

PRIMA TECH'S

# GAME DEVELOPMENT SERIES

CD INCLUDED

# BEGINNING DIRECT3D GAME PROGRAMMING

Wolfgang F. Engel  
Amir Geva

Series Editor  
André LaMothe  
CEO Xtreme Games LLC





# BEGINNING DIRECT3D® GAME PROGRAMMING

GAME DEVELOPMENT GAME DEVELOPMENT GAME

## **CHECK THE WEB FOR UPDATES!**

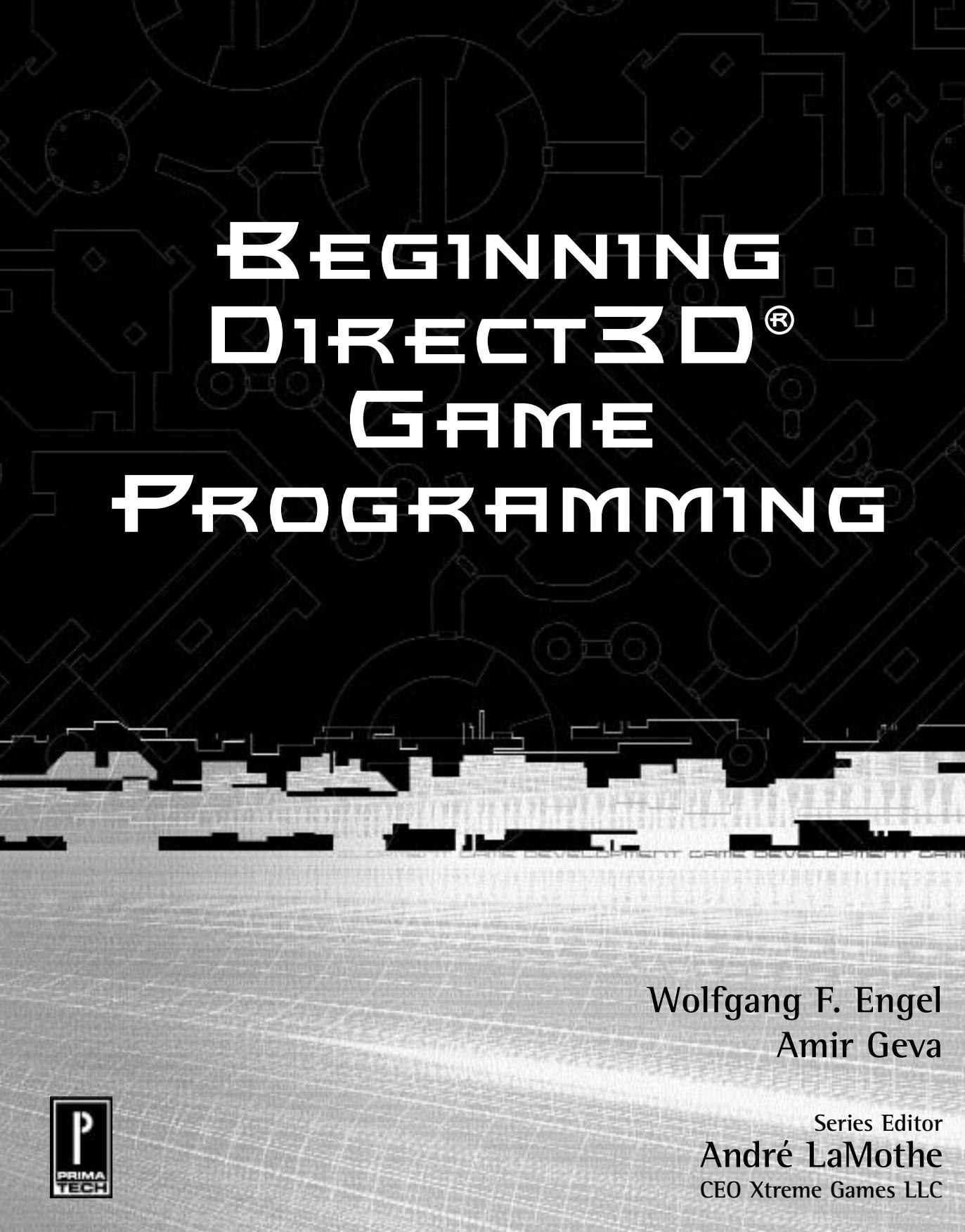
To check for updates or corrections relevant to this book and/or CD-ROM visit our updates page on the Web at <http://www.prima-tech.com/updates>.

## **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.



# BEGINNING DIRECT3D® GAME PROGRAMMING

Wolfgang F. Engel  
Amir Geva



Series Editor  
André LaMothe  
CEO Xtreme Games LLC

©2001 by Prima Publishing. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system without written permission from Prima Publishing, except for the inclusion of brief quotations in a review.



A Division of Prima Publishing

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

**Publisher:** Stacy L. Hiquet

**Managing Editor:** Sandy Doell

**Acquisitions Editor:** Emi Smith

**Associate Marketing Manager:** Jenni Breece

**Technical Reviewer:** Mason McCuskey

**Book Production and Editorial:** Argosy

**Cover Design:** Prima Design Team

Copyright Microsoft Corporation, 2000. All rights reserved.

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

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

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

ISBN: 0-7615-3191-2

Library of Congress Catalog Card Number: 0-011047

Printed in the United States of America.

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

*Für meine Frau, Katja Engel*

## ACKNOWLEDGMENTS

This book couldn't have been completed without the help of many people. In particular, I want to thank my parents, who gave me a wonderful and warm childhood.

The first 90% of a book is normally easy to write. The problems arise in the second 90%. That was the case with this book. The last four weeks of finishing up this project were really hard, both in my private and professional life. So my corrections and reviews of edits were sometimes a little bit behind schedule. Nevertheless, the team at Prima Publishing was very friendly and sensible. I would like to thank those people, who also helped to make this book possible: Caroline Roop, Emi Smith, and Eve Minkoff.

I would also like to thank André LaMothe, for teaching me game programming with his books.

A lot of people wrote tutorials on game programming and published them on the Internet. I learned a lot from these. So I would like to thank all those people on the Internet, for giving away their knowledge of game programming for free.

—Wolfgang Engel

## CONTENTS AT A GLANCE

<i>Introduction</i> .....	xiv
<b>Part I DirectX Graphics:</b>	
<b>Don't Hurt Me</b> .....	1
<i>Chapter 1: History of Direct3D/DirectX Graphics</i> .....	3
<i>Chapter 2: Overview on DirectX Graphics/HAL/COM</i> .....	7
<i>Chapter 3: C++/COM Programming Rules for Direct3D</i> .....	15
<i>Chapter 4: Geometry/Shading/Texture Mapping Basics</i> .....	23
<i>Chapter 5: The Basics</i> .....	35
<i>Chapter 6: First Steps to Animation</i> .....	73
<b>Part II Knee-Deep in DirectX Graphics Programming</b> .....	125
<i>Chapter 7: Texture Mapping Fundamentals</i> .....	127
<i>Chapter 8: Using Multiple Textures</i> .....	153
<b>Part III Hardcore DirectX Graphics Programming</b> .....	209
<i>Chapter 9: Working with Files</i> .....	211
<i>Chapter 10: Quake 3 Model Files</i> .....	247
<i>Chapter 11: Game Physics (written by Amir Geva)</i> .....	291
<i>Chapter 12: Collision Detection (written by Amir Geva)</i> .....	301
<b>Part IV Appendices</b> .....	343
<i>Appendix A: Windows Game Programming Foundation</i> .....	345
<i>Appendix B: C++ Primer</i> .....	373
<i>Appendix C: The Common Files Framework</i> .....	401
<i>Appendix D: Mathematic Primer</i> .....	463
<i>Appendix E: Game Programming Resource</i> .....	485
<i>Index</i> .....	489

## CONTENTS

Introduction .....	xiv
<b>Part I: DirectX Graphics: Don't Hurt Me.....</b>	<b>1</b>
<i>Chapter 1: History of Direct3D/ DirectX Graphics.....</i>	<i>3</i>
<i>Chapter 2: Overview of DirectX Graphics/HAL/COM .....</i>	<i>7</i>
Direct3D HAL .....	9
Pluggable Software Devices.....	11
Reference Rasterizer .....	12
Controlling Devices.....	12
COM .....	13
<i>Chapter 3: C/C++ and COM Programming Rules for Direct3D .....</i>	<i>15</i>
Code Style .....	18
Debugging DirectX.....	20
Return Codes.....	21
<i>Chapter 4: Geometry/Shading/ Texture-Mapping Basics.....</i>	<i>23</i>
Orientation .....	26
Faces .....	27
Normals.....	29
Normals and Gouraud Shading.....	29
Texture-Mapping Basics.....	31
<i>Chapter 5: The Basics.....</i>	<i>35</i>
The DirectX Graphics Common Architecture.....	37
Basic Example .....	38
OneTimeSceneInit() .....	40
InitDeviceObjects() .....	41
RestoreDeviceObjects() .....	41
FrameMove() .....	51

Render() .....	51
InvalidateDeviceObjects() .....	55
DeleteDeviceObjects() .....	55
FinalCleanup() .....	55
Basic2 Example .....	55
InitDeviceObjects() .....	57
RestoreDeviceObjects() .....	60
Render() .....	62
InvalidateDeviceObjects() .....	64
DeleteDeviceObjects() .....	65
FinalCleanup() .....	65
Basic3 Example .....	65
RestoreDeviceObjects() .....	68
Render() .....	70
InvalidateDeviceObjects() .....	72
<i>Chapter 6: First Steps to Animation .....</i>	<i>73</i>
The Third Dimension .....	75
Transformation Pipeline .....	77
Transformation Math.....	80
Matrices.....	80
The World Matrix.....	82
The View Matrix.....	86
Camera Rotation about a Camera Axis .....	88
Camera Rotation with Quaternions .....	92
The Projection Matrix .....	95
Lighting.....	96
Material .....	97
Lighting Models .....	97
Vertex Color (Optional) .....	99
Depth Buffering .....	100
Down to the Code.....	104
OneTimeSceneInit() .....	105
InitDeviceObjects() .....	110
RestoreDeviceObjects() .....	110
FrameMove() .....	113
Render() .....	117
InvalidateDeviceObjects() .....	118

DeleteDeviceObjects()	119
FinalCleanup()	119
Next Steps to Animation	119
RestoreDeviceObjects()	119
FrameMove()	120
More Enhancements	120
Quiz	120
Additional Resources	124
<b>Part II: Knee-Deep in DirectX Graphics Programming</b>	<b>125</b>
<b>Chapter 7: Texture-Mapping Fundamentals</b>	<b>127</b>
Texture Coordinates	129
Texture-Addressing Modes	132
Wrap Texture-Addressing Mode	132
Mirror Texture-Addressing Mode	133
Clamp Texture-Addressing Mode	134
Border Color Texture-Addressing Mode	135
MirrorOnce Texture-Addressing Mode	136
Texture Wrapping	137
Texture Filtering and Texture Anti-Aliasing	140
Mipmaps	142
Nearest-Point Sampling	143
Linear Texture Filteringing	143
Anisotropic Filtering	144
Full-Scene Anti-Aliasing	146
Alpha Blending	148
<b>Chapter 8: Using Multiple Textures</b>	<b>153</b>
Color Operations	156
Dark Mapping	157
Animating the Dark	161
Blending a Texture with Material Diffuse Color	161
Dark Map Blended with Material Diffuse Color	164
Glow Mapping	166
Detail Mapping	167
Alpha Operations	170
Modulate Alpha	172
Environment Mapping	173
Spherical Environment Mapping	173
Cubic Environment Mapping	175
RestoreDeviceObjects()	176
RenderSceneIntoCube()	177
RenderScene()	179
ConfirmDevice()	181
Bump Mapping	182
ApplyEnvironmentMap()	183
InitBumpMap()	187
Render()	191
ConfirmDevice()	193
Dot Product Texture Blending	194
InitDeviceObjects()	195
Render()	198
Multitexturing Support	200
Texture Management	201
Quiz	201
Additional Resources	206
Anisotropy	207
Detail Mapping	207
Cubic Environment Mapping	207
Stencil Buffers	207
Bump Mapping	207
Dot Product Texture Blending	207
<b>Part III: Hard-Core DirectX Graphics Programming</b>	<b>209</b>
<b>Chapter 9: Working with Files</b>	<b>211</b>
Building Worlds with X Files	212
3-D File Formats	213
X File Format	213

Header .....	215
Mesh .....	216
MeshMaterialList.....	218
Normals.....	221
Textures.....	221
Transformation Matrices .....	229
Animation .....	233
Using X Files .....	235
The Example .....	241
InitDeviceObjects().....	242
RestoreDeviceObjects() and InvalidateDeviceObjects() .....	243
Render() .....	243
Extending X Files.....	245
Additional Resources .....	245
<b>Chapter 10: Quake III</b>	
<b>Model Files.....</b>	<b>247</b>
Files of the Trade .....	249
Animation.cfg.....	253
The .skin File .....	255
Textures and the Shader File.....	256
Custom Sounds .....	262
The .md3 Format.....	263
Md3.h .....	264
Md3.cpp .....	271
CreateModel() .....	271
CreateTextures() .....	278
CreateVB() .....	282
Render() .....	283
DeleteTextures() .....	284
DeleteVB() .....	285
DeleteModel() .....	285
Md3view.cpp .....	285
OneTimeSceneInit() .....	286
InitDeviceObjects() .....	286
Render() .....	287
DeleteDeviceObjects() .....	287
FinalCleanup() .....	288
MsgProc() .....	288
Additional Resources .....	289

## **Chapter 11: Game Physics**

*(written by Amir Geva) .....***291**

3-D Math .....	292
Newton's Laws .....	294
Calculating the Frame Time .....	296
Air Resistance .....	297
Static Friction .....	297
Kinetic Friction .....	298

## **Chapter 12: Collision Detection**

*(written by Amir Geva) .....***301**

The Most Basic Optimization .....	302
Bounding Volumes .....	302
2-D Collision Detection .....	302
Brute Force .....	304
Bit Arrays .....	304
Sprite Bounds .....	309
Group Processing.....	313
Axis Sort .....	314
Grid.....	316
Static Objects .....	317
Automatic Transparent Static Marking....	319
3-D Collision Detection .....	319
Dealing with this Complex Problem .....	320
Portals .....	320
Calculating Distance of Cylinder from Wall .....	322
BSP (binary space partitioning) .....	323
Sliding Off Walls.....	323
3-D Mesh Collision Detection .....	324
Bounding Volumes.....	324
Convexity of Models.....	325
Convex Models Intersection.....	326
Concave Models Intersection .....	326
Axis Aligned Bounding Boxes.....	327
Axis Aligned Bounding Boxes Tree.....	327
How to Divide the Box .....	329
Oriented Box Intersections .....	330
Triangle Intersection .....	332

Using ColDet with DirectX 8.0 .....	332
Collision Reaction .....	334
3-D Object Group Processing .....	336
Quiz.....	336
Additional Resources .....	341
<b>Part IV: Appendixes .....</b>	<b>343</b>
<b>Appendix A: Windows Game Programming Foundation.....</b>	<b>345</b>
How to Look through a Window .....	346
How Windows 95/98/ME/NT/2000	
Interacts with Your Game .....	346
The Components of a Window .....	347
A Window Skeleton .....	347
Step 1: Define a Window Class .....	351
Windows Data Types .....	354
Step 2: Register the Window Class .....	354
Step 3: Creating a Window of that Class.....	354
Step 4: Display the Window .....	357
Step 5: Create the Message Loop .....	358
The Window Procedure .....	361
A Window Skeleton Optimized for Games .....	361
Windows Resources .....	366
Additional Resources .....	371
<b>Appendix B: C++ Primer .....</b>	<b>373</b>
What's Object-Oriented Programming? .....	374
Abstraction.....	374
Classes.....	377
Encapsulation .....	378
Declaring a Class .....	379
Constructor.....	384
Destructor .....	384
This Pointer .....	385
Class Hierarchies and Inheritance .....	386
Inheriting Code.....	387
Inheriting an Interface .....	390
Virtual Functions.....	391
Polymorphism .....	392
Inline Functions .....	393
C++ Enhancements to C.....	394
Default Function Arguments .....	395
Placement of Variable Declarations.....	395
Const Variable.....	396
Enumeration .....	397
Function Overloading and Operator Overloading .....	397
Function Overloading.....	397
Operator Overloading .....	399
Additional Resources.....	400
<b>Appendix C: The Common Files Framework .....</b>	<b>401</b>
Create() .....	407
Step 1: Create the Direct3D Object with Direct3DCreate8().....	409
Step 2: Search for the Proper Device Driver with the Help of BuildDeviceList() .....	410
Step 1 in BuildDeviceList() .....	419
Step 2 in BuildDeviceList() .....	427
Step 3: Create a Window with CreateWindow() .....	428
Step 4: Initialize the Geometry Data of your Game with OneTimeScene Init() .....	428
Step 5: Initialize the 3-D Environment with Initialize3DEnvironment() .....	428
Step 1 in Initialize3DEnvironment() .....	433
Step 2 in Initialize3DEnvironment(): CreateDevice() .....	433
Step 3 in Initialize3DEnvironment(): SetWindowPos() .....	435
Step 4 in Initialize3DEnvironment(): GetDeviceCaps() .....	435
Step 5 in Initialize3DEnvironment(): GetDesc() .....	437
Step 6 in Initialize3DEnvironment(): D3DUtil_SetDeviceCursor() .....	438

Step 7 in Initialize3DEnvironment():	
Initialize the Application's Device	
Objects .....	438
Step 8 in Initialize3DEnvironment() .....	439
Step 6: Starting the Timer with	
DXUtil_Timer() .....	440
Run() .....	444
Step 1 in Render3DEnvironment():	
TestCooperativeLevel() and	
Resize3DEnvironment() .....	448
Step 2 in Render3DEnvironment():	
FrameMove() .....	451
Step 3 in Render3DEnvironment():	
Render() .....	451
Step 4 in Render3DEnvironment():	
Fill the Frame Count String .....	452
Step 5 in Render3DEnvironment():	
Present() .....	453
MsgProc() .....	455
<b>Appendix D: Mathematics</b>	
<b>Primer .....</b>	<b>463</b>
Points in 3-D .....	464
Vectors.....	467
Bound Vector.....	467
Free Vector.....	468
Vector Addition: $\underline{U} + \underline{V}$ .....	469
Vector Subtraction: $\underline{U} - \underline{V}$ .....	471
Vector Multiplication .....	472
Scalar Product.....	472
Dot Product.....	473
Cross Product.....	476
Unit Vector .....	477
Matrices.....	478
Multiplication of a Matrix with	
a Vector .....	480
Matrix Addition and Subtraction .....	480
Matrix Multiplication.....	481
Translation Matrix.....	481
Scaling Matrix.....	481
Rotation Matrices .....	482
Rotation about the y-axis .....	482
Rotation about the x-axis.....	482
Rotation about the z-axis .....	483
<b>Appendix E: Game Programming Resources .....</b>	<b>485</b>
General .....	486
DirectX Graphics .....	486
FAQ .....	487
<b>Index.....</b>	<b>489</b>

## LETTER FROM THE SERIES EDITOR

Dear Reader,

The 3D API wars on the PC are over. And no matter how you feel, Direct 3D is the victor on the PC platform. Amazingly enough, it sure didn't have to do with an over-abundance of clear documentation about Direct 3D! In fact, after years of the Direct 3D API being available, only one or two books are of any merit on the subject. With this in mind the Author of *Beginning Direct 3D Game Programming*, Mr. Wolfgang Engel, set out to write a beginner's book on Direct 3D that also covered Direct X and General Game Programming theory. I can without hesitation state that he has succeeded, and succeeded where others have failed.

This text is fantastic; it has a pace that is both challenging and cutting-edge, but not intimidating. You will find yourself learning very complex ideas very easily. Moreover, this book is one of the most graphically annotated books on Direct 3D, so you won't be left wondering what something is suppose to look like!

Additionally, although this book is for beginners it doesn't mean that the material is basic. In fact, as the chapters progress you will cover advance concepts such as multitexturing, lighting, TnL, 3-D file formats, and more!

A handwritten signature in black ink that reads "André LaMothe". The signature is fluid and cursive, with "André" on the first line and "LaMothe" on the second line, ending with a short horizontal line.

André LaMothe

March 2001

## INTRODUCTION

When I finished my first degree in law back in 1993, I was very proud and a little bit exhausted from the long learning period. So I decided to relax by playing a new game called Comanche by NovaLogic.

I started the night of January 11 and ended up about three days later with only a few hours of sleep. With the new experience in my head, I decided to start computer game programming. My target was to program a terrain engine like Comanche.

My then-girlfriend—now my wife—looked a little bit confused when a young, recently finished lawyer told her that he's going to be a game programmer.

About two years later, after becoming a member of the Gamedev Forum on Compuserve and reading a few books on game programming by André La Mothe and a good article by Peter Freese on height-mapping engines, I got my own engine up and running under OS/2. I wrote a few articles on OpenGL and OS/2 game programming in German journals, coauthored a German book, and started with the advent of the Game SDK (software development kit) on Windows game programming.

In 1997 I wrote my first online tutorials on DirectX programming on my own Web site. After communicating with John Munsch and the other administrators of [www.gamedev.net](http://www.gamedev.net), I decided to make my tutorials accessible through this Web site also. In the summer of 1998, as a Beta tester of the DirectX 6.0 SDK, I decided to write the first tutorial on the Direct3D Immediate Mode Framework. At that time I used [www.netit.net](http://www.netit.net) as the URL of my Web site. There was a mailing list with a lot of interested people, and I got a lot of e-mails with positive feedback.

It started to be real fun. In 1999 I fired up my new Web site at [www.direct3d.net](http://www.direct3d.net), which is now also accessible through [www.directxgraphics.net](http://www.directxgraphics.net), with the sole purpose of providing understandable and instructive tutorials on Direct3D programming.

This is also the target of the book that lies in front of you; it should help you to understand and learn DirectX Graphics programming.

If you have any questions, don't hesitate to e-mail me. A lot of things are implemented with the questions of readers in mind.

Mainz, Germany, December 2000

Wolf ([wolf@direct3d.net](mailto:wolf@direct3d.net))

## WHAT YOU'RE GOING TO LEARN

This book covers all of the elements necessary to create a Windows 95/98/ME/NT/2000 or short Windows-based Direct3D/DirectX Graphics game for the PC:

- 3-D graphics and algorithms
- Game programming techniques and data structures

- Using 3-D files to construct game worlds
- Programming your own character engine with a character animation system
- DirectX Graphics programming

And more...

## WHAT YOU NEED TO KNOW

This book assumes that you can program in C with a dash of C++. I will use the less-esoteric features of C++, the way that the Microsoft guys who programmed the Direct3D Immediate Mode samples in the DirectX SDK did. In case you need a refresher, there's a decent C++ primer in the appendix, so check it out.

You aren't taking a stab at graphics/game programming to learn the math. If you can add, subtract, multiply, divide, and maybe square a number, you will be able to understand 90 percent of the math and what's going on in this book. There's a math primer provided in the appendix to be absolutely sure that you won't miss anything.

## HOW THIS BOOK IS ORGANIZED

This book consists of four parts. The first part will show you the essentials of Direct3D game programming. It deals with the programming conventions, basic algorithms, texture-mapping basics, 3-D math, the transformation pipeline, lighting, and using depth buffers. It also provides a small DirectInput primer on using the keyboard interface.

In the second part, you will learn how to use the transformation and lighting pipeline to map textures on objects with different effects. All of the buzzwords, such as bump mapping, environment mapping, and procedural textures, are explained, and the effects are shown in sample programs.

In the third part of this book, you'll deal with file formats and how to integrate them into your game engine. The file formats used are the greatly enhanced .X file format, introduced with the DirectX 8.0 SDK, and the MD3 file format used in most of the games driven by the Quake III engine.

The fourth part contains appendixes, which should be useful if you decide to refresh your Windows programming, C++, or math skills. The Direct3D framework, which is used throughout this book, is explained here in detail.

## USING THE CD-ROM

The companion CD includes all the code from this book and the Microsoft DirectX 8.0 SDK, which you will need to install in order to compile and run the example programs discussed throughout the book (see the installation instructions in the following section). The DirectX SDK contains the run-time files, headers, and libraries you need to compile the book examples and provides a lot of example programs for every component of DirectX.

The book example code is located in directories named after the chapters of the book. In every example directory, you'll find the provided graphics and the source files. There's a `readme.txt` file, which provides you with additional information on compiling the application.

You'll need a system that fulfills the requirements of DirectX to run the example programs. You should have Microsoft Windows 98/ME/2000 and a Pentium II or higher card with a 3-D accelerator. To get a useful development system, you should use at least 128MB RAM and a big hard disk (greater than 10 GB) to store all of the files you'll produce in your development cycle. A monitor with a resolution of at least 1,024 × 768 and a connection to the Internet to browse the news on game development may also be useful.

## INSTALLING THE DIRECTX SDK

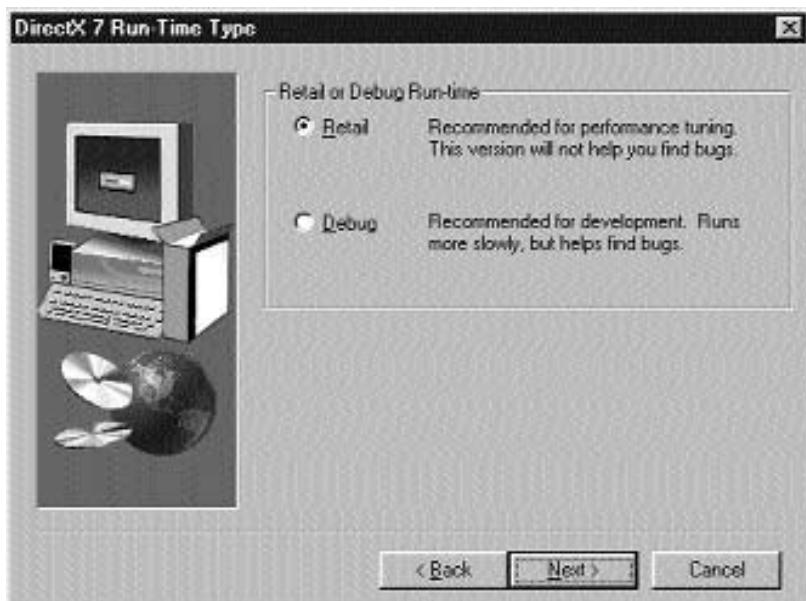
To install DirectX, you have to find the folder labeled DirectX on the CD that comes with this book. Basically, you can run the Setup program, and it will load the DirectX run time and the SDK into a directory—usually called `mssdk\`—on your hard drive. You have to choose among Complete, Custom, and drivers-only installations.



**Figure I.1:** Installation dialog box

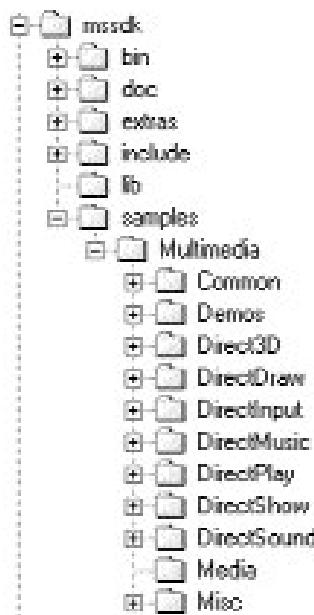
If you choose the Complete installation, everything is loaded onto your hard drive. The installation procedure checks the installed DirectX drivers, and if they are outdated, it will install the ones that are delivered with the CD-ROM. If you choose the Custom installation, the program will give you the choice to install only parts of the SDK. For example, installing the Visual Basic samples wouldn't make any sense if you develop C++ games only.

You will also have to choose between the Retail and Debug builds. The Retail builds are stripped of lots of debug checks and are also compiled to run faster. If you just plan to play DirectX games and not to write your own, the Retail builds will be the best choice. The Debug builds help coders to get their DirectX applications up and running. They help you in trapping down bugs by giving you debug messages. The trade-off, however, is that they run slower than the Retail builds.



**Figure 1.2:** Retail/Debug dialog box

After you finish installing DirectX, take a look around all of the directories and get to know the location of the libraries, include files, help, and samples. You need these locations to set up the Visual C++ compiler.



**Figure 1.3:** Directory structure

You will find the DirectX Graphics programming files in

C:\MSSDK\SAMPLES\MULTIMEDIA\Direct3D

All of the materials and concepts are compatible with all future versions of DirectX, so keep an eye on updates at [msdn.microsoft.com/directx](http://msdn.microsoft.com/directx).

The other piece of software you need to install is the newest drivers for your video card. You can pick them up on the Web site of your video card manufacturer. The newest drivers often offer speed and stability improvements and sometimes new features. Take a look at the always-provided readme file to get the proper installation instructions.

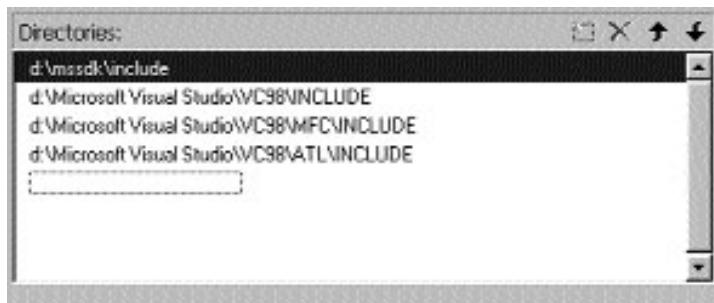
## SETTING UP VISUAL C++ 6.0

After you have DirectX installed, you need to consider the rest of your software. The most important piece of software is your IDE (integrated development environment), or short compiler. I suggest that you use the Microsoft Visual C/C++ 6.x or better compiler. This compiler generates really good Windows code, and you can get it for about \$99 from your local software store. This is the path of least resistance.

The new DirectX 8.0 SDK installation routine sets all of the proper paths for you. In case anything goes wrong, you should be able to check the right paths. Therefore, select Options from the Tools menu and choose the Directories tab. You should see something like the following dialog box:

**NOTE**

Intel makes a package called VTune, which might be integrated in the Visual C/C++ environment. It provides high optimizing compilers with profiling tools, which are faster than the standard Visual C/C++ compilers. Try to use them later in your development cycle to optimize your game before it's shipped.



**Figure 1.4:** Header files path

The main header directory should be D:\mssdk\include, where D is the hard drive and mssdk is the directory where you might have installed the DirectX SDK.

If you choose the lib paths in the drop-down menu, you should see something like the following:



**Figure 1.5:** Lib files path

The DirectX library files are located in the lib directory of the DirectX SDK installation.

These directories should always be at the top of the list, as shown in the Directories dialog box in Figures 4 and 5, because the compiler will search for these files beginning at the top of the list. You might see something like the following message if you haven't configured the include path properly:

```
d:\book\source\part 1\chapter6\animated objects\objects.cpp(68) : error C2146: syntax  
error : missing ';' before 'g_Keyboard_pDI'
```

Even if the right path to the header and library files is provided, you might have to feed the names of these files to the linker of your development environment. The proper path to these object/library modules should be listed in your Link dialog box. To reach this dialog box, select Project/Settings and then the Link tab. In the General category there is an entry field called Object/library modules. It holds all of the library files, which should be linked with the application you're currently developing. It should look like this:



**Figure I.6:** Linker path

In this entry field, you will need to name at least the following:

d3dx8.lib  
d3dxof.lib  
d3d8.lib  
winmm.lib  
dxguid.lib

If you missed a file, an error message appears, which might look like this:

```
d3dapp.obj : error LNK2001: unresolved external symbol _Direct3DCreate8@4
```

Here, the d3d8.lib is missing. The unresolved external symbols are part of the Component Object Model (COM) interfaces of Direct3D (I will explain COM later).

You should also check another include path, the one that holds the path to the directories of the common include files for the C/C++ compiler. Under Project Settings/C/C++, there's an entry in the drop-down menu called Preprocessor. You might choose as additional Include-Directories the following path:

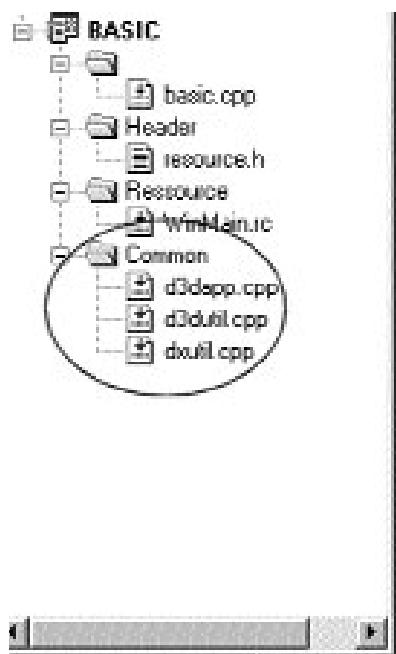
```
...\\common\\include
```

In case something goes wrong here, an error message indicates that the  
d3dframe.h

include file can't be found by the compiler in this case. This is just another show stopper; normally you include the Common files from the

```
...\\common\\source
```

in every project.



**Figure I.7: Files in IDE**

If you drag-and-drop the directory to another place, the paths to these files might not be correct anymore. So you have to add the Common files with Project->Add Files to Project->Files.

Now let's compile our first project:

- Fire up your Visual C++ configured development environment.
- Click on Open Workspace.
- Choose basics.dsp in the book\chapter4 directory.
- Check whether you've configured the paths to the directories as described above.
- Choose Build/basics.exe build.
- When everything works fine, choose run basics.exe.

That's it. If something went wrong, try to reread the previous passages on installing the DirectX SDK and Visual C++ and the provided documentation.

## ADDITIONAL RESOURCES

You should visit the Microsoft MSDN site for DirectX at [msdn.microsoft.com/directx](http://msdn.microsoft.com/directx) at regular intervals and the mailing list at [discuss.microsoft.com/SCRIPTS/WA-MSD.EXE?S1=DIRECTXDEV](http://discuss.microsoft.com/SCRIPTS/WA-MSD.EXE?S1=DIRECTXDEV). Daily news about the game developer community can be found at [www.gamedev.net](http://www.gamedev.net) or [www.flipcode.com](http://www.flipcode.com). I'll provide additional information on [www.direct3d.net](http://www.direct3d.net).

## QUIZ

Q: What's the difference between the Retail and Debug builds of the DirectX run times?

A: The Retail builds are stripped of lots of debug checks and are also compiled to run faster. The Debug builds help in trapping down bugs by giving you debug messages.

Q: How do you configure the linker environment of your IDE?

A: Project/Settings and then under the Link tab.

Q: What is the most important resource that you should check often?

A: [msdn.microsoft.com/directx](http://msdn.microsoft.com/directx)

# **PART 1**

**DIRECTX  
GRAPHICS:  
DON'T HURT ME**

We'll deal with the most basic questions on graphics programming with Direct3D in this part of the book. You will learn

- The essentials of programming with Direct3D
- The meaning of HAL
- How to use the COM binding to DX 8
- Coding conventions
- The basics of graphics programming, including orientation, faces, normals, and Gouraud shading
- Direct3D essentials
- Texture-mapping basics
- The ins and outs of the third dimension, especially 3-D math and the transformation pipeline
- The way Direct3D scenes are lit
- Scene management with depth buffering
- DirectInput essentials

# **CHAPTER I**

# **HISTORY OF DIRECT3D/ DIRECTX GRAPHICS**

**B**efore Windows, DOS was the most popular operating system for the PC. Games were programmed in DOS exclusively for many years. Game developers resisted developing for Windows because of its unacceptable graphics and audio performance at the time.

The direct access to hardware that DOS afforded came with its own complications, however. DOS games had to support the full range of video and audio hardware. This forced developers to write complex code to support dozens of different configurations just to provide consistent graphics and audio across all PCs.

With the advent of DirectX in 1995, Microsoft provided within Windows the performance previously available only through DOS, without the complexity of supporting each vendor's particular hardware solutions. Since that time, every hardware vendor delivers its product with Windows drivers.

Direct3D, part of DirectX, appeared in 1996 in DirectX 2.0. Direct3D is designed to give access to the advanced graphics capabilities of 3-D hardware accelerators, while promoting device independence and hardware abstraction by providing a common interface to the programmer. Code properly written for Direct3D will work on Direct3D devices now and in the future.

Let's dive a little bit deeper into history: In the early '90s, a lot of PC 3-D engines were built in Great Britain. There were the well-known Renderware ([www.renderware.com](http://www.renderware.com)) and the BRender from Argonaut ([www.argonaut.com](http://www.argonaut.com)), which was ported in 1994 to OS/2 and a small British company called RenderMorphics. RenderMorphics was founded in 1993 by Servan Keondjian, Kate Seekings, and Doug Rabson with their savings and produced a product called Reality Lab. Keondjian played piano in a band at night and programmed his 3-D engine by day. Seekings subsequently upped her credentials with a quick master's degree in computer graphics at Middlesex University. It's interesting to note that her 3-D rendering library, developed with input from Keondjian and Rabson, was submitted as a school project and was flunked for not following the assigned specs closely enough.

At the first trade show they attended (SIGGRAPH 94), they were spotted by Microsoft, and RenderMorphics was acquired in February 1995.

After the acquisition of RenderMorphics, Microsoft integrated Reality Lab into its DirectX family of APIs (application programming interfaces). The Immediate Mode component of Reality Lab absorbed the standard 3-D Windows API of the time, 3-D-DDI, which was created by Michael Abrash, later one of the creators of the Quake I engine at id Software.

Until the advent of DirectX 8.0, Direct3D consisted of two distinct APIs: Retained Mode and Immediate Mode. At that time, the Immediate Mode API was difficult to use, but it was a flexible, low-level API that ran as efficiently as possible. Retained Mode was built on top of Immediate Mode and provided additional services, such as frame hierarchy and animation. Retained Mode was easier to learn and use than Immediate Mode, but programmers wanted the added performance and flexibility that Immediate Mode provided. Development of the Retained Mode API has been frozen with the release of DirectX 6.0.

The major changes between the Direct3D Immediate Mode version 6.0 and version 7.0 were the support of hardware accelerated transformation and lighting, and the reorganization of the lights, materials, and viewport objects, which from now on are set directly by calling the methods of IDirect3DDevice7 and the drop of a special interface to access textures. The IDirect3DDrawSurface7 interface also provided an easier way to manage the textures.

With the advent of the DirectX 8.0 SDK came the biggest improvements in the history of Direct3D. Direct3D got a fundamentally new architecture with version 8.0, which should be even more stable in future versions. The initialization, allocation, and management of data were simplified by the integration of DirectDraw and Direct3D into one interface, called DirectX Graphics, which led to a smaller memory footprint and a simpler programming model.

With DirectX Graphics, you can now use vertex and pixel shaders with their own processing language instead of the traditionally used transformation and lighting, or fixed function, pipeline. These shaders are more flexible than the older approach and are a lot more powerful, when the graphics hardware supports them. The shader language looks similar to x86 Assembler code. At the time of this writing, no widely available hardware supports vertex or pixel shaders, and only vertex shaders may be emulated. But that will change in the upcoming months, and they might get the killer features of Direct3D.

The new ability of DirectX 8 and of upcoming hardware to render a scene more than once—called *multisampling rendering*—opens up a variety of effects, such as full-scene anti-aliasing, motion blur, and depth of field.

Another new feature of DirectX 8.0 is the ability to use hardware sprites for your particle system to generate sparks, explosions, rain, snow, and so on. So-called point sprites are supported with their own programming interface to help you in doing this.

Ever heard of voxels? Voxels are three-dimensional entities that describe a 3-D volume (not the 2-D voxel engine used in Comanche-like games). With 3-D volumetric textures, something similar to voxels can be done. Exact per-pixel lighting and atmospheric effects could be applied with these textures to your application.

With Direct3D 7.0, a new utility library called Direct3DX appeared. It provides helper functionality for enumerating device configurations, setting up a device, running full-screen or windowed mode uniformly, running resizing operations, calculating vector and matrix operations, and simplifying image file loading and texture creation. In addition, it provides functions for drawing simple shapes, sprites, and cube maps.

## CAUTION

**The first incarnation of Direct3DX in the DirectX 7.0 SDK was a little bit buggy. There are a few known bugs, especially D3DXCreateContext() and D3DXMatrixLookAtLH(). They were fixed in the DirectX 7.0a SDK, but that introduced a new bug, D3DXMatrixLookAtRH(). The bugs were removed in DirectX 8.0, but you should be careful with the older versions. Read more on Microsoft's mailing list by searching the online archive at discuss.microsoft.com/SCRIPTS/WA-MSD. EXE?SI=DIRECTXDEV.**

Direct3DX has been greatly enhanced in DirectX 8.0. It now supports additionally skinned meshes, multi-resolution level-of-detail (LOD) geometry, and higher-order surface data for .X files. It includes a skinning library to work with meshes and functions to assemble vertex and pixel shaders. The D3DX image file loader functions support BMP, TGA, PNG, JPG, DIB, PPM, and DDS files. It also provides helper methods to port OpenGL applications to DirectX Graphics.

The DirectX 8.0 SDK provides a programming framework that is similar to the framework used in the DirectX 6.0 and 7.0 SDKs. The new SDK doesn't call it *framework* anymore, instead it talks about Common files. These files give you the direct access you need and encapsulate the details of setting up Direct3D, great for your learning curve. You can concentrate on the essential things while still being able to see everything on the lowest level. This framework gives you a common ground on which you can implement your individual features. As a beginner, you can avoid a lot of basic mistakes with the fast and very well tested framework code, allowing you to concentrate your energy on learning. Intermediate and professional programmers might use the framework as a good testing platform, or perhaps professional programmers will write their own framework that suits their needs better by looking at the DirectX Graphics Common files source code.

### NOTE

Its disadvantage is the lack of source code. Every 3-D engine programmer wants to be able to control everything down to the metal. The programmer won't like another level of abstraction between him and his hardware, which is not transparent.

### NOTE

A detailed description of these Common files is provided in Appendix C. For example, ATI at [www.ati.com/na/pages/resource\\_centre/dev\\_rel/devrel.html](http://www.ati.com/na/pages/resource_centre/dev_rel/devrel.html) uses a slightly modified version of the framework used in the DirectX 7.0 SDK. NVIDIA at [www.nvidia.com](http://www.nvidia.com) has built a completely new framework, and Intel at [www.intel.com](http://www.intel.com) has built its own framework, which uses the Visual C/C++ 6.0 project wizard to give you an interactive way to build a code template that is sufficient for your needs. And there are a lot more game companies that have built their own production libraries without releasing the source.

# **CHAPTER 2**

## **OVERVIEW OF DIRECTX GRAPHICS/ HAL/COM**

Like all of the DirectX APIs, DirectX Graphics was designed to provide maximum speed on different hardware, with a common interface and backward compatibility in mind. This leads to the following demands:

- The API needs consistency to be viable.
- If a feature is not supported by hardware, there has to be a fallback mechanism.
- Interface innovation must be possible with the lowest possible learning curve for programmers.
- All games developed in early versions of DirectX must be guaranteed to run in future versions of DirectX.

The answers to these demands are the HAL (hardware abstraction layer) or pluggable software device and the COM (component object model), which you've probably heard before from other Microsoft APIs.

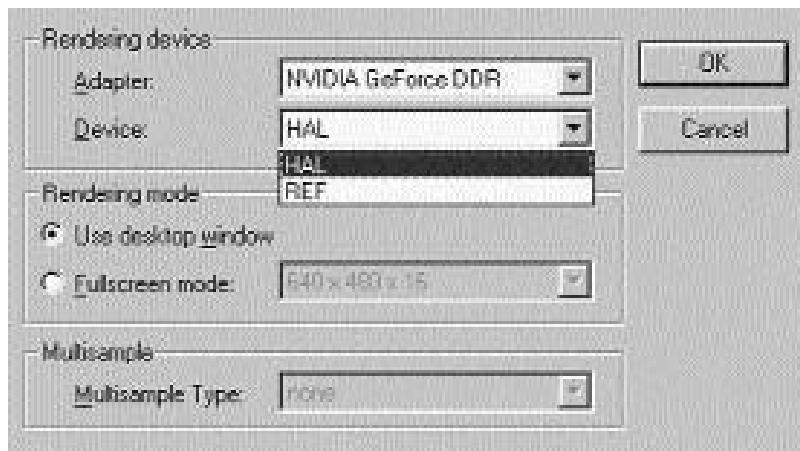
Let's face the problem from a practical viewpoint. After reading this book, you program that great game, which will make you happy and rich. It runs very well on your GeForce2-driven and Voodoo 5-driven computers. To show it to your girl/boyfriend, you'd like to install it on her/his portable computer, which is, by the way, one of those much-loved computers with its own name. After installing the newest DirectX version and your masterpiece, you finally realize that this portable has 3-D hardware, which is suitable for every IBM/Microsoft/HP/<insert the name of your company of choice> manager. After clicking on the icon of your best work, the best possible visual experience that this computer can provide should appear. The story shouldn't end with a loud cry of <insert the name of your computer of choice> by your friend, who thinks that the display of her/his small portable is broken.

The DirectX team knows this situation, so they provided a way to emulate features that are not supported by the dedicated hardware. Until DirectX 7.0, this was called the HEL (hardware emulation layer). Since DirectX 8.0, it's referred to as a pluggable software device. Whereas the DirectX 7.0 SDK provided a HEL, called the RGB device, the new pluggable software device would typically be developed by the software manufacturer, if they wanted to target non-hardware accelerated machines. The HAL is your friend within the first second you use Direct3D, because this piece of software, which is provided by the manufacturer of the graphics card, will allow you access to all of the hardware's features. Now, what about features that are not supported by hardware?

There are two possibilities: Your program checks the capabilities of your HAL device and switches the missing features out, or you switch to the pluggable software device with the Change Device dialog box.

Yes, you've seen HAL before, if you've ever played a game driven by DirectX. There's always a drop-down box somewhere in the game that gives you the choice to pick a device or driver. In DirectX 7.0, depending on the graphics hardware, a HAL, TnLHAL (transformation and lighting HAL), RGB, or reference rasterizer driver could be chosen from such a dialog box. Since DirectX 8.0, depending on hardware, a HAL, pluggable software device, or reference rasterizer might be chosen by the user or by the game automatically.

In the DirectX Graphics samples in the DirectX 8.0 SDK, you might choose a device for yourself by using the Change Device dialog box.

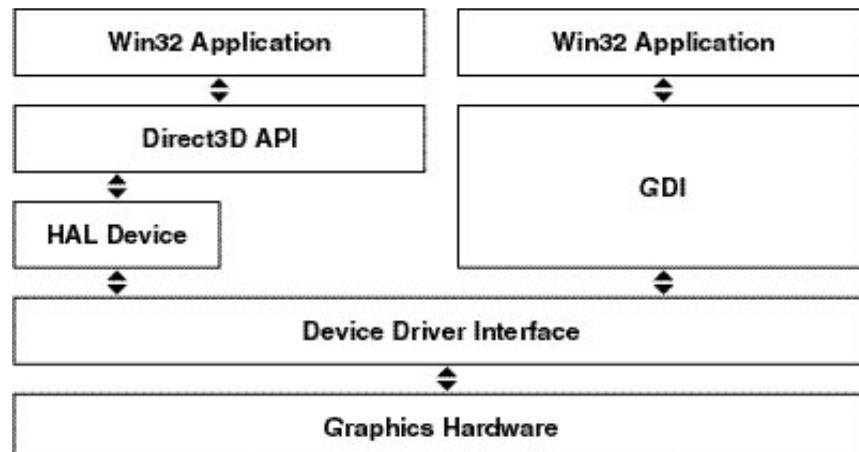


**Figure 2.1:** The Change Device dialog box

This computer provides two devices. There's no pluggable software device at the moment, but there is one HAL driver and one reference rasterizer driver.

## DIRECT3D HAL

This overview shows the relationships among the Direct3D API, HAL, and the GDI (Graphics Device Interface) API:



**Figure 2.2:**  
Direct3D/HAL/GDI

As you see, HAL is used by Direct3D to access the graphics hardware through the DDI (Device Driver Interface). HAL is the primary device type, which supports hardware-accelerated rasterization and both hardware and software vertex processing. If the display adapter of your computer supports Direct3D, HAL will exist on your computer.

The behavior of the pluggable software device and the reference device has to be identical to that of the HAL device. Application code authored to work with the HAL device should work with the software or reference devices without modifications. You have to check the capabilities for each device of course. The reference rasterizer supports all the Direct3D 8 features, while a pluggable software device could possibly not even support texturing.

One of the most exciting HAL applications is the “tree demo” of NVIDIA. You can find it with the source code at [www.nvidia.com](http://www.nvidia.com).

### NOTE

In the old days, you had to distinguish between HAL devices or drivers that support transformation and lighting on their own and devices that are used to get transformed and lit triangles. Formerly they were called HAL and transformation and lighting HAL—TnLHAL for short. With the old HAL device, Direct3D performed the transformation and lighting of the 3-D world and the 3-D models on its own and supplied lit triangles, already transformed to screen space, to HAL and the 3-D card. Nowadays, all tasks are handled by one HAL. The end user won't see two different HALs anymore.

### TIP

A note on the use of hardware transformation and lighting in games: Oftentimes the bottleneck of 3-D games is the transformation, lighting, and clipping steps, as these are very math intensive (as you will see in the following chapters) and bog down the processor. A graphic card with hardware transformation and lighting is an amazing advantage, because it off-loads these tasks. This hardware T&L (transformation and light) unit is often called a GPU (graphics processing unit). Most games today, like Quake III Arena, do their own lighting calculation but let the graphic API do the transformation. They benefit from a graphic card that supports hardware transformation and lighting. It's only a matter of time until these games also use the hardware lighting features. In the meantime, a combination of hardware and software lights will be used.



**Figure 2.3:** The tree demo of NVIDIA: gobs of textured, lit polygons rendered in real time

If there's no hardware accelerator in a user's machine, attempting to create a HAL device will fail.

## PLUGGABLE SOFTWARE DEVICES

If the user's computer hardware doesn't provide all of the 3-D operations needed at least to play the game, your application might emulate 3-D hardware by using software with a pluggable software device, formerly called an RGB device. This device doesn't support 256-color or 8-bit color modes provided by the abandoned Ramp driver. It takes advantage of any special instructions supported by the user's CPU to increase performance, including the AMD 3D-Now! instruction set on some AMD processors and the MMX/SIMD instruction set supported by many Intel processors.

Nowadays, the hardware emulation device has to be developed by the software/game manufacturer, whereas in former incarnations of the DirectX run times, it was provided by Microsoft as the RGB device. Software devices are loaded by the application and registered with the Direct3D object.

### NOTE

These software device drivers communicate to Direct3D/DirectX Graphics through an interface similar to the hardware DDI. The Direct3D DDK provides the documentation and headers for developing pluggable software devices.

## REFERENCE RASTERIZER

The reference rasterizer supports *all* Direct3D features. It should be used only for testing features that your card doesn't support. This device is optimized for accuracy, not for speed.

## CONTROLLING DEVICES

You can configure all these devices with the DirectX Properties dialog box, which can be found in Start/Settings/Control Panel/DirectX.

### TIP

Direct3D doesn't enumerate this device by default. The DirectX 8.0 SDK installer will set the **EnumReference** value in the **HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Direct3D\Drivers** registry key to a nonzero **DWORD** value.

Neither hardware, software, nor reference devices can render to 8-bit render-target surfaces.

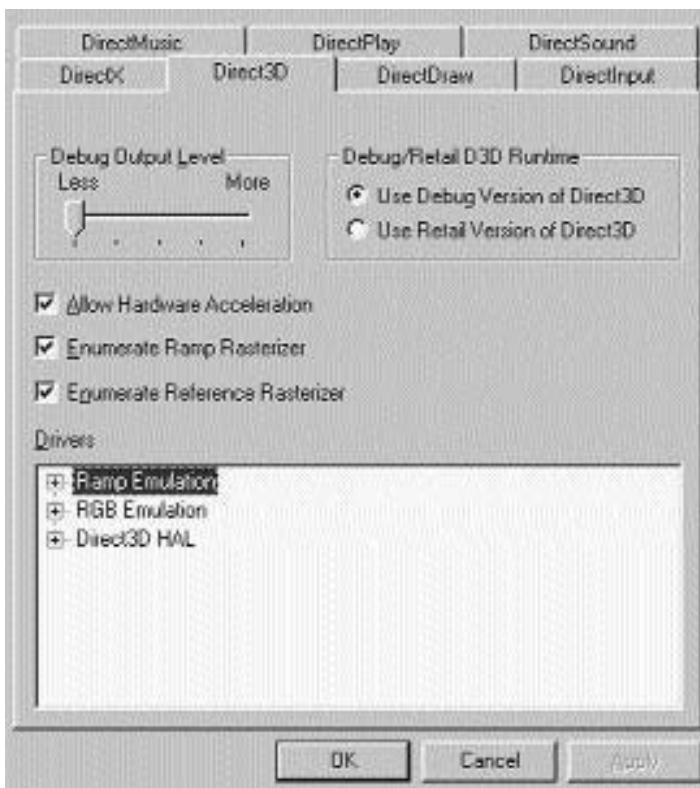


Figure 2.4: DirectX Properties dialog box

With this HAL/pluggable software driver/reference driver approach, Microsoft can guarantee a consistent API to fall back on if specialized 3-D hardware is absent. That's great isn't it? OK, what about the other two postulated demands from the beginning of this chapter: How does Direct3D handle version changes? Are they easy to learn, and are they handled so that they're transparent to the programmer? This is where the COM interface design of the DirectX SDK helps.

## COM

If you've used other Microsoft APIs, there's a good chance that you're an old COM freak who can skip over this section. All others have to invest a small learning effort and will get a good return on their investment by learning to use the COM interface, which has been used by DirectX since its inception.

In COM terms, a software component is simply a chunk of code that provides services through interfaces, and an interface is a group of related methods. A COM component is normally a .DLL file that can be accessed in a consistent and defined way. Period. Didn't sound like something someone could write books with more than 1,000 pages about, did it?

COM has a lot of advantages. I'd like to highlight only three, which I think are important for Direct3D programmers like us:

- COM interfaces can never change.
- COM is language-independent.
- You can only access a COM component's methods, never its data.

COM interfaces can never be modified in any way. Applications that use COM objects don't need to be recompiled whenever an interface changes, because COM can't provide a standard way to communicate with other objects when the interface of a published object could be changeable.

For example, the COM object A, produced by software company C, would be incompatible with COM object B produced by software company D when the interface of object B is changed and software company C is too slow in implementing the new interface.

When a COM object has to be changed, a completely new interface with a new name will be added to the DLL (dynamic-link library). So every COM DLL has to support the legacy interfaces since its release. Direct3D 8.0 or DirectX Graphics supports the following interfaces: IDirect3D, IDirect3D2, IDirect3D3, IDirect3D7, and IDirect3D8. Whereas IDirect3D was the first incarnation of the Direct3D interface in DirectX 2.0, IDirect3D2 was the interface used in DirectX 3.0, IDirect3D3 was the interface of DirectX 6.0, IDirect3D7 was used in DirectX 7.0, and IDirect3D8 is in DirectX 8.0. If a company produces a game that is compatible with Windows NT 4.0, which only supports the IDirect3D2 interface of the DirectX 3.0 SDK, the game company will have to use the IDirect3D2 interface for everything to work fine. It's implemented in every upcoming Direct3D DLL since the advent of DirectX 3.0 and will be implemented in any future versions. The aforementioned game will run on every platform that runs at least the DirectX 3.0 run times or later.

**NOTE**

**C++ and many other languages are source-based languages; therefore, they do not have a versioning strategy.**

**COM was designed to be a binary based programming model. It doesn't allow you to change an interface once you've designed it, but a COM object can support multiple interfaces.**

COM is language-independent. Whether your favorite language is Visual Basic, Pascal, or any other language, you can access COM objects with your favorite compiler package. Delphi users especially like this feature a lot, whereas Visual Basic gurus were provided with a fine implementation of their own with the arrival of the DirectX 7.0 SDK. Language independence matters when parts of the game—for example, the world editor—will be written in Delphi and other parts with the Visual C++ compiler, perhaps at different places in this world or on other planets with different time zones. What's the time on Mars now? COM can only be accessed via methods. There's no possibility of accessing data objects directly. This is a good object-oriented design. As you will see in a few pages, you can't call a method directly. Instead, you have to use double indirection through a virtual function table, called the *v-table*, or just VTBL. These v-tables are also the key to a language-independent interface. With these advantages, COM helps to get language-independent, guaranteed access to the legacy and current Direct3D interfaces. Now get access!

# **CHAPTER 3**

**C/C++ AND  
COM**

**PROGRAMMING  
RULES FOR  
DIRECT3D**

When you call a method in a COM component, you have to call the method by using a v-table. When you're deciding whether to develop your code in C or C++, you need to consider a few issues. I'll start with an example. Let's say you'd like to set a light in a scene of your 3-D world-class game. With C, you would use the following line of code:

```
hr = m_pd3dDevice->lpVtbl->SetLight(m_pd3dDevice, 0, &light);
```

You call the COM interface methods by passing the `this` pointer, called `lpdd`, as the first parameter of the method and by referencing the interface's method by using it as a pointer to the interface v-table, which is called `lpVtbl` here.

### NOTE

There are macros that help C programmers make their life easier. For example, on line 1014 of `ddraw.h`

```
#define IDirectDraw_SetDisplayMode(p, a) \
(p)->lpVtbl->GetDisplayMode(p, a)
```

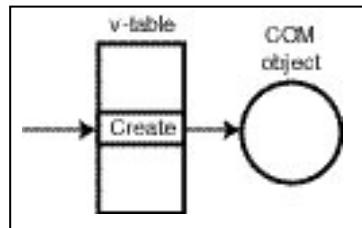


Figure 3.1: The v-table

With C++, there's one advantage: COM objects and C++ objects are binary-compatible in such a way that compilers handle COM interfaces and C++ abstract classes the same way.

So in C++, the `lpVtbl` pointer is implicitly dereferenced and the parameter is implicitly passed. Thus, the call seen above for C will look like this in C++ instead:

```
hr = m_pd3dDevice->SetLight(0, &light);
```

So in general, most DirectX calls in C++ look like this:  
`1pinterface->methodname`

### NOTE

To be more precise: the In-Memory layout in C++ of an instance of a class that inherits from a pure virtual base class (a class with only methods that are all virtual and equal to zero) has the same memory representation as a COM interface.

As you've seen in the Introduction, to compile a DirectX program, you have to include a number of import libraries:

```
d3dx8.lib  
d3dxof.lib  
d3d8.lib  
winmm.lib  
dxguid.lib
```

Most of these libraries are import libraries and provide the interface names to the linker of your development environment.

Now that you understand that COM objects are collections of interfaces, which are simply method pointers and, more specifically, v-tables, you need to see an example of how to work with COM. There are three things to be aware of:

- Direct3D run-time COM objects and DLLs must be registered to and loaded by Windows. The DirectX installer will do this for you.
- The previously mentioned libraries must be included in your Windows project so that the wrapper methods you call are linked in.
- The proper include files have to be included in your source file and in the include path entry forms of your IDE (for example, Visual C++) so the compiler can see header information, prototypes, and data types for DirectX Graphics.

Here's the data type for IDirect3D8 interface pointer:

```
LPDIRECT3D8 g_pD3D = NULL;
```

To create IDirect3D8 COM object and retrieve an interface pointer on it, all you need to do is use the Direct3DCreate8() method like this:

```
m_pD3D = Direct3DCreate8( D3D_SDK_VERSION );
```

The only parameter passed to Direct3DCreate8() should always be D3D\_SDK\_VERSION. This informs Direct3D that the correct header files are being used. This value is incremented whenever a header or other change would require applications to be rebuilt. If the version does not match, Direct3DCreate8() will fail.

The retrieved interface pointer gives you access to the interface of IDirect3D8. Now you might call a method in that interface:

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,  
                                    D3DCREATE_SOFTWARE_VERTEXPROCESSING,  
                                    &d3dpp, &g_pd3dDevice ) ) )
```

This code creates the device with the default adapter by using the D3DADAPTER\_DEFAULT flag. It indicates that you prefer a hardware device over a software device by specifying D3DDEVTYPE\_HAL for the DeviceType

parameter and uses D3DCREATE\_SOFTWARE\_VERTEXPROCESSING to tell the system to use software vertex processing (keep reading for the details).

That's it for COM. You will use COM throughout this book. If you're wondering about the strange and cryptic parameter names, they will be explained now....

## CODE STYLE

I use a simple variable tagging scheme that has its roots in the so-called Hungarian notation used by Microsoft for its own development projects. The name came from its inventor, Charles Simonyi, a now-legendary Microsoft programmer who happened to be Hungarian. It's helpful to supply variables in the right format to help others to read your code, but it could be confusing to people who haven't seen it before.

The following table shows the prefixes I use and the types they represent. You might come across other prefixes occasionally, but this table shows the common ones. Hungarian notation is just prefixing variables with their types.

Prefix	Type	Example
w	WORD	wSpeed
dw	DWORD	dwHitList
f	FLOAT	fSpeed
d	DOUBLE	dDirection
p	pointer	pArray
m_v	D3DXVECTOR/D3DXVECTOR3	m_vLight
l	LONG	lPitch
s	string	sQuestion
sz	string terminated by 0 byte	szQuestion
h	handle	hResult
I	COM Interface	IDirect3D
m_p	Member class pointer	m_pFloorVertices
m_f	member class float	m_fStartTimeKey
c	constant	cText
b	BOOL	bCheck

While the variable naming convention is to prefix the variables with their types, the naming convention for *methods* just clarifies readability and the purpose of the method. In all methods, the first letters of subnames are capitalized; an underscore is illegal.

```
HRESULT ConfirmDevice( DDCAPS* pddDriverCaps, D3DDEVICEDESC7* pd3dDeviceDesc );  
  
HRESULT CreateInputDevice( HWND hWnd,  
                           LPDIRECTINPUT7 pDI,  
                           LPDIRECTINPUTDEVICE2 pDidDevice,  
                           GUID guidDevice,  
                           const DIDATAFORMAT* pdidDataFormat,  
                           DWORD dwFlags );
```

As you can see, parameters for *functions* follow the same naming conventions that normal variables do. The parameter pddDriverCaps means a pointer on a DirectDraw device, which points to DriverCaps structure. The other parameter, pd3dDeviceDesc, means a pointer on a Direct3D device, which points to a device description structure.

*Types* and *constants* begin with an uppercase letter, but you're allowed to use underscores in the names. For example:

```
#define D3DPRESENT_BACK_BUFFERS_MAX 3L
```

All C++ *classes* must be prefixed by a capital *C*, and the first name of each subname of the class must be capitalized, too. Here's an example:

```
class CD3DMesh  
{  
public:  
};
```

CD3DMesh class is used to handle the loading of .X files. It's implemented in d3dfile.h of the Common files.

Let's try it. We'll take a look at a typical Direct3D application class:

```
class CMyD3DApplication : public CD3DApplication  
{  
    CD3DFont*          m_pFont;  
    CUSTOMVERTEX        m_QuadVertices[4];  
    LPDIRECT3DTEXTURE8 m_pCustomNormalMap;  
    LPDIRECT3DTEXTURE8 m_pFileBasedNormalMap;  
    D3DXVECTOR3         m_vLight;  
    BOOL               m_bUseFileBasedTexture;
```

```
BOOL           m_bShowNormalMap;
HRESULT CreateFileBasedNormalMap();
HRESULT CreateCustomNormalMap();
HRESULT ConfirmDevice( D3DCAPS8*, DWORD, D3DFORMAT );
LRESULT MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam );
protected:
    HRESULT OneTimeSceneInit();
    HRESULT InitDeviceObjects();
    HRESULT RestoreDeviceObjects();
    HRESULT InvalidateDeviceObjects();
    HRESULT DeleteDeviceObjects();
    HRESULT Render();
    HRESULT FrameMove();
    HRESULT FinalCleanup();
public:
    CMyD3DApplication();
};
```

There is a member class pointer on a font class at the beginning. There's another member in the class, which uses a custom vertex structure for a quad array of vertices. Two member class pointers are pointing to Direct3D texture objects. A vector light is stored in D3DXVECTOR3. If a file-based texture is used, `m_bUseFileBasedTexture` will indicate this. If the user likes to see the normal map, the switch `m_bShowNormalMap` has to be set to TRUE. The rest are methods, which will be used throughout the sample.

Please read the COM sample at the beginning of this chapter again. Got it? Yep.

## DEBUGGING DIRECTX

Debugging DirectX applications can be a challenging task. Here are a few tips to give you a starting point:

- Use the DXDiag utility from the DirectX SDK to report your bugs, but be sure that you know exactly what is on your system.
- The DirectX Control Panel allows developers to set the debug output level from 0 to 5. You can find it at Start/Settings/Control Panel/DirectX/Direct3D. On the same tab, it's possible to switch from the Retail to the Debug run-time versions, or vice versa.
- The D3DX library is a static library. To help debugging, there's a debug only dynamic library of D3DX. To use this, link with the d3dx8d.lib, which is an import lib corresponding to the D3DX8D.DLL.
- The Visual C++ GUI debugger can debug full-screen exclusive apps only when using a multi-monitor system or remote debugging.

With this in mind, it's usually wise to build the app in a windowed mode, like the one provided by the framework that is implemented in the Common files.

## RETURN CODES

A *return code* is the value returned when the method completes. It indicates the success or failure of a call and why the call fails. Checking the return codes is simplified by the macros SUCCEEDED() and FAILED() provided with the DirectX 8.0 SDK.

They take a HRESULT as an argument and return whether it indicates a success code or a failure code.

Don't try to check HRESULT against S\_OK; not every method returns S\_OK if it succeeds. A function might return a S\_FALSE to indicate that it did nothing because there was no need to do anything.

Because every DirectX method returns an HRESULT, you need to structure your code to check for and handle all potential errors. This work is frustrating at first, but it soon becomes second nature. Having a program that might be more resistant to crashing is a good reward for this work.

### NOTE

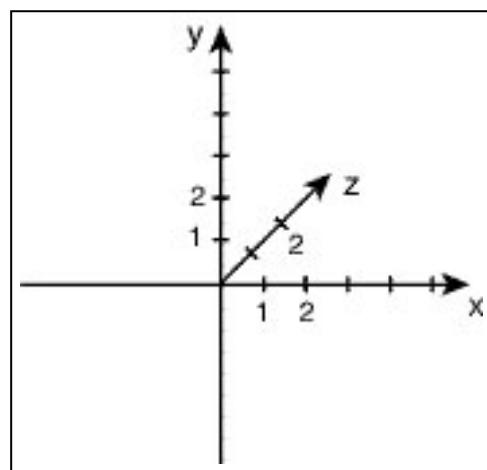
You might use CRT as a debug memory manager. It comes with your Visual C++ compiler package. You may find an extensive explanation at [msdn.microsoft.com](http://msdn.microsoft.com).

*This page intentionally left blank*

# **CHAPTER 4**

## **GEOMETRY/ SHADING/ TEXTURE- MAPPING Basics**

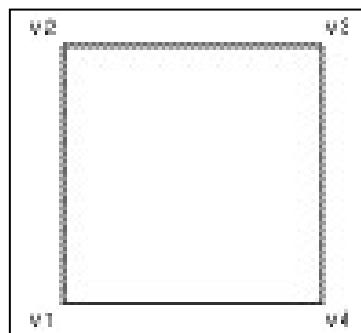
I'll have to give you a short geometry, shading, and texture primer before we take off. The Direct3D engine uses a left-handed coordinate system by default. Every positive axis (x, y, or z) is pointing away from the viewer. To make things a little bit clearer, here's a screen shot with the world coordinate system:



**Figure 4.1:** Direct3D coordinate system

The positive y is up, positive x is to the right, and the positive z-axis is into the screen, away from the user. Because the coordinate values that we are using are floating point, you don't lose any precision by using numerically small units.

Now on to vertices. Let's face a square:

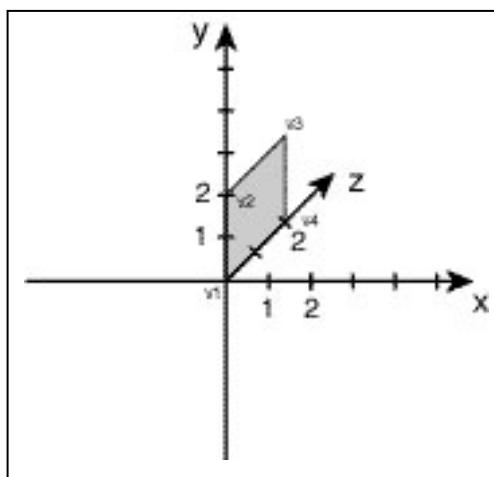


**Figure 4.2:** A square

### TIP

There's also a way to use a right-handed coordinate system, such as the one used by OpenGL. The Direct3DX utility library has built-in functions that could handle that task. This is useful for porting applications from OpenGL to Direct3D.

The square uses four vertices, v1–v4. A *vertex* defines the point of intersection of one or more lines, and these are straight lines as far as we are concerned. What's the point of all this business about coordinate systems and vertices? The point is that Direct3D needs a way to place these primitives in a scene. We need something to measure distance and space. OK, let's place our square in the coordinate system above:



**Figure 4.3:** A square in the coordinate system

To define a vertex in 3-D space, we need to specify three coordinate values: x, y, and z. The origin of our coordinate system is 0, 0, 0. The position of the vertices of the square could be described as

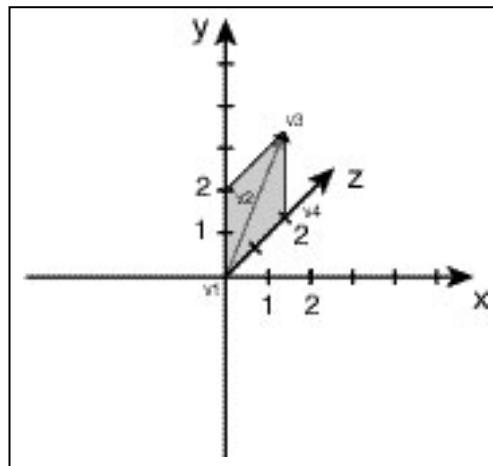
```
v1(0, 0, 0)  
v2(0, 2, 0)  
v3(0, 2, 2)  
v4(0, 0, 2)
```

And here we are: We can define the location of a vertex by specifying a vector from the origin to another point in 3-D space.

This is the so-called free vector used by game programmers. Its initial point is assumed to be the origin, and only the final point is described:

### NOTE

Vectors are one of the most important concepts in 3-D games. They are mathematical entities that describe a direction and a magnitude (which can be used for speed, for example). Vectors are usually denoted by a bold-faced letter of the alphabet, such as  $\mathbf{a}$ . So, we could say the vector  $\mathbf{v} = (1,2,3)$ . The first column is units in the x direction, the second column is units in the y direction, and the third column, units in z.



**Figure 4.4:** Free vector  
describing the third vertex of  
a square

Vectors are not only used to define the position of the vertices. They are also used to define a direction. For example, the vector  $vUp(0, 1, 0)$  defines the up direction.

As you might expect, Direct3D provides two default structures for defining vectors. You'll find the enhanced version D3DXVERTEX3 of Direct3DX—the utility library—in d3dx8math.h, and the default version D3DVECTOR in d3dtypes.h, which handles all the math routines you need when you use vectors. Vector  $v3$  would be defined as

```
D3DVECTOR v3( 0, 2, 2); // or ...
```

```
D3DXVECTOR3 v3(0, 2, 2); // used by the utility
library
```

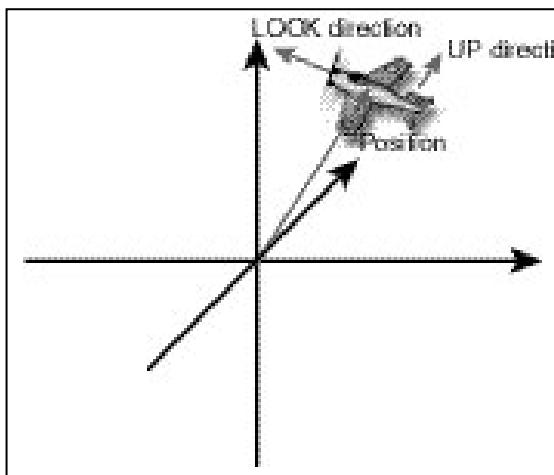
### TIP

Note that the actual length of a vector is not important when the vector is used to define a direction or a vertex.

## ORIENTATION

Measuring and organizing space in a virtual world is important for every game, but we also need a way to define the orientation of objects.

We might describe the orientation of an object by using at least three vectors: the LOOK vector, the UP vector, and the POSITION vector.

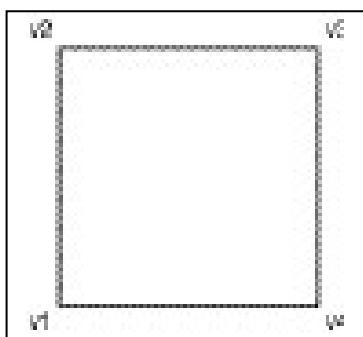


**Figure 4.5:** Orientation of a plane

One vector is used to define the position of the object (POSITION vector); another vector is used to define the way the object is pointing (LOOK vector). The third vector is needed if the object is rotated around the LOOK vector (UP vector). It tells you when the object is considered up or down. To work with these vectors, we'll store them later in a matrix.

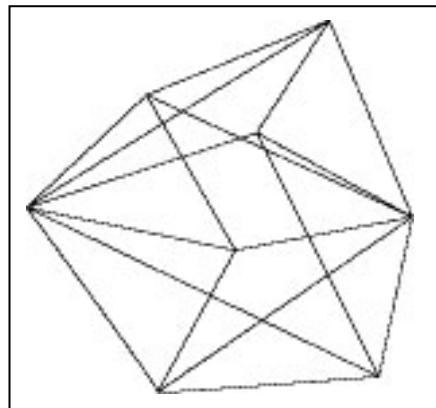
## FACES

Any 3-D object consists of faces.



**Figure 4.6:** Face of a square

To create a smooth-looking object, you need quite a lot of flat faces:



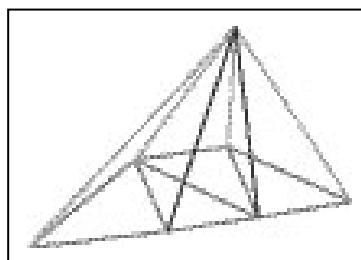
**Figure 4.7:** Cube constructed of triangles

Every model and every 3-D world is broken down into points, lines, and triangles. These are the only primitives Direct3D is able to render. If you design your shapes using some very special polygons to reduce the number of faces, Direct3D will divide every face into triangles. So if you'd like to get optimal performance, every object should be divisible into triangles as easily as possible.

A more complex example is the following object, which consists of nine faces from 10 triangles.

### NOTE

Direct3D transforms the triangles and lights them to build a scene. Transformation is the process of positioning and orienting the parts of your 3-D world and aligning them to your computer screen. Lighting, as you might suppose, is the attempt to simulate the affectation of light in a scene.

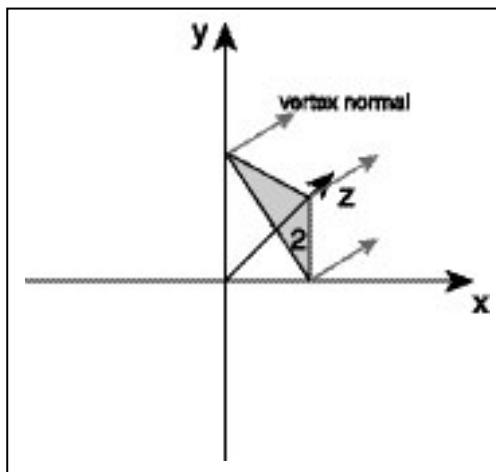


**Figure 4.8:** Faces of a more complex object

One final point you need to be aware of is that you have to specify the sets of vertices that define a face in clockwise order, as in Figure 4.6. The rendering engine will draw only the front faces of the objects, which are by default faces where the vertices are grouped in clockwise order. This process is known as *culling*. By culling the back surface of every face, Direct3D avoids drawing faces that aren't visible. This process is called *backface culling*.

## NORMALS

Normals are vectors that define the direction a face is pointing, or the visible side of a face:



**Figure 4.9:** Vertex normals of a triangle

A vertex normal is often perpendicular to the face.

## NORMALS AND GOURAUD SHADING

The vertex normal vector is used in Gouraud shading mode (the default shading mode since the advent of Direct3D 6.0) to control lighting and to make some texturing effects. So what's a shading mode?

Shading is the process of performing lighting computations and determining pixel colors from them. These pixel colors will later be drawn on the object. Flat shading lights per polygon or face; Gouraud shading lights per vertex.

### TIP

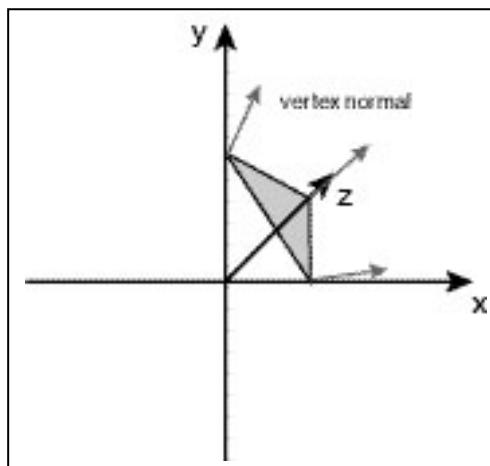
Direct3D applications do not need to specify face normals; the system calculates them automatically when they are needed.



**Figure 4.10:** Teapot as a wireframe model, flat shaded model, and Gouraud shaded model

As you can see, the user might see the faces of the teapot with flat shading, but he gets the illusion of a round teapot by using the Gouraud model. How does that work?

Let's build a simpler example. We'll use the triangle from Figure 4.9. It will appear as a flat triangle when it's shaded with Gouraud shading because all of the vertex normals point the same way, so all the points in the face in between the vertices get the same lighting. It would look the same with flat shading, because flat shading shades per face. Now we'll change the normals to be nonperpendicular:



**Figure 4.11:** Triangle with nonperpendicular normals

With flat shading, nothing would change, because the face hasn't changed. With Gouraud shading, the face appears curled down at the corners, because the direction of the normals varies from vertex to vertex. This is how the teapot is rounded.

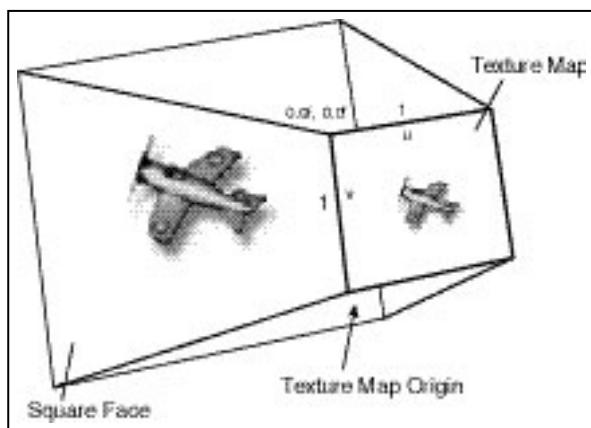
You don't have to do anything in Direct3D to choose the Gouraud shading mode because it's the default shading mode. So . . . no code :-).

### NOTE

A third shading mode, called Phong shading, isn't supported in any popular 3-D real-time API because it costs too much CPU time on current graphics hardware. It shades per pixel.

## TEXTURE-MAPPING BASICS

The 3-D objects in the games of the '80s and early '90s tended to have a shiny plastic look. These solid-colored objects looked rather bland and uninteresting. They lacked the types of markings that give 3-D objects realistic visual complexity: bullet holes, rusted parts, cracks, fingerprints, screws that hold steel plates, and so on. This is done with textures. At its most basic, a texture is simply a bitmap of pixel colors. Texture in this context means a special kind of image that we can map onto a face or a lot of faces.



**Figure 4.12:** Mapping a texture onto a square

In order to map texels, or texture elements, onto primitives, Direct3D requires a uniform address range for all texels in a texture. Therefore, it uses a generic addressing scheme in which all texel addresses are in the range of 0.0 to 1.0, inclusive.

The texture space coordinates are called *u* for the horizontal direction and *v* for the vertical direction. The bottom right corner is (1.0f, 1.0f) and the upper left corner is (0.0f, 0.0f), regardless of the actual size of the texture—even if the texture is wider than it is tall.

You can assign texture coordinates to the vertices that define the primitives in your 3-D world. Our square above has the same aspect ratio (the ratio of width to height) as the texture. You can assign the texture coordinates (0.0, 0.0), (1.0, 0.0), (1.0, 1.0), and (0.0, 1.0) to the primitive's vertices, causing Direct3D to stretch the texture over the entire square. That might work like this:

```
cvVertices [0] = {0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f};           // x, y, z, rhw, tu, tv  
cvVertices [1] = {1.0f, 0.0f, 0.5f, 1.0f, 1.0f, 0.0f};
```

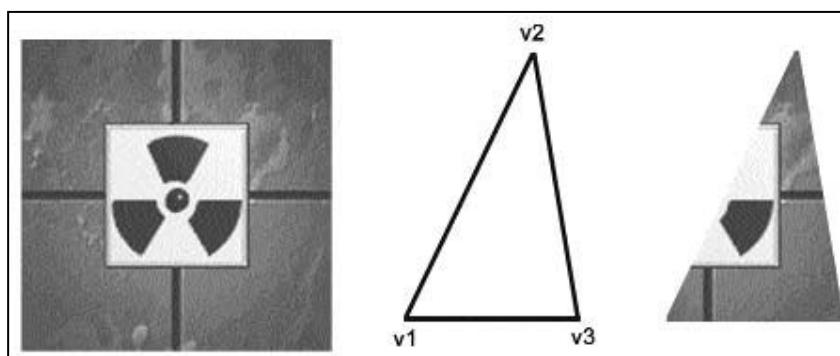
### TIP

**By assigning texture coordinates outside that range, you can create certain special texturing effects. Read more in Part 2 of this book.**

```
cvVertices [2] = {1.0f, 1.0f, 0.5f, 1.0f, 1.0f, 1.0f};  
cvVertices [3] = {0.0f, 1.0f, 0.5f, 1.0f, 0.0f, 1.0f};
```

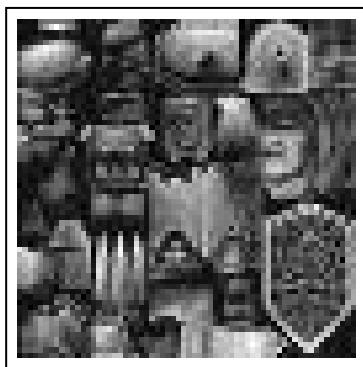
This is a typical vertex structure. It consists of the position vector, a RHW (reciprocal of homogeneous w) value, and the tu and tv coordinates of the texture. We'll concentrate on the last two variables, the texture coordinates. The value 1.0f stretches the textures over the entire object. If you choose instead to apply the texture to a rectangle that's half as wide, you have to decide how to apply it. You may apply the entire texture to the rectangle, which requires you to change the texture's aspect ratio; the texture will be squashed so that it fits the rectangle. The second option is to apply the left or right half of the texture to the rectangle. You have to assign the texture coordinates (0.0, 0.0), (0.5, 0.0), (0.5, 1.0), and (0.0, 1.0) to the vertices.

You can assign parts of a texture to a primitive by using texture coordinates. If you'd like to texture a triangle, it could work like this:



**Figure 4.13:** Mapping parts of a texture on a triangle

That's the easy part. Now let's look at a more advanced level of texture mapping. Let's examine the texture of an .md2 file from Quake II:



**Figure 4.14:** Texture of an .md2 model

Whereas the whole knight as a .md2 model looks like this:



**Figure 4.15:** Textured .md2 model

As you can see, different parts of the texture are mapped on different polygons on this model. You can use a specialized program to paint the texture and to set the texture coordinates on the model. For example, Milkshape 3-D is a wonderful tool to develop such models. You'll find it at [www.swissquake.ch/chulumsoft/ms3d1x/index.html](http://www.swissquake.ch/chulumsoft/ms3d1x/index.html).

I hope you got the big picture on texture mapping. And now we're ready to rock . . . let's get our feet wet with DirectX (try to rap that).

### NOTE

There's a way to change the texture-addressing mode to allow techniques such as wrapping, clamping, and mirroring.

Another thing I'd like to point out: If you scale a texture, the quality of the rendered image will depend on the texture-filtering method you choose. So choosing a filtering method is an important task.

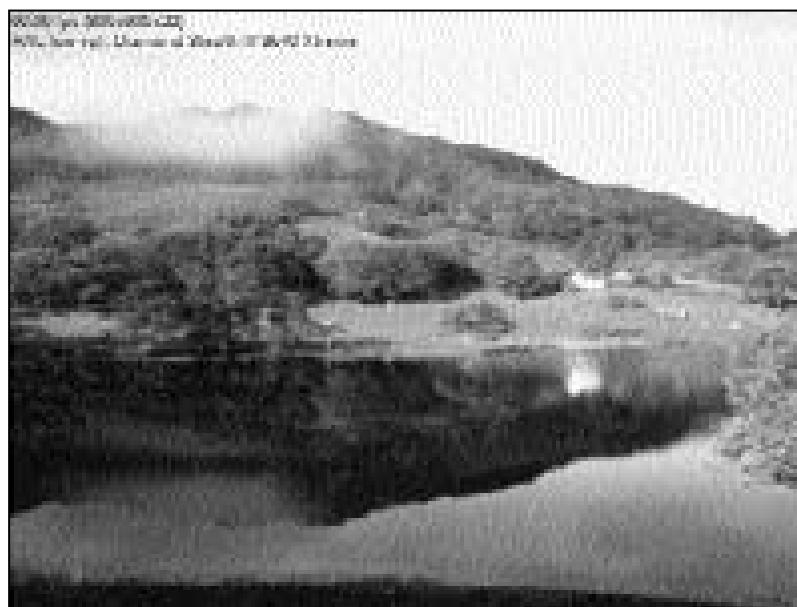
Texture mapping can do a lot of special effects, like light mapping, bump mapping, environment bump mapping, gloss mapping, detail mapping, dot3 bump mapping, and so on. You'll learn about all of these things in Part 2 of this book.

*This page intentionally left blank*

# **CHAPTER 5**

## **THE BASICS**

In our first examples, we will take a look at the basic functionality provided by the Common files framework, and we will build three of the simplest programs I can think of to use it. One of these examples will show a texture thrown at the back of a window and a timer that shows the frames per second.



**Figure 5.1:** Screen shot of the basic example

These little programs will work windowed and full-screen. Press Alt+Enter to switch between these modes. F2 will give you a selection of usable drivers, and Esc will shut down the app.

First, let's talk about the files you need to compile the program. The best place to copy the files is into your DirectX Graphics source file directory in the DirectX 8.0 SDK. On my computer that would be  
D:\mssdk\samples\Multimedia\Direct3D

where D is the letter of the hard drive where you've installed the DirectX SDK.

There are only four program files:

- **basics.cpp.** The main file.
- **winmain.rc.** The resource file (menu, accelerator, icon information, and so on).

- **resource.h.** Header of winmain.rc.
- **directx.ico.** The icon you can see if the program is minimized, or the icon in the left corner of the window's title if maximized.

To compile the whole thing, you should link it with the following lib files as shown in the section on configuring Visual C++:

```
ddraw.lib  
d3dx8.lib  
d3dxof.lib  
d3d8.lib  
winmm.lib  
dxguid.lib  
kernel32.lib  
user32.lib  
gdi32.lib  
winspool.lib  
comdlg32.lib  
advapi32.lib  
shell32.lib
```

Most of these lib files are COM wrappers, as described earlier in the COM section.

## THE DIRECTX GRAPHICS COMMON ARCHITECTURE

The Common files have been well accepted by the game programming community since the time that the betas of the DirectX 8.0 SDK appeared. A lot of demo programmers use them to show their stuff. OK, you're not convinced to use the Common files. So why will we use them? Here are my arguments:

- We avoid how-tos for Direct3D in general and can concentrate on the real stuff.
- It's common and tested ground, which helps us in reducing the debug time.
- All of the Direct3D samples in the DirectX SDK use it. Your learning time is ultrashort.
- Debugging is much easier with the windowed mode it provides.
- You can make your own production code based on the Common files, so knowing them is always a win.

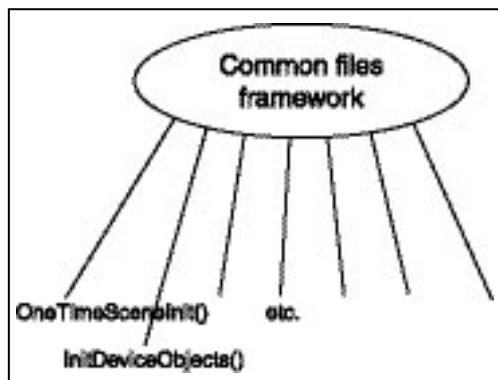
I'll now walk through a high-level view of a few Common files. There are 13 .cpp files, which encapsulate the basic functionality you need to start programming a Direct3D game. The most important files to start our project are

- **d3dapp.cpp.** Exposes the application interface used for samples.
- **d3dutil.cpp and dxutil.cpp.** Provide you with shortcut macros and functions for using DirectX objects.
- **d3dfont.cpp.** Provides a texture-based font class, which is needed to display text on the display.

The d3dapp.cpp module contains the class CD3DApplication. This class publishes the interface for sample programs in the file D3DApp.h. It provides seven functions:

```
virtual HRESULT OneTimeSceneInit() { return S_OK; }
virtual HRESULT InitDeviceObjects() { return S_OK; }
virtual HRESULT RestoreDeviceObjects() { return S_OK; }
virtual HRESULT DeleteDeviceObjects() { return S_OK; }
virtual HRESULT Render() { return S_OK; }
virtual HRESULT FrameMove( FLOAT ) { return S_OK; }
virtual HRESULT FinalCleanup() { return S_OK; }
```

These methods can be called the public interface for the Common files framework.



**Figure 5.2:** Public interface of the framework

Now, let's dive into the source of the first and simplest example of this chapter.

## BASIC EXAMPLE

Here's the application class in basic.cpp:

```
class CMyD3DApplication : public CD3DApplication
{
    LPDIRECT3DVERTEXBUFFER8 m_pVB; // Buffer to hold vertices
    DWORD m_ m_dwSizeofVertices;
    HRESULT ConfirmDevice( D3DCAPS8*, DWORD, D3DFORMAT );
protected:
    HRESULT OneTimeSceneInit();
    HRESULT InitDeviceObjects();
    HRESULT InvalidateDeviceObjects();
    HRESULT RestoreDeviceObjects();
```

```
HRESULT DeleteDeviceObjects();
HRESULT Render();
HRESULT FrameMove();
HRESULT FinalCleanup();
public:
    CMyD3DApplication();
};
```

The first two variables hold the vertices used to render the triangles. In all of the samples, we'll use one of the top features of Direct3D to store vertices: vertex buffers. So what is a vertex buffer? It's an object that encapsulates an array of vertices. You will manage your vertex data memory with a vertex buffer. By allowing the application to allocate and access memory that's usually only accessible by the driver, a developer can minimize data copying and thus maximize performance. Using vertex buffers has several advantages over managing the vertex memory yourself:

- The vertex buffer can be stored in the memory of your graphic card, where it can be accessed very quickly by 3-D hardware. This is important for hardware that accelerates transformation and lighting.
- The *lock/unlock* methods avoid a redundant copy operation by the graphics card driver. Without vertex buffers, the driver is forced to hold a copy of any block of data sent, because there is no guarantee that the application won't scribble over the data as the call returns. With a vertex buffer, the application is required to lock the buffer to change the data, giving the driver the opportunity to block or fail on the `lock()` call. These locks are superfast and unlocks are even faster, so don't worry about the cost, but the hardware may flush the pipeline when calling `Lock()`, so try to keep locks to a minimum

Especially with graphics hardware that supports transformation and lighting in the hardware, vertex buffers give you a huge performance boost. This support will be a basic feature of upcoming graphic cards, so starting with vertex buffers in our basic examples is a must.

Now back to the application class. After the vertex buffer variables, there's a method called `ConfirmDevice()`, which checks the capabilities of the graphics hardware. It's used in every sample in this book and in the SDK, and it helps with stopping the application execution in the case that the graphics hardware won't support the most important features that you might want to show. If your application won't run without the use of Dot3 product bump mapping, you might use the following code:

```
HRESULT CMyD3DApplication::ConfirmDevice( D3DCAPS8* pCaps, DWORD dwBehavior,
                                         D3DFORMAT Format )
{
    if( pCaps->TextureOpCaps & D3DTEXOPCAPS_DOTPRODUCT3 )
        return S_OK;
    return E_FAIL;
}
```

The following protected methods in the application class are sorted in the order they are called. For the permanent initialization, the method `OneTimeSceneInit()` is invoked once per application execution cycle. You can load geometry data, setup calculated values, and so on. Basically, any one-time resource allocation should be performed here. The `*DeviceObjects()` methods are used to do all of the device-dependent stuff, such as loading texture bits onto a device (Note: the word device is normally used for the device driver and sometimes for the graphics card) surface, setting matrices, and populating vertex buffers. They are called by the `d3dapp.cpp` methods in the following order:

- `HRESULT InitDeviceObjects();`
- `HRESULT InvalidateDeviceObjects();`
- `HRESULT RestoreDeviceObjects();`
- `HRESULT DeleteDeviceObjects();`

To get the big picture on the use of these methods, we might differentiate between the following events:

- The application starts up.
- The user changes the device.
- The user resizes the window (only in windowed mode).
- The application shuts down.

`InvalidateDeviceObjects()` is called when the window is resized or the device is changed by the user, whereas the response on this call,

`RestoreDeviceObjects()`, is called when the app starts, the window is resized, or the device has changed. So `InvalidateDeviceObjects()` is not called before `RestoreDeviceObjects()` when your game starts up. In all other cases, these methods build a functional pair.

The `Render()` method is self-explanatory. It is called once per frame and is the entry point for 3-D rendering. It can set up render states, clear the viewport, and render a scene. In an animated program, the method `FrameMove()` is used to hold the whole animation code, such as updating matrices, texture coordinates, object coordinates, and other time-varying activities. Our example doesn't use any animation, so it's not really used in this chapter. Well, `FinalCleanup()`, the last protected method, destroys the allocated memory for the geometry data, deletes the file objects, and so on.

To get a deeper understanding of the inner workings of every Common files method, we'll now work through every method in `basic.cpp`.

### ONETIMESCENEINIT()

The first method that is exported by the Common File framework is `OneTimeSceneInit()`:

```
HRESULT CMyD3DApplication::OneTimeSceneInit()
{
```

### CAUTION

**Be sure that your device-specific resource allocations are matched with deletions, or you will be leaking memory every time a device change happens.**

### NOTE

**FinalCleanup() is the counterpart to OneTimeSceneInit() and destroys any per-application objects.**

```
    return S_OK;  
}
```

We're not loading a scene here, so it's not used. Normally you might use it to load the geometry data of models that are part of a scene.

### INITDEVICEOBJECTS()

The second method that is exported by the Common file framework, or more specifically, by d3dapp.cpp, isn't used here in our simplest basic example. `InitDeviceObjects()` initializes the device-specific things, like textures and, especially in the following examples, textured fonts.

```
HRESULT CMyD3DApplication::InitDeviceObjects()  
{  
    return S_OK;  
}
```

#### TIP

The main functional difference between the `InitDeviceObjects()` and the `RestoreDeviceObjects()` methods is that the first one is not called when the user resizes the window. Catching the “resizing window” event is the job of `RestoreDeviceObjects()`. So if your vertex data is not dependent on the window size of your application (for example, your game uses full-screen-only modes), you might create your vertex and index buffers in `InitDeviceObjects()`. If your vertex data depends on your window size, you might create it in `RestoreDeviceObjects()` method, as in this example.

### RESTOREDEVICEOBJECTS()

The `RestoreDeviceObjects()` method is called after `InitDeviceObjects()` when the app starts up. When the user resizes the window of its application, this method is called after a call to `InvalidateDeviceObjects()`. Most of the real stuff in the first example happens here.

```
HRESULT CMyD3DApplication::RestoreDeviceObjects()  
{  
    // Set the transform matrices  
    D3DXVECTOR3 vEyePt      = D3DXVECTOR3( 0.0f, 0.0f, 2.0f );  
    D3DXVECTOR3 vLookatPt   = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );  
    D3DXVECTOR3 vUpVec      = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );  
    D3DXMATRIX matWorld, matView, matProj;  
    D3DXMatrixIdentity( &matWorld );
```

```

D3DXMatrixLookAtLH( &matView, &vEyePt, &vLookatPt, &vUpVec );
FLOAT fAspect = m_d3dsdBackBuffer.Width / (FLOAT)m_d3dsdBackBuffer.Height;
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, fAspect, 1.0f, 500.0f );
m_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );
m_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );
m_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
// fill the vertex buffer with the new data
D3DVIEWPORT8 vp;
m_pd3dDevice->GetViewport(&vp);
// Initialize four vertices for rendering two triangles
CUSTOMVERTEX cvVertices[] =
{
    { 0.0f, 0.0f, 0.5f, 1.0f, 0xffff0000, }, // x, y, z, rhw, color
    { (float)vp.Width, 0.0f, 0.5f, 1.0f, 0xffff0000, },
    { (float)vp.Width, (float)vp.Height, 0.5f, 1.0f, 0xff00ff00, },
    { 0.0f, (float)vp.Height, 0.5f, 1.0f, 0xff00ffff, },
};
m_dwNumVertices = sizeof(cvVertices);
// Create the vertex buffer
if( FAILED( m_pd3dDevice->CreateVertexBuffer( m_dwNumVertices,
                                                0, D3DFVF_CUSTOMVERTEX,
                                                D3DPPOOL_MANAGED, &m_pVB ) ) )
    return E_FAIL;
VOID* pVertices;
if( FAILED( m_pVB->Lock( 0, m_dwNumVertices, (BYTE**)&pVertices, 0 ) ) )
    return E_FAIL;
memcpy( pVertices, cvVertices, m_dwNumVertices );
m_pVB->Unlock();
return S_OK;
}

```

This code initializes the world, view, and transformation matrices and sets them. The world matrix is just set to the identity matrix with `D3DXMatrixIdentity()` and is not used anymore in the sample. The view matrix is initialized to a few default values:

```

D3DXVECTOR3 vEyePt      = D3DXVECTOR3( 0.0f, 0.0f, 2.0f );
D3DXVECTOR3 vLookatPt   = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
D3DXVECTOR3 vUpVec      = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );

```

`vEyePt` is the point in space where the camera or the eye of the viewer is located. `vLookatPt` is the point at which the viewer looks, and `vUpVec` is a vector that points up in the direction of the y-axis by default. It's useful to determine what's up and down.

So the camera, through which the viewer is looking, is located at 0.0f, 0.0f, 2.0f—two units on the positive z-axis—and looks at the origin of the coordinate system, whereas it is oriented up, in the direction of the positive y-axis. You will learn a lot more about cameras and the viewing matrix in the next chapter. So hold on.

The call to `D3DXMatrixLookAtLH()` initializes a camera for a left-handed coordinate system like that used in Direct3D by default. It offers a very useful camera model.

The aspect ratio of the perspective projection viewing matrix is set dependent on the back buffer size with

```
FLOAT fAspect = m_d3dsdBackBuffer.Width / (FLOAT)m_d3dsdBackBuffer.Height;  
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, fAspect, 1.0f, 500.0f );
```

The size of the back buffer is the size of the render target, which is used by your application to display your game content.

The initialized world, view, and projection matrices are provided to the Direct3D pipeline with a call to `SetTransform()`. But . . . give me a few seconds . . . I'll explain all this matrix stuff in the next chapter in detail. Let's concentrate on the vertex buffer stuff now.

Vertices are the corners of your triangles or other primitives from which you build your faces. Every object in your 3-D world is built from those faces. To store vertices, previous versions of Direct3D have offered you a place in the Direct3D memory chunk. With the arrival of the Direct3D 7.0 API, a new data storage and management model raised his head to the sun. It's called a vertex buffer.

A vertex buffer encapsulates an array of vertices. It allows our example application to allocate and access memory that's usually only accessible by the driver. What's the idea? Normally the driver copies the data from a Direct3D buffer, which is held by the application in its own buffer. With vertex buffers, the data is originally copied in the driver buffer. So we might save the CPU cycles that are needed to copy from Direct3D app buffer to driver buffer. That's a big performance enhancement.

### TIP

**There's also a call to**

**`D3DXMatrixLookAtRH()`, which is used for right-handed coordinate systems, like the one used by OpenGL. You might use this call to port OpenGL applications to Direct3D.**

### NOTE

**The back buffer is the surface, into which you are rendering with the `DrawPrimitive*`() functions. To render into a buffer with these functions means that the render target is the back buffer. The render target could also be a buffer which holds the texture data. So you are able to produce procedural textures by rendering into texture surfaces or texture memory buffers.**

We create the vertex buffer with

```
// Create the vertex buffer
if( FAILED( m_pd3dDevice->CreateVertexBuffer(m_dwSizeofVertices,
                                              0, D3DFVF_CUSTOMVERTEX,
                                              D3DPPOOL_MANAGED, &m_pVB ) ) )
    return E_FAIL;
```

`m_dwSizeofVertices` holds the size of the vertices, `D3DFVF_CUSTOMVERTEX` holds the vertex description, and `m_pVB` holds the returned pointer to the interface of this vertex buffer. Whereas the size of vertices is self-explanatory, we have to take a closer look into the vertex description. This parameter is a flag that describes the vertex format used for this set of primitives. In other words, it tells Direct3D something about the structure that holds the vertex information. It's defined at the beginning of the basic.cpp file with the following command:

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
```

You have to differentiate between the `CUSTOMVERTEX` structure and these flags. Whereas the first holds the information, these flags tell the Direct3D engine how the data is organized in this structure. The `d3d8types.h` header file declares these flags to explicitly describe a vertex format and also provides helper macros that act as common combinations of such flags. Each of the rendering or `CreateVertexBuffer()` methods of `IDirect3DDevice8` accepts a combination of these flags and uses them to determine how a vertex buffer is structured and how to render primitives. Basically, these flags tell the system which vertex components—position, normal, colors, and the number of texture coordinates—your application uses. In other words, the presence or absence of a particular vertex format flag communicates to the system which vertex component fields are present in memory, and which you've omitted. By using only the needed vertex components, your application can conserve memory and minimize the processing bandwidth required to render your 3-D world.

`D3DFVF_XYZRHW` tells your system that your game has already transformed the vertices and Direct3D don't have to do this. Therefore, Direct3D doesn't transform your vertices with the world, view, or projection matrices. It passes them directly to the driver to be rasterized. `D3DFVF_XYZ` tells your system that your game has not transformed vertices and calls for help by Direct3D in doing this.

### NOTE

These flexible vertex format flags and flexible vertex buffers were introduced with Direct3D 6.0, advanced in Direct3D 7.0, and are now only one of two methods to handle vertices in Direct3D 8.0. We've had to differentiate between programmable and nonprogrammable vertex shaders, or so-called fixed-function vertex processing. Vertex shaders control the loading and processing of vertices and are defined at creation as using programmable vertex processing or fixed-function vertex processing. The underlying shading algorithm is always—as stated in Chapter 4—the Gouraud shading algorithm.

**NOTE**

**It wouldn't make any sense to transform the one or two triangles in this example with the Direct3D engine, because they are static.**

D3DFVF\_DIFFUSE indicates that the vertex format includes a diffuse color component.

When using the flexible vertex format flags, mapping of the vertex input registers is fixed so that specific vertex elements such as position or normal must be placed in specified registers located in the vertex input memory. So you have to format all the vertex structures in a specific order:

- Step 1 Position of the transformed/untransformed vertex: float x, y, and z coordinate
- Step 2 RHW as the reciprocal of homogenous w coordinate (only for transformed vertices)
- Step 3 Blending weight values 1–5 floats
- Step 4 Vertex normal as float x, y, and z normal coordinate
- Step 5 Vertex point size
- Step 6 Diffuse color as diffuse RGBA
- Step 7 Specular color as specular RGBA
- Step 8 Texture coordinate sets 1–8 as float u and v coordinates

For example, your vertex structure could look like this:

```
typedef struct
{
    D3DVALUE x,y,z;
    D3DCOLOR diffuse;
    D3DVALUE u,v;
} MYVERTEX;
```

**NOTE**

**You don't have to format the vertices in a specific order when you use explicit shader declarations with names that start with D3DSDE\_\*. These pre-processor macros define the vertex input location into which specific elements must be loaded.**

You have to define the flags as follows:

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1)
```

Another sample could be

```
struct LineVertex
{
    float x, y, z;
};
```

```
#define LINE_VERTEX ( D3DFVF_XYZ )
```

And now, a more complex example:

```
typedef struct SObjVertex
{
    D3DVALUE x, y, z;
    D3DVALUE nx, ny, nz;
    DWORD diffuse;
    DWORD specular;
    TextureCoords2 t0;
    TextureCoords3 t1;
    TextureCoords2 t2;
    TextureCoords2 t3;
} SObjVertex;

DWORD gSObjVertexFVF = (D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_SPECULAR |
                        D3DFVF_NORMAL | D3DFVF_TEX4 |
                        D3DFVF_TEXCOORDSIZE2(0) | D3DFVF_TEXCO-
                        ORDSIZE3(1) |
                        D3DFVF_TEXCOORDSIZE2(2) | D3DFVF_TEXCO-
                        ORDSIZE2(3));
```

Another important concept is the management of resources by Direct3D. With the advent of Direct3D 8.0, a unified resource manager was introduced for textures and geometry. This resource management is used for mipmaps, volume maps, and cube maps and for vertex and index buffers.

The resource manager is indicated by the flag `D3DPPOOL_MANAGED` in the `CreateVertexBuffer()` method call. This flag defines the memory class that holds the vertex buffers. Now, you might ask why we use different memory classes. There's memory that is located in your graphic card (for example, 64MB DDRAM), and there's memory that is located in your system memory (for example, 128MB RAM). Direct3D could manage your application memory requirements in a way that uses the right memory for the right task. But you have to give it a sign indicating what you would like to do with the resource (texture, vertex buffer are for objects, which has to be stored in memory).

Choosing a way to manage the resources or, in other words, defining the memory class that holds the resources is called choosing a pool in Direct3D. There are three different kinds of pools. The flag `D3DPPOOL_MANAGED` is the easiest and best way to map memory to devices. The resources are copied automatically to device-accessible memory as needed. `MANAGED` resources are backed by system memory and do not need to be recreated when a device is lost. Rule of thumb: Use `MANAGED` unless you know a better way. It works for any class of driver and must be used with Unified Memory Architecture mainboards. These are boards where the graphics card memory is located, and they are in the same hardware as the system memory or RAM.

Needs to be in caps.

**NOTE**

The other memory pools are D3DPPOOL\_DEFAULT and D3DPPOOL\_SYSTEMMEM. Each different kind of memory pool is separated from the others.

D3DPPOOL\_DEFAULT places the resources in the memory pool most appropriate for the set of usages requested for the given resource. This is usually video memory, including both local video memory and AGP memory. Textures placed in this pool cannot be locked and are therefore not directly accessible. Instead, you must use functions such as IDirect3DDevice8::CopyRects and IDirect3DDevice8::UpdateTexture to lock the pool. Resources in a D3DPPOOL\_DEFAULT pool are lost when the device is lost. D3DPPOOL\_SYSTEMMEM uses memory that is not typically accessible by the graphics card. As the name implies, this memory is located in the system memory. Within this memory pool, resources do not need to be recreated when a device is lost and can be locked.

After creating the vertex buffer and holding a pointer to the interface of this buffer with `m_pVB` in your hands, you would like to fill the buffer with a call of the following code:

```
VOID* pVertices;  
if( FAILED( m_pVB->Lock( 0, m_dwSizeofVertices, (BYTE**)&pVertices, 0 ) ) )  
    return E_FAIL;  
memcpy( pVertices, cvVertices, m_dwSizeofVertices );  
m_pVB->Unlock();
```

The declaration of `Lock()` looks like

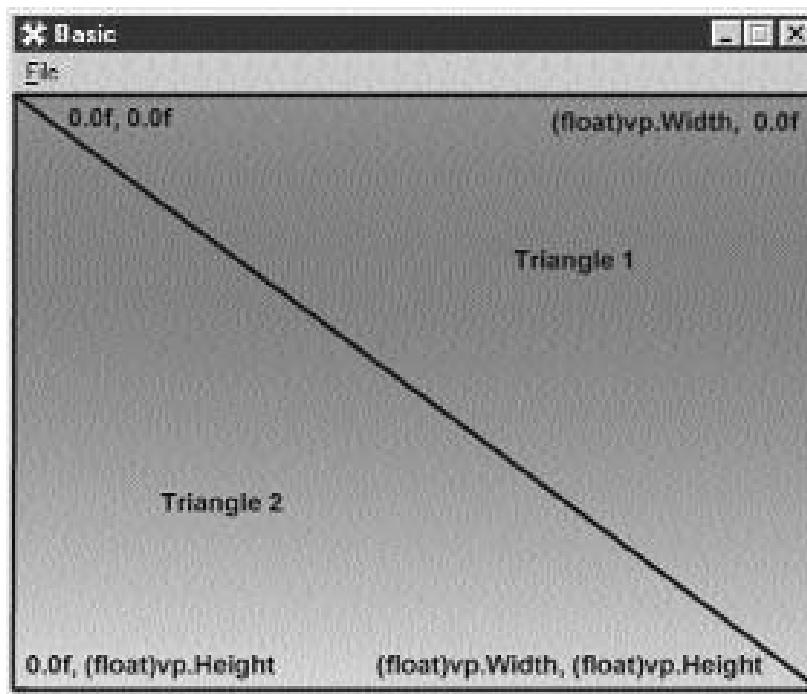
```
HRESULT Lock(  
    UINT OffsetToLock,  
    UINT SizeToLock,  
    BYTE** ppbData,  
    DWORD Flags  
) ;
```

You might lock only parts of the vertex buffer, so a starting point and the size of the data to lock might be provided with `OffsetToLock` and `SizeToLock`. In our case, we're locking the whole buffer starting at 0 and ending at `m_dwSizeofVertices`. The returned pointer `ppbData = pVertices` fills the vertex buffer with the vertices, provided by `cvVertices` with the help of `memcpy()`. Now, what's `cvVertices`?

```
CUSTOMVERTEX cvVertices[] =  
{  
    { 0.0f, 0.0f, 0.5f, 1.0f, 0xffff0000, }, // x, y, z, rhw, color  
    { (float)vp.Width, 0.0f, 0.5f, 1.0f, 0xffff0000, },  
    { (float)vp.Width, (float)vp.Height, 0.5f, 1.0f, 0xff00ff00, },  
    { 0.0f, (float)vp.Height, 0.5f, 1.0f, 0xff00ffff, },  
};
```

This structure holds the four vertices that are built from two triangles:

The first vertex is located at 0.0f, 0.0f in the upper left corner. The second vertex is located in the upper right corner. The third vertex is in the lower right, and the fourth vertex is in the lower left corner. Some x and y values, such as vp.Width and vp.Height, are retrieved from the viewport dimensions. So it's time to shed some light on viewports.



**Figure 5.3:** Triangle fan screen shot

A viewport is the viewable rectangle into which a 3-D scene is projected. This rectangle exists as coordinates within a Direct3D surface (memory chunk) that the system uses as a rendering target. So the viewport is not always the maximum rectangle in which the action is displayed. Think of a cockpit of a helicopter. A lot of screen space is used by the instruments. There's also a tracking monitor, which shows the view from the camera that sits on top of your infrared rocket when it's fired. Updating the instruments in

every frame of the game wouldn't make any sense. The tracking monitor also doesn't need to be updated every frame. So you might choose to update the instruments in their own viewport every five frames and the tracking monitor in its own viewport every second or third frame.

To get the viewport rectangle, we call the `GetViewport()` method of the device:

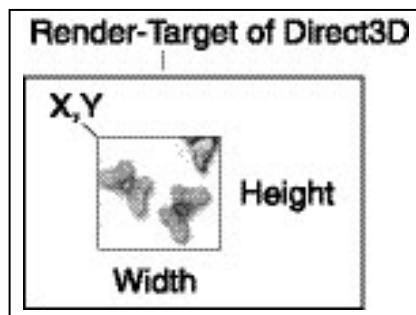
```
D3DVIEWPORT8 vp;  
m_pd3dDevice->GetViewport(&vp);
```

The `D3DVIEWPORT8` structure contains four members—`X`, `Y`, `Width`, and `Height`—that define the area of the render-target surface into which a scene will be rendered.

```
typedef struct _D3DVIEWPORT8 {  
    DWORD      X;  
    DWORD      Y;  
    DWORD      Width;  
    DWORD      Height;  
    float      MinZ;  
    float      MaxZ;  
} D3DVIEWPORT8;
```

These values correspond to the destination rectangle, or viewport rectangle:

The three butterflies are in the viewport rectangle. The one in the upper right corner is clipped to this viewport rectangle.



**Figure 5.4: Viewport rectangle**

In our sample the viewport rectangle has the same size as the rendering target. So getting the viewport values is like getting the `m_d3dsdBackBuffer.Width` and `m_d3dsdBackBuffer.Height` values, which are used for the aspect ratio of the projection matrix in the start-up phase. If the user resizes the window, the `RestoreDeviceObjects()` method is called and the new viewport values are used to fill the newly created vertex buffer. Before the call to `RestoreDeviceObjects()`, the vertex buffer was destroyed in the `InvalidateDeviceObjects()` method call.

**TIP**

**There are two additional members in the D3DVIEWPORT8 structure, called MinZ and MaxZ. These values indicate the depth ranges into which the scene will be rendered and are not used for clipping. This represents different semantics than previous releases of DirectX, in which these members were used for clipping. Normally these members are set to 0.0f and 1.0f to render to the entire range of depth values in the depth buffer. To render a heads-up display in a game, you might set both to 0.0f to force the system to render it in the foreground.**

Now back to the CUSTOMVERTEX structure. As I've shown you before, you should differentiate between the vertex structure and the vertex description. The vertex description tells the Direct3D engine which variables are used in the vertex structure so the engine can preserve the proper amount of memory and switch the proper control flags internally to work with the vertex structure. Our vertex description shows the Direct3D engine with D3DFVF\_XYZRHW that your application is using transformed and lit vertices. D3DFVF\_DIFFUSE indicates that the vertex format includes a diffuse color component. So the vertex structure has to hold x, y, z, and RHW values. There must also be a place for the diffuse color.

As you can see, there are five variables in every line of the CUSTOMVERTEX structure for every vertex. The system requires an already transformed vertex position. So the x and y values must be in-screen coordinates, and z must be the depth value of the pixel, which could be used in a z-buffer (we won't use a z-buffer here). Z values can range from 0.0 to 1.0, where 0.0 is the closest possible position to the viewer, and 1.0 is the farthest position still visible within the viewing area. Immediately following the position, transformed and lit vertices must include an RHW value.

After the vertex buffer is filled with the cvVertices structure, it's unlocked with a call to  
`m_pVB->Unlock();`

After initializing and restoring the device and all the game objects with OneTimeSceneInit(), InitDeviceObjects(), and RestoreDeviceObjects() in the start-up phase of our application, the animation of the game objects can happen in the FrameMove() method.

**NOTE**

Before rasterizing the vertices, you have to convert them from homogeneous vertices to nonhomogeneous vertices, because the rasterizer expects them this way. Homogenous coordinates are used as a fourth dimension, which is useful to translate objects without changing their orientation. In this 4-D space, every point has a fourth component that measures distance along an imaginary fourth-dimensional axis called w. Direct3D converts the homogeneous 4-D vertices to nonhomogeneous 3-D vertices by dividing the x-, y-, and z-coordinates by the w-coordinate and producing an RHW value by inverting the w-coordinate. This is only done for vertices, which are transformed and lit by Direct3D. If they are not transformed and lit by Direct3D, as in this case, the programmer has to provide this value. The RHW value is used in multiple ways: for calculating fog, for performing perspective-correct texture mapping, and for w-buffering, a more precise form of depth-buffering.

**FrameMove()**

Because we're not animating a scene, there doesn't have to be any code inside of FrameMove(). A lot of things will happen here in the upcoming examples.

