THE PREMIER PRESS
GAME DEVELOPMENT SERIES

CD INCLUDED

ADVANCED ANIMATION WITH DIRECTX®

Jim Adams

Series Editor
André LaMothe
CEO Xtreme Games LLC

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Advanced Animation with DirectX

Jim Adams
PREMIER PRESS
GAME DEVELOPMENT

Publisher: Stacy L. Hiquet

Senior Marketing Manager: Martine Edwards

Marketing Manager: Heather Hurley

Associate Marketing Manager: Kristin Eisenzopf

Manager of Editorial Services: Heather Talbot

Acquisitions Editor: Emi Smith

Project Editor/Copy Editor: Cathleen D. Snyder

Retail Market Coordinator: Sarah Dubois

Interior Layout: Scribe Tribe

Cover Designer: Mike Tanamachi

CD−ROM Producer: Brandon Penticuff

Indexer: Kelly Talbot

Proofreader: Jenny Davidson

1−59200−037−1

Library of Congress Catalog Card Number: 2002116166
Printed in the United States of America
03 04 05 06 07 BH 10 9 8 7 6 5 4 3 2 1

Premier Press, a division of Course Technology
25 Thomson Place
Boston, MA 02210

To Jeff, my dearly departed and greatly missed brother
this one's for you!

And to the rest of my family and friends
thanks for being a part of my life!

Acknowledgments

Writing a book is no easy feat. Many hours have gone into the meticulous writing, editing, layout, and printing of this fine book you hold in your hands. The dedication and patience of each and every person involved was greatly appreciated during the book's development. To those people, I'd like to say thank you!

First and foremost, I'd like to thank my family for giving me support. To 2E, my lovely wife, for having the patience to let me write another book (Mrs. T pity the fool that don't let me write my book). To Michael and John, my kids, for being there with me. To my mother, Pam; brothers John, Jeff, and Jason; my sister Jennifer; and my nephew Jordanthanks for your support. To Bill and Doris Fong, thanks for all you have done for me. To my buddy, Jeff Young, for offering to help wrap this baby up and to Richard Young, for providing me with the book's Web space.

To those of you on the publisher sideEmi Smith, Cathleen Snyder, André LaMothe, and Brandon Penticuffthanks for the chance to work with you again. Emi, your neverending patience on a book that was long overdue is astounding. To Cathleen, I'd like to thank you for keeping the editing process running smoothly. And finally to André, for helping to create such a wonderful series of books. I don't want to forget Brandonthanks for putting together the CDROM so quickly!

To those companies that have helped the gaming community at large with their cool productsto Caligari, for making trueSpace and the ohsocool Facial Animator; to discreet and their 3D Studio Max modeling program; to Curious Labs, with their awesome Poser humanfigure designer; and to Microsoft, for creating the eversouseful DirectX.

And last but not least, I'd like to thank you, the readers, for buying this book!

About the Author

JIM ADAMS started his programming career at the ripe young age of 9, when his curiosity and imagination grabbed hold and never let go. Twentyone years later, at the (overripe!) age of 30, Jim still finds himself enthralled in current game−programming techniques such as those in this book. Not one to let anything bog him down, Jim still finds time to raise a family, write a book or two, and help his wife on the rare occasion.

Between his writing and programming, you can find Jim moderating the DirectX forum on the Internet's top gameprogramming Web site, http://www.GameDev.net. Make sure to stop by and give him a ring!

Letter from the Series Editor

Welcome to *Advanced Animation with DirectX*®. This book is all about 3D animation techniques used to develop highend AAA titles such as *Unreal, Doom III,* and other games with fully articulated 3D models featuring skeletal and facial animation. The material in this book has never been published before, for the most part. You can find hints of it on the Internet, in articles, and in other books on robotics, but there isn't another game book about advanced animation available. For the author of this book, Jim Adams, and I, the goal of this book was to really show advanced animation techniques along with the fundamental physics needed to implement them. Of course, the second anyone says "physics" or "math," chills may run down your spine, but Jim has tried to use physics models and explanations in a game programmer's language.

There are some amazing things covered in this book about which I am very excited, such as "rag doll" physics. A good example of this technique is found in *Unreal Tournament;* when you frag an opponent, his body falls limp and the force of the weapon blast throws it like, well, a rag doll!

The book also covers cloth animation and spring modeling. Using these techniques, you can create everything from cloth to soft bodies to springy objects that deform and return to their original shape when forces are applied to them.

Finally, two of the coolest things in the book are facial animation and lipsyncing techniques. This is something that very few games have or will have for quite some time. Again, Jim gets you up and running with the technology in no time.

In conclusion, this is definitely an advanced book, and serious material. If you are new to game programming you will still get something out of it, but I suggest that you should be very comfortable with 3D concepts, math, and some high school physics before you delve into this madness. Nevertheless, after you get through it, you will have an edge on 90% of the game developers out there, who haven't even begun to master these techniques.

Sincerely,

[illegible]

André LaMothe

Game Development Series Editor

Introduction

So you're ready to progress beyond the basics. Having already tackled drawing polygons and meshes, blending textures, manipulating vertex buffers, and tinkering with vertex shaders, what's next for an aspiring game programmer like you to learn? Well, you've come to the right place if you're ready to move on beyond the basics.

Welcome to *Advanced Animation with DirectX®* ! This book is your guide to getting past the basics of DirectX graphics and into the bigger world of advanced animation! Take the information you already know and learn how to expand it into a vast array of various eyepopping graphical effects.

Keep your highschool textbooks at school, however, because this book won't bore you with pages of theories and algorithms. Instead, you'll see realworld examples of some of the hottest animation techniques used today. Jampacked with easy to understand concepts, fully commented code, and cool demos, this book makes learning the advanced stuff simple and fun!

What This Book Is About

As you can tell from the table of contents (you did check it out, didn't you?), this book has been developed with intermediate to advanced programmers in mind. There are no beginner sections (well, almost no beginner sections)it's all hardcore theory and programming from the get go!

This means there is no room wasted on teaching basic concepts, such as initializing Direct3D or using Windows message pumps, so you need to know a little something about Direct3D in general before you go on. Whoa! Don't put the book down just yet. I'm talking about the extreme basics, such as initializing Direct3D, using materials and textures, and handling vertex buffers. If you know all that, then you are definitely ready to move on to the advanced stuff, and this book is the place to start!

Why You Should Read This Book

That is the milliondollar questionWhy read this book? Let's face it, gaming technologies are evolving faster than most of us can keep up with. Today is one thing, and tomorrow introduces techniques that'll blow your socks off. Within these pages, you'll find information on advanced animation techniques that you can use in your own projects, and then your game will be the one that blows people's socks off!

What type of advanced animation techniques am I discussing? Well, my friend, read on to see what this book has to offer.

What's in This Book?

In this book you'll find 14 chapters full of advanced animation goodies. Each chapter concentrates on a single animation technique; aside from the first few informational chapters, the book is completely modular, meaning you can skip the chapters that don't interest you and get right to the topics that do.

Of course, I know you're really interested in every chapter in this book, so why don't you take a moment and see what you're about to get yourself into. The following list summarizes each chapter.

- **Chapter 1: Preparing for the Book.** Prepare yourself, because this book gets right to the point of using DirectX to create awesome animations in your programs! This chapter will help you install DirectX and set up your compiler to use it, and then get you programming right off the bat by using a library of objects and functions created to hasten your development. This chapter is a mustread before you journey into the rest of the book.

- **Chapter 2: Timing in Animation and Movement.** Timing is an important aspect of animation, and this chapter provides a primer of what to expect throughout the rest of the book. See how to make your meshes animate and move over time.
- **Chapter 3: Using the .X File Format.** Getting your mesh data into your project is quite an endeavor. The information in this chapter will give you a firm grasp of using Microsoft's proprietary 3D graphics storage format, .X, in your game projects. Learn how to store and retrieve 3D mesh information, as well as how to use .X to contain custom data related to your game project.
- **Chapter 4: Working with Skeletal Animation.** Probably the most technically advanced realtime animation technique used today is skeletal animation. This chapter tackles the subject by showing you how to get started using this popular technique to manipulate meshes based on an underlying set of bones.
- **Chapter 5: Using KeyFramed Skeletal Animation.** One of the most popular animation techniques is using keyframed animation sets (created with popular 3D modeling packages such as discreet's 3D Studio Max or Caligari's trueSpace) in your projects. This chapter will show you how o take the key−framed skeletal animation informaion (stored in .X files) and use it to animate your meshes onscreen.
- **Chapter 6: Blending Skeletal Animations.** Tired of your key−framed animations being so static? I'm talking about animations that never change! How about mixing things up by blending multiple animation sets to create new and unique animations in your game?
- **Chapter 7: Implementing Rag Doll Animation.** Here it is; I know you've been waiting for it. See how to create your very own rag doll animation sequences, with limbs flailing and bodies flyingit's all in this chapter. Leave your physics books at home, kids; this book gives it to you in a straightforward and easytounderstand manner.
- **Chapter 8: Working with Morphing Animation.** Skeletal animation be damnedmorphing animation still has a rightful place in the world of advanced animation! See how you can use a simple animation technique to get those meshes morphing in your own projects.
- **Chapter 9: Using KeyFramed Morphing Animation.** Even morphing animation needs to be sequenced, right? See how to define and create your own series of keyframed animation objects and apply them to your morphing animation techniques! You'll even get to extend the usefulness of .X by creating your own morphing keyframe templates!
- **Chapter 10: Blending Morphing Animations.** Once again, blending rears its eversohelpful head, this time for combining various blending animations together to create new and unique animations during run time!
- **Chapter 11: Morphing Facial Animation.** They walk, they jump, and yesthey even talk! Your game's characters really come to life when you use the facial animation techniques shown in this chapter. If you don't know what facial animation can do for you, check out a game such as Electronic Art's *Medal of Honor: Frontline* or Interplay's *Baldur's Gate: Dark Alliance.*
- **Chapter 12: Using Particles in Animation.** What game nowadays doesn't have flashy, smoking, zinging blobs of graphical pixels that we call particles? Yours, you say? Well, no worriesthis chapter will show you how to effectively add particle animation into your game and get those flashy blobs of particles flying around in no time.
- **Chapter 13: Simulating Cloth and Soft Body Mesh Animation.** Close your eyes and imagine your game's hero wearing a silky cape that reaches the ground and flutters with every small gust of wind. Using cloth simulation, you can give your hero a cape, clothes, and much more than you ever imagined. Top it off with the use of soft body meshes that deform to the slightest touch, and you have yourself some fine advanced animation techniques!
- **Chapter 14: Using Animated Textures.** I'm saving the best for last. This chapter will show you how to animate the textures you use to paint your 3D world. Animated fire, flowing water, and so much more is possible using animated textures, and this chapter will show you how!

Whew! There are some major topics in here, all for your eyes only! As I wrote this book, I couldn't help but wonder what type of applications you'll apply these techniques to. I can envision some incredible games using the various techniques, and I can tell that you're just the person for the job. After you read this book (if you're like me, that means reading it at least six times), I believe you'll find some useful ways to apply your newfound knowledge to your game projects.

I know you're anxious to get going, so let me once again say welcome, and thank you for buying my book. I hope it helps you learn some of the concepts and techniques currently used in 3D animation. Have fun, and enjoy!

Part One: Preparations

Chapter 1: Preparing for the Book

Chapter 1: Preparing for the Book

Overview

Alpha and Omega−the beginning and the end. To all things there is a beginning, and in this book, this chapter represents the start of a grand journey into the world of advanced animation. Behind you is the power of Microsoft's DirectX SDK; in front of you lies a long and winding road, filled with uncertainty. Before you set off, however, there are a number of things you need to do to ensure that your experience is a joyous one.

This chapter contains information that will help you prepare yourself for the topics and code in the book, from installing the DirectX SDK to understanding how to use a series of helper functions to speed up your development time. Before you delve into this book, I highly recommend you give this chapter a full read and take your time setting up DirectX and your compiler. After this chapter, you should be ready to start your epic journey!

By the end of this book, you'll have found that the world of advanced animation is only as difficult or as easy as you make it. With the knowledge you already possess and this book at your side, I'm sure it will be easy for you!

Installing the DirectX SDK

Welcome to the grand world of DirectX programming! If you haven't already done so, install the Microsoft DirectX Software Development Kit (DX SDK) before you delve into the text and code in this book. If you're new to the DirectX installation process, don't worry−it's all been streamlined so that you don't need to do much more than click a few buttons. As for you veteran DirectX coders, you should still take a quick gander at the installation guidelines here in case there's something you haven't seen.

At the time of this book's printing, Microsoft's DirectX SDK version 9 has been released. It's always best to remove any past installations of the DirectX SDK before you install the newest one. Don't worry−you're not losing any precious code because each new version of the DirectX SDK comes jam−packed with each previous release's code intact! That's right, with DirectX 9 (and subsequent DirectX releases), you have complete access to every component ever made in DirectX, from the oldest DirectDraw surface object to the newest 3D−accelerated device objects! Don't fret, DirectX 8 users−your version 8 code will still work with version 9, since version 9 contains all previous versions' objects.

For DirectX 8 users, the upgrade from the DirectX 8 SDK to the DirectX 9 SDK may seem a little shocking, but once you get around the fluff, you'll realize that not much has changed. It's just that the DirectX 9 SDK adds a couple of new features over version 8.

Microsoft has packed the DirectX SDK into a single, easy−to−execute installation package, which by a matter of chance (and good planning) you'll find on this book's CD−ROM. The installation process for DirectX has remained the same over the last few versions, so if you've installed it before, you already know what's involved. With DirectX 9, the installation screen (shown in Figures 1.1 and 1.2) has changed only slightly in appearance−the basic installation steps remain the same throughout all versions to date.

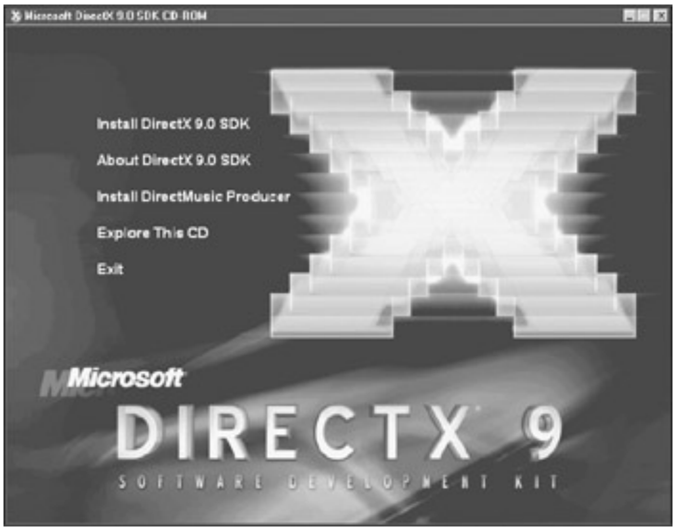
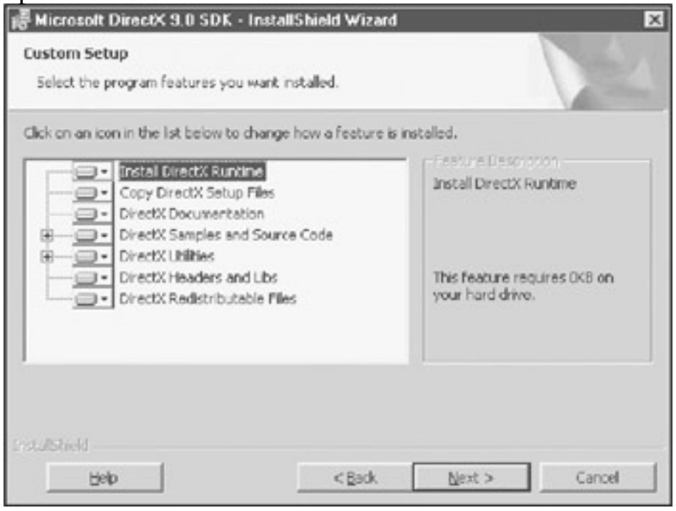


Figure 1.1: The DirectX 9 SDK introduction screen gives you a few options−most importantly the installation option for the DirectX 9 SDK.


Microsoft DirectX 9.0 SDK - InstallShield Wizard

Custom Setup

Select the program features you want installed.

Click on an icon in the list below to change how a feature is installed.

- Install DirectX Runtime
- Copy DirectX Setup Files
- DirectX Documentation
- DirectX Samples and Source Code
- DirectX Utilities
- DirectX Headers and Libs
- DirectX Redistributable Files

Feature Description

Install DirectX Runtime

This feature requires 0KB on your hard drive.

Figure 1.2: Clicking on Install DirectX 9.0 SDK will eventually lead you to the InstallShield Wizard, where you can decide which of the SDK components to install.

To make things easier, the fine folks at Premier Press have created a menu on the CD interface so you can easily locate programs. It just so happens that one of those programs is the DirectX SDK. To access the menu (if it didn't appear automatically when you inserted the CD−ROM), click on the Start button and select Run. Type **D:\start_here.html** (where D is your CD−ROM drive letter) in the text box and click the OK button. You will be presented with the Premier Press license agreement. Please read the agreement, and if you agree to the terms, click I Agree to continue. The CD−ROM's main interface will appear, at which point you can locate and click the Install DirectX SDK option.

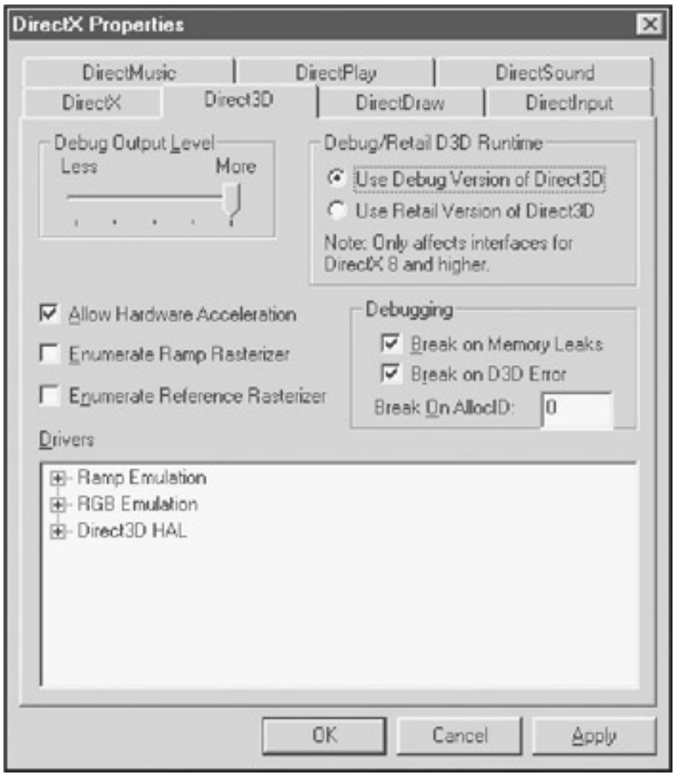
Once you're at the Microsoft DirectX 9.0 SDK − InstallShield Wizard dialog box, you get to decide which components to include with your installation. If you're a new user you should click Next to install the default components. Veteran users may want to tweak the settings a bit to fit the installation to their exact needs. Either way, make sure to click Next and follow the onscreen directions to complete your DirectX 9.0 SDK installation. Before you know it, you'll be ready to start working with the SDK!

Choosing the Debug or Retail Libraries

Another important part of using the DirectX SDK is selecting the developer run−time libraries you'll be using. These libraries are different from the end−user run−time libraries; they allow you to choose whether to use the debug or retail libraries.

Developers use the debug run−time libraries to help track bugs in their projects. The debug libraries provide helpful messages during compile time and perform special operations during execution. For instance, DirectSound fills sound buffers with static sound so you can hear it while testing for valid buffers.

When you are developing your game project, it is recommended that you use the debug run−time libraries, switching to the retail run−times to perform the final tests on your games. To switch between the two run−time libraries, open the Control Panel and select the DirectX icon. The DirectX Properties dialog box will appear, as shown in Figure 1.3.



Debug Output Level: Less — More (slider at More)

Debug/Retail D3D Runtime: [✓ Use Debug Version of Direct3D] Use Retail Version of Direct3D

Note: Only affects interfaces for DirectX 8 and higher.

- [✓] Allow Hardware Acceleration
- [ ] Enumerate Ramp Rasterizer
- [ ] Enumerate Reference Rasterizer

Debugging:
- [✓] Break on Memory Leaks
- [✓] Break on D3D Error
- Break On AllocID: 0

Drivers:
- Ramp Emulation
- RGB Emulation
- Direct3D HAL

Figure 1.3: The DirectX system properties give you a multitude of options. In this case, the Direct3D tab has been selected, and you can see that the debug libraries are in use at the highest debug output level.
In Figure 1.3, you can see that I've clicked on the Direct3D tab. In this tab, you can select a run−time library to use in the Debug/Retail D3D Runtime section. You can choose either Use Debug Version of Direct3D or Use Retail Version of Direct3D. I recommend sticking with Use Debug Version of Direct3D whenever possible.

However, if you decide to use the debug libraries, be warned that your programs may not operate at peak efficiency. The debug libraries are made for debugging your DirectX applications; certain features are put in place that purposely alter the way DirectX works. For example, the debug libraries will spew debug information about every DirectX interface, how it's used, and how it's freed. Every little nuance is logged, thus slowing down your application's execution.

To adjust the number of these debug messages Direct3D will pass to your debugger, you can adjust the slider bar in the Debug Output Level portion of the Direct3D properties. The higher the debug level, the more messages you'll receive. I recommend sliding the bar all the way to the right so you get every debug message during your project's development. Again, you'll slow down the execution of your program, but in the end you'll get to see everything that is happening behind the scenes. When your project is ready for retail, quickly change the libraries that you're using in the DirectX control panel.

When you have set the run−time library and debug level, click the OK button to close the DirectX Properties dialog box.

Configuring Your Compiler

After you've installed the DirectX SDK, you need to prepare your compiler to work with the book's code and demos. Even though the DirectX SDK installation program does a good job of setting up some of the compiler options for you, I want to go over every setting that you'll need to configure.

Setting the DirectX SDK Directories

The first (and most important) setting is for the DirectX SDK installation directories. Your compiler needs to know where to find the DirectX include and library files. Typically, the DirectX SDK installation program will insert the SDK directories into Microsoft's Visual C/ C++ compiler for you, but you might need to add these directories yourself at some point.

To add the directories in MSVC version 6, open your compiler and click Tools. Select Options from the list and click the Directories tab. The Options dialog box will open, as shown in Figure 1.4.

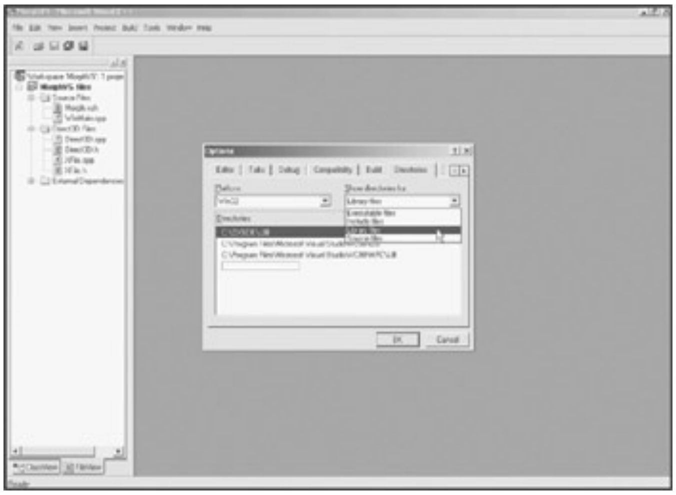


Figure 1.4: The Options dialog box displays a list of directories that Visual C/C++ searches for header and library files.

Visual Studio .NET users need to click on Tools, and then select the Projects folder in the Options dialog box. From there, select the VC++ Directories to open up the directory list on the right side of the Options dialog box (as shown in Figure 1.5).

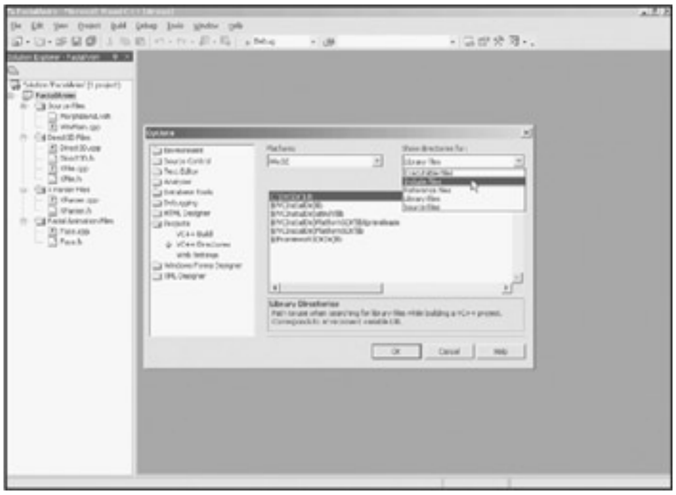


Figure 1.5: Selecting VC++ Directories will bring up the directory list on the right side of the dialog box in Visual Studio .NET.

With either compiler, you'll notice the directories listed that your compiler searches for various libraries and include files. Notice the Show Directories For drop−down menu. You should start by setting the header file directory for the DirectX SDK, so select Include Files from the drop−down menu.

Next, click the New button (the little hatched box to the left of the red X). The focus will be shifted to a new line in the Directories section of the dialog box. Click on the ellipsis button to the right of the text cursor to open the Choose Directory dialog box. Locate the DirectX header file installation directory and click OK.

Now you need to set the DirectX SDK library directory, so click on the Show Directories For drop−down menu again and select Library Files. Repeat the process of clicking on the New button and locating the library directory of your DirectX SDK installation directory. When you finish, your compiler will be ready to use the DirectX directories for compiling.

Linking to the DirectX Libraries

After you've set the DirectX installation directories, the next important step is to link the libraries that you'll be using in your project. Note that linking files is project−specific, so make sure you have your game project open before you continue.

For MSVC 6, click on Project and select Settings. Click on the Link tab. The Project Settings Link properties will appear, as shown in Figure 1.6.

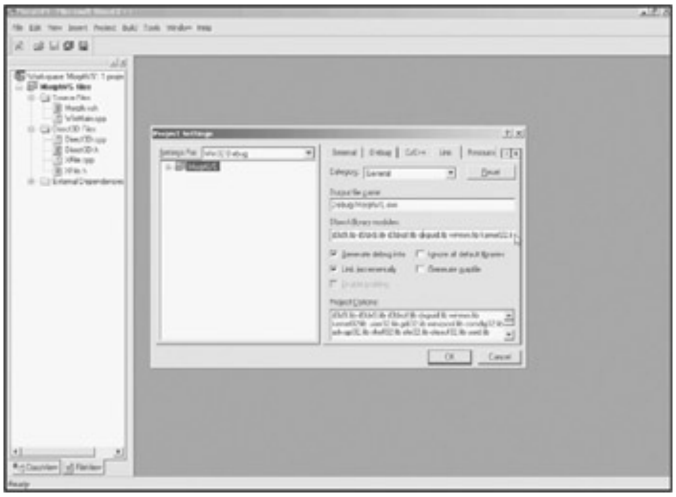


Figure 1.6: The Project Settings Link properties allow you to specify exactly which library files to link to your application.

For Visual Studio .NET, open your project and then highlight it in the Solution Explorer (as shown in Figure 1.7). Next, click on Project and then Properties to open the project's Property Pages dialog box. In the folder display, select the Linker folder and click on Input. On the right of the Property Pages dialog box, you'll see your project's link settings (as shown in Figure 1.8).

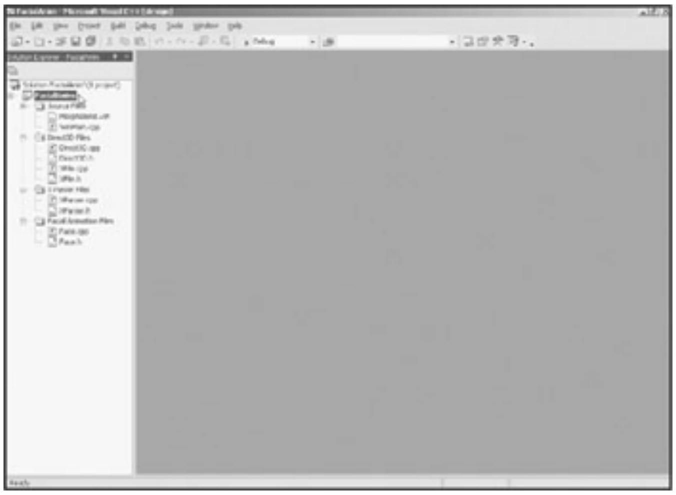


Figure 1.7: Visual Studio .NET lists all files and projects in use in the Solution Explorer.

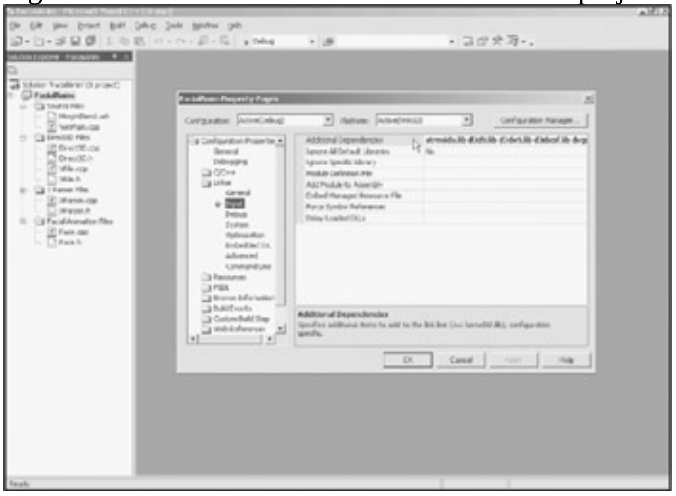


Figure 1.8: The project's Property Pages dialog box is where you are allowed to modify your linker settings,

among other things.
For this book, I only use the Direct3D, D3DX, and DirectShow libraries, so those are the only libraries you need to add to the link process (aside from the standard application libraries already listed in the Object/Library Modules box). To add the required libraries, add the following text to the Object/Library Modules text box (for MSVC 6) or the Additional Dependencies text box (for VS.NET):

```
d3d9.lib d3dx9.lib d3dxof.lib dxguid.lib winmm.lib
```

If you are using DirectShow to create animated textures, you need to add either strmbasd.lib or strmbase.lib to the Object/Library Modules text box. Consult Chapter 14, "Using Animated Textures," for the specifics on using DirectShow libraries in your projects.

At this point, you're ready to finish up by setting your compiler's default `char` state.

Setting the Default char State

Being an old−school programmer, I tend to follow the old ways, especially when it comes to dealing with the default state of the `char` data type. As its default setting, the Visual C/C++ compiler expands all `char` data types to `signed char` types. I don't know about you, but I use `unsigned char` data types more often than `signed char` types, so setting this default `unsigned` state is a priority.

To set the default state of the `char` data type to `unsigned char` in Visual C/C++ 6, select Project and then Settings. Click on the C/C++ tab (as shown in Figure 1.9), select General from the Category drop−down menu, and append /**J** to the end of the Project Options text. Click OK and you'll be set to use `unsigned char` as the default state whenever you specify a `char` variable.

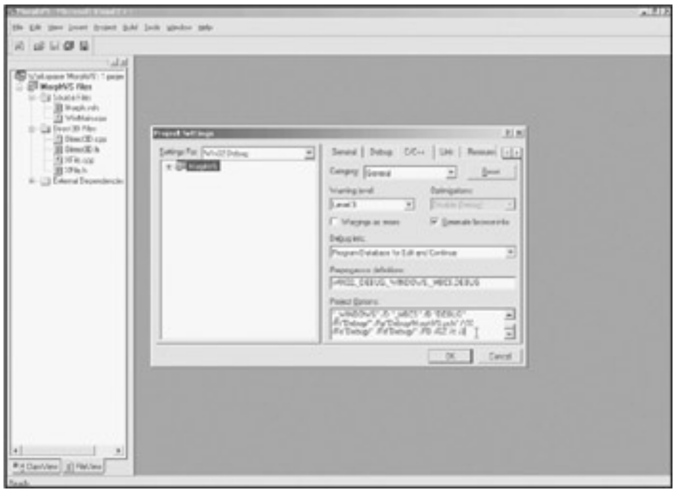


Figure 1.9: The Project Settings dialog box allows you to add specific compiler commands in the Project Options text box, as well as change a multitude of other compiler options.
Visual Studio .NET users need to select the project in the Solution Explorer and click Project, Properties. In the folder list, select the C/C++ folder and click on Command Line. In the Additional Options text box, type /**J**

And that wraps it up for setting up DirectX and your compiler! Now for you DirectX pros, I'm sure the installation is a breeze, and once you've gone through the process, you'll be ready to start coding in no time flat.

Whoa, hold it there, partner! Before you move on, let's take a moment to discuss something that is extremely important when it comes to some serious program development−creating a reusable library of helper functions to speed up your development. I've developed an entire set of functions that will help you skip the mundane DirectX work, such as initializing the display modes and loading/rendering meshes.

Let's take a closer look at these helper functions I've created, how they're used throughout the book, and how you can use them in your own projects.

Using the Book's Helper Code

Since this is an advanced book, I assume you are at least proficient with DirectX. I'm talking proficient enough to understand how Direct3D works in relation to the rendering pipeline, using vertex buffers, textures, and regular meshes−you know, the ultimate basics of getting anything done with Direct3D.

To help speed up development time and reduce time spent on the mundane repetitive code, I have created a small set of functions and objects that are used throughout this book. These functions help you deal with those Direct3D features you are likely to use the most, such as initializing Direct3D and loading and rendering meshes.

Along with the functions, I've also created a set of objects that extend the usefulness of certain D3DX objects. These objects are extended versions of the DirectX 9 D3DX objects `D3DXFRAME` and `D3DXMESHCONTAINER`, as well as some that can help you parse .X files in your own projects.

So not to make a short story longer than it has to be, let's get right to the point and see what functions and objects are at your disposal, starting with the helper objects.

Using the Helper Objects

As I previously mentioned, I've created a series of objects that extend the usefulness of the `D3DXFRAME` and `D3DXMESHCONTAINER` objects that come as part of the D3DX library. If you are not familiar with those objects, then let me give you a quick overview.

The `D3DXFRAME` object helps form a hierarchy of reference frames. These reference frames are used to connect a series of meshes together, with each frame having its own transformation to apply to the mesh connected to it. In this way of using frames to point to meshes, you can minimize the number of meshes used because you can reference meshes instead of having to reload them.

For example, imagine you have a car that consists of a body and four wheels. The body and wheel form two meshes. These two meshes are used in conjunction with five frames (one for the body and four for the tires). When rendering, each frame's transformation is used to position and render the mesh that the frame uses. That means one frame transforms and renders the body once, while the other frames transform and render the tire mesh four times.

As for the `D3DXMESHCONTAINER` object, it is used to contain a mesh as well as to link to a series of other meshes (using a linked list). Why not just use the `ID3DXBaseMesh` object instead, you ask? Well, there's more to `D3DXMESHCONTAINER` than you might expect. First, you can store any type of mesh, whether it's regular, skinned, or progressive. Second, the `D3DXMESHCONTAINER` object holds material and effect data.

For complete information on the `D3DXFRAME` and `D3DXMESHCONTAINER` objects, please consult the DirectX SDK documents. For now, let's see how I've managed to extend the usefulness of those two objects (both of which are defined in the \common\Direct3D.h source file of this book's CD−ROM), starting with D3DXFRAME.

Extending D3DXFRAME

By itself, the D3DXFRAME object is very useful, but unfortunately it lacks a few very essential tidbits of information, namely data for containing transformations when animating meshes, functions to handle the animation data, and a default constructor and destructor.

To correct these omissions, I have created an extended version of D3DXFRAME, which I call D3DXFRAME_EX. This new object adds a total of two D3DXMATRIX objects and six functions to the mix. The two matrix objects contain the original transformation of the frame (before any animation transformations are applied) and the combined transformation from all parent frames to which the frame is connected (in the hierarchy).

Here's how I defined the D3DXFRAME_EX structure along with the two matrix objects:

```
struct D3DXFRAME_EX : D3DXFRAME
{
   D3DXMATRIX matCombined; // Combined matrix
   D3DXMATRIX matOriginal; // Original transformation
```

You'll learn about these two matrix objects and how to work with them later on in the book. For now, let's just move on to the functions, starting with the constructor. The constructor has the job of clearing out the structure's data (including the original data from the base D3DXFRAME object).

```
D3DXFRAME_EX()
{
   Name = NULL;
   pMeshContainer = NULL;
   pFrameSibling = pFrameFirstChild = NULL;
   D3DXMatrixIdentity(&matCombined);
   D3DXMatrixIdentity(&matOriginal);
   D3DXMatrixIdentity(&TransformationMatrix);
}
```

On the flip side, the destructor has the job of freeing the data used by the D3DXFRAME_EX object.

```
~D3DXFRAME_EX()
{
   delete [ ] Name;         Name = NULL;
   delete pFrameSibling;    pFrameSibling = NULL;
   delete pFrameFirstChild; pFrameFirstChild = NULL;
}
```

As you can see, the constructor and destructor are pretty typical in the way those things normally go−initialize the object's data and free the resources when done. What comes next are a handful of functions that help you search for a specific frame in the hierarchy, reset the animation matrices to their original states, update the hierarchy after modifying a transformation, and count the number of frames in the hierarchy.

The first function, Find, is used to find a specific frame in the hierarchy and return a pointer to it. If you're not aware of this, each D3DXFRAME object (and the derived D3DXFRAME_EX object) has a Name data buffer, which you're free to fill in with whatever text you find appropriate. Typically, frames are named after bones that define the hierarchy (as I will discuss in Chapter 4, "Working with Skeletal Animation").

To find a specific frame (and retrieve a pointer to the frame's object), just call the `Find` function, specifying the name of the frame you wish to find as the one and only parameter.

```
// Function to scan hierarchy for matching frame name
D3DXFRAME_EX *Find(const char *FrameName)
{
   D3DXFRAME_EX *pFrame, *pFramePtr;

   // Return this frame instance if name matched
   if(Name && FrameName && !strcmp(FrameName, Name))
     return this;

   // Scan siblings
   if((pFramePtr = (D3DXFRAME_EX*)pFrameSibling)){
     if((pFrame = pFramePtr->Find(FrameName)))
       return pFrame;
   }

   // Scan children
   if((pFramePtr = (D3DXFRAME_EX*)pFrameFirstChild)) {
     if((pFrame = pFramePtr->Find(FrameName)))
        return pFrame;
    }

    // Return none found
    return NULL;
}
```

The `Find` function compares the name you passed to the current frame's name; if they match, the pointer to the frame is returned. If no match is found, then the linked list is scanned for matches using a recursive call to `Find`.

Next in the line of added functions is `Reset`, which scans through the entire frame hierarchy (which, by the way, is a linked list of child and sibling objects). For each frame found, it copies the original transformation to the current transformation. Here's the code:

```
// Reset transformation matrices to originals
void Reset()
{
  // Copy original matrix
  TransformationMatrix = matOriginal;

  // Reset sibling frames
  D3DXFRAME_EX *pFramePtr;
  if((pFramePtr = (D3DXFRAME_EX*)pFrameSibling))
    pFramePtr->Reset();

  // Reset child frames
  if((pFramePtr = (D3DXFRAME_EX*)pFrameFirstChild))
    pFramePtr->Reset();
}
```

Typically, you call `Reset` to restore the frame hierarchy's transformation back to what it was when you created or loaded the frames. Again, the entire frame hierarchy is explained better in Chapter 4 And not to keep you confused, but the next function in the list is `UpdateHierarchy`, which has the job of rebuilding the entire frame hierarchy's list of transformations after any one of those transformations has been altered.

Rebuilding the hierarchy is essential to making sure the mesh is rebuilt or rendered correctly after you have updated an animation. Again, skeletal animation stuff here−just consult Chapter 4 for more information. For now, let's just check out the code, which takes an optional transformation matrix to apply to the root frame of the hierarchy.

```
// Function to combine matrices in frame hiearchy
void UpdateHierarchy(D3DXMATRIX *matTransformation = NULL)
{
  D3DXFRAME_EX *pFramePtr;
  D3DXMATRIX matIdentity;

  // Use an identity matrix if none passed
  if(!matTransformation) {
    D3DXMatrixIdentity(&matIdentity);
    matTransformation = &matIdentity;
  }

  // Combine matrices w/supplied transformation matrix
  matCombined = TransformationMatrix * (*matTransformation);

  // Combine w/sibling frames
  if((pFramePtr = (D3DXFRAME_EX*)pFrameSibling))
    pFramePtr−>UpdateHierarchy(matTransformation);

  // Combine w/child frames
  if((pFramePtr = (D3DXFRAME_EX*)pFrameFirstChild))
    pFramePtr−>UpdateHierarchy(&matCombined);
}
```

The simplicity of the `UpdateHierarchy` code is pretty neat when you think about it. Explained better in later chapters, the `UpdateHierarchy` function transforms the frames by their own transformation matrix (stored in `matTransformation`) by a matrix that is passed as the optional parameter of the function. This way, a frame inherits the transformation of its parent frame in the hierarchy, meaning that each transformation applied winds its way down the entire hierarchy.

Last, with the `D3DXFRAME_EX` object you have the `Count` function, which helps you by counting the number of frames contained within the hierarchy. This is accomplished using a recursive call of the `Count` function for each frame contained in the linked list. For each frame found in the list, a counter variable (that you provide as the parameter) is incremented. Check out the `Count` code to see what I mean.

```
void Count(DWORD *Num)
{
  // Error checking
  if(!Num)
    return;

  // Increase count of frames
  (*Num)+=1;

  // Process sibling frames
  D3DXFRAME_EX *pFrame;
  if((pFrame=(D3DXFRAME_EX*)pFrameSibling))
    pFrame−>Count(Num);

  // Process child frames
  if((pFrame=(D3DXFRAME_EX*)pFrameFirstChild))
    pFrame−>Count(Num);
  }
```

```
};
```

And that pretty much wraps up the D3DXFRAME_EX object. If you're used to using the D3DXFRAME object (and you should be if you're a DX9 user), then everything I've just shown you should be pretty easy to understand.

Moving on, let me now introduce you to the next helper object that I've created to extend the usefulness of the D3DXMESHCONTAINER object.

Extending D3DXMESHCONTAINER

Whereas you might be used to using the ID3DXMesh object to contain your mesh data, you may have found it a pain to store the mesh's material and effects data separately. Not only that, but what about using the other D3DX mesh objects, such as ID3DXPMesh and ID3DXSkinMesh? Why not just create a single mesh object that represents all mesh types and contains all material data along with it?

In fact, there is such an object−it's called D3DXMESHCONTAINER! The D3DXMESHCONTAINER object stores a pointer to your mesh data (regardless of the mesh type used) and all material and effects data. It also contains pointers to your mesh's adjacency buffer and skinned mesh data object. And as if that wasn't enough, the D3DXMESHCONTAINER contains pointers to form a linked list of mesh objects.

What could I possibly do to extend the usefulness of the already nifty D3DXMESHCONTAINER, you ask? Well, for one thing, D3DXMESHCONTAINER has no default constructor or destructor. Also, textures data is missing−there's only a buffer that contains the names of the textures to use for the mesh. Last, there's no support for storing skinned mesh animation data.

No problem, because extending the D3DXMESHCONTAINER is simple! The new version, which I call D3DXMESHCONTAINER_EX, adds a total of four data objects and three functions. The data objects include an array of texture objects, a skinned mesh object (to store an animated skinned mesh), and two arrays of matrix objects.

Here's how I defined the D3DXMESHCONTAINER_EX object, as well as declaring the four variables I mentioned:

```
struct D3DXMESHCONTAINER_EX : D3DXMESHCONTAINER
{
  IDirect3DTexture9 **pTextures;
  ID3DXMesh          *pSkinMesh;

  D3DXMATRIX        **ppFrameMatrices;
  D3DXMATRIX         *pBoneMatrices;
```

The pTextures array of pointers contains the texture objects used to render the mesh. I build the pTextures array up by first loading a mesh and then querying the texture buffer (D3DXMESHCONTAINER::pMaterials) for the file names of the textures to use.

As for pSkinMesh, you only use it when you are using a skinned mesh (which I will discuss in Chapter 4 through 7). You see, when loading a skinned mesh, the actual mesh data is stored in D3DXMESHCONTAINER::MeshData::pMesh. The only problem is, you need another mesh container to store the skinned mesh as it is animated. That is the purpose of pSkinMesh.

Last, you'll find `ppFrameMatrices` and `pBoneMatrices`. Not to drag it out, but these are also used for skinned meshes, and these matrix objects are explained in Chapter 4. Just so it makes sense at this point, a skinned mesh animates by attaching the vertices of the mesh to an underlying hierarchy of bones. As the bones move, so do the vertices. `ppFrameMatrices` and `pBoneMatrices` are used to map the vertices to the bones.

Aside from the variables in `D3DXMESHCONTAINER_EX`, there are also a few functions. The first two are the constructor and destructor:

```
D3DXMESHCONTAINER_EX()
{
  Name               = NULL;
  MeshData.pMesh     = NULL;
  pMaterials         = NULL;
  pEffects           = NULL;
  NumMaterials       = 0;
  pAdjacency         = NULL;
  pSkinInfo          = NULL;
  pNextMeshContainer = NULL;
  pTextures          = NULL;
  pSkinMesh          = NULL;
  ppFrameMatrices    = NULL;
  pBoneMatrices      = NULL;
}

~D3DXMESHCONTAINER_EX()
{
  if(pTextures && NumMaterials) {
   for(DWORD i=0;i<NumMaterials;i++)
   ReleaseCOM(pTextures[i]);
   }
   delete [] pTextures;       pTextures = NULL;
   NumMaterials = 0;

   delete [] Name;            Name = NULL;
   delete [] pMaterials;      pMaterials = NULL;
   delete pEffects;           pEffects = NULL;

   delete [] pAdjacency;      pAdjacency = NULL;
   delete [] ppFrameMatrices; ppFrameMatrices = NULL;
   delete [] pBoneMatrices;   pBoneMatrices = NULL;

   ReleaseCOM(MeshData.pMesh);
   ReleaseCOM(pSkinInfo);
   ReleaseCOM(pSkinMesh);

   delete pNextMeshContainer; pNextMeshContainer = NULL;
}
```

The constructor and destructor have the task of initializing the data to a known state and releasing the data used by the object, respectively. You'll notice the use of the `ReleaseCOM` macro, which I'll describe to you in the upcoming section, "Checking Out the Helper Functions." Basically, `ReleaseCOM` is a macro that safely releases a COM interface and sets the interface pointer to NULL.

The third function in `D3DXMESHCONTAINER_EX` is `Find`, which lets you scan the linked list of meshes for a specifically named mesh, much like `D3DXFRAME_EX::Find`. A quick string compare is used to check the names, and a recursive call to `Find` is used to scan the entire linked list.

```
D3DXMESHCONTAINER_EX *Find(char *MeshName)
{
  D3DXMESHCONTAINER_EX *pMesh, *pMeshPtr;

  // Return this mesh instance if name matched
  if(Name && MeshName && !strcmp(MeshName, Name))
    return this;

  // Scan next in list
  if((pMeshPtr = (D3DXMESHCONTAINER_EX*)pNextMeshContainer)) {
    if((pMesh = pMeshPtr->Find(MeshName)))
      return pMesh;
   }

   // Return none found
   return NULL;
  }
};
```

And that does it for the helper objects! The `D3DXFRAME_EX` and `D3DXMESHCONTAINER_EX` objects are extremely helpful when it comes to dealing with Direct3D; as such, you should spend as much time as you can getting used to these two objects. I think you'll find them very useful in your own projects.

Aside from the helper objects, there are a number of helper functions that I'd like to introduce to you, which should help you alleviate the mundane tasks common to most Direct3D−related projects.

Checking Out the Helper Functions

The helper functions that I decided to implement for this book are few in number, but they represent the majority of code that you're likely to use in all your projects. These functions perform tasks including releasing COM interfaces, initializing Direct3D, loading meshes and vertex shaders, updating skinned meshes, and rendering meshes using standard methods and vertex shaders.

Let me start at the top of the list with a function (or rather, a macro) that you can use to safely release your COM interfaces.

Releasing COM Interfaces

Starting off the batch of helper functions (which are stored in the \common\Direct3D.cpp file of this book's CD−ROM) is the macro `ReleaseCOM`, which you can use to safely release COM interfaces in your project, even if those objects are not valid (NULL pointers).

```
#define ReleaseCOM(x) { if(x!=NULL) x->Release(); x=NULL; }
```

`ReleaseCOM` takes one parameter, which is the pointer to the COM interface you want to safely release. For example, the following bit of code demonstrates how to load and release a texture object using the ReleaseCOM macro:

```
IDirect3DTexture9 *pTexture = NULL;
D3DXCreateTextureFromFile(pDevice, "texture.bmp", &pTexture);
ReleaseCOM(pTexture);
```

In this code, the `ReleaseCOM` macro will release the `IDirect3DTexture9` interface and set the `pTexture` pointer back to NULL. Even if the texture failed to load and `pTexture` is NULL, calling

ReleaseCOM has no adverse effects.

Going along, the next helper function aids you by initializing Direct3D.

Initializing Direct3D

Next in line for the helper functions is `InitD3D`, which you use to initialize Direct3D and create a 3D device and display window. I tried to keep the code as simple as possible, performing the typical initialization code you would use in any Direct3D application, but in order to make the function work with all the demos in this book, I had to add a couple little extras.

To start with, the `InitD3D` function uses five parameters (and a standard COM `HRESULT` return code), as shown here in the function prototype:

```
HRESULT InitD3D(IDirect3D9 **ppD3D,
                IDirect3DDevice9 **ppD3DDevice,
                HWND hWnd,
                BOOL ForceWindowed = FALSE,
                BOOL MultiThreaded = FALSE);
```

As I show you the function's code, you'll gain an understanding of what each parameter does. Starting off, there are a few variables that are used throughout the `InitD3D` function:

```
HRESULT InitD3D(IDirect3D9 **ppD3D,
                IDirect3DDevice9 **ppD3DDevice,
                HWND hWnd,
                BOOL ForceWindowed,
                BOOL MultiThreaded)
{
   IDirect3D9       *pD3D       = NULL;
   IDirect3DDevice9 *pD3DDevice = NULL;
   HRESULT           hr;
```

In this code bit you see the local `IDirect3D9` and `IDirect3DDevice9` objects used to initialize Direct3D and the 3D device. These variables later are stored in the `ppD3D` and `ppD3DDevice` objects you pass to the `InitD3D` function. Finally, there is an `HRESULT` variable that contains the return code of any Direct3D calls made. If any call returns an error (as determined by the `FAILED` macro), the result code is returned to the caller of the `InitD3D` function.

The very first thing the `InitD3D` function does is make sure you have passed valid pointers to your Direct3D object, 3D device object, and window handle. Failing to do so forces `InitD3D` to return an error code. From there, `Direct3DCreate9` is called to create the Direct3D interface and store the resulting pointer in the `ppD3D` pointer (supplied by you when you called `InitD3D`).

```
// Error checking
 if(!ppD3D || !ppD3DDevice || !hWnd)
  return E_FAIL;

 // Initialize Direct3D
 if((pD3D = Direct3DCreate9(D3D_SDK_VERSION)) == NULL)
   return E_FAIL;
 *ppD3D = pD3D;
```

So far, nothing out of the ordinary. What's coming up, however, might throw you for a loop. The demo programs on this book's CD−ROM give you the option of running in a window or full−screen. Sometimes programs must run in a window, so to make sure the demo runs in a window, there is a `ForceWindowed` flag in the `InitD3D` prototype. When the flag is set to `FALSE`, the user is asked if he would like to use a full−screen video mode. If `ForceWindowed` is set to `TRUE`, the user is not asked to run in full−screen mode, and the `InitD3D` function assumes windowed mode was selected.

The following bit of code will examine the `ForceWindowed` flag; if it is set to `FALSE`, the function will display a message and wait for the user to select whether or not they want to use full−screen mode. If the `ForceWindowed` flag is set to `TRUE`, or if the user chooses to use windowed mode, `Mode` is set to `IDNO`; otherwise, `Mode` is set to `IDYES`, meaning that the application is to run in full−screen mode.

```
// Ask if user wants to run windowed or fullscreen
// or force windowed if flagged to do such
int Mode;
if(ForceWindowed == TRUE)
   Mode = IDNO;
else
  Mode = MessageBox(hWnd,                                \
                    "Use fullscreen mode? (640x480x16)", \
                    "Initialize D3D",                    \
                    MB_YESNO | MB_ICONQUESTION);
```

Now, if the user chooses to use a full−screen video mode (I use 640×480×16), then the appropriate presentation parameters are set using the standard methods, as seen in the DX SDK samples.

```
// Set the video (depending on windowed mode or fullscreen)
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory(&d3dpp, sizeof(d3dpp));

// Setup video settings based on choice of fullscreen or not
if(Mode == IDYES) {

    //////////////////////////////////////////////////////////
    // Setup fullscreen format (set to your own if you prefer)
    //////////////////////////////////////////////////////////
    DWORD     Width  = 640;
    DWORD     Height = 480;
    D3DFORMAT Format = D3DFMT_R5G6B5;

    // Set the presentation parameters (use fullscreen)
    d3dpp.BackBufferWidth  = Width;
    d3dpp.BackBufferHeight = Height;
    d3dpp.BackBufferFormat = Format;
    d3dpp.SwapEffect       = D3DSWAPEFFECT_FLIP;
    d3dpp.Windowed         = FALSE;
    d3dpp.EnableAutoDepthStencil = TRUE;
    d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
    d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
    d3dpp.PresentationInterval       = D3DPRESENT_INTERVAL_DEFAULT;
} else {
```

If the user chooses to use the windowed video mode, or if the `ForceWindowed` flag was set to `TRUE`, then a windowed mode is used instead of full−screen. Before setting up the appropriate presentation data as you did in full−screen mode, the client area of the window is resized to 640×480. From there, you set up the presentation parameters as you normally do for a windowed mode application.

```
//////////////////////////////////////////////////////////
```

```
  // Setup windowed format (set to your own dimensions below)
  //////////////////////////////////////////////////////////

  // Get the client and window dimensions
  RECT ClientRect, WndRect;
  GetClientRect(hWnd, &ClientRect);
  GetWindowRect(hWnd, &WndRect);

  // Set the width and height (set your dimensions here)
  DWORD DesiredWidth = 640;
  DWORD DesiredHeight = 480;
  DWORD Width  = (WndRect.right − WndRect.left) +        \
                 (DesiredWidth  − ClientRect.right);
  DWORD Height = (WndRect.bottom − WndRect.top) +        \
                 (DesiredHeight − ClientRect.bottom);

  // Set the window's dimensions
  MoveWindow(hWnd, WndRect.left, WndRect.top,            \
             Width, Height, TRUE);

  // Get the desktop format
  D3DDISPLAYMODE d3ddm;
  pD3D−>GetAdapterDisplayMode(D3DADAPTER_DEFAULT, &d3ddm);

  // Set the presentation parameters (use windowed)
  d3dpp.BackBufferWidth  = DesiredWidth;
  d3dpp.BackBufferHeight = DesiredHeight;
  d3dpp.BackBufferFormat = d3ddm.Format;
  d3dpp.SwapEffect       = D3DSWAPEFFECT_DISCARD;
  d3dpp.Windowed         = TRUE;
  d3dpp.EnableAutoDepthStencil = TRUE;
  d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
  d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
  d3dpp.PresentationInterval       = D3DPRESENT_INTERVAL_DEFAULT;
}
```

At this point, you're ready to initialize the 3D device using the `IDirect3D::CreateDevice` function. In the following code, `CreateDevice` is called, specifying the `D3DCREATE_MIXED_VERTEXPROCESSING` flag, as well as the `D3DCREATE_MULTITHREADED` flag if you set `MultiThreaded` to `TRUE` in your call to `InitD3D`.

```
  // Create the 3−D device
  DWORD Flags= D3DCREATE_MIXED_VERTEXPROCESSING;
  if(MultiThreaded == TRUE)
    Flags |= D3DCREATE_MULTITHREADED;
  if(FAILED(hr = pD3D−>CreateDevice(
                     D3DADAPTER_DEFAULT,
                     D3DDEVTYPE_HAL, hWnd, Flags,
                     &d3dpp, &pD3DDevice)))
     return hr;
  // Store the 3−D device object pointer
  *ppD3DDevice = pD3DDevice;
```

At the end of the last code bit, you can see that I've stored the resulting 3D device pointer in `ppD3DDevice`, which you passed to `InitD3D`. From here on out, things are pretty simple−you set your projection transformation matrix using `D3DXMatrixPerspectiveFovLH` to compute the transformations and `IDirect3DDevice9::SetTransform` to set it.

```
  // Set the perspective projection
  float Aspect = (float)d3dpp.BackBufferWidth / (float)d3dpp.BackBufferHeight;
  D3DXMATRIX matProjection;
  D3DXMatrixPerspectiveFovLH(&matProjection, D3DX_PI/4.0f, Aspect, 1.0f, 10000.0f);
  pD3DDevice−>SetTransform(D3DTS_PROJECTION, &matProjection);
```

Finally, in the `InitD3D` function, you set your default lighting, z−buffer, and alpha states. Here I'm disabling lighting, alpha blending, and alpha testing, while enabling z−buffering. Also, the default texture states are set to use modulation, and the texture samplings are set to linear min/magnification.

```
  // Set the default render states
  pD3DDevice−>SetRenderState(D3DRS_LIGHTING,         FALSE);
  pD3DDevice−>SetRenderState(D3DRS_ZENABLE,          D3DZB_TRUE);
  pD3DDevice−>SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
  pD3DDevice−>SetRenderState(D3DRS_ALPHATESTENABLE,  FALSE);

  // Set the default texture stage states
  pD3DDevice−>SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
  pD3DDevice−>etTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
  pD3DDevice−>etTextureStageState(0, D3DTSS_COLOROP,   D3DTOP_MODULATE);

  // Set the default texture filters
  pD3DDevice−>SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
  pD3DDevice−>SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);

  return S_OK;
}
```

Now that you've seen the code, how about learning how to put the `InitD3D` function to use? I tried to make `InitD3D` as easy as possible to use, so the following bit of code should work for the majority of your programs.

```
// Declare the 3−D device and Direct3D objects you'll be using
IDirect3D9 *pD3D = NULL;
IDirect3DDevice9 *pD3DDevice = NULL;

// Initialize the video mode, asking the user if they wish
// to use fullscreen or not.
InitD3D(&pD3D, &pD3DDevice, hWnd);
```

As you go through the code for the demos in this book, you'll see how the majority of them use the previously shown code to initialize Direct3D. Only one application uses multi−threading, and another forces windowed mode. You'll get a feel for using `InitD3D` fairly quickly.

Let's move on to the next helper function, one that helps you load your vertex shaders and set up your vertex declaration.

Loading Vertex Shaders

Moving on in the list of helper functions, you'll find `LoadVertexShader`. That's right, with all the vertex shader action going on in this book, you'll use this function to help you load your vertex shaders, as well as prepare your vertex shader declarations.

Take a peek at the `LoadVertexShader` function prototype:

```
HRESULT LoadVertexShader(
                 IDirect3DVertexShader9 **ppShader,
                 IDirect3DDevice9 *pDevice,
                 char *Filename,
                 D3DVERTEXELEMENT9 *pElements = NULL,
                 IDirect3DVertexDeclaration9 **ppDecl = NULL);
```

The actual code to the `LoadVertexShader` function is short, so instead of breaking it up to explain it, I'll give it to you all at once.

```
HRESULT LoadVertexShader(IDirect3DVertexShader9 **ppShader,
                         IDirect3DDevice9 *pDevice,
                         char *Filename,
                         D3DVERTEXELEMENT9 *pElements,
                         IDirect3DVertexDeclaration9 **ppDecl)
{
  HRESULT hr;

   // Error checking
   if(!ppShader || !pDevice || !Filename)
   return E_FAIL;

   // Load and assemble the shader
   ID3DXBuffer *pCode;
   if(FAILED(hr=D3DXAssembleShaderFromFile(Filename, NULL,   \
                                           NULL, 0,          \
                                           &pCode, NULL)))
      return hr;
   if(FAILED(hr=pDevice->CreateVertexShader(                 \
            (DWORD*)pCode->GetBufferPointer(), ppShader)))
     return hr;
   pCode->Release();

   // Create the declaration interface if needed
   if(pElements && ppDecl)
     pDevice->CreateVertexDeclaration(pElements, ppDecl);

   // Return success
   return S_OK;
}
```

After first checking to make sure the parameters you have passed to the `LoadVertexShader` function are valid, execution continues by loading and assembling the vertex shader via a call to `D3DXAssembleShaderFromFile`. Using the `D3DXBUFFER` object returned from that function call, you then use the `IDirect3DDevice::CreateVertexShader` function to create your vertex shader object (to which the pointer is stored in the `ppShader` object you passed to `LoadVertexShader`).

Finishing up `LoadVertexShader`, you'll see the call to `CreateVertexDeclaration`, which you use to create an `IDirect3DVertexDeclaration9` interface from the supplied array of vertex elements (`pElements` in the `LoadVertexShader` prototype). The vertex declaration object pointer is then stored in the `ppDecl` pointer you provide.

To use `LoadVertexShader`, pass it a pointer to an `IDirect3DVertexShader9` object you want to create, along with a valid `IDirect3DDevice9` object and file name of the vertex shader file. The last two parameters (`pElements` and `ppDecl`) are optional. By passing a valid `D3DVERTEXELEMENT9` array and the `IDirect3DVertexDeclaration9` object pointer, you can prepare your vertex declarations for use with the vertex shader being loaded.

Here's a small example of using LoadVertexShader. First, I declare an array of vertex elements that are used to create the vertex declaration object.

```
// Declare the vertex shader declaration elements
D3DVERTEXELEMENT9 Elements[] =
{
  { 0, 0,  D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, \
           D3DDECLUSAGE_POSITION, 0 },
  { 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, \
  D3DDECLUSAGE_NORMAL, 0 },
  { 0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, \
           D3DDECLUSAGE_TEXCOORD, 0 },
  D3DDECL_END()
};
```

Then I instance the vertex shader and vertex declaration objects and call LoadVertexShader.

```
// Instance objects
IDirect3DVertexShader9      *pShader = NULL;
IDirect3DVertexDeclaration9 *pDecl   = NULL;

// Load the vertex shader and create declaration interface
LoadVertexShader(&pShader, pDevice,                          \
                 "Shader.vsh",                               \
                 &Elements, &pDecl);
```

As you can see, it's a quick and simple function that gets the job done. From here on out, you can set the vertex shader (represented by pShader) using the IDirect3DDevice9::SetVertexShader function. To set the vertex declaration (represented by pDecl), you would call IDirect3DDevice9::SetVertexDeclaration.

```
pD3DDevice->SetFVF(NULL); // Clear FVF usages
pD3DDevice->SetVertexShader(pShader);
pD3DDevice->SetVertexDeclaration(pDecl);
```

Okay, enough of initializing and loading vertex shaders, let's move on to the cool stuff, like loading and rendering meshes.

Loading Meshes

The first of the mesh−related helper functions is LoadMesh. Actually there are three versions of the LoadMesh function. The first version is used to load a mesh from an .X file using the D3DXLoadMeshFromX function. That means all meshes contained within the .X file are compressed into a single mesh object, which is subsequently stored in a D3DXMESHCONTAINER_EX object.

All iterations of the LoadMesh function contain pointers to a valid 3D device object, the directory path to where your meshes' textures are stored, and the mesh loading flags and optional flexible vertex format that the LoadMesh functions use to clone the meshes after loading. That means you can force your loaded meshes to use specific vertex formats!

Here's the prototype of the first LoadMesh function:

```
HRESULT LoadMesh(D3DXMESHCONTAINER_EX **ppMesh,
                 IDirect3DDevice9 *pDevice,
                 char *Filename,
                 char *TexturePath = ".\\",
                 DWORD NewFVF = 0,
                 DWORD LoadFlags = D3DXMESH_SYSTEMMEM);
```

The first `LoadMesh` function takes a pointer to a `D3DXMESHCONTAINER_EX` object pointer that you want to use for storing the loaded mesh data. Notice I said *pointer to a pointer*. The `LoadMesh` function will allocate the appropriate objects for you and store the pointers in the pointer you pass to `LoadMesh`. This is similar to the way the `InitD3D` function stores the Direct3D and 3D device object pointers.

Also, you must pass a valid 3D device object to the `LoadMesh` function (as the `pDevice` pointer)−you use this device object to create the mesh container and texture buffers. The mesh that you want to load is specified as `Filename`, and the directory in which your textures are located is specified in `TexturePath`. This texture directory path is prefixed to any texture file names as they are loaded.

Finally, there are `NewFVF` and `LoadFlags`. You use the `NewFVF` parameter to force the mesh being loaded to use a specific FVF. For instance, if you only wanted to use 3D coordinates and normals, then you would set `NewFVF` to (`D3DFVF_XYZ|D3DFVF_NORMAL`). The `LoadMesh` function will use `CloneMeshFVF` to clone the mesh using the specific FVF you specified.

The `LoadFlags` parameter is used to set the mesh loading flags as specified by the `D3DXLoadMeshFromX` function in the DX SDK documents. The default value for this parameter is `D3DXMESH_SYSTEMMEM`, meaning that the mesh is loaded into system memory (as opposed to hardware memory).

Instead of showing you the entire code for the `LoadMesh` function, I'll just skim over the most important parts. For the full code, consult the code from the book−look in Direct3D.cpp for the first `LoadMesh` function listed.

Typical of most mesh−loading functions that you may already be using, the first `LoadMesh` function uses the `D3DXLoadMeshFromX` function, as shown here:

```
  // Load the mesh using D3DX routines
  ID3DXBuffer *MaterialBuffer = NULL, *AdjacencyBuffer = NULL;
  DWORD NumMaterials;
  if(FAILED(hr=D3DXLoadMeshFromX(Filename, TempLoadFlags,       \
                                 pDevice, &AdjacencyBuffer,     \
                                 &MaterialBuffer, NULL,         \
                                 &NumMaterials, &pLoadMesh)))
  return hr;
```

A couple of notes on the parameters for `D3DXLoadMeshFromX:`

- ♦ `Filename` specifies the file name of the .X file to load.
- ♦ `pDevice` is your 3D device object.
- ♦ `AdjacencyBuffer` is the `ID3DXBUFFER` object that will contain the face adjacency data.
- ♦ `MaterialBuffer` stores the material data (with `NumMaterials` holding the number of meshes loaded).
- ♦ `pLoadMesh` will store the `ID3DXMesh` object loaded.

Not mentioned in the previous list is `TempLoadFlags`, which are the flags used to load the mesh. If you are using a new FVF (as specified in `NewFVF`), then the mesh is forced into system memory using the `D3DXMESH_SYSMEMORY` flag (to allow the cloning to succeed); otherwise, the mesh is loaded using the flags you specify in the `LoadFlags` parameters of the call to `LoadMesh`.

Speaking of cloning the mesh, if you do specify a non−zero value in `NewFVF`, the `LoadMesh` function will attempt to clone the mesh using the FVF specified. If successful, it will replace the `pLoadMesh` pointer with the newly cloned mesh. Here's a small code snippet that demonstrates this cloning feature:

```
// Convert to new FVF first as needed
if(NewFVF) {
  ID3DXMesh *pTempMesh;

  // Use CloneMeshFVF to convert mesh
  if(FAILED(hr=pLoadMesh->CloneMeshFVF(LoadFlags, NewFVF, pDevice, &pTempMesh))) {
    ReleaseCOM(AdjacencyBuffer);
    ReleaseCOM(MaterialBuffer);
    ReleaseCOM(pLoadMesh);
    return hr;
  }

  // Free prior mesh and store new pointer
  ReleaseCOM(pLoadMesh);
  pLoadMesh = pTempMesh; pTempMesh = NULL;
}
```

Now, aside from the typical material−loading function calls that you should already be familiar with when loading meshes with D3DX, there are two more important aspects to the first `LoadMesh` function−allocating the `D3DXMESHCONTAINER_EX` structure (and filling it with the necessary data) and optimizing the mesh's faces.

Remember that earlier in this chapter I mentioned that the `D3DXMESHCONTAINER_EX` structure contains a variety of information about a single mesh, including the mesh's name, mesh object pointer, type of mesh (such as regular, skinned, or progressive), face adjacency data, and material and texture data. The following code demonstrates allocating the `D3DXMESHCONTAINER_EX` object and setting the appropriate data:

```
// Allocate a D3DXMESHCONTAINER_EX structure
D3DXMESHCONTAINER_EX *pMesh = new D3DXMESHCONTAINER_EX();
*ppMesh = pMesh;

// Store mesh name (filename), type, and mesh pointer
pMesh->Name = strdup(Filename);
pMesh->MeshData.Type = D3DXMESHTYPE_MESH;
pMesh->MeshData.pMesh = pLoadMesh; pLoadMesh = NULL;

// Store adjacency buffer
DWORD AdjSize = AdjacencyBuffer->GetBufferSize();
if(AdjSize) {
  pMesh->pAdjacency = new DWORD[AdjSize];
  memcpy(pMesh->pAdjacency,                          \
  AdjacencyBuffer->GetBufferPointer(), AdjSize);
}
ReleaseCOM(AdjacencyBuffer);
```

At this point, the material data is loaded into the `D3DXMESHCONTAINER_EX` object. (I'll skip the code because it's basic stuff.) Again, consult the full source code and you'll see that I'm merely loading the materials and texture using the techniques you've seen a million times in the DX SDK demos.

As for the optimization of the mesh's faces that I mentioned, this is an important step that ensures the mesh's face data is readily available to you when you want to render the mesh's polygons manually (instead of calling `DrawSubset`). Chapter 8, "Working with Morphing Animation," details using mesh face data, so for now I'll just show you the function call that optimizes the faces for you.

```
  // Optimize the mesh for better attribute access
  pMesh->MeshData.pMesh->OptimizeInplace(                      \
                D3DXMESHOPT_ATTRSORT, NULL, NULL, NULL, NULL);
```

And that's it for the first `LoadMesh` function! Let's check out how to use it. Suppose you want to load a mesh (from a file called Mesh.x) using the `LoadMesh` function just shown. To demonstrate the ability to specify a new FVF, specify that you want to use XYZ components, normals, and texture coordinates for your mesh. Also, suppose your textures are in a subdirectory called \textures. As for the mesh loading flags, leave those alone to allow the mesh to load into system memory (as per the default flag shown in the prototype). Here's the code:

```
// Instance the mesh object
D3DXMESHCONTAINER_EX *Mesh = NULL;

// Load a mesh - notice the pointer to the mesh object
LoadMesh(&Mesh, pD3DDevice, "Mesh.x", "..\\Textures\\",      \
        (D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_TEX1));
```

Once the mesh has been loaded, you can access the mesh object via the `Mesh->MeshData.pMesh` object pointer. Also, material data is stored in `Mesh->pMaterials`, and texture data is stored in `Mesh->pTextures`. The number of materials a mesh uses is stored in `Mesh->>NumMaterials`. To render a loaded mesh, you can use the following code:

```
// pMesh = pointer to D3DXMESHCONTAINER_EX object

// Go through all material subsets
for(DWORD i=0;i<pMesh->NumMaterials;i++) {

  // Set material and texture
  pD3DDevice->SetMaterial(&pMesh->pMaterials[i].MatD3D);
  pD3DDevice->SetTexture(0, pMesh->pTextures[i]);

  // Draw the mesh subset
  pDrawMesh->DrawSubset(i);
}
```

The second `LoadMesh` function used in this book is much like the first, except that instead of loading an entire .X file into one `D3DXMESHCONTAINER_EX` object, you are able to load a single mesh object (using the `D3DXLoadSkinMeshFromXof` function) as pointed to by a `IDirectXFileDataObject` object (used while parsing an .X file). Here's the prototype:

```
HRESULT LoadMesh(D3DXMESHCONTAINER_EX **ppMesh,
                 IDirect3DDevice9 *pDevice,
                 IDirectXFileData *pDataObj,
                 char *TexturePath = ".\\",
                 DWORD NewFVF = 0,
                 DWORD LoadFlags = D3DXMESH_SYSTEMMEM);
```

You'll notice that the `pDataObj` parameter is here instead of the `Filename` parameter used by the first `LoadMesh` function. The `pDataObj` parameter is of the type `IDirectXFileData`, which is an object that represents the currently enumerated data object inside an .X file. If you're not familiar with .X file data objects, you might want to check out Chapter 3, "Using the .X File Format."

Calling the second LoadMesh function is the same as calling the first (with the exception of providing a pointer to the enumerated data object), so I'll skip over using the second LoadMesh function and leave that for later in the book. For now, let's move on to the next version of the LoadMesh function.

Note You'll notice that I mentioned using the D3DXLoadSkinMeshFromXof function (instead of using D3DXLoadMeshFromXof), which loads a skinned mesh data object from the .X file data object you specify. The reason why is that D3DXLoadSkinMeshFromXof loads both regular meshes (those without skinning data) and skinned meshes for you, returning a valid pointer to an ID3DXMesh object either way. You'll read about skinned mesh objects in Chapter 4.

The third and final version of the LoadMesh function is the most advanced. It allows you to load all meshes and frames from an .X file at once. The meshes are stored in a linked list (pointed to by the root D3DXMESHCONTAINER_EX object you pass to LoadMesh), and by a hierarchy of D3DXFRAME_EX objects (also pointed to by the root object you pass to LoadMesh).

Here's the prototype for the third LoadMesh function:

```
HRESULT LoadMesh(D3DXMESHCONTAINER_EX **ppMesh,
                 D3DXFRAME_EX **ppFrame,
                 IDirect3DDevice9 *pDevice,
                 char *Filename,
                 char *TexturePath = ".\\",
                 DWORD NewFVF = 0,
                 DWORD LoadFlags = D3DXMESH_SYSTEMMEM);
```

Okay, this third function is going to take a little explaining. First there's the function's usage. Much like the first LoadMesh function, you need to specify a file name of the .X file from which you are loading the data, as well as a pointer to your 3D device, texture storage path, optional FVF override flags, and optional mesh storage flags. Also, there's the D3DXMESHCONTAINER_EX pointer, which ends up containing a linked list of meshes once it is loaded.

New to the LoadMesh function (from previous iterations of it) is the D3DXFRAME_EX pointer, which is similar to the D3DXMESHCONTAINER_EX object except that instead of storing a linked list of meshes, it is filled with the frame data from the .X file you are loading.

To call the third LoadMesh function, specify the loading parameters much as you did for the first LoadMesh function. This time, however, add in the D3DXFRAME_EX object pointer:

```
// Root mesh and frame objects that are being loaded
D3DXMESHCONTAINER_EX *pMesh = NULL;
D3DXFRAME_EX *pFrame = NULL;

// Load the meshes and frames from Mesh.x
LoadMesh(&pMesh, &pFrame, pD3DDevice, "Mesh.x");
```

When it is complete, the pMesh pointer will contain a pointer to the root mesh object (in a linked list of mesh objects), and pFrame will point to the root frame in the hierarchy. To learn more about hierarchies, check out Chapter 4.

You can also use the third LoadMesh function to load a series of meshes from an .X file, without dealing with any frames. Do this by specifying a NULL value for the frame pointer. On the flip side, you can also force LoadMesh to skip loading any of the meshes by setting the mesh pointer to NULL in your call to LoadMesh.

This ability to decide which components to load (the meshes and/or the frames) makes the third iteration of `LoadMesh` the most useful, as well as the most difficult to understand. In fact, it's really too difficult to explain at this point. You see, the third `LoadMesh` function uses a custom .X parser created using the techniques I will show you in Chapter 3.

Basically, the custom .X parser is scanning the .X file you specify for specific mesh and frame−related data. As these data bits are found, the parser decides what to do−load the mesh using the second `LoadMesh` function (loading via an `IDirectXFileData` object) or load a frame (and its transformation matrix, if found). Either way, if the data is used, an appropriate structure is allocated (either `D3DXMESHCONTAINER_EX` or `D3DXFRAME_EX`) and linked to the appropriate linked list of objects. The pointer to these linked lists is returned to you via the `LoadMesh` parameters you specify (as previously detailed).

Once you've read up on using frames and custom .X file parsers later in the book, you'll find that the third `LoadMesh` function is pretty simple, and once again, it should become one of your most used mesh−loading helper functions.

And that wraps up the mesh−loading helper functions! After you've gone to the trouble of loading these meshes, it comes time to render them to the display, but first you need to update the vertex data if you are using skinned meshes. The next helper function makes updating skinned meshes (a topic you'll read about in Chapter 4) easy as pie. Take a look.

Updating Skinned Meshes

A skinned mesh works like this: Each vertex is attached to an imaginary bone (which is specified by a frame object). As these frames move, so do the vertices attached to them. To update the coordinates of the vertices as the bones move, you need to call a special function that takes the source vertex data, transforms it according to the bones' transformations, and stores the results in a second mesh object. This special function is called `ID3DXSkinInfo::UpdateSkinnedMesh`.

Whenever you load a mesh using the `D3DXLoadSkinMeshFromXof` function (which is what the second `LoadMesh` function does), you get a pointer to an `ID3DXSkinInfo` object. This object contains the information about which vertices are attached to which bones. This way, the object knows which transformations to apply to the vertices.

To update the vertices, you must first lock the mesh's vertex buffer (which contains the source vertex coordinates), as well as the destination mesh's vertex buffer. The destination mesh will receive the updated vertices as they are transformed. Once locked, you need to call `UpdateSkinnedMesh`, also specifying a series of transformation matrices (stored as `D3DXMATRIX` objects) that represent the various bone transformations.

This will all make sense when you get around to working with skeletal meshes in Chapter 4. For now, just check out the `UpdateMesh` helper function code to see how it updates the skinned meshes for you.

```
HRESULT UpdateMesh(D3DXMESHCONTAINER_EX *pMesh)
{
  // Error checking
  if(!pMesh)
    return E_FAIL;
  if(!pMesh−>MeshData.pMesh || !pMesh−>pSkinMesh || !pMesh−>pSkinInfo)
    return E_FAIL;
  if(!pMesh−>pBoneMatrices || !pMesh−>ppFrameMatrices)
```

```
    return E_FAIL;

  // Copy the bone matrices over (must have been combined before call DrawMesh)
  for(DWORD i=0;i<pMesh->pSkinInfo->GetNumBones();i++) {
    // Start with bone offset matrix
    pMesh->pBoneMatrices[i] = (*pMesh->pSkinInfo->GetBoneOffsetMatrix(i));
```

Aside from the typical error−checking code, the `UpdateMesh` function starts by looping through each bone contained within the `ID3DXSkinInfo` object (stored in the `D3DXMESHCONTAINER_EX` object you've already loaded). For each bone, the original transformation matrix from the .X file is grabbed and stored in an array of matrices used in the call to `UpdateSkinnedMesh`.

From here the bone's transformation, as stored in the bone's respective frame object, is applied to the transformation matrix. This process continues until all transformation matrices are set up.

```
  // Apply frame transformation
  if(pMesh->ppFrameMatrices[i])
    pMesh->pBoneMatrices[i] *= (*pMesh->ppFrameMatrices[i]);
  }
```

At this point, you are ready to lock the vertex buffers and call the `UpdateSkinnedMesh` function.

```
  // Lock the meshes' vertex buffers
  void *SrcPtr, *DestPtr;
  pMesh->MeshData.pMesh->LockVertexBuffer(D3DLOCK_READONLY,          \
                                          (void**)&SrcPtr);
  pMesh->pSkinMesh->LockVertexBuffer(0, (void**)&DestPtr);

  // Update the skinned mesh using provided transformations
  pMesh->pSkinInfo->UpdateSkinnedMesh(pMesh->pBoneMatrices,          \
                                      NULL, SrcPtr, DestPtr);
```

The function is finished by unlocking the buffers and returning a success code.

```
  // Unlock the meshes vertex buffers
  pMesh->pSkinMesh->UnlockVertexBuffer();
  pMesh->MeshData.pMesh->UnlockVertexBuffer();

  // Return success
  return S_OK;
}
```

I kind of flew over the specifics, but this really won't make much sense until you've read Chapter 4. After you have read that chapter, you'll find that the `UpdateMesh` function comes in handy for getting those skinned meshes updated and ready to render!

And once again speaking of rendering, it is finally time to see the helper functions I created to get those meshes on screen!

Drawing Meshes

Now that you have your meshes loaded and you've updated those skinned meshes that needed updating, it is time to throw some pixels at the display and show off those meshes! In total, I have created four mesh−rendering functions to help you in your projects, and I depend on these four functions to render the meshes in this book.

Here are the prototypes for the four mesh−rendering functions used in this book:

```
// Draw the first mesh in a linked list of objects
HRESULT DrawMesh(D3DXMESHCONTAINER_EX *pMesh);

// Draw the first mesh in a linked list of objects
// using the specified vertex shader and declaration
HRESULT DrawMesh(D3DXMESHCONTAINER_EX *pMesh,
                 IDirect3DVertexShader9 *pShader,
                 IDirect3DVertexDeclaration9 *pDecl);

// Draw all meshes in a linked list of objects
HRESULT DrawMeshes(D3DXMESHCONTAINER_EX *pMesh);

// Draw all meshes in a linked list of objects
// using the specified vertex shader and declaration
HRESULT DrawMeshes(D3DXMESHCONTAINER_EX *pMesh,
                   IDirect3DVertexShader9 *pShader,
                   IDirect3DVertexDeclaration9 *pDecl);
```

You'll see that the mesh−drawing functions are very similar in nature. The first two are used to render a single mesh that is stored in the `D3DXMESHCONTAINER_EX` object specified. I say single mesh because the mesh object might contain a linked list of mesh objects that are loaded. If you want to render a specific mesh, then use the first `DrawMesh` function.

I won't list the code for the `DrawMesh` functions here because they are pretty simple stuff. In the "Loading Meshes" section earlier, I showed you a quick code snippet that demonstrates rendering a mesh stored in a `D3DXMESHCONTAINER_EX` object. The `DrawMesh` functions duplicate this code with one exception−alpha blending is taken into consideration. That's right; if a material being used specifies an alpha value other than 1, then alpha blending is enabled. This way, you can specify portions of a mesh to use alpha blending by merely changing the material information. Also, if a `D3DXMESHCONTAINER_EX` object contains a skinned mesh, that mesh is rendered instead of the regular mesh.

As for the second `DrawMesh` function, it skips using the `DrawSubset` function and uses its own function to render subsets of polygon faces, using the vertex shader and vertex declaration you specify. This second function is extremely useful if you are using vertex shaders to render your meshes.

The remaining two `DrawMesh` functions duplicate the exact features of the first two, except that all meshes in the linked list of mesh objects are rendered. Once again, feel free to peruse the full source code for the `DrawMesh` functions, as well as all the functions discussed in this chapter, on the CD−ROM. As you read the book and check out the demo programs I created, you'll see how I've managed to put these four functions to good use.

Moving On with the Book

Whew! That's a lot to digest−installing DirectX, setting up your compiler, and dealing with the helper code−how are you possibly going to remember all this before moving on? Just take everything one step at a time, my friend, and you'll be fine.

As I mentioned, the helper functions don't really do anything except duplicate what you most likely have done a thousand times before. If you feel more comfortable, I would recommend changing the helper functions to suit your needs. For example, if you've already got your own series of helper functions, just map them right over to those used by this book's demos. Either that or just rewrite the functions. I'm sure you're just dying to add that nifty code that displays an entire list of video modes from which the users can pick in the demos, right?

If you get stuck, just turn back to this chapter as a quick reference guide to using the helper code, and if you're really stuck, just fire off an e−mail to me. I'm more than happy to help out when I can!

Programs on the CD

In the \common directory of this book's CD−ROM, you'll find the helper code source files that were discussed in this chapter. These files include

- **Direct3D.cpp/.h.** These two files are used in nearly every project in this book. The Direct3D.cpp file contains functions to initialize Direct3D, load meshes and vertex shaders, and render meshes. The Direct3D.h file includes a couple objects used to contain frame hierarchies and mesh data (both regular and skinned meshes).
- **XFile.cpp/.h.** Also included with every project in the book, this pair of files is used to include the rmxftmpl.h and rmxfguid.h files in your projects. Why not include those files directly, you ask? Because it generates compiler errors if you try, so add the XFile.cpp file to your project and include XFile.h file in your source code instead!
- **XParser.cpp/.h.** Also used throughout the book, these two files are useful when parsing .X files in your projects. Together, these files define a base class object that you can derive to fit any .X parsing needs. Consult Chapter 3 for details on using .X files and the classes defined in these source files.

Part Two: Animation Basics

Chapter 2: Timing in Animation and Movement

Games are packed with movement. Characters running here, bullets flying there−basically there's a slew of objects moving about your game world. The smooth motion of these objects is a very important aspect that can't be overlooked. Have you ever thought about the use of movement and animation based on time? Using time−based motion is hot, and to keep up with the rest of the world you must fully understand how it can help your game project. In fact, you need to understand not only using time−based motion, but also motion in general. Do you think only characters move about in your game? Nope, your in−game cinematic cameras need your guidance as well. This chapter will show you how to use time−based motion in your projects.

Using Time−Based Motion

Although it might not seem important at first, timing plays a crucial role in your game projects. I'm not talking about the time of day; rather, I'm referring to down−to−the−millisecond timing of animation. This sort of precise timing is required for smooth animation and movement of objects in your project. Although it is a basic topic, it is one that all game programmers should understand quite well.

When it comes to moving your meshes, time−based motion is the best of the bunch. Throughout this book, I use time−based motion mainly to control the speed of the various animations. In this chapter, I want to explain this use of time (for animation and in movement) in better detail.

The whole basis of using time−based motion is simple−movement and animation always take the same amount of time to perform on any system, regardless of the computer's actual speed. For example, a 2−GHz system processing a 20−second animation will undoubtedly do so quickly and produce a very smooth motion, whereas the same animation running on a 500−MHz system will be choppy but will still maintain the 20−second animation length.

More than likely, the slower computer will drop specific frames of the animation to keep up with the speed of the faster computer. Therein lies the secret−slower computers can drop more frames and still maintain a somewhat reasonable representation of the actual motion.

Okay, I think you're starting to get the idea. As simple as the techniques sound, time−based animation and movement remain a slight mystery to fledgling game programmers. Even though this book is about using advanced animation techniques, time−based motion is something you should fully understand. For that reason, I want to introduce (or re−introduce) you to the world of time−based motion. I'll start with a brief look at reading time in Windows.

Reading Time in Windows

The easiest method of reading time in Windows is by using the `timeGetTime` function. This function does not take any parameters. As the following code demonstrates, `timeGetTime` only returns the number of milliseconds that have passed since Windows was started.

```
DWORD TimeSinceStarted = timeGetTime();
```

What good does this do you? Typically, you call the `timeGetTime` function once per frame that your game processes. For each frame, you then subtract the time from the last frame processed to obtain the number of milliseconds that have elapsed since your last frame. You can use this elapsed time to compute your time−based motion.

Storing the time of the last frame update is as easy as using a static variable. At the beginning of your frame's update function, insert a static variable that stores the current time.

```
void FrameUpdate()
{
  static DWORD LastTime = timeGetTime();
```

At the end of the `FrameUpdate` function, you can then store the current time in LastTime.

```
LastTime = timeGetTime();
```

In this manner (storing the current time at the end of your frame's update function), you have managed to store the time at which the frame update ended. During the next call to `FrameUpdate`, the `LastTime` variable will still contain the time at which the last frame update ended. Using this time value, you can calculate the amount of time that has elapsed since you last called `FrameUpdate` by subtracting `LastTime` from the current time.

```
DWORD ElapsedTime = timeGetTime() − LastTime;
```

It's this elapsed time and time of the last frame update that you'll use the most throughout this book. Now what about those instances when you want to calculate the number of elapsed milliseconds based on a specific time, such as when an animation starts, instead of counting the elapsed time or when the `FrameUpdate` function was first called?

Just as you used a static variable to store the last frame's update time, you can store the time at which the function was first called. Using that static variable, you can determine how many milliseconds have passed since the function was first called.

```
void FrameUpdate()
{
    static DWORD StartTime = timeGetTime();
    DWORD ElapsedTime = timeGetTime() − StartTime;

     // ElapsedTime is the number of milliseconds that has passed
     // since you first called FrameUpdate.
}
```

In the same way the previous bit of code tracked the total amount of time that has passed since the first call to the `FrameUpdate` function, you can embed the starting time of an animation inside your data structures. You'll get to see this concept demonstrated later on in Chapter 5, "Using Key−Framed Skeletal Animation."

If you read ahead in the book, you're probably already familiar with using time−based animation. All of the animation demos that come with this book are timed in milliseconds. Combined with key frames, time−based animation is a perfect solution for creating smooth animation. Read on to take a closer look at using time−based animation.

Animating with Time

In the olden days, games were made to animate graphics based on every frame processed. To ensure that the animations always ran at the same speed, those games sometimes limited the number of frames per second that could be processed. Of course, those old games were made for computers that couldn't easily process more than 20 to 30 frames per second, so it was safe to assume that limiting the number of frames per second would never surpass that 20 or 30 frames per second mark.

But that was then, and this is now. Modern computers can run circles around their ancestors, and limiting the number of frames to control animation is a definite no−no in this day and age. You need to base the speed of animation on the amount of time that has elapsed since the start of the animation sequence. Doing so is no problem because you already know that you can record the time when the animation started. Additionally, for each frame to update, you can read the current time and subtract the starting animation time. The result is a time value to use as an offset to your animation sequence.

Suppose you are using time−based key frames in your animation engine. You can use a simple key−frame structure that stores the time and a transformation matrix to use, such as this:

```
 typedef struct sKeyframe {
      DWORD Time;
      D3DMATRIX matTransformation;
 } sKeyframe;
```

As is typical for key frames, you can store an array of matrices, each with its own unique time. These structures are stored in chronological order, with the lower time values first. Therefore, you can create a small sequence of transformations to orient an object over time (see Figure 2.1).

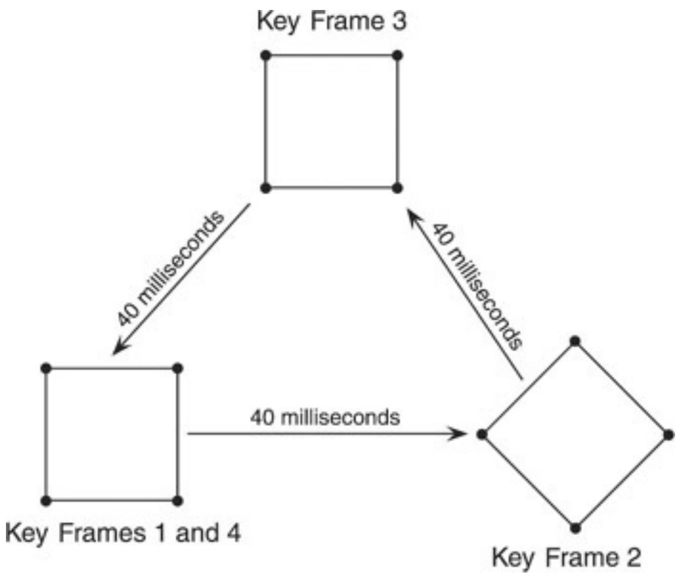


Figure 2.1: Key frames track the orientation of the cube. Each key frame is spaced 400 milliseconds from the next, and interpolation is used to calculate the in−between orientations.
To replicate the key frames shown in Figure 2.1, I've constructed the following array:

```
sKeyframe Keyframes[4] = {
  { 0,  1.00000f, 0.00000f, 0.00000f, 0.00000f,
        0.00000f, 1.00000f, 0.00000f, 0.00000f,
        0.00000f, 0.00000f, 1.00000f, 0.00000f,
        0.00000f, 0.00000f, 0.00000f, 1.00000f; },
  { 400, 0.000796f, 1.00000f, 0.00000f, 0.00000f,
        −1.00000f, 0.000796f, 0.00000f, 0.00000f,
        0.00000f, 0.00000f, 1.00000f, 0.00000f,
        50.00000f, 0.00000f, 0.00000f, 1.00000f; },
  { 800, −0.99999f, 0.001593f, 0.00000f, 0.00000f,
        −0.001593f, −0.99999f, 0.00000f, 0.00000f,
        0.00000f, 0.00000f, 1.00000f, 0.00000f,
        25.00000f, 25.00000f, 0.00000f, 1.00000f; },
  { 1200, 1.00000f, 0.00000f, 0.00000f, 0.00000f,
         0.00000f, 1.00000f, 0.00000f, 0.00000f,
         0.00000f, 0.00000f, 1.00000f, 0.00000f,
         0.00000f, 0.00000f, 0.00000f, 1.00000f; }
  };
```

Now comes the fun part. Using the timing methods you read about previously, you can record the time at which the animation started. And, for each frame to update the animation, you can calculate the elapsed time since the animation started (using that as an offset to the key frames). Create a simple frame update function that will determine which transformation to use depending on the elapsed time since the update function was first called.

```
void FrameUpdate()
{
  static DWORD StartTime = timeGetTime();
  DWORD Elapsed = timeGetTime() − StartTime;
```

With the elapsed time now in hand, you can scan the key frames to look for the two between which the time value lies. For example, if the current time is 60 milliseconds, the animation is somewhere between key frame #0 (at 0 milliseconds) and key frame #1 (at 400 milliseconds). A quick scan through the key frames determines which to use based on the elapsed time.

```
DWORD Keyframe = 0; // Start at 1st keyframe
for(DWORD i=0;i<4;i++) {
    // If time is greater or equal to a
    // key−frame's time then update the
    // keyframe to use
    if(Time >= Keyframes[i].Time)
       Keyframe = i;
}
```

At the end of the loop, the `Keyframe` variable will hold the first of the two key frames between which the animation time lies. If `Keyframe` isn't the last key frame in the array (in which there are four key frames), then you can add 1 to `Keyframe` to obtain the second key frame. If `Keyframe` is the last key frame in the array, you can use the same key−frame value in your calculations.

Using a second variable to store the next key frame in line is perfect. Remember that if `Keyframe` is the last key frame in the array, you need to set this new key frame to the same value.

```
DWORD Keyframe2 = (Keyframe==3) ? Keyframe:Keyframe + 1;
```

Now you need to grab the time values and calculate a scalar based on the time difference of the keys and the position of the key frame between the keys.

```
  DWORD TimeDiff = Keyframes[Keyframe2].Time −
                   Keyframes[Keyframe].Time;
  // Make sure there's a time difference to
  // avoid divide−by−zero errors later on.
  if(!TimeDiff)
    TimeDiff=1;
  float Scalar = (Time − Keyframes[Keyframe].Time)/TimeDiff;
```

You now have the scalar value (which ranges from 0 to 1) used to interpolate the transformation matrices of the keys. To make it easy to deal with the transformation matrices, those matrices are cast to a `D3DXMATRIX` type so that D3DX does the hard work for you.

```
// Calculate the difference in transformations
D3DXMATRIX matInt =                                       \
    D3DXMATRIX(Keyframes[Keyframe2].matTransformation) −  \
    D3DXMATRIX(Keyframes[Keyframe].matTransformation);
 matInt *= Scalar; // Scale the difference

// Add scaled transformation matrix back to 1st keyframe matrix
matInt += D3DXMATRIX(Keyframes[Keyframe].matTransformation);
```

At this point, you have the proper animated transformation matrix to use stored in `matInt`. To see your hard work come to life, set `matInt` as the world transformation and render your animated mesh.

As you can see, using time−based animation is pretty simple. Even if you don't use key frames in your animation, you can still rely on these methods of using time in your own code. Now that you've seen how easy it is to use time−based animation, take a look at how easy it is to use time−based movement.

Moving with Time

Time−based motion doesn't just apply to animation. Movement is also a major part of your game, and basing movement on time guarantees that all systems will run your game consistently, regardless of how fast or how slow they are.

The most common use for time−based movement is when you want to move an object a set distance over a period of time. For example, suppose a player moves his joystick to the right, so your game responds by moving the on−screen game character to the right a little bit−let's say 64 units over a period of one second, which equates to 0.064 units of movement per millisecond.

Using a small function, you can calculate the number of units (as a floating−point value) to move an object based on the elapsed time between frames.

```
float CalcMovement(DWORD ElapsedTime, float PixelsPerSec)
{
  return (PixelsPerSec / 1000.0f * (float)ElapsedTime);
}
```

As you can see in the CalculateMovement function, you are using the following calculation:

```
PixelsPerSec / 1000.0f * ElapsedTime;
```

The `PixelsPerSec` variable contains the number of units you want to move over the period of a second. The 1000.0 value means 1000 milliseconds. Basically, you're breaking down the number of units to move per millisecond. Finally, you need to multiply by ElapsedTime to calculate the total movement to apply.

This sort of movement based on time is very basic, but it should not be overlooked. Knowledge of this function of time−based movement is essential to using more advanced features, such as smoothly moving objects along a pre−determined path.

Moving along Paths

As you read in the previous section, time−based movement is determined by taking the distance to travel, dividing it by 1,000, and multiplying the result by the elapsed time. In that section, I used an example in which a player pressed right on the joystick, and his character moved right a set amount of units. But what about those times when you want an object to move without user intervention? For instance, suppose a player pushes a button and bullets fly out of the big gun he is carrying. Those bullets travel along a set path at a set speed. You can set a velocity for each of those bullets, negating the need to use paths, but what about those super−bullets in your game that can swoop through parts of your level, perhaps along a pre−set path?

Those special instances require you to set up the coordinates of the travel paths in advance, and to do some quick calculations to determine where an object can be placed inside those paths. And what about moving objects such as characters, power−ups, and platforms? You guessed it−using paths is the perfect solution for all your movement needs!

I am going to discuss two of the most frequently used types of paths−straight and curved. I will start by explaining how to use straight paths.

Following Straight Paths

A straight path is just that−straight. The path moves from the starting point to the ending point with no breaks or turns. Generally, you define a straight line using a pair of coordinates−the starting point and the ending point. To follow a straight path, you only need to walk along the line from Point A to Point B.

To move an object along a straight path, you must calculate the coordinates of a point along the line using some simple formulas. For instance, as Figure 2.2 illustrates, to calculate a point at the midpoint of the line using a scalar value (ranging from 0 to 1), you calculate the difference in the endpoint's coordinates, multiply by the scalar value, and add the result to the starting point's coordinates.

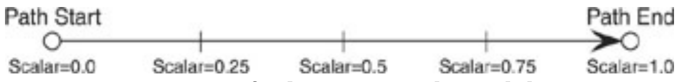


Figure 2.2: You can find a point on the path by using a scalar value that represents the percentage of the total length of the path, with 0 being the start and 1 being the end of the path.

```
// Define starting and ending points of straight path
// Scalar = position to calculate (0 to 1)
D3DXVECTOR3 vecStart = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 vecEnd = D3DXVECTOR3(10.0f, 20.0f, 30.0f);
D3DXVECTOR3 vecPos = (vecEnd − vecStart) * Scalar + vecStart;
```

If you were to set `Scalar` to 0.5, then `vecPos` would contain the coordinates 5.0, 10.0, 15.0, which happen to be the midpoint of the path. Now suppose you don't want to use a scalar value. What about using 3D units instead? For example, instead of using a scalar value of 0.5, suppose you want to know the coordinates of a point that is 32 units from the starting coordinates.

To calculate the coordinates using 3D units as a measurement, calculate the length of the path using the `D3DXVec3Length` function, and then divide the position you want to use by the resulting value to obtain a scalar value to use in the previous calculations.

