COURSE TECHNOLOGY
CENGAGE Learning
Professional • Technical • Reference

Character Animation with Direct3D®

Carl Granberg

Character Animation with Direct3D®

Carl Granberg

Charles River Media
A part of Course Technology, Cengage Learning

COURSE TECHNOLOGY
CENGAGE Learning™

Australia, Brazil, Japan, Korea, Mexico, Singapore, Spain, United Kingdom, United States

Character Animation with Direct3D®

Carl Granberg

Publisher and General Manager,
Course Technology PTR:
Stacy L. Hiquet

Associate Director of Marketing:
Sarah Panella

Content Project Manager:
Jessica McNavich

Marketing Manager: Jordan Casey

Senior Acquisitions Editor: Emi Smith

Project Editor and Copy Editor:
Dan Foster, Scribe Tribe

Technical Reviewer: Henrik Enqvist

CRM Editorial Services Coordinator:
Jennifer Blaney

Editorial Services Coordinator: Jen Blaney

Interior Layout: Jill Flores

Cover Designer: Mike Tanamachi

CD-ROM Producer: Brandon Penticuff

Indexer: Valerie Haynes Perry

Proofreader: Ruth Saavedra and
Mike Beady

Library of Congress Control Number: 2008931080

ISBN-13: 978-1-58450-570-9

ISBN-10: 1-58450-570-2

eISBN-10: 1-58450-630-X

Course Technology, a part of Cengage Learning
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: **international.cengage.com/region**

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit **courseptr.com**

Visit our corporate website at **cengage.com**

Printed in the United States of America
1 2 3 4 5 6 7 11 10 09

To Aino… again.

Acknowledgments

As always with a project such as this, there's a long list of people deserving my thanks. Writing a book is not a small feat (yes, patting myself on the back), and it is also something I could not have done alone. So first off I must thank the people of Charles River Media for wanting to publish this hunk of technical mumbo jumbo, and especially Emi Smith, my editor. Big thanks also to Dan Foster, my project editor, and Henrik Enqvist of Remedy Entertainment, my technical editor. Henrik also supplied the code for the Inverse Kinematics chapter and the Wrinkle Maps example, for which I owe him thanks.

Next I'd like to thank my friend Markus Tuppurainen for supplying some of the art for this book—sketches and pixel characters—as well as for helping me make the Soldier model.

Finally I'd like to thank my wife and my family for their support through this last year, which has been challenging in many ways.

Last, but not least, thank *you* for buying this book. I hope you enjoy it and also learn something from it.

About the Author

Carl Granberg has been creating games on a hobby basis since the old DOS-based Mode 13h graphics, after which he moved on to DirectDraw and finally Direct3D graphics. He received his master of science in computing at Curtin University, Perth, Australia, and has since been working in the Finnish game industry for 3 years.

He is currently working as a Tools programmer at Remedy Entertainment in Finland. He's also involved with a group of hobby game developers that goes by the name of BugFactory (**www.bugfactory.fi**), which has just released its first title, *The Tales of Bingwood*.

For matters relating to this book, he can be contacted at **carl@bugfactory.fi.**

Contents

Introduction

Intended Audience

This book is primarily aimed at teaching indie and hobby game developers how to create character animation with Direct3D. Also, the seasoned professional game developer may find some interesting things in this book.

You will need a solid understanding of the C++ programming language as well as general object-oriented programming skills.

As for DirectX, you will need to know the very basics at least. In other words, you will need to have completed at least an introductory book on DirectX before starting this one.

On top of all these prerequisites, you should also have basic knowledge of the High Level Shading Language (HLSL), since many of the effects done in this book will use it.

If you feel that you can't honestly say you meet these prerequisites, I suggest you brush up on these topics before continuing with this book rather than trying to learn them as you go. You will quite quickly be faced with some advanced topics, and, if you are faced with them for the first time, they will be quite hard to handle without trying to learn HLSL or similar topics as well.

But, hey, this is just my suggestion. After all, that certainly wasn't how I learned the stuff I know today.

Using This Book

This book has been divided into 16 chapters, each of which usually focuses on one or a few related components. I aim to keep this book very "hands-on," so a lot of code will be covered throughout. You're probably best off reading the book from cover to cover, since a lot of stuff covered in the earlier chapters will be built upon in later chapters.

The topic of character animation is a very general one that can be applied to all game genres. It doesn't matter if you are making your own role-playing games (RPG), real-time strategy games (RTS), first-person shooter games (FPS), or a game from another genre. As long as you plan to include characters in your game, you will benefit greatly from learning the topics covered in this book.

Because the topic is extremely code intensive, you won't find most of the code written out in full throughout this book. Rather, use the book as a manual to understand the code found on the accompanying CD-ROM. Also, if you have time I suggest that you try to implement the topics covered here completely on your own, and use the code provided only as guidelines or a helping hand. Even though this might seem like a tedious waste of time, I can guarantee that it will greatly increase your understanding of the different techniques (although, of course, I know that 95% of readers will pay no attention whatsoever to this recommendation).

To get to the fun stuff as soon as possible, I won't waste time covering simple things like basic Direct3D rendering, basic data structures, and so on. There are more books available on these topics than absolutely necessary, so if you feel you're lacking in knowledge about basic DirectX programming, I suggest you go and pick up such a book before getting back to this one. Also, I'll rely heavily on the Standard Template Library (STL) for all basic data structures such as vectors, stacks, queues, etc. For all generic 3D math functions, mesh and texture loading, and more, I will be using the D3DX library. This is a part of Direct3D and is a great help when developing 3D applications (as you'll soon see).

You'll find all the examples on the CD-ROM together with their executables, models, textures, and more. The examples are ordered according to the chapter number and the example number. Usually the examples are fairly simple and focus only on one specific thing. At the end of the book, however, there will be a character that can walk, talk, collide with objects, fall, and more.

System Requirements

Windows Vista/Windows XP
DirectX SDK
Graphic card supporting Vertex and Pixelshader version 2.0
A decent processor
Not too little RAM

1 Introduction to Character Animation

Hello, dear reader, and welcome to this book about character animation! I hope you enjoy it and find it useful. In this chapter, I'll start you off slow by looking at character animation in general as well as a brief history of the same topic. You won't get to do any coding in this chapter, but toward the end I'll include a brief overview of the coding conventions used in this book. In this first chapter, you'll find the following:

- What is character animation?
- A brief history of character animation
- Comparison of skeletal animation and morphing animation
- Coding conventions

What Is Character Animation?

This somewhat silly question may seem pretty easy to answer at a first glance, but is it really? Wikipedia defines it as follows:

> *"Character animation is a specialized area of the animation process concerning the animation of one or more characters featured in an animated work."*
>
> -Wikipedia

Animated work. Well, I guess games falls under that category. However, I would probably have tried to define it along the lines of "Making a character move in a realistic way." Although, I suppose that better answers the question, "What is the goal of character animation?"

Historically, characters were drawn (or pixeled) and animated by making multiple pictures showing the character at a slightly different pose. These pictures would then loop to give the impression of movement. With today's video cards, it is possible to have full three-dimensional characters and animate them with some of the various techniques covered in this book.

New ways of animating character models pop up each year, pushing the evolution of the field forward. The techniques covered in this book are by no means cutting edge; rather, they are the foundational techniques that all (or most) cutting-edge technologies are based upon. Techniques such as skeletal animation, morphing, ragdoll physic simulation, and inverse kinematics have already existed for a long time (in terms of game evolution at least).

Still, at the end of this book you will have all the tools you need to create your own game featuring realistic character animation.

A Brief History of Character Animation

Let's start from the beginning! Say hello to one of the first well-known computer game characters of our time: *Pac-Man* (Figure 1.1).

This 28 × 28 pixel character (developed by Namco) was released in Japan in 1980 and is still today the most famous arcade game of all time. This character (looking more like a pizza missing a slice) slowly gave way to more humanoid characters. Four years later, Sierra On-Line released *Kings Quest: Quest for the Crown*, staring Sir Graham (Figure 1.2).

FIGURE 1.1
Pac-Man.

FIGURE 1.2
Sir Graham.

Sir Graham might not feature many more pixels than Pac-Man did, but at least he was more colorful and had a great set of animations. Characters continued along the same lines through the late '80s, with steadily increasing pixel count and/or color. In 1987, LucasArts developed its first version of the SCUMM engine (Script Creation Utility for Maniac Mansion) and with it they released several adventure games, including *Maniac Mansion*, *Monkey Island*, *Loom*, and many more. The characters of this era (late '80s to early '90s) pretty much shared the same complexity—Figure 1.3 shows an example.

FIGURE 1.3
An animation sequence of Tom (*The Tales of Bingwood*). © BugFactory 2008.

Suddenly the '90s hit, and with the new decade the first 3D games brought a whole new set of problems. Some of the more famous games included *Wolfenstein 3D* and, later, *DOOM*—although these games can't really be called proper 3D games since they still used 2D sprites for enemies and characters (usually drawn from eight angles, depending on their orientation to the player). The first real 3D character was seen in the game *Alone in the Dark*, which was released in 1992. It featured characters in full 3D with interpolated animations. These characters had an extremely low polygon count and were built from several blocks (one for each limb). An example character from this era can be seen in Figure 1.4.

You can easily see the obvious gaps between the joints in this character, but back then there was usually no lighting of the models and the resolution was so small that these gaps were often hidden from the player.

Jumping ahead in time a few years, we reach 1996, when 3dfx launched the first Voodoo chipset and with it brought affordable 3D accelerator cards to the masses. One of the first reputable games taking advantage of this new technology was the game *Quake*. With *Quake* came seamless characters (albeit low-poly) animated using vertex morphing.

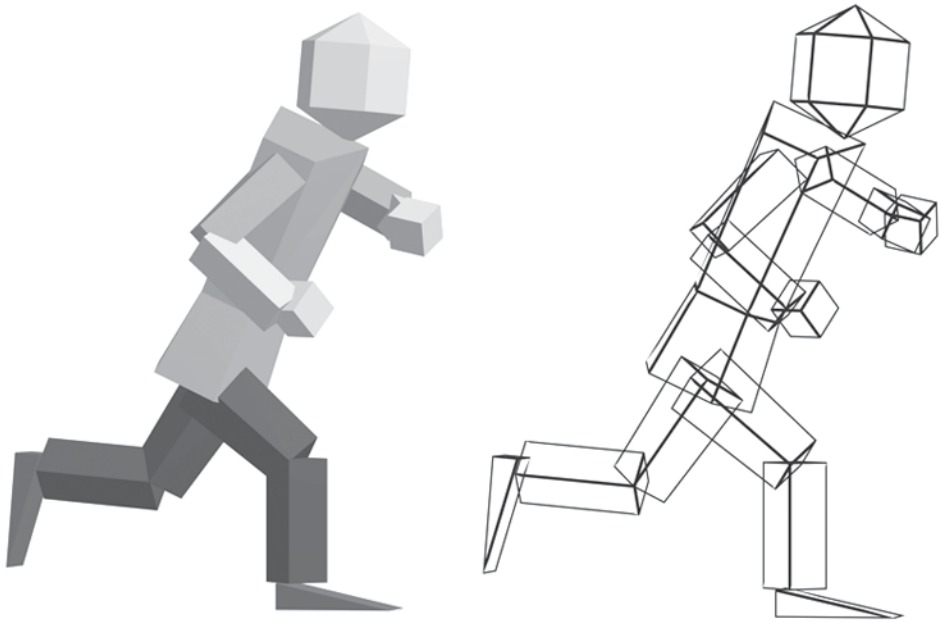


FIGURE 1.4
A character built from blocks.

Morphing Animation and Skeletal Animation

Morphing animation (or per-vertex animation) works by blending two (or more) meshes together on a per-vertex basis. The two meshes need to have the same amount of vertices, and their polygons need to be arranged in the same way for this technique to work. Each mesh representing a pose of the character is referred to as a morph target. More than one morph target may be used to blend the final mesh. The main use of morphing animation these days is facial animation. But in the past it was also used to create full-body character animations. For instance, the *Quake* I and II engines used this approach for their characters using the popular but slightly outdated MD2 file format [Schoenblum07, Leimbach02].

See Figure 1.5 for an example of *morphing animation*. In this figure, only the HAPPY and ANGRY frames are the actual target meshes. The meshes in between are created by interpolating the vertex positions smoothly over time.

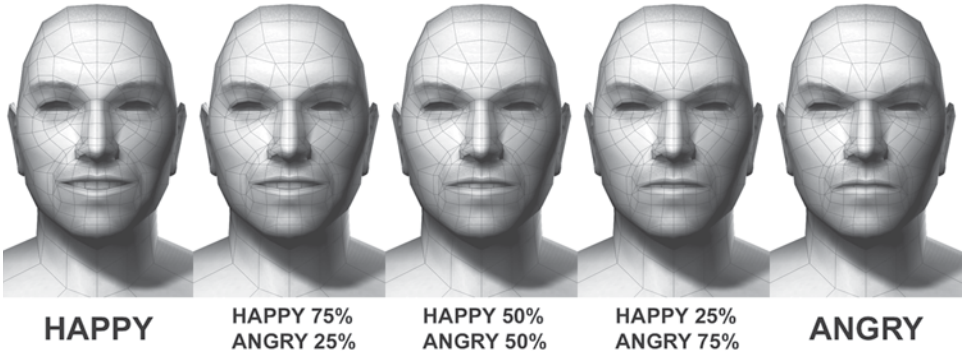


FIGURE 1.5
An example of morphing animation.

Fast-forwarding in time again brings us to 1998, when Sierra Studios released the game *Half-Life* (developed by Valve). *Half-Life* was built on top of a highly modified version of the *Quake* engine. Most notably, the game developers added a new skeletal animation system, allowing them to reuse animations on different characters.

As the name implies, *skeletal animation* is closely linked to the workings of a skeleton. An average human body has about 206 bones. The states and locations of all these bones define the pose of a person. As the bones move from one location in space to another, the surrounding muscles, tissue, and the outer skin move with it. This basic idea is the key to skeletal animation. The only difference is that for computer games you are just interested in the skin layer (i.e., what the player sees). In Chapter 3, you will learn how to "skin a character."

See Figure 1.6 for an example of the wireframe rendering of a skinned character. Notice how the skin (mesh) follows the bones as they move.

Since the days of the first *Half-Life* game, characters have been getting more polygons, larger textures, normal maps, advanced shaders, and more to make them look better and better every year. However, the basic underlying technologies haven't changed much.

These two techniques—skeletal animation and morphing animation—are widely used today in game development, and this book will cover both. They both have their advantages and disadvantages. At the end of this book, you will know how to create characters that make use of both techniques—e.g., skeletal animation for overall movement, and morphing animation for more subtle things like facial expressions.

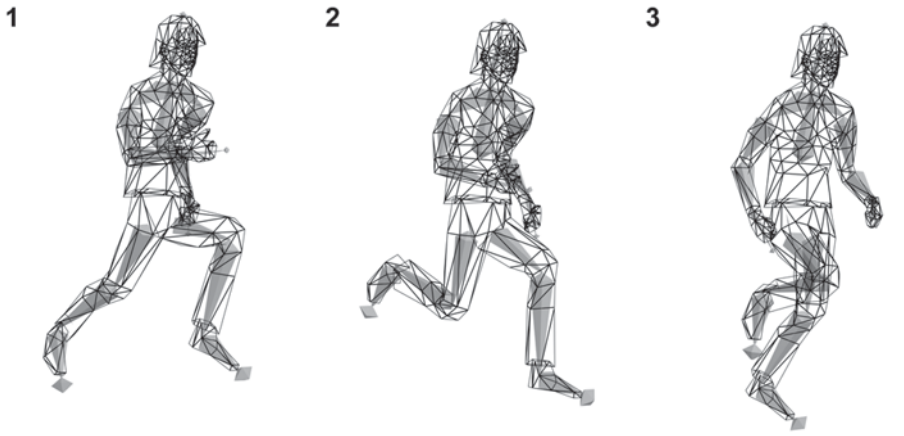


FIGURE 1.6
Three frames of a character animation using skeletal animation.

The Soldier

I will refer to the example character used throughout this book as "the Soldier." What looks like yet another futuristic-hero-figure-in-power-armor is…well, actually just that: another futuristic-hero-figure-in-power-armor.

The design for the Soldier was based on old roman soldiers, which you might detect from the shoulder pads and helmet.

Design and texturing for the Soldier was done by Markus Tuppurainen for our adventure game, *Day of Wrath*. Although that game was never finished (yes, yes, I don't manage to finish all the games I start either), the model still has its uses for this book. The important thing is that he has all the necessary limbs, some animations, skinned meshes (body and face), and some static meshes (helmet and pulse rifle).

The model complexity ranges somewhere in the low to medium range by today's standards:

Body Mesh:	2100 Polygons
Head Mesh:	1000 Polygons
Num Bones:	26
Height:	1.8 Units

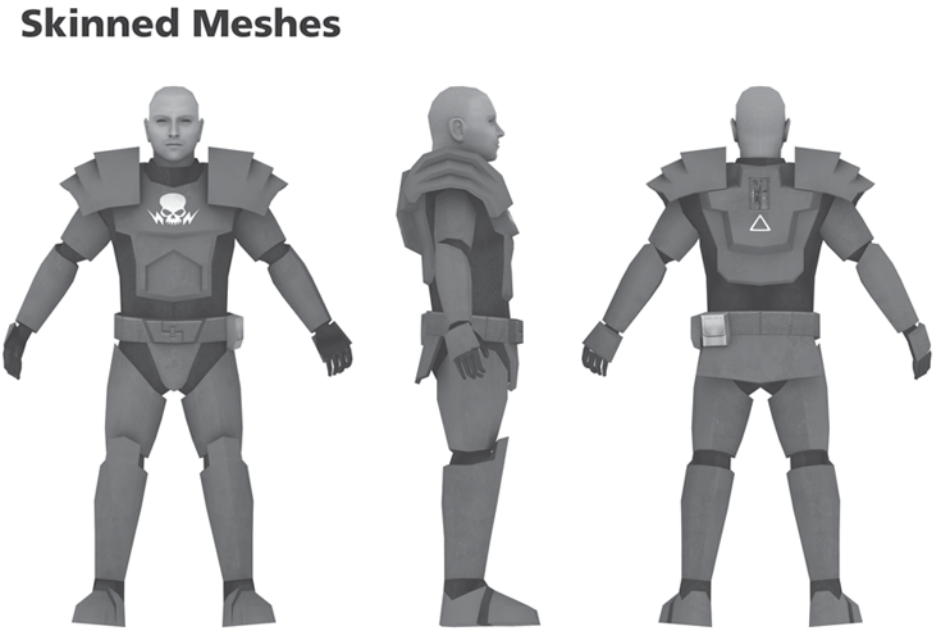


FIGURE 1.7
The Soldier.

Coding Conventions

Throughout this book I will use a subset of the Hungarian notation standard (and I'll try to be consistent). The High Level Shading Language (HLSL) effects in this book will depart slightly from this standard and use the notation used in Engel's shader books. See Table 1.1 for the coding conventions used in the C++ examples of this book.

TABLE 1.1 CODING CONVENTIONS

So your average C++ class would look something like the following:

```
class SomeClass
{
    public:
        SomeClass();
        ~SomeClass();
        void SomeFunction1(int someParameter);
        bool SomeFunction2();

    private:
        int m_memberVariable;
        float* m_pMemberPointer;

};
```

Conclusions

Hopefully after reading this chapter you've gained some perspective on the topic of character animation and the work in this field that has come before us. However, after this brief warm-up, it is time to get started and to get your hands dirty. In the next chapter, you'll be briefly introduced to Direct3D as well as the necessary steps to create a 3D application, most of which will probably be repetition for you. At the end of the next chapter, you will have a character rendered to your screen (albeit a very stiff one). Then, after Chapter 2, more functionality will be added to our now somewhat inanimate Soldier in each chapter, bringing him more and more to life.

Further Reading

[Schoenblum07] Schoenblum, Daniel E. ".md2 File Format Specification." Available online at http://www.linux.ucla.edu/~phaethon/q3/formats/md2-schoenblum.html, 2007.

[Leimbach02] Leimbach, Johannes. "Character Animation with DirectX 8.0." Available online at http://www.gamedev.net/reference/articles/article1653.asp, 2002.

2 A Direct3D Primer

This chapter covers what you need to know before continuing with the rest of this book, or, in other words…the groundwork. I'll offer a quick glance at the things you should already know (setting up Direct3D, the windows loop, and more). If any of the concepts covered in this chapter don't make sense, well, then you need to take a break and brush up on these. As mentioned earlier, you will need to be comfortable with both object-oriented C++ as well as DirectX 9. So before you tackle the more savory subjects of this book, let's look at the basics first. In this chapter I'll cover the application framework, creating the window, setting up the Direct3D device, and more. However, since these subjects don't really belong to the

core of this book, I will just brush past them and show you the minimum amount of code required to get up and running. Pay special attention to the application framework though, since this is the skeleton class upon which all the other examples in this book are built. This chapter includes the following:

- Getting started
- Application framework
- Rendering with Direct3D

Please note that all code throughout this book is written with clarity in mind, not optimization (or stability). Also to keep things brief, no error checking is done. For example, I rarely check the return values of Direct3D/D3DX functions but simply assume that they completed successfully. Similarly I assume that there is enough memory to create new classes, meshes, textures, etc. So please be mindful of this fact if you plan to use the code from this book in your own projects.

DirectX 9 versus DirectX 10

In this book I will use DirectX 9 to do all rendering and resource management. You might ask, why?—the newer DirectX 10 is already out! Well, to be honest the amount of extra work and support code required to cover the same topics in DirectX 10 simply makes it too grand a job to attempt. In DX9 a lot of support code exists in the D3DX library, much of which has been deprecated in DX10 (to the disappointment of us hobby programmers).

The biggest missing piece needed for this book is the loading of .x files (no, not the TV series, but the file format used by DX9 to store models). At the writing of this book, there's still not any easy way of doing this with DX10. Anyway, if you can write your own mesh importer, you probably don't need my help to port the examples in this book anyway, and if not, well, then DX9 will have to serve.

Another reason to stick with DX9 is that the majority of computers out there still don't have a DX10-compatible graphics card and probably won't for at least another couple of years.

However, no matter which version of DirectX you use to do your rendering, you will still benefit from the lessons in this book. The classes and structures presented in this book are nonspecific to DX9 and can easily be ported to other rendering systems such as DX10 or even OpenGL.

STL and the D3DX Library

Reinventing the wheel is something that I greatly enjoy doing myself. I'll spare you this, however, since you might not have the same fetish. I'll therefore rely heavily on the Standard Template Library (STL) for all my data container classes and the Direct3D eXtension (D3DX) Library for math functions, resource loading functions, etc. Here's a simple use of the stl::vector class, if you haven't seen it before:

```
//Create a vector of integers
vector<int> intVector;

//push some numbers
intVector.push_back(3);
intVector.push_back(1);
intVector.push_back(2);

//Sum up the numbers in the vector
int sum = 0;

for(int i=0; i<intVector.size(); i++)
{
        sum += intVector[i];   //Access each int with the [] operator
}
```

This is a pretty simple example demonstrating a vector of integers. It's not very useful in practice, however; it's more common that you have a vector of pointers to a user-defined class. Take a look at the following three classes:

```
//Monster Interface
class IMonster
{
    public:
        IMonster();
        virtual void Update(float deltaTime) = 0;
        virtual void Render() = 0;

    protected:
        D3DXVECTOR2 m_position;
};

//Your run o' the mill savage goblin
class Goblin : public IMonster
{
```

```
    public:
        Goblin(float startHealth);
        void Update(float deltaTime);
        void Render();

    private:
        float m_health;
};

//Your ghost on the attic (can be friendly or unfriendly)
class Ghost : public IMonster
{
    public:
        Ghost(bool isFriendly);
        void Update(float deltaTime);
        void Render();

    private:
        bool m_friendly;
};
```

The `Ghost` and the `Goblin` classes both inherit from the `IMonster` interface. The functions `Update()` and `Render()` are declared in the `IMonster` class as purely virtual functions that have to be implemented in the classes inheriting from it. But you should know all this already. If not, you might want to brush up on some basic object-oriented concepts like inheritance, polymorphism, etc. [Llopis03]. Anyway, getting back to the STL vector, here's how you would create, update, and render a bunch of monsters:

```
//Create a vector of monster pointers
vector<IMonster*> monsters;

//create some monsters
monsters.push_back(new Goblin(55.0f));
monsters.push_back(new Goblin(35.0f));
monsters.push_back(new Ghost(true));
monsters.push_back(new Ghost(false));
monsters.push_back(new Goblin(62.0f));

//Iterate through the monsters and call update and render
for(int i=0; i<monsters.size(); i++)
```

```
{
    monsters[i]->Update(deltaTime);
    monsters[i]->Render();
}
```

The ghost and the goblin can have completely different updating and rendering functions. However, since they both implement the `IMonster` interface, the monster vector doesn't care which exact class each item in the vector represents. You'll see a lot of STL containers used in similar ways throughout this book (vector, queue, map, etc.).

That pretty much covers how I'll be using the STL library. The D3DX library, on the other hand, is a collection of functions, structures, and classes that will also be richly used throughout this book. You can recognize the D3DX functions, etc. by their prefix (yep, you guessed it) D3DX as seen in: `D3DXMATRIX`, `D3DXVECTOR3`, `D3DXVec3Normalize()`, `D3DXMatrixIdentity()`, and much more. I'll try to introduce all these new functions as they are used in the book (instead of covering them all here). Remember that you always have the DirectX SDK documentation where all these functions and structures are covered in great detail.

Setting Up a Project in Visual Studio Express 2008

If you already know how to set up a project in Visual Studio and get up and running with it, then feel free to skip this section.

DirectX applications are now quite simple to make for free with Visual Studio Express 2008. If you have other versions of Visual Studio, the steps to set up a DirectX project are the same. Note, however, that earlier versions of Visual Studio Express such as 2005, etc. can't build Win32 applications without some extra hassle. So unless you own a copy of Visual Studio, you're better off sticking to VS Express 2008.

You can download VS Express 2008 from:

http://www.microsoft.com/express/vc/.

Go ahead and install it. Next you need the DirectX SDK, found here: **http://msdn.microsoft.com/en-gb/directx/default.aspx**.

Follow the DirectX SDK links and install this as well.

VC++ Directories

When you build you're project you'll need header and library files from the Direct SDK. To ensure that Visual Studio can find and link these files, follow these steps:

1. Open up Visual Studio Express.
2. Click Tools, and then select Options.
3. Select "VC++ Directories" from the list at the left (under Project and Solutions).
4. Select "Include Files" in the "Show directories for" drop-down box.
5. Make sure that a link to your DirectX Include folder exists in the list below; if not, add one.
6. Now select "Library Files" in the "Show directories for" drop-down box.
7. Make sure that a link to your DirectX Lib (either x86 or x64, depending on which platform you're building for) folder exists in the list below; if not, add one.

Figure 2.1 shows an example of the Options screen:

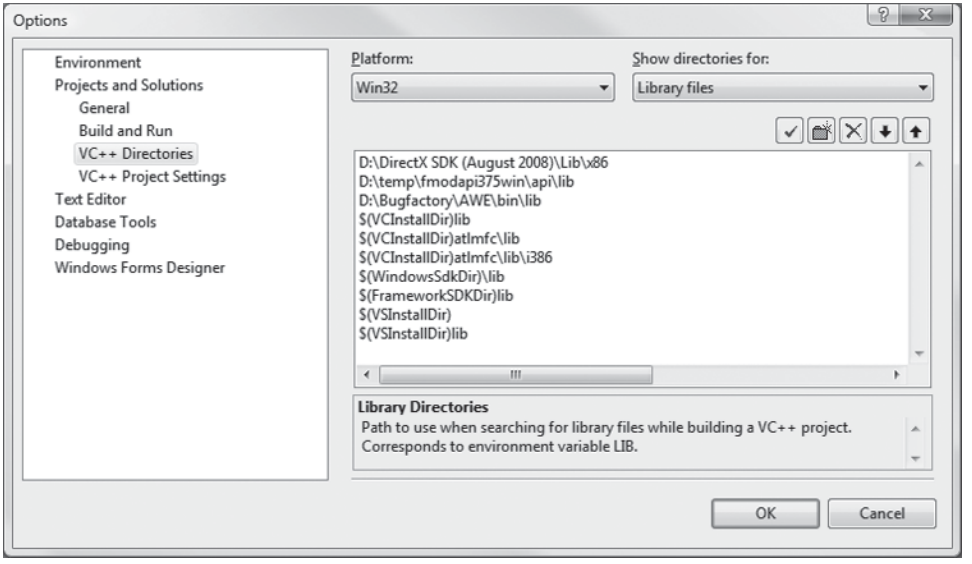


FIGURE 2.1
The Visual Studio Options screen.

Setting up the VC++ directories only needs to be done once. The following, however, needs to be done for each individual project.

Creating a New Project

To create a new empty project in Visual Studio, follow these steps:

1. Select File > New > Project.
2. Select Win32 Project.
3. Enter a project name.
4. Select the project folder.
5. Press OK.

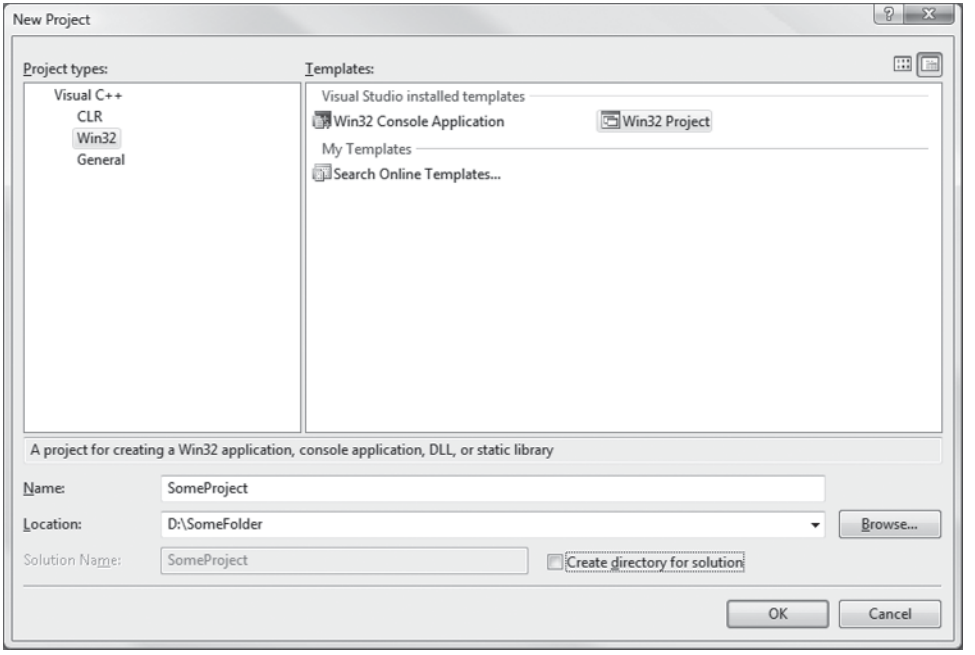


FIGURE 2.2
The New Project screen.

6. In the Application Wizard that pops up, click Next.
7. Select Windows Application as the Application Type.
8. Click "Empty Project" in Additional Options.
9. Click Finish.

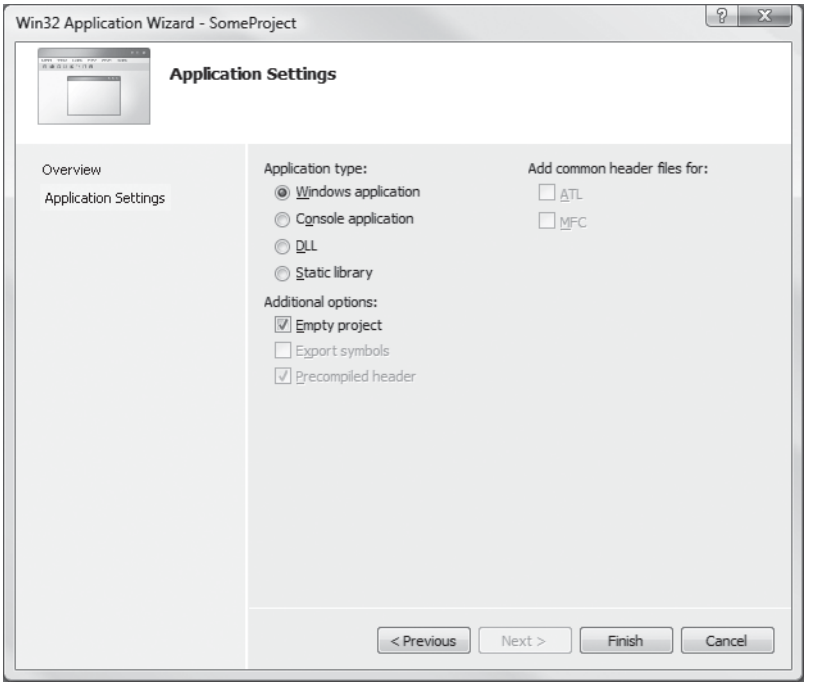


FIGURE 2.3
The Application Wizard.

You have now created a new project. Before you can start to compile and build DirectX applications, however, you need to link the DirectX libraries to your application.

Linking DirectX Libraries

You need to tell the linker which external libraries your application will be using. This may vary from project to project, of course, depending on what functionality you intend to use. To add libraries to your application, follow these steps:

1. Select the project you are working on in the Solution Explorer (a solution can contain more than one project).
2. Select Project > Properties (or press Alt + F7).
3. Expand the "Configuration Properties" node in the left tree view.
4. Expand the "Linker" node.
5. Select the "Input" node.
6. In the "Additional Dependencies" field, enter the filename of the libraries you intend to use.

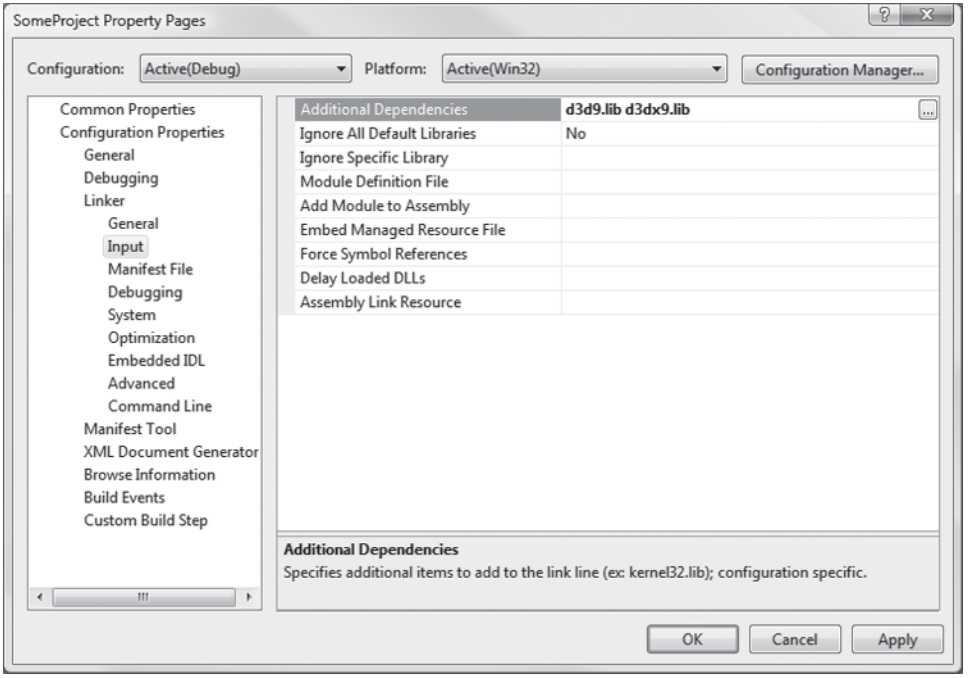


FIGURE 2.4
Project Property Pages.

For most of the examples in this book I will link only the following DirectX libraries:

d3d9.lib DirectX Core Library
d3dx9.lib DirectX eXtension Library

That's it! That pretty much covers the boring part. You are now ready to write, compile, and build DirectX applications. Next I'll show you the Application framework you'll be using throughout this book.

Application Framework

Since you'll be writing a Win32 application to run your game through, you'll need a class to deal with the main program loop, initialization of the graphics device, and more. For this purpose I'll use the `Application` class. Once again, keep in mind that I

do a minimum of error checking in this class to keep it light, so for any "advanced" application, I would suggest the DXUT framework as a starting point (accompanying the DirectX SDK). Bearing this in mind, here's the overview of the `Application` class:

```
class Application
{
    public:
        Application();
        ~Application();
        HRESULT Init(HINSTANCE hInstance, bool windowed);
        void Update(float deltaTime);
        void Render();
        void Cleanup();
        void Quit();

        void DeviceLost();
        void DeviceGained();

    private:
        HWND m_mainWindow;
        D3DPRESENT_PARAMETERS m_present;
        bool m_deviceLost;
};
```

A quite slim class, all in all. The functions' names pretty much explain what they do. After an `Application` class has been created, the `Init()` function must be called. In this function, resources are loaded and the graphics device is created.

Each frame the `Update()` function is called with the delta time since the previous frame, before the `Render()` function is called. The `Update()` function takes care of updating the world (i.e., the game), moving objects, updating the physics engine, and more. Once done, the `Render()` function renders all the objects and presents the result to the screen.

The `DeviceLost()` and `DeviceGained()` functions require some explanation. These functions are called when the device is lost or gained. This happens when the window is resized or when the user switches from full screen to windowed mode, etc. All resources stored in video memory need to be released on the device lost event and recreated when the device is regained. (This again is stuff that you hopefully know already and something this book won't touch upon.)

Once the application has run its course and the `Quit()` function is called (done by pressing Esc or Alt + F4 in the examples), the `Cleanup()` function is called and in it any resources that were created are released. So how is the `Application` class used?

WinMain

For those who are already familiar with the basics of windows programming, I apologize for the upcoming sections and beg you to skip ahead. For the rest of you…read on.

The `WinMain()` function is the entry point of your Win32 program. It is in this function that the entire application exists and runs its course. The following code shows the simple `WinMain()` function you'll be using for the upcoming examples:

```
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE prevInstance,
                   PSTR cmdLine,
                   int showCmd)
{
    //Create a new Application object
    Application app;

    //Initialize it
    if(FAILED(app.Init(hInstance, true)))
        return 0;

    //Start the windows message loop
    MSG msg;
    memset(&msg, 0, sizeof(MSG));

    //Keep track of the time
    DWORD startTime = GetTickCount();

    while(msg.message != WM_QUIT)
    {
        if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
        {
            //If there's a message, deal with it and send it onward
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else    //Otherwise update the game
        {
            //Calculate the delta time
            DWORD t = GetTickCount();
            float deltaTime = (t - startTime) * 0.001f;
```

```
            //Update the application
            app.Update(deltaTime);

            //Render the application
            app.Render();

            startTime = t;
        }
    }

    //Release all resources
    app.Cleanup();

    //... and Quit!
    return (int)msg.wParam;
}
```

As you can see, I create an instance of the `Application` class at the beginning of the `WinMain()` function. The `Application` instance is then updated and rendered each frame as long as the message loop runs (no `WM_QUIT` message has been received). Finally, the `Cleanup()` function is called, releasing any resources tied up by the application before the `WinMain()` function returns and the program terminates.

Next let's take a look at what happens in the `Init()` function of the `Application` class!

Creating the Window

In order to display your 3D world to the user, you first need to create a window. This window can work like any other window running under Windows; you can resize it, minimize, maximize, etc. The only thing I will use the window for in these examples is to display the rendered 3D world each frame.

So to create a window you first need to create and register a window class by filling out the `WNDCLASS` structure as shown:

```
//Create Window Class
WNDCLASS wc;
memset(&wc, 0, sizeof(WNDCLASS));

//Window Style
wc.style         = CS_HREDRAW | CS_VREDRAW;

//Window Event Procedure (more on this later)
wc.lpfnWndProc   = (WNDPROC)WndProc;
```

```
//The Application Instance
wc.hInstance     = hInstance;

//The Window Class Name
wc.lpszClassName = "D3DWND";

//...Finally Register the new Window Class
RegisterClass(&wc);
```

There are of course a multitude of options available when registering a window class. The code here just shows the minimum code required to get up and running. Check out the Microsoft Developer Network (MSDN) for more info on how to create windows, etc. One thing I need to explain here though is the `lpfnWndProc` variable. The `lpfn` prefix stands for Long Pointer Function, or in other words a function pointer. The window procedure handles all incoming events to the window, and the user can specify what should happen at each event. The light-weight window procedure used in this example looks like this:

```
LRESULT CALLBACK WndProc(HWND hwnd,
                         UINT msg,
                         WPARAM wParam,
                         LPARAM lParam)
{
    //User specified events
    switch( msg )
    {
        case WM_CREATE:
            //Do some window initialization here
            break;

        case WM_DESTROY:
            //Do some window cleanup here
            PostQuitMessage(0);
            break;
    }

    //Default events
    return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Only the `WM_CREATE` and `WM_DESTROY` functions are being handled in this window procedure. (See the MSDN for other events you can catch and handle in the window procedure function.) Then I return the result of the `DefWindowProc()` function,

which is basically the default procedure for all window events. After registering your window class and assigning it a window procedure, you can create an instance of this window type with the `CreateWindow()` function:

```
m_mainWindow = CreateWindow("D3DWND",        //Window class to use
                            "Window Title",  //Title
                            WS_EX_TOPMOST,   //Style
                            0,               //X
                            0,               //Y
                            WINDOW_WIDTH,    //Width
                            WINDOW_HEIGHT,   //Height
                            NULL,            //Parent window
                            NULL,            //Menu
                            hInstance,       //Application instance
                            0);              //Param

//Display the new window
ShowWindow(m_mainWindow, SW_SHOW);

//Update it
UpdateWindow(m_mainWindow);
```

That's all there is to it. You now have a window running and being updated. There are plenty of resources and tutorials available on the Web about Win32 application programming. For this book it is enough if you have a general understanding about how to create a window, the windows main loop and the window procedure, etc. That pretty much takes care of the windows code…next up, how to set up Direct3D!

Basic Rendering

In this section I'll cover how to set up the Direct3D device and get something drawn to the screen. The Direct3D Device is the interface you will use to draw objects to the screen, create resources, and much more.

Since I do expect you to have some experience with Direct3D before tackling this book, I will keep this section brief. Refer instead to the DirectX SDK documentation or one of the many introductory books available if something is unclear. For an introductory book on Direct3D game programming, I recommend Frank Luna's *Introduction to 3D Game Programing with Direct X 9.0c: A Shader Approach* [Luna06].

Creating the DirectX Device

Initializing the Direct3D device is done with the following steps:

1. Create the Direct3D interface.
2. Fill out the D3DPRESENT_PARAMETERS structure.
3. Create the Direct3D Device.

Here's the code for these steps:

```
//Create IDirect3D9 Interface
IDirect3D9* d3d9 = Direct3DCreate9(D3D_SDK_VERSION);

if(d3d9 == NULL)
{
    //Could not create the Direct3D interface, exit...
}

//Set D3DPRESENT_PARAMETERS
D3DPRESENT_PARAMETERS present;
present.BackBufferWidth            = WINDOW_WIDTH;
present.BackBufferHeight           = WINDOW_HEIGHT;
present.BackBufferFormat           = D3DFMT_A8R8G8B8;
present.BackBufferCount            = 2;
present.MultiSampleType            = D3DMULTISAMPLE_NONE;
present.MultiSampleQuality         = 0;
present.SwapEffect                 = D3DSWAPEFFECT_DISCARD;
present.hDeviceWindow              = m_mainWindow;
present.Windowed                   = windowed;
present.EnableAutoDepthStencil     = true;
present.AutoDepthStencilFormat     = D3DFMT_D24S8;
present.Flags                      = 0;
present.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
present.PresentationInterval       = D3DPRESENT_INTERVAL_IMMEDIATE;

//Create the IDirect3DDevice9
d3d9->CreateDevice(D3DADAPTER_DEFAULT,  //Primary Gfx card
                   D3DDEVTYPE_HAL,      //Hardware rasterization
                   m_mainWindow,        //Window to use
                   D3DCREATE_HARDWARE_VERTEXPROCESSING, //HW verts
                   &present,            //Present parameters
                   &g_pDevice);         //Resulting device
```

```
if(g_pDevice == NULL)
{
    //Could not create the Direct3D Device, exit...
}

//Release IDirect3D9 interface (you don't need it anymore)
d3d9->Release();
```

This code will set up a Direct3D device, assuming that you have a graphics card that supports hardware rasterization, hardware vertex processing, the selected back buffer format, etc. I also store the finished Device as a global pointer so it can be accessible from classes other than the `Application` class.

The Device output is now connected to the window created earlier. So if you wanted to clear the background of the window to a certain color, you could use the following code:

```
// Clear the viewport
g_pDevice->Clear(
         0,               //Num rectangles to clear
         NULL,            //Rectangles to clear (NULL = whole screen)
         D3DCLEAR_TARGET, //Clear the render target
         0xffffffff,      //Color AARRGGBB (in this case White)
         1.0f,            //Clear Z-buffer to 1.0f
         0);              //Clear Stencil Buffer to 0
```

Direct3D Rendering Loop

The rendering loop of Direct3D is quite simple and is governed by three functions: `BeginScene()`, `EndScene()`, and `Present()`. Between the `BeginScene()` and `EndScene()` functions is where you can do your rendering/drawing, and once done you call the `Present()` function to show the result to the screen. The `Present()` function automatically takes care of the back buffer swapping, etc., so you don't have to worry about that.

```
// Clear the viewport
g_pDevice->Clear( ... );

// Begin the scene
if(SUCCEEDED(g_pDevice->BeginScene()))
{
    //Do your rendering here!
```

```
    // End the scene.
    g_pDevice->EndScene();

    //Present the result
    g_pDevice->Present(0, 0, 0, 0);
}
```

Passing zeros to all the `Present()` functions parameters displays the result to the entire window—full screen, that is, which is what you'd want. For more info see the DirectX SDK documentation. This has been a lot of initialization code to take in now. Just stay with me a little while longer and you'll have something actually showing on the screen.

Loading a Mesh

In this chapter I'll just load a static mesh so there's something to render to the now rather blank screen.

Meshes are stored and accessed through the `ID3DXMesh` interface, a class that you'll become more familiar with by the end of this book. For now, it is sufficient that you know it holds a mesh or a model.

Throughout this book, I'll use the .x format together with the mesh loading functions available in the D3DX library, and to load a static mesh I'll use the following function:

```
HRESULT D3DXLoadMeshFromX(
  LPCTSTR pFilename,                  //Filename
  DWORD Options,                      //Mesh option
  LPDIRECT3DDEVICE9 pD3DDevice,       //Direct3D device
  LPD3DXBUFFER * ppAdjacency,         //Mesh adjacency information
  LPD3DXBUFFER * ppMaterials,         //Materials
  LPD3DXBUFFER * ppEffectInstances,   //Effects
  DWORD * pNumMaterials,              //Number of materials
  LPD3DXMESH * ppMesh                 //Resulting mesh
);
```

Since the .x format can also contain embedded materials and even shader effects, the `D3DXLoadMeshFromX()` function also has parameters for returning these. The following code uses this function to load a mesh from the hard drive into your application:

```
//Pointer that will hold the loaded mesh
ID3DXMesh *pMesh = NULL;

//Load new mesh
ID3DXBuffer * adjacencyBfr = NULL;
ID3DXBuffer * materialBfr = NULL;
DWORD noMaterials = NULL;

if(FAILED(D3DXLoadMeshFromX("someMesh.X",
                            D3DXMESH_MANAGED,
                            g_pDevice,
                            &adjacencyBfr,
                            &materialBfr,
                            NULL,
                            &noMaterials,
                            &pMesh)))
{
    //Failed to load mesh... exit
}

D3DXMATERIAL *mtrls = (D3DXMATERIAL*)materialBfr->GetBufferPointer();

for(int i=0;i<(int)noMaterials;i++)
{
    if(mtrls[i].pTextureFilename != NULL)
    {
        //Material has texture!
        //Load the texture mtrls[i].pTextureFilename as well
    }
}

adjacencyBfr->Release();
materialBfr->Release();
```

You'll find the full code for loading and storing a mesh in the upcoming example in this chapter. Now that you have your "something to render," next you need to sort out your "how to render." Today this is done with vertex and pixel shaders.

Loading an Effect

I assume that you have some knowledge of how pixel and vertex shaders work. An effect is a collection of instructions of how to render a specific effect and can include both vertex and pixel shaders. On a high-level, an Effect file can contain one or

more Techniques, which each can contain one or more passes (some effects require the geometry to be rendered more than once). The Effect files are composed with the High Level Shading Language (HLSL), which is what I'll use throughout the book to create some of the more advanced effects. The following function loads and compiles an Effect file from your hard drive at run time:

```
HRESULT D3DXCreateEffectFromFile(
  LPDIRECT3DDEVICE9 pDevice,           //Direct3D device
  LPCTSTR pSrcFile,                    //File to compile
  CONST D3DXMACRO * pDefines,          //Optional macros
  LPD3DXINCLUDE pInclude,              //Optional includes
  DWORD Flags,                         //Compile flags
  LPD3DXEFFECTPOOL pPool,              //Pool for shared parameters
  LPD3DXEFFECT * ppEffect,             //Resulting effect
  LPD3DXBUFFER * ppCompilationErrors   //Compilation error
);
```

And here's the code that uses the `D3DXCreateEffectFromFile()` function:

```
//Load Effect
ID3DXBuffer *pErrorMsgs = NULL;

HRESULT hRes = D3DXCreateEffectFromFile(
                    g_pDevice,
                    "someEffect.fx",
                    NULL,
                    NULL,
                    D3DXSHADER_DEBUG,
                    NULL,
                    &pEffect,
                    &pErrorMsgs);

if(FAILED(hRes) && (pErrorMsgs != NULL))
{
    //Failed to create Effect
    MessageBox(NULL,
               (char*)pErrorMsgs->GetBufferPointer(),
               "Effect Error",
               MB_OK);
}
```

Now the effect has been loaded, compiled (hopefully without errors), and stored in the `ID3DXEffect` interface. I won't cover the syntax of HLSL in this book (since that would require a book in itself). Suffice it to say that HLSL is close enough to C syntax that you should have no problems understanding it even if you are new to it. Look at the example code and refer to the DirectX SDK documentation for more information. You can also find lots of tutorials online on the subject [Germishuys08].

Rendering a Mesh with an Effect

Finally, here's the point where you'll see something appear on the screen. Once you've created your transformation matrices (the world, the view, and the projection matrix), you need to upload these (and any other info you need as well) to the Effect. This is done like this:

```
//Calculate Transformation Matrices
D3DXMATRIX view, proj, world;
D3DXMatrixIdentity(&world);
D3DXMatrixLookAtLH(&view, ... );
D3DXMatrixPerspectiveFovLH(&proj, ... );

//Upload info to Effect
pEffect->SetMatrix("matW", &world);
pEffect->SetMatrix("matVP", &(view * proj));
pEffect->SetVector("lightPos", &lightPos);
```

This uploads the matrices to the Effect. You can similarly upload vectors, floats, etc. with the `SetVector()` and `SetFloat()` functions, respectively. Once you've uploaded all the information the Effect needs to render, you do the actual rendering:

```
pEffect->SetTechnique(hTech);
UINT numPasses = 0;
pEffect->Begin(&numPasses, NULL);
for(int i=0; i<numPasses; i++)
{
pEffect->BeginPass(i);
//Render Geometry Here
pEffect->EndPass();
}
pEffect->End();
```

There you go. The mesh you've loaded will now be rendered onto the screen.

Conclusions

In this chapter I covered all the necessary groundwork needed to create and run a Direct3D application. I'll admit that this is probably one of the briefest introductions of this topic ever written. However, if any of the topics covered in this chapter felt foreign, I suggest you read up on those before continuing. Another thing not covered by this book is the High Level Shading Language (HLSL), which is something you need to at least understand before continuing.

Hopefully you're still reading and are not too turned off by all the groundwork code covered in this chapter. From now on until the end of the book there'll be nothing but character animation on the table.

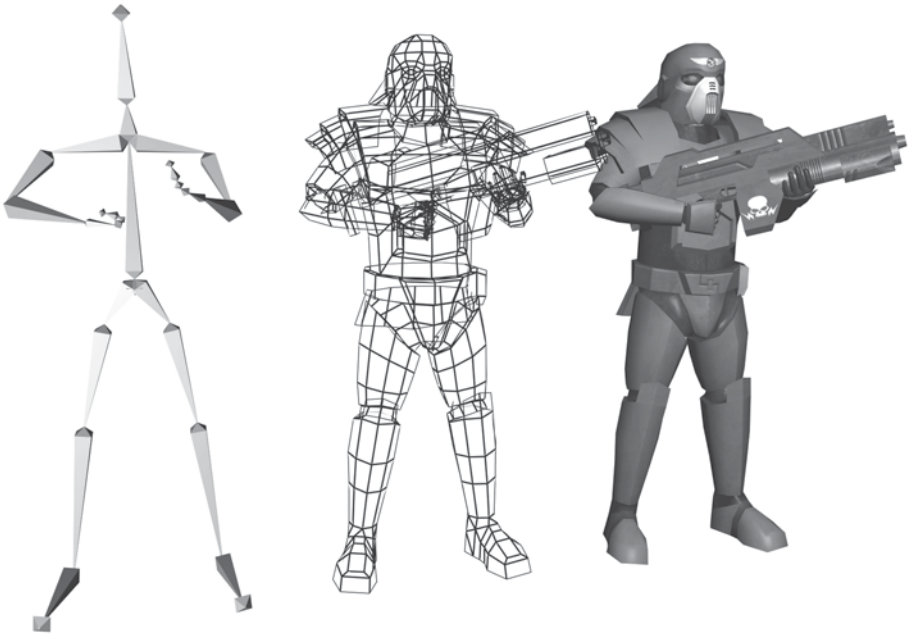
Further Reading

[Germishuys08] Germishuys, Pieter, "Basic HLSL Tutorials." Available online at http://www.pieterg.com/Tutorials/, 2008.

[Llopis03] Llopis, Noel, *C++ For Game Programmers.* Charles River Media, 2003.

[Luna06] Luna, Frank, *Introduction to 3D Game Programming with Direct X 9.0c: A Shader Approach.* Wordware Publishing, 2006.

3 Skinned Meshes



With the hip bone connected to the back bone,
and the back bone connected to the neck bone,
and the neck bone connected to the head bone,
Oh mercy how they scare!

-Dem Dry Bones, traditional spiritual

In the previous chapter you loaded and rendered a static mesh (the Soldier model). However, to create a character that you can animate, you first must give it some bones. In this chapter you will learn the basics of bone structures and, more importantly, how to skin a mesh to such a structure (called skinning). This type of structure is created from several bones linked together in a hierarchical fashion. A bone usually has a parent bone and zero or more child bones. Any transformations applied to a parent bone also affect its children (and their children, and so on). After you have built or loaded a hierarchical bone structure, you need to apply a mesh to it so that each vertex in the mesh is linked to one or more bones.

If you are looking at this topic for the first time, it might all seem a little confusing at the moment, but don't worry. It will all become clearer. In this chapter, you'll learn about the following:

- Basics of bone hierarchies
- Loading bone hierarchies from an .x file
- Software skinning
- Hardware skinning
- Rendering static meshes in a bone hierarchy

Skinned Meshes Overview

The concept of a skinned mesh is perhaps easiest understood by first looking at Figure 3.1. In it you see an arm in three different poses together with the underlying skeleton arm. As the muscles move a bone around a joint, the "flesh" will follow. This very simple idea is the essence of what you will try to accomplish in this chapter.

In computer graphics, the bones used to animate characters are just a helping structure, something that will never be rendered to the screen. Many fewer bones are needed for digital characters than for humans. Grown humans have over 200 bones, whereas the characters used in computer animation have 30 to 60 bones, depending on the level of control you need. You can see an example of a simplified bone hierarchy in the chapter cover image.

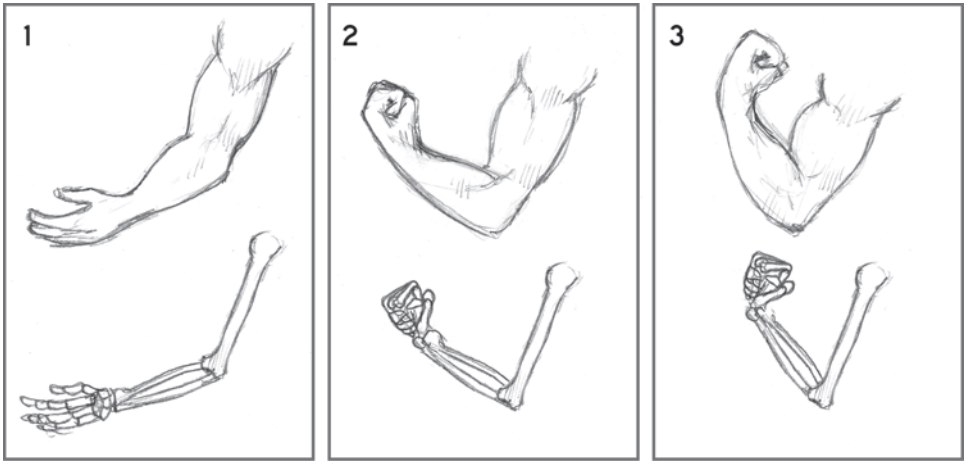


FIGURE 3.1
A human arm and its corresponding bones.

But before you see anything on the screen, there's quite a lot of behind-the-scenes work you need to tackle first. The first step is to create a bone hierarchy, either by creating it manually or, more commonly, by loading it from a 3D modeling program. After that, you need to skin the mesh to the bone hierarchy. Generally speaking, this can be done in two different ways, using either software skinning or hardware skinning. With software skinning, the new locations of the vertices are calculated by the central processing unit (CPU) each frame before the character is rendered. With hardware skinning, on the other hand, the vertices are calculated on-the-fly in the graphical processing unit (GPU) during render time. I'll cover both techniques in this chapter and look at the pros and cons of each. However, first things first; before you can render a skinned character, you need to create a bone hierarchy (regardless of which rendering approach you are aiming to take).

Bone Hierarchies

When attempting to understand what a bone hierarchy is and how it works, it helps to think of your own skeleton. Bone hierarchies work fundamentally in exactly the same way. Think of one of your many limbs. For instance, look back at the arm in Figure 3.1. Consider the three major joints in that arm: the shoulder, elbow, and wrist joints. These three joints are connected with two bones: the upper and the lower arm. In a bone hierarchy, the upper arm is the parent of the lower arm. This means

that whenever the upper arm is rotated around the shoulder joint, this transformation also affects the location of the lower arm and the hand (even though these haven't been rotated in relationship to their parents). This is something quite easy for you to test for yourself just by moving your own arm around.

If you remember some old games in the era of the first *Tomb Raider* game, these had one mesh object for each bone with the seams clearly showing, as shown in Figure 3.2.

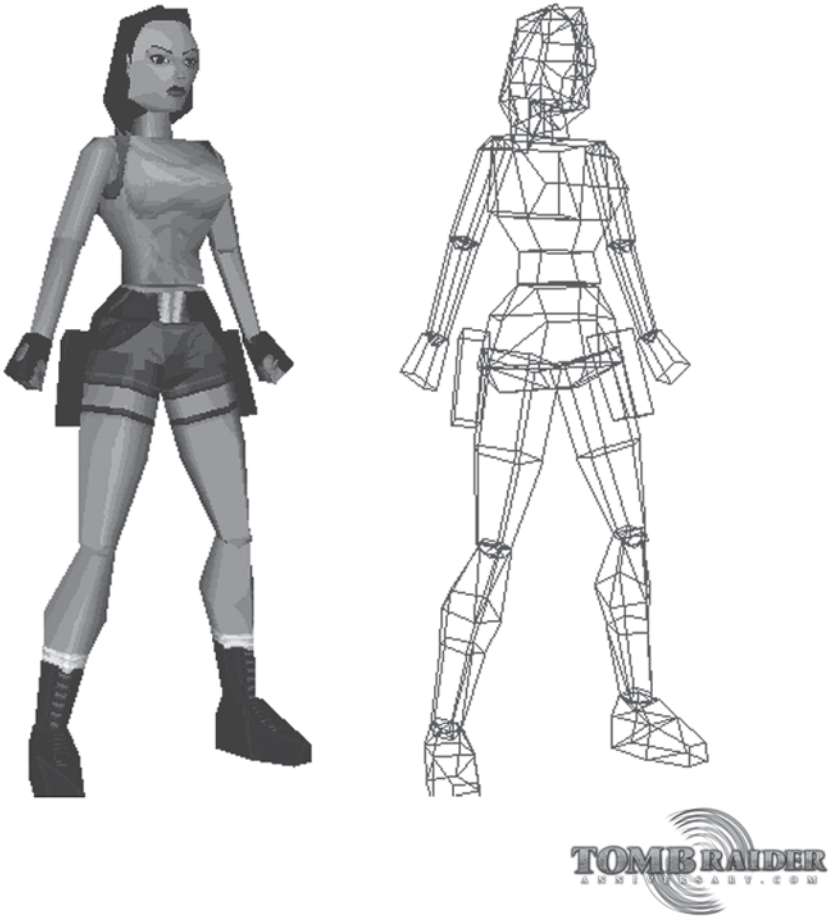


FIGURE 3.2
An early game character (Laura Croft) using a separate mesh for each bone. Notice the seams in the joints.

Back in the days when Laura Croft was taking her first steps, skinned meshes where a bit too heavy (calculation-wise) for the hardware of that time. But the idea of bone hierarchies hasn't changed much since those early days. In this chapter, however, the goal is to create a character whose mesh blends seamlessly even in the joint areas. So it's time to get cracking on building a bone hierarchy. But first the bone structure that will make up each of the individual bones in the hierarchy needs to be covered.

The D3DXFRAME Structure

In Direct3D, bones (and other similar linked hierarchies) are described using the `D3DXFRAME` structure, and it is with this structure that the complete bone hierarchy will be built.

```
struct D3DXFRAME {
    LPSTR Name;                             //Name of bone
    D3DXMATRIX TransformationMatrix;        //Local bone pos, rot & sca
    LPD3DXMESHCONTAINER pMeshContainer;     //Mesh connected to bone
    D3DXFRAME* pFrameSibling;               //Sibling bone pointer
    D3DXFRAME* pFrameFirstChild;            //First child bone
};
```

The `D3DXFRAME` structure contains a name, which, as you will see later, can help us find a specific bone in the hierarchy, such as a head bone, hand bone, etc. This can be useful if you want to attach different helmets, weapons, etc. to the head or hand of a character.

Each bone also has a transformation matrix describing its position, orientation, and scale in relation to its parent bone. This matrix describes transformations in local space only. This basically means that all position, rotation, and scale transformations are around the origin (coordinate 0,0,0) and not around the position where the bone will end up. If you rendered a skinned character using only the transformation matrices stored in the `TransformationMatrix`, you would end up with a big mess located at the origin of your world. Later on I'll extend the `D3DXFRAME` structure to also contain the finished world transformation matrix, and I'll cover how to calculate this.

Look again at the `D3DXFRAME` structure and you'll see that it also contains a `LPD3DXMESHCONTAINER` pointer, which in turn can contain a mesh. As with characters like the early Laura Croft, each bone would have an own mesh attached to it. The `D3DXFRAME` structure works like a linked list in which the sibling bones are all sharing the same parent. To better understand how a bone hierarchy can be built using the `D3DXFRAME` structure, look at Figure 3.3.

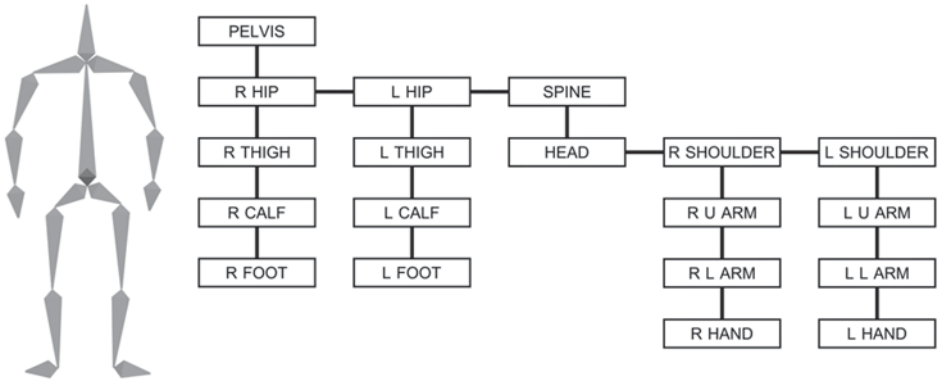


FIGURE 3.3
An example of a bone hierarchy built using the `D3DXFRAME` structure.

Vertical connections in Figure 3.3 describe the "first child" pointer, and the horizontal connections describe the "first sibling" pointer. As you can see, by using only these two pointers in each bone, you can describe very complex bone hierarchies. The top bone in Figure 3.3 (the pelvis) is the root node of the hierarchy. Whenever traversing a hierarchy like this, always start with the root node. The following code shows you how to traverse the whole hierarchy, passing through each and every bone/node.

```
void PrintHierarchy(D3DXFRAME *bone)
{
    //Print Bone Name
    cout << bone->Name;

    //Traverse Siblings
    if(bone->pFrameSibling != NULL)
        PrintHierarchy(bone->pFrameSibling);

    //Traverse Children
    if(bone->pFrameFirstChild != NULL)
        PrintHierarchy(bone->pFrameSibling);
}
```

Hierarchies like these are traversed easiest using recursive functions like the one in the example code. Make sure you understand how this function traverses the entire bone hierarchy before continuing. As an exercise, you can try to follow the function's path through the hierarchy in Figure 3.3 and write down in which order the bone names are printed.

The `PrintHierarchy()` function takes a pointer to a `D3DXFRAME` object as input, prints its name, and then calls the same function for any siblings and children that the `D3DXFRAME` object may have. Hopefully you have an idea by now of how complex skeletal structures can be built and traversed using only the child and sibling pointer per bone.

As another exercise, draw a 3D picture of a horse or a dog on a piece of paper. Next draw the major bones you would need to animate this animal. Now, decide which bone would be the root bone and draw the entire bone hierarchy on the paper in the same way as in Figure 3.3.

As stated earlier, the `D3DXFRAME` structure has only one transformation matrix containing the information of position, orientation, and scale of a certain bone. The transformation matrix of a bone is in *relation* to its parent. In most cases, the *actual* transformation matrix of a bone is what you need (i.e., the world matrix of the bone). The world matrix is the transformation matrix needed to render the bone at the correct location in the world. Therefore the `D3DXFRAME` structure will be extended to create the Bone structure as follows:

```
struct Bone: public D3DXFRAME
{
    D3DXMATRIX CombinedTransformationMatrix;
};
```

As you can see, the `Bone` structure inherits all the base components of the `D3DXFRAME` and adds only the `CombinedTransformationMatrix` variable. This matrix will contain the actual world transformation of a specific bone. For a moment, assume that all the `TransformationMatrix` variables in a bone hierarchy built with `Bone` objects contain valid transformation matrices. The following code then traverses the bone hierarchy and calculates the combined transformation matrices (i.e., world matrices) for all these bones.

```
void CalculateWorldMatrices(Bone* bone, D3DXMATRIX *parentMatrix)
{
    if(bone == NULL)
        return;

    //Calculate the combined transformation matrix
    D3DXMatrixMultiply(&bone->CombinedTransformationMatrix,
                       &bone->TransformationMatrix,
                       parentMatrix);
```

```
        //Perform the same calculation on all siblings...
        if(bone->pFrameSibling)
        {
            CalculateWorldMatrices((Bone*)bone->pFrameSibling,
                                   parentMatrix);
        }

        //... and all children
        if(bone->pFrameFirstChild)
        {
            //Note that we send a different parent matrix to
            //siblings and children!
            CalculateWorldMatrices((Bone*)bone->pFrameFirstChild,
                                   &bone->CombinedTransformationMatrix);
        }
    }
```

This function will be called on the root bone only with the world matrix of the entire character. The function will recursively traverse the entire bone hierarchy, filling the combined transformation matrix with the actual world matrix of the bone. Make sure you fully understand the last code snippet before continuing. Notice how for the siblings, the `CalculateWorldMatrices()` function call sends the same parent matrix, whereas for the child bones, the newly calculated combined matrix is passed as a parameter instead.

If this is your first time encountering linked lists and recursive function calls of this nature, it may seem a little confusing at the moment. I wish I could tell you it's about to get simpler. Unfortunately, nothing's really that simple when it comes to character animation.

I guess we can all agree that building complex hierarchies like these by hand in code would test the patience of even, well, the most patient man. A simpler and faster approach is to import a finished bone hierarchy from a 3D modeling program.

Loading a Bone Hierarchy

There are many different file formats that store mesh, bone, and animation information. In this book, I will stick to the .x file format—the native DirectX mesh format. There are several exporters for the most common 3D creation tools, as well as converters between the most common file formats that target the .x file format.

Anyway, assume that you have a skinned character complete with a bone hierarchy, skinning information, and animation. (If you don't, there are a few on the accompanying CD-ROM for you to practice with). To load a bone hierarchy, you

will have to use the ID3DXAllocateHierarchy interface. This is a very powerful interface, and, as you will see later on in the book, this interface can be used for more than just loading skinned meshes. The power of this interface lies in the fact that you have to implement its function yourself. Although this is a lot of work, in the end you'll see how this interface can be used to do a lot of different things while loading .x files. The ID3DXAllocateHierarchy interface has four functions you need to implement: CreateFrame(), CreateMeshContainer(), DestroyFrame(), and DestroyMeshContainer(), as detailed in the following sections.

The CreateFrame() Function

```
HRESULT CreateFrame(
  LPCSTR Name,                          //New bone name
  LPD3DXFRAME * ppNewFrame              //Location of the new bone
);
```

The CreateFrame() function takes a pointer to a string and a pointer to a D3DXFRAME object as input. If successful, the new D3DXFRAME object is created where the variable ppNewFrame is pointing.

The CreateMeshContainer() Function

```
HRESULT CreateMeshContainer(
  LPCSTR Name,                                  //Mesh name
  CONST D3DXMESHDATA * pMeshData,               //Mesh
  CONST D3DXMATERIAL * pMaterials,              //Material list
  CONST D3DXEFFECTINSTANCE * pEffectInstances, //Effects list
  DWORD NumMaterials,                           //Number materials
  CONST DWORD * pAdjacency,                     //Mesh adjacency array
  LPD3DXSKININFO pSkinInfo,                     //Mesh skinning info
  LPD3DXMESHCONTAINER * ppNewMeshContainer      //Mesh container output
);
```

As you can see, the CreateMeshContainer() function is a lot more complex than the CreateFrame() function. This method gets a name, mesh data, materials, effects, skinning information, and more as input. If successful, this function returns a newly created D3DXMESHCONTAINER object. Flip back a few pages and take a look at the D3DXFRAME structure, and see how this structure contains the pointer to a D3DXMESHCONTAINER object. As you may have realized, the ID3DXAllocateHierarchy reads both mesh and bone data from a file and creates a complete hierarchy using D3DXFRAME objects and D3DXMESHCONTAINER objects (or any user-defined structures that overload these—for instance, the Bone structure).

The DestroyFrame() Function

```
HRESULT DestroyFrame(
    LPD3DXFRAME pFrameToFree
);
```

This function is quite simple. In it you are supposed to release any resources tied up by a frame (`Bone`).

The DestroyMeshContainer() Function

```
HRESULT DestroyMeshContainer(
    LPD3DXMESHCONTAINER pMeshContainerToFree
);
```

As with the previous function, the `DestroyMeshContainer()` function is intended to release any resources tied up by a `D3DXMESHCONTAINER` object.

The ID3DXAllocateHierarchy

So why do you need to implement the functions defined by the `ID3DXAllocateHierarchy` yourself? Loading the hierarchies seems pretty straightforward. Why couldn't it simply handle it all? Well, the power lies in the fact that you often want to override both the `D3DXFRAME` and the `D3DXMESHCONTAINER` structures. As you've already seen, the `Bone` structure has been created inheriting from the `D3DXFRAME` structure. By implementing the four functions of the `ID3DXAllocateHierarchy` you can determine exactly what type of objects are created (and how they are initialized). To implement the `ID3DXAllocateHierarchy`, you simply create a new class inheriting from it. The class I'll use throughout this book is called `BoneHierarchyLoader` and is defined as follows:

```
class BoneHierarchyLoader : public ID3DXAllocateHierarchy
{
    public:
        STDMETHOD(CreateFrame)(
                        THIS_ LPCSTR Name,
                        LPD3DXFRAME *ppNewFrame);

        STDMETHOD(CreateMeshContainer)(
                        THIS_ LPCTSTR Name,
                        CONST D3DXMESHDATA * pMeshData,
                        CONST D3DXMATERIAL * pMaterials,
```

```
                        CONST D3DXEFFECTINSTANCE * pEffectInstances,
                        DWORD NumMaterials,
                        CONST DWORD * pAdjacency,
                        LPD3DXSKININFO pSkinInfo,
                        LPD3DXMESHCONTAINER * ppNewMeshContainer);

        STDMETHOD(DestroyFrame)(
                        THIS_ LPD3DXFRAME pFrameToFree);

        STDMETHOD(DestroyMeshContainer)(
                        THIS_ LPD3DXMESHCONTAINER pMeshContainerBase);
};
```

The `STDMETHOD` *macro used in the declaration of the member functions translate to:*

```
virtual HRESULT __stdcall
```

Virtual simply means the function can be overridden by inheriting classes, `HRESULT` *is the return type, and the* `__stdcall` *defines the calling convention with which the function is called. However, to make things easier and avoid this can of worms, you can simply use the* `STDMETHOD` *macro.*

Here's a look at a custom implementation of the `CreateFrame()` and `DestroyFrame()` functions. This implementation creates a `Bone` structure rather than a `D3DXFRAME` structure (remember that the `Bone` structure had the extra added `CombinedTransformationMatrix` member to store its world matrix).

```
HRESULT BoneHierarchyLoader::CreateFrame(LPCSTR Name,
                                         LPD3DXFRAME *ppNewFrame)
{
    Bone *newBone = new Bone;
    memset(newBone, 0, sizeof(Bone));

    //Copy name
    if(Name != NULL)
    {
        newBone->Name = new char[strlen(Name)+1];
        strcpy(newBone->Name, Name);
    }

    //Set the transformation matrices
    D3DXMatrixIdentity(&newBone->TransformationMatrix);
    D3DXMatrixIdentity(&newBone->CombinedTransformationMatrix);
```

```
        //Return the new bone...
        *ppNewFrame = (D3DXFRAME*)newBone;

        return S_OK;
    }

    HRESULT BoneHierarchyLoader::DestroyFrame(LPD3DXFRAME pFrameToFree)
    {
        if(pFrameToFree)
        {
            //Free Name String
            if(pFrameToFree->Name != NULL)
                delete [] pFrameToFree->Name;

            //Free Frame
            delete pFrameToFree;
        }

        pFrameToFree = NULL;

        return S_OK;
    }
```

For now, I'll just concentrate on creating the bones and not the mesh containers. Check out the custom version of the `CreateFrame()` function. Notice how a `Bone` object is created rather than a `D3DXFRAME` object? In this function, the members of the `D3DXFRAME` structure are initialized together with the added `CombinedTransformationMatrix` member. This way you can create your own bone structure and still use the `ID3DXAllocateHierarchy` interface when loading bone hierarchies from your .x files.

Looking at the `DestroyFrame()` function, you see that it releases only the `Name` of the bone since this was the only memory allocated that won't be automatically released before deleting the frame itself. However, if you create a more advanced bone structure inheriting from the `D3DXFRAME` structure, it is in the `DestroyFrame()` function that you have to release any extra resources allocated in the `CreateFrame()` function. Before covering a code example implementing a custom version of the `ID3DXAllocateHierarchy`, here's the D3DX function you will use to load an .x file:

```
HRESULT D3DXLoadMeshHierarchyFromX(
  LPCSTR Filename,
  DWORD MeshOptions,
  LPDIRECT3DDEVICE9 pDevice,
  LPD3DXALLOCATEHIERARCHY pAlloc,
  LPD3DXLOADUSERDATA pUserDataLoader,
  LPD3DXFRAME* ppFrameHierarchy,
  LPD3DXANIMATIONCONTROLLER* ppAnimController
);
```

The most notable parameters to this function are the `pAlloc`, `ppFrameHierarchy`, and the `ppAnimController`. The `pAlloc` is a pointer to a custom implemented `ID3DX-AllocateHierarchy` object (in this case, the `BoneHierarchyLoader` structure). The `ppFrameHierarchy` parameter contains the location where the root bone will be stored. The `ppAnimController` object is basically the structure used to animate a bone hierarchy. The `ID3DXAnimationController` interface will be covered in more detail in Chapters 4 and 5, where you will learn how to animate a character. For the other parameters see the DirectX documentation.

To contain and encapsulate all necessary data used for a skinned character, the following `SkinnedMesh` class will be created:

```
class SkinnedMesh
{
    public:
        SkinnedMesh();
        ~SkinnedMesh();
        void Load(char fileName[]);

    private:
        void UpdateMatrices(Bone* bone, D3DXMATRIX *parentMatrix);

        D3DXFRAME *m_pRootBone;
};
```

As you can see, this class doesn't contain much at the moment. Later, this class will be extended and expanded as more functionality is added to your skinned characters. The load function basically encapsulates the D3DX library function `D3DXLoadMeshHierarchyFromX()`. The `UpdateMatrices()` function was covered a few pages ago, when you learned how to calculate the combined transformation matrices of each bone in the hierarchy. Also, note that the only member in the `SkinnedMesh` class at the moment is a pointer to a `D3DXFRAME` object (which will later hold the root bone of the entire bone hierarchy). The following code shows the `SkinnedMesh` loading function:

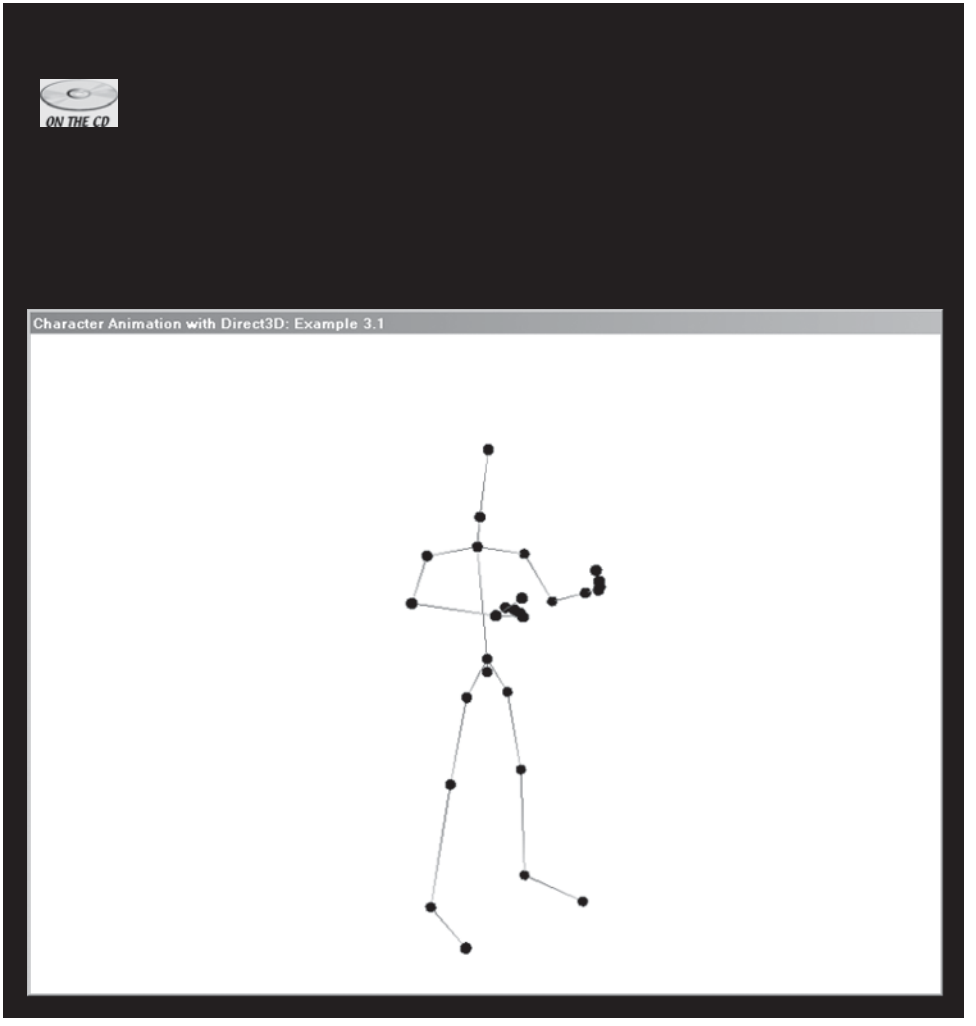
```
void SkinnedMesh::Load(char fileName[])
{
    BoneHierarchyLoader boneHierarchy;

    //Load a bone hierarchy from a file
    D3DXLoadMeshHierarchyFromX(fileName,
                               D3DXMESH_MANAGED,
                               pDevice,
                               &boneHierarchy,
                               NULL,
                               &m_pRootBone,
                               NULL);

    //Update all Bone transformation matrices
    D3DXMATRIX i;
    D3DXMatrixIdentity(&i);
    UpdateMatrices((Bone*)m_pRootBone, &i);
}
```

Sometimes it can be useful to locate a specific bone in a hierarchy—for example, if you would like to find the neck bone of a character and apply a rotation transformation matrix and make the head turn. The following D3DX function is then very useful:

```
LPD3DXFRAME D3DXFrameFind(
  CONST D3DXFRAME * pFrameRoot,   //The root bone
  LPCSTR Name                     //Name of bone you are looking for
);
```

This function returns a pointer to the correct bone in the hierarchy or returns `NULL` if the bone wasn't found. Try to use this function in Example 3.1 to find the neck bone.

Hopefully you know by now how to load a bone hierarchy by implementing the `ID3DXAllocateHierarchy` interface. Later on in the book, you'll see how you can use the same interface to load several different morph targets from a single .x file rather than keeping these meshes in separate files. However, for now it is time to actually apply a mesh to the bone hierarchy.



Applying a Mesh to the Bone Hierarchy

As you probably know, a mesh consists of several polygons that in turn consist of one or more triangles. Each triangle in turn is defined by three vertices—i.e., three points in 3D space. Before you look at how to skin a complex character mesh to a bone hierarchy, first just look at a single vertex. A vertex can be linked (influenced) by one or more bones in the bone hierarchy. The amount a bone influences a vertex is determined by a weight value as shown in Figure 3.4.

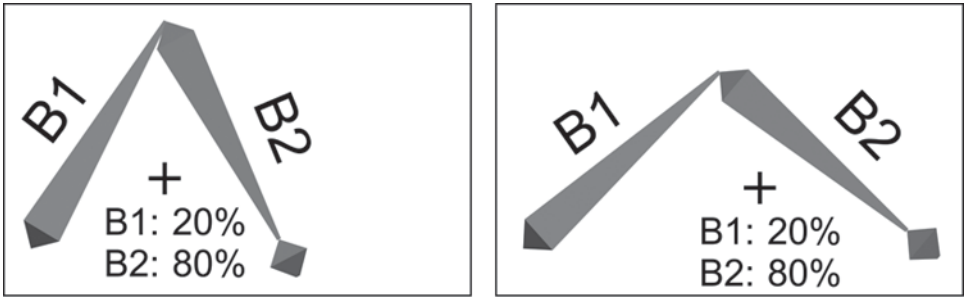


FIGURE 3.4
An example of how a vertex (the cross) is affected by two bones (B1 and B2) with the weights 20% and 80%, respectively. Notice how the vertex follows B2 more than B1 due to the weights.

It is important that the combined weights for a vertex equal 1.0 (100%).

In more mathematical terms, the transformation matrix applied to a vertex is defined as follows:

$$M_{Tot} = (w_0 M_0 + w_1 M_1 \ldots + W_n M_n)$$

This formula multiplies the bone weight (w_x) with the bone transformation matrix (M_x) for all influencing bones and sums up the result (M_{Tot}). The resulting matrix is then used to transform the vertex. In DirectX, the information about which bones influence which vertices, as well as their respective weights, etc., is stored and controlled with the `ID3DXSkinInfo` interface. One way of creating this interface is by using the following D3DX function:

```
HRESULT D3DXCreateSkinInfo(
  DWORD NumVertices,
  CONST D3DVERTEXELEMENT9 * pDeclaration,
  DWORD NumBones,
  LPD3DXSKININFO * ppSkinInfo
);
```

This function takes the amount of vertices in a mesh, their vertex declaration (i.e., what information each vertex contains), and the number of bones that will be used to skin this mesh. If you are making something in code that requires skinning, this would be the best approach. However, characters will most definitely be created and skinned in a 3D software such as 3D Studio Max, Maya, or similar. Luckily, when you export a character like this to the .x file format, the skinning information

is exported as well. If you look again at the `CreateMeshContainer()` function of the `ID3DXAllocateHierarchy` interface, you'll notice that one of the parameters to this function is indeed a pointer to a `ID3DXSkinInfo` object. So all you need to do when reading an .x file is to store this object and use it later when you skin a character.

Soon the `CreateMeshContainer()` function will be implemented, and through it the process of loading a character with bones and all will be complete. First, however, you need to understand the two major choices you have for how to render a skinned character. The first option is to do the skinning using the CPU—a.k.a. *software skinning*. The other option is to do the skinning directly with the GPU (graphics processing unit, i.e., the graphics card) as the mesh is being rendered—a.k.a. *hardware skinning*. (There are other variations of these two techniques, but these two are the major options).

Software Skinning Overview

With the first option, software skinning, the positions of each vertex in a mesh are calculated using the mathematical formula covered earlier. The result is stored in a temporary mesh that is then consequently rendered. Simple, straightforward, but also very slow compared to hardware skinning. So if it is so slow compared to hardware skinning, why use it? Well, the fact that the character is stored as a mesh in memory is the major upside of software skinning. With this temporary mesh, things like shadow casting, picking, etc., become a bit easier. With software skinning, it also doesn't matter how many bones are influencing a vertex. If you were making a first-person shooter (FPS) game, you might want to test to see whether or not a bullet you fired hit one of the enemy soldiers. With software skinning, this would be easy to test using a simple mesh–ray intersection test.

Hardware Skinning Overview

With hardware skinning, you can, of course, also do shadow casting, picking, etc., but then it requires a little more effort to get it to work. Hardware skinning also has some limits as to how many bones can influence a vertex as well as how many bones you can have per character without having to split up the mesh into several parts. However, what you lose in functionality, you make up readily in speed. Remember to choose your skinning method based on the particular game you are making. In the following two sections, both software and hardware skinning will be looked at in more detail.

For a simple mesh–ray intersection test, check out the `D3DXIntersect()` *function in the D3DX library. See the DirectX documentation for more info.*

Software Skinning Implementation

Let's first look at software skinning, since this is the more straightforward and easier method to implement. Here's a brief overview of the steps needed to render a skinned mesh with software skinning:

1. (Optional) Overload `D3DXFRAME`
2. (Optional) Overload `D3DXMESHCONTAINER`
3. Implement the `ID3DXAllocateHierarchy` interface
4. Load a bone hierarchy and associated meshes, skinning information, etc., with the `D3DXLoadMeshHierarchyFromX()` function
5. For each frame, update the skeleton pose (i.e., the `SkinnedMesh::UpdateMatrices()` function)
6. Update the target mesh using `ID3DXSkinInfo::UpdateSkinnedMesh()`
7. Render the target mesh as a common static mesh

Loading the Skinned Mesh

The first step on the path of skinning a mesh is to create your own mesh container structure. You do this by overloading the `D3DXMESHCONTAINER` structure defined as follows:

```
struct D3DXMESHCONTAINER {
    LPSTR Name;
    D3DXMESHDATA MeshData;
    LPD3DXMATERIAL pMaterials;
    LPD3DXEFFECTINSTANCE pEffects;
    DWORD NumMaterials;
    DWORD * pAdjacency;
    LPD3DXSKININFO pSkinInfo;
    D3DXMESHCONTAINER * pNextMeshContainer;
}
```

The `D3DXMESHCONTAINER` contains the mesh itself (in the `D3DXMESHDATA` structure) as well as all the necessary stuff needed to render the mesh (materials, textures, and shaders). The texture filenames are stored as a member of the `D3DXMATERIAL` structure and must be loaded separately. Another notable member of this structure is the `pSkinInfo` variable, which will contain skinning information for any skinned meshes loaded. There are, however, some things that we want to add to this structure to make it easier to render the mesh using software skinning. Therefore I've created the `BoneMesh` structure, defined as follows:

```
struct BoneMesh: public D3DXMESHCONTAINER
{
    ID3DXMesh* OriginalMesh;              //Reference mesh
    vector<D3DMATERIAL9> materials;       //List of materials
    vector<IDirect3DTexture9*> textures; //List of textures

    DWORD NumAttributeGroups;             //Number attribute groups
    D3DXATTRIBUTERANGE* attributeTable;  //Attribute table
    D3DXMATRIX** boneMatrixPtrs;          //Pointers to bone matrices
    D3DXMATRIX* boneOffsetMatrices;       //Bone offset matrices
    D3DXMATRIX* currentBoneMatrices;      //Current bone matrices
};
```

As you can see, quite a lot of extra information has been added to the `BoneMesh` structure compared to what was added in the `Bone` structure. The first three members should be quite easy to understand; the others may seem a little bit more obscure. Table 3.1 provides more details about the members.

TABLE 3.1 THE BONEMESH MEMBERS

Now that you've seen the overloaded `D3DXMESHCONTAINER` structure, take a look at how the `CreateMeshContainer()` of the `ID3DXAllocateHierarchy` is implemented to load a mesh into a `BoneMesh` object.