```
HRESULT CMyD3DApplication::FrameMove()
{
    return S_OK;
}
```

**Render()**

The Render() method is called once per frame and is the entry point for 3-D rendering. It can set up render states, clear the viewport, and render a scene.

```
HRESULT CMyD3DApplication::Render()
{
    // Begin the scene
    if( SUCCEEDED( m_pd3dDevice->BeginScene() ) )
    {
        m_pd3dDevice->SetStreamSource( 0, m_pVB, sizeof(CUSTOMVERTEX) );
        m_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
        m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
        // End the scene.
        m_pd3dDevice->EndScene();
    }
}
```

```
    }  
    return S_OK;  
}
```

Render() calls the BeginScene()/EndScene() pair. The first method is called before performing rendering, the second after rendering. BeginScene() causes the system to check its internal data structures and the availability and validity of rendering surfaces, and it sets an internal flag to signal that a scene is in progress. Attempts to call rendering methods when a scene is not in progress fail, returning D3DERR\_SCENE\_NOT\_IN\_SCENE. Once your rendering is complete, you need to call EndScene(). It clears the internal flag that indicates that a scene is in progress, flushes the cached data, and makes sure the rendering surfaces are OK.

SetStreamSource() binds a vertex buffer to a device data stream. This method provides you with a way to feed different data streams (or different vertex buffers) to the Direct3D rendering engine. That's not possible in conjunction with the fixed-function pipeline we use in this example. The declaration of this method is

### CAUTION

If you like to use GDI functions, make sure that all GDI calls are made outside of the scene functions.

### TIP

Multiple BeginScene()/EndScene() pairs in one application should be omitted. They won't work on scene capture cards like the PowerVR 3/KYRO, and in most situations it's unnecessary.

```
HRESULT SetStreamSource(  
    UINT StreamNumber,  
    IDirect3DVertexBuffer8* pStreamData,  
    UINT Stride  
);
```

A stream is defined as a uniform array of component data, where each component consists of one or more elements representing a single entity such as position, normal, color, and so on. When using the fixed-function pipeline, the stream vertex stride has to match the vertex size, computed from the FVF (flexible vertex format). So sizeof(CUSTOMVERTEX) specifies the size of the component (or vertex structure) in bytes. The second parameter is a pointer to the vertex buffer interface, representing the vertex buffer to bind to the specified data stream. The first parameter is the number of the data stream; you might deliver data to different data streams with that parameter. The fixed-function pipeline, used in this sample, can handle only one data or vertex stream.

The call to SetVertexShader() sets the current vertex shader to a previously created vertex shader (programmable shader) or to a FVF fixed-function shader (a nonprogrammable shader is the same as a fixed-function pipeline). So the parameter of this method is responsible for the choice of the kind of vertex shader.

**NOTE**

Direct3D is able to assemble each vertex that is fed into the processing portion of the pipeline from one or more vertex streams. Having only one vertex stream corresponds to the old pre-DirectX 8 model, in which vertices came from a single source. With DirectX 8, different vertex components can come from different sources; for example, one vertex buffer could hold positions and normals, while a second held color values and texture coordinates. This makes switching between single and multi texture rendering trivial. Just don't enable the stream with the second set of texture coordinates.

**NOTE**

To define a programmable vertex shader, you have to use a handle like `DWORD m_dwDolphinVertexShader2`. You will create the vertex shader with a call to `D3DUtil_CreateVertexShader()` and set it with `D3DUtil_SetVertexShader()` and delete it with `D3DUtil_DeleteVertexShader()`.

The most important method call in `Render()` goes out to

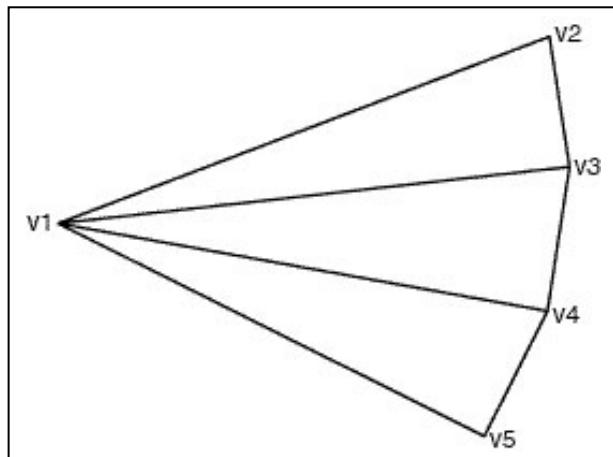
```
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
```

`DrawPrimitive()` has changed a lot since version 7.0. It renders nonindexed, geometric primitives of the specified type from the current set of data input streams, in our case, from the one and only data stream we use. The first parameter defines the primitive type we'd like to use. There are six different primitive types: `D3DPT_POINTLIST`, `D3DPT_LINELIST`, `D3DPT_LINESTRIP`, `D3DPT_TRIANGLELIST`, `D3DPT_TRIANGLESTRIP`, and `D3DPT_TRIANGLEFAN`.

```
typedef enum _D3DPRIMITIVETYPE {
    D3DPT_POINTLIST = 1,
    D3DPT_LINELIST = 2,
    D3DPT_LINESTRIP = 3,
    D3DPT_TRIANGLELIST = 4,
    D3DPT_TRIANGLESTRIP = 5,
    D3DPT_TRIANGLEFAN = 6
    D3DPT_FORCE_DWORD = 0x7fffffff,
} D3DPRIMITIVETYPE;
```

In a TRIANGLEFAN, which is used in this example, all of the triangles share one vertex:

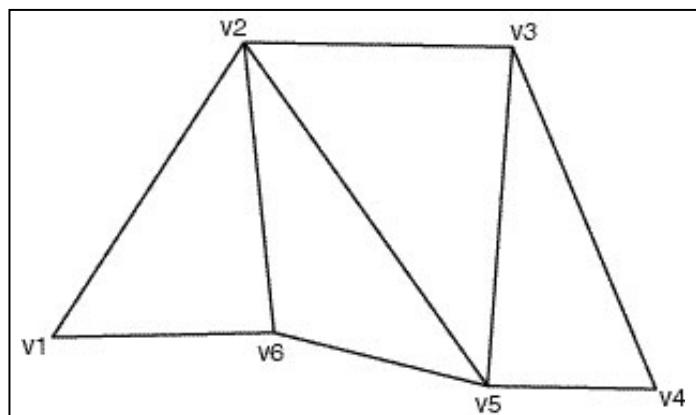
You should differentiate the triangle fan from the triangle strip. In a triangle strip, only the triangles are connected:



**Figure 5.5:** Triangle fan

The second parameter of the `DrawPrimitive()` method marks the starting point where it reads out from the vertex buffer. The last parameter depends on the primitive type you've chosen. You should provide the number of primitives you'd like to render; with a triangle fan, this is the number of triangles.

Now, down to cleanup code.



**Figure 5.6:** Triangle strip

### INVALIDATEDEVICEOBJECTS()

InvalidateDeviceObjects() is called when your game is exiting, the window is resized by the user, or the device is being changed. So it's not called in the start-up phase of your application. If you compare that functionality to DeleteDeviceObjects(), it's also called when the user resizes the window. This is the main aim of this Common file exported method. When the user resizes the window of our small sample, this function is called first and then RestoreDeviceObjects(). In this case, these two methods are a functional pair. Because we built the vertex buffer in the RestoreDeviceObjects() method, we have to destroy it in this method.

```
HRESULT CMyD3DApplication::InvalidateDeviceObjects()
{
    SAFE_RELEASE( m_pVB );
    return S_OK;
}
```

### DELETEDEVICEOBJECTS()

The main target of DeleteDeviceObjects() is the case when the user changes the device. It's a functional pair with InitDeviceObjects(). Because we're not initializing any device-dependent stuff in InitDeviceObjects(), we don't have to delete it here.

```
HRESULT CMyD3DApplication::DeleteDeviceObjects()
{
    return S_OK;
}
```

### FINALCLEANUP()

The last call will go out to FinalCleanup(). It's the last chance to deallocate memory you've grabbed or to destroy any resources you've occupied.

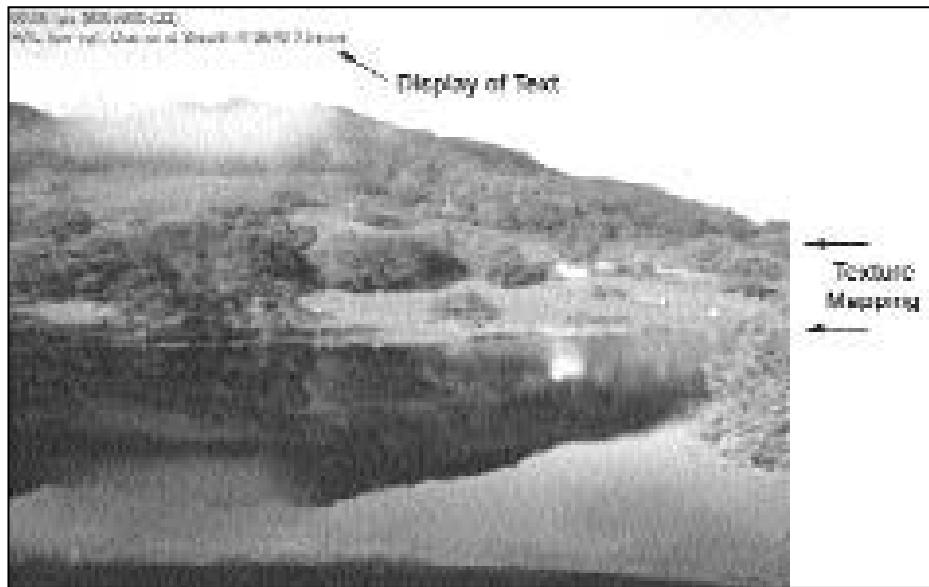
```
HRESULT CMyD3DApplication::FinalCleanup()
{
    return S_OK;
}
```

We haven't occupied anything in OneTimeSceneInit()—a matching pair with FinalCleanup()—so there's no work to do.

Boooommm . . . tada . . . you've worked through your first Direct3D sample. Congratulations!

## BASIC2 EXAMPLE

The next example should give you the big picture of the font and texture class, provided in d3dfont.cpp and d3dutil.cpp. This second example is built on top of the first example in Figure 5.1, so we'll concentrate on things we have to add to the first example.



**Figure 5.7:** Basic example 2

The second example can display text with the help of font textures, and it maps a texture in the visible area of the window.

First, we have to change the vertex description and vertex structure to support our needs, because we don't need the diffuse color component anymore.

```
// A structure for our custom vertex type
struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw; // The transformed position for the vertex
    FLOAT tu, tv; // texture coordinates
};

// Our custom FVF, which describes our custom vertex structure
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_TEX1)
```

Instead, we have to use texture coordinates to place the texture in its proper place. This is done in the vertex structure with tu and tv and in the vertex description with D3DFVF\_TEX1. The structure is used later in `RestoreObjectDevices()` to fill the vertex buffer, and both the structure and description is used in `Render()` and in the `DrawPrimitive()` method to render the whole thing. The vertex description helps the Direct3D engine in the `DrawPrimitive()` call to set the proper flags and to allocate the right amount of memory.

To use the texture font class in `d3dfont.cpp`, we have to call the texture font class constructor in the application class constructor:

```
CMyD3DApplication::CMyD3DApplication()
{
    m_strWindowTitle = _T("Basic");
    m_bUseDepthBuffer = FALSE;
    m_pVB = NULL;
    m_pFont = new CD3DFont( _T("Arial"), 12, D3DFONT_BOLD );
}
```

We use the Arial font with a font height of 12 and a bold font face. These initial values are set by calling the constructor of the CD3DFont class. After setting the default values, the font has to be loaded, using GDI methods in the `InitDeviceObjects()` method of `d3dfont.cpp`. The `InitDeviceObjects()` method of the font class is called in the `InitDeviceObjects()` method of the application class.

### INITDEVICEOBJECTS()

`InitDeviceObjects()` initializes the device-specific things. It's used when your program starts up, when the user changes the device, and when your program shuts down (but not when the user changes the window size). We set up the font objects and create the background texture here:

```
HRESULT CMyD3DApplication::InitDeviceObjects()
{
    m_pFont->InitDeviceObjects( m_pd3dDevice );
    // Load the texture for the background image
    if( FAILED( D3DUtil_CreateTexture( m_pd3dDevice, _T("Lake.bmp"),
        &m_pBackgroundTexture ) ) )
        return D3DAPPERR_MEDIANOTFOUND;
    return S_OK;
}
```

The `InitDeviceObjects()` method of the font class

- Creates one texture with the proper width and height for all font characters
- Creates the font characters
- Copies the font characters into the texture
- We'll concentrate on texture mapping.

`D3DUtil_CreateTexture()` in `d3dutil.cpp` uses the Direct3DX utility library method `D3DXCreateTextureFromFileEx()` to create a texture.

```
HRESULT D3DUtil_CreateTexture( LPDIRECT3DDEVICE8 pd3dDevice,
                               TCHAR* strTexture,
                               LPDIRECT3DTEXTURE8* ppTexture,
                               D3DFORMAT d3dFormat )

{
    // Get the path to the texture
```

```
TCHAR strPath[MAX_PATH];
DXUtil_FindMediaFile( strPath, strTexture );
// Create the texture using D3DX
return D3DXCreateTextureFromFileEx( pd3dDevice, strPath,
                                    D3DX_DEFAULT, D3DX_DEFAULT, D3DX_DEFAULT, 0, d3dFormat,
                                    D3DPPOOL_MANAGED, D3DX_FILTER_TRIANGLE|D3DX_FILTER_MIRROR,
                                    D3DX_FILTER_TRIANGLE|D3DX_FILTER_MIRROR, 0, NULL, NULL, ppTexture );
}
```

D3DXCreateTextureFromFileEx() is a monster method with an overwhelming functionality. It creates a texture from a file specified by an ANSI (American National Standards Institute) string.

```
HRESULT WINAPI D3DXCreateTextureFromFileEx(
    LPDIRECT3DDEVICE8      pDevice,
    LPCSTR                 pSrcFile,
    UINT                   Width,
    UINT                   Height,
    UINT                   MipLevels,
    DWORD                  Usage,
    D3DFORMAT              Format,
    D3DPPOOL                Pool,
    DWORD                  Filter,
    DWORD                  MipFilter,
    D3DCOLOR               ColorKey,
    D3DXIMAGE_INFO*         pSrcInfo,
    PALETTEENTRY*           pPalette,
    LPDIRECT3DTEXTURE8*     ppTexture);
```

pDevice represents the device to be associated with the texture. pSrcFile will hold an ANSI string that specifies the file from which the texture should be created. In the case of the D3DUtil\_CreateTexture() method, DXUtil\_FindMediaFile() will search first in the media path of the SDK. You might specify a width and height of the texture. By providing 0 or D3DX\_DEFAULT—as in the d3dutil.cpp—the dimensions are taken from the source file.

This method could even produce mipmaps for you. A series of mipmaps consists of the same texture in different resolutions. The textures differ with a power of two in their dimensions. Why use the same texture in different resolutions? You wouldn't want to use a texture with the highest resolution if the player is very distant from the object on which the texture is mapped; the player can't see the texture in detail at this distance. So it's cheaper performance-wise to give him the smallest texture scaled at the proper size for the farthest distance, a middle-sized texture for the middle distance, and the biggest and most detailed texture for the small distances. These textures can be produced and used automatically by Direct3D for you. You

only have to tell it to your favorite API. This is done in the `MipLevels` parameter. Providing 0 or `D3DX_DEFAULT` as the `MipLevels` will lead to a complete mipmap chain built from the source file. With any other number, the requested number of mipmaps will be built. The default value used in `d3dutil.cpp` is `D3DX_DEFAULT`, so a complete mipmap chain is built for you and used in your engine.

The `Usage` parameter shows the Direct3D engine if a texture surface will be used as a render target. There are impressive effects you can do with rendering into a texture surface. One of the most common examples in the DirectX SDK is rendering a video on a rotating cube. You might provide 0 or `D3DUSAGE_RENDER-TARGET` as the parameters. If you'd like to render in a texture surface, you have to check the capabilities of your graphics hardware with `CheckDeviceFormat()`, set `Pool` to default with `D3DPOOL_DEFAULT`, and later set the render target with `SetRenderTarget()`. The function `D3DUtil_CreateTexture()` of `d3dutil.cpp` uses 0 as the default value.

The parameter `Format` defines the surface format you'd like to use. So the format of the returned texture might have a different format than the source file. If the format is unknown, it's taken from the source file. The desired format can be provided by the programmer in `D3DXCreateTextureFromFileEx()`. We don't provide a parameter here, so the format of the source file is used.

You know the `Pool` parameter from the `CreateVertexBuffer()` method description from earlier. It defines the memory class in which the texture should reside. One of the Common files, `d3dutil.cpp`, uses `D3DPOOL_MANAGED`, the handiest memory class.

With `Filter` the programmer can control how the image is filtered (that is, picture quality). This is useful when the texture should have a different size than the source file. If the dimensions of the texture should be half of those of the source file, you might provide `D3DX_FILTER_BOX`, which computes the pixel in the texture by averaging a 2-by-2 box from the source image. If the dimensions of the texture should be a quarter of the dimension of the source file, it would be wise to use `D3DX_FILTER_LINEAR`, which computes the pixel in the texture by averaging a 4-by-4 box. This is one of the killer features of this method. The method call of `D3DXCreateTextureFromFileEx()` in `D3DUtil_CreateTexture()` uses the same size for the texture as for the source file, so it uses `D3DX_FILTER_TRIANGLE|D3DX_FILTER_MIRROR`. This tells the Direct3D engine that every pixel in the source file contributes equally to the texture and that the mirror filter, which mirrors the `u`, `v`, and `w` values, should be used.

The same parameters for `Filter` work for the `MipFilter` parameter. To handle visual quality in different dimensions of textures is even more important when using mipmaps. `D3DUtil_CreateTexture()` uses the same parameters here as `Filter`.

The `ColorKey` parameter helps in setting, for example, transparent regions in a texture. It's a `D3DCOLOR` value that replaces a provided color with transparent black. We don't use transparent regions here by default.

`pSrcInfo` might give you a pointer with a `D3DXIMAGE_INFO` structure back, which holds the width, height, number of mipmap levels and the format of the source file. It's not used in the call of `D3DUtil_CreateTexture()`.

If you use palettized textures (256 colors), pPalette will give you back the palette entries in a PALETTEENTRY structure. Using 32-bit textures gives you a higher visual quality. You might see the difference between Quake II and Quake III models: whereas the first one could only use 8-bit textures, the last one has the ability to handle the smoother-looking 32-bit textures. We aim for 32-bit quality, so our D3DXCreateTextureFromFileEx() in D3DUtil\_CreateTexture() call doesn't need to give back a palette structure.

The ppTexture pointer will give you the pointer to the texture interface IDirect3DTexture8, representing the created texture.

If D3DXCreateTextureFromFileEx() succeeds, it will return D3D\_OK.

After digging into the InitDeviceObjects() method of the texture font class and the D3DUtil\_CreateTexture() method of the d3dutil.cpp file, we'll return to the next method that is called by the Common file d3dapp.cpp.

### **RESTOREDEVICEOBJECTS()**

When the application starts up, the RestoreDeviceObjects() method is called after InitDeviceObjects(). It's a different story when the user resizes the window of its application; in this case, RestoreDeviceObjects() is called after a call to InvalidateDeviceObjects(). We have to add one line of code in this second example compared to our first example.

```
m_pFont->RestoreDeviceObjects();
```

The RestoreDeviceObjects() methods of the font class creates a vertex buffer for the font texture and sets the proper render states and texture states.

We have to reflect the change of the vertex structure here.

```
// Initialize four vertices for rendering two triangles
CUSTOMVERTEX cvVertices[] =
{
    { 0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f, }, // x, y, z, rhw, tu, tv
    { (float)vp.Width, 0.0f, 0.5f, 1.0f, 1.0f, 0.0f, },
    { (float)vp.Width, (float)vp.Height, 0.5f, 1.0f, 1.0f, 1.0f, },
    { 0.0f, (float)vp.Height, 0.5f, 1.0f, 0.0f, 1.0f},
};
```

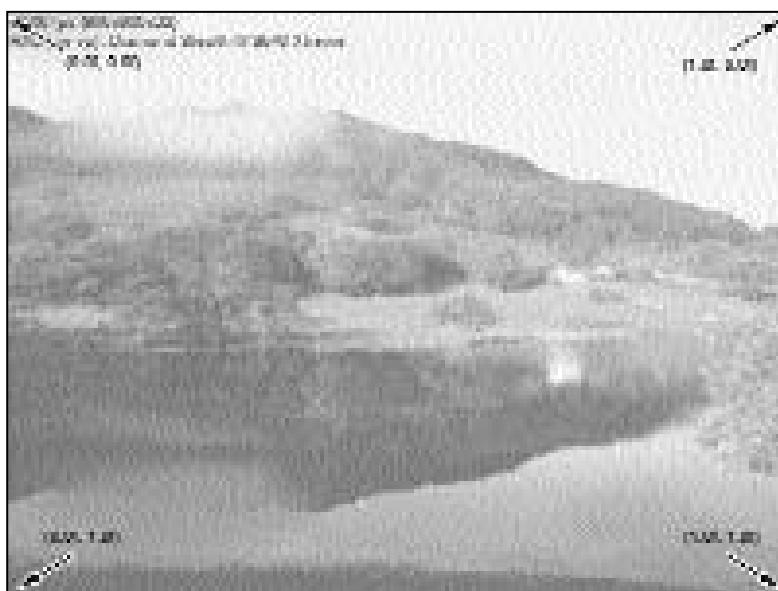
### **CAUTION**

**There are a lot more things you have to think of when using textures with older hardware. You have to limit the texture size if the driver can't handle large textures (for example, the Voodoo 2 and 3 boards can't handle textures bigger than 256 by 256; the newer VSA-100 chips, which are provided with the Voodoo 4 and 5 boards, support 2,048 by 2,048 textures). You might also write a routine that makes the texture square, because some drivers require that. Search through the discussion group provided by Microsoft for Direct3D issues to find problems with different hardware.**

**NOTE**

Setting the render and texture states in the font class is done using so-called state blocks. A state block is a group of device states, render states, lighting and material parameters, transformation states, texture stage states, and/or current texture information. What's your benefit in using state blocks? It's an optimization path; state blocks can be optimized by your device and make applying these states easier. But be careful: there are different opinions between game developers on that subject. There's one group that swears on state blocks and another group that avoids them. So it might depend on hardware and your personal performance tests.

Instead of the diffuse color element in the first example, we now use the tu/tv texture coordinate pair here. These are the coordinates of the background texture.



**Figure 5.8:** Screenshot Basics  
Sample with t

In order to map texels onto primitives, Direct3D requires a uniform address range for all texels in all textures. Therefore, it uses a generic addressing scheme in which all texel addresses are in the range of 0.0 to 1.0, inclusive. To use the whole texture for the background, the last two parameters of every vertex are set with

```
... 0.0f, 0.0f  
... 1.0f, 0.0f  
... 1.0f, 1.0f  
... 0.0f, 1.0f
```

If you'd like to read more on using texture coordinates, go back to Chapter 4.

The FrameMove() method hasn't anything to do, like in the first example, so we skip it. More changes have to be made in the Render() method.

### R E N D E R ()

Render() could set up render states, clear the viewport, and render a scene. As usual, there's the BeginScene()/EndScene() pair.

```
HRESULT CMyD3DApplication::Render()  
{  
    // Begin the scene  
    if( SUCCEEDED( m_pd3dDevice->BeginScene( ) ) )  
    {  
        m_pd3dDevice->SetTexture( 0, m_pBackgroundTexture );  
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );  
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,     D3DTOP_SELECTARG1 );  
        m_pd3dDevice->SetStreamSource( 0, m_pVB, sizeof(CUSTOMVERTEX) );  
        m_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );  
        m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );  
        // Output statistics  
        m_pFont->DrawText( 2, 0, D3DCOLOR_ARGB(255,255,255,0), m_strFrameStats );  
        m_pFont->DrawText( 2, 20, D3DCOLOR_ARGB(255,255,255,0), m_strDeviceStats );  
        // End the scene.  
        m_pd3dDevice->EndScene();  
    }  
    return S_OK;  
}
```

The next call goes out to SetTexture(). The IDirect3DDevice8::SetTexture() method assigns a texture to a given stage for a device. The first parameter must be a number in the range of 0–7, inclusive. Pass the texture interface pointer as the second parameter.

We obtained the texture interface pointer by calling the D3DXCreateTextureFromFileEx() method in the D3DUtil\_CreateTexture() method in InitDeviceObjects().

The SetTexture() method increments the reference count of the texture surface being assigned. When the texture is no longer needed, you should set the texture at the appropriate stage to NULL. If you fail to do this, the surface will not be released, resulting in a memory leak. Since version 6.0, Direct3D maintains a list of up to eight current textures, so Direct3D may support the blending of up to eight textures onto a primitive at once in hardware. But most of the current hardware doesn't support blending more than two textures at once (multitexturing). Blending more than two textures one after another (multipass texturing) is provided by most hardware, but it's slower.

When your application selects a texture as the current texture, it instructs the Direct3D device to apply the texture to all primitives that are rendered from that time until the current texture is changed again. If each primitive in a 3-D scene has its own texture, the texture must be set before each primitive is rendered.

After setting the texture, we have to set the proper texture stages with the following SetTextureStageState() calls.

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,    D3DTOP_SELECTARG1 );
...
HRESULT SetTextureStageState(
    DWORD Stage,
    D3DTEXTURESTAGESTATETYPE Type,
    DWORD Value
);
```

### NOTE

Textures under the legacy **IDirect3D2** interface were manipulated using texture handles. Since the arrival of the **IDirect3D3** (Direct3D 6.0) interface, you create and use textures through interface pointers to the texture surfaces.

### NOTE

The ATI Radeon and the Voodoo 3 chips may blend three textures at once. But the Voodoo 3 will only handle diffuse blending in its third stage and not in the first or second stage, so you're restricted in multitexturing to effects that need the diffuse blending in the third stage. The new VSA-100 chip supports two independent stages. More on that in the part on Multitexturing.

So what is SetTextureStageState() ? It's your one-function-does-it-all solution. It provides the Direct3D texture or pixel engine with the choosed operation and arguments and sets the texture you would like to set. You choose the texture you would like to see with the first parameter by providing the correct

texture stage that holds the texture interface. The second parameter is the render state type you'd like to use. There are a lot of different types, but the most common are the D3DTSS\_COLORx render states that control the flow of an RGB vector and the D3DTSS\_ALPHAx render states that govern the flow of the scalar alpha through parallel segments of the pixel pipeline. The second SetTextureStageState() selects the use of a color operation render state by using D3DTSS\_COLOROP. D3DTOP\_SELECTARG1 parameter tells the texturing unit, to use the first argument as output. The first SetTextureStageState() indicates that this first argument is the texture color of the texture set by SetTexture(). We'll dive deeper into multitexturing/multipass texturing in Part 2 of this book.

The next three calls to SetStreamSource(), SetVertexShader(), and DrawPrimitive() are the same as in the first example. New are the DrawText() methods, which are provided by the Common file d3dfont.cpp.

```
m_pFont->DrawText( 2, 0, D3DCOLOR_ARGB(255,255,255,0), m_strFrameStats );
m_pFont->DrawText( 2, 20, D3DCOLOR_ARGB(255,255,255,0), m_strDeviceStats );
```

The first two parameters determine the place where the text should be displayed. These x and y values have their origin in the upper left corner, as usual. The next parameter is the text color. The alpha, red, and green channels are set to 255, whereas the blue channel is set to 0. If you lower the alpha value, the characters will get transparent.

The last parameters m\_strFrameStats and m\_strDeviceStats are arrays that are put together in d3dapp.cpp. m\_strFrameStats shows the frames per seconds, the width and height of the window, and the color depth, whereas m\_strDeviceStats shows the device type and the device description, for example, hw vp, which means hardware vertex processing and the device name. These calls show you how your application performs and which device it uses. This information is important for when you check your game on different hardware.

### INVALIDATEDEVICEOBJECTS()

InvalidateDeviceObjects() is called when the user resizes the window and, as you have read above, in other cases as well. After that, d3dapp.cpp gives the next call to RestoreDeviceObjects(). So in this case, these two methods are a functional pair. Because we've built the texture fonts in the last method, we have to destroy them here. This is done with the InvalidateDeviceObjects() method of the font class.

```
HRESULT CMyD3DApplication::InvalidateDeviceObjects()
{
    m_pFont->InvalidateDeviceObjects();
    SAFE_RELEASE( m_pVB );
    return S_OK;
}
```

m\_pFont->InvalidateDeviceObjects() destroys the vertex buffer and the state blocks that are used by the font class.

### DELETEDeviceOBJECTS()

`DeleteDeviceObjects()`, as part of the functional pair of `InitDeviceObjects()`, has to destroy the stuff that was initialized by the later method.

```
HRESULT CMyD3DApplication::DeleteDeviceObjects()
{
    m_pFont->DeleteDeviceObjects();
    SAFE_RELEASE( m_pBackgroundTexture );
    return S_OK;
}
```

So here's the place to call the `DeleteDeviceObjects()` method of the font class and to release the allocated memory of the texture.

### FINALCLEANUP()

As usual, the last call will go out to `FinalCleanup()`. We should redo the things that we started in `OneTimeSceneInit()` here.

```
HRESULT CMyD3DApplication::FinalCleanup()
{
    return S_OK;
}
```

We haven't occupied anything in `OneTimeSceneInit()`, so no work to do.

Well . . . now you should get the big picture on using the texture mapping helper functions of `d3dutil.cpp` and the font class of `d3dfont.cpp`, both Common files. Let's dive a little bit deeper in Direct3D programming . . . ever heard of index buffers?

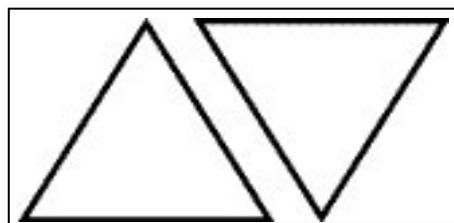
## BASIC3 EXAMPLE

The next basic example will show you the use of index buffers in conjunction with the already used vertex buffers, and it will show you the use of triangle lists. As usual, we'll concentrate on the enhancements on top of the second basic example from Figure 5.7. We declare the index buffer in the application class and initialize it in the class constructor.

```
LPDIRECT3DINDEXBUFFER8 m_pIB;
DWORD m_dwSizeofIndices;
...
CMyD3DApplication::CMyD3DApplication()
{
    m_strWindowTitle = _T("Basic");
```

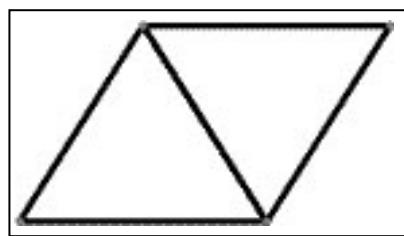
```
m_bUseDepthBuffer = FALSE;  
m_pVB = NULL;  
m_pIB = NULL;  
m_pFont = new CD3DFont( _T("Arial"), 12, D3DFONT_BOLD );  
}
```

OK . . . you'd like to ask why we use such an index buffer and what's its purpose. I have to start with the selection of the primitive type you should choose for your 3-D game. The fastest draw primitive that is supported on current hardware—especially GeForce-driven hardware—is the triangle list.



**Figure 5.9:** Triangle list

A triangle list is a list of isolated triangles. They don't have to be connected in any way, but in real life, they are normally connected. If they are not, you have to handle six vertices. If they are, it's the same story, but you might reduce that to four vertices with the help of an index buffer. Let's face a vertex buffer with only four vertices in conjunction with a triangle list.



**Figure 5.10:** Triangle list with index buffer

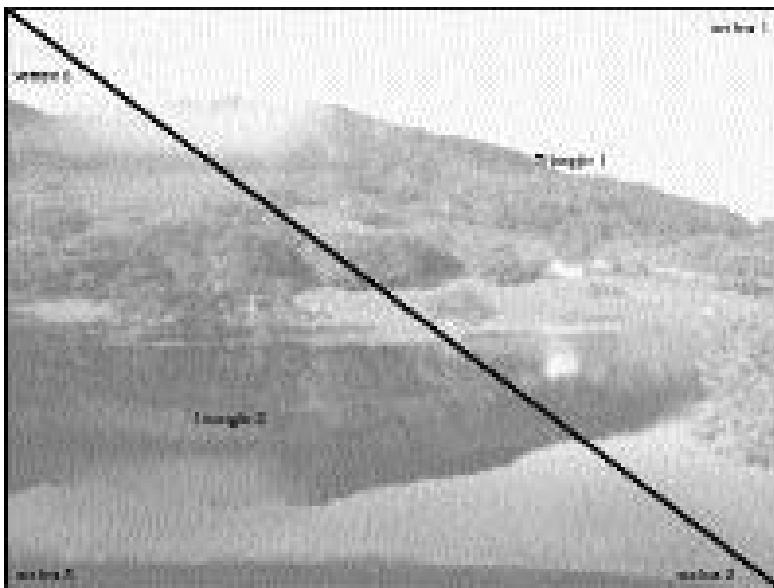
So what does the index buffer do? It holds the position of the different vertices in the vertex buffer. So it's an integer offset in the vertex buffer. If `DrawIndexedPrimitive()` draws the first triangle, it will start, for example, on the left vertex, draw a line to the upper vertex of the left triangle, and then connect the upper with the bottom right vertex and, finally, with the two bottom vertices. If it starts drawing the second triangle, it will get the upper left vertex of the second triangle from the cached data of the vertex buffer with the help of the index buffer, which holds the offset to this vertex for the second triangle. So it's reusing the vertices from the first triangle. After drawing the left upper vertex, `DrawIndexedPrimitive()` will draw the fourth and last vertex stored in the vertex buffer and then reuse the bottom vertex from the first triangle. Now in code:

```
{ 0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f, }, // x, y, z, rhw, tu, tv  
{ (float)vp.Width, 0.0f, 0.5f, 1.0f, 1.0f, 0.0f, },  
{ (float)vp.Width, (float)vp.Height, 0.5f, 1.0f, 1.0f, 1.0f, },  
{ 0.0f, (float)vp.Height, 0.5f, 1.0f, 0.0f, 1.0f},
```

The vertex buffer holds only four vertices for the triangle list. The index buffer indexes these vertices.

```
WORD dwIndices[] = {0, 1, 2, 0, 2, 3};
```

This tells the `DrawIndexedPrimitive()` that it should use vertex 0, 1, and 2 for the first triangle and vertex 0, 2, and 3 for the second triangle.



**Figure 5.11:** Triangle list with index buffer

OK, you might argue: Why didn't we stay with triangle strips or triangle fans? Using triangle strips or triangle fans this way looks easier and more consistent.

Now the basic question: Which type of primitive should I use? In a lot of applications vertices are shared by multiple polygons. You'll get maximum performance when you reduce the duplication in vertices transformed and sent across the bus to the rendering device. A nonindexed triangle list achieves no vertex sharing, so it's the least optimal method. Strips and fans imply a specific connectivity relationship between polygons and may use indexed lists. When the structure of the data falls naturally into strips and fans, this is the most appropriate choice, because they minimize the data sent to the driver. But if you have to decompose meshes into strips and fans, a large number of `DrawPrimitive()` calls have to be made

because of the large number of pieces. Because of this call overhead, the most efficient method is usually to use a triangle list with indexed primitives.

In short: If your data naturally falls into large strips or fans, then use strips or fans; otherwise, use indexed triangle lists. Now . . . back to code.

#### **RESTOREDEVICEOBJECTS()**

When the application starts up, the `RestoreDeviceObjects()` method is called after `InitDeviceObjects()`.

```
    return E_FAIL;
VOID* pVertices;
if( FAILED( m_pVB->Lock( 0, m_dwSizeofVertices, (BYTE**)&pVertices, 0 ) ) )
    return E_FAIL;
memcpy( pVertices, cvVertices, m_dwSizeofVertices );
m_pVB->Unlock();
// Initialize the Index buffer
WORD dwIndices[] = {0, 1, 2, 0, 2, 3};
m_dwSizeofIndices = sizeof(dwIndices);
// Create the index buffer
if( FAILED( m_pd3dDevice->CreateIndexBuffer( m_dwSizeofIndices,
                                              0, D3DFMT_INDEX16,
                                              D3DPPOOL_MANAGED, &m_pIB ) ) )
    return E_FAIL;
VOID* pIndices;
if( FAILED( m_pIB->Lock( 0, m_dwSizeofIndices, (BYTE**)&pIndices, 0 ) ) )
    return E_FAIL;
memcpy( pIndices, dwIndices, sizeof(dwIndices) );
m_pIB->Unlock();
return S_OK;
}
```

The real new stuff is the index buffer creation and filling. `CreateIndexBuffer()` needs the length of the index buffer as the first parameter and the format of the index buffer as the third parameter.

```
HRESULT CreateIndexBuffer(
    UINT Length,
    DWORD Usage,
    D3DFORMAT Format,
    D3DPPOOL Pool,
    IDirect3DIndexBuffer8** ppIndexBuffer
);
```

You might choose to set up the index buffer in a WORD or DWORD format with `D3DFMT_INDEX16` or `D3DFMT_INDEX32`, depending on the number of vertices you need. We don't specify anything for the `Usage` parameter. The `Usage` structure member contains general capability flags. A `D3DUSAGE_SOFTWAREPROCESSING` flag, for example, indicates that the index buffer is to be used with software vertex processing. It can be set when mixed-mode or software processing mode is enabled for that device. It is the only option when a single buffer is used with both hardware and software vertex processing.

We use the `D3DPPOOL_MANAGED` resource pool for the index buffer as we did for the vertex buffer. The pointer to the index buffer interface is returned in the last parameter `m_pIB`.

**NOTE**

The presence of the **D3DUSAGE\_WRITEONLY** flag indicates that the index buffer memory is used only for write operations. The driver will place the index data in the best memory location.

Otherwise, the driver won't place the data in a location that is inefficient for read operations. If this flag is not specified, it is assumed that applications perform read and write operations on the data in the index buffer.

The Lock() and Unlock() methods help to fill the buffer. Its parameters are the same as the Lock()/Unlock() pair of the vertex buffer interface.

**RENDER()**

As usual, there's the BeginScene()/EndScene() pair in Render().

```
HRESULT CMyD3DApplication::Render()
{
    // Begin the scene
    if( SUCCEEDED( m_pd3dDevice->BeginScene() ) )
    {
        m_pd3dDevice->SetTexture( 0, m_pBackgroundTexture );
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,    D3DTOP_SELECTARG1 );
        m_pd3dDevice->SetStreamSource( 0, m_pVB, sizeof(CUSTOMVERTEX) );
        m_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
        m_pd3dDevice->SetIndices( m_pIB, 0 );
        m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                            0,
                                            4, // number of vertices
                                            0,
                                            2); // number of primitives
        // Output statistics
        m_pFont->DrawText( 2, 0, D3DCOLOR_ARGB(255,255,255,0), m_strFrameStats
    );
}
```

**NOTE**

The performance of index processing operations depends heavily on where the index buffer exists in memory and what type of rendering device is being used.

```

    m_pFont->DrawText( 2, 20, D3DCOLOR_ARGB(255,255,255,0), m_strDeviceStats
);
    // End the scene.
    m_pd3dDevice->EndScene();
}
return S_OK;
}

```

There are two changes compared to the second example. There's a call to `SetIndices()` and a call to `DrawIndexedPrimitive()` instead of `DrawPrimitive()`. `SetIndices()` sets the current index array.

The second parameter specifies the base value for the indices. This is the starting position in the vertex data streams, which is added to all indices prior to referencing into the vertex data streams.

`DrawIndexedPrimitive()` is the primitive method that uses only indexed primitives.

```

HRESULT DrawIndexedPrimitive(
    D3DPRIMITIVETYPE Type,
    UINT MinIndex,
    UINT NumVertices,
    UINT StartIndex,
    UINT PrimitiveCount
);

```

As the primitive type, we've chosen this time the triangle list, as described above. The second parameter that `DrawIndexedPrimitive()` accepts is the minimum vertex index for vertices used during this call. `MinIndex` and all the indices in the index stream are relative to the `BaseVertexIndex`, set during the `SetIndices()` call. The third parameter is the number of indices to use during this call starting from `BaseVertexIndex` and `MinIndex`. The fourth parameter is the location in the index array to start reading indices. The final parameter that `DrawIndexedPrimitive()` accepts is the number of primitives to render. This parameter depends on the primitive count and the primitive type. The code sample above uses triangles, so the number of primitives to render is the number of indices divided by three.

You may calculate the number of primitives from the index count this way:

- D3DPT\_POINTLIST : `PrimitiveCount = n`
- D3DPT\_LINELIST : `PrimitiveCount = n/2`
- D3DPT\_LINESTRIP : `PrimitiveCount = n-1`
- D3DPT\_TRIANGLELIST : `PrimitiveCount = n/3`
- D3DPT\_TRIANGLESTRIP : `PrimitiveCount = n-2`
- D3DPT\_TRIANGLEFAN : `PrimitiveCount = n-2`

### NOTE

In case of more than one data stream, this single index set is used to index all streams.

The last improvement in the third example compared to the second example is made in the `InvalidateDeviceObjects()` method.

#### `INVALIDATEDEVICEOBJECTS()`

Because we've built the index buffer in the `RestoreDeviceObjects()` method, we have to destroy it here. This is done with the `SAFE_RELEASE` macro.

```
HRESULT CMyD3DApplication::InvalidateDeviceObjects()
{
    m_pFont->InvalidateDeviceObjects();
    SAFE_RELEASE( m_pVB );
    SAFE_RELEASE( m_pIB );
    return S_OK;
}
```

In this last basic example, you've learned the use of index buffers. That was pretty straightforward. The more difficult part was the parameters of `DrawIndexedPrimitive()`. In the next examples, we'll use `FrameMove()` a lot more.

# **CHAPTER 6**

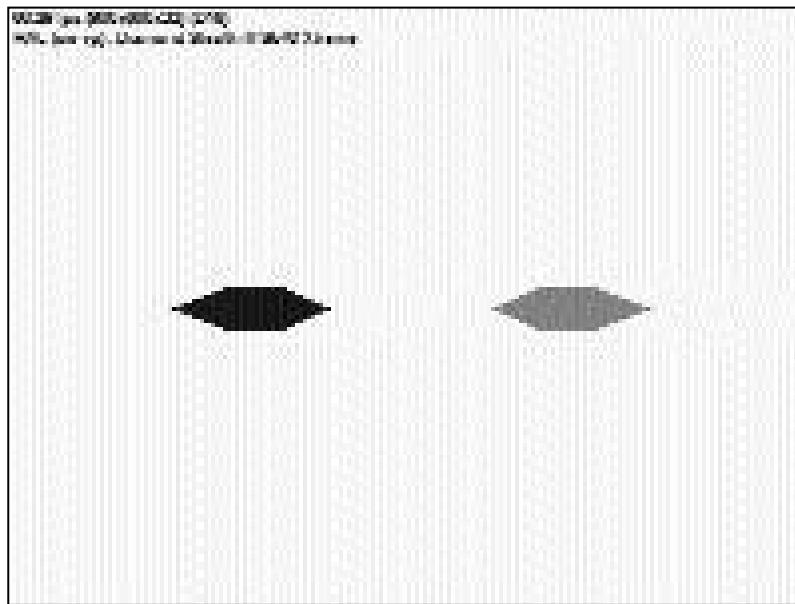
## **F1RST STEPS TO ANIMATION**

You may recall that in the last chapter we saw that all samples that are built with the help of the Common files in the DirectX SDK are created by providing overloaded versions of the CD3DApplication methods:

```
virtual HRESULT OneTimeSceneInit()      { return S_OK; }
virtual HRESULT InitDeviceObjects()       { return S_OK; }
virtual HRESULT RestoreDeviceObjects()   { return S_OK; }
virtual HRESULT DeleteDeviceObjects()    { return S_OK; }
virtual HRESULT Render()                 { return S_OK; }
virtual HRESULT FrameMove( FLOAT )      { return S_OK; }
virtual HRESULT FinalCleanup()          { return S_OK; }
```

We also learned the task of every method in this class.

In this chapter, we will start to write our first animated app. The sample provided will show a red and a yellow object, which can be rotated around their x and y axes.



**Figure 6.1:** Our first animated example

The application uses a z-buffer and the simplest keyboard interface I can think of. You can move and rotate the camera with the Up Arrow, Down Arrow, Left Arrow, Right Arrow, C, and X keys. The rotation and movement of the camera feels a little bit like the first X-Wing games. Only a space scanner is missing :-).

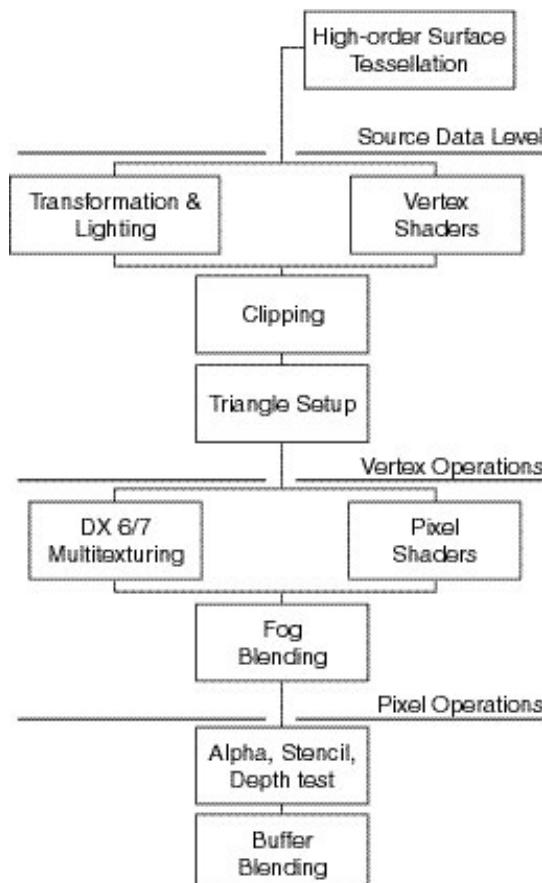
You may switch between the full-screen and windowed mode with Alt+Enter. F2 will give you a selection of usable drivers, and Esc will shut down the app.

The sample comes in two flavors: one with a camera that is rotated about the camera axis with the help of matrices, and one with a camera that is rotated with the help of a quaternion. The second example has an improved user interface. You should be able to slide the camera with the 4, 6, 8, and 2 keys of your number pad.

To compile the source, take a look at the sections on installing Visual C++ 6.0 and the DirectX SDK in the Introduction.

## THE THIRD DIMENSION

You need a good familiarity with 3-D geometric principles to program Direct3D applications. The first step in understanding these principles is understanding the transformation and lighting pipeline as part of the Direct3D pipeline:



**Figure 6.2: The Direct3D pipeline**

I'll focus here on the transformation and lighting (T&L) pipeline. You can think of the process of texturing and lighting as an assembly line in a factory, in which untransformed and unlit vertices enter one end, several sequential operations are performed on them, and at the end of the assembly line, transformed and lit vertices exit. A lot of programmers have implemented their own transformation and lighting algorithms. They can disable parts of this pipeline and send the vertices that are already transformed and lit to Direct3D.

In most cases it will be the best to use the Direct3D T&L pipeline, because it's really fast, especially with the new T&L graphics processors that are provided with the ATI Radeon, NVIDIA GeForce2 Ultra, and S3 Savage 2000 chipsets.

These graphics processors gained an important role in the last couple of years. Most tasks in the rendering pipeline are nowadays computed by the GPU:

### NOTE

The vertex shader unit was introduced with DirectX 8.0. Before then, Direct3D transformed and lit vertices using a fixed-function—so-called T&L—pipeline. Vertex shaders now supply the underlying algorithms used to transform and light the vertex data. This takes the form of small programs executed for each vertex that use instructions specific to DirectX Graphics.

Move Objects and Camera, Artificial Intelligence (AI), Realistic Physics	CPU	CPU	CPU	CPU	CPU
Scene Level Calculations, High-Order Surface Tessellation	CPU	CPU	CPU	CPU	CPU
 Transform	CPU	CPU	CPU	GPU	GPU
Lighting	CPU	CPU	CPU	GPU	GPU
Triangle Setup and Clipping	CPU	Graphics Processor	Graphics Processor	GPU	GPU
Rendering	Graphics Processor	Graphics Processor	Graphics Processor	GPU	GPU

1996      1997      1998      1999      2000

**Figure 6.3:** Use of a graphics processor

In the old days, transformation and lighting functions of the 3-D pipeline were performed by the CPU of the PC. At the moment, a lot of game programmers use only the hardware transformation pipeline, because hardware transformation pipeline lighting is not as efficient or as beautiful as they expect it to be. Others use a mixture of hardware and software lighting.

Since 1999 affordable graphic cards with dedicated hardware T&L acceleration are available. With these cards, a higher graphics performance is possible because they can process graphic functions up to four times the speed of the leading CPUs. On the other side, the CPU can now be better-utilized for functions such as sophisticated AI (artificial intelligence), realistic physics, and more complex game elements.

The existing generation of new cards provides a lot of horsepower for the new game generation. It's a great time for game programmers :-).

### NOTE

There are other equivalent features that are supported by cards without GPU. For example, good anti-aliasing is an important improvement in picture quality. But I think all the graphic hardware producers agree that a GPU is something that has to be implemented in every consumer graphics 3-D card by the end of the year 2001.

## TRANSFORMATION PIPELINE

To describe and display 3-D graphics objects and environments is a complex task. Describing the 3-D data according to different frames of reference or different coordinate systems reduces the complexity. These different frames of reference are called *spaces*, such as model space, world space, view space, and projection space. Because these spaces use different coordinate systems, 3-D data must be converted or “transformed” from one space to another.

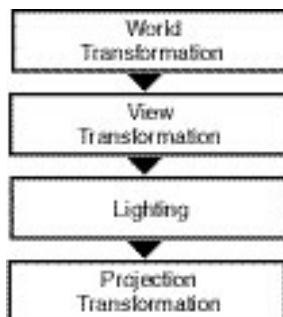


Figure 6.4: Transformation pipeline

The transformation pipeline transforms each vertex of an object from an abstract, floating-point coordinate space into pixel-based screen space, taking into account the properties of the virtual camera used to render the scene. This transformation is done with three transformation matrices: the world, view, and projection matrices. The use of world, view, and projection transformations ensures that Direct3D has to deal with only one coordinate system per step. Between those steps, the objects are oriented in a uniform manner.

The *world transformation* stage transforms an object from model or object space to world space. *Model space* is the coordinate space in which an object is defined, independent of other objects and the world itself. In model space, the points of the model or object are rotated, scaled, and translated to animate it. For

example, think of a Quake III model that rotates his torso and holds his weapon in your direction. With model space, it's easier and faster to move an object by simply redefining the transformation from model space to world space than it would be to manually change all the coordinates of the object in world space. For example, to rotate a sphere or cube around its center looks more natural and is much easier when the origin is at the center of the object, regardless of where in world space the object is positioned. *World space* is the absolute frame of reference for a 3-D world; all object locations and orientations are with respect to world space. It provides a coordinate space that all objects share, instead of requiring a unique coordinate system for each object.

To transform the objects from model space to world space, the whole object will be rotated about the x-axis, y-axis, or z -axis; scaled (enlarging or shrinking the object); and translated by moving the object along the x-axis, y-axis, or z-axis to its final orientation, position, and size.

Direct3D uses a left-handed coordinate system by default, in which every positive axis (x, y, or z) is pointing away from the viewer.

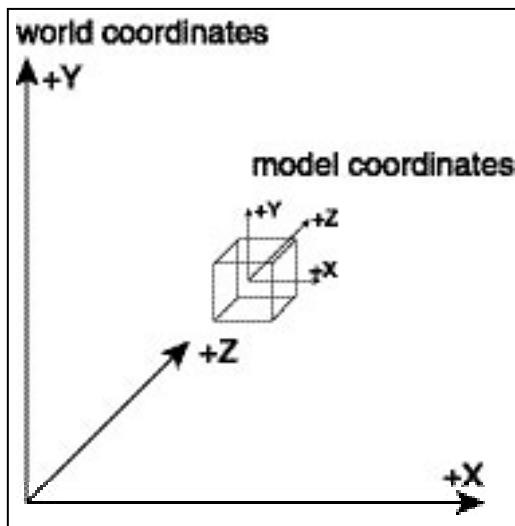
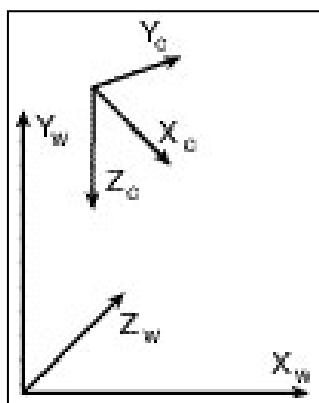
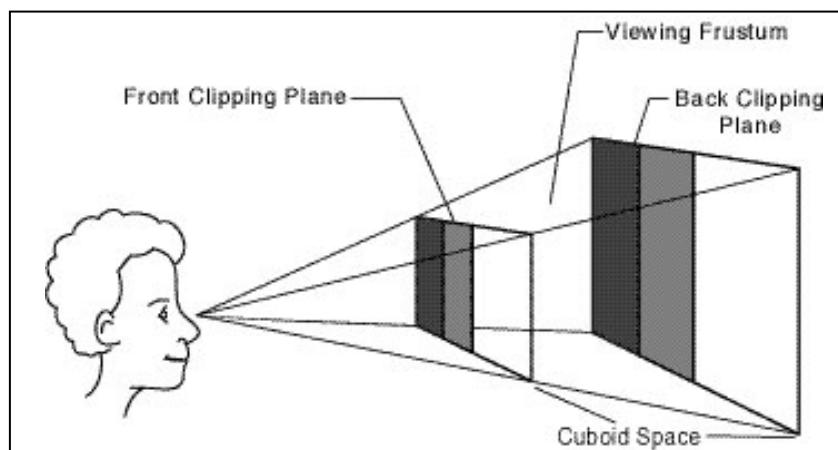


Figure 6.5: World and model space

The *view transformation* stage transforms the objects from world space into *camera space* or *view space*. This transformation puts the camera in the origin and points it directly down the positive z-axis. The geometry has to be translated in suitable 2-D shapes. It's also used for lighting and backface culling. The light coordinates, which are specified in world space, are transformed into camera space at this stage, and the effect of lights is calculated and applied to the vertices.

**Figure 6.6:** View or camera space

The *projection transformation* takes into consideration the camera's horizontal and vertical fields of view, so it applies perspective to the scene. Objects that are close to the front of the frustum are expanded, and objects close to the end are shrunk. It warps the geometry of the frustum of the camera into a cube shape by setting a 4-by-4 matrix to a perspective projection matrix built from the field-of-view or viewing frustum, aspect ratio, near plane or front clipping plane, and far plane or back clipping plane. This makes it easy to clip geometry that lies both inside and outside of the viewing frustum. You can think of the projection transformation as controlling the camera's internals; it is analogous to choosing a lens for the camera.

**Figure 6.7:** Projection space

## TRANSFORMATION MATH

Let's give our old math teachers a smile :-). I learned math from the early '70s to the mid-'80s at school (yes, we've got another education system here in Germany). At that time, I never thought that there would be such an interesting use for it (that is, game programming). I wonder if math teachers today talk about the use of math in computer games.

Any impressive game requires correct transformations. Consider the following example. An airplane, let's say an F22, is oriented such that its nose is pointing in the positive z direction, its right wing is pointing in the positive x direction, and its cockpit is pointing in the positive y direction. So the F22 local x, y, and z axes are aligned with the world x, y, and z axes. If this airplane is rotated about 90 degrees around its y-axis, its nose would be pointing toward the world negative x-axis, its right wing toward the world negative z-axis, and its cockpit will remain in the world positive y direction. From this new position, rotate the F22 about its z axis. If your transformations are correct, the airplane will rotate about its own z-axis. If your transformations are incorrect, the F22 will rotate about the world z-axis.

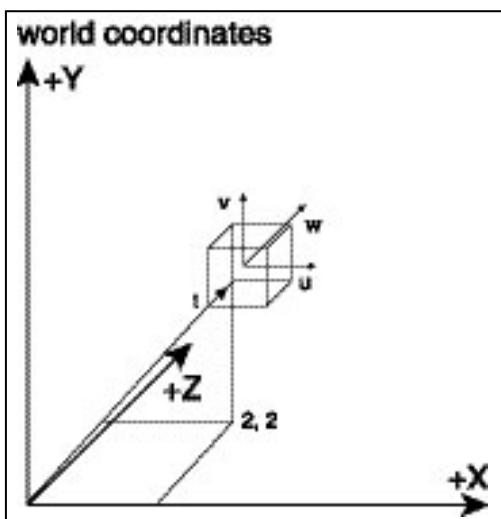
In Direct3D you can guarantee the correct transformation by using 4-by-4 matrices. A lot of people—especially mathematicians—like to speak of mapping from one space to another space instead of transforming one space to the other.

## MATRICES

Matrices are rectangular arrays of numbers. A 4-by-4 world matrix contains four vectors, which represent the world space coordinates of the x, y, and z unit axis vectors, and the world space coordinate, which is the location of these axis vectors:

ux	uy	uz	0
vx	vy	vz	0
wx	wy	wz	0
tx	ty	tz	1

The u, v, and w vectors represent the so-called rigid body. This is the matrix that defines the mapping from object space to world space. Graphically it could look like this:

Figure 6.8: The  $u$ ,  $v$ ,  $w$ , and  $t$  vectors

To describe the position of this cube, the matrix has to look like:

1	,	0	,	0	,	0
0	,	1	,	0	,	0
0	,	0	,	1	,	0
2	,	2	,	2	,	1

The cube is oriented like the coordinate system. The first row contains the world space coordinates of the local x-axis. The second row contains the local y-axis, and the third row contains the world space coordinates of the local z-axis. The vectors are unit vectors, the magnitude of which are 1. The last row contains the world space coordinates of the object's origin. You might translate the cube with these.

A special matrix is the identity matrix:

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

The identity matrix represents a set of object axes that are aligned with the world axes. The world x coordinate of the local x-axis is 1, the world y and z coordinates of the local x-axis are 0, and the origin vector is  $(0, 0, 0)$ . So the local model x-axis lies directly on the world x-axis. The same is true for the local x- and y-axes. So it's a "get back to the roots" matrix. If an object's position in model space corresponds to its position in world space, simply set the world transformation matrix to the identity matrix. This matrix and all other matrices in Direct3D could be accessed like this:

```
D3DMATRIX mat;
mat._11 = 1.0f; mat._12 = 0.0f; mat._13 = 0.0f; mat._14 = 0.0f;
mat._21 = 0.0f; mat._22 = 1.0f; mat._23 = 0.0f; mat._24 = 0.0f;
mat._31 = 0.0f; mat._32 = 0.0f; mat._33 = 1.0f; mat._34 = 0.0f;
mat._41 = 0.0f; mat._42 = 0.0f; mat._43 = 0.0f; mat._44 = 1.0f;
```

A typical transformation operation is a 4-by-4 matrix multiplication operation. A transformation engine multiplies a vector representing 3-D data, typically a vertex or a normal vector, by a 4-by-4 matrix. The result is the transformed vector. This is done with standard linear algebra:

Transform	Original		Transformed
Matrix	Vector		Vector
a b c d	x	= ax + by + cy + dw	x'
e f g h	x	= ex + fy + gz + hw =	y'
i j k l	z	= ix + jy + kz + lw	z'
m n o p	w	= mx + ny + oz + pw	w'

Before a vector can be transformed, a transform matrix must be constructed. This matrix holds the data to convert vector data to the new coordinate system. Such an interim matrix must be created for each action (scaling, rotation, and transformation) that should be performed on the vector. Those matrices are multiplied together to create a single matrix that represents the combined effects of all of those actions (matrix concatenation). This single matrix, called the transform matrix, could be used to transform one vector or one million vectors. The time to set it up amortizes by the ability to reuse it.

The concatenation of the world, view, and projection matrices is handled by Direct3D internally.

One of the pros of using matrix multiplication is that scaling, rotation, and translation all take the same amount of time to perform. So the performance of a dedicated transformation engine is predictable and consistent. This allows game developers to make informed decisions regarding performance and quality.

## THE WORLD MATRIX

Usually the world matrix is a combination of translation, rotation, and scaling the matrices of the objects. Code for a translate-and-rotate world matrix could look like this:

```
struct Object
{
    D3DXVECTOR3 vLoc;           // Location/Translation
    D3DXVECTOR3 vr;             // Rotation vector
    FLOAT fr, fg, fb;          // Color of the object
```

```
D3DMATRIX    matLocal;
};

...
class CMyD3DApplication : public CD3DApplication
{
    ...
    Object    m_pObjects[NUM_OBJECTS];
    ...
};

// in FrameMove()
for (WORD i = 0; i < dwNumberofObjects; i++)
{
    D3DXMATRIX matWorld;
    D3DXMatrixTranslation(&matWorld, m_pObjects[0].vLoc.x,
                           m_pObjects[0].vLoc.y,
                           m_pObjects[0].vLoc.z);
    D3DXMATRIX matTemp, matRotateX, matRotateY, matRotateZ;
    D3DXMatrixRotationY( &matRotateY, -m_pObjects[0].vR.x );
    D3DXMatrixRotationX( &matRotateX, -m_pObjects[0].vR.y );
    D3DXMatrixRotationZ( &matRotateZ, -m_pObjects[0].vR.z );
    D3DXMatrixMultiply( &matTemp, &matRotateX, &matRotateY );
    D3DXMatrixMultiply( &matTemp, &matRotateZ, &matTemp );
    D3DXMatrixMultiply( &matWorld, &matTemp, &matWorld );
    m_pObjects[0].matLocal = matWorld;
}

// in Render()
for (WORD i = 0; i < dwNumberofObjects; i++)
{
    ...
    m_pd3dDevice->SetTransform(D3DTS_WORLD, &m_pObject[i].matLocal );
    ...
}
```

You can make life easy for yourself by storing matrices that contain axis information in each object structure. We're only storing the world matrix here, because the object itself isn't animated, so a model matrix isn't used. A very important thing to remember is that matrix multiplication is not commutative. That means  $[a] * [b] \neq [b] * [a]$ . The formula for transformation is

$$|W| = |M| * |T| * |X| * |Y| * |Z|$$

where M is the model's matrix, T is the translation matrix, and X, Y, and Z are the rotation matrices. The above piece of code translates the object into its place with `D3DXMatrixTranslation()`. Translation can best be described as linear change in position. This change can be represented by a delta vector [tx, ty, tz], where tx, or often dx, represents the change in the object's x position; ty or dy represents the change in its y position; and tz or dz represents its change in its z position. We don't have the source of `D3DXMatrixTranslation()`, but it could look like this:

```
inline VOID D3DXMatrixTranslation (D3DXMATRIX* m, FLOAT tx, FLOAT ty, FLOAT tz )
{
    D3DXMatrixIdentity(m );
    m._41 = tx; m._42 = ty; m._43 = tz;
}
=
1  0  0  0
0  1  0  0
0  0  1  0
tx ty tz 1
```

Using our F22 example from earlier, if the nose of the airplane is oriented along the object's local z-axis, then translating this airplane in the positive z direction by using tz will make the airplane move forward in the direction its nose is pointing.

Rotation is the next operation that is performed by our code piece. Rotation can be described as circular motion about an axis. The incremental angles used to rotate the object here represent rotation from the current orientation. That means by rotating 1 degree about the z-axis, you tell your object to rotate 1 degree about its z-axis, regardless of its current orientation and regardless of how you got the orientation. This is how the real world operates.

`D3DXMatrixRotationY()` rotates the objects about the y-axis, where fRads equals the amount you want to rotate about this axis. The source of this method could look like:

```
VOID D3DXMatrixRotationY( D3DXMATRIX* mat, FLOAT fRads )
{
    D3DXMatrixIdentity(mat);
    mat._11 = cosf( fRads );
    mat._13 = -sinf( fRads );
    mat._31 = sinf( fRads );
    mat._33 = cosf( fRads );
}
=
cosf fRads      0      -sinf fRads      0
0                  0          0                  0
sinf fRads      0      cosf fRads      0
0                  0          0                  0
```

D3DXMatrixRotationX( ) rotates the objects about the x-axis, where fRads equals the amount you want to rotate about this axis:

```
VOID D3DXMatrixRotationX( D3DXMATRIX* mat, FLOAT fRads )
{
    D3DXMatrixIdentity(mat);
    mat._22 = cosf( fRads );
    mat._23 = sinf( fRads );
    mat._32 = -sinf( fRads );
    mat._33 = cosf( fRads );
}

=
1      0          0          0
0      cos fRads   sin fRads  0
0      -sin fRads  cos fRads  0
0      0          0          0
```

D3DXMatrixRotationZ() rotates the objects about the z-axis, where fRads equals the amount you want to rotate about this axis:

```
VOID D3DXMatrixRotationZ( D3DXMATRIX* mat, FLOAT fRads )
{
    D3DXMatrixIdentity(mat);
    mat._11 = cosf( fRads );
    mat._12 = sinf( fRads );
    mat._21 = -sinf( fRads );
    mat._22 = cosf( fRads );

=
cosf fRads   sinf fRads   0      0
-sinf fRads   cos fRads   0      0
0          0          0      0
0          0          0      0
```

The prototype of D3DXMatrixMultiply() looks like `D3DXMATRIX * D3DXMatrixMultiply (D3DXMATRIX* pOut, CONST D3DXMATRIX* pM1, CONST D3DMATRIX* pM2)`. In other words: `pOut=pM1*pM2`. The return value for this function is the same value returned in the `pOut` parameter. In this way, the `D3DXMatrixMultiply()` function can be used as a parameter for another function. Matrix multiplication is the operation by which one matrix is transformed by another. A matrix multiplication stores the results of the sum of the products of matrix rows and columns.

a b c d    A B C D

e f g h \* E F G H =

i j k l    I J K L

m n o p    M N O P

a\*A+b\*E+c\*I+d\*M a\*B+b\*F+c\*J+d\*N a\*C+b\*G+c\*K+d\*O a\*D+b\*H+c\*L+d\*P

e\*A+f\*E+g\*I+h\*M e\*B+f\*F+g\*J+h\*N etc.

A matrix multiplication routine could look like this:

```
D3DXMATRIX* D3DXMatrixMultiply (D3DXMATRIX* pOut,
                                  CONST D3DXMATRIX* pM1,
                                  CONST D3DMATRIX* pM2)
{
    FLOAT pM[16];
    ZeroMemory( pM, sizeof(D3DXMATRIX) );
    for( WORD i=0; i<4; i++ )
        for( WORD j=0; j<4; j++ )
            for( WORD k=0; k<4; k++ )
                pM[4*i+j] += pM1[4*i+k] * pM2[4*k+j];
    memcpy( pOut, pM, sizeof(D3DXMATRIX) );
    return (pOut);
}
```

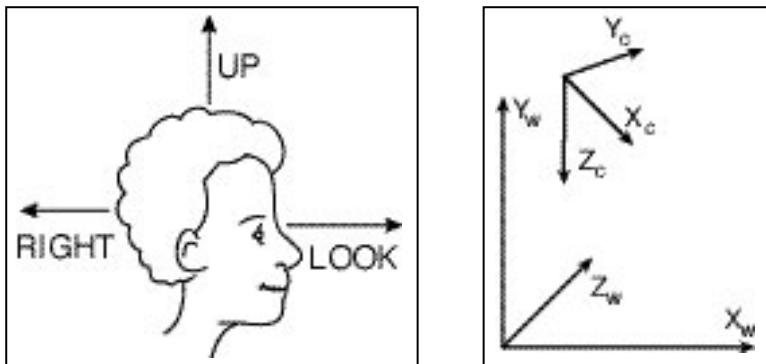
Once you've built the world transformation matrix, you need to call the `SetTransform()` method in `Render()`. You set the world transformation matrix by specifying the `D3DTS_WORLD` flag in the first parameter.

## THE VIEW MATRIX

The view matrix describes the position and the orientation of a viewer in a scene. This is normally your position and orientation, looking through the glass of your monitor into the scene. A lot of authors use the example of a camera through which you are looking into the scene.

To rotate and translate the viewer or camera in the scene, three vectors are needed. These are called the UP, RIGHT, and LOOK vectors.

They define a local set of axes for the camera and are set at the start of the application in the `RestoreDeviceObjects()` or in the `FrameMove()` method of the framework.



**Figure 6.9:** The UP, RIGHT, and LOOK vectors

The LOOK vector describes which way the camera is facing; it's the cameras local z-axis. The picture above shows it as  $z_c$ . To set the camera's direction so that it is facing into the screen, we would have to set the LOOK vector to D3DXVECTOR (0, 0, 1). The LOOK vector isn't enough to describe the orientation of the camera. The camera could stand upside down and the LOOK vector wouldn't reflect this change in orientation. The UP vector helps here; it points vertically up relative to the direction the camera points. It's like the cameras y-axis, and it's  $y_c$  on the picture above. So the UP vector is defined as D3DXVECTOR (0, 1, 0). If you turn the camera upside down, the UP vector will be D3DXVECTOR (0, -1, 0). We can generate a RIGHT vector from the LOOK and UP vectors by using the cross product of the two vectors, which is  $x_c$  on the picture above.

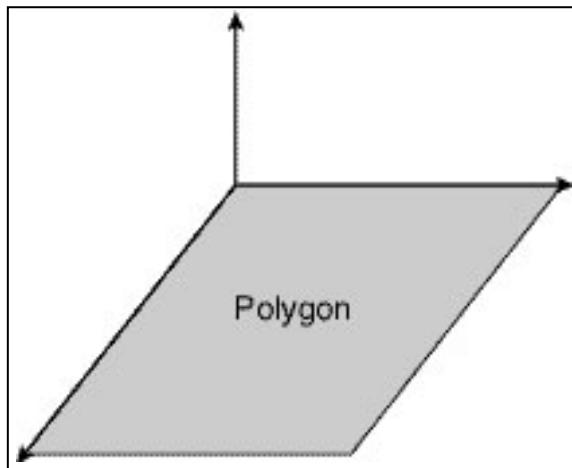
### NOTE

Taking the cross product of any two vectors forms a third vector perpendicular to the plane formed by the first two. The cross product is used to determine which way polygons are facing. It uses two of the polygon's edges to generate a normal. Thus, it can be used to generate a normal to any surface for which you have two vectors that lie within the surface. Unlike the dot product, the cross product is not commutative

$$a \times b = -(b \times a)$$

The magnitude of the cross product of a and b,  $\|axb\|$ , is given by  $[ab\ val]a[ab\ val]^* [ab\ val]b[ab\ val]^*\sin[\theta]$

The direction of the resultant vector is orthogonal to both a and b.



**Figure 6.10:** The perpendicular vector *normal*

Now, how is the camera rotated and positioned in a scene? I'd like to present two ways to do that:

- Camera rotation about a camera axis
- Camera rotation using a quaternion

#### CAMERA ROTATION ABOUT A CAMERA AXIS

To visualize camera rotation about a camera axis, imagine a player sitting in the cockpit of an F22. If the player pushes his foot pedals in his F22 to the left or right, the LOOK and the RIGHT vectors have to be rotated about the UP vector, or y-axis (yaw effect). If he pushes his flight stick to the right or left, the UP and RIGHT vectors have to be rotated around the LOOK vector, or z-axis (roll effect). If he pushes the flight stick forward and backward, we have to rotate the LOOK and UP vectors around the RIGHT vector, or x-axis (pitch effect).

There's one problem: When computers handle floating point numbers, little accumulation errors happen while doing all of this rotation math. After a few rotations, these rounding errors make the three vectors unperpendicular to each other.

This effect is often described as *drifting*. It's obviously important for the three vectors to stay at right angles to each other. Combating this problem requires keeping a matrix orthonormalized. One solution is called *base vector regeneration*. It must be performed before the vectors are rotated around one another. We'll use the following code to handle base vector regeneration:

```
static D3DXVECTOR vLook=D3DXVECTOR(0.0f,0.0f,-1.0);
static D3DXVECTOR vUp=D3DXVECTOR(0.0f,1.0f,0.0f);
static D3DXVECTOR vRight=D3DXVECTOR(1.0f,0.0f,0.0f);
D3DXVec3Normalize(&vLook, &vLook);
D3DXVec3Cross(&vRight, &vUp, &vLook); // Cross Produkt of the UP and LOOK Vector
```

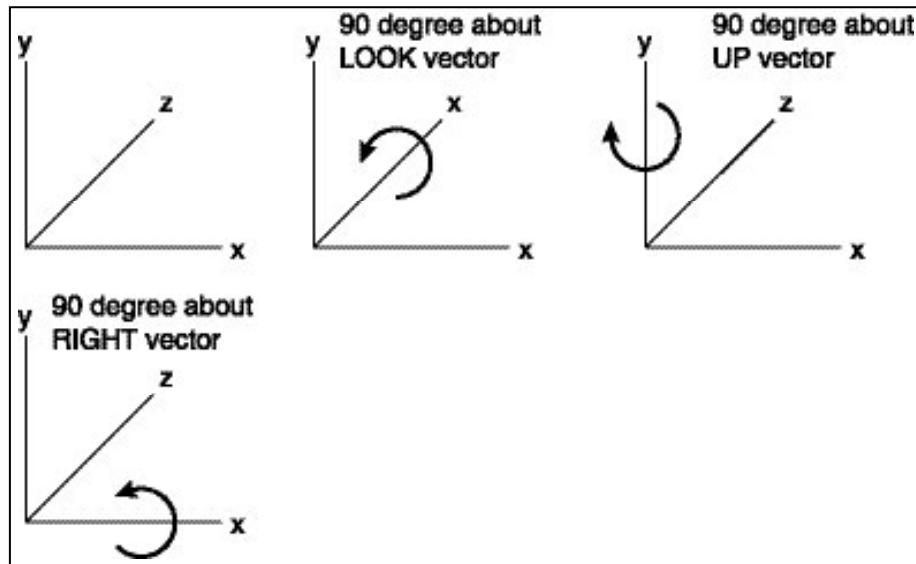


Figure 6.11: Rotation about an arbitrary axis

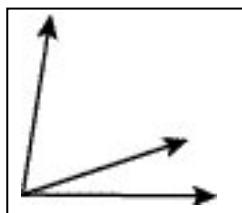


Figure 6.12: Unperpendicular vectors

```
D3DXVec3Normalize(&vRight, &vRight);
D3DXVec3Cross(&vUp, &vLook, &vRight); // Cross Produkt of the RIGHT and LOOK Vector
D3DXVec3Normalize(&vUp, &vUp);
```

First, we normalize the LOOK vector, so its length is 1. Vectors with a length of 1 are called *unit* or *normalized* vectors. How do we calculate a unit or normalized vector? To calculate a unit vector, divide the vector by its magnitude or length. You can calculate the magnitude of vectors by using the Pythagorean theorem:

$$x_+ y_+ z_+ = m_-$$

The length of the vector is retrieved by

$$||A|| = \sqrt{x_+ + y_+ + z_+}$$

It's the square root of the Pythagorean theorem. The magnitude of a vector has a special symbol in mathematics. It's a capital letter designated with two vertical bars:  $| |A| |$ .

We don't have the source of D3DXVec3Normalize(), but it could look like this:

```
D3DXVECTOR3* D3DXVec3Normalize(D3DXVECTOR3* v1, D3DXVECTOR3* v2)
{
    D3DXVECTOR3 tv1 = (D3DXVECTOR3)*v1;
    D3DXVECTOR3 tv2 = (D3DXVECTOR3)*v2;
    tv1 = tv2/(float) sqrt(tv2.x * tv2.x + tv2.y * tv2.y + tv2.z * tv2.z);
    v1 = &tv1;
    return (v1);
}
```

It divides the vector through its magnitude, which is retrieved by the square root of the Pythagorean theorem.

After normalizing the LOOK vector, we create the RIGHT vector by assigning it the cross product of UP and LOOK vectors and normalizing it. The UP vector is created out of the cross product of the LOOK and RIGHT vector and a normalization thereafter.

After that, we build the pitch, yaw, and roll matrices out of these vectors:

```
// Matrices for pitch, yaw and roll
D3DXMATRIX matPitch, matYaw, matRoll;
D3DXMatrixRotationAxis(&matPitch, &vRight, fPitch );
D3DXMatrixRotationAxis(&matYaw, &vUp, fYaw );
D3DXMatrixRotationAxis(&matRoll, &vLook, fRoll);
```

By multiplying, for example, the matYaw matrix with the LOOK and RIGHT vectors, we can rotate two vectors around the other vector.

```
// now multiply these vectors with the matrices we've just created.
// rotate the LOOK & RIGHT Vectors about the UP Vector
D3DXVec3TransformCoord(&vLook, &vLook, &matYaw);
D3DXVec3TransformCoord(&vRight, &vRight, &matYaw);
// rotate the LOOK & UP Vectors about the RIGHT Vector
D3DXVec3TransformCoord(&vLook, &vLook, &matPitch);
D3DXVec3TransformCoord(&vUp, &vUp, &matPitch);
// rotate the RIGHT & UP Vectors about the LOOK Vector
D3DXVec3TransformCoord(&vRight, &vRight, &matRoll);
D3DXVec3TransformCoord(&vUp, &vUp, &matRoll);
```

### NOTE

`sqrt()` is a mathematical function from the math library of Visual C/C++ provided by Microsoft. Other compilers should have a similar function.

Now we set the view matrix:

```
D3DXMATRIX view=matWorld;
D3DXMatrixIdentity( &view );
view._11 = vRight.x; view._12 = vUp.x; view._13 = vLook.x;
view._21 = vRight.y; view._22 = vUp.y; view._23 = vLook.y;
view._31 = vRight.z; view._32 = vUp.z; view._33 = vLook.z;
view._41 = - D3DXVec3Dot( &vPos, &vRight );
view._42 = - D3DXVec3Dot( &vPos, &vUp );
view._43 = - D3DXVec3Dot( &vPos, &vLook );
m_pd3dDevice->SetTransform(D3DTS_VIEW, &view);
=
    vx      vy      vz      0
    ux      uy      uz      0
    nx      ny      nz      0
-(u * c) -(v * c) -(n * c)   1
```

In this matrix, u, n, and v are the UP, RIGHT, and LOOK-direction vectors, and c is the camera's world space position. This matrix contains all the elements needed to translate and rotate vertices from world space to camera space.

The x, y, and z translation factors are computed by taking the negative of the dot product between the camera position and the u, v, and n vectors. They are negated, because the camera works opposite to objects in the 3-D world.

To rotate the vectors two about another, we change fPitch, fYaw, and fRoll variables like this:

```
fPitch=-0.3f * m_fTimeElapsed;
...
fPitch+=0.3f * m_fTimeElapsed;
...
fYaw=-0.3f * m_fTimeElapsed;
...
fYaw+=0.3f * m_fTimeElapsed;
...
fRoll=-0.3f * m_fTimeElapsed;
...
fRoll+=0.3f * m_fTimeElapsed;
```

To synchronize the different number of frame rates with the behavior of the objects, we have to use a variable with the elapsed time since the last frame. The camera moves forward and backward using the position variable:

```
vPos.x+=fspeed*vLook.x;  
vPos.y+=fspeed*vLook.y;  
vPos.z+=fspeed*vLook.z;  
vPos.x-=fspeed*vLook.x;  
vPos.y-=fspeed*vLook.y;  
vPos.z-=fspeed*vLook.z;
```

### CAMERA ROTATION WITH QUATERNIONS

Perhaps you've played one of the Tomb Raider titles (you remember, that girl with the guns?). It uses quaternion rotations to animate all of the camera movements. A lot of third-person games use a virtual camera placed at some distance behind or to the side of the player's character with quaternion. So why do they use it?

If you play around with the two sample programs provided on the CD-ROM, you'll see the difference. Quaternions are less computationally expensive, and the rotation will look smooth and predictable. So what are quaternions?

Although quaternions were invented back in 1843 by Sir William Rowan Hamilton as an extension to the complex numbers, it was not until 1985 that Ken Shoemake introduced them to the field of computer graphics on SIGGRAPH.

Quaternions extend the concept of rotation in three dimensions to rotation in four dimensions. They are useful for representing and processing 3-D rotations of points. They could also be used in

- Skeletal animation
- Inverse kinematics
- 3-D physics

You can use quaternions in a game to replace rotation matrices. They can describe any rotation around any axis in 3-D space. Why should we use quaternions instead of matrices?

- They take less space than matrices.
- Some operations, such as interpolation between quaternions, are also more visually pleasant.

We will concentrate on unit quaternions that can represent any three-dimensional rotation and that have the magnitude 1, like the free vectors we used until now. Each quaternion is comprised of four parts ( $x, y, z, w$ ) so that it can be plotted in 4-D space.

To represent a rotation of an angle  $\theta$  about an axis  $A(X_A, Y_A, Z_A)$  the quaternion  $q$  will be

$$q = (s \ X_A, s \ Y_A, s \ Z_A, c)$$

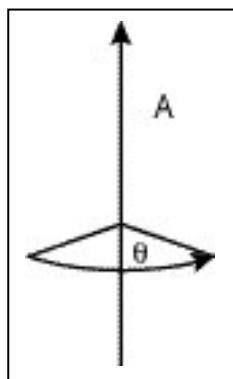
where

$$s = \sin(\theta / 2)$$

$$c = \cos(\theta / 2)$$

or

$$q = (\sin(\theta / 2)A, \cos(\theta / 2))$$



**Figure 6.13:** Representing a rotation about axis A

Given the angle and axis, the components of the quaternion are

$$\begin{aligned}x &= X_A \sin(\theta/2) \\y &= Y_A \sin(\theta/2) \\z &= Z_A \sin(\theta/2) \\w &= \cos(\theta/2)\end{aligned}$$

To rotate a vector around an axis  $u$  by an angle  $2\theta$  with the help of a quaternion, we would have to put the coordinates of a vector  $p = (p_x, p_y, p_z, p_w)$  into the components of a quaternion  $v$  and assume that we have a unit quaternion  $q = (\sin\theta u, \cos\theta)$ . Then

$$v' = q v q^{-1}$$

rotates  $v$  around the axis  $u$  by an angle of  $2\theta$ .

The result, a rotated vector  $v'$ , will always have a 0 scalar value for  $w$ , so you can omit it from your computations.

The resulting rotation in this example feels like you're rotating the camera around a joint that is as big as a pool ball. OK now, how does it work in code? Because neither Direct3D nor OpenGL supports quaternion directly, we have to convert from angles to quaternion and from quaternion to a matrix.

You have to take the following steps:

1. Translate the  $vTrans$  vector, which marks the position of the camera and the arbitrary point about the camera will rotate.
2. Build a quaternion from the angles provided by your D3DXVECTOR3 variables rotating the camera.

3. Build a matrix from the quaternion.
4. Concatenate that matrix with the position matrix.
5. Invert the position matrix to the view matrix.

If you look back at our rotation formula, these steps look more complicated than the formula. Step 2 multiplies the q and the v. Step 5 inverts the result of steps 2 and 4. Let's dive into the source:

```
D3DXVECTOR3 vTrans(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 vRot(0.0f, 0.0f, 0.0f);

// Process keyboard input
if (m_bKey[VK_HOME]) vTrans.z += 0.2f; // Move Forward
if (m_bKey[VK_END]) vTrans.z -= 0.2f; // Move Backward
if (m_bKey[VK_NUMPAD4]) vTrans.x -= 0.1f; // Slide Left
if (m_bKey[VK_NUMPAD6]) vTrans.x += 0.1f; // Slide Right
if (m_bKey[VK_NUMPAD8]) vTrans.y += 0.1f; // Slide Down
if (m_bKey[VK_NUMPAD2]) vTrans.y -= 0.1f; // Slide Up
if (m_bKey[VK_UP]) vRot.y += 0.1f; // Pitch Up
if (m_bKey[VK_DOWN]) vRot.y -= 0.1f; // Pitch Down
if (m_bKey[VK_LEFT]) vRot.x += 0.1f; // Turn Left
if (m_bKey[VK_RIGHT]) vRot.x -= 0.1f; // Turn Right
if (m_bKey['C']) vRot.z += 0.1f; // Rotate Left
if (m_bKey['X']) vRot.z -= 0.1f; // Rotate Right

// turn cfSmooth to 0.98f
const FLOAT cfSmooth = 0.9f;
// transform and rotation velocity
m_pvVelocity = m_pvVelocity * cfSmooth + vTrans;
m_pvAngularVelocity = m_pvAngularVelocity * cfSmooth + vRot;
// transform and rotation value
vTrans = m_pvVelocity * m_fTimeElapsed * m_fSpeed;
vRot = m_pvAngularVelocity* m_fTimeElapsed * m_fAngularSpeed;
// Update position and view matrices
D3DXMATRIX matT, matR;
D3DXQUATERNION qR;

D3DXMatrixTranslation(&matT, vTrans.x, vTrans.y, vTrans.z);           // step 1
D3DXMatrixMultiply(&m_matPosition, &matT, &m_matPosition);
D3DXQuaternionRotationYawPitchRoll(&qR, vRot.x, vRot.y, vRot.z);      // step 2
D3DXMatrixRotationQuaternion (&matR, &qR);                            // step 3
D3DXMatrixMultiply(&m_matPosition, &matR, &m_matPosition);            // step 4
D3DXMatrixInverse(&m_matView, NULL, &m_matPosition);                  // step 5
m_pd3dDevice->SetTransform( D3DTS_VIEW, &m_matView );
```

D3DXMatrixTranslation() builds a matrix, which translates an object by  $(x, y, z)$ . The multiplication of this matrix with the current position matrix is done by D3DXMatrixMultiply(). It translates the vector that describes the rotation point. D3DXQuaternionRotationYawPitchRoll() builds a quaternion with the given yaw, pitch, and roll. We build a matrix from the quaternion with a call to D3DXMatrixRotationQuaternion(). This matrix is multiplied by D3DXMatrixMultiply() with the position matrix.

The D3DMatrixInverse() method returns a so-called inverse matrix. Why do we need to invert a matrix?

We're using a quaternion to rotate the camera. These rotations happen about an arbitrary point, because the camera will be translated to other points where the rotation will happen. A rotation about an arbitrary point won't be linear if we don't invert the rotation matrix. Now, how does it work to invert the rotation matrix?

You get the inverse of a rotation matrix by negating the rotation angle. A rotation about the origin through an angle  $a$  will be canceled by a rotation through an angle  $-a$ . If a matrix has an inverse, this is written  $A^{-1}$ , and we have

$$A A^{-1} = A^{-1} A = I$$

Multiplying the matrix  $A$  with its inverse matrix results in the identity matrix  $I$ . Inverting a rotation matrix is used to map the origin of a rotation about an arbitrary point onto itself.

So to get a linear transformation, we have to invert the matrix here.

## THE PROJECTION MATRIX

The perspective projection converts the camera's viewing frustum (the pyramidlike shape that defines what the camera can see) into a cube space, as seen in Figure 6.7 (with a cube shape geometry, clipping is much easier). Objects close to the camera are enlarged greatly, while objects farther away are enlarged less. Here, parallel lines are generally not parallel after projection. This transformation applies perspective to a 3-D scene. It projects 3-D geometry into a form that can be viewed on a 2-D display.

**NOTE**

After this last transformation, the geometry must be clipped to the cube space and converted from homogenous coordinates to screen coordinates by dividing the x-, y-, and z-coordinates of each point by w. Direct3D performs these steps internally. So what are homogenous coordinates? Just think of a 3-by-3 matrix. As you've learned above, in a 4-by-4 matrix the first three elements, let's say i, j, and k, in the fourth row are needed to translate the object. With 3-by-3 matrices an object cannot be translated without changing its orientation. If you add some vector (representing translation) to the i, j, and k vectors, their orientation will also change. So we need a fourth dimension with the so-called homogenous coordinates. In this 4-D space, every point has a fourth component that measures distance along an imaginary fourth-dimensional axis called w. To convert a 4-D point into a 3-D point, you have to divide each component x, y, z, and w by the fourth component, w. So every multiplication of a point whose w component is equal to 1 represents that same point. For example, (4, 2, 8, 2) represents the same point as (2, 1, 4, 1).

To describe distance on the w-axis, we need another vector l. It points in the positive direction of the w-axis and its neutral magnitude is 1, like the other vectors.

In Direct3D, the points remain homogenous, even after being sent through the geometry pipeline. Only after clipping, when the geometry is ready to be displayed, are these points converted into Cartesian coordinates by dividing each component by w.

## LIGHTING

Integrated lighting is a critical requirement in a 3-D game, because it has such a dramatic impact on image quality. The human eye is more sensitive to changes in brightness than it is to changes in color.

When lighting is enabled as Direct3D rasterizes a scene in the final stage of rendering, Direct3D determines the color of each rendered pixel based on a combination of the current material color (and the texels in an associated texture map), the diffuse and specular colors of the vertex, if specified, as well as the color and intensity of light produced by light sources in the scene or the scene's ambient light level.

So there are at least three components, which are provided by you, that control the appearance of light: material, vertex color (optional), and the light source. Let's start with the material:

## MATERIAL

By default, no material is selected. When no material is selected, the Direct3D lighting engine is disabled.

D3DUtil\_InitMaterial() sets the RGBA values of the material. Color values of materials represent how much of a given light component is reflected by a surface that is rendered with that material. A material's properties include diffuse reflection, ambient reflection, light emission, and specular highlighting:

- **Diffuse reflection.** Defines how the polygon reflects diffuse lighting (any light that does not come from ambient light). This is described in terms of a color that represents the color best reflected by the polygon. Other colors are reflected less, in proportion to how different they are from the diffuse color.
- **Ambient reflection.** Defines how the polygon reflects ambient lighting. This is described in terms of a color that, as with diffuse reflection, represents the color best reflected by the polygon.
- **Light emission.** Makes the polygon appear to emit a certain color of light (this does not actually light up the world; it only changes the appearance of the polygon).
- **Specular highlighting.** Describes how shiny the polygon is.

A material whose color components are R: 1.0, G: 1.0, B: 1.0, A: 1.0 will reflect all the light that comes its way. Likewise, a material with R: 0.0, G: 1.0, B: 0.0, A: 1.0 will reflect all of the green light that is directed at it. SetMaterial() sets the material properties for the device.

```
D3DUtil_InitMaterial( mtr1, m_pObjects[0].r, m_pObjects[0].g, m_pObjects[0].b );
m_pd3dDevice->SetMaterial( &mtr1 );
```

## LIGHTING MODELS

After setting the material, we can set up the light. Color values for light sources represent the amount of a particular light component it emits. Lights don't use an alpha component, so you only need to think about the red, green, and blue components of the color. You can visualize the three components as the red, green, and blue lenses on a projection television. Each lens might be off (a 0.0 value in the appropriate member), it might be as bright as possible (a 1.0 value), or some level in-between. The colors coming from each lens combine to make the light's final color. A combination like R: 1.0, G: 1.0, B: 1.0 creates a white light, where R: 0.0, G: 0.0, B: 0.0 results in a light that doesn't emit light at all. You can make a light that emits only one component, resulting in a purely red, green, or blue light, or the light could use combinations to emit colors like yellow or purple. You can even set negative color component values to create a "dark light" that actually removes light from a scene. Or you might set the components to some value larger than 1.0 to create an extremely bright light. Direct3D employs three types of lights: point lights, spotlights, and directional lights.

### NOTE

It should be mentioned that this sort of lighting model is not based on much physical theory, but the result is still fairly good. Ultimately, the lighting model should simulate exactly the way photons interact with the physical environment, but because of the complexity of such a simulation, it is infeasible for real-time purposes.

You choose the type of light you want when you create a set of light properties. The illumination properties and the resulting computational overhead vary with each type of light source. The following types of light sources are supported since the advent of Direct3D 7.0:

- Point lights
- Spotlights
- Directional lights

**NOTE**

**With DirectX 7.0 the support of the parallel-point light type offered in previous releases of DirectX was abandoned.**

**TIP**

**You should avoid spotlights, because there are more realistic ways of creating spotlights than the default method supplied by Direct3D, such as texture blending. See the part on multitexturing.**

To set up an ambient light, you'll use the following call:

```
m_pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0x0c0c0c0c );
```

The color of the ambient light is a gray tone. An ambient light is effectively everywhere in a scene. It's a general level of light that fills an entire scene, regardless of the objects and their locations within that scene. Ambient light has no direction or position; there's only color and intensity. SetRenderState() sets the ambient light by specifying D3DRS\_AMBIENT as the dwRenderStateType parameter and the desired RGBA color as the dwRenderState parameter. Keep in mind that the color values of the material represent how much of a given light component is reflected by a surface. So the light properties are not the only properties that are responsible for the color of the object you will see.

Additionally, the example uses up to two directional lights. Although we might use directional lights and an ambient light to illuminate the objects in the scene, they are independent of one another. Directional light always has direction and color, and it is a factor for shading algorithms such as Gouraud shading. It is equivalent to using a point light source at an infinite distance.

An example should first check the capabilities of the driver. If it supports directional light, the light will be set by a call to the SetLight() method, which uses the D3DLIGHT8 structure.

```
typedef struct _D3DLIGHT8 {
    D3DLIGHTTYPE    dltType;
    D3DCOLORVALUE   dcvDiffuse;
    D3DCOLORVALUE   dcvSpecular;
    D3DCOLORVALUE   dcvAmbient;
    D3DVECTOR       dvPosition;
    D3DVECTOR       dvDirection;
    D3DVALUE        dvRange;
```

```

D3DVALUE      dvFalloff;
D3DVALUE      dvAttenuation0;
D3DVALUE      dvAttenuation1;
D3DVALUE      dvAttenuation2;
D3DVALUE      dvTheta;
D3DVALUE      dvPhi;
} D3DLIGHT8, *LPD3DLIGHT8;

```

The position, range, and attenuation properties are used to define a light's location in world space and how the light behaves over distance.

Shading is the general process of performing lighting computations and determining pixel colors. The default shading mode that is used in Direct3D and in newer graphics hardware is the Gouraud shading mode. It determines the lighting at each vertex of a triangle, and these lighting samples are interpolated over the surface of the triangle. You control the effect of Gouraud shading by providing normal vectors, as shown in Chapter 4 of this book.

If you'd like to set up a directional light, you will try to use the following code:

```

D3DUtil_InitLight( light, D3DLIGHT_DIRECTIONAL, 0.5f, -1.0f, 0.3f );
m_pd3dDevice->SetLight( 0, &light );
m_pd3dDevice->LightEnable( 0, TRUE );
m_pd3dDevice->SetRenderState( D3DRS_LIGHTING, TRUE );

```

## VERTEX COLOR (OPTIONAL)

Usually material colors are used in lighting formulas. But you can specify that material colors (emissive, ambient, diffuse, and specular) are to be overridden by diffuse or specular vertex colors.

Vertex color sources are selectable. The easiest way to set up your lighting engine to fetch vertex colors is to set the vertex colors in your vertex structure, as you've done before in the basic examples in Chapter 5.

```

// A structure for our custom vertex type
struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw; // The trans-
    formed position for the vertex
    DWORD color;        // The vertex
    color
};
// Our custom FVF, which describes our cus-
// tom vertex structure
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)

```

### NOTE

The use of vertex colors is possible since the advent of DirectX 6.0, which introduced the flexible vertex format. The old **D3DVERTEX**, **D3DLVERTEX**, and **D3DTLVERTEX** vertex types couldn't contain both color and normal information, so the Gouraud shading mode as the default shading mode hasn't used vertex color since then.

We don't use a vertex color in this example. Instead we use material colors, as shown above.

## DEPTH BUFFERING

A major problem that all game developers have to face when designing 3-D worlds is known as *overdraw*. Consider a 3-D scene where you are looking through a small window into a room beyond. Some of the walls and objects in the room will be visible through the window, and some will not.

Your graphic processors have no way of knowing which parts of the scene will be visible and which parts will be covered until they begin the rendering process. So you—as a game programmer—have the task of determining which are the visible surfaces and of finding a way to tell this to your graphic hardware. The whole process is called VSD (visible surface determination).

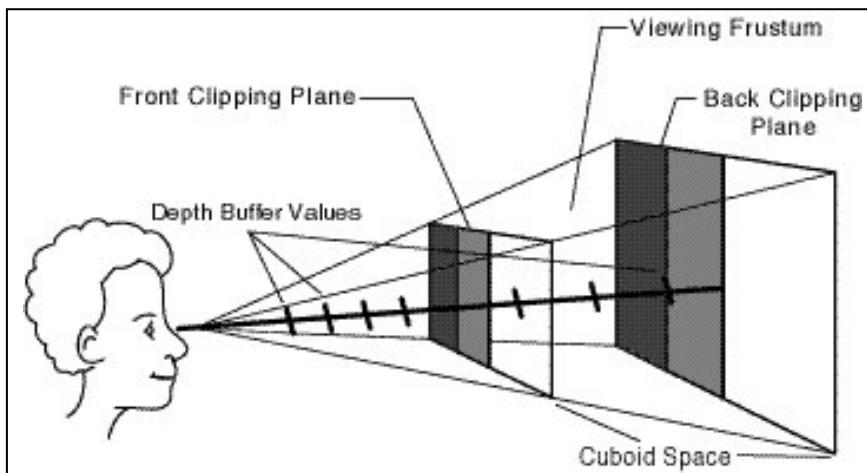
OK, you might argue: I will draw all the stuff from the objects in the back to the objects in the front of the room. That would be simple. Well . . . there's a drawback: overdraw. A lot of pixels will be overwritten in the frame or scene buffer by providing the objects from back to front, because every object will be drawn completely and then perhaps completely or partially overdrawn.

A measure of the amount of overdraw in a scene is called *depth complexity*, which represents the ratio of total pixels rendered to visible pixels. For example, if a scene has a depth complexity of 3, this means three times as many pixels were rendered as were actually visible on the screen. This also means that three times the fill rate would be needed to display the scene at a given frame rate as would be needed if there were no overdraw. Depending on the complexity of content, the depth complexity may vary from 1 to as high as 10 or more, but values around 2 or 3 are most common.

Overdraw is a major source of inefficiency in 3-D games.

Depth buffers play an important role in the task of VSD. Polygons closer to the camera must obscure polygons that are farther away. There are a number of solutions for this task; for example, as previously mentioned, you could draw all of the polygons back to front, which is slow and not supported by most hardware; binary space partition trees (one of the VSD techniques used in the Quake series and a lot of other 3-D shooters); octrees (often used in landscape engines); and so on. Direct3D supports the creation of a DirectDraw surface that stores depth information for every pixel on the display. Before displaying your virtual world, Direct3D clears every pixel on this depth buffer to the farthest possible depth value. Then when rasterizing, it determines the depth of each pixel on the polygon. If a pixel is closer to the camera than the one previously stored in the depth buffer, the pixel is displayed and the new depth value is stored in the depth buffer. This process will continue until all pixels are drawn.

The depth buffer is normally the same size and shape as the color buffer, where your scene is stored to be drawn on-screen later. So the depth buffer typically has between 16 and 32 bits for each pixel. The distance values of world space in a perspective-viewed environment are not proportional to the values, called *z-values*, of the depth buffer. This means that the distance between any adjacent depth values is not uniform; it's finer at the lower depth values.



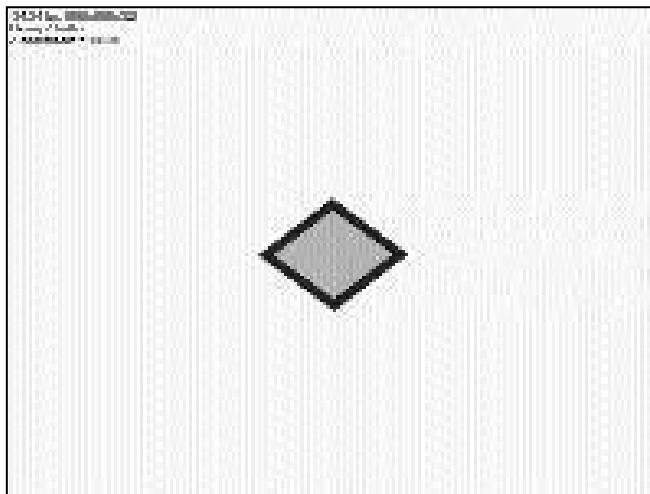
**Figure 6.14:** Depth buffer values

Losing depth buffer precision means a higher possibility of generating rendering errors. Because the depth precision depends on the precision of the color buffer, only 16-bit color resolutions can be affected. These problems raise their ugly heads normally only with the so-called z-buffer. To solve these problems, Direct3D supports two kinds of depth buffers: z-buffers and w-buffers. Think of the w-buffer as a higher-quality z-buffer with one drawback: it isn't supported in hardware as often as z-buffers.

The w-buffer reduces the problems that z-buffers exhibit with objects in the distance and has a constant performance for both near and far objects. The old DirectX 7.0 SDK provided a very instructive example of the difference between a z-buffer and a w-buffer in action; it showed the rendering errors that happen by losing depth buffer precision in 16-bit color mode. If you try that example, you have to choose a 16-bit color mode to see the errors happen:

#### NOTE

The z-buffer uses the projected z component of a vertex position scaled by projection matrix (range from 0 to 1). The w-buffer uses the z component of a camera space vertex position directly (range from near clip plane to far clip plane).



**Figure 6.15:** Rendering errors with a z-buffer

You can read a small text passage on the green square when depth buffering works correctly. Behind the green square is a blue square, which overlaps the text when the z-buffer with greater distance has lost its precision.

OK, now back to coding. The Common files framework makes using the depth buffer easy. You might tell the framework that you'd like to use it by specifying `m_bUseZBuffer = TRUE` in the constructor of your application class.

```
CMyD3DApplication::CMyD3DApplication()
{
    m_strWindowTitle = _T("First Steps to Animation");
    m_bUseDepthBuffer = TRUE;
```

The z-buffer is then set internally by the Common files framework for you. If you'd like to check the existence of a hardware w-buffer, you might use the following methods:

```
D3DPREMCAPS* pdpc = &m_pDeviceInfo->ddDeviceDesc.dpcTriCaps;
if ( 0L == ( pdpc->dwRasterCaps & D3DPRASTERCAPS_WBUFFER ) )
    m_pd3dDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_USEW ); // z-buffer
else
    m_pd3dDevice->SetRenderState( D3DRS_ZENABLE, TRUE ); // w-buffer
```

To see a depth buffer in action, switch it off and see how the polygons intersect.

To get a working depth buffer, you have to clear the depth buffer before every frame you'd like to render in the `Render()` method:

```
m_pd3DDevice->Clear( 0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER, 0x00000000, 1.0f, 0L );
```

This one clears the z- or w-buffer and the viewport with a black color.

Some cards (for example, Savage based cards) use w-buffering to “emulate” z-buffering. If you are sending textured and lighted vertices to the rendering pipeline, then make sure the rhw field is set correctly or the sorting will not work correctly.

There's a lot of talk about new depth buffering principles used by the ATI Radeon and PowerVR3/KYRO graphic chips in hardware. The ATI technology is called Hierarchical Z. It works by examining scene data before it is rendered to determine which pixels will be visible and which will not. Any pixels that will not be visible to the viewer are discarded and not rendered at all. This dramatically reduces overdraw and significantly boosts effective fill rate, with a corresponding improvement in performance. Another interesting approach: ATI compresses the transferred data to and from the z-buffer to reduce the memory bandwidth.

PowerVR is a display list renderer. That is, polygons are batched together into a display list before being processed by the 3-D rendering hardware. The scene is partitioned into small “tiles” or “regions,” each of which is rendered independently, because the resulting region is only a small subset of the whole scene.

So the whole scene doesn't have to be pumped through nearly the entire rendering pipeline, just to be skipped in one of the last rendering stages, at least partly, by a depth buffer. That's one of the most efficient hardware designs and should pay off in the production-per-unit costs.

All of this stuff doesn't need to be initialized or configured. It will work without code modification in Direct3D.

As mentioned above, `clear()` clears the whole viewport. You might also clear only parts of a viewport, which wouldn't make any sense for first-person shooters, but it could, for example, in adventure games. The method is declared as

```
HRESULT Clear(  
    DWORD Count,  
    CONST D3DRECT* pRects,  
    DWORD Flags,  
    D3DCOLOR Color,  
    float Z,  
    DWORD Stencil  
) ;
```

The method accepts one or more rectangles in pRects. The number of rectangles is stored in Count. The Flags variable indicates which kind of memory should be cleared. You might clear the z-buffer, render target, and stencil buffer. The Z variable holds the z-value and the Stencil variable holds the stencil value that this method stores in the z-buffer and stencil buffer. Neither is used here.

## DOWN TO THE CODE

First, I'll present the example of the camera rotated about the camera axis, and later on I'll show the modification that has to be made to rotate the camera with the help of a quaternion.

All examples use—as usual in this book—the Common files of the DirectX SDK. The application class in 1steps.cpp looks like:

```
class CMyD3DApplication : public CD3DApplication
{
    CD3DFont* m_pFont;
    LPDIRECT3DVERTEXBUFFER8 m_pVB; // Buffer to hold vertices
    DWORD m_dwSizeofVertices;
    LPDIRECT3DINDEXBUFFER8 m_pIB;
    DWORD m_dwSizeofIndices;

    FLOAT m_fStartTimeKey,      // Time reference for calculations
           m_fTimeElapsed;

    CUSTOMVERTEX m_pvObjectVertices[16];
    WORD         m_pwObjectIndices[30];
    Object       m_pObjects[2];

    BYTE m_bKey[256];          // keyboard state buffer

    HRESULT ConfirmDevice( D3DCAPS8*, DWORD, D3DFORMAT );

protected:
    HRESULT OneTimeSceneInit();
    HRESULT InitDeviceObjects();
    HRESULT RestoreDeviceObjects();
    HRESULT InvalidateDeviceObjects();
    HRESULT DeleteDeviceObjects();
    HRESULT Render();
    HRESULT FrameMove();
    HRESULT FinalCleanup();

public:
```

```
    LRESULT MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam );
    CMyD3DApplication();
};
```

The objects are described by vertices in `m_pvObjectVertices[16]` and by indices in `m_pwObjectIndices[30]`. There's an object structure called `object`. The fps-independent movement is guaranteed by the two time variables, which hold the start and the elapsed time between two frames. As usual, `ConfirmDevice()` is called as the first method, but it's not used here, because we won't need any special graphics card capabilities here. The other class methods will be mentioned in the following lines in the order they are called.

### ONETIMESCENEINIT()

The `OneTimeSceneInit()` loads geometry data, set-up calculated values, and so on. Basically, any one-time resource allocation should be performed here. It contains code to construct the two objects:

```
HRESULT CMyD3DApplication::OneTimeSceneInit()
{
    // Points and normals which make up a object geometry
    D3DXVECTOR3 p1 = D3DXVECTOR3( 0.00f, 0.00f, 0.50f );
    D3DXVECTOR3 p2 = D3DXVECTOR3( 0.50f, 0.00f,-0.50f );
    D3DXVECTOR3 p3 = D3DXVECTOR3( 0.15f, 0.15f,-0.35f );
    D3DXVECTOR3 p4 = D3DXVECTOR3(-0.15f, 0.15f,-0.35f );
    D3DXVECTOR3 p5 = D3DXVECTOR3( 0.15f,-0.15f,-0.35f );
    D3DXVECTOR3 p6 = D3DXVECTOR3(-0.15f,-0.15f,-0.35f );
    D3DXVECTOR3 p7 = D3DXVECTOR3(-0.50f, 0.00f,-0.50f );

    D3DXVECTOR3 n1 = D3DXVECTOR3( 0.2f, 1.0f, 0.0f );
    D3DXVec3Normalize(&n1, &n1);
    D3DXVECTOR3 n2 = D3DXVECTOR3( 0.1f, 1.0f, 0.0f );
    D3DXVec3Normalize(&n2, &n2);
    D3DXVECTOR3 n3 = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
    D3DXVec3Normalize(&n3, &n3);
    D3DXVECTOR3 n4 = D3DXVECTOR3(-0.1f, 1.0f, 0.0f );
    D3DXVec3Normalize(&n4, &n4);
    D3DXVECTOR3 n5 = D3DXVECTOR3(-0.2f, 1.0f, 0.0f );
    D3DXVec3Normalize(&n5, &n5);
    D3DXVECTOR3 n6 = D3DXVECTOR3(-0.4f, 0.0f, -1.0f );
    D3DXVec3Normalize(&n6, &n6);
    D3DXVECTOR3 n7 = D3DXVECTOR3(-0.2f, 0.0f, -1.0f );
    D3DXVec3Normalize(&n7, &n7);
    D3DXVECTOR3 n8 = D3DXVECTOR3( 0.2f, 0.0f, -1.0f );
```

```
D3DXVec3Normalize(&n8, &n8);
D3DXVECTOR3 n9 = D3DXVECTOR3( 0.4f, 0.0f, -1.0f );
D3DXVec3Normalize(&n9, &n9);

// vertices for the top
m_pvObjectVertices[0].p = p1;
m_pvObjectVertices[0].n = n1;
m_pvObjectVertices[0].tu = 0.0f;
m_pvObjectVertices[0].tv = 0.5f;
m_pvObjectVertices[1].p = p2;
m_pvObjectVertices[1].n = n2;
m_pvObjectVertices[1].tu = 0.5f;
m_pvObjectVertices[1].tv = 1.0f;
m_pvObjectVertices[2].p = p3;
m_pvObjectVertices[2].n = n3;
m_pvObjectVertices[2].tu = 0.425f;
m_pvObjectVertices[2].tv = 0.575f;
m_pvObjectVertices[3].p = p4;
m_pvObjectVertices[3].n = n4;
m_pvObjectVertices[3].tu = 0.425f;
m_pvObjectVertices[3].tv = 0.425f;
m_pvObjectVertices[4].p = p7;
m_pvObjectVertices[4].n = n5;
m_pvObjectVertices[4].tu = 0.5f;
m_pvObjectVertices[4].tv = 0.0f;

... all other vertices

// Vertex indices for the object
m_pwObjectIndices[ 0] = 0; m_pwObjectIndices[ 1] = 1; m_pwObjectIndices[ 2] =
2;
m_pwObjectIndices[ 3] = 0; m_pwObjectIndices[ 4] = 2; m_pwObjectIndices[ 5] =
3;
m_pwObjectIndices[ 6] = 0; m_pwObjectIndices[ 7] = 3; m_pwObjectIndices[ 8] =
4;
m_pwObjectIndices[ 9] = 5; m_pwObjectIndices[10] = 7; m_pwObjectIndices[11] =
6;
m_pwObjectIndices[12] = 5; m_pwObjectIndices[13] = 8; m_pwObjectIndices[14] =
7;
m_pwObjectIndices[15] = 5; m_pwObjectIndices[16] = 9; m_pwObjectIndices[17] =
8;
```

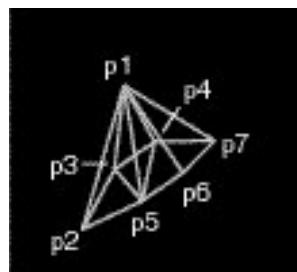
```
m_pwObjectIndices[18] = 10; m_pwObjectIndices[19] = 15; m_pwObjectIndices[20] =
11;
m_pwObjectIndices[21] = 11; m_pwObjectIndices[22] = 15; m_pwObjectIndices[23] =
12;
m_pwObjectIndices[24] = 12; m_pwObjectIndices[25] = 15; m_pwObjectIndices[26] =
14;
m_pwObjectIndices[27] = 12; m_pwObjectIndices[28] = 14; m_pwObjectIndices[29] =
13;

// yellow object
m_pObjects[0].vLoc      = D3DXVECTOR3(-1.0f, 0.0f, 0.0f);
m_pObjects[0].vR.x       = 0.0f;
m_pObjects[0].vR.y       = 0.0f;
m_pObjects[0].vR.z       = 0.0f;
m_pObjects[0].fR          = 1.0f;
m_pObjects[0].fG          = 0.92f;
m_pObjects[0].fB          = 0.0f;

// red object
... similar to the yellow object

return S_OK;
}
```

The sample project shows a simple object. Well . . . a cube would bore you. The wireframe model shows the polygons and points of the object.



**Figure 6.16:** Wireframe model  
of the object

On point 1 lies the `m_pvObjectVertices[0]` and `m_pvObjectVertices[5]`. At point 2, `m_pvObjectVertices[1]` and `m_pvObjectVertices[6]`. And at point 3 `m_pvObjectVertices[3]` and `m_pvObjectVertices[11]`, and so on.

Every point is declared as a vector with D3DXVECTOR3. For every face of the object, a normal is defined, so there are nine normals. These are not face normals.



**Figure 6.17:** Wireframe model  
with normals

The orientation of the normals is responsible for the shading with the Gouraud shading model. They are oriented in a manner that leads to the rounded look of the objects. The Gouraud shader shades the faces by interpolating between the defined normals. If you define normals that are not intersecting a vertex, you can deform the surface of the objects as shown above, because the Gouraud shading effect will interpolate the lighting effect between them. The object looks rounded with almost no edges. I've explained this effect in the section "Normals and Gouraud Shading" in Chapter 4 of this book.

The normal vectors are normalized with a call to

```
D3DXVec3Normalize(&n1, &n1);
```

The D3DXVec3Normalize() method divides the vector by its magnitude, which is retrieved by the square root of the Pythagorean theorem.

The last two variables of m\_pvObjectVertices are the texture coordinates. Most textures, like bitmaps, are a two-dimensional array of color values. The individual color values are called texture elements, or texels. Each texel has a unique address in the texture: its texel coordinate. Direct3D programs specify texel coordinates in terms of u, v values, much like 2-D Cartesian coordinates are specified in terms of x, y coordinates. The address can be thought of as a column and row number. Direct3D uses a uniform address range for all texels in all textures. Therefore, it uses a generic addressing scheme in which all texel addresses are in the range of 0.0 to 1.0, inclusive.

