check passes, you then check to see whether the completion event has been signaled by using the WaitForSingleObject() function. If it equals WAIT_OBJECT_0, then it has been signaled. Now that is out of the way, so you can check the return code from the connection attempt by looking at the g_hrConnectComplete variable. If it failed, then the host denied the connection or wasn't available. If it passed, then the connection is finally complete. The last thing to do is kill the timer so that it doesn't keep messaging the program.

Creating the Players from the Joiner's View

Guess what happens right after the connection occurs? If you guessed that players are created, then you are right. The first message you probably get is the `DPN_MSGID_CREATE_PLAYER` message, telling you to create the players in the game. Unlike the host, the joiner gets at least two create messages, one for the host and one for the joiner. If any other players are present, then you also get creation messages for them. All of this is handled with the `hrCreatePlayer()` function I discussed earlier. Oh, yeah; the host also gets a create player message when you join.

Before you move on, take a gander at Figure 9.13, which shows an illustration of the whole connection process.

The hrSendChatMessage() Function

Up until now, I have only shown you the boring connection stuff. Now I want to show you how to send chat messages back and forth. Lucky for you, this part is pretty short. Lucky for my carpal tunnel syndrome, too! Stand back; here comes a tsunami of code.

```
HRESULT     hrSendChatMessage(int player, char     *szmessage)
{
    PACKET_CHAT             msgChat;
    DPNHANDLE               hAsync;
    DWORD                   dwLength = strlen(szmessage);
    DPN_BUFFER_DESC         bufferDesc;

    // If no message to send, then just return
    if( dwLength == 0 )
        return S_OK;
    // Copy the message to send into our packet
    strcpy(msgChat.szText,szmessage);
    // Set the size of the packet to send
    bufferDesc.dwBufferSize = sizeof(PACKET_CHAT) + dwLength;
    // Copy our packet into the send buffer
    bufferDesc.pBufferData  = (BYTE*) &msgChat;
    // Send message to everyone, including ourselves if -1 passed
    if( player == -1 )
    g_pDP->SendTo( DPNID_ALL_PLAYERS_GROUP, &bufferDesc, 1, 0,
     NULL, &hAsync, 0 );
    // Send to specific player otherwise
    else
    g_pDP->SendTo( PlayerInfo[player].dpnidPlayer, &bufferDesc, 1, 0,
     NULL, &hAsync, 0 );
    return S_OK;
}
```

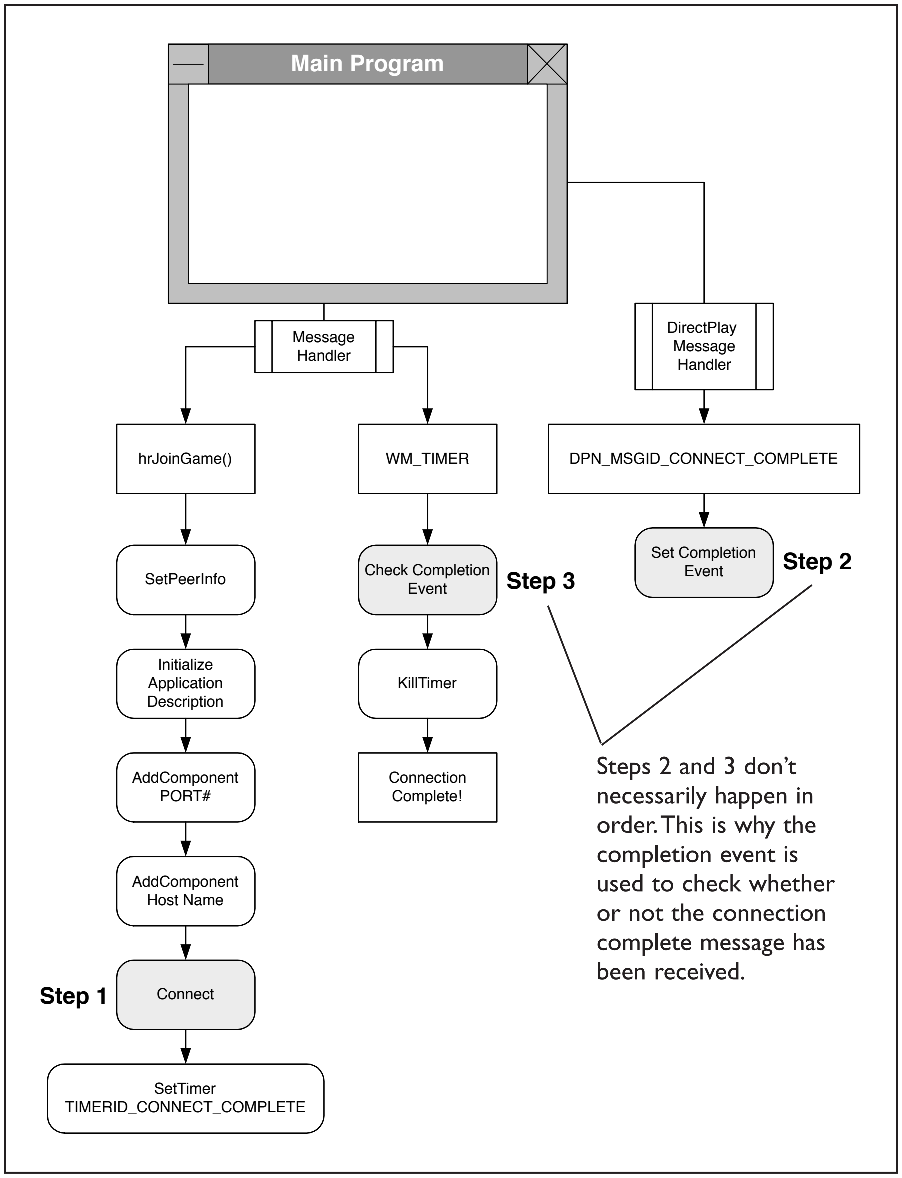

Main Program
Message Handler
DirectPlay Message Handler
hrJoinGame()
WM_TIMER
DPN_MSGID_CONNECT_COMPLETE
SetPeerInfo
Check Completion Event
Step 3
Set Completion Event
Step 2
Initialize Application Description
KillTimer
AddComponent PORT#
Connection Complete!
Steps 2 and 3 don't necessarily happen in order. This is why the completion event is used to check whether or not the connection complete message has been received.
AddComponent Host Name
Step 1
Connect
SetTimer TIMERID_CONNECT_COMPLETE

Figure 9.13

The entire connection process

Okay, maybe it was more of a ripple than a tsunami. Speaking of ripples, have you ever tried the iced fudge ripples from the Coffee Beanery? Man, they taste good. They also have about double the caffeine of any other drink the Beanery sells.

This function takes two parameters. The first parameter is used to flag to which player to send the message. If it is set to `-1`, then the program is supposed to *broadcast* the message to every player present.

A message sent to every player in a game is called a *broadcast* message.

The second parameter contains the message to send. Oops, I almost forgot to mention where the chat message comes from. A chat message is fired off when the player types a message into the window and presses the Enter key. The Enter key is captured with the `EN_UPDATE` message. Check out the main Windows message handler to see this code in action.

The first thing that happens when the function takes place is a check for the length of the message to be sent. If the message is empty, then the function returns. The last thing you want to do is send empty messages across the wire.

Once that is out of the way, the program copies the contents of the message into the `PACKET_CHAT` data structure. The `PACKET_CHAT` data structure that was defined in DPChat.h contains nothing more than a character buffer of 256 elements.

The DPN_BUFFER_DESC Data Structure

Next, the program loads the `DPN_BUFFER_DESC` data structure with the message information, such as the size of the message. This variable type sends all DirectPlay messages. To set the size of the message, you set the `dwBufferSize` member variable. The `pBufferData` member variable points to the actual data to be sent. These are the only two members available to this data structure.

The SendTo() Function

Now that the `DPN_BUFFER_DESC` structure is loaded, you can go ahead and send the message using the `SendTo()` member function. Here is the prototype for the function.

```
HRESULT SendTo(
    const DPNID dpnid,
    const DPN_BUFFER_DESC *const pBufferDesc,
    const DWORD cBufferDesc,
    const DWORD dwTimeOut,
    void *const pvAsyncContext,
    DPNHANDLE *const phAsyncHandle,
    const DWORD dwFlags
);
```

The first parameter sets which player to send the message to. If you want to broadcast the message, then you can use the `DPNID_ALL_PLAYERS_GROUP` keyword here; otherwise, you need to pass the `DPNID` of the player to whom you want to send.

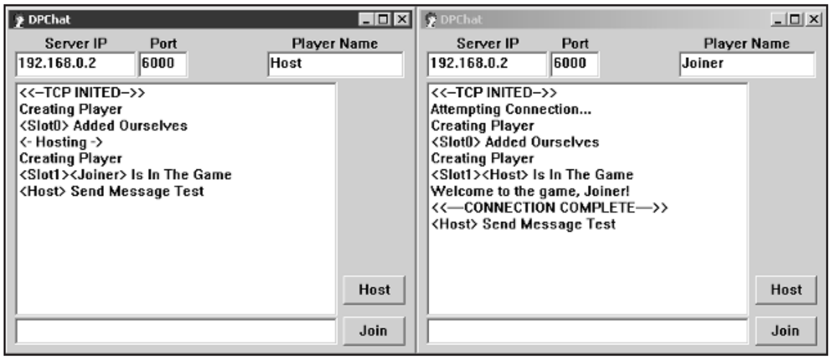
The second parameter expects the buffer description you loaded earlier with the message information. The third parameter expects the number of buffers you are sending. In this case, you are only sending one, so a `1` is passed here. The fourth parameter is a timeout setting and lets you time out after so many milliseconds have passed. Because you don't want to time out, you set it to `0` to wait forever.

The fifth parameter is the asynchronous context and is optional. Because it is optional and you are not using it, set it to `0` or `NULL`. The sixth parameter holds an asynchronous handle. You can later use this handle to cancel the operation with the `CancelAsyncOperation()` member function if you so desire. The last parameter lets you set up different flags. Refer to Table 9.4 to see a list of available flags.

That's it for the `SendTo()` function. Now that you are a veteran of transmitting data, why don't you peek at Figure 9.14 and see what the output looks like so far?

The hrDestroyPlayer() Function

There isn't too much left to the chat program, but I still have some important issues to discuss. The next function I want to cover is `hrDestroyPlayer()`. This gets called whenever a `DPN_MSGID_DESTROY_PLAYER` message is received. You can expect to receive this type of message whenever a player quits the chat program.



```
<<–TCP INITED–>>
Creating Player
<Slot0> Added Ourselves
<- Hosting ->
Creating Player
<Slot1><Joiner> Is In The Game
<Host> Send Message Test
```

```
<<–TCP INITED–>>
Attempting Connection...
Creating Player
<Slot0> Added Ourselves
Creating Player
<Slot1><Host> Is In The Game
Welcome to the game, Joiner!
<<—CONNECTION COMPLETE—>>
<Host> Send Message Test
```

Figure 9.14

Output from DPChat when sending a message

Table 9.4 SendTo() Flags

Flag	Description
DPNSEND_SYNC	Tells the program to process the send synchronously.
DPNSEND_NOCOPY	Keeps from making an internal copy, which can cause problems.
DPNSEND_NOCOMPLETE	Prevents the DPN_MSGID_SEND_COMPLETE message from being received when a message is complete.
DPNSEND_COMPLETEONPROCESS	Sends a DPN_MSGID_SEND_COMPLETE message to the message handler when the message is received and processed by the target. This can greatly slow down transmissions. You must use DPN_GUARANTEED with this flag.
DPNSEND_GUARANTEED	Guarantees delivery of the message. Can slow down transmissions considerably.
DPNSEND_PRIORITY_HIGH	Sets the priority to high.
DPNSEND_PRIORITY_LOW	Sets the priority to low.
DPNSEND_NONSEQUENTIAL	If this is set, then the target receives the messages in the order they arrive and not in the order they were sent.
DPNSEND_NOLOOPBACK	Keeps you from sending a broadcast message back to yourself.

The purpose of this function is the opposite of that for hrCreatePlayer(). Basically, if someone quits, you need to remove his player information from the PlayerInfo structure. Here is the destruction code.

```
HRESULT     hrDestroyPlayer( PVOID pvUserContext, PVOID pMsgBuffer )
{
    PDPNMSG_DESTROY_PLAYER              pDestroyPlayerMsg;
    HRESULT                             hReturn = S_OK;
   int                                      i;
    char                                    szOutput[256];
    // Get a destroy message pointer to the buffer
    pDestroyPlayerMsg = (PDPNMSG_DESTROY_PLAYER)pMsgBuffer;
    if( pDestroyPlayerMsg->dwReason == DPNDESTROYPLAYERREASON_NORMAL ) {
        vShowText(hLB_Output,"Player Left Normally");
    }
    else if( pDestroyPlayerMsg->dwReason ==
     DPNDESTROYPLAYERREASON_CONNECTIONLOST  )  {
        vShowText(hLB_Output,"Connection Lost w/Player");
    }
    else if( pDestroyPlayerMsg->dwReason ==
     DPNDESTROYPLAYERREASON_SESSIONTERMINATED  ) {
        vShowText(hLB_Output,"Player Terminated Session");
    }
    else if( pDestroyPlayerMsg->dwReason ==
     DPNDESTROYPLAYERREASON_HOSTDESTROYEDPLAYER  ) {
        vShowText(hLB_Output,"Player Kicked By Host");
    }
    // Update the number of active players in a thread-safe way
    InterlockedDecrement( &g_lNumberOfActivePlayers );
    EnterCriticalSection( &g_csModifyPlayer );
    // Remove player from list
    for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
        if( PlayerInfo[i].bActive ) {
            if( PlayerInfo[i].dpnidPlayer ==
             pDestroyPlayerMsg->dpnidPlayer ) {
                PlayerInfo[i].bActive = 0;
                sprintf(szOutput,"<Slot%d><%s>
                 Left The Game",i,PlayerInfo[i].szPlayerName);
                vShowText(hLB_Output,szOutput);
                break;
            }
        }
    }
    LeaveCriticalSection( &g_csModifyPlayer );
    return(hReturn);
}
```

Casting the Message

The first part, as you should know, casts the raw message data into the message with which you need to work. The message type you use for this function is `PDPNMSG_DESTROY_PLAYER`. One member of this message type is `dwReason`. This member variable tells you the reason why the player is being destroyed. The next few lines of code output different text depending on how the player left.

The InterlockedDecrement() Function

The `InterlockedDecrement()` function is the nemesis of `InterlockedIncrement()`. Instead of adding to the number of players in a thread-safe way, this function subtracts from the total number of players. You use this function to keep track of how many people are playing. Because someone has left, you need to reduce the global setting for the number of active players.

Critical Player Information Handling

Once again, you need to use the `EnterCriticalSection()` function to lock out any other threads from accessing the `PlayerInfo` data in the `hrDestroyPlayer()` function. Now that the program is in a critical section, it can loop through the connected players and find the one to disable.

The only cleanup you have to do is leave the critical section you entered earlier and—bam!—you are done with destroying players. (I hope the *Emeril* fans out there enjoyed that.)

The DPN_MSGID_HOST_MIGRATE Message

Wow, the engines must be running at warp factor 9; you are tearing through this chapter. There isn't much left now. If you look at the `DirectPlayMessageHandler()` function, you might notice a couple of messages I haven't covered yet. The first one is `DPN_MSGID_HOST_MIGRATE`. This message is received if the host who quits the game has migration turned on.

To find out whether you were the one chosen to take up the slack, you can cast the raw message to a `PDPNMSG_HOST_MIGRATE` data structure and check the `dpnidNewHost` member variable versus your own `DPNID`. If the two match, then you are the new host. You don't have to do anything special here; DirectPlay handles it all for you. Isn't that cool?

The DPN_MSGID_TERMINATE_SESSION Message

The last message in the message handler's arsenal for this program is DPN_MSGID_TERMINATE_SESSION. It is sent to all connecting peers when a host has ended the session and migration is not turned on. So if the host exits and the program is not migrated, then the whole session is terminated and no one can chat anymore.

Before you go, I want to leave you with this question to ponder: If FedEx and UPS merged, would the resulting company be called FedUp?

Review

The main thing you should take with you from this chapter is the knowledge of how to create a complete end-to-end DirectPlay program. Besides adding some nice graphics and flair to the code, you have the foundation for any future peer-to-peer game you write. Here are some of the key points to remember.

- DirectPlay requires COM to operate. You initialize COM using the `CoInitialize()` function.
- The `CoCreateInstance()` function is used to create the DirectPlay interfaces. The key interfaces for peer-to-peer communication are `IDirectPlay8Peer` and `IDirectPlay8Address`.
- The DirectPlay message handler receives and processes all communication messages received.
- The `SetSP()` function is used to set the service provider for your connection. The DPChat program utilizes TCP/IP as its method for communicating.
- The `SendTo()` function is used to send data to other players. You can optionally use the `DPNID_ALL_PLAYERS_GROUP` flag to send data to every player who is connected.
- Global data, such as the player data, must be locked within critical sections to avoid memory collisions.
- You have to call the `Close()` function before releasing the `IDirectPlay8Peer` interface. This closes the active connection.

This page intentionally left blank

CHAPTER 10

Lions and Clients and Servers, Oh My!

Now that you have made the long journey to the Emerald City, I guess you will want to see the wizard. Step right over here, and be sure not to look behind the curtain. Okay, the tin-man needs a server, and the scarecrow wants a client. What's that, Dorothy? You want to go back to playing *Diablo II* in Kansas?

Hey, now; you didn't think I would go off and forget to tell you about the client/server model, did you? Not only am I going to show you how to write a DirectPlay client, but I am also going to show you how to write a full server. Pack your bags now because it's time to follow the yellow brick road. *(Faint singing.)* Follow, follow, follow, follow, follow the yellow brick road. Down the brick road you will run across information about:

- Client/server architecture
- The `IDirectPlay8Server` interface
- Hosting a client/server game
- Managing players
- Processing client messages
- Sending messages to clients
- The `IDirectPlay8Client` interface
- Connecting to the server
- Sending messages to the server

The Wicked Server of the West

Now, my pretty, it's your turn to. . . . Okay, enough of the Oz theme. In the last chapter, I told you all about how to create a peer-to-peer chat program. The good part is that most of what you learned in that chapter translates into the client/server model as well. What is the bad part? Well, there really is no bad part to it.

Client/Server Architecture Overview

Because I have pounded this one into your noggin many times already, I will only briefly discuss the architecture behind the client/server model. Basically, the server is responsible for maintaining all game connections and communication. Clients are only responsible for talking to the server, not to each other. Refer to Figure 10.1 to see the client/server architecture again.

This method of multiplayer game design happens to be my favorite because of its scalability, firewalls, and simplicity.

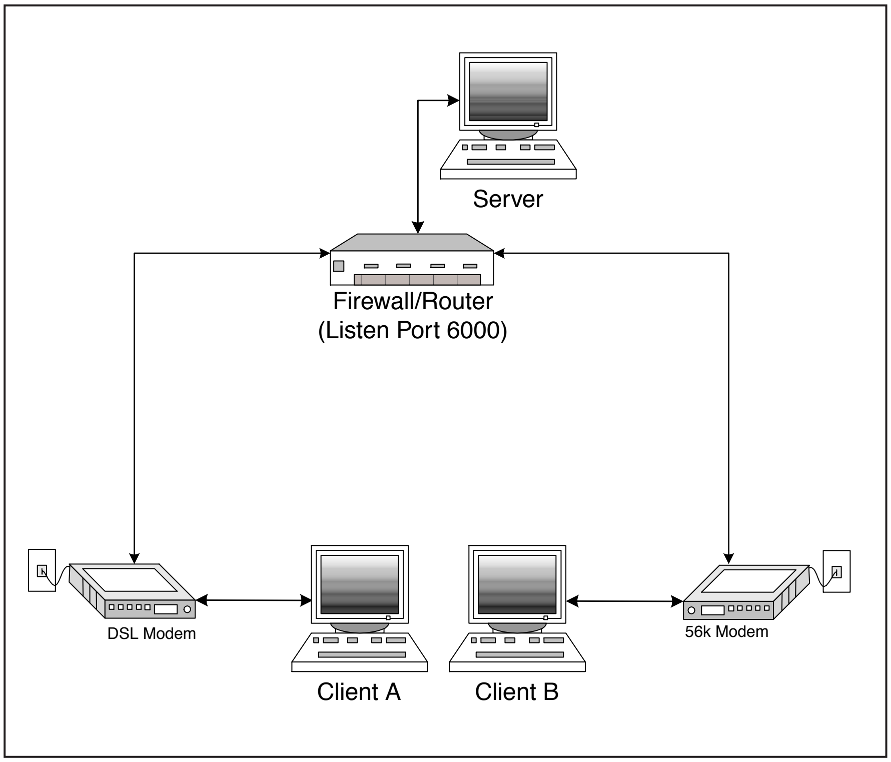


Figure 10.1

Client/server architecture

Server Scalability

First and foremost, by having a dedicated server, your game can handle more players at a time. Most peer-to-peer systems can reliably handle only four to six players at once over standard Internet connections. Client/server applications are limited only by the power of the server, not the clients. Games such as *EverQuest* and *Asheron's Call* handle thousands of players at once because they use the client/server model.

When peer-to-peer games get too many players, you can do nothing but maybe switch to a LAN or play only with people who have T-3 connections at home. Both of these options are not really valid. When servers get bogged down, you only have to upgrade the one machine, not the peers. For example, you could even run your own persistent-world game on a standard DSL connection. If you started to get too many customers, you could upgrade it to a fractional T-1. If it really started taking off, then you could probably afford to lease a T-1 or better. The great part is that a client/server setup really has no reasonable limit, given enough hardware.

Firewalls

If you have ever played a game of *Age of Empires*, then you might have experienced its problem with firewalls. *AOE* is a peer-to-peer game that requires all players to have connections with each other. This arrangement works fine until someone is behind a firewall. These players flat out can't play the game.

Firewalls didn't used to be a large problem because most Internet users connected with modems straight from their computers. With today's availability of high bandwidth in the home, many users have home networks complete with firewalls. For these users, peer-to-peer games just won't work unless they start poking holes in their firewalls. Many times, this is not an easy task because the hardware usually limits firewalls to a few open ports at once. Lobbies such as the Microsoft Gaming Zone require many more than eight ports to be open. Check out Figure 10.2 to see an illustration of peers being blocked behind a firewall.

When games use the client/server architecture, they usually require only one port to be open to accept connections. Someone behind a firewall can normally get away with opening one port if he wants to be the host. Better yet, as long as the players behind firewalls don't host the games, they can join anyone else's game. To see this system in action, refer to Figure 10.3.

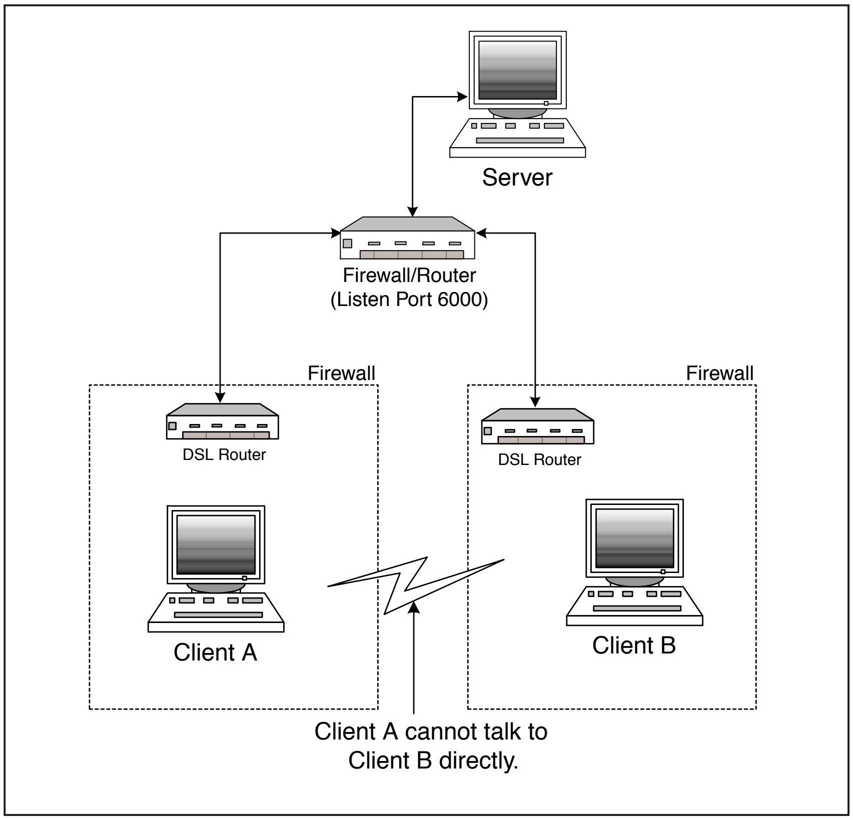


Figure 10.2

Peers blocked behind a firewall

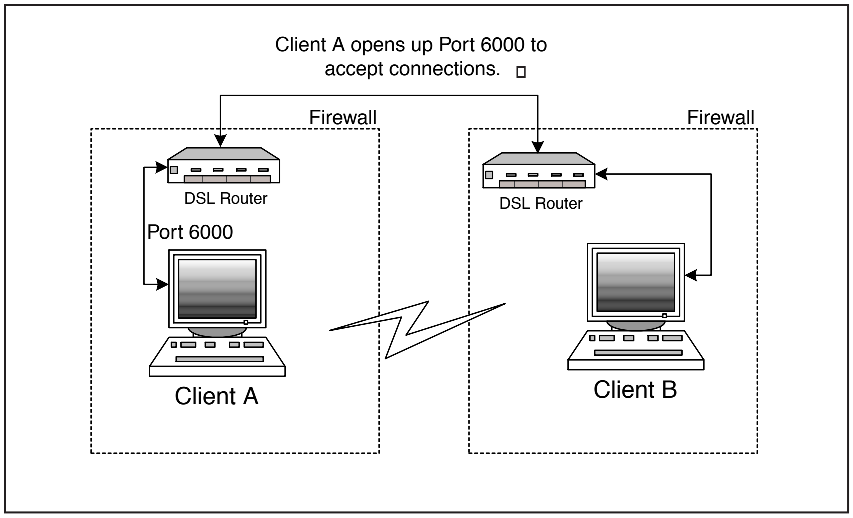


Figure 10.3

A client joining from behind a firewall

I guess one of the reasons I harp on this topic so much is that I am behind a firewall, and nothing hacks me off more than booting up a new game and finding out I can't connect to other players. Sometimes I disable the firewall, but I consider that a huge pain in the rear just to play a game.

TIP

There are some software alternatives to hardware firewalls that can be very useful. One such product, ZD Alarm, does most of the port management for you. When a game needs ports open, the software asks you if it is okay to open them. This way, you don't have to worry about setting up the firewall in advance.

Simplicity

Ah, sweet, blissful simplicity. My number-one favorite thing about the client/server model is the cleanliness of the code. The client/server system is simpler to look at for two main reasons.

- Clients have to worry only about themselves. Because they only know about the server, they don't have to worry about creating players or sending messages to everyone. Clients get to sit back and relax.
- You can create servers that are stand-alone programs. By doing this, you don't have to include the interface with the server program. This choice keeps things such as sound, graphics, and input out of the picture. It clears up your communication code for the important stuff—communicating!

A few more things make this model simpler, such as not having to worry about host migration, but I let my case rest with the preceding reasons. Okay, one more reason: Because the server program doesn't have to have an interface, you can write one that will run on multiple operating systems. Imagine having a Windows game client talking to UNIX-based servers. This scenario is more than just a fantasy because some persistent-world game companies do this already.

The IDirectPlay8Server Interface

There is nothing better than jumping into the thick of things, so here it is—the server interface. Most of the functions you will use for your server appear within the `IDirectPlay8Server` interface. This interface works just like all of the other DirectX interfaces in that it is a COM object and must be created with the `CoCreateInstance()` function. Oh, yeah; the code I reference is in the DPChatServer workspace located in the Chapter10 Zip file on the CD-ROM. Go ahead and load it up so you can follow along.

Creating the Server Interface

Before I go on, why don't you take a peek at the code used to create one of these interfaces? Please note that you cannot create the necessary DirectPlay objects without calling `CoInitialize()` to initialize COM first. You have to call it before you create any of the COM-based objects.

```
IDirectPlay8Server*          g_pDPServer;
// Create IDirectPlay8Server Object
if( FAILED( hReturn = CoCreateInstance( CLSID_DirectPlay8Server,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_IDirectPlay8Server,
     (LPVOID*) &g_pDPServer ) ) )
MessageBox( hWindow, "Can't Create DPlayServer", "DirectPlay Error", MB_ICONERROR );
```

Most of this should look fairly familiar; it works the same way as when you created the peer interfaces.

The server interface handles all communication and processing between the server and its clients. Most of the functions available to this interface appear in Table 10.1.

To see the functions in visual groupings, take a look at Figure 10.4.

Initializing the Server Interface

Before you can start listening for connections and such, you must first initialize the newly created interface using the `Initialize()` member function. Here is the prototype.

```
HRESULT Initialize(
    PVOID const pvUserContext,
    const PFNDPNMESSAGEHANDLER pfn,
    const DWORD dwFlags
);
```

The first parameter, `pvUserContext`, accepts a pointer to the user-specified context. One benefit this has is the ability to identify messages from multiple interfaces. Currently, I do not use this value.

The second parameter is the pointer to the message handler. This works just as it did for the peer interfaces. Every message that comes to the server object is routed to the message function specified here.

Table 10.1 IDirectPlay8Server Functions

Function	Purpose
`Initialize`	Sets up the interface to receive messages for connections
`Close`	Disconnects and closes the server from the current session
`Host`	Creates the session in which clients can connect
`SendTo`	Transmits data to one or many players within the session
`EnumServiceProviders`	Enumerates the service providers available to the server
`SetApplicationDesc`	Changes particular application settings for an existing session
`CreateGroup`	Creates a new group to which players can be added
`DestroyGroup`	Destroys a created group
`AddPlayerToGroup`	Adds a player to an existing group
`RemovePlayerFromGroup`	Removes a player from an existing group
`GetGroupInfo`	Retrieves the information about a group
`GetClientInfo`	Finds out the information about a particular client
`DestroyClient`	Removes a client from the session
`GetClientAddress`	Retrieves the DPlay address string for a client
`GetLocalHostAddresses`	Retrieves the DPlay address used to host the local server
`GetSendQueueInfo`	Monitors the send message queue for a particular client
`GetConnectionInfo`	Gets information about the connection with a particular client
`GetApplicationDesc`	Gets the application description for the connected application

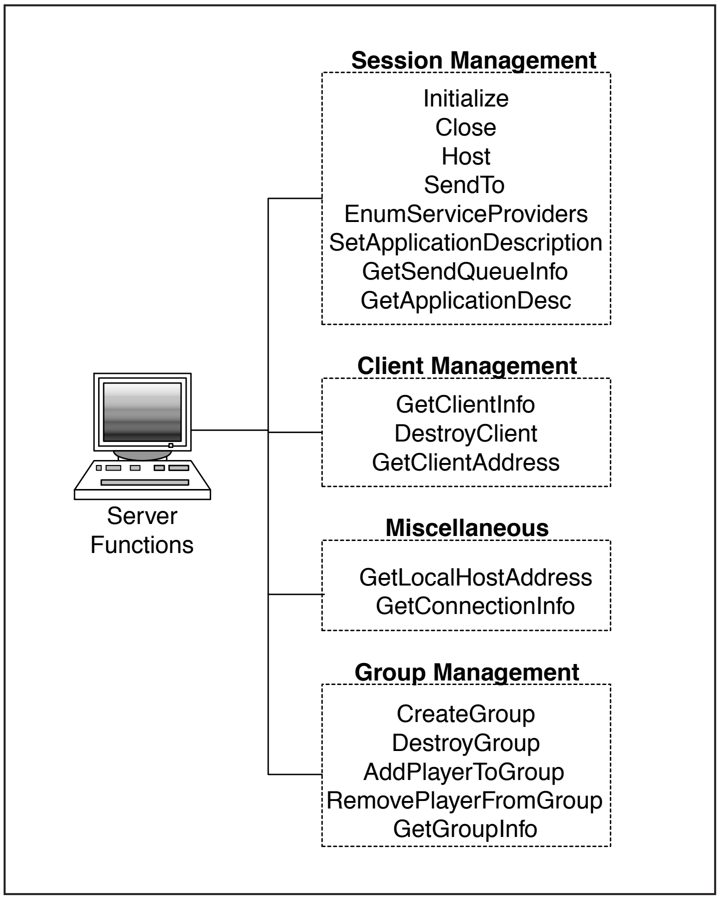


Figure 10.4
`IDirectPlay8Server` *interface functions*

The last parameter lets you set a flag. The only flag currently supported is `DPNINITIALIZE_DISABLEPARAMVAL`, which lets you disable parameter validation. Why you would want to do this is beyond me at this point. For most purposes, use the value of `NULL` here.

Now that initialization is out of the way, you can go on to create the famed address interface.

The IDirectPlay8Address Interface

Hey, guess what: This interface is the same one you have been using for the past few chapters. There is nothing different about it. Even though I have already shown you this interface, I'll go ahead and show you the code I use to set up the server's address.

```
// Create a device address
hReturn = CoCreateInstance( CLSID_DirectPlay8Address, NULL,CLSCTX_ALL,
 IID_IDirectPlay8Address, (LPVOID*) &g_pDeviceAddress );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Failed to Create Device", "CoCreateInstance()",
     MB_ICONERROR );
    return -1;
}
// Set our service provider to TCP/IP
if( FAILED( hReturn = g_pDeviceAddress->SetSP( &CLSID_DP8SP_TCPIP ) ) ) {
    MessageBox( hWindow, "Failed to SetSP() for Device Addr",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
```

Most of that should look familiar. The `CoCreateInstance()` function gets access to the interface, and then the `SetSP()` member function sets up TCP/IP. I bet you didn't expect it to go that quickly, did you?

Hosting the Session

Because you now know how to create a server interface, initialize it, and create its address, you can move on to actually hosting a game session. Hosting a client/server application works in much the same way as hosting a peer-to-peer application. You must follow a few basic steps to host a client/server game.

1. Set up the server information with the `SetServerInfo()` member function.
2. Set up the application description with the `DPN_APPLICATION_DESC` data structure.
3. Set up the address interface to handle any required information with the `AddComponent()` function.
4. Call the `Host()` member function to host the game.

These steps are illustrated in Figure 10.5.

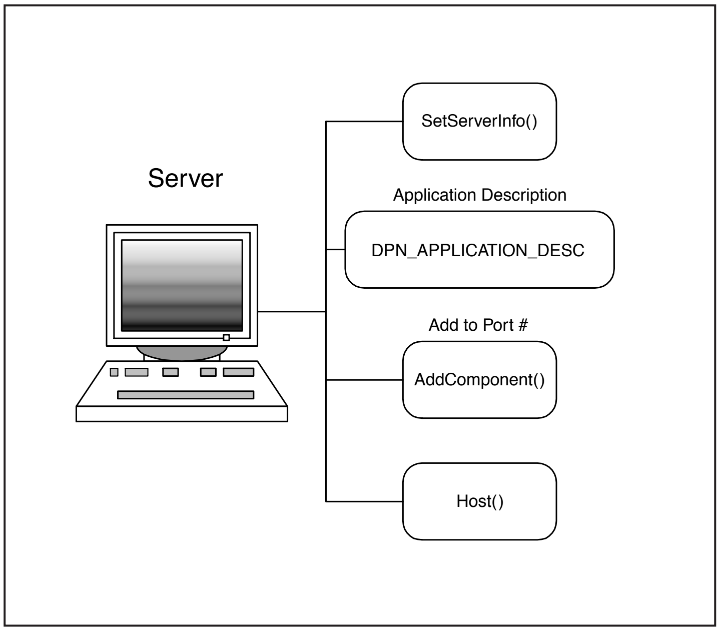


Figure 10.5

Server hosting

The SetServerInfo() Function

The `SetServerInfo()` function is a member of the `IDirectPlay8Server` interface. This function sets up the static information about a server. Clients who connect to the server can later obtain this information at their leisure. Here is the prototype for the function.

```
HRESULT SetServerInfo(
    const DPN_PLAYER_INFO *const pdpnPlayerInfo,
    PVOID const pvAsyncContext,
    DPNHANDLE *const phAsyncHandle,
    const DWORD dwFlags
);
```

The first parameter points to a `DPN_PLAYER_INFO` data structure that holds information about the server, such as its name. The second parameter can point to user-provided context. (This is optional.) The third parameter cannot be used at this time, so `NULL` is valid. The last parameter sets certain server flags. The `DPNSETSERVERINFO_SYNC` flag tells this method to process synchronously.

I don't want to leave you hanging, so here is the code snippet for setting the server information.

```
// Set up our player information
DXUtil_ConvertGenericStringToWide( wszServerName, "DPChat Server" );
ZeroMemory( &dpPlayerInfo, sizeof(DPN_PLAYER_INFO) );
dpPlayerInfo.dwSize = sizeof(DPN_PLAYER_INFO);
dpPlayerInfo.dwInfoFlags = DPNINFO_NAME;
dpPlayerInfo.pwszName = wszServerName;

// Set us up to be non-asynchronous
if( FAILED( hReturn = g_pDPServer->SetServerInfo( &dpPlayerInfo, NULL,
 NULL, DPNSETSERVERINFO_SYNC ) ) ) {
    MessageBox( hWindow, "Failed to SetServerInfo()", "Unknown Error",
     MB_ICONERROR );
    return -1;
}
```

The first six or so lines you see here are for setting up the DPN_PLAYER_INFO data structure. Once that is done, you make the call to SetServerInfo() by passing a pointer to the player info, and then set the synchronous flag. If all goes well, the function does not return a failure notice.

Setting Up the DPN_APPLICATION_DESC Structure

Once you set up the server information, you can go on to describe the application by setting up the DPN_APPLICATION_DESC data structure. You use this structure here just as you did when you hosted an application using the IDirectPlay8Peer interface. The main difference is that the dwFlags member now holds the DPNSESSION_CLIENT_SERVER flag to tell it that the program is a client/server application. Here is the code to set up the application description for a client/server environment.

```
// Set up the application description
DXUtil_ConvertGenericStringToWide( wszSessionName, "DPChat Session" );
ZeroMemory( &dnAppDesc, sizeof(DPN_APPLICATION_DESC) );
dnAppDesc.dwSize              = sizeof(DPN_APPLICATION_DESC);
dnAppDesc.guidApplication     = GUID_CHATSERVER;
dnAppDesc.pwszSessionName     = wszSessionName;
dnAppDesc.dwMaxPlayers        = MAX_PLAYERS;
dnAppDesc.dwFlags             = DPNSESSION_CLIENT_SERVER;
```

If you compare this to the IDirectPlay8Peer examples from the last chapter, you notice that the only real difference is the dwFlags setting I mentioned. That's it for the server application description; now it's time to add some components to the address.

Setting Up the IDirectPlay8Address Interface

Hey, do you remember how you had to create an address interface earlier? Well, now you need to update that address by adding the server's host port number. This port number is just like the listening port when using sockets. The port number is where all incoming client connections are made. To add the port number, you need to use the familiar AddComponent() member function. It works just as it did in the last couple of chapters, so I don't really need to cover it again. Here is the code I use for this particular application.

```
hReturn = g_pDeviceAddress->AddComponent(DPNA_KEY_PORT,&dwPort,sizeof(
DWORD),DPNA_DATATYPE_DWORD);
if( hReturn != S_OK ) {
    MessageBox( hWindow, "Failed to AddComponent()", "hrHostGame()",
     MB_ICONERROR );
    return -1;
}
```

Not much to it; the most notable item is the DPNA_KEY_PORT definition that tells the function you are adding a port number to the address.

The Server's Host() Function

It's time to celebrate; you have made it to the Host() function! All the prerequisites have been covered, so now you get to set up the host connection using the Host() member function. Before you look at the prototype, take a peek at Figure 10.6. It shows you the steps up to and including hosting.

Now that you have seen the figure, here is the prototype.

```
HRESULT Host(
    const DPN_APPLICATION_DESC *const pdnAppDesc,
    IDirectPlay8Address **const prgpDeviceInfo,
    const DWORD cDeviceInfo,
    const DPN_SECURITY_DESC *const pdpSecurity,
    const DPN_SECURITY_CREDENTIALS *const pdpCredentials,
    VOID *const pvPlayerContext,
    const DWORD dwFlags
);
```

This might look like a complicated function, but it really isn't all that bad once you factor in all of the `NULL` values. The first parameter holds the application description data structure you created earlier. The second parameter holds the device address where you just added a port number. The third parameter tells the function how many devices are present in the second parameter. This example has only one device, so pass a `1`.

The fourth and fifth parameters are not used in DirectX 8, so you can leave them `NULL`. The sixth parameter holds the optional user context. The last parameter sets

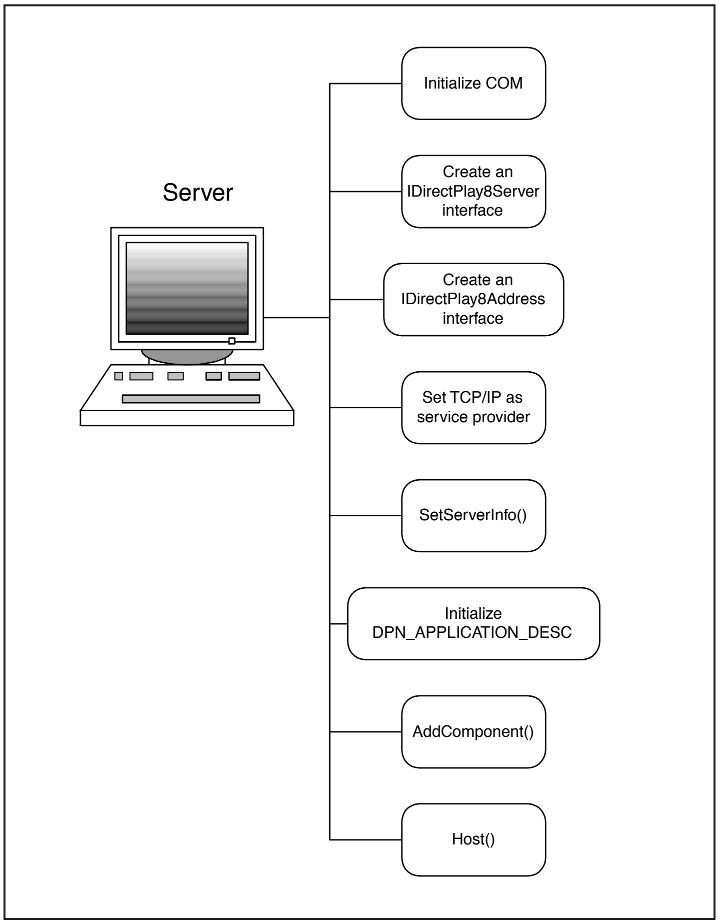

Server
Initialize COM
Create an IDirectPlay8Server interface
Create an IDirectPlay8Address interface
Set TCP/IP as service provider
SetServerInfo()
Initialize DPN_APPLICATION_DESC
AddComponent()
Host()

Figure 10.6

Server process flow up to hosting

certain hosting flags. You use the one flag named DPNHOST_OKTOQUERYFORADDRESSING if you want to allow DirectPlay to query the user for any additional connection information needed. You usually do not want to use this flag for games because it will interfere with your GUI.

That's it for the overview of the function; here is the code I used to call it.

```
hReturn = g_pDPServer->Host(     &dnAppDesc,
                                                  &g_pDeviceAddress,
                                                  1,
                                                  NULL,
                                                  NULL,
                                                  NULL,
                                                  NULL );
```

It's the wonderful world of NULLs. The only complicated values I passed to this function are the device description and the device address. There you have it, the Host() function in all its glory.

Managing Players

Now that your server is up and running, it has to do something with connections that are made to it. The message that you receive when players join is the DPN_MSGID_CREATE_PLAYER message. This works just as it did in the IDirectPlay8Peer examples. Basically, when a player connects to your game, he passes you a creation message. Once you receive this message, you need to add him to your table of active players. Here is the code I use for adding players to a client/server session.

```
// Function called when a player needs to be added to
    the game (including ourselves)
HRESULT     hrCreatePlayer( PVOID pvUserContext, PVOID pMsgBuffer )
{
    HRESULT                  hReturn = S_OK;
    PDPNMSG_CREATE_PLAYER    pCreatePlayerMsg;
    char                     strName[256];
    char                     szOutput[256];
    DWORD                    dwSize = 0;
    DPN_PLAYER_INFO          *pdpPlayerInfo = NULL;
    int                      i;

    // Get a create message pointer to the buffer
    pCreatePlayerMsg = (PDPNMSG_CREATE_PLAYER)pMsgBuffer;
```

```
// Get the peer info and extract its name
hReturn = g_pDPServer->GetClientInfo(
 pCreatePlayerMsg->dpnidPlayer, pdpPlayerInfo, &dwSize, 0 );
if( FAILED(hReturn) && hReturn != DPNERR_BUFFERTOOSMALL ) {
    if( hReturn == DPNERR_INVALIDPLAYER ) {
        vShowText(hLB_Output,"Adding Ourselves");
    }
    hReturn = -1;
}
else {
    pdpPlayerInfo = (DPN_PLAYER_INFO*) new BYTE[ dwSize ];
    ZeroMemory( pdpPlayerInfo, dwSize );
    pdpPlayerInfo->dwSize = sizeof(DPN_PLAYER_INFO);
    hReturn = g_pDPServer->GetClientInfo(
     pCreatePlayerMsg->dpnidPlayer, pdpPlayerInfo, &dwSize, 0 );
    if( FAILED(hReturn) ) {
        vShowText(hLB_Output,"Error Getting Client Info");
        hReturn = -1;
    }
    else {
        EnterCriticalSection( &g_csModifyPlayer );
        // Convert player name to ANSI
        DXUtil_ConvertWideStringToGeneric( strName,
         pdpPlayerInfo->pwszName );
        // Add player to list
        for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
            if( !PlayerInfo[i].bActive ) {
                PlayerInfo[i].bActive = 1;
                PlayerInfo[i].dpnidPlayer =
                 pCreatePlayerMsg->dpnidPlayer;
                strcpy(PlayerInfo[i].szPlayerName,strName);
                break;
            }
        }
        // Check whether no free slot found
        if( i == MAX_PLAYERS ) {
            vShowText(hLB_Output,"No free slots in game!");
        }
        // Check whether we are adding ourselves
        else if( pdpPlayerInfo->dwPlayerFlags & DPNPLAYER_LOCAL ) {
            g_dpnidLocalPlayer = pCreatePlayerMsg->dpnidPlayer;
```

```
                    sprintf(szOutput,"<Slot%d> Added Ourselves",i);
                    vShowText(hLB_Output,szOutput);
                }
                else {
                    sprintf(szOutput,"<Slot%d><%s> Is In The Game",i,strName);
                    vShowText(hLB_Output,szOutput);
                }
                SAFE_DELETE_ARRAY( pdpPlayerInfo );
                // Update the number of active players in a thread-safe way
                if( i != MAX_PLAYERS )
                    InterlockedIncrement( &g_lNumberOfActivePlayers );
                LeaveCriticalSection( &g_csModifyPlayer );
            }
        }
    return hReturn;
}
```

The DPNMSG_CREATE_PLAYER Message

The first thing you have to do with this function is turn the raw message data into something usable. Casting the raw message into a DPNMSG_CREATE_PLAYER message does this. By doing this, you can access the DPNID of the player who is being created. Once you have the DPNID, you can call the GetClientInfo() member function of the IDirectPlay8Server interface.

The GetClientInfo() Member Function

Do you remember the GetPeerInfo() member function of the IDirectPlay8Peer interface? The GetClientInfo() function is just like that one, except this one is for servers. Here is the prototype.

```
HRESULT GetClientInfo(
    const DPNID dpnid,
    DPN_PLAYER_INFO *const pdpnPlayerInfo,
    DWORD *const pdwSize,
    const DWORD dwFlags
);
```

The first parameter passes in the DPNID of the client about whom you want to retrieve information. This is where you pass in the DPNID you just got from the create player message. The second parameter expects the address of a DPN_PLAYER_INFO data structure to hold the soon-to-be-retrieved player information.

The third parameter receives the size of the information retrieved. If the receive buffer is too small, then a DPNERR_BUFFERTOOSMALL message is returned. When you receive this message, you can then allocate memory equal to the size returned and make the GetClientInfo() call again. This is the normal process.

The last parameter describes the information returned for the client. DPNINFO_NAME and DPNINFO_DATA are the only available flags. Normally, you can leave these set to NULL.

TIP

Don't forget to use critical sections when modifying global data that can be utilized by multiple threads. You don't want other threads to use the data at the same time.

If you review the code, you'll see that I called GetClientInfo() to receive the size of data to retrieve. Once I received the size of the data, I allocated enough memory for the client information. I made a second call to GetClientInfo() with the newly allocated receive buffer. If no errors occur, the application proceeds to enter the player data critical section.

Adding the Player

Once inside the critical section, I then loop through the list of character information and look for a non-active player slot. If an empty slot is found, the program activates that slot and loads it with the connecting player's information. If all slots are full, then no players are added.

Freeing the DPN_PLAYER_INFO data structure happens next, and then the application exits the modify player critical section before leaving the function. One step that happens is incrementing active players, but it is not a necessary step.

Destroying Players

Not only do you receive create player messages, but you also receive destroy player messages. These are received whenever a player leaves the session for whatever reason. The message type you need to look for is DPN_MSGID_DESTROY_PLAYER. In the current example, I look for this message and then call my hrDestroyPlayer() function to process it.

This function is a little easier to deal with because you already know the player's information from the creating stage. Here is the code I use to destroy players.

```
HRESULT     hrDestroyPlayer( PVOID pvUserContext, PVOID pMsgBuffer )
{
    PDPNMSG_DESTROY_PLAYER          pDestroyPlayerMsg;
    HRESULT                         hReturn = S_OK;
    int                             i;
    char                            szOutput[256];

    // Get a destroy message pointer to the buffer
    pDestroyPlayerMsg = (PDPNMSG_DESTROY_PLAYER)pMsgBuffer;
    // Update the number of active players in a thread-safe way
    InterlockedDecrement( &g_lNumberOfActivePlayers );
    EnterCriticalSection( &g_csModifyPlayer );
    // Remove player from list
    for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
        if( PlayerInfo[i].bActive ) {
            if( PlayerInfo[i].dpnidPlayer ==
             pDestroyPlayerMsg->dpnidPlayer ) {
                PlayerInfo[i].bActive = 0;
                sprintf(szOutput,"<Slot%d><%s> Left
                 The Game",i,PlayerInfo[i].szPlayerName);
                vShowText(hLB_Output,szOutput);
                break;
            }
        }
    }
    LeaveCriticalSection( &g_csModifyPlayer );
    return(hReturn);
}
```

The DPNMSG_DESTROY_PLAYER Message

To access the appropriate data, I first cast the raw message data into a DPNMSG_DESTROY_PLAYER message. This allows the application to find the DPNID of the player to destroy. After this is handled, I then enter the modify player critical section to keep anyone else from messing with the player data while this thread needs to modify it.

The code loops through all active player slots and looks for a matching DPNID. Once one is found, the slot is set to inactive so that the next player who comes along can use it.

Before exiting this function, I leave the critical section and decrement the number of active players. Take a look at Figure 10.7 to see the flow of the server program during this process.

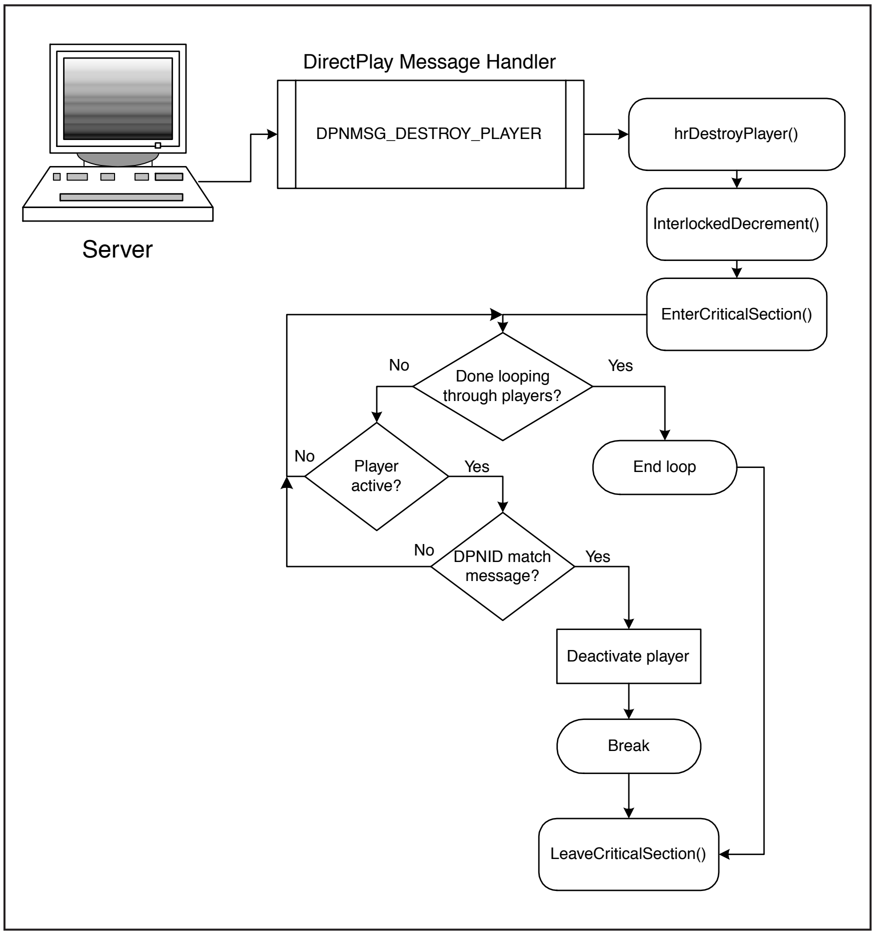


Figure 10.7

The DPChat server program deleting a player

Receiving Client Messages

You are almost done with the server! The only necessary question left is how you receive messages from clients. Processing the DPN_MSGID_RECEIVE messages allows you to receive messages from clients, and here is the code I use to do this.

```
case DPN_MSGID_RECEIVE:
{
    PDPNMSG_RECEIVE pReceiveMsg;
    PacketGeneric     *PGen;

    pReceiveMsg = (PDPNMSG_RECEIVE)pMsgBuffer;
    PGen = (PacketGeneric*)pReceiveMsg->pReceiveData;
    // Ignore the packet if we sent it originally
    if( pReceiveMsg->dpnidSender == g_dpnidLocalPlayer )
        break;
    // If it is a chat packet, send it out to all of the players
    if( PGen->dwType == PACKET_TYPE_CHAT ) {
        void                    *packet;
        PacketChat          *ChatMsg;
        // Convert the packet to a void stream
        ChatMsg = (PacketChat*)pReceiveMsg->pReceiveData;
        packet = (VOID*)ChatMsg;
        // Send the chat packet to all clients
        hrSendServerMessage(DPNID_ALL_PLAYERS_GROUP, PACKET_TYPE_CHAT,packet);
        // Output it on our screen
        vShowText(hLB_Output,ChatMsg->szText);
    }
    break;
}
```

The `DPN_MSGID_RECEIVE` message is received whenever any client sends a non-system message to the server. Once you have the message, you usually want to check it to see what type it is. DirectPlay has no built-in mechanism to differentiate client message types, so you have to create your own. What I did is create a `PacketGeneric` class object to identify different message types.

The PacketGeneric Class

This class really is nothing more than a glorified data structure. In fact, I don't even have functions for it. But in C++, structures are really stored as classes. So it doesn't matter which one you use as long as there are no members and all of the data is public. Anyway, here is the class structure I use for my messages.

```
class     PacketGeneric
{
    public:
        DWORD     dwType;
        DWORD     dwSize;
};
```

Ooh, big deal, right? The first member variable, `dwType`, stores the message type identifier. The second member stores the size of the packet. Because this class is generic in nature, it is easy to create new derived message types from it. Take, for example, the chat packet I created in the program. It is just a class that has been derived from the generic packet class.

The DPNMSG_RECEIVE Message Type

Before you can use the receive message, you must turn it into something readable. You do this by casting the raw message data into a DPNMSG_RECEIVE message, like so:

```
pReceiveMsg = (PDPNMSG_RECEIVE)pMsgBuffer;
```

Once that is accomplished, you can cast that message into a custom generic packet. Pointing a PacketGeneric pointer to the pReceiveData member of the DPNMSG_RECEIVE buffer does this.

```
PGen = (PacketGeneric*)pReceiveMsg->pReceiveData;
```

One thing I do before moving on to check the generic packet is check whether the server is the one who sent the message. Because you don't want to process your own messages, it is usually best to break out of the process and just ignore the message. You perform this check by comparing the DPNID of the message sender to the server's DPNID.

Now that you have a PacketGeneric object to look at, you can check its message type to determine what to do with it. This step is as easy as checking the dwType member versus known message types. Here is the code I use to check for a chat packet.

```
if( PGen->dwType == PACKET_TYPE_CHAT ) {
…
}
```

The Chat Packet

The PACKET_TYPE_CHAT definition is one I created and coded in the DPChatServer.h file. You want to assign a number to every packet type you create. You can also enumerate the types if you want to, but I like to keep it simple with straight define calls.

Here is the prototype for my chat packet class.

```
class     PacketChat : public PacketGeneric
{
    public:
        char     szText[128];
};
```

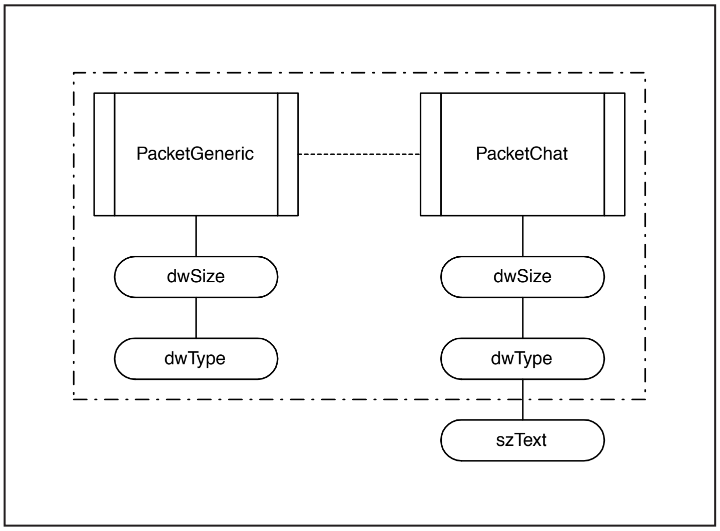


Figure 10.8

Layout of the `PacketChat` *class*

The chat packet class is derived from the `PacketGeneric` class. This allows you to use the members in the generic class. For a visual of this, take a look at Figure 10.8.

The only member unique to the packet class is the `szText` character buffer, which stores the actual chat message. If the packet type received is a chat packet, then you need to cast the packet into the chat packet class format so that you can read the text enclosed within it.

```
PacketChat     *ChatMsg;
ChatMsg = (PacketChat*)pReceiveMsg->pReceiveData;
```

What I have done here is create a pointer to a `PacketChat` class object. I then set it equal to the data received from the client. Once this is done, you can look at the chat text by referencing the `ChatMsg->szText` member.

Broadcasting the Chat Message

Because the server is responsible for all communication, it now needs to send the chat message out to every client so that they can see it. If the server didn't broadcast the message, then only the server would see it. Most chat programs are used for chatting between clients, so I'd better show you how!

To facilitate sending messages from the server, I created a function called `hrSendServerMessage()`.

The hrSendServerMessage() Function

The `hrSendServerMessage()` function sends any type of packet to a single client or multiple clients. Here is the prototype.

```
HRESULT     hrSendServerMessage(
    int player,
    DWORD dwMessageType,
    PVOID pMsgBuffer );
```

The first parameter points to the slot number to which you want to send the message. You might recall that when you create a player, you assign him to a slot number within your list of available players. This is the number used here. As an alternative, you can pass `DPNID_ALL_PLAYERS_GROUP` to this parameter, and it sends the message to all clients.

The second parameter passes the message type you want to send. This message type references the list of defined types in DPChatServer.h. The third parameter is a void pointer to the contents of the message. Using a void pointer, you can pass any type of data to this parameter.

The Function Body

Now that you have seen the prototype, it is time to look at the code for the function.

```
HRESULT     hrSendServerMessage( int player, DWORD dwMessageType,
 PVOID pMsgBuffer )
{
    DPNHANDLE                   hAsync;
    DWORD                       dwLength;
    DPN_BUFFER_DESC             bufferDesc;
    PacketGeneric               *PGen;

    // Cast to a generic packet to get the size
    PGen = (PacketGeneric*)pMsgBuffer;
    dwLength = PGen->dwSize;
    if( dwLength == 0 )
        return(0);
    bufferDesc.dwBufferSize = dwLength;
    bufferDesc.pBufferData  = (BYTE*)pMsgBuffer;
    // Broadcast it if player set to zero
    if( player == DPNID_ALL_PLAYERS_GROUP )
        g_pDPServer->SendTo( DPNID_ALL_PLAYERS_GROUP, &bufferDesc, 1, 0,
         NULL, &hAsync, DPNSEND_NOLOOPBACK );
    else
        g_pDPServer->SendTo( PlayerInfo[player].dpnidPlayer,
         &bufferDesc, 1, 0, NULL, &hAsync, 0 );

    return S_OK;
}
```

The first order of business is to cast the message buffer into a generic packet to read the size of the packet. DirectPlay needs to know the size of the message to send, so this is a necessary evil. If the packet length is zero, then you need to exit out of the function. You do this because you don't want to send empty messages.

The DPN_BUFFER_DESC Data Structure

The next thing you do is start loading a `DPN_BUFFER_DESC` data structure with the packet information. This is a DirectPlay structure used to store generic buffer information. For the current situation, you use this buffer to send a message. To set up the buffer, you first set the length with the `dwBufferSize` member. You then point the message data to the `pBufferData` member variable.

Now that the buffer is set up, you can go ahead and send the message using the `SendTo()` function.

The SendTo() Member Function

The `SendTo()` function belongs to the `IDirectPlay8Server` interface and sends any type of message to one or more clients. Here is the prototype for the function.

```
HRESULT SendTo(
    const DPNID dpnid,
    const DPN_BUFFER_DESC *const pBufferDesc,
    const DWORD cBufferDesc,
    const DWORD dwTimeOut,
    void *const pvAsyncContext,
    DPNHANDLE *const phAsyncHandle,
    const DWORD dwFlags
);
```

The first parameter expects the DPNID of the client to whom you want to send the message. Alternatively, you can pass the DPNID_ALL_PLAYERS_GROUP definition here to send the message to all clients.

The second parameter passes a pointer to the DPN_BUFFER_DESC data structure with which you set up the buffer information. This is what contains the actual message data. The third parameter specifies how many message buffers you are sending with this call.

TIP

DirectX 8 only supports sending one buffer at a time. Always set the third parameter in the SendTo() function to 1 if you are using this version of DirectX.

The fourth parameter specifies a timeout in milliseconds for the message to be transmitted. If you don't want the message to ever give up, then you can set this to 0. The fifth parameter points to user-supplied context. I don't use it at this time, so you can leave this set to NULL. The sixth parameter is used to get access to an async handle that the function will return. You can use this handle to cancel the send operation. If you use the DPNSEND_SYNC flag in this function call, you must set this parameter to NULL. The last parameter is used for various flags. Table 10.2 contains a list of available flags.

Well, you have seen the function, so here is a quick look at how I used it in the server sending function.

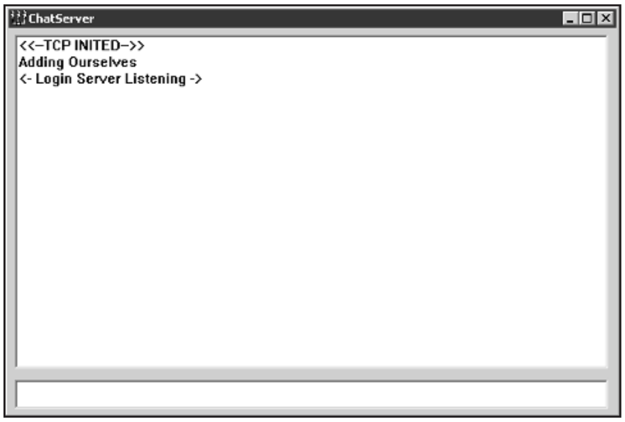
```
g_pDPServer->SendTo( PlayerInfo[player].dpnidPlayer, &bufferDesc,
 1, 0, NULL, &hAsync, 0 );
```

I pass the DPNID of the client to receive the message. I then pass in the buffer description, the number of buffers, an infinite timeout setting, no context, the async handle, and no flags. The only thing I do differently when broadcasting to everyone is that I send DPNID_ALL_PLAYERS_GROUP as the DPNID. I also set the DPNSEND_NOLOOPBACK flag so that I don't receive my own message.

Before I move on to programming the client, I think it's time to see the DPChatServer program in action, so take a look at Figure 10.9.

Table 10.2 IDirectPlay8Server::SendTo() Flags

Flag	Purpose
DPNSEND_GUARANTEED	Guarantees message delivery. This can slow down message sending quite a bit.
DPNSEND_NOLOOPBACK	Keeps the server from receiving its own messages.
DPNSEND_PRIORITY_LOW	Queues the message with low priority.
DPNSEND_PRIORITY_HIGH	Queues the message with high priority.
DPNSEND_NOCOPY	Keeps DirectPlay from making a copy of the DPN_BUFFER_DESC before sending the message. You cannot use DPNSEND_NOCOMPLETE in conjunction with this flag.
DPNSEND_SYNC	Processes the send synchronously.
DPNSEND_NOCOMPLETE	Keeps a DPN_MSGID_SEND_COMPLETE message from being generated. You cannot use the DPNSEND_NOCOPY or DPNSEND_GUARANTEED flags with this flag. The async context must be set to NULL also.
DPNSEND_NONSEQUENTIAL	Lets messages arrive at the client's computer in any possible order. If you set this flag, then messages are not guaranteed to show up in order. If you do not set this flag, then DirectPlay automatically puts messages in order. Only use this flag for non-critical packets.
DPNSEND_COMPLETEONPROCESS	Causes DirectPlay to send a DPN_MSGID_SEND_COMPLETE message upon completion of a message being received by the client. You have to set the DPNSEND_GUARANTEED flag in conjunction with this flag. This process can slow down message sending immensely. Only use it for absolutely critical messages.



```
<<—TCP INITED—>>
Adding Ourselves
<- Login Server Listening ->
```

Figure 10.9

Output from the DPChatServer program

The IDirectPlay8Client Interface

The `IDirectPlay8Client` interface is a bit simpler than the server interface because the client doesn't have to mess with hosting or managing players. Go ahead and load up the DPChatClient workspace located in the Chapter10 Zip file on the CD-ROM. It contains the source code that I'll discuss.

The `IDirectPlay8Client` interface works just like all of the other DirectPlay interfaces in that it is a COM object. The first thing you need to do is call the `CoInitialize()` function to initialize COM. If you look under the `hrInitializeDirectPlay()` function in my program, you'll see that this is the first call made. Better yet, why don't I go ahead and list the code here?

```
HRESULT hrInitializeDirectPlay( HWND hWindow )
{
    HRESULT     hReturn;

    // Initialize COM
    hReturn = CoInitialize( NULL );
    if( FAILED(hReturn) ) {
        MessageBox( hWindow, "Error Initializing COM",
         "DirectPlay Error", MB_ICONERROR );
        return hReturn;
    }
    // Create IDirectPlay8Client Object
```

```
if( FAILED( hReturn = CoCreateInstance( CLSID_DirectPlay8Client,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_IDirectPlay8Client,
    (LPVOID*) &g_pDP ) ) )
    MessageBox( hWindow, "Can't Create DPlayClient",
     "DirectPlay Error", MB_ICONERROR );
// Init IDirectPlay8Client Message Handler
if( FAILED( hReturn = g_pDP->Initialize( NULL,
 DirectPlayMessageHandler, 0 ) ) ) {
    MessageBox( hWindow, "Failed to Message Handler",
     "DirectPlay Error", MB_ICONERROR );
    return -1;
}
// Create a device address
hReturn = CoCreateInstance( CLSID_DirectPlay8Address,
 NULL,CLSCTX_INPROC_SERVER, IID_IDirectPlay8Address, (LPVOID*)
 &g_pDeviceAddress );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Failed to Create Device",
     "CoCreateInstance()", MB_ICONERROR );
    return -1;
}
// Set our service provider to TCP/IP
if( FAILED( hReturn = g_pDeviceAddress->SetSP(
 &CLSID_DP8SP_TCPIP ) ) ) {
    MessageBox( hWindow, "Failed to SetSP() for Device Address",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
// Create a host address
hReturn = CoCreateInstance( CLSID_DirectPlay8Address,
 NULL,CLSCTX_INPROC_SERVER, IID_IDirectPlay8Address,
 (LPVOID*) &g_pHostAddress );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Failed to Create Host Address()",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
// Set the host address to TCP/IP
if( FAILED( hReturn = g_pHostAddress->SetSP(
 &CLSID_DP8SP_TCPIP ) ) ) {
```

```
            MessageBox( hWindow, "Failed to SetSP() for Host Address",
             "Invalid Param", MB_ICONERROR );
            return -1;
        }
        // Create connection complete event for later use
        g_hConnectCompleteEvent = CreateEvent( NULL, FALSE, FALSE, NULL );
        vShowText(hLB_Output,"<<--TCP INITED-->>");
        return S_OK;
}
```

Creating the IDirectPlay8Client Function

As I mentioned a minute ago, the first thing to accomplish is initializing COM. Once that is out of the way, you then create an `IDirectPlay8Client` interface by calling the `CoCreateInstance()` function. This works just as it did when you created the server interface earlier in the chapter.

Initializing the Client Interface

The next function called is the `IDirectPlay8Client::Initialize()` function, which sets where all of the messages from the server will go. For this function, I pass in the address of the `DirectPlayMessageHandler()` function.

Setting Up the Client Addresses

Clients require two `IDirectPlay8Address` interfaces. The first one identifies the client's connection device. The second address identifies the host computer. Both of these addresses get set to use the TCP/IP service provider with the `SetSP()` member function.

The Connection Complete Event

You might remember the code from the last chapter where you created a peer application that connects to a host. To detect that the connection was done, you used an event. Well, the same thing happens here. To tell when the client is done connecting to the server, you use a connection event. The creation of this event is the last notable thing the `hrInitializeDirectPlay()` function accomplishes before exiting. Refer to Figure 10.10 to see the list of steps the client follows in the initialization routine.

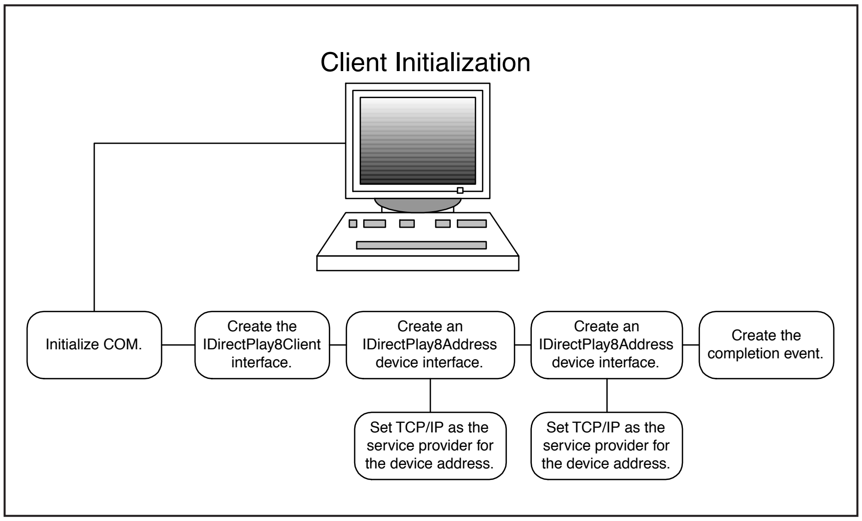

Client Initialization
Initialize COM.
Create the IDirectPlay8Client interface.
Create an IDirectPlay8Address device interface.
Set TCP/IP as the service provider for the device address.
Create an IDirectPlay8Address device interface.
Set TCP/IP as the service provider for the device address.
Create the completion event.

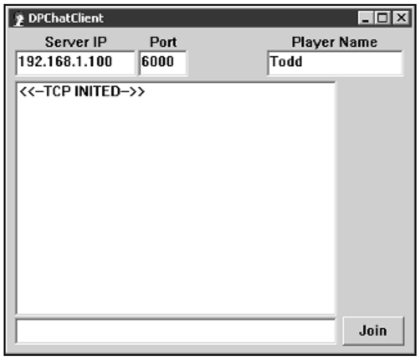
Figure 10.10

Client steps used for initialization

I know that I kind of just flew right through the initialization routine, but I hope you understand what it is doing by now. All of the steps taken have appeared in past chapters. Anyway, go ahead and look over the code to make sure you understand fully what is going on.

Joining the Server's Session

Hey, cool; you are already to the part about joining the server's game session! So far, you have initialized the client's system, and now the program is just sitting there, waiting for someone to click on the Join button. The program in all of its non-glory looks like Figure 10.11.



```
192.168.1.100   6000   Todd
<<--TCP INITED-->>
```

Figure 10.11

The client program before joining a session

Once the user clicks on the Join button, a message is sent to the program's message pump, telling it that someone clicked a button. The message handler loop receives this message and calls the `hrJoinGame()` function. This set of events appears in the following code.

```
LRESULT CALLBACK fnMessageProcessor ( HWND hWnd, UINT iMsg,
 WPARAM wParam, LPARAM lParam )
{
    char           szMessage[256];
    char           szCompMessage[256];
    char           szMyName[32];
    HRESULT        hReturn;
    void           *packet;
    PacketChat     ChatMsg;

    switch (iMsg) {
        case WM_COMMAND: {
            // Check for child window messages
            switch(LOWORD(wParam)) {
                // Check whether the user clicked the button
                case IDC_hBU_Join:
                    hrJoinGame( hWnd );
                    break;
            }
            …
        }
    }
}
```

The hrJoinGame() Function

The `hrJoinGame()` function is one I wrote to establish a connection with the server. Here is the code for this function.

```
HRESULT hrJoinGame( HWND hWnd )
{
    HRESULT                         hReturn = S_OK;
    WCHAR                           wszHostName[256];
    WCHAR                           wszClientName[256];
    char                            szClientName[256];
    char                            szIP[256];
    char                            szPort[256];
```

```
DWORD                           dwPort;
DWORD                           dwLength = 256;
DPN_APPLICATION_DESC            dpnAppDesc;
DPN_PLAYER_INFO                 dpPlayerInfo;

vShowText(hLB_Output,"Attempting Connection...");
// Set the client info
GetWindowText(hEB_InputName,szClientName,36);
// Get name from window edit box
DXUtil_ConvertGenericStringToWide( wszClientName, szClientName );
ZeroMemory( &dpPlayerInfo, sizeof(DPN_PLAYER_INFO) );
dpPlayerInfo.dwSize = sizeof(DPN_PLAYER_INFO);
dpPlayerInfo.dwInfoFlags = DPNINFO_NAME;
dpPlayerInfo.pwszName = wszClientName;
// Make this a synchronous call
if( FAILED( hReturn = g_pDP->SetClientInfo( &dpPlayerInfo, NULL,
 NULL, DPNSETCLIENTINFO_SYNC ) ) ) {
    vShowText(hLB_Output,"Failed to set client info");
    return -1;
}
// Prepare the application description
ZeroMemory( &dpnAppDesc, sizeof( DPN_APPLICATION_DESC ) );
dpnAppDesc.dwSize = sizeof( DPN_APPLICATION_DESC );
dpnAppDesc.guidApplication = GUID_CHATSERVER;
// Get IP from edit box
GetWindowText(hEB_InputServerIP,szIP,32);
// Get port from edit box
GetWindowText(hEB_InputServerPort,szPort,6);
// Convert the IP to a wide string
DXUtil_ConvertGenericStringToWide( wszHostName, szIP );
// Convert the port string to a DWORD
dwPort = atol(szPort);
// Add hostname to address
hReturn = g_pHostAddress->AddComponent(DPNA_KEY_HOSTNAME,wszHostName,
 (wcslen(wszHostName)+1)*sizeof(WCHAR),DPNA_DATATYPE_STRING);
if( hReturn != S_OK ) {
    MessageBox( hWnd, "Failed to AddComponent()", "hrJoinGame()", MB_ICONERROR );
    return -1;
}
// Add port number to address
hReturn = g_pHostAddress->AddComponent(DPNA_KEY_PORT,&dwPort,
     sizeof(DWORD),DPNA_DATATYPE_DWORD);
    if( hReturn != S_OK ) {
        MessageBox( hWnd, "Failed to AddComponent()", "hrJoinGame()",
         MB_ICONERROR );
        return -1;
    }
    // Connect to the session
    hReturn = g_pDP->Connect(     &dpnAppDesc,
                                       g_pHostAddress,
                                       g_pDeviceAddress,
                                       NULL,
                                       NULL,
                                       NULL,
                                       NULL,
                                       NULL,
                                       &g_hConnectAsyncOp,
                                       NULL); // flags

    if( hReturn != E_PENDING && FAILED(hReturn) ) {
        vShowText(hLB_Output,"Failed to Connect");
        return -1;
    }
    SetTimer( hWnd, TIMERID_CONNECT_COMPLETE, 100, NULL );
    return(hReturn);
}
```

The majority of this function works just like the `hrJoinGame()` function I covered in the last chapter. The first thing you have to do is set up the information about the player who is joining. With the client/server model, you do this using the `SetClientInfo()` member function.

The SetClientInfo() Function

The `SetClientInfo()` function belongs to the `IDirectPlay8Client` interface and sets the general information about the client. The peer counterpart is the `SetPeerInfo()` function. Here is the prototype for the function.

```
HRESULT SetClientInfo(
    const DPN_PLAYER_INFO *const pdpnPlayerInfo,
    PVOID const pvAsyncContext,
    DPNHANDLE *const phAsyncHandle,
    const DWORD dwFlags
);
```

The first parameter expects an initialized DPN_PLAYER_INFO data structure. This data structure stores the player name and other information about the client. If you look at the hrJoinGame() function, you see that I set the dwInfoFlags to equal DPNINFO_NAME. This tells DirectPlay that the structure contains a client name.

The second parameter allows you to specify an async context to use for the player. I leave this at NULL because I am not using a context.

The third parameter is set up to receive a handle that lets you cancel this operation. DirectX 8 does not let you cancel this operation at this time, though, so you should always set this to NULL.

The fourth parameter lets you set up various flags for the structure. DirectX 8 only has one flag at this time, DPNSETCLIENTINFO_SYNC, which tells the function to process synchronously. I use this flag here because I don't want the program to continue until it has finished.

The Client's Application Description

The next item of business is setting up the client's application description. The most important part of setting up the DPN_APPLICATION_DESC data structure is assigning the GUID. You do this by setting up the guidApplication member variable of the application description data structure. This is how you differentiate between the servers to which you are trying to connect. If your client's application GUID does not match the server's, then a connection cannot be made.

Setting Up the Host's Address

Onward ho! Okay, the next piece of work is setting up the host's address interface. You already created the interface back in the hrInitializeDirectPlay() function, so now it is time to add the host's IP address and port number. You do so with the AddComponent() member function of the IDirectPlay8Address interface.

If you take a look back at the code, you'll notice that I am adding two components, the address and the port number. The defines I use for this are the DPNA_KEY_HOSTNAME and DPNA_KEY_PORT presets.

TIP

You can find all of the definitions for the AddComponent() member function in the dpaddr.h header file that comes with the DirectX 8 SDK.

The Connect() Function

The light flickers at the end of the tunnel now; you are at the actual connection process. Client interfaces use the `Connect()` function to establish a connection with a server. Here is the prototype.

```
HRESULT Connect(
    const DPN_APPLICATION_DESC *const pdnAppDesc,
    IDirectPlay8Address *const pHostAddr,
    IDirectPlay8Address *const pDeviceInfo,
    const DPN_SECURITY_DESC *const pdnSecurity,
    const DPN_SECURITY_CREDENTIALS *const pdnCredentials,
    const void *const pvUserConnectData,
    const DWORD dwUserConnectDataSize,
    void *const pvAsyncContext,
    DPNHANDLE *const phAsyncHandle,
    const DWORD dwFlags
);
```

It might look like quite a few parameters, but don't worry because many of them are set to `NULL` for your current purposes. The first parameter expects the `DPN_APPLICATION_DESC` structure that you set up earlier to describe the application. The second parameter expects the host address interface where you added the IP and port number earlier. The third parameter expects the device address for the client's connection that you defined back in the `hrInitializeDirectPlay()` function. The fourth and fifth parameters apply to future security functionality. The keyword here is "future," so leave these set to `NULL`.

The sixth parameter lets you send additional connect information to the server. If you specify this data, then the server receives a `DPN_MSGID_INDICATE_CONNECT` message. Because this is optional, you can leave it as `NULL`. The seventh parameter specifies the size of the data you passed in the sixth parameter. Because I am not passing any data, I leave this `NULL`. The eighth parameter lets you specify user-created context to be passed when the `DPN_MSGID_CONNECT_COMPLETE` message is processed. The ninth parameter receives an asynchronous handle, which must be set to `NULL` if you use the `DPNCONNECT_SYNC` flag.

The last parameter lets you specify certain flags, such as the `DPNCONNECT_OKTOQUERYFORADDRESSING` flag that tells DirectPlay it is okay to query for additional connection information. The other flag, `DPNCONNECT_SYNC`, tells DirectPlay to process the connection synchronously. Refer to Figure 10.12 to see the client steps used so far.

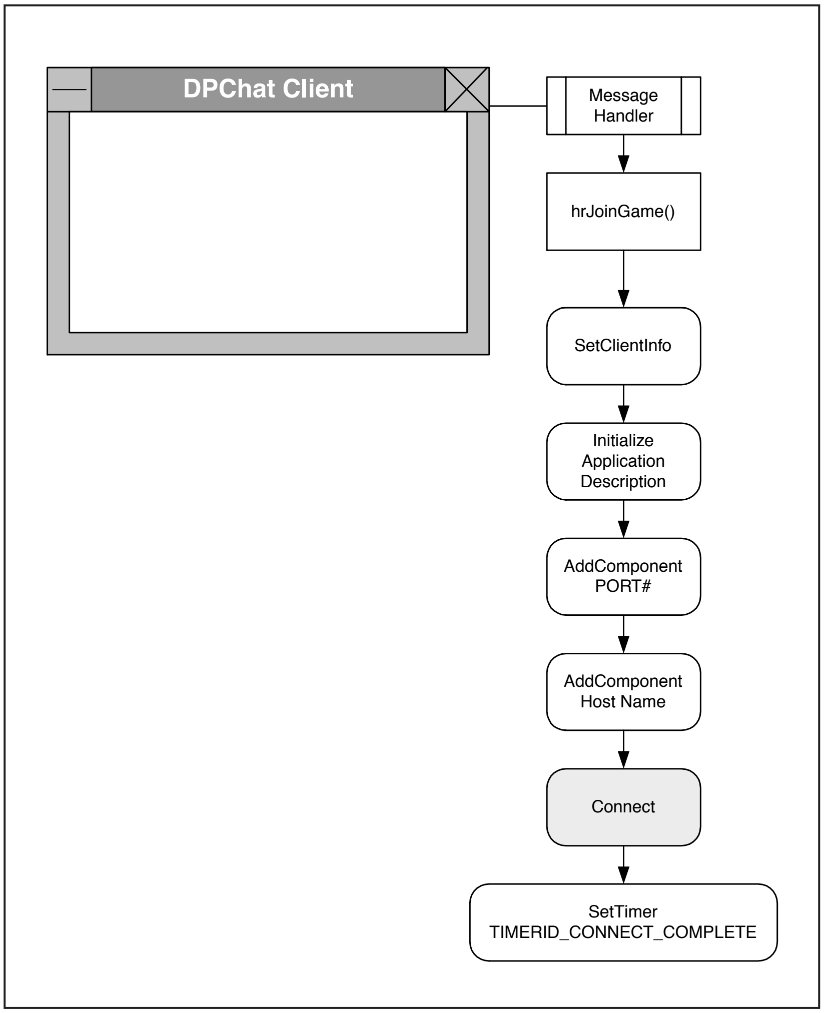


Figure 10.12

The client program flow up to connecting

Setting the Connection Timer

The `Connect()` function is out of the way, so now you get to set up the connection timer to go off every 1/10th of a second. By doing this, you tell the system to send a `WM_TIMER` message to the main message loop every 1/10th of a second. This keeps the program from having to sit in a processing-intensive loop waiting for a connection. Now take a look at the `fnMessageProcessor()` function and locate the `WM_TIMER` case.