```
HRESULT BoneHierarchyLoader::CreateMeshContainer(
                            LPCSTR Name,
                            CONST D3DXMESHDATA *pMeshData,
                            CONST D3DXMATERIAL *pMaterials,
                            CONST D3DXEFFECTINSTANCE *pEffectInstances,
                            DWORD NumMaterials,
                            CONST DWORD *pAdjacency,
                            LPD3DXSKININFO pSkinInfo,
                            LPD3DXMESHCONTAINER *ppNewMeshContainer)
{
    //Create new Bone Mesh
    BoneMesh *boneMesh = new BoneMesh;
    memset(boneMesh, 0, sizeof(BoneMesh));

    //Get mesh data
    boneMesh->OriginalMesh = pMeshData->pMesh;
    boneMesh->MeshData.pMesh = pMeshData->pMesh;
    boneMesh->MeshData.Type = pMeshData->Type;

    //Add Reference so the mesh is not deallocated
    pMeshData->pMesh->AddRef();

    //To be continued...
```

First, a new `BoneMesh` object is created and all its members are set to zero and `NULL` using the `memset()` function. Next, a reference is added to the mesh data for both the `OriginalMesh` and the `MeshData` member. You need to keep a copy of the mesh in its original form when you do software skinning (more about this when the rendering of the mesh is covered).

```
    IDirect3DDevice9 *pDevice = NULL;
    pMeshData->pMesh->GetDevice(&pDevice);    //Get pDevice ptr

    //Copy materials and load textures (just like with a static mesh)
    for(int i=0;i<NumMaterials;i++)
    {
        D3DXMATERIAL mtrl;
        memcpy(&mtrl, &pMaterials[i], sizeof(D3DXMATERIAL));
        boneMesh->materials.push_back(mtrl.MatD3D);
        IDirect3DTexture9* newTexture = NULL;
```

```
    if(mtrl.pTextureFilename != NULL)
    {
        char textureFname[200];
        strcpy(textureFname, "meshes/");
        strcat(textureFname, mtrl.pTextureFilename);

        //Load texture
        D3DXCreateTextureFromFile(pDevice,
                                  textureFname,
                                  &newTexture);
    }

    boneMesh->textures.push_back(newTexture);
}

//To be continued again...
```

In this section of the `CreateMeshContainer()` function, you first get a pointer to the current device. After that, you copy all the materials over to the `BoneMesh` structure, and if necessary any textures needed are loaded with the associated materials (this is why you needed to retrieve the device pointer). Next, the skinning information sent as a parameter to the `CreateMeshContainer()` will have to be stored.

```
if(pSkinInfo != NULL)
{
    //Get Skin Info
    boneMesh->pSkinInfo = pSkinInfo;

    //Add reference so SkinInfo isn't deallocated
    pSkinInfo->AddRef();

    //Clone mesh and store in boneMesh->MeshData.pMesh
    pMeshData->pMesh->CloneMeshFVF(D3DXMESH_MANAGED,
                                   pMeshData->pMesh->GetFVF(),
                                   pDevice,
                                   &boneMesh->MeshData.pMesh);

    //Get Attribute Table
    boneMesh->MeshData.pMesh->GetAttributeTable(
                        NULL, &boneMesh->NumAttributeGroups);

    boneMesh->attributeTable =
             new D3DXATTRIBUTERANGE[boneMesh->NumAttributeGroups];
```

```
            boneMesh->MeshData.pMesh->GetAttributeTable(
                                        boneMesh->attributeTable, NULL);

            //Create bone offset and current matrices
            int NumBones = pSkinInfo->GetNumBones();
            boneMesh->boneOffsetMatrices = new D3DXMATRIX[NumBones];
            boneMesh->currentBoneMatrices = new D3DXMATRIX[NumBones];

            //Get bone offset matrices
            for(int i=0;i < NumBones;i++)
                boneMesh->boneOffsetMatrices[i] =
                        *(boneMesh->pSkinInfo->GetBoneOffsetMatrix(i));
        }

        //Set ppNewMeshContainer to the newly created boneMesh container
        *ppNewMeshContainer = boneMesh;

        return S_OK;
    }
```

In this last part of the `CreateMeshContainer()` function, you check whether there's any skinning info available. If so, make a clone of the mesh stored in the `pMeshData` member of your `BoneMesh` structure. This cloned mesh will later be the actual skinned mesh rendered. Also remember that a pointer to the original mesh (`OriginalMesh` member) is stored. This mesh will be used as a reference to create the skinned mesh stored in `pMeshData` each frame. In this piece of code, the number of attribute groups as well as the attribute table itself is stored. Then the matrix array is created according to how many bones are defined in the skinning information (note that you copy the bone offset matrix from the `ID3DXSkinInfo` object). Lastly, you simply store the created `BoneMesh` object and return `S_OK`.

The attribute table is stored using an array of `D3DXATTRIBUTERANGE` objects:

```
struct D3DXATTRIBUTERANGE {
    DWORD AttribId;        //which material/texture to use
    DWORD FaceStart;       //Face start
    DWORD FaceCount;       //Num of faces in this attribute group
    DWORD VertexStart;     //Vertex start
    DWORD VertexCount;     //Num of vertices in this attribute group
}
```

Later, when you render the mesh you loop through the attribute table, get the `AttribId`, and use this to set which material and which texture to use to render that subset of the mesh. This basically means that you can have several combinations of materials and textures when you render a character.

Rendering the Skinned Mesh with Software Skinning

Now you know how to load a bone hierarchy and any meshes that are attached to a bone using the extended `Bone` and `BoneMesh` structures as well as the `BoneHierarchyLoader`. Now is finally where the fun begins! Now you are finally coming to the point where you'll see a skinned character on the screen.

To render a `BoneMesh` we need to calculate the current matrices for all the influencing bones and store these in the `BoneMesh boneMatrixPtrs` array. So just after loading a mesh with the `D3DXLoadMeshHierarchyFromX()` function we call the following function to set up these matrix pointers:

```
void SkinnedMesh::SetupBoneMatrixPointers(Bone *bone)
{
    //Find all bones containing a mesh
    if(bone->pMeshContainer != NULL)
    {
        BoneMesh *boneMesh = (BoneMesh*)bone->pMeshContainer;

        //For the bones with skinned meshes, set up the pointers
        if(boneMesh->pSkinInfo != NULL)
        {
            //Get num bones influencing this mesh
            int NumBones = boneMesh->pSkinInfo->GetNumBones();

            //Create an array of pointers with numBones pointers
            boneMesh->boneMatrixPtrs = new D3DXMATRIX*[NumBones];

            //Fill array
            for(int i=0;i < NumBones;i++)
            {
                //Find influencing bone by name
                Bone *b = (Bone*)D3DXFrameFind(
                                m_pRootBone,
                                boneMesh->pSkinInfo->GetBoneName(i));
```

```
                //...and store pointer to it in the array
                if(b != NULL)
                {
                    boneMesh->boneMatrixPtrs[i] =
                            &b->CombinedTransformationMatrix;
                }
                else
                {
                    boneMesh->boneMatrixPtrs[i] = NULL;
                }
            }
        }
    }

    //Traverse the rest of the hierarchy...
    if(bone->pFrameSibling != NULL)
        SetupBoneMatrixPointers((Bone*)bone->pFrameSibling);

    if(bone->pFrameFirstChild != NULL)
        SetupBoneMatrixPointers((Bone*)bone->pFrameFirstChild);
}
```

This function finds all the bones influencing a certain `BoneMesh` and stores a pointer to their `CombinedTransformationMatrix` (i.e., world matrix). So after a skeleton has been updated and put in a certain pose, these world matrices can be accessed through this array during the rendering of the character. Then, to update a mesh in software, you need to use the `ID3DXSkinInfo::UpdateSkinnedMesh()` function:

```
HRESULT UpdateSkinnedMesh(
  CONST D3DXMATRIX * pBoneTransforms,   //Bone transforms
  CONST D3DXMATRIX * pBoneInvTransposeTransforms,    //Not used
  LPCVOID pVerticesSrc,                 //Source mesh (OriginalMesh)
  PVOID pVerticesDst                    //Destination mesh (pMeshData)
);
```

The `pBoneInvTransposeTransforms` parameter may seem a little strange here. However, this is used only if you have vertices that have two position elements. In that case, this set of transform matrices is used on the second position element. In this book, however, you won't need to use this, so simply set this parameter to `NULL`.

I hope you remember the `SkinnedMesh` class that was used to encapsulate the bone hierarchy and the loading functions. Now a `Render()` function will be added to this class. In this function, the skinned mesh will be updated using the `UpdateSkinnedMesh()` function defined in the `ID3DXSkinInfo` interface. Then the mesh will be rendered as follows:

```
void SkinnedMesh::Render(Bone *bone)
{
    if(bone == NULL)
        bone = (Bone*)m_pRootBone;

    //If there is a mesh to render...
    if(bone->pMeshContainer != NULL)
    {
        BoneMesh *boneMesh = (BoneMesh*)bone->pMeshContainer;

        if (boneMesh->pSkinInfo != NULL)
        {
            // set up bone transforms
            int numBones = boneMesh->pSkinInfo->GetNumBones();
            for(int i=0;i < numBones;i++)
                D3DXMatrixMultiply(&boneMesh->currentBoneMatrices[i],
                                   &boneMesh->boneOffsetMatrices[i],
                                   boneMesh->boneMatrixPtrs[i]);

            //Update the skinned mesh
            BYTE *src = NULL, *dest = NULL;
            boneMesh->OriginalMesh->LockVertexBuffer(
                                   D3DLOCK_READONLY, (VOID**)&src);
            boneMesh->MeshData.pMesh->LockVertexBuffer(
                                   0, (VOID**)&dest);

            boneMesh->pSkinInfo->UpdateSkinnedMesh(
                           boneMesh->currentBoneMatrices,
                           NULL, src, dest);

            boneMesh->MeshData.pMesh->UnlockVertexBuffer();
            boneMesh->OriginalMesh->UnlockVertexBuffer();
```

```
                //Render the mesh
                for(int i=0;i < boneMesh->NumAttributeGroups;i++)
                {
                    int mtrl = boneMesh->attributeTable[i].AttribId;
                    pDevice->SetMaterial(&(boneMesh->materials[mtrl]));
                    pDevice->SetTexture(0, boneMesh->textures[mtrl]);
                    boneMesh->MeshData.pMesh->DrawSubset(mtrl);
                }
            }
        }

        //Render Siblings & Children
        if(bone->pFrameSibling != NULL)
            Render((Bone*)bone->pFrameSibling);

        if(bone->pFrameFirstChild != NULL)
            Render((Bone*)bone->pFrameFirstChild);
    }
```

The `SkinnedMesh::Render()` function takes a `Bone` pointer as a parameter. If this `Bone` contains a `BoneMesh`, you render it, after which you call the `Render()` function on any child or sibling the bone may have. This way you traverse the entire bone hierarchy rendering any `BoneMesh` objects found along the way. First set up the current matrices of the `BoneMesh`, and then lock the vertex buffers of both the source mesh (`OriginalMesh`) and the destination mesh (`pMeshData`). After this, call the `UpdateSkinnedMesh()` function, which calculates the new position for each vertex in the mesh according to the current pose of the skeleton. After the mesh has been updated, render each of its subsets using the attribute table stored during the loading of the mesh.

Hardware Skinning Implementation

Once you have mastered (or at least somewhat understood) the process of software skinning, it's time to take a look at its less gentle cousin: hardware skinning. With hardware skinning, the mesh gets skinned on-the-fly in the GPU rather than pre-processed each frame as is the case with software skinning. Although a little bit trickier to implement than software skinning, hardware skinning is also considerably faster. Several things are done differently, but the main thing is, of course, the fact that you use a vertex shader to do your skinning calculations. Before you take a look at what information you need to supply the shader with, here are the steps required for hardware skinning.

1. (Optional) Overload `D3DXFRAME`
2. (Optional) Overload `D3DXMESHCONTAINER`
3. Implement the `ID3DXAllocateHierarchy` interface
4. Load a bone hierarchy and associated meshes, skinning information, etc., with the `D3DXLoadMeshHierarchyFromX()` function
5. Convert the mesh to an Index Blended Mesh
6. For each frame, update the skeleton pose (i.e., the `SkinnedMesh::UpdateMatrices()` function)
7. Upload the Matrix Palette (bone matrices) to the vertex shader
8. Render the Index Blended Mesh using the vertex shader

As you can see, most of the steps here are just the same as with software skinning. There are two new concepts here though: the Matrix Palette and an Index Blended Mesh. Before looking into the code for creating these, next is a quick look at the theory behind them.

The Matrix Palette

In software skinning, the array of current bone transformation matrices was used as a parameter to the `ID3DXSkinInfo::UpdateSkinnedMesh()` function. This function used the information stored in the `ID3DXSkinInfo` object to match all vertices with the bones influencing them as well as their corresponding weights. Now you have to take care of this calculation yourself in the vertex shader. The Matrix Palette is simply another name for the array of current bone transformations.

Unlike in software skinning, where there's no limit to the size of your Matrix Palette, in hardware skinning you are limited to a fixed number of bones, depending on the amount of vertex shader constants your graphics card supports. You can find this out by checking device caps as follows:

```
D3DCAPS9 caps;
d3d9->GetDeviceCaps(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, &caps);
int approxNumBones = caps.MaxVertexShaderConst / 4;
```

Remember that you will use a lot of constants for world, view, projection matrices and textures, etc. This means that the more additional constants you use for your rendering, the less you have available for the Matrix Palette. If you have a character with more than approxNumBones, then the mesh will be split into multiple parts and rendered in several passes.

Vertex Shader 2.0 supports 256 vertex constants, which in most cases is enough. However, as mentioned earlier, if you have a skinned mesh with a huge amount of bones, then it needs to be split up into parts before rendering. Luckily, this can be done in the same step as when a mesh is converted into an Index Blended Mesh.

Index Blended Meshes

Now that you know in principle how you build the Matrix Palette and what limitations the vertex shader constants bring, here's how you convert a mesh into an Index Blended Mesh.

Figure 3.5 shows how the `ID3DXSkinInfo::ConvertToIndexedBlendedMesh()` function converts a vertex into an Index Blended Vertex. It adds the bone weights and the bone indices as vertex elements. It also sets these indices to point to the correct bone transformation matrices in the Matrix Palette, as shown in Figure 3.6.

In this case the vertex is blended using four bones—namely, bones 3, 4, 6, and 11. The weight for each bone is stored in the vertex itself as an array of float values. To make this conversion, all you need to do is call the `ConvertToIndexedBlendedMesh()` function in your `CreateMeshContainer()` function.

```
HRESULT ConvertToIndexedBlendedMesh(
  LPD3DXMESH pMesh,                       //Input mesh
  DWORD Options,                          //Mesh Options
  DWORD paletteSize,                      //Max number of bones
  CONST DWORD * pAdjacencyIn,             //Input Adjacency Info
  LPDWORD pAdjacencyOut,                  //Output Adjacency Info
  DWORD * pFaceRemap,                     //Face remapping info
  LPD3DXBUFFER * ppVertexRemap,           //Vertex remapping info
  DWORD * pMaxVertexInfl,                 //Max num bones per vertex
  DWORD * pNumBoneCombinations,           //Num bones in combo table
  LPD3DXBUFFER * ppBoneCombinationTable, //Bone combo table
  LPD3DXMESH * ppMesh                     //Output Index Blended mesh
);
```

FIGURE 3.5
This figure shows how a vertex is converted to contain the necessary information for hardware skinning.

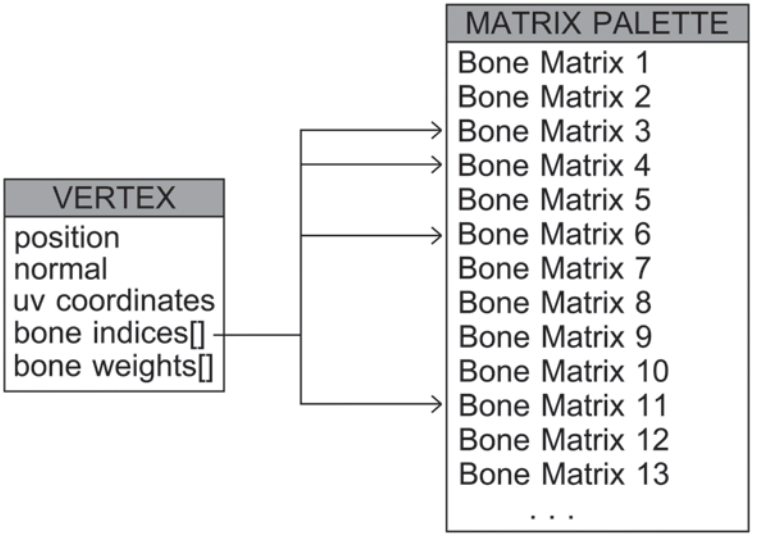


FIGURE 3.6
This figure shows the relationship between a single vertex and the Matrix Palette.

Some of these parameters you do not need to worry about. However, you do need to specify the max number of bones that can be used (i.e., the max size of the Matrix Palette). If the number of bones exceeds this value, the `ConvertToIndexedBlendedMesh()` function will split the mesh into multiple parts. For more info on this, check out the DirectX documentation. Following is an excerpt from the `CreateMeshContainer()` where the necessary changes have been made to accommodate hardware skinning:

```
...

DWORD maxVertInfluences = 0;
DWORD numBoneComboEntries = 0;
ID3DXBuffer* boneComboTable = 0;

pSkinInfo->ConvertToIndexedBlendedMesh(
                            pMeshData->pMesh,
                            D3DXMESH_MANAGED | D3DXMESH_WRITEONLY,
                            35,
                            NULL,
                            NULL,
                            NULL,
                            NULL,
                            &maxVertInfluences,
                            &numBoneComboEntries,
                            &boneComboTable,
                            &boneMesh->MeshData.pMesh);

//Bone Combination table not used
if(boneComboTable != NULL)
    boneComboTable->Release();

...
```

Well, that was fairly simple, wasn't it? The mesh has now been completely converted to an Index Blended Mesh with one simple function call. However, it is important that you understand what this function does, because now you'll come face to face with the actual vertex shader that performs the skinning.

The Skinning Vertex Shader

You'll now be introduced to the first proper shader in this book. I'm assuming that you know enough of the High Level Shading Language (HLSL) to read through this shader and understand it. If not, there are several good resources where you can start learning it—among them [Germishuys]. This excerpt covers just the vertex shader. See the example presented later on for the full effect code (.fx), including pixel shader code and how it is incorporated in the application code.

```
//The Matrix Palette
extern float4x4 MatrixPalette[35];

//This variable should be set by the application code depending
```

```
//on the max number of bones used by a certain mesh
extern int numBoneInfluences = 2;

//Vertex Input
struct VS_INPUT_SKIN
{
    float4 position : POSITION0;
    float3 normal   : NORMAL;
    float2 tex0     : TEXCOORD0;
    float4 weights  : BLENDWEIGHT0;
    int4   boneIndices : BLENDINDICES0;
};

//Vertex Shader Output / Pixel Shader Input
struct VS_OUTPUT
{
     float4 position : POSITION0;
     float2 tex0     : TEXCOORD0;
     float  shade    : TEXCOORD1;
};

VS_OUTPUT vs_Skinning(VS_INPUT_SKIN IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    float4 p = float4(0.0f, 0.0f, 0.0f, 1.0f);
    float3 norm = float3(0.0f, 0.0f, 0.0f);
    float lastWeight = 0.0f;
    int n = numBoneInfluences-1;
    IN.normal = normalize(IN.normal);

    //Blend vertex position & normal
    for(int i = 0; i < n; ++i)
    {
        lastWeight += IN.weights[i];
        p += IN.weights[i] *
             mul(IN.position, MatrixPalette[IN.boneIndices[i]]);

        norm += IN.weights[i] *
                mul(IN.normal, MatrixPalette[IN.boneIndices[i]]);
    }

    lastWeight = 1.0f - lastWeight;
```

```
    p += lastWeight *
         mul(IN.position, MatrixPalette[IN.boneIndices[n]]);

    norm += lastWeight *
            mul(IN.normal, MatrixPalette[IN.boneIndices[n]]);
    p.w = 1.0f;

    //Transform vertex to world space
    float4 posWorld = mul(p, matW);

    //... then to screen space
    OUT.position = mul(posWorld, matVP);

    //Copy UV coordinate
    OUT.tex0 = IN.tex0;

    //Calculate lighting
    norm = normalize(norm);
    norm = mul(norm, matW);
    OUT.shade = max(dot(norm, normalize(lightPos - posWorld)), 0.2f);

    return OUT;
}
```

This High Level Shading Language (HLSL) shader only supports a maximum of four bones influencing a single vertex. Notice the vertex input structure; see the weights and the bone indices and how they are used in the shader to index and weigh the bone matrices found in the Matrix Palette. Also note that you calculate the last weight manually. This is to make sure that the sum of the weights always adds up to 1.0f. Once the shader is in place, the only thing left is to edit the `SkinnedMesh::Render()` function so that it uses the vertex shader for skinning instead:

```
void SkinnedMesh::Render(Bone *bone)
{
    if(bone == NULL)bone = (Bone*)m_pRootBone;

    //If there is a mesh to render...
    if(bone->pMeshContainer != NULL)
    {
        BoneMesh *boneMesh = (BoneMesh*)bone->pMeshContainer;

        if (boneMesh->pSkinInfo != NULL)
        {
```

```
                // set up bone transforms
                int numBones = boneMesh->pSkinInfo->GetNumBones();
                for(int i=0;i < numBones;i++)
                {
                    D3DXMatrixMultiply(&boneMesh->currentBoneMatrices[i],
                                       &boneMesh->boneOffsetMatrices[i],
                                       boneMesh->boneMatrixPtrs[i]);
                }

                //Set HW Matrix Palette
                D3DXMATRIX view, proj;
                pEffect->SetMatrixArray(
                                  "MatrixPalette",
                                  boneMesh->currentBoneMatrices,
                                  boneMesh->pSkinInfo->GetNumBones());

                //Render the mesh
                for(int i=0;i < boneMesh->NumAttributeGroups;i++)
                {
                    int mtrl = boneMesh->attributeTable[i].AttribId;

                    pEffect->SetTexture("texDiffuse",
                                        boneMesh->textures[mtrl]);

                    D3DXHANDLE hTech = pEffect->GetTechniqueByName("Skinning");
                    pEffect->SetTechnique(hTech);
                    pEffect->Begin(NULL, NULL);
                    pEffect->BeginPass(0);

                    boneMesh->MeshData.pMesh->DrawSubset(mtrl);

                    pEffect->EndPass();
                    pEffect->End();
                }
            }
        }

        if(bone->pFrameSibling != NULL)
            Render((Bone*)bone->pFrameSibling);

        if(bone->pFrameFirstChild != NULL)
            Render((Bone*)bone->pFrameFirstChild);
    }
```

Not much has changed in this function compared to the software skinning example. Most notable, of course, is the use of the shader and uploading the Matrix Palette to it. Otherwise, you loop through the different attribute groups of the mesh and render it using the shader.

Rendering Static Meshes in Bone Hierarchies

Sometimes you might not want the character skinned. Making animated machinery is a prime example. Machinery rarely has "soft" parts; thus you don't really need to skin a mesh to make mechanical creations in your games. Nonetheless, you might want to have a bone hierarchy controlling the different parts of the "machine." Take the example of a robot arm, as shown in Figure 3.7.

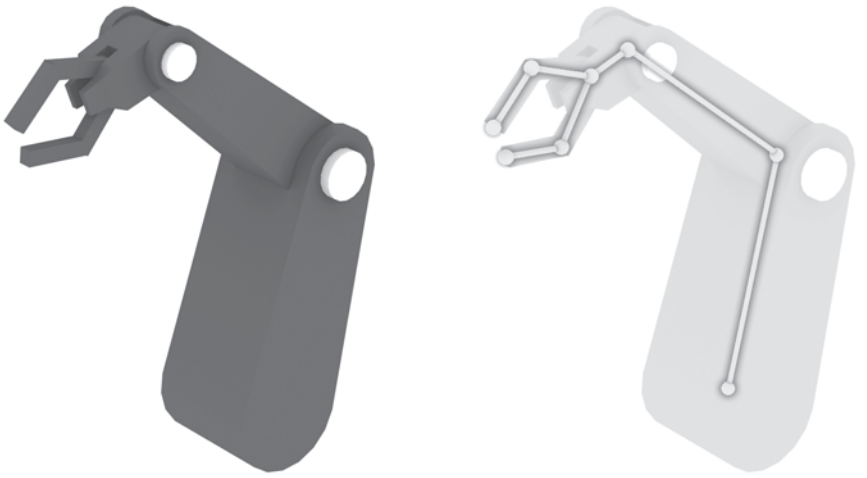


FIGURE 3.7
As you can see, each part of the robot arm is rigid and therefore does not require skinning.

Another case where you need rigid/solid objects is when they are combined with skinned meshes. In the previous examples of the skinned soldier, you may have noticed that he was missing both helmet and rifle. That's because these two objects have been rigid objects containing no skinning information. One way to include these objects would be to assign all vertices in them to one bone (the head bone, for example, in the case of the helmet). However, that would be a serious waste of CPU/GPU power.

In this section, you'll learn how to load and render both skinned meshes and static meshes from the same .x. file—although, to be frank, you have already covered the loading. Loading the meshes in the `CreateMeshContainer()` function is actually already done. So here's another high-level look at this function:

```
HRESULT BoneHierarchyLoader::CreateMeshContainer(...)
{
    //Create new Bone Mesh
      ...

    //Get mesh data here
      ...

    //Copy materials and load textures (like with a static mesh)
      ...
```

```
        if(pSkinInfo != NULL)
        {
            //Store Skin Info and convert mesh to Index Blended Mesh
              ...
        }

        //Set ppNewMeshContainer to newly created boneMesh container
          ...
    }
```

As you can see, you only convert the mesh to an Index Blended Mesh if the `pSkinInfo` parameter to this function is not `NULL`. But in the case of the helmet and the rifle for the soldier, the `pSkinInfo` parameter will of course be `NULL`, and as a result the mesh doesn't get converted. However, mesh data and the belonging materials and textures have still been copied. So all you really need to do is render them! And to do that you need only to add the case of rendering static meshes to the `SkinnedMesh::Render()` function.

```
    void SkinnedMesh::Render(Bone *bone)
    {
        if(bone == NULL)bone = (Bone*)m_pRootBone;

        //If there is a mesh to render...
        if(bone->pMeshContainer != NULL)
        {
            BoneMesh *boneMesh = (BoneMesh*)bone->pMeshContainer;

            if(boneMesh->pSkinInfo != NULL)
            {
                //Here's where the skinned mesh is rendered
                //only if the pSkinInfo variable isn't NULL
                ...
            }
            else
            {
                //Normal Static Mesh
                pEffect->SetMatrix("matW",
                                   &bone->CombinedTransformationMatrix);

                D3DXHANDLE hTech;
                hTech = pEffect->GetTechniqueByName("Lighting");
                pEffect->SetTechnique(hTech);
```

```
                //Render the static mesh
                for(int i=0;i < boneMesh->materials.size();i++)
                {
                    pEffect->SetTexture("texDiffuse",
                                        boneMesh->textures[i]);

                    pEffect->Begin(NULL, NULL);
                    pEffect->BeginPass(0);

                    boneMesh->OriginalMesh->DrawSubset(i);

                    pEffect->EndPass();
                    pEffect->End();
                }
            }
        }

        if(bone->pFrameSibling != NULL)
            Render((Bone*)bone->pFrameSibling);

        if(bone->pFrameFirstChild != NULL)
            Render((Bone*)bone->pFrameFirstChild);
    }
```

The static mesh is still locked to the bone hierarchy. As you can see, you use the combined transformation matrix of the bone to which the mesh is linked when you set the world matrix of the static mesh. So when you animate the neck bone of the character, the helmet will follow automatically. You can now use this code to render a robot character that has no skinned parts at all or a hybrid character like the soldier that has both skinned and static meshes.

Conclusions

Congratulations! If you're still reading, you've covered the meatiest chapter of the entire book. Hopefully you've managed to learn something along the way. It is a long process to attach a few vertices to a skeleton, isn't it?

At the end of this chapter you don't have much more to show for your work than you had in Chapter 2. Well, to be honest, it is in the next chapter that you will really experience the payoff—when you animate the skeleton (and with it the character).

Take time to look at each of the examples again; most likely, you'll learn a lot from playing with the code.

Chapter 3 Exercises

- Implement your own Skinned Mesh class, and support both hardware and software skinning with it.
- Check out the implementation of the character shadow in the software skinning examples. Implement it also for the hardware-skinned character.
- If you have access to 3D modeling software, create a skinned character, export it to the .x file format, and read it into your application.
- Access the Matrix Palette and multiply a small transformation (rotation/scale) to the neck bone's transformation matrix. Try to make the character turn his head.
- Study the `RenderSkeleton()` function in the `SkinnedMesh` class. Try also to visualize which bone has a `BoneMesh` attached to it.
- Implement your own version of the `Bone`, `BoneMesh`, and `BoneHierarchyLoader` classes. Add new members to these classes that you initialize in your own `CreateMeshContainer()` function.

Further Reading

[Ditchburn06] Ditchburn, Keith, "X File Hierarchy Loading." Available online at http://www.toymaker.info/Games/html/load_x_hierarchy.html, 2006.

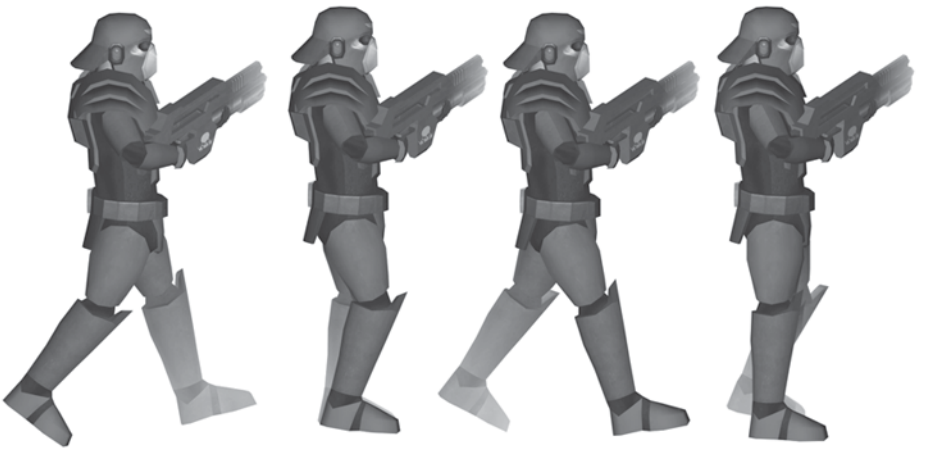
[Germishuys] Germishuys, Pieter, "HLSL Tutorials." Available online at http://www.pieterg.com/Tutorials.php.

[Jurecka04] Jurecka, Jason, "Working with the DirectX .X File Format and Animation in DirectX 9.0." Available online at http://www.gamedev.net/reference/articles/article2079.asp, 2004.

[Luna04] Luna, Frank, "Skinned Mesh Character Animation with Direct3D 9.0c." Available online at http://www.moon-labs.com/resources/d3dx_skinnedmesh.pdf, 2004.

[Taylor03], Taylor, Phil, "Modular D3D SkinnedMesh." Available online at http://www.flipcode.com/articles/article_dx9skinmeshmod.shtml, 2003.

4 Skeletal Animation



The previous chapter covered the basics of skinned meshes, as well as how to load them from an .x file. Apart from the added bone hierarchy, these meshes were still not animated and therefore not much more interesting to look at than a regular static mesh. That will change in this chapter, and you'll learn how to load animation data and apply it to a skinned mesh. This chapter covers the following:

- Keyframe animation basics
- Loading animation data
- The `ID3DXAnimationController`
- Having multiple controllers affecting the same mesh

Keyframe Animation

As you might know, a movie is made up of several still images running quickly and therefore creating the illusion of a moving picture. These still images are known as frames. Each frame has a certain place in time as well as a picture of how the "world" looks at this specific time step.

Keyframe animation has been around for quite some time. In fact, it was used in the very first TV cartoons, for example. The way it worked was that the senior animator would draw two images containing the poses of a cartoon character at two different time steps (these frames are the so-called keyframes). The senior animator would then give these keyframes to a junior animator and have him fill out the rest of the frames in between, a process also known as *tweening* (from "in-between-ing").

In many cases the keyframes are drawn by one artist in company A, and then the rest of the frames are drawn by another artist in company B (which might even be located in a completely different country). Each *Simpson's* episode, for example, is drawn mostly in India. What makes keyframing so powerful is that it can be applied to almost anything (see Figure 4.1).

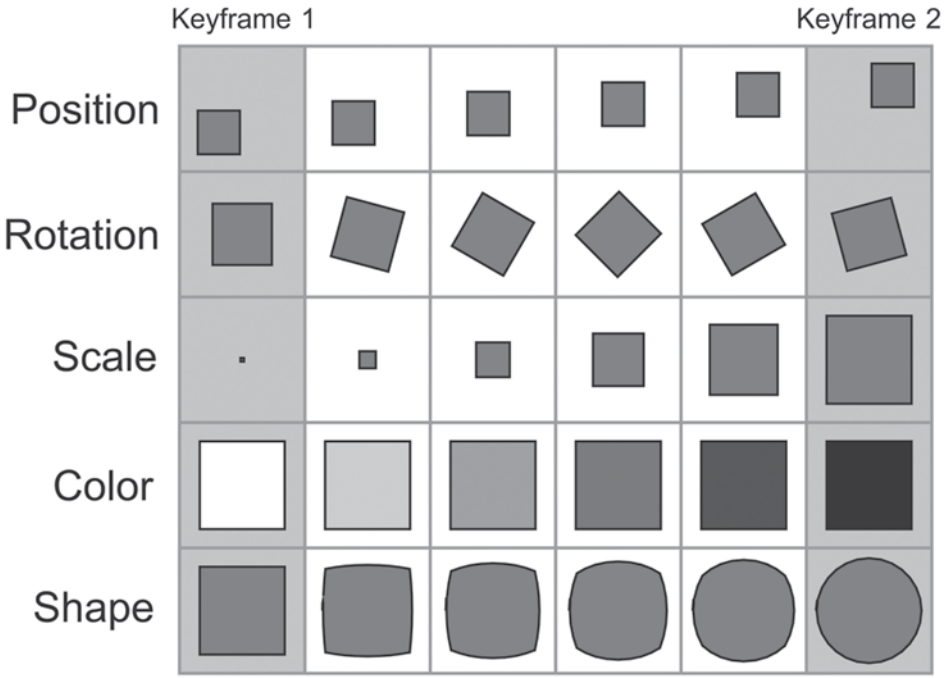


FIGURE 4.1
Several examples using the keyframing technique.

In Figure 4.1 the two keyframes are highlighted with gray background. Now take a minute and try to imagine what the little square would look like if all the transformations were applied at the same time across these two keyframes. In computer animation this technique is very powerful. Even if the time step varies in length and regularity (as the frame rate often does in games), this technique can still be used to calculate the current frame based on two keyframes.

DirectX uses these two structures to describe keyframes. The `D3DXKEY_VECTOR3` structure can describe translation and scale keyframes. Rotation, on the other hand, is described by the `D3DXKEY_QUATERNION` structure, since using Euler angles can result in a Gimbal lock. A Gimbal lock occurs when an object is rotated along one axis in such a way that it aligns two of the x, y, and z axes. As soon as this happens, one degree of freedom is lost and the condition can't be reversed no matter what rotation operation is performed on the object. Quaternions are a much safer option than Euler angles (although somewhat harder to comprehend). Anyhow, here are the two DirectX keyframe structures:

```
struct D3DXKEY_VECTOR3
{
    FLOAT Time;
    D3DXVECTOR3 Value;
};

struct D3DXKEY_QUATERNION
{
    FLOAT Time;
    D3DXQUATERNION Value;
};
```

As you can see, they both contain a timestamp as well as a value describing the translation, scale, or rotation of the object at that time. If you're not familiar with quaternions at the moment, don't worry; these will be looked into in more depth when you reach Chapter 6. The time of these key structures is in animation ticks, not in seconds. The amount of ticks an animation uses is equivalent to the time resolution used by the animation. Next, check out how to combine lots of these keyframes into an animation!

Animation Sets

Animation sets are simply collections of animations, where an animation is a collection of keyframes. Now that you know the theory behind keyframe animation, it is time to turn to the practical side of things. In this section you'll get familiar with the `ID3DXKeyframedAnimationSet` interface. This interface contains a lot of different functions, some of which will be used in this chapter. Others will be used later on when things like animation callbacks are covered. Check the DirectX documentation for the complete list of functions. To create an `ID3DXKeyframedAnimationSet` object, the following function is used:

```
HRESULT D3DXCreateKeyframedAnimationSet(
  LPCSTR pName,                                //Animation set name
  DOUBLE TicksPerSecond,                       //Ticks per second
  D3DXPLAYBACK_TYPE Playback,                  //Playback type
  UINT NumAnimations,                          //Num animations in set
  UINT NumCallbackKeys,                        //(more on this later)
  CONST LPD3DXKEY_CALLBACK * pCallKeys,        //(more on this later)
  LPD3DXKEYFRAMEDANIMATIONSET * ppAnimationSet //Output
);
```

The most interesting parameter here is the playback type, which can be one of the following: `D3DXPLAY_LOOP`, `D3DXPLAY_ONCE` or `D3DXPLAY_PINGPONG` (The ping-pong option will play the animation forward, then backward, and then start over). Once you have the empty animation set created, all you need to do is to fill it with some new keyframes, which you can do with the following function:

```
HRESULT RegisterAnimationSRTKeys(
  LPCSTR pName,                              //Animation name
  UINT NumScaleKeys,                         //Num scale keys
  UINT NumRotationKeys,                      //Num rotation keys
  UINT NumTranslationKeys,                   //Num translation keys
  CONST LPD3DXKEY_VECTOR3 * pScaleKeys,      //Scale keyframes
  CONST LPD3DXKEY_QUATERNION * pRotationKeys, //Rotation keyframes
  CONST LPD3DXKEY_VECTOR3 * pTranslationKeys, //Translation keyframes
  DWORD * pAnimationIndex                    //Resulting anim index
);
```

Easy! Arrays of scale, rotation, and translations keyframes were created (using the `D3DXKEY_VECTOR3` and the `D3DXKEY_QUATERNION` structures) and added to the animation set using this function. In action, these functions could be used in the following way:

```
//Create new Animation set
D3DXCreateKeyframedAnimationSet("AnimationSet1", 500,
D3DXPLAY_PINGPONG, 1, 0, NULL, &m_pAnimSet);

//Create Keyframes
D3DXKEY_VECTOR3 pos[3];
pos[0].Time = 0.0f;
pos[0].Value = D3DXVECTOR3(0.2f, 0.3f, 0.0f);
pos[1].Time = 1000.0f;
pos[1].Value = D3DXVECTOR3(0.8f, 0.5f, 0.0f);
pos[2].Time = 2000.0f;
pos[2].Value = D3DXVECTOR3(0.4f, 0.8f, 0.0f);

D3DXKEY_VECTOR3 sca[2];
sca[0].Time = 500.0f;
sca[0].Value = D3DXVECTOR3(1.0f, 1.0f, 1.0f);
sca[1].Time = 1500.0f;
sca[1].Value = D3DXVECTOR3(4.0f, 4.0f, 4.0f);

//Register Keyframes
m_pAnimSet->RegisterAnimationSRTKeys(
                          "Animation1", 2, 0, 3, sca, NULL, pos, 0);
```

This code creates an animation sequence with ping-pong playback, using both position and scale elements. To calculate the timestamp of a certain animation key, you need to retrieve the animation's amount of ticks per second. For that you can use the following function defined in the `ID3DXKeyframedAnimationSet` interface:

```
DOUBLE GetSourceTicksPerSecond();
```

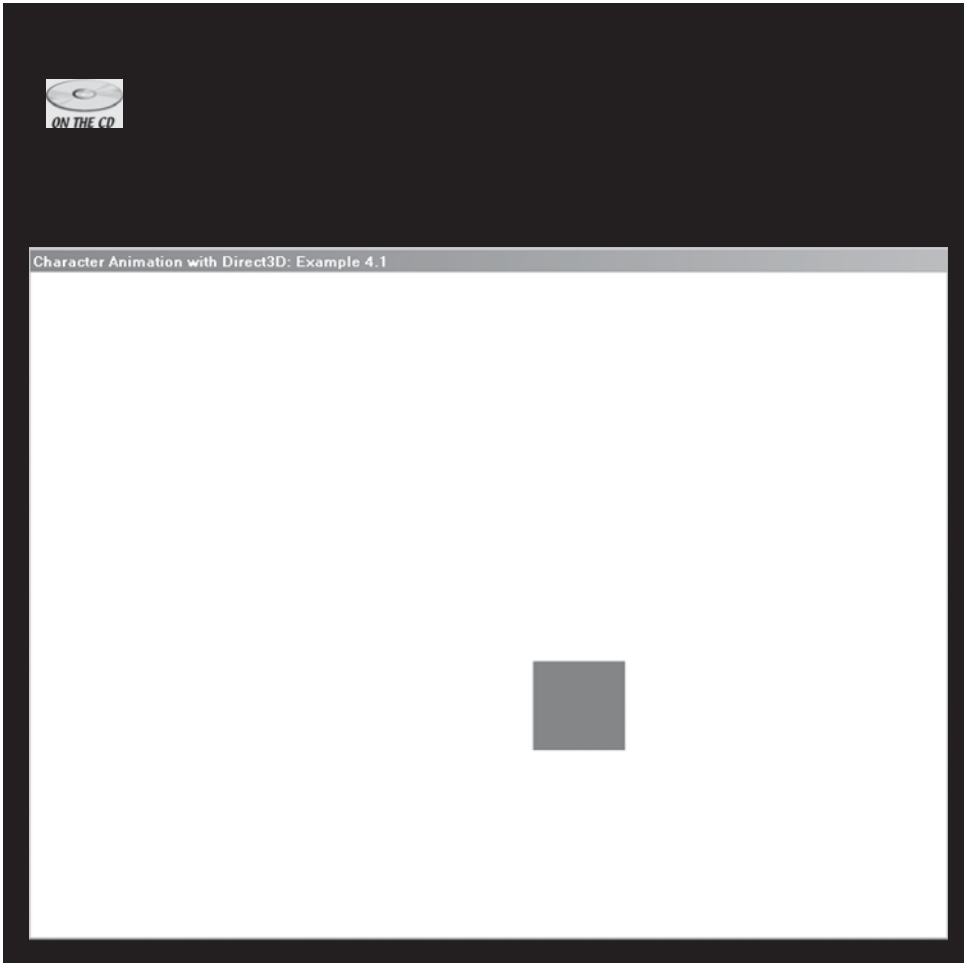
The function can be used like this to calculate the timestamp of a new animation key:

```
D3DXKEY_VECTOR3 aKey;
aKey.Value = D3DXVECTOR3(0.2f, 1.5f, -2.3f);
aKey.Time = 2.5f * aAnimSet->GetSourceTicksPerSecond();
```

This code creates a new position key and sets the time stamp of the key to 2.5 seconds. It is very seldom you need to manually create animation keys like this, but the knowledge of how to do so will come in handy in the next chapter when animation callback events are covered. Anyway, once an animation like this has been created, you need a way to read position, rotation, and scale data from the animation for any given time step. For this purpose you can use the following function:

```
HRESULT GetSRT(
  DOUBLE PeriodicPosition,        //Time step
  UINT Animation,                 //Animation index
  D3DXVECTOR3 * pScale,           //Scale output
  D3DXQUATERNION * pRotation,     //Rotation output
  D3DXVECTOR3 * pTranslation      //Translation output
);
```

This function takes a time step and an animation index as input (remember that an animation set can contain several animations). As output from this function, you get scale, rotation, and translation elements.



The ID3DXAnimationController Interface

Okay, you already know how to create a set of different animations. Why is an animation controller interface needed? Well, the `ID3DXAnimationController` interface controls all aspects of keyframed animation. It deals with anything from setting the active animation to blending multiple animations, animation callbacks, and so on (more on this in Chapter 5). In this chapter you'll learn how to obtain this interface as well as the functions needed to control the basic aspects of character animation.

Loading the Animation Data

You have already come in contact with the function used to load the `ID3DXAnimationController` object in the previous chapter. If you remember, the `D3DXLoadMeshHierarchyFromX()` function was used to load the bone hierarchy from an .x file. One of the output parameters from this function is an `ID3DXAnimationController` object containing all the animation data stored with the model. This data is loaded like this:

```
ID3DXAnimationController *m_pAnimControl = NULL;
D3DXFRAME *m_pRootBone = NULL;

D3DXLoadMeshHierarchyFromX("some_X_file.x",
                           D3DXMESH_MANAGED,
                           pDevice,
                           &someHierarchy,
                           NULL,
                           &m_pRootBone,
                           &m_pAnimControl);
```

This code loads the bone hierarchy and its meshes and stores in the `m_pRootBone` variable. It also loads the animation data affecting this bone hierarchy in the `m_pAnimControl` variable. It is now through this animation controller that you can set active animations for the character as well as update the active time, etc. The `ID3DXAnimationController` contains several animation sets (as covered in the previous section). The difference between these animation sets and those created earlier is that these are directly connected to the transformation matrices of the character bones. Here's how you would obtain any animations stored in an `ID3DXAnimationController`:

```
void SkinnedMesh::GetAnimations(vector<string> &animations)
{
        ID3DXAnimationSet *anim = NULL;

        for(int i=0;i<(int)m_pAnimControl-
>GetMaxNumAnimationSets();i++)
```

```
        {
            anim = NULL;
            m_pAnimControl->GetAnimationSet(i, &anim);

            if(anim != NULL)
            {
                    animations.push_back(anim->GetName());
                    anim->Release();
            }
        }
}
```

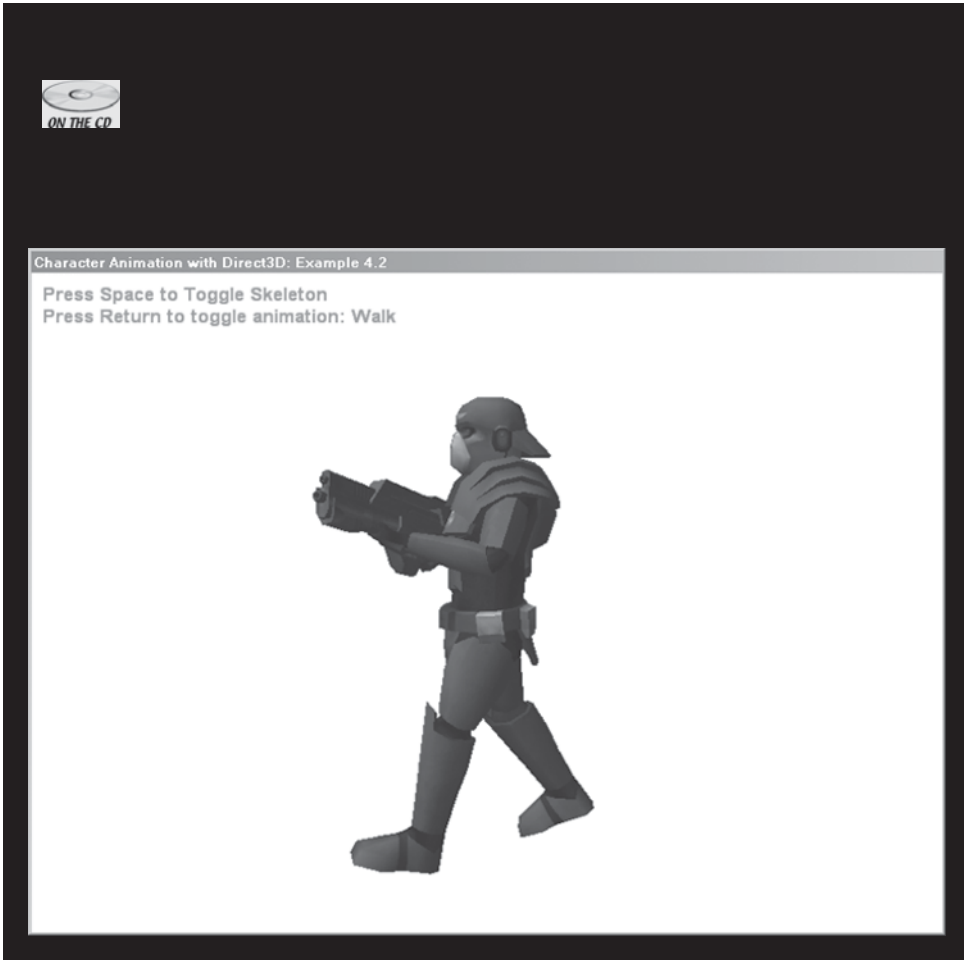
This function added to the `SKINNEDMESH` class fills a vector with all the names of the animation sets stored in the character's `ID3DXAnimationController`. First the `GetMaxNumAnimationSets()` function is used to query the number of animation sets, and then the `GetAnimationSet()` function can be used to get an actual animation set.

An `ID3DXAnimationController` object has several tracks, where each track is a slot for an animation set. This means that you can have several active animations at the same time, and even blend between them. The animation controller's tracks will be covered in more detail in the next chapter. For now, just assume that you have only one track with one active animation. In this case you set the active animation for a track using this function:

```
HRESULT SetTrackAnimationSet(
  UINT Track,                          //Track index
  LPD3DXANIMATIONSET pAnimSet          //Animation set
);
```

To set which animation to play, simply use the `GetAnimationSet()` function to retrieve the animation set, and then use the `SetTrackAnimationSet()` to activate it. Once you've set the animation set for one (or more) tracks, you're ready to start the actual animation. You can update/play the animations using the following function:

```
HRESULT AdvanceTime(
  DOUBLE TimeDelta,                 //Time to advance animation with
  LPD3DXANIMATIONCALLBACKHANDLER pCallbackHandler   //(next chapter)
);
```



The `AdvanceTime()` *function only updates the local transformation matrices of the bones. Remember that you need to update the combined transformation matrices for all the bones after you've called this function, since these are the ones used in the matrix palette. In the examples, this can be done by calling the* `UpdateMatrices()` *function in the* `SKINNEDMESH` *class.*

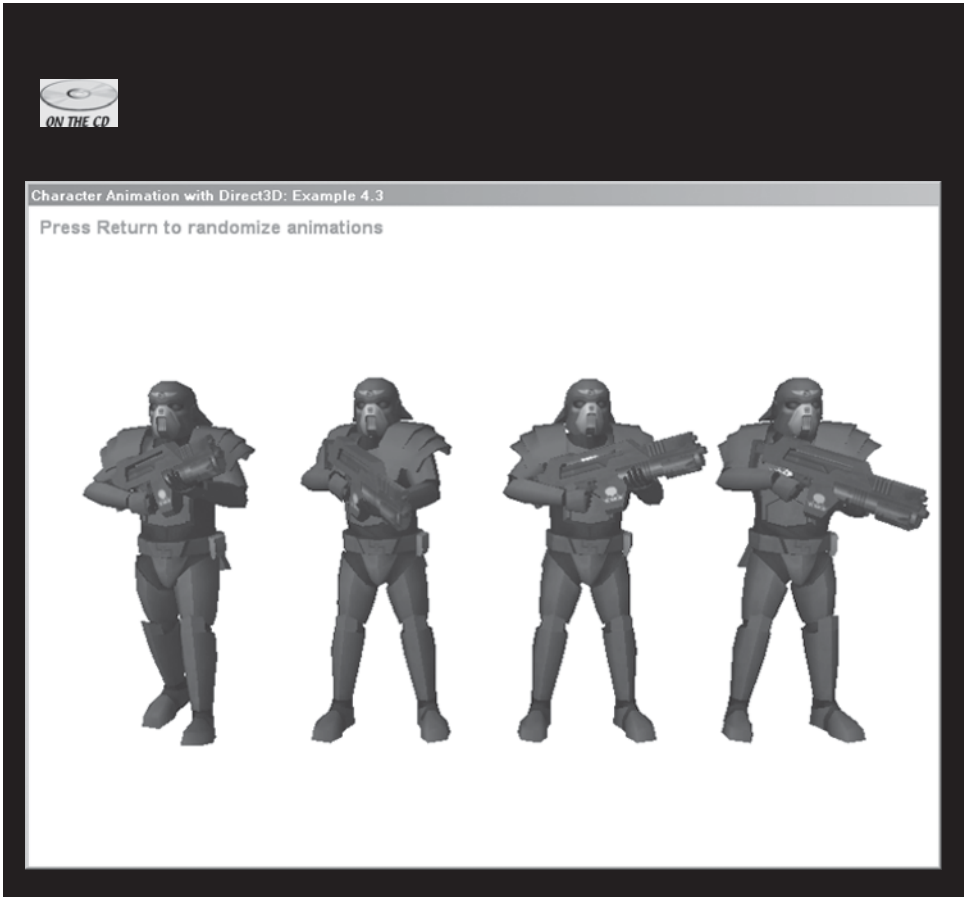
Multiple Animation Controllers

So far, so good. You have one mesh, one animation controller, and, all in all, one working character. What if you need two characters? Hmmm… The naïve way would be to have two meshes and two animation controllers. No real problem with that. However, what if you need an army? Clearly, having one mesh for each soldier instance wouldn't be the smartest approach. The solution lies in the fact that you can clone the character's animation controller using the following function:

```
HRESULT CloneAnimationController(
  UINT MaxNumAnimationOutputs,                //Num outputs (i.e. bones)
  UINT MaxNumAnimationSets,                   //Num animation sets
  UINT MaxNumTracks,                          //Num tracks
  UINT MaxNumEvents,                          //Num events
  LPD3DXANIMATIONCONTROLLER * ppAnimController //Anim controller copy
);
```

Once a clone has been created from an animation controller, it keeps an independent time count and active sets of animations, etc. This means that you should copy only the animation controller rather than the whole bone structure when you want to create multiple instances of a character. When rendering multiple instances of the same skinned mesh, follow this outline:

1. Call `AdvanceTime()` for the active animation controller.
2. Calculate the world matrix for this character instance.
3. Update the combined transformation matrices for the skinned mesh with the world matrix.
4. Render the skinned mesh.
5. Repeat with the next character instance.



Conclusions

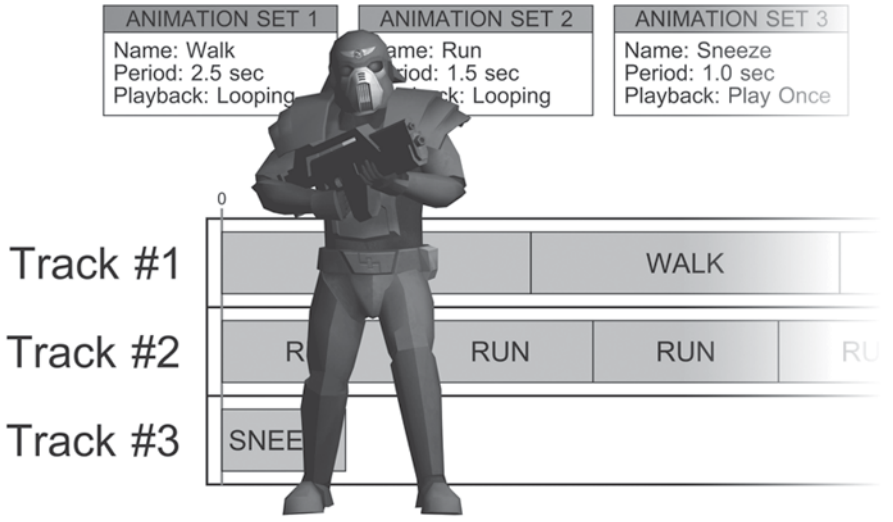
This chapter started with the keyframe, worked up to a collection of animations stored in the `ID3DXKeyframedAnimationSet()` interface, and finally covered the `ID3DXAnimationController` interface. You should now be comfortable with how animations are built from the ground up, even though in most cases you get them served on a silver platter. It never hurts to know how the animation pipeline works, especially later on when more advanced topics like dynamic animation are covered.

In this chapter you had your first look at the `ID3DXAnimationController` interface, but it won't be your last. The following chapter looks at some of the more advanced things you can do with this interface.

Chapter 4 Exercises

- Expand the `Animation` class created in Example 4.1. Make it easy to add new keyframes, set animation speed, etc.
- Connect the `Animation` class to a mesh. Make use of the scale, rotation, and translation you get from the animation set.
- Play around with the `ID3DXAnimationController` you retrieved from the Soldier. See if you can create a new animation set in code and register it with the controller. (Hint: The `RegisterAnimationSet()` and `RegisterAnimationOutput()` functions should prove useful).

5 Advanced Skeletal Animation Techniques



In this chapter I will dive deeper into some more advanced skeletal animation techniques. The first thing you will learn is how to blend several animations together. This is useful, for example, when you want smooth transitions between different animations/poses. The technique can also be used to create completely new animations. One example of this might be if you have a Run animation and a Fire-Rifle animation. By blending them you could have a Run-and-Fire-Rifle animation without having to actually animate this by hand in your 3D program. Also, at the end of this chapter, you'll look into the topic of motion capture. This chapter covers the following topics:

- Track structure
- Blending animations
- Compressing animation sets
- Animation callbacks
- Motion capture

The Track Structure

Before fading animations in/out, blending animations together, and more, one thing needs to be covered: the tracks in an animation controller. This was briefly touched on in the previous chapter, but I didn't really go in to any details. You may remember that the number of tracks was specified when creating a new animation controller using the `D3DXCreateAnimationController()` function. A track was also used to activate a certain animation for the character using the animation controller's `SetTrackAnimationSet()` function. As mentioned, an animation controller can contain several tracks. See Table 5.1 for a list of properties that you can manipulate for each track.

TABLE 5.1 ANIMATION TRACK PROPERTIES

The Position, Weight, and Speed properties are all quite easy to understand. The priority of a track can be set to either `D3DXPRIORITY_LOW` or `D3DXPRIORITY_HIGH`. High-priority tracks are blended together first before adding the low-priority tracks. This could also be used to turn off low-priority tracks when a character is far away from the player/camera.

To better illustrate the way you use tracks to blend animations, consider the following example. Figure 5.1 shows three animation sets, each containing a separate animation with details as shown.

ANIMATION SET 1	ANIMATION SET 2	ANIMATION SET 3
Name: Walk Period: 2.5 sec Playback: Looping	Name: Run Period: 1.5 sec Playback: Looping	Name: Sneeze Period: 1.0 sec Playback: Play Once

FIGURE 5.1
Three example animation sets.

Figure 5.1 shows the Walk, Run, and Sneeze animations. Both the Walk and the Run animations are looping animations, meaning that they will go on forever, whereas the Sneeze animation happens only once and then stops. Figure 5.2 shows how it would look if you assigned each animation to a separate track.

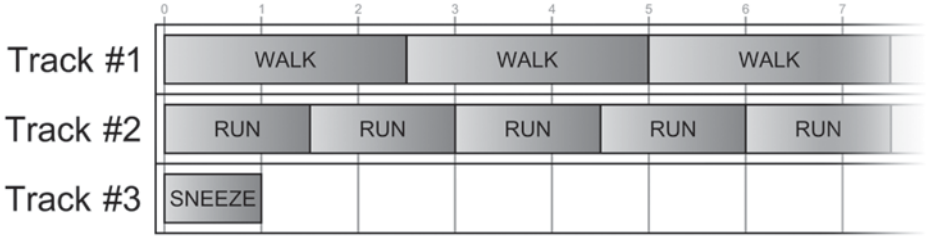


FIGURE 5.2
Three animation sets assigned to a separate animation track.

You are not limited to having a different animation set in each track. Sometimes it might make sense to have the same animation assigned to more than one track. Check out Figure 5.3, for example; the Walk animation has been assigned to both Track 1 and Track 2, the difference being that the track speed in Track 2 is 200% (i.e., the animation will play twice as fast).

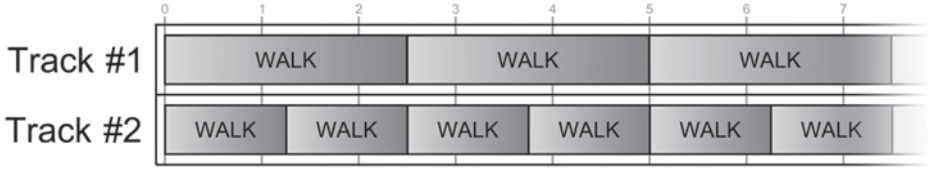


FIGURE 5.3
The track's speed property affects the animation playback.

To retrieve the current state of a track, you can use the following animation controller function:

```
HRESULT GetTrackDesc(
  UINT Track,              //Track to retrieve info about
  LPD3DXTRACK_DESC pDesc   //Track description
);
```

This function will fill the following structure:

```
struct D3DXTRACK_DESC {
    D3DXPRIORITY_TYPE Priority;
    FLOAT Weight;
    FLOAT Speed;
    DOUBLE Position;
    BOOL Enable;
};
```

The only piece of information this structure does not contain about a track is the current animation set assigned to it. For that you can use this function defined in the `ID3DXAnimationController` interface:

```
HRESULT GetTrackAnimationSet(
  UINT Track,
  LPD3DXANIMATIONSET * ppAnimSet
);
```

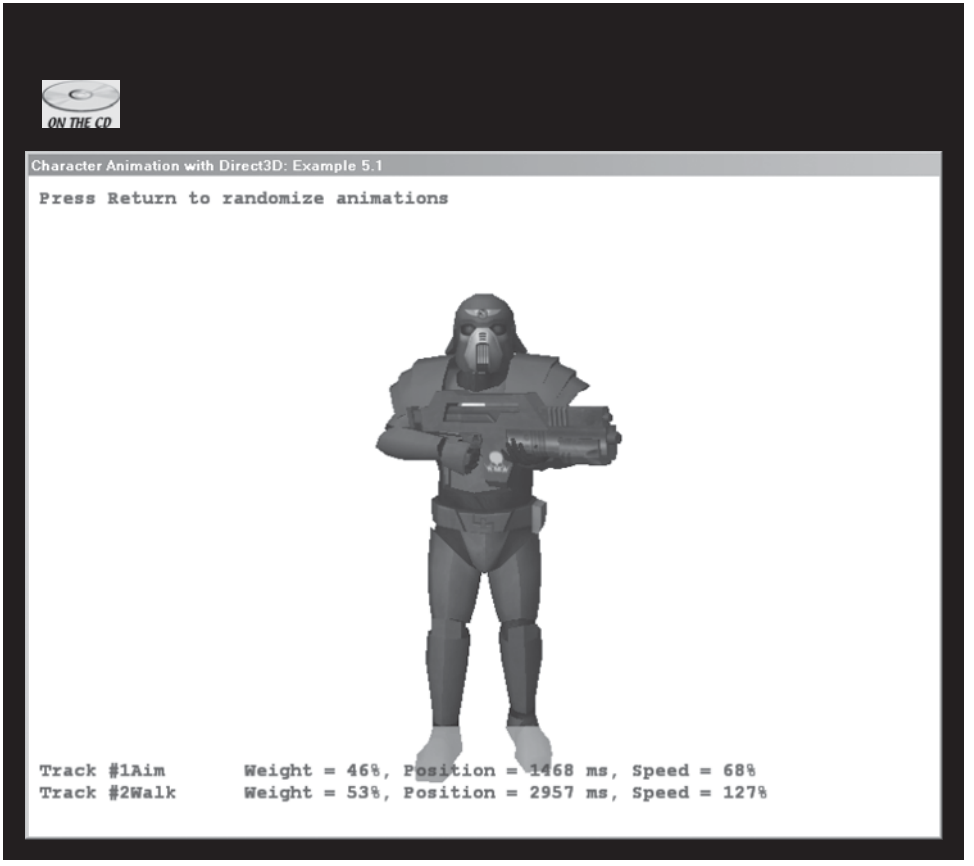
The animation controller's `GetTrackAnimationSet()` function returns a pointer to the animation set currently assigned to a specific track. Alright, now you know how to query all the necessary track properties of an animation controller. It's time to move on and try to blend two tracks together.

Blending Multiple Animations

To blend several animations together, you need to retrieve the different animation sets you want to use. Then you assign them to different tracks and set the weights, priorities, and speed of the different tracks. The following piece of code randomly blends two animations together:

```
//Reset the animation controller's time
m_animController->ResetTime();

//Get two random animations
int numAnimations = m_animController->GetMaxNumAnimationSets();
ID3DXAnimationSet* anim1 = NULL;
ID3DXAnimationSet* anim2 = NULL;
m_animController->GetAnimationSet(rand()%numAnimations, &anim1);
m_animController->GetAnimationSet(rand()%numAnimations, &anim2);

//Assign them to two different tracks
m_animController->SetTrackAnimationSet(0, anim1);
m_animController->SetTrackAnimationSet(1, anim2);

//Set random weight
float w = (rand()%1000) / 1000.0f;
m_animController->SetTrackWeight(0, w);
m_animController->SetTrackWeight(1, 1.0f - w);

//Set random speed (0 - 200%)
m_animController->SetTrackSpeed(0, (rand()%1000) / 500.0f);
m_animController->SetTrackSpeed(1, (rand()%1000) / 500.0f);

//Set track priorities
m_animController->SetTrackPriority(0, D3DXPRIORITY_HIGH);
m_animController->SetTrackPriority(1, D3DXPRIORITY_HIGH);

//Enable tracks
m_animController->SetTrackEnable(0, true);
m_animController->SetTrackEnable(1, true);
```

If two animations try to animate the same bone, their respective weights will determine how the bone is animated. For example, if one animation track has a weight of 5 and another track has a weight of 2.5, then any bone affected by both tracks will be affected twice as much by the first track compared to the second.



Compressing Animation Sets

You have already gotten to know the `ID3DXKeyframedAnimationSet` interface a little bit, and learned how you can add keyframes to it. In large games with huge amounts of animation data, it is prudent to sometimes compress the animation data to allow more of it. Again, the D3DX library is a great help. For compressed animation sets, you can use the `ID3DXCompressedAnimationSet` interface. In order to convert a keyframed animation set to a compressed animation set, you need to call the `Compress()` function of the keyframed animation set you want to compress.

```
HRESULT Compress(
  DWORD Flags,                        //Compression flags
  FLOAT Lossiness,                    //Animation Lossiness
  LPD3DXFRAME pHierarchy,             //Bone hierarchy
  LPD3DXBUFFER * ppCompressedData     //Compressed data output
);
```

The compression flag can be either D3DXCOMPRESS_DEFAULT, which is a fast compression scheme, or D3DXCOMPRESS_STRONG, which is a slower but more accurate compression method. (Note: Strong compression is not yet supported, but perhaps in future releases of DirectX it will be.) You can also set the desired lossiness (i.e., how much the compression scheme is allowed to change the data) as a value between zero and one. As output from this function, you do not get an ID3DXCompressedAnimationSet—instead, you get a chunk of data containing all the compressed animations, their keyframes, etc. After you have this compressed data, you can create a new compressed animation set using the following D3DX library function:

```
HRESULT D3DXCreateCompressedAnimationSet(
  LPCSTR pName,
  DOUBLE TicksPerSecond,
  D3DXPLAYBACK_TYPE Playback,
  LPD3DXBUFFER pCompressedData,
  UINT NumCallbackKeys,
  CONST LPD3DXKEY_CALLBACK * pCallKeys,
  LPD3DXCOMPRESSEDANIMATIONSET * ppAnimationSet
);
```

You supply the name, ticks per second, playback type, the compressed animation data, and optional callback keys (more on these later), and you'll get the new compressed animation set as a result. Here's some code showing how to use these functions to convert a keyframed animation set to a compressed animation set.

```
ID3DXKeyframedAnimationSet* animSet = NULL;

// ...
//Create or load the animation set you want to convert here...
// ...

//Compress the animation set
ID3DXBuffer* compressedData = NULL;
animSet->Compress(D3DXCOMPRESS_DEFAULT, 0.5f, NULL, &compressedData);

// Create the compressed animation set
ID3DXCompressedAnimationSet* compressedAnimSet = NULL;
D3DXCreateCompressedAnimationSet(animSet->GetName(),
                                 animSet->GetSourceTicksPerSecond(),
                                 animSet->GetPlaybackType(),
                                 compressedData,
```

```
                                        0, NULL,
                                        &compressedAnimSet);

//Release the compressed data
compressedData->Release();
```

As you can see, the name, playback type, and ticks per second are taken from the original animation set. You just supply the additional compressed animation data and as a result you get your compressed animation set.

After you have compressed an animation set, you no longer have direct access to the keyframes stored in it.

This might seem like a lot of trouble to go through just to decrease the size of the animation set. But once the number of animations starts increasing drastically, compressing your animation sets is a good trick to have up your sleeve.

Animation Callback Events

Animation callbacks are events that are synchronized with your animations. One example might be playing the sound of a footstep. Imagine that you have a walk animation like the one earlier in this chapter. Remember that you can play this animation with different speeds. If you had to connect the sound of a footstep to the animation manually, you would have to go through all kinds of worry to calculate the times where you need to play the sound. This is where animation callbacks come into the picture. You create a Callback key and add it to the animation set. Every time the animation passes this Callback key, it generates an event where the sound is played. You can also customize this event—for example, to play different sounds if the character is stepping on gravel rather than a wooden surface. The Callback keys are defined using the following structure:

```
struct D3DXKEY_CALLBACK {
    FLOAT Time;                 //Time the callback occurs
    LPVOID pCallbackData;       //User defined callback data
};
```

The `D3DXKEY_CALLBACK` structure contains one float value containing the timestamp, and one pointer to any user defined structure. As mentioned in the previous chapter, the timestamps of these animation key structures are in ticks, not seconds.

So remember to multiply the actual time (in seconds) you want the event to occur with the animation's ticks per seconds value.

```
struct A_USER_DEFINED_STRUCT
{
    int m_someValue;
};

//A global instance of the user defined structure
A_USER_DEFINED_STRUCT userData;

D3DXKEY_CALLBACK CreateACallBackKey(float time)
{
    D3DXKEY_CALLBACK key;
    key.Time = time;
    key.pCallbackData = (void*)&userData;
    return key;
}
```

This code creates a user defined structure, and defines a function that creates a new callback key linked to this user defined structure. After you've added lots of callback events, you need to create your own callback handler to deal with the events as they come in. To do this you need to implement your own version of the `ID3DXAnimationCallbackHandler` interface.

```
class CallbackHandler : public ID3DXAnimationCallbackHandler
{
    public:
    HRESULT CALLBACK HandleCallback(THIS_ UINT Track,
                                    LPVOID pCallbackData)
    {
        //Access the user defined data linked to the callback key
        A_USER_DEFINED_STRUCT *u;
        u = (A_USER_DEFINED_STRUCT*)pCallbackData;

        if(u->m_someValue == 0)
        {
            //Do something
        }
        else
        {
            //Do something else...
        }
```

```
            return D3D_OK;
        }
    };
```

Here you can see how you can implement the `ID3DXAnimationCallbackHandler` interface to deal with your own user defined data structures. All event handling is done in the `HandleCallback()` function, which is the only function defined in the `ID3DXAnimationCallbackHandler` interface. Okay, so now you know how to create callback keys and how to handle them once they have triggered an event, but what hasn't been covered yet is how to add new callback keys to an existing animation.

```
//Get a keyframed animation set
ID3DXKeyframedAnimationSet *animSet = NULL;
m_animController->GetAnimationSet(0, (ID3DXAnimationSet**)&animSet);

//Create one callback key
D3DXKEY_CALLBACK key[1];
//Fill the callback key time + callback data here...

//Add callback key to animation set
animSet->SetCallbackKey(0, key);
```

The `SetCallbackKey()` function adds a callback key to a normal keyframed animation set. You can also add callback keys to a compressed animation set like this:

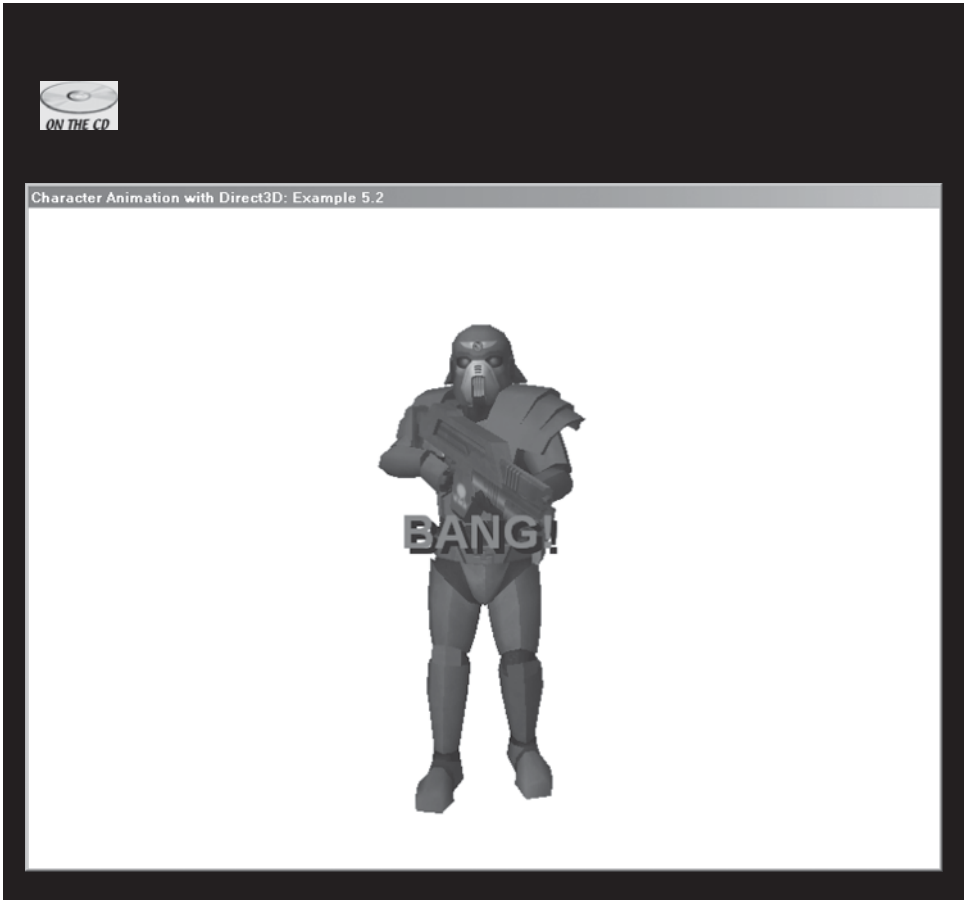
```
//Get a keyframed animation set
ID3DXKeyframedAnimationSet* animSet = NULL;
m_animController->GetAnimationSet(0, (ID3DXAnimationSet**)&animSet);

//Compress the animation set
ID3DXBuffer* compressedData = NULL;
animSet->Compress(D3DXCOMPRESS_DEFAULT, 0.5f, NULL, &compressedData);

//Create one callback key
const UINT numCallbacks = 1;
D3DXKEY_CALLBACK keys[numCallbacks];

//Create callback key(s) and set time + callback data here...

//Create a new compressed animation set
ID3DXCompressedAnimationSet* compressedAnimSet = NULL;
```

```
D3DXCreateCompressedAnimationSet(animSet->GetName(),
                          animSet->GetSourceTicksPerSecond(),
                          animSet->GetPlaybackType(),
                          compressedData,
                          numCallbacks,
                          keys,
                          &compressedAnimSet);

//Release compressed data
compressedData->Release();

//Delete the old keyframed animation set.
m_animController->UnregisterAnimationSet(animSet);
animSet->Release();

// And then add the new compressed animation set.
m_animController->RegisterAnimationSet(compressedAnimSet);
```

Like before, when you compress an animation set, you do the same exact steps. Only this time you also supply the `D3DXCreateCompressedAnimationSet()` function with a set of callback keys. After the new compressed animation set has been created, you unregister the old animation set in the animation controller and register the new compressed animation set in its place. The last thing before it all comes together is to send the callback handler to the animation controller's `AdvanceTime()` function.