We're not using a texture here, so more on texture mapping in the chapter on multitexturing.

Now on to the next part of the OneTimeSceneInit() method:

```
// Vertex indices for the object
m_pwObjectIndices[ 0 ] = 0; m_pwObjectIndices[ 1 ] = 1; m_pwObjectIndices[ 2 ] = 2;
m_pwObjectIndices[ 3 ] = 0; m_pwObjectIndices[ 4 ] = 2; m_pwObjectIndices[ 5 ] = 3;
```

### NOTE

Direct3D maps texels in texture space directly to pixels in screen space.

```
m_pwObjectIndices[ 6] = 0; m_pwObjectIndices[ 7] = 3; m_pwObjectIndices[ 8] = 4;
m_pwObjectIndices[ 9] = 5; m_pwObjectIndices[10] = 7; m_pwObjectIndices[11] = 6;
m_pwObjectIndices[12] = 5; m_pwObjectIndices[13] = 8; m_pwObjectIndices[14] = 7;
m_pwObjectIndices[15] = 5; m_pwObjectIndices[16] = 9; m_pwObjectIndices[17] = 8;
m_pwObjectIndices[18] = 10; m_pwObjectIndices[19] = 15; m_pwObjectIndices[20] = 11;
m_pwObjectIndices[21] = 11; m_pwObjectIndices[22] = 15; m_pwObjectIndices[23] = 12;
m_pwObjectIndices[24] = 12; m_pwObjectIndices[25] = 15; m_pwObjectIndices[26] = 14;
m_pwObjectIndices[27] = 12; m_pwObjectIndices[28] = 14; m_pwObjectIndices[29] = 13;
```

This piece of code generates the indices for the D3DPT\_TRIANGLELIST call in DrawIndexedPrimitive(). Direct3D allows you to define your polygons in one of two ways: by defining their vertices or by defining indices into a list of vertices. The latter approach is often faster and more flexible, because it allows objects with multiple polygons to share vertex data.

Our object consists of only seven points, which are used by 15 vertices.

The two objects are defined with the help of the `m_pObjects` structure.

```
// yellow object
m_pObjects[0].vLoc    = D3DXVECTOR(-1.0f, 0.0f, 0.0f);
m_pObjects[0].vR.x    = 0.0f;
m_pObjects[0].vR.y   = 0.0f;
m_pObjects[0].vR.z   = 0.0f;
m_pObjects[0].fR     = 1.0f;
m_pObjects[0].fG     = 0.92f;
m_pObjects[0].fB     = 0.0f;

// red object
m_pObjects[1].vLoc    = D3DXVECTOR(1.0f, 0.0f, 0.0f);
m_pObjects[1].vR.x    = 0.0f;
m_pObjects[1].vR.y   = 0.0f;
m_pObjects[1].vR.z   = 0.0f;
m_pObjects[1].fR     = 1.0f;
m_pObjects[1].fG     = 0.0f;
m_pObjects[1].fB     = 0.27f;
```

To position the first object on the screen, a location has to be chosen. The yellow object should be located on the left and the red one on the right. The colors for the material properties are managed with the `fR`, `fG`, and `fB` variables. They are set later in `Render()` with a call to

```
// yellow object
// Set the color for the object
D3DUtil_InitMaterial( mtrl, m_pObjects[0].r, m_pObjects[0].g, m_pObjects[0].b );
```

```
m_pd3DDevice->SetMaterial( &mtrl );
```

## INITDEVICEOBJECTS()

The `InitDeviceObjects()` is used to initialize per-device objects, such as loading texture bits onto a device surface, setting matrices, or populating vertex buffers. We only use it here to call the `InitDeviceObjects()` method of the font class.

```
HRESULT CMyD3DApplication::InitDeviceObjects()
{
    m_pFont->InitDeviceObjects( m_pd3DDevice );
    return S_OK;
}
```

## RESTOREDEVICEOBJECTS()

The `RestoreDeviceObjects()` method is called after `InitDeviceObjects()` when the app starts up. When the user resizes the window of its application, this method is called after a call to `InvalidateDeviceObjects()`. After calling the `RestoreDeviceObjects()` font class method, we set here a material. Direct3D lights a scene by combining the current material color, diffuse and specular vertex color, if specified, and the color and intensity of the light sources. You must use materials to render a scene if you are letting Direct3D handle lighting.

```
HRESULT CMyD3DApplication::RestoreDeviceObjects()
{
    m_pFont->RestoreDeviceObjects();
    D3DMATERIAL8 mtrl;
    D3DUtil_InitMaterial( mtrl, 1.0f, 1.0f, 1.0f );
    m_pd3DDevice->SetMaterial( &mtrl );
    ...
}
```

The sample sets up an ambient light or two directional lights.

```
// Set up the lights
if( m_d3dCaps.VertexProcessingCaps & D3DVTCAPS_DIRECTIONALLIGHTS )
{
    D3DLIGHT8 light;
    D3DUtil_InitLight( light, D3DLIGHT_DIRECTIONAL, 0.5f, -1.0f, 0.3f );
    light.Ambient.r = 0.1f;
    light.Ambient.g = 0.1f;
    light.Ambient.b = 0.1f;
    m_pd3DDevice->SetLight( 0, &light );
    m_pd3DDevice->LightEnable( 0, TRUE );
    D3DUtil_InitLight( light, D3DLIGHT_DIRECTIONAL, 0.5f, 1.0f, 1.0f );
```

```
    light.Ambient.r = 0.2f;
    light.Ambient.g = 0.2f;
    light.Ambient.b = 0.2f;
    m_pd3dDevice->SetLight( 1, &light );
    m_pd3dDevice->LightEnable( 1, TRUE );
    m_pd3dDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
}
else
{
    m_pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0x0c0c0c0c );
}
```

This piece of code checks the capabilities of the driver. If the driver supports directional light, the two directional lights will be set by a call to the `SetLight()` method, which uses the `D3DLIGHT8`. The `D3DUtil_InitLight()` method in `d3dutil.cpp` sets a few default values into this structure.

```
VOID D3DUtil_InitLight( D3DLIGHT8& light, D3DLIGHTTYPE ltType,
                        FLOAT x, FLOAT y, FLOAT z )
{
    ZeroMemory( &light, sizeof(D3DLIGHT8) );
    light.Type      = ltType;
    light.Diffuse.r = 1.0f;
    light.Diffuse.g = 1.0f;
    light.Diffuse.b = 1.0f;
    D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &D3DXVECTOR3(x, y, z) );
    light.Position.x = x;
    light.Position.y = y;
    light.Position.z = z;
    light.Range     = 1000.0f;
}
```

Only the light position is set explicitly for the first light. The light position is described using three float values. These are x-, y-, and z-coordinates in world space. The first light is located under the objects and the second light is located above them; it's a bit darker.

Directional lights don't use range and attenuation variables. A light's range property determines the distance in world space at which meshes in a scene no longer receive light. So the `dvRange` floating point value represents the light's maximum range. The attenuation variables control how a light's intensity decreases toward the maximum distance, specified by the range property. There are three attenuation values: controlling a light's constant, linear, and quadratic attenuation with floating point variables. Many applications set the `dvAttenuation1` member to `1.0f` and the others to `0.0f`.

The vertex and index buffer is filled with vertices/indices in a similar way. I've described it in the Basic3 example in Chapter 5.

```

// fill the vertex buffer
m_dwSizeofVertices = sizeof(m_pvObjectVertices);
// Create the vertex buffer
if( FAILED( m_pd3dDevice->CreateVertexBuffer( m_dwSizeofVertices,
                                              0, D3DFVF_CUSTOMVERTEX,
                                              D3DPPOOL_MANAGED, &m_pVB ) ) )
{
    return E_FAIL;
}

VOID* pVertices;
if( FAILED( m_pVB->Lock( 0, m_dwSizeofVertices, (BYTE**)&pVertices, 0 ) ) )
{
    return E_FAIL;
}
memcpy( pVertices, m_pvObjectVertices, m_dwSizeofVertices );
m_pVB->Unlock();
// fill the Index buffer
m_dwSizeofIndices = sizeof(m_pwObjectIndices);
// Create the index buffer
if( FAILED( m_pd3dDevice->CreateIndexBuffer( m_dwSizeofIndices,
                                              0, D3DFMT_INDEX16,
                                              D3DPPOOL_MANAGED, &m_pIB ) ) )
{
    return E_FAIL;
}

VOID* pIndices;
if( FAILED( m_pIB->Lock( 0, m_dwSizeofIndices, (BYTE**)&pIndices, 0 ) ) )
{
    return E_FAIL;
}
memcpy( pIndices, m_pwObjectIndices, m_dwSizeofIndices );
m_pIB->Unlock();

```

Besides setting the material and lights and filling the index and vertex buffers, the `RestoreDeviceObjects()` method sets the projection matrix with the aspect ratio of the back buffer.

```

// Set the projection matrix
D3DXMATRIX matProj;
FLOAT fAspect = m_d3dsdBackBuffer.Width / (FLOAT)m_d3dsdBackBuffer.Height;
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, fAspect, 1.0f, 500.0f );
m_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
=
w   0     0     0
0   h     0     0
0   0     0     1
0   0     -QZN  0

```

This code sets up a projection matrix, taking the aspect ratio, front (-Q\*Zn) or near plane, and back or far clipping planes and the field of view with fFOV in radians. Note that the projection matrix is normalized for element [3][4] to be 1.0. This is performed so that w-based range fog will work correctly.

## FrameMove()

The FrameMove() method handles most of the keyboard input and the matrix stuff. I've implemented a simple but useful keyboard interface in the MsgProc() method.

```
BYTE m_bKey[256]; // keyboard state buffer
...
if (m_bKey['J']) m_pObjects[0].vR.z -= 1.0f * m_fTimeElapsed;
...
HRESULT CMyD3DApplication::MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam )
{
    // Record key presses
    if( WM_KEYDOWN == uMsg )
    {
        m_bKey[wParam] = 1;
    }
    // Perform commands when keys are released
    if( WM_KEYUP == uMsg )
    {
        m_bKey[wParam] = 0;
    }
    return CD3DApplication::MsgProc( hWnd, uMsg, wParam, lParam );
}
```

### TIP

**The screen space is a frame of reference in which coordinates are related directly to 2-D locations in the frame buffer to be displayed on a monitor or other viewing device. Projection space coordinates are converted to screen space coordinates, using a transformation matrix created from the viewport parameters. After that, the backface culling will happen (to omit drawing the backface is called backface culling). To cull the backface, we need a way to determine the visibility. One of the simplest ways is the following: A front face is one in which vertices are defined in clockwise order. So Direct3D only draws the faces with vertices in clockwise order by default. To modify backface culling, use**

```
// no backface culling
m_pd3dDevice->SetRenderState(D3DRS_CULLMODE,
D3DCULL_NONE);
```

A global byte array is able to store the values of the keys that are pressed by the user. The value of the key is provided by the wParam parameter of the MsgProc() method. That's really simple. You might use DirectInput for faster and more reliable keyboard handling.

FrameMove() handles all the rotations and translations of the objects and the camera. It uses timing code to ensure that all the objects and the camera move or rotate in the same speed at every possible frames per second.

```
// timing code:  
// the object should move/rotate in the same speed  
// at every possible fps  
const cTimeScale = 5;  
// calculate elapsed time  
m_fTimeElapsed=(fTimeKey-m_fStartTimeKey)* cTimeScale;  
// store last time  
m_fStartTimeKey=fTimeKey;
```

To calculate the elapsed time, you have to subtract the m\_fStartTimeKey from fTimeKey.

To rotate the yellow object about its z- and y-axes, we have to change the variables vR.z and vR.y in the m\_pObject structure.

```
if (m_bKey['J']) m_pObjects[0].vR.z -= 1.0f * m_fTimeElapsed;  
if (m_bKey['L']) m_pObjects[0].vR.z += 1.0f * m_fTimeElapsed;  
if (m_bKey['I']) m_pObjects[0].vR.y -= 1.0f * m_fTimeElapsed;  
if (m_bKey['K']) m_pObjects[0].vR.y += 1.0f * m_fTimeElapsed;
```

They are used in the following translate and rotate matrix methods.

```
D3DXMATRIX matWorld;  
D3DXMatrixTranslation(&matWorld,  
                      m_pObjects[0].vLoc.x,  
                      m_pObjects[0].vLoc.y,  
                      m_pObjects[0].vLoc.z);  
  
D3DXMATRIX matTemp, matRotateX, matRotateY, matRotateZ;  
D3DXMatrixRotationY( &matRotateY, -m_pObjects[0].vR.x );  
D3DXMatrixRotationX( &matRotateX, -m_pObjects[0].vR.y );  
D3DXMatrixRotationZ( &matRotateZ, -m_pObjects[0].vR.z );  
D3DXMatrixMultiply( &matTemp, &matRotateX, &matRotateY );  
D3DXMatrixMultiply( &matTemp, &matRotateZ, &matTemp );  
D3DXMatrixMultiply( &matWorld, &matTemp, &matWorld );  
m_pObjects[0].matLocal = matWorld;
```

As described above, the method `D3DXMatrixTranslation()` would translate the yellow object into its place and `D3DXMatrixRotationX()` and `D3DXMatrixRotationZ()` would rotate it around the x-axis and z-axis. We won't use `D3DXMatrixRotationY()` here. Finally, the position of the yellow object in the world matrix will be stored in the `m_pObjects` structure.

The same functionality lies behind the code for the red object.

```
if (m_bKey['D'])      m_pObjects[1].vR.z -= 1.0f * m_fTimeElapsed;
if (m_bKey['A'])      m_pObjects[1].vR.z += 1.0f * m_fTimeElapsed;
if (m_bKey['S'])      m_pObjects[1].vR.y -= 1.0f * m_fTimeElapsed;
if (m_bKey['W'])      m_pObjects[1].vR.y += 1.0f * m_fTimeElapsed;
D3DXMatrixTranslation(&matWorld,
                      m_pObjects[1].vLoc.x,
                      m_pObjects[0].vLoc.y,
                      m_pObjects[0].vLoc.z);
D3DXMatrixRotationY( &matRotateY, -m_pObjects[1].vR.x );
D3DXMatrixRotationX( &matRotateX, -m_pObjects[1].vR.y );
D3DXMatrixRotationZ( &matRotateZ, -m_pObjects[1].vR.z );
D3DXMatrixMultiply( &matTemp, &matRotateX, &matRotateY );
D3DXMatrixMultiply( &matTemp, &matRotateZ, &matTemp );
D3DXMatrixMultiply( &matWorld, &matTemp, &matWorld );
m_pObjects[1].matLocal = matWorld;
```

The only differences are the use of other keys and the storage of the variables in another variable of the object structure.

After translating and rotating the two objects, the camera has to be placed and pointed in the right direction.

In our first example, this is done using three vectors that can be rotated about each other. The `vLook`, `vUp`, `vRight`, and `vPos` vectors hold the position and the LOOK, UP, and RIGHT vectors of the camera.

```
static D3DXVECTOR3 vLook=D3DXVECTOR3(0.0f,0.0f,1.0);
static D3DXVECTOR3 vUp=D3DXVECTOR3(0.0f,1.0f,0.0f);
static D3DXVECTOR3 vRight=D3DXVECTOR3(1.0f,0.0f,0.0f);
static D3DXVECTOR3 vPos=D3DXVECTOR3(0.0f,0.0f,-5.0f);
FLOAT fPitch,fYaw,fRoll;
fPitch = fYaw = fRoll = 0.0f;

FLOAT fspeed= 1.0f * m_fTimeElapsed;
```

The LOOK vector points in the direction of the positive z-axis, the UP vector points into the direction of the positive y-axis, and the RIGHT vector points in the direction of the positive x-axis. The variables

fPitch, fYaw, and fRoll are responsible for the orientation of the camera. The camera is moved backward and forward with vPos, whereas fspeed holds the backward and forward speed of it.

```

if (m_bKey[VK_UP])           fPitch=-0.3f * m_fTimeElapsed;      // fPitch
if (m_bKey[VK_DOWN])         fPitch=+0.3f * m_fTimeElapsed;
if (m_bKey['C'])             fYaw=+0.3f * m_fTimeElapsed;        // fYaw
if (m_bKey['X'])             fYaw=-0.3f * m_fTimeElapsed;
if (m_bKey[VK_LEFT])          fRoll=-0.3f * m_fTimeElapsed;       // fRoll
if (m_bKey[VK_RIGHT])         fRoll=+0.3f * m_fTimeElapsed;

// camera forward
if (m_bKey[VK_HOME]) // Key HOME
{
    vPos.x+=fspeed*vLook.x;
    vPos.y+=fspeed*vLook.y;
    vPos.z+=fspeed*vLook.z;
}

// camera back
if (m_bKey[VK_END]) // Key END
{
    vPos.x-=fspeed*vLook.x;
    vPos.y-=fspeed*vLook.y;
    vPos.z-=fspeed*vLook.z;
}

```

The three orientation vectors are normalized with base vector regeneration by normalizing the LOOK vector, building a perpendicular vector out of the UP and LOOK vectors, normalizing the RIGHT vector, and building the perpendicular vector of the LOOK and RIGHT vectors, the UP vector. Then the UP vector is normalized.

Normalization produces a vector with a magnitude of 1. The cross product method produces a vector that is perpendicular to the two vectors provided as variables.

```

// base vector regeneration
D3DXVec3Normalize(&vLook, &vLook);
D3DXVec3Cross(&vRight, &vUp, &vLook); // Cross Produkt of the UP and LOOK Vector
D3DXVec3Normalize(&vRight, &vRight);
D3DXVec3Cross(&vUp, &vLook, &vRight); // Cross Produkt of the RIGHT and LOOK Vector
D3DXVec3Normalize(&vUp, &vUp);

```

The rotation matrices are built with D3DXMatrixRotationAxis() and executed with D3DXVec3TransformCoord().

```
// Matrices for pitch, yaw and roll
D3DXMATRIX matPitch, matYaw, matRoll;
D3DXMatrixRotationAxis(&matPitch, &vRight, fPitch );
D3DXMatrixRotationAxis(&matYaw, &vUp, fYaw );
D3DXMatrixRotationAxis(&matRoll, &vLook, fRoll);
// rotate the LOOK & RIGHT Vectors about the UP Vector
D3DXVec3TransformCoord(&vLook, &vLook, &matYaw);
D3DXVec3TransformCoord(&vRight, &vRight, &matYaw);
// rotate the LOOK & UP Vectors about the RIGHT Vector
D3DXVec3TransformCoord(&vLook, &vLook, &matPitch);
D3DXVec3TransformCoord(&vUp, &vUp, &matPitch);
// rotate the RIGHT & UP Vectors about the LOOK Vector
D3DXVec3TransformCoord(&vRight, &vRight, &matRoll);
D3DXVec3TransformCoord(&vUp, &vUp, &matRoll);
D3DXMATRIX view=matWorld;
D3DXMatrixIdentity( &view );
view._11 = vRight.x; view._12 = vUp.x; view._13 = vLook.x;
view._21 = vRight.y; view._22 = vUp.y; view._23 = vLook.y;
view._31 = vRight.z; view._32 = vUp.z; view._33 = vLook.z;
view._41 = - D3DXVec3Dot( &vPos, &vRight );
view._42 = - D3DXVec3Dot( &vPos, &vUp );
view._43 = - D3DXVec3Dot( &vPos, &vLook );
m_pd3dDevice->SetTransform(D3DTS_VIEW, &view);
```

## RENDER()

The Render() method is called once per frame and is the entry point for 3-D rendering. It clears the viewport and renders the two objects with proper material.

```
HRESULT CMyD3DApplication::Render()
{
    D3DMATERIAL8 mtrl;
    // Clear the viewport | z-buffer
    m_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
        0x00000000, 1.0f, 0L );
    // Begin the scene
    if( SUCCEEDED( m_pd3dDevice->BeginScene() ) )
    {
        // yellow object
        // Set the color for the object
        D3DUtil_InitMaterial( mtrl, m_pObjects[0].fR, m_pObjects[0].fG,
            m_pObjects[0].fB );
```

```
m_pd3dDevice->SetMaterial( &mtrl );  
  
// Apply the object's local matrix  
m_pd3dDevice->SetTransform(D3DTS_WORLD, &m_pObjects[0].matLocal );  
  
m_pd3dDevice->SetStreamSource( 0, m_pVB, sizeof(CUSTOMVERTEX) );  
m_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );  
m_pd3dDevice->SetIndices( m_pIB, 0 );  
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,  
                                    0,  
                                    16, // number of vertices  
                                    0,  
                                    10); // number of primitives  
...  
...
```

The used z- or w-buffer is cleared with the `Clear()` method, one of the routines of the Common files framework.

As usual, the `Render()` method uses the `BeginScene()`/`EndScene()` pair. See the basic example above to read a deeper explanation of these methods. The call to set the world transformation for the yellow object is also done in `Render()` with

```
m_pd3dDevice->SetTransform(D3DTS_WORLD, &m_pObjects[0].matLocal );
```

The parameters of `DrawIndexedPrimitive()` has changed dramatically since Direct3D 7.0 as we've discussed in the Basic3 example in Chapter 5. Keep in mind that the final parameter that `DrawIndexedPrimitive()` accepts is the number of primitives to render. This number depends on the primitive count and the primitive type. The code sample above uses triangles, so the number of primitives to render is the number of indices divided by three.

As stated above, using the right primitive type depends on the data you would like to use. When the structure of the data falls naturally into strips and fans, this is the most appropriate choice, because they minimize the data sent to the driver.

If you have to decompose meshes into strips and fans, a large number of `DrawPrimitive` calls has to be made because of the large number of pieces. The most efficient method is usually to use a triangle list with indexed primitives.

## INVALIDATEDEVICEOBJECTS()

`InvalidateDeviceObjects()` is called when the user resizes the window and in other cases. After that, `d3dapp.cpp` gives the next call to `RestoreDeviceObjects()`. So in this case, these two methods are a functional pair. Because we've built the index and vertex buffer in `RestoreDeviceObjects()`, we have to destroy it here. The `InvalidateDeviceObjects()` method of the font class is also called.

```
HRESULT CMyD3DApplication::InvalidateDeviceObjects()
{
    m_pFont->InvalidateDeviceObjects();
    SAFE_RELEASE( m_pVB );
    SAFE_RELEASE( m_pIB );
    return S_OK;
}
```

### **DELETEDeviceOBJECTS()**

The method `DeleteDeviceObjects()` is called when the app exits or the device is being changed. Because we don't need to free any device-specific resource allocations, there's only a call to `DeleteDeviceObjects()` of the font class.

```
HRESULT CMyD3DApplication::DeleteDeviceObjects()
{
    m_pFont->DeleteDeviceObjects();
    return S_OK;
}
```

### **FINALCLEANUP()**

`FinalCleanup()` as the counterpart of `OneTimeSceneInit()` destroys any per-application objects. It's the last method called by the framework. It's empty, because we have not acquired any system resources in `OneTimeSceneInit()`.

## **NEXT STEPS TO ANIMATION**

As announced above, I'd like to enhance the sample by using a quaternion to rotate the camera. I'll concentrate on the source pieces, which have to be changed. You'll find the source to this example in the `1steps2` directory.

### **RESTOREDEVICEOBJECTS()**

At first, we have to set the view matrix and the position of the camera in `RestoreDeviceObjects()` with

```
D3DUtil_SetTranslateMatrix(m_matView, 0.0f, 0.0f, -20.0f);
D3DUtil_SetTranslateMatrix(m_matPosition, 0.0f, 0.0f, -5.0f);
```

The camera is positioned behind the two objects. It looks in the negative z direction.

I expanded the input control so that you might be able to slide the camera on its x- and y-axes.

```
if (m_bKey[VK_NUMPAD4])      vTrans.x -= 0.1f; // Slide Left
if (m_bKey[VK_NUMPAD6])      vTrans.x += 0.1f; // Slide Right
```

```
if (m_bKey[VK_NUMPAD8])      vTrans.y += 0.1f; // Slide Down  
if (m_bKey[VK_NUMPAD2])      vTrans.y -= 0.1f; // Slide Up
```

## FRAMEMOVE()

Most of the camera orientation happens in `FrameMove()`. What do we need to rotate a camera? We need a point that is described by a vector and translated on arbitrary places. And we need a rotation “description.”

```
D3DXMatrixTranslation (&matT, vTrans.x, vTrans.y, vTrans.z);           // step 1  
D3DXMatrixMultiply (&m_matPosition, &matT, &m_matPosition);  
D3DXQuaternionRotationYawPitchRoll (&qR, vRot.x, vRot.y, vRot.z);       // step 2  
D3DXMatrixRotationQuaternion (&matR, &qR);                           // step 3  
D3DXMatrixMultiply (&m_matPosition, &matR, &m_matPosition);           // step 4  
D3DXMatrixInverse (&m_matView, NULL, &m_matPosition);                  // step 5
```

The translation of the arbitrary rotation point is done by `D3DXMatrixTranslation()`, which builds a matrix that translates by (x, y, z), and `D3DXMatrixMultiply()`, which translates the vector by multiplying the current position matrix with the new position matrix.

The quaternion is provided from `D3DXQuaternionRotationYawPitchRoll()`, which receives a vector that describes the rotation.

We built a matrix from the quaternion with a call to `D3DXMatrixRotationQuaternion()`.

`D3DXMatrixMultiply()` rotates the camera around the already translated position. To get a linear rotation about the arbitrary point, we have to invert the resulting matrix with `D3DMatrixInverse()`.

## MORE ENHANCEMENTS

There are a few things you can do to enhance this example even more. You might rotate the objects with quaternions, which will result in a smoother rotation of the objects. Another popular use of quaternions is to interpolate between two quaternion orientations to achieve smoother orientation or animation.

## QUIZ

Q: What's the purpose of the pluggable software device?

A: If a feature is not supported in HAL, the pluggable software device can take control and emulate the hardware. It's programmed and supported by the hardware manufacturer.

Q: What's the purpose of the reference rasterizer?

A: To test features that your card doesn't support.

Q: What are the most important COM features for game programmers?

A: COM interfaces can never change

Language independence  
Only access to methods, not data

Q: What is a v-table?

A: A virtual function table references the methods implemented in a COM object. You can't access COM methods directly.

Q: How would you define a location of a vertex in 3-D space?

A: By specifying a vector from the origin to the point where the vertex resides.

Q: How do you define the orientation of an object in 3-D space?

A: By using three vectors: the LOOK, UP, and POSITION vectors.

Q: Why do we have to specify face vertices in clockwise order?

A: Direct3D omits drawing back faces of objects to gain more speed. By default, it only draws faces of objects where the vertices are oriented in clockwise order. This is called backface culling. So you have to recommend the proper ordering of your vertices.

Q: What's the purpose of Gouraud shading?

A: It shades light per vertex. So the effect of lighting depends on the position and direction of the vertex normals. You may, for example, round up objects by changing the position of the vertex normals.

Q: How are textures mapped on a face?

A: By using a uniform address range for all texels in a texture. The upper left corner is (0.0f, 0.0f) and the bottom right corner is (1.0f, 1.0f).

Q: What are the Common files?

A: They encapsulate the basic functionality to start a Direct3D game. It's proven code and well tested.

Q: What's the purpose of the `OneTimeSceneInit()`/`InitDeviceObjects()` methods?

A: `OneTimeSceneInit()` performs all of the things that have to be done to load the geometry data of a scene. It's called only once. `InitDeviceObjects()` initializes all per-device objects, such as loading texture bits onto a device surface, setting matrices, populating vertex buffers, and setting render states. It's called when the app starts or the device is being changed.

Q: What's the difference between `DeleteDeviceObjects()` and `FinalCleanup()`?

A: `DeleteDeviceObjects()` is called when the app exits or when the device changes. `FinalCleanup()` is the last framework call in your app before it exits. It destroys allocated memory, for example, for textures and deletes file objects.

Q: What textures could be managed by `D3DUtil_CreateTexture()`?

A: It loads bitmaps from .bmp, .jpg, \*.png, \*.ppm, \*.dds and .tga files.

Q: What is a viewport?

A: In order for Direct3D to know what frustum to clip to, we need the viewport dimensions. These dimensions are used to clip to the cube shape of the viewing frustum, of the perspective projection.

Q: What's the purpose of the BeginScene()/EndScene pair?

A: BeginScene() causes the system to check its internal data structures, the availability and validity of rendering surfaces, and it sets the internal flag to signal that a scene is in progress. EndScene() clears the internal flag that indicates that a scene is in progress, flushes the cached data, and makes sure the rendering surfaces are OK.

Q: What happens if you set a texture?

A: A call to SetTexture() instructs the Direct3D device to apply the texture to all primitives that are rendered from that time until the current texture is changed again.

Q: What is a triangle strip?

A: It creates a series of connected triangles.

Q: What is a vertex flag?

A: It tells Direct3D something about the used vertex structure in which the vertex information is stored. The presence or absence of a particular vertex format flag communicates to the system which vertex component fields are present in memory and which you've omitted. By using only the needed vertex components, your application can conserve memory and minimize the processing bandwidth required to render your 3-D world.

Q: Do you have to use the transformation and lighting pipeline of Direct3D?

A: No. You're free to skip it. Just use the proper vertex flags and the proper vertex structure.

Q: What's the purpose of the world, view, and projection transformation?

A: The world transformation stage transforms an object from model to world space. World space is the absolute frame of reference for a 3-D world. The view transformation stage transforms your 3-D world from world space to camera space by placing the camera in the origin and pointing it directly down its positive z-axis. The projection transformation controls the camera's internals. It's analogous to choosing a lens for the camera.

Q: Why is the use of matrices a good choice in game programming?

A: The performance of a dedicated transform engine is predictable and consistent with matrices. This allows game developers to make informed decisions regarding performance and quality.

Q: To build a concept for a camera system in a game is not trivial; it depends on the vehicle in which the player is sitting. A helicopter pilot needs a different camera system than an X-wing fighter pilot. You know two ways to implement a camera system. I called them rotation about a camera axis and rotation using a quaternion. What's the problem with the first approach?

A: It looks jerky and is computationally expensive because of base vector regeneration.

Q: Describe the rotation of the camera using a quaternion.

A: The quaternion is built from the rotation vector. There's another position vector that describes the point where the rotation should happen. The rotation matrix, which is built by rotating about an arbitrary point, has to be inverted to provide a linear transformation.

Q: What are homogenous coordinates?

A: These are 4-D vertices, which consist of four elements: x, y, z, and w. Because with a 3-by-3 matrix an object could not be translated without changing the orientation, we have to use a fourth element in every vector, which is called w. To convert a 4-D point into a 3-D point, you have to divide each component—x, y, z, and w—by w.

Q: Which components that you provide to Direct3D control at least the appearance of light?

A: Material, vertex color, and light source.

Q: What's the purpose of a material in lighting a scene?

A: It reflects a given light component by specifying color values of materials and the material properties.

Q: Which type of light is provided by Direct3D?

A: Point lights, spotlights, and directional lights.

Q: What is a point light?

A: It's a light source that radiates equally in all directions from its origin.

Q: Why do we have to use a depth buffer?

A: There shouldn't be any overdraw in a scene, because that costs a lot of CPU cycles. A solution to reduce the overdraw is the use of z- or w-buffers, which hold the depth of each pixel in the scene. If a pixel is closer to the camera than one previously stored in the depth buffer, the pixel is displayed and the new depth value is stored in the depth buffer.

Q: What is the fastest primitive to use?

A: This depends on the primitive data you use. If it falls naturally into large strips or fans, then use strips or fans. Otherwise, use indexed triangle lists.

Q: What can be done by taking the cross product of two vectors?

A: The result will form a third vector, which is perpendicular to the plane formed by the first two vectors. It's used to determine which way polygons are facing. It might also be used to derive the plane equation for a plane determined by two intersecting vectors.

## ADDITIONAL RESOURCES

A nice and simple COM tutorial can be found at [www.gamedev.net/reference/articles/article1299.asp](http://www.gamedev.net/reference/articles/article1299.asp)

I've found good pages on camera orientation at:

Mr. Gamemaker: [www.tasteofhoney.freeserve.co.uk/math\\_rot4.html](http://www.tasteofhoney.freeserve.co.uk/math_rot4.html)

Flipcode: [www.flipcode.com/document/otmmatx.txt](http://www.flipcode.com/document/otmmatx.txt)

Crystalspace: [crystal.linuxgames.com](http://crystal.linuxgames.com)

Laura Downs on Quaternions: [http.cs.berkeley.edu/~laura/cs184/quat/quaternion.html](http://http.cs.berkeley.edu/~laura/cs184/quat/quaternion.html)

Pages on math stuff:

Dave's Math Tables: [www.sisweb.com/math/tables.htm](http://www.sisweb.com/math/tables.htm)

Paul Bourke: [www.swin.edu.au/astronomy/pbourke](http://www.swin.edu.au/astronomy/pbourke)

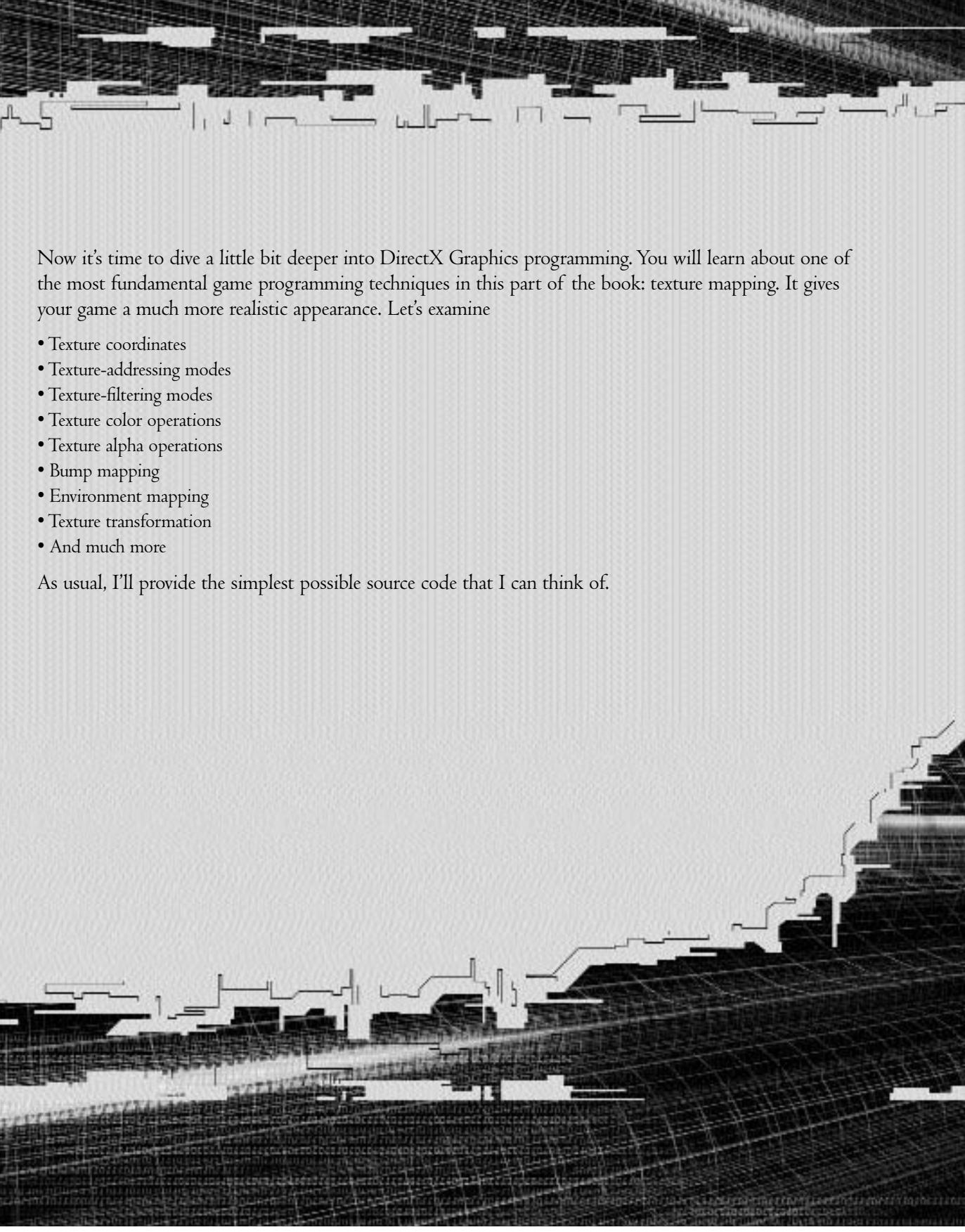
The following Web sites might also be useful:

The Web site of Witchlord at [www.witchlord.com](http://www.witchlord.com) who moderates the DirectX/OpenGL/Glide/Genesis3D discussion forum.

If you haven't found it before now, here's one of the best sites for game programmers: [www.gamasutra.com](http://www.gamasutra.com)

**PART 11**

**DATA-DRIVEN IN  
DIRECTX  
GRAPHICS  
PROGRAMMING**



Now it's time to dive a little bit deeper into DirectX Graphics programming. You will learn about one of the most fundamental game programming techniques in this part of the book: texture mapping. It gives your game a much more realistic appearance. Let's examine

- Texture coordinates
- Texture-addressing modes
- Texture-filtering modes
- Texture color operations
- Texture alpha operations
- Bump mapping
- Environment mapping
- Texture transformation
- And much more

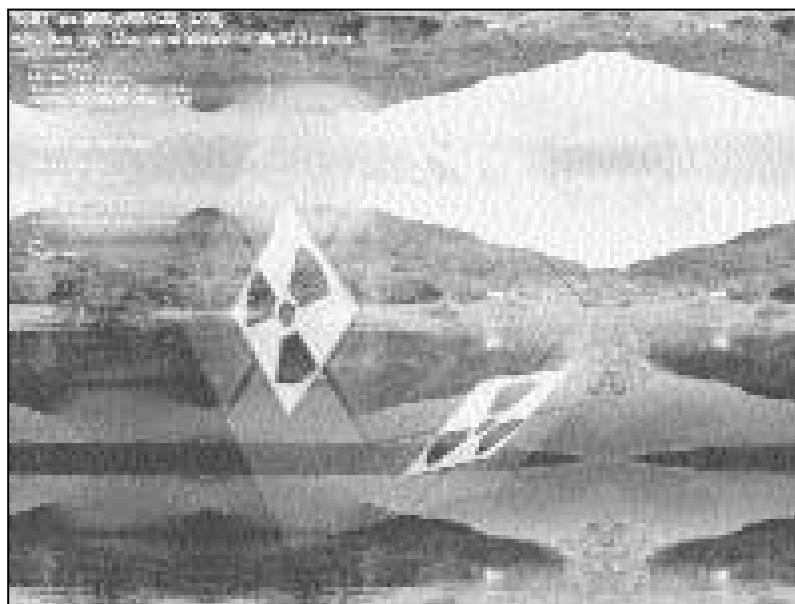
As usual, I'll provide the simplest possible source code that I can think of.

# **CHAPTER 7**

## **TEXTURE- MAPPING FUNDAMENTALS**

**I**magine a wall consisting of bricks. You do not want to build every brick and set them together, transform them, and light them. Instead, you choose to map a texture on a rectangular polygon several times or, if it's big enough, once, and the wall will look like it consists of several bricks.

I made a sample program that should show you a few different texture-mapping techniques. Figure 7.1 below shows a screen shot:



**Figure 7.1:** Example program

If you press the F1 key, you get a lot of options to choose from. Try it for yourself. This shot is taken with border color addressing mode and blending with frame buffer.

### Caution

There's one bug in the Common files, which raises its ugly head if a .bmp file cannot be found:

```
if( FAILED( D3DUtil_CreateTexture(m_pd3dDevice,
    _T("Lake.bmp"),
    &m_pBackgroundTexture)))
    return D3DAPPERR_MEDIANOTFOUND;
```

This method should return a **D3DAPPERR\_MEDIANOTFOUND** error. That doesn't work, because `D3DUtil_CreateTexture()` returns **TRUE** in the case of a missing .bmp file.

This sample program frequently uses the most important method for texture mapping, `SetTextureStageState()`. The parameters this method likes to get are:

```
HRESULT SetTextureStageState(  
    DWORD Stage,  
    D3DTEXTURESTAGESTATETYPE Type,  
    DWORD Value);
```

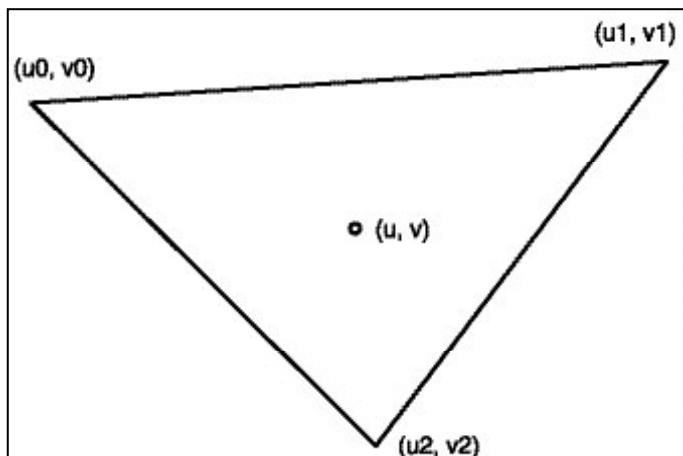
The first parameter is the stage identifier, which identifies the stage that holds the texture. Devices can handle up to eight texture stages, so the maximum value allowed here is 7. The *Type* parameter can handle operation types for

- Color operations
- Alpha operations
- Bump/environment mapping operations
- Handling different texture coordinates
- Texture addressing modes
- Texture filtering modes
- Texture transformation

The *value* corresponds to the operation type you choose. Most of the following pages deal with these parameters. Let's start by examining texture coordinates.

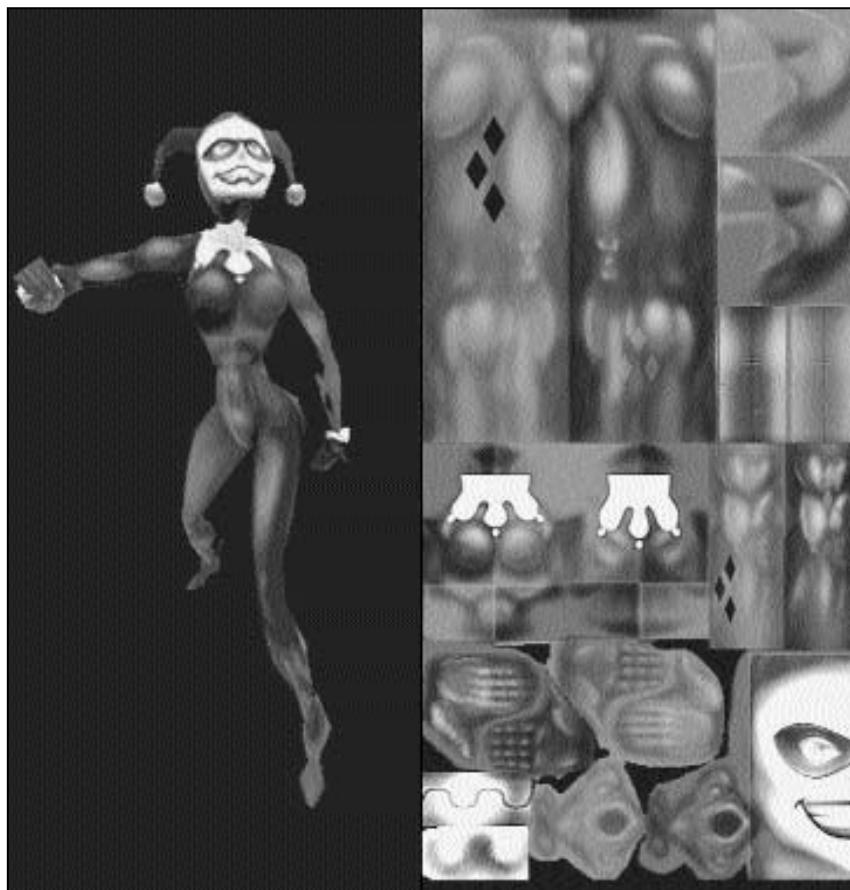
## TEXTURE COORDINATES

A texture associates some color values to two coordinates (*u*, *v*) of a polygon's area. This common approach stores color values for each *u* (column), *v* (row) coordinate pair in a two-dimensional array, so that each one has its unique address. This address is called a texture coordinate and is represented in the texture's own coordinate space. A single element of this array is often called a texel (for texture element).



**Figure 7.2:** A polygon and its texture

You can access each texel in a texture by specifying its texel coordinates. To do this, Direct3D needs a uniform address range for all the texels in a texture. Direct3D uses a normalized addressing scheme in which texture addresses consist of texel coordinates that map to the range 0.0 to 1.0. This allows you to deal with texture coordinates without worrying about the dimensions for the texture map you're using. You might assign a texture that looks like a wall of bricks on a rectangular polygon with (0.0, 0.0), (1.0, 0.0), (1.0, 1.0), and (0.0, 1.0), causing Direct3D to stretch the texture over the entire rectangle. You might only apply the left or right half of the texture to the polygon. If you choose to apply the left side of the texture, assign the texture coordinates (0.0, 0.0), (0.5, 0.0), (0.5, 1.0), and (0.0, 1.0). Keep an eye on the aspect ratio (the ratio of width to height) of the texture and of the polygon. The idea behind and use of texture coordinates is most apparent in the textures for game characters. Two bitmaps (normally 256-by-256 pixels) hold the textures for the whole character.



**Figure 7.3:** Harley Quinn, a model made by Wrath at [www.planetquake.com/polycount/cottages/wrath/](http://www.planetquake.com/polycount/cottages/wrath/)

Texture mapping doesn't change the geometric form of objects, but the color patterns used could make it appear bumpy.

If you'd like to use texture coordinates, you have to define the number of texture coordinate pairs. The texture coordinate values should be set in the vertex structure, as you remember from Part 1. If you're using only one texture coordinate pair, you will specify D3DFVF\_TEX1.

```
#define FVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_TEX1)
CUSTOMVERTEX cvVertices[] =
{
{ 0.0f, 0.0f, 1.0f, 1.0f, -0.5f, 0.0f, }, // x, y, z, rhw, tu, tv
{ (float)vp.Width, 0.0f, 1.0f, 1.0f, 1.5f, 0.0f, },
{ (float)vp.Width, (float)vp.Height, 1.0f, 1.0f, 1.5f, 3.0f, },
{ 0.0f, (float)vp.Height, 1.0f, 1.0f, -0.5f, 3.0f},
};
```

If you'd like to use more than one texture coordinate pair, you have to specify D3DFVF\_TEX2 through D3DFVF\_TEX8:

```
#define FVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_TEX1| D3DFVF_TEX2)
CUSTOMVERTEX cvVertices[] =
{
{ 0.0f, 0.0f, 1.0f, 1.0f, -0.5f, 0.0f, 1.0f, 0.5f}, // x, y, z, rhw, tu, tv, tu2,
tv2
...
};
```

You have to choose the proper texture coordinate pair with D3DTSS\_TEXCOORDINDEX.

```
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 1 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, 3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
```

The second line indicates that the `m_pWallTexture` should use the second texture coordinate pair.

**NOTE**

Instead of storing a texture as a bitmap, you might also use so-called procedural textures, where some function computes a color for arbitrary u and v coordinates.

## TEXTURE-ADDRESSING MODES

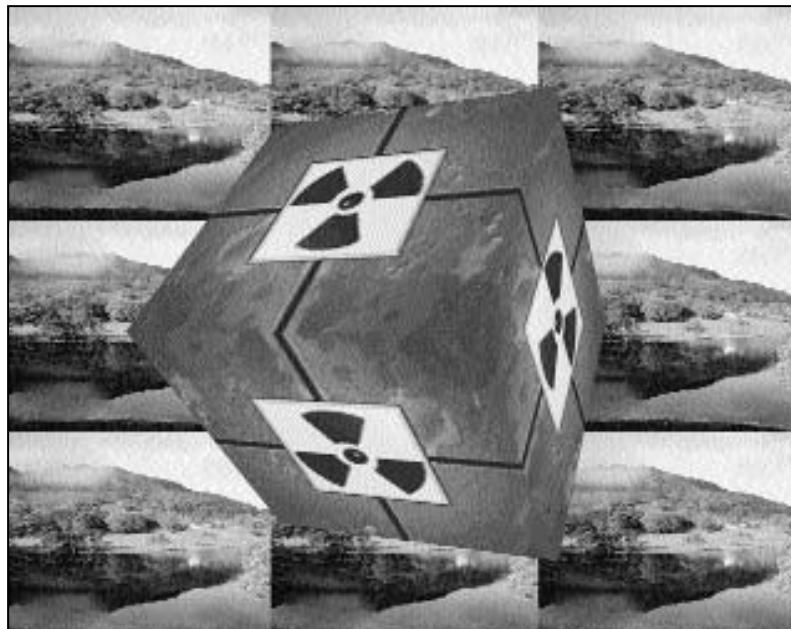
OK... the behavior for choosing texels at our vertices between 0.0 to 1.0 is pretty well defined, but what happens to texels outside of that range? You can create special effects by using these outside values. How the values outside the 0.0 to 1.0 range modify the appearance of the textures depends on the texture addressing mode you choose. The different addressing modes are called wrap, mirror, clamp, border color, and mirroronce.

### WRAP TEXTURE-ADDRESSING MODE

You might use the wrap texture-addressing mode to repeat the texture on every integer injunction. This is the default mode used by Direct3D. Try the following one in the example application:

```
CUSTOMVERTEX cvVertices[] =  
{  
    { 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, }, // x, y, z, rhw, tu, tv  
    { (float)vp.Width, 0.0f, 1.0f, 1.0f, 3.0f, 0.0f, },  
    { (float)vp.Width, (float)vp.Height, 1.0f, 1.0f, 3.0f, 3.0f, },  
    { 0.0f, (float)vp.Height, 1.0f, 1.0f, 0.0f, 3.0f},  
};
```

The background texture will be applied three times in the u- and v-directions. It should look like the following screen shot.

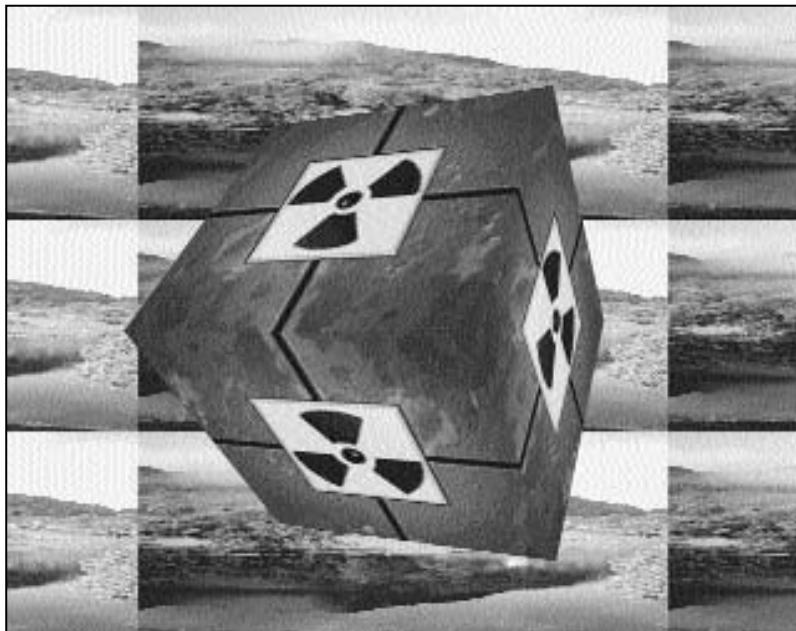


**Figure 7.4:** Wrap address mode

If you choose to use the following texture coordinates, the texture will be tiled in another way:

```
CUSTOMVERTEX cvVertices[] =  
{  
    { 0.0f, 0.0f, 1.0f, 1.0f, -0.5f, 0.0f, }, // x, y, z, rhw, tu, tv  
    { (float)vp.Width, 0.0f, 1.0f, 1.0f, 1.5f, 0.0f, },  
    { (float)vp.Width, (float)vp.Height, 1.0f, 1.0f, 1.5f, 3.0f, },  
    { 0.0f, (float)vp.Height, 1.0f, 1.0f, -0.5f, 3.0f},  
};
```

And here's the screen shot:

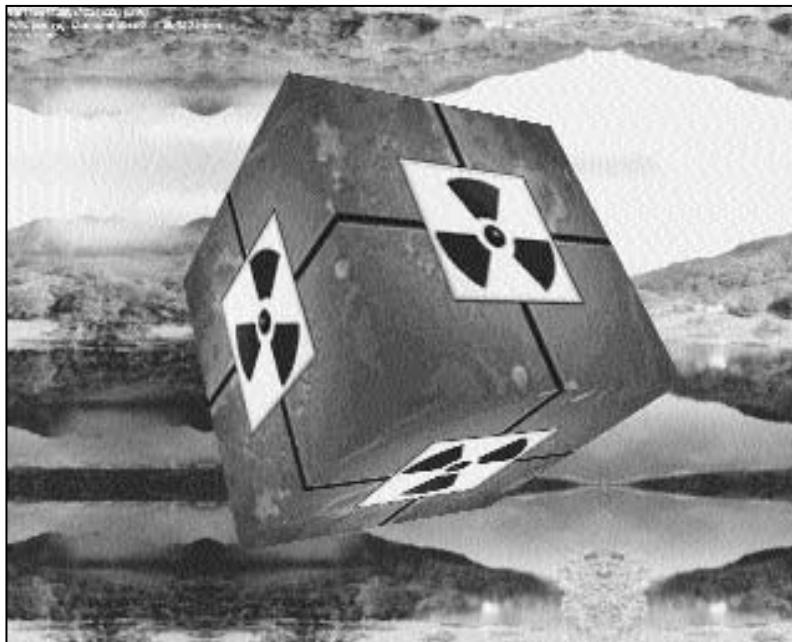


**Figure 7.5:** Wrap address mode

The upper left texture starts at  $-0.5f$  and repeats 1.5 times from left to right. If you change the  $1.5f$  values to  $2.5f$  values, it will be repeated 2.5 times.

## MIRROR TEXTURE-ADDRESSING MODE

The mirror texture-addressing mode causes Direct3D to mirror the texture at every integer boundary, so the texels are flipped outside of the 0.0 to 1.0 region. The texture is mirrored along each axis in this way.



**Figure 7.6:** Mirror addressing mode

You can set the mirror addressing mode by using `SetTextureStageState()` methods with the parameter `D3DTADDRESS_MIRROR`.

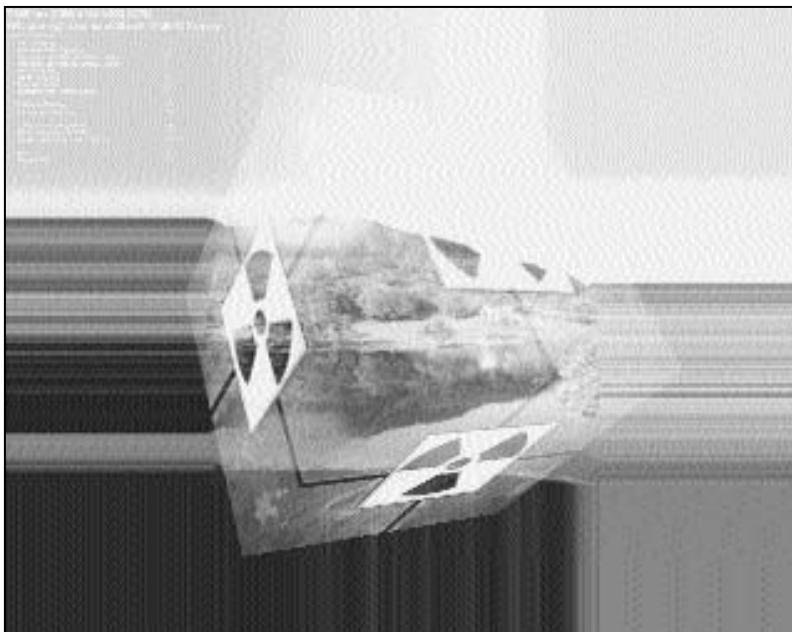
```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ADDRESSU, D3DTADDRESS_MIRROR );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ADDRESSV, D3DTADDRESS_MIRROR );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ADDRESSW, D3DTADDRESS_MIRROR );
```

I've used the following vertex structure:

```
CUSTOMVERTEX cvVertices[] =
{
    { 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, }, // x, y, z, rhw, tu, tv
    { (float)vp.Width, 0.0f, 1.0f, 1.0f, 3.0f, 0.0f, },
    { (float)vp.Width, (float)vp.Height, 1.0f, 1.0f, 3.0f, 3.0f, },
    { 0.0f, (float)vp.Height, 1.0f, 1.0f, 0.0f, 3.0f},
};
```

## CLAMP TEXTURE-ADDRESSING MODE

The clamp texture-addressing mode applies the texture only once on the polygon and then smears the color of the edge pixels of the texture. With the same vertices used in the last example, that might look like the following screen shot:



**Figure 7.7:** Clamp addressing mode

The pixel colors at the bottom of the columns and the end of the rows are extended to the bottom and right of the primitive, respectively. The `SetTextureStageSet()` method might look like the following call:

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ADDRESSU, D3DTADDRESS_CLAMP );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ADDRESSV, D3DTADDRESS_CLAMP );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ADDRESSW, D3DTADDRESS_CLAMP );
```

## BORDER COLOR TEXTURE- ADDRESSING MODE

With the border color texture-addressing mode set, a border will show up around the texture with the color you choose. You set the border color mode with

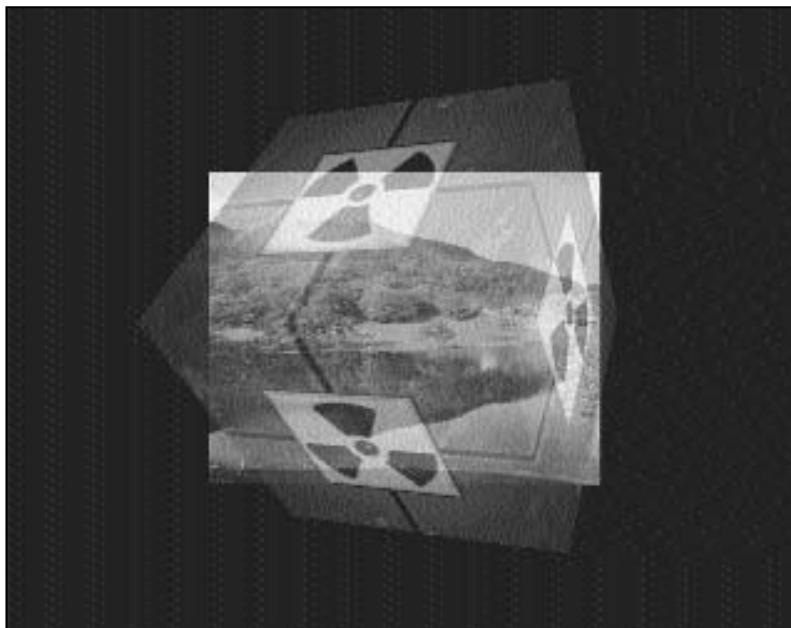
```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ADDRESSU, D3DTADDRESS_BORDER );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ADDRESSV, D3DTADDRESS_BORDER );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ADDRESSW, D3DTADDRESS_BORDER );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_BORDER- COLOR, 0x00000000 );
```

### TIP

The clamp mode is convenient if you want to stitch multiple textures together over a mesh, since the bilinear interpolation at the edges of the default wrap mode introduces artifacts from the opposite edge of the textures. It doesn't fix things entirely; for that, you have to modify your texture coordinates.

The vertices might look like this:

```
CUSTOMVERTEX cvVertices[] =  
{  
    { 0.0f, 0.0f, 1.0f, 1.0f, -0.5f, -0.5f, }, // x, y, z, rhw, tu, tv  
    { (float)vp.Width, 0.0f, 1.0f, 1.0f, 1.5f, -0.5f, },  
    { (float)vp.Width, (float)vp.Height, 1.0f, 1.0f, 1.5f, 1.5f, },  
    { 0.0f, (float)vp.Height, 1.0f, 1.0f, -0.5f, 1.5f},  
};
```



**Figure 7.8:** Border color addressing mode

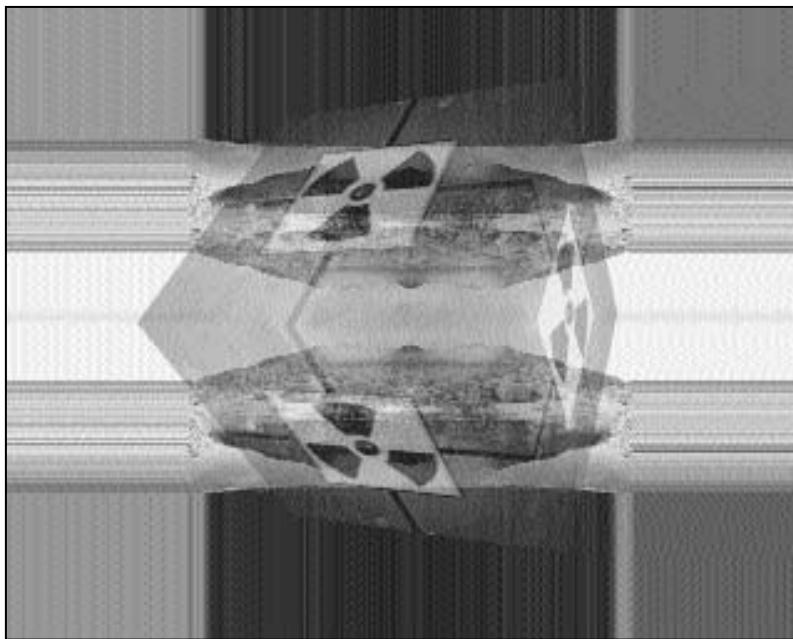
You might fake large textures on walls with this texture mode by using a smaller texture surrounded by the texture's color.

### MIRRORONCE TEXTURE-ADDRESSING MODE

With DirectX 8.0, a new texture-addressing mode that is sort of a hybrid of mirror and clamp modes appeared. The texture is mirrored within the range of  $-1.0$  to  $1.0$ , and outside of this range, it is clamped. This one is cool.

#### NOTE

As far as I can see, this mode is not used often. Besides, it's not supported by GeForce-driven graphic cards.



**Figure 7.9** MirrorOnce addressing mode

```
m_pd3dDevice->SetTextureStageState(0, D3DTSS_ADDRESSU, D3DTADDRESS_MIRRORONCE);  
m_pd3dDevice->SetTextureStageState(0, D3DTSS_ADDRESSV, D3DTADDRESS_MIRRORONCE);  
m_pd3dDevice->SetTextureStageState(0, D3DTSS_ADDRESSW, D3DTADDRESS_MIRRORONCE);
```

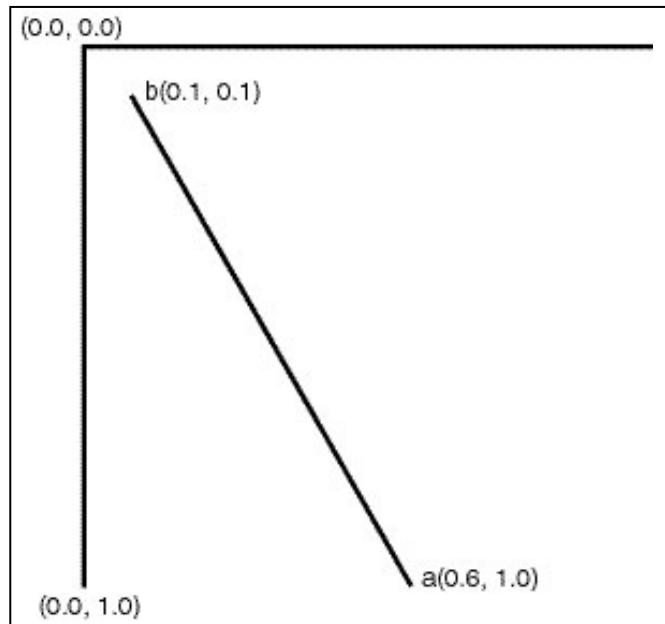
## TEXTURE WRAPPING

Texture wrapping is different than the wrap texture-addressing mode. Instead of deciding how texel coordinates outside the boundary of (0.0, 1.0) should be mapped, it decides how to interpolate

between texture coordinates. It affects the way Direct3D rasterizes textured polygons and the way it utilizes the texture coordinates specified for each vertex. Normally, the system treats the texture as a 2-D plane. It interpolates new texels by taking the shortest route from point a to point b within a texture, assuming that 0.0 and 1.0 are coincident. If point a represents the u, v position (0.6, 1.0), and point b is at (0.1, 0.1), the line of interpolation looks like the following illustration.

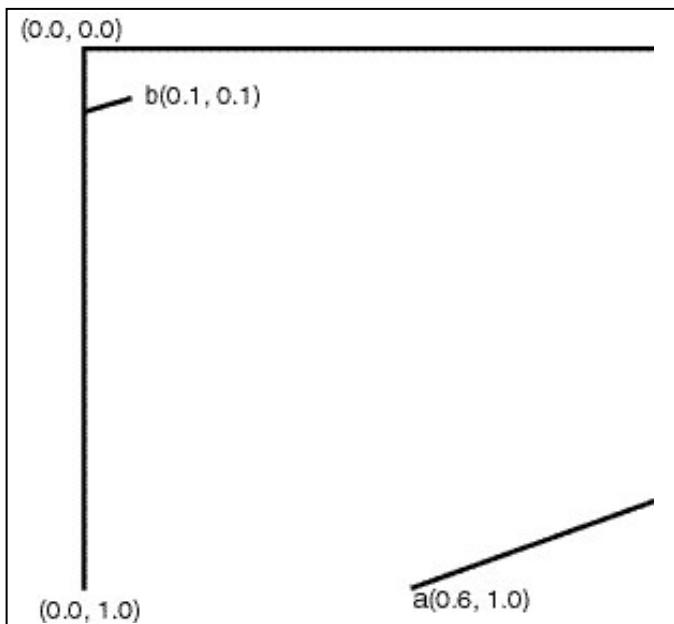
### NOTE

The **D3DTADDRESS\_MIRRORONCE** addressing mode was motivated by axial symmetry typically exhibited by volumetric light maps.



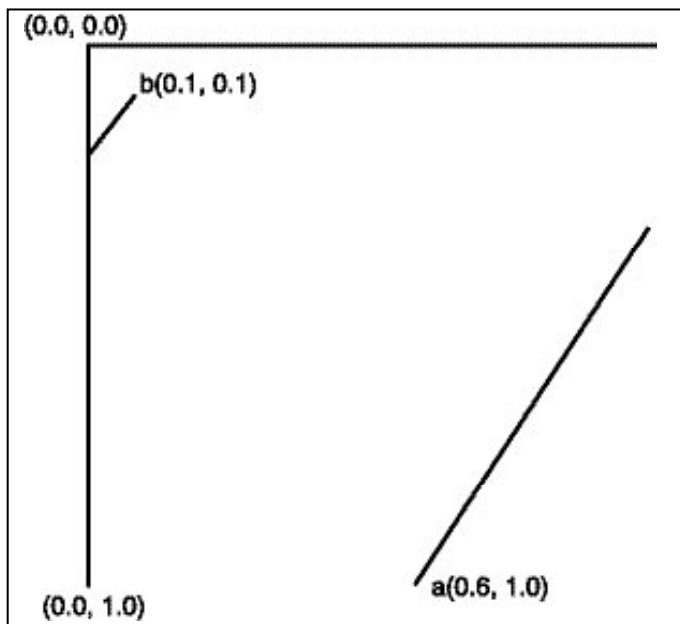
**Figure 7.10:** Default texture wrapping

The shortest distance between a and b runs roughly through the middle of the texture. If texture wrapping in the v-direction is enabled, the rasterizer interpolates in the shortest direction:



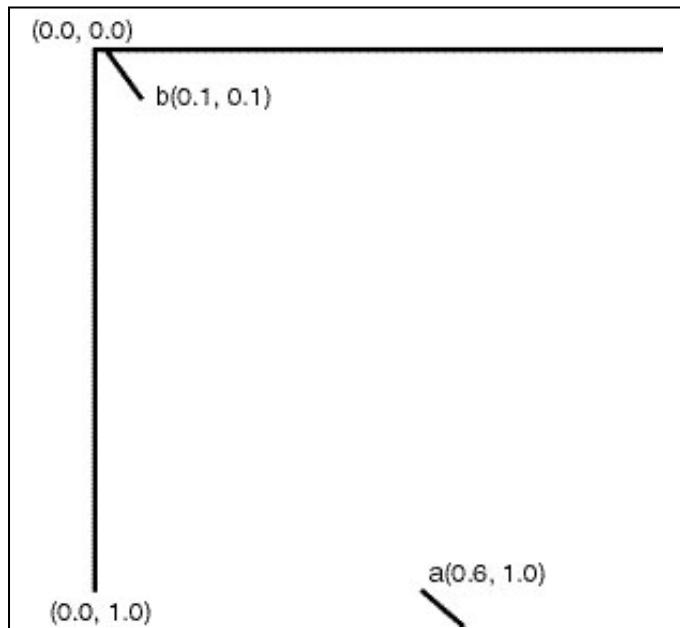
**Figure 7.11:** Texture wrapping in the v-direction

Whereas texture wrapping in the u-direction would look like this:



**Figure 7.12:** Texture wrapping in the u-direction

If you combine both, you might get the following illustration:

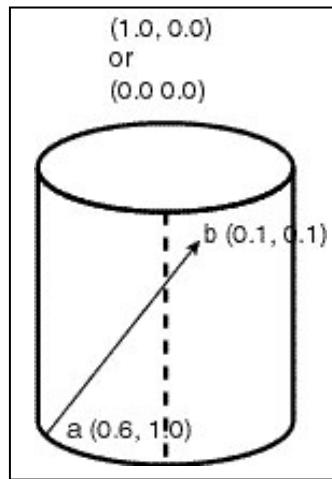


**Figure 7.13:** Texture wrapping in the u- and v-directions

To enable texture wrapping for each stage of the texture pipeline, we have to set the render state with D3DRS\_WRAPx, where x is the desired texture stage (from 0 to 7). To enable the texture-wrapping direction, we include D3DWRAP\_U for the u-direction and D3DWRAP\_V for the v-direction.

So what's the deal with all that stuff?

You can imagine that enabling texture wrapping in one direction causes the system to treat a texture as though it were wrapped around a cylinder.



**Figure 7.14:** Texture wrapping in the u-direction

You can see that the shortest route to interpolate the texture is no longer across the middle of the texture. It's now across the border where 0.0 and 1.0 exist together. Wrapping in the v-direction works for a cylinder that is lying on its side.

Wrapping in both the u- and v-directions is more complex. In this situation, you can envision the texture as a torus, or doughnut. Wrapping in both directions is used if you map a texture around a sphere, for example. It's enabled for the first texture stage by calling

```
m_pd3dDevice->SetRenderState( D3DRS_WRAP0, D3DWRAP_U | D3DWRAP_V );
```

## TEXTURE FILTERING AND TEXTURE ANTI-ALIASING

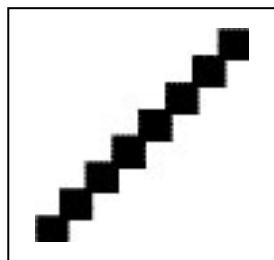
Imagine a checkerboard that moves away from the viewer. At some point, the projection of the squares

### CAUTION

Activating texture wrapping makes texture coordinates outside of the 0.0 and 1.0 range invalid and texture addressing modes unavailable.

becomes smaller than pixels on the screen. Naturally, if both black and white pixels project to the same pixel, this pixel should appear as some shade of gray. By default, a texture mapper will not consider areas, so either black or white pixels will be drawn depending on the rounding. This can cause the image to look unnatural and produce artifacts.

Another example is the so-called staircase effect. Lines consisting of different plotted pixels produce edges with the default straightforward rasterization algorithms used in the popular 3-D graphic APIs.



**Figure 7.15:** Staircase effect

Both examples demonstrate aliasing problems caused by the nature of raster graphics. To overcome these shortcomings, Direct3D provides three texture-filtering methods and the so-called mipmaps. (MIP is an acronym for “multum in paro” in Latin, or “many things in a small place.” Wait a few seconds for a deeper explanation.) The texture-filtering methods are:

- Nearest point sampling
- Linear texture filtering
- Anisotropic texture filtering

Filtering is the way in which we get texels from the texture map given a u, v coordinate pair. We might differentiate between two separate issues in the checkerboard example shown earlier: magnification and minification.

Magnification occurs when we try to map, for example, a 64-by-64 texture onto a 400-by-400 pixel polygon. The staircase effect will come up. Minification occurs when we do the opposite by drawing a 64-by-64 texture onto a 10-by-10 pixel polygon. Swimming pixels will occur.

Most of the newer hardware supports four different varieties of filtering to alleviate magnification and minification artifacts: point sampling, bilinear filtering (or linear filtering in Direct3D terms), trilinear filtering (or linear filtering plus mipmapping in Direct3D terms), and anisotropic filtering.

### NOTE

There are also **D3DTEXF\_FLATCUBIC** and **D3DTEXF\_GAUSSIANCUBIC** filtering methods. They were introduced in DirectX 7.0, but nothing supports them, and I don't know of anybody even planning to support them. Just ignore them.

## MIPMAPS

Mipmaps consist of a series of textures, each containing a progressively lower resolution of an image that represents the texture.

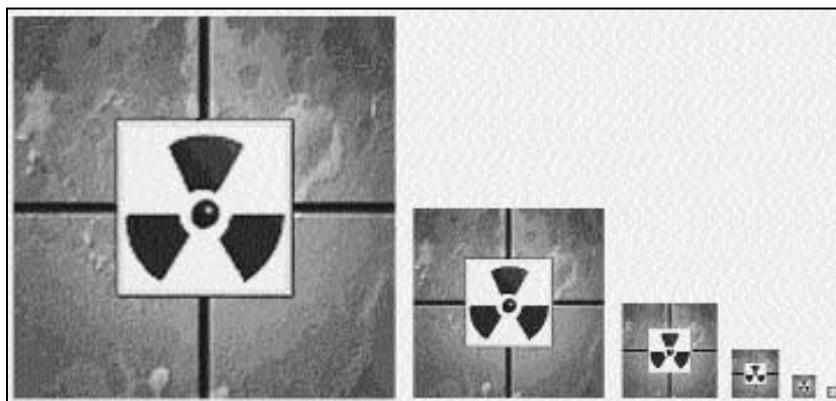


Figure 7.16: Mipmap

Each level in the mipmap sequence has a height and a width that is half of the height and width of the previous level. The levels could be either square or rectangular.

Mipmaps are produced and used automatically for you if you provide the right parameters to the `CreateTexture()` method, which uses the `D3DXCreateTextureFromFileEx()` method explained in Part I, Chapter 5 in Basic example 2.

Using mipmaps ensures that textures retain their realism and quality as you move closer (minification problem) or further away (magnification problem) from them. There are two additional arguments: filtering a 256-by-256 texture into a 2-by-2 polygon might look great, but it takes too long, and mipmaps provide an effective way to reduce memory traffic for textures.

Direct3D will automatically pick the appropriate mip level to texture from. It picks the mip level with the texel-to-pixel ratio closest to 1. If a polygon is approximately 128-by-128 pixels, Direct3D uses the 128-by-128 mip level. If the polygon will have 95-by-95 pixels, Direct3D will pick the 64-by-64 mip level.

### TIP

If you would like to access single mipmaps, you should retrieve a pointer to the individual map surfaces with `IDirect3DTexture::GetSurfaceLevel()`. Be sure to call `Release` on these surface when finished, as their reference count is incremented when you retrieve the surface pointer.

## NEAREST-POINT SAMPLING

Nearest-point sampling is the simplest kind of a filter. It snaps the texture coordinates to the nearest integer, and the texel at that coordinate is used as the final color. It's a fast and efficient way to process textures in a size that is similar to the size of the polygon's image on the screen.

Direct3D maps a floating point texture coordinate ranging from 0.0 to 1.0 to an integer texel space value ranging from  $-0.5, n - 0.5$ , where  $n$  is the number of texels in a given dimension on the texture. The resulting texture index is rounded to the nearest integer, which may introduce inaccuracies at texel boundaries. The system might choose one sample texel or the other, and the result can change abruptly from one texel to the next as the boundary is crossed. So artifacts that do not look visually pleasant can occur. This often happens when you map a small texture onto a big polygon (magnification).

Because the majority of graphics hardware today is optimized for linear filtering, applications should avoid using nearest-point sampling wherever possible.

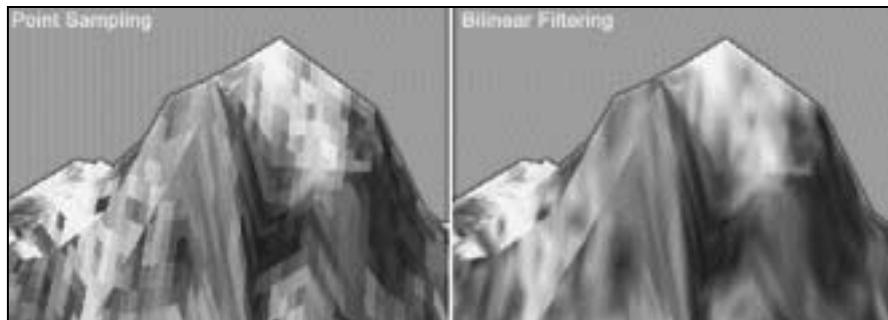
## LINEAR TEXTURE FILTERING

Linear texture filtering uses bilinear texture filtering, which averages the four nearest texels together based on the relative distances from the sampling point . . . OK . . . step by step. Bilinear filtering

- First computes a texel address, which is usually not an integer address
- Then finds the texel whose integer address is closest to the computed address
- After that, computes a weighted average of the texels that are immediately above, below, to the left of, and to the right of, the nearest sample point.

You invoke linear texture filtering with

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_MINFILTER, D3DTEXF_LINEAR );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_MAGFILTER, D3DTEXF_LINEAR );
```



### NOTE

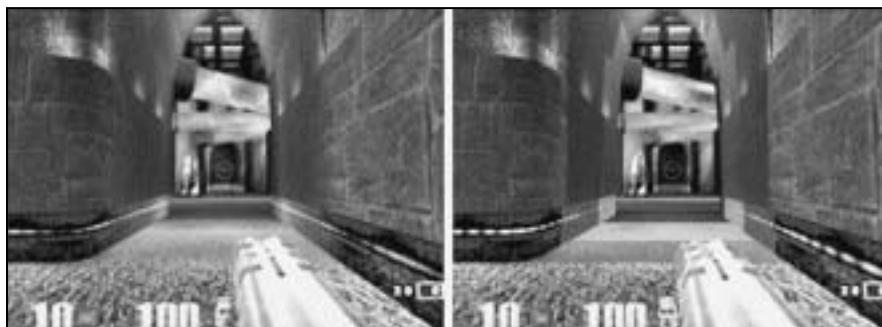
Point sampling is used in the PlayStation I console and in the first generation of 3-D games like Descent and Quake. Quake got past some visual artifacts of point sampling by using mipmaps: choosing the mipmap based on distance and point sampling out of that.

**Figure 7.17:** Bilinear Filtering (courtesy of [www.reactorcritical.com](http://www.reactorcritical.com))

You might also pass D3DTSS\_MIPFILTER as a second argument. It indicates the texture filter to use between mipmap levels. The default value is D3DTEXF\_NONE. If you provide D3DTEXF\_LINEAR, you are using trilinear texture filtering, or mipmap filtering.

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_MAGFILTER, D3DTEXF_LINEAR );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_MINFILTER, D3DTEXF_LINEAR );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_MIPFILTER, D3DTEXF_LINEAR );
```

Trilinear texture filtering looks much better than bilinear filtering, especially as we move through a scene. How does it work? It chooses for each pixel the two nearest mipmaps, which are bilinear filtered. The two bilinear-filtered pixels are combined together using the proximity of each mipmap to the ideal mip level. If the ideal mip level is 3.4, then the combination should be 0.6 (1 minus 0.4) times the bilinear-filtered result from mip level 3, plus 0.4 times the bilinear-filtered result from mip level 4.



**Figure 7.18:** Trilinear filtering/bilinear filtering (courtesy of [www.reactorcritical.com](http://www.reactorcritical.com))

## ANISOTROPIC FILTERING

One of the drawbacks in bilinear and trilinear filtering is that texels are sampled using square sampling areas. If a texture is angled sharply away from you, distortion effects can occur. This distortion effect is called anisotropy.

Direct3D measures the anisotropy of a pixel as the elongation—that is, length divided by width—of a screen pixel that is inverse-mapped into texture space. Anisotropic filtering samples more texels when a screen pixel is elongated to reduce the blurriness that the standard linear filtering can produce. The number of texels could be between 16 and 32 from the texture maps. Using more texels requires an even bigger memory bandwidth and is almost impossible on traditional rendering systems, unless you use a very expensive memory architecture. Tile-based rendering (used in PowerVR and ATI Radeon cards) saves a lot of bandwidth and allows the implementation of anisotropic filtering.

### NOTE

Trilinear filtering is supported by the ATI Radeon and GeForce cards in three respective two texture stages at once. A lot of graphic cards will only support trilinear filtering in one texture stage.

Anisotropic rendering offers a big visual improvement by giving you better depth detail and an accurate representation of texture maps on polygons that are not parallel to the screen. There are different levels of quality available, depending on the number of pixel samples, or *taps*. Sixty-four-tap anisotropic filtering offers much higher quality than 8-tap anisotropic filtering, but again, it's slower. Anisotropic filtering is the latest filtering type to be implemented in 3-D accelerators.



**Figure 7.19:** Point sampling/bilinear filtering/anisotropic filtering (courtesy of [www.reactorcritical.com/id](http://www.reactorcritical.com/id) software)

You will get anisotropic filtering by setting the following SetTextureStageState() methods:

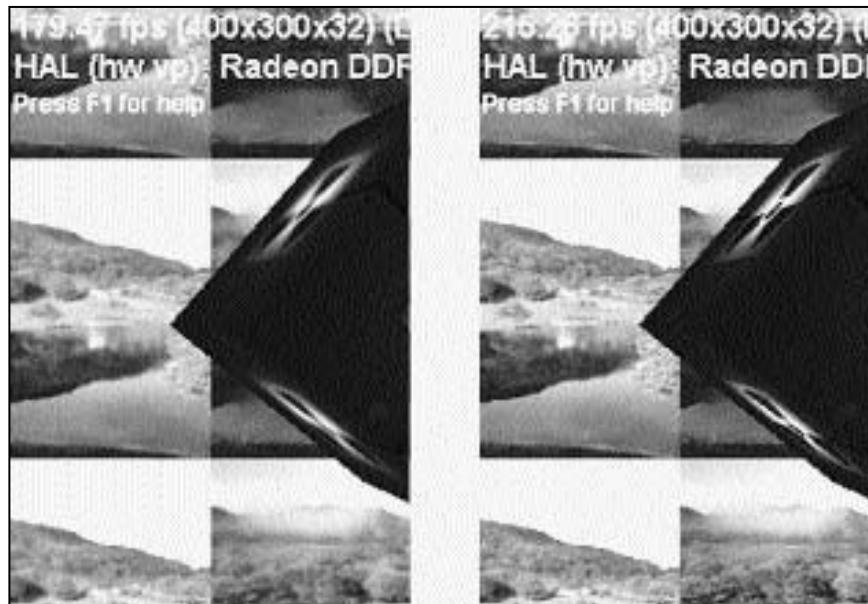
```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_MAGFILTER, D3DTEXF_ANISOTROPIC);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_MINFILTER, D3DTEXF_ANISOTROPIC);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_MIPFILTER, D3DTEXF_ANISOTROPIC);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_MAXANISOTROPY, 16);
```

My ATI Radeon supports a maximum anisotropy level of 16. My GeForce DDR card supports a maximum anisotropy level of 2. You might get the maximum anisotropy level from the DirectX Caps Viewer or by checking the proper flag in the ConfirmDevice() method. You can see the difference in the sample program by pressing the F5 and F6 keys:

Look at the yellow sign in the middle of the wall texture and you see the difference. The anisotropic filtering is much sharper.

### NOTE

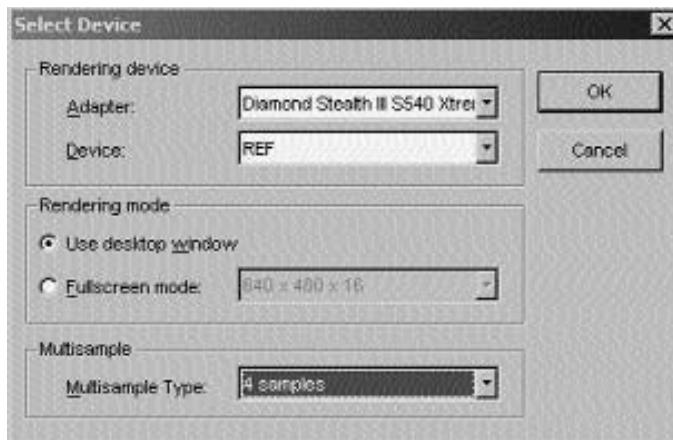
The quality of the support of anisotropic filtering is very different among the graphic cards at the moment of this writing. A lot of people think that the ATI Radeon has the best support. Most other cards don't produce a really visually pleasant anisotropic filtering, but that might change very soon.



**Figure 7.20:** Trilinear filtering/anisotropic filtering

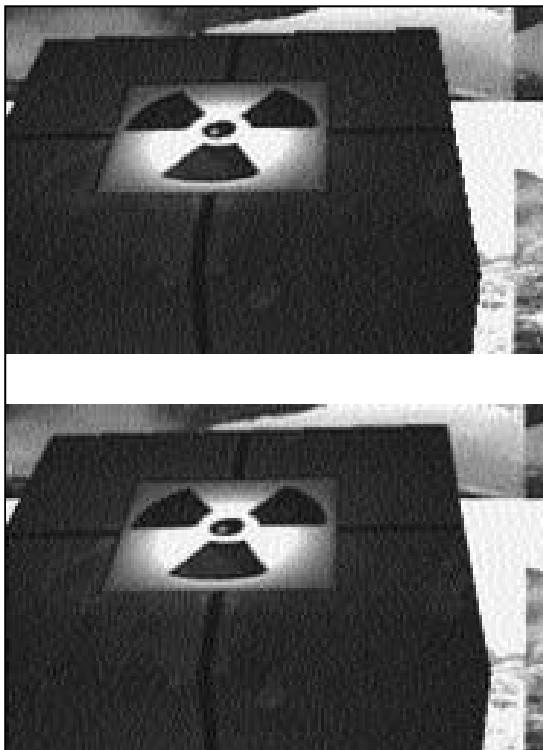
## FULL-SCENE ANTI-ALIASING

Full-scene anti-aliasing is an instant upgrade to all PC games. It has long been the Holy Grail in 3-D computer graphics. 3dfx, especially, has promoted this feature a lot. It's now a default feature in DirectX Graphics and is called multisampling. FSAA hides the jagged effect of image diagonals by modulating the intensity on either side of the diagonal boundaries. This creates a local blurring along these edges and reduces the appearance of stepping. The result is a smoother, far more realistic image. You will see the effect instantly. Switch it on in the Options/Change Device dialog box.



**Figure 7.21:** Switch on multisampling

On my slow computer, the multitexture example with dark mapping runs with about 330 fps. When I switch on multisampling with four samples, it turns down to about 80 fps, but the edges are much sharper.



**Figure 7.22:** Without multisampling (upper image)/with four samples multisampling (bottom)

So what does it do?

Each pixel is sampled multiple times. The various samples recorded for each pixel are blended together and output to the screen. This enables the improved image quality of anti-aliasing or other effects.

Multisampling is used for a few special effects, which will be part of the new game generation. You might specify that only some samples are affected by a given rendering pass, enabling simulation of motion blur, depth-of-field focus effects, reflection blur, and so on.

### TIP

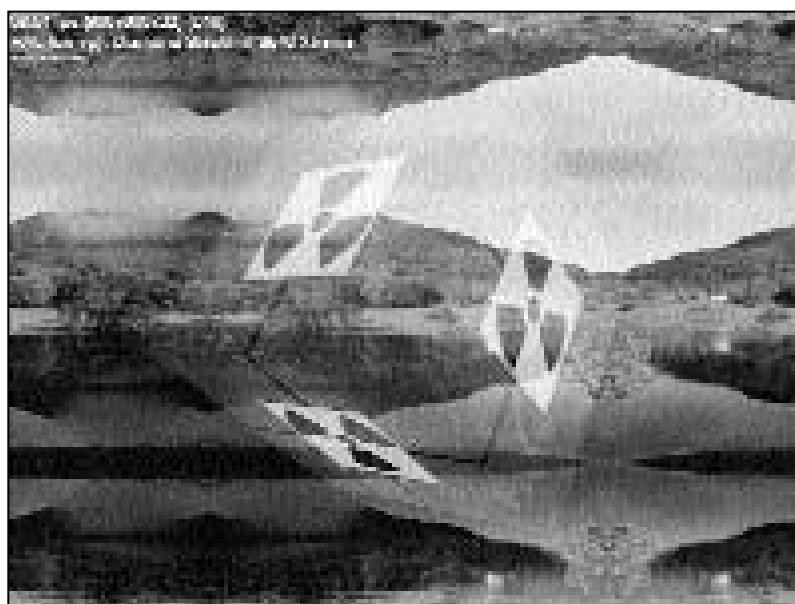
You might check the support of multisampling with a call to `CheckDeviceMultiSampleType()`. Setting a multi-sampling mode could be done by a call to `CreateDevice()`. Both are done in the Common files framework for you.

**NOTE**

How do you obtain a motion blur effect? You have to render your object several times. Before each render call, the position of the object is updated slightly. Additionally, the textures and the object itself are rendered, for example, the first time with 90 percent transparency, the second time with 60 percent, the third time with 30 percent, and the last time without transparency. So each time the object has to be rendered, the transparency decreases. To display the frame on the screen, your application should use the `Present()` method. If your application is simulating the effect of a user moving through a scene at high speed, it can add motion blur to the entire scene by moving the viewpoint slightly after each render call.

## ALPHA BLENDING

You can use alpha blending to combine the primitive's color with the color previously stored in that pixel of the frame buffer. This is called blending with the frame buffer. Using this form of alpha blending, you can simulate semitransparent objects, combine two images, and add special effects such as force fields, flames, plasma beams, and light mapping.



**Figure 7.23:** Alpha blending

The alpha-blending step is one of the last in the pixel pipeline. The pixel in the alpha-blending step is combined with the pixel that is currently in the frame buffer using blending factors. We can add the two pixels together, multiply them together, combine them linearly using the alpha component, and so on. So when performing alpha blending, two colors are being combined: a source color and a destination color. The source color is the pixel we are attempting to draw to the frame buffer; the destination color is the pixel that already exists in the frame buffer. Alpha blending uses a formula to control the ratio between the source and destination objects:

```
FinalColor = SourcePixelColor * SourceBlendFactor + DestPixelColor * DestBlendFactor
```

SourcePixelColor is the contribution from the primitive being rendered at the current pixel location. DestPixelColor is the contribution from the frame buffer at the current pixel location. You can change the SourceBlendFactor (`D3DRS_SRCBLEND`) and DestBlendFactor (`D3DRS_DESTBLEND`) flags to generate the effects you want. The FinalColor is the color that is written to the frame buffer.

Now let's think about a few examples. If you want the alpha blending to do nothing, just draw the pixel and don't consider what was already there in the frame buffer.

```
FinalColor = SourcePixelColor * 1.0 + DestPixelColor * 0.0
```

With a DestBlendFactor of 0.0, we might reduce the equation to:

```
FinalColor = SourcePixelColor
```

Your source might look like:

```
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);  
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);  
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ZERO);
```

If we want to multiply the source and destination color together, we have to set the SourceBlendFactor to 0.0 and the DestBlendFactor to SourcePixelColor:

```
FinalColor = SourcePixelColor * 0.0 + DestPixelColor * SourcePixelColor
```

Your source might look like:

```
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);  
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ZERO);  
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_SRCCOLOR);
```

The code we have used here to get the transparent cube looks like:

```
if (m_bTex8 == TRUE)
{
    m_pd3dDevice->SetRenderState (D3DRS_ALPHABLENDENABLE, TRUE);
    m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);
    m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCOLOR);
    m_pd3dDevice->SetTexture( 0, m_pWallTexture);
}

// render primitives
if (m_bTex8 == TRUE)
{
    // switch off alpha blending
    m_pd3dDevice->SetRenderState (D3DRS_ALPHABLENDENABLE, FALSE);
}
```

As a result of the calls in the preceding code fragment, Direct3D performs a linear blend between the source color (the color of the primitive being rendered at the current location) and the destination color (the color at the current location in the frame buffer). This gives an appearance similar to tinted glass. Some of the color of the destination object seems to be transmitted through the source object. The rest of it appears to be absorbed. The equation looks like:

FinalColor = SourcePixelColor \* SourceColor + DestPixelColor \* InverseSourceColor

Whereas InverseSourceColor is:

(1 - SourcePixelColor<sub>red</sub>, 1 - SourcePixelColor<sub>green</sub>, 1 - SourcePixelColor<sub>blue</sub>, 1 - SourcePixelColor<sub>alpha</sub>).

There are a lot of alpha-blending factors, which you will find explained in the DirectX 8.0 SDK documentation. Just try to search for “blending.”

Alpha blending requires a fair bit of extra math and memory access, so turning it on and off with ALPHABLENDENABLE to False after the call to Render() is worth the effort.

### CAUTION

If we use a transparent object, we might get in trouble with the depth buffer. For the transparent object to appear correctly, the value in the depth buffer must be what we would naturally see behind the transparent object. We have to sort our alpha-blended objects in a back-to-front list to be used correctly by the depth buffer. They have to be drawn after all the non-alpha-blended objects.

The least we have to worry about is whether our graphics hardware supports alpha blending. This can be done by checking the special caps.

```
if( 0 == ( pCaps->DestBlendCaps & D3DPBLENDCAPS_INVSRCOLOR) ) &&  
    (0 == ( pCaps->SrcBlendCaps & D3DPBLENDCAPS_SRCCOLOR)) &&  
    ...
```

D3DPBLENDCAPS\_\* is the flag to check the capability of the graphics card to support D3DBLEND\_\* flags. The DestBlendCaps or the SrcBlendCaps member of the D3DCAPS8 structure checks the capabilities of these flags for the D3DRS\_DESTBLEND / D3DRS\_SRCBLEND parameters.

After getting the alpha-blending stuff in our brains, we can move another step. Until now, we have talked only about the use of one texture. Now is the time to juggle more than one ball/texture in the air.

*This page intentionally left blank*

# **CHAPTER 8**

## **USING MULTIPLE TEXTURES**

**O**ne of the most interesting features introduced to Direct3D in DirectX 6.0 was the possibility to map more than one texture on a polygon at once. We'll examine this in the following chapter.

There are basically two ways to map more than one texture on a polygon. Step by step, which means (pseudocode):

```
// texture #1
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE );
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
m_pd3dDevice->SetStreamSource( 0, m_pCubeVB, sizeof(CUBEVERTEX) );
m_pd3dDevice->SetVertexShader( FVF_CUBEVERTEX );
m_pd3dDevice->SetIndices( m_pCubeIB, 0 );
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 24, 0, 36/3 );
// texture #2
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE );
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE );
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE );
m_pd3dDevice->SetTexture(0, m_pDetailTexture );
m_pd3dDevice->SetStreamSource( 0, m_pCubeVB, sizeof(CUBEVERTEX) );
m_pd3dDevice->SetVertexShader( FVF_CUBEVERTEX );
m_pd3dDevice->SetIndices( m_pCubeIB, 0 );
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 24, 0, 36/3 );
```

This is called *multipass rendering*, because we use more than one pass for the rendering of the textures. Brian Hook tells us in his course 29 notes at SIGGRAPH '98, that Quake III uses 10 passes:

- (passes 1–4: accumulate bump map)
- pass 5: diffuse lighting
- pass 6: base texture (with specular component)
- (pass 7: specular lighting)
- (pass 8: emissive lighting)
- (pass 9: volumetric/atmospheric effects)
- (pass 10: screen flashes)

Only on the fastest machine are up to 10 passes done to render a single frame. If the graphics accelerator cannot maintain a reasonable frame rate, various passes (those in parentheses) can be eliminated.

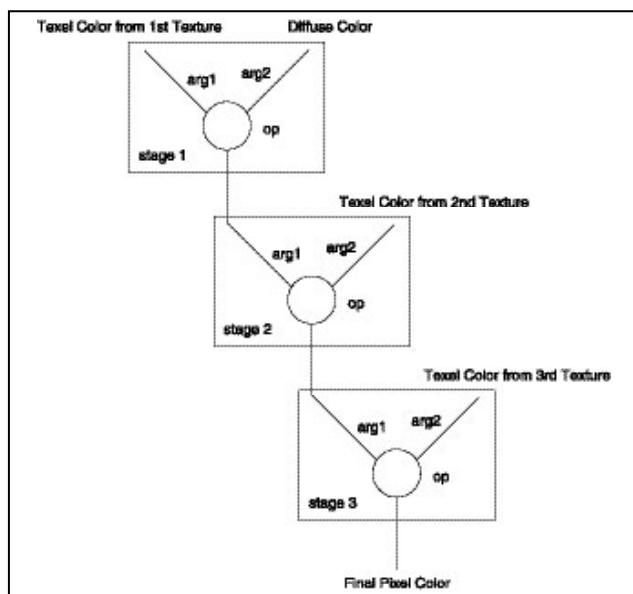
It's obvious that the more passes a renderer must take, the lower its overall performance will be.

To reduce the number of passes, some graphics accelerators support multitexturing, in which two or more textures are accessed during the same pass. Multitexturing works like this (pseudocode):

```
// texture #1
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );

// texture #2
m_pd3dDevice->SetTexture(1, m_pDetailTexture );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADDSIGNED );
m_pd3dDevice->SetStreamSource( 0, m_pCubeVB, sizeof(CUBEVERTEX) );
m_pd3dDevice->SetVertexShader( FVF_CUBEVERTEX );
m_pd3dDevice->SetIndices( m_pCubeIB, 0 );
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 24, 0, 36/3 );
```

There's only one render call (one pass) to map the two textures at once on a polygon. If the second texture uses a result produced by the texture operation with the first texture, this is called *texture blending*. You may put together the results of up to eight texture passes. This is called *texture blending cascade*.



**Figure 8.1:** Texture blending cascade

Before Direct3D 6.0, the pipeline stages determined the texel color and blended it with the color of the primitive interpolated from the vertices. Since Direct3D 6.0, up to eight texture operation units can be cascaded together to apply multiple textures to a common primitive in eight passes (multipass) or, if the graphics hardware supports it, in one single pass (multitexturing). The results of each stage are carried over to the next one, and the result of the final stage is rasterized on the polygon.

Most of the current 3-D hardware will only support applying two textures at the same time to a common primitive. Newer hardware—such as the ATI Radeon—handles three texture operations at once. Caution: Old 3-D hardware won't support multitexturing at all; this example won't run on those cards (you'll see an informative message box :-)).

Let's examine the color operations of the SetTextureStageState() method.

## COLOR OPERATIONS

Color operations are set with D3DTSS\_COLOROP. An abstract statement using color operations might look like

```
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, arg );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, arg );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP,     op );
```

The ATI Radeon supports the following op parameters, which are located in the D3DTEXTURESTAGESTATETYPE structure.

- D3DTOP\_DISABLE
- D3DTOP\_SELECTARG1
- D3DTOP\_SELECTARG2
- D3DTOP\_MODULATE
- D3DTOP\_MODULATE2X
- D3DTOP\_MODULATE4X
- D3DTOP\_ADD
- D3DTOP\_ADDSIGNED
- D3DTOP\_ADDSIGNED2X
- D3DTOP\_SUBTRACT
- D3DTOP\_BLENDDIFFUSEALPHA
- D3DTOP\_BLENDTEXTUREALPHA
- D3DTOP\_BLENDFACTORALPHA
- D3DTOP\_BLENDTEXTUREALPHAPM
- D3DTOP\_BLENDCURRENTALPHA
- D3DTOP\_MODULATEALPHA\_ADDCOLOR
- D3DTOP\_MODULATEINVALPHA\_ADDCOLOR

- D3DTOP\_BUMPENVMAP
- D3DTOP\_DOTPRODUCT3
- D3DTOP\_MULTIPLYADD
- D3DTOP\_LERP

Every color operation can get up to three arguments (arg): D3DTSS\_COLORARG0, D3DTSS\_COLORARG1, and D3DTSS\_COLORARG2 (ATI Radeon supports mapping three textures at once). These color arguments can be set to any combination of

- **D3DTA\_DIFFUSE.** Use the diffuse color interpolated from vertex components during Gouraud shading.
- **D3DTA\_SPECULAR.** Use the specular color interpolated from vertex components during Gouraud shading.
- **D3DTA\_TFACTOR.** Use the texture factor set in a previous call to the SetRenderState() with the D3DRS\_TEXTUREFACTOR render-state value.
- **D3DTA\_TEXTURE.** Use the texture color for this texture stage.
- **D3DTA\_CURRENT.** Use the result of the previous blending stage.
- **D3DTA\_TEMP.** Use a temporary register color for read or write.

One or both of the D3DTA\_ALPHAREPLICATE and D3DTA\_COMPLEMENT modifier flags may be bitwise or'ed in to alpha-replicate and/or invert the argument. Let's examine real-world examples.

## DARK MAPPING

Most effects that modify the appearance of a surface are calculated on what's called a *per-vertex* basis. This means that the actual calculations are done for each vertex of a triangle, as opposed to each pixel that gets rendered. Sometimes with this technique you get noticeable artifacts. Think of a large triangle with a light source close to the surface. As long as the light is close to one of the vertices of the triangle, you can see the lighting effects on the triangle: when it moves towards the center of the triangle, then the triangle gradually loses the lighting effect. In the worst case, the light is directly in the middle of the triangle and you see a triangle with very little shining on it, instead of a triangle with a bright spot in the middle. If no light shines on the vertices, the surface properties are not calculated in properly.

The best way to generate the illusion of pixel-based lighting is to use a texture map of the desired type of light shining on a dark surface.

Multiplying the colors of two textures together is called *light mapping*, or sometimes *dark mapping*, because it's often used to darken the texture. To set a dark map, we have to use it in the Render() method with the following commands:

```
// Set texture for the cube
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,    D3DTOP_SELECTARG1 );
```

```
// Set the dark map
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP,    D3DTOP_MODULATE );
```

### In Detail:

```
// first texture operation unit
// Associate texture with the first texture stage
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
// use texture coordinate pair #1
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
// Set the first color argument to the texture associated with this stage
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
// Use this texture stage's first color unmodified, as the output.
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,    D3DTOP_SELECTARG1 );

// second texture operation unit
// Associate texture with the second texture stage
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
// use texture coordinate pair #1
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 0 );
// use the texture color from this texture stage.
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
// Set the second color argument to the output of the last texture stage
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );
// multiply result of stage 1 with result of stage 2
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP,    D3DTOP_MODULATE );
```

This code combines the following textures illustrated in Figure 8.2:

This kind of multitexturing is called dark mapping because the resulting texel is a darker version of the unlit texel of the primary map. This technique is used a lot in 3-D shooters. You can see it in GLQuake1.

For RGB color, the render states D3DTSS\_COLORARG1 and D3DTSS\_COLORARG2 control arguments, while D3DTSS\_COLOROP controls the operation on the arguments.

The first texture operation unit passes the data from texture 0 to the next stage. The control argument D3DTA\_TEXTURE means that the texture argument is the texture color for this texture stage. The second texture operation unit receives these texels via Arg2. It modulates (= D3DTOP\_MODULATE) the texels from texture 0 with the texels from texture 1, which were received via Arg1.

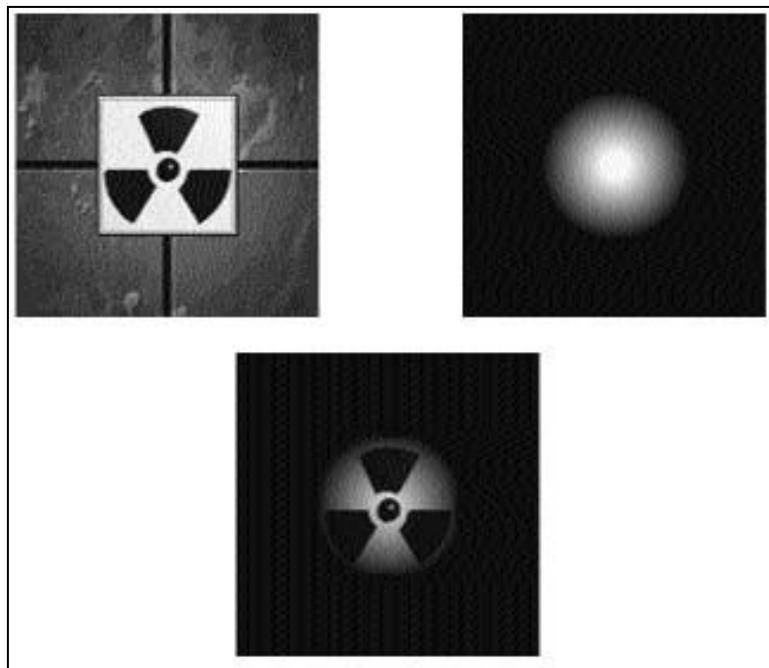


Figure 8.2: D3DTOP\_MODULATE

There are two other modulation operations:

D3DTOP_MODULATE	Multiply the components of the arguments together. Short: <code>BaseMap * LightMap</code>
D3DTOP_MODULATE2X	Multiply the components of the arguments and shift the products to the left 1 bit (effectively multiplying them by 2) for brightening. Short: <code>BaseMap * LightMap &lt;&lt; 1</code>
D3DTOP_MODULATE4X	Multiply the components of the arguments and shift the products to the left 2 bits (effectively multiplying them by 4) for brightening. Short: <code>BaseMap * LightMap &lt;&lt; 2</code>

The default value for the first texture stage (stage 0) is D3DTOP\_MODULATE, and for all other stages the default is D3DTOP\_DISABLE.

The `SetTexture()` method assigns a texture to a given stage for a device. The first parameter must be a number in the range of 0–7, inclusive. Pass the texture interface pointer as the second parameter. This method increments the reference count of the texture surface being assigned. When the texture is no longer needed, you should set the texture at the appropriate stage to NULL. If you fail to do this, the surface will

not be released, resulting in a memory leak. Since version 6.0, Direct3D is capable of blending up to eight current textures at once.

When your application selects a texture as the current texture, it instructs the Direct3D device to apply the texture to all primitives that are rendered from that time until the current texture is changed again. If each primitive in a 3-D scene has its own texture, the texture must be set before each primitive is rendered.

The code above shows the use of multitexturing; now let's see the whole thing with multipass rendering.

### NOTE

The reference device or driver supports mapping up to eight textures at once.

### NOTE

Textures under the **IDirect3D2** interface were manipulated using texture handles. With the **IDirect3D7** interface (and the legacy **IDirect3D3** interface), you create and use textures through interface pointers to the texture surfaces. You obtain a texture surface interface pointer when you create the texture surface by calling the **IDirectDraw7::CreateSurface**, which is called in the framework by the **D3Dtextr\_CreateTextureFromFile()** method.

```
// Set texture for the cube
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
// draw polygon
// Set darkmap
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE );
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ZERO );
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_SRCCOLOR );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTexture(0, m_pEnvTexture );
// draw polygon
```

The alpha-blending formula looks like:

```
FinalColor = SourcePixelColor * 0.0 + DestPixelColor * SourcePixelColor
```

It should mimic the modulate color operation used in the multitexture rendering code.

## ANIMATING THE DARK

I made an animated sample with the three modulation types:

```
// animate darkmap
if (i < 40)
{
    m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE);
}
else if (i < 80)
{
    m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE2X);
}
else if (i < 120)
{
    m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE4X);
}
else if (i = 120)
{
    i = 0;
}
i++;
```

This is a quick hack to show a simple effect. This effect is not time-based. In the first 40 frames, SetTextureStage() uses D3DTOP\_MODULATE, in the next 40 frames it uses D3DTOP\_MODULATE2X, and in the next 40 frame it uses D3DTOP\_MODULATE4X. So the effect gets brighter in two steps until 120 frames are shown. Then the whole thing starts again.

## BLENDING A TEXTURE WITH MATERIAL DIFFUSE COLOR

Sometimes the sun shines so bright that the colors on things get brighter. You can imitate that effect by blending the texture with the material diffuse color:

How did I get such an effect?

A directional light is reflected by a white material. More reflection means less texture color. So the cube appears white at places where the light shines directly on its side.



**Figure 8.3:** Light color plus texture color

### NOTE

There are three ways to get a diffuse color: from material, vertex diffuse color, and vertex specular color. D3DRS\_COLORVERTEX is by default switched on. So where the color is taken from depends on the D3DRS\_DIFFUSEMATERIALSOURCE parameter in the RenderState() method:

```
m_pd3dDevice->SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE,  
                                D3DMCS_MATERIAL);
```

D3DMCS\_MATERIAL means to use the color from the current material. But D3DMCS\_COLOR1 uses the diffuse vertex color, if specified in the FVF flags and structure. If you haven't specified one, opaque white is taken. D3DMCS\_COLOR2 uses the specular vertex color. If you haven't chosen one, the default is opaque white.

If you haven't set a D3DMCS\_x parameter with RenderState() at all, the material diffuse color is taken as the default. That's the case in this example.

The material is set with calls to the following methods:

```
D3DMATERIAL8 Material;
D3DUtil_InitMaterial( Material, 1.0f, 1.0f, 1.0f, 1.0f );
m_pd3dDevice->SetMaterial( &Material );
```

You set a directional light that is located above the cube with SetLight() and LightEnable(0, TRUE):

```
HRESULT CMyD3DApplication::SetLights()
{
    if (m_bTex3 == TRUE || m_bTex4 == TRUE)
    {
        m_pd3dDevice->SetRenderState(D3DRS_AMBIENT, 0);
        D3DLIGHT8 light;
        D3DUtil_InitLight( light, D3DLIGHT_DIRECTIONAL, 0.0f,
-5.0f, -5.0f );
        m_pd3dDevice->SetLight( 0, &light );
        m_pd3dDevice->LightEnable( 0, TRUE );
    }
    else if (m_bTex7 == TRUE)
    {
        m_pd3dDevice->LightEnable( 0, FALSE );
        // Set the ambient light.
        m_pd3dDevice->SetRenderState(D3DRS_AMBIENT, 0x00aaffaa);
    }
    return S_OK;
}
```

The code might look like this:

```
// Set texture for the cube
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_ADD );
```

In detail:

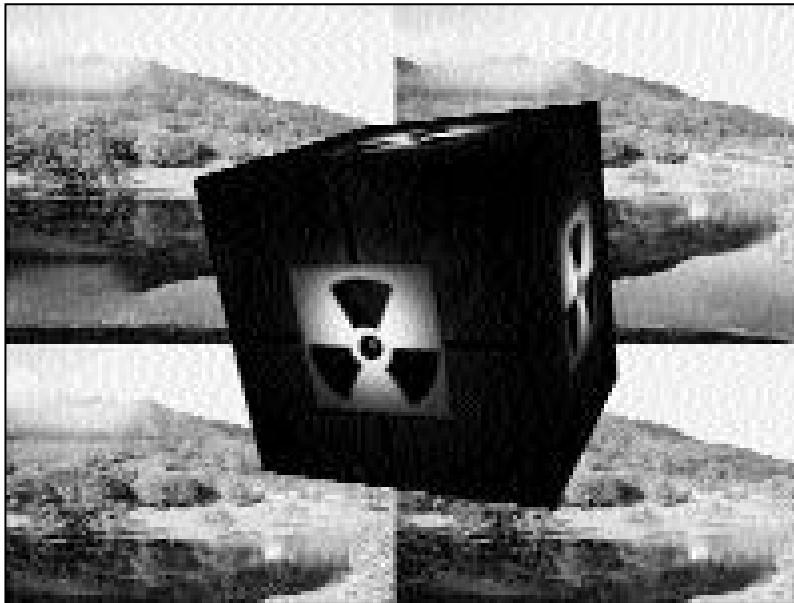
```
// Associate texture with the first texture stage
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
// use the texture color from this texture stage.
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
// use the diffuse lighting information produced by gouraud shading
```

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
// add texture color and diffuse color
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_ADD );
```

As you might see, this effect only needs one pass. So there's no multitexture code here. The short description looks like: BaseMap + DiffuseInterpolation.

### DARK MAP BLENDED WITH MATERIAL DIFFUSE COLOR

You are standing in a very dark room, and you cannot see the colors of things that are standing around. You switch on your lighter. Suddenly, colors of things appear as they are lighted by the flame. This is the basic idea for the following effect.



**Figure 8.4:** (Light Color \* Wall Texture) \* Dark Map

We're combining the dark mapping with the blending a texture with material diffuse color effect here:

```
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
// Set darkmap
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
```

```
m_pd3dDevice->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE);
```

In detail:

```
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
// use texture coordinate pair #1
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
// use as first argument the texture color of this texture
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
// use as the second argument the color of the material
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
// multiply both
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE);
```

If there's no light, the multiplication will be the texture color multiplied by 0, so the wall texture is not visible. In this case, only the second texture is visible.

```
// Set darkmap
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
m_pd3dDevice->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0 );
// use the texture color of this stage
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
// use the result of the previous texture stage = tex color X light color
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
// multiply the result of the previous texture with the color of this texture
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE);
```

In case the first texture is invisible, the multiplication of the 0 value of the first stage with the color of second stage will also result in black. So the first texture color fades to black when no light shines on that side of the cube. This might be useful in really dark places where you are not able to see the colors of things because of the weak light.

You might shorten that to (Base Map \* Light Color) \* Dark Map.