For example, to find the coordinates of the point that is 32 units into the path defined previously, you can use the following code:

```
// Pos = position (in 3−D units) of point in path to calculate
// Define starting and ending points of straight path
D3DXVECTOR3 vecStart = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 vecEnd = D3DXVECTOR3(10.0f, 20.0f, 30.0f);

// Get the length of the path
float Length = D3DXVec3Length(&(vecEnd−vecStart));
```

```
// Calculate the scalar by dividing pos by length
float Scalar = Pos / Length;

// Use scalar to calculate coordinates
D3DXVECTOR3 vecPos = (vecEnd − vecStart) * Scalar + vecStart;
```

Now that you can calculate the exact position of any point along the path, you can use this knowledge to move an object along the path. Following the time−based theory of movement, suppose you want to move an object from one point to another over a period of 1,000 milliseconds. The following code (processed once per frame) will accomplish this, continuously looping back from the end to the start of the path in an endless cycle.

```
// vecPoints[2] = path's starting and ending coordinate vectors
// Every frame, use the following code to position an object
// along the straight path based on the current time.
float Scalar = (float)(timeGetTime() % 1001) / 1000.0f;      \
D3DXVECTOR3 vecPos = (vecPoints[1] − vecPoints[0]) *
                      Scalar + vecPoints[0];
// Use vecPos.x, vecPos.y, and vecPos.z coordinates for object
```

Walking Curved Paths

In your game, the paths need not be so straight. You can have your objects move along a nice, curvy path, such as when a character walks around in a circle. Trying to define a smooth circular path using straight lines is nearly impossible, so you must develop a second type of path−one that can handle curves. Not just any type of curve, however. Remember that this is advanced animation−we're going for the big leagues here, and that major hitter you want is a *cubic Bezier curve!* As Figure 2.3 illustrates, a cubic Bezier curve uses four control points (two end points and two midpoints) to define the various aspects of the curve.

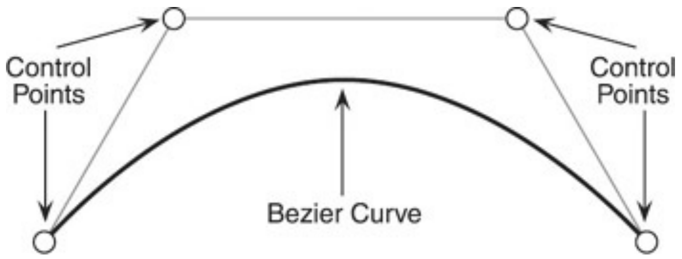


Figure 2.3: A cubic Bezier curve uses four points to determine the direction of the path as it moves from beginning to end.

As you can see, a cubic Bezier curve is not a typical curve−it can bend and twist in a myriad of curved shapes. By manipulating the four control points you can create some really useful paths to use in your projects. The way a cubic Bezier curve works is fairly easy in theory, but a little difficult to implement.

To understand the theory behind a cubic Bezier curve, take a look at Figure 2.4, which shows how the curve is drawn using the four control points.

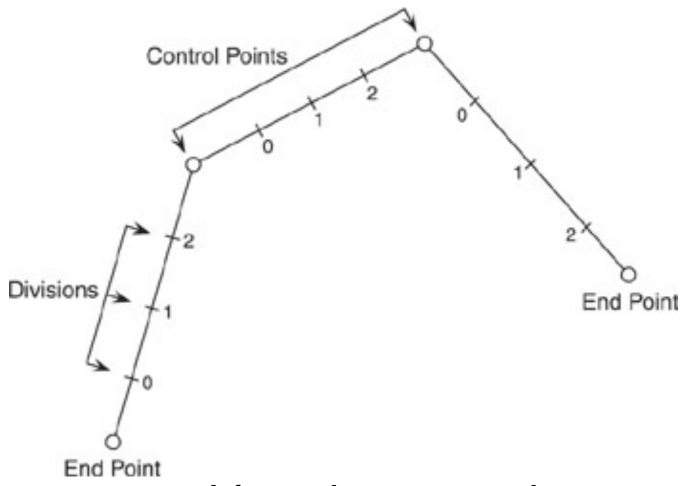


Figure 2.4: You define a cubic Bezier curve by connecting the four points and dividing the lines connecting the points by a set amount. Each division is numbered for later reference.

The purpose of dividing the lines that connect the curve's points is both for visual aid and to serve as the curve's *granularity* (or smoothness). The more additional divisions you add to each line, the smoother the resulting curve will look. To actually see the curve that the points create, you need to connect the divisions on either side of the line, as you can see in Figure 2.5

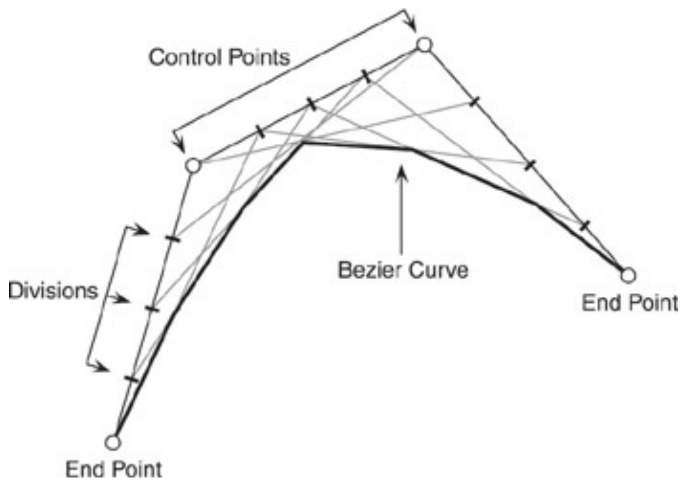


Figure 2.5: You can see a cubic Bezier curve by highlighting the newly connected lines created by joining the numbered divisions.

Although it's cool to draw the curve in the manner I just showed you, it won't make much sense to your computer, nor will it help you figure out the coordinates of a point in the curve. What you need to do is come up with a way to calculate the exact coordinates of any point along the curve. That way, you can do anything you want with the coordinates, from drawing curves to calculating the coordinates where you want to position an object along the curve path! The formula to calculate the coordinates along the curve is

$$C(s) = P_0 * (1 \ \ s)^3 + P_1 * 3 * s * (1 \ \ s)^2 + P_2 * 3 * s^2 * (1 \ \ s) + P_3 * s^3$$

In the formula, the control points are defined as P_0, P_1, P_2, and P_3, which represent the starting point, first midpoint, second midpoint, and ending point, respectively. The resulting coordinates along the curve are defined as C(s), where s is a scalar value (or a time value) ranging from 0 to 1 that determines the position along the curve for which the coordinates should be calculated.

A value of s=0 designates the starting point, whereas a value of s=1 designates the ending point. Any value of s from 0 to 1 designates a point between the two end points. Therefore, to calculate the midpoint of the curve, you would specify s=0.5. The one−quarter position of the curve would be s=0.25, and so on.

To make things easy, you can create a function that takes the four control points (as vector objects) and a scalar value as parameters. The function will return another vector object that contains the coordinates of the point along the curve as specified by the four points and the scalar value. Call the function `CubicBezierCurve`, and use the following prototype to define it.

```
void CubicBezierCurve(D3DXVECTOR3 *vecPoint1, // Start point
                      D3DXVECTOR3 *vecPoint2, // Midpoint 1
                      D3DXVECTOR3 *vecPoint3, // Midpoint 2
                      D3DXVECTOR3 *vecPoint4, // End point
                      float Scalar,
                      D3DXVECTOR3 *vecOut)
{
```

Now get ready for this−you're going to recreate the cubic Bezier curve formula in program code by replacing the appropriate variables with the control point vectors and the scalar value.

```
// C(s) =
*vecOut =                                                   \
  // P0 * (1 − s)3 +
  (*vecPoint1)*(1.0f−Scalar)*(1.0f−Scalar)*(1.0f−Scalar) +  \
  // P1 * 3 * s * (1 − s)2 +
  (*vecPoint2)*3.0f*Scalar*(1.0f−Scalar)*(1.0f−Scalar) +    \
  // P2 * 3 * s2 * (1 − s) +
  (*vecPoint3)*3.0f*Scalar*Scalar*(1.0f−Scalar) +           \
  // P3 * s3
 (*vecPoint4)*Scalar*Scalar*Scalar;
}
```

That's it! Yep, from now on you can calculate the coordinates along a cubic Bezier curve by passing the four control points' coordinates, a scalar, and a returning vector object. For example, going back to the sample curve, you can use the following function call to `CubicBezierCurve` to find the parametric midpoint:

```
D3DXVECTOR3 vecPos;
CubicBezierCurve(&D3DXVECTOR3(−50.0f, 25.0f, 0.0f),      \
                 &D3DXVECTOR3(0.0f, 50.0f, 0.0f),        \
                 &D3DXVECTOR3(50.0f, 0.0f, 0.0f),        \
                 &D3DXVECTOR3(25.0f, −50.0f, 0.0f) ,     \
                 0.5f, &vecPos);
```

Getting back to the point, you can use the return coordinates from the `CubicBezierCurve` function (contained in the vecPos vector object) as the coordinates in which to place an object in the game. By slowly changing the scalar value from 0 to 1 (over a specified amount of time), you move the object from the start of the path to the end. For instance, to travel a curved path over a period of 1,000 milliseconds, you can use the following code:

```
// vecPoints[4] = Starting, midpoint 1, midpoint 2, and end points
// Every frame, use the following code to position an object
// along the curve based on the current time.
D3DXVECTOR3 vecPos;
float Scalar = (float)(timeGetTime() % 1001) / 1000.0f;
CubicBezierCurve(&vecPoints[0], &vecPoints[1], \
                 &vecPoints[2], &vecPoints[3], \
                 Scalar, &vecPos);
// Use vecPos.x, vecPos.y, and vecPos.z coordinates for object
```

That's cool, but having to deal with a scalar value is a little unorthodox when you need to work with actual 3D unit measurements. I mean, how are you supposed to know which scalar value to use when you want to move an object 50 units along the curved path? Isn't there a way to calculate the length of the curve and use that, much like you did with straight lines?

Strangely enough, no. There is no easy way to calculate the length of a Bezier curve. However, you can approximate the length using a few simple calculations. Assuming the four control points of the curve are denoted as `p0, p1, p2,` and `p3`, you can add the lengths between the points `p0` and `p1, p1` and `p2`, and `p2` and `p3`, divide the result in half, and add the length between points `p0` and `p3` (also divided in half). In code, those calculations would look like this:

```
// p[4] = four control points' coordinate vectors
float Length01 = D3DXVec3Length(&(p[1]−p[0]));
float Length12 = D3DXVec3Length(&(p[2]−p[1]));
float Length23 = D3DXVec3Length(&(p[3]−p[2]));
float Length03 = D3DXVec3Length(&(p[3]−p[0]));
float CurveLength = (Length01+Length12+Length23) * 0.5f + \
                    Length03 * 0.5f;
```

The `CurveLength` variable will therefore contain the estimated length of the curve. You'll use the `CurveLength` value much like you did in the straight−path calculations to convert the unit length to a scalar value to calculate the exact coordinates along the curve.

```
// Pos = position in curve (from 0−CurveLength)
float Scalar = Pos / CurveLength;
CubicBezierCurve(&vecPoints[0], &vecPoints[1], \
                 &vecPoints[2], &vecPoints[3], \
                 Scalar, &vecPos);
```

As you can see, cubic Bezier curves aren't too difficult to use. The formulas are pretty basic, and I'd rather leave it up to the math textbooks to go into the details of the calculations (or a fine book like Kelly Dempski's *Focus On Curves and Surfaces*−see Appendix A, "Book and Web References," for details). For now, I'm only interested in making it work for your game project. Speaking of that, let's see what you can do with your newfound knowledge of using straight and curved paths to create routes.

Defining Routes

A path by its lonesome self does you little good; there are times when you need to string together a series of paths that an object must follow. I'm talking about complex paths that are both straight and curved. In fact, we're no longer discussing paths; we've moved on to the advanced topic of routes!

As you can see in Figure 2.6, a route is a series of paths that are commonly connected from endpoint to endpoint.

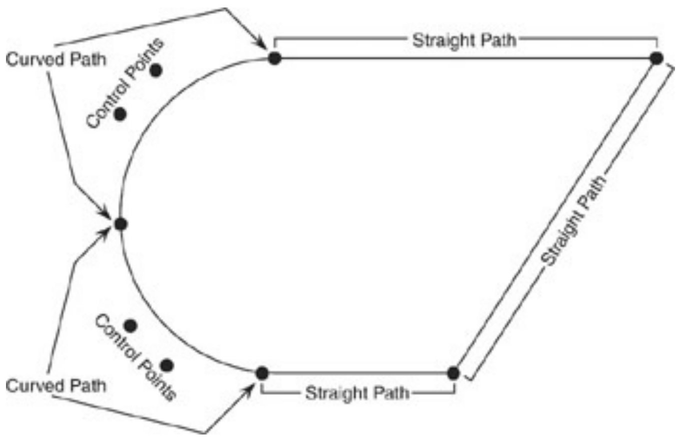


Figure 2.6: You can create a complex route using a series of straight and curved paths. As you can see here, paths do not need to be connected to complete a route.

As you have probably surmised, you can define a route using an array of path objects. By creating a generic path structure, you can store information for both straight and curved paths in one structure. The secret is to look for commonalities and expand on those. For example, the straight and curved paths both have starting and ending points. Therefore, you can define two sets of coordinates that represent the starting and ending coordinates inside your generic path structure, as follows:

```
typedef struct {
  D3DXVECTOR3 vecStart, vecEnd;
} sPath;
```

The only real difference between the two path types is that the curved paths have two additional control points. Adding another two vector objects to your budding `sPath` structure will work just fine for holding the control point coordinates.

```
typedef struct {
  D3DXVECTOR3 vecStart, vecEnd;
  D3DXVECTOR3 vecPoint1, vecPoint2;
} sPath;
```

Now the only thing missing is a flag in the `sPath` structure to determine which type of path is defined−straight or curved. Include a `DWORD` variable and an `enum` declaration to determine which type of path is defined.

```
enum { PATH_STRAIGHT = 0, PATH_CURVED };
typedef struct {
  DWORD       Type;
  D3DXVECTOR3 vecStart, vecEnd;
  D3DXVECTOR3 vecPoint1, vecPoint2;
} sPath;
```

From here on, you only need to store a `PATH_STRAIGHT` value or a `PATH_CURVED` value in the `sPath::Type` variable to determine the use of the contained data−either for a straight path with starting and ending points or for a curved path with starting, ending, and two mid−path control points.

Allocating an array of `sPath` structures is easy, and filling that array with your path's data (such as the data shown in Figure 2.7) is as simple as the following code demonstrates.

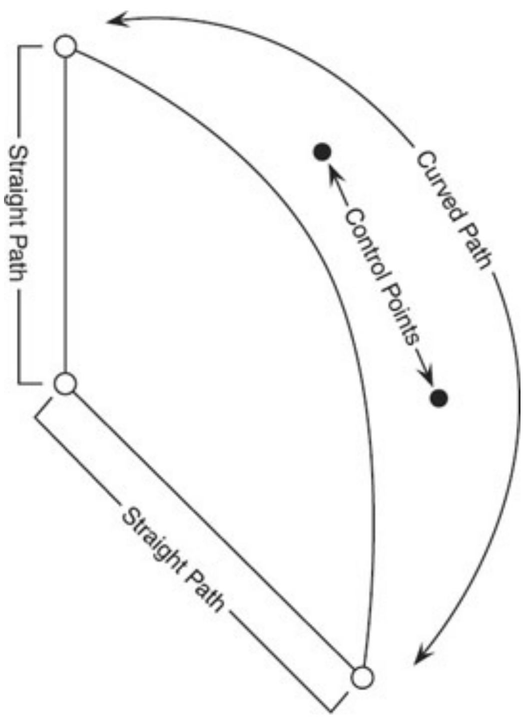


Figure 2.7: A combination of two straight paths and a curved path form a complex route.

```
sPath Path[3] = {
  { PATH_STRAIGHT, D3DXVECTOR3(−50.0f, 0.0f, 0.0f),    \
                   D3DXVECTOR3(−50.0f, 0.0f, 25.0f),   \
                   D3DXVECTOR3(0.0f, 0.0f, 0.0f),      \
                   D3DXVECTOR3(0.0f, 0.0f, 0.0f) },    \
  { PATH_CURVED,   D3DXVECTOR3(−50.0f, 0.0f, 25.0f),   \
                   D3DXVECTOR3(0.0f, 0.0f, 50.0f),     \
                   D3DXVECTOR3(50.0f, 0.0f, 0.0f),     \
                   D3DXVECTOR3(25.0f, 0.0f, −50.0f) }, \
  { PATH_STRAIGHT, D3DXVECTOR3(25.0f, 0.0f, −50.0f),   \
                   D3DXVECTOR3(−50.0f, 0.0f, 0.0f),    \
                   D3DXVECTOR3(0.0f, 0.0f, 0.0f),      \
                   D3DXVECTOR3(0.0f, 0.0f, 0.0f) }     \
 };
```

Of course, you really shouldn't hand−code routes into your project; it's best to use an external source such as an .X file to contain your route data.

Creating an .X Path Parser

The easiest place from which to obtain your path data is−you guessed it−an .X file! That's right, you can construct a couple simple templates to use with a custom .X parser to obtain the paths you want to use for your project. You can even construct routes from your path templates to make things easier!

You can duplicate the generic path structure you developed in the previous section to use as a template in your .X files. This generic path template will contain four vectors−the first two being the starting and ending coordinates of the path (for either a straight or curved path), and the last two being the handles' coordinates (for curved paths). Take a look at the single template definition you can use.

```
// {F8569BED−53B6−4923−AF0B−59A09271D556}
// DEFINE_GUID(Path,
//             0xf8569bed, 0x53b6, 0x4923,
//             0xaf, 0xb, 0x59, 0xa0, 0x92, 0x71, 0xd5, 0x56);
```

```
template Path {
   <F8569BED-53B6-4923-AF0B-59A09271D556>
   DWORD Type;       // 0=straight, 1=curved
   Vector Start;  // Start point
   Vector Point1; // Midpoint 1
   Vector Point2; // Midpoint 2
   Vector End;    // End point
}
```

After you've defined your .X file `Path` template, you can instance as many times as you need in your data files. To load those paths, you should create a route template (called `Route`) that allows you to define multiple path data objects. This route template merely contains an array of `Path` data objects, as you can see here:

```
// {18AA1C92-16AB-47a3-B002-6178F9D2D12F}
// DEFINE_GUID(Route,
//             0x18aa1c92, 0x16ab, 0x47a3,
//             0xb0, 0x2, 0x61, 0x78, 0xf9, 0xd2, 0xd1, 0x2f);
template Route {
    <18AA1C92-16AB-47a3-B002-6178F9D2D12F>
     DWORD NumPaths;
     array Path Paths[NumPaths];
}
```

For an example of using the `Route` template, take a look at how the route defined in the previous section would look in an .X file.

```
Route MyRoute {
   3; // 3 paths
   0; // Straight path type
   -50.0, 0.0, 0.0;
   0.0, 0.0, 0.0;
   0.0, 0.0, 0.0;
   -50.0, 0.0, 25.0;,
1; // Curved path type
   -50.0, 0.0, 25.0;
   0.0, 0.0, 50.0;
   50.0, 0.0, 0.0;
   25.0, 0.0, -50.0;,
   0; // Straight path type
   25.0, 0.0, -50.0;
   0.0, 0.0, 0.0;
   0.0, 0.0, 0.0;
  -50.0, 0.0, 0.0;;
}
```

You can access the route data objects from your .X files by using a custom .X parser. This parser only needs to look for `Route` objects. When it finds one, it will allocate an array of `sPath` structures and read in the data. The route data itself is kept inside a linked list of structures so that you can load multiple routes. This route data uses the following class:

```
class cRoute
{
  public:
      DWORD m_NumPaths;  // # paths in list
      sPath *m_Paths;    // List of paths
      cRoute *m_Next;    // Next route in linked list
  public:
       cRoute() { m_Paths = NULL; m_Next = NULL; }
       ~cRoute() { delete [ ] m_Paths; delete m_Next; }
};
```

The sPath structure also needs to be spruced up a bit. You need to add the length of each path to its respective structure, as well as to the starting position of the path in the series of paths. This is a simple process. The length, as you saw in the previous few sections, is only a floating−point number, and the starting position of the path is the combination of the lengths of all prior paths in the list. Your new sPath structure should look like this:

```
typedef struct {
  DWORD       Type;
  D3DXVECTOR3 vecStart, vecEnd;
  D3DXVECTOR3 vecPoint1, vecPoint2;
  float Start;   // Starting position
  float Length;  // Length of path
} sPath;
```

The reason to include the length and starting position of the path in the sPath structure is really a matter of speed. By pre−computing the length values loading the path data, you can quickly access that data (the length and starting position) when you are determining the path in which an object is located based on its distance into the route.

I know it sounds strange, but think of it like this−the starting positions and lengths of each path are like key frames; instead of measuring time, you are measuring the lengths of the paths. By taking the position of an object in the route (say 516 units), you can scan through the list of paths and see the path within which the object lies.

Suppose the route uses six paths, and the fourth path starts at 400 units. The fourth path is 128 units long, meaning that it covers the lengths from 400 to 528 units. The object at 516 units is located in the fourth path; by subtracting the object's position (516) from the ending position of the path (528), you can discover the offset in the path that you can use as a scalar value to calculate the object's coordinates along the path. In this case, that position would be 528−516, or 12 units, and the scalar value would be 12/128, or 0.09375.

Enough talk, let's get to some code! The following class, cXRouteParser, is derived from the cXParser class, meaning that you have access to the data−object parsing code. All you want to do with the cXRouteParser class is scan for Route data objects and load the appropriate path data into a newly allocated Route class that is linked to a list of routes.

Check out the cXRouteParser declaration, which contains the root Route class pointer and six functions (three of which contain code inline to the class).

```
class cXRouteParser : public cXParser
{
  protected:
       BOOL ParseTemplate(IDirectXFileData *pDataObj,          \
                          IDirectXFileData *pParentDataObj,    \
                          DWORD Depth,                         \
                          void **Data, BOOL Reference);

     public:
        cRoute *m_Route;

     public:
        cXRouteParser() { m_Route = NULL; }
        ~ cXRouteParser () { Free(); }
        void Free() { delete m_Route; m_Route = NULL; }
        void Load(char *Filename);
        void Locate(DWORD Distance, D3DXVECTOR3 *vecPos);
 };
```

The most important function of `cXRouteParser` is the template data object parser, of course. I'll show you the parser function in a moment. The `Load` function merely sets up the call to `Parse`, which in turn loads all your `Route` templates into the linked list. The `Locate` function calculates the position along the route's path that you can use to position an object. Again, I'll get to the `Locate` function in a moment. For now, I want to get back to the `ParseTemplate` function.

The `ParseTemplate` function only scans for one template data object−`Route`. Once it is found, a `cRoute` class and the path structures are allocated, and the data is loaded. The `cRoute` class is linked in the list of loaded routes, and the parsing of the .X file continues. Here's what the `ParseTemplate` function code looks like:

```
BOOL cXRouteParser::ParseTemplate(IDirectXFileData *pDataObj, \
                       IDirectXFileData *pParentDataObj,      \
                       DWORD Depth,                           \
                       void **Data, BOOL Reference)
{
    const GUID *Type = GetTemplateGUID(pDataObj);

    // Only process Route data objects
    if(*Type == Route) {
       // Get pointer to data
       DWORD *DataPtr = (DWORD*)GetTemplateData(pDataObj, NULL);

       // Allocate and link in a route
       cRoute *Route = new cRoute();
       Route−>m_Next = m_Route;
       m_Route = Route;

       // Get # paths in route and allocate list
       Route−>m_NumPaths = *DataPtr++;
       Route−>m_Paths = new sPath[Route−>m_NumPaths];

       // Get path data
       for(DWORD i=0;i<Route−>m_NumPaths;i++) {

       // Get path type
       Route−>m_Paths[i].Type = *DataPtr++;

       // Get vectors
       D3DXVECTOR3 *vecPtr = (D3DXVECTOR3*)DataPtr;
       DataPtr+=12; // skip ptr ahead
       Route−>m_Paths[i].vecStart   = *vecPtr++;
       Route−>m_Paths[i].vecPoint1 = *vecPtr++;
       Route−>m_Paths[i].vecPoint2 = *vecPtr++;
       Route−>m_Paths[i].vecEnd     = *vecPtr++;

       // Calculate path length based on type
       if(Route−>m_Paths[i].Type == PATH_STRAIGHT) {
         Route−>m_Paths[i].Length = D3DXVec3Length(          \
                        &(Route−>m_Paths[i].vecEnd −         \
                          Route−>m_Paths[i].vecStart));
       } else {
         float Length01 = D3DXVec3Length(                    \
                  &(Route->m_Paths[i].vecPoint1 -            \
                    Route->m_Paths[i].vecStart));
         float Length12 = D3DXVec3Length(                    \
                  &(Route->m_Paths[i].vecPoint2 -            \
                    Route->m_Paths[i].vecPoint1));
         float Length23 = D3DXVec3Length(                    \
                  &(Route->m_Paths[i].vecEnd -               \
                    Route->m_Paths[i].vecPoint2));
         float Length03 = D3DXVec3Length(                     \
                  &(Route->m_Paths[i].vecEnd -                \
                    Route->m_Paths[i].vecStart));
         Route->m_Paths[i].Length = (Length01+Length12+       \
                                 Length23)*0.5f+Length03*0.5f;
       }
       // Store starting position of path
       if(i)
         Route->m_Paths[i].Start = Route->m_Paths[i-1].Start + \
                                   Route->m_Paths[i-1].Length;
       else
         Route->m_Paths[i].Start = 0.0f;
     }
   }

   // Parse child templates
   return ParseChildTemplates(pDataObj, Depth, Data, Reference);
}
```

As usual, you don't call `ParseTemplate` directly−it's up to your `cXRouteParser::Load` function to call `Parse`, which in turn calls `ParseTemplate`. Knowing this, take a look at the `Load` function (which takes the file name of the .X file to parse as the only parameter).

```
void cXRouteParser::Load(char *Filename)
{
  Free(); // Free loaded routes
  Parse(Filename);
}
```

Short and sweet, just how I like them! The `Load` function is really just a gateway to ensure that prior route data is freed and the `Parse` function is called. There's not much more to it!

Now that you've defined the templates, created your custom class, and loaded the route data, it's time to start moving those objects! It's time to get back to the `cXRouteParser::Locate` function. You might have noticed that the function prototype for `Locate` only specifies a floating−point value and a vector object. I want to keep this simple, so I'm only going to scan the first route in the linked list to position an object.

Tip To scan multiple routes inside the `Locate` function, you might want to retain each `Route` object's instance name, which you'll use in the function call to `Locate` to match which route to scan.

By taking the current time and the distance (in 3D units) that an object can move, you can iterate through each path in your route to determine exactly which path an object would be on at a specific time. From there, you can calculate exactly how far between the beginning and end of that path the object lies and correctly position your object.

Suppose you have an object that is moving at 200 units per second, which is 0.2 units per millisecond. Multiply the distance per second by the total time along the path to obtain the object's location within the route. You'll use this total distance inside the route as the `Distance` parameter in the call to `Locate`.

The `Locate` function takes the distance you provided and scans through each path contained in the route. Remember, you've already calculated the starting position and length for each path, so this is merely a check to see whether the object's distance is greater than the start of the path and less than the distance of the path. Take a look at `Locate`'s code to see what I mean.

```
void cXRouteParser::Locate(float Distance, D3DXVECTOR3 *vecPos)
{
  // Scan through first route class in list
  cRoute *Route = m_Route;
  if(!Route)
    return;

  // Scan through each path in route
  for(DWORD i=0;i<Route−>m_NumPaths;i++) {
  // See if distance falls into current path
  if(Distance >= Route−>m_Paths[i].Start &&               \
     Distance < Route−>m_Paths[i].Start +                 \
                Route−>m_Paths[i].Length) {
     // Distance is within current path, use that
     // Get offset into path using start
     Distance −= Route−>m_Paths[i].Start;

     // Calculate the scalar value to use
     float Scalar = (float)Distance/Route−>m_Paths[i].Length;

     // Calculate coordinate based on path type
     if(Route−>m_Paths[i].Type == PATH_STRAIGHT) {
       *vecPos = (Route−>m_Paths[i].vecEnd −              \
                  Route−>m_Paths[i].vecStart) *           \
                  Scalar + Route−>m_Paths[i].vecStart;
     } else {
       CubicBezierCurve(&Route−>m_Paths[i].vecStart,       \
                        &Route−>m_Paths[i].vecPoint1,     \
                        &Route−>m_Paths[i].vecPoint2,     \
                        &Route−>m_Paths[i].vecEnd,         \
                         Scalar, vecPos);
     }
   }
 }
}
```

There you have it−a perfect method of using routes in your own game projects! Check out the Route demo for this chapter to see routes in action. (See the end of this chapter for more information.) The Route demo slowly moves an object along a path at a set speed, which is based on time.

Although routes are cool for moving your game's characters and other assorted objects, how about a use for routes that will blow your socks off? One of the greatest uses of routes is to control in−game cameras to create cinematic sequences.

Creating In−Game Cinematic Sequences

Using time−based animation is crucial to achieving smooth playback, but what good could using time−based movement possibly do? Sure, moving a few objects around a set path is neat, but is that all you can do? The answer is a resounding no! There's much more you can do with time−based movement, including creating in−game cinematic sequences, like those from games such as Silicon Knights' *Eternal Darkness: Sanity's Requiem* In *Eternal Darkness*, the player is treated to animation sequences that play out using the game's 3D engine.

Typically, the game's camera shifts focus from the player's point of view to capture the scene from another view that shows the player and some other character confronting each other and advancing the story line.

To use a cinematic camera, you can rely on the techniques you read about earlier in this chapter, and you can use the pre−calculated animation sequences you will see in Chapter 3 As Figure 2.8 illustrates, it's only a matter of plotting out the path that your camera will follow over time. Mix that with a complete pre−calculated animation, and you've got yourself a complete in−game cinematic engine!

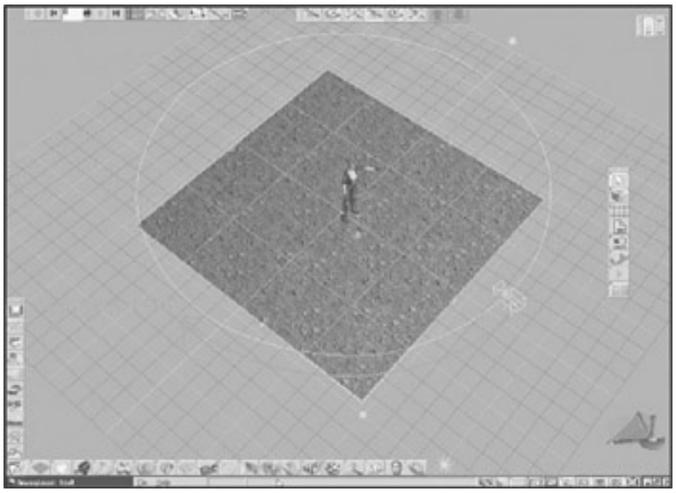


Figure 2.8: Just like a movie, the camera follows a predetermined path, showing the pre−calculated animation from different angles. The angles are determined by setting a path for both the camera and the camera's target location (where the camera is looking).

Rather than reiterate what you already saw in this chapter, I'll leave it up to you to check out the Cinematic demo, which shows a small cinematic sequence. In a nutshell, the demo merely loads a series of keys (using the .X path parser class) that represent the paths the camera follows. In every frame, the position of the camera is calculated using the keys, and the viewport is oriented. Then the pre−calculated animation is updated and the entire scene is rendered.

Check Out the Demos

This chapter introduced you to animating based on time using key−frame data structures, as well as moving along paths and routes over time. On the CD−ROM, you'll find four demo programs that illustrate what you've read in this chapter. For exact information on these demos' locations, check out the end of this chapter. The next few sections show you what each demo does.

TimedAnim

The TimedAnim demo, as illustrated in Figure 2.9, demonstrates how key−framed animation structures can be used to animate objects (such as the nifty robot) over time. This demo runs continuously until you close the program.

Figure 2.9: Key−framed animation in action! The robot rotates and moves according to the transformation key frames set in the demo source.

TimedMovement

Timed movement is just as important as timed animation, and the TimedMovement demo shows you how to pull it off. Figure 2.10 shows the TimedMovement demo in action. This demo runs continuously until you close the program.

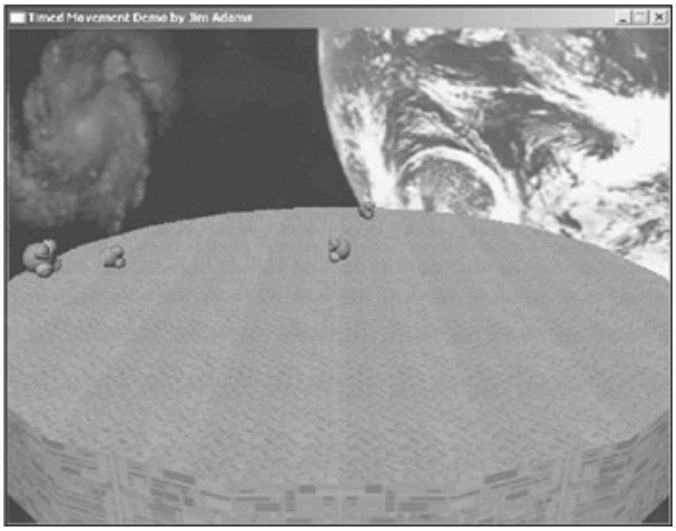


Figure 2.10: The TimedMovement demo shows you how to move a series of robots up and down straight and curved paths over time.

The TimedMovement demo (as well as the following Route and Cinematic demos) demonstrates how to point objects in the direction in which they are traveling by determining the vector in which an object moved since its last update. With that movement vector, you can calculate an angle to orient your meshes to point in the proper direction while moving.

Route

The Route demo (seen in Figure 2.11) shows how to string together a series of straight and curved paths to create complex routes along which you can move your objects. This program continues until you exit the application.

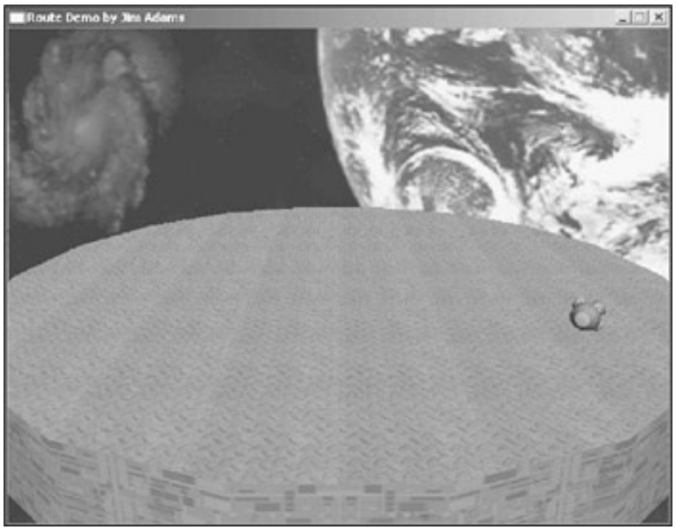


Figure 2.11: Take command of your robot by laying down complex routes in which to travel around your worlds. Here, the robot demonstrates the use of straight and curved paths.

Cinematic

Rounding out the list of demos for this chapter is Cinematic. As shown in Figure 2.12, you get to see how complex routes can be applied to cameras in order to traverse a 3D scene in real time. This technique of moving a camera is perfect to use for an in−game cinematic system.

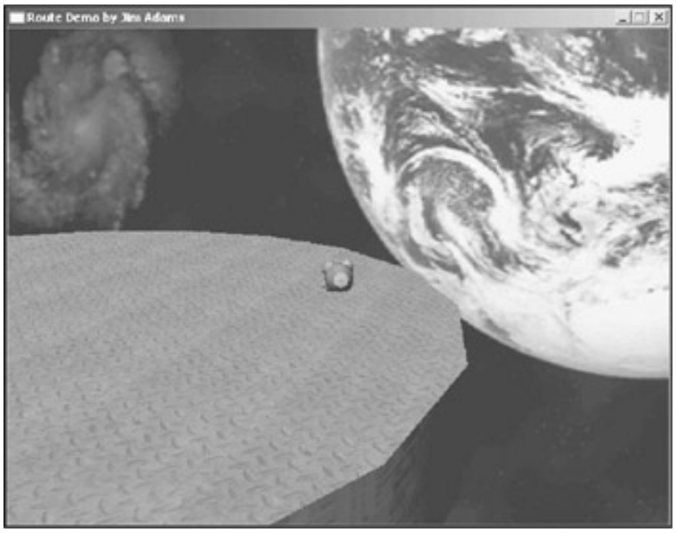


Figure 2.12: The cinematic camera demo adds a moving camera to the Route demo.

Programs on the CD

The CD−ROM contains four demo programs that illustrate the animation techniques you learned in this chapter. These four programs, located in the Chapter 2 directory of the enclosed disc, are

- **TimedAnim.** This project shows you how to use time−based animation with key frames, as shown in this chapter. It is located at \BookCode\Chap02\TimedAnim.
- **TimedMovement.** This demo shows you how straight and curved paths are used to move objects around your 3D world. It is located at \BookCode\Chap02\TimedMovement.
- **Route.** This demo shows the route parser class in action by slowly moving an object around a route. It is located at \BookCode\Chap02\Route.

- **Cinematic.** This is an in−game cinematic sequence, complete with a route−following camera and pre−calculated animations. It is located at \BookCode\Chap02\Cinematic.

Chapter 3: Using the .X File Format

Your 3D meshes need a place to liverather, you need a place to store your 3D mesh data (not to mention all that other data your game project requires). What's a developer to do−develop his own file format or go with a third−party format? With so many popular formats out there, it's an easy choice to make, but what about the restrictions some formats impose? Why can't you just use somebody else's file format and configure it to work the way you want?

That somebody else is none other than Microsoft, and the format to use is .X! Now uncross those eyes, mister−those .X files are really easy to use once you understand them, and this chapter will teach you what you need to know.

Working with .X Templates and Data Objects

If you haven't already, I invite you to take a look at one of those mysterious .X files that comes packaged with the DirectX SDK (located in the \Samples\Multimedia\Media directory of your DirectX install). Go on, I dare you. More than likely, you'll be greeted with something like this:

```
xof 0302txt 0032

template Header {
  <3D82AB43−62DA−11cf−AB39−0020AF71E433>
  DWORD major;
  DWORD minor;
  DWORD flags;
}

template Frame {
  <3D82AB46−62DA−11cf−AB39−0020AF71E433>
  [FrameTransformMatrix]
  [Mesh]
}

Header {
  1;
  0;
  1;
}

Frame Scene_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;;
  }
  Frame Pyramid_Frame {
    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;;
    }
    Mesh PyramidMesh {
      5;
      0.00000;10.00000;0.00000;,
      −10.00000;0.00000;10.00000;,
      10.00000;0.00000;10.00000;,
      −10.00000;0.00000;−10.00000;,
      10.00000;0.00000;−10.00000;;
      6;
      3;0,1,2;,
      3;0,2,3;,
      3;0,3,4;,
      3;0,4,1;,
      3;2,1,4;,
      3;2,4,3;;
      MeshMaterialList {
        1;
        6;
        0,0,0,0,0,0;;
        Material Material0 {
          1.000000; 1.000000; 1.000000; 1.000000;;
          0.000000;
          0.050000; 0.050000; 0.050000;;
          0.000000; 0.000000; 0.000000;;
          }
      }
    }
  }
}
```

Scary looking, isn't it? Actually, you can break down every .X file into a small handful of easy−to−manage components, which makes the files easy to understand and process. Let me explain what I mean. Every .X file starts with a small header, which in the preceding example looks like this:

```
xof 0302txt 0032
```

This small blurb of text informs programs that load the file that it is indeed an .X file. (The `xof` portion signifies an .X file.) It also informs programs that the file uses the DirectX .X file version 3.2 templates (represented by the `0302` text). Following the version number is `txt`, which signifies that all of the following .X data is stored in a text format as opposed to a binary format. The line of text ends with `0032`, which defines the number of bits reserved for floating−point values (`0032` for 32−bit or `0064` for 64−bit).

Note Binary, a second .X file storage format, is useful for compacting data into a format that is unreadable by humans. I won't discuss the binary format in this book; however, the techniques used to process .X files in this chapter still apply to binary .X files, so don't worry about missing out on any good stuff!

After the file header there are a slew of data chunks, referred to as *templates* and *data objects*. You can tell the difference between a template and a data object because all templates begin with the word `template`. As you can see from the .X file code, templates look much like a C structure definition. Data objects are instances of those templates.

You use templates to define the information that data objects contain in the .X file. (A template defines the layout of a data object.) Each template can contain any type of data defined by a small set of data types, and any combination of data types can be used inside a template. A data object is merely an instance of a template. You can think of a template much like a C++ class−they both define the data that an instance of the object can contain.

Taking another look at the example .X file, you can see that the first template you'll encounter is `Header`, which is the template's *class name*. The Header template contains three `DWORD` values (as well as a large number called a GUID, which is enclosed in angle brackets), which you set when you create a data object from the template. Creating data objects is much like instancing a class or structure. In the previous .X file code, the instancing of the Header template looks like this:

```
Header {
  1; // major
  0; // minor
  1; // flags
}
```

Notice that you must define every variable contained in the `Header` template in your data object, and in the same order. You might be wondering about that large number (the template's GUID) defined in the template, however. What does that have to do with instancing your template? Nothing, actually, because DirectX uses that large number to identify templates as they are loaded. I'll get back to the template GUID (*Globally Unique Identification Number*) in a moment.

Tip Much like C/C++, you can also use the handy // operator to signify comments in your .X file.

The next template you'll see in the .X file is `Frame`. This is a special template−it doesn't define any data types, but it does reference other template classes. The other template classes, enclosed in square brackets, are named `FrameTransformMatrix` and `Mesh`. Using this manner of referencing other templates from within a template, you can create a hierarchy of data objects.

Also, by declaring additional templates within another template, you are creating a set of *template restrictions*, which enable you to create templates that only allow specific data objects to be embedded within another data object. In this case, only the data objects of the type `FrameTransformMatrix` and `Mesh` can be embedded in a `Frame` data object. You'll read more about template restrictions later in this chapter. For now, move on to examining the rest of the .X file.

Following the template definitions (which should also be at the beginning of the .X file) are the data objects. These are declared much like C data structures would be−you instance the structure by its template class name, followed by the data object's *instance name*. The instance name is optional, however, so don't worry if you come across some data objects that are missing it.

In the .X file you're examining, the first data object has an instance name of `Scene_Root`. The `Scene_Root` object is of the template class type `Frame`. You've already seen the `Frame` template defined. Looking back to that template definition, you can see that there is no data to store, but there are two optional data objects you can embed in `Frame`−`FrameTransformMatrix` and `Mesh`.

Just by a matter of luck, both a `FrameTransformMatrix` and a `Mesh` data object are embedded in `Scene_Root`. Missing from the .X file, however, are the template definitions for `FrameTransformMatrix` and `Mesh`. How are you supposed to know what data those objects contain? Well, an .X file doesn't have to define every template with the file itself−you can define those template definitions inside your program!

You'll get to see how to define these templates within your programs later in this chapter. For now, let's get back to the example. A data object of the template class type `FrameTransformMatrix` is embedded in the `Scene_Root` data object. This data object contains floating−point values that represent a transformation matrix. After that data object there is another data object of the template class type `Mesh`, which contains information about a mesh.

Okay, enough of this example−I'm sure you're getting the gist of it. As you can see, templates are completely user−defined, meaning that you can create any type of template to contain any type of data. Want to contain raw sound data in an .X file? How about storing heartbeat−sensor readings? Using .X, you can store sound data, heartbeat readings, and any other type of data you want!

Defining Templates

Since an .X file's open−ended design is so, well, *open−ended*, you must predefine each template that you intend to use for DirectX to understand how to access the template's data. Typically templates are defined inside an .X file, although you can define them from within your program (as I mentioned earlier).

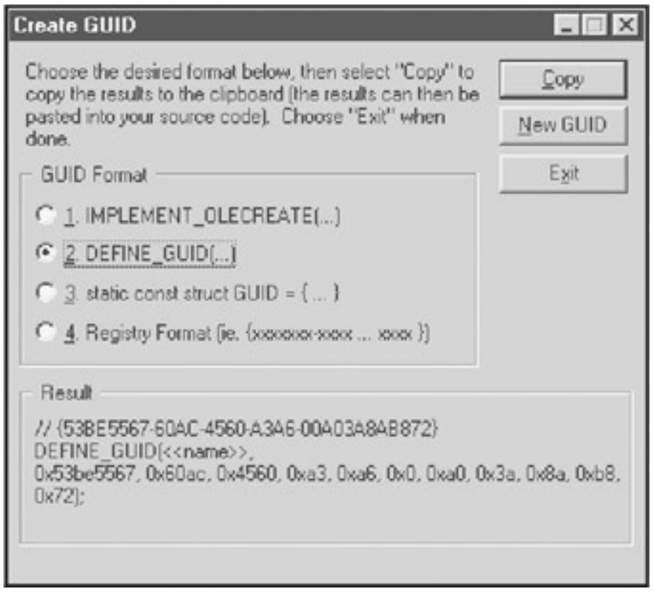
You define a template (contained in an .X file) by assigning it a unique class name preceded by the word `template`, as I have done in the following line of text. (Notice the opening bracket, which signifies the start of the template's definition.)

```
template ContactEntry {
```

Cool−now you've started the declaration of a template that you will use to store a person's contact information. We're calling the template class `ContactEntry`, as you can see from the code. Even though you have assigned your template a unique class name, you need to go one step further and also assign it a unique identification number−a GUID.

When you get around to reading an .X file into your program, you'll only have access to the GUIDs of each template, not the class names. The class names are important only to your .X file data objects; you want your program to differentiate those data objects by their template GUIDs.

To define a GUID for your template, fire up the guidgen.exe program that comes with your Microsoft Visual C/C++ compiler installation (located in the \Common\Tools directory of your MSVC installation). After you've found and executed the guidgen.exe file, you'll be presented with a small dialog box, shown in Figure 3.1.



Choose the desired format below, then select "Copy" to copy the results to the clipboard (the results can then be pasted into your source code). Choose "Exit" when done.

- 1. IMPLEMENT_OLECREATE(...)
- [✓ 2. DEFINE_GUID(...)]
- 3. static const struct GUID = { ... }
- 4. Registry Format (ie. {xxxxxxxx-xxxx ... xxxx })

```
// {53BE5567-60AC-4560-A3A6-00A03A8AB872}
DEFINE_GUID(<<name>>,
0x53be5567, 0x60ac, 0x4560, 0xa3, 0xa6, 0x0, 0xa0, 0x3a, 0x8a, 0xb8,
0x72);
```

Figure 3.1: The guidgen.exe's Create GUID dialog box allows you to create a unique identification number in

various formats.
As you can see in Figure 3.1, the Create GUID dialog box allows you to choose the format of the GUID you want to create. In this case you'll use format #2, `DEFINE_GUID().` Select the option and click the Copy button.

Now a completely unique identification number is on the Clipboard, waiting for you to paste it into your code. Go back to the .X file you are creating and paste the contents of the Clipboard into your template declaration.

```
template ContactEntry {
// {4C9D055B-C64D-4bfe-A7D9-981F507E45FF}
DEFINE_GUID(<<name>>,
0x4c9d055b, 0xc64d, 0x4bfe, 0xa7, 0xd9, 0x98,         \
0x1f, 0x50, 0x7e, 0x45, 0xff);
```

Whoops! That's a little too much text for the template, so you need to cut out the `DEFINE_GUID` macro stuff and paste that into your project's source code. Yes, that's right−every template you define requires a matching GUID definition (via the `DEFINE_GUID` macro, for example) inside your code. This means you need to include the `initguid.h` file in your code and use `DEFINE_GUID`, as I have done here.

```
#include "initguid.h"
// At beginning of source code file - add DEFINE_GUIDs
DEFINE_GUID(ContactEntry,                               \
     0x4c9d055b, 0xc64d, 0x4bfe, 0xa7, 0xd9, 0x98,      \
     0x1f, 0x50, 0x7e, 0x45, 0xff);
```

Notice that in the `DEFINE_GUID` macro, I've replaced the `<<name>>` text with the actual class name of the template I am defining. In this case, I am using `ContactEntry` as a macro name. From this point on, the `ContactName` macro will contain a pointer to my template's GUID (which must match the template's GUID in the .X file).

Getting back to the `ContactEntry` template, you also need to remove the comment tag from the pasted text and change the GUID's brackets to angle brackets, as I have done here:

```
template ContactEntry {
  <4C9D055B-C64D-4bfe-A7D9-981F507E45FF>
```

Now you're ready to move on and define the template's data. Templates are much like C structures and classes; they contain variables and pointers to other templates, as well as access restrictions. The types of variables you can use are much like the ones you use in C. Table 3.1 shows you the data types at your disposal for defining templates, as well as matching C/C++ data types.

Table 3.1: .X Template Data Types

Data Type	Description
`WORD`	16−bit value (`short`)
`DWORD`	32−bit value (32−bit `int` or `long`)
`FLOAT`	IEEE float value (`float`)
`DOUBLE`	64−bit floating−point value (`double`)

`CHAR`	8−bit signed value (`signed char`)
`UCHAR`	8−bit unsigned value (`unsigned char`)
`BYTE`	8−bit unsigned value (`unsigned char`)
`STRING`	A NULL−terminated string (`char[]`)
`array`	Signifies an array of following data type to follow (`[]`)

Much like C/C++ variable declarations, you follow the data type keyword with an instance name and finish with a semicolon (signifying the end of the variable declaration).

```
DWORD Value;
```

In Table 3.1, you'll notice the `array` keyword, which defines an array of data types. To define an array, you specify the `array` keyword followed by the data type, instance name, and array size (enclosed in square brackets). For example, to declare an array of 20 `STRING` data types, you could use

```
array STRING Text[20];
```

Note The cool thing about arrays is that you can use another data type to define the array size, as I have done here:

```
DWORD ArraySize
; array STRING Names[ArraySize];
```

Now you need to go back to the `ContactEntry` template and define a person's name, phone number, and age. The three variables−two strings (name and phone number) and one numerical value (age)−can be defined in the `ContactEntry` template as follows.

```
template ContactEntry {
  <4C9D055B−C64D−4bfe−A7D9−981F507E45FF>
  STRING Name;         // The contact's name
  STRING PhoneNumber;  // The contact's phone number
  DWORD Age;           // The contact's age
}
```

Cool! You finish your template definition with a closing bracket, and you're ready to go.

Creating Data Objects from Templates

After you have defined a template, you can begin creating data objects and defining their data. Data objects are defined by their respective template class types and an optional instance name. You can use this instance name to later reference the data object inside the .X file or from within your project (a feature you'll read about later in this chapter).

Moving on with the example, take the `ContactEntry` template and create a data object from it. This data object will contain a person's name, phone number, and age.

```
ContactEntry JimsEntry {
  "Jim Adams";
  "(800) 555−1212";
  30;
}
```

Notice that I've declared the data object's instance name as `JimsEntry`. From now on, I can reference this data object by using the name enclosed in brackets, like this:

```
{JimsEntry}
```

Referencing a data object in this manner is called *data referencing*, or *referencing* (as if you couldn't guess!), and it allows you to point one data object to another. For example, an animation sequence template (`AnimationSet`) requires you to reference a `Frame` data object for the sequence's embedded objects.

You can also use referencing to duplicate an object's data without having to retype it. This is useful when you are creating a few identical `Mesh` data objects in an .X file, with each `Mesh` object being oriented differently inside various `Frame` objects.

Embedding Data Objects and Template Restrictions

Data referencing has one caveat−the template restrictions set in place must allow you to use a reference. That might not make sense at first, but you can't use a data reference without the proper restrictions. An .X file represents an entire hierarchy of data objects, which can only be siblings or children of other objects. Thus, data objects embedded in other objects need the proper restrictions to be referenced or instanced.

For example, consider the following three template declarations:

```
template ClosedTemplate {
  <4C9D055B−C64D−4bfe−A7D9−981F507E45FF>
  DWORD ClosedData;
}
template OpenTemplate {
  <4C9D055B−C64D−4bff−A7D9−981F507E45FF>
  DWORD OpenData;
  [...]
}
template RestrictedTemplate {
  <4C9D055B−C64D−4c00−A7D9−981F507E45FF>
  DWORD RestrictedData;
  [ClosedTemplate]
  [OpenTemplate]
}
```

They are pretty standard template declarations, except for the lines that contain square brackets. The information inside those square brackets is important. The first template, `ClosedTemplate`, doesn't have square brackets, so it is considered a closed template. You can only instance and define the `ClosedData` value inside `ClosedTemplate`.

The `OpenTemplate`, however, contains the [] line, which signifies that it is an open template. An open template allows any data object to be embedded in place of the [] line. For example, you can instance `OpenTemplate`, define the `OpenData` variable, and then embed an instance of `ClosedTemplate` within the `OpenTemplate`.

`RestrictedTemplate` has two lines of bracket text. Restricting templates only allow data objects of those template types listed; in this case, those templates are `ClosedTemplate` and `OpenTemplate`. Attempts to embed any other data object other than the two listed will fail (causing the parse to fail).

Whew−you might have to reread this section a few times to fully understand the ability to embed and restrict templates within other templates. Once you have a firm grasp on embedding and restricting, it's time to move on and learn about DirectX's pre−defined standard templates, which are packaged with the DirectX SDK.

Working with the DirectX Standard Templates

Now that you've worked your way through templates and data objects, you can move up a step and see what you can do with them in your project. If you've taken the time to play around with the DirectX SDK, you'll notice that .X is widely used for containing mesh information. To that end, Microsoft has packaged DirectX with a number of templates, which I call the DirectX *standard templates*. These templates are used to contain all mesh−related data.

The standard templates are useful because they define almost every aspect of 3D meshes, so take a moment to study them here. I won't go into an incredible amount of detail regarding the standard templates in general because the DirectX SDK contains a plethora of information for them, but I'll give you the lowdown for each template.

The standard templates, shown in Table 3.2, each have a matching GUID macro that you use to determine which data object is which in your program. These macros are defined (using `DEFINE_GUID`) inside a special file named `rmxfguid.h`. The standard templates' GUID macros are easy to remember because you just prefix the template's name with `D3DRM_TID`. For instance, the `Animation` template is defined by the macro `D3DRM_TIDAnimation`.

Table 3.2: DirectX .X Standard Templates

Template Name	Description
`Animation`	Defines animation data for a single frame.
`AnimationKey`	Defines a single key frame for the parent animation template.
`AnimationOptions`	Contains animation playback information.
`AnimationSet`	Contains a collection of animation templates.
`Boolean`	Holds a Boolean value.
`Boolean2d`	Holds two Boolean values.
`ColorRGB`	Contains red, green, and blue color values.
`ColorRGBA`	Contains red, green, blue, and alpha color values.
`Coords2d`	Defines two coordinate values.
`FloatKeys`	Contains an array of floating−point values.
`FrameTransformMatrix`	Holds the transformation matrix for a parent `Frame` template.
`Frame`	A frame−of−reference template that defines a hierarchy.
`Header`	The .X file header that contains version numbers.
`IndexedColor`	Contains an indexed color value.
`Material`	Contains material color values.
`Matrix4x4`	Holds a 4x4 homogenous matrix container.
`Mesh`	Contains a single mesh's data.
`MeshFace`	Holds a mesh's face data.
`MeshFaceWraps`	Contains the texture wrapping for mesh faces.
`MeshMaterialList`	Contains the material for face−mapping values.

`MeshNormals`	Holds normals used for mesh data.
`MeshTextureCoords`	Holds texture coordinates used for mesh data.
`MeshVertexColors`	Holds vertex color information used for mesh vertices.
`Patch`	Defines a control patch.
`PatchMesh`	Contains a patch mesh (much like the `Mesh` template).
`Quaternion`	Holds a quaternion value.
`SkinWeights`	Contains an array of weight values mapped to a mesh's vertices. Used in skinned meshes.
`TextureFilename`	Contains the texture file name to use for a material.
`TimedFloatKeys`	Contains an array of `FloatKeys` templates.
`Vector`	Holds a 3D coordinate value.
`VertexDuplicationIndices`	Informs you which vertices are duplicates of other vertices.
`XSkinMeshHeader`	Used by skinned meshes to define the number of bones contained in a mesh.

You can see that there are quite a few templates−too many to discuss in this book. Thank−fully, however, you'll find that you only need to deal with a handful of the standard templates while parsing .X files. You'll pick up on which standard templates I'm speaking of as you go through the rest of this book. For now, let's get a move on so you can see how to access .X files in your own projects.

Accessing .X Files

Regardless of the version of DirectX you are using (either DirectX 8 or 9), the methods you use to access .X files are the same. In fact, the interfaces have not changed names between the two newest versions of DirectX (8 and 9), making it possible for you to quickly port your version 8 code to the newer version 9 (and vice versa if you want).

The first step to accessing any .X file is to create an `IDirectXFile` interface. You need to call the `DirectXFileCreate` function, as shown in the following bit of code:

```
IDirectXFile *pDXFile = NULL;
HRESULT Result = DirectXFileCreate(&pDXFile);
```

As you can see from the previous lines of code, the `DirectXFileCreate` function takes one parameter−the pointer to an `IDirectXFile` interface. You can quickly determine whether the function has succeeded in creating a valid, `IDirectXFile` interface by using the `SUCCEEDED` or `FAILED` macro on the return code from the `DirectXFileCreate` call.

Once you've successfully created the `IDirectXFile` interface, you can optionally register any templates you'll be using (such as the DirectX standard templates) and create an enumeration interface that weeds through the top−level data objects within your .X files.

Registering Custom and Standard Templates

To save storage space and improve your data security, the .X interfaces allow you to remove all template definitions from .X files and embed them into your executable. This means that instead of the .X files defining templates, your program has to do it. Don't worry−it's not as difficult as it sounds. As you'll see in a moment, Microsoft has taken the liberty of doing the hard work by defining the standard templates inside a couple include files, making everything as simple as possible.

To register the standard templates (or any template, for that matter) from within your program, you'll need to call upon the `IDirectXFile::RegisterTemplates` function.

```
HRESULT IDirectXFile::RegisterTemplates(
  LPVOID pvData, // buffer containing template definitions
  DWORD cbSize); // # of bytes of data
```

The `pvData` parameter is merely a data buffer that contains the template definitions in the exact format you'd see in an .X file. For example, you can define a template data buffer like this:

```
char *Templates = "
      "xof 0303txt 0032 \
      template CustomTemplate { \
        <4c944580−9e9a−11cf−ab43−0120af71e433> \
        DWORD Length; \
        array DWORD Values[Length]; \
   }";
```

Note Notice that the `template` definition in `Templates` uses the backslash character to represent a new line, and that the first line of text is a standard .X file header.

Going back to `RegisterTemplates`, the `cbSize` parameter represents the size of the template data buffer, which you can determine in this case by using the `strlen` of the `Templates` buffer. Put together, you can register the templates defined in `Templates` using the following code:

```
pFile->RegisterTemplates(Templates, strlen(Templates));
```

Now let's get back to the topic at hand−registering the standard templates. You've seen `RegisterTemplates` at work. In order to register the standard templates, you need to include two additional files in your project−`rmxftmpl.h` and `rmxfguid.h`. These two files define the template definitions and GUIDs of the standard templates, respectively.

Tip To remember rmxftmpl.h and rmxfguid.h, just remember that rmxf stands for retained mode x−file, tmpl means template, and guid means globally unique identifier.

Inside the rmxftmpl.h file, you'll find the `D3DRM_XTEMPLATES` template data buffer and the `D3DRM_XTEMPLATE_BYTES` macro. These are used in the call to `RegisterTemplates` to register the standard templates, as you can see here:

```
pFile->RegisterTemplates(D3DRM_XTEMPLATES,            \
                         D3DRM_XTEMPLATE_BYTES);
```

That's right; just by calling the above bit of code, you have successfully registered the standard templates, and you're ready to move on! A word of advice before you do: Once you begin using the .X format for your own custom templates and data, don't forget that using `RegisterTemplates` works perfectly for registering your own custom template definitions!

Opening an .X File

After you've created an `IDirectXFile` interface and registered the templates you'll be using, you need to open the .X file and begin enumerating the data objects within it. The process of opening the .X file and creating an enumeration object occurs in one call to the `IDirectXFile::CreateEnumObject` function.

```
HRESULT IDirectXfile::CreateEnumObject(
  LPVOID pvSource,                        // .X filename
  DXFILELOADOPTIONS dwLoadOptions,        // Load options
  LPDIRECTXFILEENUMOBJECT* ppEnumObj);    // Enum interface
```

When you call the `CreateEnumObject` function, specify the file name of the .X file to load as `pvSource` and the interface you'll be using as `ppEnumObj`. As for `dwLoadOptions`, you should specify the value `DXFILELOAD_FROMFILE`, which tells DirectX to load the file from disk. Other possible values for `dwLoadOptions` are `DXFILELOAD_FROMRESOURCE, DXFILELOAD_FROMMEMORY`, and `DXFILELOAD_FROMURL`. These values tell DirectX to load the .X file from a resource, memory buffer, or network URL, respectively. Yep, that's right−you can load .X files directly over the Internet!

Tip To load an .X file from the Internet using a URL, specify the complete network path in `pvSource`.To load from a resource or memory location, just specify the resource handle or memory pointer (both cast as `LPVOID`) in `pvSource`.

Continue the example and create an enumeration object for the .X file. The following code will create an enumeration object used to parse a file from a disk.

```
// Filename = filename to load ("test.x" for example)
IDirectXFileEnumObject *pEnum;
pFile−>CreateEnumObject((LPVOID)Filename,                \
          DXFILELOAD_FROMFILE, &pEnum);
```

From the code's comments, you can see that `Filename` points to a valid file name−in this case, `test.x`. Once successfully called, the `CreateEnumObject` gives you a valid enumeration object (only one is required per open .X file), ready to do all your enumeration dirty work.

Enumerating Data Objects

At this point, you have opened your .X file and registered the templates you'll be using (such as the DirectX standard templates). The enumeration object has been created, and you are now ready to pull data from the .X file.

In its current state, the `IDirectXFileEnumObject` object you created points to the first data object in the file, which is typically the `Header` object. All top−level data objects are siblings of the `Header` object (or the first object in the file). Each data object you read might contain embedded objects (child objects) or references to other data objects; you can query for both of these.

The enumerator object itself doesn't handle a data object's data. Rather, you need to obtain a data object interface, called `IDirectXFileData`, to access the data. To obtain an `IDirectXFileData` interface, you need to call the `IDirectXFileEnumObject::GetNextDataObject` function.

```
HRESULT IDirectXFileEnumObject::GetNextDataObject(          \
          LPDIRECTXFILEDATA* ppDataObj);
```

With only one parameter, the `GetNextDataObject` is a breeze to use. You just need to instance an `IDirectXFileData` object and use it in your call to `GetNextDataObject`.

```
IDirectXFileData *pData;
HRESULT hr = pEnum−>GetNextDataObject(&pData);
```

Notice how I'm saving the return value of the `GetNextDataObject` call? If the return code is an error (which you can check by using the `FAILED` macro), it signifies that the enumeration is complete. If the call to `GetNextDataObject` is successful, then you have yourself a spiffy new interface for accessing the data object's data!

Before you get into working with the object's data, let's finish the discussion on enumeration. So far, you've been able to enumerate the first data object in a file and retrieve its data interface. What do you do when you want to go to the next data object in the .X file or query for embedded data objects?

Once you're finished with a data interface, you need to free it to go to the next data object. Simply calling `IDirectXFileData::Release` will free the data interface, and repeating the call to `IDirectXFileEnumObject::GetNextDataObject` will get the next enumerated sibling (top−level) data object for you. You can wrap the entire enumeration of siblings (grabbing their respective data interfaces) into a code bite such as this one:

```
while(SUCCEEDED(pEnum−>GetNextDataObject(&pData))) {
  // Do something with pData data object

  // Free the data interface in order to continue
  pData−>Release();
}
```

All that's left is to add the ability to query for child (lower−level) data objects, and to allow those child objects to be enumerated and accessed. To query for a child data object, you use the `IDirectXFileData::GetNextObject` function to first see whether a data object contains any embedded objects.

```
HRESULT IDirectXFileData::GetNextObject(          \
         LPDIRECTXFILEOBJECT* ppChildObj);
```

This is another simple function with only one parameter−the pointer to an `IDirectXFileObject` interface. If the call to `GetNextObject` is successful, then you need to process the child data object. Once you've done that, you can free it (by calling `Release`) and continue calling `GetNextObject` until it returns an error code, which signifies that no more child objects remain.

You can wrap the continuous calling of `GetNextObject` into a small loop, as I have done here.

```
IDirectXFileObject *pObject;

while(SUCCEEDED(pData−>GetNextObject(&pObject))) {
  // A child data object exists, need to query for it

  // Free file object interface
  pObject−>Release();
}
```

Once you have a valid `IDirectFileObject` interface (after the call to `GetNextObject`), you can quickly determine which child data object it is currently enumerating (using the techniques coming up in the next section). There's a slight snag, however. A data object can either be referenced or instanced, and the way you access the object varies a bit depending on which type it is.

For instanced objects (those defined normally in an .X file), you can query the `IDirectXFileObject` for an `IDirectXFileData` interface.

```
IDirectXFileData *pSubData;
```

```
// Check if child object is instanced (fails if not)
if(SUCCEEDED(pObject->QueryInterface(                           \
               IID_IDirectXFileData, (void**)&pSubData))) {

  // Child data object exists, do something with it.

  // Free data object
  pSubData->Release();
}
```

You saw the `IDirectXFileData` object in action earlier in this chapter. Using what you've just learned, you can query a child data object's `IDirectXFileData` object for its own embedded child objects.

As for referenced data objects, you need to first query for the `IDirectXFileDataReference` object and resolve the reference into an `IDirectXFileData` object. The following code will query and resolve the referenced data object for you.

Tip If an instanced data object does not exist when you query for it, the call to `QueryInterface` will fail. This is a quick way to tell the type of the data object. The same goes for referenced objects−the query will fail, meaning the object is not referenced.

```
 IDirectXFileDataReference *pRef;
IDirectXFileData *pSubData;

// Check if the data object is referenced (fails if not)
if(SUCCEEDED(pSubObj->QueryInterface(                       \
                              IID_IDirectXFileDataReference,  \
                              (void**)&pRef))) {

  // A data object reference exists. Resolve the reference
  pRef->Resolve(&pSubData);

  // Do something with data object

  // Release the interfaces used
  pRef->Release();
  pSubData->Release();
}
```

Would you believe me if I told you that the hardest part is over? Enumerating the data objects and child objects is simple, and if that's as hard as it gets, then you're in for an easy ride! To make your programming job much easier, I suggest wrapping up the entire enumeration of data objects into two simple functions.

The first function (called `Parse`) will open an .X file, create the enumeration object, and enumerate all top−level data objects. The function will then take each enumerated object and pass it to the second function (`ParseObject`), which will process the data object data based on its template type and scan for embedded child data objects. The `ParseObject` function will call itself using any child objects it finds, thus processing a child's embedded objects.

The code for the `Parse` function follows.

```
// Need to include rmxftmpl.h and rmxfguid.h
BOOL Parse(char *Filename)
{
  IDirectXFile           *pFile = NULL;
  IDirectXFileEnumObject *pEnum = NULL;
  IDirectXFileData       *pData = NULL;

  // Create the enumeration object, return on error
  if(FAILED(DirectXFileCreate(&pFile)))
    return FALSE;

  // Register the standard templates, return on error
  if(FAILED(pFile->RegisterTemplates(                         \
          (LPVOID)D3DRM_XTEMPLATES, D3DRM_XTEMPLATE_BYTES)))
     return FALSE;

  // Create the enumeration object, return on error
  if(FAILED(pDXFile->CreateEnumObject((LPVOID)Filename,       \
  DXFILELOAD_FROMFILE,                                  \
           &pEnum))) {
    pFile->Release();
    return FALSE;
  }

  // Loop through all top-level data objects
  while(SUCCEEDED(pEnum->GetNextDataObject(&pData))) {
    // Parse the data object by calling ParseObject
    ParseObject(pData);

    // Release the data object
    pData->Release();
  }

  // Release used COM objects
  pEnum->Release();
  pFile->Release();
  return TRUE;
}
```

The `Parse` function doesn't hold back any punches, and it certainly isn't overly complicated. I have already explained everything in the function, so there's no need to recap here. Instead, move on to the `ParseObject` function, which takes a data object and queries it for child objects.

```
void ParseObject(IDirectXFileData *pData)
{
  IDirectXFileObject        *pObject  = NULL;
  IDirectXFileData          *pSubData = NULL;
  IDirectXFileDataReference *pRef     = NULL;

  // Scan for embedded objects
  while(SUCCEEDED(pData->GetNextObject(&pObject))) {
    // Look for referenced objects
    if(SUCCEEDED(pObject->QueryInterface(                     \
            IID_IDirectXFileDataReference, (void**)&pRef))) {

      // Resolve the data object
      pRef->Resolve(&pSubData);

      // Parse the object by calling ParseObject
      ParseObject(pSubData);

      // Free interfaces
      pSubData->Release();
      pRef->Release();
    }

    // Look for instanced objects
    if(SUCCEEDED(pObject->QueryInterface(                    \
            IID_IDirectXFileData, (void**)&pSubData))) {

      // Parse the object by calling ParseObject
      ParseObject(pSubData);

      // Free the object interface
      pSubData->Release();
    }

    // Free the interface for next object to use
    pObject->Release();
  }
}
```

Again, the `ParseObject` function doesn't contain anything new. The one thing you'll notice about `Parse` and `ParseObject` is that they don't really do anything except enumerate every data object in an .X file. When it comes time to work with an object's data, what do you do?

Retrieving Data from a Data Object

Remember that data objects are containers for data, and if you're going to the trouble to enumerate data objects, it's a safe bet that you're after the data in each one. Once you've got a valid `IDirectXFileData` object that points at an enumerated data object, you can retrieve the object's instance name, template GUID, and data using a trio of functions. The first function, `IDirectXFileData::GetName`, retrieves the name of the data object instance.

```
HRESULT IDirectXFileData::GetName(
  LPSTR pstrNameBuf,  // Name buffer
  LPDWORD pdwBufLen); // Size of name buffer
```

The `GetName` function takes two parameters−a pointer to a buffer that contains the name and a pointer to a variable that contains the name buffer's size (in bytes). Before you obtain a name from the `GetName` function, you first have to obtain the name's data size by specifying a NULL value for `pstrNameBuf` and supplying a value `DWORD` pointer for `pdwBufLen`.

```
// pData = pre−loaded IDirectXFileData object
// Get size of name, in bytes
DWORD Size;
pData->GetName(NULL, &Size);
```

Once you've got the size of the name buffer, you can allocate an appropriate buffer and read in the name.

```
// Allocate name buffer and get name
char *Name = new char[Size];
pData->GetName(Name, &Size);
```

While having the data object's instance name helps, you really need the GUID of the object's template to distinguish which template an object uses. To retrieve the GUID of the object's template, you use the `IDirectXFileData::GetType` function.

```
HRESULT IDirectXFileData::GetType(
  const GUID ** ppguid);
```

With only one parameter to use−a pointer to a const GUID pointer−you can call the `GetType` function using the following code:

```
const GUID *TemplateGUID = NULL;
pData−>GetType(&TemplateGUID);
```

Now that you have the GUID, you can compare it to a list of internal GUIDs (such as those from the standard templates or from your custom templates) and process the data appropriately. For instance, to check whether a data object's type matches that of the `MeshNormals` standard template, you can use the following code:

```
// TemplateGUID = template's GUID to check
if(*TemplateGUID == TID_D3DRMMeshNormals) {
   // Process MeshNormals template
}
```

Of course, knowing the object's template GUID can only get you so far. The real trick is to get at the data object's data. No problem! With one more simple function call at your disposal, your .X file parsing abilities will be nearly complete! The last function you use to access an object's data is `GetData`.

```
HRESULT IDirectXFileData::GetData(
  LPCSTR szMember,   // Set to NULL
  DWORD *pcbSize,    // Size of data
  void  **ppvData);  // Data pointer
```

To use the `GetData` function, you need to provide a pointer to access the data object's data buffer and a `DWORD` variable to contain the buffer's size (in bytes). Here's a snippet of code that shows how you can use `GetData` to obtain a pointer to the object's data and its size.

```
char *DataPtr;
DWORD DataSize;
pData−>GetData(NULL, &DataSize, (void**)&DataPtr);
```

The pointer to the data buffer now points to a block of contiguous memory that is structured just like the data object's template definition. You can access the data as a large buffer or, if you want to be crafty, you can create a structure to match the template's definition for easier access. For example, suppose you have enumerated the `ColorRGBA` standard template, which is defined as follows:

```
template ColorRGBA {
  <35FF44E0−6C7C−11cf−8F52−0040333594A3>
  FLOAT red;
  FLOAT green;
  FLOAT blue;
  FLOAT alpha;
}
```

To access the `red, green, blue`, and `alpha` values, you can grab the data pointer and cast it to a `float` data type.

```
DWORD DataSize;
float *DataPtr;
pData−>GetData(NULL, &DataSize, (void**)&DataPtr);
float red   = *DataPtr++;
float green = *DataPtr++;
float blue  = *DataPtr++;
float alpha = *DataPtr++;
```

While this approach is fine and dandy, you can process the object's data much easier by using a matching C structure.

```
typedef struct {
  float red, green, blue, alpha;
} sColorRGBA;
sColorRGBA *Color;
DWORD DataSize;
pData->GetData(NULL, &DataSize,
(void**)&Color);
```

Note Notice that the template's GUID or class name is not part of the data retrieved using `IDirectXFileData: :GetData`.

Once it is complete, the preceding code gives you the ability to access the colors using the structure instance.

```
float red = Color->red;
float blue = Color->blue;
float green = Color->green;
float alpha = Color->alpha;
```

Accessing single variables is easy, but what about strings and arrays? Arrays, being the easier of the two, are stored contiguously in memory, meaning that you can just increase the memory pointer that contains the object's data. For example, the following code shows you how to access the array of float values stored in a data object of the template type `FloatKeys`.

```
// Get the object's data size & pointer
DWORD DataSize;
DWORD *DataPtr;
pData->GetData(NULL, &DataSize, (void**)&DataPtr);

// The FloatKeys template has a DWORD value 1st that
// defines how many float values are in the array
DWORD NumKeys = *DataPtr++;

// Next, an array of float values follows
for(DWORD i=0;i<NumKeys;i++) {
  float fValue = *(FLOAT*)DataPtr++;
```

Accessing arrays wasn't too difficult, so how about accessing strings? Again, it's an easy chore because strings are stored as pointers to a text buffer, which you can access much like I do in the following code. (I'm using the `TextureFilename` template as an example; it stores the name of a file to use for a texture.)

```
// Get the data pointer & size
DWORD DataSize;
DWORD *DataPtr;
pData->GetData(NULL, &DataSize, (void**)&DataPtr);

// Now, access the filename text buffer
char *StringPtr = (char*)*DataPtr;
MessageBox(NULL, StringPtr, "Texture Filename", MB_OK);
```

With a simple cast to a `char` pointer, you were able to display the file name contained in the `TextureFilename` template. Now I know you've got to be banging your forehead and yelling, "Why didn't

I see how easy this was before?" Whoa, down boy! I didn't immediately realize just how easy it was to work with .X files either. Now that the secret is out, nothing can stop you from using .X files almost exclusively in your own projects. All you need is a way to wrap up all of this .X parser functionality into a class, making it even easier to work with .X.

Constructing an .X Parser Class

So, you want to create a class to handle all aspects of parsing .X files, eh? Sounds great to me! In this .X file parser class, you can wrap up the `Parse` and `ParseObject` functions you saw earlier in this chapter, in the "Enumerating Data Objects" section. Use the code from those two functions and write the parser class to allow yourself to override the data object parsing functions, which will allow you to scan for specific objects.

Start the parser class with a simple definition and go from there.

```
class cXParser
{
  protected:
  // Function called for every template found
  virtual BOOL ParseObject(                              \
                 IDirectXFileData *pDataObj,             \
                 IDirectXFileData *pParentDataObj,       \
                 DWORD Depth,                            \
                 void **Data, BOOL Reference)
          {
            return ParseChildObjects(pDataObj, Depth,     \
                                         Data, Reference);
          }

  // Function called to enumerate child templates
  BOOL ParseChildObjects(IDirectXFileData *pDataObj,       \
                           DWORD Depth, void **Data,       \
                           BOOL ForceReference = FALSE);

  public:
    // Function to start parsing an .X file
    BOOL Parse(char *Filename, void **Data = NULL);
};
```

Whoa! I know I said you should start with a simple definition, not what I've just shown here! Bear with me friends, because you'll quickly realize just how simple this class is going to be. So far, you have three functions in your new `cXParser` .X file parser class. You use these three functions (`ParseObject, ParseChildObjects,` and `Parse`) to process a single data object, scan for embedded child objects, and parse an entire file, respectively.

`cXParser::Parse`, which is the easiest of the functions, merely duplicates the code in the `Parse` function you used earlier in this chapter. I'll leave out the code here, but if you look at the code from the CD−ROM (check out \BookCode\Common\XParser.cpp and XParser.h), you'll notice the addition of a supposed data pointer and a couple lines of code that call two unknown functions, `BeginParse` and `EndParse`. I'll talk about these two functions in a moment; for now, just ignore them.

The second function, `ParseObject`, is your .X parser's workhorse. `ParseObject` is called for every single data object found in an .X file. You need to override the `ParseObject` function (a virtual function) for it to do something useful. As you can see from the `ParseObject` function prototype, there's a lot going on that'll need some explanation.

The first parameter for `ParseObject` is an `IDirectXFileData` object which, as you saw earlier in this chapter, represents the data object that is currently being enumerated. Inside your overridden function, you can access the object's data via the `pDataObj` pointer.

The second parameter, `pParentDataObj` (also an `IDirectXFileData` object), represents the parent (higher−level object) of the current data object that is being enumerated. This is provided in case you want to see whether the current object is a child of another object.

The `Depth` parameter measures the depth of the object in the hierarchy. The highest−level data objects are at a depth of 0, whereas child objects have their parent's depth plus one. As an example, I have shown a few `Frame` objects here, with their respective depths listed.

```
Frame RootFrame { // Depth = 0
  Frame ChildofRoot { // Depth = 1
    Frame ChildofChild { // Depth = 2
    }
  }
  Frame SiblingofRootChild { // Depth = 1
  }
}
Frame RootSibling { // Depth = 0
}
```

`Data` is the fourth parameter of `ParseObject`. It is a user−defined data pointer (or rather, a pointer to a data pointer) that you use to pass information to your parser functions. For example, you can create a data structure to contain all of the information you need.

```
typedef struct sDATA {
  long NumObjects;
  sDATA() { NumObjects = 0; }
} sDATA:
```

Note The depth of a data object is extremely useful for sorting hierarchies, such as frame hierarchies used in skeletal animation.

To pass an `sDATA` structure to your parsing functions, you instance it and use it during a call to `cXParser::Parse`, as shown here:

```
sDATA Data;
cXParser Parser;
Parser.Parse("test.x", (void**)&Data);
```

From then on, every time `ParseObject` is called you can cast an appropriate pointer to access your data structure.

```
BOOL cXParser::ParseObject(IDirectXFileData *pDataObj,        \
                           IDirectXFileData *pParentDataObj,  \
                           DWORD Depth,                       \
                           void **Data, BOOL Reference)
{
  cDATA *DataPtr = (sDATA*)*Data;
  DataPtr−>NumObjects++; // Increase object count

  return ParseChildObjects(pDataObj,Depth,Data,Reference);
}
```

I know I'm getting ahead of myself again by showing you some sample code for `cXParser`, so let's jump back to the fifth (and last) parameter for `ParseObject−Reference`. The `Reference` Boolean variable determines whether the data object being enumerated is referenced or instanced. You can use the `Reference` variable to determine whether you want to load a referenced object's data or wait for the object to be instanced. This is useful when it comes time to load animation data, which needs data object references rather than actual object instances.

Whew! With the `ParseObject` function set aside, you're left with the last of the trio of `cXParser` functions−`ParseChildObjects`. Thankfully, the `ParseChildObjects` function is easy−it merely enumerates any child data objects of the object you pass it. Typically, you call `ParseChildObjects` at the end of your `ParseObject` function, as I did in the last code bit.

You can see that you need to pass the current `IDirectXFileData` object, data object depth, data pointer, and reference flag to `ParseChildObjects` because it is responsible for increasing the depth and setting the parent data object as needed for the next call to `ParseObject`. If you don't want to parse any child data objects, however, you can skip the call to `ParseChildObjects` and return a `TRUE` or `FALSE` value. (`TRUE` forces enumeration to continue, whereas `FALSE` stops it.) You'll see examples of the `ParseObject` and `ParseChildObjects` functions throughout this book.

Now that the basics are in place, you need to expand on your parser class a bit. How about adding some functions to retrieve a data object's name, GUID, and data pointer, as well as inserting a couple of functions that are called before and after an .X file is parsed? Take a look at the following code to see what your new parser class should look like.

```
class cXParser
{
  protected:
    // Functions called when parsing begins and end
    virtual BOOL BeginParse(void **Data) { return TRUE; }
    virtual BOOL EndParse(void **Data) { return TRUE; }

    // Function called for every data object found
    virtual BOOL ParseObject(                            \
                   IDirectXFileData *pDataObj,           \
                   IDirectXFileData *pParentDataObj,     \
                   DWORD Depth,                          \
                   void **Data, BOOL Reference)
            {
              return ParseChildObjects(pDataObj, Depth,    \
                                           Data, Reference);
            }

    // Function called to enumerate child objects
    BOOL ParseChildObjects(IDirectXFileData *pDataObj,     \
                             DWORD Depth, void **Data,     \
                             BOOL ForceReference = FALSE);

  public:
    // Function to start parsing an .X file
    BOOL Parse(char *Filename, void **Data = NULL);

    // Functions to help retrieve data object information
    const GUID *GetObjectGUID(IDirectXFileData *pDataObj);
    char *GetObjectName(IDirectXFileData *pDataObj);
    void *GetObjectData(IDirectXFileData *pDataObj,DWORD *Size);
};
```

You can see the addition of the BeginParse, EndParse, GetObjectGUID, GetObjectName, and GetObjectData functions in cXParser. You've already seen the code for the three Get functions−it's the virtual BeginParse and EndParse functions that are unknown.

In their current form, both BeginParse and EndParse return TRUE values, which signify a successful function call. It's your job to override these two functions in a derived class so that you can perform any operations prior to and following a file parse. For instance, you might want to initialize any data or provide a data pointer inside your BeginParse function and clean up any used resources inside the EndParse function.

Both the BeginParse and EndParse functions are called directly from the Parse function−you merely need to override them and write the code for them. You'll see these two functions in use throughout the book and in the upcoming "Loading Frame Hierarchies from .X" section in this chapter.

As for the three Get functions, you use those by passing a valid IDirectXFileData object; in return, you'll get the name in a newly−allocated data buffer, a pointer to the template's GUID, or a pointer to the object's data buffer and data size value. For example, here's some code that calls the three functions to get and access an object's data:

```
// pData = pre−loaded data object
char *Name = pParser−>GetObjectName(pData);
const GUID *Type = pParser−>GetObjectGUID(pData);
DWORD Size;
char *Ptr = (char*)pParser−>GetObjectData(pData, &Size);

// Do something with data and free up resource when done
delete [] Name;
```

As I was saying, using your brand−new cXParser class is going to be really simple. Throughout the remainder of this chapter and the rest of this book, you'll come to rely on the cXParser class to parse .X files for you. Advanced readers might want to take it upon themselves to modify the parser class to fit their individual needs. I personally find the class very useful in its current form.

As a quick test of your new cXParser class, let's see just how to derive and use it. Suppose you want to parse .X files for all Mesh data objects and display each one's name in a message box. Here's the code for the parser that will do just that:

```
class cParser : public cXParser
{
  public:
    cParser() { Parse("test.x"); }
    BOOL ParseObject(IDirectXFileData *pDataObj,             \
                     IDirectXFileData *pParentDataObj,   \
                     DWORD Depth,                        \
                     void **Data, BOOL Reference)
    {
      if(*GetObjectGUID(pDataObj) == TID_D3DRMMesh) {
        char *Name = GetObjectName(pDataObj);
        MessageBox(NULL, Name, "Mesh template", MB_OK);
        delete [] Name;
      }
      return ParseChildObjects(pDataObj,Depth,Data,Reference);
    }
};
cParser Parser; // Instancing this will run the parser
```

Now tell me that wasn't easy! Enough basics, let's get into using .X files for something useful, such as 3D mesh data.

Loading Meshes from .X

Now that you've got a firm grip on how the .X file format works, consider how Microsoft first intended for you to use it−to contain 3D mesh information for your games. The D3DX library comes with a number of functions you can use to load mesh data from an .X file. With the addition of the .X parser developed in this chapter, you have even more options available to you. Check out just how easy it is to work with D3DX to load mesh data.

Loading Meshes Using D3DX

The D3DX library defines the handy `ID3DXMesh` object that contains and renders 3D meshes for you. Although you can use your own custom mesh storage containers, I find sticking to the `ID3DXMesh` object perfectly sensible. That's the object I'll use for the rest of this chapter (with the exception of the also−handy `ID3DXSkinMesh` object, which you'll see in a moment).

The fastest way to load mesh data using D3DX is to call on the `D3DXLoadMeshFromX` and `D3DXLoadMeshFromXof` functions. Both of these meshes will take the data contained within an .X file and convert it to an `ID3DXMesh` object. The `D3DXLoadMeshFromX` file loads an entire .X file at once (compressing all meshes into a single output mesh), whereas the `D3DXLoadMeshFromXof` function loads a single mesh pointed at by an `IDirectXFileData` object.

The `D3DXLoadMeshFromX` function provides a file name to load, some flags to control loading aspects, a 3D device pointer, pointers to buffers for containing material data, and some miscellaneous data pointers that you can ignore for now. Take a look at the prototype for `D3DXLoadMeshFromX`.

```
HRESULT D3DXLoadMeshFromX(
  LPSTR pFilename,           // Filename of .X file to load
  DWORD Options,             // Loading options
  LPDIRECT3DDEVICE9 pDevice, // 3D device pointer
  LPD3DXBUFFER* ppAdjacency, // Set to NULL
  LPD3DXBUFFER* ppMaterials, // Buffer for materials
  LPD3DXBUFFER* ppEffectInstances, // Not used here − NULL
  PDWORD pNumMaterials,      // # materials loaded
  LPD3DXMESH* ppMesh);       // Pointer to mesh interface
```

The comments I've shown are pretty self−explanatory, so to speed things up take a look at `D3DXLoadMeshFromX` in action and go from there. First, you need to instance an `ID3DXMesh` object.

```
ID3DXMesh *Mesh = NULL;
```

From there, suppose you want to load the .X file called `test.x`. Simple enough−you just need to specify the file name in the call to `D3DXLoadMeshFromX` but wait! What's with the `Options` parameter? You use it to tell D3DX how to load the mesh−into system memory or video memory, using write−only memory, and so on. A flag represents each option. Table 3.3 lists some of the most common macros.

Table 3.3: D3DXLoadMeshFromX Option Flags

Macro	Description
`D3DXMESH_32BIT`	Uses 32−bit indices (not always supported).
`D3DXMESH_USEHWONLY`	Uses hardware processing only. Use this only for devices that are definitely hardware accelerated.
`D3DXMESH_SYSTEMMEM`	Forces the mesh to be stored in system memory.
`D3DXMESH_WRITEONLY`	Sets a mesh's data as write−only, thus allowing Direct3D to find the best location to store the mesh data (typically in video memory).
`D3DXMESH_DYNAMIC`	Uses dynamic buffers (for meshes that change during run time).
`D3DXMESH_SOFTWAREPROCESSING`	Forces the use of software vertex processing, which is used in place of the hardware transform and lighting engine.

From Table 3.3, you can see there are really not many options for loading meshes. In fact, I recommend using only `D3DXMESH_SYSTEMMEM` or `D3DXMESH_WRITEONLY` The first option, `D3DXMESH_SYSTEMMEM`, forces your mesh's data to be stored in system memory, making access to the mesh data faster for both read and write operations.

Specifying `D3DXMESH_DYNAMIC` means you are going to change the mesh data periodically. It's best to specify this flag if you plan to periodically change the mesh's data (such as vertices) in any way during run time.

If speed is your need, then I suggest using the `D3DXMESH_WRITEONLY` flag, which tells D3DX to use memory that will not allow read access. Most often that means you will use video memory because it is usually (but not always) write−only. If you're not going to read a mesh's vertex data, then I suggest using this flag.

Tip If you're not using system memory or write−only access, what's left to use? Just specify a value of 0 for `Options` in the call to `D3DXLoadMeshFromX`, and you'll be fine.

Getting back to the parameters of `D3DXLoadMeshFromX`, you'll see the pointer to a 3D device interface. No problem−you should have one of those hanging around in your project! Next is the `ID3DXBUFFER` pointer, `ppAdjacency`. Set that to NULL−you won't be using it here.

The next three parameters, `ppMaterials, ppEffectInstance,` and `pNumMaterials`, contain the mesh's material information, such as material color values and texture file names, as well as effects data. If you're using DirectX 8, you can safely exclude the `ppEffectInstance` reference−it doesn't exist in that version. If you're using DirectX 9, you can set `ppEffectInstance` to NULL because you don't require any effects information.

The `ppMaterials` pointer points to an `ID3DXBuffer` interface, which is merely a container for data. `pNumMaterials` is a pointer to a `DWORD` variable that will contain the number of materials in a mesh that was loaded. You'll see how to use material information in a moment.

Finishing up with `D3DXLoadMeshFromX`, you see the actual `ID3DXMesh` object pointer−`ppMesh`. This is the interface you supply to contain your newly loaded mesh data. And there you have it! Now put all of this stuff together into a working example of loading a mesh.

Load a mesh, again called `test.x`, using write−only memory. After you've instanced the mesh object pointer, you need to instance an `ID3DXBuffer` object to contain material data and a `DWORD` variable to contain the number of materials.

```
ID3DXBuffer *pMaterials = NULL;
DWORD NumMaterials;
```

From here, call `D3DXLoadMeshFromX`.

```
// pDevice = pointer to a valid IDirect3DDevice9 object
D3DXLoadMeshFromX("test.x", D3DXMESH_WRITEONLY, pDevice,        \
               NULL, &pMaterials, NULL, &NumMaterials, &Mesh);
```

Great! If everything went as expected, `D3DXLoadMeshFromX` will return a success code, and your mesh will have been loaded in the `ID3DXMesh` interface `Mesh!` Of course, every single mesh contained in the .X file was crunched into a single mesh object, so how about those times when you want access to each separately−defined mesh in the file?

This is where the `D3DXLoadMeshFromXof` file comes in. You use the `D3DXLoadMeshFromXof` function in conjunction with your .X parser to load mesh data from an enumerated `Mesh` object. Just take a look at the `D3DXLoadMeshFromXof` function prototype to see what this entails.

```
HRESULT D3DXLoadMeshFromXof(
  LPDIRECTXFILEDATA pXofObjMesh,
  DWORD Options,
  LPD3DXBUFFER* ppMaterials,
  LPD3DXBUFFER* ppEffectInstances,
  PDWORD pNumMaterials,
  LPD3DXMESH* ppMesh);
```

Now wait a sec! `D3DXLoadMeshFromXof` looks almost exactly like `D3DXLoadMeshFromX!` The only difference is the first parameter; instead of a pointer to an .X file name to load, `D3DXLoadMeshFromXof` has a pointer to an `IDirectXFileData` object. By providing the pointer to a currently enumerated `IDirectXFileData` object, D3DX will take over and load all of the appropriate mesh data for you! And since every other parameter is the same as in `D3DXLoadMeshFromX`, you'll have no trouble using `D3DXLoadMeshFromXof` in your .X parser class!

Now hang on to your horses for just a bit, because you'll see how to use `D3DXLoadMeshFromXof` in your parser class later in this chapter, in the "Loading Meshes with Your .X Parser" section.

Regardless of which function you use to load the mesh data (`D3DXLoadMeshFromX` or `D3DXLoadMeshFromXof`), all you have left to do is process the material information once a mesh has been loaded into an `ID3DXMesh` object.

To process the material information, you need to retrieve a pointer to the material's `ID3DXBuffer`'s data buffer (used in the call to `D3DXLoadMeshFromX` or `D3DXLoadMeshFromXof`) and cast it to a `D3DXMATERIAL` type. From there, iterate all materials, using the number of materials set in `NumMaterials` as your index. Then you need to instance an array of `D3DMATERIAL9` structures and `IDirect3DTexture9` interfaces to contain the mesh's material data. Use the following code to process the material information:

```
// Objects to hold material and texture data
D3DMATERIAL9 *Materials = NULL;
IDirect3DTexture9 *Textures = NULL;

// Get a pointer to the material data
```

```
D3DXMATERIAL *pMat;
pMat = (D3DXMATERIAL*)pMaterials->GetBufferPointer();

// Allocate material storage space
if(NumMaterials) {
Materials = new D3DMATERIAL9[NumMaterials];
Textures = new IDirect3DTexture9*[NumMaterials];

// Iterate through all loaded materials
for(DWORD i=0;i<NumMaterials;i++) {
  // Copy over the material information
  Materials[i] = pMat[i].MatD3D;
  // Copy diffuse color over to ambient color
  Materials[i].Ambient = Materials[i].Diffuse;

  // Load a texture if one is specified
  Textures[i] = NULL;
  if(pMat[i].pTextureFilename) {
    D3DXCreateTextureFromFile(pDevice,               \
    pMat[i].pTextureFilename, &Textures[i]);
  }
 }
} else {
  // Allocate a default material if none were loaded
  Materials = new D3DMATERIAL9[1];
  Textures = new IDirect3DTexture9*[1];

  // Set a default white material and no texture
  Textures[0] = NULL;
  ZeroMemory(&Materials[0], sizeof(D3DMATERIAL9));
  Materials[0].Diffuse.r = Materials[0].Ambient.r = 1.0f;
  Materials[0].Diffuse.g = Materials[0].Ambient.g = 1.0f;
  Materials[0].Diffuse.b = Materials[0].Ambient.b = 1.0f;
  Materials[0].Diffuse.a = Materials[0].Ambient.a = 1.0f;
}
// Free material data buffer
pMaterials->Release();
```

You can see in the preceding code that I added a case to check whether no materials were loaded, in which case you need to create a default material to use. Once those materials are loaded, you're ready to begin using the mesh interface to render. Chapter 1 showed you how to use the mesh interface to render graphics. For now, you can move forward by jumping back to a topic I pushed aside earlier−loading meshes using your .X parser.

Loading Meshes with Your .X Parser

Just as I promised, it's time to check out how to merge the mesh−loading functions into your .X parser class. Since you're going to be accessing the mesh data objects directly, you need to use the `D3DXLoadMeshFromXof` function to load mesh data. That means you need to parse each data object, and look for `Mesh` objects as you go. Start by deriving a parser class with which to work.

```
class cXMeshParser : public cXParser
{
  public:
    ID3DXMesh *m_Mesh;

  public:
    cXMeshParser() { m_Mesh = NULL; }
    ~cXMeshParser() { if(m_Mesh) m_Mesh−>Release(); }

    BOOL ParseObject(IDirectXFileData *pDataObj,   \
             IDirectXFileData *pParentDataObj,     \
             DWORD Depth,                          \
             void **Data, BOOL Reference);
};
```

As you can see from the class declaration, I'm only declaring one mesh object. If you want more, you need to create a linked list (or another type of list) to contain the mesh objects. I'll leave that up to you because for now I just want to demonstrate using the `D3DXLoadMeshFromXof` function.

Override the `ParseObject` function, allowing it to scan for `Mesh` objects.

```
BOOL cXMeshParser::ParseObject(                           \
                       IDirectXFileData *pDataObj,         \
                       IDirectXFileData *pParentDataObj,   \
                       DWORD Depth,                        \
                       void **Data, BOOL Reference)
{
  // Skip reference objects
  if(Reference == TRUE)
    return TRUE;

  // Make sure object being parsed is Mesh
  if(*GetObjectGUID(pDataObj) == D3DRM_TIDMesh) {
    // It's a mesh, use D3DXLoadMeshFromXof to load it
    ID3DXBuffer *Materials;
    DWORD NumMaterials;
    D3DXLoadMeshFromXof(pDataObj, 0, pDevice, NULL,        \
               &Materials, NULL, &NumMaterials, &m_Mesh);

    // Finish by processing material information

    // return FALSE to stop parsing
    return FALSE;
  }
  // Parse child objects
  return ParseChildObjects(pDataObj,Depth,Data,Reference);
}
```

There you have it! With one quick call, you've loaded a mesh from a `Mesh` data object! If you think that's cool, I've got something new for you−skinned meshes. That's right; those nifty skinned meshes you read about in Chapter 2 are just as easy to work with as standard meshes. Read on to see how to load skinned mesh data from .X files.

Loading Skinned Meshes

As I mentioned in Chapter 1, a skinned mesh contains a hierarchy of bones (a skeletal structure) that you can use to deform the mesh to which the bones are attached. Although Chapter 1 detailed the use of skinned meshes, I left the question of loading skinned mesh data for this chapter because you can only load skinned mesh data using an .X parser class. Lucky for you, you're prepared!

Loading skinned meshes from .X files is just like loading regular meshes. By enumerating the data objects, you can determine which ones to load skinned mesh data from and put the data into an `ID3DXSkinMesh` object.

The surprising thing here is that a skinned mesh's data is contained in the same `Mesh` objects as a regular mesh! If it's the same data object, how could you possibly know the difference between a skinned mesh and a regular mesh?

The only way to determine whether a `Mesh` data object contains skinned mesh data is to use the `D3DXLoadSkinMeshFromXof` function to load the mesh into an `ID3DXSkinMesh` object. After the mesh data is loaded, you can query the newly created skinned mesh object to see whether it contains bone information (which is contained in special data objects embedded within the `Mesh` object). If bone information exists, the mesh is skinned. If no bones exist, the mesh is regular and can be converted to an `ID3DXMesh` object.

I'm starting to get ahead of myself, so jump back to the whole `ID3DXSkinMesh` and `D3DXLoadSkinMeshFromXof` thing. Much like regular meshes, you must instance an `ID3DXSkinMesh` object.

```
ID3DXSkinMesh *SkinMesh = NULL;
```

Inside your `ParseObject` function, you need to change the `D3DXLoadSkinMeshFromXof` call. Instead of calling `D3DXLoadMeshFromXof` this time, use `D3DXLoadSkinMeshFromXof`.

```
HRESULT D3DXLoadSkinMeshFromXof(
  LPDIRECTXFILEDATA pXofObjMesh,
  DWORD Options,
  LPDIRECT3DDEVICE9 pDevice,
  LPD3DXBUFFER* ppAdjacency,
  LPD3DXBUFFER* ppMaterials,
  LPD3DXBUFFER* ppEffectInstances,
  DWORD* pMatOut,
  LPD3DXSKINMESH* ppMesh);
```

Note DirectX 8 users can cut out the `LPD3DXBUFFER* ppEffectInstances` line from the call to `D3DXLoadSkinMeshFromXof`.

I know you've got to be saying that `D3DXLoadSkinMeshFromXof` looks almost exactly like `D3DXLoadMeshFromXof`, and you're right! Loading the skinned mesh using `D3DXLoadSkinMeshFromXof` looks something like this:

```
D3DXLoadSkinMeshFromXof(pDataObj, 0, pDevice, NULL,          \
                 &Materials, NULL, &NumMaterials, &SkinMesh);
```

Once you have called `D3DXLoadSkinMeshFromXof` using a valid `Mesh` object, you'll have a cool new `ID3DXSkinMesh` object at your disposal.

Loading Frame Hierarchies from .X

Skeletal animation systems require a frame hierarchy (which represents the bone structure) to orient each bone during rendering. The .X file format defines a frame−of−reference data template that you can use to define your bone hierarchy. This template, `Frame`, is merely a placeholder of sorts. It allows any type of data object to be embedded in it so you can reference the instanced `Frame` object by its assigned instance name and allow all contained objects to be addressed as well.

Building a frame hierarchy involves parsing an .X file, in which you link each `Frame` object to another. The relationship is very important−a `Frame` object can either be a sibling or a child to another `Frame` object, and it can have an unlimited number of siblings and/or child objects linked to it.