Checking the g_hConnectCompleteEvent Event

The following piece of code shows how you can wait for the completion event to take place.

```
case WM_TIMER:
    if( wParam == TIMERID_CONNECT_COMPLETE ) {
        // Check whether the message is telling us our connection is complete
        if( WAIT_OBJECT_0 == WaitForSingleObject( g_hConnectCompleteEvent, 0 ) ) {
            if( FAILED( g_hrConnectComplete ) ) {
                vShowText(hLB_Output,"FAILED");
            }
            else
                vShowText(hLB_Output,"CONNECTED");
            KillTimer( hWnd, TIMERID_CONNECT_COMPLETE );
        }
    }
    break;
```

Basically, this chunk of code is processed every time the message loop gets a `WM_TIMER` function. Because you set a timer earlier to go off every 1/10th of a second, this piece of code is executed that often. The first thing you do here is check whether the timer is equal to the `TIMERID_CONNECT_COMPLETE` timer you created earlier. After that, you check whether the completion event has occurred by using the `WaitForSingleObject()` function.

The `WaitForSingleObject()` function returns a `WAIT_OBJECT_0` if the event passed in has completed. After that is done, all you have to do is check whether the `g_hrConnectComplete` flag is set to success or failure.

One question you might have is, how does the completion event take place? Well, it happens in the DirectPlay message loop located in the `DirectPlayMessageHandler()` function you created earlier.

Setting the g_hConnectCompleteEvent Event

Move on over to the `DirectPlayMessageHandler()` function, and you'll find the `DPN_MSGID_CONNECT_COMPLETE` message. The client receives this message once the server has accepted the connection. When you get this message, you should signal the `g_hConnectCompleteEvent` event with the `SetEvent()` function. But before you do this, it is best to set the return state of the connection to find out whether the server accepted or rejected the connection. Here is the code I used to do all of this.

```
case DPN_MSGID_CONNECT_COMPLETE:
{
      PDPNMSG_CONNECT_COMPLETE pConnectCompleteMsg;
      pConnectCompleteMsg = (PDPNMSG_CONNECT_COMPLETE)pMsgBuffer;
      g_hrConnectComplete = pConnectCompleteMsg->hResultCode;
      SetEvent( g_hConnectCompleteEvent );
      break;
}
```

As you can see here, I first point the raw message data received to a DPNMSG_CONNECT_COMPLETE message type. This allows me to look at the return code from the server.

Once this is out of the way, the next WM_TIMER message that gets processed finds out that the completion event has transpired. This results in a connection complete message. That's it; you are done with connecting a client to a server!

Sending Messages to the Server

Now that you are connected to the server, I think it is probably a good idea to figure out how to send chat messages to everyone. You use the hrSendClientMessage() function I provide in the DPChatClient.cpp code. Oh, yeah; the client looks like Figure 10.13 once a connection is established.

The main message loop looks for an EN_UPDATE message. Once one is received, the code checks whether the user pressed the Enter key. This kicks off the code to send a message to the server. Here is the code in the message loop.

```
// Get the text from the edit box
GetWindowText(hEB_InputField,szMessage,256);
// Check whether he pressed Enter
if( szMessage[strlen(szMessage)-1] == 10 ) {
      // Get rid of trailing garbage
      szMessage[strlen(szMessage)-2] = '\0';
      GetWindowText(hEB_InputName,szMyName,32);
      sprintf(szCompMessage,"<%s> %s", szMyName, szMessage);
      // Clear input field
      SetWindowText(hEB_InputField,"");
      // Send a chat packet to the server
      ChatMsg.dwSize = sizeof(PacketChat);
```

```
        ChatMsg.dwType = PACKET_TYPE_CHAT;
        strcpy(ChatMsg.szText,szCompMessage);
        // Convert the packet to a void stream
        packet = (VOID*)&ChatMsg;
        // Send the chat packet
        hReturn = hrSendClientMessage(PACKET_TYPE_CHAT,packet);
}
```

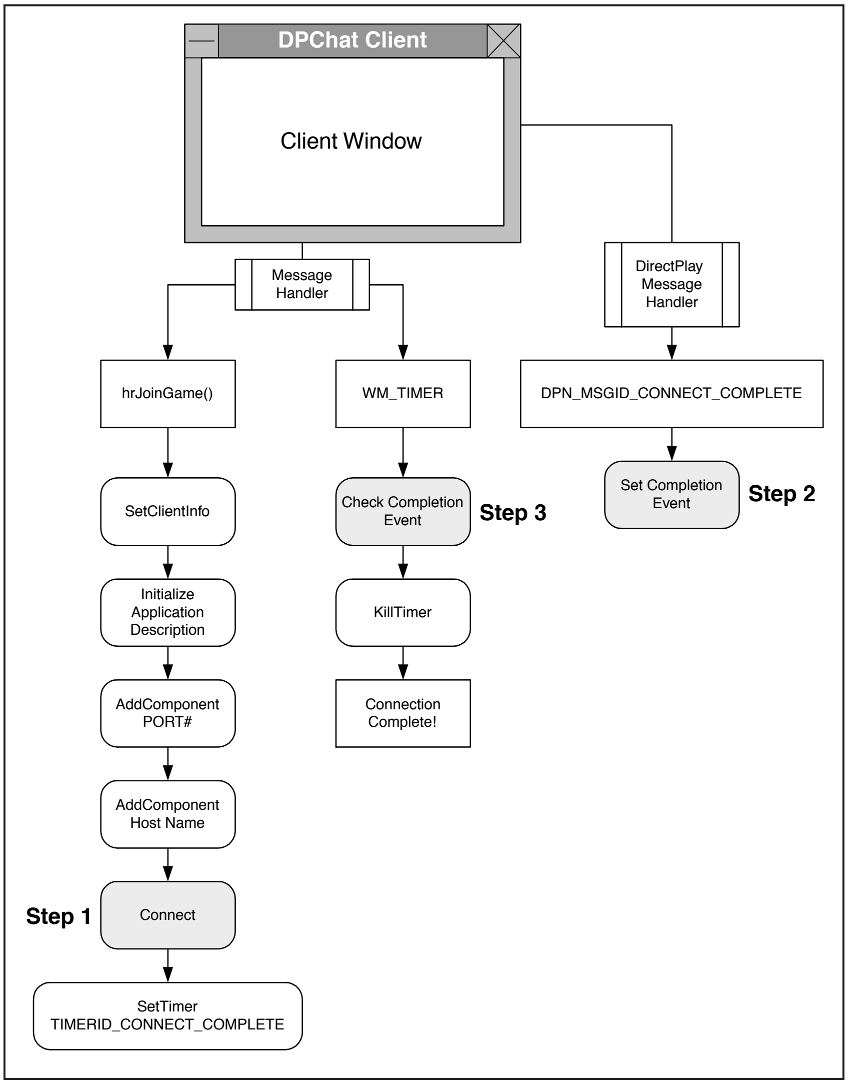


Figure 10.13

The client program flow once a connection is completed

The first thing I do here is retrieve the text from the client's edit box. I then combine it with the player's name to create a composite chat message to send.

The next item of interest is setting the chat packet's parameters. I do this by setting up a `PacketChat` class object. If you recall from the server section of this chapter, the `PacketChat` class has three member variables: `dwSize`, `dwType`, and `szText`. I set the size to equal the `sizeof()` the packet. I then set the type to equal `PACKET_TYPE_CHAT`. Last, I copy the message to send into the `szText` variable. Once all of this is out of the way, I point to the packet with a void pointer. This lets me send the raw form of the packet to the `hrSendClientMessage()` function.

The hrSendClientMessage() Function

Speaking of the `hrSendClientMessage()` function, here it is.

```
HRESULT     hrSendClientMessage( DWORD dwMessageType, PVOID pMsgBuffer )
{
    DPNHANDLE           hAsync;
    DWORD               dwLength;
    DPN_BUFFER_DESC     bufferDesc;
    PacketGeneric       *PGen;
    // Cast to a generic packet to get the size
    PGen = (PacketGeneric*)pMsgBuffer;
    dwLength = PGen->dwSize;
    if( dwLength == 0 )
        return(0);
    bufferDesc.dwBufferSize = dwLength;
    bufferDesc.pBufferData  = (BYTE*)pMsgBuffer;
    g_pDP->Send( &bufferDesc, 1, 0, NULL, &hAsync, 0 );
    return S_OK;
}
```

I'm not going to cover this entire function because it is virtually identical to the `hrSendServerMessage()` function I covered earlier in this chapter. The only real difference is that this call uses the `Send()` function instead of the `SendTo()` function. The reason the client interface doesn't have a `SendTo()` function is that clients can send only to the server, so there is no need to specify to whom it sends.

The IDirectPlay8Client::Send() Function

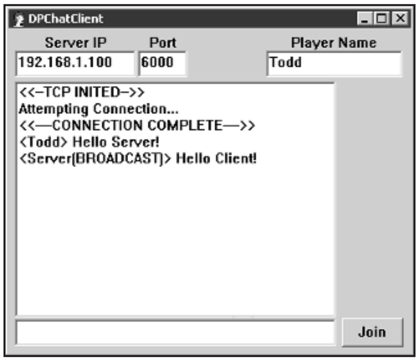
The `IDirectPlay8Client::Send()` function's sole purpose is to send messages to the server. Here is the prototype.

```
HRESULT Send(
    const DPN_BUFFER_DESC *const pBufferDesc,
    const DWORD cBufferDesc,
    const DWORD dwTimeOut,
    void *const pvAsyncContext,
    DPNHANDLE *const phAsyncHandle,
    const DWORD dwFlags
);
```

The first parameter expects the DPN_BUFFER_DESC data structure that contains the message to send. The second parameter expects the number of buffers passed in the first parameter. Currently, you can only send one buffer at a time, so always leave this at 1. The third parameter lets you specify a timeout in milliseconds. If you set this to 0, it keeps trying to send the packet forever.

The fourth parameter lets you specify an optional context. I just use NULL here most of the time. The fifth parameter receives a handle to the asynchronous call. If you use the DPNSEND_SYNC flag, you must set this to NULL. The last parameter lets you specify various flags for the operation. The same flags that you saw for the server's SendTo() function apply here.

Go ahead and run the server and client applications, establish a connection, and then send a few chat packets across the wire. The client application now should resemble something like Figure 10.14.



Server IP: 192.168.1.100
Port: 6000
Player Name: Todd

```
<<–TCP INITED–>>
Attempting Connection...
<<—CONNECTION COMPLETE—>>
<Todd> Hello Server!
<Server(BROADCAST)> Hello Client!
```

Figure 10.14

Output from the client application

Review

Okay, so now you have finished yet another wonderful chapter. Well, maybe not wonderful; how about amazing? Before you move on, I want to leave you with a list of the steps necessary to be a server or client.

Here is a brief rundown of the steps necessary to be a server.

1. Initialize COM.
2. Create an `IDirectPlay8Server` interface.
3. Create a device address.
4. Set TCP/IP as the service provider.
5. Set the server information using `IDirectPlay8Server::SetServerInfo()`.
6. Initialize the application description.
7. Add the port number to the device address.
8. Call the `IDirectPlay8Server::Host()` function.

Here is a brief rundown of the steps necessary to be a client.

1. Initialize COM.
2. Create an `IDirectPlay8Client` interface.
3. Create a device address.
4. Set TCP/IP as the service provider.
5. Create a host address.
6. Set TCP/IP as the service provider.
7. Create a completion event.
8. Set the client information using the `IDirectPlay8Client::SetClientInfo()` function.
9. Initialize the application description.
10. Add the host's port number and IP address to the host's address interface.
11. Call the `IDirectPlay8Client::Connect()` function.
12. Set the connection timer to go off every 1/10th of a second.

13. Accept the DPN_MSGID_CONNECT_COMPLETE message and signal the completion event.
14. Check for the completion event and signal that connection is complete.
15. Kill the connection timer.

I will see you at the next tribal council, er, chapter.

CHAPTER 11

DirectX Graphics

Up to this point, I have yet to show you any programs that look nice. I mean, Windows dialog-based examples are great for teaching the basics of network code, but they sure don't look very nice. In this chapter, I'll show you everything you need to know to get started with DirectX graphics. Originally, I had planned to show you how to use DirectDraw, but guess what: Microsoft decided that in DirectX 8, it would remove DirectDraw and use Direct3D 100 percent. It has combined both DirectDraw and Direct3D into one system called DirectGraphics. It is actually easier now to set up and get started, but it does mean learning some new tricks if you don't already know how to do 3D. Don't worry, though; I make it as painless as possible and just concentrate on how to get started with it. I'll cover the following topics in this chapter.

- Device independence
- Software emulation
- Texture mapping
- Mip-mapping
- Gouraud and flat shading
- Multiple hardware lights
- Hardware acceleration
- Page flipping
- Z-buffers
- Fog
- Writing your first 3D program
- Writing your second 3D program

What Is Direct3D?

Even though the system is called DirectGraphics, you will still find that the 3D portion is referred to as Direct3D. This component of DirectGraphics communicates with 3D hardware on a device-independent level. What this means is that with Direct3D, you can write your code to work on any video card without having to worry about the manufacturer. For those of you accustomed to DOS, this is a welcome change. No longer do you have to write drivers just to access hardware.

Direct3D provides several interfaces and operations to allow you to create very impressive visual experiences.

Device Independence

I already mentioned device independence, but this really is a powerful portion of DirectGraphics. Everything isn't completely rosy, though, because not every 3D card is created equal. One of the headaches you might experience when developing 3D applications is trying to write your application to work with all possible video cards. This is where developing for a closed system, such as Microsoft's Xbox, is nice. If you develop for the Xbox, then you have a fixed set of hardware features to worry about. You don't have to keep in mind the other hundreds of cards. To get a visual of how device independence works, take a look at Figure 11.1.

In Figure 11.1, you can see how the same graphics call maps to multiple hardware types. You have to keep in mind the following features when developing your game.

- Maximum texture size
- Color depth
- Texture memory

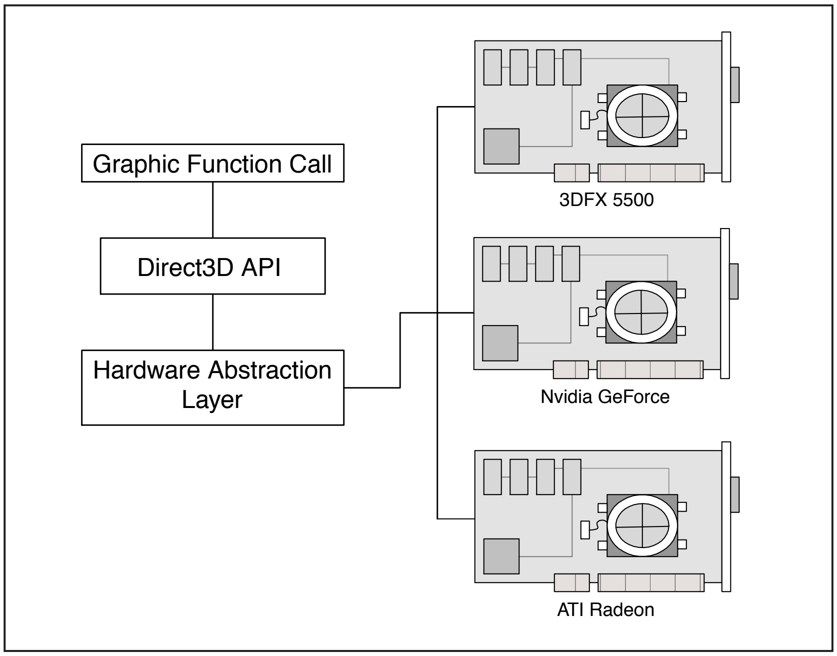


Figure 11.1
Device-independent architecture

Maximum Texture Size

Some 3D cards, such as the Voodoo 3500, have a limit on the size of textures they handle. The Voodoo 3500 can only handle textures that are 256 pixels wide by 256 pixels high. This means that if you want to display a texture that is larger than 256 × 256, you have to piece together multiple textures. This can be a real pain in the rear, as you will find out later when I show you how to do it. Unfortunately, there is not much you can do about such limitations, unless you want to disenfranchise many possible users.

Color Depth

Color depth has always been a concern when creating games because you have to plan ahead for all possible resolutions and colors available to a user. Some graphics cards let you have 16-bit color, and other cards let you have 32-bit color. You should always plan to support 16-, 24-, and 32-bit color in your applications to be compatible with the largest possible audience. Also, be sure to keep in mind that the higher the color depth, the slower the application. If you design your application's speed for 32-bit support, then you won't have a problem switching down to 16-bit.

Texture Memory

Not all 3D cards have the same amount of texture memory. Some cards have 8MB of memory, but others have 64MB of memory. In some cases, the cards might have much less memory than this. Whenever you load up a graphic image to map onto a 3D object, that image takes up texture memory. If you run out of texture memory, then the system has to copy it from slow system memory. You should always plan your texture usage very carefully to avoid wasting it. A good rule of thumb is to expect no more than 16MB of texture memory. Even this amount is more than many cards have, but you can't hold yourself back too much!

Software Emulation

Here it is, the holy grail of 3D graphics. Okay, not really, but it sounds cool. Anyway, software emulation lets your application do things in software whenever hardware is not present to accelerate it for you. This allows you to write a 3D game that takes advantage of 3D hardware but doesn't necessarily require it. Imagine adding fog to your game. If the hardware does not support fog, then the software emulator takes over and renders it for you. Take a look at Figure 11.2 to see this principle illustrated.

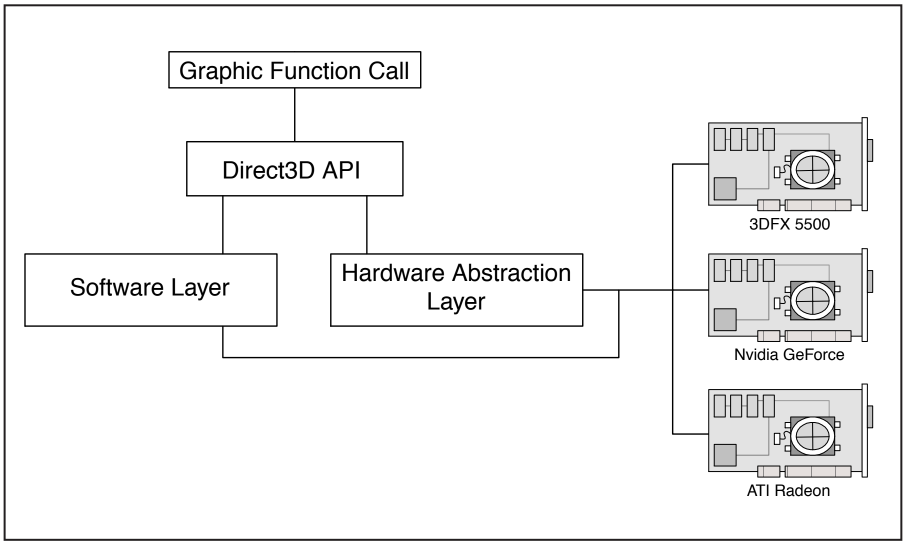


Figure 11.2

Software emulation architecture

In Figure 11.2, you can see how the system routes commands to either hardware or software depending on the features of the graphics card.

CAUTION

Software rendering slows down your game quite a bit. Be sure to only use software emulation where speed is not critical.

Texture Mapping

Whenever you draw a 3D object that has a graphic image on it, it is *texture-mapped*. Take a peek at Figure 11.3 to see the difference between a texture-mapped cube and a non-texture-mapped cube.

The cube on the left is a standard 3D primitive with no texture mapping. The cube on the right has an image mapped onto all six sides—this is what texture mapping is all about. 3D graphics cards are very good about accelerating this CPU-intensive process. Without 3D acceleration, texture mapping is much slower. Direct3D lets you do all sorts of neat texture-mapping effects such as scaling textures, moving textures, and even altering textures in real time.

Mip-Mapping

Mip-mapping is kind of an advanced topic to be bringing up so early in the chapter, but here goes anyway. Normally in 3D rendering, an object on the screen uses

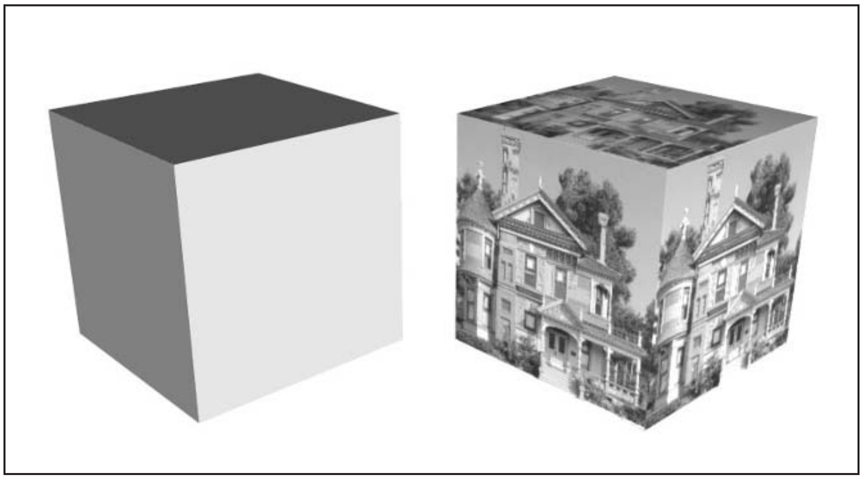


Figure 11.3

Texture mapping

one texture. This isn't much of a problem, but think about objects that are very far away from the camera. You wouldn't want to use a full-blown detailed texture on these objects because you can barely even see them. Take a look at Figure 11.4 to see an example of this concept.

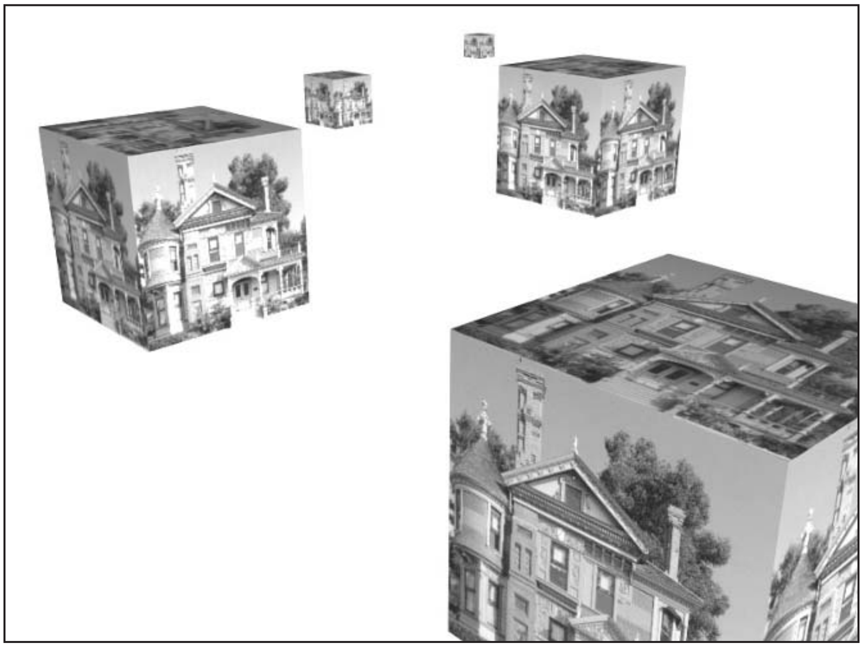


Figure 11.4

The same texture rendered at multiple distances

In Figure 11.4, you can see that the cube close to the camera deserves to use the detailed texture. But the cube way off in the distance can barely be seen. The problem here is that it takes just as much CPU power to draw the texture on the cube in the distance as it does to draw the texture on the cube in the foreground. This is a big waste of processing power.

Enter the white knight, mip-mapping. *Mip-mapping* lets you specify multiple textures to be used at various distances. When you turn on this feature, Direct3D automatically uses the appropriately-sized texture at the appropriate distance. It does consume more memory because you are keeping many copies of the same texture, but it also results in faster rendering times.

Take a look at Figure 11.5 to see an example of mip-mapping. In it, you can see five versions of the original texture. The cube that is closest to the camera gets the most detailed texture. As the other cubes get farther away from the camera, they use smaller versions of the texture.

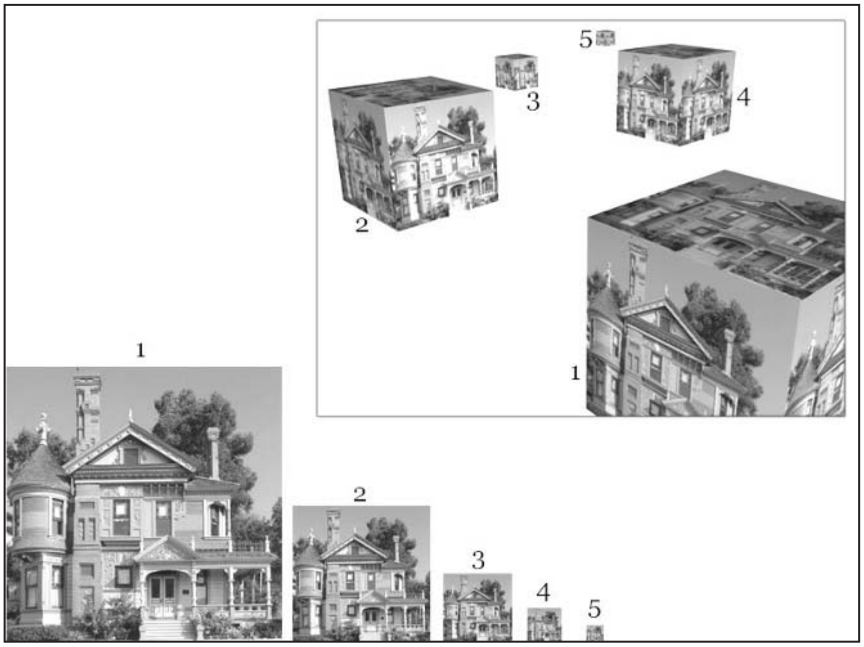


Figure 11.5

Mip-mapping in action

Gouraud and Flat Shading

Gouraud—that's a strange sounding name isn't it? When you render a 3D object, it can have one of two shading methods, gouraud or flat. The shading method determines how the surface of the object appears to the viewer.

Gouraud shading gives objects a smooth appearance. Most present-day games use smooth shading to represent everything. In Direct3D, it is easy to select this method of shading because you can accomplish it with only one function call. You can see a gouraud-shaded sphere in Figure 11.6.

NOTE

In case you were wondering, gouraud is pronounced "goorow" or "garrow."

Back in the old days, everything was flat shaded because flat shading takes up much less processing power than gouraud shading does. When something is flat shaded, you can see the facets that make up the object. The best way I can explain this is by having you take a look Figure 11.7.

As you can see in Figure 11.7, the sphere is flat shaded. You can see the facets that make up the object. Sometimes you will want to use flat shading, but most of the time you will use gouraud shading.

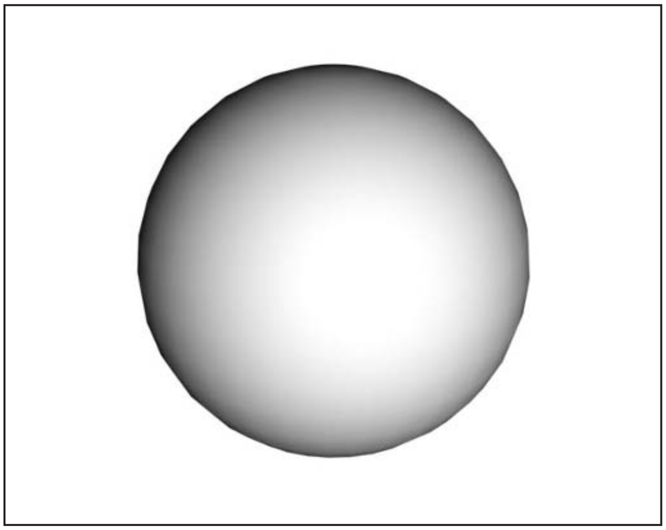


Figure 11.6

Gouraud shading

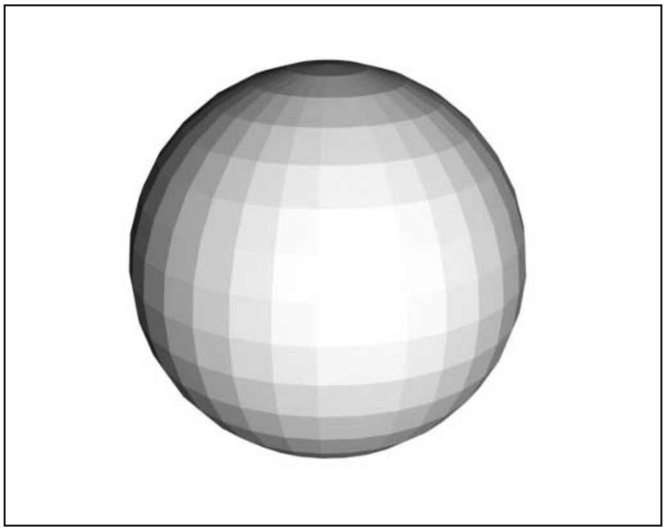


Figure 11.7

Flat shading

Multiple Hardware Lights

If you didn't have any lighting in your room, then you wouldn't be able to see very much. This rule also applies in the wonderful world of 3D programming. There are exceptions to every rule, though.

Simulated Lighting

It is possible to simulate lighting in 3D by giving every object 100 percent ambient light. What this does is make everything appear completely lit. The downside to this is that when something is completely bright, then the geometry of the object gets washed out. To fix this problem, game programmers light the textures that are mapped onto the object to simulate lighting.

In Figure 11.8, you can see the simulated lighting effect. The wall really doesn't have a light shining on it from above; the texture just has the shading built into it. This is also referred to as *light mapping*.

Hardware Lighting

The flip side of the coin is using hardware lights. Hardware lights have an actual position in the 3D scene and light up objects accordingly. The downside to hard-ware lights is that they take up a lot of processing power. The upsides are that you

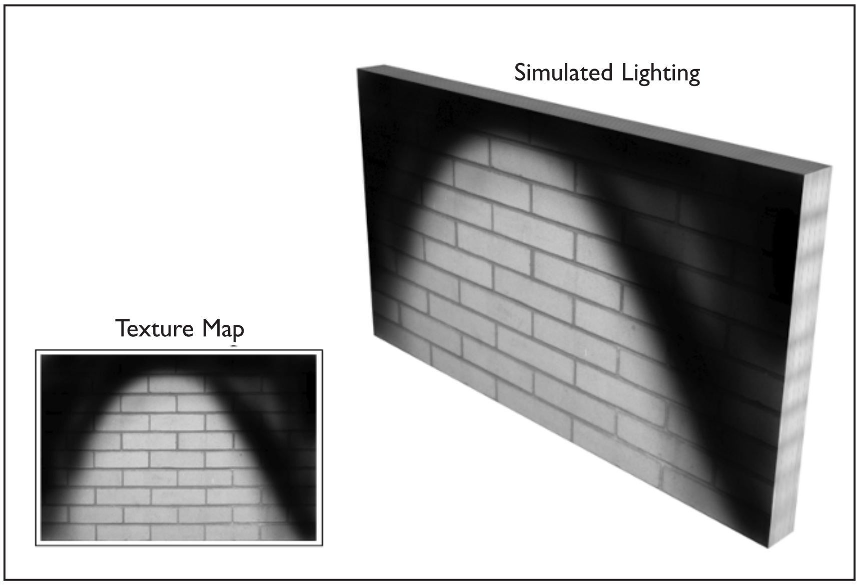


Figure 11.8

Simulated lighting

don't have to create pre-lit textures and things usually turn out looking more realistic.

Direct3D lets you specify a finite number of hardware lights in your scene. These lights have many settings, and the two most common are intensity and color. By specifying the color, you can create all sorts of neat effects for a cool atmosphere in your game.

Hardware Acceleration

Hardware acceleration is the opposite of the software rendering, which I mentioned earlier. By supporting hardware acceleration, Direct3D lets you tap into the power of 3D video cards. These cards have special hardware that takes the load off the processor during key operations. The really cool cards even have transform and lighting chips that take the load of processing geometry off the main CPU.

When processing gets turned over to the hardware, it goes through the HAL (*Hardware Abstraction Layer*). This layer keeps you from having to know what hardware to address; you just send commands to the HAL, and it knows how to operate the hardware. The architecture appears in Figure 11.9.

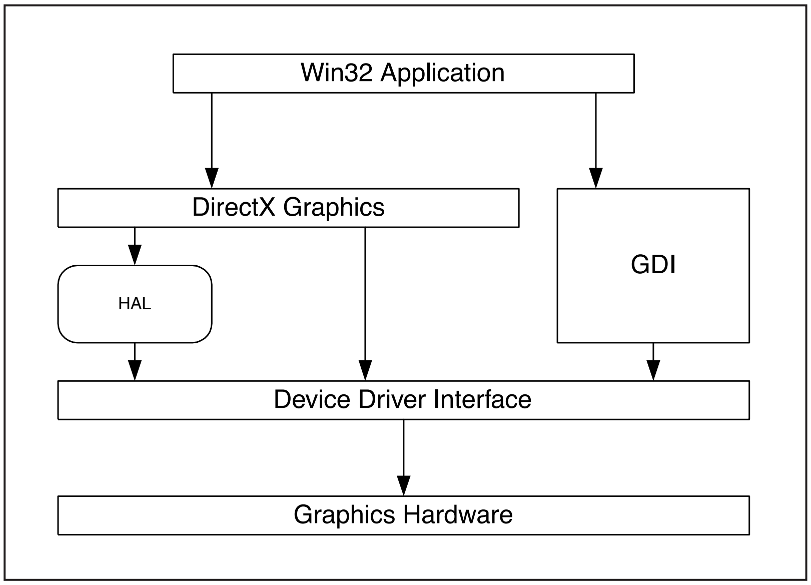


Figure 11.9

The hardware abstraction layer

Page Flipping

Whenever you finish rendering a 3D scene, you have to present it to the user in one way or another. A common way to present the finished scene to the user is to utilize *page flipping*. With this method, you draw everything in your scene to a background page that is not visible to the user. When you are done drawing, you flip the background page with the foreground page so that the user can see the new rendering. By doing this, you are able to skip the computational-intensive process of copying data from one buffer to another. Take a look at Figure 11.10 to see the concept of page flipping illustrated.

Z-Buffers

If you put objects into a 3D scene in no particular order, then you can end up with objects that do not appear correctly. Imagine rendering a cube very close to the camera and then rendering a sphere that is far away. If you didn't perform any depth processing, then the sphere could overlap the cube. Take a look at Figure 11.11 to see an example of this idea.

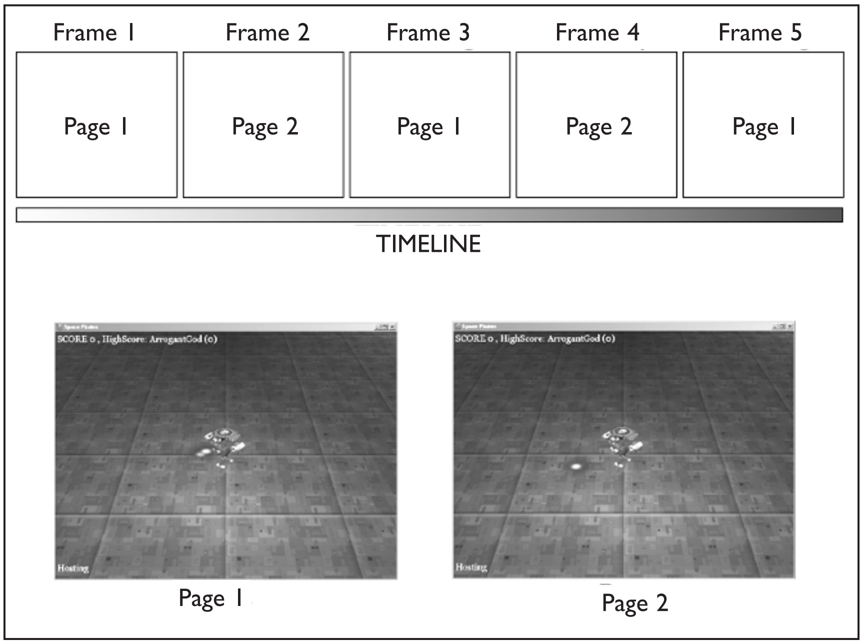


Figure 11.10

Page flipping

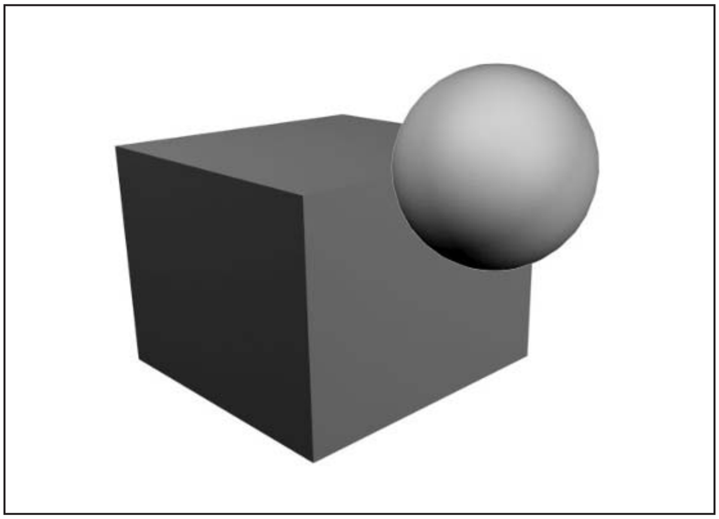


Figure 11.11

Lack of depth buffering

In Figure 11.11, the sphere is actually behind the cube in 3D space, but because it was rendered last, it appears on top of the cube. One way to fix this is to incorporate a *z-buffer*. The buffer gets its name because the z-coordinate determines the depth of an object. When you render objects with z-buffering turned on, they appear in the proper order no matter what order you render them. Each object rendered stores information into the z-buffer about its position. If you render another object on top of an existing one, it actually draws pixels where it should. Look at Figure 11.12 to see the sphere and cube example with z-buffering turned on.

DirectX allows you to use hardware-accelerated z-buffers on cards that support it. This greatly enhances the speed at which you can calculate depth buffering. With DX8, it is as simple as turning on an option.

Fog

In the not-so-old days, programmers used fog to hide the fact that geometry in the distance couldn't be drawn. They did this because the video cards and processors of the time could not handle drawing an entire scene; only the objects that were close to the camera could afford to be presented. Games such as *Turok: Dinosaur Hunter* were notorious for the fog effect.

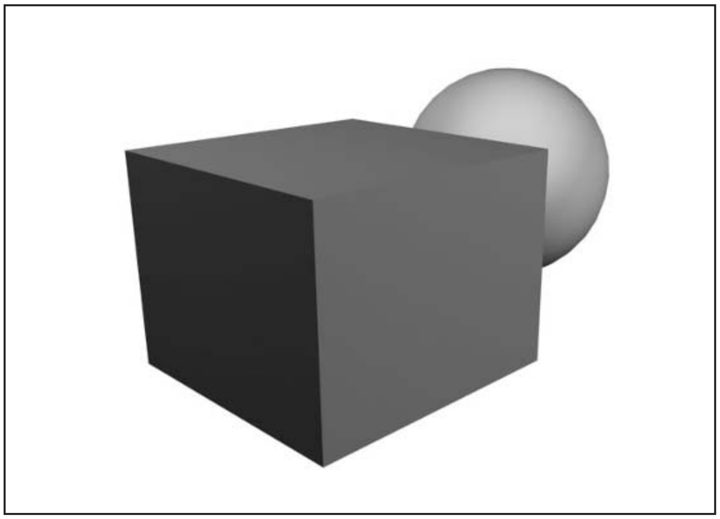


Figure 11.12

Proper z-buffering

With DirectX, you can implement fog into your games to have objects at a given distance disappear into the fog. Even though you shouldn't have to code in fog to hide geometry problems, you can still use it for all sorts of special effects. Underwater games, for example, could benefit from fog. In that particular case, you might want everything to disappear into a deep blue background. DirectX gives you the ability to use fog in your games, and it is very simple to implement.

How to Use DirectGraphics

Well, I have told you about some of the great features of Direct3D, but I really haven't showed you how to do anything yet. Now I want to show you how to get started with this great API. The first thing I'll cover is the `IDirect3D8` interface. The `IDirect3D8` interface connects you with the Direct3D system.

The IDirect3D8 Interface

You use the `IDirect3D8` interface to set up the device that controls your 3D environment. In other words, this interface sets up the screen resolution and indicates the video card to use. The first order of business is how to actually create the `IDirect3D8` interface—with the `Direct3DCreate8()` function.

The Direct3DCreate8() Function

This function's sole purpose is to create an `IDirect3D8` interface. Here is the prototype.

```
IDirect3D8* Direct3DCreate8(
    UINT SDKVersion
);
```

The first and only parameter passes in a define named `D3D_SDK_VERSION`. The reason this formality exists is to make sure that you are using the correct header files. If the header files you are using do not match the version, then an error occurs.

If there are no errors, then the function returns a handle to an `IDirect3D8` interface. If something goes wrong, a `NULL` is returned. I bet you didn't think it was going to be that simple, did you?

Before I get too much further into this, I want to show you all of the functions available to the `IDirect3D8` interface, so take a peek at Table 11.1.

Table 11.1 IDirect3D8 Interface Functions

Function	Purpose
`CheckDepthStencilMatch`	Checks whether a stencil buffer is compatible with a render target
`CheckDeviceFormat`	Checks whether a surface can be used for a variety of operations with the current display adapter
`CheckDeviceMultiSampleType`	Checks whether a particular multisampling method is available on the display adapter
`CheckDeviceType`	Checks whether a device can handle the setting desired completely in hardware
`CreateDevice`	Starts the video mode
`EnumAdapterModes`	Enumerates available display modes for an adapter
`GetAdapterCount`	Tells you how many video adapters are plugged into the computer
`GetAdapterDisplayMode`	Retrieves information about the current display mode of an adapter
`GetAdapterIdentifier`	Fills a `D3DADAPTER_IDENTIFIER8` data structure with information about the adapters in the system when Direct3D was initialized
`GetAdapterModeCount`	Retrieves the number of display modes available to the video adapter
`GetAdapterMonitor`	Gets the handle of the monitor attached to the display adapter
`GetDeviceCaps`	Obtains information about a display adapter's capabilities
`RegisterSoftwareDevice`	Sets a certain capability to be rendered in software for when hardware is not available

The IDirect3D8::GetAdapterDisplayMode() Function

The first function you want to call after getting an IDirect3D8 interface is the GetAdapterDisplayMode() function. This function retrieves information about the current video mode. Here is the prototype.

```
HRESULT GetAdapterDisplayMode(
    UINT Adapter,
    D3DDISPLAYMODE* pMode
);
```

The first parameter is rather easy because it expects the number of the adapter you want to query. I use the D3DADAPTER_DEFAULT definition here because it always returns the primary display adapter. The second parameter passes back a D3DDISPLAYMODE data structure describing the video mode. The function returns a D3D_OK result if everything works. If the function fails, then it returns a D3DERR_INVALIDCALL return value.

Okay, now that you have seen the prototype, here is the code I use to make the call.

```
if( FAILED( pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &d3ddm ) ) )
    return E_FAIL;
```

The d3ddm variable in this call is type D3DDISPLAYMODE. You might be asking what is in the D3DDISPLAYMODE variable. Well, read on!

The D3DDISPLAYMODE Data Structure

The D3DDISPLAYMODE data structure does nothing more and nothing less than describe the display mode of an adapter. Here is what the data structure looks like.

```
typedef struct _D3DDISPLAYMODE {
    UINT                Width;
    UINT                Height;
    UINT                RefreshRate;
    D3DFORMAT   Format;
} D3DDISPLAYMODE;
```

Most of the data structure should speak for itself; the only difficult one is the D3DFORMAT variable. The width tells you the width of the current display mode. The height tells you the height of the current display mode. Both the width and height are measured in pixels.

The refresh rate tells you the refresh rate of the video card. A 0 here indicates that it is set at the adapter's default setting. As I mentioned a second ago, the Format data member is type D3DFORMAT.

The D3DFORMAT Data Structure

The D3DFORMAT data structure defines the different types of surfaces available in Direct3D. The two that you see in the demo code are the D3DFMT_D16 and D3DFMT_D32 formats. These two are used when the z-buffer is either 16 bits or 32 bits in depth. For a complete list of formats, I recommend that you look at the DirectX 8 help file.

Now that you have retrieved the adapter display information, you can go on to set up the display mode you want to use. You do this by first setting up a D3DPRESENT_PARAMETERS data structure.

The D3DPRESENT_PARAMETERS Data Structure

Before I show you the prototype, I want you to see how I set up a D3DPRESENT_PARAMETERS data structure. Here is the code I use.

```
D3DPRESENT_PARAMETERS                d3dpp;
// Clear out device parameters
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.hDeviceWindow                    = hWnd;
d3dpp.BackBufferWidth                  = 640;
d3dpp.BackBufferHeight                 = 480;
d3dpp.BackBufferFormat                 = d3ddm.Format;
d3dpp.BackBufferCount                  = 1;
d3dpp.SwapEffect                       = D3DSWAPEFFECT_FLIP;
d3dpp.MultiSampleType                  = D3DMULTISAMPLE_NONE;
d3dpp.EnableAutoDepthStencil           = TRUE;
d3dpp.AutoDepthStencilFormat           = D3DFMT_D16;
d3dpp.Windowed                         = TRUE;
```

In this code, I set up a 640 × 480 screen in the display adapter's current color depth. The code will run in windowed mode and will use page flipping. Here is the prototype for the data structure.

```
typedef struct _D3DPRESENT_PARAMETERS_ {
    UINT                        BackBufferWidth;
    UINT                        BackBufferHeight;
    D3DFORMAT                   BackBufferFormat;
    UINT                        BackBufferCount;
    D3DMULTISAMPLE_TYPE         MultiSampleType;
    D3DSWAPEFFECT               SwapEffect;
    HWND                        hDeviceWindow;
    BOOL                        Windowed;
    BOOL                        EnableAutoDepthStencil;
    D3DFORMAT                   AutoDepthStencilFormat;
    DWORD                       Flags;
    UINT                        FullScreen_RefreshRateInHz;
    UINT                        FullScreen_PresentationInterval;
} D3DPRESENT_PARAMETERS;
```

Hey, you know what? I think you are due for another illustration. Look at Figure 11.13 to see the list of steps necessary to set up the 3D environment.

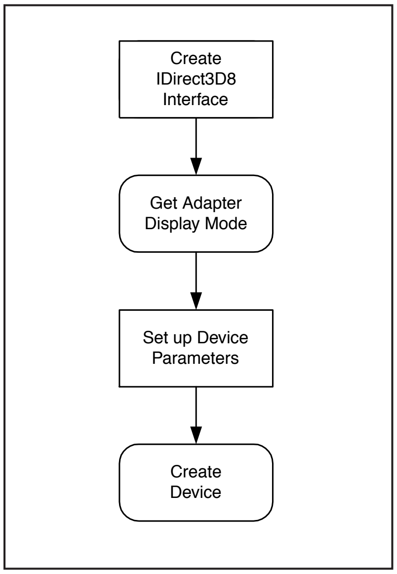


Figure 11.13

Initializing Direct3D

If you look at the steps, you'll see that you are only one step away from initializing a display! Before I move on, though, let me finish talking about the D3DPRESENT_PARAMETERS.

The first four parameters deal with how the back buffer is set up. When you use buffer swapping for displaying graphics, you first copy everything to a back buffer. Once you are ready to display the graphics, you copy the back buffer to the front buffer and then display the front buffer all at once. This method lets you maintain a steady display and prevent tearing.

The BufferHeight and BufferWidth parameters specify the size of the back buffer. If you set these to 0 in windowed mode, then the buffer is the same size as the device window. The BufferFormat describes the buffer's format in a D3DFORMAT structure. I usually pass the Format member from the D3DDISPLAYMODE data structure here.

The BufferCount tells the graphics system how many buffers are in the swapping chain. Normally, you set this to 1, but it is possible to do double and triple buffering if you want. The MultiSampleType member variable tells how multisampling is handled. Just remember to set this to D3DMULTISAMPLE_NONE for now. The SwapEffect member tells the system how to perform swapping, be it page flipping or buffer swapping. The hDeviceWindow just expects the handle to the window your application is using.

The Windowed member should be set to 1 for windowed mode or 0 for full-screen mode. The EnableAutoDepthStencil member sets whether Direct3D will manage depth buffers. If you set this to TRUE, then you need to set up the next member as well. The AutoDepthStencilFormat, which must be set if EnableAutoDepthStencil is TRUE, specifies the format of depth buffers to use.

Normally, you set the Flags variable to 0. If you want to be able to modify the back buffer in the future, you need to set the flags to D3DPRESENTFLAG_LOCKABLE_BACKBUFFER.

You use the FullScreen_RefreshRateInHz member only in full-screen modes. If you want the display to update at its current level, then you pass a D3DPRESENT_RATE_DEFAULT here. If you want the application to update the display as fast as it can, then you use the D3DPRESENT_RATE_UNLIMITED setting.

The FullScreen_PresentationInterval sets how often the back buffer is presented. For windowed mode, you must set this to D3DPRESENT_INTERVAL_DEFAULT. If you are in full-screen mode, you can set it to one of values listed in Table 11.2.

Table 11.2 FullScreen_PresentationInterval Values

Value	Description
D3DPRESENT_INTERVAL_IMMEDIATE	Presents the buffer immediately. No vertical retrace is waited upon.
D3DPRESENT_INTERVAL_ONE	Waits for exactly one vertical retrace to pass before displaying the buffer.
D3DPRESENT_INTERVAL_TWO	Waits for exactly two vertical retraces to pass before displaying the buffer.
D3DPRESENT_INTERVAL_THREE	Waits for exactly three vertical retraces to pass before displaying the buffer.
D3DPRESENT_INTERVAL_FOUR	Waits for exactly four vertical retraces to pass before displaying the buffer.

The IDirect3D8::CreateDevice() Function

Don't worry; you're almost to the point of initializing a 3D environment. The last function in your way is the CreateDevice() function. This function creates a device that represents the video display you desire. Here is the prototype for the function.

```
HRESULT CreateDevice(
    UINT                            Adapter,
    DDEVTYPE                        DeviceType,
    HWND                            hFocusWindow,
    DWORD                           BehaviorFlags,
    D3DPRESENT_PARAMETERS*          pPresentationParameters,
    IDirect3DDevice8**              ppReturnedDeviceInterface
);
```

The first parameter specifies which adapter in the system you want to set up. Normally, you can just pass the D3DADAPTER_DEFAULT value here to specify the primary display adapter. The second parameter sets the type of device you want to create. For hardware acceleration, you should use the D3DDEVTYPE_HAL type. For software rendering, you can set this to D3DDEVTYPE_SW.

The third parameter lets you specify which window will receive the device. Make sure that you pass in the top-level window here if you are using full-screen mode. The fourth parameter lets you use flags that determine the behavior of the device. Table 11.3 lists the values available for this parameter.

The fifth parameter expects the presentation parameters to be used. In this parameter, you pass the `D3DPRESENT_PARAMETERS` data structure that you created earlier. The last parameter receives the `IDirect3DDevice8` device returned. Here is the code used to call the `CreateDevice()` function.

```
if( FAILED( pD3D->CreateDevice(      D3DADAPTER_DEFAULT,
                     D3DDEVTYPE_HAL,
                     hWnd,
                     D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                     &d3dpp,
                     &pd3dDevice ) ) )  {
    return E_FAIL;
}
```

Table 11.3 hFocusWindow Values

Value	Description
`D3DCREATE_HARDWARE_VERTEXPROCESSING`	Uses hardware transform and lighting such as that available in Nvidia graphics cards.
`D3DCREATE_SOFTWARE_VERTEXPROCESSING`	Uses software transform and lighting.
`D3DCREATE_MIXED_VERTEXPROCESSING`	Uses a mixture of hardware and software.
`D3DCREATE_FPU_PRESERVE`	Specifies that the application requires double-precision floating-point-unit (FPU) exceptions. This is usually never used.
`D3DCREATE_PUREDEVICE`	Disallows the use of get function calls.
`D3DCREATE_MULTITHREADED`	Tells Direct3D to run in thread-safe mode, which can degrade performance.

In this code, I am getting a device for the primary display. The device will use hardware acceleration, but will use software for vertex processing. If I want to initialize a video card that has hardware T&L (*transform and lighting*), then I use the `D3DCREATE_HARDWARE_VERTEXPROCESSING` flag.

Once you create a device, you can go on to do all sorts of cool 3D rendering.

Your First Direct3D Application

All theory and no play makes Jack a dull boy, or so the saying goes. Now that you have read through a lot of theory, I think it is time to write a 3D application. This application won't do anything besides fill a window with color, but it does actually initialize the 3D environment. Figure 11.14 shows the output from the application I am about to show you.

I haven't done this in a while, so here is the complete listing of the Init3D application.

```
#define INITGUID
#define WIN32_LEAN_AND_MEAN     // No MFC
#include <d3d8.h>
#include <stdio.h>
```

Figure 11.14

Output from the Init3D application

```
// Direct3D globals
LPDIRECT3D8                                         pD3D;
LPDIRECT3DDEVICE8                                   pd3dDevice;
// Class name & app name
const char     *szWndClass                          = "Init3D";
const char     *szProgramName                       = "Init3D Example";
// Windows globals
HINSTANCE                                           g_hInstance;
// Function prototypes
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow);
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam,
 LPARAM lParam );
HRESULT hrInit3D( HWND hWnd );
void vCleanup(void);
void vRenderScene(void);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASS          wc;
    HWND              hWnd;
    MSG               msg;

    // Set up and register window class
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = MsgProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( NULL, IDI_APPLICATION );
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH )GetStockObject(BLACK_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = szWndClass;
    RegisterClass(&wc);
    // Create a window
    hWnd = CreateWindowEx(WS_EX_TOPMOST, szWndClass, szProgramName,
                        WS_OVERLAPPEDWINDOW, 0, 0, 640, 480,
                        NULL, NULL, hInstance, NULL);
    // Set up global window handles
    g_hInstance = hInstance;
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    if( hrInit3D(hWnd) == E_FAIL ) {
        exit(1);
    }
    // Display the 3D environment
    vRenderScene();
    // Process messages until the program is terminated
    while( GetMessage ( &msg, NULL, 0, 0 ) ) {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    // Clean up the environment
    vCleanup();
    return(msg.wParam);
};
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam,
 LPARAM lParam )
{
    switch( msg ) {
        case WM_DESTROY:
            PostQuitMessage( 0 );
        return 0;
    }
    return DefWindowProc( hWnd, msg, wParam, lParam );
}
HRESULT hrInit3D( HWND hWnd )
{
    D3DPRESENT_PARAMETERS          d3dpp;
    D3DDISPLAYMODE                 d3ddm;
    RECT                           rcWindowBounds;
    RECT                           rcWindowClient;
    // Create the D3D object
    if( NULL == ( pD3D = Direct3DCreate8( D3D_SDK_VERSION ) ) )
        return E_FAIL;
    // Get adapter settings
    if( FAILED( pD3D->GetAdapterDisplayMode(
     D3DADAPTER_DEFAULT, &d3ddm ) ) )
        return E_FAIL;
    // Get the size of the window
    GetWindowRect( hWnd, &rcWindowBounds );
    GetClientRect( hWnd, &rcWindowClient );
    // Clear out device parameters
    ZeroMemory( &d3dpp, sizeof(d3dpp) );
    // Set up device parameters
    d3dpp.hDeviceWindow                  = hWnd;
    d3dpp.BackBufferWidth                = (rcWindowClient.right
                                          - rcWindowClient.left);
    d3dpp.BackBufferHeight               = (rcWindowClient.bottom
                                          - rcWindowClient.top);
    d3dpp.BackBufferFormat               = d3ddm.Format;
    d3dpp.BackBufferCount                = 1;
    d3dpp.SwapEffect                     = D3DSWAPEFFECT_FLIP;
    d3dpp.MultiSampleType                = D3DMULTISAMPLE_NONE;
    d3dpp.EnableAutoDepthStencil         = TRUE;
    d3dpp.AutoDepthStencilFormat         = D3DFMT_D16;
    d3dpp.Windowed                       = TRUE;
    // Create the D3DDevice
    if( FAILED( pD3D->CreateDevice( D3DADAPTER_DEFAULT,
                D3DDEVTYPE_HAL,
                hWnd,
                D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                &d3dpp,
                &pd3dDevice ) ) )
    {
        // Try 32-bits
        d3dpp.AutoDepthStencilFormat     = D3DFMT_D32;
        if( FAILED( pD3D->CreateDevice( D3DADAPTER_DEFAULT,
                            D3DDEVTYPE_HAL,
                            hWnd,
                            D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                            &d3dpp,
                            &pd3dDevice ) ) )
        {
            return E_FAIL;
        }
    }
    return S_OK;
}
void vCleanup(void)
{
    if( pd3dDevice != NULL )
        pd3dDevice->Release();
    if( pD3D != NULL )
        pD3D->Release();
    UnregisterClass( szWndClass, g_hInstance );
}
void vRenderScene(void)
{
    // Clear the scene
    pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET,
     D3DCOLOR_XRGB(150,150,255), 1.0f, 0 );
    pd3dDevice->BeginScene();
    pd3dDevice->EndScene();
    // Output the scene
    pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

If you don't feel like typing in all of the code, you can find it on this book's CD, in the Chapter11 Zip file.

Setting Up Visual C++ for 3D Applications

To set up Visual C++ for a Direct3D application, you first create a Win32 Application project using the empty project template. Once that is done, you need to add the d3d8.lib library to the project. I generally use the Release mode for my final projects because it doesn't have the debugging overhead. Look at Figure 11.15 to see the project layout for the Init3D application.

The WinMain() Function

There isn't very much special about the `WinMain()` function for this program. It resembles most of the code I have shown you so far in the book. Here are the steps that are accomplished in this function.

1. Create a window class with the `CreateWindowEx()` function.
2. Display the created window with the `ShowWindow()` function.
3. Update the window with the `UpdateWindow()` function.
4. Initialize the 3D system with the `hrInit3D()` function.

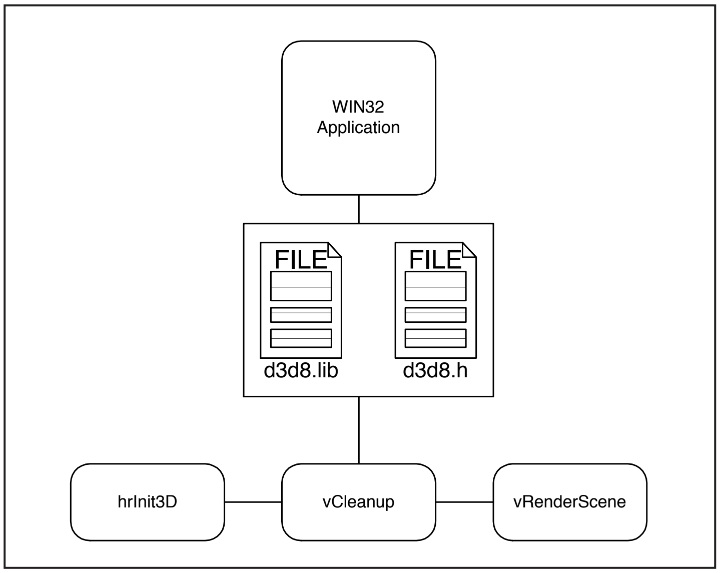


Figure 11.15

Project layout for the Init3D application

5. Render the 3D scene with the `vRenderScene()` function.
6. Process messages with the `GetMessage()` function.
7. Close down the application with the `vCleanup()` function.

The three functions that should look new to you are `hrInit3D()`, `vRenderScene()`, and `vCleanup()`.

The hrInit3D() Function

The `hrInit3D()` function sets up the 3D system for use. This function contains the code that I referred to back in the "How to Use DirectGraphics" section. This function must

1. Create an `IDirect3D8` interface with the `Direct3DCreate8()` function.
2. Retrieve the adapter settings with the `GetAdapterDisplayMode()` function.
3. Set up the display parameters with the `D3DPRESENT_PARAMETERS` structure.
4. Create an `IDirect3DDevice8` device with the `CreateDevice()` function.

The settings I show you in this function set up a 640 × 480, 16-bit window to display graphics, if the function is successful. If the `CreateDevice()` function fails, I then try to set up the window in 32-bit mode.

Creating the Interface

The first few lines of code create the IDirect3D8 interface using the Direct3DCreate8() function. I pass the D3D_SDK_VERSION definition to the function to make sure that all of the header files are proper.

```
if( NULL == ( pD3D = Direct3DCreate8( D3D_SDK_VERSION ) ) )
        return E_FAIL;
```

The pD3D variable is a global variable of type LPDIRECT3D8. The LPDIRECT3D8 type is just a pointer to an IDirect3D8 interface.

Retrieving the Adapter Settings

The next step to initialization is to poll the primary display adapter for its current settings. This allows you to set up your video system in a compatible way. To poll the adapter, you should use the GetAdapterDisplayMode() function.

```
if( FAILED( pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &d3ddm ) ) )
        return E_FAIL;
```

The D3DADAPTER_DEFAULT value I am using for the first parameter tells it to grab information about the primary display adapter. If you want to use a secondary adapter, you pass something else. The d3ddm variable is type D3DDISPLAYMODE and holds the retrieved information. I just create a local variable in this function because it is only needed during initialization.

Once you retrieve the adapter information, you can go ahead and set up the video display of your dreams. Well, maybe not the display of your dreams, but you can set it up however you like.

Setting the Device Parameters

To set up the video mode, you must populate a D3DPRESENT_PARAMETERS data structure. This data structure holds wonderful information such as the width and height of the display. Here are the settings I am using.

```
d3dpp.hDeviceWindow                    = hWnd;
d3dpp.BackBufferWidth                  = (rcWindowClient.right
                                   - rcWindowClient.left);
d3dpp.BackBufferHeight                 = (rcWindowClient.bottom
                                   - rcWindowClient.top);
d3dpp.BackBufferFormat                 = d3ddm.Format;
```

```
d3dpp.BackBufferCount                   = 1;
d3dpp.SwapEffect                        = D3DSWAPEFFECT_FLIP;
d3dpp.MultiSampleType                   = D3DMULTISAMPLE_NONE;
d3dpp.EnableAutoDepthStencil            = TRUE;
d3dpp.AutoDepthStencilFormat            = D3DFMT_D16;
d3dpp.Windowed                          = TRUE;
```

The main item of interest here is the `d3ddm.Format` setting. What I am doing here is using the format from the adapter to set up the back buffer. This makes sure that the video setting is compatible. Because the adapter is already in this mode, it is a safe bet.

The buffer width and height are set by retrieving the size of the client window and figuring out how much room is available for display. This is why I used the `GetWindowRect()` and `GetClientRect()` functions earlier.

CAUTION

Not all graphics cards can initialize in a window. If this is the case, then the initialization of the program might still fail. Adjust the width and height of the buffer along with the stencil format until your application works.

Creating the Device

The last step is to create the actual device used with the aforementioned `CreateDevice()` function.

```
if( FAILED( pD3D->CreateDevice( D3DADAPTER_DEFAULT,
                    D3DDEVTYPE_HAL,
                    hWnd,
                    D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                    &d3dpp,
                    &pd3dDevice ) ) )
```

In the first parameter, I tell the system to create a device for the primary display adapter. In the second parameter, I set up the program to be hardware accelerated. If you want to use software acceleration, you can use the `D3DDEVTYPE_SOFTWARE` setting. The third parameter holds the global window handle.

In the fourth parameter, I set up the system to use software vertex processing. If you have a T&L card, then you might want to set up hardware processing. In the fifth parameter, I pass in the parameter settings. The last parameter holds the returned device. For this parameter, I use a global variable of type `LPDIRECT3DDEVICE8`. This is a pointer to an `IDirect3DDevice8` interface.

Well, now that you have initialized the 3D environment, it's time to do something with it. Follow the `WinMain()` function, and you'll see that the `vRenderScene()` function is called next.

The vRenderScene() Function

Well, this is where things get interesting. The `vRenderScene()` function is short, but it actually accomplishes a lot. Here is the function again.

```
void vRenderScene(void)
{
    // Clear the scene
    pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET,
     D3DCOLOR_XRGB(150,150,255), 1.0f, 0 );
    pd3dDevice->BeginScene();
    pd3dDevice->EndScene();
    // Output the scene
    pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

All of the functions in this function are members of the `IDirect3DDevice8` interface you created in `hrInit()`.

The Clear() Function

The first thing that you do when rendering a 3D scene is clear out the video buffers with the `Clear()` function. Here is the prototype.

```
HRESULT Clear(
    DWORD Count,
    CONST D3DRECT* pRects,
    DWORD Flags,
    D3DCOLOR Color,
    float Z,
    DWORD Stencil
);
```

Most of the time, you clear the entire buffer area, but the `Clear()` function gives you the power to clear only portions if you so desire. The first parameter lets you specify how many rectangular areas you are going to clear. If you set this to `0`, then the entire viewport is cleared.

The second parameter expects pointers to the rectangles you are going to clear. The rectangles are the D3DRECT data type. This data type has four member variables: x1, y1, x2, and y2. They represent the upper-left and lower-right extents of a rectangle. Be sure to set this parameter to NULL if you used 0 in the first parameter.

The third parameter lets you specify certain flags to use with the function call. The optional values appear in Table 11.4.

The fourth parameter stores the color that the video display will be filled with if the D3DCLEAR_TARGET flag is set. The color is a 32-bit ARGB value. The fifth parameter stores the Z value to clear the z-buffer with if the D3DCLEAR_ZBUFFER flag is set. This value can be set from 0.0 to 1.0. The value 0.0 is closest to the camera, and 1.0 is farthest from the camera. The last parameter sets the integer value that will be used each time a stencil buffer entry is made. This value can be up to $2^n - 1$ in size. That's quite a large number there, eh?

Be sure to check for a D3D_OK return value when making this call. Anyway, now that I have showed you what the function does, here is how I call it.

```
pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET,
 D3DCOLOR_XRGB(150,150,255), 1.0f, 0 );
```

Because I usually like to clear the entire display, I set the first two parameters to 0 or NULL. This ensures that the entire viewport is cleared. I am only clearing the viewport so I set the next parameter to D3DCLEAR_TARGET.

Table 11.4 IDirect3DDevice8::Clear() Flags

Flag	Purpose
D3DCLEAR_STENCIL	Sets the stencil buffer to be cleared
D3DCLEAR_TARGET	Sets the render buffer to be cleared using the Color parameter
D3DCLEAR_ZBUFFER	Sets the z-buffer to be cleared with the value set in the Z parameter

To set the color, I pass in a D3DCOLOR_XRGB macro. This macro lets you pass in three values ranging from 0 to 255. The three values represent the red, green, and blue components of the color you want to use.

The last two parameters are irrelevant because I am not clearing the z-buffer or the stencil buffer. There you have it; that's how you clear the display!

The IDirect3DDevice8::BeginScene() Function

Before you can actually display any polygons within a 3D scene, you must first tell the 3D system to get into rendering mode using the BeginScene() function.

Hey, guess what: There is nothing to the BeginScene() function! That's right; you just call BeginScene(), and you are ready to rock and roll. Because the Init3D program only demonstrates how to clear a window with blue, there is no geometry to render. Therefore, you can go ahead and end the scene.

The IDirect3DDevice8::EndScene() Function

For every BeginScene() function, you must have a matching EndScene() function. All this does is tell the 3D system to shut down its rendering mode. Once a scene ends, the resulting graphic that is ready to be displayed is stored within the device.

Just like its brother BeginScene(), EndScene() has no parameters either; just call it as is.

The IDirect3DDevice8::Present() Function

Once the scene is loaded into the 3D device, it is primed to be displayed. The method you call to display renderings from the device is the Present() function. Here are the parameters for it.

```
HRESULT Present(
      CONST RECT* pSourceRect,
      CONST RECT* pDestRect,
      HWND hDestWindowOverride,
      CONST RGNDATA* pDirtyRegion
);
```

The first parameter passes a RECT-type data structure that represents the source rectangle from which to copy. You do not need to set this parameter unless you specified the D3DSWAPEFFECT_COPY or D3DSWAPEFFECT_COPY_VSYNC flags in the creation of the swap chain. If you make a rectangle that exceeds the limit of the source buffer, the rectangle is clipped to the size of the source.

The second parameter sets the destination rectangle. The same rules apply here as did in the first parameter.

The third parameter sets which window gets the output. If you set this to NULL, the program uses the window that was specified when the device was created.

The last parameter is not used in DirectX 8.0, so set it to NULL.

If you look at the way I called the Present() function, you'll see that I set every parameter to NULL.

```
pd3dDevice->Present( NULL, NULL, NULL, NULL );
```

The vCleanup() Function

Before the program exits out completely, you should clean up all of the devices and objects you created. If you take a peek at the vCleanup() function, you'll see how to clean up the necessary objects.

```
void vCleanup(void)
{
    if( pd3dDevice != NULL )
        pd3dDevice->Release();
    if( pD3D != NULL )
        pD3D->Release();
    UnregisterClass( szWndClass, g_hInstance );
}
```

The first item I clean up is the IDirect3DDevice interface that was created. I do this by using the Release() member function. There are no parameters to it; just call it as is.

The next object I release is the IDirect3D8 interface. The last thing to take care of is using the UnregisterClass() function to un-register the window that was created. You pass in the window class and window instance to this function.

TIP

Always be sure to check whether pointers are NULL before trying to run operations on them.

Your Second Direct3D Program

How's that for a title? Well, your second Direct3D program might not sound that exciting, but it will teach you some key elements. In this program, I'll show you how to set up the camera in your 3D scene. By setting up the camera, you get the scene ready to render some actual geometry. Without a camera view, you won't be able to look at anything.

Go ahead and load up the Init3DView project that is located in the Chapter11 Zip file on the CD. You might notice that the Init3DView program is similar to the program I just covered. The main difference is the inclusion of some camera variables and a few new function calls. There is no need to waste paper, so I'll only list the differences here.

The D3DXMATRIX Data Type

The beginning of the code contains the following new variables.

```
D3DXMATRIX              matProj;                    // Projection data
D3DXMATRIX              matCameraView;              // Camera data
D3DXVECTOR3             vecCameraSource;
D3DXVECTOR3             vecCameraTarget;
```

The first two variables, `matProj` and `matCameraView`, are matrixes. You use a matrix to store 3D rotation, scale, and position information. This is not really the time or place to show you everything about matrices, but I'll show you how to use them for various tasks later.

The D3DXVECTOR3 Data Type

The second two variables, `vecCameraSource` and `vecCameraTarget`, are 3D vectors. A vector stores three numbers: x, y, and z. You can use these three numbers to generate a path, or as the name implies, a vector. For an example, take a look at Figure 11.16, which shows a 2D vector. The x of the vector equals 1, and the y of the vector equals 1. What this means is that the vector or path is traveling equally in both the x and y directions. So when a 2D vector equals (1,1), the vector is pointing northeast in compass terms. Vectors are good for calculating trajectories and other things; I just use vectors here to store x, y, and z information.

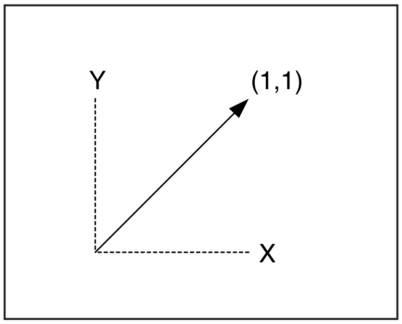


Figure 11.16

A 2D vector

The vSetupView() Function

If you follow along the `WinMain()` function, you'll see that everything looks familiar until you get to the `vSetupView()` function. This function's purpose is to set up the camera in the 3D scene. To get a better understanding of how cameras work in 3D space, look at Figure 11.17.

In Figure 11.17, you can see that the sphere is in the middle of the scene. A camera in the scene is offset from the sphere. The camera is pointing at the sphere in the

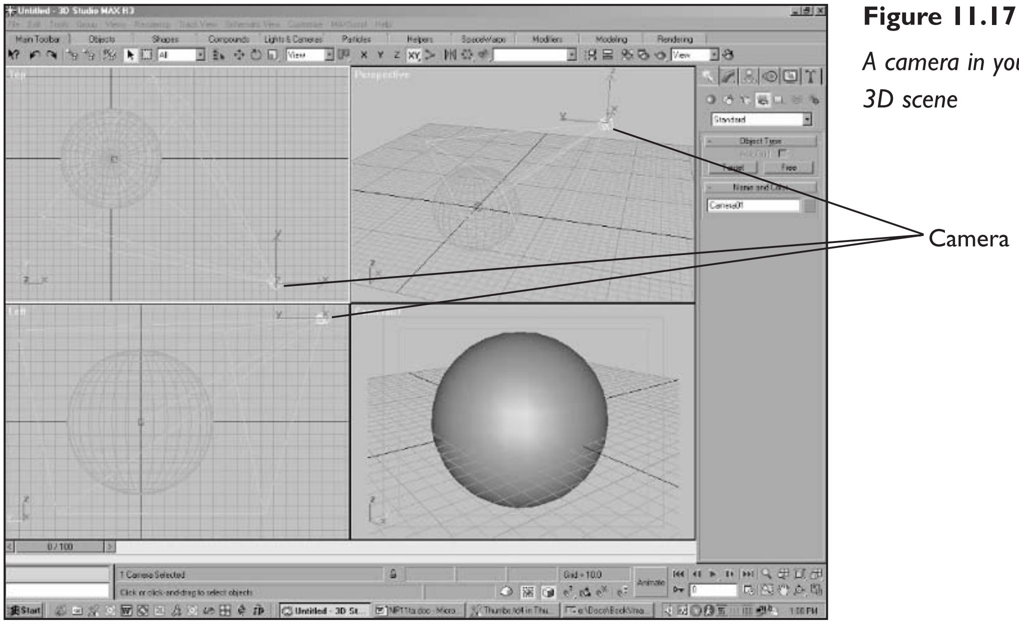


Figure 11.17

A camera in your 3D scene

center. If you look at it from the top view, you can see the layout of the entire scene. However, when you look at it from the camera's view, you can see what the 3D scene will look like when rendered. If the camera moves closer to the sphere, then the sphere appears to get bigger. If the camera moves away from the sphere, then the sphere appears to get smaller.

All of this might sound rather simple, and it really is. We all live in 3D space, so it shouldn't be a big stretch to visualize how 3D programming works. Just remember that to display 3D objects, you must have lights, cameras, and objects to display.

Setting the Camera Source

The first piece of code in `vSetupView()` sets up where the camera sits in 3D space.

```
vecCameraSource.x = 0.0f;
vecCameraSource.y = 0.0f;
vecCameraSource.z = 10.0f;
```

From these settings, the camera is sitting 10 units straight back from the *origin* of the scene.

The `f` at the end of each number means that the values are in floating-point units. Everything in 3D space uses floating-point numbers. If you try to use fixed-point values, you'll get tons of warnings.

The origin of a 3D scene is always (0,0,0). This means that the origin is the center of the universe. The three numbers refer to the x, y, and z positions.

Setting the Camera Destination

The next little piece of code sets up where the camera is looking. Because the camera's source sets where it is looking from, the camera's destination sets where it is looking at.

```
vecCameraTarget.x = 0.0f;
vecCameraTarget.y = 0.0f;
vecCameraTarget.z = 0.0f;
```

This one is pretty straightforward because the camera is looking at the origin. In the next program, you'll see how to put geometry at the origin.

Initializing the Camera's Matrix

You might recall the global variable, `matCameraView`. Well, this matrix is used to store the camera's rotation, scale, and position. Now that the camera's look from and look at settings are in place, it is time to create the matrix for the camera. To do this I use the D3DXMatrixLookAtLH() helper function.

The D3DXMatrixLookAtLH() Function

The D3DXMatrixLookAtLH() function has one purpose and one purpose only—to setup a camera view matrix. Here is the prototype.

```
D3DXMATRIX* D3DXMatrixLookAtLH(
    D3DXMATRIX* pOut,
    CONST D3DXVECTOR3* pEye,
    CONST D3DXVECTOR3* pAt,
    CONST D3DXVECTOR3* pUp
);
```

The first parameter returns a completed matrix. For this, I pass in the matCameraView variable's address. The second parameter sets the camera's origin, or eye. The third parameter sets the camera's target position. The fourth parameter sets up how the world is oriented. Normally, you set this to (0,1,0).

Here is the code I used to call this function for the Init3DView program.

```
// Set up camera view
D3DXMatrixLookAtLH(   &matCameraView,
                                  &vecCameraSource,
                                  &vecCameraTarget,
                                  &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
```

There shouldn't be too many surprises here except for maybe the D3DXVECTOR inclusion in the call. I just used it there to keep from having to declare a new variable.

The IDirect3DDevice8::SetTransform() Function

To activate the camera matrix, you must call the SetTransform() function. You use this function to set device states such as the camera view or the projection matrix. Here is the prototype.

```
HRESULT SetTransform(
     D3DTRANSFORMSTATETYPE State,
     CONST D3DMATRIX* pMatrix
);
```

The first parameter tells the 3D device interface which device-state variable is being set. The value you use to set up the camera view is the D3DTS_VIEW value. The second parameter expects the matrix with which you want to define the device-state. For this parameter, I just pass in the camera's matrix. Here is the code I used to call this function for the camera.

```
// Tell the device to use the camera view for the viewport
pd3dDevice->SetTransform( D3DTS_VIEW, &matCameraView );
```

Notice that I am using the IDirect3DDevice8 interface that was created in the hrInit3D() function earlier. The D3DTS_VIEW definition tells the system that this function call is to set up the camera view. Once the SetTransform() function is called, the camera view is active. Look at Figure 11.18 to see where the camera is in the 3D scene now.

Setting Up the Projection Matrix

The next few lines of code set up the projection matrix for the 3D scene. You use a projection matrix to set how the scene looks through the lens of the camera. Most programs in this book use the same projection matrix, so I'll be brief about it. Here is the code I use to set up the matrix.

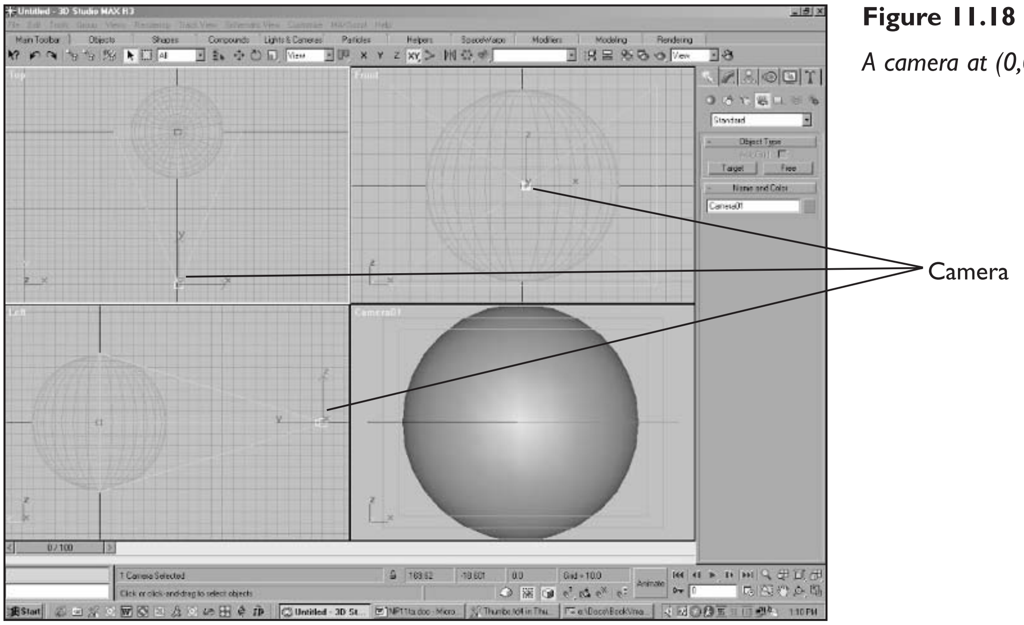


Figure 11.18

A camera at (0,0,10)

```
// Set up projection
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
// Tell the device to use the above matrix for projection
pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
```

The D3DXMatrixPerpectiveFovLH() Function

The `D3DXMatrixPerpectiveFovLH()` function might look complicated, but it's really not that bad. Gee, how many times have you heard me say that? But hey, if I can do it, then surely you can! Here is the prototype.

```
D3DXMATRIX* D3DXMatrixPerspectiveFovLH(
    D3DXMATRIX* pOut,
    FLOAT fovy,
    FLOAT Aspect,
    FLOAT zn,
    FLOAT zf
);
```

The first parameter receives the results of the function call. When the function call completes, it loads whichever matrix you put in here with the resulting values. The second parameter sets up the field of view in *radians*. For this, I almost always use `D3DX_PI/4`.

A *radian* is smaller than a degree. There are 57.29577951308232286465 radians for every degree of movement. For example, to find the number of radians in a 45-degree angle, calculate 45/57.29577951308232286465.

The third parameter sets up the aspect ratio. I normally use an aspect ratio of `1.0f`; this sets up the viewport to be square. The fourth parameter sets up the new viewplane. Any objects before this viewplane are not visible. Keep this at `1`, and you will be safe. The last parameter sets up the far viewplane. Any objects past this plane are not visible.

The last thing to do once you set up the projection matrix is call `SetTransform()`. By passing a `D3DTS_PROJECTION` value to this function, you are telling it to set up the device-state for the projection matrix.

Go ahead and run the program, and you'll see something like what is shown in Figure 11.19.

All right, that might not look too thrilling yet. Don't despair, though; in the next chapter, I'll show you how to display all sorts of stuff in the scene.

Figure 11.19

Output from the Init3DView program

Review

In this chapter, you have learned many things about the nature of 3D graphics. You really need to commit this chapter to memory, because the concepts are put to use in all 3D games. Here are some key points to remember.