```
m_animController->AdvanceTime(m_deltaTime, &callbackHandler);
```

This essentially means you can also have different callback handlers handling the same callback events. So, for example, if your character were wounded, you could have a different callback handler than when the character is healthy. In other words, different code could be executed every time a certain callback event is triggered, depending on what callback handler you are using.



Motion Capture (Mocap)

This section provides a brief glimpse into the advanced topic of motion capture, also known as *Mocap*. Motion capture is the process of recording movements from real-life actors and applying these movements/animations to 3D characters. The use of motion capture is most common in the movie and game industry. With Mocap equipment you can produce much more life-like animations than you can with more traditional 3D animation tools. The rates with which you can create new character animations are also so much faster than in the traditional way.

There are a few different types of Mocap systems. Generally speaking, they can be divided into three categories: optical, magnetic, and mechanical. Although these systems have many differences, they also have some general things in common. They are all ridiculously expensive, require lots of technical expertise, and also require lots

of calibration. Because of this, it is very common that game companies (and other companies) outsource their motion capture needs to studios specializing in Mocap. At the end of this chapter is an interview with some of the folks of Lapland Studio who do a lot of Mocap for other companies.

Optical Motion Capture Systems

In a nutshell, an optical Mocap system works with several cameras mounted on the walls of a room facing the center. These cameras are usually very expensive high-contrast cameras. An actor is then dressed in a suit that has a large number of small white balls (markers) attached to it. These markers are captured by the camera, which uses triangulation to calculate the position of the marker in 3D space. Figure 5.4 shows how a system like this could be set up. The markers usually come in two flavors, depending on the system: active (containing a small infra-red light) and non-active (a reflective marker).

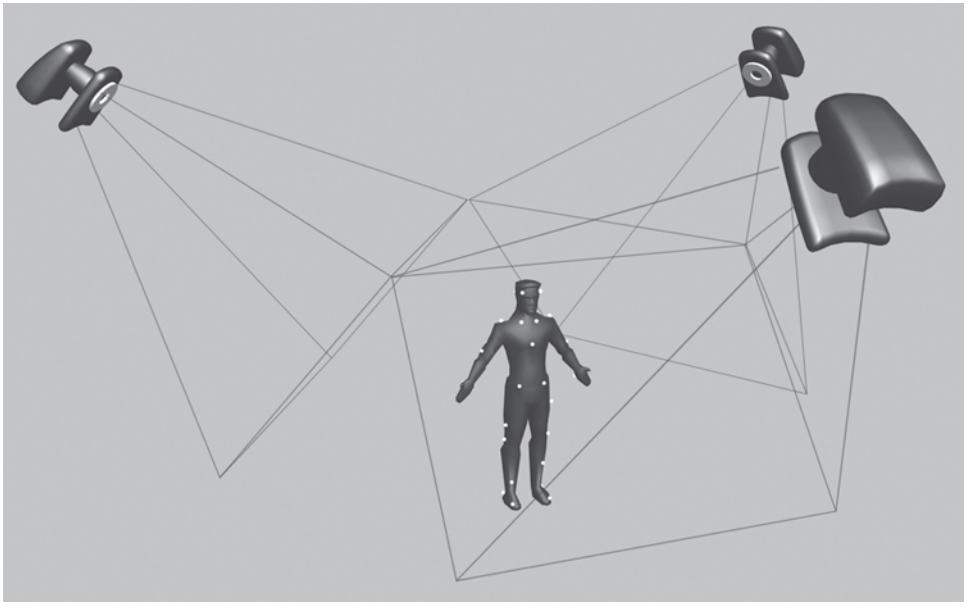


FIGURE 5.4
An optical Mocap system.

Figure 5.4 shows only three cameras (which is the theoretical minimum for a system like this to work); however, the more cameras you have, the more accurate and robust the system will be. Figure 5.5 shows what the images from the three cameras in Figure 5.4 would look like.

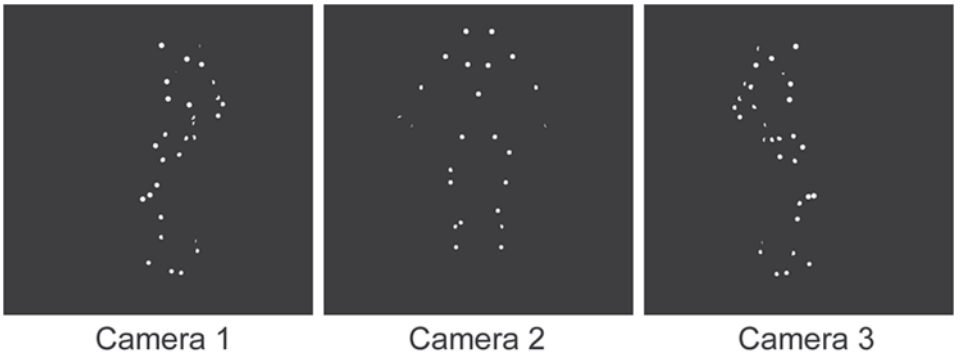


FIGURE 5.5
Images recorded from an optical Mocap system.

Even from these simple images you can easily see the outline of the person wearing the markers. It becomes even more apparent when you are watching a live feed of these markers moving. Once the images from all the cameras have been used to calculate the 3D positions of the many markers, these are mapped onto a virtual skeleton. The motion of this virtual skeleton is then exported and can be used in a 3D modeling software, and finally in a game or a movie.

Marker-Less Motion Capture

Lately there has been a lot of research in the field of marker-less motion capture. At the time of writing, this technology is just beginning to make its way into the market [1]. Essentially, marker-less Mocap works like any other optical system but without markers. The motion is extracted using multiple cameras and advanced computer vision algorithms focusing on certain spots of your body, contour detection, etc. Marker-less Mocap is especially good for things like facial animation [2].

Magnetic Motion Capture Systems

Magnetic motion capture systems work almost like optical systems. Instead of visual markers, wired sensors are attached to a person's limbs. These sensors are connected via a shielded cable to a control unit that measures their position and orientation in a low-frequency magnetic field. The electronic magnetic field is created by a static magnetic emitter. The great thing about magnetic Mocap is that it also gives you the orientation of the sensor (something which had to be calculated off-line with optical systems). This makes magnetic Mocap systems good for real-time motion capture (used in different live TV shows, conventions, and so on). One big downside of magnetic Mocap systems is that they are wired (and the sensors can also weigh quite a bit). This means that they are cumbersome and restrict the actor's movement while

recording. Another big downside of magnetic Mocap systems is that they are very sensitive to noise and other magnetic fields. Any metallic surface will interfere with the magnetic field and cause faulty readings from the sensors. The components of a magnetic motion capture system can be seen in Figure 5.6.

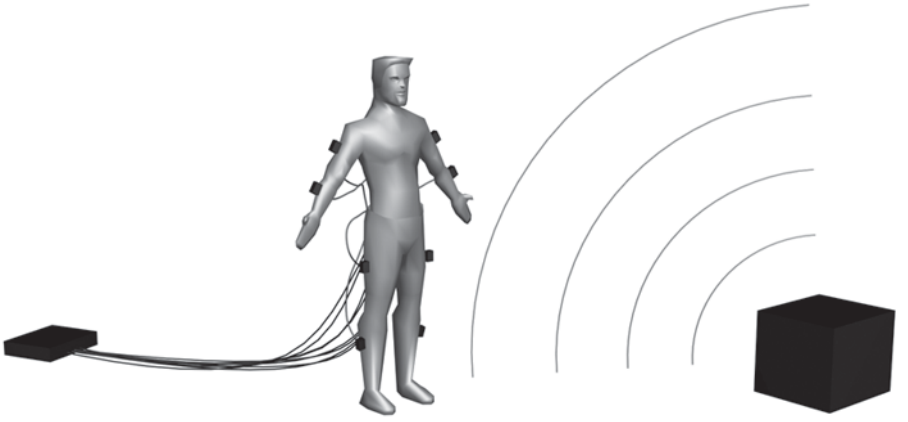


FIGURE 5.6
A magnetic Mocap system.

Mechanical Motion Capture Systems

Mechanical motion capture systems usually build on an exoskeleton worn by the actor. The different joint orientations of the exoskeleton are recorded and used to produce the Mocap data. The major downside to this technology is that no position data is recorded, so things like jumping, realistic running animations, etc. can't be recorded directly but need some manual touch up afterward. Another downside to this technology is that the exoskeleton often tends to be quite bulky and can restrict the actor somewhat. However, not all is bad about a mechanical Mocap system. The fact that it is mechanical means that it doesn't suffer from interference, occlusion, and similar problems. There are also examples of mechanical Mocap systems that have the recording computer and power supply in a backpack, effectively making the suit completely independent of location. You can see an example a mechanical Mocap system with an exoskeleton in Figure 5.7.

FIGURE 5.7
A mechanical Mocap system.

A more resent implementation of motion capture using a body suit has been done by Moven [3]. They have built a slim suit with miniature inertial sensors, which aren't cumbersome at all. Since this system isn't relying on cameras, etc., it has the great advantage that it can be used anywhere.

Comparison of the Different Mocap Systems

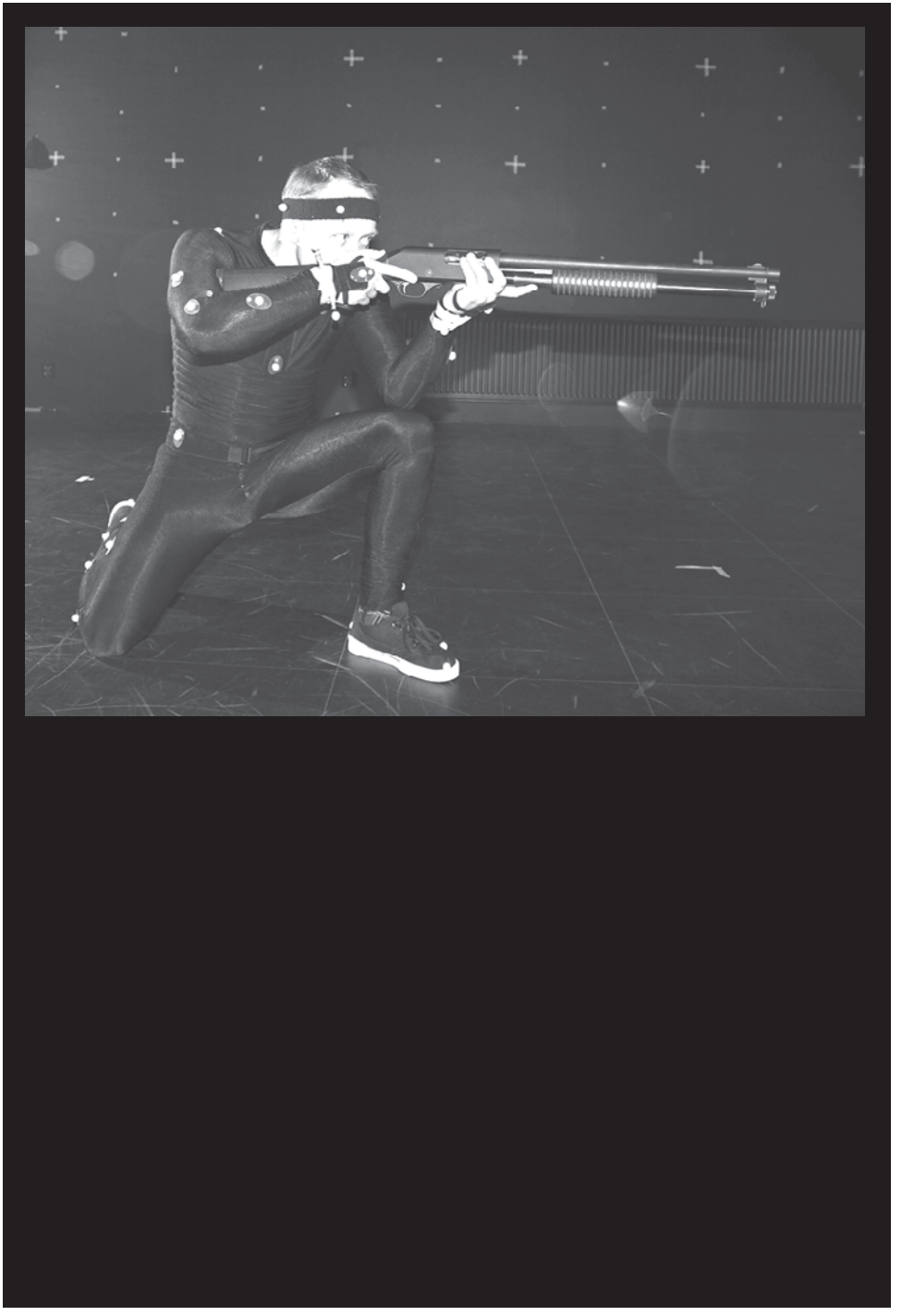
Needless to say, all these technologies have their own pros and cons. There are also several variations of each of these, all with their own individual strengths and weaknesses. Table 5.2 provides an overview of the pros and cons of each system.

Despite the shortcomings of optical systems, their pros outweigh their cons when it comes to Mocap for games and movies. In time, marker-less Mocap may replace regular optical systems. For now, at least, it seems that the high sampling rate and high accuracy is what makes the optical technology the best approach for game character motion capture.

TABLE 5.2 MOCAP TECHNOLOGY COMPARISON

Lapland Studio Interview

I had the opportunity to visit a Lapland Studio's motion capture studio in Rovaniemi, Finland. They have a VICON [4] optical motion capture system using 14 cameras and a capture area of 4 x 4 x 3 meters. The following interview is an excerpt from the discussion I had with Jari Niskanen and Jouko Manninen, both CG artists at Lapland Studio.




Lapland
STUDIO

Conclusions

In this chapter you learned about some more advanced topics of character animation. You learned about animation blending, which is something you'll definitely need if you ever aim to create a realistic character. With it, you can create new animations by blending two others together. You can also create a transition between two animations by blending between them. Another advanced topic covered in this chapter was animation callbacks, with which you can time events in your game to when a specific animation occurs. If you made a fighting game for instance, you might want to time the "punch" sound to a specific time in the punch animation. Lastly, the topic of motion capture and its variations were briefly covered. That about wraps up the skeletal animation part of this book, which the last three chapters have focused on. Now it's time to turn to dynamic animation and look at implementing a ragdoll system.

Chapter 5 Exercises

- Try modifying Example 5.1 so that you run the same animation in two different tracks, with different weights and slightly different speeds. See what happens?
- Create animation callbacks for the footsteps in the Soldier's walk animation.

Further Reading

[1] http://www.organicmotion.com

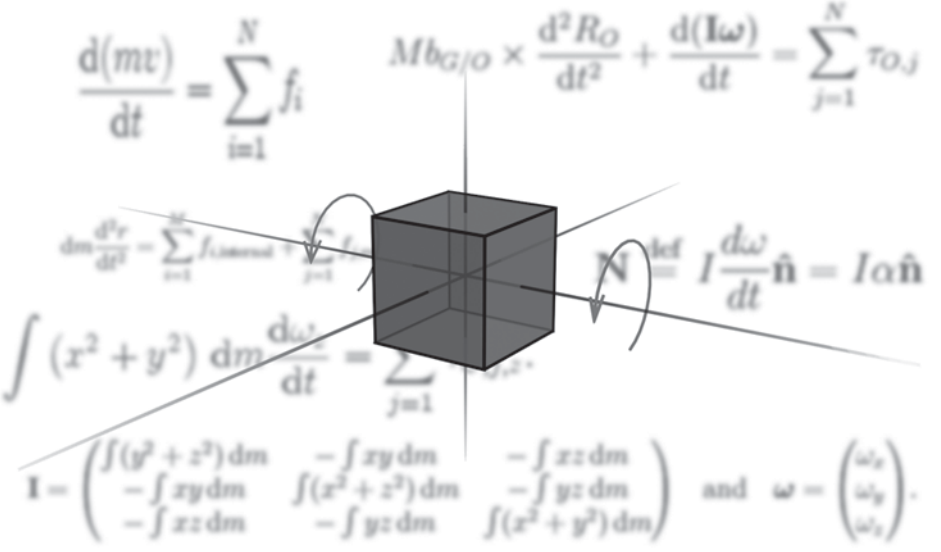
[2] http://www.mova.com

[3] http://www.moven.com

[4] http://www.vicon.com

This page intentionally left blank

6 Physics Primer



As is common practice with most programming books that include only one chapter covering physics, I'll start this one with a disclaimer. This book is not about physics! There are books focusing solely on the topic of game physics. However, since I will be covering ragdoll animations in this book, I need cover the "bare bones," so to speak, of creating a physics engine. The physics engine demonstrated in this chapter builds on simulating particles connected with springs. In this chapter you'll learn about the following topics:

- Basics of rigid body physics
- Quaternions
- Oriented bounding boxes
- Intersection tests
- Simulating a particle
- Simulating a spring

In the beginning of this chapter, many basic physical concepts will be covered. If you are already comfortable with Newton's three laws of motion, gravity, quaternions, etc., feel free to skip ahead to the implementation part at the end of this chapter.

Introduction to Rigid Body Physics

If you've never come across the topic of rigid bodies before, don't worry; it is quite easy to understand (even though it can be quite hard to simulate). A rigid body is by definition a solid mass in space that does not change shape. A real-life basketball is an example of a non-rigid body. The fact that the basketball is not rigid is what causes it to bounce once it hits the floor. Instead, imagine a sphere the size of a basketball in solid steel hitting a steel floor. Very likely there wouldn't be a very large bounce from a collision like this. This is because the steel sphere won't change its shape. It is rigid!

In computer graphics there are accurate mathematical systems where rigid bodies are simulated colliding with the environment and with each other. In games, however, this simulation has to be able to run in real time. This essentially means several shortcuts need to be taken, and some serious optimizations must be done in order to get it fast enough for real-time applications. So when making physics engines for games, things like speed, stability, and appearance take precedence over accuracy and realism.

Before I dive into rigid body physics, I'll cover the basics of physics in general. Over 300 years ago a fellow named Isaac Newton published a three-volume work called *Philosophiae Naturalis Principia Mathematica*, or *Principia* for short. In plain English, the title was "Mathematical Principles of Natural Philosophy" and contained, as you may already know, Newton's three laws of motion. The first of the three volumes was called *De motu corporum* ("On the motion of bodies"), and now, more than 300 years later, these laws are still used when simulating physics in games. Table 6.1 shows a simplified version of Newton's laws of motion.

TABLE 6.1 NEWTON'S LAWS OF MOTION

The second law can also be described with the famous formula:

$$F = ma \quad \text{or} \quad a = \frac{\mathrm{F}}{m}$$

The force equals the mass of an object times its acceleration. However, more often you are interested in the acceleration resulting from an external force, in which case the acceleration *a* is the force *F* divided by an object's mass *m*. Later on I'll discuss how the acceleration of an object affects its velocity and its position. But first it is time to cover some important concepts needed to create your own physics engine.

Forces

As you may have seen in Newton's three laws, there was a lot of talk about forces. A force has both a magnitude and a direction. Imagine, for example, two equally strong men pushing a box from opposite sides. Since the directions of their efforts are opposing, the box won't move an inch. However, if the two men were pushing from the same side, the two forces would combine and the box would move in the direction they are pushing. This little thought-experiment proves the fact that two opposing forces cancel each other out.

The most common force you face on a daily basis is gravity (unless you happen to be an astronaut). In his *Principia*, Newton also defined the law of gravity:

$$F_G = G\frac{m_1 \times m_2}{d^2}$$

F_G is the resulting gravitational force, G is the universal gravitational constant [Wikipedia], and m_1 and m_2 are the masses of the two objects attracting each other. Finally, *d* is the distance between the objects. Simply put: Two objects attract each other with a force proportional to the product of their masses divided by the square of the distance between them (quite a mouthful). Take the simple example of the Sun and the Earth, as shown in Figure 6.1.

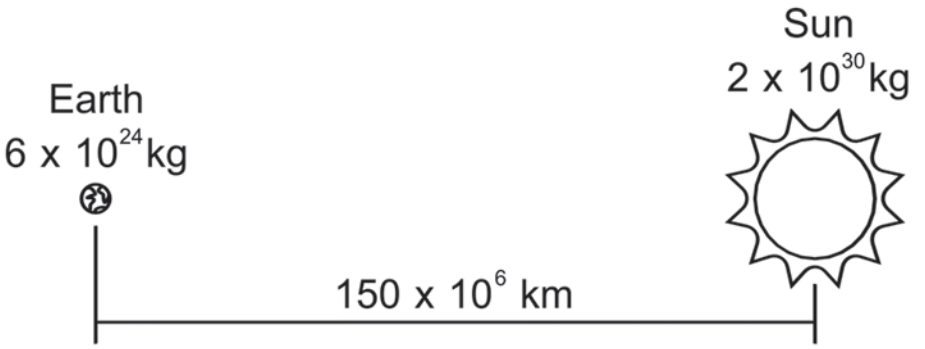


FIGURE 6.1
Gravitational pull between the Earth and the Sun.

It might be hard to see from the numbers representing the mass of the Sun and the Earth. But the Sun has 333,000 times more mass than the Earth and it represents 98% of all the mass in our solar system. This means that whatever the gravitational force is between the Sun and the Earth, 99.9997% of that force affects the earth, and 0.0003% of it affects the sun. This gravitational force is the only thing keeping the Earth in orbit around the Sun.

Now turn your attention to the Earth itself, where there's a similar gravitational force affecting all smaller objects on the Earth's surface (for instance, an apple). Just like in the example of the Sun and the Earth, each small object actually also attracts the Earth toward it. However, this force is so small that it is negligible. This leaves us with one force pulling all objects toward the Earth's surface.

In games, this is almost always represented as a constant force in the negative Y direction (and the curvature of the Earth is completely dismissed). Just as in real life, games try to simulate the Earth's gravitational pull (9.8 m/s2), making objects behave as realistically as possible. Later on I'll cover how we apply gravity to game objects, updating their velocity and position over time.

That pretty much covers the gravitational force. However, there are plenty of other non-constant forces in the world, such as wind, collision impacts, etc.—for example, in an action game when a bullet hits an object, the bullet will affect the object with a force proportional to its speed and mass. This brings us to the different ways a force can affect an object.

The Effect of Forces on a Rigid Body

Okay, so you know that there are different forces affecting an object (i.e., a rigid body). The next thing to cover is how these bodies react when affected by an external force. First, consider what happens to a rigid body's position when you push or pull it with an external force. Figure 6.2 shows an example.

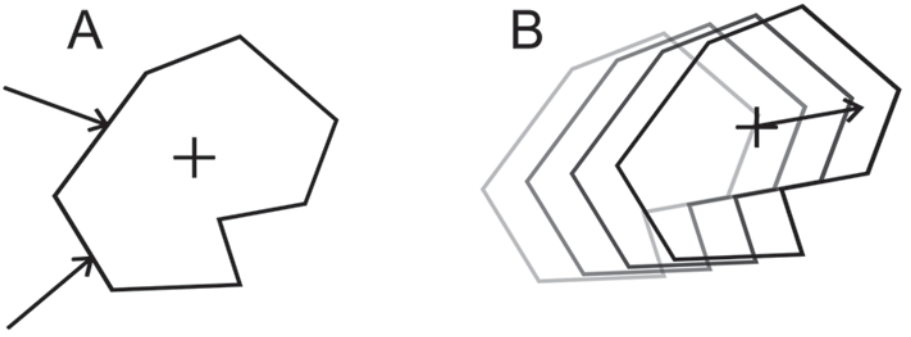


FIGURE 6.2
How a force affects the position of a rigid body.

As you can see in Figure 6.2A, the body is being affected by two external forces pushing at the object from different directions. Just like with the analogy of the two men pushing a box, the rigid body will move over time in the combined direction of the forces as long as the forces are applied. In this example you can see the center of the rigid body's mass represented as a cross. The forces in Figure 6.2 are pointing straight at the object's center of mass, resulting in the energy being used 100% to move the object. However, this is of course not always the case; sometimes forces are applied to an object causing it to spin, rather than to move linearly.

Try it yourself. Find a small object (like a pen) and poke it close to its center of mass. No doubt the object will move in the direction you poked it. Try it again, but this time poke it far from its center of gravity. This time the object will spin instead. This phenomenon is shown Figure 6.3.

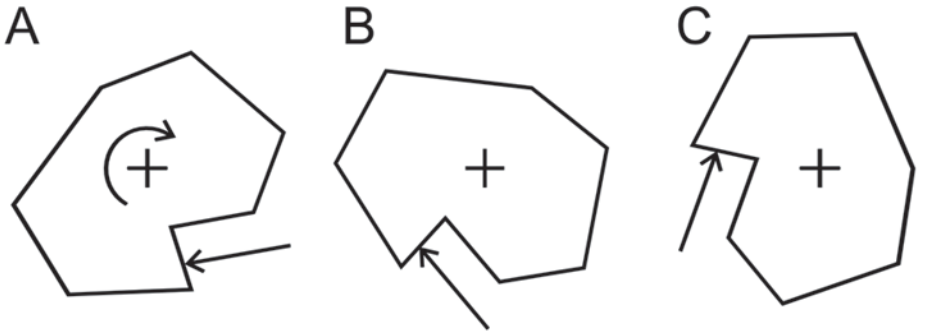


FIGURE 6.3
How a force affects the orientation of a rigid body.

Just like in the previous case, when there are several forces affecting the orientation of an object, the forces are all summed up before applying the final rotation of the object. So far I have only been talking about objects in 2D space, but the same fundamental reactions naturally occur with objects in 3D space as well. To describe the orientation of an object in a physics simulation, you basically have three different options:

- Euler angles
- Rotation matrix
- Quaternions

If you have covered the basics of 3D math, you have probably come in contact with Euler angles. Euler angles are easy to understand; you have one rotation value for the yaw, pitch, and roll of an object. Even though the Euler angles are easy to grasp, they come with some limitations such as Gimbal lock (covered in Chapter 4). Another option is to use a rotation transformation matrix. Although this is an often used option, it suffers from the fact that small imperfections creep in (due to rounding of float values) and the matrix becomes skewed over time. This pretty much leaves us with quaternions!

Quaternions

Before continuing with the implementation of the physics engine, and later the ragdoll system, you need to understand the concept of quaternions. Quaternions are stored as four values; the first value is a scalar value, and the three following values describe a vector. Since a quaternion is a four-dimensional entity, it's close to impossible to create a mental image of it. Instead, the best approach for any non-mathematician is just to learn how to create an arbitrary orientation using quaternions, and then get more comfortable with them by using them in your code. The word *quaternion* comes from Latin's "*Quaternio*," which means "set of four." A quaternion is defined as the following four values:

$$q = [w, x, y, z]$$

Quaternions were invented by an Irish mathematician named Sir William Hamilton some 200 years ago. His motivation was to come up with a method to describe the transformation required to transform a vector V1 into a vector V2. In the case of two points A and B, there exists a vector V that transforms point A into point B, as shown in Figure 6.4.

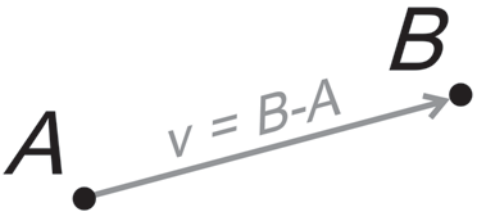


FIGURE 6.4
Transforming one point into another.

A quaternion performs the same operation but on two vectors. A quaternion can transform one specific vector into another specific vector. Usually a vector is described as its x, y, and z components, but a vector can also be described with a direction and a length. Here's the general theory of how you can transform one vector into another when they are described as a direction and a length:

FIGURE 6.5
Vectors V1 and V2 differ both in magnitude and orientation.

Figure 6.5 shows the two vectors V1 and V2. The first step of changing V1 into V2 is to make sure their length (or magnitude) is the same. For this you simply ignore their orientation and calculate the scale factor S you need to apply to V1 so that its length matches that of V2.

$$S = \frac{|V2|}{|V1|}$$

The scale factor *S*, is calculated by simply dividing the length of vector V2 with the length of vector V1. Next you need a way of transforming a direction into another, and for this you need to use versors. A *versor* describes the difference in orientation of two vectors of equal length, as shown in Figure 6.6.

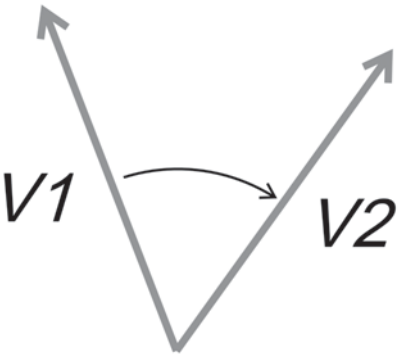


FIGURE 6.6
Transforming the direction of a vector to another.

All you need to reorient V1 into V2 is the angle between the vectors and the axis around which to rotate. The angle α is obtained by calculating the inverse cosine value of the vectors' dot product:

$$a = a\cos(V1 \bullet V2)$$

The axis around which the rotation should take place is simply the cross product of the vectors:

$$A = V1 \times V2$$

The quaternion that combines the scaling and rotating of V1 into V2 can then be created using one value for scale, one for the angle, and two values for the plane on which V1 and V2 lie. When I talk about orientations and rotations using quaternions throughout this book, only a small subset of all possible quaternions are of particular interest. This subset is all the unit quaternions (i.e., quaternions with a length of 1). All possible unit quaternions form a hyper-sphere, which basically is a four-dimensional sphere (something quite hard to visualize).

Here's a practical problem—one which you will no doubt run into once you switch to quaternions. Have a look at the problem shown in Figure 6.7.

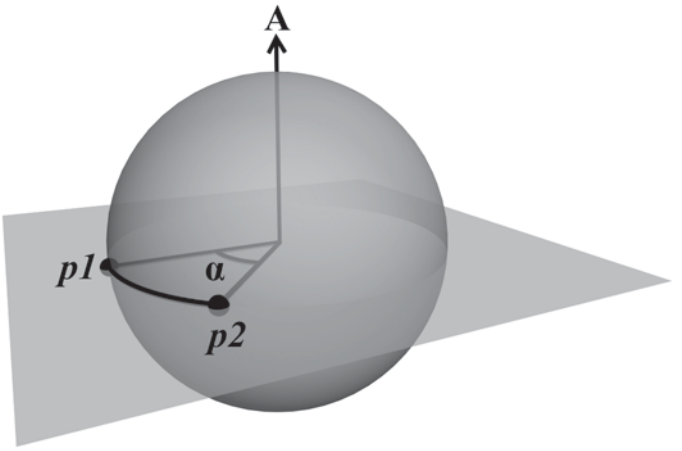


FIGURE 6.7
Transforming the unit vector p1 into p2 using quaternions.

In Figure 6.7, a rotation of **α**-radians around Axis **A** is described. This rotation will transform the vector **p1** into **p2**. Remember that the axis around which you want to rotate can be any arbitrary axis. Here's how you can create a quaternion to define this rotation. Remember that quaternions were defined as:

$$q = [w, x, y, z]$$

In the example of rotating **α**-radians around Axis **A** (x, y, z), the quaternion for this would be:

$$q = \begin{bmatrix} cox(a/2), \\ x \bullet \sin(a/2), \\ y \bullet \sin(a/2), \\ z^{*} \sin(a/2) \end{bmatrix}$$

In DirectX, a quaternion is stored using the `D3DXQUATERNION` structure:

```
struct D3DXQUATERNION {
    FLOAT x;
    FLOAT y;
    FLOAT z;
    FLOAT w;
};
```

```
...

//Previous example in code
D3DXVECTOR3 A(0.2f, 0.6f, -0.3f);    //Any arbitrary axis
D3DXVec3Normalize(&A, &A);           //Normalized
float angle = 0.342f;                //Arbitrary angle

//Quaternion is then defined as
D3DXQUATERNION q;
q.w = cos(angle * 0.5f);
q.x = sin(angle * 0.5f) * A.x;
q.y = sin(angle * 0.5f) * A.y;
q.z = sin(angle * 0.5f) * A.z;
```

Luckily the D3DX library is full of quaternion helper functions. Table 6.2 lists the names of the most useful quaternion functions. These will be used throughout the following chapters, but if you want to familiarize yourself with them now, check out the DirectX Documentation.

TABLE 6.2 D3DX QUATERNION FUNCTIONS

For a more in-depth explanation of quaternions and quaternion math, check out [Ibanez01] or [Svarovsky00].

Matrices have the identity matrix, which when applied to an object leaves the translation, rotation, and scale unaffected:

$$M_i = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In the same way, quaternions have the identity quaternion, which does not affect the rotation when applied to an object:

Describing the World

If you have ever implemented things like viewport culling or picking, you may have encountered the Axis-Aligned Bounding Box (AABB). This bounding volume has two vectors, one containing the minimum corner (x, y, z) and once containing the maximum corner (x, y, z) of the box. Although it is fast to perform different intersection tests, etc. with this volume, it is quite inaccurate in many cases.

To describe the world your objects will interact with in the physics simulation, Oriented Bounding Boxes (OBB) will be used instead. By combining several of these Oriented Bounding Boxes, complex-enough worlds can be built for the purpose of this book. Figure 6.8 shows an image comparing the AABB and the OBB in two dimensions:

As shown in Figure 6.8, the OBB has one great advantage over the AABB: It fits much closer to the object (especially when the object is rotated, as is the case in Figure 6.8). With the OBB you can build up a much more accurate representation of the game world.

As explained earlier, an AABB is often described as a MAX and a MIN vector. These hold the maximum and minimum coordinates of the box in the x, y, and z dimensions. With an OBB, however, you need one vector with the size of the box, one vector for its position, and one vector for its orientation. However, in most physics applications, vectors (Euler angles) are avoided for orientation because of the Gimbal lock. So instead I'll use a quaternion for the orientation as covered in the previous section.

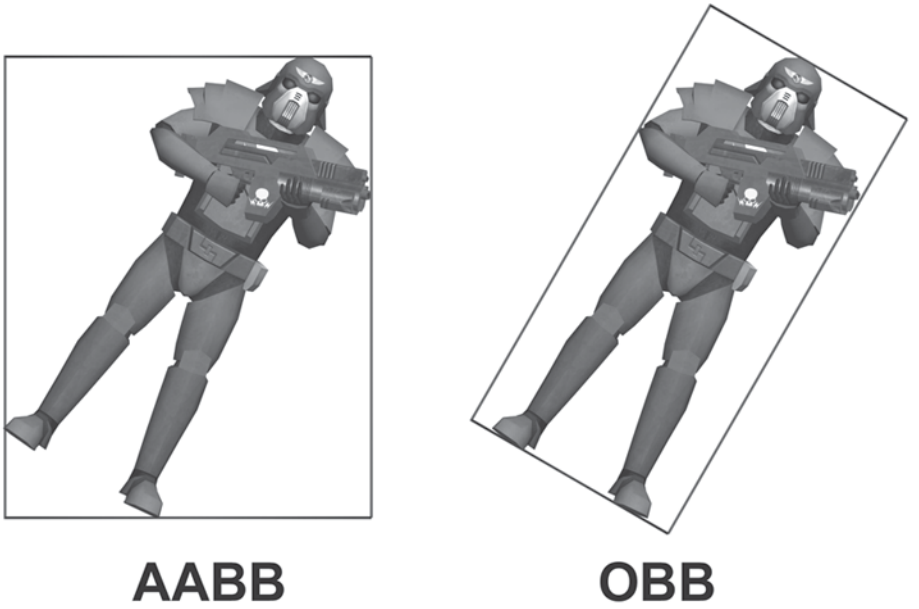


FIGURE 6.8
Axis-Aligned Bounding Box (AABB) versus Oriented Bounding Box (OBB).

The Oriented Bounding Box Class

The Oriented Bounding Box (OBB) is a structure that describes a volume with width, height, length, and orientation. For it to be useful in physic simulations, you need to be able to perform a number of different intersection tests. Often you also need to know how far an intersecting object is into the OBB so the physics engine can handle the collision accordingly. The `OBB` class is defined as follows:

```
class OBB
{
    public:
        OBB();
        OBB(D3DXVECTOR3 size);
        bool Intersect(D3DXVECTOR3 &point);
        bool Intersect(OBB &b);

    public:
        D3DXVECTOR3 m_size, m_pos;
        D3DXQUATERNION m_rot;
};
```

At the moment I've only defined two intersection tests: one Point-OBB and one OBB-OBB test. The OBB-OBB intersection test is a bit beyond the scope of this book. You'll find a good article on intersection tests online [Gomez99]. You can also find the code for the OBB-OBB test in Example 6.1 on the accompanying CD-ROM. However, you need to have a closer look at the more important Point-OBB test. First consider the point intersection test of an Axis-Aligned Bounding Box (AABB):

```
class AABB
{
    public:
        AABB(D3DXVECTOR3 max, D3DXVECTOR3 min)
        {
            m_max = max;
            m_min = min;
        }

        bool Intersect(D3DXVECTOR3 &p)
        {
            if(p.x < m_min.x || p.x > m_max.x)return false;
            if(p.y < m_min.y || p.y > m_max.y)return false;
            if(p.z < m_min.z || p.z > m_max.z)return false;
            return true;
        }

    public:
        D3DXVECTOR3 m_max, m_min;
};
```

The Point-AABB intersection test is very simple; you simply check for each dimension if the point is smaller than the minimum or larger than the maximum value of the bounding box. If so, the point must be outside the box. If after checking all the dimensions (x, y, z) none of these tests failed, then the point must be inside the box. To do this same test for an OBB is not quite as simple because it also has an orientation. Figure 6.9 shows this problem.

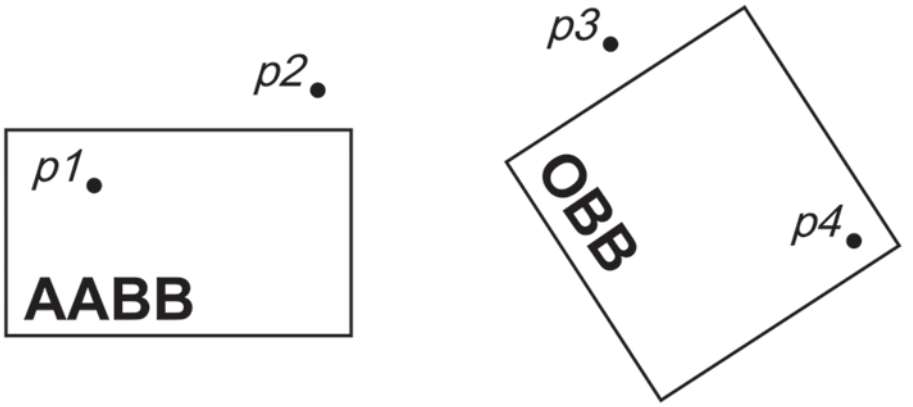


FIGURE 6.9
Point intersection, AABB versus OBB.

Look at Figure 6.9 and perform the intersection test listed earlier for the AABB. Now if you try to do the same for the OBB, you would soon realize that it is not quite as easy to tell whether the points are inside or outside the box using this method. However, with one simple trick it becomes just as easy to do for the OBB as for the AABB. You simply need to look at it from a different point of view. Tilt your head until the OBB becomes an AABB and then check the points using the same algorithm used for the Point-AABB intersection test. With this in mind, here's the code for the Point-OBB intersection test:

```
bool OBB::Intersect(D3DXVECTOR3 &point)
{
    //Calculate the translation & rotation matrices
    D3DXMATRIX p, r, world;
    D3DXMatrixTranslation(&p, m_pos.x, m_pos.y, m_pos.z);
    D3DXMatrixRotationQuaternion(&r, &m_rot);

    //Calculate the world matrix for the OBB
    D3DXMatrixMultiply(&world, &r, &p);

    //Calculate the inverse world matrix
    D3DXMatrixInverse(&world, NULL, &world);

    //Transform the point with the inverse world matrix
    D3DXVECTOR4 pnt;
    D3DXVec3Transform(&pnt, &point, &world);
```

```
        //Check the point just as if it was an AABB
        if(abs(pnt.x) > m_size.x)return false;
        if(abs(pnt.y) > m_size.y)return false;
        if(abs(pnt.z) > m_size.z)return false;

        return true;
    }
```

With the OBB structure you can describe some fairly complex (although angular) environments for us to run physics simulations in. In Figure 6.10 you can see an example of this: on the left side, the mesh used in the rendering pipeline, and on the right, the physical representation of the same mesh using OBBs.

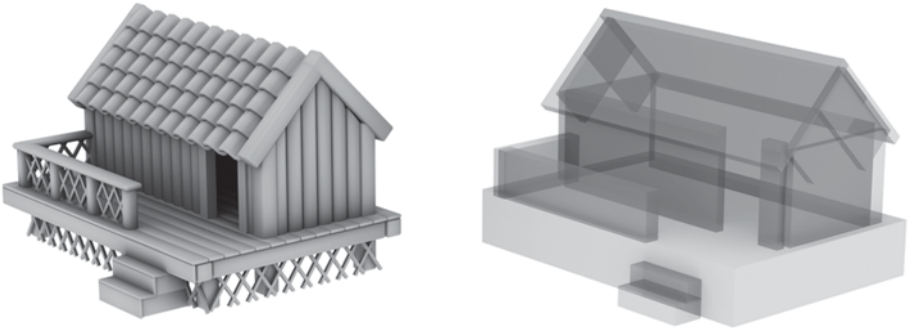
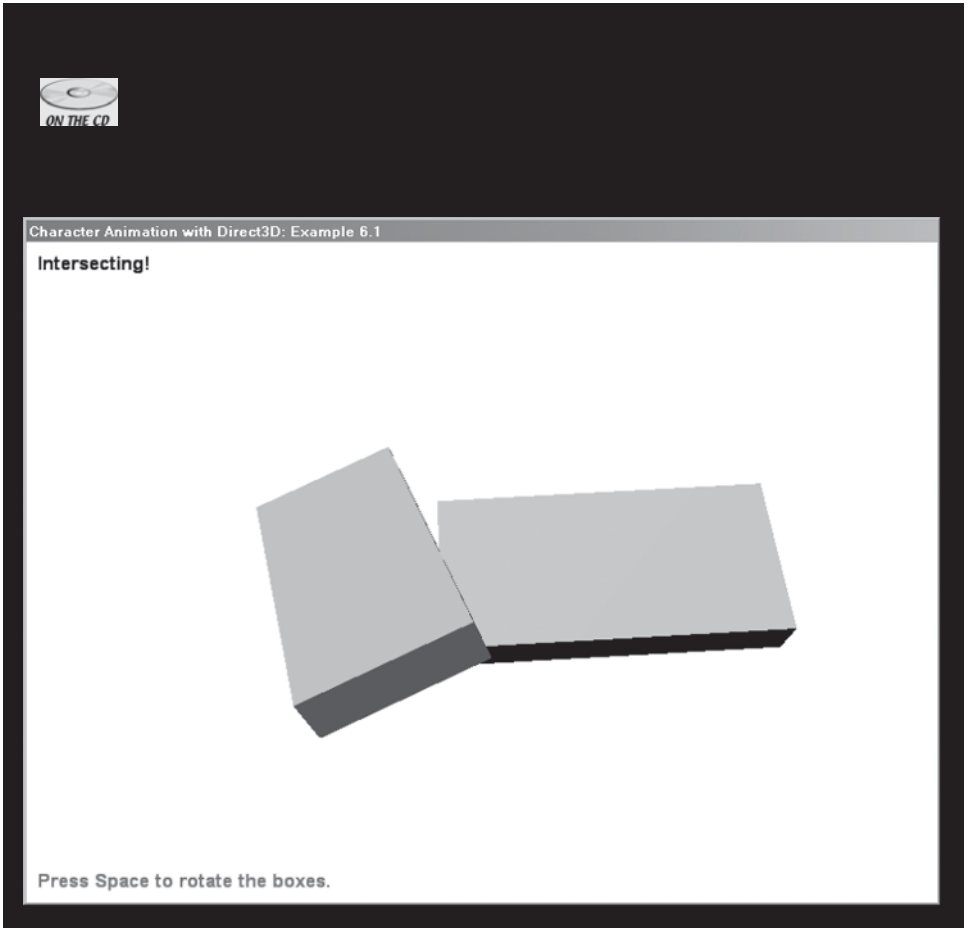


FIGURE 6.10
Example of an environment described using Oriented Bounding Boxes.

The mesh being rendered can have thousands of polygons and would therefore be very expensive to do collision detection with. The OBB representation, on the other hand, is very fast, without losing too much detail.



Physics Simulation

Remember that the long-term goal is to create a ragdoll animation and have a character fall/collide with objects in a realistic way. There's still a long way to go before realizing this, although a lot of things have now been covered. You know how to represent the environment (or at least how to create a simplified version of it). Now it is time to start looking at the interaction bit.

I guess a quick recap is in order. So far you have learned the theory of rigid bodies, how forces affect an object, the basics of quaternions, and how to describe the environment using Oriented Bounding Boxes. You will now finally get to put these lessons into practice. Now it is time to look at how to implement a few physical properties of a rigid body and how to update (i.e., simulate) these in a realistic manner. In this book I'll use the following interface to describe objects in the physics simulation.

```
class PHYSICS_OBJECT
{
    public:
        virtual void Update(float deltaTime) = 0;
        virtual void Render() = 0;
        virtual void AddForces() = 0;
        virtual void SatisfyConstraints(vector<OBB*> &obstacles) = 0;
};
```

The `Update()` function updates the position of the object, the `Render()` function renders it, and the `AddForces()` function adds forces to the objects such as gravity, wind, etc. The `SatisfyConstraints()` function takes a list of Oriented Bounding Boxes as a parameter. The OBBs are the physical representation of the world. The `SatisfyContraints()` function handles the object's collision with the world. Next, there's the `PHYSICS_ENGINE` class, which takes care of a collection of `PHYSICS_OBJECT` objects and runs a simulation with these in the physical representation of the world.

```
class PHYSICS_ENGINE
{
    public:
        PHYSICS_ENGINE();
        void Init();
        void Update(float deltaTime);
        void AddForces();
        void SatisfyConstraints();
        void Render();
        void Reset();

    private:
        vector<PHYSICS_OBJECT*> m_physicsObjects;
        vector<OBB*> m_obstacles;
        float m_time;
};
```

As you can see, this class contains a vector of physics objects and a list of OBBs. Each time step, every object in the physics simulation is updated in the following manner:

1. Add forces (gravity, wind).
2. Update position.
3. Satisfy constraints (check for collisions, and move object to a legal position).

That's pretty much all you need for a lightweight physics simulation. Note that there are light-years between a physics engine like this and a proper engine. For example, this "engine" doesn't handle things like object–object collision, etc. Next, you'll learn how the position of an object is updated, and after that the first physics object (the particle) will be implemented.

Position, Velocity, and Acceleration

Next I'll cover the three most basic physical properties of a rigid body: its position *p*, its velocity *v*, and its acceleration *a*. In 3D space, the position of an object is described with x, y, and z coordinates. The velocity of an object is simply how the position is changing over time. In just the same way, the acceleration of an object is how the velocity changes over time. The formulas for these behaviors are listed below.

$$p_n = p + v \bullet \Delta t$$

where p_n is the new position.

$$v_n = v + a \bullet \Delta t$$

where v_n is the new velocity.

$$a = \frac{f}{m}$$

In the case of a 3D object, these are all described as a 3D vector with an x, y, and z component. The following code shows an example class that has the `m_position`, `m_velocity`, and the `m_acceleration` vector. This class implements the formulas above using a constant acceleration:

```
class OBJECT{
    public:
        OBJECT()
        {
            m_position = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
            m_velocity = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
            m_acceleration = D3DXVECTOR3(0.0f, 1.0f, 0.0f);
        }

        void Update(float deltaTime)
        {
```

```
            m_velocity += m_acceleration * deltaTime;
            m_position += m_velocity * deltaTime;
        }

    private:
        D3DXVECTOR3 m_position;
        D3DXVECTOR3 m_velocity;
        D3DXVECTOR3 m_acceleration;
};
```

As you can see, the `m_position` and the `m_velocity` vectors are initialized to zero, while the `m_acceleration` vector is pointing in the Y-direction with a constant magnitude of one. Figure 6.11 shows a graph of how the acceleration, velocity, and position of an object like this change over time.

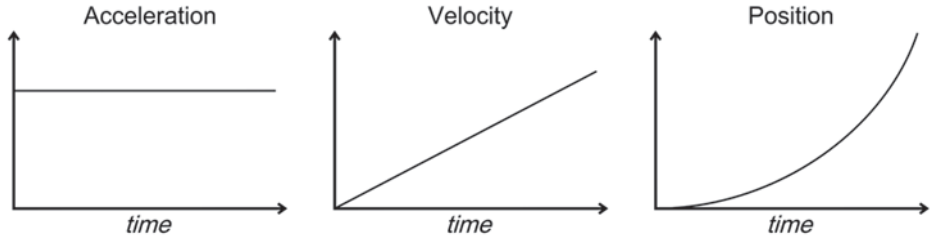


FIGURE 6.11
Acceleration, velocity, and position over time.

In this section you have looked at how an object's position is affected by velocity, which in turn is affected by its acceleration. In the same manner, an object's orientation is affected by its angular velocity, which in turn is affected by its torque. However, since you won't need the concepts of angular velocity and torque to create a simple ragdoll animation, I won't dive into the detailed math of these. If you want to look into the specifics of angular velocity and torque, I suggest reading the article "Integrating the Equations of Rigid Body Motion" by Miguel Gomez [Gomez00].

The Particle

These days, physicists use billions of volts to try and accelerate tiny electrically charged particles and collide them with each other. What you're about to engage in is, thankfully, much simpler than that (and also a lot less expensive). You'll be looking at the smallest (and simplest) entity you can simulate in a physics engine: the particle!

A particle can have a mass, but it does not have a volume and therefore it doesn't have an orientation either. This makes it a perfect physical entity for us beginners to start with. The particle system I am about to cover here is based on the article "Advanced Character Physics" by Thomas Jakobsen [Jakobsen03].

Instead of storing the velocity of a particle as a vector, you can also store it using an object's current position and its previous position. This is called *Verlet integration* and works like this:

$$p_n = 2p_c - p_o + a\,\Delta t$$
$$p_o = p_c$$

p_n is the new position, p_c is the current position, and p_o is the previous position of an object. After you calculate the new position of an object, you assign the previous position to the current. As you can see, there's no velocity in this formula since this is always implied using the difference between the current and the old position. In code, this can be implemented like this:

```
D3DXVECTOR3 temp = m_pos;
m_pos += (m_pos - m_oldPos) + m_acceleration * deltaTime * deltaTime;
m_oldPos = temp;
```

In this code snippet, `m_pos` is the current position of the particle, `m_oldPos` is the old position, and `m_acceleration` is the acceleration of the particle. This Verlet integration also requires that the `deltaTime` is fixed (i.e., not changing between updates). Although this method of updating an object may seem more complicated than the one covered previously, it does have some clear advantages. Most important of these advantages is that it makes it easier to build a stable physics simulation. This is due to the fact that if a particle suddenly were to collide with a wall and stop, its velocity would also become updated (i.e., set to zero), and the particle's velocity would no longer point in the direction of the wall.

The following code shows the `PARTICLE` class. As you can see, it extends the `PHYSICS_OBJECT` base class and can therefore be simulated by the `PHYSICS_ENGINE` class.

```
class PARTICLE : public PHYSICS_OBJECT
{

    public:
        PARTICLE();
        PARTICLE(D3DXVECTOR3 pos);
        void Update(float deltaTime);
        void Render();
```

```
        void AddForces();
        void SatisfyConstraints(vector<OBB*> &obstacles);

    private:
        D3DXVECTOR3 m_pos;
        D3DXVECTOR3 m_oldPos;
        D3DXVECTOR3 m_forces;
        float m_bounce;
};
```

The `m_bounce` member is simply a float between [0, 1] that defines how much of the energy is lost when the particle bounces against a surface. This value is also known as the "coefficient of restitution," or in other words, "bounciness." With a high bounciness value, the particle will act like a rubber ball, whereas with a low value it will bounce as well as a rock. The next thing you need to figure out is how a particle behaves when it collides with a plane (remember the world is described with OBBs, which in turn can be described with six planes).

> *"To describe collision response, we need to partition velocity and force vectors into two orthogonal components, one normal to the collision surface, and the other parallel to it."*
>
> [Witkin01]

This same thing is shown more graphically in Figure 6.12.

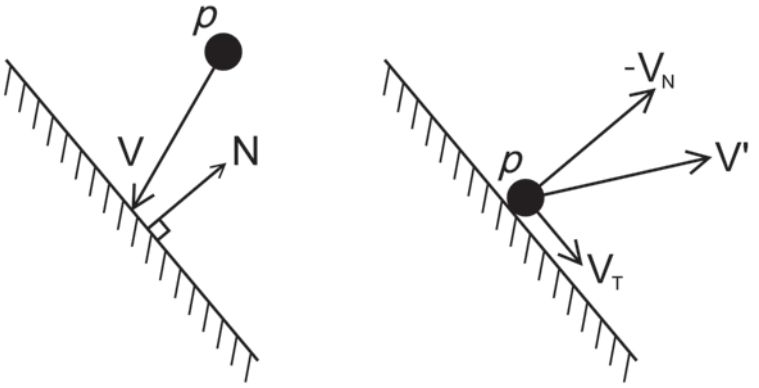


FIGURE 6.12
Particle and forces before collision (left).
Particle and forces after collision (right).

$V_N = (N \cdot V)N$

V_N is the current velocity projected onto the normal N of the plane.

$V_T = \mathrm{V} - \mathrm{V_N}$

V_T is the velocity parallel to the plane.

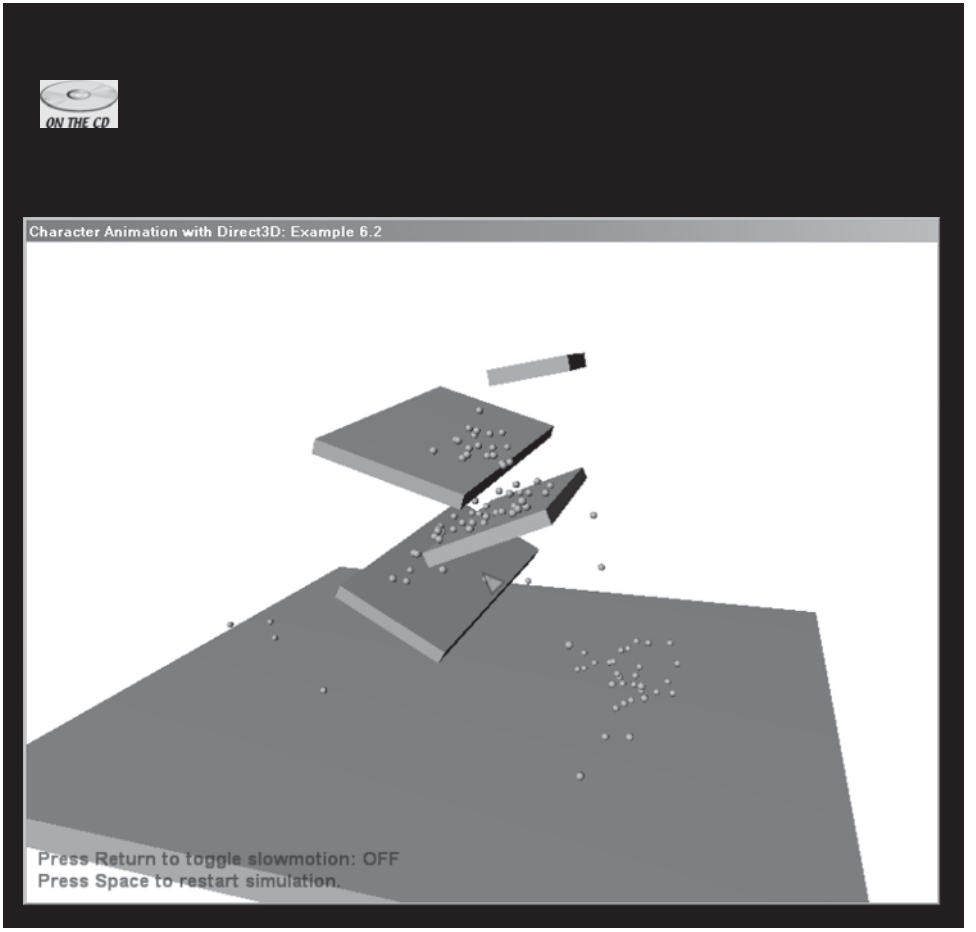
$V' = \mathrm{V_T} - \mathrm{V_N}$

Finally, V' is the resulting velocity after the collision.

The following code snippet shows you one way to implement this particle-plane collision response using Verlet integration. In this code I assume that a collision has occurred and that I know the normal of the plane with which the particle has collided.

```
//Calculate Velocity
D3DXVECTOR3 V = m_pos - m_oldPos;

//Normal Velocity
D3DXVECTOR3 VN = D3DXVec3Dot(&planeNormal, &V) * planeNormal;

//Tangent Velocity
D3DXVECTOR3 VT = V - VN;

//Change the old position (i.e. update the particle velocity)
m_oldPos = m_pos - (VT - VN * m_bounce);
```

First, the velocity of the particle was calculated by subtracting the position of the particle with its previous position. Next, the normal and tangent velocities were calculated using the formulas above. Since Verlet integration is used, you need to change the old position of the particle to make the particle go in a different direction the next update. You can also see that I have added the `m_bounce` variable to the calculation, and in this way you can simulate particles with different "bounciness."



The Spring

Next I'll show you how to simulate a spring in a physics simulation. In real life you find coiled springs in many different places such as car suspensions, wrist watches, pogo-sticks, etc. A spring has a resting length—i.e., the length when it doesn't try to expand or contract. When you stretch a spring away from this equilibrium length it will "pull back" with a force equivalent to the difference from its resting length. This is also known as Hooke's Law.

$$F = -kx$$

F is the resulting force, *k* is the spring constant (how strong/stiff the spring is), and x is the spring's current distance away from its resting length. Note that if the spring is already in its resting state, the distance will be zero and so will the resulting force. If an object is hanging from one end of a spring that has the other end attached to the roof, it will have an oscillating behavior whenever the object is moved away from the resting length, as shown in Figure 6.13.

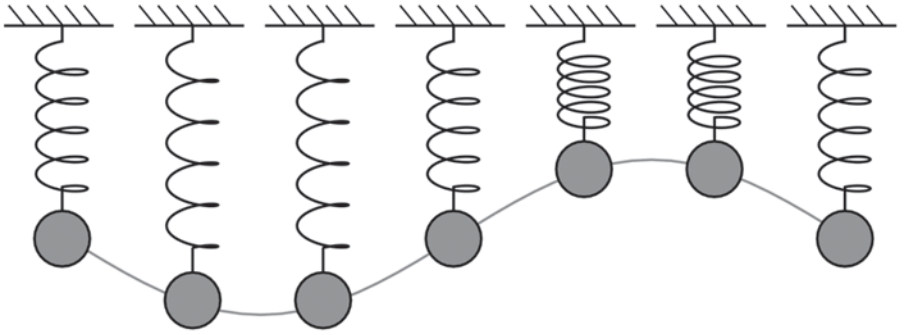
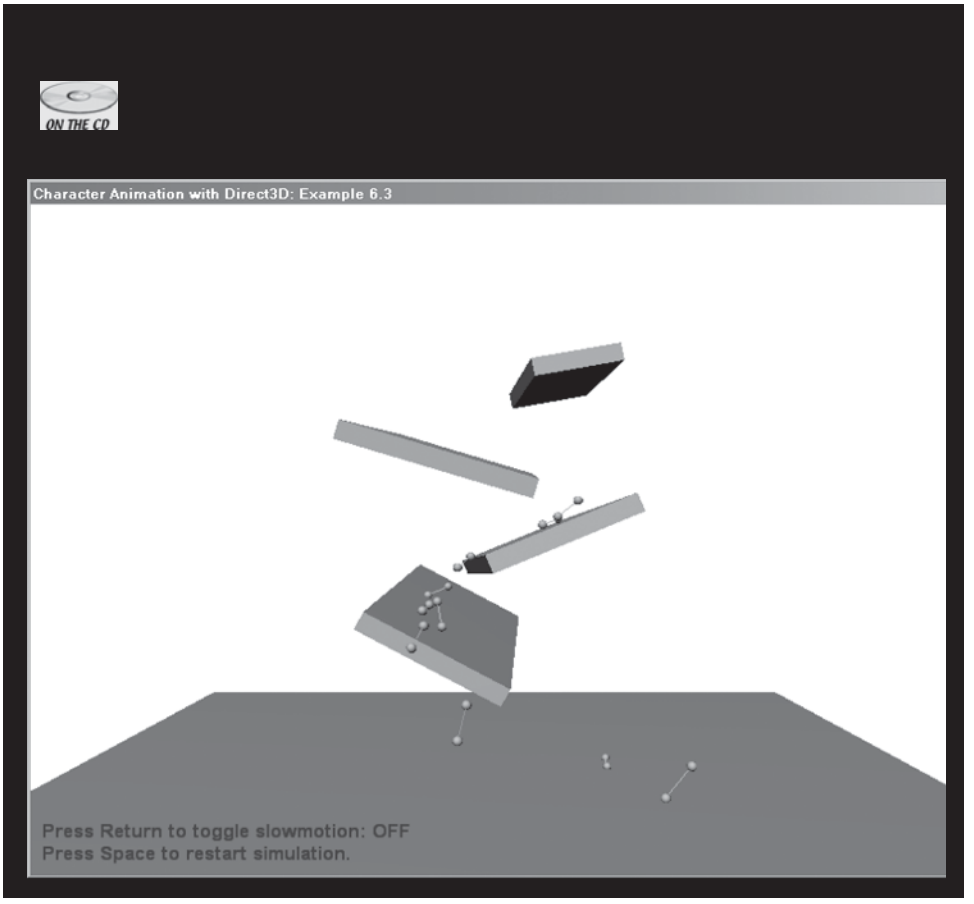


FIGURE 6.13
The oscillating behavior of a spring.

As you can see in Figure 6.13, the spring makes a nice sinus shaped motion over time. Eventually, however, friction will bring this oscillation to a stop (this is, incidentally, what forces us to manually wind up old mechanical clocks). In this book a specific subset of springs are of particular interest: springs with an infinite strength. If you connect two particles with a spring like this, the spring will automatically pull or push these particles together or apart until they are exactly at the spring's resting length from each other. Using springs with infinite strength can be used to model rigid volumes like tetrahedrons, boxes, and more. The following code snippet shows the implementation of the `SPRING` class. As you can see, you don't need anything other than pointers to two particles and a resting length for the spring. The `SPRING` class also extends the `PHYSICS_OBJECT` class and can therefore also be simulated by the `PHYSICS_ENGINE` class.

```
class SPRING : public PHYSICS_OBJECT
{
    public:
        SPRING(PARTICLE *p1, PARTICLE *p2, float restLength);
        void Update(float deltaTime){}
        void Render();
        void AddForces(){}
        void SatisfyConstraints(vector<OBB*> &obstacles);

    private:
        PARTICLE *m_pParticle1;
        PARTICLE *m_pParticle2;
        float m_restLength;
};

void SPRING::SatisfyConstraints(vector<OBB*> &obstacles)
{
    D3DXVECTOR3 delta = m_pParticle1->m_pos - m_pParticle2->m_pos;
    float dist = D3DXVec3Length(&delta);
    float diff = (dist-m_restLength)/dist;
    m_pParticle1->m_pos -= delta * 0.5f * diff;
    m_pParticle2->m_pos += delta * 0.5f * diff;
}
```

This code shows the `SPRING` class and its most important function, the `SatisfyConstraints()` function. In it, the two particles are forcibly moved to a distance equal to the resting length of the spring.



Conclusions

As promised, this chapter contained only the "bare bones" of the knowledge needed to create a physics simulation. It is important to note that all the code in this chapter has been written with clarity in mind, not optimization. The simplest and most straightforward way to optimize a physics engine like this is to remove all square root calculations. Write your own implementation of `D3DXVec3Length()`, `D3DXVec3Normalize()`, etc. using an approximate square root calculation. For more advanced physics simulations, you'll also need some form of space partitioning speeding up nearest neighbor queries, etc.

In this chapter the game world was described using Oriented Bounding Boxes. A basic particle system was simulated as well as particles connected with springs. Although it may seem like a lot more knowledge is needed to create a ragdoll

system, most of it has already been covered. All you need to do is start creating a skeleton built of particles connected with springs. In Chapter 7, a ragdoll will be created from an existing character using an open source physics engine.

Chapter 6 Exercises

- Implement particles with different mass and bounciness values and see how that affects the particles and springs in Example 6.3.
- Implement springs that don't have infinite spring strength (use Hooke's Law).
- Try to connect particles with springs to form more complex shapes and objects, such as boxes, etc.

Further Reading

[Eberly99] Eberly, David, "Dynamic Collision Detection Using Oriented Bounding Boxes." Available online at http://www.geometrictools.com/Documentation/DynamicCollisionDetection.pdf, 1999.

[Gomez99] Gomez, Miguel, "Simple Intersection Tests for Games." Available online at http://www.gamasutra.com/features/19991018/Gomez_1.htm, 1999.

[Gomez00] Gomez, Miguel, "Integrating the Equations of Rigid Body Motion." Game Programming Gems, Charles River Media, 2000.

[Ibanez01] Ibanez, Luis, "Tutorial on Quaternions." Available online at http://www.itk.org/CourseWare/Training/QuaternionsI.pdf, 2001.

[Jakobsen03] Jakobsen, Thomas, "Advanced Character Physics." Available online at http://www.gamasutra.com/resource_guide/20030121/jacobson_01.shtml, 2003.

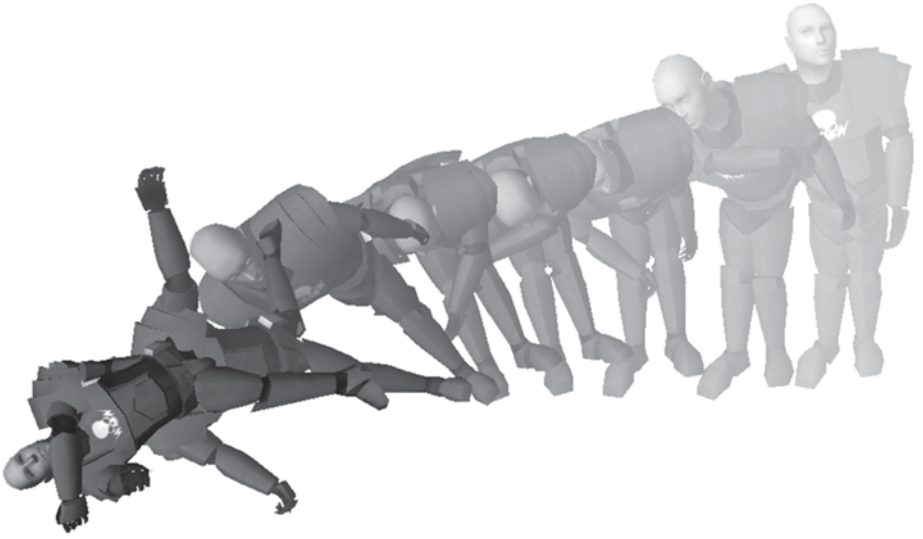
[Svarovsky00] Svarovsky, Jan, "Quaternions for Game Programming." Game Programming Gems, Charles River Media, 2000.

[Wikipedia] "Gravitational Constant." Available online at http://en.wikipedia.org/wiki/Gravitational_constant.

[Witkin01] Witkin, Andrew, "Physically Based Modeling, Particle System Dynamics." Available online at http://www.pixar.com/companyinfo/research/pbm2001/pdf/notesc.pdf, 2001.

This page intentionally left blank

7 Ragdoll Simulation



Ragdoll animation is a procedural animation, meaning that it is not created by an artist in a 3D editing program like the animations covered in earlier chapters. Instead, it is created in runtime. As its name suggests, ragdoll animation is a technique of simulating a character falling or collapsing in a believable manner. It is a very popular technique in first person shooter (FPS) games, where, for example, after an enemy character has been killed, he tumbles down a flight of stairs. In the previous chapter you learned the basics of how a physics engine works. However, the "engine" created in the last chapter was far from a commercial engine, completely lacking support for rigid bodies (other than particles). Rigid body physics is something you will need when you implement ragdoll animation. So, in this chapter I will race forward a bit

and make use of an existing physics engine. If you ever work on a commercial game project, you'll find that they use commercial (and pretty costly) physics engines such as Havok™, PhysX™, or similar. Luckily, there are several good open-source physics engines that are free to download and use. In this chapter I'll use the Bullet physics engine, but there are several other open-source libraries that may suit you better (for example, ODE, Box2D, or Tokamak). This chapter covers the following:

- Introduction to the Bullet physics engine
- Creating the physical ragdoll representation
- Creating constraints
- Applying a mesh to the ragdoll

Before you continue with the rest of this chapter, I recommend that you go to the following site:

http://www.fun-motion.com/physics-games/sumotori-dreams/

Download the amazing 87 kb Sumotori Dreams *game and give it a go (free download). It is a game where you control a Sumo-wrestling ragdoll. This game demo is not only an excellent example of ragdoll physics, but it is also quite fun! Figure 7.1 shows a screenshot of* Sumotori Dreams*:*

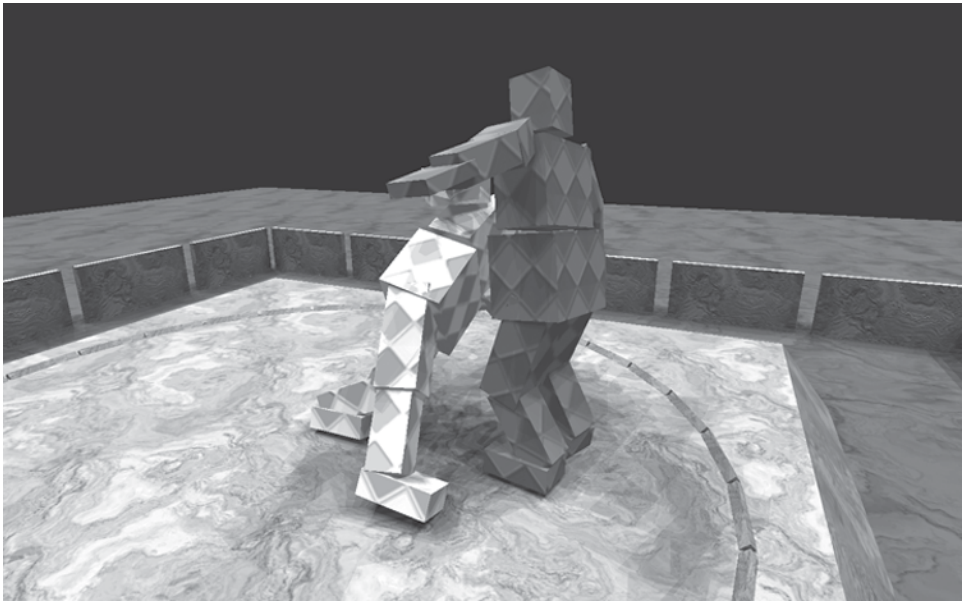


FIGURE 7.1
A screenshot of *Sumotori Dreams*.

Introduction to the Bullet Physics Engine

A complete physics engine is a huge piece of software that would take a single person a tremendous amount of time to create on his own. Luckily, you can take advantage of the hard work of others and integrate a physics engine into your game with minimum effort. This section serves as a step-by-step guide to installing and integrating the Bullet physics engine into a Direct3D application. The Bullet Physics Library was originally created by Erwin Coumans, who previously worked for the Havok project. Since 2005, the Bullet project has been open source, with many other contributors as well. To get Bullet up and running, you only need to know how to create and use the objects shown in Table 7.1.

TABLE 7.1 BULLET CORE CLASSES

As you can see, there are some classes with duplicated functionality when using Bullet together with DirectX. I have created a set of helper functions to easily convert some Bullet structures to the corresponding DirectX structures. These helper functions are listed below:

```
//Bullet Vector to DirectX Vector
D3DXVECTOR3 BT2DX_VECTOR3(const btVector3 &v)
{
    return D3DXVECTOR3(v.x(), v.y(), v.z());
}

//Bullet Quaternion to DirectX Quaternion
D3DXQUATERNION BT2DX_QUATERNION(const btQuaternion &q)
{
    return D3DXQUATERNION(q.x(), q.y(), q.z(), q.w());
}

//Bullet Transform to DirectX Matrix
D3DXMATRIX BT2DX_MATRIX(const btTransform &ms)
{
    btQuaternion q = ms.getRotation();
    btVector3 p = ms.getOrigin();

    D3DXMATRIX pos, rot, world;
    D3DXMatrixTranslation(&pos, p.x(), p.y(), p.z());
    D3DXMatrixRotationQuaternion(&rot, &BT2DX_QUATERNION(q));
    D3DXMatrixMultiply(&world, &rot, &pos);

    return world;
}
```

As you can see in these functions, the information in the Bullet vector and quaternion classes are accessed through function calls. So you simply create the corresponding DirectX containers using the data from the Bullet classes. However, before you can get this code to compile, you need to set up your project and integrate the Bullet library.

Integrating the Bullet Physics Library

This section describes in detail the steps required to integrate the Bullet Physics Library to your own Direct3D project. You can also find these steps described in detail in the Bullet user manual (part of the library download).

Download Bullet

The first thing you need to do is to download Bullet from:
http://www.bulletphysics.com

At the time of writing, the latest version of the Bullet Physics Library was 2.68.

After you have downloaded the library (**`bullet-2.68.zip`**, in my case), unpack it somewhere on your hard drive. I will assume that you unpacked it to **"`C:\Bullet`"** and will use this path throughout the book (You can of course put the Bullet library wherever it suits you best). A screenshot of the folder structure can be seen in Figure 7.2.

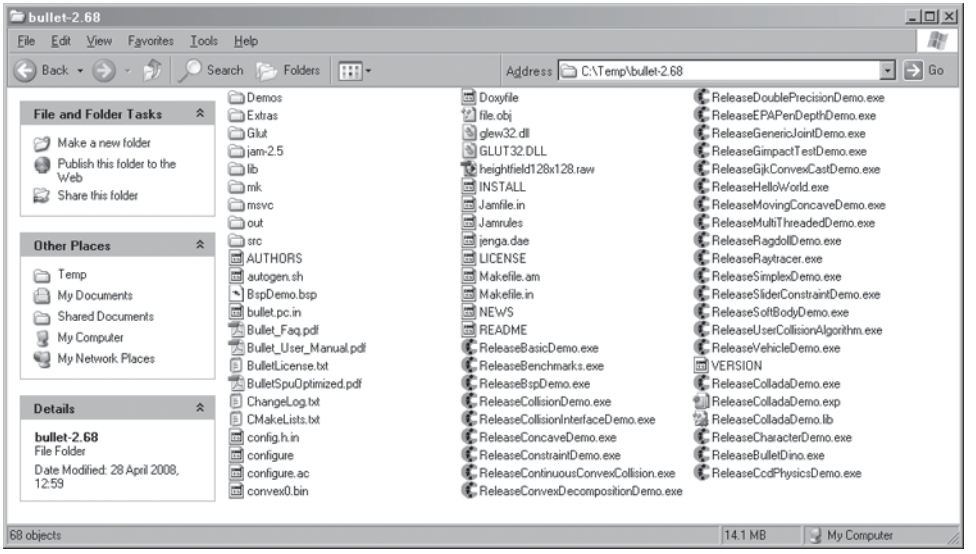


FIGURE 7.2
The Bullet Physics Library.

The example exe files will not be available until you have made your first build of the Bullet physics engine.

Build the Bullet Libraries

The next thing you need to do is to compile the Bullet libraries. In the root folder of the Bullet library, find the **"`C:\Bullet\msvc`"** folder and open it. In it you'll find project folders for Visual Studio 6, 7, 7.1, and 8. Select the folder corresponding to your version of Visual Studio and open up the **`wksbullet.sln`** solution file located therein.

This will fire up Visual Studio with the Bullet project. You will see a long list of test applications in the solution explorer. These are a really good place to look for example source code, should you get stuck with a particular problem.

Next, make a release build of the entire solution and sit back and wait for it to finish (it takes quite a while). Press the Play button to start a collection of Bullet examples. Be sure to check these out before moving on (especially the Ragdoll example, as seen in Figure 7.3).

FIGURE 7.3
The Bullet Ragdoll example.

Not only did you build all these cool physics test applications when compiling the Bullet solution, you also compiled the libraries (lib files) you will use in your own custom applications. Assuming you put the Bullet library in **`"C:\Bullet"`** and that you use Visual Studio 8, you will find the compiled libraries in **`"C:\Bullet\out\release8\libs"`**.

Setting Up a Custom Direct3D Project

The next thing you need to do is create a new project and integrate Bullet into it. First make sure the Bullet include files can be found. You do this by adding the Bullet source folder to the VC++ directories, as shown in Figure 7.4. You will find this menu by clicking the Options button in the Tools drop-down menu in Visual Studio.

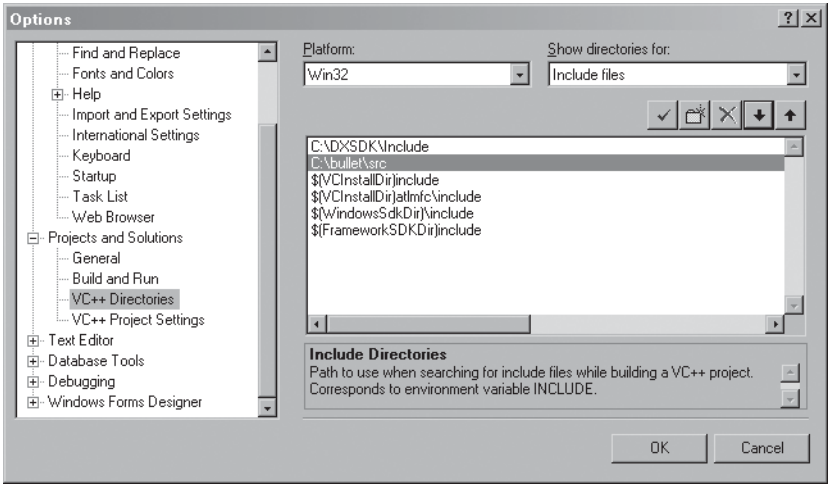


FIGURE 7.4
Adding the Bullet source folder to the VC++ directories.

Select the "Include Files" option from the "Show Directories for" drop-down menu. Create a new entry and direct it to the **"C:\Bullet\src"** folder. You also need to repeat this process but for the library folder. Select "Library Files" from the "Show Directories for"-dropdown menu. Create a new entry in the list and direct it to **"C:\Bullet\out\release8\libs"**.

Then link the Bullet libraries to the project. You do this through the project properties (Alt + F7), or by clicking the Properties button in the Project drop-down menu. That will open the Properties menu shown in Figure 7.5.

Now find the "Linker – Input" option in the left menu. In the "Additional Dependencies" field to the right, add the following library files: **libbulletdynamics.lib**, **libbulletcollision.lib**, and **libbulletmath.lib**, as shown in Figure 7.5.

Finally you need to include the **btBulletDynamicsCommon.h** header file in any of your source files making use of the Bullet library classes. After following these directions, you should now be ready to create and build your own physics application.

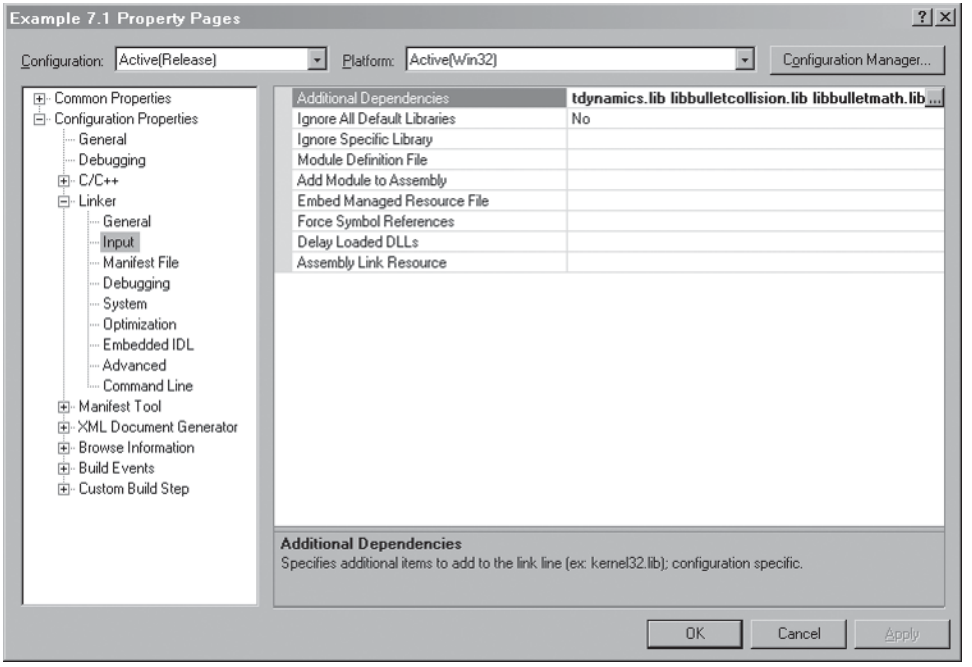


FIGURE 7.5
The Project Properties menu.

HELLO BTDYNAMICSWORLD

In this section you will learn how to set up a `btDynamicsWorld` object and get started with simulating physical objects. The `btDynamicsWorld` class is the high-level interface you'll use to manage rigid bodies and constraints, and to update the simulation. The default implementation of the `btDynamicsWorld` is the `btDiscreteDynamicsWorld` class. It is this class I will use throughout the rest of this book, or at least for the parts concerning physics (see the Bullet documentation for more information about other implementations). The following code creates a new `btDiscreteDynamicsWorld` object:

```
//New default Collision configuration
btDefaultCollisionConfiguration *cc;
cc = new btDefaultCollisionConfiguration();

//New default Constraint solver
btConstraintSolver *sl;
sl = new btSequentialImpulseConstraintSolver();
```

```
//New axis sweep broadphase
btVector3 worldAabbMin(-1000,-1000,-1000);
btVector3 worldAabbMax(1000,1000,1000);
const int maxProxies = 32766;
btBroadphaseInterface *bp;
bp = new btAxisSweep3(worldAabbMin, worldAabbMax, maxProxies);

//New dispatcher
btCollisionDispatcher *dp;
dp = new btCollisionDispatcher(cc);

//Finally create the dynamics world
btDynamicsWorld* dw;
dw = new btDiscreteDynamicsWorld(dp, bp, sl, cc);
```

As you can see, you need to specify several other classes in order to create a `btDiscreteDynamicsWorld` object. You need to create a Collision Configuration, a Constraint Solver, a Broadphase Interface, and a Collision Dispatcher. All these interfaces have different implementations and can also be custom implemented. Check the Bullet SDK for more information on all these classes and their variations.

Next I'll show you how to add a rigid body to the world and finally how you run the simulation. To create a rigid body, you need to specify four things: mass, motion state (starting world transform), collision shape (box, cylinder, capsule, mesh, etc.) and local inertia. The following code creates a rigid body (a box) and adds it to the dynamics world:

```
//Create Starting Motion State
btQuaternion q(0.0f, 0.0f, 0.0f);
btVector3 p(51.0f, 30.0f, -10.0f);
btTransform startTrans(q, p);
btMotionState *ms = new btDefaultMotionState(startTrans);

//Create Collision Shape
btVector3 size(1.5f, 2.5f, 0.75f);
btCollisionShape *cs = new btBoxShape(size);

//Calculate Local Inertia
float mass = 35.0f;
btVector3 localInertia;
cs->calculateLocalInertia(mass, localInertia);
```

```
//Create Rigid Body
btRigidBody *body = new btRigidBody(mass, ms, cs, localInertia);

//Add the new body to the dynamics world
pDynamicsWorld->addRigidBody(body);
```

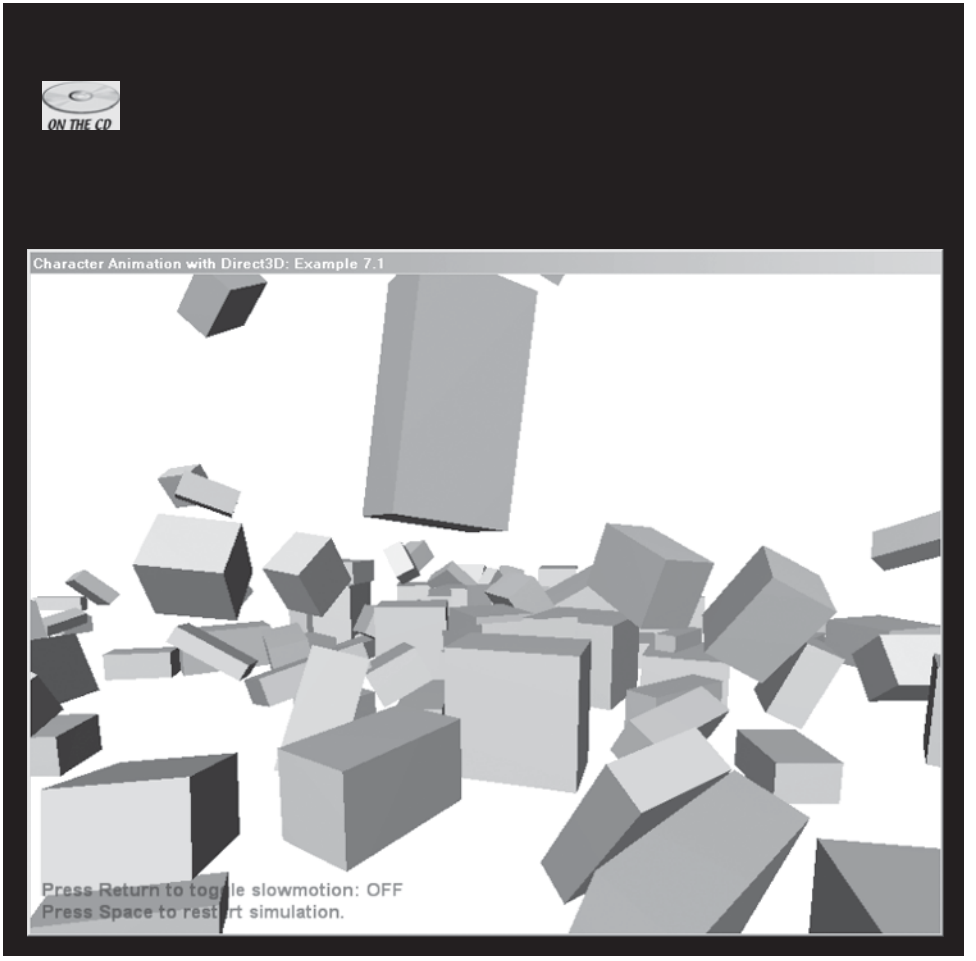
Then to run the simulation all you need to do is call the `stepSimulation()` function each frame, like this:

```
pDynamicsWorld->stepSimulation(deltaTime);
```

That's it! The rigid body you have now created will be simulated each frame, colliding with other rigid bodies, etc. However, you still won't see anything on the screen because the box is nothing but a logical representation. You need to hook up the current motion state of the rigid body to a mesh. In the next example I create a rigid body for the Oriented Bounding Box (OBB) class that was covered in the previous chapter. So, each frame the rigid body is updated using the Bullet physics library and then rendered using DirectX like this:

```
void OBB::Render()
{
    //Get Motion state from rigid body
    btMotionState *ms = m_pBody->getMotionState();
    if(ms == NULL)return;

    //Convert the motion state to a DX matrix
    //and use it to set the world transform
    pEffect->SetMatrix("matW", &BT2DX_MATRIX(*ms));

    //Render the mesh as usual using whichever lighting technique
    D3DXHANDLE hTech = pEffect->GetTechniqueByName("Lighting");
    pEffect->SetTechnique(hTech);
    pEffect->Begin(NULL, NULL);
    pEffect->BeginPass(0);
    m_pMesh->DrawSubset(0);
    pEffect->EndPass();
    pEffect->End();
}
```

You'll find the source code for the first example using the Bullet physics engine in Example 7.1.



Constraints

After that little detour of getting to know the Bullet physics library, it is time to get back to what I was trying to achieve in this chapter: ragdoll animation! You need to choose some of the major bones of the character and create a physical representation for them that can be simulated in the physics engine. Then as the ragdoll is simulated, the position and orientation are updated for the bones using the transforms taken from the rigid bodies, before rendering the mesh. Sounds easy? Well, not quite. It takes a lot of effort to make ragdoll animation look good and free from artifacts. You should also note that not all bones in the character are simulated (such as finger bones and other small bones). Figure 7.6 shows a picture of a character and its ragdoll setup.

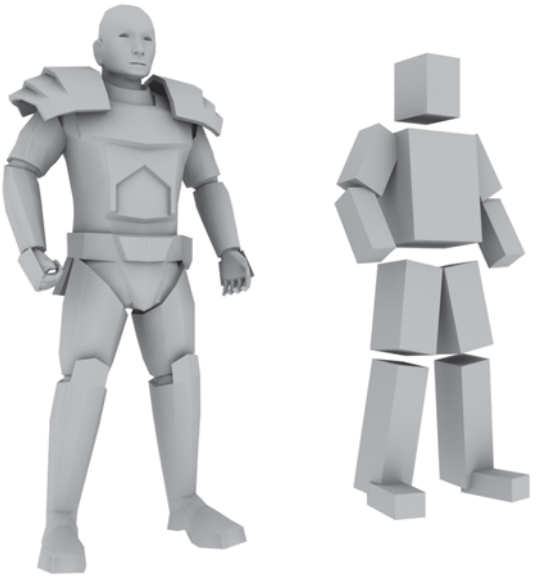


FIGURE 7.6
An example ragdoll setup.

You don't necessarily have to use boxes as in Figure 7.6. You can also use cylinders, capsules, or any other shape you see fit. Later on I will cover in more detail how to position the shapes to match the skeleton. However, should you run the simulation after just placing boxes, they would all fall to the floor, disjointed from each other. You need some form of "connection" between the under arm and the upper arm, for example. This is where constraints come into the picture. A constraint is just what it sounds like; it is a rule telling two rigid bodies how they can move in relation to each other. The two simplest forms of constraints are shown in Figure 7.7.

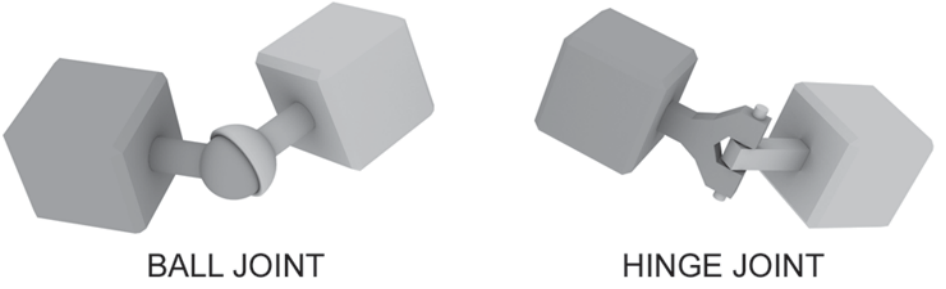


FIGURE 7.7
The ball and hinge joints.

Bullet supports the following constraints:

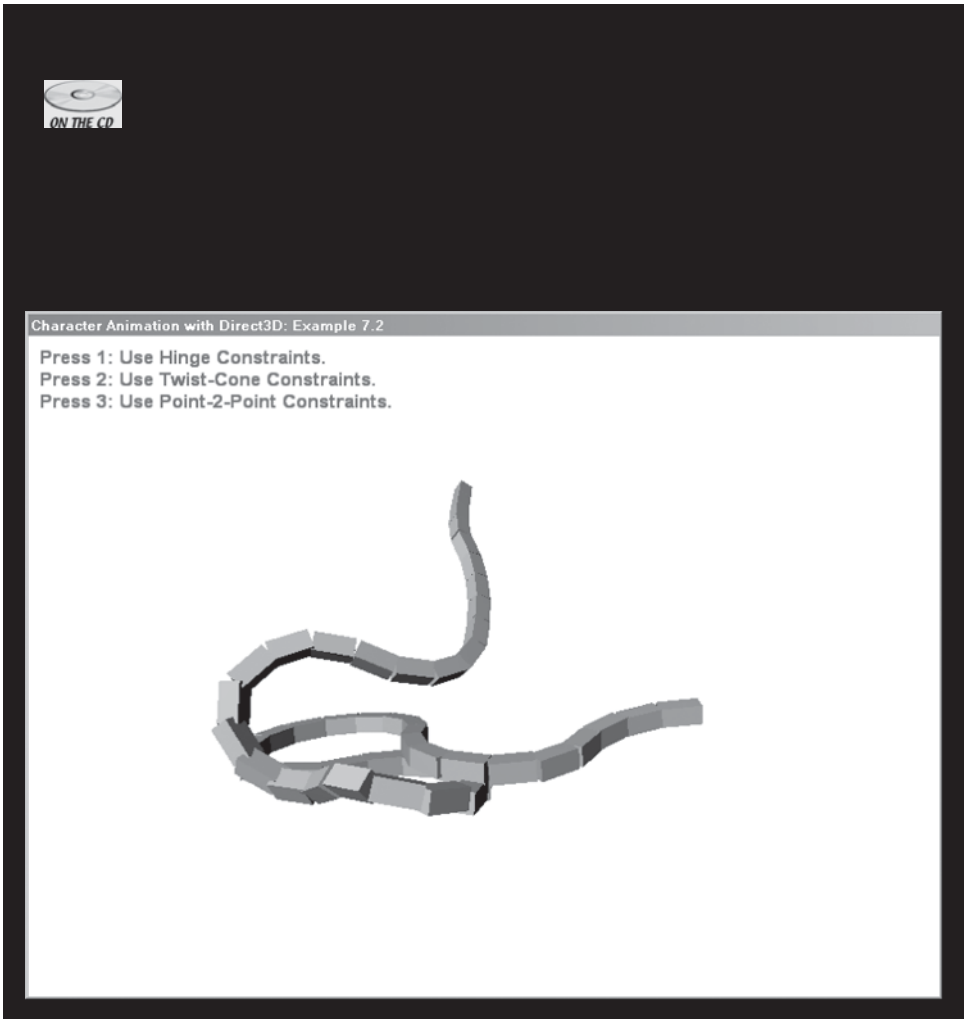
- Point-to-point constraint (a.k.a. ball joint)
- Hinge constraint
- Twist cone constraint
- 6 degrees of freedom (DoF) constraint

The ball joint doesn't have any restrictions on angle or twist amount, whereas the hinge joint allows no twisting of the connected rigid bodies. The twist cone is a mix between the ball and hinge joints. With the twist cone constraint, you can specify the angle range and twist amount allowed (which is very useful when creating a ragdoll). With the 6DoF constraint, you can specify exactly the angle ranges of each DoF. This is a bit more functionality than you need to implement a simple ragdoll animation, but check out the Bullet SDK for more information on these constraints.

Here's how you would create a simple constraint with the Bullet physics engine. Let's assume you have two rigid bodies (`A` and `B`) created as shown in the previous example. You would create a hinge constraint between them like this:

```
//Set transforms and axis for the hinge (for each rigid body)
btTransform localA, localB;
localA.setIdentity();
localB.setIdentity();
localA.getBasis().setEulerZYX(0,0,0);
localA.setOrigin(btVector3(0.0f, -0.5f, 0.0f));
localB.getBasis().setEulerZYX(0,0,0);
localB.setOrigin(btVector3(0.0f, 0.5f, 0.0f));

//Create Hinge Constraint
btHingeConstraint *hingeC;
hingeC = new btHingeConstraint(A, B, localA, localB);
hingeC->setLimit(-0.5f, 0.5f);

//Add the constraint to the dynamics world
pDynamicsWorld->addConstraint(hingeC, true);
```

That's how simple it is to add a constraint between two rigid bodies. The constraint limit specifies how much the hinge can bend back and forth. Example 7.2 shows you how this is done with the twist cone and ball joint constraints.



Constructing the Ragdoll

Now that you know how to connect physical rigid bodies using constraints, the next step of creating a ragdoll is not far off…in theory. All you have to do is to create a series of boxes (or other shapes) connected via different constraints. However, in practice it is slightly more difficult than that. First you need to make sure that the boxes match the skeleton (and the character mesh) as close to perfect as possible. Otherwise you would get weird results updating the bone hierarchy of the character using an ill-fitting physical representation. So the problem you are about to face is the one shown in Figure 7.8.

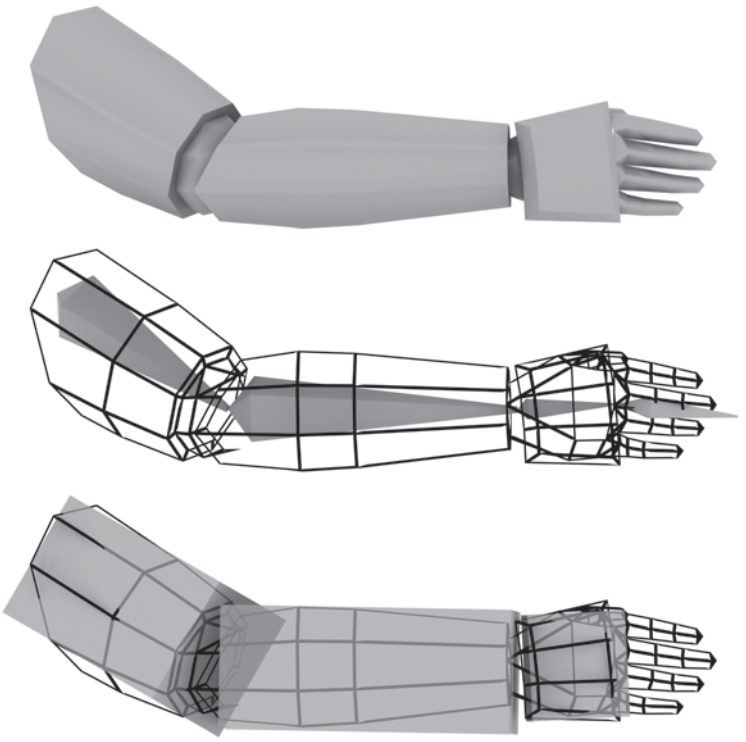


FIGURE 7.8
Solid character arm mesh (top). Arm wireframe and bones (middle). Arm wireframe and physical representation (bottom).

In Figure 7.8 you see a part of a character—namely, an arm. For a working ragdoll animation you must create a physical representation of the character that you can simulate in your physics engine. At each frame you update the skeleton of the character to match the physical representation and, *voila*!, You've got yourself a ragdoll. For the purpose of creating, updating, simulating, and rendering a ragdoll, I've created the following class with the somewhat unimaginative name `RagDoll`:

```
class RagDoll : public SkinnedMesh
{
    public:
    RagDoll(char fileName[], D3DXMATRIX &world);
    ~RagDoll();
    void InitBones(Bone *bone);
    void Release();
    void Update(float deltaTime);
```

```
    void Render();
    void UpdateSkeleton(Bone* bone);
    OBB* CreateBoneBox(Bone* parent, Bone *bone,
                       D3DXVECTOR3 size, D3DXQUATERNION rot);
    void CreateHinge(Bone* parent, OBB* A, OBB* B,
                     float upperLimit, float lowerLimit,
                     D3DXVECTOR3 hingeAxisA, D3DXVECTOR3 hingeAxisB,
                     bool ignoreCollisions=true);
    void CreateTwistCone(BONE* parent, OBB* A, OBB* B,
                     float limit, D3DXVECTOR3 hingeAxisA,
                     D3DXVECTOR3 hingeAxisB,
                     bool ignoreCollisions=true);

    private:
    vector<OBB*> m_boxes;         //Boxes for physics simulation
};
```

I'll cover the more technical functions and creation of this class throughout the coming sections. But first there are some challenges you'll face when approaching this problem. For example, how do you place the Oriented Bounding Boxes so that they fit the mesh as closely as possible? You could, of course, attempt an algorithmic approach. This might be best if you need to create physical representations for a large number of characters, or if your characters are generated or randomized in some way. In that case you should probably traverse through the bones and determine which ones are big enough to merit a physical representation (remember, small bones like fingers are ignored). Next you would have to find the vertices linked to this bone and, for example, use Principal Component Analysis (PCA) to fit an Oriented Bounding Box to the bone and its vertices [VanVerth04]. This is outside the scope of this book, however, so I'll stick with the old-fashioned way of doing things: "by hand."

Even the "by hand" approach will need some supporting calculations to place the Oriented Bounding Box as optimally as possible. See Figure 7.9.

With the "by hand" fitting scheme I will only supply the size of the Oriented Bounding Box and use the orientation of the bone itself. Having the size and the orientation, you only need to calculate the position of the OBB before you can place it in the world. Figure 7.9 shows a simplified 2D image of a character's arm. If you need to place the upper arm bounding box, you just take the two end points (points A and B) of the upper arm bone and place the bounding box at the middle point of these two end points. The following piece of code comes from the Ragdoll class and does just this:

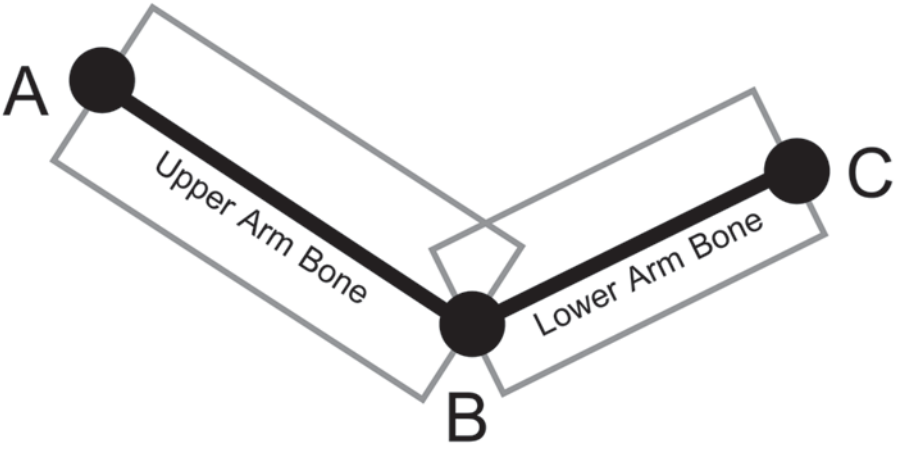


FIGURE 7.9
Fitting an OBB to a bone.