In DirectX 9, you have access to a special DirectX structure called `D3DXFRAME` to contain each frame in your hierarchy. I introduced you to this structure in Chapter 1. In this section, you'll use `D3DXFRAME` to contain a frame hierarchy.

The exact way you parse and build your `Frame` objects is really up to you. You can use the D3DX library to help, or you can parse/build the hierarchy yourself using your custom .X parser. Which is better for you? Whereas the D3DX library is great to use, I find the methods of using the library to load a frame hierarchy overly complicated, with no less than two interfaces to use and an entire memory−handling class of your own to be written. Why bother when you can load a frame hierarchy using one small .X parser class you created?

Instead, load up your trusty .X parser (which you developed earlier in this chapter, in the "Constructing an .X Parser Class" section) and derive a version that scans for `Frame` data objects. I'll start you off by showing you the derived class you can use.

```
class cXFrameParser : public cXParser
{
  public:
    // declare an extended version of D3DXFRAME
    // that contains a constructor and destructor
    struct D3DXFRAME_EX : public D3DXFRAME {
       D3DXFRAME_EX()
       {
          Name = NULL;
          pFrameSibling = NULL; pFrameFirstChild = NULL;
       }
       ~D3DXFRAME_EX()
       {
         delete [] Name;
         delete pFrameFirstChild;
         delete pFrameSibling;
       }
    } D3DXFRAME_EX;

    // Instance the root frame of the hierarchy
    D3DXFRAME_EX *m_RootFrame;

  public:
    cXFrameParser()  { m_RootFrame = NULL; }
    ~cXFrameParser() { delete m_RootFrame; }

    BOOL BeginParse(void **Data)
    {
      // Clear hierarchy
      delete m_RootFrame; m_RootFrame = NULL;
    }

    BOOL ParseObject(IDirectXFileData *pDataObj,            \
                     IDirectXFileData *pParentDataObj,   \
                     DWORD Depth,                        \
                     void **Data, BOOL Reference)
    {
      // Skip reference frames
      if(Reference == TRUE)
```

```
        return TRUE;

      // Make sure template being parsed is a frame
      if(*GetObjectGUID(pDataObj) == TID_D3DRMFrame) {

        // Allocate a frame structure
        D3DXFRAME_EX *Frame = new D3DXFRAME_EX();

       // Get frame name (assign one if none found)
       if((Frame->Name = GetObjectName(pDataObj)) == NULL)
         Frame->Name = strdup("No Name");

      // Link frame structure into list
      if(Data == NULL) {
        // Link as sibling of root
        Frame->pFrameSibling = m_RootFrame;
        m_RootFrame = Frame;
        Data = (void**)&m_RootFrame;
      } else {
        // Link as child of supplied frame
        D3DXFRAME_EX *FramePtr = (D3DXFAME_EX*)*Data;
        Frame->pFrameSibling = FramePtr->pFrameFirstChild;
        FramePtr->pFrameFirstChild = Frame;
        Data = (void**)&Frame;
    }
  }
  return ParseChildObjects(pDataObj,Depth,Data,Reference);
};
cXFrameParser Parser;
Parser.Parse("frames.x");
// Parser.m_RootFrame now points to root frame of hierarchy
```

There you have it. Once the `cXFrameParser::Parse` function is complete, you'll have a self−contained frame hierarchy ready to use in your project. To better understand the use of this class, check out the ParseFrame demo on this book's CD−ROM. (See the end of this chapter for more information.) The ParseFrame demo loads an .X file of your choice and displays the frame hierarchy inside a list box.

Throughout the remainder of this book, you'll see how you can put a frame hierarchy to good use in your animation techniques.

Loading Animations from .X

Although a basic concept such as animation data certainly deserves a spot here, I want to delay the discussion of loading animation data until the chapters on working with precalculated animation, because it fits better with the flow of the book. Animation data can be any format (just like any data), but in the remaining chapters of this book, you'll see how you can develop your own set of animation templates to use for your projects. Don't worry, you're almost there; just don't give up yet!

Loading Custom Data from .X

As I've expressed throughout this chapter, the .X file format is completely open−ended; there is no limit to the type of data you can store. With that in mind, you can create any type of data storage you want, and accessing that data will be just as easy as accessing the mesh and frame data we've already covered.

Jump back to your `ContactEntry` template and see just how to parse an .X file and display every instance of `ContactEntry`. Again, a single small derived class of `cXParser` will work perfectly here.

```
// First declare the ContactEntry GUID
#include "initguid.h"
DEFINE_GUID(ContactEntry,                                   \
            0x4c9d055b, 0xc64d, 0x4bfe, 0xa7, 0xd9, 0x98,   \
            0x1f, 0x50, 0x7e, 0x45, 0xff);

// Now, define the .X parser class
class cXContactParser : public cXParser
{
  public:
    BOOL ParseObject(IDirectXFileData *pDataObj,            \
                       IDirectXFileData *pParentDataObj,    \
                       DWORD Depth,                         \
                       void **Data, BOOL Reference)
    {
      // Skip reference objects
      if(Reference == TRUE)
        return TRUE;

      // Make sure object being parsed is ContactEntry
      if(*GetObjectGUID(pDataObj) == CONTACTENTRY) {
        // Get the data pointer and size
        DWORD DataSize;
        DWORD *DataPtr;
        DataPtr = (DWORD*)GetObjectData(pDataObj, &DataSize);
        // Get name from object data
        char *Name = (char*)*DataPtr++;
        // Get phone # from object data
        char *PhoneNum = (char*)*DataPtr++;
        // Get age from object data
        DWORD Age = *DataPtr++;

        // Display contact information
        char Text[1024];
        sprintf(Text, "Name: %s\r\nPhone #: %s\r\nAge: %lu", \
               Name, PhoneNum, Age);
               MessageBox(NULL, Text, "Contact", MB_OK);
      }
      return ParseChildObjects(pDataObj,Depth,Data,Reference);
    }
};
cXContactParser Parser;
Parser.Parse("contacts.x");
```

With a single call to `cXContactParser::Parse`, you're treated to a list of people's names, phone numbers, and ages−all contained in the contacts.x file! Now wasn't that easy? See, you don't have to be afraid of the .X file format. I personally use .X to store as much custom data as I can.

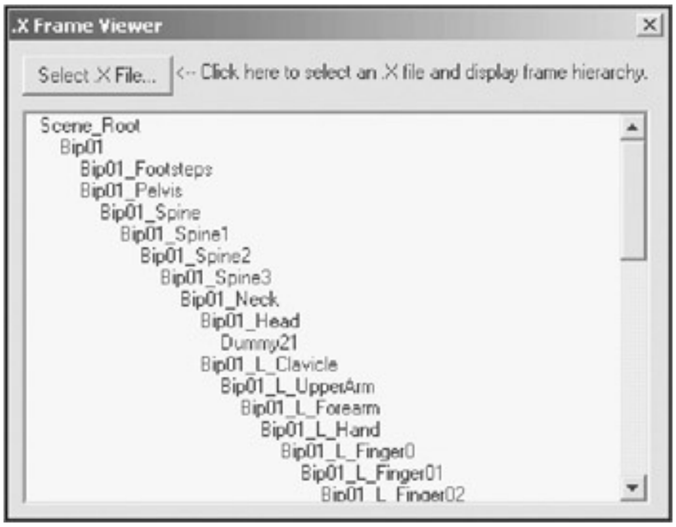
I suggest that you reread this chapter a few times before you move on; get a grasp on the concept of creating your own templates and accessing the data objects' data. Throughout the rest of this book, you will rely on the .X format to store custom data related to each animation topic, so I want you to feel comfortable with .X.

Check Out the Demos

In this chapter you got to check out what .X is all about, from creating your own templates to using those templates to create data objects that hold your game's vital data. To help demonstrate the usefulness of .X and how to use the code presented in this chapter, I have included two demonstration programs (ParseFrame and ParseMesh) on this book's CD−ROM. Check out what each demo does.

ParseFrame

The first demo for this chapter is ParseFrame, which uses the information on loading a frame hierarchy from an .X file. As shown in Figure 3.2, the ParseFrame demo has a button labeled Select .X File. Click that button, locate an .X file to load, and click Open.


.X Frame Viewer
Select .X File...
<-- Click here to select an .X file and display frame hierarchy.

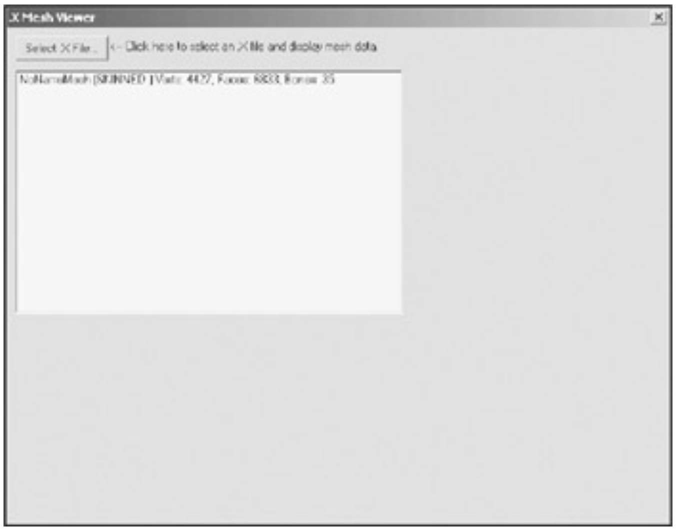
```
Scene_Root
  Bip01
    Bip01_Footsteps
    Bip01_Pelvis
      Bip01_Spine
        Bip01_Spine1
          Bip01_Spine2
            Bip01_Spine3
              Bip01_Neck
                Bip01_Head
                  Dummy21
                Bip01_L_Clavicle
                  Bip01_L_UpperArm
                    Bip01_L_Forearm
                      Bip01_L_Hand
                        Bip01_L_Finger0
                          Bip01_L_Finger01
                            Bip01_L_Finger02
```

Figure 3.2: The ParseFrame demo's dialog box contains two controls−a button you click to load an .X file and a list box that displays the frame hierarchy.

After selecting an .X file to load and clicking Open, you'll be treated to a display of the frame hierarchy contained within the file. In Figure 3.2, the frame hierarchy shown is from the Tiny.x file located in your DirectX SDK's \samples\media directory. To make things easier to understand, the frames are indented by their level in the hierarchy.

ParseMesh

The second demo in the Chapter 3 directory is ParseMesh. Much like the ParseFrame demo program, ParseMesh contains a button (shown in Figure 3.3) that you click to locate and open an .X file.



```
NoNameMesh [SKINNED] Verts: 4427, Faces: 8833, Bones: 35
```

Figure 3.3: After locating and opening an .X file, you are shown some vital data on each mesh contained in that file.

As opposed to the frame−hierarchy listing the ParseFrame demo produced, the ParseMesh demo lists information about the meshes it finds in the .X file you opened. This data includes the type of mesh (standard or skinned), the number of vertices, the number of faces, and (when applicable) the number of bones. You can use this handy demo program as an example for your own .X file, to make sure all the meshes contain the proper information.

Programs on the CD

In the Chapter 3 directory, you'll find two projects that demonstrate the use of the .X parsing class developed in this chapter. These two projects are

- **ParseFrame.** This is a demo program that uses the `cXParser` class to load and create a frame hierarchy from an .X file. It is located at \BookCode\Chap03\ParseFrame.
- **ParseMesh.** You can use this program to load and display information about meshes (regular and skinned). It is located at \BookCode\Chap03\ParseMesh.

Part Three: Skeletal Animation

Chapter 4: Working with Skeletal Animation

Overview

As the darkness envelops my character, I can't help but let out a slight chuckle. "Those fools will never see it coming," I say to myself. As I push up on the joystick, my character slowly crawls forward. Coming to a corner, I press a button and suddenly he scales the wall and sits in wait for his prey. Waiting in silent anticipation, my character slowly checks his night−vision goggles, weapon, and other fancy gadgets he brought along for his devious endeavors.

All of these animations are happening in real time, so there's no jumping or breaking as the character changes his actions. From running to crawling to climbing walls and nonchalantly checking his gear, all of the animations are fluid and blend from one to another. In my mind I can see perfectly how the character's underlying skeletal structure is bending and moving to match every one of his moves. The character's mesh conforms to the bones perfectly with every nuance shown, from the ripples of his muscles to the creases in his camouflage suit.

You can create these animation features (and much more) using what's known as skeletal animation−quite possibly the most awesome animation technique you can use in your projects. This technique is definitely easier to work with than it first appears. With games such as *Tom Clancy's Splinter Cell* showing the world just what skeletal animation can do, this is one topic you definitely don't want to skip. This chapter will show you how to get started. From working with skeletal structures to dealing with skinned meshes, it's all here.

Taking on Skeletal Animation

Skeletal animation−two words that bring to mind thoughts of B−rate horror movies in which the dead have risen from the grave to stalk the living. However, those two words mean something entirely different to programmers. If you're like me, this topic gives you more tingles down your spine than any cheesy horror movie ever could.

Skeletal animation is quickly becoming the animation technique of choice for programmers because it is quick to process and it produces incredible results. You can animate every detail of a character using skeletal animation. It gives you control of every aspect of the character's body, from the wrinkles in his skin to the bulges in his muscles. You can use every joint, bone, and muscle to deform the shape of your character's meshes.

Think of skeletal animation like this: Your body (or at least your skin) is a mesh, complete with an underlying set of bones. As your muscles push, pull, and twist your bones, your body changes shape to match. Instead of thinking of the muscles changing the shape of your body, think of the bones altering the rotation of each body part.

If you lift your arm your shoulder rotates, which in turn causes your entire arm to rotate and your skin to change shape. Your body (the mesh) changes shape to accommodate the changes in the bones. Skeletal animation works the same way. As the underlying skeletal structure changes orientation from the rotating of the joints, the overlaid mesh (appropriately called a *skinned mesh*) changes form to match.

As you can see, there are two separate entities to deal with when you are working with skeletal animation−the skeletal structure and the skinned mesh. Take a closer look at each entity in more detail to see how they work in unison, starting with the skeletal structure.

Using Skeletal Structures and Bone Hierarchies

The skeletal structure, as you can imagine, is a series of connected bones that form a hierarchy (a *bone hierarchy*, to be exact). One bone, called the *root bone*, forms the pivotal point for the entire skeletal structure. All other bones are attached to the root bone, either as child or sibling bones.

The word "bone" refers to a frame−of−reference object (a frame object, which is represented in DirectX by the `D3DXFRAME` structure or a `Frame` template inside .X files). If you were to examine the `D3DXFRAME` structure, you would indeed find the linked list pointers (`D3DXFRAME::pFrameSibling` and `D3DXFRAME::pFrameFirstChild`) that form the hierarchy. The `pFrameSibling` pointer links one bone to another on the same level in the hierarchy, whereas the `pFrameFirstChild` pointer links one bone to another as a child bone, which is one level lower in the hierarchy.

Generally, you would use a 3D−modeling package to create these skeletal structures for your projects. Exporting the bone hierarchy in the form of an .X file is a perfect example. Microsoft has released exporters for 3D Studio Max and Maya that allow you to export skeletal and animation data into .X files, and many modeling programs have the same exporting capabilities. I'll assume you have a program that will export these hierarchies to an .X file for you.

You'll find a number of things inside an .X file that contains skeletal animation data. First (and most important at this point), you'll find a hierarchy of `Frame` templates, which is your bone hierarchy in disguise. If you were to take a simple skeletal structure like the one shown in Figure 4.1, you'd have the resulting frame hierarchy detailed in the same figure.

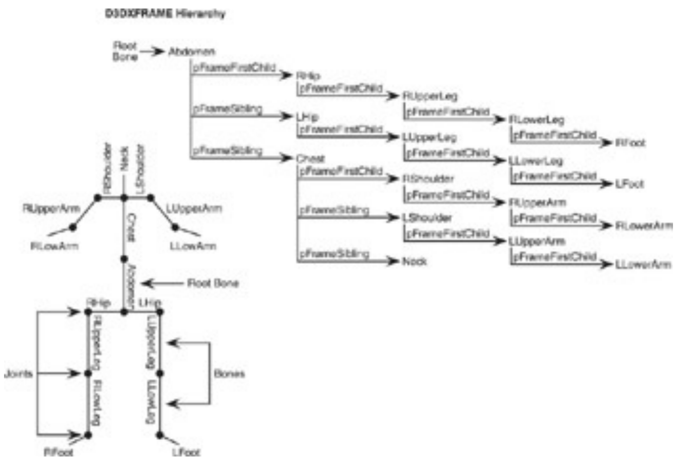


Figure 4.1: The skeletal structure on the left is represented by the hierarchy on the right. Notice the usage of the `D3DXFRAME` pointers to form a linked list of sibling and child frames.

You should find a standard `Mesh` data object embedded in the `Frame` data object hierarchy. The `Mesh` data object contains information about your skeletal animation object and the bones used in your skeletal structure. That's right−the `Frame` data object and the `Mesh` object both contain information about your skeletal structure! Whereas the `Frame` objects define the actual hierarchy, the `Mesh` object defines which frames represent the bones.

For now, however, the importance of the bone data is irrelevant. Because the bones depend on the frame hierarchy, it's important to concentrate solely on the frames at this point. You simply need to load the hierarchy (from an .X file, for example) and set it up for later use. Read on to see how to load hierarchies from .X.

Loading Hierarchies from .X

Not to beat a dead horse (why would I do a horrible thing like that?), but I want to quickly review how to load a frame hierarchy from an .X file. Even though Chapter 3 detailed using .X files and loading frame hierarchies, I want to go over the topics here using specific structures to contain the hierarchy.

For your frame hierarchy you should use the `D3DXFRAME` structure (or the `D3DXFRAME_EX` structure shown in Chapter 1). As I mentioned earlier in this chapter, the `D3DXFRAME` structure (or the derived `D3DXFRAME_EX` structure) contains two pointers that you use to create the frame hierarchy−`pFrameSibling` and `pFrameFirstChild`. Your job is to link each frame you load from an .X file using those two pointers.

Starting with a root frame object, begin iterating every data object from an .X file you specify. When you encounter a `Frame` object, link it either as a sibling or a child of the previous frame. Continue through the .X file until you have loaded all frames. For this example, use the `D3DXFRAME_EX` structure to contain the frames in the hierarchy.

Since Chapter 3 contained much more information on parsing .X files, I'm going to take it easy on the explanatory text for this portion. Basically, you'll open an .X file for access, and then iterate every data object in the file. For each `Frame` object you find, you need to create a matching `D3DXFRAME` (or `D3DXFRAME_EX`) object and link it to a hierarchy of frames.

To process an .X file, you can construct a class to handle the majority of the work for you. You simply instance the class's `ParseObject` function, which gives you access to each data object's data. Again, Chapter 3 explained the use of this class in full detail, so I'll leave it at that.

For now, take a look at the `ParseObject` function that is called for each data object that is enumerated.

```
BOOL cXFrameParser::ParseObject(
                         IDirectXFileData *pDataObj,
                         IDirectXFileData *pParentDataObj,
                         DWORD Depth,
                         void **Data, BOOL Reference)
{
   const GUID *Type = GetObjectGUID(pDataObj);

   // If the object type is a Frame (non−referenced),
   // then add that frame to the hierarchy.
   if(*Type == TID_D3DRMFrame && Reference == FALSE) {

      // Allocate a frame container
      D3DXFRAME_EX *pFrame = new D3DXFRAME_EX();

      // Get the frame's name (if any)
      pFrame−>Name = GetObjectName(pDataObj);

      // Link frame into hierarchy
      if(Data == NULL) {
         // Link as sibling of root
         pFrame−>pFrameSibling = m_RootFrame;
         m_RootFrame = pFrame; pFrame = NULL;
         Data = (void**)&m_RootFrame;
      } else {
         // Link as child of supplied frame
         D3DXFRAME_EX *pFramePtr = (D3DXFRAME_EX*)*Data;
         pFrame->pFrameSibling = pFramePtr->pFrameFirstChild;
         pFramePtr->pFrameFirstChild = pFrame; pFrame = NULL;
         Data = (void**)&pFramePtr->pFrameFirstChild;
      }
   }

   // Load a frame transformation matrix
   if(*Type==TID_D3DRMFrameTransformMatrix && Reference==FALSE) {
      D3DXFRAME_EX *Frame = (D3DXFRAME_EX*)*Data;
      if(Frame) {
         Frame->TransformationMatrix = *(D3DXMATRIX*)          \
                                  GetObjectData(pDataObj, NULL);
         Frame->matOriginal = Frame->TransformationMatrix;
      }
   }

   // Parse child objects
   return ParseChildObjects(pDataObj, Depth, Data, Reference);
}
```

If you haven't read Chapter 3 yet (shame on you if you didn't!), some of the preceding code is going to look a little confusing. Basically, the `ParseObject` function is called for each data object that is enumerated. Inside the `ParseObject` function, you check the currently enumerated object's type (using the object's template GUID). If that type is a `Frame`, then you allocate a frame structure and load the frame's name into it.

Next, you link the frame into the hierarchy of frames, which is where things look a little strange. The `cXFrameParser` class maintains two pointers−one for the root frame object that is being built up (`m_RootFrame`, a member of the class), and one for a data object (`Data`) that is passed to each call of `ParseObject`. The data pointer keeps track of the last frame data object that was loaded.

As you begin parsing the .X file, the data pointer `Data` is set to NULL, meaning that it doesn't point to any frame object being loaded. When you load a frame object into a frame structure, you are checking that data pointer to see whether it points to another frame structure. If it doesn't, it is assumed that the current frame is a sibling of the root. If the data pointer does point to another frame, it is assumed that the currently enumerated frame is a child of the frame to which the data pointer points.

Knowing whether the currently enumerated frame is a sibling or a child is a factor when you are creating the hierarchy. Sibling frames are linked to each other using the `pFrameSibling` pointer of the `D3DXFRAME` structure, whereas child frames are linked using `pFrameFirstChild`. Once a frame has been loaded, the data pointer is adjusted to point at the new frame or back to the sibling frame. In the end, all frames become linked either as siblings or children.

One more thing that you'll notice in the `ParseObject` function is the code to load a frame's transformation matrix (represented by the `FrameTransformMatrix` template). A `FrameTransformMatrix` object is typically embedded in a `Frame` data object. This `FrameTransformMatrix` object defines the initial orientation of the `Frame` being loaded.

For skeletal animation, this frame transformation matrix defines the initial pose of your skeletal structure. For example, a standard skeletal structure might be posed with the body standing erect and the arms extended. However, suppose all of your animations are based on the character standing in a different pose, perhaps with his arms dropped down to his sides and with his legs slightly bent. Instead of reorienting all the vertices or bones to match that pose before saving the .X file in your 3D modeling program, you can change the frame transformations. From that point forward, all motions of the bones will be relative to that pose. This becomes more apparent as you try to manipulate the bone orientations and during animation, so I'll leave the topic alone for the moment. Just know that inside each frame structure you are loading, there is space to store an initial transformation matrix (in the `D3DXFRAME::TransformationMatrix` object).

After all is said and done, your frame hierarchy will be loaded. Of course, the root frame is stored in the `m_RootFrame` linked list of `D3DXFRAME_EX` objects inside the frame−loading class. It's your job to grab that pointer and assign it to one you'll use in your program. After you've done that, you can start messing around with the orientation of the bones.

Modifying Bone Orientation

After you have loaded the bone hierarchy, you can manipulate it. To modify the orientation of a bone, you first need to locate its respective frame structure by creating a function that recursively searches the frames for a specific bone name. Once it is found, a pointer to the frame is provided so you can directly access the frame's transformation matrix. The recursive search function might look something like this:

Note You can apply any transformation to any bone in the hierarchy, but it's recommended that you only work with rotations. Why only rotations? Think of it this way−when you bend your elbow, it rotates. How would you explain it if you translated your elbow instead? That would make your elbow pop off your arm−something you definitely don't want!

If you are trying to move the entire mesh through the world, just translate the root bone; all other bones will inherit that transformation. Better yet, use the world transformation to move the skinned mesh object through the 3D world.

```
D3DXFRAME_EX *FindFrame(D3DXFRAME_EX *Frame, char *Name)
{
   // Only match non−NULL names
   if(Frame && Frame−>Name && Name) {
      // Return frame pointer if matching name found
      if(!strcmp(Frame−>Name, Name))
         return Frame;
   }

   // Try to find matching name in sibling frames
   if(Frame && Frame−>pFrameSibling) {
      D3DXFRAME_EX *FramePtr =                                \
             FindFrame((D3DXFRAME_EX*)Frame−>pFrameSibling,   \
                       Name);
      if(FramePtr)
         return FramePtr;
   }

   // Try to find matching name in child frames
   if(Frame && Frame−>pFrameFirstChild) {
      D3DXFRAME_EX *FramePtr =                                 \
             FindFrame((D3DXFRAME_EX*)Frame−>pFrameFirstChild, \
                       Name);
      if(FramePtr)
         return FramePtr;
   }

   // No matches found, return NULL
   return NULL;
}
```

Suppose you want to find a bone called "Leg" using the `FindFrame` function. You simply provide the name of the bone to find and a pointer to your root frame, as shown here:

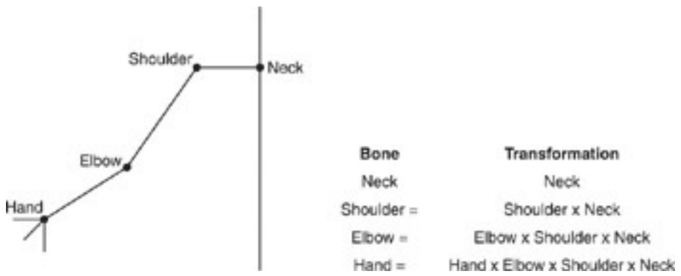
```
// pRootframe = D3DXFRAME_EX root frame pointer
D3DXFRAME_EX *Frame = FindFrame(pRootFrame, "Leg");
if(Frame) {
   // Do something with frame, like changing the
   // D3DXFRAME_EX::TransformationMatrix to something
   // you want. For here, let's rotate the bone a little
   D3DXMatrixRotationY(&Frame->TransformationMatrix, 1.57f);
}
```

Updating the Hierarchy

Once you've modified the bone transformations, you need to update the entire hierarchy so you can use it later for rendering. Even if you haven't modified the bone transformations, you still need to update the hierarchy because you need to set certain variables before rendering.

During the hierarchy update, you must combine each successive transformation down through the hierarchy. Starting at the root, you apply the bone's transformation matrix to the frame's combined transformation matrix. The bone's transformation matrix is passed to any siblings of the root to be combined as well. From there, the combined transformation matrix you just calculated is passed to each child of the root. This process propagates itself throughout the hierarchy.

Although it is hard to understand at first, you can think of the process this way: Take the skeletal structure in Figure 4.2, start at the root, and multiply it by a transformation matrix that positions the root in the world.



Bone	Transformation
Neck	Neck
Shoulder =	Shoulder x Neck
Elbow =	Elbow x Shoulder x Neck
Hand =	Hand x Elbow x Shoulder x Neck

Figure 4.2: The simple skeletal structure on the left uses the bone transformations on the right to orient the frames.

As you can see in Figure 4.2, the combined transformation from the root is passed to all of its child bones, which in turn are combined. The results are passed to the child bones of those bones. However, trying to compute the transformation matrices in the manner shown is very difficult, so other means are necessary.

The easiest way to update your frame hierarchy is to create a recursive function that combines the frame's transformation with a provided transformation matrix. From there, the transformation matrix is passed to the frame's siblings, and the combined matrix is passed to the frame's child frames. Take a look at the function in question.

```
void UpdateHierarchy(D3DXFRAME_EX *Frame,                    \
                     D3DXMATRIX matTransformation = NULL)
{
   D3DXFRAME_EX *pFramePtr;
   D3DXMATRIX matIdentity;

   // Use an identity matrix if none passed
   if(!matTransformation) {
      D3DXMatrixIdentity(&matIdentity);
      matTransformation = &matIdentity;
   }

   // Combine matrices w/supplied transformation matrix
   matCombined = TransformationMatrix * (*matTransformation);

   // Combine w/sibling frames
   if((pFramePtr = (D3DXFRAME_EX*)pFrameSibling))
      pFramePtr−>UpdateHierarchy(matTransformation);

   // Combine w/child frames
   if((pFramePtr = (D3DXFRAME_EX*)pFrameFirstChild))
      pFramePtr−>UpdateHierarchy(&matCombined);
}
```

As you can see, the `UpdateHierarchy` function takes a `D3DXFRAME_EX` object as the first parameter−this is the current frame being processed. You only need to call `UpdateHierarchy` once, to provide a pointer to your root frame; the function will recursively call itself for each frame.

Notice the second parameter of `UpdateHierarchy−matTransformation`. The `matTransformation` parameter is the transformation matrix to apply to the frame's transformation. By default, the `matTransformation` pointer is NULL, meaning that an identity matrix is used during the call to `UpdateHierarchy`. After a frame's matrix is combined with the provided transformation, the resulting transformation is passed to the child frames by setting `matTransformation` during the next call.

Note If you've already read Chapter I (which you probably have), then you'll notice the `UpdateHierarchy` function has been encapsulated in the `D3DXFRAME_EX` class itself, so instead of calling `UpdateHierachy` with the root frame as the parameter, you can use the root frame's `::UpdateHierarchy` member function! Refer back to Chapter I for further information on this member function.

As I just mentioned, you only need to call the `UpdateHierarchy` function using your root frame. Don't provide a transformation matrix as the second parameter−this should be left up to the recursive calls. If you do provide a transformation matrix with the root frame, you'll be moving the entire mesh using that transformation matrix. That's the same as setting the world transformation matrix to position and orient the mesh to render.

```
// pRootFrame = D3DXFRAME_EX root frame object
UpdateHierarchy(pRootFrame);
```

Now that you have a little understanding of the skeletal structure and how to work with bone hierarchies, it's time to move on to the second piece of the animation puzzle−the overlaid skinned mesh that deforms to match the orientation of the bone hierarchy.

Working with Skinned Meshes

In the first half of this chapter, you learned how to manipulate a hierarchy of bones that forms the basis of skeletal animation. That's all fine and dandy, but playing with imaginary bones isn't going to cut the mustard. Your game's players need to see all your hard work in the form of rendered meshes, which is where skinned meshes come in.

Skinned meshes are almost like the standard meshes with which you are already familiar. Using a `D3DXMESHCONTAINER_EX` object (as you saw in Chapter 1), you can store your mesh's data, from the vertices and indices to the materials and texture data, all wrapped up in one convenient `ID3DXMesh` object. As for the actual skinned mesh data, that information is contained in a special object called `ID3DXSkinInfo`.

I'll skip the `ID3DXSkinInfo` introductions for the moment and instead explain what makes a skinned mesh unique to other meshes. A skinned mesh deforms to match the orientation of the underlying skeletal structure. As the bones twist and turn, so do the mesh's vertices. The mesh−s vertices make the skinned mesh unique. You'll be dealing with the changing positions of the vertices when it comes to skinned meshes.

Take a look at Figure 4.3, which shows a skeleton surrounded by a simplistic mesh.

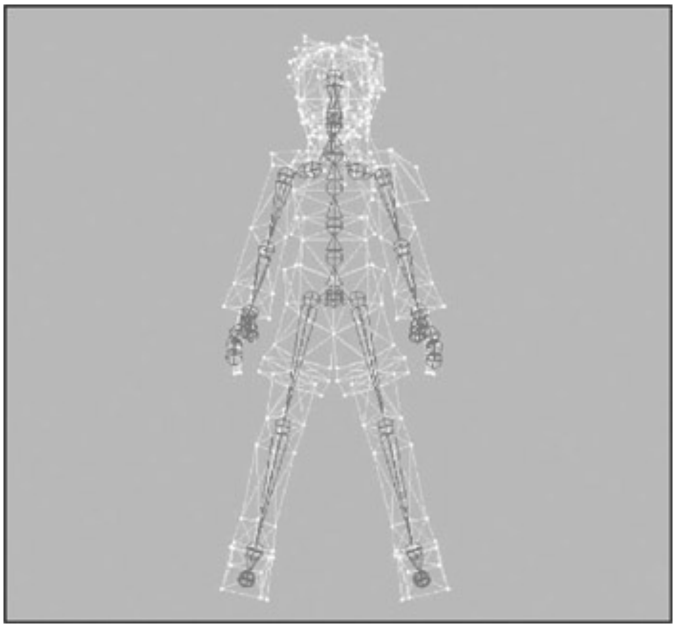


Figure 4.3: When connected to a skeletal structure, a mesh is mapped in such a way that each vertex is connected to the bones.

In Figure 4.3, each vertex is connected to a bone. As a bone moves, so do the vertices that are attached to it. For example, if you were to rotate the bone 45 degrees about the x−axis, the attached vertices would rotate 45 degrees as well, with the bone's joint acting as the pivot point or the origin of the rotation.

Now take a closer look at Figure 4.3, and you'll see that a couple vertices are attached to more than one bone. That's right−you're not limited to attaching a vertex to a single bone. In fact, you can connect a vertex to as many bones as you want with DirectX, using the methods you learn in this book. Whenever one of the bones to which the vertex is attached moves, the vertex inherits a percentage of the motion. For example, if a bone rotates 60 degrees about the z−axis, an attached vertex may inherit only 25 percent of the motion, meaning the vertex will rotate only 15 degrees about the z−axis.

The exact percentage of motion the vertex inherits is called the *vertex weight*. Each vertex in the skinned mesh is assigned one vertex weight per bone to which it is attached. Those weights are typically 1.0 for vertices that are attached to only one bone, meaning that the vertex inherits the full motion of the bone. The weights are divided among the bones for vertices attached to multiple bones, and are usually calculated by taking into consideration the vertex's distance from each bone. (Most 3D modeling programs will graciously calculate this for you.) For example, suppose a vertex is attached to two bones, meaning that both weights are set to 0.5. The vertex will inherit 50 percent of the motion from each bone. Notice that the total of all weights summed must always equal 1.

The purpose of using skin weights is quite ingenious. By allowing certain bones to influence specific vertices, you can have awesome effects such as wrinkling skin, bulging muscles, and stretching clothes−all in real time as your characters animate!

The way DirectX treats the vertex weights is quite simple. After you've loaded a mesh to use as your skinned mesh and you've loaded the vertex weights (also called *skin weights*), you can transform the vertices to match the bones' orientations using the following steps.

1. Iterate all vertices. For each vertex, proceed to Step 2.
2. For each bone to which the current vertex is connected, get the bone transformation.
3. For each bone transformation, multiply the matrix by the vertex's weight and apply the result to a combined transformation for the vertex.
4. Repeat Step 3 for each bone connected, and repeat Steps 2 through 4 for each vertex. When you're finished, apply the combined transformation matrix to the specific vertex being iterated (from Step 1).

How exactly do you obtain these skin weights? With the help of the `ID3DXSkinInfo` object I mentioned earlier, you can load the weights from an .X file. The skin weights are stored within a `Mesh` data object, usually at the end of the `Mesh` object's data.

For each bone in your skeletal structure, there is a matching `SkinWeights` data object. Inside the `SkinWeights` object is the name of the bone, followed by a number of vertices attached to it. A skinned mesh header determines the number of bones to which each vertex in the mesh can be connected. If some of the vertices are attached to two bones, then all vertices must be attached to two bones. To get around the oddity of having vertices that connect to different numbers of bones, you can assign a weight of 0 to the second bone.

As I mentioned, the `SkinWeights` object includes the number of vertices that are connected to the bone. It lists an array of vertex index numbers. After the array of vertex indices, there is an array of vertex weight values. Finally, there is an inversed bone transformation to help you orient the vertices around the bone's joint.

Take a look at this sample `SkinWeights` template data object:

```
SkinWeights {
   "Bip01_R_UpperArm";
   4;
   0, 3449,  3429, 1738;
   0.605239,  0.605239, 0.605239, 0.979129;
   −0.941743,  −0.646748,  0.574719, 0.000000,
   −0.283133,  −0.461979, −0.983825, 0.000000,
    0.923060,  −1.114919,  0.257891, 0.000000,
   −65.499557, 30.497688, 12.852692, 1.000000;;
}
```

In this data object, a bone called `Bip01_R_UpperArm` is used. There are four vertices attached to the bone, and the vertex indices are `0, 3449, 3429,` and `1738`. Vertex 0 has a weight of 0.605239, vertex 1 has a weight of 0.605239, and so on. A transformation matrix aligns the vertices listed around the bone's joint. This matrix is very important. Without it, the vertices will rotate around the origin of the world instead of the bone's joint.

Thankfully, you don't have to deal directly with the `SkinWeights` templates. The data is handled for you while you are loading the skinned mesh from an .X file using the D3DX helper functions.

Loading Skinned Meshes from .X

Loading a skinned mesh from an .X file is much like loading a standard mesh. Using a custom .X parser, you must enumerate your .X file objects using `ParseObject`. When it comes to processing a `Mesh` object, instead of calling the `D3DXLoadMeshFromXof` function to load the mesh data, you call the `D3DXLoadSkinMeshFromXof` function, which takes one additional parameter−a pointer to an `ID3DXSkinInfo` object. Check out the `D3DXLoadSkinMeshFromXof` prototype to see what I mean.

```
HRESULT D3DXLoadSkinMeshFromXof(
   IDirectXFileData *pXofObjMesh,      // .X file data object
   DWORD Options,                      // Load options
   IDirect3DDevice9 *pDevice,          // 3−D device in use
   ID3DXBuffer **ppAdjacency,          // Adjacency buffer object
   ID3DXBuffer **ppMaterials,          // Material buffer object
   ID3DXBuffer **ppEffectInstances,    // Effects instance object
   DWORD *pMatOut,                     // # of materials
   ID3DXSkinInfo **ppSkinInfo,         // SKIN INFO OBJECT!!!
   ID3DXMesh **ppMesh);                // Loaded mesh object
```

When you are ready to load a mesh from an enumerated `Mesh` template, call the `D3DXLoadSkinMeshFromXof` function instead of calling `D3DXLoadMeshFromXof`. Make sure to supply an `ID3DXSkinInfo` object where it is shown in the prototype. Whether or not the `Mesh` template contains a skinned mesh doesn't matter−the `D3DXLoadSkinMeshFromXof` function will load regular and skinned meshes without a hitch. Here's an example:

```
// Define the mesh and skinned mesh info objects
ID3DXMesh     *pMesh;
ID3DXSkinInfo *pSkinInfo;

// Define buffers to hold the material data and adjacency data
ID3DXBuffer *pMaterialBuffer = NULL, *pAdjacencyBuffer = NULL;

// DWORD to hold the number of materials being loaded DWORD NumMaterials;

// Load the skinned mesh from IDirectXFileDataObject pDataObj
D3DXLoadSkinMeshFromXof(pDataObj, D3DXMESH_SYSTEMMEM,       \
                        pDevice, &pAdjacencyBuffer,         \
                        &pMaterialBuffer, NULL,             \
                        &NumMaterials, &pSkinInfo, &pMesh);
```

Just because you used the `D3DXLoadSkinnedMeshFromXof` function, that doesn't mean a skinned mesh was loaded. First you need to check the `pSkinInfo` object. If it's set to NULL, then a skinned mesh wasn't loaded. If it's a valid object (non−NULL), then you need to check whether any bones exist.

The easiest way to see whether bones exist is to call `ID3DXSkinInfo::GetNumBones`. The `GetNumBones` function will return the number of bones loaded from the `Mesh` template. If the number is 0, then there are no bones, and you can free the `ID3DXSkinInfo` object (using `Release`). If bones do exist, then you can continue using the skinned mesh.

Check out this example, which tests whether a skinned mesh was loaded. If so, the example checks to see whether the mesh contains any bones.

```
// Set a flag is there's a skinned mesh and bones to use
```

```
BOOL SkinnedMesh = FALSE;
if(pSkinInfo && pSkinInfo->GetNumBones())
   SkinnedMesh = TRUE;
else {
   // Free the skinned mesh info data object
   if(pSkinInfo) {
      pSkinInfo->Release();
      pSkinInfo = NULL;
   }
}
```

If the `SkinnedMesh` flag is set to `TRUE`, then the `pSkinInfo` object is valid and you're ready to work with the skinned mesh. The next step is to create another mesh object that will contain the actual deforming mesh as you change the bones' orientations.

Creating a Secondary Mesh Container

After you create the skinned mesh, you need to create a second mesh container. Why, you ask? Well, the skinned mesh object you loaded from the `D3DXLoadSkinMeshFromXof` function is sort of the base of reference for your mesh's vertex data. Since these vertices are in the right positions to match the orientations of the bones, it would mess up things quite a bit if you started altering those positions.

Let's leave things well enough alone and instead create a second mesh object (an `ID3DXMesh` object) that contains an exact duplicate of the skinned mesh. You need to read the vertex data from the skinned mesh data, apply the various bone transformations, and write the resulting vertex data to this duplicate mesh container (which I call the *secondary mesh* or *secondary mesh container*) that you use to render. Makes sense, doesn't it?

As I mentioned, the secondary mesh is an identical match to the skinned mesh; everything from the number of vertices to the indices needed is the same. The easiest way to duplicate the skinned mesh object is to use the `ID3DXMesh::CloseMeshFVF` function.

```
HRESULT ID3DXMesh::CloneMeshFVF(
   DWORD Options,              // Mesh attribute flags
   DWORD FVF,                  // FVF to use for cloning
   IDirect3DDevice9 *pDevice,  // 3−D device in use
   ID3DXMesh *ppCloneMesh);    // Secondary mesh object
```

The `Options` parameter of `CloneMeshFVF` is just like the one from the calls to `D3DXLoadMeshFromX`, `D3DXLoadMeshFromXof`, and `D3DXLoadSkinMeshFromXof`, so take your pick. I tend to set `Options` flags to 0, but feel free to change it.

As for the `FVF` parameter, you only need to supply the `FVF` from the skinned mesh object using the skinned mesh's `GetFVF` function. Also, don't forget to supply the valid `IDirect3DDevice9` object you are using, as well as a pointer to an `ID3DXMesh` object that will be your secondary mesh container.

Here's a bit of code that demonstrates cloning a skinned mesh to create your secondary mesh:

```
// pSkinMesh = ID3DXMesh object
ID3DXMesh *pMesh; // Secondary mesh container
pSkinMesh->CloneMeshFVF(0, pMesh->GetFVF(), pDevice, &pMesh);
```

All this talk of cloning reminds me of *Star Wars Episode II: Attack of the Clones*. Good thing your cloned secondary meshes aren't going to try to take over the universeor are they? Well heck, those clones aren't going anywhere without a little effort, so let's get back to work and see what's next in line.

After you've created the secondary mesh container, it's time to map your bones to the frame hierarchy. Why didn't we do this previously, when I was discussing bones and frames? Easy−the bone data was loaded until you called `D3DXLoadSkinMeshFromXof`!

Mapping Bones to Frames

If you peruse an .X file, you might notice some similarities between the `Frame` data objects and the `SkinWeights` objects. For every bone in your skeletal structure, there is a matching `SkinWeights` object embedded inside a `Mesh` object that contains the name of a `Frame` object (or a reference to a `Frame` object). That's right−each bone is named after its corresponding `Frame` data object!

After you load your skinned mesh, you need to connect each bone to its corresponding frame. This is simply a matter of iterating all bones, getting the name of each, and searching the list of frames for a match. Each matching frame pointer is stored in a special bone structure of your design.

For this book, I embedded the bone−mapping data in the `D3DXMESHCONTAINER_EX` structure detailed in Chapter 1. The `D3DXMESHCONTAINER_EX` structure (created especially for this book and detailed as follows) adds an array of texture objects, a secondary mesh container object, and the bone−mapping data to the `D3DXMESHCONTAINER` structure.

```
struct D3DXMESHCONTAINER_EX : D3DXMESHCONTAINER
{
   IDirect3DTexture9 **pTextures;
   ID3DXMesh          *pSkinMesh;

   D3DXMATRIX        **ppFrameMatrices;
   D3DXMATRIX         *pBoneMatrices;

   // .. extra data and functions to follow
};
```

For this chapter, the important variables are `ppFrameMatrices` and `pBoneMatrices`. The `pBoneMatrices` array contains the transformations from your bone hierarchy; one transformation matrix is applied to each vertex belonging to the appropriate bone. The only problem is, the transformations from your bones are not stored in an array; they're stored as a hodgepodge of single transformations spread throughout the hierarchy.

The `D3DXMESHCONTAINER_EX` structure provides a pointer to each bone transformation matrix contained within the hierarchy of `D3DXFRAME_EX` objects inside an array of pointers (`ppFrameMatrices`). Using these pointers, you can pull each bone transformation and place it into the `pBoneMatrices` array you'll create and use during the call to update your skinned mesh.

You can create the array of pointers and the array of matrices after you load the bone hierarchy by taking the number of bones from the hierarchy and allocating an array of `D3DXMATRIX` pointers and `D3DXMATRIX` objects, like this:

```
// pSkinInfo = skinned mesh object

// Get the number of bones in the hierarchy
DWORD NumBones = pSkinInfo−>GetNumBones();
```

```
// Allocate an array of D3DXMATRIX pointers to point
// to each bones' transformation.
D3DXMATRIX *ppFrameMatrices = new D3DXMATRIX*[NumBones];

// Allocate an array of D3DXMATRIX matrix objects
// to contain the actual transformations used to update
// the skinned mesh.
D3DXMATRIX *pBoneMatrices = new D3DXMATRIX[NumBones];
```

After you load your skinned mesh, you can set up the pointers to each bone transformation by querying the skinned mesh info object for each bone name. Using that, you can scan the list of frames for a match. For each matched bone, set the pointer to that frame's transformation matrix. When all bones and frames are matched up, you can then iterate the entire list and copy the matrices to the `pBoneMatrices` array.

First let me show you how to match up the bones and frames. Remember that earlier in this chapter I mentioned that the bones are named after the frames. Using the `ID3DXSkinInfo::GetBoneName` function, you can obtain the name of the bone and frame to match.

```
// Go through each bone and grab the name of each to work with
for(DWORD i=0;i<pSkinInfo->GetNumBones();i++) {

   // Get the bone name
   const char *BoneName = pSkinInfo->GetBoneName(i);
```

When you have the bone's name, you can scan through the list of frames in the hierarchy to look for a match. To do so, you use the recursive `FindFrame` function developed in the "Modifying Bone Orientation" section earlier in this chapter, as follows.

```
  // pRootFrame = D3DXFAME_EX root frame object

  // Find matching name in frames
  D3DXFRAME_EX *pFrame = pRootFrame->Find(BoneName);
```

If a frame with the name provided by the bone is found, you can link to the frame's combined transformation matrix. If no match is found, then the link is set to NULL.

```
   // Match frame to bone
   if(pFrame)
      pMesh->ppFrameMatrices[i] = &pFrame->matCombined;
   else
      pMesh->ppFrameMatrices[i] = NULL;
}
```

You might not understand the exact reasons for mapping the bones to the frame at this moment, but it will make more sense when you get into manipulating the skinned mesh and rebuilding the mesh to render it. For now, take each step in stride, and start by learning how to manipulate the skinned mesh.

Manipulating the Skinned Mesh

Now nothing is stopping you from twisting up that skeletal structure and going crazy. Just make sure it's your mesh's imaginary skeletal structure you're manipulating and not your own−I just hate it when I accidentally manipulate my bones into a pose I can't get out of for an hour! Kidding aside, you can now alter the frame orientations in your frame hierarchy. It's those frames that represent your bones.

Speaking of altering the frame orientations, be aware that you should only rotate your bones; you should never translate them. Scaling is acceptable, but be careful−remember that all transformations propagate throughout the hierarchy. If you were to scale your character's upper arm, the lower arm would be scaled as well.

I covered changing the orientations of the various bones earlier in this chapter, in the "Modifying Bone Orientation" section, so I won't rehash anything here. After you've loaded the skeletal structure and skinned mesh, feel free to start working with the bone transformations using those techniques covered earlier. When you're ready, you can update the skinned mesh and prepare it for rendering.

Updating the Skinned Mesh

When your skeletal structure is in the pose you desire, it's time to update (or rebuild) the skinned mesh to match. Before you rebuild the skinned mesh, you must make sure you have constructed the secondary mesh container and updated the frame hierarchy. To review how to construct the mesh container, consult the "Creating a Secondary Mesh Container" section earlier in this chapter. To refresh your memory about how to update the frame hierarchy, review the "Updating the Hierarchy" section earlier in this chapter. After you're sure of these two things, you can continue.

To update the skinned mesh, you must first lock the vertex buffers of the skinned mesh and the secondary mesh. This is critical because DirectX will pull the vertex data from the skinned mesh object, apply the bone transformations, and write the resulting vertex data to the secondary mesh object.

First, though, you need to copy the transformations from the frames to the array of matrices (`pBoneMatrices`) stored in the mesh container. At the same time, you have to combine the transformations with the bones' inversed transformations. The inversed bone transformations are responsible for moving the mesh's vertices to the origin of the mesh before you apply the actual transformation. To better understand this, take a look at Figure 4.4

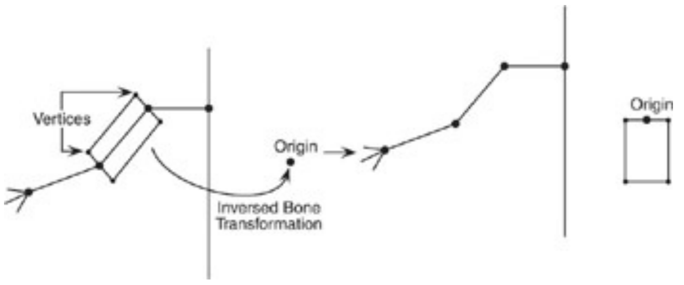


Figure 4.4: Vertices only rotate around their source mesh's origin. Before you apply the bone transformation, you must build a transformation that orients the vertices around the mesh's origin.
The mesh in Figure 4.4 is composed of three bones (frames) and a number of vertices. To apply a transformation to any frame, you must move the vertices belonging to the frame to the origin and then apply the transformations.

You move the vertices around the origin of the mesh before you apply a transformation because a rotation matrix simply rotates vertices around an origin. If you were to rotate a vertex belonging to any bone, the vertex would rotate around the origin of the mesh instead of the bone's joint. For example, if your body was a mesh and you bent your elbow, the vertices constructing your arm's mesh would rotate around your elbow, not the center of your body. After the vertices are moved to the center of the mesh, the transformation is applied (thus rotating the vertices to match the rotation of the bone) and finally translated into position.

Normally, these inversed bone transformations are stored in the .X file by the 3D modeler used to create the meshes. If you don't have access to this information from an .X file, you can compute it yourself by first updating the frame hierarchy, and then inverting each frame's combined transformation using the `D3DXMatrixInverse` function. Here's a quick example.

```
// pRoot = root D3DXFRAME_EX object
// pMesh = D3DXMESHCONTAINER_EX object w/mesh data

// Update the frame hierarchy
pRoot->UpdateHierarchy();

// Go through each bone and calculate the inverse
for(DWORD i=0;i<NumBones;i++) {
   // Grab the transformation using the bone matrix
   D3DXMATRIX matBone = (*pMesh->ppFrameMatrices);
   // Invert the matrix
   D3DXMatrixInverse(&matBone, NULL, &matBone);

   // Store the inversed bone transformation somewhere
}
```

Instead of going through all the trouble of calculating the inversed bone transformations yourself, however, you can rely on the skinned mesh object to supply that information. By calling `ID3DXSkinInfo::GetBoneOffsetMatrix`, you'll get the inversed bone transformation matrix pointer. Multiply this matrix by a frame transformation matrix, and you're set!

Using what you just learned, iterate through all the bones, grab the inversed bone transformation, combine it with the frame transformation, and store the result in the `pBoneMatrices` array.

```
for(DWORD i=0;i<pSkinInfo->GetNumBones();i++) {

   // Set the inversed bone transformation
   pMesh->pBoneMatrices[i]=(*pSkinInfo->GetBoneOffsetMatrix(i));

   // Apply frame transformation
   if(pMesh->ppFrameMatrices[i])
     pMesh->pBoneMatrices[i] *= (*pMesh->ppFrameMatrices[i]);
}
```

Now that you've copied the bones' transformations into the `pBoneMatrices` array, you can move on to updating the skinned mesh by first locking the vertex buffers for the skinned mesh and the secondary mesh.

```
// pSkinMesh = skinned mesh container
// pMesh = secondary mesh container

// Lock the meshes' vertex buffers
void *SrcPtr, *DestPtr; pSkinMesh->LockVertexBuffer(D3DLOCK_READONLY,(void**)&SrcPtr);
pMesh->LockVertexBuffer(0, (void**)&DestPtr);
```

After you lock the vertex buffers, you need to perform a call to `ID3DXSkinInfo::UpdateSkinnedMesh` to apply all the bones' transformations to the vertices and write the resulting data to the secondary mesh container. To finish, you simply unlock the vertex buffers, and you're ready to render!

```
// pSkinInfo = skinned mesh info object

// Update the skinned mesh using provided transformations
pSkinInfo->UpdateSkinnedMesh(pBoneMatrices, NULL,           \
                             SrcPtr, DestPtr);
```

```
// Unlock the meshes vertex buffers
pSkinMesh−>UnlockVertexBuffer();
pMesh−>UnlockVertexBuffer();
```

Rendering the Skinned Mesh

Now comes the good part−rendering your secondary mesh and showing the world what it's like to play with powerthe power of skeletal animation and skinned meshes, that is. You only need to depend on the typical mesh−rendering functions to render the secondary mesh. Loop through each material, set the material and texture, and call the `ID3DXMesh::DrawSubset` function. Loop and continue until all of the subsets have been drawn.

If you are using the `D3DXMESHCONTAINER_EX` object from Chapter 1, this code should work perfectly for you to render the secondary mesh.

```
// pMesh = D3DXMESHCONTAINER_EX object w/material data
// pMeshToDraw = secondary mesh pointer to render
for(DWORD i=0;i<pMesh−>NumMaterials;i++) {

   // Set material and texture
   pD3DDevice−>SetMaterial(&pMesh−>pMaterials[i].MatD3D);
   pD3DDevice−>SetTexture(0, pMesh−>pTextures[i]);

   // Draw the mesh subset
   pMeshToDraw−>DrawSubset(i);
}
```

That wraps up skeletal animation basics! In the next few chapters, you'll learn how to put your newfound skeletal animation techniques to work with pre−computer key−frame animations, animation blending, and rag doll animation techniques. Have fun!

Check Out the Demo

Not so fast, buster! You want to know about this chapter's demonstration program on the CD−ROM, don't you? As shown in Figure 4.5, the SkeletalAnim mesh demonstrates what you learned in this chapter by loading a skinned mesh (the Tiny.x mesh provided in the DirectX SDK samples) and rendering it to the display.

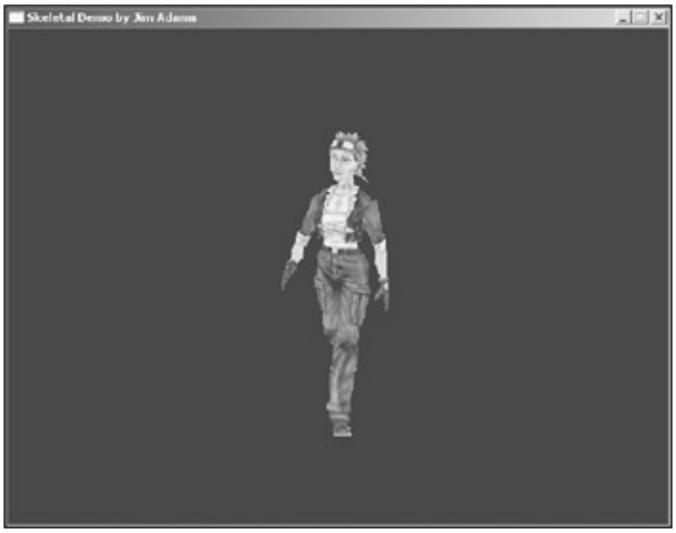


Figure 4.5: Meet Tiny, Microsoft's woman of skeletal meshes. She is constructed from a single mesh and an underlying hierarchy of invisible bones.

Programs on the CD

In the Chapter 4 directory for this book, you'll find a single project that demonstrates how to work with skeletal animation.

- **SkeletalAnim.** This demo program demonstrates the use of skeletal structures and skinned meshes. It is located at \BookCode\Chap04\SkeletalAnim.

Chapter 5: Using Key−Framed Skeletal Animation

Precalculated key−frame animations are the life−blood of today's game engines. With little work, an animator can design a complete animation sequence inside a popular 3D modeler and export the animation data to a format readily usable by the game engine. Without any extra effort, you can change or modify those animations without rewriting the game code.

I'm know you've seen quite a few of these games, and I know that you want to have the same ability to play these key−framed animations back in your own games. This chapter is just what you need!

Using Key−Framed Skeletal Animation Sets

If you have explored the DirectX SDK samples, you might have come across a little demo called SkinnedMesh, which shows you how to use a pre−calculated key−frame animation stored in an .X file to animate an on−screen character. The problem is, that sample's code is so convoluted and hard to understand that it'll make your head spin. With no real documentation of how to use .X file animation data, the skinned mesh animation sample remains full of mystery.

In Chapter 4, you saw how to work with skeletal−based meshes, called *skinned meshes*, which adhere to an underlying hierarchy of bones (a *skeletal structure*). The vertices of the skinned mesh are attached to those bones, and they move as the bones do. Basically, animation is achieved by slowly applying a set of transformations to the hierarchy of bones and letting the vertices follow along with the movements.

The sequence of these animation transformations is called a *key frame*. You use key frames to define both the transformation and the time of the transformation to use in the animation sequence. Where do you get the transformations you use to apply the movements to those bones? There are many file formats at your disposal, but to keep on the DirectX route, I'll be concentrating on using .X files.

If you take a look at the SkinnedMesh demo's .X file (Tiny.x) from the DirectX SDK, you'll notice that along with the typical `Frame` and `Mesh` templates, there is an `AnimationSet` template with a number of embedded `Animation` and `AnimationKey` objects. It's from these animation data objects that you obtain the transformations used to animate your skinned mesh's bone hierarchy. Take a closer look at some of these animation objects within an .X file to see what I mean.

```
AnimationSet Walk {
  Animation {
   {Bip01}
   AnimationKey {
    4;
    3;
       0; 16; 1.00000, 0.00000, 0.00000, 0.00000,
              0.00000, 1.00000, 0.00000, 0.00000,
              0.00000, 0.00000, 1.00000, 0.00000,
              0.00000, 0.00000, 0.00000, 1.00000;;,
    1000; 16; 1.00000, 0.00000, 0.00000, 0.00000,
              0.00000, 1.00000, 0.00000, 0.00000,
              0.00000, 0.00000, 1.00000, 0.00000,
              0.00000, 0.00000, 0.00000, 1.00000;;,
    2000; 16; 1.00000, 0.00000, 0.00000, 0.00000,
              0.00000, 1.00000, 0.00000, 0.00000,
              0.00000, 0.00000, 1.00000, 0.00000,
              0.00000, 0.00000, 0.00000, 1.00000;;;
   }
}
Animation {
   {Bip01_LeftArm}
    AnimationKey {
   0;
   1;
   0; 4; 1.00000, 0.000000, 0.00000, 0.000000;;;
  }
AnimationKey {
   1;
   1;
   0; 4; 1.000000, 1.00000, 1.000000;;;
  }
AnimationKey {
   2;
   1;
   0; 3; 0.000000, 0.00000, 0.000000;;;
   }
 }
}
```

What you're looking at is a simple animation that works with two bones. Each animation is defined inside an `AnimationSet` data object; in the previous instance, this animation has been assigned the name `Walk`. Two `Animation` objects that contain the various keys of animation for each bone are embedded in this `AnimationSet` object. Keys?! What the heck am I talking about? Well, my friend, let me take a moment to explain the concept of keys in animation.

Using Keys in Animation

A key, short for an *animation key*, is a timeline marker that signifies a change in a bone's position and/or orientation. An animation that uses keys is called a *key−framed animation*. The reasons for using keys are quite significant, with the most important one being memory conservation.

You see, an animation is a series of movements (bone movements, in this case) over a set period of time. During this animation, your bone hierarchy is modified to convey the motion in the animation. Trying to store every bone's position and orientation for every millisecond of animation is impossible; there's just too much data to store it effectively. Instead, you can space out the movements over a longer period of time (every second or two)or better yet, whenever a major change in each bone's position or orientation takes place. For example, imagine your armrather, imagine the arm illustrated in Figure 5.1 .

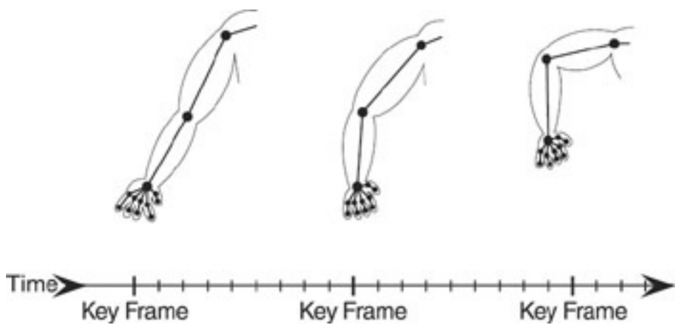


Figure 5.1: A bone's animation over a period of time; a key marks the extent of each movement

The bones that construct the arm in Figure 5.1 are pointing straight out at the start of the animation. Over time, the bones bend at the imaginary elbow, come to a rest, and then bend at a different angle. So there are three major changes in the bones' orientationstraight (the default position), slightly bent, and a major bend in the joint. These three changes are the three keys in the animation.

Now, instead of storing the orientation of the bones every millisecond, store those three keys and the exact time (in milliseconds) that the bones would reach the appropriate orientation. In this example, suppose the arm animation starts at 0 milliseconds, reaches the first key (half−bent) at 500 milliseconds, and reaches the last key (fully bent) at 1,200 milliseconds.

Here's where using key frames comes in handy. Suppose you want to calculate the orientation of the bones at a specific timesay, at 648 milliseconds. That time just so happens to fall between the second and third keys (148 milliseconds past the second key). Now, assume that the two transformation matrices represent the orientations of each bone in the animation.

```
D3DXMATRIX matKey1, matKey2;
```

By taking each key and interpolating the values between them, you can come up with a transformation to use at any time between the keys. In this example, at 648 milliseconds in the animation, you can interpolate the transformations as follows:

```
// Get the difference in the matrices
D3DXMATRIX matTransformation = matKey2 − MatKey1;

// Get keys' times
float Key1Time = 500.0f;
float Key2Time = 1200.0f;

// Get the time difference of the keys
float TimeDiff = Key2Time − Key1Time;

// Get the scalar from animation time and time difference
float Scalar = (648.0f − Key1Time) / TimeDiff;

// Calculate the interpolated transformation matrix
matTransformation *= Scalar;
matTransformation += matKey1;
```

And there you have it! The `matTransformation` matrix now holds the interpolated transformation that you would apply to the bone in question to perfectly synchronize it to the animation! To increase precision, you can use translation, rotation, and scaling values instead of transformation matrices during the interpolation. I'll get back to doing just that in a bit, but for now let's get back to the `Animation` template with which you'll be dealing.

For every bone in your hierarchy, there should be a matching `Animation` object. Immediately following the `Animation` object declaration, you'll see a referenced data object name. This is the name of the bone that has its animation data defined using the preceding `AnimationKey` objects. This means that in the previous example, the two bones, `Bip01` and `Bip01_LeftArm`, are being animated.

One or more `AnimationKey` objects follow the data object reference. The `AnimationKey` objects define the keys of the animation a bone uses, which can include translation, rotation, scale, or transformation keys. Take a closer look at each key type and how you store its information in the objects.

Working with the Four Key Types

Currently, there are four types of keys you can use in your animation sets, each signified by a value ranging from 0 to 4 that is listed in the .X file following the frame reference inside an `AnimationKey` template. These four keys and their respective values are

- **Rotational keys (type 0)**. These are quaternion rotational values, stored using four componentsw, x, y, and z.
- **Scaling keys (type 1)**. You can also use this type of key to animate scaling values. A scale key uses three components that represent the x, y, and z scaling values to use.
- **Translation keys (type 2)**. These keys specify a position in 3D space using three components that represent the x, y, and z coordinates. You can easily store these three values as a vector.
- **Transformation matrix keys (type 4)**. You can use this key to compound all transformations into matrices. This key uses 16 floats that represent a homogenous 4x4 transformation matrix that transforms a bone.

So getting back to the previous Animation data objects, you can see that the very first AnimationKey object (which affects the Bip01 bone) defines a transformation matrix key (represented by the value 4), as shown here:

```
{Bip01}
  AnimationKey {
    4;
    3;
        0; 16; 1.00000, 0.00000, 0.00000, 0.00000,
               0.00000, 1.00000, 0.00000, 0.00000,
               0.00000, 0.00000, 1.00000, 0.00000,
               0.00000, 0.00000, 0.00000, 1.00000;;,
     1000; 16; 1.00000, 0.00000, 0.00000, 0.00000,
               0.00000, 1.00000, 0.00000, 0.00000,
               0.00000, 0.00000, 1.00000, 0.00000,
               0.00000, 0.00000, 0.00000, 1.00000;;,
     2000; 16; 1.00000, 0.00000, 0.00000, 0.00000,
               0.00000, 1.00000, 0.00000, 0.00000,
               0.00000, 0.00000, 1.00000, 0.00000,
               0.00000, 0.00000, 0.00000, 1.00000;;;
 }
```

As for the second AnimationKey object (which affects the Bip01_LeftArm bone), there are three keys in usetranslation (value 2), scaling (value 1), and rotation (value 0).

```
{Bip01_LeftArm}
AnimationKey {
  0;
1;
  0; 4; 1.00000, 0.000000, 0.00000, 0.000000;;;
}
AnimationKey {
  1;
  1;
  0; 4; 1.000000, 1.00000, 1.000000;;;
}
AnimationKey {
  2;
  1;
  0; 3; 0.000000, 0.00000, 0.000000;;;
}
```

As you may have surmised by now, you can have any number of AnimationKey objects per Animation object, with each AnimationKey object using one specific key type. Following the key's type value (0=rotational, 1=scaling, 2=position, 4=matrix) is the number of keys to use in the animation sequence for that specific bone. In the first bone's set (Bip01) there are three matrix type keys defined, whereas the remaining

AnimationKey objects (that affect the Bip01_LeftArm bone) use only one key for each of the remaining transformation types (rotation, scaling, and position).

Next comes the key data. The first value for each key is the time value, which is specified using an arbitrary value that you choose (such as seconds, milliseconds, frames, or any other form of measurement you wish to use). In my examples, I always specify time as milliseconds. A number that defines how many key values are to follow comes after the time value. Take the following key data, for example:

```
AnimationKey {
 2; // Key type
 1; // # of keys
 0; // Key time
 3; // # of values to follow for key's data
 10.00000, 20.00000, 30.00000;;; // key's data }
```

The first value, 2, means the key is used to contain translation animation keys. The 1 means there is one key to follow. The first and only key is located at time 0. The value 3 follows the time, which means that three more values (10, 20, and 30) are to follow. The three values represent the coordinates to use for that time in the animation.

Going back to the earlier example, you can see that the first animation key (the transformation matrix key) has three matrices that are used at times 0, 1000, and 2000. At those exact times during the animation, you will set the transformation matrix for the Bip01 bone.

For the time values between keys, you need to interpolate the matrices to come up with the correct transformations. In fact, you can interpolate all key types to get the correct values to use between keys. The easiest way to interpolate is to use the transformation matrix, scaling, translation, or rotation values from the animation keys, divide by the time between two keys, and multiply the result based on the time into the key. You saw me use linear interpolation for a transformation matrix in the previous section. Before long, I'll show you how to interpolate translation, scaling, and rotation values as well.

That's basically it for the AnimationKey! You just need to read in each key contained within your AnimationKey data objects and apply it to the proper bone transformations using the interpolated matrices over time.

Okay, enough of the animation templates, data objects, keys, and interpolation for now; we'll get back to that stuff in a bit. For now, let's get your hands into some real code and see how to load the animation data into your game. Then I'll get back to showing you how to work with the actual data.

Reading Animation Data from .X Files

In Chapter 3, you learned how to load meshes and frame hierarchies from an .X file, as well as how to use frame hierarchies to deform (modify) the mesh while rendering. This chapter's purpose is to teach you how to read in the animation data contained in an .X file so you can play back key−framed animations.

The first step to reading in the animation data is to look at the templates you'll be using and build a couple classes to contain the data from those templates' data objects. First, here are the templates that you'll be working with, along with their declarations:

```
template AnimationKey {
  <10DD46A8−775B−11cf−8F52−0040333594A3>
  DWORD keyType;
  DWORD nKeys;
  array TimedFloatKeys keys[nKeys];
}

template Animation {
  <3D82AB4F−62DA−11cf−AB39−0020AF71E433>
  [AnimationKey]
  [AnimationOptions]
  [...]
}

template AnimationSet {
  <3D82AB50−62DA−11cf−AB39−0020AF71E433>
  [Animation]
}
```

At the top of the list you have `AnimationKey`, which stores the type of animation key data, the number of key values to follow, and the key data itself, which is stored in an array of `TimedFloatKey` objects that store the time and an array of floating−point values in the form `Time:NumValues:Values`.

Data objects of the `Animation` template class type can store an `AnimationKey` object and an `AnimationOptions` object. Notice that the `Animation` template is left open because it needs a frame data object reference to match the animation key to a bone.

Last, there's the `AnimationSet` template, which only contains `Animation` objects. You can store any number of animations within an animation set; typically, you'll have one animation for each bone.

Note The `AnimationOptions` template, while not used in this book, is highly useful if you want your artists to specify playback options. Inside the `AnimationOptions` template, you'll find two variables`openclosed` and `positionquality`.If `openclosed` is set to 0, then the animation in which the object is embedded doesn't loop; a value of 1 means the animation loops. As for `positionquality`, setting it to a value of 0 means to use spline positions, whereas a value of 1 means to use linear positions. Typically, you'd set `positionquality` to 1.

You'll want to use some custom classes to store your animation data; those classes will pretty much mirror the `Animation` templates' data exactly. First, you want a class that contains the values of the various key typesscaling, translation, rotation, and transformation. The first two types, scaling and translation, both use a vector, so one class will suffice.

```
class cAnimationVectorKey
{
  public:
    float       m_Time;
    D3DXVECTOR3 m_vecKey;
};
```

Note You can find the animation classes discussed in this section (along with the complete source code to a skeletal animation system) on the book's CD−ROM. Look at the end of this chapter for more information on the BoneAnim demo program.

Rotation keys use a quaternion (a four−dimensional vector).

```
class cAnimationQuaternionKey
{
  public:
    float          m_Time;
    D3DXQUATERNION m_quatKey;
};
```

Last, the transformation key uses a 4x4 homogenous matrix.

```
class cAnimationMatrixKey
{
  public:
    float      m_Time;
    D3DXMATRIX m_matKey;
};
```

So far, so good. Remember that each bone in your animation has its own list of keys to use, which is the purpose of the `Animation` template. For each bone in your hierarchy, there is a matching `Animation` data object. Your matching animation class will therefore contain the name of the bone to which it is connected, the number of keys for each type (translation, scaling, rotation, and transformation), a linked list data pointer, and a pointer to the bone (or frame) structure you're using in your hierarchy. Also, you need to include a constructor and destructor that clear out the class's data.

```
class cAnimation
{
  public:
    char       *m_Name; // Bone's name
    D3DXFRAME  *m_Bone; // Pointer to bone
    cAnimation *m_Next; // Next animation object in list

    // # each key type and array of each type's keys
    DWORD                    m_NumTranslationKeys;
    cAnimationVectorKey     *m_TranslationKeys;
    DWORD                    m_NumScaleKeys;
    cAnimationVectorKey     *m_ScaleKeys;
    DWORD                    m_NumRotationKeys;
    cAnimationQuaternionKey *m_RotationKeys;
    DWORD                    m_NumMatrixKeys;
    cAnimationMatrixKey     *m_MatrixKeys;

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

Finally, the `AnimationSet` template contains the `Animation` objects for an entire bone hierarchy. At this point, all your animation set class needs to do is track an array of `cAnimation` classes (remember that each bone in the hierarchy has a matching `cAnimation` class), as well as the length of the complete animation.

```
class cAnimationSet
{
  public:
    char          *m_Name;   // Name of animation
    DWORD          m_Length; // Length of animation
    cAnimationSet *m_Next;   // Next set in linked list
    DWORD          m_NumAnimations;
    cAnimation    *m_Animations;

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

Assuming you want more than one animation set loaded at once, you can even create a class that contains an array (or rather, a linked list) of `cAnimationSet` classes, which means that you can access a whole slew of animations with one interface! This class, called `cAnimationCollection`, is also derived from the `cXParser` class developed in Chapter 2, so you can parse .X files directly from the class in which you'll be storing the animations.

Here's the class declaration for `cAnimationCollection`:

```
class cAnimationCollection : public cXParser
{
  public:
    DWORD          m_NumAnimationSets;
    cAnimationSet *m_AnimationSets;

  protected:
    // Parse an .X file object
    BOOL ParseObject(IDirectXFileData *pDataObj,
                       IDirectXFileData *pParentDataObj,
                       DWORD Depth,
                       void **Data, BOOL Reference);

    // Find a frame by name
    D3DXFRAME *FindFrame(D3DXFRAME *Frame, char *Name);

  public:
    cAnimationCollection();
    ~cAnimationCollection();

    BOOL Load(char *Filename);
    void Free();

    void Map(D3DXFRAME *RootFrame);
    void Update(char *AnimationSetName, DWORD Time);
};
```

The details of each function in the `cAnimationCollection` class are not very important at this point so I'll get back to them in a bit. At this point, all you're interested in is reading in that animation data from an .X file. The custom .X parser contained in the `cAnimationCollection` class does just thatit loads the data from the `Animation` objects' data into the dizzying array of objects you've just seen.

For every `AnimationSet` object you encounter in the .X file being parsed, you need to allocate a `cAnimationSet` class and add it to the linked list of animation sets already loaded. The most current `cAnimationSet` object is stored at the start of the linked list, which makes it easy to determine which animation−set data you are currently using.

From here, you can appropriately parse the `Animation` objects. If you were to keep the most current `cAnimationSet` object at the start of your linked list, every following `Animation` object that you parse would belong to that animation−set object. The same goes for the `AnimationKey` objectstheir data would belong to the first `cAnimation` object in the linked list.

I will skip the constructors and destructors for all the different classes because you only need them to clear and release each class's data. You're only interested in a couple functions, the first being `cAnimationCollection::ParseObject`, which deals with each animation object being parsed from an .X file.

The ParseObject function starts by checking whether the currently enumerated object is an AnimationSet. If it is, a new cAnimationSet object is allocated and linked to the list of objects, while the animation−set object is simultaneously named for further reference.

```
BOOL cAnimationCollection::ParseObject(                  \
                   IDirectXFileData *pDataObj,           \
                   IDirectXFileData *pParentDataObj,     \
                   DWORD Depth,                          \
                   void **Data, BOOL Reference)
{
  const GUID *Type = GetObjectGUID(pDataObj);
  DWORD i;

  // Check if object is AnimationSet type
  if(*Type == TID_D3DRMAnimationSet) {

   // Create and link in a cAnimationSet object
   cAnimationSet *AnimSet = new cAnimationSet();
   AnimSet->m_Next = m_AnimationSets;
   m_AnimationSets = AnimSet;

   // Increase # of animation sets
   m_NumAnimationSets++;

   // Set animation set name (set a default one if none)
   if(!(AnimSet->m_Name = GetObjectName(pDataObj)))
   AnimSet->m_Name = strdup("NoName");
}
```

As you can see, nothing special goes on with the animation set objectsyou're merely allocating an object that will eventually hold the upcoming Animation data objects. Speaking of which, you want to parse the Animation objects next.

```
// Check if object is Animation type
if(*Type == TID_D3DRMAnimation && m_AnimationSets) {

  // Add a cAnimation class to top−level cAnimationSet
  cAnimation *Anim = new cAnimation();
  Anim->m_Next = m_AnimationSets->m_Animations;
  m_AnimationSets->m_Animations = Anim;

  // Increase # of animations
  m_AnimationSets->m_NumAnimations++;
}
```

Again, nothing special going on there. In the preceding code, you're simply ensuring that there's a cAnimationSet object allocated at the start of the linked list. If there is, you can allocate and link a cAnimation object to the list in the cAnimationSet object.

While we're on the topic of the cAnimation object, the next bit of code retrieves the name of the frame instance located within the Animation object.

```
// Check if a frame reference inside animation object
if(*Type == TID_D3DRMFrame && Reference == TRUE &&           \
                           m_AnimationSets &&                \
                           m_AnimationSets−>m_Animations) {

// Make sure parent object is an Animation object
if(pParentDataObj && *GetObjectGUID(pParentDataObj) ==       \
                                    TID_D3DRMAnimation) {

// Get name of frame and store it as animation
if(!(m_AnimationSets−>m_Animations−>m_Name =                 \
                           GetObjectName(pDataObj)))
m_AnimationSets−>m_Animations−>m_Name=strdup("NoName");
}
}
```

You can see in this code that only referenced frame objects are allowed in the `Animation` object, a fact that you can verify by checking the parent object's template GUID. Whew! So far this code is pretty easy, isn't it? Well, I don't want to burst your bubble, but the hardest is yet to come! In fact, the most difficult part of loading animation data from an .X file is loading the key data. Don't let me scare you away, though; the key data is nothing more than a time value and an array of values that represent the key data.

The remaining code in the `ParseObject` function checks to see which type of key data an `AnimationKey` object holds. Depending on the type of data, the code branches off and reads the data into the specific key objects (`m_RotationKeys`, `m_TranslationKeys, m_ScaleKeys`, and `m_MatrixKeys`) inside the current `cAnimation` object. Take a closer look to see how simple this code really is.

```
// Check if object is AnimationKey type
if(*Type == TID_D3DRMAnimationKey && m_AnimationSets &&   \
                         m_AnimationSets−>m_Animations) {

// Get a pointer to top−level animation object
cAnimation *Anim = m_AnimationSets−>m_Animations;

// Get a data pointer
DWORD *DataPtr = (DWORD*)GetObjectData(pDataObj, NULL);

// Get key type
DWORD Type = *DataPtr++;

// Get # of keys to follow
DWORD NumKeys = *DataPtr++;
```

In addition to checking to see whether there are valid `cAnimationSet` and `cAnimation` objects at the start of the linked list of objects, the preceding code gets a pointer to the key data and pulls out the key type value and the number of keys to follow. Using the key type, the code then branches off to allocate the key−frame objects and load in the key data.

```
  // Branch based on key type
  switch(Type) {
    case 0: // Rotation
      delete [] Anim−>m_RotationKeys;
      Anim−>m_NumRotationKeys = NumKeys;
      Anim−>m_RotationKeys = new                            \
                           cAnimationQuaternionKey[NumKeys];

      for(i=0;i<NumKeys;i++) {
        // Get time
        Anim−>m_RotationKeys[i].m_Time = *DataPtr++;
        if(Anim->m_RotationKeys[i].m_Time >                 \
                              m_AnimationSets->m_Length)
         m_AnimationSets->m_Length =                        \
                         Anim->m_RotationKeys[i].m_Time;

      // Skip # keys to follow (should be 4)
      DataPtr++;

      // Get rotational values
      float *fPtr = (float*)DataPtr;
      Anim->m_RotationKeys[i].m_quatKey.w = *fPtr++;
      Anim->m_RotationKeys[i].m_quatKey.x = *fPtr++;
      Anim->m_RotationKeys[i].m_quatKey.y = *fPtr++;
      Anim->m_RotationKeys[i].m_quatKey.z = *fPtr++;
      DataPtr+=4;
  }
  break;
```

You'll recall from earlier in this chapter that rotation keys use quaternion values. These values are stored in w, x, y, z order; to make sure you use the proper values, you must read them into the key's quaternion object appropriately.

Next comes the code to load in the scaling and translation keys, which both use vectors to store the x−, y−, and z−axis information.

```
    case 1: // Scaling
      delete [] Anim->m_ScaleKeys;
      Anim->m_NumScaleKeys = NumKeys;
      Anim->m_ScaleKeys = new cAnimationVectorKey[NumKeys];
      for(i=0;i<NumKeys;i++) {
        // Get time
        Anim->m_ScaleKeys[i].m_Time = *DataPtr++;
        if(Anim->m_ScaleKeys[i].m_Time >                    \
                               m_AnimationSets->m_Length)
        m_AnimationSets->m_Length =                         \
                           Anim->m_ScaleKeys[i].m_Time;

      // Skip # keys to follow (should be 3)
      DataPtr++;

      // Get scale values
      D3DXVECTOR3 *vecPtr = (D3DXVECTOR3*)DataPtr;
      Anim->m_ScaleKeys[i].m_vecKey = *vecPtr;
      DataPtr+=3;
    }
    break;

  case 2: // Translation
    delete [] Anim->m_TranslationKeys;
    Anim->m_NumTranslationKeys = NumKeys;
    Anim->m_TranslationKeys = new                           \
                          cAnimationVectorKey[NumKeys];

    for(i=0;i<NumKeys;i++) {
      // Get time
      Anim->m_TranslationKeys[i].m_Time = *DataPtr++;
      if(Anim->m_TranslationKeys[i].m_Time >                \
                             m_AnimationSets->m_Length)
       m_AnimationSets->m_Length =                          \
                    Anim->m_TranslationKeys[i].m_Time;
      // Skip # keys to follow (should be 3)
      DataPtr++;

      // Get translation values
      D3DXVECTOR3 *vecPtr = (D3DXVECTOR3*)DataPtr;
      Anim->m_TranslationKeys[i].m_vecKey = *vecPtr;
      DataPtr+=3;
    }
    break;
```

Last is the code to read an array of transformation matrix keys.

```
    case 4: // Transformation matrix
      delete [] Anim->m_MatrixKeys;
      Anim->m_NumMatrixKeys = NumKeys;
      Anim->m_MatrixKeys = new cAnimationMatrixKey[NumKeys];
      for(i=0;i<NumKeys;i++) {
        // Get time
        Anim->m_MatrixKeys[i].m_Time = *DataPtr++;
        if(Anim->m_MatrixKeys[i].m_Time >                  \
                                m_AnimationSets->m_Length)
         m_AnimationSets->m_Length =                       \
                            Anim->m_MatrixKeys[i].m_Time;

        // Skip # keys to follow (should be 16)
        DataPtr++;

        // Get matrix values
        D3DXMATRIX *mPtr = (D3DXMATRIX *)DataPtr;
        Anim->m_MatrixKeys[i].m_matKey = *mPtr;
        DataPtr += 16;
    }
    break;
  }
 }
```

Okay now, take a quick breather and look back at what you've just accomplished. So far, you've processed every `AnimationSet, Animation`, and `AnimationKey` object (not to mention referenced `Frame` objects that contain the bones' names), plus you've loaded the key objects full of the animation data. You're almost ready to start animating!

Almost is right; there is one small step leftmatching the animation objects to their respective bone objects.

Matching Animations to Bones

After you've loaded the animation data, you need to map the animation classes to their respective bones in the bone hierarchy. Mapping the hierarchies is important because whenever an animation is updated, you need a quick way to access the bone's transformations. By mapping, you create an easier method of accessing the bones.

In this instance, the bone hierarchy will be represented in a `D3DXFRAME` hierarchy. If you're using DirectX 8, you might notice that you don't have access to the `D3DXFRAME` object; it's a structure specific to DirectX 9. Don't fret, however; the Direct3D helper code used by all the demos in this book compensates for the missing structure by creating a mock version of `D3DXFRAME` you can use. You can check out the mock `D3DXFRAME` structure in Direct3D.h in this chapter's directory on the CD−ROM.

Inside the D3DXFRAME structure, there are two linked list pointers that you'll use to help construct the hierarchy. From the root D3DXFRAME structure you are using, you can access child objects through the D3DXFRAME::pFrameFirstChild pointer and sibling objects through the D3DXFRAME::pFrameSibling pointer.

The next function in cAnimationCollection to which you want to pay attention is Map. You use the Map function to map the animation structure's m_Bone pointer to a frame in the frame hierarchy that shares the same name.

The Map function scans through every cAnimationSet object and iterates every cAnimation object contained in each of the animation set objects. The name of each cAnimation object is compared to each of the frame's names; if a match is found, the cAnimation::m_Bone pointer is set to the frame's address.

The Map function takes the hierarchy's root frame parameter.

```
void cAnimationCollection::Map(D3DXFRAME *RootFrame)
{
  // Go through each animation set
  cAnimationSet *AnimSet = m_AnimationSets;
  while(AnimSet != NULL) {

    // Go through each animation object
    cAnimation *Anim = AnimSet->m_Animations;
    while(Anim != NULL) {

      // Go through all frames and look for match
      Anim->m_Bone = FindFrame(RootFrame, Anim->m_Name);

      // Go to next animation object
      Anim = Anim->m_Next;
    }

    // Go to next animation set object
    AnimSet = AnimSet->m_Next;
  }
}
```

Whereas the Map function only scans through each of the cAnimationSet and cAnimation objects, the FindFrame function recursively works through the frame hierarchy to look for a match to the name you provide. When it finds a matching name, the FindFrame function returns the pointer to the specific frame. Take a look at the FindFrame code on which the Map function depends.

```
D3DXFRAME *cAnimationCollection::FindFrame(D3DXFRAME *Frame, char *Name)
{
  D3DXFRAME *FramePtr;

  // Return NULL if no frame
  if(!Frame)
    return NULL;

  // Return current frame if no name used
  if(!Name)
    return Frame;

  // Process child frames
  if((FramePtr = FindFrame(Frame->pFrameFirstChild, Name)))
    return FramePtr;
  // Process sibling frames
  if((FramePtr = FindFrame(Frame->pFrameSibling, Name)))
    return FramePtr;

  // Nothing found
  return NULL;
}
```

Again, take a deep breath. The animation data has been loaded, and you've mapped the animation objects to the bone hierarchy. All that's left to do is update the animation and set the transformation matrices for the bones.

Updating Animations

After you've matched the animation classes to the bone hierarchy, you can begin animating your meshes! All you have to do is scan the animation keys for each bone, applying the interpolated transformations to each bone's transformation before rendering. This is merely a matter of iterating through each animation class and its keys to find the proper key values to use.

Going back to the `cAnimationCollection` class, you can see that one function will do all that for you. By supplying the `cAnimationCollection::Update` function with the name of the animation set you want to use, as well as the time in the animation, all of the transformation matrices in your entire mapped bone hierarchy will be set and ready for rendering.

Take a closer look at the `Update` function to see how you can update your animation data.

```
void cAnimationCollection::Update(char *AnimationSetName,   \
                                  DWORD Time)
{
  cAnimationSet *AnimSet = m_AnimationSets;
  DWORD i, Key, Key2;

  // Look for matching animation set name if used
  if(AnimationSetName) {

    // Find matching animation set name
    while(AnimSet != NULL) {

      // Break when match found
      if(!stricmp(AnimSet->m_Name, AnimationSetName))
        break;

      // Go to next animation set object
      AnimSet = AnimSet->m_Next;
    }
  }
  // Return no set found
  if(AnimSet == NULL)
    return;
```

The `Update` function starts by scanning the list of animation sets loaded into the linked list. If you instead supply a NULL value for `AnimationSetName`, `Update` will merely use the first animation set in the list (which happens to be the last set loaded). If no matching sets are found using the name you specified, the function returns without further delay.

Once a matching animation set is found, however, the code continues by scanning each `cAnimation` object in it. For each animation object, the entire list of keys (translation, scaling, rotation, and transformation) is searched, and the time you specify is checked to see which key to use.

After you've found the proper key to use, the values (rotation, scaling, translation, or transformation) are interpolated, and a final transformation matrix is computed. This final transformation matrix is then stored in the mapped bone (as pointed to by the `m_Bone` pointer).

You've already seen how to scan a list of keys to look for the ones between which a specific time falls, so I'll skip the code here. I'll leave it to you to check out the exact code on the book's CD−ROM; consult the end of this chapter for more information on the BoneAnim demo.

Once you've calculated the transformations to apply to each bone from the animation data, you can jump right back into the game and render the mesh using the techniques you learned in Chapter 1. Remember, you must apply the transformation matrices for each bone to the appropriate vertices in the mesh, and the best way to do so is to use a vertex shader. Consult Chapter 1 for more information on drawing skeletal−based meshes if you need help.

Obtaining Skeletal Mesh Data from Alternative Sources

Microsoft's .X file format is not the only kid on the block when it comes to mesh and animation data storage. As ease of use and simplicity go, there are two other formats that tend to be perfect mediums for your mesh and animation datachUmbaLum sOft's Milkshape 3D .MS3D format and id Software's Quake 2 .MD2 format.

The Milkshape .MS3D file format is somewhat like a binary .X, except that an .MD3D file only stores a single mesh and bone hierarchy. In fact, the .MS3D file format is extremely linear in that it doesn't use templates; rather, it uses a pre−defined series of structures stored one after another in the file.

The Quake 2 .MD2 file format is merely a bunch of meshes thrown together in a single file. Each mesh represents a single frame of animation from a series of animation sets. Whereas .MS3D files contain skeletal animations, the .MD2 format contains only morphing animation sets.

Note Morphing mesh animations is another awesome topic covered elsewhere in this book, so I thought I'd just mention the dual functionality of the `MeshConv` program here. For now, you can ignore any mention of morphing animation and get right to the topic at handconverting skeletal−based key−framed animation sets from .MS3D to .X.

So, for skeletal−based animations you can use .MS3D files, and for morphing animations you can use .MD2 files. Exactly how do you use those files? Information on the formats is widely available. Check out the book *Focus On 3D Models* (Premier Press, 2002) or Web sites like http://www.gamedev.net or http://nehe.gamedev.net.

The CD−ROM includes a program called MeshConv, which you can use to convert .MS3D and .MD2 files into .X files. After you execute the program, you are presented with the MeshConv dialog box, shown in Figure 5.2.

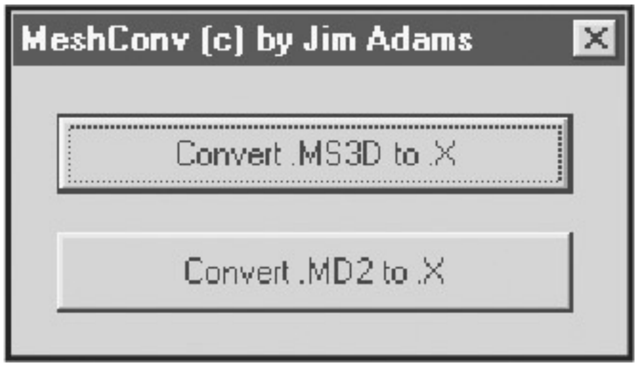


Figure 5.2: The MeshConv dialog box contains two buttons you can click on to convert .MS3D and .MD2 files to .X.

Don't let the lack of controls in the MeshConv program scare youit does a great job of converting all .MS3D and .MD2 files into .X files using the layout and templates you've seen throughout this chapter. The .MS3D files will be saved using a frame hierarchy and a single `AnimationSet` object, whereas .MD2 files will be saved using a series of `Mesh` objects and a `MorphAnimationSet` object that contains the names of the meshes to use in succession for morphing animation.

Note The CD−ROM includes the completely commented source code for the MeshConv program. Check the end of this chapter for more information on the various programs and their locations.

To convert a file (whether it is an .MS3D or an .MD2 file) into an .X file, click on the appropriate button in the MeshConv dialog box. The Open File dialog box will appear. This dialog box allows you to navigate your drives and locate the file you want to convert to .X format. Select an appropriate file to convert and click Open.

Next, the Save .X File dialog box will appear. You can use this dialog box to locate and select an .X file name into which you want to save the mesh and animation data. Enter a file name and click Save. After a moment, you should see a message box informing you that the conversion was successful.

Now you are ready to use your .X file with one of the classes you developed earlier in this chapter, either for skeletal−based animation sets or morphing animation sets. The skeletal−based animation set uses a single source mesh that is deformed (shaped) by the bone hierarchy; review this chapter and Chapter 2 for more information on using skinned meshes.

For morphing animations (from .MD2), you'll have a series of `Mesh` objects that contain every target morphing mesh used in the source file. A single `MorphAnimationSet` object will help you load animation data into your project using the classes and techniques you've studied in this chapter.

For an example of how to work with the .X files you created using the MeshConv program, check out the demo programs included for this chapter. That's rightboth the BoneAnim and MorphAnim demos used converted .MS3D and .MD2 files to demonstrate skeletal and morphing animation. Check out those demos, and have fun!

Check Out the Demos

In this chapter, you learned how to load animation sets and use that data to animate your on−screen meshes. To better demonstrate these animation concepts, I have created a program (SkeletalAnim) that shows your favorite lady of skeletal−based animation, Microsoft's Tiny (from the DirectX SDK samples), doing what she does bestwalking around! When you run the demo application, you'll be greeted with a scene like the one shown in Figure 5.3.

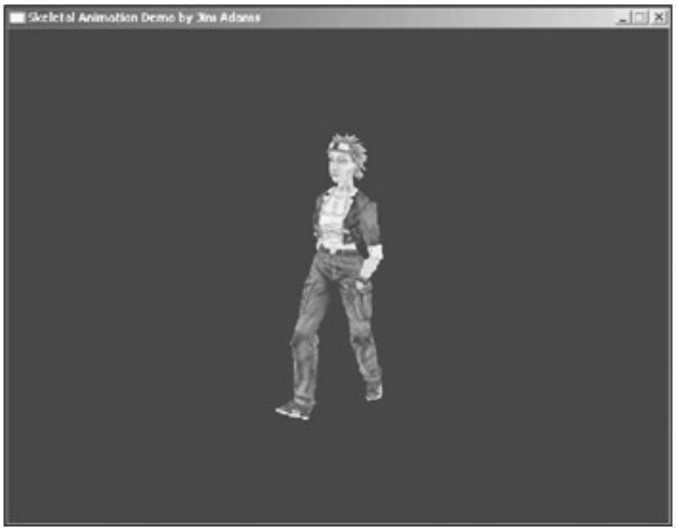


Figure 5.3: Tiny on the move in the SkeletalAnim demo! This demo shows you how to use skeletal−based animated meshes.

Programs on the CD

In the Chapter 5 directory on this book's CD−ROM, you'll find the following two demos to peruse and use for your own game projects.

- **MeshConv**.You can use this utility program to convert your .MS3D and .MD2 files to .X files. The source code is fully commented and shows the format of those two file types. It is located at \BookCode\Chap05\MeshConv.
- **SkeletalAnim**.This program demonstrates how to read and use skeletal key−framed animations. It is located at \BookCode\Chap05\SkeletalAnim.

Chapter 6: Blending Skeletal Animations

Overview

In Chapter 5, "Using Key−Framed Skeletal Animation," you saw how you could take a series of pre−designed animation sequences and insert them into your game project. That's cool; the only problem is that those animations were so static. That's right, the animations never changed and will always remain the same no matter how many times you play them. What if there was a way to make your animations a bit more dynamic?

Well, in fact, with a little bit of extra work you can combine a series of those bland repeating animations into a set of new dynamic animations; animations that are unique every time you play them. For example, combine your game character's walking animation with his separate punching animation to make your character walk and punch at the same time!

That's right, by combining (or *blending*, as it is known) the motions of the various animations, you can create literally hundreds of new animations from your pre−calculated key−frame animation sets. This chapter is here to show you how to blend those animations.

Blending Skeletal Animations

As I mentioned in this chapter's introduction, you would normally use a series of pre−calculated key−framed animations in your game projects. You create these animations using 3D modeling programs such as discreet's 3DStudio Max or Caligari's trueSpace. Although they served their purpose quite nicely, those pre−created animation sequences did lack one major aspectuniqueness. Once an animation, always an animationmeaning that the animations are the same, regardless of how many times you play them.

Moving into a more dynamic world of animation, the technique of *animation blending* has become more than a buzzword. What's thatyou don't know what animation blending is? Animation blending is the ability to take separate animations and blend, or rather *combine*, them to create a new animation.

For instance, as Figure 6.1 shows, you can blend an animation of your character walking and another animation of him waving his arm to create an animation of him walking and waving his arm at the same time!

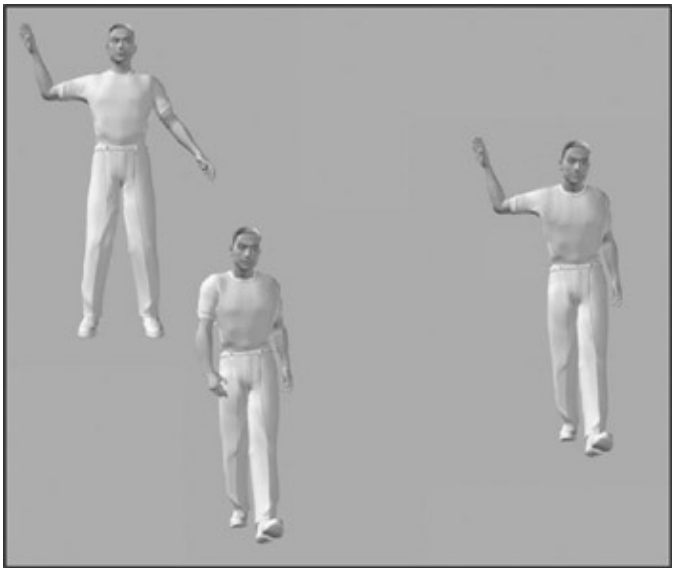


Figure 6.1: Even though the two animations shown on the left are separately defined, you can combine them into one unique animation, as shown on the right.

You don't have to stop with only blending two animations, you could go on to combine three, four, or even ten different animations into one unique animation! With each new animation you add, the possibilities of blending increase exponentially. With only a small set of animations at your disposal, you could literally create hundreds of new animations using animation blending.

Now, I won't lie to youthe theory and implementation of animation blending is extremely, excruciatinglysimple. That's right; animation blending is one of those things that makes you wonder why the heck you weren't doing it earlier. It's that easy! It all has to do with the way you combine the various transformations of the skeletal structure's bones.

Combining Transformations

As you saw in Chapter 4 and 5, your skeletal animations are merely series of transformation matrices applied to the bones of your mesh's skeletal structure. These transformations include translations, scaling, and rotations. For the most part, the transformations are rotations. The bones rotate at the joint; only the root bone is typically allowed to translate around the world, and even then that's best left up to the world transformation (rather than directly translating the bones themselves). Those points aside, the transformations create the animation.

As you can see in Figure 6.2, you can create new poses by adding various transformations to the existing transformations of the skeletal structure. For example, to make the skeleton's arm move, add a rotational transformation matrix to the arm bone transformation. Slowly increasing the rotational value added to the bone transformation creates smooth animation.

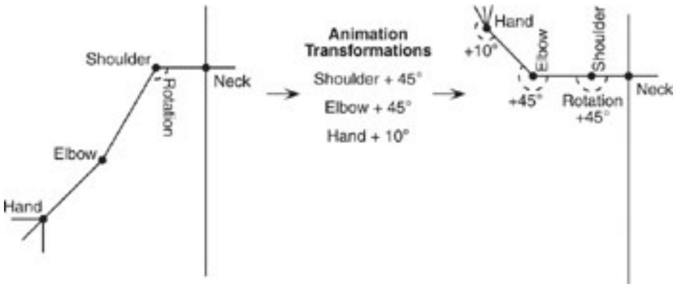


Figure 6.2: The skeleton's default pose (on the left) has an associated set of transformation matrices; when combined with animation set transformation matrices, these will create new poses.
You can see that you achieve animation by combining (through matrix concoction) or directly storing a set of animation transformations with your skeleton's transformation matrices. To smoothly animate a mesh, you can use linear interpolation to scale the animation set's transformation matrices over time.

So at the most basic level, you are dealing with transformation matrices to create animation; there's one transformation matrix to apply for each bone in the mesh. The pre−calculated key−frame animation set is the source of the transformation matrices that are applied to the bone's transformation.