- Always take into consideration your target audience's hardware limitations. If you are targeting high-end users, then you can get away with more graphical detail. If you are targeting low-end users, then you should tone down your game engine to match. The best practice is to check the player's hardware for compatibility with your game features.
- Try to use as few polygons as possible. It is best to simulate model detail with good textures instead of additional geometry.
- The z-buffer is used to sort objects in your rendering display according to their depths.
- The `IDirect3DDevice8` interface is the heart of your 3D rendering engine when using DirectX. The `IDirect3D8` interface is used to create the device interface.
- Always begin your scenes with the `BeginScene()` function and end them with the `EndScene()` function. 3D calls made outside of these two functions will cause program instability or crashing. The `Present()` function is used to display your rendered scene.
- Include support for hardware vertex processing. This can greatly improve the speed of your game when the user's hardware supports it.

CHAPTER 12

Inside Direct3D Rendering

In the last chapter, you learned the basics of DirectX's 3D system. In this chapter, I'll show you how to.

- Create and display polygons
- Add textures to your polygons
- Add real-time lighting to your scenes
- Load and render x-files
- Move objects around in a scene

You might be wondering where all of this 3D instruction is leading. Well, in a short time, I'll show you how to write a complete networked game, and it uses all 3D graphics!

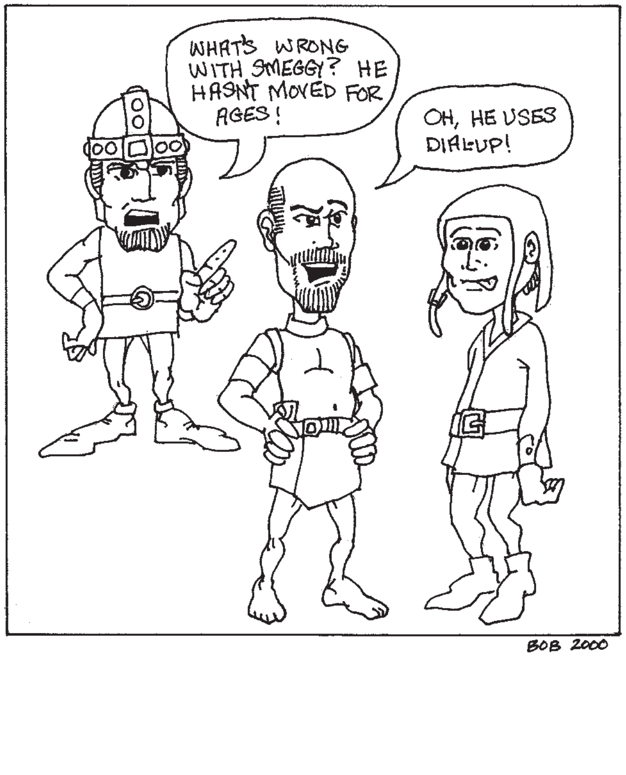


Figure 12.1

The perils of dial-up connections

Polygons

Polygons, polygons, polygons—it's all about the polygons. No matter where you turn nowadays, you hear about how many polygons per second this system or that system can push. Loyalists claim the PlayStation 2 is better than Microsoft's Xbox. Blah, blah, blah, it's all a load of you know what. Disregard what you hear in the press releases of graphics companies; they all give numbers based on unrealistic settings.

Take, for example, Nvidia's TNT2-Ultra card. Sure, it's an old card, but I have heard claims of millions of polygons per second with it. Well, I can barely get 15,000 polygons per second with one. Why is this, you might ask? When companies publish how many polygons their products can push, they give the number based on polygons that are no larger than one pixel on the screen. That's right, one pixel. As you probably know if you have played any 3D games, most objects are much larger than one pixel. So the next time you see someone expounding the virtues of a system, take it with a grain of salt. When developing a 3D game, I suggest you try it on multiple systems. You can never rely on published specifications.

Cartesian Coordinate Systems

You'll recall that a little while back I explained that a polygon is nothing more than a triangle in 3D space. In 3D space, it has x, y, and z coordinates. Figure 12.2 shows a single polygon.

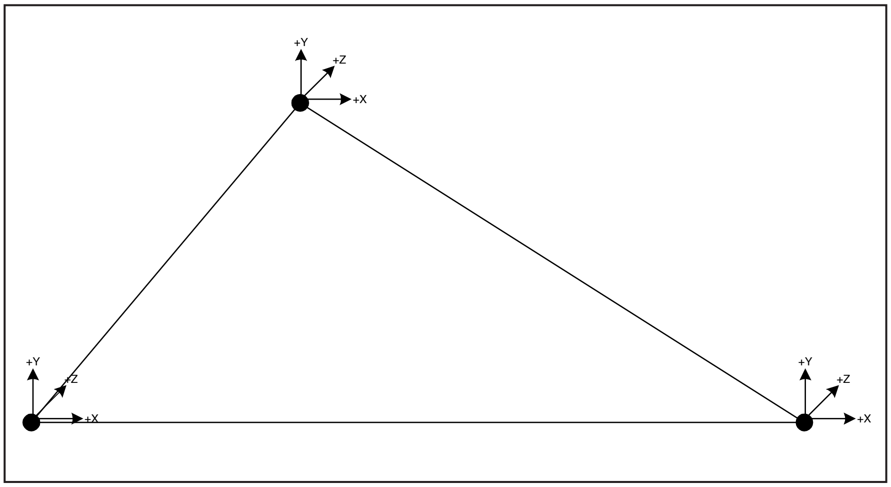


Figure 12.2

A polygon in 3D space

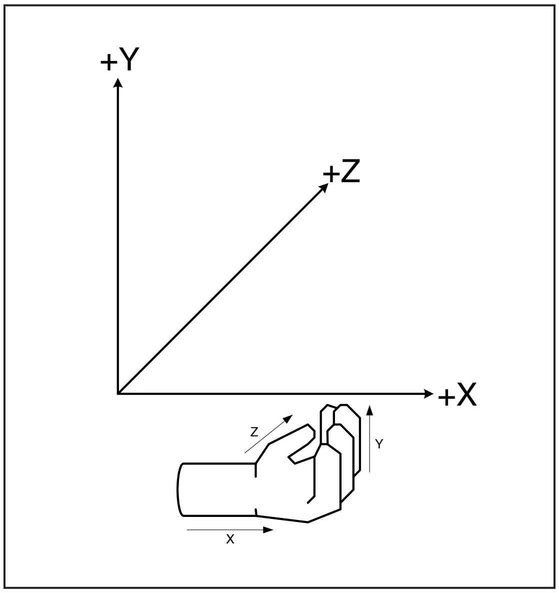


Figure 12.3

The left-handed coordinate system

Each of the points on the polygon is called a vertex and has its own x, y, and z coordinates. The 3D coordinate system in DirectX works on what is called the left-handed system. Take a peek at Figure 12.3 to see the left-handed system illustrated.

You will hear about the left-handed versus the right-handed coordinate system quite a bit. To see how this works in real life, hold your left hand out flat. Your fingers are pointing in the positive x direction. If you curl your hand, your fingers point in the positive y direction. Your thumb points in the positive z direction. The x and y positives always point in the same direction regardless of which system you use; it is the z that changes depending on which hand you use. If you switch to your right hand and perform the same routine, your thumb points the positive z-axis in the other direction. See, wasn't that easy? The bottom line is that with the left-handed system, the positive z-axis normally goes into the screen.

Primitives

Even though I have been talking about polygons exclusively, DirectX Graphics actually supports multiple types of geometry. When referring to the entire family of supported geometry, you use the term *primitives*. The six basic types of primitives supported by DirectX are *point lists, line lists, line strips, triangle lists, triangle strips,* and *triangle fans.*

Point Lists

You can use point lists to render such exciting effects as rain, snow, or stars. Basically, point lists are lists of points in 3D space. Figure 12.4 demonstrates a few points in 3D space.

You can even assign textures to points. The texture affects only the color of the points, though, and not anything in between the points.

Line Lists

If you guessed that a line list is nothing more than a list of lines in 3D space, then you win the prize. To create a line list, you have to fill an array of vertices with at least two sets of points. You can specify more than one line per list; just make sure that the number of points in your list is divisible by two. Here is a code snippet that demonstrates initializing two lines in a line list.

```
struct MYVERTEX {
    float x,y,z;
};
MYVERTEX Vertices[] = {
    { 0.0, 0.0, 0.0},
    { 0.0, 5.0, 0.0},
    { 5.0, 0.0, 0.0},
    { 5.0, 5.0, 0.0}
};
```

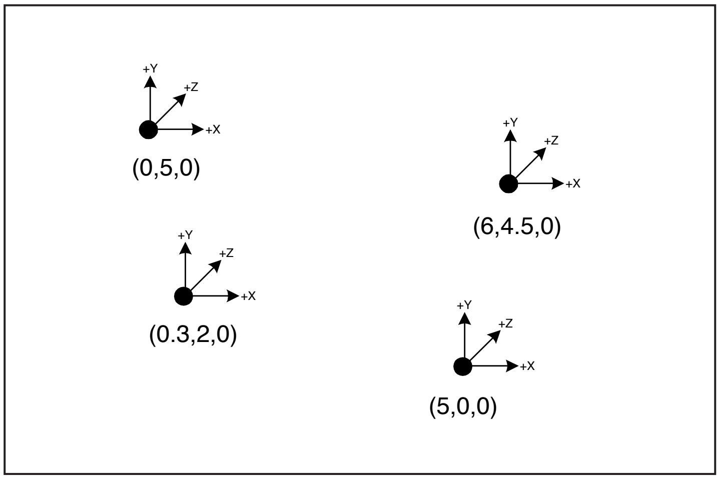


Figure 12.4

A point list

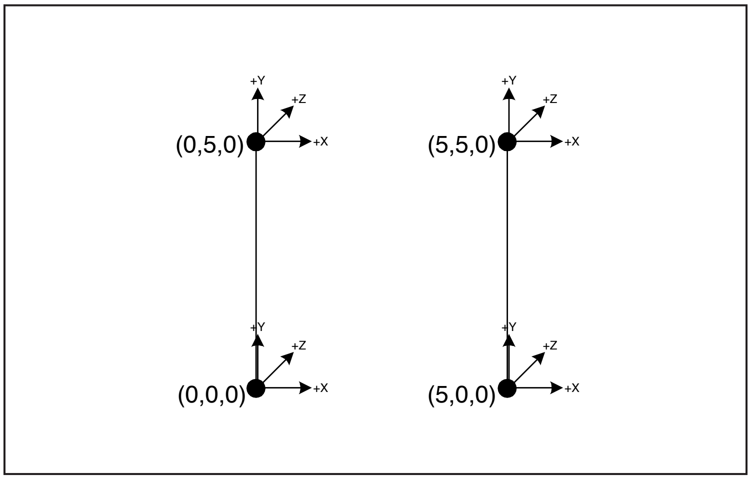


Figure 12.5

A line list

The code here results in a list of lines like those shown in Figure 12.5.

As with points, lines can have materials and textures. Only the color of the lines is affected by materials and textures, though.

Line Strips

Line strips create connected lists of lines. The space between connected lines is not filled in. One use for line strips is creating unfilled polygons. When you create a list of lines in a line strip, the vertices are continuous. In other words, there is never a break in the line. For this reason, you do not need two vertices per line. Take a look at Figure 12.6 to see a line strip.

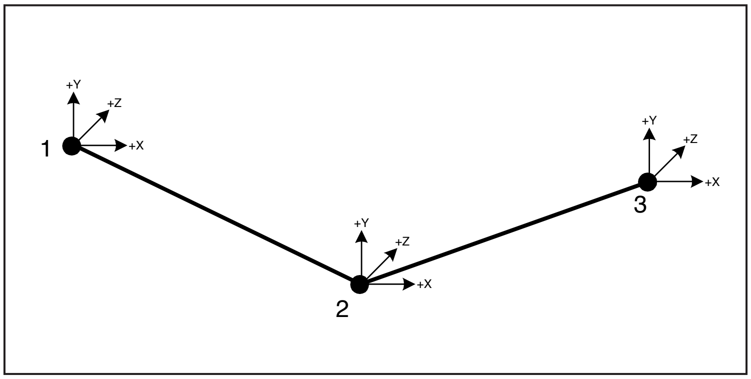


Figure 12.6

A line strip

Notice how there are only three vertices? The 3D system keeps track of the last vertex in the list and connects it to the next one entered.

Triangle Lists

Triangle lists store and display a list of filled triangles. That's right, filled triangles! This means that you can assign textures to the triangles and actually see the filled-in product. The one limitation to triangle lists is that the triangles are not connected. Snow and rain are two effects you can create with triangle lists. Because there is an entire triangle to view, the snow or rain looks much thicker than if you use lines or points. Figure 12.7 demonstrates a triangle list.

Triangle Strips

As with point strips and line strips, triangle strips consist of continuous primitives. The main advantage to triangle strips is that they use less memory than triangle lists. Because you don't have to maintain three vertices for each triangle, you can conserve more memory and transformation time. The downside to triangle strips is that all of the triangles in the list must be connected. It is also difficult to calculate strips from existing 3D objects. Figure 12.8 shows a triangle strip.

Notice how the two triangles require only four vertices. The 3D system fills in the blanks. Triangle strips can have materials and textures. In fact, I'll show you how to create both textured and non-textured triangle strips shortly.

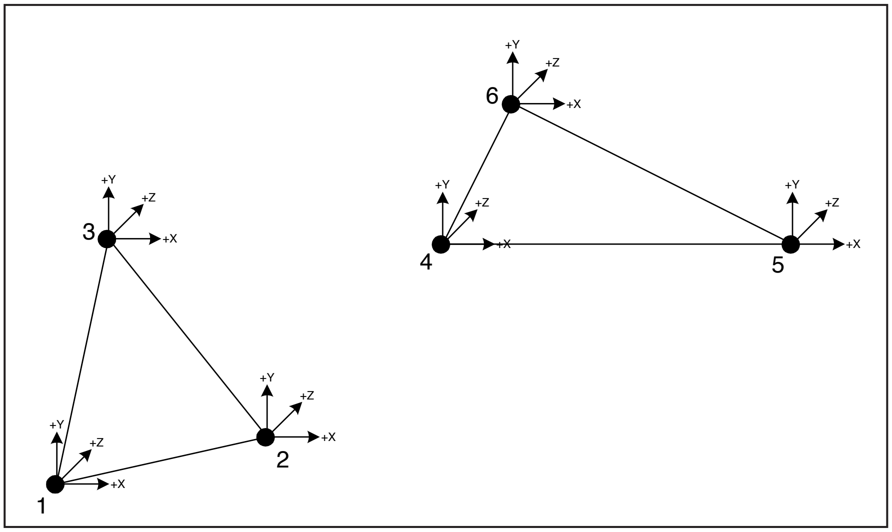


Figure 12.7

A triangle list

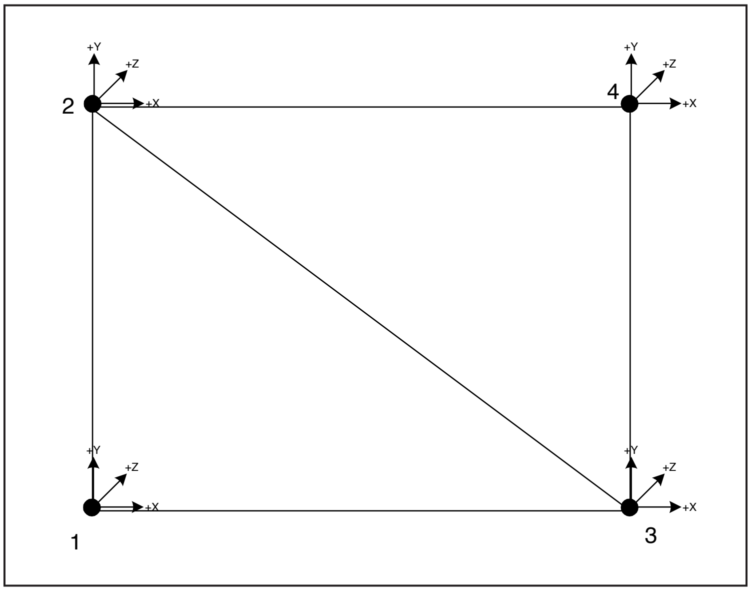


Figure 12.8

A triangle strip

Triangle Fans

Triangle fans work like triangle strips, but fans all share a common vertex. There are few uses for this type of primitive, but it does conserve a lot of memory. To see a triangle fan, take a look at Figure 12.9. Notice how the triangles all share a common vertex. This is the basis for a triangle fan. I haven't encountered a use for triangle fans yet, but you never know.

Drawing a Primitive

Now that you know all the types of primitives available in Direct3D, I think it is time to show you how to create some. If you load up the Draw3DPrimitive project, you can follow along with me. The Draw3DPrimitive program demonstrates how to create and display two triangles within a triangle list. The output from the application appears in Figure 12.10. If you skip ahead and look at Figure 12.12, you can see how the square is comprised of two triangles.

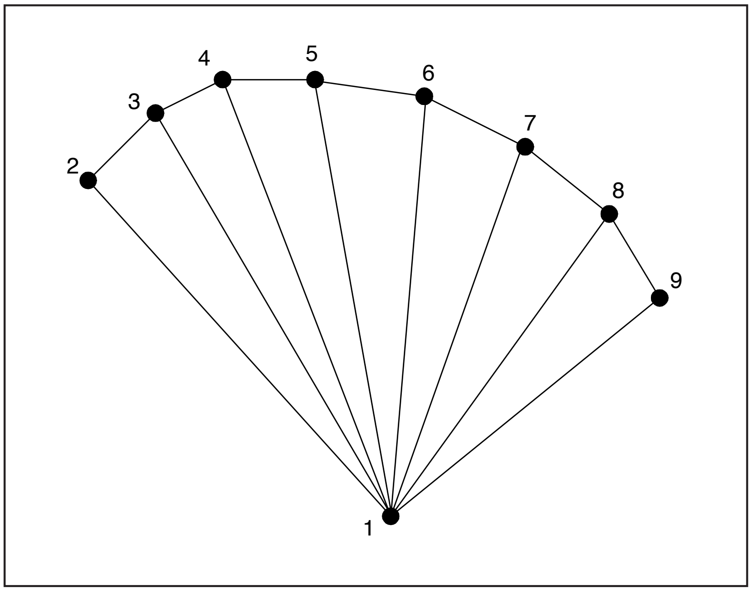


Figure 12.9

A triangle fan

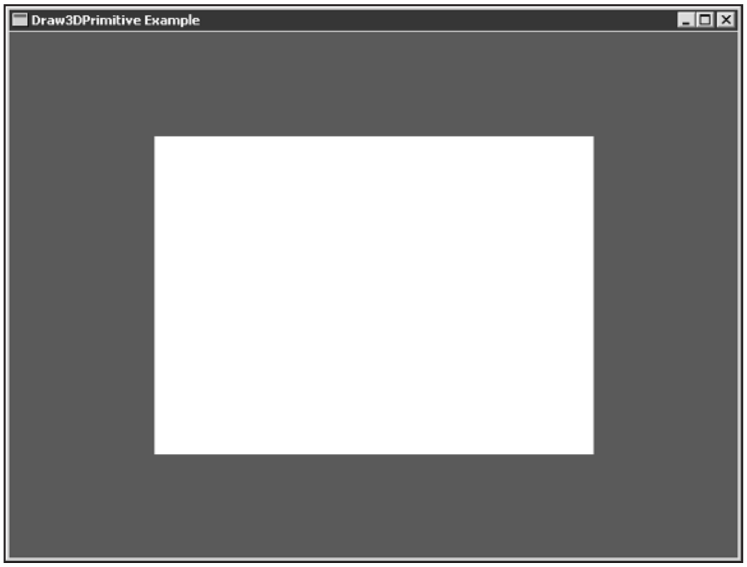


Figure 12.10

Output from the Draw3DPrimitive application

Without further ado, I present to you the source code.

```
#define INITGUID
#define WIN32_LEAN_AND_MEAN     // No MFC
#include <d3d8.h>
#include <d3dx8.h>
#include <stdio.h>

// Direct3D globals
LPDIRECT3D8                     pD3D;
LPDIRECT3DDEVICE8               pd3dDevice;
D3DXMATRIX                      matProj;                // Projection data
D3DXMATRIX                      matCameraView;          // Camera data
D3DXVECTOR3                     vecCameraSource;
D3DXVECTOR3                     vecCameraTarget;
LPDIRECT3DVERTEXBUFFER8         lpVertexBuffer = NULL;
// Definition for our vertex shader
#define D3DFVF_MYVERTEX (D3DFVF_XYZ)
// Vertex structure
typedef struct MYVERTEX
{
    float     x;
    float     y;
    float     z;
} MYVERTEX, *LPMYVERTEX;
// Global vertices
MYVERTEX     g_Vertices[4];

// Windows globals
const char          *szWndClass       = "Draw3DPrimitive";
const char          *szProgramName    = "Draw3DPrimitive Example";
HINSTANCE           g_hInstance;
HWND                g_hWnd;

// Function prototypes
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow);
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam,
 LPARAM lParam );
HRESULT hrInit3D( HWND hWnd );
void vCleanup(void);
void vRenderScene(void);
void vSetupView(void);
void vInitGeometry(void);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASS            wc;
    HWND                hWnd;
    MSG                 msg;

    // Set up and register window class
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = MsgProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( NULL, IDI_APPLICATION );
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH )GetStockObject(BLACK_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = szWndClass;
    RegisterClass(&wc);

    // Create a window
    hWnd = CreateWindowEx(WS_EX_TOPMOST,
                        szWndClass, szProgramName, WS_OVERLAPPEDWINDOW,
                        0, 0, 640,
                        480, NULL, NULL, hInstance, NULL);
    // Set up global window handles
    g_hInstance = hInstance;
    g_hWnd = hWnd;
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    if( hrInit3D(hWnd) == E_FAIL ) {
        exit(1);
    }
    // Set up the quad to be rendered
    vInitGeometry();
    // Set up the view
    vSetupView();
    // Display the 3D environment
    vRenderScene();
    // Process messages until the program is terminated
    while( GetMessage ( &msg, NULL, 0, 0 ) ) {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }

    // Clean up the environment
    vCleanup();

    return(msg.wParam);
};

LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam,
 LPARAM lParam )
{
    switch( msg ) {
        case WM_DESTROY:
            PostQuitMessage( 0 );
                return 0;
    }

    return DefWindowProc( hWnd, msg, wParam, lParam );
}

HRESULT hrInit3D( HWND hWnd )
{
    D3DPRESENT_PARAMETERS       d3dpp;
    D3DDISPLAYMODE              d3ddm;
    RECT                        rcWindowBounds;
    RECT                        rcWindowClient;

    // Create the D3D object
    if( NULL == ( pD3D = Direct3DCreate8( D3D_SDK_VERSION ) ) )
        return E_FAIL;
    // Get adapter settings
    if( FAILED( pD3D->GetAdapterDisplayMode(
     D3DADAPTER_DEFAULT, &d3ddm ) ) )
        return E_FAIL;
    // Get the size of the window
    GetWindowRect( hWnd, &rcWindowBounds );
    GetClientRect( hWnd, &rcWindowClient );
    // Clear out device parameters
    ZeroMemory( &d3dpp, sizeof(d3dpp) );
    // Set up device parameters
    d3dpp.hDeviceWindow                  = hWnd;
    d3dpp.BackBufferWidth
     = (rcWindowClient.right - rcWindowClient.left);
    d3dpp.BackBufferHeight
     = (rcWindowClient.bottom - rcWindowClient.top);
    d3dpp.BackBufferFormat               = d3ddm.Format;
    d3dpp.BackBufferCount                = 1;
    d3dpp.SwapEffect                     = D3DSWAPEFFECT_FLIP;
    d3dpp.MultiSampleType                = D3DMULTISAMPLE_NONE;
    d3dpp.EnableAutoDepthStencil         = TRUE;
    d3dpp.AutoDepthStencilFormat         = D3DFMT_D16;
    d3dpp.Windowed                       = TRUE;

    // Create the D3DDevice
    if( FAILED( pD3D->CreateDevice( D3DADAPTER_DEFAULT,
                                   D3DDEVTYPE_HAL,
                                   hWnd,
                                   D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                   &d3dpp,
                                   &pd3dDevice ) ) )
   {
        return E_FAIL;
   }
    // Turn on the zbuffer
    pd3dDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
    // Turn on ambient lighting
    pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0xffffffff );
    // Turn off D3D lighting
    pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

    return S_OK;
}

void vSetupView(void)
```

```
{
    // Set where camera is viewing from
    vecCameraSource.x = 0.0f;
    vecCameraSource.y = 0.0f;
    vecCameraSource.z = 10.0f;
    // Set what camera is looking at
    vecCameraTarget.x = 0.0f;
    vecCameraTarget.y = 0.0f;
    vecCameraTarget.z = 0.0f;
    // Set up camera view
    D3DXMatrixLookAtLH( &matCameraView,
                                  &vecCameraSource,
                                  &vecCameraTarget,
                                  &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
    // Tell the device to use the camera view for the viewport
    pd3dDevice->SetTransform( D3DTS_VIEW, &matCameraView );
    // Set up projection
    D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
    // Tell the device to use the above matrix for projection
    pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
}

void vCleanup(void)
{
    if( lpVertexBuffer != NULL )
        lpVertexBuffer->Release();
    if( pd3dDevice != NULL )
        pd3dDevice->Release();
    if( pD3D != NULL )
        pD3D->Release();
    UnregisterClass( szWndClass, g_hInstance );
}

void vRenderScene(void)
{
    D3DXMATRIX     matTranslation;

    // Clear the backbuffer and the zbuffer
    pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
 D3DCOLOR_XRGB(0,0,255), 1.0f, 0 );
```

```
    // Begin rendering
    pd3dDevice->BeginScene();
    // Set default position, scale, and rotation
    D3DXMatrixIdentity( &matTranslation );
    pd3dDevice->SetTransform(D3DTS_WORLD, &matTranslation);
    // Set the vertex buffer to render
    pd3dDevice->SetStreamSource(0, lpVertexBuffer, sizeof(MYVERTEX));
    // Set the vertex format
    pd3dDevice->SetVertexShader(D3DFVF_MYVERTEX);
    // Draw the primitive
    pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
    // Stop rendering
    pd3dDevice->EndScene();
    // Output the scene
    pd3dDevice->Present( NULL, NULL, NULL, NULL );
}

void vInitGeometry(void)
{
    void        *lpVertices;
    float       fSizeX = 5.0f;
    float       fSizeY = 5.0f;

    // Create the geometry
    g_Vertices[0].x = -fSizeX/2;
    g_Vertices[0].y = fSizeY/2;
    g_Vertices[0].z = 0.0f;
    g_Vertices[1].x = -fSizeX/2;
    g_Vertices[1].y = -fSizeY/2;
    g_Vertices[1].z = 0.0f;
    g_Vertices[2].x = fSizeX/2;
    g_Vertices[2].y = fSizeY/2;
    g_Vertices[2].z = 0.0f;
    g_Vertices[3].x = fSizeX/2;
    g_Vertices[3].y = -fSizeY/2;
    g_Vertices[3].z = 0.0f;
    // Create the vertex buffer
    if(FAILED(pd3dDevice->CreateVertexBuffer(4*sizeof(MYVERTEX),
 0, D3DFVF_MYVERTEX, D3DPOOL_DEFAULT, &lpVertexBuffer))) {
        return;
    }
    // Lock the buffer
    if(FAILED(lpVertexBuffer->Lock(0, sizeof(g_Vertices),
     (BYTE**)&lpVertices, 0))) {
        return;
    }
    // Copy the g_Vertices information into the buffer
    memcpy(lpVertices, g_Vertices, sizeof(g_Vertices));
    // Unlock the buffer
    lpVertexBuffer->Unlock();
}
```

The Header Information

To use Direct3D, you must include a few header files.

```
#define INITGUID
#define WIN32_LEAN_AND_MEAN
#include <d3d8.h>
#include <d3dx8.h>
#include <stdio.h>
```

Not only did I include the d3d8.h and d3dx8.h files, but I also set up a couple of definitions. The `INITGUID` definition sets up some helper defines. The `WIN_32_LEAN_AND_MEAN` definition ensures that MFC is not used.

The d3d8.h and d3dx8.h header files are required to use Direct3D. Remember to always include them. Lastly, you need to include the stdio.h file for standard functionality.

The Globals

The next block of code you come across declares all of the global variables used by the program. Normally, I don't like to use globals, but they make for easier-to-read code examples.

```
LPDIRECT3D8                     pD3D;
LPDIRECT3DDEVICE8               pd3dDevice;
D3DXMATRIX                      matProj;
D3DXMATRIX                      matCameraView;
D3DXVECTOR3                     vecCameraSource;
D3DXVECTOR3                     vecCameraTarget;
LPDIRECT3DVERTEXBUFFER8         lpVertexBuffer = NULL;
```

The pD3D variable of type IDirect3D8 is your interface into the 3D system. It creates your device. The pd3dDevice variable of type IDirect3DDevice8 is the holder of the created 3D device. This device processes most 3D functions.

The matProj variable, which is a 3D matrix, is used for the projection matrix of the scene. The matCameraView matrix sets up the camera's position and orientation in the scene.

You use two vectors in this program, vecCameraSource and vecCameraTarget. Keep these global so that you can alter the camera's position during the application if you so desire.

The last and perhaps the most interesting variable declared is lpVertexBuffer, which is a pointer to an IDirect3DVertexBuffer8 interface.

Introduction to the IDirect3DVertexBuffer8 Interface

When displaying geometry in DirectX 8, you must use what are called *vertex buffers*. Basically, these buffers contain the information for all of your objects. Because a 3D object consists of vertices, this is appropriate. Take a look at Figure 12.11 for a refresher of how vertices make up a 3D object.

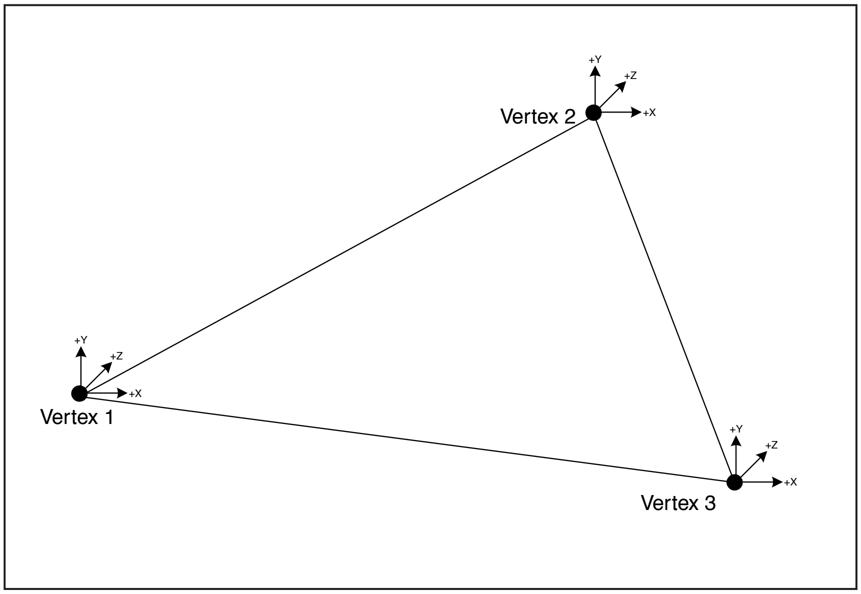


Figure 12.11

Three vertices form a triangle

Before you can use a buffer, you must first create it with the IDirect3DDevice8 interface object's CreateVertexBuffer() function.

Once you create the vertex buffer, you then go on to lock it in place using the IDirect3DvertexBuffer::Lock() function. The reason why you have to lock vertex buffers is that they are maintained in dynamic memory space. The lock is performed to reserve and hold the memory for modification.

During the lock phase, you copy the vertex data into the buffer. You can accomplish this fairly straightforward process using a standard memcpy() function call.

The last thing you have to do as far as initialization is concerned is unlock the buffer using the IDirect3DvertexBuffer::Lock() function call. Once you follow the previous steps, you can go on to use the buffer with the IDirect3DDevice8::SetStreamSource() function call. I'll go into this topic more when the code comes up.

Introduction to Vertex Formats

The next block of code you encounter defines the vertex data that goes into the vertex buffer. You do this by creating a custom vertex data structure.

```
#define D3DFVF_MYVERTEX (D3DFVF_XYZ)
typedef struct MYVERTEX {
    float     x;
    float     y;
    float     z;
} MYVERTEX, *LPMYVERTEX;
MYVERTEX     g_Vertices[4];
```

The define tells the 3D system how to render the vertices. The D3DFVF_XYZ value specified here tells the system that the vertex consists of three coordinates: x, y, and z. At a bare minimum, you should specify this value for flexible vertex formats. Because this program doesn't do any texture mapping or lighting, you don't need to worry about any additional vertex information.

The MYVERTEX data structure defined next holds vertex information before being loaded in the vertex buffer. In this instance, you need to store only three variables: the x, y, and z coordinates for each vertex.

The last thing to do here is create a global array of vertices. Because this program creates a square and a square can be created with only four vertices, that is all you need to put in the array.

Windows Globals and Prototypes

The next few globals are standard variables you should set for most Windows programs. These should be familiar to you by now.

```
const char       *szWndClass          = "Draw3DPrimitive";
const char       *szProgramName       = "Draw3DPrimitive Example";
HINSTANCE        g_hInstance;
HWND             g_hWnd;
// Function prototypes
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow);
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam,
 LPARAM lParam );
HRESULT hrInit3D( HWND hWnd );
void vCleanup(void);
void vRenderScene(void);
void vSetupView(void);
void vInitGeometry(void);
```

The `WinMain()` function is required for Windows programming and is used as the main loop of the program. The `MsgProc()` function is required to process any messages that get sent to the application. The `vCleanup()` function is called whenever the program exits. You should make sure you always clean up any memory or objects allocated during the application.

The `vRenderScene()` function displays the polygon that this program creates. The `vSetupView()` function sets up the viewport, camera, and projection for the scene. This affects the location from which the polygons are viewed. The `vInitGeometry()` function creates and stores the polygon information in the vertex buffer.

The WinMain() Function

By now, you have been through the `WinMain()` function several times. There is nothing really significant here except the calls to `hrInit3D()`, `vInitGeometry()`, `vSetupView()`, `vRenderScene()`, and `vCleanup()`.

Because the functions are called in successive order, there really isn't much to discuss about them. You might want to note that the scene is only rendered once, and nothing gets updated in the message loop. You could put the `vRenderScene()` function into the message loop, but all that does is eat CPU cycles. Later, I'll show you programs that require the render function to be present in the message loop.

The hrInit3D() Function

If you recall the hrInit3D function from Chapter 11, "DirectX Graphics," you'll find that this one isn't much different. The steps for initializing a 3D environment are as follows.

1. Create the IDirect3D8 interface.
2. Retrieve information about video cards in the system.
3. Create an IDirect3DDevice8 interface.

Even though I covered these steps before, you can learn some new information from this particular initialization function.

```
// Turn on the zbuffer
pd3dDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
// Turn on ambient lighting
pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0xffffffff );
// Turn off D3D lighting
pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
```

The three rendering states that you need to set for this application are for the z-buffer, ambient lighting, and hardware lighting.

Technically, you do not have to have a z-buffer present to render 3D geometry. However, you need to use it later, so you might as well get used to it now. Luckily, DirectX makes turning on z-buffers very easy because all you have to do is set the D3DRS_ZENABLE render state to TRUE.

The D3DRS_AMBIENT setting sets how much default light will be in a 3D scene. This program does not use any hardware lights, so you need to brighten up the entire scene with ambient lighting. When you have ambient lighting, nothing in the scene is lit darker than the ambient setting. Once you start using hardware lighting, you will want to turn ambient lighting down to a minimum because it tends to wash out the scene.

The D3DRS_LIGHTING state sets whether the scene uses hardware lighting. This is not a lighting demo, so just keep it at FALSE for now.

The vInitGeometry() Function

Ah, here it is, the fun part! This function is where all the magic happens as far as geometry is concerned. In this function, you'll accomplish the following tasks.

1. Create a vertex buffer.

2. Lock the vertex buffer.
3. Copy data into the vertex buffer.
4. Unlock the vertex buffer.

Before you create a vertex buffer, you probably should go ahead and set up the vertex information to put into it. You do this by either loading in vertices or creating them from scratch. In this program, you create them from scratch, so watch closely.

```
g_Vertices[0].x = -fSizeX/2;
g_Vertices[0].y = fSizeY/2;
g_Vertices[0].z = 0.0f;
g_Vertices[1].x = -fSizeX/2;
g_Vertices[1].y = -fSizeY/2;
g_Vertices[1].z = 0.0f;
g_Vertices[2].x = fSizeX/2;
g_Vertices[2].y = fSizeY/2;
g_Vertices[2].z = 0.0f;
g_Vertices[3].x = fSizeX/2;
g_Vertices[3].y = -fSizeY/2;
g_Vertices[3].z = 0.0f;
```

Each vertex consists of an x, y, and z coordinate. To prepare the geometry information, you need to set up these coordinates. If you recall the information about triangle strips, this doesn't seem too strange. The `fSize` variables set the size of the square you are creating. Putting the value into variables makes things easier to adjust. Take a peek at Figure 12.12 to see how the square is made up.

The IDirect3DDevice8::CreateVertexBuffer() Function

The next chunk of code is where you create the vertex buffer to hold your vertex information.

```
if(FAILED(pd3dDevice->CreateVertexBuffer(
     4*sizeof(MYVERTEX),
     0,
     D3DFVF_MYVERTEX,
     D3DPOOL_DEFAULT,
     &lpVertexBuffer))) {
          return;
}
```

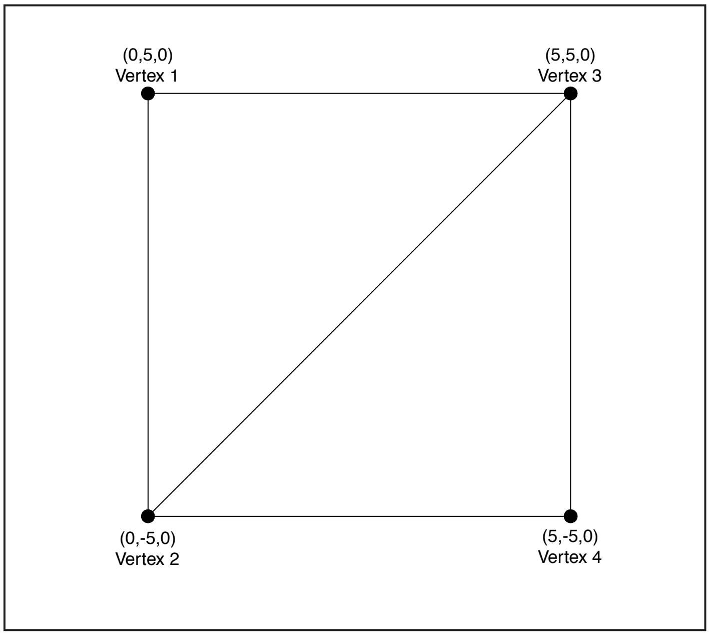


Figure 12.12

The vertex information for a square

The `CreateVertexBuffer()` function belongs to the `IDirect3DDevice8` interface and is pretty important as far as geometry is concerned. To display vertices, you must put them in a buffer.

One thing to keep in mind with vertex buffers is that you should try to limit the number of vertex buffers you create. If you were going to display a grid of 3D squares, then you would want to put them all in one vertex buffer. The only reason you should have to create a new buffer is if you want to change the material or texture of your object. The reason you want to keep the number of buffers to a minimum is that each buffer in the system takes up valuable memory. It is also taxing on the 3D hardware to switch between multiple buffers. The less switching you have to do, the better.

The prototype of the function looks like the following code.

```
HRESULT CreateVertexBuffer(
    UINT Length,
    DWORD Usage,
    DWORD FVF,
    D3DPOOL Pool,
    IDirect3DVertexBuffer8** ppVertexBuffer
);
```

The first parameter expects the length of the buffer. If you are not using a flexible vertex format, then this parameter is ignored. You need at least one vertex present if you are specifying a size here. I normally use the number of vertices multiplied by the size of my vertex format.

The second flag specifies how the buffer is to be used. Table 12.1 describes the available flags for this parameter.

The third parameter expects the flexible vertex format you want to use with the vertices. If you are not using a flexible vertex format, then you should set this parameter to `0`. The fourth parameter specifies which memory pool the buffer will occupy. I just pass `D3DPOOL_DEFAULT` here to use the default memory pool. The last parameter retrieves a pointer to the `IDirect3DVertexBuffer8` object that is created upon success.

Well, there you have it, the creation of a vertex buffer. Once the call succeeds, the `lpVertexBuffer` pointer points to the newly created buffer interface. Once you set up this pointer, it is time to load the vertex information. But before you can do the loading, you must first lock the buffer.

Table 12.1 CreateVertexBuffer() Usage Flags

Function	Purpose
`D3DUSAGE_DONOTCLIP`	Only use this flag if you know that the vertices you are loading will never require clipping. This flag tells the buffer to not perform clipping.
`D3DUSAGE_DYNAMIC`	This flag sets up the buffer to use dynamic memory.
`D3DUSAGE_RTPATCHES`	This flag sets up the buffer to draw high-order primitives.
`D3DUSAGE_NPATCHES`	This flag sets up the buffer to draw N patches.
`D3DUSAGE_POINTS`	This flag tells the buffer that it will be drawing points or point sprites.
`D3DUSAGE_SOFTWAREPROCESSING`	This flag tells the buffer to be used with software processing and not hardware.
`D3DUSAGE_WRITEONLY`	This flag tells the system that the buffer can only be written to.

The IDirect3DVertexBuffer8::Lock() Function

Before you can modify the data within a vertex buffer, you must first lock it down. You can think of the buffer as a book in a library. When you lock a buffer, you are checking it out and no one else can use it. When you are done, you unlock it. This lets others read it or modify it. The Lock() function has the following prototype.

```
HRESULT Lock(
    UINT OffsetToLock,
    UINT SizeToLock,
    BYTE** ppbData,
    DWORD Flags
);
```

The first parameter specifies an optional offset into the buffer to be locked down. Normally, you just set this to zero. The second parameter sets how many bytes of data need to be locked. If you want to specify the entire buffer, just set this to zero. The third parameter receives a BYTE pointer that points to the locked data.

The last parameter lets you specify certain flags for the operation. Table 12.2 lists the flags for this parameter.

Table 12.2 Lock() Flags

Function	Purpose
D3DLOCK_DISCARD	Causes the system to write to a new memory location.
D3DLOCK_NOOVERWRITE	Tells the system that the vertices have not been rendered lately. This can be used for certain optimizations.
D3DLOCK_NOSYSLOCK	Causes the system to not wait for the unlock to occur. This can help free up the system for other tasks.
D3DLOCK_READONLY	Tells the system that the buffer will not be written to. This cannot be set if the D3DUSAGE_WRITEONLY flag was used in the creation of the buffer.

Once you have the buffer locked, you can go on to put the data into it using a simple memcpy() function.

```
memcpy(lpVertices, g_Vertices, sizeof(g_Vertices));
```

For the memcpy() function, you use the pointer to the buffer that was retrieved in the Lock() function call. You just load it up with the vertex information created earlier. Once the memory copy is completed, you can finish the operation by unlocking the buffer.

The IDirect3DVertexBuffer8::Unlock() Function

You must call the IDirect3DVertexBuffer8::Unlock() function once for every Lock() that is performed. Once a vertex buffer is unlocked, it is ready for use in rendering.

Luckily for you, this function has no parameters to worry about. Just call it like this:

```
lpVertexBuffer->Unlock();
```

That's all there is to it! You just completed all of the required steps for creating a 3D object. Now it is time to set up the view and then render the square.

The vSetupView() Function

The vSetupView() function for this application is just like the one you used for the last program in Chapter 11, "DirectX Graphics." If you are feeling rusty about this, please go back and read up on it again.

```
vecCameraSource.x = 0.0f;
vecCameraSource.y = 0.0f;
vecCameraSource.z = 10.0f;
vecCameraTarget.x = 0.0f;
vecCameraTarget.y = 0.0f;
vecCameraTarget.z = 0.0f;
D3DXMatrixLookAtLH( &matCameraView,
                                      &vecCameraSource,
                                      &vecCameraTarget,
                                      &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
pd3dDevice->SetTransform( D3DTS_VIEW, &matCameraView );
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
```

This function completes the following steps.

1. Set the camera back 10 units from the origin of the world.
2. Set up the camera's viewport.
3. Set up the projection matrix.

That's about it for setting up the view. Now it's time to display the square into which you have put so much work. Before you go, though, take a look at Figure 12.13 to visualize how the camera is set up in the scene.

The vRenderScene() Function

Welcome to the wide world of scene rendering. In this incarnation of the `vRenderScene()` function, you display the wonderful geometry created earlier. The first item of note is a matrix variable.

```
D3DXMATRIX      matTranslation;
```

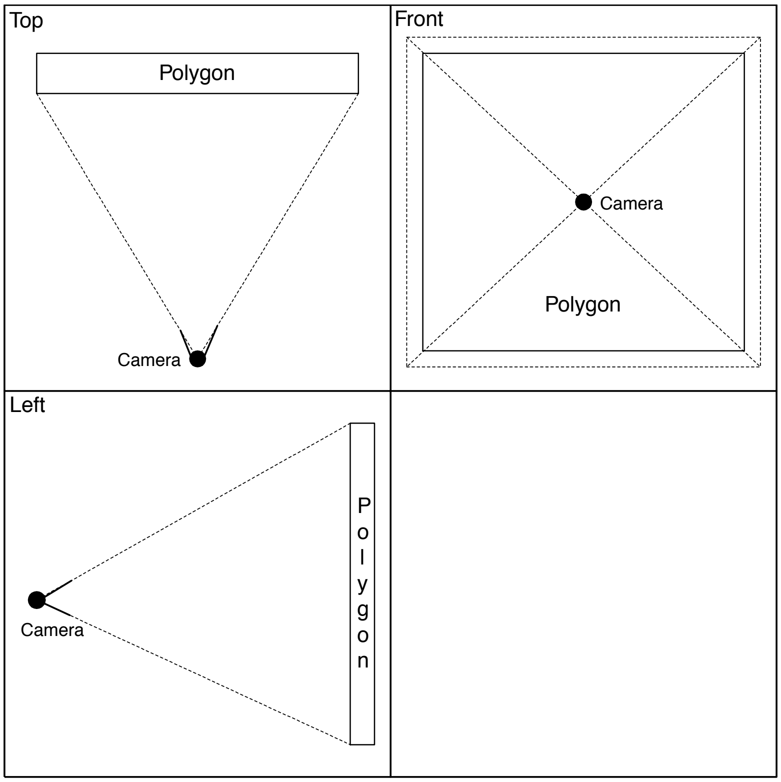


Figure 12.13

The camera's position in the scene

The Translation Matrix

The matTranslation matrix is going to position your 3D geometry in the scene. You actually can do without this if you only want to place your geometry in the center of the world. But, as they say, there is no better time than the present, so why not learn it now? I call this particular matrix matTranslation because it is only going to set up the position, or translation, of your object. The bottom line here is that *translation* means position in 3D-speak.

Clearing the Scene

The next item up for bid is the Clear() function. This one should look familiar to you by now.

```
pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
 D3DCOLOR_XRGB(0,0,255), 1.0f, 0 );
```

By specifying both the D3DCLEAR_TARGET and D3DCLEAR_ZBUFFER flags, you are telling the system to clear the display and the z-buffer.

Beginning the Rendering

Behind door number three, you have the Begin() function. If you recall from earlier in the book, this function tells the 3D hardware to start rendering a new scene.

```
pd3dDevice->BeginScene();
```

TIP

When you are using a z-buffer, you need to be sure to clear it every frame or you might get unexpected results.

CAUTION

Always have a matching EndScene() **function call for every** BeginScene() **call.**

Setting the Geometry's Translation

The following piece of code sets up the default position for the geometry to be rendered.

```
D3DXMatrixIdentity( &matTranslation );
pd3dDevice->SetTransform(D3DTS_WORLD, &matTranslation);
```

Well, now it is time to use that translation matrix you defined earlier. The `D3DXMatrixIdentity()` function sets up a matrix to its default values. You should call this function whenever you create a matrix to initialize it. Because you want to display your geometry at the center of the world, the default matrix is perfect for this operation.

The next item of business is calling the `SetTransform()` function to set up the transformation matrix for the world. This sets where the next geometry to be displayed will appear in the scene.

Drawing the Geometry

Before you can draw your geometry, you must first tell the 3D system from where to pull the vertex information. This is as easy as calling the `SetStreamSource()` function.

The IDirect3DDevice8::SetStreamSource() Function

Whenever you want to use a vertex buffer, you must first tell the system to use its data. This is where the `SetStreamSource()` function comes into play. The prototype for this function looks like this:

```
HRESULT SetStreamSource(
    UINT StreamNumber,
    IDirect3DVertexBuffer8* pStreamData,
    UINT Stride
);
```

The first parameter specifies which stream you are using. I pass a `0` for this parameter. The second parameter sets from where the stream data is coming. You should pass the pointer to your vertex buffer here. The last parameter sets the size of each vertex. For this application, you should pass the `sizeof()` your vertex structure here.

Here is how the current application sets the stream.

```
// Set the vertex buffer to render
pd3dDevice->SetStreamSource(0, lpVertexBuffer, sizeof(MYVERTEX));
```

Once you've set up the stream, it is time to tell the system which vertex shader is being used.

The IDirect3DDevice8::SetVertexShader() Function

The `SetVertexShader()` function is rather straightforward because all you have to pass to it is a defined flexible vertex format. Here is the prototype.

```
HRESULT SetVertexShader(
    DWORD Handle
);
```

In this case, you should pass in the `D3DFVF_MYVERTEX` definition created at the beginning of the program. This definition tells the 3D system that the vertices contain x, y, and z information only. In future programs, you will see how to add other information to each vertex.

The IDirect3DDevice8::DrawPrimitive() Function

You next use the `DrawPrimitive()` function to actually draw the geometry that you have worked so hard to set up. Here is the prototype.

```
HRESULT DrawPrimitive(
    D3DPRIMITIVETYPE PrimitiveType,
    UINT StartVertex,
    UINT PrimitiveCount
);
```

The first parameter expects the type of primitive you are trying to draw. The following types are supported.

- `D3DPT_POINTLIST`
- `D3DPT_LINELIST`
- `D3DPT_LINESTRIP`
- `D3DPT_TRIANGLELIST`
- `D3DPT_TRIANGLESTRIP`
- `D3DPT_TRIANGLEFAN`

The second parameter expects the index into the vertex buffer where geometry should start. Normally, you set this to `0`, but you could set it to any point within the vertex buffer if you so desire. For this program, set this parameter to `D3DPT_TRIANGLESTRIP`.

The last parameter expects the number of primitives to render. The vertex buffer you are creating for this application has two polygons in it, so you should specify a `2` here.

Ending the Rendering

Well, all good things must come to an end, so you'd better call the `EndScene()` function before leaving.

```
pd3dDevice->EndScene();
```

Right after this is done, you can present the rendering to the window. That's all there is to it! Yeah, I know it isn't very simple, but once you get the hang of it, it's not that bad.

The vCleanup() Function

Before the program completely disappears into the wild blue yonder, you should clean up all allocated interfaces and memory.

```
if( lpVertexBuffer != NULL )
    lpVertexBuffer->Release();
if( pd3dDevice != NULL )
    pd3dDevice->Release();
if( pD3D != NULL )
    pD3D->Release();
UnregisterClass( szWndClass, g_hInstance );
```

The first thing to clean up is the vertex buffer you created. The next item to go is the 3D device, and the last item to release is the `IDirect3D8` interface. Lastly, you should unregister the window class you created in the `WinMain()` function.

Show Me the Textures!

Although you now know how to display polygons, they sure don't look very exciting. The key to good-looking 3D applications is the textures. Without good textures, your game is going to look very bland. Go ahead and load up the Draw3DTexturedPrimitive workspace from the book's CD.

In this program, you are going to learn how to add textures to your polygons. Most of the code from the last application is replicated here with a few additions.

CAUTION

On some systems, the sample programs can take a minute to initialize. If you do not see graphics immediately, give the program a few seconds to render before closing the window. If you still have problems, e-mail me at webmaster@lostlogic.com.

Before I get going with the code, take a look at Table 12.3 to see the libraries required to compile the code.

The Header Changes

If you compare the global variables in this program to the ones in the last program, you'll notice a new global present.

```
LPDIRECT3DTEXTURE8       g_pTexture    = NULL;
```

This global variable is a pointer to an `IDirect3DTexture8` interface. The `IDirect3DTexture8` interface manages all textures you use in a 3D scene.

The D3DFVF_TEX1 Vertex Format

The next difference you might notice is that the `MYVERTEX` flexible vertex format now has some new members.

```
#define D3DFVF_MYVERTEX (D3DFVF_XYZ|D3DFVF_TEX1)
typedef struct MYVERTEX {
     float     x;
     float     y;
     float     z;
     float     u;
     float     v;
} MYVERTEX, *LPMYVERTEX;
MYVERTEX     g_Vertices[4];
```

Table 12.3 Draw3DTexturedPrimitive Project Libraries

Library	Description
d3d8.lib	Direct3D main library
d3dx8.lib	Direct3D helper library

The texture coordinates for the vertex format are the most important things added here. You do this by adding the D3DFVF_TEX1 flag to the format. When you have this flag present, you must include two floating-point variables in the format's structure. These are represented here as u and v.

The vInitGeometry() Changes

If you follow the code, you'll see that hrInit3D() and vSetupView() are the same as they were in the last program. Zoom on over to vInitGeometry(), and I'll show you the differences there and what they mean. Okay, here is the code for the new and improved version of vInitGeometry().

```
void      *lpVertices;
float     fSizeX = 5.0f;
float     fSizeY = 5.0f;
// Create the geometry
g_Vertices[0].x = -fSizeX/2;
g_Vertices[0].y = fSizeY/2;
g_Vertices[0].z = 0.0f;
g_Vertices[0].u = 0.0f;
g_Vertices[0].v = 0.0f;
g_Vertices[1].x = -fSizeX/2;
g_Vertices[1].y = -fSizeY/2;
g_Vertices[1].z = 0.0f;
g_Vertices[1].u = 0.0f;
g_Vertices[1].v = 1.0f;      // Y
g_Vertices[2].x = fSizeX/2;
g_Vertices[2].y = fSizeY/2;
g_Vertices[2].z = 0.0f;
g_Vertices[2].u = -1.0f;      // X
g_Vertices[2].v = 0.0f;
g_Vertices[3].x = fSizeX/2;
g_Vertices[3].y = -fSizeY/2;
g_Vertices[3].z = 0.0f;
g_Vertices[3].u = -1.0f;      // X
g_Vertices[3].v = 1.0f;      // Y
// Create the vertex buffer
if(FAILED(pd3dDevice->CreateVertexBuffer(4*sizeof(MYVERTEX), 0,
 D3DFVF_MYVERTEX, D3DPOOL_DEFAULT, &lpVertexBuffer))) {
     return;
}
```

```
// Lock the buffer
if(FAILED(lpVertexBuffer->Lock(0, sizeof(g_Vertices),
 (BYTE**)&lpVertices, 0))) {
    return;
}
// Copy the g_Vertices information into the buffer
memcpy(lpVertices, g_Vertices, sizeof(g_Vertices));
// Unlock the buffer
lpVertexBuffer->Unlock();
If( FAILED( D3DXCreateTextureFromFile( pd3dDevice,
 "texture.jpg", &g_pTexture ) ) ) {
    return;
}
```

Setting the Texture Coordinates

Did you notice the initialization of the u and v values for each vertex? If you didn't, go check it out now. The u and v coordinates work kind of like x and y coordinates. Take a peek at Figure 12.14 to visualize how the square you are initializing is set up.

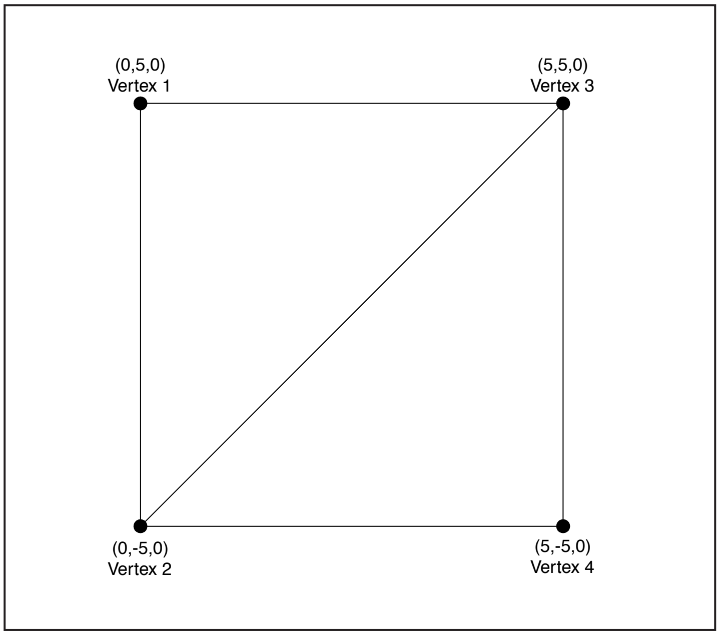


Figure 12.14

The vertices of the textured square

U and V Texture Coordinates

For each vertex, you have to set the u and v values. These two values tell the 3D system how the texture will appear on the geometry. The first set of uv coordinates specifies where the texture will start drawing from the upper-left side of the square. The second set of uv coordinates specifies where the texture will be drawn from the lower-left side of the square. The third set of uv coordinates specifies where the texture will be drawn from the upper-right side of the square. The fourth set of uv coordinates specifies where the texture will be drawn from the lower-right side of the square.

If you look at Figure 12.15, you can see how the u and v values correspond to the square. You might also notice that the coordinates are inverted. This keeps the texture map facing the correct direction.

With texture coordinates, values of (0.0,0.0) indicate the upper-left portion of the texture; (0.0,1.0) equates to the lower-left; (1.0,0.0) equates to the upper-right; and (1.0,1.0) equates to the lower-right. Look at Figure 12.16 to see this in action.

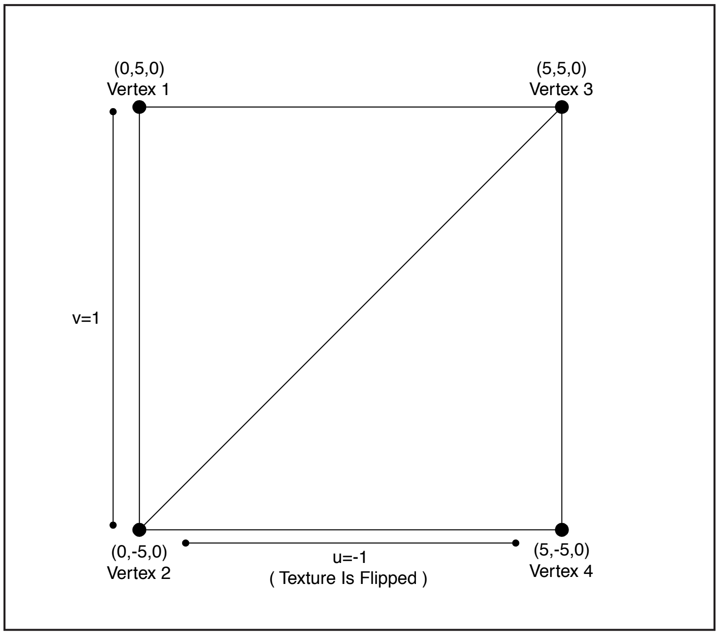


Figure 12.15

Texture values and their positions on the square

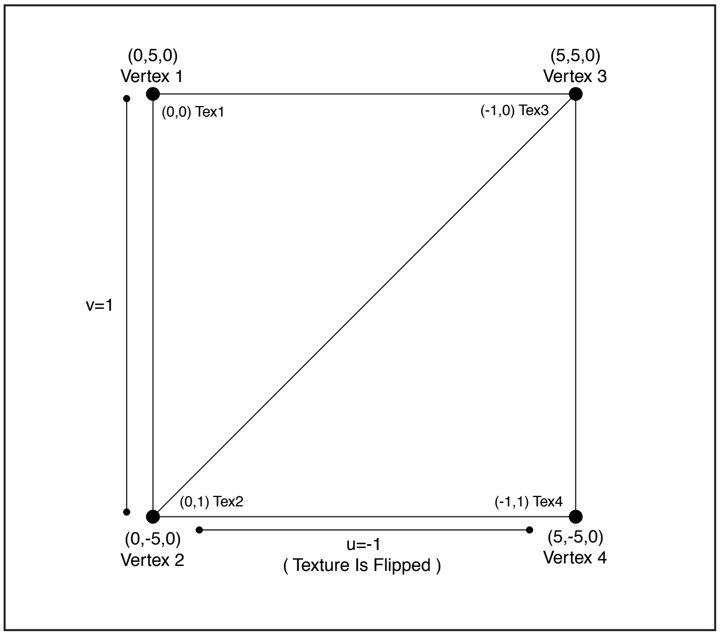


Figure 12.16

Texture coordinates within a texture

How to Load a Texture

The next block of code deals with creating the vertex buffer and loading it with the vertex information. This part has not changed since last time, so I'll skip ahead to loading a texture. Here is the code to do so.

```
if( FAILED( D3DXCreateTextureFromFile( pd3dDevice, "texture.jpg",
 &g_pTexture ) ) ) {
    return;
}
```

The D3DXCreateTextureFromFile() Function

The `D3DXCreateTextureFromFile()` function is provided within the Direct3D helper library. It is very useful because it can load from many different file formats. Some formats include .png, .jpg, .tga, and .bmp. This function ends up loading the texture into an `IDirect3DTexture8` interface. That is why you pass the `g_pTexture` variable to it. Once the function completes, it sets the pointer equal to the texture resource.

NOTE

The .jpg format is best for small texture files because of its lossy compression format, but the image quality suffers. The other formats use lossless compression; therefore, they have the best image quality. There is no clear winner format-wise, but my personal preference is to use .tga files.

Here is the prototype for the function.

```
HRESULT D3DXCreateTextureFromFileA(
    LPDIRECT3DDEVICE8 pDevice,
    LPCSTR pSrcFile,
    LPDIRECT3DTEXTURE8* ppTexture
);
```

The first parameter here expects a pointer to the `IDirect3DDevice8` interface. This one is easy because all you have to pass is the global `pd3dDevice` variable. The second parameter expects the filename to load. The extension on the filename determines how the function will load the particular file. The last parameter expects a pointer to hold the returned `IDirect3DTexture8` interface. For this, you should just pass in the `g_pTexture` variable that was declared at the beginning of the program.

If everything works out okay with the function, then it returns a `D3D_OK`. As long as that happens, you are free to move on. See, that wasn't so bad, was it?

The vRenderScene() Changes

Now that you have the geometry ready to go and the texture loaded, you can go on to display the scene. Here is the latest `vRenderScene()` function.

```
void vRenderScene(void)
{
    D3DXMATRIX     matTranslation;
    // Clear the backbuffer and the zbuffer
    pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
 D3DCOLOR_XRGB(0,0,255), 1.0f, 0 );
    // Begin rendering
    pd3dDevice->BeginScene();
    // Set default position, scale, and rotation
    D3DXMatrixIdentity( &matTranslation );
    pd3dDevice->SetTransform(D3DTS_WORLD, &matTranslation);
    // Set the texture to render
    pd3dDevice->SetTexture( 0, g_pTexture );
    // Set the vertex buffer to render
    pd3dDevice->SetStreamSource(0, lpVertexBuffer, sizeof(MYVERTEX));
    // Set the vertex format
    pd3dDevice->SetVertexShader(D3DFVF_MYVERTEX);
    // Rendering
    // Draw the primitive
    pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
    // Clear the reference
    pd3dDevice->SetTexture( 0, NULL );
    // Stop rendering
    pd3dDevice->EndScene();
    // Output the scene
    pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

I took the liberty of putting the relevant changes in bold text. Everything else is business as usual as far as rendering is concerned. To tell the 3D system which texture to use, you must use the SetTexture() function.

The IDirect3DDevice8::SetTexture() Function

The IDirect3DDevice8::SetTexture() function sets which texture is active for future rendering. Here is the prototype.

```
HRESULT SetTexture(
    DWORD Stage,
    IDirect3DBaseTexture8* pTexture
);
```

The first parameter wants you to specify which texture stage you want to assign to the texture. DirectX 8 supports up to eight textures, but you need to set this to 0 for now. The second parameter expects a pointer to the IDirect3DTexture8 interface you created earlier.

NOTE

There are multiple texture stages available. For basic purposes, you will only use texture stage zero. When you learn how to perform multi-texturing, you will use more than one stage.

The reason I set texture stage zero to `NULL` after I am done rendering is because I want to make sure the texture is turned off for next time.

CAUTION

Be sure to always set texture stages to `NULL` when you are done with them. This is a good practice to keep the reference counter for the texture at bay.

The vCleanup() Changes

Here you are, at the end of yet another sample program. The last thing to do before turning out the lights is clear the memory associated with the texture you loaded.

```
if( g_pTexture != NULL )
    g_pTexture->Release();
```

There is nothing special here, just your run-of-the-mill `Release()` function call. Once you accomplish that, you are done! Before you go, you might want to check out what the program looks like when run (see Figure 12.17).

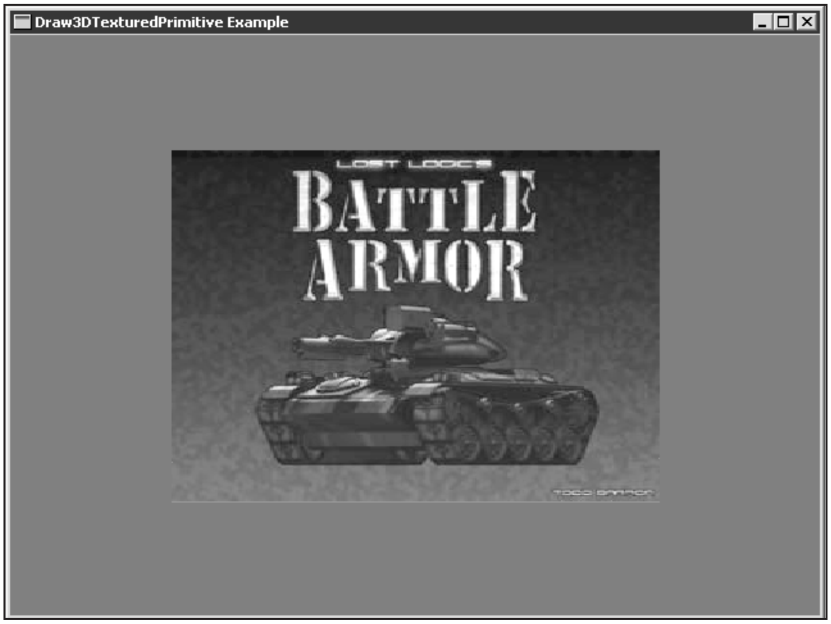


Figure 12.17

Output from the Draw3DTexturedPrimitive program.

Lights, Cameras, Action!

Well, two out of three isn't bad, right? In the 3DLight project on the CD, you will find the sample project to implement hardware lighting into your scenes. Go ahead and load it now to follow along with me. The point of this program is to show you how to create and add dynamic lighting to your 3D scenes. DirectX 8 lets you add up to eight hardware lights of any intensity and color. This lets you add some cool effects to your games. Keep in mind that lighting does slow down the system, so use it wisely.

The 3DLight project is just like the last one as far as code structure and project space go. The library files d3d8.lib and d3dx8.lib are included under the link section.

As with the last section, I am only going to highlight the changes made. So without further ado, move on to the header changes.

The D3DFVF_NORMAL Vertex Format

The first part of the program looks very similar to the texture example. The main difference is the inclusion of a new flexible vertex format flag.

```
// Definition for our vertex shader
#define D3DFVF_MYVERTEX (D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_TEX1)
// Vertex structure
typedef struct MYVERTEX
{
    D3DXVECTOR3 vecPos;
    D3DXVECTOR3 vecNorm;
    float          u;
    float          v;
} MYVERTEX, *LPMYVERTEX;
// Global vertices
MYVERTEX     g_Vertices[4];
```

Adding Normals to the Vertex Format

The relevant code changes in bold add normals to the vertex format. Actually, the vecPos variable is just a better way of storing the x, y, and z position of the vertex. In the previous examples, I stored the variables separately. I have now put them into a vector, which contains an x, y, and z coordinate. The relevant change is the inclusion of the D3DFVF_NORMAL format in the vertex definition. This tells the 3D system to expect normals information for each vertex.

Normals in a 3D object determine how lights and textures affect the faces of the object. If you do not include normals information, then textures and lighting do not appear properly. Figure 12.18 shows normals for the square object with which you have been working. As you can see, the normals point perpendicular to the surface. This ensures that light is reflected back properly. You don't really have to worry about normals too much because the 3D object loading routines you will

TIP

A D3DXVECTOR3 data structure that contains an x, y, and z coordinate is cleaner to use than three separate coordinate variables.

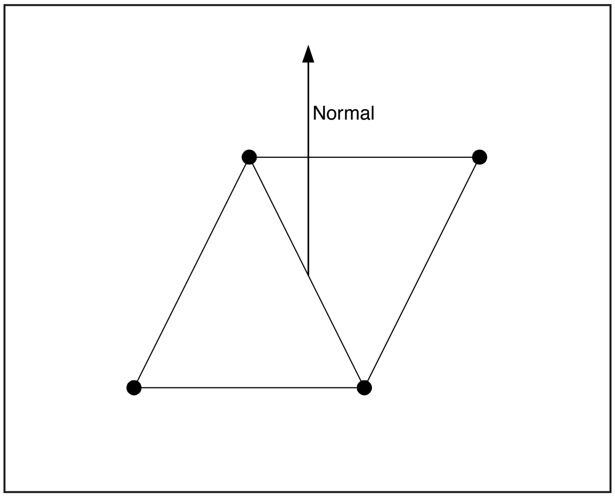


Figure 12.18

Vertex normals for a square

learn later will calculate them for you. In fact, most of the complicated vertex format stuff is automatic with .x objects.

Defining Normals in the vInitGeometry() Function

The program stays the same for the `hrInit3D()` function, and the next one down the line is the `vInitGeometry()` function. This is where you need to set up the vertex normals information. Here is the code.

```
void vInitGeometry(void) {
    void     *lpVertices;
    float     fSizeX = 5.0f;
    float     fSizeY = 5.0f;
    // Create the geometry
    g_Vertices[0].vecPos = D3DXVECTOR3(-fSizeX/2,fSizeY/2,0.0f);
    g_Vertices[0].u = 0.0f;
    g_Vertices[0].v = -1.0f;
    g_Vertices[0].vecNorm = D3DXVECTOR3(0.0f,0.0f,1.0f);
    g_Vertices[1].vecPos = D3DXVECTOR3(-fSizeX/2,-fSizeY/2,0.0f);
    g_Vertices[1].u = 0.0f;
    g_Vertices[1].v = 0.0f;      // Y
    g_Vertices[1].vecNorm = D3DXVECTOR3(0.0f,0.0f,1.0f);
    g_Vertices[2].vecPos = D3DXVECTOR3(fSizeX/2,fSizeY/2,0.0f);
    g_Vertices[2].u = -1.0f;      // X
    g_Vertices[2].v = -1.0f;
    g_Vertices[2].vecNorm = D3DXVECTOR3(0.0f,0.0f,1.0f);
    g_Vertices[3].vecPos = D3DXVECTOR3(fSizeX/2,-fSizeY/2,0.0f);
    g_Vertices[3].u = -1.0f;      // X
    g_Vertices[3].v = 0.0f;      // Y
    g_Vertices[3].vecNorm = D3DXVECTOR3(0.0f,0.0f,1.0f);
    // Create the vertex buffer
    if(FAILED(pd3dDevice->CreateVertexBuffer(4*sizeof(MYVERTEX), 0,
 D3DFVF_MYVERTEX, D3DPOOL_DEFAULT, &lpVertexBuffer))) {
        return;
    }
    // Lock the buffer
    if(FAILED(lpVertexBuffer->Lock(0, sizeof(g_Vertices),
     (BYTE**)&lpVertices, 0))) {
        return;
    }
    // Copy the g_Vertices information into the buffer
    memcpy(lpVertices, g_Vertices, sizeof(g_Vertices));
    // Unlock the buffer
    lpVertexBuffer->Unlock();
    // Load the texture
    if( FAILED( D3DXCreateTextureFromFile( pd3dDevice, "texture.jpg",
     &g_pTexture ) ) ) {
        return;
    }
}
```

The portion of this code that sets the vecPos field is just a simpler replacement of setting the x, y, and z coordinates explicitly. For example, take the following code.

```
g_Vertices[0].vecPos = D3DXVECTOR3(-fSizeX/2,fSizeY/2,0.0f);
```

In the previous programs, this looked like the following:

```
g_Vertices[0].x = -fSizeX/2;
g_Vertices[0].y = fSizeY/2;
g_Vertices[0].z = 0.0f;
```

All I am doing with the D3DXVECTOR3 structure is cleaning up the code from three lines of initialization to one. The real changes in this function are the vecNorm initializations.

For each normal, I set the x to 0.0, the y to 0.0, and the z to 1.0. What this does is make the faces point toward the camera. Because the camera is at (0.0,0.0,10.0), it is pointing back at the origin. This is somewhat hard to explain, so take a look at Figure 12.19. Notice how the normals of the faces point toward the camera. If the normals pointed the other way, then the polygons wouldn't even show up at render time. If you want to, change the values to (0.0,0.0,-1.0) and see how the polygons disappear.

Besides the normals initialization, the rest of the function is the same, so it's time to move on.

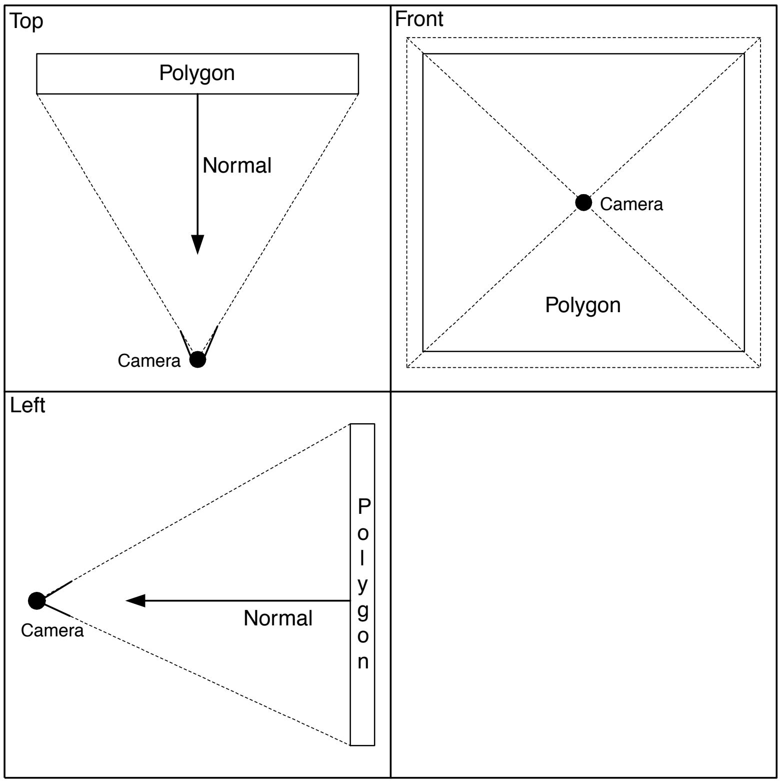


Figure 12.19
Normals facing the camera

How to Create Hardware Lighting

The next different function to come along is the `vCreateLight()` function. Previously, this function wasn't even present, so here it is in its entirety.

```
void vCreateLight(void)  {
    D3DMATERIAL8     mtrl;
    D3DLIGHT8     d3dLight;
    // Clear out material structure
    ZeroMemory( &mtrl, sizeof(D3DMATERIAL8) );
    mtrl.Diffuse.r = 1.0f;
    mtrl.Diffuse.g = 1.0f;
    mtrl.Diffuse.b = 1.0f;
    pd3dDevice->SetMaterial( &mtrl );
    // Clear out light structure
    ZeroMemory(&d3dLight, sizeof(D3DLIGHT8));
    d3dLight.Type = D3DLIGHT_POINT;
    d3dLight.Diffuse.r  = 1.0f;
    d3dLight.Diffuse.g  = 1.0f;
    d3dLight.Diffuse.b  = 1.0f;
    d3dLight.Position.x = 0.0f;
    d3dLight.Position.y = 5.0f;
    d3dLight.Position.z = 120.0f;
    d3dLight.Attenuation0 = 1.0f;
    d3dLight.Range       = 1000.0f;
    // Set the active light
    pd3dDevice->SetLight( 0, &d3dLight );
    // Enable the light
    pd3dDevice->LightEnable( 0, TRUE );
    // Tell D3D to utilize lights
    pd3dDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
}
```

The purpose of this function is pretty straightforward—it is to create a 3D light in your scene. However, the function actually accomplishes two things—it creates a default material and a 3D light.

Materials

Surfaces of objects in 3D games consist of *materials*, just as objects do in real life. Normally, a material denotes how rough a surface is or what color it is. Materials in the 3D game world have similar properties. The main use of materials in 3D gaming is to set how a surface reflects light. Because light reflection is what causes things to appear, it is quite important. Imagine having a material that reflected only yellow light—that material would appear yellow! Yes, I know, this all sounds very much like common sense, but you need to realize that materials in 3D gaming work much like their real-life counterparts. To define materials, you use the D3DMATERIAL8 data structure.

The D3DMATERIAL8 Data Structure

The D3DMATERIAL8 data structure holds any and all information about a material. Here is the structure.

```
typedef struct _D3DMATERIAL8 {
    D3DCOLORVALUE    Diffuse;
    D3DCOLORVALUE    Ambient;
    D3DCOLORVALUE    Specular;
    D3DCOLORVALUE    Emissive;
    float            Power;
} D3DMATERIAL8;
```

Material Diffusion

The diffuse value for a material sets how the material reflects light. The possible values range from 0.0 to 1.0 for each color component—red, green, and blue. If you set all of these values to 1.0, then the material reflects white light as normal. You can play around with changing the values to achieve differently colored materials. For example, if you only set the red component to 1.0, then the object only reflects red and white light. If you have a green light in the scene, then the red material will appear to be black.

Material Ambience

The ambient values work just like the diffuse value except the ambient color only takes into consideration the *ambient light* of the scene.

Ambient light is light that comes from everywhere in the scene. You will not see ambient light cast shadows; rather it just gives everything a minimum light level. It's sort of like turning up the brightness on your monitor. Most of the time, you will want to avoid setting ambient light above zero. The main reason is that objects look washed out when they have ambient lighting applied.

The diffuse color is used when other lights are in the scene, but the ambient color only reflects the ambient light in the scene. This value usually doesn't have much of an effect because most 3D games use very dim ambient settings.

Material Specularity

The specular value of a material sets how a material treats highlights. With no specularity, the material appears dull. With a high specularity, the material looks something like plastic or glass. The power value in the data structure sets how powerful the specularity color is. Some people call specularity *shininess.* Figure 12.20 shows the difference between a specular material and a non-specular one.

Material Emissiveness

You can use the emissive value of a material to make the material look as though it is emitting light. Other objects in the scene are not lit by it, but the material at least appears to be emitting the light value.

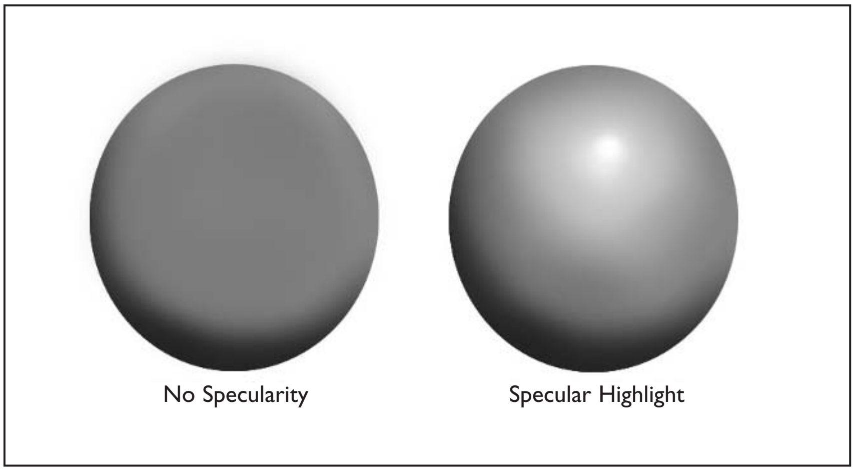


Figure 12.20

Specular highlight versus no specularity

Setting Up the Default Material

In the `vCreateLight()` function, you need to create the default material for the scene. This is accomplished by using the following block of code.

```
ZeroMemory( &mtrl, sizeof(D3DMATERIAL8) );
mtrl.Diffuse.r = 1.0f;
mtrl.Diffuse.g = 1.0f;
mtrl.Diffuse.b = 1.0f;
pd3dDevice->SetMaterial( &mtrl );
```

The first part of the code is straightforward. All you have to do is zero out the memory for the structure and then set the diffuse color for the material. You can set the other parameters if you so desire, but for this application's purpose, the diffuse is all you need.

The last part of the code is where the magic happens. To set the active material in a scene, you must make a call to the `IDirect3DDevice8::SetMaterial()` function. This activates the material you pass to the function.

The D3DLIGHT8 Data Structure

The next block of code deals with creating the hardware light for your scene. You use the `D3DLIGHT8` data structure to specify the properties of the light. Here is the layout for this data structure.

```
typedef struct _D3DLIGHT8 {
    D3DLIGHTTYPE     Type;
    D3DCOLORVALUE    Diffuse;
    D3DCOLORVALUE    Specular;
    D3DCOLORVALUE    Ambient;
    D3DVECTOR        Position;
    D3DVECTOR        Direction;
    float            Range;
    float            Falloff;
    float            Attenuation0;
    float            Attenuation1;
    float            Attenuation2;
    float            Theta;
    float            Phi;
} D3DLIGHT8;
```

Wow, would you look at that, another big data structure. Lucky for you, this program only has to deal with a subset of all the values. This demo creates a point light, so I'll show you the relevant parameters.

The first parameter expects the type of light you want to create. To create a point light, you must specify the light type to be D3DLIGHT_POINT. The second parameter expects the color of the light. This works just as materials do. To create a red light, you would use colors (1.0,0.0,0.0). The third parameter sets what the color specular highlights are when emitted from the light. The fourth parameter sets how much ambient light is emitted from the light. The fifth parameter sets up the position of the light.

The sixth parameter sets up a vector that defines the direction of the light. You only need to set this for directional lights and spotlights. The seventh parameter sets the range at which light no longer is emitted. This does not have an effect for directional lights. The eighth parameter sets the rate at which lights diminish when using spotlights. Most of the time, you will set this to 1.0. The next three parameters deal with how the light changes over distance. The last two parameters affect the cone of a spotlight.

Activating Hardware Lighting

Once the material and light data structures are set up, it's time to activate the light with the following calls.