The multipass rendering version could look like this:

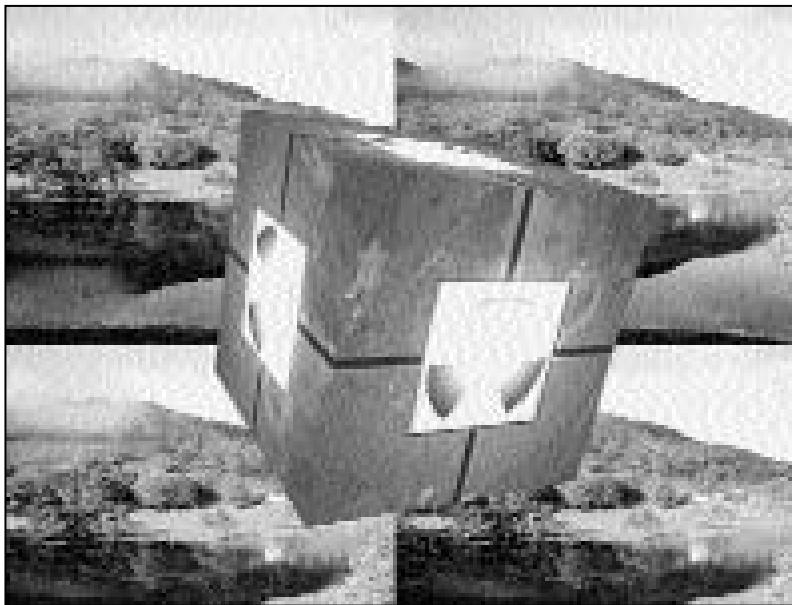
```
// Set texture for the cube
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
```

```
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
// draw polygon
// Set darkmap
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ZERO);
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_SRCCOLOR);
m_pd3dDevice->SetTexture(0, m_pEnvTexture);
// draw polygon
```

The alpha blending part is the same as used in the dark mapping example.

## GLOW MAPPING

The opposite of dark mapping is glow mapping. It's useful for creating objects that have glowing parts independently from the base map, such as LEDs, buttons, and lights in buildings or on spaceships.



**Figure 8.5:** Glow mapping

The glow map should have no effect on the base map except on the glowing area, so we have to add the glow effect and not modulate it.

```
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
```

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
// Set glow map
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADD);
```

The color of the first texture stage is used unmodified as the output with D3DTOP\_SELECTARG1 in the next texture stage. D3DTOP\_ADD adds the color of the second texture stage.

A multipass version might look like this:

```
// Set texture for the cube
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_SELECTARG1 );
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
// draw polygon
// Set darkmap
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
// draw polygon
```

The alpha-blending formula looks like this:

```
FinalColor = SourcePixelColor * 1.0 + DestPixelColor * 1.0
```

This simulates the D3DTOP\_ADD functionality in the multitexture code.

## DETAIL MAPPING

If your game is targeted to high-end user systems, you might use an additional texturing pass for detail mapping.

As you see, the wall looks rougher, like a rough plaster. What's the deal with this?

Think of the following scene: You move with your nice little armory through a level. A monster appears, and you crouch behind one of those crates one normally finds on foreign planets. While thinking about your destiny as a hero who saves worlds with a jigsaw, you look at the wall right in front of you. It shows a blurred texture. This texture looked perfectly normal at a distance of a footstep. But if your nose feels that cold rusted metal that is always on the walls of the cellars on those foreign worlds, the texture blurs.

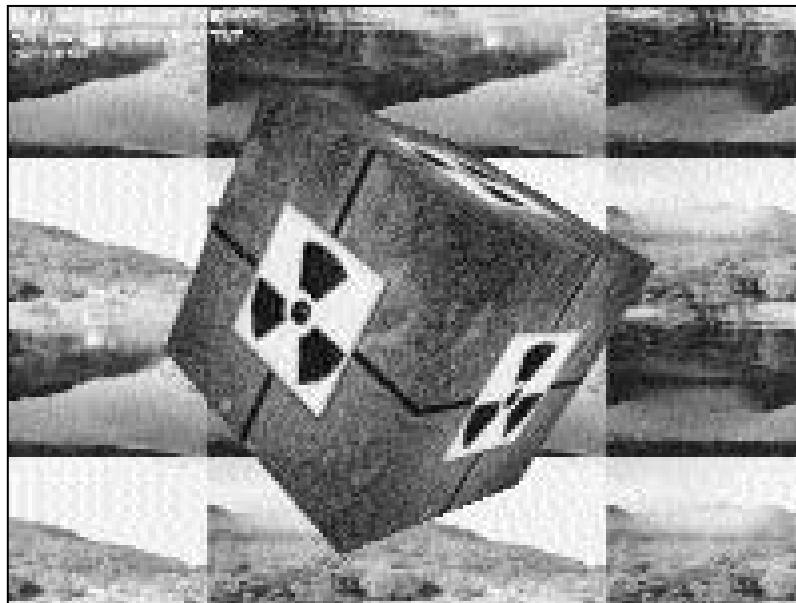


Figure 8.6: Detail mapping

OK . . . if your nose is not located on your face . . . let's say you're standing in front of a wall and you're really near to that wall with your eyes.

Back to the real world: You might sharpen the visual appearance of this wall by adding an additional detail texture, which happens to show up in these cases. So what does this detail texture do?

The color of the base map, or first texture, is used unmodified in the second texture stage as the second argument. The detail texture, which is a gray-colored texture, will be added to the base map with D3DTOP\_ADDSIGNED. It essentially does an addition, having one of the textures with signed color values (-127 . . . 128) instead of the unsigned values (0 . . . 255) that we're used to. In the ADDSIGNED texture, -127 is black and 128 is white. It adds the components of the arguments with a -0.5 bias, making the effective range of values from -0.5 through 0.5.

So the lighter gray texels in the detail map will brighten the base map, and the darker gray texels will darken it. The wall will show a rougher surface and will appear, therefore, more realistic. Detail mapping is used with the following source piece:

```
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
```

### TIP

Because D3DTOP\_ADDSIGNED is not supported on every device, a replacement might be D3DTOP\_MODULATE2X. But that will not look as good as ADDSIGNED, because it washes out the colors.

```
// Set detail map
m_pd3dDevice->SetTexture(1, m_pDetailTexture);
m_pd3dDevice->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADDSIGNED);
```

Doing that with multipass rendering looks like this:

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
m_pd3dDevice->SetTexture( 0, m_pWallTexture );
// draw polygon
// Set darkmap
m_pd3dDevice->SetRenderState(D3DRENDERSTATE_ALPHABLENDENABLE, TRUE);
m_pd3dDevice->SetRenderState(D3DRENDERSTATE_SRCBLEND, D3DBLEND_DESTCOLOR);
m_pd3dDevice->SetRenderState(D3DRENDERSTATE_DESTBLEND, D3DBLEND_SRCCOLOR);
m_pd3dDevice->SetTexture(1, m_pDetailTexture);
// draw polygon
```

The alpha blending formula should look like:

```
FinalColor = SourcePixelColor * DestPixelColor + DestPixelColor * SourcePixelColor
```

This one simulates the D3DTOP\_ADDSIGNED functionality of the multitexture code. How does that work? A replacement for ADDSIGNED is MODULATE2X. If you build a formula for MODULATE2X, it might look like this:

```
FinalColor = 2 * Arg1 * Arg2
FinalColor = Arg1 * Arg2 + Arg2 * Arg1
FinalColor = SourcePixelColor * DestPixelColor + DestPixelColor * SourcePixelColor
```

It's a pity that alpha blending costs a lot of CPU cycles. So switch it off.

At least a few words on the color operations that I have not covered with source here:

- D3DTOP\_BLENDDIFFUSEALPHA
- D3DTOP\_BLENDTEXTUREALPHA
- D3DTOP\_BLENDFACTORALPHA
- D3DTOP\_BLENDTEXTUREALPHAPM
- D3DTOP\_BLENDCURRENTALPHA

The texture stage operation results in the linear blending of both color operations with the

- Iterated diffuse alpha
- Current iterated texture alpha
- Scalar alpha (set with D3DRS\_TFACTOR)
- Premultiplied texture alpha
- Alpha that resulted from the previous stage

ATI uses the D3DTOP\_BLENDCURRENTALPHA and the D3DTOP\_BLENDFACTORALPHA parameters in its Radeon 1.3 SDK to show the effect of frosted glass. I haven't found any real-world examples for the other parameters.

D3DTOP\_MODULATEALPHA\_ADDCOLOR modulates the color of the second argument, using the alpha of the first argument; it then adds the result to argument 1.

$$\text{Result} = \text{Arg1}_{\text{RGB}} + \text{Arg1}_{\text{Alpha}} * \text{Arg2}_{\text{RGB}}$$

- D3DTOP\_MODULATEINVALPHA\_ADDCOLOR uses the inversed alpha to produce a result.

$$\text{Result} = \text{Arg1}_{\text{RGB}} + (1 - \text{Arg1}_{\text{Alpha}}) * \text{Arg2}_{\text{RGB}}$$

The following two parameters work only with cards with multitexture support for three texture stages. So they are called *triadic texture blending members*. As I write this, there's only one card that supports blending three texture stages at once: the ATI Radeon. The announced NVIDIA cards should support it, too.

D3DTOP\_MULTIPLYADD performs a multiply-accumulate operation. It takes the last two arguments, multiplies them together, adds them to the remaining input/source argument, and places that into the result.

$$\text{Result} = \text{Arg1} + \text{Arg2} * \text{Arg3}$$

D3DTOP\_LERP linearly interpolates between the second and third source arguments by a proportion specified in the first source argument.

$$\text{Result} = (\text{Arg1}) * \text{Arg2} + (1-\text{Arg1}) * \text{Arg3}$$

Now on to the alpha operations.

## ALPHA OPERATIONS

As with the color operations, I'll concentrate on the alpha operations most cards support. An abstract statement using color operations might look like this:

```
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_ALPHAARG1, arg );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_ALPHAARG2, arg );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_ALPHAOP,     op );
```

The op parameter might be

- D3DTOP\_DISABLE
- D3DTOP\_SELECTARG1
- D3DTOP\_SELECTARG2
- D3DTOP\_MODULATE
- D3DTOP\_MODULATE2X
- D3DTOP\_MODULATE4X
- D3DTOP\_ADD
- D3DTOP\_ADDSIGNED
- D3DTOP\_ADDSIGNED2X
- D3DTOP\_SUBTRACT
- D3DTOP\_MULTIPLYADD
- D3DTOP\_LERP

Every alpha operation might get up to three arguments (arg): D3DTSS\_ALPHAARG0, D3DTSS\_ALPHAARG1, and D3DTSS\_ALPHAARG2 (ATI Radeon supports mapping three textures at once). These alpha arguments can be set to any combination of

- **D3DTA\_DIFFUSE.** Use the diffuse color interpolated from vertex components during Gouraud shading.
- **D3DTA\_SPECULAR.** Use the specular color interpolated from vertex components during Gouraud shading.
- **D3DTA\_TFACTOR.** Use the texture factor set in a previous call to the SetRenderState() with the D3DRS\_TEXTUREFACTOR render-state value.
- **D3DTA\_TEXTURE.** Use the texture color for this texture stage.
- **D3DTA\_CURRENT.** Use the result of the previous blending stage.
- **D3DTA\_TEMP.** Use a temporary register color for read or write.

The D3DTA\_COMPLEMENT modifier flag may be bitwise or'ed in to invert the argument.

You know all of these arguments from the color operations part.

As Direct3D renders a scene, it can integrate color information from several sources: vertex color, the current material, the texture map, and the color previously written to the render target. It can blend several of these colors. A factor called alpha, which can be stored in vertices, materials, and texture maps, can be used to indicate how blending should be weighted. An alpha value of 0 means full transparency; an alpha value of 1 means some level of semitransparency.

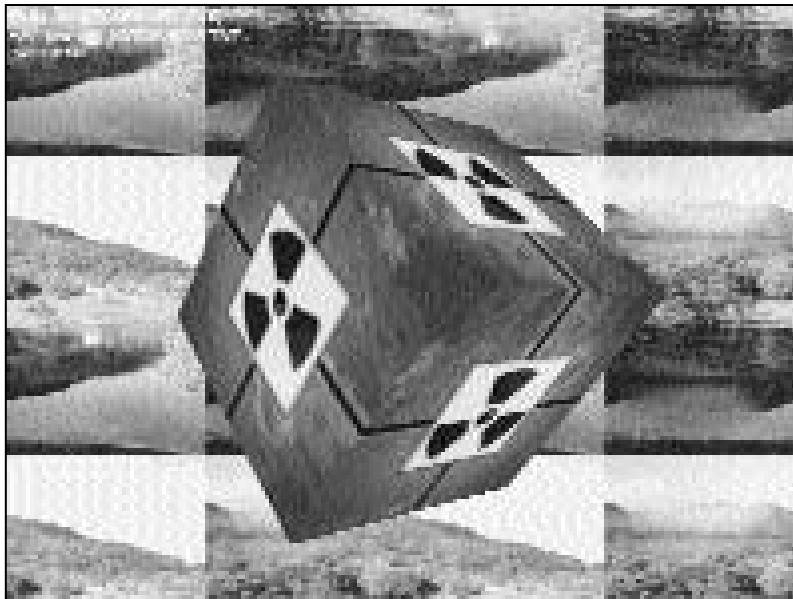
If you want to fetch alpha from a texture, use D3DTA\_TEXTURE as arg. If you want to use alpha that comes from the color component within a vertex (and interpolated across the polygon), use D3DTA\_DIFFUSE as alphaarg and ensure D3DRS\_DIFFUSEMATERIALSOURCE = D3DMCS\_COLOR1 (default value). If you want to use alpha that comes from the material color, use D3DTA\_DIFFUSE as alphaarg and ensure D3DRS\_DIFFUSEMATERIALSOURCE = D3DMCS\_MATERIAL is set.

```
m_pd3dDevice->SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE ,  
                               D3DMCS_MATERIAL);
```

If you haven't set a D3DMCS\_x parameter with RenderState() at all, the material diffuse color is taken as the default. That's the case in the following example.

## MODULATE ALPHA

In 1993 I played Comanche from NovaLogic for the first time. I wondered about making the night flight feature in some missions. The whole terrain, the clouds, and the horizon were green (it's done via palette animation: thanks for the bunch of e-mails). They looked like modulating a green ambient light with the material alpha.



**Figure 8.7:** Modulate alpha

So I use a green ambient light, which is reflected by the material color, to modulate the light with the texture color and the alpha value:

```
// Set the ambient light.  
D3DCOLOR d3dclrAmbientLightColor = D3DRGBA(0.0f,1.0f,0.0f,1.0f);  
m_pd3dDevice->SetRenderState(D3DRENDERSTATE_AMBIENT, d3dclrAmbientLightColor);
```

To modulate the ambient color with the texture:

```
m_pd3dDevice->SetTexture( 0, D3DTextr_GetSurface("wall.bmp"));  
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );  
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );  
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );  
  
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );  
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE );  
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_MODULATE );
```

`SetTexture()` associates the texture map with the first stage. The first `SetTextureStageState()` sets the first color argument to the texture associated with this stage. The second sets the second color argument to diffuse lighting information. The color of the base map and the diffuse color is multiplied. The alpha-blending code takes the alpha value from the texture and multiplies it with the diffuse color.

A short description could look like: BaseMap  $\times$  DiffuseInterpolation  $\times$  Alpha  $\times$  DiffuseInterpolation.

## ENVIRONMENT MAPPING

Environment mapping is a method to reflect the environment onto a polygon. Environment maps can be used to create both objects that appear to reflect their surroundings and objects that look like specular highlights are falling on them. These two methods are called *spherical* and *cubic environment mapping*.

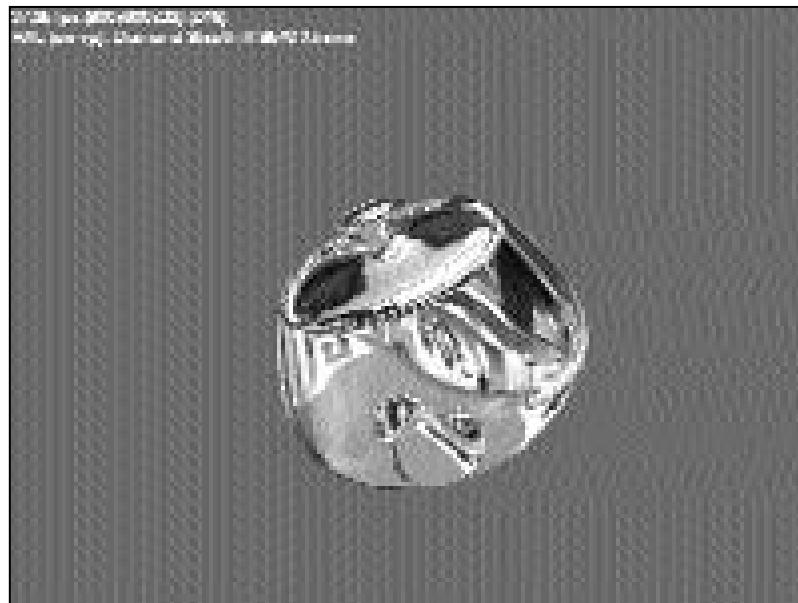
### SPHERICAL ENVIRONMENT MAPPING

Spherical environment mapping uses a single texture map that contains an image of the environment. To use the environment map to light a primitive, you can blend it with the base map by adding or multiplying them both. If you don't want to light the primitive in this way, you might use the environment map as the primitive's texture.

To achieve the appearance of reflection, you have to orient the environment map in a way that depends on the orientation of the primitive relative to the viewer. If you would like to use environment mapping for a primitive to reflect its surroundings, then you have to provide a "fish-bowl" image of these surroundings.

Because I can't think of a more instructive example than the one provided with the DirectX 8.0 SDK called Sphere Map, I'll use that one here.

How does it work? The sphere map is always oriented toward the viewer, so the texture coordinates have to be changed in real-time relative to the vertices. The idea is to consider each vertex, compute its normal, find where the normal matches on the chrome sphere, and then assign that texture coordinate to the vertex.



**Figure 8.8:** Sphere mapping  
(example from Microsoft)

The DirectX 8.0 SDK example is really straightforward. Most of the stuff happens in the following lines:

```
// Generate spheremap texture coords, and shift them over
D3DXMATRIX mat;
mat._11 = 0.5f; mat._12 = 0.0f; mat._13 = 0.0f; mat._14 = 0.0f;
mat._21 = 0.0f; mat._22 = -0.5f; mat._23 = 0.0f; mat._24 = 0.0f;
mat._31 = 0.0f; mat._32 = 0.0f; mat._33 = 1.0f; mat._34 = 0.0f;
mat._41 = 0.5f; mat._42 = 0.5f; mat._43 = 0.0f; mat._44 = 1.0f;
m_pd3dDevice->SetTransform( D3DTS_TEXTURE0, &mat );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
                                         D3DTFF_COUNT2 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
                                         D3DTSS_TCI_CAMERASPACENORMAL );
```

The only texture that is used in this example is the environment map that is mapped onto the teapot. Direct3D can transform the texture coordinates for vertices by applying a 4-by-4 matrix, the way it transforms geometry. So it can scale, rotate, translate, project, or shear a texture map or apply any combination of these. Additionally, you might reuse a transformed texture coordinate set for multiple purposes in multiple stages.

You set matrices for texture coordinate transformations like you set them for geometry transformations with `SetTransform()`. The main difference is the use of the `D3DTS_TEXTURE0` through `D3DTS_TEXTURE7` members of the `D3DTRANSFORMSTATETYPE` enumerated type to identify the transformation matrices for texture stages 0 through 7, respectively. By setting `D3DTSS_TEXTURETRANSFORMFLAGS` to `D3DTFFF_COUNT2`, you enable the texture coordinate transformation by instructing the system to pass the first two elements of the modified texture coordinates to the rasterizer.

The most important render state operation is `D3DTSS_TCI_CAMERASPACENORMAL`, which causes the system to use the vertex normal, transformed to camera space as the input texture coordinates.

There are two drawbacks with sphere mapping:

- Warping effects occur at the edges of the map, because it has a limited resolution.
- Sphere maps are not easy to generate in real time, so they always reflect a static scene.

### CAUTION

**There are precision issues that lead to noticeable artifacts. While texels in the center of the sphere map correspond to relatively small changes in normal direction, along the edges there are big changes and an infinite change when you reach the edge of the polygon. But sphere mapping is usually faster than cubic environment mapping and suffices a lot of needs.**

### TIP

**You may find a few special effects in the chapter Using Texture Coordinate Processing in the DirectX 8.0 SDK documentation. Note: Texture transformation does not work with transformed and lit vertices.**

### TIP

**If you play around with this sample, try to switch on the glass effect with `m_bGlassEffect` ... really nice.**

## CUBIC ENVIRONMENT MAPPING

Direct3D supports cubic environment mapping since the advent of version 7.0. This kind of mapping doesn't suffer from the warping artifacts that plague spherical environment maps and can reflect changing environments. Cube maps are also generated in real time, so they can show changing environments.

I'll use the example provided with the DirectX 8.0 SDK, which is really instructive and the simplest example I can think of. This example loads a cube that has six different textures inside of it. The six textures show a scene from one point in six different angles. Inside of that cube floats a teapot that is surrounded by an airplane (here's the tale: the cube is made by Borgs and the teapot hovers in an antigravity field . . . the Klingon pilot of the plane flies with an autopilot constructed by dwarfs living on planet Cave, who hate flying or being thrown. As we all know, the pilot is angry).



**Figure 8.9:** Cube mapping  
(example from Microsoft)

The reflections on the teapot are made by rendering the whole scene into a cube map from different camera views. This map is applied to the teapot. The camera is located at the place where the teapot resides. The cubic environment map consists of six square textures loaded into a separate surface for each one.

Down to code.

#### RESTOREDEVICEOBJECTS()

The example loads the six square texture surfaces with a call to CreateCubeTexture() in RestoreDeviceObjects().

```
if( FAILED( hr = m_pd3dDevice->CreateCubeTexture( CUBEMAP_RESOLUTION, 1,
D3DUSAGE_RENDERTARGET,
m_d3dsdBackBuffer.Format,
D3DPPOOL_DEFAULT, &m_pCubeMap ) ) )
    return E_FAIL;
```

We have to create a stencil buffer, because the user could resize the rendering window so that it is smaller than a cube face. In this case we cannot render into the window using the normal z-buffer.

```
if( FAILED( hr = m_pd3dDevice->CreateDepthStencilSurface(CUBEMAP_RESOLUTION,
```

```
CUBEMAP_RESOLUTION,
D3DFMT_D16,
D3DMULTISAMPLE_NONE,
&m_pCubeFaceZBuffer) ) )

return E_FAIL;
```

Nothing else is new to you in the `RestoreDeviceObjects()` method.

There are three Render methods in this example: the usual `Render()` method, which is called in the framework built by the methods of the Common files; a `RenderScene()`; and a `RenderSceneIntoCube()` method.

### `RENDERSCENEINTOCUBE()`

`RenderSceneIntoCube()` is called by `FrameMove()`. It works in two steps: it renders the camera view from every different angle into the six cube map surfaces, then it uses `RenderScene()` to render the scene without the teapot into the frame buffer. The teapot is rendered later with a call to `Render()`, which uses `RenderScene()` with the `bRenderTeapot` flag to true.

```
HRESULT CMyD3DApplication::RenderSceneIntoCubeMap()
{
    // Save transformation matrices of the device
    D3DXMATRIX matProjSave, matViewSave;
    m_pd3dDevice->GetTransform( D3DTS_VIEW, &matViewSave );
    m_pd3dDevice->GetTransform( D3DTS_PROJECTION, &matProjSave );
    // Set the projection matrix for a field of view of 90 degrees
    D3DXMATRIX matProj;
    D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/2, 1.0f, 0.5f, 100.0f );
    m_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
    // Get the current view matrix, to concat it with the cubemap view vectors
    D3DXMATRIX matViewDir;
    m_pd3dDevice->GetTransform( D3DTS_VIEW, &matViewDir );
    matViewDir._41 = 0.0f; matViewDir._42 = 0.0f; matViewDir._43 = 0.0f;
```

### NOTE

A stencil buffer is an extra per-pixel test and set of update operations that are closely coupled with the depth test. In addition to the color and depth bit-planes for each pixel, stenciling adds additional bit-planes to track the stencil value of each pixel. A good way to think of per-pixel stenciling is that stenciling means to “tag” pixels in one rendering pass to control their update in subsequent rendering passes. That is useful for applying shadows. We use the stencil buffer as a z-buffer to render into cube maps.

```
// Store the current backbuffer and zbuffer
LPDIRECT3DSURFACE8 pBackBuffer, pZBuffer;
m_pd3dDevice->GetRenderTarget( &pBackBuffer );
m_pd3dDevice->GetDepthStencilSurface( &pZBuffer );
// Render to the six faces of the cube map
for( DWORD i=0; i<6; i++ )
{
    // Set the view transform for this cubemap surface
    D3DXMATRIX matView;
    matView = D3DUtil_GetCubeMapViewMatrix( (D3DCUBEMAP_FACES)i );
    D3DXMatrixMultiply( &matView, &matViewDir, &matView );
    m_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );
    // Set the rendertarget to the i'th cubemap surface
    LPDIRECT3DSURFACE8 pCubeMapFace;
    m_pCubeMap->GetCubeMapSurface( (D3DCUBEMAP_FACES)i, 0, &pCubeMapFace );
    m_pd3dDevice->SetRenderTarget( pCubeMapFace, m_pCubeFaceZBuffer );
    pCubeMapFace->Release();
    // Render the scene (except for the teapot)
    m_pd3dDevice->BeginScene();
    RenderScene( FALSE );
    m_pd3dDevice->EndScene();
}
// Change the rendertarget back to the main backbuffer
m_pd3dDevice->SetRenderTarget( pBackBuffer, pZBuffer );
pBackBuffer->Release();
pZBuffer->Release();
// Restore the original transformation matrices
m_pd3dDevice->SetTransform( D3DTS_VIEW,           &matViewSave );
m_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProjSave );
return S_OK;
}
```

Let's examine the interesting parts in this code piece. What does this piece of code do?

1. Position a camera in the middle of the object.
2. Rotate the camera in the six directions necessary to build a cube.
3. Render the camera view into the six cube map surfaces.

D3DUtil\_GetCubeMapViewMatrix() in d3dutil.cpp—one of the Common files—returns the six different view matrices for the camera (it is also used in the fish-eye example). The camera is oriented with a call to D3DXMatrixMultiply() and SetTransform(). Before that, the user's view matrix is saved with a call to GetTransform().

SetRenderTarget() sets the cube map surface and the stencil buffer as the render targets:

```
m_pd3dDevice->SetRenderTarget( pCubeMapFace, m_pCubeFaceZBuffer );
```

The call to RenderScene() happens with bRenderTeapot = FALSE, because the teapot should be rendered later, with the user's view matrix. After the cube map is filled, the user's view matrix is restored with SetTransform().

That is the way the environment map, or cube map, is produced.

### RENDERSCENE()

The RenderScene() method renders the skybox and the plane into the cube map and renders both later into the frame buffer. So it's called twice every frame. The specific task depends on from which method it is called. If it's called from RenderSceneIntoCube(), then it renders the skybox and plane into the cube map. When it's called from within Render(), then it renders the skybox, plane, and teapot into the frame buffer.

```
HRESULT CMyD3DApplication::RenderScene( BOOL bRenderTeapot )
{
    // Clear the viewport
    m_pd3dDevice->Clear( 0L, NULL, D3DCLEAR_ZBUFFER, 0x000000ff, 1.0f, 0L );
    // Set the texture stage states
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,     D3DTOP_SELECTARG1 );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_MINFILTER, D3DTEXF_LINEAR );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_MAGFILTER, D3DTEXF_LINEAR );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ADDRESSU,  D3DTADDRESS_MIRROR );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ADDRESSV,  D3DTADDRESS_MIRROR );
    // Render the Skybox
    {
        // Save the current matrix set
        D3DXMATRIX matViewSave, matProjSave;
        m_pd3dDevice->GetTransform( D3DTS_VIEW,           &matViewSave );
        m_pd3dDevice->GetTransform( D3DTS_PROJECTION, &matProjSave );
        // Disable zbuffer, center view matrix, and set FOV to 90 degrees
        D3DXMATRIX matView = matViewSave;
        D3DXMATRIX matProj = matViewSave;
        matView._41 = matView._42 = matView._43 = 0.0f;
        D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/2, 1.0f, 0.5f, 10000.0f );
        m_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
        m_pd3dDevice->SetTransform( D3DTS_VIEW,       &matView );
        m_pd3dDevice->SetRenderState( D3DRS_ZENABLE, FALSE );
```

```

// Render the skybox
D3DXMATRIX matWorld;
D3DXMatrixIdentity( &matWorld );
m_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );
m_pSkyBox->Render( m_pd3dDevice );
// Restore the render states
m_pd3dDevice->SetTransform( D3DTS_VIEW,           &matViewSave );
m_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProjSave );
m_pd3dDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
}

// Render the main file-based object
{
    m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matAirplane );
    m_pAirplane->Render( m_pd3dDevice );
}

// Render the object with the environment-mapped body
if( bRenderTeapot )
{
    D3DXMATRIX matWorld;
    D3DXMatrixIdentity( &matWorld );
    m_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );
    // Turn on texture-coord generation for cubemapping
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTFF_COUNT3
);
    // Render the object with the environment-mapped body
    m_pd3dDevice->SetTexture( 0, m_pCubeMap );
    m_pShinyTeapot->Render( m_pd3dDevice );
    // Restore the render states
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_PASSTHRU );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTFF_DISABLE
);
}
return S_OK;
}

```

The most interesting parts of the source in the if(bRenderTeapot) statement are the two SetTextureStageState(). Texture coordinates of the cube map are set with D3DTSS\_TCI\_CAMERASPACEREFLECTIONVECTOR. This instructs the system to automatically generate texture coordinates as reflection vectors for cube mapping. This is easy for us programmers.

By setting D3DTSS\_TEXTURETRANSFORMFLAGS to D3DTFFF\_COUNT3, you enable the texture coordinate transformation by instructing the system to pass the three elements of the modified texture coordinates to the rasterizer. The cube map is set like a normal texture map with SetTexture().

Don't forget that we require hardware support for cube mapping.

### NOTE

**Getting the reflection vector is not a trivial task. You might find the formula Direct3D uses in Advanced 3-D Programming Using DirectX 7.0 (see the Additional Resources at the end of Part 2).**

### NOTE

**I didn't mention the whole x-file stuff here. You will absorb it in Part 3 of this book.**

### CONFIRMDDEVICE()

As usual, ConfirmDevice() checks the capabilities of the rendering device.

```
HRESULT CMyD3DApplication::ConfirmDevice( D3DCAPS8* pCaps, DWORD dwBehavior,
                                         D3DFORMAT Format )
{
    if( dwBehavior & D3DCREATE_PUREDEVICE )
        return E_FAIL; // GetTransform doesn't work on PUREDEVICE
    // Check for cubemapping devices
    if( 0 == ( pCaps->TextureCaps & D3DPTEXTURECAPS_CUBEMAP ) )
        return E_FAIL;
    // Check that we can create a cube texture that we can render into
    if( FAILED( m_pD3D->CheckDeviceFormat( pCaps->AdapterOrdinal,
                                             pCaps->DeviceType, Format, D3DUSAGE_RENDERTARGET,
                                             D3DRTYPE_CUBETEXTURE, Format ) ) )
    {
        return E_FAIL;
    }
    return S_OK;
}
```

If the device/graphics hardware driver is D3DCREATE\_PUREDEVICE, it does not support Get\* calls for anything that can be stored in state blocks. Because of GetTransform() the cube mapping example won't

run on such a device. The D3DPTEXTURECAPS\_CUBEMAP check is useful because this texture mode has to be supported by the device. The capability to render into a cube map texture is checked by the last call to CheckDeviceFormat().

## BUMP MAPPING

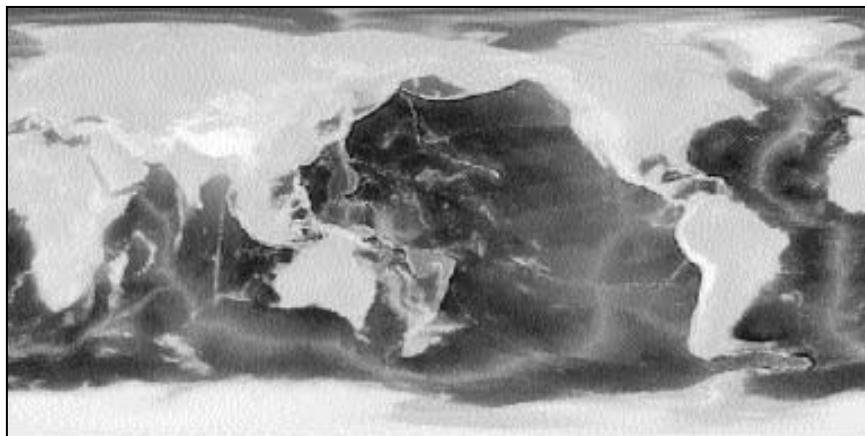
Bump mapping is a texture-blending method that models a realistic rough surface on primitives. The bump map contains depth information (tessellation) in the form of values indicating high and low spots on the surface. To be more precise: Bump mapping is a method for changing the visual appearance of a surface by using a different set of normals for lighting than the surface normals.

So what's a bump map? You need at least three texture maps to produce the effect of contour or bump mapping. In the case of the Bump Earth example in the DirectX 8.0 SDK, this is earth.bmp, earthbump.bmp, and earthenvmap.bmp. The first texture map holds the color information, the second texture map holds the height or contour information, and the third texture map holds the environment reflection. The second texture map is the bump map. It is used to bump the environment map's texture coordinates. So, bump mapping is just a special form of specular environment mapping (which you already know) that simulates the reflections of finely contoured objects.

But to be even more precise: A bump map is a height map that stores contour data. Each pixel in a bump map stores the delta values for a u/v coordinate pair and sometimes a luminance value. These values are used to fake the appearance of a bumpy surface by setting a different set of normals for lighting.



**Figure 8.10:** Bump mapping example (example by Microsoft)



**Figure 8.11:** Bump map  
(courtesy of Microsoft)

The best example to explain bump mapping is the example provided with the DirectX 8.0 SDK called Bump Earth.

Figure 8.11 is the height map image from the Bump Earth example.

We will concentrate on the two important methods in this example: `ApplyEnvironmentMap()` and `InitBumpMap()`.

The first method calculates the vertex normals that depend on the current world matrix to determine what the texture coordinates should be for the environment map. It is called by `FrameMove()` in every frame.

### APPLYENVIRONMENTMAP()

`ApplyEnvironmentMap()` fills the previously created vertex buffer with all the data that the `BUMPVERTEX` structure needs.

```
struct BUMPVERTEX
{
    D3DXVECTOR3 p;
    D3DXVECTOR3 n;
    FLOAT        tu1, tv1;
    FLOAT        tu2, tv2;
};
```

The whole method looks like this:

```
VOID CMyD3DApplication::ApplyEnvironmentMap()
{
    // Get the World-View(WV) matrix set
    D3DXMATRIX matWorld, matView, matWorldView;
```

```
m_pd3dDevice->GetTransform( D3DTS_WORLD, &matWorld );
m_pd3dDevice->GetTransform( D3DTS_VIEW, &matView );
D3DXMatrixMultiply( &matWorldView, &matWorld, &matView );
// Lock the vertex buffer
BUMPVERTEX* vtx;
m_pEarthVB->Lock( 0, 0, (BYTE**)&vtx, 0 );
// Establish constants used in sphere generation
DWORD dwNumSphereRings = m_bHighTesselation? 15 : 5;
DWORD dwNumSphereSegments = m_bHighTesselation? 30 : 10;
FLOAT fDeltaRingAngle = ( D3DX_PI / dwNumSphereRings );
FLOAT fDeltaSegAngle = ( 2.0f * D3DX_PI / dwNumSphereSegments );
D3DXVECTOR4 vt;
FLOAT fScale;
// Generate the group of rings for the sphere
for( DWORD ring = 0; ring < dwNumSphereRings; ring++ )
{
    FLOAT r0 = sinf( (ring+0) * fDeltaRingAngle );
    FLOAT r1 = sinf( (ring+1) * fDeltaRingAngle );
    FLOAT y0 = cosf( (ring+0) * fDeltaRingAngle );
    FLOAT y1 = cosf( (ring+1) * fDeltaRingAngle );
    // Generate the group of segments for the current ring
    for( DWORD seg = 0; seg < (dwNumSphereSegments+1); seg++ )
    {
        FLOAT x0 = r0 * sinf( seg * fDeltaSegAngle );
        FLOAT z0 = r0 * cosf( seg * fDeltaSegAngle );
        FLOAT x1 = r1 * sinf( seg * fDeltaSegAngle );
        FLOAT z1 = r1 * cosf( seg * fDeltaSegAngle );
        // Add two vertices to the strip which makes up the sphere
        // (using the transformed normal to generate texture coords)
        (*vtx).p = (*vtx).n = D3DXVECTOR3(x0,y0,z0);
        (*vtx).tu1 = (*vtx).tu2 = -(FLOAT)seg/dwNumSphereSegments;
        (*vtx).tv1 = (*vtx).tv2 = (ring+0)/(FLOAT)dwNumSphereRings;
        D3DXVec3Transform( &vt, &(*vtx).n, &matWorldView );
        fScale = 1.37f / D3DXVec4Length( &vt );
        (*vtx).tu1 = 0.5f + fScale*vT.x;
        (*vtx).tv1 = 0.5f - fScale*vT.y;
        vtx++;
        (*vtx).p = (*vtx).n = D3DXVECTOR3(x1,y1,z1);
        (*vtx).tu1 = (*vtx).tu2 = -(FLOAT)seg/dwNumSphereSegments;
        (*vtx).tv1 = (*vtx).tv2 = (ring+1)/(FLOAT)dwNumSphereRings;
        D3DXVec3Transform( &vt, &(*vtx).n, &matWorldView );
        fScale = 1.37f / D3DXVec4Length( &vt );
```

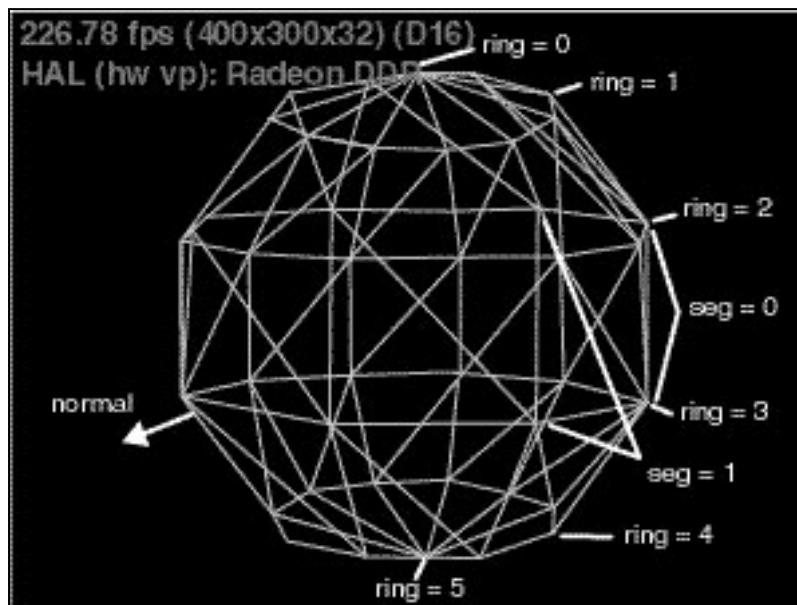
```

        (*vtx).tu1 = 0.5f + fScale*vT.x;
        (*vtx).tv1 = 0.5f - fScale*vT.y;
        vtx++;
    }
}

m_pEarthVB->Unlock();
}

```

Every ring of the sphere is located horizontally. In the low tessellation model, there are four rings consisting of 10 segments (the line between two (vtx).p points) that run vertically, and the north and south poles.



**Figure 8.12:** Vertices of the Bump Earth example (courtesy of Microsoft)

The use of the texture coordinate pairs to map the textures on the sphere is the interesting part. The tu2/tv2 texture coordinate pair is calculated with

```

(*vtx).tu2 = -((FLOAT)seg)/dwNumSphereSegments;
(*vtx).tv2 = (ring+0)/(FLOAT)dwNumSphereRings;

```

For example, the tu2 value for the sixth segment is  $-6/10$ ; that is,  $-0.6$ . The range for tv2 is 0.0 for seg = 0 until  $-1.0$  for seg = 10. The tv2 value for ring 0 is  $0/5$ ; that is, 0, whereas on ring 5, tv2 would be  $5/5$ ; that is, 1.0. So the tu2/tv2 pair (chosen by TEXCOORDINDEX = 1) maps the whole earth and bump map texture on the sphere. The code portion of Render() looks like this:

```
if( m_bTextureOn )
```

```

    m_pd3dDevice->SetTexture( 0, m_pEarthTexture );
else
    m_pd3dDevice->SetTexture( 0, m_pBlockTexture );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 1 );
...
if( m_bBumpMapOn && m_bEnvMapOn )
{
    m_pd3dDevice->SetTexture( 1, m_psBumpMap );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 1 );
...

```

The tu1/tv1 texture coordinate pair is calculated with the help of the normal and the world matrices. The normal gets the same values as the point, so it is pointing away from the middle of the sphere.

```

D3DXVec3Transform( &vT, &(*vtx).n, &matWorldView );
fScale = 1.37f / D3DXVec4Length( &vT );
(*vtx).tu1 = 0.5f + fScale*vT.x;
(*vtx).tv1 = 0.5f - fScale*vT.y;

```

This texture pair is used to map the environment map by choosing TEXCOORDINDEX = 0 on the sphere.

```

if( m_bEnvMapOn )
{
    m_pd3dDevice->SetTexture( 2, m_pEnvMapTexture );
    m_pd3dDevice->SetTextureStageState( 2, D3DTSS_TEXCOORDINDEX, 0 );
...

```

The texture coordinate pair is scaled and transformed in a similar way as that described above for sphere mapping.

To summarize: The `ApplyEnvironmentMap()` method builds the sphere and a bump map consisting of vertices with a

```

struct BUMPVERTEX
{
    D3DXVECTOR3 p;
    D3DXVECTOR3 n;
    FLOAT        tu1, tv1;
    FLOAT        tu2, tv2;
};

```

structure. These vertices hold two texture coordinate pairs, and the first texture coordinate pair helps in orienting the environment map toward the viewer.

## INITBUMPMAP()

The bump map is built after creating the vertex buffer. This happens long before the `ApplyEnvironmentMap()` is called in `FrameMove()`. `InitBumpMap()` is called once at `InitDeviceObjects()`. It builds the bump map from `m_pEarthBumpTexture`. This texture is loaded with

```
if( FAILED( D3DUtil_CreateTexture( m_pd3dDevice, _T("Earthbump.bmp"),
                                    &m_pEarthBumpTexture,
                                    D3DFMT_A8R8G8B8 ) ) )
```

The flag `D3DFMT_A8R8G8B8` told `D3DUtil_CreateTexture()` to load the texture in a 32-bit ARGB pixel format with alpha. The bitmap format is changeable by the user in the Options menu. `InitBumpMap()` builds a second empty texture called `m_psBumpMap`, where it copies the finished environment texture that holds the delta values `du` and `dv` and a luminance value in four 32-bit values.

```
HRESULT CMyD3DApplication::InitBumpMap()
{
    LPDIRECT3DTEXTURE8 psBumpSrc = m_pEarthBumpTexture;
    D3DSURFACE_DESC d3dsd;
    D3DLOCKED_RECT d3dlr;
    psBumpSrc->GetLevelDesc( 0, &d3dsd );
    // Create the bumpmap's surface and texture objects
    if( FAILED( m_pd3dDevice->CreateTexture( d3dsd.Width, d3dsd.Height, 1, 0,
                                              m_BumpMapFormat,
                                              D3DPPOOL_MANAGED,
                                              &m_psBumpMap ) ) )
    {
        return E_FAIL;
    }
    // Fill the bits of the new texture surface with bits from
    // a private format.
    psBumpSrc->LockRect( 0, &d3dlr, 0, 0 );
    DWORD dwSrcPitch = (DWORD)d3dlr.Pitch;
    BYTE* pSrc      = (BYTE*)d3dlr.pBits;
    m_psBumpMap->LockRect( 0, &d3dlr, 0, 0 );
    DWORD dwDstPitch = (DWORD)d3dlr.Pitch;
    BYTE* pDst      = (BYTE*)d3dlr.pBits;
    for( DWORD y=0; y<d3dsd.Height; y++ )
    {
        BYTE* pDstT  = pDst;
        BYTE* pSrcB0 = (BYTE*)pSrc;
```

```
BYTE* pSrcB1 = ( pSrcB0 + dwSrcPitch );
BYTE* pSrcB2 = ( pSrcB0 - dwSrcPitch );
if( y == d3dsd.Height-1 ) // Don't go past the last line
    pSrcB1 = pSrcB0;
if( y == 0 ) // Don't go before first line
    pSrcB2 = pSrcB0;
for( DWORD x=0; x<d3dsd.Width; x++ )
{
    LONG v00 = *(pSrcB0+0); // Get the current pixel
    LONG v01 = *(pSrcB0+4); // and the pixel to the right
    LONG vM1 = *(pSrcB0-4); // and the pixel to the left
    LONG v10 = *(pSrcB1+0); // and the pixel one line below.
    LONG v1M = *(pSrcB2+0); // and the pixel one line above.
    LONG iDu = (vM1-v01); // The delta-u bump value
    LONG iDv = (v1M-v10); // The delta-v bump value
    if( (v00 < vM1) && (v00 < v01) ) // If we are at valley
    {
        iDu = vM1-v00; // Choose greater of 1st order
        if( iDu < v00-v01 )
            iDu = v00-v01;
    }
    // The luminance bump value (land masses are less shiny)
    WORD uL = ( v00>1 )? 63 : 127;
    switch( m_BumpMapFormat )
    {
        case D3DFMT_V8U8:
            *pDstT++ = (BYTE)iDu;
            *pDstT++ = (BYTE)iDv;
            break;
        case D3DFMT_L6V5U5:
            *(WORD*)pDstT = (WORD)( ( (iDu>>3) & 0x1f ) << 0 );
            *(WORD*)pDstT |= (WORD)( ( (iDv>>3) & 0x1f ) << 5 );
            *(WORD*)pDstT |= (WORD)( ( (uL>>2) & 0x3f ) << 10 );
            pDstT += 2;
            break;
        case D3DFMT_X8L8V8U8:
            *pDstT++ = (BYTE)iDu;
            *pDstT++ = (BYTE)iDv;
            *pDstT++ = (BYTE)uL;
            *pDstT++ = (BYTE)0L;
```

```

        break;
    }
    // Move one pixel to the left (src is 32-bpp)
    pSrcB0+=4;    pSrcB1+=4;    pSrcB2+=4;
}
// Move to the next line
pSrc += dwSrcPitch;    pDst += dwDstPitch;
}
m_psBumpMap->UnlockRect(0);
psBumpSrc->UnlockRect(0);
return S_OK;
}

```

To get access to the texture's surface, we have to use LockRect().

```

HRESULT LockRect(
    D3DLOCKED_RECT* pLockedRect,
    CONST RECT* pRect,
    DWORD Flags
);

```

To request that the entire surface be locked, we set pRect to 0. We use the surface description of the allocated bump map with

```
psBumpSrc->GetLevelDesc( 0, &d3dsd );
```

There's one pointer that points into the destination surface (pDst) and three pointers to the source surface (pSrcB0, pSrcB1, and pSrcB2).

The two for() statements read out the source surface with height × width pixel. In the inner loop, five pixels are read out of the source surface: the current, right, left, below, and above pixels.

```

LONG v00 = *(pSrcB0+0); // Get the current pixel
LONG v01 = *(pSrcB0+4); // and the pixel to the right
LONG vM1 = *(pSrcB0-4); // and the pixel to the left
LONG v10 = *(pSrcB1+0); // and the pixel one line below.
LONG v1M = *(pSrcB2+0); // and the pixel one line above.

```

The right and the left pixels and the below and above pixels are subtracted from each other by

```
LONG iDu = (vM1-v01); // The delta-u bump value
LONG iDv = (v1M-v10); // The delta-v bump value
```

These are the bump values that will be written into the new texture surface, which will be the new bump map.

To differentiate the valley from the rest of the land mass, the green values have to be lower than the land with other colors.

```
if( (v00 < vM1) && (v00 < v01) ) // If we are at valley
{
    iDu = vM1-v00;                  // Choose greater of 1st order diffs
    if( iDu < v00-v01 )
        iDu = v00-v01;
}
```

We will give the land masses a less shiny luminance by the following code:

```
// The luminance bump value (land masses are less shiny)
WORD uL = ( v00>1 )? 63 : 127;
```

Depending on the format that the user has chosen in the Options menu, the bump map is filled with the proper values in the following code:

```
switch( m_BumpMapFormat )
{
    case D3DFMT_V8U8:
        *pDstT++ = (BYTE)iDu;
        *pDstT++ = (BYTE)iDv;
        break;
    case D3DFMT_L6V5U5:
        *(WORD*)pDstT  = (WORD)( ( (iDu>>3) & 0x1f ) << 0 );
        *(WORD*)pDstT |= (WORD)( ( (iDv>>3) & 0x1f ) << 5 );
        *(WORD*)pDstT |= (WORD)( ( (uL>>2) & 0x3f ) << 10 );
        pDstT += 2;
        break;
    case D3DFMT_X8L8V8U8:
        *pDstT++ = (BYTE)iDu;
        *pDstT++ = (BYTE)iDv;
        *pDstT++ = (BYTE)uL;
        *pDstT++ = (BYTE)0L;
        break;
}
```

The whole InitBumpMap() method is called in MsgProc() every time the user chooses another bitmap format.

```
...
case IDM_U8V8L8:
    SAFE_RELEASE( m_psBumpMap );
    m_BumpMapFormat = D3DFMT_X8L8V8U8;
    InitBumpMap();
    break;
case IDM_U5V5L6:
    SAFE_RELEASE( m_psBumpMap );
    m_BumpMapFormat = D3DFMT_L6V5U5;
    InitBumpMap();
    break;
case IDM_U8V8:
    SAFE_RELEASE( m_psBumpMap );
    m_BumpMapFormat = D3DFMT_V8U8;
    InitBumpMap();
    break;
...
...
```

In the case of the D3DFMT\_X8L8V8U8 format, there are the two delta values, du and dv; one luminance value, 63 or 127; and 0L. You might save a scaling factor in the last value, which can be multiplied with the luminance value, as you will see in a few seconds.

## R E N D E R ()

As usual, Render() displays the final scene. There are a few interesting SetTextureStageState() calls here:

```
m_pd3dDevice->SetRenderState( D3DRS_WRAPO, D3DWRAP_U | D3DWRAP_V );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_DISABLE );
if( m_bTextureOn )
    m_pd3dDevice->SetTexture( 0, m_pEarthTexture );
else
    m_pd3dDevice->SetTexture( 0, m_pBlockTexture );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 1 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
```

```

if( m_bBumpMapOn && m_bEnvMapOn )
{
    m_pd3dDevice->SetTexture( 1, m_psBumpMap );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 1 );
    if( m_BumpMapFormat == D3DFMT_V8U8 )
        m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_BUMPENVMAP );
    else
        m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP,
D3DTOP_BUMPENVMAPLUMINANCE );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT00, F2DW(0.5f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT01, F2DW(0.0f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT10, F2DW(0.0f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT11, F2DW(0.5f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLSCALE, F2DW(4.0f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLOFFSET, F2DW(0.0f) );
    if( m_bEnvMapOn )
    {
        m_pd3dDevice->SetTexture( 2, m_pEnvMapTexture );
        m_pd3dDevice->SetTextureStageState( 2, D3DTSS_TEXCOORDINDEX, 0 );
        m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLOROP, D3DTOP_ADD );
        m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG1, D3DTA_TEXTURE );
        m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG2, D3DTA_CURRENT );
    }
    else
        m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLOROP, D3DTOP_DISABLE );
...
}

```

In the case of a non-D3DFMT\_V8U8 texture, the call to D3DTOP\_BUMPENVMAPLUMINANCE tells the system to include luminance values in bump mapping. So you must set the so-called luminance controls, which are D3DTSS\_BUMPENVLSCALE and D3DTSS\_BUMPENVLOFFSET. This second texture stage takes the result of the previous blending stage (D3DTA\_CURRENT), illuminates it with the bump map illumination values, scales it with 4.0 from D3DTSS\_BUMPENVLSCALE, and starts reading at the offset 0.0 from D3DTSS\_BUMPENVLOFFSET.

The two delta values, du and dv, are taken from the bump map and transformed by a 2-by-2 matrix. This matrix is built from the values provided to D3DTSS\_BUMPENVMAT00, D3DTSS\_BUMPENVMAT01, D3DTSS\_BUMPENVMAT10, and D3DTSS\_BUMPENVMAT11. The matrix is filled here with the DWORD values

0.5 0.0  
0.0 0.5

Once the new delta values  $du'$  and  $dv'$  are calculated, they are added to the texture coordinate in the next texture stage—here, the third texture stage—and Direct3D adds the selected color according to its luminance in order to compute the color to apply to the polygon.

#### CONFIRMDEVICE()

`ConfirmDevice()` has to check a lot of the rendering device's capabilities rendering device for bump mapping:

```
        return S_OK;
    }

}

// Else, reject the device
return E_FAIL;
}
```

As in the previous example, the D3DCREATE\_PUREDEVICE specifies that the driver chosen (by the Common files framework) does not support Get\* calls for anything that can be stored in state blocks. So the example will exit if this flag is set. The D3DTEXOPCAPS\_BUMPMAPLUMINANCE and D3DTEXOPCAPS\_BUMPMAP flags check whether this device supports one of the two bump-mapping blend operations. The three CheckDeviceFormat() calls check the support for the three different texture formats. My graphics card is restricted on one: D3DFMT\_V8U8.

## DOT PRODUCT TEXTURE BLENDING

Dot product texture blending was introduced in hardware with the GeForce 256 graphic cards by NVIDIA. It should become one of the standard features in games in the next two years, because it's supported by all upcoming graphic chips. So what does it do for you? It performs per-pixel lighting in real time. Moreover, it supports bump mapping as shown previously, so it's often called DotProduct3 or Dot3 bump mapping.

As I explained in the dark mapping section earlier, Direct3D lighting is calculated on a per-vertex basis. That means that a smooth-looking object needs a lot of vertices or, in other words, a high level of tessellation.

To get a better visual experience, textures are often used as lighting look-up tables, as shown earlier in the dark mapping and glow mapping examples. The lighting solution is precalculated in the light or dark map. Because the lighting calculation is precomputed, it is difficult to combine with dynamic lights calculated at run time.

Dot product texture blending gives you a lighting environment with a surface description at maximum detail (that is, per pixel), which contains enough information to recalculate the lighting at run time.

What's the idea behind dot product texture blending? This blend mode can be used to compute a per-pixel dot product, so it can compute per-pixel normalized vectors. (This is also the fundamental operation behind diffuse and specular lighting calculation.) Such a vector could be used to determine how closely the pixel is facing the light direction.

How is it done? The normalized vector for every pixel is stored in a texture as a 32-bit value, built at the start time of your program. You might generate such a texture in `InitDeviceObjects()`.

The DotProduct3 example in the DirectX 8.0 SDK shows this technique in a very instructive and clear manner.



**Figure 8.13:** DotProduct3 example (example by Microsoft)

## INITDEVICEOBJECTS()

The normal maps that hold per-pixel vectors are produced in `InitDeviceObjects()`.

```
...
// Create the normal maps
if( FAILED( hr = CreateFileBasedNormalMap() ) )
    return hr;
if( FAILED( hr = CreateCustomNormalMap() ) )
    return hr;
...
```

The call to `CreateFileBasedNormalMap()` loads the `earthbump.bmp` texture map and builds the normal map from it.

```
HRESULT CMyD3DApplication::CreateFileBasedNormalMap()
{
    HRESULT hr;
    // Load the texture from a file
    if( FAILED( hr = D3DUtil_CreateTexture( m_pd3dDevice, _T("EarthBump.bmp"),
                                            &m_pFileBasedNormalMap,
                                            D3DFMT_A8R8G8B8 ) ) )
        return D3DAPPERR_MEDIANOTFOUND;
    // Lock the texture
```

```

D3DLOCKED_RECT d3dlr;
D3DSURFACE_DESC d3dsd;
m_pFileBasedNormalMap->GetLevelDesc( 0, &d3dsd );
m_pFileBasedNormalMap->LockRect( 0, &d3dlr, 0, 0 );
DWORD* pPixel = (DWORD*)d3dlr.pBits;
// For each pixel, generate a vector normal that represents the change
// in the height field at that pixel
for( DWORD j=0; j<d3dsd.Height; j++ )
{
    for( DWORD i=0; i<d3dsd.Width; i++ )
    {
        DWORD color00 = pPixel[0];
        DWORD color10 = pPixel[1];
        DWORD color01 = pPixel[d3dlr.Pitch/sizeof(DWORD)];
        FLOAT fHeight00 = (FLOAT)((color00&0x00ff0000)>>16)/255.0f;
        FLOAT fHeight10 = (FLOAT)((color10&0x00ff0000)>>16)/255.0f;
        FLOAT fHeight01 = (FLOAT)((color01&0x00ff0000)>>16)/255.0f;
        D3DXVECTOR3 vPoint00( i+0.0f, j+0.0f, fHeight00 );
        D3DXVECTOR3 vPoint10( i+1.0f, j+0.0f, fHeight10 );
        D3DXVECTOR3 vPoint01( i+0.0f, j+1.0f, fHeight01 );
        D3DXVECTOR3 v10 = vPoint10 - vPoint00;
        D3DXVECTOR3 v01 = vPoint01 - vPoint00;
        D3DXVECTOR3 vNormal;
        D3DXVec3Cross( &vNormal, &v10, &v01 );
        D3DXVec3Normalize( &vNormal, &vNormal );
        // Store the normal as an RGBA value in the normal map
        *pPixel++ = VectortoRGBA( &vNormal, fHeight00 );
    }
}
// Unlock the texture and return successful
m_pFileBasedNormalMap->UnlockRect(0);
return S_OK;
}

```

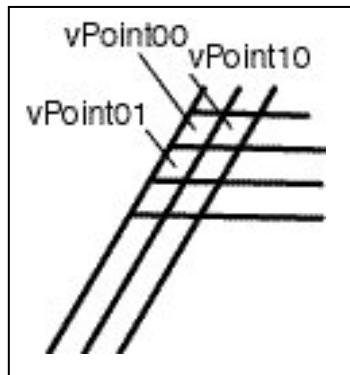
The normal map stores the per-pixel vector and a height value in a pixel. The code reads three height values from three different points from the bitmap file:

Subtracting vPoint00 from vPoint10 or vPoint01 from vPoint00 leads to the case in which the height value of vPoint00 is higher than the height values of the other points, to negative height for v10 and v01.

```

D3DXVECTOR3 v10 = vPoint10 - vPoint00;
D3DXVECTOR3 v01 = vPoint01 - vPoint00;

```

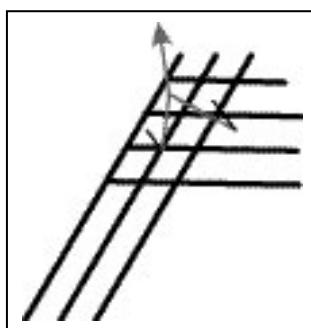


**Figure 8.14:** Three height values: *vPoint00*, *vPoint10*, *vPoint01*

Vector *v10* will have the following values:  $(1.0, 0.0, fHeight10 - fHeight00)$ , and *v01* will have the following values:  $(0.0, 1.0, fHeight01 - fHeight00)$ . The origin of the two vectors *v10* and *v01* lies in  $(0.0, 0.0, 0.0)$ . This is the same place where *vPoint00* resides. *v10*, *v01*, and *vPoint00* form a triangle.

```
D3DXVECTOR3 vNormal;
D3DXVec3Cross( &vNormal, &v10, &v01 );
```

Taking the cross product of any two vectors forms a third vector perpendicular to the plane formed by the first two. The cross product is used to determine which way polygons are facing. It can be used to generate a normal to any surface for which you have two vectors that lie within the surface. The green arrow symbolizes the normal vector and the two red vectors show the *v10* and *v01* vectors.



**Figure 8.15:** The upper arrow is the normal vector, and the two lower vectors are *v10* and *v01*.

The cross product is determined using the following code:

```
D3DXVECTOR3 v;
v.x = pV1->y * pV2->z - pV1->z * pV2->y;
```

```

v.y = pV1->z * pV2->x - pV1->x * pV2->z;
v.z = pV1->x * pV2->y - pV1->y * pV2->x;
*pOut = v;
=
v.x = 0.0 - (fHeight10 - fHeight00) * 1.0;
v.y = 0.0 - 1.0 * (fHeight01 - fHeight00);
v.z = 1.0 * 1.0 - 0.0 * 0.0;

```

Let's use a few virtual numbers: `fHeight00` should be 0.8, `fHeight10` should be 0.6, and `fHeight01` should be 0.4.

```

v.x = 0.0 - (0.6 - 0.8) * 1.0;
v.y = 0.0 - 1.0 * (0.4 - 0.8);
v.z = 1.0 * 1.0 - 0.0 * 0.0;

```

So, `v` is (0.2, 0.4, 1.0).

Getting the height value works like this:

```
FLOAT fHeight00 = (FLOAT)((color00&0x00ff0000)>>16)/255.0f;
```

Using `0x00ff0000` with the `&` operator leads to masking out the first, third, and fourth byte in the `color00` value. The height value is located in the second byte. Dividing it by 255 leads to a floating point value that will lie between 0 and 1, because it has stored values ranging from 0 to 255.

In `CreateFileBasedNormalMap()` and `CreateCustomNormalMap()`, the normals are stored in a signed 8-bit integer and packed into a RGBA color structure (32-bit) with the following call.

```

DWORD VectortoRGBA( D3DXVECTOR3* v, FLOAT fHeight )
{
    DWORD r = (DWORD)( 127.0f * v->x + 128.0f );
    DWORD g = (DWORD)( 127.0f * v->y + 128.0f );
    DWORD b = (DWORD)( 127.0f * v->z + 128.0f );
    DWORD a = (DWORD)( 255.0f * fHeight );
    return( (a<<24L) + (r<<16L) + (g<<8L) + (b<<0L) );
}

```

The normal map is called in `Render()`.

## RENDER()

As usual in this part of the book, the main action takes place with the `SetTextureStageStates()` call, which is mostly located in the `Render()` method. You will be surprised what a few lines of code can produce.

A so-called light vector produced in FrameMove() holds the position of the mouse. This vector is used as the D3DRS\_TEXTUREFACTOR value. Its origin lies in the middle of the example window.

```
HRESULT CMyD3DApplication::Render()
{
    // Clear the render target
    m_pd3dDevice->Clear( 0L, NULL, D3DCLEAR_TARGET, 0x00000000f, 1.0f, 0L );
    // Begin the scene
    if( SUCCEEDED( m_pd3dDevice->BeginScene() ) )
    {
        // Store the light vector, so it can be referenced in D3DTA_TFACTOR
        DWORD dwFactor = VectortoRGBA( &m_vLight, 0.0f );
        m_pd3dDevice->SetRenderState( D3DRS_TEXTUREFACTOR, dwFactor );
        // Modulate the texture (the normal map) with the light vector (stored
        // above in the texture factor)
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,    D3DTOP_DOTPRODUCT3
);
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_TFACTOR );
        // If user wants to see the normal map, override the above renderstates and
        // simply show the texture
        if( TRUE == m_bShowNormalMap )
            m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1
);
        // Select which normal map to use
        if( m_bUseFileBasedTexture )
            m_pd3dDevice->SetTexture( 0, m_pFileBasedNormalMap );
        else
            m_pd3dDevice->SetTexture( 0, m_pCustomNormalMap );
        // Draw the bumpmapped quad
        m_pd3dDevice->SetVertexShader( CUSTOMVERTEX_FVF );
        m_pd3dDevice->DrawPrimitiveUP( D3DPT_TRIANGLESTRIP, 2, m_QuadVertices,
                                         sizeof(CUSTOMVERTEX) );
        // Output statistics
        m_pFont->DrawText( 2, 0, D3DCOLOR_ARGB(255,255,255,0), m_strFrameStats );
        m_pFont->DrawText( 2, 20, D3DCOLOR_ARGB(255,255,255,0), m_strDeviceStats );
        // End the scene.
        m_pd3dDevice->EndScene();
    }
    return S_OK;
}
```

D3DTOP\_DOTPRODUCT3 modulates the light vector as TextureFactor and the normal map as TextureColor.

```
FinalColor = TextureFactor * TextureColor
```

TextureColor consists of four RGBA values, each filled with a vector normal and a height value. You might see the color of the texture map by switching m\_bShowNormalMap on.

As you can see, dot product blending mode is named after the mathematical operation that combines a light vector with a surface normal.

## MULTITEXTURING SUPPORT

If a program likes to use more than one texture at once, you have to check the multitexturing support of your 3-D hardware in ConfirmDevice():

```
if( pCaps->MaxSimultaneousTextures < 2 )
    return E_FAIL;
```

This code checks the maximum number of supported simultaneous texture-blending stages. It must be higher than one. If you would like to be sure that your multitexturing game is supported on specific hardware, you should check the different texturing possibilities of the user's hardware. The sample program provided for this part of the book checks the following graphic card capabilities:

```
if( (0 == ( pCaps->DestBlendCaps & D3DPBLENDCAPS_INVSRCOLOR)) && // alpha blending
    (0 == ( pCaps->SrcBlendCaps & D3DPBLENDCAPS_SRCCOLOR)) &&
    (0 == ( pCaps->TextureOpCaps & D3DTEXOPCAPS_ADDSIGNED )) && // texture blending
    (0 == ( pCaps->TextureOpCaps & D3DTEXOPCAPS_MODULATE )) &&
    (0 == ( pCaps->TextureOpCaps & D3DTEXOPCAPS_MODULATE2X)) &&
    (0 == ( pCaps->TextureOpCaps & D3DTEXOPCAPS_MODULATE4X)) &&
    (0 == ( pCaps->TextureOpCaps & D3DTEXOPCAPS_ADD )) &&
    (0 == ( pCaps->TextureAddressCaps & D3DPTADDRESSCAPS_CLAMP)) && // texture addressing
    (0 == ( pCaps->TextureAddressCaps & D3DPTADDRESSCAPS_BORDER )) &&
    (0 == ( pCaps->TextureAddressCaps & D3DPTADDRESSCAPS_MIRRORONCE)))
return E_FAIL;
```

For the alpha-blending code, these are the two alpha-blending parameters I use. I would like to use the ADD, ADDSIGNED, MODULATE, MODULATE2X, and MODULATE4X color operations. The three texture-addressing modes also have to be supported by the graphics hardware: clamp, border (not supported by GeForce), and mirroronce (totally new in DirectX 8.0).

## TEXTURE MANAGEMENT

If you only have 64MB of video texture memory in your graphics hardware, you should think about which texture should be loaded or held in memory—or better, an algorithm should let your computer think about that. So what's the task? You need to track the amount of available texture memory and to get an overview of which textures are needed more often or less often. At the least, a texture management algorithm has to decide which existing texture resources can be reloaded with another texture image, and which surfaces should be destroyed and replaced with new texture resources.

Direct3D has an automatic texture-management system. As you learned earlier, you request its support when you create a texture with `CreateTexture()` and specify the `D3DPPOOL_MANAGED` flag for the `Pool` parameter. You cannot use the `D3DPPOOL_DEFAULT` or `D3DPPOOL_SYSTEMMEM` flags when creating a managed texture.

The texture manager tracks textures with a time stamp that identifies when the texture was last used. It uses the least recently used algorithm to determine which textures should be removed. Texture priorities are used as tiebreakers when two textures are targeted for removal from memory. If they have the same priority, the least recently used texture is removed. If they have the same time stamp, the texture that has a lower priority is removed first.

You can assign a priority to managed textures by calling the `IDirect3DResource8::SetPriority()` method for the texture surface.

Keep with `D3DPPOOL_MANAGED` until you know more.

## QUIZ

Q: Why should we use textures?

A: To simulate a rough or irregular surface or to create the illusion that the object consists of more polygons.

Q: How does the addressing scheme of Direct3D work?

A: It uses a normalized addressing scheme in which texture addresses consist of texel coordinates that map to the range of 0.0 to 1.0.

Q: May I use more than one texture coordinate pair per texture?

A: Yes, you can use up to eight texture coordinate pairs for one texture by declaring them in your vertex definition and defining them in your vertex structure.

Q: What do the so-called texture-addressing modes do?

A: You can generate special effects by using texture addresses outside of the 0.0 to 1.0 range and by using another texture-addressing mode.

Q: Which is the name of default texture-addressing mode and of the other modes?

A: Wrap, mirror, clamp, border, and mirroronce.

Q: What is the purpose of the mirror texture address mode?

A: It mirrors the texture at every integer boundary, so that the texels are flipped outside of the 0.0 to 1.0 range.

Q: What is the purpose of the clamp texture address mode? Why is that useful?

A: It smears the color of the edge pixels to the borders of the window. Bilinear interpolation at the edges sometimes produces artifacts with the wrap mode.

Q: What is the purpose of the border color address mode? What is its drawback?

A: It sets a border around the texture that will have the color you've chosen, so you could fake larger textures on walls, for example. It's not supported on GeForce cards.

Q: What is the purpose of the newly introduced mirroronce texture address mode?

A: It mirrors within the range of -1.0 to +1.0, outside of this range it is clamped. It's useful for volumetric light maps.

Q: What is texture wrapping and why is it useful?

A: It gives the programmer the possibility to decide how to interpolate between the texture coordinates. To use a simple example: It's not always useful to interpolate a texture that is wrapped around a cylinder in the shortest way between the texture. It is often more useful to cross the boundaries of the texture.

Q: What prevents texture wrapping?

A: Using texture addressing modes, because it makes texture coordinates outside of the 0.0 to 1.0 range invalid.

Q: What are texture filtering and texture anti-aliasing?

A: These are methods to decrease the drawbacks of using raster graphics. They prevent staircase, magnification, and minification problems.

Q: What are mipmaps?

A: These are a series of textures in which every mipmap sequence has a height and width that is half of the height and width of the previous level. Direct3D will automatically pick the appropriate mip level to texture from depending on the width and height of the textured polygon it has to draw.

Q: What is linear texture filtering?

A: It averages the four nearest texels together based on the relative distance from the sampling point.

Q: What is trilinear filtering?

A: It's bilinear filtering plus mipmaps.

Q: What is anisotropic filtering? What is its purpose?

A: It averages not only the four nearest texels, but a lot more, depending on the support of your graphics card and the quality you choose. It also uses mipmaps. It gives you more depth of detail and a more accurate representation of the texture.

Q: How does full-scene anti-aliasing work?

A: It samples each pixel multiple times. The various samples are blended together and output to the screen. It enables a few interesting special effects, like motion blur, depth of field, reflection blur, and so on.

Q: What is the formula for alpha blending?

A:  $\text{FinalColor} = \text{SourcePixelColor} * \text{SourceBlendFactor} + \text{DestPixelColor} * \text{DestBlendFactor}$

Q: How does alpha blending work?

A: It blends the pixel currently in the pixel pipeline with the pixel already in the frame buffer, depending on the blending factors used.

Q: What are multipass texturing and multitexturing?

A: Multipass texturing uses more than one pass to map textures on a polygon. Multitexturing renders more than one texture on a single pass on a polygon. The latter is faster.

Q: What is a texture blending cascade?

A: It's the result of one texture stage used in the next texture stage. It's possible to blend together the results of up to eight texture stages.

Q: Why use dark/light mapping?

A: Direct3D lights are generated on a per-vertex basis. Think of a large triangle with a light source close to the surface. As long as the light is close to one of the vertices of the triangle, you can see the lighting effect. If the light is in the middle of the triangle, there's very little shining on it. Dark/light maps help to generate the illusion of pixel-based lighting.

Q: How does dark/light mapping work?

A: The colors of two textures are multiplied together.

Q: In which ways may I get a diffuse color?

A: Pseudocode:

```
switch (D3DRS_DIFFUSEMATERIALSOURCE)
case D3DMCS_MATERIAL
    use the diffuse color of the material
break;
case D3DMCS_COLOR1
    use the diffuse color of the vertex. If you haven't chosen one, opaque white is
taken.
```

```
break;  
case D3DMCS_COLOR2  
    use the specular color of the vertex. If you haven't chosen one, opaque white is  
    taken.  
break;  
default:  
    use the diffuse color of the material.
```

Q: What is the short description of blending a texture with material diffuse color?

A: BaseMap + DiffuseInterpolation

Q: What is the difference between the glow mapping and the dark mapping effects?

A: In glow mapping, we add the two textures, whereas with dark mapping, we multiply them.

Q: What is the sufficient replacement for D3DTOP\_ADDSIGNED?

A: D3DTOP\_MODULATE2X.

Q: Which color/alpha operation parameters need multitexture support for three texture stages?

A: D3DTOP\_MULTIPLYADD: Result = Arg1 + Arg2 \* Arg3

D3DTOP\_LERP: Result = (Arg1) \* Arg2 + (1-Arg1) \* Arg3

Q: Where do you get the so-called alpha value for texture-blending operations from?

A: From vertices, material, and texture maps. Pseudocode:

```
switch(D3DRS_DIFFUSEMATERIALSOURCE)  
case D3DMCS_DIFFUSE:  
    use the alpha from the material color + ALPHAARG2 = D3DTA_DIFFUSE  
break;  
case D3DMCS_COLOR1: (default)  
    use the alpha from the color component within a vertex + ALPHAARG2 = D3DTA_DIFFUSE  
break;  
default:  
    use the alpha from material color
```

If you would like to use the alpha from the texture, use D3DTA\_TEXTURE as the ALPHAARG1.

Q: What is the purpose of environment mapping? Which two kinds of environment mapping do you know?

A: Environment mapping reflects the environment of a polygon. We've discussed spherical environment mapping and cubic environment mapping.

Q: What is the idea behind spherical environment mapping? What are its drawbacks?

A: The idea is to project a static bitmap onto an object that shows a similar image as the environment. The bitmap is not built in real time, so it won't cover changes in the environment, and there are warping effects at the edges of the map.

Q: How do you set matrices for texture coordinate transformation?

A: You use SetTransform() the way you set them for geometry transformation, but you have to provide D3DTS\_TEXTURE0 through D3DTS\_TEXTURE7.

Q: What is the most important render state operation in the spherical environment mapping example?

A: D3DTSS\_TCI\_CAMERASPACENORMAL.

Q: How does cubic environment bump mapping work?

A: Six textures from the six different directions are mapped on the object. The textures are produced by rendering six different views from the point where the object resides into the six textures.

Q: What is a stencil buffer?

A: A stencil buffer is an extra per-pixel test buffer.

Q: What's the purpose of SetRenderTarget()?

A: It sets a new color buffer and/or depth/stencil buffer.

Q: How does the production of a cube map work?

A:

1. Position a camera in the middle of the object.
2. Rotate it into the six directions necessary to build a cube.
3. Render the camera view into the six cube map surfaces.

Q: Which is the most important texture stage parameter in the cubic environment example?

A: D3DTSS\_TCI\_CAMERASPACEREFLECTIONVECTOR.

Q: Which caps variable do you have to check to ensure that your graphics hardware has the capability to support cube maps?

A: D3DPTEXTURECAPS\_CUBEMAP.

Q: How does bump mapping work?

A: A bump map with contour data in the form of delta u and v values is used to fake the appearance of a bumpy surface by setting a different set of normals for lighting.

Q: How is the bump map produced in the bump map example used here?

A: An ARGB value holds the delta u and v values and a luminance value to differentiate between the valleys and the mountains.

Q: What does the luminance value do in bump mapping? How is it used?

A: It illuminates the pixels in the next texture stage, here, the environment map texture stage. With D3DTOP\_BUMPENVMAPLUMINANCE as Color operation and D3DTSS\_BUMPENVLSCALE and D3DTSS\_BUMPENVLOFFSET as arguments.

Q: How is the 2-by-2 matrix built to transform the du and dv values?

A: It's built with D3DTSS\_BUMPENVMAT00, D3DTSS\_BUMPENVMAT01, D3DTSS\_BUMPENVMAT10, and D3DTSS\_BUMPENVMAT11. The matrix looks like

```
00 01  
10 11
```

Q: What are the benefits of using dot product blending?

A: You will do per-pixel lighting bump mapping, and most of the horsepower needed to do this is contributed by the graphics processor.

Q: What do we need to get dot product blending?

A: We need the per-pixel vector and a height value, stored in the pixels of the normal map. This normal map is multiplied as TextureColor with a Texture factor that holds the light vector.

```
FinalColor = TextureFactor * TextureColor
```

With

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );  
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_DOTPRODUCT3 );  
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_TFACTOR );
```

## ADDITIONAL RESOURCES

You can find one of the best articles on multitexturing with DirectX from Mitchell, Tatro, and Bullard in Gamasutra at [www.gamasutra.com/features/programming/19981009/multitexturing\\_01.htm](http://www.gamasutra.com/features/programming/19981009/multitexturing_01.htm). They have developed a tool to visualize the texture operations . . . try it. You might find another tool to visualize these operations from NVIDIA at [www.nvidia.com](http://www.nvidia.com). An interesting article is "Multipass Rendering and the Magic of Alpha Rendering" by Brian Hook in *Game Developer* magazine, August 1997, page 12. I've also found the examples from ATI at [www.ati.com](http://www.ati.com) very interesting. They use partly the Direct3D 7.0 and framework and partly the DirectX 8.0 Common files to show a few nice effects.

## ANISOTROPY

[home.swipnet.se/~w-12597/3dxtc/articles/anisotropic.htm](http://home.swipnet.se/~w-12597/3dxtc/articles/anisotropic.htm)

[www.ati.com/na/pages/resource\\_centre/dev\\_rel/sdk/radeon sdk/Html/Samples/OpenGL/Radeon AnisoFilter.html](http://www.ati.com/na/pages/resource_centre/dev_rel/sdk/radeon sdk/Html/Samples/OpenGL/Radeon AnisoFilter.html)

[www.ping.be/powervr/Anisotropic.htm](http://www.ping.be/powervr/Anisotropic.htm)

## DETAIL MAPPING

Adrian Perez, *Advanced 3-D Game Programming Using DirectX 7.0*, 530–536, Wordware Publishing, Inc., 2000.

## CUBIC ENVIRONMENT MAPPING

Adrian Perez, *Advanced 3-D Game Programming Using DirectX 7.0*, 525–529, Wordware Publishing, Inc., 2000.

## STENCIL BUFFERS

Mark Kilgard, “Improving Shadows and Reflections via the Stencil Buffer,” white paper from NVIDIA,  
[www.nvidia.com/Developer.nsf](http://www.nvidia.com/Developer.nsf).

## BUMP MAPPING

Peter J. Kovach, *Inside Direct3D*, 252–257, Microsoft Press, 2000.

## DOT PRODUCT TEXTURE BLENDING

Sim Dietrich, “Dot Product Texture Blending and Per-Pixel Lighting,” white paper from NVIDIA,  
[www.nvidia.com/Developer.nsf](http://www.nvidia.com/Developer.nsf).

The people at [www.reactorcritical.com](http://www.reactorcritical.com) have built a Java glossary, which explains a few of the techniques shown here with a Java example. It is very nice and instructive.

*This page intentionally left blank*

# **PART III**

# **HARD-CORE DIRECTX GRAPHICS PROGRAMMING**

You have learned Direct3D up until now by understanding the basics. Now, you have decided to write your 3-D world-class game. So, what is the first thing you have to think about?

Your game data storage system. How should you store your geometry and color data so that you get fast data access with minimum storage requirements?

I will give you two examples of successful file formats for games: the so-called X file format provided with the DirectX SDK, which is a very flexible game file format, and the .md3 format, favored for storing Quake III models' data. The latter one is used in a lot of 3-D shooters that are built on the licensed Quake III engine.

After you have your models and landscapes/dungeons up and running, you would probably like to know how to move your models in a realistic way, and how to keep track and handle collisions of models with other models, the landscape, or the dungeon.

So we will deal with the following advanced topics in this part of the book. You will learn

- To work with the standard DirectX X file format
- To program a Quake III model viewer
- How to port a program that is based on the right-handed coordinate system used by OpenGL to the left-handed coordinate system used by Direct3D
- How all those groovy-looking game physics work
- How to detect a collision, react on it, or prevent it

I will give you the basic concept of how to use game file formats so that you will be able to build your own format. I'm pretty sure that your unique game idea will need one of its own.

# **CHAPTER 9**

# **WORKING WITH FILES**

**T**here are a lot of file formats out there. We might distinguish between file formats typically used by graphic artists and file formats typically used in game engines. File formats that are used by 3-D tools are .3ds, .max, .cof, and so on, which are produced by those tools. These formats have a lot of info in them and are not optimized for use in real-time environments like game engines. So games use faster and simpler file formats, which differ between game genres and are generally unique to one game.

One of the most challenging economical considerations in game production is to build a proper workflow for the graphic information that has to be exchanged between the 3-D graphic artists and the game engine programmer. The typical solution to this problem is to write custom tools, which import, for example, the 3D Studio Max file and exports it, after more or less modification, in the game engine format.

The game engine formats used in id Software's Quake series are widely used, because a lot of game companies licensed the Quake engines.

So I'll concentrate on the following file formats, which I think are easy to learn: the X file format used in DirectX, and the Quake III model format. If you know these two formats, you should be able to understand and use other formats as well.

## BUILDING WORLDS WITH X FILES

To make a game, programmers and graphic designers need to work closely together. 3-D characters, models, and 3-D worlds are built and animated in specialized 3-D programs, such as 3D Studio Max, Maya, LightWave, trueSpace, and so on. After the graphics are built with such tools, they have to be imported into the game engine.

So if you'd like to be a game programmer, you need to understand how to load a 3-D model or world file generated by external tools into your game engine.

As a sample application, I've modified the example from Chapter 6, so you know most of the source and can concentrate on the new X file code.

To be honest, the example X files are a result of a research order we received. We have to simulate the new Super Star Destroyer for a not-to-be-named client from the capital planet of our galaxy. This new Star Destroyer should be unboardable by rebel pilots and rebel queens, and there should be no way to destroy it by destroying its powerhouse.

If you're having trouble compiling this example program, you might read the section on compiling in the Essentials.

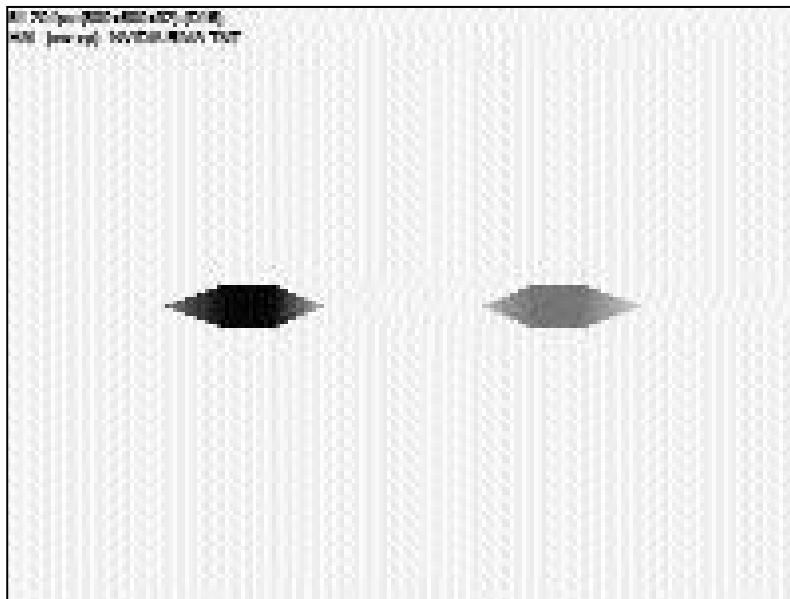


Figure 9.1: Example program

## 3-D FILE FORMATS

3-D scene file formats present two fundamental problems: how to store the objects that will make up the scene, and how to store the relationship between these objects.

User-defined data types are necessary. Scene data requires hierarchical relationships, such as parent, child, and sibling relationships, and associational relationships, such as meshes attached to frames, materials attached to meshes, and so on. In addition, a file format design should be robust enough to handle forward and backward compatibility.

The X file format handles all of these tasks, so it's a good solution to show the loading of 3-D models into a game engine in this book.

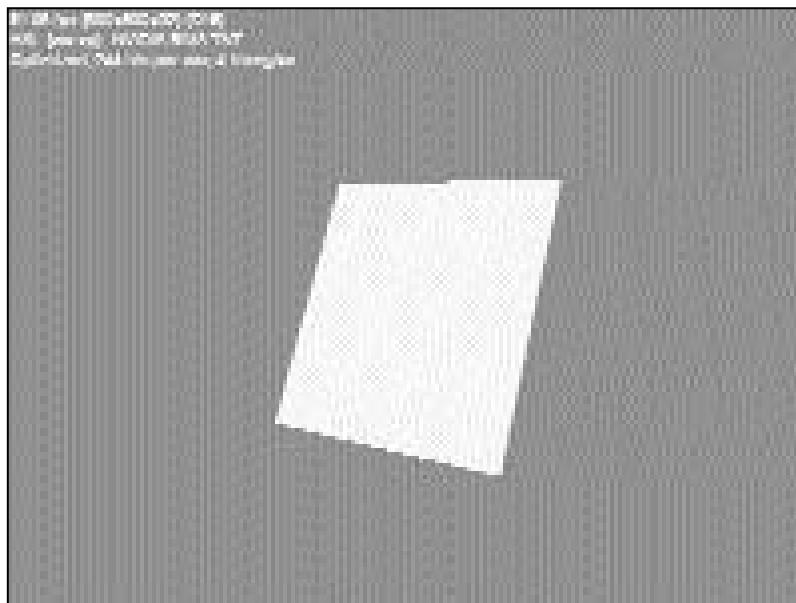
## X FILE FORMAT

The Direct3D X file format was built for the legacy retained mode of Direct3D and expanded with the advent of DirectX 6.0 for Immediate Mode. Let's see how the X file format solves the problems described above on an abstract level.

- **User-defined data types.** X files are driven by templates, which can be defined by the user. A *template* is the definition of how a data object should be stored. The predefined templates are contained in the directory headers rmxfmpl.h in d3dfile.cpp and rmxfmpl.x in x:\mssdk\include (try to pronounce that :-)), and their identification signatures are in rmxfguid.h, which is also included in d3dfile.cpp, one of the Common files.

- **Hierarchical relationships.** Data types allowed by the template are called *optional members*. These optional members are saved as children of the data object. The children can be another data object, a reference to an earlier data object, or a binary object.

OK, let's dive into the X file format with a sample: a square. It's located in square.x. To try out and play with the following files, I suggest using one of the example programs in the DirectX 8.0 SDK that deal with the X file format. I have used the Optimized Mesh example to check these files.



**Figure 9.2:** Square.x

And here's the source:

```
xof 0302txt 0064
```

```
// square
Mesh Square {
// front face and back face
8;                                // number of vertices
1.0; 1.0; 0.0;;                     // vertice 0
-1.0; 1.0; 0.0;;                   // vertice 1
-1.0;-1.0; 0.0;;                  // vertice 2
1.0;-1.0; 0.0;                     // vertice 3
1.0; 1.0; 0.0;;                   // vertice 4
1.0;-1.0; 0.0;;                  // vertice 5
```

```

-1.0;-1.0; 0.0;;           // vertex 6
-1.0; 1.0; 0.0;;           // vertex 7
4;                          // number of triangles
3;0,1,2;;                  // triangle #1
3;0,2,3;;                  // triangle #2
3;4,5,6;;                  // triangle #3
3;4,6,7;;                  // triangle #4

MeshMaterialList {
    1;                      // one material
    4;                      // one face
    0,                      // face #0 use material #0
    0,                      // face #1 use material #0
    0,                      // face #2 use material #0
    0;;                     // face #3 use material #0
Material {
    0.0;1.0;0.0;1.0;;       // material #0
    0.0;                   // face color
    0.0;                   // power
    0.0;0.0;0.0;;           // specular color
    0.0;0.0;0.0;;           // emissive color
}
}
}
}

```

## HEADER

The magic 4-byte number is xof. The major version number is 03, and the minor version number is 02. The file format is txt for text or bin for a binary format. A newer version number could be 0303.

Other possible file formats are tzip and bzip, which stand for MSZip compressed text or binary files, respectively. The floating point data type is set to the default 0064 instead of 0032.

### TIP

You can convert between the txt and bin formats with the help of convx.exe, which is located in d:\mssdk\DXUtils\Xfiles.

### NOTE

The DirectX 8.0 SDK provides converters for 3D Studio Max and Maya 2.5 and 3.0 to produce X files.

### NOTE

With Direct3D Retained Mode, a template was used called Header. Its declaration is located in the rmxftmpl.x file, like all other templates.

Comments are set, as you can see above, with either the double slash as in C++ or the hash symbol (#).

## MESH

Most of the sample file consists of a Mesh template and its integrated templates:

```
template Mesh {
<3D82AB44-62DA-11cf-AB39-0020AF71E433>
DWORD nVertices;
array Vector vertices[nVertices];
DWORD nFaces;
array MeshFace faces[nFaces];
[...]
}
```

The Mesh template needs the Vector and MeshFace templates:

```
template Vector {
<3D82AB5E-62DA-11cf-AB39-0020AF71E433>
FLOAT x;
FLOAT y;
FLOAT z;
}

template MeshFace {
<3D82AB5F-62DA-11cf-AB39-0020AF71E433>
DWORD nFaceVertexIndices;
array DWORD faceVertexIndices[nFaceVertexIndices];
}
```

Templates consist of four parts: The unique name, which consists of numbers, characters, and the underscore, is the first part. It shouldn't start with a number. The second part is the UUID (*universally unique identifier*), and the third part consists of the data types of the entries. The last part regulates the degree of restriction; a template can be open, closed, or restricted. Open templates can contain every other data type, closed templates contain no other data types, and restricted templates can only integrate specific other data types.

```
// square
Mesh Square {
// front face and back face
```

### NOTE

**With Direct3D Retained Mode, a template was used called Header. Its declaration is located in the rmxftmpl.x file, like all other templates:**

```
template Header {
<3D82AB43-62DA-11cf-AB39-0020AF71E433>
WORD major;
WORD minor;
DWORD flags;
}
```

**This template defines the application-specific header for the Direct3D Retained Mode usage of the DirectX file format. Retained Mode uses the major and minor flags to specify the current major and minor versions.**

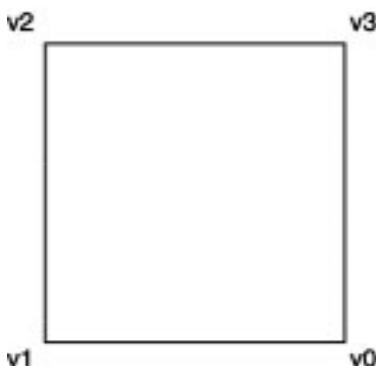
**We're not using Retained Mode here, so we can skip that by commenting it out or deleting it. You might use the Header template for your graphics version control system.**

```
8;                      // number of vertices
1.0; 1.0; 0.0;;          // vertice 0
-1.0; 1.0; 0.0;;          // vertice 1
-1.0;-1.0; 0.0;;          // vertice 2
1.0;-1.0; 0.0;           // vertice 3
1.0; 1.0; 0.0;;          // vertice 4
1.0;-1.0; 0.0;;          // vertice 5
-1.0;-1.0; 0.0;;          // vertice 6
-1.0; 1.0; 0.0;;          // vertice 7
4;                      // number of triangles
3;0,1,2;;                // triangle #1
3;0,2,3;;                // triangle #2
3;4,5,6;;                // triangle #3
3;4,6,7;;                // triangle #4
```

Here, the first number is the number of vertices used. After that, the vertices are set. The front face uses four vertices consisting of two triangles. There's a front and a back face.

### NOTE

In every game, only the front side of a face should be visible. A front face is one in which vertices are defined in clockwise order. Any face that is not a front face is a back face. Direct3D does not always render back faces, therefore, back faces are said to be culled, known as back-face culling.



**Figure 9.3:** Location of vertices on the front face of square.x

**NOTE**

In Direct3D 7.0 we could use quads instead of triangle information in MeshFace:

```
// Instead of ...
4;                                // number of triangles
3;0,1,2;;                         // triangle #1
3;0,2,3;;                         // triangle #2
3;4,5,6;;                         // triangle #3
3;4,6,7;;                         // triangle #4
// we could use this
2;                                // number of quads
4;0,1,2,3;;                         // quad #1
4;4,5,6,7;;                         // quad #2
```

But with Direct3D 8.0, I couldn't provide valid MeshNormals for a quad.

The Mesh template is an open template, because it uses open brackets [...] at the end. So it can contain every other data type.

**MESH MATERIAL LIST**

The MeshMaterialList is a child object of the Mesh object and is incorporated in the Mesh object. It needs the number of faces and materials and concatenates one face with one or more materials.

```
MeshMaterialList {
    1;                                // one material
    2;                                // two faces
    0,                                // face #0 use material #0
    0;;                               // face #1 use material #0
    Material {                         // material #0
        0.0;1.0;0.0;1.0;;              // face color
        0.0;                            // power
        0.0;0.0;0.0;;                  // specular color
        0.0;0.0;0.0;;                  // emissive color
    }
}
```

The template in rmxftmpl.x looks like:

```
template MeshMaterialList {  
    <f6f23f42-7686-11cf-8f52-0040333594a3>  
    DWORD nMaterials;  
    DWORD nFaceIndexes;  
    array DWORD faceIndexes[nFaceIndexes];  
    [Material]  
}
```

The first variable holds the number of materials used by this sample. The second variable holds the number of faces. The square uses the front and back face, so the variable has to be set to 2. The concatenation of the materials with the faces happens with the faceIndexes array. Every face is concatenated with a material by naming the material number. The X file reader knows the proper face by counting the number of faces provided.

Beneath that array, Material templates could be integrated. So the MeshMaterialList template is a restricted template, because only specified templates could be integrated.

```
template Material {  
    <3D82AB4D-62DA-11cf-AB39-0020AF71E433>  
    ColorRGBA faceColor;  
    FLOAT power;  
    ColorRGB specularColor;  
    ColorRGB emissiveColor;  
    [...]  
}
```

This template incorporates the following two templates:

```
template ColorRGBA {  
    <35FF44E0-6C7C-11cf-8F52-0040333594A3>  
    FLOAT red;  
    FLOAT green;  
    FLOAT blue;  
    FLOAT alpha;  
}  
template ColorRGB {  
    FLOAT red;  
    FLOAT green;  
    FLOAT blue;  
}
```

The Material data object holds the material color, power, specular color, and emissive color. The open brackets show that it can integrate other data objects. The Material data object could be referenced by MeshMaterialList. The file square2.x shows two referenced Material data objects.

```
xof 0302txt 0064
Material GreenMat {                                // material #0
    0.0;1.0;0.0;1.0;;
    0.0;
    0.0;0.0;0.0;;
    0.0;0.0;0.0;;
}
Material RedMat {                                // material #1
    1.0;0.0;0.0;1.0;;
    0.0;
    0.0;0.0;0.0;;
    0.0;0.0;0.0;;
}
// square
Mesh Square {
    // front face and back face
    8;                                // number of vertices
    1.0; 1.0; 0.0;;                    // vertice 0
    -1.0; 1.0; 0.0;;                  // vertice 1
    -1.0;-1.0; 0.0;;                 // vertice 2
    1.0;-1.0; 0.0;                   // vertice 3
    1.0; 1.0; 0.0;;                  // vertice 4
    1.0;-1.0; 0.0;;                  // vertice 5
    -1.0;-1.0; 0.0;;                 // vertice 6
    -1.0; 1.0; 0.0;;                 // vertice 7
    4;                                // number of triangles
    3;0,1,2;;                         // triangle #1
    3;0,2,3;;                         // triangle #2
    3;4,5,6;;                         // triangle #3
    3;4,6,7;;                         // triangle #4
}
MeshMaterialList {
    2;                                // two material
    4;                                // four faces
    0,                                // face #0 use material #0
    0,                                // face #1 use material #0
    1,                                // face #0 use material #0
    1;;                               // face #1 use material #0
    {GreenMat}
```

```
{RedMat}
}
}
```

{GreenMat} and {RedMat} are the references for the Material objects.

## NORMALS

Even though Direct3D (since version 7.0) performs lighting calculations on all vertices, including those without vertex normals, you should integrate normals with the MeshNormals template to get the desired lighting.

```
MeshTextureCoords {
    8;                      // 4 texture coords
    1.0; 1.0;;              // coord 0
    1.0; 0.0;;              // coord 1
    0.0; 0.0;;              // coord 2
    0.0; 1.0;;              // coord 3
    1.0; 1.0;;              // coords 4
    1.0;-1.0;;              // coords 5
    -1.0;-1.0;;             // coords 6
    -1.0; 1.0;;              // coords 7
}
template MeshNormals {
<f6f23f43-7686-11cf-8f52-0040333594a3>
    DWORD nNormals;
    array Vector normals[nNormals];
    DWORD nFaceNormals;
    array MeshFace faceNormals[nFaceNormals];
}
```

The vertex normal vector is used in Gouraud shading mode (default shading mode since Direct3D 6.0), to control lighting and create some texturing effects.

## TEXTURES

Textures could be referenced in a TextureFilename data object as child objects of the Material object:

```
template TextureFilename {
```

### NOTE

If you'd like to give an object a rounded look, for example, as in the example in Chapter 6, you'll have to specify the vertex normals. That is described in the Essentials part of this book.

```
    STRING filename;  
}
```

The reference to the texture file name has to be incorporated in the Material object like this:

```
Material GreenMat { // material #0  
    0.0;1.0;0.0;1.0;;  
    0.0;  
    0.0;0.0;0.0;;  
    0.0;0.0;0.0;;  
    TextureFilename{"wall.bmp";}  
}
```

Additionally, you need to specify texture coordinates:

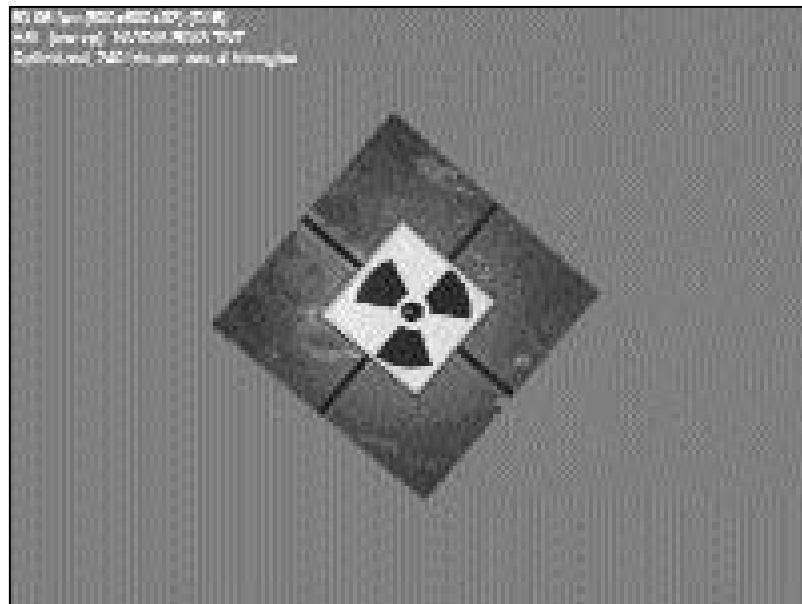
```
template MeshTextureCoords {  
    <f6f23f40-7686-11cf-8f52-0040333594a3>  
    DWORD nTextureCoords;  
    array Coords2d textureCoords[nTextureCoords];  
}  
template Coords2d {  
    <f6f23f44-7686-11cf-8f52-0040333594a3>FLOAT u;  
    FLOAT v;  
}
```

The first variable holds the number of vertices that have to be used in conjunction with the texture coordinates. The following array holds the tu/tv pairs of the textures. To map a texture to that square, we could use the following texture coordinates.

If you'd like to map one texture on the whole square, you have to use this:

```
MeshTextureCoords {  
    4;           // 4 texture coords  
    1.0; 1.0;;   // coord 0  
    1.0; 0.0;;   // coord 1  
    0.0; 0.0;;   // coord 2  
    0.0; 1.0;;   // coord 3  
}
```

The bottom right corner is (1.0f, 1.0f) and the upper left corner is (0.0f, 0.0f), regardless of the actual size of the texture, even if the texture is wider than it is tall.



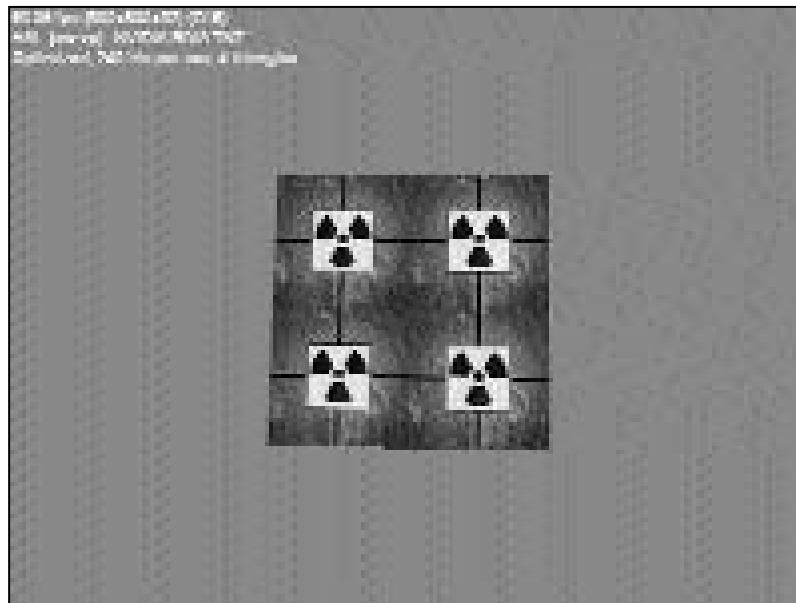
**Figure 9.4:** Texture coordinates for one texture

To get four textures, use the following coordinates:

```
MeshTextureCoords {  
    4;  
    // four textures  
    1.0; 1.0;;           // coord  0  
    -1.0; 1.0;;          // coord  1  
    -1.0;-1.0;;          // coord  2  
    1.0;-1.0;;          // coord  3  
}
```

This example is driven by the default texture wrapping mode that I explained in Part 2. The complete file square3.x looks like this:

```
xof 0302txt 0064  
Material GreenMat {                      // material #0  
    0.0;1.0;0.0;1.0;;  
    0.0;  
    0.0;0.0;0.0;;  
    0.0;0.0;0.0;;  
    TextureFilename{“wall.bmp”;}  
}  
// square
```



**Figure 9.5:** Texture coordinates for four textures

```
0;;                                // face #3 use material #0
{GreenMat}
}
MeshTextureCoords {
    8;                            // 4 texture coords
    1.0; 1.0;;                   // coord 0
    1.0; 0.0;;                   // coord 1
    0.0; 0.0;;                   // coord 2
    0.0; 1.0;;                   // coord 3
    1.0; 1.0;;                   // coords 4
    1.0;-1.0;;                  // coords 5
    -1.0;-1.0;;                 // coords 6
    -1.0; 1.0;;                  // coords 7
}
MeshNormals {
    2;                            // 2 normals
    0.0; 0.0; 1.0;;              // normal #1
    0.0; 0.0;-1.0;;              // normal #2
    4;                            // 4 faces
    3;0,0,0;;,
    3;0,0,0;;,
    3;1,1,1;;,
    3;1,1,1;;
}
}
```

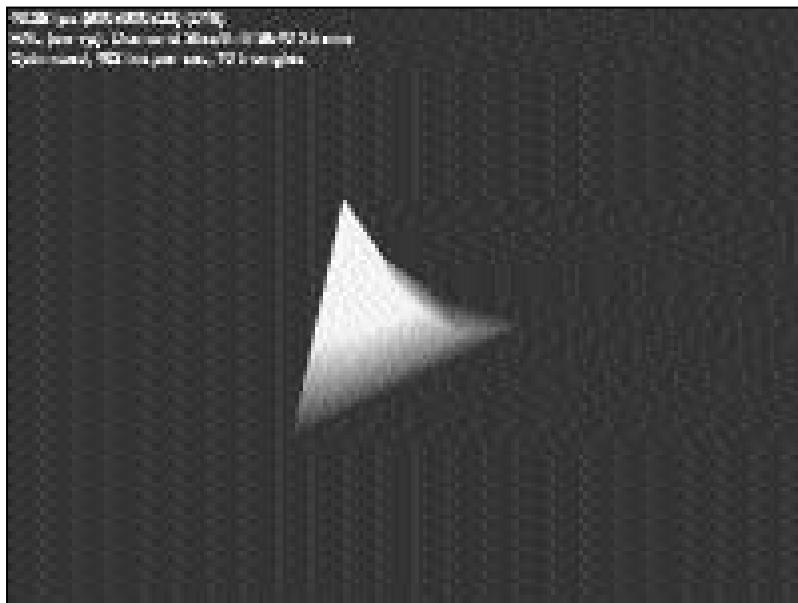
The Optimized Mesh example of the DirectX SDK blends the material and the texture so that the texture looks green. This alpha blending effect is described in Part 2. Textures with the same texture names won't be loaded twice; a flag will handle internally duplicated textures. OK, let's start an exercise: You produce an X file out of the specs of the object in Chapter 6. NO cheating . . . finished? OK, here's my version:

```
xof 0302txt 0064
// boid
Mesh boid {
    // vertices
    7;                            // number of vertices
    0.00; 0.00; 0.50;;            // vertice 0
    0.50; 0.00;-0.50;;           // vertice 1
    0.15; 0.15;-0.35;;           // vertice 2
    -0.15; 0.15;-0.35;;          // vertice 3
    0.15;-0.15;-0.35;;           // vertice 4
    -0.15;-0.15;-0.35;;          // vertice 5
    0.00;-0.50; 0.00;;           // vertice 6
}
```

```
-0.50; 0.00;-0.50;;
10;                                // number of triangles
3;0,1,2;;                           // triangle #1
3;0,2,3;;                           // triangle #2
3;0,3,6;;                           // triangle #3
3;0,4,1;;                           // ...
3;0,5,4;;
3;0,6,5;;
3;1,4,2;;
3;2,4,3;;
3;3,4,5;;
3;3,5,6;;
MeshMaterialList {
    1;                                // one material
    10;                               // ten faces
    0,                                // face #0 use material #0
    0;;                               // material #0
    Material {
        1.0;0.92;0.0;1.0;;
        1.0;
        0.0;0.0;0.0;;
        0.0;0.0;0.0;;
    }
    MeshNormals {
        9;                                // 9 normals for every face
        0.2; 1.0; 0.0;;                  // normal #1
        0.1; 1.0; 0.0;;                  // normal #2
        0.0; 1.0; 0.0;;                  // normal #3
        -0.1; 1.0; 0.0;;                 // normal #4
        -0.2; 1.0; 0.0;;                 // normal #5
        -0.4; 0.0;-1.0;;                // normal #6
        -0.2; 0.0;-1.0;;                // normal #7
        0.2; 0.0;-1.0;;                // normal #8
    }
}
```

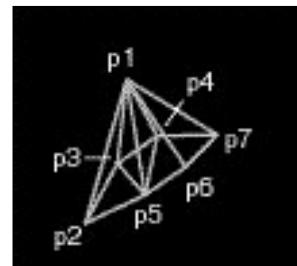
```
0.4; 0.0;-1.0;;           // normal #9
9;                      // nine faces
3; 0, 1, 2;;
3; 0, 2, 3;;
3; 0, 3, 4;;
3; -4, -2, -3;;
3; -4, -1, -2;;
3; -4, -0, -1;;
3; 5, 6, 6;;
3; 6, 6, 7;;
3; 7, 6, 7;;
3; 7, 7, 8;;
}
}
}
}
```

And here's the screen shot:



**Figure 9.6:** Super Star Destroyer

The tricky thing is to set the normals manually.



**Figure 9.7:** Wireframe view of the model

I have labeled the seven points of the object with p1 through p7. The vertices of the faces are placed at these points. On point 1 lies v0 and v5; at point 2, v1 and v6; at point 3, v3 and v11; and so on.

```
p1 = v0 = v5  
p2 = v1 = v6 = v10;  
p3 = v2 = v11;  
p4 = v3 = v12;  
p5 = v7 = v15;  
p6 = v8 = v14;  
p7 = v4 = v9 = v13;
```

So, the faces consist of the following vertices:

```
Face #1: v0, v1, v2  
Face #2: v0, v2, v3  
Face #3: v0, v3, v4  
Face #4: v5, v7, v6  
Face #5: v5, v8, v7  
Face #6: v5, v9, v8  
Face #7: v10, v15, v11  
Face #8: v11, v15, v12  
Face #9: v12, v15, v14  
Face #10: v12, v14, v13
```

The are nine normals in the 1steps.cpp of the example in the Next Steps to Animation section of Chapter 6. These normals have to be used by the following vertices:

```
v0 = n1  
v1 = n2  
v2 = n3  
v3 = n4  
v4 = n5  
v5 = -n5
```

```
v6 = -n4  
v7 = -n3  
v8 = -n2  
v9 = -n1  
v10 = n6  
v11 = n7  
v12 = n8  
v13 = n9  
v14 = n8  
v15 = n7
```

So, the faces have the following normals:

```
Face #1: 0, 1, 2  
Face #2: 0, 2, 3  
Face #3: 0, 3, 4  
Face #4: -4, -2, -3  
Face #5: -4, -1, -2  
Face #6: -4, -0, -1  
Face #7: 5, 6, 6  
Face #8: 6, 6, 7  
Face #9: 7, 6, 7  
Face #10: 7, 7, 8
```

If you use an X file converter such as that provided in the DirectX 8.0 SDK for 3D Studio Max and Maya, you won't have to set your normals manually.

So after getting the big picture on creating X file geometry manually, we delve a little deeper into the requirements for animation.

## TRANSFORMATION MATRICES

To transform parts of an object independently from each other, you have to divide the model into different frames. For example, a tank could turn its cannon up and down and to the left or right side. Its chains and wheels will turn when it drives through the wild enemy terrain. So there is a frame for the hull, for the turret, for the cannon, and one for the chains and wheels. Every frame will hold the matrix for the specified part of the tank. A Frame data object is expected to take the following structure:

```
Frame Hull {  
    FrameTransformMatrix {  
        1.000000, 0.000000, 0.000000, 0.000000,  
        0.000000, 1.000000, 0.000000, 0.000000,  
        0.000000, -0.000000, 1.000000, 0.000000,
```

```
206.093353, -6.400993, -31.132195, 1.000000;;  
}  
Mesh Hull {  
    2470;  
    41.310013; -26.219450; -113.602348;;  
    ...  
Frame Wheels_L {  
    FrameTransformMatrix {  
        1.000000, 0.000000, 0.000000, 0.000000,  
        0.000000, 1.000000, -0.000000, 0.000000,  
        0.000000, 0.000000, 1.000000, 0.000000,  
        -56.020325, -31.414078, 3.666503, 1.000000;;  
    }  
    Mesh Wheels_L {  
        2513;  
        -4.642166; -11.402874; -98.607910;;  
        ...  
    }  
    Frame Wheels_R {  
        FrameTransformMatrix {  
            1.000000, 0.000000, 0.000000, 0.000000,  
            0.000000, 1.000000, -0.000000, 0.000000,  
            0.000000, 0.000000, 1.000000, 0.000000,  
            56.687805, -31.414078, 3.666503, 1.000000;;  
        }  
        Mesh Wheels_R {  
            2513;  
            4.642181; -11.402874; -98.607910;;  
        }  
        Frame Turret {  
            FrameTransformMatrix {  
                1.000000, 0.000000, 0.000000, 0.000000,  
                0.000000, 1.000000, 0.000000, 0.000000,  
                0.000000, 0.000000, 1.000000, 0.000000,  
                -2.077148, 84.137527, 29.323750, 1.000000;;  
            }  
            Mesh Turret {  
                2152;  
                52.655853; -36.225544; -16.728998;;  
                ...  
            }  
        }  
    }  
}
```

The templates used in this sample in rmxftmpl.x are:

```
template Frame {
```

```
<3D82AB46-62DA-11cf-AB39-0020AF71E433>
[...]
}
template FrameTransformMatrix {
    Matrix4x4 frameMatrix;
}
template Matrix4x4 {
    array FLOAT matrix[16];
}
```

As an old DirectX Graphics programming freak, you might notice the use of 4-by-4 matrices as described in Chapter 6. Another simple sample you should look at is the file car.x in the d:\mssdk\samples\Multimedia\Media directory. And here's an advanced version of the X file of the sample program boidsy2.x:

```
xof 0302txt 0064
Frame BOID_Root {
    FrameTransformMatrix {
        1.000000, 0.000000, 0.000000, 0.000000,
        0.000000, 1.000000, 0.000000, 0.000000,
        0.000000, 0.000000, 1.000000, 0.000000,
        0.000000, 0.000000, 0.000000, 1.000000;;
    }
    // boid
    Mesh boid {
        // vertices
        7;                                // number of vertices
        0.00; 0.00; 0.50;;                  // vertice 0
        0.50; 0.00;-0.50;;                // vertice 1
        0.15; 0.15;-0.35;;                // vertice 2
        -0.15; 0.15;-0.35;;               // vertice 3
        0.15;-0.15;-0.35;;,
        -0.15;-0.15;-0.35;;,
        -0.50; 0.00;-0.50;;
        10;                                // number of triangles
        3;0,1,2;;                         // triangle #1
        3;0,2,3;;                         // triangle #2
        3;0,3,6;;                         // triangle #3
        3;0,4,1;;                         // ...
        3;0,5,4;;
        3;0,6,5;;
        3;1,4,2;;
```

```
3;2,4,3;;
3;3,4,5;;
3;3,5,6;;
MeshMaterialList {
    1;                                // one material
    10;                               // ten faces
    0,0;;                            // face #1 use material #0
    0,0;;                            // face #2 use material #0
    0,0;;                            // face #3 use material #0
    0,0;;                            // face #4 use material #0
    0,0;;                            // face #5 use material #0
    0,0;;                            // face #6 use material #0
    0,0;;                            // face #7 use material #0
    0,0;;                            // face #8 use material #0
    0,0;;                            // face #9 use material #0
    0,0;;                            // face #10 use material #0
    Material {                      // material #0
        1.0;0.92;0.0;1.0;;
        1.0;
        0.0;0.0;0.0;;
        0.0;0.0;0.0;;
    }
    MeshNormals {
        9;                                // 9 normals for every face
        0.2; 1.0; 0.0;;                  // normal #0
        0.1; 1.0; 0.0;;                  // normal #1
        0.0; 1.0; 0.0;;                  // normal #2
        -0.1; 1.0; 0.0;;                 // normal #3
        -0.2; 1.0; 0.0;;                 // normal #4
        -0.4; 0.0;-1.0;;                // normal #5
        -0.2; 0.0;-1.0;;                // normal #6
        0.2; 0.0;-1.0;;                // normal #7
        0.4; 0.0;-1.0;;                // normal #8
        9;                                // nine faces
        3; 0, 1, 2;;
        3; 0, 2, 3;;
        3; 0, 3, 4;;
        3; -4, -2, -3;;
        3; -4, -1, -2;;
        3; -4, -0, -1;;
        3; 5, 6, 6;;
    }
}
```

```
    3; 6, 6, 7;;
    3; 7, 6, 7;;
    3; 7, 7, 8;;
}
}
}
}
}
```

## ANIMATION

To animate an X file, you have to provide an animation set of animations with a reference to the proper frame matrix. The example file square4.x is, as usual, as simple as possible. Just load it into the Skinned Mesh example of the DirectX 8.0 SDK. It translates the square on its own x-axis.