Think about thisinstead of taking that single transformation matrix from your animation set (from a matrix key frame or combined from a series of position, translation, and rotation key frames), why couldn't you just take a series of transformations that affect the same bone from multiple animation sets and combine them? After all, you're using matrix concoction to combine multiple transformations, so why not just throw in a few more transformations from multiple animations while you're at it?

Whoa! You caught me thereyou can't just concoct the matrices and expect the transformations to come out correctly. Think of it: Matrix concoction is non−commutative, meaning that the order in which you multiply the various transformations is crucial. If you were to multiply two transformations that both were rotated and then translated, you would end up with a final transformation that rotates, translates, rotates, and finally translates. That's obviously too much transformation data for a single bone that typically rotates and then translates.

To correct this problem, you need to add the transformations instead of multiplying them. So, for instance, the previous two transformations that rotate and then translate would combine into a transformation that only rotates and then translates (as opposed to rotating, translating, rotating, and finally translating). Adding transformations is perfectly acceptable!

Adding two matrices (represented by `D3DXMATRIX` objects) is as simple as the following line of code:

```
D3DXMATRIX matResult = Matrix1 + Matrix2;
```

From there on, you can use the `matResult` matrix for your transformations; rest assured, it represents the combined transformations of `Matrix1` and `Matrix2`. To combine more animation transformations, just add another matrix to `matResult` and continue until you have combined all the transformations you want to use.

Now that you know this information, you can begin combining the various transformations of separate animation sets. To make things easier, you can even extend the animation objects you developed in Chapter 5.

Enhancing Skeletal Animation Objects

Now that you've seen how simple it is to blend multiple skeletal animations, why not take this new knowledge and add on to the skeletal animation objects and code that you saw in Chapter 5? Sounds like a great idea; by adding a single function to the `cAnimationCollection` class, you can be on your way to blending animations like the pros.

In fact, rather than messing with the code from `cAnimationCollection`, just derive a new class that handles blended animations. This new derived class, `cBlendedAnimationCollection`, is defined as follows:

```
class cBlendedAnimationCollection : public cAnimationCollection
```

```
{
  public:
    void Blend(char *AnimationSetName,
               DWORD Time, BOOL Loop,
               float Blend = 1.0f);
};
```

Wow, that's a small class! The one and only function declared in `cBlendedAnimationCollection` is `Blend`, which is meant to take over the `cAnimationCollection::Update` function. Why not just derive a new `Update` function, you ask? Well, with `cBlendedAnimationCollection`, you can use the regular animation sets you used in Chapter 5, as well as your (soon to be) newly developed blended animation sets.

Take a close look at the `Blend` function to see what's going on, and then I'll show you how to put your new class to good use.

```
void cBlendedAnimationCollection::Blend(                    \
                         char *AnimationSetName,            \
                         DWORD Time, BOOL Loop,             \
                         float Blend)
{
```

The `Blend` function prototype takes four parameters, the first of which is `AnimationSetName`. When calling `Blend`, you need to set `AnimationSetName` to the name of the animation set you are going to blend into the animation. Remember from Chapter 5 that each animation set in your source .X file has a unique animation name (as defined by the `AnimationSet` data object's instance name). You must set `AnimationSetName` to a matching name from the .X file.

I'll get back to the animation sets in a bit. For now, I want to get back to the `Blend` prototype. The second parameter of `Blend` is `Time`, which represents the time in the animation that you are using to blend the animation. If you have an animation that is 1000 milliseconds in length, then `Time` can be any value from 0 to 999. Specifying a value larger than the animation's length forces the `Blend` function to use the last key frame in the list to blend the animation.

What about looping the animation? Well, that's the purpose of the third parameter, `Loop`. If you set `Loop` to `FALSE`, then your animations will refuse to update if you try to update using a time value that is greater than the length of the animation. However, if you set `Loop` to `TRUE`, the `Blend` function bounds−checks the time value (`Time`) to always fall within the range of the animation's time.

The previous paragraph may not make perfect sense at first, so to help you understand, imagine the following function:

```
void UpdateAnimation(DWORD Elapsed)
{
   static DWORD AnimationTime = 0; // Animation time

   // Call Blend, using AnimationTime as the time in the animation
   AnimationBlend.Blend("Walk", AnimationTime, FALSE, 1.0f);

   // Update the time of the animation
   AnimationTime += ELapsed;
}
```

In the `UpdateAnimation` function, you are tracking the animation time via a static variable. Every time

UpdateAnimation is called, the Blend function is used to blend in an animation called Walk, using a time value specified as AnimationTime. Assuming the Walk animation is 1000 milliseconds in length and the elapsed time between calls to UpdateAnimation is 50 ms, you can see tell that the animation would reach its end after 20 calls to the function. This means after you call UpdateAnimation 20 times, the animation will stop (because you set Loop to FALSE).

Going back and changing the Loop value to TRUE forces Blend to bounds−check the timing value and make sure it always uses a valid timing value. When I say bounds−check, I mean to use a *modulus* calculation. I'll show you how to use the modulus calculation in a moment; for now I want to get back to the fourth and final parameter.

The last parameter is Blend, which is a floating−point value that represents a scalar value used to modify the blended transformation matrix before it is applied to the skeletal structure. For example, if you are blending a walking animation, but you only want 50 percent of the transformation to be applied, then you would set Blend to 0.5.

Okay, that's enough for the parameters; let's get into the function code! If you've perused the cAnimationCollection::Update function, you'll notice that the majority of code in the Blend function is the same. Starting off, you'll find a bit of code that scans the linked list of animation sets to find the one that matches the name you provided as AnimationSetName.

```
cAnimationSet *AnimSet = m_AnimationSets;

// Look for matching animation set name if used
if(AnimationSetName) {

   // Find matching animation set name
   while(AnimSet != NULL) {

      // Break when match found
      if(!stricmp(AnimSet->m_Name, AnimationSetName))
         break;

      // Go to next animation set object
      AnimSet = AnimSet->m_Next;
    }
 }

  // Return no set found
  if(AnimSet == NULL)
     return;
```

If you set AnimationSetName to NULL, then Blend will use the first animation set in the linked list of animation sets. If you specified a name in AnimationSetName and none was found in the linked list, then Blend will return without any further ado.

Now that you have a pointer to the appropriate animation set object, you can bounds−check the time according to the value set in Time and the looping flag Loop.

```
   // Bounds time to animation length
   if(Time > AnimSet->m_Length)
     Time = (Loop==TRUE)?Time %                                 \
                       (AnimSet->m_Length+1):AnimSet->m_Length;
```

Quite an ingenious little bit of code, the previous tidbit does one of two things, depending on the `Loop` flag. If `Loop` is set to `FALSE`, then `Time` is checked against the length of the animation `(AnimSet->m_Length).` If `Time` is greater than the length of the animation, then `Time` is set to the length of the animation, thus locking it at the last millisecond (and later on, the last key frame) of the animation. If you set `Loop` to `TRUE`, then a modulus calculation forces `Time` to always lie within the range of the animation's length (from 0 to `AnimSet->m_Length`).

After you have calculated the appropriate `Time` to use for the animation, it is time to scan the list of bones in your skeletal structure. For each bone, you are going to track the combined transformations from the appropriate key frames. For each key frame found in the animation, you need to add (not multiply) the transformation to the skeletal structure's transformations.

```
// Go through each animation
cAnimation *Anim = AnimSet->m_Animations;
while(Anim) {

     //Only process if it's attached to a bone
     if(Anim->m_Bone) {

         // Reset transformation
         D3DXMATRIX matAnimation;
         D3DXMatrixIdentity(&matAnimation);

         // Apply various matrices to transformation
```

From here, scan each key frame (depending on the type of keys used) and calculate the transformation to apply to your skeletal structure. For the sake of space, I'm only going to list the code that scans matrix keys.

```
// Matrix
if(Anim->m_NumMatrixKeys && Anim->m_MatrixKeys) {
     // Loop for matching matrix key
     DWORD Key1 = 0, Key2 = 0;
     for(DWORD i=0;i<Anim->m_NumMatrixKeys;i++) {
       if(Time >= Anim->m_MatrixKeys[i].m_Time)
         Key1 = i;
     }

     // Get 2nd key number
     Key2 = (Key1>=(Anim->m_NumMatrixKeys-1))?Key1:Key1+1;

     // Get difference in keys' times
     DWORD TimeDiff = Anim->m_MatrixKeys[Key2].m_Time-
                      Anim->m_MatrixKeys[Key1].m_Time;
     if(!TimeDiff)
       TimeDiff = 1;

     // Calculate a scalar value to use
     float Scalar = (float)(Time -                         \
        Anim->m_MatrixKeys[Key1].m_Time) / (float)TimeDiff;

     // Calculate interpolated matrix
     D3DXMATRIX matDiff;
     matDiff = Anim->m_MatrixKeys[Key2].m_matKey -          \
               Anim->m_MatrixKeys[Key1].m_matKey;

     matDiff *= Scalar;
     matDiff += Anim->m_MatrixKeys[Key1].m_matKey;
```

```
        // Combine with transformation
        matAnimation *= matDiff;
      }
```

I discussed the code just shown in Chapter 5, so I won't explain it again here. Basically, the code is searching the key frames and calculating an appropriate transformation to use. This transformation is stored in `matAnimation.`

From this point on, things take a decidedly different course than the `cAnimationCollection::Update` function code. Instead of storing the transformation matrix (`matAnimation`) in the skeletal structure's frame object, you will calculate the difference in the transformation from `matAnimation` to the skeletal structure's initial transformation (stored in `matOriginal` when the skeletal structure was loaded). This difference in transformation values is scaled using the floating−point `Blend` value you provided, and the resulting transformation is then added (not multiplied, as you do with concoction) to the skeletal structure's frame transformation. This ensures that the transformations are properly blended at the appropriate blending values.

After that, the next bone's key frames are scanned, and the loop continues until all bones have been processed.

```
    // Get the difference in transformations
    D3DXMATRIX matDiff = matAnimation − Anim−>m_Bone−>matOriginal;

    // Adjust by blending amount
    matDiff *= Blend;

    // Add to transformation matrix
    Anim−>m_Bone−>TransformationMatrix += matDiff;
   }
   // Go to next animation
   Anim = Anim−>m_Next;
  }
}
```

Congratulations, you've just completed your `Blend` function'! Let's put this puppy to work! Suppose you have a mesh and frame hierarchy already loaded, and you want to load a series of animations from an .X file. Suppose this .X file (called Anims.x) has four animation sets`Stand,Walk`, `Wave`, and `Shoot`. That's two animations for the legs and two for the arms and torso. Here's a bit of code to load the animation sets:

```
// pFrame = root frame in frame hierarchy
cBlendedAnimationSet BlendedAnims;
BlendedAnims.Load("Anims.x");

// Map animations frame hierarchy
BlendedAnims.Map(pFrame);
```

Now that you have an animation collection loaded, you can begin blending the animations before updating and rendering your skinned mesh. Suppose you want to blend the `Walk` and `Shoot` animations, both using 100 percent of the transformations. To start, you must reset your frame hierarchy's transformations to their original states. This means you need to copy the `D3DXFRAME_EX::matOriginal` transformation into the `D3DXFRAME_EX::TransformationMatrix` transformation. This is very important because it serves as a base to which your animation set transformations are added during the blending operation.

Note The `D3DXFRAME_EX` object is an extended version of Direct3D's `D3DXFRAME` object. You can read about `D3DXFRAME_EX` in Chapter 1.

```
// Use D3DXFRAME_EX::Reset to reset transformations
pFrame−>Reset();
```

Once the transformations have been reset to their original states, you can blend your animation sets.

```
// AnimationTime = time of animation, which is the
// elapsed time since start of the animation

// Blend in the walk animation
BlendedAnims.Blend("Walk", AnimationTime, TRUE, 1.0f);

// Blend in the shoot animation
BlendedAnims.Blend("Shoot", AnimationTime, TRUE, 1.0f);
```

Once you've blended all the animation sets you're going to use, you need to update your frame hierarchy, which is then used to update your skinned mesh.

```
// Update the frame hierarchy
pFrame−>UpdateHierarchy();
```

After you have updated the hierarchy (the transformation matrices have been combined and the results are stored in `D3DXFRAME_EX::matCombined`), you can update your skinned mesh and render away! I won't go into more detail here; I'll leave it up to you to check out the blended animation demo on the CD−ROM to see how the helper functions and objects developed in Chapter 1 are put to good use in your blended animation techniques.

Check Out the Demo

Although this chapter only has one demo to tout, it sure is a whopper! Demonstrating the technique of blended animation, the SkeletalAnimBlend demo program (see Figure 6.3) shows off Microsoft's Tiny character in all her blended glory!

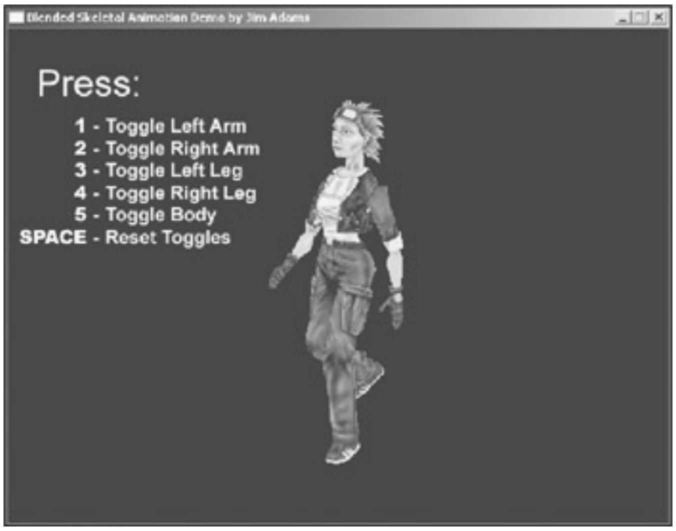


Figure 6.3: Explore your newfound blended skeletal animation techniques by choosing which animations to blend in real time.

I edited the Tiny.x file to split her animation up into multiple sets. There are animation sets for each of her arms and legs, as well as an animation set for her body. Pressing any of the keys displayed on the screen toggles the blending of the appropriate animation set. For instance, hitting 1 toggles the blending of her left arm animation sequence. When enabled, the left arm animation has Tiny swinging her arm in sequence with her step. When disabled, her left arm hangs limp.

To really illustrate the power of blending, suppose you add a new animation set to the Tiny.x file that has Tiny waving her arm as opposed to swinging it back and forth. You only need to turn off blending of the swinging animation and blend in the waving animation to create a new and totally unique animation!

Programs on the CD

In the Chapter 6 directory of this book's CD−ROM, you'll find a single project that demonstrates the use of skeletal animation blending. This project is

- **SkinAnimBlend.** This project demonstrates how to blend multiple animations to create a new unique animation. It is located at \BookCode\Chap06\SkeletalAnimBlend.

Chapter 7: Implementing Rag Doll Animation

Overview

Hiding atop a ledge on the east side of the compound, I lie waiting to catch sight of my unwary foe. My trigger finger twitches anxiously, awaiting its only course of action. I can see it now−at any moment my enemy will step out from a door below and I'll strike. My bazooka shell will blast him and send his virtual body flying through the air, bouncing off the stone walls of the compound's barracks. My plan is perfect. Little do I know that my opponent is just as sneaky as I am. I hear his sinister laugh, and I look over just in time to see his grenade plunk down right beside me. I guess I'll be the one whose imaginary body bounces off the walls tonight. What the heck, there's always the next game!

A typical moment from a typical first person shooter game; there's nothing special about this story except for the part about the characters bouncing around as a result of the impact from the game's various weapons. Effects such as these unique death sequences are possible through the use of a technique known as rag doll animation, in which you turn your characters into flimsy pieces of digital stitch work. As your characters are flung back, they embark on a short, yet completely unique, series of motions−bouncing off obstacles with limbs flailing, as if you took an actual rag doll and flung it across a room!

Games such as Epic Games'*Unreal Tournament 2003* use rag doll animation for all characters' death sequences, and believe me, if you haven't seen this effect, you are definitely missing out. If you want to use rag doll animation in your own game, you're in luck−it's all here in this chapter!

Creating Dolls from Characters

Rag doll animation is the new fad in advanced animation. Just as the name implies, it is akin to picking up an object (a character in your game, for example) and flinging it around, with its appendages flailing as if it were a limp rag doll. Imagine taking a life−sized stuffed doll and throwing it up in the air. Its arms and legs would twist around, and if the body hit a hard object, it would react and bounce in even more disturbing ways.

This is where rag doll animation gets its appeal−by creating totally unique animations every time you run through the simulation. You can apply factors such as collisions, gravity, wind, and any other force to your bouncy, flailing characters, thus changing the way they are flung about.

Games such as *Unreal Tournament 2003* demonstrate what a great technique such as rag doll animation can do for a game. Nothing is cooler than blasting your opponent's character with a rocket launcher and having the body flung around like a pile of limp noodles. That's one animation technique you'll certainly want for your own game, and if you keep reading, you'll definitely learn about it.

The whole concept of rag doll animation is pretty simple. Take a single character from your game as an example to see what's going on. This character (imagine he looks like the character in Figure 7.1) is constructed from a skinned mesh and a series of bones that form a skeletal structure. No problem here−you learned all about skinned meshes and skeletal structures in Chapter 4.

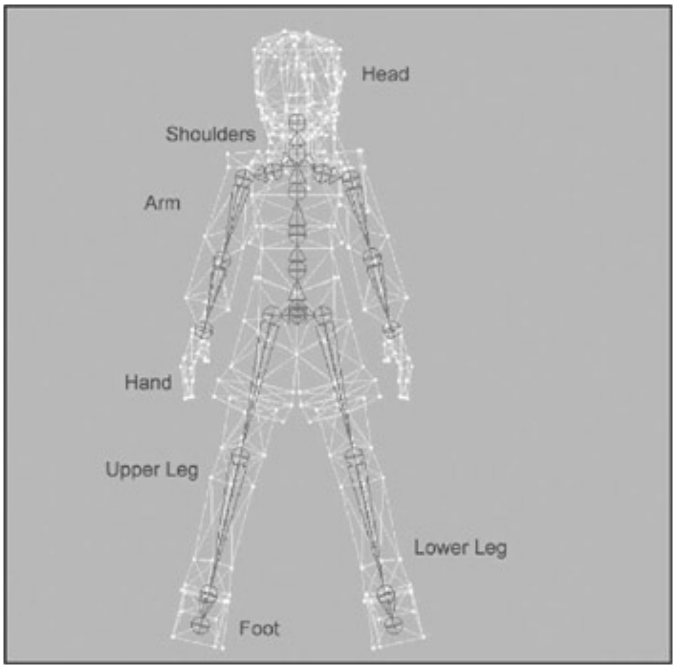


Figure 7.1: You can split a sample character, complete with a skeletal structure and skinned mesh, into a series of individual components.

Imagine surrounding each bone (and each vertex belonging to each bone) in Figure 7.1 with a box−a *bounding box,* to be exact. (Even though it seems odd to work with boxes instead of the actual skeletal structure or mesh, it will begin to make sense as you go along.) You see, it's these boxes that you really care about; they represent the parts of the character's body that can twist, turn, and flop around in your rag doll animation.

Figure 7.2 shows you what the sample character would look like if he were represented using only the bounding boxes.

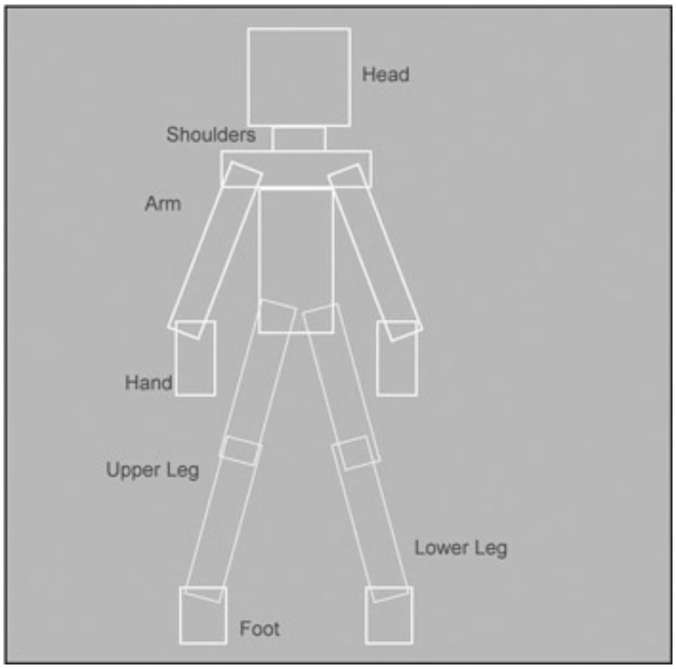


Figure 7.2: The sample character's bones and vertices have been replaced by bounding boxes, with each box encompassing the area occupied by the bones and vertices.

It's your job to process and track the motion of these boxes (rather than the bones and vertices) as they move during the animation. You are free to move and rotate each box as you see fit. Since the boxes represent your character's bones, each bone in the skeletal structure moves as each box does (even if that motion causes the bones to drift apart, breaking the bone−to−bone connections, which I'll talk about in a moment).

How do your bones move to match the motion of the boxes? They inherit the same transformations as the bounding boxes that move around your 3D world. So basically, whenever a box moves, the bone moves to match. (It's almost as if the bones don't exist, since you only work with the boxes during simulation.)

It should be slowly starting to make sense why I picked boxes to represent the bones and their vertices. You can represent each bone and its vertices by just eight points (the corners of the bounding box), rather than by an unknown (and most likely high) number of vertices in the mesh. Also, the math involved in tracking the motion of the boxes is much simpler than the math for tracking the motion of each bone and vertex.

Now assume you have taken the pains to construct this collection of boxes that represent your bones and vertices. As you can see in Figure 7.3, each box completely encloses its respective bone's vertices and bone−to−bone connection points. These boxes are transformed to match their respective bones' positions and orientations.

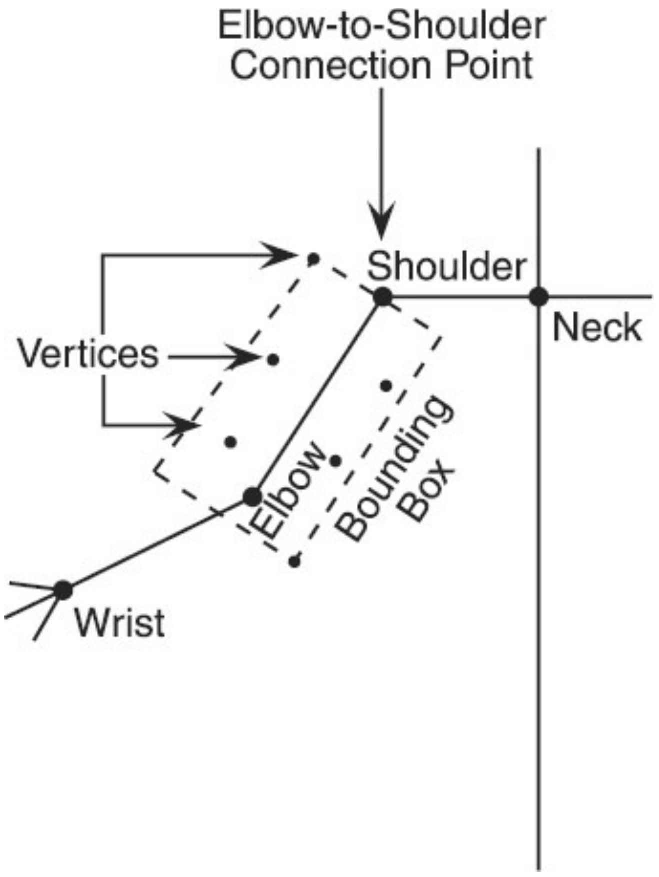


Figure 7.3: Each bounding box surrounds a bone's vertices and bone−to−bone connection points.
After these boxes are created and positioned around the 3D mesh, you can start moving them. Suppose you want your character's arms to flail wildly. Add a little force to a couple of the boxes, and they will begin to move. As the boxes move, so do the bones that each box represents.

These boxes are truly separate entities at this point; they don't share the bone−to−bone connections that exist in your skeletal structure (the same connections that hold your skeletal structure together in a recognizable form). In a way, this is perfect because you can manipulate directly the character's skeletal structure using the boxes' transformations in place of the bones' combined transformations. This means you don't have to combine each bone's local transformation with its parent's transformation to orient all bones correctly−just copy the box's transformations and you're all set!

The only problem is these boxes can drift away from one another, literally tearing your characters apart. There must be some way to enforce the bone−to−bone connections using your bounding boxes, thus holding your characters together at the joints (and saving you from a macabre scene of limbs flying in different directions).

In fact, there are a number of ways to make sure your bounding boxes remain connected at the exact points where their respective bones join one another. The method I'll show you in this chapter uses springs to pull the bounding boxes back together every time one of them moves. In Figure 7.4, you can see a couple out−of−control bounding boxes that are flying about the scene and tearing the poor character apart. Between each box, you can see some springs that are used to bring those limbs back together.

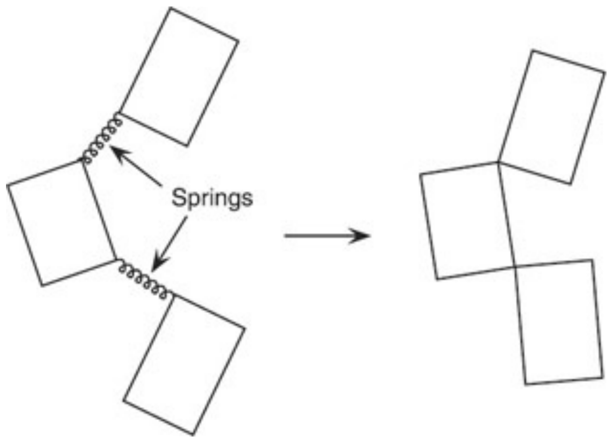


Figure 7.4: A series of springs helps you bring the separated boxes back into shape.
After you move the boxes and use the springs to restore the connection points, it's up to you to copy the boxes' orientations into the bone hierarchy, update your skinned mesh, and render away. See how easy it is!

It's easy in theory, of course−it's the implementation that's tough. Everything you just read is in fact a huge physics problem−how do you track the orientation, movement, and collisions of those boxes? By using rigid−body physics, that's how!

Working with Rigid−Body Physics

As you just read, your characters' bodies can be split into boxes that represent the various bones that make up their skeletal structures. You want to be able to track the orientations of these boxes as you move them around during the animation, and you also want to able to copy those orientations back to their original bones.

Okay, okay, so what does this have to do with so−called *rigid−body* physics? Well, the rigid−body part means that your meshes are considered solid objects−the boxes that represent the bones never change shape and never penetrate the area of another object. Therefore, the bounding boxes that represent your bones *are* rigid bodies.

The study of rigid−body physics (also commonly referred to as *rigid−body dynamics*) tracks the motion of those solid objects, including the effects of forces such as gravity and friction on them. Collision also plays a huge part because those solid objects that represent your characters' bodies need to bounce off one another as well as the surrounding terrain.

Before you go trudging off to your bookcase to grab your high−school physics textbook, let me tell you now that using rigid−body physics isn't really as hard as it seems. Sure, there are a lot of formulas and calculations that would give even your math teacher nightmares, but once they are broken down, you'll wonder why you ever worried.

In fact, I'll take all this rigid−body physics stuff step by step so it will make sense to you, starting with the creation of a rigid−body object.

Creating a Rigid Body

Rigid−body physics is the system of tracking the motion and collision of solid objects. To keep things as simple as possible at this point, you can think of those solid objects as three−dimensional bounding boxes that enclose each of your skeletal structure's bones. Thinking in 3D terms, you might think of the boxes as being composed of eight points (one in each corner).

The points are analogous to vertices, the box analogous to a mesh. Using a simple transformation (rotation and translation), you can position the box and its eight corner points anywhere in your 3D world. The box and its corner points each have their own role in this rigid−body system. The box itself represents the points as a whole−what affects the box affects the points. So if you move the box, the points move with it−you don't have to worry about the exact positions of the points inside the box.

As for the points, they not only help determine a bounding box's size, they also help with collision detection. You see, if one of those points is penetrating another object's space, then you can say those objects are colliding, and you need to handle it. This makes it much easier to perform collision detection; instead of checking every vertex in your mesh to see whether it hits an object, you check whether a point from the bounding box does. I'll get back to collision detection in a bit; for now, I want to get back to defining a rigid body.

As I mentioned, each bounding box is composed of eight points. To determine the location of these eight points, you need to completely surround each bone (and its vertices and bone−to−bone connection points) in your skeletal structure with a box. I'll do that later in this chapter, once we start using rigid−body physics in the animations; for now, just assume you have a box specified by a width, depth, and height of your choice.

As Figure 7.5 shows, the center of this box is considered its origin. The dimensions of the box range from −width/2, −height/2, −depth/2 to +width/2, +height/2, +depth/2. Using these values (the dimensions), you can calculate the coordinates of your bounding box.

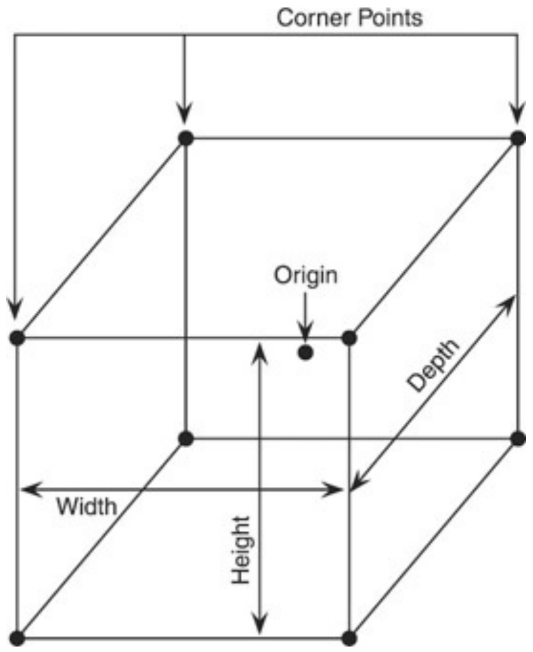


Figure 7.5: A box (which represents a rigid body) is defined by positioning eight points around its origin. The positions of these points are determined by halving the body's width, height, and depth.
To store these eight points, you can use two sets of eight `D3DXVECTOR3` objects. The first eight objects will store the coordinates of the box in local space, much like your vertex buffers do for your meshes.

```
D3DXVECTOR3 vecLocalPoints[8];
```

The second set of eight points will store the coordinates of the points as they move around in the world space.

```
D3DXVECTOR3 vecWorldPoints[8];
```

You can think of the second set of coordinates as the transformed coordinates, whereas the first set is the untransformed coordinates. Whenever you move a rigid body, you take the coordinates from the first set of points, transform them, and store the resulting coordinates in the second set of points. Again, this is just like you would normally do when you're working with the vertices of a mesh.

At this point, you only need to store the eight coordinates of the rigid body's corners in the `vecLocalPoints` array of vector objects (using the body's dimensions, width, depth, and height).

```
// Width, Height, Depth = 3 float's with body's dimensions
// Note that all dimensions are specified in Meters

// Store the body's dimensions in a vector
D3DXVECTOR3 vecSize = D3DXVECTOR3(Width, Height, Depth);

// Store the dimensions halved in a vector object
D3DXVECTOR3 vecHalf = vecSize * 0.5f;

// Store the coordinates of the corners
vecLocalPoints[0]=D3DXVECTOR3(−vecHalf.x, vecHalf.y,−vecHalf.z);
vecLocalPoints[1]=D3DXVECTOR3(−vecHalf.x, vecHalf.y, vecHalf.z);
vecLocalPoints[2]=D3DXVECTOR3( vecHalf.x, vecHalf.y, vecHalf.z);
vecLocalPoints[3]=D3DXVECTOR3( vecHalf.x, vecHalf.y,−vecHalf.z);
vecLocalPoints[4]=D3DXVECTOR3(−vecHalf.x,−vecHalf.y,−vecHalf.z);
vecLocalPoints[5]=D3DXVECTOR3(−vecHalf.x,−vecHalf.y, vecHalf.z);
vecLocalPoints[6]=D3DXVECTOR3( vecHalf.x,−vecHalf.y, vecHalf.z);
```

```
vecLocalPoints[7]=D3DXVECTOR3( vecHalf.x,−vecHalf.y,−vecHalf.z);
```

Not only do a box's dimensions affect its size, they also affect its mass. Oh yes, a rigid body has mass−it's this mass that affects its movements. Objects with greater mass require more force to move, whereas objects with less mass take less force to move. I'll get into using mass in your calculations in a bit; for now, I want to show you how to determine the mass of an object.

You can use any method you want to calculate a box's mass, but to keep it simple I'll use the box's dimensions. To calculate the mass of a box, I use the product of the lengths of each dimension (stored in the `vecSize` vector from the previous code bit), as in the following bit of code:

Note You'll notice from the code's comments that I'm specifying my 3D units in meters (as opposed to using generic measurements, which you may be used to). You shouldn't have to worry about this fact when it comes to your meshes, however, because the calculations will work out fine without you having to worry about conversions to other measurement systems.

```
float Mass = vecSize.x * vecSize.y * vecSize.z;
```

Once you have calculated the dimensions and mass of the box, you can position and orient it within your 3D world.

Positioning and Orienting Your Rigid Bodies

When you've created a bounding box from your rigid body, it is time to position it in your world. A body's position is defined using a vector object that represents the 3D coordinates of the rigid body. Like a mesh, these coordinates represent the center of the rigid body−the body's origin.

As for the rotation of the object, you can represent it using one of three things−a set of Euler angles (x, y, and z rotational values), a rotational transformation matrix, or a quaternion. Although I know this might make a few of you cringe, I'm choosing a quaternion to represent the body's rotation. Bottom line: Quaternions are numerically stable and easier to work with than the other methods.

For those of you who just hated hearing that last bit, let me give you a brief breakdown of how a quaternion works. A quaternion (or to be more exact, a *unit quaternion*) is a set of four component values that define a vector and scalar. The vector components are defined as x, y, z, and w. The x, y, z trio can be described as v; the w can be described as s. So, the two ways to define a quaternion are

```
    q = [s, v]
    q = [w, [x, y, z]]
```

In Direct3D, a quaternion is stored in a `D3DXQUATERNION` object, which uses the x, y, z, w convention.

```
typedef struct D3DXQUATERNION {
  FLOAT x;
  FLOAT y;
  FLOAT z;
  FLOAT w;
} D3DXQUATERNION;
```

As you can see in Figure 7.6, the x, y, z components define a directional vector. This directional vector, v, represents a rotational axis.

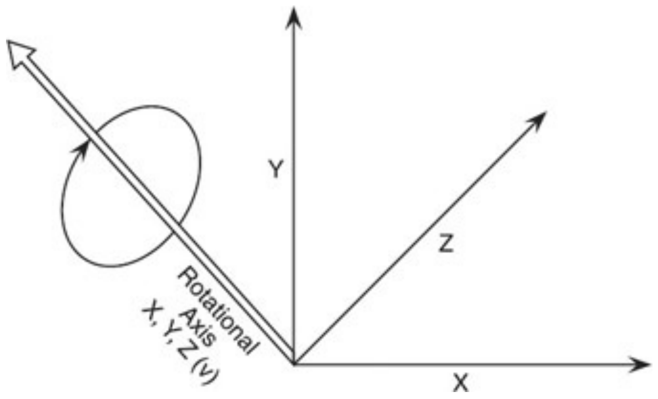


Figure 7.6: The vector component (v = x, y, z) of a quaternion defines the rotational axis.
The angle of rotation, specified in radians, is actually stored in the quaternion components (w, x, y, z) using the following calculations:

```
      q = [w = cos(/2), xyz = sin(/2)xyz]
```

In English, this calculation breaks down to the w component containing the cosine of the angle of rotation (divided by two) and the normalized x,y,z vector scaled by the sine of the angle (divided by two). In code, this breaks down to:

```
q = [cos(Angle/2.0f), normalized(x,y,z)*sin(Angle/2.0f)]
```

To put this in other terms, suppose you want to create a quaternion that represents a rotation of 45 degrees (0.785 radians) around the y−axis. You create a vector that points upward in the direction of the positive y−axis, and you set the magnitude of this vector to the sine of half the angle. You then set the w component to the cosine of half the angle.

```
// Instance a quaternion to use
D3DXQUATERNION quatRotation;

// Create a vector that represents the axis of rotation
D3DXVECTOR3 vecAxis = D3DXVECTOR3(0.0f, 1.0f, 0.0f);

// Normalize the vector in order to set its magnitude
D3DXVec3Normalize(&vecAxis, &vecAxis);

// Scale the vector to the sine of half the angle
vecAxis *= (float)sin(0.785 / 2.0f);

// Store the vector in the quaternion
quatRotation.x = vecAxis.x;
quatRotation.y = vecAxis.y;
quatRotation.z = vecAxis.z;

// Compute the w component using the cosine of half the angle
quatRotation.w = (float)cos(0.785f / 2.0f);
```

A unit quaternion, such as the one you are going to use, has the constraint ($x^2 + y^2 + z^2 + w^2 = 1$) placed on it, meaning that the sum of all squared components must equal one. If the sum does not equal one, then the quaternion is not of unit length, and henceforth it must be normalized before you can use it. With Direct3D, normalizing a quaternion is easy using the `D3DXQuaternionNormalize` function. For example, to normalize a quaternion stored as `quatOrientation`, you can use the following bit of code:

```
D3DXQuaternionNormalize(&quatOrientation, &quatOrientation);
```

So where is all this quaternion stuff leading? Now that you've defined the axis and angle of rotation, you can use the quaternion to transform your rigid body's points. That's right; the quaternion takes the place of your everyday transformation matrix and Euler angles! Well, sort of.

Because Direct3D really doesn't work with quaternion transformations (yet!), you need to convert the quaternion to a rotational transformation matrix, which Direct3D can use. You can perform this conversion using the `D3DXMatrixRotationQuaternion` function, as demonstrated here. (Notice how I'm transposing the resulting matrix because quaternions are right−handed, and I'm using left−handed transformations in this book.)

```
// quatOrientation = Quaternion w/rotational values set
D3DXMATRIX matOrientation;
D3DXMatrixRotationQuaternion(&matOrientation,           \
                             &quatOrientation);

// Convert transformation matrix to left−handed
D3DXMatrixTranspose(&matOrientation, &matOrientation);
```

Why did I go to the trouble of showing you the secrets of unit quaternions if you're eventually going to work with a rotation transformation matrix? The transformation matrix is really a secondary object that is used to transform your rigid body's bounding−box points (among a couple other things). By secondary, I mean that while the orientation of the body is maintained with a quaternion, the matrix does the transformation work.

So here we are, back to where we started, only now you know how quaternions work and that you'll use them to represent the orientation of your rigid bodies. After you've calculated the quaternion you want to use to orient your rigid body and stored the world−space coordinates in the position vector, you can transform your rigid body's local points into 3D world−space coordinates.

```
// vecLocalPoints[] = array of body points in local space
// vecWorldPoints[] = array of body points in world space
// quatOrientation = quaternion containing rotation of body
// vecPosition = position of rigid body

// Create a transformation matrix from quaternion
D3DXMATRIX matOrientation;
D3DXMatrixRotationQuaternion(&matOrientation,              \
                             &quatOrientation);

// Go through each of the eight points and transform them
for(DWORD i=0;i<8;i++) {

       // Orient point using transformation matrix
       D3DXVec3Transform(&vecWorldPoints[i],               \
                         &vecLocalPoints[i],               \
                         &matOrientation);

       // Translate point using vector
       vecWorldPoints[i] += vecPosition;
}
```

Each point is now oriented properly in world space, according to the 3D position coordinates and rotation values that you have specified. Although this is all neat and exciting, there really isn't anything going on−the body is just sitting still. You need to start shaking that puppy up and making it move!

Processing the Motion of Rigid Bodies

Motion of a rigid body takes two forms−linear movement and angular rotation. *Linear movement* is when your body moves in a straight line in any direction, and *angular rotation* is when your rigid body rotates around its axis. Simple, isn't it?

All motion is the result of forces applied to the rigid body. If you apply a force to the body, it can produce linear movement and/or angular rotation as a result. Linear movement is easy to calculate; just move the body in the direction of the force. Rotation is somewhat similar−apply a force, and the body rotates in response. The direction and amount of rotation to apply depend on where you apply the force.

Take a closer look at each type of motion.

Moving Rigid Bodies

Linear movement is the result of an object being pushed or pulled in a single direction. As various forces are applied, the direction can change and the speed of the object's movement can be altered. It all depends on the forces applied to the body. To make thing easier on you, all forces that affect a rigid body are combined, giving you only one force to work with.

Forces themselves are stored as vector objects `(D3DXVECTOR3)`, which define the direction and the amount of force applied (the magnitude). For our purposes, forces represent an acceleration value of sorts that you want to apply to an object with a mass of 1. What about objects with a mass other than 1, or what about forces that always accelerate objects at the same rate regardless of their mass? You can turn to Newton's second law of motion to help you out here.

```
      F = ma
```

Newton's second law states that the amount of force (F) you must apply to achieve a specific acceleration (a) depends on the mass (m) of the object. For example, suppose you want to define gravity as a force that accelerates all objects 9.8 m/s^2 in the negative y−axis, regardless of the object's mass. Using Newton's second law, you have to multiply mass (m) by acceleration (a) to obtain force. Therefore, to define gravity using an object's mass value stored in the variable `Mass`, you can use the following bit of code:

```
D3DXVECTOR3 vecGravity = D3DXVECTOR3(0.0f, −9.8f, 0.0f) * Mass;
```

What about using forces that accelerate objects without compensating for their mass? For instance, you can declare the following implied force, which accelerates objects with a mass of 1 5.0 m/s^2 in the positive x−axis:

```
D3DXVECTOR3 vecImplied = D3DXVECTOR3(5.0f, 0.0f, 0.0f);
```

If you tried to apply the implied force to an object with a mass of 2, you would only achieve an acceleration of 2.5 m/s^2 in the positive x−axis. How did I come up with 2.5 m/s^2? Easy−just flip the F=ma calculation:

```
      a = F/m
```

When you get around to computing the acceleration to apply to an object's velocity, you need to divide your force vector by the mass of the object. Since force was 5.0 m/s^2 and mass was 2, then a = 5/2.

To summarize: All forces you set represent the acceleration of an object with a mass of 1. If you want to make sure the acceleration is constant regardless of the object's mass, you multiply the force vector by the object's mass.

Moving on in the example, you now have to force the vectors (gravity and an implied force) that you want to use to move an object.

```
D3DXVECTOR3 vecGravity = D3DXVECTOR3(0.0f, −9.8f, 0.0f) * Mass;
D3DXVECTOR3 vecImplied = D3DXVECTOR3(5.0f, 0.0f, 0.0f);
```

Before you apply the two forces to any body, combine them into one net force.

```
D3DXVECTOR3 vecForce = vecGravity + vecImplied;
```

You want to work with this combined force. Later in this chapter you'll read about the various forces and how to calculate them. For now, just assume you've gone through the trouble to compile the forces into one vector object.

As I mentioned earlier, applying force to any point on a body will cause it to move. You really don't need to know where on the body this force is applied because the body will move in the direction of that force regardless (or decelerate as a result of an opposing force).

So, using the two forces that were combined into one, you can linearly move your rigid body by directly applying the force to the body's position vector. Suppose that a rigid body's position is stored in a vector called `vecPosition`.

```
// Add force vector to position vector
vecPosition += vecForce;
```

I know that some of you are going to take notice and stop me here. What about all that acceleration stuff I talked about earlier? Now that I mention it, what about taking time and velocity into consideration? I know I mentioned physics in there somewhere, so it's about time I straightened things out.

An object has a linear velocity, which is the speed and direction that the object moves. Forces represent an acceleration factor of sorts. The magnitude of the forces represents the amount of acceleration; whereas the direction of the vector determines the direction in which the force is applied. Since objects have mass, you must scale accordingly these forces (or rather, the net force to be applied) to create a "real" acceleration value. That way, objects with greater mass don't accelerate as much as objects with less mass. Remember F=ma and a=F/m?

Going back to the gravity vector, you can see that you definitely want an acceleration of 9.8 m/s^2, regardless of the object's mass. This means scaling the force by the object's mass to compensate for a=F/m, which would have scaled the gravity force into something other than the constant acceleration of 9.8 m/s^2. As for the implied force (5.0 m/s^2), it will be scaled accordingly to a=F/m, so if your object doesn't have a mass of 1, the acceleration will be something other than 5.0 m/s^2.

So, going back to the issue of applying force to a rigid body, you divide the net force by the mass and add the result to the linear velocity of a rigid body. Wait a second−I forgot to mention time! Not only do you scale the force by the mass, you must also multiply the resulting vector (which now represents acceleration) by the amount of time the acceleration was applied. You then apply this time−based linear velocity to the body to make it move (or slow down, depending on the direction the forces are applied).

Store velocity in a vector called `vecVelocity` and mass in floating−point variable `Mass`. The acceleration doesn't require a variable because you're working it directly into the velocity in conjunction with the mass.

Also, you need to factor in time again (also stored as a floating−point variable, `Time`), which states how much velocity has built up over a period of time (measured in seconds) and how much velocity has been applied to the position of the body.

```
// vecVelocity = D3DXVECTOR3 object
// Mass = float variable
// vecForce = D3DXVECTOR3 object w/force to apply

// Scale force (which represents acceleration) by mass
// and add directly to velocity
vecVelocity += Time * (vecForce / Mass);

// Add velocity to position
vecPosition += Time * vecVelocity;
```

To recap what I've said so far: Combine all forces that you want to apply to a body into one net force vector. Scale this force vector by an object's mass, multiply it by the amount of time passed, and add the resulting vector to the object's velocity vector. Multiply the velocity by the same amount of time passed, and add the result to your position vector. Voila−linear movement!

Now it's time to graduate to the field of rotational motion.

Rotating Rigid Bodies

Much like linear movements, rotational motion uses velocity, acceleration, and momentum to determine the direction and speed your rigid body rotates. Unlike linear movement, which uses linear forces to determine the direction of movement, rotational forces use what's called *angular torque* (or just *torque*, for short) to determine how an object rotates according to the force applied. Torque, also defined as a vector, directly affects the *angular velocity*, which in turn affects the *angular momentum*.

The mass of an object affects how much of the velocity is applied to actually move the body, and it is also taken into consideration when you are rotating an object. This introduces the topic of inertia. *Inertia* determines the amount of force applied to rotate an object about its axes to the point at which the force is applied.

That's right; the same force that caused linear movement also causes angular rotation. The only difference here is the points where the various forces are applied to the body matter. As you are adding the various forces affecting your rigid body, you need to keep track of where those forces are applied and how they change your body's axis of rotation.

For example, push on one corner of the body and it spins in one direction; push another corner and the body spins in another direction. How do you know in which direction and how fast the body is spinning? Remember earlier in this chapter I talked about using quaternions for rotation? Well, guess what−you'll use quaternions to track the direction and angle of the objects! It will blow your mind when you see how handy a quaternion is here.

Assume you want to apply a force to your rigid body−a linear force that creates a torque. This torque is a directional vector as well, which, instead of pointing in the direction of movement, actually points in the direction of the angular axis (much like quaternions use angular axis vectors to orient objects). As you begin adding various forces, the torque vector might change direction, thus changing the axis as well. It works out great because the total amount of torque (the *net torque*) affecting the body is just a combination of all torque affecting the body, just like with linear forces.

So the question is, how do you convert force to torque? Take a look at Figure 7.7, which shows a simple rigid body. The arrow designates the linear force being applied to a point on the rigid body.

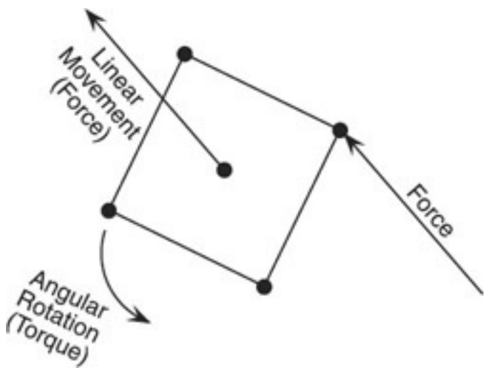


Figure 7.7: The force being applied affects not only linear movement, but also angular motion.
Of course the rigid body shown in Figure 7.7 is going to move in the same direction as the force vector being applied, but what about the rotation? It's definitely going to rotate because the force is hitting an off−center point on the body.

To calculate the torque vector, calculate a cross product between the linear force vector and a vector between the body's center and point where the force is being applied. This cross product just so happens to point in the direction of the rotation axis, just like your quaternions! Check out Figure 7.8 to see what I mean. Using the same rigid body as in Figure 7.7, I'm creating a cross product that defines the axis of rotation around which the object will spin.

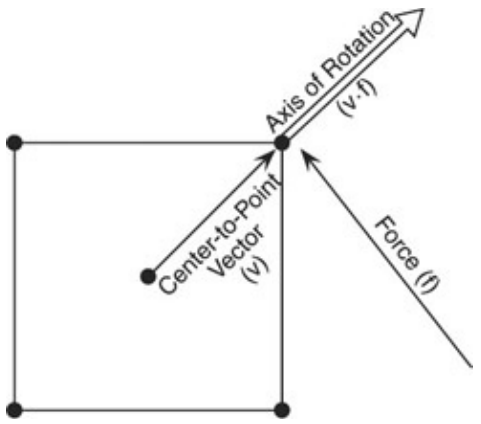


Figure 7.8: The force vector and the vector from the center to the point of application are used to compute a cross product that designates your axis of rotation.
Now that you know the axis of rotation, how do you know how much force to apply to make the body spin? Much like the mass of your object determined the amount of force applied to actually move the object, the body's inertia determines the amount of torque applied to make it rotate.

The inertia is actually split into three components, one for each axis of rotation (x, y, and z). These three components are called *inertia scalars*, and they are used to define three more values, which are referred to as the *moments of inertia*.

These moment of inertia values are analogous to your body's mass−the greater the value, the less torque applied to the body. The lower the mass of the rigid body, the more torque applied.

The values of your inertia scalars and moments of inertia are dependent on the shape and size of your rigid body. We've already decided to go with bounding boxes to represent your rigid bodies, which means you only need to worry about the width, depth, and height of the body. Since you're using a vector object (`vecSize`) to store the size of a rigid body, you can say the x component is width, the y component is height, and the z component is depth.

Using these three components (x, y, and z) as well as the mass of the object (`Mass`), you can define your inertia scalars as follows:

```
// X−axis inertia scalar
float xs = vecSize.x * vecSize.x;

// Y−axis inertia scalar
float ys = vecSize.y * vecSize.y;

// Z−axis inertia scalar
float zs = vecSize.z * vecSize.z;
```

You then compute the moment of inertia values using the three scalars you just created. These moment of inertia scalars represent the extents of your body along each axis, scaled by the mass of your body. For example, the moment of inertia for the x−axis is the combination of the y and z inertia scalars, multiplied by the mass of your body. Here's how to compute these three moment of inertia values:

```
// Ixx = moment of inertia for x−axis
float Ixx = Mass * (ys + zs);

// Ixx = moment of inertia for x−axis
float Iyy = Mass * (xs + zs);

// Ixx = moment of inertia for x−axis
float Izz = Mass * (xs + yz);
```

A collection of three moment of inertia tensors is known as a *moment of inertia tensor*, or an *inertia tensor* for short. The inertia tensor is made up using a 3x3 matrix, which takes the following form:

```
Ixx   0   0
  0 Iyy   0
  0   0 Izz
```

While the inertia tensor is an important aspect in determining how much torque to apply, you'll actually want to use the inverse of the inertia tensor (for reasons I'll cover in a moment). The inverse of the inertia tensor takes the following form:

```
1/Ixx     0     0
    0 1/Iyy     0
    0     0 1/Izz
```

You know all this inertia stuff is building up to something, don't you? Of course you do! After you have combined all the torque vectors affecting a body into one net torque vector, you add the resulting torque vector to the angular momentum of your object (taking time into consideration as well). This angular momentum value, just like torque, is specified in world coordinates. Unfortunately, you can't use world coordinates−you must convert them to coordinates local to the body.

This is one of the reasons I had you create an inverse inertia tensor matrix. You see, you must specify angular velocity in body space. (In other words, the body must rotate around its own origin, not the world's origin.) To convert from the coordinates of the angular momentum (which are in world coordinates) to the coordinates of the angular velocity (which are in body−space coordinates) and factor in the moments of inertia, you need to multiply the inverse inertia tensor by the orientation of your rigid body. You then multiply the resulting matrix by the transposed orientation of your rigid body, giving you a transformation that factors the world−and body−space conversions and the moments of inertia. With this final matrix, you can transform your angular momentum to an angular velocity vector.

You then use this angular velocity to compute the amount of rotation to apply to the rigid body. Much like linear velocity is scaled by a time factor to compute the total amount of velocity built up over time, angular momentum is scaled as well. The converted−to−angular velocity vector is scaled by time and applied to the angular rotation values of your quaternion (which represent the rotation of your body).

Okay, I've gone long enough without some code, so let me show you how to create the inverse inertia tensor matrix you'll be using, and how to use it to get your rigid body spinning.

```
// vecSize = vector w/size of rigid body's bounding box
// Mass = mass of body

// Compute the inertia scalars
float xScalar = vecSize.x * vecSize.x;
float yScalar = vecSize.y * vecSize.y;
float zScalar = vecSize.z * vecSize.z;

// Instance a matrix object and compute inverse inertia tensor
D3DXMATRIX matInvInertiaTensor;
D3DXMatrixIdentity(&matInvInertiaTensor);
matInvInertiaTensor._11 = 1.0f / (Mass * (yScalar + zScalar));
matInvInertiaTensor._22 = 1.0f / (Mass * (xScalar + zScalar));
matInvInertiaTensor._33 = 1.0f / (Mass * (xScalar + yScalar));
```

Now suppose you have a vector with your body's 3D coordinates and a quaternion that represents your rigid body's orientation. The first order of business is to apply a force to one of the points. How about applying a vector force of 10, 0, 40 to point 5 of the rigid body. Something to note first: This point is specified in world coordinates, so you'll use the position vector from the `vecWorldPoints` array in the following calculations.

```
// vecForce = vector w/force to apply (10,0,40 in this case)
// vecMomentum = angular momentum of body
// quatOrientation = quaternion w/orientation of object
// matInvInertiaTensor = inverse inertia tensor matrix

// Get the coordinates where force is applied (point 5)
D3DXVECTOR3 vecPos = vecWorldPoints[5];

// Compute a vector from point to center of body
D3DXVECTOR3 vecPtoC = vecPos − vecPosition;

// Calculate the cross product of the force vector
// and the vecPtoC vector. This vector is the torque.
D3DXVECTOR3 vecTorque;
D3DXVec3Cross(&vecTorque, &vecPtoC, &vecForce);

// Add torque to angular momentum
vecMomentum += vecTorque;

// Create a rotational transformation matrix from quaternion
D3DXMATRIX matOrientation;
D3DXMatrixRotationQuaternion(&matOrientation,               \
                              &quatOrientation);

// Transpose transform to make it left−handed
D3DXMatrixTranspose(&matOrientation, &matOrientation);

// Create a matrix to convert from world space to
// body space. This is used to calculate the angular velocity
D3DXMATRIX matConversion, matTransposedOrientation;
D3DXMatrixTranspose(&matTransposedOrientation, &matOrientation);
matConversion = matOrientation *                               \
                matInvInertiaTensor *                          \
                matTransposedOrientation;
```

You now have a transformation that converts from the angular momentum to angular velocity. You can apply this conversion (or rather, transformation) to the momentum to obtain the velocity as follows:

```
// Use conversion matrix to convert momentum to velocity
D3DXVec3TransformCoord(&vecAngularVelocity,                 \
                       &vecMomentum, &matConversion);
```

Getting back to working with time in the preceding equations, you need to multiply the torque applied to the momentum by the amount of time passed.

```
vecMomentum += (Time * vecTorque);
```

Using the momentum you just computed, calculate the angular velocity using that long block of code you just saw. With this angular velocity, you can then calculate the orientation, which is also scaled by time.

```
// Scale angular velocity by amount of time elapsed
D3DXVECTOR3 vecVelocity = Time * vecAngularVelocity;

// Apply velocity to orientation
quatOrientation.w −= 0.5f *
                     (quatOrientation.x * vecVelocity.x +
                      quatOrientation.y * vecVelocity.y +
                      quatOrientation.z * vecVelocity.z);
quatOrientation.x += 0.5f *
                     (quatOrientation.w * vecVelocity.x −
                      quatOrientation.z * vecVelocity.y +
                      quatOrientation.y * vecVelocity.z);
quatOrientation.y += 0.5f *
                     (quatOrientation.z * vecVelocity.x +
                      quatOrientation.w * vecVelocity.y −
                      quatOrientation.x * vecVelocity.z);
quatOrientation.z += 0.5f *
                     (quatOrientation.x * vecVelocity.y −
                      quatOrientation.y * vecVelocity.x +
                      quatOrientation.w * vecVelocity.z);
```

I'm computing orientation by factoring in the timed−based velocity. Remember that the velocity was computed from the momentum, and the momentum was computed from the torque. Torque contained a rotation axis, just like your quaternions, so the two go hand−in−hand. Basically, you're multiplying the velocity and the quaternion, and then cutting the resulting quaternion in half, as shown here:

$$q_1 = \frac{1}{2} vt\, q$$

Finally, you have managed to apply force to your rigid body, and you can calculate the angle and axis of rotation. You now have enough information to move and rotate your rigid bodies!

Okay, I'll admit that I was a little fast and loose in this section. My reasons were justified−I didn't want to bog you down with mounds of formulas and calculations. The field of rigid−body physics has been around a long time, and the resources are there for you to read. I think that more people don't use rigid−body physics because the math is so darn confusing. I want anybody to be able to understand the basics of rigid−body dynamics without having to go and grab a math textbook. Because of this, I skipped a lot of formulas that advanced readers might want to check out. As I mentioned, this field has been around a while now, and several great papers and articles have been written about it. A particularly helpful series of articles are Chris Hecker's Behind the Screen series on physics. You can check out those articles at Chris' home page, at http://www.d6.com/users/checker.

With that confession out of my system and out of the way, now comes the fun part−creating a bunch of forces to apply to your rigid bodies to make them move.

Using Forces to Create Motion

You should now be able to track the movement and rotation of a rigid body quite easily. It's when outside forces start affecting the movement that things get hairy. To keep things as simple as possible, I'm going to limit the forces affecting a rigid body to applied force, air resistance, gravity, and spring.

An applied force is one that you directly apply, such as the force from an explosion, a character pushing an object, or any type of force that doesn't apply to the other forces mentioned. As you can imagine, gravity pulls objects downward (or upward, if you want), whereas air resistance slows the motion of a rigid body as it moves. Spring force is used to connect rigid bodies to one another or to specific locations in your 3D world.

As I mentioned in the last two sections, your forces are stored in vector objects. Each force defines a direction of force to apply, as well as the amount of force to apply. This amount of force is stored as an acceleration value (in meters per second for an object with a mass of 1). For angular rotation, this value is the amount of radians per second. The magnitude of the force vector determines the amount of force. So for gravity, which accelerates objects around 9.8 m/s^2 in the negative y−axis (down), and taking mass into consideration, you can create a vector as follows:

```
D3DXVECTOR3 vecGravity = D3DXVECTOR3(0.0f, −9.8f, 0.0) * Mass;
```

How about an applied force, for example one that accelerates bodies (with a mass of 1) 10 meters per second in the positive x−axis?

```
D3DXVECTOR3 vecApplied = D3DXVECTOR3(10.0f, 0.0f, 0.0f);
```

And air resistance, how is that calculated? Air resistance isn't really a force that you directly define like the others; rather, it's created from another force. What I mean is that air resistance is an opposing force that is calculated by multiplying a body's velocity by a small amount (a negative amount, to be exact). To simulate air resistance in your rigid−body system, you can create two vectors that represent damping forces. These forces are used to slow the motion created by your linear velocity and angular momentum.

Assuming your rigid body's velocity is stored in a vector called `vecVelocity` and the angular momentum is stored in a vector called `vecMomentum`, you can create two vectors (a force and torque) to use.

```
D3DXVECTOR3 vecLinearDamping = vecVelocity * LinearDamping;
```

```
D3DXVECTOR3 vecAngularDamping = vecMomentum * AngularDamping;
```

`LinearDamping` and `AngularDamping` are floating−point variables that are negative in value. The higher the value, the more air resistance your bodies encounter and the more they slow down. Typical values that I use for `LinearDamping` and `AngularDamping` are 0.5 and 0.4, respectively.

Once you've computed two vectors that represent forces that oppose your linear and angular movement, you can apply them much as you did previously. In fact, you can just add the `vecLinearDamping` and `vecAngularDamping` vectors to your net force and torque vectors.

Hmm, you know what? I somehow managed to skip explaining spring forces. This wasn't an error; rather, I wanted to put off spring forces until this point, so I could better explain them. Springs connect your rigid bodies to one another, making sure the bodies that represent your character's bones stick to one another at the appropriate spots.

Connecting Rigid Bodies with Springs

Simulating the motion of a rigid body and the forces to apply is fairly simple once you understand the basic concepts. It's when you have multiple bodies connected to one another that things can get a little hairy. Each change in the orientation of a rigid body can force a change in other things. If you have many rigid bodies connected to one another, it could mean a ton of movements caused by a single motion to process.

Since I'm on the subject, these interconnected rigid bodies represent your rag doll character. You know the tune−*the hand bone's connected to the arm bone, the arm bone's connected to the.* Well, you get the idea. Each rigid body is connected to another body the same way each bone in your skeletal−based mesh is connected to its parent bone (except for the root bone, that is).

Whereas the skeletal structure connects bones via a frame hierarchy, your rigid bodies have no such luxury; those bodies are free, flying throughout your world during simulation. Something needs to hold them together, and that something is springs.

As you can see in Figure 7.9, you create a spring for each point where two bones in your source skeletal structure connect.

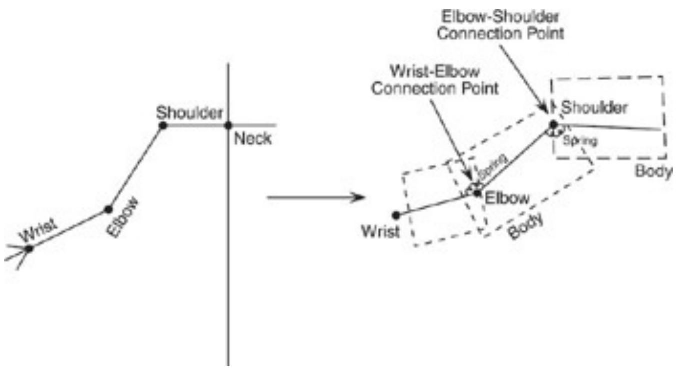


Figure 7.9: Springs connect a number of rigid bodies at the bone joint positions.

When your start moving your rigid bodies, you need to calculate the force that each spring exerts to maintain the overall shape of the body structure. Springs merely create forces that move each object in the appropriate direction. Unlike the forces you read about earlier, these springs need to bring the rigid bodies back together immediately, as opposed to over a period of time. This is a matter of directly modifying each body's position and angular momentum.

First things first, you need to know where these bones connect to one another. When you create each bounding box, you also need to add a point that determines where the box joins the box that represents the parent bone. You also add another point that represents the offset from the parent bone's center to where the bodies are connected. Figure 7.10 illustrates these two points.

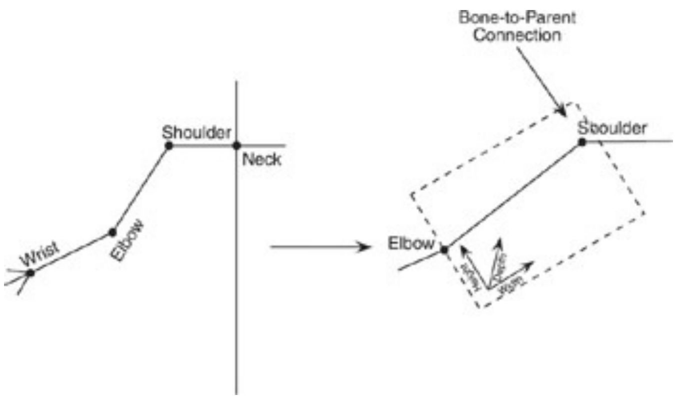


Figure 7.10: Each bone is defined by the size of its bounding box and the point where the bone connects to its parent bone.
Remember earlier in this chapter, when you defined eight vectors to represent the corners of the rigid body? There were two sets of eight vertices−one that defined in local coordinates and another that defined the same points transformed into world coordinates. Now you need to add two more points to this list.

The first point is the offset from the center of the bone to the point where the bone meets its parent bone. This point, called the *joint offset point* (or just the *joint offset*, for short), represents one end of the spring you are going to create to rejoin your rigid bodies after moving them.

The second point is the offset from the center of the bone's parent to the point where the bone meets the parent bone. This second point, called the *parent offset* point (or just the *parent offset*, for short), uses the transformation of the parent bone to orientate it, as opposed to using the bone's transformation (as you learned earlier). The second point represents the other end of the spring you are creating to rejoin the bones.

Because the second point you are adding uses the parent bone's transformation, it really doesn't need to be stored in the set of local−space vectors; it only needs to be stored in the set of transformation vectors. So that gives you two new sets of vectors to define.

```
D3DXVECTOR3 vecLocalPoints[9];
D3DXVECTOR3 vecWorldPoints[10];
```

You've already seen how to store the coordinates of the corner points in the vectors, so I'll cut to the chase and show you how to store the joint offset and parent offset vector values. Point number 8 will represent the joint offset coordinates, and point number 9 will represent the parent offset coordinates.

The joint offset vector is stored in a `D3DXVECTOR3` object called `vecJointOffset`, whereas the parent offset vector is stored in a `D3DXVECTOR3` object called `vecParentOffset`. The first thing to do is store the joint vector offset in the appropriate local space vector.

```
vecLocalPoints[8] = vecJointOffset;
```

You can now transform the nine points from local space to world space, as you did previously. From that point forward, point 8 will contain the world−space coordinates of where the bone joins to its parent.

As for the parent offset vector, you have to place it into one of the `vecWorldPoints` vectors. You use the transformation of the bone's parent to do this. If you have the orientation of the bone's parent stored as a transformation matrix called `matParentOrientation` and the position of the bone's parent stored as `vecParentPosition`, you can calculate the coordinates of the transformed point as follows:

```
D3DXVec3TransformCoord(&vecWorldPoints[9],           \
                       &vecParentOffset,             \
                       &matParentOrientation);
vecWorldPoints[9] += vecParentPosition;
```

Now that you've transformed the points into world−space coordinates, you can use those to create a spring, such as the one shown in Figure 7.11.

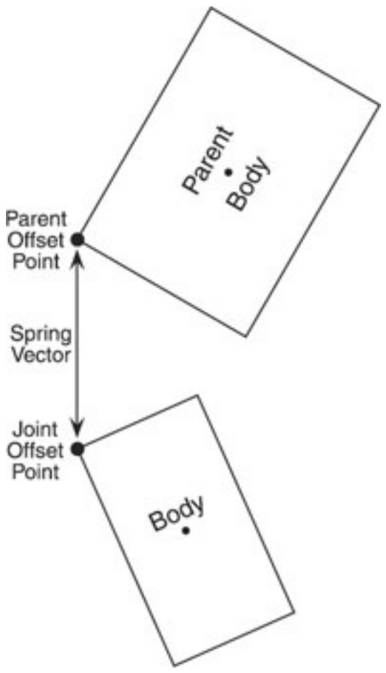


Figure 7.11: You create a spring vector by joining the joint offset point and the parent offset point. Both points are specified in world coordinates.

```
// vecPosition = position of body

// Get the position of the joint offset in world coordinates
D3DXVECTOR3 vecBonePos = vecWorldPoints[8];

// Get the vector from the point to the center of the body
// This is used to calculate the torque
D3DXVECTOR3 vecPtoC = vecPosition − vecBonePos;

// Get the position of the parent offset in world coordinates
D3DXVECTOR3 vecParentPos = vecWorldPoints[9];

// Calculate a spring vector from joint to parent offset points
D3DXVECTOR3 vecSpring = vecBonePos − vecParentPos;
```

The `vecSpring` vector now contains the difference in coordinates, which you can use to move the bone to connect to its parent. To emulate a real person, only a child bone will move to attach to the parent bone. (As an example, your arm would follow the movement of your chest.) So the root bone (which should present the character's chest or center of mass) will move, and all attached bones will scramble to stay connected to it (instead of the root bone moving to stay connected to the child bones).

Since you're only going to move the bone itself to reconnect it to its parent bone, you can add the `vecSpring` vector directly to the vector containing your body's position (thus ignoring spring constants). Also, you can calculate a cross product from the point's center to the joint offset position and the spring vector to alter the angular momentum of the bone, thus rotating it to the proper orientation to ensure the points of connection touch.

```
vecPosition += vecSpring;
D3DXVECTOR3 vecCross;
D3DXVec3Cross(&vecCross, &vecBtoC, &vecSpring);
vecAngularMoment += vecCross;
```

Now that the momentum has changed, you can transform it by the inverse inertia tensor of the bone and store the result in the angular velocity vector.

```
D3DXVECTOR3 vec
D3DXVec3TransformCoord(&vecAngularVelocity,            \
                       &vecAngularMomentum,            \
                       &matInvWorldInertiaTensor);
```

With all this talk about the joint offset and parent offset vectors, I forgot to mention where you actually get those vector values. When you get around to calculating the size of the bounding box, you need to consider the coordinates of the bone and the parent bone in world space.

The vector from the center of the bounding box to the coordinates of the bone is the joint offset. The offset from the parent's bone coordinates (which are transformed by the inverse of the bone's transformation) to the bone's coordinates is the parent offset vector. You'll learn more about this a bit later. For now, you've learned how to apply forces to your linear velocity and angular momentum, and how to force the bones to reconnect to one another. What's next? Process collisions, that's what!

Providing Collision Detection and Response

Because your rigid bodies are considered solid objects, they never change shape or penetrate the area of another object. Colliding with another object means the rigid body needs to react accordingly and change its movement and rotation as a result of the collision. This means you need to come up with a way to detect when one of your rigid bodies collides with another object, and how you'll process that collision accordingly. Take a close look at collision detection, and then we'll move on from there.

Testing for Collisions

Remember when I mentioned that using eight points for your rigid bodies is useful for more than just saving memory? If you haven't already guessed, those points help with collision detection as well! Because your rigid bodies are made up of eight points, you simply need to check each of those points to see if it penetrates another object. If one of the points does penetrate another object, you can adjust the motion of the rigid body to deflect away from the colliding object.

Of course, there are a number of other ways to detect collisions, usually involving face−to−point, point−to−point, and face−to−face collision checks, but I'm trying to keep things simple so your rag doll animations will be fast, which means simple point−to−plane and point−in−sphere distance checks will suffice.

In other words, for each of the eight points in a rigid body, you perform a quick check to see whether that point has passed through a plane (using a dot−product) or penetrated a sphere's radius (using a distance check). Each of these two checks are trivial; for a plane, you take each point's coordinates and perform a dot−product using the plane's normal and distance parameters.

```
// vecPos = D3DXVECTOR3 w/point's coordinates
// Plane = D3DXPLANE w/plane's data (normal and distance)

// Construct a vector from plane's data
D3DXVECTOR3 vecPlane = D3DXVECTOR3(Plane.a, Plane.b, Plane.c);

// Perform a dot product to get distance of point from plane
float Dist = D3DXVec3Dot(&vecPos, &vecPlane) + Plane.d;
```

Because the `d` parameter of the `D3DXPLANE` structure represents the distance of the plane from the origin, you can determine the distance from the point to the plane by adding that distance to the dot−product of the point's coordinates and the plane's normal, as I just showed you. If this distance value is equal to or less then 0, then the point is colliding with the plane.

Actually, the point is colliding with the plane if the distance is 0, and the point is penetrating the plane if the distance is less than 0. What's the different between colliding and penetrating? Well, you can have two types of collisions when you are working with rigid bodies−touching collisions and penetration collisions. A *touching collision* is one in which an object is just touching (within a tolerance) another object. This is when the objects would bounce off one another. A *penetrating collision* is one in which an object is embedded inside another object−clearly the result of the objects not bouncing off one another in time.

You are looking for touching collisions because you need to process a rebounding motion from them. For bodies that penetrate another object, you have two options−backtrack the motion of the body to find the exact moment of collision (not penetration), or push the body outside the space of the object and continue as if the point had merely collided (not penetrated).

As you can tell, backtracking time is the most accurate method. Unfortunately, once things really start bouncing around, things can slow down to a crawl while your system tries to find the exact moment of collision. While accuracy is something you want, it doesn't work out too well when speed is the key factor. For that reason, I'm only going to show you how to resolve collisions by pushing the penetrating points outside the area of the colliding object and continuing as if the point never penetrated.

Getting back to the task at hand, a simple distance check is all you need to determine whether a point is colliding with a sphere. Not just any distance check, however, because most distance checks perform a square−root operation. To speed things up, you can skip the square−root operation and instead work with squared values, including the distance of the point from the center of the sphere and the sphere's radius.

Let me give you an example. The position of the point is stored in `vecPos`, and the sphere's coordinates are stored in `vecSphere`. The radius of the sphere is stored in `Radius`. To check whether the point is within the sphere's radius, you can use the following code:

```
// vecPos = position of point to check
// vecSphere = coordinates of sphere's center
// Radius = float w/sphere's radius

// Get a vector from point to sphere's center
D3DXVECTOR3 vecDiff = vecSphere − vecPos;

// Get the squared distance from point to sphere
```

```
float Dist = vecDiff.x * vecDiff.x +
             vecDiff.y * vecDiff.y +
             vecDiff.z * vecDiff.z;

// Check if distance <= Radius of sphere squared
if(Dist < (Radius * Radius)) {
   // Point is within sphere!
}
```

As the code demonstrates, you are creating a vector from the sphere's center to the point in question. The squared distance of this vector is compared to the squared radius of the sphere. If the distance is less than or equal to the radius, then the point is inside the sphere.

After you've determined a point is penetrating an object (a sphere or plane), you need to push it outside of the object and then compute how the point rebounds off the object's surface. To push the point outside the plane or sphere, you need to take the object's normal (the normal of the plane or the normalized vector from the sphere's center to the point) and scale it according to how far the point has to be pushed out.

For a plane, the normal is stored in the a, b, and c components of the plane structure. Because these components are already normalized, you only need to scale the vector by the dot−product value you calculated to determine whether the point is colliding with the plane.

```
// Plane = plane structure to work with
// Dot = dot product of point's vector and plane normal
// vecPosition = position of point
D3DXVECTOR vecNormal = D3DXVECTOR3(Plane.a,
                                   Plane.b,
                                   Plane.c);

// Scale the normal by the dot product
vecNormal *= Dot;

// Add the scaled vector to the point's position
vecPosition += vecNormal;
```

To push a point outside a sphere, you need to calculate the difference between the radius of the sphere and the distance of the point from the center of the sphere. This means you need to compute the square root of the distance value you've already calculated when checking for penetration. In the following code, you calculate the square root of the distance and use the sphere's radius to compute the amount to scale the vector you created from the sphere's center to the point.

```
// vecDiff = vector from point to sphere
// Radius = radius of sphere
// Dist = squared distance from point to sphere's center

// Normalize the vector
D3DXVec3Normalize(&vecDiff, &vecDiff);

// Get the real distance value by calculating the
// square root of the Dist variable
Dist = (float)sqrt(Dist);
// Subtract new Dist value from Radius to get
// distance to push point outside sphere
float ToPush = Radius − Dist;

// Scale normalized vector using ToPush
vecDiff *= ToPush;
```

```
// Add the scaled vector to point's position
vecPosition += vecDiff;
```

So now that you know how to perform a distance check against planes and spheres, and you can use the results of this test to tell whether an object has collided with another object (and push it outside the object if it penetrates too far), what's left to do? Well, you have to bounce the point off the object's surface.

Reacting to Collisions

As you can tell, collision detection is a somewhat trivial task at this point−you're more interested in the collision response. Once you've determined that a rigid body has collided with another object, you need to figure out how that collision has affected the motion of your object.

Just like other forces (such as gravity and springs) alter the motion of your rigid bodies, so does collision response. Like a spring force, collision response creates a force that instantaneously alters your body's position and rotation.

That's right, Luke; you're going to ignore the force for now and instead work with impulse! Much like the dark side of the force (I hope I'm not wasting metaphors on non−*Star Wars* fans), *impulse* is sort of the fast lane to power. Instead of taking the long and slow road of applying force to an object and having it slowly move in response to a collision, impulse immediately moves an object away from the point of collision.

Calculating this impulse power is just like calculating a force to apply. For collisions, the calculation involves determining the speed at which the point is approaching the collision object's surface and using the surface's normal, calculating how much of an impulse force is acting against the motion of the rigid body.

You've already got the normal of the surface off which the body was bouncing, and the speed the body is traveling is a combination of the linear and angular velocities. All that's left to determine is how much of that velocity is used to create a rebounding force. Using what's called the *coefficient of restitution*, you can scale the colliding object's normal by a specific amount based on the colliding point's velocity, and then use the resulting vector as the force to rebound a rigid body.

The coefficient of restitution is a scalar value that determines how much bounce a body has when colliding with an object (actually, how much power is dissipated during the collision).

The lower the coefficient value used, the less a body bounces off an object. The higher the value, the more bounce. In fact, your bodies can even gain energy when bouncing off objects if the coefficient is greater than 1. The coefficient of restitution is stored as a floating−point variable, which I'll refer to as `Coefficient`.

Things are going to get crazy because the formulas involved in calculating the amount of velocity applied to bounce the point are a bit involved. As I previously mentioned, you first need to calculate the velocity of the point using the body's linear and angular velocities. To do this, create two vectors−one that represents a vector from the center of the body to the point of collision and one that represents the linear velocity added to the cross product of the previous vector and the angular velocity.

```
// vecPosition = coordinates of rigid body
// vecPointPos = coordinates of colliding point
// vecCollisionNormal = normal of colliding object

// Get vector from center of body to point of collision
D3DXVECTOR3 vecPtoP = vecPosition − vecPointPos;
```

```
// Get cross product from vecPtoP to angular velocity.
D3DXVECTOR3 vecCross;
D3DXVec3Cross(&vecCross, &vecAngularVelocity, &vecPtoP);

// Add the two vectors to get the point's velocity
D3DXVECTOR3 vecPointVelocity = vecLinearVelocity + vecCross;
```

You now have the velocity vector of the colliding point stored in `vecPointVelocity`. Using this vector, you compute an impulse force that is applied directly to the rigid body's position and angular momentum. To calculate the impulse force, you must first calculate an impulse numerator and denominator to scale the colliding object's normal. To keep this following bit of code readable, I'm going to use two functions called `DotProduct` and `CrossProduct` that work inline with the code to calculate the dot−product of two vectors and the cross product, respectively.

```
// Calculate the impulse numerator
real ImpulseNumerator = DotProduct(vecPointVelocity,           \
                                   vecCollisionNormal) *       \
                        −(1.0f + Coefficient);

real ImpulseDenominator = (1.0f / Mass) +                      \
          DotProduct(CrossProduct(matInvWorldInertiaTensor *  \
          CrossProduct(vecPtoP, vecCollisionNormal),           \
          vecPtoP), CollisionNormal);
```

As if you couldn't tell, the `ImpulseNumerator` and `ImpulseDenominator` calculations are extremely complicated. Although I want to devote the time and space to fully explaining them, I just can't bear to do so. Entire papers have been written on these two calculations, and I'll leave it up to these readily accessible papers to better explain them. To check out one such paper, written by Chris Hecker for Game Developer magazine, go to Chris's homepage at http://www.d6.com/users/checker and download his Collision Response paper.

For now, just apply these two values to your collision normal to calculate an impulse vector used to bounce the colliding point off the object the point hit.

```
D3DXVECTOR3 vecImpulse = vecCollisionNormal *                \
                     (ImpulseNumerator/ImpulseDenominator);
```

You now have the vector that represents the opposing force to apply to the colliding body. You can add this velocity vector, `vecImpulse`, directly to the linear velocity and angular momentum of your rigid body.

```
// Add forces to running total
vecLinearVelocity += vecImpulse;

// Calculate cross product to create torque vector
D3DXVECTOR3 vecCross;
D3DXVec3Cross(&vecCross, &vecPtoP, &vecImpulse);
vecAngularMomentum += vecCross;
```

However, there's one problem that I didn't mention. What about points that hit the same collision object at the same time? If you process the points one at a time and adjust their positions and rotation according to the collisions, you'll find that bodies parallel to a collision object will bounce off in unrealistic ways.

To compensate for this fact, you should process every point of the rigid body first, keeping track of all impulse forces to apply to the linear velocity and angular momentum. After you've worked through all the points, you can average the impulse vectors by the number of collisions detected and add the results to the velocity and momentum.

Storing Collision Object Data

With all this talk about collision objects, I failed to mention how you are going to store the various bits of information that define these objects. Since I'm keeping it simple and working only with planes and spheres, you can create a couple of classes that contain a single collision object and an array of objects, respectively.

The first object, which contains the information about a single collision object, is called `cCollisionObject`.

```
class cCollisionObject {
   public:
      DWORD       m_Type;      // Type of object

      D3DXVECTOR3 m_vecPos;    // Sphere coordinates
      float       m_Radius;    // Sphere radius
      D3DXPLANE   m_Plane;     // Plane values

      cCollisionObject *m_Next; // Next in linked list

   public:
      cCollisionObject()  { m_Next = NULL;                }
      ~cCollisionObject() { delete m_Next; m_Next = NULL; }
};
```

You will read about collision objects in much greater detail in Chapter 13, "Simulating Cloth and Soft Body Mesh Animation." Why am I waiting that long to explain collision objects, you ask? Well, thanks to the modern miracle of copy editing (and my ever−changing mind), I moved around some information. I won't let that stop you from checking out how to store collision objects, however, so let's quickly go over this class.

Inside the `cCollisionObject` class, you can see a `D3DXPLANE` object that defines a plane, whereas a vector object and a floating−point value define a sphere's position and radius. Essentially, the `cCollisionObject` class can contain information about a plane or a sphere. That's the reason for the m_Type variable, which defines the object defined in the class−either `COLLISION_SPHERE` if the data defines a sphere, or `COLLISION_PLANE` if the data defines a plane.

The `m_Next` pointer points to the next collision object in the list of collision objects loaded. That's right; the collision objects are stored using a linked list. For that reason, I included a constructor and destructor that handle the initialization and release of the linked list pointer.

Speaking of more than one collision object, I'd like to introduce the second collision object class−`cCollision`.

```
class cCollision {
   public:
      DWORD            m_NumObjects;  // # of objects
      cCollisionObject *m_Objects;     // Object list

   public:
      cCollision();
      ~cCollision();

      void Free();
      void AddSphere(D3DXVECTOR3 *vecPos, float Radius);
      void AddPlane(D3DXPLANE *PlaneParam);
};
```

The `cCollision` class maintains the linked list of collision objects. You can add objects using the `AddSphere` and `AddPlane` functions, or you can clear out the linked list by calling `Free`. The parameters to the `AddSphere` and `AddPlane` functions are fairly evident; you specify the center and radius of a sphere to add, or the parameters of a plane to add to the linked list.

You can skip to Chapter 13 to check out more detailed information about collision objects, or you can check out the source code on this book's CD−ROM (located in the Collision.cpp and Collision.h files in the Chapter 7 directory).

For now, let me demonstrate how to create a plane and a sphere object to use during your simulation.

```
// Instance a collision collection object
cCollision Collision;

// Add a ground plane at origin of world
Collision.AddPlane(&D3DXPLANE(0.0f, 1.0f, 0.0f, 0.0f));

// Add a sphere at 0,10,40 with a radius of 20
Collision.AddSphere(&D3DXVECTOR3(0.0f, 10.0f, 40.0f), 20.0f);
```

Well, my friend, I do believe you've learned everything you need to know to create your own rag doll animation system! I'm going to hit top speed for the remainder of this chapter, showing you how to take what you've learned up to this point and apply it to a series of classes you'll use in your rag doll animation system. Make sure you fully understand everything you've read up to this point because things are going to move quickly!

Creating a Rag Doll Animation System

Now that you've learned the inner secrets of rigid body physics, what is left to do? Create your own rag doll animation system, that's what! A rag doll is nothing more than a series of linked rigid bodies that are constructed from your character's bones. By creating a series of classes to contain these rigid bodies and a class to control their motion, you can have your very own rag doll animation system up and running in no time!

But I want to take this whole rag doll animation system bit by bit, starting with how to define the state of a single rigid body.

Defining the Rigid Body State

Earlier in this chapter you learned how you can define a rigid body using a series of vectors that define the body's position, orientation, velocity, momentum, and so on. These vectors define the current state of a rigid body, which you can store in a structure as follows:

```
class cRagdollBoneState
{
   public:
      D3DXVECTOR3    m_vecPosition;        // Position
      D3DXQUATERNION m_quatOrientation;    // Orientation
      D3DXMATRIX     m_matOrientation;     // Orientation

      D3DXVECTOR3    m_vecAngularMomentum; // Angular momentum

      D3DXVECTOR3    m_vecLinearVelocity;  // Linear velocity
      D3DXVECTOR3    m_vecAngularVelocity; // Angular velocity

      // Transformed points, including connection−to−parent
      // position and parent−to−bone offset
      D3DXVECTOR3    m_vecPoints[10];

      // Body's inverse world moment of inertia tensor matrix
      D3DXMATRIX     m_matInvWorldInertiaTensor;
};
```

Aptly named, `cRagdollBoneState` stores the position, orientation (quaternion and matrix), linear velocity, angular velocity, and momentum, as well as the inverse inertia tensor (in world coordinates) and the transformed points. These transformed points include the offset from the rigid body to the parent body and the offset from the parent's body to the current body.

As you process your simulation, the state of the bone is updated according to the forces applied. For the remaining information about your bones, such as the size, mass, and so on, you can define a second class to contain the pertinent information.

Containing Bones

The remaining bone data, such as the bone's source frame, size, mass, coefficient of restitution, force, and torque (just to name a few things), is stored in another class. This class, `cRagdollBone`, is defined as follows:

```
class cRagdollBone
{
   public:
      // Frame that this bone is connected to
      D3DXFRAME_EX *m_Frame;

      // Size of bounding box
      D3DXVECTOR3 m_vecSize;

      // Mass and 1/Mass (one−over−mass)
      float m_Mass;

      // Coefficient of restitution value
      // 0 = no bounce
      // 1 = 'super' bounce
      // >1 = gain power in bounce
      float m_Coefficient;

      cRagdollBone *m_ParentBone;      // Pointer to parent bone

      // Connection−to−parent offset and
      // parent−to−bone offset
      D3DXVECTOR3 m_vecJointOffset;
      D3DXVECTOR3 m_vecParentOffset;

      // Linear force and angular momentum
      D3DXVECTOR3 m_vecForce;
      D3DXVECTOR3 m_vecTorque;

      // Original orientation of bone
      D3DXQUATERNION m_quatOrientation;

      // Rate of resolution (0−1) to resolve slerp interpolation
      // This is used to make bones return to their initial
      // orientation relative to its parent.
      float m_ResolutionRate;

      // Body's inverse moment of inertia tensor matrix
      D3DXMATRIX m_matInvInertiaTensor;

      // Points (in body space) that form bounding box
      // and connection−to−parent offset position
      D3DXVECTOR3 m_vecPoints[9];

      // Bone state
      cRagdollBoneState m_State;
};
```

The comments for each of the class' members are pretty self−explanatory, and you've read about most of the data in this chapter. The only things with which you might not be familiar are the quaternion value stored here, the resolution rate, and the way the points and bone state are used.

The quaternion (`m_quatOrientation`) stored in `cRagdollBone` represents the relative difference in rotation from the parent bone. This is useful for maintaining the overall shape of the character when it undergoes simulation. Instead of having to painstakingly define the extent that each bone can rotate in relation to its parent, you can force the orientation of each bone to be restored to its original orientation. I'll show you how to do this in a bit; for now, I want to get back to the `cRagdollBone` class.

You can see in `cRagdollBone` that I've embedded a state class, which represents the state of the bone as it undergoes simulation. There are also nine points defined, which represent the local coordinates of each corner point, as well as the point where the bone connects to its parent.

Okay, I'm skipping a lot of details here, but I'll get back to the `cRagdollBone` class in a bit. For now, I want to show you how a collection of these bone classes is maintained by a class that controls all aspects of your rag doll animation.

Creating the Rag Doll Controller Class

The third and final class you'll use to control your rag doll animation system is a big one, so I'm going to show it to you bit by bit. To begin, you have a pointer to the frame hierarchy used to create the rigid bodies in your simulation. Also, you have the number of bones in your bone hierarchy and an array of `cRagdollBone` class objects to contain the information about each bone.

```
class cRagdoll
{
   protected:
      D3DXFRAME_EX      *m_pFrame;   // Frame hierarchy root

      DWORD              m_NumBones; // # bones
      cRagdollBone      *m_Bones;    // Bone list
```

So far this class doesn't look so intimidating, so what's the big fuss? Believe me, there's a lot more to come. Next in line are the protected functions.

```
   protected:
      // Function to compute a cross product inline
      D3DXVECTOR3 CrossProduct(D3DXVECTOR3 *v1, D3DXVECTOR3 *v2);
```

```
    // Function to multiply a vector by a 3x3 matrix
    // and add an optional translation vector
    D3DXVECTOR3 Transform(D3DXVECTOR3 *vecSrc,
                          D3DXMATRIX *matSrc,
                          D3DXVECTOR3 *vecTranslate = NULL);

    // Get a frame's bone bounding box size and joint offset
    void GetBoundingBoxSize(D3DXFRAME_EX *pFrame,
                            D3DXMESHCONTAINER_EX *pMesh,
                            D3DXVECTOR3 *vecSize,
                            D3DXVECTOR3 *vecJointOffset);

    // Build a bone and set its data
    void BuildBoneData(DWORD *BoneNum,
                       D3DXFRAME_EX *Frame,
                       D3DXMESHCONTAINER_EX *pMesh,
                       cRagdollBone *ParentBone = NULL);

    // Set gravity, damping, and joint forces
    void SetForces(DWORD BoneNum,
                   D3DXVECTOR3 *vecGravity,
                   float LinearDamping,
                   float AngularDamping);

    // Integrate bone motion for a time slice
    void Integrate(DWORD BoneNum, float Elapsed);

    // Process collisions
    DWORD ProcessCollisions(DWORD BoneNum,
                            cCollision *pCollision,
                            D3DXMATRIX *matCollision);

    // Process bone connections
    void ProcessConnections(DWORD BoneNum);

    // Transform the state's points
    void TransformPoints(DWORD BoneNum);
```

Things are really starting to stack up, with nine protected functions to sift through. To start, there are two functions (`CrossProduct` and `Transform`) that you use to compute a cross product and transform a vector using a transformation matrix and an optional translation vector. Why not just use the `D3DX` functions, you ask? Well, the `CrossProduct` and `Transform` functions use `D3DX` to work their magic, but I wanted to be able to calculate the cross product and transform vectors inline with other code, as the following bit of code demonstrates:

```
D3DXVECTOR3 vecResult = Transform(&CrossProduct(&vec1,   \
                                                &vec2),  \
                                  &matTransform);
```

You use the third protected function, `GetBoundingBoxSize`, to calculate the bounding−box size for a bone. This bounding box encases all vertices attached to the bone, as well as the points where a bone connects to its parent and child bones (if any). The next function is `SetForces`, which you call to set up the initial forces for a bone−gravity and linear and angular damping.

Next in the list of functions is `Integrate`, which processes the force and torque applied to the bone, updates the velocities and momentum, and moves the rigid body's points into their new positions based on the body's motion. Then you have `ProcessCollisions` (which handles your collision detection and response), `ProcessConnections` (which ensures all bones are connected to one another at the joints), and `TransformPoints` (which transforms your local points into world−space coordinates using the orientation and position of the bone, as stored in the bone's state class object `m_State`).

You will find the full source code to the `cRagdoll` class, as well as the other classes defined here, in the Ragdoll.cpp and Ragdoll.h files on this book's CD−ROM. The functions shown so far duplicate everything you've read in this chapter, so there's no need to explain the code here. Well, except for the `GetBoundingBoxSize` and `Integrate` functions. I'll get back to those functions in a bit; for now, I want to keep things rolling by showing you the rest of the functions in the `cRagdoll` class.

```
   public:
      cRagdoll();
      ~cRagdoll();

      // Create ragdoll from supplied frame hierarchy pointer
      BOOL Create(D3DXFRAME_EX *Frame,
                  D3DXMESHCONTAINER_EX *Mesh,
                  D3DXMATRIX *matInitialTransformation = NULL);

      // Free ragdoll data
      void Free();

      // Resolve the ragdoll using gravity and damping
      void Resolve(float Elapsed,
                   float LinearDamping   = −0.04f,
                   float AngularDamping  = −0.01f,
                   D3DXVECTOR3 *vecGravity  = &D3DXVECTOR3(0.0f, −9.8f, 0.0f),
                   cCollision *pCollision   = NULL,
                   D3DXMATRIX *matCollision = NULL);
      // Rebuild the frame hierarchy
      void RebuildHierarchy();

      // Functions to get the number of bones and
      // retrieve a pointer to a specific bone
      DWORD GetNumBones();
      cRagdollBone *GetBone(DWORD BoneNum);
};
```

Aside from the class's constructor and destructor functions, which are used to clear out and release the class's data, there are six functions for you to tackle. You use the first function, `Create`, to set up the rag doll class's data. This entails iterating through each bone in the frame hierarchy provided and creating a rigid body bone object (a `cRagdollBone` object) for each. These bones are then transformed to match the position and orientation of each frame. Once you have created them, you can free the rag doll data by calling `Free`.

In between your calls to `Create` and `Free`, the function you'll deal with most is `Resolve`. The `Resolve` function takes the number of seconds to process, the amount of linear and angular damping to apply, the gravity vector to use, and a pointer to an array of collision objects used to check for collisions.

After you've resolved a portion of your simulation using the `Resolve` function, you call `RebuildHierarchy` to update your frame hierarchy. After you update your frame hierarchy, you can update your skinned mesh and render away!

The last two functions tell you how many bones are contained in the rigid body and allow you to grab a pointer to a specific bone. These two functions are handy if you want to directly access a bone to obtain its position or velocity.

Now that you've seen the three classes you're going to be working with, take a closer look at the functions you haven't read about in this chapter, starting with the function that builds the data pertinent to each bone in your mesh's frame hierarchy.

Building Bone Data

I don't want to beat a dead horse, so I'm only going to touch on those bits of bone data I haven't already discussed. Aside from a bone's position and orientation, the rate of rotation resolution determines how hard your mesh tries to maintain its initial shape over time. The lower the rate of resolution, the more your body can contort. The higher the resolution, the more resistant your mesh becomes to contortion.

I'll talk about this resolution rate in a bit; for now, I want to talk about other things, such as calculating the initial orientation of each bone, the size of each bone's bounding box, a bone's local points that form the corners of the rigid body's bounding box, the coefficient of restitution, and parent offset vectors.

You've already read about most of these variables, but I haven't mentioned anything about calculating the bounding−box size or the initial orientation of each bone. I'll start with how to calculate the beginning orientation.

Remember way back in the "Positioning and Orienting Your Rigid Bodies" section, when I told you that you can use a quaternion to represent your rigid body's orientation? If you are using a frame hierarchy as the source for building your rag doll's bones, you have to convert from a transformation matrix to a quaternion transformation. How the heck do you do that?

Here comes `D3DX` to the rescue! Using the frame's combined transformation matrix, you can call the `D3DXQuaternionRotationMatrix` function, which conveniently converts a transformation matrix to a quaternion transformation! For example, suppose you have a frame pointed to by the `D3DXFRAME_EX` pointer `pFrame`. To create the quaternion, use the following code bit:

```
// pFrame = pointer to frame
// quatOrientation = resulting quaternion
D3DXQuaternionRotationMatrix(&quatOrientation,              \
                               &m_Frame->matCombined);
D3DXQuaternionInverse(&quatOrientation, &quatOrientation);
```

You'll notice that while converting the transformation matrix to a quaternion, I added a call to `D3DXQuaternionInverse`, which inverses the quaternion values. The reason why is that quaternions are defined using a right−handed system. Since we're using a left−handed system with Direct3D, this quaternion needs to be appropriately converted (inversed).

Now that you have an initial orientation to work with, you can compute the size of your bone's bounding box; create a bunch of points to represent the corners of the bounding box and connection points (the points that connect the bone to its parent); set the mass and coefficient of restitution; and create the inverse inertia tensor. The source code I'm talking about is located in the `cRagdoll::Create` function. Please consult the heavily commented source code to see how to set the appropriate data during creation of the rag doll's bones.

For now, let me explain how to calculate the size of your bounding boxes.

Computing the Bone Bounding Box

When creating the rigid body bone class object that represents each bone, the first thing you need to do is grab the bone's inverse transformation matrix using the skinned mesh's `ID3DXSkinInfo::GetBoneOffsetMatrix` function. This inverse bone transformation matrix takes your skinned mesh's vertices and orients them around the mesh's origin (as opposed to the frame's origin).

Remember back in Chapter 4, when I explained how the vertices in your skinned mesh must be located about the mesh's origin to be properly rotated around the bone's origin? The whole process of transforming vertices consisted of applying the inverse transformation followed by the bone's rotation and translation transformations combined with the frame's parent transformation.

Once you have a bone's inverse transformation matrix, you can iterate through each vertex that is attached to the bone. What you're going to do is transform those vertices using the bone's inverse transformation. Using the coordinates of these newly transformed vertices, you can compute the extents of your bounding box (which will eventually enclose each vertex and bone−to−bone connection point).