```
// Set the active light
pd3dDevice->SetLight( 0, &d3dLight );
// Enable the light
pd3dDevice->LightEnable( 0, TRUE );
```

```
// Tell D3D to utilize lights
pd3dDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
```

The first function call, `SetLight()`, specifies the properties of a particular light. The next function call, `LightEnable()`, tells the 3D system to activate the set light. If you don't call this function, no light calculations are made. The third function call, `SetRenderState()`, tells the 3D system to perform lighting calculations. If you don't set this, no lighting will work.

Hey, guess what? It's that time again—time to run the program and see the output! The output for the lighting program appears in Figure 12.21.

If you are interested in playing around with the lighting, I suggest you try the following:

- Adjust the material for the scene to see how diffusion affects the polygons.
- Adjust the color of the light to see how it affects the material.
- Change the texture.jpg file to something other 256 × 256 graphics to see how lighting has an effect.
- Try all of the properties for lights and materials to get a feel for their uses.

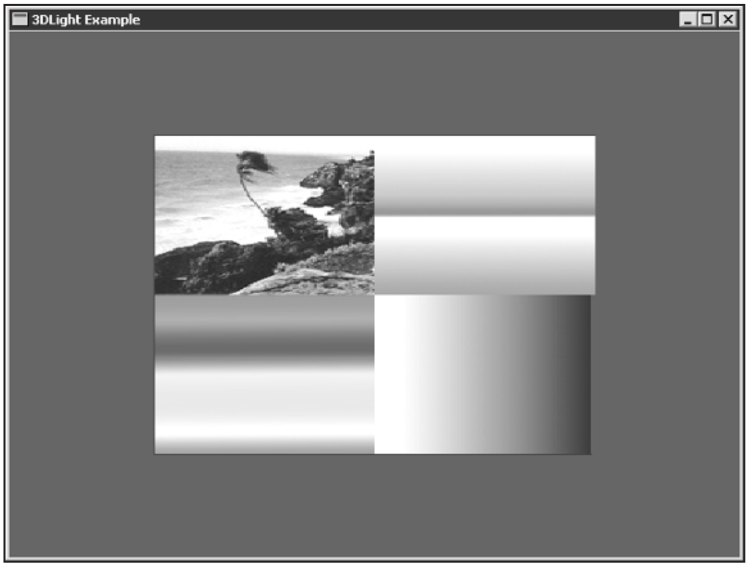


Figure 12.21

Output from the 3DLight project

Loading X-Files

Even though Fox Mulder is apparently out of the series, some of you might still want to load x-files. All right, I know; that was predictable. Anyway, if you don't already know, an *x-file* is a 3D-formatted data file created by a utility that comes with the DirectX SDK. A program called conv3ds.exe converts 3D-Studio 3D files to x-files. You can then use the utility functions within DirectX to load the files. In the XFileLoader project, I'll show you how to load and display an x-file. Go ahead now and load up the project, and you can get started.

The XFileLoader Project Space

The XFileLoader project space is much like the other demo programs in this chapter. The following libraries are needed for the link section.

- d3d8.lib
- d3dx8.lib
- mmsystem.lib

The mmsystem.lib file is needed by the dxutil.cpp helper code. You'll recall that you learned about the mmsystem.lib at the beginning of this book. Anyway, it has various multimedia calls; `timeGetTime()` is the one dxutil.cpp needs.

The next difference with this project is the inclusion of the dxutil.cpp and dxutil.h files. These files come with the DirectX SDK and provide various helper functions to make life easier. You can thank Microsoft for them; believe me, they deserve it.

The XFileLoader Header

There is no better time than the present to jump into the code, so here is the first part of code needed to load an x-file.

```
#define INITGUID
#define WIN32_LEAN_AND_MEAN     // No MFC
#include <d3d8.h>
#include <d3dx8.h>
#include <stdio.h>
#include "dxutil.h"
// Direct3D globals
```

```
LPDIRECT3D8                 pD3D;
LPDIRECT3DDEVICE8           pd3dDevice;
D3DXMATRIX                  matProj;                // Projection data
D3DXMATRIX                  matCameraView;          // Camera data
D3DXVECTOR3                 vecCameraSource;
D3DXVECTOR3                 vecCameraTarget;
LPD3DXMESH                  pMesh;
D3DMATERIAL8                *pMeshMaterials;
LPDIRECT3DTEXTURE8          *pMeshTextures;
DWORD                       dwNumMaterials;
LPD3DXBUFFER                pD3DXMtrlBuffer;
```

Most of this should look familiar to you by now. The new part comes in at the inclusion of the dxutil.h file. As I said a minute ago, the dxutil family has several useful functions in it.

The next bit of new information shows itself when you get to the LPD3DXMESH variable, which is a pointer to an ID3DXMesh interface. The ID3DXMesh interface is used for all sorts of 3D object handling. It is the main workhorse for x-files.

The D3DMATERIAL8 holds the data about the future x-file's materials. The LPDIRECT3DTEXTURE8 pointer holds data about the future x-file's textures. You use a DWORD next to store how many materials you have to deal with in the x-file. X-files can have multiple materials and textures that you have to render separately.

The last new variable is type LP3DXBUFFER and it is used as a generic buffer in which to store material information while you load it.

Loading the X-File

Once the window is created and the 3D environment is initialized, the program makes a call to the hLoadXFile() function. The purpose of this function is to load an x-file with the filename passed to it. When the function executes, it not only loads the geometry for the object, but it also loads the materials and textures for the x-file object. All of the data for the object is stored in the global variables I talked about a minute ago. Here is the source code for the function.

```
HRESULT hLoadXFile(char *szName)
{
    HRESULT          hResult = S_OK;
    int                      i;
    char                    szFullPath[256];

    dwNumMaterials = 0;
    // Load geometry
    if( FAILED( D3DXLoadMeshFromX(
        szName,
        D3DXMESH_SYSTEMMEM,
        pd3dDevice,
        NULL,
        &pD3DXMtrlBuffer,
        &dwNumMaterials,
        &pMesh ) ) ) {
        return(-1);
    }
    D3DXMATERIAL* d3dxMaterials =
     (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
    // Allocate memory for materials and textures
    pMeshMaterials = new D3DMATERIAL8[dwNumMaterials];
    pMeshTextures = new LPDIRECT3DTEXTURE8[dwNumMaterials];
    // Load in the materials and textures
    for( i = 0; i < (signed)dwNumMaterials ; i++ ) {
        // Copy the material
        pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
        // Set the ambient color for the material (D3DX does not do this)
        pMeshMaterials[i].Ambient = pMeshMaterials[i].Diffuse;
        sprintf(szFullPath,"3DData//%s",d3dxMaterials[i].pTextureFilename);
        // Create the texture
        if( FAILED( D3DXCreateTextureFromFile(
            pd3dDevice,
            szFullPath,
            &pMeshTextures[i] ) ) ) {
                delete pMeshTextures[i];
                pMeshTextures[i] = NULL;
        }
    }
    // Release the material buffer
    SAFE_RELEASE(pD3DXMtrlBuffer);
    return( hResult );
}
```

If you follow the code, you can see that the function performs the following operations.

1. Load the mesh.
2. Allocate memory for the materials.
3. Allocate memory for the textures.
4. Copy the materials.
5. Load the textures.
6. Release temporary buffers.
7. Return.

The D3DXLoadMeshFromX() Function

The `D3DXLoadMeshFromX()` function is provided as a helper function with DirectX 8. The sole purpose of this function is to load an x-file into memory. Here is the prototype.

```
HRESULT D3DXLoadMeshFromX(
    LPSTR pFilename,
    DWORD Options,
    LPDIRECT3DDEVICE8 pDevice,
    LPD3DXBUFFER* ppAdjacency,
    LPD3DXBUFFER* ppMaterials,
    PDWORD pNumMaterials,
    LPD3DXMESH* ppMesh
);
```

The first parameter specifies the filename of the x-file you want to load. The second parameter lets you specify various options. The one you will use most of the time is the `D3DXMESH_SYSTEMMEM` flag. This flag tells the function to store the memory for the vertices and the indices in system memory.

The third parameter expects the `IDirect3DDevice8` interface you want to use with the loaded mesh. The fourth parameter lets you specify a pointer to an `ID3DXBuffer` interface. If you specify a pointer, the buffer is filled with information that specifies the neighbors of each face in the mesh. I normally set this parameter to `NULL`.

The fifth parameter retrieves the materials used in the x-file. The materials loaded are put into the `D3DXMATERIAL` buffer passed here. The sixth parameter lets you know how many materials were passed back in the fifth parameter. The last parameter receives the actual mesh that gets loaded. The type used here is an `ID3DXMesh` pointer.

Loading and Creating the Materials

Once the code executes that calls this function, the pD3DXMtrlBuffer buffer holds all relevant material information in a raw format. The raw data cannot be used as is. You need to convert the raw data to actual material data for use with the mesh. This is where the next line of code comes into play.

```
D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
```

This line of code creates a new data structure called d3dxMaterials and sets it equal to the temporary material buffer. To do this, it points to the GetBufferPointer() member of the material buffer.

The next couple of lines of code deal with creating memory for both the materials and the textures.

```
pMeshMaterials = new D3DMATERIAL8[dwNumMaterials];
pMeshTextures = new LPDIRECT3DTEXTURE8[dwNumMaterials];
```

Because dwNumMaterials holds the number of materials loaded, you can just allocate memory for that number. There is no real way to know exactly how many of the materials have textures, but you can go ahead and allocate memory under the assumption that each material has one. If you run across a material that does not have a texture, you can just delete that particular texture and move on.

The loop that follows loops through each material loaded and assigns it to the newly created pMeshMaterials buffer. For each material you come across, you should try to load a corresponding texture. If the texture is not found, then go ahead and delete the texture for the material.

Once all of the textures and materials are loaded, the code cleans up the temporary material buffer using the SAFE_RELEASE macro. The SAFE_RELEASE macro, which is part of the dxutil.cpp utility set, is useful for deleting interfaces that have the Release() functionality.

Now that the function has completed, the pMesh object is ready to be rendered.

Rendering the X-File

Once the mesh is loaded, the program goes on to set up the view in the normal fashion. The only difference here is that the camera has been moved to a more fitting position in which to view the x-file object. To get a feel for where the camera is located, take a look at Figure 12.22.

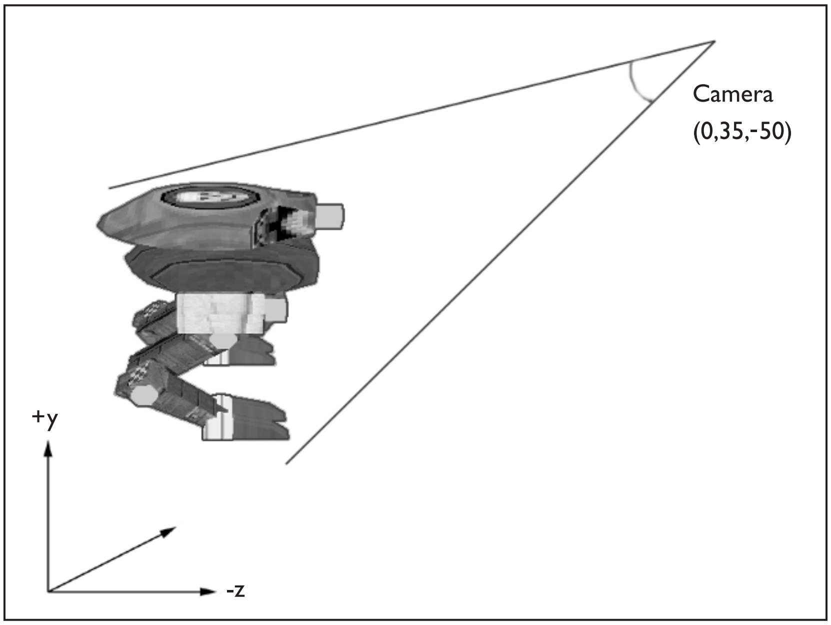


Figure 12.22

The camera, positioned to view the x-file

Once the setup view function returns, the system executes the rendering function. Most of what you see in the render function is the same as in previous programs. The first minor difference is that this program uses a black background. You can tell this from the `D3DCOLOR_XRGB` values in the `Clear()` function. The main part you should note is the rendering of the actual x-file, which happens within a fairly simple loop.

```
// Render the objects
for( mat = 0 ; mat < (signed)dwNumMaterials ; mat++ ) {
      // Activate the material
      pd3dDevice->SetMaterial( &pMeshMaterials[mat] );
      // Set the appropriate texture
      pd3dDevice->SetTexture( 0, pMeshTextures[mat] );
      // Draw the mesh subset
      pMesh->DrawSubset( mat );
}
```

Recall when the mesh was loaded that the `dwNumMaterials` variable holds how many materials were loaded. This is very important to the program because you have to render each material separately. This is why the code uses a loop to go through each material within the x-file object.

The DrawSubset() Function

Basically, the loop activates the next material in the list, activates the texture, and then renders the material with the `DrawSubset()` function. I wish I could make it more complicated than that, but that's really all there is to it! Once you render each material, you're done with the x-object. If you run the program and everything works out as planned, then you should see something similar to Figure 12.23.

The Living World

So far, the 3D examples have been pretty stagnate. Nothing has moved around; everything has been lifeless and rather boring. Well, all of that changes with the XFileAnimator project. Go ahead and load it up so you can start learning how to move 3D objects. In this program, you load the robot mesh from the last project. Once the mesh is loaded, you render the robot to the screen. The new part about this program is that you can use the left and right arrow keys to rotate the robot about its y-axis.

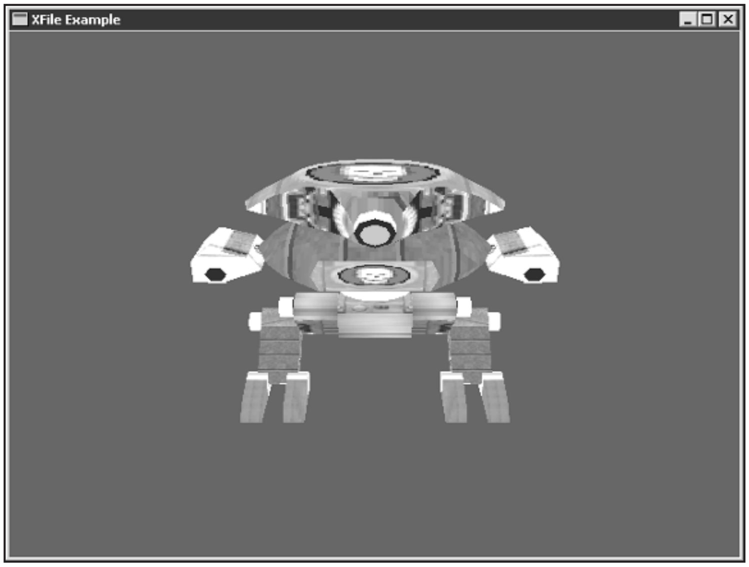


Figure 12.23
Output from the XFileLoader program

Radians or Die!

One of the more obscure things about 3D animation is the formula for rotating objects. Most people would think that you just tell the system how many degrees you want something to turn and it does it. This is not the case for 3D animation systems. Turns are indicated in units called *radians*. To find the number of radians in a certain degree, you must use the following formula.

```
(degrees/180.0f)* 3.1415926535897932384626433832795
```

I don't know about you, but I'm not the biggest mathematics fan in the world, and that formula looks a bit intimidating. It's really not that bad once you stick it into a macro that you can use over and over again. This is where the following piece of code helps. It is declared at the beginning of the program and is used later for calculating radians from degrees.

```
#define PI 3.1415926535897932384626433832795
#define DegToRad(deg) ((deg/180.0f)*PI)
```

From now on, all you have to do is use the `DegToRad()` function when you want to know how many radians there are in a particular degree.

The Dynamic Message Loop

In previous programs, the rendering function was called only once because there was nothing to animate. Now that you want to rotate the x-file object, you need to re-render the scene each time it moves. This is why the message loop now incorporates the rendering function with each cycle.

```
// Process messages until the program is terminated
ZeroMemory( &msg, sizeof(msg) );
while( msg.message!=WM_QUIT ) {
    if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) ) {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    else {
        vRenderScene();
    }
}
```

Each time the message loop checks for a message and no message is found, the `vRenderScene()` function is called.

Reading the Arrow Keys

Another new addition to this program is keyboard checking. This program actually needs to check the keyboard for user input, which is all handled at the window's messaging level.

```
case WM_KEYDOWN:
    // Handle any non-accelerated key commands
    switch (wParam)
    {
    case VK_ESCAPE:
        break;
    // Left arrow key
    case 37:
        fRotation += 2.0f;
        break;
    // Right arrow key
    case 39:
        fRotation -= 2.0f;
        break;
    default:
        break;
    }
    break;
```

If you look closely, you can see that key codes 37 and 39 relate to the arrow keys on the keyboard. A global floating-point variable called `fRotation` keeps track of which way the robot is rotated in this program. Whenever the user presses the left-arrow key, the robot rotates two degrees clockwise. Whenever the user presses the right-arrow key, the robot rotates two degrees counterclockwise.

So there you have it, how to rotate the robot! Oh, wait, I forgot to show you how to render the little rascal.

The Rotation Matrix

If you go ahead and hop on down to the `vRenderScene()` function, you get to see how you go about rotating the robot. Here is the code.

```
void vRenderScene(void)
{
    D3DXMATRIX      matRotation;
    int                         mat;

    // Clear the backbuffer and the zbuffer
    pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
D3DCOLOR_XRGB(0,0,0), 1.0f, 0 );
    // Begin rendering
    pd3dDevice->BeginScene();
    // Set the rotation
    D3DXMatrixRotationY( &matRotation, (float)DegToRad(fRotation) );
    pd3dDevice->SetTransform(D3DTS_WORLD, &matRotation);
    // Render the objects
    for( mat = 0 ; mat < (signed)dwNumMaterials ; mat++ ) {
        // Activate the material
        pd3dDevice->SetMaterial( &pMeshMaterials[mat] );
        // Set the appropriate texture
        pd3dDevice->SetTexture( 0, pMeshTextures[mat] );
        // Draw the mesh subset
        pMesh->DrawSubset( mat );
    }
    // Stop rendering
    pd3dDevice->EndScene();
    // Output the scene
    pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

The important part to rotating the 3D object appears in the bold text. To set up a rotation matrix for the robot's y-axis, you need to use the D3DXMatrixRotationY() helper function.

The D3DXMatrixRotationY() Function

The D3DXMatrixRotationY() function creates a default rotation matrix with a given radian value. The matrix returned is filled with the proper data to rotate the object as many radians as requested. For this function, you pass the DegToRad() value of the fRotation global variable. This tells the matrix function to return to you a rotation matrix that contains the proper values for rotating the robot.

The SetTransform() function then activates the rotation matrix created. Once this is done, any object rendered is rotated about the y-axis.

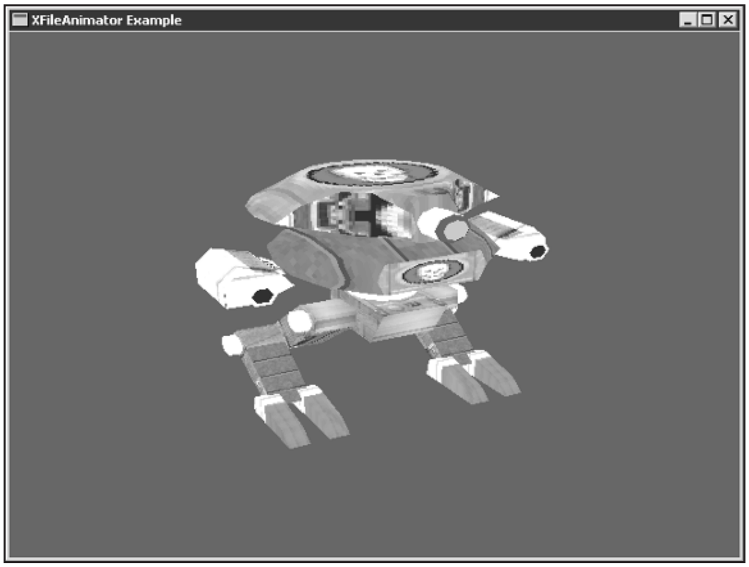


Figure 12.24

Output from the XFileAnimator program

If you want to play around with this functionality, replace the `D3DXMatrixRotationY()` function with the `D3DXMatrixRotationX()` or `D3DXMatrixRotationZ()` function. By doing this, you change the axis on which the robot rotates. Output for this program appears in Figure 12.24.

Review

Now that you have finally learned how to render something other than a blank screen, it's time to review. Take the following key points with you.

- All 3D objects are made up of polygons. Each polygon is made up of three points. Each of the points has its own x, y, and z coordinate on the Cartesian plane.
- Of the various primitive types, most 3D games use triangle lists and triangle strips for rendering.
- Normals are used to determine how the surface of a 3D object reflects light. If your normals are not set up properly, your object will look strange when rendered.

- Translation matrices determine the position of geometry in your scenes. Rotation matrices determine the orientation of the geometry, and scale matrices determine the scale (or size) of the geometry. You use the `SetTransform()` function to activate the current matrix.
- All rotation values in Direct3D are used in radians and not degrees.

Hey! Guess what! It's that time again—time for another chapter!

CHAPTER 13

Now Showing, DirectInput and DirectAudio!

You are almost there; the taste of battle is overpowering. All throughout the land, everything is still and lifeless. Neither a bird sings nor a cannon fires. You cannot move for you are frozen, a sitting target. What's that on the horizon? Why, it's DirectInput and DirectAudio to the rescue!

Dorky introductions aside, this chapter is all about audio and input. By learning DirectInput, you will learn how to add responsive controls to your games. The standard Windows keyboard controls are clunky and unresponsive, and don't even get me started about joystick support. By the same token, DirectSound is much better than using the standard Windows calls to play a mere one sound per channel. With DirectAudio, you can add multichannel playback to your games for a truly immersive experience. In this chapter, you will learn how to

- Initialize DirectInput
- Create a keyboard device
- Read buffered keyboard input
- Initialize DirectAudio
- Load a wave file
- Play a wave file

Let's get started!

Using a Keyboard with DirectInput

DirectX 8.0 has made initializing DirectInput quite simple. The heart of using the DirectInput system is the `IDirectInput8` interface.

The IDirectInput8 Interface

The `IDirectInput8` interface is what you use to create and access devices for input. The main purpose of DirectInput is to use keyboard, joystick, or unique devices. You do not gain very much by using DirectInput to control the mouse. For mouse controls, you should stick with the standard Windows methods.

To create an `IDirectInput8` interface, you use the `DirectInput8Create()` function. There is no better way to show you how to do something than with source code, so go ahead and load up the DirectInputKeyboard project found in the Chapter13 Zip file on the book's CD. In this project, you will learn how to initialize DirectInput and create a keyboard device.

The DirectInputKeyboard Project

Instead of mixing graphics and other stuff with the project, this project only initializes DirectInput and creates a keyboard device. For one thing, I believe this will help you see how to do the task at hand instead of having you spend a lot of time wading through source code.

To use DirectInput, you must include the dinput8.lib and dxguid.lib libraries in your project. The only header file you need for DirectInput is the dinput.h header file. Here is the source code.

```
#include <stdio.h>
#include <dinput.h>
#define INPUTERROR_NODI                  0x81000001
#define INPUTERROR_NOKEYBOARD            0x81000002
LPDIRECTINPUT8           pDI = NULL;
LPDIRECTINPUTDEVICE8     pKeyboard = NULL;

// Windows globals
const char          *szWndClass        = "DirectInputKeyboard";
const char          *szProgramName     = "Direct-Input Keyboard Example";
HINSTANCE       g_hInstance;
// Function prototypes
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow);
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam,
 LPARAM lParam );
void vCleanup(void);
HRESULT     hrInitDirectInput(void);
HRESULT     hrInitKeyboard(HWND hWnd);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
```

```
WNDCLASS      wc;
HWND          hWnd;
MSG           msg;
HRESULT       hr;

// Set up and register window class
wc.style = CS_HREDRAW | CS_VREDRAW;
wc.lpfnWndProc = MsgProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInstance;
wc.hIcon = LoadIcon( NULL, IDI_APPLICATION );
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH )GetStockObject(BLACK_BRUSH);
wc.lpszMenuName = NULL;
wc.lpszClassName = szWndClass;
RegisterClass(&wc);
// Create a window
hWnd = CreateWindowEx(WS_EX_TOPMOST,
    szWndClass, szProgramName, WS_OVERLAPPEDWINDOW,
    0, 0, 640, 480, NULL, NULL, hInstance, NULL);
// Set up global window instance
g_hInstance = hInstance;
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
hr = hrInitDirectInput();
if( hr == INPUTERROR_NODI ) {
    MessageBox( hWnd, "DirectInput Error", "Unable to initialize
     DirectInput.", MB_ICONERROR );
    vCleanup();
    exit(1);
}
hr = hrInitKeyboard(hWnd);
if( hr == INPUTERROR_NOKEYBOARD ) {
    MessageBox( hWnd, "DirectInput Error", "Unable to initialize
     Keyboard.", MB_ICONERROR );
    vCleanup();
    exit(1);
}
// Process messages until the program is terminated
ZeroMemory( &msg, sizeof(msg) );
```

```
    while( msg.message!=WM_QUIT ) {
        if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) ) {
            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }
        else {
        }
    }
    // Clean up the environment
    vCleanup();
    return(msg.wParam);
};
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
    case WM_DESTROY:
        PostQuitMessage( 0 );
        return 0;
    default:
        break;
    }
    return DefWindowProc( hWnd, msg, wParam, lParam );
}
void vCleanup(void)
{
    if( pKeyboard ) {
        pKeyboard->Unacquire();
        pKeyboard->Release();
    }
    if( pDI ) {
        pDI->Release();
    }
    UnregisterClass( szWndClass, g_hInstance );
}
HRESULT     hrInitDirectInput(void)
{
    HRESULT     hReturn = 0;

    // Do not try to create DirectInput if already created
    if( !pDI ) {
        // Create a DInput object
        if( FAILED( hReturn = DirectInput8Create( GetModuleHandle(NULL),
         DIRECTINPUT_VERSION,
            IID_IDirectInput8, (VOID**)&pDI, NULL ) ) )
            return(INPUTERROR_NODI);
    }
    return(hReturn);
}
HRESULT hrInitKeyboard(HWND hWnd)
{
    HRESULT          hReturn = 0;
    DIPROPDWORD          dipdw;

    // Obtain an interface to the system keyboard device
    if( FAILED( hReturn = pDI->CreateDevice( GUID_SysKeyboard,
     &pKeyboard, NULL ) ) )
        return(INPUTERROR_NOKEYBOARD);

    // Create buffer to hold keyboard data
    ZeroMemory(&dipdw,sizeof(DIPROPDWORD));
    dipdw.diph.dwSize             = sizeof(DIPROPDWORD);
    dipdw.diph.dwHeaderSize       = sizeof(DIPROPHEADER);
    dipdw.diph.dwObj              = 0;
    dipdw.diph.dwHow              = DIPH_DEVICE;
    dipdw.dwData                  = 10;
    // Set the size of the buffer
    if( FAILED( hReturn = pKeyboard->SetProperty( DIPROP_BUFFERSIZE,
     &dipdw.diph ) ) )
        return(INPUTERROR_NOKEYBOARD);
    // Set the format of the keyboard
    if( FAILED( hReturn = pKeyboard->SetDataFormat( &c_dfDIKeyboard ) ) )
        return(INPUTERROR_NOKEYBOARD);
    // Set the cooperative level to exclusive access
    if( FAILED( hReturn = pKeyboard->SetCooperativeLevel( hWnd,
     DISCL_EXCLUSIVE | DISCL_FOREGROUND ) ) ) {
        return(INPUTERROR_NOKEYBOARD);
    }
    // Acquire the keyboard device
    pKeyboard->Acquire();
    return(0);
}
```

The Keyboard Program Layout

The keyboard project breaks down into five main functions.

- `WinMain()`
- `MsgProc()`
- `vCleanup()`
- `hrInitDirectInput()`
- `hrInitKeyboard()`

The first three functions, `WinMain()`, `MsgProc()`, and `vCleanup()`, should all look familiar to you by now. The two new ones are the `hrInitDirectInput()` and `hrInitKeyboard()` functions. To see how the project is laid out, check out Figure 13.1.

As you see in Figure 13.1, the program executes the `WinMain()` function, which then calls the `hrInitDirectInput()` function. If this function succeeds, then the `hrInitKeyboard()` function is called. Nothing is actually displayed on the screen if everything works out okay. The only time you get feedback in this program is if any of the operations fail.

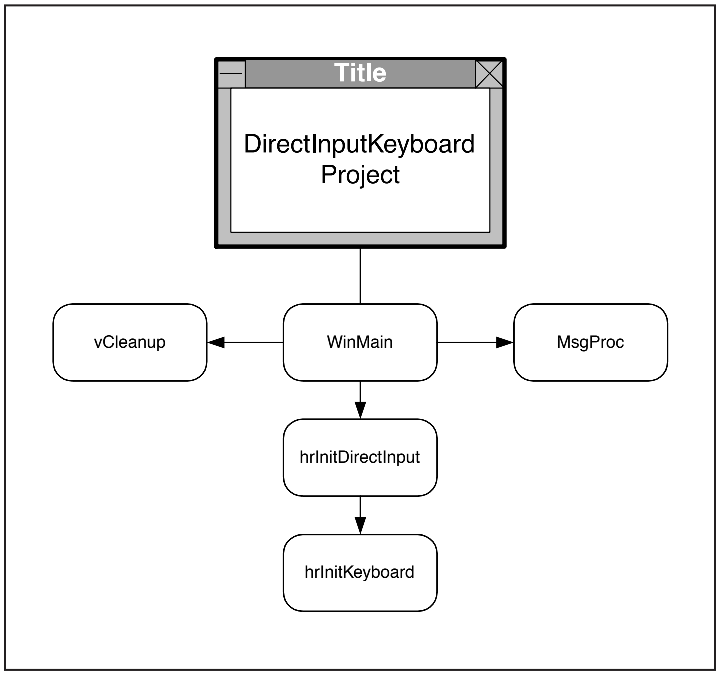


Figure 13.1

The DirectInputKeyboard project

The hrInitDirectInput() Function

The first fun function to come along is the `hrInitDirectInput()` function. This function is actually quite short because DirectX 8.0 has made creating input interfaces very easy. The one and only purpose of this function is to create an `IDirectInput8` interface. This happens using the aforementioned `DirectInput8Create()` function call.

The DirectInput8Create() Function

The `DirectInput8Create()` function is similar to the other DirectX create functions in that it retrieves an interface. The parameters for the function are as follows.

```
HRESULT WINAPI DirectInput8Create(
    HINSTANCE hinst,
    DWORD dwVersion,
    REFIID riidltf,
    LPVOID* ppvOut,
    LPUNKNOWN punkOuter
);
```

The first parameter is a little different in that it expects the instance of the application or DLL that is creating the DirectInput interface. For this, you can pass in the global `HINSTANCE` stored at the beginning of the program.

The second parameter lets you specify the version of DirectX you are using. This will ensure that you are using the proper header file. For this, you should pass in the `DIRECTINPUT_VERSION` definition.

The third parameter specifies which version of the interface you want to use, either ASCII or Unicode. The flags available for the parameter can be seen in Table 13.1.

The fourth parameter receives the created interface. The last parameter lets you specify a pointer for aggregation. Normally, you just pass a `NULL` here.

Table 13.1 DirectInput8Create() Interface Flags

Flag	Description
`IID_IDirectInput8`	Selects ASCII or Unicode based on the #`UNICODE` define
`IID_IDirectInput8A`	Selects the ASCII interface
`IID_IDirectInput8W`	Selects the Unicode interface

As for the DirectInputKeyboard program, it implements the interface creation like so:

```
HRESULT          hrInitDirectInput(void) {
    HRESULT     hReturn = 0;

    // Do not try to create DirectInput if already created
    if( !pDI ) {
        // Create a DInput object
        if( FAILED( hReturn = DirectInput8Create( g_hInstance,
         DIRECTINPUT_VERSION,
            IID_IDirectInput8, (VOID**)&pDI, NULL ) ) )
            return(INPUTERROR_NODI);
    }
    return(hReturn);
}
```

The first thing you do in the `hrInitDirectInput()` function is check whether the DirectInput interface has already been created. If it has, then you should just leave the function. If the interface has not been created, you go on to call the `DirectInput8Create()` function. As you can see from the code, you are passing the global `g_hInstance` variable for the current instance of the application. Next comes the `DIRECTINPUT_VERSION` definition. For the interface type, you pass `IID_IDirectInput8`, which tells the system to choose either ASCII or Unicode depending on the current settings for the project. For the next and most important parameter, you pass the global `pDI` variable. Finally, you need to pass `NULL` for the last parameter.

Okay, once you've created the DirectInput interface, you are ready to create some devices. Remember, devices are things such as keyboards, joysticks, and mice.

The hrInitKeyboard() Function

The next function to come along is the `hrInitKeyboard()` function. This is where you create the actual keyboard device. To see what you have accomplished so far, take a look at Figure 13.2.

You must accomplish five things before the keyboard device is ready to go.

1. Create the device.
2. Set the buffer size.
3. Set the format.
4. Set the cooperative level.
5. Acquire the device.

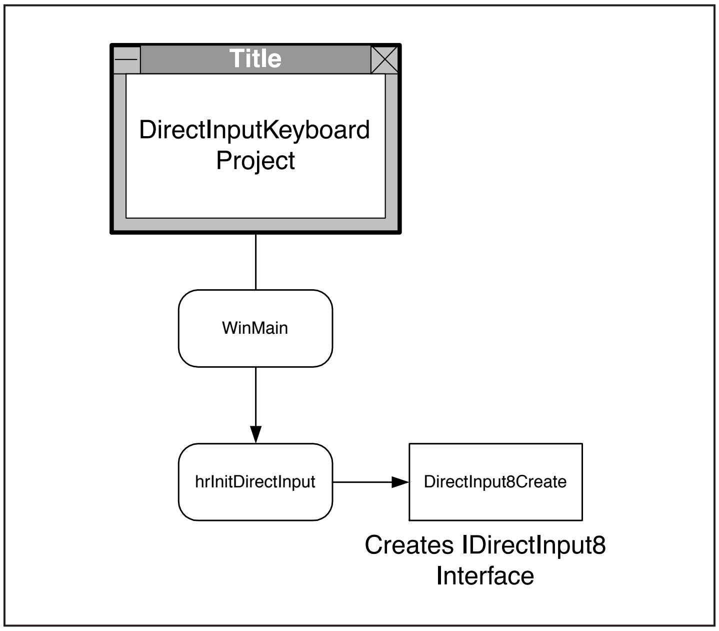


Figure 13.2

Processes up to creating the device

The IDirectInput8::CreateDevice() Function

The first thing you need to do is create the actual keyboard device using the `CreateDevice()` member function of the `IDirectInput8` interface you created earlier. Here are the parameters for the function.

```
HRESULT CreateDevice(
    REFGUID rguid,
    LPDIRECTINPUTDEVICE *lplpDirectInputDevice,
    LPUNKNOWN pUnkOuter
);
```

The first parameter specifies the GUID of the device you are trying to create. If you don't know which devices are available, you can use the `IDirectInput8::EnumDevices()` function. However, you do know that there is a keyboard attached to the system, so you can go ahead and specify the GUID of `GUID_SysKeyboard` here.

The second parameter receives the `IDirectInputDevice8` device pointer you are creating. The last parameter is used for aggregation. Normally, you just set this to `NULL`.

As you can see in the code for the keyboard project, you are using all of the values I suggested. The main item of interest here is the `GUID_SysKeyboard` value being passed for the device's GUID. This tells the system to give you the keyboard device.

```
// Obtain an interface to the system keyboard device
if( FAILED( hReturn = pDI->CreateDevice( GUID_SysKeyboard, &pKeyboard, NULL ) ) )
    return(INPUTERROR_NOKEYBOARD);
```

Now that you have the keyboard device, you can go ahead and set up the buffer to receive input from the user.

Setting Up the Keyboard Buffer

To set up a buffer for devices, you must initialize a `DIPROPDWORD` data structure. The code used in the current project works like the following:

```
// Create buffer to hold keyboard data
ZeroMemory(&dipdw,sizeof(DIPROPDWORD));
dipdw.diph.dwSize                = sizeof(DIPROPDWORD);
dipdw.diph.dwHeaderSize          = sizeof(DIPROPHEADER);
dipdw.diph.dwObj                 = 0;
dipdw.diph.dwHow                 = DIPH_DEVICE;
dipdw.dwData                     = 10;     // Buffer Size
```

Buffered Versus Immediate Data

For all DirectInput devices, you can use either buffered or immediate data. When you retrieve immediate data, you retrieve the absolute current state of the device at the time of the call. This works fine for devices that only need to worry about their current state. For all other devices, you can use buffered data. When using buffered data, the system records input from the user and stores it in a specified buffer. The application can then go and read data from the buffer. When data is read, it is removed from the buffer. If the application overflows the buffer, new data is lost.

In most cases, you will want to use buffered data. The main reason is that with immediate data, input is only as accurate as the frame rate of the game. If it takes 30ms to render a frame, the user can only input data at a rate of 30ms per entry. This can cause serious problems with keyboard input. Imagine how you would feel if half of your keyboard presses were lost while you were playing a game—this is what can happen if you don't use buffered data.

The main part of setting up the data structure comes in setting up the parameters for the diph member structure of type DIPROPHEADER. The DIPROPHEADER data structure sets up the properties of a device.

All you really need to notice here is the dwData member of the DIPROPDWORD data structure. This tells DirectInput how many elements to store in the keyboard's input buffer. If you set this to 0, then nothing is buffered. If you set the value too low, keyboard data will overflow. Normally, a value ranging from 10 to 20 is sufficient. To see how a program should operate when using buffered data, check out Figure 13.3.

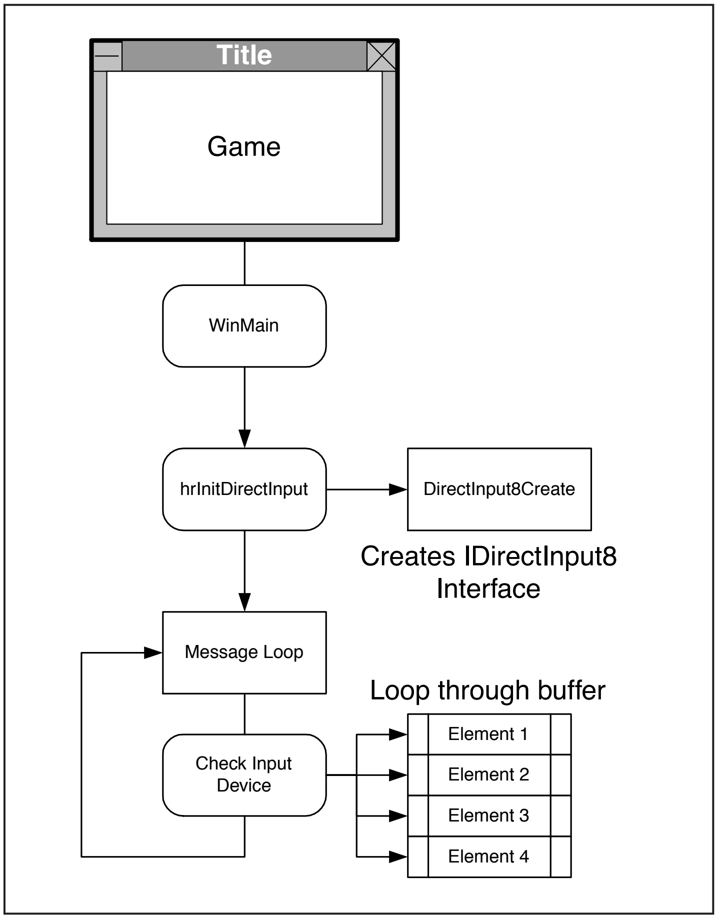


Figure 13.3

Buffered device data processing

The IDirectInputDevice8::SetProperty() Function

The `IDirectInputDevice8::SetProperty()` function is what you call to tell the keyboard device to use buffered data. So now that you have the buffer data structure initialized, you can go ahead and set the keyboard to be buffered. Here are the parameters for this function.

```
HRESULT SetProperty(
    REFGUID rguidProp,
    LPCDIPROPHEADER pdiph
);
```

The first parameter specifies the GUID of the property you are trying to set. For setting the buffer property, you use the `DIPROP_BUFFERSIZE` GUID. The last parameter passes in the address of the `DIPROPHEADER` data structure that defines the property.

Once the property is set, it is time to set up the keyboard format.

```
// Set the size of the buffer
if( FAILED( hReturn = pKeyboard->SetProperty( DIPROP_BUFFERSIZE,
 &dipdw.diph ) ) )
    return(INPUTERROR_NOKEYBOARD);
// Set the format of the keyboard
if( FAILED( hReturn = pKeyboard->SetDataFormat( &c_dfDIKeyboard ) ) )
    return(INPUTERROR_NOKEYBOARD);
```

The IDirectInputDevice8::SetDataFormat() Function

The `SetDataFormat()` function sets the format of the device you created. Here is the prototype.

```
HRESULT SetDataFormat(
    LPCDIDATAFORMAT lpdf
);
```

The one and only parameter for this function specifies the format of the device in the form of a `DIDATAFORMAT` data structure. If you want to, you can use one of the predefined types.

- c_dfDIKeyboard
- c_dfDIMouse
- c_dfDIMouse2
- c_dfDIJoystick
- c_dfDIJoystick2

The IDirectInputDevice8::SetCooperativeLevel() Function

Once you create the keyboard device, set up its buffer, and set the data format, it is time to set the cooperative level of the device using the `SetCooperativeLevel()` function.

The *cooperative level* of a device tells the instance of the device how to operate in regard to other instances of the device. Because multiple applications can use the keyboard at once, you need to set the cooperative level. Here is the prototype for the function.

```
HRESULT SetCooperativeLevel(
    HWND hwnd,
    DWORD dwFlags
);
```

The first parameter specifies the handle of the window that will use the device. Make sure you specify a top-level window here. The last parameter lets you specify certain cooperative level flags. You can use the following flags.

- `DISCL_BACKGROUND`
- `DISCL_EXCLUSIVE`
- `DISCL_FOREGROUND`
- `DISCL_NONEXCLUSIVE`
- `DISCL_NOWINKEY`

The main thing to remember when setting a device's cooperative level is that you must use either the `DISCL_FOREGROUND` or the `DISCL_BACKGROUND` value for one flag. You also must use either the `DISCL_EXCLUSIVE` or the `DISCL_NONEXCLUSIVE` value for the other flag. You only use `DISCL_NOWINKEY` to tell the device not to accept the Windows key.

When you set a device flag to use the `DISCL_FOREGROUND` value, you tell the device to become inactive when its window is sent to the background. This is what applications normally do because you don't want input from the user going to your application when the application is minimized. If you follow along with the code for the project, you see that it sets the cooperative level to both exclusive mode and foreground mode.

```
// Set the cooperative level to exclusive access
if( FAILED( hReturn = pKeyboard->SetCooperativeLevel( hWnd,
 DISCL_EXCLUSIVE | DISCL_FOREGROUND ) ) ) {
    return(INPUTERROR_NOKEYBOARD);
}
```

The IDirectInputDevice8::Acquire() Function

The final step to setting up a DirectInput device is acquiring it with the duly-named `Acquire()` function. This function does not take any parameters, so it's pretty easy to use. If the function succeeds, it passes back a `DI_OK` value.

```
// Acquire the keyboard device
pKeyboard->Acquire();
```

Well, now you have created a keyboard device, and everything should be just dandy. Yeah, I know, dumb word. Anyway, the only thing left to do is show you how to clean up the mess that was made—using our old favorite, the `vCleanup()` function.

The vCleanup() Function

When the application exits, the `vCleanup()` function needs to accomplish a few things.

1. Unacquire the keyboard.
2. Release the keyboard.
3. Release DirectInput.

If you look at the code, you'll see how this happens.

```
void vCleanup(void)  {
    if( pKeyboard ) {
        pKeyboard->Unacquire();
        pKeyboard->Release();
    }
    if( pDI ) {
        pDI->Release();
    }
    UnregisterClass( szWndClass, g_hInstance );
}
```

The IDirectInputDevice8::Unacquire() Function

The `IDirectInputDevice8::Unacquire()` function is the counterpart to the `Acquire()` function. This function takes no parameters and returns `DI_OK` upon success. You use this function to free the acquired device.

You have already seen the `Release()` function used in many places, so I won't cover that one again. Just keep in mind that you need to release both devices and interfaces for DirectInput. To see how the initialization and cleanup operations relate to each other, look at Figure 13.4.

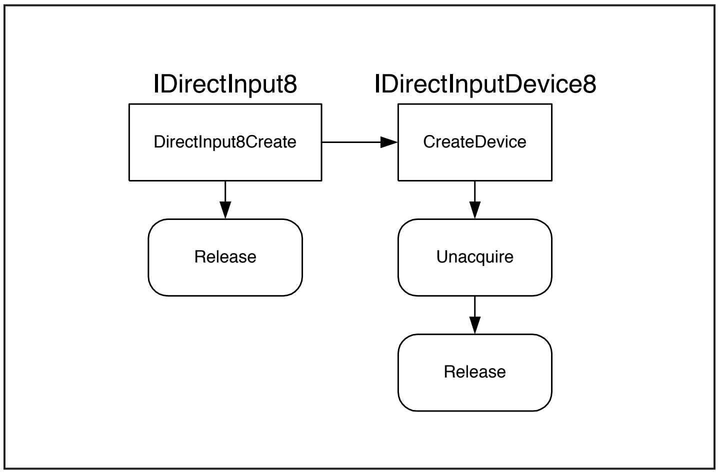


Figure 13.4

Initialization and cleanup of DirectInput objects

All right, now that you have a working keyboard device, it probably would be nice to do something with it. In other words, you still need to know how to read the buffered keyboard input. This is where the DIK_Read project comes into play.

The DIK_Read Project

In the last project, I showed you how to create an `IDirectInput8` interface along with an `IDirectInputDevice8` keyboard device. The main thing missing from that project was reading the data from the keyboard. In the DIK_Read project, I'll show you how to do the following tasks.

1. Initialize an `IDirectInput8` interface.
2. Create an `IDirectInputDevice8` keyboard device.
3. Read the keyboard's buffered data.
4. Check to see whether the Esc key is being pressed.

You have already learned how to do the first two steps, so I'll mainly concentrate on the last two items. Go ahead and load up the project to follow along with the code.

Changes to the Header

If you compare this project to the last project, you'll notice a few changes. Here is the latest code.

```
#include <stdio.h>
#include <dinput.h>

#define KEYBOARD_BUFFERSIZE          10
#define INPUTERROR_NODI              0x81000001
#define INPUTERROR_NOKEYBOARD        0x81000002
#define INPUTERROR_KEYBOARDEXISTS    0x81000003
LPDIRECTINPUT8                       pDI = NULL;
LPDIRECTINPUTDEVICE8                 pKeyboard = NULL;
BYTE                                 diks[256];
BYTE                                 ascKeys[256];
HKL                                  g_Layout;
// Windows globals
const char     *szWndClass           = "DirectInputKeyboardRead";
const char     *szProgramName        = "Direct-Input Keyboard Read Example";
HINSTANCE      g_hInstance;
// Function prototypes
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow);
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam,
 LPARAM lParam );
void vCleanup(void);
HRESULT     hrInitDirectInput(void);
HRESULT hrInitKeyboard(HWND hWnd);
int Scan2Ascii(DWORD scancode, unsigned short* result);
HRESULT hrReadKeyboard( void );
```

The two main additions, shown in bold, store the state of keys on the keyboard. When you read key data for a video game, you should not think in terms of which single key is being pressed. Instead, you must consider all the keys on the keyboard at once. This allows your game to accept multiple keys being pressed and released at the same time. Of course, I always recommend you keep keyboard input to a minimum, but sometimes it is necessary for gameplay.

Scan-Codes Versus ASCII Codes

The `diks[]` array stores the state of each scan-code key. The `acsKeys[]` array stores the state of each ASCII key. In its default state, a keyboard sends back codes called scan-codes. DirectInput has created several macros to identify special scan-codes. The main problem with scan-codes is that they are not compatible with standard string functionality. In other words, you cannot use scan-codes for alphabetic input and expect the letter B to follow the letter A. Scan-codes are stored out of order and rarely seem to follow a pattern. This is where ASCII codes come in. Using special functions, you can convert scan-codes to ASCII codes that are string-friendly.

The HKL Keyboard Layout Variable

The next global variable that is new to this project is the `g_Layout` object. This object stores the layout of the keyboard. Knowing the keyboard layout is important for language and regional issues. Some keyboards do not follow standard conventions, so you need to know the layouts before you can start converting scan-codes to ASCII codes. The `g_Layout` variable is actually initialized later in the `hrInitDirectInput()` function. Figure 13.5 offers a visual of how the new variables relate to the functions in the project.

Reading the Keyboard

If you follow the code along in the `WinMain()` function, you'll see that the message loop has changed a little bit.

```
// Process messages until the program is terminated
ZeroMemory( &msg, sizeof(msg) );
while( msg.message != WM_QUIT ) {
    if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) ) {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    else {
        hrReadKeyboard();
        // Exit the program if the Esc key is hit
        if( diks[DIK_ESCAPE] )
            break;
    }
}
```

The first change is the inclusion of a new function called `hrReadKeyboard()`.

The hrReadKeyboard() Function

The `hrReadKeyboard()` function checks the status of keys on the keyboard and sets their status flags. If you recall, the `diks[]` and `ascKeys[]` arrays each store 256 flags representing both scan-codes and ASCII key codes. If a key is up, then its flag is set to `0`. If a key is down, then its flag is set to `1`. The `hrReadKeyboard()` function operates by first checking the keyboard input buffer. If the buffer has data in it, the

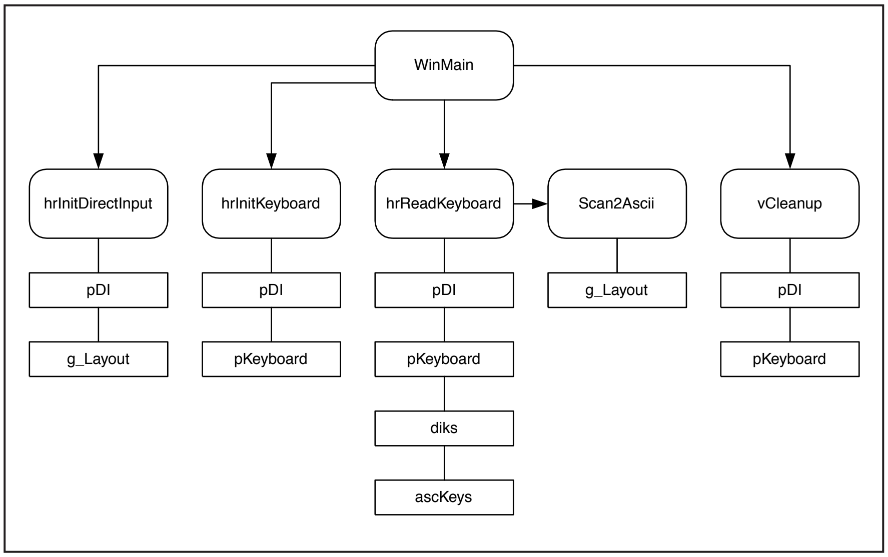


Figure 13.5

Global variable relationships in the DIK_Read project

function loops through the buffer and sets the appropriate flags for each key. Here is the code for the function.

```
HRESULT hrReadKeyboard( void )
{
    HRESULT                hr;
    DWORD                      dwCurBuffer;
    DIDEVICEOBJECTDATA          didKeyboardBuffer[KEYBOARD_BUFFERSIZE];
    DWORD                      dwItems = KEYBOARD_BUFFERSIZE;
    BYTE                           byteASCII;

    // Don't try to read the keyboard if the interface or device is invalid
    if( !pKeyboard || !pDI )
        return(INPUTERROR_NOKEYBOARD);
    // Read the buffered data
    hr = pKeyboard->GetDeviceData( sizeof(DIDEVICEOBJECTDATA),
 didKeyboardBuffer, &dwItems, 0);
    // Process data if there is data to read
    if ( dwItems ) {
        // Process the data
        for(dwCurBuffer = 0; dwCurBuffer < dwItems; dwCurBuffer++) {
            // Map scan-code to ASCII code
```

```
                byteASCII = Scan2Ascii(didKeyboardBuffer[dwCurBuffer].dwOfs);
                // Set key to be down (pressed)
                if( didKeyboardBuffer[dwCurBuffer].dwData & 0x80 ) {
                        ascKeys[byteASCII] = 1;
                        diks[didKeyboardBuffer[dwCurBuffer].dwOfs] = 1;
                }
                // Set key to be up
                else {
                    ascKeys[byteASCII] = 0;
                    diks[didKeyboardBuffer[dwCurBuffer].dwOfs] = 0;
                }
            }
        }
    return S_OK;
}
```

Okay, roll up your sleeves; it's time to get messy. The first things the function performs are what I like to call idiot checks. The function checks to make sure that `pKeyboard` and `pDI` are valid pointers. If either one of the pointers is `NULL`, then the function terminates. It's always a good practice to check pointers on which you are dependent. I know that many of the programs in this book do not include these checks, but it's never too late to start!

The IDirectInputDevice8::GetDeviceData() Function

The `GetDeviceData()` function retrieves buffered data from a DirectInput device. Here is the prototype.

```
HRESULT GetDeviceData(
    DWORD cbObjectData,
    LPDIDEVICEOBJECTDATA rgdod,
    LPDWORD pdwInOut,
    DWORD dwFlags
);
```

The first parameter is pretty easy because all you have to pass to it is the size of a `DIDEVICEOBJECTDATA` structure. The second parameter expects you to pass a pointer to an array of `DIDEVICEOBJECTDATA` data structures. This array will receive the buffered data.

The third parameter tells the function call how many elements are in the array used in the second parameter. Once the function completes, this variable is set to the number of elements actually received.

The last parameter lets you set certain read flags for the data. If you pass a DIGDD_PEEK here, the data you read is not removed from the buffer. Normally, you just pass a NULL value here.

Looping through the Buffered Data

If you look at the code, you'll see that it passes the dwItems variable to hold the number of elements read. The didKeyboardBuffer array holds the actual elements returned. Once the function completes, the code looks at the dwItems variable that holds the buffered data. If there is buffered data to read, then the program goes on to loop through the returned data.

The first thing that happens in the loop is a call to the Scan2Ascii() function. This call is made to convert the scan-code found in the didKeyboardBuffer[dwCurBuffer].dwOfs data member to an ASCII code. The byteASCII variable holds the converted value.

Next, the code checks the state of the key returned in the buffer. If the key can pass a logical AND with the value of 0 × 80, then you know the key is being held down. Otherwise, the key is in the released, or up, state. Once the state is determined, the appropriate diks[] and ascKeys[] arrays are updated.

The Scan2Ascii() Function

Unfortunately, the DirectX helper libraries do not include functions to convert scan-codes to ASCII codes. But, no need to despair; I have written one for you. The Scan2Ascii() function prototype follows.

```
BYTE Scan2Ascii(
    DWORD scancode,
);
```

The first parameter expects the scan-code you want to convert to ASCII. The function returns the converted ASCII code upon completion. I suggest that you copy the source code for this function and incorporate it into all of your projects if you don't have one of your own. It has proven time and time again to be very useful.

You might be asking, how does it work? First of all, you must get the current keyboard layout with the GetKeyboardLayout() function. If you look at the code, you'll see that it was called in the hrInitDirectInput() function. Remember that you only need to check the keyboard layout once per application instance. Once you have the layout, you can go into the meat of the function.

```
BYTE Scan2Ascii(DWORD scancode)
{
    UINT vk;

    vk = MapVirtualKeyEx( scancode, 1, g_Layout);
    return(vk);
}
```

The first and only function call is the `MapVirtualKeyEx()` function. This function takes a scan-code or ASCII code and converts it to a scan-code or ASCII code. The keyboard layout tells the function how to map the key returned. The `vk` variable holds the converted ASCII code. That's all there is to it!

Checking for the Escape or Enter Keys

Now that the `hrReadKeyboard()` function and all of its pals are out of the way, you can begin the fun part of checking for certain keys. Figure 13.6 shows how the program flows in relation to keyboard events. Once you have looked at the figure, check out the following code.

```
// Exit the program if the Esc key is hit
if( diks[DIK_ESCAPE] )
    break;
else if ( ascKeys[13] )
    break;
```

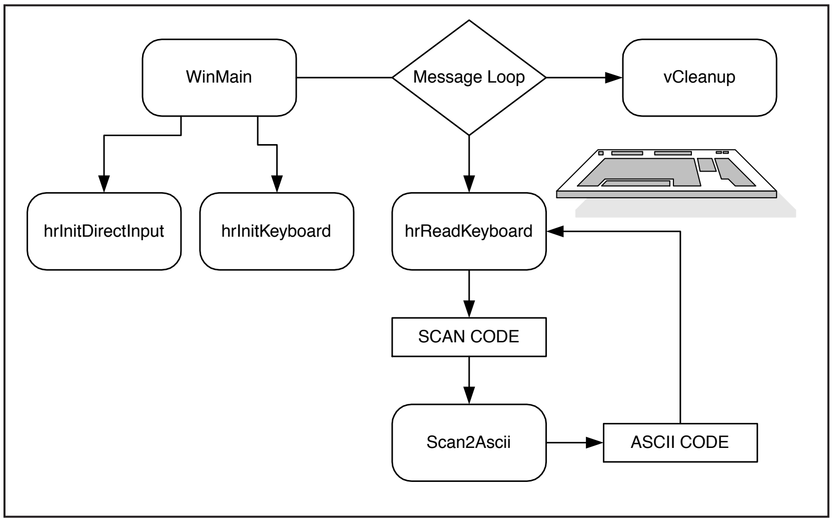


Figure 13.6

Program flow for the DIK_Read program

The DIK_ESCAPE definition is provided within the dinput.h header file. If you open that file, you'll find many other useful keyboard definitions. Basically, by checking for either the DIK_ESCAPE scan-code or the ASCII value of 13, the program exits whenever the user presses the Escape key or the Enter key.

Of course, you can do all sorts of other things here, but to keep things simple, I'll leave the program as is. If you want, you can output any keyboard data to a text file to see just how it works. Try some other DirectInput key codes such as DIK_LEFT and DIK_RIGHT.

You are now finished with the absolute basics of DirectInput. I didn't show you how to use mice because mouse controls are better left to standard Windows calls. The main reason is that the mouse is not dependant on Windows messages, which allows it to be very responsive. Another reason not to use the DirectInput mouse is that it records all information in relative coordinates. In other words, you never really know where the mouse is on the screen. It mainly detects how much the mouse has moved.

Introduction to DirectAudio

At the heart of the DirectX sound system is DirectAudio. DirectAudio is the combination of the former DirectSound and DirectMusic APIs that came with DirectX. Before I can show you how to initialize DirectAudio, you must first understand the following terms.

- Loader
- Performance
- Segment

The IDirectMusicLoader8 Interface

In DirectAudio, you have an interface called a *loader*. A loader loads all types of audio data, from wave files to MIDI files. The interface used to do all of this comes in the nice tidy package of the IDirectMusicLoader8 interface. You normally only need to create one of these objects per application, and it really isn't all that hard to do. The IDirectMusicLoader8 interface does not have a helper function to create it, so you must first initialize COM before you can make one.

The IDirectMusicPerformance8 Interface

You can think of the IDirectMusicPeformance8 interface as the sound manager for the system. It regulates playback, sets audio channels, sets sound parameters, and manages messages. This interface is the heart of the sound system. Each performance has one main channel and *n* number of secondary channels. Only one sound can play per primary channel, but multiple sounds are allowed on secondary channels. If you want two primary channels, you can create two performance interfaces.

The IDirectMusicSegment8 Interface

The IDirectMusicSegment8 interface controls and contains the actual sound data used for playback. Multiple sound tracks can be present in the object when storing music, or you can use only one track to store sound effects. When you play back segments, you can play them in either the primary or secondary channel of the performance object. As I mentioned in the performance description, only one primary segment can play per performance object.

Now that you have been exposed to the three object types within DirectAudio, you can reflect on what each type does.

- The IDirectMusicLoader8 interface loads sound.
- The IDirectMusicPerformance8 interface plays back sound.
- The IDirectMusicSegment8 interface holds the sound data that is loaded.

If you look at Figure 13.7, you'll see that I have made a neat little graphic to represent these objects. (Okay, maybe it's not that neat.)

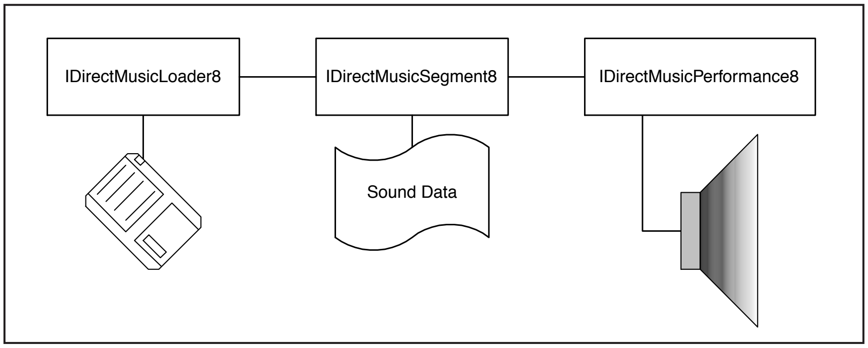


Figure 13.7
Interfaces in DirectAudio

How to Initialize DirectAudio

There is no better way to get your feet wet than to jump right into the pool. So go ahead and load up the DirectAudioInit project, and you can get started. The DirectAudioInit project's only purpose is to initialize the DirectAudio system and then shut it down. No sound is actually loaded or played back, but you will see how to set up the system to play sounds. The next project will show you how to play the actual sounds.

The DirectAudioInit Header Information

Most of the code in the project matches that of the examples you have seen so far, so I will not list the entire code base here. What I will do is point out relevant information. The first thing you need to know is what header files to include. Here is the header code for this project.

```
#include <stdio.h>
#include <Dmusici.h>
#define SOUNDERROR_MUSICLOADER                  0x85000001
#define SOUNDERROR_MUSICPERFORMANCE             0x85000002
#define SOUNDERROR_INITAUDIO                    0x85000003
#define SOUNDERROR_PATH                         0x85000004
#define SOUNDERROR_VOLUME                       0x85000005
// Windows globals
const char       *szWndClass            = "DirectAudio";
const char       *szProgramName         = "Direct-Audio Init Program";
HINSTANCE     g_hInstance;
// DirectAudio globals
IDirectMusicLoader8               *pLoader = NULL;
IDirectMusicPerformance8      *pPerformance = NULL;
// Function prototypes
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow);
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam,
 LPARAM lParam );
void vCleanup(void);
HRESULT hrInitDirectAudio(HWND hWnd);
```

The only header file you must have for DirectAudio is the Dmusici.h file. It actually includes the various header files within it so that you don't have to. You also need to add the dsound.lib and dxguid.lib libraries to your DirectAudio projects. Once you have these three items in place, you are good to go.

You may have noticed the `SOUNDERROR` definitions. These are just custom errors that I created to be returned when certain aspects of the initialization fail. You are free to add or remove these error codes at your leisure. They exist purely for error tracking.

Probably the most important objects in the preceding code listing are the `pLoader` and `pPerformance` objects. You only really need one loader in your project, and most of the time you only need one performance object.

The last thing that is noteworthy is the `hrInitDirectAudio()` function. This function does all of the initialization for the audio system. And don't forget about `vCleanup()`; even though I did not highlight it, it is used to shut down the audio interfaces.

The hrInitDirectAudio() Function

If you follow along in `WinMain()`, you'll see that it makes a call to the `hrInitDirectAudio()` function.

```
// Initialize the audio system
hr = hrInitDirectAudio(hWnd);
if( hr != S_OK ) {
    MessageBox( hWnd, "DirectAudio Error", "Unable to initialize
     DirectAudio.", MB_ICONERROR );
    vCleanup();
    exit(1);
}
```

If the function returns anything other than `S_OK`, a message box is displayed and the program exits. Nothing else happens in `WinMain()` except for the standard message handling. Without further ado, here is the code for the `hrInitDirectAudio()` function.

```
HRESULT hrInitDirectAudio(HWND hWnd)
{
    HRESULT                  hResult;
    IDirectMusicAudioPath8   *path;

    // Initialize COM
    CoInitialize(NULL);
    // Create the loader
    if( FAILED( hResult = CoCreateInstance(CLSID_DirectMusicLoader, NULL,
        CLSCTX_INPROC, IID_IDirectMusicLoader8,
         (void**)&pLoader) )) {
        return(SOUNDERROR_MUSICLOADER);
    }
    // Create the performance
    if( FAILED( hResult = CoCreateInstance(CLSID_DirectMusicPerformance,
     NULL,
        CLSCTX_INPROC, IID_IDirectMusicPerformance8,
         (void**)&pPerformance ))) {
        return(SOUNDERROR_MUSICPERFORMANCE);
    }
    // Initialize the audio
    if( FAILED( hResult = pPerformance->InitAudio(
        NULL,
        NULL,
        hWnd,
        DMUS_APATH_DYNAMIC_STEREO,
        4,
        DMUS_AUDIOF_ALL,
        NULL
        ))) {
        return(SOUNDERROR_INITAUDIO);
    }
    // Get the default path
    if( FAILED( pPerformance->GetDefaultAudioPath( &path ) ) )
        return(SOUNDERROR_PATH);
    // Set the default volume
    if( FAILED( path->SetVolume(0,0) ) )
        return(SOUNDERROR_VOLUME);
    return(S_OK);
}
```

It might look a little wordy, but believe me; it's not all that bad.

Flow of the hrInitDirectAudio() Function

To better visualize how the function operates, take a look at Figure 13.8. As you can see, the first thing that is accomplished is the initialization of COM. This is necessary because all DirectAudio functions are COM-based.

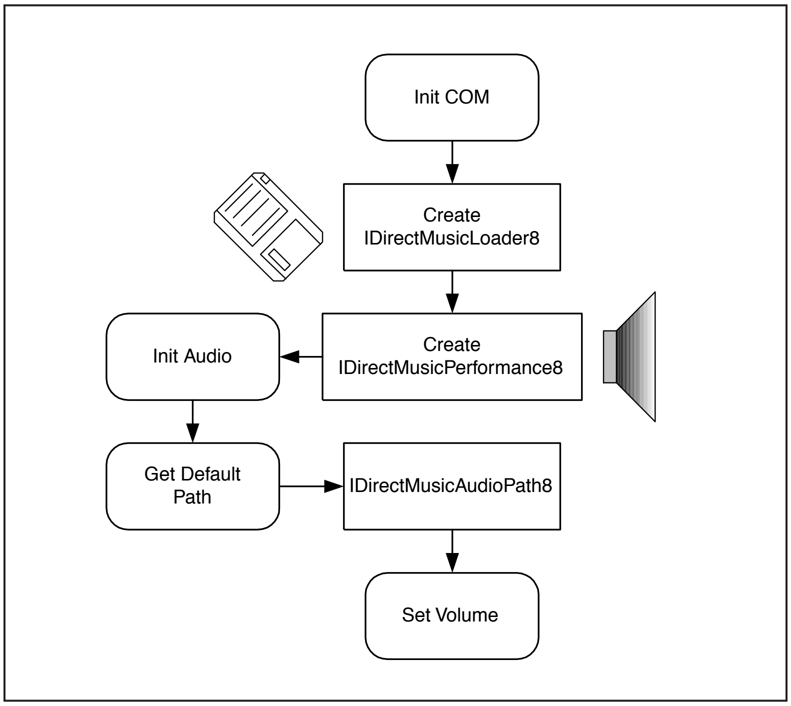


Figure 13.8

Structure of the `hrInitDirectAudio()` *function*

The second step is the creation of the `IDirectMusicLoader8` interface. The third step is the creation of the `IDirectMusicPerformance8` interface. The fourth step is to call the `InitAudio()` member function of the performance interface. The fifth step is to call the `GetDefaultAudioPath()` member function of the performance interface. The last step is to set the default volume.

Creating the IDirectMusicLoader8 Interface

Before you can create any interfaces, you must first initialize COM using the all too familiar `CoInitialize()` function. Once that is out of the way, you can create the `IDirectMusicLoader8` interface using the `CoCreateInstance()` function.

```
if( FAILED( hResult = CoCreateInstance(CLSID_DirectMusicLoader, NULL,
    CLSCTX_INPROC, IID_IDirectMusicLoader8,
     (void**)&pLoader) )) {
```

```
    return(SOUNDERROR_MUSICLOADER);
}
```

If the function fails, then it returns an error of SOUNDERROR_MUSICLOADER. If the function succeeds, then the pLoader pointer is set to hold the interface.

Creating the IDirectMusicPerformance8 Interface

Once the loader is created, you can create the performance interface with the same call used in creating the loader.

```
// Create the performance
if( FAILED( hResult = CoCreateInstance(CLSID_DirectMusicPerformance, NULL,
    CLSCTX_INPROC, IID_IDirectMusicPerformance8,
     (void**)&pPerformance ))) {
    return(SOUNDERROR_MUSICPERFORMANCE);
}
```

If the function fails, it returns an error of SOUNDERROR_MUSICPERFORMANCE. If the function succeeds, the pPerformance pointer is set to hold the interface.

The IDirectMusicPerformance8::InitAudio() Function

The InitAudio() member function of the performance interface initializes the performance interface for use. You cannot play any sounds or use the interface until this function is called. You can also set up the default audio path using this call, if you so choose. Here is the prototype.

```
HRESULT InitAudio(
    IDirectMusic** ppDirectMusic,
    IDirectSound** ppDirectSound,
    HWND hWnd,
    DWORD dwDefaultPathType,
    DWORD dwPChannelCount,
    DWORD dwFlags,
    DMUS_AUDIOPARAMS *pParams
);
```

The first parameter lets you specify an IDirectMusic8 interface to either use or receive from the function call. If you leave this at NULL, an internal interface is created. The second parameter lets you specify an IDirectSound8 interface to either use or receive from the function call. If you leave this at NULL, an internal interface is created. The third parameter expects the handle of the window to be used for the sound system.

The fourth parameter specifies the type of the default audio path. The following types are valid.

- DMUS_APATH_DYNAMIC_STEREO
- DMUS_APATH_DYNAMIC_MONO
- DMUS_APATH_DYNAMIC_3D
- DMUS_APATH_SHARED_STEREOPLUSREVERB

The fifth parameter lets you specify the number of performance channels with the default audio path.

The sixth parameter specifies various flags for use with the interface. Table 13.2 lists the available flags.

The last parameter lets you pass in a DMUS_AUDIOPARAMS data structure containing parameters for the performance interface. If you leave this at NULL, the default settings are used.

Table 13.2 InitAudio() Flags

Flag	Description
DMUS_AUDIOF_3D	Allow 3D buffers
DMUS_AUDIOF_ALL	Use all features
DMUS_AUDIOF_BUFFERS	Allow multiple buffers
DMUS_AUDIOF_ENVIRON	Use environmental modeling
DMUS_AUDIOF_EAX	Use EAX effects
DMUS_AUDIOF_STREAMING	Allow streaming wave files

If you look back at the code the program uses to call the `InitAudio()` function, you'll see that the program is trying to set up a stereo playback system with all effects enabled. If all goes well, then `S_OK` is returned from the function.

Getting the Default Audio Path

Before you are finished with initialization, you must retrieve the default audio path. The audio path is what you use to play back sounds, so you need to have access to it. This is accomplished using the `IDirectMusicPerformance8::GetDefaultAudioPath()` function.

```
// Get the default path
if( FAILED( pPerformance->GetDefaultAudioPath( &path ) ) )
    return(SOUNDERROR_PATH);
```

The only parameter you pass to this function is the pointer of an `IDirectMusicAudioPath8` interface to receive the audio path information.

Now that you have the audio path, you can go ahead and set up the system volume using the `IDirectMusicAudioPath8::SetVolume()` function.

TIP

You don't necessarily have to get the default audio path; it's just good practice to keep it handy. In this program, you are getting the path to set the volume for it.

The IDirectMusicAudioPath8::SetVolume() Function

The `SetVolume()` function lets you specify the volume for an individual audio path. Here is the prototype.

```
HRESULT SetVolume(
    long lVolume,
    DWORD dwDuration
);
```

The first parameter is the actual volume settings. The range allowed here is from 0 to –9600. Zero is considered full volume.

The second parameter specifies how many milliseconds it should take for the volume change to be fully in place. You can use this to perform ramping effects. By setting the second parameter to `0`, you tell the program to change the volume immediately.

Cleaning Up DirectAudio

You are almost done with DirectAudio. All you have left to do is clean up the system before your program exits. This is handled by the `vCleanup()` function.

```
// Clear the loader
if( pLoader ) {
    pLoader->Release();
    pLoader = NULL;
}
// Clear the performer
if( pPerformance ) {
    pPerformance->Release();
    pPerformance = NULL;
}
// Shut down COM
CoUninitialize();
// Clear the window class object
UnregisterClass( szWndClass, g_hInstance );
```

The two interfaces you created for DirectAudio are cleaned up with the `Release()` function. After those are released, you can shut down the COM system you initialized earlier. The last thing to do before leaving is clear out the window class. That's it; you have just initialized DirectAudio! What's that? You want to know how to play a sound? Well, geez, okay. . . . Oh, but before you go, be sure to look at Figure 13.9 to see how the whole program operated.

Playing Sound with DirectAudio

All right, now it is time for the fun part of DirectAudio—playing sound! The DirectAudioPlaySound project loads up a test wave file of yours truly and plays it back three times in a row. Each time the sound plays, the next one kicks in just a half a second later. This results in the sounds overlapping one another. The reason

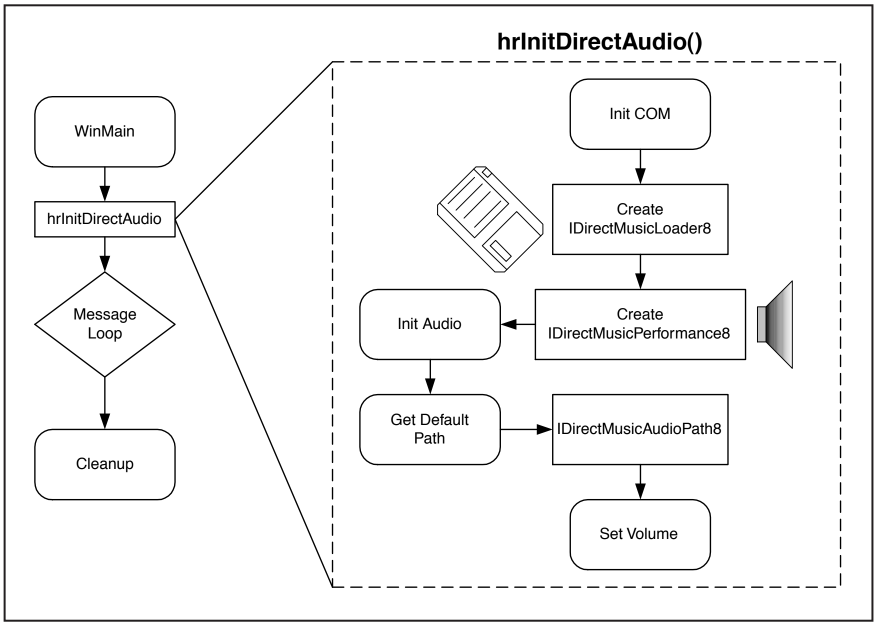


Figure 13.9

Operation of the DirectAudioInit program

I do this is to show you how cool DirectAudio is, in that you can overlay all sorts of sound effects. Feel free to change out the sound that is played to see the effect with your own files. Go ahead and load up the project now to follow along.

New Additions to the Project

I added a few files to the project to play sounds. The new additions are

- `dxutil.cpp`
- `dxutil.h`
- `mmsystem.lib`

You should already be familiar with these files because you have used them in the past. The dxutil files are provided to aid in some of the rather mundane tasks associated with DirectX. The mmsystem.lib library file is needed for one of the functions in the dxutil.cpp file.

The header portion of the code has changed a little bit to include the new functionality as well. Here is the new header code.

```
#include <stdio.h>
#include <Dmusici.h>
#include "dxutil.h"

#define SOUNDERROR_MUSICLOADER                0x85000001
#define SOUNDERROR_MUSICPERFORMANCE           0x85000002
#define SOUNDERROR_INITAUDIO                  0x85000003
#define SOUNDERROR_PATH                       0x85000004
#define SOUNDERROR_VOLUME                     0x85000005
#define SOUNDERROR_LOAD                       0x85000006
#define SOUNDERROR_DOWNLOAD                   0x85000007
#define SOUNDERROR_NOSEGMENT                  0x85000008
#define SOUNDERROR_PLAYFAIL                   0x85000009
// Windows globals
const char     *szWndClass                    = "DirectAudio";
const char     *szProgramName                 = "Direct-Audio Init Program";
HINSTANCE      g_hInstance;
// DirectAudio globals
IDirectMusicLoader8                           *pLoader = NULL;
IDirectMusicPerformance8                      *pPerformance = NULL;
IDirectMusicSegment8                          *pSound = NULL;
// Function prototypes
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow);
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam,
 LPARAM lParam );
void vCleanup(void);
HRESULT hrInitDirectAudio(HWND hWnd);
HRESULT hrLoadSound(char *szFileName);
HRESULT hrPlaySound(IDirectMusicSegment8 *pSegment);
```

I took the liberty of putting all of the changes into bold text for you. The first change is the inclusion of the dxutil.h header file that is necessary in order to use the dxutil helper functions. The next few changes deal with some new error messages. The important change comes in the form of the `IDirectMusicSegment8` global variable named `pSound`. This variable holds the loaded wave file.

The next two lines of code deal with the function prototypes used to load and play the wave file. If you follow along in `WinMain()`, you'll see that `hrLoadSound()` is called first. If you look at Figure 13.10, you can see the main components of the project.

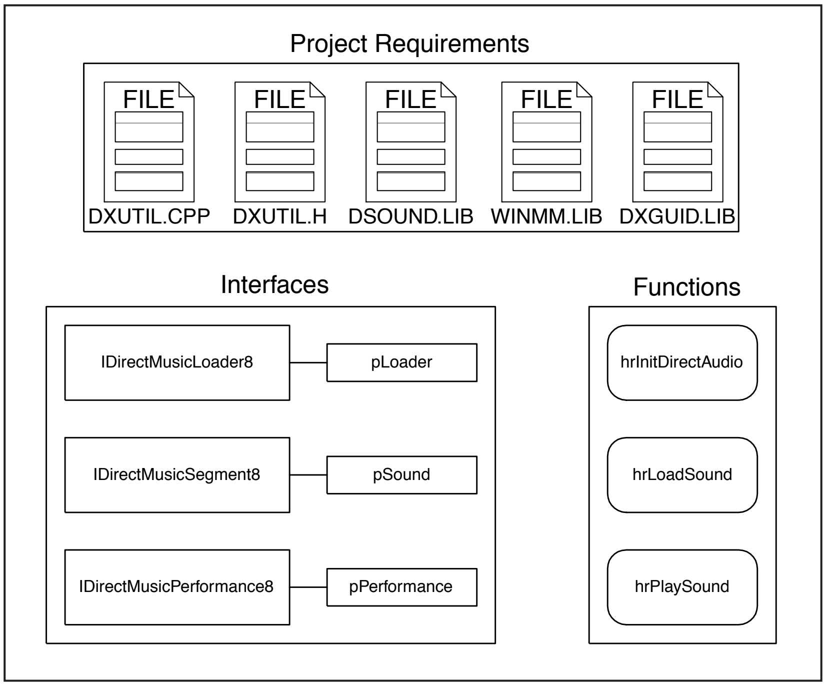


Figure 13.10

Components of the DirectAudioPlaySound project

The hrLoadSound() Function

The `hrLoadSound()` function takes only one parameter, the name of the sound to load. The function uses the global `IDirectMusicSegment8` interface to hold the sound once it is loaded. Here is the code for the function.

```
HRESULT hrLoadSound( char *szFileName )
{
    WCHAR     wstrFileName[256];
    // Convert the filename to wide character format
    DXUtil_ConvertGenericStringToWide( wstrFileName, szFileName );

    // Make sure audio is initialized
    if( !pLoader )
        return(SOUNDERROR_MUSICLOADER);
    if( !pPerformance )
        return(SOUNDERROR_MUSICPERFORMANCE);
    // Clean up sound if it already exists
    if( pSound ) {
        pSound->Unload( pPerformance );
        pSound->Release();
        pSound = NULL;
    }
    // Load the sound from a file
    if ( FAILED(pLoader->LoadObjectFromFile(
        CLSID_DirectMusicSegment,
        IID_IDirectMusicSegment8,
        wstrFileName,
         (LPVOID*) &pSound
        )))
    {
        return(SOUNDERROR_LOAD);
    }
    // Download the data
    if ( FAILED ( pSound->Download( pPerformance ) ) ) {
        return(SOUNDERROR_DOWNLOAD);
    }
    return(S_OK);
}
```

Converting the Name to Wide String Format

The first strange thing you might notice is that the function converts the filename from normal string format to wide string format. The reason this is necessary is that DirectX 8.0 and above expect you to pass wide strings for any string parameters. This allows DirectX to not worry about whether you are using Unicode. In fact, I recommend that you always use Unicode when possible. As you probably have noticed, I don't use it very much, but I am trying! The dxutil helper function, `DXUtil_ConvertGenericStringToWide()`, performs the conversion. Once the filename is converted, it is held in the `wstrFileName` variable.

Checking the Required Interfaces

The next few lines of code deal with the dummy-checking aspects of the code. Basically, it is checking to make sure that the required interfaces are present. Remember, you must have a loader and a performance object present to load a sound. If the `pSound` object is already in use, the code unloads and releases it so that it can load in new data.

The IDirectMusicLoader8::LoadObjectFromFile() Function

The `IDirectMusicLoader8::LoadObjectFromFile()` function loads up a sound resource from a file and places it into a segment object. Here is the prototype.

```
HRESULT LoadObjectFromFile(
    REFGUID rguidClassID,
    REFIID iidInterfaceID,
    WCHAR *pwzFilePath,
    void ** ppObject
);
```

The load function can actually load many types of data for DirectAudio. This is where the first parameter comes into play. It lets you specify the GUID for the type of object you want to load. The available types appear in Table 13.3.

Table 13.3 Loader Type GUIDs

Type	GUID	Extension
Audiopath	`CLSID_DirectMusicAudioPathConfig`	.aud
Band	`CLSID_DirectMusicBand`	.bnd
Chordmap	`CLSID_DirectMusicChordMap`	.cdm
Container	`CLSID_DirectMusicContainer`	.con
DLS collection	`CLSID_DirectMusicCollection`	.dls
Script	`CLSID_DirectMusicScript`	.spt
Segment	`CLSID_DirectMusicSegment`	.sgt
Style	`CLSID_DirectMusicStyle`	.sty
Template	`CLSID_DirectMusicSegment`	.tpl
Toolgraph	`CLSID_DirectMusicGraph`	.tgr
Wave	`CLSID_DirectSoundWave`	.wav

The second parameter specifies the unique identifier of the interface. I use the `IID_IDirectMusicSegment8` ID here. The third parameter passes in the filename of the object to load. The last parameter receives the loaded object once the function completes.

If the function succeeds, then the `pSound` global segment variable will be holding the newly loaded wave file. Once you have a valid audio segment, you need to send it to the performance interface.

The IDirectMusicSegment8::Download() Function

Before a sound can be played, it must first be transferred to the performance interface that is going to play it. You transfer it using the `Download()` function. Here is the prototype.

```
HRESULT Download(
    IUnknown *pAudioPath
);
```

The one and only parameter specifies the audio path in which you want to put the object.

```
// Download the data
if ( FAILED ( pSound->Download( pPerformance ) ) ) {
return(SOUNDERROR_DOWNLOAD);
}
```

As you can see in the code used to download the segment, the code passes in the `pPerformance` pointer. This sends the segment data to the performance interface created earlier. If the function fails, then the code returns an error message.