```
struct Bone: public D3DXFRAME
{
D3DXMATRIX CombinedTransformationMatrix;
OBB *m_pObb;
};

...

OBB* RagDoll::CreateBoneBox(Bone* parent, Bone *bone,
                            D3DXVECTOR3 size, D3DXQUATERNION rot)
{
    if(bone == NULL || parent == NULL)
        return NULL;

    //Get bone starting point
    D3DXMATRIX &parentMat = parent->CombinedTransformationMatrix;
    D3DXVECTOR3 parentPos(parentMat(3, 0),
                          parentMat(3, 1),
                          parentMat(3, 2));

    //Get bone end point
    D3DXMATRIX &boneMat = bone->CombinedTransformationMatrix;
    D3DXVECTOR3 bonePos(boneMat(3, 0), boneMat(3, 1), boneMat(3, 2));
```

```
        //Extract the rotation from the bone
        D3DXQUATERNION q;
        D3DXVECTOR3 p, s;
        D3DXMatrixDecompose(&s, &q, &p, &parentMat);

        //Offset rotation (in some cases only)
        q *= rot;
        D3DXQuaternionNormalize(&q, &q);

        //Calculate the middle point
        p = (parentPos + bonePos) * 0.5f;

        //Create new OBB
        OBB *obb = new OBB(p, size, q, true);

        //Add the OBB to the physics engine
        physicsEngine.GetWorld()->addRigidBody(obb->m_pBody);

        //Add OBB to the ragdoll's own list
        m_boxes.push_back(obb);

        //Connect the bone to the OBB
        parent->m_pObb = obb;

        return obb;
    }
```

As you can see, I've added a pointer to an OBB in the `Bone` structure. Each bone now has a pointer to an Oriented Bounding Box. Through this pointer the bone can retrieve the current position and orientation of the physical representation as the simulation runs. Other than this, the OBB is created and placed as explained earlier. Creating the Oriented Bounding Boxes is, of course, only the first step. If you run the physics simulation now, you would see the boxes fall to the floor disjointed from each other. Next you'll need to connect them in a proper manner before you have a ragdoll. This is the real tricky part and the hardest part to get right (i.e., to produce good-looking results). As covered earlier in Example 7.2, I'll use the hinge and twist cone constraints to hold the boxes in place. Take another look at Figure 7.9. When you place the constraints you will now place them in between the boxes instead, in the points A, B, and C. The following function in the Ragdoll class creates a twist cone constraint between two Oriented Bounding Boxes (a similar function exists to create a hinge constraint):

```
void RagDoll::CreateTwistCone(Bone* parent, OBB* A, OBB* B,
                        float limit, D3DXVECTOR3 hingeAxisA,
                        D3DXVECTOR3 hingeAxisB, bool ignoreCollisions)
{
    if(parent == NULL || A == NULL || B == NULL)
        return;

    //Extract the constraint position
    D3DXMATRIX &parentMat = parent->CombinedTransformationMatrix;
    btVector3 hingePos(parentMat(3, 0),
                       parentMat(3, 1),
                       parentMat(3, 2));

    D3DXVECTOR3 hingePosDX(parentMat(3, 0),
                           parentMat(3, 1),
                           parentMat(3, 2));

    //Get references to the two rigid bodies you want to connect
    btRigidBody *a = A->m_pBody;
    btRigidBody *b = B->m_pBody;

    //Get world matrix from the two rigid bodies
    btTransform aTrans, bTrans;
    a->getMotionState()->getWorldTransform(aTrans);
    b->getMotionState()->getWorldTransform(bTrans);
    D3DXMATRIX worldA = BT2DX_MATRIX(aTrans);
    D3DXMATRIX worldB = BT2DX_MATRIX(bTrans);

    //Calculate pivot point for both rigid bodies
    D3DXVECTOR3 offA, offB;
    D3DXMatrixInverse(&worldA, NULL, &worldA);
    D3DXMatrixInverse(&worldB, NULL, &worldB);
    D3DXVec3TransformCoord(&offA, &hingePosDX, &worldA);
    D3DXVec3TransformCoord(&offB, &hingePosDX, &worldB);
    btVector3 offsetA(offA.x, offA.y, offA.z);
    btVector3 offsetB(offB.x, offB.y, offB.z);

    //Set constraint axis
    aTrans.setIdentity();
    bTrans.setIdentity();
    aTrans.setOrigin(offsetA);
    bTrans.setOrigin(offsetB);
    aTrans.getBasis().setEulerZYX(
                    hingeAxisA.x, hingeAxisA.y, hingeAxisA.z);
```

```
    bTrans.getBasis().setEulerZYX(
                    hingeAxisB.x, hingeAxisB.y, hingeAxisB.z);

    //Create new twist cone constraint
    btConeTwistConstraint *twistC;
    twistC = new btConeTwistConstraint(*a, *b, aTrans, bTrans);

    //Set Constraint limits
    twistC->setLimit(limit, limit, 0.05f);

    //Add constraint to the physics engine
    physicsEngine.GetWorld()->addConstraint(twistC, true);
}
```

This function is generally pretty straightforward. The only tricky thing in here is to calculate the pivot point for the two rigid bodies. Since the pivot point needs to be in the local space of the rigid body, you have to multiply the world-space location of the pivot point with the inverse of the rigid body's world matrix. Next I set the constraint axes and limits, and finally the constraint is added to the physics simulation. There's now only one final thing left to do, and that is to make use of these two functions and create the ragdoll. The following is an excerpt from the constructor of the `RagDoll` class:

```
RagDoll::RagDoll(char fileName[], D3DXMATRIX &world) : SkinnedMesh()
{
    //Load the character from an .x file
    SkinnedMesh::Load(fileName);

    //Set beginning pose
    SetPose(world);

    //Find bones to use in the construction of the ragdoll
    //...
    Bone* U_R_Arm=(Bone*)D3DXFrameFind(m_pRootBone,"Upper_Arm_Right");
    Bone* L_R_Arm=(Bone*)D3DXFrameFind(m_pRootBone,"Lower_Arm_Right");
    Bone* R_Hand=(Bone*)D3DXFrameFind(m_pRootBone, "Hand_Right");
    //...

    D3DXQUATERNION q;
    D3DXQuaternionIdentity(&q);
```

```
    //...
    //Right arm (two bounding boxes)
    OBB* o03 = CreateBoneBox(U_R_Arm, L_R_Arm,
                             D3DXVECTOR3(0.3f, 0.12f, 0.12f), q);
    OBB* o04 = CreateBoneBox(L_R_Arm, R_Hand,
                             D3DXVECTOR3(0.3f, 0.12f, 0.12f), q);
    //...

    //Constraints
    //...
    CreateTwistCone(U_R_Arm, o08, o03, D3DX_PI * 0.6f,
                    D3DXVECTOR3(0.0f, D3DX_PI * 0.75f, 0.0f),
                    D3DXVECTOR3(0.0f, 0.0f, 0.0f));

    CreateHinge(L_R_Arm, o03, o04, 0.0f, -2.0f,
                D3DXVECTOR3(0.0f, 0.0f, D3DX_PI * 0.5f),
                D3DXVECTOR3(0.0f, 0.0f, D3DX_PI * 0.5f));
}
```

To keep this code excerpt from taking up too many pages, I show only how the arm of the ragdoll is set up. You'll find the complete initialization of the ragdoll in the next example. However, as you can see, the character is first loaded as a skinned mesh from an .x file. Next I set the pose I want to use as the bind pose for the character while creating the ragdoll. Then I use the `CreateBoneBox()`, `CreateTwistCone()`, and `CreateHinge()` functions to create the full physical representation of the character. As always, you'll find the full code for the ragdoll setup in Example 7.3.

Updating the Character Mesh from the Ragdoll

The final step before having complete ragdoll animation is of course to connect the bone hierarchy to the physical representation. This way, the bones (and in turn, the mesh) will be updated as the ragdoll flails about. Remember that you use many fewer Oriented Bounding Boxes than you use bones to avoid simulating the small unnecessary bones like fingers and toes, etc. However, if the hand bone (the parent of a finger bone) has a physical representation, then when the combined transformation matrix of this bone is updated, its child bone(s) will also be updated. Already in the previous section of this chapter I added a pointer to an OBB in the `Bone` structure. Now that I want to be able to update a bone hierarchy, some more information is needed in this structure:

```
struct Bone: public D3DXFRAME
{
    D3DXMATRIX CombinedTransformationMatrix;

    //Pointer to an OBB (if any)
    OBB *m_pObb;

    //The bone's pivot point (offset from OBB center)
    D3DXVECTOR3 m_pivot;

    //Original orientation of this bone
    D3DXQUATERNION m_originalRot;
};
```

You'll see later how I use the pivot point and the original orientation of the bone to calculate the new position and orientation based on the bone's OBB. I have already covered the creation of the Oriented Bounding Boxes and how they are assigned to their corresponding bones. Now all you need to do is calculate the new bone matrices as the boxes are simulated by the physics engine. First off, let's start with the position of the bones.

Getting a Bone's Position from an OBB

Before you continue further, have another look at how the revised `OBB` class now looks (a lot has changed in this class since Chapter 6). Some of the ragdoll-specific functions of this class will be explained in further detail throughout the rest of this chapter.

```
class OBB
{
    public:
        OBB(D3DXVECTOR3 pos,
            D3DXVECTOR3 size,
            bool dynamic=true);

        OBB(D3DXVECTOR3 pos,
            D3DXVECTOR3 size,
            D3DXQUATERNION rot,
            bool dynamic=true);

        void Init(D3DXVECTOR3 pos,
                  D3DXVECTOR3 size,
                  D3DXQUATERNION rot,
                  bool dynamic=true);
```

```
        ~OBB();
        void Release();
        void Update(float deltaTime);
        void Render();
        D3DXVECTOR3 SetPivot(D3DXVECTOR3 pivot);
        D3DXVECTOR3 GetPivot(D3DXVECTOR3 pivot);
        D3DXQUATERNION GetRotation(D3DXQUATERNION orgBoneRot);

    public:
        btRigidBody *m_pBody;
        D3DXVECTOR3 m_size;
        D3DXQUATERNION m_orgRot;

    private:
        ID3DXMesh *m_pMesh;
};
```

To be able to calculate the bone position on the fly from an OBB's transformation matrix, you need to first calculate the bone's pivot point. This is, in other words, the position of the bone in the OBB world space. The pivot point is calculated at the initialization of the ragdoll and then used during runtime to calculate the new position for the bone. To get this initial pivot point I've added the following function to the OBB class:

```
D3DXVECTOR3 OBB::SetPivot(D3DXVECTOR3 pivot)
{
    btMotionState *ms = m_pBody->getMotionState();
    if(ms == NULL)return D3DXVECTOR3(0.0f, 0.0f, 0.0f);

    D3DXMATRIX world = BT2DX_MATRIX(*ms);
    D3DXVECTOR3 newPivot;
    D3DXMatrixInverse(&world, NULL, &world);
    D3DXVec3TransformCoord(&newPivot, &pivot, &world);

    return newPivot;
}
```

Here I simply extract the world matrix from the initial motion state of the box. Next I multiply the position of the bone (the `pivot` parameter) with the inverse of the OBB's world matrix. The result (the pivot point in relation to the OBB) is returned. In runtime you should use the following function to do the exact opposite of the `SetPivot()` function:

```
D3DXVECTOR3 OBB::GetPivot(D3DXVECTOR3 pivot)
{
    btMotionState *ms = m_pBody->getMotionState();
    if(ms == NULL)return D3DXVECTOR3(0.0f, 0.0f, 0.0f);

    D3DXMATRIX world = BT2DX_MATRIX(*ms);
    D3DXVECTOR3 newPivot;
    D3DXVec3TransformCoord(&newPivot, &pivot, &world);

    return newPivot;
}
```

Here the `pivot` you supply is the one that was calculated in the initialization using the previous function, but in runtime the motion state of the box has changed and so the result from this function is the new position for the bone. Next you need to perform this exact operation, but for the orientation instead.

Getting a Bone's Orientation from an OBB

This one is perhaps a little trickier since it involves some quaternion math. But the thought process is the same. You should compare the box's current orientation with its original orientation, getting the change in orientation, and apply it to the bone's original orientation. The result is a bone whose orientation matches that of the physical representation (the box).

```
D3DXQUATERNION OBB::GetRotation(D3DXQUATERNION orgBoneRot)
{
    btMotionState *ms = m_pBody->getMotionState();
    btTransform t;
    ms->getWorldTransform(t);
    D3DXQUATERNION rot = BT2DX_QUATERNION(t.getRotation());

    D3DXQUATERNION invOrgRot;
    D3DXQuaternionInverse(&invOrgRot, &m_orgRot);
    D3DXQUATERNION diff = invOrgRot * rot;

    return orgBoneRot * diff;
}
```

Here the current orientation of the box is extracted. Then you calculate the delta quaternion—i.e., the rotation operation that rotates the box's original orientation to its current orientation. This is done by multiplying the inverse of the starting orientation with destination orientation. Then this delta quaternion is

applied to the original orientation of the bone. This operation rotates the bone from its original orientation as much as the box's orientation has changed from its original orientation. The orientation is calculated this way using the difference between the current and the original orientations of the box, since the bone and the box might have had different orientations to begin with. Finally you just need to traverse through the bone hierarchy and update the transformation matrices.

Updating the Bone Hierarchy

To update the bone hierarchy, all you need to do is traverse the hierarchy each frame, query the position and orientation of any bones connected to an OBB, and update the transformation matrix of that bone. Once a bone's matrix has been updated, it is also important to pass this change forward to any child bones that bone may have (using the `UpdateMatrices()` function). To do all this I've created the `UpdateSkeleton()` function in the Ragdoll class which should be called each frame:

```
void RAGDOLL::UpdateSkeleton(BONE* bone)
{
    if(bone == NULL)
        return;

    if(bone->m_pObb != NULL)
    {
        //Calculate new position for the bone
        D3DXMATRIX pos;
        D3DXVECTOR3 pivot = bone->m_pObb->GetPivot(bone->m_pivot);
        D3DXMatrixTranslation(&pos, pivot.x, pivot.y, pivot.z);

        //Calculate new orientation for the bone
        D3DXMATRIX rot;
        D3DXMatrixRotationQuaternion(&rot,
            &bone->m_pObb->GetRotation(bone->m_originalRot));

        //Combine to create new transformation matrix
        bone->CombinedTransformationMatrix = rot * pos;

        //Update children bones with our new transformation matrix
        if(bone->pFrameFirstChild != NULL)
            UpdateMatrices((BONE*)bone->pFrameFirstChild,
                &bone->CombinedTransformationMatrix);
    }
```

```
    //Traverse the rest of the bone hierarchy
    UpdateSkeleton((BONE*)bone->pFrameSibling);
    UpdateSkeleton((BONE*)bone->pFrameFirstChild);
}
```

Done! That about wraps it up! If you run the simulation now, the mesh will follow the physical representation of the character and seemingly fall and interact with the environment, etc. You can see a series of images in Figure 7.10 where a character falls and is updated at runtime using the ragdoll animation presented in this chapter.

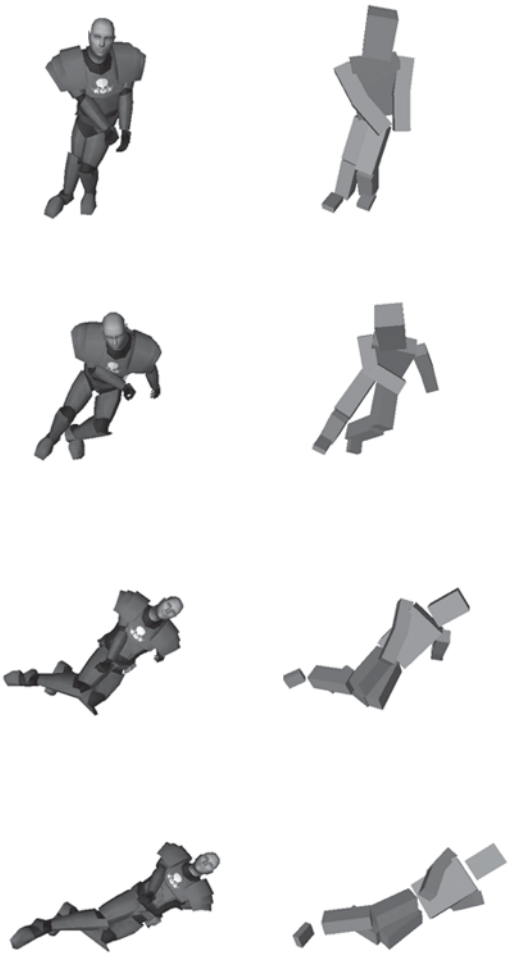


FIGURE 7.10
A sequence of images from the ragdoll simulation.

To the left in Figure 7.10 you see the character mesh and to the right is the corresponding physical representation. Check out Example 7.4 where you'll find the complete ragdoll simulation code.

Conclusions

Although this chapter is perhaps the most technically advanced in this book, it is also one of the most rewarding. By giving the character a physical representation, you can simulate the character in real-time as he falls or gets hit, etc. As cool as this may seem, the simple Ragdoll example presented in this chapter is still far from the quality you see in commercial games. However, I hope that this serves as a good primer or starting point for you to implement your own ragdoll animation. There

are several improvements to be made to this system. For example, you may want to try using shapes other than boxes. The Bullet engine's Ragdoll example used capsules, for instance. Another thing you should look into is damping and figuring out when to turn off the physical simulation of a ragdoll (both to save CPU cycles and to stop that micro twitching you can see in the examples of this chapter). Check out the Bullet library (or whatever physics engine you chose to use) documentation for more information. In the next chapter I will move away from skeletal animation and will cover morphing animation. Over the course of the next couple of chapters, I will show you how to implement organic animation, such as facial expression, talking characters, and more.

Chapter 7 Exercises

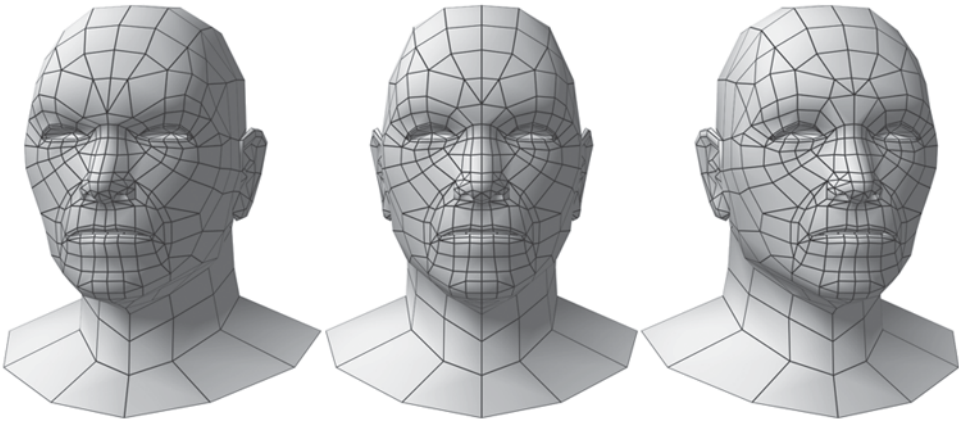
- Explore the Bullet physics library. Try to implement a bullet (a force) issuing from the mouse cursor that affects the items in any of the examples in this chapter.
- Add more geometry to the scene with which the ragdoll can collide.
- Try to mess with the gravity of the physics engine. For example, try zero-gravity simulations (which look pretty cool with the ragdoll).
- Explore other shapes, such as cylinders and capsules instead of boxes.
- Try different configurations for the joints. Make use of the hinge, twist cone, and perhaps even the Bullet engine's 6DoF constraint.
- Extend Example 7.4 to simulate more than one ragdoll.
- Implement damping of the ragdoll.

Further Reading

[VanVerth04] Van Verth, Jim, "Using Covariance Matrix for Better-Fitting Bounding Objects." *Game Programming Gems 4*, Charles River Media, 2004.

This page intentionally left blank

8 Morphing Animation



So far this book has looked only at skeletal animation. In today's games this method is used almost exclusively to animate the game characters' movements. However, it wasn't always so. For example, the first *Quake* game used characters animated using morphing animation instead. This chapter covers the basics of morphing animation (also known as per-vertex animation). This concept will also be taken one step further by combining morphing animation with skeletal animation In this chapter, you'll find:

- Introduction to morphing animation
- Morphing animation on the GPU with vertex shaders
- Combining morphing animation with skeletal animation

Basics of Morphing Animation

In skeletal animation, each vertex was linked to one or more bones with associated weights. In morphing animation, however, two or more positions are stored per vertex and are simply blended using linear interpolation (LERP). Each predefined vertex position is called a morph target. Once you have a list of morph targets, you can blend between them using weights (just as in skeletal animation), as shown in the following formula:

$$v_1 = [x_1, y_1, z_1]$$
$$v_2 = [x_2, y_2, z_2]$$

$$v = v_2 \bullet p + v_1 \bullet (1 - p)$$

The equation above describes how to create a blended vertex v between the two morph targets v_1 and v_2 (using simple LERP). This same example is also illustrated in Figure 8.1, where a new vertex position is calculated with a weight of 32%:

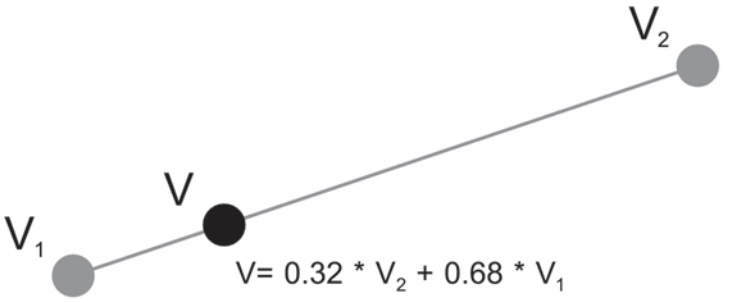
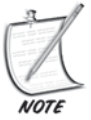


FIGURE 8.1
Blending the position of a single vertex using two morph targets and one weight.

In the same way the position of the vertex is animated, you can also animate the vertex normal, UV coordinates, etc. The following code is an excerpt from Example 8.1, where a morphed mesh is created from two target meshes. A morphed mesh is created from two target meshes by performing the blend before rendering in the CPU:

```
BYTE *target01, *target02, *face;

//Lock morph target vertex buffers
m_pTarget01->LockVertexBuffer(D3DLOCK_READONLY, (void**)&target01);
m_pTarget02->LockVertexBuffer(D3DLOCK_READONLY, (void**)&target02);

//Lock destination vertex buffer
m_pFace->LockVertexBuffer(0, (void**)&face);

//Blend the morph targets and store in the destination mesh
for(int i=0; i<m_pFace->GetNumVertices(); i++)
{
    //Get position of the two target vertices
    D3DXVECTOR3 t1 = *((D3DXVECTOR3*)target01);
    D3DXVECTOR3 t2 = *((D3DXVECTOR3*)target02);
    D3DXVECTOR3 *f = (D3DXVECTOR3*)face;

    //Perform morphing
    *f = t2 * m_blend + t1 * (1.0f - m_blend);

    //Move to next vertex
    target01 += m_pTarget01->GetNumBytesPerVertex();
    target02 += m_pTarget01->GetNumBytesPerVertex();
    face += m_pFace->GetNumBytesPerVertex();
}

//Unlock all vertex buffers
m_pTarget01->UnlockVertexBuffer();
m_pTarget02->UnlockVertexBuffer();
m_pFace->UnlockVertexBuffer();
```

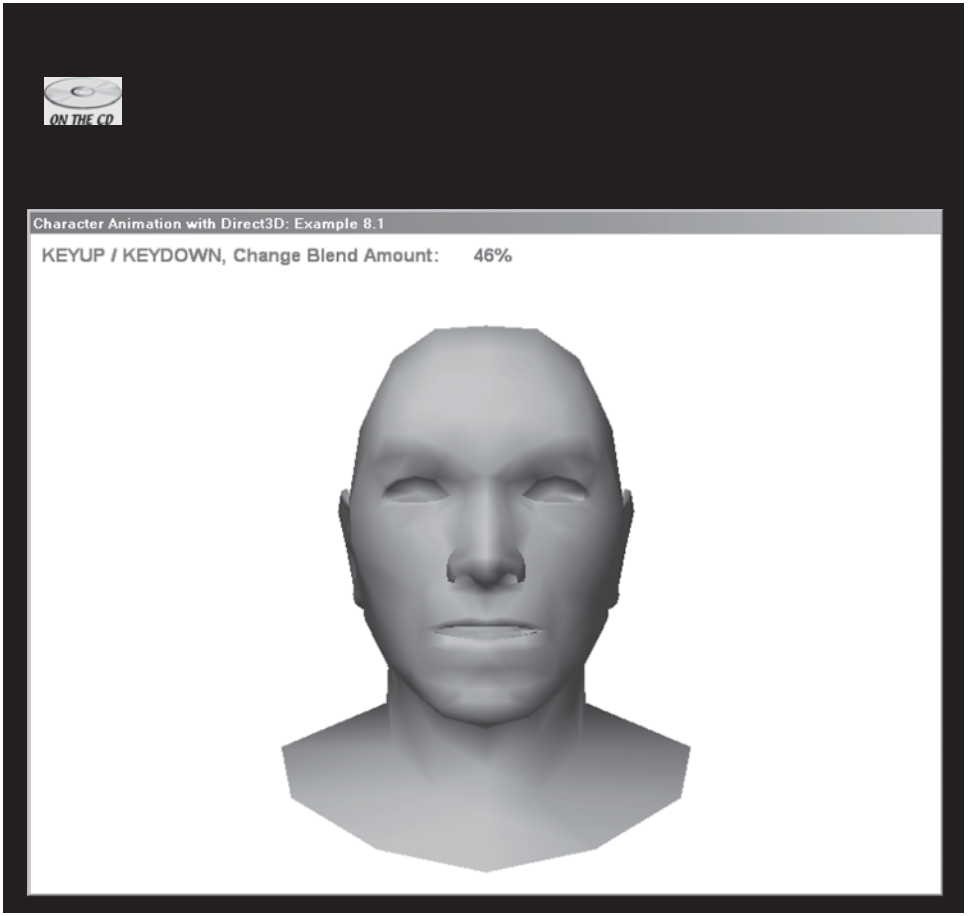


Since the position element is always the first thing in a vertex, you don't really need to know the actual vertex format of the mesh. You can simply cast the `BYTE` *pointer to a* `D3DXVECTOR3` *object to get the position element of a vertex. Then you add the number of bytes per vertex to the* `BYTE` *pointer to access the next vertex.*

However, this is a hack that you shouldn't use in a "proper" application. Instead you should cast to whatever vertex structure you are using and perform the blending on all elements: position, normal, UV coordinate, etc.

This code shows how morphing can be done easily on the CPU. First you lock all the vertex buffers (both target meshes and destination mesh). Then you iterate through all the vertices in the destination mesh and set its new vertex positions to

a blend between the two target meshes. The blend amount is defined by the `m_blend` variable. As you can see, it doesn't require that much code to get a basic example of morphing animation up and running. Have a look at Example 8.1 on the CD-ROM for the full code.



Using Multiple Morph Targets

In the previous example you learned how to blend between two morph targets. The next step is to blend between more than just two morph targets. Imagine, for example, that you have a set of mouth shapes you want to use to make the character look like he's talking. You also have a second set of morph targets controlling the blinking of the eyelids. If it were only possible to blend two

morph targets simultaneously, it would be impossible to have the character blink his eyes and talk at the same time. Luckily, of course, this is not the case.

To blend more than one morph target, you need a base mesh from which all the morph targets are compared. In the case of a character face, the base mesh would be the face without expressions and emotions, etc. Figure 8.2 shows the expressionless base mesh and the different target meshes:

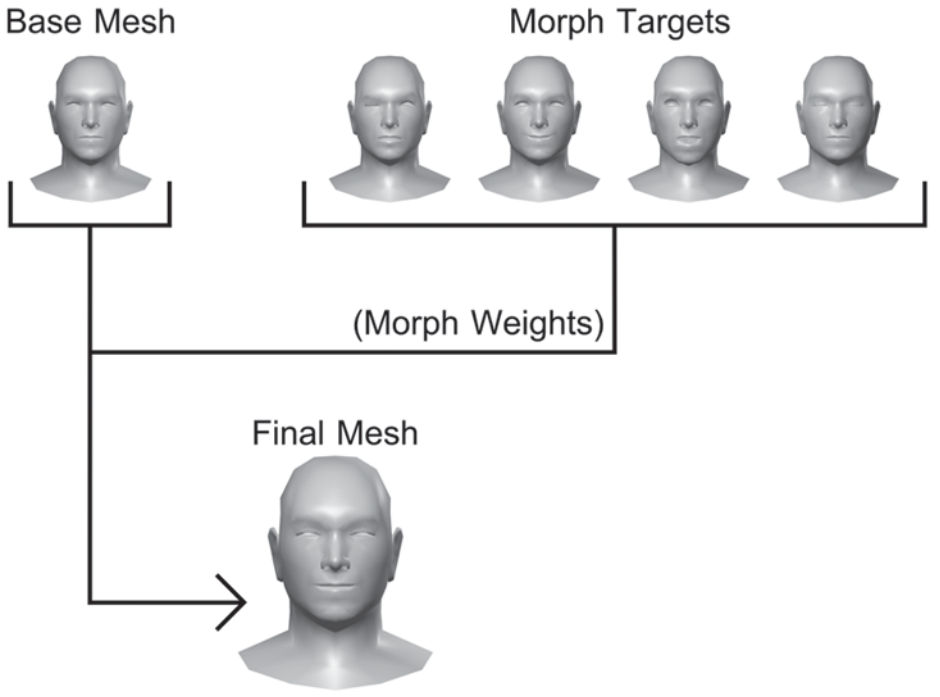


FIGURE 8.2
Blending more than two morph targets to produce the final mesh.

In mathematical terms, this could be described something like this:

$$Base = [x_b, y_b, z_b]$$
$$Morph_1 = [x_1, y_1, z_1]$$
$$Morph_2 = [x_2, y_2, z_2]$$
$$\ldots$$
$$Morph_n = [x_n, y_n, z_n]$$

$$W = [w_1, w_2, \ldots w_n]$$

$$Morph_n = [x_n, y_n, z_n]$$
$$v = base + \sum_{n}^{i=1}((Morph_i - base) * w_i)$$

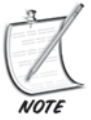
Base denotes a vertex from the base mesh. $Morph_1$ to $Morph_n$ describes the corresponding vertices in the morph targets. *W* is the collection of weights—one weight for each morph target. The final vertex *v* is then calculated by adding the weighted difference between the base and the morph targets to the original base vertex.

As shown in Figure 8.2, the morph targets are weighted before being added to the base mesh. To blend more than one morph target, you then use the following algorithm:

```
ID3DXMesh *pBaseMesh;
ID3DXMesh *pDestMesh;
vector<ID3DXMesh*> morphTargets;
vector<float> weights;


//Load base mesh, morph target meshes, and set weights
//Also create the destination mesh as a clone of the base mesh


//For each vertex in the base mesh
for(int vertex = 0; vertex < pBaseMesh->GetNumVertices(); vertex++)
{
    //Get vertex position
    D3DXVECTOR3 pos = GetVertexPosition(pBaseMesh, vertex);
```

```
    //Create a new position
    //(which will be the final blended vertex position)
    D3DXVECTOR3 newPos = pos;

    //For each active morph target (it's weight != zero)
    for(int target = 0; target < morphTargets.size(); target++)
    {
        If(weights[vertex] == 0.0f)
            continue;

        //Get morph targets vertex position
        D3DXVECTOR3 targetPos;
        targetPos = GetVertexPosition(morphTargets[target], vertex);

        //Add the weighted difference to the final position
        newPos += (targetPos – pos) * weights[vertex];
    }

    //Assign the new position to the destination mesh
    SetVertexPosition(pDestMesh, vertex, newPos);
}
```

The preceding code demonstrates how to blend multiple morph targets to produce the final morphed mesh. For each vertex of the mesh, you iterate through the morph targets; compare the vertex position of the base mesh and the target mesh. Then add the weighted difference to the final vertex position. This means that if the weight is zero or the vertex position of the target mesh is the same as the vertex position of the base mesh, then the final position of the vertex won't be changed.



Revisit Example 8.1. On the CD-ROM you will also find a third morph target (face03.x) in the resource folder of Example 8.1. Try to edit Example 8.1 to blend between all three morph targets on the CD using the pseudo-code above. Remember that "face01.x" is the base mesh to which you should compare "face02.x" and "face03.x."

Morphing Animation on the GPU

So far, all the morphing has been done in software, which, as you can imagine, can be pretty slow (especially for large meshes with many morph targets). Instead, here's how you can do the morphing animation in the GPU. The problem with

performing the morphing animation in hardware lies in the fact that the vertex shader operates on one vertex at a time. You have to upload more than one position element per vertex (one position for the base mesh, and one for each active morph target). The same applies to vertex normals (and, if necessary, UV coordinates and other vertex elements you want to blend between).

So far the format of a vertex has been defined using the old flexible vertex format (FVF). A typical vertex may then be defined something like this:

```
struct Vertex
{
    D3DXVECTOR3 position;
    D3DXVECTOR3 normal;
    D3DXVECTOR2 uv;

    static const DWORD FVF;
};

...

const DWORD Vertex::FVF = D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1;
```

A struct is declared holding position, normal, and UV information. The flexible vertex format (FVF) is also defined using the corresponding FVF codes. As you learn more advanced animation techniques, the flexible vertex format (however flexible) is not enough. Next I'll show you how to declare your own custom-made vertex formats that can be made available to a vertex shader.

Custom Vertex Formats

To create your own vertex format, you need to create an array of `D3DVERTEXELEMENT9` objects. This array tells the rendering pipeline how to interpret the stream of input data. Before the vertex data is sent to the vertex shader, you first need to interpret the long bit stream of ones and zeros, as shown in Figure 8.3.

The bits of ones and zeros come in bytes (groups of eight), which in this case are interpreted to float values (each consisting of four bytes). The float values in turn are used by the position, normal, and UV coordinates using 3, 3, and 2 floats, respectively. Finally, each vertex (in this example) consists of one position, one normal, and one UV coordinate. To help the vertex shader interpret this seemingly random data, you need to create a vertex declaration (`IDirect3DVertexDeclaration9`). To do this you first need to define an array of vertex elements (`D3DVERTEXELEMENT9`). The `D3DVERTEXELEMENT9` structure looks like this:

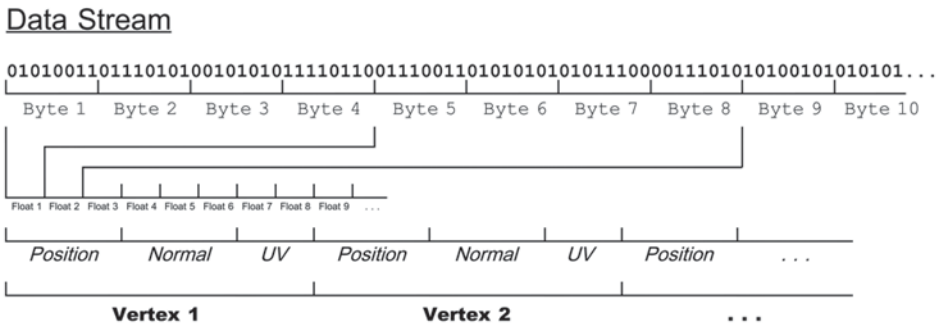


FIGURE 8.3
Interpreting the data input stream to vertex data.

```
struct D3DVERTEXELEMENT9 {
    WORD Stream;     //Stream No
    WORD Offset;     //Start of this element in bytes
    BYTE Type;       //Element type (float1 ... float4, D3DCOLOR, etc.)
    BYTE Method;     //Tesselation method
    BYTE Usage;      //Usage of element (position, normal, color, etc.)
    BYTE UsageIndex; //Suffix number used in the vertex shader
};
```

Stream

The stream number is nothing more than an index to the data stream from which this element will be interpreted. During rendering it is possible to have several active streams at the same time. With the concept of multiple streams it becomes possible to mix and match data from several sources (vertex buffers) to the final vertex shader.

Offset

This is the byte offset in the data input stream to the data you want interpreted to a specific vertex element. In a vertex buffer, for example, the position data is usually in the beginning of each element (i.e., at offset zero). The position contains three float values each containing four bytes. This means that whatever vertex data follows the position data, it will be located at offset 12.

Type

The type of the vertex element tells the vertex shader how to interpret the data stream. It also tells the vertex shader how much data to read in from the stream (since each type also has a corresponding size). The list of different vertex element

types is long—see the DirectX documentation for the entire list. However, some of the most important types are listed in Table 8.1. Knowing these are enough for you to understand the examples in this book.

TABLE 8.1 VERTEX ELEMENT TYPES

Method

The method member in the vertex element structure deals with tessellation only and is beyond the scope of this book.

Usage

More important than the type is the usage of the data. The value of this member in the `D3DVERTEXELEMENT9` structure will define how the vertex shader uses the data. The ones you need to know for this book are listed in Table 8.2 (of course there are more than these; again, see the DirectX documentation for the entire list).

TABLE 8.2 VERTEX ELEMENT USAGE

UsageIndex

The usage index tells the vertex shader which index the data belongs to. For example, in the case of multiple positions being sent to the same vertex shader, the UsageIndex is used to tell them apart. They are then referred to `POSITION0`, `POSITION1`, `POSITION2`, etc. according to the usage index.

Creating the Morph Vertex Declaration

With the `D3DVERTEXELEMENT9` structure, you can build vertex formats with the exact information you need for your specific application. For example, in the case of morphing animation you need to have several positions for each vertex and you want each of these positions to come from an individual mesh. This data from multiple sources is then merged to form the final vertex that your morphing shader can process (as shown in Figure 8.4).

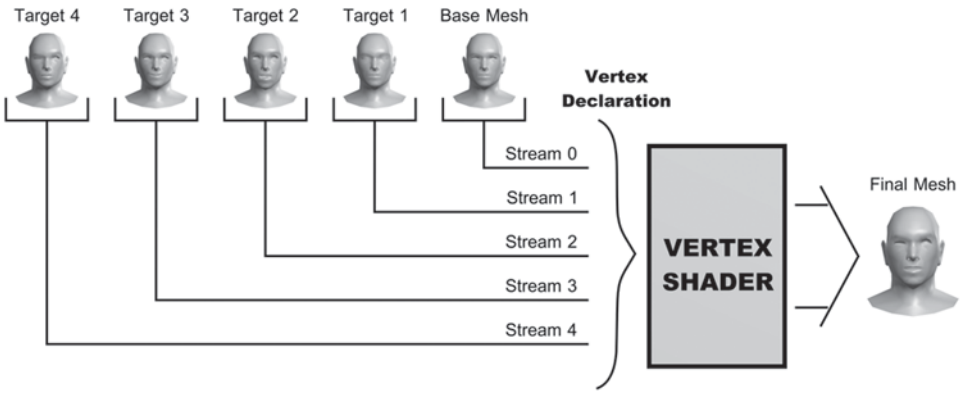


FIGURE 8.4
Creating data input streams from multiple meshes.

The following code shows the array of vertex elements that make up the morph vertex declaration:

```
//The morph vertex format
D3DVERTEXELEMENT9 morphVertexDecl[] =
{
    //Stream 0: Base 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},

        //Stream 1: 1st Morph Target
        {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},

        //Stream 2: 2nd Morph Target
        {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},

        //Stream 3: 3rd Morph Target
        {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},

        //Stream 4: 4th Morph Target
        {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()
    };
```

The values in this array are listed in the order (Stream No, Offset, Type, Method, Usage, and UsageIndex). Note that five streams are used: one for the base mesh and four for the different morph targets. In this case the stream number corresponds with the usage index (although this is not necessarily always the case). From each

stream (source vertex buffer), I am (in this vertex shader) interested only in the position, normal, and texture coordinate. Later, you'll learn to use things like blend weights and blend indices as well, but more on that later. Each vertex element array must end with the `D3DDECL_END()` macro.

Before you can use your custom vertex element array, you need to compile it into a vertex declaration. This is done in the following manner:

```
//The Custom Vertex Declaration
IDirect3DVertexDeclaration9 *m_pDecl = NULL;

//Create the vertex declaration using the D3DVERTEXELEMENT9 array
pDevice->CreateVertexDeclaration(morphVertexDecl, &m_pDecl);
```

After you have created the vertex declaration (which is using multiple streams), you can assign a vertex buffer to each stream like this:

```
//Get size per vertex in bytes
DWORD vSize = D3DXGetFVFVertexSize(m_pBaseMesh->GetFVF());

//Set base stream
IDirect3DVertexBuffer9* baseMeshBuffer = NULL;
m_pBaseMesh->GetVertexBuffer(&baseMeshBuffer);
pDevice->SetStreamSource(0, baseMeshBuffer, 0, vSize);

//Set target streams
for(int i=0; i<4; i++)
{
    IDirect3DVertexBuffer9* targetMeshBuffer = NULL;
    m_pTargets[i]->GetVertexBuffer(&targetMeshBuffer);
    pDevice->SetStreamSource(i + 1, targetMeshBuffer, 0, vSize);
}

//Set index buffer
IDirect3DIndexBuffer9* ib = NULL;
m_pBaseMesh->GetIndexBuffer(&ib);
pDevice->SetIndices(ib);
```

The base mesh is set to be stream source 0 and each of the subsequent morph targets set to be source 1 to 4. Note also that the active index buffer is set using the index buffer of the base mesh (remember that it is a requirement for morphing animation that all targets have the same polygon structure).

Next you'll need to send the weights for the four morph targets to the vertex shader. Since you need one float for each of the morph targets, you can use a D3DXVECTOR4 to hold the weights. The following code creates some random weights and uploads these to the vertex shader:

```
//Create some weights as a D3DXVECTOR4
D3DXVECTOR4 weights((rand()%1000) / 1000.0f,
                    (rand()%1000) / 1000.0f,
                    (rand()%1000) / 1000.0f,
                    (rand()%1000) / 1000.0f);

//Upload weights to shader
m_pEffect->SetVector("morphWeights", &weights);
```

The weights can then be accessed in the shader as (x, y, z, w) or (r, g, b, a). Finally, you are ready to send all the vertex data to the vertex shader and render something to the screen.

The Morphing Vertex Shader

Alright, so far you have learned how to create a custom vertex format, how to create a vertex declaration from that, and how to hook up the different input meshes as different stream sources. The input data will be available to the vertex shader according to how you set the UsageIndex in the vertex declaration. The corresponding data will be available to you using the suffix number of the data element you want to access (`POSITION0`, `POSITION1`, `POSITION2`, etc.). In the vertex shader, the following input and output structures are defined for the morphing vertex data:

```
//Vertex Input
struct VS_INPUT
{
     float4 basePos     : POSITION0;
     float3 baseNorm    : NORMAL0;
     float2 baseUV      : TEXCOORD0;

     float4 targetPos1  : POSITION1;
     float3 targetNorm1 : NORMAL1;

     float4 targetPos2  : POSITION2;
     float3 targetNorm2 : NORMAL2;
```

```
    float4 targetPos3  : POSITION3;
    float3 targetNorm3 : NORMAL3;

    float4 targetPos4  : POSITION4;
    float3 targetNorm4 : NORMAL4;
};

//Vertex Output / Pixel Shader Input
struct VS_OUTPUT
{
    float4 position : POSITION0;
    float2 tex0     : TEXCOORD0;
    float  shade    : TEXCOORD1;
};
```

The input structure matches the custom vertex declaration created in the earlier section. (Note, however, that you don't have to make use of the UV coordinates of all the target meshes since using the UV coordinates from the base mesh is enough.) The output structure simply returns one position (the result of the morphing animation), one texture coordinate, and one lighting value (from basic directional lighting).

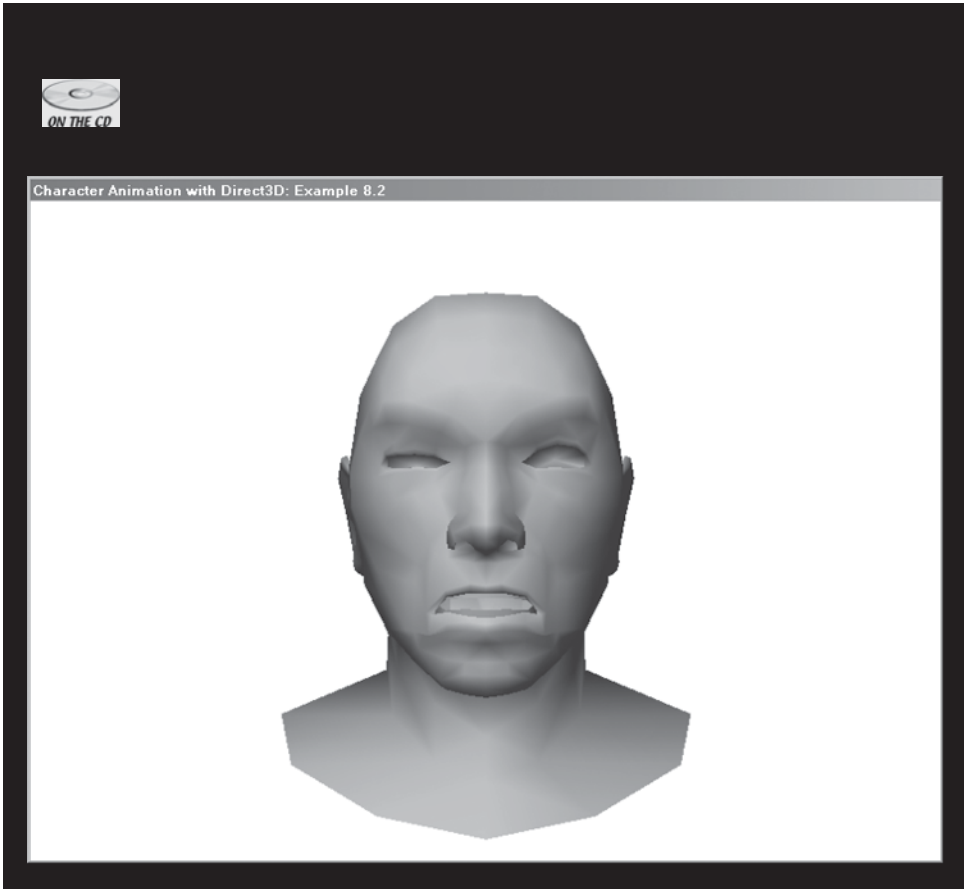
Note that it is very important that the vertex shader input structure matches the vertex declaration (data types + usage index). Otherwise, you'll most likely experience a blue-screen crash that requires a hard reboot of your computer.

Next is the actual vertex shader itself:

```
//Vertex Shader
VS_OUTPUT vs_lighting(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    float4 pos  = IN.basePos;
    float3 norm = IN.baseNorm;

    //Blend Position
    pos += (IN.targetPos1 - IN.basePos) * weights.r;
    pos += (IN.targetPos2 - IN.basePos) * weights.g;
    pos += (IN.targetPos3 - IN.basePos) * weights.b;
    pos += (IN.targetPos4 - IN.basePos) * weights.a;
```

```
    //Blend Normal
    norm += (IN.targetNorm1 - IN.baseNorm) * weights.r;
    norm += (IN.targetNorm2 - IN.baseNorm) * weights.g;
    norm += (IN.targetNorm3 - IN.baseNorm) * weights.b;
    norm += (IN.targetNorm4 - IN.baseNorm) * weights.a;

    //Getting the position of the vertex in the world
    float4 posWorld = mul(pos, matW);
    float4 normal = normalize(mul(norm, matW));

    //Transforming to screen space
    OUT.position = mul(posWorld, matVP);

    OUT.shade = max(dot(normal, normalize(lightPos - posWorld)), 0.2f);

    OUT.tex0 = IN.baseUV;

    return OUT;
}
```

You can see here how the four morph targets are compared to the base mesh, and the difference is weighted and added to the final position of the vertex. In the exact same way, the normal is weighted and added. After the final position and normal have been calculated, the shader is fairly elementary, transforming the position to screen space, calculating a per-vertex lighting value, and copying the texture coordinate for the pixel shader. You can find the entire effect (.fx) file on the accompanying CD-ROM, along with the pixel shader and technique.



Combining Skeletal and Morphing Animation

In this section you'll learn how to combine skeletal animation with morphing animation. There are many cases when you might want to use this technique. As a basic rule you should use it when an object is changing shape in ways other than bending limbs around joints (transformations of the skeletal kind). It can also be used when trying to achieve the desired result using only skeletal animation would cause you to add an unreasonable amount of bones. In this section, the two morph targets shown in Figure 8.5 will be used to create an example of a skinned and morphing character.

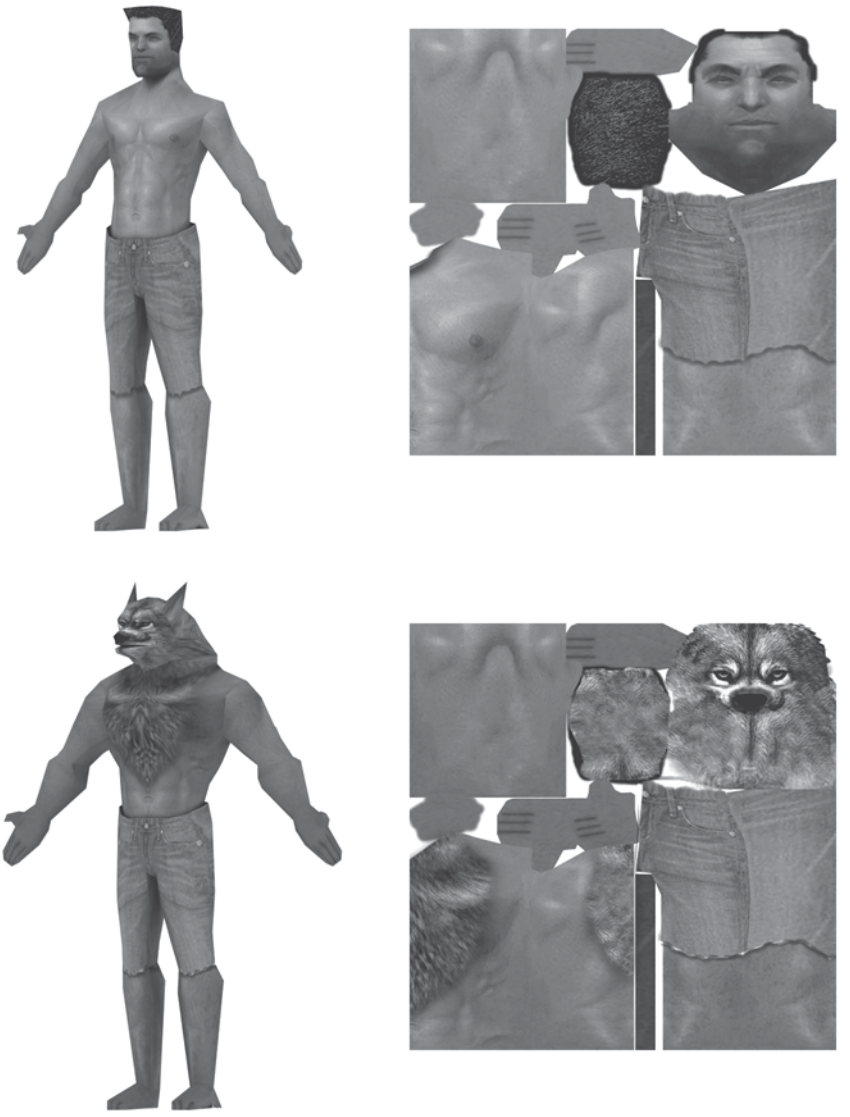


FIGURE 8.5
The two morph targets used for the werewolf example. (Both models have 467 vertices and 930 polygons.)

In Chapter 3 I covered how to do the skeletal animation on the GPU. First the mesh was converted to an Index Blended Mesh, and then a vertex shader was used, to which the matrix palette (bone matrices) was uploaded. To combine a skinned mesh with a morphing animation, you simply apply what you have learned in this

chapter with what you learned back in Chapter 3. In the vertex shader the morphing animation is first performed like it was done in Example 8.2. Then the morphed vertex position is used with the skeletal index blended transformations. Figure 8.6 shows an overview of how this is done:

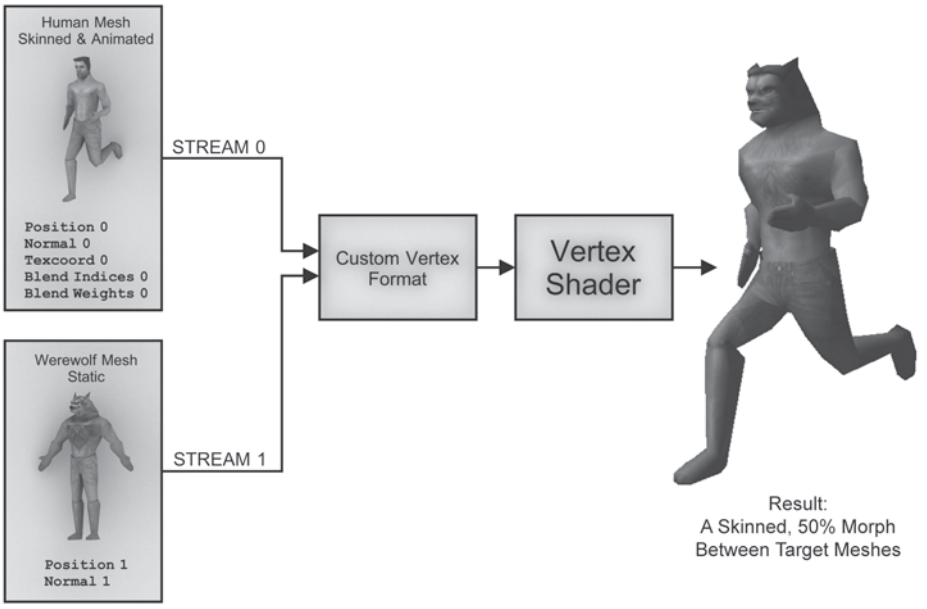


FIGURE 8.6
Combining skeletal animation with morphing animation.

As previously stated, whenever you do a morphing animation it is important that all target meshes have the same amount of vertices and that the polygons are configured the same way. The position, normal, and texture coordinates of a vertex can change, of course, but other mesh attributes should remain the same across all target meshes.

Skeletal/Morphing Vertex Format

You already know from previous sections how to set up new custom vertex formats. In the last example there were several input streams, one for each morph target. In this example, however, you will have only two morph targets: the human mesh and the werewolf mesh. The human mesh has skinning information as well as a running animation. Here's the custom vertex format that will be used (note the blend indices and the blend weights; also, you only get the texture coordinates from the human mesh since they are the same for both meshes):

```
D3DVERTEXELEMENT9 morphVertexDecl[] =
{
    //Stream 0: Human Skinned Mesh
    {0,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_POSITION, 0},
    {0, 12, D3DDECLTYPE_FLOAT1, D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_BLENDWEIGHT, 0},
    {0, 16, D3DDECLTYPE_UBYTE4, D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_BLENDINDICES, 0},
    {0, 20, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_NORMAL,   0},
    {0, 32, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_TEXCOORD, 0},

    //Stream 1: Werewolf Morph Target
    {1,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_POSITION, 1},
    {1, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_NORMAL,   1},

    D3DDECL_END()
};
```

The next trick to perform is to set up the different streams. In this example the two meshes are stored in the same .x file. The meshes are loaded using the same code used to load the skinned meshes back in Chapter 3. Hopefully you remember how the bone hierarchy was created from the .x file and how it was traversed to render the skinned mesh. Now there are two meshes in the bone hierarchy: the skinned human mesh and the static werewolf mesh. Here's the code that finds the static werewolf mesh in the hierarchy and sets it as stream source 1:

```
//Set werewolf stream
//Find bone named "werewolf" located in the m_pRootBone hierarchy
D3DXFRAME* wolfBone = D3DXFrameFind(m_pRootBone, "werewolf");

if(wolfBone != NULL)
{
    //If bone contains a mesh container then this is the werewolf mesh
    if(wolfBone->pMeshContainer != NULL)
    {
        //Get werewolf vertex buffer
        ID3DXMesh* wolfmesh;
        wolfMesh = wolfBone->pMeshContainer->MeshData.pMesh;
```

```
        DWORD vSize = D3DXGetFVFVertexSize(wolfmesh->GetFVF());
        IDirect3DVertexBuffer9* wolfMeshBuffer = NULL;
        wolfmesh->GetVertexBuffer(&wolfMeshBuffer);

        //Set vertex buffer as stream source 1
        pDevice->SetStreamSource(1, wolfMeshBuffer, 0, vSize);
    }
}
```

Now all you need to do is search though the hierarchy and find the mesh that has skinning information (this will be the skinned human mesh). Then set this mesh to be stream source 0 as well as the index buffer and render the mesh using the `DrawIndexedPrimitive()` function:

```
void RenderHuman(BONE *bone)
{
    //If there is a mesh to render...
    if(bone->pMeshContainer != NULL)
    {
        BONEMESH *boneMesh = (BONEMESH*)bone->pMeshContainer;

        if (boneMesh->pSkinInfo != NULL)
        {
            // Set up bone transforms and the matrix palette here
            ...
            //Get size per vertex in bytes
            DWORD vSize = D3DXGetFVFVertexSize(
                                boneMesh->MeshData.pMesh->GetFVF());

            //Set base stream (human)
            IDirect3DVertexBuffer9* baseMeshBuffer = NULL;
            boneMesh->MeshData.pMesh->GetVertexBuffer(
                                            &baseMeshBuffer);
            pDevice->SetStreamSource(0, baseMeshBuffer, 0, vSize);

            //Set index buffer
            IDirect3DIndexBuffer9* ib = NULL;
            boneMesh->MeshData.pMesh->GetIndexBuffer(&ib);
            pDevice->SetIndices(ib);

            //Start shader
            D3DXHANDLE hTech;
            hTech = pEffect->GetTechniqueByName("Skinning");
```

```
            pEffect->SetTechnique(hTech);
            pEffect->Begin(NULL, NULL);
            pEffect->BeginPass(0);

            //Draw mesh
            pDevice->DrawIndexedPrimitive(
                      D3DPT_TRIANGLELIST, 0, 0,
                      boneMesh->MeshData.pMesh->GetNumVertices(), 0,
                      boneMesh->MeshData.pMesh->GetNumFaces());
            pEffect->EndPass();
            pEffect->End();
        }
    }

    if(bone->pFrameSibling != NULL)
        RenderHuman((BONE*)bone->pFrameSibling);

    if(bone->pFrameFirstChild != NULL)
        RenderHuman((BONE*)bone->pFrameFirstChild);
}
```

That about covers all you need to do on the application side to set up skinned morphing animation. The next thing to look at is the vertex shader that will read all this data in and make the final calculations before presenting the result onto the screen.

Skeletal/Morphing Vertex Shader

This vertex shader is basically just the offspring of the marriage between the skinned vertex shader in Chapter 3 and the morphing shader from this chapter. The input structure matches the custom vertex format created in the previous section:

```
//Morph Weight
float shapeShift;

//Vertex Input
struct VS_INPUT_SKIN
{
     float4 position    : POSITION0;
     float3 normal      : NORMAL0;
     float2 tex0        : TEXCOORD0;
     float4 weights     : BLENDWEIGHT0;
     int4   boneIndices : BLENDINDICES0;
```

```
     float4 position2 : POSITION1;
     float3 normal2   : NORMAL1;
};

//Vertex Output / Pixel Shader Input
struct VS_OUTPUT
{
     float4 position : POSITION0;
     float2 tex0     : TEXCOORD0;
     float  shade    : TEXCOORD1;
};

VS_OUTPUT vs_SkinningAndMorphing(VS_INPUT_SKIN IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    //Perform the morphing
    float4 position = IN.position +
                      (IN.position2 - IN.position) * shapeShift;

    //Perform the skinning (just as in Chapter 3)
    float4 p = float4(0.0f, 0.0f, 0.0f, 1.0f);
    float3 norm = float3(0.0f, 0.0f, 0.0f);
    float lastWeight = 0.0f;
    int n = NumVertInfluences-1;

    IN.normal = normalize(IN.normal);
    for(int i = 0; i < n; ++i)
    {
        lastWeight += IN.weights[i];

        p += IN.weights[i] *
             mul(position, FinalTransforms[IN.boneIndices[i]]);
        norm += IN.weights[i] *
                mul(IN.normal, FinalTransforms[IN.boneIndices[i]]);
    }
    lastWeight = 1.0f - lastWeight;

    p += lastWeight *
         mul(position, FinalTransforms[IN.boneIndices[n]]);
    norm += lastWeight *
            mul(IN.normal, FinalTransforms[IN.boneIndices[n]]);
```

```
    p.w = 1.0f;
    float4 posWorld = mul(p, matW);
    OUT.position = mul(posWorld, matVP);
    OUT.tex0 = IN.tex0;

    //Calculate Lighting
    norm = normalize(norm);
    norm = mul(norm, matW);
    OUT.shade = max(dot(norm, normalize(lightPos - posWorld)), 0.2f);

    return OUT;
}

//Pixel Shader
float4 ps_lighting(VS_OUTPUT IN) : COLOR0
{
    //Sample human texture
    float4 colorHuman = tex2D(HumanSampler, IN.tex0);

    //Sample wolf texture
    float4 colorWolf = tex2D(WolfSampler, IN.tex0);

    //Blend the result based on the shapeShift variable
    float4 c = (colorHuman*(1.0f-shapeShift) + colorWolf*shapeShift);
    return c * IN.shade;
}
```

Here's the pixel shader that blends between the two textures (human/werewolf) as well. Note that it is based on the same `shapeShift` variable used to blend the two meshes. You can find the full shader code on the CD-ROM in Example 8.3.

Conclusions

This chapter covered the basics of morphing animation, starting with morphing done in software and then progressing to advanced morphing done on the GPU with several morph targets, etc. There was also a brief glimpse of combining skeletal animation with morphing animation. The next chapter focuses on how to make a proper face for the character with eyes looking around, emotions showing, eye lids blinking, and much more.

Chapter 8 Exercises

- Create a simple object in any 3D modeling software. Make a clone of the object and change the UV coordinates of this clone. Implement morphing of the UV coordinates as explained in this chapter.
- This technique can be used for more than just characters. Experiment with other biological shapes (plant life, blobs, fungi, etc). Create, for example, a tree swaying in the wind.
- Try to preprocess the morph targets so that they contain the difference between the original morph target and the base mesh. Update the vertex shader accordingly. This way you can save some GPU cycles during the runtime morphing.

9 Facial Animation

This chapter expands upon what you learned in the previous chapter. Building on simple morphing animation, you can create complex facial animations quite easily. The most problematic thing is always to create a good "infrastructure," making loading and setting of the stream sources and so on as simple as possible. I'll also cover how to add eyes to the character and make him look at a specific point. To top it all off, I'll conclude this chapter by showing you how to create a facial factory system much like those seen in games like *Oblivion*™ or *Fallout 3*™. With a system like this you can let the user create a custom face for his/her character or even use it to generate large crowds with unique faces. In this chapter, you'll find the following:

- Adding eyes to the character
- Loading multiple facial morph targets from a single .x file
- The `Face` and the `FaceController` classes
- A face factory system for generating faces in runtime

Facial Animation Overview

In the creation of believable computer game characters, it is becoming increasingly important that characters convey their emotions accurately through body language and facial expressions. Giving the player subtle information like NPC facial expressions can greatly increase the immersion of a particular game. Take Alyx in *Half Life 2*™, for example—her face conveys worry, fear, happiness, and many other emotions.

You have already learned in the previous chapter all you need to know to technically implement facial animation. All it comes down to is blending multiple meshes together. However, there are several other things you need to think about before you start blending those meshes. In real human beings, facial expression is controlled by all those muscles just under the skin called the mimetic muscles. There are just over 50 of these muscles, and with them the whole range of human emotion can be displayed. Digital animation movies may go so far as to model the muscles in a character's face, but in computer games that level of realism still lies in the future. So for interactive applications like computer games, we are (for now) left with morphing animation as the best approach to facial animation. However, no matter which technique you choose, it is important that you understand the underlying principles of facial expressions.

Facial Expressions

Facial expressions are a form of non-verbal communication that we primates excel in. They can convey information about a person's emotion and state of mind. Facial expressions can be used to emphasize or even negate a verbal statement from a person. Check out Figure 9.1 for an example.

It is also important to realize that things like the orientation of the head and where the character is looking plays a big part in how you would interpret a facial expression. For example, if a character avoids looking you in the eye when talking to you it could be taken as a sign that he or she is not telling you the truth.

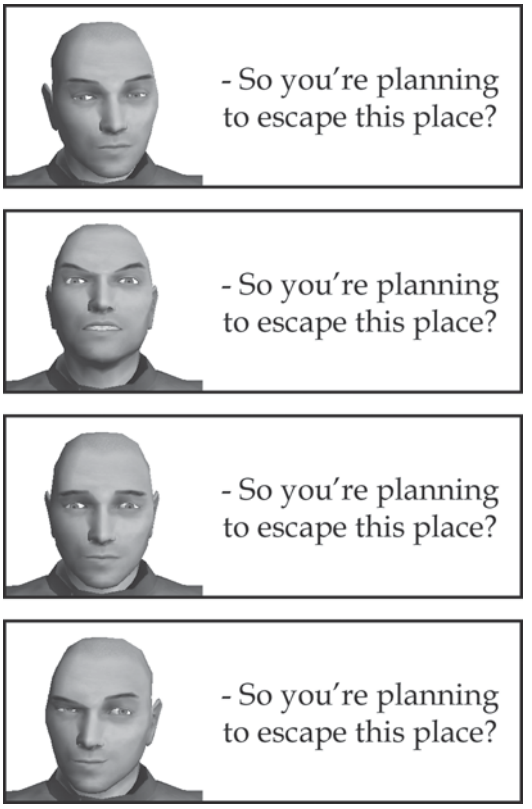


FIGURE 9.1
The same verbal message combined with different emotions can produce different meanings.

This chapter will focus on the most obvious types of facial motion:

- Speech
- Emotion
- Eye movements

I will only briefly touch on the subject of character speech in this chapter since the entire next chapter deals with this topic in more depth. In this chapter you'll learn one approach to setting up the infrastructure needed for facial animation.

The Eye of the Beholder

So far throughout this book the character has had hollows where his eyes are supposed to be. This will now be corrected. To do this you simply take a spherical mesh (eyeball mesh) and apply a texture to stick it in the two hollows of the face. Next you'll need the eyes to focus on the same location, thus giving the impression that the character is looking at something. This simple look-at behavior is shown in Figure 9.2.

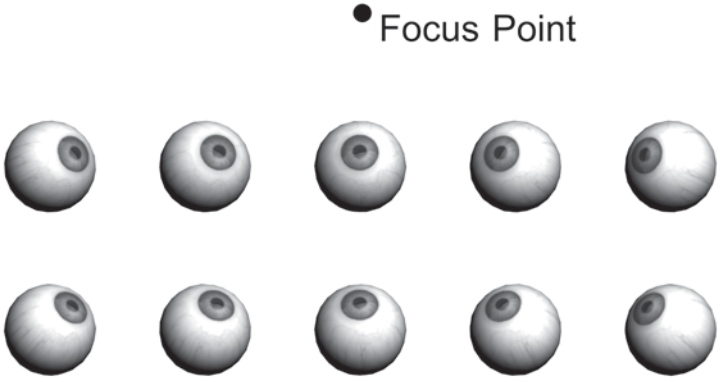


FIGURE 9.2
A somewhat freaky image showing several eyeballs focusing on the same focus point.

To implement this simple behavior, I've created the `Eye` class as follows:

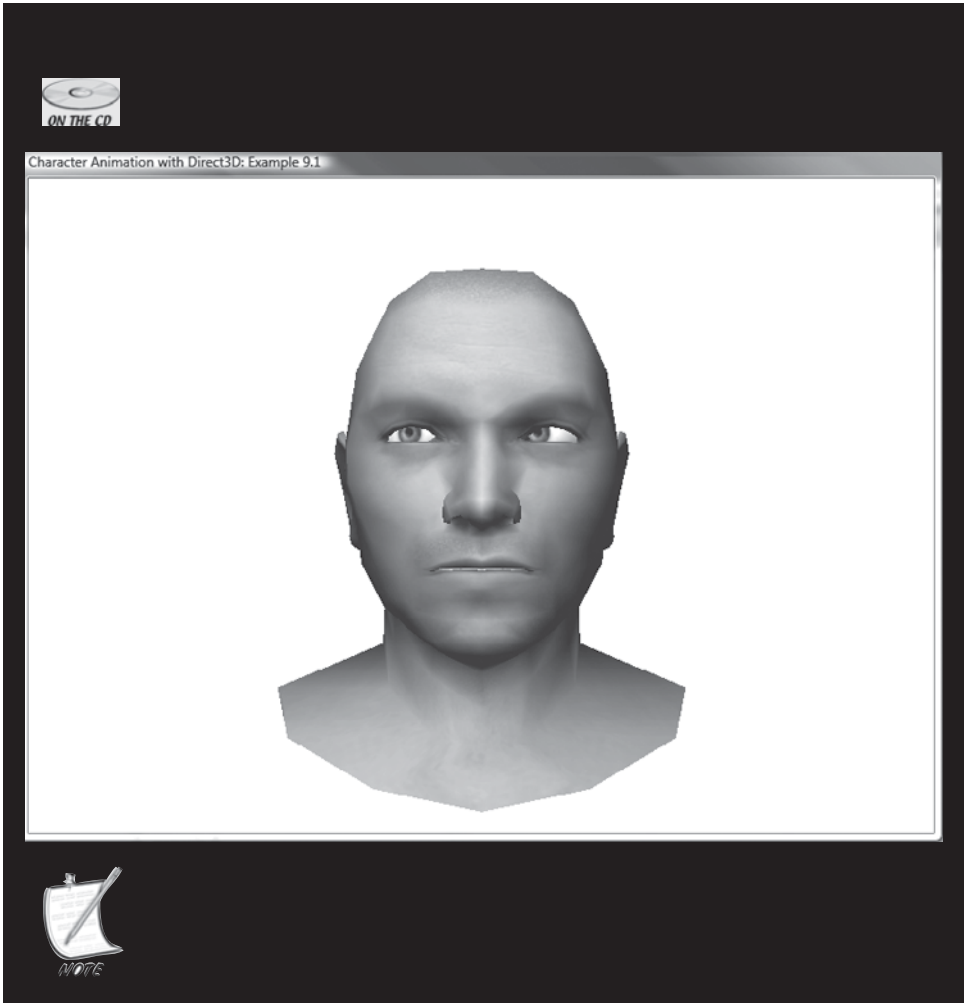
```
class Eye
{
    public:
        Eye();
        void Init(D3DXVECTOR3 position);
        void Render(ID3DXEffect *pEffect);
        void LookAt(D3DXVECTOR3 focus);
```

```
    private:
        D3DXVECTOR3 m_position;
        D3DXVECTOR3 m_lookAt;
        D3DXMATRIX m_rotation;
};
```

The `Init()` function sets the eye at a certain position; the `Render()` function renders the eye using the provided effect. The most interesting function is of course the `LookAt()` function, which calculates the eye's m_rotation matrix. The rotation matrix is created by calculating the angle difference between the position of the eye and the focus point. For this you can use the `atan2()` function, which takes a delta x and a delta y value and calculates the angle from these:

```
void Eye::LookAt(D3DXVECTOR3 focus)
{
    //Rotation around the Y axis
    float rotY = atan2(m_position.x - focus.x,
                       m_position.z - focus.z) * 0.8f;

    //Rotation around the Z axis
    float rotZ = atan2(m_position.y - focus.y,
                       m_position.z - focus.z) * 0.5f;

    D3DXMatrixRotationYawPitchRoll(&m_rotation, rotY, rotZ, 0.0f);
}
```

The `Eye` class is implemented in full in Example 9.1 on the CD-ROM.



The Face Class

It is now time to put all you've done so far into a single class: the `Face` class. It will contain all the render targets, eyes, and vertex declarations as well as the morphing shader used to render it. Later this class will be extended to cooperate with the skinned mesh and ragdoll characters created in the earlier chapters. For now, however, let us just consider a single face!

You will find when you try to send several render targets to a morphing vertex shader that you eventually run out of either instruction slots or input registers (depending on which vertex shader version your graphic card supports). Also,

blending a large amount of render targets in real-time would take its toll on the frame rate, especially if you blend faces with large amounts of vertices. In this book I'll stick with four render targets since that is about as much as can be crammed into the pipeline when using vertex shaders of version 2.0.

You can have only four active render targets at a time per face (without diving into more advanced facial animation techniques). Note, however, that I'm speaking about active render targets. You will need to have plenty more render targets in total to pull off believable facial animation. Here's a list of some render targets you would do well to create whenever creating a new face for a game:

- Base mesh
- Blink mesh
- Emotion meshes (smile, frown, fear, etc.)
- Speech meshes (i.e., mouth shapes for different sounds; more on this in the next chapter)

I won't cover the process of actually creating the meshes themselves. There are plenty of books in the market for each of the major 3D modeling programs available. I stress again though that for morphing animation to work, the vertex buffer of each render target needs to contain the same amount of vertices and the index buffer needs to be exactly the same. The easiest way to achieve this is to first create the base mesh and then create clones of the base mesh and alter them to produce the different render targets.

Performing operations on a render target after copying it from the base mesh, such as adding or deleting faces or vertices, flipping faces, etc., will result in an invalid render target.

I'll assume now that you have a base mesh, blink mesh, emotion meshes, and speech meshes created in your 3D modeling program. There are two approaches to how you can store these meshes and make them available to your game. Either you store each mesh in individual .x files, or you store them all in the same file. Although the simpler approach (to implement) would be to load the different render targets from individual files using the `D3DXLoadMeshFromX()` function, we will attempt the trickier approach. You'll see in the end that the extra effort of writing code to import the render targets from a single file per face will save you a lot of hassle and time exporting the many faces.

Loading Multiple Targets from One .X File

You may be thinking that this topic has already been covered. Well, that's true. You already know how to load multiple meshes from a single .x file. This was done when you learned how to create a skinned character. The only difference now is that you don't want all these meshes to be contained in a `D3DXFRAME` hierarchy like in the case of a skinned character. If you loaded an .x file containing several meshes using the `D3DXLoadMeshFromX()` function, it would collapse all the separate meshes into one single `ID3DXMesh` object for you. Since this is not what is wanted, another way must be found. As when a skinned mesh was loaded, you implemented your own custom version of the `ID3DXAllocateHierarchy` interface. This time around I will only use the name of the D3DXFRAME to identify which mesh is which. Here follows the full listing of the `FaceHierarchyLoader` class (implementing the `ID3DX-AllocateHierarchy` interface) used to load multiple meshes from a single .x file:

```
class FaceHierarchyLoader : public ID3DXAllocateHierarchy
{
    public:
    STDMETHOD(CreateFrame)(THIS_ LPCSTR Name,
                           LPD3DXFRAME *ppNewFrame);

    STDMETHOD(CreateMeshContainer)(THIS_ LPCTSTR Name,
                    CONST D3DXMESHDATA * pMeshData,
                    CONST D3DXMATERIAL * pMaterials,
                    CONST D3DXEFFECTINSTANCE * pEffectInstances,
                    DWORD NumMaterials, CONST DWORD * pAdjacency,
                    LPD3DXSKININFO pSkinInfo,
                    LPD3DXMESHCONTAINER * ppNewMeshContainer);

    STDMETHOD(DestroyFrame)(THIS_ LPD3DXFRAME pFrameToFree);

    STDMETHOD(DestroyMeshContainer)(
                      THIS_ LPD3DXMESHCONTAINER pMeshContainerBase);
};

HRESULT FaceHierarchyLoader::CreateFrame(LPCSTR Name,
                                         LPD3DXFRAME *ppNewFrame)
{
    D3DXFRAME *newBone = new D3DXFRAME;
    memset(newBone, 0, sizeof(D3DXFRAME));

    //Copy name (used to tell one mesh from another)
    if(Name != NULL)
    {
```

```
        newBone->Name = new char[strlen(Name)+1];
        strcpy(newBone->Name, Name);
    }

    //Return the new bone...
    *ppNewFrame = newBone;

    return S_OK;
}

HRESULT FaceHierarchyLoader::CreateMeshContainer(LPCSTR Name,
                            CONST D3DXMESHDATA *pMeshData,
                            CONST D3DXMATERIAL *pMaterials,
                            CONST D3DXEFFECTINSTANCE *pEffectInstances,
                            DWORD NumMaterials,
                            CONST DWORD *pAdjacency,
                            LPD3DXSKININFO pSkinInfo,
                            LPD3DXMESHCONTAINER *ppNewMeshContainer)
{
    //Add reference so that the mesh isn't de-allocated
    pMeshData->pMesh->AddRef();

    //Return pointer to mesh casted to a D3DXMESHCONTAINER pointer
    *ppNewMeshContainer = (D3DXMESHCONTAINER*)pMeshData->pMesh;

    return S_OK;
}
```

I create a frame pretty much the same way as I did in the earlier examples when a skinned mesh was loaded. The only difference is that the basic `D3DXFRAME` structure is used and the initialization of the transformation matrix, etc. is ignored. In the `CreateMeshContainer()` function, input such as materials, skin information, and so on is completely ignored. Instead the function just returns a pointer to the loaded mesh data. You now have a minimum `D3DXFRAME` hierarchy containing only the frame name and the meshes without any skin information and textures, etc. The next step is to traverse this structure and extract the different meshes and store them in the `Face` class instead.

Extracting Meshes from a D3DXFRAME Hierarchy

Hopefully you remember from Chapter 3 that a hierarchy is built up using the two pointers `pFrameSibling` and `pFrameFirstChild` stored in a `D3DXFRAME` object. The `D3DXFRAME` structure also stores a pointer to a `D3DXMESHCONTAINER` object, which can

contain a mesh. So all you need to do now in order to extract a certain mesh is to traverse the structure and find the `D3DXFRAME` that has the name of the mesh you are looking for. The following function does just that by searching the hierarchy recursively and returning a mesh with a certain name (if there is one to be found):

```
ID3DXMesh* ExtractMesh(D3DXFRAME *frame, string name)
{
    //Does this frame have a mesh?
    if(frame->pMeshContainer != NULL)
    {
        ID3DXMesh *mesh = (ID3DXMesh*)frame->pMeshContainer;

        //This is the mesh we are searching for!
        if(frame->Name != NULL &&
           strcmp(frame->Name, name.c_str()) == 0)
        {
            mesh->AddRef();
            return mesh;
        }
    }

    //Otherwise check siblings and children
    ID3DXMesh *result = NULL;

    if(frame->pFrameSibling != NULL)
    {
        result = ExtractMesh (frame->pFrameSibling, name);
    }

    if(result == NULL && frame->pFrameFirstChild != NULL)
    {
        result = ExtractMesh (frame->pFrameFirstChild, name);
    }

    return result;
}
```

Implementing the Face Class

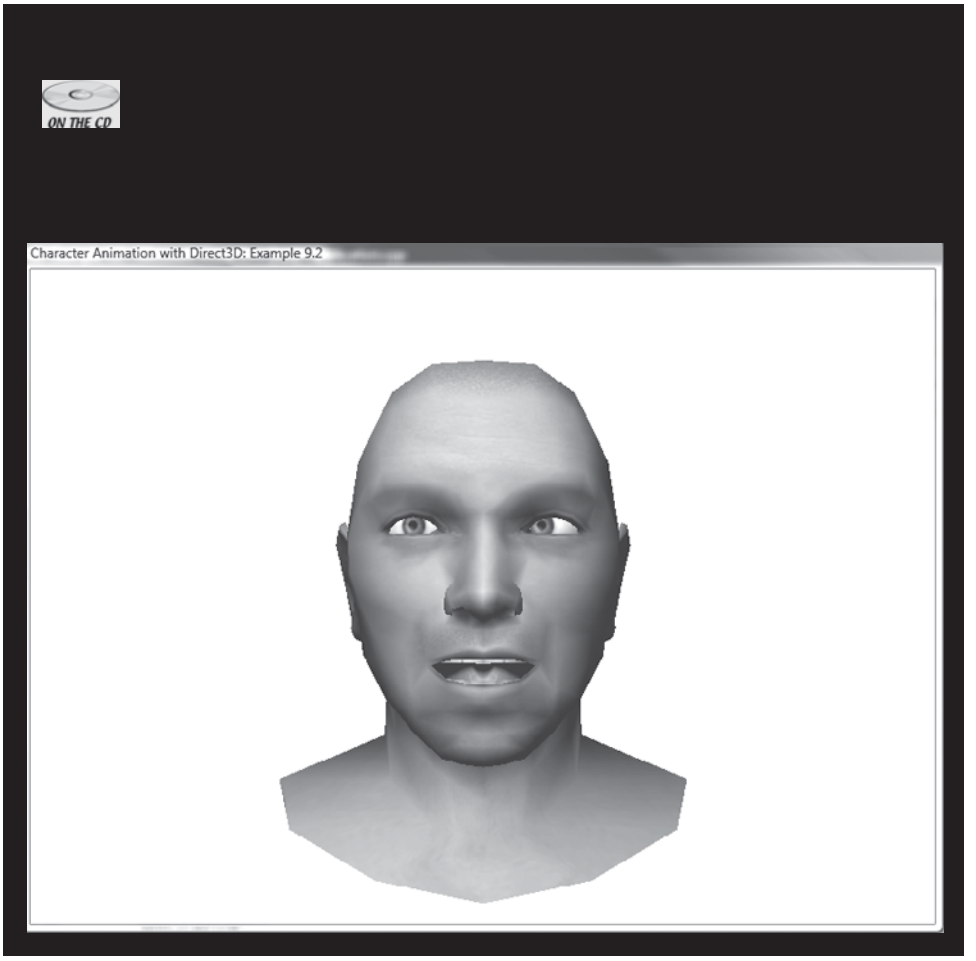
You now know all you need in order to move on to the first implementation of the `Face` class. I will build on this class over the next couple of chapters until we finally have a complete character at the end of the book.