```
xof 0302txt 0064
Frame SQUARE_Root {
    FrameTransformMatrix {
        1.000000, 0.000000, 0.000000, 0.000000,
        0.000000, 1.000000, 0.000000, 0.000000,
        0.000000, 0.000000, 1.000000, 0.000000,
        0.000000, 0.000000, 0.000000, 1.000000;;
    }
    Material GreenMat {                               // material #0
        0.0;1.0;0.0;1.0;;
        0.0;
        0.0;0.0;0.0;;
        0.0;0.0;0.0;;
        TextureFilename{"wall.bmp";}
    }
    // square
    Mesh Square {
        // front face and back face
        8;                                // number of vertices
        1.0; 1.0; 0.0;;                      // vertice 0
        -1.0; 1.0; 0.0;;                     // vertice 1
        -1.0;-1.0; 0.0;;                    // vertice 2
        1.0;-1.0; 0.0;;                     // vertice 3
        1.0; 1.0; 0.0;;                     // vertice 4
        1.0;-1.0; 0.0;;                     // vertice 5
        -1.0;-1.0; 0.0;;                    // vertice 6
        -1.0; 1.0; 0.0;;                    // vertice 7
    }
}
```

```
4;                                // number of triangles
3;0,1,2;;                         // triangle #1
3;0,2,3;;                         // triangle #2
3;4,5,6;;                         // triangle #3
3;4,6,7;;                         // triangle #4

MeshMaterialList {
    1;                                // one material
    4;                                // one face
    0,                                // face #0 use material #0
    0,                                // face #1 use material #0
    0,                                // face #2 use material #0
    0;;                               // face #3 use material #0
    {GreenMat}
}

MeshTextureCoords {
    8;                                // 4 texture coords
    1.0; 1.0;;                         // coord 0
    1.0; 0.0;;                         // coord 1
    0.0; 0.0;;                         // coord 2
    0.0; 1.0;;                         // coord 3
    1.0; 1.0;;                         // coords 4
    1.0;-1.0;;                        // coords 5
    -1.0;-1.0;;                        // coords 6
    -1.0; 1.0;;                        // coords 7
}

MeshNormals {
    2;                                // 2 normals
    0.0; 0.0; 1.0;;                   // normal #1
    0.0; 0.0;-1.0;;                  // normal #2
    4;                                // 4 faces
    3;0,0,0;;
    3;0,0,0;;
    3;1,1,1;;
    3;1,1,1;;
}

AnimationSet {
    Animation {
        AnimationKey {
            4;
```

```
    5;           // 5 keys
    0;16;1.0,0,0,0,0,0,0,
               0.0,1.0,0,0,0,0,
               0.0,0,0,1.0,0,0,
               0.1,0.0,0,0,1.0;;,
    80;16;1.0,0,0,0,0,0,0,
               0.0,1.0,0,0,0,0,
               0.0,0,0,1.0,0,0,
               0.2,0.0,0,0,1.0;;,
   160;16;1.0,0,0,0,0,0,0,
               0.0,1.0,0,0,0,0,
               0.0,0,0,1.0,0,0,
               0.3,0.0,0,0,1.0;;,
   240;16;1.0,0,0,0,0,0,0,
               0.0,1.0,0,0,0,0,
               0.0,0,0,1.0,0,0,
               0.2,0.0,0,0,1.0;;,
   320;16;1.0,0,0,0,0,0,0,
               0.0,1.0,0,0,0,0,
               0.0,0,0,1.0,0,0,
               0.1,0.0,0,0,1.0;;,
}
{SQUARE_Root}
}
}
```

This small example shows you the use of five animation keys, each consisting of a 4-by-4 matrix. The only value that changes is the first value in the fourth row—as you already know, the x-position of the square. {SQUARE\_ROOT} is the reference to the frame matrix. The 4 indicates that we are using matrix keys. It's the so-called keyType. The DirectX 8.0 SDK documentation indicates that the keyType member specifies whether the keys are rotation, scale, position, or matrix keys. The numbers to specify the correct keyType ranges in the documentation from 0–3, whereas the tiny.x example in the media directory uses 4 as the number to indicate the use of matrix keys.

## USING X FILES

There are two main ways provided in the DirectX 8.0 SDK to load an X file. The first one works with the CD3DFile class, which uses the CD3DFrame class; the second one works with the CD3DMesh class to load the X files. Both are located in the d3dfile.cpp file.

### TIP

If you want to keep the frame hierarchy, you have to use the CD3DFile class.

I will concentrate on the CD3DMesh class here:

```
class CD3DMesh
{
public:
    TCHAR           m_strName[512];
    LPD3DXMESH     m_pSysMemMesh;      // SysMem mesh, lives through resize
    LPD3DXMESH     m_pLocalMesh;       // Local mesh, rebuilt on resize
    DWORD          m_dwNumMaterials;   // Materials for the mesh
    D3DMATERIAL8*  m_pMaterials;
    LPDIRECT3DTEXTURE8* m_pTextures;
    BOOL           m_bUseMaterials;

public:
    // Rendering
    HRESULT Render( LPDIRECT3DDEVICE8 pd3dDevice,
                    BOOL bDrawOpaqueSubsets = TRUE,
                    BOOL bDrawAlphaSubsets = TRUE );

    // Mesh access
    LPD3DXMESH GetSysMemMesh() { return m_pSysMemMesh; }
    LPD3DXMESH GetLocalMesh()  { return m_pLocalMesh; }

    // Rendering options
    VOID UseMeshMaterials( BOOL bFlag ) { m_bUseMaterials = bFlag; }
    HRESULT SetFVF( LPDIRECT3DDEVICE8 pd3dDevice, DWORD dwFVF );
    // Initializing
    HRESULT RestoreDeviceObjects( LPDIRECT3DDEVICE8 pd3dDevice );
    HRESULT InvalidateDeviceObjects();
    // Creation/destruction
    HRESULT Create( LPDIRECT3DDEVICE8 pd3dDevice, TCHAR* strFilename );
    HRESULT Create( LPDIRECT3DDEVICE8 pd3dDevice, LPDIRECTXFILEDATA pFileData );
    HRESULT Destroy();
    CD3DMesh( TCHAR* strName = _T("CD3DFile_Mesh") );
    virtual ~CD3DMesh();

};
```

To open an X file with the help of the CD3DMesh class, you need at least the following method calls in InitDeviceObjects():

```
m_pObject->Create(m_pd3dDevice, _T("boidy2.x"));
m_pObject->SetFVF(m_pd3dDevice, FVF_CUSTOMVERTEX);
m_dwNumVertices  = m_pObject->GetSysMemMesh()->GetNumVertices();
m_dwNumFaces    = m_pObject->GetSysMemMesh()->GetNumFaces();
m_pObject->GetSysMemMesh()->GetVertexBuffer( &m_pVB );
m_pObject->GetSysMemMesh()->GetIndexBuffer( &m_pIB );
```

This code creates the X file by filling the vertex buffer and the index buffer, setting the flexible vertex format you would like to use, and telling you the number of vertices and faces to provide in the `DrawIndexedPrimitive()` method call.

You might render the X file with a call to

```
m_pObject->Render(pd3dDevice);
```

or use your own `Render()` routine.

There are methods provided for the `RestoreDeviceObjects()`/`InvalidateDeviceObjects()` method pair. To destroy the object, you might call

```
m_pObject->Destroy();
```

After that first overview, we will explore further. The `Create()` method of the `CD3DMesh` class uses `D3DXLoadMeshFromXof()` from the Direct3DX library to load the X file.

```
HRESULT CD3DMesh::Create( LPDIRECT3DDEVICE8 pd3dDevice,
                           LPDIRECTXFILEDATA pFileData )
{
    LPD3DXBUFFER pMtrlBuffer = NULL;
    HRESULT hr;
    // Load the mesh from the DXFILEDATA object
    hr = D3DXLoadMeshFromXof( pFileData, D3DXMESH_SYSTEMMEM, pd3dDevice,
                              NULL, &pMtrlBuffer, &m_dwNumMaterials,
                              &m_pSysMemMesh );
    if( FAILED(hr) )
        return hr;
    // Get material info for the mesh
    // Get the array of materials out of the buffer
    if( pMtrlBuffer && m_dwNumMaterials > 0 )
    {
        // Allocate memory for the materials and textures
        D3DXMATERIAL* d3dxMtrls = (D3DXMATERIAL*)pMtrlBuffer->GetBufferPointer();
        m_pMaterials = new D3DMATERIAL8[m_dwNumMaterials];
        m_pTextures = new LPDIRECT3DTEXTURE8[m_dwNumMaterials];
        // Copy each material and create its texture
        for( DWORD i=0; i<m_dwNumMaterials; i++ )
        {
            // Copy the material
            m_pMaterials[i] = d3dxMtrls[i].MatD3D;
            m_pMaterials[i].Ambient = m_pMaterials[i].Diffuse;
            m_pTextures[i] = NULL;
        }
    }
}
```

```
// Create a texture
if( d3dxMtrls[i].pTextureFilename )
{
    TCHAR strTexture[MAX_PATH];
    CHAR strTextureANSI[MAX_PATH];
    DXUtil_ConvertGenericStringToAnsi( strTextureANSI,
d3dxMtrls[i].pTextureFilename );
    DXUtil_FindMediaFile( strTexture, strTextureANSI );
    if( FAILED( D3DXCreateTextureFromFile( pd3dDevice, strTexture,
                                         &m_pTextures[i] ) ) )
        m_pTextures[i] = NULL;
}
}
SAFE_RELEASE( pMtrlBuffer );
return S_OK;
}
```

Most of the work is done by `D3DXLoadMeshFromXof()`. It reads out the whole geometry and material data and fills a LPD3DXMESH interface, which holds a vertex and index buffer, via `m_pSysMemMesh`. A detailed overview of the myriads of parameters possible for this call is available in the DirectX 8.0 SDK documentation. There is just one thing to mention: Using `D3DXMESH_SYSTEMMEM` (`D3DPOOL_SYSTEMMEM`) as the second parameter places the vertex and index buffers into system memory. This memory does not need to be recreated when a device is lost (`InitDeviceObjects()`/`DeleteDeviceObjects()`), so it's easier to use, but it might not be as fast as using the `D3DDXMESH_MANAGED` (`D3DPOOL_MANAGED`) flag, which copies the memory resources automatically to device-accessible memory as needed.

---

A work-around could be to create your own vertex and index buffers with the flag `D3DPOOL_MANAGED` like this:

```
{
    LPDIRECT3DVERTEXBUFFER8 pMeshSourceVB;
    LPDIRECT3DINDEXBUFFER8 pMeshSourceIB;
    CUSTOMVERTEX*          pSrc;
    CUSTOMVERTEX*          pDst;
    m_pFont->InitDeviceObjects( m_pd3dDevice );

    // load the file based objects
    if( FAILED(m_oYellow.mesh->Create( m_pd3dDevice, _T("boidy2.x") ) ) )
        return E_FAIL;
```

```
if( FAILED(m_oRed.mesh->Create( m_pd3dDevice, _T("boidr2.x") ) ) )
    return E_FAIL;
// Set the FVF type to match the vertex format we want
m_oYellow.mesh->SetFVF( m_pd3dDevice, FVF_CUSTOMVERTEX );
m_oRed.mesh->SetFVF( m_pd3dDevice, FVF_CUSTOMVERTEX );
// Get the number of vertices and faces for the meshes
m_dwNumVertices = m_oYellow.mesh->GetSysMemMesh()->GetNumVertices();
m_dwNumFaces = m_oYellow.mesh->GetSysMemMesh()->GetNumFaces();
m_dwNumVertices2 = m_oRed.mesh->GetSysMemMesh()->GetNumVertices();
m_dwNumFaces2 = m_oRed.mesh->GetSysMemMesh()->GetNumFaces();
// Create the vertex and index buffers
m_pd3dDevice->CreateVertexBuffer( m_dwNumVertices * sizeof(CUSTOMVERTEX),
                                    D3DUSAGE_WRITEONLY, 0,
                                    D3DPPOOL_MANAGED,
                                    &m_pVB );
m_pd3dDevice->CreateVertexBuffer( m_dwNumVertices2 * sizeof(CUSTOMVERTEX),
                                    D3DUSAGE_WRITEONLY, 0,
                                    D3DPPOOL_MANAGED,
                                    &m_pVB2 );
m_pd3dDevice->CreateIndexBuffer( m_dwNumFaces * 3 * sizeof(WORD),
                                   D3DUSAGE_WRITEONLY,
                                   D3DFMT_INDEX16, D3DPPOOL_MANAGED,
                                   &m_pIB );
m_pd3dDevice->CreateIndexBuffer( m_dwNumFaces2 * 3 * sizeof(WORD),
                                   D3DUSAGE_WRITEONLY,
                                   D3DFMT_INDEX16, D3DPPOOL_MANAGED,
                                   &m_pIB2 );
// Copy vertices for yellow object
m_oYellow.mesh->GetSysMemMesh()->GetVertexBuffer( &pMeshSourceVB );
m_pVB->Lock( 0, 0, (BYTE**)&pDst, 0 );
pMeshSourceVB->Lock( 0, 0, (BYTE**)&pSrc, 0 );
memcpy( pDst, pSrc, m_dwNumVertices * sizeof(CUSTOMVERTEX) );
m_pVB->Unlock();
pMeshSourceVB->Unlock();
pMeshSourceVB->Release();
// Copy vertices for red object
m_oRed.mesh->GetSysMemMesh()->GetVertexBuffer( &pMeshSourceVB );
m_pVB2->Lock( 0, 0, (BYTE**)&pDst, 0 );
pMeshSourceVB->Lock( 0, 0, (BYTE**)&pSrc, 0 );
```

```
    memcpy( pDst, pSrc, m_dwNumVertices2 * sizeof(CUSTOMVERTEX) );
    m_pVB2->Unlock();
    pMeshSourceVB->Unlock();
    pMeshSourceVB->Release();
    // Copy indices for yellow object
    m_oYellow.mesh->GetSysMemMesh()->GetIndexBuffer( &meshSourceIB );
    m_pIB->Lock( 0, 0, (BYTE**)&pDst, 0 );
    pMeshSourceIB->Lock( 0, 0, (BYTE**)&pSrc, 0 );
    memcpy( pDst, pSrc, 3 * m_dwNumFaces * sizeof(WORD) );
    m_pIB->Unlock();
    pMeshSourceIB->Unlock();
    pMeshSourceIB->Release();
    // Copy indices for red object
    m_oRed.mesh->GetSysMemMesh()->GetIndexBuffer( &meshSourceIB );
    m_pIB2->Lock( 0, 0, (BYTE**)&pDst, 0 );
    pMeshSourceIB->Lock( 0, 0, (BYTE**)&pSrc, 0 );
    memcpy( pDst, pSrc, 3 * m_dwNumFaces2 * sizeof(WORD) );
    m_pIB2->Unlock();
    pMeshSourceIB->Unlock();
    pMeshSourceIB->Release();
    return S_OK;
}
```

This might replace the `InitDeviceObjects()` method in the example I provided. A more elegant method would be to write your own `CD3DMesh` class with its own `Create()` method.

`D3DXCreateTextureFromFile()` returns a texture interface pointer from a file. The other interesting method of the `CD3DMesh` class is `SetFVF()`.

```
HRESULT CD3DMesh::SetFVF( LPDIRECT3DDEVICE8 pd3dDevice, DWORD dwFVF )
{
    LPD3DXMESH pTempSysMemMesh = NULL;
    LPD3DXMESH pTempLocalMesh = NULL;
    if( m_pSysMemMesh )
    {
        if( FAILED( m_pSysMemMesh->CloneMeshFVF( D3DXMESH_SYSTEMMEM, dwFVF,
                                                    pd3dDevice, &pTempSysMemMesh ) )
    )
        return E_FAIL;
    }
}
```

```
if( m_pLocalMesh )
{
    if( FAILED( m_pLocalMesh->CloneMeshFVF( 0L, dwFVF, pd3dDevice,
                                                &pTempLocalMesh ) ) )
    {
        SAFE_RELEASE( pTempSysMemMesh );
        return E_FAIL;
    }
}
SAFE_RELEASE( m_pSysMemMesh );
SAFE_RELEASE( m_pLocalMesh );
if( pTempSysMemMesh ) m_pSysMemMesh = pTempSysMemMesh;
if( pTempLocalMesh ) m_pLocalMesh = pTempLocalMesh;
// Compute normals in case the meshes have them
if( m_pSysMemMesh )
    D3DXComputeNormals( m_pSysMemMesh );
if( m_pLocalMesh )
    D3DXComputeNormals( m_pLocalMesh );
return S_OK;
}
```

CloneMeshFVF() clones a mesh by using the FVF (flexible vertex format) you provided. It clones into the system memory or local memory, and it also generates the normals with D3DXComputeNormals() for each vertex in the mesh.

That looks very easy—doesn't it? As with most of the DirectX 8.0 SDK examples, the X file access methods in d3dfile.cpp are a nice example to see how all this stuff works. If you write your own game engine, you will use your own file access routines and perhaps your own file format (for example, an extended X file format or one of the popular file formats used by 3-D shooters or landscape engines).

## THE EXAMPLE

The example in this section shows two X files, which are held totally independent from each other. Whereas in the example in Chapter 5 I used one geometry data model with two different materials, I decided to use here two geometry data objects. You might play around with it and load other X files; for example, use the tiny.x model and fly with your camera around it . . . that makes it fun.

Let's start looking at the used code.

Geometry, material, position, and orientation data are held by the following object structure:

```
struct Object
{
    D3DXVECTOR3 vLoc;           // Location/Translation
```

```
D3DXVECTOR3 vR;           // Rotation vector
D3DMATRIX matLocal;
D3DMATERIAL8 mtrl;        // material data
CD3DMesh* mesh;          // mesh data
};

};
```

This structure holds the CD3DMesh class shown above. It also holds the material data that is retrieved by this class. There is also a place to store its own world matrix and a location and rotation vector.

## INITDEVICEOBJECTS()

InitDeviceObjects() sets the FVF and returns the number of vertices and faces and the vertex and index buffers created in system memory. At the least, the materials for the two objects are built and stored.

```
HRESULT CMyD3DApplication::InitDeviceObjects()
{
    m_pFont->InitDeviceObjects( m_pd3dDevice );
    // load the file based objects
    if( FAILED(m_oYellow.mesh->Create( m_pd3dDevice, _T("boidy2.x") ) ) )
        return E_FAIL;
    if( FAILED(m_oRed.mesh->Create( m_pd3dDevice, _T("boidr2.x") ) ) )
        return E_FAIL;
    // Set the FVF type to match the vertex format we want
    m_oYellow.mesh->SetFVF( m_pd3dDevice, FVF_CUSTOMVERTEX );
    m_oRed.mesh->SetFVF( m_pd3dDevice, FVF_CUSTOMVERTEX );
    // Get the number of vertices and faces for the meshes
    m_dwNumVertices = m_oYellow.mesh->GetSysMemMesh()->GetNumVertices();
    m_dwNumFaces = m_oYellow.mesh->GetSysMemMesh()->GetNumFaces();
    m_dwNumVertices2 = m_oRed.mesh->GetSysMemMesh()->GetNumVertices();
    m_dwNumFaces2 = m_oRed.mesh->GetSysMemMesh()->GetNumFaces();
    m_oYellow.mesh->GetSysMemMesh()->GetVertexBuffer( &m_pVB );
    m_oRed.mesh->GetSysMemMesh()->GetVertexBuffer( &m_pVB2 );
    m_oYellow.mesh->GetSysMemMesh()->GetIndexBuffer( &m_pIB );
    m_oRed.mesh->GetSysMemMesh()->GetIndexBuffer( &m_pIB2 );
    D3DUtil_InitMaterial(m_oYellow.mtrl,
                         m_oYellow.mesh->m_pMaterials[0].Diffuse.r,
                         m_oYellow.mesh->m_pMaterials[0].Diffuse.g,
                         m_oYellow.mesh->m_pMaterials[0].Diffuse.b,
                         m_oYellow.mesh->m_pMaterials[0].Diffuse.a);
    D3DUtil_InitMaterial(m_oRed.mtrl,
                         m_oRed.mesh->m_pMaterials[0].Diffuse.r,
                         m_oRed.mesh->m_pMaterials[0].Diffuse.g,
```

```

        m_oRed.mesh->m_pMaterials[0].Diffuse.b,
        m_oRed.mesh->m_pMaterials[0].Diffuse.a);

    return S_OK;
}

```

## **RESTOREDEVICEOBJECTS() AND INVALIDATEDeviceOBJECTS()**

RestoreDeviceObjects() and InvalidateDeviceObjects() calls the CD3DMesh methods with the same names.

```

HRESULT CMyD3DApplication::RestoreDeviceObjects()
{
    m_pFont->RestoreDeviceObjects();
    m_oYellow.mesh->RestoreDeviceObjects( m_pd3dDevice );
    m_oRed.mesh->RestoreDeviceObjects( m_pd3dDevice );
    ...
}

HRESULT CMyD3DApplication::InvalidateDeviceObjects()
{
    m_pFont->InvalidateDeviceObjects();
    m_oYellow.mesh->InvalidateDeviceObjects();
    m_oRed.mesh->InvalidateDeviceObjects();
    return S_OK;
}

```

## **RENDER()**

As shown in the Basic and other examples, Render() is used to call the two DrawIndexedPrimitive() calls necessary to draw the both objects independently.

```

HRESULT CMyD3DApplication::Render()
{
    // Clear the viewport | z-buffer
    m_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
                           0x00000000, 1.0f, 0L );

    // Begin the scene
    if( SUCCEEDED( m_pd3dDevice->BeginScene() ) )
    {
        // yellow object
        m_pd3dDevice->SetMaterial( &m_oYellow.mtr1 );
        // Apply the object's local matrix
        m_pd3dDevice->SetTransform(D3DTS_WORLD, &m_oYellow.matLocal );

        m_pd3dDevice->SetStreamSource( 0, m_pVB, sizeof(CUSTOMVERTEX) );

```

```
m_pd3dDevice->SetVertexShader( FVF_CUSTOMVERTEX );
m_pd3dDevice->SetIndices( m_pIB, 0 );
m_pd3dDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                                     0,
                                     m_dwNumVertices,
                                     0,
                                     m_dwNumFaces);

// number of vertices

// number of primitives
// red object
m_pd3dDevice->SetMaterial( &m_oRed.mtr1 );
// Apply the object's local matrix
m_pd3dDevice->SetTransform(D3DTS_WORLD, &m_oRed.matLocal );

m_pd3dDevice->SetStreamSource( 0, m_pVB2, sizeof(CUSTOMVERTEX) );
m_pd3dDevice->SetVertexShader( FVF_CUSTOMVERTEX );
m_pd3dDevice->SetIndices( m_pIB2, 0 );
m_pd3dDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                                     0,
                                     m_dwNumVertices2,
                                     0,
                                     m_dwNumFaces2);

// number of vertices

// number of primitives
// Output statistics
m_pFont->DrawText( 2, 0, D3DCOLOR_ARGB(255,255,255,0), m_strFrameStats );
m_pFont->DrawText( 2, 20, D3DCOLOR_ARGB(255,255,255,0), m_strDeviceStats );
// End the scene.
m_pd3dDevice->EndScene();
}

return S_OK;
}
```

The number of vertices and faces and the material handles were retrieved in the `InitDeviceObjects()` method with the help of the `CD3DMesh` helper methods. The world matrix for the objects was built in the `FrameMove()` method.

The use of two vertex and index buffers is overkill for such small objects as in `boidy2.x` or `boidr2.x`, but they're useful for bigger objects.

After getting the big picture of the X file format, you might dig a little bit deeper into the examples of the DirectX 8.0 SDK. There are five examples dealing with X files:

- Enhanced Mesh or N-Patch
- RT-Patch Examples
- Optimized Mesh
- Progressive Mesh
- Skinned Mesh

These examples extend the X file loading routine or the X file format. You should not have any problems understanding these extensions.

## EXTENDING X FILES

One of the most fascinating aspects of the concept behind X files is extensibility. You may recall the file rmxftmpl.x, which holds the definitions of the templates. You might define your own templates in it and make your own X file format this way. For example, a shader template with all of the texturing stuff explained in Part 2 might be implemented this way with different parameters, and so on. Your files can hold their own shader classes, collision detection stuff, physics, specialized animations, paths to sound files, and more.

If you take a look at the following section, you might get an idea about what has been done in this area before. I'm sure you'll be thinking of building your own file format with this stuff in mind in the next few hours :-).

## ADDITIONAL RESOURCES

I've found the Mr. Gammemaker article at [www.tasteofhoney.freeserve.co.uk/loadxfiles.html](http://www.tasteofhoney.freeserve.co.uk/loadxfiles.html) very useful. Paul Bourke has compiled a must-have for every X file junkie: an X file reference at [www.swin.edu.au/astronomy/pbourke/3dformats/directx](http://www.swin.edu.au/astronomy/pbourke/3dformats/directx).

*This page intentionally left blank*

# **CHAPTER 10**

## **QUAKE 111 MODEL FILES**

**I**d Software has expressed that Quake III Arena should be the easiest game to edit. That's especially true for its model file format with the extension .md3.

So your first question is, What's my profit in using the .md3 files instead of, say, the X file format?

The .md3 format is an often-used file format, which has been tested extensively and has proven to be a good model format :-) in Quake III and a lot of games that use this engine. In diving into the .md3 format, you'll dive also into the Quake III coding universe. It's very impressive to learn from the guys at id Software how Quake III was made by looking at their models and their ideas. It wouldn't harm you to take a quick look at an approved model file spec before starting your own.

What can you do with this format, exactly?

You might support three levels of detail: In Quake III, 800–900 faces are used in the highest-detailed level, around 500 faces are used in the middle-detailed level, and 300 faces are used in the less-detailed level. You might use many more.

These model files store animations in frames with fixed names, different lengths, and different speeds for every move. This is a really fast method of animation.

Most of the Quake III models own different skins—a.k.a. different texture maps—that can be loaded on demand. You might use the same geometry for a static statue, a normal soldier, and an officer with a golden helmet, for example.

As usual, I have built a small example program that is able to load .md3 files. It's not really a finished .md3 viewer, but it gives you the basic knowledge to make it fully functional or to build your own file format and your own file format viewer with it.

This example will show you the loading of one .md3 file and how to use a right-handed coordinate system, which is used by all OpenGL applications. I've used the ArcBall class, provided in the Common files, so you might be able to turn the model with the mouse

## CAUTION

An important limitation of my example program is the following: I read out the name of the textures from the .md3 file, that has its own slot for the texture names used. As far as I see now, this place isn't used by all .md3 tools to store the texture names. There is a file with the extension .skin, which stores texture names too. To read out the texture names of this file seems to be the favorite method. There are .md3 models, where the texture name in the .md3 file has the .tga extension, whereas the texture name in the .skin file has the .jpg extension. Normally the texture names in the .skin file are more up to date. I haven't implemented a .skin file reader in this .md3 viewer.



**Figure 10.1:** Screen shot of lower.md3 of the Sarge model (courtesy of id Software)

in every direction you like. There are myriads of .md3 converter programs out there; some of them don't work very precisely. I have run into a few problems with .md3 files that are produced by these converter programs, and I'll show you the work-arounds in code in the following passages.

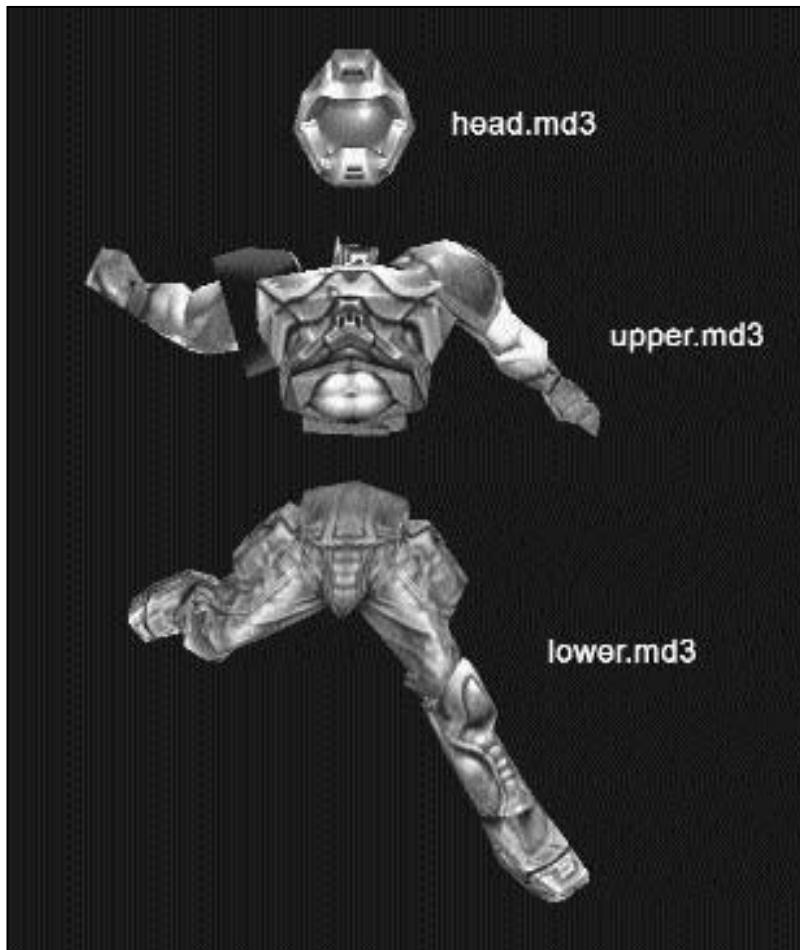
## FILES OF THE TRADE

An .md3 model is normally made of three body parts and the weapon model. With three or more body parts, it's possible to animate a person in a much more natural and smooth way. So we have to open three model files—if we are using a weapon, we have to open four—and put them together. In Quake III they are called head.md3, upper.md3, and lower.md3.

The parts are put together by so-called tags. There are often four tags that hold the model together, but there could be more or fewer.

Inside of the .md3 files, you can have any number of components. For example, in the Doom marine's head.md3, there is an h\_hat and an h\_visor. You could apply different textures or shader effects to certain components of the model. There's no limit to the number of components, but they have to follow the naming convention, which is h\_ for head components, u\_ for torso components, and l\_ for lower components.

The .md3 files store the character animation in *key frames*. It helps to imagine the use of sprites in old sprite engines. These were long lists of pictures called one after another so that the viewer gets the impression of a moving object, like those old celluloid films. The same thing happens in .md3 files with the



**Figure 10.2:** Doom marine  
(courtesy of id Software)

vertices of polygon objects. There's no limit for frames in the engine, but a maximum of 300 might be a good number. The animation file that stores the animation data is called `animation.cfg`. This file directs the call of the key frames.

Three different levels of detail are provided with three different `.md3` files for each model. The naming convention of the files for the three levels of detail is as follows: for the lowest detailed level, a `2` is placed after the file name; for the middle detailed level, a `1` is placed after it; whereas the highest detail level gets no additional number. So, `head.md3`, `head1.md3`, `head2.md3`, and so on.

The long pointy end of the triangle must face the front. The base of the triangle is the actual pivot point. It determines where the respective model will be rotated. This tag does not have material, so it's not visible.

The tags that connect the pieces together are just a `1` face geometry, which is built like Figure 10.3.

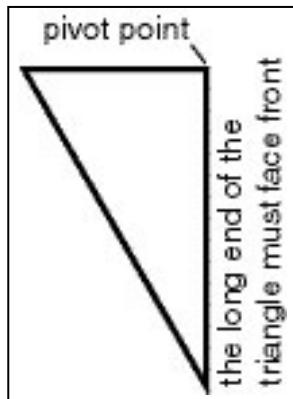


Figure 10.3: Tag

In every .md3 model there could be three tags, and the tag name corresponds to its location. The base of the neck is attached to the upper torso with tag\_head. At the base of the spine, the torso is attached to the legs with tag\_torso, and the weapon is attached to the hand of the model with tag\_weapon. Each model has at least one tag. The head.md3 has the tag\_head; the upper.md3 has the tag\_head, tag\_torso, and tag\_weapon; and the lower.md3 has the tag\_torso.

The tag\_head needs to be placed at the pivot point of the head bone, the tag\_torso at the pivot point of the base spine bone, and so on. You might tell your .md3 file application which tag to attach each object to with a .skin file. It's possible to attach the pieces in any way you like. You might also define different skins—texture maps of .tga or .jpg files—with different skin files, so you might be able to use the same geometry with different textures.

A visual advancement in the player model format is in the skins (the texture file, which is normally a .tga file or .jpg file, not the .skin file). Instead of being restricted to 8-bit palettes as in Quake II or to just one skin, we can use 32-bit color power and as many skins as we like. Texture sizes will work best as a power of 2, as a square size (think of the Voodoo 3 and older boards), and with a limit of 256-by-256-pixel skins. Every component of a model might get its own skin.

Shader files are the most exciting feature of .md3 models. With shader effects, you can control texture mapping, transparency, a lot of light parameters, and much more. In Quake III Arena, there is one shader file called models.shader.

To hear your model, you might attach sounds to it using .wav files with 22.5KHz, 16-bit, 1-channel (mono) format in any length (theoretically).

OK . . . now let's look at what we have learned.

There are a lot of files that make a Quake III model breathe and live:

- .md3 files: normally head.md3, upper.md3, and lower.md3
- Icon file: icon\_default.tga

#### NOTE

There is an equivalent provided with the DirectX 8 SDK, called effect file. It works in a different way, but has the same intention.

- Animation file: animation.cfg
- Skin files: .tga or .jpg
- Tag file: .skin
- Shader file: for example, models.shader
- Sound files: .wav

The .md3 files contain the geometry of the model, whereas the .tga or .jpg files hold the textures for the components of the .md3 files. The shader file controls the mapping of the texture and a lot more. With the help of the .skin file, you might tag more than one .md3 file together to form a model and define specialized skins—.tga or .jpg files—with it. To animate the model's different parts stored in their own .md3 files, an animation.cfg file exists. To hear your models speaking or shooting, you might provide .wav files with the proper sounds.

To look into the Quake III pak0.pk3 file, you have to rename it pak0.zip and use a Zip utility to see the directories and files. The files for the Biker .md3 model are stored in the following directory:

- The .md3 files, animation.cfg, .tga, .skin in *models/players/biker*
- The models.shader file for all models in *scripts*
- The .wav files in *sound/player/biker*

To see the content of a typical directory that holds a Quake III model, you might look at the models/... directories. You'll find at least the following files:

- animation.cfg
- head.md3
- upper.md3
- lower.md3
- head\_default.skin
- upper\_default.skin
- lower\_default.skin
- at least one texture map: .tga or .jpg
- icon\_default.tga (64-by-64 pixel, 24-bit image with alpha mask)

If you'd like to use the model in a Capture the Flag mode, you need additional CTF skins as well:

- head\_blue.skin
- head\_red.skin
- upper\_blue.skin
- upper\_red.skin
- lower\_blue.skin
- lower\_red.skin
- at least two texture maps, one for each color: .tga or .jpg
- icon\_blue.tga
- icon\_red.tga

If you'd like to use your .md3 model with other textures so that you have additional characters with the same geometry in your game, you might provide additional skin, icon, and texture map files:

- head\_charactername.skin
- upper\_charactername.skin
- lower\_charactername.skin
- the textures for the characters
- icon\_charactername.tga

Now let's examine the details of the different files.

## ANIMATION.CFG

The animation of Quake III models works with the so-called key frame. For this kind of animation, the same model is saved in different positions. The model will be drawn as a series of different models.

In the case of the Quake III models, the different positions of a model are saved in different vertices and tags. So if a model supports 246 different frames, there have to be 246 different vertex groups in every mesh of the model and 246 different tags, which connect the .md3 files, in every .md3 file. The torso, for example, often has three tags, so there have to be 3 times 246 different tags connected with the torso geometry.

Animation of Quake III files can generally be of any length and any frame rate. The naming convention must be exactly the same. So in some cases you might use different lengths of animation (more or fewer frames), but you have to use the same name for the different animations to use your models with the Quake III engine.

Now let's look at a animation.cfg file of the Biker model:

```
// animation config file
sex      m
// first frame, num frames, looping frames, frames per second

headoffset -4 0 0
footsteps boot
0          30     0      25           // BOTH_DEATH1
29         1      0      25           // BOTH_DEAD1
30         30     0      25           // BOTH_DEATH2
59         1      0      25           // BOTH_DEAD2
60         30     0      25           // BOTH_DEATH3
```

### NOTE

Instead of key frames, the skeletal animation of Half-Life or F.A.K.K. 2 works by giving a model a skeleton, which is set up with joints or bones to determine how the unit will animate.

89	1	0	25	// BOTH_DEAD3
90	40	0	20	// TORSO_GESTURE
130	6	0	15	// TORSO_ATTACK
136	6	0	15	// TORSO_ATTACK2
142	5	0	20	// TORSO_DROP
147	4	0	20	// TORSO_RAISE
151	1	0	15	// TORSO_STAND
152	1	0	15	// TORSO_STAND2
153	9	9	20	// LEGS_WALKCR
162	8	8	20	// LEGS_WALK
170	13	13	24	// LEGS_RUN
183	10	10	24	// LEGS_BACK
193	10	10	15	// LEGS_SWIM
203	7	0	15	// LEGS_JUMP
210	5	0	17	// LEGS_LAND
215	8	0	15	// LEGS_JUMPB
223	4	0	17	// LEGS_LANDB
227	10	10	15	// LEGS_IDLE
237	9	9	15	// LEGS_IDLECR
246	7	6	15	// LEGS_TURN

As far as I can see, any animation.cfg file is built like this: sex could be m for masculine, f for feminine, and n for none in the case of, for example, machines. It determines who the game will display messages about (for example, he or she) and which sound is used in the absence of custom sounds.

There are two optional variables named footsteps and headoffset. Footsteps could be boot, normal, energy, mech, or flesh, which is used for the sound of footsteps. The headoffset is the offset of the HUD in x, y, and z coordinates.

Farther down comes the frame-based animation script. The numbers in the first four columns change, but not the number of rows or paragraphs. The first column shows the first frame, where the animation with the name after the double slash starts. The second column shows the number of frames; it's the length of the animation. A few animations are so-called looping animations. The number of frames that will be looped stands in the third column. The speed of the animation is shown in the fourth column, and the

name of the animation is located in the last column. The names of the animations are always the same in every model.

Now, a little deeper: The BOTH\_DEATH\* animations are for both the legs and upper torso objects. The last frame of a death animation is located in the BOTH\_DEAD\* animation.

The TORSO\_\* animations are for the—you guessed it—torso object. TORSO\_GESTURE can be of any length and even has its own sound. TORSO\_ATTACK is the attack animation for ranged weapons. It must be exactly six frames long, as it is synced to the weapon firing. The first frame of the animation should be the aim, the second frame is the fire and recoil, and the rest is just recovering to the first frame. TORSO\_ATTACK2 must be a six-frame-long attack sequence showing the attack with the gauntlet. The first frame should be the anticipation of the punch, the second frame the punch itself, and the rest is recovery. The TORSO\_DROP is the first part of the weapon switch and has to be exactly five frames long. The second part of the weapon switch is TORSO\_RAISE, which must be five frames long as well. The static pose holding a ranged weapon is animated by TORSO\_STAND, whereas TORSO\_STAND2 is the static pose holding the gauntlet.

The LEGS\_\* animations are for the legs. The LEGS\_WALKCR is the crouch walking cycle. There are walk (LEGS\_WALK), run (LEGS\_RUN), swim (LEGS\_SWIM), jump forward (LEGS\_JUMP), land forward (LEGS\_LAND), jump back (LEGS\_JUMPB), land back (LEGS\_LANDB), and backpedal or running backward cycles (LEGS\_BACK). There are idle (LEGS\_IDLE), idle for crouching (LEGS\_IDLECR), and a turn animation when the player rotates far enough for the legs to rotate (LEGS\_TURN).

All of the animation frames are stored by md3.cpp in arrays. You might call the proper animation by providing the frame number to that array. OK, now on to the skin.

## THE .SKIN FILE

All of the necessary tag and texture info is kept in the .skin files provided in your model directory. These files are just text files that hold the tags that the model is attached to and the skins. The name of the .skin file corresponds to the .md3 file that uses it. Here are the \_default.skin files of the Biker model:

head\_default.skin file:

```
tag_head,  
h_head,models/players/biker/biker_h.tga
```

upper\_default.skin file:

```
tag_head,  
tag_weapon,  
u_larm,models/players/biker/biker.tga  
u_rarm,models/players/biker/biker.tga  
u_torso,models/players/biker/biker.tga
```

```
tag_torso,  
  
lower_default.skin file:  
l_legs,models/players/biker/biker.tga  
tag_torso,
```

The head skin file describes the only tag that is provided with the head.md3 file and the texture that should be used for the head. The upper skin file describes three meshes and three tags, and the three meshes should be textured with the same .tga file. The last mesh in the lower.md3 will be textured with the biker.tga bitmap file and holds only one tag.

Textures are located in the same directory where the .md3 files reside. You can also find the names of the default textures in the .md3 file, as you might see in a few seconds. If you'd like to use custom textures, they have to be declared in a custom .skin file.

## TEXTURES AND THE SHADER FILE

The textures used for the .md3 files can have a color depth of 32 bits, should be in a power of 2 size, and should be limited to 256 pixels. Textures are shaded with the commands of the shader file, which is located in the Scripts directory. Every mesh of a model can have its own texture and its own shader properties.

The shader file is a text file, like the animation.cfg and the .skin files, that can be manipulated with a text editor. It's written in its own language with a ton of commands. Shaders are simple scripts that are used by the Quake III engine to determine the properties of the polygon surfaces and how they are rendered. The most common use of shaders is to control the properties of walls in levels. Shaders control the texturing of objects by providing commands that are similar to the texturing commands of OpenGL, the 3-D API that is used in the Quake series. The Direct3D equivalents of these texturing commands are shown in Part 2. You might browse around in the models.shader file in the scripts directory of your Quake III pak file.

Here's the anatomy of a shader file.

```
// a comment  
[the shader name]  
{  
    [surface attributes]  
  
    // a stage  
    {  
        [stage attributes]  
    }  
  
    [more stages]  
}
```

As you might suppose, comments are set with the standard C++ double slash. Every shader starts with its name consisting of less than 64 characters and lowercase letters. Slashes have to be forward slashes. The shader body is contained in curly brackets.

There are three attribute types: the *surface* attributes control the behavior of the overall polygon surface to which the shader is linked, for example, if the surface is water or lava. *Stage* or *content* attributes control the look of a surface. *Deformation* attributes deform the vertices of a surface in some way.

With the .md3 models, we might concentrate on the stage or content attributes. As far as I can see, only these are used in the models.shader file. To get a reference for all the shader commands used by id Software in Quake III, you might check out these files:

- [ftp://ftp.idsoftware.com/idstuff/quake3/tools/Q3Ashader\\_manual.doc](ftp://ftp.idsoftware.com/idstuff/quake3/tools/Q3Ashader_manual.doc)
- [maj.gamedesign.net/shaderDocs/HTML4+CSS/index.shtml](http://maj.gamedesign.net/shaderDocs/HTML4+CSS/index.shtml)

The first file is the one written by the id Software people. The second one was written before the official shader doc existed and is targeted to a more entry level.

Let's examine the commands for the Doom marine in the model.shader file. I've copied the shader scripts for the Doom models from this file.

```
models/players/doom/phobos_f
{
    {
        map textures/effects/tinfx.tga
        tcGen environment
        blendFunc GL_ONE GL_ZERO
        rgbGen lightingDiffuse
    }
    {
        map models/players/doom/phobos_f.tga
        blendFunc GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA
        rgbGen lightingDiffuse
    }
}
models/players/doom/phobos
{
    {
        map models/players/doom/phobos_fx.tga
        blendFunc GL_ONE GL_ZERO
        tcmod scale 7 7
        tcMod scroll 5 -5
        tcmod rotate 360
        rgbGen identity
    }
}
```

```
        }
    //}

    //      map textures/effects/tinfx2.tga
    //      tcGen environment
    //      blendFunc GL_ONE GL_ONE
    //      rgbGen lightingDiffuse
    //}
}

{
    map models/players/doom/phobos.tga
    blendFunc GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA
    rgbGen lightingDiffuse
}

models/players/doom/f_doom
{
{
    map models/players/doom/f_doom.tga
}

{
    map models/players/doom/fx_doom.tga
    tcGen environment
    rgbGen lightingDiffuse
    blendfunc gl_ONE gl_ONE
}

models/players/doom/doom_f
{
{
    map models/players/doom/doom_f.tga
    rgbGen lightingDiffuse
}

{
    map models/players/doom/doom_fx.tga
    tcGen environment
    rgbGen lightingDiffuse
    blendfunc gl_ONE gl_ONE
}
```

There are three texture maps for the Doom model. The first default one is doom.tga, the second default one is doom\_f.tga, and the third default one is doom\_fx.tga. The first texture is the one used for the upper and lower meshes, whereas the doom\_f.tga is used for the mesh h\_visor of the helmet. The third texture is used for environment mapping.

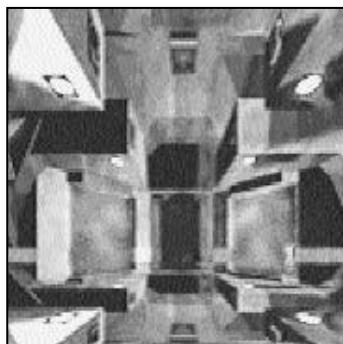
There are two other models that use the Doom marine geometry, with the textures called phobos.tga, phobos\_f.tga, and phobos\_fx.tga, and a statue that uses the same geometry with the textures doom\_statue.tga, doom\_f.tga, and no environment mapping texture.

So the geometry of the Doom model is used with three different texture packs. Now let's examine the phobos\_f model.

The line

```
map textures/effects/tinfx.tga
```

tells us that the following texture map has to be used for the model:



**Figure 10.4:** The environment map

This texture is a global effect texture, which is accessible by a lot of models, walls, and so on. The next line tells the renderer to produce the texture coordinates that are necessary for environment mapping.

```
tcGen environment
```

To blend the texture with diffuse lighting, these lines hold the OpenGL blending parameters:

```
blendFunc GL_ONE GL_ZERO  
rgbGen lightingDiffuse
```

The blend functions are the keyword commands that tell the Quake III graphic engine's renderer how graphic layers—a.k.a. different texture maps or texture maps with diffuse color or lighting, for example—are to be mixed together.

BlendFunc or Blend Function is the equation at the core of processing shader graphics. The formula reads as follows:

```
FinalColor = SourcePixelColor * SourceBlendFactor + DestPixelColor * DestBlendFactor
```

This is the alpha blending formula, as you know it from Part 2. *Source* is usually the RGB color data in a texture TGA file (remember, it's all numbers) modified by any rgbbgen and alphagen. In the shader, the source is generally identified by command MAP, followed by the name of the image. *Destination* is the color data currently existing in the frame buffer.

As you learned in Part 2, each texturing pass or stage of blending is combined in a cumulative manner with the color data passed on to it by the previous stage. How that data combines together depends on the values chosen for the Source Blends and Destination Blends at each stage. In the Quake III engine, any value for srcBlend other than GL\_ONE or GL\_SRC\_ALPHA (where the alpha channel is entirely white) will cause the source to become darker. There are a bunch of values for the srcBlend parameters. These commands are used by OpenGL as the blending parameters for texture mapping. Please check out the shader manual for the different meanings of these parameters and every OpenGL reference (search for glBlendFunc()). Keep in mind that Direct3D provides equivalent parameters. We'll concentrate on our small model example.

With the parameter GL\_ONE used here, the value of the source color information doesn't change when multiplied by source. The parameters for destination blending dstBlend are GL\_ZERO. This is the value 0. So multiplied by the destination, all RGB data in the destination becomes zero, or black.

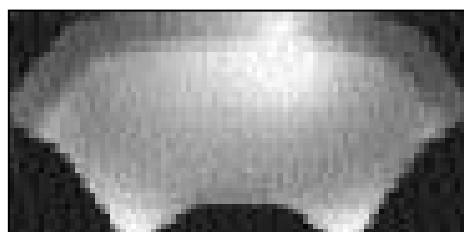
Back to our formula: The product of the source side of the equation is added to the product of the destination side of the equation. The sum is then placed into the frame buffer to become the destination for the next stage.

rgbGen lightingdiffuse illuminates the object by using the vertex normals.

So this first stage tells the renderer to blend the texture tinfox.tga by illuminating it with diffuse lighting. Besides that, the texture coordinates for environment mapping are created here.

Now on to the next stage. A second texture phobos\_f.tga should be blended with the first texture:

```
map models/players/doom/phobos_f.tga
```



**Figure 10.5:** The visor of the Doom model (courtesy of id Software)

Alpha blending should happen with the following OpenGL parameters:

```
blendFunc GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA
```

These are similar to the Direct3D parameters:

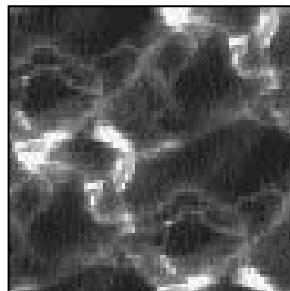
```
D3DBLEND_SRCALPHA D3DBLEND_INVSRCALPHA
```

Check out the multitexturing section in Chapter 8 for a description of these parameters.

The last attribute `rgbGen` tells the renderer to multiply alphablending by a value generated by applying Gouraud shaded directional light to the surface.

OK, this was a first look at the `phobos_f` model. Let's examine the `phobos` model.

A texture map called `phobos_fx.tga` should be blended with the `GL_ONE` and `GL_ZERO` parameters, which are similar to the `D3DBLEND_ONE` and `D3DBLEND_ZERO` commands in Direct3D.



**Figure 10.6:** The effects texture map of the *Doom/phobos* model (courtesy of id Software)

The command

```
tcMod scale 7 7
```

resizes (enlarges or shrinks) the texture coordinates by multiplying them against the given factors. The formula scales the stage along the appropriate axis by  $1/n$ . `tcMod scale 0.5 0.5` will double your image, while `tcMod scale 2 2` will halve it. `tcMod scroll` continuously scrolls the stage by the given x and y offsets per second. The scroll speed is measured in textures per second; a texture is the dimension of the texture being modified and includes any previous shader modifications to the original .tga. A negative x value would scroll the texture to the left. A negative y value would scroll the texture down. So `tcMod scroll 0.5 -0.5` moves the texture down and to the right at the rate of a half texture each second of travel.

`tcMod rotate` rotates the stage by the given number of degrees per second. A positive number rotates in clockwise order, a negative number, in

### CAUTION

**tcMod scroll should be the last tcMod in a stage. Otherwise there may be popping or snapping visual effects in some shaders.**

counterclockwise order. So the stage is rotated around 360 degrees in a second. The command `rgbGen identity` means to leave the RGB values unchanged.

So the first stage loads an effect texture, then scales, scrolls, and rotates it.

In the next stage, the default texture map `phobos.tga` is loaded and lit by diffuse lighting. This texture map is blended with the map used in the first stage. The destination, where the texture map from the first stage resides, is multiplied with `GL_ONE_MINUS_SRC_ALPHA` and added to the source, where the texture map from this stage resides, which is multiplied by `GL_SRC_ALPHA`.

In Direct3D the first parameter corresponds to `D3DBLEND_INVSRCALPHA`, and the second, to `D3DBLEND_SRCALPHA`.

As you might notice, the shader language is a very powerful scripting language that maps directly onto the OpenGL commands used by the rendering engine of Quake III.

I hope you got the big picture on using shaders.

### NOTE

As I stated above, you might use so-called effect files in DirectX Graphics that are even more advanced than the shader files in Quake III. DirectX Graphics provides you with a complete and easy to use interface for these effect files. These effect files have the extension `.sha` (that sounds like shader file ... doesn't it ?) and are readable in a ASCII editor. There is only one example in the DirectX 8 SDK in the media directory called `water.sha`. Take a look into the DirectX 8 documentation for a deeper explanation.

## CUSTOM SOUNDS

Sounds need to be put into the `sounds/player/modelname/` directory. In Quake III the `.wav` files should be in the 22.5KHz, 16-bit, one channel—mono—format. The Quake III engine needs a particular number of sound files with particular names. These files could be of any length, but—as always—watch out for performance issues. If a model doesn't provide its custom sound files, the default sound files are the large or major sound files, depending on the sex of your model. The following sound files are needed by the Quake III engine:

- `death1.wav` death sound
- `death2.wav`
- `death3.wav`
- `drown.wav` death by drowning
- `gasp.wav` gasping for breath after being underwater
- `falling1.wav` falling from something
- `fall1.wav` impact from a fall or big jump
- `jump1.wav` jump sound
- `pain100_1.wav` pain sounds dependent on health

- pain75\_1.wav
- pain50\_1.wav
- pain25\_1.wav
- taunt.wav            sound played during a gesture

For all the models there are sound files located in the sounds/player/footsteps/directory. Depending on the footsteps parameter provided with the animation.cfg file, the proper files are chosen. There are always four files per chosen footstep sound. If you chose energy in your animation.cfg file, the files are energy1.wav to energy4.wav.

Let's move on with the internals of the .md3 file format. To get to know more about Quake III .md3 files, we have to unwrap our scalpel.

## THE .MD3 FORMAT

The .md3 file format has not been officially documented by id Software until recently. As far as I know, it has changed in a few points in the last two years, so this explanation might be subject to change. The sample program generates two text files in the directory of the model called .md3 geometry data.txt and .md3 textures.txt, which hold a lot of information on the different models. It's useful to check out the anatomy of the different models and their default textures.

An .md3 file is a binary file, so normally you wouldn't be able to take a look into it with Notepad. So how will you use it?

There are different ways to build your own .md3 files. You might take a look at one of the Web sites—such as [www.polycount.com](http://www.polycount.com)—that deal with the construction of Quake III models. There is always a list of tools that could be used for this purpose.

You'll find a lot of exporters for 3D Studio Max and other modeling software tools. If you're familiar with these tools, that's the best way to go. If you're not familiar with these tools, like me, I would suggest using MilkShape 3D. It's a shareware tool, and its only purpose is to build models for the various 3-D shooters. After download, you can register it with your credit card online for—as far as I know—\$20. I found an excellent tutorial on using it at [www.quake3mods.net/rungy/lorimar/contents.html](http://www.quake3mods.net/rungy/lorimar/contents.html).

MilkShape helps you build your .md3 file with animation. If you'd like to build a shader file, you have to write it by hand or use a tool like the Quake III Arena Shader Editor by Bert Peers at [www.larian.com/rat/q3ase.html](http://www.larian.com/rat/q3ase.html), which was provided as freeware at the time I wrote this. You'll make your skins—.tga or .jpg files—with your preferred graphic painting tool, such as Adobe Photoshop or JASC Paint Shop Pro.

To take a look at your creations, you might use the Java MD3 Model Viewer at [fragland.net/md3view/download.html](http://fragland.net/md3view/download.html). At the moment, it seems like the best program for the job. It's a little bit tricky to install, so watch out for the installation instruction.

But hey . . . we're coders . . . so back to coding.

You might divide the content of a .md3 file into the following categories:

- md3 Header
- md3 Boneframes
- md3 Tags
- md3 Meshes

Every .md3 mesh consists of

- Mesh header
- Mesh texture names
- Mesh triangles
- Mesh texture coordinates
- Mesh vertices

The order of the different pieces shows the read-in order from the .md3 file.

## MD3.H

The .md3 header holds all the info on the whole .md3 file. A structure holding this info is located in the md3.h file of the provided source package:

```
typedef struct
{
    char id[4];                      // id of file, always "IDP3"
    int iVersion;                     // version number
    char cFileName[68];
    int iBoneFrameNum;               // Number of animation key frames in the
whole model=every mesh
    int iTagNum;                      // Number of tags
    int iMeshNum;                     // Number of meshes
    int iMaxTextureNum;              // maximum number of unique textures used in a
md3 model
    int iHeaderSize;                 // size of header
    int iTagStart;                   // starting position of tag frame structures
    int iMeshStart;                  // starting position of mesh structures
} MD3HEADER;
```

I wasn't forced to use structures in the sample program, but I thought that they would lead to an example that's easier to read and understand.

The iVersion variable should hold 15 as a version number and the iHeaderSize should always be 108 bytes. The number of animation key frames of the whole model should be equal to the number of animation key frames of every mesh of this model. As far as I can see, the iMaxTextureNum is always 0, so it seems that it's not used. The starting points of the different file contents are useful to check the right read length of our .md3 file reader.

After checking the header structure, the boneframe structure has to be read from the .md3 file.

```
typedef struct
{
    FLOAT fMins[3];
    FLOAT fMaxs[3];           // extrema of the bounding box
    FLOAT pos[3];             // position of the bounding box
    float scale;               // scaling the bounding box
    char creator[16];
} MD3BONEFRAME;
```

Every boneframe structure holds a bounding box for a single animation frame. The bounding box values are used for collision detection. The name of the file from which the bounding box is created is sometimes stored in creator. There are iBoneFrameNum number of key frames in the whole model.

As you might recall from earlier, a key-framed model is saved in different positions, so it's drawn as a series of different models. In an .md3 file, the different positions of the model are saved in different tags, bounding boxes, and vertices.

Every mesh holds the same number of tags, bounding boxes, and vertices. Every stored position of the model is called a frame. A model with one mesh with 12 key frames and one tag, for example, mesh[12], consists of

- 12 bounding boxes
- 12 tags
- 12 vertex groups

Now on to the tags. There are as many tags as boneframes:

```
typedef struct
{
    char cTagName[12];
    char unknown [52];
    FLOAT pos[3];                  // pos. of tag relative to the model
    FLOAT fMatrix[3][3];            // the direction the tag is facing
} MD3TAG
```

The direction the tag is facing is stored in the 3-by-3 matrix, whereas the position is stored in three float values. I don't know the meaning of the unknown array; some people use it as storage for the cTagName too, but the length of the tag names never exceeds 12.

After the tags, which connect the parts of a model, come the meshes, the most important part of the .md3 model. Every .md3 model can consist of as many different meshes as you like. Normally the number of meshes corresponds with the need of specialized texturing for different meshes. For example, the helmet of the Doom marine consists of two meshes, the visor and the rest of the helmet. The visor can be textured with the normal visor texture map and an environment texture map and the rest of the helmet with the default texture map.

The sample program generates two ASCII files. The one that takes the geometry data is called .md3 geometry data.txt. The other one that stores information on the used textures is called .md3 textures.txt. Let's take a look at the output of the sample program for head.md3 of the Doom model:

```
--- MD3 Header ---
id: IDP3 □
iVersion: 15
Name of file:
Number of bone frames: 1
Number of tags: 1
Number of meshes: 2
Max. number of textures: 0
Header size: 108
Starting position of tag data: 164
End tag data: starting position of mesh data: 276
Size of file: 4896
--- Tags ---
--- MD3 Meshes ---
id: IDP3h_helmet
Name of mesh: h_helmet
Number of mesh frames: 1
Number of textures: 1
Number of vertices: 148
Number of triangles: 120
Starting position of triangle data: 176
Header size: 108
Starting position of texture coordiantes data: 1616
Starting position of vertex data: 2800
Size of MD3MESHFILE: 7032268
End of Mesh in md3File at: 4260
id: IDP3h_visior
Name of mesh: h_visior
```

```
Number of mesh frames: 1
Number of textures: 1
Number of vertices: 16
Number of triangles: 17
Starting position of triangle data: 176
Header size: 108
Starting position of texture coordinates data: 380
Starting position of vertex data: 508
Size of MD3MESHFILE: 7032268
End of Mesh in md3File at: 4896
--- Model statistic ---
Number of vertices: 164
Number of triangles: 137
```

I skipped the boneframe and tags section, but what you can see here are the headers of the meshes with the names of the different meshes. You can enhance this .txt file output a lot. The .md3 textures.txt file handles the used textures:

```
--- Textures ---
Number of meshes: 2
Number of textures in mesh h_helmet: 1
texture path: Models/players/doom/doom.tga
texture name: doom.tga
Number of textures in mesh h_visor: 1
texture path: Models/players/doom/doom_f.tga
texture name: doom_f.tga
```

In the first mesh—*h\_helmet*—there's only one texture with the name *doom.tga* stored. The second mesh—*h\_visor*—also holds only one texture with the name *doom\_f.tga*. A more complex output is received by taking a look at the grapple. I have converted all the .jpg files into .tga files before using it.

```
--- Textures ---
Number of meshes: 14
Number of textures in mesh w_grapple01: 1
texture path: Models/weapons2/grapple/grapple01.TGA
texture name: grapple01.TGA
Number of textures in mesh w_grapple02: 1
texture path: Models/weapons2/grapple/grapple02.TGA
texture name: grapple02.TGA
Number of textures in mesh w_grapple03: 1
texture path: Models/weapons2/grapple/grapple03.TGA
texture name: grapple03.TGA
```

Number of textures in mesh w\_grapple04: 1  
texture path: Models/weapons2/grapple/grapple04.TGA  
texture name: grapple04.TGA  
Number of textures in mesh w\_grapple05: 1  
texture path: Models/weapons2/grapple/grapple02.TGA  
texture name: grapple02.TGA  
Number of textures in mesh w\_grapple06a: 1  
texture path: Models/weapons2/grapple/grapple06.TGA  
texture name: grapple06.TGA  
Number of textures in mesh w\_grapple06b: 1  
texture path: Models/weapons2/grapple/grapple06.TGA  
texture name: grapple06.TGA  
Number of textures in mesh w\_grapple06c: 1  
texture path: Models/weapons2/grapple/grapple06.TGA  
texture name: grapple06.TGA  
Number of textures in mesh w\_grapple07: 1  
texture path: Models/weapons2/grapple/grapple07.TGA  
texture name: grapple07.TGA  
Number of textures in mesh w\_grapple08: 1  
texture path: Models/weapons2/grapple/grapple08.TGA  
texture name: grapple08.TGA  
Number of textures in mesh w\_grapple09: 1  
texture path: Models/weapons2/grapple/grapple09.TGA  
texture name: grapple09.TGA  
Number of textures in mesh w\_grapple10a: 1  
texture path: Models/weapons2/grapple/grapple10.TGA  
texture name: grapple10.TGA  
Number of textures in mesh w\_grapple10b: 1  
texture path: Models/weapons2/grapple/grapple10.TGA  
texture name: grapple10.TGA  
Number of textures in mesh w\_grapple10c: 1  
texture path: Models/weapons2/grapple/grapple10.TGA  
texture name: grapple10.TGA

This model has 14 meshes, and these meshes use 10 different textures.

Now on to the mesh headers, which are needed to load the texture names, triangles, texture coordinates, and vertices of this mesh.

```
typedef struct
{
    char cId[4];
```

```

char cName[68];
int iMeshFrameNum;           // number of frames in mesh
int iTextureNum;             // number of textures=skins in this mesh
int iVertexNum;              // number of vertices in this mesh
int iTriangleNum;            // number of triangles
int iTriangleStart;          // start of triangle data, rel. to MD3MESHFILE
int iHeaderSize;              // Headersize = starting position of texture data
int iTecVecStart;             // starting position of the texture vector data
int iVertexStart;              // starting position of the vertex data
int iMeshSize;
} MD3MESHFILE;

```

Most of the variable names are self-explanatory. The texture names start at the end of the iHeaderSize. That means normally, 108. If you add the number of textures times 68, you get the starting point of the triangle data, iTriangleStart. If you multiply iTriangleNum with the size of the TRIANGLEVERT data type and add it to iTriangleStart, you should get the starting point of the texture coordinates iTecVecStart. By multiplying the number of vertices with the size of the TEXCOORDS structure, you'll get the starting point of the vertex data, iVertexStart. Because the file will hold vertex groups for every frame, you have to multiply the number of vertices with the number of frames, multiplied with the size of the vertex data . . .

```

typedef struct
{
    SHORT vec[3];
    UCHAR unknown;
} MD3VERTEX;

```

. . . to reach the end of the mesh. The MD3VERTEX structure holds one array of shorts and an unknown UCHAR. I suppose that this is used to round up to four bytes.

The length of these structures corresponds to the length of the data, which is stored in the .md3 file. Whereas all of these structures arranges the data in the same way—stored in the .md3 files—the following MD3MESH data structure is homemade by me:

```

typedef struct
{
    char cName[68];                  // 65 chars, 32-bit aligned == 68 chars in file
    int iNumVertices;                // Mesh vertices
    int iMeshFrameNum;               // animation frames in mesh
    VMESHFRAME *vMeshFrames;        // stores vertices per animation frame in
                                    // vertices[FrameNum][VertexNum]
    int iNumTriangles;              // Mesh triangles
    TRIANGLEVERT *pTriangles;
}

```

```

    int iNumTextures;           // Mesh textures
    TEXNAMES *pTexNames;
    LPDIRECT3DTEXTURE8* pTexturesInterfaces;
    TEXCOORDS *pfTextureCoords; // Mesh tex coordinates
    TEXCOORDS *pfEnvTextureCoords;// not used here
} MD3MESH;

```

VMESHFRAME is a pointer on a D3DXVECTOR3 structure, so it's comparable to a two-dimensional array of D3DXVECTOR3s. There are five variables that hold texture information. The slot for the environment texture coordinates is not used at the moment. The whole .md3 file reader is encapsulated in a class called CMD3Model.

```

class CMD3Model
{
private:
    int                      iNumMeshes;
    MD3MESH                 *pMd3Meshes;
    MD3BONEFRAME             *md3BoneFrame;
    MD3TAG                   **md3Tags;
    FILE *LogFile;           // the log file for md3 geometry data.txt
                           // and md3 textures.txt
    LPDIRECT3DVERTEXBUFFER8 m_pVB;          // Buffer to hold vertices
    long ReadLong(FILE* File);               // file i/o
helper
    short ReadShort(FILE* File);
    FLOAT ReadFloat(FILE* File);
    void CreateVB(LPDIRECT3DDEVICE8 1pD3Ddevice); // vertex buffer system
    void DeleteVB();
    void CreateTextures(LPDIRECT3DDEVICE8 1pD3Ddevice); // texture system
    void DeleteTextures();
public:
    BOOL CreateModel( char *fname, LPDIRECT3DDEVICE8 1pD3Ddevice);
// class methods
    void DeleteModel();
    void InitDeviceObjects(LPDIRECT3DDEVICE8 1pD3Ddevice);
    void DeleteDeviceObjects();
    void Render(LPDIRECT3DDEVICE8 1pD3Ddevice);
    CMD3Model();                  // constructor/destructor
    ~CMD3Model();
};

```

The four variables at the top of the class are the most important. `pMd3Meshes` holds the data of all meshes. That means texture names, triangles, texture coordinates, and the vertices. The bounding boxes of the different meshes are stored in `md3BoneFrame`. The number of boneframes for every mesh is equivalent to the number of key frames. The tags, which might connect different models out of .md3 files, are stored in a double pointer `md3Tags`, which holds `iBoneFrameNum * iTagNum` Tag structures. There's also the pointer `LogFile` to the two log files (which are called .md3 geometry data.txt and .md3 textures.txt). The vertex buffer that will hold the geometry data ordered as triangle lists comes next. To read an .md3 file, the file i/o stuff is located in the following lines. The methods `ReadLong`, `ReadShort`, and `ReadFloat` are helper methods to make the code more readable.

Two methods create and delete the vertex buffer and two methods create and delete the texture interfaces. They are only accessible by `CMD3Model`. The public methods `CreateModel()` and `DeleteModel()` create and delete the model data, whereas the `InitDeviceObjects()` and `DeleteDeviceObjects()` methods handle device changes that might happen if the user presses F2 and chooses another device. There's also a specialized method to render the data. The constructor helps in setting the default values for the model in the .md3.cpp file.

## MD3.CPP

The constructor of the `CMD3Model` class is located at the beginning of the .md3.cpp file. It sets the default values for the soon-to-be-loaded model.

```
CMD3Model::CMD3Model()
{
    m_pVB = NULL; // vertex buffer
    iNumMeshes = 0;
}
```

## CREATEMODEL()

Our sample isn't capable of loading more than one .md3 file, so you can't connect, for example, the upper.md3, lower.md3, and head.md3 models. You might see just one part of the model. This is done to simplify the code and to make it more understandable. The method `CreateModel()` loads one .md3 file. Here are the first lines of this method:

```
BOOL CMD3Model::CreateModel( char *fname, LPDIRECT3DDEVICE8 lpD3Ddevice)
{
    MD3HEADER                      md3Header;
    MD3MESHFILE                     md3MeshFile;
    FILE *md3File;
    DWORD m_dwNumberOfVertices = 0;           // to fprintf
    the number of vertices
    DWORD m_dwNumberOfTriangles = 0;          // to fprintf the number
```

```

of triangles
    // if there are old vertex buffers and textures: delete them
    DeleteVB();
    DeleteTextures();
    DeleteModel();                                // delete old model data
    // open file
    md3File = fopen( fname, "rb" );
    if( !md3File )
        return FALSE;
    // create and open log file
    LogFile = fopen("md3 geometry data.txt","w");
    // read MD3HEADER
    fread( &md3Header, sizeof( MD3HEADER ) ,1, md3File );
    fprintf(LogFile,"-- MD3 Header -- \n");
    fprintf(LogFile,"id: %s\niVersion: %d\n",
    md3Header.iVersion);
    fprintf(LogFile,"Name of file: %s\n", md3Header.cFileName);
    fprintf(LogFile,"Number of bone frames: %d\n", md3Header.iBoneFrameNum);
    fprintf(LogFile,"Number of tags: %d\n", md3Header.iTagNum);
    fprintf(LogFile,"Number of meshes: %d\n", md3Header.iMeshNum);
    fprintf(LogFile,"Max. number of textures: %d\n",
    md3Header.iMaxTextureNum);
    fprintf(LogFile,"Header size: %d\n", md3Header.iHeaderSize);
    fprintf(LogFile,"Starting position of tag data: %d\n",
    md3Header.iTagStart);
    fprintf(LogFile,"End tag data: starting position of mesh data: %d\n",
    md3Header.iMeshStart);
    fprintf(LogFile,"Size of file: %d\n\n", md3Header.iFileSize);
    ...

```

It opens the .md3 file, opens the log file for .md3 geometry data.txt, reads in the .md3 header, and displays the content of the .md3 header in .md3file.txt. These lines of code delete the vertex buffer, textures, and model data, if existent. This is the case if a model was loaded before.

In the next lines the app checks the size of the header, size of the file, and version number so that we can be sure that it's a valid .md3 file.

```

if ((strcmp(md3Header.id,"IDP3")==0)&&
    (md3Header.iHeaderSize == 108)&&
    (md3Header.iFileSize > md3Header.iTagStart)&&
    (md3Header.iFileSize > md3Header.iMeshStart)&&
    (md3Header.iVersion == 15))

```

```
{  
    //-----  
    // Reads in the following order  
    //      1. read bone frames  
    //      2. read tags  
    //      3. read meshes  
    //-----  
    // read MD3BONEFRAME  
    md3BoneFrame = new MD3BONEFRAME[md3Header.iBoneFrameNum];  
    for(i = 0; i < md3Header.iBoneFrameNum; i++)  
    {  
        fread( &md3BoneFrame[i], sizeof( MD3BONEFRAME ), 1, md3File  
    );  
    }  
    delete[] md3BoneFrame;           // we won't use it here  
    fprintf(LogFile,"-- Tags -- \n");  
    // if there are tags: get them  
    md3Tags = (MD3TAG**)new MD3TAG[md3Header.iBoneFrameNum];  
    if (md3Header.iTagNum != 0)  
    {  
        for (i=0;i < md3Header.iBoneFrameNum;i++)  
        {  
            md3Tags[i] = (MD3TAG*)new  
MD3TAG[md3Header.iTagNum];  
            //  
            fprintf(LogFile,"\nBone Frame Num: %d\n", i);  
            for (j=0;j < md3Header.iTagNum; j++)  
            {  
                fread( &md3Tags[i][j], sizeof(  
MD3TAG ), 1, md3File );  
                //  
                fprintf(LogFile,"Tag name:  
%s\n",  
                //  
d3Tags[i][j].cTagName);  
            }  
        }  
        delete[] md3Tags;           // we won't use it here  
    }  
}
```

It reads in the bounding boxes and the tags. We need the same number of bounding boxes as key frames. The number of tags consists of the number of bounding boxes multiplied by the number of tags. For example, the file lower.md3 normally has one tag, but the upper.md3 normally has three tags, tag\_head,

tag\_weapon, and tag\_torso. If there are, for example, 215 boneframes or key frames, there have to be 215 times 3 tag structures. When the boneframes and tags are read in, the meshes have to be stored in the proper MD3MESHEs structure.

```
// read MD3MESHEs
iNumMeshes = md3Header.iMeshNum;
pMd3Meshes = new MD3MESH[iNumMeshes];
fprintf(LogFile,"\\n-- MD3 Meshes --\\n");
for(j = 0; j < iNumMeshes; j++)
{
    // Read MD3MESHFILE
    fread( &md3MeshFile, sizeof( MD3MESHFILE ), 1, md3File );
    // Log MD3MESHFILE
    fprintf(LogFile,"id: %s\\n", md3MeshFile.cId);
    fprintf(LogFile,"Name of mesh: %s\\n", &md3MeshFile.cName);
    fprintf(LogFile,"Number of mesh frames: %d\\n",
    md3MeshFile.iMeshFrameNum);
    fprintf(LogFile,"Number of textures: %d\\n", md3MeshFile.iTextureNum);
    fprintf(LogFile,"Number of vertices: %d\\n", md3MeshFile.iVertexNum);
    fprintf(LogFile,"Number of triangles: %d\\n", md3MeshFile.iTriangleNum);
    fprintf(LogFile,"Starting position of triangle data: %d\\n",
    md3MeshFile.iTriangleStart);
    fprintf(LogFile,"Header size: %d\\n", md3MeshFile.iHeaderSize);
    fprintf(LogFile,"Starting position of texture coordiantes data: %d\\n",
    md3MeshFile.iTecVecStart);
    fprintf(LogFile,"Starting position of vertex data: %d\\n",
    md3MeshFile.iVertexStart);
    fprintf(LogFile,"Size of MD3MESHFILE: %d\\n\\n", &md3MeshFile.iMeshSize);
    strcpy (pMd3Meshes[j].cName, md3MeshFile.cName);

    //_____
    // Loading the mesh data in the following order:
    // 1. texnames
    // 2. triangles
    // 3. texcoords
    // 4. vertices
    //_____
    pMd3Meshes[j].iNumTextures = md3MeshFile.iTextureNum;
    pMd3Meshes[j].pTexNames = new TEXNAMES[pMd3Meshes[j].iNumTextures];
    // always one mesh >= one texture
    for(i = 0; i < pMd3Meshes[j].iNumTextures; i++)
    {
```

```
/* a few graphic tools write the wrong texture path
   in the *.md3 file. Instead of
   models\players\conni
   they write
   odels\players\conni
   with a space at the beginning. fread()
ends reading
when a space shows up. The hack is reading
in the
whole stuff with fgetc() ... a little bit
slower.

This one takes me hours :-(

*/
char cTemp[68];
for (int w = 0; w < 68; w++)
{
    cTemp[w] = fgetc(md3File);
}
cTemp[0] = 'M';

// hack !!!!!!
strcpy (pMd3Meshes[j].pTexNames[i], cTemp);

}

// triangles
pMd3Meshes[j].pTriangles= new TRIANGLEVERT[md3MeshFile.iTriangleNum];
for(i = 0; i < md3MeshFile.iTriangleNum; i++)
{
    // Another hack:
    // some tools do not produce the right triangle data here.
    // This is normally the case at the end of the
    // triangle data stream.
    // I try to filter out the grab by using a high and low
    // limit, which I have choosen by trial and error.
    // BTW.: Loading the corrupted md3 file into milkshape and
    // saving it unchanged helps a
lot :-)

    if (((x = ReadLong(md3File))< 0) || (x > 20000))
        x = 0;
    if (((y = ReadLong(md3File))< 0) || (y > 20000))
        y = 0;
    if (((z = ReadLong(md3File))< 0) || (z > 20000))
        z = 0;
```

```
pMd3Meshes[j].pTriangles[i][0]= x;
pMd3Meshes[j].pTriangles[i][1]= y;
pMd3Meshes[j].pTriangles[i][2]= z;
}
// texture coordinates
pMd3Meshes[j].pfTextureCoords = new TEXCOORDS md3MeshFile.iVertexNum];
for(i = 0; i < md3MeshFile.iVertexNum; i++)
{
    pMd3Meshes[j].pfTextureCoords[i].u = ReadFloat(md3File);
    pMd3Meshes[j].pfTextureCoords[i].v = ReadFloat(md3File);
}
// vertices ...
pMd3Meshes[j].vMeshFrames= new MESHFRAME[md3MeshFile.iMeshFrameNum];
for(i = 0; i < md3MeshFile.iMeshFrameNum; i++)
{
    pMd3Meshes[j].vMeshFrames[i] = new D3DVECTOR[md3MeshFile.iVertexNum];
    for(int w = 0; w < md3MeshFile.iVertexNum; w++)
    {
        /* To handle a file from a right handed coordinates
           system, you might exchange the y and z value.
        */
        pMd3Meshes[j].vMeshFrames[i][w].dvX = ReadShort(md3File) / 64.0f;
        pMd3Meshes[j].vMeshFrames[i][w].dvZ = ReadShort(md3File) / 64.0f;
        pMd3Meshes[j].vMeshFrames[i][w].dvY = ReadShort(md3File) / 64.0f;
        ReadShort(md3File); // unknown
    }
}
pMd3Meshes[j].iNumVertices = md3MeshFile.iVertexNum;
pMd3Meshes[j].iMeshFrameNum = md3MeshFile.iMeshFrameNum;
pMd3Meshes[j].iNumTriangles = md3MeshFile.iTriangleNum;
m_dwNumberOfVertices += pMd3Meshes[j].iNumVertices;
m_dwNumberOfTriangles += pMd3Meshes[j].iNumTriangles;
// md3 file read length
int End = ftell(md3File);
fprintf(LogFile,"End of Mesh in md3File at: %d\n\n", End);
} //for nummeshes
// close the *.md3 file
fclose(md3File);
}
else
    return FALSE;
fprintf(LogFile,"\n-- Model statistic --\n");
```

```
fprintf(LogFile,"Number of vertices: %d\n", m_dwNumberofVertices);
fprintf(LogFile,"Number of triangles: %d\n", m_dwNumberofTriangles);
// close the LogFile
fclose(LogFile);
// creates vertex buffer and textures
CreateVB(lpD3DDevice);
CreateTextures(lpD3DDevice);
return TRUE;
}
```

As you might notice, the source code is fairly straightforward. I had a hard time finding out that a few modeling tools set a space as the first character of the texture path, so the character *m* is overwritten. I don't know which tools used to do that, but I've found models on the Web with that anomaly.

Another thing I'd like to point out is that you can only load the default texture of the mesh this way. In other words, when a mesh is used with different textures, like the Doom model shown above in the shader section, the .md3 file only holds the default textures. If you'd like to load other textures than that, for example an environment texture, you should read its paths and names from the .skin file.

CreateModel() loads triangle structures. I suppose that Quake III and most of the Quake III model viewer uses triangle fans as their primitive type for rendering, so the vertices have to be ordered as triangles. There have to be one or more .md3 tools that save .md3 files with mistakes in that triangle order. I try to filter out grab by that terrible hack you can see above, which was made by trial and error. Without that filter, this .md3 viewer would crash.

Try to save such .md3 files with the help of MilkShape. This tool is great.

The Direct3D render method used here uses triangle lists because they are the most optimized primitive type on newer graphics hardware (NVIDIA GeForce, GeForce 2).

After loading the triangles with vertex indices, we have to load the vertices.

The number of vertices we have to load is the number of key frames multiplied with the number of vertices. They are stored in \*vMeshFrame, a pointer on a D3DVECTOR class.

I have switched to a right-handed coordinate system by using the proper view and projection matrix methods, which are provided by Direct3D 8.0. With a left-handed coordinate system, all models will be orientated to the right; using a right-handed coordinate system will align the models to the left.

There are three steps to use a right-handed coordinate system:

1. Flip the order of triangle vertices so that the system traverses them clockwise from the front. You have to pass the vertices in an x, z, y order instead of an x, y, z order, as I have done above.
2. Change all the methods with a suffix LH to RH.
3. Use the view matrix to scale the world space by -1 in the z-direction. To do this, flip the sign of the \_31, \_32, \_33, and \_34 members of your view matrix.

In .md3.cpp I have skipped the order of the vertices in the triangle loading routine. This rotates the model around the x-axis in counterclockwise order of 90 degrees.

I have also changed the \*LH methods to \*RH methods. So in `RestoreDeviceObjects()` in the .md3view.cpp file `D3DXMatrixPerspectiveFovRH()` is used:

```
D3DXMatrixPerspectiveFovRH( &matProj, D3DX_PI/4, fAspect, m_fObjectRadius/64.0f,  
                           m_fObjectRadius*200.0f);
```

This method builds a right-handed perspective projection matrix. The other method that has to be changed by changing the suffix is `D3DXMatrixLookAtRH()`, which is used here in `FrameMove()` in the file .md3view.cpp. To change the sign of the view matrix members, I used the following piece of code in `FrameMove()`:

```
matView._31 = -matView._31;  
matView._32 = -matView._32;  
matView._33 = -matView._33;  
matView._34 = -matView._34;  
m_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );
```

That's all to use a right-handed system. Easy, isn't it? It's a lot easier to port your existing OpenGL applications to Direct3D with that functionality in mind. It is easier to use both common 3-D APIs to get the best performance on the Windows platforms with Direct3D and to be portable with OpenGL on Linux/Mac platforms.

At the end of every mesh load process, the end point of the file pointer is written in the .md3 geometry data.txt.

Keep in mind that most of the .md3 files store more than one mesh, and that every mesh has its own textures. There could be more than one texture per mesh. The example program is able to handle more than one texture per mesh.

At the end of the `CreateModel()` method, the textures will be created by a call to `CreateTextures()`.

### **CREATETEXTURES()**

As I have mentioned above, textures have to be used on a per-mesh basis. To get the texture map path, you might read the path and name stored in the .md3 file. If you'd like to use custom textures, the texture map path and name should be provided in a .skin file. At the least, you might try to catch the texture map path and name by searching through the directory where the .md3 file resides. If there's no texture entry in the .md3 file, a .skin file is nonexistent, and a texture map is not placed in your .md3 directory; try to speak to your graphic artist :-).

```
void CMD3Model::CreateTextures(LPDIRECT3DDEVICE8 lpD3Ddevice)  
{  
    // create and open log file
```

```
LogFile = fopen("md3 textures.txt","w");
fprintf(LogFile,"-- Textures -- \n");
fprintf(LogFile,"Number of meshes: %d\n", iNumMeshes);
for(int j = 0; j < iNumMeshes; j++)
{
    pMd3Meshes[j].pTexturesInterfaces =
        new
LPDIRECT3DTEXTURE8[pMd3Meshes[j].iNumTextures];
    fprintf(LogFile,"Number of textures in mesh %s: %d\n",
            pMd3Meshes[j].cName
, pMd3Meshes[j].iNumTextures);
    // load and create pMd3Meshes->iNumTextures
    for (int i = 0; i < pMd3Meshes[j].iNumTextures; i++)
    {
        // temporary: holds the texture path, which
is stored in *.md3
        CHAR *cTempTexturePath;
        // to store the texture name f.e. plasma.tga
        // because D3DTextr_CreateTextureFromFile()
only accepts
        // textures this way
        CHAR *cTextureName;
        cTempTexturePath = pMd3Meshes[j].pTexNames[i];
        int length=strlen(cTempTexturePath);
        // hardcoded alert !!
        // models/players/ == 15 chars
        // if the texture path is shorter than 15
chars, there isn't
        // any entry in the *.md3 file at all ...
        if (length > 15)
        {
            int count = 0;
            int iTempLength = length;
            // count down until /
            while
(cTempTexturePath[iTempLength--] != '/')
                count++;
            // allocate memory for the tex-
ture name
            cTextureName = new CHAR[count];
```

```
// copy the texture name
while (count--)
{
    cTextureName[count] =
        length--;
}
// show the used texture path
fprintf(LogFile,"texture path:
fprintf(LogFile,"texture name:
if( FAILED(
D3DXCreateTextureFromFile(1pD3Ddevice,
cTextureName,
&pMd3Meshes[j].pTexturesInterfaces[i])))

{
    fprintf(LogFile,"D3DXCreateTextureFromFile: could not create
    texture: %s\n", cTextureName);
    pMd3Meshes[j].iNumTextures = 0;
    pMd3Meshes[j].pTexturesInterfaces[i] = 0;
}
/* else
and load texture as above
the directory, where the md3 file
file as tex name.
*/
else
{
    pMd3Meshes[j].iNumTextures = 0;
```

tex-

1. open skin file, read tex name

2. try to catch the texture from

resides, with the name of this

```
    pMd3Meshes[j].pTexturesInterfaces[i] = 0;
}
}
}
// close the LogFile
fclose(LogFile);
}
```

Every mesh can have more than one texture map. The texture names with their paths are stored per mesh in `pMd3Meshes[j].pTexNames[i]`. The texture interface pointers are in `pMd3Meshes[j].pTexturesInterfaces[i]`.

`D3DXCreateTextureFromFile()` only works with texture names without paths, so we have to extract the texture name from the texture path. The simple way is to count down from the end of the texture path to the first slash character, then allocate the proper memory and read in the texture name starting at the end of the path. `D3DXCreateTextureFromFile()` maps to `D3DXCreateTextureFromFileA()` or `D3DXCreateTextureFromFileW()` depending on ANSI or UNICODE strings. These are methods that simplify the load of textures; they need only a texture name and return a texture interface. They load bitmaps from .bmp, .jpg, .png, .ppm, .dds and .tga files. If you need a more flexible method, use the monster texture create method `D3DXCreateTextureFromFileEx()`.

After getting the texture interface from `D3DXCreateTextureFromFile()`, we deallocate the temporary texture name memory. In case the texture path is shorter than 15 characters, the texture name and the texture interface are set to 0.

I haven't implemented here the possibility to read in textures with a .skin file, because of simplicity and readability of the code. When all of the textures are created, the LogFile is closed. If you would like to create a bulletproof professional .md3 character animation system, you should build your own .skin file format reader and erase the texture name slot in the .md3 file.

**Caution:** The `CreateFile()` and `CreateTextures()` routines have worked with most of the .md3 files from Quake III I've tried. They also worked with most of the .md3 files available on Web sites like [www.polycount.com](http://www.polycount.com), if they are built properly. But this is not guaranteed. A new tool with a different behavior or a slight change in the file format could change that.

OK . . . now on to the `CreateVB()` method.

### CREATEVB()

The vertex buffer will be created and filled with a call to CreateVB(). This method is called when the user changes a device or a new model is loaded.

```
void CMD3Model::CreateVB(LPDIRECT3DDEVICE8 lpD3Ddevice)
{
    DWORD m_dwNumofIndices = 0;
    int iVertexNum;
    // get the indices number to provide the size of the vertex buffer
    for(int k = 0; k < iNumMeshes; k++)
        for(int i = 0; i < pMd3Meshes[k].iNumTriangles; i++)
            for(int j = 0; j < 3; j++)
                m_dwNumofVertices++;
    // create vertex buffer
    lpD3Ddevice->CreateVertexBuffer(m_dwNumofVertices* sizeof(MD3VERTEXBUFFERSTRUCT),
                                      D3DUSAGE_WRITEONLY, 0, D3DPPOOL_MANAGED,
                                      &m_pVB);

    MD3VERTEXBUFFERSTRUCT* v;
    m_dwNumofVertices= 0;
    int iAnimationFrameNum = 0;

    m_pVB->Lock( 0, 0, (BYTE**)&v, 0 );

    for(k = 0; k < iNumMeshes; k++)
    {
        for(int i = 0; i < pMd3Meshes[k].iNumTriangles; i++)
        {
            for(int j = 0; j < 3; j++)
            {
                iVertexNum = pMd3Meshes[k].pTriangles[i][j];
                v[m_dwNumofVertices].p= D3DXVECTOR3(
                    pMd3Meshes[k].vMeshFrames[iAnimationFrameNum][iVertexNum].x,
                    pMd3Meshes[k].vMeshFrames[iAnimationFrameNum][iVertexNum].y,
                    pMd3Meshes[k].vMeshFrames[iAnimationFrameNum][iVertexNum].z);
                v[m_dwNumofVertices].tu = pMd3Meshes[k].pfTextureCoords[iVertexNum].u;
                v[m_dwNumofVertices].tv = pMd3Meshes[k].pfTextureCoords[iVertexNum].v;
                m_dwNumofVertices++;
            }
        }
    }
}
```

```
    m_pVB->Unlock();  
}
```

This method counts the number of vertices and creates the vertex buffer in the correct size. It fills the vertex buffer with a vector and the two texture coordinate pairs as defined in the MD3VERTEXBUFFERSTRUCT structure.

```
struct MD3VERTEXBUFFERSTRUCT  
{  
    D3DXVECTOR3 p;  
    FLOAT        tu, tv;  
};
```

The example uses vertices that are indexed by pMd3Meshes[k].pTriangles[i][j]. It only uses the animation with the frame number 0. The variable iAnimationFrameNum holds the number of the key frame, or the different position of the model. You might store the different positions of the model in the vertex buffer by counting up that variable. This will use a lot of memory. If the model has 1,000 vertices and 200 animation frames, you have to store  $5 * 4 \text{ bytes} * 1,000 * 200 = 4,000,000 \text{ bytes}$ ; these are around 3,906KB or around 3.8MB for one model.

## RENDER()

The Render() method renders the model.

```
void CMD3Model::Render(LPDIRECT3DDEVICE8 lpD3Ddevice)  
{  
    for(int k = 0; k < iNumMeshes; k++)  
    {  
        DWORD dwStartVertex;  
        if (k != 0)  
            dwStartVertex += pMd3Meshes[k-1].iNumTriangles * 3;  
        else  
            dwStartVertex = 0;  
        if (pMd3Meshes[k].iNumTextures > 0)  
        {  
            for (int i = 0; i < pMd3Meshes[k].iNumTextures; i++)  
            {  
                lpD3Ddevice->SetTexture( 0, pMd3Meshes[k].pTexturesInterfaces[i]);  
                lpD3Ddevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );  
                lpD3Ddevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );  
                lpD3Ddevice->SetVertexShader( FVF_MD3VERTEXBUFFERSTRUCT);  
                lpD3Ddevice->SetStreamSource( 0, m_pVB, sizeof(MD3VERTEXBUFFERSTRUCT) );  
            }  
        }  
    }  
}
```

```
    1pD3Ddevice->DrawPrimitive( D3DPT_TRIANGLELIST, dwStartVertex,
pMd3Meshes[k].iNumTriangles);
    }
}
else // just in case there are no textures in the mesh
{
    1pD3Ddevice->SetTexture( 0, 0 );
    1pD3Ddevice->SetVertexShader( FVF_MD3VERTEXBUFFERSTRUCT );
    1pD3Ddevice->SetStreamSource( 0, m_pVB, sizeof(MD3VERTEXBUFFERSTRUCT) );
    1pD3Ddevice->DrawPrimitive( D3DPT_TRIANGLELIST, dwStartVertex,
pMd3Meshes[k].iNumTriangles);
}
}
```

The render method renders every mesh by indexing the proper mesh in the vertex buffer array. The starting point of the mesh in this array is provided by dwStartVertex. The end point of the mesh is pMd3Meshes[k].iNumTriangles. Every texture in a mesh is rendered via multipass rendering. If there are two textures in one mesh, this mesh is drawn two times, first with the first texture and in the second pass with the second texture.

I haven't supported shading with the models.shader file in this sample yet. To support texture blending of more than one texture at a time, you have to write a shader class that reads out the models.shader file.

The example supports only hardware transformation and not lighting.

After rendering our model a few times, the user will load another model or exit our small little .md3 viewer. If that happens, DeleteTextures() and DeleteVB() will be called.

#### DELETETEXTURES()

This method deletes all the texture interface pointers. Nothing spectacular.

```
void CMD3Model::DeleteTextures()
{
    for(int j = 0; j < iNumMeshes; j++)
    {
        // destroy textures
        for (int i = 0; i < pMd3Meshes[j].iNumTextures; i++)
        {
            SAFE_RELEASE(pMd3Meshes[j].pTexturesInterfaces[i]);
        }
    }
}
```

```
    }  
}
```

### DELETEVB()

The macro SAVE\_RELEASE releases the vertex buffer if the user changes the device or quits the application.

```
void CMD3Model::DeleteVB()  
{  
    if (m_pVB != NULL)  
        SAFE_RELEASE( m_pVB );  
}
```

### DELETEMODEL()

The C++ delete[] class deallocates all of the memory when more than one mesh exists.

```
BOOL CMD3Model::DeleteModel()  
{  
    for(int j = 0; j < iNumMeshes; j++)  
    {  
        delete[] pMd3Meshes[j].pTexNames;  
        delete[] pMd3Meshes[j].pTriangles;  
        delete[] pMd3Meshes[j].pfTextureCoords;  
        delete[] pMd3Meshes[j].vMeshFrames;  
    }  
    if (iNumMeshes)  
        delete[] pMd3Meshes;  
    return TRUE;  
}
```

Now we've moved through the whole .md3.cpp file. Let's examine the .md3view.cpp file, which calls all of these methods above.

### MD3VIEW.CPP

The file .md3view.cpp is responsible for viewing the model. It's the main file of the small sample program and is the place where the Common file framework code lies, as in all of the other sample programs in this book. I have used the default ArcBall class provided by the DirectX 8.0 SDK to rotate the model in the window.

### ONETIMESCENEINIT()

In OneTimeSceneInit() the mouse cursor is loaded. This happens in a different way for 64-bit Windows than for 32-bit Windows.

```
HRESULT CMyD3DApplication::OneTimeSceneInit()
{
    // Set cursor to indicate that user can move the object with the mouse
    #ifdef _WIN64
        SetClassLongPtr( m_hWnd, GCLP_HCURSOR, (LONG_PTR)LoadCursor( NULL, IDC_SIZEALL ) )
    );
    #else
        SetClassLong( m_hWnd, GCL_HCURSOR, (LONG)LoadCursor( NULL, IDC_SIZEALL ) );
    #endif
    return S_OK;
}
```

### INITDEVICEOBJECTS()

InitDeviceObjects() loads the model with the help of the LoadFile() method.

```
HRESULT CMyD3DApplication::InitDeviceObjects()
{
    // Initialize the font
    m_pFont->InitDeviceObjects( m_pd3dDevice );
    // Load a *.md3 file the first time around
    static BOOL bFirstInstance = TRUE;
    if( bFirstInstance )
    {
        Pause( TRUE );
        LoadFile(cmd3Model1);
        Pause( FALSE );
        bFirstInstance = FALSE;
    }
    else
    {
        cmd3Model1->InitDeviceObjects( m_pd3dDevice );
    }
    return S_OK;
}
```

CMD3Model::InitDeviceObjects() calls the CreateVB() and the CreateTexture() methods after the first model is created and the device has changed. LoadFile() calls the standard Open File dialog box with GetOpenFileName() and calls CreateModel() by providing the proper model path.

```
HRESULT CMyD3DApplication::LoadFile(CMD3Model *model)
{
    TCHAR strCurrentName[512] = “*.md3”;
    OPENFILENAME ofn = { sizeof(OPENFILENAME), m_hWnd, NULL,
        “MD3 Files (*.md3)\0*.md3\0\0”, NULL, 0, 1, strCurrentName, 512, m_strFileName, 512,
        m_strInitialDir, “Open MD3 File”, OFN_FILEMUSTEXIST, 0, 1,
        “.md3”, 0, NULL, NULL };

    // Run the OpenFileName dialog.
    if( FALSE == GetOpenFileName( &ofn ) )
        return E_FAIL;
    // Store the initial directory for next time
    strcpy (m_strInitialDir, strCurrentName );
    strstr (m_strInitialDir, m_strFileName )[0] = ‘\0’;
    if( FAILED (model->CreateModel(m_strFileName, m_pd3dDevice) ) )
    {
        MessageBox( NULL, TEXT(“Error loading specified MD3 file”), TEXT(“MD3 Viewer”), MB_OK|MB_ICONERROR );
        return E_FAIL;
    }
    // Return successful
    return S_OK;
}
```

The dialog box lets the user choose a .md3 model. It stores the directory path for our texture routines.

### RENDER()

The model is rendered with a call to

```
cmd3Model1->Render(m_pd3dDevice);
```

in the Render() method. Nothing special happens here. You might implement a wireframe view by commenting out the following statement:

```
m_pd3dDevice->SetRenderState( D3DRS_FILLMODE, D3DFILL_WIREFRAME );
```

Sometimes it’s impressive how a few polys build a model.

### DELETEDeviceOBJECTS()

DeleteDeviceObjects() will call the proper DeleteDeviceObjects() methods of the font and .md3 model classes.

```
HRESULT CMyD3DApplication::DeleteDeviceObjects()
{
    m_pFont->DeleteDeviceObjects();
    cmd3Model1->DeleteDeviceObjects();
    return S_OK;
}
```

### FINALCLEANUP()

The final cleanup happens in the similar-sounding method. It deletes the allocated memory by the font, the .md3 model class, and the .md3 model.

```
HRESULT CMyD3DApplication::FinalCleanup()
{
    SAFE_DELETE( m_pFont );
    SAFE_DELETE(cmd3Model1);
    return S_OK;
}
```

### MSGPROC()

I've used a message procedure in this sample to query for user interaction with the File menu.

```
LRESULT CMyD3DApplication::MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam,
                                    LPARAM lParam )
{
    // Pass mouse messages to the ArcBall so it can build internal matrices
    m_ArcBall.HandleMouseMessages( hWnd, uMsg, wParam, lParam );
    // Trap the context menu
    if( uMsg == WM_CONTEXTMENU )
        return 0;
    if( uMsg == WM_COMMAND )
    {
        // Handle the open file dialog
        if( LOWORD(wParam) == IDM_OPENFILE )
        {
            Pause( TRUE );
            LoadFile(cmd3Model1);
            Pause( FALSE );
        }
    }
    return CD3DApplication::MsgProc( hWnd, uMsg, wParam, lParam );
}
```

If the user selects Open File, it loads the new .md3 model and tracks the mouse inside of the `m_ArcBall.HandleMouseMessages()` method. The load of an .md3 model was shown in the `InitDeviceObject()` section earlier.

You might enhance a lot of things in this sample:

- To load more than one .md3 file, you might use a linked list, starting with the lower.md3 and working up to the head.md3 file.
- You might write routines that read out the models.shader script to shade the models with different `SetTextureStageState()` commands.
- If you read out the commands in the animation.cfg, you might write a time-based key frame animation system.
- You might optimize the triangle drawing routine by using indexed triangle lists.
- If you'd like to use bigger textures, you might use compressed textures.

## ADDITIONAL RESOURCES

The must-read is [q3arena.net/mentalvortex/md3view/](http://q3arena.net/mentalvortex/md3view/), which explains the file format. Another description of the .md3 format can be found at [www.planetquake.com/polycount/cottages/wrath/](http://www.planetquake.com/polycount/cottages/wrath/). It's more from a graphic artist's view. If you'd like to find .md3 models, look at [www.planetquake.com/polycount/](http://www.planetquake.com/polycount/), which is one of the best Web sites for model files. An overview of making Quake III models, shaders, and levels can be found at [www.claudec.com/lair\\_of\\_shaders/](http://www.claudec.com/lair_of_shaders/). Other useful Quake III resources are [www.gamers.org/dEngine/](http://www.gamers.org/dEngine/), [www.planetquake.com/aftershock/](http://www.planetquake.com/aftershock/), and [talika.fie.us.es/~titan/](http://talika.fie.us.es/~titan/). The unofficial Quake III map file format is [graphics.stanford.edu/~kekao/q3/](http://graphics.stanford.edu/~kekao/q3/).

*This page intentionally left blank*

# **CHAPTER II**

## **GAME PHYSICS**

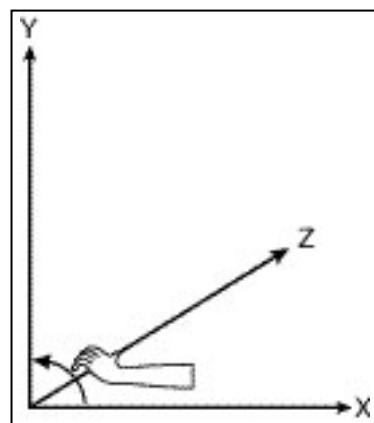
**(BY AMIR GEVR)**

**G**ames often contain a representation of a somewhat real situation. Game objects (balls, spaceships, and so on) have two main features that should be handled by the programmer. One is the object's look, which basically adds up to drawing a bunch of pixels on the screen. The second is the object's behavior and, more specifically, how it moves. Since the game is supposed to be at least somewhat real, objects that behave in what we consider a natural way are better perceived and contribute to how immersive the game is.

What does "behave naturally" mean for a game object? Simply put, it should behave according to the laws of nature, or when dealing with physical objects, according to the laws of physics.

## 3-D MATH

The physical world is a three-dimensional space and thus is represented by 3-D mathematical constructs and concepts. The coordinate system used here will be the same as the one used in Direct3D, which is the left-handed 3-D coordinate system.



**Figure 11.1:** Left-handed coordinate system

Unlike with graphics, there is not a lot of sense in defining intuitive directions for the axes for most purposes when discussing physics. The main exception to that is when modeling free fall, in which up and down directions need to be defined. In this case, the y-axis is usually selected as the up/down axis, and positive y values are usually up.

3-D space can be addressed in various ways, each usually fitting a certain calculation need. The most common representation is the Cartesian coordinate system (depicted in the drawing). Others include spherical and cylindrical coordinate systems, but we will not discuss them here, as they are not as useful as the Cartesian system for our purposes.

Any position in 3-D space (using the Cartesian coordinate system) is represented by a 3-D vector, which is an array of three real numbers, each indicating where along the appropriate axis (x, y, or z) the point is located. For example, the vector (5,1,-3) indicates a point that is 5 units along the x-axis in the positive direction, 1 unit along the y-axis in the positive direction, and 3 units along the z-axis in the negative direction.

Another basic use for the 3-D vectors is indicating directions in 3-D space. This is done by using positions that lie on the unit sphere. Imagine a sphere with a center in the origin 0,0,0 and a radius of exactly 1. Each such vector has a length of 1 since the unit sphere has a radius of 1, and for a viewer standing at position 0,0,0 the vector indicates a specific direction.

3-D vectors have the following arithmetic operations defined for them:

Subtraction, subtracting each vector element

Example:  $(7,5,9) - (1,2,3) = (6,3,6)$

Addition, adding each vector element

Example:  $(7,5,9) + (1,2,3) = (8,7,12)$

Multiplication by scalar, multiplying each element by the scalar

Example:  $2 * (1,2,3) = (2,4,6)$

Dot product, the sum of element multiplication

Example:  $(7,5,9) \cdot (1,2,3) = 7*1 + 5*2 + 9*3 = 44$

Cross product

Definition:  $(a,b,c) \times (d,e,f) = (b*f - e*c, c*d - a*f, a*e - d*b)$

Example:  $(7,5,9) \times (1,2,3) = (-3,12,9)$

Size of a vector, distance from the origin to the point (also called magnitude)

Definition:  $\text{size}(v) = \sqrt{v \cdot v}$

Example:  $\text{size}(1,2,3) = \sqrt{1*1 + 2*2 + 3*3} = \sqrt{14} = 3.741657386774$

Normalizing a vector, making it size 1

Definition:  $(1/\text{size}(v)) * v$

Some useful mathematical formulas using vectors are:

Direction from point A to point B is defined by:  $\text{Normalized}(B-A)$

Matrices are used to perform various operations on vectors. The standard matrices used in 3-D space are for rotating a vector, translating (moving) a vector, and scaling a vector. The matrices used are 4-by-4 matrices that work on (x,y,z,1) vectors. Doing the math in 4-D space allows a single matrix to perform all of these operations. Applying an operation is simply multiplying the vector by the matrix. Applying a series of matrices to a vector can also be done by first multiplying the series of matrices (resulting in one matrix) and then applying that to the vector. The order of multiplication is important if the meaning is to be preserved.

In the header file *math3d.h* you can see classes that implement a 3-D vector (*Vector3D*) and a transformation matrix (*Matrix3D*). Also defined there are operators and functions that allow performance of the necessary math.

## NEWTON'S LAWS

When doing game physics, we want to get a reasonable level of realism with the simplest possible mathematics. In most cases, such physical relations can be handled pretty well by using Newtonian physics and neglecting all other elements that do not contribute a lot to changing an object's state. A simple rigid object (an object that does not change its form) can have its state defined by its location (position and orientation) and velocity (speed) at a given time point.

These are Newton's three laws of motion:

I. Every object in a state of uniform motion tends to remain in that state of motion unless an external force is applied to it.

II. The relationship between an object's mass  $m$ , its acceleration  $a$ , and the applied force  $F$  is  $F = ma$

III. For every action there is an equal and opposite reaction.

In short, a force changes an object's velocity, and the object's velocity changes its location. In games the action of calculating the physics is not a continuous operation but a discrete one. This means that the calculation has to modify the object's state for the passage of a certain (usually small) period of time. We will denote this period  $dt$  (difference in time between frames) and use this figure in our calculations.

Some other needed notations for the object's state are

- $T$ . The object's transformation matrix, indicating its position and orientation
- $v$ . The object's velocity (a 3-D vector)
- $m$ . The object's mass

To begin, let's look at a simple example of a free-falling object. The force applied to the object is the force of gravity (intentionally neglecting air resistance and such elements). Gravity's force is a vector that has a direction of down (usually, negative  $y$ ) and a magnitude that depends on game requirements and design but is also proportional to the object's mass. We'll just use the constant  $g$ , which can be defined as any constant value that fits the game's needs. Therefore, the gravity vector is of the form  $(0, -mg, 0)$ .

The physics calculation consists of these stages:

Calculate the acceleration,  $a$ , which is the force divided by the object's mass (as described in Newton's second law)

$$a = (1/m) * (0, -m \cdot g, 0) = (0, -g, 0)$$

The velocity is changed according to the acceleration and the time difference.

$$v = v + a * dt = v + (0, -dt \cdot g, 0)$$

The position (which is the first three elements in the fourth row of the transformation matrix) is modified according to the velocity and the time difference. We'll denote it  $P$  for short.

$$P += v \cdot dt$$

Notice that if there is no force, the acceleration is zero and the velocity remains constant, as described in Newton's first law.

For example, using initial position  $(0, 1, 0)$ , initial velocity of  $0$ ,  $g=10$  and  $dt=0.1$ :

```

a=(0,-10,0)
v=(0,0,0) + 0.1 * (0,-10,0) = (0,-1,0)
p=(0,1,0) + 0.1 * (0,-1,0) = (0,0.9,0)
next frame:
a=(0,-10,0)
v=(0,-1,0) + 0.1 * (0,-10,0) = (0,-2,0)
p=(0,0.9,0) + 0.1 * (0,-2,0) = (0,0.7,0)
and so on.

```

As you can see, with a constant acceleration resulting from a constant force, you get the result of the object falling faster and faster all the time, which is what we expect from a free fall. For a more general case, the force that is applied to an object in each frame is the total of all forces that affect the object at that time. For example, if a rocket has a force driving it forward (0,0,10), and Superman is trying to hold it back (0,0,-5), then the total force is  $(0,0,10) + (0,0,-5) = (0,0,5)$ .

Momentum is a very important concept in physics. Its useful part in games physics will be described with the collision detection as it is used to determine collision response (bouncing directions and speeds). Just a note at this time that an object's momentum is simply its velocity times its mass ( $m \cdot v$ ).

Here is a simple base class of a physical object:

```

class PhysicalObject
{
public:
    void Advance(float dt)
    {
        // Calculate the current acceleration.  a=F/m
        Vector3D acceleration = (1.0f / m_Mass) * m_Thrust;
        // Modify the velocity
        m_Velocity += dt * acceleration;
        // Modify the position
        m_Position += dt * m_Velocity;
    }

protected:
    Vector3D m_Position;    // Position in 3-D space
    Vector3D m_Velocity;   // Current velocity
    Vector3D m_Thrust;     // Total applied force
    float    m_Mass;       // Mass of the object
};

```

The *Advance* function is called every time the game loop does a physics calculation. It modifies the object according to the time difference (*dt*). Using the position variable to position a mesh in 3-D space integrates the physics and the graphics parts so they work together and creates the visible physical object.

## CALCULATING THE FRAME TIME

The PhysicalObject shown above uses a parameter in its *Advance* function to indicate the duration of the frame. This parameter is originally calculated or measured by the calling application and passed to this function. An easy way to measure this duration is simply to note the system time before the frame begins and then again after it ends and subtract the two values. This piece of code appears in the code example *src/col3d* and does this calculation.