TIP

If a segment has used the `Download()` **function, you should be sure to call** `Unload()` **before releasing the segment.**

The hrPlaySound() Function

Once you have the IDirectMusicSegment8 interface loaded up with sound data, you can go ahead and play it! For this, you use the hrPlaySound() function.

```
HRESULT hrPlaySound(IDirectMusicSegment8 *pSegment)
{
    // Make sure there is a performance object present
    if( !pPerformance )
        return( SOUNDERROR_MUSICPERFORMANCE );
    if( !pSegment )
        return( SOUNDERROR_NOSEGMENT );
    // Play the sound segment
    if( FAILED ( pPerformance->PlaySegmentEx(
        pSegment,
        NULL,
        NULL,
        DMUS_SEGF_DEFAULT | DMUS_SEGF_SECONDARY,
        0,
        NULL,
        NULL,
        NULL
        )))
        return( SOUNDERROR_PLAYFAIL );
    return(S_OK);
}
```

The hrPlaySound() function takes one parameter, and that parameter passes in the segment to be played. There is only one segment in this project, pSound, but I made the function parameter-based so that you can use it in other applications.

The first few lines of code just do your normal dummy-checking to make sure that the performance and segment objects are valid. The main part of the function is the call to the IDirectMusicPerformance8::PlaySegmentEx() function.

The IDirectMusicPerformance8::PlaySegmentEx() Function

The only purpose of the PlaySegmentEx() function is to play back a segment. It is very similar to the PlaySegment() function except the PlaySegmentEx() function has more capabilities. Here is the prototype.

```
HRESULT PlaySegmentEx(
    IUnknown* pSource,
    WCHAR *pwzSegmentName,
    IUnknown* pTransition,
    DWORD dwFlags,
    __int64 i64StartTime,
    IDirectMusicSegmentState** ppSegmentState,
    IUnknown* pFrom,
    IUnknown* pAudioPath
);
```

The first parameter expects the segment you want to play. The second parameter is not implemented in DirectX 8.0, so leave it at NULL. Don't you just love these types of parameters? The third parameter lets you specify a transition object to associate with the playback. This is optional and can be left at NULL. The fourth parameter lets you specify flags to be used in the playback. There are tons of these flags available, but the most commonly used are DMUS_SEGF_DEFAULT | DMUS_SEGF_SECONDARY. Refer to the DirectX documentation for a full list.

The fifth parameter lets you specify when to start playing the sound. If you set this to 0, then the sound attempts to play immediately. The sixth parameter receives the IDirectMusicSegmentState8 pointer for the playback. You can set this to NULL if you do not want to retrieve the state pointer. The seventh parameter lets you specify an audio path or segment state to stop when the passed-in segment starts playing. This can be set to NULL. The last parameter lets you specify the audio path to use when playing the sound. If you leave it at NULL, then the sound plays on the default path.

If you look at the code, you'll see that it uses the DMUS_SEGF_DEFAULT | DMUS_SEGF_SECONDARY flags to play the sound. This tells the sound to play on the secondary channel. Because only one sound can play at a time on the primary channel, you need to use the secondary channel for sounds you want to play simultaneously.

The code in `WinMain()` is calling the `hrPlaySound()` function multiple times. In between each call, the program waits 500 milliseconds, or half of a second. This results in the sounds overlapping each other. You can remove all but one of the `hrPlaySound()` function calls if you just want to hear the sound play back normally.

Cleaning Up the Segment

The final part of the project deals with cleaning up the interfaces. Nothing is new here except for the cleanup of the segment interface.

```
// Clear the sound
if( pSound ) {
    pSound->Unload( pPerformance );
    pSound->Release();
    pSound = NULL;
}
```

The new function here is the `Unload()` function. This is required if you downloaded the segment into a performance object. Once you unload it, you can go on to release it and then set the pointer to `NULL`. Check out Figure 13.11 to see all of the steps to playing a sound.

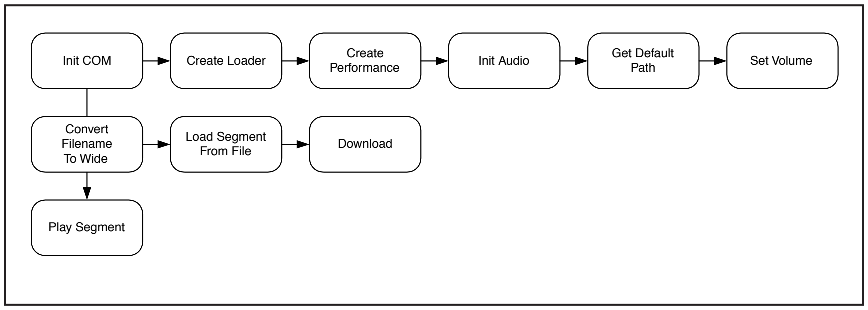


Figure 13.11

Complete steps to playing and releasing a sound

Review

Before you move on, take a moment to reflect on some key points covered in the chapter.

- The IDirectInput8 interface is the heart of the DirectInput system.
- Input devices such as keyboards and mice are created using IDirectInput8's CreateDevice() member function.
- Buffered input is desired so that your game never loses user input. If you do not use buffered data, players will get frustrated when the game misses their input.
- Before using a device, you must acquire it with the Acquire() function. When you are done with the device, you must use the Unacquire() function to properly release it.
- Keyboard devices return scan-code input codes. You must convert the scan-codes to ASCII if you want to check ASCII values. This is done using the Scan2Ascii() function. The DIK definitions are useful for checking the native scan-codes.
- The IDirectMusicLoader8 interface is used to load sound media.
- The IDirectMusicPerformance8 interface is used to manage loaded sounds.
- The IDirectMusicSegment8 interface is used to hold loaded sound media.

Now you are armed with the knowledge to load and play back sounds. You also know how to accept responsive input from the keyboard. With this new knowledge, you are ready to compose a complete game. Turn the page to begin your next adventure!

CHAPTER 14

SpacePirates: Your First Complete Game

I think the chapter title pretty much sums it up. You are now ready to write a complete peer-to-peer video game. Every chapter before this one has prepared you for this day. Get ready; the adventure begins anew!

Although you have learned a great deal so far, there is still much to digest within this chapter. Here is a list of the topics I will cover.

- Game design
- Timing
- Network synchronization
- Classes
- Video initialization
- Graphics
- Player management

SpacePirates: The Game Design

Before you write a game, you really need to write out a game design. Most nonprofessional developers just think of an idea and start coding it. This style might work for simple projects, but serious games require a ton of work on the front end. In fact, you shouldn't be surprised when the planning phase of your project takes up almost as much time as the implementation phase.

The Game Objective

The first thing you need to know before you write a game is the player's objective. Is it to advance a virtual character in skill? Is it to get the most frags? Is it to take over the world? The objective is completely up to you, but you should be able to sum it up in one sentence.

The game objective for *SpacePirates* is to get the highest score. It's really that simple—just get the highest score. All right, now that you have the objective, how does the player accomplish it? Well, in this game, the players will fire pulses of energy that explode upon impact on the arena or other players. Whenever one of your pulses hits another player, you get points. So the objective is to get the highest score by shooting other players.

The Game Objects

The game objects are the pieces that make up the game. You can think of the whole process as similar to creating a sculpture. The design is what the sculpture is going to look like, the objective is what the sculpture is going to be doing pose-wise, and the objects represent what the sculpture is made of. There you go, a bona fide analogy. Anyway, *SpacePirates* will have players who play in an arena. The game will include the following objects.

- Players
- Arena
- Bullets
- Explosions
- Sound
- Text

The Players

Because this is a multiplayer game, multiple players will be present. You can be assured that one player will always be there, but the total number is an uncertainty. One good way to keep track of each player is to create a player data structure or class. The *SpacePirates* game has a data structure called `PLAYER_INFORMATION`. This data stores everything about each player in the game.

NOTE

There are various packages you can use to create or modify 3D models for your games. My personal favorite is 3ds max (*3D Studio Max*). The only downside is that it is a very expensive package if you are a hobbyist programmer. Some cheaper packages that get the job done are MilkShape 3D and trueSpace.

Besides the player information, you also need graphics for the players. In *SpacePirates*, the players will control mechanized warriors. Each warrior comes from the same 3D object files, though. This saves on memory and storage because you must load only one set of geometry for all of the players. The player's object looks like that shown in Figure 14.1.

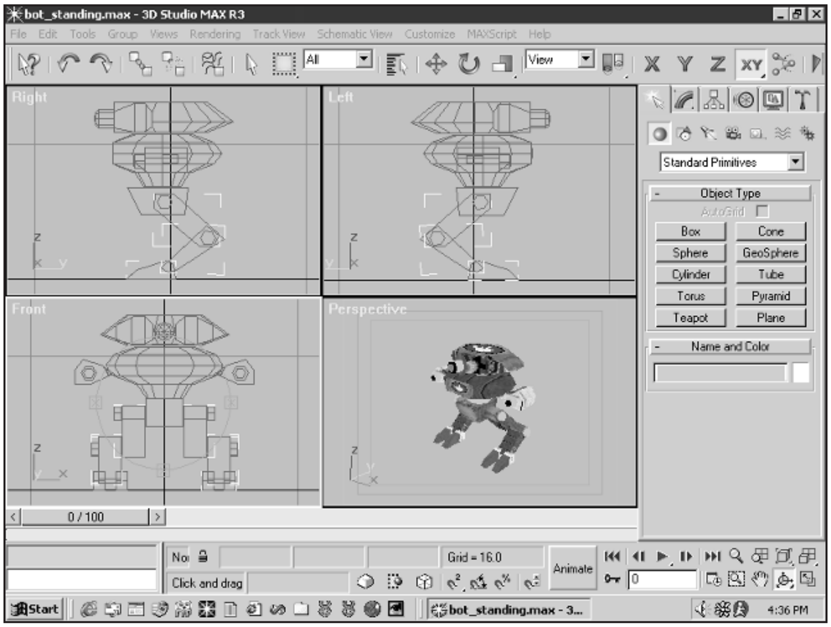


Figure 14.1

The player in 3ds max

The Arena

The players in the game need somewhere to play, so in *SpacePirates*, you will create a game arena from scratch. The arena has the shape of a square and four walls. You create all of this by rendering a single quad in many different positions. This saves a lot of memory because you must load only two polygons in memory for the entire arena. The layout for the arena appears in Figure 14.2.

Because the arena consists of a quad, you need a storage structure to hold the vertex information. In the program's header file, you'll find the all-too-familiar `MYVERTEX` data structure. The structure holds all of the vertex data, including the position, normals, color, and texture information.

The Bullets

It was determined in the design that players would be able to shoot pulses of energy at each other. This means that you need to keep track of each bullet in the scene while the game is being played. To do this, the program has a `BulletObject` data structure that holds all pertinent information.

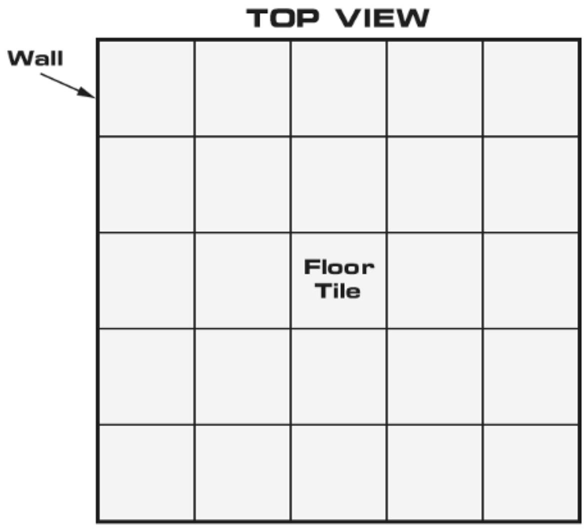


Figure 14.2

The arena

You also need a graphic for the energy pulses, which you add with a targa graphics file called pulse.tga. This image holds the graphic of the energy pulse along with an alpha map to define transparency. Because you don't want blocks of energy bolts flying around, you need a texture map that includes transparency information. The bullet image and its alpha channel appear in Figure 14.3.

NOTE

Adobe Photoshop is by far my favorite tool for editing 2D graphics. It costs about $600, but it is well worth the price.

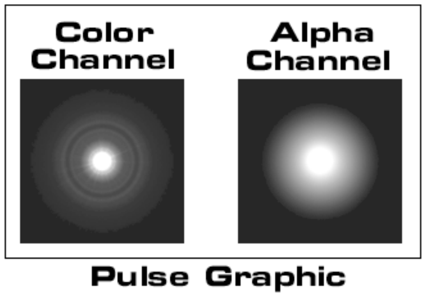


Figure 14.3

The energy pulse graphic

The Sounds

Every successful game has sounds to go with it. Unfortunately, many game developers design sound into their projects as a last step. You should pre-plan all sound activity when creating a game and use it extensively. Believe me, sound makes a huge difference in the gaming experience. Good sound design results in your game environments being more lifelike and rich. Players also benefit by experiencing a more real sense of immersion. Okay, now that I have climbed off my high horse, I can show you how sound is handled in *SpacePirates.*

To make sound handling easier on the code-base, I created `SoundSystem` and `GameSound` classes. The `SoundSystem` class loads and plays back `GameSound` objects. You have to look at a little C++, but don't fret; I'll show you how all of it was written.

The game uses only two sounds—one for when the player fires his cannon and the other for when the energy pulses hit something. I guess you could bypass the whole class system for only two game sounds, but I think you will see the advantage.

The Text

At this point, you might be wondering why text is needed in a 3D game! The reason you need text in your game is to show the player his score. You also use text to display which player has the highest score and the command prompt.

The problem with a 3D game is that there is no simple way to put 2D text on the display. Instead, you have to map 3D objects to 2D space and simulate 2D graphics. Does that sound a little confusing? The cool part is that DirectX includes a helper interface called `ID3DXFont`. The `ID3DXFont` interface can load system fonts and display them in 2D space on a 3D display. If you look at the *SpacePirates* header file closely, you'll see the `pD3DXFont` variable that holds the font interface.

Game Timing

The subject of timing is rather important, so please don't skip over it! Many novice game programmers make the mistake of putting hardware-specific timing into their game loops. What do I mean by this? I mean that they might do such things as update their game objects based on the frame counter and not on how much actual time has passed. There is no better way to explain this than with the following pseudo-code.

```
Begin Game Loop
     Player's X-Position += 5
End Game Loop
```

In this code, the game might be moving a player across the screen 5 units at a time. Well, suppose the player has a 500MHz computer, and it takes his computer 1/30th of a second for the game loop to process. So in one second, his game updates 30 times. This results in the player moving 150 units per second. The formula works in the following way:

Frames Per Second * Speed = Units Per Second

Suppose you program your entire game based around this speed. Based on this setting, you might limit the player's speed to 10 to keep him from moving too fast. You might also limit the player to only firing every 50 game updates. The whole game would work just fine on your computer, but this is where the problem begins. Imagine if you then ran the game on your friend's computer with a 1GHz system. Imagine, for simplicity's sake, that his system runs the game at 60 frames per second, twice as fast as yours does. Now apply the movement formula.

60 * 5 = 300 Units Per Second

Wow! He scoots across the game world at 300 units per second. Now imagine playing against each other on the Internet. He would be able to transverse the map twice as fast as you could. Just imagine the kind of advantage he would have.

Time-Based Movement

There is light at the end of the tunnel, though. The most common way to regulate game updates is with timers. Internal timers available in Windows tell you how many milliseconds have passed since the system was turned on. Because the timers work in milliseconds, you can be accurate up to 1/1000th of a second.

Now if you were using the same example from the last section, you would have player speed set to five units. Suppose you want to limit the player to 150 units per second of movement. One good measurement is to keep everything locked at 30 frames per second. The reason 30 frames per second is good is that the human eye is tricked into seeing smooth movement at that speed. Anything over 30 frames per second pretty much wastes frames. Of course, I am sure many people would argue with that statement, but as far as motion goes, 30 frames per second is just fine. Just don't get the motion limit confused with performance. It is best that your game run at hundreds of frames per second; you just need to regulate updates to 30 frames per second.

Okay, you have a predetermined limit of 150 units of movement per second. So apply the following formula:

Total Units / 30 = Units Per Frame

In this case, the value of units per frame is 5. Now that you know this, you have a locked-in speed to use. To implement time-based movement, your game loops have to look different. Here is the pseudo-code for time-based movement.

```
Old Time = Get Current Time
Begin Game Loop
     Get Current Time
     Is Current Time > (Old Time + 33)
          Player's X-Position += 5
          Old Time = Current Time
End Game Loop
```

The pseudo-code makes sure that your game does not update more than 30 frames per second. Because the player's position is not updated unless 33 milliseconds have passed, you can be sure that he does not move too quickly. If you caught the glaring mistake in this code, then give yourself a pat on the back.

Time Multipliers

The aforementioned mistake is really not a mistake per se, but rather an omission. Imagine what happens if the player's computer cannot run fast enough to update 30 times a second. I'll tell you what happens: His game runs slower than anyone else's who can run at least 30 frames per second. This has the same bad result as before. You have succeeded in keeping fast players from exceeding the limits, but now how do you keep slow players from lagging behind? The answer lies within time multiplication. The basis of this concept is that you take the time elapsed since the last update and divide it by your locked frame rate. Here is the formula.

```
Elapsed Time = Current Time - Old Time
Multiplier = Elapsed Time / Frame Lock
```

For example, assume that the player in your game has only enough computing power to update 15 times per second instead of 30 times per second. To figure out the multiplier, you use the preceding formula like so:

```
Elapsed Time = 66 milliseconds ( speed of 2 updates per second )
Multiplier = (66 / 33) = 2
```

In this example, you ended up with a multiplier of 2. Now that you have the multiplier, you just update the player's position with his speed multiplied by 2. So your pseudo-code now looks like the following.

```
Old Time = Get Current Time
Begin Game Loop
```

```
    Get Current Time
    Is Current Time > (Old Time + 33)
        Multiplier = ( (Current Time - Old Time ) / 33 )
        Player's X-Position += (5 * Multiplier)
        Old Time = Current Time
End Game Loop
```

With the new and improved code, the player's speed adjusts to the frame rate. Things won't appear as smooth for the player with the slow machine, but at least the playing field is even. The *SpacePirates* game uses a 30 frames-per-second locking mechanism that works just like the preceding code. To see this concept illustrated, take a look at Figure 14.4. In the illustration, you can see that the 1000MHz machine player moves at twice the speed of the 500MHz machine player. If you didn't throttle the speed of the game updates, then the 1000MHz player would have a distinct advantage.

Network Synchronization

Network synchronization covers the same ground as frame synchronization in the last section. Basically, you need to keep everyone in the game synchronized with each other. Many multiplayer games limit their network updates to no more than 10 times per second. By keeping updates limited to 10 times per second, you can limit the amount of bandwidth needed to play the game. If your game requires updates constantly, then you run into big problems with laggy players.

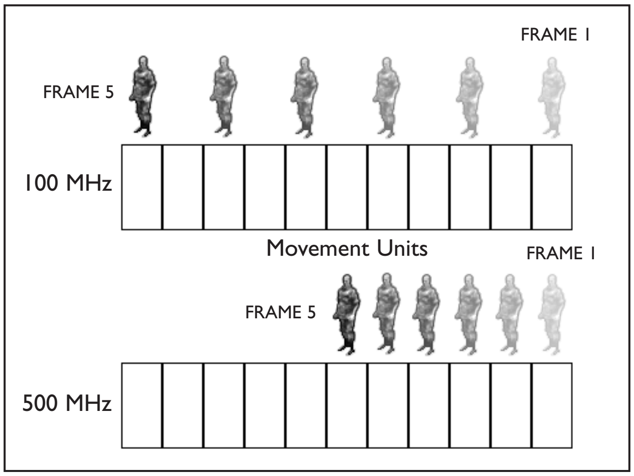


Figure 14.4

Time multipliers

Scheduled Updates

SpacePirates maintains network synchronization by sending update packets only once every 10 seconds. You can see this in the `vUpdateNetwork()` function. If it is time to update, the player broadcasts his position, heading, velocity, and score to everyone else in the game.

Now, if you sent packets only 10 times per second, then you might see jerky movements from other players. To keep this from happening, the game lets players broadcast changes in their state to other players whenever they happen. For an example of this, take a look at Figure 14.5.

In Figure 14.5, Player A is sending scheduled updates to Player B and vice versa. Player A decides to fire his cannon, so a bullet packet is sent to Player B immediately. This prevents a lag in time from when the bullet is fired and when it should appear. If Player A waited for the next network synch to happen, then the bullet would fall behind in time.

Non-Synchronization

One thing you'll notice with *SpacePirates* is that what one player sees on his screen might not be exactly what another player sees. The reason is that none of the packets is sent via guaranteed delivery. Another reason is that scoring is left completely up to the victim. In the game, you do not tell another player when he is hit. Instead, players decide for themselves when they take damage.

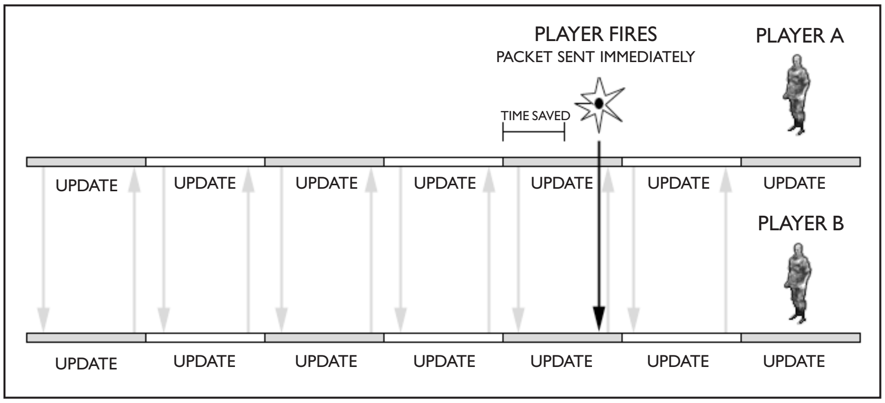


Figure 14.5
Network synchronization

One benefit of non-synchronization is that players won't gripe when they take unseen damage. Another benefit is that it reduces cheating. If I can't tell you when you take damage, then I can't send you fake damage packets. I have seen many games where cheating happened because clients trusted other clients for detrimental news.

The bottom line is that you shouldn't be surprised to see that the game does not look identical when running on two side-by-side computers. In fact, you can try this with many networked games, and you will notice that very few maintain absolute synchronization.

The Game Classes

If you haven't already done so, load up the *SpacePirates* project in the Chapter14 Zip file on the CD. If you look at the project space, you might notice the presence of more files than usual. The game project now uses two new classes: one for sound management and the other for 3D object management.

The Sound Classes

As I mentioned earlier, the `SoundSystem` class loads and plays the `GameSound` class objects. I created these two wrappers to help manage multiple sounds within a game. The first thing you need to look at is the header file, SoundSystem.h.

The SoundSystem Header File

The first part of the sound header file deals with including required headers, setting up error codes, and defining the `GameSound` class. Here is the code to do all of this.

```
#include <dmusici.h>
#include <dsound.h>
#include <dshow.h>

#define SOUNDERROR_MUSICLOADER              0x85000001
#define SOUNDERROR_MUSICPERFORMANCE         0x85000002
#define SOUNDERROR_INITAUDIO                0x85000003
#define SOUNDERROR_PATH                     0x85000004
#define SOUNDERROR_VOLUME                   0x85000005
```

```
#define SOUNDERROR_LOAD                     0x85000006
#define SOUNDERROR_DOWNLOAD                 0x85000007
#define SOUNDERROR_NOSEGMENT                0x85000008
#define SOUNDERROR_PLAYFAIL                 0x85000009

class GameSound
{
    public:
        IDirectMusicSegment8                 *pSound;
        IDirectMusicPerformance8             *pPerformance;
        ~GameSound();
        GameSound();
};
```

The GameSound Class Header

The GameSound class only holds actual sound data. If you remember the chapter on sound, then you might recall that sounds are held in IDirectMusicSegment8 interfaces. This is why the first variable within the game sound class is the pointer for an IDirectMusicSegment8 interface.

The next variable is a pointer for an IDirectMusicPerformance8 interface. You will not actually create a performance object, but you do need a pointer to the performance interface that plays the sound. The reason you need this pointer is that you need to call the sound's Unload() function when the sound is released. If you don't keep track of the performance object, then your program will leak memory.

Besides the two interface pointers, the only two items of note for the class are the constructor and destructor functions. Be sure to look at Figure 14.6 to see the layout of the GameSound class.

The SoundSystem Class Header

The SoundSystem class isn't much more complicated then the GameSound class. This class's main purpose is to initialize the sound system and load and play sounds. Here is the class header code.

```
class SoundSystem
{
    private:

    public:
        HWND                               hWnd;
        IDirectMusicLoader8             *pLoader;
        IDirectMusicPerformance8        *pPerformance;

        // Functions
        SoundSystem();
        ~SoundSystem();
        HRESULT     hrInitSoundSystem(void);
        HRESULT     hrLoadSound(char *szname,GameSound *gs);
        HRESULT     hrPlaySound(GameSound *gs);
};
```

The first variable, `hWnd`, holds the handle to the window that will play the sounds. This is necessary for DirectSound to work properly. The second variable, `pLoader`, points to the loader interface. The loader will load all game sounds. The last variable, `pPerformance`, points to the performance interface that will play the actual sounds.

The first two functions deal with the constructor and destructor for the class. The next function, `hrInitSoundSystem()`, must be called only once to initialize DirectAudio. This function creates the loader and performance interfaces. The next function, `hrLoadSound()`, loads the passed-in filename and returns a pointer to the new sound data. The last function, `hrPlaySound()`, plays the passed-in game sound object. To see how the variables and functions correspond to one another, check out Figure 14.7.

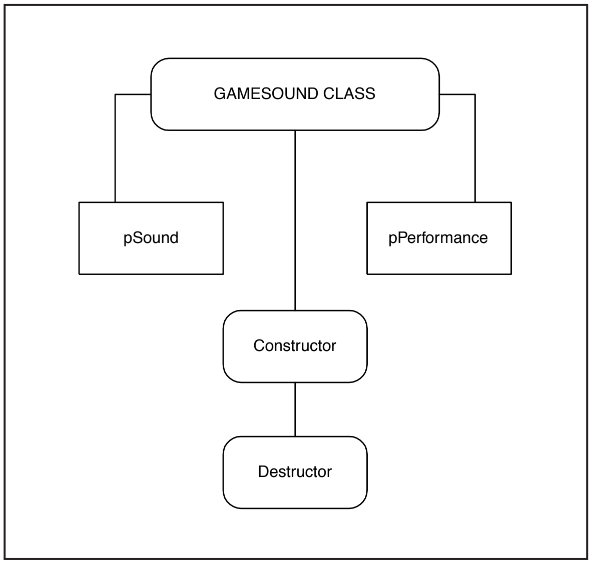


Figure 14.6

The `GameSound` *class architecture*

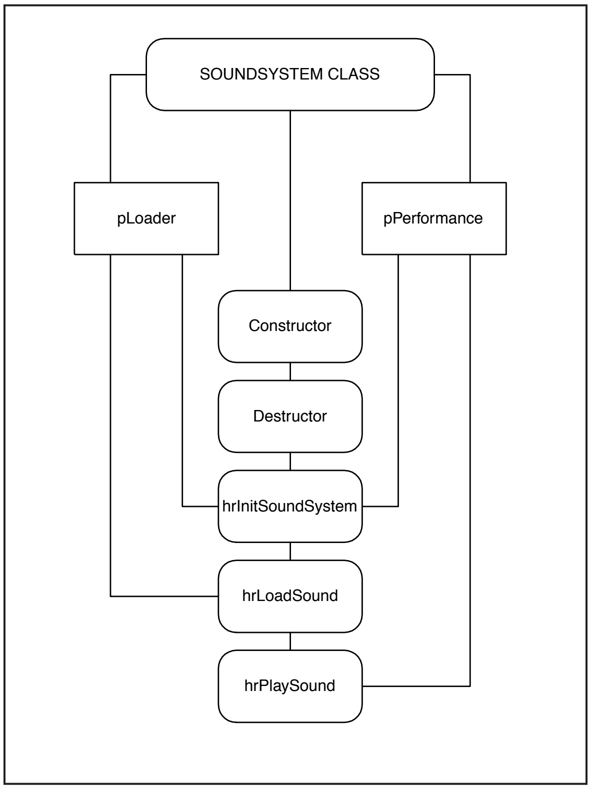


Figure 14.7

The `SoundSystem` *class architecture*

The SoundSystem Class Functions

The first two functions, the constructor and the destructor, don't do very much except set the interfaces to `NULL` and release the interfaces upon shutdown. Here is the code for the functions.

```
// Destructor
SoundSystem::~SoundSystem()
{
    SAFE_RELEASE(pLoader);
    SAFE_RELEASE(pPerformance);
}
// Constructor
SoundSystem::SoundSystem()
{
    pLoader = NULL;
    pPerformance = NULL;
}
```

You can see the constructor setting pLoader and pPerformance to NULL. You do this so the destructor does not accidentally destroy an un-initialized interface. The destructor completes a SAFE_RELEASE() on each interface. The SAFE_RELEASE() function, which is part of the dxutil.cpp file, checks whether the interface is NULL before calling Release().

The SoundSystem::hrInitSoundSystem() Function

The SoundSystem::hrInitSoundSystem() function initializes DirectAudio. If you look at the initialization code in Chapter 13, "Now Showing, DirectInput and DirectAudio," you'll see that the initialization code here is identical. Here is the code used for *SpacePirates*.

```
HRESULT SoundSystem::hrInitSoundSystem(void)
{
    HRESULT                     hResult;
    IDirectMusicAudioPath8      *path;

    // Initialize COM
    CoInitialize(NULL);
    // Create the loader
    if( FAILED( hResult = CoCreateInstance(CLSID_DirectMusicLoader, NULL,
        CLSCTX_INPROC, IID_IDirectMusicLoader8,
         (void**)&pLoader) )) {
        return(SOUNDERROR_MUSICLOADER);
    }
    // Create the performance
    if( FAILED( hResult = CoCreateInstance(CLSID_DirectMusicPerformance,
     NULL,
        CLSCTX_INPROC, IID_IDirectMusicPerformance8,
         (void**)&pPerformance ))) {
        return(SOUNDERROR_MUSICPERFORMANCE);
    }
    // Initialize the audio
    if( FAILED( hResult = pPerformance->InitAudio(
        NULL,
        NULL,
        hWnd,
        DMUS_APATH_DYNAMIC_STEREO,
        4,
        DMUS_AUDIOF_ALL,
        NULL
        ))) {
        return(SOUNDERROR_INITAUDIO);
    }
    // Get the default path
    if( FAILED( pPerformance->GetDefaultAudioPath( &path ) ) )
        return(SOUNDERROR_PATH);
    // Set the default volume
    if( FAILED( path->SetVolume(0,0) ) )
        return(SOUNDERROR_VOLUME);
    return(S_OK);
}
```

Because everything that happens here is old hat by now, I'll just list the actions that take place.

1. The COM is initialized.
2. The loader is created.
3. The performance is created.
4. The audio is initialized.
5. The default path is retrieved.
6. The sound volume is set.

The SoundSystem::hrLoadSound() Function

Now that the initialization of the audio system is out of the way, you can concentrate on the function to load sound data—the `SoundSystem::hrLoadSound()` function. The function is actually similar to the code in Chapter 13, except this function is passed a `GameSound` object pointer as well as a filename. Here is the prototype.

```
HRESULT SoundSystem::GameSound(char *szname,GameSound *gs);
```

The first parameter passes in the filename for the sound resource you want to load. The second parameter passes in a `GameSound` pointer that will receive the newly loaded data.

Now that you know the prototype, here is the code for the function itself.

```
HRESULT SoundSystem::hrLoadSound(char *szname,GameSound *gs)
{
    WCHAR     wstrFileName[256];
    // Convert the filename to wide character format
    DXUtil_ConvertGenericStringToWide( wstrFileName, szname );
    // Make sure audio is initialized
    if( !pLoader )
        return(SOUNDERROR_MUSICLOADER);
    if( !pPerformance )
        return(SOUNDERROR_MUSICPERFORMANCE);
    // Clean up sound if it already exists
    if( gs->pSound ) {
        gs->pSound->Unload( pPerformance );
        gs->pSound->Release();
        gs->pSound = NULL;
    }
    // Load the sound from a file
    if ( FAILED(pLoader->LoadObjectFromFile(
        CLSID_DirectMusicSegment,
        IID_IDirectMusicSegment8,
        wstrFileName,
         (LPVOID*) &gs->pSound
        )))
     {
        return(SOUNDERROR_LOAD);
    }
    gs->pPerformance = pPerformance;
    // Download the data
    if ( FAILED ( gs->pSound->Download( pPerformance ) ) ) {
        return(SOUNDERROR_DOWNLOAD);
    }
    return(S_OK);
}
```

Look through the code for a minute, and you'll see how Figure 14.8 corresponds to the order of events.

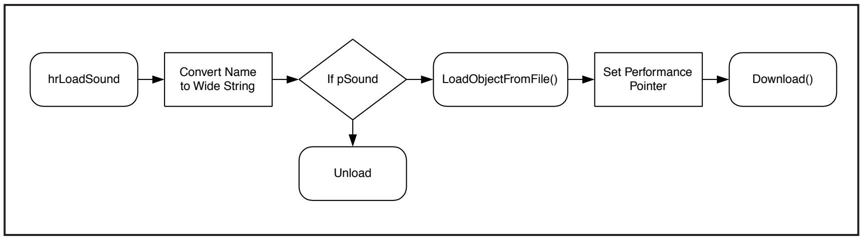


Figure 14.8

The `hrLoadSound()` *function*

The only thing new here is the code that sets the GameSound's internal performance pointer to the performance interface created in the SoundSystem class. By doing this, the GameSound object can unload the downloaded sound upon program exit.

The SoundSystem::hrPlaySound() Function

The only function left is the SoundSystem::hrPlaySound() function that plays loaded sounds. This function is also similar to the code used in Chapter 13, except this function plays a passed-in GameSound instead of a global segment. Here is the code for the function.

```
HRESULT SoundSystem::hrPlaySound(GameSound *gs)
{
    // Make sure there is a performance object present
    if( !pPerformance )
        return( SOUNDERROR_MUSICPERFORMANCE );
    if( !gs->pSound )
        return( SOUNDERROR_NOSEGMENT );
    // Play the sound segment
    if( FAILED ( pPerformance->PlaySegmentEx(
        gs->pSound,
        NULL,
        NULL,
        DMUS_SEGF_DEFAULT | DMUS_SEGF_SECONDARY,
        0,
        NULL,
        NULL,
        NULL
        )))
        return( SOUNDERROR_PLAYFAIL );
    return(S_OK);
}
```

The first thing the function does is make sure that the performance and segment interfaces are valid. If they are not valid, the function returns an error code. The next thing that happens is the loading of the actual sound data. The loaded data is stored in the GameSound's segment interface.

The GameSound Class Functions

Now that you have the sound system class out of the way, you can look at the GameSound class functions. The only two functions are the constructor and destructor, and they look like this:

```
GameSound::GameSound()
{
    pSound = NULL;
    pPerformance = NULL;
}
GameSound::~GameSound()
{
    if( pSound ) {
        if( pPerformance )
            pSound->Unload( pPerformance );
    }
    SAFE_RELEASE(pSound);
}
```

The constructor initializes the interface pointers to `NULL`. You do this to keep the destructor from illegally destroying interfaces. The destructor unloads the sound from the performance object and releases the loaded sound.

That's it for the sound classes. Go ahead and look through the SpacePirates.cpp code to see where and how the sound classes are called, to get a better idea of how the game sound works.

The 3D Object Class

The 3D object class encapsulates all of the data needed to load and manage an x-file object. When you don't encapsulate everything, you have to keep track of meshes, materials, textures, and buffers. The class keeps all of that out of the way and lets the programmer concentrate on loading and displaying models. The main functions of this class are to load and display the object.

CAUTION

Always unload sounds that have been downloaded into a performance interface. This keeps the sounds from leaking memory.

The Object3DClass Header

Load up the 3DObjectClass.h header file so you can see how the class is laid out. Here is the source code.

```
#include <d3dx8.h>
#include <stdio.h>
#include "dxutil.h"
#define RADIAN_TO_DEGREES 57.29577951308232286465f
```

```
class Object3DClass
{
    private:

    public:
        LPD3DXMESH                      pMesh;
        D3DMATERIAL8                    *pMeshMaterials;
        LPDIRECT3DTEXTURE8              *pMeshTextures;
        DWORD                           dwNumMaterials;
        LPD3DXBUFFER                    pD3DXMtrlBuffer;
        LPDIRECT3DDEVICE8               pd3dDevice;
        Object3DClass(void);
        ~Object3DClass(void);
        HRESULT hLoad(char *szName,LPDIRECT3DDEVICE8 p3d);
        void vCleanup(void);
        void vDisplayXYZ(float x, float y, float z, float rx=0.0f,
         float ry=0.0f, float rz=0.0f);
};
```

The first thing of note is the definition of radians to degrees that is used in the vDisplayXYZ() function to set up rotations. You might recall from previous chapters that rotations in 3D are measured in radians and not degrees.

The next part of the code deals with the class member variables and functions. The first five variables hold information about the loaded 3D object. The pd3dDevice pointer stores a pointer to the IDirect3DDevice8 interface used in the program. You need this interface to load and display the 3D object.

TIP

To find the equivalent radians for degrees, divide the degree value by 57.2957795130823228646 5.

The Member Variables

The pMesh variable stores the loaded mesh, which contains the geometry for the object. The pMeshMaterials variable stores any number of materials needed by the 3D object. Keep in mind that there can be multiple materials within one object. The pMeshTextures variable stores any number of textures needed by the 3D object. The same rules apply here as do with meshes.

The dwNumMaterials counter stores the number of materials used in the object. This is useful at render time so that you know how many materials to display. You use the pD3DXMtrlBuffer variable as a temporary material buffer when loading the 3D data.

The pd3dDevice pointer points to the 3D interface used by the system. The object class does not actually create this interface, which gets passed in the hLoad() function.

The Member Functions

The first function, hLoad(), loads the 3D data. The second function, vCleanup(), clears any loaded 3D data. This function is called within the object's destructor. The last function, vDisplayXYZ(), displays the 3D object at the given x, y, and z coordinates. You can also specify the rotation of the object.

Okay, you have seen the nickel tour of the variables and functions; now it is time to see the actual code at work. Before you go, look at Figure 14.9 to see how the functions and variables correspond to one another.

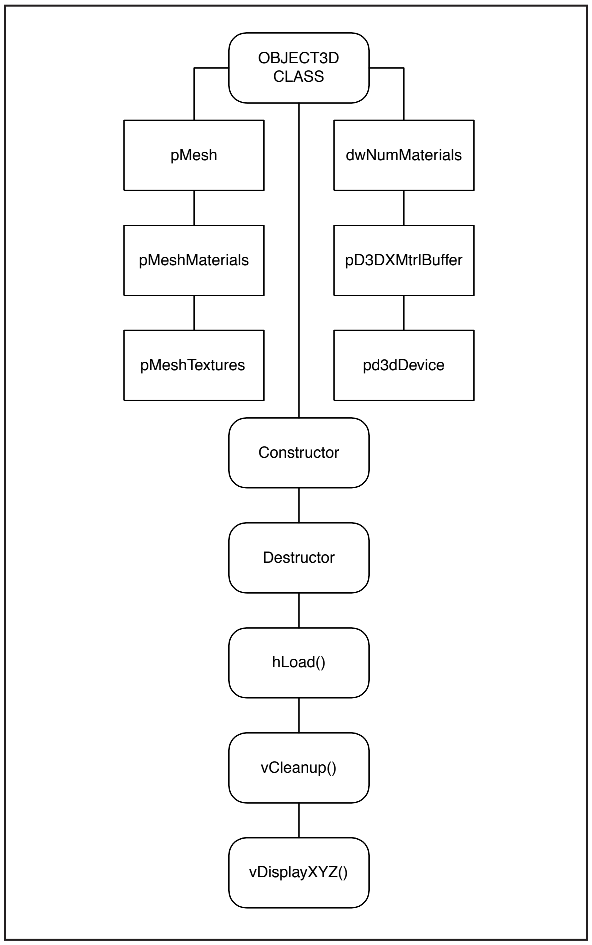


Figure 14.9

Layout of the Object3DClass

The Object3DClass Member Functions

All of the member functions for the `Object3DClass` appear in the Object3DClass.cpp file that is included in the *SpacePirates* project space. Most of the code in the functions is similar to the code in the XFileLoader and XFileAnimator projects in Chapter 12, "Inside Direct3D Rendering."

The Constructor and Destructor

As with any class, you normally have a constructor and a destructor. Remember that the constructor is called when the object is created, and the destructor is called when the object is deleted or leaves program scope. Here is the code for the constructor and destructor functions.

```
#include "Object3DClass.h"
// Constructor
Object3DClass::Object3DClass(void)
{
    pMesh = NULL;
    pMeshMaterials = NULL;
    pMeshTextures = NULL;
    pD3DXMtrlBuffer = NULL;
    pd3dDevice = NULL;
}
// Destructor
Object3DClass::~Object3DClass(void)
{
    vCleanup();
}
```

There isn't much to the constructor because all it does is set all of the pointers for the class to `NULL`. This step is necessary so that the destructor does not accidentally delete an invalid pointer.

The Object3DClass::hLoad() Function

The `Object3DClass::hLoad()` function loads the geometry, materials, and textures for the 3D object. The code is similar to that used in the XFileLoader project. Here is the code in all of its glory.

```
HRESULT Object3DClass::hLoad(char *szName,LPDIRECT3DDEVICE8 p3d)
{
    HRESULT              hResult = S_OK;
```

```
int                     i;
char                    szFullPath[256];
D3DXMATERIAL*           d3dxMaterials;

// Set the IDirect3DDevice8 pointer to the internal one
pd3dDevice = p3d;
// Initialize number of materials to 0
dwNumMaterials = 0;
// Load login geometry
if( FAILED( D3DXLoadMeshFromX(
    szName,
    D3DXMESH_SYSTEMMEM,
    pd3dDevice,
    NULL,
    &pD3DXMtrlBuffer,
    &dwNumMaterials,
    &pMesh ) ) ) {
    return(-1);
}
// Set temp pointer for easier-to-read code
d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
// Create the materials and textures
pMeshMaterials = new D3DMATERIAL8[dwNumMaterials];
pMeshTextures = new LPDIRECT3DTEXTURE8[dwNumMaterials];
// Load in material information and textures
for( i = 0; i < (signed)dwNumMaterials ; i++ ) {
    // Copy the material
    pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
    // Set the ambient color for the material (D3DX does not do this)
    pMeshMaterials[i].Ambient = pMeshMaterials[i].Diffuse;
    sprintf(szFullPath,"Data//3DObjects//%s",
    d3dxMaterials[i].pTextureFilename);
    // If the texture did not load, set it to NULL
    if( FAILED( D3DXCreateTextureFromFile(
        pd3dDevice,
        szFullPath,
        &pMeshTextures[i] ) ) ) {
        // Delete the texture from memory
        delete pMeshTextures[i];
        pMeshTextures[i] = NULL;
        }
    }
    SAFE_RELEASE(pD3DXMtrlBuffer);
    return( hResult );
}
```

The first part of the code sets the internal `IDirect3DDevice8` interface pointer to that of the one passed in. This is necessary so that you can load and display the object. The next thing that happens is the loading of the actual data through the `D3DXLoadMeshFromX()` function. Once the mesh is loaded, the function allocates memory for the materials and textures. The code then loops through the number of materials present and attempts to load a texture for each one. If a texture is present, it is loaded; if not, the memory allocated for the texture is deleted. Lastly, the code releases the temporary material buffer. That's it in a nutshell; to see the steps illustrated, refer to Figure 14.10.

The Object3DClass::vDisplayXYZ() Function

The `Object3DClass::vDisplayXYZ()` function renders the 3D object based on its position and rotation within the world. Here is the prototype.

```
void Object3DClass::vDisplayXYZ(
    float x,
    float y,
    float z,
    float rx,
    float ry,
    float rz
);
```

The first three variables pass in the x, y, and z coordinates of the object to render. The last three variables pass in the x, y, and z rotation of the 3D object. Unlike

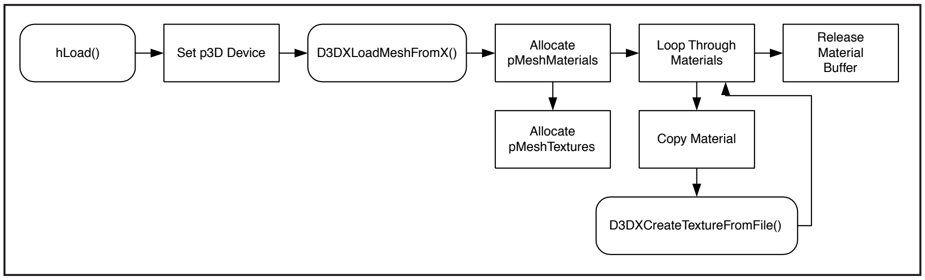


Figure 14.10

The `hLoad()` *function*

most rotation functions, this one uses degrees for the values. All degree values are translated to radians within the function itself. Here is the code.

```
void Object3DClass::vDisplayXYZ(float x, float y, float z, float rx,
 float ry, float rz)
{
    D3DXMATRIX     matWorld,matRotation,matTranslation,matScale;
    int                       mat;

    // Set default translation
    D3DXMatrixIdentity( &matTranslation );
    // Rotate on x
    D3DXMatrixRotationX( &matRotation, rx/RADIAN_TO_DEGREES );
    D3DXMatrixMultiply(&matTranslation,&matRotation,&matTranslation);
    // Rotate on y
    D3DXMatrixRotationY( &matRotation, ry/RADIAN_TO_DEGREES );
    D3DXMatrixMultiply(&matTranslation,&matRotation,&matTranslation);
    // Rotate on z
    D3DXMatrixRotationZ( &matRotation, rz/RADIAN_TO_DEGREES );
    D3DXMatrixMultiply(&matTranslation,&matRotation,&matTranslation);
    // Move to x,y,z coordinates
    matTranslation._41 = x;
    matTranslation._42 = y;
    matTranslation._43 = z;
    // Set the matrix
    pd3dDevice->SetTransform( D3DTS_WORLD, &matTranslation );
    // Display each material
    for( mat = 0 ; mat < (signed)dwNumMaterials ; mat++ ) {
        pd3dDevice->SetMaterial( &pMeshMaterials[mat] );
        // Activate the appropriate texture
        pd3dDevice->SetTexture( 0, pMeshTextures[mat] );
        // Draw the mesh subset
        pMesh->DrawSubset( mat );
        // Turn off the texture
        pd3dDevice->SetTexture( 0, NULL );
    }
}
```

Okay, this type of function hasn't appeared in previous examples, so I'll discuss it a bit more than usual.

The Identity Matrix

Whenever you want to place a 3D object into a scene, you need to set up its identity matrix. An identity matrix places the object at the origin of the world with no rotations. In this function, you use the `D3DXMatrixIdentity()` helper function to create a base identity matrix.

Refer to Figure 14.11 to see how the 3D robot looks with just an identity matrix. Notice how the object is set to the center of the world with no rotation. This is the default setting. Now, if you want to rotate the object, you need to set up rotation matrices. This is where the next chunk of code comes into play.

TIP

The origin of a 3D world is the (0,0,0) position. This means that x = 0, y = 0, and z = 0.

The Rotation Matrices

For each of the three rotations (x, y, and z), you need to create a rotation matrix. The code uses the `D3DXMatrixRotationX()`, `D3DXMatrixRotationY()`, and `D3DXMatrixRotationZ()` helper functions to set the rotation of the object. Each time the rotation is set, you have to multiply the rotation matrix by the current translation matrix, which makes the effects of rotation cumulative. Now look at Figure 14.12 to see the robot with all three rotations in place.

Now that you have the rotation matrices in place, you need to translate the object to its end destination.

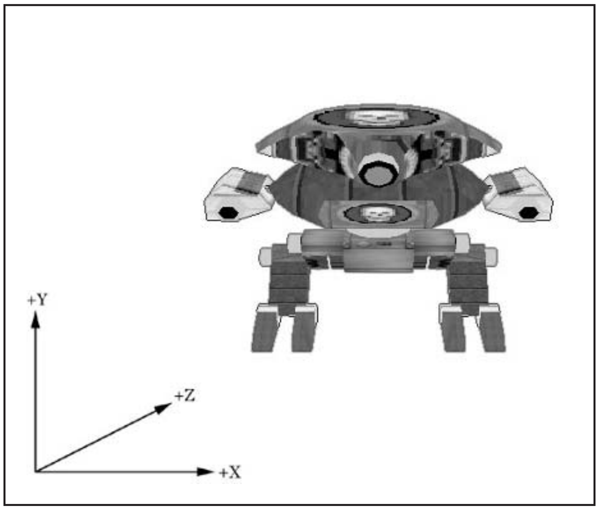


Figure 14.11

3D object with identity matrix only

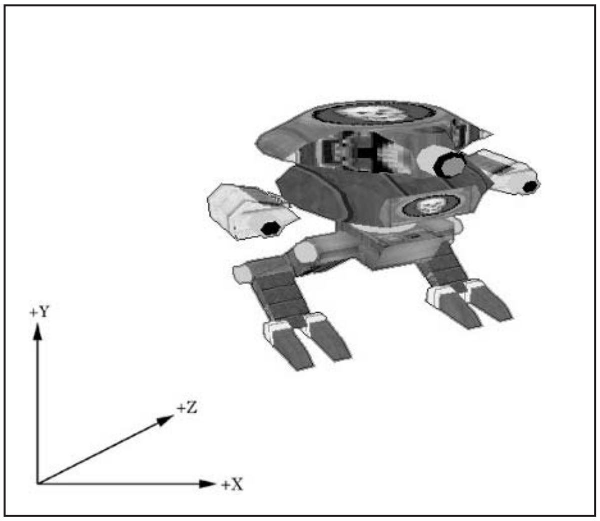


Figure 14.12

3D object with rotation matrices applied

Setting the Translation

To move the 3D object to its end destination, you need to update the matrix with the three coordinate values. You do this by changing the _41, _42, and _43 members of the current translation matrix. By doing this, you set the location for the 3D object to be rendered. To see how this has affected the robot in the previous figures, check out Figure 14.13.

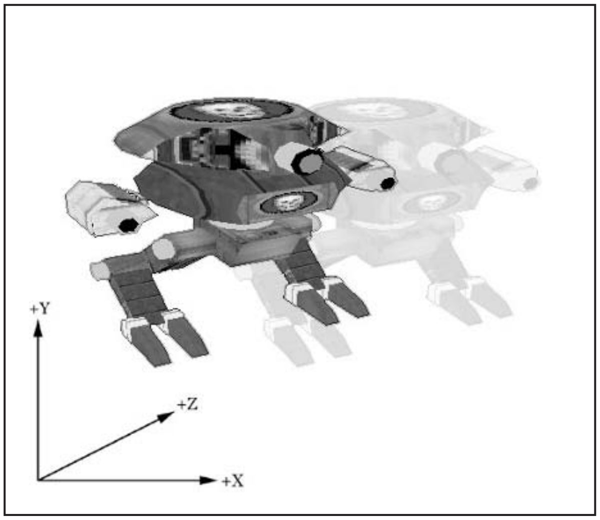


Figure 14.13

3D object with rotation and translation

Notice in Figure 14.13 how the object has been rotated and translated to its desired location. You can play around with changing the order of rotation and translation; you might be surprised by the results!

Rendering the Object

Now that the object is in its current rotation and translation, you can go ahead and lock in the transformation using the familiar `SetTransform()` function. For this function to operate as you need it to, you pass in `D3DTS_WORLD` as the matrix to which to apply the transformation. This sets the world translation for all future renderings. Now the object is in place; it's time to render it!

The actual rendering of the object happens as it did in the XFileLoader and XFileAnimator projects. Basically, you loop through each material, set the material, set the texture, and then draw the material. After you are done with each material, you set the texture to `0` and then continue the loop.

Guess what: You are done with the display function, so you can move on to the cleanup phase.

The Object3DClass::vCleanup() Function

Whenever you delete an `Object3DClass` or leave program scope with one in memory, the program calls the `vCleanup()` function. The purpose of this function is to release all geometry, materials, and textures that have been loaded. Here is the code.

```
void Object3DClass::vCleanup(void)
{
    SAFE_RELEASE(pMesh);
    SAFE_DELETE_ARRAY(pMeshMaterials)
    SAFE_DELETE_ARRAY(pMeshTextures);
    SAFE_RELEASE(pD3DXMtrlBuffer);
}
```

To free the memory, I use the `SAFE_RELEASE()` and `SAFE_DELETE_ARRAY()` helper macros in the dxutil files. Because you can have more than one material or texture loaded, you need to use the array version of the functions when releasing these pointers.

Go ahead and take a break now; maybe go play a game of *Unreal Tournament* or something. Hey, you know what? That sounds like a good idea. I will see you back here in 20.

Space Pirates: The Game Architecture

All of the classes are out of the way now, so you can go ahead and load up the SpacePirates.cpp file to see the main body of source code. It might look a little intimidating at first, but don't worry too much; I'll cover each piece. Just looking through the code doesn't give you a very good picture of what the application is doing. In Figure 14.14, you can see the layout for the entire program.

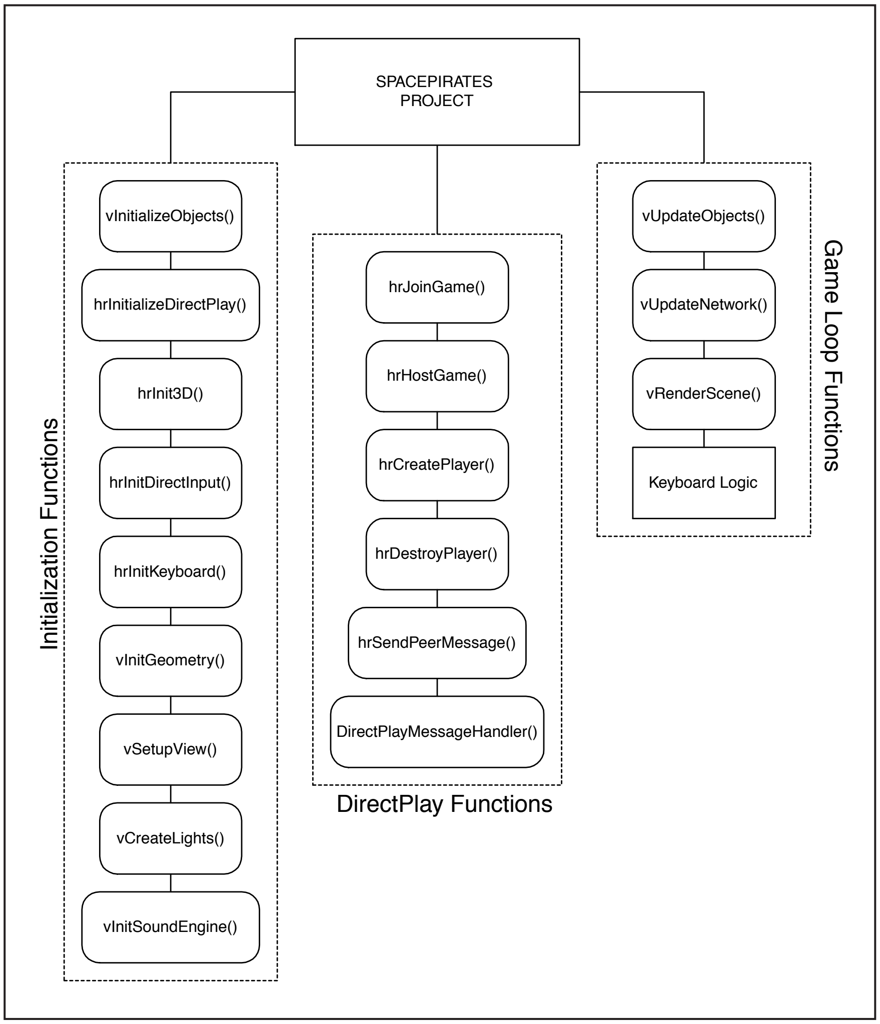


Figure 14.14

Layout of the SpacePirates.cpp file

The WinMain() Function

As with all Windows programs, the first thing you normally do is create the window in the `WinMain()` function. Check out the function and you'll see that I follow the usual steps in setting up the window. In this particular `WinMain()` function, you accomplish several initialization steps, each one through a function. Here is a list of the initialization functions the `WinMain()` function calls.

- `vInitializeObjects()`
- `hrInitializeDirectPlay()`
- `hrInit3D()`
- `hrInitDirectInput()`
- `hrInitKeyboard()`
- `vInitGeometry()`
- `vSetupView()`
- `vCreateLights()`
- `vInitSoundEngine()`

The vInitializeObjects() Function

The first function called by `WinMain()` sets up some of the global data structures. This is necessary to clear out the object structures for later use. Here is the code used.

```
void vInitializeObjects(void)
{
      int i;
      for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
            PlayerInfo[i].bActive = 0;
      }
      for( i = 0 ; i < MAX_BULLETS ; i++ ) {
            Bullet[i].bActive = 0;
      }
      for( i = 0 ; i < MAX_EXPLOSIONS ; i++ ) {
            Explosion[i].bActive = 0;
      }
      for( i = 0 ; i < MAX_TEXTURES ; i++ ) {
            g_pTexture[i] = NULL;
      }
      g_GameSounds.gsExplosion = NULL;
      g_GameSounds.gsFire = NULL;
}
```

The first thing initialized is the PlayerInfo array. The PlayerInfo array stores all the information about each player in the game. When new players join the game, they are added to the PlayerInfo list. The game keeps track of which player slots have been used with the bActive flag. By setting each bActive flag to 0, you turn off all player slots.

The next two arrays to get initialized, Bullet[] and Explosion[], keep track of the bullets and explosions in the game. By setting the bActive flag for each one to 0, you turn them off. This ensures that all of the bullets and explosions are inactive when the game first starts.

The next array to get set up is the g_pTexture[] array. This array stores all of the textures used in the game that do not belong to x-objects. It is important to set them to NULL so that the vCleanup() function does not try to remove non-initialized textures.

The last two variables to be set up, g_GameSounds.gsExplosion and g_GameSounds.gsFire, are used for the in-game sounds. They are loaded later.

The hrInitializeDirectPlay() Function

Wow, it's been a little while since you've seen this function, eh? The hrInitializeDirectPlay() function in *SpacePirates* sets up everything needed for DirectPlay. You might remember from the peer-to-peer example in Chapter 9, "Chatting with DirectPlay," that a few objects must be in place for DirectPlay to work, including

- COM
- IDirectPlay8Peer interface
- DirectPlay message handler
- IDirectPlay8Address interface for the device
- IDirectPlay8Address interface for the host
- Completion event

The most important of these is initializing the COM interface, a required step when you create the IDirectPlay8Peer interface. Without COM, you cannot use DirectPlay. Take a look at the hrInitializeDirectPlay() function in the code and follow along with me.

```
HRESULT hrInitializeDirectPlay( HWND hWindow )
{
    HRESULT     hReturn;
```

```
// Initialize COM
hReturn = CoInitialize( NULL );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Error Initializing COM",
     "DirectPlay Error", MB_ICONERROR );
    return hReturn;
}
// Initialize critical sections for multithreading
InitializeCriticalSection( &g_csModifyPlayer );
// Create IDirectPlay8Peer object
if( FAILED( hReturn = CoCreateInstance( CLSID_DirectPlay8Peer,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_IDirectPlay8Peer,
     (LPVOID*) &g_pDP ) ) )
MessageBox( hWindow, "Can't Create DPlayPeer", "DirectPlay Error",
 MB_ICONERROR );
// Init IDirectPlay8Peer message handler
if( FAILED( hReturn = g_pDP->Initialize( NULL,
 DirectPlayMessageHandler, 0 ) ) ) {
    MessageBox( hWindow, "Failed to Message Handler",
                               "DirectPlay Error", MB_ICONERROR );
    return -1;
}
// Create a device address
hReturn = CoCreateInstance( CLSID_DirectPlay8Address, NULL,
                    CLSCTX_INPROC_SERVER, IID_IDirectPlay8Address,
                    (LPVOID*) &g_pDeviceAddress );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Failed to Create Device",
                               "CoCreateInstance()", MB_ICONERROR );
    return -1;
}
// Set our service provider to TCP/IP
if( FAILED( hReturn = g_pDeviceAddress->SetSP(
 &CLSID_DP8SP_TCPIP ) ) ) {
    MessageBox( hWindow, "Failed to SetSP() for Device Address",
                               "Invalid Param", MB_ICONERROR );
    return -1;
    }
    // Create a host address
    hReturn = CoCreateInstance( CLSID_DirectPlay8Address,
                   NULL,CLSCTX_INPROC_SERVER,
                   IID_IDirectPlay8Address, (LPVOID*) &g_pHostAddress );
    if( FAILED(hReturn) ) {
        MessageBox( hWindow, "Failed to Create Host Address()",
                              "Invalid Param", MB_ICONERROR );
        return -1;
    }
    // Set the host address to TCP/IP
    if( FAILED( hReturn = g_pHostAddress->SetSP( &CLSID_DP8SP_TCPIP ) ) ) {
        MessageBox( hWindow, "Failed to SetSP() for Host Address",
                              "Invalid Param", MB_ICONERROR );
        return -1;
    }
    // Create connection complete event for later use
    g_hConnectCompleteEvent = CreateEvent( NULL, FALSE, FALSE, NULL );
    return S_OK;
}
```

The first thing that happens is the initialization of COM with the `CoInitialize()` function. The next thing to get set up is the critical section used to keep track of player data in the game. This step is necessary to make sure that the code does not run over itself when updating the global player data.

Now that COM is initialized, you can create the `IDirectPlay8Peer` interface that is needed for communication, using the `CoCreateInstance()` function.

Once you have the `IDirectPlay8Peer` interface, you can go ahead and set the default DirectPlay message handler to receive all game communication. Remember that this function is called whenever a packet arrives. For communication, *SpacePirates* uses the `DirectPlayMessageHandler()` function. If you look at the handler, you'll see that it has the usual suspects for creating players, destroying players, and processing game packets.

CAUTION

SpacePirates **will not initialize if you do not have an active connection to the Internet. If you use dial-up, be sure to connect to the Internet before launching the game.**

You might recall that you need DirectPlay addresses to store the information necessary to connect to the host or to join a game. The address stores information about the player and the connection. For each address, the program sets the service provider to be TCP/IP.

Once all of the addresses and service providers are set, the `g_hConnectCompleteEvent` event is set up. This event signals when the player has successfully connected to the host. Well, that's it for initialization. None of it should have come as a surprise to you, considering it is mostly copied straight from previous examples. To see how the functions relate to their global variable counterparts, check out Figure 14.15.

The hrInit3D() Function

The `hrInit3D()` function is where the 3D initialization magic happens. The function sets up the video device to display the wonderful graphics. Okay, maybe they aren't that wonderful in this game, but they don't stink either! In the previous 3D sample

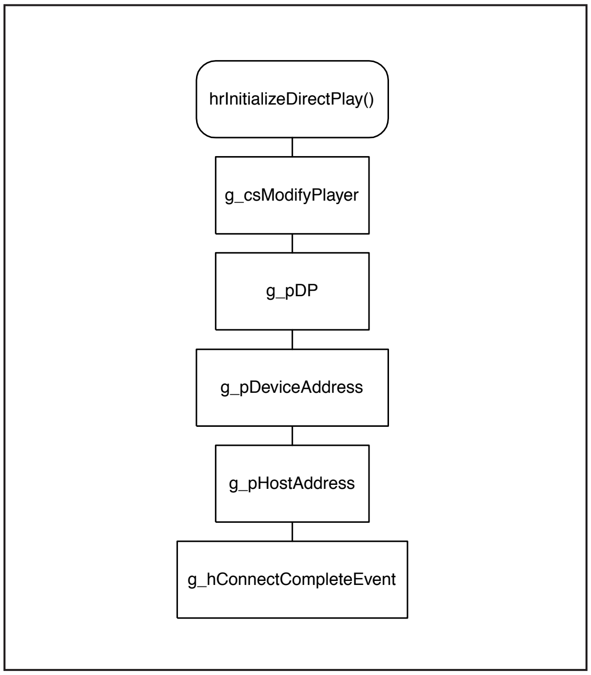


Figure 14.15

Global variable usage in the `hrInitializeDirectPlay()` *function* programs, I only showed you how to force the 3D graphics system to set the display mode. I performed no actual checking of the hardware for the best possible mode. I did not show you how to do full-screen graphics either, and we all know that real games use full-screen. Well, the pattern stops with *SpacePirates*: I'll now show you how to enumerate all available graphics cards and their modes. It does require quite a bit more work to get things done, but your application will be better for it. Here is the code for the new and improved `hrInit3D()` function.

```
HRESULT hrInit3D( HWND hWnd )
{
    D3DPRESENT_PARAMETERS     d3dpp;
    HRESULT     hResult;
    RECT                rcWindowBounds;
    // Saved window bounds for mode switches
    RECT                rcWindowClient;
    // Saved client area size for mode switches
    D3DAdapterInfo      *pAdapterInfo = &g_Adapters[g_dwAdapter];
    D3DDeviceInfo       *pDeviceInfo  = &pAdapterInfo->
    devices[pAdapterInfo->dwCurrentDevice];
    D3DModeInfo         *pModeInfo    = &pDeviceInfo->
    modes[pDeviceInfo->dwCurrentMode];

    // Create the D3D object
    if( NULL == ( pD3D = Direct3DCreate8( D3D_SDK_VERSION ) ) ) {
        MessageBox( hWnd, "Direct3D Error",
         "Unable to initialize Direct3D, please check version.",
         MB_ICONERROR );
        return(E_FAIL);
    }
    // Build a list of available devices
    hResult = hrBuildDeviceList(hWnd);
    if( hResult == VIDEODEVICEERROR_NOCOMPATIBLE ) {
        MessageBox( hWnd, "Adapter Error",
            "No compatible 3D adapters found.", MB_ICONERROR );
        return(E_FAIL);
    }
    else if( hResult == VIDEODEVICEERROR_NOWINDOWED ) {
        MessageBox( hWnd, "Adapter Error", "No available
        windowable 3D adapter found, please switch to 16
```

```
         or 32 bit mode.", MB_ICONERROR );
        return(E_FAIL);
    }
    // Get the size of the window
    GetWindowRect( hWnd, &rcWindowBounds );
    GetClientRect( hWnd, &rcWindowClient );
    // Set to full-screen
    if( !g_bWindowed )
        SetWindowLong( hWnd, GWL_STYLE, WS_POPUP|WS_SYSMENU|WS_VISIBLE );
    // Set temp pointers
    pAdapterInfo = &g_Adapters[g_dwAdapter];
    pDeviceInfo  = &pAdapterInfo->devices[pAdapterInfo->dwCurrentDevice];
    pModeInfo    = &pDeviceInfo->modes[pDeviceInfo->dwCurrentMode];
    // Set up the presentation parameters
    ZeroMemory( &d3dpp, sizeof(d3dpp) );
    d3dpp.Windowed                    = pDeviceInfo->bWindowed;
    d3dpp.BackBufferCount             = 1;
    d3dpp.MultiSampleType             = pDeviceInfo->MultiSampleType;
    d3dpp.SwapEffect                  = D3DSWAPEFFECT_FLIP;
    d3dpp.EnableAutoDepthStencil      = g_bUseDepthBuffer;
    d3dpp.AutoDepthStencilFormat      = pModeInfo->DepthStencilFormat;
    d3dpp.hDeviceWindow               = hWnd;
    // Run this code if in windowed mode
    if( g_bWindowed ) {
        d3dpp.BackBufferWidth   = (rcWindowClient.right -
         rcWindowClient.left);
        d3dpp.BackBufferHeight  = (rcWindowClient.bottom -
         rcWindowClient.top);
        d3dpp.BackBufferFormat  = pAdapterInfo->d3ddmDesktop.Format;
    }
    // Run this code if in full-screen mode
    else {
        d3dpp.BackBufferWidth   = pModeInfo->Width;
        d3dpp.BackBufferHeight  = pModeInfo->Height;
        d3dpp.BackBufferFormat  = pModeInfo->Format;
    }
    // Create the device
    hResult = pD3D->CreateDevice( g_dwAdapter,
                        pDeviceInfo->DeviceType, hWnd,
                        pModeInfo->dwBehavior, &d3dpp, &pd3dDevice );
    if( !SUCCEEDED(hResult) ) {
        MessageBox( hWnd, "Device Error", "Failed to create
         display device!",
            MB_ICONERROR );
        return(E_FAIL);
    }
    // Turn on the z-buffer
    pd3dDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
    // Turn on ambient lighting
    pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0x00606060 );
    return S_OK;
}
```

The first part of the function creates the IDirect3D8 system object. The next item of interest is that call to the hrBuildDeviceList() function. This function is where the new part comes into play. It creates a list of compatible video devices and modes for the given application. Before I go further into it, let me finish talking about the init function.

Once the list of devices is created, the init function then checks whether the game is to be played in full-screen mode. If the game is to run in full-screen mode, the program updates the current window settings. The purpose of this is to turn off the controls for the window, such as the minimize, maximize, and exit buttons. All of this happens in the SetWindowLong() function provided by Windows.

The next part of the code sets up the presentation parameters for the display device. The difference here is that the initialization function uses the default device found in the hrBuildDeviceList() function to set up the parameters. This is why you see the pDeviceInfo and pModeInfo pointers.

The next section of the function sets up the back buffer. If the program is to run in windowed mode, the function uses the size of the client window for the back buffer. If the program is to run in full-screen mode, the function uses the actual size of the desired resolution for the back buffer.

The last thing to do is create the IDirect3DDevice8 interface using the presentation parameters with the CreateDevice() function. Take a look at Figure 14.16 to see the steps involved in the hrInit3D() function.

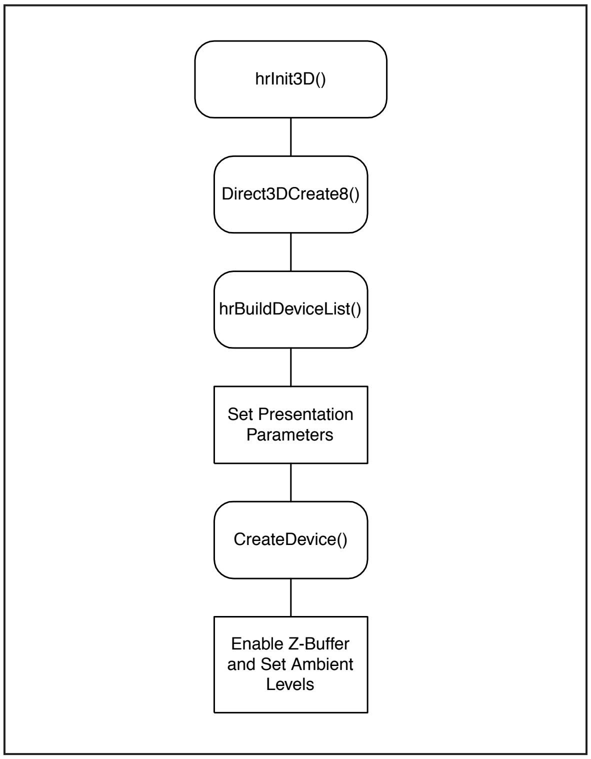

hrInit3D()
Direct3DCreate8()
hrBuildDeviceList()
Set Presentation Parameters
CreateDevice()
Enable Z-Buffer and Set Ambient Levels

Figure 14.16

The `hrInit3D()` *function*

The hrBuildDeviceList() Function

The `hrBuildDeviceList()` function is a pretty complicated function that creates a list of compatible adapters and devices, and their modes for a given application. It consists of the following subfunctions.

- `hrConfirmDevice()`
- `bFindDepthStencilFormat()`
- `bValidateFormat()`

At first glance, the `hrBuildDeviceList()` function might look intimidating. Your assumption is correct, in this case. There is nothing pretty about the process, but it is a necessary evil. Look at Figure 14.17 to see the flow of the `hrBuildDeviceList()` function.

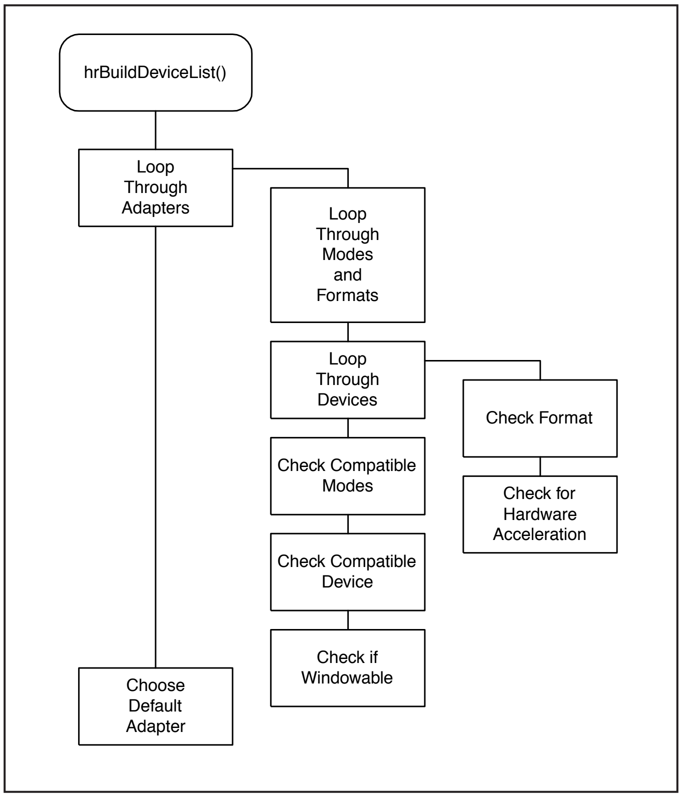


Figure 14.17

The `hrBuildDeviceList()` *function*

Enumerating the Adapters

You might be able to tell from the diagram that the first thing the function does is loop through all of the adapters in the system. This is necessary because some people have systems with multiple video cards. To figure out how many adapters are in a system, the program uses the `GetAdapterCount()` function that belongs to the `IDirect3D8` interface. This function returns the total number of adapters in a system.

At the beginning of each new adapter, the program grabs the desktop setting for the device. The setting is then stored in the adapter's data structure under the `d3ddmDesktop` variable. This is stored for later use when playing the game in windowed mode. You can see this in action in the following lines of code.

```
pAdapter  = &g_Adapters[g_dwNumAdapters];
pD3D->GetAdapterDisplayMode( iAdapter, &pAdapter->d3ddmDesktop );
```

```
pAdapter->dwNumDevices    = 0;
pAdapter->dwCurrentDevice = 0;
dwNumAdapterModes = pD3D->GetAdapterModeCount( iAdapter );
```

Enumerating the Video Modes

The last line of code mentioned here gets the number of video modes available for the adapter. As you can see, you find the number of modes by calling the `GetAdapterModeCount()` function. Once it knows the numbers of video modes, the program can loop through each mode looking for a compatible one.

In the video mode loop, the first function called is the `EnumAdapterModes()` function. This function finds out all available refresh rate settings for the particular mode. Some video mode refresh rates can greatly enhance performance, so this is a necessary step.

Checking the Video Mode Resolution

The first thing that happens after the refresh rates are enumerated is checking the mode's width and height. Because you don't want to add an incompatible mode to the list of available modes, you need to filter out any that are smaller in resolution than the desired game size. In the case of *SpacePirates*, the minimum resolution is 800 × 600. Here is the code.

```
if( (DisplayMode.Width < SCREEN_WIDTH) || (DisplayMode.Height < SCREEN_HEIGHT) )
    continue;
```

Adding Modes to the List

Once a mode has passed the resolution test, the program goes on to check whether the mode is new to the current list of compatible ones. If it is a new mode, then it is added to the list of compatible modes.

```
if( m == dwNumModes ) {
    modes[dwNumModes].Width          = DisplayMode.Width;
    modes[dwNumModes].Height         = DisplayMode.Height;
    modes[dwNumModes].Format         = DisplayMode.Format;
    modes[dwNumModes].RefreshRate    = 0;
    dwNumModes++;
    // Check if the mode's format already exists
    for( f = 0; f < dwNumFormats; f++ ) {
        if( DisplayMode.Format == formats[f] )
            break;
    }
    // If the format is new, add it to the list
    if( f == dwNumFormats )
        formats[dwNumFormats++] = DisplayMode.Format;
}
```

Within the process of adding the new mode to the list, the program also makes a check to see whether the format is new. If the format is new, it is added to the list of compatible formats.

Adding Compatible Devices

Now that the modes and formats are enumerated, it is time to check the video devices. This function checks two types of devices, hardware and software. If the player's computer has hardware acceleration, you want to take advantage of it. This is where device checking comes in handy.

At the beginning of the device-enumeration loop, each device is initialized with its name and default settings. Once that is accomplished, the function begins to loop through each of the formats found earlier. The first check is done with a call to the `CheckDeviceType()` function. This function returns whether the passed-in device and format will work together. If not, the function goes on to the next device.

Checking for Hardware Support in a Window

Once a device has proven that it can render in the checked format, the program makes a check to see whether the format and device can handle hardware acceleration in a window. This step is very important if the game is run in windowed mode. You would not want to select a hardware device only to find out that windowed mode will not work.

```
if( pDevice->DeviceType == D3DDEVTYPE_HAL ) {
    // This system has a HAL device
    bHALExists = TRUE;
    // Check if windowed mode is possible
    if( pDevice->d3dCaps.Caps2 & D3DCAPS2_CANRENDERWINDOWED ) {
        // HAL can run in a window for some mode
        bHALIsWindowedCompatible = TRUE;
        if( f == 0 ) {
            // HAL can run in a window for the current desktop mode
            bHALIsDesktopCompatible = TRUE;
        }
    }
}
```

To check whether the device supports hardware acceleration, the program checks the DeviceType member variable against the D3DDEVTYPE_HAL definition. If the check succeeds, then you know that the device is a hardware device. After that check is passed, the code then checks the capabilities of the device against the D3DCAPS2_CANRENDERWINDOWED definition. This tells the system whether the device supports hardware-accelerated window rendering. If all of the checks pass, then the bHALIsDesktopCompatible variable is set to True.

Checking for Hardware Transform and Lighting Support

The next few lines of code deal with checking for transform and lighting (T&L) support at the hardware level. Some video cards, such as the GeForce series from Nvidia, support hardware-accelerated vertex processing. By processing vertices in hardware, these cards can greatly increase the speed at which games run.

You perform the check for hardware T&L by looking at the card's device capabilities with the current mode and format. If the capabilities include the D3DDEVCAPS_HWTRANSFORMANDLIGHT flag, then you know that hardware T&L are supported.

Once hardware T&L are located, the function goes on to check whether the device is a pure device. If it is not a pure device, the function checks whether it supports hardware vertex processing. If the device does not support hardware vertex processing, the function checks whether it supports mixed vertex processing. You perform these checks by comparing the device capabilities to the D3DDEVCAPS_PUREDEVICE, D3DCREATE_HARDWARE_VERTEXPROCESSING, and D3DCREATE_MIXED_VERTEXPROCESSING flags. The actual comparison check is accomplished using the hrConfirmDevice() function defined later in the application. In *SpacePirates*, the hrConfirmDevice() function checks whether the video card can support alpha blending, which is required to do transparency with textures. In this game, the bullets and explosions need transparency, hence the requirement. Feel free to add your own requirements to the hrConfirmDevice() function.

Finding a Compatible Depth Buffer

Once the device is proven to support hardware or software, the code performs a loop check to see whether it has a compatible depth buffer. This is necessary if your game will use a z-buffer.

```
// Find a suitable depth/stencil buffer format for this device/format
if( bFormatConfirmed[f] && g_bUseDepthBuffer ) {
     if( !bFindDepthStencilFormat( 8, 0, iAdapter, pDevice->DeviceType,
         formats[f], &fmtDepthStencil[f] ) ) {
         bFormatConfirmed[f] = FALSE;
     }
}
```

To find out whether the device is compatible, you make a call to the `bFindDepthStencilFormat()` function. This function is defined in *SpacePirates* and checks for a compatible depth format with the passed parameters. In this case, *SpacePirates* requires that the user have at least an 8-bit depth buffer. This is where the first parameter applies. If you require a 16-bit buffer, then you pass in a `16` to the first parameter. As new depth formats come out with DirectX releases, you should add them to the `bFindDepthStencilFormat()` function. Hey, it's been a little while since you've seen an illustration. Look at Figure 14.18 to see how the device formats are checked.

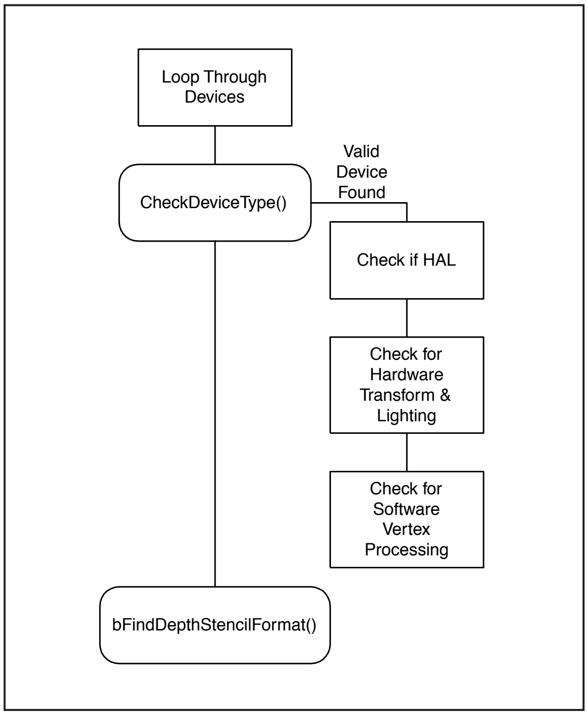

Loop Through Devices
CheckDeviceType()
Valid Device Found
Check if HAL
Check for Hardware Transform & Lighting
Check for Software Vertex Processing
bFindDepthStencilFormat()

Figure 14.18

Processing the device formats

Adding Device Modes to the List

Once all of the formats are checked for compatibility with a particular device, a loop moves through all formats and modes to see which ones work together. When a match is found, it is added to the permanent list of modes for the device.

```
for( m = 0; m < dwNumModes; m++ ) {
    for( f = 0; f < dwNumFormats; f++ ) {
        if( modes[m].Format == formats[f] ) {
            if( bFormatConfirmed[f] == TRUE ) {
                // Add this mode to the device's list of valid modes
                pDevice->modes[pDevice->dwNumModes].Width
                = modes[m].Width;
                pDevice->modes[pDevice->dwNumModes].Height
                = modes[m].Height;
                pDevice->modes[pDevice->dwNumModes].Format
                 = modes[m].Format;
                pDevice->modes[pDevice->dwNumModes].dwBehavior
                = dwBehavior[f];
                pDevice->modes[pDevice->dwNumModes].DepthStencilFormat
                 = fmtDepthStencil[f];
                pDevice->dwNumModes++;
                if( pDevice->DeviceType == D3DDEVTYPE_HAL )
                    bHALIsSampleCompatible = TRUE;
            }
        }
    }
}
```

As you can see in the source code, if the mode's format equals the format and the format has passed compatibility checks, then the mode is added to the device's list. The code performs a final check to see whether the device is hardware accelerated. It checks the `DeviceType` member against the `D3DDEVTYPE_HAL` definition.