```
class Face
{
    public:
        Face(string filename);
        ~Face();
        void TempUpdate();
        void Render();
        void ExtractMeshes(D3DXFRAME *frame);

    public:
        ID3DXMesh *m_pBaseMesh;
        ID3DXMesh *m_pBlinkMesh;
        vector<ID3DXMesh*> m_emotionMeshes;
        vector<ID3DXMesh*> m_speechMeshes;

        IDirect3DVertexDeclaration9 *m_pFaceVertexDecl;

        IDirect3DTexture9 *m_pFaceTexture;

        ID3DXEffect *m_pEffect;

        D3DXVECTOR4 m_morphWeights;

        Eye m_eyes[2];
};
```

Table 9.1 describes the members of the `Face` class.

TABLE 9.1 FACE MEMBERS

First a `Face` class is created by loading multiple meshes from a single .x file as covered earlier in this chapter. Then the `ExtractMeshes()` function is called to assign the correct meshes to the `m_pBaseMesh`, `m_pBlinkMesh`, `m_emotionMeshes`, and `m_speechMeshes`. The `ExtractMeshes()` function is an extension of the general `ExtractMesh()` function covered earlier. The `ExtractMeshes()` function sorts each of the meshes of a face hierarchy (remember that there can be more than one mesh with the same name). After that, the face is rendered as was done in Chapter 8 where morphing animation was covered.



At the moment there is some logic in the `Face` class updating the morph targets and the morph weights. However, the goal is to make the `Face` class a simple resource container and put the facial logic in another class. Therefore, let's move on

and look at implementing a class that will take care of updating a face and render multiple instances of the same face.

The Face Controller Structure

So far I have only cared about rendering one face. However, many times you want to use the same face and render several characters with it (although with different expressions, etc.). Therefore, think of the `Face` class only as a resource container containing the necessary meshes (render targets). The information of how the face is supposed to be rendered I will stick into a new class, which I'll call the `FaceController` class. This class will point to a `Face` class of which the `FaceController` class will set the active render targets and their weights before rendering the face. The `FaceController` class will also contain the eyes and control the rendering and updating of these as well.

Animation Channels

Remember that you only have a limited number of render targets available when you do your morphing animation using a vertex shader. In the case of VS 2.0 (which I use in the examples), you can push one base mesh around four render targets. I will refer to each of these possible render targets as an *animation channel*. There are a few different ways you can choose to use these animation channels. I have chosen to use one channel for the eye blinking, one for emotion render targets, and the final two channels for speech. This is, however, only one of many methods, and you should pick the configuration that best suits your needs. Figure 9.3 shows how I intend to use the four animation channels throughout this book.

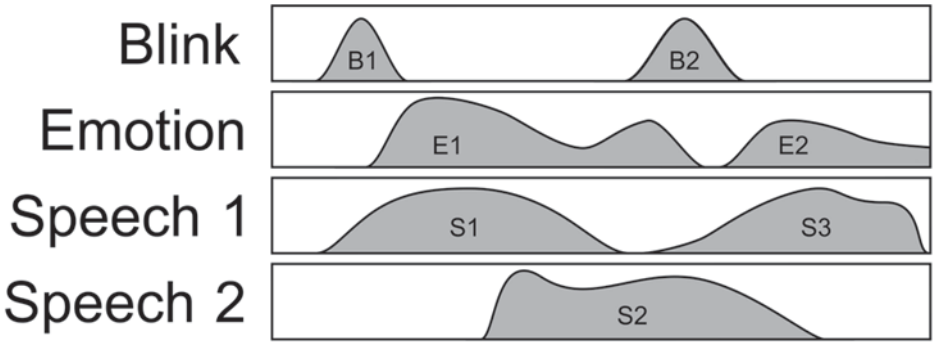


FIGURE 9.3
The use of the four animation channels.

I suppose Figure 9.3 requires some additional explanation.

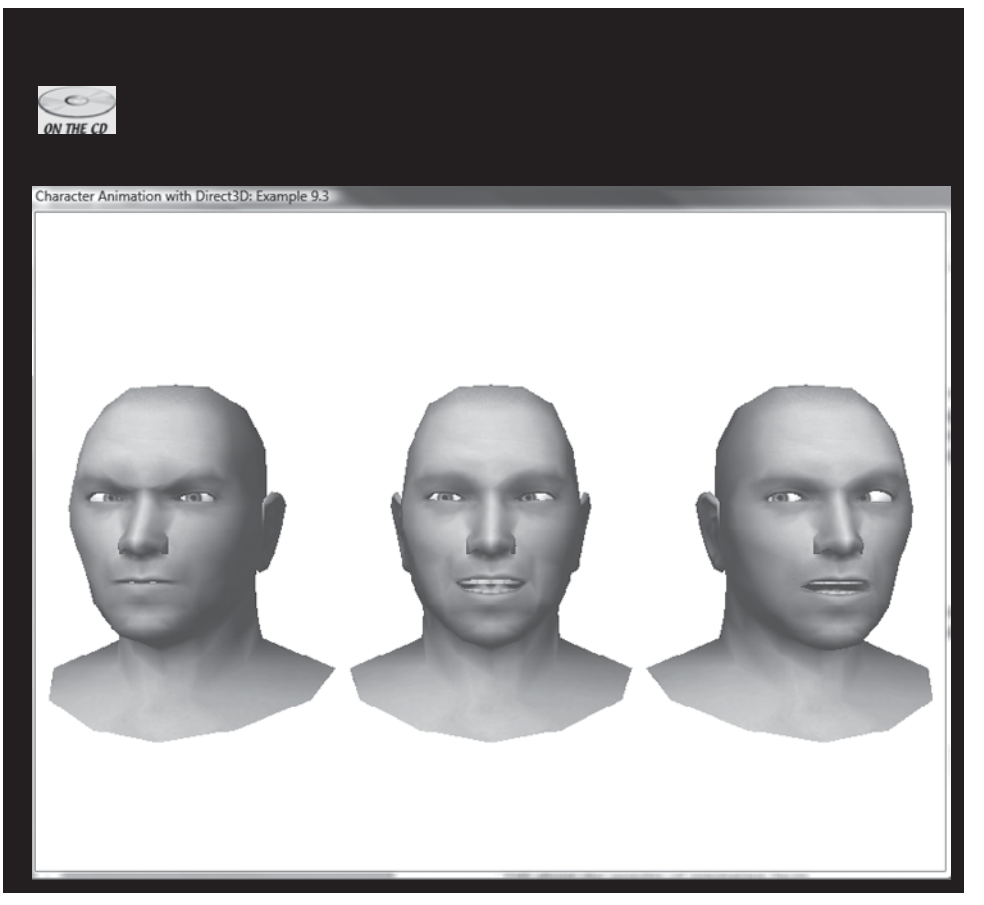
- **Blink Channel:** This channel uses only one render target (the face with the eyelids closed). In Figure 9.3 there are two times that the eyes blink (B1 and B2). You see the weights go up and down in a nice bell-shaped curve when this happens.
- **Emotion Channel:** The emotion channel can be used by many different render targets but only one at a time. This means that if you want to change from a happy to a sad render target, you first have to fade out the happy render target to 0% before fading in the sad render target. You can see an example of this in Figure 9.3, where E1 is faded out to give way for E2.
- **Speech Channels:** To create nice-looking speech, you'll need at least two animation channels. This is to avoid always fading out a render target before starting the next. You can see this in Figure 9.3 with S1, S2, and S3. See how S2 starts before S1 has ended (same with S3 and S2). This is possible because more than one animation channel is used.

Each animation channel has one render target and one render weight at all times. The `FaceController` simply keeps track of this information and renders a `Face` by first setting the correct render targets and their corresponding weights. The definition of the `FaceController` class is as follows:

```
class FaceController
{
    friend class Face;
    public:
        FaceController(D3DXVECTOR3 pos, FACE *pFace);
        void Update(float deltaTime);
        void Render();

    public:
        Face *m_pFace;

        int m_emotionIndex;
        int m_speechIndices[2];

        D3DXVECTOR4 m_morphWeights;
        D3DXMATRIX m_headMatrix;

        Eye m_eyes[2];
};
```

Table 9.2 describes the `FaceController` members.

TABLE 9.2 FACECONTROLLER MEMBERS



Face Factory

Anyone who has modeled a face for a game knows it takes a long time to get right. First you'll have to make the model, and then you'll have to UV-map the face so that textures are applied correctly. Then you will have to create normal maps for the face, which in itself is a very time-consuming process. After this, you'll have to create the texture for the face, and finally you will have to create slightly edited copies of the face for the render targets. All this work goes into making just a single face. Now imagine that you have to make an army of individual-looking faces.… Surely there must be a better way then to repeat this time-consuming process for each of the soldiers in the army?

There is, of course. For the game *Oblivion*™, developers generated faces and also let the players design their own faces for the characters they would be playing. Figure 9.4 shows some screenshots of *Oblivion* and the characters created within it.

FIGURE 9.4
Some faces created in the game *Oblivion*™.

In this section I will look at creating a system for generating faces in runtime, just as in *Oblivion*. To achieve this, you will of course have to spend some more time and energy to create the generation system, but the result makes it possible for you to generate armies of individual faces at the click of a button.

To begin, I suggest that you revisit the code in Example 8.1, where a simple morphing calculation on the CPU was done, and the result was stored in a mesh. This is basically the whole idea behind creating a facial generation system. So far you have had render targets that change the face by giving it emotions, blinking eyelids, etc. However, there is nothing stopping us from changing the actual shape of a face altogether. Imagine that you have two copies of the same face, one with a

broad nose and one with a thin nose. *Voila*! You now can interpolate between the two meshes to create a wide variety of noses. Now, take this idea a bit further and add all possible variations you can think of (not just the nose). Here's a list of some of the render targets you can add into the equation:

- Nose width, height, length, and shape
- Mouth width, position, and shape
- Eye position and size
- Ear shape and position
- Jaw shape
- Head shape

Imagine that you have a long array of these meshes. All you need to do now in order to generate a new unique face is to randomize a weight for each of the faces and blend them together with additive blending using the CPU and store the result in a new mesh. This process is shown in Figure 9.5.

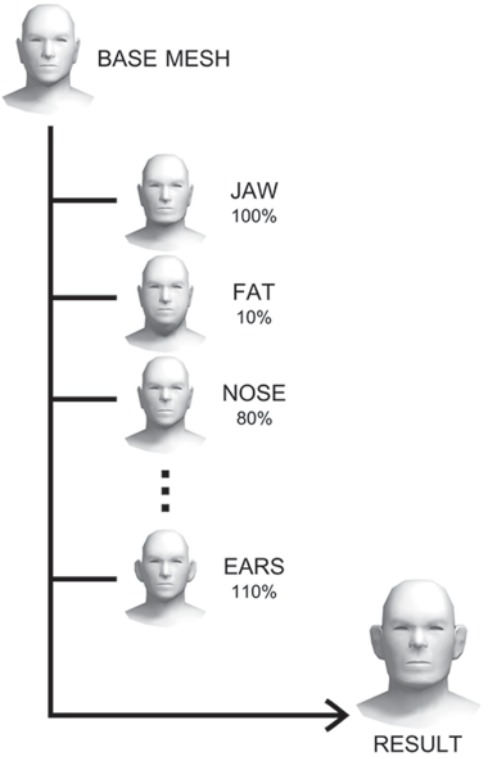


FIGURE 9.5
The process of generating a new face.

Figure 9.5 shows you how the base mesh is transformed by blending multiple weighted render targets and storing the result in a new mesh. However, the `Face` class has more meshes than just the base mesh. You need to perform the exact same procedure (with the same weights) for all of the emotion, speech, and blinking meshes within that face before you have a new face. To take care of the generation of new faces, I've created the `FaceFactory` class:

```
class FaceFactory
{
    public:
        FaceFactory(string filename);
        ~FaceFactory();
        FACE* GenerateRandomFace();

    private:
        void ExtractMeshes(D3DXFRAME *frame);
        ID3DXMesh* CreateMorphTarget(ID3DXMesh* mesh,
                                     vector<float> &morphWeights);

    public:
        ID3DXMesh *m_pBaseMesh;
        ID3DXMesh *m_pBlinkMesh;
        ID3DXMesh *m_pEyeMesh;
        vector<ID3DXMesh*> m_emotionMeshes;
        vector<ID3DXMesh*> m_speechMeshes;
        vector<ID3DXMesh*> m_morphMeshes;

        vector<IDirect3DTexture9*> m_faceTextures;
};
```

This class has a lot in common with the `Face` class. There's the base mesh, plus the blinking, emotion, and speech meshes. There's also a similar function for loading all the meshes from an .x file and extracting them from the hierarchy. What's new in this class is the array of render targets called `m_morphMeshes`. In this array, the render target that holds the different head, mouth, eye, and nose shapes, etc., is stored. There's also a function for generating a random face, and, as you can see, it returns a `Face` class that can be used with a face controller just as in previous examples. The following code is an excerpt from the `FaceFactory` class where a new random face is generated:

```
Face* FaceFactory::GenerateRandomFace()
{
    //Create random 0.0f - 1.0f morph weight for each morph target
    vector<float> morphWeights;
    for(int i=0; i<(int)m_morphMeshes.size(); i++)
    {
        float w = (rand()%1000) / 1000.0f;
        morphWeights.push_back(w);
    }

    //Next create a new empty face
    Face *face = new Face();

    //Then clone base, blink, and all emotion and speech meshes
    face->m_pBaseMesh = CreateMorphTarget(m_pBaseMesh,
                                          morphWeights);
    face->m_pBlinkMesh = CreateMorphTarget(m_pBlinkMesh,
                                           morphWeights);

    for(int i=0; i<(int)m_emotionMeshes.size(); i++)
    {
        face->m_emotionMeshes.push_back(
            CreateMorphTarget(m_emotionMeshes[i], morphWeights));
    }

    for(int i=0; i<(int)m_speechMeshes.size(); i++)
    {
        face->m_speechMeshes.push_back(
            CreateMorphTarget(m_speechMeshes[i], morphWeights));
    }

    //Set a random face texture as well
    int index = rand() % (int)m_faceTextures.size();
    m_faceTextures[index]->AddRef();
    face->m_pFaceTexture = m_faceTextures[index];

    //Return the new random face
    return face;
}
```

In this function I first create an array of floats (one weight for each morph mesh). Then using this array I create a new morph target for each of the face meshes (base, blink, emotion, and speech meshes) using the `CreateMorphTarget()` function:

```
ID3DXMesh* FaceFactory::CreateMorphTarget(
                        ID3DXMesh* mesh, vector<float> &morphWeights)
{
    if(mesh == NULL || m_pBaseMesh == NULL)
        return NULL;

    //Clone mesh
    ID3DXMesh* newMesh = NULL;
    if(FAILED(mesh->CloneMeshFVF(D3DXMESH_MANAGED,
                                 mesh->GetFVF(), pDevice, &newMesh)))
    {
        //Failed to clone mesh
        return NULL;
    }

    //Copy base mesh data
    FACEVERTEX *vDest = NULL, *vSrc = NULL;
    FACEVERETX *vMorph = NULL, *vBase = NULL;
    mesh->LockVertexBuffer(D3DLOCK_READONLY, (void**)&vSrc);
    newMesh->LockVertexBuffer(0, (void**)&vDest);
    m_pBaseMesh->LockVertexBuffer(D3DLOCK_READONLY, (void**)&vBase);

    for(int i=0; i < (int)mesh->GetNumVertices(); i++)
    {
        vDest[i].m_position = vSrc[i].m_position;
        vDest[i].m_normal = vSrc[i].m_normal;
        vDest[i].m_uv = vSrc[i].m_uv;
    }

    mesh->UnlockVertexBuffer();
    //Morph base mesh using the provided weights
    for(int m=0; m<(int)m_morphMeshes.size(); m++)
    {
    if(m_morphMeshes[m]->GetNumVertices() == mesh->GetNumVertices())
    {
```

```
            m_morphMeshes[m]->LockVertexBuffer(D3DLOCK_READONLY,
                                               (void**)&vMorph);

            for(int i=0; i < (int)mesh->GetNumVertices(); i++)
            {
                vDest[i].m_position +=
                 (vMorph[i].m_position - vBase[i].m_position) *
                 morphWeights[m];

                vDest[i].m_normal +=
                 (vMorph[i].m_normal - vBase[i].m_normal) *
                 morphWeights[m];
            }

            m_morphMeshes[m]->UnlockVertexBuffer();
        }
        }

        newMesh->UnlockVertexBuffer();
        m_pBaseMesh->UnlockVertexBuffer();

        return newMesh;
    }
```

The `CreateMorphTarget()` function creates a new target mesh for the new face by blending all the morph meshes with the provided weights. Note that this process runs on the CPU and is not limited to any amount of affecting morph meshes; it simply takes longer if you use more meshes to generate your random face. This is something to keep in mind if you plan to generate lots of faces. Also, since the faces are unique, it might affect your memory usage quite a lot. As said before, the resulting face generated by a `FaceFactory` can be used exactly like the original face with the `FaceController` class, the `Eye` class, etc. Some faces generated using this technique can be seen in Figure 9.6.

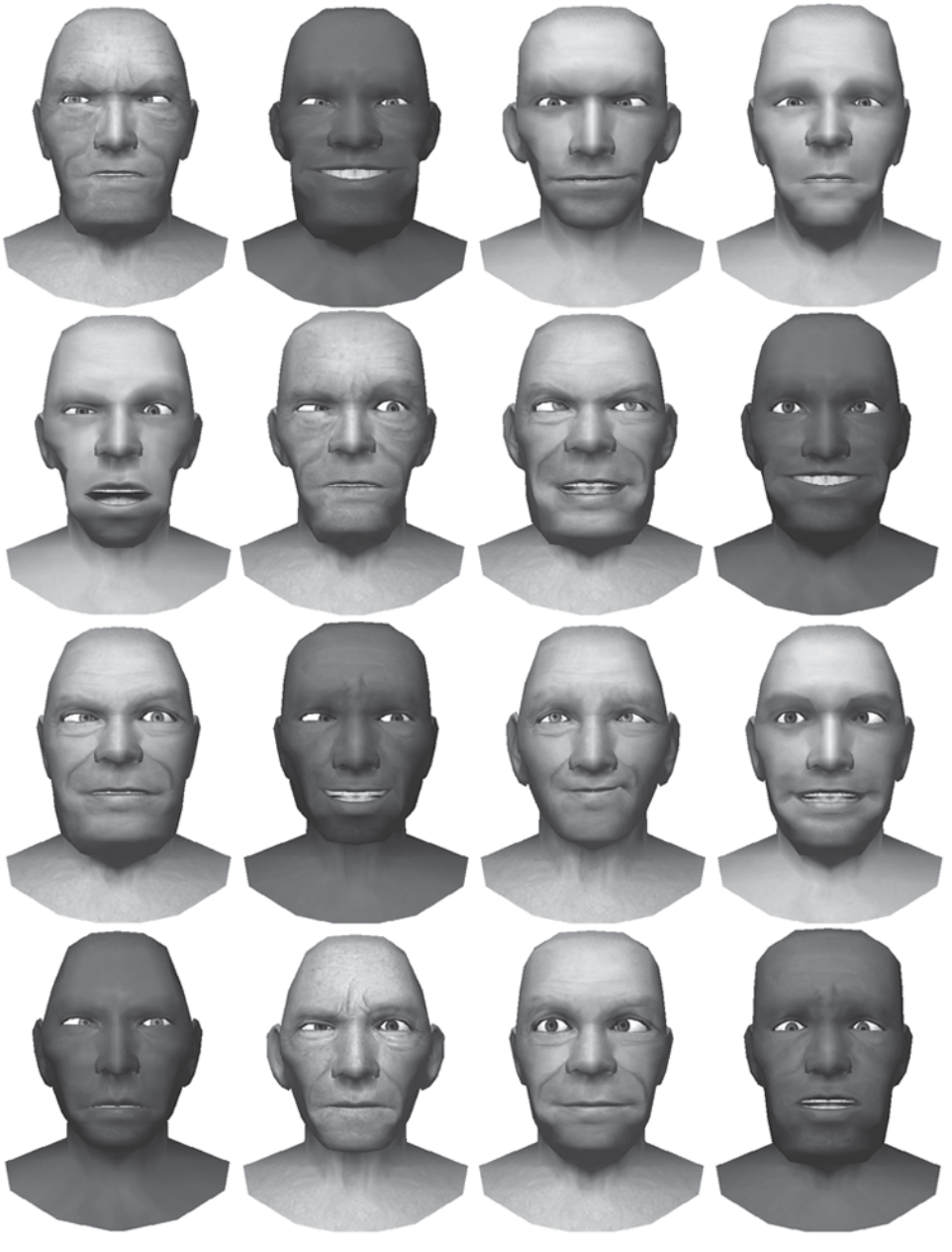


FIGURE 9.6
Custom faces generated using the `FaceFactory` class.

Conclusions

In this chapter you learned the basics of facial animation and how to use morphing animation to put together a simple `Face` class. I also separated the logic from the `Face` class and stuffed it into the `FaceController`, making the `Face` class a strict resource container. This way, many characters can reference the same face and render it with different expressions using the `FaceController` class.

Finally, we looked at a way of generating faces using CPU morphing as pre-processing stage. This can be a great way to produce variety in the non-player characters (NPCs) you meet in games such as RPGs, etc. It can also be one way for a small team to produce a large number of faces without having to create a new face for each character they intend to create.

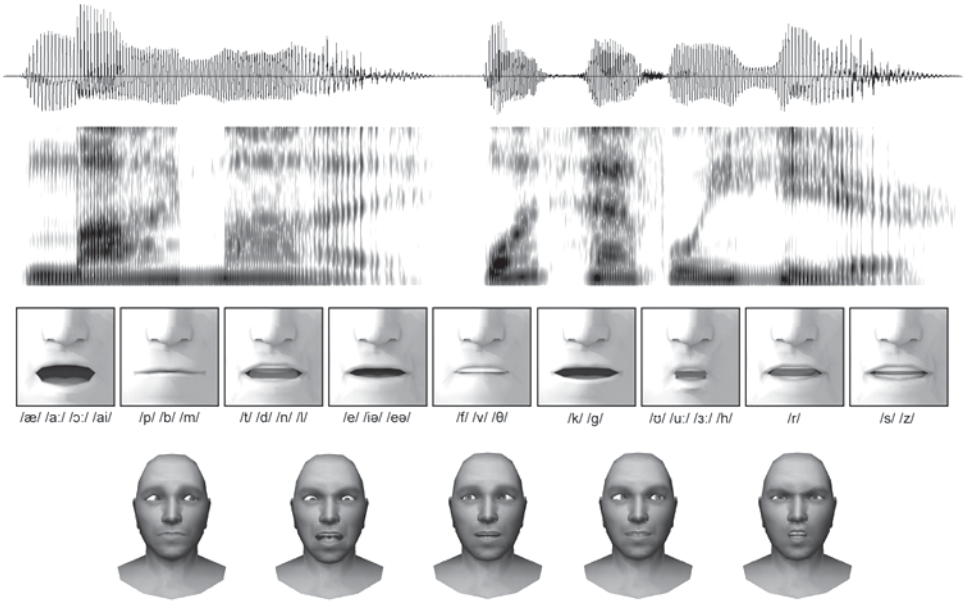
As an additional benefit, this system is easily extended to let the players themselves create their own faces for their characters (such as was seen in *Oblivion*, for example).

In the next chapter I'll focus on making talking characters, and I will cover topics such as lip-syncing.

Chapter 9 Exercises

- Create/blend the following emotions: anger, concentration, contempt, desire, disgust, excitement, fear, happiness, confusion, sadness, surprise.
- Make the eyes twitch—i.e., shift to random spots every once in a while as a part of the `Eye` class.
- Make functions for the `FaceController` class to set and control the current emotion.
- Make a program wherein the user can create a face by tuning the morph weights.

10 Making Characters Talk



You now have some idea of how to animate a character face using morphing animation as shown in the previous chapter. In this chapter I'll try to show you the basics of how to map speech to different mouth shapes of a character (a.k.a. lip-syncing). First I'll cover *phonemes* (the different sounds we make while talking), and then I'll cover the *visemes* (the phonemes' visual counterparts). After that I'll briefly cover in general terms how speech analysis is done to extract the phonemes from a recorded speech. Finally, I'll build a simple automated lip-syncing system.

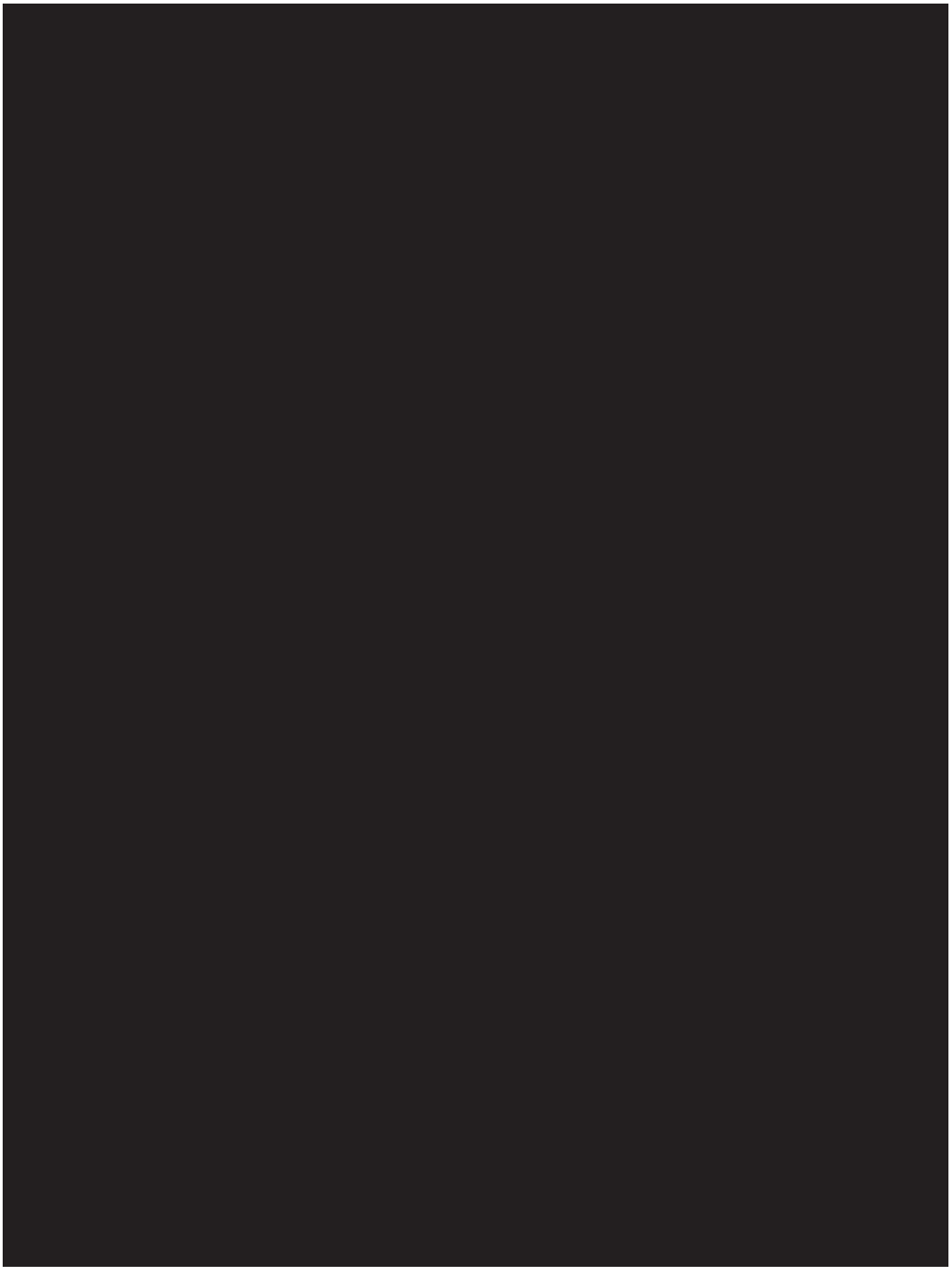
This chapter covers the following:

- Phonemes
- Visemes
- Basics of speech analysis
- Automatic lip-syncing

Phonemes

A phoneme could be called the atom of speech. In other words, it is the smallest discernable sound of a word that you hear. In the English language there are about 44 phonemes. I say about, because with the various dialects and regional differences you can add (or remove) a few of these phonemes. Table 10.1 shows a list of the most common phonemes found in the English language.

TABLE 10.1 ENGLISH PHONEMES



continued

There are many different notations for depicting phonemes. It doesn't matter much which notation you use as long as you understand the concept of phonemes. For example, check out Figure 10.1 where you can see the waveform of the sentence, "My, my…what have we here?"

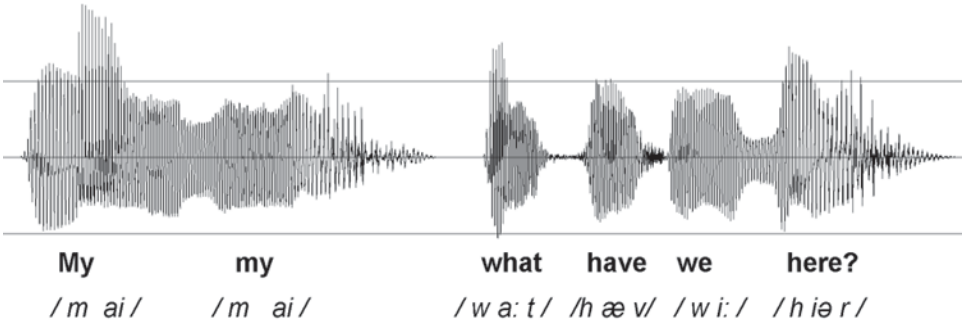


FIGURE 10.1
An example sentence with phonemes.

In Figure 10.1 the phonemes are shown below the actual words. Try to record a sentence yourself and use the phonemes in Table 10.1 to place the right phonemes in the right places. Just speak the words slowly and match the sounds to the corresponding phoneme. This is the easiest way to *manually* extract phonemes from a sentence. Later on I'll discuss the theory of how this can be done with software.

There are a lot of text-to-speech applications that take text as input, transforming it into a series of phonemes that can be played back. A good place to start for text-to-speech programming is Microsoft's Speech API (SAPI). This API contains a lot of tools you can use, such as phoneme extraction, text-to-speech, and more.

When creating lip-syncing for game characters, it is not that important that you match all phonemes in a sentence just right. There are two reasons for this. First, several phonemes will have the same mouth shape (i.e., viseme). So whether you classify a sound as /*a:*/ or /*æ*/ has no impact on the end result. Secondly, timing is more important because it is easier for people to notice this type of error. If you have ever seen a dubbed movie, you know what I'm talking about. So, let's say you now have a recorded speech and you have dotted down the phonemes used in the sentence on a piece of paper… Now what?

Visemes

Whereas a phoneme is the smallest unit of speech that you can *hear*, a viseme is the smallest unit of speech that you can *see*. In other words, a viseme is the shape your mouth makes when making a particular phoneme. Not only is the mouth shape important, but the positions of your tongue and your teeth matter too. For example, try saying the following phonemes: */n/* (**n**ews) and */a:/* (f**a**ther). The mouth shape remains pretty much the same, but did you notice the difference? The tongue went up to the roof of the mouth while saying */n/*, and when you said */a:/* the tongue was lying flat on the "floor" of the mouth. This is one of many small visual differences that our subconscious easily detects whenever it is wrong, or "off."

Deaf people have learned to take great advantage of this concept when they do lip reading. If you've ever seen a foreign movie dubbed to your native tongue, you have witnessed a case where the phonemes (what you hear) and the visemes (what you see) don't match. In games these days we try to minimize this gap between phonemes and visemes as much as possible.

So if there are about 44 phonemes for the English language, how many visemes are there? Well, the Disney animators of old used 13 archetypes of mouth positions when they created animations. (You don't have to implement these 13 visemes for each character you create, however; you can get away with less [Lander00]). Figure 10.2 shows a template of visemes you can use when creating your own characters:

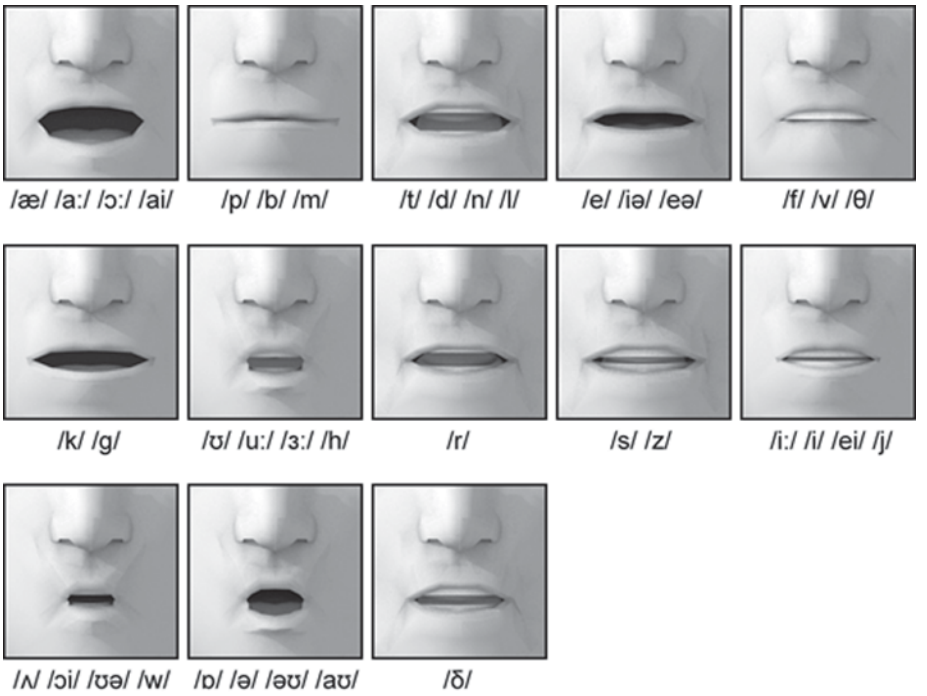


FIGURE 10.2
Viseme templates.

Okay, it is about time I show you some code in this chapter. Here's the class I'll use to describe a viseme keyframe:

```
class Viseme
{
    public:
        VISEME();
        VISEME(int target, float amount, float time);

    public:
        int m_morphTarget;
        float m_blendAmount;
        float m_time;
};
```

This class works as a simple keyframe where the morph target (`m_morphTarget`) will be blended into at time `m_time`. The `m_blendAmount` variable determines the final blend amount for this keyframe. I have also added the two following functions to the `FaceController` class to handle speech lip-syncing:

```
void FaceController::Speak(vector<Viseme> &visemes)
{
    //Copy visemes to the m_visemes array
    m_visemes.clear();
    for(int i=0; i<visemes.size(); i++)
        m_visemes.push_back(visemes[i]);

    //Reset playback variables
    m_visemeIndex = 1;
    m_speechTime = 0.0f;
}

void FaceController::UpdateSpeech(float deltaTime)
{
    m_speechTime += deltaTime;

    if(m_visemeIndex > 0 && m_visemeIndex < m_visemes.size())
    {
        //Get visemes to blend between
        VISEME &v1 = m_visemes[m_visemeIndex - 1];
        VISEME &v2 = m_visemes[m_visemeIndex];

        //Set speech meshes
        m_speechIndices[0] = v1.m_morphTarget;
        m_speechIndices[1] = v2.m_morphTarget;

        //Set blend amounts
        float timeBetweenVisemes = v2.m_time - v1.m_time;
        float p = (m_speechTime - v1.m_time) / timeBetweenVisemes;
        m_morphWeights.z = (1.0f - p) * v1.m_blendAmount;
        m_morphWeights.w = p * v2.m_blendAmount;

        //Update index
        if(m_speechTime >= v2.m_time)
            m_visemeIndex++;
    }
    else
    {
        m_morphWeights.z = 0.0f;
        m_morphWeights.w = 0.0f;
    }
}
```

The `Speak()` function is the command to a face controller that will start the playback/lip-syncing of a character. The `UpdateSpeech()` function is called each frame the blending of the viseme key frames is done in this function.

To keep this book focused, I won't go into details on how to play sound or speech. There are several libraries available online, not to mention the DirectSound API, DirectShow API, etc. But since I do need to play sounds, I'll use the simple `PlaySound()` *function available in the* `winmm.lib`*. So to play a sound with this function you simply call it like this:*

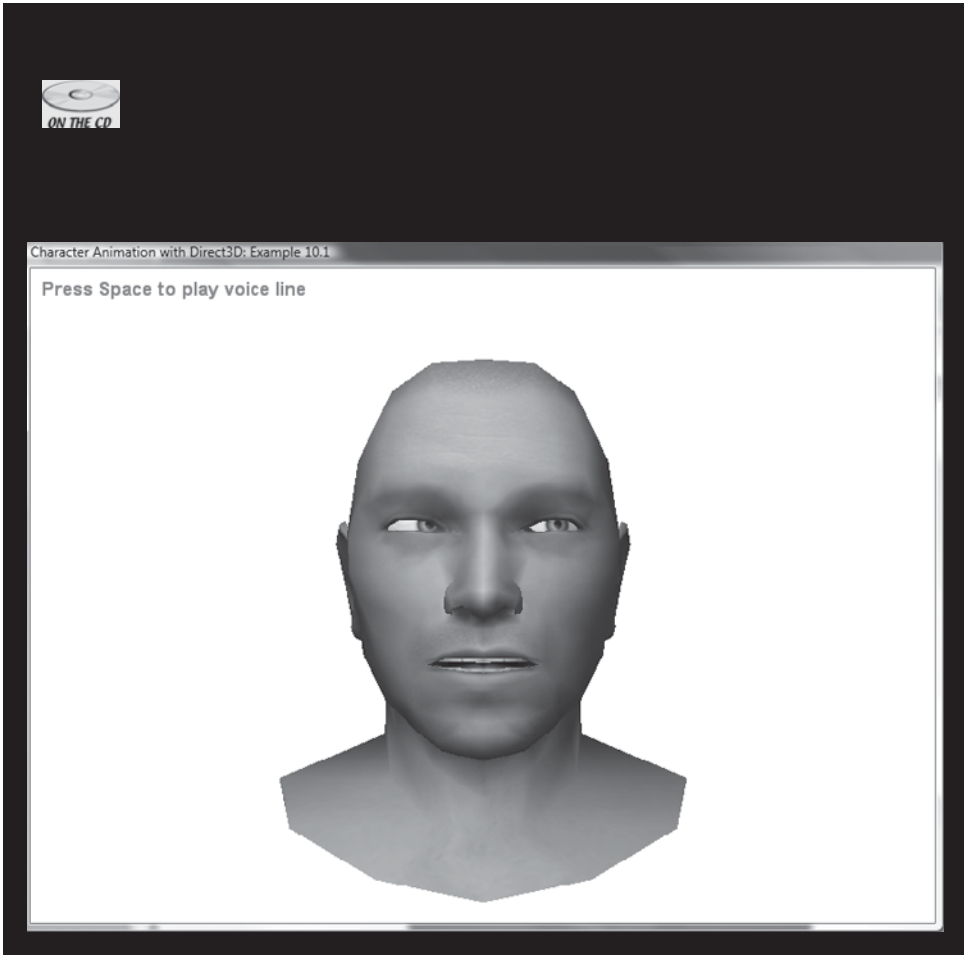
```
//Stop old sound
PlaySound(0, 0, 0);

//Play sound
PlaySound("somefile.wav", NULL, SND_FILENAME | SND_ASYNC);
```

However, I recommend that you use something other than this in real-time applications.

So to make the face of the character "mime" words, you need to create an array of viseme keyframes and call the `Speak()` function of the `FaceController` class like this:

```
//Create viseme queue
vector<Viseme> visemes;
visemes.push_back(Viseme(VISEME::O, 0.0f,  0.0f));
visemes.push_back(Viseme(VISEME::R, 1.0f,  0.25f));
...
visemes.push_back(Viseme(VISEME::SZ, 0.0f,  2.5f));

m_pFaceController->Speak(visemes);
```

In this example code I've used an enumeration (`VISEME`) to contain the indices of the different morph targets. These indices are then used when initializing the Viseme objects. That's it! You'll find all this stuff in action in Example 10.1.



Basics of Speech Analysis

These days, games have thousands of voice lines. Doing manual lip-syncing for all these lines is downright uneconomical (not to mention tedious). Therefore most commercial game engines use systematic lip-syncing. These game engines employ offline preprocessing algorithms where the phonemes are extracted and linked to the various voice lines in a game. Often these algorithms take both the text of the line being read as well as the sound data to create the highest-quality lip-syncing possible.

So how can you analyze speech algorithmically? Well, when you speak you use certain frequencies of sound for certain phonemes. By classifying which combination of frequencies goes with each of the 44 different phonemes, you can make a

fairly educated guess as to which phoneme is being spoken. Figure 10.3 shows the waveform from the previous speech sample together with the spectrograph of the same sample (a spectrograph shows the frequency and amplitude of a signal).

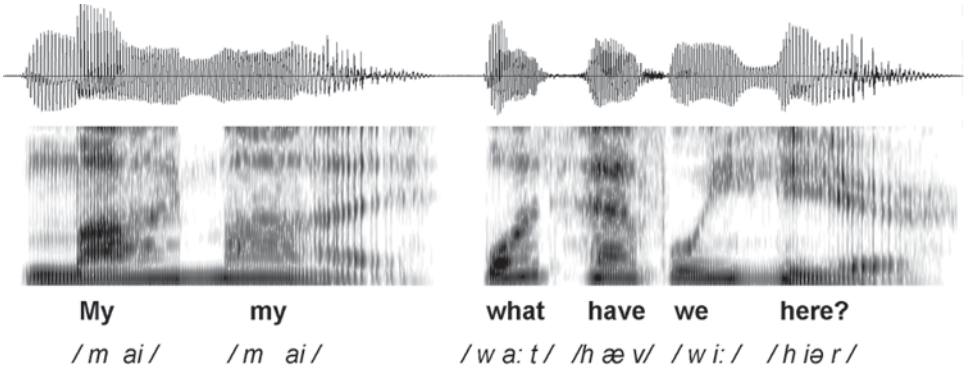


FIGURE 10.3
Waveform and spectrograph of a voice sample.

As you can see in Figure 10.3, distinct patterns can be seen in the spectrograph as the phonemes are spoken. As a side note, speech-to-text applications take this analysis one step further and use Hidden Markov Models (HMM) to figure out which exact word is being spoken. Luckily we don't need to dive that deep in order to create reasonable lip-syncing.

If you are interested in analyzing speech and making your own phoneme extractor, you'll need to run the speech data through a Fourier Transform. This will give you the data in the frequency domain, which in turn will help you build the spectrogram and help you classify the phonemes. Check out **www.fftw.org** *for a Fast-Fourier-Transform library in C.*

Analyzing speech and extracting phonemes is a rather CPU-intense process and is therefore pre-processed in all major game engines. However, some games in the past have used a real-time lip-syncing system based simply on the current amplitude of the speech [Simpson04]. With this approach the voice lines are evaluated just a little ahead of the playback position to determine which mouth shape to use. In the coming sections I will look at a similar system and get you started on analyzing raw speech data.

Sound Data

Before you can start to analyze voice data, I'll need to go off on a tangent and cover how to actually load some raw sound data. No matter which type of sound format you will actually use, once the sound data has been decompressed and decoded, the raw sound data will be the same across all sound formats. In this chapter I'll just use standard uncompressed WAVE files for storing sound data. However, for projects requiring large amounts of voice lines, using uncompressed sound is of course out of the question.

Two open-source compression schemes available for free are OGG and SPEEX, which you can find online:

http://www.vorbis.com/
http://www.speex.org/

OGG is aimed mainly at music compression and streaming, but it is easy enough to get up and running. SPEEX, on the other hand, focuses only on speech compression.

The WAVE Format

There are several good tutorials on the Web explaining how to load and interpret WAVE (.wav) files, so I won't dig too deep into it here. The WAVE format builds on the Resource Interchange File Format (RIFF). RIFF files store data in chunks, where the start of each chunk is marked with a 4-byte ID describing what type of chunk it is, as well as 4 bytes containing the size of the chunk (a `long`). The WAVE file contains all information about the sound—number of channels, sampling rate, number of bits per sample, and much more. Figure 10.4 shows how a WAVE file is organized.

There are many other different types of chunks that can be stored in a WAVE file. Only the Format and Data chunks are mandatory. Table 10.2 shows the different fields of the Format chunk and their possible values.

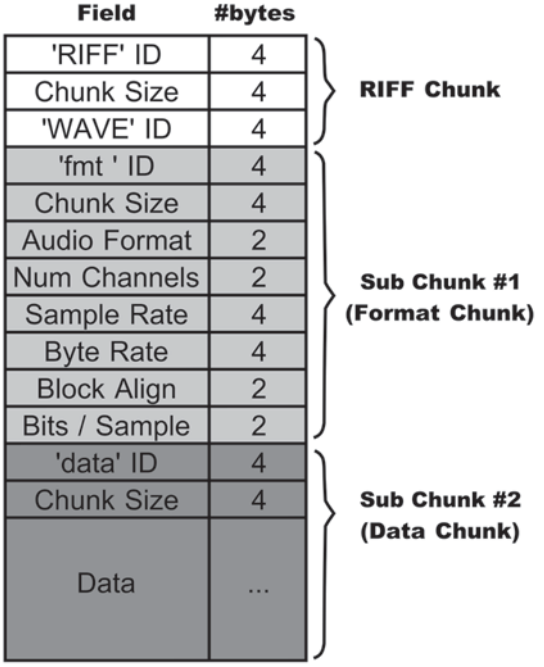


FIGURE 10.4
WAVE file format.

TABLE 10.2 THE WAVE FORMAT CHUNK

For a full description of the WAVE file format and the different chunks available, check out http://www.sonicspot.com/guide/wavefiles.html.

I'll assume that the data stored in the "data" chunk is uncompressed sound data in the Pulse-Code Modulation (PCM) format. This basically means that the data is stored as a long array of values where each value is the amplitude of the sound at a specific point in time. The quickest and dirtiest way to access the data is to simply open a stream from the sound file and start reading from byte 44 (where the data field starts). Although this will work if you know the sound specifications, it isn't really recommended. The `WaveFile` class I'll present here will do minimal error checking before reading and storing the actual sound data:

```
class WaveFile
{
    public:
        WaveFile();
        ~WaveFile();
    void Load(string filename);
        short GetMaxAmplitude();
        short GetAverageAmplitude(float startTime, float endTime);
        float GetLength();

    public:
        long m_numSamples;
        long m_sampleRate;
        short m_bitsPerSample;
        short m_numChannels;
        short *m_pData;
};
```

The `Load()` function of the `WaveFile` class loads the sound data and performs some minimal error checking. For example, I assume that only uncompressed, 16-bit WAVE files will be used. You can easily expand this class yourself if you need to load 8-bit files, etc. If a `WaveFile` object is created successfully and a WAVE file is loaded, the raw data can be accessed through the `m_pData` pointer. The following code shows the code for the `Load()` function of the `WaveFile` class:

```
void WaveFile::Load(string filename)
{
    ifstream in(filename.c_str(), ios::binary);

    //RIFF
    char ID[4];
    in.read(ID, 4);
```

```
if(ID[0] != 'R' || ID[1] != 'I' || ID[2] != 'F' || ID[3] != 'F')
{
    //Error: 4 first bytes should say 'RIFF'
}

//RIFF Chunk Size
long fileSize = 0;
in.read((char*)&fileSize, sizeof(long));

//The actual size of the file is 8 bytes larger
fileSize += 8;

//WAVE ID
in.read(ID, 4);
if(ID[0] != 'W' || ID[1] != 'A' || ID[2] != 'V' || ID[3] != 'E')
{
    //Error: ID should be 'WAVE'
}

//Format Chunk ID
in.read(ID, 4);
if(ID[0] != 'f' || ID[1] != 'm' || ID[2] != 't' || ID[3] != ' ')
{
    //Error: ID should be 'fmt '
}

//Format Chunk Size
long formatSize = 0;
in.read((char*)&formatSize, sizeof(long));

//Audio Format
short audioFormat = 0;
in.read((char*)&audioFormat, sizeof(short));
if(audioFormat != 1)
{
    //Error: Not uncompressed data!
}

//Num Channels
in.read((char*)&m_numChannels, sizeof(short));

//Sample Rate
in.read((char*)&m_sampleRate, sizeof(long));
```

```
    //Byte Rate
    long byteRate = 0;
    in.read((char*)&byteRate, sizeof(long));

    //Block Align
    short blockAlign = 0;
    in.read((char*)&blockAlign, sizeof(short));

    //Bits Per Sample
    in.read((char*)&m_bitsPerSample, sizeof(short));
    if(m_bitsPerSample != 16)
    {
        //Error: This class only supports 16-bit sound data
    }

    //Data Chunk ID
    in.read(ID, 4);
    if(ID[0] != 'd' || ID[1] != 'a' || ID[2] != 't' || ID[3] != 'a')
    {
        //Error: ID should be 'data'
    }

    //Data Chunk Size
    long dataSize;
    in.read((char*)&dataSize, sizeof(long));
    m_numSamples = dataSize / 2; //<-- Divide by 2 (short has 2 bytes)

    //Read the Raw Data
    m_pData = new short[m_numSamples];
    in.read((char*)m_pData, dataSize);

    in.close();
}
```

At the end of this function the raw sound data will be stored at the `m_pData` pointer as a long array of short values. The value of a single sample ranges from -32768 to 32767, where a value of 0 marks silence. The other functions of this class I will cover later as we do our amplitude-based lip-syncing system.

Automatic Lip-Syncing

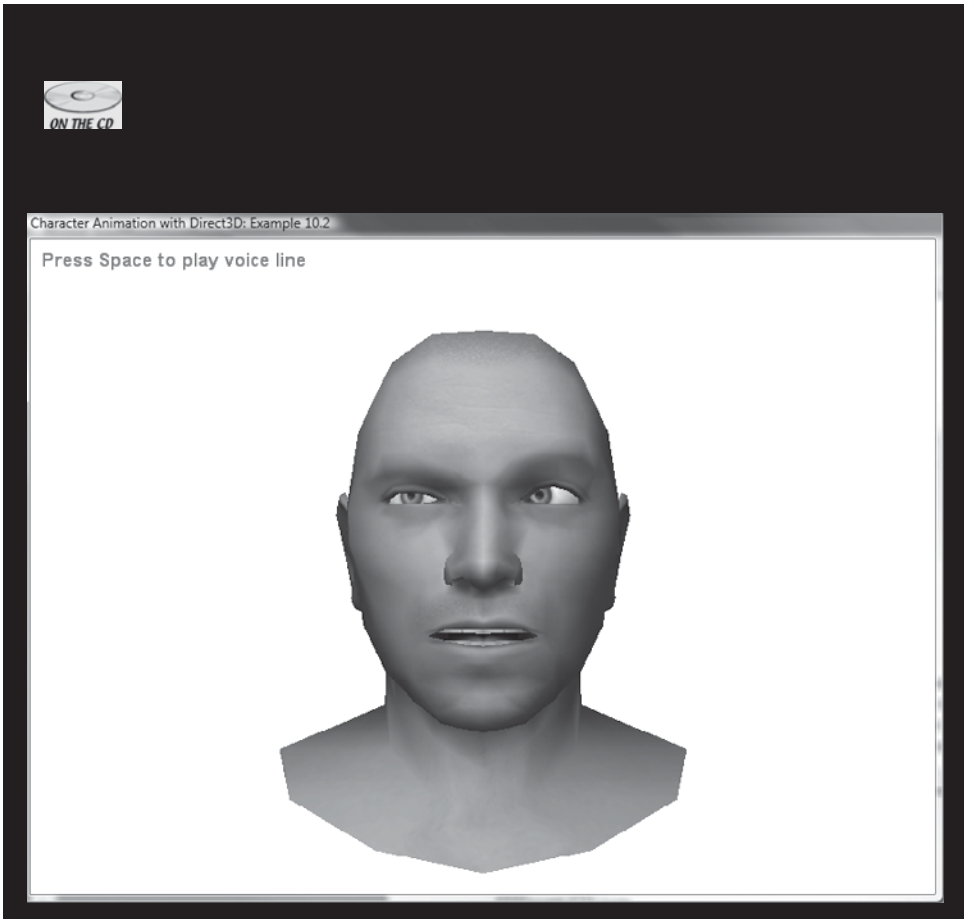
In the previous section you learned how to load a simple WAVE file and how to access the raw PCM data. In this section I will create a simplified lip-syncing system by analyzing the amplitude of a voice sample [Simpson04]. The main point of this approach is not to create perfect lip-syncing but rather to make the lips move in a synchronized fashion as the voice line plays. So, for instance, when the voice sample is silent, the mouth should be closed. The following function returns the average amplitude of a voice sample between two points in time:

```
short WAVE::GetAverageAmplitude(float startTime, float endTime)
{
    if(m_pData == NULL)
        return 0;

    //Calculate start & end sample
    int startSample = (int)(m_sampleRate * startTime) * m_numChannels;
    int endSample = (int)(m_sampleRate * endTime) * m_numChannels;

    if(startSample >= endSample)
        return 0;

    //Calculate the average amplitude between start and end sample
    float c = 1.0f / (float)(endSample - startSample);
    float avg = 0.0f;
    for(int i=startSample; i<endSample && i<m_numSamples; i++)
    {
        avg += abs(m_pData[i]) * c;
    }

    avg = min(avg, (float)(SHRT_MAX - 1));
    avg = max(avg, (float)(SHRT_MIN + 1));

    return (short)avg;
}
```

With this function you can easily create an array of visemes by matching a certain amplitude range to a certain viseme. This is done in the `FaceController::Speak()` function:

```
void FaceController::Speak(WAVE &wave)
{
    m_visemes.clear();

    //Calculate which visemes to use from the WAVE file data
    float soundLength = wave.GetLength();

    //Since the wave data oscillates around zero,
    //bring the max amplitude down to 30% for better results
    float maxAmp = wave.GetMaxAmplitude() * 0.3f;

    for(float i=0.0f; i<soundLength; i += 0.1f)
    {
        short amp = wave.GetAverageAmplitude(i, i + 0.1f);
        float p = min(amp / maxAmp, 1.0f);

        if(p < 0.2f)
        {
            m_visemes.push_back(VISEME(0, 0.0f, i));
        }
        else if(p < 0.4f)
        {
            float prc = max((p - 0.2) / 0.2f, 0.3f);
            m_visemes.push_back(VISEME(3, prc, i));
        }
        else if(p < 0.7f)
        {
            float prc = max((p - 0.4f) / 0.3f, 0.3f);
            m_visemes.push_back(VISEME(1, prc, i));
        }
        else
        {
            float prc = max((p - 0.7f) / 0.3f, 0.3f);
            m_visemes.push_back(VISEME(4, prc, i));
        }
    }

    m_visemeIndex = 1;
    m_speechTime = 0.0f;
}
```

Here I create a viseme for every 100 milliseconds, but you can try out different amounts of visemes per second. Of course the result will be a bit worse comparing this method to the previous one where the visemes were created manually, but the major upside with this approach is that you can quickly get "decent" looking lip-syncing with very little effort and no pre-processing.



Conclusions

This chapter covered the basics of lip-syncing and how to make a character "speak" a voice line. This is still a hot research topic that is constantly being improved upon. However, for games using thousands of voice lines, the focus is almost always on making the process as cheap and pain free as possible as long as the results are

"good enough." In this chapter I showed one way of doing the lip-synching automatically using only the amplitude of a voice sample. Granted, this wouldn't be considered high enough quality to work in a next-generation project, but at least it serves as a starting point for you to get started with analyzing voice samples. If you want to improve this system, I suggest you look into analyzing the voice data with Fourier Transforms and try to classify the different phonemes.

Further Reading

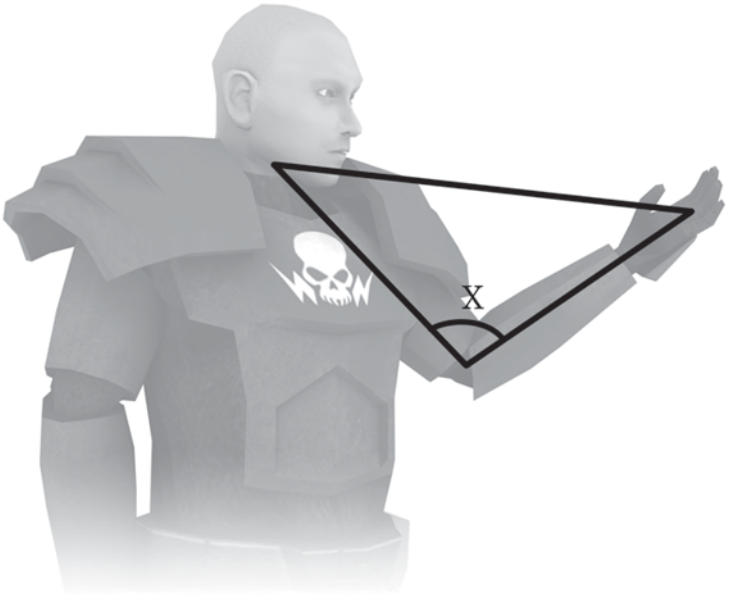
[Lander00] Lander, Jeff, "Read My Lips: Facial Animation Techniques." Available online at http://www.gamasutra.com/view/feature/3179/read_my_lips_facial_animation_.php, 2000.

[Simpson04] Simpson, Jake, "A Simple Real-Time Lip-Synching System." *Game Programming Gems 4*, Charles River Media, 2004.

[Lander00b] Lander, Jeff, "Flex Your Facial Muscles." Available online at http://www.gamasutra.com/features/20000414/lander_pfv.htm, 2000.

This page intentionally left blank

11 Inverse Kinematics



This chapter will introduce you to the concept of *inverse kinematics* (IK). The goal is to calculate the angles of a chain of bones so that the end bone reaches a certain point in space. IK was first used in the field of robotics to control robotic arms, etc. There are plenty of articles about IK in this field if you give it a search on Google. In this chapter, however, I'll show you how to put this idea to work on your game character. IK can be used for many different things in games, such as placing hands on items in the game world, matching the feet of a character to the terrain, and much more.

So why should you bother implementing inverse kinematics? Well, without it your character animations will look detached from the world, or "canned." IK can be used together with your keyframed animations. An example of this is a character opening a door. You can use IK to "tweak" the door-opening animation so that the hand of the character always connects with the door handle even though the handle may be placed in different heights on different doors.

This chapter presents two useful cases of IK. The first is a simple "Look-At" example, and the second is a Two-Joint "Reach" example. In short, this chapter covers the following:

- Inverse kinematics overview
- "Look-At" IK
- Two-Joint "Reach" IK

A big thanks goes out to Henrik Enqvist at Remedy Entertainment for the sample code of the IK examples covered in this chapter.

Introduction to Inverse Kinematics

Before I cover inverse kinematics I'll first cover the concept of *forward kinematics* (FK). Forward kinematics is something you've come across many times throughout this book already. Forward kinematics is the problem of solving the end point, given a chain of bones and their angles. An example of forward kinematics can be seen in Figure 11.1.

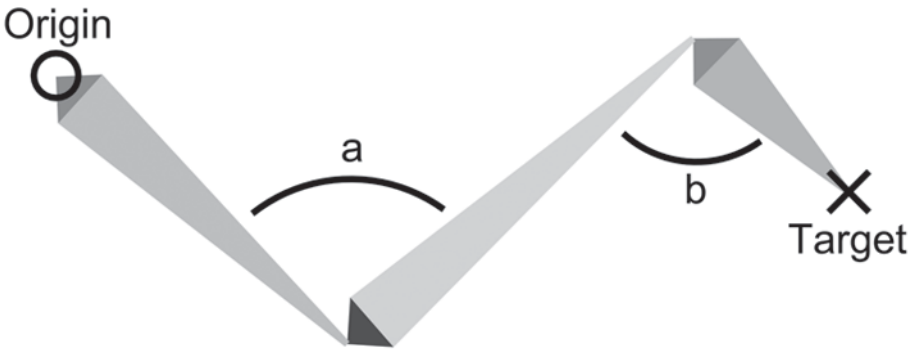


FIGURE 11.1
Forward kinematics: Calculating the target, when knowing the origin, the bones, and their angles.

The linked bones are also known as a kinematics chain, where the change in a bone's orientation also affects the children of that bone (something with which you should already be familiar after implementing a skinned character using bone hierarchies).

Forward kinematics come in very handy when trying to link something to a certain bone of a character. For example, imagine a medieval first-person shooter (FPS) in which you're firing a bow. A cool effect would be to have the arrows "stick" to the enemy character if you have gotten a clean hit. The first problem you would need to solve is to determine which bone of the character was hit. A simple way to do this is to check which polygon of the character mesh was pierced and then see which bone(s) govern the three vertices of this polygon. After this you would need to calculate the position (and orientation) of the arrow in the bone space of the bone you're about to link the arrow to. Next you would update the position of the arrow each frame using forward kinematics. Alternatively, the bone may have its own bounding volume, and then you can just check if the arrow intersects any of these instead.

With forward kinematics you don't know the end location of the last bone in the kinematics chain. With inverse kinematics, on the other hand, you *do* know the end location (or target) that you want the kinematics chain to reach. What you don't know are the different angles (and orientations) of the joints (**a** and **b** in Figure 11.1). Solving the forward kinematics problem is relatively easy, but coming up with an efficient (and general) solution for the inverse kinematics problems is much harder (see Figure 11.2).

FIGURE 11.2
Example of inverse kinematics.

In Figure 11.2 you can see a 2D example of inverse kinematics. With inverse as opposed to forward kinematics, there is often more than one solution to a problem. In Figure 11.2, three example solutions to the same problem are shown (although there are more). Imagine then how many more solutions there are in 3D! Luckily when it comes to game programming, cutting corners is allowed (and often necessary). By reducing and adding more information about the

problem, you can go from this near-impossible problem to a quite manageable one. This chapter will cover some approaches to solving the problem of inverse kinematics for characters.

Solving the IK Problem

Solutions to IK problems come in two flavors: *analytical* and *numerical*. With analytical solutions, you have an equation that can be solved directly. This is the fast and preferred way of solving IK problems. However, with several links in the IK chain, the analytical solution is rarely an option. Numerical solutions attempt to find approximate (but somewhat accurate) solutions to the problem. The approximations are usually done by either iterating over the result, which finally converges toward the solution, or dividing the problem into smaller, more manageable chunks and solving those separately. Numerical IK solutions also tend to be more expensive compared to their analytical counterpart.

Two popular numerical methods for solving IK problems are *cyclic coordinate decent* (CCD) and the *Jacobian matrix*. Cyclic coordinate decent simplifies the problem by looking at each link separately. CCD starts at the leaf node and moves up the chain, trying to minimize the error between the end point and the goal. This approach can require a fair amount of passes over the IK chain before the result is acceptable. CCD also suffers from the problem of sometimes creating unnatural-looking solutions. For example, since the first link to be considered is the leaf (e.g., the wrist or ankle), it will first try to minimize the error on this link, which might result in really twisted hands or feet.

The Jacobian matrix, on the other hand, describes the entire IK chain. Each column in the Jacobian matrix describes the change of the end point (approximated linearly) as one of the links is rotated. Solving the Jacobian matrix is slow but produces better-looking results (in general) than the cyclic coordinate decent solution does. Since the Jacobian method is rather heavy on math, I'll leave it out of this book, but for those interested, simply search for "The Jacobian IK" in your favorite search engine.

Look-At Inverse Kinematics

To start you off with IK calculations, I'll start with the simplest example: having only one bone orientation to calculate. Figure 11.3 shows an example of Look-At IK.

FIGURE 11.3
Look-At Inverse Kinematics.

In Figure 11.3 the character is facing the target (black ball) no matter where the ball is compared to the character. Since the target might be moving dynamically in the game, there is no way to make a keyframed animation to cover all possible "view angles." In this case, the IK calculation is done on the head bone and can easily be blended together with normal keyframed animations.

One more thing you need to consider is, of course, what should happen when the Look-At target is behind the character or outside the character's field of view (FoV). The easiest solution is just to cap the head rotation to a certain view cone. A more advanced approach would be to play an animation that turns the character around to face the target and then use the Look-At IK to face the target. In either case you need to define the character's field of view. Figure 11.4 shows an example FoV of 120 degrees.

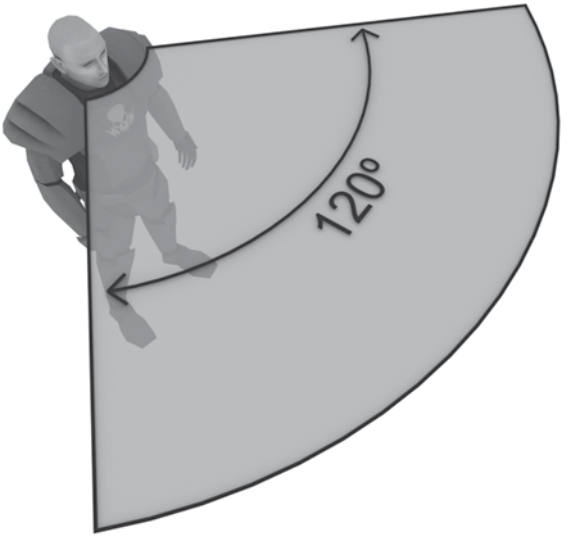


FIGURE 11.4
Limiting the field of view (FoV).

So what I'll try to achieve in the next example is a character that can *look at* a certain target in its field of view (i.e., turn the character's head bone dynamically). To do this I'll use the `InverseKinematics` class. This class encapsulates all the IK calculations, the updating of the bone matrices, etc:

```
class InverseKinematics
{
    public:
        InverseKinematics(SkinnedMesh* pSkinnedMesh);
        void UpdateHeadIK();
        void ApplyLookAtIK(D3DXVECTOR3 &lookAtTarget, float maxAngle);

    private:
        SkinnedMesh *m_pSkinnedMesh;
        Bone* m_pHeadBone;
        D3DXVECTOR3 m_headForward;
};
```

The constructor of the `InverseKinematics` class takes a pointer to the skinned mesh you want to "operate on." The constructor finds the head bone and does the necessary initializations for the IK class. The magic happens in the `ApplyLookAtIK()` function. As you can see, this function takes a Look-At target (in world space) and a max angle defining the view cone (FoV) of the character. Here's the initialization code of the `InverseKinematics` class as found in the class constructor:

```
InverseKinematics::InverseKinematics(SkinnedMesh* pSkinnedMesh)
{
    m_pSkinnedMesh = pSkinnedMesh;

    // Find the head bone
    m_pHeadBone = (Bone*)m_pSkinnedMesh->GetBone("Head");

    // Exit if there is no head bone
    if(m_pHeadBone != NULL)
    {
        // Calculate the local forward vector for the head bone

        // Remove translation from head matrix
        D3DXMATRIX headMatrix;
        headMatrix = m_pHeadBone->CombinedTransformationMatrix;
        headMatrix._41 = 0.0f;
        headMatrix._42 = 0.0f;
```

```
            headMatrix._43 = 0.0f;
            headMatrix._44 = 1.0f;

            D3DXMATRIX toHeadSpace;
            if(D3DXMatrixInverse(&toHeadSpace, NULL, &headMatrix) == NULL)
                return;

            // The model is looking toward -z in the content
            D3DXVECTOR4 vec;
            D3DXVec3Transform(&vec, &D3DXVECTOR3(0, 0, -1), &toHeadSpace);
            m_headForward = D3DXVECTOR3(vec.x, vec.y, vec.z);
        }
    }
```

First I locate the head bone (named Head in the example mesh). Next I remove the transformation from the combined transformation matrix by setting element 41, 42, 43, and 44 in the matrix to 0, 0, 0, and 1 respectively. I then calculate the inverse of the resulting matrix. This lets you calculate the head forward vector (in the local head bone space) shown in Figure 11.5.

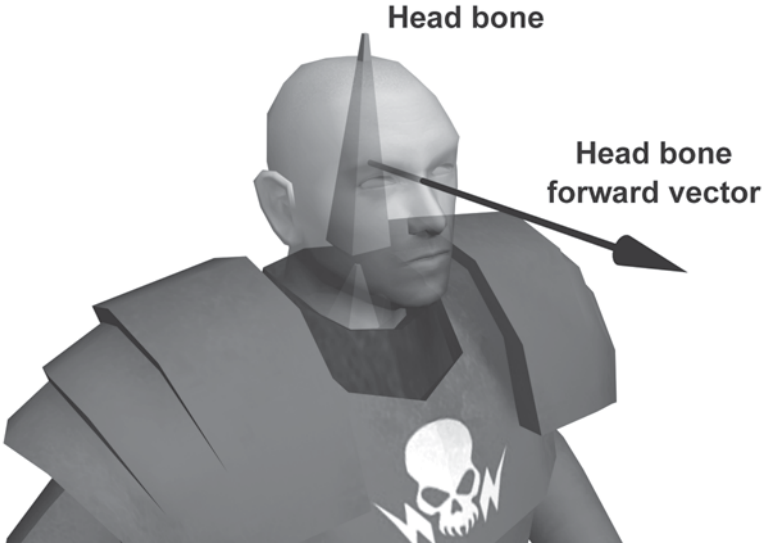


FIGURE 11.5
The forward vector of the head bone.

The forward vector of the head bone is calculated when the character is in the reference pose and the character is facing in the negative Z direction. You'll need this vector later on when you update the Look-At IK. Next is the `ApplyLookAtIK()` function:

```
void InverseKinematics::ApplyLookAtIK(D3DXVECTOR3 &lookAtTarget,
float maxAngle)
{
    // Start by transforming to local space
    D3DXMATRIX mtxToLocal;
    D3DXMatrixInverse(&mtxToLocal, NULL,
                      &m_pHeadBone->CombinedTransformationMatrix);

    D3DXVECTOR3 localLookAt;
    D3DXVec3TransformCoord(&localLookAt, &lookAtTarget, &mtxToLocal );

    // Normalize local look at target
    D3DXVec3Normalize(&localLookAt, &localLookAt);

    // Get rotation axis and angle
    D3DXVECTOR3 localRotationAxis;
    D3DXVec3Cross(&localRotationAxis, &m_headForward, &localLookAt);
    D3DXVec3Normalize(&localRotationAxis, &localRotationAxis);

    float localAngle = acosf(D3DXVec3Dot(&m_headForward,
                                         &localLookAt));

    // Limit angle
    localAngle = min( localAngle, maxAngle );

    // Apply the transformation to the bone
    D3DXMATRIX rotation;
    D3DXMatrixRotationAxis(&rotation, &localRotationAxis, localAngle);
    m_pHeadBone->CombinedTransformationMatrix = rotation *
                m_pHeadBone->CombinedTransformationMatrix;

    // Update changes to child bones
    if(m_pHeadBone->pFrameFirstChild)
    {
        m_pSkinnedMesh->UpdateMatrices(
                      (Bone*)m_pHeadBone->pFrameFirstChild,
                      &m_pHeadBone->CombinedTransformationMatrix);
    }
}
```

This function uses the shortest arc algorithm [Melax00], to calculate the angle to rotate the head bone so that it faces the Look-At target. Figure 11.6 shows the shortest arc algorithm in action.

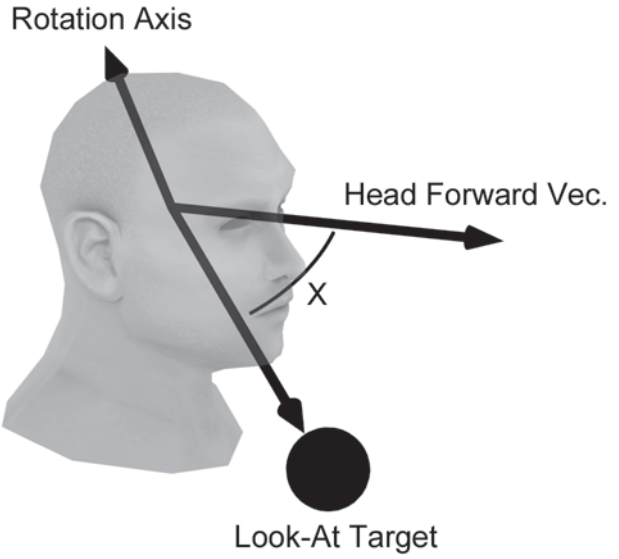


FIGURE 11.6
The shortest arc algorithm, with X being the rotation angle you need to calculate.

The head forward vector is calculated in the initialization of the `InverseKinematics` class. The target you already know; all you need to do is calculate the normalized vector to the target in bone space (since the head forward vector is in bone space). Calculate the cross product of these two vectors (head forward and target vector) and use that as the rotation axis. Then the angle is calculated and cap'd to the max rotation angle and used to create the new rotation matrix (this is the matrix that will turn the head to face the target). Finally update the combined transformation matrix with the new rotation matrix and be sure to update any child bones of the head bone as well using the `SkinnedMesh::UpdateMatrices()` function.

Two-Joint Inverse Kinematics

Now I'll show you how to attack the Two-Joint "Reach" IK problem. To solve this problem easier, you must take the information you know about people in general and put it to good use. For example, in games the elbow joint is treated like a hinge joint with only one degree of freedom (1-DoF), while the shoulder joint is treated like a ball joint (3-DoF).

The fact that you treat the elbow (or knee) joint as a hinge makes this a whole lot simpler. You know that the arm can be fully extended, completely bent, or something in between. So, in other words, you know that the angle between the upper and lower arm has to be between 0 and 180 degrees. This in turn makes it pretty easy for you to calculate the reach of an arm when you know the length of the upper and lower arm. Consider Figure 11.7, for example.

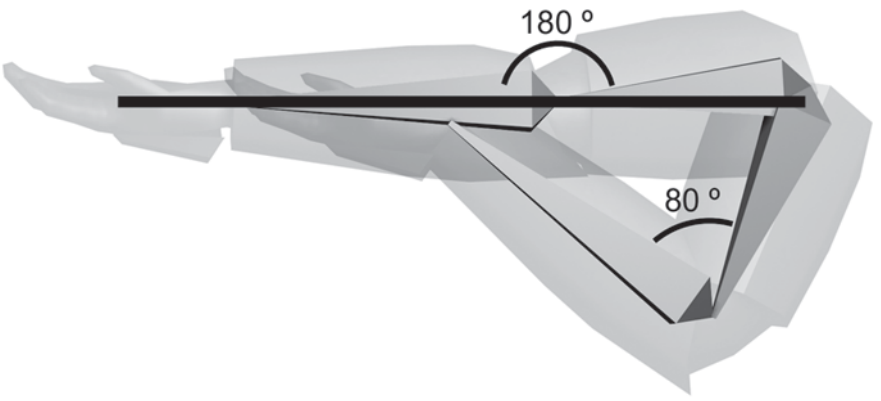


FIGURE 11.7
Within an arm's reach?

The black line in Figure 11.7 defines all the points that this arm can reach, assuming that the elbow joint can bend from 0 to 180 degrees. Let's say that you're trying to make your character reach a certain point with his arm. Your first task is to figure out the angle of the elbow joint given the distance to the target. Using the Law of Cosines, this becomes a pretty straightforward task, since you know the length of all sides of the triangle. The formula for the Law of Cosines is:

$$C^2 = A^2 + B^2 - 2AB\cos(x)$$

Trivia*: You might recognize part of the Law of Cosines as the Pythagorean Theorem. Actually, the Pythagorean Theorem is a special case of the Law of Cosines where the angle x is 90 degrees. Since the cosine for 90 degrees is zero, the term -2ABcos(x) can be removed.*

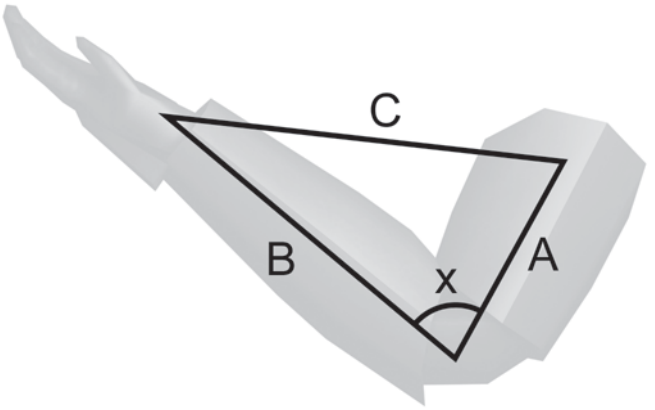


FIGURE 11.8
The Law of Cosines.

Figure 11.7 shows the Law of Cosines applied to the elbow problem.

In Figure 11.8, **C** is known because it is the length from the shoulder to the IK target. **A** and **B** are also known because they are simply the length of the upper and lower arm. So to solve the angle *x*, you just need to reorganize the Law of Cosines as follows:

$$x = a\cos\left(\frac{A^2 + B^2 - C^2}{2AB}\right)$$

First you have to bend the elbow to the angle that gives you the right "length." Then you just rotate the shoulder (a ball joint, remember?) using the same simple Look-At IK approach covered in the previous example. The `ApplyArmIK()` function has been added to the `InverseKinematics` class to do all this:

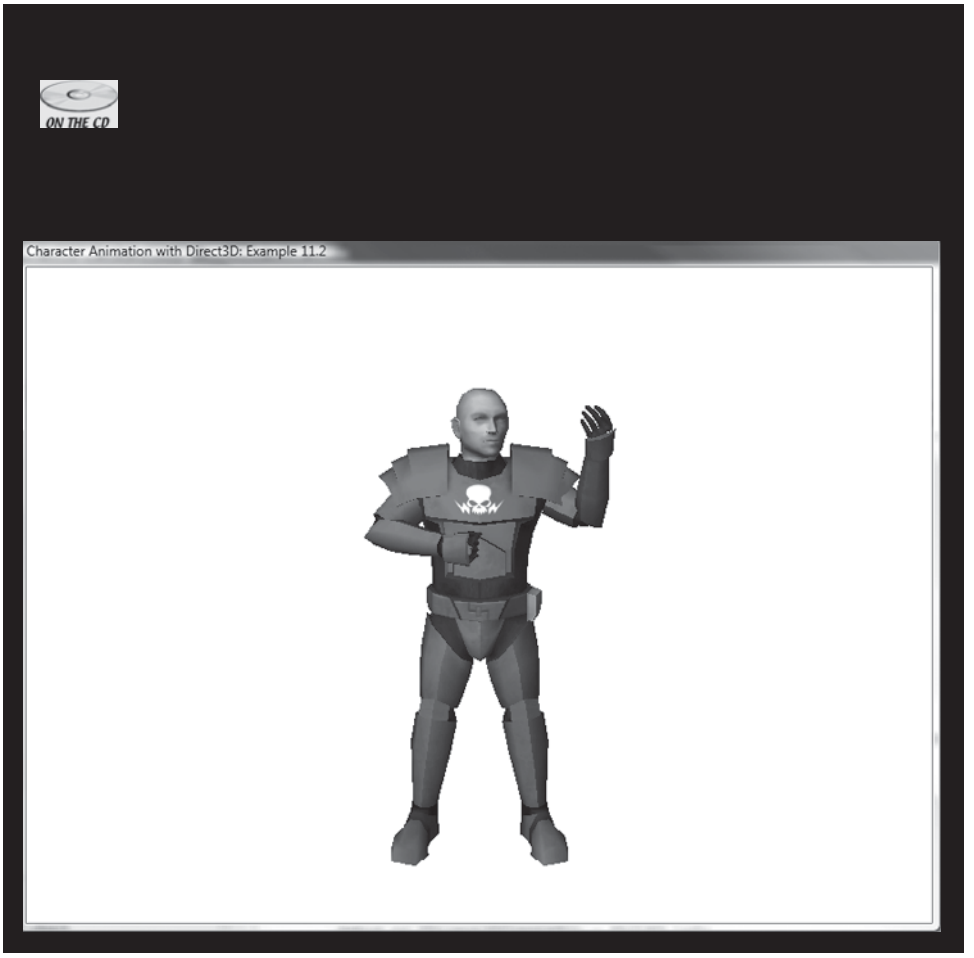
```
void InverseKinematics::ApplyArmIK(D3DXVECTOR3 &hingeAxis,
D3DXVECTOR3 &target)
{
    // Set up some vectors and positions
    D3DXVECTOR3 startPosition = D3DXVECTOR3(
                  m_pShoulderBone->CombinedTransformationMatrix._41,
                  m_pShoulderBone->CombinedTransformationMatrix._42,
                  m_pShoulderBone->CombinedTransformationMatrix._43);
    D3DXVECTOR3 jointPosition = D3DXVECTOR3(
                  m_pElbowBone->CombinedTransformationMatrix._41,
                  m_pElbowBone->CombinedTransformationMatrix._42,
                  m_pElbowBone->CombinedTransformationMatrix._43);
```

```
D3DXVECTOR3 endPosition = D3DXVECTOR3(
              m_pHandBone->CombinedTransformationMatrix._41,
              m_pHandBone->CombinedTransformationMatrix._42,
              m_pHandBone->CombinedTransformationMatrix._43);

D3DXVECTOR3 startToTarget = target - startPosition;
D3DXVECTOR3 startToJoint = jointPosition - startPosition;
D3DXVECTOR3 jointToEnd = endPosition - jointPosition;

float distStartToTarget = D3DXVec3Length(&startToTarget);
float distStartToJoint = D3DXVec3Length(&startToJoint);
float distJointToEnd = D3DXVec3Length(&jointToEnd);

// Calculate joint bone rotation
// Calculate current angle and wanted angle
float wantedJointAngle = 0.0f;

if(distStartToTarget >= distStartToJoint + distJointToEnd)
{
    // Target out of reach
    wantedJointAngle = D3DXToRadian(180.0f);
}
else
{
    //Calculate wanted joint angle (using the Law of Cosines)
    float cosAngle = (distStartToJoint * distStartToJoint +
                      distJointToEnd * distJointToEnd –
                      distStartToTarget * distStartToTarget) /
                      (2.0f * distStartToJoint * distJointToEnd);
    wantedJointAngle = acosf(cosAngle);
}

//Normalize vectors
D3DXVECTOR3 nmlStartToJoint = startToJoint;
D3DXVECTOR3 nmlJointToEnd = jointToEnd;
D3DXVec3Normalize(&nmlStartToJoint, &nmlStartToJoint);
D3DXVec3Normalize(&nmlJointToEnd, &nmlJointToEnd);

//Calculate the current joint angle
float currentJointAngle =
          acosf(D3DXVec3Dot(&(-nmlStartToJoint), &nmlJointToEnd));
```

```
//Calculate rotation matrix
float diffJointAngle = wantedJointAngle - currentJointAngle;
D3DXMATRIX rotation;
D3DXMatrixRotationAxis(&rotation, &hingeAxis, diffJointAngle);

//Apply elbow transformation
m_pElbowBone->TransformationMatrix = rotation *
                   m_pElbowBone->TransformationMatrix;

//Now the elbow "bending" has been done. Next you just
//need to rotate the shoulder using the Look-at IK algorithm

//Calcuate new end position
//Calculate this in world position and transform
//it later to start bones local space
D3DXMATRIX tempMatrix;
tempMatrix = m_pElbowBone->CombinedTransformationMatrix;
tempMatrix._41 = 0.0f;
tempMatrix._42 = 0.0f;
tempMatrix._43 = 0.0f;
tempMatrix._44 = 1.0f;

D3DXVECTOR3 worldHingeAxis;
D3DXVECTOR3 newJointToEnd;
D3DXVec3TransformCoord(&worldHingeAxis, &hingeAxis, &tempMatrix);
D3DXMatrixRotationAxis(&rotation,&worldHingeAxis,diffJointAngle);
D3DXVec3TransformCoord(&newJointToEnd, &jointToEnd, &rotation);

D3DXVECTOR3 newEndPosition;
D3DXVec3Add(&newEndPosition, &newJointToEnd, &jointPosition);

// Calculate start bone rotation
D3DXMATRIX mtxToLocal;
D3DXMatrixInverse(&mtxToLocal, NULL,
                  &m_pShoulderBone->CombinedTransformationMatrix);

D3DXVECTOR3 localNewEnd;  //Current end point
D3DXVECTOR3 localTarget;  //IK target in local space

D3DXVec3TransformCoord(&localNewEnd,&newEndPosition,&mtxToLocal);
D3DXVec3TransformCoord(&localTarget, &target, &mtxToLocal);
D3DXVec3Normalize(&localNewEnd, &localNewEnd);
D3DXVec3Normalize(&localTarget, &localTarget);
```

```
    D3DXVECTOR3 localAxis;
    D3DXVec3Cross(&localAxis, &localNewEnd, &localTarget);
    if(D3DXVec3Length(&localAxis) == 0.0f)
        return;

    D3DXVec3Normalize(&localAxis, &localAxis);
    float localAngle = acosf(D3DXVec3Dot(&localNewEnd, &localTarget));

    // Apply the rotation that makes the bone turn
    D3DXMatrixRotationAxis(&rotation, &localAxis, localAngle);
    m_pShoulderBone->CombinedTransformationMatrix = rotation *
                m_pShoulderBone->CombinedTransformationMatrix;
    m_pShoulderBone->TransformationMatrix = rotation *
                m_pShoulderBone->TransformationMatrix;