```
DWORD cur=GetTickCount();
float dt=(cur-m_Last)*0.001f;
if (m_Last==0) dt=0.02f;
m_Last=cur;
```

*GetTickCount* returns the number of milliseconds that passed since Windows was started. *m\_Last* is a value remembered between frames that stores the time point of the beginning of the frame. Obviously, the end point of one frame is the beginning point of the next. The *dt* parameter is in seconds, therefore, we need to divide by 1,000 (\*0.001).

## Game Forces

There are several force types that are very common in a lot of games. Here is a description of their natures and uses:

Name	Uses	Formula	Details
Gravity	Simulates real world gravity (falling)	(0,-mg,0)	The force is proportional to the object's mass. Real world g is 9.8 m/s <sup>2</sup> .
Air resistance	Simulates resistance of air for falling or other moving objects	-k·v	The force is opposite to the object's velocity, and k is a constant that depends on the object's shape, air density, and so on.
Static friction	Force that keeps an object from starting to move	-u·F	The force is opposite to the sum of other forces applied to the object. This force only comes into play when the object is stationary relative to the surface.  <i>u</i> is a constant ( $0 < u < 1$ ) that depends on the nature of the object and the surface (smooth or rough).
Kinetic friction	Force that slows an object down	$-\frac{ u \cdot N  \cdot v}{ v }$	The force is in a direction opposite to the object's velocity and has a magnitude proportional to the object's mass. <i>u</i> has the same meaning as with static friction, but the kinetic coefficient is smaller.

## AIR RESISTANCE

Adding air resistance can increase the level of realism considerably, especially when dealing with high speeds. This is because the magnitude of the resistance is proportional to the object's velocity. If you consider a real free-falling object, for example a sky diver, you can see that after accelerating to a high falling speed in the beginning, sky divers spend most of their falling time at the maximum speed that they have reached. This happens when the air resistance force increases to equal the force of gravity. At that point the forces cancel each other out, and the object (sky diver) remains at constant speed (Newton's first law). You'll also notice that if the sky diver changes arm positions, the streamlining changes and the maximum speed varies. This is because the constant  $k$  depends on the object's shape. When the chute opens,  $k$  becomes much bigger, causing the diver to slow down to a much lower speed.

Air resistance equations are also true for moving in fluids, but the constant's value in these cases is usually much higher, explaining the difficulty of moving in fluids.

## STATIC FRICTION

As described in the table, the static friction force opposes forces applied to the object. These forces attempt to move the object, and friction tries to keep it immobile.

In the simple case seen in this diagram in Figure 11.2, the force of gravity is cancelled out by an equal and opposite force applied by the surface called the *normal force*. The friction force is therefore simply the inverse of  $F$  multiplied by the surface static friction constant.

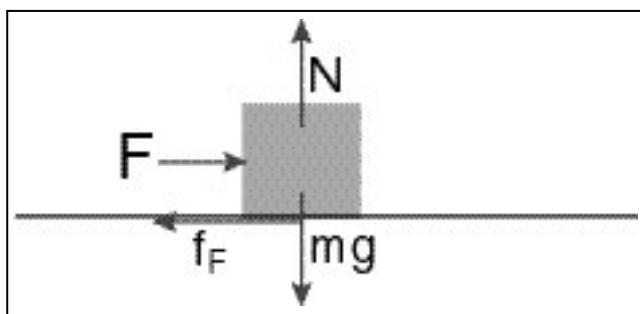
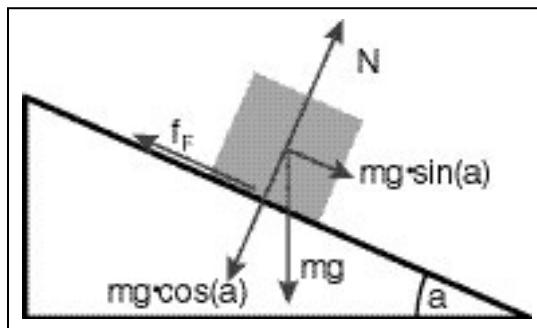


Figure 11.2: Static friction

In a more complex example, let's look at an object on a slanted surface.



**Figure 11.3:** Static friction  
on a sloped surface

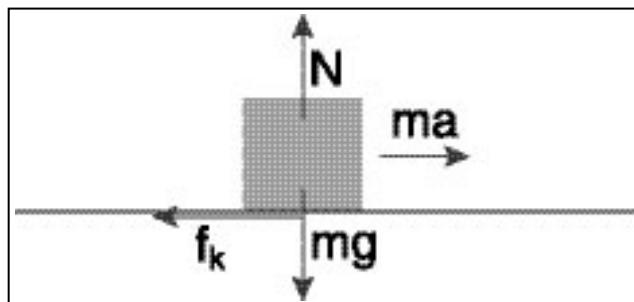
Gravity still applies to the object and still remains directly downward. The normal force, however, is perpendicular to the surface, and its magnitude is equal to the proportional part of the gravity force that is applied perpendicular to the surface:  $mg \cdot \cos(a)$ .

This leaves the other part of the gravity force,  $mg \cdot \sin(a)$ , which is parallel to the surface and fully affects the object. In this case, the static friction force would be  $u \cdot mg \cdot \sin(a)$  in the uphill direction.

## KINETIC FRICTION

Kinetic friction is the force of friction that applies to an object moving on a surface. This is a very important force in games, since it is usually used to slow down an object that is sliding on a surface. A good example would be a speedboat. As long as the engine drives the boat, the boat moves at a constant speed. As soon as the engine is off, the boat will gradually slow down and eventually stop. The force that slows the boat down is kinetic friction.

As can be seen in the diagram in Figure 11.4, the object is moving to the right with an acceleration of  $a$  (induced by the force that drives the object, the boat's engine). It also has a velocity of  $v$ . The kinetic friction force is  $u \cdot N$  in the direction opposite to the direction of movement.



**Figure 11.4:** Kinetic friction

The total force applied to the object is  $F = ma - uN$ .

Remember that this is only valid as long as the object is actually moving to the right. If the friction force stops the object, it will not push it backward, as well.

As you can see,  $N$  depends on the object's mass, which explains why heavy boats need more force to drive them through the water.

*This page intentionally left blank*

# **CHAPTER 12**

## **COLLISION DETECTION**

**(BY AMIR GEVA)**

**A**fter getting our objects to move around, it's time to consider how they interact with each other. This interaction is dependent on the type of game and game design, but you can always divide the issue into two parts: detection and response.

Collision detection is the action of determining whether or not the objects actually touch each other. Collision response is the action of deciding how to change the objects as a result of this collision. Collision response will be discussed in more detail later on.

Since we're dealing with game objects, we can separate collision detection into two different groups: 2-D objects (sprites) and 3-D objects (meshes). These are totally different cases because of the granularity in each. The 2-D case deals with pixels and is thus mathematically discrete. The 3-D case does not have this luxury and has to consider values that are not necessarily whole numbers.

As with all game programming issues, collision detection has to be as optimized speed-wise as possible. Since collision detection calculations can be very processor-intensive, you will find various optimization considerations throughout the text and code.

## THE MOST BASIC OPTIMIZATION

It is important to understand that in most games, collisions are relatively rare. Two objects (for example, a ship and a missile) that move toward each other may spend hundreds or thousands of frames with no collision happening, then one frame when the collision finally happens and both ship and missile are destroyed. For this reason, it is necessary to be able to detect that there is no collision very quickly. If the objects are far enough from each other, it can become mathematically evident that they do not collide with a relatively small amount of calculation.

## BOUNDING VOLUMES

One of the best ways to detect a no-collision case easily is to use *bounding volumes*. Since objects can be complex, it is useful to find a simpler object that contains the original object in its entirety. The simple objects (for example, a sphere or box) are called bounding volumes since they bound the original object in their volume, and they are easier to test for collisions. Since the original object is fully contained, if the bounding volumes do not collide, there is no chance that the original objects inside collide.

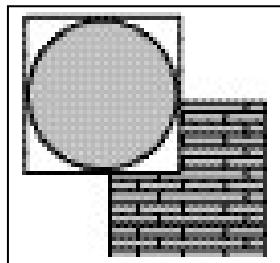
## 2-D COLLISION DETECTION

Although 3-D games are becoming more widespread, 2-D games are still popular, and a lot of beginners find it easier to start with a 2-D game. Also, most of the 2-D concepts are applicable to 3-D games and are easier to understand first in two dimensions, rather than plunging straight into 3-D.

2-D games basically involve objects that are visually a 2-D array of pixels (usually colored). These objects are usually referred to as sprites. This chapter will not discuss the technical means of maintaining and utilizing DirectX surfaces and textures, but it will focus on how to detect collision between these sprites. For this purpose, the sprites are assumed to be a simple array of pixel values. Naturally, any sprite is bound by its rectangle, which is very helpful to us, since this rectangle provides an instant bounding volume.

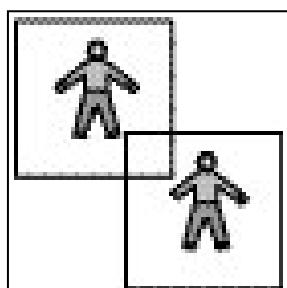
It is important to note that a sprite does not usually fill the entire rectangle, and the rest of the rectangle is simply transparent area, which is not to be considered as part of the sprite when testing for collisions. In most cases this area is defined by having a certain color value that is called the transparent color because it is not actually drawn along with the sprite. Although in most cases this color will appear outside the sprite's body, it is possible that the sprite will contain transparent parts in its interior.

In some cases, where the sprites are rectangular enough, it is possible to use the bounding volumes test as the only collision detection test (see Figure 12.1). If the results are visually acceptable, this is a preferable way to go, since it will improve the performance significantly.



**Figure 12.1:** Collisions with  
bounding rectangles

Unfortunately, a lot of games cannot take advantage of this simple solution (see Figure 12.2) and require a pixel-perfect collision detection scheme. I will discuss three methods of collision detection in 2-D: brute force, bit arrays, and sprite bounds. Each method has its advantages and disadvantages; you will have to decide for yourself which method suits your game's needs.



**Figure 12.2:** Example where  
bounding rectangles fail

## BRUTE FORCE

This method is usually the trivial solution to a lot of computing problems. In this case, it would mean looping over all the pixels in one sprite and checking to see if there is a pixel in the same position for the other sprite. Although it's a very simple solution, it suffers greatly in the performance area. Since performance is very important to us, we will not discuss it any further.

## BIT ARRAYS

We already know that the sprite is basically a 2-D array of pixels, and a pixel is basically a color value. When drawing a sprite, the pixels' color is very important, but when doing collision testing, all that is important is whether a pixel is solid or transparent. This translates to checking the pixel against the sprite's transparent color, and the result is simply one bit per pixel: 0 if the pixel is transparent, and 1 if it is not. This reduces our sprite's information considerably, which is good because processing less information usually results in improved performance.

Most compilers do not optimize bit array operations as much as we would like. This is why we will take advantage of the processor's ability to do fast bitwise operations on integers. Since the common integer is only 32 bits long, it is sometimes necessary to split sprites when they are larger than 32 pixels into groups of 32 bits and store each group in one integer. We will need to have an array to represent the pixel values (*pix*), which will be taken from the loaded bitmap or texture. The variables *width* and *height* indicate the sprite's size, *transparent* is the transparent color, and an array of *unsigned* holds the result (*bitarray*). Here is a sample class declaration that will describe the sprite:

```
class PhysicalSprite
{
public:
    // Constructor. Initializes all data structures needed
    PhysicalSprite(PIXELVALUE* pix, int w, int h, PIXELVALUE transparent);
    // Destructor. Frees allocated buffers
    virtual ~PhysicalSprite();

    // Test this sprite against another to see if they collide
    bool DetectCollision(const PhysicalSprite& other);

    void SetPosition(int X, int Y) { x=X; y=Y; }

private:
    // Calculate the bit array from the pixel array
    void CalculateBitArray(PIXELVALUE* pix, PIXELVALUE transparent);
    // Retrieve a 32 bit array starting from a certain pixel position
    unsigned GetBitArray(int x, int y) const;

protected:
    int width;
```

```
int           height;
int           x;
int           y;
unsigned*     bitarray;
int           groups_number;
};
```

The constructor will call the CalculateBitArray function, and it will create *bitarray* and fill it with the bit's data. Here is the bit calculation function:

```
void PhysicalSprite::CalculateBitArray(PIXELVALUE* pix,
                                         PIXELVALUE transparent)
{
    groups_number = width/32;
    int extra=width%32;
    if (extra != 0) groups_number++;
    bitarray=new unsigned[height*groups_number];
    // Loop to go over all the rows
    for(int i=0;i<height;i++)
    {
        int base=i*groups_number;
        int basepix = i*width;
        // Loop to go over all the 32 bit groups in one row
        for(int k=0;k<groups_number;k++)
        {
            // Initialize the integer bit array, before using it.
            bitarray[base+k]=0;
            // Loop to go over 32 pixels for one group
            for(int j=k*32;j<width && j<(k+1)*32;j++)
            {
                // Determine if the pixel is solid or transparent
                bool solid = (pix[basepix+j] != transparent);
                // Shift the bit array left by 1
                bitarray[base+k] <<= 1;
                // Set the lowest bit to this pixel's value
                bitarray[base+k] |= (solid? 1 : 0);
            }
        }
        if (extra!=0)
            bitarray[base+groups_number-1]<<=(32-extra);
    }
}
```

This has to be done only once on loading the sprite, so it doesn't have to be terribly optimized. Note that all of the divisions in the code will not slow us down too much, since any decent compiler will recognize that the denominator is a power of 2 and will optimize the division to be a shift instead.

Each sprite obviously has the width and height defined for it, but it also has a position on the 2-D world ( $x,y$ ). These positions might be limited to the screen only or use any coordinates for off-screen objects. All of these values define a rectangle that bounds the sprite. The first thing to do when doing the actual collision test is to check whether the rectangles intersect.

To determine whether the bounding rectangles collide, simply calculate the common part, also called the *intersection*, which is a rectangle as well, and then check to see if this rectangle has a positive size. If the rectangle's size is zero or negative in either axis, there is no collision. In order to calculate the intersection, we need two utility functions for minimum and maximum. Here are template implementations, in case you don't have any minimum/maximum functions:

```
template<class T>
T Min(T t1, T t2)
{
    return (t1 < t2? t1 : t2);
}

template<class T>
T Max(T t1, T t2)
{
    return (t1 > t2? t1 : t2);
}
```

Since these are generic functions, they can be used not only for *int* values, but also for any type that has the  $<$  and  $>$  operators defined for it. We will also use the Windows structure called RECT (defined in windows.h). This is its definition, and its usage is self-evident:

```
typedef struct tagRECT
{
    LONG    left;
    LONG    top;
    LONG    right;
    LONG    bottom;
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECET;
```

Before we can actually get to the collision test function, it's important to note that the sprites will not always be aligned on a 32-pixel grid. We are therefore forced to do some kind of bit shifting and joining of two bit arrays to get the correct 32 bits. For this purpose, we'll use the utility function GetBitArray. It determines whether we are aligned on a 32-pixel spot, and if not (which usually happens), it takes the

appropriate parts from two adjacent bit arrays to form one 32-bit value that will be used in the comparison. This is the implementation of this function:

```
unsigned PhysicalSprite::GetBitArray(int x, int y) const
{
    // Check vertical bounds
    if (y<0 || y>=height) return 0;

    // Prepare calculated values.
    int gx = x / 32;
    int extra = x % 32;
    int base = y * groups_number;

    // If the position is 32 bit aligned, simply return the value
    if (extra == 0) return bitarray[base+gx];

    // If the horizontal position is invalid, return 0
    if (gx < 0 || gx >= groups_number) return 0;
    // Take the bits from the value to the left of this position
    unsigned result = bitarray[base+gx] << extra;
    // If no other value is available (right edge) return what we've got
    if (gx+1 >= groups_number) return result;
    // Add the bits from the value to the right of this position
    return result | (bitarray[base+gx+1] >> (32 - extra));
}
```

Now we can get on with the actual collision test. Here is the first part of the collision test function:

```
bool PhysicalSprite::DetectCollision(const PhysicalSprite& other)
{
    // Calculate the other sprite's offset relative to this one.
    int ox = other.x - x;
    int oy = other.y - y;

    // Calculate the intersection rectangle
    // Remember that the coordinates are relative to the current sprite.
    RECT isect;
    isect.left = Max(ox,0);
    isect.right = Min(ox+other.width,width);
    isect.top = Max(oy,0);
    isect.bottom = Min(oy+other.height,height);
    // See if the intersection rectangle is valid (width and height > 0)
```

```
// If the rectangle is not valid, there is no collision
if (isect.right < isect.left || isect.bottom < isect.top) return false;
```

Notice that the obvious no-collision test is done first, because most of the tests in regular games will end at this point.

Now that we know the intersection of the two sprites, it is only necessary to check bits that lie in the intersection rectangle. Notice that the coordinates of the intersection rectangle are relative to this sprite. The variables *ox* and *oy* will be used later for offsets into the second sprite. This is the continuation of the function:

```
// Width of the rectangle
int bits_to_check = isect.right - isect.left;
// Height of the rectangle
int rows_to_check = isect.bottom - isect.top;
int groups_to_check = bits_to_check / 32;
int extra = bits_to_check % 32;
if (extra != 0) groups_to_check++;
```

These are just preliminary calculations for the loop bounds. The rest of the function with the loops that actually go through the data and compare is as follows:

```
for(int i=0;i<rows_to_check;i++)
{
    int row=isect.top+i;
    int group = groups_to_check;
    // Check in groups of 32 pixels
    for(int j=isect.left;j<=isect.right;j+=32)
    {
        unsigned bits1=GetBitArray(j,row);
        unsigned bits2=other.GetBitArray(j-ox,row-oy);
        if ((bits1 & bits2) != 0) return true;
    }
}
return false;
```

As you can see, the *GetBitArray* function really helps to simplify this part of the code. Notice the use of *ox* and *oy* as offsets for the other sprite. The actual bit test is done with the *&* operator.

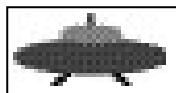
### TIP

**Possible optimizations:** Instead of calculating the bit array for the certain position, you can start from an aligned position and mask irrelevant bits.

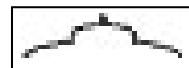
## SPRITE BOUNDS

This is a slightly different method, which takes into account a certain assumption that may be beneficial in some cases and not in others. It assumes that a sprite is a closed object and uses an imaginary line to trace the boundaries of the sprite. This approach has an advantage when dealing with larger sprites, since the information it has to store is relative to the sprite's circumference and not to its area. Another advantage is that it is simpler than the bit array approach and is easier to understand and implement.

It works by using four arrays of integers, left, right, top, and bottom. Each array contains the edge position along the sprite. For example, the sprite in Figure 12.3 has a top edge shown in Figure 12.4.



**Figure 12.3: Sprite**



**Figure 12.4: Top edge**

Or, as it is stored in the array tracing the y position of the edge:

{..., inf, inf, inf, 13, 12, 12, 11, 11, 11, 11, 10, 10, 10, 10, 10, 10, 10, ...etc.}

The *inf* implies infinity (or some large enough number), which means that the sprite doesn't have any non-transparent pixels in this column. Setting this to a large number will help later to eliminate special tests for this case.

We will now redefine the *PhysicalSprite* class described in the previous section using the sprite bounds method. Here is the class declaration:

```
class PhysicalSprite
{
public:
    // Constructor. Initializes all data structures needed
    PhysicalSprite(PIXELVALUE* pix, int pitch,
                   int w, int h, PIXELVALUE transparent);
    // Destructor. Frees allocated buffers
    virtual ~PhysicalSprite();

    // Test this sprite against another to see if they collide
    bool DetectCollision(const PhysicalSprite& other);

    // Set the sprite's position in the 2-D world
    void SetPosition(int X, int Y) { x=X; y=Y; }

private:
    // Calculates the sprite's bounds during initialization
```

```
void CalculateBounds(PIXELVALUE* pix, int pitch, PIXELVALUE transparent);
protected:
    int x;
    int y;
    int width;
    int height;
    int* top;
    int* bottom;
    int* left;
    int* right;
};
```

As you can see, the public interface part has remained pretty much the same. There is one addition of the pitch to the pixel array. This is especially relevant when using the DirectX locked rectangles or win32 bitmap bits. The private part of the class still includes the same position and size variables, but it also contains the four bounds arrays.

Unlike the interface, the implementation of the class is totally different. The constructor initializes the arrays by allocating memory from the heap:

```
PhysicalSprite::PhysicalSprite(PIXELVALUE* pix, int pitch, int w, int h,
                               PIXELVALUE transparent)
: top(new int[w]),
  bottom(new int[w]),
  left(new int[h]),
  right(new int[h]),
  width(w),
  height(h),
  x(0),
  y(0)
{
    CalculateBounds(pix,pitch,transparent);
}
```

Although the arrays' sizes may be a little confusing at first (*top* is of size width, unlike the common usage of *top* as a y value), they are correct. The function *CalculateBounds* goes over all the pixels, traces the bounds, and stores them in the four arrays:

```
void PhysicalSprite::CalculateBounds(PIXELVALUE* pix, int pitch,
                                      PIXELVALUE transparent)
{
    for(int i=0;i<width;i++)
    {
```

```
    top[i]=INFINITY;
    bottom[i]=-INFINITY;
}
for(int y=0;y<height;y++)
{
    left[y]=INFINITY;
    right[y]=-INFINITY;
    PIXELVALUE* line = &pix[y*pitch];
    for(int x=0;x<width;x++)
    {
        if (line[x]!=transparent)
        {
            left[y]=Min(left[y],x);
            right[y]=Max(right[y],x);
            top[x]=Min(top[x],y);
            bottom[x]=Max(bottom[x],y);
        }
    }
}
```

The collision test is done again in two stages. First is a test of the bounding rectangle to eliminate most of the tests in the early stage. Second is comparing the overlapping parts of each of the four bounds arrays of both sprites. The bounds are compared to their opposite side, that is, this top to the other's bottom, this right to the other's left, and so on. Since the comparisons of all the bounds are mathematically the same, a utility function called `VectorsDisjoint` is used. It tries to find a separating path. For example, take the bottom and top case. For every overlapping pixel in the bounding box, if one sprite's bottom bound is lower than the other sprite's top bound, then there cannot be a collision, because one sprite is clearly above the other.

If a separating path cannot be found, it still does not mean that there is a collision unless all four bounds report the same result. This is a good design, because as we saw earlier, no-collision cases are by far more common than collisions. Here is the `VectorsDisjoint` function:

```
static bool VectorsDisjoint(int *maxima, const int maximaCount,
                            int *minima, const int minimaCount,
                            const int offset, const int delta)
{
    int count;
    if (offset >= 0) {
        maxima += offset;
        count = Min(maximaCount - offset, minimaCount);
    }
    else {
        minima += offset;
        count = Min(minimaCount - offset, maximaCount);
    }
    for (int i = 0; i < count; i++) {
        if (maxima[i] < minima[i])
            return false;
    }
    return true;
}
```

```
    } else
    { // negative offset
        minima -= offset;
        count = Min(maximaCount, minimaCount + offset);
    }
    for (; count- > 0; maxima++, minima++)
        if ( *maxima >= (*minima + delta) )
            return false;
    return true;
}
```

In this function, `maxima` is a sprite's upper bound (bottom or right), and `minima` is a sprite's lower bound (top or left). It is therefore important to call this function with the appropriate order for the arrays. The first part of the actual collision test function does the bounding rectangle collision test:

```
bool PhysicalSprite::DetectCollision(const PhysicalSprite& other)
{
    // Calculate the offset between the sprites
    int dx = other.x - x;
    int dy = other.y - y;

    // Check bounding rectangle intersection
    if (width <= dx || height <= dy ||
        other.width <= -dx || other.height <= -dy)
        return false;
```

Again in this method, we don't consider the absolute position of the sprites when checking the collisions, but rather look at the offset of one relative to another. In this case, `dx` and `dy` hold the offset from this sprite to the other.

The rest of the function actually tests the four bounds for collision:

```
// Check the top of this sprite against the bottom of the other
if (VectorsDisjoint(other.bottom, other.width, top, width, -dx, -dy))
    return false;

// Check the bottom of this sprite against the top of the other
if (VectorsDisjoint(bottom, width, other.top, other.width, dx, dy))
    return false;

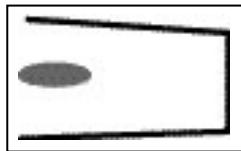
// Check the left of this sprite against the right of the other
if (VectorsDisjoint(other.right, other.height, left, height, -dy, -dx))
    return false;
```

```
// Check the right of this sprite against the left of the other
if (VectorsDisjoint(right, height, other.left, other.height, dy, dx))
    return false;

// Couldn't find any separation, so collision is reported.
return true;
}
```

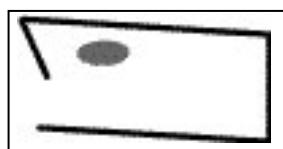
As you can see, this method's code is simpler compared with the bit array method.

This method works very well on most sprites, but it can have problems in various cases. For example, Figure 12.5 shows two sprites in close proximity, however, the bounds method accurately detects that there is no collision. Although the small sprite is inside the big sprite (at least in the top/down part), the left boundary of the big sprite is defined to be the right vertical line, since there is nothing else to the left.



**Figure 12.5:** No collision detected

In the following example (Figure 12.6), however, the algorithm fails to detect that there is no collision, because although none of the pixels collide according to all four bounds, the small sprite is inside the large sprite. If you get such a situation in your game, you probably should not use the bounds method, or modify the game design to avoid the situation.



**Figure 12.6:** Bounds method fails, collision detected

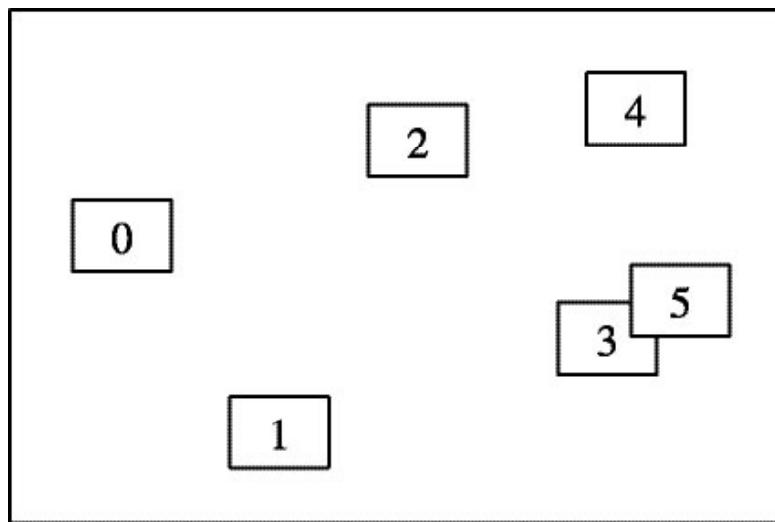
## GROUP PROCESSING

So far, we have only checked for collisions between two objects. Although very important, this is not the only consideration when designing the collision detection part of the game. A game scene can usually involve a large number of objects that have to be taken into account. The simple approach to this matter is

to use the brute force approach and check every couple of objects possible. As we have noticed before, the brute force method is not particularly performance friendly. We will denote the number of objects in the scene by  $n$ . Checking all possible couples means checking the first object against  $n-1$  objects, the second objects against  $n-2$  objects (the test against the first object is already counted for), and so on. In total we get  $n \cdot (n-1)/2$  checks. This is not too bad if  $n$  is a small number, but as  $n$  gets larger, the performance of our collision detection will degrade in proportion to  $n^2$ . The goal is to reduce the number of checks to a minimum. In order to do that, we will need to take advantage of information that we have about the objects to eliminate unnecessary checks.

#### AXIS SORT

In this type of group-processing algorithm, we take the list of objects in the scene and sort them according to their positions. We can choose an arbitrary axis and sort according to it. The algorithm then only considers adjacent objects that intersect in the selected axis. Once an adjacent object is found that does not intersect with the current object, any other objects in the list are guaranteed not to collide since they are farther away. For example, Figure 12.7 shows a group of objects, each marked by its bounding box, and the numbers represent their order in the sorted list. In this example, the sort axis selected is the x-axis.



**Figure 12.7:** Objects group to be tested

It is obvious to the eye that there is only one possible collision in this scene (remember that the boxes are not the objects themselves, but rather only the bounding boxes for the objects). To this algorithm, however, there could be three possible collisions between objects 3, 4, and 5. This means that there are a total of three full collision checks that we need to perform. Considering that using the brute force approach we would need to perform 15 checks, this is a pretty good result.

Here is the function that implements this algorithm:

```
void axis_sort(SpriteIterator begin, SpriteIterator end)
{
    // Sort the objects, according to the X axis
    std::sort(begin,end,PredX);

    // Go over all objects, according to sorted order
    for(;begin!=end;begin++)
    {
        // Go over all adjacent objects, until no intersection
        // in the X axis is found.
        for(SpriteIterator i=begin+1;i!=end;i++)
        {
            if (IntersectX(*begin,*i))
            {
                // Intersection in X means possible collision
                FullCollisionTest(*begin,*i);
            }
            else
                break;
        }
    }
}
```

The function is written in the generic STL (*Store Longword*) fashion. It takes two iterators to an array of sprites as input. It first sorts the sprites according to their x-axis value using the standard sort function but with a predicate function to do the element comparison.

```
bool PredX(const SpriteRect& s1, const SpriteRect& s2)
{
    return (s1.x < s2.x);
}
```

It then goes over all of the objects in order and, for each one, checks for an x-axis intersection against all adjacent objects until no intersection is found. All intersections found are possible collisions, and a full collision test is performed. The x-axis intersection test function is very simple, too:

```
bool IntersectX(const SpriteRect& s1, const SpriteRect& s2)
{
    return (s2.x < s1.x+s1.w);
}
```

This code uses a struct called `SpriteRect` for its sprite description, and the basic attributes contained there are the `x,y` position and the `w,h` (width and height) size.

On the whole, this algorithm is very simple to understand and implement. Its performance is especially good when the scene is sparse, or when objects are relatively far away from each other. If the scene is dense, the performance quickly deteriorates to the brute force level, plus the performance penalty of the sort.

### GRID

Unlike the axis sort method, with the grid method we will use more of the information that the sprites hold. Instead of a single-dimension sort, we will divide the sprites into a 2-D grid. The granularity of the grid is such that each cell is at least as big as the biggest sprite in the scene. This is necessary in order to make sure that sprites that collide are, at most, one cell away from each other.

The positioning of the sprites in the grid is according to their midpoint (that is,  $x + width/2$ ). This way, no more than half of the sprite's size can be outside its grid cell, which means that sprites located two cells away can never touch.

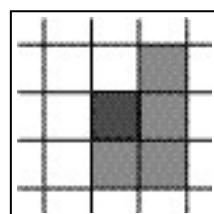
In addition to dividing the sprites into cells and checking for collisions (which will be explained shortly), we will take advantage of another piece of information we have about game sprites. Since games draw frames at a rate that lets very little time pass between two frames, the objects in the scene move very little each frame. It can be expected, in our grid system, that an object will not change grid cells every frame and, in fact, may change cells rarely (relatively speaking). Using this feature of coherence, we can maintain the sprites in the grid between frames and only do modifications when necessary, if a sprite moves to a different cell. This also saves us a lot of processing in each frame by eliminating the need to clear all the grid cells and then go over all the objects and divide them into cells again.

The collision test works by going over all of the grid cells sequentially and doing two things for each cell:

If more than one object is in this cell, do a regular brute force test on these objects.

If this cell is not empty and one of neighbors to the right or to the bottom is not empty as well, do a brute force pair matching between the two cells.

We only look at neighbor cells to the right and to the bottom simply in order to avoid double-checking. All of the neighbors to the left and top have already checked for collisions with this cell before, so doing it again is unnecessary. Figure 12.8 shows which neighbors each cell checks. The dark grey cell is the current cell, and the light grey ones are the neighbors.



**Figure 12.8: Grid checks**

Here is the actual collision test function:

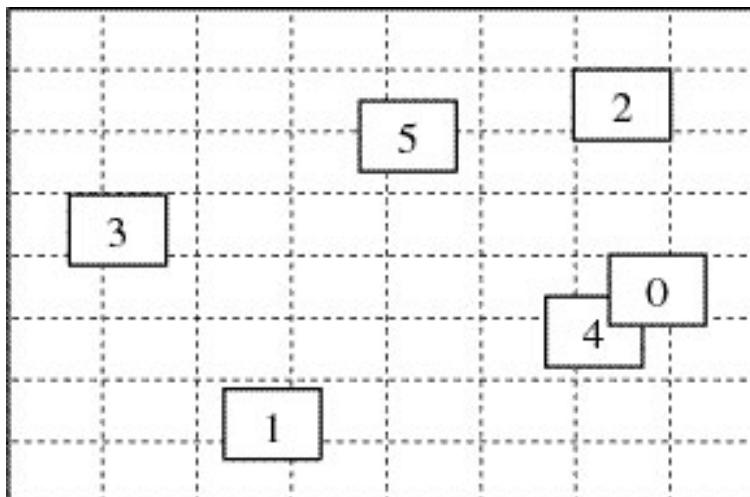
```
void CheckForCollisions()
{
    for(int y=0;y<m_CellsNumberY;y++)
    {
        for(int x=0;x<m_CellsNumberX;x++)
        {
            GridCell& cur=GetCell(x,y);
            // Check if there is any sprite in this cell.
            if (!cur.IsPopulated()) continue;
            // If there is more than one, check all sprites in this cell
            if (cur.MultipleSprites()) cur.CheckSelf();
            // Right and bottom edge of grid have no appropriate neighbors
            if (x<m_CellsNumberX-1 && y<m_CellsNumberY-1)
            {
                // Check all four neighbors. Three if y=0
                if (y>0)
                    if (GetCell(x+1,y-1).IsPopulated()) cur.Check(GetCell(x+1,y-1));
                if (GetCell(x+1,y) .IsPopulated()) cur.Check(GetCell(x+1,y) );
                if (GetCell(x+1,y+1).IsPopulated()) cur.Check(GetCell(x+1,y+1));
                if (GetCell(x ,y+1).IsPopulated()) cur.Check(GetCell(x ,y+1));
            }
        }
    }
}
```

The rest of the code can be found in the *grid* source directory.

Looking at the same scene we had in the axis sort method (Figure 12.9), we can see that the only two objects in neighboring cells are 0 and 4, which in fact collide.

## STATIC OBJECTS

As explained, our main goal in group processing is to reduce the number of collision checks performed to a minimum. If we can eliminate a part of the tests by using some information that will ensure that certain collisions will never happen, we can achieve a good measure of performance improvement. One of the most basic types of such information involves movement. If we know that two objects are static, that is, they do not move at all, these objects will never collide. Most of the objects in a scene are static in a lot of games. Using this information will improve that game's group processing dramatically.



**Figure 12.9:** Objects group placed on a grid

Basically, this involves dividing the objects into two groups: static and dynamic. An example is a run-and-jump-type game. The floors that the character steps on are static (most of them, anyway), and the character that runs and jumps around is definitely dynamic. In this case, we only need to check for collisions between the character and the floor tiles. In the general algorithm, we will need to check for collisions between the dynamic objects and the static objects as well as processing all of the collisions between dynamic objects.

The algorithm looks something like this:

```
foreach dynamic_object
    check against all other objects
```

For a scene that contains  $n$  objects in total and only  $m$  of them are dynamic, the number of checks is

$$\frac{m(m-1)}{2} + m(m-m)$$

The first part is the number of checks between the dynamic objects. The second part is the number of checks between the dynamic objects and the static objects. Considering that  $m$  is much smaller than  $n$ , the number of total checks is reduced greatly. For example, if  $n = 100$  and  $m = 10$ ,

then the number of checks is reduced from  $\frac{100(100-1)}{2} = 4950$  to

$$\frac{10(10-1)}{2} + 10(100-10) = 45 + 900 = 945$$

which is about an 80 percent reduction. It doesn't end there, though. Using the static feature of objects can be in addition to using one of the previously mentioned methods for handling groups of objects. By arranging the static objects on a custom grid, the major part of the checks  $m \cdot (n-m)$  is reduced to about  $m$  at the most.

## AUTOMATIC TRANSPARENT STATIC MARKING

A nice improvement on the previous method is to allow the objects to mark themselves as static or dynamic. If the object can do so, the algorithm can divide the objects into the two groups without requiring the programmer to make special arrangements to define which object is static and which is dynamic. It makes the whole algorithm more generic and easier to use.

This relies on good design of generic game objects. For example, if we use the `PhysicalSprite` introduced previously as a base class for our game objects and add to it the physics `Advance` function (as described in Chapter 11), we can extend the `Advance` function to check the object's position in the beginning of the frame and compare it to the position after physics are applied. If the position is the same, it means that the object did not move, at least for this frame. For the time being, the object can be safely marked as a static object.

However, this does not work if you maintain group information between frames to take advantage of time/place coherence.

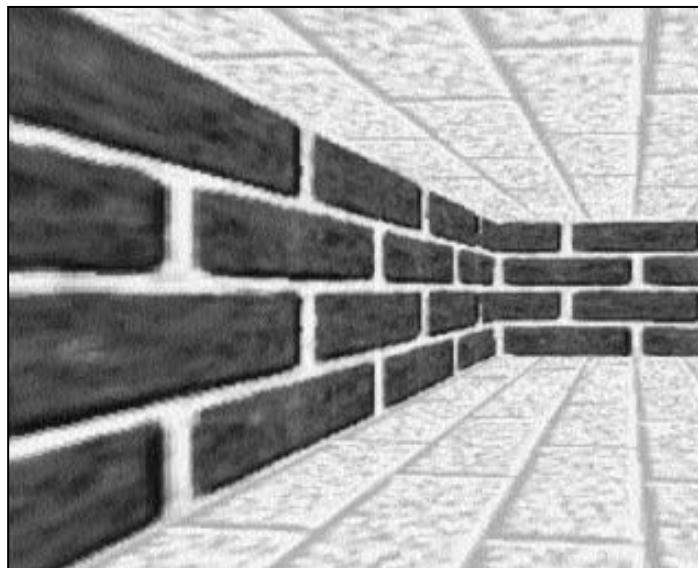
## 3-D COLLISION DETECTION

After the relative simplicity of 2-D collision detection, the question, How bad can 3-D collision detection be? comes to mind. Well, obviously it's not that bad, but it's definitely more complicated than with 2-D. As explained before, unlike the way 2-D deals with discrete values on the pixel level, the 3-D case cannot utilize the pixels of the rendered object for its calculation, simply because there are the concepts in front of and behind. Instead, 3-D collision detection works on the 3-D model's geometry, which is, in mathematical terms, continuous.

3-D models are generally described as a bunch of polygons. For graphics purposes, the polygons have material and light attributes, such as color and texture coordinates. But for collision detection purposes, it is usually enough to consider the coordinates of the polygons' vertices.

Also totally different from the 2-D case is that 3-D models not only move around in 3-D space, but also rotate around all three axes. This complicates things, because parts of the object that will be checked will have to have their vertices transformed during the test. Graphics transformations happen during rendering, and the transformed vertices are not stored anywhere usable.

Although very different, a lot of the concepts of 2-D collision detection are still applicable here. For example, 2-D tiles that cover almost their entire bounding box allow simplification of the collision detection process by limiting the tests to bounding box tests alone. In 3-D first-person games, the player's model can usually be replaced by a bounding object of some type without losing any realism in the game. The player sees the scene from the character's eyes and does not see or need to see what part of the character's body has hit the wall.



**Figure 12.10:** First-person 3-D scene

## DEALING WITH THIS COMPLEX PROBLEM

Since 3-D collision detection is far more complex and computationally intensive, it is necessary to do optimizations even more than before. In this case, however, it's not enough to optimize a general solution for the problem. We must carefully examine the specifics and understand the nature of collisions in the specific type of game and tailor the collision detection to that case.

In a first-person type of game, it is possible to utilize the data structures that hold the scenery for rendering. These data structures often arrange the data in a way that allows extracting the geometry that is near the camera to be rendered and ignoring most of the rest. This can be helpful because we only need to test for collisions with objects that are near a specific point (the position of the object being processed).

## PORTALS

In portals systems, which are a popular way for rendering 3-D scenes, the scenery is stored as a collection of convex rooms that are connected by portals. In any given frame, the current room (the room the camera is in) is rendered and the rooms visible through this room's portals are rendered, too. The advantage of this data structure is that you can get a very limited part of the scenery geometry (the current room) to check for collisions with the player.

The collision test itself can be a test between the character's model and the room's model (this will be described later). This, however, is taking the easy way out and is not especially fast. Rather a lot of computation is done in most no-collision cases because the character is always contained within the room's model and thus no early separation can be done.

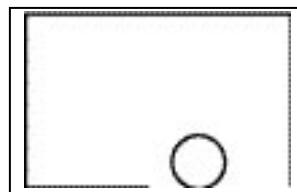
The faster and somewhat more difficult way is to organize each room in a data structure that contains information about its polygons' planes and vertices. Quick classifications can then be done regarding possible intersection of the character with any of the walls' planes.

As explained before, it is possible in this case to describe the character by its bounding volume without losing realism. A good bounding volume for this case is a cylinder. The advantage of using a cylinder is that to detect a collision between a wall and the cylinder, it is only necessary to check whether the center of the cylinder is located within a distance from the wall that is less than the cylinder's radius. Given that we have the wall's plane parameters ( $N, d$ ), it is easy to calculate the distance of a point from the plane. The distance ( $a$ ) can be simply expressed with the plane's parameters and  $P$  as the center position of the cylinder.

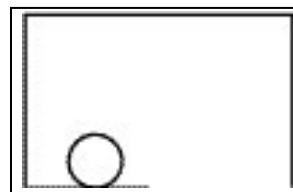
$$\text{Equation 1: } a = P \cdot N - d$$

This distance can have negative values if the position is behind the wall. It is possible to take the absolute value of this distance to avoid such problems.

Another consideration is whether the walls are infinite within the confines of the room. If the walls are finite, checking against the wall's plane is not enough. It is necessary to make sure that the cylinder's point ( $P$ ) is not somewhere farther along the wall's edge (Figures 12.11 and 12.12).

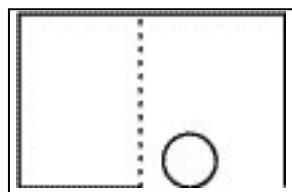


**Figure 12.11:**  
*Cylinder along the  
wall's plane*



**Figure 12.12:**  
*Cylinder along the  
wall itself*

It is possible to solve this in a different way without doing more complex calculations for each wall. If we ensure that each wall is infinite in the confines of the room by splitting problem rooms into parts that have no problems (Figure 12.13), then the problem is solved.



**Figure 12.13:**  
*Dividing the room  
into two simpler  
ones*

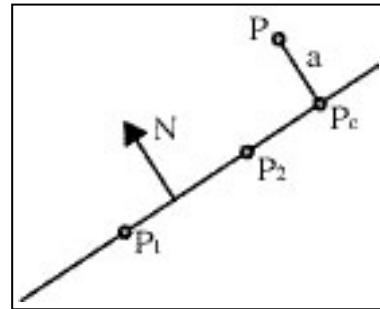
In this case, the left part is a room with three walls, and the right part is a room with two walls, with each wall covering the entire room.

This solution may not be the best choice, as splitting walls like this may slow down the rendering process by creating more polygons in the scene.

#### CALCULATING DISTANCE OF CYLINDER FROM WALL

Let's revisit the first solution to the problem, where the distance between the cylinder's position ( $P$ ) and the wall is either the simple distance described before or the distance between  $P$  and the edge of the wall. In order to find whether the cylinder is beyond the wall's edge, we will formulate a mathematical solution. We'll denote one wall edge as  $P_1$  and the other edge as  $P_2$ . The point of collision with the plane will be denoted as  $P_c$ . The point of collision can be represented parametrically according to its distance from  $P_1$ .

$$\text{Equation 2: } P_c - P_1 = t(P_2 - P_1)$$



**Figure 12.14:** Distance of cylinder from wall

As you can see, if  $t$  is 0, then the point of collision is exactly on  $P_1$ . If  $t$  is 1, then the point of collision is exactly on  $P_2$ . Any value between 0 and 1 means that the point of collision is somewhere along the actual wall, and therefore the distance calculated is correct. Values above 1 mean that the point of collision is beyond  $P_2$ , and values below 0 mean that the point of collision is beyond  $P_1$ . In that case, it is necessary to calculate the distance between the cylinder's center ( $P$ ) and the appropriate edge using simple distance between points.

In order to calculate the value of  $t$ , we will develop Equation 3 and isolate  $t$ . We will denote the length of the wall by  $L$ .

$$\text{Equation 3: } (P_c - P_1) = t(P_2 - P_1)$$

$$(P_c - P_1) \cdot (P_2 - P_1) = t(P_2 - P_1) \cdot (P_2 - P_1)$$

$$(P_c - P_1) \cdot (P_2 - P_1) = t|P_2 - P_1|^2 = tL^2$$

$$t = \frac{(P_c - P_1) \cdot (P_2 - P_1)}{L^2}$$

As you see, the calculation is not too bad, and if we store the value of  $\frac{1}{l^2}$  with the wall's information, we can really speed up this computation. The only missing piece of information is the actual point of collision. This, however, is not a problem to calculate, using the information we found regarding the distance between the cylinder's center and the wall's plane ( $a$ ). As you can see from Figure 12.14, the point of collision ( $P_c$ ) is a point located on a path from point  $P$  along the plane's normal (in reverse direction), and its distance from  $P$  is  $a$ .

$$\text{Equation 4: } P_c = P - aN$$

### BSP (BINARY SPACE PARTITIONING)

trees are data structures that describe a complex (and usually static) geometry so that it is easy to access parts of the geometry according to their relative positions.

The BSP tree is a binary tree in which each node contains a separating plane that divides the geometry into two parts.

This data structure is usually used for hidden surface removal in rendering, but it can also be used for collision detection. It allows you to retrieve the polygons that compose walls next to a certain point (the position of the player) and to perform an intersection test that is similar to the one described in the previous section (the one describing calculating the distance of a cylinder from a wall).

This method will not be described in this book, but more information can be found in the BSP Tree FAQ at [reality.sgi.com/bspfaq/](http://reality.sgi.com/bspfaq/).

### SLIDING OFF WALLS

After detecting a collision with walls, you usually want to make sure that the object is not actually colliding with the wall and position it at a minimal distance that ensures no intersection. Although the object cannot continue to move in the same direction and go into the wall, it is intuitive for the object to slide in the general direction of its movement.

Obviously, if the object is directed straight into the wall at 90 degrees, there should be no movement, and if the object is directed exactly along the wall at 0 degrees, there should be full movement. The question is what to do between these two extremes. Using simple trigonometry, it becomes evident that the amount of movement is relative to the cosine of the angle. The speed that should be modified according to this cosine is the regular object's speed, and the direction should be along the wall. How do we describe all of these in mathematical formulas?

The cosine can easily be described using a feature of the dot product of vectors. If two vectors are normalized, their dot product will give the cosine of the angle between them.

You can determine the direction of the movement by first calculating one direction along the wall using a feature of the cross product that gives a result perpendicular to both vectors. So a direction along the wall

can be  $Up \cdot N$ , where  $Up$  is the vector pointing upward (usually 0,1,0) and  $N$  is the wall's normal.

After we have this direction, we can use another feature of the dot product derived from the cosine feature. If the angle between two vectors is larger than 90 degrees, the result will be negative. This means simply that if we get a positive dot product between the direction along the wall and the object's direction, then we should use this wall direction; if the product is negative, then we simply use the opposite vector (multiply it by -1).

The formula would then be  $speed \cdot (\text{direction } N) \cdot \text{wall\_direction}$

## 3-D MESH COLLISION DETECTION

We've performed all kinds of optimizations and took into account specific game information to allow shortcuts in collision detection. But there comes a time when you cannot avoid doing actual collision detection between two objects that cannot be described by a simple mathematical representation.

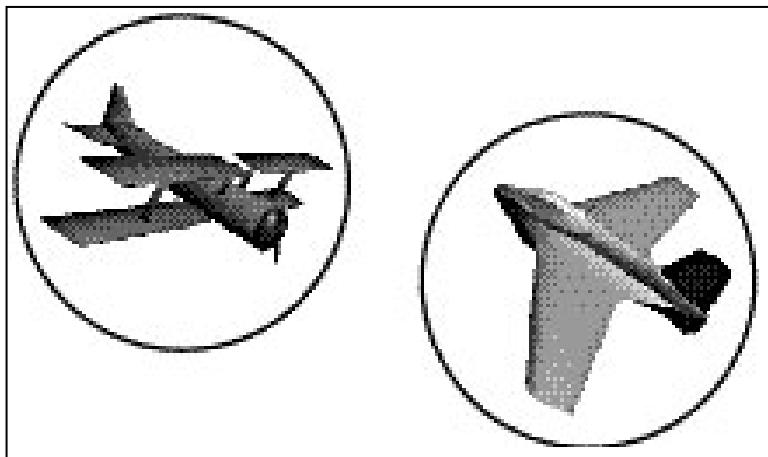
In such a case, it is necessary to check for collisions between the two objects. Each object is described as a collection of polygons, usually triangles. The brute force solution for this problem has much worse performance penalties compared with the 2-D case. Since there is no kind of coherence between the models, we have to check for intersection between every pair of triangles. If each model has  $n$  triangles, the number of triangle intersection tests needed is  $n^2$ . Since each triangle intersection test is not a trivial test, consuming a considerable amount of processing power, and  $n$  can easily rise to the level of thousands, we cannot even consider using the brute force approach. Fortunately, there are various ways of dealing with this problem more efficiently.

## BOUNDING VOLUMES

Just as in the 2-D case, it is very helpful if we can eliminate the collision test in the early stages by detecting an obvious no-collision situation. Remember that in normal scenes, the vast majority of the tests will result in no collision. Since the 3-D models rotate in addition to moving, using a box as the bounding volume is not the simplest solution because checking for intersection between oriented boxes is not trivial. Instead, we can use a bounding sphere, which is easy to calculate and use in the continuous geometry used in 3-D. The bounding sphere doesn't have to be optimal; although that would give us better performance, finding the optimal bounding sphere for a set of points is not easy. More information about this subject can be found at [www-vision.ucsd.edu/~dwhite/ball.html](http://www-vision.ucsd.edu/~dwhite/ball.html).

If we assume that the model is built around its center even approximately, then we can use a nonoptimal (but close enough) bounding sphere. To calculate this sphere, simply denote the center of the model (usually 0,0,0) as the center of the sphere, iterate over all of the model's vertices, and find the one that has a maximum distance from the center. This distance from the center is the radius of the sphere.

As you can see in Figure 12.15, we have two models, each with its bounding sphere in a scene. It is obvious to the eye that there is no collision between the actual models and even no collision between the bounding spheres. But how is this to be determined mathematically? This part is easy enough.



**Figure 12.15:** Bounding spheres

If the distance between the centers of the spheres is greater than the sum of their radii, there is no collision between the spheres. This is true because the straight line that goes from the center of one to the center of the other is composed of three segments: the radius of the first sphere, the radius of the second sphere, and the segment that connects the two spheres. Calculating the distance between the two centers is simply calculating the distance between two points.

$$\text{Equation 5: } R_1 + R_2 < \sqrt{|C_2 - C_1|^2}$$

A small optimization to this eliminates the square root and uses the squared values.

$$\text{Equation 6: } (R_1 + R_2)^2 < |C_2 - C_1|^2 = (C_2 - C_1) \cdot (C_2 - C_1)$$

This works just as well, and since the actual distance is not needed, we do not lose anything.

### CONVEXITY OF MODELS

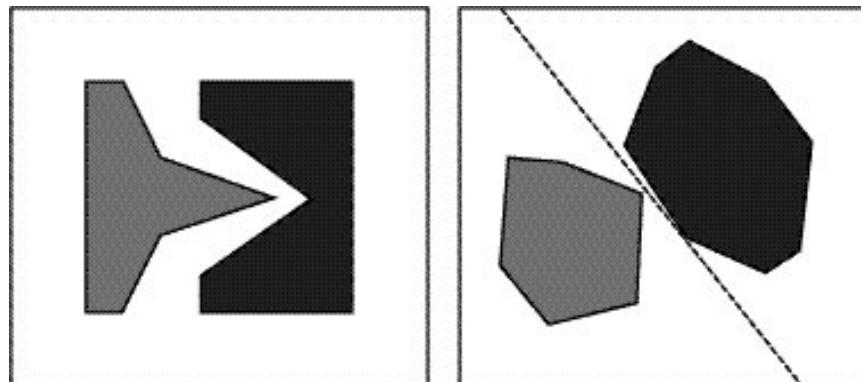
After the first test of bounding spheres has shown that a collision is possible, it is necessary to go into more details to find whether a collision has occurred. Once again, to optimize things as much as possible, we classify the 3-D objects according to their convexity. A convex model can simply be described as an object where a line between every two points on its surface is fully contained by the object.

Separating the convex from the concave (nonconvex) is important because checking for collisions between two convex objects is a faster process than between concave or generic objects. Sometimes concave objects can be separated into a set of convex objects. This is usually possible (efficiently, that is) if the designer of the object takes it into consideration and provides the information needed to perform this separation.

### CONVEX MODELS INTERSECTION

The main principle used to help detect intersections between convex objects is the fact that it is possible to find a separating plane between the objects if the objects do not intersect.

Figure 12.16 shows the idea in a simple 2-D case. The left side shows two concave objects, and although they do not intersect, there is no line that separates them. The right side, however, shows two convex objects, and no matter how you rotate or position them next to each other, as long as they do not intersect, there will always be a straight line that will separate them.



**Figure 12.16:** Concave  
vs. convex

The benefit is that we do not need to check for intersections between the triangles of the model to accurately detect collisions. If we can find that there is no separating plane, then there is certainly a collision.

A popular method for accomplishing this is using the algorithm described in this IEEE article from 1988

- “A Fast Procedure for Computing the Distance between Complex Objects in Three-Dimensional Space” by E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, *IEEE Trans. Robotics and Automation* 4(2):193-203, April 1988.

This method will not be described in this book, but here are some improvements of this algorithm, information, and implementations available on the Web:

- “Computing the Distance between Objects” by Stephen Cameron, [users.comlab.ox.ac.uk/stephen.cameron/distances.html](http://users.comlab.ox.ac.uk/stephen.cameron/distances.html)
- SOLID, Software Library for Interference Detection by Gino van den Bergen, [solid.sourceforge.net/](http://solid.sourceforge.net/)

### CONCAVE MODELS INTERSECTION

Concave models are the generic case, which assumes nothing about the model except that it is made out of polygons or triangles. This problem has various approaches to it, however, most of them involve building

some kind of hierarchy of bounding volumes for the model and using this data structure to check for intersections between two models. The method that will be described in detail is the one used in the open source software package ColDet ([photoneffect.com/coldet/](http://photoneffect.com/coldet/)). Examples of using ColDet with Direct3D 8.0 will also be given later in this chapter.

The most common hierarchy types for this purpose are AABB (*axis aligned bounding box*), OBB (*oriented bounding box*), and sphere trees.

Sphere trees take advantage of the fact that testing for intersection between two spheres in 3-D is very cheap in CPU usage, but they pay for that by usually having to be deeper in order to cover the model with satisfying accuracy. Generating the sphere tree is not a simple operation and will not be described in this book. Further reading on the subject can be found in Philip M. Hubbard's work ([www.acm.org/tog/hubbard96/](http://www.acm.org/tog/hubbard96/)) and in I. J. Palmer's work ([www.scm.brad.ac.uk/research/GIP/coldet.html](http://www.scm.brad.ac.uk/research/GIP/coldet.html)).

OBB trees use 3-D boxes that are rotated to fit tightly around the object that they enclose. Although the intersection test between two such boxes is not trivial, the relatively low depth of the tree compensates for this. This method provides excellent performance, but the process of generating an oriented bounding box tree is beyond the scope of this book. Further information on this method can be found at the university of North Carolina's Research Group on Geometry ([www.cs.unc.edu/~geom/](http://www.cs.unc.edu/~geom/)) and, more specifically, in their software, RAPID ([www.cs.unc.edu/~geom/OBB/OBBT.html](http://www.cs.unc.edu/~geom/OBB/OBBT.html)).

#### AXIS ALIGNED BOUNDING BOXES

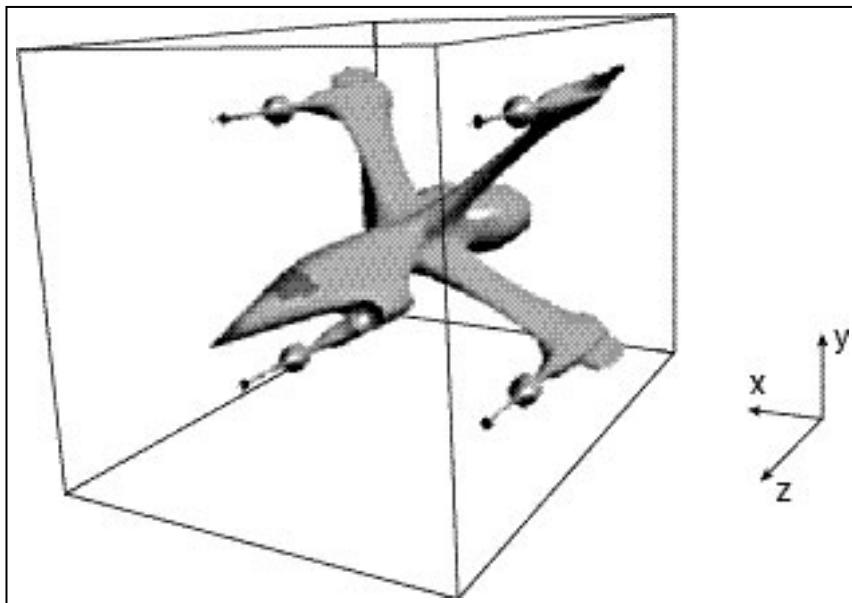
What are AABBs? Simply said, they are 3-D boxes that have faces and edges that are aligned with the coordinate system's axes. An AABB is the analogue of the bounding box that we used in the 2-D case. This type of box is used because of its simplicity. It is very easy to calculate an AABB for an object. All you need to do in order to calculate the bounding box is simply to go over all of the object's vertices, taking the minimum and maximum of each vertex's three position components (x,y,z). The minimum of all the x values, y values, and z values defines one corner of the box. The maximum of all x values, y values, and z values defines the opposite corner.

A more convenient way to represent the box is by using one of the corners or the center and storing the size of the edges in all three dimensions. This does not reduce the amount of storage needed for each box (still two 3-D vectors), but this representation will be useful in the algorithms we will use later on.

Figure 12.17 shows an AABB containing a whole object.

#### AXIS ALIGNED BOUNDING BOXES TREE

Just as in the 2-D case, checking for the intersection of the bounding box for the object does not provide enough accuracy. It is easy to see in Figure 12.17 that objects can easily penetrate the confines of the bounding box without colliding with the object. We need to perform a more accurate test in order to determine for certain whether a collision has occurred.



**Figure 12.17:** AABB for the whole object

The brute force approach has been ruled out, because of its intolerably bad performance in this case. What we need is some kind of algorithm that will allow us to divide the problem into smaller pieces and allow us to ignore the irrelevant parts.

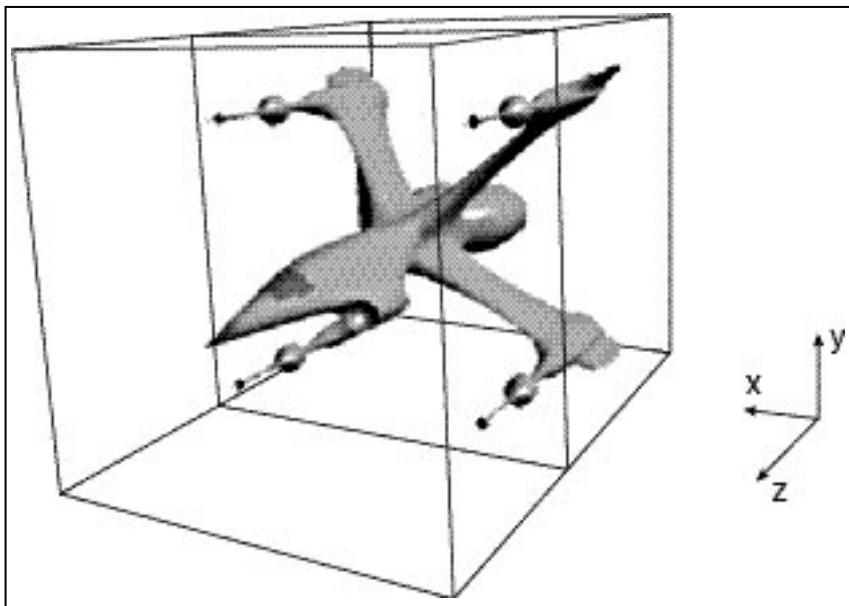
If we split the bounding box into two halves (Figure 12.18), each of the halves would be an axis aligned box by itself. What about the bounding part? What does each half bound? Simply put, each of the halves bounds the triangles that it contains.

Consider this scenario: An object is being checked for collisions with our object. A first test of bounding spheres has resulted in intersection, which means possible collision. The bounding boxes of the objects have been checked for intersection, and they, too, indicate that a collision is possible. Next, a check of the other object's bounding box against the back box (the back half) shows that there is no intersection.

This means that there is no chance that any of the polygons contained in the back box will collide with any part of the other object. Assuming that the object is somewhat balanced in its distribution of polygons, we have just *saved* about *half* of the polygons that have to be checked in the first model and, in fact, in the whole test.

Why stop there? If we continue to divide each of the halves into two parts until we reach boxes that contain, for example, one polygon each, we could eliminate most of the unnecessary tests and still produce an accurate result.

Where is the flaw in this plan? When the boxes get smaller, it is possible (and indeed likely) that the other object's box being tested against the two small halves will intersect both halves, leaving us in no better of a position than before. We will have to pay a price in an increased amount of box checks.



**Figure 12.18:** Split  
AABB

What other potential problems must we deal with?

- Objects will not necessarily be constructed evenly. Dividing a box arbitrarily may result in a deeper tree and reduced performance. The previous figures show, for example, that most of the object's polygons are in the back, and the division portrayed there is probably a bad choice.
- When dividing a box, some polygons may span the partition. Finding a partition that actually separates the box smoothly and divides the polygons into two totally separate groups may become impossible.
- Checking for intersections between boxes seems simple, but in real scenarios the objects (and therefore the boxes) will be rotated and translated. Simply checking each axis for overlap will not suffice.
- Some way has to be devised for testing the actual triangles for intersection. We will obviously eliminate as many triangles as we can before reaching this step.

#### HOW TO DIVIDE THE BOX

As we know, using a box hierarchy scheme has two issues. One is how to use the data structures (which will be discussed) and the other is how to build them. A model that is the data source for this data structure is basically a collection of triangles. We must find some means to build a position-based tree that will contain all of these triangles.

There are two main approaches to doing this: bottom-up and top-down.

The bottom-up approach, for example, is putting each triangle into its own box and then merging two boxes to create a node in the tree. Doing this recursively will eventually leave us with one box for the entire

model. Determining which boxes are best to be merged at each step is a complicated matter.

The top-down approach is, as described in previous sections, bounding the whole model in one box and then recursively dividing it until the leaf node boxes contain very little information (that is, one triangle). The question then is, How do we divide the boxes at each step?

One of the main goals when building the hierarchy tree is to keep the size or depth of the tree to a minimum. This is best achieved if every box is divided so the triangles are evenly distributed among the boxes. In order to find out where to cut the box, we take some candidates (three ways to halve a box) and run all of the triangles in the box against a classification function that the suggested dividing plane as a parameter. When we find the best candidate, the one with most evenly divided triangles, then we know where to cut the box. This is a rather simple solution and may not provide the best performance. A better solution is not to split in the middle of the box, but rather in a location that divides the triangles evenly.

The next question is, How do we classify a triangle in relation to the dividing plane? Obviously, if all of the triangle's vertices are on one side of the plane, there is no problem. The question is what to do if the triangle spans the plane. In such a situation, we simply take the center of the triangle (sum of its vertices divided by 3) and use that point for classification.

Some triangles may still stick out of their half of the box after the division. This is handled by extending the box's size so it will contain all of its triangles completely. This modification has a negative impact on the algorithm's performance, but it cannot be avoided completely. Selecting a better division plane can reduce this effect to a minimum.

All of these things are implemented in ColDet in the source files coldet\_bld and box\_bld.

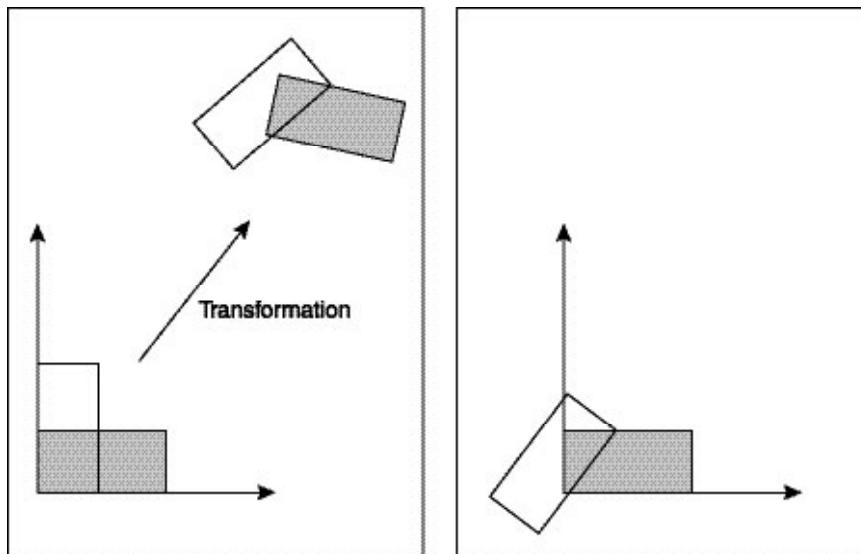
#### ORIENTED BOX INTERSECTIONS

After the box hierarchy tree is built, we get to the actual testing for collisions. So far, the code did not have to be superoptimized, because the hierarchy-building process is done once in advance and not in real game time. The collision-testing code has to be as optimized as possible.

Now we have two objects, each with its hierarchy tree, and we want to check for a collision. What do we do?

First of all, we have to remember that each of the models has some transformation (rotation and/or translation) applied to it. This means that the coordinates that appear in the hierarchy tree are no longer valid as world space coordinates. In order to use the boxes' coordinates, we will need to transform them according to the transformation matrix. Transforming the two models according to their matrices is simple and straightforward in this case, but since we are trying to optimize this algorithm as much as possible, we should look at better solutions.

Instead of transforming both models to bring them to their correct world coordinates, we can transform only one of the models. This way, it will be positioned in the same manner relative to the second model as it would if both models were transformed. This is achieved by transforming one model by its own transformation matrix and then transforming it again by the inverse of the other model's transformation matrix.



**Figure 12.19:** Relative positioning

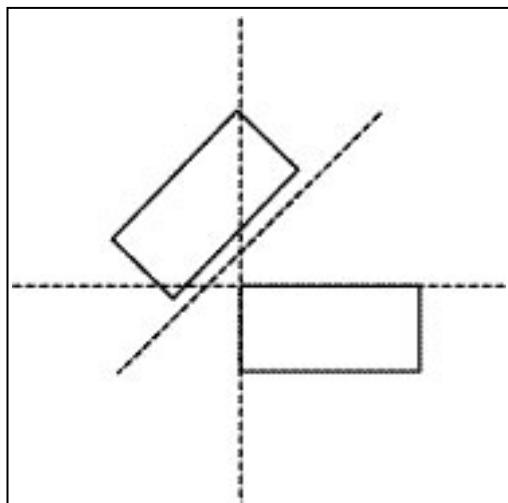
The left part of Figure 12.19 shows the two objects (represented by the blank and shaded boxes) as they are positioned in their original coordinates next to the origin. A transformation is applied, and they are relocated to their world coordinates. The right part of Figure 12.19 shows the same objects, but instead of transforming both of them, only the blank box is transformed to be in the same location relative to the shaded box. The collision results of both cases will be identical. The only difference later will be that the collision point has to be calculated with a little more effort in order to put it in world coordinates. This is a small price to pay.

We next have to decide how the trees are going to be checked against each other. After checking the first bounding box for each model against the other and seeing that there is an intersection, how should we proceed?

It would make sense to descend one level in the tree and check the children of one box against the children of the other. This, however, would cause the traversal on the tree to be breadth first, causing us to check too many unnecessary boxes. Instead, we will compare the size of the two boxes (the ones that have just intersected) and split the bigger one. We will then perform two checks of the two halves against the entire box of the second model. Since we always split the bigger box, the other model's box will eventually get split, too. This way we will shorten the path to get to the leaf nodes in the trees containing the triangles.

The checks are managed simply by using a queue. Whenever we decide that a check has to be made, it is inserted into the queue. When the algorithm loops, it will check the queue and perform the first check at the top of the queue. This queue saves us the overhead of using recursion and optimizes the code a little, too.

The heart of the algorithm is the test for intersection between two boxes. Since one of the boxes is oriented, checking axis intervals will not do. It is necessary to perform a more complex test. Instead of looking for intersection, we will use the method of elimination and look for separation. Since the boxes are both convex shapes, if they do not intersect there must be a separating plane between them. If we take all of the possible candidate planes for this separation and find one that fits (each box is fully classified on a different side), then there is no intersection. If we cannot find such a plane, then there is intersection.



**Figure 12.20:** Separating plane

The box intersection test is implemented in the ColDet source file box.cpp.

#### TRIANGLE INTERSECTION

The matter of arbitrarily oriented triangles intersection, in which each triangle is described by its three vertices, is a case very similar to the boxes case. Instead of looking for the intersection, it is more efficient to find separation. Fortunately, the *Journal of Graphics Tools* published an article in 1997 by Tomas Moller that covered this topic and provided full C source code. The code is located in the ColDet package in file tritri.c.

## **USING COLDET WITH DIRECTX 8.0**

The ColDet software package is a generic library for detecting collisions between two polygonal models. It works by building a collision model for each mesh using the mesh's triangles. Then the hierarchy tree is calculated, and the model is ready to be tested for collisions. The code example Col3D shows how to integrate ColDet into a framework application that uses CD3DMesh.

We first need to inherit from CD3DMesh and enhance it to support collisions. For this purpose, we create the class CCollidableMesh.

```
class CCollidableMesh : public CD3DMesh
```

This class will contain mainly a pointer to a `CollisionModel3D` object. This is an abstract interface to the ColDet collision model. Overriding the `Create` function in `CD3DMesh`, we add some code to the actual model loading code to initialize and create the collision model.

```
HRESULT CCollidableMesh::Create(LPDIRECT3DDEVICE8 pd3dDevice,
                                TCHAR* strFilename )
{
    HRESULT rc=CD3DMesh::Create(pd3dDevice,strFilename);
    if (SUCCEEDED(rc))
    {
        m_ColModel=newCollisionModel3D();
    }
}
```

The next thing we have to do is to retrieve the model's triangles. We clone the model to get a specific vertex format and then lock the vertex and index buffers.

```
LPD3DXMESH clone;
if (SUCCEEDED(m_pSysMemMesh->CloneMeshFVF(D3DXMESH_SYSTEMMEM,
                                              D3DFVF_XYZ, pd3dDevice, &clone)))
{
    int vnum=clone->GetNumVertices();
    int fnum=clone->GetNumFaces();
    Vertex* v;
    if (SUCCEEDED(clone->LockVertexBuffer(D3DLOCK_READONLY,(BYTE**)&v)))
    {
        WORD* ind;
        if (SUCCEEDED(clone->LockIndexBuffer(D3DLOCK_READONLY,(BYTE**)&ind)))
        {

```

The type `Vertex` used in this code is simply a struct that implements the desired vertex format. `D3DFVF_XYZ` is a format that includes only the coordinates of the vertex. This basically means that `Vertex` is a simple 3-D vector. The next step is to go over all of the indices and send the triangle information into the collision model.

```
        (float*)&v[ind[i*3+2]]);  
    }
```

The actual building of the hierarchy tree is done after all of the triangles have been added. The function that takes care of that is called *finalize*.

```
m_ColModel->finalize();
```

The rest of the code is simply releasing the locked buffers and cleaning up.

```
    clone->UnlockIndexBuffer();  
}  
clone->UnlockVertexBuffer();  
}  
clone->Release();  
}  
}  
return rc;  
}
```

The example moves and rotates two teapots on a collision course. Each frame, the collision detection is activated by calling one model's *collision* function with the other mesh's model as a parameter.

This is done in the *FrameMove* function that moves the objects. This example does not do anything to modify the objects' state in case of a collision. It simply remembers that a collision has occurred and then, in rendering time, it prints a message on the screen.

The next section will cover how to handle the collision and how objects should react.

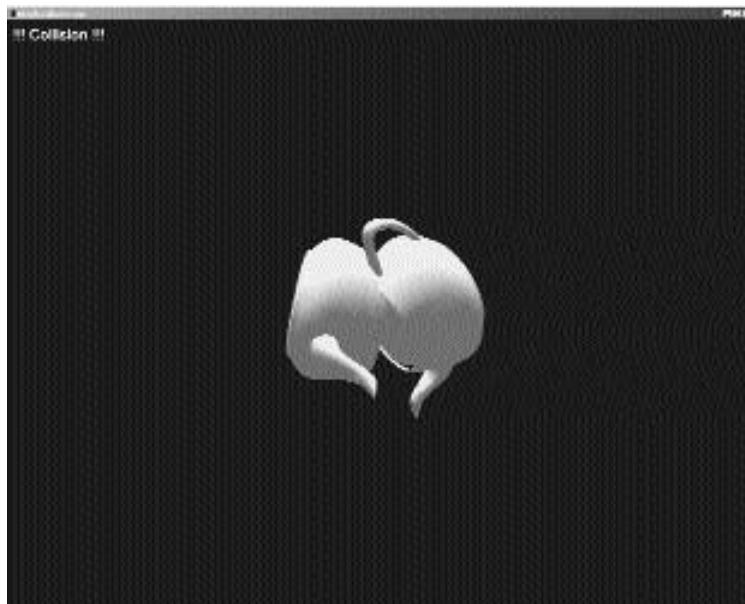
## COLLISION REACTION

As you remember from the physics section, the matter of momentum was left for later so that it could be described in the context of 3-D collisions. This is where we look at momentum.

An object's momentum is the product of its mass and its velocity ( $m \cdot v$ ). In the context of collisions, the importance of momentum is that of the case of elastic collisions, where things bounce off each other without causing dents. The sum of all momentums remains the same after the collision as it was before.

Although implementing this in an accurate and generic way may be hard to do, games are not required to be accurate, but rather believable. In this section, we will go over a simple way to handle the collision of two objects with eye-pleasing results even if they are not 100 percent mathematically accurate.

On collision, we need to determine two things for each object. One is the direction in which the object will move after the collision, and the other is the speed at which it will move. A simple way to determine the direction of movement is to say that it will move straight away from the point of collision. Using the



**Figure 12.21:** 3-D test application screen shot

formula  $object\_position - collision\_point$  gives good results, assuming the object is relatively symmetrical around its position. The speed issue is a little more complex, as it depends on the other object's speed and mass (its momentum).

Since we already determined the direction of movement, we only need to look at the momentum's magnitude. A good approximation would be that the first object will get the second object's momentum, and vice versa.

If we denote the original speed's magnitude of the objects with  $v_1$  and  $v_2$ , the new speeds will be as follows:

$$v_1' = \frac{v_2 m_2}{m_1}$$

$$v_2' = \frac{v_1 m_1}{m_2}$$

Here is a piece of code from the code example src/react that shows this calculation:

```
void CCollidableMesh::React(CCollidableMesh& other)
{
    Vector3D t1[3],t2[3];
    Vector3D p;
    m_ColModel->getCollisionPoint((float*)&p, false);
    Vector3D dir1=Vector3D(m_Transform(3,0),
                          m_Transform(3,1),
```

```
    m_Transform(3,2)) - p;
Vector3D dir2=Vector3D(other.m_Transform(3,0),
                      other.m_Transform(3,1),
                      other.m_Transform(3,2)) - p;
dir1 /= dir1.Magnitude();
dir2 /= dir2.Magnitude();

float momentum1 = m_Speed.Magnitude() * m_Mass;
float momentum2 = other.m_Speed.Magnitude() * other.m_Mass;

SetSpeed((momentum2/m_Mass)*dir1);
other.SetSpeed((momentum1/other.m_Mass)*dir2);
}
```

As you can see, the point of collision is supplied by ColDet after a collision is detected. The directions are then simply calculated by subtracting the collision point from the object's position (fourth row in the transformation matrix) and divided by the magnitude to normalize the vector.

Notice that each object's speed is set according to the other object's momentum.

The code example shows two teapots, but one of them travels twice as fast in the beginning and has four times the mass of the other. The result of the collision is that the light object gets shot out of the scene quickly, while the other object moves very slowly afterwards.

### 3-D OBJECT GROUP PROCESSING

Just as in the 2-D case, there is a common need to handle collision testing between large groups of objects. Surprisingly, the same principles that work in the 2-D case work quite well in 3-D, too. If the scene is relatively sparse, you can sort the objects according to one axis and check for collisions if the intervals of position and bounding sphere radius overlap on that axis.

It's possible also to create a 3-D grid for small dense scenes and use the same techniques as in 2-D, simply extended to three dimension arrays.

### QUIZ

Q: What's the difference between file formats used in 3-D Studio Max and in games?

A: The game file format is simpler and faster to load for the game engine.

Q: What is one of the most challenging economical considerations in game production regarding graphic information?

A: To build up a workflow that needs less time to bring the graphic information from the graphics tool into the game engine.

Q: How can you produce X files the easy way?

A: Use convx.exe or one of the converters provided in the DirectX 8.0 SDK for 3-D Studio Max and Maya 2.5 or 3.0.

Q: How does the X file format use templates?

A: They are used by the X file reader to read in specific data types. The template definition is located in rmxftmpl.x. You might define your own templates to build your own X file format.

Q: How do you create an X file, set the FVF, get the number of vertices/faces and the pointers to the vertex and index buffer with the help of the CD3DMesh class?

A:

```
m_pObject->Create(m_pd3dDevice, _T("boidy2.x"));
m_pObject->SetFVF(m_pd3dDevice, FVF_CUSTOMVERTEX);
m_dwNumVertices = m_pObject->GetSysMemMesh()->GetNumVertices();
m_dwNumFaces = m_pObject->GetSysMemMesh()->GetNumFaces();
m_pObject->GetSysMemMesh()->GetVertexBuffer( &m_pVB );
m_pObject->GetSysMemMesh()->GetIndexBuffer( &m_pIB );
```

Q: What can D3DXLoadMeshFromXof(), also used in the CD3DMesh class, do for you?

A: It reads out the whole geometry and material data and fills a LPD3DXMESH interface, which holds a vertex and index buffer.

Q: What can CloneMeshFVF() do for you?

A: It clones a mesh by using the flexible vertex format you provided.

Q: How can you extend X files?

A: By defining new templates in rmxftmpl.x.

Q: Which parts are necessary to build a Quake III model?

A: The head.md3, upper.md3, and lower.md3 files and an .md3 file with the weapon, for example, plasma.md3.

Q: Quake III models support different levels of detail. How is this done?

A: By using three different files for three different detail levels of the same model. So there are head.md3, head1.md3, and head2.md3.

Q: What is the purpose of tags in Quake III models?

A: They connect the different parts of a model.

Q: Try to get together the different file classes necessary to make a Quake III model breathe and live.

A:

- md3 files: normally head.md3, upper.md3, and lower.md3
- Icon file: for example, icon\_default.tga
- Animation file: animation.cfg
- Skin files: .tga or .jpg
- Tag file: .skin
- Shader file: for example, models.shader
- Sound files: .wav

Q: Which color depth is supported by the textures for .md3 files?

A: 32-bit.

Q: What is a shader file?

A: It's a text file that directs the texture-mapping engine of Quake III, so the polygons are rendered with the proper surface. For example, a diffuse color is blended with the color of the texture map, and so on.

Q: What is the purpose of the .txt files that are generated by the example program?

A: To show the geometry data and texture data of an .md3 file.

Q: What are the categories that you might divide the content of an .md3 file into?

A:

Header

Boneframes

Tags

Mesh:

- Header
- Texture Names
- Triangles
- Texture Coordinates
- Vertices

Q: What is key frame animation?

A: A model is stored in different positions in its model file. To animate the model, the different positions are viewed one after another.

Q: How many tags are stored in an .md3 model with one mesh with 200 key frames and two tags ?

A: 400.

Q: What's the purpose of the MD3BONEFRAME structure?

A: To hold the bounding box needed for collision detection.

Q: How do you use a right-handed coordinate system like in OpenGL?

A:

1. You have to flip the order of triangle vertices so that the system traverses them clockwise from the front. You have to pass the vertices in an x, z, y order instead of an x, y, z order.
2. Change all the methods with a suffix LH to RH.
3. Use the view matrix to scale the world space by -1 in the z-direction. To do this, flip the sign of the \_31, \_32, \_33, and \_34 members of your view matrix.

Q: Where are the texture names stored?

A: In every mesh there is a slot for the default texture names. Another place to store the texture names is in the .skin files, where custom textures can also be stored. This way you can use different textures for the same geometry.

Q: What's the drawback of storing all the key frames in one vertex buffer one after another?

A: This will take a lot of memory.

Q: What is static friction proportional to?

A: External force applied to the object.

Q: What is dynamic friction proportional to?

A: The object's mass.

Q: What is air resistance force proportional to?

A: The object's velocity.

Q: What is the first and most important optimization done in collision detection?

A: Early detection of an obvious no-collision.

Q: Why isn't the bounding box collision test enough to determine whether a collision occurred?

A: Bounding boxes for objects often have space that doesn't belong to the object.

Q: What is the main difference between 2-D and 3-D collision detection?

A: 2-D handles pixels that are discrete units, while 3-D handles continuous math.

Q: What is the advantage of using bit arrays?

A: Perfect accuracy in detection for all cases.

Q: Why do bit arrays ignore a pixel's color and only represent it by 1 bit?

A: For collisions, the color doesn't matter, only whether the object has presence in that position or not.

Q: What is the advantage of using sprite bounds?

A: Simplicity and adequate detection for most cases.

Q: Where is axis sort group processing most effective?

A: In large sparse environments. The number of collision tests is almost linear to the number of objects.

Q: Where is the grid group processing most effective?

A: In a small bounded area, potentially containing a large number of objects.

Q: Why don't we need to test against objects to our left in the grid method?

A: These objects have already tested for collisions against the current object. The current object is to their right.

Q: Why is the separation of static and dynamic objects important?

A: Static objects cannot collide since they never change position relative to another.

Q: Why can the player be represented by a cylinder in first-person games?

A: Since the player does not see which body part has hit a wall, an approximation can be used without any visual discrepancies.

Q: How do portals help in collision detection?

A: Using the current room, you can eliminate collision tests to the nearest few walls.

Q: How do BSP trees help in collision detection?

A: Similar to portals, BSP trees let us find the nearest walls to the object.

Q: What is the simplest bounding volume for 3-D ?

A: A bounding sphere.

Q: Why is collision detection between convex models easier?

A: Two convex models that do not collide always have a separating plane between them.

Q: What is common in most ways of handling concave collision detection?

A: Using bounding volume hierarchies.

Q: Why is using a hierarchy important?

A: 3-D models have a large number of polygons, and hierarchies help eliminate the majority of unnecessary tests.

Q: What is the advantage of using AABB?

A: Axis aligned bounding boxes are simple to understand and implement.

Q: How can Direct3D 8.0 CD3DMesh objects be used with ColDet?

A: Using the LockVertexBuffer() and LockIndexBuffer() functions, the triangle information can be retrieved.

Q: Why is momentum important in collision reactions?

A: The total momentum of all objects after a collision remains the same as it was before the collision.

## ADDITIONAL RESOURCES

BSP Tree FAQ

[reality.sgi.com/bspfaq/](http://reality.sgi.com/bspfaq/)

Bounding sphere calculation

[www-vision.ucsd.edu/~dwhite/ball.html](http://www-vision.ucsd.edu/~dwhite/ball.html)

Computing the Distance between Objects, by Stephen Cameron

[users.comlab.ox.ac.uk/stephen.cameron/distances.html](http://users.comlab.ox.ac.uk/stephen.cameron/distances.html)

SOLID, Software Library for Interference Detection

[solid.sourceforge.net/](http://solid.sourceforge.net/)

ColDet, 3-D Collision Detection Library

[photoneffect.com/coldet/](http://photoneffect.com/coldet/)

Sphere trees

[www.acm.org/tog/hubbard96/](http://www.acm.org/tog/hubbard96/)

[www.scm.brad.ac.uk/research/GIP/coldet.html](http://www.scm.brad.ac.uk/research/GIP/coldet.html)

UNC geometry department

[www.cs.unc.edu/~geom/](http://www.cs.unc.edu/~geom/)

Oriented bounding box trees (UNC)

[www.cs.unc.edu/~geom/OBB/OBBT.html](http://www.cs.unc.edu/~geom/OBB/OBBT.html)

I\_Collide: An interactive and exact collision-detection library for convex polyhedra

[www.cs.unc.edu/~geom/I\\_COLLIDE.html](http://www.cs.unc.edu/~geom/I_COLLIDE.html)

Q\_COLLIDE: Quick Collision Detection Library

[www.cs.hku.hk/~tlchung/collision\\_library.html](http://www.cs.hku.hk/~tlchung/collision_library.html)

QuickCD: A general purpose collision detection library

[www.ams.sunysb.edu/~jklosow/quickcd/QuickCD.html](http://www.ams.sunysb.edu/~jklosow/quickcd/QuickCD.html)

V-Clip Collision Detection Library

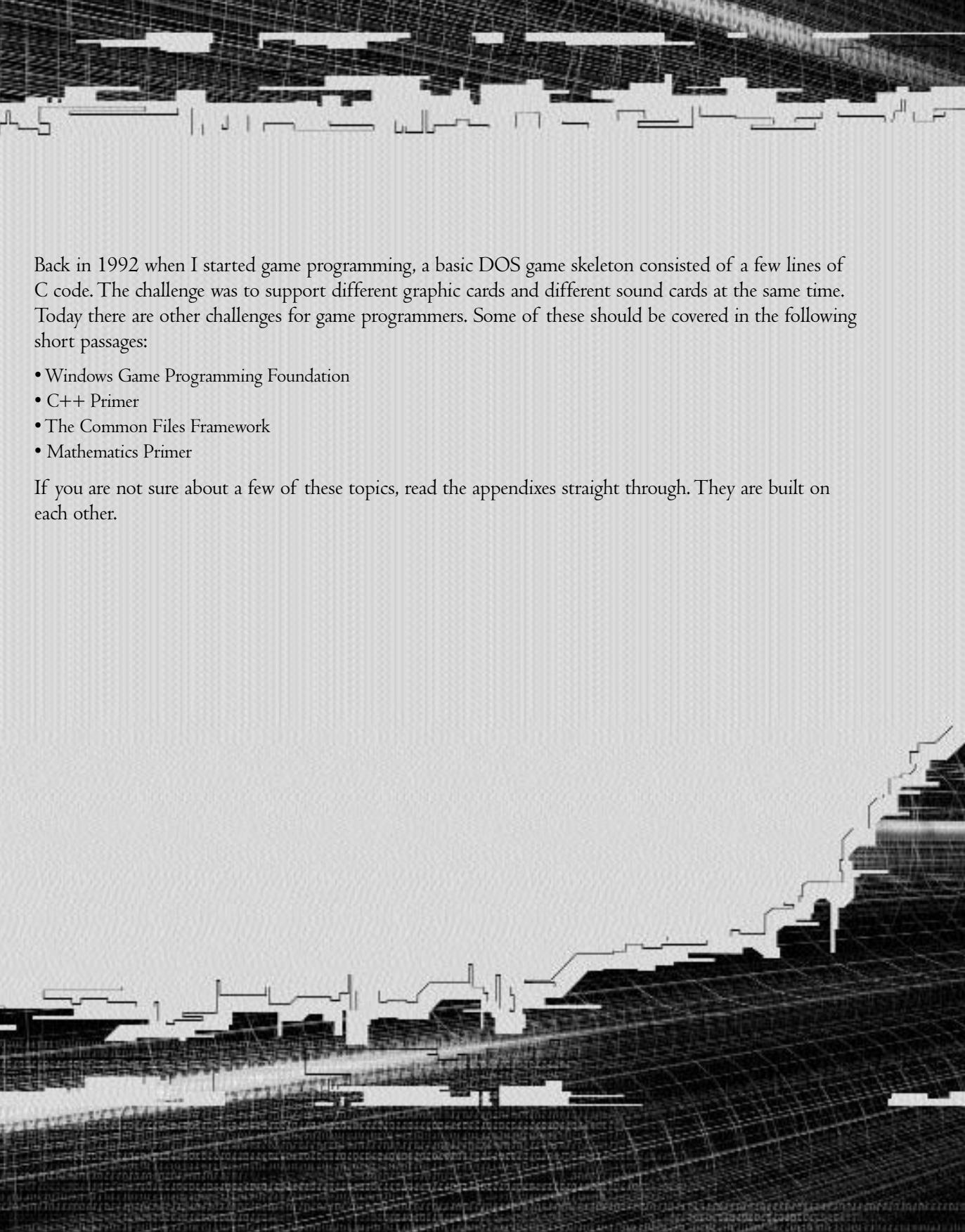
[www.merl.com/projects/vclip/](http://www.merl.com/projects/vclip/)

V\_Collide: A collision detection library for general polygon soups

[www.cs.unc.edu/~geom/V\\_COLLIDE/](http://www.cs.unc.edu/~geom/V_COLLIDE/)

# **PART IV**

# **APPENDICES**



Back in 1992 when I started game programming, a basic DOS game skeleton consisted of a few lines of C code. The challenge was to support different graphic cards and different sound cards at the same time. Today there are other challenges for game programmers. Some of these should be covered in the following short passages:

- Windows Game Programming Foundation
- C++ Primer
- The Common Files Framework
- Mathematics Primer

If you are not sure about a few of these topics, read the appendixes straight through. They are built on each other.

# **APPENDIX A**

# **WINDOWS GAME PROGRAMMING FOUNDATION**

**W**hat do game programmers have to do with windows Programming? Well . . . to be honest, not much. I think a game programmer needs mainly to know three topics on windows programming:

- How to create a window
- How to use the window message procedure
- How to use resources (icons, menus, short cuts, and so on) with the help of Visual C/C++

You will find a lot of articles and books that will help you understand this stuff. Take a look in the Additional Resources section in the end of this appendix. Now, let's start investigating some of these topics.

## HOW TO LOOK THROUGH A WINDOW

You are interested in programming a game that can be used full-screen and in a windowed mode. I will describe the programming of both modes here.

Before setting up the code for our first window that is able to present your game output, we have to learn a few general windows concepts.

## HOW WINDOWS 95/98/ME/NT/2000 INTERACTS WITH YOUR GAME

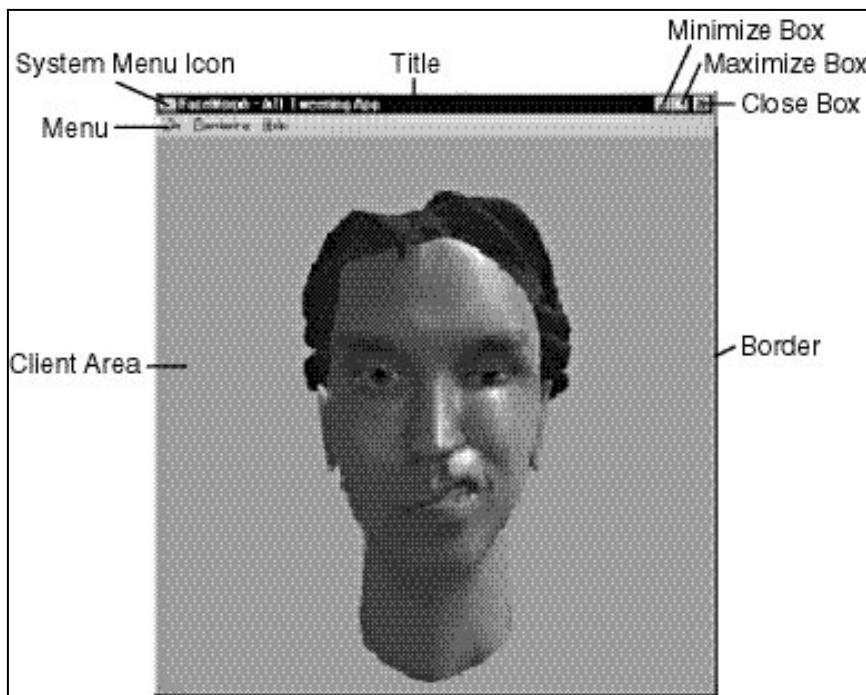
A DOS game requests such things as resources, input, and output. It doesn't have to share resources or ask someone before it takes them. However, a Windows-based game has to wait until it is sent a message by Windows. This message is passed to your program through a special function that is called by Windows. Once a message is received, your program is expected to take an appropriate action. Messages arrive randomly, so every Windows game has to check for new messages constantly.

In this way, Windows not only initiates the activity, it also grants the right to use resources (such as graphic card, hard disk, and so on). This has to be considered when you program windowed or full-screen applications. There are ways, especially for full-screen games, to get full control over the hardware from Windows. So the resources of your game are only shared in windowed mode.

Before we move on to specific aspects of Windows programming, a few important terms need to be defined.

## THE COMPONENTS OF A WINDOW

All windows have a border that defines the limits of the window and is used to resize the window. You find the system menu at the far left, and the minimize, maximize, and close boxes at the far right side of the title bar. I've made a screen shot of the FaceMorph example from ATI:



**Figure A.1:** Components of a window

Your world-class game is shown in the client area.

## A WINDOW SKELETON

Let's develop a minimal Windows application that provides the necessary features that all Windows applications have in common. Such a minimal Windows program contains two functions:

- `WinMain()`
- A window procedure such as `WndProc()`

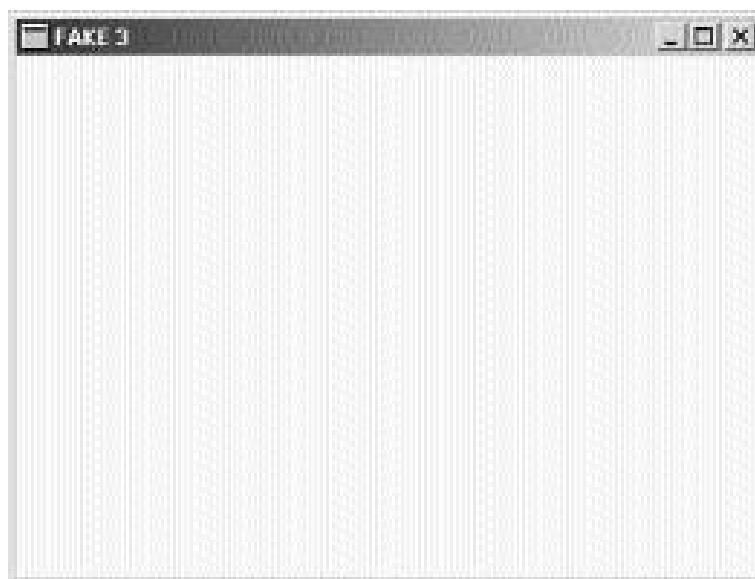
The `WinMain()` performs the following general steps:

1. Define a window class.
2. Register that class with Windows.
3. Create a window of that class.

4. Display window.
5. Begin running the message loop.

WinMain() initializes the application by defining a window class, registering that class, and creating the window. Then it displays the window and enters a message retrieval-and-dispatch loop. It terminates the message loop when it receives a WM\_QUIT message and exits your game by returning the value passed in WM\_QUIT's wParam parameter.

Here's a screen shot of our first—a little bit unattractive—example program, winskel.exe:



**Figure A.2:** Window skeleton

Here's the source:

```
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
char szWinName[] = "Crusher"; /* name of window class */
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                    LPSTR lpszArgs, int nWinMode)
{
    HWND hwnd;
    MSG msg;
    /* Step 1: Define a window class. */
    WNDCLASS wcl;
    wcl.hInstance = hThisInst; /* handle to this instance */
    wcl.lpszClassName = szWinName; /* window class name */
```

```
wcl.lpfnWndProc = WndProc; /* window function */
wcl.style = 0; /* default style */
wcl.hIcon = LoadIcon(NULL, MAKEINTRESOURCE( IDI_APPLICATION)); /*
icon style */

wcl.hCursor = LoadCursor(NULL, IDC_ARROW); /* cursor style */
wcl.lpszMenuName = NULL; /* no menu */
wcl.cbClsExtra = 0; /* no extra */
wcl.cbWndExtra = 0; /* 0 information needed */
wcl.hbrBackground =
    (HBRUSH) GetStockObject(WHITE_BRUSH) ; /* Make the window back-
ground white. */

/* Step 2: Register the window class. */
if(!RegisterClass (&wcl))
    return 0;

/* Step 3: Now that a window class has been registered, a window can be
created. */
hwnd = CreateWindow(szWinName, /* name of window class */
                    "FAKE 4", /* title */
                    WS_OVERLAPPEDWINDOW, /* window style - normal */
                    CW_USEDEFAULT, /* X coordinate - let Windows
decide */

                    CW_USEDEFAULT, /* y coordinate - let Windows
decide */

                    CW_USEDEFAULT, /* width - let Windows decide */
                    CW_USEDEFAULT, /* height - let Windows decide */
                    HWND_DESKTOP, /* no parent window */
                    NULL, /* no menu */
                    hThisInst, /* handle of this instance of the
program */

                    NULL /* no additional arguments */
);

/* Step 4: Display the window. */
ShowWindow(hwnd, nWinMode);

/* Step 5: Create the message loop. */
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg); /* allow use of keyboard */
    DispatchMessage(&msg); /* return control to Windows */
}

return msg.wParam;
```

```
}

//-----
// Name: WndProc
// Desc: This function is called by Windows and is passed
//           messages from the message queue
//-----
LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
                        WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_DESTROY: /* terminate the program */
            PostQuitMessage(0);
            break;
        default: /* Let Windows process any messages not
                  specified in the preceding switch statement. */
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}
```

The window function used by this program is called `WndProc()`. It is the function that Windows calls to communicate with your program. To indicate this call, it's declared as a callback function. Program execution starts with `WinMain()`, which is passed four parameters.

```
int WINAPI WinMain(
    HINSTANCE hInstance,          // handle to current instance
    HINSTANCE hPrevInstance,      // handle to previous instance
    LPSTR lpCmdLine,             // command line
    int nCmdShow                 // show state
);
```

`hInstance` refers to the current instance of your game. `hPrevInstance` is obsolete with the newer Windows versions. So for a Win32-based application, this parameter is always `NULL`.

`lpCmdLine` is a pointer to a string that holds any command line arguments specified when the application was begun. It excludes the program name. To retrieve the entire command line, use the `GetCommandLine()` function. `nCMDShow` specifies how the window is to be shown. Typical flags are

SW_HIDE	Hides the window and activates another window
SW_MAXIMIZE	Maximizes the specified window
SW_MINIMIZE	Minimizes the window
SW_RESTORE	Restores the window to its original size and position from a maximized or minimized position

### NOTE

What are handles and why are they used so often in Windows-based programs? In Windows, a handle is a pointer to a pointer, so it points to an address stored in a table or in a list. The address the handle points to can be used to access the object associated with the handle.

This kind of indirect access is necessary because the memory manager often moves objects around in memory—for example, to compact memory—without notifying your application that the address of the object has changed. The memory manager ensures that the object's handle is still valid. Windows uses many different kinds of objects through handles. These objects are named Windows objects. This name has nothing to do with the C++ objects. Windows objects can be used in C language-based as well as in C++ language-based programs.

As usual, take a look into your Win32 documentation provided with Visual C/C++ or online at [msdn.microsoft.com](http://msdn.microsoft.com) for more flags. Now let's work through the five steps needed to show you a window.

## STEP 1: DEFINE A WINDOW CLASS

Your Windows program has to first register a window class. When you register a window class, you are telling Windows about the form and function of the window you need. The word *class* is not used in its C++ sense, rather, it means style or type.

The `hInstance` field is assigned the current instance handle as specified in `hThisInst`. The name of the window class is pointed by `lpszClassName`, which points to the string `Crusher` in this case. The address of the window function is assigned to `lpfnWndProc`. No default style is specified, and the application icon

and the arrow cursor is used when the mouse is in the client area. We don't want to use a menu, but we would like to have a white background in the client area.

There are different class styles that can be used in WNDCLASS structure. To assign a style to a window class, assign the style to the style member of the WNDCLASS structure. Game programmer won't use a lot of these. The more interesting ones are

CS_NOCLOSE	Disables Close on the window menu
CS_DROPSHADOW	Enables the drop shadow effect on a window. This effect is turned on and off through SPI_SETDROPSHADOW and is supported by the next-generation window operating system with the code name Whistler.

The other styles don't seem to be of any benefit for us.

You always have to define a default shape for the mouse cursor and for the application's icon. You can define your own versions of these resources, or you can use one of the built-in styles, as this skeleton does. The style of the icon is loaded by LoadIcon(), whose prototype is shown here:

```
HICON LoadIcon (HINSTANCE hInst, LPCSTR lpIconName);
```

The first parameter is a handle to the instance of the module whose executable file contains the icon to be loaded. This parameter must be NULL when a standard icon is being loaded, like here. The second parameter is a NULL terminated string that contains the name of the icon resource to be loaded.

Alternatively, this parameter can contain the resource identifier in the low-order word and zero in the high-order word. Use the MAKEINTRESOURCE macro to create the value.

To use one of the predefined icons, set the hInst parameter to NULL and the lpIconName parameter to one of the following values:

IDI_APPLICATION	Default application icon
IDI_ASTERISK	Same as IDI_INFORMATION
IDI_ERROR	Hand-shaped icon
IDI_EXCLAMATION	Same as IDI_WARNING
IDI_HAND	Same as IDI_ERROR
IDI_INFORMATION	Asterisk icon
IDI_QUESTION	Question mark icon
IDI_WARNING	Exclamation point icon
IDI_WINLOGO	Windows logo icon

More interesting are the icons you might make on your own . . . just wait a few seconds.

If LoadIcon() succeeds, the return value is a handle to the newly loaded icon. If it fails, the return value is NULL.

To load the mouse cursor, use `LoadCursor()`. This function has the following prototype:

```
HCURSOR LoadCursor (HINSTANCE hInst, LPCSTR lpCursorName);
```

The first parameter is, as usual, the handle to the instance of the module whose executable file contains the cursor to be loaded. The second parameter is a pointer to a null-terminated string that contains the name of the cursor resource to be loaded. Alternatively, this parameter can consist of the resource identifier in the low-order word and zero in the high-order word. Use the `MAKEINTRESOURCE` macro to create this value.

To use one of the predefined cursors, the application must set the `hInstance` parameter to `NULL` and the `lpCursorName` parameter to one the following values:

<code>IDC_APPSTARTING</code>	Standard arrow and small hourglass
<code>IDC_ARROW</code>	Standard arrow
<code>IDC_CROSS</code>	Crosshair
<code>IDC_HAND</code>	Hand (Windows 2000 only)
<code>IDC_HELP</code>	Arrow and question mark
<code>IDC_IBEAM</code>	I-beam
<code>IDC_ICON</code>	Obsolete for applications marked version 4.0 or later
<code>IDC_NO</code>	Slashed circle
<code>IDC_SIZE</code>	Obsolete for applications marked version 4.0 or later; use <code>IDC_SIZEALL</code>
<code>IDC_SIZEALL</code>	Four-pointed arrow pointing north, south, east, and west
<code>IDC_SIZENESW</code>	Double-pointed arrow pointing northeast and southwest
<code>IDC_SIZENS</code>	Double-pointed arrow pointing north and south
<code>IDC_SIZENWSE</code>	Double-pointed arrow pointing northwest and southeast
<code>IDC_SIZEWE</code>	Double-pointed arrow pointing west and east
<code>IDC_UPARROW</code>	Vertical arrow
<code>IDC_WAIT</code>	Hourglass

In this example, a handle to the background color brush is obtained using `GetStockObject()`. A brush is a resource that paints the screen using a predetermined size, color, or pattern. This function can retrieve a handle to one of the stock pens, brushes, fonts, or palettes.

### NOTE

`LoadIcon()` can only load an icon whose size conforms to the `SM_CXICON` and `SM_CYICON` system metric values.

```
HGDIOBJ GetStockObject(int fnObject);
```

fnObject specifies the type of stock object. Game programmers won't use this method a lot. A few flags that might be useful are:

BLACK_BRUSH	Black brush
DKGRAY_BRUSH	Dark gray brush
GRAY_BRUSH	Gray brush
LTGRAY_BRUSH	Light gray brush
NULL_BRUSH	Null brush
WHITE_BRUSH	White brush

#### WINDOWS DATA TYPES

The skeleton program does not extensively use C/C++ data types, such as *int* or *char \**. Instead, all data types used by Windows have been *typedefed* within the *windows.h* include file or its related include files. This header file has to be included in all Windows programs.

Some of the most common data types are HANDLE (32-bit int), HWND, BYTE (8-bit sized), WORD (16-bit unsigned short integer), DWORD (32-bit unsigned long integer), UINT (32-bit unsigned integer), LONG (32-bit long), BOOL (integer), LPSTR (pointer to a string), LPCSTR (const pointer to a string), and HANDLE. As you see, there are a number of handle types, which start with an *H*, but they are all the same size as HANDLE. A HANDLE is simply a value that identifies some resources.

Windows defines several structures, such as MSG and WNDCLASS. As you will see in a few seconds, MSG holds a Windows message.

Back to our skeleton code: Once the window class has been fully specified, it is registered with Windows using *RegisterClass()*.

#### STEP 2: REGISTER THE WINDOW CLASS

*RegisterClass()* registers a window class for subsequent use in calls to *CreateWindow()* or *CreateWindowEx()*. All window classes that an application registers are unregistered when it terminates.

```
ATOM RegisterClass(CONST WNDCLASS *lpWndClass);
```

The function returns a value that identifies the window class. ATOM is a *typedef* that means WORD.

#### STEP 3: CREATING A WINDOW OF THAT CLASS

Once a window class has been defined and registered, your game can actually create a window of that class with *CreateWindow()*:

```
HWND CreateWindow(
    LPCTSTR lpClassName,           // registered class name
    LPCTSTR lpWindowName,          // title of window
    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,                  // menu handle or child identifier
    HINSTANCE hInstance,           // handle to application instance
    LPVOID lpParam                // window-creation data
);
```

A lot of the parameters to `CreateWindow()` may be defaulted or specified as `NULL`. In fact, most of the `x`, `y`, `nWidth`, and `nHeight` parameters will simply use the macro `CW_USEDEFAULT` in most applications, which tells Windows to select an appropriate size and location for the window. However, in a game application we would like to set the window with a predefined width and height. As you will see, `AdjustWindowRect()` can help by calculating the required size of the window rectangle based on the desired client rectangle size.

The handle to the parent window `hWndParent` must be specified as `HWND_DESKTOP`. If the window will not contain a menu, then `hMenu` must be `NULL`. If no additional information is required, as is most often the case, then `lpParam` is `NULL`.

You must set the remaining four parameters explicitly for your game. `lpClassName` must point to the name of the window class. The title of the window is a string pointed by `lpWindowName`. The style of the window actually created is by the value of `dwStyle`. The macro `WS_OVERLAPPEDWINDOW` specifies a standard window that has a system menu, a border, and minimize, maximize,, and close boxes. As a game programmer, you will set your styles on your own in a more differentiated manner. For example, you won't use a close box if your game only uses the Esc key to quit, and you won't use a minimize/maximize button if your game only supports 640-by-480 or 800-by-600 windows, because these are the rectangles it is optimized for.

Here's a selection of the different window styles:

`WS_BORDER`

Creates a window that has a thin line border.

`WS_CAPTION`

Creates a window that has a title bar (includes the `WS_BORDER` style).

## CAUTION

If an application calls `CreateWindow()` to create a multiple document interface (MDI) client window, `lpParam` must point to a `CLIENTCREATESTRUCT` structure.

WS_CHILD	Creates a child window. A window with this style cannot have a menu bar. This style cannot be used with the WS_POPUP style.
WS_CHILDWINDOW	Same as the WS_CHILD style.
WS_DISABLED	Creates a window that is initially disabled. A disabled window cannot receive input from the user. To change this after a window has been created, use EnableWindow.
WS_DLGFRA ME	Creates a window that has a border of the style typically used with dialog boxes. A window with this style cannot have a title bar.
WS_ICONIC	Creates a window that is initially minimized. Same as the WS_MINIMIZE style.
WS_MAXIMIZE	Creates a window that is initially maximized.
WS_MAXIMIZEBOX	Creates a window that has a maximize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.
WS_MINIMIZE	Creates a window that is initially minimized. Same as the WS_ICONIC style.
WS_MINIMIZEBOX	Creates a window that has a minimize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.
WS_OVERLAPPED	Creates an overlapped window. An overlapped window has a title bar and a border. Same as the WS_TILED style.
WS_OVERLAPPEDWINDOW	Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_TILEDWINDOW style.
WS_POPUP	Creates a pop-up window. This style cannot be used with the WS_CHILD style.
WS_POPUPWINDOW	Creates a pop-up window with WS_BORDER, WS_POPUP, and WS_SYSMENU styles. The WS_CAPTION and WS_POPUPWINDOW styles must be combined to make the window menu visible.
WS_SIZEBOX	Creates a window that has a sizing border. Same as the WS_THICKFRAME style.
WS_SYSMENU	Creates a window that has a window menu on its title bar. The WS_CAPTION style must also be specified.

WS_THICKFRAME	Creates a window that has a sizing border. Same as the WS_SIZEBOX style.
WS_TILED	Creates an overlapped window. An overlapped window has a title bar and a border. Same as the WS_OVERLAPPED style.
WS_TILEDEWINDOW	Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_OVERLAPPEDWINDOW style.
WS_VISIBLE	Creates a window that is initially visible.

This style can be turned on and off by using ShowWindow or SetWindowPos.

The styles used in the Common files in d3dapp.cpp are

```
m_dwWindowStyle = WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME |
                  WS_MINIMIZEBOX | WS_VISIBLE;
```

If you remove WS\_THICKFRAME, the window is not resizable. The WS\_VISIBLE flag indicates that this window should be visible after creation and not after a call to ShowWindow(). A title bar is used because of WS\_CAPTION. There's no WS\_MAXIMIZEBOX, so that the maximize button is grayed.

CreateWindow() returns the handle of the window it creates or NULL in case the window cannot be created. If you haven't set the WS\_VISIBLE flag, you have to make your window visible with a call to ShowWindow().

#### STEP 4: DISPLAY THE WINDOW

If you would like to be able to determine when your window shows up, you might use ShowWindow():

```
BOOL ShowWindow(HWND hWnd, int nCmdShow);
```

The handle of the window to display is specified in hWnd. nCmdShow specifies how the window is to be shown. The first time the window is displayed, you will want to pass WinMain()'s nCmdShow as the nCmdShow parameter. In subsequent calls, you can use, for example, one of the following values:

SW_FORCEMINIMIZE	Minimizes a window, even if the thread that owns the window is hung. This flag should only be used when minimizing windows from a different thread (Windows 2000 only).
SW_HIDE	Hides the window and activates another window.
SW_MAXIMIZE	Maximizes the specified window.

#### NOTE

**What's the difference between CreateWindow() and CreateWindowEx()?**  
**The latter one creates windows with an extended window style.**

SW_MINIMIZE	Minimizes the specified window and activates the next top-level window in the Z order.
SW_RESTORE	Activates and displays the window. If the window is minimized or maximized, the system restores it to its original size and position. An application should specify this flag when restoring a minimized window.
SW_SHOW	Activates the window and displays it in its current size and position.
The ShowWindow() function returns the previous display status of the window. If the window was displayed, then nonzero is returned. If the window has not been displayed, zero is returned.	

## STEP 5: CREATE THE MESSAGE LOOP

The final part of WinMain() is the message loop. It receives and processes messages sent by Windows. When an application is running, it is continually being sent messages, which are stored in the application's message queue until they can be read and processed. Each time your application is ready to read another message, it must call GetMessage():

```
BOOL GetMessage(
    LPMMSG lpMsg,           // message information
    HWND hWnd,              // handle to window
    UINT wMsgFilterMin,     // first message
    UINT wMsgFilterMax      // last message
);
```

The message is received by lpMsg. All Windows messages are of structure type MSG, shown here:

```
typedef struct tagMSG
{
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG, *PMSG;
```

You will find the message data in message with additional information in wParam and lParam. In its hwnd and message fields, a MSG structure identifies the message being referred to by the window that the message affects. In its wParam and lParam fields, this structure stores information about the kind of event the message refers to and the source of the event; for example, if the event is caused by a keyboard input, the wParam and lParam fields identify the key being pressed and also reveal whether a command key was pressed at the same time. There is a time stamp in milliseconds in time, and pt will hold the coordinates of the mouse in a POINT structure:

```
typedef struct tagPOINT
{
    LONG x, y;
} POINT;
```

The `wMsgFilterMin` and `wMsgFilterMax` specify a range of messages that will be received. If you want your application to receive all of the messages, you will specify both *min* and *max* as 0.

`GetMessage()` returns zero when the user terminates the program, causing the message loop to terminate. Otherwise it returns nonzero.

A game programmer won't use `GetMessage()` in all situations; he will use `PeekMessage()` in several situations.

```
BOOL PeekMessage(
    LPMMSG lpMsg,           // message information
    HWND hWnd,              // handle to window
    UINT wMsgFilterMin,     // first message
    UINT wMsgFilterMax,     // last message
    UINT wRemoveMsg         // removal options
);
```

There's one additional parameter, `wRemoveMsg`. It specifies how messages are handled. This parameter can be one of the following values:

`PM_NOREMOVE` Messages are not removed from the queue after processing by `PeekMessage()`.

`PM_REMOVE` Messages are removed from the queue after processing by `PeekMessage()`.

You should use `PM_REMOVE`.

By default, all message types are processed. One of the more interesting features that came up with the advent of Windows 98, ME, and 2000 is the ability to specify that only a certain message should be processed.

`PM_QS_INPUT` Process mouse and keyboard messages.

`PM_QS_PAINT` Process paint messages.

`PM_QS_POSTMESSAGE` Process all posted messages, including timers and hotkeys.

`PM_QS_SENDFMESSAGE` Process all sent messages.

OK, now what's the difference, and why should game programmers use `PeekMessage()`?

Internally to Windows, `GetMessage()` and `PeekMessage()` execute the same code. The major difference between the two is the case where there are no messages to be returned to the application. In this case, `GetMessage()` puts the application to sleep, while `PeekMessage()` returns to the application with a `NULL` value. We wouldn't like our game to sleep until the next message arrives in the message queue, so

it's better to use `PeekMessage()` when the game is active and `GetMessage()` when the game pauses. Pseudocode might look like this:

```
BOOL bGotMsg;
MSG msg;
While(WM_QUIT != msg.message)
{
    if (m_bActive)
        bGotMsg = PeekMessage(&msg, NULL, 0, 0, PM_REMOVE);
    else
        bGotMsg = GetMessage(&msg, NULL, 0, 0);
    ...
}
```

In the `while(GetMessage ...)` routine, there are two more function calls:

```
...
TranslateMessage(&msg); /* allow use of keyboard */
DispatchMessage(&msg); /* return control to Windows */
...
```

`TranslateMessage()` translates virtual key codes generated by Windows into character messages.

Once the message has been read and, if necessary, translated, `DispatchMessage()` will send the message to the window procedure, here `WndProc()`. The figure shows you the relationship between the message queue, the message pump in `WinMain()`, and the window procedure. `GetMessage` stands for `GetMessage()` and `PeekMessage()`:

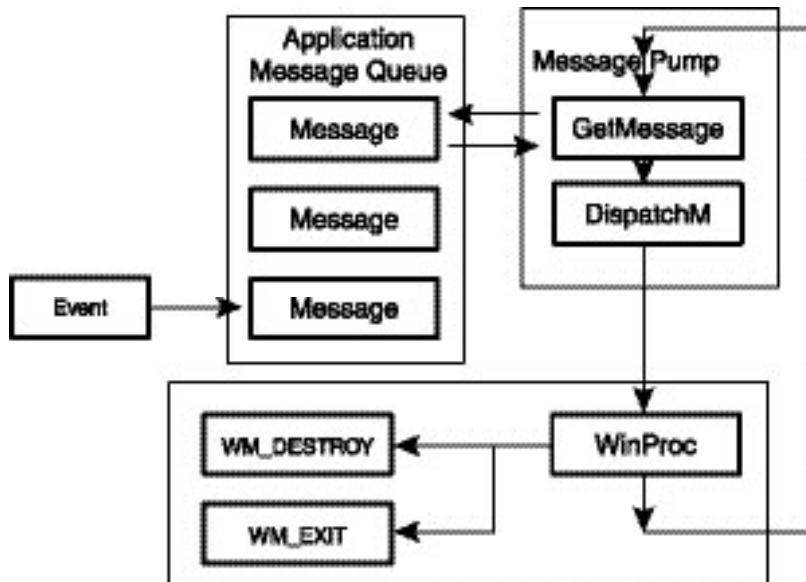


Figure A.3: Message pump

Once the message loop terminates, WinMain() returns the value of msg.wParam to Windows. This value contains the return code generated when your program terminates.

## THE WINDOW PROCEDURE

The second function in our skeleton is the window procedure. Its name is provided to the window class function. I chose WndProc(). This procedure receives the first four members of the MSG structure as parameters. The skeleton window procedure responds to only one message, the WM\_DESTROY message. This message is sent when the user terminates the program. PostQuitMessage() causes a WM\_QUIT message to be sent to your application, which causes GetMessage() / PeekMessage() to return FALSE, thus stopping the program.

msg.wParam will return with the WM\_QUIT value if nothing goes wrong.

Any other messages received by WinProc() are passed to Windows via a call to DefWindowProc(). It calls the default window procedure to provide processing for any window messages that an application does not process. This function ensures that every message is processed.

## A WINDOW SKELETON OPTIMIZED FOR GAMES

After digging through the default window skeleton that is shown in nearly every window programming book, we will take a look at an game-optimized version:

```
#include <windows.h>
#include "resource.h"
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
VOID Render();
char szWinName[] = "MyWin"; /* name of window class */
BOOL bActive = TRUE;
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                    LPSTR lpszArgs, int nWinMode)
{
    HWND hwnd;
    MSG msg;
    /* Step 1: Define a window class. */
    WNDCLASS wcl;
    wcl.hInstance = hThisInst; /* handle to this instance */
    wcl.lpszClassName = szWinName; /* window class name */
    wcl.lpfnWndProc = WndProc; /* window function */
    wcl.style = 0; /* default style */
```

### NOTE

Javier F. Otaegui at [www.gamedev.net/reference/articles/article1249.asp](http://www.gamedev.net/reference/articles/article1249.asp) has another interesting approach to get rid of the window message pump. He uses a second thread that handles the window message routine.