The `cRagdoll::GetBoundBoxSize` function calculates this bounding box. The function takes a pointer to a frame structure (which represents the bone), as well as two vectors that will contain the size of the bounding box and the offset from the bounding box's center to the point where the bone connects to the parent bone.

```
void cRagdoll::GetBoundingBoxSize(D3DXFRAME_EX *pFrame,
                                  D3DXVECTOR3 *vecSize,
                                  D3DXVECTOR3 *vecJointOffset)
{
```

I'll explain the size and offset vectors in a bit. For now, start your `GetBoundingBoxSize` function by creating and clearing out a couple of vectors that contain the coordinates of your bounding box's extents and will eventually be used to create the eight corner points of your rigid body.

```
   // Set default min and max coordinates
   D3DXVECTOR3 vecMin = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
   D3DXVECTOR3 vecMax = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
```

The first order of business in `GetBoundingBoxSize` is to find the bone that matches the frame's name. This bone, or rather the skinned mesh bone interface object (`ID3DXSkinInfo`), queries which vertices are connected to the bone.

```
      // Only process bone vertices if there is a bone to work with
      if(pFrame−>Name) {

         // Get a pointer to ID3DXSkinInfo interface for
         // easier handling.
         ID3DXSkinInfo *pSkin = pMesh−>pSkinInfo;

         // Search for a bone by same name as frame
         DWORD BoneNum = −1;
         for(DWORD i=0;i<pSkin−>GetNumBones();i++) {
            if(!strcmp(pSkin−>GetBoneName(i), pFrame−>Name)) {
              BoneNum = i;
              break;
            }
         }

         // Process vertices if a bone was found
         if(BoneNum != -1) {
```

After you've found an `ID3DXSkinInfo` for the bone in question, you query it for the number of vertices attached and allocate an array of `DWORD` and `float` values to hold the vertex indices and weights.

```
    // Get the number of vertices attached
    DWORD NumVertices = pSkin->GetNumBoneInfluences(BoneNum);
    if(NumVertices) {

      // Get the bone influences
      DWORD *Vertices = new DWORD[NumVertices];
      float *Weights  = new float[NumVertices];
      pSkin->GetBoneInfluence(BoneNum, Vertices, Weights);
```

Now that the vertex indices are stored in the `Vertices` buffer (which you accomplished by calling `GetBoneInfluence`), you can begin iterating through each vertex, transforming the vertices by the bone's inverse transformation and using the transformed vertices to calculate the size of the bounding box.

```
      // Get stride of vertex data
      DWORD Stride = D3DXGetFVFVertexSize(                  \
                          pMesh->MeshData.pMesh->GetFVF());

      // Get bone's offset inversed transformation matrix
      D3DXMATRIX *matInvBone =                              \
                          pSkin->GetBoneOffsetMatrix(BoneNum);

      // Lock vertex buffer and go through all of
      // the vertices that are connected to bone
      char *pVertices;
      pMesh->MeshData.pMesh->LockVertexBuffer(              \
                     D3DLOCK_READONLY, (void**)&pVertices);
      for(i=0;i<NumVertices;i++) {

        // Get pointer to vertex coordinates
        D3DXVECTOR3 *vecPtr =                               \
                 (D3DXVECTOR3*)(pVertices+Vertices[i]*Stride);

        // Transform vertex by bone offset transformation
        D3DXVECTOR3 vecPos;
        D3DXVec3TransformCoord(&vecPos, vecPtr, matInvBone);

        // Get min/max values
        vecMin.x = min(vecMin.x, vecPos.x);
        vecMin.y = min(vecMin.y, vecPos.y);
        vecMin.z = min(vecMin.z, vecPos.z);

        vecMax.x = max(vecMax.x, vecPos.x);
        vecMax.y = max(vecMax.y, vecPos.y);
        vecMax.z = max(vecMax.z, vecPos.z);

      }
      pMesh->MeshData.pMesh->UnlockVertexBuffer();

      // Free resource
      delete [] Vertices;
      delete [] Weights;
      }
    }
  }
```

At the end of this bit of code, you'll have the extents of the bounding box stored in the two vectors (`vecMin` and `vecMax`) you instanced at the beginning of the function. The array of vertex indices is freed (as well as the vertex weights), and processing continues by factoring in the point where the bone connects to its parent and child bones.

```
  // Factor in child bone connection points to size
  if(pFrame->pFrameFirstChild) {

    // Get the bone's inverse transformation to
    // position child connections.
    D3DXMATRIX matInvFrame;
    D3DXMatrixInverse(&matInvFrame,NULL,&pFrame->matCombined);

    // Go through all child frames connected to this frame
    D3DXFRAME_EX *pFrameChild =                             \
                  (D3DXFRAME_EX*)pFrame->pFrameFirstChild;
    while(pFrameChild) {
      // Get the frame's vertex coordinates and transform it
      D3DXVECTOR3 vecPos;
      vecPos = D3DXVECTOR3(pFrameChild->matCombined._41,
                           pFrameChild->matCombined._42,
                           pFrameChild->matCombined._43);
      D3DXVec3TransformCoord(&vecPos, &vecPos, &matInvFrame);

      // Get min/max values
      vecMin.x = min(vecMin.x, vecPos.x);
      vecMin.y = min(vecMin.y, vecPos.y);
      vecMin.z = min(vecMin.z, vecPos.z);

      vecMax.x = max(vecMax.x, vecPos.x);
      vecMax.y = max(vecMax.y, vecPos.y);
      vecMax.z = max(vecMax.z, vecPos.z);

      // Go to next child bone
      pFrameChild = (D3DXFRAME_EX*)pFrameChild->pFrameSibling;
    }
  }
```

To factor in the connection points, you basically grab the world−space coordinates of the connected bones and transform them by the bone's inverse transformation. These coordinates are then compared to the coordinates stored in the `vecMin` and `vecMax` vectors.

You can now finish the function by storing the size of the box. If the box is too small, set the size to a minimum amount (as defined by the `MINIMUM_BONE_SIZE` macro, which is set to 1.0).

```
  // Set the bounding box size
  vecSize->x = (float)fabs(vecMax.x − vecMin.x);
  vecSize->y = (float)fabs(vecMax.y − vecMin.y);
  vecSize->z = (float)fabs(vecMax.z − vecMin.z);

  // Make sure each bone has a minimal size
  if(vecSize->x < MINIMUM_BONE_SIZE) {
    vecSize->x = MINIMUM_BONE_SIZE;
    vecMax.x = MINIMUM_BONE_SIZE*0.5f;
```

```
   }
   if(vecSize->y < MINIMUM_BONE_SIZE) {
      vecSize->y = MINIMUM_BONE_SIZE;
      vecMax.y = MINIMUM_BONE_SIZE*0.5f;
   }

   if(vecSize->z < MINIMUM_BONE_SIZE) {
      vecSize->z = MINIMUM_BONE_SIZE;
      vecMax.z = MINIMUM_BONE_SIZE*0.5f;
   }

   // Set the bone's offset to center based on half the size
   // of the bounding box and the max position
   (*vecJointOffset) = ((*vecSize) * 0.5f) - vecMax;
}
```

At the very end of the function, you finally encounter the `vecJointOffset` vector object that I mentioned when you started creating the `GetBoundingBoxSize` function. Because a rigid body bone can be any size, and you track the bone by its center coordinates, you need to create an extra point that represents the point in the bounding box where the bone connects to its parent. This is the *joint offset vector*. You'll read more about the joint offset vector when you enforce the bone−to−bone connections.

Now that you've computed the bounding−box size and set the various bones' data, you can set the various forces and resolve the motion of your bones.

Setting the Forces

For any applied forces you want to use (which I'll leave up to you), you must handle gravity and damping. In the `cRagdoll` class, I defined a function that clears out a single bone's force and torque vectors and then applies gravity and damping forces.

```
void cRagdoll::SetForces(DWORD BoneNum,
                         D3DXVECTOR3 *vecGravity,
                         float LinearDamping,
                         float AngularDamping)
{
   // Get a pointer to the bone for easier handling
   cRagdollBone *Bone = &m_Bones[BoneNum];

   // Get pointer to the current state for easier handling
   cRagdollBoneState *BCState = &Bone->m_State;

   // Set gravity and clear torque
   Bone->m_vecForce = ((*vecGravity) * Bone->m_Mass);
   Bone->m_vecTorque = D3DXVECTOR3(0.0f, 0.0f, 0.0f);

   // Apply damping on force and torque
   Bone->m_vecForce += (BCState->m_vecLinearVelocity *     \
                        LinearDamping);

   Bone->m_vecTorque += (BCState->m_vecAngularVelocity *   \
                         AngularDamping);
}
```

You read about gravity and damping forces earlier in this chapter, so I shouldn't need to explain anything here. You'll notice that I'm scaling the gravity vector by the mass of the bone. Remember that this is necessary so that gravity will pull all objects with the same force when you later scale the forces due to mass.

Once you have set these forces, you can resolve (integrate) the motion of the specified bone.

Integrating the Bones

Once you have set the bone's force and torque, you can use those vectors to resolve the motion of your rigid−body bone. You'll notice that up to this point I've been working with a single bone at a time. It is perfectly fine to move the bones one by one and eventually join them before you render the mesh.

Back in the "Processing the Motion of Rigid Bodies" section, you saw how to apply the force and torque vectors to the linear velocity and angular momentum to create motion. In the `cRagdoll::Resolve` function, I duplicate what you read in that section.

You'll notice that the position, orientation, velocity, and momentum values are stored in a bone state class object, `cRagdollBoneState`. These vectors are used during resolution. To call `Integrate`, you need to specify which bone to resolve, as well as the time to resolve (how much time has elapsed).

```
void cRagdoll::Integrate(DWORD BoneNum, float Elapsed)
{
   // Get pointer to bone
   cRagdollBone *Bone = &m_Bones[BoneNum];

   // Get pointers to states for easier handling
   cRagdollBoneState *State = &Bone->m_State;
```

After you get a pointer to the bone class object (as well as a pointer to the bone's state object), you can calculate the new position of the bone based on the linear velocity, the change in angular movement based on the torque, and the change in linear velocity based on the amount of force (scaled by the mass of the object).

```
   // Integrate position
   State->m_vecPosition += (Elapsed*State->m_vecLinearVelocity);

   // Integrate angular momentum
   State->m_vecAngularMomentum += (Elapsed * Bone->m_vecTorque);

   // Integrate linear velocity
   State->m_vecLinearVelocity += Elapsed * Bone->m_vecForce /   \
                                 Bone->m_Mass;
```

You now compute the new orientation (stored in the quaternion) using the angular velocity multiplied by the amount of time passed.

```
// Integrate quaternion orientation
   D3DXVECTOR3 vecVelocity = Elapsed * State->m_vecAngularVelocity;
   State->m_quatOrientation.w -= 0.5f *
                      (State->m_quatOrientation.x * vecVelocity.x +
                       State->m_quatOrientation.y * vecVelocity.y +
                       State->m_quatOrientation.z * vecVelocity.z);
   State->m_quatOrientation.x += 0.5f *
                      (State->m_quatOrientation.w * vecVelocity.x −
                       State->m_quatOrientation.z * vecVelocity.y +
                       State->m_quatOrientation.y * vecVelocity.z);
   State->m_quatOrientation.y += 0.5f *
                      (State->m_quatOrientation.z * vecVelocity.x +
                       State->m_quatOrientation.w * vecVelocity.y −
                       State->m_quatOrientation.x * vecVelocity.z);
   State−>m_quatOrientation.z += 0.5f *
                      (State−>m_quatOrientation.x * vecVelocity.y −
                       State−>m_quatOrientation.y * vecVelocity.x +
                       State−>m_quatOrientation.w * vecVelocity.z);

   // Normalize the quaternion (creates a unit quaternion)
   D3DXQuaternionNormalize(&State−>m_quatOrientation,
                           &State−>m_quatOrientation);
```

Up to this point, I haven't addressed how you're going to stop your rag doll mesh's bones from contorting out of control. Think about it−because your bones are rigid bodies, they can rotate in any direction by any amount, which can cause your characters' heads to rotate through their chest, for instance. This is why I introduced the use of a rotation resolution factor in the declaration of each rag doll bone object.

After you've resolved the new orientation of a bone, you need to slowly bring it back to its initial orientation. You can accomplish this by pre−computing the difference in orientation from the bone to its parent. Using this pre−computed difference, you then compute the orientation that the bone should try to match.

The higher the rate of resolution you set, the faster the bone will try to match its initial orientation relative to its parent's orientation. To reorient the bone and match its initial orientation, you can slerp (spherically interpolate) from the bone's current orientation to the initial orientation relative to the parent's orientation. The amount of interpolation is−you guessed it−the amount you set in the resolution rate variable. Higher values force the bone not to rotate, whereas lower values make the bone slowly (or never) return to its initial orientation.

```
   // Force rotation resolution
   if(BoneNum && Bone−>m_ResolutionRate != 0.0f) {

      // Slerp from current orientation to beginning orientation
      D3DXQUATERNION quatOrientation =                         \
             Bone−>m_ParentBone−>m_State.m_quatOrientation *   \
             Bone−>m_quatOrientation;
      D3DXQuaternionSlerp(&State−>m_quatOrientation,           \
                       &State−>m_quatOrientation,              \
                       &quatOrientation,                       \
                        Bone−>m_ResolutionRate);
}
```

Moving on, the remaining code in the `Integrate` function creates a transformation matrix that later transforms your rigid body bones' points and creates the angular velocity. Because you're working with left−handed coordinate systems, you must transpose the transformation matrix after you create it, using `D3DXMatrixRotationQuaternion`. This step was discussed earlier in this chapter.

```
   // Compute the new matrix−based orientation transformation
   // based on the quaternion just computed
   D3DXMatrixRotationQuaternion(&State−>m_matOrientation,
                                &State−>m_quatOrientation);
   D3DXMatrixTranspose(&State−>m_matOrientation,
                       &State−>m_matOrientation);

   // Calculate the integrated inverse world inertia tensor
   D3DXMATRIX matTransposedOrientation;
   D3DXMatrixTranspose(&matTransposedOrientation, &State−>m_matOrientation);
   State−>m_matInvWorldInertiaTensor = State−>m_matOrientation *
                                       Bone−>m_matInvInertiaTensor *
                                       matTransposedOrientation;
   // Calculate new angular velocity
   State->m_vecAngularVelocity = Transform(&State->m_vecAngularMomentum,
                                           &State->m_matInvWorldInertiaTensor);
}
```

At this point, your bone's motion has been resolved, and it is time to process any collisions.

Processing Collisions

You'll recall from earlier in this chapter that collisions are handled by checking each of the rigid−body bone's points against a list of collision objects. If a point is located within a collision object's space, then the point is pushed out and the impulse vectors are averaged between all collisions so that the body is rebounded off the surface of the object in a realistic manner. This is the purpose of the `ProcessCollisions` function.

I won't list the full code for the `ProcessCollisions` function code here; rather, I'll just lightly touch on it. The `ProcessCollisions` function starts off with its function prototype, which takes the bone number you are checking, as well as a pointer to the root collision object to check for collisions.

```
BOOL cRagdoll::ProcessCollisions(DWORD BoneNum,             \
                                 cCollision *pCollision)
{
   // Error checking
   if(!pCollision || !pCollision->m_NumObjects ||           \
      !pCollision->m_Objects)
    return TRUE;

   // Get a pointer to the bone for easier handling
   cRagdollBone *Bone = &m_Bones[BoneNum];

   // Get a pointer to the state for easier handling
   cRagdollBoneState *State = &Bone->m_State;

   // Keep count of number of collisions
   DWORD CollisionCount = 0;

   // Keep tally of collision forces
   D3DXVECTOR3 vecLinearVelocity  = D3DXVECTOR3(0.0f,0.0f,0.0f);
   D3DXVECTOR3 vecAngularMomentum = D3DXVECTOR3(0.0f,0.0f,0.0f);
```

So `ProcessCollisions` starts off by getting a pointer to the bone object and a pointer to the bone's state object. From there, you must keep track of the number of collisions detected (to later average out the motions) and define two vectors that define the built−up impulses to move and rotate the bone according to any collisions.

Moving on in the code, you start scanning through the eight points of the bone's rigid−body object. For each point, you also scan through each collision object. Before you check for any point−to−object collisions, you need to define a flag that determines three things: whether the point collided with the object, the normal of the object, and the distance the point penetrates into the object.

```
// Go through all bone vertices looking for a collision
   for(DWORD i=0;i<8;i++) {

      // Loop through all collision objects
      cCollisionObject *pObj = pCollision->m_Objects;
      while(pObj) {
         // Flag if a collision was detected
         BOOL Collision = FALSE;

         // Normal of collision object
         D3DXVECTOR3 vecCollisionNormal;

         // Distance to push point out of collision object
         float CollisionDistance = 0.0f;

         // Process sphere collision object
         if(pObj->m_Type == COLLISION_SPHERE) {
```

I'm going to cut the code off here because you've already seen how to compute whether a collision occurs, how to compute the normal, and how to calculate the distance in which the point penetrates the object. If you browse the full source code, you'll see that I'm checking for point−to−sphere collisions and point−to−plane collisions.

Once you have performed the collision detection, you should have the `Collision` flag set to `TRUE` if there was a collision or `FALSE` if there was no collision. If the flag is `TRUE`, then the point is pushed outside the object, and the proper impulse vectors are computed.

```
      // Process a collision if detected
        if(Collision == TRUE) {

          // Push the object onto the collision object's surface
          State->m_vecPosition += (vecCollisionNormal *         \
                                   CollisionDistance);

          // Get the point's position and velocity
          D3DXVECTOR3 vecPtoP = State->m_vecPosition -          \
                                State->m_vecPoints[i];
          D3DXVECTOR3 vecPtoPVelocity =                         \
                                State->m_vecLinearVelocity +    \
                   CrossProduct(&State->m_vecAngularVelocity,   \
                                                   &vecPtoP);   \

          // Get the point's speed relative to the surface
          float PointSpeed = D3DXVec3Dot(&vecCollisionNormal,   \
                                         &vecPtoPVelocity);

          // Increase number of collisions
          CollisionCount++;

          // Calculate the impulse force based on the coefficient
          // of restitution, the speed of the point, and the
          // normal of the colliding object.
          float ImpulseForce = PointSpeed *                      \
                               (-(1.0f + Bone->m_Coefficient));
          float ImpulseDamping = (1.0f / Bone->m_Mass) +         \
                               D3DXVec3Dot(&CrossProduct(        \
                           &Transform(&CrossProduct(&vecPtoP,    \
                               &vecCollisionNormal),             \
                           &State->m_matInvWorldInertiaTensor),  \
                               &vecPtoP), &vecCollisionNormal);
          D3DXVECTOR3 vecImpulse = vecCollisionNormal *          \
                                 (ImpulseForce/ImpulseDamping);

          // Add forces to running total
          vecLinearVelocity += vecImpulse;
```

```
            vecAngularMomentum += CrossProduct(&vecPtoP, &vecImpulse);
         }
```

Following the last bit of code, the rest of the collision objects are checked and the remaining points are processed. In the end, if any collisions were detected, the `ProcessCollisions` function averages the impulse vectors and applies them to the linear velocity and angular momentum.

```
   // Was there any collisions
   if(CollisionCount) {

      // Add averaged forces to integrated state
      State->m_vecLinearVelocity  +=  ((vecLinearVelocity /    \
                     Bone->m_Mass) / (float)CollisionCount);
      State->m_vecAngularMomentum += (vecAngularMomentum /     \
                                     (float)CollisionCount);

      // Calculate angular velocity
      State->m_vecAngularVelocity = Transform(                 \
                          &State->m_vecAngularMomentum,        \
                          &State->m_matInvWorldInertiaTensor);
}
```

The next function I want to show you will help you enforce the bone−to−bone connections by pulling the bones back together to enforce the shape of your rag doll character.

Enforcing Bone−to−Bone Connections

After you have resolved all your forces and handled the collisions, there's one final step to perform before you rebuild your frame hierarchy and render your rag doll character's mesh. You must resolve the bone−to−bone connections.

You'll recall that in the "Connecting Rigid Bodies with Springs" section, you learned how to create a spring between bones and use it to calculate a force vector that immediately moves and rotates a body into the proper position. That's just what the `ProcessConnections` function does−it takes the coordinates of the transformed point that represents the joint offset (the offset from the center of the bone to where the bone joins with its parent) and the coordinates of where it connects to the parent (stored as another transformed point) to create the spring.

```
void cRagdoll::ProcessConnections(DWORD BoneNum)
{
   // Get a pointer to the bone and
   // parent bone for easier handling
   cRagdollBone *Bone = &m_Bones[BoneNum];
   cRagdollBone *ParentBone = Bone->m_ParentBone;

   // Don't continue if there's no parent bone
   if(!ParentBone)
      return;

   // Get the pointer to the bone's state
   cRagdollBoneState *BState = &Bone->m_State;

   // Get pointer to parent's state
   cRagdollBoneState *PState = &ParentBone->m_State;
   // Get joint connection position and vector to center
   D3DXVECTOR3 vecBonePos = BState->m_vecPoints[8];
   D3DXVECTOR3 vecBtoC    = BState->m_vecPosition - vecBonePos;

   // Get the parent connection point coordinates
   D3DXVECTOR3 vecParentPos = BState->m_vecPoints[9];

   // Calculate a spring vector from point to parent's point
   D3DXVECTOR3 vecSpring = vecBonePos - vecParentPos;

   // Move point to match parent's point and adjust
   // the angular velocity and momentum
   BState->m_vecPosition -= vecSpring;
   BState->m_vecAngularMomentum -= CrossProduct(&vecBtoC,     \
                                                &vecSpring);
   BState->m_vecAngularVelocity = Transform(                  \
                     &BState->m_vecAngularMomentum,
                     &BState->m_matInvWorldInertiaTensor);
}
```

I know I'm moving quickly, but I have already explained all the code up to this point; I just wanted to reiterate some vital points. The next and last function that I want to show you rebuilds the hierarchy, allowing you to update your skinned mesh.

Rebuilding the Hierarchy

After resolving the rigid−body simulation, the last bit of processing you perform every frame is rebuilding the frame's transformations. Because the rigid−body bones in your rag doll are specified in world coordinates, there's no need to go through each bone in the hierarchy and combine each transformation with the bone's parent transformation. Basically, all you need to do is copy the bone's orientation (from the state object) to the frame's combined transformation matrix.

The only problem is that each bone is oriented and translated about the rigid body's center coordinates, not the coordinates of the joint connection point as defined by the `vecJointOffset` vector. You need to transform the `vecJointOffset` vector by the orientation of the bone and add the resulting vector to the position of the bone, which will give you the proper coordinates to position the bone. The comments in `RebuildHierarchy` should explain things nicely.

```
void cRagdoll::RebuildHierarchy()
{
   if(!m_pFrame)
      return;
   if(!m_NumBones || !m_Bones)
      return;

   // Apply bones' rotational matrices to frames
   for(DWORD i=0;i<m_NumBones;i++) {

      // Transform the joint offset in order to position frame
      D3DXVECTOR3 vecPos;
      D3DXVec3TransformCoord(&vecPos,
                             &m_Bones[i].m_vecJointOffset,
                             &m_Bones[i].m_State.m_matOrientation);

      // Add bone's position
      vecPos += m_Bones[i].m_State.m_vecPosition;
      // Orient and position frame
      m_Bones[i].m_Frame->matCombined = m_Bones[i].m_State.m_matOrientation;
      m_Bones[i].m_Frame->matCombined._41 = vecPos.x;
      m_Bones[i].m_Frame->matCombined._42 = vecPos.y;
      m_Bones[i].m_Frame->matCombined._43 = vecPos.z;
   }
}
```

You've reached the end of the line! All that's left is to put your rag doll animation class to good use in your own projects. To do so, you simply need to instance a `cRagdoll` class object, call `Create` to build the data, and continuously call `Resolve` and `RebuildHierarchy`. The demo for this chapter shows you how easy this is, so I highly recommend you go through the demo's source code before continuing.

Check Out the Demo

As you can tell, rag doll animation systems are nothing more than rigid−body simulations in disguise. Why pay thousands of dollars for a rag doll animation system when you've seen how easy it is to create your very own? The demo for this chapter shows you one such rag doll animation system you can use in your projects.

This demo, shown in Figure 7.12, takes a sample character and tosses it through the air. This doesn't sound too exciting, so to spice things up, a bunch of collision objects (spheres) are littered about, causing the poor character to flop like a limp noodle.

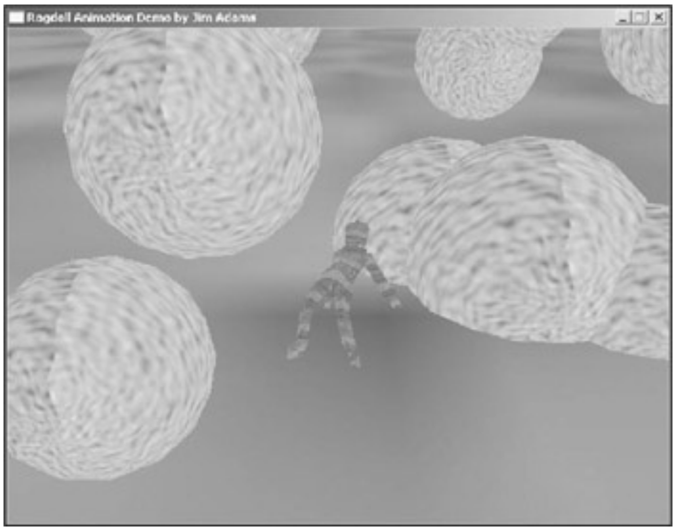


Figure 7.12: A wooden dummy meets a gruesome death, flying through the air and bouncing off a bunch of floating spheres.

The field of rigid−body physics is certainly an exciting one; if you find yourself craving a bit more information than I provided here, the resources are definitely out there.

Programs on the CD

There is only one project for Chapter 7, but believe me, it's a whopper! You can find the project in the Chapter 7 directory of this book's CD−ROM:

- **Ragdoll.** This project demonstrates rag doll animation by showing you what happens when a character is thrown through a field of floating spheres. It is located at `\BookCode\Chap07\Ragdoll.`

Part Four: Morphing Animation

Chapter 8: Working with Morphing Animation

With a loud clank, the wall across the compound begins to slide down. The grinding of stone mixes with the ferocious roars of unseen evil creatures that lurk just beyond the ever−shrinking barrier. I can only wonder what horrors are about to be unleashed upon me. As the wall settles, the onslaught begins. Wave after wave of pixelated demons pour forth, only to meet their doom at the hands of my trusty pulse rifle.

With demons running, teeth gnashing, and bits of flesh flying, I'm sure animation techniques are not at the top of your list of priorities. But if you *did* have to think about animation, I'm sure images of complex skeletal structures, frame hierarchies, and skinned meshes would come to mind, wouldn't they? Would it surprise you to find out that all the animations for the previously mentioned onslaught might be brought to you through the use of a morphing animation system? That's right−with games such as those in id's *Doom* and *Quake* series, those little demonic critters are animated by morphing animation, a technique that ensures smooth and easy playback on even the lowliest of systems. Morphing animation is so simple to use that you'll wonder why you didn't look into it sooner. Don't fret any longer, because this chapter is here to help you!

Morphing in Action

Back in the early 90s, a revolutionary computer−graphics animation technique known as *morphing* hit the big league and was brought into the mainstream, thanks to a man known as Michael Jackson. No, I'm not referring to one of his plastic surgery fiascos−rather, the use of morphing in one of his music videos. Yep, the King of Pop used morphing techniques in his video for the song "Black or White" and created an animation phenomenon that continues to this day.

In case you haven't seen the video, let me explain. It includes a segment in which a person is grooving to the tune, and the camera is centered on that person's face. Every few seconds, the person's face morphs into another person's face. This continues while the face morphs more than 10 times. The results of the morphing in the video are incredible, and I still remember them clearly to this day!

As the years rolled by, morphing eventually made its way to gaming. Whereas the older days of morphing involved digitally editing video footage to smoothly change one image into another (such as in "Black or White"), nowadays morphing (or at least the morphing we're going to discuss here) involves the smooth change of 3D meshes over time.

Probably the most popular example of morphing in gaming has got to be with id's *Quake.* In *Quake,* all of the characters' animation sequences are constructed from a series of morphing meshes. One mesh slowly changes shape to a second mesh, the second mesh changes shape to match a third mesh, and so on.

Spaced over a short period of time and using enough meshes from which to morph, all animation sequences are smooth and extremely easy to process. Even the lowliest of computer systems can run *Quake* decently. That's because morphing animation is extremely easy to work with, as you'll see in the next few chapters.

So as you can surmise, morphing−or *tweening,* as it is sometimes referred to (such as in the DirectX SDK)−is the process of changing one shape into another over time. For you, those shapes are meshes. The process of morphing a mesh involves slowly changing the coordinates of the mesh vertices, starting at one mesh's shape and progressing to another.

The mesh that contains the orientation of the vertices at the beginning of the morphing cycle is called the *source mesh.* The second mesh, which contains the orientation of the vertices at the end of the morphing cycle, is called the *target mesh.* Take a closer look at these two meshes to better understand the whole morphing process.

Defining Source and Target Meshes

The source and target meshes you'll deal with in this book are everyday `ID3DXMesh` objects. However, you can't use just any two meshes for a morphing operation; there are some rules to follow. First, each mesh must share the same number of vertices. The morphing operation merely moves vertices from the source mesh positions to match the target mesh positions. This brings up the second rule: Each vertex in the source mesh must have a matching vertex (that is, a matching index number) in the target mesh. Take the meshes shown in Figure 8.1 as an example.

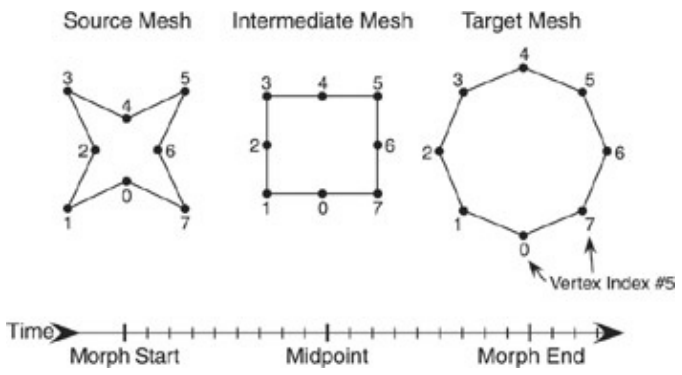


Figure 8.1: During the morphing process, the vertices in the source mesh gradually move to match the positions of the target mesh. Each vertex shares the same index number in both the source and target meshes.
Vertex ordering is important here. For a vertex to move from the source position to the target position, it must share the same index number. If you were to renumber the order, the vertices would move in the wrong direction while morphing and produce some funky−looking results such as those shown in Figure 8.2

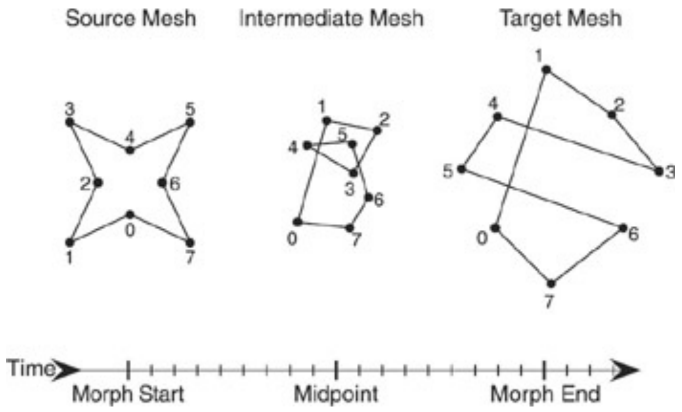


Figure 8.2: Morphing gone bad−the vertex order differs between the source and target meshes, producing some odd results.
As long as you design the meshes to have the same number of vertices and so that the vertex ordering matches up, you'll do just fine. As for getting the actual mesh data, I'll leave that in your capable hands. You can use the `D3DXLoadMeshFromX` function, or feel free to use the functions you developed in Chapter 1 to load your meshes. After you've got two valid meshes loaded and ready to use, you can begin morphing them!

Morphing the Meshes

Now that you have two meshes to work with (the source and target meshes), you can begin the morphing process. Remember that morphing is the technique of changing one shape to another. You want the vertices in the source mesh, in their initial positions, to gradually move to match the positions of the target mesh's vertices.

You can measure the time period used to track the motion of the vertices from the source mesh coordinates to the target mesh coordinates using a scalar value (ranging from 0 to 1). With a scalar value of 0 the vertices will be positioned at the source mesh coordinates, whereas with a scalar value of 1 the vertices will be positioned at the target mesh coordinates. As Figure 8.3 illustrates, any scalar value between 0 and 1 will place the vertices somewhere between the source mesh and target mesh coordinates.

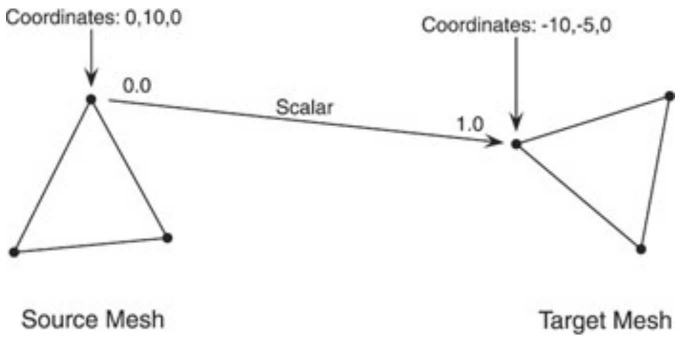


Figure 8.3: Starting at the source mesh coordinates (and a scalar value of 0), a vertex gradually moves toward the target mesh coordinates as the scalar value increases.

It is quite simple to calculate the coordinates in which to position a vertex between the source mesh coordinates and target mesh coordinates. Take a vertex from the source mesh and multiply the vertex's coordinates by the inversed scalar value (1.0−scalar). Using the inversed scalar means that the original vertex coordinates will use 100 percent of the vertex's coordinate position when the scalar is at 0.0, and zero percent of the vertex's coordinate position when the scalar is at 1.0.

Next, using the same indexed vertex's coordinates from the target mesh, multiply the vertex's coordinates by the scalar value. Adding the two resulting values gives you the final coordinates to use for the vertex during the morphing animation.

At first, this concept of multiplying the vertex coordinates by a scalar value and adding the results together might seem strange. If you're unsure of the math, perform the following calculations to see that the results are indeed correct. Use a one−dimensional value to represent the vertex coordinates. Set the source vertex coordinate to 10 and the target vertex coordinate to 20. To make things easy, use a scalar value of 0.5, which should give you a resulting morphing vertex coordinate of 15.

Multiplying the source coordinate (10) by 1−0.5 gives you 5. Multiplying the target coordinate (20) by 0.5 gives you 10. Adding the two results (5 and 10) gives you 15. Isn't that special−it comes out to the correct value after all!

This procedure would resemble the following code, assuming that the vertex's source coordinates are stored in `vecSource,` the target coordinates are stored in `vecTarget,` and the scalar is stored in `Scalar.`

```
// vecSource = D3DXVECTOR3 w/source coordinates
// vecTarget = D3DXVECTOR3 w/target coordinates
// Scalar = FLOAT w/scalar value

// Multiply source coordinates by inversed scalar
D3DXVECTOR3 vecSourcePos = vecSource * (1.0f−Scalar);

// Multiply target coordinates by scalar
D3DXVECTOR3 vecTargetPos = vecTarget * Scalar;
```

```
// Add the two resulting vectors together
D3DXVECTOR vecPos = vecSourcePos + vecTargetPos;
```

After that last bit of code, the `vecPos` vector will contain the coordinates to use for positioning a vertex during the morphing animation. Of course, you would repeat the same calculations for each vertex in the source mesh. In the next section, you'll get to see how to perform these calculations to build your own morphing meshes. Before that, however, I'd like to mention something about timing your morphing animations.

Up to this point, I've been ignoring the factor of time−both the length of the animation cycle (how long the morphing animation takes to progress from the source coordinates to the target coordinates) and exactly what point in time you are sampling the coordinates from in the animation sequence. Assuming you are measuring time in milliseconds, with the animation's length stored as `Length` (a `FLOAT` value) and the time at which you are sampling the coordinates stored as `Time` (also a `FLOAT` value), you can compute a proper scalar value to use in your calculations as follows:

```
Scalar = Time / Length;
```

Using the calculations you've already seen in this section, you can use the scalar value to compute the coordinates of the vertices to build your morphing mesh.

That's the second time I've mentioned building a morphing mesh, so I won't keep delaying things. Read on to see how to build a morphing mesh you can use to render. Although I previously hinted at using a vertex shader, first I will show you the easiest way to build a morphing mesh−by manipulating the mesh's vertex buffers directly.

Building a Morphed Mesh through Manipulation

Directly manipulating a mesh's vertex buffers is probably the easiest way to work with morphing. For this method you'll need a third mesh that contains the final coordinates of each vertex after morphing; it's this third mesh that you'll render.

To create the third mesh, which I call the *resulting morphed mesh*, you can clone the source mesh and be on your way.

```
//  Declare third mesh to use for resulting morphed mesh
ID3DXMesh *pResultMesh = NULL;

//  Clone the mesh using the source mesh pSourceMesh
pSourceMesh−>CloneMeshFVF(0, pSourceMesh−>GetFVF(),         \
                          pDevice,&pResultMesh);
```

After you've created the resulting morphed mesh (`pResultMesh`), you can begin processing the morphing animation by locking the source, target, and resulting morphed mesh's vertex buffers. Before you do that, however, you need to declare a generic vertex structure that contains only the vertex coordinates, which you'll use to lock and access each vertex buffer.

```
typedef struct {
  D3DXVECTOR3 vecPos;
} sGenericVertex;
```

Also, because each vertex buffer contains vertices of varying sizes (for example, the source might use normals whereas the target doesn't), you need to calculate the size of the vertex structure used for each mesh's vertices. You can do so using the `D3DXGetFVFVertexSize` function.

```
// pSourceMesh = source mesh object
// pTargetMesh = target mesh object
// pResultMesh = resulting morphed mesh object
DWORD SourceSize = D3DXGetFVFVertexSize(pSourceMesh−>GetFVF());
DWORD TargetSize = D3DXGetFVFVertexSize(pTargetMesh−>GetFVF());
DWORD ResultSize = D3DXGetFVFVertexSize(pResultMesh−>GetFVF());
```

Now you can lock the vertex buffers and assign the pointers to them.

```
// Declare vertex pointers
char *pSourcePtr, *pTargetPtr, *pResultPtr;
pSourceMesh−>LockVertexBuffer (D3DLOCK_READONLY,               \
                              (void**)&pSourcePtr);
pTargetMesh−>LockVertexBuffer (D3DLOCK_READONLY,               \
                              (void**)&pTargetPtr);
pResultMesh−>LockVertexBuffer (0, (void**)&pResultPtr);
```

Notice how I assigned a few `char *` pointers to the vertex buffers instead of using the generic vertex structure? You need to do that because the vertices in the buffers could be of any size, remember? Whenever you need to access a vertex, you cast the pointer to the generic vertex structure and access the data. To go to the next vertex in the list, add the size of the vertex structure to the pointer. Get it? If not, don't worry−the upcoming code will help you make sense of it all.

After you've locked the buffers you can begin iterating through all vertices, grabbing the coordinates and using the calculations from the previous section to calculate the morphed vertex positions. Assuming that the length of the animation is stored in `Length` and the current time you are using is stored in `Time,` the following code will illustrate how to perform the calculations:

```
//  Length = FLOAT w/length of animation in milliseconds
//  Time = FLOAT w/time in animation to use

//  Calculate a scalar value to use for calculations
float Scalar = Time / Length;

// Loop through all vertices
for(DWORD i=0;i<pSourceMesh−>GetNumVertices();i++) {

// Cast vertex buffer pointers to a generic vertex structure
sGenericVertex *pSourceVertex = (sGenericVertex*)pSourcePtr;
sGenericVertex *pTargetVertex = (sGenericVertex*)pTargetPtr;

sGenericVertex *pResultVertex = (sGenericVertex*)pResultPtr;

//  Get source coordinates and scale them
D3DXVECTOR3 vecSource = pSourceVertex−>vecPos;
vecSource *= (1.0f − Scalar);

//  Get target coordinates and scale them
D3DXVECTOR3 vecTarget = pTargetVertex−>vecPos;
vecTarget *= Scalar;

//  Store summed coordinates in resulting morphed mesh
pResultVertex−>vecPos = vecSource + vecTarget;
//  Go to next vertices in each buffer and continue loop
pSourcePtr += SourceSize;
pTargetPtr += TargetSize;
pResultPtr += ResultSize;
}
```

Up to this point I've skipped over the topic of vertex normals because normals are identical to vertex coordinates in that you use scalar and inversed scalar values on the normals to perform the same calculations as you do for the vertex coordinates.

In the preceding code, you can calculate the morphing normal values by first seeing whether the mesh uses normals. If so, during the loop of all vertices you grab the normals from both the source and target vertices, multiply by the scalar and inversed scalar, and store the results. Take another look at the code to see how to do that:

```
// Length = FLOAT w/length of animation in milliseconds
// Time = FLOAT w/time in animation to use

// Calculate a scalar value to use for calculations
float Scalar = Time / Length;

// Set a flag if using normals
BOOL UseNormals = FALSE;
if(pSourceMesh->GetFVF() & D3DFVF_NORMAL &&       \
   pTargetMesh->GetFVF() & D3DFVF_NORMAL)
  UseNormals = TRUE;

// Loop through all vertices
for(DWORD i=0;i<pSourceMesh->GetNumVertices();i++) {

   // Cast vertex buffer pointers to a generic vertex structure
   sGenericVertex *pSourceVertex = (sGenericVertex*)pSourcePtr;
   sGenericVertex *pTargetVertex = (sGenericVertex*)pTargetPtr;

sGenericVertex *pResultVertex = (sGenericVertex*)pResultPtr;

// Get source coordinates and scale them
D3DXVECTOR3 vecSource = pSourceVertex->vecPos;
vecSource *= (1.0f - Scalar);

// Get target coordinates and scale them
D3DXVECTOR3 vecTarget = pTargetVertex->vecPos;
vecTarget *= Scalar;

// Store summed coordinates in resulting morphed mesh
pResultVertex->vecPos = vecSource + vecTarget;

// Process normals if flagged
if(UseNormals == TRUE) {
   // Adjust generic vertex structure pointers to access
   // normals, which are next vector after coordinates.
   pSourceVertex++; pTargetVertex++; pResultVertex++;

   // Get normals and apply scalar and inversed scalar values
   D3DXVECTOR3 vecSource = pSourceVertex->vecPos;
   vecSource *= (1.0f - Scalar);
   D3DXVECTOR3 vecTarget = pTargetVertex->vecPos;
   vecTarget *= Scalar;
   pResultVertex->vecPos = vecSource + vecTarget;
```

```
 }

 // Go to next vertices in each buffer and continue loop
 pSourcePtr += SourceSize;
 pTargetPtr += TargetSize;
 pResultPtr += ResultSize;
}
```

Everything looks great! All you need to do now is unlock the vertex buffers and render the resulting mesh! I'll skip the code to unlock the buffers and get right to the good part−rendering the meshes.

Drawing Morphed Meshes

If you're building the morphing meshes by directly manipulating the resulting mesh's vertex buffer, as shown in the previous section, then rendering the morphing mesh is the same as for any other `ID3DXMesh` object you've been using. For example, you can loop through each material in the mesh, set the material and texture, and then draw the currently iterated subset. No need to show any code here−it's just simple mesh rendering.

On the other hand, if you want to move past the basics and start playing with real power, you can create your own vertex shader to render the morphing meshes for you. Take my word for it−this is something you'll definitely want to do. Using a vertex shader means you have one less mesh to deal with because the resulting mesh object is no longer needed; the speed increase is well worth a little extra effort.

Before you can move on to using a vertex shader, however, you need to figure out how to render the mesh's subsets yourself.

Dissecting the Subsets

To draw the morphing mesh, you need to set the source mesh's vertex stream as well as the target mesh's stream. Also, you need to set only the source mesh's indices. At that point, it's only a matter of scanning through every subset and rendering the polygons related to each subset.

Wait a second! How do you render the subsets yourself? By duplicating what the `ID3DXMesh::DrawSubset` function does, that's how! The `DrawSubset` function works in one of two ways. The first method, which you use if your mesh has not been optimized to use an attribute table, is to scan the entire list of attributes and render those batches of polygons belonging to the same subset. This method can be a little slow because it renders multimaterial meshes in small batches of polygons.

The second method, which is used after you optimize the mesh to use an attribute table, works by scanning the built attribute table to determine which grouped faces are drawn all in one shot. That is, all faces that belong to the same subset are grouped together beforehand and rendered in one call to `DrawPrimitive or DrawIndexedPrimitive.` That seems like the way to go!

To use the second method of rendering, you need to first optimize your source mesh. You can (and should) do this when you load the mesh. It's a safe habit to optimize all meshes you load using the `ID3DXMesh::OptimizeInPlace` function, as shown in the following bit of code:

```
// pMesh = just−loaded mesh
pMesh−>
OptimizeInPlace(D3DXMESHOPT_ATTRSORT,             \
                NULL, NULL, NULL, NULL);
```

Once the mesh is optimized, you can query the `ID3DXMesh` object for the attribute table it is using. The attribute table is of the data type `D3DXATTRIBUTERANGE,` which is defined as follows:

```
typedef struct_D3DXATTRIBUTERANGE {
   DWORD AttribId;
   DWORD FaceStart;
   DWORD FaceCount;
   DWORD VertexStart;
   DWORD VertexCount;
 } D3DXATTRIBUTERANGE;
```

The first variable, `AttribId` is the subset number that the structure represents. For each material in your mesh, you have one `D3DXATTRIBUTERANGE` structure with the `AttribId` set to match the subset number.

Next come `FaceStart` and `FaceCount.` You use these two variables to determine which polygon faces belong to the subset. Here's where the optimization comes in handy−all faces belonging to the same subset are grouped together in the index buffer. `FaceStart` represents the first face in the index buffer belonging to the subset, whereas `FaceCount` represents the number of polygon faces to render using that subset.

Last, you see `VertexStart` and `VertexCount,` which, much like `FaceStart` and `FaceCount,` determine which vertices are used during the call to render the polygons. `VertexStart` represents the first vertex in the vertex buffer to use for a subset, and `VertexCount` represents the number of vertices you can render in one call. When you optimize a mesh based on vertices, you'll notice that all vertices are packed in the buffer to reduce the number of vertices used in a call to render a subset.

For each subset in your mesh you must have a matching `D3DXATTRIBUTERANGE` structure. Therefore, a mesh using three materials will have three attribute structures. After you've optimized a mesh (using `ID3DXMesh::OptimizeInPlace`), you can get the attribute table by first querying the mesh object for the number of attribute structures using the `ID3DXMesh::GetAttributeTable` function, as shown here:

```
// Get the number of attributes in the table
DWORD NumAttributes;
pMesh−>GetAttributeTable(NULL, &NumAttributes);
```

At this point, you only need to allocate a number of `D3DXATTRIBUTERANGE` objects and call the `GetAttributeTable` function again, this time supplying a pointer to your array of attribute objects.

```
// Allocate memory for the attribute table and
// query for the table data
D3DXATTRIBUTERANGE *pAttributes;
pAttributes = new D3DXATTRIBUTERANGE[NumAttributes];
pMesh−>GetAttributeTable(pAttributes, NumAttributes);
```

Cool! After you've got the attribute data, you can pretty much render the subsets by scanning through each attribute table object and using the specified data in each in a call to `DrawIndexedPrimitive.` In fact, do that now by first grabbing the mesh's vertex buffer and index buffer pointers.

```
// Get the vertex buffer interface
IDirect3DVertexBuffer9 *pVB;
pMesh−>GetVertexBuffer(&pVB);

// Get the index buffer interface
IDirect3DIndexBuffer9 *pIB;
pMesh−>GetIndexBuffer(&pIB);
```

Now that you have both buffer pointers, go ahead and set up your streams, vertex shader, and vertex element declaration, and loop through each subset, setting the texture and then rendering the polygons.

```
// Set the vertex shader and declaration
pDevice->SetFVF(NULL); // Clear FVF usage
pDevice->SetVertexShader(pShader);
pDevice->SetVertexDeclaration(pDecl);

// Set the streams
pDevice->SetStreamSource(0, pVB,                            \
                         0, pMesh->GetNumBytesPerVertex());

pDevice->SetIndices(pIB);

// Go through each subset
for(DWORD i=0;i<NumAttributes;i++) {

   // Get the material id#
   DWORD MatID = pAttributes[i];

   // Set the texture of the subset
   pDevice->SetTexture(0, pTexture[AttribID]);

   // Render the polygons using the table
   pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0,         \
                                 pAttributes[i].VertexStart,    \
                                 pAttributes[i].VertexCount,    \
                                 pAttributes[i].FaceStart * 3,  \
                                 pAttributes[i].FaceCount);
  }
```

After you've rendered the subsets you can free the vertex buffer and index buffer interfaces you obtained.

```
pVB->Release(); pVB = NULL;
pIB->Release(); pIB = NULL;
```

When you're done with the attribute table, make sure to free that memory as well.

```
delete [] pAttributes; pAttributes = NULL;
```

All right, now you're getting somewhere! Now that you know how to render the subsets yourself, it's time to move on to using a vertex shader.

Creating a Morphing Vertex Shader

Something as simple as morphing meshes just demands the use of a vertex shader. In fact, once you've seen how easy it is to use a morphing vertex shader, you'll never go back to manipulating the vertex buffers yourself again!

Remember that at its essence, a morphing mesh is composed of interpolated position and normal coordinates that are calculated from a source mesh and a target mesh. Since those position and normal coordinates are part of the vertex stream, you can create a vertex shader that takes two vertex streams at a time and the same scalar values as before, and calculates the vertex values using a few simple commands.

That's right−no more locking and rebuilding a morphing mesh each frame. It's all happening in line with the shader! All you need to do is set the vertex stream sources. Using the rendering techniques shown in the previous section, you can render the groups of polygons belonging to the meshes.

Let's kick things up a notch and get started with your vertex shader. First, start with the vertex element declaration that will map the vertex data to your vertex registers in the shader.

Note Included with the Chapter 1 helper functions you'll find the `DrawMesh` function, which uses a vertex shader and a vertex element declaration interface to render a mesh object. By setting the target mesh to stream #1, you can call on `DrawMesh` to render the morphing mesh, much like this chapter's morphing mesh vertex shader demo. Check out the end of this chapter for details on locating the demo. As for the rest of this chapter, I'll show you how to render the mesh with raw code−no wrapper functions for you!

```
D3DVERTEXELEMENT9 MorphMeshDecl[] =
{
      // 1st stream is for source mesh
      { 0,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
      { 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL,   0 },
      { 0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0 },
      // 2nd stream is for target mesh
      { 1,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 1 },
      { 1, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL,   1 },
      { 1, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 1 },
      D3DDECL_END()
};
```

As you can see from the vertex element declaration, two streams are being used. Each stream is using a set of position, normal, and texture coordinates. The first stream uses a usage index of 0, whereas the second stream uses a usage index of 1.

When you are setting up the vertex streams, you set the source mesh's vertex buffer as stream #0 and the target mesh's vertex buffer as stream #1. You only use one index buffer−the source's, which is set using `IDirect3DDevice9::SetIndices.` Make sure both the source and target meshes use the same indices and vertex orders, or the morphing animation may be distorted.

I'll get back to the streams in a bit; for now I want to move on to the shader code. At the start of the vertex shader, you'll find the typical version number and mapping declarations.

```
vs.1.0

; declare mapping
dcl_position  v0
dcl_normal    v1
dcl_texcoord  v2
dcl_position1 v3
dcl_normal1   v4
```

You can see that only five vertex registers are being used, so where's the sixth? Because the texture coordinates are the same for both meshes, the second set of texture coordinates can be skipped entirely. All the better−that's less to deal with. The registers that are used map directly to the vertex element declaration. There are the source mesh's position coordinates `(v0)`, normal `(v1)`, and texture coordinates `(v2)`, followed by the target mesh's position coordinates `(v3)` and normal `(v4)`.

Moving on in the vertex shader, you begin calculating the vertex position coordinates by first multiplying `v0` by the inversed scalar value you provide in the vertex−shader constant register `c4.x.` Remember, this scalar value ranges from 0 to 1, with 0 indicating that the vertex coordinates match the source mesh's coordinates and 1 meaning that the vertex coordinates match the target mesh's coordinates.

Next you have to multiply the vertex coordinates stored in `v3` by the scalar value stored in the vertex−shader constant register `c4.y.` The vertex−shader code that performs these two calculations follows.

```
; apply scalar and inversed scalar values to coordinates
mul r0, v0, c4.x
mad r0, v3, c4.y, r0
```

Notice that you first multiply `v0` by the inversed scalar and then multiply `v3` by the scalar. Both values are summed and stored in a temporary register that you need to project using a combined and transposed world * view * projection transformation. I'll get to that in a second; for now, you need to process the normal values in the same way you did the vertex coordinates.

```
; apply scalar and inversed scalar values to normals
mul r1, v1, c4.x
mad r1, v4, c4.y, r1
```

All that's left now is to transform the coordinates by the transposed world * view * projection transformation matrix (stored in the vertex−shader constants `c0` though `c3`), dotproduct the normal by the inversed light direction (the same light direction you use in a `D3DLIGHT9` structure) stored in the vertex−shader constant `c5`, and store the texture coordinates (from the `v2` register).

```
; Project position
m4x4 oPos, r0, c0

; Dot normal with inversed light direction
; to get diffuse color
dp3 oD0, r1, −c5

; Store texture coordinates
mov oT0.xy, v2
```

See, I told you this vertex shader would be a simple one. Assuming you already have the code to load your vertex shader, skip to the good stuff and see how to render using the vertex shader.

The first step to rendering with your vertex shader is to set the constant registers. Earlier in this section, you learned that those constants are the world * view * projection transformation, light direction, and morph scalar values. Suppose you've stored the world, view, and projection transformations in the three following matrix objects:

```
// matWorld = world transformation matrix (D3DXMATRIX)
// matView = view transformation matrix (D3DXMATRIX)
// matProj = projection transformation matrix (D3DXMATRIX)
```

In case you didn't keep track of the three transformations, you can obtain them using the `IDirect3DDevice9::GetTransform` function, as shown here.

```
pDevice−>GetTransform(D3DTS_WORLD, &matWorld);
pDevice−>GetTransform(D3DTS_VIEW, &matView);
pDevice−>GetTransform(D3DTS_PROJECTION, &matProj);
```

After you've obtained the three transformations, you can combine them and set the transposed matrix in the vertex−shader constants `c0` through `c3` as follows.

```
D3DXMATRIX matWVP = matWorld * matView * matProj;
D3DXMatrixTranspose(&matWVP, &matWVP);
pDDevice−>SetVertexShaderConstantF(0, (float*)&matWVP, 4);
```

Next, set the morphing inversed−scalar value in `c4.x` and the scalar value in `c4.y.`

```
// Set the scalar and inverse scalar values to use
D3DXVECTOR4 vecScalar = D3DXVECTOR4(1.0f − Scalar,     \
                                    Scalar,            \
                                    0.0, 0.0);
pDevice−>SetVertexShaderConstantF(4, (float)&Scalar, 1);
```

Last, set the normalized direction that your light is pointing in the vertex−shader constant register `c5`. This direction value is the same directional vector stored in the `D3DLIGHT` structure that you should be used to by now, although this vector is cast as a `D3DXVECTOR4.`

```
// Set the light direction in the c5 constant register
D3DXVECTOR4 vecLight(0.0f, −1.0f, 0.0f, 0.0f);
pDevice−>SetVertexShaderConstantF(5, (float*)&vecLight, 1);
```

Note that the light direction is in object space, meaning that as your object rotates, so does your light. This is to ensure that the object is lit exactly as you want, regardless of the viewing direction. If you want the light in view space instead of object space, you need to transform the light direction by the inversed view transformation.

Whew! A little work goes a long way, and you're now ready to render the mesh using the vertex shader. At this point, it's only a matter of setting the vertex streams, texture, shader, and vertex declaration and calling your draw primitive function. Of course, you need to use the methods from the previous section to render the individual subsets of the source mesh. Instead of repeating the code you've already seen, I'll leave it up to you to check out the code from the vertex−shader demo for this chapter. Happy morphing!

Check Out the Demos

As you can see, morphing is an awesome animation technique that you can implement in your current projects with a minimum of effort. For this chapter, I created two demo programs (Morphing and MorphingVS) that demonstrate morphing, from using the vertex buffer manipulation technique to using a vertex shader.

These two programs operate in basically the same way. When you run either demo, you are treated to an animated display (shown in Figure 8.4) of a dolphin jumping through the animated waves of a cool ocean. Both programs continue until you exit the application.

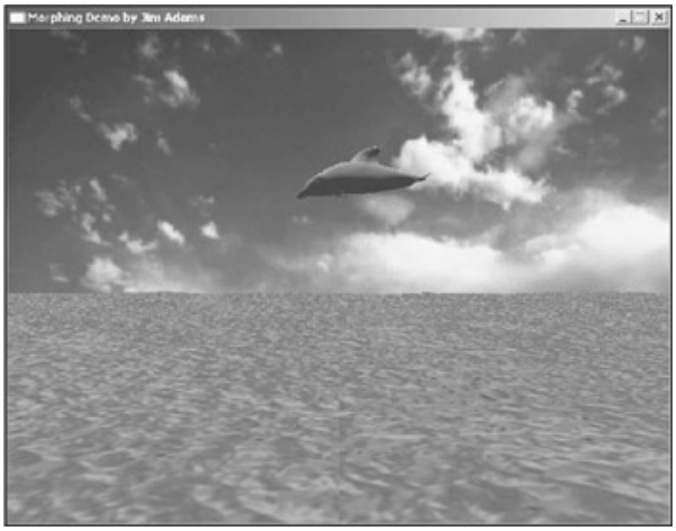


Figure 8.4: The animated dolphin jumps over morphing sea waves! Both objects (the sea and the dolphin) use morphing animation techniques.

Programs on the CD

Two projects for this chapter help demonstrate the use of morphing meshes. These two projects are located in the Chapter 8 directory of the book's CD−ROM.

- **Morphing.** This project demonstrates building morphing meshes by directly manipulating a mesh's vertex buffer. It is located at \BookCode\Chap08\Morphing.
- **MorphingVS.** With this project, you can check out how to implement vertex shaders to save memory and speed up rendering. It is located at \BookCode\Chap08\MorphingVS.

Chapter 9: Using Key−Framed Morphing Animation

Pre−calculated animations are the life−blood of today's game engines. With little work, an animator can design a complete animation sequence inside a popular 3D modeler and export the animation data to a format readily usable by the game engine. Without any extra effort, the programmer can change or modify those animations without rewriting the game code.

I know you've seen quite a few of these games, and I know that you want to have the same ability to play back pre−calculated animations in your own games. This chapter is just what you need!

Using Morphing Animation Sets

If you want to include morphing animation in your projects, you need to develop the means to include morphing animation sets. Unlike skeletal animation sets, morphing animation sets are extremely easy to use. Take a look at Figure 9.1, where you can see a common morphing animation sequence and how it is ordered.

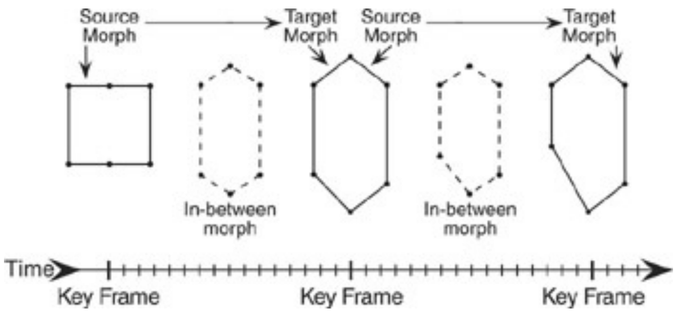


Figure 9.1: Morphing animation uses a series of source and target morphing meshes spaced over time (using animation keys) to create a continuous morphing animation.
Officially, there is no support for morphing animation set templates in DirectX, but don't let that stop you. You just need to create your own custom templates, and everything will be just fine.

Creating .X Morphing Animation Templates

Morphing animation data is foreign to the .X file format−it doesn't exist officially. For that reason, it's time to pump up those brain cells and devise your own morphing animation data templates. Now push out those evil scientist visions of you hysterically laughing at the thought of creating some wild animation data templates−you only need to follow a few simple steps to construct a morphing animation data template.

If you recall from Chapter 1, morphing animation requires two meshes−the source mesh and the target mesh. A single time value is used to morph from the source to the target mesh. By specifying a range of time values (key frames), you can use interpolation to morph into the final mesh that you'll render to the display. This means that you need to define a morphing animation key−frame template that defines a time value and a mesh to use as both the source and the target (for subsequent and previous keys) for morphing. For each key, a single mesh is defined. If a key precedes another key, then the current mesh represents the target mesh during the morph operation. A key following another will set the current mesh as the source in the morph operation.

The following two templates should suit your purposes for .X morphing animation data:

```
template MorphAnimationKey
{
   <2746B58A−B375−4cc3−8D23−7D094D3C7C67>
   DWORD Time;       // Key's time
   STRING MeshName;  // Mesh to use (name reference)
}

template MorphAnimationSet
{
   <0892DE81−915A−4f34−B503−F7C397CB9E06>
   DWORD NumKeys; // # keys in animation
   array MorphAnimationKey Keys[NumKeys];
}
```

Of course, every template needs a matching GUID declaration in your source, so add the following code to the top of your main source file (or wherever it's appropriate for you):

```
// {2746B58A−B375−4cc3−8D23−7D094D3C7C67}
DEFINE_GUID(MorphAnimationKey,
            0x2746b58a, 0xb375, 0x4cc3,
            0x8d, 0x23, 0x7d, 0x9,
            0x4d, 0x3c, 0x7c, 0x67);
{0892DE81−915A−4f34−B503−F7C397CB9E06}
DEFINE_GUID(MorphAnimationSet,
            0x892de81,  0x915a, 0x4f34,
            0xb5, 0x3,  0xf7, 0xc3,
            0x97, 0xcb, 0x9e, 0x6);
```

The comments in each of the two templates pretty much speak for themselves, but take a quick look back to make sure you're clear on a few things. In `MorphAnimationKey`, there are only two variables−`Time`, which is the timing value of the animation (the time the key is placed in the timeline, commonly specified in milliseconds), and `MeshName`, a `STRING` value that holds the name of the mesh used in the morphing operation.

As for the `MorphAnimationSet` template, there are only two variables. The first variable, `NumKeys`, is the number of `MorphAnimationKeys` objects that are contained within the animation. The second variable is the `Keys` array, which holds every `MorphAnimationKey` data object used in the animation.

Note You'll notice that the `MorphAnimationKey` template defines the mesh name as a `STRING` data type, not as a template reference. That means you have to match the mesh name to the animation key after the morphing animation keys have been loaded.

Now that you've been introduced to the templates, take a look at how you can use them to contain your morphing animation data. For a quick example of using your newly created template, create a simple animation. First, you'll need some meshes to work with:

```
Mesh MyMesh1 {
   // Mesh data goes here
}

Mesh MyMesh 2 {
   // Mesh data goes here
}

Mesh MyMesh 3 {
   // Mesh data goes here
}
```

To keep things simple, I left out the specific mesh data and merely showed you how to instance the `Mesh` data objects for later use. These three `Mesh` objects are named `MyMesh1`, `MyMesh2`, and `MyMesh3`. For the animation data, I want to define two animations, called `MyAnimation1` and `MyAnimation2`.

```
MorphAnimationSet MyAnimation1
{
   2;
      0; "MyMesh1";,
      500; "MyMesh2";,
}

MorphAnimationSet MyAnimation2
{
   4;
      0; "MyMesh1";,
    500; "MyMesh2";,
   1000; "MyMesh3";,
   1500; "MyMesh2";;
}
```

The first animation, `MyAnimation1`, contains two animation keys. The first key is at time 0 and uses `MyMesh1` as a source mesh. The second key is at time 500 and uses `MyMesh2` as the target mesh. The animation is therefore 500 time units in length.

For the second animation, `MyAnimation2`, there are four animation keys. These keys, spaced 500 time units apart, use all three meshes in the series. If you wanted to update the animation at 700 time units, you would use `MyMesh2` for the source mesh and `MyMesh3` as the target mesh in a morphing animation. The scalar value for the morphing operation would be calculated using the key's time values.

Well, that's pretty easy to understand. It's the part where you actually load this animation data that will be a little tough.

Loading Morphing Animation Data

After you've defined your morphing animation templates, you can load your animation data and start putting it to good use. The first step is to define three classes that match the templates' data structures, containing data for animation keys, sets, and a collection of animation sets.

```
class cMorphAnimationKey
{
   public:
      DWORD m_Time;       // Time of key
      char *m_MeshName;   // Name of mesh to use
      D3DXMESHCONTAINER_EX *m_MeshPtr; // Pointer to mesh data

   public:
      cMorphAnimationKey()
      {
         m_MeshName = NULL;
         m_MeshPtr = NULL;
      }

      ~cMorphAnimationKey()
      {
         delete [] m_MeshName;
         m_MeshName = NULL;
      m_MeshPtr = NULL;
    }
};

class cMorphAnimationSet
{
  public:
    char               *m_Name;     // Name of animation
    DWORD               m_Length;   // Length of animation
    cMorphAnimationSet *m_Next;     // Next animation in linked list

    DWORD               m_NumKeys;  // # keys in animation
    cMorphAnimationKey *m_Keys;     // Array of keys

  public:
    cMorphAnimationSet()
    {
      m_Name    = NULL;
      m_Length  = 0;
      m_Next    = NULL;
      m_NumKeys = 0;
      m_Keys    = NULL;
    }

    ~cMorphAnimationSet()
    {
      delete [] m_Name; m_Name    = NULL;
      m_Length  = 0;
      m_NumKeys = 0;
      delete [] m_Keys; m_Keys    = NULL;
      delete m_Next;    m_Next    = NULL;
    }
};

class cMorphAnimationCollection : public cXParser
{
  protected:
    DWORD                     m_NumAnimationSets;   // # animation sets
    cMorphAnimationSet *m_AnimationSets;            // Animation sets

  protected:
    // Parse an .X file for mass and spring data
    BOOL ParseObject(IDirectXFileData *pDataObj,
                     IDirectXFileData *pParentDataObj,
                     DWORD Depth,
                     void **Data, BOOL Reference);

  public:
    cMorphAnimationCollection()
    {
      m_NumAnimationSets = 0;
      m_AnimationSets    = NULL;
    }

    ~cMorphAnimationCollection() { Free(); }

    BOOL Load(char *Filename);
    void Free();

    void Map(D3DXMESHCONTAINER_EX *RootMesh);
    void Update(char *AnimationSetName,               \
                    DWORD Time, BOOL Loop,                \
                    D3DXMESHCONTAINER_EX **ppSource,      \
                    D3DXMESHCONTAINER_EX **ppTarget,      \
                    float *Scalar);
};
```

The first class shown, cMorphAnimationKey, stores the time of the key, the mesh's name, and a pointer to a mesh object (which you need to set after you load all meshes from the .X file).

The cMorphAnimationSet class contains an array of cMorphAnimationKey objects that constitute the entire animation. Each cMorphAnimationSet class contains a name buffer that is filled with the matching animation set's data object from the .X file. This means you can have any number of animations (each contained within its own cMorphAnimationSet object), each identified by a unique name.

This list of animation set objects is contained in a linked list, which you access via the cMorphAnimationSet::m_Next pointer. (One animation set points to the next set in the list.) The entire list of animation set objects is stored in the cMorphAnimationCollection object, which stores only a pointer to the root animation set object. To access any animation set object, you need to scan through the animation set objects and look for the specific animation.

You should be familiar with the member variables from the cMorphAnimationSet object (except for the m_Length variable, which stores the length of the animation as determined by the time of the last key in the array of cMorphAnimationKey objects). You'll use m_Length later to determine the length of the animation without having to continuously scan through the array of keys.

Whereas the cMorphAnimationSet class contains a single animation, it's the responsibility of the cMorphAnimationCollection class to contain a series of cMorphAnimationSet objects (through the use of the aforementioned linked list pointers).

To load the morphing mesh animations, you derive the cMorphAnimationSet class from the cXParser class you developed in Chapter 3. Deriving from the cXParser class gives you access to the ParseObject function, which is where you'll slip in the code to retrieve the animation data from any MorphAnimationSet objects.

However, you don't have to call ParseObject directly because there is a Load function that does it for you. You only need to call Load, specifying the name of the .X file from which you want to load the morphing animation data sets, and let the class functions take care of the rest. When you are finished with the animation data, a call to Free releases all resources used to contain the animation data.

The constructor, destructor, Load, and Free functions are trivial in this instance; I'll leave it up to you to look at the code for those on the CD−ROM. However, you do want to take a close look at the ParseObject function here. Basically, you want to code the ParseObject function to look for any MorphAnimationSet object instances and create a cMorphAnimationSet object to contain the animation data. You can use the following code to do so:

```
BOOL cMorphAnimationCollection::ParseObject(              \
                    IDirectXFileData *pDataObj,           \
                    IDirectXFileData *pParentDataObj,     \
                    DWORD Depth,                          \
                    void **Data, BOOL Reference)
{
   const GUID *Type = GetObjectGUID(pDataObj);
   // Read in animation set data
   if(*Type == MorphAnimationSet) {

      // Create and link in a cMorphAnimationSet object
      cMorphAnimationSet *AnimSet = new cMorphAnimationSet();
      AnimSet->m_Next = m_AnimationSets;
      m_AnimationSets = AnimSet;

      // Increase # of animation sets
      m_NumAnimationSets++;

      // Set the animation set's name
      AnimSet->m_Name = GetObjectName(pDataObj);

      // Get data pointer
      DWORD *Ptr = (DWORD*)GetObjectData(pDataObj, NULL);

      // Get # of keys and allocate array of keyframe objects
      AnimSet->m_NumKeys = *Ptr++;
      AnimSet->m_Keys = new cMorphAnimationKey[AnimSet->m_NumKeys];

      // Get key data - time and mesh names
      for(DWORD i=0;i<AnimSet->m_NumKeys;i++) {
         AnimSet->m_Keys[i].m_Time     = *Ptr++;
         AnimSet->m_Keys[i].m_MeshName = strdup((char*)*Ptr++);
      }

      // Store length of animation
      AnimSet->m_Length = AnimSet->m_Keys[AnimSet->m_NumKeys-1].m_Time;
   }

   return ParseChildObjects(pDataObj, Depth, Data, Reference);
}
```

The only critical code in the `cMorphAnimationCollection::ParseObject` function is the bit that loads morphing animation set objects. After you create an instance of the `cMorphAnimationSet` object and link it into a linked list of objects, you need to read in the number of keys used in the animation. Following the number of keys, the code enters a loop that reads every key's time value and mesh name. After the loop is finished, the total length of the animation is grabbed (using the last key's time value).

After you load all the animation sets in the .X file, it is time to link the appropriate mesh objects to the animation key objects. You need to do this so that you have fast access to each mesh that an animation set object needs to render.

To match the meshes to the animation sets, create a function that scans through each animation set. For each key in the set, scan through a list of meshes you provide for ones that are named the same as the names stored in the keys. This is exactly the purpose of the `cMorphAnimationCollection::Map` function.

```
void cMorphAnimationCollection::Map(                         \
                         D3DXMESHCONTAINER_EX *RootMesh)
{
   // Error checking
   if(!RootMesh)
      return;

   // Go through each animation set
   cMorphAnimationSet *AnimSet = m_AnimationSets;
   while(AnimSet != NULL) {
      // Go through each key in animation set and match
      // meshes to key's mesh pointers
      if(AnimSet->m_NumKeys) {
         for(DWORD i=0;i<AnimSet->m_NumKeys;i++)
               AnimSet->m_Keys[i].m_MeshPtr = \
                           RootMesh->Find(AnimSet->m_Keys[i].m_MeshName);
         }

      // Go to next animation set object
      AnimSet = AnimSet->m_Next;
   }
}
```

In the `Map` function, I use the `D3DXMESHCONTAINER_EX::Find` function to iterate all meshes contained with the object to look for a specifically named mesh. If found, the pointer to that mesh is stored in the key object.

When it is time to render the morphing animation set, you perform a quick scan of the keys, find the mesh pointers, and perform a morphing mesh render. You can follow the same steps to render your own morphing animation. Just follow along with me in the next section, and you'll be rendering morphing animations in no time!

Rendering the Morphing Mesh

Rendering with the morphing mesh animation set is easy, now that you've loaded all your data. By taking a time value in your animation timeline, you can scan through the array of animation keys and look for the two keys between which the time falls. The first key contains a pointer to the source mesh to use in the morphing operation, and the second key contains a pointer to the target mesh.

Since you can have any number of animations loaded into your animation collection object, you need to create a function that scans for a specifically named animation, and then scan through that animation, grabbing the appropriate key values you need to render with. This is what the `cMorphAnimationCollection::Update` function does.

```
void cMorphAnimationCollection::Update(                     \
                            char *AnimationSetName,          \
                            DWORD Time, BOOL Loop,           \
                            D3DXMESHCONTAINER_EX **ppSource, \
                            D3DXMESHCONTAINER_EX **ppTarget, \
                            float *Scalar)
```

Although it appears quite formidable, the `Update` function is really easy to use. You need to supply the name of the animation set you want to use as the `AnimationSetName` parameter, the time of the animation as `Time`, and whether of not to use looping (set Loop to `TRUE` or `FALSE`).

Also, you need to provide two pointers (`ppSource` and `ppTarget`) that are filled with the source and target mesh objects you'll use to perform the morphing render function, along with a pointer to a variable that will contain the scalar value to use for morphing.

The `Update` function starts by getting a pointer to the animation set linked list and clearing the pointers you passed to the function.

```
{
   cMorphAnimationSet *AnimSet = m_AnimationSets;

   // Clear targets
   *ppSource = NULL;
   *ppTarget = NULL;
   *Scalar = 0.0f;
```

The pointers are cleared in case an error occurs. If it does, you can check the pointers to see whether they are set to `NULL`. `Non−NULL` values mean the call to `Update` worked.

Moving on with the function, you scan the list of animation sets to look for a match to the name you passed in the `AnimationSetName` parameter. You can force the `Update` function to use the first animation set in the linked list by setting `AnimationSetName` to NULL.

The name of the animation must match what is stored in the .X file. The name from the .X file is the animation set's data object instance name. For example, the following animation set data object has an instance name of `Walk:`

```
MorphAnimationSet Walk
{
   2;
     0; "Figure1";,
   500; "Figure2";;
}
```

To get the pointers to the `Figure1` and `Figure2` meshes, you would specify `Walk` as the `AnimationSetName`. I think you get the picture, so here's the code that will scan the list of animation sets (returning from the function if no set was found or if the animation set found has no animation keys set).

```
   // Look for matching animation set name if used
   if(AnimationSetName) {

      // Find matching animation set name
      while(AnimSet != NULL) {

         // Break when match found
         if(!stricmp(AnimSet−>m_Name, AnimationSetName))
            break;

         // Go to next animation set object
         AnimSet = AnimSet−>m_Next;
      }
   }

   // Return no set found
   if(AnimSet == NULL)
      return;

   // Return if no keys in set
   if(!AnimSet−>m_NumKeys)
      return;
```

At this point, you have a pointer to the `cMorphAnimationSet` object that contains the key data to use for your morphing animation. With this pointer (`AnimSet`), you can bounds−check the time of the animation from which you want to obtain the key data with the actual length of the animation (as stored in the animation set's `m_Length` data member).

```
   // Bounds time to animation length
   if(Time > AnimSet->m_Length)
     Time = (Loop==TRUE)?Time%(AnimSet->m_Length+1):AnimSet->m_Length;
```

From here, you can scan through each key object in the animation set and determine which keys to use to obtain the pointers to the source and target meshes for rendering. You saw how to scan the keys in Chapter 5, so I'll skip the explanation and get to the code.

```
   // Go through animation set and look for keys to use
   DWORD Key1 = AnimSet->m_NumKeys-1;
   DWORD Key2 = AnimSet->m_NumKeys-1;
   for(DWORD i=0;i<AnimSet->m_NumKeys-1;i++) {
      if(Time >= AnimSet->m_Keys[i].m_Time &&              \
         Time < AnimSet->m_Keys[i+1].m_Time) {

      // Found the key, set pointers and break
      Key1 = i;
      Key2 = i+1;
      break;
   }
}
```

Now that you've found the keys you want to use in the morphing animation (stored as `Key1` and `Key2`), you can calculate a morphing scalar value based on the times of the two keys.

```
   // Calculate a time scalar value to use
   DWORD Key1Time = AnimSet->m_Keys[Key1].m_Time;
   DWORD Key2Time = AnimSet->m_Keys[Key2].m_Time;
   float KeyTime = (float)(Time - Key1Time);
   float MorphScale = 1.0f/(float)(Key2Time-Key1Time)*KeyTime;
```

Finally, you can set the source, target, and scalar pointers (which you passed to the `Update` function) with the appropriate values.

```
   // Set pointers
   *ppSource = AnimSet->m_Keys[Key1].m_MeshPtr;
   *ppTarget = AnimSet->m_Keys[Key2].m_MeshPtr;
   *Scalar   = MorphScale;
}
```

Now that all your classes and functions are defined, you can get to work! Take a look at an example that uses the classes you just created to load and use a morphing animation collection from a file called MorphAnim.x. First, you need to instance a `cAnimationCollection` object and load a series of animation sets.

```
cMorphAnimationCollection MorphAnim;
MorphAnim.Load("MorphAnim.x");
```

You also need to load some meshes. You can use the handy helper functions you developed in Chapter 1 to load a series of meshes (from the same MorphAnim.x file shown here).

```
D3DXMESHCONTAINER_EX *Meshes;
LoadMesh(&Meshes, NULL, pDevice, "MorphAnim.x");
```

After the meshes and animation sets are loaded, you need to map the animation sets to the meshes using the `cMorphAnimationCollection::Map` function.

```
MorphAnim.Map(MorphAnim);
```

From this point, you can use the `Update` function to get the source mesh, target mesh, and scalar values to create your sequenced morphing animation. Going way back to when you created these morphing animation set templates, assume you have two animation sets−one called `MyAnimation1` and another called `MyAnimation2.`

You're going to use `MyAnimation2` here, and you want to determine how to update the animation at a time of 700 units (specifying looping). The following code will determine which source mesh, target mesh, and scalar values you must use:

```
// pAnimCollection = preloaded cMorphAnimationCollection object
// Time = DWORD value w/time of animation to use,
//        which is 700 in this case

// Pointers to the source and target meshes used for rendering
D3DXMESHCONTAINER_EX *pSource, *pTarget;

// Scalar value to use for morphing
float Scalar;

// Call Update to object morphing mesh and scalar data
MorphAnim.Update("MyAnimation2", Time, TRUE, \
                 &pSource, &pTarget, &Scalar);
```

The source mesh and target mesh pointers now point to the meshes to use in your call to render your morphing mesh, along with `Scalar`, which contains the scalar value. Things are starting to wrap up now; all that's left is for you to draw the morphing mesh using the techniques you learned in Chapter 8. If you want, check out the demo for this chapter, which shows a morphing animation in action. Go ahead, you know you want to. The source code for the demo is just sitting there, waiting for you to use it in your project!

Obtaining Morphing Mesh Data from Alternative Sources

Well, designing your own morphing animation templates and classes is really a top−notch effort on your part, but what good is it if you can't obtain that morphing animation data from anywhere, such as from a popular 3D rendering package? Don't fret, there are a few things you can do to get that morphing animation set data you crave in your projects.

The CD−ROM includes a program called MeshConv, which you can use to convert .MD2 files into .X files. What are .MD2 files, you ask? Developed by id Software, an .MD2 file (used in games such as id's *Quake*) contains morphing mesh and animation set data. You can find hundreds of .MD2 files on the Internet (such as from http://www.planetquake.com/polycount) to use in your own programs. Using the MeshConv program, you can convert those files into .X files for use in your projects.

You probably remember this handy program from Chapter 5. In case you didn't read that chapter yet, let me give you a brief glimpse of the program here. After you execute the program, you are presented with the MeshConv dialog box, shown in Figure 9.2.

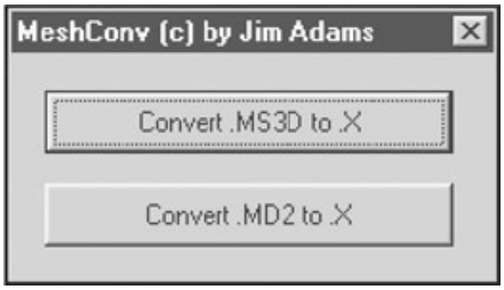


Figure 9.2: The MeshConv dialog box contains two buttons you can click to convert .MS3D and .MD2 files to .X files.

To convert an .MD2 file to an .X file, click on the Convert .MD2 to .X button in the MeshConv dialog box. The Open File dialog box will appear. This dialog box allows you to navigate your drives and locate the file you want to convert to .X format. Select an appropriate file to convert and click Open.

Next, the Save .X File dialog box will appear. You can use this dialog box to locate and select an .X file into which you want to save the mesh and animation data. Enter a file name and click Save. After a moment, you'll see a message box informing you that the conversion was successful.

Now you're ready to use your .X file with one of the classes you developed earlier in this chapter. You have a series of Mesh templates that contain every target morphing mesh used in the source file. A single `MorphAnimationSet` template will help you load animation data into your project using the classes and techniques you studied in this chapter.

For an example of how to work with the .X files you created using the MeshConv program, check out the MorphAnim demo included for this chapter.

Check Out the Demos

This chapter contains two projects−one that converts .MD2 files into .X files, and another that demonstrates how to use morphing animation sets in your own projects. Fire up the MorphAnim demo (shown in Figure 9.3) to check out how effective morphing animation sets are.

Figure 9.3: An animator's dream comes true via a morphing music box ballerina animation!

Programs on the CD

Check Out the Demos

In the Chapter 9 directory on this book's CD−ROM, you'll find the following projects to peruse and use for your own game projects.

- **MeshConv.** You can use this utility program to convert your .MS3D and .MD2 files to .X files. The source code is fully commented and shows the format of the two file types. It is located at \BookCode\Chap09\MeshConv.
- **MorphAnim.** You can use this demo program to see how to use morphing animation sets. It is located at \BookCode\Chap09\MorphAnim.

Chapter 10: Blending Morphing Animations

Punching, ducking, kicking, walking, and jumpingthat's a whole lot of animation to work with. Can you imagine taking the time to painstakingly define each and every one of those animations, only to have your boss decide he wants you to make characters jump and punch at the same time, or duck and kick, or perform any number of other combined animations? What are you to do?

Well, if you're using morphing animation, then all you can do is to work your programming magic and develop a blended morphing animation technique in which the meshes of your previously created animations can be blended to form new and unique animations on the fly and in real time. This chapter is here to show you how to do just that!

Blending Morphing Animations

Way back in Chapter 6, you saw how to create new and dynamic animations by combining, or rather *blending*, various mesh movements as defined in multiple animation sets. Whereas Chapter 6 concentrated on blending skeletal−based animation sets, this chapter will show you how to achieve the same blended animation effects with your morphing meshes. That's rightin this chapter you'll see how to combine the various movements of your pixel−based morphing meshes to create new and dynamic animations on the fly!

Blending morphing mesh animations is a little bit tougher to accomplish than blending skeletal−based mesh animations. Don't worry, though. It's not that much harderit just takes a different approach. With blended morphing animation, you want to be able to take the results of multiple morphed meshes and combine them into one mesh.

For example, suppose you have a mesh that represents a person's face, and you have two morphing animationsone that opens and closes the mesh's mouth and another that blinks the mesh's eyes. Figure 10.1 shows the cycle of each animation.

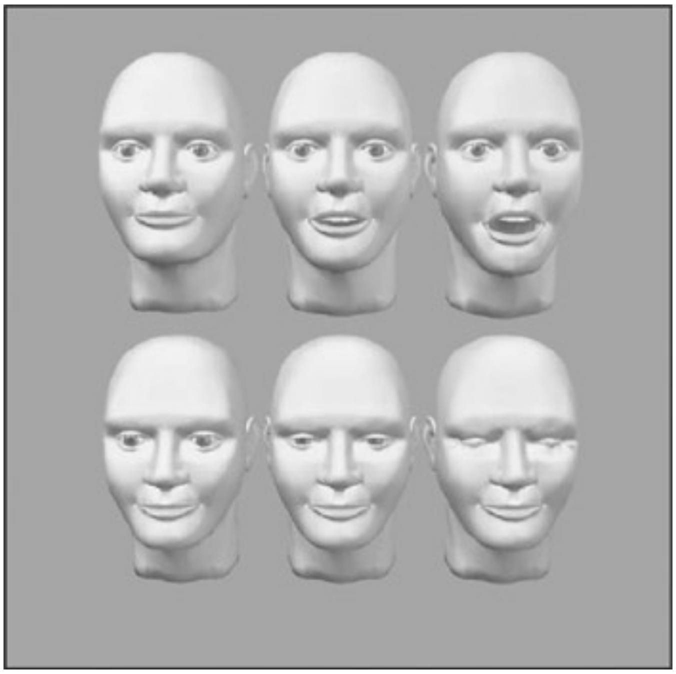


Figure 10.1: Using the same mesh, you move (morph) various vertices to create two unique animation sequences.

Now suppose you want to spice things up by blending the two animations (as shown in Figure 10.2) so your mesh has the ability to open its mouth and blink its eyes at the same time, and at various speeds for each animation. Sounds difficult, doesn't it? Well, it's not that tough once you know the secret.

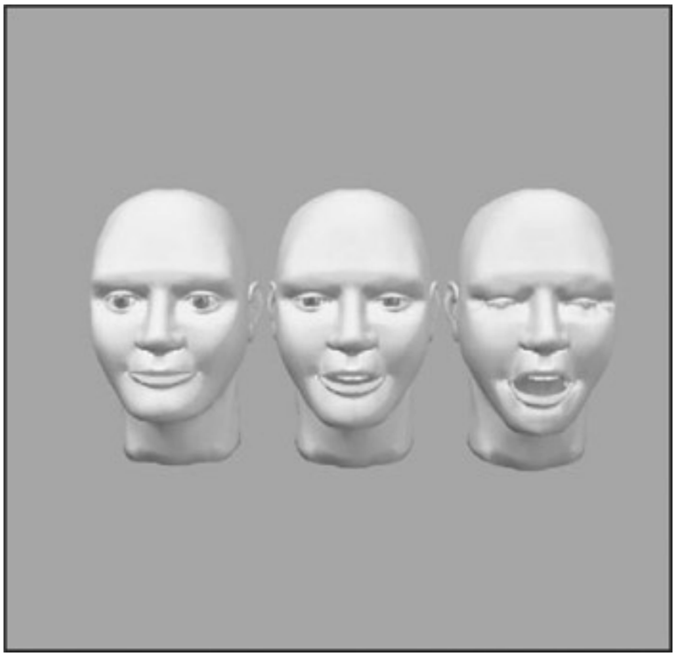


Figure 10.2: You can combine two animations to create a single blended mesh.

The trick to blending morphing animations is to isolate only those vertices that actually move during each animation sequence, and to combine each vertex's motions into a single mesh that represents the final mesh you want to render. Trying to determine which vertices move during an animation sequence sounds tricky, but believe me, with the help of an additional reference mesh (called a *base mesh*), you can tackle this problem with your hands tied behind your back!

Using a Base Mesh in Blended Morphing Animation

The real secret to using blended morphing animation is to introduce a reference mesh into the equation. The *base mesh* defines the initial coordinates of each vertex in the mesh before any morphing is applied. This base mesh is commonly the source mesh of the morph operation. In the example of a person's face, the base mesh is the person's face with its mouth closed and eyes open.

When you are ready to blend two morphed meshes (or any number of meshes, for that matter), you determine the difference in vertex coordinates between each morphed mesh and the base mesh. You blend these differences in coordinates into a final blended morphing mesh. Read on to see how to calculate those differences.

Calculating the Differences

Calculating the difference values is pretty basic. Each animation you blend requires a source mesh (the base mesh) and a target mesh. For the first animation, the target mesh is the one with the person's mouth fully open. The second animation's target mesh is the one with the person's eyes fully closed. And of course, you remember the base meshit's the one with the person's mouth fully closed and eyes fully open.

In this example, those meshes will be the ones shown in Figure 10.2. I'll assume you have gone through the trouble of loading each mesh in the following three mesh objects.

```
ID3DXMesh *pBaseMesh;    // Contains the base mesh
ID3DXMesh *pTargetMesh1; // Contains 1st morph target mesh
ID3DXMesh *pTargetMesh2; // Contains 2nd morph target mesh
```

Aside from the base mesh and the two target meshes, you need one more mesh to store the final blended mesh. This blended mesh is identical to the others with regard to the number of vertices and faces, so you can clone the blended mesh from the base mesh to make sure everything matches.

To clone the base mesh (after it has been loaded) into the blended mesh (an `ID3DXMesh` object that you'll call `pBlendedMesh`), you can use the following code.

```
ID3DXMesh *pBlendedMesh;

// Get the base mesh's FVF and 3-D device
DWORD FVF = pBaseMesh->GetFVF();
IDirect3DDevice9 *pMeshDevice;
pBaseMesh->GetDevice(&pMeshDevice);

// Clone the mesh
pBaseMesh->CloneMeshFVF(0, FVF, pMeshDevice, &pBlendedMesh);
```

So, you'll be working with four mesh objectsthe base mesh, two target meshes, and a blended mesh. To keep things easy, I'll assume each mesh uses the same vertex structure and FVF, using only a vertex position, normals, and pair of texture coordinates, as I have declared here. If need be, just clone all meshes to use the same FVF as I did previously.

```
typedef struct {
    D3DXVECTOR3 vecPos;    // Vertex coordinates
    D3DXVECTOR3 vecNormal; // Vertex normals
    float u, v;            // Texture coordinates
} sVertex;
#define VERTEXFVF (D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1)
```

When your meshes are all ready, you can lock the vertex buffers of each to access the vertex data.

```
// Lock all meshes and get pointers
sVertex *pBaseVertices, *pBlendedVertices;
sVertex *pTarget1Vertices, *pTarget2Vertices;

pBaseMesh->LockVertexBuffer(D3DLOCK_READONLY,                      \
                            (void**)&pBaseVertices);
pTargetMesh1->LockVertexBuffer(D3DLOCK_READONLY,                   \
                            (void**)&pTarget1Vertices);
pTargetMesh2->LockVertexBuffer(D3DLOCK_READONLY,                   \
                            (void**)&pTarget2Vertices);
pBlendedMesh->LockVertexBuffer(0, (void**)&pBlendedVertices);
```

Now that you have the pointers to each mesh's vertex buffer, you can begin iterating through each target mesh's vertices, subtracting the vertex (and normal) coordinates from the matching base mesh's vertices (with each different value stored in temporary registers that you will tally later).

```
// Iterate all vertices
for(DWORD i=0;i<pBaseMesh->GetNumVertices();i++) {

   // Get the difference in vertex coordinates
   D3DXVECTOR3 vecPosDiff1 = pTarget1Vertices->vecPos -       \
                          pBaseVertices->vecPos;
   D3DXVECTOR3 vecPosDiff1 = pTarget2Vertices->vecPos -       \
                          pBaseVertices->vecPos;
   // Get the difference in normals
   D3DXVECTOR3 vecNormalDiff1 = pTarget1Vertices->vecNormal - \
                             pBaseVertices->vecNormal;
   D3DXVECTOR3 vecNormalDiff2 = pTarget2Vertices->vecNormal - \
                             pBaseVertices->vecNormal;
```

You're halfway therejust hang in a little bit longer! You've calculated the differences, so now you need to blend them.

Blending the Differences

At this point, you have the difference values for the vertex and normal coordinates. The next step is to scale each of the differences based on the amount of blending you want for each mesh. For example, if you only want the first animation (the mouth opening) to use 50 percent of the difference (meaning the mouth would be halfway open), then multiple the values by 0.5.

Tip In addition to determining the percentage of the differences to blend, you can use the blending values as a factor to animate the blended mesh. Slowly increase the blending values from 0 to 1 over time to achieve smooth animation.

To make things easy, specify the amount of blending in two variablesone for the blending amount for the first mesh and the other for the blending amount for the second mesh.

```
float Blend1 = 1.0f; // Use 100% of the differences
float Blend2 = 1.0f;
```

To blend the differences, you only need to multiply them by the blending values you just defined.

```
// Apply blending values
vecPosDiff1 *= Blend1; vecNormalDiff1 *= Blend1;
vecPosDiff2 *= Blend2; vecNormalDiff2 *= Blend2;
```

When you've got the differences scaled by your blending factors, you can add them together to achieve the blended difference values.

```
// Get tallied blended difference values
D3DXVECTOR3 vecBlendedPos = vecPosDiff1 + vecPosDiff2;
D3DXVECTOR3 vecBlendedNormal = vecNormalDiff1+vecNormalDiff2;
```

The final step is to add the blended differences to the coordinates of the base mesh's vertex coordinates and normal, and store the results in the blended mesh's vertex buffer.

```
// Add the differences back to base mesh values and store
pBlendedVertices->vecPos = vecBlendedPos +                  \
                           pBaseVertices->vecPos;
// Normalize the normals before storing them!
D3DXVECTOR3 vecNormals = BlendedNormal+pBaseVertices->vecNormal;
D3DXVec3Normalize(&pBlendedVertices->vecNormal, &vecNormals);
```

All that's left now is to increase the vertex buffer pointers so the next vertex in each mesh is iterated, close the `fornext` code block to finish iterating the vertices, and unlock the vertex buffers.

```
// Go to next vertices
  pBaseVertices++; pBlendedVertices++;
  pTarget1Vertices++; pTarget2Vertices++;
} // Next loop iteration

// Unlock the vertex buffers
pBlendedMesh->UnlockVertexBuffer();
pTarget2Mesh->UnlockVertexBuffer();
pTarget1Mesh->UnlockVertexBuffer();
pBaseMesh->UnlockVertexBuffer();
```

And there you have it! One complete blended mesh, ready to render! You've got to admit that blending two meshes is pretty cooland easy, for that matter. What would you say to blending four or more meshes? You'd think I was crazy, but if you check out the source code for the blended morphing mesh demo included on the CD−ROM, you'll see that it's been done! That's right, the complete code for blending four morphing animations is sitting on that disc, waiting for you to use it in your next project.

What's that, you say? You'd prefer a faster way to blend the meshes? Well, the programming gods have answered your prayers, my friend, for you're about to see how you can use a vertex shader to do the blending dirty work for you!

Building a Blending Morph Vertex Shader

Vertex shaders are going to be your savior when it comes to working with blended morphing animations. Earlier in this chapter, you saw what was involved in blending mesheslocking each vertex buffer, calculating the differences, and finally blending the values togethereach and every frame, the same slow process.

Talk about a major slowdown, especially if those meshes' buffers are contained in video memory. Using a vertex shader is guaranteed to increase the speed at which your blended morph animations are processed because you can store all your mesh data in fast video memory and let the vertex shader deal with the processing.

Your blended vertex shader will work pretty much the same way you blended the vertex coordinates directly earlier in this chapter. For each mesh that you want to blend, you need access to the vertex buffers. With vertex shaders, you have to map each mesh's vertex buffer to a vertex stream. Instead of locking the vertex buffer to get at the vertex data, you use the vertex stream data (as mapped out by the vertex declaration) to obtain the coordinate and normal data.

The only problem is that vertex shaders can only use so many streams. In addition, each vertex shader can only access a limited number of vertex registers (such as coordinates, normal data, and texture coordinates). DirectX 8 and 9 limit the number of vertex registers to 16, so you have to make every little bit of data count.

If, at minimum, your vertex structure contains the coordinates of the vertex, normal, and texture (for a total of three registers per stream), you could only use five meshes, one of which is the base mesh. You could use up to four blending meshes inside a vertex shader. Not to worry, howeverfour blending meshes is probably more than you'll ever need!

The first step in creating your blending morph vertex shader is defining the vertex structure and declaration. Remember, you want to keep the amount of vertex data to a minimum (three registers at most), so your vertex structure might look something like this:

```
typedef struct {
  D3DXVECTOR3 vecPos;    // Vertex coordinates
  D3DXVECTOR3 vecNormal; // Vertex normals
  float u, v;            // Texture coordinates
} sBlendVertex;
```

The `sBlendVertex` structure's comments speak for themselves, so I'll skip ahead to the vertex declaration.

```
// Vertex shader declaration and interfaces
D3DVERTEXELEMENT9 g_MorphBlendMeshDecl[] =
{
  // 1st stream is for base mesh
   // specify position, normal, texture coordinate
   { 0,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,      \
            D3DDECLUSAGE_POSITION, 0 },
   { 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_NORMAL,   0 },
   { 0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_TEXCOORD, 0 },

   // 2nd stream is for mesh 1
   { 1,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_POSITION, 1 },
   { 1, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_NORMAL,   1 },
   { 1, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_TEXCOORD, 1 },

   // 3rd stream is for mesh 2
   { 2,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_POSITION, 2 },
   { 2, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_NORMAL,   2 },
   { 2, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_TEXCOORD, 2 },

   // 4th stream is for mesh 3
   { 3,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_POSITION, 3 },
   { 3, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_NORMAL,   3 },
   { 3, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_TEXCOORD, 3 },

   // 5th stream is for mesh 4
   { 4,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_POSITION, 4 },
   { 4, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_NORMAL,   4 },
   { 4, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_TEXCOORD, 4 },
   D3DDECL_END()
};
```

In `g_BlendMorphDecl`, you can see that you are defining five streams, each specifying a three−dimensional position and a normal, as well as a two−dimensional texture coordinate. The first stream uses an index usage value of 0; the second stream uses an index usage value of 1; and so on. This means the vertex shader will have access to five usage types for each of the vertex components (position, normal, texture coordinates). Table 10.1 shows the data stored in each vertex register.

Table 10.1: Blended Morph Animation Vertex Register Assignments

Vertex Register	Assignment
`v0`	First (base) mesh's 3D coordinates
`v1`	First (base) mesh's normal
`v2`	First (base) mesh's texture coordinates
`v3`	Second mesh's 3D coordinates
`v4`	Second mesh's normal
`v5`	Second mesh's texture coordinates
`v6`	Third mesh's 3D coordinates
`v7`	Third mesh's normal
`v8`	Third mesh's texture coordinates
`v9`	Fourth mesh's 3D coordinates
`v10`	Fourth mesh's normal
`v11`	Fourth mesh's texture coordinates
`v12`	Fifth mesh's 3D coordinates
`v13`	Fifth mesh's normal
`v14`	Fifth mesh's texture coordinates

Note You can find the blended morph vertex shader (MorphBlend.vsh) on the CD−ROM. Check out the end of this chapter for information on the blended morphing demo.

After you create the vertex structure and declaration, you can move on to creating the vertex shader itself. The shader will duplicate what you've already done previously in this chapter. For each vertex to be blended, you calculate the difference in vertex coordinates from the target morphing mesh to the base mesh.

That difference is scaled by a percentage you specify (via a vertex shader constant ranging from 0 to 1), and the results are tallied from each blended mesh into a resulting set of position coordinates, normals, and texture coordinates used to output a vertex. Simple, isn't it?

Take a look at the vertex code bit by bit to see just what's occurring.

```
; v0 = Base mesh's position.xyz
; v1 = Base mesh's normal.xyz
; v2 = Base mesh's texture.xy
;
; v3 = Blend 1 mesh's position.xyz
; v4 = Blend 1 mesh's normal.xyz
; v5 = Blend 1 mesh's texture.xy
;
; v6 = Blend 2 mesh's position.xyz
; v7 = Blend 2 mesh's normal.xyz
; v8 = Blend 2 mesh's texture.xy
;
; v9 = Blend 3 mesh's position.xyz
; v10 = Blend 3 mesh's normal.xyz
; v11 = Blend 3 mesh's texture.xy
;
```

```
; v12 = Blend 4 mesh's position.xyz
; v13 = Blend 4 mesh's normal.xyz
; v14 = Blend 4 mesh's texture.xy
;
; c0-c3 = world+view+projection matrix
; c4    = Blend amounts 0-1 (mesh1, mesh2, mesh3, mesh4)
; c5    = light direction
vs.1.0
```

To begin, you see a bunch of comments to help you decipher which vertex registers and constants are in use. You need to set six vertex constants before calling the vertex shader.

- `c0` through `c3` need to contain the transposed world * view * projection transformation matrix.
- `c4` holds the blending amounts for each mesh (`c4.x` for mesh #1, `c4.y` for mesh #2, `c4.z` for mesh #3, and `c4.w` for mesh #4).
- `c5` is set to the light direction vector.

I'll get back to the constants in a bit; for now, I'll continue with the vertex shader code. After the comments, I added the vertex shader version requirement. As you can see, the blended morph vertex shader only requires vertex shader 1.0, which should be supported by most cards by the time you read this book.

After the comments and version requirement comes the vertex element mapping declarations. These declarations are used to map the vertex components you defined in the `D3DVERTEXELEMENT9` array to the shader's vertex registers (`v0` through `v14`). These mappings are identical to those seen in Table 10.1, so you should be able to follow along quite easily.

```
; declare mapping
dcl_position   v0 ; base mesh
dcl_normal     v1
dcl_texcoord   v2

dcl_position1  v3 ; 1st mesh
dcl_normal1    v4
dcl_texcoord1  v5

dcl_position2  v6 ; 2nd mesh
dcl_normal2    v7
dcl_texcoord2  v8

dcl_position3  v9 ; 3rd mesh
dcl_normal3    v10
dcl_texcoord3  v11

dcl_position4  v12 ; 4th mesh
dcl_normal4    v13
dcl_texcoord4  v14
```

With the vertex registers now mapped, you can access the various vertex data via the `v0` through `v14` registers. The `v0` register will contain the coordinates of the vertex in the base mesh; the `v6` register will contain the coordinates of the vertex in the second mesh; and so on.

You start the actual vertex shader code by putting the base mesh's vertex coordinates and normal into two temporary registers (`r0` and `r1`).

```
; Get base coordinates and normal into registers r0 and r1
mov r0, v0 ; coordinates (r0)
mov r1, v1 ; normal      (r1)
```

The shader only uses these two registers for reference, so they will not be overwritten during functions that follow (at least until the end of the shader, which you'll see soon). The next bit of code calculates the difference between the base mesh's coordinates and the mesh specified as the first blended mesh, which uses vertex registers `v3` through `v5`.