    // Update matrices of child bones.
    if(m_pShoulderBone->pFrameFirstChild)
        m_pSkinnedMesh->UpdateMatrices(
             (BONE*)m_pShoulderBone->pFrameFirstChild,
             &m_pShoulderBone->CombinedTransformationMatrix);
}
```

There! This humongous piece of code implements the concept of Two-Joint IK as explained earlier. As you can see in this function we apply any rotation of the joints both to the transformation matrix and the combined transformation matrix of the bone. This is because the `SkinnedMesh` class recalculates the combined transformation matrix whenever the `UpdateMatrices()` function is called. So if you haven't applied the IK rotation to both matrices it would be lost when the `UpdateMatrices()` function is called.



Conclusions

This chapter covered the basics of inverse kinematics (IK) and explained that as a general problem it is quite tough to solve (even though there are quite a few approaches to doing so). I covered two specific IK applications for character animation: Look-At and Two-Joint "Reach" IK. The Two-Joint IK can also be used for placing legs on uneven terrain, making a character reach for a game-world object, and much more.

You would also need IK to make a character hold an object (such as a staff, for example) with both hands. This could, of course, be done with normal keyframe animation as well, but it then often results in one hand not "holding on" perfectly and sometimes floating through the staff (due to interpolation between keyframes).

Hopefully this chapter served as a good IK primer for you to start implementing your own "hands-on" characters.

This chapter essentially wraps up the many individual parts of character animation in this book.

Chapter 11 Exercises

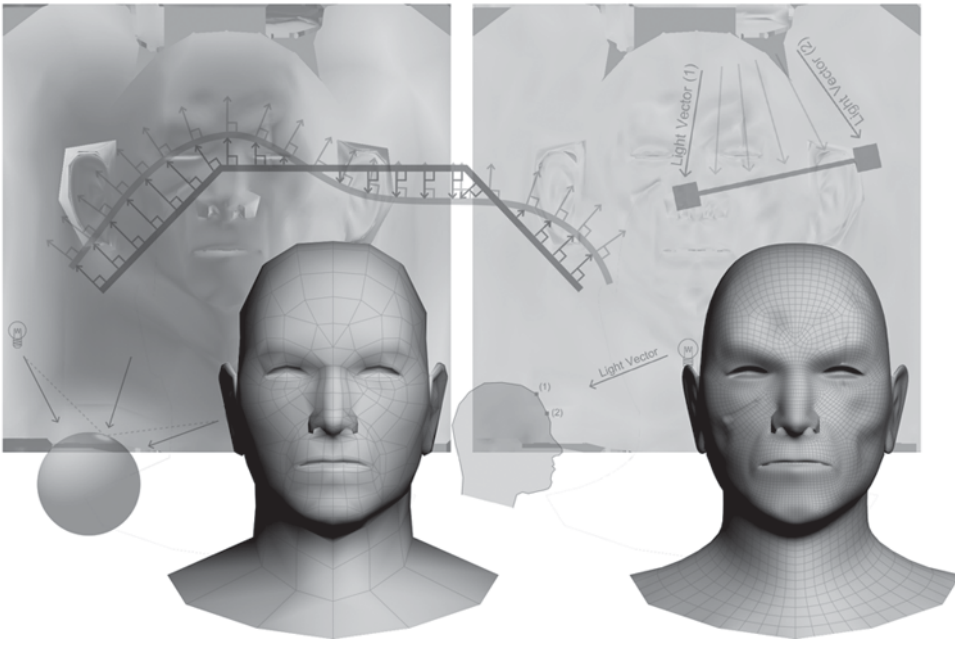
- Add weights to the IK functions enabling you to blend between keyframed animation and IK animation.
- A good next step for you would be to combine a keyframed animation such as opening a door with IK. As the animation is in the state of holding the door handle, blend in the Two-Joint IK with the door handle as the IK target.
- The soldier is holding the rifle with two hands. Glue the other hand (the one that is not the parent of the rifle) to it using IK.
- Implement IK for the legs and make the character walk on uneven terrain.
- Implement aiming for the soldier.

Further Reading

[Melax00] Melax, Stan, "The Shortest Arc Quaternion," *Game Programming Gems.* Charles River Media, 2000.

This page intentionally left blank

12 Wrinkle Maps



I'll admit that this chapter is a bit of a tangent and it won't involve much animation code. This chapter will cover the fairly recent invention of wrinkle maps. In order to make your future characters meet the high expectations of the average gamer out there, you need to know, at the very least, how to create and apply standard normal maps to your characters. Wrinkle maps take the concept of normal maps one step further and add wrinkles to your characters as they talk, smile, or frown, etc. Albeit this is a pretty subtle effect, it still adds that extra little thing missing to make your character seem more alive.

Before you get in contact with the wrinkle maps you need to have a solid understanding of how the more basic normal mapping technique works. Even though normal mapping is a very common technique in today's games, it is surprisingly hard to find good (i.e., approachable) tutorials and information about this online (again, I'm talking about the programming side of normal maps, there's plenty of resources about the art side of this topic). I'm hoping this chapter will fill a little bit of this gap.

Normal mapping is a bump mapping technique—in other words, it can be used for making flat surfaces appear "bumpy." Several programs make use of the term *bump map*, which in most cases takes the form of a grayscale height map. As an object is rendered in one of these programs, a pixel is sampled from the height map (using the UV coordinates of the object) and used to offset the surface normal. This in turn results in a variation of the amount of light this pixel receives. Normal mapping is just one of the possible ways of doing this in real time (and is also currently the *de facto* standard used in the games industry). Toward the end of the chapter I'll also show you how to add specular lighting to your lighting calculations (something that again adds a lot of realism to the end result).

In this chapter you will learn the basics of normal mapping and how to implement the more advanced wrinkle maps:

- Introduction to normal maps
- How to create normal maps
- How to convert your geometry to accommodate normal mapping
- The real-time shader code needed for rendering
- Specular lighting
- Wrinkle maps

Introduction to Normal Mapping

So far in the examples, the Soldier character has been lit by a single light. The lighting calculation has thus far been carried out in the vertex shader, which is commonly known as vertex lighting. Normal mapping, on the other hand, is a form of pixel lighting, where the lighting calculation is done on a pixel-by-pixel level instead of the coarser vertex level.

How much the light affects a single vertex on the character (how lit it is) has been determined previously by the vertex normal. Quite simply, if the normal faces the light source, the vertex is brightly lit; otherwise it is dark. On a triangle level, this

means each triangle is affected by three vertices and their normals. This also means that for large triangles there's a lot of surface that shares relatively little lighting information. Figure 12.1 demonstrates the problem with vertex-based lighting:

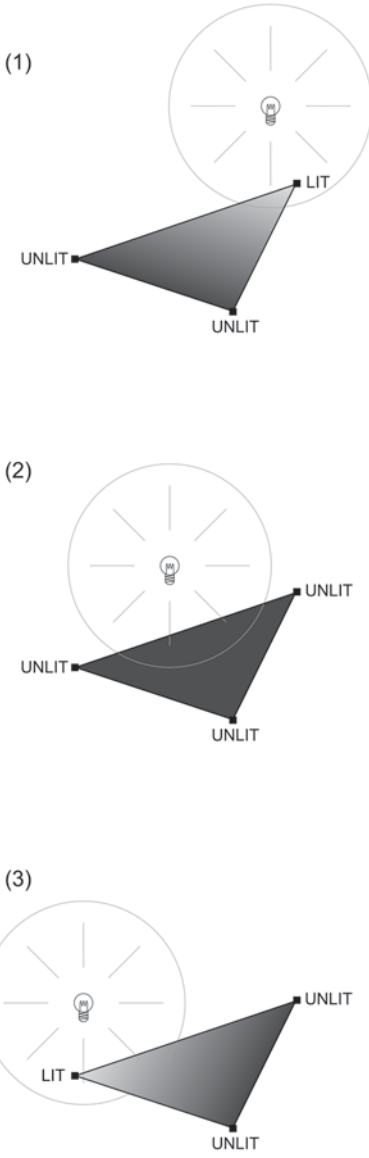


FIGURE 12.1
The problem with vertex-based lighting.

As you can see in Figure 12.1, the concept of vertex lighting can become a problem in areas where triangles are sparse. As the light moves over the triangle, it becomes apparent what the downside of vertex-based lighting is. In the middle image the light is straight above the triangle, but since none of the triangle's vertices are lit by the light, the entire triangle is rendered as dark, or unlit. One common way to fight this problem is, of course, to add more triangles and subdivide areas that could be otherwise be modeled using fewer triangles.

People still use vertex lighting wherever they can get away with it. This is because any given lighting scheme usually runs faster in a vertex shader compared to a pixel shader, since you usually deal with fewer vertices than you do pixels. (The exception of this rule is, of course, when objects are far away from the camera, in which case some form of level of detail (LOD) scheme is used.) So in the case of character rendering, when you increase the complexity (add more triangles) to increase the lighting accuracy, you're also getting the overhead of skinning calculations performed on each vertex, etc.

So to increase the complexity of a character without adding more triangles, you must perform the lighting calculations on a pixel level rather than a vertex level. This is where the normal maps come into the picture.

What Are Normal Maps?

The clue is in the name. A normal map stores a two-dimensional lookup table (or map) of normals. In practice this takes the form of a texture, which in today's shader technology can be uploaded and used in real time by the GPU. The technique we use today in real-time applications, such as games, were first introduced in 1998 by Cignoni *et al.* in the paper "A general method for recovering attribute values on simplified meshes." This was a method of making a low-polygon version look similar to a high-polygon version of the same object.

It's quite easy to understand the concept of a height map that is grayscale, where white (255) means a "high place," and black (0) means a "low place." Height maps have one channel to encode this information. Normal maps, on the other hand, have three channels (R, G, and B) that encode the X, Y, and Z value of a normal. This means that to get the normal at a certain pixel, we can just sample the RGB values from the normal map, transform it to X, Y, and Z, and then perform the lighting calculation based on this sampled normal instead of the normals from the vertices.

Generally speaking, there are two types of normal maps. Either a normal map is encoded in *object space* or in *tangent space*. If the normal map stores normals encoded in object space, it means that the normals are facing the direction that they do in the game world. If the normals are stored in tangent space, the normals are stored relative to the surface normal of the object that they describe. Figure 12.2 attempts to show this somewhat fuzzy concept.

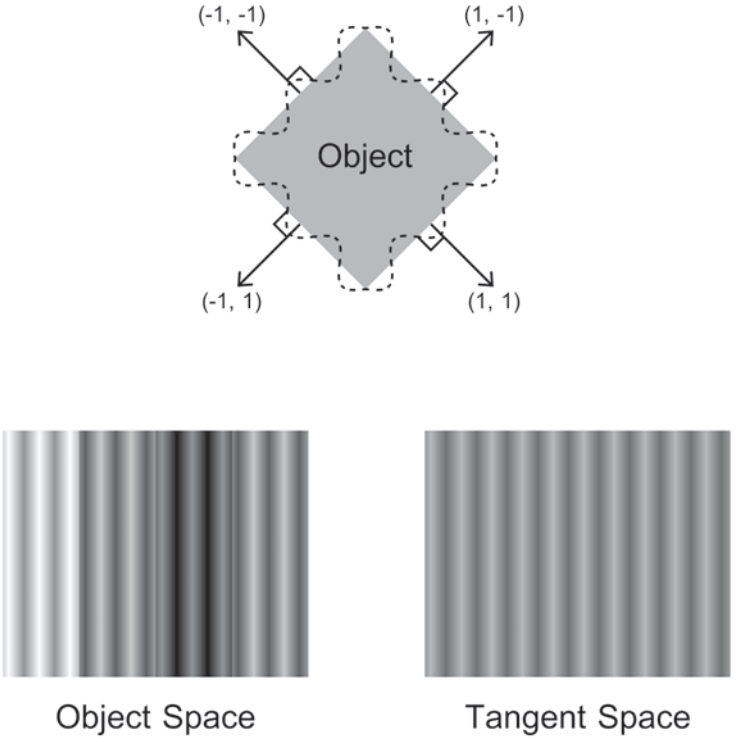


FIGURE 12.2
An object-space normal map compared with a tangent-space normal map.

This picture is not really mathematically correct since it would need two channels for the normals (X and Y). Instead, I've tried to illustrate the point using only grayscale, which I'm afraid messes up the point a bit. I recommend that you do an image search on Google for an "object-space normal map" and a "tangent-space normal map" to see the real difference. The rainbow-colored one will be the object-space normal map, whereas the mostly purplish one will be your more common tangent-space normal map.

As Figure 12.2 hopefully conveys, you should be able to tell the fundamental difference between object- and tangent-space normal maps. The gray box marked "Object" shows the geometry to which we are about to apply the normal map. The wavy dotted line shows the "complex" geometry, or the normals, we want to apply to the normal map.

Since the tangent-space normal map is calculated as an offset of the surface normal, it remains largely uniform in color (depicted in the image as gray), where

the object-space normal map uses the entire range of colors (from white to black in this example). Figure 12.3 shows a real comparison of the appearances of a normal map encoded in object space and a normal map encoded in tangent space.

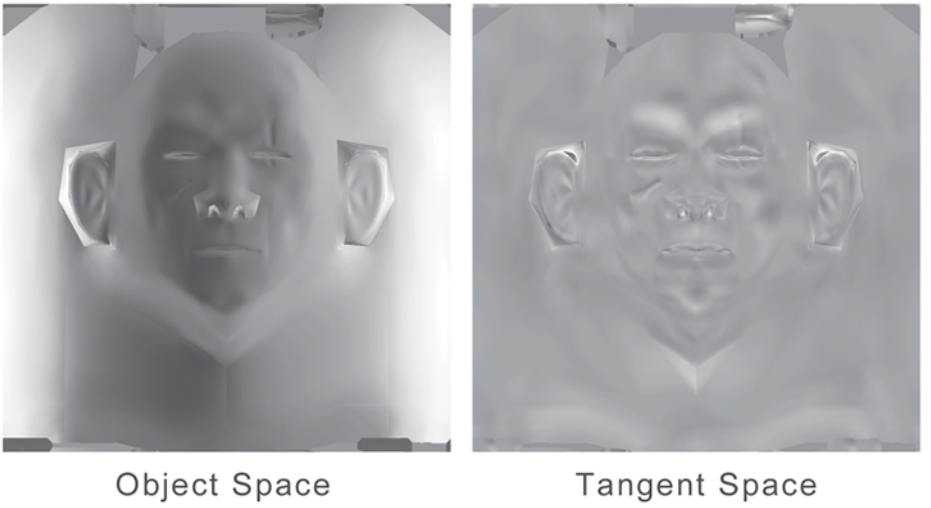


FIGURE 12.3
Real object space vs. tangent space.

Even though Figure 12.3 is in black and white, you should be able to see the difference in the two pictures. However, you'll also find these two normal maps in full color on the accompanying CD-ROM in the Resources folder of Example 12.1.

So what are the differences, you may ask, between these two ways of encoding normal maps, except the fact that they look different? Well, actually the differences are quite significant. Object-space normal maps have the world transformation baked in to the normal map, which means that the pixel shader, in some cases, can skip a few matrix transformations. An example of when it can be a good idea to use object-space normal maps is when you generate the normal maps yourself on-the-fly for static geometry. Then, you might as well bake in the final transformation of the normal in the texture since you know this won't change. The major disadvantage of object-space normal maps is that they are tied to the geometry, which means you can't reuse the normal map over tiled surfaces. Imagine, for example, that you have a brick wall with a tiled diffuse texture. You wouldn't be able to tile this object-space normal map over the wall without getting lighting artifacts. Luckily, tangent-space normal maps don't have this restriction, because with tangent-space normal maps, the final normal is calculated on-the-fly (instead of at the normal map creation time

as is the case with object-space normal maps). So with characters that will both move around in the world and deform (due to skinning or morphing), it becomes clear that tangent-space normal maps are the way to go. So for the rest of this chapter I will focus only on tangent-space normal maps and from here on just refer to them as normal maps.

Encoding Normals as Color

Let's take a closer look at how these normal maps actually store the information in a texture. Remember that we need to encode the X, Y, and Z component of a normal in each pixel. When you store the normals in a texture, you can make the assumption that the normals are unit vectors (i.e., they have a length of 1). There are, of course, schemes that work differently and may also store an additional height value using the Alpha channel, for example. This is, however, outside the scope of this chapter, and I'll focus only on your run-of-the-mill normal maps.

Since the component of a unit vector can range from –1 to 1, and a color component can range from 0 to 255, you have the small problem of converting between these two ranges. Mathematically, this isn't much of a challenge and it can be accomplished in the following manner:

$$R = ((X*0.5)+0.5)*255$$
$$G = ((Y*0.5)+0.5)*255$$
$$B = ((Z*0.5)+0.5)*255$$

That's how simple it is to encode a normal as a single RGB pixel. Then, in the pixel shader, we need to perform the opposite transformation, which is just as simple:

$$X = (R*2.0)-1.0$$
$$Y = (G*2.0)-1.0$$
$$Z = (B*2.0)-1.0$$

This may look a little bit weird since you're expecting the R, G, and B values to be byte values. But since you sample them from a texture, the pixel shader automatically converts the color bytes into float values (which are in the range of 0 to 1 for each color channel). This means that a gray color value of 128 will be sampled from a texture and then used as a float value of 0.5f in the pixel shader.

In a pixel shader, these three lines can be conveniently baked together into the following line:

```
float3 normal = 2.0f * tex2D(NormalSampler, IN.tex0).rgb - 1.0f;
```

In this line of code, a single pixel is sampled from the normal map using the texture coordinates of the model, and then decoded back into a normal. So far the theory, if you will, has been pretty easy to follow and not too advanced. But I'm afraid that this is where the easy part ends. Next up is how to convert the incoming vector from the light source to the coordinate system of the normal map.

Putting the Normal Map to Use

In normal vertex lighting you have two vectors: the normal of the vertex and the direction of the light. To calculate the amount of light the vertex receives, you convert the vertex normal from object space into world space (so that the normal and the light direction are in the same coordinate space). After that you can happily take the dot product of these to vector and use it as a measure of how lit the vertex is. In the case of the per-pixel lighting using the normal mapping scheme, you have x amount of normals per triangle, and one light direction as before. Now instead of transforming all of these surface normals into the same space as the light, we can take the lonely light direction vector and transform it into the same coordinate space as the normal map normals. This coordinate space is known as *tangent space* (hence the name tangent-space normal maps).

So, in order for you to transform a coordinate (be it a direction, position, or whatever) from world space into tangent space, you will need a *tangent-space matrix*. This matrix works just like any of the other transformation matrices; it converts between two different coordinate systems. Take the projection matrix, for example. It converts a 3D image from view space into a flat image in screen space. Figure 12.4 shows the tangent space.

Any given vertex has a tangent space as defined in Figure 12.4. The normal of the vertex that you're already familiar with points out from the triangle. The tangent and the binormal, on the other hand, are new concepts. The tangent and the binormal both lie on the plane of the triangle. The triangle is also UV mapped to the texture (be it a diffuse map or a normal map). So what the tangent space actually describes is a form of 3D texture space.

TRIVIA: Here's some semi-useless knowledge for you. It is actually incorrect to talk about binormals in this context. The mathematically correct term is actually bitangent*! However, people have been using the term binormal since no one knows when. This is loosely because there can be only one normal per surface, but there can be infinite amount of tangents on the surface. The term "bi" means two or "second one," which is why it is incorrect to be talking about a second normal in this case.*

You can read more about this (and other interesting things) at Tom Forsyth's blog:

http://home.comcast.net/~tom_forsyth/blog.wiki.html

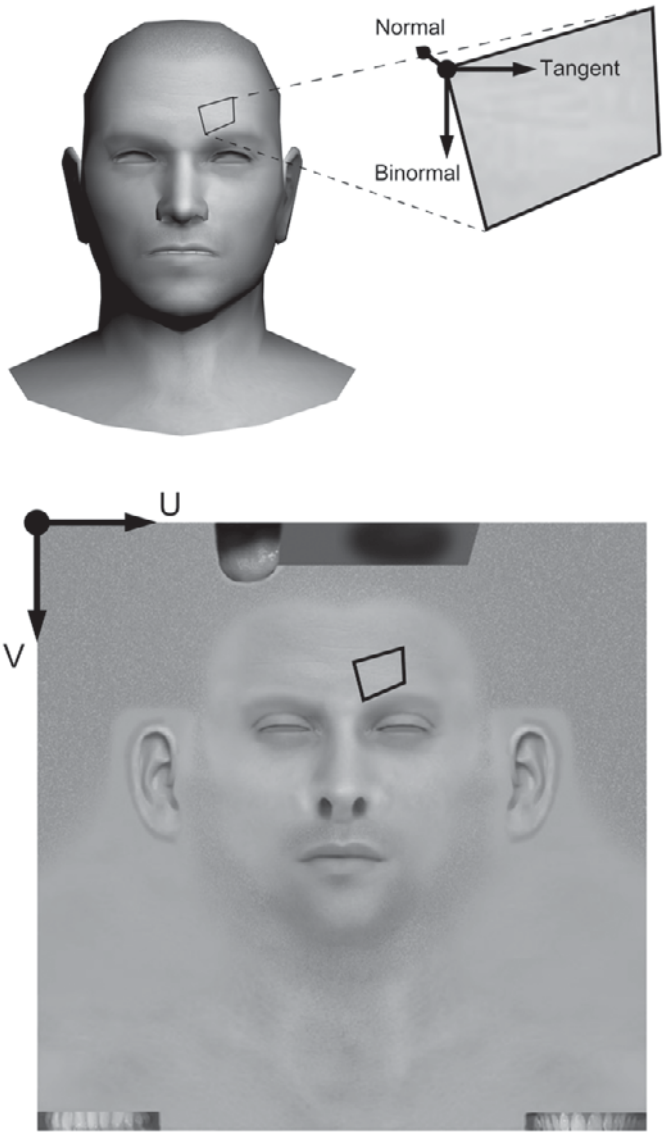


FIGURE 12.4
The tangent space.

Now, the cool thing is that you take a vector in world space and transform it to this 3D texture space (i.e., the tangent space) for each of the vertices of a triangle. When this data is passed to the pixel shader, it is interpolated, giving you a correct world-to-pixel vector for each of the pixels. This means that the light vector is in the

same coordinate system as the normal in a normal map, which in turn means that it is okay to perform the light calculation. Figure 12.5 shows a 2D example of this process in action.

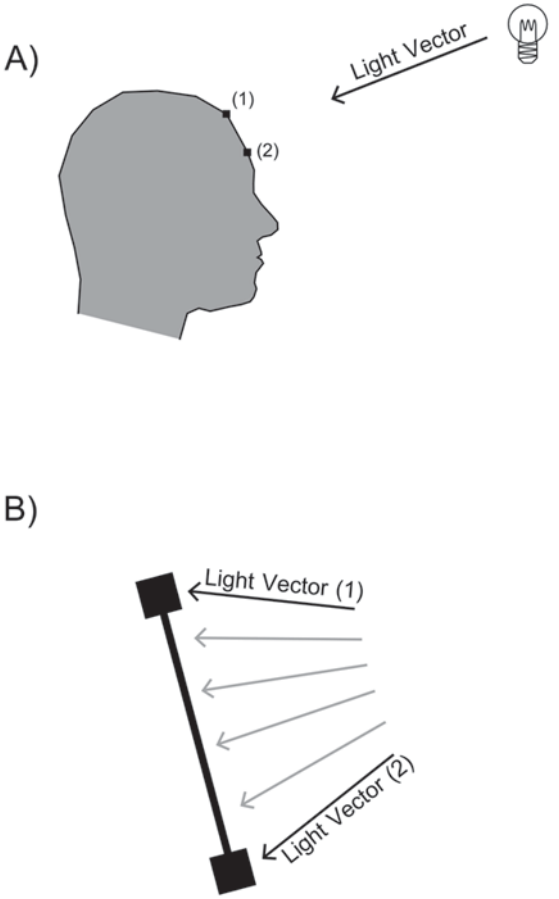


FIGURE 12.5
Transforming a world light direction to tangent space.

In Figure 12.5 (A) you see the incoming light hitting two vertices, (1) and (2), marked as black squares. The incoming light vector is transformed into tangent space, marked by the two corresponding black lines in Figure 12.5 (B). These transformed light vectors are then sent to the pixel shader, which interpolates the light vectors for each pixel. This interpolated light vector can then be compared against the normal stored in the normal map (since they are now in the same coordinate space). This process should be a bit simpler to understand in 2D, but the exact same thing happens in 3D as well.

The TBN-Matrix

The TBN-Matrix stands for Tangent-Binormal-Normal Matrix, which are the basic components of tangent space. I won't go into all the gruesome details behind this 3 x 3 matrix; suffice it to say that it converts between the world space and the tangent space. You would construct your TBN-Matrix in the following fashion:

$$TBN = \begin{bmatrix} Tangent.x & Binormal.x & Normal.x \\ Tangent.y & Binormal.y & Normal.y \\ Tangent.z & Binormal.z & Normal.z \end{bmatrix}$$

After this you can transform any point in world space (v_w) to a vector in tangent space (v_t) like this:

$$v_t = v_w * TBN$$

In shader code, this all looks like this:

```
//Get the position of the vertex in the world
float4 posWorld = mul(vertexPosition, matW);

//Get vertex to light direction
float3 light = normalize(lightPos - posWorld);

//Create the TBN-Matrix
float3x3 TBNMatrix = float3x3(vertexTangent,
                              vertexBinormal,
                              vertexNormal);

//Setting the lightVector
lightVec = mul(TBNMatrix, light);
```

The `lightVec` vector then gets sent to the pixel shader and is interpolated as shown in Figure 12.5. The normal you already have for all the vertices. The next problem to solve is how to calculate the tangent and the binormal for all vertices.

Converting a Mesh to Support Normal Mapping

So far, we've dealt with vertices having position, normals, texture coordinates, bone indices, and bone weights. Next, we need to be able to add a tangent component and a binormal component. As you may remember, a vertex can be defined with the Fixed Vertex Format (FVF), but for more advanced things you need to create an

array of D3DVERTEXELEMENT9 objects. With these elements you control every aspect of how the bitstream from a mesh is interpreted.

As a quick recap, the following function shows you how to get the vertex declaration from a mesh and access the different elements in it (very useful for debugging purposes).

```
void PrintMeshDeclaration(ID3DXMesh* pMesh)
{
    //Get vertex declaration
    D3DVERTEXELEMENT9 decl[MAX_FVF_DECL_SIZE];
    pMesh->GetDeclaration(decl);

    //Loop through valid elements
    for(int i=0; i<MAX_FVF_DECL_SIZE; i++)
    {
        if(decl[i].Type != D3DDECLTYPE_UNUSED)
        {
            g_debug << "Offset: " << (int)decl[i].Offset
                    << ", Type: " << (int)decl[i].Type
                    << ", Usage: " << (int)decl[i].Usage
                    << "\n";
        }
        else break;
    }
}
```

This function prints the offset, type, and usage of all active elements in a vertex declaration. Sometimes, when you are building your own vertex formats, it can be very useful to know at what offset a certain element is stored (and what type it is); especially when you deal with different meshes from different sources and or formats.

Remember that you're already dealing with meshes containing different elements. In the bone hierarchy of the SkinnedMesh class, for example, you have static meshes containing position, normal, and texture coordinates. You also have the skinned meshes there as well, and on top of the position, normal, and texture coordinates, they also contain the bone index and bone weight components.

So we need to be able to add components to any arbitrary vertex declaration. For this purpose I've implemented the AddTangentBinormal() function. This function is not much different from the PrintMeshDeclaration() function. It takes a mesh as input, extracts the current mesh declaration, and adds the tangent and the binormal elements to it. Then, it clones the original mesh by using the newly created vertex declaration. Lastly, it computes the tangents and the binormals for all the vertices in

the mesh using the `D3DXComputeTangentFrame()` function. Once this has been done it releases the old mesh and replaces it with the newly created mesh containing valid tangents and binormals:

```
void AddTangentBinormal(ID3DXMesh** pMesh)
{
    //Get vertex declaration from mesh
    D3DVERTEXELEMENT9 decl[MAX_FVF_DECL_SIZE];
     (*pMesh)->GetDeclaration(decl);

    //Find the end index of the declaration
    int index = 0;
    while(decl[index].Type != D3DDECLTYPE_UNUSED)
    {
        index++;
    }

    //Get size of last element (in bytes)
    int size = 0;

    switch(decl[index - 1].Type)
    {
        case D3DDECLTYPE_FLOAT1:
            size = 4;
            break;

        case D3DDECLTYPE_FLOAT2:
            size = 8;
            break;

        case D3DDECLTYPE_FLOAT3:
            size = 12;
            break;

        case D3DDECLTYPE_FLOAT4:
            size = 16;
            break;

        case D3DDECLTYPE_D3DCOLOR:
            size = 4;
            break;
```

```
    case D3DDECLTYPE_UBYTE4:
        size = 4;
        break;

    default:
        //Unhandled declaration type
};

//Create tangent element
D3DVERTEXELEMENT9 tangent =
{
    0,
    decl[index - 1].Offset + size,
    D3DDECLTYPE_FLOAT3,
    D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_TANGENT,
    0
};

//Create binormal element
D3DVERTEXELEMENT9 binormal =
{
    0,
    tangent.Offset + 12,
    D3DDECLTYPE_FLOAT3,
    D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_BINORMAL,
    0
};

//End element
D3DVERTEXELEMENT9 endElement = D3DDECL_END();

//Add new elements to the old vertex declaration
decl[index++] = tangent;
decl[index++] = binormal;
decl[index] = endElement;

//Convert mesh to the new vertex declaration
ID3DXMesh* pNewMesh = NULL;
```

```
    if(FAILED((*pMesh)->CloneMesh(
                (*pMesh)->GetOptions(),
                decl,
                g_pDevice,
                &pNewMesh)))
    {
        //Failed to clone mesh
        return;
    }

    //Compute the tangents and binormals
    if(FAILED(D3DXComputeTangentFrame(pNewMesh, NULL)))
    {
        //Failed to compute tangents and binormals for new mesh
        return;
    }

    //Release old mesh
    (*pMesh)->Release();

    //Assign new mesh to the mesh pointer
    *pMesh = pNewMesh;
}
```

As you can see, this function takes a pointer to a pointer to a mesh (or a double pointer). This means that we can actually reassign the pointer being sent in and replace what it is pointing to. Most of the resource-loading and mesh-handling functions in the D3DX library take a double pointer and operate in pretty much the same way as this function. The `AddTangentBinormal()` function very much reminds one of the `ConvertToIndexedBlendedMesh()` function defined in the ID3DXSkinInfo interface. What that function did was to add the bone weights and bone indices elements to a mesh in exactly the same way. It also filled the newly created elements with some sensible information (just like what is done with the `D3DXComputeTangentFrame()` function).

Sometimes you have data stored in a mesh using a certain vertex declaration that you want to change; however, the data is fine as it is and you just want to change the declaration. Well, instead of using the `CloneMesh()` *function to create a copy, you can use the* `UpdateSemantics()` *function in the* `ID3DXBaseMesh` *class for this. So if you want to add new elements to the vertex declaration, use the* `CloneMesh()` *function, but if you just want to re-label an element (for example, switching the tangent and the binormal, or texture coordinate 1 with texture coordinate 2, etc.) use the* `UpdateSemantics()` *function.*

After you've sent whatever mesh you want normal mapped through this function you have a mesh ready to be normal mapped. I won't dive into the math behind tangent and binormal calculations, but if you're interested you can read more about that in [Lengyel01]. Next is the final piece of the puzzle: the shader.

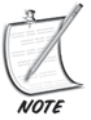
The Normal Mapping Shader

The shader code takes all that theory you've been reading about, as well as the prepared meshes, and outputs something that looks a lot better than what you've seen so far. In this chapter I have implemented normal mapping for the morphing meshes and the `Face` class. You should have little trouble, though, porting it to the skinned mesh shader yourself. After adding the tangent and the binormal to the vertex declaration of the base mesh in the `Face` class, the full vertex declaration of the `Face` class looks like the following:

```
//Face Vertex Format
D3DVERTEXELEMENT9 faceVertexDecl[] =
{
//1st Stream: Base 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},
{0, 32, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                            D3DDECLUSAGE_TANGENT,  0},
{0, 44, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                            D3DDECLUSAGE_BINORMAL, 0},
```

```
    //2nd Stream
    {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
    {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
    {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
    {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()
    };
```

Note the new tangent and binormal elements in the first stream (the base mesh stream).



As an optimization we only add the tangent and binormal elements to the base mesh of the `Face` *class. It would be more correct to add it to all the meshes in the* `Face` *class and then blend these (in the same manner you blend the normals). However, the results are still fine as long as you don't perform deformations of ridiculous proportions.*

Next, you need the input structure to the vertex shader to match the vertex declaration, like this:

```
//Vertex Input
struct VS_INPUT
{
    //Stream 0: Base Mesh
    float4 pos0  : POSITION0;
    float3 norm0 : NORMAL0;
    float2 tex0  : TEXCOORD0;
    float3 tangent  : TANGENT0;
    float3 binormal : BINORMAL0;

    //Stream 1: Morph Target 1
    float4 pos1  : POSITION01;
    float3 norm1 : NORMAL1;

    //Stream 2: Morph Target 2
    float4 pos2  : POSITION2;
    float3 norm2 : NORMAL2;

    //Stream 3: Morph Target 3
    float4 pos3  : POSITION3;
    float3 norm3 : NORMAL3;

    //Stream 4: Morph Target 4
    float4 pos4  : POSITION4;
    float3 norm4 : NORMAL4;
};
```

Nothing surprising here; the new tangent and binormal vectors have been added to stream 0 just like in the vertex declaration. What is new, though, is the `VS_OUTPUT` structure (describing what comes out from the vertex shader and into the pixel shader):

```
//Vertex Output / Pixel Shader Input
struct VS_OUTPUT
{
    float4 position   : POSITION0;
    float2 tex0       : TEXCOORD0;
    float3 lightVec   : TEXCOORD1;
};
```

Instead of the old `shade` float value that we used to send in to the pixel shader, we send in the light vector (in tangent space). This is the vector that gets interpolated (just like any other value you send into the pixel shader), as illustrated in Figure 12.5. Then, to transform information stored in the `VS_INPUT` structure to the `VS_OUTPUT` structure, the following vertex shader performs the morphing and the conversion of the light vector to tangent space:

```
//Vertex Shader
VS_OUTPUT morphNormalMapVS(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    float4 position  = IN.pos0;
    float3 normal = IN.norm0;

    //Blend Position
    position += (IN.pos1 - IN.pos0) * weights.r;
    position += (IN.pos2 - IN.pos0) * weights.g;
    position += (IN.pos3 - IN.pos0) * weights.b;
    position += (IN.pos4 - IN.pos0) * weights.a;

    //Blend Normal
    normal += (IN.norm1 - IN.norm0) * weights.r;
    normal += (IN.norm2 - IN.norm0) * weights.g;
    normal += (IN.norm3 - IN.norm0) * weights.b;
    normal += (IN.norm4 - IN.norm0) * weights.a;

    //Getting the position of the vertex in the world
    float4 posWorld = mul(position, matW);
    OUT.position = mul(posWorld, matVP);

    //Get normal, tangent, and binormal in world space
    normal = normalize(mul(normal, matW));
    float3 tangent = normalize(mul(IN.tangent, matW));
    float3 binormal = normalize(mul(IN.binormal, matW));
```

```
    //Getting vertex -> light vector
    float3 light = normalize(lightPos - posWorld);

    //Calculating the binormal and setting
    //the tangent binormal and normal matrix
    float3x3 TBNMatrix = float3x3(tangent, binormal, normal);

    //Setting the lightVector
    OUT.lightVec = mul(TBNMatrix, light);

    OUT.tex0 = IN.tex0;

    return OUT;
}
```

It is very common that the binormal is actually left out of this whole process and then calculated on-the-fly in the vertex shader. This can end up saving a lot of memory—12 bytes per vertex, in fact. In large projects this can add up to a whole lot. The binormal can then be calculated as a cross-product between the normal and the tangent in the following manner:

```
float3 binormal = normalize(cross(normal, tangent));
```

Once the position and normal of the face has been calculated, the direction from the light source to the vertex is calculated. This is fed into the TBN Matrix, which transforms the light vector to tangent space. This information, together with the texture coordinates (as usual) are stored in the `VS_OUTPUT` structure and sent onward to the pixel shader.

```
//Pixel Shader
float4 morphNormalMapPS(VS_OUTPUT IN) : COLOR0
{
    //Calculate the color and the normal
    float4 color = tex2D(DiffuseSampler, IN.tex0);

    //This is how you uncompress a normal map
    float3 normal = 2.0f * tex2D(NormalSampler, IN.tex0).rgb - 1.0f;

    //Normalize the light
    float3 light = normalize(IN.lightVec);
```

```
    //Set the output color
    float shade = max(saturate(dot(normal, light)), 0.2f);
    return color * shade;
}
```

In the pixel shader, the diffuse color is first sampled from the diffuse map. Then the normal map is sampled using the same texture coordinate. For this pixel, the normal is calculated from the normal map color as described earlier and compared with the light vector sent from the vertex shader. The resulting dot product is then multiplied with the color pixel and drawn onscreen. Figure 12.6 shows a comparison between normal vertex lighting and the more advanced per-pixel normal map lighting scheme.

FIGURE 12.6
Vertex lighting vs. normal mapping.

As you can see, the normal mapped version has a lot more detail compared to the simpler vertex lighting scheme; this despite the fact that both faces have the exact same polygon count. In the normal map, I've added some scars and bumps to the head and tried to make the cheekbones and forehead more pronounced. Finally, here's the code example for this somewhat complex and long chapter.

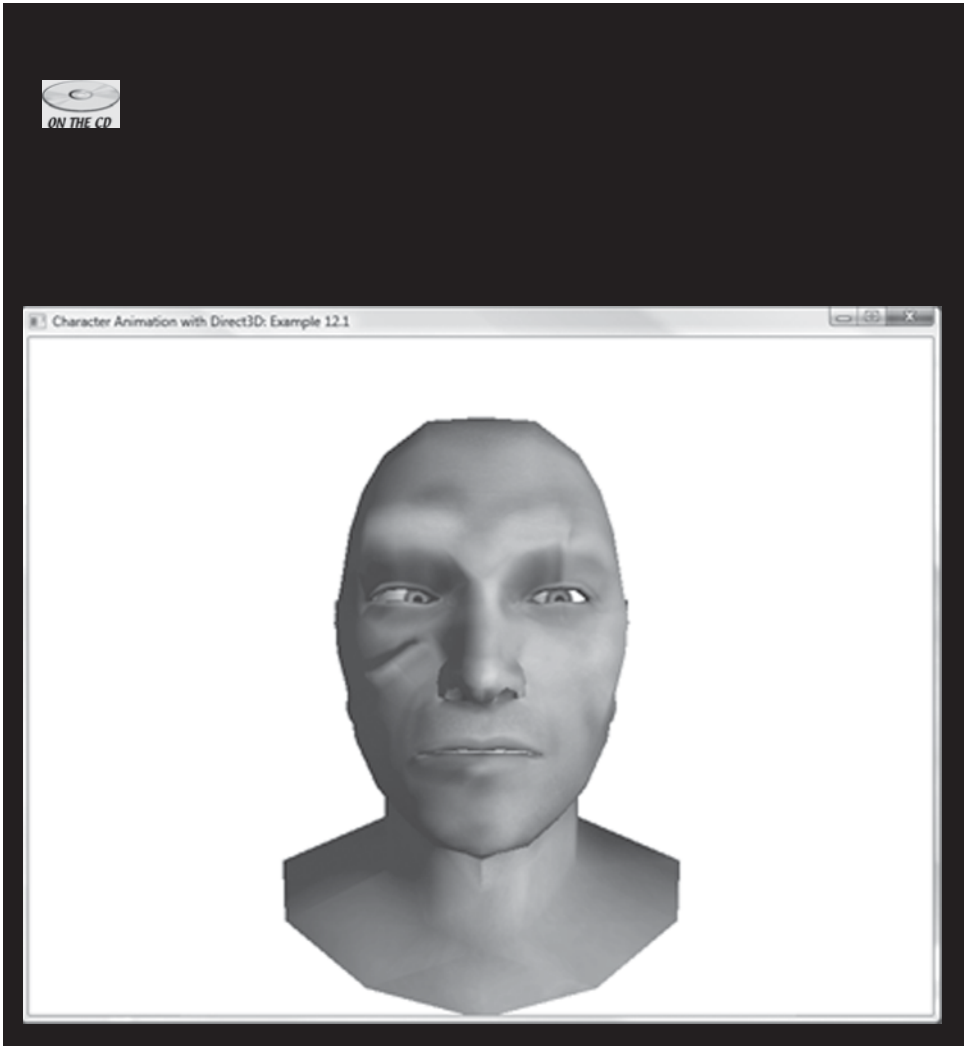


Figure 12.7 shows four snapshots of the example code in action. The light source has been animated to better emphasize the normal map lighting.

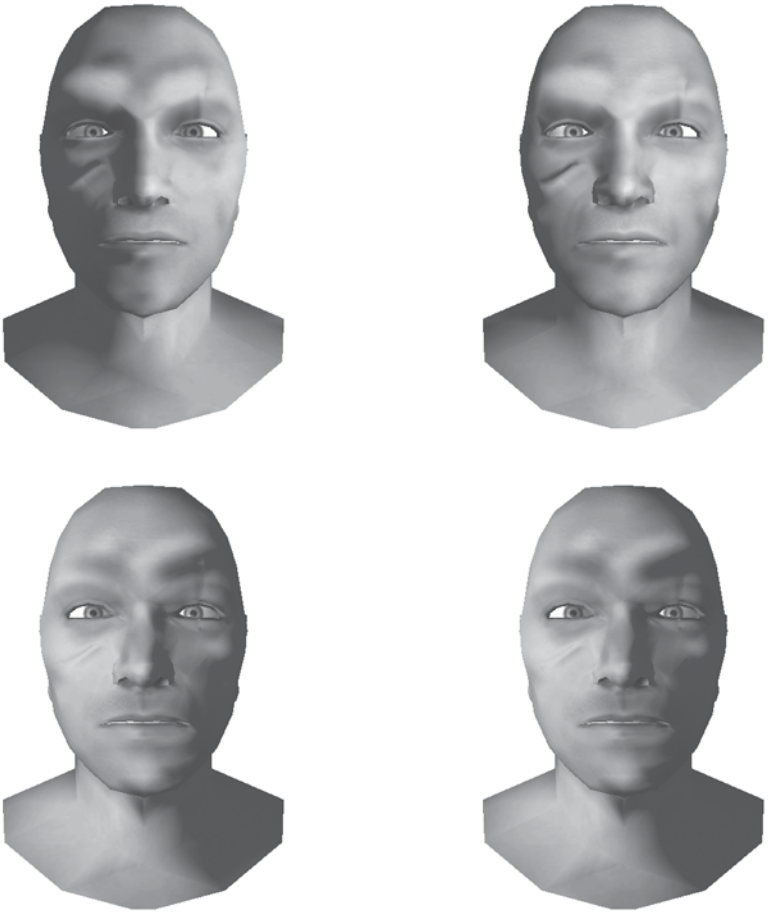


FIGURE 12.7
Normal mapping with animated light source.

Creating Normal Maps

Here's a short section about how normal maps are created—something which in itself is a bit of a science. The process needs two things: the low-polygon mesh you intend to use in the game and a high-polygon mesh having all that extra detail. Figure 12.8 shows the two meshes needed to create a normal map.

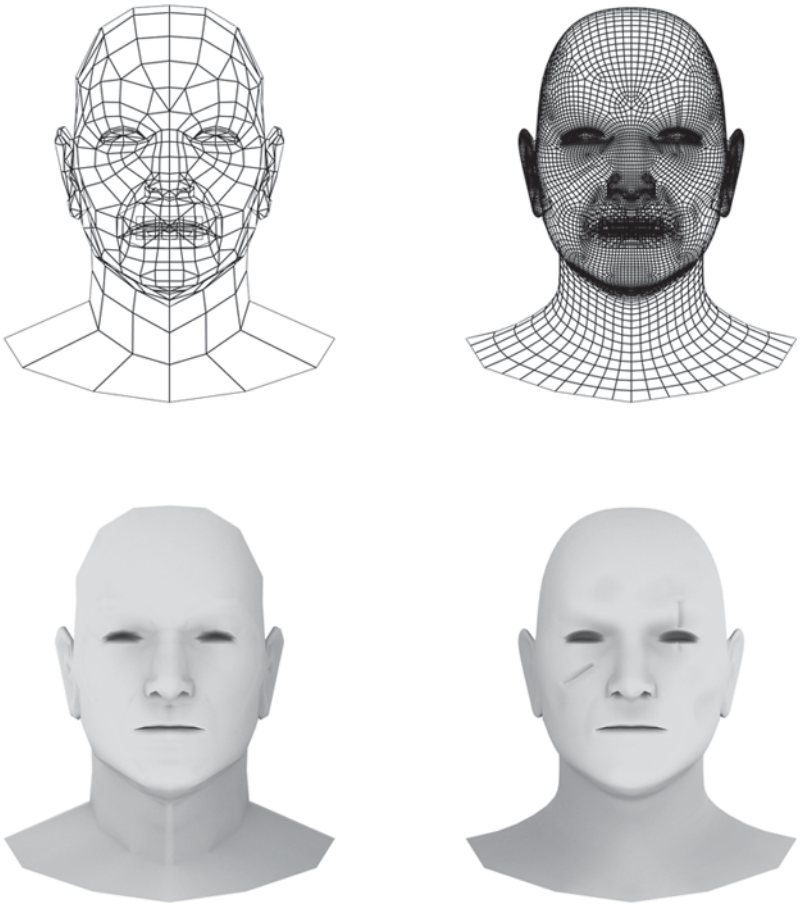


FIGURE 12.8
Meshes needed to create a normal map.

You are already familiar with the low-polygon mesh. It may have a strict polygon limit (and other restrictions) depending on whatever game requirements you may have. The high-polygon mesh, however, has no theoretical upper limit on the amount of polygons, and it can have millions upon millions of triangles (as long as you have a decent enough computer to support it). It doesn't make sense, however, to have more detail in the mesh than can be represented in your normal map. So if you're planning to have a 1024 x 1024 resolution normal map, there is no point in having a high-detail mesh with more detail than can be represented by this normal map.

The low- and high-polygon meshes are first placed at the same location so that they are intersecting. Next, you loop over all the pixels in the normal map, and for each pick the actual position on the low-polygon model using the UV map. Once you have this position you find where the normal of the low-polygon surface intersects the high-polygon model, sample the normal of the high-polygon model instead, and write this value to the normal map (encoded in RGB as explained earlier). Figure 12.9 shows this process in a 2D example.

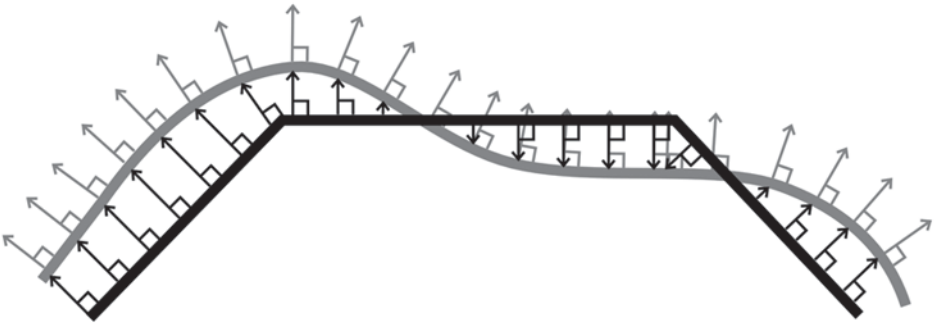


FIGURE 12.9
Calculating normals from low- and high-polygon meshes.

The big black and blocky line in Figure 12.9 represents the low-polygon mesh where the gray smooth line represents the high-polygon model. Each of the small black normals represents one pixel sample point in the normal map. First you can see the sample points (black normals) extend until they hit the high-polygon model, where finally the gray normal is recorded and stored in the normal map. So later in the game, when we render the low-polygon model using the normal map taken from the high-polygon model, we can create the illusion of a much more detailed surface.

However, there are some pitfalls when creating normal maps. The low-polygon mesh needs to be UV mapped, but the high-polygon mesh has no such requirement; it can be pure geometry. Another restriction is that your low-polygon mesh cannot have overlapping UV coordinates when it goes through the normal map creation process. This means that all points on the model must have a unique place on the UV map; otherwise, the program creating the normal map won't know where on the high-polygon model to sample the normal from. Often, artists model only one half of a character and then copy this half, flip the copy, and merge it with the original half, thus producing the full character. In essence this also means that the UV coordinates of both halves are the same, which is a big "no-no" when creating normal maps. So no surfaces using tiled or mirrored UV coordinates.

Just because a model cannot have a mesh with overlapping UV coordinates at the time the normal map is created doesn't mean that it can't have it at runtime. This rule about having shared UV space (either tiled or mirrored UV space) is not really a strict rule. At runtime it is fine to use a tiled normal map (something that is often done for floors, walls, etc.). A mirrored normal map, on the other hand, is more problematic. Think, for example, of a character that has had its left side created as a mirror image of its right side. This means that they use the same UV space in the diffuse and normal map. This in turn means that when you light a pixel from the right shoulder it will work correctly. However, when you light a pixel on the mirrored part of your character, this normal will also be mirrored, which leads to incorrect lighting. With some additional programming, though, you can implement a shader that handles this problem. By storing an additional sign value in each of the vertices of the mesh (depending on whether or not it is a mirrored vertex), you can easily switch between using the left-handed or right-handed coordinate system when sampling the normals from the normal map.

Creating Normal Maps in Practice

That's about all you need to know about how to create normal maps…in theory. In most cases you'll play the role of a programmer and have someone else worry about creating the normal maps for you (most often this falls on the artist's task list). However, you as a programmer still need to know how this is done, since in the end it affects your job as well. In practice, there are a lot of different programs that will create the normal maps for you. Some of the most popular (artist) tools for creating and editing normal maps are:

Pixologic's ZBrush
http://www.pixologic.com

Autodesk's Mudbox
http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=10707763

Both of these programs have free trial versions that you can download and try out. However, normal maps can also be created with the free Normal Mapper tool (including source code) from ATI, which you can download from here:

http://www2.ati.com/developer/NormalMapper-3_2_2.zip

This tool comes with an exporter to both 3D Studio Max and Maya that exports a model to the NMF format, which can then be used by the Normal Mapper tool.

The readme file bundled with the tool explains how to use it and how to install the 3D Studio and Maya exporter plug-ins.

You can also use the free Melody tool from NVidia available from here:

http://developer.nvidia.com/object/melody_home.html

Melody also supports the more common .3ds and .obj formats.

I have also included a max file on the accompanying CD-ROM containing the high- and low-polygon version of the Soldier's face (also together with the exported NMF files). You'll find these files in the "Head Model" folder together with Example 12.1.

There is also a great Photoshop plug-in tool from NVIDIA available here:

http://developer.nvidia.com/object/photoshop_dds_plugins.html

With this tool you can convert bump maps or height maps into normal maps. This is a great way of creating normal maps for flat surfaces such as walls and floors, etc. This tool also allows you to manually edit and then re-normalize normal maps, which is great for small fixes and such.

Specular Highlight

Another tangent (pun intended) before we look at wrinkle maps, is how to implement specular lighting—also known as specular highlight. So far I've only shown you how diffuse lighting works. Now with the normal maps in place it really pays off to also implement a specular lighting model. Specular lighting in real life is actually reflections of the light source on a surface. The shinier a surface is, the more the light source will reflect in it.

A specular highlight is dependent on where the viewer or camera is located in the world. The specular highlight will appear on the model where the surface normal is pointing halfway between the incoming light and the incoming view direction, as shown in Figure 12.10.

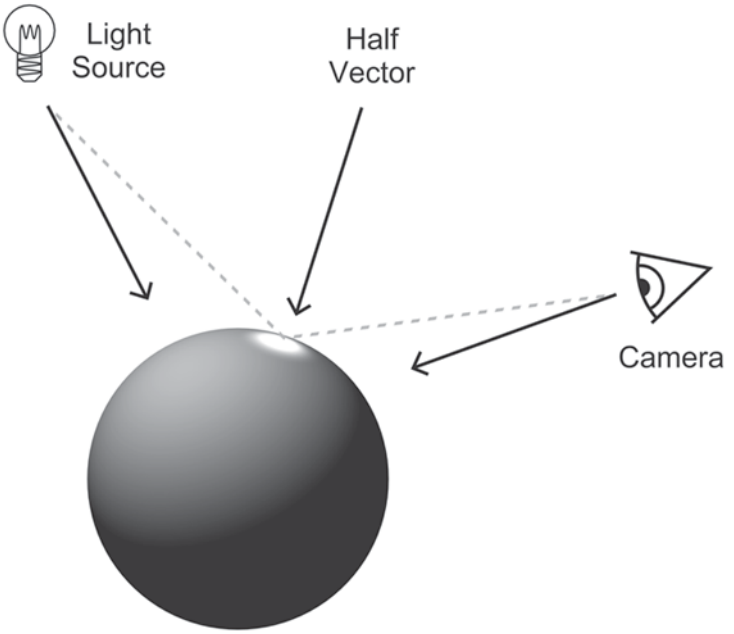


FIGURE 12.10
Specular highlight.

Figure 12.10 shows the halfway vector between the light source and the view direction. Both the position of the light and the camera get sent to the vertex shader, which then calculates the incoming vectors and the halfway vector, which it sends on to the pixel shader. Note that if you are using normal mapping you will also need to convert the halfway vector to tangent space using the TBN Matrix. The following shader code will calculate the halfway vector (the `posWorld` variable denotes the position of the vertex in world coordinates):

```
//Getting light-to-vertex direction
float3 lightDir = normalize(lightPos - posWorld);

//Get camera-to-vertex direction
float3 viewDir = normalize(cameraPos - posWorld);

//Calculate the halfway vector
float3 vHalf = normalize(light + viewDir);
```

Once you have the halfway vector in the pixel shader, you need to determine how much of the light source will be reflected, or, in other words, how large the specular highlight will be. This is actually governed by how perfect the surface is. If the surface is rough, more of the light will scatter, making the highlight larger and duller. On the other hand, if the surface is perfectly smooth it will create a sharp reflection of the light source and have a small but bright specular highlight. Compare, for example, the perfect surface of a bowling ball with the rough surface of human skin. Figure 12.11 shows some different specular highlights.

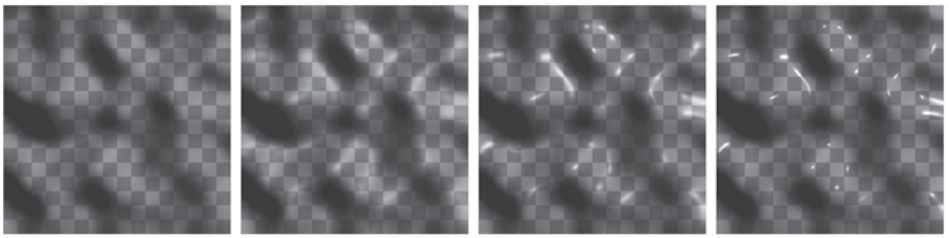


FIGURE 12.11
Specular highlights on different surfaces.

The images in Figure 12.11 show the same surface with an increasing amount of surface smoothness. As you can see from this figure, having specular highlights gives you more information about the object you are looking at. With specular highlights you get a feel of what surface something is made of as well as information about where lights are placed in relation to the object.

So how are these highlights calculated in the pixel shader? The following code snippet shows how the specular highlight is calculated from the halfway vector in the pixel shader:

```
//Get the dot product between the surface normal and the halfway vector
float specular = max(saturate(dot(normal, normalize(lightHalf))), 0.0f);

//Raise the specular value to the power of the shininess value
specular = pow(specular,  shineValue);
```

First we do the usual light calculation using the dot product between the surface normal and the halfway vector. Then we raise this value (which will be in the range of 0 to 1) with the shininess value of the surface. The resulting value we multiply with the specular color and then add to the diffuse color of the pixel. *Voila!* You've implemented specular highlights.

Specular Maps

Different materials have different specular colors. For example, some materials like mirrors or human skin (which has a thin layer of oil) reflect most of the color spectrum in the specular highlights. Other materials, like metals, reflect only the color of the material. The previous section used only one shininess value for the whole object/material. For characters, however, this is usually not enough since you might want to have a different shininess value for different parts of your character, skin, clothes, shoes, armor, etc. Therefore, most game engines these days make use of specular maps. These maps contain the color (and intensity) of the specular highlight. Figure 12.12 shows an example of the specular map used in Example 12.2.

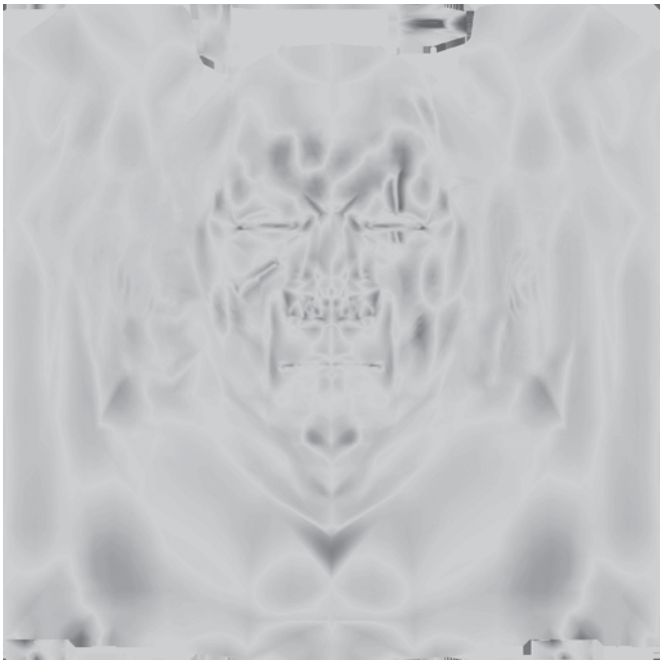


FIGURE 12.12
Specular map example.

This texture is mostly skin colored (you'll find it on the CD-ROM in full color). As a general rule, specular maps have brighter pixels on flat surfaces and dull colors on corners and curved surfaces. There are several tutorials online about how to create specular maps, but again this is something better left to the artists. The following tutorial gives you a good starting point; it covers how to convert a normal map to a specular map using Photoshop:

http://www.modwiki.net/wiki/Start_a_Specular_map_with_a_Normal_map

The following code shows the entire vertex and pixel shader code for rendering a normal-mapped face with a specular map (note that the input and output structures are the same as in the previous example):

```
//Vertex Shader
VS_OUTPUT morphNormalMapVS(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    float4 position  = IN.pos0;
    float3 normal = IN.norm0;

    //Blend Position
    position += (IN.pos1 - IN.pos0) * weights.r;
    position += (IN.pos2 - IN.pos0) * weights.g;
    position += (IN.pos3 - IN.pos0) * weights.b;
    position += (IN.pos4 - IN.pos0) * weights.a;

    //Blend Normal
    normal += (IN.norm1 - IN.norm0) * weights.r;
    normal += (IN.norm2 - IN.norm0) * weights.g;
    normal += (IN.norm3 - IN.norm0) * weights.b;
    normal += (IN.norm4 - IN.norm0) * weights.a;

    //Getting the position of the vertex in the world
    float4 posWorld = mul(position, matW);
    OUT.position = mul(posWorld, matVP);

    normal = normalize(mul(normal, matW));
    float3 tangent = normalize(mul(IN.tangent, matW));
    float3 binormal = normalize(mul(IN.binormal, matW));

    //Getting light-to-vertex direction
    float3 lightDir = normalize(lightPos - posWorld);

    //Get camera-to-vertex direction
    float3 viewDir = normalize(cameraPos - posWorld);

    //Calculate the half vector
    float3 vHalf = normalize(lightDir + viewDir);
```

```
    //Calculating the binormal and setting
    //the tangent binormal and normal matrix
    float3x3 TBNMatrix = float3x3(tangent, binormal, normal);

    //Setting the lightVector
    OUT.lightVec = mul(TBNMatrix, lightDir);
    OUT.lightHalf = mul(TBNMatrix, vHalf);

    OUT.tex0 = IN.tex0;

    return OUT;
}

//Pixel Shader
float4 morphNormalMapPS(VS_OUTPUT IN) : COLOR0
{
    //Calculate the color and the normal
    float4 color = tex2D(DiffuseSampler, IN.tex0);

    //This is how you uncompress a normal map
    float3 normal = 2.0f * tex2D(NormalSampler, IN.tex0).rgb - 1.0f;

    //Get specular
    float4 specularColor = tex2D(SpecularSampler, IN.tex0);

    //Set the output color
    float diffuse = max(saturate(
                        dot(normal, normalize(IN.lightVec))), 0.2f);

    float specular = max(saturate(
                         dot(normal, normalize(IN.lightHalf))), 0.0f);
    specular = pow(specular, 85.0f) * 0.4f;

    return color * diffuse + specularColor * specular;
}
```

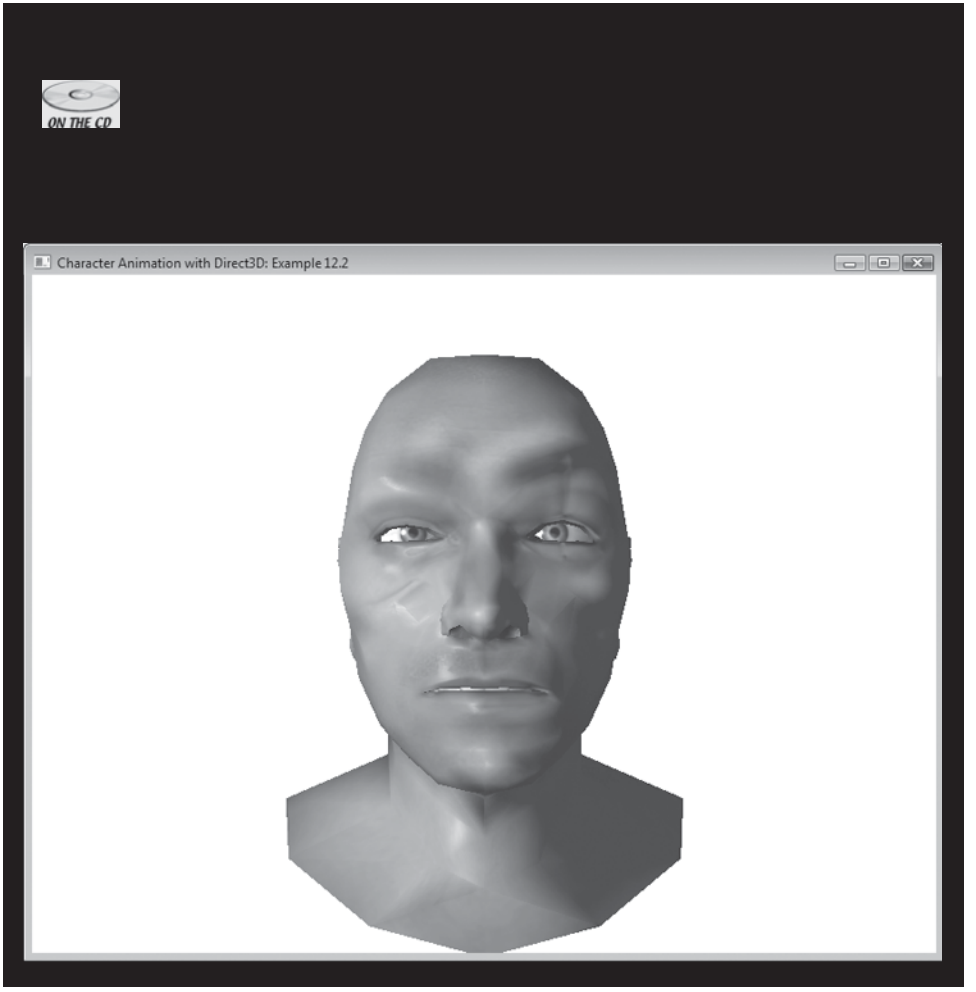


Figure 12.13 shows another screenshot of the Soldier's face using somewhat "exaggerated" highlights. Note that this isn't the kind of result you'd actually want for skin. The examples and images in this chapter are a bit exaggerated to emphasize the effect of the specular highlights.

FIGURE 12.13
Exaggerated highlights.

Wrinkle Maps

You've now arrived at the goal of this chapter: the wrinkle maps. These maps are basically an animated or weighted normal map that is connected to the movement of the face. For example, smiling may reveal the dimples in the cheeks of the characters. These small deformations occur as a result of the underlying muscles in the face moving. Another example of this phenomenon is wrinkles that appear (or disappear) on the forehead as a person raises or lowers his or her eyebrows.

The use of wrinkle maps in games is still a recent addition, and most games today don't bother with it unless the characters are shown in close-up. Figure 12.14 shows a grayscale image of a normal map containing wrinkles for the forehead and dimples.

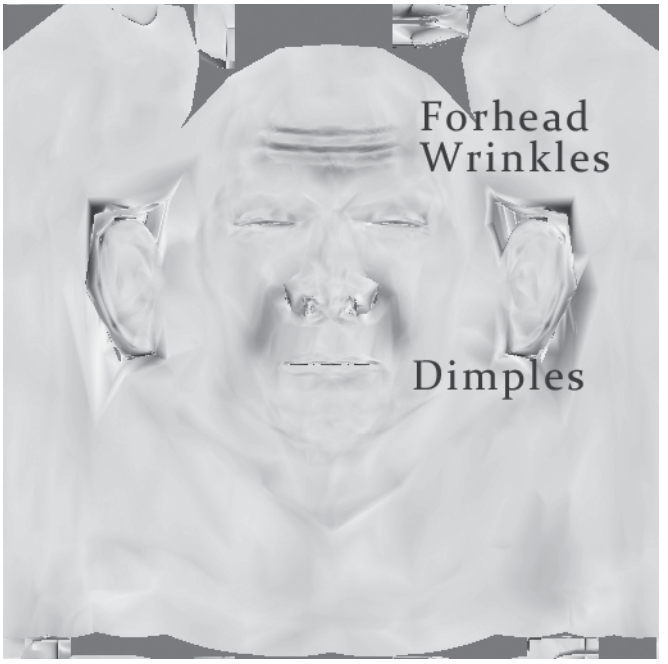


FIGURE 12.14
Wrinkle normal map.

Note that the wrinkles in Figure 12.14 have been made somewhat extreme to stand out a bit better (for educational purposes). Normally, wrinkle maps are something you don't want sticking out like a sore thumb. Rather, they should be a background effect that doesn't steal too much of the focus. Next, you need to encode which regions of the wrinkle map should be affected by which facial movements. In the upcoming wrinkle map example, I've used a separate texture to store masking of the wrinkle map regions. You could, however, also store this data in the vertex color, for example. Figure 12.15 shows the mask used to define the three different regions of the wrinkle map.

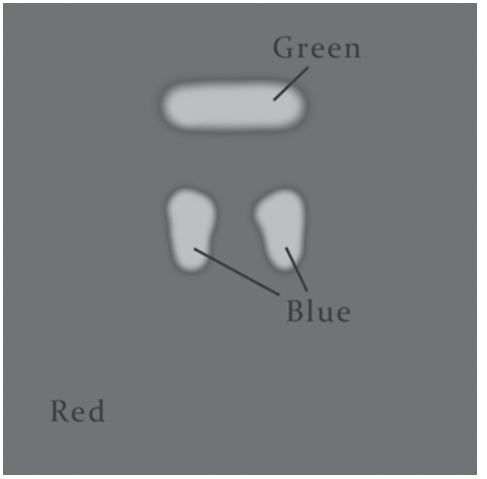


FIGURE 12.15
Wrinkle map mask.

In Figure 12.15, three different regions are defined. The Red channel defines the part of the face that isn't affected by animated wrinkles. The Green channel defines the forehead wrinkles, and the Blue channel defines the two dimple areas at either side of the mouth. To the shader we will upload two blend values (depending on what emotion or shape the character's face has). These two values are called `foreheadWeight` and `cheekWeight`. At each pixel, we sample the blend map, multiply the Green channel with the `foreheadWeight` and the Blue channel with the `cheekWeight`. The resulting value is used to fade the normal in/out from the wrinkle normal map. The following code snippet shows how this is done in the pixel shader:

```
//Pixel Shader
float4 morphNormalMapPS(VS_OUTPUT IN) : COLOR0
{
    //Sample color
    float4 color = tex2D(DiffuseSampler, IN.tex0);

    //Sample blend from wrinkle mask texture
    float4 blend = tex2D(BlendSampler, IN.tex0);

    //Sample normal and decompress
    float3 normal = 2.0f * tex2D(NormalSampler, IN.tex0).rgb - 1.0f;

    //Calculate final normal weight
    float w = blend.r + foreheadWeight * blend.g + cheekWeight * blend.b;
```

```
    w = min( w, 1.0f );
    normal.x *= w;
    normal.y *= w;

    //Re-normalize
    normal = normalize( normal );

    //Normalize the light
    float3 light = normalize(IN.lightVec);

    //Set the output color
    float diffuse = max(saturate(dot(normal, light)), 0.2f);
    return color * diffuse;
}
```

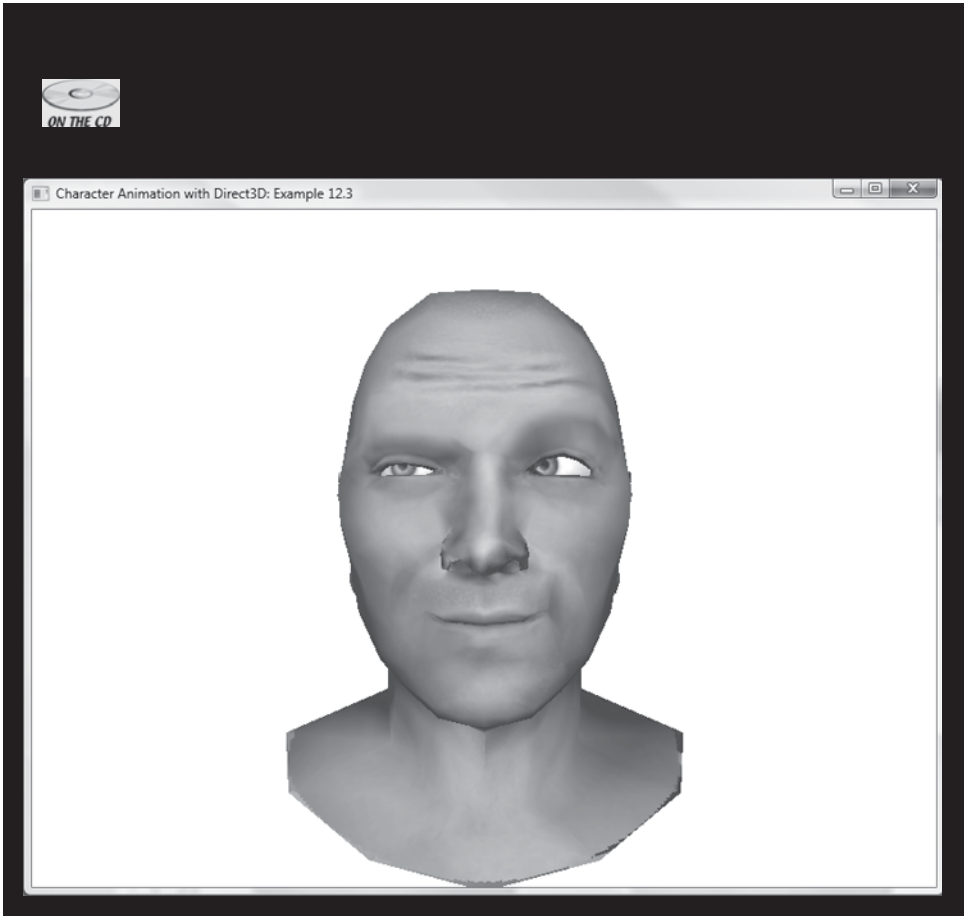


Figure 12.16 shows the wrinkle maps in action.

FIGURE 12.16
Wrinkle maps in action.

Thanks to Henrik Enqvist at Remedy Entertainment for the idea of covering wrinkle maps. He also graciously supplied the example code for the wrinkle map example.

Conclusions

This chapter covered all aspects of normal mapping, from the theory of normal maps to how to create them and how to apply them on a real-time character. This base knowledge then allows you to implement the more advanced wrinkle maps as an animated extension to normal maps. I hope you managed to understand all the steps of this somewhat complex process so that you'll be able to use it in your own projects. The DirectX SDK also has skinned characters with high-quality normal maps, which are excellent to play around with.

I also touched briefly on implementing a specular lighting model—something that, together with normal maps, really makes your character "shine." After the slight sidetrack this chapter has taken, I'll return to more mainstream character animation again. Next up is how to create crowd simulations.

Chapter 12 Exercises

- Implement normal mapping for the `SkinnedMesh` class using the code in the `Face` class as a base.

- Implement normal mapping without supplying the binormal from the mesh, but calculate it on-the-fly in the vertex shader.
- Implement support for multiple lights (for both normal mapping and specular highlights).

Further Reading

[Cloward] Cloward, Ben, "Creating and Using Normal Maps." Available online at http://www.bencloward.com/tutorials_normal_maps1.shtml.

[Gath06] Gath, Jakob, "Derivation of the Tangent Space Matrix." Available online at http://www.blacksmith-studios.dk/projects/downloads/tangent_matrix_derivation.php, 2006.

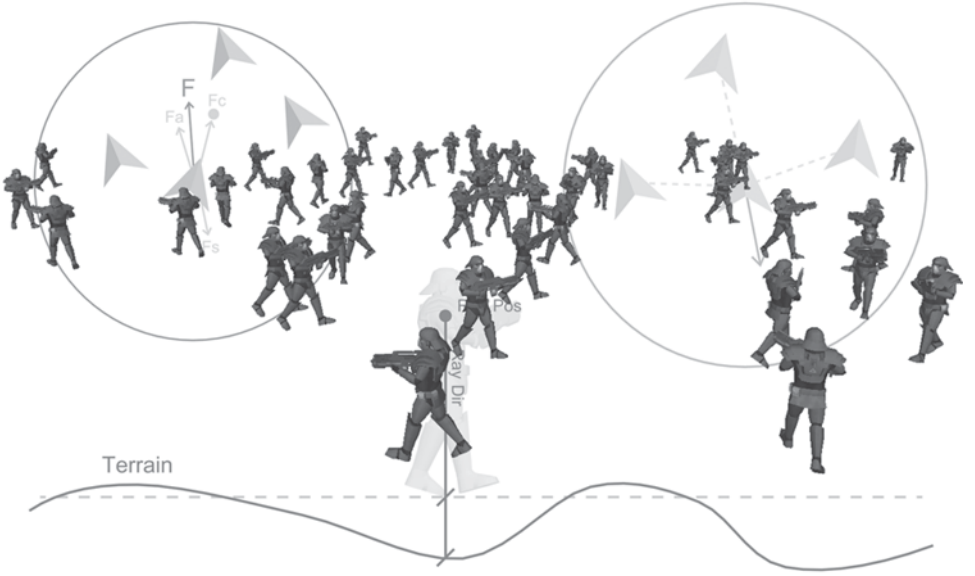
[Green07] Green, Chris, "Efficient Self-Shadowed Radiosity Normal Mapping." Available online at http://www.valvesoftware.com/publications/2007/SIGGRAPH2007_EfficientSelfShadowedRadiosityNormalMapping.pdf, 2007.

[Hess02] Hess, Josh, "Object Space Normal Mapping with Skeletal Animation Tutorial." Available online at: http://www.3dkingdoms.com/tutorial.htm, 2002.

[Lengyel01] Lengyel, Eric. "Computing Tangent Space Basis Vectors for an Arbitrary Mesh." Terathon Software 3D Graphics Library. Available online at http://www.terathon.com/code/tangent.html, 2001.

This page intentionally left blank

13 Crowd Simulation



This chapter introduces the concept of *crowd simulation* and how it can be used in games to control large groups of NPCs. You will first get familiar with the ancestor of crowd simulation—namely, *flocking behaviors*. With flocking behaviors it is possible to control a large group of entities, giving them a seemingly complex group behavior using only a few simple rules. This idea is then carried over to crowd simulation and extended to create some even more complex behaviors. Here's what will be covered in this chapter:

- Flocking behaviors
- "Boids" implementation
- Basic crowd simulation
- Crowd simulation and obstacle avoidance

Flocking Behaviors

Let's start from the beginning. Like many other algorithms in computer science, flocking behaviors (aka swarm behaviors or swarm intelligence) try to emulate what already occurs in nature. Birds, fish, insects, and many other groups of animals seem to exhibit something called *emergent behavior*.

In philosophy, systems theory, and science, emergence is the way complex systems and patterns arise out of a multiplicity of relatively simple interactions. Emergence is central to the theories of integrative levels and of complex systems.

-Wikipedia

Flocking algorithms have been around a while now, and in theory they are simple to both understand and implement. Flocking algorithms are also lightweight (i.e., not very CPU intensive), which is also a huge plus (especially since some of the flocks can have quite a few entities in them). The theory is to have a set of rules that are evaluated for each entity per frame, to determine where this entity should move. The result from the different individual rules are summed up (and often weighted) to produce the final move direction.

One example of this is how ants navigate. If an ant is out on a random hike and finds a source of food, it will leave a trail of pheromones on its way back to the stack, carrying with it a piece of the food. Another ant on its way out from the stack will encounter this trail and then be more likely to follow it to the food source. On its way back, the second ant will lay its own trail of pheromones (reinforcing the first one), making it even more likely that a third ant will follow the first two. Once the food source has been depleted, the ants will stop laying the trail and the pheromones will evaporate with time, stopping ants from wasting time down that trail. So with these seemingly simple rules that each individual follows, the community as a whole still runs a pretty complex operation. This specific example has even spawned its own algorithm called Ant Colony Optimization (ACO), which is used to find good paths through a graph/search space. ACO is an adaptable algorithm, which makes it perfect for changing environments. For example, if a certain ant trail is blocked, the pheromones will evaporate and the ants will start using other trails instead. This technique has been successfully applied to packet routing in networks.

Boids

In 1986, Craig Reynolds made a computer simulation of three simple steering behaviors for a group of creatures he called "Boids." With only three simple steering rules he managed to show some surprisingly complex emergent behavior. Each Boid just considers a local area around itself and then bases its steering on whatever objects or other Boids are in this area.

Separation

The first rule is to make the Boids avoid colliding with other Boids and to avoid crowding each other. This rule is named *Separation*. It calculates a force pointing away from local neighbors, as shown in Figure 13.1.

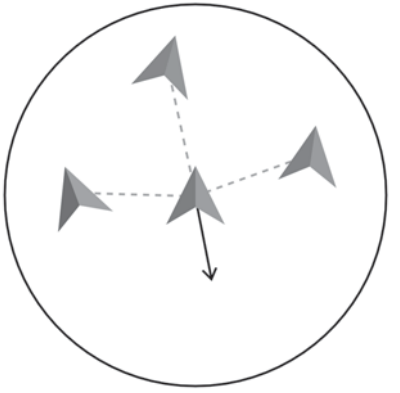


FIGURE 13.1
Separation.

Alignment

The second rule makes Boids keep the same heading as other Boids. This rule is called *Alignment*, and it states that a Boid should steer toward the average heading of its local neighbors. This rule is shown in Figure 13.2.

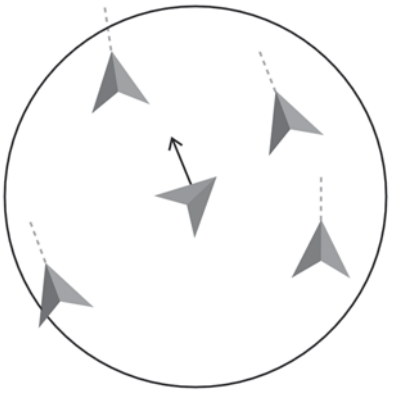


FIGURE 13.2
Alignment.

Cohesion

The third and last rule of the Boid steering behaviors is called *Cohesion*. It keeps the flock together by making a Boid steer toward the center location of its local neighbors. This last rule is shown in Figure 13.3.

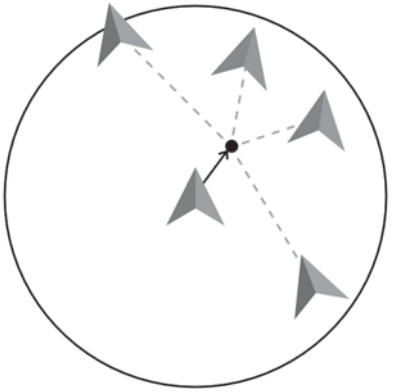


FIGURE 13.3
Cohesion.

Summing Up

For each frame, these three rules each produce a force vector according to the location of a Boid's local neighbors. For now, let's just consider these three simple rules (later you'll see how you can add your own rules to create your own custom behaviors). Figure 13.4 shows the three steering behaviors summed up to produce the final force for the Boid.

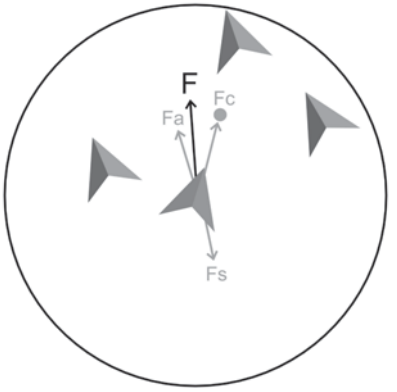


FIGURE 13.4
Summing up the forces.

In Figure 13.4, *Fs*, *Fa*, and *Fc* stand for the forces of the Separation, Alignment, and Cohesion rules, respectively. The resulting force *F* is the force that will be used to update the velocity and position of the Boid. In the upcoming example I'll use the `Boid` class to control an individual entity:

```
class Boid
{
    friend class Flock;
    public:
        Boid(Flock *pFlock);
        ~Boid();
        void Update(float deltaTime);
        void Render(bool shadow);

    private:
        static Mesh* sm_pBoidMesh;

        Flock* m_pFlock;
        D3DXVECTOR3 m_position;
        D3DXVECTOR3 m_velocity;

};
```

The `Boid` class contains a pointer to the flock it belongs to as well as a position and a velocity. In the Boid's `Update()` function the different steering behaviors and their resulting forces are calculated and used to update the velocity and position of the Boid. To manage a flock of Boids, I've created the `Flock` class like this:

```
class Flock
{
    public:
        Flock(int numBoids);
        ~Flock();
        void Update(float deltaTime);
        void Render(bool shadow);
        void GetNeighbors(Boid* pBoid,
                          float radius,
                          vector<Boid*> &neighbors);
    private:
        vector<Boid*> m_boids;
};
```

The only special thing about the `Flock` class is the `GetNeighbors()` function, which just fills a list of Boids within a radius of the querying Boid:

```
void Flock::GetNeighbors(Boid* pBoid,
                         float radius,
                         vector<Boid*> &neighbors)
{
    for(int i=0; i<(int)m_boids.size(); i++)
    {
        if(m_boids[i] != pBoid)
        {
            D3DXVECTOR3 toNeighbor;
            toNeighbor = pBoid->m_position - m_boids[i]->m_position;

            if(D3DXVec3Length(&(toNeighbor)) < radius)
            {
                neighbors.push_back(m_boids[i]);
            }
        }
    }
}
```

Note that the `GetNeighbors()` *function has a rather naïve implementation in this example. For very large flocks it would be unnecessary to loop through the entire flock to find the closest neighbors (especially since we need to do this for each of the entities in the flock). A better way of getting the nearest neighbors would be to use a space partitioning tree, such as a KD-tree. See the following URL for a good introduction to KD-trees:*

http://en.wikipedia.org/wiki/Kd-tree

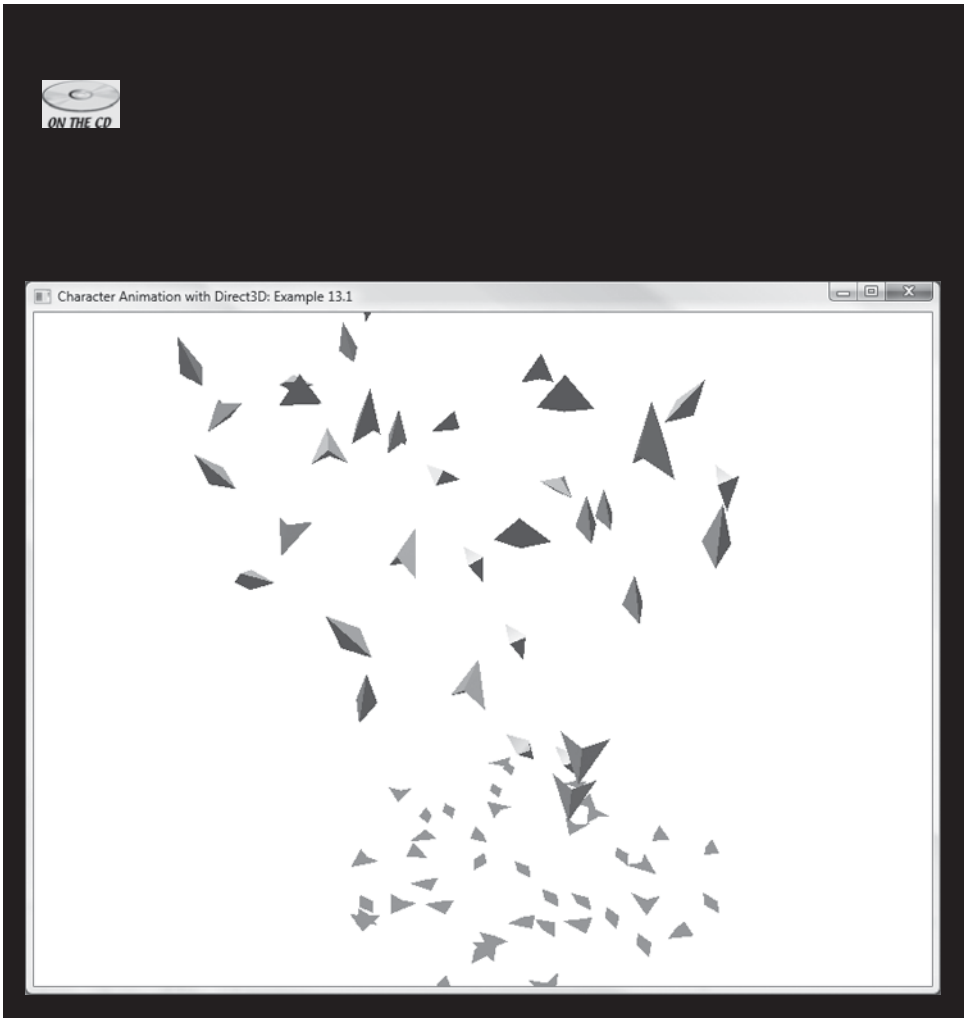
Since each `Boid` object contains a pointer to its flock, it can use the `GetNeighbors()` function to find any neighboring Boids. Then it is easy to calculate the three rather simple steering behaviors as covered earlier, sum these up, and apply the resulting force to the velocity and position of the Boid. The `Boid::Update()` function shows you how:

```
void Boid::Update(float deltaTime)
{
    //Tweakable values
    const float NEIGHBORHOOD_SIZE = 3.5f;
    const float SEPARATION_LIMIT = 2.5f;
    const float SEPARATION_FORCE = 15.0f;
    const float BOID_SPEED = 3.0f;

    //Get neighboring Boids
    vector<Boid*> neighbors;
    m_pFlock->GetNeighbors(this, NEIGHBORHOOD_SIZE, neighbors);

    //Forces
    D3DXVECTOR3 acceleration(0.0f, 0.0f, 0.0f);
    D3DXVECTOR3 separationForce(0.0f, 0.0f, 0.0f);
    D3DXVECTOR3 alignmentForce(0.0f, 0.0f, 0.0f);
    D3DXVECTOR3 cohesionForce(0.0f, 0.0f, 0.0f);
    D3DXVECTOR3 toPointForce(0.0f, 0.0f, 0.0f);
    D3DXVECTOR3 floorForce(0.0f, 0.0f, 0.0f);

    if(!neighbors.empty())
    {
        //Calculate neighborhood center
        D3DXVECTOR3 center(0.0f, 0.0f, 0.0f);
        for(int i=0; i<(int)neighbors.size(); i++)
        {
            center += neighbors[i]->m_position;
        }
        center /= (float)neighbors.size();

        //RULE 1: Separation
        for(int i=0; i<(int)neighbors.size(); i++)
        {
            D3DXVECTOR3 vToNeighbor = neighbors[i]->m_position -
                                      m_position;
            float distToNeightbor = D3DXVec3Length(&vToNeighbor);
```

```
            if(distToNeightbor < SEPARATION_LIMIT)
            {
                //Too close to neighbor
                float force = 1.0f - (distToNeightbor / SEPARATION_LIMIT);
                separationForce -= vToNeighbor * SEPARATION_FORCE * force;
            }
        }

        //RULE 2: Alignment
        for(int i=0; i<(int)neighbors.size(); i++)
        {
            alignmentForce += neighbors[i]->m_velocity;
        }
        alignmentForce /= (float)neighbors.size();

        //RULE 3: Cohesion
        float distToCenter = D3DXVec3Length(&(center - m_position)) + 0.01f;
        cohesionForce = (center - m_position) / distToCenter;
    }

    //RULE 4: Steer to point
    toPointForce = D3DXVECTOR3(0.0f, 15.0f, 0.0f) - m_position;
    D3DXVec3Normalize(&toPointForce, &toPointForce);
    toPointForce *= 0.5f;


    //RULE 5: Dont crash!
    if(m_position.y < 3.0f)
        floorForce.y += (3.0f - m_position.y) * 100.0f;


    //Sum up forces
    acceleration = separationForce +
                   alignmentForce +
                   cohesionForce +
                   toPointForce +
                   floorForce;

    //Update velocity & position
    D3DXVec3Normalize(&acceleration, &acceleration);
    m_velocity += acceleration * deltaTime * 3.0f;
    D3DXVec3Normalize(&m_velocity, &m_velocity);
    m_position += m_velocity * BOID_SPEED * deltaTime;
```

```
        //Cap Y position
        m_position.y = max(m_position.y, 1.0f);
    }
```

In addition to the three normal Boid steering behaviors, I have added a steer-toward-point force (for keeping the Boids in the camera's view frustum) and an avoid-the-floor force, for making sure the Boids don't crash with the floor. Other than that this function is fairly straightforward and implements the steering behaviors covered earlier.



Introduction to Crowd Simulation

There's a lot of academic research being done in the crowd simulation field of computer science. There are several uses for crowd simulations (aside from games). One example is when simulating area or building evacuations. By using crowd simulations it is possible to simulate the time it would take for a crowd to evacuate a building in case of fire, for example. It can also show where any potential choke points are or help to determine where the most efficient place for a fire exit would be, and so on. Another example of using crowd simulation is in civil engineering when simulating pedestrian crossings, sidewalks, city planning, and more. The bottom line is that this is a relatively cheap method of testing all thinkable scenarios and getting a fair idea of how it would play out in reality.

As the processing power of today's computers and consoles are ever increasing, it is suddenly feasible to render a multitude of characters in real time. Many games have recently utilized crowd simulations for "innocent" bystanders—most notable among these games is probably the *Grand Theft Auto* series.

Crowd simulation is basically not that different from flocking algorithms. Usually, crowd simulation takes the steering behaviors used by flocking algorithms to the next level. The entities used in crowd simulation often have much more logic governing their actions, making them much smarter than your average Boid. That is why entities in a crowd simulation are most often referred to as agents instead of entities.

An agent may have its own internal state machine. For example, an agent may act differently if it is in the hypothetical "wounded" state compared to the "normal" state. The crowd simulation may also govern which animations to play for the agents. Say, for example, that two agents are heading toward each other on a collision course. Many things could happen in a situation like this. They could stay and talk to each other, or one could step out of the way for the other one, and so on.

The most basic addition to a crowd simulation system is that of path finding. Instead of just using a simple point in space and steering toward this as the Boids did, your crowd agent could query the environment for a smart path to reach its goal. The agent follows the path in the same manner as the simple "to-point" steering behavior, but the difference is that the agent steers toward the active waypoint instead.

Although a crowd agent usually gets less "smarts" than, say, an enemy opponent character in a first-person shooter game, it still needs to look smart and obey the rules of the environment. At the very least, this means avoiding other crowd members and obstacles placed in the environment. In this section we'll look at extending the flocking steering behaviors to a simple crowd simulation. The following `CrowdEntity` class governs an entity in the crowd (I still call it an entity instead of an agent since there aren't yet quite enough brains in this class at the moment to merit an agent status):

```
class CrowdEntity
{
    friend class Crowd;
    public:
        CrowdEntity(Crowd *pCrowd);
        ~CrowdEntity();
        void Update(float deltaTime);
        void Render();
        D3DXVECTOR3 GetRandomLocation();

    private:
        static SkinnedMesh* sm_pSoldierMesh;

        Crowd* m_pCrowd;
        D3DXVECTOR3 m_position;
        D3DXVECTOR3 m_velocity;
        D3DXVECTOR3 m_goal;

        ID3DXAnimationController* m_pAnimController;
};
```

The `CrowdEntity` class definition looks very much like the `Boid` class definition. The only major differences are the additions of an individual goal, a shared skinned mesh (as opposed to a simple static mesh), and the individual animation controller with which to control the skinned mesh. The `GetRandomLocation()` function is just a simple helper function to generate the goal of a crowd entity. Just as the Boid example had a `Flock` class, this crowd simulation has the following `Crowd` class to govern multiple crowd entities:

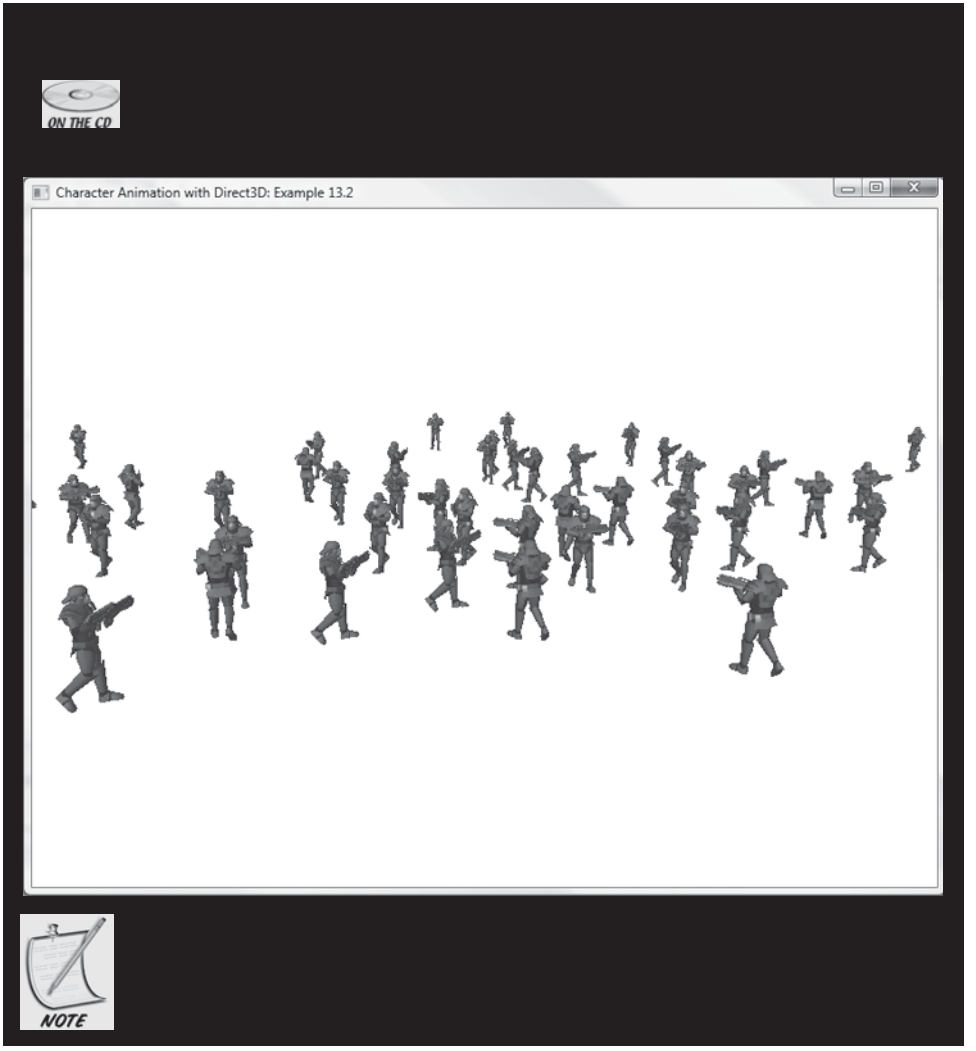
```
class Crowd
{
    public:
        Crowd(int numEntities);
        ~Crowd();
        void Update(float deltaTime);
        void Render();
        void GetNeighbors(CrowdEntity* pEntity,
                          float radius,
                          vector<CrowdEntity*> &neighbors);

    private:
        vector<CrowdEntity*> m_entities;
};
```

On the surface this class also looks like a straight port from the `Flock` class, and in most senses it is. However, later I'll add more stuff to this class, including world obstacles, etc. The only really special code worth looking at in the upcoming example is the `Update()` method of the `CrowdEntity` class, which implements the few current steering behaviors:

```
void CrowdEntity::Update(float deltaTime)
{
    const float ENTITY_INFLUENCE_RADIUS = 3.0f;
    const float NEIGHBOR_REPULSION = 5.0f;
    const float ENTITY_SPEED = 2.0f;
    const float ENTITY_SIZE = 1.0f;

    //Force toward goal
    D3DXVECTOR3 forceToGoal = m_goal - m_position;

    //Has goal been reached?
    if(D3DXVec3Length(&forceToGoal) < ENTITY_INFLUENCE_RADIUS)
    {
        //Pick a new random goal
        m_goal = GetRandomLocation();
    }
    D3DXVec3Normalize(&forceToGoal, &forceToGoal);

    //Get neighbors
    vector<CrowdEntity*> neighbors;
    m_pCrowd->GetNeighbors(this,
                           ENTITY_INFLUENCE_RADIUS,
                           neighbors);

    //Avoid bumping into close neighbors
    D3DXVECTOR3 forceAvoidNeighbors(0.0f, 0.0f, 0.0f);
    for(int i=0; i<(int)neighbors.size(); i++)
    {
        D3DXVECTOR3 toNeighbor;
        toNeighbor = neighbors[i]->m_position - m_position;
        float distToNeighbor = D3DXVec3Length(&toNeighbor);

        toNeighbor.y = 0.0f;
        float force = 1.0f-(distToNeighbor / ENTITY_INFLUENCE_RADIUS);
        forceAvoidNeighbors += -toNeighbor * NEIGHBOR_REPULSION*force;

        //Force move intersecting entities
        if(distToNeighbor < ENTITY_SIZE)
```

```
            {
                D3DXVECTOR3 center = (m_position +
                                      neighbors[i]->m_position) * 0.5f;
                D3DXVECTOR3 dir = center - m_position;
                D3DXVec3Normalize(&dir, &dir);

                //Force move both entities
                m_position = center - dir * ENTITY_SIZE * 0.5f;
                neighbors[i]->m_position = center + dir*ENTITY_SIZE*0.5f;
            }
        }

        //Sum up forces
        D3DXVECTOR3 acc = forceToGoal + forceAvoidNeighbors;
        D3DXVec3Normalize(&acc, &acc);

        //Update velocity & position
        m_velocity += acc * deltaTime;
        D3DXVec3Normalize(&m_velocity, &m_velocity);
        m_position += m_velocity * ENTITY_SPEED * deltaTime;

        //Update animation
        m_pAnimController->AdvanceTime(deltaTime, NULL);
    }
```

There is a very simple "move-toward-goal" logic implemented in this function. As long as the goal is out of reach, a force toward the goal is calculated. Once the goal is within reach of the entity, a new goal is just created at random (note that this is where you would implement a more advanced goal/path finding scheme). This function also implements a simple separation of neighbor repulsion scheme that makes the entities avoid each other (as best they can). However, having the simple separation rule (covered in the earlier section) is in itself not enough. I've also added a hard condition (similar to the way the springs pushed two particles apart in Chapter 6) that forcibly moves two entities apart should they get too close to each other. As an end note I also update the animation controller of the entities, making them seemingly move with a walk animation.



Smart Objects

So far your crowd still looks more or less like a bunch of ants milling around seemingly without purpose. So what is it that makes a crowd agent look and behave like a part of the environment? Well, mostly any interaction your agent does with the environment makes the agent seem "aware" of its environment (even though this is seldom the case).

One way of implementing this is to distribute the object interaction logic to the actual objects themselves. This has been done with great success in, for example,

the *Sims*™ series. The objects contain information about what the object does and how the agent should interact with it. In a simple crowd simulation this could mean that the agent plays a certain animation while standing in range of the object. I'll demonstrate this idea with a simple example of an environment obstacle. The obstacle will take up some certain space in the environment. The agents should then avoid bumping into these obstacles. The `Obstacle` class is defined as follows:

```
class Obstacle
{
    public:
        Obstacle(D3DXVECTOR3 pos, float radius);
        D3DXVECTOR3 GetForce(CrowdEntity* pEntity);
        void Render();

    public:
        static ID3DXMesh* sm_cylinder;
        D3DXVECTOR3 m_position;
        float m_radius;
};
```

The `Obstacle` class has a `GetForce()` function that returns a force pushing crowd entities away from it. Of course you can make your objects take complete control over crowd agents and not just add a force. For example, if you ever have to implement an elevator it would make sense that the elevator takes control of the agent as long as the agent is in the elevator. Nevertheless, here's the `GetForce()` function of the `Obstacle` class:

```
D3DXVECTOR3 Obstacle::GetForce(CrowdEntity* pEntity)
{
    D3DXVECTOR3 vToEntity = m_position - pEntity->m_position;
    float distToEntity = D3DXVec3Length(&vToEntity);

    //Affected by this obstacle?
    if(distToEntity < m_radius * 3.0f)
    {
        D3DXVec3Normalize(&vToEntity, &vToEntity);
        float force = 1.0f - (distToEntity / m_radius * 3.0f);
        return vToEntity * force * 10.0f;
    }

    return D3DXVECTOR3(0.0f, 0.0f, 0.0f);
}
```

This function simply works like a simple force field, pushing away agents in its vicinity with a pretty strong force. This force is simply added to the steering behaviors of the crowd entity. Next, I'll show you how to have the crowd entities follow a mesh, such as a terrain.

Following a Terrain

To follow a terrain mesh you need to sample the height of the terrain at the current position of your agent. This can be done in many different ways, mostly depending on what kind of terrain/environment you have. If, for example, you are having an outside terrain generated from a height map, it is probably better to query this height from the height map rather than querying the terrain mesh. In this example I'll use the suboptimal mesh querying, foremost because there's no advanced terrain representation in place, and secondly because that will introduce the `D3DXIntersect()` function that you will need to use in the next chapter anyway. This function is defined as follows:

```
HRESULT D3DXIntersect(
  LPD3DXBASEMESH pMesh,          //Mesh to query
  CONST D3DXVECTOR3 * pRayPos,   //Ray origin
  CONST D3DXVECTOR3 * pRayDir,   //Ray direction
  BOOL * pHit,                   //Does the ray hit the mesh?
  DWORD * pFaceIndex,            //Which face index was hit?
  FLOAT * pU,                    //Hit U coordinate
  FLOAT * pV,                    //Hit V coordinate
  FLOAT * pDist,                 //Distance to hit
  LPD3DXBUFFER * ppAllHits,      //Buffer with multiple hits
  DWORD * pCountOfHits           //Number of hits
);
```

This function can be used to get the results from a Ray-Mesh intersection test. The `pHit` pointer will point to a Boolean that contains true or false depending on whether or not the mesh was hit by the ray. If the mesh is intersecting the ray, the `pFaceIndex`, `pU`, `pV`, and `pDist` pointers will fill their respective variables with the information from the hit closest to the ray origin. In most cases you're only interested in the hit closest to the ray origin, but sometimes you also want to know where the ray exited the mesh, etc. If so, you can access all hits/intersection locations of the mesh with the `ppAllHits` buffer.

You can use this function to place a character on the environment, such as in Figure 13.5.

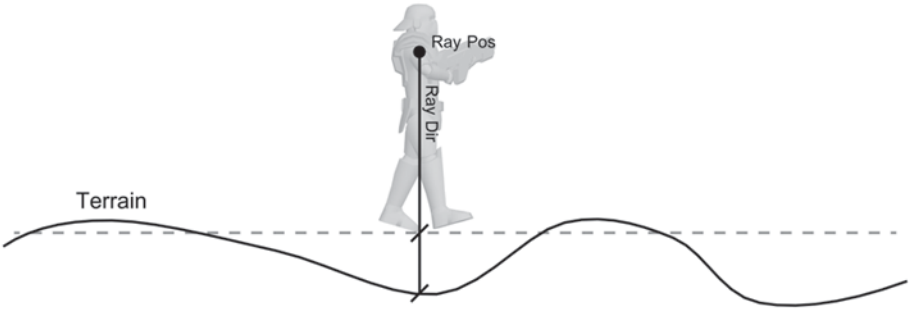


FIGURE 13.5
Placing a character on the terrain.

Creating the ray to test against the terrain mesh is a simple task. Simply take the X and Z coordinate of your character and set the Y coordinate to an arbitrary number greater than the highest peak of the terrain (should your ray origin be lower than the terrain at the testing point, your intersection test will fail). When successful you get the distance to the terrain mesh from the `D3DXIntersect()` function. Then, to get the height of the terrain at the testing point, you simply add this intersection distance to the Y coordinate of the ray origin. The new `SetEntityGroundPos()` function in the `Crowd` class adjusts the position to follow the terrain:

```
void Crowd::SetEntityGroundPos(D3DXVECTOR3 &pos)
{
    //Create the test ray
    D3DXVECTOR3 org = pos + D3DXVECTOR3(0.0f, 10.0f, 0.0f);
    D3DXVECTOR3 dir = D3DXVECTOR3(0.0f, -1.0f, 0.0f);

    BOOL Hit;
    DWORD FaceIndex;
    FLOAT U;
    FLOAT V;
    FLOAT Dist;

    //Floor-ray intersection test
    D3DXIntersect(m_pFloor->m_pMesh,
                  &org,
                  &dir,
                  &Hit,
                  &FaceIndex,
                  &U,
                  &V,
```

```
                    &Dist,
                    NULL,
                    NULL);

    if(Hit)
    {
        //Adjust position according to the floor height
        pos.y = org.y - Dist;
    }
}
```

Conclusions

This chapter provided a brief glimpse into the subject of crowd simulation. The code provided with this chapter will hopefully provide a good base for your own expansions. Start with something as simple as the three Boid steering behaviors, to which you can easily add more and more specific steering behaviors to get your desired result.

Crowd simulation can be used for both enemy and NPC steering and is currently one of the best options for controlling a large mass of characters. You can pretty much take all of what you've learned so far in the book and apply it to your crowd agents, ragdoll, inverse kinematics, facial animation, and more.

Chapter 13 Exercises

- Implement a more efficient way of finding the nearest neighbors of a flock or crowd entity. (Tip: Look into KD-trees.)
- Implement a `Prey` class that takes the roll of a predator hunting the poor Boids. Make the prey attack and devour Boids within a small attack radius. Also, make all the Boids flee the predator at all costs.
- Implement a path-finding algorithm (such as A-star, Djikstra's, or similar) and use this to guide your agents through a complex environment while using the crowd steering behaviors to resolve collisions, etc.
- Implement non-constant speed in the crowd simulation, making it possible for entities to pause if they're temporarily blocked by another entity. Also, be sure to switch the animation in this case to the still animation.
- Create a smart object that makes a crowd entity stop and salute the object before continuing its milling around.
- Create a leader agent that the other crowd entities follow.

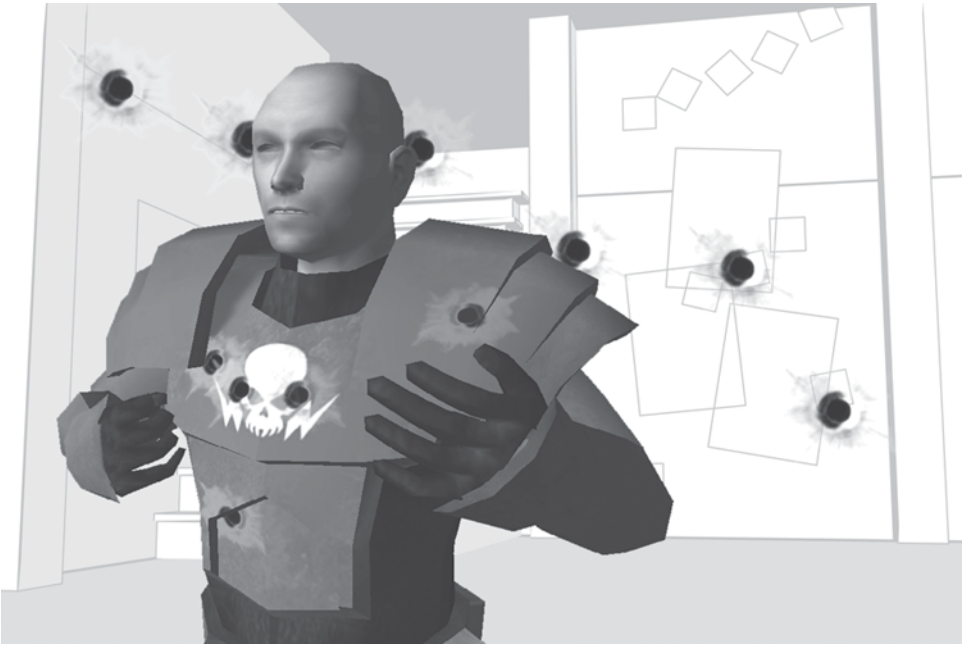
Further Reading

Sakuma, Takeshi *et al.*, "Psychology-Based Crowd Simulation." Available online at: http://www.val.ics.tut.ac.jp/project/crowd/, 2005.

Sung, Mankyu, "Scalable behaviors for crowd simulation." Available online at: http://www.cs.wisc.edu/graphics/Papers/Gleicher/Crowds/crowd.pdf, 2004.

This page intentionally left blank

14 Character Decals



This chapter touches on another cousin of character animation: character decals! This is a somewhat obscure topic that also can be quite hard to find tutorials about online—even though it has been around since some of the first 3D games. Applying decals to geometry in your scene is a concrete problem you will be faced with at some point if you ever try to make a game in which guns or similar things are fired. In this chapter I'll cover everything from the history of decals to how to add decals to your animated characters.

- Decals in games
- Picking a hardware-rendered mesh
- Creating decal geometry
- Calculating decal UV coordinates

Introduction to Decals

Like many other techniques in computer graphics, the concept of decals has its roots in the real world. The word decal is defined as follows:

> "A design or picture produced in order to be transferred to another surface either permanently or temporarily."
>
> Or,
>
> "A decorative sticker."
>
> -Wikipedia

A decorative sticker…that pretty much sums it up nicely. In games, decals are used to decorate otherwise plain surfaces or add more detail. Simple decals are implemented as a small quad with an alpha-blended texture on it that is placed exactly on the plane of the wall to which it is supposed to "stick." The decal is then rendered as usual after the wall. In Direct3D the default Z buffer test is to allow anything with less Z value or *equal* through. Since the decal is at the exact same Z distance as the wall behind it, the decal will be painted on top of the wall (but more on this later on). Figure 14.1 shows a basic 3D scene with some decals applied.

To implement this scene without the use of decals would require an unnecessary amount of texture memory. Figure 14.2 shows the scene in Figure 14.1 wireframe rendered.

As you can see, there's a lot of texture memory saved by not having to put these extra details (posters, bullet holes, etc.) in the base texture. On top of preserving texture memory, there is another very important aspect of decals—you are able to add these decals dynamically to the game. Decals have been around since some of the very first 3D games.

FIGURE 14.1
A 3D example scene.

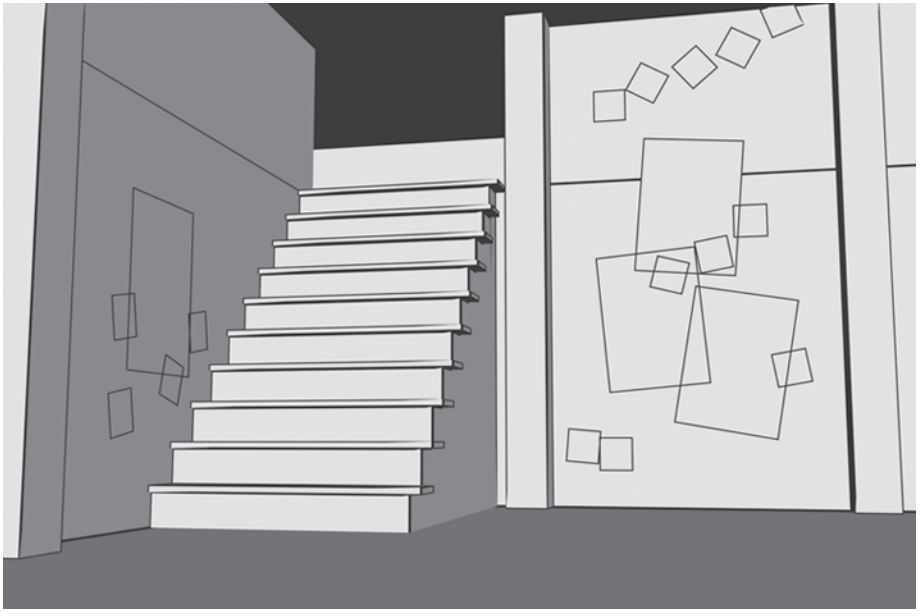


FIGURE 14.2
Wireframe rendering of the scene in Figure 14.1.

The most common usage of decals is to add bullet holes to the walls. These are put there as a result of the player firing a gun. Adding decals to a static scene is a relatively easy task, especially if all the walls and floors, etc,. are planar surfaces. Then, all you need to do is calculate the plane of the surface and add your decal quad to this plane and render away.

However, with characters you don't really have the luxury of planar surfaces. You also have to deal with the fact that your characters are skinned meshes that move around. Decals have to "stick" to their base surface. Otherwise, you will have lots of flickering as a result of Z-fighting, or decals that seem detached from the character—either way, the illusion is broken. So in order to create and render decals on a character, there are five steps you need to take:

1. Pick the triangle on the character through which a ray intersects.
2. Grow the selection of triangles until you have a large enough surface to fit the decal.
3. Calculate new UV coordinates for the decal mesh.
4. Copy the skinning information (blending weights and blending indices) to the decal mesh.
5. Render the decal as any other bone mesh (with the small exception of using clamped UV).

OK, now it's time to get down to the nitty-gritty and start looking at the implementation.

Picking a Hardware-Rendered Mesh

Picking is another name for a collection of ray intersection tests. A ray is usually made up of two 3D vectors—an origin and a direction. The most common example is a ray in your 3D world that is calculated from the position of your mouse (in screen space)—something you have probably come across in an introductory book on 3D graphics. In this chapter I won't cover how the mouse ray is calculated; rather, I'll stick to the general case of having any arbitrary ray in world space and using it to paint a decal on a character.

There are several ray intersection tests that may prove useful when you write your game code. The most common ray intersection tests are ray-bounding box, ray-bounding sphere, ray-plane, and finally, the ray-mesh intersection test. You can (and should) use the cheaper bounding volume intersection tests before using the more expensive ray-mesh intersection test. However, I leave such optimizations up to you.

The following D3DX functions implement the ray intersection tests with the box and sphere bounding volume as well as the plane:

```
BOOL D3DXBoxBoundProbe(
    CONST D3DXVECTOR3 *pMin,
    CONST D3DXVECTOR3 *pMax,
    CONST D3DXVECTOR3 *pRayPosition,
    CONST D3DXVECTOR3 *pRayDirection
);

BOOL D3DXSphereBoundProbe(
    CONST D3DXVECTOR3 *pCenter,
    FLOAT Radius,
    CONST D3DXVECTOR3 *pRayPosition,
    CONST D3DXVECTOR3 *pRayDirection
);

D3DXVECTOR3 * D3DXPlaneIntersectLine(
    D3DXVECTOR3 *pOut,
    CONST D3DXPLANE *pP,
    CONST D3DXVECTOR3 *pV1,
    CONST D3DXVECTOR3 *pV2
);
```

For the first two functions, all you have to do is supply the dimensions of the bounding volumes along with your ray postion (aka the ray origin) and your ray direction. You will get a simple Boolean telling you whether or not the ray intersected the volume.

Note that the plane intersection test works a bit differently. It takes the beginning and the end point of the ray/line you wish to test and returns the point where the line intersects the plane.

The ray-mesh test I will use for finding the place where a ray intersects with our character is implemented in the D3DX library in the little bit more advanced `D3DXIntersect()` function:

```
HRESULT D3DXIntersect(
    LPD3DXBASEMESH pMesh,          //Mesh to test
    CONST D3DXVECTOR3 * pRayPos,   //Ray origin
    CONST D3DXVECTOR3 * pRayDir,   //Ray direction
    BOOL * pHit,                   //Did the ray hit or not?
    DWORD * pFaceIndex,            //Index of triangle which was hit
    FLOAT * pU,                    //Barycentric U coordinate of hit
```

```
        FLOAT * pV,                     //Barycentric V coordinate of hit
        FLOAT * pDist,                  //Distance to hit (from ray origin)
        LPD3DXBUFFER * ppAllHits,       //List of all hits
        DWORD * pCountOfHits            //Number of hits
    );
```

In addition to testing the ray and the mesh, there are also a lot of pointers to data containers that you need to pass to this function. The `pHit` will write a Boolean telling whether or not the ray hit the mesh. Sometimes this is all the information you are interested in—for example, when you want to just select a 3D object with the mouse. For creating decals, however, you need all the information this function returns: face index, barycentric coordinates of the hit (more on this later), and the distance to the hit. A ray may also hit a mesh in more than just one place, as shown in Figure 14.3.

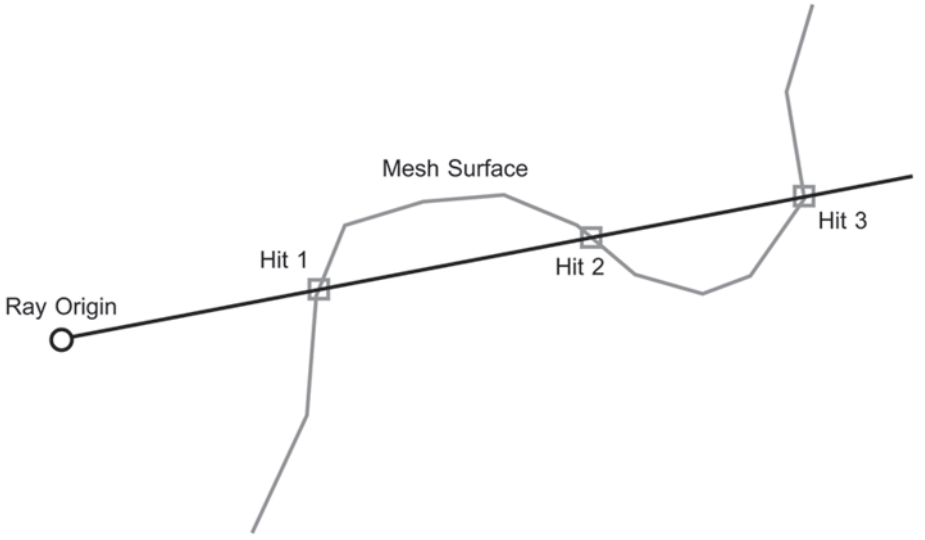


FIGURE 14.3
A ray intersecting a mesh.

If you want all the hits, you can get these from the `ppAllHits` buffer (which has `pCountOfHits` number of hits). The information for each of these hits is stored with the `D3DXINTERSECTINFO` structure:

```
struct D3DXINTERSECTINFO {
    DWORD FaceIndex;
    FLOAT U;
    FLOAT V;
    FLOAT Dist;
}
```

So, returning to the problem at hand…. The `D3DXIntersect()` function takes a pointer to a mesh on which you want to run your ray-mesh intersection test. However, since I'm using hardware-skinned characters in this book (and which in all likelihood you will be using as well in real-life applications), there's only the original mesh available for testing. Figure 14.4 shows this dilemma.

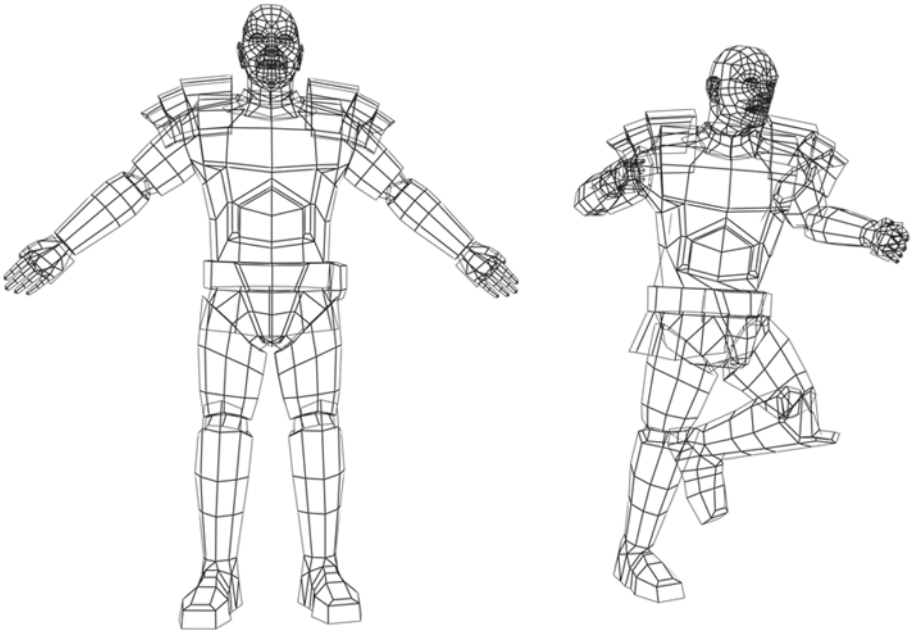


FIGURE 14.4
Intersecting a hardware-skinned character.

Doing this intersection test with a software-skinned character wouldn't be a problem because you have the actual skinned mesh stored in memory as opposed to being skinned on-the-fly as is the case with hardware-skinned characters. Despite this obvious flaw of not having the final skinned mesh in memory, there are a few different ways you can still perform ray intersection tests on a hardware-skinned character. Figure 14.5 shows one of the most common ways of doing this.

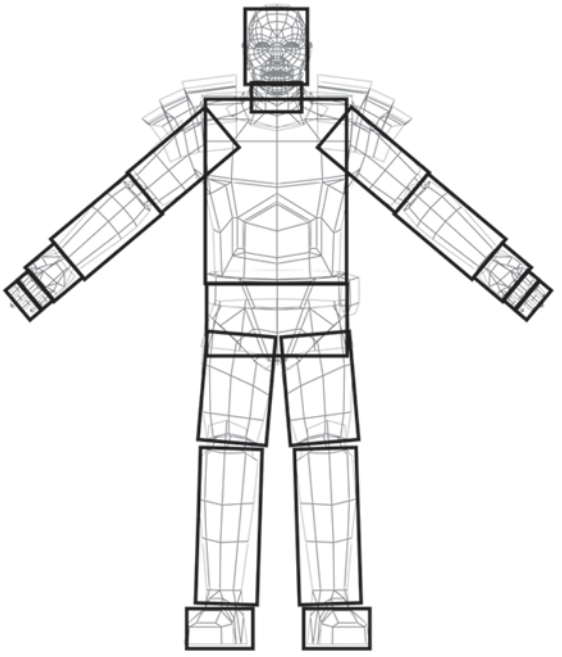


FIGURE 14.5
Bone bounding volumes.

Since using bounding volumes for the different bones may be something an engine already supports, for example, for ragdolls, this is a popular way to do character intersection tests. All you need to do then is transform the ray from world space to the local space of the bone in question and perform a simple ray-bounding volume test. Once you've found a bone that the ray intersects with you can use a variety of techniques to pick a good spot for your decal on the mesh.

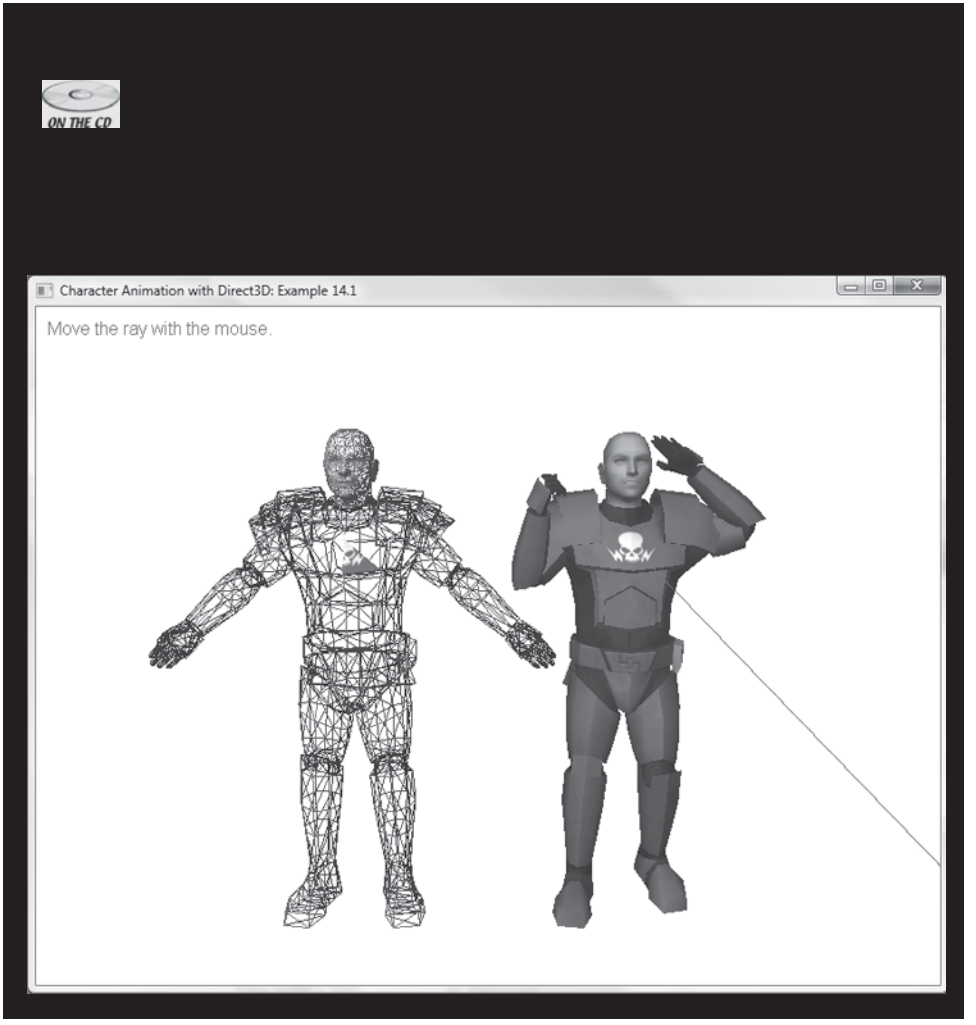
I'll use a somewhat more straightforward, albeit not so efficient, method of getting the intersection data I need. I'll simply take the original mesh (which I happened to have stored away in the `BoneMesh` class) and perform a software skinning to a temporary mesh in memory. Then I use the `D3DXIntersect()` function on this temporary mesh to get the data I need before releasing it.

I will add most of this new decal functionality to the `BoneMesh` class, since this class extends `D3DXMESHCONTAINER` and contains the skinning information, original mesh, etc. The following `GetFace()` function in the `BoneMesh` class returns the intersection data of a hardware- (or software) skinned model intersecting with the provided ray (origin + direction):

```
D3DXINTERSECTINFO BoneMesh::GetFace(
                      D3DXVECTOR3 &rayOrg,
                      D3DXVECTOR3 &rayDir)
{
    D3DXINTERSECTINFO hitInfo;

    //Must test against software-skinned model
    if (pSkinInfo != NULL)
    {
        //Make sure vertex format is correct
        if(OriginalMesh->GetFVF() != Vertex::FVF)
        {
            hitInfo.FaceIndex = 0xffffffff;
            return hitInfo;
        }

        //Set up bone transforms
        int numBones = pSkinInfo->GetNumBones();
        for(int i=0;i < numBones;i++)
        {
            D3DXMatrixMultiply(&currentBoneMatrices[i],
                               &boneOffsetMatrices[i],
                               boneMatrixPtrs[i]);
        }

        //Create temp mesh
        ID3DXMesh *tempMesh = NULL;
        OriginalMesh->CloneMeshFVF(D3DXMESH_MANAGED,
                                   OriginalMesh->GetFVF(),
                                   g_pDevice,
                                   &tempMesh);

        //Get source and destination buffer
        BYTE *src = NULL;
        BYTE *dest = NULL;
        OriginalMesh->LockVertexBuffer(D3DLOCK_READONLY,
                                       (VOID**)&src);
        tempMesh->LockVertexBuffer(0,
                                   (VOID**)&dest);

        //Perform the software skinning
        pSkinInfo->UpdateSkinnedMesh(currentBoneMatrices,
                                     NULL,
                                     src,
                                     dest);
```

```
            //Unlock buffers
            OriginalMesh->UnlockVertexBuffer();
            tempMesh->UnlockVertexBuffer();

            //Perform the intersection test
            BOOL Hit;
            D3DXIntersect(tempMesh,
                          &rayOrg,
                          &rayDir,
                          &Hit,
                          &hitInfo.FaceIndex,
                          &hitInfo.U,
                          &hitInfo.V,
                          &hitInfo.Dist,
                          NULL,
                          NULL);

            //Release temporary mesh
            tempMesh->Release();

            if(Hit)
            {
                //Successful hit
                return hitInfo;
            }
        }

        //No hit
        hitInfo.FaceIndex = 0xffffffff;
        hitInfo.Dist = -1.0f;
        return hitInfo;
    }
```

As you can see, this function returns a `D3DXINTERSECTINFO` object containing all the necessary intersection info. Should the ray completely miss the character, then the `FaceIndex` (a `DWORD` variable) will be set to `0xffffffff` (the maximum value of a `DWORD`). This is also the default invalid value for `DWORD`s used by the DirectX API.



Creating Decal Geometry

In the previous section you learned how to find which triangle of a character that a certain ray intersects with (even if the character happens to be hardware skinned). The next problem that follows is how you calculate which surrounding triangles should be selected for the decal mesh (remember that the decal is, in most cases, larger than the average triangle on your character). Again, there are a few different approaches to this, and as always they have varying accuracy and efficiency. The brute force approach would be to test all triangles of the entire character and select those inside the decal test volume (which could be calculated as a sphere around the

hit position with a radius of the decal size). The problem with this approach, however, is quite obvious: not only will it be quite slow (especially for characters with tens of thousands of triangles), but it also has the potential to include lots of unnecessary triangles, as shown in Figure 14.6.

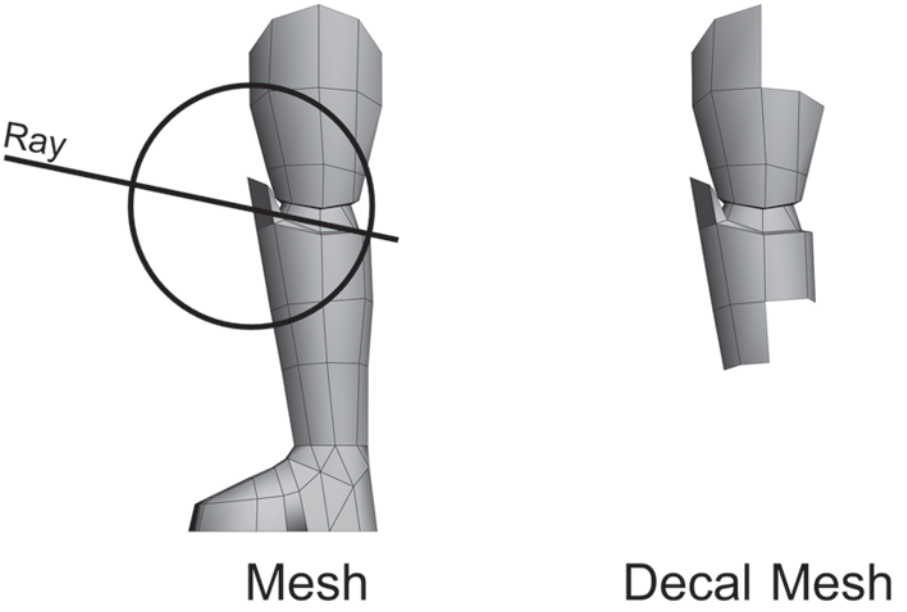


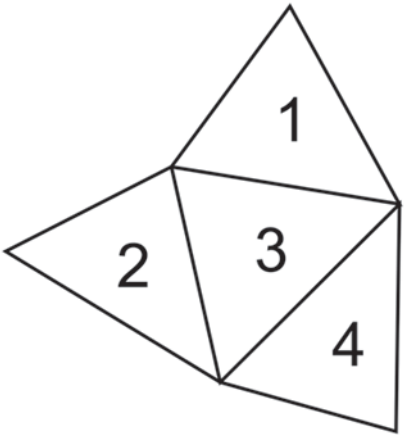
FIGURE 14.6
The problem with brute force selection of the decal mesh.

As you can see in Figure 14.6, the ray hits the front of the character's leg. Since the decal size is larger than the leg itself, it will end up selecting polygons on the backside of the leg, which is something you'd like to avoid. This will happen as long as the decal size is larger than the thickness of the body part being tested (and note that the polygons don't even have to be adjacent when selected with this method). So when a decal is added to the front of the character's torso, there might be some polygons added on the back as well. So, even though the brute force method would work, let's try a better one.

It is quite easy to calculate the adjacency information of a mesh (information of which triangles share which edges). The adjacency information of a mesh can be calculated using the following D3DX library function defined in the `ID3DXBaseMesh` class:

```
HRESULT GenerateAdjacency(
    FLOAT Epsilon,
    DWORD * pAdjacency
);
```

This function returns a list of indices containing the information of which faces are adjacent to each other. Any vertices closer to each other than the Epsilon variable will be treated as coincident. The resulting adjacency data will be written to the `pAdjacency` pointer. An example mesh with its calculated adjacency information is shown in Figure 14.7.



Tri 1	Tri 2	Tri 3	Tri 4
0xffffffff	0xffffffff	2	3
0xffffffff	3	1	0xffffffff
3	0xffffffff	4	0xffffffff

FIGURE 14.7
Four example triangles.

In Figure 14.7 there's an example mesh consisting of four triangles (numbered 1 to 4). The adjacency information consists of double words (`DWORD`). If a certain edge of a triangle doesn't have a neighboring triangle, this slot will be filled with the value `0xffffffff`. The neighbors can then be extracted in code like this:

```
//Extract adjancency info from mesh
DWORD* adj = new DWORD[numFaces * 3];
pSomeMesh->GenerateAdjancency(0.01f, adj);

DWORD someTriangle = 32;

//Get neighbors of "someTriangle"
DWORD neighbor1 = adj[someTriangle * 3 + 0];
DWORD neighbor2 = adj[someTriangle * 3 + 1];
DWORD neighbor3 = adj[someTriangle * 3 + 2];
```

With the adjacency information you can start from the triangle that was hit and do a simple flood-fill to create the decal mesh. The neighbors of the triangle that was hit are added to an open list and evaluated in turn. If these faces, in turn, have neighboring meshes within the decal radius, these are also added to the open list (unless they already are in the list). Even if you use a bounding sphere around the decal hit position, this will generate much better decal meshes than the brute force approach since it requires the triangles to be *connected*.

Calculating the Exact Hit Position

The next thing you need to calculate is the location of the ray hit. You will need to use this location to create the bounding sphere with which you test any neighboring triangles. It is quite easy to calculate the exact hit position of the ray in world space since you have the ray origin, ray direction, and the distance to the hit. The world space hit location can then be calculated easily:

```
D3DXVECTOR3 hitPos = rayOrg + rayDir * distToHit;
```

The problem with using the hit location in world space is that it may not correspond well to the original mesh since the character is skinned. This means that the triangles in the character are affected by bone transformations and are therefore moved, rotated, stretched, etc.

Instead, you must calculate the hit position of the ray in the local space of the original un-skinned mesh character. An easy way to do this would be to take the center of the triangle that was hit (the hit index will be the same in both the skinned character as in the original mesh). This is easy to do since you can access the vertex information of the three triangle corners. The triangle center would probably work well enough if your character has a detailed enough mesh. However, I think we can do one better in this matter as well.

A more detailed hit position (in the un-skinned mesh local space) can be calculated using the UV barycentric coordinates given to you from the `D3DXIntersect()` function. Consider the triangle in Figure 14.8.

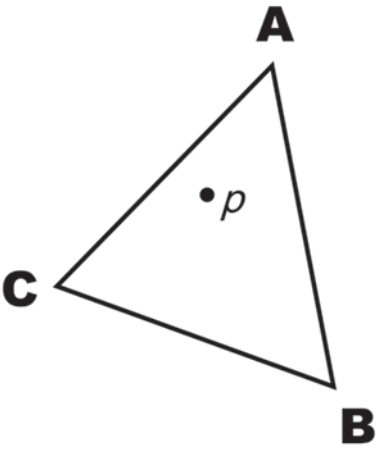


FIGURE 14.8
An example triangle with corners A, B, and C.

Given any arbitrary barycentric coordinates (u and v), any point (p) on this triangle (A, B, C) can be described as:

$$p = A + u(B-A) + v(C-A)$$

Try, in your head, to calculate the barycentric coordinate of a few points on the triangle in Figure 14.8. After just trying a few points you'll realize the truth that any given point on a triangle can be described using only two barycentric coordinates, regardless of the shape of the triangle. The simple formula above is also implemented by the following D3DX library function:

```
D3DXVECTOR3 * D3DXVec3BaryCentric(
    D3DXVECTOR3 * pOut,             //Resulting point on triangle
    CONST D3DXVECTOR3 * pV1,        //Corner 1 (A)
    CONST D3DXVECTOR3 * pV2,        //Corner 2 (B)
    CONST D3DXVECTOR3 * pV3,        //Corner 3 (C)
    FLOAT f,                        //Barycentric U coordinate
    FLOAT g                         //Barycentric V coordinate
);
```

With the `D3DXINTERSECTINFO` object returned from the `GetFace()` function created in the previous example, you can easily extract the vertices of the triangle hit by the ray. Then, feed them into the `D3DXVec3BaryCentric()` function together with the barycentric coordinates of the hit. Out comes the *exact* hit position of the ray in the un-skinned character's local space (which is the space in which all the vertex positions are stored).

Selecting Triangles for the Decal Mesh

Before I cover the code on how to create the decal mesh there is one more issue to discuss. When you create the decal mesh you must ensure that *all* affected triangles are added to the decal mesh. The idea is to do a flood-fill of triangles out from the triangle that was hit by the ray and stop once you've selected a large enough submesh to "house" the decal. With each triangle you test you can extract the three corners (i.e., mesh vertices) and test against the decal testing volume. However, this brings us to the following problem described in Figure 14.9.

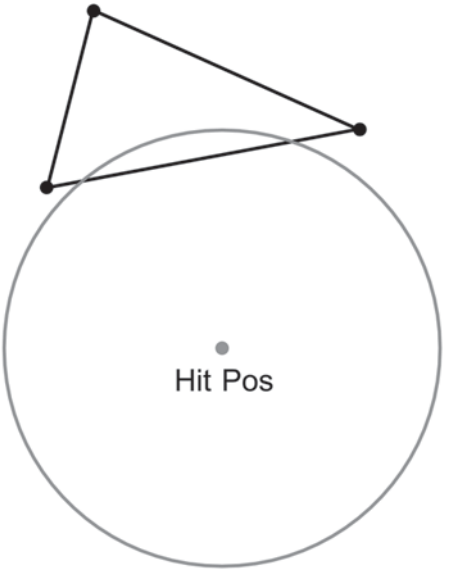


FIGURE 14.9
A simplified sphere-triangle test.

Here, the problem becomes apparent if you only test the vertices of the triangle against the bounding sphere. As you can see, the bounding sphere intersects the triangle but misses all of the triangle vertices. This will lead to the triangle not being included in the decal mesh, which in turn can give your decal a visual artifact

(making it look like a piece of the decal is missing). Again, there are many different ways to handle this. One easy way to solve this would be to calculate the bounding sphere of the triangle itself and use a sphere–sphere intersection test to determine whether the triangle should be included. Alas, this approach often results in too many triangles being selected, with large triangles having a much larger chance of being selected. So the simple solution is just to increase the testing radius of the decal bounding sphere. This will, of course, have to be increased with a magical number differing from character to character. In most cases a simple 10%–20% increase should do it. Again, there are more accurate ways to go about this, but in the end, we're not calculating the trajectories of a ballistic missile here. Feel free to spend time on perfecting the solution to this problem though.

Copying the Skinning Information

Finally, you've come to the point where you need to create the actual decal mesh using the subset of triangles selected using the techniques covered in the previous sections. However, I've been talking a lot about the original mesh of the character and how you need to select the triangles to use for the decal mesh from this original mesh. Now that the time has come to actually copy the selected submesh and create the new decal mesh, there is one more important thing to consider—namely, that of the skinning information. Since one of the basic problems you were faced with when wanting to create decals for characters was that they are dynamic and animated meshes, you need to make sure that your decals are also dynamic. The decal needs to map exactly to the underlying character and follow the character's every move. For this to happen, the decal needs to also copy the skin information of the skinned character (for the affected triangles only).

I must admit that the first time I approached this problem I took the long way around. I created a new `ID3DXSkinInfo` *object and manually copied over the skinning data from the skin info object stored in the* `BoneMesh` *class to the new skin info object of the decal mesh. Although this worked (and was a great exercise), it was also completely unnecessary. However, if you're ever faced with the prospect of copying skinning information from one character to another, here's the basic outline on how to proceed:*

1. *Create a new* `ID3DXSkinInfo` *object using the* `D3DXCreateSkinInfo()` *or* `D3DXCreateSkinInfoFVF()` *function.*
2. *Use the* `ID3DXSkinInfo::GetBoneInfluence()` *function to retrieve the vertices and weights from the source character.*
3. *Map bone influences to actual vertices (these are not the same) using the* `ID3DXSkinInfo::GetBoneVertexInfluence()` *function.*
4. *Finally, write the newly created vertices and weights to the target skin info object using the* `ID3DXSkinInfo::SetBoneInfluence()` *function.*

Since you already have the finished index-blended mesh stored in the `BoneMesh` class, you have all you need ready at your disposal. If you keep the bone setup the same for the decal meshes as you do for the underlying skinned mesh, you are actually saving a lot of unnecessary instructions, even though that might seem confusing. Had you instead done like I first did and calculated a new skin info object for the decal, you would have to recalculate the matrix palette and upload it to the vertex shader for each decal you want to draw. Instead, it is better just to keep the bone setup of the parent mesh and store all the decal meshes in a list to be drawn *after* the base mesh using the same bone setup.

So, back to the issue at hand. You now have a list of faces selected from the character mesh using some more or less accurate schemes. The next step is to create a new decal mesh containing the right number of faces and vertices. This can be done with the following D3DX function:

```
HRESULT D3DXCreateMesh(
    DWORD NumFaces,                              //Num faces
    DWORD NumVertices,                           //Num vertices
    DWORD Options,                               //Creation/memory flags
    CONST LPD3DVERTEXELEMENT9 * pDeclaration, //Vertex declaration
    LPDIRECT3DDEVICE9 pD3DDevice,                //Graphics device
    LPD3DXMESH * ppMesh                          //Resulting mesh
);
```

Now you can lock the vertex or index buffer by this and the source mesh and simply copy over the data. To make it easier to handle the data in a vertex buffer, for example, it pays off to create a small vertex structure corresponding to the vertex declaration. In the upcoming example I'll use the following structure to handle an index blended vertex:

```
struct DecalVertex{
    D3DXVECTOR3 position;
    Float blendweights;
    Byte blendindices[4];
    D3DXVECTOR3 normal;
    D3DXVECTOR2 uv;
};
```

Note that it is very important that the layout of the information in this structure *exactly* corresponds to the layout of the information in the vertex declaration. Especially important is that the size of your vertex structure corresponds to the size of your mesh vertex. You can always check the size of your vertex in bytes using the `ID3DXBaseMesh::GetNumBytesPerVertex()` function.

Here now follows the giant code snippet that has been explained in this and the previous sections. The upcoming `CreateDecalMesh()` function has been added to the `BoneMesh` class to calculate a skinned decal mesh taking a ray origin, ray direction, and decal size as parameters:

```
ID3DXMesh* BoneMesh::CreateDecalMesh(
                D3DXVECTOR3 &rayOrg,
                D3DXVECTOR3 &rayDir,
                float decalSize)
{
    //Only supports skinned meshes for now
    if(pSkinInfo == NULL)
        return NULL;

    D3DXINTERSECTINFO hitInfo = GetFace(rayOrg, rayDir);

    //No face was hit
    if(hitInfo.FaceIndex == 0xffffffff)
        return NULL;

    //Generate adjacency lookup table
    DWORD* adj = new DWORD[OriginalMesh->GetNumFaces() * 3];
    OriginalMesh->GenerateAdjacency(0.001f, adj);

    //Get vertex and index buffer of temp mesh
    Vertex *v = NULL;
    WORD *i = NULL;
    OriginalMesh->LockVertexBuffer(D3DLOCK_READONLY, (VOID**)&v);
    OriginalMesh->LockIndexBuffer(D3DLOCK_READONLY, (VOID**)&i);

    //Calculate hit position on original mesh
    WORD i1 = i[hitInfo.FaceIndex * 3 + 0];
    WORD i2 = i[hitInfo.FaceIndex * 3 + 1];
    WORD i3 = i[hitInfo.FaceIndex * 3 + 2];
    D3DXVECTOR3 hitPos;
    D3DXVec3BaryCentric(&hitPos,
                        &v[i1].position,
                        &v[i2].position,
                        &v[i3].position,
                        hitInfo.U,
                        hitInfo.V);
```

```
//Find adjacent faces within range of hit location
queue<WORD> openFaces;
map<WORD, bool> decalFaces;

//Add first face
openFaces.push((WORD)hitInfo.FaceIndex);

while(!openFaces.empty())
{
    //Get first face
    WORD face = openFaces.front();
    openFaces.pop();

    //Get triangle data for open face
    WORD i1 = i[face * 3 + 0];
    WORD i2 = i[face * 3 + 1];
    WORD i3 = i[face * 3 + 2];
    D3DXVECTOR3 &v1 = v[i1].position;
    D3DXVECTOR3 &v2 = v[i2].position;
    D3DXVECTOR3 &v3 = v[i3].position;

    float testSize = max(decalSize, 0.1f);

    //Should this face be added?
    if(D3DXVec3Length(&(hitPos - v1)) < testSize ||
       D3DXVec3Length(&(hitPos - v2)) < testSize ||
       D3DXVec3Length(&(hitPos - v3)) < testSize ||
       decalFaces.empty())
    {
        decalFaces[face] = true;

        //Add adjacent faces to open queue
        for(int a=0; a<3; a++)
        {
            DWORD adjFace = adj[face * 3 + a];

            if(adjFace != 0xffffffff)
            {
                //Check that it hasn't been added to decal faces
                if(decalFaces.count((WORD)adjFace) == 0)
                    openFaces.push((WORD)adjFace);
```

```
            }
        }
    }
}

OriginalMesh->UnlockIndexBuffer();
OriginalMesh->UnlockVertexBuffer();

//Create decal mesh
ID3DXMesh* decalMesh = NULL;

//No faces to create decal with
if(decalFaces.empty())
    return NULL;

//Create a new mesh from selected faces
D3DVERTEXELEMENT9 decl[MAX_FVF_DECL_SIZE];
MeshData.pMesh->GetDeclaration(decl);

D3DXCreateMesh((int)decalFaces.size(),
               (int)decalFaces.size() * 3,
               D3DXMESH_MANAGED,
               decl,
               g_pDevice,
               &decalMesh);

//Lock dest & src buffers
DecalVertex* vDest = NULL;
WORD* iDest = NULL;
DecalVertex* vSrc = NULL;
WORD* iSrc = NULL;
decalMesh->LockVertexBuffer(0, (VOID**)&vDest);
decalMesh->LockIndexBuffer(0, (VOID**)&iDest);
MeshData.pMesh->LockVertexBuffer(D3DLOCK_READONLY, (VOID**)&vSrc);
MeshData.pMesh->LockIndexBuffer(D3DLOCK_READONLY, (VOID**)&iSrc);

//Iterate through all faces in the decalFaces map
map<WORD, bool>::iterator f;
int index = 0;
for(f=decalFaces.begin(); f!=decalFaces.end(); f++)
{
    WORD faceIndex = (*f).first;
```

```
            //Copy vertex data
            vDest[index * 3 + 0] = vSrc[iSrc[faceIndex * 3 + 0]];
            vDest[index * 3 + 1] = vSrc[iSrc[faceIndex * 3 + 1]];
            vDest[index * 3 + 2] = vSrc[iSrc[faceIndex * 3 + 2]];

            //Create indices
            iDest[index * 3 + 0] = index * 3 + 0;
            iDest[index * 3 + 1] = index * 3 + 1;
            iDest[index * 3 + 2] = index * 3 + 2;

            index++;
        }

        //Unlock buffers
        decalMesh->UnlockIndexBuffer();
        decalMesh->UnlockVertexBuffer();
        MeshData.pMesh->UnlockIndexBuffer();
        MeshData.pMesh->UnlockIndexBuffer();

        return decalMesh;
    }
```

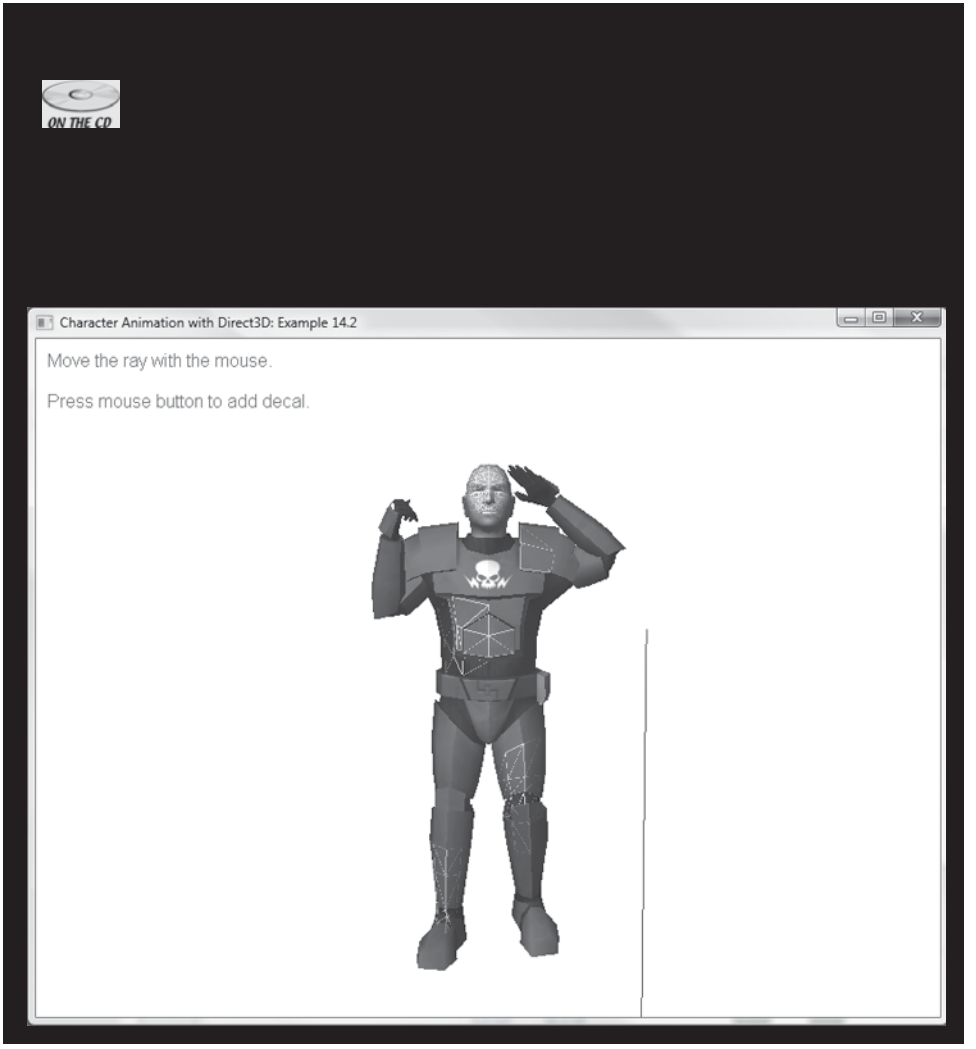
This code performs all the steps covered so far about how to create the decal mesh. There is nothing really difficult about this piece of code that requires more explaining. The one thing I could mention is the way I do the flood-fill out from the triangle hit by the ray. I create one queue of faces that needs to be considered for the decal mesh called `openFaces`. I also create a map of faces that *have* been selected to be included in the decal mesh called decalFaces. The reason for having a map instead of a regular vector is that it is quicker to look up whether or not a certain face has already been added to the set of decal faces using a map. If a face is added to the `decalFaces` map, I also add the neighbors of this face to the `openFaces` queue for future consideration. Once the `openFaces` queue is empty, I know that all connected faces within the decal size radius have been considered and no time has been wasted on unnecessary faces.

The CharacterDecal Class

To store and render the decal, I've created a very simple class called `CharacterDecal`. The class is defined very simply as follows:

```
class CharacterDecal
{
    public:
        CharacterDecal(ID3DXMesh* pDecalMesh);
        ~CharacterDecal();
        void Render();

    public:
        ID3DXMesh* m_pDecalMesh;
};
```

Note that to this class you can add all sorts of information needed to vary your decals. The most obvious addition is that of individual textures for your decals. You can create a lot of variation by simply having a vector of different textures from which you randomly assign a texture once a new decal is created. Other things you can add are varying alpha value or color information, which you can pass to the decal vertex and pixel shader. I've also added a vector of `CharacterDecal` objects to the `BoneMesh` class, which will be rendered after the bone mesh itself has been rendered (remember not to change the matrix palette between rendering the bone mesh and its decals). I've also added an `AddDecal()` function to the `SkinnedMesh` class that takes a ray origin and a ray direction as parameters. This function finds one or more bone meshes that intersect the ray, creates a `CharacterDecal` mesh, and adds it to the list stored in the `BoneMesh` object.



Note that there are plenty of optimizations you can do to the selection of the decal mesh. An obvious optimization is, of course, to pre-compute as much as possible. One easy example of something that could be pre-computed is the adjacency information for the character mesh (since this is valid for all instances of a character and will be used multiple times).

Also, I've been using a bounding sphere to test which faces and vertices should be included in the decal mesh. Although this is accurate enough, it is still just a simplification, as shown in Figure 14.10.

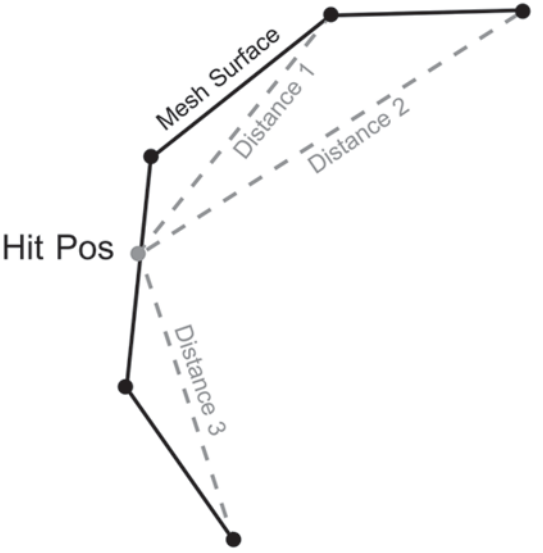


FIGURE 14.10
The real distance between a vertex and the hit point.

I am then comparing the distance from the ray hit position to the vertex positions as a straight line when I really should be comparing the distance over the mesh. Implementing this, though, takes some more work and is again something I leave for you to do on a rainy day when you've run out of more sensible things to do.

Calculating Decal UV Coordinates

The hard part is behind you. You got a skinned decal mesh attached to the character and you have a way of adding these to a character. The last remaining piece of the puzzle is to calculate the UV coordinates of the decal mesh and then render it. Figure 14.11 shows an image of a run-of-the-mill decal texture.

FIGURE 14.11
Decal texture with alpha (alpha visualized with square pattern).

The most important thing about the decal texture is that all the edges are 100% transparent. If not, any pixel on the edge will be stretched out when applied to the decal mesh. This is because you will be rendering the decal mesh with clamped UV coordinates (i.e., clamped to the range of zero to one). Figure 14.12 shows how you'll apply the decal texture to the decal mesh.

The dotted square represents the area that will be covered by the decal texture. Note the UV coordinates for the virtual square corners. The area outside the texture will be rendered completely transparent (since the edge pixels of the decal texture will be stretched). Because there aren't actually any vertices placed exactly where you want the decal to go, you have to calculate the UV coordinates of all the other vertices in the decal mesh so that the UV coordinates for the virtual decal square corners form the square shown in Figure 14.12.

To do this you take the normal of the triangle hit by the ray and use it to create an *up* and *right* vector. First, you set the *up* vector to point straight up in the world (for example), and then you calculate the *right* vector as the cross product between the triangle normal and the *up* vector. Finally, you recalculate the *up* vector as the cross product between the *right* vector and the triangle normal.

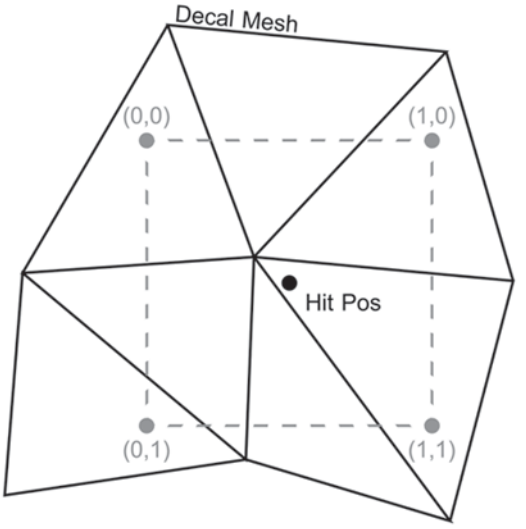


FIGURE 14.12
Calculating the UV coordinates for the decal mesh.

```
D3DXVECTOR3 up(0.0f, 1.0f, 0.0f);
D3DXVECTOR3 right;

//Calculate the right vector
D3DXVec3Cross(&right, &faceNormal, &up);
D3DXVec3Normalize(&right, &right);

//Calculate up vector
D3DXVec3Cross(&up, &faceNormal, &right);
D3DXVec3Normalize(&up, &up);
```

With these two vectors we can calculate the following vectors:

```
D3DXVECTOR3 decalCorner, UCompare, VCompare;

decalCorner = (hitPos - right * decalSize - up * decalSize);

UCompare = -right * decalSize * 2.0f;
VCompare = -up * decalSize * 2.0f;
```

These vectors are shown and explained in Figure 14.13.

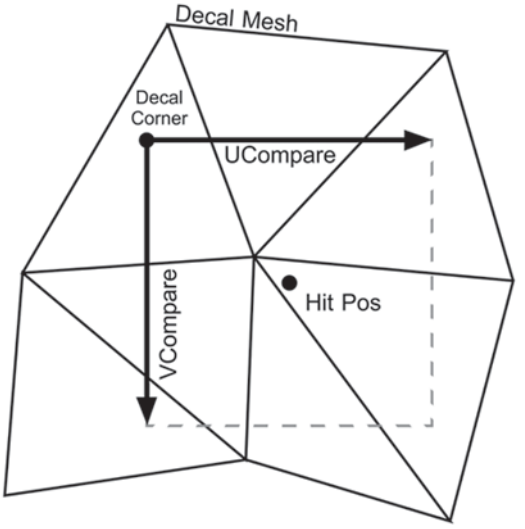


FIGURE 14.13
Decal mesh with decal corner, U and V compare vectors.

Now, to calculate the UV coordinates of all the vertices in the decal mesh, you simply take the difference between the vertex position and the decal corner. Then, project the X coordinate of this delta vector to the `UCompare` vector and vice versa with the Y coordinate and the `VCompare` vector. In other words, the UV coordinates of the vertices are projected to the plane of the decal. I've added the `CalculateDecalUV()` function to the `BoneMesh` class, which will calculate the decal UV coordinates based on the hit position of the ray:

```
void BoneMesh::CalculateDecalUV(
          ID3DXMesh* decalMesh,
          D3DXVECTOR3 &hitPos,
          float decalSize)
{
    DecalVertex *v = NULL;
    decalMesh->LockVertexBuffer(0, (VOID**)&v);

    //Get hit normal (first 3 vertices make up the hit triangle)
    DecalVertex &v1 = v[0];
```

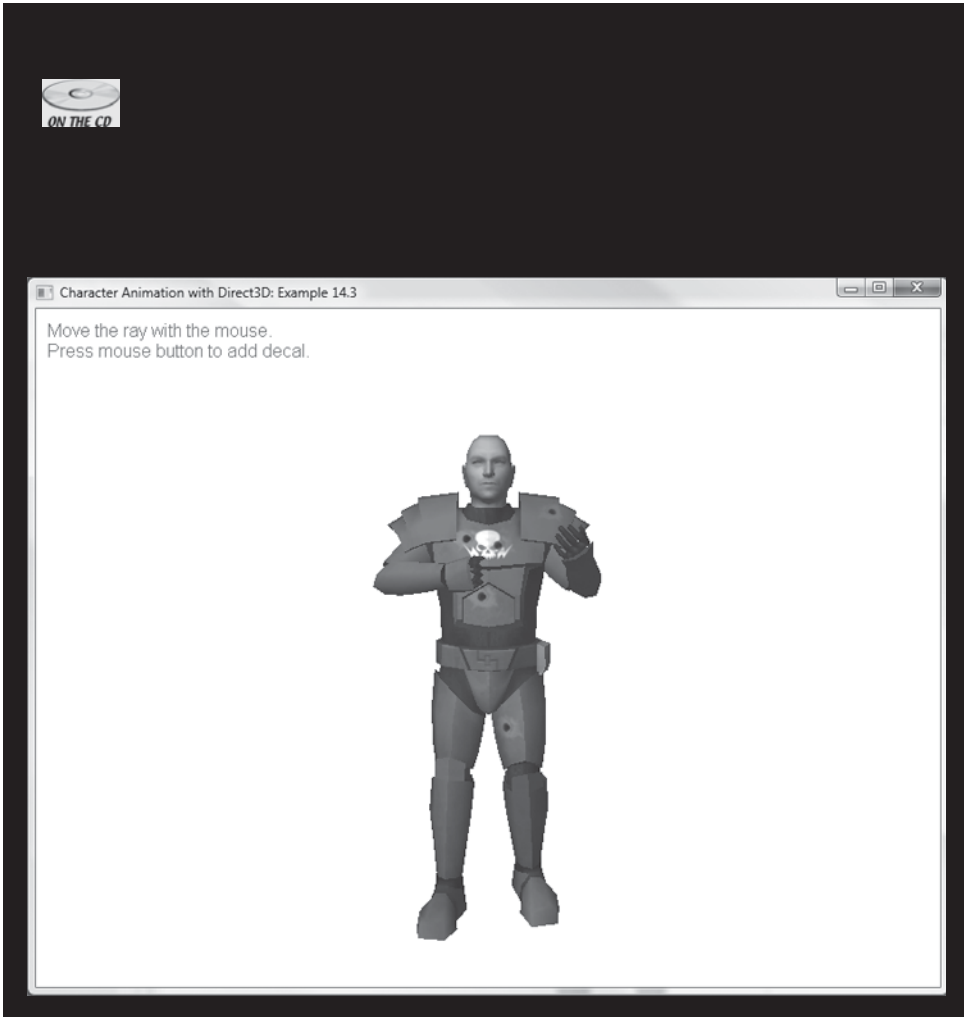
```
    DecalVertex &v2 = v[1];
    DecalVertex &v3 = v[2];
    D3DXVECTOR3 faceNormal = (v1.normal +
                              v2.normal +
                              v3.normal) / 3.0f;
    D3DXVec3Normalize(&faceNormal, &faceNormal);

    //Calculate Right & Up vector
    D3DXVECTOR3 up(0.0f, 1.0f, 0.0f);
    D3DXVECTOR3 right;
    D3DXVec3Cross(&right, &faceNormal, &up);
    D3DXVec3Normalize(&right, &right);
    D3DXVec3Cross(&up, &faceNormal, &right);
    D3DXVec3Normalize(&up, &up);

    D3DXVECTOR3 decalCorner, UCompare, VCompare;

    decalCorner = (hitPos - right * decalSize - up * decalSize);

    UCompare = -right * decalSize * 2.0f;
    VCompare = -up * decalSize * 2.0f;

    //Loop through vertices in mesh and calculate their UV coordinates
    for(int i=0; i<(int)decalMesh->GetNumVertices(); i++)
    {
        D3DXVECTOR3 cornerToVertex = decalCorner - v[i].position;

        float U = D3DXVec3Dot(&cornerToVertex, &UCompare) /
                  (decalSize / 4.0f);

        float V = D3DXVec3Dot(&cornerToVertex, &VCompare) /
                  (decalSize / 4.0f);

        //Assign new UV coordinate to the vertex
        v[i].uv = D3DXVECTOR2(U, V);
    }

    decalMesh->UnlockVertexBuffer();
}
```

That's it. The decal mesh now has had its UV coordinates recalculated to accommodate the decal texture. To render the decal, you'll also need a slightly different texture sampler than normal. The following texture and sampler has been added to the shader code to be used by the decal mesh:

```
texture texDecal;

...

sampler DecalSampler = sampler_state
{
    Texture = (texDecal);
    MinFilter = Linear;
    MagFilter = Linear;
    MipFilter = Linear;
    AddressU  = Clamp;    //Important! Clamp UVW coordinates
    AddressV  = Clamp;
    AddressW  = Clamp;
    MaxAnisotropy = 16;
};
```

The clamping of the UVW coordinates is what will stretch the edge pixels of the decal texture, making sure the decal only appears once in the middle of the decal mesh. Figure 14.14 shows a close-up of the Soldier rendered in real time with some decals applied.

FIGURE 14.14
Decals in use.



I've tried to divide the creation of the decals into logical sections and thus into different functions. You can, of course, do some optimizations by combining these functions into one giant function. By doing so you can reduce the number of locks required to some of the vertex and index buffers. I leave that for you, however.

There is one downside to the way the UV coordinates of the decal mesh are calculated in this example. You will probably notice the problem when you place a decal on a curved surface. Since the UV coordinates are calculated from the plane of the triangle hit by the ray, this results in some clearly visible stretching of the decal texture. Figure 14.15 demonstrates this problem.

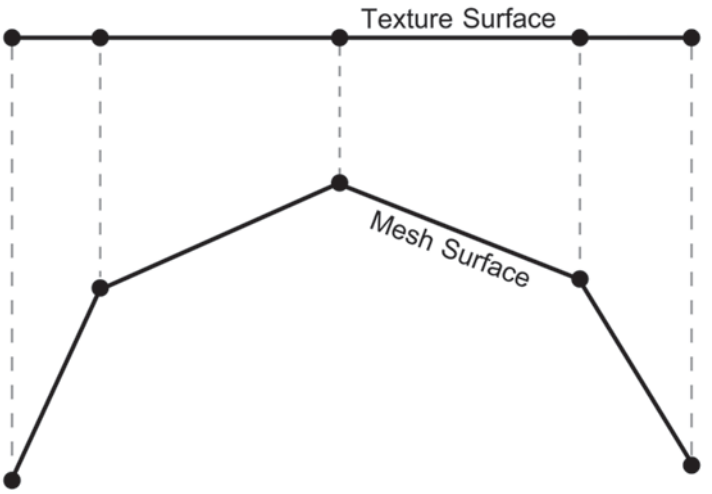


FIGURE 14.15
Decal UV coordinates over a curved surface.

Because I'm using a plane (calculated from the hit position and the triangle normal) to calculate the UV coordinates of the decal, there will be some stretching when the decal is applied to curved surfaces. You can see this in Figure 14.15 where the positions of the vertices in the texture surface aren't spread out uniformly (even though the distance on the mesh surface between the vertices is).

This is a problem related to that of using the straight line distance from the hit to the vertex described earlier. If you can calculate the actual distance over the mesh surface from the hit position to the vertex, then you can also improve on the UV coordinate calculation.

Conclusions

This chapter introduced the concept of adding real-time scratches, bullet holes, etc., to your character using decals. Decals are a great way of adding extra detail or randomness to your characters without having a major impact on texture memory.

One thing I haven't covered in this chapter is how to handle a large number of decals. In a first-person shooter game, for example, the number of bullet-hole-induced decals can quickly rise to several thousands. At some point you need to start removing the decals from the scene or your frame rate will start to drop. Usually, you'll need to implement some form of decal manager that keeps track of all the decals in the world and which can remove them according to:

- Decal position (relative to the player)
- Decal age
- First-in, first-out

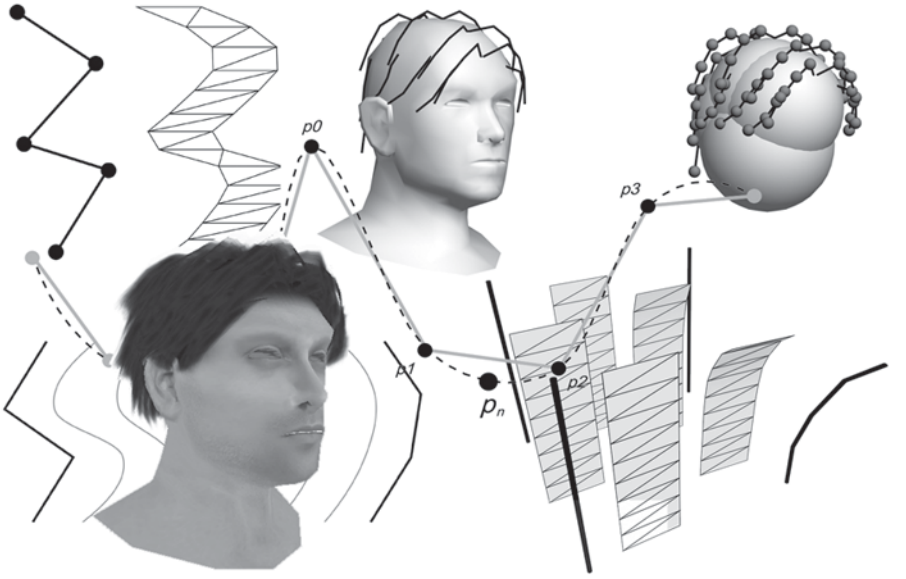
Once a decal has been flagged for removal, you can either just pop it out of existence, fade it out, or wait until the player is looking the other way and then remove it (you can easily check the decal position against the player view frustum). One big improvement to this decal system would also be to have normal-mapped decals, making it possible to add dents, etc., to the character. Hopefully you've gotten enough out of this chapter to be able to create your own decal system (with all the possible improvements mentioned in this chapter and more).

Chapter 14 Exercises

- Also create decals for static meshes (pulse rifle, helmet). Note that you need to disregard the skinning information when creating the decal mesh; otherwise, it is pretty much the same process.
- Use the D3DXIntersect method to find all intersections of a mesh. Use this also to create exit wounds of the ray.
- Create a normal-mapped wound decal.
- Create a particle system and tie it to the decal (e.g., blood pouring from a wound).
- Make a more accurate implementation of the "triangle-bounding sphere" test used to determine whether or not a triangle belongs to the decal mesh. Find the point on the triangle closest to the bounding sphere.
- Use different decals for the character's armor and the face.

This page intentionally left blank

15 Hair Animation



In this chapter I'll briefly cover the topic of dynamic hair animation. When I say "hair animation" I don't mean a simple pony tail animated with a few bones, but the generic "any-hair-style" case. This topic is closely linked to the topic of cloth animation (they share many similarities).

Hair animation is a component that most games leave out completely. The primary reason is because it is quite costly (thousands of small triangles are required, which can, in most cases, give you more bang-for-your-buck somewhere else). The second reason is that it is very hard to get hair animation to look good. So, because of these issues, most developers just opt for leaving out animated hair

completely from their game characters. Usually, they stick with either a static mesh for the characters' hair or cover the heads with some form of helmets or other headwear.

The following topics will be covered in this chapter:

- Single hairs versus strips of hair
- Generating hair strips from control splines
- Animating control splines

Hair Representation

OK, so you want to have a dynamically animated haircut for your game character. Well, the first issue to solve is how to represent, or render, the hair. The two most common ways to represent hair are either as a bunch of splines or as a set of meshes (strips) with an alpha-blended hair texture (see Figure 15.1).

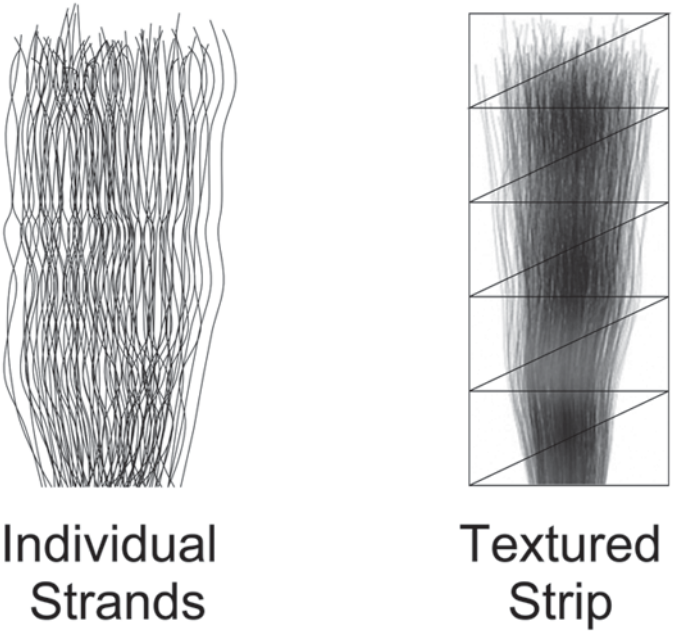


FIGURE 15.1
Individual hair strands versus hair strips.

NVIDIA has made a very nice demo of a mermaid, which renders individual hair strands as splines. Although this option may be viable in future games, it is, at the time this book was written, a far too costly option (nevertheless, I strongly recommend you check out NVIDIA's Nalu demo). You can find a really good article about the hair animation in this demo at the following URL:

http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter23.html

It seems rendering/simulating individual hair strands is a bit too expensive for today's hardware. This leaves us with the option of simulating multiple hair strands using a mesh strip and a texture.

Hair Modeling

The problem of creating the hair mesh for the character can be approached in many ways. The most straightforward approach to grasp is, of course, to do it by hand. Simply model and texture all the hair strips of the character in modeling software like 3D Studio Max, Maya, or Lightwave, etc. Then, import this hair mesh into your game somehow and animate and render it. This is, of course, a rather expensive process and one that requires considerable artistic talent (something that, not surprisingly, most programmers lack). It also means that for each new haircut you will have to do all the work over again from scratch. So, instead, let's try to make a system for this and "grow" hair for our characters in code.

This will be done by importing just a few splines (aka control splines) from some modeling software and then generating the hairs (either strands or strips) between the control splines in code. Figure 15.2 shows an example of this.

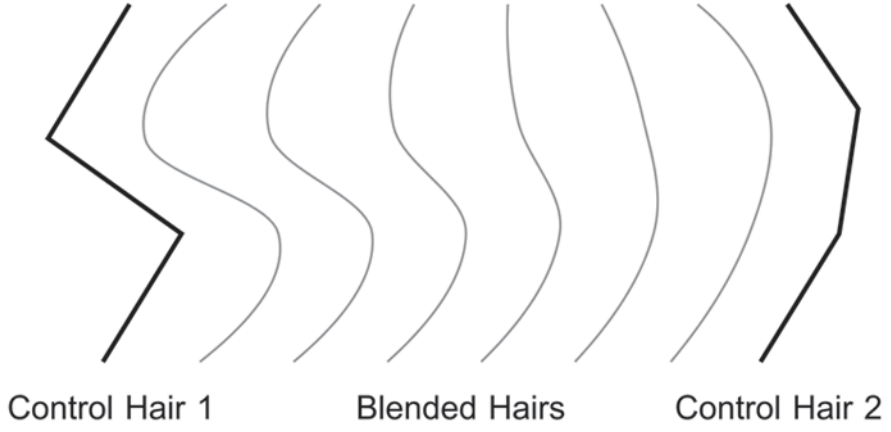


FIGURE 15.2
Example of control hairs and interpolated hairs.

Even though Figure 15.2 only shows how this could be used to generate hair strands, you can easily use this technique to generate strips as well. In either case, this technique lets you (or your artist) quickly create or shape new haircuts without all the manual polygonal modeling and texturing.

The Control Hair Class

Let's start by looking at a single control hair. The control hair consists of a list of points that define the location of the control hair. Since the control hairs will be used only for the growing of the more detailed hair strips and to update the hair simulation, it is good to keep the number of points used in the control hair to a minimum. Just because the control hair has a low level of detail (i.e., few points) doesn't mean the final hair strips will, since these are cubically interpolated from the points in the control hair, as shown in Figure 15.3.

This is a great thing, because it gives you complete control over the amount of polygons and detail you have for your haircuts (as opposed to pre-created hair meshes). To represent a control hair, I've created the following class:

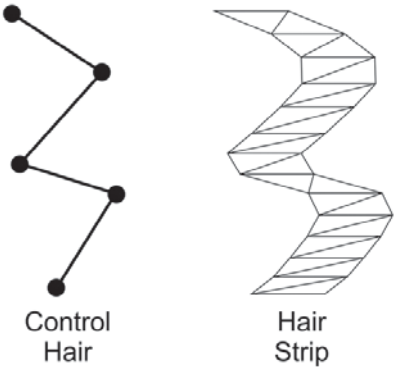


FIGURE 15.3
Control hair and generated strip.

```
class ControlHair
{
    public:
        ControlHair();

        float GetSegmentPercent(float prc);
        pair<int, int> GetBlendIndices(float prc);
        D3DXVECTOR3 GetBlendedPoint(float prc);
```

```
        void Render();

    public:
        vector<D3DXVECTOR3> m_points;
};
```

As said before, the control hair just consists of a list of control points. The `GetSegmentPercent()`, `GetBlendIndices()`, and `GetBlendedPoint()` helper functions are used when calculating the cubically blended spline. These three functions take a percentage float value in the range 0 to 1, where 0 is the start of the hair and 1 is the end point of the hair. Figure 15.4 shows how these three functions are used.

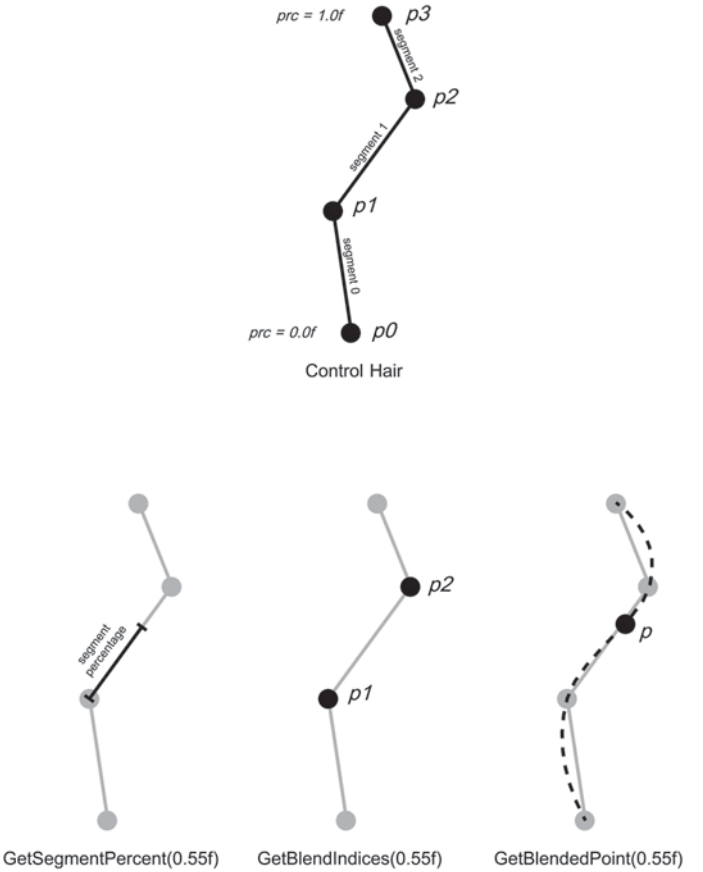


FIGURE 15.4
The control hair helper functions.

Figure 15.4 shows the result if you pass 55% (or 0.55f) to these three helper functions. The first helper function, `GetSegmentPercent()`, converts a percentage value from the whole hair to a percentage value between two points of the hair. So first you need to calculate between which two points the value lies and then calculate the new percentage value as shown in code here:

```
float ControlHair::GetSegmentPercent(float prc)
{
    //Calculate percentage step between two points in the control hair
    float step = 1.0f / (float)(m_points.size() - 1);

    int numSteps = (int)(prc / step);

    //Convert prc to the [0-1] range of the segment
    return (prc - numSteps * step) / step;
}
```

Next is the `GetBlendIndices()` function. This function retrieves the two indices of the segment described by a certain hair percentage value as shown here:

```
pair<int, int> ControlHair::GetBlendIndices(float prc)
{
    //Less than zero
    if(prc <= 0.0f)
        return pair<int, int>(0, 0);

    //Greater than one
    if(prc >= 1.0f)
        return pair<int, int>(
         (int)m_points.size() - 1,
         (int)m_points.size() - 1);

    //Get first segment index
    int index1 = (int)(prc * (m_points.size() - 1));

    //Get second segment index (no greater than num points - 1)
    int index2 = min(index1 + 1, (int)m_points.size() - 1);

    return pair<int, int>(index1, index2);
}
```

This function simply figures out which two points define the segment on which a certain percentage value lies. This information must be known when you attempt to blend between two points, which coincidentally is something that the next helper function does. To get smooth blending between the control points (as opposed to basic linear interpolation, which would produce very blocky hairs), I use cubic interpolation for the hair strips. Consider the points shown in Figure 15.5.

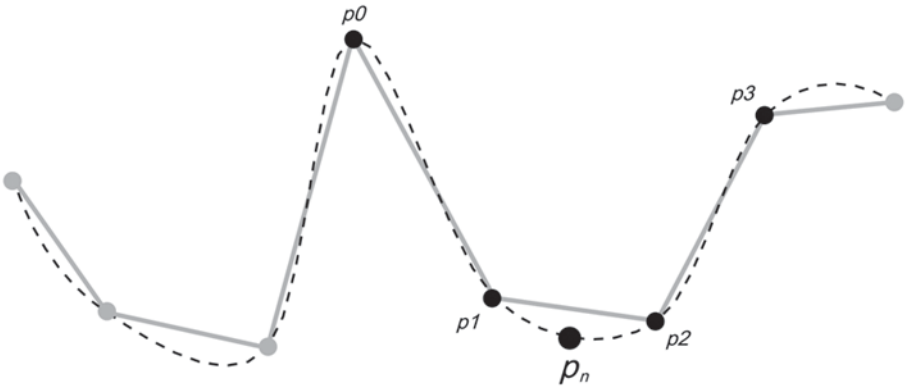


FIGURE 15.5
Cubic interpolation.

A cubically interpolated point (p_n) needs to take the four neighboring points (p_1, p_2, p_3 and p_4) into consideration. The point p_n can be calculated according to the following formula, where t is the percentage between the points p_1 and p_2:

$$P = (p_3 - p_2) - (p_0 - p_1)$$
$$Q = (p_0 - p_1) - P$$
$$R = (p_2 - p_0)$$
$$S = p_1$$
$$p_n = P \times t^3 + Q \times t^2 + R \times t + S$$

This formula is used in the `GetBlendedPoint()` function, which makes use of the two previous helper functions to return a cubically blended point along the control hair:

```
D3DXVECTOR3 ControlHair::GetBlendedPoint(float prc)
{
    //Get two affected indices
    pair<int, int> indices = GetBlendIndices(prc);

    //Get the four point indices
    int index0 = max(indices.first - 1, 0);
```

```
    int index1 = indices.first;
    int index2 = indices.second;
    int index3 = min(indices.second + 1, (int)m_points.size() - 1);

    //Get segment percentage
    float t = GetSegmentPercent(prc);

    //Perform the cubic interpolation
    D3DXVECTOR3 P = (m_points[index3] - m_points[index2]) –
                    (m_points[index0] - m_points[index1]);
    D3DXVECTOR3 Q = (m_points[index0] - m_points[index1]) - P;
    D3DXVECTOR3 R = m_points[index2] - m_points[index0];
    D3DXVECTOR3 S = m_points[index1];

    return (P * t * t * t) +
           (Q * t * t) +
           (R * t) +
            S;
}
```

That about covers the control hair. Later on, when you simulate the hair, all you need to do is update the points in your control hairs and the rest of the haircut will follow suit. The next step is to actually create the strips from these control hairs. After that you'll have something "hair-ish" that can be rendered onto the screen.

The HairPatch Class

The naïve way of implementing the hair strips would be to have a single mesh for each of the hair strips and perhaps use some form of skinning to implement an animated hair strip. However, this is very inefficient! You're bound to have hundreds of hair strips per haircut, so to improve performance you need to bundle strips together in one mesh to reduce the number of draw calls necessary to render the hair. So for this purpose I've created the `HairPatch` class, which implements a patch of hair defined as the area between four control hairs. Each of the four control hairs marks one corner of the "squarish" area of the hair patch. Figure 15.6 shows how the hair patch will be built.

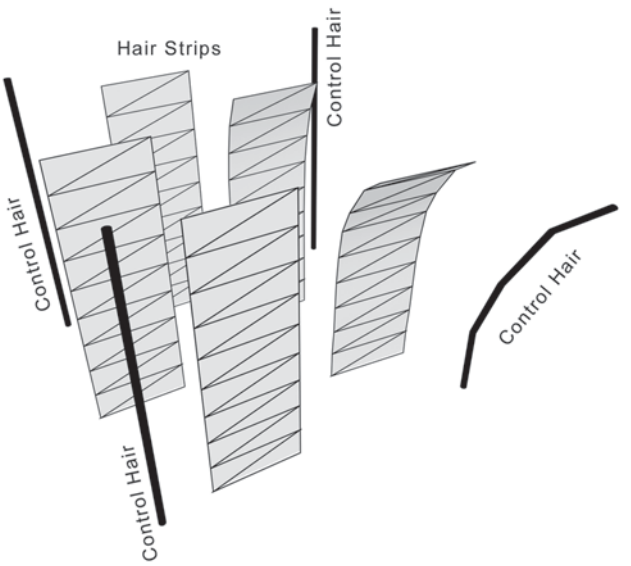


FIGURE 15.6
The hair patch.

Note that all the strips in Figure 15.6 belong to the same mesh. The idea is that as one of the control hairs bends or animates, the hair strips close to this control hair will be influenced and bend in a similar manner. A strip placed exactly in the middle of the hair patch will be influenced equally by the four control hairs. On the other hand, a strip placed in the exact same spot as a control hair will only be influenced by this control hair and no others.