```
wcl.hIcon = LoadIcon(hThisInst, MAKEINTRESOURCE(IDI_ICON1)); /* icon
style */
wcl.hCursor = LoadCursor( hThisInst, MAKEINTRESOURCE(IDC_HULLA)) ; /* cursor style */
wcl.lpszMenuName = NULL; /* no menu */
wcl.cbClsExtra = 0; /* no extra */
wcl.cbWndExtra = 0; /* information needed */
/* Make the window background white. */
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH) ;
/* Step 2: Register the window class. */
if(!RegisterClass (&wcl))
    return 0;
DWORD dwWindowStyle = WS_SYSMENU;
DWORD dwCreationWidth = 640;
DWORD dwHeight = 480;
RECT rc;
SetRect(&rc, 0, 0, dwWidth, dwHeight);
AdjustWindowRect(&rc, dwWindowStyle, TRUE);
/* Step 3: Now that a window class has been registered,
           a window can be created. */
hwnd = CreateWindow(szWinName, /* name of window class */
                    "FAKE 4", /* title */
                    dwWindowStyle, /* window style - normal */
                    CW_USEDEFAULT, /* X coordinate - let Windows
decide */
                    CW_USEDEFAULT, /* y coordinate - let Windows
decide */
                    (rc.right - rc.left), /* */
                    (rc.bottom - rc.top), /* */
                    HWND_DESKTOP, /* no parent window */
                    NULL, /* no menu */
                    hThisInst, /* handle of this instance of the pro-
gram */
                    NULL /* no additional arguments */
);
/* Step 4: Display the window. */
ShowWindow(hwnd, nWinMode);
BOOL bGotMsg;
/* Step 5: Create the message loop. */
while (WM_QUIT != msg.message)
{
    if( bActive)
```

```
        bGotMsg = PeekMessage (&msg, NULL, 0U, 0U,
PM_REMOVE);
    else
        bGotMsg = GetMessage (&msg, NULL, 0U, 0U);
    if(bGotMsg)
    {
        TranslateMessage(&msg); /* allow use of keyboard
*/
        DispatchMessage(&msg); /* return control to
Windows */
    }
    else
    {
        if(bActive)
            Render();
    }
}
return msg.wParam;
}

//_____
// Name: WndProc
// Desc: This function is called by Windows and is passed
//           messages from the message queue
//_____
LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
                        WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_DESTROY: /* terminate the program */
            PostQuitMessage(WM_QUIT);
            break;
        case WM_KEYDOWN:
        {
            switch (wParam)
            {
                case VK_ESCAPE:
                    PostQuitMessage(WM_QUIT);
                    break;
                case VK_F1:
                {
```

```

        bActive = FALSE;
        MessageBox( hwnd, "Here comes your help text",
                    "Help for FAKE 4", MB_ICONQUESTION|MB_OK |
MB_SYSTEMMODAL );
    }
    bActive = TRUE;
}
break;
}
}

return DefWindowProc(hwnd, message, wParam, lParam);
}
//-----
// Name: Render
// Desc: dummy function to show the use of the enhanced skeleton
//-----
VOID Render()
{
};

```

You may find the first enhancements just before Step 3:

```

DWORD dwWindowStyle = WS_POPUP | WS_CAPTION;
DWORD dwWidth = 640;
DWORD dwHeight = 480;
RECT rc;
SetRect(&rc, 0, 0, dwWidth, dwHeight);
AdjustWindowRect(&rc, dwWindowStyle, TRUE);

```

We define a window style only with a title bar. It should be 640 by 480 pixels in width and height. SetRect() sets the coordinates of the specified rectangle. AdjustWindowRect() calculates the required size of the window rectangle, based on the desired client rectangle size. So we really get a client window size of 640 by 480. The real stuff is located in the message loop in Step 5:

```

/* Step 5: Create the message loop. */
while (WM_QUIT != msg.message)
{
    if( bActive)
        bGotMsg = PeekMessage (&msg, NULL, 0U, 0U,
PM_REMOVE);

```

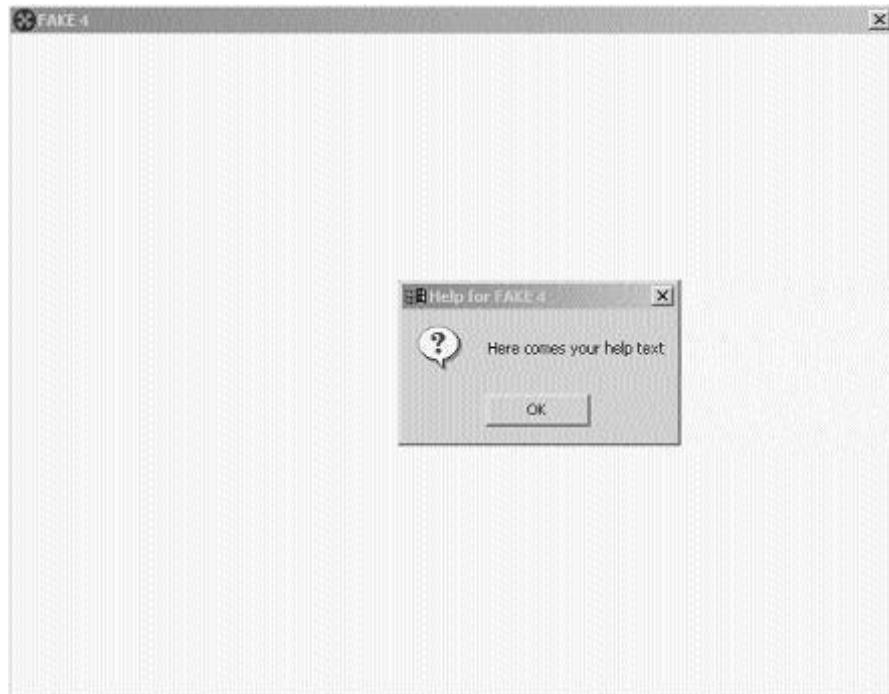
```
    else
        bGotMsg = GetMessage (&msg, NULL, 0U, 0U);
    if(bGotMsg)
    {
        TranslateMessage(&msg); /* allow use of keyboard
*/                                DispatchMessage(&msg); /* return control to
Windows */
    }
    else
    {
        if(bActive)
            Render();
    }
}
```

When the app is active, PeekMessage() returns to the application with NULL when there is no message in the message queue. When the app is inactive, GetMessage() puts the application to sleep when no message is in the message queue. TranslateMessage() and DispatchMessage() are only called when a message has worn out. Render() is only called when the app is active. I made a small example for the case that the app is not active:

```
case VK_F1:
{
    bActive = FALSE;
    MessageBox( hwnd, "Here comes your help text",
                "Help for FAKE 4", MB_ICONQUESTION|MB_OK |
MB_SYSTEMMODAL );
    bActive = TRUE;
}
break;
```

If the message box shows, bActive is FALSE and Render() is not called.

That's it. I hope you got the idea of how to set up window code for use in games. You will not use a lot of window code, because in full-screen games all the dialog boxes have to be hand-built by you. Usual window dialog boxes don't fit into games.

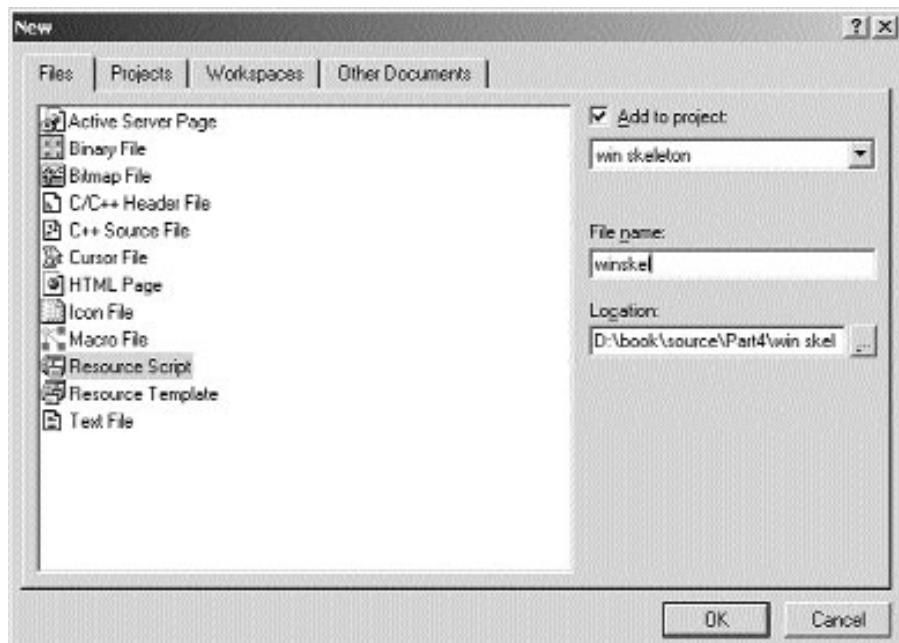


**Figure A.4:** Window skeleton 2

## WINDOWS RESOURCES

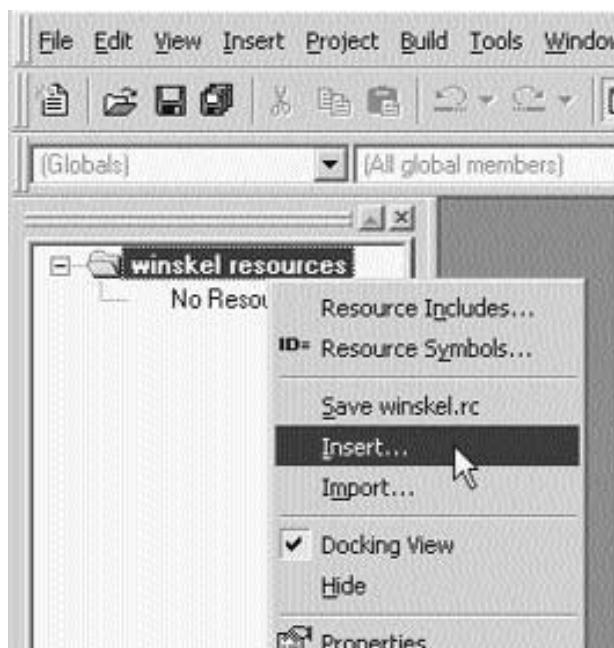
In the beginning of Windows programming, all programmers had to set resources by hand. Nowadays, the Visual C/C++ integrated development environment helps a lot. You have to work through the following steps to build and include resources: File/New/Resource Script.

Give your script a nice name, for example, winskel. Click on OK. The new resource file will have the name winskel.rc.



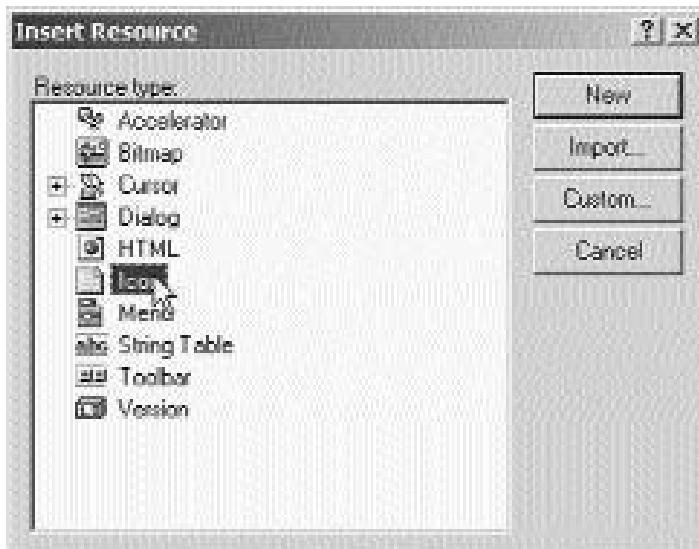
**Figure A.5:** Creating a resource script

Now click on the Resource View tab in your workspace, then right-click on winskel resources:



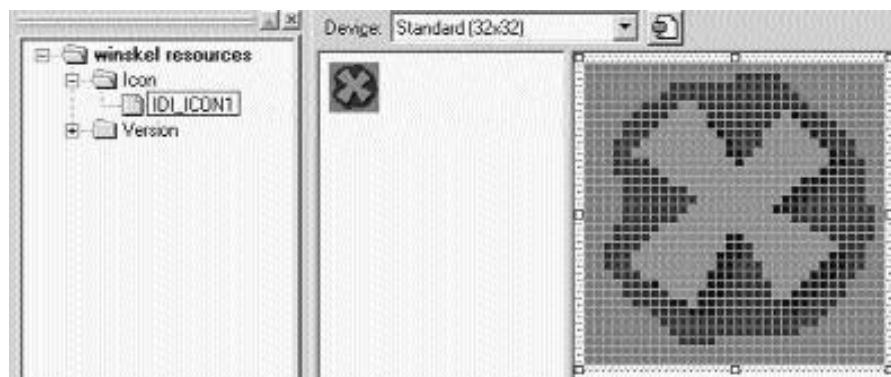
**Figure A.6:** Insert a resource

After clicking on Insert, a dialog box appears that lets you choose a resource.



**Figure A.7:** Create an icon

I produced an icon with the help of the standard directx.ico by using a blue color (shown grey shaded here) as the background and transparent as the foreground color.

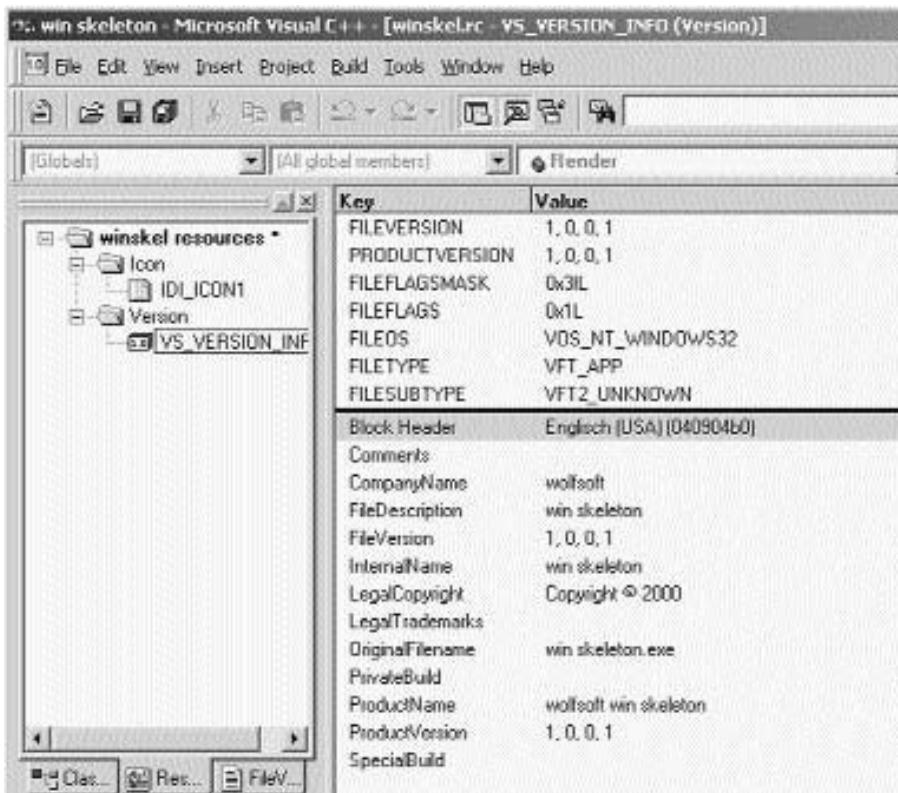


**Figure A.8:** Coloring an icon

To load the icon, you have to include the file resource.h and load the icon with

```
wcl.hIcon = LoadIcon(hThisInst, MAKEINTRESOURCE(IDI_ICON1)); /* icon style */
```

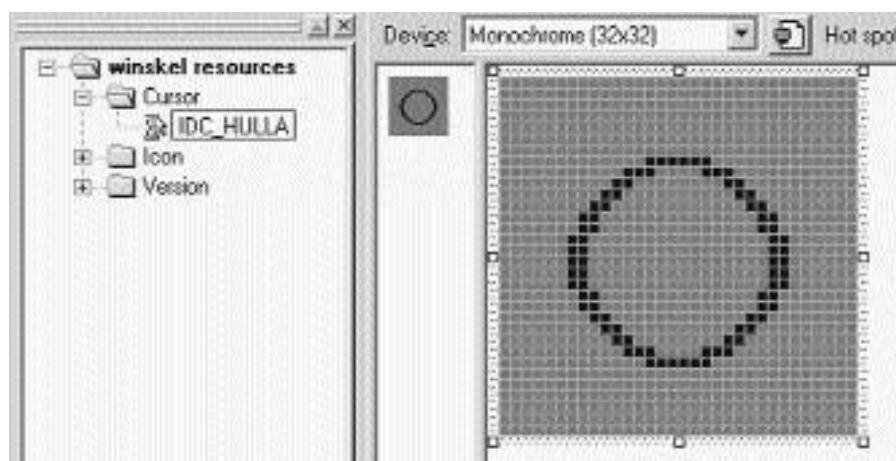
I also chose to create a version control resource.



**Figure A.9:** Creating a version control resource

Using a version control resource helps you to arrange different versions of your program and to hold your copyright information in the .exe file.

Oh, yes, a special cursor would be fine. Click on the ResourceView tab in your workspace, right-click on winskel resources, then choose Cursor.



**Figure A.10:** Cursor production

So we have three resources for our simple skeleton.



**Figure A.11:** Three resources

Here we are: You know how to build your first skeleton with Resources. Whereas you have to enter all of the skeleton code by hand into the Visual C/C++ editor, you get a lot of help in creating resources from the IDE.

That's it with our first window skeleton . . . ah, just one thing . . . did you notice that the whole example was in C? OK, now it's time to do this stuff in C++.

## ADDITIONAL RESOURCES

You will find more info on GetMessage() and PeekMessage() at [msdn.microsoft.com/library/techart/msdn\\_getpeek.htm](http://msdn.microsoft.com/library/techart/msdn_getpeek.htm).

Another interesting approach to optimize the message pump can be found at [www.gamedev.net/reference/articles/article1249.asp](http://www.gamedev.net/reference/articles/article1249.asp). Javier F. Otaegui uses a second thread for the message queue.

*This page intentionally left blank*

# **APPENDIX B**

## **C++ PRIMER**

**A** few years ago, there were a lot of discussions going around in the newsgroups on using C or C++ in games. One faction said that C produces faster code and named games like Quake and Unreal as proof. The other faction said that C++ is more elegant, helps structure your game in a better way, and is economically cheaper when used by production teams.

Practice—as always—finds the practical way: Take the speed of C and the ability to structure game code from C++. So I suppose that a lot of game programmers don't use a lot of C++ stuff. If you take a look at the examples in the DirectX SDK, you will see a lot of C++ code and traditional C code.

This chapter covers a host of topics, including the following:

- A brief explanation of the uses and features of object-oriented programming
- An overview of some C++ keywords and data types that aren't available in C or that are used differently in C++ than in C
- Creating and using classes, or how the C-language structure has evolved into C++ class, one of the keystones of object-oriented programming in C++
- Other C++ programming techniques such as access specifiers, operators for accessing member methods and member variables of classes, constructors, and destructors, copy constructors, and the *this* pointer

## WHAT'S OBJECT-ORIENTED PROGRAMMING?

In an object-oriented view of programming, a program describes a system of objects interacting. This is contrary to a procedure-oriented view of programming, where a program describes a series of steps to be performed.

Object-oriented programming involves a few key concepts. The most basic of these is abstraction, which makes writing large programs simpler. Another is encapsulation, which makes it easier to change and maintain a program. The concept of class hierarchies is a powerful classification tool that can make a program easily extensible.

To understand C++ as an object-oriented language, you have to understand how C++ takes advantages of these concepts.

### ABSTRACTION

Abstraction is the process of ignoring details in order to concentrate on essential things. A so-called high-level programming language supports a high level of abstraction. The usual example is the comparison of a program written in assembly and one written in C. The assembly source contains a very detailed description of what the computer does to perform the task, whereas the C source gives a much more abstract

description of what the computer does, and that abstraction makes the program clearer and easier to understand.

All procedural languages support procedural abstraction, where a piece of source code is put into a function and reused this way.

For example, matrix multiplication:

```
FLOAT pM[16];
ZeroMemory( pM, sizeof(D3DXMATRIX) );
for( WORD i=0; i<4; i++ )
    for( WORD j=0; j<4; j++ )
        for( WORD k=0; k<4; k++ )
            pM[4*i+j] += pM1[4*i+k] * pM2[4*k+j];
memcpy( pOut, pM, sizeof(D3DXMATRIX) );
```

You wouldn't like to write this piece of code every time two 4-by-4 matrices have to be multiplied. Instead, you abstract it by embedding it into a function call like this:

```
D3DXMATRIX* D3DXMatrixMultiply (D3DXMATRIX* pOut,
                                CONST D3DXMATRIX* pM1,
                                CONST D3DMATRIX* pM2)
{
    FLOAT pM[16];
    ZeroMemory( pM, sizeof(D3DXMATRIX) );
    for( WORD i=0; i<4; i++ )
        for( WORD j=0; j<4; j++ )
            for( WORD k=0; k<4; k++ )
                pM[4*i+j] += pM1[4*i+k] * pM2[4*k+j];
    memcpy( pOut, pM, sizeof(D3DXMATRIX) );
    return (pOut);
}
```

Using `D3DXMatrixMultiply()` will lead to a more readable and understandable source and saves you a lot of typing.

Another kind of abstraction you already know is data abstraction, which lets you ignore details of how a data type is represented. The C language provides data types like `int` or `float`, which let you type in something like 42.5, rather than some hexadecimal bytes. Thankfully, floating-point arithmetic performed in binary is something that C programmers don't have to worry about. On the other hand, C does not support the abstraction of strings, because it requires you to manipulate strings as a series of characters.

But C supports user-defined data abstraction through structures and `typedefs`. Programmers might use structures to manipulate several pieces of information as a unit instead of individually.

```
Struct Mesh
{
    char cName[68];                                // 65 chars, 32-bit aligned ==
68 chars in file
    int iNumVertices;                            // Mesh vertices
    int iMeshFrameNum;                           // animation frames in mesh
    VMESHFRAME *vMeshFrames; // stores vertices per animation frame in
                             //
vertices[FrameNum][VertexNum]
    int iNumTriangles;                         // Mesh triangles
    TRIANGLEVERT *pTriangles;
    int iNumTextures;                          // Mesh textures
    TEXNAMES *pTexNames;
    LPDIRECT3DTEXTURE8* pTexturesInterfaces;
    TEXCOORDS *pfTextureCoords; // Mesh tex coordinates
    TEXCOORDS *pfEnvTextureCoords; // not used here
};

This user-defined data type lets you manipulate the variables in a more understandable way. For example:
```

```
Mesh.iNumVertices = 5;
```

But there is no conceptual advantage. You get a better data abstraction by defining data types and data structure with `typedef`.

```
typedef int TRIANGLEVERT[3];
typedef char TEXNAMES[68];
typedef D3DXVECTOR3 *VMESHFRAME;
typedef struct
{
    float u;
    float v;
} TEXCOORDS;
typedef struct
{
    char cName[68];                                // 65 chars, 32-bit aligned ==
68 chars in file
    int iNumVertices;                            // Mesh vertices
    int iMeshFrameNum;                           // animation frames in mesh
    VMESHFRAME *vMeshFrames; // stores vertices per animation frame in
// vertices[FrameNum][VertexNum]
```

```

    int iNumTriangles;           // Mesh triangles
    TRIANGLEVERT *pTriangles;
    int iNumTextures;           // Mesh textures
    TEXNAMES *pTexNames;
    LPDIRECT3DTEXTURE8* pTexturesInterfaces;
    TEXCOORDS *pfTextureCoords;           // Mesh tex coordinates
    TEXCOORDS *pfEnvTextureCoords;        // not used here
} MD3MESH;

```

Now MD3MESH is an abstract data type that might be used in combination with functions as their parameter. These functions handle the details of filling the specific fields.

This traditional procedural approach lets you view data abstraction and procedural abstraction as two distinct techniques.

A more elegant way would be to link them, and here is the object-oriented programming approach with classes.

## CLASSES

A class combines procedural and data abstraction by describing everything about a high-level entity at once. Think of our CMD3 class, for example, which holds all of the geometry data and texture interface pointers for as many .md3 models as you like.

```

class CMD3Model
{
private:
    int                               iNumMeshes;
    MD3MESH                          *pMd3Meshes;
    MD3BONEFRAME                     *md3BoneFrame;
    MD3TAG                            **md3Tags;
    FILE *LogFile;                   // the log file for md3 geometry data.txt
                                    // and md3 textures.txt
    LPDIRECT3DVERTEXBUFFER8 m_pVB;    // Buffer to hold vertices
    long ReadLong(FILE* File);       // file i/o helper
    short ReadShort(FILE* File);
    FLOAT ReadFloat(FILE* File);
    void CreateVB(LPDIRECT3DDEVICE8 1pD3Ddevice); // vertex
buffer system
    void DeleteVB();
    void CreateTextures(LPDIRECT3DDEVICE8 1pD3Ddevice); // texture
system
    void DeleteTextures();

```

```
public:  
    BOOL CreateModel( char *fname, LPDIRECT3DDEVICE8 lpD3Ddevice);  
// class methods  
    void DeleteModel();  
    void InitDeviceObjects(LPDIRECT3DDEVICE8 lpD3Ddevice);  
    void DeleteDeviceObjects();  
    void Render(LPDIRECT3DDEVICE8 lpD3Ddevice);  
    CMD3Model();           // constructor/destructor  
    ~CMD3Model();  
};
```

Classes make it easy for you to create an additional layer of separation between your program and the computer. The high-level entities you define have the same advantages that floating-point numbers and print statements have when compared to bytes and MOV instructions.

This class is only accessible through five methods: CreateModel(), DeleteModel(), InitDeviceObjects(), DeleteDeviceObjects(), and Render(). There is no way to access variables from outside the class. This leads to another advantage of object-oriented programming: encapsulation.

## ENCAPSULATION

Encapsulation is the process of hiding the internal workings of a class to support or enforce abstraction. This requires drawing a sharp distinction between a class's interface, which has public visibility, and its implementation, which has private visibility. These are expressed by the

```
private:  
public:
```

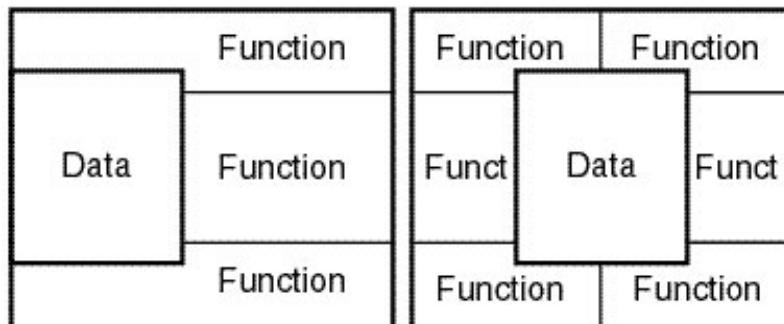
labels.

A class interface describes what a class can do, while its implementation describes how it does it. A user views an object in terms of the operations it can perform, not in terms of its data structure.

A truly encapsulated class surrounds or hides its data with its functions so that you can access the data only by calling the functions. An illustration might look like this:

### NOTE

The keyword-protected label enables you to make members visible to derived classes but not to the user. You can give derived classes more information about your class than you give users.



**Figure B.1:** Data encapsulation with functions (public vs. private data members)

On the left are the public data members, and on the right, the private data members.

The aim of hiding data is to make each module as independent of one another as possible. Ideally, a module has no knowledge of the data structures used by other modules, and it refers to those modules only through their interfaces.

As you have seen above, this is the case in the CMD3 class. Even if the class interface changes in the future, it is still a good idea to use an encapsulated class rather than accessible data structures. You may add the changes to the interface through the existing interface. For example, if the older interface is Direct3D7, you might add Direct3D8. Any code that uses the old interface still works correctly.

Now, after all this theory, let us get a little bit of practice.

### NOTE

The C++ encapsulation does not provide a guarantee of safety. A programmer who is intent on using a class's private data can always use the & and \* operators to gain access to them.

## DECLARING A CLASS

We will start where we left in Appendix A by building a C++ version of the window skeleton program we built there. As usual, I start with a piece of source:

```
#ifndef APP_H
#define APP_H
class Capp
{
protected:
    TCHAR* m_strWindowTitle;
    DWORD m_dwWidth;
    DWORD m_dwHeight;
private:
    BOOL m_bActive;
    HRESULT Render();
```

```
public:  
    // used in Fake.cpp to create, run and pause fake  
    virtual HRESULT Create (HINSTANCE hInstance);  
    virtual LRESULT MsgProc (HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM  
    lParam);  
    virtual WPARAM Run();  
    virtual VOID Pause(BOOL bPause);  
    // internal constructor  
    CApp();  
};  
#endif
```

You will find that class declaration in the app.h header file, which you can find in the win skeleton++ directory in the example source code directories.

The

```
#ifndef APP_H  
#define APP_H  
...  
#endif
```

preprocessor directives are used for conditional compilation. This prevents multiple inclusion of header files in a multimodule program. The class declaration looks similar to a structure declaration, except that it has both functions and data as members, instead of just data. It declares the following:

- The contents of each instance of CApp: window title, width and height, and the status of rendering
- The prototypes of five methods that you can use with these data

I supplied the definitions of the member functions in App.cpp:

```
#define STRICT  
#include <windows.h>  
#include <windowsx.h>  
#include <tchar.h>  
#include "App.h"  
#include "resource.h"  
//  
// global this pointer  
//  
static CApp* g_pCApp = NULL;  
//  
// Name: FAKEApp()  
// Desc: Constructor
```

```
//-----
CApp::CApp()
{
    g_pCApp = this;
    m_bActive = FALSE;
    m_strWindowTitle = _T("FAKE 4");
    m_dwWidth = 640;
    m_dwHeight = 480;
}

//-----
// Name: WndProc()
// Desc: static msg handler which passes messages to the application class
//           in the fake.cpp file
//-----

HRESULT CALLBACK WndProc (HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    return g_pCApp->MsgProc (hWnd, uMsg, wParam, lParam);
}

//-----
// Name: Create()
// Desc: static msg handler which passes messages to the application class
//           in the fake.cpp file
//-----

HRESULT CApp::Create(HINSTANCE hThisInst)
{
    HWND hwnd;
    /* Step 1: Define a window class. */
    WNDCLASS wcl;
    wcl.hInstance = hThisInst; /* handle to this instance */
    wcl.lpszClassName = _T("Crusher") ; /* window class name */
    wcl.lpfnWndProc = WndProc; /* window function */
    wcl.style = 0; /* default style */
    wcl.hIcon = LoadIcon(hThisInst, MAKEINTRESOURCE(IDI_ICON1)); /* icon
style */
    wcl.hCursor = LoadCursor( hThisInst, MAKEINTRESOURCE(IDC_HULLA)) ; /* cursor style */
    wcl.lpszMenuName = NULL; /* no menu */
    wcl.cbClsExtra = 0; /* no extra */
    wcl.cbWndExtra = 0; /* information needed */
    /* Make the window background white. */
    wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH) ;
```

```
/* Step 2: Register the window class. */
if(!RegisterClass (&wcl))
    return 0;
DWORD dwWindowStyle = WS_POPUP | WS_CAPTION | WS_VISIBLE | WS_SYSMENU;
RECT rc;
SetRect(&rc, 0, 0, m_dwWidth, m_dwHeight);
AdjustWindowRect(&rc, dwWindowStyle, TRUE);
/* Step 3: Now that a window class has been registered,
           a window can be created. */
hwnd = CreateWindow(_T("Crusher"), /* name of window class */
                    m_strWindowTitle, /* title */
                    dwWindowStyle, /* window style -
normal */  
                               CW_USEDEFAULT, /* X coordinate -
let Windows decide */  
                               CW_USEDEFAULT, /* y coordinate -
let Windows decide */  
                               (rc.right - rc.left), /* * */
                               (rc.bottom - rc.top), /* * */
                               HWND_DESKTOP, /* no parent win-
dow */  
                               NULL, /* no menu */
                               hThisInst, /* handle of this
instance of the program */
                               NULL /* no additional arguments
*/;  
/* Step 4: Display the window. */
// ShowWindow(hwnd, nWinMode); // obsolete because of the use of WS_VISI-
BLE
return S_OK;
}
//_____
// Name: MsgProc()
// Desc: Message handling function
//_____
HRESULT CApp::MsgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_CLOSE:
```

```
        case WM_DESTROY:
            PostQuitMessage(WM_QUIT);
            break;
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}

//_____
// Name: Render()
// Desc: Render the app
//_____
VOID CApp::Render()
{
    // do anything
}

//_____
// Name: Pause()
// Desc: Sets bActive
//_____
VOID CApp::Pause(BOOL bPause)
{
    m_bActive = bPause? 0 : 1;
}

//_____
// Name: Run()
// Desc: Message Loop and render method
//_____
WPARAM CApp::Run()
{
    BOOL bGotMsg;
    MSG msg;
    PeekMessage (&msg, NULL, 0, 0, PM_REMOVE);
    /* Step 5: Create the message loop. */
    while (WM_QUIT != msg.message)
    {
        if( m_bActive)
            bGotMsg = PeekMessage (&msg, NULL, 0, 0, PM_REMOVE);
        else
            bGotMsg = GetMessage (&msg, NULL, 0, 0);
        if(bGotMsg)
        {
            TranslateMessage(&msg); /* allow use of key-
board */
```

```
Windows */
    }
    else
    {
        if(m_bActive)
            Render();
    }
}
return msg.wParam;
}
```

As you might notice, to encapsulate the `m_bActive` variable, I used `Pause()`. But now let's start with the constructor.

## CONSTRUCTOR

The constructor ensures that objects always contain valid values. A constructor is a special initialization function that is called automatically whenever an instance of your class is declared. A constructor prevents errors resulting from the use of uninitialized objects and checks whether a value is out of range. It must have the same name as the class itself. You aren't required to use constructors, but it is generally a good idea.

I initialized all members `g_pCApp`, `m_bActive`, `m_strWindowTitle`, `m_dwWidth`, and `m_dwHeight` in `CApp()`. Constructors create objects and do not return values.

There could be more than one constructor declared for a class if each constructor has a different parameter list. These are called overloaded constructors. This is useful if you want to initialize your object in more than one way. I will show you how to overload constructors in a few seconds.

## DESTRUCTOR

The counterpart to the constructor is the destructor. Its purpose is to perform any cleanup work necessary before an object is destroyed. The destructor's name is the class name with a tilde (~) as a prefix. It is a member function that is called automatically when a class object goes out of scope.

What is scope? I will give you a simple example:

```
int number = 123;
VOID ShowMeANumber()
{
    int number = 456;
    Showme (::number);
    Showme (number);
```

}

You will see

```
123  
456
```

as an output. There is a global variable called *number* and a local variable called *number*. In both cases, C will use the local variable. You might use the scope operator :: to get the global variable in C++. So scope is the range of the class.

The scope operator is also used to access member functions. So it appears between the name of the class and the name of a function in a C++ program:

```
VOID CApp::Render()
```

This means that the specified function is a member of the specified class.

Now back to destructors. I have not used a destructor here, because they are usually required for more complicated classes, where they are used to release dynamically allocated resources. A destructor cannot be overloaded. It takes no parameters and has no return value.

### THIS POINTER

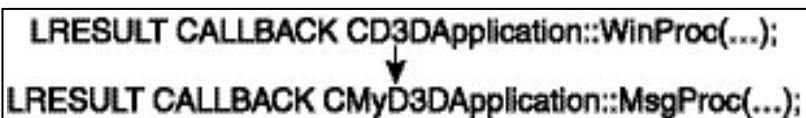
There's one interesting variable in the constructor of win skeleton++: g\_pCApp. It is used as a so-called this pointer. When you call a member function for an object, the compiler assigns the address of the object to the this pointer and then calls the function. Every time a member function accesses one of the class's data members, it is implicitly using the this pointer. Let's build an example:

```
void CApp::Render (BOOL active)  
{  
    BOOL bPause;  
    bPause = active;  
}  
  
myApp.Render(FALSE);
```

This is equivalent to

```
void CApp::Render (BOOL active)  
{  
    BOOL bPause;  
    this->bPause = active;  
    // or (*this).pause = active  
}  
  
myApp.Render(FALSE);
```

Both statements in `Render()` are equivalent. Why do we need such a `this` pointer? The answer lies in the `WinProc()`. We would like to call the `MsgProc()` in any derived class. Because we can't know the name of the class used in the future, we have to use the `this` pointer. The derived class `MsgProc()` will override the `MsgProc()` in the base class. In our example, the `MsgProc()` is located in the `MyFrame` class. So the calling order goes like this:



**Figure B.2:** Message procedures

This is also called static binding, a possible way to avoid virtual functions that you will smell in a few seconds.

You might also use a `this` pointer to avoid self-assignment. Let's build an example:

```
void String::operator= (const String &other)
{
    if (&other == this)
        return;
    ...
}
```

This method exits without doing anything if a self-assignment is being attempted.

To summarize this piece of code: To encapsulate all of the data, the variables are labeled *private*: so every variable has to be accessed by a member function. To be sure that every variable has a valid value, we use a constructor that initializes them. The `this` pointer is used to access functions in derived classes.

This brings us to the next fundamental principle of object-oriented programming.

## CLASS HIERARCHIES AND INHERITANCE

One standalone feature of object-oriented programming is class hierarchy. In C++ you can define one class as a subtype or special category of another class by deriving them all from one base class. This is not possible with C, which treats all types as completely independent of one another.

The benefits of defining a class hierarchy are code sharing and interface sharing. The derived class shares the code of the base class or/and interface of the base class.

Our example program uses both.

## INHERITING CODE

You might reduce redundant code by deriving a class from a base class. An example for inheriting code is a program that maintains a database of all the aliens in a game.

It would not be a good idea to use a C structure called *alien*, because each type of alien requires slightly different information. You wouldn't define a structure type for each type of alien; it would be a waste of space to include all possible fields in one structure. On the other side, unions would help, but the resulting C program will be difficult to read and difficult to maintain.

It's easier to do this in C++ with a base class and derived classes (pseudocode):

```
class CAlien
{
public:
    CAlien();
    VOID Attack(HUMAN human);
    VOID RunTo(WAYPOINT waypoint);
    VOID SetHealth(DWORD dwHealth);
    DWORD GetHealth();

private:
    char cName[32];
    DWORD dwHealth;
}
```

For simplicity, this *CAlien* class stores only a few values; in real life, there might be a lot more. Let's assume that there are different aliens that can fly, only crouch, only swim, or teleport. Some of them bite, fire weapons on humans, or do magic.

```
class CMagician : public CAlien
{
public:
    Magician();
    VOID SetGlow(BOOL bGlow);

private:
    BOOL bGlow;
}
```

or

```
class CFrankenstein : class CAlien
{
public:
    Frankenstein();
    VOID StruckByLightning(BOOL bStruck);
```

```

private:
    BOOL bStruck;
}

```

CMagician and CFrankenstein are derived classes from the base class CAlien. To declare a derived class, you follow its name with a colon and the keyword *public*, followed by the name of its base class.

In the declaration of the derived class, you declare the members that are specific to it by describing the additional qualities that distinguish it from the base class. Each instance of CFrankenstein contains all of the CAliens data members in addition to its own.

### NOTE

With the keyword ***private***, the public and protected members of the base class are private members of the derived class. So they are only accessible to the derived class, but not to anyone using the derived class.

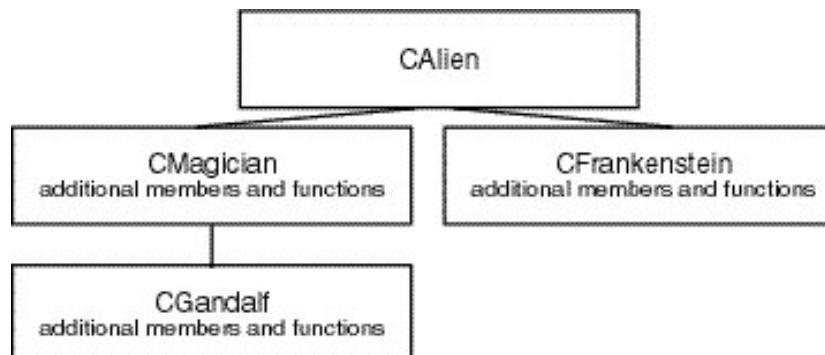


Figure B.3: Class hierarchy

The member functions of a derived class do not have access to the private members of its base class. So the derived class has to use the base class's public interface to access the private members of its base class.

If you use different magicians with, for example, different levels and kinds of magic, you might derive special magicians from CMagician:

```

class CGandalf : CMagician
{
public:
    CGandalf();
    VOID TellWisdoms(HUMAN bHuman);
    SetWeakness(WORD wWeakness);
private:
    char cWisdoms[32][128];
    WORD wWeakness;
}

```

This means that CMagician is both a derived and a base class. It derives from the CAlien class and serves as the base for the CGandalf class. You can define as many levels of inheritance as you want. In our win skeleton++ example, we only derive CMyFrame from CApp in Fake.cpp:

```
#define STRICT
#include <windows.h>
#include <tchar.h>
#include "App.h"
//-----
// Name: class App
// Desc: Main class to run this application. Most functionality is inherited
//       from the App base class.
//-----
class CMyFrame : public CApp
{
public:
    LRESULT MsgProc (HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
    CMyFrame(); // constructor
};

//-----
// Name: WinMain()
// Desc: Entry point to the program. Initializes everything, and goes into a
//       message-processing loop. Idle time is used to render the scene.
//-----
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT )
{
    CMyFrame d3dApp;
    if( FAILED( d3dApp.Create( hInst ) ) )
        return 0;
    return d3dApp.Run();
}

//-----
// Name: CMyD3DApplication()
// Desc: Constructor
//-----
CMyFrame::CMyFrame()
{
    // Override base class members
    m_strWindowTitle = _T("Fake 4 ++");
    m_dwWidth = 400;
    m_dwHeight = 300;
}
```

```
//-----  
// Name: MsgProc()  
// Desc: message procedure  
//-----  
HRESULT CMYFrame::MsgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)  
{  
    if (message == WM_KEYDOWN)  
    {  
        switch (wParam)  
        {  
            case VK_ESCAPE:  
                PostQuitMessage(WM_QUIT);  
                break;  
            case VK_F1:  
                {  
                    Pause(TRUE);  
                    MessageBox( hWnd, "Here comes your help  
text",  
                                "Help for FAKE 4++", MB_ICON-  
QUESTION|MB_OK |  
                                MB_SYSTEMMODAL );  
                    Pause(FALSE);  
                }  
                break;  
        }  
    }  
    return CApp::MsgProc( hWnd, message, wParam, lParam );  
}
```

CFrame uses its own MsgProc() and its own constructor. Three base class members are overwritten there.

Using a class hierarchy designed for code sharing has most of its code in the base classes. This way the code can be reused by many classes. The derived classes represent specialized or extended versions of the base class.

#### INHERITING AN INTERFACE

You might inherit just the names of the base class's member functions, not the code. This is called inheriting an interface. What's the idea behind that?

Think of the CAlien class used before, which is used to inherit all of those different alien classes. You might be able to use the derived alien classes as generic CAlien objects when all of the derived classes share its interface.

That's all for now with classes and inheritance. I don't want to dig deeper into this kind of stuff. There are a few things I left out; try to investigate time to learn them with the resources I show you in the Additional Resources section.

## VIRTUAL FUNCTIONS

When you call a virtual function through a pointer to a base class, the derived class's version of the function is executed. This is the opposite behavior of ordinary member functions.

A virtual function is declared by placing the keyword *virtual* before the declaration of the member function in the base class. All of the public interface methods in d3dapp.h of the Common files framework are virtual functions.

```
virtual HRESULT ConfirmDevice(D3DCAPS8*, DWORD, D3DFORMAT) { return S_OK; }
virtual HRESULT OneTimeSceneInit()
{ return S_OK; }
virtual HRESULT InitDeviceObjects()
{ return S_OK; }
virtual HRESULT RestoreDeviceObjects()
{ return S_OK; }
virtual HRESULT FrameMove()
{ return S_OK; }
virtual HRESULT Render()
{ return S_OK; }
virtual HRESULT InvalidateDeviceObjects()
{ return S_OK; }
virtual HRESULT DeleteDeviceObjects()
{ return S_OK; }
virtual HRESULT FinalCleanup()
{ return S_OK; }
```

They are called from the derived class in your main file. This could be, for example, .md3viewer.cpp:

```
HRESULT CMyD3DApplication::RestoreDeviceObjects()
{
    return S_OK;
}
```

This method is called in d3dapp.cpp by the base class:

```
HRESULT CD3DApplication::Resize3DEnvironment()
{
    ...
}
```

```
// Initialize the app's device-dependent objects
hr = RestoreDeviceObjects();
if( FAILED(hr) )
    return hr;
...
return S_OK;
}
```

So calling `RestoreDeviceObjects()` in the base class method `Resize3DEnvironment()` leads to a call to the derived class function `RestoreDeviceObjects()` in `.md3viewer.cpp`.

The `virtual` keyword is only necessary in the declaration in the base class.

Using virtual functions is also called *dynamic binding*, because the compiler must evaluate the statement at run time, when it can tell what type of object it gets.

A game programmer will now ask a typical question: “Well that’s nice, but how much overhead is involved?” Calling a virtual function takes only slightly longer than calling a normal function.

A trick to reduce that overhead is the use of static binding. We have done that with the `MsgProc()` function.

```
static CApp* g_pCApp = NULL;
CApp::CApp()
{
    g_pCApp = this;
...
}
LRESULT CALLBACK WndProc (HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    return g_pCApp->MsgProc (hWnd, uMsg, wParam, lParam);
}
```

To summarize the above paragraphs: Virtual functions are an elegant way to call functions in the derived class from within base class. The dynamic binding used comes with a small overhead. The inelegant but efficient way is to use static binding with no overhead.

## POLYMORPHISM

The ability to call member functions for an object without specifying the object’s exact type is known as *polymorphism*. This means that a single statement can invoke many different functions. Let’s build an example: You are interested in checking the health of all aliens so that you are able to compare their health with the health of all humans.

In C you have to use a switch statement to find the exact type of alien. This might look like:

```
DWORD ComputeHealth (struct ALIEN *alien)
{
    switch (alien->type)
    {
        case Magician:
            return alien.dwhealth;
            break;

        case Frankenstein:
            return alien.dwhealth;
            break;

        ...
    };
}
```

In C++ you are able to call the following function:

```
DWORD ComputeHealth (CALIEN *alien)
{
    return alien->GetHealth();
}
```

The C++ version of ComputeHealth() calls the appropriate function automatically, without requiring you to examine the type of object that alien points to. There is only a tiny amount of overhead, as described in the section on virtual functions, but no switch statement is needed.

## INLINE FUNCTIONS

A performance optimization path for game programmers is the use of inline functions. This keyword causes a new copy of the function to be inserted in each place it is called. If you call an inline function from 21 places in your program, the compiler will insert 21 copies of that function into your .exe file.

Inline eliminates the overhead of calling a function, so your program runs faster, but having multiple copies of a function can make your program larger. It is similar to macros declared with the #define directive. The difference is that inline functions are recognized by the compiler, and macros are implemented by a simple text substitution. But inline functions are more powerful. The compiler performs type-checking, and an inline function behaves just as an ordinary function, without any side effects that macros might have. An example could be:

```
#define MAX (A, B) ((A) > (B)? (A) : (B))
inline int max (int a, int b)
{
    if (a > b)
```

```
    return a;  
return b;  
}
```

This is one of our favorite C++ features, isn't it? Well, experience has shown that all the things you like need more work and more attention. This is also the case with the `inline` keyword.

Inlining only happens if the compiler's cost/benefit analysis shows it to be profitable. A `__forceinline` keyword overrides the cost/benefit analysis and relies on the judgment of the programmer instead. But this might lead to performance losses if it's not properly used.

You cannot force the compiler to inline a function with `__forceinline` when conditions other than cost/benefit analysis prevent it. You cannot inline a function if:

- The function or its caller is compiled with `/Ob0` (the default option for debug builds).
- The function and the caller use different types of exception handling (C++ exception handling in one, structured exception handling in the other).
- The function has a variable argument list.
- The function uses inline assembly and is not compiled with `/Og`, `/Ox`, `/O1`, or `/O2`.
- The function returns an unwindable object by value and is not compiled with `/GX`, `/EHs`, or `/EHa`.
- The function receives a copy-constructed object passed by value when compiled with `/GX`, `/EHs`, or `/EHa`.
- The function is recursive and is not accompanied by `#pragma(inline_recursion, on)`. With the pragma, recursive functions can be inlined to a default depth of eight calls. To change the inlining depth, use `#pragma(inline_depth, n)`.

Check out your Visual C/C++ Compiler Reference for the meaning of the compiler directives. If the compiler cannot inline a function declared `__forceinline`, it will generate a Level 1 warning (4714).

This was the first one of the language enhancements of C++. Let's take a look at a few others.

## C++ ENHANCEMENTS TO C

C++ introduces some simple enhancements and improvements to C. I would like to explain the following C++ enhancements in this section:

- Default function arguments
- More flexible placement of variable declarations
- The `const` keyword
- Enumerations
- Function overloading and operator overloading

## DEFAULT FUNCTION ARGUMENTS

A C++ function prototype can list default values for some of the parameters.

```
VOID Pause(BOOL TRUE);
```

You might declare this in the header file. If you provide your own arguments, the compiler will use them instead of the declared defaults.

If you use a more complex example it might look like:

```
VOID DoSomething (DWORD w = 2; FLOAT f = 1.5);
```

You might call that in the following ways:

```
DoSomething ( , 2.5);
```

In this example, the first value w is 2 and the second value f is 2.5. Another way is

```
DoSomething (3);
```

The first value w is 3, and the second value f is 1.5. Be careful: A syntax like this is error-prone and makes reading and writing function calls more difficult.

## PLACEMENT OF VARIABLE DECLARATIONS

There is a big improvement in C++, which I guess you already know. C requires you to declare variables at the beginning of a block. C++ gives you the ability to declare variables anywhere in the code. A pseudocode function with C would look like:

```
VOID Myfunction ()  
{  
    int z;  
    AnotherFunction (d);  
    doSomethingWith(z);  
}
```

The same function with C++:

```
VOID Myfunction ()  
{  
    AnotherFunction (d);  
    int z;  
    doSomethingWith(z);  
}
```

That's a big improvement, because it is much easier to read the code when the variable and function are nearer to each other. Another nice expression that is often used in C++ is

```
for (int z = 0; z < 5; z++)  
{  
    // do something  
}
```

But it is not possible to use the following statements:

```
if (int z == 0)  
;  
while (int j == 0)
```

Besides, they are meaningless :-).

### CONST VARIABLE

As a rule of thumb, you can assume that wherever you can use a constant expression or would define a #define expression, you can use a const variable.

There is a difference between the const qualifier used by C and the one used in C++. In C you can initialize it and use it in place of its value in your program. In C++ you can use consts in many different ways with pointers, references, and functions.

Most C programs define constant values using the #define preprocessor directive.

```
#define MAX_HEIGHT 800
```

The drawback is that the #define directive does not generate type-safe constants because it is just a pre-processor directive, not a compiler statement. When you generate a constant using #define, your constant has no data type, so your compiler has no way to ensure that operations on the constant are carried out properly—at design time, at run time, or even at debugging time.

In contrast, when you define a const in a C++ program, you execute a real compiler statement, such as this one:

```
const int MAX_HEIGHT 800
```

You can also use const in pointer declarations:

```
char *const ptr = cString;
```

You can change the value to which *ptr* points:

```
*ptr = 'a';
```

But you can't change the const value:

```
ptr = cString2;
```

It is another case if the pointer points to a constant:

```
Const char *ptr = cString;
ptr = cString2;
*ptr = 'a';
```

To change the pointer in the second row is OK, but changing the value will give you an error message. You might use the const qualifier to prevent changing a parameter in a function:

```
int readonly (const struct NODE *nodeptr);
```

As you have seen, const is a powerful qualifier to make your programs more type-safe. So use it often.

## ENUMERATION

An *enumeration* is an integral data type that defines a list of named constants. Each element of an enumeration has an integer value that, by default, is one greater than the value of the previous element. The first element has the value 0, unless you specify another value. The following example demonstrates how a C++ program can reference an enumeration:

```
enum color {yellow, orange, red, green }; // values 0, 1, 2, 4
enum day {Monday, Tuesday, Wednesday, Thursday = 8, Friday}; // values 0, 1, 2, 8, 9
```

You are able to convert an enumeration into an integer, but you can not reverse conversion unless you use a cast. Casting an integer into an enumeration is not safe. If the integer is outside the range of the enumeration, or if the enumeration contains duplicate values, the result of the cast is undefined.

## FUNCTION OVERLOADING AND OPERATOR OVERLOADING

C++ uses two kinds of overloading: function overloading and operator overloading. Both of these are major—and beneficial—features of the C++ language.

### FUNCTION OVERLOADING

To implement, you use two or more functions that share the same name but have different argument lists. The compiler decides which version to call by using argument matching. It compares the number of arguments and the types of arguments that are passed to the function.

Function overloading is often used in C++ because it imposes almost no run-time penalty and requires practically no overhead.

A special case of function overloading is constructor overloading. You find the following passage in d3dvec.inl:

```
inline  
_D3DVECTOR::_D3DVECTOR(D3DVALUE f)  
{  
    x = y = z = f;  
}  
inline  
_D3DVECTOR::_D3DVECTOR(D3DVALUE _x, D3DVALUE _y, D3DVALUE _z)  
{  
    x = _x; y = _y; z = _z;  
}  
inline  
_D3DVECTOR::_D3DVECTOR(const D3DVALUE f[3])  
{  
    x = f[0]; y = f[1]; z = f[2];  
}
```

Typical function overloading is used in the D3DXVECTOR3 initialization in d3dmath.inl.

```
D3DXINLINE  
D3DXVECTOR3::D3DXVECTOR3( CONST FLOAT *pf )  
{  
#ifdef D3DX_DEBUG  
    if(!pf)  
        return;  
#endif  
    x = pf[0];  
    y = pf[1];  
    z = pf[2];  
}  
D3DXINLINE  
D3DXVECTOR3::D3DXVECTOR3( CONST D3DVECTOR& v )  
{  
    x = v.x;  
    y = v.y;  
    z = v.z;  
}  
D3DXINLINE  
D3DXVECTOR3::D3DXVECTOR3( FLOAT fx, FLOAT fy, FLOAT fz )  
{
```

```
x = fx;  
y = fy;  
z = fz;  
}
```

These overloaded functions pursue the same target by using different parameters. Three variables, x, y, and z, have to be filled with a value.

Complete vector/math libraries are built using function overloading and operator overloading. By using function overloading, you can give the same name to member functions that perform different, but similar, operations. You can even give the same name to entire groups of functions.

#### OPERATOR OVERLOADING

With operator overloading, you can customize operators, such as the addition operator (+), the subtraction operator (-), the assignment operator (=), the increment and decrement operators (++ and --), and many more, to make the operators behave differently when they are used with objects of different classes. Such a self-defined operator is a dressed-up function.

To overload an operator in C++, you must define an operator overloading function (usually a member function). A function that overloads an operator always contains the keyword *operator*.

The syntax for operator overloading is

```
ReturnType operator @ (arguments) {code} // global function  
ReturnType classname::operator @ (arguments) {code} // class function
```

The @ sign is a replacement for the operator. The syntax

```
z = x @ y
```

is replaced by

```
z = operator @ (x, y)
```

or by

```
z = x.operator @(y);
```

A real world example might look like this:

```
D3DXINLINE D3DXVECTOR3&  
D3DXVECTOR3::operator += ( CONST D3DXVECTOR3& v )  
{  
    x += v.x;  
    y += v.y;
```

```
    z += v.z;
    return *this;
}
```

This one adds the D3DXVECTOR3 v structure to the D3DXVECTOR3 that stands on the left of the `+=` operator.

```
z += v;
```

The operator `+=` produces a new D3DXVECTOR3 structure, which is returned. This temporary structure is destroyed as soon as it is no longer needed.

Another nice example is

```
D3DXINLINE D3DXVECTOR3
operator * ( FLOAT f, CONST struct D3DXVECTOR3& v )
{
    return D3DXVECTOR3(f * v.x, f * v.y, f * v.z);
}
```

The syntax prototype of this one looks like:

```
z = operator * (f, v)
```

The return value `D3DXVECTOR3(f * v.x, f * v.y, f * v.z)` is also a temporary structure. We use here the so-called returned value optimization. The compiler builds the object directly into the location of the outside return value.

OK ... I think you got it. Take a look into the d3dxmath.inl file to see a lot more examples and play around with them.

Let's end this small C++ primer now. I tried to concentrate on the most important C++ features for game programmers. A lot of things were unsaid, and the use of references and multi-inheritance especially should be topics that you might take a look at. So ... keep on learning.

## ADDITIONAL RESOURCES

There is a free C++ book in HTML/PDF/PalmPilot format online at [www.bruceeckel.com/ThinkingInCPP2e.html](http://www.bruceeckel.com/ThinkingInCPP2e.html). Watch out for the Palm version.

Other C++ tutorials online are at [www.intap.net/~drw/cpp/index.htm](http://www.intap.net/~drw/cpp/index.htm), [www.cplusplus.com/doc/tutorial/](http://www.cplusplus.com/doc/tutorial/), and [devcentral.iftech.com/learning/tutorials/subcpp.asp](http://devcentral.iftech.com/learning/tutorials/subcpp.asp).

# **APPENDIX C**

## **THE COMMON FILES FRAMEWORK**

The Common files framework is one of the big pluses that a beginning-to-intermediate game programmer will get with the DirectX 8.0 SDK. It consists of tested code that you can trust. This code will handle most of the tricky situations that might come up when your first game prototype runs on different hardware. DirectX is supported on all Windows hardware, but . . . there are differences. If you read a little bit in the online discussion groups, you will find a lot of programmers who get into trouble with different graphic cards in different situations. So the Common files framework lightens the burden of getting fast and reliable code for the basic initialization and start-up of your game. You can find more arguments in Chapter 5.

If you decide later to build up a framework that suits the needs of your game better—for example, you need a full-screen-only window—the Common files framework will be a good starting point. Taking a look at the code of the Common files framework will help in any situation. The main file is d3dapp.cpp, which handles all of the basic stuff. Its header file gives you the ability to look at the class definition, which is pretty clear and straightforward. So let's dive into the d3dapp.h. It is located in your

x:\mssdk\samples\Multimedia\Common\source

and

x:\mssdk\samples\Multimedia\Common\include

directory.

```
class CD3Dapplication
{
protected:
    // Internal variables for the state of the app
    D3DAdapterInfo    m_Adapters[10];
    DWORD             m_dwNumAdapters;
    DWORD             m_dwAdapter;
    BOOL              m_bWindowed;
    BOOL              m_bActive;
    BOOL              m_bReady;
    // Internal variables used for timing
    BOOL              m_bFrameMoving;
    BOOL              m_bSingleStep;
    // Internal error handling function
    HRESULT DisplayErrorMsg( HRESULT hr, DWORD dwType );
    // Internal functions to manage and render the 3-D scene
```

```
HRESULT BuildDeviceList();
BOOL    FindDepthStencilFormat( UINT iAdapter, D3DDEVTYPE DeviceType,
                               D3DFORMAT TargetFormat, D3DFORMAT* pDepthStencilFormat );
HRESULT Initialize3DEnvironment();
HRESULT Resize3DEnvironment();
HRESULT ToggleFullscreen();
HRESULT ForceWindowed();
HRESULT UserSelectNewDevice();
VOID    Cleanup3DEnvironment();
HRESULT Render3DEnvironment();
virtual HRESULT AdjustWindowForChange();
static INT_PTR CALLBACK SelectDeviceProc( HWND hDlg, UINT msg,
                                         WPARAM wParam, LPARAM lParam );

protected:
    // Main objects used for creating and rendering the 3-D scene
    D3DPRESENT_PARAMETERS m_d3dpp;      // Parameters for CreateDevice/Reset
    HWND                 m_hWnd;          // The main app window
    HWND                 m_hWndFocus;     // The D3D focus window
    (usually same as m_hWnd)
    LPDIRECT3D8          m_pD3D;          // The main D3D object
    LPDIRECT3DDEVICE8    m_pd3dDevice;    // The D3D rendering device
    D3DCAPS8             m_d3dCaps;       // Caps for the device
    D3DSURFACE_DESC      m_d3dsdBackBuffer;
    buffer
    DWORD                m_dwCreateFlags; // Indicate sw or hw vertex
    processing
    DWORD                m_dwWindowSize; // Saved window style for
    mode switches
    mode switches
    RECT                 m_rcWindowBounds; // Saved window bounds for
    mode switches
    RECT                 m_rcWindowClient; // Saved client area size
    for mode switches
    // Variables for timing
    FLOAT               m_fTime;          // Current time in seconds
    FLOAT               m_fElapsedTime;   // Time elapsed since last
    frame
    FLOAT               m_fFPS;           // Instantaneous frame rate
    TCHAR               m_strDeviceStats[90]; // String to hold D3D device
    stats
    TCHAR               m_strFrameStats[40]; // String to hold frame
    stats
```

```

// Overridable variables for the app
TCHAR*           m_strWindowTitle;                      // Title for the app's window
BOOL             m_bUseDepthBuffer;                     // Whether to autocreate depthbuffer
DWORD            m_dwMinDepthBits;                      // Minimum number of bits needed in depth buffer
DWORD            m_dwMinStencilBits;                    // Minimum number of bits needed in stencil buffer
DWORD            m_dwCreationWidth;                   // Width used to create window
DWORD            m_dwCreationHeight;                  // Height used to create window
BOOL             m_bShowCursorWhenFullscreen;        // Whether to show cursor when fullscreen

// Overridable functions for the 3-D scene created by the app
virtual HRESULT ConfirmDevice(D3DCAPS8*,DWORD,D3DFORMAT) { return S_OK; }
virtual HRESULT OneTimeSceneInit() { return S_OK; }
virtual HRESULT InitDeviceObjects() { return S_OK; }
virtual HRESULT RestoreDeviceObjects() { return S_OK; }
virtual HRESULT FrameMove() { return S_OK; }
virtual HRESULT Render() { return S_OK; }
virtual HRESULT InvalidateDeviceObjects() { return S_OK; }
virtual HRESULT DeleteDeviceObjects() { return S_OK; }
virtual HRESULT FinalCleanup() { return S_OK; }

public:
    // Functions to create, run, pause, and clean up the application
    virtual HRESULT Create( HINSTANCE hInstance );
    virtual INT     Run();
    virtual LRESULT MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam );
    virtual VOID    Pause( BOOL bPause );
    // Internal constructor
    CD3DApplication();

};

#endif

```

Let's start with the target of the whole Common files framework, to export the following public interface functions to the application:

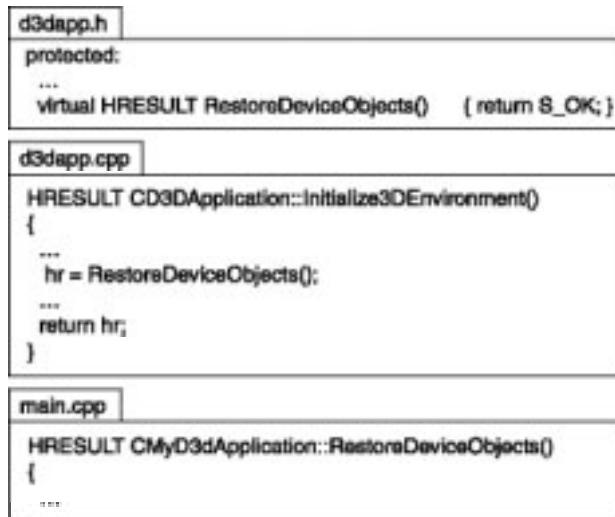
```

virtual HRESULT ConfirmDevice(D3DCAPS8*,DWORD,D3DFORMAT) { return S_OK; }
virtual HRESULT OneTimeSceneInit()
{ return S_OK; }

```

```
virtual HRESULT InitDeviceObjects()
{ return S_OK; }
virtual HRESULT RestoreDeviceObjects()
{ return S_OK; }
virtual HRESULT FrameMove()
{ return S_OK; }
virtual HRESULT Render()
{ return S_OK; }
virtual HRESULT InvalidateDeviceObjects()
{ return S_OK; }
virtual HRESULT DeleteDeviceObjects()
{ return S_OK; }
virtual HRESULT FinalCleanup()
{ return S_OK; }
```

These are the functions that appear in your main game file that is called, for example, main.cpp. They are protected virtual functions that are called from inside the d3dapp.cpp file by accessing the functions with the same name in the derived class in your main.cpp. I try to show that graphically:



**Figure C.1:** How the public interface functions are called in `CD3DApplication` class

As you have learned in the C++ primer, virtual functions are called through a pointer in the base class, here located in d3dapp.cpp, the derived class's version. The derived class is in main.cpp and it is called CMyD3DApplication.

```
class CMyD3DApplication : public CD3DApplication
```

The derived class can use these functions with their special purpose or can resist using a few of them. The basic idea is to lighten the main source files from the usual start-up and initialization code. You will find a description of the purpose of the functions in Chapter 5.

The other functions used in d3dapp.h are used to create a window instance, to run the message loop and a message procedure, and to pause the message loop.

```
public:  
    // Functions to create, run, pause, and clean up the application  
    virtual HRESULT Create( HINSTANCE hInstance );  
    virtual INT Run();  
    virtual LRESULT MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam );  
    virtual VOID Pause( BOOL bPause );
```

You know them from the win skeleton++ example. Create() creates the window with the main function CreateWindow(). Run() starts the message pump. And MsgProc() is a statically linked function via the this pointer. The Pause() function stops calling the rendering methods. This functions are the public interface of the Common files framework. They are typically located in the DirectX examples in the main file:

```
...  
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT )  
{  
    CMyD3DApplication d3dApp;  
    if( FAILED( d3dApp.Create( hInst ) ) )  
        return 0;  
    return d3dApp.Run();  
}  
...  
LRESULT CMyD3DApplication::MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam,  
                                    LPARAM lParam )  
{  
    if( uMsg == WM_COMMAND )  
    {  
        // Handle the open file dialog  
        if( LOWORD(wParam) == IDM_OPENFILE )  
        {  
            Pause( TRUE );  
            LoadFile(cmd3Mode11);  
            Pause( FALSE );  
        }  
    }  
    return CD3DApplication::MsgProc( hWnd, uMsg, wParam, lParam );  
}
```

We will now work through these public interface calls step by step . . . it is a nice idea to take a cup of coffee or another stimulant like tea (my favorite caffeine source) now :-).

## CREATE()

The first method a game will call with the Common files framework is the Create() method with the code shown above in the WinMain() function. Let's have a look at the source first. I have added a few comments to make things clearer:

```
HRESULT CD3DApplication::Create( HINSTANCE hInstance )
{
    HRESULT hr;
    //_____
    // Step 1: Create the Direct3D object
    //_____
    m_pD3D = Direct3DCreate8( D3D_SDK_VERSION );
    if( m_pD3D == NULL )
        return DisplayErrorMsg( D3DAPPERR_NODIRECT3D, MSGERR_APPMUSTEXIT );
    //_____
    // Step 2: Build a list of Direct3D adapters, modes and devices
    //_____
    // The ConfirmDevice() callback is used to confirm that only devices that
    // meet the app's requirements are considered.
    if( FAILED( hr = BuildDeviceList() ) )
    {
        SAFE_RELEASE( m_pD3D );
        return DisplayErrorMsg( hr, MSGERR_APPMUSTEXIT );
    }
    //_____
    // Step 3: Create window
    //_____
    // Unless a substitute hWnd has been specified, create a window to
    // render into
    if( m_hWnd == NULL )
    {
        // Register the windows class
        WNDCLASS wndClass = { 0, WndProc, 0, 0, hInstance,
            LoadIcon( hInstance, MAKEINTRESOURCE(IDI_MAIN_ICON)
        ),
            LoadCursor( NULL, IDC_ARROW ),
            (HBRUSH)GetStockObject(WHITE_BRUSH),
            NULL, _T("D3D Window") };
    }
}
```

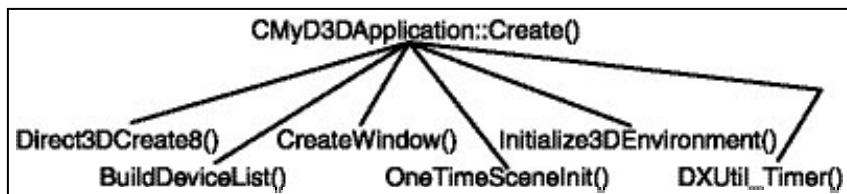
```
RegisterClass( &wndClass );
// Set the window's initial style
m_dwWindowStyle = WS_POPUP|WS_CAPTION|WS_SYSMENU|WS_THICKFRAME|
                  WS_MINIMIZEBOX|WS_VISIBLE;
// Set the window's initial width
RECT rc;
SetRect( &rc, 0, 0, m_dwCreationWidth, m_dwCreationHeight );
AdjustWindowRect( &rc, m_dwWindowStyle, TRUE );
// Create the render window
m_hWnd = CreateWindow( _T("D3D Window"), m_strWindowTitle, m_dwWindowStyle,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      (rc.right-rc.left), (rc.bottom-rc.top), 0L,
                      LoadMenu( hInstance, MAKEINTRESOURCE(IDR_MENU) ),
                      hInstance, 0L );
}
// The focus window can be a specified to be a different window than the
// device window. If not, use the device window as the focus window.
if( m_hWndFocus == NULL )
    m_hWndFocus = m_hWnd;
// Save window properties
m_dwWindowStyle = GetWindowLong( m_hWnd, GWL_STYLE );
GetWindowRect( m_hWnd, &m_rcWindowBounds );
GetClientRect( m_hWnd, &m_rcWindowClient );
//_____
// Step 4: Initialize geometry data of your game
//         with OneTimeSceneInit()
//_____
// Initialize the app's custom scene stuff
if( FAILED( hr = OneTimeSceneInit() ) )
{
    SAFE_RELEASE( m_pD3D );
    return DisplayErrorMsg( hr, MSGERR_APPMUSTEXIT );
}
//_____
// Step 5: Initialize the 3-D environment
//_____
if( FAILED( hr = Initialize3DEnvironment() ) )
{
    SAFE_RELEASE( m_pD3D );
    return DisplayErrorMsg( hr, MSGERR_APPMUSTEXIT );
}
```

```

//_____
// Step 6: Start a timer
//_____
DXUtil_Timer( TIMER_START );
// The app is ready to go
m_bReady = TRUE;
return S_OK;
}

```

This piece of code makes all of the initializing and setup work. It uses the following functions:



**Figure C.2:** The functions that are called by *CD3DApplication::Create()*

I commented on the following steps in the *Create()* function:

- Create the Direct3D object with *Direct3DCreate8()*.
- Search for the proper device driver with the help of *BuildDeviceList()*.
- Create the window with *CreateWindow()*.
- Initialize the geometry data of your game with *OneTimeSceneInit()*.
- Initialize the 3-D environment with *Initialize3DEnvironment()*.
- Start a timer with *DXUtil\_Timer()*.

## STEP 1: CREATE THE DIRECT3D OBJECT WITH *Direct3DCreate8()*

*Direct3DCreate8()* is the starting point for all of your Direct3D activities. Its prototype looks like this:

```

IDirect3D8* Direct3DCreate8(
    UINT SDKVersion;
);

```

The only parameter is *D3D\_SDK\_VERSION*. This identifier ensures that an application was built against the correct header files. If the version does not match, *Direct3DCreate8()* will fail. This function samples the current set of active display adapters. The adapters are enumerated only once during the first call.

This function returns a pointer to a IDirect3D8 interface. After receiving this interface, Create() calls BuildDeviceList().

## STEP 2: SEARCH FOR THE PROPER DEVICE DRIVER WITH THE HELP OF BuildDeviceList()

BuildDeviceList() is a monster function, so we need small steps to understand it. Here is the source:

```
HRESULT CD3DApplication::BuildDeviceList()
{
    const DWORD dwNumDeviceTypes = 2;
    const TCHAR* strDeviceDescs[] = { _T("HAL"), _T("REF") };
    const D3DDEVTYPE DeviceTypes[] = { D3DDEVTYPE_HAL, D3DDEVTYPE_REF };
    BOOL bHALExists = FALSE;
    BOOL bHALIsWindowedCompatible = FALSE;
    BOOL bHALIsDesktopCompatible = FALSE;
    BOOL bHALIsSampleCompatible = FALSE;
    //_____
    // Step 1: The two main purposes of this for() loop are:
    //          - Fill the D3DModeInfo, D3DDeviceInfo and D3DAdapterInfo structures
    //          - pick a default 640x480x16 mode
    //_____
    // Loop through all the adapters on the system (usually, there's just one
    // unless more than one graphics card is present).
    for( UINT iAdapter = 0; iAdapter < m_pD3D->GetAdapterCount(); iAdapter++ )
    {
        // Fill in adapter info
        D3DAdapterInfo* pAdapter = &m_Adapters[m_dwNumAdapters];
        m_pD3D->GetAdapterIdentifier( iAdapter, 0, &pAdapter->d3dAdapterIdentifier );
        m_pD3D->GetAdapterDisplayMode( iAdapter, &pAdapter->d3ddmDesktop );
        pAdapter->dwNumDevices = 0;
        pAdapter->dwCurrentDevice = 0;
        // Enumerate all display modes on this adapter
        D3DDISPLAYMODE modes[100];
        D3DFORMAT formats[20];
        DWORD dwNumFormats = 0;
        DWORD dwNumModes = 0;
        DWORD dwNumAdapterModes = m_pD3D->GetAdapterModeCount( iAdapter );
        // Add the adapter's current desktop format to the list of formats
        formats[dwNumFormats++] = pAdapter->d3ddmDesktop.Format;
        for( UINT iMode = 0; iMode < dwNumAdapterModes; iMode++ )
```

```
{  
    // Get the display mode attributes  
    D3DDISPLAYMODE DisplayMode;  
    m_pD3D->EnumAdapterModes( iAdapter, iMode, &DisplayMode );  
    // Filter out low-resolution modes  
    if( DisplayMode.Width < 640 || DisplayMode.Height < 400 )  
        continue;  
    // Check if the mode already exists (to filter out refresh rates)  
    for( DWORD m=0L; m<dwNumModes; m++ )  
    {  
        if( ( modes[m].Width == DisplayMode.Width ) &&  
            ( modes[m].Height == DisplayMode.Height ) &&  
            ( modes[m].Format == DisplayMode.Format ) )  
            break;  
    }  
    // If we found a new mode, add it to the list of 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( DWORD 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;  
    }  
    // Sort the list of display modes (by format, then width, then height)  
    qsort( modes, dwNumModes, sizeof(D3DDISPLAYMODE), SortModesCallback );  
    // Add devices to adapter  
    for( UINT iDevice = 0; iDevice < dwNumDeviceTypes; iDevice++ )  
    {  
        // Fill in device info  
        D3DDeviceInfo* pDevice;
```

```
pDevice           = &pAdapter->devices[pAdapter->dwNumDevices];
pDevice->DeviceType   = DeviceTypes[iDevice];
m_pD3D->GetDeviceCaps( iAdapter, DeviceTypes[iDevice], &pDevice->d3dCaps );
pDevice->strDesc      = strDeviceDescs[iDevice];
pDevice->dwNumModes    = 0;
pDevice->dwCurrentMode = 0;
pDevice->bCanDoWindowed = FALSE;
pDevice->bWindowed       = FALSE;
pDevice->MultiSampleType = D3DMULTISAMPLE_NONE;
// Examine each format supported by the adapter to see if it will
// work with this device and meets the needs of the application.
BOOL bFormatConfirmed[20];
DWORD dwBehavior[20];
D3DFORMAT fmtDepthStencil[20];
for( DWORD f=0; f<dwNumFormats; f++ )
{
    bFormatConfirmed[f] = FALSE;
    fmtDepthStencil[f] = D3DFMT_UNKNOWN;
    // Skip formats that cannot be used as render targets on this
device
    if( FAILED( m_pD3D->CheckDeviceType( iAdapter, pDevice->DeviceType,
formats[f], formats[f],
FALSE ) ) )
        continue;
    if( pDevice->DeviceType == D3DDEVTYPE_HAL )
    {
        // This system has a HAL device
        bHALExists = TRUE;
        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;
            }
        }
    }
    // Confirm the device/format for HW vertex processing
```

```
if( pDevice->d3dCaps.DevCaps&D3DDEVCAPS_HWTRANSFORMANDLIGHT )
{
    if( pDevice->d3dCaps.DevCaps&D3DDEVCAPS_PUREDEVICE )
    {
        dwBehavior[f] = D3DCREATE_HARDWARE_VERTEXPROCESSING |
                        D3DCREATE_PUREDEVICE;
        if( SUCCEEDED( ConfirmDevice( &pDevice->d3dCaps,
dwBehavior[f],
                                            formats[f] ) ) )
            bFormatConfirmed[f] = TRUE;
    }
    if( FALSE == bFormatConfirmed[f] )
    {
        dwBehavior[f] = D3DCREATE_HARDWARE_VERTEXPROCESSING;
        if( SUCCEEDED( ConfirmDevice( &pDevice->d3dCaps,
dwBehavior[f],
                                            formats[f] ) ) )
            bFormatConfirmed[f] = TRUE;
    }
    if( FALSE == bFormatConfirmed[f] )
    {
        dwBehavior[f] = D3DCREATE_MIXED_VERTEXPROCESSING;
        if( SUCCEEDED( ConfirmDevice( &pDevice->d3dCaps,
dwBehavior[f],
                                            formats[f] ) ) )
            bFormatConfirmed[f] = TRUE;
    }
}
// Confirm the device/format for SW vertex processing
if( FALSE == bFormatConfirmed[f] )
{
    dwBehavior[f] = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
    if( SUCCEEDED( ConfirmDevice( &pDevice->d3dCaps, dwBehavior[f],
                                    formats[f] ) ) )
        bFormatConfirmed[f] = TRUE;
}
// Find a suitable depth/stencil buffer format for this device/format
if( bFormatConfirmed[f] && m_bUseDepthBuffer )
{
    if( !FindDepthStencilFormat( iAdapter, pDevice->DeviceType,
```

```
        formats[f], &fmtDepthStencil[f] ) )
    {
        bFormatConfirmed[f] = FALSE;
    }
}
//_____
// fill a D3DModeInfo structure
//_____
// Add all enumerated display modes with confirmed formats to the
// device's list of valid modes
for( DWORD m=0L; m<dwNumModes; m++ )
{
    for( DWORD 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;
            }
        }
    }
}
//_____
// Pick the 640x480x16 mode
//_____
// Select any 640x480 mode for default (but prefer a 16-bit mode)
```

```
for( m=0; m<pDevice->dwNumModes; m++ )
{
    if( pDevice->modes[m].Width==640 && pDevice->modes[m].Height==480 )
    {
        pDevice->dwCurrentMode = m;
        if( pDevice->modes[m].Format == D3DFMT_R5G6B5 || 
            pDevice->modes[m].Format == D3DFMT_X1R5G5B5 || 
            pDevice->modes[m].Format == D3DFMT_A1R5G5B5 )
        {
            break;
        }
    }
}
// Check if the device is compatible with the desktop display mode
// (which was added initially as formats[0])
if( bFormatConfirmed[0] && (pDevice->d3dCaps.Caps2 & D3DCAPS2_CANRENDER-
WINDOWED) )
{
    pDevice->bCanDoWindowed = TRUE;
    pDevice->bWindowed      = TRUE;
}
// If valid modes were found, keep this device
if( pDevice->dwNumModes > 0 )
    pAdapter->dwNumDevices++;

}
// If valid devices were found, keep this adapter
if( pAdapter->dwNumDevices > 0 )
    m_dwNumAdapters++;

}
// Return an error if no compatible devices were found
if( 0L == m_dwNumAdapters )
    return D3DAPPERR_NOCOMPATIBLEDEVICES;
//-----
// Step 2: Select Adapter and a windowed Device
//-----
// Pick a default device that can render into a window
// (This code assumes that the HAL device comes before the REF
// device in the device array).
for( DWORD a=0; a<m_dwNumAdapters; a++ )
{
    for( DWORD d=0; d < m_Adapters[a].dwNumDevices; d++ )
```

```

    {
        if( m_Adapters[a].devices[d].bWindowed )
        {
            m_Adapters[a].dwCurrentDevice = d;
            m_dwAdapter = a;
            m_bWindowed = TRUE;
            // Display a warning message
            if( m_Adapters[a].devices[d].DeviceType == D3DDEVTYPE_REF )
            {
                if( !bHALExists )
                    DisplayErrorMsg( D3DAPPERR_NOHARDWAREDEVICE,
MSGWARN_SWITCHEDTOREF );
                else if( !bHALIsSampleCompatible )
                    DisplayErrorMsg( D3DAPPERR_HALNOTCOMPATIBLE,
MSGWARN_SWITCHEDTOREF );
                else if( !bHALIsWindowedCompatible )
                    DisplayErrorMsg( D3DAPPERR_NOWINDOWEDHAL, MSGWARN_SWITCHED-
TOREF );
                else if( !bHALIsDesktopCompatible )
                    DisplayErrorMsg( D3DAPPERR_NODESKTOPHAL, MSGWARN_SWITCHED-
TOREF );
                else // HAL is desktop compatible, but not sample compatible
                    DisplayErrorMsg( D3DAPPERR_NOHALTHISMODE, MSGWARN_SWITCHED-
TOREF );
            }
            return S_OK;
        }
    }
    return D3DAPPERR_NOWINDOWABLEDEVICES;
}

```

Let's start with a high-level view of the purpose of this function.

`BuildDeviceLists()` consists of two major parts:

Step 1: It fills the `D3DModeInfo`, `D3DDeviceInfo`, and `D3DAdapterInfo` structures, which are located in `d3dapp.h`, and selects a 640-by-480-by-16 mode.

Step 2: It picks an adapter and device that support a window mode.

Before we switch into warp mode, we need to know a few basic structures and BOOL values. First let's take a look at the structures and their "links."

```
struct D3DAdapterInfo
{
    // Adapter data
    D3DADAPTER_IDENTIFIER8 d3dAdapterIdentifier;
    D3DDISPLAYMODE d3ddmDesktop;      // Desktop display mode for this adapter
    // Devices for this adapter
    DWORD             dwNumDevices;
    D3DDeviceInfo   devices[5];
    // Current state
    DWORD             dwCurrentDevice;
};

struct D3DDeviceInfo
{
    // Device data
    D3DDEVTYPE       DeviceType;      // Reference, HAL, etc.
    D3DCAPS8         d3dCaps;        // Capabilities of this device
    const TCHAR*      strDesc;        // Name of this device
    BOOL              bCanDoWindowed; // Whether this device can work in windowed mode
    // Modes for this device
    DWORD             dwNumModes;
    D3DModeInfo     modes[150];
    // Current state
    DWORD             dwCurrentMode;
    BOOL              bWindowed;
    D3DMULTISAMPLE_TYPE MultiSampleType;
};

struct D3DModeInfo
{
    DWORD             Width;          // Screen width in this mode
    DWORD             Height;         // Screen height in this mode
    D3DFORMAT        Format;         // Pixel format in this mode
    DWORD             dwBehavior;    // Hardware / Software / Mixed vertex processing
    D3DFORMAT        DepthStencilFormat; // Which depth/stencil format to use with this
    mode
};
```

I'll try to show that in a graphic:



**Figure C.3:** Adapter/device/mode structures used in the Common files framework

The connection between these structures is

```
...
// access to a D3DAdapterInfo structure through pAdapter
D3DAdapterInfo* pAdapter = &m_Adapters[m_dwNumAdapters];
...

// access to a D3DDeviceInfo structure through pDevice
D3DDeviceInfo* pDevice;
pDevice = &pAdapter->devices[pAdapter->dwNumDevices];
...

// access to a D3DModeInfo through
pDevice->modes[];
```

There are four BOOL values at the beginning of this function:

```
BOOL bHALExists = FALSE;
BOOL bHALIsWindowedCompatible = FALSE;
BOOL bHALIsDesktopCompatible = FALSE;
BOOL bHALIsSampleCompatible = FALSE;
```

The first BOOL value decides if the HAL or the reference rasterizer is used. There is no third choice for a pluggable software device, for example. So if a pluggable software device is used, this framework has to be extended. The second BOOL value will show whether the device driver will be able to render into a window. This is not the case with the Voodoo 2 boards, for example. `bHALIsDesktopCompatible` is TRUE when `bHalIsWindowedCompatible` is TRUE and the device supports the current desktop mode. The last BOOL value will be TRUE when the format is confirmed `ConfirmDevice()`, one of the virtual functions in the derived class, and a HAL device is used.

OK, after checking the basic structures and variables, we are able to take the first step.

#### STEP 1 IN BUILDDVICELIST()

`BuildDeviceList()` fills the `D3DModeInfo`, `D3DDeviceInfo`, and `D3DAdapterInfo` structures with values and picks a default 640-by-480-by-16 mode.

The structures are filled in the following `for()` loop:

```
for( UINT iAdapter = 0; iAdapter < m_pD3D->GetAdapterCount(); iAdapter++ )  
{
```

If you only have one graphic card, this `for()` loop will only run one time. It might be possible that you have an additional graphic card, for example, a Voodoo 2 card.

We retrieve the number of available modes that are supported by this adapter by using

```
...  
DWORD dwNumAdapterModes = m_pD3D->GetAdapterModeCount( iAdapter );  
...
```

In the next `for()` loop we enumerate the display modes into temporary used `D3DDISPLAYMODE` structures by using `EnumAdapterModes()`

```
for( UINT iMode = 0; iMode < dwNumAdapterModes; iMode++ )  
{  
    // Get the display mode attributes  
    D3DDISPLAYMODE DisplayMode;  
    m_pD3D->EnumAdapterModes( iAdapter, iMode, &DisplayMode );  
    // Filter out low-resolution modes  
    if( DisplayMode.Width < 640 || DisplayMode.Height < 400 )  
        continue;  
    ...
```

Low-resolution modes are not processed by the following `for()` loop. The `DisplayMode` structures will have doubles in it. So there is a second temporary `D3DDISPLAYMODE` structure called `modes[]` that are used to remove doubles:

```
...
for( DWORD m=0L; m<dwNumModes; m++ )
{
    if( ( modes[m].Width == DisplayMode.Width ) &&
        ( modes[m].Height == DisplayMode.Height ) &&
        ( modes[m].Format == DisplayMode.Format ) )
        break;
}
...
...
```

A new mode is stored in modes[] and formats[]:

```
...
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( DWORD 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;
}
}
```

At the end of the for() loop, which counts until dwNumAdapterModes, we have all the modes that are supported by this adapter stored in modes[] structures without doubles.

To sort these modes by format, width, and height, the Common files framework uses qsort(), a function from the math library of Visual C/C++.

```
qsort( modes, dwNumModes, sizeof(D3DDISPLAYMODE), SortModesCallback );
```

After getting the modes the specific graphic adapter supports, we dig for the device formats it supports with

```
for( UINT iDevice = 0; iDevice < dwNumDeviceTypes; iDevice++ )
{
    // Fill in device info
    D3DDeviceInfo* pDevice;
    pDevice              = &pAdapter->devices[pAdapter->dwNumDevices];
    pDevice->DeviceType = DeviceTypes[iDevice];
    m_pD3D->GetDeviceCaps( iAdapter, DeviceTypes[iDevice], &pDevice->d3dCaps );
    pDevice->strDesc      = strDeviceDescs[iDevice];
    pDevice->dwNumModes   = 0;
    pDevice->dwCurrentMode = 0;
    pDevice->bCanDoWindowed = FALSE;
    pDevice->bWindowed     = FALSE;
    pDevice->MultiSampleType = D3DMULTISAMPLE_NONE;
    // Examine each format supported by the adapter to see if it will
    // work with this device and meets the needs of the application.
    BOOL bFormatConfirmed[20];
    DWORD dwBehavior[20];
    D3DFORMAT fmtDepthStencil[20];
    for( DWORD f=0; f<dwNumFormats; f++ )
    {
        bFormatConfirmed[f] = FALSE;
        fmtDepthStencil[f] = D3DFMT_UNKNOWN;
        // Skip formats that cannot be used as render targets on this device
        if( FAILED( m_pD3D->CheckDeviceType( iAdapter, pDevice->DeviceType,
                                              formats[f], formats[f], FALSE ) ) )
            continue;
    ...
    pDevice points to a D3DDeviceInfo located in pAdapter. Device formats that cannot be used as a render target are skipped. CheckDeviceType() verifies whether a certain device type can be used on this adapter and expects hardware acceleration using the given format. We are able to check whether the device driver is a HAL driver with the second parameter of this function.
```

...

```
if( pDevice->DeviceType == D3DDEVTYPE_HAL )
{
    // This system has a HAL device
    bHALExists = TRUE;
    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;
        }
    }
}
...

```

This code sets two BOOL values if the HAL driver is capable of running windowed in the current desktop mode.

BuildDeviceList() checks now the vertex processing capabilities of the device driver:

```
...
if( pDevice->d3dCaps.DevCaps&D3DDEVCAPS_HWTRANSFORMANDLIGHT )
{
    if( pDevice->d3dCaps.DevCaps&D3DDEVCAPS_PUREDEVICE )
    {
        dwBehavior[f] = D3DCREATE_HARDWARE_VERTEXPROCESSING |
                        D3DCREATE_PUREDEVICE;
        if( SUCCEEDED( ConfirmDevice( &pDevice->d3dCaps,
dwBehavior[f],
                                    formats[f] ) ) )
            bFormatConfirmed[f] = TRUE;
    }
    if( FALSE == bFormatConfirmed[f] )
    {
        dwBehavior[f] = D3DCREATE_HARDWARE_VERTEXPROCESSING;
        if( SUCCEEDED( ConfirmDevice( &pDevice->d3dCaps,
dwBehavior[f],
                                    formats[f] ) ) )
            bFormatConfirmed[f] = TRUE;
    }
    if( FALSE == bFormatConfirmed[f] )
    {
        dwBehavior[f] = D3DCREATE_MIXED_VERTEXPROCESSING;
        if( SUCCEEDED( ConfirmDevice( &pDevice->d3dCaps,
dwBehavior[f],
                                    formats[f] ) ) )
            bFormatConfirmed[f] = TRUE;
    }
}
```

```
        }
    }
    // Confirm the device/format for SW vertex processing
    if( FALSE == bFormatConfirmed[f] )
    {
        dwBehavior[f] = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
        if( SUCCEEDED( ConfirmDevice( &pDevice->d3dCaps, dwBehavior[f],
                                     formats[f] ) ) )
            bFormatConfirmed[f] = TRUE;
    }
...
}
```

There is also a check for a pure device. I haven't promoted this special device until now. It gives you only a subset of the Direct3D functionality that a specific card supports. You can't use, for example, Get\*() calls with a pure device, but a pure device is a lot faster because Direct3D is not intercepting all the calls, keeping track of states, or worrying about whether it needs to step in and emulate stuff. It's a "close to the metal" interface.

The following lines in the source check for the ability to support software, hardware, or mixed vertex processing. Software vertex processing provides a guaranteed set of vertex processing capabilities, including an unbounded number of lights and full support for programmable vertex shaders. The only requirement is that vertex buffers used for software processing must be allocated in system memory.

With hardware vertex processing, the capabilities of the driver are variable, depending on the display adapter and driver. Hardware vertex processing is required to create a pure device. You have to consult the VertexProcessingCaps member of the D3DCAPS8 structure to determine the capabilities of the device. For software vertex processing the following capabilities are supported:

- D3DVTXPCAPS\_DIRECTIONALLIGHTS
- D3DVTXPCAPS\_LOCALVIEWER
- D3DVTXPCAPS\_MATERIALSOURCE7
- D3DVTXPCAPS\_POSITIONALLIGHTS
- D3DVTXPCAPS\_TEXGEN
- D3DVTXPCAPS\_TWEENING

The D3DCAPS8 structure will give you the following values for a device in software vertex processing mode:

• MaxActiveLights	Unlimited
• MaxUserClipPlanes	6
• MaxVertexBlendMatrices	4
• MaxStreams	16
• MaxVertexIndex	0xFFFFFFFF

The capability to use a specific type of vertex processing mode determines the kind of resource management used with vertex buffers and index buffers. Vertex processing, including transformation and lighting, in hardware works best when the vertex buffers are allocated in driver-optimal memory, while software vertex processing works best with vertex buffers allocated in system memory. For textures, hardware rasterization works best when textures are allocated in driver-optimal memory, while software rasterization works best with system-memory textures.

Applications control the memory allocation for vertex buffers when they are created. When the D3DPOOL\_SYSTEMMEM memory flag is set, the vertex buffer is created in system memory. When the D3DPOOL\_DEFAULT memory flag is used, the device driver determines where the memory for the vertex buffer is best allocated, often referred to as driver-optimal memory. The D3DPOOL\_MANAGED flag copies resources automatically to device-accessible memory as needed. Managed resources are backed by system memory and do not need to be recreated when a device is lost.

OK... back to `BuildDeviceList()`. After confirming the vertex processing capabilities of the device, the supported depth and stencil buffer format is searched with a call to `FindDepthStencilFormat()`.

```
...
    if( bFormatConfirmed[f] && m_bUseDepthBuffer )<
    {
        if( !FindDepthStencilFormat( iAdapter, pDevice->DeviceType,
            formats[f], &fmtDepthStencil[f] ) )
        {
            bFormatConfirmed[f] = FALSE;
        }
    }
}
...
}
```

This function checks the `m_dwMinDepthBits` and `m_dwMinStencilBits` values. The first one holds the minimum number of bits needed in the depth buffer, and the second holds the minimum number of bits needed in the stencil buffer. These values are member values of the base class and are normally overwritten in the derived class. You can see this in the constructor of the `CMyD3DApplication` class, which is derived from the base class `CD3DApplication` in, for example, `stencilmirror.cpp`:

```
CMyD3DApplication::CMyD3DApplication()
{
    m_strWindowTitle      = _T("StencilMirror: Doing Reflections with Stencils");
    m_bUseDepthBuffer     = TRUE;
    m_dwMinDepthBits      = 16;
    m_dwMinStencilBits    = 4;
    m_pFont               = new CD3DFont( _T("Arial"), 12, D3DFONT_BOLD );
    m_pTerrain            = new CD3DFile();
```

```

    m_pHelicopter      = new CD3DFile();
    m_pMirrorVB        = NULL;
}

```

This application wants to use a 16-bit depth buffer and a 4-bit stencil buffer. The check for a proper depth and stencil buffer format is done with `CheckDeviceFormat()` and `CheckDepthStencilMatch()`.

```

...
    if( m_dwMinDepthBits <= 24 && m_dwMinStencilBits <= 8 )
    {
        if( SUCCEEDED( m_pD3D->CheckDeviceFormat( iAdapter, DeviceType,
            TargetFormat, D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_SURFACE, D3DFMT_D24S8 ) ) )
    }
    {
        if( SUCCEEDED( m_pD3D->CheckDepthStencilMatch( iAdapter, DeviceType,
            TargetFormat, TargetFormat, D3DFMT_D24S8 ) ) )
        {
            *pDepthStencilFormat = D3DFMT_D24S8;
            return TRUE;
        }
    }
}
...

```

`CheckDeviceFormat()` determines whether a specified depth/stencil format is available on a device. `CheckDepthStencilMatch()` determines whether a depth/stencil format is compatible with the render target format in a particular display mode. The `D3DFMT_D24S8` format flag tells the functions to check the availability of a 24-bit depth buffer and an 8-bit stencil buffer at once.

After finding a suitable depth and stencil buffer format for the specific device format, all of the enumerated modes with their vertex processing capabilities in `dwBehaviour[]` and depth and stencil buffer format are stored in `D3DModeInfo` structures, which are part of `D3DAdapterInfo` structures.

```

// Add all enumerated display modes with confirmed formats to the
// device's list of valid modes
for( DWORD m=0L; m<dwNumModes; m++ )
{
    for( DWORD 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;
}
}
}
}

```

If one of the modes supports a HAL device, the BOOL value `bHALIsSampleCompatible` is set to TRUE. Just keep in mind that `pDevice` is a pointer that is located in a `D3DAdapterInfo` structure and points to a `D3DDeviceInfo` structure, which holds 150 `D3DModeInfo` structures in the `modes` array.

After storing all of the available values in the `D3DModeInfo` `modes[]` array, a default 640-by-480 mode is selected in `dwCurrentMode`:

```
for( m=0; m<pDevice->dwNumModes; m++ )
{
    if( pDevice->modes[m].Width==640 && pDevice->modes[m].Height==480 )
    {
        pDevice->dwCurrentMode = m;
        if( pDevice->modes[m].Format == D3DFMT_R5G6B5 || 
            pDevice->modes[m].Format == D3DFMT_X1R5G5B5 || 
            pDevice->modes[m].Format == D3DFMT_A1R5G5B5 )
        {
            break;
        }
    }
}
```

The `for()` loop will be left with `break` when a 16-bit mode is found. If no adapter was found, an error will be returned later.

Now we have finished Step 1 in `BuildDeviceList()`. All of the values needed for handling different adapters, device drivers, and modes are stored and sorted without doubles in a proper format. Additionally, one mode is selected as the default 640-by-480-by-16 mode. Let's pick a default adapter and device in Step 2.

**STEP 2 IN BuildDeviceList()**

After we have filled all of the different modes into their proper adapter structures, we have to choose an adapter with a device that supports a windowed mode.

```
...
for( DWORD a=0; a<m_dwNumAdapters; a++ )
{
    for( DWORD d=0; d < m_Adapters[a].dwNumDevices; d++ )
    {
        if( m_Adapters[a].devices[d].bWindowed )
        {
            m_Adapters[a].dwCurrentDevice = d;
            m_dwAdapter = a;
            m_bWindowed = TRUE;
            // Display a warning message
            if( m_Adapters[a].devices[d].DeviceType == D3DDEVTYPE_REF )
            {
                if( !bHALExists )
                    DisplayErrorMsg( D3DAPPERR_NOHARDWAREDEVICE, MSGWARN_SWITCHEDTOREF );
                else if( !bHALIsSampleCompatible )
                    DisplayErrorMsg( D3DAPPERR_HALNOTCOMPATIBLE, MSGWARN_SWITCHEDTOREF );
                else if( !bHALIsWindowedCompatible )
                    DisplayErrorMsg( D3DAPPERR_NOWINDOWEDHAL, MSGWARN_SWITCHEDTOREF );
                else if( !bHALIsDesktopCompatible )
                    DisplayErrorMsg( D3DAPPERR_NODESKTOPHAL, MSGWARN_SWITCHEDTOREF );
                else // HAL is desktop compatible, but not sample compatible
                    DisplayErrorMsg( D3DAPPERR_NOHALTHISMODE, MSGWARN_SWITCHEDTOREF );
            }
            return S_OK;
        }
    }
}
```

The number of the adapter is stored in the global variable `m_dwAdapter`, and the number of the chosen device is in its adapter structure. A global flag is set, which shows that the application will run in windowed mode.

If a device with the specified features is found, the `for()` loop is left with `return S_OK`. If no device driver of any adapter supports a HAL, or one of the four `BOOL` variables is set to `FALSE`, an error message appears that explains the problem.

If a windowed mode can't be found in BuildDeviceList(), D3DAPPERR\_NOWINDOWABLEDEVICES is returned and an error message is sent out.

Now let's move to Step 3 in our Create() function.

### STEP 3: CREATE A WINDOW WITH CreateWindow()

We skip this step, because you know how to create a window from Appendix A.

### STEP 4: INITIALIZE THE GEOMETRY DATA OF YOUR GAME WITH OneTimeSceneInit()

The virtual function OneTimeSceneInit() is called from within the Create() function. It is used as a function of the derived class CMyD3DApplication. If your application wants to allocate memory for geometry data or to set up geometry data, this might be the right place.

### STEP 5: INITIALIZE THE 3-D ENVIRONMENT WITH Initialize3DEnvironment()

Initialize3DEnvironment() needs seven—sometimes eight—steps to initialize the 3-D environment for your game. I have made additional comments in the source to mark these steps:

```
HRESULT CD3DApplication::Initialize3DEnvironment()
{
    HRESULT hr;
    D3DAdapterInfo* pAdapterInfo = &m_Adapters[m_dwAdapter];
    D3DDeviceInfo* pDeviceInfo = &pAdapterInfo->devices[pAdapterInfo-
>dwCurrentDevice];
    D3DModeInfo* pModeInfo = &pDeviceInfo->modes[pDeviceInfo->dwCurrentMode];
    //_____
    // Step 1: sets another window style depending on windowed or fullscreen
    //_____
    // Prepare window for possible windowed/fullscreen change
    AdjustWindowForChange();
    //_____
    // Step 2: create the Direct3D device
    //_____
    // Set up the presentation parameters
    ZeroMemory( &m_d3dpp, sizeof(m_d3dpp) );
    m_d3dpp.Windowed = pDeviceInfo->bWindowed;
    m_d3dpp.BackBufferCount = 1;
    m_d3dpp.MultiSampleType = pDeviceInfo->MultiSampleType;
    m_d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
```



```
// Step 4: store the vertex processing type and adapter identifier if
//           provided in m_strDeviceStats to show it in the application
//-----
// Store device Caps
m_pd3dDevice->GetDeviceCaps( &m_d3dCaps );
m_dwCreateFlags = pCreateInfo->dwBehavior;
// Store device description
if( pDeviceInfo->DeviceType == D3DDEVTYPE_REF )
    lstrcpy( m_strDeviceStats, TEXT("REF") );
else if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
    lstrcpy( m_strDeviceStats, TEXT("HAL") );
else if( pDeviceInfo->DeviceType == D3DDEVTYPE_SW )
    lstrcpy( m_strDeviceStats, TEXT("SW") );
if( pCreateInfo->dwBehavior & D3DCREATE_HARDWARE_VERTEXPROCESSING &&
    pCreateInfo->dwBehavior & D3DCREATE_PUREDEVICE )
{
    if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
        lstrcat( m_strDeviceStats, TEXT(" (pure hw vp)") );
    else
        lstrcat( m_strDeviceStats, TEXT(" (simulated pure hw vp)") );
}
else if( pCreateInfo->dwBehavior & D3DCREATE_HARDWARE_VERTEXPROCESSING )
{
    if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
        lstrcat( m_strDeviceStats, TEXT(" (hw vp)") );
    else
        lstrcat( m_strDeviceStats, TEXT(" (simulated hw vp)") );
}
else if( pCreateInfo->dwBehavior & D3DCREATE_MIXED_VERTEXPROCESSING )
{
    if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
        lstrcat( m_strDeviceStats, TEXT(" (mixed vp)") );
    else
        lstrcat( m_strDeviceStats, TEXT(" (simulated mixed vp)") );
}
else if( pCreateInfo->dwBehavior & D3DCREATE_SOFTWARE_VERTEXPROCESSING )
{
    lstrcat( m_strDeviceStats, TEXT(" (sw vp)") );
}
if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
{
```

```
    lstrcat( m_strDeviceStats, TEXT(": ") );
    lstrcat( m_strDeviceStats, pAdapterInfo->d3dAdapterIdentifier.Description
);
}

//-----
// Step 5: get the back buffer description
//-----
// Store render target surface desc
LPDIRECT3DSURFACE8 pBackBuffer;
m_pd3dDevice->GetBackBuffer( 0, D3DBACKBUFFER_TYPE_MONO, &pBackBuffer );
pBackBuffer->GetDesc( &m_d3dsdBackBuffer );
pBackBuffer->Release();
//-----
// Step 6: set the fullscreen cursor
//-----
// Set up the fullscreen cursor
if( m_bShowCursorWhenFullscreen && !m_bWindowed )
{
    HCURSOR hCursor;
#ifdef _WIN64
        hCursor = (HCURSOR)GetClassLongPtr( m_hWnd, GCLP_HCURSOR );
#else
        hCursor = (HCURSOR)GetClassLong( m_hWnd, GCL_HCURSOR );
#endif
    D3DUtil_SetDeviceCursor( m_pd3dDevice, hCursor );
    m_pd3dDevice->>ShowCursor( TRUE );
}
//-----
// Step 7: initialize the application's device dependent objects by
//         calling the two virtual functions InitDeviceObjects() and
//                 RestoreDeviceObjects().
//-----
hr = InitDeviceObjects();
if( SUCCEEDED(hr) )
{
    hr = RestoreDeviceObjects();
    if( SUCCEEDED(hr) )
    {
        m_bActive = TRUE;
        return S_OK;
    }
}
```

```
        }

    }

//-----
// Step 8: if it doesn't work with a HAL, try to search for a REF and
restart
//           Initialize3DEnvironment() with a recursive call
//-----
// Cleanup before we try again
InvalidateDeviceObjects();
DeleteDeviceObjects();
SAFE_RELEASE( m_pd3dDevice );

}

// If that failed, fall back to the reference rasterizer
if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
{
    // Let the user know we are switching from HAL to the reference rasterizer
DisplayErrorMsg( hr, MSGWARN_SWITCHEDTOREF );
    // Select the default adapter
m_dwAdapter = 0L;
pAdapterInfo = &m_Adapters[m_dwAdapter];
    // Look for a software device
for( UINT i=0L; i<pAdapterInfo->dwNumDevices; i++ )
{
    if( pAdapterInfo->devices[i].DeviceType == D3DDEVTYPE_REF )
    {
        pAdapterInfo->dwCurrentDevice = i;
        pDeviceInfo = &pAdapterInfo->devices[i];
        m_bWindowed = pDeviceInfo->bWindowed;
        break;
    }
}
    // Try again, this time with the reference rasterizer
if( pAdapterInfo->devices[pAdapterInfo->dwCurrentDevice].DeviceType ==
    D3DDEVTYPE_REF )
{
    hr = Initialize3DEnvironment();
}
}

return hr;
}
```

Initialize3DEnvironment() sets a few pointers on the three structures that hold the adapter, device, and mode information. As you remember, these structures are filled in BuildDeviceList():

```
D3DAdapterInfo* pAdapterInfo = &m_Adapters[m_dwAdapter];
D3DDeviceInfo* pDeviceInfo = &pAdapterInfo->devices[pAdapterInfo->dwCurrentDevice];
D3DModeInfo* pModeInfo = &pDeviceInfo->modes[pDeviceInfo->dwCurrentMode];
```

After that, AdjustWindowForChange() is called in Step 1.

#### STEP 1 IN Initialize3DEnvironment()

AdjustWindowForChange() only sets the proper window styles depending on whether the application is windowed or full-screen.

```
HRESULT CD3DApplication::AdjustWindowForChange()
{
    if( m_bWindowed )
    {
        // Set windowed-mode style
        SetWindowLong( m_hWnd, GWL_STYLE, m_dwWindowStyle );
    }
    else
    {
        // Set fullscreen-mode style
        SetWindowLong( m_hWnd, GWL_STYLE, WS_POPUP|WS_SYSMENU|WS_VISIBLE );
    }
    return S_OK;
}
```

If the application runs windowed, the same styles used in CreateWindow() are taken. If it runs in full-screen mode, only a system menu is taken. After setting the proper window style, the device has to be created.

#### STEP 2 IN Initialize3DEnvironment(): CreateDevice()

The CreateDevice() function uses a D3DPRESENT\_PARAMETERS structure called m\_d3dpp as its fifth parameter:

```
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;
...

// Set up the presentation parameters
ZeroMemory( &m_d3dpp, sizeof(m_d3dpp) );
m_d3dpp.Windowed         = pDeviceInfo->bWindowed;
m_d3dpp.BackBufferCount   = 1;
m_d3dpp.MultiSampleType    = pDeviceInfo->MultiSampleType;
m_d3dpp.SwapEffect        = D3DSWAPEFFECT_DISCARD;
m_d3dpp.EnableAutoDepthStencil = m_bUseDepthBuffer;
m_d3dpp.AutoDepthStencilFormat = pModeInfo->DepthStencilFormat;
m_d3dpp.hDeviceWindow     = m_hWnd;
if( m_bWindowed )
{
    m_d3dpp.BackBufferWidth  = m_rcWindowClient.right - m_rcWindowClient.left;
    m_d3dpp.BackBufferHeight = m_rcWindowClient.bottom - m_rcWindowClient.top;
    m_d3dpp.BackBufferFormat = pAdapterInfo->d3ddmDesktop.Format;
}
else
{
    m_d3dpp.BackBufferWidth  = pModeInfo->Width;
    m_d3dpp.BackBufferHeight = pModeInfo->Height;
    m_d3dpp.BackBufferFormat = pModeInfo->Format;
}
// Create the device
hr = m_pD3D->CreateDevice( m_dwAdapter, pDeviceInfo->DeviceType,
                            m_hWndFocus, pModeInfo->dwBehavior, &m_d3dpp,
                            &m_pd3dDevice );

```

`m_d3dpp.windowed` is filled with the `BOOL` value from `D3DDeviceInfo` structure named `pDeviceInfo` used in `pAdapter`. There should be one back buffer, so triple buffering is not what we want here. `BackBufferCount` tells Direct3D the number of back buffers that will be in the swap chain (the buffer-exchange process in full-screen and the blit process in windowed mode). Whether our game supports multisampling depends on the value stored in the `pDeviceInfo`, a.k.a. `pAdapter->devices[]`.

D3DSWAPEFFECT\_DISCARD enables the display driver to select the most efficient presentation technique for the swap chain. Setting EnableAutoDepthStencil to TRUE leads to automatic depth and stencil buffering by Direct3D. You have to feed AutoDepthStencilFormat a valid depth-stencil format. We use the one supported by the chosen mode. The back buffer width and height depends on the width and height of the window or of the full-screen.

OK, that was the fifth parameter of CreateDevice(). m\_dwAdapter is the number of the adapter you chose, and pDeviceInfo->DeviceType shows whether the device driver is a HAL driver or the reference rasterizer. pModeInfo->dwBehaviour provides the vertex processing behavior of this device: hardware, software, or mixed vertex processing.

You should be familiar with these parameters from the BuildDeviceList() function. The next step is setting the window position.

#### STEP 3 IN Initialize3DEnvironment(): SetWindowPos()

There is nothing spectacular at this function call. We use the window rectangle we received just before the CreateWindow() function.

```
if( m_bWindowed )
{
    SetWindowPos( m_hWnd, HWND_NOTOPMOST,
                  m_rcWindowBounds.left, m_rcWindowBounds.top,
                  ( m_rcWindowBounds.right - m_rcWindowBounds.left ),
                  ( m_rcWindowBounds.bottom - m_rcWindowBounds.top ),
                  SWP_SHOWWINDOW );
}
```

Why we set the window position once again after setting it in WinCreate() is more interesting. The explanation can be found in the comment in d3dapp.cpp. If you choose to switch from a 640-by-480 mode in full-screen to a 1,000-by-600 window in windowed mode, you can't set the window size before the display mode has changed. The display mode changes with the call to CreateDevice(), so you might not be able to set the proper window size and position before that.

After setting the window size and position, which is only necessary in windowed mode, a few device flags are retrieved.

#### STEP 4 IN Initialize3DEnvironment(): GetDeviceCaps()

In the fourth step, we populate the m\_strDeviceStats string, which is shown in your application client window:



**Figure C.4:** Displaying the  
`m_strDeviceStats` string

```
// Store device Caps
m_pd3dDevice->GetDeviceCaps( &m_d3dCaps );
m_dwCreateFlags = pModeInfo->dwBehavior;
// Store device description
if( pDeviceInfo->DeviceType == D3DDEVTYPE_REF )
    lstrcpy( m_strDeviceStats, TEXT("REF") );
else if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
    lstrcpy( m_strDeviceStats, TEXT("HAL") );
else if( pDeviceInfo->DeviceType == D3DDEVTYPE_SW )
    lstrcpy( m_strDeviceStats, TEXT("SW") );
    if( pModeInfo->dwBehavior & D3DCREATE_HARDWARE_VERTEXPROCESSING &&
        pModeInfo->dwBehavior & D3DCREATE_PUREDEVICE )
    {
        if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
            lstrcat( m_strDeviceStats, TEXT(" (pure hw vp)") );
        else
            lstrcat( m_strDeviceStats, TEXT(" (simulated pure hw vp)") );
    }
    else if( pModeInfo->dwBehavior & D3DCREATE_HARDWARE_VERTEXPROCESSING )
    {
```

```
if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
    lstrcat( m_strDeviceStats, TEXT(" (hw vp)") );
else
    lstrcat( m_strDeviceStats, TEXT(" (simulated hw vp)") );
}
else if( pModeInfo->dwBehavior & D3DCREATE_MIXED_VERTEXPROCESSING )
{
    if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
        lstrcat( m_strDeviceStats, TEXT(" (mixed vp)") );
    else
        lstrcat( m_strDeviceStats, TEXT(" (simulated mixed vp)") );
}
else if( pModeInfo->dwBehavior & D3DCREATE_SOFTWARE_VERTEXPROCESSING )
{
    lstrcat( m_strDeviceStats, TEXT(" (sw vp)") );
}
if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
{
    lstrcat( m_strDeviceStats, TEXT(": ") );
    lstrcat( m_strDeviceStats, pAdapterInfo->d3dAdapterIdentifier.Description
);
}
```

Depending on whether it's a pure device and whether hardware, software, or mixed vertex processing is supported by the device, a proper string will be copied into `m_strDeviceStats`. If it's a HAL, the adapter identifier description will also be used, as you can see in Figure C.4, where the identifier description of my GeForce is shown.

#### STEP 5 IN Initialize3DEnvironment(): GetDesc()

`GetDesc()` fills the back buffer description structure `D3DSURFACE_DESC`, which is a member of the `CD3DApplication` class.

```
LPDIRECT3DSURFACE8 pBackBuffer;
m_pd3dDevice->GetBackBuffer( 0, D3DBACKBUFFER_TYPE_MONO, &pBackBuffer );
pBackBuffer->GetDesc( &m_d3dsdBackBuffer );
pBackBuffer->Release();
```

The next step is to set a cursor.

**STEP 6 IN Initialize3DEnvironment(): D3DUtil\_SetDeviceCursor()**

The following code fragment is responsible for the missing cursor, if we use an application in full-screen:

```
if( m_bShowCursorWhenFullscreen && !m_bWindowed )
{
    HCURSOR hCursor;
#ifdef _WIN64
    hCursor = (HCURSOR)GetClassLongPtr( m_hWnd, GCLP_HCURSOR );
#else
    hCursor = (HCURSOR)GetClassLong( m_hWnd, GCL_HCURSOR );
#endif
    D3DUtil_SetDeviceCursor( m_pd3dDevice, hCursor );
    m_pd3dDevice->ShowCursor( TRUE );
}
```

There is also a different function call for the upcoming 64-bit windows.

GetClassLong()/GetClassLongPtr() retrieves the existing cursor. D3DUtil\_SetDeviceCursor() sets this cursor in full-screen mode. You just have to set the BOOL value in the constructor of the derived class in the main file to TRUE:

```
CMyD3DApplication::CMyD3DApplication()
{
    // Override base class members
    m_strWindowTitle      = _T("DolphinVS: Tweening Vertex Shader");
    m_bUseDepthBuffer     = TRUE;
    m_bShowCursorWhenFullscreen = TRUE;
```

OK, head on to device initialization.

**STEP 7 IN Initialize3DEnvironment(): INITIALIZE THE APPLICATION'S DEVICE OBJECTS**

This code section calls the well-known functions InitDeviceObjects() and RestoreDeviceObjects() in your derived application class. In the DirectX 8.0 examples, this class is called CMyD3DApplication. Normally they set up textures, set up the world, projection, and view matrices, and so on. It depends on your application code.

```
hr = InitDeviceObjects();
if( SUCCEEDED(hr) )
{
    hr = RestoreDeviceObjects();
    if( SUCCEEDED(hr) )
    {
```

```
    m_bActive = TRUE;
    return S_OK;
}
}
```

In case both calls succeed, the `Initialize3DEnvironment()` function returns `S_OK`. If one of these functions fails, an additional step is necessary.

#### STEP 8 IN Initialize3DEnvironment()

If one of the functions in `Initialize3DEnvironment()` fails, the Common files framework calls the `InvalidateDeviceObjects()` and `DeleteDeviceObjects()` functions, which invalidate and delete application specific device objects, and the framework tries to use a reference rasterizer.

The code assumes that a HAL driver has failed, searches a reference rasterizer, and makes a recursive function call to `Initialize3DEnvironment()` to start the whole thing again with that driver.