The difference is scaled (multiplied) by the constant value in `c4.x` (a value ranging from 0 to 1), added back to the original vertex coordinates from `r0`, and stored in the `r4` register. Through the remainder of the vertex shader, `r4` will contain the final coordinates of the blended mesh's vertex. The same process is repeated with the normal and tallied into the register `r5`. Take a look:

```
; Get differences from 1st blended mesh and add into result
sub r2, v3, r0         ; Get difference in coordinates
mad r4, r2, c4.x, r0   ; Put resulting coordinates into r4
sub r3, v4, r1         ; Get difference in normal
mad r5, r3, c4.x, r1   ; Put resulting normal into r5
```

The same process is repeated three more times, once each for the remaining blended meshes. From here on, however, the multiply and add instructions are tallied back in their respective registers so that the tallied coordinates and normal differences can be used later. Here's the rest of the code to calculate the difference values to use:

```
; Get differences from 2nd blended mesh and add into result
sub r2, v6, r0          ; Get difference in coordinates
mad r4, r2, c4.y, r4    ; Add resulting coordinates to r4
sub r3, v7, r1          ; Get difference in normal
mad r5, r3, c4.y, r5    ; Add resulting normal to r5

; Get differences from 3rd blended mesh and add into result
sub r2, v9,  r0         ; Get difference in coordinates
mad r4, r2,  c4.z, r4   ; Add resulting coordinates to r4
sub r3, v10, r1         ; Get difference in normal
mad r5, r3,  c4.z, r5   ; Add resulting normal to r5

; Get differences from 4th blended mesh and add into result
sub r2, v12, r0         ; Get difference in coordinates
mad r4, r2,  c4.w, r4   ; Add resulting coordinates to r4
sub r3, v13, r1         ; Get difference in normal
mad r5, r3,  c4.w, r5   ; Add resulting normal to r5
```

Now that you've got the final blended vertex coordinates to use (in `r4`), your vertex shader will need to transform the position for the combined world, view, and projection matrices (stored in constants `c0` through `c3`). As for the normal (stored in `r5`), you can perform a dot−product with the inversed light direction (stored in `c5`) to come up with the diffuse color component to use for shading the polygons.

```
; Project position using world*view*projection transformation
m4x4 oPos, r4, c0

; Dot−normal normal with inversed light direction
dp3 oD0, r5, −c5
```

Last, the texture coordinates are picked from the base mesh's vertex register `v2` and stuffed into the texture out register `t0`.

```
; Store texture coordinates
```

```
mov oT0.xy, v2
```

At this point, your vertex shader is complete. All you need to do is plug the shader into your project and figure out how to get the darned thing going.

Using the Blending Morph Vertex Shader

Now that you've written your vertex shader and you have the supporting vertex structure and declaration, you can finally get things rocking and rolling! Assume you have your base mesh and four target meshes already loaded into the following mesh objects.

```
ID3DXMesh *pBaseMesh;
ID3DXMesh *pMesh1, *pMesh2, *pMesh3, *pMesh4;
```

The same for your vertex shader and vertex declarationassume you already have loaded them and that you have the following valid interfaces to them:

```
IDirect3DVertexShader9 *pShader;
IDirect3DVertexDeclaration9 *pDecl;
```

After you have loaded the vertex shader, you can set it to render your blended meshes with the following lines of code:

```
pDevice->SetFVF(NULL); // Clear FVF usage
pDevice->SetVertexShader(pShader); // Set vertex shader
pDevice->SetVertexDeclaration(pDecl); // Set declarations
```

To begin drawing the blended mesh, you must set the vertex streams to point to each mesh's vertex buffer. Even if you're not using four blending meshes, you should always assign a vertex stream; just use the base mesh's vertex buffer multiple times as a stream if you have fewer than four meshes to blend.

```
// Get the size of a vertex
DWORD VertexStride = D3DXGetFVFVertexSize(pBaseMesh->GetFVF());
// Get the vertex buffer pointers
IDirect3DVertexBuffer9 *pBaseVB;
IDirect3DVertexBuffer9 *pMesh1VB, *pMesh2VB;
IDirect3DVertexBuffer9 *pMesh3VB, *pMesh4VB;

pBaseMesh->GetVertexBuffer(&pBaseVB);
pMesh1->GetVertexBuffer(&pMesh1VB);
pMesh2->GetVertexBuffer(&pMesh2VB);
pMesh3->GetVertexBuffer(&pMesh3VB);
pMesh4->GetVertexBuffer(&pMesh4VB);
// Set the vertex streams
pDevice->SetStreamSource(0, pBaseVB,  VertexStride);
pDevice->SetStreamSource(1, pMesh1VB, VertexStride);
pDevice->SetStreamSource(2, pMesh2VB, VertexStride);
pDevice->SetStreamSource(3, pMesh3VB, VertexStride);
pDevice->SetStreamSource(4, pMesh4VB, VertexStride);
```

Now you need to grab the current world, view, and projection matrices from your 3D device. These matrices are combined, transposed, and stored in the vertex shader constant registers `c0` through `c3`. The following code handles things quite nicely.

```
// Get the world, view, and projection matrices
```

```
D3DXMATRIX matWorld, matView, matProj;
pDevice->GetTransform(D3DTS_WORLD, &matWorld);
pDevice->GetTransform(D3DTS_VIEW, &matView);
pDevice->GetTransform(D3DTS_PROJECTION, &matProj);

// Get the world*view*proj matrix and set it
D3DXMATRIX matWVP;
matWVP = matWorld * matView * matProj;
D3DXMatrixTranspose(&matWVP, &matWVP);
g_pD3DDevice->SetVertexShaderConstantF(0, (float*)&matWVP, 4);
```

To control the amount of blending for each blend mesh, just alter the values stored in the vertex shader constant `c4`. The `x` register of constant `c4` represents the amount of blending in the first mesh, and it ranges from 0.0 to 1.0 (or more if you want to achieve some exaggerated results).

Note Realistically, you shouldn't call `GetTransform` to get your various transformation matrices. These should be maintained by your application, possibly on (but not limited to) a global level.

The same goes for `c4.y, c4.z,` and `c4.w`each mesh has one register to store its blending amounts (`y` for mesh #2, `z` for mesh #3, and `w` for mesh #4). For now, set the blending values to 100 percent (by storing a value of 1.0 for each blending value in a `D3DXVECTOR4` object), and store the values in the vertex shader's `c4` constant using the `SetVertexShaderConstantF` function.

```
// Set the blending amounts
D3DXVECTOR4 vecBlending = D3DXVECTOR4(1.0f, 1.0f, 1.0f, 1.0f);
pDevice->SetVertexShaderConstantF(4, (float*)&vecBlending, 1);
```

Last, you need to specify your scene's light direction (the same light direction you used in a `D3DLIGHT9` structure) in the vertex shader constant `c5`. For example, if you want a light (positioned in object space) to point downward in your scene, you can use a vector of 0, 1, 0. (Note that you use a `D3DXVECTOR4` object to contain the light direction as opposed to a `D3DXVECTOR3` objectjust specify 0.0 for the `w` component.)

```
// Set the light vector
D3DXVECTOR3 vecLight = D3DXVECTOR4(0.0f, 1.0f, 0.0f, 0.0f);
pDevice->SetVertexShaderConstantF(5, (float*)&vecLight, 1);
```

Because you're grabbing the vertex data from a series of `ID3DXMesh` objects, you need to set the index buffer as well because all `ID3DXMesh` objects use indexed primitive lists. You only need to set the index buffer of the base mesh object because the indices are the same for all of your mesh objects. The following code sets the base mesh's index buffer for you:

```
// Set the index buffer
IDirect3DIndexBuffer9 *pIndices;
pBaseMesh-> GetIndexBuffer(&pIndices);
pDevice-> SetIndices(pIndices, 0);
```

Finally, you can render your blended mesh! Skipping the typical code for setting your rendering material and textures, you can render the entire mesh using the following code:

```
// Render the mesh
pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,      \
          0, pBaseMesh->GetNumVertices(),              \
          0, pBaseMesh->GetNumFaces());
```

If you want to check out a fully−functional example of the blended morphing vertex shader, take a look at the BlendMorphVS demo on the CD−ROM. In Chapter 11, "Morphing Facial Animation," you'll see how to put your vertex shader to great use by creating some awesome facial animations!

Check Out the Demos

The demos provided for Chapter 10 (BlendMorph and BlendMorphVS) show you how to use the information from this chapter to blend a series of meshes into one animated mesh. Both demos, while alike in appearance (see Figure 10.3), use different techniques to render the blended morphing animation.

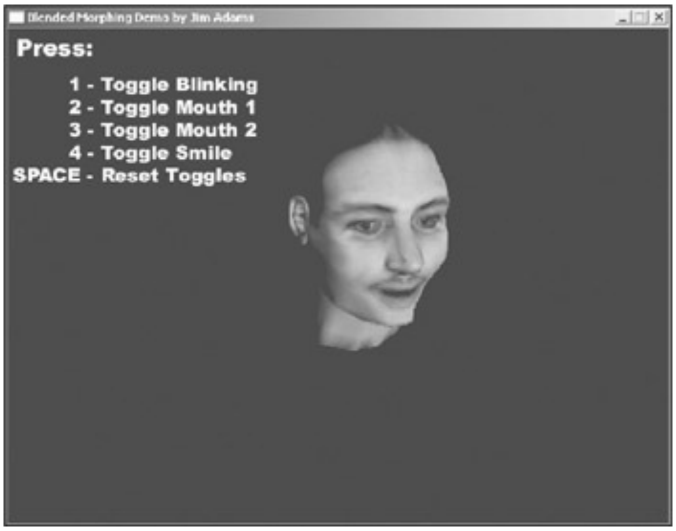


Figure 10.3: A prelude to facial animation. Watch as multiple meshes are blended at various levels to produce a simplistic facial animation.

In BlendMorph, the animation is accomplished by locking, updating, and unlocking the vertex buffers of the various meshes. In BlendMorphVS, the locking, updating, and unlocking bits are tossed out, and the demo shows its power by modifying the vertex data via a vertex shader! With either demo, you're sure to be pleased with the results!

Programs on the CD

In the Chapter 10 directory on the CD−ROM, you'll find two projects that demonstrate the use of blended morphing animation. These projects are

- **BlendMorph.** In this project you'll see how to create animation by blending a morphing animation by directly manipulating the vertices of your meshes. This project is located at \BookCode\Chap10\BlendMorph.
- **BlendMorphVS.** Check out how to perform blended morphing animations using the vertex shader developed in this chapter. This project is located at \BookCode\Chap10\BlendMorphVS.

Chapter 11: Morphing Facial Animation

To make a long explanation short, facial animation is the technique of animating a mesh that represents a character's face in a realistic manner, including movements of the mouth, eyes, and brows. Using facial animation adds that extra panache to your game that makes gamers' jaws drop. With perfectly audio−synchronized mouth motions and varying facial expressions, your game's characters come to life. Those characters are no longer stiff, lifeless dummies, but living, breathing, talking, screaming, and smiling game denizens.

Current games that use advanced facial animation, such as Electronic Art's *Medal of Honor: Frontline* and Interplay's *Baldur's Gate: Dark Alliance* have raised the bar, and the use of facial animation is quickly becoming mainstream. The facial animations in both of these games are nothing short of incredible. The added feeling of reality just increases the gamer's potential enjoyment of each game. Now, don't you want that added realism in your game project? If so, then this chapter is right up your alley.

The Basics of Facial Animation

You will use four main features in your facesthe eyes, brow, mouth, and head orientation. The most basic of the four features is the head orientation. With just a cursory glance, you can see that most people constantly move their heads around as they go about their daily duties. Rarely does a person hold his head still. People's most evident head movements often occur when they talk; a person's head is constantly moving as he speaks. You want your facial animation engine to imitate the same motions.

The next thing you might notice is that people's eyes are also constantly moving. Do you want this feature in your animation package? If so, you have to think about what your characters are looking at. Most people have a reason for moving their eyesthey are watching their surroundings and the people around them.

To keep things simple, you should limit eye movements. If your characters need to look around, you should separate the eyes from the face (as separate meshes) and maintain them that way. It's much easier to rotate a few eyeballs than to construct a bunch of morphing meshes that represent all the possible eye orientations.

The eyelids and eyebrows are also related to the eyes. Like you and I, your characters need to blink their eyes periodically to appear realistic. Using blended morphing, adding the ability to blink is fairly straightforwardall you need is a morph target mesh that represents your base mesh with its eyes closed. By varying the blending amount of the blinking mesh over time, you can create a believable blinking animation.

Did I just mention using blended morphing animation? You bet I did! Blended morphing animation is easy to work with, and using morphing animation is perfect when it comes to facial animations. Take a closer look to see why you'll want to use blended morphing meshes with your own facial animation engine.

Blended Morphing Back in Action

As I mentioned in Chapter 10, your blended facial animation depends on the use of a base mesh. The base mesh determines the default orientation that other meshes use to derive which vertices deform during animation. Only those vertices that deviate in position from a mesh and the base mesh are altered and used to render the final blended mesh.

The base mesh represents your facial mesh with no expression whatsoever. For each expression you want your mesh to take on, such as blinking eyelids, raising of the brow, and the various mouth shapes, you merely blend the target morphing mesh with the base mesh.

Take the series of meshes shown in Figure 11.1 as an example of what blended morphing does for facial animation.

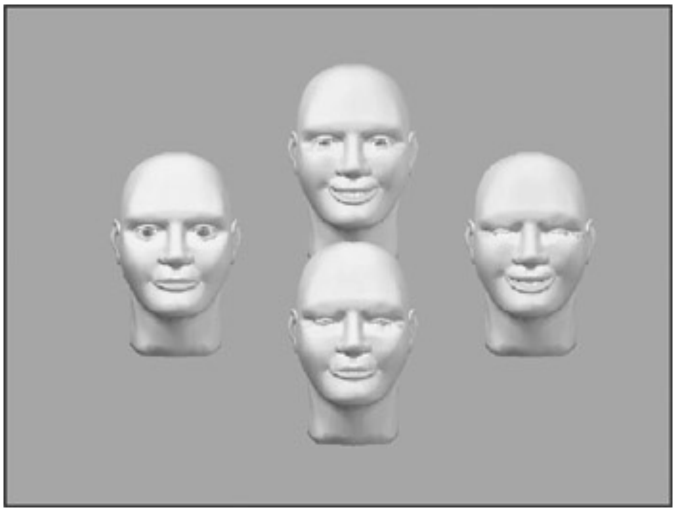


Figure 11.1: You can blend the base mesh on the left with multiple meshes to create unique animations. Here, a smiling expression is blended with a blinking expression.

Again, simple morph targets are used to animate each unique aspect of the facial mesh. The eyes blinking is one aspect, and the mesh's eyebrows could be another aspect. You could raise one or both of the eyebrows. They are tied into the emotion of the character, so they can be hard−coded in some way inside your animation package. For instance, suppose you have a flag that determines whether a character is angry. When you render a facial animation, your engine can blend in a target mesh that has both eyebrows lowered. If a character is asking a question, perhaps your engine will raise one of the mesh's eyebrows.

Speaking of emotions, you can use more than just the brow to help convey a character's feelings. The mouth might change shape as well. Typically, the edges of a character's mouth are raised or lowered as his emotion changes. Unhappy characters lower the edges of their lips, forming a frown, whereas happy characters raise the edges of their lips into a smile.

You can use target morphing meshes to convey emotions as a whole for the mesh. You can modify a single mesh to have the brow lowered and lips slightly lowered to convey a believable sense of anger. As Figure 11.2 demonstrates, there's no need to use two separately combined morph meshes when one mesh will do the job just as well!

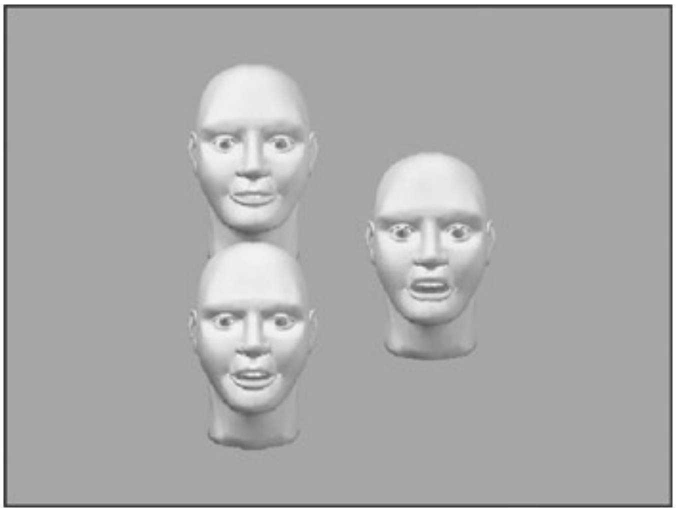


Figure 11.2: Instead of using two target morph meshes, you can combine the two to use as the target mesh, thus saving time and space.

Let's get back to the mouth. In addition to helping display emotion, the mouth is great for communicating. The lips are powerful little muscle machines that can contort themselves into many shapes. As you speak, your mouth changes shape to help create the sounds that compose your speech.

The various shapes your mouth makes during speech are called *visemes,* whereas the sounds you make are called *phonemes.* In this chapter, I commonly mix the usage of visemes and phonemes because they mean pretty much the same thing when it comes to animation. Take a moment to learn more about phonemes and how you can use them for your own lipsyncing animation needs.

Using Phonemes for Speech

As I just mentioned, phonemes are the smallest sounds we can make; each word in our language (or any language) is constructed from a series of these phonemes. For example, the word *program* is constructed of the following seven phonemes: p, r, ow, gh, r, ae, and m.

To pronounce each of those phonemes, your mouth takes on a unique shape (a viseme). Hence, you can see why I mix the two terms; there's one viseme per phoneme. In facial animation, you want to create a target morphing mesh with a mouth that has the same shape as yours when you pronounce the various phonemes.

Note Rather than referencing IPA Unicode values using decimal notation, it is much easier (and it is the standard) to use a hexadecimal notation. Therefore, throughout this chapter I will reference the IPA values in a hexadecimal value that ranges from 0x0000 to 0x0FFF.

I'll get back to creating the phoneme facial meshes in a bit. For now, I want you to study phonemes a little closer to see how you can use them in your project. Phonemes are identified by unique sets of symbols; each symbol is assigned a unique value to make it more identifiable. These values, known as IPA (*International Phonetic Alphabet*) Unicode values, range from 0 to 1024 (for English users), with each grouping of values assigned to various languages and pronunciations (as shown in Table 11.1).

Table 11.1: IPA Phoneme Unicode Groupings

Value Range	Language
0x0041 to 0x00FF	Standard Latin
0x0010 to 0x01F0	European and extended Latin
0x0250 to 0x02AF	Standard phonetic characters
0x02B0 to 0x02FF	Modifier letters
0x0300 to 0x036F	Diacritical marks

English speakers use a wide assortment of the values shown in Table 11.1, but for the most part the values (and phonemes) will be the ones shown in Table 11.2

Table 11.2: American English Phonemes

Value	Phoneme	Example
0x0069	iy	Feel
0x026A	ih	Fill
0x00E6	ae	Carry
0x0251	aa	Father
0x028C	ah	Cut
0x0254	ao	Lawn
0x0259	ax	Ago
0x0065	ey	Ate
0x025B	eh	Ten
0x025A	er	Turn
0x006F	ow	Own
0x028A	uh	Pull
0x0075	uw	Crew
0x0062	b	Big
0x0070	p	Put
0x0064	d	Dug
0x0074	t	Talk
0x0067	g	Go
0x006B	k	Cut
0x0066	f	Forever
0x0076	v	Veil
0x0073	s	Sit
0x007A	z	Lazy
0x03B8	th	Think
0x00F0	dh	Then
0x0283	sh	She
0x0292	zh	Azure

0x006C	l	Length
0x0279	r	Rip
0x006A	y	Yacht
0x0077	w	Water
0x0068	hh	Help
0x006D	m	Marry
0x006E	n	Never
0x014B	nx	Sing
0x02A7	ch	Chin
0x02a4	jh	Joy

You use the IPA values shown in Table 11.2 as an index into your array of phoneme facial meshes used when rendering. To construct an animation sequence, you string together a sequence of those IPA values to form words and sentences. You'll learn more about building phoneme sequences in the "Building Phoneme Sequences" section later in this chapter.

As for the facial animations, it's a matter of animating (or rather, blending) the various meshes that represent the phonemes and facial expressions to create a complete facial animation system. Of course, that means you'll have to build a series of meshes to use in your animations.

Building Facial Meshes

The first step to using facial animation is to create a set of facial meshes that represent the various features of your game characters' faces. Since you're using blended morphing animation techniques, you only need to create a single base mesh and a series of target morphing meshes for each facial feature you'll be using. For instance, you might only need a few meshes that show your character smiling, blinking his eyes, and moving his mouth to match a phoneme sequence.

This is where the hardest part of your facial animation job liesin designing and building the various facial meshes to use in your engine. With the help of various 3D modeling programs, such as Caligari's trueSpace and Curious Labs' Poser, you can create a whole slew of facial meshes quickly and easily.

Caligari's trueSpace (versions 5.1 and newer) comes packaged with Facial Animator, a cool plug−in that helps you create, texture, and animate facial meshes. I used the Facial Animator plug−in to create the demo for this chapter.

Poser is a complete character−creation package that gives you the ability to model an entire person. With shapes, clothing, textures, and facial features, the Poser 3D package will definitely come in handy.

Regardless of the 3D package you use, it all boils down to the same thingcreating a base facial mesh to use.

Creating the Base Mesh

Both 3D packages I mentioned come with a generic set of facial meshes to use. With trueSpace, you can easily import your own meshes and prepare them for use with the Facial Animator plug−in. With Poser, you can use the facial generator to develop an almost unlimited number of unique faces.

Again, regardless of which package you use, you need to create a base mesh. Remember that this base mesh must be devoid of expression; the mouth should be closed and the eyes fully open. To make things easier, I'm going to use one of the facial meshes that comes packaged with trueSpace's Facial Animator. Figure 11.3 shows the mesh I'll use in this chapter.

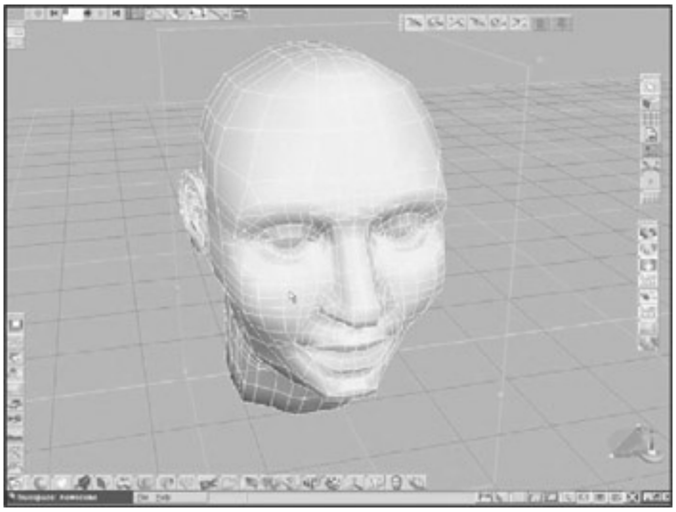


Figure 11.3: The Chris (low poly) facial mesh from trueSpace will serve perfectly as the base mesh. Per Caligari's user license, feel free to use the Chris (low poly) model in your own applications

After picking the mesh to use as your base mesh, you need to texture−map it appropriately. The cool thing about trueSpace's Facial Animator plug−in is that you can design texture maps to use for your facial meshes by taking a side and front view picture of yourself and using the Texturize Head tool to fit it to the mesh. For this book's demo, I used my own face to texture the mesh.

After two short steps, the base mesh is ready! I know I skipped over the specifics such as modeling the head, but this book isn't about modeling, it's about animating! Truthfully, the two 3D packages I mentioned, trueSpace and Poser, do an admirable job of making the facial modeling process extremely easy, so I'll leave it to each package's manuals and tutorials to teach you how to create a head mesh.

Note The Chris (low poly) facial mesh I'm using as the base mesh is missing a couple features, most notably the eyes. I used trueSpace's Add Face tool to add a few polygons where the eyes would be, which allowed me to apply an eyeball texture map.

For now, your base mesh is ready, and you can start building the facial expressions you want to include in your animations.

Creating Facial Expressions

Before you continue, be sure to save your base mesh to disk using a descriptive file name, such as Base.x. Remember, you're using the .X format, so you might want to export the mesh as .X. If that's not available, export the mesh as a .3DS file. Once you have saved the mesh as a .3DS file, you can use the Conv3DS.exe program that comes with the DirectX SDK to convert the file to .X. Typically you will find the Conv3DS.exe program in your DirectX SDK's \Bin\DXUtils directory, or you can download the newest version from Microsoft's Web site at http://www.microsoft.com/DirectX.

Now that you've saved your base mesh to disk, you can start creating the various facial features you'll be using. It is easiest to start with the expressions, such as smiling, frowning, and blinking. Again, I want to make this as simple as possible, so I'll depend on trueSpace's Facial Animator to help.

It just so happens that Facial Animator comes with a list of predefined expressions you can apply to your facial mesh with just a click of your mouse button! Come to think of it, Poser has the same abilities, so your bases are covered regardless of which program you use!

To create your mesh's expressions, click on the Expressions tab in the Facial Animator dialog box. As you can see in Figure 11.4, a list of expressions your mesh is capable of using will appear.

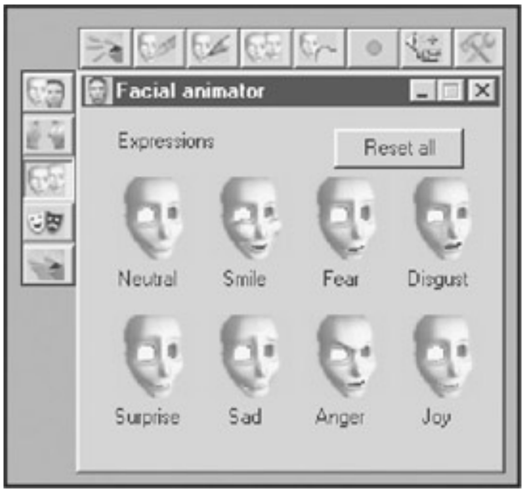


Figure 11.4: Facial Animator's Expressions list gives you eight expressions from which to choose.

I want my facial animation demo to be simple enough to follow, and since I'm such a happy fellow, I want to use the Smile expression. Click on the Smile button, and notice that the facial mesh in the 3D editor changes to match the expression. If you feel adventurous, click on the other expression buttons to see the effect of each on the mesh. When you're ready, click on the Smile button to get back to the smiling mesh setup.

After you've selected the expression you want to use (in this case, the smile), export the mesh. Call the mesh Smile.x to keep things simple. Place the Smile.x file in the same directory as the Base.x file. If you want to use more expressions, click on the appropriate expression button in the Facial Animator dialog box, wait for the mesh to change, and then export it to an .X file.

I don't want to fool you into thinking there are only eight expressions to use with Facial Animator, so click on the Gestures tab. Voila! Fourteen more expressions should appear (see Figure 11.5).

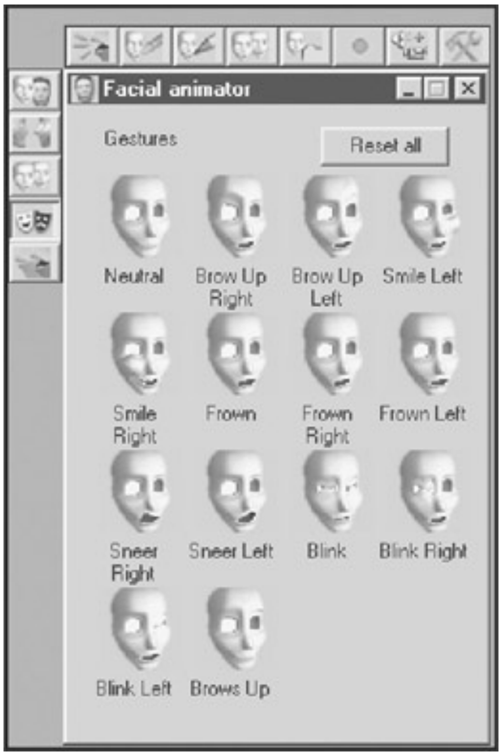


Figure 11.5: Facial Animator's Gestures list includes 14 more expressions you can apply to your mesh.
Before you use any of the expressions on the Gestures tab, click the Reset All button once. This will cause the facial mesh to revert to its base orientation. Feel free to play around with the collection of gestures. Decide which gestures you want to use and export the facial mesh with the appropriate gestures applied. For my demo, I am using only the Blink gesture.

Caution As you export your facial animation meshes and begin using them in your projects, make sure you don't change the ordering of the vertices. This is essential to proper morphing, as detailed in Chapter 8.

After you export all your expressions and gestures, move on to creating the various meshes for your phoneme sequences.

Creating Viseme Meshes

The majority of meshes you'll construct will be the phoneme shapes your mesh's mouth can form. While you don't need a complete set of meshes to match every conceivable phoneme shape, it does help to create a small set that represents the majority of shapes your phoneme sequences use

As I mentioned previously, the unique sounds you use to create words are called *phonemes*. The shape of your mouth and the position of your tongue, which create the sounds, are called *visemes*. To create *lip*−synced animation, you need to construct a set of *visemes* your game can use to match the phonemes in a recorded voice.

Depending on how realistic your want your lip−syncing animations to look, you can get away with using as few as four viseme shapes for lower−quality animations, or you can go all the way up to 30 or more viseme shapes for high−quality animations.

With trueSpace's Facial Animator plug−in, you can use the set of eight visemes located in the Viseme section, which you can access by clicking on the Viseme tab. Although they are somewhat limited, the eight visemes are a good place to start your facial animations. I would highly recommend expanding this list of visemes as soon as possible, using Facial Animator's Head Geometry Setup tools. Consult your product's manual to see how you can add your own facial expressions and visemes to the list of shapes you can use. For the examples in this book, the eight visemes are sufficient.

Using the same techniques you used in the previous section, click on each of the visemes that you want to use for your animations, and export each mesh as you go. For my demo, I named each exported file based on its viseme nameff.x, consonant.x, ii.x, aa.x, oo.x, uu.x, and ee.x. These .X files go in the same directory as the base.X and Smile.x files.

If you're going the other route and creating more viseme meshes, you need to take the time to create the various shapes for your mesh's mouth. As I mentioned, you can use more than 30 different visemes in your animations, so make sure you're up to the task.

Looking back at Table 11.2, you can see the most common phonemes used in the English language. It's those phonemes for which you want to create matching facial meshes. The easiest way to create these phoneme meshes is to grab a mirror and watch your mouth as you pronounce the various phonemes. Model the exact shape of your mouth, save the result, and move on to the next phoneme.

To help you out a bit, Michael B. Comet (creator of Comet Cartoons at http://www.comet−cartoons.com) has provided various phoneme shapes you can use (see Figure 11.6).

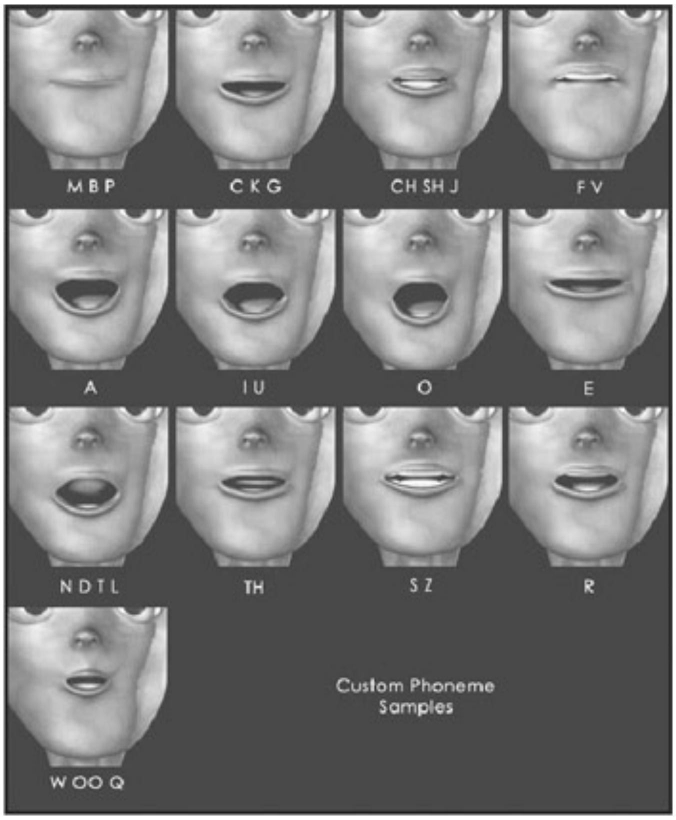


Figure 11.6: Michael B. Comet's demonstration set of phonemes should provide you with a guide to creating your own facial meshes.

Try to match the shapes as best you can when creating your own meshes. Since the animations are so fast, slight deviations from the actual phoneme shapes shouldn't be a problem. After you've created all the phoneme facial meshes you're going to use, it's time for the fun partputting those meshes to work in your own animations!

Creating Animation Sequences

Now you're ready to get going with your animations! At this point, you should have a basic understanding of what phonemes are, and you should have constructed a series of target meshes to use in blended morphing animation. These meshes should include the various features you'll want to work with, such as blinking eyes, raising brows, and changing mouth shapes (the visemes).

Before you start blending your way to animation success (now that would be a great tag line for an infomercial!), there are some things you need to handle. I'll start with the easiest features of your facial animations and then move on to the tough stuff! What's easy when it comes to facial animation, you ask? The automated features, of course!

Automating Basic Features

Before you jump into the hardest part of facial animationthe phoneme sequencestake a quick look at the things you normally take for granted. The blinking of your eyes, for instance, is something all too common, and you really don't notice itunless the person you are talking to doesn't blink at all, in which case it becomes very apparent!

The same goes for a person's expressions. As you talk, your face changes expression to match the tone of your voice. Cheerful people tend to smile when they talk, and people frown when they are sad. As your emotions change during speech, your expressions changeto match them. Every subtle movement of your lips, eyes, and brow can change the look of emotion on your face. Without this emotional appearance, we'd all look like mannequins!

Notice also that a person's head always moves when he speaks. From small nods and bobs to turns, most people's heads are constantly moving. With emotions aside, the small movements of a person's head and the blinking of his eyes are considered automated. In other words, those features should be animated without you needing to explicitly define them in your sequences.

Let me go back to the eyes to explain this concept better. An adult normally blinks his eyes around 10 to 20 times per minute. When you concentrate on something, you tend to blink less often, about three to eight times per minute. Also, as you glance back and forth, you tend to blink more often.

Other factors, such as emotions, change the rate at which you blink your eyes. A nervous or excited person blinks more often, at close to twice the normal rate, and an angry person blinks less. The golden rule: If it involves concentration, the blink rate drops; if it involves a lot of movement, the blink rate increases.

To automate the blinking of your facial mesh's eyes, you can create a timer that forces the blending of the blinking mesh every so often. If you want to maintain a constant rate, you can force the eyes to blink every 3,000 milliseconds (every three seconds). If you want to be really creative, you can add the ability to mark sections of your script to alter the blink rate.

For this chapter, I want to maintain a constant blink rate. The blinking target mesh will therefore be blended every three seconds, and will take one−third of a second to animate. This means that the blending scale of the blend morphing mesh will range from 0 to 1 over the first 33 milliseconds of the animation and remain fixed at 0 for the remainder of the time.

Now what about head movements while talking? Again, this feature is easy to incorporate by applying small, random values to the rotational values of the mesh you are drawing. Nothing majorjust make sure you don't over−rotate the mesh, or you'll end up with some distorted B−rate movie monster!

The facial animation demo for this chapter demonstrates the use of automated blinking and head movements; you can use this demo as a starting ground for your own facial animations. (Refer to the discussion of the FacialAnim demo at the end of this chapter for more information.)

With the automated animations out of the way, it's time to move on to a much more difficult subjectcreating your phoneme animation sequences.

Building Phoneme Sequences

Aside from the rudimentary animations you can use, such as eye blinking or face twitching, the most important aspect of facial animation is lip−syncing. Remember, your lips change shape to match each phoneme in your speech, and it's those phoneme sequences that you want to reproduce during animation.

When it comes to lip−syncing animation, there are a number of methods at your disposal. Currently, the most popular method of creating lip−synced animation sequences is to use scripts (spoken and written) in combination with a *phoneme dictionary* to break every spoken (and written) word down into its phoneme sequence.

A script contains the exact sequence that is to be spoken and lip−synced. For example, suppose you want your game's character to lip−sync to you saying "Hello and welcome." Using a text editor, enter that exact phrase and save it as a text file. Next, using a sound−editing program of some sort, record yourself saying that phrase. Make sure to save it as a standard .WAV file.

After you've done this, you can break up your phrase into its phoneme sequence. You accomplish this by using a phoneme dictionary, which is a data file that contains phoneme sequences for each word contained in the dictionary. By piecing together the sequences for each word, you can create sequences to match any script.

For instance, the phoneme dictionary definition for the word "Hello" would be the phonemes hh, ah, l, and ow. Each of those four phonemes has a matching facial mesh associated with it. As those various phonemes are processed in your animation playback, your facial mesh morphs to match the sounds. For the entire "Hello and welcome" sequence, you would use the phonemes hh, ah, l, ow, ae, n, d, w, eh, l, k, ah, and m. Using the script and dictionary in unison makes the real magic happens.

So far this is a cool idea, but how does it work? Take a look at the entire process of writing, recording, and processing data that will eventually become your lip−sync animation. Start by writing a script file (as a text file). This script should contain everything that is to be spoken by you or your voice actor.

Next, using a high−quality recording format, record the script word for word. Be sure to speak slowly and clearly, adding brief silences between words. When you are finished with the script, you can clean up the sound file by maximizing the volume and cleaning out any static. It also helps to run the sound through a filter to convert the sounds between words to silence.

Figure 11.7 shows you a wave form for a couple of words I recorded to demonstrate this lip−syncing technique.

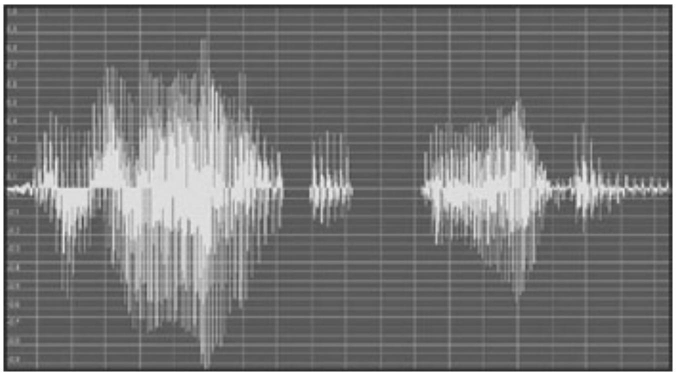


Figure 11.7: A wave form of me saying "Hello and welcome!"

Now here's where the cool part comes in! Using the phoneme dictionary, take the written script and convert every word to a sequence of phonemes. Using the sound file, examine each word (by its sound wave) and determine its spoken length. The length of each word determines the animation speed for each phoneme sequence.

How do you determine the length of each word in the sound file? For that matter, how do you know what each word is? Remember, you have the script that has the exact sequence of words spoken. That takes care of knowing what each word is, but what about determining the position and length of each word?

Remember how I said you should speak slowly and deliberately, adding a slight pause between words? Those pauses are like signal markers. A pause marks the end of one word and the beginning of another. Take a look back at the sound wave from Figure 11.7. This time, I've marked the silence between words and used the pauses to isolate each word. Figure 11.8 shows the new sound wave, with each pause and word marked.

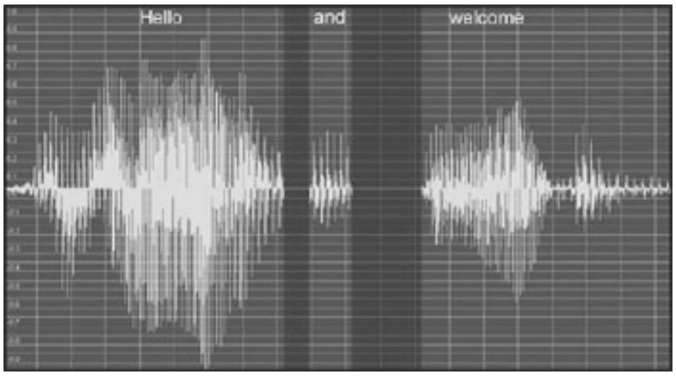


Figure 11.8: The sound wave has been sectioned into words and silence.

Now I know you're starting to understand the significance of the script combined with the dictionary! Using the time value (based on the sound's playback frequency and position), you can go back to your phoneme sequence and use the time values to animate it. What could be simpler?

To get the ball rolling, check out a program that works much like what you've just studiedMicrosoft's Linguistic Information Sound Editing Tool.

Using Microsoft's Linguistics Software

With the concept of using scripts, spoken voice, and phoneme dictionaries in the bag, it's time to turn to an actual program that compiles phoneme sequences for you. This program, Microsoft's Linguistic Information Sound Editing Tool, or LISET for short, analyzes. WAV files of recorded voice. In conjunction with a typed script, it will produce a phoneme sequence you can plug right into your game.

The LISET program is part of Microsoft's excellent Agent software package. Agent is a complete speech and animation package that allows users to interact with animated characters embedded in applications and Web pages. Imagine thathaving a virtual guide lead visitors around your Web site, and for that matter, having your guide look and sound just like you!

With full speech processing, such as text−to−speech synthesis, phoneme processing, and lip−syncing abilities, the Agent package is definitely something you should check out. For more information about Agent, check out Microsoft's Web site at http://www.microsoft.com/msagent.

If you haven't done so yet, install Agent on your system. Make sure to install LISET as well, because you'll be using it quite a bit in this section. Once you have installed everything, fire up LISET. (Look for it in your Programs menu.) Figure 11.9 shows the main LISET window you'll use.

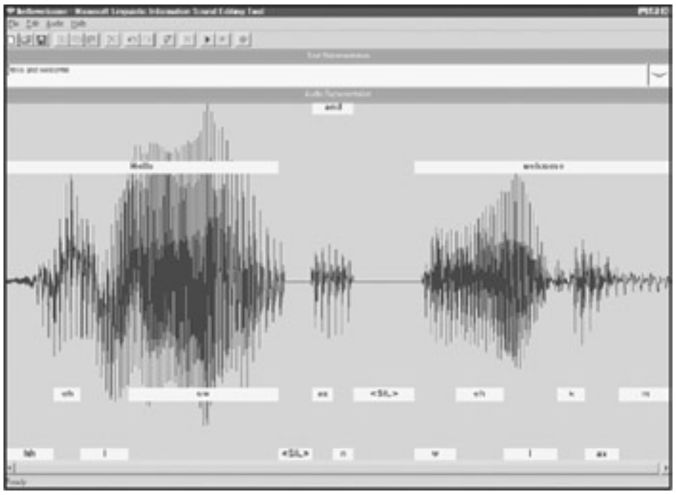


Figure 11.9: Microsoft's Linguistic Information Sound Editing Tool allows you to load. WAV files and mark portions of the sound waves with phoneme markers.

Once LISET is running, you can load a .WAV file that contains your spoken voice scripts. Click on File and select Open. The Open dialog box will appear, allowing you to locate a .WAV file. Find a .WAV file, select it, and click on Open to continue.

Once loaded, the .WAV file is ready to be processed. However, LISET needs the text script to match to the sound wave. See the Text Representation edit box at the top of the LISET application? That's where you'll paste your text script. Take it easy, however, because the edit box can hold only a finite number of letters. Try to parse your scripts in chunks of 65,000 words or fewer at a time.

Take a look back at Figure 11.9, where you can see that I've loaded a .WAV file and entered the corresponding text. To test the script's phoneme sequence, click Audio and select Play. See that little mouth at the top−right corner of LISET? As your sound plays, that little mouth changes shape to match the phonemes in the script. Cool, isn't it?

For some real fun, click on Edit and select Generate Linguistic Info. After a few moments (and a pop−up progress window), your phoneme sequence will be analyzed and the phoneme information will be overlaid on the sound wave. Check out Figure 11.10 to see what I mean.

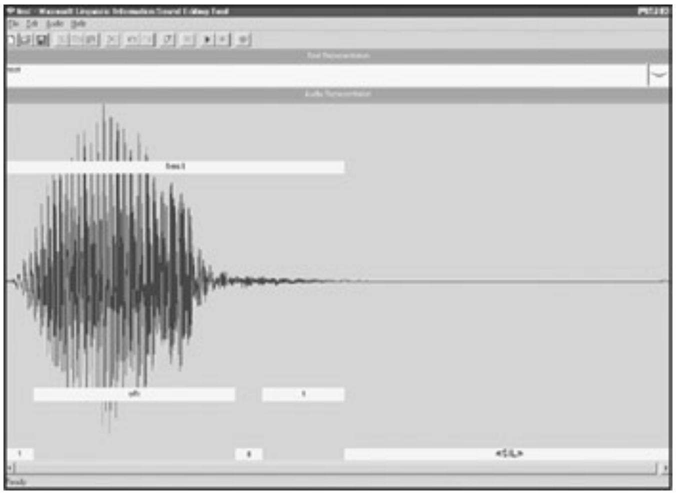


Figure 11.10: The word "test" consists of four phonemes (t, eh, s, and t), which are overlaid on top of the sound wave. Notice that the silence at the end of the word is also marked.

LISET does a pretty decent job of putting the phonemes in the proper positions over the sound wave, but if something isn't quite right, you can modify the positions and lengths of each phoneme manually.

As you move your mouse over each phoneme, you'll notice that the arrow changes. At the left and right edges of the phoneme, the arrow changes to a double bar with arrows, meaning that if you click and drag the mouse, the phoneme will be resized. Clicking inside the phoneme itself (not on the edges) will highlight portions of the sound wave.

You can highlight portions of the sound wave, click Edit, and select Replace Phoneme, Delete Phoneme, Insert Phoneme, or Insert Word. Replace Phoneme allows you to change from one phoneme to another, and Delete Phoneme removes an entire phoneme from the sequence. Using the two Insert functions, you can manually place a new word or phoneme into the sound wave.

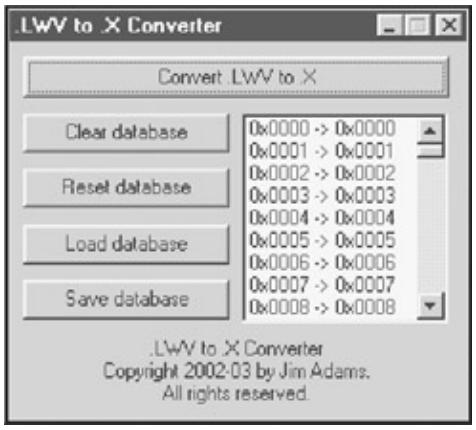
After you've processed and placed the phoneme sequence LISET displays, you can save the phoneme information. Click on File and select Save As. Select a path and file name to save the resulting file, and you're done!

As a result of using LISET, your phoneme sequences are stored in special files signified by the extension .LWV. The only problem is, how do you use those .LWV files in your programs? How about converting those files to something more readable?

Converting from LISET to .X

The .LWV file format used by the LISET application contains all the phoneme information your program needs to create your lip−synced facial animations. The only problem is that the .LWV file format is something you don't want to mess around with in your project. What you need is a way to convert the phoneme sequences from the .LWV format into a more readable format, such as .X.

That's rightit's .X to the rescue again! This book's CD−ROM includes a utility program called .LWV to .X Converter (ConvLWV for short), which converts .LWV files into .X files. Check out the end of this chapter for the location of the ConvLWV program. The program is fairly straightforward and easy to use. To convert an .LWV into an .X file, all you have to do is click the Convert .LWV to .X button, as shown in Figure 11.11.



```
0x0000 -> 0x0000
0x0001 -> 0x0001
0x0002 -> 0x0002
0x0003 -> 0x0003
0x0004 -> 0x0004
0x0005 -> 0x0005
0x0006 -> 0x0006
0x0007 -> 0x0007
0x0008 -> 0x0008
```

.LWV to .X Converter
Copyright 2002-03 by Jim Adams.
All rights reserved.

Figure 11.11: The ConvLWV program has six controls at your disposal, the most important being the Convert .LWV to .X button.

When you click on the Convert .LWV to .X button, you are presented with the Select .LWV File for Import dialog box. Use the controls in the dialog box to locate the source .LWV file that you want to convert. After you've selected the file, click Open.

Next, you'll see the Select .X File for Export dialog box. Enter a file name and path to which to export the phoneme sequence, and then click Save. Poof! Within a few seconds, you'll have a brand−new .X file that contains your phoneme sequence!

The format of the exported .X file should resemble the following:

```
xof 0303txt 0032

// Exported with .LWV to .X Converter v1.0 (c) 2002−03 by Jim Adams

Header
{
    1;
    0;
    1;
}
```

Like a typical .X file, your exported phoneme sequence .X file starts by defining the format header. This is followed by a shameless plug and finally a `Header` data object. There's nothing you haven't seen already going on there. You'll want to pay close attention to what comes next, though.

```
// DEFINE_GUID(Phoneme, 0x12b0fb22, 0x4f25, 0x4adb, 0xba, 0x0, 0xe5, 0xb9, 0xc1, 0x8a,
0x83, 0x9d)
template Phoneme
{
  <12B0FB22−4F25−4adb−BA00−E5B9C18A839D>
  DWORD PhonemeIndex;
  DWORD StartTime;
  DWORD EndTime;
}

// DEFINE_GUID(PhonemeSequence,
//         0x918dee50, 0x657c, 0x48b0,
//         0x94, 0xa5, 0x15, 0xec, 0x23, 0xe6, 0x3b, 0xc9);
template PhonemeSequence
{
   <918DEE50−657C−48b0−94A5−15EC23E63BC9>
   DWORD NumPhonemes;
   array Phoneme Phonemes[NumPhonemes];
}
```

Your phoneme sequences are stored in .X files using two custom templates. The first template, `Phoneme,` stores information about a single phoneme. There's the phoneme identification number (the phoneme mesh number, which you'll soon read about), as well as the beginning and ending time (in milliseconds) for the phoneme to animate.

That's rightthe `Phoneme` template is actually an animation key frame! That's where the `PhonemeSequence` template comes into play. The `PhonemeSequence` template stores an array of `Phoneme` objects that define an entire animation sequence. The number of key frames in the animation is defined by the first value in `PhonemeSequence,` followed by the array of key frames. This structure makes loading your phoneme animations quick and painless.

Let's continue looking deeper into the .X file you've just created (or rather, one that I've created to demonstrate the converter). The following `PhonemeSequence` object, called `PSEQ`, contains a short animation with five key frames.

```
PhonemeSequence PSEQ {
  5;
  116; 0; 30;, // 0x0074
  603; 30; 200;, // 0x025b
  115; 200; 230;, // 0x0073
  116; 230; 270;, // 0x0074
  95; 270; 468;; // 0x005f
}
```

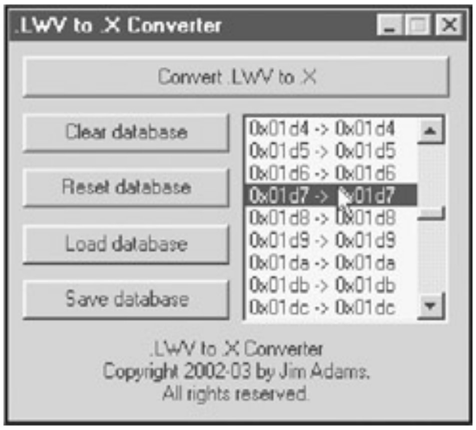
As you can see, each phoneme is represented by its decimal IPA Unicode value (with each key frame commented to show the hex value of the IPA). To make your facial animation system really powerful, you can create a facial mesh to match each phoneme. As you can probably tell, there's no way you'll ever use more than a few hundred different meshes for your facial animation, so you need to limit the number of IPA Unicode values with which you work.

This is where the ConvLWV utility application really shinesin reassigning the IPA Unicode values you know into smaller, more manageable values you will use in your programs. Looking back at Figure 11.11, you'll notice that the lower section details the use of a conversion database. This database is used to remap IPA Unicode values into values you specify. For instance, instead of using 0x025b to represent the phoneme "eh" (as in the words "ten" and "berry"), you might prefer to use the value 0x0001 (which might just happen to be the index number of the phoneme mesh in your animation engine).

These values are later used as indices into an array of meshes that represent your phonemes. Now instead of hundreds of phoneme meshes, you can work with a small set of meshes that are reused for multiple phonemes. Whenever you remap an IPA Unicode value and convert an .LWV file to an .X file, those IPA Unicode values are converted inside your `PhonemeSequence` object. For instance, after you remap a set of IPA Unicode values, the previous `PSEQ` object might look like this:

```
PhonemeSequence PSEQ {
  5;
  2; 0; 30;, // 0x0074
  3; 30; 200;, // 0x025b
  6; 200; 230;, // 0x0073
  2; 230; 270;, // 0x0074
  0; 270; 468;; // 0x005f
}
```

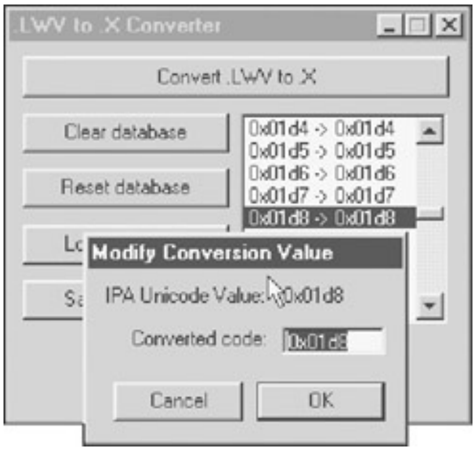
By default, all IPA Unicode values are remapped to the same value0x025b always converts to 0x025b, 0x0075 is always 0x0075, and so on. To change the remapped IPA Unicode value, locate it in the list box in the lower−right corner of the ConvLWV dialog box (as shown in Figure 11.12)



```
0x01d4 -> 0x01d4
0x01d5 -> 0x01d5
0x01d6 -> 0x01d6
0x01d7 -> 0x01d7
0x01d8 -> 0x01d8
0x01d9 -> 0x01d9
0x01da -> 0x01da
0x01db -> 0x01db
0x01dc -> 0x01dc
```

.LWV to .X Converter
Copyright 2002-03 by Jim Adams.
All rights reserved.

Figure 11.12: Double−click on the IPA Unicode value you want to remap in the list box. The number on the left is the Unicode value, and the number on the right is the remapped value.
As an example, suppose you want to remap the "eh" phoneme (IPA Unicode value 0x025b) to a value of 0x0001. In the list box, locate 0x025b (on the left) and double−click it. The Modify Conversion Value dialog box will open, allowing you to enter a new value (see Figure 11.13). Enter 0x0001 and click OK. The list box values will be updated, and you can continue to remap them.



```
0x01d4 -> 0x01d4
0x01d5 -> 0x01d5
0x01d6 -> 0x01d6
0x01d7 -> 0x01d7
0x01d8 -> 0x01d8
```

IPA Unicode Value: 0x01d8
Converted code: 0x01d8

Figure 11.13: The Modify Conversion Value dialog box allows you to remap an IPA Unicode value to another number. Enter the new value to use and click OK.
So all your hard work is not lost, you have the option to save and later reload your remapped values. This is handy when you are working on many different sequences using the same conversion values. To save a conversion database, click the Save Database button and enter a file name. All conversion databases use the .IPA file extension. To load a conversion database, click the Load Database button, locate your .IPA file, and click Load.

The last two buttons in the ConvLWV program are Clear Database and Reset Database. The Clear Database button sets all conversion values to 0x0000, meaning that all IPA Unicode values are remapped to the number 0. Clicking the Reset Database button sets all conversion values to match each IPA Unicode value (so that 0x025b will convert to 0x025b, 0x0075 will remap to 0x0075, and so on).

And that's it for using the ConvLWV program. Go ahead and give it a tryrecord, process, and convert a few .LWV files to .X files. When you're ready, you can move on to loading the phoneme sequences into your program using a custom .X parser.

Using an .X Parser for Sequences

After you've run your .LWV files through the ConvLWV program, you're left with an .X file that contains a couple template definitions and a key−framed phoneme sequence. All you need to do is write an .X parser that loads that sequence into an array of structures that contains the phoneme mesh reference numbers, as well as the beginning and ending times of each key frame. One structure definition should do the trick for storing the key frames.

```
typedef struct {
   DWORD Code;       // Phoneme mesh #
   DWORD StartTime;  // Starting time of morph
   DWORD EndTime;    // Ending time of morph
 } sPhoneme;
```

When you're parsing the .X file containing your phoneme sequence, allocate an array of `sPhoneme` structures to match the number of key frames used in the animation. For example, consider the following `PhonemeSequence` object (which contains a phoneme sequence):

```
PhonemeSequence PSEQ {
  5;
  2; 0; 30;,
  3; 30; 200;,
  6; 200; 230;,
  2; 230; 270;,
  0; 270; 468;;
}
```

In `PSEQ`, there are five key frames in the animation, as determined by the first value in the object. As you are parsing the `PSEQ` object, you must allocate five `sPhoneme` structures to match. Once you allocate the structures, you can iterate the list of key frames.

The first key frame uses phoneme mesh #2 and achieves 100−percent blending in 30 milliseconds (starting at 0 milliseconds and ending at 30). The second key frame uses mesh #3 and achieves 100−percent blending in 170 milliseconds (from 30 milliseconds to 200 milliseconds). This continues throughout the list.

One of the issues I've been dodging up to this point is how the blending actually works in facial animation. In Chapter 8, you saw how to blend multiple meshes. For facial animations, such as making your mesh's eyes blink and brow move, blending the appropriate meshes at various percentages is perfect. The snag comes when you try to morph from one phoneme mesh to another. Morphing from the base mesh to a phoneme mesh is no problem. However, trying to morph from one phoneme mesh to another is impossible. If the base mesh is the source of the morphing operation and the phoneme mesh is the target mesh, how can you blend from one phoneme mesh to another without having to specify a new base mesh?

Of course, you can always specify a new base mesh for the blending operation, but the problem is that all your animations are designed around the initial base mesh (in which your mesh's mouth is closed and its eyes are open). Using a phoneme mesh as the new base mesh is just asking for trouble!

What's the solution? It's actually quite simple. By blending the base mesh with two different phoneme meshes at the same time, you can make it look like your mesh is morphing from one phoneme mesh to another. All you need to do is slowly decrease the source phoneme mesh's blending values from 1 to 0 and slowly increase the target phoneme mesh's blending values from 0 to 1.

I'll get back to this blending solution in a bit. For now, let's get back to your .X parser and take a look at the class declaration.

```
class cXPhonemeParser : public cXParser
{
 public:
  char     *m_Name;   // Name of sequence
  DWORD     m_NumPhonemes; // # phonemes in sequence
  sPhoneme *m_Phonemes;    // Array of phonemes
  DWORD     m_Length; // Length (milliseconds) of sequence

protected:
  BOOL ParseObject(IDirectXFileData *pDataObj,             \
                    IDirectXFileData *pParentDataObj,      \
                    DWORD Depth,                           \
                    void **Data, BOOL Reference);

public:
     cXPhonemeParser();
     ~cXPhonemeParser();

// Free loaded resources
void Free();

// Find the phoneme at specific time
DWORD FindPhoneme(DWORD Time);

// Get mesh #'s and time scale values
void GetAnimData(DWORD Time,                              \
                DWORD *Phoneme1, float *Phoneme1Time,     \
                DWORD *Phoneme2, float *Phoneme2Time);
};
```

I want to take the `cXPhonemeParser` class bit by bit so you can better understand what's occurring. The first thing you'll notice in the class is the variable declarations. Each phoneme animation sequence is contained within a `PhonemeSequence` object. Remember that each object instance can be assigned a name. That is the purpose of the `m_Name` variableto contain the object's instance name.

Note In this case, the `cXPhonemeParser` class only contains the data from a single `PhonemeSequence`, so there's only one buffer to contain the name and other phoneme information. If you're feeling anxious, I recommend expanding the class to contain multiple phoneme sequences.

Following `m_Name` is the number of key frames in the sequence (contained within the .X file), as well as an array of `sPhoneme` structures that contain the phoneme sequence information. Last comes `m_Length`, which is the length of the entire animation sequence in milliseconds. The `m_Length` variable is useful for bounds−checking time values to the length of the animation.

Next in the `cXPhonemeParser` class are the functions, starting with the typical `ParseObject` function you've come to know and love. Remember, the `ParseObject` function is called every time a data object is enumerated from an .X file and you want to load the phoneme sequence data in this function.

Since you're only looking for the `PhonemeSequence` objects, the `ParseObject` function is short and to the point. Starting with the GUID declaration of the `PhonemeSequence` template, you can move on to the actual `ParseTemplate` code to see what's occurring.

```
// Define the PhonemeSequence template GUID
DEFINE_GUID(PhonemeSequence,
            0x918dee50, 0x657c, 0x48b0,
            0x94, 0xa5, 0x15, 0xec, 0x23, 0xe6, 0x3b, 0xc9);

BOOL cXPhonemeParser::ParseObject(                              \
                  IDirectXFileData *pDataObj,                   \
                  IDirectXFileData *pParentDataObj,             \
                  DWORD Depth,                                  \
                  void **Data, BOOL Reference)
{
 const GUID *Type = GetObjectGUID(pDataObj);

  // Only process phoneme sequence objects
  if(*Type == PhonemeSequence) {
   // Free currently loaded sequence
   Free();

   // Get name and pointer to data
   m_Name = GetObjectName(pDataObj);
   DWORD *DataPtr = (DWORD*)GetObjectData(pDataObj, NULL);

   // Get # phonemes, allocate structures, and load data
   m_NumPhonemes = *DataPtr++;
   m_Phonemes = new sPhoneme[m_NumPhonemes];
```

At this point, you've retrieved the instance name of the object you are parsing, and you've pulled out the number of phoneme key frames in the sequence. Next, an array of `sPhoneme` structures is allocated, and processing continues by iterating through each key frame in the sequence, retrieving the phoneme mesh number, starting time, and ending time.

```
for(DWORD i=0;i<m_NumPhonemes;i++) {
      m_Phonemes[i].Code       = *DataPtr++;
      m_Phonemes[i].StartTime  = *DataPtr++;
      m_Phonemes[i].EndTime    = *DataPtr++;
   }
   m_Length = m_Phonemes[m_NumPhonemes−1].EndTime + 1;
 }

 // Parse child objects
 return ParseChildTemplates(pDataObj, Depth, Data, Reference);
}
```

As the `ParseObject` function wraps up, you can see that the length of the animation sequence is saved and any child objects are enumerated. Remember, only the first phoneme sequence object is parsed; any object after the first will be loaded, but earlier sequences will be erased. Again, you might want to expand on that by loading more than one sequence for your own work.

The `Free` function comes next in the `cXPhonemeParser` class. This function frees the class's data, such as the phoneme key−frame array. I'll skip the code for the `Free` function and move on to the next function, `FindPhoneme`, which is responsible for finding the phoneme mesh number inside the key frames for a specified time.

```
DWORD cXPhonemeParser::FindPhoneme(DWORD Time)
{
  if(m_NumPhonemes) {
     // Search for time
     for(DWORD i=0;i<m_NumPhonemes;i++) {
       if(Time >= m_Phonemes[i].StartTime &&                 \
                                Time <= m_Phonemes[i].EndTime)
          return i;

     }
   }
   return 0;
}
```

The `FindPhoneme` function simply iterates the array of phoneme key frames, looking for the one in which the specified time falls. The `FindPhoneme` function is usually called by the `GetAnimData` function, which you use to get the two phoneme meshes that need to be blended, as well as the timescale to blend each of the two meshes (in a range from 0 to 1).

```
void cXPhonemeParser::GetAnimData(                       \
             DWORD Time,                                 \
             DWORD *Phoneme1, float *Phoneme1Time,       \
             DWORD *Phoneme2, float *Phoneme2Time)
{
     // Quick check if past end of animation
     if(Time >= m_Length) {
       *Phoneme1 = m_Phonemes[m_NumPhonemes−1].Code;
       *Phoneme2 = 0;
       *Phoneme1Time = 1.0f;
       *Phoneme2Time = 0.0f;
       return;
}
```

As you can see from the last bit of code, you must insert a special case to test whether the animation is past the end of its length. If so, only the last key frame is used, with the first mesh being blended at 100 percent and the second mesh at 0 percent.

Moving on, the `GetAnimData` function iterates the list of phonemes and looks for one that matches the time specified.

```
  // Find the key to use in the phoneme sequence
  DWORD Index1 = FindPhoneme(Time);
  DWORD Index2 = Index1+1;
  if(Index2 >= m_NumPhonemes)
     Index2 = Index1;

 // Set phoneme index #'s
 *Phoneme1 = m_Phonemes[Index1].Code;
 *Phoneme2 = m_Phonemes[Index2].Code;
```

The phoneme found at the specified time is used as the first blended mesh. The second mesh to be blended is pulled from the following phoneme structure. Again, if the animation sequence is at its end, only the last key−frame values are used.

Here's where you get back to using three meshes to morph from one phoneme mesh to another. As the sequence animates from the first phoneme mesh to the second, the first mesh slowly decreases its blending value from 1 to 0. The second phoneme mesh starts at a blending value of 0 and slowly moves to 1.

To calculate the blending factors for the two meshes, you need to finish the `GetAnimData` function with a few calculations that scale the current time between the two used key frames' times and store the blending values as a factor of that scaled time.

```
  // Calculate timing values
  DWORD Time1 = m_Phonemes[Index1].StartTime;
  DWORD Time2 = m_Phonemes[Index1].EndTime;
  DWORD TimeDiff = Time2 − Time1;
  Time −= Time1;
  float TimeFactor = 1.0f / (float)TimeDiff;
  float Timing = (float)Time * TimeFactor;

  // Set phoneme times
  *Phoneme1Time = 1.0f − Timing;
  *Phoneme2Time = Timing;
}
```

And that's it for the `cXPhonemeParser` class! To use the class, you only need to instance it and call `Parse` on the .X file that contains your phoneme sequence data.

```
cXPhonemeParser PhonemeParser;
PhonemeParser.Parse("Phoneme.x");
```

For each frame of animation, call the `GetAnimData` function to determine which meshes and blending values to use for drawing your facial mesh. You need to calculate the animation timing by calling a function such as `timeGetTime` and bounding the result by the length of the animation, as I've done here:

```
  // Get animation time
  DWORD Time = (DWORD)timeGetTime();
  Time %= PhonemeParser.m_Length;

  // Get mesh numbers and blending values
  DWORD Mesh1, Mesh2;
  float Time1, Time2;
  PhonemeParser.GetAnimData(Time,&Mesh1,&Time1,&Mesh2,&Time2);
```

I think you already know what's coming next. Using the values you retrieved from the `GetAnimData` function call, you can now draw your blended mesh. Consult Chapter 8 for details on drawing blended meshes, or check out the facial animation demo included on the CD−ROM for a working example. Check the end of this chapter for details on the facial animation demo.

Congratulations! You've successfully loaded and played back a phoneme animation sequence! What's next on the list? You can blend more meshes to make your facial animations more believable, by having the eyes blink or adding the ability to change the facial expressions. Now that you've learned the basics, it's all easy to do.

Playing Facial Sequences with Sound

Now that you've recorded and played your facial animation sequences, it's time to go that extra step and begin synchronizing those animations to sound. If you're using facial animation to lip−sync to sound files, this is as easy as playing the sound file and running your animation at the same time.

With so many ways to load and play back your sound files, what's a DirectX programmer like you supposed to do? To cut through the confusing array of available media libraries, I'd like to turn to the best one to dateMicrosoft's DirectShow! DirectShow is really a powerful media−authoring system in disguise. Lucky for us, it's quite easy to use.

If all you need is to play media files (audio media files, in this case), then you're in luck because that is one of the simplest functions to perform using DirectShow. In this section, I'll give you the whirlwind tour of making DirectShow play back an audio media file, whether that file is a .WAV, .MP3, .WMA, or any other sound file that has a codec registered with Windows. Of course, these media files will contain the spoken dialog that you want to synchronize with your facial animation.

For the exact details on DirectShow and the objects you'll use here, consult Chapter 14, "Using Animated Textures." Like I said, this is the whirlwind tour, so things are going to happen fast!

Using DirectShow for Sound

DirectShow is a collection of interfaces and objects that works with video and audio media. I'm talking about recording and playing media from just about any source, including live video, streaming Web content, DVD, and pre−recorded files. As if that isn't enough, DirectShow even allows you to create your own media decoders and encoders, making it the only media system of choice.

To add DirectShow to your project, you must first include dshow.h in your source code.

```
#include "dshow.h"
```

Also, make sure to add the strmiids.lib file to your project's link files. The strmiids.lib file is located in the same directory as your other DirectX libraries (commonly \dxsdk\lib). After you've included and linked the proper files, you can instance the following four DirectShow interfaces to use in your code:

```
IGraphBuilder  *pGraph    = NULL;
IMediaControl  *pControl  = NULL;
IMediaEvent    *pEvent    = NULL;
IMediaPosition *pPosition = NULL;
```

The first interface, `IGraphBuilder`, is the head honcho here. It deals with loading and decoding your media files. The second interface, `IMediaControl`, controls playback of the audio file. The third interface, `IMediaEvent`, retrieves events, such as playback completion. The last interface, `IMediaPosition`, sets and retrieves the position in which playback occurs. (For instance, you can set the audio to play five seconds into the sound, or you can tell that playback is currently at 20 seconds into the sound.)

Note Since you're using the COM system, you need to call `CoInitialize` inside your program's initialization code. Once that is complete, you must call `CoUninitialize` to make the application shut down the COM system.

Use the `CoCreateInstance` function to create your `IGraphBuilder` object (from which the remaining three interfaces are queried), as I've done here:

```
// Initialize the COM system
CoInitialize(NULL);

// Create the IGraphBuilder object
CoCreateInstance(CLSID_FilterGraph, NULL,                    \
                    CLSCTX_INPROC_SERVER, IID_IGraphBuilder, \
                    (void**)&pGraph);
```

After you've created the `IGraphBuilder` object, you can call `IGraphBuilding::RenderFile` to begin using the object immediately to load the audio media file you want. (This is called *rendering.*) As the file is rendered, DirectShow loads all necessary codecs for decoding the media data.

The `RenderFile` function takes the file name of the media file you want to play as a wide−character string, which you can construct using the `L` macro. For example, to load a file named MeTalking.mp3, you would use the following code bit. (Note that the second parameter of `RenderFile` is always NULL.)

```
pGraph−>RenderFile(L"MeTalking.mp3", NULL);
```

After you've loaded the media file, you can query the remaining three interfaces from the `IGraphBuilder` object, as I've done here:

```
pGraph−>QueryInterface(IID_IMediaControl, (void**)&pControl);
pGraph−>QueryInterface(IID_IMediaEvent,   (void**)&pEvent);
pGraph−>QueryInterface(IID_IMediaPosition,(void**)&pPosition);
```

You're almost there! Getting your sound to begin playing is as simple as calling `IMediaControl::Run.`

```
pControl−>Run();
```

There you have it. If everything went as planned, you should hear your sound being played back! Now you just need to synchronize your facial animation with the sound that's being played.

Synchronizing Animation with Sound

After your sound is loaded and playing, it's time to synchronize the animation to the sound. Because your animation runs off time values (milliseconds, in this case), you can query DirectShow for the exact time of the playing sound. Using this time value, you can update your lip−syncing facial animation every frame, making it perfectly synchronized to your playing sound.

To obtain the sound's play time, you use the `IMediaPosition` interface you created in the previous section. The specific function you're looking for is `IMediaPosition::get_CurrentPosition`, which takes a pointer to a `REFTIME` (a float data type) variable as the only parameter, as you can see here:

```
REFTIME SndTime; // REFTIME = float
pPosition−>get_CurrentPosition(&SndTime);
```

To obtain the current play time in milliseconds, you simply multiply the `REFTIME` value you receive from `get_CurrentPosition` by 1000.0 and cast the resulting number into a `DWORD` variable. That `DWORD` variable will contain the time you'll use to update your lip−syncing facial animation. Following is an example bit of code that converts the time received from `get_CurrentPosition` to milliseconds.

Note Automated features, such as the blinking of your mesh's eyes, should run off the internal timer, not the DirectSound playback time. To obtain the internal timer's value (in milliseconds) you can use the `timeGetTime` function, which returns a `DWORD` value that contains the number of milliseconds that have passed since Windows started.

```
DWORD MillisecondTime = (DWORD)(SndTime * 1000.0f);
```

You use this time value `(MillisecondTime)` to locate the correct sequence of phoneme facial meshes between which to morph the animation data you loaded with your .X parser. Take a moment to check out this chapter's demo program to see just how to use these timing values in your animation. (See the end of this chapter for more details about the demo program.)

If you jumped ahead and tried playing the sound and animation perfectly synchronized, you might have noticed that something is wrong. In the real world, a person moves his mouth before you hear a sound. In its current state, your synchronization method doesn't compensate for this fact. Not to worry, howeveryou only need to offset the animation timing a tiny bit to correct this problem. Simply subtract a small amount of time from the animation. I suggest a value of 30 to 80 milliseconds, but you might want to play with the value a bit to get the animation synchronized just right.

Looping Sound Playback

Something seems to be missing. Hmm, what could it be? It's `IMediaEvent,` the third extra interface you queried from the `IGraphBuilder` object! The `IMediaEvent` interface shows its true colors whenever you need to know whether your sound has completed playback. (That's all you're concerned with at this point.) This is useful when you want to determine whether the sound should restart, or whether some other event needs to occur when playback is complete.

Events are represented by a command code and two parameters, all cast to long data types.

```
long Code, Param1, Param2;
```

To retrieve the event code and parameters, you call `IMediaEvent::GetEvent,` as shown here:

```
HRESULT hr = pEvent->GetEvent(&Code, &Param1, &Param2, 1);
```

It's important to record the return value from calling `IMediaEvent::GetEvent.` A value of `S_OK` means that an event was retrieved, and any other error value means that there are no further events to process. Because any number of events can be waiting for you to process, you should continuously call `GetEvent` until no more events are waiting. You can see that process here, contained within a `while` loop.

Note The fourth parameter of `IMediaEvent::GetEvent` is how many milliseconds you want to wait for an event to be retrieved. For the purposes of this book, I always wait for one millisecond before continuing.

```
while(SUCCESS(pEvent->GetEvent(&Code,&Param1,&Param2,1))) {
  // Process playback completion event
  if(Code == EC_COMPLETE) {
    // Do something, as playback is done
  }

 // Free event data
 pEvent->FreeEventParams(Code, Param1, Param2);
 }
```

From the comments in the preceding code, you can see that I've taken the liberty of checking the type of event code that was retrieved from the call to `GetEvent.` The one and only event you're looking for is playback completion, which is represented by the `EC_COMPLETE` macro. Inside the conditional block, you can do whatever your little heart desires. For example, you can seek back to the beginning of the sound and start playback, as I've done here:

```
// Process playback complete event
if(Code == EC_COMPLETE) {
  // Seek to beginning of sound and replay it
  pPosition−>put_CurrentPosition(0.0f);
  pControl−>Run();
}
```

One other thing you might have noticed is the call to `IMediaEvent::FreeEventParams.` You must always call `FreeEventParams` to let DirectShow free any resources that were allocated for the event you retrieved from a prior call to `GetEvent`.

With that, your sound playback functions are complete! Well, almost complete. When you're finished with the sound, it's customary to call `IMediaControl::Stop` to stop any sound from playing and to free all COM interfaces using their respective `Release` functions.

Check Out the Demo

Whew! Facial animation is incredibly easy to use when you have the know−how, and the effects are definitely worth it. On the book's CD−ROM, you'll find two programs of interest when it comes to facial animationFacialAnim and ConvLWV.

The first program, FacialAnim, is the only project that comes with source code. The FacialAnim demo, shown in Figure 11.14, shows off the power of the facial animation engine developed in this chapter.

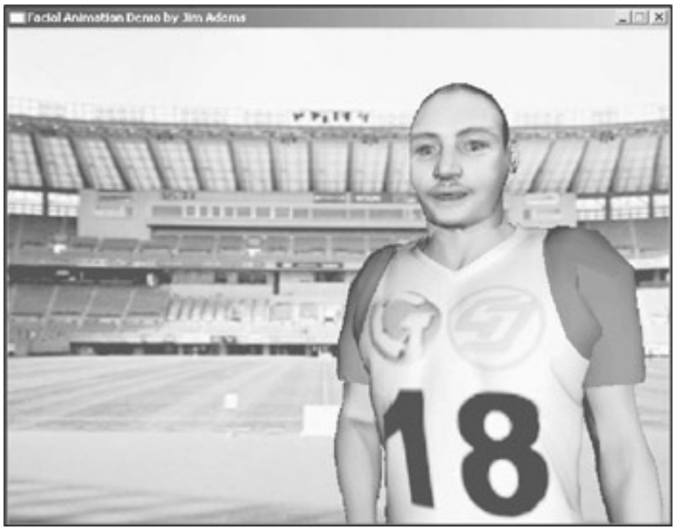


Figure 11.14: Get a hands−on soccer report from a fully lip−synced game character in the FacialAnim demo!
The final program, ConvLWV, helps you create your own phoneme sequences for use with the facial animation package developed in this chapter. Check out this chapter's text for more information about using ConvLWV.

When you think you've got this whole facial animation system down, I suggest you improve on the techniques covered here by giving your face mesh new features, such as teeth, hair, and eyebrows. Add animation to your mesh's eyes or try increasing your set of phonemes. When you've got your facial meshes just right, start playing with texturing. Texturing plays a major role in your systema neatly textured face model beats a plain one any day.

Programs on the CD

Check Out the Demo

The Chapter 11 directory on the CD−ROM includes a project (the FacialAnim demo) and two programs (Microsoft's Agent software package and the ConvLWV program). Specifically, these include

- **FacialAnim.** Check out facial animation with this demo, which shows a mesh talking and changing its reactions in real time. It is located at \BookCode\Chap11\FacialAnim.
- **Agent.** Microsoft's Agent software package includes the Linguistic Information Sound Editing Tool. It is located at \BookCode\Chap11\Agent.
- **ConvLWV.** This program converts .LWV files to .X files and saves phoneme sequences in objects easily imported into your game projects. For licensing reasons, the source code for this project is not available. The program is located at \BookCode\Chap11\ConvLWV.

Part Five: Miscellaneous Animation

Chapter 12: Using Particles in Animation

Aflashy explosion, a clump of grass, a towering tree, a puff of smoke, a screaming pedestrian, and a sketchy little fishwhat could they possibly have in common? The fact that you can draw them in your game using particles, that's what! Particles are the number one special−effect doodads of gaming, capable of displaying everything from droplets of rain to glowing bursts of explosive shrapnel. With a little know−how and the help of this chapter, you can turn these simple little particles into huge game−enhancing visuals!

Working with Particles

First things firstwhat are particles? A *particle* is a simple graphical object used as a visual enhancement. Particles are typically used to draw small graphical effects such as fire, smoke, and those sparkly little lights that come from a magic missile trail (see Figure 12.1). Your game could certainly operate without particles, but it's the particles that make the visuals so vivid.

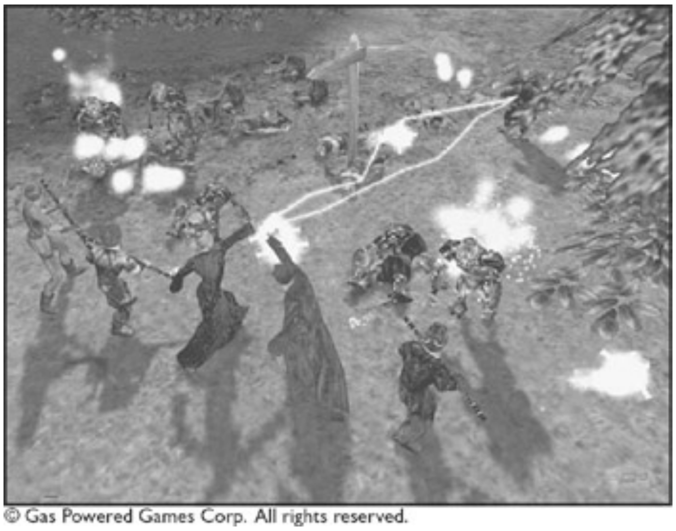


Figure 12.1: A blast from a spell creates a shower of particles in Gas Powered Games' Dungeon Siege.

In fact, particles are good for more than just those sparkly little lights and puffs of smoke. A couple games use particles in manners you might not suspect. For instance, Namco's *Tekken* series uses particles to represent flowing tufts of grass in specific levels.

What are some other advanced uses for particles? Imagine thisyour newest game consists of gargantuan monsters that wreak havoc on the various cities of the nation. In one level of your game, your monster of choice can smash a local dam. After a few direct hits, the dam cracks and water begins to pour out, bouncing off the scaly back of the titanic creature and pouring down into the city. Where are the particles in this scenario?

I'll bet you said the water, right? You are correct, but there are also the chunks flying off the dam, terrified citizens fleeing the city's buildings, exploding transformers from flooded power lines, and maybe even the occasional bullet being shot at the monster by the braver citizens. That's not even mentioning the portions of your level that use billboards to render the city's various signs, trees, and cars, which are also particles.

So there you have ityou can throw a bunch of simple particles into your game, enhancing the visual appearance with a minimum of effort. The cool thing is, you can handle most particles with one or two small class objects that you can plug into any of your game projects.

Before you can go as far as using classes to control particles, however, you'll have to start with the extreme basics. First you need to see how to draw particles, and then you can move on from there.

Starting with the Basics

Particles are typically quad−shaped polygons that are rendered using a small set of textures. Using texture−mapped quad polygons, you can fool people into thinking the particle is actually a 3D mesh; the texture map gives it the appearance of such. It makes sense because if you are going to show only one side of a mesh, you can render it out to a texture map and apply it to a quad polygon instead.

Because a particle is drawn using only two triangles (forming a quad), as shown in Figure 12.2, you must rely on billboard rendering to ensure that the polygons are always facing the viewer.

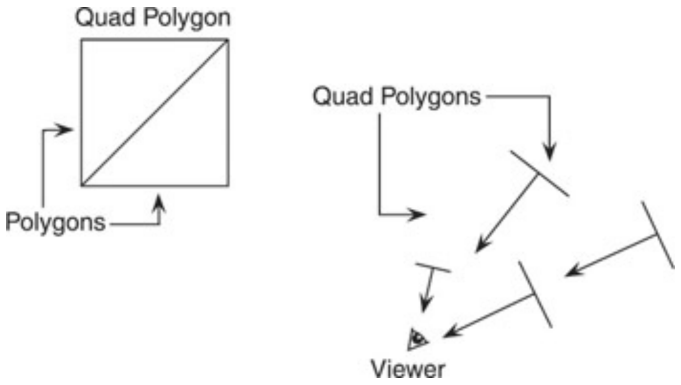


Figure 12.2: Two triangles are sandwiched together to form a quad polygon. Looking down from above, you can see that billboarding ensures that the polygons are always facing the viewer.
In case you haven't heard the term before, *billboarding* is the technique of orienting an object (such as a polygon) so that it is always facing the viewer. Initially, a billboarded object is created so that it points at the negative z−axis (as shown in Figure 12.3). As the viewer moves, the billboarded object rotates so that it always faces the viewer.

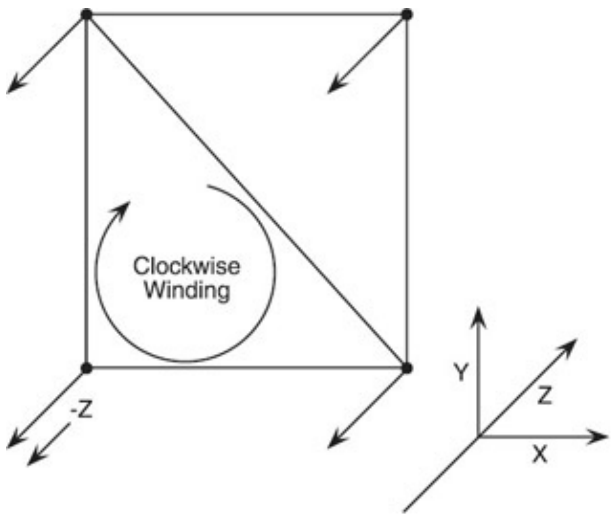


Figure 12.3: The billboarded quad polygon on the left initially points at the negative z−axis, only to be rotated when drawn so that it faces the viewer.
The reasons for using billboards are quite simpleto save memory and speed up rendering. To billboard, you would typically take a bitmap image of what you want the particle to look like (such as a puff of smoke for a smoke particle), and you would draw this image onto the particle's polygons. Then, instead of having to render a 3D mesh that represents the smoke, you could render the particle polygons using the smoke bitmap

image as a texture.

This is not to say that rendering 3D meshes for particles is a bad thing. In fact, a particle can be composed of any type of primitive, from pixels and lines to polygons and entire 3D meshes. You can bet that there will be times when a simple quad polygon won't cut it and you'll need to use 3D meshes instead.

Going back to your monster game, you could use particles to represent the tiny vehicles driving your city's streets. A simple textured (and billboarded) quad polygon could suffice for those vehicles, but what if you had a simple 3D mesh to use instead? That only requires a few additional polygons to render each frameand believe me, the effects of having all those little cars driving around your virtual city would be well worth the extra rendering time.

Okay, rendering with meshes aside, the majority of particles you'll be drawing are from billboarded quad polygons. You can draw these polygons in a number of ways, but I will show you three in this book. The first method of drawing particles is quite possibly the easiest, but you'll soon find out that it has its drawbacks.

Drawing Particles with Quad Polygons

Drawing particles is as simple as drawing polygons because a particle is merely a variable−sized texture−mapped quad polygon. Figure 12.4 illustrates a typical layout of the two triangular polygons you'll use to create a particle. In this figure, I have shown a particle that is 10 units in size.

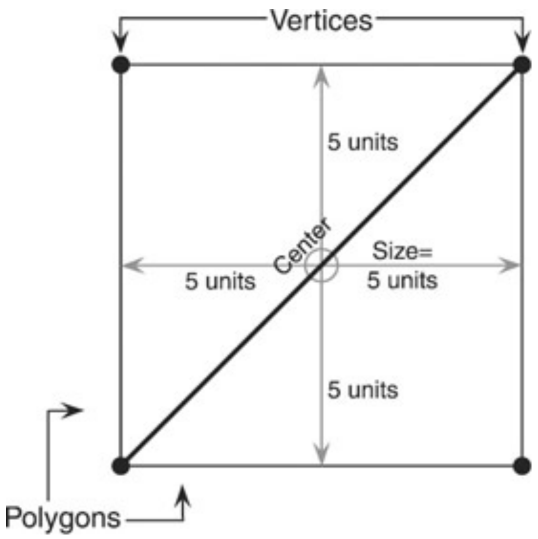


Figure 12.4: A particle that is 10 units in size extends 5 units from the origin in both the x and y axes.
In addition to the coordinates of the vertices you use to draw the particles, you need a couple other things. First, there's the texture you'll use to enhance the particle's appearance. Whether that texture is smoke, a blob of light, or a terrified person, you need to store the image in a texture object. Along with the texture, your polygons also need texture coordinate information to map the texture image to the polygons.

To keep things simple, I will use one texture per particle type for all the particles in this chapter. In other words, if I have a smoke particle and a fire particle, I'll load two textures. Any instance of either particle will use its respective texture to render. You might want to compile the particle textures into one texture surface to improve your particle.

The next bit of information you need to create your particle's polygon data is the diffuse color component. The ability to change the colors of your particles during run time is very useful because the changing colors can represent the various life cycles of your particles. For instance, fire slowly cools and changes color as it moves away from the heat source. Instead of using a multitude of texture images to represent the various levels of heat, you can use the same texture image and slowly modify the diffuse color of the particle.

The last bit of information you need is the particle's size. Particles can pretty much be any size, from a tiny speck of dust to an Earth−crushing meteor. To define the size of a particle, you just choose the dimensions you want the particle to be using the same world−space coordinates as your 3D meshes.

For example, suppose you want a particle to be 20 units wide by 50 units high. This size extends in the x and y axes only (since a particle is really a flat object that is rotated to always face the viewer). When it is being created, the center of the particle is placed at the origin of the world. Using the particle's size as a guide, you then create some vertices to represent the corners of the particle. These vertices are placed using half of the particle's dimensions as an offset. For example, your 20×50 particle extends from 10 to 10 along the x−axis, and from 25 to 25 along the y−axis.

That's about ityou should now have enough information to start drawing your particles! To recap, you should have the particle's size and vertex coordinates (using the size of the particle as a reference), as well as the texture coordinates and diffuse−color component for each vertex. Go ahead and put those components into a vertex structure, as I've done here:

```
typedef struct {
     D3DXVECTOR3 vecPos; // Particle vertex coordinates
     D3DCOLOR Diffuse;   // Diffuse color
     float u, v;         // Texture coordinates
} sVertex;
```

Don't forget to define your FVF declaration for your newly created vertex structure.

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

Using the vertex structure and FVF, you can create a small vertex buffer that contains enough vertices to draw a single particle. You're using two triangular polygons, which means you need to create six vertices (three per polygon). Using a triangle strip, you can reduce the number of vertices to four.

You can create the vertex using the following bit of code:

```
IDirect3DVertexBuffer9 *pVB = NULL;
pDevice->CreateVertexBuffer(4*sizeof(sVertex),           \
                            D3DUSAGE_WRITEONLY,          \
                            VERTEXFVF, D3DPOOL_DEFAULT,  \
                            &pVB, NULL);
```

Now that you have created the vertex buffer (which you only do once in your program), you can stuff your particle data in it. Suppose you want to draw a particle that is 10 units in size (for both the x and y axes), uses the entire texture surface, and has a white color component. To do so, you would use the following code:

```
// Size = size of particle, 10.0 in this case
float Size = 10.0;

// Get half the size for setting vertex coordinates
float HalfSize = Size / 2.0f;

// Lock the vertex buffer and fill with vertex data
sVertex *Vertex = NULL;
```

```
pVB->Lock(0, 0, (void**)&;Vertex, 0);

// Top-left corner, vertex #0
pVB[0].vecPos = D3DXVECTOR3(-Size, Size, 0.0f);
pVB[0].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
pVB[0].u = 0.0f; pVB[0].v = 0.0f;

// Top-right corner, vertex #1
pVB[1].vecPos = D3DXVECTOR3(Size, Size, 0.0f);
pVB[1].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
pVB[1].u = 1.0f; pVB[1].v = 0.0f;

// Bottom-left corner, vertex #2
pVB[2].vecPos = D3DXVECTOR3(-Size, -Size, 0.0f);
pVB[2].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
pVB[2].u = 0.0f; pVB[2].v = 1.0f;

// Bottom-right corner, vertex #3
pVB[3].vecPos = D3DXVECTOR3(Size, -Size, 0.0f);
pVB[3].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
pVB[3].u = 1.0f; pVB[3].v = 1.0f;

// Unlock vertex buffer
pVB->Unlock();
```

At this point, your vertex buffer is ready to render. However, there is one little catch. You'll notice that the vertex coordinates place the polygons at the origin of your 3D world, extending in the x and y axes. Since the viewpoint can be anywhere in the world, you must position the polygons using the world transformation prior to rendering.

You also have to rotate the polygons to face the view, which you'll recall is the purpose of billboarding. You need to calculate a billboard transformation to rotate the particle polygons to face the viewer. With that transformation, you add the coordinates of the particle where it should be drawn (in world space coordinates).

Caution Using the `GetTransform` function to retrieve a transformation from Direct3D is extremely slow and sometimes not allowed. It's best to maintain global transformations for your world, view, and projection transformations so you can use those instead. For now, however, I'll go the bad route and use `GetTransform` for demonstration purposes.

To create a billboard transformation, you need to grab the view transformation matrix and calculate its inversed transformation (thus reversing the order of transformation contained in the transformation) using the `D3DXMatrixInverse` function, as shown here:

```
// Grab the view transformation matrix and inverse it
D3DXMATRIX matView;
pDevice->GetTransform(D3DTS_VIEW, &matView);
D3DXMatrixInverse(&matView, NULL, &matView);
```

The purpose of using the inversed transformation is that it will rotate the particle's vertices in the opposite direction that the view is facing, thus aligning the coordinates to the viewer. After you get the inversed view transformation, you need to add the particle's coordinates directly in order to position the particle in the 3D world. You can do this by storing the x, y, and z coordinates in the just−calculated inversed transformation matrix's _41, _42, and _43 variables and setting the resulting transformation matrix as the world transformation

```
// Assuming ParticleXPos, ParticleYPos, and ParticleZPos
// contain the particle's coordinates in world space where
// you want it drawn.

// Add in coordinates of particle to draw
matView._41 = ParticleXPos;
matView._42 = ParticleYPos;
matView._43 = ParticleZPos;

// Set resulting matrix as the world transformation
pDevice->SetTransform(D3DTS_WORLD, &matView);
```

Now that you have set your world transformation matrix, you can render the polygons. To make sure your particles blend perfectly with your scene, you should ensure that you are using z−buffering and that alpha testing is enabled. Using z−buffering ensures that the particles are drawn into the scene properly, and alpha testing ensures that transparent portions of the particle's texture map are not drawn (leaving your geometry to show through those transparent parts). Also, you can enable alpha blending if you want to create some cool color−blending effects.

Skipping the z−buffer setup (which you should have done during setup), you can enable alpha testing and alpha blending as follows:

```
// Turn on alpha testing
pDevice->SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
pDevice->SetRenderState(D3DRS_ALPHAREF, 0x08);
pDevice->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);

// Turn on alpha blending (simple additive type)
pDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);
pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_DESTCOLOR);
```

For the alpha testing, you'll notice that I chose to use the greater than or equal to comparative rule. This means that pixels in your particle's texture map that have an alpha value of 8 or greater are rendered, while the others are skipped. Using alpha testing means you need to use a function such as `D3DXCreateTextureFromFileEx` to load your textures, specifying a color mode that uses alpha channels (such as `D3DFMT_A8R8G8B8`) and a color key of opaque black (`D3DCOLOR_RGBA(0,0,0,255)`), such as in the following code bit:

```
D3DXCreateTextureFromFileEx(
                 pDevice,
                 "Particle.bmp",
                 D3DX_DEFAULT, D3DX_DEFAULT, D3DX_DEFAULT,
                 0, D3DFMT_A8R8G8B8, D3DPOOL_DEFAULT,
                 D3DX_DEFAULT, D3DX_DEFAULT,
                 D3DCOLOR_RGBA(0,0,0,255), NULL, NULL,
                 &pTexture);
```

Now that you have the appropriate alpha render states set up, you can set the FVF, streams, and texture, and then render away!

```
// Set vertex shader and stream source
pDevice->SetVertexShader(NULL);
pDevice->SetFVF(PARTICLEFVF);
pDevice->SetStreamSource(0, pBuffer, 0, sizeof(sVertex));

// Set the texture
```

```
pDevice−>SetTexture(0, &pTexture);

// Draw the particle
pDevice−>DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
```

All right, now you're getting somewhere! You have drawn a single particle! Because you'll most likely render more than one particle at a time, you won't have to repeat all this code for each particle. Once you've set up the stream and texture, you can repeatedly lock the vertex buffer, fill in the data, unlock the buffer, and render the particle. If the size of your particles doesn't change, you can avoid locking and unlocking the vertex buffer each time and just skip to storing the particle's coordinates in the inverse transformation matrix, setting the appropriate texture, and rendering away.

Note The Particles demo on the CD−ROM demonstrates drawing huge lists of particles using the techniques you just learned. Check out the end of this chapter for information on the Particles demo.

After you've played around with the particle−rendering code, you'll notice a couple of things. First, there's a lot of vertex data to process. Second, the constant locking, filling, and unlocking of the vertex buffer can really knock down your performance. You need something to render particles that uses less memory and is more optimized. What you need are Point Sprites!

Working with Point Sprites

As you can tell, particles are game staplesmany games use them for various special effects. Microsoft was well aware of this fact, and they added the ability to render billboarded quad polygons in Direct3D using Point Sprites. A *Point Sprite* is merely a billboarded quad polygon represented by a set of minimal data. Each particle is represented by a single 3D coordinate (representing the center of the particle) and the size of the particle (as measured in the previous section, except you use only one size measure that extends equally in both the x and y axes). This provides you with an enormous memory savings over using quad polygons.

Remember from earlier in this chapter that a quad−polygon particle requires four vertices. For the previous `sVertex` structure, you're stuck using twenty floats and four DWORDs per particle. That's a total of 96 bytes of data to draw just one particle! So how do Point Sprites compare?

A Point Sprite uses the following vertex structure:

```
typedef struct {
   D3DXVECTOR3 vecPos;   // Center coordinates of particle
   float       Size;     // Size of particle
   D3DCOLOR    Diffuse;  // Diffuse color
} sPointSprite;
```

It looks somewhat the same, doesn't it? The big difference is that you only need one `sPointSprite` per particle. That's rightthe `sPointSprite` structure uses only 20 bytes of data, beating the `sVertex` particles by 76 bytes. What a savings!

So what's the catch to using Point Sprites? You knew there had to be some kind of downfall, didn't you? The bad thing about Point Sprites is that they are limited in size. Whereas you can create particles of any size using quad polygons, a Point Sprite is limited to the maximum size set in the `D3DCAPS9::MaxPointSize` variable. That means the maximum size of a Point Sprite particle is dependent on the end user's video hardware.

Note

To ensure that the hardware can render Point Sprites, make sure to check the driver's hardware capabilities using the `IDirect3D9::GetDeviceCaps` function. If the `D3DCAPS9::MaxPointSize` value obtained from that function is set to 1.0f, then the hardware doesn't support Point Sprites.

Another downfall is that the video card drivers must be able to handle Point Sprites. Frequently, I have encountered bad drivers that make the Point Sprites flicker or use the incorrect size to render. You can just hope that most end users will have updated drivers that guarantee accurate use of Point Sprites!

Downfalls aside, Point Sprites save you a tremendous amount of memory; for those video cards that can handle them, they are great for drawing particles. As you saw in the `sPointSprite` vertex structure, Point Sprites only use four floats and a single DWORD to store the particle coordinates, size, and diffuse color, respectively.

Note Point Sprites use the entire texture surface to render onto the particle, meaning that you need one texture per particle image you are using. This also means you don't have to specify texture coordinates in your Point Sprite vertex structure.