Checking Whether a Valid Device Was Found

The next rather complicated-looking loop checks whether a compatible device mode was found to match the game's requirements. If one is found, the `bFoundVidMode` variable is set. This variable is used in a later check to see whether any compatible devices were located.

Checking for a Window-Compatible Device

The next check that comes along is a check to see whether the device can operate in a window.

```
// Check if device can work in a window
if( bFormatConfirmed[0] && (pDevice->d3dCaps.Caps2 &
 D3DCAPS2_CANRENDERWINDOWED) ) {
    pDevice->bCanDoWindowed  = TRUE;
    pDevice->bWindowed       = TRUE;
}
```

You do the check by comparing the device capabilities to the `D3DCAPS2_CANRENDERWINDOWED` value. If the device can be windowed, then the `bCanDoWindowed` and `bWindowed` flags are set to `True`.

Choosing the Default Adapter

The final block of code in the `hrBuildDeviceList()` function sets the default adapter returned to the calling function. The check is done by looping through all of the adapters and their devices.

If the device supports windowing and the game is to be windowed, then the adapter is chosen. If the game is to be run in full-screen, then the first available adapter is picked. Once the default adapter is chosen, it is stored in the `g_dwAdapter` global variable. This variable sets up the display parameters in the `hrInit3D()` function.

Whew! Talk about a complicated function. If you are a little confused about it, don't worry; you can pretty much plug it in and forget about it. For other examples of this, check out the Direct3D framework that comes with the DirectX SDK. It has examples of setting up the video mode.

TIP

Many video cards do not support windowed 3D. Be sure to make your application flexible enough to switch between windowed and full-screen mode based on the user's settings. *SpacePirates* allows the switch with the `g_bWindowed` flag. The best-case scenario is to make this variable adjustable through a configuration file or GUI.

The hrInitDirectInput() Function

SpacePirates uses DirectInput because it provides for much smoother control over the keyboard. However, DirectInput is not always the best for text input, so the game actually uses a little bit of both DirectInput and standard keyboard handling. The actual movement control and firing control are activated through DirectInput. Text entry for in-game commands is handled through the Windows keyboard handler. Here is the code for the initialization function.

```
HRESULT     hrInitDirectInput(void)
{
    HRESULT     hReturn = 0;

    // Do not try to create DirectInput if already created
    if( !pDI ) {
        // Create a DInput object
        if( FAILED( hReturn = DirectInput8Create( g_hInstance,
            DIRECTINPUT_VERSION,
            IID_IDirectInput8, (VOID**)&pDI, NULL ) ) )
            return(INPUTERROR_NODI);
    }
    // Get the keyboard layout
    g_Layout = GetKeyboardLayout(0);
    return(hReturn);
}
```

You can give thanks now for short functions, eh? This should look familiar to you because it is copied from the DirectInput chapter. Anyway, the main thing accomplished here is the creation of the `IDirectInput8` interface. Once this is accomplished, a call is made to the `GetKeyboardLayout()` function. You might recall that this happens for when you want to convert scan codes to ASCII codes later.

The hrInitKeyboard() Function

The `hrInitKeyboard()` function is used just as it was in the DIK_Read project—it creates and initializes the DirectInput keyboard device. You use buffered input here so no keystrokes are lost. As long as everything works out okay, no errors are returned. Here is the code.

```
HRESULT hrInitKeyboard(HWND hWnd)
{
    HRESULT                       hReturn = 0;
    DIPROPDWORD                   dipdw;

    // Don't try to create the keyboard twice
    if( pKeyboard )
        return(INPUTERROR_KEYBOARDEXISTS);
    // Exit out if no DirectInput interface found
    else if ( !pDI )
        return(INPUTERROR_NODI);
    // Obtain an interface to the system keyboard device
    if( FAILED( hReturn = pDI->CreateDevice( GUID_SysKeyboard,
     &pKeyboard, NULL ) ) )
        return(INPUTERROR_NOKEYBOARD);
    // Create buffer to hold keyboard data
    ZeroMemory(&dipdw,sizeof(DIPROPDWORD));
    dipdw.diph.dwSize            = sizeof(DIPROPDWORD);
    dipdw.diph.dwHeaderSize      = sizeof(DIPROPHEADER);
    dipdw.diph.dwObj             = 0;
    dipdw.diph.dwHow             = DIPH_DEVICE;
    dipdw.dwData                 = KEYBOARD_BUFFERSIZE;

    // Set the size of the buffer
    if( FAILED( hReturn = pKeyboard->SetProperty( DIPROP_BUFFERSIZE,
     &dipdw.diph ) ) )
        return(INPUTERROR_NOKEYBOARD);
    // Set the format of the keyboard
    if( FAILED( hReturn = pKeyboard->SetDataFormat( &c_dfDIKeyboard ) ) )
        return(INPUTERROR_NOKEYBOARD);
    // Set the cooperative level to exclusive access
    if( FAILED( hReturn = pKeyboard->SetCooperativeLevel( hWnd,
     DISCL_NONEXCLUSIVE | DISCL_FOREGROUND ) ) ) {
        return(INPUTERROR_NOKEYBOARD);
    }
    // Acquire the keyboard device
    pKeyboard->Acquire();
    return(0);
}
```

As you can see from the code, the following events transpire.

1. The keyboard device is created with the `CreateDevice()` function.
2. The buffer is set up with the `SetProperty()` function.
3. The format is set up with the `SetDataFormat()` function.
4. The cooperative level is set with the `SetCooperativeLevel()` function.

In *SpacePirates*, the cooperative level is set to be non-exclusive. This allows more than one iteration of the game to run on the user's desktop. If you set the cooperative level to exclusive and try to run two iterations of the game, you'll notice that each keystroke is sent to both windows! Go ahead and try it if you want to see it in action. You can look at Figure 14.19 to see the setup steps illustrated.

The vInitGeometry() Function

You only have three more initialization functions after this one to wade through; hang in there! The `vInitGeometry()` function actually does more than just initialize the 3D data. It also loads up the textures used in the game.

Loading the Robot Models

To animate the robot in the game, the program actually loads multiple copies of it in different poses. There are 12 frames in all to simulate a walking motion. You can see these frames of movement in Figure 14.20. You do not have to use 12 frames for every walk cycle you create—I just use a number of frames that looks good.

To load the 12 frames, all the program has to do is loop through some code that loads the separate models into their own `Object3DClass` object. Look at the following code, and you can see how each frame is loaded into one element of the `o3dShip[]` global array. This array is used in the rendering function later to display the robots.

```
for( i = 0 ; i < 12 ; i++ ) {
      sprintf(szFileName,"Data\\3DObjects\\droid%d.x",i);
      o3dShip[i].hLoad(szFileName,pd3dDevice);
}
```

Creating the Arena Geometry

The next part of the `hrInitGeometry()` function deals with creating the custom geometry for the arena. The arena actually consists of several quads, but you only need to

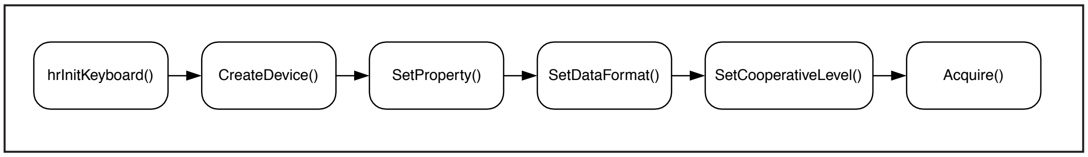


Figure 14.19 *Keyboard creation*

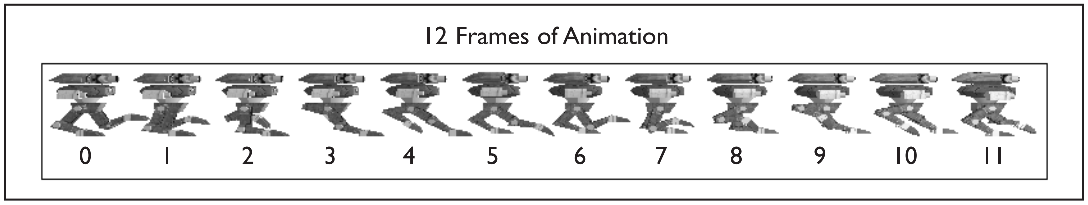


Figure 14.20 *Frames for the 3D robot*

create one and use it multiple times. There is nothing really new with creating the quad; you use the same techniques found in the Draw3DTexturedPrimitive project. One thing to keep in mind is that the quads used in *SpacePirates* have texture, color, and normal information stored in them. This is necessary to take advantage of texturing and lighting. To see the arena, check out Figure 14.21.

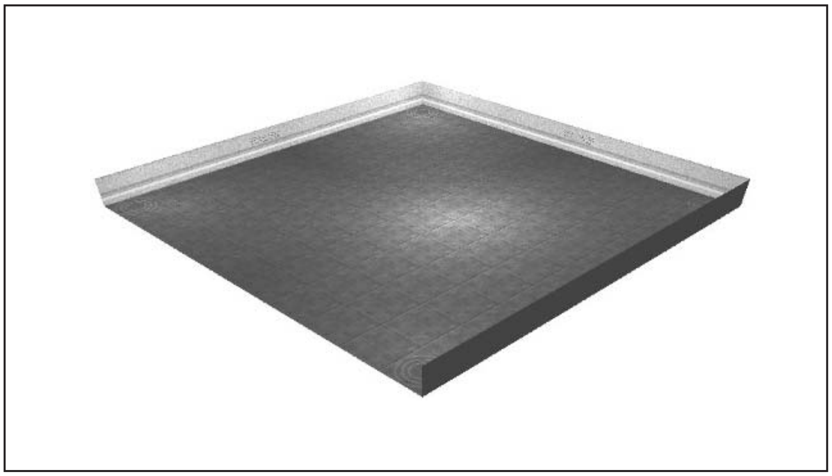


Figure 14.21

The 3D arena

Loading the Textures

The game uses a few textures to create such things as the arena, explosions, and bullets. The first set of textures to be loaded is used for the explosion animation when a player gets shot or when a bullet hits the arena.

```
for(i=0;i<8;i++) {
    sprintf(szFileName,"Data\\3DObjects\\exp%d.tga",i);
    if( FAILED( D3DXCreateTextureFromFile( pd3dDevice, szFileName,
     &g_pTexture[i] ) ) ) {
        return;
    }
}
```

You can look at this code and see that the program loops through eight frames worth of loading. Each time a texture is loaded, it is stored in the next `g_pTexture[]` array element. The next several lines of code load the remaining textures. To see all of the textures used, refer to Figure 14.22.

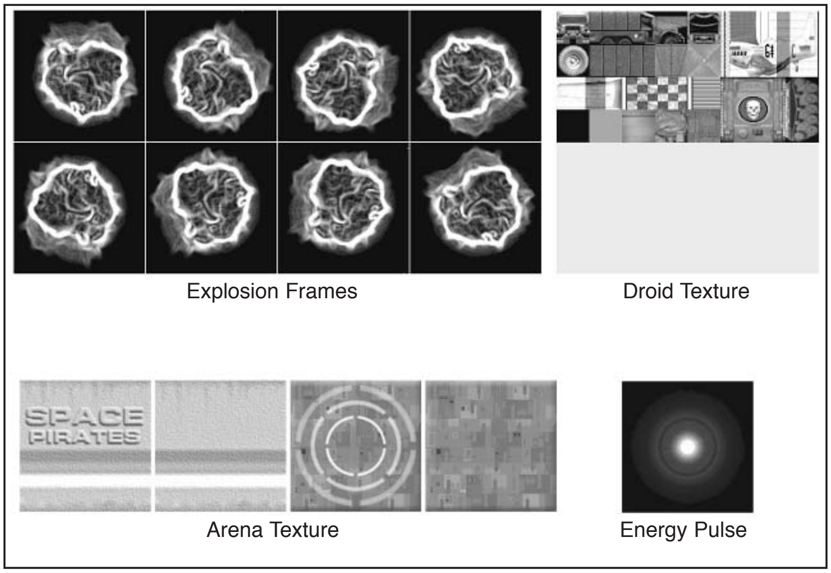


Figure 14.22

Custom textures for SpacePirates

Loading the Game Font

You might recall from the design phase of the project that the game is to have text on the screen for the player's score and other messages. To use the Direct3D font system, you must load up a system font. I like the Georgia font, so it is the one loaded. Here is the code to load the font.

```
hFont = CreateFont(26, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, PROOF_QUALITY,
 0, "georgia");
D3DXCreateFont(pd3dDevice, hFont, &pD3DXFont);
```

The `CreateFont()` function is provided by Windows and loads up the system font. The `D3DXCreateFont()` function is a Direct3D function that converts the system font to a 3D font. Once the font is loaded, you can use it to display text in a 3D scene.

NOTE

The `CreateFont()` function will load the default system font if you try to use one that it cannot find. So, if you don't have the Georgia font, don't worry; the system will load a default font for you.

The vSetupView() Function

The `vSetupView()` function is similar to the `vSetupView()` function in past projects. The only real difference here is where the camera is located. If you look at the source code for it, you see that the camera is set back and up from the center of the world. This gives the player a bird's-eye view of the arena.

```
void vSetupView(void)
{
    // Set where camera is viewing from
    vecCameraSource.x = -300.0f;
    vecCameraSource.y = 50.0f;
    vecCameraSource.z = 150.0f;
    // Set what camera is looking at
    vecCameraTarget.x = 0.0f;
    vecCameraTarget.y = 0.0f;
    vecCameraTarget.z = 0.0f;
    // Set up camera view
    D3DXMatrixLookAtLH( &matCameraView,
                                &vecCameraSource,
                                &vecCameraTarget,
                                &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
    // Tell the device to use the camera view for the viewport
    pd3dDevice->SetTransform( D3DTS_VIEW, &matCameraView );
    // Set up projection
    D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 800.0f );
    // Tell the device to use the above matrix for projection
    pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
}
```

The vCreateLights() Function

The `vCreateLights()` function has been updated for *SpacePirates* to include multiple colored lights. Four lights of different colors are put into the arena's four corners. In the center of the arena is one white light. The multiple colored lights give the arena a cool effect and keep things interesting.

You perform the setup for the lights using the `SetLight()` and `LightEnable()` functions. Look at Figure 14.23 to see how the lights are arranged in the arena.

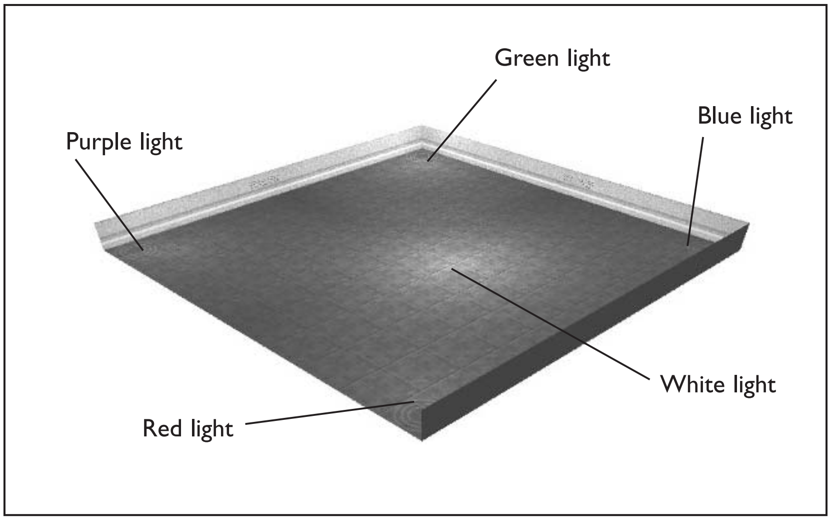


Figure 14.23

Lighting layout in the arena

The vInitSoundEngine() Function

The sound class requires initialization, so I put that functionality into the `vInitSoundEngine()` function. Here is the code used.

```
void vInitSoundEngine(HWND hWnd)
{
    SoundEngine.hWnd = hWnd;
    if( !FAILED(SoundEngine.hrInitSoundSystem()) ) {
        g_GameSounds.gsExplosion = new GameSound;
        SoundEngine.hrLoadSound("data\\sfx\\hit.wav",
        g_GameSounds.gsExplosion);
        g_GameSounds.gsFire = new GameSound;
        SoundEngine.hrLoadSound("data\\sfx\\attack.wav",
        g_GameSounds.gsFire);
    }
}
```

The first function called is `hrInitSoundSystem()`. This function belongs to the global `SoundEngine` class variable. If it succeeds, the game sounds are loaded. There are two game sounds, one for firing and one for explosions.

Hey, you are done with game initialization! Yay, throw a party, go to Mardi Gras—oh, wait; you still have to write the actual game.

Controlling the Game

Now that the game is initialized and ready to go, you need to plan how the control mechanism is going to work. Actually, all of this was planned before the code was ever written, but who's counting anyway? In the game design, it was determined that each player would be able to move his robot around the arena and shoot at other robots. When a player's bullet hits another player, the firing player gets points. This is fine and works very well for the game's purposes. There is one problem, though: How will the game let the player choose to join a game or host a game?

The Command Console

The actual connection parameters are read from the config.txt file, but choosing to host or join happens through the game interface. One way that games such as *Quake* handle this type of input is through a text console. Figure 14.24 shows the *Quake III* command console.

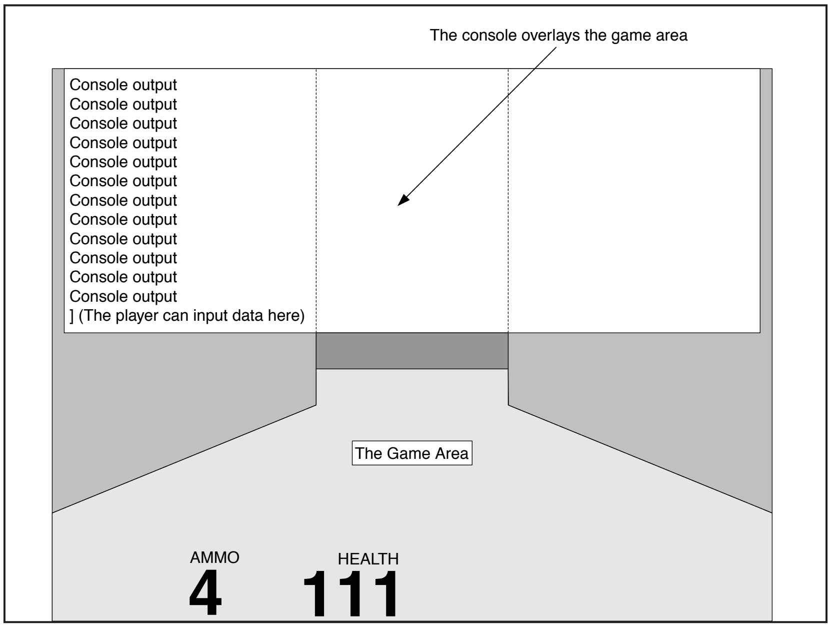


Figure 14.24

Command console layout

As you can see in Figure 14.24, the command console appears in the game area. This allows players to enter various information and set flags while playing the game. *SpacePirates* uses a watered-down version of this concept. When players press the Enter key, they are put into command mode. While they are in command mode, the text "Command:" is displayed on the screen. Anything they type is added to the command. When the player is done entering his command, he can press Enter to execute the command.

The Commands

SpacePirates needs to process only two commands, host and join. If the player uses the host command, then he attempts to host the game. If the player uses the join command, then he attempts to connect to the IP address found in the config.txt file. The `hrJoin()` and `hrHost()` functions are responsible for joining and hosting the game. Each of these is virtually identical to the ones shown in the peer-to-peer DPChat project. I suggest that you go back and look at that project if you are a little fuzzy on how joining and hosting work.

The reason *SpacePirates* uses a config.txt file is that it makes it a little easier to demonstrate the game without spending too much time on interfaces. If *SpacePirates*

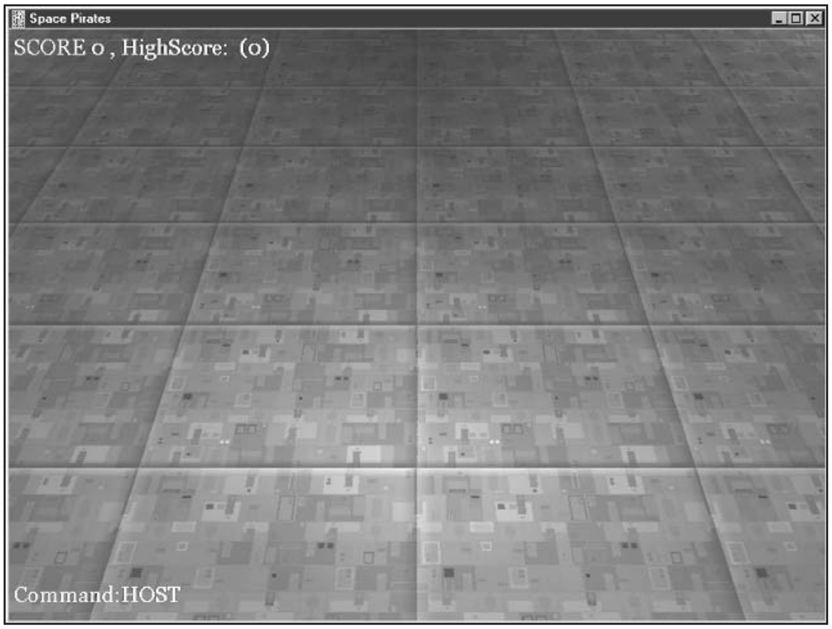


Figure 14.25

The SpacePirates *command console*

were a commercial game, it would definitely use in-game menus to set up the parameters for play. You can see the console used in *SpacePirates* in Figure 14.25.

Movement Keys

Most games let you specify the keys you want to use for controlling your character. This is the optimal solution because players can configure the game to their liking. *SpacePirates* does not follow this method because it would introduce tons of code to the system. Instead, players control their characters using a fixed set of keys. Table 14.1 shows the keys used to control the robots and each key's respective function.

Table 14.1 SpacePirates Key Controls and Functions

Key	Function
Left arrow	Rotate left
Right arrow	Rotate right
Up arrow	Speed up
Down arrow	Slow down/reverse
Spacebar	Fire bullet

Each of the movement arrow keys looks for the DirectInput macro to check the values. The fire button looks to the ASCII code to be activated. There is no reason you have to use the ASCII code for the spacebar; I just put it in there for demonstration purposes.

The Game Loop

The most important part of the source code is the game loop. The game loop is where all of the magic happens. Everything in the game, from graphics to communication, is handled in the game loop. To get a better understanding of the flow of the loop, look at Figure 14.26. You can see that the game constantly loops through checking Windows messages. If there is no Windows message to process, the game

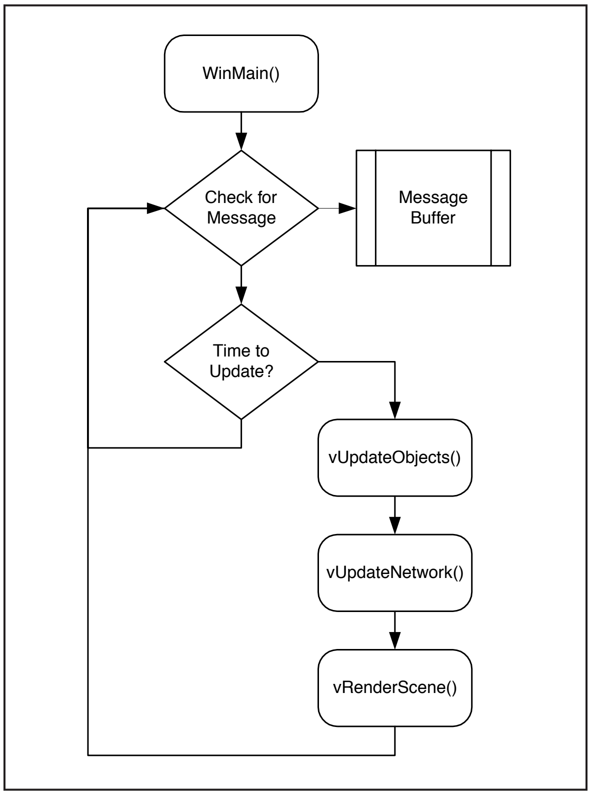


Figure 14.26

The game loop flow

checks to see whether it is time to update the game. When game updates happen, the following functions are called.

- `vUpdateObjects()`
- `vUpdateNetwork()`
- `vRenderScene()`

The `vUpdateObjects()` function is called to update all elements in the scene. The `vUpdateNetwork()` function updates network players of current positions and orientation. The `vRenderScene()` function displays all of the game objects.

Once the check for game updates is completed, the game loop checks the user's input to see whether he has issued a command. The `hrReadKeyboard()` function is called here to update the keyboard state. It works just as it did in the DIK_Read project.

TIP

To keep faster machines from having an advantage, the game is locked at 30 frames per second with the `dwGovernor` timer.

Now that the keyboard state is updated, the program checks each of the command keys for input. If the player has pressed one of the keys, then a global variable is set with his command. The next time the game is updated, the global command variables are checked to see whether the player has issued a command. If the player has issued a command, then action is taken.

CAUTION

If you execute player commands outside of the update functions, then faster players have an advantage. By forcing updates to happen within the timed update loop, all players have equal opportunity.

Updating the Game

The three functions that deal with updating the game are `vUpdateObjects()`, `vUpdateNetwork()`, and `vRenderScene()`. These three functions are called no more than 30 times per second due to the time governor in place. This is an essential aspect of game updates for multiplayer games.

The vUpdateObjects() Function

The vUpdateObjects() function actually serves two purposes. The first purpose is to send action packets based on the user's input, and the second purpose is to update all in-game objects.

Processing Player Actions

The five possible actions the player can take are accelerate, decelerate, turn left, turn right, and fire bullet. Each of these actions is flagged in the main game loop by pressing a particular key. With this scheme, it is actually possible to process both a turn-left and a turn-right message in the same loop; the two just negate each other.

Processing Acceleration

To test whether a particular action should be executed, the program checks the global Boolean variable for the action. For example, here is the code to execute an acceleration action.

```
// Accelerate
if( bAccelerate ) {
    bAccelerate = 0;
    if( PlayerInfo[g_iMyPlayerId].fVelocity < 2.0f )
        PlayerInfo[g_iMyPlayerId].fVelocity += 1.0f;
    pacVel.dwSize = sizeof(PacketVelocity);
    pacVel.dwType = PACKET_TYPE_VELOCITY;
    pacVel.fVelocity = (float)PlayerInfo[g_iMyPlayerId].fVelocity;
    // Convert the packet to a void stream
    packet = (VOID*)&pacVel;
    // Send the velocity packet
    hReturn = hrSendPeerMessage(-1,PACKET_TYPE_VELOCITY,packet);
}
```

Take notice of how the bAccelerate variable is checked before executing the player action. You might recall that this variable gets set in the main game loop back in WinMain().

The first thing the code does is set the bAccelerate global to 0. This keeps the action from being called again unless the player presses the up-arrow again. The next logical check is performed to see whether the player is already moving along at the maximum speed. If the player's speed is less than 2.0, then he speeds up by 1.0 unit. You might want to change this maximum setting to see what happens.

Once the new velocity is figured out, a packet is sent to all of the other players connected to the game, using the packet type of PACKET_TYPE_VELOCITY. The hrSendPeerMessage() function sends the actual packet. This function is just like the one used in the peer-to-peer DPChat project. Once the packet is sent, the code goes on to check the next action.

Processing Rotation

If the player presses the left- or right-arrow keys, he flags a rotation action to take place during the next game update. To demonstrate, here is the code for turning left.

```
if( bTurnLeft ) {
    bTurnLeft = 0;
    PlayerInfo[g_iMyPlayerId].fRot -= 10.0f;
    if( PlayerInfo[g_iMyPlayerId].fRot < 0.0f )
        PlayerInfo[g_iMyPlayerId].fRot = 350.0f;
    pacAngle.dwSize = sizeof(PacketAngle);
    pacAngle.dwType = PACKET_TYPE_ANGLE;
    pacAngle.fAngle = PlayerInfo[g_iMyPlayerId].fRot;
    // Convert the packet to a void stream
    packet = (VOID*)&pacAngle;
    // Send the rotation packet
    hReturn = hrSendPeerMessage(-1,PACKET_TYPE_ANGLE,packet);
}
```

The PlayerInfo[g_iMyPlayerId].fRot variable stores player rotations. By setting the g_iMyPlayerId element of the PlayerInfo[] array, the program is rotating the right robot. You might have noticed that the player rotates his robot in increments of 10 degrees. You can change this if you want smoother rotations. Once the rotation is set, a packet of type PACKET_TYPE_ANGLE is sent to all players.

Firing a Bullet

There wouldn't be much competition in the game if the players couldn't shoot at one another, so the game includes the ability to fire pulses of energy at other players. When a player presses the spacebar, the fire bullet action is set. Here is the code to process the bullet request.

```
if( bFireBullet ) {
    vCreateBullet(PlayerInfo[g_iMyPlayerId].dpnidPlayer,
    PlayerInfo[g_iMyPlayerId].vecCurPos.x,
    PlayerInfo[g_iMyPlayerId].vecCurPos.z,
     (float)PlayerInfo[g_iMyPlayerId].fVelocity,
    PlayerInfo[g_iMyPlayerId].fRot);
    bFireBullet = 0;
}
```

The vCreateBullet() Function

The `vCreateBullet()` function creates a bullet from any player. For parameters, it takes the `DPNID` of the shooter; the x, y, and z coordinates; the velocity; and the angle of the bullet. The angle of the bullet determines which direction it is moving. Here is the code for that function.

```
void vCreateBullet(DPNID dpowner, float fx, float fz,
 float fvel, float frot)
{
    int                 i;
    PacketBullet        pacBul;
    void                *packet;
    HRESULT             hReturn;

    for( i = 0 ; i < MAX_BULLETS ; i++ ) {
        if( !Bullet[i].bActive ) {
            break;
        }
    }
    // Exit if no room for bullets
    if( i == MAX_BULLETS )
        return;
    Bullet[i].bActive = 1;
    Bullet[i].fx = fx;
    Bullet[i].fz = fz;
    Bullet[i].fVelocity = fvel+3.0f;
    Bullet[i].dpOwner = dpowner;
    Bullet[i].fRot = frot;
    SoundEngine.hrPlaySound(g_GameSounds.gsFire);
    // This is our bullet; send a bullet packet to everyone
    if( dpowner == g_dpnidLocalPlayer ) {
        pacBul.dwSize = sizeof(PacketBullet);
        pacBul.dwType = PACKET_TYPE_BULLET;
        pacBul.Bullet.bActive = 1;
        pacBul.Bullet.dpOwner = g_dpnidLocalPlayer;
```

```
            pacBul.Bullet.fRot = frot;
            pacBul.Bullet.fVelocity = fvel;
            pacBul.Bullet.fx = fx;
            pacBul.Bullet.fz = fz;
            // Convert the packet to a void stream
            packet = (VOID*)&pacBul;
            // Send the bullet packet
            hReturn = hrSendPeerMessage(-1,PACKET_TYPE_BULLET,packet);
        }
}
```

The first thing the create bullet function does is loop through the number of possible bullets looking for an empty slot. This is done by checking the `bActive` flag. If no slots are found, the function exits.

The next part of code sets up the parameters for the `Bullet[]` array element that was found to be available. By doing this, the next game update includes the new bullet in it.

The last order of business is to check whether the *local player* is the one who fired the bullet. If the local player did fire the bullet, then a packet is sent to all of the other players, telling them that a new bullet has entered the game. The packet used to do this is type `PACKET_TYPE_BULLET`.

The term *local player* refers to the player who is actually playing on the computer. Remote players are players who are competing from other locations.

That just about sums it up for possible player actions. Look at Figure 14.27 to see the actions illustrated.

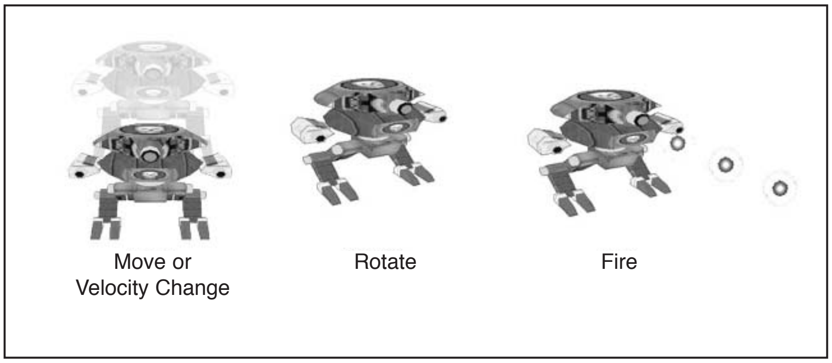


Figure 14.27

Possible player actions

Updating Player Objects

Each and every player in the game, be it the local player or a remote player, must have his position updated every game update. This is necessary to maintain smooth motion in the game. If you updated players only when packets arrived, then you would see jerky motion. *SpacePirates* isn't the prime example for smooth motion, considering that it doesn't use interpolation, but it gets the job done well enough for the current level of difficulty.

The `vUpdateObjects()` function is responsible for updating all in-game objects, including players, bullets, and explosions. The first things to be updated are the players. You do this by looping through the `PlayerInfo[]` array to look for active players. If a player is active, then the game updates him.

Updating Player Positions and Animation

The player loop not only updates the position of each player, it also updates the animation frame for each player. Because there are 12 robot animation frames, there can be 12 animation positions for each player.

The first thing the player loop does is figure out the velocity for the player using the `vGetVelocity()` function that I included with the project. The `vGetVelocity()` function tells you how much to offset the model on the x and z coordinate, based upon its heading.

```
vGetVelocity(PlayerInfo[i].fRot,fx,fz);
```

Once you have the x and z velocity, you multiply them by the timing multiplier. Recall from the section about timing that you do this to make up for machines that are either too fast or too slow.

```
fx *= fMultiplier;
fz *= fMultiplier;
```

Now that you have the modified x and z offsets, you just add them to the player information structure to get the new position.

```
PlayerInfo[i].vecCurPos.x += (float)(fx*PlayerInfo[i].fVelocity)*-1;
PlayerInfo[i].vecCurPos.z += (float)(fz*PlayerInfo[i].fVelocity)*-1;
```

You update the player's animation by checking whether his robot has moved far enough from his last animated position. If he has moved the required distance, then the player's animation frame is incremented. All of this is handled by the `PlayerInfo[i].vecLastPos` and `PlayerInfo[i].vecCurPos` data members. Once the animation frame counter is updated, a final check makes sure that the animation frame does not go over the last frame. If it does, then the animation loops back to the beginning.

```
// Check if animation needs to loop
if( PlayerInfo[i].iFrame >= 12 )
    PlayerInfo[i].iFrame = 0;
else if( PlayerInfo[i].iFrame < 0 )
    PlayerInfo[i].iFrame = 11;
```

The next few lines of code check to see whether the player's robot has hit one of the arena walls. If it does hit a wall, then its velocity is set to `0`. This keeps the players within the arena.

Updating the Bullets

The bullets in the game are updated just as the players are. The code runs through the list of bullets and processes each one that is active. When the program finds a bullet that is active, its position is updated. Bullets in *SpacePirates* don't have animation frames, so at least you don't have to worry about that aspect of it.

One thing that is unique to the bullets is that you have to check to see whether they hit a player in the game. If they hit anyone but their owner, then an explosion is created and the player who shot the bullet gains some points. The collision detection used is a standard distance algorithm that checks whether the bullet is within range of a player. If the bullet gets close enough to a player, an explosion happens.

When a bullet hits another player or a wall, it is flagged for destruction. The bullet is deactivated, and an explosion is created in its place. To create an explosion, the code loops through all possible explosion objects until it finds an open slot. Once an open slot is found, the system sets the explosion up and activates it for future rendering.

```
if( iKillBullet ) {
    Bullet[i].bActive = 0;
    iKillBullet = 0;
    // Find next explosion object
    for( j = 0 ; j < MAX_EXPLOSIONS ; j++ ) {
        if( !Explosion[j].bActive ) {
            Explosion[j].bActive = 1;
            Explosion[j].fx = Bullet[i].fx;
            Explosion[j].fz = Bullet[i].fz;
            Explosion[j].iFrame = 0;
            Explosion[j].dwFrameTime = timeGetTime()+100;
```

```
                SoundEngine.hrPlaySound(g_GameSounds.gsExplosion);
                break;
            }
        }
}
```

Notice that the explosion sound is played when the explosion is created. This happens with the `hrPlaySound()` member function of the `SoundClass`.

Updating the Explosions

The explosions are static entities, but they do have frames of animation. The frame for each explosion is updated if the frame timer has exceeded 100 milliseconds.

```
for( i = 0 ; i < MAX_EXPLOSIONS ; i++ ) {
    if( Explosion[i].bActive ) {
        // Update explosion frame
        if( timeGetTime() > Explosion[i].dwFrameTime ) {
            Explosion[i].iFrame ++;
            Explosion[i].dwFrameTime += 100;
        }
        if( Explosion[i].iFrame >= 8 ) {
            Explosion[i].bActive = 0;
        }
    }
}
```

Once an explosion has reached its last frame, it is deactivated.

Updating the Highest Score

Every action game from the old school had a high score to beat. *SpacePirates* follows along these lines and keeps track of who has the highest score. It does this by looping through all active players in the game and sorting the scores. The player with the highest score has his name displayed at the top of the play window. You can see this in Figure 14.28.

The loop for the highest score is rather simple.

```
// Check who has highest score
for( j = 0 ; j < MAX_PLAYERS ; j++ ) {
    if( PlayerInfo[j].bActive ) {
        if( PlayerInfo[j].lScore > lHighestScore ) {
                iHighestScoreHolder = j;
                lHighestScore = PlayerInfo[j].lScore;
            }
        }
}
```

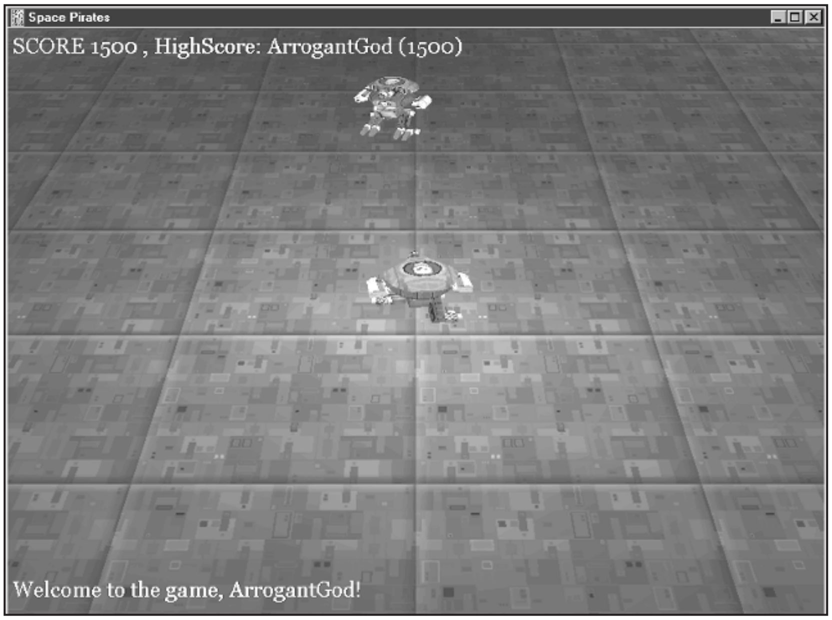
Once the highest score is determined, the `szHighestScoreWinner` global character array is loaded with the name of the high score holder.

Updating the Remote Players

Although you send a packet to everyone whenever you complete an action, games can still get out of sync. This can cause big problems if left unattended, so the `vUpdateNetwork()` function comes in handy.

The vUpdateNetwork() Function

The `vUpdateNetwork()` function is called whenever the game loop updates, which is 30 times per second. Normally, you do not want to update your network 30 times a second, so the `vUpdateNetwork()` function has its own time governor built in. The



SCORE 1500 , HighScore: ArrogantGod (1500)

Welcome to the game, ArrogantGod!

Figure 14.28

The high score and name displayed in the game window function actually limits itself to 10 updates per second, which is a pretty standard frequency for this type of game. Here is the code for the function.

```
void vUpdateNetwork(void)
{
    void                    *packet;
    PacketAngle             pacAngle;
    PacketVelocity          pacVel;
    PacketPosition          pacPos;
    PacketScore             pacScore;
    HRESULT                 hReturn;

    if( timeGetTime() > dwUpdateTime+100 ) {
        // Only send packets if other players are in the game
        if( g_lNumberOfActivePlayers > 1 ) {
            // Send our angle
            pacAngle.dwSize = sizeof(PacketAngle);
            pacAngle.dwType = PACKET_TYPE_ANGLE;
            pacAngle.fAngle = PlayerInfo[g_iMyPlayerId].fRot;
            packet = (VOID*)&pacAngle;
            hReturn = hrSendPeerMessage(-1,PACKET_TYPE_ANGLE,packet);
            // Send our velocity
            pacVel.dwSize = sizeof(PacketVelocity);
            pacVel.dwType = PACKET_TYPE_VELOCITY;
            pacVel.fVelocity = (float)PlayerInfo
            [g_iMyPlayerId].fVelocity;
            packet = (VOID*)&pacVel;
            hReturn = hrSendPeerMessage(-1,PACKET_TYPE_VELOCITY,packet);
            // Send our position
            pacPos.dwSize = sizeof(PacketPosition);
            pacPos.dwType = PACKET_TYPE_POSITION;
            pacPos.fx = PlayerInfo[g_iMyPlayerId].vecCurPos.x;
            pacPos.fz = PlayerInfo[g_iMyPlayerId].vecCurPos.z;
            packet = (VOID*)&pacPos;
            hReturn = hrSendPeerMessage(-1,PACKET_TYPE_POSITION,packet);
            // Send our score
            pacScore.dwSize = sizeof(PacketScore);
            pacScore.dwType = PACKET_TYPE_SCORE;
            pacScore.lScore = PlayerInfo[g_iMyPlayerId].lScore;
            packet = (VOID*)&pacScore;
            hReturn = hrSendPeerMessage(-1,PACKET_TYPE_SCORE,packet);
        }
        dwUpdateTime = timeGetTime();
    }
}
```

The first logic check that the function performs is the check of the update timer. If the function is trying to run at more than the specified 10 times per second, then the function just exits.

Once you are inside the loop to update the other network players, a second check is made to see whether there is anyone in the game except for you. If there are no other players, then there is no need to send packets!

If the game has players, then the function sends one of each packet type to all of the other players. Your speed, position, heading, and score stay current on the other players' machines.

Lastly, the frequency timer is updated to keep the pesky updates at the specified limit. If you want, change the timer to go off 30 times per second and see what it does. You might be surprised at the results.

Rendering the Scene

As much as I like to talk only about multiplayer game programming, the little (but booming) voice in my head reminds me that I have to talk about graphics. Okay, well, the render loop is illustrated in Figure 14.29.

By looking at Figure 14.29, you can tell that the objects in the scene render in a particular order. The arena is rendered first, followed by the players, then the explosions, the bullets, and then the score. The reason the objects are rendered in this order is to ensure that the elements are layered properly from back to front. The code might look complicated, but most of it is due to the special effects such as different wall and ground pieces.

Updating the Camera

The camera is updated using the `vUpdateCamera()` function that is present at the beginning of the rendering loop. This function positions the camera over the player controller robot. You accomplish this by setting the camera's x and z coordinates relative to the controlled robot.

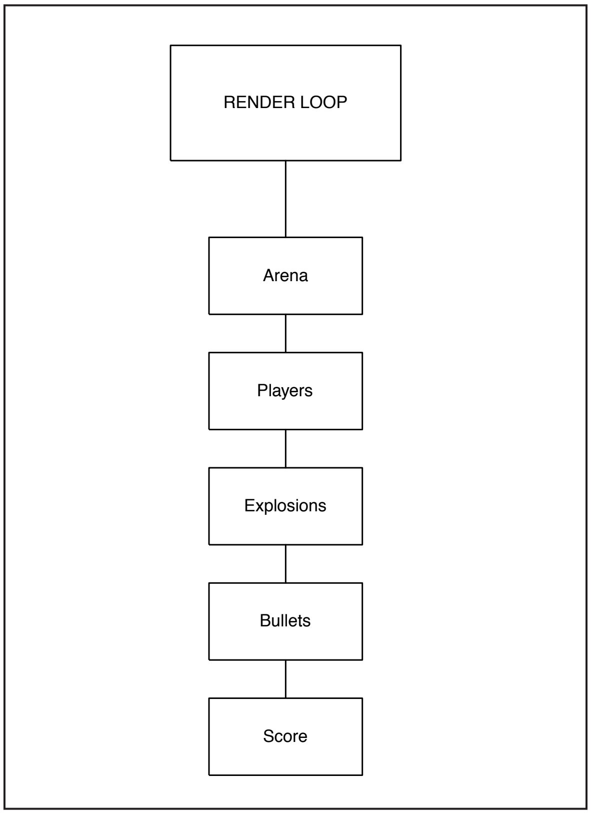


Figure 14.29

The rendering function

Building the Arena

The first part of the rendering loop deals with creating the arena in which the players can compete. The first thing you have to do before displaying your own polygons is set up the vertex stream source. You might recall from the DrawPrimitive project that this is done using the `SetStreamSource()` function.

```
// Set the vertex buffer to render
pd3dDevice->SetStreamSource( 0, lpVertexBuffer, sizeof(MYVERTEX) );
// Set the vertex format
pd3dDevice->SetVertexShader( D3DFVF_MYVERTEX );
```

Once you have the stream set, you then set the vertex shader to use. The one used in *SpacePirates* is the `D3DFVF_MYVERTEX` shader defined in the SpacePirates.h header file.

Now that you have the stream and shader all set up, you can display the floor and walls.

Rendering the Floor

The floor is actually organized into a five-by-five grid of 25 tiles. The rendering function starts in two loops to accomplish the rendering of the tiles. The top-left tile is rendered first, moving horizontally until the entire row is finished. Once a row of tiles is rendered, the next row down is started. Look at Figure 14.30 to see the rendering order.

Rendering the Walls

As the floor is rendered, one tile at a time, the row and column numbers are checked for the existence of a wall piece. If the row or column matches where a wall should be, then a wall is drawn. One neat item involved is the rendering of the

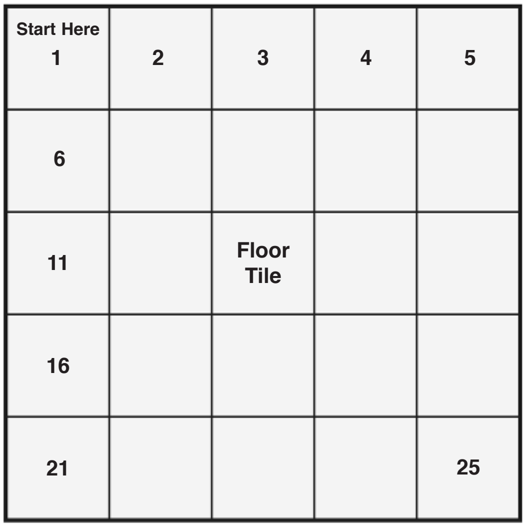


Figure 14.30

The rendering order for the floor tiles game title, “SpacePirates,” on the middle walls. You render the title by changing the active texture when the particular wall pieces need to be drawn. Here is an example.

```
if( j == 0 )
    pd3dDevice->SetTexture( 0, g_pTexture[12] );
else
    pd3dDevice->SetTexture( 0, g_pTexture[11] );
```

When `j` equals `0`, the logo is displayed; otherwise, the normal wall piece is displayed. This logic is what puts the game logo on the top and bottom walls.

Another neat trick used in the game is changing from solid rendering to wire-frame rendering when the player goes behind the bottom wall. This is done so that the player can see his own robot when it goes behind the wall. The code to do this looks like the following:

```
if( PlayerInfo[g_iMyPlayerId].vecCurPos.z >= 200.0f )
    pd3dDevice->SetRenderState(D3DRS_FILLMODE,D3DFILL_WIREFRAME);
else
    pd3dDevice->SetRenderState(D3DRS_FILLMODE,D3DFILL_SOLID);
```

The `SetRenderState()` function accomplishes the task by setting the `D3DRS_FILLMODE` to either solid or wire-frame, depending on the need.

Other checks happen at the end of the arena-rendering loop to see whether a corner piece needs to be displayed. If you launch the game and look at the floor pieces, you’ll notice a special piece for the four corners. I included this piece in anticipation of you adding some new code to the game. Maybe you could set random starting positions at the four corners. On the other hand, maybe you could place power-ups in the corners. It’s really up to you, but I do encourage you to play with the code.

Rendering the Players

The player rendering is handled after the arena is drawn. It is fairly easy to accomplish because you have the `Object3DClass` functionality to do it for you. Here is the code.

```
// Display robots
for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
    if( PlayerInfo[i].bActive )
        o3dShip[PlayerInfo[i].iFrame].vDisplayXYZ(
        PlayerInfo[i].vecCurPos.x,
        0.0f,PlayerInfo[i].vecCurPos.z,
        0.0f,
        PlayerInfo[i].fRot,
        0.0f);
}
```

The program is looping through the entire list of possible players and only rendering the ones that are active. Each player object is rendered according to its current frame, position, and orientation.

Rendering the Explosions

The explosions in the game are displayed whenever a bullet hits a wall or a player. Because the explosions have translucency to them, the render state must be adjusted. You do this by setting the alpha state for the render system.

```
pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
```

With this code, the explosion textures are displayed according to their alpha channel. The result is semi-transparent explosions.

The rendering loop for the explosions works much like the other render loops. It just loops through all possible explosions and renders the ones that are active.

```
for( i = 0 ; i < MAX_EXPLOSIONS ; i++ ) {
    if( Explosion[i].bActive ) {
        D3DXMatrixTranslation( &matTranslation, Explosion[i].fx,
         0.0f+(0.25f*i), Explosion[i].fz );
        D3DXMatrixRotationX( &matRotation, -90.0f/RADIAN_TO_DEGREES );
        matRotation *= matTranslation;
        pd3dDevice->SetTransform(D3DTS_WORLD, &matRotation);
        // Set the explosion's texture frame
        pd3dDevice->SetTexture( 0, g_pTexture[Explosion[i].iFrame] );
        // Draw the primitive
        pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
    }
}
```

You might be wondering what the `0.0f+(0.25f*i)` code is for. This renders each explosion at a slightly different height. This way, the explosions never render into each other. This keeps visual anomalies from happening too often.

Rendering the Bullets

The bullets in the game are rendered much like the explosions, except that the bullets do use the depth buffer for rendering. If the bullets did not use the depth buffer, then they would appear on top of the players. Here is the source code.

```
for( i = 0 ; i < MAX_BULLETS ; i++ ) {
    if( Bullet[i].bActive ) {
        D3DXMatrixScaling( &matScale, 0.5f, 0.5f, 0.5f );
        D3DXMatrixTranslation( &matTranslation, Bullet[i].fx, 15.0f,
         Bullet[i].fz );
        D3DXMatrixRotationX( &matRotation, -90.0f/RADIAN_TO_DEGREES );
        matScale *= matRotation;
        matScale *= matTranslation;
        pd3dDevice->SetTransform(D3DTS_WORLD, &matScale);
        // Draw the primitive
        pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
    }
}
```

The reason lighting is turned off for the explosions and the bullets is that they do not need to be lit with dynamic lighting. By turning off the lights and setting the ambient color to full blast, you render the explosions and bullets at absolute brightness. You do this because you wouldn't want shaded explosions or pulses of energy! Keep this in mind when rendering objects that are self-illuminated.

Displaying the Score

The last elements to be displayed are the score and system message. You accomplish this with the font system's `DrawText()` function.

The color of each font is determined by using the `D3DCOLOR_RGBA` macro, as in the following:

```
D3DCOLOR_RGBA(255, 255, 255, 255)
```

You can change the RGB values for different colored fonts. Feel free to experiment with them until you find a color you like. Personally, I think white is just fine for the task at hand.

Player Management

At the heart of the *SpacePirates* multiplayer engine is the DirectPlay message handler in the form of the `DirectPlayMessageHandler()` function. This function is responsible for handling all incoming network traffic. If you didn't have the message handler, then you wouldn't have a multiplayer game.

The DirectPlayMessageHandler() Function

If you look at the `DirectPlayMessageHandler()` function, you'll see that it has the message types to which you have become accustomed in past projects. Unlike the graphics coding for the game that required new logic, the message handler hasn't changed very much.

The function handles creating and destroying players, host migration, session termination, and game packet decoding. Each of these functions operates almost identically to its counterpart in the peer-to-peer DPChat project.

The hrCreatePlayer() Function

Scroll on through the code until you get to the `hrCreatePlayer()` function, and you'll see the order of the steps in Figure 14.31.

Once the creation packet is decoded, the peer information for the added player is pulled using the `GetPeerInfo()` function. Down further in the code, you'll see that the player is added to the `PlayerInfo[]` data array with the `iAddPlayer()` function.

The `iAddPlayer()` function loops through the entire list of possible players and assigns the new player to the first available slot. All of the parameters for the player are set to the defaults until an update packet is received.

Once the player is added, the function leaves the update player critical section and returns to the message handler.

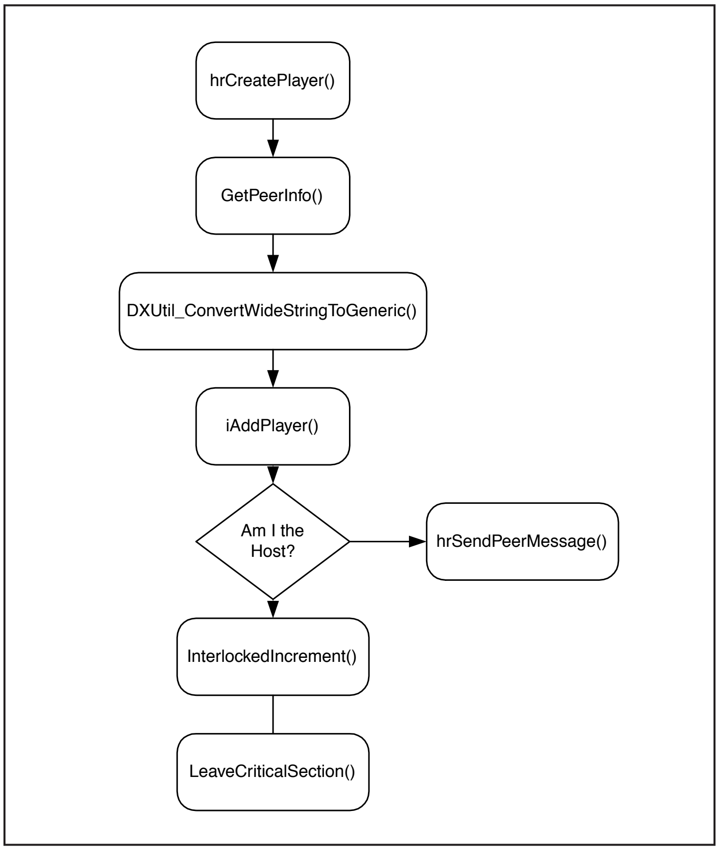


Figure 14.31

The `hrCreatePlayer()` *function*

The hrDestroyPlayer() Function

The nemesis of the `hrCreatePlayer()` function is the dreaded `hrDestroyPlayer()` function. Okay, maybe it isn't that dreaded. . . . Anyway, `hrDestroyPlayer()` is called when a player exits or is kicked out of a networked game. The main purpose of the destroy player function is to remove the player from the global player data array.

```
HRESULT     hrDestroyPlayer( PVOID pvUserContext, PVOID pMsgBuffer )
{
    PDPNMSG_DESTROY_PLAYER      pDestroyPlayerMsg;
    HRESULT                     hReturn = S_OK;
    int                         i;
    char                        szOutput[256];
    // Get a destroy message pointer to the buffer
    pDestroyPlayerMsg = (PDPNMSG_DESTROY_PLAYER)pMsgBuffer;
    // Update the number of active players in a thread-safe way
    InterlockedDecrement( &g_lNumberOfActivePlayers );
    EnterCriticalSection( &g_csModifyPlayer );
    // Remove player from list
    for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
        if( PlayerInfo[i].bActive ) {
            if( PlayerInfo[i].dpnidPlayer ==
             pDestroyPlayerMsg->dpnidPlayer ) {
                PlayerInfo[i].bActive = 0;
                sprintf(szOutput,"<%s> Left The Game",
                PlayerInfo[i].szPlayerName);
                sprintf(szSystemMessage,szOutput);
                break;
            }
        }
    }
    LeaveCriticalSection( &g_csModifyPlayer );
    return(hReturn);
}
```

As you can see in the code, the destroy player function loops through the entire list of players and looks for the matching player ID to destroy. Once the match is found, the player is set to be inactive. This removes him from all future rendering and game updates.

Handling Game Packets

The main portion of the code remaining in the message handler is dedicated to handling the game packets such as position, rotation, and speed updates. Check out Figure 14.32 to see how the packets relate to game functions and objects.

In Figure 14.32, you can see that angle, velocity, and position packets all affect player data. The bullet packet adds a new bullet to the game, and the score packet is used for a score update.

None of the packet-handling routines require knowledge of nuclear physics because they are straightforward. The only subfunction used is the `vCreateBullet()` function that was covered earlier. The rest of the operations just update the appropriate global data and move on.

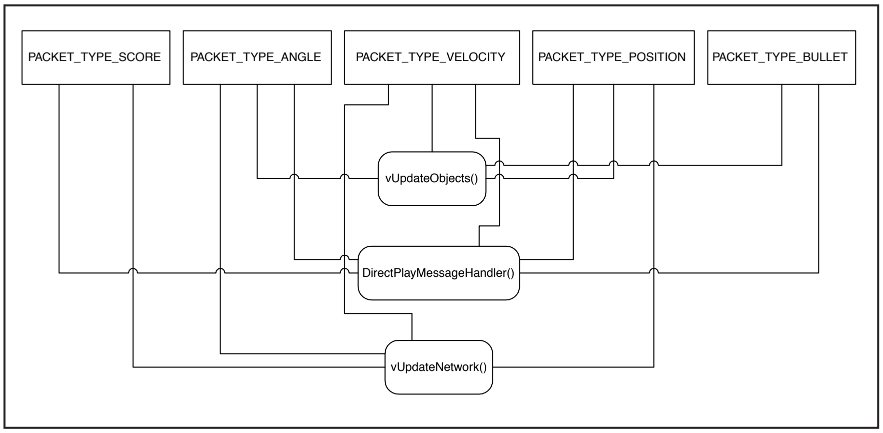


Figure 14.32

Packet-to-object relationships

Review

Isn't that great? The network code portion of *SpacePirates* is actually the easiest part, in my opinion. It just goes to show you that once you understand multiplayer gaming, the multiplayer part becomes the easiest portion. As a quick review, browse the following points.

- You should always pre-plan sound implementation in your games. Players will appreciate it.
- Always update your game events based on the amount of real time that has elapsed. This helps your game run evenly regardless of hardware.
- By including time multipliers in your game update function, you can help keep players with slow machines competitive.
- Be careful how often you update players across the network. A good rule of thumb is to update everyone a maximum of ten times per second.
- Identity matrices are used to set up 3D objects in their default position, orientation, and scale.
- Render 3D objects from back to front for proper ordering. Z-buffers are used to compensate for out of order objects, but they are not as effective as good old ordering.

Hey wait; you haven't seen MMORPG programming and design yet. The best is yet to come!

Part Four

Developing Online Worlds

CHAPTER 15

MMORPG Design and Implementation

Wow, isn't that a great chapter title? In case you have slept since reading the previous chapters, MMORPG stands for *Massively Multiplayer Online Role Playing Game*. Of course, the game does not have to be a role-player game per se; it can be any game in which many people can play at once. The other rule this type of game must follow is that the game servers must save players for as long as the game is in existence. This means that players can log out, log back in five days later, and pick up where they left off. Some of the games that follow this method include *Asheron's Call*, *Ultima Online*, and *EverQuest*. Because these types of games are such great sellers, you probably want to know how to write your own. Well, get ready: You are about to learn what you need to know to get started down the path to glory.

This chapter has three main parts:

- Game architecture
- Server architecture
- Client architecture

You are not building a house here or anything, but you have to understand the architecture of MMORPGs before you can make your own.

Game Architecture

So what do I mean by game architecture? I mean what it takes for a game to deliver a massively multiplayer experience. To begin with, your game must be persistent. This is fairly easy to comprehend, but implementation is another story.

Persistent Worlds

A persistent-world game maintains players' states even when they are not playing the game. This means that the game world must permanently store all pertinent information about a player. The main advantage to players is that they don't have to store their characters locally on their machines. If players don't have to store the information locally, the game lets them play from any computer or console that has the game software.

The work that goes into saving player data might seem trivial, but it is anything but that. First of all, you must store everything about the player and the game world.

Multiply the player's data by thousands, and it starts to add up. Now, you must throw into the mix information about every object in the game. Just think about how many *Ultima Online* players have thousands of items. Each item requires a little bit of storage, so the data alone gets to be huge.

Data Storage

You really have only a couple of solutions for storing game world data—either customized flat files or off-the-shelf databases. The first method, a customized file system, might seem like the obvious choice. I mean, all of us programmers know that everything deserves to be reinvented, right? Okay, maybe I'm kidding.

Customized storage solutions offer the flexibility of an in-house product. By this, I mean that you have complete control over how the system works. You also don't have to wait for last-minute patches from vendors or deal with bugs that they do not want to fix. So what are the downsides? Well, for one, you have to reinvent the proverbial wheel. There are reasons why software by Microsoft and Oracle sell so well. It is because products such as SQL Server and Oracle specialize in only one thing, data storage and retrieval. The other downside is that you have to spend quite a bit of time writing a comprehensive file solution. If you are just starting out, then a custom file system might be your only choice due to a lack of funds. If you can afford it, I recommend that you at least try Microsoft's SQL Server.

Pre-built solutions are great because someone else has already gone through the trouble of developing them for you. Not only do the products provide one-stop solutions, but they have also been tested by thousands of users. The fact that thousands of people use Oracle's products or Microsoft's products goes a long way toward proving a valuable product.

Data Storage Layout

Maybe you have shelled out the $3,000 or so it takes to get started with Microsoft's SQL Server product; what do you do now? I hope you have created an entire game design document that outlines every bit of information you have to store. Nevertheless, don't worry; for now you can act as if you have one. Imagine that you want to store a player's name, address, phone number, account number, and date of purchase. Now, that sounds a lot like account information. With this in mind, you'd probably create a database called AccountInfo. Within this database, you could create a table called ContactInfo. Within the table, you could store the names, addresses, and phone numbers as columns of data. Look at Figure 15.1 to see how the database is laid out so far.

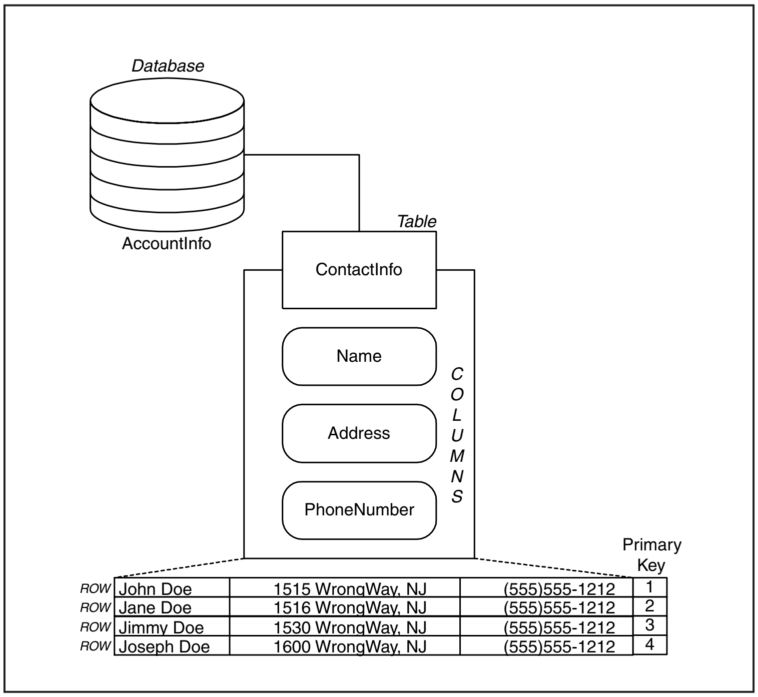

Database
AccountInfo
Table
ContactInfo
Name
Address
PhoneNumber
COLUMNS
Primary Key
ROW John Doe 1515 WrongWay, NJ (555)555-1212 1
ROW Jane Doe 1516 WrongWay, NJ (555)555-1212 2
ROW Jimmy Doe 1530 WrongWay, NJ (555)555-1212 3
ROW Joseph Doe 1600 WrongWay, NJ (555)555-1212 4

Figure 15.1

The account information database

In Figure 15.1, I have labeled the database, table, and columns. Databases can contain multiple tables. In this case, however, there is only one table, the ContactInfo table. Tables consist of columns of data. Each record entered into a database has information stored in each of the columns. These records are referred to as rows of data.

One important item you need to add to the ContactInfo table is a primary key. Primary keys in databases uniquely identify records or rows. For a real-world example, you can look to the United States government's use of Social Security numbers. You can think of a Social Security number as a primary key to government's database of citizens.

Data Access

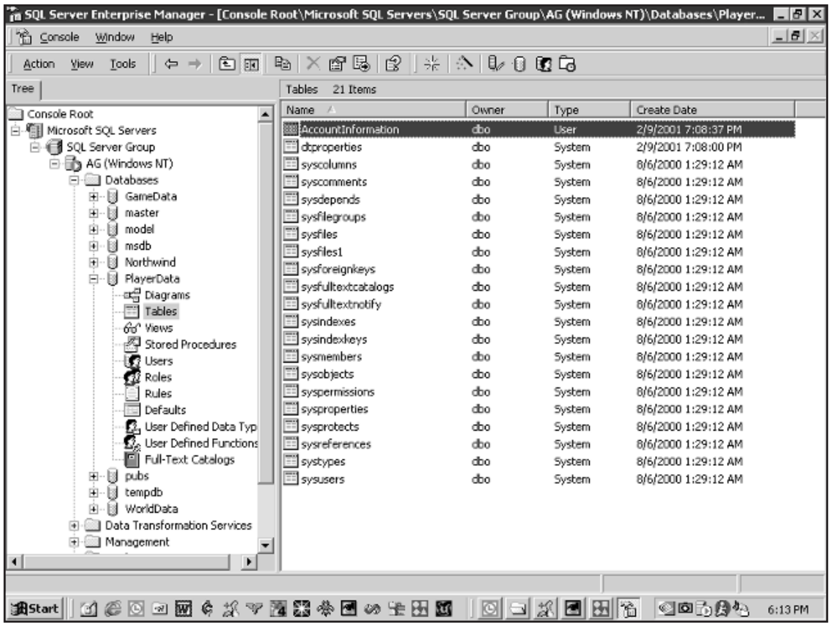
Once you lay out all of your databases and tables, you can start to worry about how to access the data in a timely fashion. The main problem here is that once you start getting millions of entries, the database gets slower and slower. To combat this, you must plan carefully.

If you have created your own file system, then your options are only limited by the functionality you have created. Unfortunately, I can't cover the options because I didn't create them! However, I can cover the options you have if you go with a ready-made product such as SQL Server. To see the table information GUI for Microsoft's SQL Server, look at Figure 15.2.

Figure 15.2 shows an example of data storage where I have created a database called PlayerData. Within the database, I have created a table called AccountInformation. The good part about using a pre-packaged solution is that you normally have nice GUIs with which to work.

The Standard Query Language

Many database products, ranging from Oracle to Access, allow you to use what is called SQL (*Structured Query Language*). SQL lets you do all sorts of neat things to add, modify, delete, and access data in a database. The cool part about it is that it uses a rather English-looking language to process the transactions you require.



Tables 21 Items

Name	Owner	Type	Create Date
AccountInformation	dbo	User	2/9/2001 7:08:37 PM
dtproperties	dbo	System	2/9/2001 7:08:00 PM
syscolumns	dbo	System	8/6/2000 1:29:12 AM
syscomments	dbo	System	8/6/2000 1:29:12 AM
sysdepends	dbo	System	8/6/2000 1:29:12 AM
sysfilegroups	dbo	System	8/6/2000 1:29:12 AM
sysfiles	dbo	System	8/6/2000 1:29:12 AM
sysfiles1	dbo	System	8/6/2000 1:29:12 AM
sysforeignkeys	dbo	System	8/6/2000 1:29:12 AM
sysfulltextcatalogs	dbo	System	8/6/2000 1:29:12 AM
sysfulltextnotify	dbo	System	8/6/2000 1:29:12 AM
sysindexes	dbo	System	8/6/2000 1:29:12 AM
sysindexkeys	dbo	System	8/6/2000 1:29:12 AM
sysmembers	dbo	System	8/6/2000 1:29:12 AM
sysobjects	dbo	System	8/6/2000 1:29:12 AM
syspermissions	dbo	System	8/6/2000 1:29:12 AM
sysproperties	dbo	System	8/6/2000 1:29:12 AM
sysprotects	dbo	System	8/6/2000 1:29:12 AM
sysreferences	dbo	System	8/6/2000 1:29:12 AM
systypes	dbo	System	8/6/2000 1:29:12 AM
sysusers	dbo	System	8/6/2000 1:29:12 AM

Figure 15.2

SQL Server's table view screen

Selecting Data with SQL

Suppose that you want to pull the account information for user account number 10045. To do this with the database you created earlier, you use the following SQL.

```
Select * from ContactInfo where AccountNumber = '10045';
```

The `select` keyword in the query tells the database that you want to retrieve data. The asterisk tells the database that you want to retrieve all of the columns in the table you are selecting. Using the `from` keyword, you indicate the table from which you want to pull the data. In this case, you want to pull from the ContactInfo table. The `where` clause lets you specify criteria for your search. In this example, you limit your retrieval to the record that has an account number of 10045.

Counting with SQL

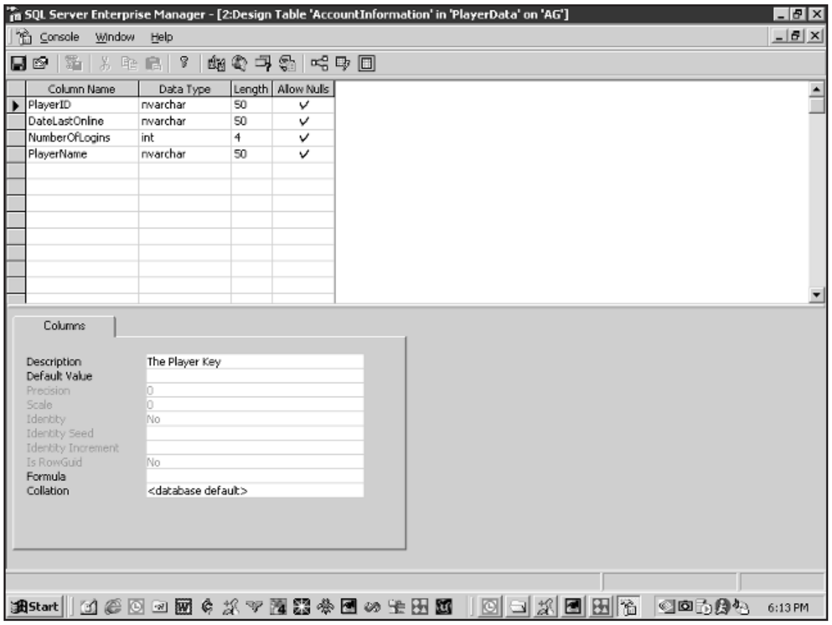
To really comprehend the power of the SQL syntax, you must play around with various tables of data. Take, for instance, a marketing campaign. Maybe you want to send a mail piece to people in the areas where most of your customers are. You figure that the people in these areas are likely to play your game, so you want to target them. To do this, you can issue a series of count requests to the database. The area code that has the highest count contains your greatest concentration of customers. To create a count request, you use the following syntax.

```
Select count(*) from ContactInfo where AreaCode = '501';
```

This query counts all of the records with an area code of 501 and returns the total to you. Not only can you use counting for marketing, but you can also use it to figure out the greatest concentrations of items in your game. Imagine that you have a table containing information about the items in your game world. This table might have data such as the item name, identifier, weight, type, zone, and owner. To figure out the number of items in zone 1, you would use the following query.

```
Select count(*) from ItemInfo where Zone = '1';
```

See how easy that is? If you take the time to think about it, SQL can be a very useful language. Imagine how long it would take you to code every possible query for a custom data solution. This is where the true value of an off-the-shelf product comes into play. If money is a concern, which it normally is, then try some of the free databases out there, such as MySQL. Figure 15.3 shows the table detail screen for SQL Server.



SQL Server Enterprise Manager - [2:Design Table 'AccountInformation' in 'PlayerData' on 'AG']

Column Name	Data Type	Length	Allow Nulls
PlayerID	nvarchar	50	✓
DateLastOnline	nvarchar	50	✓
NumberOfLogins	int	4	✓
PlayerName	nvarchar	50	✓

Columns

Description	The Player Key
Default Value	
Precision	0
Scale	0
Identity	No
Identity Seed	
Identity Increment	
Is RowGuid	No
Formula	
Collation	<database default>

Figure 15.3

SQL Server's table detail screen

Adding Data with SQL

Selecting and counting data is fun and all, but you really must have data present before you can mess around with it. This is where the `insert` command comes into play. In the previous example, suppose that you want to add an item to a zone. You do this using the following SQL.

```
Insert into ItemInfo set ItemName='sword', Identifier=9005,
 Weight=5, Type=10, Zone=1;
```

This SQL adds a sword to the item table. Using SQL, you don't have to worry about seeking to the end of the file, replacing out-of-date items, or anything else. You just call the insert function, and the database does the rest.

ODBC

Now that you have seen just some of the power available in an off-the-shelf database, you might be wondering how to access such a powerful tool. Well, you lucky Windows users have a thing called ODBC (*Open Database Connectivity*). What this means in human terms is that ODBC is a standard protocol for accessing relational databases that are based upon SQL.

When you install a package such as SQL Server and add a database, you also must create a pipeline to access the database. This is how you use ODBC. Once you create your database, you then create an ODBC connection to it within the Windows control panel. Take a peek at Figure 15.4 to see the ODBC icon in Windows 2000.

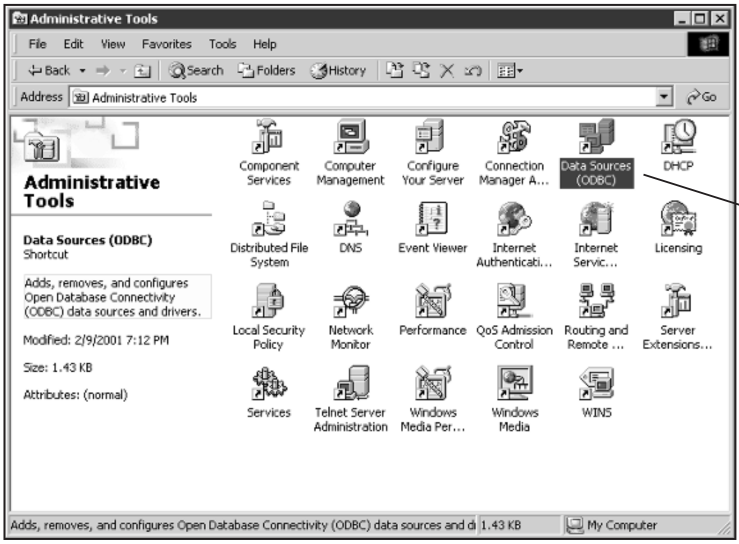
TIP

In Windows NT, you can find the ODBC icon in the top level of the control panel. In Windows 2000, you must look in Administrative Tools within the control panel.

Each SQL-compliant database that comes out usually has an ODBC driver that is installed when you install the package. To create a connection to a database you have created, you go into the ODBC panel and select the driver that you want to use for the connection. Once that is out of the way, you select the database to which you want to connect. You can do all sorts of fancy things here, such as set the user name and password for the database.

CAUTION

You should always select difficult user names and passwords for your databases. The last thing you want is some hacker out there gaining ODBC access to your game data.



Data Sources (ODBC)
Shortcut

Adds, removes, and configures Open Database Connectivity (ODBC) data sources and drivers.

Modified: 2/9/2001 7:12 PM

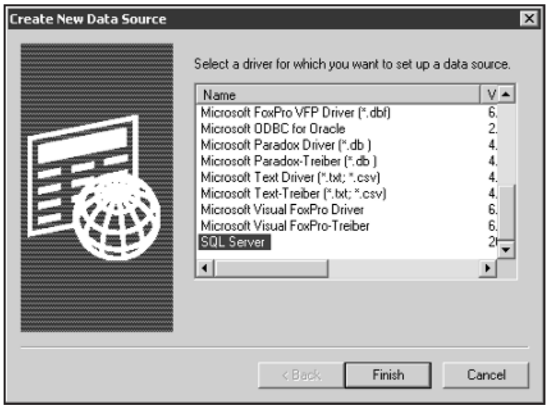
Size: 1.43 KB

Attributes: (normal)

Adds, removes, and configures Open Database Connectivity (ODBC) data sources and d 1.43 KB My Computer

Figure 15.4

The control panel view in Windows 2000



Select a driver for which you want to set up a data source.

Name	V
Microsoft FoxPro VFP Driver (*.dbf)	6.
Microsoft ODBC for Oracle	2.
Microsoft Paradox Driver (*.db)	4.
Microsoft Paradox-Treiber (*.db)	4.
Microsoft Text Driver (*.txt; *.csv)	4.
Microsoft Text-Treiber (*.txt; *.csv)	4.
Microsoft Visual FoxPro Driver	6.
Microsoft Visual FoxPro-Treiber	6.
SQL Server	2

Figure 15.5

The ODBC driver panel

In Figure 15.5, you can see the ODBC window that shows the available database drivers.

Data Hardware

Before you can ride off into the sunset with your newly found database knowledge, you must learn about data storage hardware design. You can just store your data on any old system, but you will not get the best performance possible. You must also contemplate several security issues.

Hardware Security

One mistake people make is housing their databases on the same machines that have access to the Internet. This is a huge security risk that you do not need to take. The reason this is a security risk is that if your data resides on a machine with Internet connectivity, anyone can try to connect directly to your databases.

Private Networking

One method of keeping your data safe is creating a private network within your public network. The network that can see the Internet is considered the public network. To create a private network within the public network, you give each machine in your network two Ethernet cards. One card goes to the Internet router/switch, and the other card goes to the internal network switch. For all practical purposes, each machine maintains two IP addresses. You can see this illustrated in Figure 15.6.

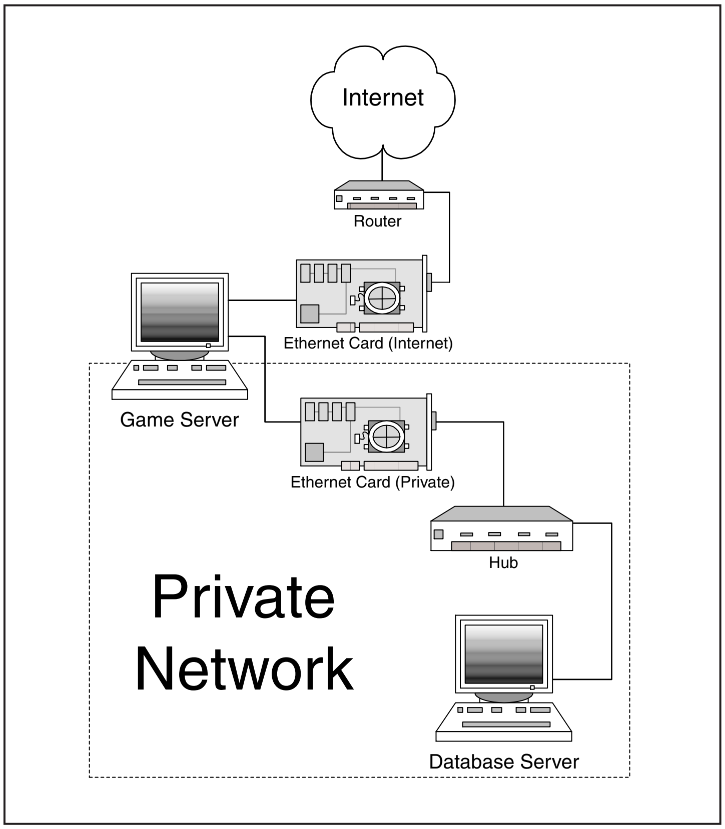


Figure 15.6

Dual NIC network setup

In Figure 15.6, you can see that people on the Internet can only see the public network. To see the private network, computers must be physically connected to the machine they want to access. There is no better protection than to require a physical connection to your data.

CAUTION

Even if your data resides in a private network, you should require a password to access it.

Data Integrity

Another problem with MMORPGs is that the game world data is very important and should never be lost. If you were playing a game of *Quake III* and the game crashed, it wouldn't be any big deal since player data is reset for every new game anyway. The players would all just reconnect and resume business as usual. If a game such as *EverQuest* crashed, you would have thousands of hacked-off customers because many of them have worked months or even years building up their characters. The worst scenario possible is if the game world crashes and loses all of the player data. If you didn't have backups, then you would probably be out of business. So as you can see, data integrity is a very important topic for massively multiplayer games.

Backups

You have several ways to preserve your data, the easiest one being daily backups of your databases. Even with daily backups, though, you still have the potential for players to lose all of their hard-earned work for an entire day. This is where replication software can help.

Replication Software

Replication software is designed in such a way that you can have multiple computers act as mirrors of one another. If they all have replication software installed, whenever a machine changes, they all do. In this case, you have replication software installed on your database machines. Each time data is updated or added to the database, it is replicated across all database servers. To see how the hardware would look, take a gander at Figure 15.7.

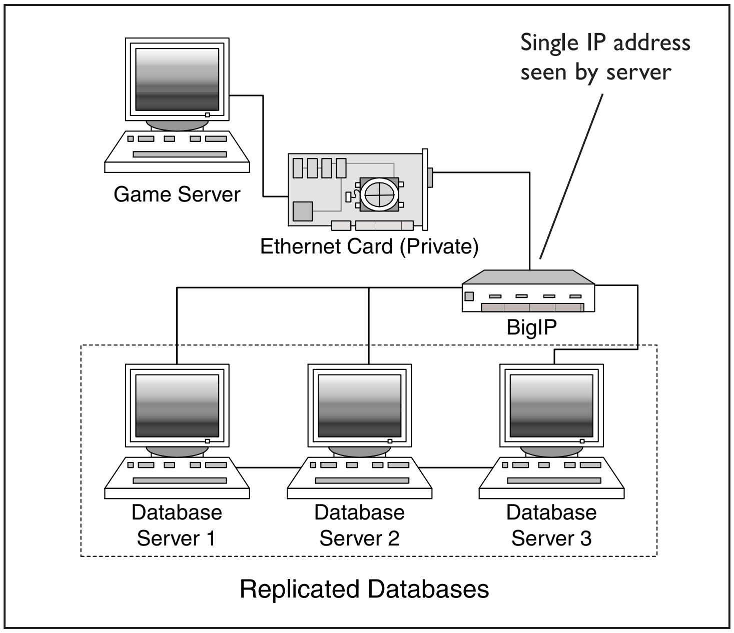


Figure 15.7

Replication software setup

Transaction Logging

Most databases out there today have the ability to keep transaction logs. A transaction log for a database keeps a record of all data inserted or updated to the system. It can also keep track of every read performed. This is helpful in that you can "back out" the data to a previous time. Imagine that in your game world, people found a way to duplicate gold. You might catch the bug four hours into the first use of it. Well, by the time four hours have passed, the game world probably has millions of duplicated gold stashes. The problem this causes is that your entire in-game economy is out of balance. Rather than resort to a previous backup that might be 12 hours old, you can backtrack the transaction log and revert everything that happened to the database in the last four hours. This is where you see the true power of the transaction log. Since the logs are constantly running, you can revert the game world to any previous state that day without having to go all the way back to the daily backup. The main disadvantage of transaction logs is that they consume tons of storage space. You will probably want to clean out the logs every day.