```
class HairPatch
{
public:
    HairPatch(ControlHair* pCH1,
              ControlHair* pCH2,
              ControlHair* pCH3,
              ControlHair* pCH4);
    ~HairPatch();
    D3DXVECTOR3 GetBlendedPoint(D3DXVECTOR2 pos, float prc);

    HairVertex GetBlendedVertex(D3DXVECTOR2 pos,
                                float prc,
                                bool oddVertex);
```

```
    vector<D3DXVECTOR2> GetStripPlacements(float sizePerHairStrip);

    void CreateHairStrips(int numSegments,
                          float sizePerHairStrip,
                          float stripSize);
    void Render();

public:
    ID3DXMesh* m_pHairMesh;
    ControlHair* m_controlHairs[4];
};
```

The `HairPatch` class keeps four pointers to control hairs (which are set in the constructor) as well as an `ID3DXMesh` object that the class is responsible for creating and rendering. The `GetBlendedPoint()` function returns a point anywhere on the patch along a certain percentage (again, zero being the base of the skull and one being the tip of the hairs):

```
D3DXVECTOR3 HairPatch::GetBlendedPoint(D3DXVECTOR2 pos, float prc)
{
    //Get blended points along the control hairs (for this prc)
    D3DXVECTOR3 p1 = m_controlHairs[0]->GetBlendedPoint(prc);
    D3DXVECTOR3 p2 = m_controlHairs[1]->GetBlendedPoint(prc);
    D3DXVECTOR3 p3 = m_controlHairs[2]->GetBlendedPoint(prc);
    D3DXVECTOR3 p4 = m_controlHairs[3]->GetBlendedPoint(prc);

    //Perform a linear 2D blend
    D3DXVECTOR3 blendedX1 = p2 * pos.x + p1 * (1.0f - pos.x);
    D3DXVECTOR3 blendedX2 = p3 * pos.x + p4 * (1.0f - pos.x);

    return blendedX2 * pos.y + blendedX1 * (1.0f - pos.y);
}
```

First you retrieve the blended position from each of the four control hairs (for the desired percentage). Then you need to perform a 2D blend depending on the hairs' position on the patch (i.e., the closer a hair is to one of the control hairs, the more this control hair will influence it). Now that you can calculate a point anywhere on the hair patch it is quite simple to start generating the hair strips.

Growing the Hair

Before you actually create the hair strips, there are some questions that need answering. First is how detailed you want to make the strips (i.e., how many faces/vertices per strip), and second is how tightly you want to pack the strips. Here again is where you enter the gray zone of performance versus what looks good. Most likely this will differ from one situation to the next, depending on what performance requirements the game you're working on has. However, with this way of growing the hair dynamically you can even plug in these two values in an LOD system. Simply increase the number of strips and the detail of these the closer to the camera a character is.

Let's start with the simpler of the two problems—namely, where to place the strips. One way is, of course, to place them uniformly by creating a uniform grid between the four control hairs and placing a hair strip at each point of this grid. This tends to create a bit too-regular-looking hair patch. You can easily get a better looking hair patch using fewer strips by simply placing them at random. One important restriction is to make sure that no two strips are placed too close to each other. The `GetStripPlacements()` function in the `HairPatch` class implements this:

```
vector<D3DXVECTOR2> HairPatch::GetStripPlacements(
                                float sizePerHairStrip)
{
    //Place hair strips at random
    vector<D3DXVECTOR2> strips;
    for(int i=0; i<200; i++)
    {
        //Create random hair position
        D3DXVECTOR2 hairPos = D3DXVECTOR2((rand()%1000) / 1000.0f,
                                          (rand()%1000) / 1000.0f);

        //Check that this hair isn't too close to another hair
        bool valid = true;
        for(int h=0; h<(int)strips.size() && valid; h++)
        {
            D3DXVECTOR3 diff = hairPos - strips[h];
            if(D3DXVec2Length(&diff) < sizePerHairStrip)
                valid = false;
        }
```

```
            //Add hair if valid
            if(valid)
                strips.push_back(hairPos);
        }

        //Order strips for correct alpha blending
        for(int i=0; i<(int)strips.size(); i++)
        {
            for(int j=i+1; j<(int)strips.size(); j++)
            {
                if(strips[j].y < strips[i].y)
                {
                    D3DXVECTOR2 temp = strips[i];
                    strips[i] = strips[j];
                    strips[j] = temp;
                }
            }
        }

        return strips;
    }
```

This piece of code simply tries to place an arbitrary number of hair strips (in this case 200) in the hair patch. The `sizePerHairStrip` parameter determines how closely packed the hair strips are allowed to be. After the strips have been placed, they are ordered for correct alpha blending. Later, when you have a head full of hair, it makes sense to order the hair strips according to how far they are from the skull. In the later examples, when you have multiple patches of hair, these will have to be ordered in real time as well for correct alpha blending.

The `GetStripPlacement()` returns a vector of 2D vectors (confusing wording, I know). These 2D points have an X and Y value in the range of 0 to 1, which then determines how much a certain control hair influences this strip. Next up is the `CreateHairStrips()` function that creates the mesh object and fills it up with all the hair strips:

```
    void HairPatch::CreateHairStrips(int numSegments,
                                     float sizePerHairStrip,
                                     float stripSize)
    {
        //Get random hair strip positions
        vector<D3DXVECTOR2> strips = GetStripPlacements(sizePerHairStrip);
```

```
//Create a mesh containing all strips
int numFacesPerStrip = numSegments * 2;
int numVertsPerStrip = (numSegments + 1) * 2;

D3DXCreateMesh(numFacesPerStrip * (int)strips.size(),
               numVertsPerStrip * (int)strips.size(),
               D3DXMESH_MANAGED,
               hairVertexDecl,
               g_pDevice,
               &m_pHairMesh);

HairVertex *vb = NULL;
WORD *ib = NULL;
m_pHairMesh->LockVertexBuffer(0, (void**)&vb);
m_pHairMesh->LockIndexBuffer(0, (void**)&ib);

int vIndex = 0;
int iIndex = 0;

//Create hair strips
for(int i=0; i<(int)strips.size(); i++)
{
    int startVertIndex = vIndex;

    //Angle the strip randomly
    float rAngle = ((rand()%1000) / 1000.0f) * 2.0f - 1.0f;
    D3DXVECTOR2 rOffset = D3DXVECTOR2(cos(rAngle),
                                      sin(rAngle)) * stripSize;

    //Create vertices
    for(int s=0; s<=numSegments; s++)
    {
        float prc = s / (float)numSegments;
        vb[vIndex++] = GetBlendedVertex(strips[i] + rOffset,
                                        prc,
                                        false);

        vb[vIndex++] = GetBlendedVertex(strips[i] - rOffset,
                                        prc,
                                        true);
    }
```

```
            //Create indices
            for(int s=0; s<numSegments; s++)
            {
                //Tri 1
                ib[iIndex++] = startVertIndex + s * 2 + 0;
                ib[iIndex++] = startVertIndex + s * 2 + 3;
                ib[iIndex++] = startVertIndex + s * 2 + 1;

                //Tri 2
                ib[iIndex++] = startVertIndex + s * 2 + 0;
                ib[iIndex++] = startVertIndex + s * 2 + 2;
                ib[iIndex++] = startVertIndex + s * 2 + 3;
            }
        }

        m_pHairMesh->UnlockVertexBuffer();
        m_pHairMesh->UnlockIndexBuffer();
    }
```

This function takes the amount of segments in the hair, along with the size of the strips and their spread, as parameters. It creates the single mesh of the hair patch, locks the vertex and index buffer, and starts filling it with data. Note that I give the different hair strips a random angle (something which makes the hair look better when viewed from different angles). The `GetBlendedVertex()` function simply creates and returns a vertex for the specified 2D position of the strip and the percentage value (skull to hair tip). That is all you need to know about how the mesh of the hair patch is grown.

Rendering the Hair Patch

Rendering the hair mesh should be straightforward, shouldn't it? Well, not quite. So far I haven't talked about how I'm going to animate the hair. You could, of course, animate the control hairs, then lock the vertex buffer of the hair patch and loop over all its vertices and update their positions. Although this would certainly work, it's not quite optimal. Let's instead update all the vertex positions on the fly as we're rendering them on the GPU. For this to happen, you need to work some serious magic. Instead of having the position of a hair vertex in object space, I'll fill the vertex data with some custom information that will let the GPU calculate the animated position of the vertex on the fly. This means you have to perform the cubic interpolation for each of the four control hairs, and then blend the result linearly in 2D, all in shader code. To do this the shader needs the following information:

The positions for all control hair points

The 2D blend position of the vertex (range 0 to 1)

The percentage value of the vertex (range 0 to 1)

The UV coordinates of the vertex

In theory, this is all that is needed. However, since some things don't change as you render the hair, you can easily pre-compute the following information in the vertex data and supply the shader with it to save some instructions:

Start and end blend index of the active segment

Segment percentage

So to contain all this rather cryptic data, I've created the following vertex structure and declaration:

```
struct HairVertex{
    D3DXVECTOR3 position;
    Byte blendindices[4];
    D3DXVECTOR3 normal;
    D3DXVECTOR2 uv;
};

...

//Hair Vertex Declaration
D3DVERTEXELEMENT9 hairVertexDecl[] =
{
    {0,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_POSITION, 0},
    {0, 12, D3DDECLTYPE_UBYTE4, D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_BLENDINDICES, 0},
    {0, 16, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_NORMAL,   0},
    {0, 28, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_TEXCOORD, 0},
    D3DDECL_END()
};
```

You may remember from the previous section how the vertex buffer of the hair patch mesh was filled using the `GetBlendedVertex()` function. This function just creates a `HairVertex` object and returns it:

```
HairVertex HairPatch::GetBlendedVertex(D3DXVECTOR2 pos, float prc, bool
oddVertex)
{
    //Create a new hair vertex
    HairVertex hv;
    memset(&hv, 0, sizeof(HairVertex));

    //Set vertex data
    hv.position = D3DXVECTOR3(
                  pos.x,
                  pos.y,
                  m_controlHairs[0]->GetSegmentPercent(prc));

    hv.normal = D3DXVECTOR3(0.0f, 1.0f, 0.0f);
    hv.uv = D3DXVECTOR2(oddVertex ? 0.0f : 1.0f, min(prc, 0.99f));

    //Get segment indices
    pair<int, int> indices = m_controlHairs[0]->GetBlendIndices(prc);
    hv.blendindices[0] = indices.first;
    hv.blendindices[1] = indices.second;

    return hv;
}
```

Next, you need to process this rather special vertex data in the shader. Since the control hairs are updated for each frame when they are animated, you need to also upload the positions of all the points in the control hairs. This can be done in much the same way you uploaded the bone matrices in the skinned mesh examples. I've added a simple control hair table to the shader:

```
extern float3 ControlHairTable[20];
```

The points are stored in the following order: Control hair 1 – Point 1, Point 2 … Control hair 2 – Point 1, Point 2 … etc. So to get the second point of the third hair, you'd simply look it up from the `ControlHairTable` like this:

```
point_3_2 = ControlHairTable[3 * numPointsPerHair + 2];
```

Some of the blending functionality stored in the `ControlHair` class also needs to be implemented in shader code. For instance, to do the cubic interpolation with the help of the `ControlHairTable` in the vertex shader, I've created the following HLSL helper function:

```
float3 GetHairPos(int hair, int index1, int index2, float prc)
{
    //Calculate index 0 & 3
    int index0 = max(index1 - 1, 0);
    int index3 = min(index2 + 1, numPointsPerHair - 1);

    //Offset index to correct hair in ControlHairTable
    index0 += hair * numPointsPerHair;
    index1 += hair * numPointsPerHair;
    index2 += hair * numPointsPerHair;
    index3 += hair * numPointsPerHair;

    //Perform cubic interpolation
    float3 P =(ControlHairTable[index3]-ControlHairTable[index2]) –
              (ControlHairTable[index0]-ControlHairTable[index1]);
    float3 Q =(ControlHairTable[index0]-ControlHairTable[index1]) - P;
    float3 R = ControlHairTable[index2]-ControlHairTable[index0];
    float3 S = ControlHairTable[index1];

    return (P * prc * prc * prc) +
           (Q * prc * prc) +
           (R * prc) +
            S;
}
```

The hair parameter is an index to which of the four control hairs of the patch to use; index1and index2 are the indices to the points in the hair to blend between (from which index0 and index3 are calculated). The prc parameter is then the segment percentage, or, in other words, how far between the two points the target point lies. This function is then used by the vertex shader to interpret the special vertex data we're passing in:

```
//Hair Vertex Shader
VS_OUTPUT vs_hair(VS_INPUT_HAIR IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;
```

```
    //Get position from the four control hairs
    float3 ch1 = GetHairPos(0, IN.hairIndices[0],
                            IN.hairIndices[1], IN.position.z);
    float3 ch2 = GetHairPos(1, IN.hairIndices[0],
                            IN.hairIndices[1], IN.position.z);
    float3 ch3 = GetHairPos(2, IN.hairIndices[0],
                            IN.hairIndices[1], IN.position.z);
    float3 ch4 = GetHairPos(3, IN.hairIndices[0],
                            IN.hairIndices[1], IN.position.z);

    //Blend linearly in 2D
    float3 px1 = ch2 * IN.position.x + ch1 * (1.0f - IN.position.x);
    float3 px2 = ch3 * IN.position.x + ch4 * (1.0f - IN.position.x);

    float3 pos = px2 * IN.position.y + px1 * (1.0f - IN.position.y);

    //Transform to world coordinates
    float4 posWorld = mul(float4(pos.xyz, 1), matW);
    OUT.position = mul(posWorld, matVP);

    //Copy texture coordinates
    OUT.tex0 = IN.tex0;

    return OUT;
}
```

Here, the exact same operations are done as described earlier in the `GetBlended-Point()` function of the `HairPatch` class. The blended position of each of the four control hairs is obtained from the `ControlHairTable` using the `GetHairPos()` helper function. Note that the indices passed to the helper function have been pre-computed and stored in the vertex data. Next, the points returned from the control hairs are blended depending on the 2D position of the hair vertex. The resulting point will be in object space, so to get it to the correct position on the screen it is multiplied with the world and the view-projection matrix. As said before, the beauty of going the long way about this is that you now can update the control hairs on the CPU and the mesh in the hair patch will just follow suit "automagically."

Creating a Haircut

So how would you go about creating these 10 to 20 control hairs needed to create a decent-looking haircut for a character? Well, the most obvious way is to enter them manually as was done with the four control hairs in Example 15.1. Although it doesn't take many minutes to realize that to model a set of 3D lines with a text editor is probably not the way to go. The best way is to use 3D modeling software (Max, Maya, or whatever other flavor you prefer). Figure 15.7 shows an image of a "haircut" created in 3D Studio Max as a set of lines.

FIGURE 15.7
Control hairs used to create a haircut.

I was lucky enough to know Sami Vanhatalo (Senior Technical Artist at Remedy Entertainment) who was kind enough to write an exporter for 3D Studio Max for me. With this exporter it becomes very easy to dump a set of lines from 3D Studio Max to either a text or a binary file. I won't cover the exporter here in this book since it is written in Max Script and is out of the scope of this book. However, you'll find both the text and binary version of the exporter on the accompanying CD-ROM, along with detailed instructions on how to use them. For the next example I'll use the data that has been outputted from the binary exporter. The data is in Table 15.1.

TABLE 15.1 THE BINARY HAIR FORMAT

The file structure is very simple and doesn't contain any superfluous information. Feel free to modify the exporter according to your own needs. In any case, to read a set of lines from a binary file with this format, I've created the `LoadHair()` function to the `Hair` class. The following code segment is an excerpt from this function showing how to read one of these haircut data files:

```
ifstream in(hairFile, ios::binary);

if(!in.good())
    return;

//Version number
long version;
in.read((char*)&version, sizeof(long));

//Number splines
long numSplines = 0;
in.read((char*)&numSplines, sizeof(long));
```

```
//Read splines
for(int i=0; i<numSplines; i++)
{
    ControlHair* ch = new ControlHair();

    //Read points
    long numPoints = 0;
    in.read((char*)&numPoints, sizeof(long));

    for(int p=0; p<numPoints; p++)
    {
        D3DXVECTOR3 point;
        in.read((char*)&point, sizeof(D3DXVECTOR3));
        ch->AddPoint(point);
    }

    m_controlHairs.push_back(ch);
}

in.close();
```

This "pipeline" is extremely simple and doesn't have anything more than the bare essentials. However, already with something simple as as this it becomes a breeze to create and edit "spline haircuts" and export to your game.

Animating the Control Hairs

With all the work in the previous sections, you've accomplished one important thing: You've managed to greatly reduce the amount of splines or points that you need to animate. Even though you would use a very lightweight physics system to update the individual hair strips, it would, in all likelihood, still be too heavy for a real-time application. Now you can do your hair simulation on the control hairs only and in this way save a lot of effort. Next comes the final problem, which is how to animate the control hairs in a somewhat realistic and good-looking way. In Chapter 6 you had a look at creating a simple physics engine with particles and springs. Well, you don't need much more than that to create a simplified physics simulation for your control hairs. At each point of the control hair I'll place a simple bounding sphere that I will test against a set of spheres defining the shape of the character head (you could also perform collision checks between the control hair spheres, but the end result doesn't really justify the extra amount of collision checks required). Figure 15.8 shows the setup of our simple physics simulation:

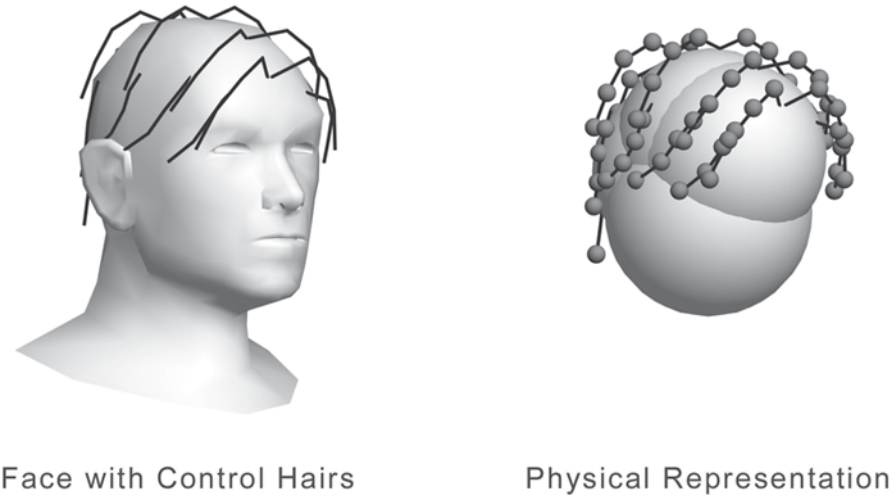


FIGURE 15.8
Physical setup of the hair animation.

I've created a very simple bounding sphere class to be used for the character head representation. The `BoundingSphere` class is defined as follows:

```
class BoundingSphere
{
    public:
        BoundingSphere(D3DXVECTOR3 pos, float radius);
        void Render();
        bool ResolveCollision(D3DXVECTOR3 &hairPos, float hairRadius);

    private:
        static ID3DXMesh* sm_pSphereMesh;

        D3DXVECTOR3 m_position;
        float m_radius;
};
```

Not so surprisingly, this class contains a position and a radius used to define the bounding sphere. The static `sm_pSphereMesh` and the `Render()` function is just for debugging and visualization purposes. The most important function in this class is the `ResolveCollision()` function. This function tests whether a point collides with the bounding sphere, and, if so, it moves the point away from the sphere until the point no longer touches the sphere:

```
bool BoundingSphere::ResolveCollision(D3DXVECTOR3 &hairPos,
                                      float hairRadius)
{
    //Difference between control hair point and sphere center
    D3DXVECTOR3 diff = hairPos - m_position;

    //Distance between points
    float dist = D3DXVec3Length(&diff);

    if(dist < (m_radius + hairRadius))
    {
        //Collision has occurred; move hair away from bounding sphere
        D3DXVec3Normalize(&diff, &diff);
        hairPos = m_position + diff * (m_radius + hairRadius);
        return true;
    }

    return false;
}
```

Once you have this function up and running and a few spheres placed to roughly represent the character head, you can start simulating in some fashion. To get a hair animation to look good, there's a whole lot of black magic needed. For instance, you can't use gravity in the same way you would for other physical simulations. If you used only gravity for simulating the hairs, the result would end up looking like a drenched cat with hair hanging straight down. In reality, haircuts tend to roughly stay in their original shape. To quickly emulate this, I've stored the original points of the control hair points as the haircut is created. Then, at run time, I have a small spring force between the current control hair's position and its original position. This will keep the haircut from deforming completely. To show a quick hair simulation in action, I've added the `UpdateSimulation()` function to the `ControlHair` class:

```
void ControlHair::UpdateSimulation(
                        float deltaTime,
                        vector<BoundingSphere> &headSpheres)
{
    const float SPRING_STRENGTH = 10.0f;
    const D3DXVECTOR3 WIND(-0.2f, 0.0f, 0.0f);

    for(int i=1; i<(int)m_points.size(); i++)
    {
        //Point's percentage along the hair
        float prc = i / (float)(m_points.size() - 1);
```

```
            D3DXVECTOR3 diff = m_originalPoints[i] - m_points[i];
            float length = D3DXVec3Length(&diff);
            D3DXVec3Normalize(&diff, &diff);

            //Update velocity of hair control point (random wind)
            float random = rand()%1000 / 1000.0f;

            D3DXVECTOR3 springForce = diff * length * SPRING_STRENGTH;
            D3DXVECTOR3 windForce = WIND * prc * random;

            m_velocities[i] += (springForce + windForce) * deltaTime;

            //Update position
            m_points[i] += m_velocities[i] * deltaTime;

            //Resolve head collisions
            for(int h=0; h<(int)headSpheres.size(); h++)
            {
                if(headSpheres[h].ResolveCollision(m_points[i], 0.01f))
                {
                    m_velocities[i] *= 0.5f;
                }
            }
        }
    }
```

In this function, each point along the control hairs is updated with a random wind force as well as a spring force toward its original location. This function also performs the collision checks between the character head representation (i.e., the head bounding spheres), making sure that the hair doesn't go through the face and thus break the illusion.

The Hair Class

Finally, to clean up all I've gone through in this chapter, I'll tie together all the classes. To do this I encapsulate all the control hairs and hair patches in a single class called `Hair`. This class is responsible for loading hair binary files, creating all the patches from the loaded control hairs, managing the hair simulation, and finally rendering the haircut. This class will also contain the hair texture used and the physical representation of the character head:

```
class Hair
{
    public:
        Hair();
        ~Hair();

        void LoadHair(const char* hairFile);

        void CreatePatch(int index1,
                         int index2,
                         int index3,
                         int index4);

        void Update(float deltaTime);
        void Render(D3DXVECTOR3 &camPos);

        int GetNumVertices();
        int GetNumFaces();

    public:
        vector<ControlHair*> m_controlHairs;
        vector<HairPatch*> m_hairPatches;
        IDirect3DTexture9* m_pHairTexture;
        vector<BoundingSphere> m_headSpheres;
};
```

One important thing to mention is that the `Render()` function in this class sorts all the hair patches in Z depth from the camera before rendering them to the screen. The `Update()` function takes care of calling the `UpdateSimulation()` function in the `ControlHair` objects, providing the `m_headSpheres` vector for the physical simulation.

If you want to extend this example, it would probably be in the `Hair` class that you would, for example, keep a pointer to the head bone of the character. Using this bone pointer you would then update the position of all the control hairs as the head moves around.


ON THE CD
Character Animation with Direct3D: Example 15.2
SPACE: Draw Debug Info
Click n Drag to rotate view
Num Vertices: 4888
Num Faces: 4512

Figure 15.9 shows a series of frames taken from the animated hair.

FIGURE 15.9
A haircut animated with the system covered in this chapter.

Conclusions

After reading this chapter, you should have a good understanding of the difficulties you're faced with when trying to implement hair animation. To increase the quality of the hair animation presented in this chapter, there are two obvious improvements. First, create a proper physical simulation instead of the "random wind" force I've used here. Second, attach the hair to the character head bone and let it inherit motion from the head of the character. When doing this you'll see some wonderful secondary motion as the character moves and then suddenly stops, etc.

Another thing I haven't covered in this chapter is the rendering of hair. There are many advanced models of how light scatters on a hair surface. A really great next stop for you is the thesis titled "Real-Time Hair Simulation and Visualization for Games" by Henrik Halen [Halen07]. You can find this thesis online using the following URL:

http://graphics.cs.lth.se/theses/projects/hair/

In the next and final chapter of this book, I'll put most of the concepts covered throughout this book into a single character class.

Chapter 15 Exercises

- Create a `Hair` class supporting multiple levels of details. Scale the number of segments used in hair strips and the number of hair strips used in total.
- Try to create a longer haircut. (This may require a more detailed physical representation of the character.)
- Attach the haircut to a moving character head.
- Implement a triangle hair patch relying on three control hairs instead of four.

Further Reading

[Halen07] Halen, Henrik, "Real-Time Hair Simulation and Visualization for Games." Available online at: http://graphics.cs.lth.se/theses/projects/hair/report.pdf, 2007.

[Jung07] Jung, Yvonne, "Real Time Rendering and Animation of Virtual Characters." Available online at http://www.ijvr.org/issues/issue4-2007/6.pdf, 2007.

[Nguyen05] Nguyen, Hubert, "Hair Animation and Rendering in the Nalu Demo." Available online at http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter23.html, 2005.

[Ward05] Ward, Kelly, "Modeling Hair Using Levels-of-Detail." Available online at: http://www.cs.unc.edu/Research/ProjectSummaries/hair05.pdf, 2005.

This page intentionally left blank

16 Putting It All Together

Welcome, dear reader, to the final chapter of this book! Throughout this book you've had a quick glance at some of the major topics and techniques needed to create a moving, talking, and falling character with Direct3D. In this chapter I won't present anything new, really, but will instead try to tie it all together and create the final glorious `Character` class. Finally, for some of the topics I have not covered in this book, I've added a few pages of discussion at the end. In this final chapter, you'll find the following:

- Mixing facial animation with skeletal animation
- The `Character` class
- Future improvements
- Alan Wake case study
- Final thoughts

Attaching the Head to the Body

So far, the facial animation examples in this book have been based on a standalone face (i.e., one of those faces not attached to a body). I guess you'll agree with me when I say that this isn't how you usually see characters in a game. So to remedy this problem, this section covers how to attach an animated head (i.e., the `Face` class) to a skinned mesh.

You've already come across this problem a bit in Chapter 8, Example 3, where a running human character was morphed into a werewolf. I'll show you in this section how to do (almost) the same thing with a character's face. Remember that the face can be generated by the `FaceFactory` class, for example, and doesn't have to look the same as the one in the actual skinned model. There is, of course, the limitation that both the source mesh (random generated face) and the target mesh (face in skinned model) must have the same amount of vertices and index buffer. Figure 16.1 shows the pieces involved.

The head of the skinned mesh (black wireframe) is just another skinned mesh with blend weights, blend indices, etc. The `Face` class, on the other hand, supports all facial animation features (and can also be generated by the `FaceFactory` class). Your job now is to replace the skinned mesh head with a `Face` object but while still keeping the skinned information. This way, the new animated face will also turn if there's some keyframe animation involving the head (or if you use some Look-At IK, and so on).

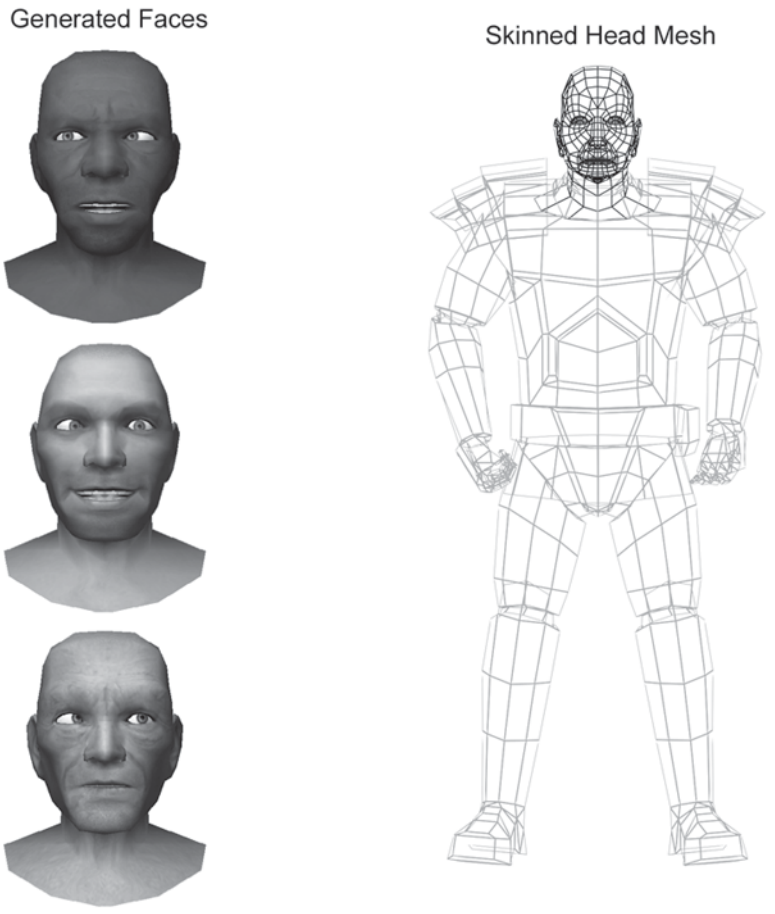


FIGURE 16.1
Attaching a face to a skinned mesh.

Okay…moving on. The first thing to do is, of course, to define the vertex declaration you'll use for the skinned and morphed face:

```
//Face Vertex Format
D3DVERTEXELEMENT9 faceVertexDecl[] =
{
//Stream 0: Base 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},

//Stream 1: Morph Target
{1,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION, 1},
{1, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_NORMAL,   1},

//Stream 2: Morph Target
{2,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION, 2},
{2, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_NORMAL,   2},

//Stream 3: Morph Target
{3,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION, 3},
{3, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_NORMAL,   3},

//Stream 4: Morph Target
{4,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION, 4},
{4, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_NORMAL,   4},

//Stream 5: Skeletal Info
{5, 12, D3DDECLTYPE_FLOAT1, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_BLENDWEIGHT, 5},
{5, 16, D3DDECLTYPE_UBYTE4, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_BLENDINDICES, 5},

D3DDECL_END()
};
```

You might recognize most of this from the first face-morphing examples. Stream 0 contains the base mesh of the face, streams 1 through 4 contain the morph targets, and finally, stream 5 contains the skinning information (bone indices and blend weights).

I've added the `SetStreamSources()` function to the `Face` class to set the stream sources of the face (streams 0–4) as well as the index buffer:

```
void Face::SetStreamSources(FaceController *pController)
{
    //Set Streams
    DWORD vSize = D3DXGetFVFVertexSize(m_pBaseMesh->GetFVF());
    IDirect3DVertexBuffer9* baseMeshBuffer = NULL;
    m_pBaseMesh->GetVertexBuffer(&baseMeshBuffer);
    pDevice->SetStreamSource(0, baseMeshBuffer, 0, vSize);

    //Set Blink Source
    IDirect3DVertexBuffer9* blinkBuffer = NULL;
    m_pBlinkMesh->GetVertexBuffer(&blinkBuffer);
    pDevice->SetStreamSource(1, blinkBuffer, 0, vSize);

    //Set Emotion Source
    IDirect3DVertexBuffer9* emotionBuffer = NULL;
    m_emotionMeshes[pController->m_emotionIndex]->
                    GetVertexBuffer(&emotionBuffer);
    pDevice->SetStreamSource(2, emotionBuffer, 0, vSize);

    //Set Speech Sources
    for(int i=0; i<2; i++)
    {
        IDirect3DVertexBuffer9* speechBuffer = NULL;
        m_speechMeshes[pController->m_speechIndices[i]]->
                        GetVertexBuffer(&speechBuffer);
        pDevice->SetStreamSource(3 + i, speechBuffer, 0, vSize);
    }

    //Set Index buffer
    IDirect3DIndexBuffer9* ib = NULL;
    m_pBaseMesh->GetIndexBuffer(&ib);
    pDevice->SetIndices(ib);
}
```

That takes care of the first five streams, which leaves only the skinning information that you need to set from the skinned mesh before rendering the face. The following piece of code is from the soon-to-be-unveiled `Character` class. During the rendering of the skinned mesh it performs a check to see if the skinned mesh being rendered is the special case, "the head." If so, the rendering function of the skinned mesh calls this function with the `BoneMesh` containing the skinned mesh head sent as a parameter (`pFacePlaceholder`):

```
void Character::RenderFace(BoneMesh *pFacePlaceholder)
{
    if(m_pFace == NULL ||
       m_pFaceController == NULL ||
       pFacePlaceholder == NULL)
        return;

    //Set Active Vertex Declaration
    pDevice->SetVertexDeclaration(m_pFaceVertexDecl);

    //Set Stream Sources  (Stream 0 – 4)
    m_pFace->SetStreamSources(m_pFaceController);

    //Add Bone Blending Info Stream  (Stream 5)
    DWORD vSize = D3DXGetFVFVertexSize(
                        pFacePlaceholder->MeshData.pMesh->GetFVF());
    IDirect3DVertexBuffer9* headBuffer = NULL;
    pFacePlaceholder->MeshData.pMesh->GetVertexBuffer(&headBuffer);
    pDevice->SetStreamSource(5, headBuffer, 0, vSize);

    //Set Shader Variables
    ...

    //Draw Mesh
    pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0,
            pFacePlaceholder->MeshData.pMesh->GetNumVertices(), 0,
            pFacePlaceholder->MeshData.pMesh->GetNumFaces());

    //Cleanup
    ...
}
```

Okay, so basically what has happened so far is that you have located the skinned head mesh to be replaced by an actual `Face` object. You've set the vertex format for the new skinned and morphed face, and you've also set the respective vertex streams. Next is the vertex shader that reads the streams and outputs it all to the screen:

```
//Input structure to match the vertex declaration
struct VS_BONE_MORPH_INPUT
{
    float4 pos0  : POSITION0;
    float3 norm0 : NORMAL0;
    float2 tex0  : TEXCOORD0;
```

```
    float4 pos1  : POSITION01;
    float3 norm1 : NORMAL1;

    float4 pos2  : POSITION2;
    float3 norm2 : NORMAL2;

    float4 pos3  : POSITION3;
    float3 norm3 : NORMAL3;

    float4 pos4  : POSITION4;
    float3 norm4 : NORMAL4;

    float4 weights  : BLENDWEIGHT5;
    int4   boneIndices : BLENDINDICES5;
};

VS_OUTPUT vsFaceBoneMorph(VS_BONE_MORPH_INPUT IN)
{
    //First morph the mesh, then apply skinning!

    VS_OUTPUT OUT = (VS_OUTPUT)0;

    float4 position  = IN.pos0;
    float3 normal = IN.norm0;

    //Blend Position
    position += (IN.pos1 - IN.pos0) * weights.r;
    position += (IN.pos2 - IN.pos0) * weights.g;
    position += (IN.pos3 - IN.pos0) * weights.b;
    position += (IN.pos4 - IN.pos0) * weights.a;

    //Blend Normal
    normal += (IN.norm1 - IN.norm0) * weights.r;
    normal += (IN.norm2 - IN.norm0) * weights.g;
    normal += (IN.norm3 - IN.norm0) * weights.b;
    normal += (IN.norm4 - IN.norm0) * weights.a;

    //Getting the position of the vertex in the world
    float4 posWorld = float4(0.0f, 0.0f, 0.0f, 1.0f);
    float3 normWorld = float3(0.0f, 0.0f, 0.0f);
    float lastWeight = 0.0f;
    int n = NumVertInfluences-1;
    normal = normalize(normal);
```

```
    //Skin the vertex!
    for(int i = 0; i < n; ++i)
    {
        lastWeight += IN.weights[i];
        posWorld += IN.weights[i] *
                    mul(position, FinalTransforms[IN.boneIndices[i]]);

        normWorld += IN.weights[i] *
                    mul(normal, FinalTransforms[IN.boneIndices[i]]);
    }
    lastWeight = 1.0f - lastWeight;

    posWorld += lastWeight *
                mul(position, FinalTransforms[IN.boneIndices[n]]);

    normWorld += lastWeight *
                mul(normal, FinalTransforms[IN.boneIndices[n]]);

    posWorld.w = 1.0f;

    //Project the vertex to screen space
    OUT.position = mul(posWorld, matVP);

    //Lighting...
    OUT.shade = max(dot(normWorld,
                    normalize(lightPos - posWorld)), 0.2f);

    OUT.tex0 = IN.tex0;

    return OUT;
}
```

There! On the screen you'll now have a skinned and morphed face on your character. This code is all implemented in the new `Character` class. The result is shown in Figure 16.2.

As you can see in Figure 16.2, the face is no longer a static standalone face, but is now attached to the body. When the head moves the neck area stretches according to how the original face mesh was skinned.

FIGURE 16.2
Skinned and morphed face.

The Character Class

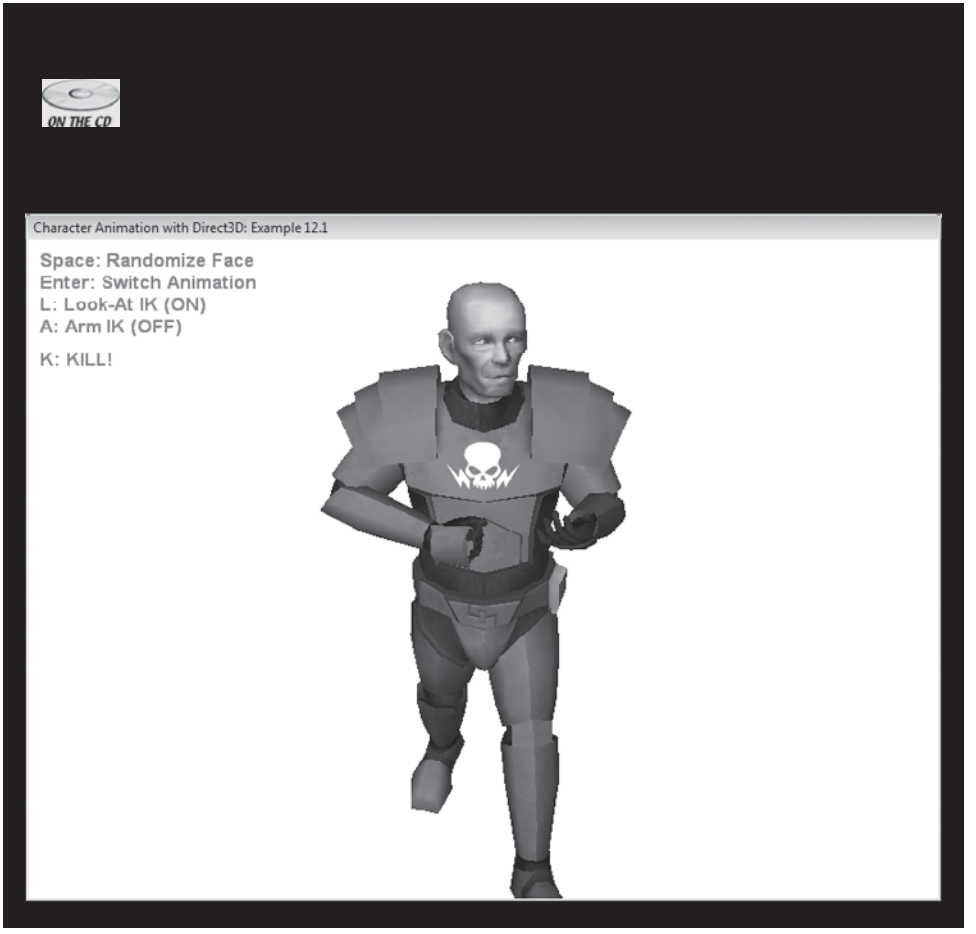
The `Character` class takes everything you've learned in this book and puts it together under one interface! The `Character` class can play keyframed animation, morphed facial animation, physical-based ragdoll animation, and inverse kinematics-based animation. The class is defined as follows:

```
class Character : public RagDoll
{
    public:
        Character(char fileName[], D3DXMATRIX &world);
        ~Character();
        void Update(float deltaTime);
        void Render();
        void RenderMesh(Bone *bone);
        void RenderFace(BoneMesh *pFacePlaceholder);
        void PlayAnimation(string name);
        void Kill();

    public:
        bool m_lookAtIK, m_armIK;
        bool m_dead;
```

```
    private:
        Face *m_pFace;
        FaceController *m_pFaceController;
        ID3DXAnimationController* m_pAnimController;
        InverseKinematics *m_pIK;
        IDirect3DVertexDeclaration9 *m_pFaceVertexDecl;
        int m_animation;
};
```

As you can see, this class inherits from the `RagDoll` class, which in turn inherits from the `SkinnedMesh` class. On top of inheriting the functionality of those two classes, it also stores a `Face` object, a `FaceController` object, an `InverseKinematics` object, and an animation controller. The putting together of this class is pretty straightforward; the only thing worth mentioning is the `m_dead` variable. As long as this variable is false, the character is "alive," meaning that you can play animations, etc. But as soon as the `Kill()` function has been called (and the `m_dead` variable has been set to true), the `RagDoll` class kicks in and the other interfaces are overruled.

I'll refrain from increasing the page count of this chapter by pasting the code for this class here (you've seen most of it in the previous chapters anyway). It would be simplest to just have a look at the code of Example 16.1 instead.



Future Work

This section is dedicated to all the things that for one reason or another I did not cover in this book (in more detail, that is). Since all the examples in this book have been aimed at getting one specific feature or point across, they are usually oversimplified and not fit for a real game application. In this section I'll address some of these issues well enough (I hope) for you to do some of your own research and implementation.

Character Level-of-Detail

Something I've left completely out of this book is Level-of-Detail (LOD). In my examples there has been, in most cases, only one character. If you were making a role-playing game (RPG) or a game containing a large number of characters on the screen at the same time, then character LOD is something you would have to address. Figure 16.3 shows the Soldier in three different levels-of-detail.

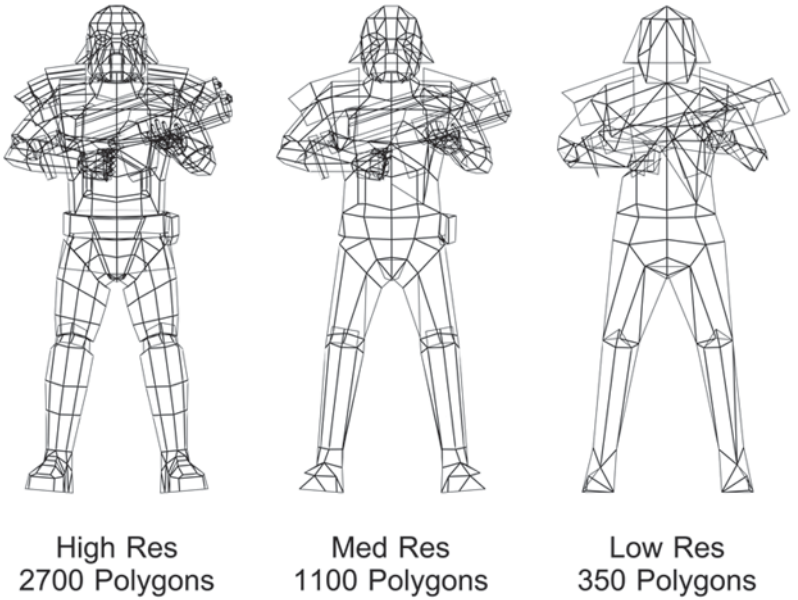


FIGURE 16.3
The Soldier in three different LODs.

The basic idea is that you render the lower-resolution model the further away from the camera the character is, as shown in Figure 16.4.

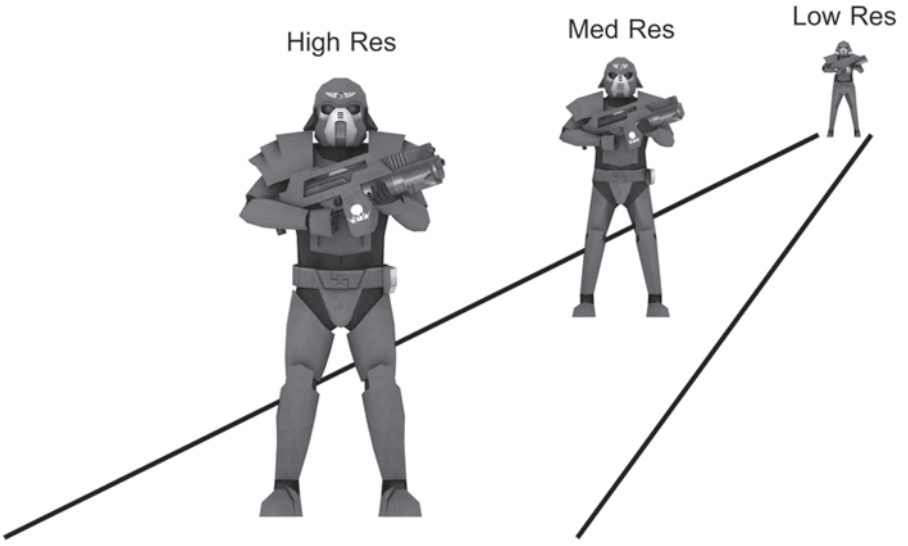


FIGURE 16.4
LOD in action.

The concept of levels-of-detail can be applied to more than just the skinned mesh. It can also be used with:

- Mesh
  Low: Low-resolution mesh, no eyes
  Medium: Medium-resolution mesh
  High: High-resolution mesh

- Animation
  Low: No animations
  Medium: No animation blending/callbacks, etc.
  High: Full animations

- Face
  Low: No morphing/low-res mesh
  Medium: Render the most dominant render target instead of the original mesh
  High: All facial animation features/morphing, etc.

- Other
  Low: No IK, physics, collisions, shadows, etc.
  Medium: Some IK, physics, etc.
  High: All features

If the character is far away, it doesn't make sense to do facial animation if the player isn't going to notice it. By just rendering the original face mesh you'll save a lot of power that otherwise would have been wasted on blending five different meshes together for the final face. This concept is pretty simple and should be easy for you to implement in your own game.

Root Motion versus Non-Root Motion

Another concept I haven't really touched on is the concept of *root motion*. With skinned meshes you had the root bone that contained the whole hierarchy of bones. Having root motion or not simply means whether or not this bone has any animation tied to it. If not, this bone stays at the origin (0, 0, 0) and doesn't move as the animation plays (see Figure 16.5).

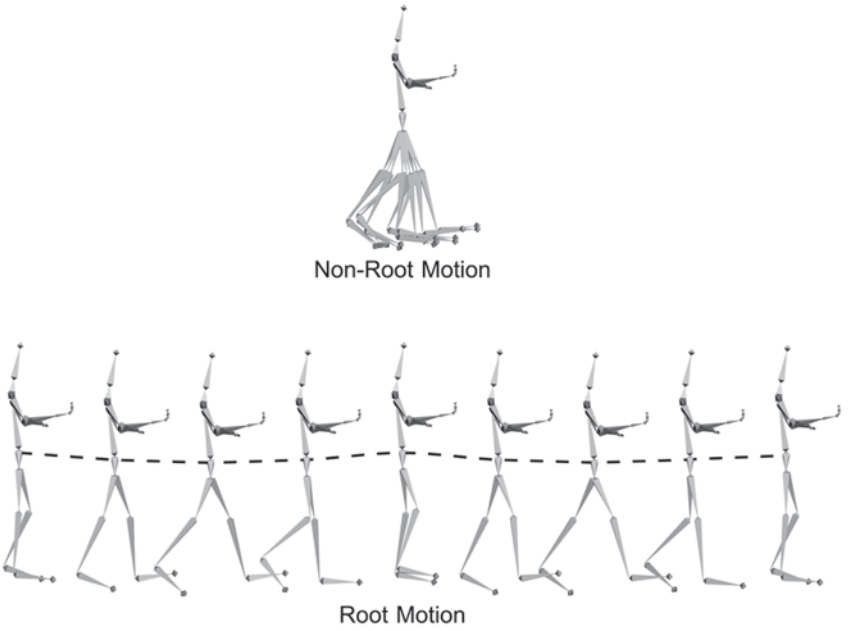


FIGURE 16.5
Non-root motion versus root motion.

An animation may or may not have root motion. When a walk cycle, for example, is captured in a motion-capture studio, it contains root motion. So when you play back the animation on the computer, the character moves away from his or her original position (just as they do in real life). In the case of non-root motion, the character stays at the origin as the animation plays (as if they were on a treadmill) and it is up to you, the programmer, to move the character forward in the game as the animation plays.

Both approaches have their pros and cons, of course. With non-root motion, the moving speed of the character is determined by the programmer. Usually, this is set to a constant speed, which may cause the character's feet to look like they are sliding whenever the actual animation speed doesn't match.

With root motion this problem is eliminated, since it is no longer up to the programmer to move the character (that data is now stored in the animation itself), but it also brings other problems to the table. One such problem is that it becomes more difficult to blend animations together since that might also blend the root motion, causing the character to end up in a different position than planned.

In the end, most games end up using both approaches. A freefall animation, for example, where the character is plummeting to his death, is a good example of an animation where root motion isn't really wanted. There the animation can just flail the characters arms while the physics engine moves the character closer to the ground (and the big splat). On the other hand, if the character is going to do a summersault or some similar move, these types of animations where the character is moving quickly benefit greatly from having root motion. Having root motion may require some extra work from you as a programmer, but in the end it produces better-looking animation (although often enough you can get away with using non-root motion). So it is basically up to you to decide how picky you want to be!

Animation Trees/Animation Graph

Today, the biggest game engines organize their animations in an animation tree or an animation graph. The animation tree or graph describes how to blend between different animations and how to transition from one animation to another. Imagine, for example, that your character is crouched and sneaking. Suddenly the player wants the character to get up and start running. You can't just blend in the running animation, since that might end up looking silly, first you have to run the "stand up" animation and then maybe even run the "walk" animation before finally blending into the "run" animation.

The animation tree takes care of this by knowing which animation can transition to which other animations. Another example is if you have a gun holstered, you can't play the "shoot" animation from the "stand" animation, you must first play the "draw gun" animation, and so on. The bottom line is that once you have large numbers of animations, you need some way of managing them. You can see an example of the Unreal Engine 3 Animation Tree Editor in Figure 16.6.

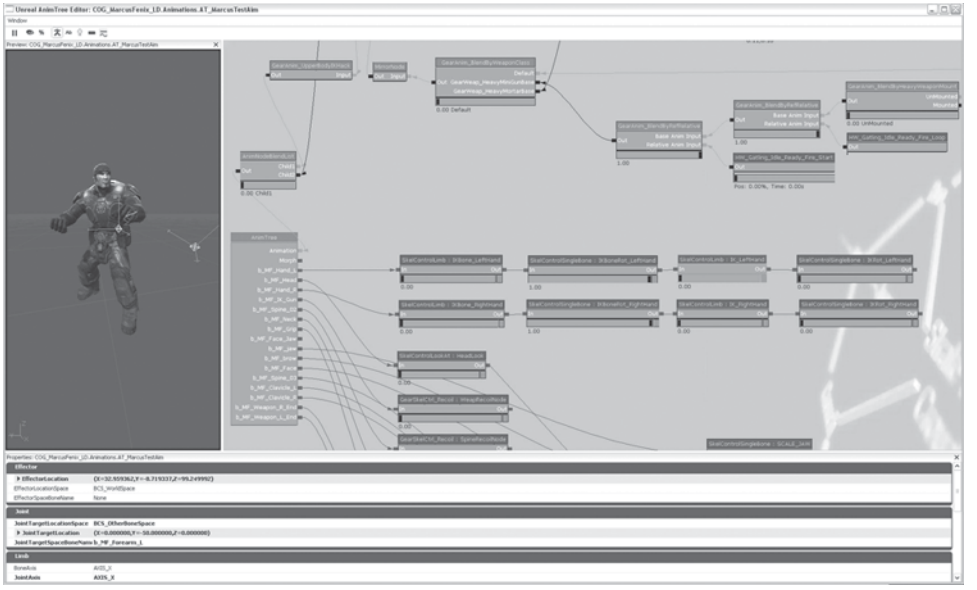


FIGURE 16.6
The Unreal Engine 3 Animation Tree Editor.

The animation tree is made up of nodes, where each node describes an animation together with some metadata describing how to play that animation (playback speed, blend weights, looping type, etc.). The animation tree can also be connected to external events such as play input, etc. You can also blend IK or physics simulations in an animation tree, and much more.

Some looping animations can have an intro animation and an outro animation as well—for example, if a character is supposed to tie his shoelace and you want this animation to take a variable amount of time each time it's done. You would have one animation called "crouch" another looping animation called "tie shoe," and finally, an outro animation called "stand up." If you had to do this in code you would soon go crazy trying to maintain special case code, and so on.

Track Masks

In Chapter 5 I discussed how to blend animations together using the ID3DX-AnimationController interface. If you want to blend an upper-body shooting animation with a lower-body run animation, this can only be done as long as the two animations don't animate common bones. If, for example, the upper body animation has keyframes holding the legs, these will still blend with the run animation's leg movement and the result will be something like a "half run."

With a track mask you can play animation and specify which bones you want the animation to affect. This way you have more control over how different animations are blended together. Unfortunately, DirectX doesn't support this feature, but it does offer you all the tools you need to implement it yourself. The ID3DXSkinInfo and ID3DXAnimationController interfaces contain all the functionality needed to implement this.

Separate Mesh and Animation Files

Throughout this book I've been using the DirectX format to store models and animation data. However, this file format isn't really meant to be used for anything other than demonstration purposes. Imagine, for example, that you have a Massive Multiplayer Online Role-Playing Game (MMORPG) with hundreds of different characters. Each and every one of them would need their own walk, run, sit, and jump animations, etc. The animation data alone would take up half your hard drive trying to run this game.

The solution is, of course, to define a common skeleton format and separate the animation data from the skinned meshes. This approach also works well if you have a long cut scene with huge amounts of animation data that only gets played once in the entire game. You can then easily load the animation data for this cut scene and then release it before continuing with the game.

The easiest (and most flexible) way to do this is probably to write your own animation importer (from whatever animation format you prefer) and do the bone mapping yourself. Be warned, however; this is not a small job.

Alan Wake Case Study

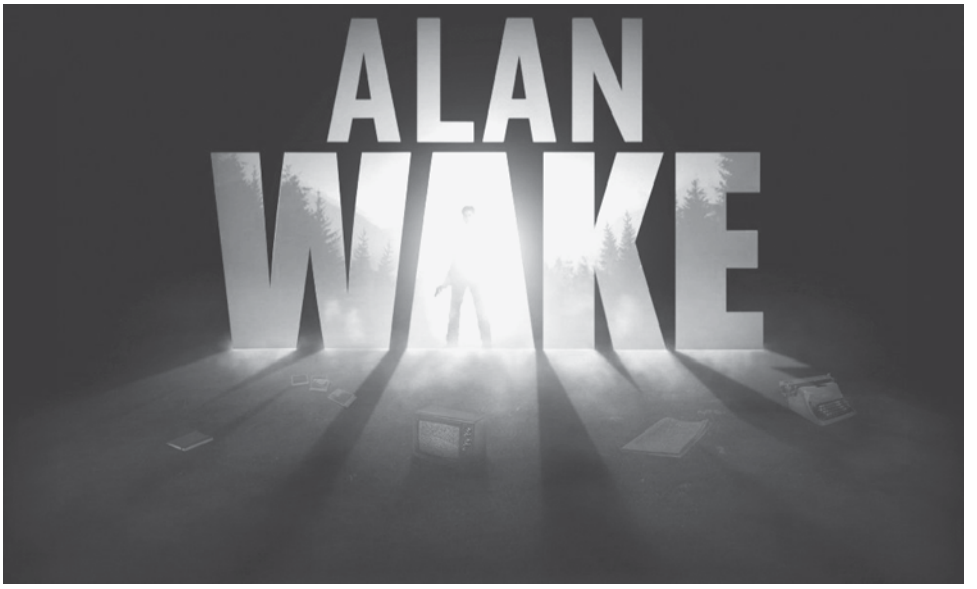


FIGURE 16.7
Copyright © 2009 Remedy Entertainment.

Before ending this book I thought it would be a good idea to let you meet a real game character. So far in this book my goal has only been to introduce you to some certain aspects of character animation using the Soldier character. This means that the quality of what you've come across in this book so far still leaves a lot to be desired if it were ever to be used in a real game. Fortunately, the good people of Remedy Entertainment (makers of *Max Payne*) have been gracious enough to let me give you a sneak peak at the main game character of their upcoming game: *Alan Wake*. I'm hoping this will give you some insight into what it takes to make a real triple-A character these days.

Alan Wake, a bestselling writer, hasn't managed to write anything in over two years. Now his wife, Alice, brings him to the idyllic small town of Bright Falls to recover his creative flow. But when she vanishes without a trace, Wake finds himself trapped in a nightmare.

Word by word, his latest work, a thriller he can't even remember writing, is coming true before his eyes.

Find out more at: ***www.AlanWake.com.***

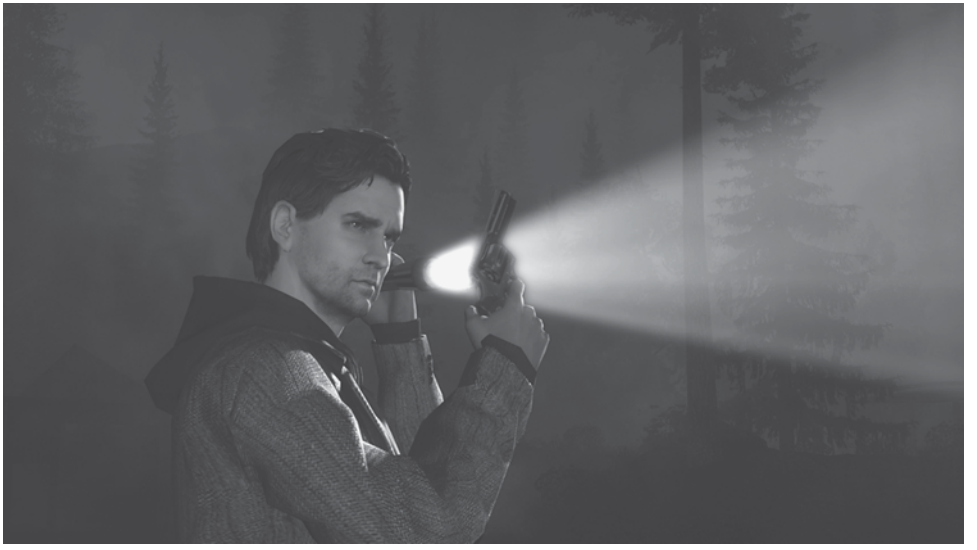


FIGURE 16.8

Interview with Sami Vanhatalo, Senior Technical Artist

Q: Would you care to make a rough guess at how many man hours were spent modeling/texturing/creating the bone setup, etc., for him?
A: Since he's our lead character, we've spent much more time on him than most other characters. I would say the character has about 8 to 10 weeks of work put on him. The number might even go up still, as he's gone through a few transformations during the development of the game.

Q: Could you tell us a bit about the different textures used for Alan Wake? How many textures (and what dimensions) are used for the final character? Diffuse maps, Normal maps, Specular maps, etc.
A: Most of the textures are either 512×512 or 1024×1024. At some point we'll probably still do an optimization pass to collapse most of the textures into a single "Atlas" texture. You've pretty much answered the question of what kind of maps we have: Diffuse map, Normal map, Occlusion map, Specular map, Wrinkle map (Normal and Diffuse). There's also some data baked into the RGBA color channel of the vertices.

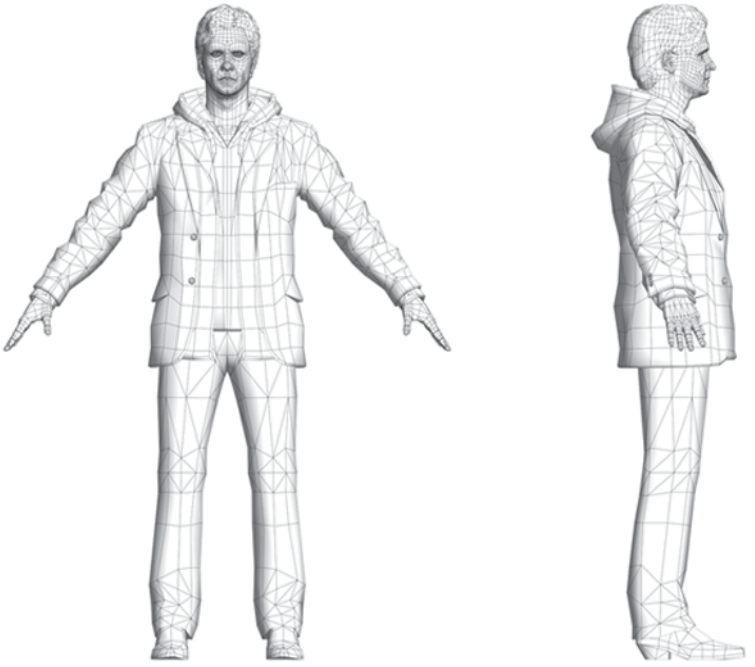


FIGURE 16.9
Copyright © 2009 Remedy Entertainment.

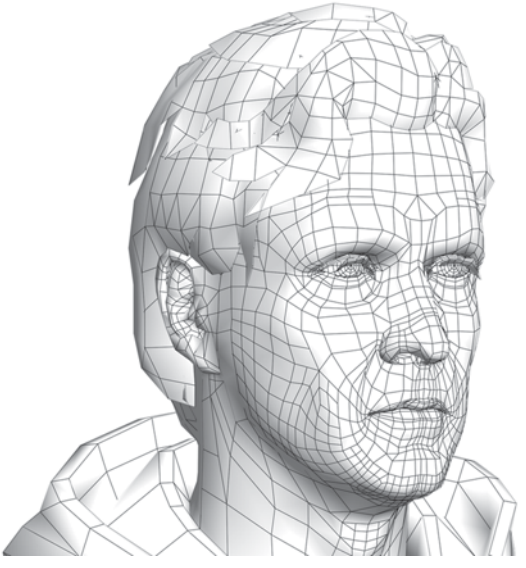


FIGURE 16.10
Copyright © 2009 Remedy Entertainment.

FIGURE 16.11

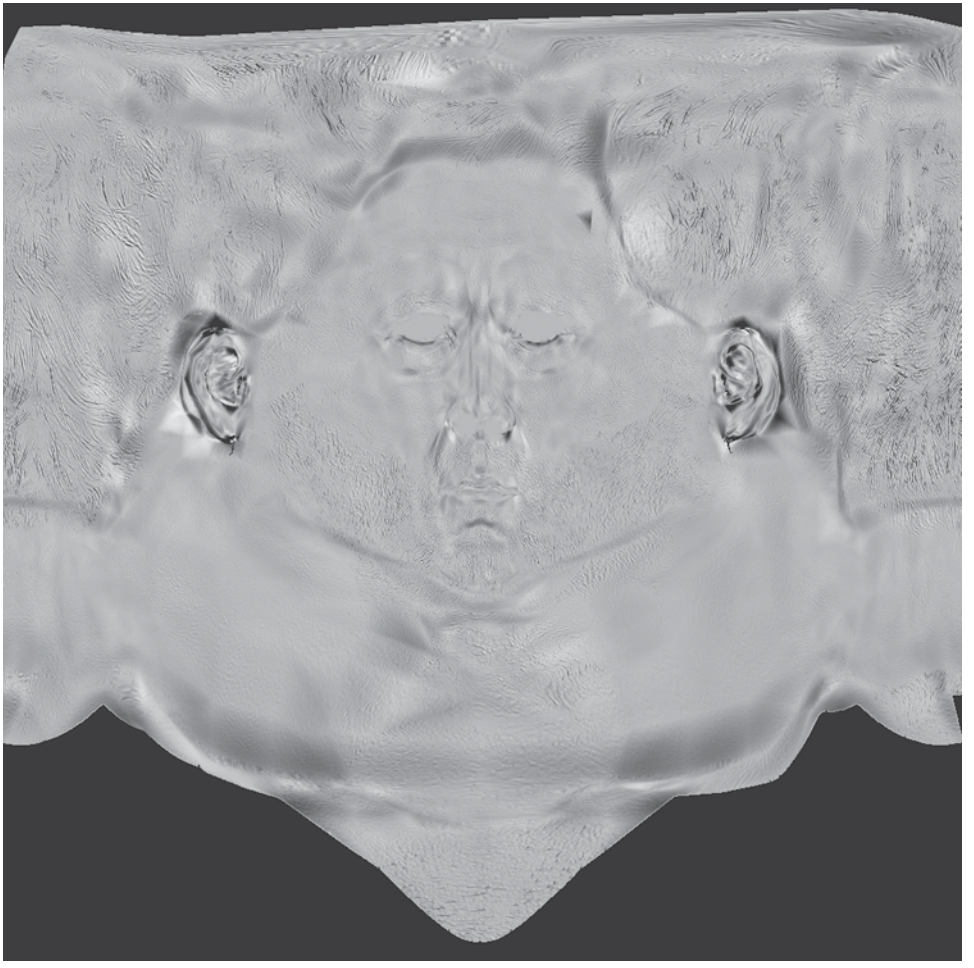


FIGURE 16.12
Copyright © 2009 Remedy Entertainment.

Q: What tools were used to create Alan? (Other tools on top of the normal ones?)
A: Not sure what exactly are the normal tools, so I'll list some… 3dsmax, Photoshop, Mudbox, Crazy Bump, and a bunch of proprietary in-house helper tools for 3dsmax.

Q: Are there different versions of the character in the game, and, if so, how do they differ?
A: There's the in-game version with LODs as well as a separate version for cut scenes. Biggest difference is the texture resolution. The cut scene version of the character also uses a bit more advanced setup for the face bones.

Q: Obviously Alan gets more attention in the game being the main character and all, but how much more complex would you say Alan is compared to other non-player characters (NPCs) in the game?
A: I would say the in-game version of Wake is about 2–3 times more complex than an average NPC. The NPCs are a bit simpler since they have to be faster to render, take less memory, and be easier to produce while still providing enough variations.

Q: What can you tell us about the facial animation of Alan Wake?
A: We currently have two different setups for Wake. A fairly standard FaceFX setup for in-game needs and a more complicated setup for facial motion capture.

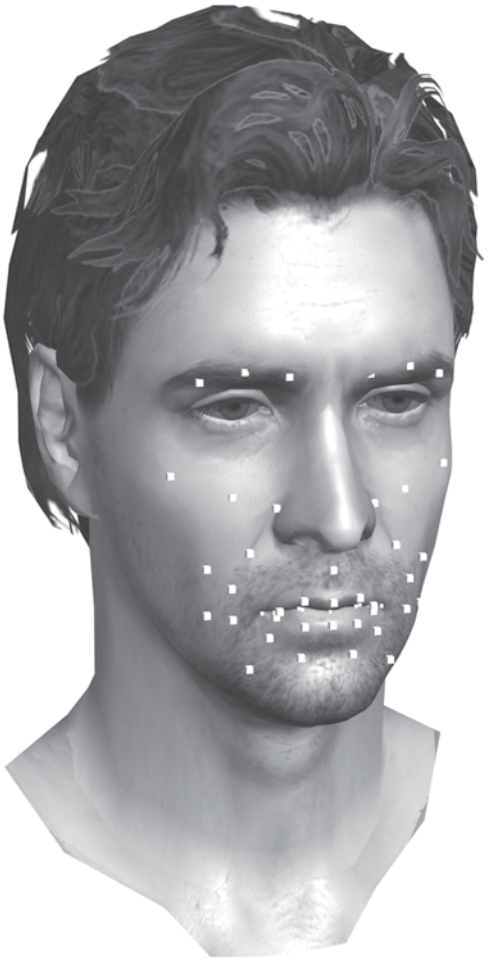


FIGURE 16.13

Q: Can you tell us a bit about the Wrinkle maps used for Alan's face?
A: We divided the face into multiple regions. Then there are separate Wrinkle maps (Normal and Diffuse) that are blended in to those regions. The blending in is driven by a setup in 3dsmax. We follow certain vertices and produce a graph in the 0 to 1 range. The animator can tweak the parameters that produce the graph and choose the vertices to follow. Once the graphs are looking "fairly good," you can still manually tweak them using 3dsmax's regular curve-editing tools. The stress map animation graphs are then exported using standard animation export tools, but instead of being bone data they are just simple float tracks with spline compression.

Q: What can you tell us about the skinning of Alan?
A: Nothing special here; basic four-bones-per-vertex skinning. There's a bunch of helper bones around armpits, knees, and elbows, some of which are code driven so things like IK don't break the animations and skinning.

Q: Is Alan the most complex character you've ever worked on/with?
A: Absolutely, and he will hopefully continue to evolve to be even more complex and lifelike in the future.

Q: What was the most difficult aspect of creating Alan?
A: Since he is based on a real actor, it was always a tough challenge to meet the artistic vision of a stressed up writer, while trying to make sure he looks as close to the original model as possible. For example, we want to maintain the similarity as we have a tradition of composing material from photographic images with CG materials, and the closer the two match, the better.

Q: Any other pearls of wisdom you want to part with to people attempting to create similar characters?
A: Get a very talented person on the job. The lead modeler behind Wake is Mikko Huovinen, who's done outstanding work on the character.

Interview with Henrik Enqvist, Animation Programmer

Q: What's the complexity of the bone setup for Alan Wake? Does each individual finger, for example, have bones?
A: We have currently about 160 bones in the Alan Wake character. The skeleton has, in theory, four parts: the body, the head, the jacket, and the hands. We use around 50 for the body, 32 for hands (16 for each hand); 24 are used to drive the jacket, and the remaining 60 (give or take a few) are used by the head.

There are a lot of bones for the fingers and the face since the animations for these need to be driven quite accurately. On the other hand, the feet only have two bones, but luckily all our characters have shoes.

In our case we went for a bone-driven face setup instead of using morph shapes. We are using motion capture for the facial animations in the cut scenes. For the in-game facial animations we use FaceFX, which has great support for bone-driven facial animations.

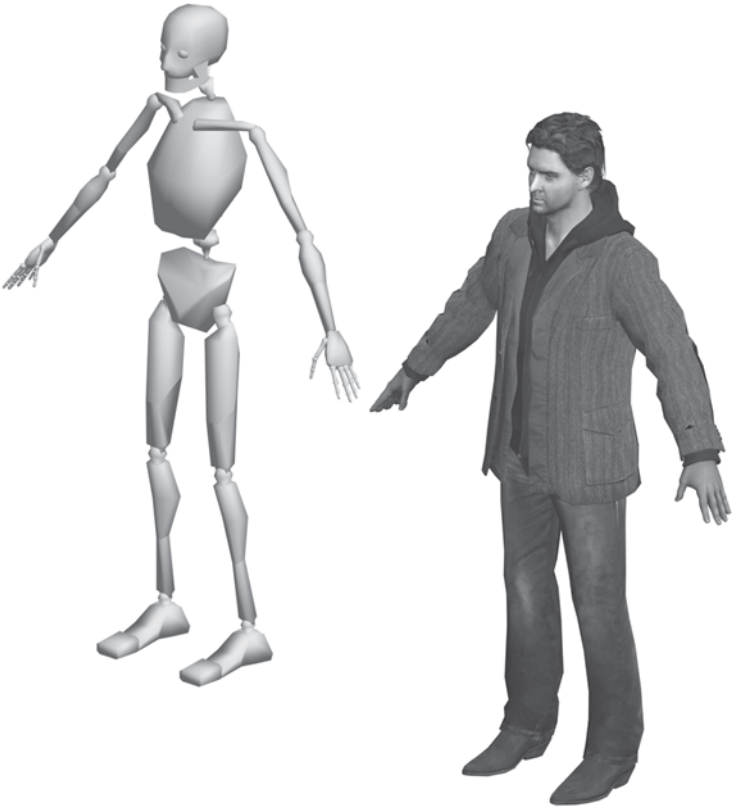


FIGURE 16.14
Copyright © 2009 Remedy Entertainment.

Q: Do you use the same bone setup for all characters in the game?
A: We basically have one male and one female skeleton. The Alan Wake skeleton is an extended version of the male skeleton. For example, the rest of the characters don't use the jacket. During combat scenes we need to be careful with CPU load so the enemies that don't need the same fidelity as Alan Wake will a have a bone setup with less bones.

Q: How many canned animations (roughly) do you have (or planned) for Alan Wake?
A: We estimate the final amountof in-game animations for the Alan Wake character to be around 200. To that we will add 200 for enemy-specific animations and another 150 for females. We will also have specific context-sensitive animations.

Then we have all the cut scenes. I don't have any number on these but I guess in the end it is about the minutes of motion capture data and not the numbers files when it comes to cut scenes.

Q: Are there any "handmade" animations used for Alan Wake or is it all motion captured animations?
A: The lion's share of the animations is motion captured, but there will be keyframed animations as well in the game. For example, with motion capture, it is easy to do something like a reload animation. Others, such as sliding down a hill or falling down, are not feasible to do with motion capture. We also have keyframed animations that are required to blend in a specific way—for example, additive animations. An example of this would be a recoil animation. The recoil animation is played on top of all movement animations, so special attention has to be paid to make sure it blends correctly in all cases.

Q: What is the biggest difficulty working with such a large number of different animations?
A: The biggest challenge is to verify that everything works together. The number of possible transitions from one animation to another is huge. For example, a run animation and a jump animation might look ok when viewed separately. But when transitioning from a running to a jumping animation the character might look weird. Before going to motion capture you basically have to define a set of base poses and then try to stick to those; otherwise you will get strange movement in body when switching animations. The shoulder area especially is something that gets a lot of jerky motions if you don't pay attention.

Q: Do you handle large "one-time only" animations differently from smaller walk loops, etc. (i.e., animations used constantly throughout the game)?
A: The basic set of movement is always available in memory; these include all the running, jumping, and shooting animations. Our cut scenes are loaded on demand; otherwise these animations would eat up hundreds of megabytes of ram. Our game engine is built from the ground up to support streaming of large amounts of data and the animations use the same system as the rest of the engine to stream assets.

Q: What parts of Alan use inverse kinematics?
A: Well, it would be easier to tell you what parts of Alan Wake do not use IK. But you asked for it, so here it comes, let's start from top down.

First, the eyes target other characters during conversations. You might think that this doesn't matter, but in cut scenes you will actually notice the difference. Of course, during hectic combat scenes the eye IK is turned off.

The rest of the IK systems are then turned on and off depending on how close the character is to the camera. This LOD system allows us to decrease the CPU load when having lots of characters in the screen. Anyway, we have a custom IK system for the head. In addition to the head bone we have two neck bones that turn gradually when the head turns. We also have separate ranges for looking vertically and horizontally.

The spine is turned so that characters can aim better with weapons. The arms and the hands are aimed toward the target during combat and there is IK that attaches the left hand to the weapon when we use rifles or axes. When a character picks up ammo or turns on a light, we use IK to steer the hand toward the correct target. We also use IK to fix the shoulder pads of the jacket when the character moves his arms.

The legs have a retarget type of IK that allows the character to modify run animations into strafe-style run loops. The feet are also raised or lowered to match the terrain. We also use foot-locking IK that prevents the feet from sliding when moving.

Q: Is there anything special to consider when you blend multiple systems together (IK, keyframed animation, cloth simulation, etc.)?
A: As I mentioned earlier, some sort of level of detail is necessary if you have many characters in the screen. The order in which you call the system is also important, but with a bit of planning this won't be any problem. The biggest problem is to get the additive layers to work correctly. You easily get some nasty looking arm twitching if you apply an additive animation to something that it is not intended for.

Q: What kind of dynamic animation systems do you have in place for Alan Wake?
A: The jacket is the most visible one. It is a Verlet-based cloth simulation with some black magic on top. The hood on top of the jacket is dynamic as well. And then we have some subtle dynamic motion for fat tissues.

Q: On the physics side, what kind of representation does Alan have?
A: We ended with a very simple representation for our ragdolls with only 11 bones. It is fascinating how far you can get with only capsules and boxes.

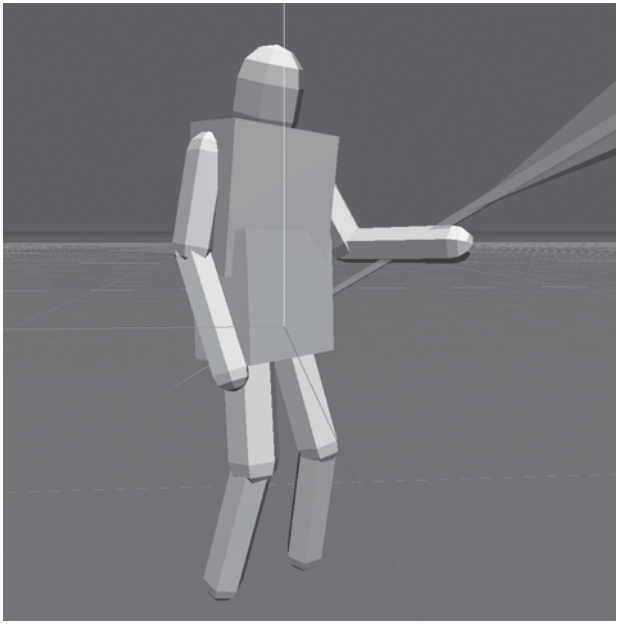


FIGURE 16.15
Copyright © 2009 Remedy Entertainment.

Q: How is the player movement hooked into the animation system? Walk us through what happens when the player presses the thumbstick on the gamepad.
A: It starts by the joypad giving a value between 0.0 and 1.0 to the animation system, telling how much the user has pressed the stick. The system then decides if the input is big enough for us to issue a run animation or to do a walk animation. If it is a walk animation we scale the speed of the animation depending on how much the player is pressing the thumbstick. If the character is standing still, we play a transition animation before we enter the actual walk or run cycle. The velocity of the character is embedded into the animation. We use this approach because we want to avoid having the feet sliding against the ground. The velocity is then fed into the physics engine, which moves the character capsule for us. There is a lot of smoothing and clamping going on in the evaluation of the joypad; real humans can't turn 180 degrees in one frame, so we need to emulate some of this behavior into the input logic.

Q: What has been the biggest challenge with making Alan Wake move?
A: The hardest thing is to find a good balance between visuals and responsiveness. When real humans move, they tend to prepare for their actions beforehand. For example, before we jump, we squat a little bit to get some momentum before doing the actual jump. In a game, on the other hand, we want the character to jump immediately when the player presses the jump button. As you can see it is impossible

to predict what the player will be doing next. If we want characters that look natural we need to introduce a little bit delay to all the actions. Tuning this delay to get a good compromise between visuals and responsiveness is a delicate task.

Assassin's Creed is game for which the development team has done a great job of getting stunning looking animations while keeping the controllers responsive.

Q: Any other pearls of wisdom you want to part with to those attempting to animate characters similar to Alan Wake?
A: There is a lot material available for doing stunning-looking graphics but not much about doing cutting-edge animations, so do your research before starting your project. Look at how other games have done it and analyze what they are doing right and what you could do better. Try to understand why they have taken a specific approach. You don't want to spend months coding a sophisticated physical system that in the end doesn't look natural.

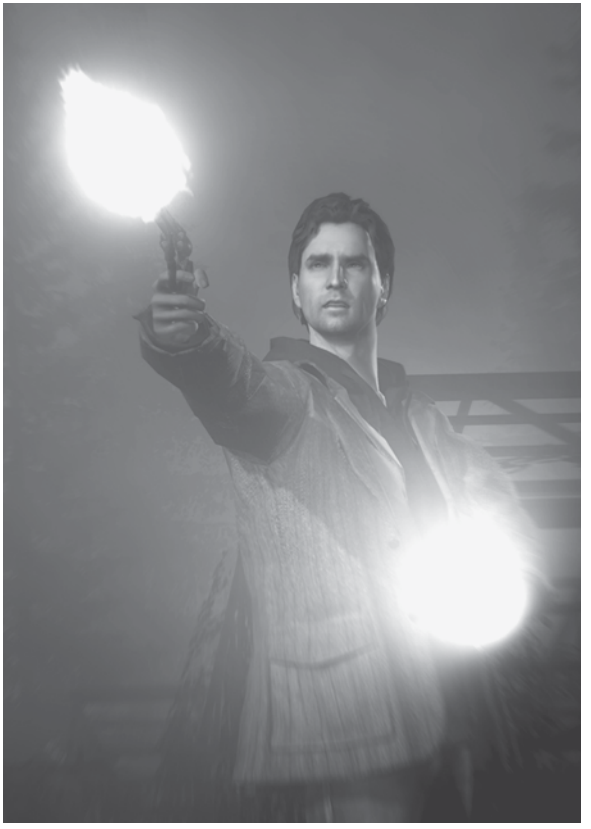


FIGURE 12.16
Copyright © 2009 Remedy Entertainment.

Final Thoughts

The aim of this book has been to offer a brief glance into the area of character animation for games. But what you've learned in this book is a long way away from some of the next-generation characters you see in games today. Already there are some systems out there with characters that respond much more realistically to physical collisions than the simple ragdoll that was implemented in Chapter 7. See, for example, "Euphoria or Endorphin by NaturalMotion" (www.naturalmotion.com). Lord knows you've only skimmed the surface of character animation after finishing this book, and there's plenty more out there to learn about the topic. Luckily, you have the Internet, where you can read more about all this.

Well, I guess this is where we must part ways. I hope you've enjoyed this book and that you will have some use of whatever you learned from it.

Further Reading

Baille-de Byl, Penny, *Programming Believable Characters for Computer Games*. Charles River Media, 2004.

Gray, Kris, *Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press, 2003.

Liverman, Matt, *The Animator's Motion Capture Guide: Organizing, Managing, Editing*. Charles River Media, 2004.

Oispa, Jason, *Stop Staring: Facial Modeling and Animation Done Right*. Sybex, 2007.

Parent, Rick, *Computer Animation: Algorithms and Techniques*. Morgan Kaufmann, 2001.

Pipho, Evan, *Focus On 3D Models*. Course Technology PTR, 2002.

Young, Vaughan, *Programming a Multiplayer FPS in DirectX*. Charles River Media, 2004.

Williams, Richard, *The Animator's Survival Kit*. Faber & Faber, 2002.

Index

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

This page intentionally left blank

This page intentionally left blank

License Agreement/Notice of Limited Warranty

By opening the sealed disc container in this book, you agree to the following terms and conditions. If, upon reading the following license agreement and notice of limited warranty, you cannot agree to the terms and conditions set forth, return the unused book with unopened disc to the place where you purchased it for a refund.

License:
The enclosed software is copyrighted by the copyright holder(s) indicated on the software disc. You are licensed to copy the software onto a single computer for use by a single user and to a backup disc. You may not reproduce, make copies, or distribute copies or rent or lease the software in whole or in part, except with written permission of the copyright holder(s). You may transfer the enclosed disc only together with this license, and only if you destroy all other copies of the software and the transferee agrees to the terms of the license. You may not decompile, reverse assemble, or reverse engineer the software.

Notice of Limited Warranty:
The enclosed disc is warranted by Course Technology to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disc combination. During the sixty-day term of the limited warranty, Course Technology will provide a replacement disc upon the return of a defective disc.

Limited Liability:
THE SOLE REMEDY FOR BREACH OF THIS LIMITED WARRANTY SHALL CONSIST ENTIRELY OF REPLACEMENT OF THE DEFECTIVE DISC. IN NO EVENT SHALL COURSE TECHNOLOGY OR THE AUTHOR BE LIABLE FOR ANY OTHER DAMAGES, INCLUDING LOSS OR CORRUPTION OF DATA, CHANGES IN THE FUNCTIONAL CHARACTERISTICS OF THE HARDWARE OR OPERATING SYSTEM, DELETERIOUS INTERACTION WITH OTHER SOFTWARE, OR ANY OTHER SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES THAT MAY ARISE, EVEN IF COURSE TECHNOLOGY AND/OR THE AUTHOR HAS PREVIOUSLY BEEN NOTIFIED THAT THE POSSIBILITY OF SUCH DAMAGES EXISTS.

Disclaimer of Warranties:
COURSE TECHNOLOGY AND THE AUTHOR SPECIFICALLY DISCLAIM ANY AND ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY, SUITABILITY TO A PARTICULAR TASK OR PURPOSE, OR FREEDOM FROM ERRORS. SOME STATES DO NOT ALLOW FOR EXCLUSION OF IMPLIED WARRANTIES OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THESE LIMITATIONS MIGHT NOT APPLY TO YOU.

Other:
This Agreement is governed by the laws of the State of Massachusetts without regard to choice of law principles. The United Convention of Contracts for the International Sale of Goods is specifically disclaimed. This Agreement constitutes the entire agreement between you and Course Technology regarding use of the software.