Point Sprites use the following FVF declaration:

```
#define POINTSPRITEFVF (D3DFVF_XYZ|D3DFVF_PSIZE|D3DFVF_DIFFUSE)
```

You need to specify the `D3DUSAGE_POINTS` flag during your call to `CreateVertexBuffer`, as in the following code bit:

```
pDevice->CreateVertexBuffer(8 * sizeof(sPointSprite),      \
                     D3DUSAGE_POINTS | D3DUSAGE_WRITEONLY, \
                     POINTSPRITEFVF, D3DPOOL_DEFAULT,      \
                     &pBuffer, 0);
```

After you've created your vertex buffer (remembering to specify the number of vertices you want to contain in the vertex buffer), you can begin filling the buffer with the particles you want to render. For example, suppose there are eight particles you want to render, each of which is 10 units in size (extending 5 units in both the x and y axes) and use a white diffuse color. With the vertex buffer you just created, you can lock, fill, and unlock the vertex buffer using the following code:

```
float Size = 10.0f; // Make particles 10 units in size

sPointSprite PointSprites[8] = {
     // Particle #0
     {  D3DXVECTOR3(0.0f,0.0f,0.0f), Size,
        D3DCOLOR_RGBA(255,255,255,255) },
     // Particle #1
     {  D3DXVECTOR3(10.0f,0.0f,0.0f), Size,
        D3DCOLOR_RGBA(255,255,255,255) },
     // Particle #2
     {  D3DXVECTOR3(20.0f,0.0f,0.0f), Size,
        D3DCOLOR_RGBA(255,255,255,255) },
     // Particle #3
     {  D3DXVECTOR3(30.0f,0.0f,0.0f), Size,
        D3DCOLOR_RGBA(255,255,255,255) },
     // Particle #4
     {  D3DXVECTOR3(-10.0f,0.0f,0.0f), Size,
        D3DCOLOR_RGBA(255,255,255,255) },
     // Particle #5
```

```
    {  D3DXVECTOR3(-20.0f,0.0f,0.0f), Size,
       D3DCOLOR_RGBA(255,255,255,255) },
    // Particle #6
    {  D3DXVECTOR3(-30.0f,0.0f,0.0f), Size,
       D3DCOLOR_RGBA(255,255,255,255) },
    // Particle #7
    {  D3DXVECTOR3(-40.0f,0.0f,0.0f), Size,
       D3DCOLOR_RGBA(255,255,255,255) },
  };

  // Lock the vertex buffer
  sPointSprite *Ptr;
  pBuffer->Lock(0,0,(void**)&Ptr,0);

  // Copy vertex data into buffer
  memcpy(Ptr, PointSprites, sizeof(PointSprites));

  // Unlock vertex buffer
  pBuffer->Unlock();
```

Now that you've created and filled in the vertex buffer with the Point Sprite data, you can render away. Wait! I forgot to have you set some important render states. Direct3D needs to know a few things, namely that you want to use textured Point Sprites that are positioned in 3D space (as opposed to screen space). To specify that you want to use Point Sprites that are positioned in 3D space, you must set the appropriate render states, as shown here:

```
// Use entire texture for rendering point sprites
pDevice->SetRenderState(D3DRS_POINTSPRITEENABLE, TRUE);

// Scale in camera space
pDevice->SetRenderState(D3DRS_POINTSCALEENABLE, TRUE);
```

Also, you need to let Direct3D know the minimum size of a Point Sprite, and how big to make it if your vertex declaration is missing the size. For now, I'll tell Direct3D that Point Sprites are to use a size of 1 if the vertex declaration is missing the data, and to use a minimum size of 0.

```
// Set default and minimum size of point sprites
pDevice->SetRenderState(D3DRS_POINTSIZE,     FLOAT2DWORD(1.0f));
pDevice->SetRenderState(D3DRS_POINTSIZE_MIN, FLOAT2DWORD(0.0f));
```

Finally, you need to set a few distance−based attenuation scaling factors. These tell Direct3D how to size the particles depending on their distance from the viewer. We're not getting fancy here, so the default values (as specified by the DirectX SDK documents) will do. These factors (which are actually render states) are set using the following code:

```
// Define a function to convert from a float to a DWORD
inline DWORD FLOAT2DWORD(FLOAT f) { return *((DWORD*)&f); }

// Set attenuation values for scaling
pDevice->SetRenderState(D3DRS_POINTSCALE_A, FLOAT2DWORD(1.0f));
pDevice->SetRenderState(D3DRS_POINTSCALE_B, FLOAT2DWORD(0.0f));
pDevice->SetRenderState(D3DRS_POINTSCALE_C, FLOAT2DWORD(0.0f));
```

You'll notice something funny about that last code bitthe inclusion of the `FLOAT2DWORD` function. As you know, the `SetRenderState` function only accepts DWORD values for the second parameter. The attenuation factors are floating−point values, so you need a way to convert from those floating−point values to a DWORD value. That's where FLOAT2DWORD comes in. Using FLOAT2DWORD, you can pass any floating−point value to the SetRenderState function, and rest assured the value will be converted into an acceptable DWORD value.

Finally, you are able to render the Point Sprites! Remember, Point Sprites are merely vertex buffers that use the Point Sprite primitive type, signified by the D3DPT_POINTLIST flag in your call to DrawPrimitive. Without further delay, here's the call to enable alpha testing and blending and set the FVF and stream source, followed by the call to DrawPrimitive.

```
// Turn on alpha testing
pDevice−>SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
pDevice−>SetRenderState(D3DRS_ALPHAREF, 0x08);
pDevice−>SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);

// Turn on alpha blending (simple additive type)
pDevice−>SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
pDevice−>SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);
pDevice−>SetRenderState(D3DRS_DESTBLEND, D3DBLEND_DESTCOLOR);

// Set vertex shader and stream source
pDevice−>SetVertexShader(NULL);
pDevice−>SetFVF(POINTSPRITEFVF);
pDevice−>SetStreamSource(0, pBuffer, 0, sizeof(sPointSprite));

// Render the Point Sprites (8 of 'em)
pDevice−>DrawPrimitive(D3DPT_POINTLIST, 0, 8);
```

As you can see, working with Point Sprites is very easy. It's certainly much easier than dealing with billboarded quad polygonsthere's much less data involved. The only problem is that Point Sprites are limited in size and not fully supported on all video cards. It would be really nice if you had the ability to use large particles *and* the speed of rendering Point Sprites, wouldn't it? Great newsyou can have both using vertex shaders!

Improving Particle Rendering with Vertex Shaders

What could I possibly show you that would improve your particle−rendering abilities? I might as well tell it to you straightusing vertex shaders, you can mix the ease of using quad polygons with the speed of rendering Point Sprites.

Previously, in the "Drawing Particles with Quad Polygons" section, I showed you how to draw particles one at a time by setting the coordinates of four vertices to create a quad polygon, and then using an inversed view transformation matrix mixed with the particle's world coordinates to render the polygons. Each particle was rendered one at a time, meaning that you had to lock, fill, and unlock the vertex buffer each time.

Even if the particles didn't change in size, meaning you didn't have to lock and unlock the vertex buffer for each particle drawn, you still had to modify and set the transformation matrix each time to make sure the particles were in their proper locations in the 3D world before rendering. Think of ita thousand particles means a thousand SetTransform function calls!

Because vertex shaders work in line with the rendering pipeline, there's no need to bother with the transformations every time you want to render a single particle. That's rightno more of this drawing−one−particle−at−a−time business! Using vertex shaders, you can fill the vertex buffer with as many vertices as you can and draw a whole slew of particles with one call. That means your vertex shader particles will match the speed of Point Sprites!

Using vertex shaders, the vertex structure is much like the structure of Point Sprites. Figure 12.5 and the following vertex structure show the center coordinates of the particle in 3D world space, as well as the diffuse color and texture coordinates to track. The only difference between the following vertex structure and the Point Sprite vertex structure is that the size of the particle is stored a little differently.

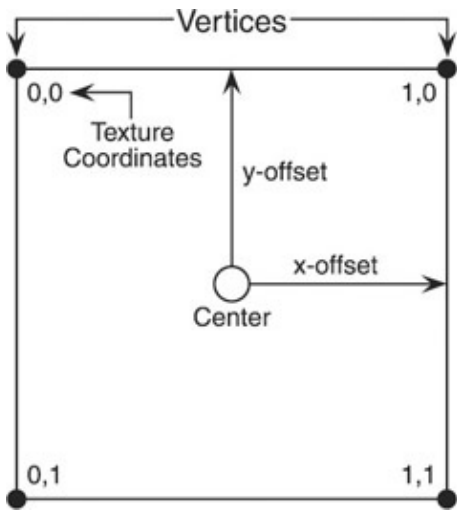


Figure 12.5: The vertex shader particle is composed of four vertices that are defined by a central point, the offset from the center, the diffuse color, and texture coordinates.

```
typedef struct {
     D3DXVECTOR3  vecPos;      // Coordinates of particle
     D3DXVECTOR2  vecOffset;   // Vertex coordinate offsets
     DWORD        Diffuse;     // Diffuse color of particle
     float        u, v;        // Texture coordinates
} sShaderVertex;
```

As Figure 12.5 shows, each vertex is measured by a distance away from the center of the particle, called an *offset*. The offsets are stored in a `D3DXVECTOR2` vector object, with each vector component relating to its respective axis (`vecOffset.x` for the x−axis and `vecOffset.y` for the y−axis).

The offset vector values perform much like the standard polygon and Point Sprite size variables; they determine the size of the particle in each axis. For example, an offset value of 10 in the x offset means the particle extends from 10 to 10 in the x−axis (giving you a particle width of 20 units). The same is true for the y−axis; the offset value can be any number that determines the size of the particle in the y−axis. Typically, you set these offset values to the same amount to ensure a square particle.

To create a particle using the vertex structure just declared, you create a vertex buffer that contains enough vertices for each particle you want to draw. Using triangle lists, this means each particle uses six vertices; if you use an index buffer as well, you can knock the number of vertices down to four.

I'll get back to the index buffer in a moment. For now, I want to talk more about the vertex buffer. Because you're using a vertex shader to render the particles, you must create a vertex element declaration to create the vertex buffer and map the vertex structure components to the vertex shader registers. For DirectX9 users, this means instancing an array of `D3DVERTEXELEMENT9` structures, as shown in the following bit of code:

```
D3DVERTEXELEMENT9 ParticleDecl[ ] =
{
     // Vertex coordinates − D3DXVECTOR3 vecPos
     { 0, 0, D3DDECLTYPE_FLOAT3,    D3DDECLMETHOD_DEFAULT,      \
                                    D3DDECLUSAGE_POSITION, 0 },
     // Vertex corner offset − D3DXVECTOR2 vecOffset
     { 0, 12, D3DDECLTYPE_FLOAT2,   D3DDECLMETHOD_DEFAULT,      \
                                    D3DDECLUSAGE_POSITION, 1 },
     // Diffuse color − DWORD Diffuse
     { 0, 20, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,      \
                                    D3DDECLUSAGE_COLOR,    0 },
     // Texture coordinates − float u, v
     { 0, 24, D3DDECLTYPE_FLOAT2,   D3DDECLMETHOD_DEFAULT,      \
                                    D3DDECLUSAGE_TEXCOORD, 0 },
     D3DDECL_END()
};
```

The `ParticleDecl` vertex element declaration is pretty straightforwardit maps the vertex structure components to their respective counterparts in the vertex shader. First, there are two position components (the 3D coordinates and the corner−offset vector). These are followed by the diffuse color and texture coordinates. Each component uses index #0 except for the corner−offset vector, which uses index #1 of the position usage type.

The vertex elements will make more sense when you see the vertex shader. For now, just create the vertex buffer and fill it with the particle data. Suppose you want room for four particles, which works out to sixteen vertices (four vertices per particle, with each vertex defining a corner of the particle). The following code bit will create the vertex buffer using the previously declared vertex elements:

```
IDirect3DVertexBuffer9 *pVB = NULL;
pDevice−>CreateVertexBuffer(16 * sizeof(sShaderVertex), \
                 D3DUSAGE_WRITEONLY, 0, \
                 D3DPOOL_DEFAULT, &pVB, 0);
```

As I mentioned earlier, you also need to create an index buffer to render the particles. Because each particle uses two polygons (with three vertices each), you need six indices per particle. In this case you have four particles, so you need to create an index buffer that can hold 24 indices.

```
IDirect3DIndexBuffer9 *pIB = NULL;
pDevice−>CreateIndexBuffer(24 * sizeof(short), \
                 D3DUSAGE_WRITEONLY, D3DFMT_INDEX16, \
                 D3DPOOL_DEFAULT, &pIB, 0);
unsigned short *IBPtr;
pIB−>Lock(0, 0, (void**)&IBPtr, 0);
for(DWORD i=0;i<4;i++) { // # particles
   IBPtr[i*6+0] = i * 4 + 0;
   IBPtr[i*6+1] = i * 4 + 1;
   IBPtr[i*6+2] = i * 4 + 2;
   IBPtr[i*6+3] = i * 4 + 3;
   IBPtr[i*6+4] = i * 4 + 2;
   IBPtr[i*6+5] = i * 4 + 1;
}
pIB−>Unlock();
```

All that's left to do at this point is lock your vertex buffer, fill in the data, unlock, and render away! When you are filling the vertex buffer, make sure to set the correct offset vector values for each corner of the particle being drawn, and set the center coordinates of the particle in each vertex. The following code demonstrates setting up four particles to draw, each at a random position and size:

```
// Lock the vertex buffer
```

```
sShaderVertex *VPtr = NULL;
pVB−>Lock(0, 0, (void**)VPtr, 0);

for(DWORD i=0;i<4;i++) {
   // Get a random position of the particle
   float x = (float)(rand()%20)−10.0f;
   float y = (float)(rand()%20)−10.0f;
   float z = (float)(rand()%20)−10.0f;

   // Get a random size of the particle to use
   float Size = (float)(rand()%10)+1.0f;

   // Get half the size of the particle for setting data
   float HalfSize = Size / 2.0f;

   // Top−left vertex
   pVB[0].vecPos = D3DXVECTOR3(x, y, z);
   pVB[0].vecOffset = D3DXVECTOR2(−HalfSize, HalfSize);
   pVB[0].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
   pVB[0].u = 0.0f; pVB[0].v = 0.0f;

   // Top−right vertex
   pVB[1].vecPos = D3DXVECTOR3(x, y, z);
   pVB[1].vecOffset = D3DXVECTOR2(HalfSize, HalfSize);
   pVB[1].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
   pVB[1].u = 1.0f; pVB[0].v = 0.0f;

   // Bottom−left vertex
   pVB[2].vecPos = D3DXVECTOR3(x, y, z);
   pVB[2].vecOffset = D3DXVECTOR2(−HalfSize, −HalfSize);
   pVB[2].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
   pVB[2].u = 0.0f; pVB[0].v = 1.0f;

   // Bottom−right vertex
   pVB[3].vecPos = D3DXVECTOR3(x, y, z);
   pVB[3].vecOffset = D3DXVECTOR2(HalfSize, −HalfSize);
   pVB[3].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
   pVB[3].u = 1.0f; pVB[0].v = 1.0f;

   // Go to next four vertices
   pVB+=4;
}
```

Now that you have set the vertex buffer, you can render the particles. Well, that's not entirely trueyou still have to create and load the vertex shader; create the declaration interface; set the vertex sources, texture, alpha testing and blending; and set the vertex shader constants.

Loading a vertex shader and creating the declaration interface is standard in DirectX9, so I'll skip that. (If you need help, that code is in the ParticlesVS demo for this chapter, or check out the helper functions in Chapter 1.) You've also seen how to set vertex sources, textures, and alpha testing and blending states, so I'll skip that too. Now I want to show you the actual vertex shader you'll use to render the particles.

Remember that each particle is composed of four vertices. Previously in this chapter, you needed to position each of those vertices using an inversed view transformation matrix before rendering the particle. Things are going to be a little different with your vertex shader, however. Before going on, I'd like to pause and introduce you to the view transformation's directional vector components.

Aside from being a tongue twister, the directional vector components describe the direction your view is pointingforward, up, and to the right (as illustrated in Figure 12.6).

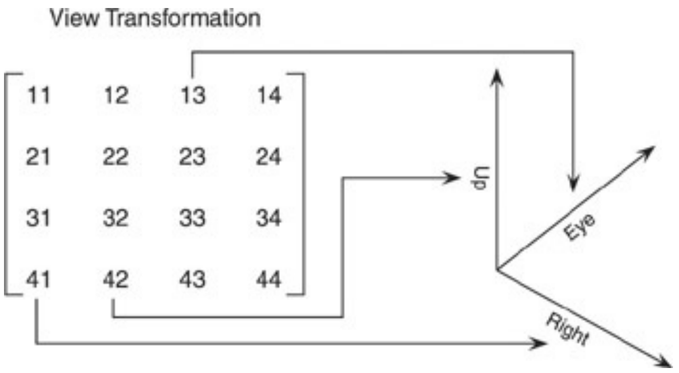
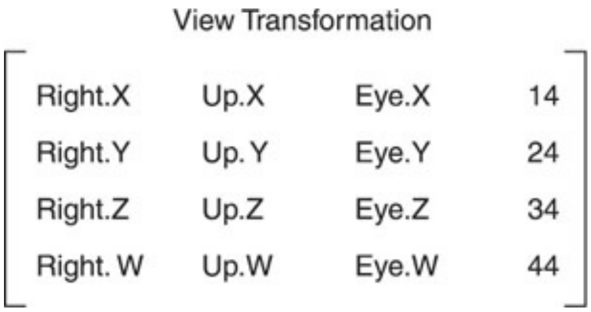


Figure 12.6: The view transformation tells you which direction the view is facing, as well as which way is up and which is right from its orientation.

To better understand the directional components, stand straight up and look forward. The direction you are looking is called the *eye vector*it describes the direction you are facing. In your mind, draw a line from your heels to the top of your head. The direction of this line is called the *up vector*it points up from your current orientation. Finally, raise your right arm at a 90−degree angle to your body and point your finger in the direction your arm is pointing. The direction you are pointing is the *right vector*it always points to the right of your current orientation.

These vectors (eye, up, and right) tell you which directions are relative to your current orientation. For example, if you walk forward, then the eye vector is the direction you are moving; if you walk backward, then the inverse eye vector is the direction you are moving. Likewise, if you moved to your right, you would be moving in the direction of the right vector. Move left and you are going in the inverse direction of the right vector. The same goes for the up vector if you move up or down along that same imaginary line from your feet to your head.

Now, you know all this vector component stuff is leading somewhere, don't you? Well, your view transformation actually contains the three directional vectors I just mentioned. Each component is stored in a column of the view transformation. As shown in Figure 12.7, the right vector is column 1, the up vector is column 2, the eye vector is column 3, and column 4 is left alone.



View Transformation

$$\begin{bmatrix} \text{Right.X} & \text{Up.X} & \text{Eye.X} & 14 \\ \text{Right.Y} & \text{Up.Y} & \text{Eye.Y} & 24 \\ \text{Right.Z} & \text{Up.Z} & \text{Eye.Z} & 34 \\ \text{Right.W} & \text{Up.W} & \text{Eye.W} & 44 \end{bmatrix}$$

Figure 12.7: Each column in the view transformation contains a directional vector you can use to position your particle's vertices.

Now I want to get back to the point. Using these directional vectors (or actually the normalized directional vectors), you can position the particle's corner vertices by first moving them in the appropriate directionup, down, left, or right, as determined by the up and right directional vectors. These vectors are scaled by the offset coordinates of the particle.

In plain English, you first want to place each vertex at the origin of the world. From there, move the vertex left or right using the normalized right directional vector scaled by the x offset (`sShaderParticle::vecOffset.x`) of the particle. Now move the vertex up or down using the normalized up directional vector scaled by the y offset (`sShaderParticle::vecOffset.y`) of the particle. Finally, add the particle's center coordinates to come up with the vertex's final coordinates. Rinse, lather, and repeat for each vertex. Now your vertices are in their proper world−space positions, and you can render away!

I'll show you how to grab those directional vector components from the view transformation in a moment; for now, I want to get back to the vertex shader. To start the shader, I list in comments its mapping and constants used, along with the version requirement:

```
; v0 = Particle coordinates
; v1 = X/Y offsets to position vertex
; v2 = Diffuse color of particle
; v3 = Texture coordinates
;
; c0−c3 = view*projection matrix
; c4 = normalized right directional vector
; c5 = normalized up directional vector
vs.1.1
```

As you can see, the constants you will use are the combined view and projection matrices, which you place in constants `c0` through `c3`. Next come the normalized right and up directional vectors I mentioned earlier. You put these in constants `c4` and `c5`. I'll get back to setting up the constants later; for now, I want to get back to the shader code.

Next comes the actual vertex register mapping declarations, which ensure that the proper components from the vertex structure can be accessed in the vertex shader.

```
dcl_position  v0
dcl_position1 v1
dcl_color     v2
dcl_texcoord  v3
```

And now the fun begins! Remember earlier in this section, when I said you need to first position the vertex at the world's origin and move it along the scaled right and up vectors? Well, that's the purpose of the following code, which comes next in the vertex shader.

```
; Scale the corner's offsets by the right and up vectors
mov r2, v1
mad r1, r2.xxx, c4, v0
mad r1, r2.yyy, c5, r1
```

The previous bit of code takes the normalized scaling vectors (placed in the constant registers `c4` and `c5`) and multiplies them by the offset values in the vertex structure. The results are then added to the particle's center coordinates to come up with the final vertex coordinates. All you have to do now is apply your view*projection transformation to the vertex's coordinates and store the diffuse color and texture coordinates.

```
; Apply view * proj transformation
m4x4 oPos, r1, c0

; Store diffuse color
mov oD0, v2
```

```
; Store texture coordinates
mov oT0.xy, v3
```

Wow, that's one small and efficient shader! All that's left to do after you create and load your new vertex shader is set the vertex shader constants and draw away! These constants include your view*projection transformation and the right and up directional vectors. Let's get the transformation bit out of the way and move on to the directional vectors.

I'm going to assume you have the view and projection transformation matrices stored in two different `D3DXMATRIX` objects. All you have to do is multiply these two, transpose the resulting matrix, and store it in the constants via the `SetVertexShaderConstantF` function.

```
// matView = view transformation matrix
// matProj = projection transformation matrix
D3DXMATRIX matViewProj = matView * matProj;
D3DXMatrixTranspose(&matViewProj, &matViewProj);
pDevice->SetVertexShaderConstantF(0, (float*)matViewProj, 4);
```

Now come the directional vectors. Remember that I said the directional vectors are stored in the columns of the view transformation? Since you already have your view transformation matrix object, you can pull out the components directly and normalize them at the same time with the following code. (Notice you're only dealing with the right and up components, leaving the eye vector alone.)

```
// Get normalized right/up vectors from view transformation
D3DXVECTOR4 vecRight, vecUp;

// Right vector is 1st columnn
D3DXVec4Normalize(&vecRight, \
            &D3DXVECTOR4(matView._11, \
               matView._21, \
               matView._31, 0.0f));

// Up vector is 2nd column
D3DXVec4Normalize(&vecUp, \
               &D3DXVECTOR4(matView._12, \
                   matView._22, \
                   matView._32, 0.0f));
```

Once you have the normalized vectors, you can store them in the constants `c4` and `c5`.

```
pDevice->SetVertexShaderConstantF(4, (float*)&vecRight, 1);
pDevice->SetVertexShaderConstantF(5, (float*)&vecUp, 1);
```

At last! It's all ready, and you can render your particles using your cool vertex shader! Make sure to set the streams, texture, shader, and declaration, and be sure to use the indexed primitive drawing methods because you're using index buffers. Check it out here in this bit of code:

```
// pShader = vertex shader interface
// pDec = vertex element declaration interface

// Set vertex shader, declaration, and stream sources
pDevice->SetFVF(NULL);
pDevice->SetVertexShader(pShader);
pDevice->SetVertexDeclaration(pDecl);
```

```
pDevice->SetStreamSource(0, pVB, 0, sizeof(sShaderVertex));
pDevice->SetIndices(pIB);

// Turn on alpha testing
pDevice->SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
pDevice->SetRenderState(D3DRS_ALPHAREF, 0x08);
pDevice->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);

// Turn on alpha blending (simple additive type)
pDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);
pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_DESTCOLOR);

// Render the vertex shader particles (4 of 'em)
pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,0,0,16,0,8);
```

As you can see, drawing particles using a vertex shader is the best way to go. When you are comfortable with the way the vertex shader works with the particle data, check out the demo for this chapter, which demonstrates how to handle large amounts of particles in your own projects.

When you're ready to get past the basics of drawing particles, you can move on and see how to bring them to life in your own game projects.

Bringing Your Particles to Life

Now that you know how to draw a particle, you can put that knowledge to work and learn how to create, control, and destroy many instances of your particles in your 3D world. The first thing you can do is create a particle class that will contain the various bits of information about your particle.

Aside from the vertex data that is kept separate from the particle data, you might want to keep tabs on the particle's position, color, and type. You can also store the direction and speed in which a particle moves, as well as how long the particle has to live before your engine removes it.

For this book, I decided to use the following bits of information for each particle:

- **Type.** Particles can be many different types, and this data defines the type of particle. For example, I can store fire and smoke particles together using the same particle structure.
- **Position.** This set of 3D coordinates determines where in the world a particle is located.
- **Velocity.** Speed and directional data are stored using a vector. The x, y, and z components of the vector define how fast a particle moves in that axis.
- **Life.** Particles can exist in your 3D world for only so long; this piece of data lets your engine know when to remove the particle from the list of active particles to render. However, you don't have to remove particles because you can make them live indefinitely if you want.
- **Size.** A particle can change size during its lifetime. This data contains the value you want to use for the particle's size (in 3D units). Remember, a particle extends equally in both the x and y axes, so specifying a size of 10 will make the particle 20 units wide by 20 units high (−10 to 10 units in both axes).
- **Color.** To simulate changes in a particle, you use a color modifier to change the red, green, and blue color components of the polygons during rendering. For example, a fire particle can change color from white to red to yellow over time.

You can pack all of this data into a simple class, as follows.

```
class cParticle
{
  public:
    DWORD       m_Type;        // Type of particle

    D3DXVECTOR3 m_vecPos;      // Position of particle
    D3DXVECTOR3 m_vecVelocity; // Velocity of particle
    DWORD       m_Life;        // Life of particle in ms
    float       m_Size;        // Size of particle
    DWORD       m_Color;       // Diffuse color of particle
};
```

Each particle requires its own `cParticle` class instance. You can compile a collection of these classes into an array to make dealing with multiple particles a breeze. Or, as I'll show you later, you can maintain a linked list of these particle containers. I'll teach you how to deal with more than one particle later in this chapter. For now, I want to keep things simple and deal with one particle at a time.

The first step is to fill in the `cParticle` class with the information about the particleits type, where you want it placed, its color, and so on. The following bit of code sets the particle to type 1 and places it at the origin of the world. The size is nominal (set at 5), and the color is set to bright white (the original color of the texture map). Velocity doesn't come into play yet, so just zero out the vector's components.

```
// Instance a particle and fill w/data
cParticle Particle;
Particle.m_Type = 1;
Particle.m_vecPos = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
Particle.m_Size = 5.0f;
Particle.m_Color = D3DCOLOR_RGBA(255,255,255,255);
Particle.m_vecVelocity = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
```

That's all there is to creating a particle! Of course, the particle is just sitting there doing nothing, so let's make that sucker fly around by giving it some velocity.

Moving Particles Using Velocity

Once you've birthed a particle int he oyour 3D world, you need to give it motion, or a method of updating its position in the world. Each particle has a special way of updating its coordinates. You want fire particles to slowly rise while changing color from the hottest to coldest values (red to orange, for example). Smoke particles tend to drift in the wind, so you want to check for a wind source nearby and use that to control those particles.

Most particles are very simple in nature, moving in a specific direction that alters over time based on external forces. Particles have a velocity that tells each one how fast or slow to move. The external forces you use can increase or decrease a particle's velocity over time, thus speeding up or slowing down its movement.

For each frame you process, you can tally the forces to apply to the velocity of the particles. These forces can come from any source, such as wind, gravity, drag, or propulsion. Not all forces are required for your engine, howeverthey just make things a little more realistic. At a minimum, you'll want to incorporate propulsion and gravity into your particle engine so that your particles will be able to move and come to a rest on the ground.

You can add the functions that apply force to the particle class. Or, if you are using another class to manage the particles (like the one you'll soon see), you can place the functions in there. For now, I'll just show you how to calculate a force vector from multiple sources to apply to a particle's velocity (notice that forces are measured per millisecond).

```
// The force to apply to the particle(s)
D3DXVECTOR3 vecForce = D3DXVECTOR3(0.0f, 0.0f, 0.0f);

// Add a per−millisecond gravity force of 0.02
vecForce += D3DXVECTOR3(0.0f, −0.02f, 0.0f);

// Add a per−millisecond wind force of 0.01f
vecForce += D3DXVECTOR3(0.01f, 0.0f, 0.0f);
```

After you've calculated the force to apply to each particle, you need to update the velocity. Updating the particle's velocity is only a matter of adding the force vector to the velocity vector. Remember, forces are measured per millisecond, meaning that you must multiply the force vector by the number of milliseconds that has passed since you last moved the particles. Suppose the amount of time passed is stored in a floating−point variable called `TimeElapsed`.

```
// TimeElapsed = time, in milliseconds, that has passed
// since the particle was last moved.
Particle.m_vecVelocity += (m_vecForce * TimeElapsed);
```

In addition to using a force vector to change your particle's velocity, there are alternative means to update your particles using a more intelligent processing method.

Using Intelligence in Processing

So far I've only discussed simple particle movements using simple forces. What about those fleeing pedestrians I spoke about at the beginning of this chapter? You knowthose terrified people running for their lives from the towering monsters destroying their city's dam. Well nobody said you couldn't use a little intelligence in your particle's processing, did they?

Instead of applying a set force to the particles, you can intelligently move them using minimal time and processing. For example, suppose your game's town is divided into a grid. Each gridline is a street on which cars are allowed to drive. Each car has a specific direction it is traveling; whenever it hits an intersection of the gridlines, it can choose a new, random direction to move.

Various grid intersections are marked as dangerous when the monster attacks the city, and from that point on all cars will try to move away from those points at twice the normal speed. Eventually, all cars will have fled the city, been crushed by the ravaging monster, or become hopelessly trapped and abandoned by the occupants. Therefore, all car particles can be removed over time, thus freeing up processing time for other things.

The intelligent particles know exactly what velocity to travel in what direction. Rather than slowly changing velocity in a specific direction over time, those particles change velocity immediately.

This sort of intelligent processing is specific to your game project, and it is easy to perform using the techniques in this chapter. In the demo programs included on the CD−ROM, you'll see how I incorporated intelligent processing into my particle engine.

In the demo, there's a batch of particles that represent people. These people are just standing around doing nothingthat is, until a helicopter buzzes overhead, making the poor particle people duck for cover!

To learn about some ways you can use intelligent particle processing, take a look at games like Ingoc's *War of the Monsters* for the PlayStation 2. Remember the scenario of terrorized citizens I mentioned earlier? In *War* *of the Monsters*, huge beasts lay waste to small towns as hundreds of tiny little particle people run for their lives. Definitely a cool use of particles in action!

Creating and Destroying Particles

Particles are commonly used to help spice up game visuals. Your game engine can use literally thousands of polygons to display a single effect. For that reason, it is essential that you use only as many particles as you must, and that you periodically destroy old particles that are no longer of use to your visuals.

For example, a blast at a wall from a laser rifle might produce a batch of particles that represent flying debris. That debris quickly flies through the air and dissipates. This is typical of most particles you will usethey exist only for a short time and then disappear. Of course, there are exceptions for particles that live throughout the entire span of your game. Take the billboarded trees and signs that populate your city's landscape, for example. Those never move and are never removed (except perhaps if a monster stomps them to a pulp). The best way to handle those long−living particles is to draw only those that are within a certain distance of the viewer, thus saving the number of particles drawn every frame. For the demos on the CD−ROM, I use particles for the trees and peoplethese particles are never destroyed. Also, I use particles for puffs of smokethese particles are destroyed after a short time.

Maintaining lists of active particles is easy using a linked list. For each active particle, there is a matching structure of sorts in the linked list of particles. Whenever you want to update the particles in your engine, you iterate through the entire linked list and update the particles contained in it. If a particle is no longer useful, its structure is freed and removed from the linked list.

Going back to the rudimentary `cParticle` class you created earlier in this chapter, you can add a couple pointers to maintain a linked list of particles. Also, you can add a constructor and destructor to the class to handle those pointers during creation and destruction of a class instance.

```
class cParticle
{
  public:
    // Previous particle data, such as type, position, etc.

    cParticle *m_Prev; // Prev particle in linked list
    cParticle *m_Next; // Next particle in linked list

  public:
    // Constructor and destructor to clear/release data
    cParticle() { m_Prev = NULL; m_Next = NULL; }
    ~cParticle() { delete m_Next; m_Next=NULL; m_Prev=NULL; }
};
```

Being a linked list, one particle will become the root to which all other particles are linked. Even this root particle can be destroyed, so the root can change at any time. You can set every particle you want to add to the linked list as the root particle, and then link the old root particle as the next in line (using the `m_Prev` and `m_Next` pointers).

```
// pRoot = current root of particle list

// Create a new instance of a particle class to use
cParticle *pParticle = new cParticle();

// Link new particle as root and link to old root
pParticle−>m_Prev = NULL;   // Clear prev linked list pointer
```

```
pParticle->m_Next = pRoot;  // Link new particle to root
pRoot->m_Prev = pParticle;  // Link root to new particle
pRoot = pParticle;          // Reassign root to new particle
```

To remove a specific particle, you can use the linked list pointers to make any linked particles connect to one another and free the resources of the particle you are removing.

```
// pParticle = particle pointer to remove from list
// pRoot = root particle

// Link previous particle to next particle
if(pParticle->m_Prev) {
  pParticle->m_Prev->m_Next = pParticle->m_Next;
} else {
   // If there's no previous particle in the linked list,
   // then that means this particle is the root. You need
   // to assign the next particle as the root then
   pRoot = pParticle->m_Next;
}

// Link next particle to previous particle
if(pParticle->m_Next)
  pParticle->m_Next->m_Prev = pParticle->m_Prev;

// Clear the particle's pointers and release resources
pParticle->m_Prev = pParticle->m_Next = NULL;
delete pParticle;
```

Maintaining a linked list is basic programming stuff, so there's really no need to go into any more detail here. I'm sure there are much more efficient ways to handle lists of particles, but to be perfectly honest (and I'm sure I'll get some e−mails about this!), using a linked list is fast and easy.

I'll leave it up to you to check out the Particles demo source code to see how the linked list is used to store large numbers of particles. For now, read on to see how to take a linked list of particles and render them.

Drawing Your Particles

What more can I say here that you already haven't seen? Well, the way you draw your particles is very important when you're dealing with what could be thousands of them. Because you don't want a bunch of huge vertex buffers eating away at your memory, you should start managing your buffers before you go any further.

Managing your vertex buffers means that you need to lock and fill them each frame. I know I said that you shouldn't do this, but if you do it correctly you can effectively manage large numbers of particles with the vertex bufferswithout the slowdown commonly caused by using the lock, load, and unlock methodology.

To effectively manage your buffers, you need to keep track of the number of particles being drawn and the number of vertices being inserted during the rendering of the particles. For each particle to be drawn, you insert the appropriate vertices into the vertex buffer. If the buffer is full after the insertion, you can render the entire buffer and start anew with the next particle in line.

After all particles have been processed, you check to see whether any particles have not yet been rendered. In other words, you check to see whether there is still vertex data in the buffer. If so, you render the last batch of particles and move on.

For each batch of particles you render, you need to unlock the vertex buffer and render the polygons. After rendering, you need to re−lock the buffer and continue. Suppose you are using the vertex−shader method of rendering particles and you have a linked list of particles pointed to by a `cParticle` object (`pRoot`). The following code demonstrates how to manage your vertex buffers by batch−rendering groups of particles.

```
// pRoot = root cParticle object w/all particles to draw
// matView, matProj = view and projection transformations
// pTexture = texture object to use for rendering particles
// pShader, pDecl = vertex shader and declaration objects
// pVB, pIB = vertex buffer and index buffer objects
// NumParticles = # of particles vertex buffer can hold

// Set vertex shader, declaration, and stream sources
pDevice->SetFVF(NULL);
pDevice->SetVertexShader(pShader);
pDevice->SetVertexDeclaration(pDecl);
pDevice->SetStreamSource(0, pVB, 0, sizeof(sShaderVertex));
pDevice->SetIndices(pIB);

// Turn on alpha testing
pDevice->SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
pDevice->SetRenderState(D3DRS_ALPHAREF, 0x08);
pDevice->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);

// Turn on alpha blending (simple additive type)
pDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);
pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_DESTCOLOR);

// Set the texture
pDevice->SetTexture(0, pTexture);

// Stored transposed view*projection matrix in constants
D3DXMATRIX matViewProj = (*matView) * (*matProj);
D3DXMatrixTranspose(&matViewProj, &matViewProj);
pDevice->SetVertexShaderConstantF(0, (float*)&matViewProj, 4);

// Get normalized right/up vectors from view transformation
D3DXVECTOR4 vecRight, vecUp;

// Right vector is 1st columnn
D3DXVec4Normalize(&vecRight, \
        &D3DXVECTOR4(matView._11, \
                     matView._21, \
                     matView._31, 0.0f));

// Up vector is 2nd column
D3DXVec4Normalize(&vecUp, \
         &D3DXVECTOR4(matView._12, \
                      matView._22, \
                      matView._32, 0.0f));

// Store vectors in constants
pDevice->SetVertexShaderConstantF(4, (float*)&vecRight, 1);
pDevice->SetVertexShaderConstantF(5, (float*)&vecUp, 1);
```

Up to this point, the code is merely setting up your vertex buffer, streams, alpha states, texture, and constant registers (with the transformation and directional vector values). The next bit of code is what's important here.

It will lock the vertex buffer to prepare to add the particles' vertices. From there, a loop will iterate through every particle.

Note Since you're constantly locking, accessing, and unlocking the vertex buffer here, you might want to use the `D3DUSAGE_DYNAMIC` flag in your call to `IDirect3DDevice9::CreateVertexBuffer.` This ensures that Direct3D knows you'll be doing a lot of work with the buffer, and that it will leave the buffer in some easily accessible and efficiently used memory.

```
// Start at first particle in list
cParticle *Particle = pRoot;

// Set a count for flushing vertex buffer
DWORD Num = 0;

// Lock the vertex buffer for use
sShaderVertex *Ptr;
pVB->Lock(0, 0, (void**)&Ptr, D3DLOCK_DISCARD);

// Loop for all particles
while(Particle != NULL) {
```

After the loop has begun, you can copy the currently iterated particle's vertex data into the vertex buffer.

```
  // Copy particle data into vertex buffer
  float HalfSize   = Particle->m_Size / 2.0f;

  Ptr[0].vecPos    = Particle->m_vecPos;
  Ptr[0].vecOffset = D3DXVECTOR2(-HalfSize, HalfSize);
  Ptr[0].Diffuse   = Particle->m_Color;
  Ptr[0].u = 0.0f;
  Ptr[0].v = 0.0f;
  Ptr[1].vecPos    = Particle->m_vecPos;
  Ptr[1].vecOffset = D3DXVECTOR2(HalfSize, HalfSize);
  Ptr[1].Diffuse   = Particle->m_Color;
  Ptr[1].u = 1.0f;
  Ptr[1].v = 0.0f;
  Ptr[2].vecPos    = Particle->m_vecPos;
  Ptr[2].vecOffset = D3DXVECTOR2(-HalfSize, -HalfSize);
  Ptr[2].Diffuse   = Particle->m_Color;
  Ptr[2].u = 0.0f;
  Ptr[2].v = 1.0f;
  Ptr[3].vecPos    = Particle->m_vecPos;
  Ptr[3].vecOffset = D3DXVECTOR2(HalfSize, -HalfSize);
  Ptr[3].Diffuse = Particle->m_Color;
  Ptr[3].u = 1.0f;
  Ptr[3].v = 1.0f;

  Ptr+=4; // Go to next four vertices
```

After the vertices have been copied to the vertex buffer, you can increase the number of particles contained within the buffer. If this number reaches the maximum number of particles a vertex buffer can contain, then the buffer is unlocked, rendered, and relocked, ready to take on a new set of particles.

```
  // Increase vertex count and flush buffer if full
  Num++;
  if(Num >= NumParticles) {

     // Unlock buffer and render polygons
```

```
      pVB->Unlock();
      ppDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, \
                                     0, 0, Num*4, 0, Num*2);

      // Lock vertex buffer again
      pVB->Lock(0, 0, (void**)&Ptr, D3DLOCK_DISCARD);

      // Clear vertex count
      Num=0;
    }

    // Go to next particle
    Particle = Particle->m_Next;
}
```

After you've looped through all the particles, you need to unlock the vertex buffer one last time. If there are vertices still in the buffer, you must render them.

```
// Unlock vertex buffer
pVB->Unlock();

// Render any polygons left
if(Num)
   pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, \
                                 0, 0, Num*4, 0, Num*2);
}
```

And that's itan efficient way to use a small vertex buffer to render an unlimited number of particles! Now let's take everything you've learned up to this point and create a couple classes that will help you control every aspect of your particles.

Controlling Particles with Class

Earlier in this chapter, in the "Bringing Your Particles to Life" section, you saw how to create a class to contain particle data. This class, `cParticle`, is the perfect starting place to build up a couple of classes to help you. Actually, there are at least two classes you'll want to create. The `cParticle` class contains information about an individual particle, such as position, size, and type. There's not much to the `cParticle` class except that you want it to store data and maintain the pointers to a linked list of particles.

The second class handles an entire list of particles and manages the creation and destruction of particles over time. It also allows you to render all the particles contained in this class's linked list. This type of class object is referred to as a *particle emitter* because it is responsible for emitting particles. One such emitter class I created is defined as follows:

```
class cParticleEmitter
{
  protected:
    IDirect3DDevice9       *m_pDevice; // Parent 3-D device

    // Type of emitter
    DWORD                   m_EmitterType;

    // Vertex buffer and index buffer to contain vertices
    IDirect3DVertexBuffer9  *m_VB;
    IDirect3DIndexBuffer9   *m_IB;
    // Max # particles in buffer
    DWORD                   m_NumParticles;

    // Position of emitter (in 3D space)

    D3DXVECTOR3             m_vecPosition;

    // Root object of particle linked list
    cParticle              *m_Particles;

    // Class reference count
    static DWORD m_RefCount;

    static IDirect3DVertexShader9 *m_pShader; // Vertex shader
    static IDirect3DVertexDeclaration9 *m_pDecl; // Vertex decl
    static IDirect3DTexture9 **m_pTextures; // Textures

  public:
    cParticleEmitter();
    ~cParticleEmitter();

    BOOL Create(IDirect3DDevice9 *pDevice,
                D3DXVECTOR3 *vecPosition,
                DWORD EmitterType,
                DWORD NumParticlesPerBuffer = 32);
    void Free();

    void Add(DWORD Type, D3DXVECTOR3 *vecPos, float Size,
             DWORD Color, DWORD Life,
             D3DXVECTOR3 *vecVelocity);

     void ClearAll();
     void Process(DWORD Elapsed);

     // Functions to prepare for particle rendering, wrap
     // up rendering, and to render a batch of particles.
     BOOL Begin(D3DXMATRIX *matView, D3DXMATRIX *matProj);
     void End();
     void Render();
};
```

Wowthat's a lot to take in! Let's take it bit by bit so you can understand what this particle emitter class does for you. To begin, you have a bunch of protected variables. For each particle emitter class there's a 3D device pointer, as well as a vertex buffer and index buffer pair that you use to render the particles.

Because each emitter might have a different purpose (one emitter might emit fire particles while another emits debris particles, for example), there's a particle type variable (`m_EmitterType`). This variable is dependent on the types of particles you use. For the demo included with this book, I use the following types of particle emitters:

```
// Particle emitter types
#define EMITTER_CLOUD 0
#define EMITTER_TREE 1
#define EMITTER_PEOPLE 2
```

Depending on the type of particle emitter you want, you pass the value as defined in the macros to the `cParticleEmitter::Create` function, along with the 3D device object to use and the number of particles the vertex buffer can hold. Also, you need to pass a vector object that determines the location of the particle emitter in your 3D world.

Next in the list of protected variables you have the root particle object (`m_Particles`), which you use to contain the list of particles created by the emitter. As you can see from the `cParticleEmitter` class declaration, there is a single function (`Add`) that lets you add particles to the scene. You simply call `Add`, specifying the type of particle to add.

I base the type of particle on the textures I use. For example, I have three textures for the three types of emitters. One contains a fire particle image, one contains a smoke particle image, and the third one contains a flashy particle image. The type of particle is defined as follows:

```
#define PARTICLE_FIRE 0
#define PARTICLE_SMOKE 1
#define PARTICLE_FLASH 2
```

Getting back to the `Add` function, you specify the location (in world space) of the particle to add to the scene. Each particle has its own lifespan, color, size, and starting velocity, all of which you can set in the call to `Add`. The lifespan is measured in milliseconds, the color is measured by a `D3DCOLOR` value, the size is measured as a floating−point value, and the velocity is measured as a `D3DXVECTOR` object.

Normally you wouldn't use the `Add` function directly to add particles to your worldthat's the job of the emitter's `Update` function (although that shouldn't stop you from using `Add` when necessary). In fact, the `Update` function has two purposesto update all particles in the linked list and to determine whether more particles need to be added to the list.

A few object pointers finish up the protected variables. These pointers are the vertex shader and element declaration, as well as the array of textures you use to render the particles. Notice that each of these objects is static, meaning that all instances of the particle emitter share them, which helps to save memory.

A reference count is maintained in the `m_RefCount` variable to keep track of the three static objects. When an emitter is created (by calling `Create`), the reference count is increased; when an emitter is destroyed (by calling `Free` or by the class deconstructor), the reference count is decreased. The vertex shader and textures are loaded during the first emitter's initialization (inside the `Create` function); when the last emitter is freed (the reference count is 0), all objects are released.

So far I've described everything except for four functions`ClearAll, Begin, End`, and `Render`. The `ClearAll` function clears the list of particles from the emitter, giving you a fresh slate. You call this function whenever you want to force the emitter to remove all particles.

As for `Begin, End`, and `Render`, these functions work in conjunction to render the particles. When you're using vertex−shader−based particles, there's bound to be some render states and other settings that are constant between all particle emitters. You can save some time by setting up that data beforehand and then rendering the particles, and then finishing up by resetting the appropriate rendering states and data. This is the purpose of those three functions.

In `Begin`, which you really only need to use for vertex−shader−based particles, you set the FVF to NULL, set the vertex shader and declaration to use, store the transposed view*projection transformation, and store the right and up directional vectors. Once you've called the `Begin` function (which takes your view and projection transformation matrices as parameters), you can call `Render` to draw all your particles. This goes for all emitters instanced from the same class, since you are using the same textures and vertex shader. Once you're finished rendering particles, just call `End` to clear the vertex shader and declaration.

The particle emitter class is very straightforward in its constructionit merely maintains a list of particles and renders them. For every frame of your game, you call `Update` to let the class decide which particles to add to the list, which to update, and which to remove. This is simple linked−list manipulation, so I'll skip the code here and leave it for you on the CD−ROM.

In fact, you've already seen everything about the particle emitter in this chapter, from the linked list of particles to processing motion and collision to rendering particles. There's not much to review at this point. If you check out the code for this chapter's demo, you'll see how I created a particle emitter class that handles all types of particles, emitter types, and rendering via the three methods you saw at the beginning of this chapter.

Using the Emitter in Your Project

Okay, okay, that's enough of the class code; let's see how to put all this stuff to work for you. To create an emitter, you need to instance the emitter class, as follows:

```
cParticleEmitter Emitter;
```

Now call the emitter's `Create` function to place the emitter in the center of your 3D world. Also, specify which type of emitter to use.

```
Emitter.Create(pDevice,                         \
         &D3DXVECTOR3(0.0f, 0.0f, 0.0f),        \
          EMITTER_CLOUD);
```

Once you've created the emitter, you can start adding particles to the world using the `Add` function, or you can call `Update` to add those particles for you.

```
Emitter.Add(PARTICLE_SMOKE,                     \
         &D3DXVECTOR3(0.0f, 0.0f, 0.0f),        \
          10.0f, 0xFFFFFFFF, 2000,              \
         &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
```

After you've added some particles to the scene, you can update them using the `Update` function, and then render the particles.

```
Emitter.Update(ElapsedTime);
Emitter.Begin(&matView, &matProj);
Emitter.Render();
Emitter.End();
```

This is sort of a half attempt at demonstrating how to use the particle classes, but I recommend checking out the demos on the book's CD−ROM. These demos do a far better job of showing you just what you can do with particles in your own projects.

Creating Particle Engines in Vertex Shaders

Well, you knew somebody had to create it! There does in fact exist a way to create a complete particle engine that runs inside a vertex shader. At the time of this writing, you could find this vertex shader on Nvidia's Web site at http://developer.nvidia.com/view.asp?IO=Particle_System

The way this shader works is that you define a vertex structure that contains a particle's velocity; using this velocity, you develop a means to move the particle based on an elapsed time. To render the particle, you use

Point Sprites, meaning that you can represent each particle using a single vertex (contained in a vertex buffer).

I don't want to reinvent the wheel, but with the information you've read in this chapter, you should be able to understand what is going on in a particle vertex shader such as this. For each vertex the shader receives to draw (as a Point Sprite), the position, color, and size are calculated. This calculation involves an elapsed time variable that is set via a vertex shader constant. You've already seen how to move a particle based on its velocityit's just a matter of multiplying the velocity vector by the amount of time passed. The same goes for the color and sizethose are calculated by multiplying the appropriate values by the amount of time elapsed.

Check Out the Demos

On this book's CD−ROM you'll find three projects that I created to demonstrate how to manage particles. Including adding, removing, and rendering particles of various types, these projects contain code you can plug right into your own projects to achieve some awesome particle effects.

The output of these three demos is the same (see Figure 12.8). A helicopter flies above a forest and a group of people. There are a number of particles used in the demowisps of smoke that are blown off the ground by the overhead helicopter, as well as trees and people. In fact, there are three different types of trees and two types of people. The particle emitter has a couple extra functions that add smoke particles and change the way the people particles appear during the demo. Whenever the helicopter flies over a person, the particle representing that person changes to a different type. When the helicopter flies away from a person, the particle changes to another particle. Check out the demo to see exactly what I mean.

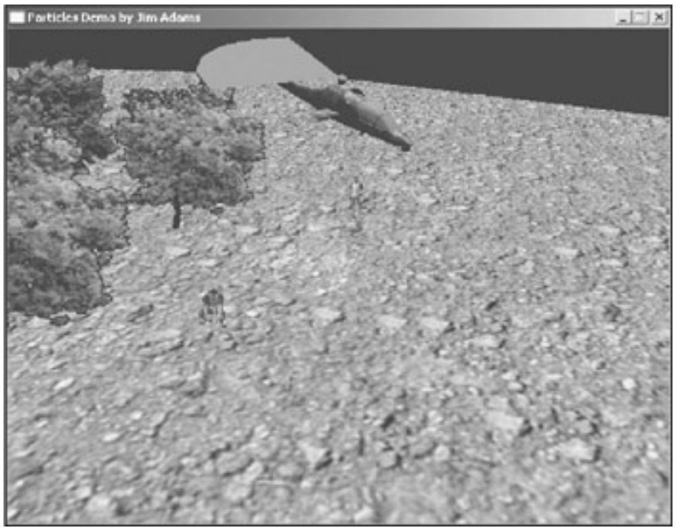


Figure 12.8: An Apache buzzes the heads of some tree−loving bystanders.
As you can see, particle usage is really a project−to−project ordeal. What goes for one project might not work in the next. For that reason, there's not much to go on except the basic theory and knowledge I showed you in this chapter. For a little guidance, make sure to check out the demos on the CD−ROM.

Programs on the CD

In the Chapter 12 directory of this book's CD−ROM, you'll find three projects that demonstrate the three methods of drawing particles, as shown in this chapter. These projects are

- ♦ **Particles.** This project demonstrates how to draw particles one at a time using the quad−polygon−based−method shown first in this chapter. It is located at \BookCode\Chap12\Particles.

- **ParticlesPS.** This project shows you how to use Point Sprites to draw your particles. It is located at \BookCode\Chap12\ParticlesPS.
- **ParticlesVS.** This final project shows you how to use vertex shaders to render your particles. It is located at \BookCode\Chap12\ParticlesVS.

Chapter 13: Simulating Cloth and Soft Body Mesh Animation

Animation in your game need not be so rigid and preshaped. The characters that are walking around in your game−you know the ones I'm talking about−need to appear more realistic. Their clothes are so bland and stiff, their bodies so hard and impenetrable. How is a programmer to deal with these problems? How can you fix them?

I'll tell you what you need−cloth and soft body mesh animation! Give your characters realistic clothing that flows off their body and flaps in the wind, or a lush head of hair that bobs as they are running after fresh prey. Let those characters take blow after blow from an enemy, only to have their bodies bend inward and bounce back out. That's right, realistic cloth and soft body meshes are yours for the taking. It's all here in this chapter!

Simulating Cloth in Your Projects

In its very essence, a cloth is merely a 3D mesh composed of vertices, indices, and polygon faces. While they are understandably important for rendering, those vertices do double−duty when it comes to cloth simulation because they are affected by its physics. A cloth's vertices are therefore referred to as *cloth points* (or *points*, for short). By manipulating the coordinates of the cloth's points over time (with forces such as gravity, wind, or anything else you can throw at it), you can create the appearance of a flowing, elegant piece of material.

To closely emulate a real piece of cloth, these cloth points have a specific mass value associated with them that determines how the point moves according to how much external force is applied. Points with a higher mass require more force to accelerate them, whereas points with a lower mass tend to accelerate more due to external forces being applied. This is a factor of momentum, which states that the force required to move an object is equal to the mass times the acceleration you wish to achieve (F=ma).

Points don't have to move, however; they can be "pinned" in place, as if they are stuck to a part of your game level or attached to a mesh. For the purposes of this chapter, I consider these points to have no mass. (In reality, these points can be thought to have infinite mass, because the amount of force required to move them would by infinitely too high.)

The cloth mesh's polygon edges also have an important role−they hold together the cloth's points. Well, the polygon edges really don't hold the points together−rather, as Figure 13.1 illustrates, those edges represent a bunch of tiny little springs that hold together the points. As the points in your cloth move, each spring tries to maintain a state of static equilibrium by pushing and pulling the points until the forces are balanced. Visually, this makes it look like the points are being kept at a certain distance from one another.

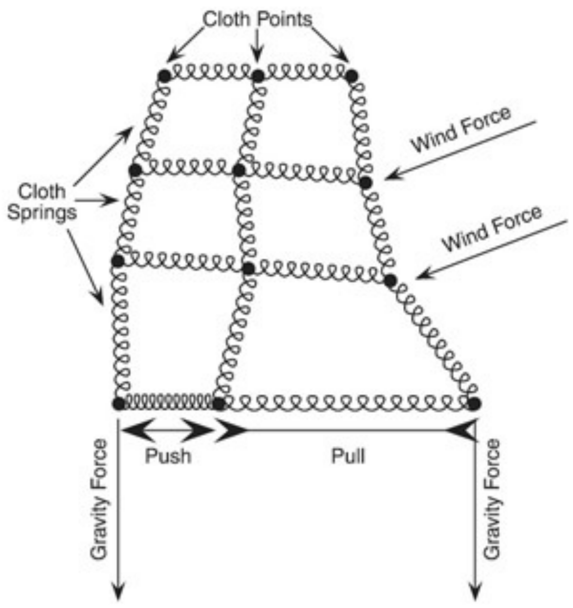


Figure 13.1: *As external forces are applied to the cloth's points, the springs push and pull those points back into shape, thus maintaining the overall shape of the cloth's mesh.*

The distance these springs try to maintain is the initial length between the points at the beginning of the simulation. Each spring has its own way of "snapping" back into position using variable spring stiffness and damping values. The *stiffness* allows you to define how much force a spring exerts in an attempt to return to its original length, whereas the *damping* reduces the amount of force those springs exert to smooth out the motion of points due to spring forces.

It's the relationship of a cloth's points, mass, and springs that you're interested in; you'll be working mostly with those three aspects in cloth simulation. Take a closer look at how to simulate cloth motion using these points and springs.

Defining Cloth Points and Springs

A cloth point, being analogous to a vertex, is simply a point in 3D space. As such, you can define its coordinates using a `D3DXVECTOR3` object. This object initially contains the same coordinates as the matching cloth mesh's vertex, as well as the mass of the cloth point.

Actually, you need to use two mass−related values, the first being the actual mass of the object and the second being the value of 1 divided by the mass value. (This second value is used to aid in certain calculations.)

I'll explain why you need these two mass values in a little bit; for now, I want to show you how to define those cloth points. Because your cloth mesh can consist of many points, it makes sense to use an array of vectors and floating−point values. Creating a class to contain each point's data is perfect, and creating an array of those class objects to hold the cloth mesh's points is just what you need to do.

```
class cClothPoint  {
  D3DXVECTOR3 m_vecPos;  // 3−D coordinates of point
  float m_Mass;          // Point's mass (0=pinned)
  float m_OneOverMass;   // 1 / Mass (0=pinned in place)
};
cClothPoint *ClothPoints = new cClothPoint[NumPoints];
```

As for the springs, you need to store the two index numbers (in the array of points) of the two points to which the spring connects. Each spring also has an initial distance between the points, known as the spring's resting length. The resting length value of the spring is important because it is used to calculate whether a spring is being stretched or constricted during cloth simulation.

Finally, you need to define a spring's stiffness and damping values. These values are important because they determine how the spring reacts to force. I'll show you how these two values work shortly; for now, just define them as a couple of floating−point variables.

You can define a class to contain each spring's data, as well as an array of spring class objects to use, as follows:

```
class cClothSpring {
     DWORD m_Point1;         // 1st point in spring
     DWORD m_Point2;         // 2nd point in spring
     float m_RestingLength;  // Resting length of spring
     float m_Ks;             // Stiffness of spring
     float m_Kd;             // Spring's damping value
};
cClothSpring *ClothSprings = new cClothSpring[NumSprings];
```

Having an array of points and springs is really nothing special, especially if those arrays of data don't contain the essential information about the cloth mesh's points and springs. Take a moment to see where to get that point and spring data.

Obtaining Cloth Data from Meshes

Assume that you've gone through the steps and loaded a mesh into an `ID3DXMesh` object, and that it is the mesh object from which you want to create a cloth object. In the previous section, I defined two variables to allocate the points and springs arrays−`NumPoints` and `NumSprings`. At that time those two variables were undefined, but now that you have a valid mesh object (assume that it's called `pClothMesh`) from which to obtain information, you can calculate those two variables as follows:

```
// pClothMesh = pre−loaded ID3DXMesh object
NumPoints = pClothMesh−>GetNumVertices();
NumSprings = pClothMesh−>GetNumFaces() * 3;
```

Now that you've got the number of points and springs in the cloth's arrays, you can start pulling out the vertex data and constructing the array. The first step to obtaining the mesh's vertex data is to calculate the size of each mesh vertex using `D3DXGetFVFVertexSize`, as shown here:

```
DWORD VertexStide = D3DXGetFVFVertexSize(pClothMesh−>GetFVF());
```

When you know the vertex's size (called the *vertex stride*), you can lock the mesh's vertex buffer, thus obtaining a vertex buffer data pointer used to access the vertex data. Using the vertex buffer data pointer, you can iterate through each vertex in the mesh and obtain the coordinates of the vertices (the first three float values of each vertex).

So why did you go to the trouble of calculating the size of each vertex? When you are iterating the vertices in the vertex buffer, you aren't sure how much data each vertex contains. Since you're only interested in the coordinates of each vertex, you can use the vertex size to skip the remaining data and get to the next vertex in the list.

Note It's really not a good idea to use the number of faces as a reference to how many springs to create in your cloth. Since each face in the mesh can share any number of vertices, you're sure to have many duplicate springs. Later in this chapter, I'll show you a better way to construct a list of springs that ensures you don't have any duplicate ones in your list.

Now that you know what to do with the vertex size, try locking the vertex buffer, obtaining a data pointer, and iterating the list of vertices.

```
// Create a generic vertex structure to access vertex coords
typedef struct {
   D3DXVECTOR3 vecPos;
} sVertex;

// Lock the vertex buffer
BYTE *pVertices;
pClothMesh−>LockVertexBuffer(D3DLOCK_READONLY,                \
                             (BYTE**)&pVertices);

// Iterate list of vertices to get coordinates
for(DWORD i=0;i<NumPoints;i++) {

// Cast to the generic vertex structure
sVertex *pVertex = (sVertex*)pVertices;

// Store the cloth point coordinates
ClothPoints[i].m_vecPos = pVertex−>vecPos;
```

At this point, you also need to define a point's mass. Since you have no real source for this value (because a mesh doesn't define mass values), you can just set a default value of, say, 1. The same goes for the other mass value (`m_OneOverMass`)−set that to the value of 1 divided by the mass.

```
// Assign a mass of 1 and 1/mass
ClothPoints[i].m_Mass = 1.0f;
ClothPoints[i].m_OneOverMass = 1.0f / ClothPoints[i].m_Mass;
```

After setting the two mass values, you can go on to the next vertex and point, and continue on until all cloth points have been initialized with the proper data. You can then unlock the vertex buffer and continue, as shown here:

```
// Go to the next vertex in the list
pVertices += VertexStride;
}

// Unlock vertex buffer
pClothMesh−>UnlockVertexBuffer();
```

Cool, you're halfway to getting the essential cloth data you need to simulate cloth in your project! Next you need to convert the polygon edges into springs that will hold together your cloth. This time, you need to lock the mesh's index buffer and pull out the three indices that construct each face. Three points connect to form three edges, and each edge is assigned as a spring.

Accessing the index buffer is much easier than accessing the vertex buffer. I assume you're using 16−bit indices, because 32−bit indices are not widely supported at this time. If so, you need to grab three 16−bit values in a row for each face, starting at the beginning, and construct three springs, using every combination that connects two of the indices.

You can start by locking the index buffer.

```
unsigned short *pIndices;
pClothMesh->LockIndexBuffer(D3DLOCK_READONLY, \
                                     (BYTE**)&pIndices);
```

Now iterate through each face and grab the three indices used to construct the face. (You also need to declare a variable that tracks the spring you're currently creating.)

```
DWORD SpringNum = 0;
for(i=0;i<pClothMesh->GetNumFaces();i++) {
   unsigned short Index1 = *pIndices++;
   unsigned short Index2 = *pIndices++;
   unsigned short Index3 = *pIndices++;
```

Using the three indices, create three springs that represent each edge.

```
   // Create spring from 1->2
   ClothSprings[SpringNum].m_Point1 = Index1;
   ClothSprings[SpringNum].m_Point2 = Index2;
   SpringNum++; // Increment spring count

   // Create spring from 2->3
   ClothSprings[SpringNum].m_Point1 = Index2;
   ClothSprings[SpringNum].m_Point2 = Index3;
   SpringNum++; // Increment spring count

   // Create spring from 1->3
   ClothSprings[SpringNum].m_Point1 = Index1;
   ClothSprings[SpringNum].m_Point2 = Index3;
   SpringNum++; // Increment spring count
}
```

When you finish the entire list of indices and springs, you need to go back through the list and calculate each spring's resting length. This is the length between the points, calculated using the infamous Pythagorean Theorem, which states that the square of the length of the hypotenuse of a right triangle is equal to the sum of the squares of the legs. No need to whip out your math books to look up this theorem and code a function to handle it; instead, you can have D3DX do it for you with the following code:

```
for(i=0;i<NumSprings;i++) {
   // Get indices for each point in the spring
   unsigned short Point1 = ClothSprings[i].m_Point1;
   unsigned short Point2 = ClothSprings[i].m_Point2;

   // Calculate the vector difference in points
   D3DXVECTOR3 vecDiff = ClothPoints[Point2].m_vecPos - /
   ClothPoints[Point1].m_vecPos;

   // Use D3DX to calculate the length of the vector difference
   ClothSprings[i].m_RestingLength = D3DXVec3Length(&vecDiff);
}
```

At last, you're done creating the cloth points and springs! Now you can get those cloth points moving!

Applying Force to Create Motion

Your cloth is very bland and static at this point because you really haven't done anything to it except store each point's coordinates and spring data. To actually make your cloth move and flow, you need to apply *force*.

Every cloth point has a separate force affecting it, or rather, a collection of applied and natural forces. These forces can come from any source−the wind, gravity, friction, or just somebody tugging on the cloth. The springs act as another type of force; whenever two points connected by a spring move away from or toward each other, the spring responds by expanding or contracting according to Hooke's Law−that is, until the force from the spring counters the external forces. In other words, the springs are trying to apply enough force to maintain static equilibrium.

Another aspect of cloth force and motion is velocity. As forces, such as gravity, affect your cloth points, they gain momentum and velocity. That means your points will continue to move in the direction a force is pushing them until something, such as friction or an opposing force, slows that point down. Using acceleration and velocity results in a very realistic piece of flowing cloth, and realism is one thing you definitely want for your project.

Note Momentum is calculated by multiplying the mass by the acceleration vector (F=ma). It's the momentum of a point, not the force vectors, that increases the point's velocity and eventually causes the point to move.

In your cloth simulation, forces such as gravity and velocity are represented as directional vectors that are used to move each point in your cloth. The magnitude of each vector determines the amount of force exerted. To emulate these forces correctly, you need to add two relevant vectors to your cloth point class.

```
class cClothPoint {
   D3DXVECTOR3 m_vecPos;      // 3−D coordinates of point
   float       m_Mass;        // Point's mass (0=pinned)
   float       m_OneOverMass; // 1 / Mass (0=pinned in place)
   D3DXVECTOR3 m_vecForce;    // Force vector (acceleration)
   D3DXVECTOR3 m_vecVelocity; // Velocity vector
};
```

Initially, a cloth point's force and velocity are set to 0, meaning no directional movement or force is applied. For every frame of animation, you need to reset your force vectors to 0 and begin applying your external forces, such as wind and gravity, as well as the internal forces from the springs. This net amount of forces is what you are tracking in the `m_vecForce` vector.

To clear the force vector, just iterate the list and reset the values, as I have done here.

```
for(DWORD i=0;i<NumPoints;i++)
   ClothPoints[i].m_vecForce = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
```

You should clear the velocity of each point at the beginning of your simulation using the following bit of code:

```
for(DWORD i=0;i<NumPoints;i++)
   ClothPoints[i].m_vecVelocity = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
```

After you clear out the force and velocity vectors, you can begin applying some sort of force to your points. Let's start with the most basic of forces−gravity and wind.

Applying Gravity and Wind

Gravity is a natural force that attracts objects to one another. Heavier objects attract light ones. On Earth, that means all objects fall to the ground (another instance in which the big guy wins). Your cloth mesh is no different. To properly simulate the motion of cloth, each point in the cloth mesh should be allowed to fall to the ground (with the exception of cloth points that are pinned or attached to some object, thus keeping those points in place).

In real life, objects increase in velocity as they fall. For all objects, the acceleration is more or less a constant value of 9.8 m/s^2. One second after you drop an object, its velocity is 9.8 m/s^2. After two seconds, the velocity of the object is 19.6 m/s^2. The reason that some objects fall faster (achieve a higher velocity) than others is that objects with different shapes tend to encounter more air resistance (an opposing force) as they fall, thus slowing down or stopping their acceleration. *Terminal velocity* is the maximum velocity an object can obtain before the upward force from air resistance halts the increase in the object's acceleration.

You can represent gravity in your simulation as a directional vector. This directional vector is added to each point's force vector to make the points move. You can use the following code to define your gravity vector (assuming that you want gravity to pull points downward in the negative y−axis):

```
// Create a gravity force vector (−9.8 is the amount of pull)
D3DXVECTOR vecGravity = D3DXVECTOR3(0.0f, −9.8f, 0.0f);
```

After you define the gravity vector, you can add it to each cloth point. (Remember to clear out your points' force vectors first.) Notice in the following bit of code that you're scaling the gravity vector by the mass value. This is very important because all forces are eventually scaled down according to their mass. (To calculate *momentum*, remember that the force to push an object is the mass times its acceleration.) This scaling of the gravity vector at this point ensures that all objects will fall at the proper speed, regardless of their mass.

Tip Because the gravity vector uses a directional vector, you can have gravity that pushes up, left, right, or any combination of directions you want. Imagine using reverse gravity settings to make cloth float upward!

```
for(DWORD i=0;i<NumPoints;i++)
   ClothPoints[i].m_vecForce += (vecGravity * \
                                                         ClothPoints[i].m_Mass);
```

Wind is another force common on Earth. You know how cloth just loves to catch the wind and flutter about? Well, it's the wind that creates the forces that moves your cloth's points. You can easily model the same effect in your cloth simulation functions. Wind is also represented as a directional vector.

```
// Blow wind sideways along the x−axis
D3DXVECTOR3 vecWind = D3DXVECTOR3(0.5f, 0.0f, 0.0f);
```

At first, you might think that you apply a wind force the same way you apply gravitational forces−by adding a directional vector to each point's force. I'm sorry to say it's not that easy. To apply wind forces, you have to loop through every face in the cloth and, based on the direction each face is pointing, calculate the directional force received by the wind's directional vector.

As you can see in Figure 13.2, each face in your mesh has a normal, which is the direction that the face is pointing. Figure 13.2 also shows the direction of the wind you are applying, and the angle between these two vectors.

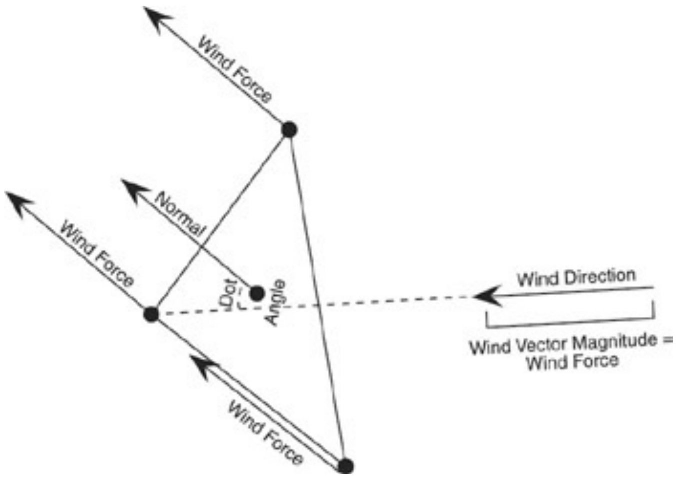


Figure 13.2: *The angle between the face's normal and the wind vector is used to calculate the amount of force to apply to each point.*

The angle between the face's normal vector and wind vector is very important−it determines how much force to apply to the points. To help understand this concept, think about when you are driving your car and you stick your hand out the window. If your hand is parallel to the ground, the wind whips above and below your hand, and does not push it back. Turn your hand, however, and the amount of force (from the wind) pushing back your hand increases.

As you can see, the angle between the vectors determines how much of the wind's force to apply. To calculate this angle, you perform a dot−product calculation on the face's normalized vector with the wind vector. The only problem at this point is, where do you get the face's normal?

Since you're using an `ID3DXBaseMesh` object, you can obtain the index list that defines which vertices are used by each face in your mesh. If you grab these indices beforehand, you can use them to grab the current positions of each vertex, and then use those to calculate a cross−product vector that represents your face's normal.

To get the indices before you begin your cloth simulation, you can allocate an array of 16−bit values (or 32−bit, depending on your mesh settings) and fill them with the values from your mesh's index buffer.

```
// pClothMesh = pre−loaded ID3DXMesh object

// The index buffer to contain your mesh's face indices
unsigned short *FaceIndices = NULL;

// Allocate the array of indices based on the number of
// faces contained in the mesh object. Remember there
// are 3 indices per face. There's room for 32−bit indices.
DWORD NumFaces = pClothMesh−>GetNumFaces();
FaceIndices = new DWORD[NumFaces * 3];

// Now, lock the mesh to get the indices
unsigned short *pIndices;
pClothMesh−>LockIndexBuffer(D3DLOCK_READONLY,(void**)&pIndices);

// Go through each index and store it
for(DWORD i=0;i<NumFaces*3;i++)
   FaceIndices[i] = (unsigned short)#x002A;Indices++;

// Unlock the index buffer
pClothMesh−>UnlockIndexBuffer();
```

Upon completion, the previous bit of code will have created an array of indices for you− three indices per face. Using these indices, you can then iterate through the list and calculate each face's normal.

```
for(i=0;i<m_NumFaces;i++) {
   // Get three vertices that construct face
   DWORD Vertex1 = FaceIndices[i*3];
   DWORD Vertex2 = FaceIndices[i*3+1];
   DWORD Vertex3 = FaceIndices[i*3+2];

   // Calculate face's normal

   D3DXVECTOR3 vecV12 = ClothPoints[Vertex2].m_vecPos  −                  \
                                                ClothPoints[Vertex1].m_vecPos;
   D3DXVECTOR3 vecV13 = ClothPoints[Vertex3].m_vecPos − \
                                                ClothPoints[Vertex1].m_vecPos;
   D3DXVECTOR3 vecNormal;
   D3DXVec3Cross(&vecNormal, &vecV12, &vecV13);
   D3DXVec3Normalize(&vecNormal, &vecNormal);
```

Now that you have the face's normal (stored in the `vecNormal` vector object), you calculate the dot product with the wind's vector to determine the angle between the two.

```
// Get dot product between normal and wind
float Dot = D3DXVec3Dot(&vecNormal, vecWind);
```

With the dot product in hand, you can scale the face's normal vector to calculate the amount of force to apply to each point used by the face. (Remember that each vertex has a matching point.)

```
   // Scale normal by dot product
   vecNormal * = Dot;
   // Apply normal to point's force vector
   ClothPoints[Vertex1].m_vecForce += vecNormal;
   ClothPoints[Vertex2].m_vecForce += vecNormal;
   ClothPoints[Vertex3].m_vecForce += vecNormal;
}
```

Once you've iterated through all the face indices, you are left with all your points' force vectors filled to the brim with the appropriate values, which allows your cloth to flutter in the wind! Now you're ready to move on to computing the next important set of forces used in your cloth simulation−the forces from your cloth's springs.

Applying Spring Forces

Each spring in the cloth mesh starts with an initial resting length. This represents the distance between the two points that the spring connects. When it is time to resolve the springs and push/pull the cloth points, you need to calculate the current length of every spring and decide how it affects the forces to apply to the points.

For springs that are shorter than their resting lengths, you must stretch out the spring and push the attached points away from each other. For springs that are longer than their resting lengths, you need to constrict the spring and pull the attached points closer together.

This stretching and pulling all happens thanks to Hooke's Law, which states that the power of any springy body is in the same proportion to the extension of the body. This means that the force that a spring exerts is proportionate to the length of the spring (or rather, the difference in length between the spring's current and resting lengths). Expressed mathematically, Hooke's Law looks like this:

```
F = k . x
```

F represents the applied force, and x is the spring's extension (the difference in length between the spring's current and resting lengths, as illustrated in Figure 13.3). k is the *spring constant*; it represents a ratio of sorts that determines how to scale the extension amount to generate a force value. The larger the spring constant amount, the greater the force applied.

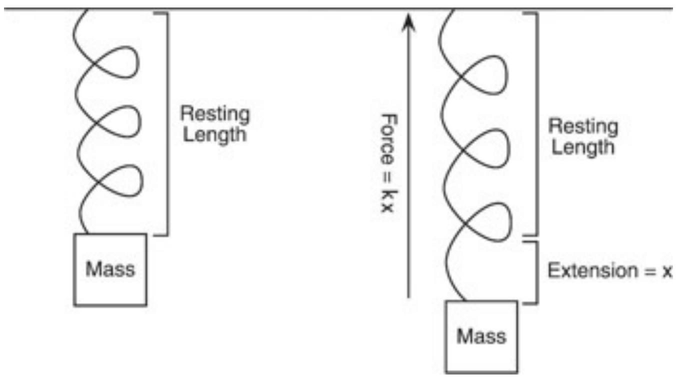


Figure 13.3: The spring on the left is at rest (it has obtained equilibrium), whereas the spring on right has been stretched. The force applied by the spring is calculated from the extension of the spring and a spring constant value.

For example, suppose you want to calculate the amount of force a spring exerts. This spring has a resting length of 128 units and is currently stretched out to 200 units. Subtracting the resting length from the current length gives you a value of 200−128= 72; this is the extension value x. Multiply the extension value by the spring constant k. Suppose k equals 0.4. This will give you a result of −0.4*72= 28.8, which is the amount of force applied (F).

28.8 is the force used to pull together your spring's points. Each point on either end of the spring receives equal and opposite forces of the other. Once you have this force value, you can add it to the other forces of the points. Later, you will use this net force you have been accumulating to calculate acceleration, which in turn alters the velocity of the points.

One thing I haven't mentioned yet is where you get the spring constant value to use in your formula to calculate the applied force. Remember earlier, when I had you define a floating−point value that represents the stiffness of your spring? Well, that stiffness value is your spring constant! Normally represented by the symbol k or ks, you can use this spring constant/stiffness value in your formula to determine how to scale the applied force of a spring during resolution.

I'm sure you're wondering what acceptable values of stiffness you can use. Well, there isn't an easy answer. The higher the stiffness, the greater the force applied from a spring. Typically, this is just fine−setting a high stiffness value would seem to work great. The only problem is that during simulation, this high stiffness value can cause your springs to apply too much force, resulting in your cloth dancing around in an unpredictable way. If you use a stiffness value that is too low, your cloth will stretch out, making it look like a piece of rubber.

So again, which stiffness value is just right? For the examples in this chapter, I use a stiffness value of 4. When you get your cloth simulation up and running, I invite you to try different values and see what works best for you.

To resolve the springs during the cloth simulation, you iterate all springs in the mesh using a `for...next` loop and get the current length of the spring (the distance between its two points).

```
for(DWORD i=0;i<NumSprings;i++) {

   // Get the two point index numbers
   DWORD Point1 = ClothSprings[i].m_Point1;
   DWORD Point2 = ClothSprings[i].m_Point2;

   // Get the vector between points
   D3DXVECTOR3 vecSpring = ClothPoints[Point2].m_vecPos − \
                               ClothPoints[Point1].m_vecPos;

   // Get the length of the vector between points

   float SpringLength = D3DXVec3Length(&vecSpring);
```

The current distance between the spring's two points is important. Remember from earlier in this chapter that a spring can stretch or constrict. By first calculating the vector between the spring's points, you can derive the current length of the spring using the `D3DXVec3Length` function. To compute the force of the spring, use Hooke's Law. This involves multiplying the spring's extension amount (the resting length subtracted from the current length) by the spring's stiffness value to obtain a spring force scalar.

```
   // Calculate applied force scalar
   float SpringForce = Spring−>m_Ks *
                              (SpringLength − Spring−>m_RestingLength);
```

The `vecSpring` vector is also very important in determining the direction to move your points. By normalizing the current spring's length vector and scaling it by `SpringForce`, you can quickly calculate the force vector to apply to each point connected to the spring.

```
   // Normalize spring vector
   vecSpring /= SpringLength;

   // Multiply by the force value
   vecSpring * = SpringForce;
```

At this point you have a vector that represents the amount of force to apply to each of the spring's points. Add this force vector to the first point's accumulated force vector, and subtract the force vector from the second point's accumulated force vector to move them in the proper direction (either toward or away from each other).

```
   // Apply force to points' force vector
   ClothPoints[Point1].m_vecForce += vecSpring;
   ClothPoints[Point2].m_vecForce −= vecSpring;
}
```

That's it for resolving the cloth's springs! At this point, you're almost ready to update the velocity and position of each point based on the net force you have accumulated, but first you must apply the most important force−friction.

Damping Motion with Friction

To keep things realistic, your cloth needs to adhere to the laws of friction. As your cloth moves through a medium such as air, it slows down. Friction is actually an opposing force, whether this force is from gravity, air turbulence, or friction. To keep things simple I'll refer to this combination of forces as simply *friction*.

If you don't apply friction to the cloth points, you'll be presented with a problem−the cloth will never stop moving, and the cloth simulation will suffer from poor stability (the velocities will cause your mesh to flutter about out of control). Sure, the force of gravity will pull the cloth downward, but the horizontal motion will never cease. By applying friction, you force the motion of the cloth to slowly come to an end, as long as no strong forces act against the friction. Also, by using a damping friction on the springs, you increase stability by reducing the amount of applied force from the spring.

To apply friction (called *linear damping* in this case) to the cloth's points, you need to reduce the force vector of each point by a small percentage of the point's velocity vector. (Damping is actually a force proportionate to the velocity, F=kV.) This percentage is specified as a floating−point number, with typical values ranging from −0.04 to −0.1. (The negative range implies that you are reducing the force rather than increasing it.) The higher the value, the greater the amount of friction applied. For example, a value of −1.0 gives the effect of your cloth floating in a vat of oil!

Suppose you are using a linear damping value of −0.04. Go ahead and iterate all your points right now. For each point you want to add the point's velocity vector, scaled by the damping value, to the point's force vector, as in the following code:

```
for(DWORD i=0;i<NumPoints;i++) {
   ClothPoints[i].m_vecForce += (−0.04f *  \
                        ClothPoints[i].m_vecVelocity);
}
```

For the friction to apply to springs (called *spring damping friction*), you need to add a damping value to the spring's force calculation, as shown previously. This damping value, as you can probably guess, was defined as part of your spring class's data! The friction value must be a little higher than the linear damping value, with typical values being 0.1 to 0.5. For the examples here, I will use a value of 0.5.

To apply the spring damping, go back to your spring force calculations.

```
for(DWORD i=0;i<NumSprings;i++) {

   // Get the two point index numbers
   DWORD Point1 = ClothSprings[i].m_Point1;
   DWORD Point2 = ClothSprings[i].m_Point2;

   // Get the vector between points
   D3DXVECTOR3 vecSpring = ClothPoints[Point2].m_vecPos − \
                           ClothPoints[Point1].m_vecPos;

   // Get the length of the vector between points
   float SpringLength = D3DXVec3Length(&vecSpring);

   // Calculate applied force scalar
   float SpringForce = Spring−>m_Ks *
                       (SpringLength − Spring−>m_RestingLength);
```

At this point, you need to calculate your spring damping scalar value, based on the normalized relative velocity of the two points (the difference in the points' velocities) and the spring's damping value. To calculate the normalized relative velocity of the points, you only need to subtract the two velocity vectors and divide by the current length of the spring.

```
   // Get the relative velocity of the points
   D3DXVECTOR3 vecVelocity = CState2−>m_vecVelocity − \
                             CState1−>m_vecVelocity;
   float Velocity = D3DXVec3Dot(&vecVelocity, \
                             &vecSpring) / SpringLength;
   // Apply the spring's damping value (m_Kd)
   float DampingForce = Spring−>m_Kd *  Velocity;
```

Continue with the rest of the code to compute the spring's force, but instead of only applying the spring's force (`SpringForce`) to the spring vector, you need to add the spring's force scalar and damping force scalar (`DampingForce`) together and then apply the result to the spring vector.

```
   // Normalize spring vector
   vecSpring /= SpringLength;

   // Multiply by the force scalars (spring and damping scalars)
   vecSpring * = (SpringForce + DampingForce);

   // Apply force to points' force vector
   ClothPoints[Point1].m_vecForce += vecSpring;
   ClothPoints[Point2].m_vecForce −= vecSpring;
}
```

After you adjust your forces by a frictional value, you can apply those forces to each point's velocity and apply the motion to each cloth point.

Applying Forces and Moving Points over Time

At this point, each point should have its forces computed and stored in the force vector. These force vectors represent the amount of acceleration you need to apply to each point's velocity over time. These velocity values are also directional vectors that determine the direction and speed that each point moves during simulation.

So, if you had a force vector of (0, −9.8, 1), your acceleration is said to be 9.8 m/s in the negative y−axis and 1 m/s in the positive z−axis. This translation from force to acceleration (using the same force vector to represent acceleration) is a little misleading, but it works perfect for your simulation, so we'll ignore the laws of physics on this point.

Something of concern at this point in time is, well...time. You see, every time you update your simulation, you need to express how much time has passed so that your cloth's points can move according to this time. Suppose, for example, that you only want to calculate the motion of the points over a period of 400 milliseconds. How do you do this without having to run the simulation 400 times, once for each millisecond? You want real−time cloth simulation, darn it!

Now to worry my friend! Using what is called *explicit integration*, you can estimate the acceleration achieved over time, and how that acceleration affects the velocity of the point over the same period of time. So instead of running the simulation every millisecond, you can run it every 40 milliseconds, thus tremendously speeding up things!

In this example, I'll use *Forward Euler integration*, which allows you to scale a vector or value by a set sampling amount (time). Using a point's force (which represents the acceleration at this point), you can calculate how much velocity has accumulated over a period of time, as follows:

```
// TimeElapsed = number of SECONDS elapsed
ClothPoints[Num].m_vecVelocity += TimeElapsed *                    \
                                 ClothPoints[Num].m_vecForce;
```

Tip If you want to specify time in milliseconds instead of seconds, as shown in the code here, just specify a fractional decimal value. To calculate this fractional value, divide 1 by 1000 and multiply by the number of milliseconds. For example, one millisecond is 0.001, ten milliseconds is 0.01, and a hundred milliseconds is 0.1.

Over that same period of time, you can calculate how much the point would have moved based on the velocity.

```
ClothPoints[Num].m_vecPos += TimedElapsed *                         \
                             ClothPoints[Num].m_vecVelocity;
```

So Forward Euler integration is nothing more than scaling your vectors. Pretty simple, eh? One thing I didn't mention is that you need to take into consideration each point's mass. Remember, the more mass an object has, the more force you must apply to move that object. I'm talking about *momentum* here, which states that the point's combined forces (represented by your force vector) need to be scaled by the mass to calculate the actual acceleration to apply. Scaling the force vector by the mass value (actually, the 1/mass value) during integration does this nicely.

```
ClothPoints[Num].m_vecVelocity += TimeElapsed *                       \
                                  ClothPoints[Num].m_OneOverMass *  \
                                  ClothPoints[Num].m_vecForce;
```

To sum this up, you must multiply each point's force vector by the mass (1 /mass) and elapsed time, and add the resulting vector to the point's velocity vector. Here's a snippet of code that does all this for each point in your cloth:

```
// TimeElapsed = number of seconds to process
for(DWORD i=0;i<NumPoints;i++) {

   // Integrate velocity
   ClothPoints[Num].m_vecVelocity += TimeElapsed *  \
                               ClothPoints[Num].m_OneOverMass *  \
                               ClothPoints[Num].m_vecForce;

   // Integrate position
  ClothPoints[Num].m_vecPos += TimedElapsed *  \
                            ClothPoints[Num].m_vecVelocity;
}
```

And there you have it, my friend! You have successfully calculated the velocity and updated the position of each cloth point in your mesh! The next thing on your agenda is to take that point data and use it to rebuild and render your cloth mesh.

Rebuilding and Rendering the Cloth Mesh

Because your cloth's points have changed positions, you must rebuild your original mesh so the changes will be visible. Since you've gone to the trouble of loading the original mesh from an .X file into an `ID3DXMesh` object, all you have to do is lock the mesh's vertex buffer, stuff in the points' coordinates, and rebuild the normals (if your original mesh used them).

Once again, create a generic vertex structure that contains only a vector. This vector is used to access the coordinate data of your mesh's vertices.

```
typedef struct {
   D3DXVECTOR3 vecPos;
} sVertex;
```

From here, it's a matter of locking the mesh's vertex buffer and iterating all cloth points (again!), and then stuffing back in the data from the coordinates.

```
// Lock the mesh's vertex buffer
BYTE * pVertices;
pClothMesh−>LockVertexBuffer(0, (BYTE**)&pVertices);

for(DWORD i=0;i<NumPoints;i++) {

   // Cast a vertex pointer
   sVertex * pVertex = (sVertex*)pVertices;

   // Store vertex coordinates
   * pVertex = ClothPoints[i].m_vecPos;

   // Go to next vertex
   pVertices += VertexStride; }

   // Unlock the vertex buffer
   pClothMesh−>UnlockVertexBuffer();
```

After you've rebuilt the mesh, you can render it normally by looping through each of the mesh's materials and using the `ID3DXMesh::DrawSubset` function to draw each subset. If you plan to reuse your original mesh, perhaps to reset the cloth simulation, you need to restore the mesh's original vertex coordinates.

Restoring the Original Mesh

One thing you might have noticed is that the cloth point's data is very dynamic. There's no easy way to restore the initial orientation of each cloth point in the mesh. Why would you want to restore the original mesh? Perhaps to restart the simulation or reuse the cloth mesh in its original state.

The easiest method of restoring the original mesh's cloth point data is to add another vector to your `cClothPoint` class.

```
class cClothPoint {
   D3DXVECTOR3 m_vecOriginalPos; // Initial coordinates
   D3DXVECTOR3 m_vecPos;         // 3−D coordinates of point
   float       m_Mass;           // Point's mass (0=pinned)
   float       m_OneOverMass;    // 1 / Mass (0=pinned in place)
   D3DXVECTOR3 m_vecForce;       // Force vector (acceleration)
   D3DXVECTOR3 m_vecVelocity;    // Velocity vector
};
```

The additional vector (`vecOriginalPos`) stores the initial coordinates of your cloth's points. When you are creating the cloth point array, make sure to store the initial position in this new vector.

```
// ... After loading cloth points into array
for(DWORD i=0;i<NumPoints;i++)
   ClothPoints[i].m_vecOriginalPos = ClothPoints[i].m_vecPos;
```

When it's time to restore your original cloth point data, just copy the data from this new vector into the point's position vector.

```
for(DWORD i=0;i<NumPoints;i++)
```

```
   ClothPoints[i].m_vecPos = ClothPoints[i].m_vecOriginalPos;
```

Using an initial position is perfect for restoring the original mesh's data, and it is also useful when you need to determine the distance between a cloth point's current and initial positions, such as when you are dealing with soft body meshes later in this chapter.

For now, just continue to improve your cloth simulation by adding more springs to your mesh, thus improving the stability of the cloth during simulation.

Adding More Springs

Up to this point, I've referred to springs as being constructed from the polygon edges of a mesh. Realistically, this is a very sloppy way to create springs from mesh data if precise cloth simulation is your goal. Take Figure 13.4, for example. You'd think the springs shown would hold together the cloth mesh perfectly.

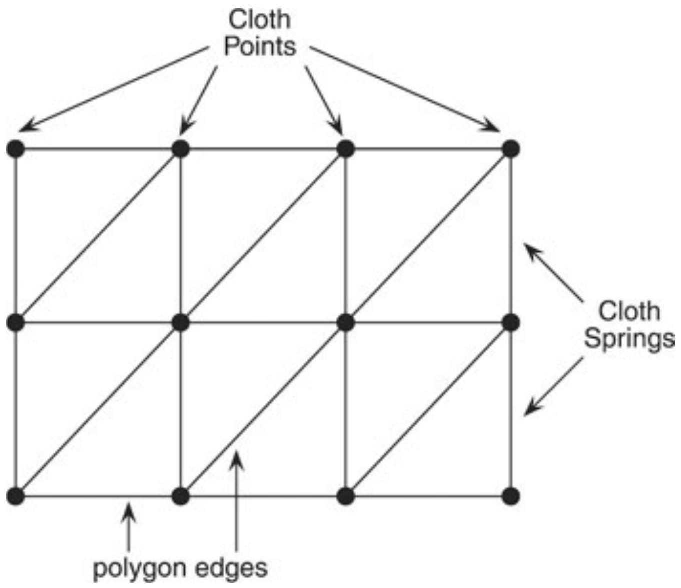


Figure 13.4: A set of springs was created using the polygon edges of the cloth mesh shown.

Everything in Figure 13.4 looks fine until you run it through cloth simulation. The problem here is a cloth mesh that uses only the polygon edges as the springs might fold inward on itself over time, as if it were made of a very thin material. That might be fine for most instances, but what about those times when you want a stiffer cloth that is very stubborn about changing shape?

The secret is that the more springs you have in your mesh, the stiffer the cloth becomes. That's right−by adding a few more springs you can make your cloth crease and bend instead of bunching up into a flimsy lump of vertices. Of course, the placement of those springs is what really matters, so take a look back at your mesh in Figure 13.4 to see what to do. In Figure 13.5, you see the new mesh, this time with a few new springs added.

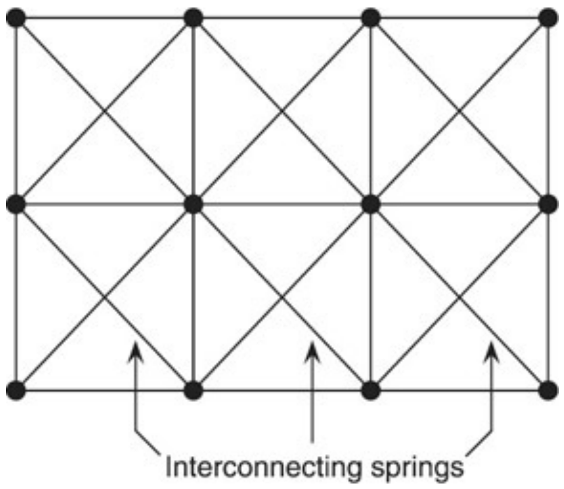


Figure 13.5: The cloth mesh now has a series of interconnected springs spread across its faces.
You remember that a spring will pull or push two points toward or away from each other, depending on their distance from one another. Taking a closer look at Figure 13.5, you can see that with the strategic placement of a few additional springs, you can enforce the structure of the cloth because those springs will force every cloth point away from other points if the cloth tries to fold in on itself. It's hard to visualize what I'm talking about at first, but give it a minute or two to sink in. Just think of the new springs as a sort of crease−repellent. As folds and creases form, the new springs force the creases open.

The easiest way to add more springs to your mesh is to create a list of springs you want and build them using that list. You can obtain this list by using a custom .X parser. While you're at it, why not use a custom .X parser to also obtain your points' mass data!

Loading Mass and Spring Data from .X

Aside from the mesh data that you can load from an .X file, what else is there for you to use? Well, for a start, that mesh data doesn't contain information about your cloth points' mass values. Also, what about all those extra springs you might need to add to your cloth−you know, those springs that add more support to the cloth mesh?

Well, no need to fret−.X is perfect for storing all your extra cloth−related data, such as the points' mass values and springs to use. By adding two small templates in your .X files and a series of class objects in your source files, getting access to external point mass and spring data is extremely easy.

The two templates in question are defined as follows (with supporting GUID declaration macros to insert into your source code):

```
// DEFINE_GUID(ClothMasses,
//             0xf5ad0f93, 0x9bf2, 0x4bcf,
//             0xb7, 0xcf, 0xd6, 0x8c, 0xd6, 0xb5, 0x41, 0x56);
template ClothMasses {
   <F5AD0F93−9BF2−4bcf−B7CF−D68CD6B54156>
   DWORD NumPoints; // # of point mass values
   array FLOAT Mass[NumPoints]; // mass values
}

// DEFINE_GUID(ClothSprings,
//             0x8c08b088, 0x728e, 0x46c8,
//             0xbe, 0x87, 0x72, 0x67, 0x2b, 0x81, 0xdb, 0x11);
template ClothSprings {
   <8C08B088−728E−46c8−BE87−72672B81DB11>
   DWORD NumSprings;  // # of springs to load
   DWORD NumVertices; // NumSprings *  2
   array DWORD Vertex[NumVertices]; // spring points to use
}
```

The first template, `ClothMasses`, contains two variables. The first variable, `NumPoints`, describes the number of points in your mesh and the number of mass values to follow in the object. The second object, `Mass`, is an array of point mass values, which can be anywhere from 0 (the point is pinned in place) on up.

Here's an example data object that uses the `ClothMasses` template to contain six points' mass values:

```
ClothMasses {
   6;
   1.0, 0.0, 0.0, 1.0, 1.0, 1.0;
}
```

This list of mass values is analogous to the points in your cloth mesh−the first mass value is for the first cloth point, the second mass value is for the second cloth point, and so on. In my example, the first, fourth, fifth, and sixth mass values are set to 1, whereas the second and third mass values are 0.

The second template shown, `ClothSprings`, is used to define the points in your cloth mesh that are joined by springs. The `NumSprings` variable states how many springs are to be defined, and the `NumVertices` variable defines how many vertices (points) are affected. `NumVertices` should always equal twice the number of springs. Finally, `Vertex` is an array of index numbers that define which points are joined.

For example, suppose you want to define three springs that connect six vertices. Here's a sample data object that connects the points 0 and 1, 1 and 2, and 2 and 0.

```
ClothSprings {
   3;
   6;
   0,1,
   1,2,
   2,0;
}
```

Once these templates and data object are defined in your .X file, you can use a custom .X parser to read the appropriate values into your game. When you are creating your cloth data .X parser, you can even include your cloth simulation code in the derived class. In the cloth demo included with this book, I've done just that for you−included the cloth simulation in a single class.

Building an .X Parser for Cloth Data

Once again, the `cXParser`.X parser class from Chapter 3 comes to the rescue. Here you will derive the `cXParser` class into one that will scan for two different templates−one that contains cloth point mass values (`ClothMasses`) and the other that contains spring information (`ClothSprings`).

The data from each data object fits perfectly into the point and spring classes defined earlier in this chapter−all you need is a parser class to enumerate the data objects and grab the appropriate data. Here's a sample .X parser class that does just that for you:

```
class cClothMesh : cXParser
{
   protected:
      DWORD m_NumPoints; // # points in cloth
      cClothPoint *m_ClothPoints; // Point data

      DWORD m_NumSprings; // # springs in cloth
      cClothSpring *m_ClothSprings; // Spring data

   protected:
      // Parse an .X file for mass and spring data
      BOOL ParseObject(IDirectXFileData *pDataObj,
                         IDirectXFileData *pParentDataObj,
                         DWORD Depth,
                         void **Data, BOOL Reference)
   {
      const GUID *Type = GetObjectGUID(pDataObj);
      DWORD *DataPtr = (DWORD*)GetObjectData(pDataObj, NULL);

      // Read in cloth point masses
      if(*Type == ClothMasses) {

        // Get number of mass value assignments
        DWORD NumPoints = *DataPtr++;

        // Copy over mass values
        float MassPtr = (float*)DataPtr;
        for(DWORD i=0;i<NumPoints;i++) {
        m_ClothPoints[i].m_Mass = *MassPtr++;

        // Calculate one-over-mass value
        m_ClothPoints[i].m_OneOverMass = \
                (m_ClothPoints[i].m_Mass==0.0f) ? \
                0.0f:(1.0f/m_ClothPoints[i].m_Mass);
   }
  }

   // Read in spring data
   if(*Type == ClothSprings) {

      // Free prior springs' data
      delete [] m_ClothSprings; m_ClothSprings = NULL;

      // Get new number of springs and vertices
      DWORD NumSprings = *DataPtr++;
      DWORD NumVertices = *DataPtr++;

      // Allocate the springs
      m_ClothSprings = new cClothSpring[NumSprings];

      // Load each springs' data
      for(DWORD i=0;i<NumSprings;i++) {
         m_ClothSprings[i].m_Point1 = *DataPtr++;
         m_ClothSprings[i].m_Point2 = *DataPtr++;
      }
   }

   return ParseChildObjects(pDataObj, Depth, \
                            Data, Reference);
  }
}
```

The code I've just shown for the `cXParser` class is very basic, so I won't explain it in detail. All that the `ParseObject` function accomplishes is loading any spring or point mass data into an array of class objects to use in your program. To make your parser class really powerful, you can combine it into a class that creates and controls all aspects of a cloth mesh.

To actually use the .X parser object, just call `cClothMesh::Parse`, specifying the name of the .X file to use. You'll notice the cloth's point and spring data arrays are embedded in the parser class−just make sure you've already allocated the proper point class objects before parsing the .X file. Later in this chapter, you'll see how to construct a complete cloth mesh class that inherits the .X parser class to load the mass and spring data.

For now, it's time to start livening up your simulation by adding collision detection and response.

Working with Collision Detection and Response

One important aspect that I haven't mentioned up to this point is collision. Remember that your mesh is considered a solid object. It, too, will collide with objects in your virtual world, and it will need to react accordingly. For example, your character's cloak will roll off his shoulders and flap against his back. That cloak should never penetrate the character's mesh.

Perhaps your cloth is dangling from a pole. Your character (or any other object that brushes against the cloth) will cause the cloth to push away from the pole and flutter about as it settles. Talk about some cool ideas there−with collision detection and response, you can unlock some awesome potential in your cloth simulation!

Working with collision detection and response is suspiciously simple. After you apply each point's velocity, you check to see whether the point resides inside another solid object. For example, you can check whether a cloth point is inside a sphere by using a simple distance check, or you can perform a plane−point check to see on which side of the plane the point is located (see Figure 13.6).