Fail Over

Now that you have your replication system in place, you need to plan for hardware failure. One problem arises when a complete database computer goes down. How do you handle it? Well, you can use a load balancer such as BigIP. The BigIP hardware system lets you hook multiple computers to it, and they all have the same IP address to the outside world.

You can program the hardware to use one machine as the primary system. If that one machine goes down, then the BigIP automatically routes traffic to the next system in line. Because all of your databases are replicated, it doesn't really matter which machine is picked next. The beauty of this system is that the servers that connect to the database only have to know one IP address, that of the BigIP. The best way for you to visualize this is with a figure, so take a look at Figure 15.8.

The main drawback to the BigIP solution is that it is very expensive. But if you haven't figured it out yet, massively multiplayer games are expensive to start up and operate.

NOTE

One of the more expensive parts of maintaining an MMORPG tends to fall into the bandwidth category. You can expect to pay several thousand dollars a month for the Internet connection alone.

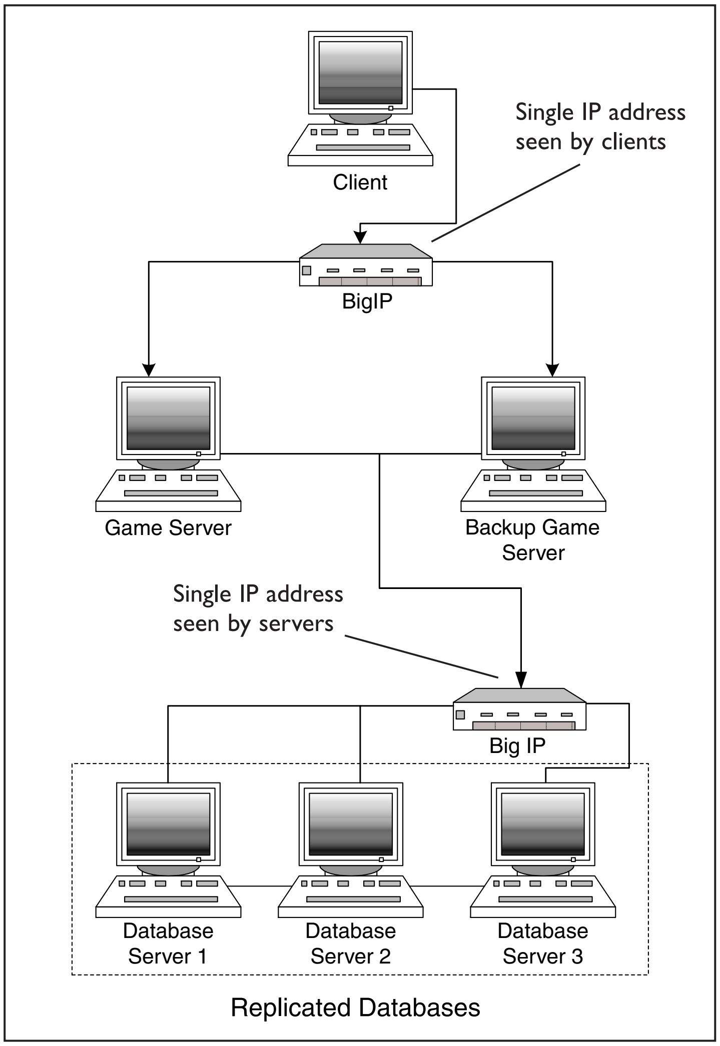


Figure 15.8

BigIP hardware setup

Offsite Storage

This topic might sound like an obvious tip, but you would be surprised how many people out there don't keep offsite backups. Offsite storage refers to the practice of keeping an alternate database system running at all times. You actually maintain the alternate system in a different physical location from the primary systems. This type of setup helps in case your building burns down, gets blown away by a tornado, or meets any other disaster. This isn't something that many companies think about, but you are asking for trouble if you don't plan for it.

The cheapest method is to keep backups of your data off-site. This allows you to at least maintain world data integrity from the previous 24 hours. Your game might be down for a very long time, but you do not lose the data.

CAUTION

Your number-one concern should be your world data. If you ever have to reset the player world, you are guaranteed to lose several customers, so that is the last thing you would ever want to do in a persistent-world game.

Every Player Is a Hero

Unlike single-player games, every player in an MMORPG must have the ability to be a hero. You don't want to create adventures and quests where the first player to solve it is the only hero in the game. Most quests and adventures should be solvable by every player in the game. One of the problems with such a system is how you maintain balance while keeping it fun for everyone. Just imagine if you created a quest where the end goal was freeing some prisoner. Well, this goal wouldn't work very well if someone had already freed the prisoner. Situations such as this can be difficult to handle properly.

The main game that comes to mind in the area of quests is *Asheron's Call*, which has the most in-depth quest system I have ever experienced. Not only does everyone have a chance to solve the quests, but new quests are introduced every month. This is what keeps players coming back for more.

Repeatable Quests

The lesson to learn here is that you need to design your game in such a way that you can add new quests on a regular basis. You must also allow the quests to be solved over and over again. One problem this creates is quest *camping*.

Camping **refers to players who sit at one spot in the game to either kill a certain spawning creature or retrieve a spawning treasure. This is usually considered very rude behavior.**

Quest Timers

To prevent camping, some games use timers. The timers prevent players from solving the quest over again within too short a period of time. A quest in *Asheron's Call* lets players get a special hilt. The hilt, called a Bandit Hilt, lets you modify one of your light weapons to make double hits every time. Because this is a very powerful quest item, it has a four-month timer. Once you solve the quest and get the item, you must wait four real months to solve it again. I would like to see the goof who tries to camp that one!

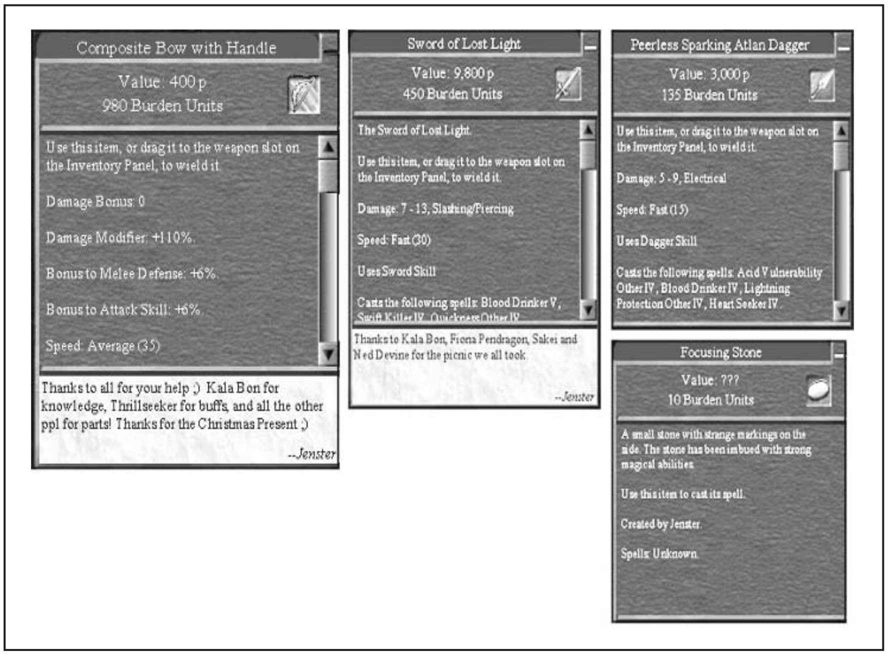
Spawnable Items

The normal way to allow for a repeatable quest is to let the end goal be a magic item or special reward. An example for a magic caster quest is a quest in which the final item is a powerful magic wand. There is no reason every magic user in the game can't have the wand, so you make the quest repeatable.

TIP

Make sure that you do not create quest items that are too powerful. This can lead to a major imbalance of power.

The problem you run into with recurring items is that people might literally line up to get them. This can be very boring for the players who have completed a quest and must wait in line to get the reward. An example of this is a dungeon in which you must fight several creatures to get to the end. Suppose that once you get past all the evil minions, the treasure chest at the end contains a special quest item. Well, you would probably end up with dozens of people standing around at the end waiting for the treasure to appear. The lesson to learn here is to let quest items regenerate every few seconds. As long as you put timers on each quest, you will not have a problem. To see some of the quest items in *Asheron's Call*, look at Figure 15.9.


Composite Bow with Handle

Value: 400 p
980 Burden Units

Use this item, or drag it to the weapon slot on the Inventory Panel, to wield it.

Damage Bonus: 0

Damage Modifier: +110%.

Bonus to Melee Defense: +6%.

Bonus to Attack Skill: +6%.

Speed: Average (35)

Thanks to all for your help ;) Kala Bon for knowledge, Thrillseeker for buffs, and all the other ppl for parts! Thanks for the Christmas Present ;)

--*Jenster*

Sword of Lost Light

Value: 9,800 p
450 Burden Units

The Sword of Lost Light.

Use this item, or drag it to the weapon slot on the Inventory Panel, to wield it.

Damage: 7 - 13, Slashing/Piercing

Speed: Fast (30)

Uses Sword Skill

Casts the following spells: Blood Drinker V, Swift Killer IV, Quickness Other IV

Thanks to Kala Bon, Fiona Pendragon, Sakei and Ned Devine for the picnic we all took.

--*Jenster*

Peerless Sparking Atlan Dagger

Value: 3,000 p
135 Burden Units

Use this item, or drag it to the weapon slot on the Inventory Panel, to wield it.

Damage: 5 - 9, Electrical

Speed: Fast (15)

Uses Dagger Skill

Casts the following spells: Acid Vulnerability Other IV, Blood Drinker IV, Lightning Protection Other IV, Heart Seeker IV.

Focusing Stone

Value: ???
10 Burden Units

A small stone with strange markings on the side. The stone has been imbued with strong magical abilities.

Use this item to cast its spell.

Created by Jenster.

Spells: Unknown.

Figure 15.9

A few quest items in Asheron's Call

One-Time Quests

Now, don't get me wrong: Not every quest should be repeatable. To maintain a sense of competition, you might want to create special quest events that can only be accomplished once. For these quests, you should recognize the victor and remind players of their accomplishments for the life of the game. For example, you might add a banner to a town sign that recognizes winners' actions. This isn't too hard to do, and it encourages others in the game to win one-time quests.

Boasting Rights

One pitfall that some MMORPGs fall into is making the game too difficult for new players. No matter what the level or skill of the players, they should have something to do to get a sense of accomplishment. The lesson here is that you should create quests for all ranges of skill levels.

Of course, with every good thing, there is bad. One problem that arises with newbie quests is the arrival of skilled players in the quest area. Sometimes, you get very skilled players who rush in and solve the easy quests just to aggravate people. I know this sounds very immature, but it happens. To combat this, you should put both minimum and maximum skill restrictions on the quests. A common method is to prevent certain level characters from even entering the quest area. *Asheron's Call* uses this method effectively—higher-level characters can't go into some low-level areas.

Visual Rewards

Not only do players like to boast about their accomplishments, but they also want something to show for them. You should seriously consider creating quest items that stand out visually. You know, what fun is that new fire sword you just got if it looks like any other sword? The king of the offenders in this category is *Ultima Online*. Only recently did they start to add items to the game that look somewhat interesting. Even now, most magic items in the game look identical to their normal counterparts.

Dynamic Questing System

As far as game design goes, you should design your game engine in such a way that you can add and remove quests from the game without any code changes. Normally, this requires an extensive scripting system, but in the end, you are better off. The scripting system can even resemble code-like instructions, but it should allow for dynamic loading of new scripts as you create them.

Player Communities

One thing that always occurs in MMORPGs is the evolution of player communities. Most people like to amass into groups, so you'll see people join together and form virtual communities within your game. The key to benefiting from this phenomenon is encouraging community development.

Guilds

In most medieval adventure-type MMORPGs, the games allow for the creation of guilds. A *guild* is a sort of club in which like-minded players join together. For instance, you might have a thieves guild or a mages guild. The types of guilds really don't matter as long as like-minded players have the chance to band together.

One way to encourage guilds is to provide a formal in-game system in which players can create, disband, and modify their own guilds. *Ultima Online* accomplishes this with guild stones. In *UO*, you can purchase a guild stone for around 10,000 gold pieces. Once a player has a guild stone, he can invite other players to join his guild. He can also declare war on other guilds in the game. This allows for a certain group mentality that just wouldn't be possible without formal clubs.

Allegiance Systems

Another popular method for communities is allegiance systems. *Asheron's Call* employs this method quite effectively. In its allegiance system, players can swear allegiance to other players. Each time someone swears allegiance to you, he becomes your vassal. From that point on, whenever he gets experience points, you get a little extra for being his patron. By giving patrons experience points, the game encourages people to form alliances. It really boils down to a pyramid scheme, but it does make for interesting play. To see how the pyramid works, look at Figure 15.10.

Player Towns

Ultima Online actually lets players create their own towns. A player-run town is really nothing more than a collection of player-owned housing. To add to the ambience of the collection, the game masters in *UO* placed permanent decorations around the area to make it look like a real town. This method is the ultimate example of a player community. What better way to bring players together than to let them create their own virtual village?

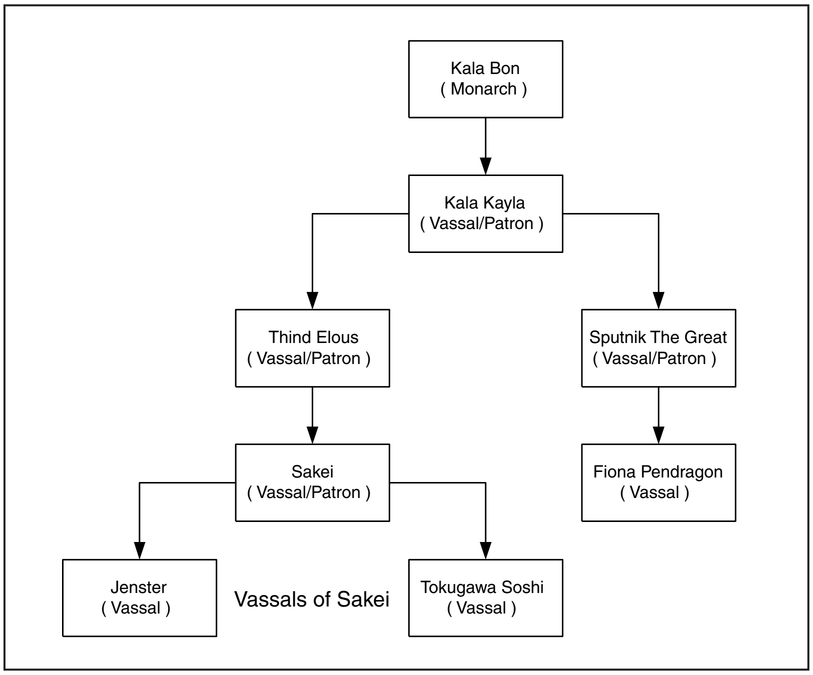


Figure 15.10

The allegiance system in Asheron's Call

The Good, the Bad, and the PK

As I have mentioned before, some games let people kill each other in the game. This can lead to all sorts of anti-social behavior, and if it is left unchecked, it can wreck your entire game. If you must have player-versus-player combat, then I recommend that you set up dedicated areas where players can fight one another. I do not recommend having open season on all players. *Ultima Online* suffered from this, and many people quit the game. Origin finally came to its senses and created a duplicate world in which player killing is not allowed. The funny part is that everyone complained that the game would be ruined without player killing. Well, guess what: The old PK lands are practically deserted now, and the new, non-PK lands have more players than ever.

On the other hand, player killing can be an enjoyable experience for players. So you might want to plan how to allow player killing for those who want to participate. The main thing I suggest is making it voluntary. Only let players fight other players who have agreed to participate in that aspect of gaming. Some MMORPGs require that players complete a PK quest to be able to fight other PKs. This ensures that only PKs fight other PKs.

Player Advancement

One thing common to all MMORPGs is that players get better with time. This is a key success factor to the genre. If players don't get any better over time, then why should they keep coming back for more? The only real downside to this method of gameplay is that you usually have to take out any physical skill factor. By physical skill factor, I mean the skill it takes to play games such as *Quake* and *Unreal*. In those games, some players are physically better at playing the games due to a mixture of practice and reflexes.

In MMORPGs, players get better virtually over time. By this, I mean that their in-game characters develop better skills and abilities as they play. Some physical skill is required to play, but not as much as in other games.

Practice Makes Perfect

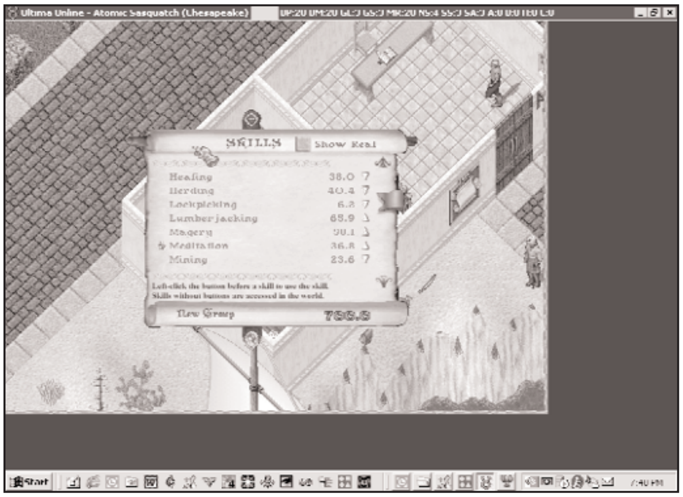
One common way to allow for character advancement is through practicing skills. For example, take a fighter's ability to use a sword. Each time the fighter uses his sword, his skill increases a little bit. Over time, the fighter gets better at using his sword. With better sword skill, the fighter has a better chance of hitting creatures with every swing.

Measured Advancement

A good rule to follow is to let players see how they are advancing. The last thing you want to do is keep players in the dark about how they are advancing in a skill. Suppose you only give skill gains in whole points. The system might be keeping track of the decimal place, but the player's actual skill is based upon whole numbers. For such a system, you want to at least show the player the decimal place for his skill. *UO* made this mistake early on, and it frustrated many players. The problem was that it could take weeks for a single skill point to go up. Because players could not see the decimal place, they had to guess when the skill would finally increase a point. Origin finally added the decimal place to the skill window, and now players can see how they are progressing. Figure 15.11 shows the *UO* skill window.

Advancement Units

Players must be able to measure their advancement, but they also need something by which to measure. Normally, this takes the form of experience points, levels, or skill points. You really want players to be able to see a numerical representation of their skill gains. If they have nothing tangible to represent their efforts, then they'll



Healing	38.0
Herding	40.4
Lockpicking	6.2
Lumberjacking	65.9
Magery	90.1
Meditation	36.8
Mining	23.6

Left-click the button before a skill to use the skill.
Skills without buttons are accessed in the world.

New Group 766.8

Figure 15.11

The skill window in Ultima Online (©2000 Electronic Arts. All rights reserved.)

get discouraged. The ultimate solution is for players to actually change in physical appearance according to their ability. Just think how cool it would be for a fighter to appear more muscular as his strength increases. I think that with the advent of 3D graphics, this feature is not too far off.

Item Ownership

I touched upon the subject of ownership in previous chapters, and it is a very important part of the MMORPG experience. All you have to do is look at eBay to see how much people value virtual goods. Players in MMORPGs want stuff to play with. A good rule of thumb is to let players buy only the essentials at in-game stores. They should have to adventure to get good items. By doing this, you give a sense of accomplishment to those who obtain the good equipment. You will also find that players in the game trade items extensively.

Look, Shiny Objects

As strange as it might seem, players of MMORPGs like to have all of their equipment color-coded. You will see players go to extreme lengths to acquire a whole suit of black plate-mail. A good trick to keep players coming back for more is to color items in the game. *Ultima Online* did this first by giving players the ability to dye their clothing any color they wanted. *Asheron's Call* came about and offered the same thing with dye plants. They made a quest out of getting stuff dyed different colors. In *AC,* players have to collect colored plants. Once they have enough of a certain color, they take them to an alchemist and cook to convert them to dye. The cook then applies the dye to the desired piece of armor. As long as the cook's skill is high enough, the player ends up with colored armor.

The bottom line here is that you should plan to let characters create their looks in the game. *EverQuest* suffers from a lack of options, and it shows because many players in the game look identical to one another. I think the main problem there is limited texture memory. When you design a 3D world, you must keep in mind the numbers of textures present onscreen at once. You can't go over the limit without slowing down the game engine.

TIP

You should create a texture manager in your 3D games to manage priority textures in the game. When textures are not in use, they should be unloaded to make room for active ones.

Server Architecture

One common question I see on game-development forums concerns what kind of machine is required to operate an MMORPG. Well, my answer is several powerful machines. You can't run an MMORPG off your everyday PC. You must use powerful servers if you hope to host a good number of players.

Server Specifications

Although production-ready MMORPGs require large servers, you can prototype one on a relatively cheap machine. I recommend that you at least purchase the following hardware for your first server.

- Dual 1GHz processors
- 1GB of RAM
- 40GB of SCSI storage
- Two 100MBit Ethernet cards
- Nvidia MX dual-head card

This system might set you back a little more than $1,000, but it is well worth it. Of course, by the time you read this, you might want to up the specifications. The bottom line is that you should buy the largest machine you can afford.

The reason I recommend the dual-head video card is that it supports two monitors. The game server does not actually view the game, but you can use it to view all aspects of the game, such as database information. You will find that the game server can have several information screens open at once. With two monitors, you can view it all at once.

Internet Connection

As great as DSL or ADSL might seem, it really is not suited to host an MMORPG. You need a T-3 line or better to host the actual game servers. In fact, you probably need several T-3 lines in the end. Because these types of lines can be very expensive, I suggest that you work out a deal with a local ISP. It might let you put your own server at its place of business for a piece of the action. I have found ISPs that let me house a server on their Internet backbones for as little as $250 per month.

One good thing about MMORPGs is that you can spread them out across multiple servers. By doing this, you can put each server on its own Internet connection. This might let you get by with a lower cost than having to lease one huge line.

Server Layout

One question you might be asking is why you would need multiple game servers. Well, you want to distribute the load across multiple machines. In MMORPGs, it is not unheard of to have thousands of players playing at the same time. It is unlikely that you could handle all of them on one computer, so you must distribute the players across multiple servers.

Zone Servers

As I mentioned in previous chapters, a common way to distribute players is by using zones. The *RPGQuest* game in Chapter 16, "Writing the RPGQuest Game Server," uses a zone method even though there is only one game server. The purpose of the zone method is so you can house each zone on its own server if you want. Whenever you want to add new areas to your game, you just add a new server to handle it. This can get to be quite expensive over time, though, because you will eventually have too much overhead to add new places in the game. To see how you can lay out zone servers, take a look at Figure 15.12.

Game Cells

Another method for handling the distribution of players is creating a cell system of servers. The game world is divided into hundreds of cells. Within each cell are *n* number of players at one time. Each server is given *n* number of cells to maintain. If

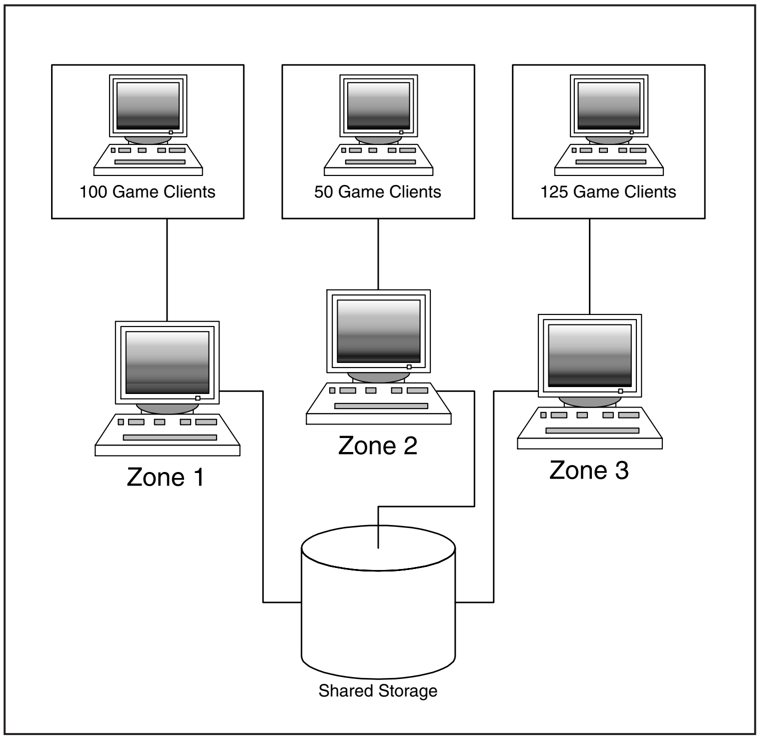


Figure 15.12

Zone server layout

a game cell gets too many players in it, the cell is divided into two, and half the players are sent to another server. All of this happens behind the scenes and without the players' knowledge.

The game cell method is pretty cool because each game cell is dynamically handled by a particular game server. If a game server gets too many loaded-down cells, it can transfer some of its cells to another server. You can check out one company that claims to be using a similar method, at http://www.bigworldgames.com.

Server Security

All of your game servers should sit behind at least one firewall. You should only open up the ports that are absolutely necessary to host the game. This rule is very important because a single hacker can financially ruin you. You should also follow some other rules to prevent security problems.

- Disable pinging to your server; this keeps people from slamming your server with too many ping requests.

- Keep your operating system up to date with the latest security patches. It surprises me to see how many server administrators fail to follow this simple rule.
- Do not share any folders or drives that reside on your server. You should not have any shared directories open on your server.
- Do not store any game data on the main Internet server. You should keep all game data stored on servers within your private network.
- Use complicated passwords. Never use user names such as SA. I recommend that you use the most complicated password you can think of. You should mix uppercase and lowercase with numerical characters as well. Also, be sure to change your password at least once per week.

I cannot stress enough how important it is to be vigilant in the area of server security. The day your game becomes popular, you will have tons of immoral hackers trying to take it down. I recommend that you purchase several books on security because I cannot even begin to scratch the surface here.

TIP

The Security Focus Web site offers great articles and utilities regarding system security. Visit it at http://www.securityfocus.com for more information.

Data Stream Security

Not only must you physically protect your game servers, but you must also protect the data stream between the server and its clients. One problem with MMORPGs is that many game players will try to hack into the data stream. They invent all sorts of neat utility programs to intercept game packets and mess with them.

You can follow simple or even complex methods to prevent the hacking of your data stream. One easy method is to bit-shift the data within packets. This makes it a little difficult to interpret. Because hackers have to reverse-engineer your game packets, you can at least make it hard for them to recognize patterns in the data.

Another method to maintain data-stream security is to actually encrypt the packet data. Many utilities out there will let you encrypt data, and they can be quite useful. The main problem with packet encryption is that it slows things down. Not only does the server have to encrypt tons of data, but also the player ends up having to decrypt it on his end.

The best rule for data-stream security is to not rely on the data stream for integrity. You should never let a player tell the server something and take it at face value. If a player sends me his coordinates, I am not going to accept them blindly. I first check to see whether he should be in that location before I accept it. The bottom line here is that the best protection is to not trust the client at all.

Player Validation

Because players for any type of MMORPG have to establish an initial connection to the game, you need a place for them to connect. Usually, you don't want them to connect to any of the game servers. Instead, you can create a central place for them to connect. The place where players establish their initial connection to a game world is called the login server.

The login server's role is to accept new player connections, validate their account information, and send them to the appropriate game server. By letting the login server handle connections, your game servers can concentrate on actual game events. The other benefit to having a login server is that players don't have to know the IP addresses for the game servers; they only have to know the address of the login server. You can see this illustrated in Figure 15.13.

Client Architecture

The game clients in MMORPGs have to take care of all the graphical elements in the game. Because game servers don't deal with graphics, the client is on its own in this department. Clients are really not that complicated when it comes to MMORPGs. The main reason is that they only have to worry about one thing, the game server. All information, whether request or response, is sent through the game server.

Action Validation

Take, for example, when the player wants to move forward. The client doesn't actually move the player forward; rather, it requests a move forward from the server. If the server accepts the movement, the player gets to move. Sometimes, it is a good idea to let the player start moving anyway; you just might pop him back into place if he tries to go to an illegal location.

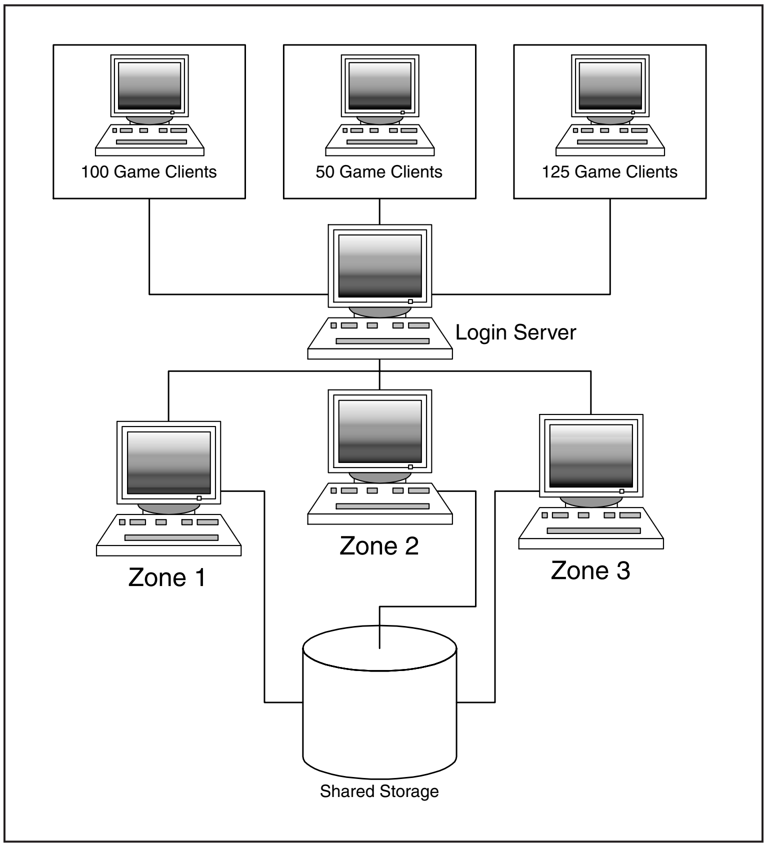


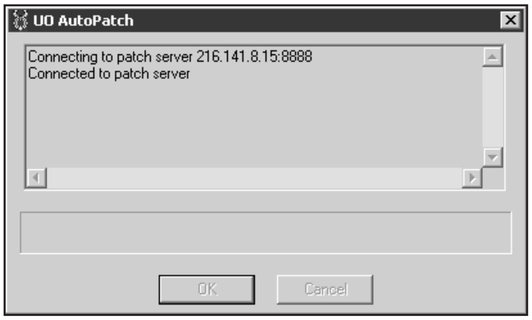
Figure 15.13

The login server architecture

The reason MMORPGs are designed this way is that clients cannot be trusted. It's unfortunate that human nature forces this type of system, but it is a necessary precaution.

Dynamic Patching

One cool feature about online games is that you will see a lot of game patching. Because MMORPGs are dynamic systems, client software must be updated throughout the lifespans of the games. When you design a game client, you should incorporate a patching mechanism. The first thing the client should do is check for updated client software. If a new patch is available, then the client downloads the patch and upgrades itself. This allows for some cool stuff because you don't have to worry about new updates; the client automatically downloads them. Look at Figure 15.14 to see *Ultima Online*'s patch GUI.



```
Connecting to patch server 216.141.8.15:8888
Connected to patch server
```

Figure 15.14

Patching the game client in Ultima Online

Client Portability

I recommend that you create your game clients in such a way that they can be ported to multiple operating systems. Because all of the hard stuff is handled by the servers, clients shouldn't be too difficult to design in a nice, clean, object-oriented fashion.

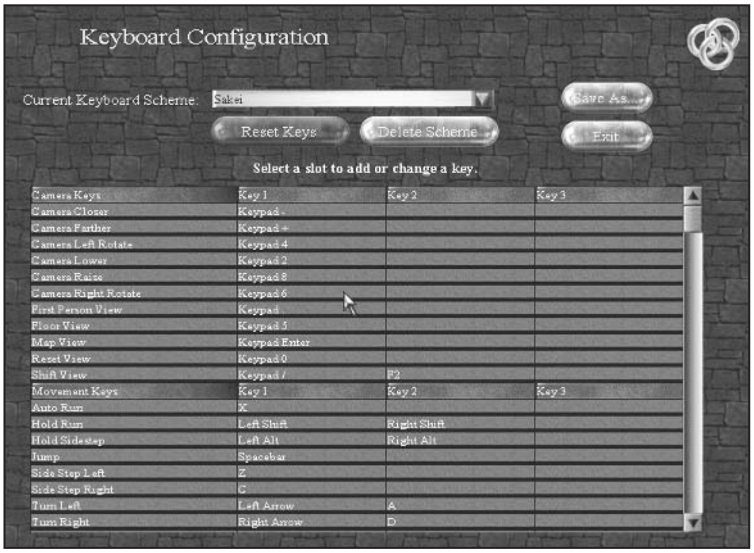
Multiple Clients

Another cool part about MMORPGs is that you could technically have different types of clients out there. Just take a look at *Ultima Online Third Dawn* to see this effect. It has one 3D client, but another client is 2D-based. Because the servers only care about the data stream, it doesn't matter what the client looks like. As long as the data transfer between client and server is compatible, it doesn't matter which operating system or graphics system is present on the client.

The User Interface

It is interesting to note that most MMORPGs have complicated user interfaces. Some could attribute it to the depth of the genre; others might attribute it to bad design. Personally, I never mind a complicated display as long as it doesn't distract from gameplay.

A good rule to follow is that if you are designing a massively multiplayer game for consoles, then you want a very simple interface. If you are designing for personal computers, then you can get away with more elements. No matter what you do, though, you should let players configure their displays to their liking. At the very least, let players set their own keyboard configuration. Figure 15.15 shows how *Asheron's Call* does this.



Current Keyboard Scheme: Sakei

Select a slot to add or change a key.

Camera Keys	Key 1	Key 2	Key 3
Camera Closer	Keypad -		
Camera Farther	Keypad +		
Camera Left Rotate	Keypad 4		
Camera Lower	Keypad 2		
Camera Raise	Keypad 8		
Camera Right Rotate	Keypad 6		
First Person View	Keypad .		
Floor View	Keypad 5		
Map View	Keypad Enter		
Reset View	Keypad 0		
Shift View	Keypad /	F2	
Movement Keys	Key 1	Key 2	Key 3
Auto Run	X		
Hold Run	Left Shift	Right Shift	
Hold Sidestep	Left Alt	Right Alt	
Jump	Spacebar		
Side Step Left	Z		
Side Step Right	C		
Turn Left	Left Arrow	A	
Turn Right	Right Arrow	D	

Figure 15.15

The keyboard configuration screen in Asheron's Call

Review

In this chapter, you have learned some of the key elements to designing and implementing an MMORPG. Some of the key points are as follows.

- Off-the-shelf database packages can save you time and money for your data storage solution.
- Always maintain a robust backup system for your game world. Customers will not put up with losing hard-earned progress.
- Pre-plan your network architecture. Be sure to include several layers of security between your game data and the outside world.
- Design your game world around a scripting system. This allows you to add content dynamically to the game world, which will keep players interested.
- Architect your load-balancing schema early on to avoid problems when your game becomes popular.
- Be sure to reward players for their time investment in the game. Players should see a noticeable difference as their characters gain experience and power.
- Validate all critical player actions. When your game is popular, people will try to hack into it.

Well, it has been a shorter visit than usual, but you are finished with this chapter. In the next two chapters, you will see the beginnings of an MMORPG. See you soon!

CHAPTER 16

Writing the RPGQuest Game Server

Now that you have seen the basics of game-server design, you can begin to write your own applications. In this chapter, you will learn how to write the server for a client/server-based persistent-world game. In this case, the game is called *RPGQuest*. It is a simple game system in that it allows players to create accounts, log in, and run around the game world. When a player leaves, his position and orientation are saved for the next time he logs in. The only rule in the game is that players cannot go outside of the arena boundaries, and they cannot go through obstructions.

I will discuss the following points in this chapter.

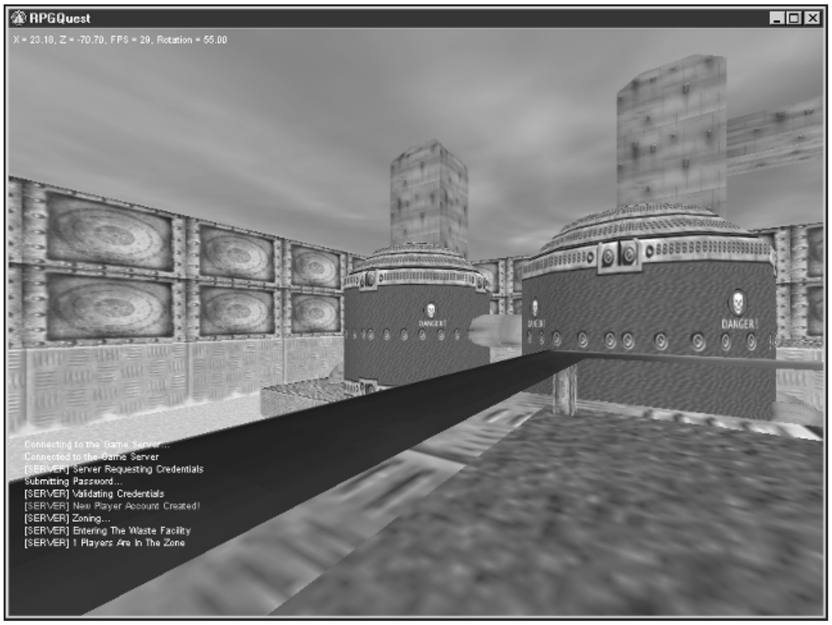
- How to set up the project space
- Game play mechanics
- Account authorization and storage
- Message queuing
- Processing player messages
- Updating time of day
- Updating player data

The RPGQuestGameServer project, located in the Chapter16 Zip file on the book's CD, contains all of the files necessary for the *RPGQuest* server application. Go ahead and load it up now so you can get started with the code. To see the client's view of *RPGQuest*, look at Figure 16.1.

The Game Project Space

The project space for the *RPGQuest* game server is rather simple considering the server doesn't have to handle the graphics for the game. The only two unusual libraries required are winmm.lib and dplay.lib. The required list of files can be seen in Table 16.1.

Before I go into the code too much, I think you need to know what *RPGQuest* is all about.



```
X = 23.18, Z = -70.70, FPS = 29, Rotation = 55.00
```

```
Connecting to the Game Server...
Connected to the Game Server
[SERVER] Server Requesting Credentials
Submitting Password...
[SERVER] Validating Credentials
[SERVER] New Player Account Created!
[SERVER] Zoning...
[SERVER] Entering The Waste Facility
[SERVER] 1 Players Are In The Zone
```

Figure 16.1

RPGQuest *from the client's view*

Table 16.1 RPGQuest Project Files

File	Description
RPGQuestGameServer.cpp	Main program
RPGQuestGameServer.h	Main header file
dxutil.cpp	Utility code file
dxutil.h	Utility header file
dplay.lib	DirectPlay library file
winmm.lib	Windows multimedia library, used for `timeGetTime()`

RPGQuest Gameplay

The title for the game might be a little deceiving because there is no role-playing present, but nothing is keeping you from adding some! Basically, *RPGQuest* has three zones. Each zone is accessible through a doorway. When players get close enough to one of the doors, they are sent to the appropriate zone.

Players who connect to the game can see other connected players via their avatars. An *avatar* in a game is nothing more than an object representing another player. In *RPGQuest*, each player looks like a floating cube-like head. Check out Figure 16.2 to see the player avatar.

You might be wondering why I chose a floating cube head. Well, as it turns out, the space marine I'd originally planned involved an additional 3,000 polygons in the scene. This brought some machines to their knees. So, I resorted to a super-low model with only enough faces to form a cube. The good part is that the rest of the graphics look pretty decent.

The Zones

As I said earlier, the game has three zones. In each zone, you must keep track of obstacles and portals to other zones. This is important because as the server, you

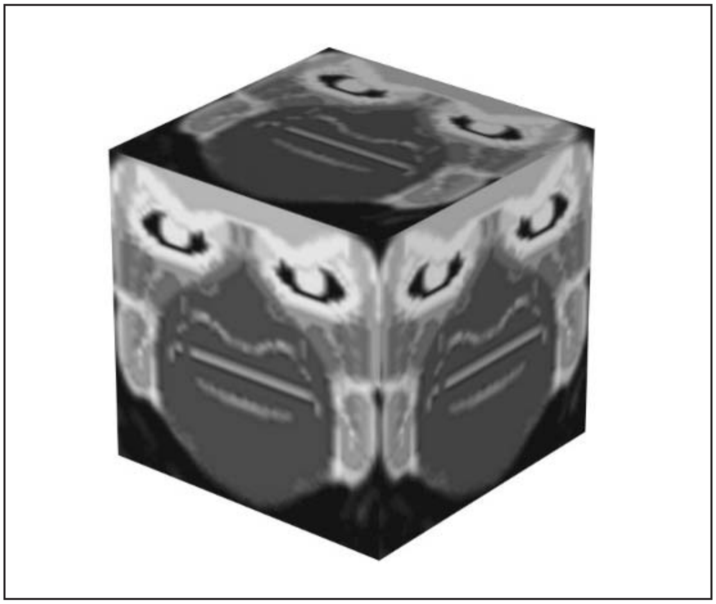


Figure 16.2

The player's avatar in RPGQuest

must confirm or deny player movement. Currently, the code uses standard bounding boxes to determine player collision with the scenery. A production game would probably use a more precise method such as vertex-to-vertex collision detection, but it gets the point across well enough here. Look at Figure 16.3 to see the collision areas in the first game zone.

Not only does each zone have its own set of collision boxes, but each zone also has its own 3D data. I created the zones in 3D Studio MAX and then saved them to x-files. Figure 16.4 shows how the first zone looks in the 3D program.

Account Verification

RPGQuest uses a flat-file system for game-data storage. Each player has his own file that is stored in the PlayerData folder. If a player connects with a user name that is not known to the server, a new account is created for that player. If a player connects to the game with a registered account, then the password is checked versus the one saved for that player. If his password fails, then he is not given access to the game.

The player data files also store the last position of the player. Whenever a player logs out of the game, his position is saved so that he can resume where he left off.

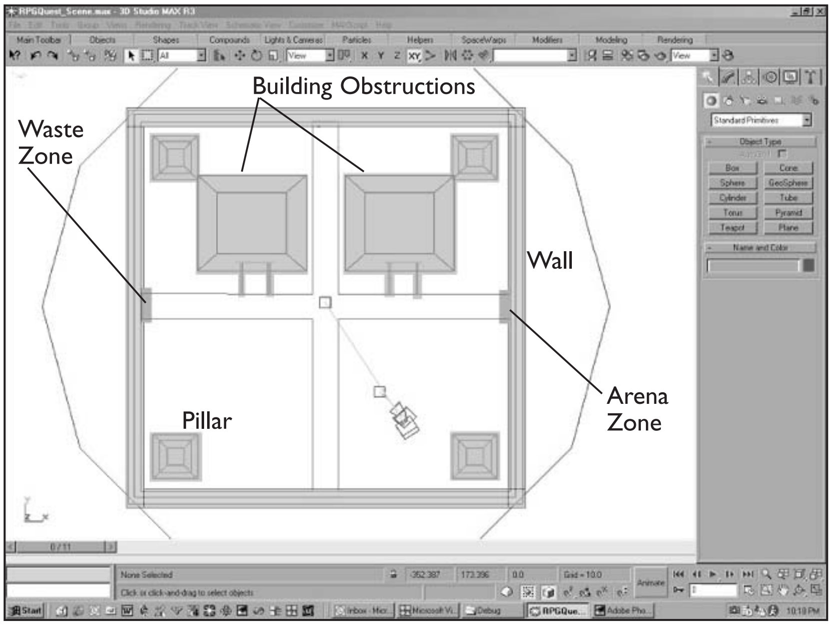


Figure 16.3

Collision areas in the merchant zone

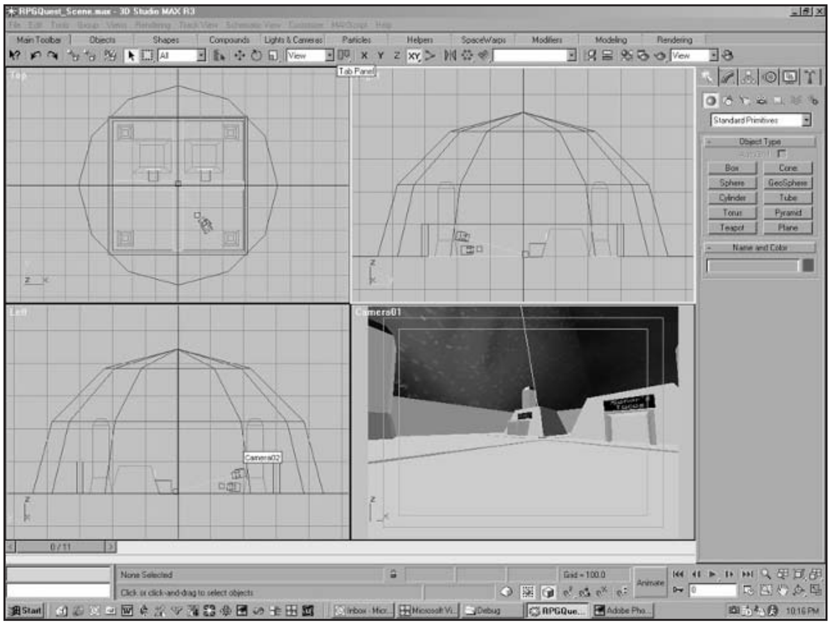


Figure 16.4

The merchant zone in 3D Studio MAX

I suggest that you experiment with the player files by adding new and interesting information to store. Perhaps you can add attributes and skill levels. Take a look at Table 16.2 to see the current layout of the player data file.

Table 16.2 Player Data File Layout

Data	Type
Player rotation	Floating-point value, in degrees, stored in the `fRot` data member
Player X coordinate	Floating-point value stored in the `vecCurPos` data member
Player Y coordinate	Floating-point value stored in the `vecCurPos` data member
Player Z coordinate	Floating-point value stored in the `vecCurPos` data member
Player zone	Integer stored in the `iZone` data member
Player password	Character buffers stored in the `szPassword` data member

Player Commands

The controls for the game are pretty basic because all the player can do is move around the game zones. Because the server is responsible for all movement verification, all the client gets to send from a movement perspective are angle and velocity change requests. You can look at the `vProcessMessages()` function to see the various messages processed by the server.

From the player's point of view, there are a few steps involved in playing the game.

1. Connect to the server.
2. Submit password.
3. Receive credential confirmation.
4. Move around the game.

Program Structure

The game-server code is laid out very much like the code in the chat server program covered in Chapter 10, "Lions and Clients and Servers, Oh My!" The main difference with RPGQuestGameServer is that it does not react to messages immediately in the DirectPlay message-handling loop. Instead, the program adds them to a queue to be processed later with the `vAddMessage()`, `iGetMessage()`, and `vProcessMessages()` functions.

Message Queuing

The main reason messages are queued is that the DirectPlay handler does not accept messages while others are being processed. In a message-intensive application such as a game server, you want to spend as little time as possible handling messages. When you put the messages into another queue, they can be processed later without holding up current incoming messages.

The second reason messages are queued is that you would run into threading problems if you altered player data in the DirectPlay message handler. You see, the DirectPlay message handler runs in its own thread and can be called at any time. If you happened to be altering the player data when a message came through, the program might try to alter the

CAUTION

Always be very careful when modifying global data within a threaded function. If you do not handle it properly, you can crash the computer. The best practice is to use critical sections or mutexes whenever you modify global data.

player data at the same time. This can result in the blue screen of death on Windows machines.

Player Handling

Because all network messages are put into the message queue, the game server only has to worry about handling player data within the `vProcessMessages()` function. Within this function, the server retrieves all player commands and processes them.

The other area of interest for player processing is the `WinMain()` function. Within the main message handler, the game server calls the `vUpdatePlayers()` function. This function updates player positions and verifies player movement. If a player hits an obstruction or needs to go to another zone, it is handled within this function. To see how the game server is laid out, check out Figure 16.5.

As you can see from Figure 16.5, the server program does not have too many main components. Most of the code actually performs various player logic functions. For the most part, the game server is just like the chat server program. Of course, the game server has a lot more logic to it. Hey, I was just flipping through *Linux Game Programming* (Prima, 2001), and it is pretty cool. Maybe I'll try my hand at writing a Linux-based game after I'm done with this book. Oh, wait; my wife just reminded me that I have about 100 things to do around the house first. Oh, well. . . .

The Game-Server Code

Wow, doesn't that title say it all? "The Game-Server Code!" Okay, you can go home now; you are done here. What's that? You want to know how the code works? Oh, okay; here you go.

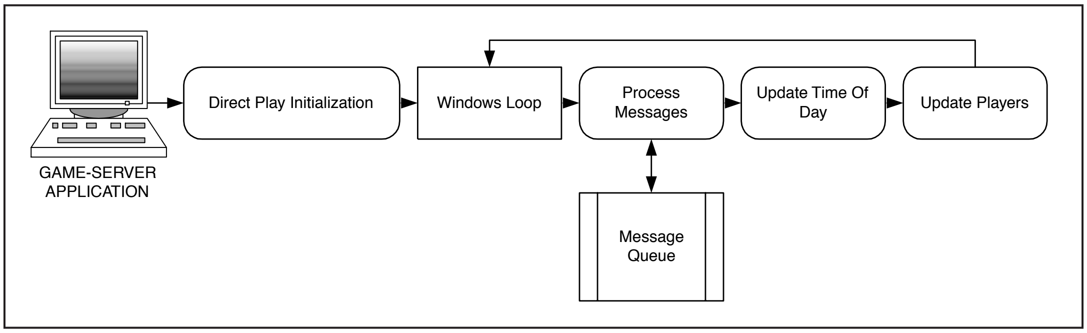


Figure 16.5 *Main components of the RPGQuestGameServer program*

The WinMain() Function

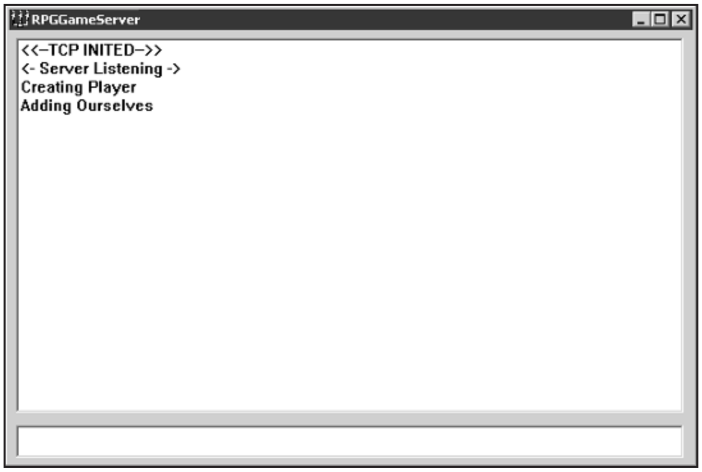
It all comes down to this, the `WinMain()` function. By this time, you should be very familiar with this function. The `WinMain()` function is the function required by Windows programs to handle the functionality for the program. Its equivalent in the rest of the world is the `Main()` function you see in most other types of programs. All right, enough nostalgia. For the game-server project, the `WinMain()` function accomplishes the following tasks.

1. Create the server window.
2. Initialize DirectPlay.
3. Host the game world.
4. Process game messages from clients.
5. Update world time of day.
6. Update player positions and settings.

Creating the Server Window

The server window is a pretty basic-looking application window whose main purpose is to display game-world information. You can look at Figure 16.6 to see the server window in action.

Looking at Figure 16.6, you can see that there are only two elements to the window, an output box and an input box. The input box is for future functionality where you might want to issue server commands. For now, you won't use it. The output



```
<<—TCP INITED—>>
<- Server Listening ->
Creating Player
Adding Ourselves
```

Figure 16.6

The game server window

box shows various server-side messages such as when players join or exit. The following vCreateInterface() function creates these two elements.

```
void vCreateInterface(HWND hWnd,HINSTANCE hInstance)
{
    // Output window
    hLB_Output = CreateWindowEx(
        WS_EX_CLIENTEDGE,
        "LISTBOX",
        NULL,
        WS_CHILD | WS_VISIBLE | LBS_NOTIFY | WS_VSCROLL | WS_BORDER,
        5, 5, 580, 340,
        hWnd,(HMENU)IDC_hLB_Output,hInstance,NULL);
    // Edit field to enter chat messages into
    hEB_InputField = CreateWindowEx(
        WS_EX_CLIENTEDGE,
        "EDIT",NULL,
        WS_CHILD | WS_VISIBLE | WS_BORDER | ES_LEFT | ES_AUTOVSCROLL |
        ES_MULTILINE,
        5, 340, 580, 30,
        hWnd,(HMENU)IDC_hEB_InputField,hInstance,NULL);
}
```

As you can see, this code creates two Windows elements, a LISTBOX and an EDIT box. There is nothing very special here, just your run-of-the-mill Windows controls.

Initializing DirectPlay

The next task is initializing DirectPlay. This step should also be familiar to you by now because it follows the pattern of its predecessor, the chat server program. If you look at WinMain(), you'll see the following piece of code.

```
// Initialize DirectPlay
hReturn = hrInitializeDirectPlay( hWnd );
if( FAILED(hReturn) ) {
    vCleanup();
    exit(1);
}
```

The hrInitializeDirectPlay() function is called here. If the function returns as FAILED(), then the code calls the vCleanup() function to clean up the program environment before exiting.

The hrInitializeDirectPlay() Function

The `hrInitializeDirectPlay()` function is similar to server-initialization functions of the past. Five main actions take place in this function, as far as communication is concerned.

- Initialization of COM
- Creation of the `IDirectPlay8Server` interface
- Initialization of the message handler
- Creation of the `IDirectPlay8Address` interface
- Setting the service provider to use TCP/IP

Initializing COM

Because DirectX is COM-based, you must initialize COM before creating objects for it. This is all handled using the all-too-familiar `CoInitialize()` function. The following code initializes COM for the game server.

```
hReturn = CoInitialize( NULL );
if( FAILED(hReturn) ) {
    MessageBox( hWindow, "Error Initializing COM", "DirectPlay Error",
     MB_ICONERROR );
    return hReturn;
}
```

The Critical Sections

One new item of note for this project is the inclusion of two critical sections. Sample programs of the past used only one critical section for player data. Well, this game server needs another one for the message-queuing mechanism.

```
// Initialize critical sections for multithreading
InitializeCriticalSection( &g_csModifyPlayer );
InitializeCriticalSection( &g_csMessageSection );
```

The `ModifyPlayer` critical section locks any changes to global player data. The `MessageSection` critical section locks any changes to the global message queue.

Creating the DirectPlay Interfaces

To run a server with DirectPlay, you must create at least two interfaces, one for the server interface and one for the address interface. You accomplish this step using the `CoCreateInstance()` function you have come to love. If you look at the code, you'll see that two interfaces are created in this manner.

```
if( FAILED( hReturn = CoCreateInstance( CLSID_DirectPlay8Server,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_IDirectPlay8Server,
    (LPVOID*) &g_pDPServer ) ) )
        MessageBox( hWindow, "Can't Create DPlayServer", "DirectPlay Error",
         MB_ICONERROR );
```

In this code, you can see the `IDirectPlay8Server` interface being created. This interface performs all server-required functions.

The second interface created, the `IDirectPlay8Address` interface, specifies the type of connection the server requires. In the game server's case, the address is the internal NIC card or modem that the server uses for its Internet connection.

Setting the Message Handler

One of the most important steps that you must not forget is the initialization of the function to handle all DirectPlay communication. This function will receive all messages from clients. The RPGQuestGameServer uses the `DirectPlayMessageHandler()` function to receive all client messages. To set the function to receive messages, you use the `Initialize()` member function of the `IDirectPlay8Server` interface.

Setting the Service Provider

So far, all the programs I have shown you use TCP/IP as the means of communication. You can feel free to try other methods, but I prefer TCP/IP, especially for game servers. You will be hard-pressed to write an efficient game server that sits on a 28.8K modem! The code to set the service provider is as follows.

```
// Set our service provider to TCP/IP
if( FAILED( hReturn = g_pDeviceAddress->SetSP( &CLSID_DP8SP_TCPIP ) ) ) {
    MessageBox( hWindow, "Failed to SetSP() for Device Address",
     "Invalid Param", MB_ICONERROR );
    return -1;
}
```

If you look at this code, you can see that the `SetSP()` member function of the `IDirectPlay8Address` interface sets up TCP/IP as the service provider.

NOTE

The GUIDs for the various service provider types appear in the dplay8.h header file that comes with the DirectX SDK.

The last few lines of code in the initialize function deal with deactivating players and sending

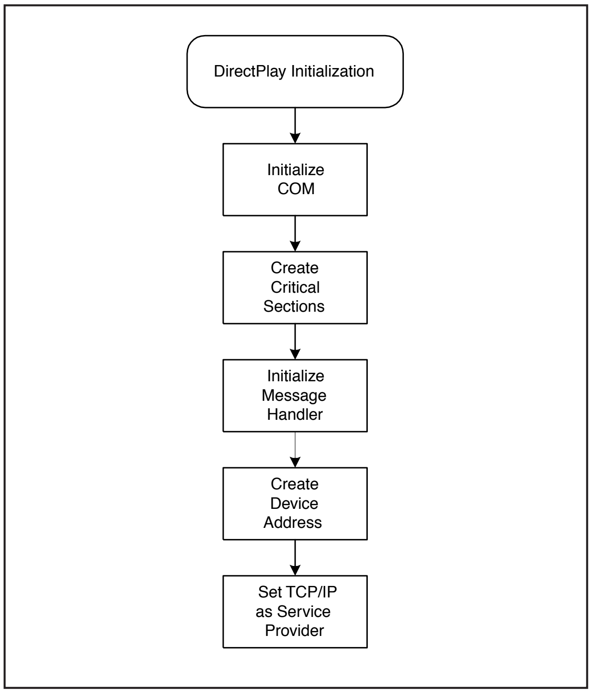

DirectPlay Initialization
Initialize COM
Create Critical Sections
Initialize Message Handler
Create Device Address
Set TCP/IP as Service Provider

Figure 16.7

The `hrInitializeDirectPlay()` *function*

a message to the server window. Since that is out of the way, you can move on to hosting. To see what was accomplished in this function, take a look a Figure 16.7.

Hosting the Game World

The next part of `WinMain()` deals with hosting the game world using the `hrHostGame()` function. Because the game server should always be active, there is no need to tie the host functionality to a keypress or command. That is why this particular program just hosts immediately.

The hrHostGame() Function

The `hrHostGame()` function is similar to the host functions of previous examples; the main difference here is that the port number is hard-coded. There is no real reason I have it hard-coded; I mainly did it to keep the code simple.

NOTE

Hard-coded values are fine for demonstration programs and prototypes, but you should always soft-code settings for production games and applications.

If you look closely at the `hrHostGame()` function, you'll see that it accomplishes a few key steps. It

- Sets up the server information
- Sets the application description
- Adds the port number component to the address
- Hosts the game

Setting the Server Information

You must set up the server's information so that players know the name of the game session, among other things. You do this by setting up a `DPN_PLAYER_INFO` data structure with the pertinent data. Once the data structure is set with the proper data, you call the `SetServerInfo()` function. The `SetServerInfo()` function belongs to the `IDirectPlay8Server` interface that you created earlier.

```
DXUtil_ConvertGenericStringToWide( wszServerName, "RPGQuest Server" );
ZeroMemory( &dpPlayerInfo, sizeof(DPN_PLAYER_INFO) );
dpPlayerInfo.dwSize = sizeof(DPN_PLAYER_INFO);
dpPlayerInfo.dwInfoFlags = DPNINFO_NAME;
dpPlayerInfo.pwszName = wszServerName;
// Set us up to be non-asynchronous
if( FAILED( hReturn = g_pDPServer->SetServerInfo( &dpPlayerInfo, NULL,
 NULL, DPNSETSERVERINFO_SYNC ) ) ) {
    MessageBox( hWindow, "Failed to SetServerInfo()", "Unknown Error",
     MB_ICONERROR );
    return -1;
}
```

NOTE

Don't forget that all string parameters in DirectPlay use wide character strings. You can always use the very useful `dxutil` functions, such as `DXUtil_ConvertGenericStringToWide()`, to help with conversions.

Setting the Application Description

You might recall that the application description for DirectPlay games describes the game in terms of communication. Different information such as the maximum number of players and the type of game is defined here. You define this by loading a DPN_APPLICATION_DESC data structure with the appropriate information.

```
ZeroMemory( &dnAppDesc, sizeof(DPN_APPLICATION_DESC) );
dnAppDesc.dwSize = sizeof(DPN_APPLICATION_DESC);
dnAppDesc.guidApplication = GUID_GAMESERVER;
dnAppDesc.pwszSessionName = wszSessionName;
dnAppDesc.dwMaxPlayers      = MAX_PLAYERS;
dnAppDesc.dwFlags = DPNSESSION_CLIENT_SERVER;
```

In this code, the program sets the game to be of client/server architecture. It also sets the GUID that identifies the application for other programs out there. The maximum number of players is set to a define value in the header file for the game project. In this scenario, the maximum number of players is set to 64.

Setting the Port Number

As with any good sockets-based program, the game server must specify the port used to accept player connections with the AddComponent() function.

```
hReturn = g_pDeviceAddress->AddComponent(DPNA_KEY_PORT,&dwPort,
sizeof(DWORD),DPNA_DATATYPE_DWORD);
if( hReturn != S_OK ) {
    MessageBox( hWindow, "Failed to AddComponent()", "hrHostGame()",
     MB_ICONERROR );
    return -1;
}
```

If you look at this code, you'll see that I pass a DPNA_KEY_PORT keyword to the function. This tells the function to add a port number setting to the host address.

NOTE

The key names for the various components appear in the dpaddr.h header file that comes with the DirectX SDK. Just search for DPNA_KEY_PORT in that file, and you will see the others.

The IDirectPlay8Server::Host() Function

The Host() function kicks off the actual game-hosting process. It requires the device address and application description that you have so diligently created. If all things go well, then the game is hosted and the server can await client connections. To see what was accomplished in the host function, look at Figure 16.8.

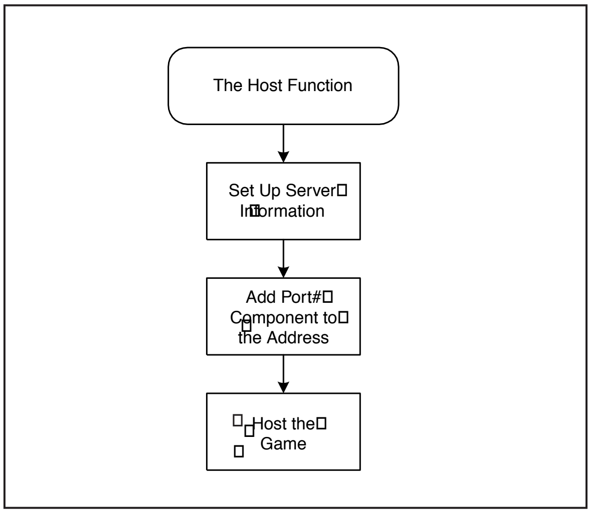


Figure 16.8

The `hrHostGame()` *function*

The Windows Message Loop

The Windows message loop, which should not be confused with the DirectPlay message loop, processes all Windows messages. Back in the `WinMain()` function, you can see the loop in action.

```
ZeroMemory( &msg, sizeof(msg) );
while( msg.message != WM_QUIT ) {
    if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) ) {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    else {
        // Read player messages
        vProcessMessages();
        if( timeGetTime() > dwGovernor+33 ) {
            // Update players and execute pending actions
            vUpdateTimeOfDay();
            vUpdatePlayers();
            dwGovernor = timeGetTime();
        }
    }
}
```

Basically, the message loop sits there and constantly checks for Windows messages. If one is found, it is sent to the `fnMessageProcessor()` function to be handled. The game server program does not do much with standard Windows messages, so it is pretty plain. It's the other functions present in the message loop that deserve attention.

Processing Player Messages

The `vProcessMessages()` function present in the `WinMain()` portion of the program processes all communications from clients. If you take a minute to study the code, you'll see that it loops through a maximum of 250 pending messages and acts upon them.

```
while( (iMessage = iGetMessage()) != -1 && iCount < 250 ) { … }
```

The code snippet here shows where the loop begins. To pull the next message out of the queue, the `vProcessMessages()` function calls the `iGetMessage()` function. As long as `iGetMessage()` does not return a `-1`, the message processor knows to continue. Take a look at the different types of messages.

- `PACKET_TYPE_CREATEPLAYER`
- `PACKET_TYPE_DESTROYPLAYER`
- `PACKET_TYPE_SUBMITPASSWORD`
- `PACKET_TYPE_ANGLE`
- `PACKET_TYPE_VELOCITY`
- `PACKET_TYPE_REQUESTPLAYERINFO`

The Create-Player Message

The create-player message works just as it did in previous program examples. When a player requests to join the game, he sends a create-player message. The code here shows how the call is processed.

```
if( gameMessage[iMessage].iType == PACKET_TYPE_CREATEPLAYER ) {
    vShowText(hLB_Output,"Creating Player");
    hrCreatePlayer(gameMessage[iMessage].byteMessageBuffer);
    continue;
}
```

If you look at the code, you'll see that the hrCreatePlayer() function is actually called to create the player.

The hrCreatePlayer() Function

The hrCreatePlayer() function activates a player slot in the game. Because all player data is stored in the PlayerInfo[] array, the new data is updated there. The create-player function accomplishes the following tasks.

- Retrieves information about the client
- Finds and activates an empty player slot
- Requests a password from the player
- Increments the number of active players

Retrieving Information about the Client

Before the game server can process the new player request, it must find out valuable information about the client, such as its name. It does this with the GetClientInfo() function that belongs to the IDirectPlay8Server interface. Once the call is made, the pointer to the DPN_PLAYER_INFO data structure is loaded with the pertinent information.

Activating the Player

Information about the players in the game is stored in the PlayerInfo[] data structure array. The bActive member variable in the array indicates whether a slot is open. If the slot is not active, then a player can be allocated there. The following code finds the free slot.

```
// Add player to list
for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
    if( !PlayerInfo[i].bActive ) {
        PlayerInfo[i].bCredentialsAccepted = 0;
        PlayerInfo[i].bActive = 1;
        PlayerInfo[i].dpnidPlayer = pCreatePlayerMsg->dpnidPlayer;
        strcpy(PlayerInfo[i].szPlayerName,strName);
        vRequestPlayerPassword(i);
        break;
    }
}
```

Requesting the Password

The program loops through the player information array and checks for an inactive player slot. Once one is found, it is activated and filled with the player's `DPNID` and name. After that is accomplished, the `vRequestPlayerPassword()` function is called to request the player's password.

The vRequestPlayerPassword() Function I organized most of the packet-sending logic into specialty functions, which makes it a little easier to follow what is happening. One of the most important of these functions is the `vRequestPlayerPassword()` function. This function tells the player that he needs to send his password for account verification. If his password does not match, then he is not allowed to play. This is accomplished with a call to the `hrSendServerMessage()` function.

```
void              *packet;
PacketGeneric     pacPass;
vSendUniqueMessage(iplayer,"Server Requesting
 Credentials",D3DCOLOR_RGBA(255, 255, 255, 255));
pacPass.dwSize = sizeof(PacketMessage);
pacPass.dwType = PACKET_TYPE_REQUESTPASSWORD;
packet = (VOID*)&pacPass;
hrSendServerMessage(iplayer,pacPass.dwType,packet,DPNSEND_GUARANTEED);
```

The main item of note here is the `DPNSEND_GUARANTEED` value passed to the flag's parameter. The password request is necessary to gameplay, so it is sent using guaranteed delivery. You will find that most of the other packets in the game use non-guaranteed delivery. Check out Figure 16.9 to see the flow of the program when processing the create-player message.

NOTE

Guaranteed packet delivery is extremely slow when compared to non-guaranteed delivery. Use guaranteed delivery only for mission-critical messages.

The Destroy-Player Message

Whenever a player leaves the game, he generates a destroy-player message. You handle the message by calling the `hrDestroyPlayer()` function. The `hrDestroyPlayer()` function is rather basic because all it has to do is deactivate the player who has left the game.

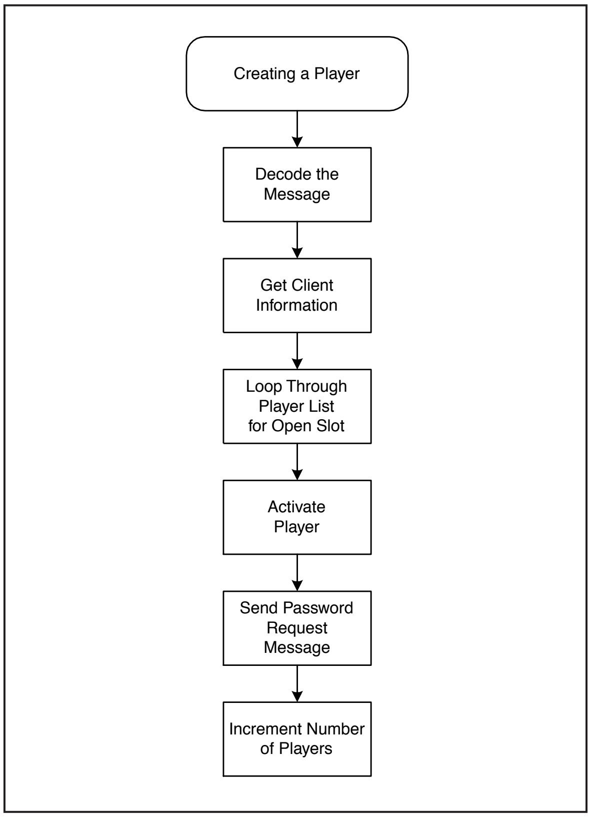

Creating a Player
Decode the Message
Get Client Information
Loop Through Player List for Open Slot
Activate Player
Send Password Request Message
Increment Number of Players

Figure 16.9

Creating a player

The hrDestroyPlayer() Function

In the `hrDestroyPlayer()` function, the program loops through the player data array and looks for a matching `DPNID`. Once the match is found, the player is deactivated.

```
HRESULT      hrDestroyPlayer( PVOID pMsgBuffer )
{
PDPNMSG_DESTROY_PLAYER                  pDestroyPlayerMsg;
HRESULT                                 hReturn = S_OK;
int                                     i;
char                                    szOutput[256];

// Get a destroy message pointer to the buffer
pDestroyPlayerMsg = (PDPNMSG_DESTROY_PLAYER)pMsgBuffer;
// Update the number of active players in a thread-safe way
InterlockedDecrement( &g_lNumberOfActivePlayers );
// Remove player from list
for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
    if( PlayerInfo[i].bActive ) {
        if( PlayerInfo[i].dpnidPlayer == pDestroyPlayerMsg->dpnidPlayer ) {
             PlayerInfo[i].bActive = 0;
             sprintf(szOutput,"<Slot%d><%s> Left The Game",i,
              PlayerInfo[i].szPlayerName);
             vShowText(hLB_Output,szOutput);
             // Save the player information for next login
             bSavePlayerInfo(i);
             vInformOfRemoval(i);
             break;
         }
    }
}
return(hReturn);
}
```

The bSavePlayerInfo() Function

There are a couple of new functions here—`bSavePlayerInfo()` and `vInformOfRemoval()`. The `bSavePlayerInfo()` function is called to save the current position and orientation of the exiting player. You do this so that the next time the player logs on, he will start where he left off.

The vInformOfRemoval() Function

The `vInformOfRemoval()` function, which is called after the save-player function, tells all of the other players that someone has left. You accomplish this by sending a `PACKET_TYPE_REMOVEPLAYER` packet to all other players in the game.

```
PacketRemovePlayer              pacRemove;
void                            *packet;
int                             i;
pacRemove.dwSize = sizeof(PacketRemovePlayer);
pacRemove.dwType = PACKET_TYPE_REMOVEPLAYER;
pacRemove.iPlayerID = iplayer;
packet = (VOID*)&pacRemove;
for(i=0;i<MAX_PLAYERS;i++) {
    if( PlayerInfo[i].bActive ) {
        if( i != iplayer ) {
            hrSendServerMessage(i,pacRemove.dwType,packet);
        }
    }
}
```

The program loops through the list of players and sends the removal message to all active players. If you look at the code, you'll see that it also makes sure not to send a removal message to the player who is leaving. I've jumped around a bit showing you the destroy-player function, so take a look at Figure 16.10 to see the flow.

The Submit-Password Message

The next message type checked is the password-submission message. This message type is sent from players who are trying to authenticate their connections. Once the packet is processed, the program calls the `bValidatePlayer()` function to validate the submitted password.

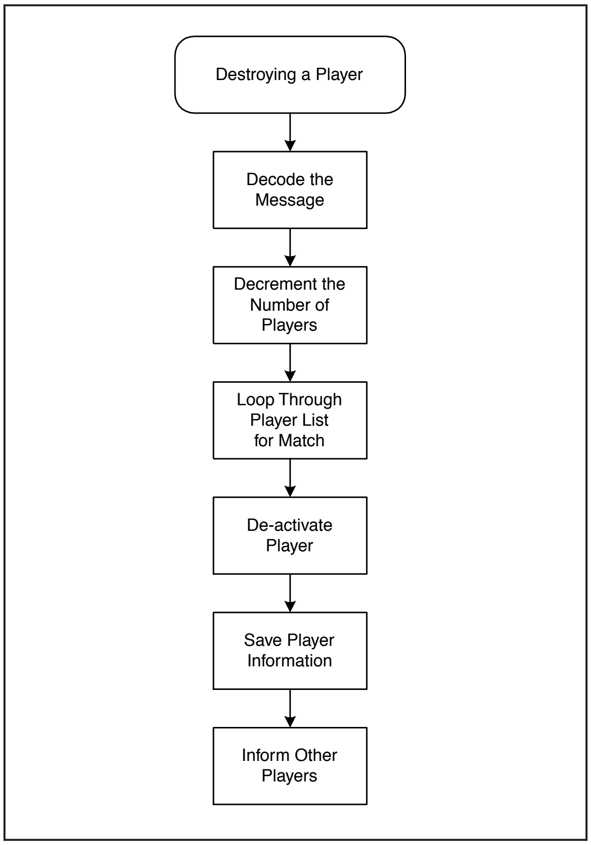


Figure 16.10

Flow of the program for destroying a player

The bValidatePlayer() Function

The `bValidatePlayer()` function validates player-submitted credentials. If the player's password is accepted, a `1` is returned. Upon failure, the function returns a `0`.

This function involves two main logic paths. The first logic check sees whether the player account can be loaded. All player accounts are based on data files stored in the PlayerData folder on the game server's hard drive. So to check for an existing player, the program must only try to load the data file. If the data file does not exist, then this is a new player trying to connect. In this case, a new account is created using the supplied password. The `vCreatePlayerAccount()` function handles the creation of the account.

If the player data can be loaded, the submitted password is checked against the loaded one. If the two passwords match, the player is accepted and a `1` is returned.

The vCreatePlayerAccount() Function

The `vCreatePlayerAccount()` function is rather straightforward because all it has to do is call the `bSavePlayerInfo()` function with the new player information.

```
void vCreatePlayerAccount(int iplayer, char *szpassword)
{
    PlayerInfo[iplayer].bCredentialsAccepted = 1;
    strcpy(PlayerInfo[iplayer].szPassword,szpassword);
    PlayerInfo[iplayer].iZone = 0;
    bSavePlayerInfo(iplayer);
    vSendUniqueMessage(iplayer,"New Player Account Created!",
     D3DCOLOR_RGBA(255, 255, 255, 255));
}
```

Before the function calls the save-player function, it sets the player's zone to the first zone in the game. The player's password is also copied over to ensure proper saving. After everything is taken care of, the server sends a message to the player informing him of his new account status.

The vZonePlayer() Function

Once a player is validated, he needs to know which zone he logged off from, so you call the `vZonePlayer()` function. In this function, the player is sent which zone he needs to be in and what his position should be. Because you don't just pop into new zones in your current position, this step is necessary.

To update the player with his new position, you make a call to the vUpdatePlayerPosition() function. This function does nothing more than send a position update packet to the player.

Once the player has his new position, his arrival is broadcast to all the players in the new zone. Not only is the zoning player's information sent to all those in the new zone, but the zoning player also receives information about everyone in the zone. This happens so all the players involved are kept up to date about who is in their zone. Figure 16.11 might make this functionality a little easier to digest.

The Angle Message

When a player wants to rotate his avatar, he sends an angle change message to the server. This message takes form as the PACKET_TYPE_ANGLE message type. There is not much work required to process the actual angle message; all the server has to do is acknowledge the message and send the new heading to all other players involved. To send the new heading to everyone, the angle handler calls the vSendPlayerAttitude() function.

The vSendPlayerAttitude() Function

The vSendPlayerAttitude() function sends a player's rotation, velocity, and position to other players in the game. It can also send the player's information to himself. This functionality is built in so that the server can tell a player to adjust his movement. The following code sends the player's own statistics.

```
if( bEcho ) {
    // Send the message to the player
    pacAttitude.iPlayerID = -1;
    hrSendServerMessage(iplayer,pacAttitude.dwType,packet,
     DPNSEND_GUARANTEED);
}
```

The bEcho flag tells the function to echo the packet to the player who the attitude involves. So when you want to echo the attitude, just set this flag to TRUE.

The Velocity Message

The velocity message works just like the angle message except that the player's velocity is changing, not his angle. Whenever a player accelerates or decelerates, he sends the velocity-change message. Once the server receives the message, it processes it and then broadcasts it to all other players involved. This all works in the same way the angle packet does.

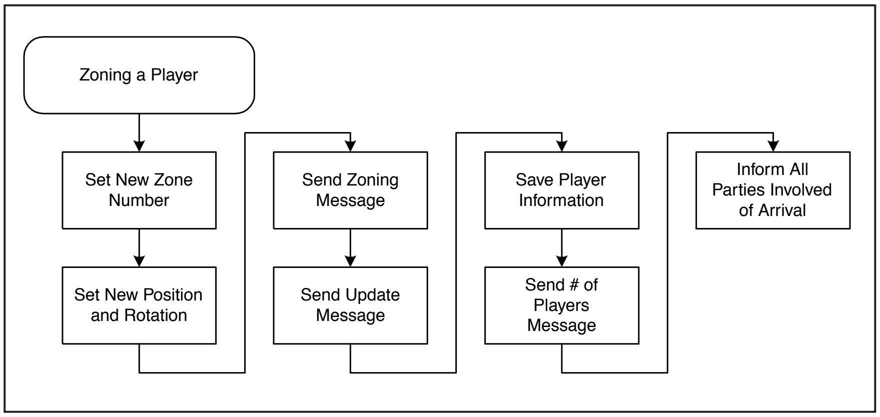


Figure 16.11

Process flow for zoning a player

The Request-Player-Information Message

The `PACKET_TYPE_REQUESTPLAYERINFO` is sent by a client when it wants information about a player that it does not recognize. There are a few scenarios in which this might happen; the main one is a player who misses the zoning packet. Anyway, if a player receives a packet about another player that he does not have active, he sends the request-player-information packet to the server. The actual sending of player information happens with the `vSendPlayerInformation()` function.

The vSendPlayerInformation() Function

The `vSendPlayerInformation()` function sends all pertinent information regarding a player to another player or to the player himself.

```
void vSendPlayerInformation(int iplayer, int ireceive)
{
    PacketPlayerInfo          pacPInfo;
    void                      *packet;
    pacPInfo.dwSize = sizeof(pacPInfo);
    pacPInfo.dwType = PACKET_TYPE_ADDPLAYER;
    pacPInfo.iPlayerID = iplayer;
    pacPInfo.piInfo.fRot = PlayerInfo[iplayer].fRot;
    pacPInfo.piInfo.vecCurPos.x = PlayerInfo[iplayer].vecCurPos.x;
    pacPInfo.piInfo.vecCurPos.y = PlayerInfo[iplayer].vecCurPos.y;
    pacPInfo.piInfo.vecCurPos.z = PlayerInfo[iplayer].vecCurPos.z;
    pacPInfo.piInfo.fVelocity = PlayerInfo[iplayer].fVelocity;
    pacPInfo.piInfo.iFrame = PlayerInfo[iplayer].iFrame;
    pacPInfo.piInfo.iZone = PlayerInfo[iplayer].iZone;
    pacPInfo.piInfo.iZoneMap = PlayerInfo[iplayer].iZoneMap;
    strcpy(pacPInfo.piInfo.szPlayerName,PlayerInfo[iplayer].szPlayerName);
    packet = (VOID*)&pacPInfo;
    hrSendServerMessage(ireceive,pacPInfo.dwType,packet,DPNSEND_GUARANTEED);
}
```

Take a look at the code, and you'll see that the rotation, position, velocity, animation frame, zone, zone map, and player name are all sent in the transmission. Items such as the password are not sent because you do not want to give out the player's password to other people.

Whenever a player enters a zone or enters the game, his player information is sent to everyone in the zone. This happens so that all players present know what to do with the new player. You can also send the information when other players request it. Check out Figure 16.12 to see when this function is called.

Calculating the Time of Day

Most persistent-world online games have a persistent time of day. If you change the setting of the sun, the game world seems more alive than if time stands still. I'll show you how to do this in *RPGQuest* using the `vUpdateTimeOfDay()` function.

The vUpdateTimeOfDay() Function

The `vUpdateTimeOfDay()` function works by adjusting a value that ranges from 0 to 255. There is nothing very special about the time of day; all it affects is the ambient levels for rendering the 3D scenes on the client.

To keep the light setting from seesawing too much, I put in timers to make it sit for a longer period of time at high noon and at midnight. This provides an effect of night and day. To see how the timeframe works out, take a look at Figure 16.13.

Once the proper time of day is updated, a `PACKET_TYPE_TIMECHANGE` packet is sent containing the new setting.

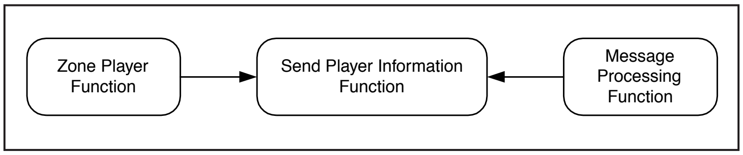


Figure 16.12
Methods that call the `vSendPlayerInformation()` *function*

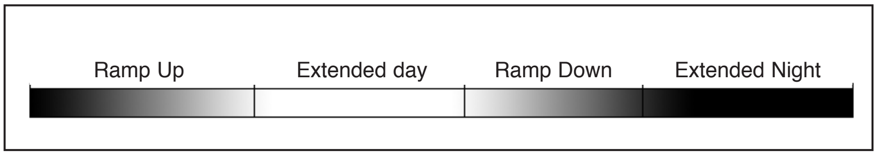


Figure 16.13

Time-of-day scale

Updating the Players

Once the time of day is sent, the players in the game need their positions updated based on their velocity and angle. You accomplish all of this using the `vUpdatePlayers()` function.