```
// Cleanup before we try again
InvalidateDeviceObjects();
DeleteDeviceObjects();
SAFE_RELEASE( m_pd3dDevice );
}

// If that failed, fall back to the reference rasterizer
if( pDeviceInfo->DeviceType == D3DDEVTYPE_HAL )
{
    // Let the user know we are switching from HAL to the reference rasterizer
    DisplayErrorMsg( hr, MSGWARN_SWITCHEDTOREF );
    // Select the default adapter
    m_dwAdapter = 0L;
    pAdapterInfo = &m_Adapters[m_dwAdapter];
    // Look for a software device
    for( UINT i=0L; i<pAdapterInfo->dwNumDevices; i++ )
    {
        if( pAdapterInfo->devices[i].DeviceType == D3DDEVTYPE_REF )
        {
            pAdapterInfo->dwCurrentDevice = i;
            pDeviceInfo = &pAdapterInfo->devices[i];
            m_bWindowed = pDeviceInfo->bWindowed;
            break;
        }
    }
    // Try again, this time with the reference rasterizer
    if( pAdapterInfo->devices[pAdapterInfo->dwCurrentDevice].DeviceType ==
```

```
    D3DDEVTYPE_REF )  
    {  
        hr = Initialize3DEnvironment();  
    }  
}
```

If everything works fine, our 3-D environment is now initialized. Let's see what comes next in the Create() function. Oh yes, starting the timer.

## STEP 6: STARTING THE TIMER WITH DXUTIL\_TIMER()

DXUtil\_Timer() is a powerful timer function that should fulfill any needs you have. Flags that direct this function:

TIMER_RESET	Resets the timer
TIMER_START	Starts the timer
TIMER_STOP	Stops (or pauses) the timer
TIMER_ADVANCE	Advances the timer by 0.1 seconds
TIMER_GETABSOLUTETIME	Gets the absolute system time
TIMER_GETAPPTIME	Gets the current time
TIMER_GETELAPSEDTIME	Gets the time that elapsed between TIMER_GETELAPSED-TIME calls

We restart the timer here:

```
...  
if( m_bFrameMoving )  
    DXUtil_Timer( TIMER_START );  
...
```

The source of this function is located in the dxutil.cpp file:

```
FLOAT __stdcall DXUtil_Timer( TIMER_COMMAND command )  
{  
    static BOOL      m_bTimerInitialized = FALSE;  
    static BOOL      m_bUsingQPF         = FALSE;  
    static LONGLONG  m_llQPFTicksPerSec = 0;  
    // Initialize the timer  
    if( FALSE == m_bTimerInitialized )  
    {  
        m_bTimerInitialized = TRUE;  
    }
```

```
// Use QueryPerformanceFrequency() to get frequency of timer. If QPF is
// not supported, we will timeGetTime() which returns milliseconds.
LARGE_INTEGER qwTicksPerSec;
m_bUsingQPF = QueryPerformanceFrequency( &qwTicksPerSec );
if( m_bUsingQPF )
    m_llQPFTicksPerSec = qwTicksPerSec.QuadPart;
}
if( m_bUsingQPF )
{
    static LONGLONG m_llStopTime      = 0;
    static LONGLONG m_llLastElapsed = 0;
    static LONGLONG m_llBaseTime     = 0;
    double fTime;
    double fElapsed;
    LARGE_INTEGER qwTime;
    // Get either the current time or the stop time, depending
    // on whether we're stopped and what command was sent
    if( m_llStopTime != 0 && command != TIMER_START && command !=
    TIMER_GETABSOLUTETIME)
        qwTime.QuadPart = m_llStopTime;
    else
        QueryPerformanceCounter( &qwTime );
    // Return the elapsed time
    if( command == TIMER_GETELAPSEDTIME )
    {
        fElapsed = (double) ( qwTime.QuadPart - m_llLastElapsed ) /
                    (double) m_llQPFTicksPerSec;
        m_llLastElapsed = qwTime.QuadPart;
        return (FLOAT) fElapsed;
    }
    // Return the current time
    if( command == TIMER_GETAPPTIME )
    {
        double fAppTime = (double) ( qwTime.QuadPart - m_llBaseTime ) /
                           (double) m_llQPFTicksPerSec;
        return (FLOAT) fAppTime;
    }
    // Reset the timer
    if( command == TIMER_RESET )
    {
        m_llBaseTime      = qwTime.QuadPart;
        m_llLastElapsed = qwTime.QuadPart;
```

```
        return 0.0f;
    }
    // Start the timer
    if( command == TIMER_START )
    {
        m_llBaseTime += qwTime.QuadPart - m_llStopTime;
        m_llStopTime = 0;
        m_llLastElapsed = qwTime.QuadPart;
        return 0.0f;
    }
    // Stop the timer
    if( command == TIMER_STOP )
    {
        m_llStopTime = qwTime.QuadPart;
        m_llLastElapsed = qwTime.QuadPart;
        return 0.0f;
    }
    // Advance the timer by 1/10th second
    if( command == TIMER_ADVANCE )
    {
        m_llStopTime += m_llQPFTicksPerSec/10;
        return 0.0f;
    }
    if( command == TIMER_GETABSOLUTETIME )
    {
        fTime = qwTime.QuadPart / (double) m_llQPFTicksPerSec;
        return (FLOAT) fTime;
    }
    return -1.0f; // Invalid command specified
}
else
{
// code redundant
// uses timeGetTime()
}
}
```

This code uses two possible ways to get the system time. `QueryPerformanceFrequency()` is only supported on Pentium-class and higher hardware because it uses Pentium-specific instructions (high-resolution performance counter). `timeGetTime()` is supported on every hardware since 80 by 386, but it costs a few cycles more and is not as accurate as `QueryPerformanceFrequency()`.

OK, let's dive a little bit deeper into the source. QueryPerformanceFrequency() (QPC) returns 64-bit values, so you shouldn't be surprised to see `LONGLONG` and `LARGE_INTEGER` data types.

QPC outputs a `LARGE_INTEGER` structure that is used to store 64-bit values:

```
LARGE_INTEGER qwTime;  
...  
QueryPerformanceCounter( &qwTime );  
...  
typedef union _LARGE_INTEGER {  
    struct {  
        DWORD LowPart;  
        LONG HighPart;  
    };  
    LONGLONG QuadPart;  
} LARGE_INTEGER, *PLARGE_INTEGER;
```

You can access the first 32 bits of this structure with, for example, `qwTime.LowPart` and the last 32 bits with `qwTime.HighPart`. The whole `LARGE_INTEGER` can be accessed with `qwTime.QuadPart`.

Let's check that in one example:

```
if( command == TIMER_GETELAPSEDTIME )  
{  
    fElapsedTime = (double) ( qwTime.QuadPart - m_llLastElapsedTime ) /  
                           (double) m_llQPFTicksPerSec;  
    m_llLastElapsedTime = qwTime.QuadPart;  
    return (FLOAT) fElapsedTime;  
}
```

The ticks per second we need is retrieved via

```
QueryPerformanceFrequency(&qwTicksPerSec);
```

So, this if statement returns `qwTime.QuadPart - m_llLastElapsedTime / m_llQPFTicksPerSec`. It also stores the last `qwTime` structure in `m_llLastElapsedTime`. It shouldn't be a problem for you to understand the other if statements.

After the starting of the timer, the `Create()` function has done its work. We can start to climb the next know-how mountain. I would like to remind you of the function calls in the `WinMain()` function of the applications file:

```
...  
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT )  
{
```

```
CMyD3DApplication d3dApp;
if( FAILED( d3dApp.Create( hInst ) ) )
    return 0;
return d3dApp.Run();
}

...
LRESULT CMyD3DApplication::MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam,
                                    LPARAM lParam )
{
    if( uMsg == WM_COMMAND )
    {
        // Handle the open file dialog
        if( LOWORD(wParam) == IDM_OPENFILE )
        {
            Pause( TRUE );
            LoadFile(cmd3Mode1);
            Pause( FALSE );
        }
    }
    return CD3DApplication::MsgProc( hWnd, uMsg, wParam, lParam );
}
```

Now Run() enters the battlefield.

## Run()

The Run() function is the message pump that we know from the window skeleton and win skeleton++ examples. There are only two new things you might explore.

```
INT CD3DApplication::Run()
{
    // Load keyboard accelerators
    HACCEL hAccel = LoadAccelerators( NULL, MAKEINTRESOURCE(IDR_MAIN_ACCEL) );
    // Now we're ready to receive and process Windows messages.
    BOOL bGotMsg;
    MSG msg;
    PeekMessage( &msg, NULL, 0U, 0U, PM_NOREMOVE );
    while( WM_QUIT != msg.message )
    {
        // Use PeekMessage() if the app is active, so we can use idle time to
        // render the scene. Else, use GetMessage() to avoid eating CPU time.
        if( m_bActive )
```

```
bGotMsg = PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE );
else
    bGotMsg = GetMessage( &msg, NULL, 0U, 0U );
if( bGotMsg )
{
    // Translate and dispatch the message
    if( 0 == TranslateAccelerator( m_hWnd, hAccel, &msg ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
}
else
{
    // Render a frame during idle time (no messages are waiting)
    if( m_bActive && m_bReady )
    {
        if( FAILED( Render3DEnvironment() ) )
            SendMessage( m_hWnd, WM_CLOSE, 0, 0 );
    }
}
}

return (INT)msg.wParam;
}
```

The handling of the accelerator shortcuts and the call to `Render3DEnvironment()` are new. `LoadAccelerators()` loads the specified accelerator table from the resources. `TranslateAccelerator()` processes accelerator keys for menu commands. The function translates a `WM_KEYDOWN` or `WM_SYSKEYDOWN` message to a `WM_COMMAND` or `WM_SYSCOMMAND` message (if there is an entry for the key in the specified accelerator table) and then sends the `WM_COMMAND` or `WM_SYSCOMMAND` message directly to the appropriate window procedure. It does not return until the window procedure has processed the message.

`Render3DEnvironment()` moves and renders your game. As always, I have made additional comments in the source to mark the important steps:

```
HRESULT CD3DApplication::Render3DEnvironment()
{
    HRESULT hr;
    //_____
    // Step 1: Reports the current cooperative-level status and
    //         resize 3-D environment
```

```
//-----  
// Test the cooperative level to see if it's okay to render  
if( FAILED( hr = m_pd3dDevice->TestCooperativeLevel() ) )  
{  
    // If the device was lost, do not render until we get it back  
    if( D3DERR_DEVICELOST == hr )  
        return S_OK;  
    // Check if the device needs to be resized.  
    if( D3DERR_DEVICENOTRESET == hr )  
    {  
        // If we are windowed, read the desktop mode and use the same format  
        for  
        {  
            // the back buffer  
            if( m_bWindowed )  
            {  
                D3DAdapterInfo* pAdapterInfo = &m_Adapters[m_dwAdapter];  
                m_pD3D->GetAdapterDisplayMode( m_dwAdapter, &pAdapterInfo-  
>d3ddmDesktop );  
                m_d3dpp.BackBufferFormat = pAdapterInfo->d3ddmDesktop.Format;  
            }  
            if( FAILED( hr = Resize3DEnvironment() ) )  
                return hr;  
        }  
        return hr;  
    }  
    //-----  
    // Step 2: Animate all the geometry data  
    //-----  
    // Get the app's time, in seconds. Skip rendering if no time elapsed  
    FLOAT fAppTime      = DXUtil_Timer( TIMER_GETAPPTIME );  
    FLOAT fElapsedAppTime = DXUtil_Timer( TIMER_GETELAPSEDTIME );  
    if( ( 0.0f == fElapsedAppTime ) && m_bFrameMoving )  
        return S_OK;  
    // FrameMove (animate) the scene  
    if( m_bFrameMoving || m_bSingleStep )  
    {  
        // Store the time for the app  
        m_fTime      = fAppTime;  
        m_fElapsedTime = fElapsedAppTime;  
        // Frame move the scene  
        if( FAILED( hr = FrameMove() ) )
```

```
        return hr;
    m_bSingleStep = FALSE;
}

//_____
// Step 3: Render the scene
//_____
// Render the scene as normal
if( FAILED( hr = Render() ) )
    return hr;
//_____
// Step 4: fill the frame count string
//_____
// Keep track of the frame count
{
    static FLOAT fLastTime = 0.0f;
    static DWORD dwFrames = 0L;
    FLOAT fTime = DXUtil_Timer( TIMER_GETABSOLUTETIME );
    ++dwFrames;
    // Update the scene stats once per second
    if( fTime - fLastTime > 1.0f )
    {
        m_fFPS = dwFrames / (fTime - fLastTime);
        fLastTime = fTime;
        dwFrames = 0L;
        // Get adapter's current mode so we can report
        // bit depth (back buffer depth may be unknown)
        D3DDISPLAYMODE mode;
        m_pD3D->GetAdapterDisplayMode(m_dwAdapter, &mode);
        _stprintf( m_strFrameStats, _T("%.02f fps (%dx%dx%d)"),
            m_d3dsdBackBuffer.Width, m_d3dsdBackBuffer.Height,
            mode.Format==D3DFMT_X8R8G8B8?32:16 );
        if( m_bUseDepthBuffer )
        {
            D3DAdapterInfo* pAdapterInfo = &m_Adapters[m_dwAdapter];
            D3DDeviceInfo* pDeviceInfo = &pAdapterInfo->devices[pAdapterInfo-
>dwCurrentDevice];
            D3DModeInfo* pModeInfo = &pDeviceInfo->modes[pDeviceInfo-
>dwCurrentMode];
            switch( pModeInfo->DepthStencilFormat )
            {
                case D3DFMT_D16:
```

```
    lstrcat( m_strFrameStats, _T(" (D16)") );
    break;
case D3DFMT_D15S1:
    lstrcat( m_strFrameStats, _T(" (D15S1)") );
    break;
case D3DFMT_D24X8:
    lstrcat( m_strFrameStats, _T(" (D24X8)") );
    break;
case D3DFMT_D24S8:
    lstrcat( m_strFrameStats, _T(" (D24S8)") );
    break;
case D3DFMT_D24X4S4:
    lstrcat( m_strFrameStats, _T(" (D24X4S4)") );
    break;
case D3DFMT_D32:
    lstrcat( m_strFrameStats, _T(" (D32)") );
    break;
}
}
}
}
//_____
// Step 5: present the whole thing
//_____
// Show the frame on the primary surface.
m_pd3dDevice->Present( NULL, NULL, NULL, NULL );
return S_OK;
}
```

OK, so there are five steps that the Render3DEnvironment() function takes to render and present a scene.

#### STEP 1 IN Render3DEnvironment(): TestCooperativeLevel() AND Resize3DEnvironment()

TestCooperativeLevel() reports the current cooperative-level status of the Direct3D device. It works for full-screen and windowed applications. If the device is operational, this method returns D3D\_OK. If the device is not operational, this method returns D3DERR\_DEVICELOST or D3DERR\_DEVICENOTRESET. The first error message might happen when a full-screen device has lost focus. The second error message will show up after the first one, when a lost device needs to be reset. This must happen with Reset(). What's the idea behind that function?

If an application detects a lost device, it should pause and periodically call `TestCooperativeLevel()` to check the return code. If it returns `D3DERR_DEVICENOTRESET`, the application must reset the device with `Reset()`. That is exactly what the following code is doing:

```
if( FAILED( hr = m_pd3dDevice->TestCooperativeLevel() ) )
{
    // If the device was lost, do not render until we get it back
    if( D3DERR_DEVICELOST == hr )
        return S_OK;
    // Check if the device needs to be resized.
    if( D3DERR_DEVICENOTRESET == hr )
    {
        // If we are windowed, read the desktop mode and use the same format
        for
            // the back buffer
            if( m_bWindowed )
            {
                D3DAdapterInfo* pAdapterInfo = &m_Adapters[m_dwAdapter];
                m_pD3D->GetAdapterDisplayMode( m_dwAdapter, &pAdapterInfo-
>d3ddmDesktop );
                m_d3dpp.BackBufferFormat = pAdapterInfo->d3ddmDesktop.Format;
            }
        if( FAILED( hr = Resize3DEnvironment() ) )
            return hr;
    }
    return hr;
}
```

Well... the call to `Reset()` happens in `Resize3DEnvironment()`. So we have to take a look at that first:

```
HRESULT CD3DApplication::Resize3DEnvironment()
{
    HRESULT hr;
    // Release all vidmem objects
    if( FAILED( hr = InvalidateDeviceObjects() ) )
        return hr;
    // Reset the device
    if( FAILED( hr = m_pd3dDevice->Reset( &m_d3dpp ) ) )
        return hr;
    // Store render target surface desc
    LPDIRECT3DSURFACE8 pBackBuffer;
    m_pd3dDevice->GetBackBuffer( 0, D3DBACKBUFFER_TYPE_MONO, &pBackBuffer );
```

```
pBackBuffer->GetDesc( &m_d3dsdBackBuffer );
pBackBuffer->Release();
// Set up the fullscreen cursor
if( m_bShowCursorWhenFullscreen && !m_bWindowed )
{
    HCURSOR hCursor;
#ifdef _WIN64
    hCursor = (HCURSOR)GetClassLongPtr( m_hWnd, GCLP_HCURSOR );
#else
    hCursor = (HCURSOR)GetClassLong( m_hWnd, GCL_HCURSOR );
#endif
    D3DUtil_SetDeviceCursor( m_pd3dDevice, hCursor );
    m_pd3dDevice->ShowCursor( TRUE );
}
// Initialize the app's device-dependent objects
hr = RestoreDeviceObjects();
if( FAILED(hr) )
    return hr;
// If the app is paused, trigger the rendering of the current frame
if( FALSE == m_bFrameMoving )
{
    m_bSingleStep = TRUE;
    DXUtil_Timer( TIMER_START );
    DXUtil_Timer( TIMER_STOP );
}
return S_OK;
}
```

Hey . . . yes, you know all that stuff from `Initialize3DEnvironment()`; only one function is new, guess which? If a call to `Reset()` fails, `TestCooperativeLevel()` will return `D3DERR_DEVICELOST`. This might happen until the device is in a “not reset” state, indicated by a return value of `D3DERR_DEVICENOTRESET` from `TestCooperativeLevel()`. If `Reset()` fails, the only valid methods that can be called are `Reset()`, `TestCooperativeLevel()`, and various `Release()` member functions.

`Reset()` causes all texture memory surfaces to be lost, managed textures to be flushed from video memory, and all state information to be lost.

Before calling `Reset()`, an application should release any explicit render targets, depth stencil surfaces, additional swap chains, and resources that are associated with the device and the `D3DPPOOL_DEFAULT` flag. This must happen in the `InvalidateDeviceObject()` function in `Resize3DEnvironment()`. After checking for a lost device, action conquers the arena with `FrameMove()`.

**STEP 2 IN** Render3DEnvironment(): FrameMove()

The virtual function *FrameMove()* is used to animate the whole scene that you would like to present. Every Common files framework application has the ability to start a single-step modus that is switched on with the Enter or the spacebar key. To move one step further, you have to press the spacebar key. To leave the single-step mode, you have to press the Enter key.

```
// Get the app's time, in seconds. Skip rendering if no time elapsed
FLOAT fAppTime      = DXUtil_Timer( TIMER_GETAPPTIME );
FLOAT fElapsedAppTime = DXUtil_Timer( TIMER_GETELAPSEDTIME );
if( 0.0f == fElapsedAppTime ) && m_bFrameMoving )
    return S_OK;
// FrameMove (animate) the scene
if( m_bFrameMoving || m_bSingleStep )
{
    // Store the time for the app
    m_fTime      = fAppTime;
    m_fElapsedTime = fElapsedAppTime;
    // Frame move the scene
    if( FAILED( hr = FrameMove() ) )
        return hr;
    m_bSingleStep = FALSE;
}
```

The two variables *m\_bFrameMoving* and *m\_bSingleStep* control the single-step modus: The *m\_bFrameMoving* switches it on, and the *m\_bSingleStep* gives you the ability to make the next step. This piece of code stores the current time and the elapsed time in two member variables so that your application can access them.

**STEP 3 IN** Render3DEnvironment(): Render()

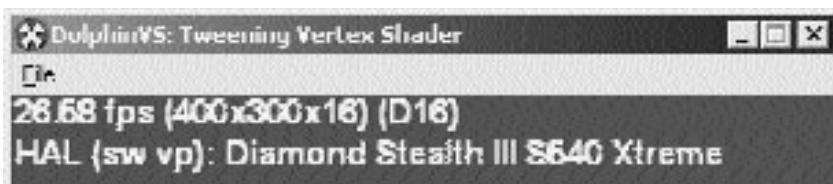
The virtual function call to *Render()* in your applications class, which is derived from the *CD3DApplication* class, calls one of the *DrawPrimitive()* methods to render the vertices that are animated in *FrameMove()*. Normally a lot of state changes and texture mapping happen here, too.

```
// Render the scene as normal
if( FAILED( hr = Render() ) )
    return hr;
```

Because there's nothing special about this call, we go to the next step.

## STEP 4 IN Render3DEnvironment(): FILL THE FRAME COUNT STRING

You know the frame count string from every DirectX 8.0 example. It's located in the upper left corner in these examples.



**Figure C.5:** Displaying the `m_strFrameStats` string

On my graphic card, there are the typical frames-per-second counter, the display mode, and the depth/stencil buffer support.

```
// Keep track of the frame count
{
    static FLOAT fLastTime = 0.0f;
    static DWORD dwFrames = 0L;
    FLOAT fTime = DXUtil_Timer( TIMER_GETABSOLUTETIME );
    ++dwFrames;
    // Update the scene stats once per second
    if( fTime - fLastTime > 1.0f )
    {
        m_fFPS = dwFrames / (fTime - fLastTime);
        fLastTime = fTime;
        dwFrames = 0L;
        // Get adapter's current mode so we can report
        // bit depth (back buffer depth may be unknown)
        D3DDISPLAYMODE mode;
        m_pD3D->GetAdapterDisplayMode(m_dwAdapter, &mode);
        _stprintf( m_strFrameStats, _T("%.02f fps (%dx%dx%d)"),
            m_d3dsdBackBuffer.Width, m_d3dsdBackBuffer.Height,
            mode.Format==D3DFMT_X8R8G8B8?32:16 );
        if( m_bUseDepthBuffer )
        {
            D3DAdapterInfo* pAdapterInfo = &m_Adapters[m_dwAdapter];
            D3DDeviceInfo* pDeviceInfo = &pAdapterInfo->devices[pAdapterInfo->dwCurrentDevice];
        }
    }
}
```

```
D3DModeInfo* pModeInfo = &pDeviceInfo->modes[pDeviceInfo->dwCurrentMode];
    switch( pModeInfo->DepthStencilFormat )
    {
        case D3DFMT_D16:
            lstrcat( m_strFrameStats, _T(" (D16)") );
            break;
        case D3DFMT_D15S1:
            lstrcat( m_strFrameStats, _T(" (D15S1)") );
            break;
        case D3DFMT_D24X8:
            lstrcat( m_strFrameStats, _T(" (D24X8)") );
            break;
        case D3DFMT_D24S8:
            lstrcat( m_strFrameStats, _T(" (D24S8)") );
            break;
        case D3DFMT_D24X4S4:
            lstrcat( m_strFrameStats, _T(" (D24X4S4)") );
            break;
        case D3DFMT_D32:
            lstrcat( m_strFrameStats, _T(" (D32)") );
            break;
    }
}
```

Here, a 16-bit depth buffer and no stencil buffer are used.

#### STEP 5 IN Render3DEnvironment(): Present()

The last and most important call in `Render3DEnvironment()` goes out to `Present()`. This is the functional equivalent to the older `Blit()` and `Flip()` functions since the advent of DirectX 8.0.

```
m_pd3dDevice->Present( NULL, NULL, NULL, NULL );
```

This function—as its name implies—presents the contents that you have rendered. That happens by presenting the next in the sequence of back buffers owned by the device.

```
HRESULT Present(
    CONST RECT* pSourceRect,
    CONST RECT* pDestRect,
```

```
    HWND hDestWindowOverride,  
    CONST RGNDATA* pDirtyRegion  
);
```

Present() copies a buffer region from one memory surface to another. So pSourceRect is the rectangle memory surface region that should be copied into the pDestRect rectangle memory surface. The third parameter is a pointer to a destination window whose client area is taken as the target. The last parameter is not used and should be NULL, but its name sounds promising :-).

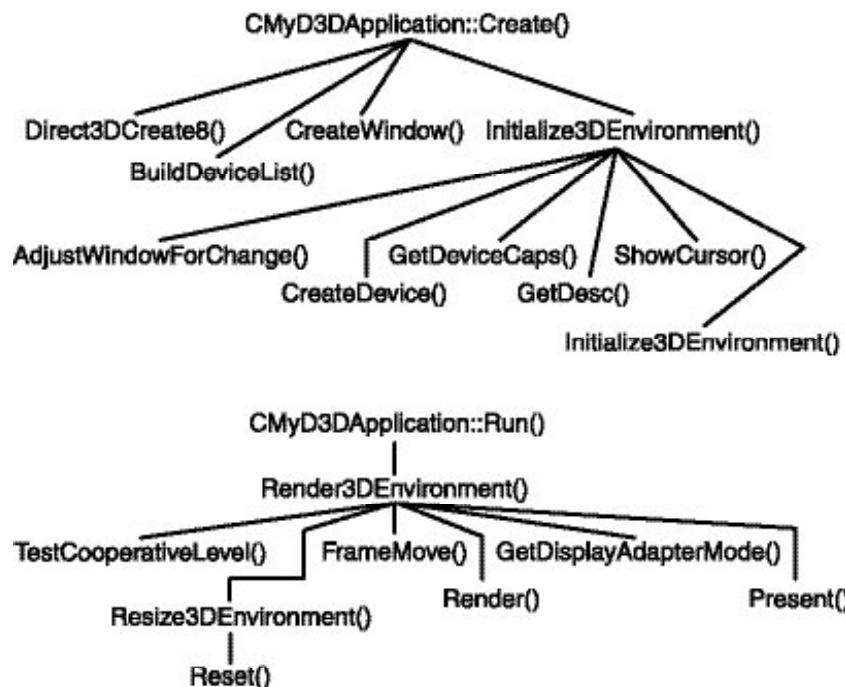
We have provided a NULL to every parameter. For pSourceRect and pDestRect, this means that the entire source and destination surface should be taken. If hDestWindowOverride is NULL, Present() uses the D3DPRESENT\_PARAMETERS structure as the default. The return values are D3D\_OK, D3DERR\_INVALIDCALL, D3DERR\_DEVICELOST . . . you know their meanings. Just one thing: Like the old Blit() method, this method applies a stretch operation to the transfer of the pixels from the source to the destination rectangle.

So we finished Run(). As you see, the functions to start the application with the help of the Common files framework are called in the main file:

```
...  
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT )  
{  
    CMyD3DApplication d3dApp;  
    if( FAILED( d3dApp.Create( hInst ) ) )  
        return 0;  
    return d3dApp.Run();  
}  
...
```

The Run() function will run on and on until the message loop is quit with WM\_QUIT, as we have seen in Appendix A.

A simplified overview of the functions that are called might look like this:



**Figure C.6:** Common files framework functions called

I know such diagrams are not complete nor scientific, but I use such things to help me to visualize the function calls, hoping that it also helps you do this.

OK, let's start running through the `MsgProc()`.

## MsgProc()

As I have shown you above, `WinProc()` in the `CD3DApplication` class calls `MsgProc()` in the derived class via a `this` pointer. So messages sent to `WinProc()`, the default message handling routine, are sent to `MsgProc()` of the derived class. If they are not handled there, they are sent to the `MsgProc()` from the base class and handled there.

```

HRESULT CMyD3DApplication::MsgProc( HWND hWnd, UINT msg, WPARAM wParam,
                                    LPARAM lParam )
{
...
    return CD3DApplication::MsgProc( hWnd, msg, wParam, lParam );
}
  
```

Let's take a look at the `MsgProc()` in the base class that gets the messages that are not handled by the `MsgProc()` in the derived class. I will concentrate on the interesting messages:

```
...
case WM_PAINT:
    // Handle paint messages when the app is not ready
    if( m_pd3dDevice && !m_bReady )
    {
        if( m_bWindowed )
            m_pd3dDevice->Present( NULL, NULL, NULL, NULL );
    }
    break;
...
```

In the `WM_PAINT`, a call goes out to `Present()` if the app is windowed and not ready. That could be the case when the app pauses.

The `WM_EXITSIZEMOVE` message is sent once to a window after it has exited the moving or sizing modal loop. A window enters the moving or sizing modal loop when the user clicks the window's title bar or sizing border, or when the window passes the `WM_SYSCOMMAND` message with a `wParam` parameter that specifies a `SC_MOVE` or `SC_SIZE` value.

```
...
case WM_EXITSIZEMOVE:
    if( m_bFrameMoving )
        DXUtil_Timer( TIMER_START );
    if( m_bActive && m_bWindowed )
    {
        RECT rcClientOld;
        rcClientOld = m_rcWindowClient;
        // Update window properties
        GetWindowRect( m_hWnd, &m_rcWindowBounds );
        GetClientRect( m_hWnd, &m_rcWindowClient );
        if( rcClientOld.right - rcClientOld.left !=  

            m_rcWindowClient.right - m_rcWindowClient.left ||  

            rcClientOld.bottom - rcClientOld.top !=  

            m_rcWindowClient.bottom - m_rcWindowClient.top )
        {
            // A new window size will require a new backbuffer
            // size, so the 3-D structures must be changed accordingly.
            m_bReady = FALSE;
            m_d3dpp.BackBufferWidth = m_rcWindowClient.right -
            m_rcWindowClient.left;
```

```
m_d3dpp.BackBufferHeight = m_rcWindowClient.bottom -  
m_rcWindowClient.top;  
    // Resize the 3-D environment  
    if( FAILED( hr = Resize3DEnvironment() ) )  
    {  
        DisplayErrorMsg( D3DAPPERR_RESIZEFAILED, MSGERR_APPMUSTEXIT  
);  
        return 0;  
    }  
    m_bReady = TRUE;  
}  
}  
break;  
...
```

This code starts the timer again, which was stopped in the WM\_ENTERSIZEMOVE message, and retrieves the size of the client window to change the size of the background buffer.

The WM\_SETCURSOR / WM\_MOUSEMOVE pair sets and moves the mouse cursor when m\_bShowCursorWhenFullscreen is set to TRUE.

```
...  
case WM_SETCURSOR:  
    // Turn off Windows cursor in fullscreen mode  
    if( m_bActive && m_bReady && !m_bWindowed )  
    {  
        SetCursor( NULL );  
        if( m_bShowCursorWhenFullscreen )  
            m_pd3dDevice->>ShowCursor( TRUE );  
        return TRUE; // prevent Windows from setting cursor to window class  
cursor  
    }  
    break;  
case WM_MOUSEMOVE:  
    if( m_bActive && m_bReady && m_pd3dDevice != NULL )  
    {  
        POINT ptCursor;  
        GetCursorPos( &ptCursor );  
        ScreenToClient( m_hWnd, &ptCursor );  
        m_pd3dDevice->SetCursorPosition( ptCursor.x, ptCursor.y, 0L );  
    }  
    break;  
...
```

WM\_ENTERMENULOOP stops the rendering loop by stopping the timer, and WM\_EXITMENULOOP starts it again by starting the timer.

```
...
case WM_ENTERMENULOOP:
    // Pause the app when menus are displayed
    Pause(TRUE);
    break;
case WM_EXITMENULOOP:
    Pause(FALSE);
    break;
...
```

Another interesting message is the WM\_POWERBROADCAST message. It notifies an application of power-management events. The PBT\_APMQUERYSUSPEND in wParam requests permission to suspend.

```
...
case WM_POWERBROADCAST:
    switch( wParam )
    {
        #ifndef PBT_APMQUERYSUSPEND
            #define PBT_APMQUERYSUSPEND 0x0000
        #endif
        case PBT_APMQUERYSUSPEND:
            // At this point, the app should save any data for open
            // network connections, files, etc., and prepare to go into
            // a suspended mode.
            return TRUE;
        #ifndef PBT_APRESUMESUSPEND
            #define PBT_APRESUMESUSPEND 0x0007
        #endif
        case PBT_APRESUMESUSPEND:
            // At this point, the app should recover any data, network
            // connections, files, etc., and resume running from when
            // the app was suspended.
            return TRUE;
    }
    break;
...
```

Returning TRUE allows the power management to suspend. Interesting window messages that might come up in wParam of a WM\_COMMAND message are the IDM\_TOGGLESTART and IDM\_SINGLESTEP pair that control

the single step modus. IDM\_CHANGEDEVICE starts the Device Select dialog box with UserSelectNewDevice(). This function toggles into windowed mode if F2 is pressed in full-screen mode and shows up the dialog box with DialogBoxParam(), which creates a modal dialog box from a dialog box template resource.

One of my favorites is IDM\_TOGGLEFULLSCREEN, which calls the function with the same name:

```
HRESULT CD3DApplication::ToggleFullscreen()
{
    // Get access to current adapter, device, and mode
    D3DAdapterInfo* pAdapterInfo = &m_Adapters[m_dwAdapter];
    D3DDeviceInfo* pDeviceInfo = &pAdapterInfo->devices[pAdapterInfo-
>dwCurrentDevice];
    D3DModeInfo* pModeInfo = &pDeviceInfo->modes[pDeviceInfo->dwCurrentMode];
    // Need device change if going windowed and the current device
    // can only be fullscreen
    if( !m_bWindowed && !pDeviceInfo->bCanDoWindowed )
        return ForceWindowed();
    m_bReady = FALSE;
    // Toggle the windowed state
    m_bWindowed = !m_bWindowed;
    pDeviceInfo->bWindowed = m_bWindowed;
    // Prepare window for windowed/fullscreen change
    AdjustWindowForChange();
    // Set up the presentation parameters
    m_d3dpp.Windowed = pDeviceInfo->bWindowed;
    m_d3dpp.MultiSampleType = pDeviceInfo->MultiSampleType;
    m_d3dpp.AutoDepthStencilFormat = pModeInfo->DepthStencilFormat;
    m_d3dpp.hDeviceWindow = m_hWnd;
    if( m_bWindowed )
    {
        m_d3dpp.BackBufferWidth = m_rcWindowClient.right - m_rcWindowClient.left;
        m_d3dpp.BackBufferHeight = m_rcWindowClient.bottom - m_rcWindowClient.top;
        m_d3dpp.BackBufferFormat = pAdapterInfo->d3ddmDesktop.Format;
    }
    else
    {
        m_d3dpp.BackBufferWidth = pModeInfo->Width;
        m_d3dpp.BackBufferHeight = pModeInfo->Height;
        m_d3dpp.BackBufferFormat = pModeInfo->Format;
    }
    // Resize the 3D device
```

```
if( FAILED( Resize3DEnvironment() ) )
{
    if( m_bWindowed )
        return ForceWindowed();
    else
        return E_FAIL;
}
// When moving from fullscreen to windowed mode, it is important to
// adjust the window size after resetting the device rather than
// beforehand to ensure that you get the window size you want. For
// example, when switching from 640x480 fullscreen to windowed with
// a 1000x600 window on a 1024x768 desktop, it is impossible to set
// the window size to 1000x600 until after the display mode has
// changed to 1024x768, because windows cannot be larger than the
// desktop.
if( m_bWindowed )
{
    SetWindowPos( m_hWnd, HWND_NOTOPMOST,
                  m_rcWindowBounds.left, m_rcWindowBounds.top,
                  ( m_rcWindowBounds.right - m_rcWindowBounds.left ),
                  ( m_rcWindowBounds.bottom - m_rcWindowBounds.top ),
                  SWP_SHOWWINDOW );
}
m_bReady = TRUE;

return S_OK;
}
```

This piece of code switches between full-screen and windowed mode and vice versa. There is one new function in this code: ForceWindowed(). This function switches to a windowed mode, even if the currently selected device driver or adapter doesn't support it, by changing the adapter or device driver. The source shouldn't give you any problems with the know-how you gained as you worked through the BuildDeviceList() function.

The famous last message is WM\_CLOSE. The interesting part is Cleanup3DEnvironment(), the counterpart to Initialize3DEnvironment().

```
VOID CD3DApplication::Cleanup3DEnvironment()
{
    m_bActive = FALSE;
    m_bReady = FALSE;
    if( m_pd3dDevice )
```

```
{  
    InvalidateDeviceObjects();  
    DeleteDeviceObjects();  
    m_pd3dDevice->Release();  
    m_pD3D->Release();  
    m_pd3dDevice = NULL;  
    m_pD3D      = NULL;  
}  
FinalCleanup();  
}
```

This one releases the device-dependent objects with the help of the virtual functions `InvalidateDeviceObjects()`, `DeleteDeviceObjects()`, and the `Release()` function of the device object. At last the virtual function `FinalCleanup()` is called, which cleans up the stuff that was created or allocated in `OneTimeSceneInit()`.

OK . . . that was an overview of the Common files framework. Just build your own framework with that information. I think sometimes about a full-screen framework that would be a replacement for the Common files framework, and faster :-).

After digging a little bit around in Common files source, we will now dive into the world of mathematics.

*This page intentionally left blank*

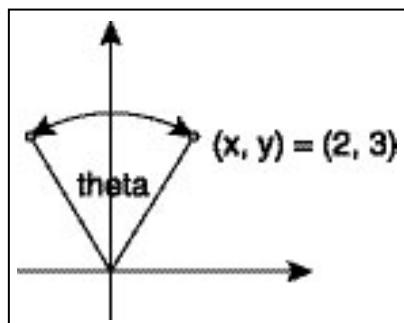
# **APPENDIX D**

## **MATHEMATICS PRIMER**

This appendix provides supplemental information to augment your understanding of the math and 3-D concepts introduced in Part 1 and Part 2. We will kick-start the process of understanding mathematics for 3-D by rotating points in 3-D.

## POINTS IN 3-D

Let's first imagine a point rotating in 2-D space. The two axes make a plane that can be like a piece of paper. This plane is called the *x-y plane*, since it is defined by the two principal axes:



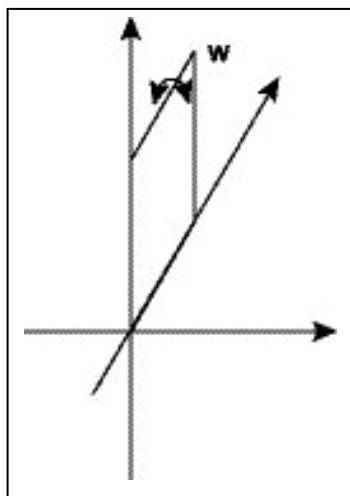
**Figure D.1:** Two-dimensional coordinate system

To rotate a point by theta radians counterclockwise, you have to use the following formula:

$$x_{\text{new}} = x * \cos(\theta) - y * \sin(\theta)$$

$$y_{\text{new}} = x * \sin(\theta) + y * \cos(\theta)$$

OK, now let's jump into 3-D...just add a third axis. Let's call it z; I think this is a nice name for an axis. The difference between 2-D and 3-D is that the z-axis introduces two new principal planes: the x-z plane and the y-z plane. Now the essence: A full 3-D rotation is done by rotating one plane at a time by a specified angle. Here is the same rotation as above in 3D:



**Figure D.2:** Three-dimensional coordinate system

To rotate a point w ( $x, y, z$ ) by theta radians counterclockwise in a 3-D system, you have to use the following formula:

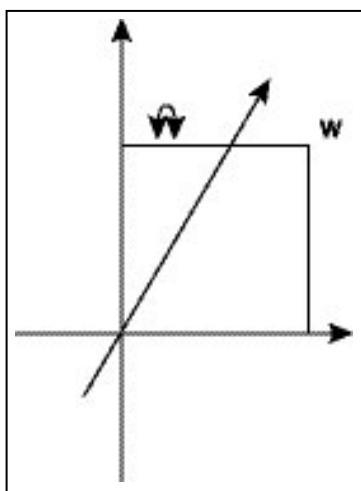
$$x_{\text{new}} = x * \cos(\theta) - y * \sin(\theta)$$

$$y_{\text{new}} = x * \sin(\theta) + y * \cos(\theta)$$

$$z_{\text{new}} = z$$

It's nearly the same formula as above, only there is an additional z value.

The resulting point  $x_{\text{new}}$ ,  $y_{\text{new}}$ , and  $z_{\text{new}}$  is used as the source point for the next rotation. The next rotation might happen about the x-axis:

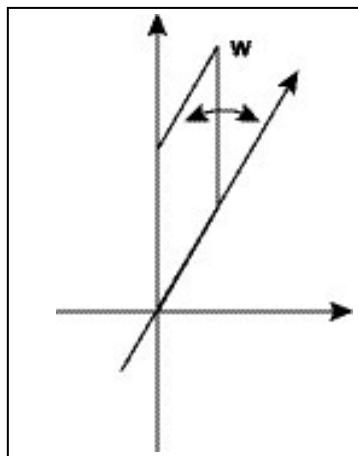


**Figure D.3:** Rotating about the x-axis

To rotate a point w ( $x, y, z$ ) by theta radians about the x-axis, you have to use the following formula:

$$\begin{aligned}x_{\text{new}} &= x \\y_{\text{new}} &= y * \cos(\theta) - z * \sin(\theta) \\z_{\text{new}} &= y * \sin(\theta) + z * \cos(\theta)\end{aligned}$$

Now let's face a rotation about the y-axis:



**Figure D.4: Rotating about the y-axis**

To rotate a point w ( $x, y, z$ ) by theta radians about the y-axis, you have to use the following formula:

$$\begin{aligned}x_{\text{new}} &= x * \cos(\theta) - z * \sin(\theta) \\y_{\text{new}} &= y \\z_{\text{new}} &= x * \sin(\theta) + z * \cos(\theta)\end{aligned}$$

To rotate a point w ( $x, y, z$ ) in 3-D space, we

1. Rotate it in the x-y plane and store the results in  $x_{\text{new}}$ ,  $y_{\text{new}}$ , and  $z_{\text{new}}$ .
2. Rotate the resulting point around the y-z plane and store the results in  $x_2$ ,  $y_2$ , and  $z_2$ .
3. Rotate the resulting point around the z-x plane and store the results in  $x_{\text{new}}$ ,  $y_{\text{new}}$ , and  $z_{\text{new}}$ .

A piece of pseudocode might look like this:

```
xnew = x * cos(theta) - y * sin(theta)
ynew = x * sin(theta) + y * cos(theta)
znew = z
x2 = xnew
y2 = ynew * cos(theta) - znew * sin(theta)
z2 = ynew * sin(theta) + znew * cos(theta)
```

```
xnew = x2 * cos(theta) - z2 * sin(theta)
ynew = y2
znew = x2 * sin(theta) + z2 * cos(theta)
```

To summarize 3-D point rotation: To rotate a point in 3-D, you have to rotate it about the three axes, one after another.

The translation of a point in 3-D is much easier. Moving a point in a direction requires only adding three offsets (representing the direction) to that point: one offset each for x, y, and z. For example, translating the point (2, 3, 0) to (8, 7, 0) requires adding (6, 4, 0) to each component.

## VECTORS

Vectors are mathematical entities that describe a direction and a magnitude (which might be speed). This is the difference from scalars, which describe a magnitude alone. A typical scalar value is, for example, the temperature 32 degrees Celsius. This information is enough to know everything about the temperature at that point. It is more difficult to describe the direction of an airplane and how fast it is going. This would be a job for a vector.

You can differentiate between a general purpose, or bound, vector and the so-called free vector used in games.

### BOUND VECTOR

A bound vector  $\underline{v}$  has the following attributes:

- A starting point
- A final point
- A direction
- A magnitude

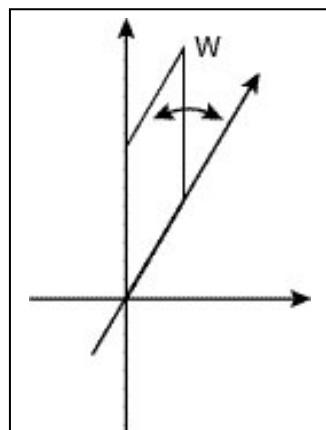
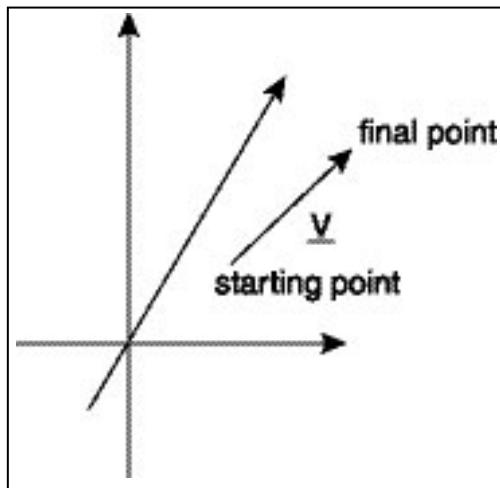


Figure D.5: Bound vector

A bound vector might look like this in the coordinate system:



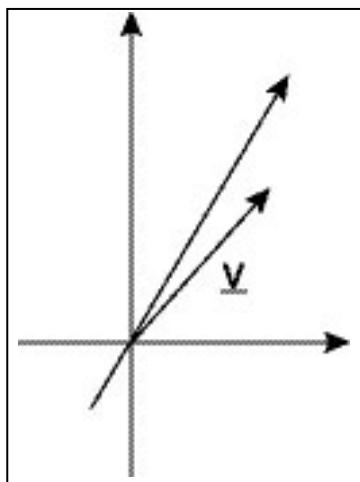
**Figure D.6:** Bound vector  
in coordinate system

A bound vector consists of two coordinates. You can see the direction of these vectors by drawing a line between the two coordinates, and the distance between the points represents the magnitude. The first coordinate is called the starting point and the second is the final point. We write down a vector with an underlined capital: v.

## FREE VECTOR

For game programmers, a vector has no starting point. If all points are moving in the same direction with the same speed, it would be inefficient to use a vector for each point. It is enough to describe the speed and direction with one vector that is used with all points. So we don't have to represent a starting point for a free vector.

By definition, all free vectors have their starting points in the origin, so only three numbers are required to describe a free vector.



**Figure D.7:** Free vector  
in coordinate system

Oops ... you say, "That sounds like a point in 3-D." I say, "That's an important remark." So what's the difference between a point and a vector in 3-D? A free vector is represented additionally by magnitude and direction. The magnitude of a vector  $(2, 3, 2)$  is the distance between the origin  $(0, 0, 0)$  and the final point  $(2, 3, 2)$ .

If you only need the final point, then it is a 3-D point. If you need magnitude and direction as well, then it is a vector. You might say a free vector describes a 3-D point, but that is a more philosophical point of view. A 3-D point might also be described as a null vector with magnitude 0.

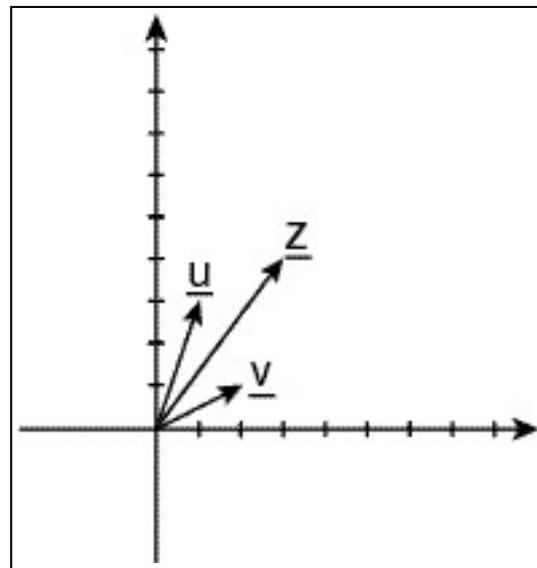
Just like real values, you can use arithmetic operations on vectors. For example, you can add, subtract, and multiply vectors.

#### VECTOR ADDITION: $\underline{U} + \underline{V}$

To add two vectors, you have to add their components:

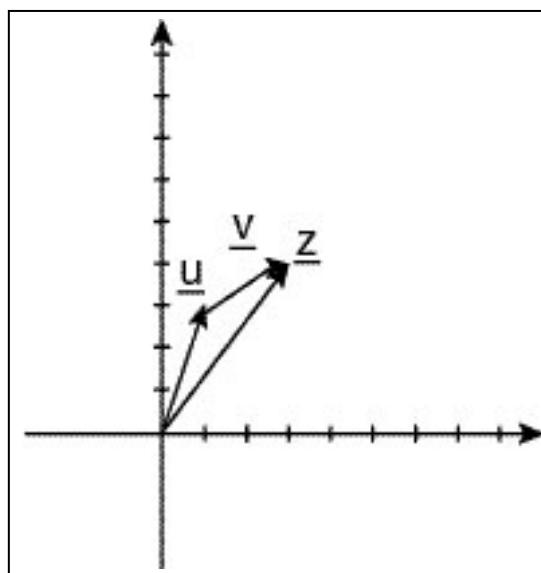
$$\begin{aligned}\underline{U} (1, 3, 2) + \underline{V} (2, 1, 2) \\ = \underline{Z} (3, 4, 4)\end{aligned}$$

This addition can be visualized easier in 2-D space. So I skip the z-value for simplification:



**Figure D.8:** Vector addition

If you would like to add vectors in a more visual way, you can lay the second vector  $\underline{v}$  with its starting point at the final point of the vector  $\underline{u}$  and draw the sum  $\underline{u} + \underline{v}$  between the starting point of  $\underline{u}$  and the final point of  $\underline{v}$ .



**Figure D.9:** Vector addition  
in a graphical way

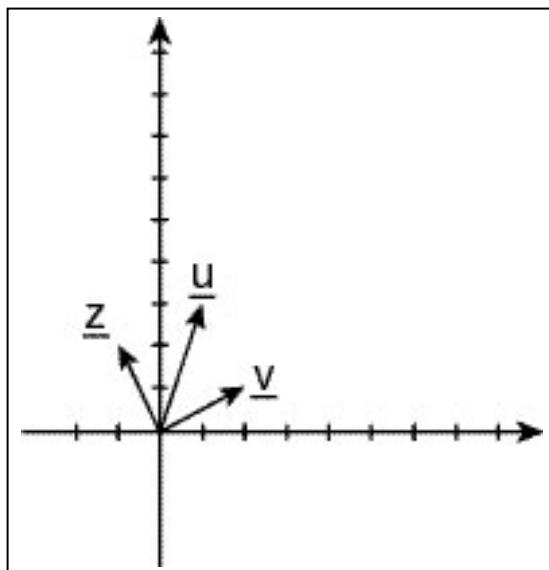
Vector additions are useful to show, for example, the drift you get if you are sailing with your sailing ship. The addition of more vectors works the same. Another important arithmetic is subtraction.

#### VECTOR SUBTRACTION: $\underline{U} - \underline{V}$

Subtracting of vectors is performed by subtracting one vector's component from the other. Vector subtraction undoes vector addition. An example:

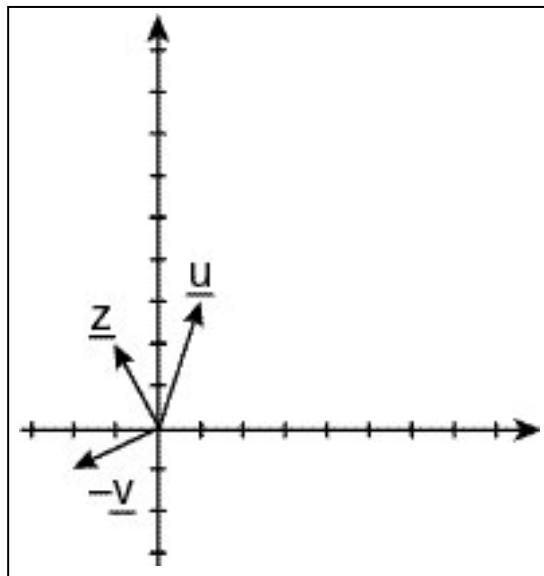
$$\begin{aligned}\underline{U} (1, 3, 2) - \underline{V} (2, 1, 2) \\ = \underline{Z} (-1, 2, 0)\end{aligned}$$

Simplified by using only the x and y values, that might look like:



**Figure D.10:** Vector subtraction

If you would like to subtract the two vectors in a visual way, you might multiply the vector  $\underline{V}$  by  $-1$  and add that to  $\underline{U}$ .



**Figure D.11:** Vector subtraction  
in a graphical way

You will use vector subtraction to remove the effect of vector addition.

### VECTOR MULTIPLICATION

There are three kinds of vector multiplications:

- Scalar product
- Dot product
- Cross product

#### SCALAR PRODUCT

Multiplication of a vector by a scalar (real number) changes the vector's magnitude. It changes the distance between its starting and final points.

$$\begin{aligned} \underline{U} & (1, 3, 2) * 2 \\ & = \underline{Z} (2, 6, 4) \end{aligned}$$

A visual simplified picture might look like:

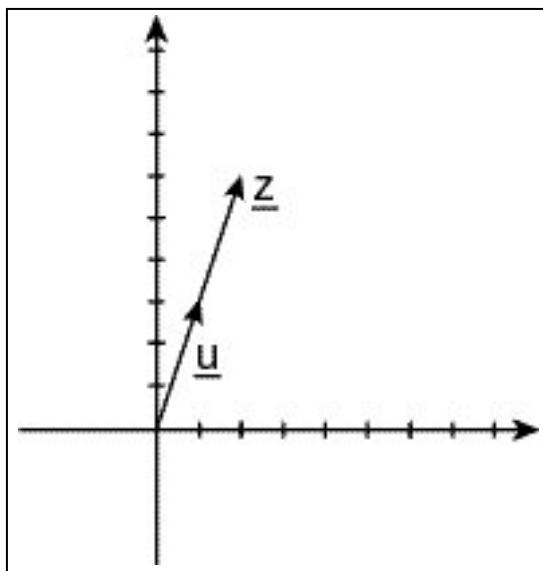


Figure D.12: Scalar product

$\underline{z}$  has double the magnitude of  $\underline{u}$ . So the scalar product results in a vector.

#### DOT PRODUCT

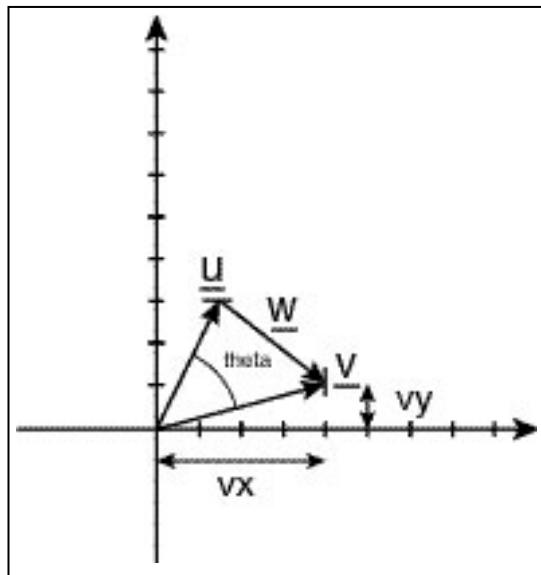
The dot product of two vectors is designated by the dot symbol \*. It results in a real value. This is defined as:

$$\underline{u} * \underline{v}$$

We would like to get the angle between two 2-D vectors  $\underline{u}$  and  $\underline{v}$ .

#### CAUTION

The magnitude of a vector needs to be positive, so if you scale a vector with a negative number w, then you need to multiply the magnitude with `abs(w)` and switch the sense of the vector. `abs()` is a function of the math library of Visual C/C++ and can be found in every compiler package you use. It returns the absolute value of its parameter.



**Figure D.13:** Angle of dot product

The enclosed angle between  $\underline{U}$  and the x-axis should be alpha. The enclosed angle between  $\underline{V}$  and the x-axis should be beta. So we can say:

$$\cos\alpha = u_x / ||U||$$

$$\sin\alpha = u_y / ||U||$$

$$\cos\beta = v_x / ||V||$$

$$\sin\beta = v_y / ||V||$$

You can calculate the magnitude of vectors by using the Pythagorean theorem:

$$x_- + y_- = m$$

So the length (magnitude) of the vector is retrieved by

$$||V|| = \sqrt{(x_- + y_-)}$$

The angle between  $\underline{U}$  and  $\underline{V}$  is the difference between alpha and beta.

$$\theta = \alpha - \beta$$

The law of cosines states that for a triangle such as the one formed by the vectors  $\underline{U}$ ,  $\underline{V}$ , and  $\underline{W}$ , where  $||U||$ ,  $||V||$ , and  $||W||$  are the lengths of the sides of the triangle, the following is true:

$$\underline{U} * \underline{V} = ||U|| * ||V|| * \cos(\alpha - \beta)$$

```

= ||U|| * ||V|| * (cos alpha * cos beta + sin alpha * sin beta)
= ||U|| * ||V|| * (ux / ||U|| * vx / ||V|| + uy / ||U|| * vy / ||V||)
= ||U|| * ||V|| * (ux * vx + uy * vy) / (||U|| * ||V||)
= ux * vx + uy * vy

```

I would like to solve that for the angle between the two vectors above:

$$\cos(\theta) = \underline{U} \cdot \underline{V} / ||U|| * ||V||$$

$$U (1.5, 3)$$

$$V (4, 1)$$

$$||U|| = \sqrt{ux_+ + uy_-}$$

$$= \sqrt{2.25 + 9}$$

$$= 3.4$$

$$||V|| = \sqrt{vx_+ + vy_-}$$

$$= \sqrt{16 + 1}$$

$$= 4.12$$

$$U \cdot V = ux * vx + uy * vy$$

$$= 1.5 * 4 + 3 * 1$$

$$= 9$$

$$\cos(\theta) = 9 / 3.4 * 4.12$$

$$\cos(\theta) = 0.64$$

$$\theta = 49.99^\circ$$

Let's do that for a 3-D vector.

For a triangle such as the one formed by the vectors  $\underline{U}$ ,  $\underline{V}$ , and  $\underline{W}$ , where  $||U||$ ,  $||V||$ , and  $||W||$  are the lengths of the sides of the triangle, the following is true:

$$W = U - V$$

$$U - V = U + V - 2 * U * V$$

To solve it after  $U - V$ :

$$2 * U * V = U + V - (U - V)_-$$

To find the magnitude of the vectors, the following is true:

$$||U||_- = ux_- + uy_- + uz_-$$

$$||V||_- = vx_- + vy_- + vz_-$$

$$||U - V||_- = (ux - vx)_- + (uy - vy)_- + (uz - vz)_-$$

To solve the equation

$$2 * U * V = ux_- + uy_- + uz_- + vx_- + vy_- + vz_- - (ux - vx)_- - (uy - vy)_- - (uz - vz)_-$$

$$= ux_- + uy_- + uz_- + vx_- + vy_- - ux_- - vx_- + 2 * ux * vx - uy_- - vy_- + 2 * ux * vy - uz_- - vz_- + 2 *$$

$$\begin{aligned}
 & u_z * v_z \\
 & = 2 * u_x * v_x + 2 * u_y * v_y + 2 * u_z * v_z \\
 & = 2 * (u_x * v_x + u_y * v_y + u_z * v_z)
 \end{aligned}$$

that leads to

$$\underline{U} * \underline{V} = u_x * v_x + u_y * v_y + u_z * v_z$$

To get the angle between those vectors you might use

$$\cos(\theta) = \underline{U} * \underline{V} / ||\underline{U}|| * ||\underline{V}||$$

$$U (1.5, 3, 4)$$

$$V (4, 1, 2)$$

$$||\underline{U}|| = \sqrt{u_x^2 + u_y^2 + u_z^2}$$

$$= \sqrt{2.25 + 9 + 16}$$

$$= 4$$

$$||\underline{V}|| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

$$= \sqrt{16 + 1 + 4}$$

$$= 4.4$$

$$\underline{U} * \underline{V} = u_x * v_x + u_y * v_y + u_z * v_z$$

$$= 1.5 * 4 + 3 * 1 + 4 * 2$$

$$= 17$$

$$\cos(\theta) = 17 / 4 * 4.4$$

$$\cos(\theta) = 0.96$$

$$\theta = 15.00^\circ$$

The dot product is used to find the angle between two vectors; as such, it is used for shading polygons and producing many other effects.

#### CROSS PRODUCT

Taking the cross product of any two vectors forms a third vector perpendicular to the plane formed by the first two. This one is called a normal vector and points in the direction of the plane formed by the other two vectors.

The cross product is used to determine which way polygons are facing. It uses two of the polygon's edges to generate a normal. Thus, it can be used to generate a normal to any surface for which you have two vectors that lie within the surface. Unlike the dot product, the cross product is not commutative.

$$\underline{U} \times \underline{V} = -(\underline{V} \times \underline{U}).$$

#### NOTE

A normal vector is not the same as a normalized vector. A normalized vector has a magnitude of 1, as you will see in a few seconds.

The magnitude of the cross product of  $\underline{U}$  and  $\underline{V}$ ,  $||\underline{U} \times \underline{V}||$ , is given by  $||\underline{U}|| * ||\underline{V}|| * \sin(\theta)$ . The direction of the resultant vector is perpendicular to both  $\underline{U}$  and  $\underline{V}$ .

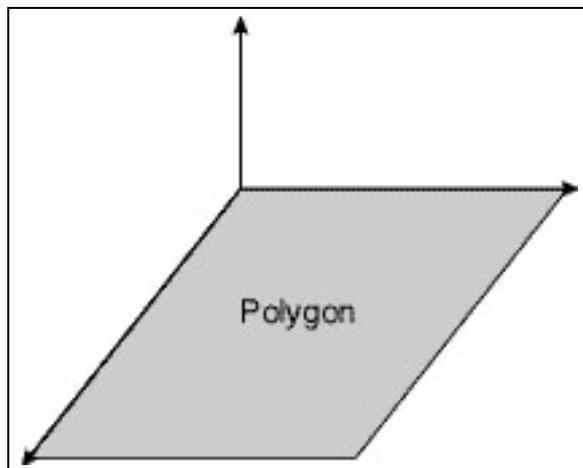


Figure D.14: Cross product

You get the sine value if you divide the cross product by the magnitudes of both vectors:

$$\sin(\theta) = \underline{U} \times \underline{V} / ||\underline{U}|| * ||\underline{V}||$$

You shouldn't use this to find a angle between the two vectors because the dot product is faster.

## UNIT VECTOR

There is another class of vectors that is important for us: the unit vector. A unit vector has a magnitude of 1. If you combine that with the free vector, you get a vector with all possible directions and a magnitude of one. All possible free and unit vectors form a sphere with their final points. If you define the starting point as the origin of, for example, a camera, you might rotate that camera by altering the vector values.

How do we calculate a unit vector or normalize a vector? To calculate a unit vector, divide the vector by its magnitude or length.

$$\underline{N} = \underline{V} / ||\underline{V}||$$

As shown above, you can calculate the magnitude of vectors by using the Pythagorean theorem:

$$x_+y_+z_+ = m_-$$

The length of the vector is retrieved by

$$\|V\| = \sqrt{x_+ + y_+ + z_+}$$

It's the square root of the Pythagorean theorem. The magnitude of a vector has a special symbol in mathematics, a capital letter designated with two vertical bars:  $\|V\|$ . A function performing this task might look like:

```
D3DXVECTOR3* D3DXVec3Normalize(D3DXVECTOR3* v1, D3DXVECTOR3* v2)
{
    D3DXVECTOR3 tv1 = (D3DXVECTOR3)*v1;
    D3DXVECTOR3 tv2 = (D3DXVECTOR3)*v2;
    tv1 = tv2/(float) sqrt(tv2.x * tv2.x + tv2.y * tv2.y + tv2.z * tv2.z);
    v1 = &tv1;
    return (v1);
}
```

It divides the vector by its magnitude, which is retrieved by the square root of the Pythagorean theorem.

### NOTE

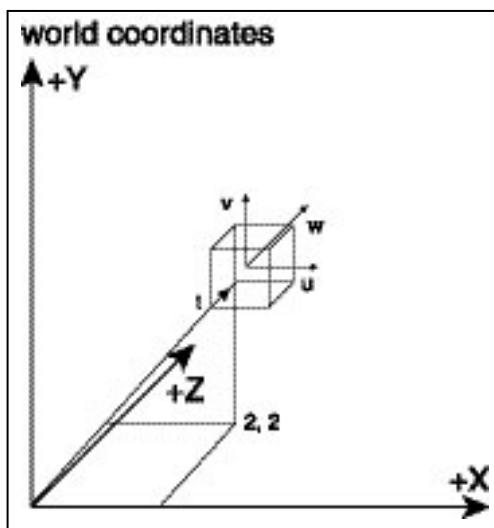
`sqrt()` is a mathematical function from the math library of Visual C/C++ provided by Microsoft. Other compilers should have a similar function.

## MATRICES

Matrices are rectangular arrays of numbers. A 4-by-4 world matrix contains four vectors, which represent the world space coordinates of the x, y, and z unit axis vectors, and the world space coordinate, which is the location of these axis vectors:

ux	uy	uz	0
vx	vy	vz	0
wx	wy	wz	0
tx	ty	tz	1

The u, v, and w vectors represent the so-called rigid body. This is the matrix that defines the mapping from object space to world space. Graphically, it looks like this:

Figure D.15: The  $u$ ,  $v$ , and  $w$  vector

To describe the position of this cube, the matrix has to look like:

1,	0,	0,	0
0,	1,	0,	0
0,	0,	1,	0
2,	2,	2,	1

The cube is oriented like the coordinate system. The first row contains the world space coordinates of the local x-axis. The second row contains the local y-axis, and the third row contains the world space coordinates of the local z-axis. The vectors are unit vectors whose magnitudes are 1. The last row contains the world space coordinates of the object's origin. You might translate the cube with these.

A special matrix is the identity matrix:

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

The identity matrix represents a set of object axes that are aligned with the world axes. The world x-coordinate of the local x-axis is 1, the world y- and z-coordinates of the local x-axis are 0, and the origin vector is (0, 0, 0). So the local model x-axis lies directly on the world x-axis. The same is true for the local y- and z-axes. So it's a "get back to the roots" matrix. If an object's position in model space corresponds to its position in world space, simply set the world transformation matrix to the identity matrix. This matrix and all other matrices in Direct3D could be accessed like this:

```
D3DMATRIX mat;
mat._11 = 1.0f; mat._12 = 0.0f; mat._13 = 0.0f; mat._14 = 0.0f;
mat._21 = 0.0f; mat._22 = 1.0f; mat._23 = 0.0f; mat._24 = 0.0f;
mat._31 = 0.0f; mat._32 = 0.0f; mat._33 = 1.0f; mat._34 = 0.0f;
mat._41 = 0.0f; mat._42 = 0.0f; mat._43 = 0.0f; mat._44 = 1.0f;
```

## MULTIPLICATION OF A MATRIX WITH A VECTOR

A typical transformation operation is a 4-by-4 matrix multiplication operation. A transformation engine multiplies a vector representing 3-D data, typically a vertex or a normal vector, by a 4-by-4 matrix. The result is the transformed vector. This is done with standard linear algebra:

Transform Matrix	Original Vector		Transformed Vector
a b c d	x	=	x'
e f g h	y	=	y'
i j k l	z	=	z'
m n o p	w	=	w'

$$\begin{array}{cccc|ccccc}
 a & b & c & d & x & & ax + by + cy + dw & & x' \\
 e & f & g & h & y & = & ex + fy + gz + hw & = & y' \\
 i & j & k & l & z & & ix + jy + kz + lw & = & z' \\
 m & n & o & p & w & & mx + ny + oz + pw & = & w'
 \end{array}$$

Before a vector can be transformed, a transform matrix must be constructed. This matrix holds the data to convert the vector data to the new coordinate system. Such an interim matrix must be created for each action (scaling, rotation, and transformation) that will be performed on the vector. Those matrices are multiplied together to create a single matrix that represents the combined effects of all of those actions (*matrix concatenation*). This single matrix, called the *transform matrix*, can be used to transform one vector or one million vectors. The time to set it up amortizes by the ability to reuse it.

## MATRIX ADDITION AND SUBTRACTION

It is simple to add two matrices:

$$\begin{array}{rrrr|rrrr|rrrr}
 1 & 2 & 3 & 4 & 17 & 18 & 19 & 20 & 18 & 20 & 22 & 24 \\
 5 & 6 & 7 & 8 & + & 21 & 22 & 23 & 24 & = & 26 & 28 & 30 & 32 \\
 9 & 10 & 11 & 12 & & 25 & 26 & 27 & 28 & & 34 & 36 & 38 & 40 \\
 13 & 14 & 15 & 16 & & 29 & 30 & 31 & 32 & & 42 & 44 & 46 & 48
 \end{array}$$

You just add the first number in the first column of the first matrix with the first number in the first column of the second matrix, and so on. It works the same with subtraction.

## MATRIX MULTIPLICATION

It is best to show an example:

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array} \times \begin{array}{cccc} 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \end{array}$$

The idea is to multiply the first row of the left matrix with the first column of the second and add the result. Then, do the same with the second column of the second matrix, then with the third and with the fourth.

$$1 * 17 + 2 * 21 + 3 * 25 + 4 * 29 = 1 * 18 + 2 * 22 + 3 * 26 + 4 * 30 \text{ etc.}$$

Then we start multiplying the second row of the first matrix with the first column of the second matrix and add the result. We go on by multiplying the second row of the first matrix with the second column of the second matrix and so on. After all, we will get the 16 values of the 4-by-4 matrix.

## TRANSLATION MATRIX

To move an object in 3-D space, you normally manipulate the fourth row of the world matrix of the object (object matrix) or you multiply the world matrix with a translation matrix:

$$\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{array}$$

Instead of holding the coordinate, a translation matrix holds how far the object has to be moved. The object matrix is used to store x-, y-, and z-values.

## SCALING MATRIX

This type of matrix will scale a coordinate. So it's useful to make an object smaller or bigger.

$$\begin{array}{cccc} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{array}$$

It's also possible to manipulate the three values directly without using a matrix.

## ROTATION MATRICES

There are three rotation matrices to rotate about the y-, x-, and z-axes.

### ROTATION ABOUT THE Y-AXIS

The Direct3D function D3DXMatrixRotationY() rotates about the y-axis. We don't have the source, but it might look like the following code:

```
VOID D3DXMatrixRotationY( D3DXMATRIX* mat, FLOAT fRads )
{
    D3DXMatrixIdentity(mat);
    mat._11 = cosf( fRads );
    mat._13 = -sinf( fRads );
    mat._31 = sinf( fRads );
    mat._33 = cosf( fRads );
}

=
cosf fRads      0      -sinf fRads      0
0              0          0
sinf fRads      0      cosf fRads      0
0              0          0
```

### ROTATION ABOUT THE X-AXIS

D3DXMatrixRotationX() rotates the objects about the x-axis, where fRads equals the amount you want to rotate. It might look like this in source:

```
VOID D3DXMatrixRotationX( D3DXMATRIX* mat, FLOAT fRads )
{
    D3DXMatrixIdentity(mat);
    mat._22 = cosf( fRads );
    mat._23 = sinf( fRads );
    mat._32 = -sinf( fRads );
    mat._33 = cosf( fRads );

}

=
1      0          0          0
0      cos fRads   sin fRads  0
0      -sin fRads  cos fRads  0
0      0          0          0
```

### ROTATION ABOUT THE Z-AXIS

D3DXMatrixRotationZ() rotates the objects about the z-axis, where fRads equals the amount you want to rotate. Source code for this function might look like:

```
VOID D3DXMatrixRotationZ( D3DXMATRIX* mat, FLOAT fRads )
{
    D3DXMatrixIdentity(mat);
    mat._11 = cosf( fRads );
    mat._12 = sinf( fRads );
    mat._21 = -sinf( fRads );
    mat._22 = cosf( fRads );
}

=
cosf fRads    sinf fRads    0    0
-sinf fRads    cos fRads    0    0
0            0            0    0
0            0            0    0
```

These matrices are concatenated by Direct3D. A very important thing to remember is that matrix multiplication is not commutative. That means  $[a] * [b] \neq [b] * [a]$ . The formula for transformation is

$$|W| = |M| * |T| * |X| * |Y| * |Z|$$

where M is the model's matrix, T is the translation matrix, and X, Y, and Z are the rotation matrices. The resulting matrix W is used by Direct3D internally.

*This page intentionally left blank*

# **APPENDIX E**

## **GAME PROGRAMMING RESOURCES**

I would like to recommend a few Internet resources, which you might want to check out on a regular basis to be in sync with the fast development of game programming.

## GENERAL

[www.voodooextreme.com](http://www.voodooextreme.com). Gives you a lot of info and screen shots of the newest games. This might help you to keep an eye on what is possible in game programming. Sometimes it provides you with leaked drivers and information on game development, but it's not for developers only.

[www.bluesnews.com](http://www.bluesnews.com). Follows the same rules as Voodoo Extreme. Both are nice sites; they are like your daily newsletter, entertaining and informative.

[www.reactorcritical.com](http://www.reactorcritical.com). Gives you a lot of hardware information, which is very useful for game programming . . . know your hardware.

[www.demonews.com/news](http://www.demonews.com/news). Provides information on new game demos and demos from the coder scene.

[www.realtimerendering.com](http://www.realtimerendering.com). A list of interesting links on graphics programming.

## DIRECTX GRAPHICS

[msdn.microsoft.com/directx](http://msdn.microsoft.com/directx). This is the source. Read the articles of Philip Taylor.

[www.directxgraphics.net](http://www.directxgraphics.net). This is the accompanying Web site for this book. You will find updates and corrections here. Besides, there are always a few interesting tutorials on Direct3D.

[www.gamedev.net](http://www.gamedev.net). You will find a lot of DirectX Graphics tutorials here. A few of them are written by me.

[www.flipcode.com](http://www.flipcode.com). A Web site for game programmers. Every day I look at the image, code, and tip of the day.

[hometown.aol.com/billybop7/index.htm](http://hometown.aol.com/billybop7/index.htm). The Web site of William Chin, alias billybop, who, together with Peter Kovach, has written the roadrage engines used in *Inside Direct3D* from Microsoft Press.

[www.zen-x.net](http://www.zen-x.net). ZEN-X is a DirectX Graphics engine with source. You can learn a lot by checking the provided source.

[www.cbloom.com](http://www.cbloom.com). The Web site of Charles Bloom, who provides a few very interesting papers and demos to download.

[www.dx-game.com](http://www.dx-game.com). Provides useful examples for DirectX Graphics starters.

[members.home.net/eckiller/resources.htm](http://members.home.net/eckiller/resources.htm). The Web site of eckiller, with a few nice DirectX Graphics samples and useful source code.

[www.mrgamemaker.com](http://www.mrgamemaker.com). Great Direct3D tutorials.

[www.swissquake.ch/chumbalum-soft/](http://www.swissquake.ch/chumbalum-soft/). The home of MilkShape 3D, the best modeling tool you can get.

[discuss.microsoft.com/archives/directxdev.html](http://discuss.microsoft.com/archives/directxdev.html). The most important know-how database you can use.

## FAQ

[www.faqs.org/faqs/graphics/algorithms-faq/](http://www.faqs.org/faqs/graphics/algorithms-faq/). The newest FAQ on graphics programming can be found here.

*This page intentionally left blank*

# INDEX

## Symbols

1steps.cpp application classes, 104

2-D, 78

bit arrays, 304–308

brute force, 304

collision detection, 302–304

sprite bounds, 309–313

3-D

collision detection, 319–320

BSP (Binary Space Partitioning), 323–324

calculating distance, 322–323

portals, 320–322

sliding off walls, 323–324

Direct3D pipeline, 75–77

environments, 428–440

file formats, 213

group object processing, 336

mathematics, 292–294

mesh collision detection, 324

bounding volumes, 324–325

concave models, 326–332

convexity of models, 325

intersection, 326

points, 464–467

transformation pipeline, 77–79

## A

AABB (axis aligned bounding box), 327

abstraction (C++), 374–377

addition. *See also* mathematics

matrices, 480

vectors, 469–471

ADD SIGNED texture, 168

administration, multiple textures, 201

Advance function, 319

AI (artificial intelligence), 77

air resistance, 297

alpha blending, 148–151, 160

alpha operations, 172–173

ambient light, 98. *See also* lighting

American National Standards Institute (ANSI), 58

angles, cosines, 323

animation, 74–75

code, 104–119

dark, 161

DeleteDeviceObjects(), 119

depth buffering, 100–104

FinalCleanup(), 119

FrameMove(), 113–117

InitDeviceObjects(), 110

InvalidateDeviceObjects(), 118–119

lighting, 96

material, 97

models, 97–99

vertex colors, 99–100

matrices, 80–82

projection matrix, 95–96

view matrix, 86–95

world matrix, 80–86

OneTimeSceneInit(), 105–110

quaternions, 119–120

Render(), 117–118

RestoreDeviceObjects(), 110–113

transformation

math, 80

pipeline, 77–79

X files, 233–235

animation.cfg file, 253–255

anisotropy, 144–146, 207

ANSI (American National Standards Institute), 58

anti-aliasing

full-scene, 146–148

textures, 140–141

applications

classes

1steps.cpp, 104

Direct3D, 19–20

ColDet, 327, 332–334

RAPID, 327

Windows

components, 347

features, 347–361

programming, 346

ApplyEnvironmentMap(), 183–186

applying X files, 235–241

architecture, 5

DirectX, 37–38

Unified Memory, 46

arguments, default functions, 395

Arial fonts, 57. *See also* fonts  
arrays, bit, 304–313  
assigning texture coordinates, 31  
attributes, surface, 257  
audio, Quake III, 262–263  
automatic transparent static marking, 319  
axis aligned bounding box (AABB), 327  
axis sorts, 314–316

## B

back buffer, 43  
base vector regeneration, 88  
basic.cpp file, 38–39  
BeingScene(), 52  
bilinear texture filtering, 143  
Binary Space Partitioning (BSP), 323  
bit arrays, 304–313  
bitmaps, 131  
blending  
    alpha blending, 148–151  
    dot product texture blending, 194–195, 207  
    textures, 155, 161–164  
border color texture-addressing mode, 135–136  
bound vectors, 467–468  
bounding volumes, 302, 324–325  
boxes  
    dividing, 329  
    oriented box intersections, 330–332  
brushes, 353  
brute force, 304–308  
BSP (Binary Space Partitioning), 323  
buffers  
    back, 43  
    depth, 100–104  
    filling, 47  
    frame, 148–151  
    index, 65–72  
    stencil, 207  
    vertices, 39, 43  
    z-buffer, 74  
BuildDeviceList() function, 410–428  
building worlds, 212–213  
bump mapping, 182–183, 207  
    ApplyEnvironmentMap(), 183–186  
ConfirmDevice(), 193–194

InitBumpMap(), 187–191  
Render(), 191–194  
BUMPVERTEX structure, 183

## C

C programming rules, 16–18, 394–400  
C++, 14, 373–374  
    abstraction, 374–377  
    classes, 19, 377–378  
    declaring, 379–384  
    hierarchies/inheritance, 386–391  
    constructors, 384  
    destructors, 384–386  
    encapsulation, 378–379  
    enhancements to C, 394–400  
    inline functions, 393–394  
    polymorphism, 392–393  
    programming rules, 16–18  
    virtual functions, 391–392  
CalculateBit Array function, 305  
CalculateBounds function, 310  
calculations  
    AABB (axis aligned bounding box), 327  
    distance, 322–323  
    frame time, 296  
    lighting, 10  
    window sizes, 355  
camera rotation  
    camera axis, 88–92  
    quaternions, 92–95  
cascades, texture blending, 155  
CCollidableMesh class, 333  
CD3DApplication class, 38  
Change Device dialog box, 8, 9  
CheckDeviceMultiSampleType(), 147  
clamp texture-addressing mode, 134–135  
classes  
    C++, 19, 377–378  
    declaring, 379–384  
    hierarchies/inheritance, 386–391  
CCollidableMesh, 333  
CD3DApplication, 38  
fonts, 55–57  
PhysicalSprite, 309  
textures, 55–57

- windows, 251–354  
`clear()`, 103  
CLIENTCREATESTRUCT structure, 355  
code  
    animation, 104–119  
    common files framework. *See* common files framework  
    developing, 16–18  
    inheriting, 387  
    return, 21  
    styles, 18–20  
ColDet, 327, 332–336  
collision detection, 302  
    2-D, 302–304  
        bit arrays, 304–313  
        brute force, 304–308  
        sprite bounds, 309–313  
    3-D, 319–320  
        BSP (Binary Space Partitioning), 323–324  
        calculating distance, 322–323  
        portals, 320–322  
        sliding off walls, 323–324  
group processing, 313–319  
mesh (3-D), 324  
    bounding volumes, 324–325  
    concave models, 326–332  
    convexity of models, 325  
    intersection, 326  
SOLID (Software Library for Interference Detection), 326  
collision reaction, 334–336  
CollisionModel3D object, 333  
ColorKey parameter, 59  
colors  
    icons, 368  
    material diffuse color, 161–170  
    vertices, 99–100  
COM (component object model), 8, 13–18  
commands. *See also* functions  
    shader, 257  
    Start menu  
        Control Panel, 12  
    Direct3D, 20  
    DirectX, 12  
    Settings, 12  
common architecture, DirectX, 37–38  
common files framework, 36–37, 402–407  
    Create() method, 407–409  
    BuildDeviceList(), 410–428  
    CreateWindow(), 428  
    Direct3DCreate8(), 409–410  
    DXUtil\_Timer(), 440–444  
    Initialize3DEnvironment(), 428–440  
    OneTimeSceneInit(), 428  
    MsgProc() function, 455–461  
    Run() function, 444–455  
component object model. *See* COM  
components  
    COM, 16–18  
        windows, 347  
    concave models, 326–332  
configuration  
    ambient light, 98  
    dark maps, 157  
ConfirmDevice()  
    bump mapping, 193–194  
    cubic environment mapping, 181–182  
ConfirmDevice() method, 39  
const variable, 396  
constants, 19  
constructors  
    C++, 384  
        texture font class, 56  
Control Panel command (Start menu), 12  
controlling devices, 12  
conventions, naming, 19  
converting vertices, 51  
convexity of models, 325  
coordinates, textures, 129–131  
copying fonts, 57  
cosines, angles, 323  
Create() method, 333  
    BuildDeviceList(), 410–428  
    CreateWindow(), 428  
    Direct3DCreate8(), 409–410  
    DXUtil\_Timer(), 440–444  
    Initialize3DEnvironment(), 428–440  
    OneTimeSceneInit(), 428  
CreateFileBasedNormalMap(), 195  
CreateModel(), md3.cpp, 271–278

CreateTextures(), md3.cpp, 278–282  
CreateVertexBuffer() method, 46  
CreateVB(), md3.cpp, 282–283  
CreateWindow() function, 354–357, 428  
cross products, 476–477  
cubic environment mapping, 175–182, 207  
cursors, 370  
custom sounds, Quake III, 262–263  
CUSTOMVERTEX structure, 44, 50  
cvVertices structure, 50  
cylinders, 320–321

## D

D3D\_SDK\_VERSION parameter, 409  
d3dapp.cpp file, 37  
D3DCreateFromFileEx(), 58  
D3DFMT\_A8RG8B8 flag, 187  
d3dfont.cpp file, 37  
D3DFVF\_XYZ vertex format, 333  
D3DLIGHT8 structure, 98  
D3DMatrixRotationY(), 84  
D3DMatrixTranslation(), 84  
D3DMCS\_COLOR2 parameter, 162  
D3DPBLENDCAPS\_\* flag, 151  
D3DPT\_TRIANGLELIST, 109  
D3DRS\_DIFFUSEMATERIALSOURCE parameter, 162  
D3DTOP\_MODULATE, 159  
D3DTOP\_MODULATE2X, 159  
D3DTOP\_MODULATE4X, 159  
D3DTS\_WORLD flag, 86  
d3dutil.cpp file, 37  
D3DUtil\_CreateTexture() flag, 58, 187  
D3DUtil\_InitMaterial(), 97  
D3DX library, 20  
D3DXMatrixMultiply(), 85  
D3DXMatrixRotationX(), 85  
D3DXMatrixRotationZ(), 85  
D3dXMatrixTranslation(), 95  
D3DXVec3Normalize(), 90, 108  
dark  
    animating, 161  
    mapping, 157–160, 164–166  
data types, Windows, 354  
DDI (Device Driver Interface), 10

deallocating memory, 55  
debugging DirectX, 20  
declarations  
    explicit shaders, 45  
    variables, 395–396  
default function arguments, 395  
defining  
    directions, 26  
    flags, 45  
    vertices, 25  
    window classes, 351–354  
deformation, 257  
DeleteDeviceObjects(), 55, 65, 119, 287–288  
DeleteTextures(), md3.cpp, 284  
DeleteVB(), md3.cpp, 284  
Delphi, 14  
depth  
    buffering, 100–104  
    ranges, 50  
DestBlendFactor flag, 149  
destructors (C++), 384–386  
detail mapping, 167–170, 207  
detection, collisions. *See* collision detection  
developing code, 16–18  
Device Driver Interface (DDI), 10  
device drivers, searching, 410–428  
DeviceObjects() method, 40  
devices  
    controlling, 12  
    InitDeviceObjects(), 57  
    pluggable software devices, 11  
dialog boxes  
    Change Device, 8, 9  
    DirectX Properties, 12  
Direct3D  
    application classes, 19–20  
    command (Start menu), 20  
    coordinate system, 24  
    HAL (hardware abstraction layer), 9–11  
Direct3D pipeline, 75–77  
Direct3DCreate8() function, 409–410  
directions, defining, 26  
DirectX  
    ColDet, 332–334  
    3-D object group processing, 336

collision reaction, 334–336  
command (Start menu), 12  
common architecture, 37–38  
debugging, 20  
Properties dialog box, 12  
resources, 486–487  
directx.ico file, 37  
displaying windows, 357–358  
distance, calculating, 322–323  
dividing  
    boxes, 329  
    rooms, 321  
DLL (dynamic-link library), 13  
Doom, marine head, 249–250  
dot products, 473–476  
    blending, 194–195, 207  
    InitDeviceObjects(), 195–198  
    Render(), 198–200  
Dot3 product bump mapping, 39  
DrawIndexedPrimitive(), 66  
DrawPrimitive(), 53, 64  
drifting, 88  
DXUtil\_FindMediaFile(), 58  
DXUtil\_Timer() function, 440–444  
dynamic objects, 318  
dynamic-link library (DLL), 13

## E

encapsulation (C++), 378–379  
EndScene(), 52  
enumeration (C++), 397  
environments, 173  
    3-D, 428–440  
    cubic environment mapping, 175–182  
        Quake III, 259  
    spherical environment mapping, 173–175  
errors, rendering z-buffers, 102  
exiting, 55  
explicit shader declarations, 45  
extending X files, 245

## F

faces, geometry, 27–29  
fans, triangles, 54

FAQs (frequently asked questions), 487  
fields  
    hInstance, 351  
    IParam, 358  
    wParam, 358  
files  
    3-D formats, 213  
    basics.cpp, 36, 38–39  
    common files framework, 36–37. *See also* common files framework  
    d3dapp.cpp, 37  
    d3dfont.cpp, 37  
    d3dutil.cpp, 37  
    directx.ico, 37  
    lib, 37  
    md3.cpp  
        CreateModel(), 271–278  
        CreateTextures(), 278–282  
        CreateVB(), 282–283  
        DeleteTextures(), 284  
        DeleteVB(), 284  
        Render(), 282–283  
    md3.h, 264–271  
    md3view.cpp, 285  
        DeleteDeviceObjects(), 287–288  
        FinalCleanup(), 288  
        InitDeviceObjects(), 286  
        MsgProc(), 288  
        OneTimeSceneInit(), 286  
        Render(), 287  
    Quake III, 248–249  
        animation.cfg, 253–255  
        custom sounds, 262–263  
        .md3, 249–253  
        .md3 formats, 263–289  
        shaders, 256–262  
        .skin, 255–256  
    resource.h, 37  
    .wav, 262–263  
    winmain.rc, 36  
X files  
    building worlds, 212–213  
    formats, 213–235  
filling buffers, 47  
filtering

anisotropic, 144–146  
linear texture, 143–144  
textures, 140–141  
`FinalCleanup()`, 55, 65, 119, 288  
flags  
    D3DFMT\_A8RG8B8, 187  
    D3DPBLENDCAPS\_\*, 151  
    D3DTS\_WORLD flag, 86  
    D3DUtility\_CreateTexture(), 187  
DestBlendFactor, 149  
flexible vertex format, 45  
format, 44  
SourceBlendFactor, 149  
WS\_VISIBLE, 357  
flexible vertex format (FVF), 45, 52  
fonts, 37, 55–57  
Format parameter, 59  
formats  
    .md3, 263–264  
        .md3.cpp file, 271–285  
        .md3.h file, 264–271  
        .md3view.cpp file, 285–289  
files  
    3-D, 213  
    X files, 213–235  
flags, 44  
fonts, 57  
icons, 368  
scripts, 367  
version control resources, 369  
vertext buffers, 44  
windows, 354–357, 428  
formulas  
    alpha-blending, 160  
    object\_position\_collision\_point, 335  
`FrameMove()`, 40, 51, 113–117  
frames  
    buffers, 148–151  
    key, 249  
    rates, 91  
    time, 296  
framework, common files. *See* common files framework  
free vectors, 467–477  
frequently asked questions (FAQs), 487  
friction

kinetic, 298–299  
static, 297  
functions  
    Advance, 319  
    BuildDeviceList(), 410  
    CalculateBitArray, 305  
    CalculateBounds, 310  
    collision tests, 307  
    Create, 333  
    CreateWindow(), 354–357, 428  
    default arguments, 395  
    Direct3DCreate8(), 409–410  
    DXUtil\_Timer(), 440–444  
    GetBitArray, 306  
    GetCommandLine(), 350  
    GetMessage(), 358  
    inline (C++), 393–394  
    LoadCursor(), 353  
    lpfnWndProc, 351  
    minimum/maximum, 306  
    MsgProc(), 455–461  
    OneTimeSceneInit(), 428  
    overloading, 397  
    parameters, 19  
    PeekMessage(), 359  
    Run(), 444–455  
    ShowWindow(), 357  
    VectorsDisjoint, 311  
    virtual (C++), 391–392  
    WinProc(), 350  
    WndProc(), 361  
FVF (flexible vertex format), 45, 52

## G

game physics (Amir Geva), 292–299  
GDI (Graphics Device Interface), 9  
general resources, 486  
geometry, 24–26  
    BSP (Binary Space Partitioning), 323  
    faces, 27–29  
    initializing, 428  
    normals, 29  
    orientation, 26–27  
`GetBitArray` function, 306  
`GetCommandLine()` function, 350

GetMessage() function, 358  
 Geva, Amir, 292–299  
 glow mapping, 166–167  
 Gouraud shading, 29–30  
 GPU (graphics processing unit), 10  
 graphic processors, 76  
 Graphics Device Interface (GDI), 9  
 grids, methods, 316–317  
 group processing, collision detection, 313–319

**H**

HAL (hardware abstraction layer), 8–11  
 Hamilton, Sir William Rowan, 92  
 hardware abstraction layer. *See* HAL  
 hardware emulation layer. *See* HEL  
 Header template, X files, 215–216  
 height, 197, 355  
 HEL (hardware emulation layer), 8  
 hierarchies (C++), 386–391  
 hInstance field, 351  
 history of Direct3D/DirectX graphics, 4–6

**I**

icons, 368  
 IDirect3DDevice8 :: SetTexture(), 62  
 import libraries, 17  
 In-Memory layout (C++), 16  
 index buffers, 65–72  
 indices, D3DPT\_TRIANGLELIST, 109  
 inheritance (C++), 386–391  
 InitBumpMap(), 187–191  
 InitDeviceObjects(), 57, 110  
     dot product texture blending, 195–198  
     md3view.cpp, 286  
     X files, 242–243  
 InitDeviceObjects() method, 4I  
 initializing  
     3-D environments, 428–440  
     geometry, 428  
     matrices, 42  
 inline functions (C++), 393–394  
 inserting scripts, 367  
 int values, 306  
 interaction, Windows, 346

**I**  
interfaces

    COM (component object model), 16–18  
     Device Driver Interface (DDI), 10  
     GDI (Graphics Device Interface), 9  
     inheriting, 390–391  
     language-independent, 14  
     legacy, 13  
     MDI (multiple document interface), 355  
     publishing, 38  
 intersections, 306, 326  
     oriented box intersections, 330–332  
     triangles, 332  
 InvalidateDeviceObjects(), 64, 72, 118–119  
 IO triangles, 27–29

**K**

key frames, 249  
 kinetic friction, 298–299

**L**

languages, 14–18. *See also* C; C++  
 legacy interfaces, 13  
 lib files, 37  
 libraries, 17, 20  
 lighting, 96  
     calculations, 10  
     mapping. *See* dark mapping  
     materials, 97  
     models, 97–99  
     vertex colors, 99–100  
 linear texture filtering, 143–144  
 lists, triangles, 65–72  
 LoadCursor() function, 353  
 lock/unlock methods, 39  
 LockRect(), 189  
 LOD (level-of-detail), 6  
 LOOK vectors, 27, 86  
 lParam fields, 358  
 lpfnWndProc function, 351

**M**

M-strDeviceStats parameter, 64  
 m\_strFrameStats parameter, 64  
 macros, 16, 37

- MAKEINTRESOURCE, 352
- WS\_OVERLAPPEDWINDOW, 355
- MAKEINTRESOURCE macro, 352
- management
  - multiple textures, 201
  - resources, 46
- mapping
  - bump, 182–194, 207
  - cubic environment, 207
  - dark, 157–160, 164–166
  - detail, 167–170, 207
  - Dot3 product bump mapping, 39
  - environment, 111, 173, 259. *See also* environment mapping
  - glow, 166–167
  - mipmaps, 58, 142
  - per-pixel vectors, 195–198
  - textures, 24–26, 128–129
    - faces, 27–29
    - normals, 29
    - objects, 31–33
    - orientation, 26–27
- materials
  - diffuse colors, 161–170
  - lighting, 97
- mathematics
  - alpha blending, 150
  - matrices, 478–480
    - addition/subtraction, 480
    - multiplication, 480–481
    - rotation, 482
    - scaling, 481
    - transformation, 481
  - points in 3-D, 464–467
  - transformation, 80
  - vectors, 467
    - bound vectors, 467–468
    - free vectors, 468–477
    - unit vectors, 477–478
- matrices, 80–82
  - initializing, 42
  - matYaw, 90
  - projection, 95–96
  - transformation, 229–233
  - view
- camera axis, 88–92
- quaternions, 88–92
- viewing, 43
- world, 82–86
- matYaw matrix, 90
- maximum functions, 306
- md3.cpp, 271–285
  - CreateTextures(), 278–282
  - DeleteVB(), 284
  - files
    - CreateModel(), 271–278
    - CreateVB(), 282–283
    - DeleteTextures(), 284
    - Render(), 282–283
  - .md3 files, 249–253, 263–264
  - md3.h file, 264–271
  - md3view.cpp
    - DeleteDeviceObjects(), 287–288
    - files, 285
    - FinalCleanup(), 288
    - MsgProc(), 288
    - OneTimeSceneInit(), 286
    - Render(), 287
    - InitDeviceObjects(), 286
- MDI (multiple document interface), 355
- memory
  - deallocating, 55
  - pools, 47
  - vertex buffers, 43
- mesh (3-D) collision detection, 324
  - bounding volumes, 324–325
  - concave models, 326–332
  - convexity of models, 325
  - intersection, 326
- Mesh template, X files, 216–218
- MeshMaterialList object, 218–221
- messages
  - pumps, 360
  - windows, 358–361
- methods
  - ApplyEnvironmentMap(), 183–186
  - CheckDeviceMultiSampleType(), 147
  - clear(), 103
  - COM (component object model), 14–16
  - ConfirmDevice(), 39

- bump mapping, 193–194  
cubic environment mapping, 181–182
- Create()  
    BuildDeviceList(), 410–428  
    CreateWindow(), 428  
    Direct3DCreate8(), 409–410  
    DXUtil\_Timer(), 440–444  
    Initialize3DEnvironment(), 428–440  
    OneTimeSceneInit(), 428
- CreateFileBasedNormalMap(), 195
- CreateVertexBuffer(), 46
- D3DCreateFromFileEx(), 58
- D3DMatrixRotationY(), 84
- D3DMatrixTranslation(), 84
- D3DUtility\_CreateTexture(), 58
- D3DUtility\_InitMaterial(), 97
- D3DXMatrixMultiply(), 85
- D3DXMatrixRotationX(), 85
- D3DXMatrixRotationZ(), 85
- D3dXMatrixTranslation(), 95
- D3DXVec3Normalize(), 90, 108
- DeleteDeviceObjects(), 55, 65, 119
- DeviceObjects(), 40
- DrawIndexedPrimitive(), 66
- DrawPrimitive(), 64
- DXUtil\_FindMediaFile(), 58
- FinalCleanup(), 55, 65, 119
- FrameMove(), 40, 51, 113–117
- grids, 316–317
- IDirect3DDevice8 :: SetTexture(), 62
- InitBumpMap(), 187–191
- InitDeviceObjects(), 41, 57, 110  
    dot product texture blending, 195–198  
    X files, 242–243
- InvalidateDeviceObjects(), 64, 72, 118–119
- lock/unlock, 39
- MsgProc(), 190
- naming conventions, 19
- OneTimeSceneInit(), 40–41, 105–110
- Render(), 40, 51–55, 62–64, 70–72, 117–118  
    bump mapping, 191–194  
    dot product texture blending, 198–200  
    X files, 243–245
- RenderScene(), 179–181
- RenderSceneIntoCube(), 177–179
- RestoreDeviceObjects(), 40, 41–51, 60–62, 68–70, 110–113  
    cubic environment mapping, 176–177  
    X files, 243
- SetLight(), 98
- SetStreamSource(), 64
- SetTexture(), 62
- SetTextureStageState(), 63, 128
- SetTextureStageStates(), 198
- SetTransform(), 86  
    SetVertexShader(), 64
- Milkshape 3-D Web site, 33
- minimum functions, 306
- MIP (multum in para), 141
- MipLevels parameter, 59
- mipmaps, 58, 142
- mirror texture-addressing mode, 133–134
- models  
    collision, 333  
    concave, 326–332  
    convexity of, 325  
    Gouraud, 29–30  
    lighting, 97–99  
    Quake 111  
        animation.cfg, 253–255  
        custom sounds, 262–263  
        .md3, 249–253  
        .md3 formats, 263–289  
        shader, 256–262  
        .skin, 255–256  
    wireframe, 108
- modes  
    border color texture-addressing, 135–136  
    clamp texture-addressing, 134–135  
    mirror texture-addressing, 133–134  
    MirrorOnce texture-addressing, 136–137  
    shading, 29–30  
    texture-addressing, 33  
    wrap texture-addressing mode, 132–133
- modifying  
    texture-addressing modes, 33  
    textures, 261  
    variables, 91
- monitors, tracking, 48
- MsgProc(), 190, 288, 455–461

multipass rendering, 154  
 multiple document interface (MDI), 355  
 multiple textures, 154–156  
     alpha operations, 170–173  
     bump mapping, 182–183  
     color operations, 156–157  
         animating the dark, 161  
         blending, 161–164  
         dark mapping, 157–160  
         detail mapping, 167–170  
         diffusing color, 164–166  
         glow mapping, 166–167  
     dot product texture blending, 194–195  
     environment mapping, 173  
         cubic environment mapping, 175–182  
         spherical environment mapping, 173–175  
     management, 201  
     multitexturing support, 200  
 multiplication  
     matrices, 480–481  
     vectors, 472–477  
 multisampling, 5, 146  
 multum in para (MIP), 141

**N**

naming conventions, 19  
 nCmdShow parameter, 357  
 nearest-point sampling, 143  
 Newton's laws, 294–296  
 nodes, BSP (Binary Space Partitioning), 323  
 non-D3DFMT\_V8U8 textures, 192  
 nonperpendicular normals, 30  
 normalized vectors, 89  
 normals, 29  
     wireframe models, 108  
 X files, 221  
 NVIDIA Web site, 10

**O**

object-oriented programming, 374. *See also* C++  
 object\_position\_collision\_point formula, 335  
 objects  
     3-D group object processing, 336  
     AABB (axis aligned bounding box), 327

collision detection. *See* collision detection  
 CollisionModel3D, 333  
 COM. *See* COM  
 Direct3D, 409–410  
 dynamic, 318  
 faces, 27–29  
 matrices, 80–86  
 MeshMaterialList, 218–221  
 normals, 29  
 orientation, 26–28  
 static, 317–318  
 texture mapping, 31–33, 128–131. *See also* texture  
     mapping  
 transformation pipeline, 78  
 transparent, 150  
 vectors. *See* vectors  
 OneTimeSceneInit(), 105–110, 286, 428  
 operations  
     alpha, 170–173  
     color, 156–157  
         animating the dark, 161  
         blending textures, 161–164  
         dark mapping, 157–160, 164–166  
         detail mapping, 167–170  
         glow mapping, 166–167  
 operators, overloading, 397  
 optimization  
     bit arrays, 304–312  
     C++, 394–400  
     windows, 361–366  
 orientation, geometry, 26–27  
 oriented box intersections, 330–332  
 overdraw, 100  
 overloading functions, 397  
 overview, 8

**P**

PALETTEENTRY structure, 60  
 parameters  
     ColorKey, 59  
     D3D\_SDK\_VERSION, 409  
     D3DMCS\_COLOR2, 162  
     D3DRS\_DIFFUSEMATERIALSOURCE, 162  
     Format, 59  
     functions, 19

height, 355  
m\_strDeviceStats, 64  
m\_strFrameStats, 64  
MipLevels, 59  
nCmdShow, 357  
width, 355

Pascal, 14

PeekMessage() function, 359

per-pixel vectors, InitDeviceObjects(), 195–198

perpendicular vector normal, 88

PhysicalSprite class, 309

physics, game (Amir Geva), 292–299

pipelines

- Direct3D, 75–77
- T&L (transformation and lighting), 76
- transformation, 77–79

pixels

- bit arrays, 304–312
- dot product texture blending, 194–200
- taps, 145

placement, variable declarations, 395–396

pluggable software devices, 11

points, 3-D, 464–467

polygons, textures, 129

polymorphism (C++), 392–393

pools, memory, 47

portals, 320–322

POSITION vector, 27

positioning

- object\_position\_collision\_point formula, 335
- relative, 331

prefixes, code types, 18

procedures, windows

processors, graphics, 76

programming

- COM (component object model), 16–18
- object-oriented, 374. *See also* C++
  - windows, 346

projection, matrices, 95–96

projection transformation, 79

properties, DirectX, 12

public interfaces, 38. *See also* interfaces

publishing interfaces, 38

pumps, message, 360

## Q

Quake III

files

- animation.cfg, 253–255
- custom sounds, 262–263
- .md3, 249–253
- .md3 formats, 263–289
- shaders, 256–262
- .skin, 255–256
- model files, 248–249
- textures, 256–262

quaternions, 92–95, 119–120

Quinn, Harley, 130

## R

ranges, depth, 50

rasterizers, reference, 12

rates, frame, 91

reciprocal of homogeneous w (RHW), 32

rectangles

- values, 49
- viewports, 48

reference rasterizer, 12

registration, window classes, 354

relative positioning, 331

Render(), 51–55, 62–64, 70–72, 117–118

- bump mapping, 191–194

- dot product texture blending, 198–200

- md3.cpp, 282–283

- md3view.cpp, 287

- X files, 243–245

rendering

- anisotropic, 145–146

- errors (z-buffer), 102

- faces, 29

- multipass, 154

- tile-based, 144

Rendermorphics, 4

RenderScene(), cubic environment mapping, 179–181

RenderSceneIntoCube(), cubic environment mapping, 177–179

Renderware, 4

resistance, air, 297

resizing, 55

- textures, 261
  - windows, 4I
  - resource.h file, 37
  - resources
    - DirectX, 486–487
    - general, 486
    - managing, 45
    - version control, 369
    - Windows, 366
  - RestoreDeviceObjects(), 60–62, 68–70, 110–113
    - cubic environment mapping, 176–177
    - X files, 243
  - RestoreDeviceObjects() method, 40, 41–51
  - return codes, 2I
  - RHW (reciprocal of homogeneous w), 32
  - RIGHT vectors, 86
  - rotation
    - camera
      - camera axis, 88–92
      - quaternions, 92–95
    - matrices, 482
      - world matrix, 82–86
  - rules, programming, 16–18
  - Run() function, 444–455
- ## S
- sampling, 37
    - nearest-point, 143
    - taps, 145
  - scalar products, 472–473
  - scaling
    - matrices, 82–86
    - world matrix, 82–86
  - scripts
    - creating, 367
    - inserting, 367
    - shader, 257
  - searching proper device drivers, 410–428
  - selecting textures, 63
  - separation, convex from concave, 325
  - SetLight() method, 98
  - SetStreamSource(), 52, 64
  - SetTexture(), 62
  - SetTextureStageState(), 63, 128
  - SetTextureStageStates(), 198
  - Settings command (Start menu), 12
  - Set Transform(), 86
  - SetVertexShader(), 52, 64
  - shader files, 256–262
  - shading, 24–26
    - faces, 27–29
    - Gouraud, 29–30
    - normals, 29
    - orientation, 26–27
    - vertices, 52–53
  - shapes, 2-D, 78
  - shortcuts, macros, 37
  - ShowWindow() function, 357
  - Simonyi, Charles, 18
  - sizing windows, 355
  - skeletons, windows, 347–361
  - .skin files, 255–256
  - sliding off walls, 323–324
  - software. *See also* applications
    - ColDet, 327, 332–334
    - pluggable software devices, 1I
    - RAPID, 327
  - Software Library for Interference Detection (SOLID), 326
  - SOLID (Software Library for Interference Detection), 326
  - sorts, axis, 314–316
  - sounds, Quake 111, 262–263
  - source code, 6
  - SourceBlendFactor flag, 149
  - spheres, bounding, 325
  - spherical environment mapping, 173–175
  - split AABB (axis aligned bounding box), 329
  - sprite bounds, 309–313
  - squares, 24–25
  - staircase effects, 14I
  - Start menu commands
    - Control Panel, 12
    - Direct3D, 20
    - Settings, 12
  - starting timers, 440–444
  - static
    - friction, 297
    - objects, 317–318

- stencil buffers, 207  
STL (Store Longwood), 315  
Store Longwood (STL), 315  
storing  
  matrices, 83  
  textures, 131  
structures  
  BUMPVERTEX, 183  
  CLIENTCREATESTRCUT, 355  
  CUSTOMVERTEX, 44, 50  
  cvVertices, 50  
  D3DLIGHT8, 98  
  PALETTEENTRY, 60  
  vertex, 45  
WNDCLASS, 352  
styles  
  code, 18–20  
  windows, 355–357  
subtraction  
  matrices, 480  
  vectors, 471–472  
support, multiple textures, 200  
surface attributes, 257  
synchronizing frame rates, 91
- T**
- T&L (transformation and lighting) pipeline, 10, 76  
tagging schemes, 18–20  
taps, 145  
tcMod scroll, 261  
templates  
  Header, 215–216  
  Mesh, 216–218  
testing  
  bounding spheres, 328  
  collision functions, 307  
text, viewing, 37  
textures  
  ADDSIGNED, 168  
  alpha blending, 148–151  
  anisotropic filtering, 144–146  
  anti-aliasing, 140–141  
  blending, 155, 161–164  
  coordinates, 129–131  
  filtering, 140–141  
  fonts, 55–57  
  full-scene anti-aliasing, 146–148  
  linear filtering, 143–144  
  managing, 201  
  mapping, 24–26, 128–129  
    faces, 27–29  
    normals, 29  
    objects, 31–33  
    orientation, 26–27  
  mipmaps, 142  
  modifying, 261  
  multiple textures. *See* multiple textures  
  nearest-point sampling, 143  
  non-D3DFMT\_V8U8, 192  
  Quake III, 256–262  
  selecting, 63  
  storing, 131  
  texture-addressing modes  
    border color texture-addressing, 135–136  
    clamp texture-addressing, 134–135  
    mirror texture-addressing, 133–134  
    MirrorOnce texture-addressing, 136–137  
    wrap texture-addressing mode, 132–133  
  wrapping, 137–140  
  X files, 221–229  
this pointers, 385  
tile-based rendering, 144  
time frames, calculating, 296  
timers, starting, 440–444  
TnHAL (transformation and lighting HAL), 8  
tools, Milkshape 3-D, 33  
tracking monitors, 48  
transformation  
  math, 80  
  matrices, 229–233  
  projection, 79  
transformation and lighting. *See* T&L  
transformation and lighting HAL. *See* TnHAL  
transformation matrices  
translation  
  matrices, 481  
  world matrix, 82–86  
transparent objects, 150  
triadic texture blending members, 170

triangles, 27–29  
fans, 54  
intersections, 332  
lists, 65–72  
trilinear texture filtering, 144  
troubleshooting, 20. *See also* debugging  
multiple textures, 200  
return codes, 21  
types, 19

## U

Unified Memory Architecture, 46  
unit vectors, 89, 477–478  
unperpendicular vectors, 89  
UP vectors, 86  
updating tracking monitors, 49  
utilities, Milkshape 3-D, 33

## V

v-table (VTBL), 14  
values  
AABB (axis aligned bounding box), 327  
depth buffers, 101  
height, 197  
int, 306  
rectangles, 49  
variables  
const, 396  
modifying, 91  
placement declarations, 395–396  
tagging schemes, 18–20  
vectors, 25, 467  
bound vectors, 467–468  
free vectors, 467–477  
LOOK, 27, 86  
per-pixel, 195–198  
perpendicular, 88  
POSITION, 27  
RIGHT, 86  
rotating, 93  
unit vectors, 477–478  
unperpendicular, 89  
UP, 86  
VectorsDisjoint function, 311

version control resources, creating, 369  
vertices  
buffers, 43, 65–72  
Bump Earth, 185  
colors, 99–100  
converting, 50  
D3DFVF\_XYZ format, 333  
defining, 25  
FVF (flexible vertex format), 52  
Gouraud shading, 29–30  
shaders, 52–53  
structures, 45  
transformation pipelines, 77–79  
view matrix, 86–88  
camera axis, 88–92  
quaternions, 92–95  
view transformation stage, 78  
viewing  
matrix, 43  
text, 37  
windows, 357–358  
viewports, 48  
virtual functions (C++), 391–392  
visible surface determination (VSD), 100  
visors, 260  
Visual Basic, 14  
Visual C++ GUI debugger, 20. *See also* C++  
volumes, bounding, 302  
VSD (visible surface determination), 100  
VTBL (v-table), 14

## W

walls, sliding off, 323–324  
.wav files (Quake III), 262–263  
Web sites, 486–487  
Milkshape 3-D, 33  
NVIDIA, 10  
Renderware, 4  
width, 355  
windows  
classes, 351–354  
components, 347  
creating, 428  
formatting, 354–357

optimizing, 361–366  
programming, 346  
resizing, 41  
styles, 355–357  
viewing, 357–358  
Windows (95/98/ME/NT/2000)  
components, 347  
features, 347–351  
    creating classes, 354–357  
    defining classes, 351–354  
    displaying windows, 357–358  
    messages, 358–361  
    registering classes, 354  
    windows procedures, 361  
interaction, 346  
programming, 346  
WinMain() procedure, 347  
winmain.rc file, 36  
WinProc() function, 350  
wireframe models, 108  
WNDCLASS structure, 352  
WndProc() function, 361  
world matrix, 82–86  
world transformation stage, 77  
worlds, building, 212–213  
wParam fields, 358

wrap texture-addressing mode, 132  
wrapping textures, 137–140  
WS\_OVERLAPPEDWINDOW macro, 355  
WS\_VISIBLE flag, 357

## X

X files, 212  
    applying, 235–241  
    extending, 245  
    formats, 213–215  
        animation, 233–235  
    Header template, 215–216  
    Mesh template, 216–218  
    MeshMaterialList object, 218–221  
    MeshNormals template, 221  
    textures, 221–229  
    transformation matrices, 229–233  
x-axis, rotation, 482

## Y

y-axis, rotation, 482

## Z

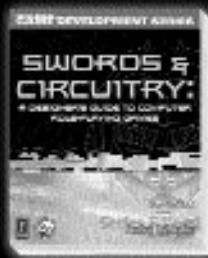
z-axis, rotation, 483  
z-buffers, 74, 102

# GAME DEVELOPMENT.

IT'S SERIOUS BUSINESS.

"Game programming is without a doubt the most intellectually challenging field of Computer Science in the world. However, we would be fooling ourselves if we said that we are 'serious' people! Writing (and reading) a game programming book should be an exciting adventure for both the author and the reader."

—André LaMothe,  
Series Editor



[www.prima-tech.com](http://www.prima-tech.com)  
[www.PrimaGameDev.com](http://www.PrimaGameDev.com)



## **License Agreement/Notice of Limited Warranty**

**By opening the sealed disk container in this book, you agree to the following terms and conditions. If, upon reading the following license agreement and notice of limited warranty, you cannot agree to the terms and conditions set forth, return the unused book with unopened disk to the place where you purchased it for a refund.**

### **License:**

The enclosed software is copyrighted by the copyright holder(s) indicated on the software disk. You are licensed to copy the software onto a single computer for use by a single concurrent user and to a backup disk. You may not reproduce, make copies, or distribute copies or rent or lease the software in whole or in part, except with written permission of the copyright holder(s). You may transfer the enclosed disk only together with this license, and only if you destroy all other copies of the software and the transferee agrees to the terms of the license. You may not decompile, reverse assemble, or reverse engineer the software.

### **Notice of Limited Warranty:**

The enclosed disk is warranted by Prima Publishing to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disk combination. During the sixty-day term of the limited warranty, Prima will provide a replacement disk upon the return of a defective disk.

### **Limited Liability:**

THE SOLE REMEDY FOR BREACH OF THIS LIMITED WARRANTY SHALL CONSIST ENTIRELY OF REPLACEMENT OF THE DEFECTIVE DISK. IN NO EVENT SHALL PRIMA OR THE AUTHORS BE LIABLE FOR ANY OTHER DAMAGES, INCLUDING LOSS OR CORRUPTION OF DATA, CHANGES IN THE FUNCTIONAL CHARACTERISTICS OF THE HARDWARE OR OPERATING SYSTEM, DELETERIOUS INTERACTION WITH OTHER SOFTWARE, OR ANY OTHER SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES THAT MAY ARISE, EVEN IF PRIMA AND/OR THE AUTHOR HAVE PREVIOUSLY BEEN NOTIFIED THAT THE POSSIBILITY OF SUCH DAMAGES EXISTS.

### **Disclaimer of Warranties:**

PRIMA AND THE AUTHORS SPECIFICALLY DISCLAIM ANY AND ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY, SUITABILITY TO A PARTICULAR TASK OR PURPOSE, OR FREEDOM FROM ERRORS. SOME STATES DO NOT ALLOW FOR EXCLUSION OF IMPLIED WARRANTIES OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THESE LIMITATIONS MAY NOT APPLY TO YOU.

### **Other:**

This Agreement is governed by the laws of the State of California without regard to choice of law principles. The United Convention of Contracts for the International Sale of Goods is specifically disclaimed. This Agreement constitutes the entire agreement between you and Prima Publishing regarding use of the software.