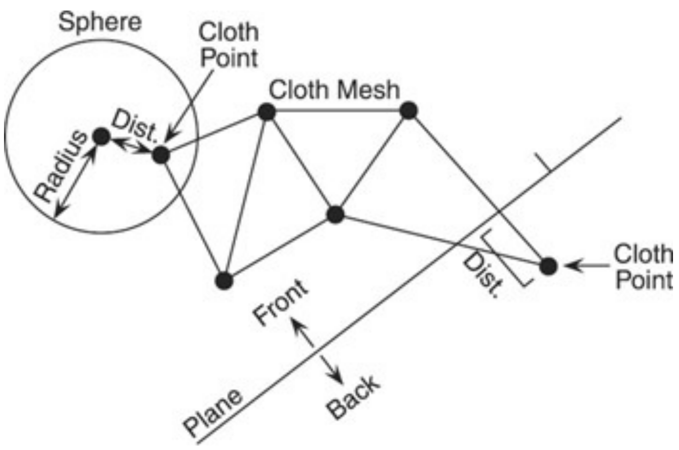


Figure 13.6: *A point has collided with a sphere if it's closer than the sphere's radius or if it is located on the back side of a plane.*

So there are two simple objects that you can use to perform collision detection−spheres and planes. You need to create a class to contain these collision objects.

Defining Collision Objects

To store the data that pertains to a collision object, you can use the following class (with two macros):

```
// Macros to define types of collision objects
#define COLLISION_SPHERE 0
#define COLLISION_PLANE 1

class cCollisionObject {
   public:
      DWORD            m_Type;               // Type of object

      D3DXVECTOR3      m_vecPos;             // Sphere coordinates
      float            m_Radius;             // Sphere radius
      D3DXPLANE        m_Plane;              // Plane values

      cCollisionObject     *m_Next;              // Next in linked list

   public:
      cCollisionObject() { m_Next = NULL; }
      ~cCollisionObject() { delete m_Next; m_Next = NULL; }
};
```

There are four main variables inside the `cCollisionObject` class. You need to set the `m_Type` variable to one of the two macros that represents the type of object the class contains−either a sphere (`COLLISION_SPHERE`) or a plane (`COLLISION_PLANE`).

You need to store the center coordinates of a sphere collision object in `m_vecPos`. These coordinates are in 3D space, just like your 3D meshes. Also, if you're using a sphere object, make sure to set the radius of the sphere in the `m_Radius` variable.

If you're using a plane collision object, however, you only need to set the respective plane parameters inside the `m_Plane` object. The `m_Plane.a, m_Plane.b,` and `m_Plane.c` parameters are the normalized directional vectors of the plane, whereas `m_Plane.d` is the distance along the directional vector where the plane is located.

Moving on in the class, you see the `m_Next` pointer, which you use to contain a linked list structure. Using a linked list, you can create a whole slew of collision objects to use in your cloth simulation. Later in this section, you'll see how to use this linked list of objects.

With the variable declarations finished, it is time for the `cCollisionObject` functions. The class object contains only two functions, the constructor and destructor, which are used to clear the linked list pointer on initialization and free the linked list on destruction.

Aside from the `cCollisionObject` class, you can create another class that maintains the linked list of collision objects. I'm recommending a separate class so you can maintain multiple lists of collision objects, such as one for the static geometry and another for dynamic geometry. Your collision objects can move around, bumping into cloth objects and causing them to flutter.

This second class, called `cCollision`, is defined as follows:

```
class cCollision {
   public:
      DWORD             m_NumObjects; // # of objects
      cCollisionObject *m_Objects;    // Object list

   public:
      cCollision() { m_NumObjects = 0; m_Objects = NULL; }
      ~cCollision() { Free(); }
      void Free()
      {
         // Delete linked list of objects
         delete m_Objects; m_Objects = NULL;
         m_NumObjects = 0;
      }

      void AddSphere(D3DXVECTOR3 *vecPos, float Radius)
      {
         // Allocate a new object
         cCollisionObject *Sphere = new cCollisionObject();

         // Set sphere data
         Sphere->m_Type = COLLISION_SPHERE;
         Sphere->m_vecPos = (*vecPos);
         Sphere->m_Radius = Radius;

         // Link sphere into list and increase count
         Sphere->m_Next = m_Objects;
         m_Objects = Sphere;
         m_NumObjects++;
      }

      void AddPlane(D3DXPLANE *PlaneParam)
      {
         // Allocate a new object
         cCollisionObject *Plane = new cCollisionObject();

         // Set plane data
         Plane->m_Type = COLLISION_PLANE;
         Plane->m_Plane = (*PlaneParam);

         // Link plane into list and increase count
         Plane->m_Next = m_Objects;
         m_Objects = Plane;
         m_NumObjects++;
      }
};
```

The `cCollision` class declaration includes two variables (`m_NumObjects` and `m_Objects`) that contain the number of collision objects loaded and the collision object linked list, respectively. There are five functions at your disposal, starting with the constructor and destructor. These functions clear out the class's data and call the `Free` function, respectively.

The `Free` function has the job of deleting the linked list and zeroing out the number of objects loaded in the linked list. As for `AddSphere` and `AddPlane`, those two functions allocate a new `cCollisionObject` object and link it into the linked list maintained by the `cCollision` class object.

To add a sphere to the linked list, call `AddSphere` and specify the coordinates of the sphere's center and radius. To add a plane, pass a `D3DXPLANE` object that specifies that plane's normal and distance from the origin.

To actually put the collision object data to good use, you need to construct a function that scans each point in your cloth, and for each point, checks to see whether a collision with a collision object has occurred. This is a matter of some simple distance checking, as you'll soon see.

Detecting and Responding to Collisions

The function that you create to check collisions needs to iterate through all cloth points and, for each point, check whether a collision has occurred with any of the collision objects maintained in the linked list of objects. Call this function `CheckCollision`; it will take a pointer to the `cCollision` object for checking point−to−object collisions, as well as a transformation matrix that you can use to position and orient the collision objects in your 3D world.

```
void CheckCollisions(cCollision *pCollision, \
                     D3DXMATRIX *matTransform)
{
      // Go through each point
      for(DWORD i=0;i<NumPoints;i++) {

      // Don't process points w/0 mass
      if(ClothPoints[i].m_Mass != 0.0f) {
```

In the previous bit of code, you begin to iterate through all cloth points. I'm assuming the number of points is defined in the `NumPoints` variable and the points themselves have their data defined in an array called `ClothPoints`. During iteration, you want to first make sure that the point has a non−zero mass, which means it is allowed to move. If the point is allowed to move, then you can scan through each collision object and see whether the point collides.

```
       // Go through each collision object
       cCollisionObject *pObject = pCollision->m_Objects;
       while(pObject) {

       // Check if point collides with a sphere object
       if(pObject->m_Type == COLLISION_SPHERE) {
```

The first type of object with which you will check collision is a sphere. Remember that the sphere is positioned using a three−dimensional vector, and the sphere's radius is defined by a floating−point value that you set with a call to `AddSphere`. As Figure 13.7 demonstrates, a point collides with a sphere if the distance from the sphere's center to the point is less than the sphere's radius.

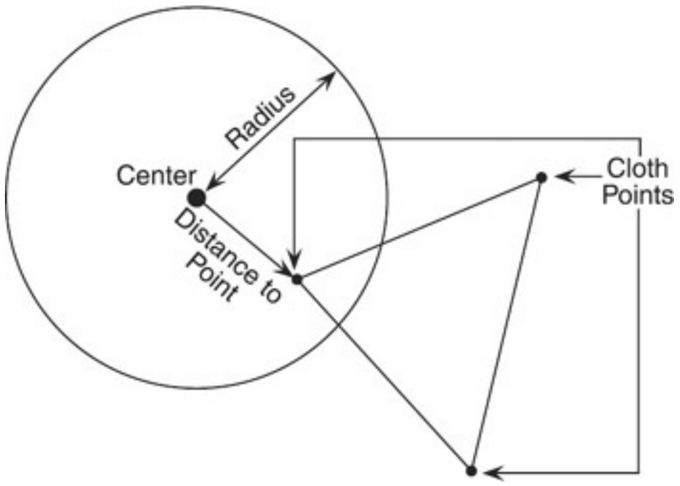


Figure 13.7: *A point collides with a sphere if the distance from the sphere's center to the point is less than the sphere's radius.*

Since you're using a transformation to position the collision objects, you need to apply the translation portion to the sphere's position vector before checking for collisions.

```
        // Put sphere coordinates into local vector
        D3DXVECTOR3 vecSphere = pObject->m_vecPos;

        // Translate sphere if needed
        if(matTransform) {
          vecSphere.x += matTransform->_41; // Translate x
          vecSphere.y += matTransform->_42; // Translate y
          vecSphere.z += matTransform->_43; // Translate z
        }
```

Now that you have a vector representing the position of the sphere, calculate a vector that represents the length from the point to the sphere's center.

```
         // Calculate a distance vector
         vecDist = vecSphere - ClothPoints[i].m_vecPos;
```

You can now compare the length of this vector to see whether it is equal to or less than the radius of the sphere. To avoid using a sqrt to calculate distances, just compare the squared values instead.

```
         // Get the squared length of the difference
         float Length = vecDist.x * vecDist.x + \
                        vecDist.y * vecDist.y + \
                        vecDist.z * vecDist.z;

         // If the length of the difference less than radius?
         if(Length <= (pObject->m_Radius*pObject->m_Radius)) {

         // Collision occurred!
```

Once you've determined that a collision has occurred (the distance between the point and the sphere is less than or equal to the radius of the sphere), you can handle the collision. I'm going to throw physics and time out the window to keep things simple at this point, and merely push the point outside the sphere. Also, you're going to assume that something as flimsy as a piece of cloth won't bounce off a collision object in any measurable fashion, so I'll also skip any coefficient of restitution calculations.

To push the point outside the sphere, you need to scale the normalized distance vector (the vector that represents the distance from the point to the sphere's center) by the difference in distance from the point to the sphere's edge, and subtract this vector from the point's position (and velocity, in order to decelerate the point).

```
           // Normalize the distance value and vector
           Length = (float)sqrt(Length);
           vecDist /= Length;

           // Calculate the difference in distance from the
           // point to the sphere's edge
           float Diff = pObject->m_Radius - Length;

           // Scale the vector by difference
           vecDist *= Diff;

           // Push the point out and adjust velocity
           ClothPoints[i].m_Pos -= vecDist;
           ClothPoints[i].m_Velocity -= vecDist;
          }
         }
```

That does it for checking and responding to point−to−sphere collisions! Next comes checking collisions against planes.

```
// Check if point collides with a plane object
if(pObject->m_Type == COLLISION_PLANE) {
```

To check whether a point has collided with a plane, you must first transform the plane and then perform a dot−product on the plane's normal and the point's position vector. This dotproduct, combined with the plane's distance component, will determine whether the point is located in front of the plane (not colliding) or behind the plane (colliding). Points behind the plane must be pushed out to the surface of the plane.

To transform the plane, you must first inverse and transpose the transformation matrix. This only needs to be done once in your function because you can reuse the matrix many times if you have more than one plane. The demo program for this chapter shows you how to create this transformation once. For now, I'll just calculate it every time. With this inversed and transposed matrix, you can then call `D3DXPlaneTransform` to transform the plane for you.

```
// Put plane in a local variable
D3DXPLANE Plane = pObject->m_Plane;

// Transform plane if needed
if(matTransform) {

 // Inverse and transpose the transformation matrix
 D3DXMATRIX matITTransform;
 3DXMatrixInverse(&matITTransform, NULL, matTransform);
 D3DXMatrixTranspose(&matITTransform, &matITTransform);

 // Transform the plane
 D3DXPlaneTransform(&Plane, &Plane, &matITTransform);
}
```

After you've transformed the plane, you can pull its normal vector out and use that to compute the dot−product with the point's position vector.

```
// Get the normal vector
D3DXVECTOR3 vecNormal = D3DXVECTOR3(Plane.a, \
                                    Plane.b, \
                                    Plane.c);

// Get the dot product between the plane's normal
// and the point's position
float Dot = D3DXVec3Dot(&ClothPoints[i].m_vecPos, \
                          &vecNormal) + Plane.d;
```

You'll notice that I added the plane's distance component (d) to the resulting dot−product. This ensures that the distance of the plane to the point is computed properly. If the resulting dot−product value is less than 0, then the point has collided with the plane and needs to be pushed out. To calculate the vector to add to your point's position and velocity vectors, you simply need to multiply the plane's normal by the dot−product value and add the result to your position and velocity vectors.

```
// Check if point is behind plane
if(Dot < 0.0f) {

 // Scale the plane's normal by the
 // absolute dot product.
```

```
          vecNormal *= (-Dot);

          // Move point and adjust velocity by normal vector
          ClothPoints[i].m_vecPos += vecNormal;
          ClothPoints[i].m_vecVelocity += vecNormal;
         }
       }
```

From here, you can go to the next collision object to check and wrap up the loop to iterate the rest of the cloth's points.

```
         // Go to next collision object
         pObject = pObject->m_Next;
      }
    }
  }
}
```

To use the `CheckCollisions` function, load your cloth mesh data and begin your simulation. After you've processed the cloth's forces and updated the cloth's points, call `CheckCollisions`. Here's a small snippet of code to demonstrate:

```
// Instance a collision object and transformation matrix
cCollision Collision;
D3DXMATRIX matCollision;

// Add a sphere to collision list and set matrix to identity Collision.AddSphere(&D3DXVECTOR3(0

// Process cloth mesh forces and update cloth point's
// Process collisions
CheckCollisions(&Collision, &matCollision);
```

Although not the most precise methods of collision detection and response in the world, the methods I just showed you should work for the majority of your game projects. For those of you who need exact precision, such as knowing the exact moment a cloth point hits a collision object, you need to investigate methods such as back−stepping time or time−stepping. I covered time−stepping back in Chapter 7; you can use the same techniques and apply them to cloth simulation.

For now, I want to show you how to take all the knowledge you've read so far and create a complete cloth mesh class that you can use to handle your cloth simulation.

Creating a Cloth Mesh Class

I'm sure you can't wait to add cloth simulation to your project now that you've seen it in action (and drooled over the results). Using what you've learned in this chapter, you can easily construct a class that handles a single cloth mesh. I've taken the liberty of creating such a class (or rather, a collection of classes) that you can use in your own projects.

You'll use three classes for your cloth mesh class.

- `cClothPoint` contains the information for every cloth point in the mesh.
- `cClothSpring` contains the information about the springs, including which cloth vertices are attached, the resting length of each spring, and the spring's stiffness and damping values

- `cClothMesh` contains a single mesh, an array of cloth points, a list of springs, and an array that contains the indices for each face in your mesh. It includes functions to load a mesh, add springs, set point masses, assign forces, and rebuild the mesh.

Note You will find the cloth mesh classes on this book's CD−ROM. (Check out the "Programs on the CD" sidebar at the end of this chapter for information.) To use the classes, just insert the appropriate source files into your project.

The only class you'll use directly is `cClothMesh`, which uses `cClothPoint` and `cClothSpring` to contain cloth point and spring data, respectively. Take a quick look at the `cClothPoint` and `cClothSpring` class declarations.

```
// Class to contain information about cloth points class cClothPoint
{
      public:
            D3DXVECTOR3 m_vecOriginalPos;                 // Original position of point
            D3DXVECTOR3 m_vecPos;                         // Current point coords

            D3DXVECTOR3 m_vecForce;                       // Force applied to point
            D3DXVECTOR3 m_vecVelocity;                    // Velocity of point

            float       m_Mass;              // Mass of object (0=pinned)
            float       m_OneOverMass;       // 1/Mass
};

// Class to contain information about springs class cClothSpring
{
      public:
            DWORD m_Point1; // First point in spring
            DWORD m_Point2; // Second point in spring
            float m_RestingLength; // Resting length of spring

            float m_Ks;    // Spring constant value
            float m_Kd;    // Spring damping value

            cClothSpring *m_Next; // Next in linked list

      public:
            cClothSpring() { m_Next = NULL; }
            ~cClothSpring() { delete m_Next; m_Next = NULL; }
};
```

As you can see, the `cClothPoint` and `cClothSpring` classes use the same data as the `cClothPoint` and `cClothSpring` structures you read about earlier in this chapter, so there's nothing new going on here. In fact, you've already seen the code for these two classes spread throughout this chapter, so I'll just jump past those classes and concentrate on the `cClothMesh` class.

`cClothMesh` uses the following class declaration:

```
// Class to contain a complete cloth mesh (w/.X parser) class cClothMesh : public cXParser
{
      protected:
            DWORD        m_NumPoints;                    // # points in cloth
            cClothPoint  *m_Points;                      // Points

            DWORD        m_NumSprings;                   // # springs in cloth
            cClothSpring *m_Springs;                     // Springs
```

At this point, there's an array of points in the mesh and the root spring in the linked list of springs. From here, some medial information is defined, such as the number of faces in the mesh, an array of face indices, and the size of a vertex (in bytes).

```
            DWORD m_NumFaces;     // # faces in mesh
            DWORD *m_Faces;       // Faces

            DWORD m_VertexStride; // Size of a vertex
```

Following the protected data members is a single protected function that you use to parse your .X file data objects and load the appropriate point mass and spring data. You use the .X parser functionality to process point masses and spring data, as you saw earlier in the "Building an .X Parser for Cloth Data" section.

```
      protected:
            // Parse an .X file for mass and spring data
            BOOL ParseObject(IDirectXFileData *pDataObj,
                             IDirectXFileData *pParentDataObj,
                             DWORD Depth,
                             void **Data, BOOL Reference);
```

From here, it's all public access to the functions! Starting your class's public function, you have the typical constructor and destructor that you use to set up the class's data, followed by the `Create` and `Free` functions.

```
      public:
            cClothMesh();
            ~cClothMesh();

            // Create cloth from supplied mesh pointer
            BOOL Create(ID3DXMesh *Mesh, char *PointSpringXFile = NULL);

            // Free cloth data
            void Free();
```

Using the `Create` function, you can supply a source mesh to convert to cloth data, as well as an optional file name of an .X file that contains point mass and spring data to load. The `Free` function is there for you to free the resources allocated to storing the cloth data; you should only call `Free` when you're finished using the cloth mesh class.

Next in the list of public functions are those which allow you to specify the forces to apply to your cloth points during simulation, update the points based on those forces, and process collisions.

```
      // Set forces to apply to points
      void SetForces(float LinearDamping,
                     D3DXVECTOR3 *vecGravity,
                     D3DXVECTOR3 *vecWind,
                     D3DXMATRIX *matTransform,
                     BOOL TransformAllPoints);

      // Process forces
      void ProcessForces(float Elapsed);

      // Process collisions
      void ProcessCollisions(cCollision *Collision,
                                        D3DXMATRIX *matTransform);
```

You've already seen the code to the `ProcessCollisions` function; it's the other two that I want to show you. Well, actually, I still want to show you the class's declaration, so I'll get back to the function code in a bit. The next two functions in the class rebuild the mesh after you've processed the cloth points (including recomputing the mesh's normal, if necessary, using the `D3DXComputeNormals` function) and restore the cloth's points to their initial positions.

```
      // Rebuild cloth mesh
      void RebuildMesh(ID3DXMesh *Mesh);

      // Reset points to original pose and reset forces
      void Reset();
```

Not that you should depend on an .X file to contain your spring data or point mass data−you also have a couple functions that allow you to add a spring to your list of springs and set the mass of a specific point.

```
      // Add a spring to list
      void AddSpring(DWORD Point1, DWORD Point2,
                             float Ks = 8.0f, float Kd = 0.5f);

      // Set a point's mass
      void SetMass(DWORD Point, float Mass);
```

To conserve memory and to ensure that no duplicate springs exist in your cloth mesh, the `AddSpring` function will scan the existing linked list for duplicates. If it finds a matching spring you are attempting to add, it will discard the new one. The `SetMass` function takes the point number you are altering and the new mass value to use. It also computes the new 1/mass value for you.

Finishing up the class declaration, you have a number of functions that merely retrieve the number of points, springs, and faces in your mesh, as well as pointers to the array of points, springs, and faces. You can use these functions to extend the usefulness of your cloth mesh class at a later time.

```
      // Functions to get point/spring/face data
      DWORD GetNumPoints();
      cClothPoint *GetPoints();

      DWORD GetNumSprings();
      cClothSpring *GetSprings();

      DWORD GetNumFaces();
      DWORD *GetFaces();
};
```

Now that you have declared the `cClothMesh` class, it is time to see the code for each of its functions. You've already seen the `ParseObject` and `ProcessCollisions` code, so I'll skip those. Starting at the top of the list, that leaves three important functions I want to show you: `Create, SetForces,` and `ProcessForces`. (You can check out the rest on the CD−ROM.)

You use the `Create` function to create a cloth mesh from an `ID3DXMesh` object you provide.

```
BOOL cClothMesh::Create(ID3DXMesh *Mesh, char *PointSpringXFile)
{
   DWORD i;

   // Free a prior mesh
   Free();
```

```
// Error checking
if(!Mesh)
return FALSE;

// Calculate vertex pitch (size of vertex data)

m_VertexStride = D3DXGetFVFVertexSize(Mesh->GetFVF());

////////////////////////////////////////////////////////////
// Calculate the spring information from the loaded mesh
////////////////////////////////////////////////////////////

// Get the # of faces and allocate array
m_NumFaces = Mesh->GetNumFaces();
m_Faces = new DWORD[m_NumFaces*3];

// Lock index buffer and copy over data (16-bit indices)
unsigned short *Indices;
Mesh->LockIndexBuffer(0, (void**)&Indices);
for(i=0;i<m_NumFaces*3;i++)
   m_Faces[i] = (DWORD)*Indices++;
Mesh->UnlockIndexBuffer();

// Get the # of points in mesh and allocate structures
m_NumPoints = Mesh->GetNumVertices();
m_Points = new cClothPoint[m_NumPoints]();

// Lock vertex buffer and stuff data into cloth points
char *Vertices;
Mesh->LockVertexBuffer(0, (void**)&Vertices);
for(i=0;i<m_NumPoints;i++) {

   // Get pointer to vertex coordinates
   sClothVertexPos *Vertex = (sClothVertexPos*)Vertices;

   // Store position, velocity, force, and mass
   m_Points[i].m_vecOriginalPos = Vertex->vecPos;
   m_Points[i].m_vecPos = m_Points[i].m_vecOriginalPos;
   m_Points[i].m_Mass = 1.0f;
   m_Points[i].m_OneOverMass = 1.0f;

   // Setup point's states
   m_Points[i].m_vecVelocity = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
   m_Points[i].m_vecForce = D3DXVECTOR3(0.0f, 0.0f, 0.0f);

   // Go to next vertex
   Vertices += m_VertexStride;
}
Mesh->UnlockVertexBuffer();

// Build list of springs from face vertices

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

   // Get vertices that construct a face
   DWORD Vertex1 = m_Faces[i*3];
   DWORD Vertex2 = m_Faces[i*3+1];
   DWORD Vertex3 = m_Faces[i*3+2];

   // Add springs from 1->2, 2->3, and 3->1
   AddSpring(Vertex1, Vertex2);
```

```
      AddSpring(Vertex2, Vertex3);
      AddSpring(Vertex3, Vertex1);
   }

   // Parse cloth masses and springs from file
   if(PointSpringXFile)
      Parse(PointSpringXFile);

   return TRUE;
}
```

The `Create` function starts by freeing a prior mesh via a call to `Free`. From there, it calls `D3DXGetFVFVertexSize` to compute the size of a single vertex's data, and then continues by getting the number of faces in the mesh and allocating an array to contain the indices. At that point, the face indices are read into that array, and an array of cloth points is allocated.

Locking the vertex buffer, you continue by stuffing each vertex's coordinates into the matching cloth point object and setting the point's mass, velocity, force, and 1/mass values. After you go through each vertex and build the cloth point data, you finish by creating the springs and parsing your .X file. (You create the springs by taking the three indices that define each face and calling `AddSpring` three times to join each of the three points.)

The next important function I want to point out is `SetForces`, which goes through each point in the cloth and sets the appropriate forces in preparation for integration.

```
void cClothMesh::SetForces(float LinearDamping,
                           D3DXVECTOR3 *vecGravity,
                           D3DXVECTOR3 *vecWind,
                           D3DXMATRIX *matTransform,
                           BOOL TransformAllPoints)
{
   DWORD i;

   // Error checking
   if(!m_NumPoints || m_Points == NULL)
      return;
```

The `SetForces` function starts with the prototype and a few lines of code that ensure you have some cloth points with which to work. The parameters to `SetForces` include the linear damping amount to apply (set as a negative value, such as −0.05f), a vector that represents the direction and magnitude of the gravity and wind forces, a transformation matrix to apply to the points, and a flag that determines which points to transform.

You've already read about the linear damping and force vectors (for gravity and wind), but up to this point I haven't said anything about transforming your points' coordinates. What good would your cloth mesh be if you couldn't move it through your 3D world? By passing a transformation matrix to `SetForces`, you transform certain points before their forces are calculated.

Which points are transformed, you ask? That depends on the `TransformAllPoints` flag. If you set `TransformAllPoints` to `TRUE`, which you should do the very first time you position and orient your cloth mesh in the 3D world, then all the cloth's points are affected by the transformation matrix. Setting `TransformAllPoints` to `FALSE` means that only those points with zero mass are affected (meaning that those points with mass need to have forces applied to catch up to the transformed points).

You can break the `SetForces` function into three major code bits. The first bit prepares the points by clearing out their force vectors, transforming the points based on the supplied transformation matrix, applying the gravity force, and applying the linear damping force.

```
// Clear forces, apply transformation, set gravity,
// and apply linear damping
for(i=0;i<m_NumPoints;i++) {

   // Clear force
   m_Points[i].m_vecForce = D3DXVECTOR3(0.0f, 0.0f, 0.0f);

   // Move point using transformation if specified
   if(matTransform && (TransformAllPoints == TRUE || \
      m_Points[i].m_Mass == 0.0f)) {

      D3DXVec3TransformCoord(&m_Points[i].m_vecPos, \
                             &m_Points[i].m_vecOriginalPos, \
                                     matTransform);
   }

   // Only apply gravity and linear damping to points w/mass
   if(m_Points[i].m_Mass != 0.0f) {

      // Apply gravity if specified
      if(vecGravity != NULL)
         m_Points[i].m_vecForce += (*vecGravity) * \
                                      m_Points[i].m_Mass;

      // Apply linear damping
      m_Points[i].m_vecForce += (m_Points[i].m_vecVelocity * \

   }
}
```

I explained how to calculate a linear damping force and gravity force earlier in this chapter, so I won't bore you with the details again. As I mentioned, there's also some code in there that transforms the cloth points based on a transformation matrix you supply. The next code bit applies wind forces to your cloth points using the techniques shown earlier in this chapter.

```
// Apply wind
if(vecWind != NULL && m_NumFaces) {

   // Go through each face and apply wind vector to
   // vertex on all faces
   for(i=0;i<m_NumFaces;i++) {

      // Get three vertices that construct face
      DWORD Vertex1 = m_Faces[i*3];
      DWORD Vertex2 = m_Faces[i*3+1];
      DWORD Vertex3 = m_Faces[i*3+2];

      // Calculate face's normal
      D3DXVECTOR3 vecV12 = m_Points[Vertex2].m_vecPos - \
                           m_Points[Vertex1].m_vecPos;
      D3DXVECTOR3 vecV13 = m_Points[Vertex3].m_vecPos - \
                           m_Points[Vertex1].m_vecPos;
      D3DXVECTOR3 vecNormal;
      D3DXVec3Cross(&vecNormal, &vecV12, &vecV13);
      D3DXVec3Normalize(&vecNormal, &vecNormal);
```

```
         // Get dot product between normal and wind
         float Dot = D3DXVec3Dot(&vecNormal, vecWind);

         // Amplify normal by dot product
         vecNormal *= Dot;

         // Apply normal to point's force vector
         m_Points[Vertex1].m_vecForce += vecNormal;
         m_Points[Vertex2].m_vecForce += vecNormal;
         m_Points[Vertex3].m_vecForce += vecNormal;
      }
   }
```

Last, and most important, comes the third code bit that applies the springs' forces.

```
   // Process springs
   cClothSpring *Spring = m_Springs;
   while(Spring) {

      // Get the current spring vector
      D3DXVECTOR3 vecSpring;
      vecSpring = m_Points[Spring->m_Point2].m_vecPos - \
                  m_Points[Spring->m_Point1].m_vecPos;

      // Get the current length of the spring
      float SpringLength = D3DXVec3Length(&vecSpring);

      // Get the relative velocity of the points
      D3DXVECTOR3 vecVelocity;
      vecVelocity = m_Points[Spring->m_Point2].m_vecVelocity - \
                    m_Points[Spring->m_Point1].m_vecVelocity;
      float Velocity = D3DXVec3Dot(&vecVelocity, &vecSpring) / \
                           SpringLength;

      // Calculate force scalars
      float SpringForce = Spring->m_Ks * (SpringLength - \
                                      Spring->m_RestingLength);
      float DampingForce = Spring->m_Kd * Velocity;

      // Normalize the spring
      vecSpring /= SpringLength;

      // Calculate force vector
      D3DXVECTOR3 vecForce = (SpringForce + DampingForce) * \
                                                vecSpring;
      // Apply force to vectors
      if(m_Points[Spring->m_Point1].m_Mass != 0.0f)
         m_Points[Spring->m_Point1].m_vecForce += vecForce;

      if(m_Points[Spring->m_Point2].m_Mass != 0.0f)
         m_Points[Spring->m_Point2].m_vecForce -= vecForce;

      // Go to next spring
      Spring = Spring->m_Next;
   }
}
```

Again, there's really nothing here that you haven't already seen. After you've called `SetForces`, it is time to process the forces and move your cloth's points based on their accumulated forces.

```
void cClothMesh::ProcessForces(float Elapsed)
{
      // Error checking
      if(!m_NumPoints || !m_Points)
         return;

      // Resolve forces on points
      for(DWORD i=0;i<m_NumPoints;i++) {

         // Points w/0 mass don't move
         if(m_Points[i].m_Mass != 0.0f) {

         // Update velocity
         m_Points[i].m_vecVelocity += (Elapsed * \
                                   m_Points[i].m_OneOverMass * \
                                   m_Points[i].m_vecForce);

         // Update position
         m_Points[i].m_vecPos += (Elapsed *
                                  m_Points[i].m_vecVelocity);
      }
   }
}
```

Short and sweet, the `ProcessForces` functions get right to the point by iterating the list of cloth points, and for each point, linearly integrating the velocity and position based on the points' forces and amount of time elapsed (which you specify in the call to `ProcessForces`). Note that the `Elapsed` parameter represents the number of seconds to integrate. If you want to specify milliseconds, you need to specify a fractional value, as I mentioned earlier in this chapter.

How about some code to show you how to use the `cClothMesh` class in your own project? To start, make sure you have an `ID3DXMesh` object to work with, and then call `Create` to generate the necessary cloth data.

Note You shouldn't process (integrate) more than 10−20 milliseconds at a time. If you want to process more than 20 milliseconds at a time, you should call `SetForces` and `ProcessForces` multiple times, each time specifying a small time value. For example, you can call `ProcessForces` twice, specifying 20 milliseconds each time, to process a total of 40 milliseconds of time.

```
// pMesh = pre−loaded ID3DXMesh object
cClothMesh ClothMesh;
// Generate cloth data by calling
Create ClothMesh.Create(pMesh);
```

Now that you have created the cloth mesh data, you can use the class object to simulate the motion of your cloth mesh each frame of your game. Suppose you have a function that is called each frame your game is updated. In this function, you want to track the number of milliseconds that have passed since the last update. This elapsed time is used to integrate the cloth's points. Once the forces have been set and the cloth has been updated, you can rebuild the mesh and render it.

```
void FrameUpdate()
{
   static DWORD LastTime = timeGetTime()−1;
   DWORD ThisTime = timeGetTime();
   DWORD Elapsed;

   // Calculate elapsed time
   Elapsed = ThisTime − LastTime;
   LastTime = ThisTime;

   // Set the gravity and wind vectors
   D3DXVECTOR3 vecGravity = D3DXVECTOR3(0.0f, −9.8f, 0.0f);
   D3DXVECTOR3 vecWind = D3DXVECTOR3(0.0f, 0.0f, 1.0f);

   // Set the cloth's forces
   ClothMesh.SetForces(−0.05f,&vecGravity,&vecWind,NULL,FALSE);

   // Process the cloth's forces based on elapsed time
   // Make sure to specify time in milliseconds − divide
   // the elapsed time by 1000 to do this
   ClothMesh.ProcessForces((float)Elapsed / 1000.0f);

   // Process any collisions if you have them

   // Rebuild the mesh
   ClothMesh.RebuildMesh(pMesh);

   // Render the cloth mesh using preferred methods
}
```

To help you get comfortable using the `cClothMesh`, you can fire up the accompanying demo from the CD−ROM. The demo is heavily commented, and it shows off the cloth mesh class' capabilities. Enjoy!

Using Soft Body Meshes

Soft body meshes are becoming increasingly popular in today's games. They allow your objects to appear soft, stretchy, and/or rubbery. The objects can bend, twist, bounce, and deform in myriad manners, only to revert to their original shapes in mere moments.

What can soft body meshes do for your game? Well, take a look at a current game that uses soft body meshes. Interplay's *Baldur's Gate: Dark Alliance* uses soft body meshes in their character animations to give the female characters that extra...um...bounce in their step.

You can also use soft body meshes to simulate hair on a character. As a character's hair bobs, it slowly reverts to its default shape over time. Your game's maps and levels can also benefit from soft body meshes. Imagine having a level in your game that bends and twists in response to the game character's actions, as if the character were walking on top of a hot−air balloon. Awesome!

I know you're wondering why I bunched cloth simulation and soft body mesh animation together in one chapter. The answer is simple−soft body meshes are cloth meshes! Yep, you heard me right−soft body meshes are almost identical to cloth meshes, with one major exception. Soft body meshes will always revert to their original shapes over time, instead of just hanging limp like pieces of cloth.

Reverting a Soft Body Mesh

Unlike its cloth mesh cousins, a soft body mesh will slowly regain its shape over time. If you've been paying attention up to this point, you realize that reverting the soft body mesh is as easy as resolving the forces needed to return the mesh's vertices to their original coordinates. So on top of resolving a mesh's springs, you need to resolve its points.

Unlike the forces that your cloth's springs exert, your soft body mesh instead directly modifies each point's position and velocity vectors. By iterating through each point in the mesh, you can calculate a spring vector that represents the distance from the point's current and initial positions.

Remember when you saved each point's initial position? By applying a spring constant to the distance between the point's current and initial positions, you determine the force to add to each point's position and velocity vectors.

Without further ado, here's the code to calculate the force required to revert each point in the mesh:

```
// SpringStiffness = float value w/stiffness to revert points
for(DWORD i=0;i<NumPoints;i++) {
   // Get difference vector between current initial position
   D3DXVECTOR3 vecDiff = ClothPoints[i].m_vecInitialPos − \
                         ClothPoints[i].m_vecPos;

// Apply a stiffness value to spring vector
   vecDiff *= SpringStiffness;

   // Apply spring vector to position and velocity vectors
   ClothPoints[i].m_vecPos += vecDiff;
   ClothPoints[i].vecVelocity += vecDiff;
}
```

Quick and simple, the preceding code calculates the difference vector from the current and initial positions of a cloth point. This difference vector is then scaled by a spring stiffness value you specify. Unlike your point's spring earlier in this chapter, this soft body spring stiffness value should remain small−say from 0.05 to 0.4. The stiffness value all depends on how much time you specify in your call to the base class's `UpdateForces` function. I personally like using a stiffness value of 0.2 for each 20 to 30 milliseconds processed.

Since we're talking about using the same code for the soft body mesh simulation as we used for the cloth simulation, you can add the preceding code that reverts the mesh's points to their initial positions into your function that resolves a cloth's forces and moves the points.

Creating a Soft Body Mesh Class

The soft body mesh class is identical to the cloth mesh class in every detail, except that in the soft body mesh class you have one extra function that reverts the mesh to its original shape. Because you are only adding that one function, you can skip the majority of the soft body mesh class declaration and get right into the function to revert the soft body mesh.

```
// Derive a soft body mesh class
class cSoftbodyMesh : public cClothMesh
{
  public:
     void Revert(float Stiffness, D3DXMATRIX *matTransform)
     {
        // Error checking
        if(!m_NumPoints || m_Points == NULL)
           return;

         // Process softbody forces (revert shape)
         for(DWORD i=0;i<m_NumPoints;i++) {
           // Only process points that can move
           if(m_Points[i].m_Mass != 0.0f) {

              // Transform original coordinates if needed
              D3DXVECTOR3 vecPos = m_Points[i].m_vecOriginalPos;
              if(matTransform)
                D3DXVec3TransformCoord(&vecPos, &vecPos, matTransform);

               // Create a spring vector from original position
               // of point (transformed) to its current position
               D3DXVECTOR3 vecSpring = vecPos − m_Points[i].m_vecPos;

               // Scale spring by stiffness value
               vecSpring *= Stiffness;

               // Directly modify velocity and position
               m_Points[i].m_vecVelocity += vecSpring;
               m_Points[i].m_vecPos += vecSpring;
          }
       }
    }
};
```

As you can see from the previous `cSoftbodyMesh` class declaration, the Revert function goes through each point in your cloth, moves it toward its initial position, and adjusts the point's velocity accordingly. You'll notice that you can specify the stiffness value in the call to `Revert`, as well as specifying a transformation matrix that is applied to the points' initial positions before the spring vector is computed. This allows you to move the mesh around and allow the points to catch up to their proper positions.

Using the same techniques as those in your cloth mesh class, you can work with the soft body mesh class. Loading, setting forces, rendering−it's all the same for both classes. The only difference is that you need to call `cSoftBodyMesh::Revert` directly after calling `cSoftBodyMesh::Resolve`. Check out the soft body mesh demo program included on the CD−ROM to see how to use `cSoftBodyMesh` in your own projects.

Check Out the Demos

Whew, this chapter was a long one, but the information contained here sure made it a worthwhile read! Now that you've seen how easy it is to work with cloth simulation, I'm sure you're dying to get to work putting these newly discovered techniques into your own game projects. To help you understand how to use cloth meshes and soft body meshes in your own projects, I have included two sample programs with this book.

The first project (ClothMesh), shown in Figure 13.8, demonstrates one subtle use for cloth−a flapping cape on a superhero.

Figure 13.8: *Our superhero flies through the air, his cape fluttering about him.*

The second project included with this book, Softbody, demonstrates how to use soft body meshes to give characters extra bounce in their step. Check out the demo to see what I'm talking about, or consult Figure 13.9.

Figure 13.9: *Karate class is in session! A soft body mesh gives extra bounce to this master's attacks.*

Programs on the CD

The cloth mesh and soft body mesh classes in this chapter are contained in two project files located in the Chapter 13 directory of this book's CD−ROM. These two projects are

- **ClothMesh.** This program demonstrates the use of the cloth mesh class developed in this chapter. It is located at \BookCode\Chap13\ClothMesh.
- **SoftBody.** This demo shows how you can use soft body meshes and the soft body mesh class in your own projects. It is located at \BookCode\Chap13\SoftBody.

Chapter 14: Using Animated Textures

Three−dimensional animation tends toward the manipulation of vertices, polygons, and meshes. Truthfully, your complete animation system can revolve around such manipulations, but you'll miss out on using one of the coolest animation techniques available today−texture animation.

With texture animation, your worlds come alive in ways you can't imagine. Using everything from basic texture transformations to advanced media playback techniques, you can recreate some awesome effects such as dynamic level backdrops, flowing water, and in−game cinematic sequences.

Using Texture Animation in Your Project

Typical animation consists of the manipulation of vertices in your 3D meshes. For the most part, the manipulation of those vertices is enough for your game. But what about animating the polygon surfaces? Maybe you want to change the appearance of a polygon over time, play a movie on the surface of the polygons, or just smoothly scroll a texture across the polygon's surfaces. I'm talking about making those pretty polygons of yours dance! I'm talking about *texture animation*.

With texture animation, you work on a polygon level (by manipulating the image source or texture coordinates a polygon uses) as opposed to working with vertices, as you do in the typical animation I mentioned. For this book, I chose two of the most popular texture animation techniques used today−texture transformations and video media texture animation.

Let's start with the easier of the two and take a look at how you can use texture transformations in your game project.

Working with Texture Transformations

One of the most underrated (and quite possibly the easiest) forms of texture animation is texture transformation. Much like world transformations alter a model's vertex coordinate data before it is drawn, texture transformations alter a model's texture coordinate data before it is rendered.

What possible use is there for texture transformations? Suppose you need an animation for a few polygons. For example, suppose you want the appearance of water gushing forth from a waterfall and rolling lazily down a stream. Can you achieve this effect using conventional vertex deformation animation? Not easily, I assure you.

Using texture animation, you can smoothly scroll a texture of water across the polygons that make up your waterfall and stream. Without any changes to the mesh, your scene comes alive! Scrolling textures aren't all that you can do, though−you can also rotate your textures. Imagine your water texture rolling and spinning all at once! The secret to using texture transformations is...well, the transformations.

Creating a Texture Transformation

Unlike their 3D equivalents, texture transformations are two−dimensional and use a 3x3 transformation matrix. You no longer have to do anything with translation along the z−axis and rotation along the x and y axes. All you're left with is x/y translation (with the y−axis flipped, negative going upward) and z−rotation. Don't fret, because those are good enough for what you need!

The easiest texture transformation you can perform is translation. Direct3D texture translations are specified in values from 0 to 1, much like they are assigned as texture coordinates. For example, if your texture is 256 pixels and you want to translate right by 64 pixels, you specify a translation value of 0.25 (which is one−fourth the texture's size, or 64 pixels).

In their default state, textures wrap around whenever they are transformed. That means if you translate a texture 64 pixels to the right, the last 64 columns of pixels will wrap around to the left side of the texture. This texture wrapping is perfect, so we'll stick with it.

Note DirectX bounds−checks any value you specify for a texture translation that is greater than 1 or less than 0 to fall in the 0 to 1 range−there's no extra work on your part to perform this action. That way, you can continuously increase or decrease the translation's values, and rest assured, they will always be converted to a value from 0 to 1.

To make things easy, you can use the `D3DX` matrix objects to construct your texture transformations. I know I said that texture transformations use a 3x3 matrix, but if you manage to stick to using only `x/y` translations and z−rotations, you can work a little magic and make `D3DX`'s 4x4 matrices work for you, as you'll see in the upcoming "Setting Texture Transformation Matrices" section.

Knowing you can use the `D3DX` matrix object, you can create a translation matrix like this:

```
// D3DXMatrixTranslation prototype from DX SDK
D3DXMATRIX *D3DXMatrixTranslation(D3DXMATRIX *pOut,
                                  FLOAT x, FLOAT y, FLOAT z);

D3DXMATRIX matTranslation;
D3DXMatrixTranslation(&matTranslation, 0.5f, 0.5f, 0.0f);
```

This code, while showing the `D3DXMatrixTranslation` function prototype you'll be using, demonstrates setting up a translation transformation that will scroll a texture left and up by half the width and height of the texture (in pixels). Notice that the z−translation is left at 0.0, as it should be for all texture transformations.

You'll remember that I said translations are dependent on the size of the texture. Even though the values you specify represent a percentage of the texture's dimensions to scroll (with 0.0 being 0% and 1.0 being 100%), you need to work on the pixel level if you need exact pixel precision. For example, if you were to apply the transformation I just mentioned (a value of 0.5, meaning half the size of the texture, for the x/y axes) to a 256x128 texture, it would scroll left 128 pixels and up 64 pixels. As long as you remember how translation values work, you'll be fine.

For texture rotation transformations, you can use the `D3DXMatrixRotationZ` function. Remember that a texture can only rotate along the `z−axis`. As long as you keep to the `z−axis`, you can use a 4x4 matrix. Here's an example of using the `D3DXMatrixRotationZ` function to rotate a texture by 1.57 radians.

```
// D3DXMatrixRotationZ function prototype
D3DXMATRIX *D3DXMatrixRotationZ(D3DXMATRIX *pOut, FLOAT Angle);

D3DXMATRIX matRotation;
D3DXMatrixRotationZ(&matRotation, 1.57f);
```

Rotation occurs around the origin of the texture, which happens to be the top−left pixel. If you want your textures to rotate around another point, you have to transform the texture, rotate it, and then transform it back to its original position. Here's an example of rotating a texture around its center by 0.47 radians:

```
D3DXMATRIX matTrans1, matTrans2, matRotation;
D3DXMatrixTranslation(&matTrans1, −0.5f, −0.5f, 0.0f);
D3DXMatrixTranslation(&matTrans2, 0.5f, 0.5f, 0.0f);
D3DXMatrixRotationZ(&matRotation, 0.47f);

// Combine matrices into a single transformation
D3DXMATRIX matTransformation;
matTransformation = matTrans1 * matRotation * matTrans2;
```

As you can see from this example code, you can combine any number of matrices to come up with your final transformation matrix. Once you've got your final matrix, it's time to hand it to Direct3D and render your textures.

Setting Texture Transformation Matrices

Now that you've got a valid `D3DXMATRIX` set up with all of the transformations you want to apply, you can render the transformed texture. First, however, you need to convert the 4x4 matrix to a 3x3 matrix that Direct3D uses for texture transformations. A small function like the following one will convert the matrix for you:

```
void Matrix4x4To3x3(D3DXMATRIX *matOut, D3DXMATRIX *matIn)
{
  matOut−>_11 = matIn−>_11; // Copy over 1st row
  matOut−>_12 = matIn−>_12;
  matOut−>_13 = matIn−>_13;
  matOut−>_14 = 0.0f;

  matOut−>_21 = matIn−>_21; // Copy over 2nd row
  matOut−>_22 = matIn−>_22;
  matOut−>_23 = matIn−>_23;
  matOut−>_24 = 0.0f;

  matOut−>_31 = matIn−>_41; // Copy bottom row
  matOut−>_32 = matIn−>_42; // used for translation
  matOut−>_33 = matIn−>_43;
  matOut−>_34 = 0.0f;

  matOut−>_41 = 0.0f; // Clear the bottom row
  matOut−>_42 = 0.0f;
  matOut−>_43 = 0.0f;
  matOut−>_44 = 1.0f;
}
```

Calling the `Matrix4x4To3x3` function is as simple as calling any `D3DX` matrix function. All you need to do is provide a source and destination matrix pointer, as I have done here. (Note that the source and destination matrices can be the same.)

```
// Convert matrix to a 3x3 matrix
Matrix4x4To3x3(&matTexture, &matTexture);
```

Once you've passed your 4x4 texture transformation matrix to `Matrix4x4` To`3x3`, you can set the resulting 3x3 texture transformation matrix using `IDirect3DDevice9::SetTransform`, specifying the `D3DTS_TEXTURE0` flag in the call.

```
// Set transformation in Direct3D pipeline
pDevice−> SetTransform(D3DTS_TEXTURE0, &matTexture);
```

At this point, Direct3D is almost ready to use your texture transformation! The only thing left to do is tell Direct3D to process two−dimensional texture coordinates in its texture transformation calculations. You can accomplish this with the following call to `IDirect3DDevice9::SetTextureStageState:`

```
pDevice-> SetTextureStageState(0,
   D3DTSS_TEXTURETRANSFORMFLAGS,
   D3DTTFF_COUNT2);
```

Now you're cooking! Every texture (from stage 0) you render from this point will have the transformation applied to it. When you're finished with the transformation, you can disable it by calling `SetTextureStageState` again, this time setting `D3DTSS_TEXTURETRANSFORMFLAGS` to `D3DTTFF_DISABLE`.

Note The code shown here only allows you to use texture transformations in stage 0. If you are using textures in another texture stage, simply replace the 0 with the stage number you are using.

```
pDevice->SetTextureStageState(0,
                                    D3DTSS_TEXTURETRANSFORMFLAGS,
                                    D3DTTFF_DISABLE);
```

Using Texture Transformations in Your Project

As you can see, using texture coordinate transformations is pretty straightforward and simple. The hardest part is keeping track of the various transformations. I recommend that you create a manager of sorts to keep track of each texture's transformation. (Maybe just maintaining an array of translation and rotation values will work for you.)

For an example of how you can actually use texture transformations in your project, you might want to check out the texture transformation demo located on the CD−ROM. (Look under the Chapter 14 subdirectory or consult the end of this chapter for more information.) The demo shows you how to create the effect of a flowing waterfall using simple translation transformations. Let the demo serve as your starting point into the world of texture transformation animation!

Using Video Media Files for Textures

Textures, although great for enhancing the appearance of your 3D graphics, are merely bland photo−stills. Not even the use of texture transformations can do certain textures justice. What you need is the ability to apply a video sampling to the surface of your polygons.

That's right! You can actually play a video media file on the surface of your 3D model, which means that you can achieve inconceivable effects! Imagine recording some live−action footage and playing that video back in your game. For example, a sports game might have crowds of fans cheering on their favorite team. You could paint this live−action footage onto the surfaces of your empty seats, thus filling the stadium with thousands of adoring fans. Talk about realism!

Don't be limited by just using video media for textures, however. There's so much more you can do with this technique. How about in−game cinematic sequences? Simply apply a full−screen polygon that is textured with your video sequence. What's the catch? Nothing−you're using the same techniques you used to texture your game character's facial mesh!

If the texture transformations got you riled up, I'm sure the use of video media textures has really got you going! How does it work, you ask? It all starts with Microsoft's DirectShow.

Importing Video with DirectShow

DirectShow is the component of DirectX that you rely on to control video playback. It is an extremely useful set of components that is capable of working with many different types of media including various audio formats, such as .mp3 and .wma, and numerous video formats, such as .mpg, .avi, and even DVD−encoded files!

Since you're only concerned with video media formation, you can safely ignore most of DirectShow's components and stick to those that deal with video. Hmm−now that I think about it, the easiest way to deal with video media files is to create a custom filter that processes the video data that DirectShow can import for you. Take a look at how you can use filters for texture animation.

Note Those of you who have taken the time to explore the DirectX SDK might have come upon the Texture3D demo program that demonstrates how to use a single video media file to texture−map your polygons. In this chapter, I expand on that demo by developing a way to use an unlimited number of video media files in your 3D scenes.

Using Filters in DirectShow

DirectShow works with media formats through the use of filters. Each filter is responsible for encoding or decoding media data as it is written to or read from a file. Filters can be strung together, with one filter handling the decoding of one type of media data and the next filter processing the media data and displaying it to the viewer (see Figure 14.1).

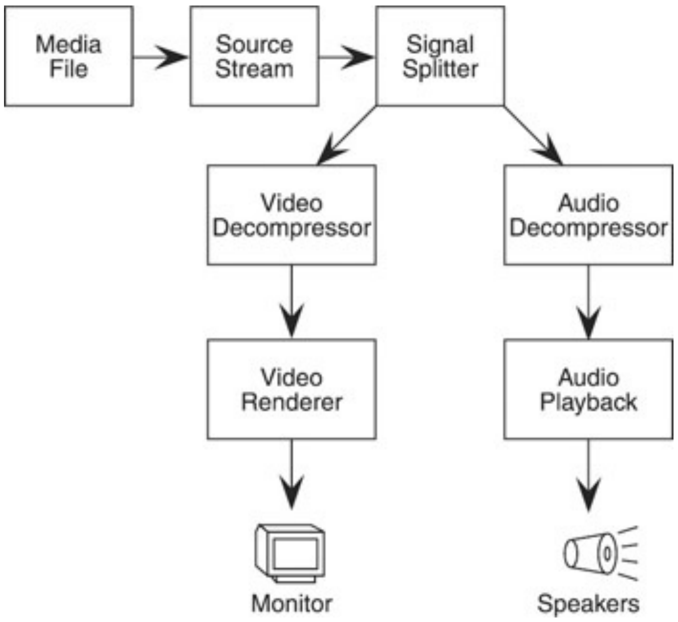


Figure 14.1: *A media file might pass through various filters to let viewers see and hear its contents.*
You begin by creating your own DirectShow filter and inserting it into the flow of the video media decoding. Whenever you tell DirectShow to decode a video media file, your filter will kick in and pass the video data straight to a texture surface you use to paint your polygons. Sounds simple, doesn't it?

The problem is, at last count DirectShow uses more than 70 filters and 70 interfaces! Wow−that's enough to make even the best programmers cringe! What's a programmer like you or I supposed to do against these overwhelming interfaces? I'll tell you what you can doyou can use the Base Classes.

Using the Base Classes

Here comes Microsoft to the rescue! Knowing that working directly with the video decoders and various DirectShow interfaces was a bit daunting, Microsoft constructed a series of classes you can use to aid in the development of your filters. These classes, called the DirectShow Base Classes, are located in your DirectX SDK installation directory at \Samples\Multimedia\DirectShow\BaseClasses.

Tip To switch configurations inside the Visual C/C++ compiler, select Build, Set Active Configuration from the main menu. In the SetActive Project Configuration dialog box, select the desired configuration (either debug or release) from the Project Configurations list and click the OK button.

You need to compile the Base Classes to use them in your projects. Start Visual C/C++ and open the BaseClasses project that is located in the DirectX SDK installation at \Samples\Multimedia\DirectShow\BaseClasses\BaseClasses.dsw. Compile the Base Classes project using both the debug and release configurations.

After you complete the compilation (make sure to compile in both configurations!), you need to copy the resulting two library files (\debug\strmbasd.lib for the debug configuration and \release\strmbase.lib for the release configuration) and all of the .h files located in the BaseClasses directory into your project's directory.

In your project's Settings/Link dialog box, add strmbasd.lib or strmbase.lib (debug or release, respectively, depending on your configuration) to your project's Object/ Library Modules list. Also, whenever you use DirectShow in your projects, make sure to include the streams.h header file (from the Base Classes directory), as well as the dshow.h header file.

Note If you want to avoid putting the Base Classes' files in your project's directory (to save on disk space, or because you don't want a hundred or so files residing there), simply add the Base Classes as an include directory within the compiler. Make sure to do the same for the library directories; rather, just link directly to the library from your compiler's link settings.

You're now ready to rock! To start importing video data, you first need to create your own filter.

Creating a Custom Filter

In the previous section, you learned how DirectShow uses filters to process streams of data that are being decoded or encoded. In this case, those streams represent frames of video. DirectShow comes with a number of video−capable filters for you to use in your own projects, but unfortunately, none of those filters are really what you need here. You need to create your own custom DirectShow filter that takes incoming video data and "pastes" it onto a texture surface, thus creating a video texture animation.

Creating your own custom filter really isn't difficult. The trick is to derive your custom filter from DirectShow's base filter class, assign it a unique identification number, and override the appropriate functions to process incoming video data. That doesn't sound so difficult, does it? The following sections detail the steps to create your custom filter.

Deriving the Filter Class

The first step to creating your own custom DirectShow filter is to derive your own filter class from DirectShow's Base Classes. You're going to derive `CBaseVideoRenderer`, which is responsible for processing video samples from an input stream. Go ahead and derive a class (called `cTextureFilter`) that you'll use for your filter.

```
class cTextureFilter : public CBaseVideoRenderer
{
```

Aside from the functions that you'll use (which you'll read about in a moment), you need to declare a few member variables to store the pointer to your Direct3D device (`IDirect3DDevice9`), texture object (`IDirect3DTexture9`), texture format (`D3DFORMAT`), and the video source's image width, height, and pitch. You can define these six variables in the `cTextureFilter` class.

```
public:
  IDirect3DDevice9 *m_pD3DDevice; // 3D device interface
  IDirect3DTexture9 *m_pTexture; // Texture object
  D3DFORMAT m_Format; // Format of texture
  LONG m_lVideoWidth; // Pixel width of video
  LONG m_lVideoHeight; // Pixel height of video
  LONG m_lVideoPitch; // Video surface pitch
```

The first three variables, `m_pD3DDevice, m_pTexture`, and `m_Format` are pretty standard fare for storing a texture. First there's the 3D device interface you are using in your project, then there's the texture object that will contain the animated texture, and finally there's the texture's color format descriptor.

The last three variables, `m_lVideoWidth, m_lVideoHeight,` and `m_lVideoPitch,` describe the dimensions of the video source. DirectShow imports video data by creating a bitmap surface in memory and streaming video data onto that surface. That surface has its own unique height (`m_lVideoHeight`), width (`m_lVideoWidth`), and surface pitch (`m_lVideoPitch`) that determine how many bytes a row of video data uses.

Give the variables a rest for the moment. (You'll get back to those later in the "Working with the Custom Filter" section.) For now, read on to see how to assign a unique identification number to your texture filter.

Defining a Unique Identification Number

For DirectShow to distinguish your filter from all the others, you need to assign it a unique identification number. This unique number (a class ID number, to be exact) is represented by a UUID that you can define using the following code in your source file. (Typically, you would include this code at the beginning of your custom filter source code file.)

Note The filter's UUID can be anything; running Microsoft's guidgen.exe program will give you an acceptable GUID number you can use. I took the liberty of using the UUID shown for all of the examples for this chapter. If you want to see how to create your own GUIDs, refer to Chapter 3.

```
struct __declspec( \
  uuid(";{61DA4980-0487-11d6-9089-00400536B95F}") \
) CLSID_AnimatedTexture;
```

Once you have defined the UUID, you'll only need to use it once inside your custom filter's constructor function. The purpose is to register your UUID with the DirectShow filter system so it knows where to go to find your filter. It is interesting to note that once your filter is set up and registered with the system (a feature you'll read about shortly), it will be in memory at all times. Whenever a media file is loaded, your filter gets a shot at decoding it. For your filter to decode media data, you need to override a few common functions.

Overriding the Essential Functions

Now that you have declared your custom filter class and you've assigned your filter a unique identification number, it is time to define the functions your filter will use. Aside from the typical class constructor, you need to override three essential DirectShow filter functions that are common to all filters.

The first function, `CheckMediaType`, determines whether your filter can handle the incoming media data. Since DirectShow can handle almost any media file, your filter will use `CheckMediaType` to determine whether the incoming media data is actually video data stored in a format you want to use. Take a look at the `CheckMediaType` function prototype.

```
virtual HRESULT CBaseRenderer::CheckMediaType(
  const CMediaType *pmt) PURE;
```

The `CheckMediaType` function only has one parameter−`CMediaType`, the incoming media's data interface. Using the `CMediaType` interface's `FormatType` function, you can query for the type of media the interface contains (video, sound, or something else).

Since you're only interested in video media files, the return code from `CMediaType::FormatType` you'll be looking for is `FORMAT_VideoInfo`. If the media data is indeed video, you need to perform one further check to determine whether it is the proper format. You can accomplish this by checking the GUID value of the video's type and subtype.

Again, the media type you want is video (represented by a `MEDIATYPE_Video` GUID macro), whereas the subtype you want is the `MEDIASUBTYPE_RGB24` color−depth GUID macro (meaning a 24−bit color depth with the color arranged in red, green, and blue chunks).

Once you've verified that you have a valid video stream, you can return a value of `S_OK` from your `CheckMediaType` function to let DirectShow know you can use it. If the media data isn't what you want, you can return a value of `E_FAIL` from `CheckMediaType`.

That sounds like a lot to do to verify a medium's data type, but believe me, it's not. Take a look at the overridden `CheckMediaType` function that you use.

```
HRESULT cTextureFile::CheckMediaType( \
  const CMediaType *pMediaType)
{
  // Only accept video type media
  if(*pMediaType−>FormatType() != FORMAT_VideoInfo)
    return E_INVALIDARG;
  // Make sure media data is using RGB 24−bit color format
  if(IsEqualGUID(*pMediaType−> Type(), MEDIATYPE_Video) && \
    IsEqualGUID(*pMediaType−>Subtype(), MEDIASUBTYPE_RGB24))
    return S_OK;
  return E_FAIL;
}
```

The second function of the three you need to override is `SetMediaType`, which your filter can use to configure the internal variables it has defined and prepare for the incoming media data. Again, this function can reject incoming media data based on its format.

SetMediaType uses only one parameter, const CMediaType, just as the CheckMediaType function did. Inside the SetMediaType function, you want to retrieve the resolution of the video data (width, height, and surface pitch) and store it within the class's variables.

Also, you need to create a valid 32−bit texture surface (which might default to a 16−bit surface if no 24−bit modes are available) to hold the video data as it is being streamed in. After you have created the texture, you need to check its format, which must be 32−bit or 16−bit, and store the color information for later use.

A return value of S_OK from the SetMediaType function signifies that your filter is ready to use the video data; otherwise, you need to return an error value (such as E_FAIL).

You can program the SetMediaType function to retrieve the video's resolution and create the texture surface using the following code. (I'll let the comments point out the relevant parts of the code.)

```
HRESULT cTextureFilter::SetMediaType(const CMediaType *pMediaType)
{
  HRESULT hr;
  VIDEOINFO *pVideoInfo;
  D3DSURFACE_DESC ddsd;

  // Retrieve the size of this media type
  pVideoInfo = (VIDEOINFO *)pMediaType−>Format();
  m_lVideoWidth = pVideoInfo−>bmiHeader.biWidth;
  m_lVideoHeight = abs(pVideoInfo−>bmiHeader.biHeight);
  m_lVideoPitch = (m_lVideoWidth * 3 + 3) & ~(3);

  // Create the texture that maps to this media type
  if(FAILED(hr = D3DXCreateTexture(m_pD3DDevice, \
  m_lVideoWidth, m_lVideoHeight, \
    1, 0, D3DFMT_A8R8G8B8, \
    D3DPOOL_MANAGED, &m_pTexture)))
  return hr;

 // Get texture description and verify settings
 if(FAILED(hr = m_pTexture−>GetLevelDesc(0, &ddsd)))
  return hr;
 m_Format = ddsd.Format;
 if(m_Format != D3DFMT_A8R8G8B8 && m_Format != D3DFMT_A1R5G5B5)
  return VFW_E_TYPE_NOT_ACCEPTED;
 return S_OK;
}
```

The last of the three functions you need to override, DoRenderSample, is the most important. DoRenderSample is called every time your filter needs to process a frame of data from the media file. In this case, the data represents a video clip that needs to be pasted onto a texture surface.

The overridden DoRenderSample function has the following prototype:

```
virtual HRESULT CBaseRenderer::DoRenderSample(
    IMediaSample *pMediaSample);
```

DoRenderSample uses one parameter, IMediaSample, which is an interface that contains the information regarding a single media sample (the bitmap image representing a single frame of video). Your job is to grab a pointer to the media's data and paste it onto your texture surface.

Start by declaring the overridden `DoRenderSample` function and grabbing a pointer to the media's data via the `IMediaSample::GetPointer function`.

```
HRESULT cTextureFilter::DoRenderSample( \
  IMediaSample *pMediaSample)
{
 // Get a pointer to video sample buffer
 BYTE *pSamplePtr;
 pMediaSample−>GetPointer(&pSamplePtr);
```

At this point, you need to lock your texture surface to copy over the pixel data from the video data. You can use your texture object's `LockRect` method to accomplish this.

```
// Lock the texture surface
D3DLOCKED_RECT d3dlr;
if(FAILED(m_pTexture−>LockRect(0, &d3dlr, 0, 0)))
  return E_FAIL;

// Get texture pitch and pointer to texture data
BYTE *pTexturePtr = (BYTE*)d3dlr.pBits;
LONG lTexturePitch = d3dlr.Pitch;
```

Once you have locked the texture surface and retrieved the texture surface's pitch and data pointer, you can begin copying over the video data. First, however, you need to point to the bottom row of the texture because for some odd reason, the video data is stored upside down.

```
// Offset texture to bottom line, since video
// is stored upside down in buffer
pTexturePtr += (lTexturePitch * (m_lVideoHeight−1));
```

Cool, now you're ready to copy over the video data. Remember back when you created the texture surface, the color format of the texture could have been either 32−bit or 16−bit? Now you need to consider the color depth when copying over the video data, because you need to copy it 32 or 16 bits at a time.

The following code starts with a `switch` statement and branches off to a block of code that will copy over the video data using 32 bits of color information per loop (if the color format is 32−bit, of course).

```
// Copy the bits using specified video format
int x, y, SrcPos, DestPos;
switch(m_Format) {
 case D3DFMT_A8R8G8B8: // 32−bit
```

The next bit of code loops through each row of the video sample and copies each pixel over to the texture surface.

```
// Loop through each row, copying bytes as you go
 for(y=0;y<m_lVideoHeight; y++) {

 // Copy each column
 SrcPos = DestPos = 0;
 for(x=0;x<m_lVideoWidth;x++) {
  pTexturePtr[DestPos++] = pSamplePtr[SrcPos++];
  pTexturePtr[DestPos++] = pSamplePtr[SrcPos++];
  pTexturePtr[DestPos++] = pSamplePtr[SrcPos++];
  pTexturePtr[DestPos++] = 0xff;
}
// Move pointers to next line
pSamplePtr += m_lVideoPitch;
pTexturePtr −= lTexturePitch;
}
break;
```

The second `switch` case statement copies the 16−bit color depth video data. This code is basically the same as the 32−bit color pixel code, but this time only 16−bit pixels are copied.

```
case D3DFMT_A1R5G5B5: // 16-bit
  // Loop through each row, copying bytes as you go
for(y=0;y<m_lVideoHeight;y++) {

  // Copy each column
  SrcPos = DestPos = 0;
  for(x=0;x<m_lVideoWidth;x++) {
    *(WORD*)pTexturePtr[DestPos++] = 0x8000 +
            ((pSamplePtr[SrcPos+2] & 0xF8) << 7) +
            ((pSamplePtr[SrcPos+1] & 0xF8) << 2) +
            (pSamplePtr[SrcPos] >> 3);
  SrcPos += 3;
}

// Move pointers to next line
pSamplePtr += m_lVideoPitch;
pTexturePtr −= lTexturePitch;
}
break;
}
```

To wrap up the `DoRenderSample` function, you just need to unlock the texture surface and return a resulting success or error code.

```
// Unlock the Texture
if (FAILED(m_pTexture−>UnlockRect(0)))
  return E_FAIL;

return S_OK;
}
```

DirectShow calls directly each of the three overridden functions you've just written (`CheckMediaType`, `SetMediaType`, and `DoRenderSample`). In other words, you never directly call any of the three functions; rather, you let DirectShow call them whenever it needs them. For that reason, all DirectShow filters are kept resident in memory at all times. This means that once you've created an instance of your filter, you should never delete it from memory. DirectShow will remove filters from memory for you.

Finishing Your Filter Class

Now that your filter class is accepting incoming media data, you only have to write a couple more functions to complete your filter. These functions are the constructor and a function that returns the texture surface's object pointer. Each function is defined as follows (declared as public functions within your `cTextureFilter` class):

```
class cTextureFilter {
  // ... Previous declaration stuff
  public:
```

```
cTextureFilter(IDirect3DDevice9 *pD3DDevice, \
               LPUNKNOWN pUnk = NULL, \
               HRESULT *phr = NULL);
IDirect3DTexture9 *GetTexture();
};
```

The one and only constructor to your `cTextureFilter` class registers your filter with the DirectShow system and declares your 3D device object interface. Don't let the constructor's function prototype scare you. Other than the 3D device object pointer, the constructor takes a pointer to an `IUnknown` object (which you can set to NULL) and a pointer to an `HRESULT` variable you can use to store the success or error result. Here's the code for the `cTextureFilter` class constructor:

```
cTextureFilter::cTextureFilter(IDirect3DDevice9 *pD3DDevice, \
                                               LPUNKNOWN pUnk, HRESULT *phr)
  : CBaseVideoRenderer(__uuidof(CLSID_AnimatedTexture),
   NAME("ANIMATEDTEXTURE"), pUnk, phr)
{
  // Save device pointer
  m_pD3DDevice = pD3DDevice;

  // Return success
  *phr = S_OK;
}
```

To register your filter with DirectShow, you need to call the `CBaseVideoRenderer` constructor function, as shown in your filter's constructor call. `The CBaseVideoRenderer's` constructor takes the UUID of the filter you created, as well as the name you want to assign to your filter (`ANIMATEDTEXTURE`, in this case). The other two variables are merely passed from your filter's constructor function parameters.

Inside the constructor, you only need to save the 3D object pointer to a class member variable (`m_pD3DDevice`) and store a successful return code in the supplied `HRESULT` pointer. Quickly enough, that's the end of your filter class' constructor!

The second (and last) function you need to add to your filter class will return a pointer to your texture surface object. Remember, the texture object was declared in your class definition as `m_Texture`, so the following function code will do the job for you:

```
IDirect3DTexture9 *cTextureFilter::GetTexture()
{
  return m_Texture;
}
```

And with that last function, your filter class is ready to use!

Working with the Custom Filter

All right, things are starting to heat up! At this point, your custom filter is ready to use. The only thing stopping you is you need to create a class that initializes the proper DirectShow objects (including your filter), loads a video media file, and controls playback of the video.

In fact, you'll come to rely on one main DirectShow interface for use with your filter−`IGraphBuilder`. The `IGraphBuilder` interface, called a *graph builder*, builds and maintains a list of filters (called a *filter graph*) that are used during the decoding of a media file. The graph builder will load a media file for you and configure all of the proper filters for processing the media data. Take a closer look at this very important object.

Using a Graph Builder

The `IGraphBuilder` object is much like any other COM interface. To use `IGraphBuilder`, you need to instance it and call `CoCreateInstance` to create the object and retrieve the interface.

```
IGraphBuilder *pGraph;
CoCreateInstance(CLSID_FilterGraph, NULL, \
  CLSCTX_INPROC_SERVER, IID_IGraphBuilder, \
  (void**)&Graph);
```

After you have created the `IGraphBuilder` interface, you can use it to register your filter for use and load a media file. Before you register your filter, however, you must first instance a copy of it that the graph builder will keep in memory during the course of the media playback. You can use the new operator to allocate an instance of your filter class, and you can register the filter instance with the graph builder using the `IGraphBuilder::AddFilter function`.

Tip Before using any COM object, you must first ensure that the COM system is initialized. You can initialize the COM system by inserting the following line of code at the beginning of your application's execution (typically at the beginning of your `WinMain` function):

```
CoInitialize(NULL);
```

When your application is ready to exit, you must de−initialize the COM system using the following line of code:

```
CoUninitialize();
```

Note The `CoCreateInstance` function returns a value of `S_OK` if the function call was successful. Any other return value means an error occurred during the creation of the COM object. Consult the Win32 SDK for information on what each return code signifies.

Here's some code that will allocate the filter and register it with the graph builder:

```
// pD3DDevice = pre−initialized Direct3D device object pointer

// Allocate an instance of the filter
cTextureFilter *pTextureFilter = \
                  new cTextureFilter(pD3DDevice, NULL, &hr);
// Add the filter to the graph builder
IBaseFilter *pFilter = (IBaseFilter*)pTextureFilter;
pGraph−>AddFilter(pFilter, L"ANIMATEDTEXTURE");
```

The graph builder uses the `AddFilter` function to accept a pointer to a filter (cast to an `IBaseFilter` class interface, from which all filters are derived), as well as a name to assign to the filter you are adding. In the previous example of `AddFilter`, you passed a pointer to your texture filter (using the `pFilter` pointer) and called the filter `ANIMATEDTEXTURE`. The filter's name is not used from this point on; it's merely for your own use and for an external filter viewer that examines all filters in use by DirectShow.

After you have registered the filter, you can begin using it by selecting a source video file.

Selecting a Source File

Now that your filter is registered with DirectShow (via the `IGraphBuilder` interface), you must tell DirectShow which file to use for importing video. The `IGraphBuilder::AddSourceFilter` function takes care of setting the source file and creating the appropriate filters for importing the video data.

```
HRESULT IGraphBuilder::AddSourceFilter(
  LPCWSTR lpwstrFileName,
  LPCWSTR lpwstrFilterName,
  IBaseFilter **ppFilter);
```

The `AddSourceFilter` function takes three parameters−the name of the media file to load, the name of the source filter (`lpwstrFilterName`, which you'll read about in a moment), and a pointer (`ppFilter`) to an `IBaseFilter` filter object that you'll use to access the source video filter object.

What's all this talk about a source filter? I know I told you there are a number of filters you can use in the process of decoding a media file. As Figure 14.2 illustrates, the source filter is really a conduit to all of the filters the graph builder is using.

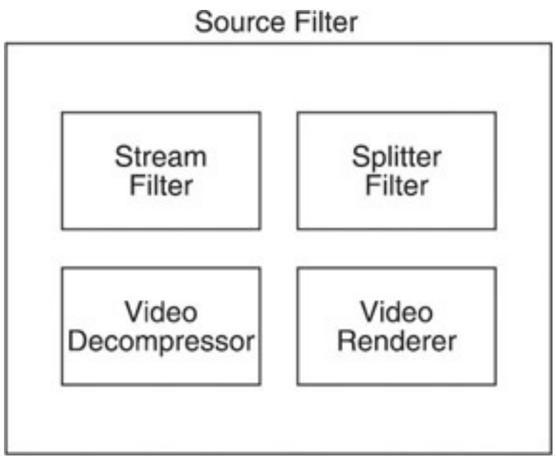


Figure 14.2: *The source filter uses a single interface to represent a collage of filter objects.*

By creating a source filter (using an `IBaseFilter` interface to represent it) and naming it (via the `lpwstrFilterName` parameter), you can quickly access the various filters in the filter graph through the single source filter interface.

Makes sense now, doesn't it? Call `AddSourceFilter` to create your base filter and load the media file. The base filter will be named `SOURCE` (cast as a Unicode character string using the `L""` macro) and requires an instanced `IBaseFilter` interface.

```
// Filename = name of video media filename to use
IBaseFilter *pSrcFilter;

// Convert from ASCII text to Unicode text:
WCHAR wFilename[MAX_PATH];
mbstowcs(wFilename, Filename, MAX_PATH);

// Add source filter to graph
pGraph->AddSourceFilter(wFilename, L"SOURCE", &pSrcFilter);
```

Once you have called `AddSourceFilter`, you will receive a pointer to the source filter. This pointer, `pSrcFilter`, is used to connect the video source's output pin to your texture filter's input pin.

Note You'll notice that I convert the multi−byte `Filename` text string to a wide−character string because the AddSourceFilter functions (like most DirectShow calls) require you to use Unicode characters.

Connecting the Pins

Just what the heck am I talking about−what are these pins and what are they doing here? Each filter has a set of pins, which are sort of like funnels that push the media's data in and out of the filter. The media file's data is fed to the first filter through its In pin. The data is processed by the filter and fed to the next filter in line though the Output pin. This process continues until the media data reaches the end of the line, which in this case is the texture surface you are using. Check out Figure 14.3 to get a better understanding of what I mean.

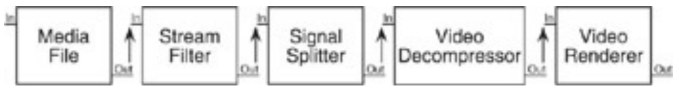


Figure 14.3: *A sample set of four filters is used to grab data from the media file, decode it, and finally render it to the video display*

Each filter must do whatever is necessary to the incoming media data and pass it to the next filter in line. As for your filter, it only waits for the media data and pastes it onto your texture surface. So again, what does all this stuff about pins have to do with you?

The next step to using a video texture is to find your filter's In pin (the pin that accepts incoming data) and the source filter's Output pin (the one that the video data comes out of) and connect them. Thus, your filter will receive all data coming out of the source filter.

To find these two pins (your filter's In pin and the source filter's Output pin), you use the FindPin function.

```
HRESULT IBaseFilter::FindPin(
  LPCWSTR Id, // Name of pin to find
  IPin **ppPin); // Pin interface object
```

I know I told you that you wouldn't mess with too many DirectShow interfaces, but I promise you, there's not much left to do. The FindPin function takes only two parameters−the pin's name that you want to find (either In or Out in this case, both represented in a Unicode character string) and an IPin interface.

The IPin interfaces are not important for you here; you only need them so your graph builder can connect them. Check out the code for finding the pins first, and then move on to connecting them.

```
// Find the input and out pins and connect together
IPin *pFilterPinIn;
IPin *pSourcePinOut;
pFilter->FindPin(L"In", &pFilterPinIn);
pSourceFilter->FindPin(L"Output", &pSourcePinOut);
```

It can't get any easier than that. With one more call, you can connect the pins.

```
pGraph->Connect(pSourcePinOut, pFilterPinIn);
```

At this point, the media file is loaded, and the filters are all connected and ready to rock! The only thing left is to begin playback of the video stream. The problem is the graph builder doesn't have any functions to control playback. What a bummer! Don't fret, though, because you can query the graph builder for just the right interfaces you need to control video−stream playback.

Querying for Interfaces

To control the playback of the video data, you must query the graph builder for a media control interface−`IMediaControl`. The media control interface can play, stop, pause, and resume playback of the video stream.

To determine when playback of the video is complete, you need to query the graph builder for a media event interface−`IMediaEvent`. Last, you want to query for one additional interface to determine the current position, or rather the time, of the video playback.

Use the following code to query for the three interfaces you need:

```
// Instance media object interfaces
IMediaControl *pMediaControl;
IMediaPosition *pMediaPosition;
IMediaEvent *pMediaEvent;

// Query for interfaces
pGraph−>QueryInterface(IID_IMediaControl,   \
    (void **)&pMediaControl);
pGraph−>QueryInterface(IID_IMediaPosition,   \
    (void **)&pMediaPosition);
pGraph−>QueryInterface(IID_IMediaEvent,      \
    (void **)&pMediaEvent);
```

Don't give up; you're almost there! All that's left is to start playback of the video, determine the video's position as it is playing, and watch for various media events to occur.

Controlling Playback and Handling Events

The last step to get your animated texture going is to begin playback of the video source using the `IMediaControl` object you just created. There are two functions of interest in `IMediaControl`−`IMediaControl::Run` and `IMediaControl::Stop`.

The `IMediaControl::Run` function begins playback of the video at the current playback position, which in turn streams the data into your filter and likewise onto your texture surface. The `Run` function takes no parameters and can be used as in the following code:

```
pMediaControl−>Run();
```

Once a video is playing, you can stop it by calling the `IMediaControl::Stop` function, which also uses no parameters. The following code will stop a playing video:

```
pMediaControl−>Stop();
```

The cool thing about the `Run` and `Stop` functions is that you can also use them to pause and resume your video playback. Calling the `Stop` function will pause the video playback; a subsequent call to `Run` will resume playback.

To seek a specific position within the video file (such as the starting position of the video whenever you want to "rewind" it), you turn to the `IMediaPosition` interface. To seek to a specific time in the video file, you call the `IMediaPosition::put_CurrentPosition` function.

```
HRESULT IMediaPosition::put_CurrentPosition(
  REFTIME llTime);
```

The REFTIME parameter of the put_CurrentPosition function is really a float data type in disguise; you set the llTime time to any time value you want to seek in the video stream. (For example, 2.0 means to seek to 2.0 seconds in the stream.) You'll most likely use the put_CurrentPosition function to rewind the video and begin playback at the beginning, much like I have done in this code.

```
// Seek to start of video stream (0 seconds)
pMediaPosition->put_CurrentPosition(0.0f);
```

Last in the trio of media objects that you're using is IMediaEvent, which retrieves any pending events. The only event that you're concerned with is the one that signifies the end of the video, so you can restart the video, process some function to stream in another video, or whatever other option you choose.

To retrieve an event from the IMediaEvent interface, you use the IMediaEvent::GetEvent function.

```
HRESULT IMediaEvent::GetEvent(
long *lEventCode, // Event code
long *lParam1, // 1st event parameter
long *lParam2, // 2nd event parameter
long msTimeout); // Time to wait for event
```

The GetEvent function has four parameters for you to set, with the first three being pointers to long variables and the last one being a long value. After the call to GetEvent, the first three long variables (lEventCode, lParam1, and lParam2) are filled with the current event's information, such as the event code and two parameters used by the event. You can set the fourth variable, msTimeout, to the amount of time you want to wait for an event to occur, or you can set it to 0 to wait indefinitely for an event to occur.

The only event you want to check for is the video ending, which is represented by the EC_COMPLETE macro. You shouldn't wait indefinitely for an event; instead, wait for a few milliseconds and then continue querying for events until none remain. How do you know if no events remain to query? Whenever the GetEvent function returns an error code, you can safely assume there are no events waiting.

Here's some code that continuously queries the media event interface for the current event (breaking when there are no events waiting), as well as a little block of code that handles the EC_COMPLETE event.

```
// Process all waiting events
long lEventCode, lParam1, lParam2;
while(1) {
  // Get the event, waiting 1 milliseconds per event
  if(FAILED(pMediaEvent->GetEvent(&lEventCode, &lParam1, &lParam2, 1)))
    break;

// Handle the end-of-video event by calling a special function
if(lEventCode == EC_COMPLETE)
  EndOfAnimation(); // Any function you want

// Free the event resources
pMediaEvent->FreeEventParams(lEventCode, lParam1, lParam2);
}
```

Hah! I tried to fool you by throwing in a new function call from the IMediaEvent interface−FreeEventParams. Every time you retrieve an event using the GetEvent function, you must follow it up with a call to FreeEventParams so the media event object has a chance to free any resources associated with the event.

And so you have it. You now have the ability to play, stop, pause, resume, and position the playback of your video texture, as well as check for the completion of the video playback (at which point you can restart playback or whatever else your little heart desires).

Creating an Animated Texture Manager

Not that it's very hard to use animated textures (with the help of this book, that is), but your life will be much easier if you take the time to create a manager to handle the filters and textures for you. I've taken the liberty of wrapping all that you've seen about video textures in this chapter into a single class that you can use for your game projects. You'll find this class, called `cAnimationTexture`, on the CD−ROM, under the \BookCode\Chap14\TextureAnim\ directory. The class declaration is defined as follows:

```
class cAnimatedTexture
{
  protected:
    IGraphBuilder     *m_pGraph;         // Filter graph
    IMediaControl     *m_pMediaControl;  // Playback control
    IMediaPosition    *m_pMediaPosition; // Positioning control
    IMediaEvent       *m_pMediaEvent;    // Event control

    IDirect3DDevice9  *m_pD3DDevice;     // 3−D device
    IDirect3DTexture9 *m_pTexture;       // Texture object

  public:
cAnimatedTexture();
~cAnimatedTexture();
// Load and free an animated texture object
BOOL Load(IDirect3DDevice9 *pDevice, char *Filename);
BOOL Free();

// Update the texture and check for looping
BOOL Update();

// Called at end of animation playback
virtual BOOL EndOfAnimation();

// Play and stop functions
BOOL Play();
BOOL Stop();

// Restart animation or go to specific time
BOOL Restart();
BOOL GotoTime(REFTIME Time);

// Return texture object pointer
IDirect3DTexture9 *GetTexture();
};
```

You should recognize most of `cAnimatedTexture`'s class members. There are your ever−faithful DirectShow interfaces that are used for media playback, events, and position information, as well as a couple Direct3D objects used to point to the 3D device in use and a texture surface object that points to your filter's texture surface.

Aside from the class' variables, you have access to 11 class functions. As you would expect, there's the class constructor and destructor, which initialize the class' variables and release the object's resources, respectively. Then there's the `Load` function, which loads a video file and prepares your filter for use. You only need to supply `Load` with the pointer to the `IDirect3DDevice9` object you are using and the file name of the video file to use as a texture, and you're off! The `Load` function will instance a copy of your `cTextureFilter` class and begin using it to load video data for you. When you're done with the `cAnimatedTexture` class object, call the Free function to `free` all used resources and interfaces.

Moving on, there's the `Update` function, which you should call every iteration of your message pump (every frame of your game). The Update function polls the media event object to see whether any messages are waiting; if so, those messages are handled. If the video has completed playback, then the `EndOfAnimation` function is called.

You'll notice that you can override the `EndOfAnimation` function, meaning that you can write your own functions to determine what should be done when the video playback is complete. For instance, you might call on the `Restart` function, which `restarts` the video at the beginning, or you might call `GotoTime`, which takes a `REFTIME` value (a `float` value) of the time to scan to in the video and then begins playing.

Finally, there are three more functions. `Start` will start playback of the video at the current playback position, and `Stop` will stop the video playback. It's possible to pause a video by calling `Stop` and then resume playback by calling `Play`.

Last is the `GetTexture` function, which is identical to your filter's `GetTexture` function; both return the pointer to your texture object's interface, which allows you to set the texture via a call to `IDirect3DDevice9::SetTexture`.

I'm not going to show any code to `cAnimatedTexture` here, since you've pretty much seen it all throughout this chapter. Again, I'll refer you to the source code in the `TextureAnim` project on this book's CD−ROM. To check out what the `cAnimatedTexture` class can do for you, take a look at a quick example that shows you what you can accomplish with animated textures.

Applying Animated Media Textures

Now that you've got your animated−video texture engine working, it's time to put it to the test and create something worthwhile. In this example, you'll create a quad polygon (a polygon with four points) and apply an animated texture to it. To keep things clean, I'll skip the mundane Direct3D initialization code and get right into the example by first creating a vertex buffer that will contain the polygon data.

Note When using the DirectShow filters developed in this chapter, you must specify the `D3DCREATE_MULTITHREADED` flag in your call to `IDirect3D9::CreateDevice`.This lets Direct3D know you're using a multithreaded application (in which the filters are separate threads) and ensures that you need access to those filters for the texture surface data. Also, you should use a multithreaded run−time library (as set in your compiler's settings).

Creating a Vertex Buffer

To use the animated texture, you need to apply it as a regular texture on a polygon (or series of polygons). To demonstrate the animated texture on which we've been building, create a simple quad using two triangles. You can accomplish this using a vertex buffer with four vertices (in a triangle strip).

```
// pDevice = pre−initialized Direct3D Device
typedef struct {
  float x, y, z; // 3D coordinates
  float u, v; // Texture coordinates
} sVertex;
#define VERTEXFVF (D3DFVF_XYZ | D3DFVF_TEX1)
// Define a quad polygon data (two triangles in a strip)
sVertex Verts[4] = {
  { −128.0f, 128.0f, 0.0f, 0.0f, 0.0f },
  { 128.0f, 128.0f, 0.0f, 1.0f, 0.0f },
  { −128.0f, −128.0f, 0.0f, 0.0f, 1.0f },
  { 128.0f, −128.0f, 0.0f, 1.0f, 1.0f }
};
// Instance and create a vertex buffer
IDirect3DVertexBuffer9 *pVertices;
pDevice−>CreateVertexBuffer(sizeof(Verts), \
                            D3DUSAGE_WRITEONLY, VERTEXFVF, \
                            D3DPOOL_MANAGED, &pVertices);

// Set the vertex data
BYTE *VertPtr;
pVertices−>Lock(0, 0, (BYTE**)&VertPtr, D3DLOCK_DISCARD);
memcpy(VertPtr, Verts, sizeof(Verts));
pVertices−>Unlock();
```

In this code, you create a vertex buffer to hold four vertices, which are arranged in a triangle strip to form a quad polygon. The vertex buffer you're using here is a very common one that contains the vertex coordinates and a single set of texture coordinates.

After you've created the vertex buffer, it is time to load your animated texture.

Tip If you're going for two−dimensional video−media playback (as opposed to rendering the texture to a series of polygons), you can substitute the untransformed 3D vertex coordinates for a set of transformed 2D vertex coordinates (thus specifying the `D3DFVF_XYZRHW` FVF flag).

Loading the Animated Texture

You've got the vertex buffer ready, and of course you've got your animated texture file. All that's left is to load up the texture and draw away! For this example, derive the `cAnimatedTexture` class into one that loops the video playback by overriding the `EndOfAnimation` function so that the function calls `Reset` whenever the video is finished. Here's the derived class you use:

```
class cAnimTexture : public cAnimatedTexture
{
  public:
   BOOL EndOfAnimation() { Restart(); return TRUE; }
};
```

From here, just create an instance of the `cAnimTexture` class and, using the `Load` function, load your video.

```
cAnimTexture *g_Texture = new cAnimTexture();
g_Texture−>Load(pDevice, "Texture.avi");
```

At this point, your video texture is loaded, and you're ready to start drawing the scene.

Preparing to Draw

It's time to prepare to draw the polygons that use your animated texture. This is typical rendering code−no need for anything special here. Just set your vertex stream source, shader, material, and the texture. That's right−you only need to set the texture surface that contains the video data; DirectShow will handle the rest!

The following code follows along with the sample in this chapter to set the vertex stream source, FVF shader, material, and texture:

```
pDevice->SetFVF(VERTEXFVF);
pDevice->SetStreamSource(0, pVB, sizeof(sVertex));
pDevice->SetMaterial(&Material);
pDevice->SetTexture(0, g_Texture->GetTexture());
```

After you set the source, FVF shader, material, and texture, all you have to do is draw your primitives and present the scene.

Drawing and Presenting the Scene

Again, you don't need to do anything special to draw your graphics primitives and present the scene. Since DirectShow handles the streaming of video data in the background, you can draw your primitives as you normally would and present the scene as usual.

The following code will render the vertex buffer, free the texture and vertex source usage (to avoid memory leaks), and present the scene.

```
pDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
pDevice->SetStreamSource(0, NULL, 0);
pDevice->SetTexture(0, NULL);
pDevice->Present(NULL, NULL, NULL, NULL);
```

And there you have it−a fully animated texture, compliments of DirectShow and you! Now that you've seen just how easy it is to work with animated textures, you can start adding effects to your games, such as animated facial textures, full motion backdrop images, and even cinematic sequences playing from within your 3D engine.

Check Out the Demos

To demonstrate the texture animation techniques you read about in this chapter, there are two demos at your disposal. These two demos, both displaying a waterfall pouring over a mountainside (see Figure 14.4), offer you a small glimpse of what's possible with texture animation.

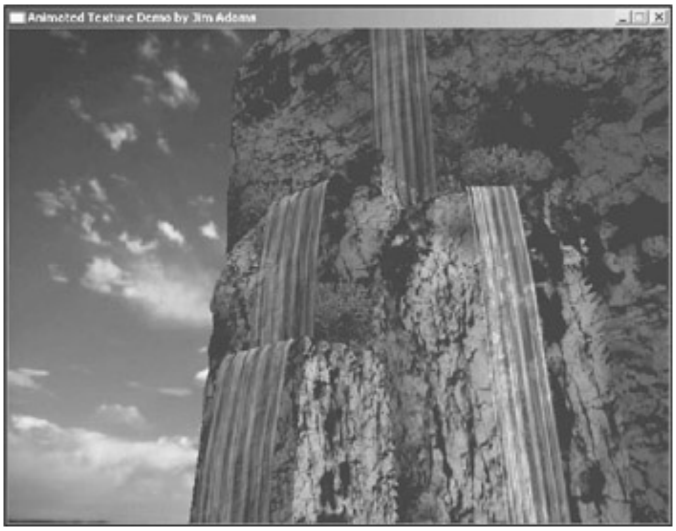


Figure 14.4: *A cool waterfall gushes from unseen sources over a mountainside in the TextureAnim and Transformations demos.*

The first demo, Transformations, uses texture transformations to smoothly scroll the water texture, giving the appearance of a flowing waterfall. The second demo, TextureAnim, uses the video media texture filters developed in this chapter to animate the surface of the water.

Wrapping Up Advanced Animation

Sniff.... Unfortunately, my friend, we have reached the end of the road. You have completed the last chapter. Don't fret, however, because this book is only the tip of the iceberg. There are so many more advanced animation techniques out there for you to learn! If you're feeling adventurous, check out Appendix A, "Web and Book References," to see what resources are out there for you. From further reading to cool Web sites, there's sure to be something that'll catch your fancy.

So go forth, my lad, and see what the world of animation has yet to offer. Good luck!

Programs on the CD

The Chapter 14 directory on the CD−ROM includes two projects.

- **Transformations**. This project demonstrates using texture transformations to animate the water shown in the scene. It is located at \BookCode\Chap14\Transformations.
- **TextureAnim**. This source file uses the texture filter class developed in this chapter to demonstrate how to draw scenes using animated textures. It is located at \BookCode\Chap14\TextureAnim.

Part Six: Appendixes

Appendix A: Web and Book References

We live in a time in which information is plentiful and easily accessible−it's only a matter of knowing where to look. In this appendix I have compiled a list of Web sites and books that you might find useful in your quest for knowledge on using advanced animation in your own projects.

Web Sites to Check Out

Knowing where to look for help and information on the Net is difficult enough; with thousands of Web pages to weed through, your quest for knowledge can seem like an unyielding adventure. Where are programmers to turn when they need help or information to get their latest projects rolling? Well, here's a short list of Web sites that I find useful. Hopefully the information from these sites will help you as much as it has helped me.

The Collective Mind

http://www.theCollectiveMind.net

I begin the list with my own Web site, where you can check out information about my books, download the newest code updates, read game−related articles, download demos, and basically find stuff that I'm involved in. If you ever need help on one of my books or projects, stop by my Web site and see if there's an update, or feel free to drop me an e−mail.

GameDev.net

http://www.GameDev.net

This is THE place to go for all your game development needs. Hook up with other developers on the message boards, read up on articles, enter coding contests, and download hundreds of home−brew games and demos created by GameDev's devoted followers. Aside from my own Web site, this is the best place to find me, hanging out in the message boards.

Premier Press Books

http://www.PremierPressBooks.com

Here you can check out Premier's lineup of cool game−related programming books, as well as upcoming titles that might catch your fancy. Aside from game programming, Premier offers a full line of technical books that extends into many computer−related fields, such as graphics modeling, digital imaging, network and security, and much, much more.

Microsoft

http://www.microsoft.com

If you're reading this book, chances are you already know about Microsoft's Web site. Specifically, Microsoft's DirectX and Agent Web pages are where you want to be. With articles, demos, and downloads galore, you'll want to make sure you keep an eye on this site and see what's new.

Caligari

http://www.caligari.com

Low cost and high quality is the name of the game, and Caligari knows just how to play. Caligari is the company behind the easy−to−use trueSpace 3D modeling package. This Web site is a definite must−visit for those of you who are on a budget and have need for great modeling software. With features such as facial animation (as seen in this book), non−linear editing, and texture−baking, as well as the ability to export right to the .X file format, you should give Caligari's Web site and modeling packages a look.

NVIDIA

http://www.nvidia.com

Creators of the GeForce series of video cards, NVIDIA maintains a long list of DirectX and OpenGL developer−related documents and demos worth checking out. Check this Web site for advanced information on vertex shaders, pixel shaders, and NVIDIA's own high−level shader language, Cg.

ATI Technologies, Inc.

http://www.atitech.com

Second to none, ATI is the company behind the Radeon series of video cards. Like NVIDIA's Web site, ATI maintains a long list of documents, utilities, and demos aimed at DirectX and OpenGL developers. Check out this Web site for information on shader development using ATI's own RenderMonkey software package.

Curious Labs, Inc.

http://www.curiouslabs.com

Mentioned briefly in this book, Curious Labs' Web site is the home of the awesome Poser 3D character modeling and animation software package. Allowing you to freely pose, animate, render, and export 3D characters to popular file formats, Curious Labs' Poser software package is something to seriously consider, and this is definitely a Web site you'll want to visit.

Polycount

http://www.polycount.com

Polycount is the premier source for 3D character models you can plug into your favorite games such as *Quake II* and *III, Half−Life,* and *Grand Theft Auto 3*. If you need some models for your game, this is definitely the place to find them, as well as being a terrific site to find artists for your next project!

Flipcode

http://www.flipcode.com

Another infamous programming Web site. Here you can read up on the latest happenings in the programming field, message other users, and even check out the cool Image of the Day gallery. While you're there, check out Kurt's links to some interesting science discoveries and innovations!

Chris Hecker and Jeff Lander Physics

Chris Hecker: http://www.d6.com/users/checker

Jeff Lander: http://www.darwin3d.com

The goal of this book is to help you produce some awesome animation effects as quickly as possible. I'll admit that at times I was a little lax on the physics, but the purpose was to provide you with only what you need to get the work done. For further reading, I recommend picking up any one of Chris Hecker or Jeff Lander's articles or demos on using physics in games.

Recommended Reading

Behind every great programmer is his or her collection of books. I am no different, so I offer you a glimpse at those books that I find helpful in relation to the book you are now holding in your hands.

Programming Role Playing Games with DirectX ***by Jim Adams (Premier Press, 2002)***

ISBN: 1−931841−09−8

Another shameless plug for myself! Actually, my role−playing game programming book is a great resource for those who want to make a complete game from scratch. Not just limited to role−playing games, my book shows key components that you can use in any game engine, such as an octree graphics engine and character and inventory control. It also shows you how to build a complete core of game−programming libraries to ease the development of your game projects.

Focus On 3D Models ***by Evan Pipho (Premier Press, 2002)***

ISBN: 1−59200−033−9

The butter to my bread, the Sonny to my Cher, this book is your guide to the various 3D model formats being used in the market's hottest games. A great companion book for those readers who want to use more than the .X, .MD2, and .MS3D file formats I outlined in this book.

Real−Time Rendering Tricks and Techniques in DirectX ***by Kelly Dempski (Premier Press, 2002)***

ISBN: 1−931841−27−6

Kelly Dempski shows you how to get started using some advanced rendering methods such as vertex shaders to achieve some cool effects. This book is handy if you're getting started in vertex and pixel shaders, and it also explores other game−enhancing topics.

Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks ***by Wolfgang F. Engel (Wordware, 2002)***

ISBN: 1−556220−41−3

This is another great book packed with pixel shader and vertex shader goodness that works hand−in−hand with this book. Learn how to improve your graphics by using effects such as bubble rendering and rippling water. Using these techniques with what you've read in this book, you're guaranteed to create some awesome special animation effects!

Appendix B: What's on the CD

Overview

You've read the book; now it's time to check out the software! This book's CD−ROM contains the source code to every one of those cool demos mentioned in each chapter, as well as a handful of useful utilities and applications that I used to create the effects mentioned in this book.

If you haven't already done so, I recommend tearing open that CD−ROM sleeve and plopping the CD in your computer's drive. If Autorun is enabled, you should be greeted with an interface that will guide you through the CD−ROM's contents. If Autorun is disabled, you can still run the interface by following these steps:

1. Insert the CD−ROM into your computer's CD drive.
2. Right−click on My Computer and select Open from the menu.
3. Click on your CD drive in the list of drives.
4. In the list of the CD's contents, double−click on the Start_Here.html file. After reading the licensing agreement, click I Agree if you accept the terms (or click I Disagree to quit the interface).

If you accepted the licensing terms and clicked on the I Agree button, you'll be presented with the Premier Press user interface. From there you can browse the contents of the CD−ROM or select from a number of applications to install. Following is a list of the applications you will find on the CD−ROM.

DirectX 9.0 SDK

The main application is Microsoft's DirectX 9 SDK. Used in all the projects in this book, the SDK is the first thing you should install. For help installing the DirectX 9.0 SDK, consult Chapter 1, "Preparing for the Book."

GoldWave Demo

Each of the sounds you find in this book's demos was created with GoldWave, a lightweight sound−editing tool. Created by GoldWave Inc., this handy little program allows you to work on multiple sounds at the same time. Recording, playing back, and altering sounds was never so easy as with this program. With GoldWave, you can

- Edit sound files up to 1 GB in size
- Use real−time graphs to view amplitude, spectrum, and so on
- Zip around using fast−forward and rewind features
- Save or load to and from multiple sound formats, such as .WAV, .AU, .MP3, .OGG, .AIFF, .VOX, .MAT, .SND, and .VOC
- Use drag−and−drop cue−points to mark specific areas of your sounds

Paint Shop Pro Trial Version

Image editing on a budget. With Paint Shop Pro, you can edit still images just like the pros, without having to pay as much! This 30−day trial version of Jasc's image−editing program is top−notch, allowing you to open and save multiple image formats, work with multiple layers, use extensive plug−ins, and retouch images.

TrueSpace Demo

Want powerful 3D editing capabilities at a great price? Then Caligari's trueSpace 3D modeling program is right up your alley. This demo program lets you explore the various aspects of Caligari's newest version of trueSpace. With advanced features such as a facial animation system, hybrid radiosity rendering, and non−linear time editing, this program is something you'll certainly want to check out!

Microsoft Agent and LISET

Microsoft's Agent package contains some extremely useful language development tools to help you create text−to−speech, speech recognition, and lip−synced animation applications. With the LISET program distributed with Agent, you can create your own lip−syncing animation for your game projects, as shown in this book.

List of Figures

Chapter 3: Using the .X File Format

Chapter 11: Morphing Facial Animation

List of Tables

List of Sidebars

Chapter 9: Using Key−Framed Morphing Animation

Programs on the CD

Chapter 10: Blending Morphing Animations

Programs on the CD

Chapter 11: Morphing Facial Animation

Programs on the CD

Chapter 12: Using Particles in Animation

Programs on the CD

Chapter 13: Simulating Cloth and Soft Body Mesh Animation

Programs on the CD

Chapter 14: Using Animated Textures

Programs on the CD