The vUpdatePlayers() Function

The `vUpdatePlayers()` function updates players' positions and velocity settings. It also checks for collision detection and sends updated player attitudes when necessary.

The function starts out by looping through each zone. Within each zone, it loops through the list of players, checking for active players in that zone. If an active player is found, the program updates his position according to his angle and velocity.

```
for( iZone = 0 ; iZone < 3 ; iZone++ ) {
    for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
       // Update the player if active
       if( PlayerInfo[i].bActive ) { …
```

In this code snippet, you can see that the zone loop is first, followed by the player loop. Only active players are updated. The `vGetVelocity()` function, which is identical to the one in *SpacePirates*, determines their position changes.

After a player's position and frame are updated, his new position is checked with the `iValidatePlayerPosition()` function to see whether he has collided with something in the scene.

The iValidatePlayerPosition() Function

In the `iValidatePlayerPosition()` function, I included checks for all bounding boxes in the three zones. If the player collides with a collision zone, the function returns a `0`. A more efficient method is for the program to read in a list of collision zones, but I have done it the former way to keep things simple.

```
if( x > -300.0f && x < -258.0f ) {
    if( z > -18.0f && z < 17.0f ) {
        return( 3 );
    }
}
```

You might have noticed that sometimes the function returns a value greater than `1`. A return code greater than `1` means that the player has entered a zoning box. This means that the player should be sent to another zone. By subtracting `1` from the return value, you can find the zone to which he needs to go. So in the preceding example, he should be transported to zone 2 (3-1). Players who need to zone are transported by the `vZonePlayer()` function that I mentioned earlier.

Review

This entire chapter was dedicated to showing you how to create the game server application, so there is not a whole lot to review here. Nevertheless, please keep in mind the following points.

- The player data is stored on the server's hard drive in the PlayerData folder. You can delete any player you want, simply by removing his data file from the folder.
- The *RPGQuest* server utilizes a secondary message queue method to process player messages. This is the desired method since it keeps the DirectPlay message handler from having to wait for messages to process.
- Critical sections are required to keep the secondary message queue from stepping over global memory. Be very careful with critical sections, because you can write code in such a way that it locks up. This happens when code tries to enter a critical section while the program is already in the critical section.

That wraps it up for the server code. You now have a solid foundation of knowledge, so the rest is downhill from here.

CHAPTER 17

Writing the RPGQuest Game Client

In the last chapter, you learned how to write the *RPGQuest* game server. In this chapter, you will learn how to write its counterpart, the game client. One of the nice things about client/server-based multiplayer game programming is that once you have the server written, you can concentrate on mostly non-network issues. The client mainly consists of graphic, sound, and input calls. The actual network code ends up rather small in comparison. In this chapter, I'll cover the following topics.

- Project layout
- Graphics handling
- Sound and input handling
- Network handling

The Project Space

Go ahead and load up the project located on the CD in the Chapter17 Zip file. It contains all of the files necessary to compile the game client. One thing you probably want to do is launch the game-server application so that you can test the client. The project space for the client is actually a bit more complicated than the server, but it's not too bad. The libraries used in the project can be found in Table 17.1.

Table 17.1 RPGQuest Libraries

Library	Description
dinput8.lib	DirectInput library
dxguid.lib	GUIDs for DirectX
dsound.lib	DirectSound library
winmm.lib	Windows multimedia library
d3d8.lib	Direct3D library
d3dx8.lib	Direct3D helper library

There shouldn't be too many surprises in the list of libraries; most, if not all of them, are present in *SpacePirates*. Now that you have seen the libraries, take a look at Table 17.2 to see the source files needed to compile the project. Then, take a look at Figure 17.1 to see the project's file view.

The Object3DClass Files

The `Object3DClass` files provide a simple class that loads and manages x-file objects. It is the same class that you used back in the *SpacePirates* game, so you should be rather familiar with it. As a brief refresher, the following functions are provided by the class.

- `hLoad()`
- `vDisplayXYZ()`

The `hLoad()` function loads x-file object data and textures. Its real power comes from the fact that it allocates all the memory needed for the geometry, material, and texture data associated with the object.

The `vDisplayXYZ()` function puts the 3D object anywhere in your scene at the given coordinates. You can also set the orientation of the object by passing in rotation parameters for all three axes.

Table 17.2 RPGQuest Source Files

File	Description
dxutil.h	Helper functions header file
dxutil.cpp	Helper functions source file
Object3DClass.h	X-file class header file
Object3DClass.cpp	X-file class source file
SoundSystem.h	Sound functions header file
SoundSystem.cpp	Sound functions source file
RPGQuest.h	Main game code header file
RPGQuest.cpp	Main game code source file

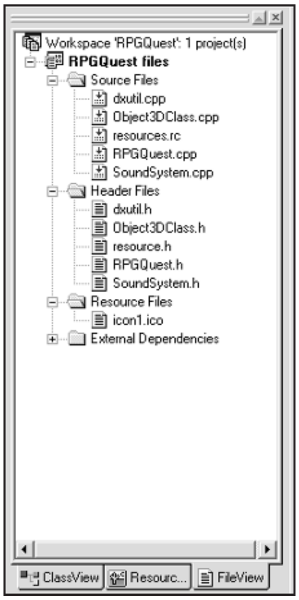


Figure 17.1

The project's file view

The SoundSystem Class Files

The `SoundSystem` class files provide a simple foundation for loading and playing sound files. The files have not changed since their first appearance in *SpacePirates*, so you might recognize most of the code. The following functions are provided by the `SoundSystem` class.

- `hrInitSoundSystem()`
- `hrLoadSound()`
- `hrPlaySound()`

You call the `hrInitSoundSystem()` function only once in your application because all it does is initialize the sound system for use. It handles all of the nitty-gritty details involved in creating the interfaces and such.

The `hrLoadSound()` function does just what its name implies: It loads sound data from wave files. You can then use the loaded sound data for playback.

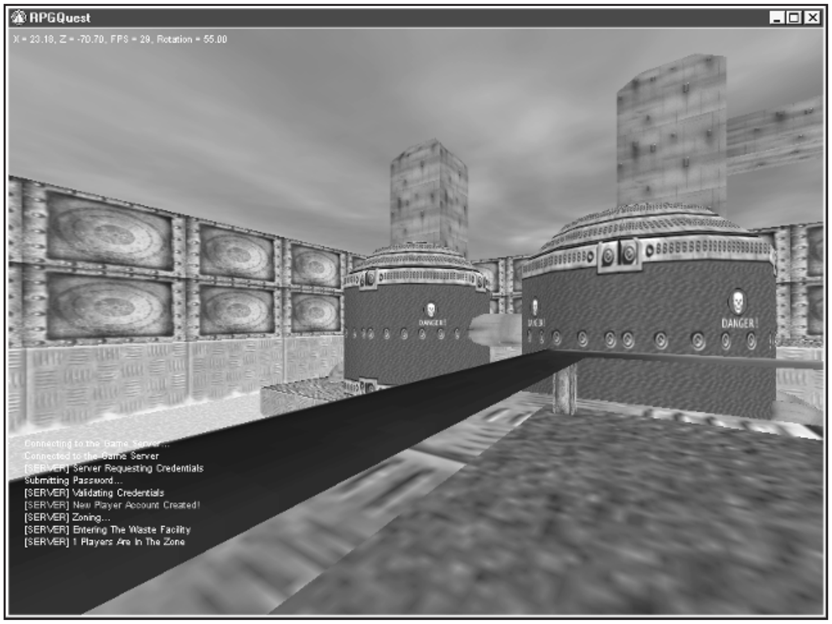
The `hrPlaySound()` function plays back a previously-loaded game sound. It doesn't get any simpler than this!

The Graphics System

The graphics system for *RPGQuest* is very similar to that of *SpacePirates* except that *RPGQuest* uses a first-person view of the world. By a first-person view, I mean that the camera is positioned at the eye level of the player. This gives a much greater sense of immersion in the game world and provides for a different experience than in *SpacePirates*. Besides the camera perspective, there is not a whole lot different for the *RPGQuest* game client. The graphics system can be organized into the following parts.

- Direct3D initialization
- 3D data loading
- Light initialization
- Camera positioning
- Time-of-day adjustment
- Scene rendering

Before you go any further into the graphics system, you might want to see the final output for the client application. The output for the client appears in Figure 17.2.



```
X = 23.18, Z = -70.70, FPS = 29, Rotation = 55.00
```

```
Connecting to the Game Server...
Connected to the Game Server
[SERVER] Server Requesting Credentials
Submitting Password...
[SERVER] Validating Credentials
[SERVER] New Player Account Created!
[SERVER] Zoning...
[SERVER] Entering The Waste Facility
[SERVER] 1 Players Are In The Zone
```

Figure 17.2

The client's view

Direct3D Initialization

As with the other 3D programs demonstrated in the book, all of the 3D system initialization happens with the `hrInit3D()` function. This function's one and only purpose is to set up the client's display in a manner that is compatible with the application's requirements.

NOTE
Some of the functions in *RPGQuest* are duplicates of those found in *SpacePirates*. For these functions, I'll only gloss over how they work. For complete explanations, refer to the chapters in which the functions originally appeared.

The hrInit3D() Function

The `hrInit3D()` function, which you should have seen before, contains all of the function calls necessary to set up the client's graphics system to play *RPGQuest*. If everything works as planned, the player's computer will be ready to display graphics. Initializing the graphics system involves the following steps.

- Creation of the `IDirect3D8` interface
- Enumeration of available adapters, devices, and modes
- Creation of the `IDirect3DDevice8` interface

Creating the IDirect3D8 Interface

The `IDirect3D8` interface is the heart of the 3D system. Without it, you cannot set up the graphics display adapter, device, or mode. In other words, you won't have anywhere to display your graphics. Although DirectX is COM-based, there is a special function for creating the 3D interface. This special function is called `Direct3DCreate8()`.

```
if( NULL == ( pD3D = Direct3DCreate8( D3D_SDK_VERSION ) ) ) {
    MessageBox( hWnd, "Direct3D Error", "Unable to initialize Direct3D.",
     MB_ICONERROR );
    return(E_FAIL);
}
```

The only parameter you have to worry about when creating the Direct3D interface is the version macro. For this, you pass `D3D_SDK_VERSION`. All this does is

NOTE
The DirectX interface objects are easy to pick out because they all start out with an I—as in the `IDirect3D8`, `IDirectPlay8Client`, and `IDirectInput8` interfaces.

ensure that you have the proper header and library files included in your project. If you have a mismatch of versions, the compiler lets you know.

Building the Device List

Once you create the 3D interface, you need to figure out which video adapter to use. You also need to know what type of devices and modes are supported by the adapter. You find all this using the `hrBuildDeviceList()` function.

The `hrBuildDeviceList()` function is a little too complicated for me to cover again, so I'll just scratch the surface. Basically, the function detects all the video adapters connected to the user's system and finds display modes that meet the game requirements.

NOTE

It is important to plan on users having more than one video adapter. Many motherboards come with one built on-board, but quite a few people with these types of motherboards go on to purchase better display adapters. If you just assume that the first adapter you enumerate is fine to use, then your program probably uses the slower graphics system. This is why it is important to check for all adapters and let the user decide which one he wants to use.

Creating the Display Device

Now that you have a list of available devices, you have to set up the system to use the default one returned. You don't actually have to use the default adapter returned; you could present the list of identified modes and let the user pick the one he wants to use.

The first part of code that sets up the display device specifies the presentation parameters for the device. These parameters set such things as the screen resolution and color depth.

```
// Set up the presentation parameters
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed   = pDeviceInfo->bWindowed;
```

```
d3dpp.BackBufferCount   = 1;
d3dpp.MultiSampleType   = pDeviceInfo->MultiSampleType;
d3dpp.SwapEffect   = D3DSWAPEFFECT_FLIP;
d3dpp.EnableAutoDepthStencil   = g_bUseDepthBuffer;
d3dpp.AutoDepthStencilFormat   = pModeInfo->DepthStencilFormat;
d3dpp.hDeviceWindow   = hWnd;
```

The next block of code sets up some device parameters based upon whether the game is to be displayed in a window. Because full-screen games require different parameters, you must use this logic.

```
// Run this code if in windowed mode
If( g_bWindowed ) {
    d3dpp.BackBufferWidth  = (rcWindowClient.right - rcWindowClient.left);
    d3dpp.BackBufferHeight = (rcWindowClient.bottom - rcWindowClient.top);
    d3dpp.BackBufferFormat = pAdapterInfo->d3ddmDesktop.Format;
}
// Run this code if in full-screen mode
else {
    d3dpp.BackBufferWidth  = pModeInfo->Width;
    d3dpp.BackBufferHeight = pModeInfo->Height;
    d3dpp.BackBufferFormat = pModeInfo->Format;
}
```

The last block of code in the initialization function deals with creating the actual device interface. If all goes well, then you end up with a usable 3D device interface.

```
hResult = pD3D->CreateDevice( g_dwAdapter, pDeviceInfo->DeviceType,
 hWnd, pModeInfo->dwBehavior, &d3dpp, &pd3dDevice );
if( !SUCCEEDED(hResult) ) {
    MessageBox( hWnd, "Device Error", "Failed to create display device!",
     MB_ICONERROR );
    return(E_FAIL);
}
```

The initialization function has a few more lines of code; all they do is set up the z-buffer and ambient light levels for the 3D scene. To see all of the steps used to set up the display environment, look at Figure 17.3.

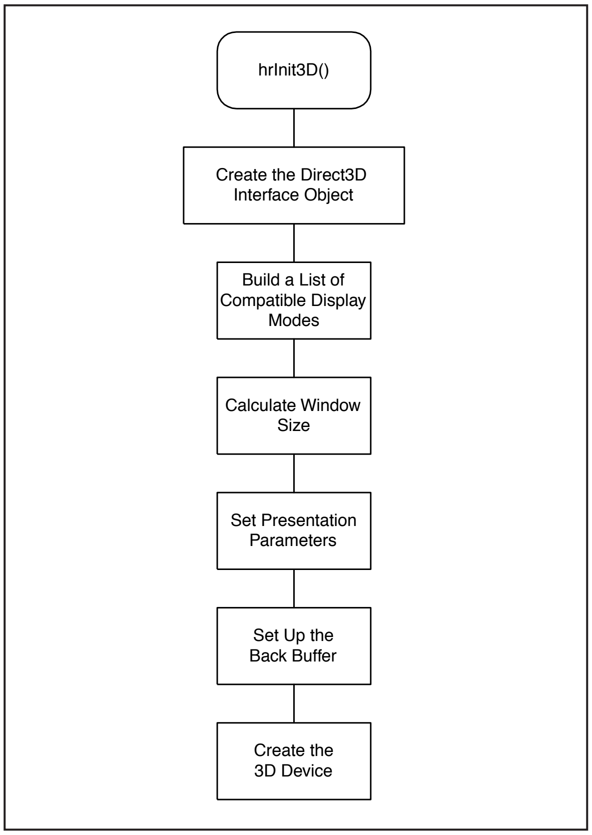

hrInit3D()
Create the Direct3D Interface Object
Build a List of Compatible Display Modes
Calculate Window Size
Set Presentation Parameters
Set Up the Back Buffer
Create the 3D Device

Figure 17.3

Setting up the display environment

Loading the 3D Data

As with all 3D games, you must load data. Because all objects in a 3D environment contain geometric information, this is a necessary step. In *RPGQuest*, all of the work happens in the `vInitGeometry()` function. The code for the function follows.

```
void vInitGeometry(void)
{
    char      szFileName[256];
    float     fSizeX = 50.0f;
    float     fSizeY = 50.0f;
   DWORD      dwColor = D3DCOLOR_XRGB(255,255,255);

    sprintf(szFileName,"Data\\3DObjects\\rpgquest0.x");
    o3dZone[0].hLoad(szFileName,pd3dDevice);
```

```
     sprintf(szFileName,"Data\\3DObjects\\rpgquest1.x");
     o3dZone[1].hLoad(szFileName,pd3dDevice);
     sprintf(szFileName,"Data\\3DObjects\\rpgquest2.x");
     o3dZone[2].hLoad(szFileName,pd3dDevice);
     // Load the player (cubehead)
     sprintf(szFileName,"Data\\3DObjects\\cube_head.x");
     o3dPlayer.hLoad(szFileName,pd3dDevice);
     // Create the common text font
     hFont = CreateFont(12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, PROOF_QUALITY,
      0, "arial");
    D3DXCreateFont(pd3dDevice, hFont, &pD3DXFont);
}
```

The first thing that jumps out at you is that this function is so short. In *SpacePirates*, a lot of the geometry was created within the program, thus the larger initialization function. In *RPGQuest*, all of the geometry is created in 3D Studio MAX; therefore, it all gets loaded in. Take a peek at Figure 17.4 to see the merchant zone when viewed from afar.

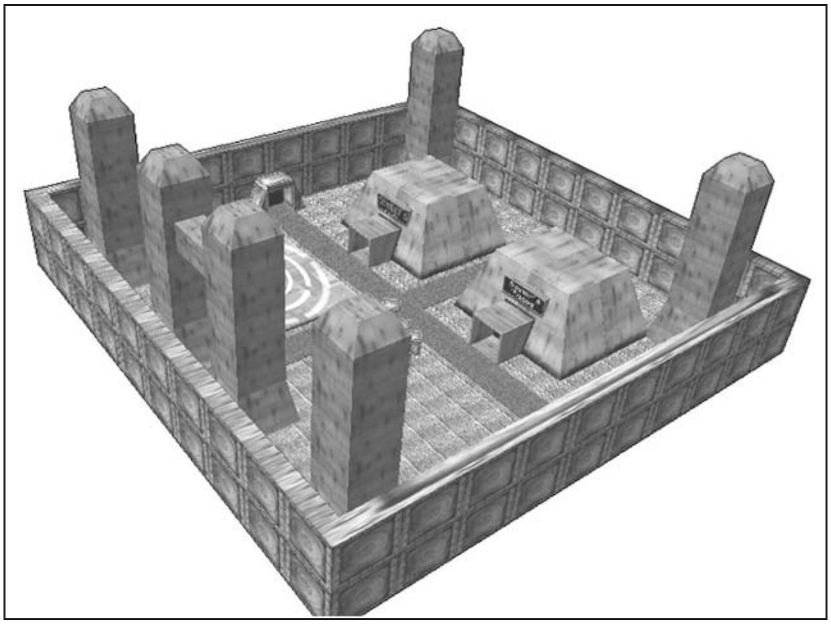


Figure 17.4

The merchant zone

Loading the Zone Data

Each zone in *RPGQuest* has different geometry. To keep things simple, I made the entire zone all one big piece of data. Because all of the data is contained in a single x-file mesh object, you have to call the load function only once per zone.

```
sprintf(szFileName,"Data\\3DObjects\\rpgquest0.x");
o3dZone[0].hLoad(szFileName,pd3dDevice);
sprintf(szFileName,"Data\\3DObjects\\rpgquest1.x");
o3dZone[1].hLoad(szFileName,pd3dDevice);
sprintf(szFileName,"Data\\3DObjects\\rpgquest2.x");
o3dZone[2].hLoad(szFileName,pd3dDevice);
```

Each zone is named `rpgquest` plus a number. So zone 1 is `rpgquest0.x`, zone 2 is `rpgquest1.x`, and zone 3 is `rpgquest2.x`. You load each of these using the `hLoad()` function I discussed earlier.

The player's avatar data is loaded next in the form of the `cube_head.x` file object. After that is done, the font for the text display is loaded. Well, as I said, the geometry initialization routine is rather simple, and that is all there is to it!

Lighting the Scene

You set up the lights in *RPGQuest* in the same manner as you did in *SpacePirates*. This means that you put a single white light in the middle of each zone and add four colored lights, one in each corner.

```
d3dLight.Diffuse.r  = 1.0f;
d3dLight.Diffuse.b  = 1.0f;
d3dLight.Diffuse.g  = 1.0f;
d3dLight.Position.x = 0.0f;
d3dLight.Position.y = 50.0f;
d3dLight.Position.z = 50.0f;
d3dLight.Attenuation0 = 1.0f;
d3dLight.Attenuation1 = 0.0025f;
d3dLight.Range      = 1000.0f;
// Set the active light
pd3dDevice->SetLight( 0, &d3dLight );
// Enable the light
pd3dDevice->LightEnable( 0, TRUE );
```

You create each light by setting its diffuse color, attenuation, and range. Once the settings for the light are in place, you create the light using the `SetLight()` function. Next, you use the `LightEnable()` function to activate the light. Because all of the lighting is the same for each zone, you create the lights only once.

Setting the Camera Position

The camera in *RPGQuest* follows the character as he walks around the zone. To keep up with the player, the camera's rotation is kept in step with the player's rotation. All of this is handled by the `vUpdateCamera()` function.

The vUpdateCamera() Function

At the beginning of each rendering frame, you call the `vUpdateCamera()` function to set the camera in the proper position. In *RPGQuest*, the proper position is eye level with the player's coordinates.

```
void vUpdateCamera(void)
{
    D3DXMATRIX                matRotation;
    // Set where camera is viewing from
    vecCameraSource.x = LocalPlayer.vecCurPos.x;
    vecCameraSource.y = 15.0f+LocalPlayer.vecCurPos.y;
    vecCameraSource.z = LocalPlayer.vecCurPos.z+1;
    // Set what camera is looking at
    vecCameraTarget.x = LocalPlayer.vecCurPos.x;
    vecCameraTarget.y = 15.0f+LocalPlayer.vecCurPos.y;
    vecCameraTarget.z = LocalPlayer.vecCurPos.z;
    // Set up camera view
    D3DXMatrixLookAtLH( &matCameraView,
        &vecCameraSource,
        &vecCameraTarget,
        &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
    // Rotate the camera view now
    D3DXMatrixRotationY( &matRotation, -LocalPlayer.fRot/RADIAN_TO_DEGREES );
    matCameraView *= matRotation;
    // Tell the device to use the camera view for the viewport
    pd3dDevice->SetTransform( D3DTS_VIEW, &matCameraView );
}
```

For the most part, the update camera function is identical to the one in *SpacePirates*. The main difference is that this version contains a rotation adjustment for the camera. Just like any other object in 3D space, the camera is set into the scene with matrices. Once the camera is positioned, its rotation is adjusted to match the player's orientation. You accomplish this by multiplying the camera's view matrix by the player's rotation matrix. The end result is a properly-rotated camera.

Setting the Client's Time of Day

In Chapter 16, "Writing the RPGQuest Game Server," you learned about how the server sends time-of-day updates to all connected clients. This gives a sense of time passing in the game. It is really a simple effect; all the game engine has to do is set the ambient light level according to the time value sent from the server.

In the DirectPlay message-processing function, a time packet that comes through calls the `vUpdateTimeOfDay()` function. Here is the snippet of code that calls the function.

```
else if( PGen->dwType == PACKET_TYPE_TIMECHANGE ) {
    pacTime = (PacketTimeChange*)pReceiveMsg->pReceiveData;
    vUpdateTimeOfDay(pacTime->iTime);
    break;
}
```

The vUpdateTimeOfDay() Function

The update time-of-day function is rather simple because all it must do is set the global variable, `colorTimeColor`, to the proper setting. The proper setting is the one passed by the server. The value from the server, 0 to 255, denotes the brightness level for the graphic display. By converting this number to a color value, the render function can plug it straight in for use.

```
void vUpdateTimeOfDay(int newtime)
{
    if( newtime < 0 )
        newtime = 0;
    else if ( newtime > 255 )
        newtime = 255;
    bUpdateTime = 1;
    colorTimeColor = D3DCOLOR_XRGB(newtime,newtime,newtime);
}
```

The first part of the function makes sure the value received is valid. If you try to create a color value greater than 255 or less than 0, you could bring up the blue screen of death. This is why the function checks `newtime` to make sure it is between 0 and 255.

NOTE

It is always a good idea to check data received from the server. Sometimes packets can be mangled. In that case, you've probably got bad data within the packet. As they always say, better safe than sorry.

The `bUpdateTime` flag is set to `TRUE` so that the next rendering phase knows to adjust the brightness of the scene. You can just update the brightness every frame, but that would slow down the rendering unnecessarily.

The last bit of code sets the actual color value based on the `newtime` variable. You do this using the `D3DCOLOR_XRGB()` macro provided in the `dxutil` function set. Figure 17.5 outlines the series of events that transpire when an update time packet is received.

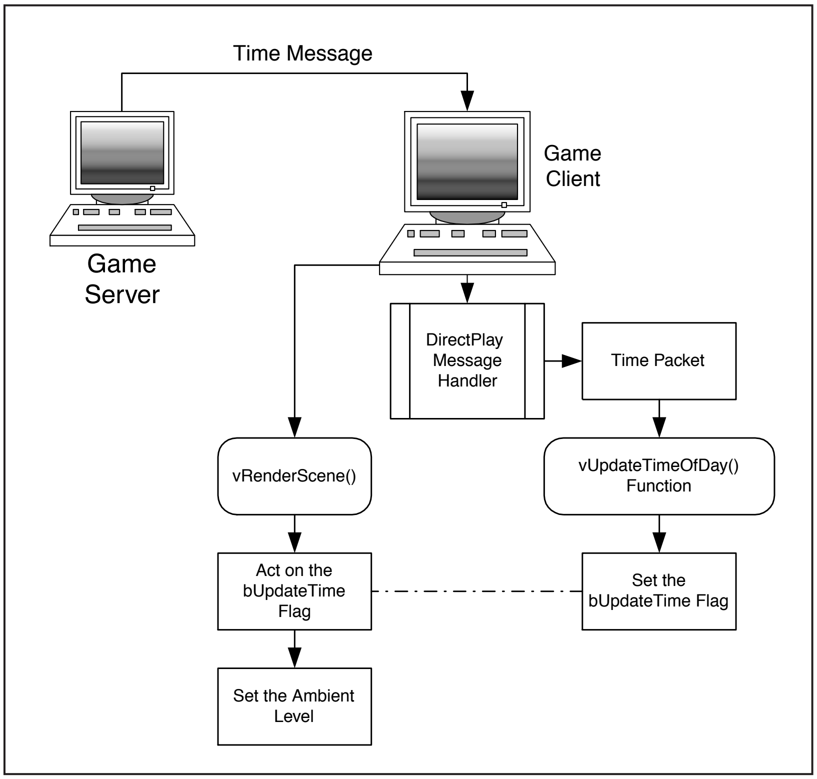


Figure 17.5

Events that take place when a time-of-day packet is received

Rendering the Scene

To create the graphical display for the player, you must render the 3D scene. It is called a scene because you have actors, lights, and at least one camera. So you see, the vernacular comes from the movie industry. In this case, the 3D models are the actors. Anyway, on with the show! All of the rendering for *RPGQuest* happens in the good ol' `vRenderScene()` function.

The vRenderScene() Function

The render scene function's only purpose is to create the graphical display for the player. To do this, it completes the following tasks.

- Adjust light levels if necessary
- Clear the display buffers
- Update the camera's orientation
- Set texture filtering levels
- Render the players
- Render the arena
- Calculate frames per second
- Display various information
- Output the message log
- Present the created scene

Adjusting the Light Level

As I discussed earlier, if the `bUpdateTime` flag is signaled, the rendering function updates the ambient light levels. This is as easy as calling the `SetRenderState()` function with a `D3DRS_AMBIENT` value.

```
if( bUpdateTime ) {
    pd3dDevice->SetRenderState( D3DRS_AMBIENT, colorTimeColor );
    bUpdateTime = 0;
}
```

The last step is setting the `bUpdateTime` variable to *non-signaled.*

When something is *signaled*, it means it has been activated. In this case, `bUpdateTime` equals `1` when it is signaled and `0` when it is *non-signaled*.

Clearing the Display Buffers

Every time you render your 3D scene, you must clear the display buffers. This is necessary to make sure that you don't have any leftover garbage in your display. Because you are also using a depth buffer in *RPGQuest*, you must clear it to make sure objects are displayed properly. You do all of this with the `Clear()` function.

```
// Clear the backbuffer and the z-buffer
pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
 D3DCOLOR_XRGB(0,0,0), 1.0f, 0 );
```

In this code, the program clears the display buffer along with the z-buffer. Both buffers are set to (0,0,0), which is true black.

NOTE
Always be sure to clear your depth buffer if you are using one. If you forget to do this, you'll end up with strange rendering anomalies.

Updating the Camera's Orientation

Updating the camera is easy because all you have to do is make a call to the `vUpdateCamera()` function that I mentioned earlier.

Setting the Anistropic Level

Anistropic texture filtering fixes texture distortion caused by textures at certain angles to the camera. This subject is kind of coming at you from left field, but I thought you might be interested in knowing how to turn it on. To activate the anistropic texture filter, all you have to do is set the texture stage state to use it.

```
// Turn on texture filtering
pd3dDevice->SetTextureStageState(0,D3DTSS_MINFILTER ,D3DTEXF_ANISOTROPIC);
pd3dDevice->SetTextureStageState(0,D3DTSS_MAGFILTER ,D3DTEXF_ANISOTROPIC);
```

This isn't really the time or place to get into how it works, but to see the difference, comment out the preceding code in the render loop. The best way to notice the difference is to look at textures that are at 90 degrees to the camera. Go ahead and try compiling the code with it turned off. You should notice the change almost immediately.

NOTE
Not all video cards support anistropic filtering. If you are using a card that does not support it, you will not notice a difference with filtering turned on or off.

Rendering the Players

Ah, now comes the fun part, rendering the cube heads! Okay, maybe it's not *that* much fun, but at least it's easy. Because the game can have multiple players, you must loop through the list of players and render ones that are active. You accomplish this using the following loop.

```
// Display players
for( i = 0 ; i < MAX_PLAYERS ; i++ ) {
    if( PlayerInfo[i].bActive && PlayerInfo[i].iZone == iZone )
        o3dPlayer.vDisplayXYZ(PlayerInfo[i].vecCurPos.x,12.0f,
        PlayerInfo[i].vecCurPos.z,0.0f,PlayerInfo[i].fRot,0.0f);
}
```

The program loops through the total list of possible players and renders out the ones who are active and in the same zone. The actual rendering happens in the `vDisplayXYZ()` member function of the `Object3DClass`.

Rendering the Arena

The rendering of the player's surroundings is the easiest part because the zones consist of single objects. So to render the zone, all you have to do is call the `vDisplayXYZ()` function.

```
// Display the arena
o3dZone[iZoneMap].vDisplayXYZ(0.0f,0.0f,0.0f);
```

You might have noticed that the rotation parameters are missing. The zones never rotate; they are always at a fixed orientation.

Calculating Frames per Second

RPGQuest never operates at more than 30 frames per second because of the time-governing system built into the `WinMain()` function. But the engine might operate slower than 30 frames per second on machines that lack the horsepower to display the scene. To give the player an idea of the game performance, *RPGQuest* includes a frames-per-second indication. By frames-per-second, I mean how many times the entire 3D scene is rendered per second of gameplay. You will find that most games rate themselves this way. The following code illustrates how to perform the calculations.

```
if( timeGetTime() > dwFrameTimer+1000 ) {
    iFPS = iFrames;
    dwFrameTimer = timeGetTime();
    iFrames = 1;
}
```

```
else {
    iFrames++;
}
```

The `dwFrameTimer` variable is a global variable that keeps track of the amount of time that has passed since the last time update. You use this to see whether a second has passed. Once a full second has passed, the `iFPS` variable is set to the number of frames that were rendered during that second. Once that is accomplished, the `dwFrameTimer` is reset and `iFrames` is set to `1`. If a full second has not passed, then the global variable, `iFrames`, is incremented. This variable counts the number of frames that have passed since the last time update.

Outputting Information

Now that the program knows the frames per second, it needs to output this information using the `DrawText()` function. In *RPGQuest*, you are outputting the player's position, frames per second, and rotation. All of these values are displayed in a small Arial font at the top-left corner of the user's screen.

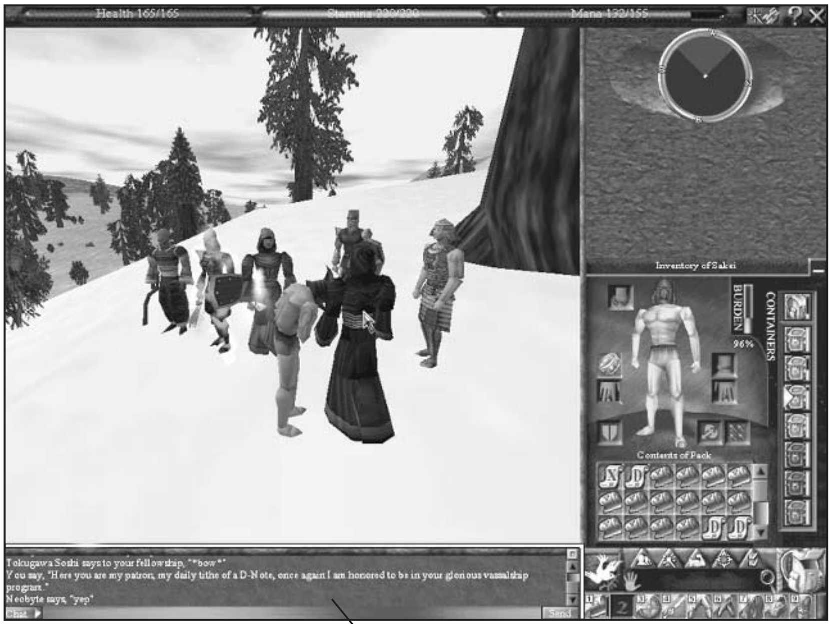
```
// Display the player cords & fps
sprintf(szMessage,"X = %.2f, Z = %.2f, FPS = %d, Rotation = %.2f",
LocalPlayer.vecCurPos.x,LocalPlayer.vecCurPos.z,iFPS,LocalPlayer.fRot);
pD3DXFont->DrawText(szMessage, -1, &rTemp4, DT_LEFT,
 D3DCOLOR_RGBA(255, 255, 255, 255));
```

Displaying the Message Log

If you have ever played *Asheron's Call* or *EverQuest*, you should be familiar with a message box. Basically, the message box conveys messages from the server and from other players. Figure 17.6 shows the message box that *Asheron's Call* uses.

In *RPGQuest*, the only messages sent come from the server or the local client, but you still need a message box. To accomplish this, a loop outputs text in the message buffer.

```
// Output message log
for( i = iMessageLogPos ; i < iLastMessage ; i++ ) {
    rTemp1.bottom = SCREEN_HEIGHT;
    rTemp1.left = 15;
    rTemp1.right = SCREEN_WIDTH;
    rTemp1.top = (SCREEN_HEIGHT-200)+((i-iMessageLogPos)*12);
    pD3DXFont->DrawText(MessageLog[i].szMessage, -1, &rTemp1, DT_LEFT,
     MessageLog[i].color);
}
```



Tokugawa Soshi says to your fellowship, "*bow*"
You say, "Here you are my patron, my daily tithe of a D-Note, once again I am honored to be in your glorious vassalship program."
Neobyte says, "yep"

Figure 17.6

The message output box in Asheron's Call

Because the `DrawText()` function uses a rectangle to determine where it displays text, the message loop adjusts the position of the rectangle for each line of text. Once the rectangle is defined, the text is output.

Presenting the Scene

Now that you have a great 3D scene rendered, you need to present it to the user with the `Present()` function.

```
// Output the scene
if( (pd3dDevice->Present( NULL, NULL, NULL, NULL )) ==
 D3DERR_DEVICELOST ) {
}
```

You might have noticed that the code checks for a device-lost return code. It doesn't do anything if the code is returned, but it does illustrate how to check the return code of the `Present()` function.

Sound and Input Handling

The sound and input routines in *RPGQuest* are identical to those in *SpacePirates*, so I'm not going to spend a heck of a lot of time here. There are a few key functions involved.

Handling DirectInput

Before you can start checking for buffered input from the user, you must initialize DirectInput using the `hrInitDirectInput()` function. The code does little more than create an `IDirectInput8` interface. Once that is accomplished, the keyboard layout is captured for later use.

```
// Do not try to create DirectInput if already created
if( !pDI ) {
    // Create a DInput object
    if( FAILED( hReturn = DirectInput8Create( g_hInstance,
     DIRECTINPUT_VERSION,
IID_IDirectInput8, (VOID**)&pDI, NULL ) ) )
    return(INPUTERROR_NODI);
}
```

Once you initialize DirectInput, you can go on to create the keyboard device using the `hrInitKeyboard()` function.

Initializing the Keyboard Device

In the keyboard initialization routine, the program creates the device and sets up buffers to receive input from the device. Buffered input is important because without it, you miss valuable input from the user. Look at Figure 17.7 to see the steps involved in creating the keyboard device.

In Figure 17.7, you can see that once a keyboard device is created, the code goes on to set up the input buffer using the `SetProperty()` function. The last thing the code must do is acquire the keyboard device with the `Acquire()` function.

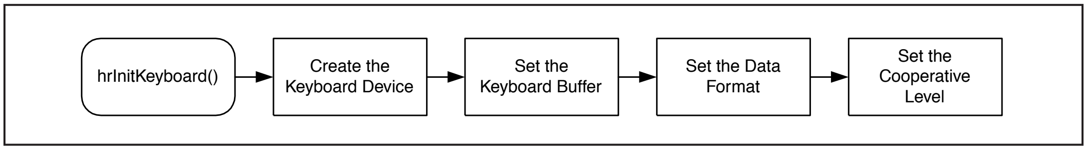


Figure 17.7 *Creating the keyboard device*

Reading the Keyboard

Inside of the WinMain() function, you call the hrReadKeyboard() function. This function updates the key flags stored in the keyboard buffer. When a key is pressed, it is signaled; otherwise, it is equal to 0. The following code checks the cursor keys for their state.

```
if( diks[DIK_LEFT] ) {
    bTurnLeft = 1;
}
if( diks[DIK_RIGHT] ) {
    bTurnRight = 1;
}
if( diks[DIK_UP] ) {
    bAccelerate = 1;
}
if( diks[DIK_DOWN] ) {
    bDeaccelerate = 1;
}
```

The DIK values check the cursor key states for the movement keys in the game. Because all the player can do is move around, these are the only keys that you have to deal with. In the preceding code, global flags are set when the keys are pressed. These flags tell the system to send update communication packets to the game server. The only real difference between this code and that in *SpacePirates* is that the code in *RPGQuest* must send a packet only to the game server and not to a bunch of peers.

The last thing you have to deal with when it comes to DirectInput is the cleanup portion of the game code. The vCleanup() function has code to clean up the DirectInput system.

NOTE

DIK codes for all the main keys supported by DirectInput appear in the dinput.h header file that comes with DirectX.

```
// Release keyboard
if( pKeyboard )
    pKeyboard->Unacquire();
SAFE_RELEASE( pKeyboard );
SAFE_RELEASE( pDI );
```

Before you can release the keyboard and DirectInput interfaces, you must unacquire the keyboard device. This is a very important step to complete because it frees up the keyboard resource. After that step is out of the way, you can use SAFE_RELEASE to free up the device and interface.

Handling DirectSound

If you have launched *RPGQuest* and tried it out, which I hope you have, you might have noticed the lack of sound in the game. The program doesn't actually play any sounds at this time. You might be asking a very normal question: Why is there a section about sound, then? Well, I went ahead and included sound in the project so that you can add your own effects and events as desired, without having to figure out where to put sound routines.

Luckily, the entire sound system is encapsulated in the `SoundSystem` class object you used back in *SpacePirates*. The only function you need to deal with from *SpacePirates* is the `vInitSoundEngine()` function. This function both initializes the sound class object and loads any game sounds you might want to use.

```
void vInitSoundEngine(HWND hWnd)
{
    SoundEngine.hWnd = hWnd;
    if( !FAILED(SoundEngine.hrInitSoundSystem()) ) {
        g_GameSounds.gsFire = new GameSound;
        SoundEngine.hrLoadSound("data\\sfx\\attack.wav",g_GameSounds.gsFire);
    }
}
```

The `hrInitSoundSystem()` member function is called to perform the one-time sound initialization necessary to play sounds in the game. After that is done, the `hrLoadSound()` function actually loads up a game sound. The sound is never played in the game, but nothing is keeping you from changing that!

Network Handling

Hey, look, I saved the best for last. Considering network handling is what this book is all about, you might have expected this section to be the first in the chapter. The truth is that because *RPGQuest* is a culmination of everything learned so far, there is little in the way of new information here. So relax, open up a cold one, and read on about the pure joy of network implementation.

The basic elements of network handling for the *RPGQuest* client involve the following areas.

- Message sending
- Message receiving
- Player synchronization

Sending Messages to the Server

Because all communication in games involves either sending or receiving data, it only seems natural that sending messages to the server is a key element of *RPGQuest*. All message transmission for the client is handled in the `hrSendClientMessage()` function. This function is very similar to its counterpart in *SpacePirates*, `hrSendPeerMessage()`. The main difference here is that messages are only sent to one computer, the game server.

In the *RPGQuest* client, messages are sent to the server in only a few instances.

- When submitting the player password
- When submitting a player action
- When requesting player information

Submitting the Player's Password

It's a safe bet to say that every MMORPG requires a user name and password to gain access. In *RPGQuest*, when the client logs into the game, the server sends a packet requesting the player's password. In response, the player sends his password to the server via a `PACKET_TYPE_SUBMITPASSWORD` type packet. Figure 17.8 illustrates this process.

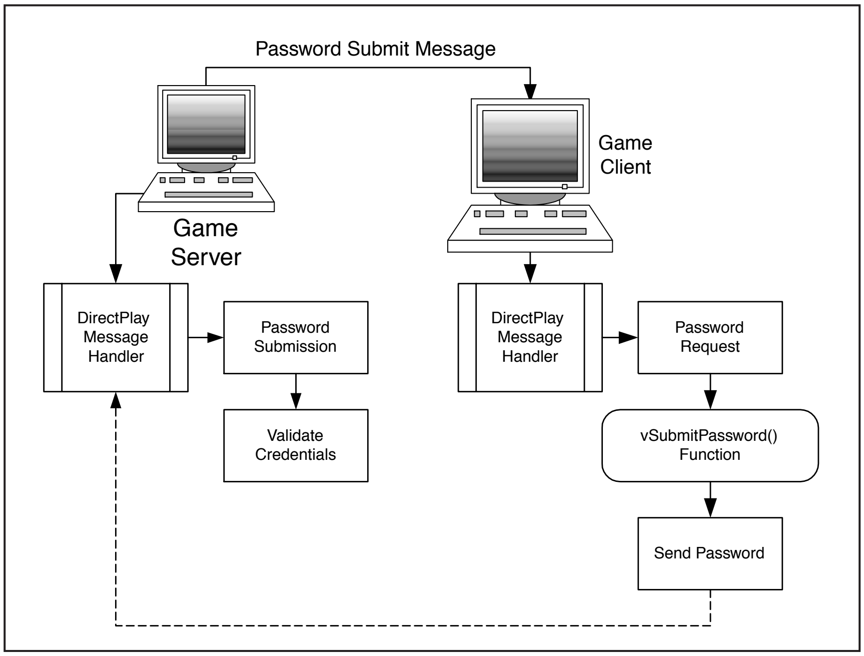


Figure 17.8

The process for submitting the player's password

Submitting a Player Action

When a player decides to accelerate, decelerate, or turn, he needs to inform the server of his intentions. All of this is handled by player action packets, such as the following packet types.

- `PACKET_TYPE_VELOCITY`
- `PACKET_TYPE_ANGLE`

Wow, not a very long list, eh? The truth of the matter is that because the player is not trusted to send position packets, he only has to deal with sending velocity and angle changes.

The `vUpdateObjects()` function is where the actual movement packets are sent. This function can be called up to 30 times per second by the `WinMain()` function. The following flags determine which actions to send to the server during the update process.

- `bTurnLeft`
- `bTurnRight`
- `bAccelerate`
- `bDeaccelerate`

These global variables are set in the `WinMain()` function whenever the player presses a particular key. The left- and right-turn flags are activated when the respective left and right cursor keys are pressed. When the player wants to speed up or slow down, he presses the respective up or down cursor keys. Figure 17.9 illustrates to which packets the keys correspond.

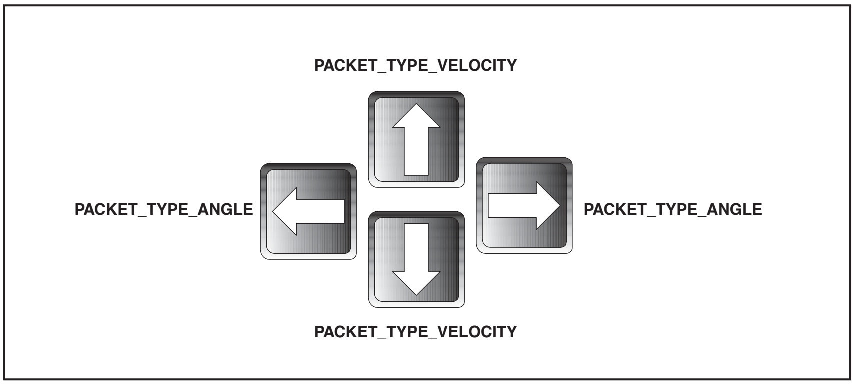


Figure 17.9

Key-to-packet mapping

Requesting Player Information

Whenever a player performs a meaningful action, the server sends his action to all of those involved. To be involved, other players must be in the same zone. Because some packets can get lost, the odds are that players might not be aware of the players about whom they are hearing. Take the following example of two players named Argoth and Belos.

1. Argoth is in zone 1.
2. Belos is in zone 2.
3. Belos walks into zone 1.
4. The server sends a zoning message to Argoth to tell him Belos is present.
5. Argoth misses the packet and doesn't know Belos exists.
6. Belos turns left.
7. The server sends an angle update packet to Argoth.
8. Argoth receives the packet but doesn't recognize Belos as a player.
9. Argoth requests Belos's information with a `PacketRequestPlayerInfo` packet.

In these steps, Argoth ends up requesting Belos's information to find out who he is and what he is all about. The sole purpose of the `PACKET_TYPE_REQUESTPLAYERINFO` is to perform the request.

The logic used to kick off the request packet is as follows.

```
// If the packet ID is not -1, then it is for another player
if( PGen->iPlayerID != -1 ) {
    // Figure out who is sending the message
    iPlayerID = iGetPlayerID(PGen->iPlayerID);
    // If the player does not exist in the game,
    // request information about him
    If( iPlayerID == -1 ) {
        vAddMessage("Requesting Player Information",
        D3DCOLOR_RGBA(255, 255, 255, 255));
        pacRequest.dwSize = sizeof(PacketRequestPlayerInfo);
        pacRequest.dwType = PACKET_TYPE_REQUESTPLAYERINFO;
        pacRequest.iPlayerID = PGen->iPlayerID;
        packet = (VOID*)&pacRequest;
        hReturn = hrSendClientMessage(pacRequest.dwType,packet);
        break;
    }
}
else {
    // Set the target player to ourselves
    iPlayerID = -2;
}
```

The preceding logic, located in the `DirectPlayMessageHandler()` function, checks the `iPlayerID` variable to see whether the packet is about another player in the game. Messages with an `iPlayerID` of `-1` are meant for the local player; messages with a value greater than `0` give information about other players. So when a packet comes in with an ID other than `-1`, you know that it pertains to information about another player. When this happens, the code calls the `iGetPlayerID()` function to match the `iPlayerID` with a currently active player. If no match is found, then a `-1` is returned from the get-player-ID function. When this happens, it means that no information is on record for the player about whom the server is talking. In these cases, the local player needs to request information about the unknown player. This is where the information-request packet comes into play. As you can see in the code, a request packet is sent with the `iPlayerID` variable set to that of the unknown player. All of this might sound a bit confusing, so I drew a flowchart for you in Figure 17.10.

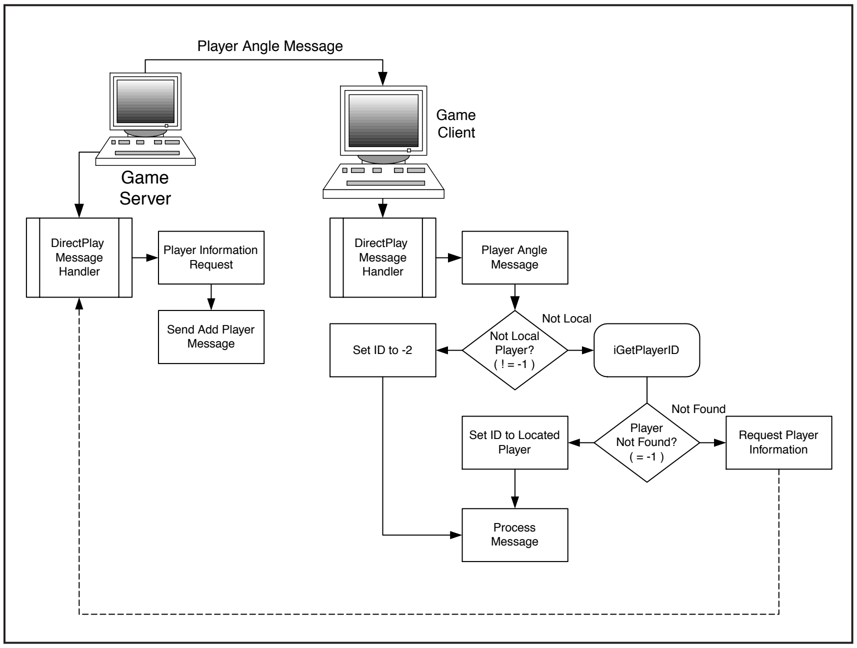


Figure 17.10

Logic for requesting player information

Processing Incoming Messages

Because the *RPGQuest* client lives in a client/server world, all incoming messages come from the game server. This makes life easier because you don't have to worry about packets coming from all over the place. The main function to process all of this traffic is the `DirectPlayMessageHandler()` function. It handles the following types of messages.

- `DPN_MSGID_TERMINATE_SESSION`
- `DPN_MSGID_RECEIVE`
- `PACKET_TYPE_REQUESTPASSWORD`
- `PACKET_TYPE_GAMEUPDATE`
- `PACKET_TYPE_MESSAGE`
- `PACKET_TYPE_ZONECHANGE`
- `PACKET_TYPE_CHANGEPOSITION`
- `PACKET_TYPE_TIMECHANGE`
- `PACKET_TYPE_ADDPLAYER`
- `PACKET_TYPE_REMOVEPLAYER`
- `PACKET_TYPE_ANGLE`
- `PACKET_TYPE_VELOCITY`
- `PACKET_TYPE_POSITION`
- `PACKET_TYPE_ATTITUDE`

The DPN_MSGID_TERMINATE_SESSION Message

When the game server disconnects the local player from the game, the local player receives the terminate-session message. Its main purpose is to let clients know that the game server is no longer available. *RPGQuest* doesn't really do anything with this message, so you might try adding some functionality to it.

```
case DPN_MSGID_TERMINATE_SESSION:
{
    PDPNMSG_TERMINATE_SESSION pTerminateSessionMsg;
    pTerminateSessionMsg = (PDPNMSG_TERMINATE_SESSION)pMsgBuffer;
    break;
}
```

The DPN_MSGID_RECEIVE Message

Whenever the game server sends a non-system message, it is sent as a DPN_MSGID_RECEIVE message. This is how the game knows when to convert messages into game messages.

Once you determine that you have a game message, the first step is to convert it into a generic message to read the type variable. You can do so using the following code.

```
PDPNMSG_RECEIVE pReceiveMsg;
PacketGeneric      *PGen;
pReceiveMsg = (PDPNMSG_RECEIVE)pMsgBuffer;
// Cast it to a generic packet so we can check the type
PGen = (PacketGeneric*)pReceiveMsg->pReceiveData;
```

In the code, the message buffer is first cast as a PDPNMSG_RECEIVE pointer. You can then use this pointer to retrieve the raw message data in the packet. To retrieve the raw message data, the receive data is cast into a PacketGeneric pointer. This lets you look at the dwType variable to determine the message type.

Now that you have the message type, you can perform the logic pertaining to each type of packet.

The PACKET_TYPE_REQUESTPASSWORD Message

Earlier, I covered how to submit your password. This is where you receive the notice to submit that packet. When this type of packet comes in, you know that the server is requesting your password for authentication. To handle this, I created a separate function called vSubmitPassword().

The vSubmitPassword() function reads the config.txt file located in the player's installation directory. Once the password is read in, the function sends it to the server with a PACKET_TYPE_SUBMITPASSWORD message. One item of note is that the packet is sent with the guaranteed flag turned on because the password is such an important packet.

CAUTION

Don't forget that guaranteed packets take a lot more time to send than non-guaranteed ones. Use them sparingly.

The PACKET_TYPE_GAMEUPDATE Message

So the player knows how often the game is updated by the server, you send a game-update message. In each cycle where players are updated, the game server broadcasts the update message. This keeps the player's game world in synchronization with the server. The main reason this is necessary is that slow computers might lag behind. In these cases, you want the slow computers to make up for this speed difference by multiplying their updates accordingly. This is why you use the iUpdates counter—it determines the rate of updates performed by the client. You will find that iUpdates appears in the vUpdateObjects() function.

The PACKET_TYPE_MESSAGE Message

Every once in a while, the game server sends out a text message to players. These messages generally inform the clients about what is going on. For example, when a player zones, the name of the new zone is sent to the player via this messaging mechanism.

```
else if( PGen->dwType == PACKET_TYPE_MESSAGE ) {
    pacMessage = (PacketMessage*)pReceiveMsg->pReceiveData;
    vAddMessage(pacMessage->szMessage,pacMessage->color);
    break;
}
```

Once the message is cast into the proper message format, the vAddMessage() function adds it to the message log. The next rendering cycle for the client displays the actual message. One cool part about the function is that the packet also contains the message color. This lets the server send different message types that are easily discernable by their colors.

NOTE

Asheron's Call **makes good use of colored messages. When you do damage to a creature, the message appears in pink. When a creature does damage to you, the message appears in red. This makes it very easy for players to know what the messages concern without having to read them.**

The PACKET_TYPE_ZONECHANGE Message

The zone-change packet is sent whenever the local player goes into another zone. Because players don't always end up in the same position for each zone, they have to set their position and rotation when this message is received. The rest of the logic takes care of setting the proper zone graphic/map to use and setting the player's velocity to `0`.

The PACKET_TYPE_CHANGEPOSITION Message

The change-position packet tells a player to change position. This packet is actually sent to the player before a zoning packet is fired off. One thing this accomplishes is setting the player's velocity to `0`.

The PACKET_TYPE_TIMECHANGE Message

Whenever the game server changes the time of day, the information is sent to all players in the form of a time-change packet. As you learned earlier, the `vUpdateTimeOfDay()` function handles the new time setting.

The PACKET_TYPE_ADDPLAYER Message

Whenever another player enters your zone, the server sends an add-player message. This provides you with everything you need to know about the player, such as his name, position, and angle. All of the data is stored in a `CLIENT_PLAYER_INFORMATION` data structure. The `iAddPlayer()` function finds a new player slot and copies the data structure to his information block. The flow for the `iAddPlayer()` function appears in Figure 17.11.

The PACKET_TYPE_REMOVEPLAYER Message

The remove-player message is the arch-nemesis of the add-player message. This message's purpose is to let you know that a player has either disconnected or left your zone. In this case, you loop through your list of active players and deactivate the matching player ID. All of this logic is handled in the `vRemovePlayer()` function.

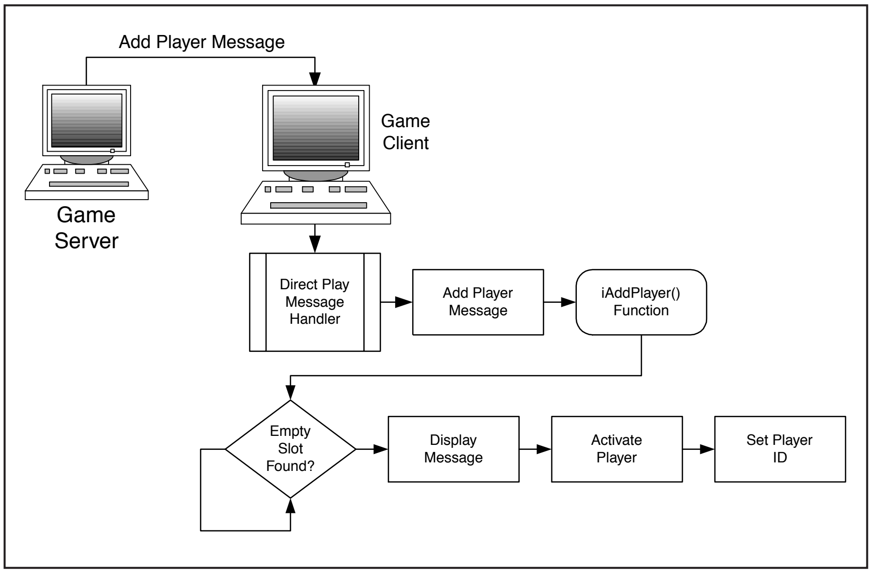


Figure 17.11

The flow of the `iAddPlayer()` *function*

The PACKET_TYPE_ANGLE Message

As you'll find with the rest of the game message types, you can use the angle message to send angle information either about your player or about other players. There isn't much to it, really. The main logic branch checks whether the packet is for you. If its ID is set to `-2`, then you know it is for your player; otherwise, it is for another player.

```
if( PGen->dwType == PACKET_TYPE_ANGLE ) {
    pacAngle = (PacketAngle*)pReceiveMsg->pReceiveData;
    EnterCriticalSection( &g_csModifyPlayer );
    if( iPlayerID != -2 )
        PlayerInfo[iPlayerID].fRot = pacAngle->fAngle;
    else
         LocalPlayer.fRot = pacAngle->fAngle;
         LeaveCriticalSection( &g_csModifyPlayer );
}
```

The PACKET_TYPE_VELOCITY Message

The `PACKET_TYPE_VELOCITY` message works just like the angle message except that it updates the velocity of the player.

The PACKET_TYPE_POSITION Message

The PACKET_TYPE_POSITION message works just like the angle message except that it updates the position of the player.

The PACKET_TYPE_ATTITUDE Message

The PACKET_TYPE_ATTITUDE message works just like the angle message except that it updates the attitude of the player. The attitude for the player is a culmination of his position, rotation, and velocity. It makes for less network traffic when you send all of the information in a single packet.

TIP

Although you should strive for packets under 1,024 bytes, you should also avoid sending many small packets. Each packet requires a hidden header, which eats up bandwidth. One efficient method of packet transmission is sending a single fixed-size update packet that contains many small packets.

Player Synchronization

Although the game server is responsible for keeping everyone informed about the game world, clients still have to use some synchronization logic. The main purpose of this logic is to guess where other players will be in the future. One reason this is needed is the event of lag. If you quit receiving messages from the server for some reason, you don't want other players to grind to a halt. Instead, the other players should continue in their current direction without interruption.

This effect is very noticeable in most online games. Take a look around, and sometimes you'll see other players head in a straight line for a few seconds only to pop back into place elsewhere. This happens because you estimated where they were going while the connection to the server lagged. Once you started getting packets again, you were informed that those players were not where you expected; therefore, your client program put them into the proper place. *RPGQuest* doesn't include the code to smooth out client corrections, but it does do some predicting.

Take a look at the vUpdateObjects() function, and you'll see where player movement is predicted. Player-action requests are also sent within this function. The player-action requests should look very familiar to you because they are almost exact duplicates of the ones in *SpacePirates*.

The actual prediction code is similar to the code in *SpacePirates*. Basically, it loops through the list of active players and adjusts their positions based upon their velocity and angle. The code also slows the players down by multiplying their velocity by 50 percent. You do this so that players come to a natural stop without having to explicitly do so.

The last bit of update code updates your own player. If you didn't update yourself, then you would have to rely on the server to tell you where you were at all times. That type of scenario results in extremely jerky movement.

Review

You might not like review sections, but guess what? This is the last one you have to read. Luckily, the client program is a rehash of everything learned so far, so there are not many new points to remember. Here are two points to keep in mind, in all their glory.

- Be sure to validate data sent to the client. Just as in the time-of-day update function, you should not completely trust values received. By following this rule, you will eliminate program problems caused by bad packets.
- Use guaranteed message delivery for vital packets only. Guaranteed messaging takes up a lot of time and bandwidth.

Oh, no! You are done with the final chapter! Okay, wait; I have something more for you. Read the appendixes for some great game-development resource information and other tidbits of knowledge.

This page intentionally left blank

Part Five

Appendixes

APPENDIX A

Game-Development Resources

You can find game-development information in many places, but nothing comes close to the amount of information available on the Internet. In this appendix, I'll share with you some of the places I like to visit for game-development information.

Gaming Sites

The following sites are good at providing some of the latest information about the gaming industry and the games scene in general.

http://www.msxbox.com. This is my favorite site for information about Microsoft's Xbox game console. It has up-to-date news and information about the console.

http://www.gamespot.com/gamespot. This site is a good place to read about new games that are coming out.

http://www.happypuppy.com. This is a great site for downloading the latest game demos. It also has several game reviews for all genres.

Development Sites

The following sites deal primarily with game-development news, articles, and forums. They should be able to provide you with answers to any game-development questions and needs you might have.

http://www.lostlogic.com. Lost Logic Inc. is the company I own and operate, and this site is where you'll find updates, fixes, or additions to this book. I plan to add several articles and information not covered in the book, so visit when you get a chance.

http://www.gamedev.net. Gamedev.net is a great site that has many articles on all aspects of games programming. I especially like its message forums. This is a great place to pose questions about any development problem you might have.

http://www.flipcode.com. The Flip Code site is similar to Gamedev.net, but it also offers a few unique items of interest, such as the screenshot of the day and contests.

http://www.microsoft.com/directx. This page contains information about the latest versions of DirectX. You can also download the latest version of the API from this site.

Product Sites

The following sites are specific to different commercial products. You might find these products useful in your game-development projects.

http://www.fastgraph.com. Fastgraph is a great programming library that reduces the overhead of graphics programming. The latest version even includes a great 3D software-rendering system that makes 3D graphics programming a breeze. Fastgraph is especially useful if you find DirectX's graphics a bit difficult to use. A demo version of Fastgraph/Light 6.0 is available on this book's CD-ROM.

http://www.discreet.com. Discreet is the maker of 3D Studio MAX. I believe 3D Studio MAX to be the best 3D animation and modeling software available. Check out this site if you are interested in purchasing a modeling package.

http://www.garagegames.com. This site offers publishing services for indie developers. You might also want to check it out for ready-made game engines.

http://www.asheronscall.com. This is the main page for one of my favorite MMORPGs—Microsoft's *Asheron's Call*.

Magazines

The Internet is full of great information, and you can find great content in online periodicals.

http://www.gdmag.com. This is a great magazine for game-development articles and news.

http://www.next-generation.com. Unlike many gaming magazines, NextGen has stories that pertain to the business side of the gaming industry. The job-offer advertisements at the back of the magazine are one of the best features for aspiring game developers.

APPENDIX B

Installing the Software

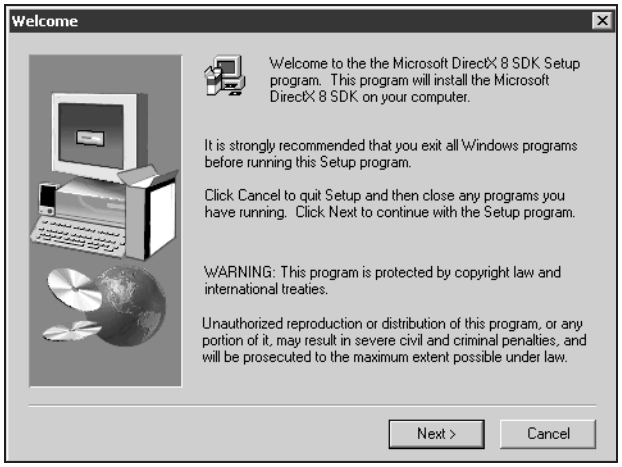
One of the most important software components you need to use this book is the DirectX SDK. Without this software development kit, you will not be able to use most of the code. This appendix takes you through the following steps.

- Installing the DirectX 8 SDK
- Configuring Microsoft's Visual C++ for the DirectX 8 SDK

Installing the DirectX 8 SDK

On the CD that comes with this book, you will find a directory containing the setup program to install the DirectX 8 SDK. Launch the setup program now to begin installation.

The first screen you see is a standard introduction page that prompts you to continue. Go ahead and select Next. Figure B.1 illustrates what you should see on your screen.



Welcome

Welcome to the the Microsoft DirectX 8 SDK Setup program. This program will install the Microsoft DirectX 8 SDK on your computer.

It is strongly recommended that you exit all Windows programs before running this Setup program.

Click Cancel to quit Setup and then close any programs you have running. Click Next to continue with the Setup program.

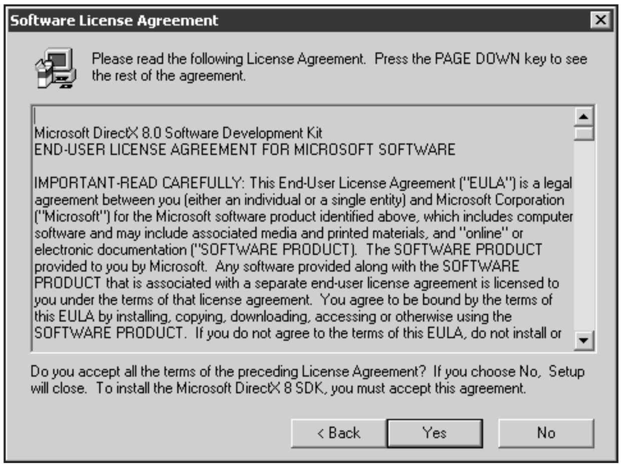
WARNING: This program is protected by copyright law and international treaties.

Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law.

Figure B.1

The DirectX 8 SDK installation welcome screen

After you view the introduction page and select Next, you are greeted with the software license agreement for the SDK. Feel free to consult your lawyer at this time, or if you are like most people, select Yes to continue. The license screen appears in Figure B.2.



Software License Agreement

Please read the following License Agreement. Press the PAGE DOWN key to see the rest of the agreement.

Microsoft DirectX 8.0 Software Development Kit
END-USER LICENSE AGREEMENT FOR MICROSOFT SOFTWARE

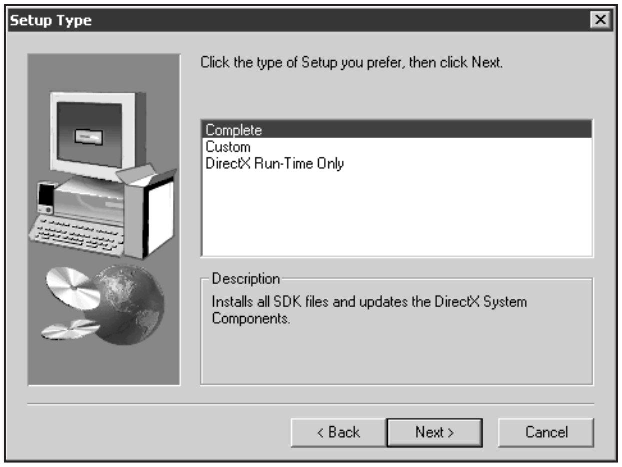
IMPORTANT-READ CAREFULLY: This End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and Microsoft Corporation ("Microsoft") for the Microsoft software product identified above, which includes computer software and may include associated media and printed materials, and "online" or electronic documentation ("SOFTWARE PRODUCT). The SOFTWARE PRODUCT provided to you by Microsoft. Any software provided along with the SOFTWARE PRODUCT that is associated with a separate end-user license agreement is licensed to you under the terms of that license agreement. You agree to be bound by the terms of this EULA by installing, copying, downloading, accessing or otherwise using the SOFTWARE PRODUCT. If you do not agree to the terms of this EULA, do not install or

Do you accept all the terms of the preceding License Agreement? If you choose No, Setup will close. To install the Microsoft DirectX 8 SDK, you must accept this agreement.

Figure B.2

The installation license screen

Once you select Yes, you are greeted with the screen in Figure B.3.



Setup Type

Click the type of Setup you prefer, then click Next.

Complete
Custom
DirectX Run-Time Only

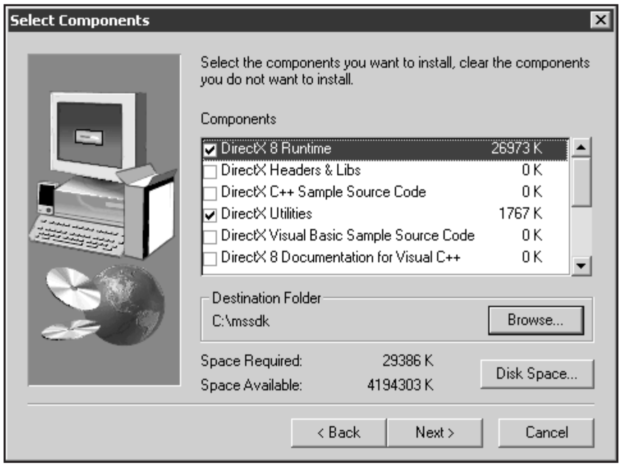
Description

Installs all SDK files and updates the DirectX System Components.

Figure B.3

The setup type selection screen

The screen in Figure B.3 gives you the option to install the Complete, Custom, or DirectX Run-Time Only versions of the software. If you were a non-developer, you might install the Run-Time Only version, but you are a developer, so leave the selection at Complete and select Next. Now, if you have a rebellious streak and you decide to select Custom, you'll see the screen in Figure B.4.



Select the components you want to install, clear the components you do not want to install.

☑ DirectX 8 Runtime	26973 K
☐ DirectX Headers & Libs	0 K
☐ DirectX C++ Sample Source Code	0 K
☑ DirectX Utilities	1767 K
☐ DirectX Visual Basic Sample Source Code	0 K
☐ DirectX 8 Documentation for Visual C++	0 K

Destination Folder: C:\mssdk

Space Required: 29386 K
Space Available: 4194303 K

Figure B.4

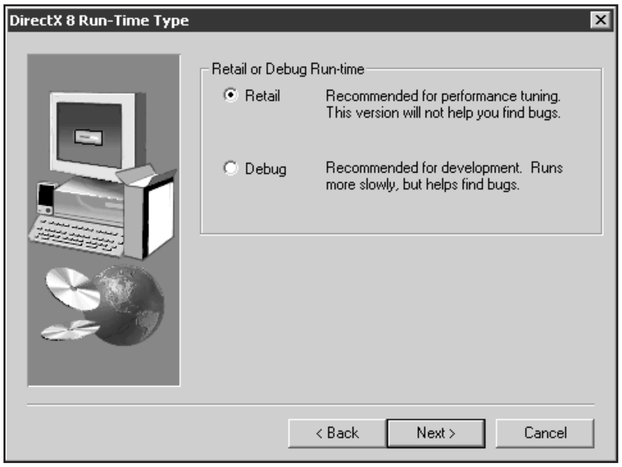
The Custom setup selection screen

As long as you selected Complete, the next screen you'll see asks whether you want to install the Retail or Debug version of DirectX. The Retail version offers the most speed but limits the amount of information shown when you debug your applications. The Debug version is slower than the Retail version, but it gives you much more information when debugging a DirectX game. The choice is yours, so make it and select Next. Oh, yes, this wonderful screen appears in Figure B.5.

The next screen asks you where you want to install the SDK. I personally keep all of my source code in C:\source, so I put DirectX into C:\source\directx8. Go ahead and browse to wherever you want the code, and select Next. You should be able to follow the rest of the prompts without much trouble. Once the installation is complete, you're almost ready to go!

TIP

If you have Visual C++ installed, the DirectX 8 SDK installation program prompts you to indicate whether you want the libraries put into your compiler's paths. Select Yes at the prompt so that your paths will be correct when compiling.



DirectX 8 Run-Time Type

Retail or Debug Run-time

(●) Retail — Recommended for performance tuning. This version will not help you find bugs.

( ) Debug — Recommended for development. Runs more slowly, but helps find bugs.

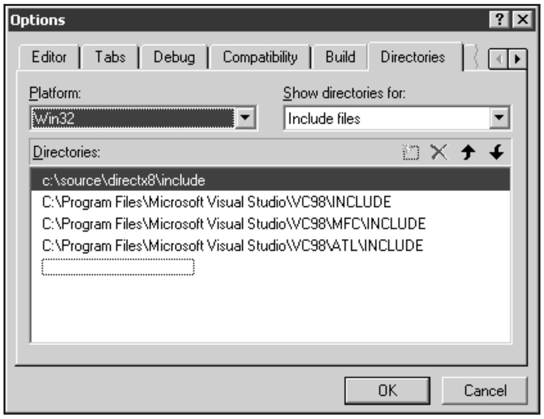
Figure B.5

The runtime type selection screen

Configuring the Microsoft Visual C++ Compiler

If you own Microsoft's Visual C++ compiler, you need to set up your paths to compile the source code that comes with this book. If you chose to let the DX8 SDK installation program set up your paths, you're already halfway there.

Launch the Visual C++ integrated development environment now. Once you have the program running, go to the Tools menu and select Options. Once you have the Options menu open, select the Directories tab. If you have followed these steps successfully, your screen should resemble Figure B.6.



Options

Platform: Win32

Show directories for: Include files

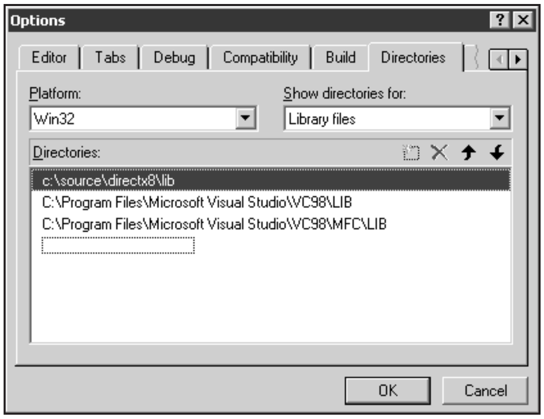
Directories:

```
c:\source\directx8\include
C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\ATL\INCLUDE
```

Figure B.6

The Options dialog box in the Visual C++ program

Figure B.6 shows an example of what your current include paths might look like, and Figure B.7 shows what your library paths might look like. The drop-down menu labeled Show Directories For is where you select which paths you are editing. In this drop-down menu, you'll find selections for executable, include, library, and source files.



Platform: Win32

Show directories for: Library files

Directories:

```
c:\source\directx8\lib
C:\Program Files\Microsoft Visual Studio\VC98\LIB
C:\Program Files\Microsoft Visual Studio\VC98\MFC\LIB
```

Figure B.7

The library paths

To add a path, double-click your left-mouse button on the next available entry in the Directories box. Once you do that, you can either type in the path manually or select the Browse button to browse for it.

Go ahead and add the following paths to your include paths.

CAUTION

The following path settings assume you have installed the DirectX 8 SDK to C:\source\directx8. If you have installed to another path, replace the C:\source\directx8\ base path with yours.

- C:\source\directx8\include
- C:\source\directx8\samples\multimedia\common\include

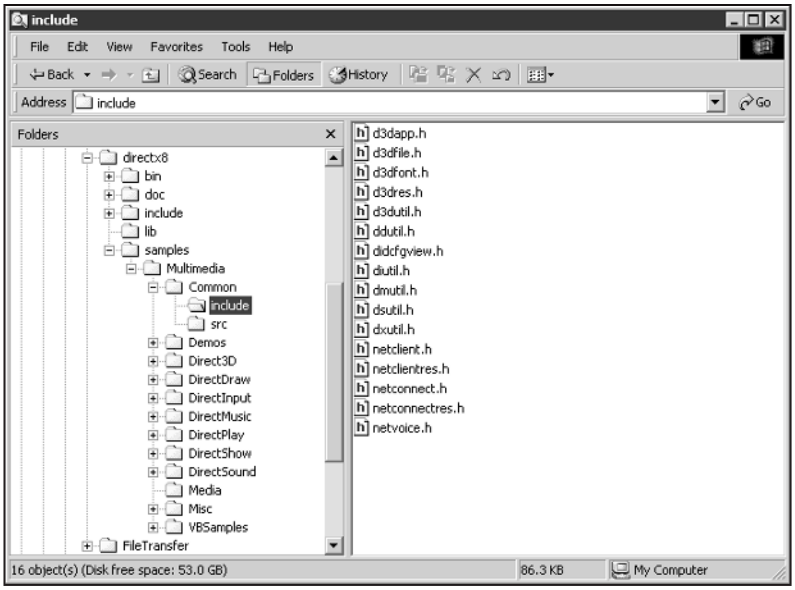
Add the following path to your library paths.

- C:\source\directx8\lib

Add the following path to your source paths.

- C:\source\directx8\samples\multimedia\common\src

To see the files you are gaining access to, open Windows Explorer and browse to C:\source\directx8\samples\multimedia\common\include. Here you see some of the extra files to which you are creating paths. The most common file you'll use here is the dxutil.h header file that contains useful helper code. Figure B.8 shows you the directory listing for the common include folder.



d3dapp.h
d3dfile.h
d3dfont.h
d3dres.h
d3dutil.h
ddutil.h
didcfgview.h
diutil.h
dmutil.h
dsutil.h
dxutil.h
netclient.h
netclientres.h
netconnect.h
netconnectres.h
netvoice.h

16 object(s) (Disk free space: 53.0 GB) | 86.3 KB | My Computer

Figure B.8

Files in the common include folder for the DX8 SDK

Before you leave, you should make yourself familiar with the directory structure of where you installed the DX8 SDK. You might need to access some of the programs in there later.

APPENDIX C

ASCII Chart

Dec	Hex	ASCII	Dec	Hex	ASCII
000	00	null	032	20	space
001	01	☺	033	21	!
002	02	☻	034	22	“
003	03	♥	035	23	#
004	04	♦	036	24	$
005	05	♣	037	25	%
006	06	♠	038	26	&
007	07	•	039	27	‘
008	08	◘	040	28	(
009	09	○	041	29	)
010	0A	◙	042	2A	*
011	0B	♂	043	2B	+
012	0C	♀	044	2C	’
013	0D	♪	045	2D	-
014	0E	♫	046	2E	.
015	0F	☼	047	2F	/
016	10	►	048	30	0
017	11	◄	049	31	1
018	12	↕	050	32	2
019	13	‼	051	33	3
020	14	¶	052	34	4
021	15	§	053	35	5
022	16	–	054	36	6
023	17	↨	055	37	7
024	18	↑	056	38	8
025	19	↓	057	39	9
026	1A	→	058	3A	:
027	1B	←	059	3B	;
028	1C	∟	060	3C	<
029	1D	↔	061	3D	=
030	1E	▲	062	3E	>
031	1F	▼	063	3F	?

Dec	Hex	ASCII	Dec	Hex	ASCII
064	40	@	096	60	`
065	41	A	097	61	a
066	42	B	098	62	b
067	43	C	099	63	c
068	44	D	100	64	d
069	45	E	101	65	e
070	46	F	102	66	f
071	47	G	103	67	g
072	48	H	104	68	h
073	49	I	105	69	i
074	4A	J	106	6A	j
075	4B	K	107	6B	k
076	4C	L	108	6C	l
077	4D	M	109	6D	m
78	4E	N	110	6E	n
079	4F	O	111	6F	o
080	50	P	112	70	p
081	51	Q	113	71	q
082	52	R	114	72	r
083	53	S	115	73	s
084	54	T	116	74	t
085	55	U	117	75	u
086	56	V	118	76	v
087	57	W	119	77	w
088	58	X	120	78	x
089	59	Y	121	79	y
090	5A	Z	122	7A	z
091	5B	[	123	7B	{
092	5C	\	124	7C	¦
093	5D	]	125	7D	}
094	5E	^	126	7E	~
095	5F	–	127	7F	Δ

Dec	Hex	ASCII	Dec	Hex	ASCII
128	80	Ç	160	A0	á
129	81	ü	161	A1	í
130	82	é	162	A2	ó
131	83	â	163	A3	ú
132	84	ä	164	A4	ñ
133	85	à	165	A5	Ñ
134	86	å	166	A6	ª
135	87	ç	167	A7	º
136	88	ê	168	A8	¿
137	89	ë	169	A9	⌐
138	8A	è	170	AA	¬
139	8B	ï	171	AB	½
140	8C	î	172	AC	¼
141	8D	ì	173	AD	¡
142	8E	Ä	174	AE	«
143	8F	Å	175	AF	»
144	90	É	176	B0	█
145	91	æ	177	B1	█
146	92	Æ	178	B2	█
147	93	ô	179	B3	│
148	94	ö	180	B4	┤
149	95	ò	181	B5	╡
150	96	û	182	B6	╢
151	97	ù	183	B7	╖
152	98	ÿ	184	B8	╕
153	99	Ö	185	B9	╣
154	9A	Ü	186	BA	║
155	9B	¢	187	BB	╗
156	9C	£	188	BC	╝
157	9D	¥	189	BD	╜
158	9E	₧	190	BE	╛
159	9F	ƒ	191	BF	┐

Dec	Hex	ASCII	Dec	Hex	ASCII
192	C0	└	224	E0	α
193	C1	┴	225	E1	β
194	C2	┬	226	E2	Γ
195	C3	├	227	E3	π
196	C4	─	228	E4	Σ
197	C5	+	229	E5	σ
198	C6	╞	230	E6	µ
199	C7	╟	231	E7	γ
200	C8	╚	232	E8	Φ
201	C9	╔	233	E9	Θ
202	CA	╦	234	EA	Ω
203	CB	╦	235	EB	δ
204	CC	╠	236	EC	∞
205	CD	═	237	ED	∅
206	CE	╬	238	EE	∈
207	CF	╧	239	EF	∩
208	D0	╨	240	F0	≡
209	D1	╤	241	F1	±
210	D2	╥	242	F2	≥
211	D3	╙	243	F3	≤
212	D4	╘	244	F4	⌠
213	D5	╒	245	F5	⌡
214	D6	╓	246	F6	±
215	D7	╫	247	F7	≈
216	D8	╪	248	F8	°
217	D9	┌	249	F9	•
218	DA	┌	250	FA	·
219	DB	█	251	FB	√
220	DC	▄	252	FC	^{n}
221	DD	▌	253	FD	^{2}
222	DE	▐	254	FE	■
223	DF	▀	255	FF	

APPENDIX D

What's On the CD

The CD that comes with this book contains some very important software that you need to copy to your machine. Here is the run-down of what is included.

Source Code

All of the source code for the in-book programs is located on the CD. Make sure you install the source code to get the most out of the book.

DirectX 8 SDK

If you don't already have at least DirectX 8.0 installed on your machine, you need to install it from the book's CD. You must have it if you want to compile or use most of the code discussed in the book.

Games

You'll also find on the CD a trial version of a cool robot combat game by Microsoft, called *MechWarrior 4: Vengeance*. It has a very decent networking system that you might want to analyze. It also happens to be a pretty cool game! Also included is a trial version of another Microsoft game, called *Crimson Skies*.

Fastgraph/Light 6.0

Fastgraph is a cool alternative to DirectX. In fact, you can use the two together. It is a programmer's library for doing all sorts of graphic operations. The nice thing about it is that it makes game development easier on beginners. I highly recommend it if you want an easy way into programming game graphics. The version on the CD is an evaluation copy, but don't let that fool you; it is extremely powerful!

Index