COURSE TECHNOLOGY
CENGAGE Learning
Professional • Technical • Reference

Advanced 2D Game Development

Jonathan S. Harbour

INCLUDES CD-ROM

Advanced 2D Game Development

Jonathan S. Harbour

Course Technology PTR
A part of Cengage Learning

COURSE TECHNOLOGY
CENGAGE Learning™

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

COURSE TECHNOLOGY
CENGAGE Learning™

Advanced 2D Game Development
Jonathan S. Harbour

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

Associate Director of Marketing: Sarah Panella

Manager of Editorial Services: Heather Talbot

Marketing Manager: Jordan Casey

Senior Acquisitions Editor: Emi Smith

Project Editor/Copy Editor: Cathleen D. Small

Technical Reviewer: Dave Calkins

PTR Editorial Services Coordinator: Erin Johnson

Interior Layout Tech: ICC Macmillan Inc.

Cover Designer: Mike Tanamachi

CD-ROM Producer: Brandon Penticuff

Indexer: Katherine Stimson

Proofreader: Tonya Cupp

Library of Congress Control Number: 2006909739

ISBN-13: 978-1-59863-342-9

ISBN-10: 1-59863-342-2

eISBN-10: 1-59863-697-9

Course Technology
25 Thomson Place
Boston, MA 02210
USA

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

For teachers Greg and Joann Dallmann and the inaugural student body at Vision Christian Academy: Jeremiah, Caleb, Ashley, Madison, Chris, Kayleigh, Stephen, Luke, Nathan, Sarah, Macy, Braden, Julie, and Bryce.

Acknowledgments

I owe a special thank you to many people who shared their technical knowledge, advice, software, and encouragement while this book was being written. Thanks to Cathleen Small for managing and editing; to Emi Smith at Cengage Learning; to Dave Calkins for technical review and help with math; to Bloodshed Software (www.bloodshed.net) for their free compiler, Dev-C++; to G-Productions (www.g-productions.net) for their DirectX DevPak, and to Jeremy Kerfs for his tutorial; to PUC-Rio for maintaining Lua (www.lua.org); to the members of the www.jharbour.com forum for testing and feedback; to Todd Spencer for allowing the use of his tracks entitled "Strange Exploration" and "Mixed Martial Arts"; to Brett Paterson of Firelight Technologies (www.fmod.org) for FMOD; to Thom Wetzel (www.lmnopc.com) for Bitmap Font Builder; to the Space Telescope Science Institute (www.stsci.edu) for the Hubble photographs; to Microsoft for their free compiler, Visual C++ 2005 Express; to Ari Feldman (www.flyingyogi.com) for SpriteLib; to Reiner Prokein (www.reinerstileset.de) for his artwork; to the developers and contributors of Anim8or, Audacity, Blender, GIMP, Notepad++, and Pro Motion—the tools that make my life easier; to Chuck and Tim at Comics Legends and Heroes (www.thelegendshop.com) for an enjoyable weekly diversion; to my lovely wife, Jennifer, and our noisy children: Jeremiah, Kayleigh, Kaitlyn, and Kourtney; to my extended family at Vision (www.visionbaptistchurch.com); to my parents, Ed and Vicki; to George Noory at Coast To Coast AM (www.coasttocoastam.com); to my students and fellow faculty at UAT (www.uat.edu), thanks for making each day an adventure.

About the Author

Jonathan S. Harbour is an Associate Professor of Game Development at the University of Advancing Technology in Tempe, Arizona. His current game project is *Starflight: The Lost Colony* (www.starflightgame.com). He lives in Arizona with his wife, Jennifer, four children (Jeremiah, Kayleigh, Kaitlyn, Kourtney), a dog (Lucy), a cat (Missy), and six temperamental computers (ages 1 to 8). He can be reached at www.jharbour.com.

This page intentionally left blank

Contents

CHAPTER 1

Building a 2D Game Engine

Game development is analogous to architecture and engineering. Just as an architect designs a construction project, such as a bridge or a skyscraper, so a game designer or *game architect* designs game construction projects, with many of the same designer workbench tools. Just as a construction engineer builds the bridge or skyscraper designed by an architect, so a *software engineer* builds the game created by the designer. Game development involves art *and* engineering.

As I'm sure you would agree, a construction engineering team can do nothing on their own without a blueprint created by an architect; they cannot even start working on a foundation without the blueprint. In like fashion, a software engineering team can do nothing on their own without a software blueprint (or *design document*).

Game development encompasses both of these fields, and game development methodologies have been formulated in the past decade to emulate the two fields of architecture and engineering. Many of the tools are the same. A game designer creates a design document (or blueprint) filled with concept artwork and highly detailed specifications for the game. A truly well-done design document could theoretically be passed on to an engineering team in order to completely build the game entirely from the design. But in practice, this is seldom the case because—like a skyscraper—a game is a monumental project that is just too large for a designer to completely encompass in the design up front. So, the architect or designer works closely with the engineering team during construction.

The example programs featured in this book are mostly derived from popular old arcade games, to aid in the learning process. When you are already familiar with the play mechanics of a game (that is, its *funativity*—a term borrowed from the game design field known as *Ludology*), then it is easier to work on the source code for a game because the gameplay is already familiar, and the rules are often simple. Examples include Atari's classics such as *Space Invaders* and *Breakout,* which are something of a cliché in this field today, but that is only because these simple games are good as educational examples. (For instance, both of these games are useful when explaining sprite collision detection.)

Compiler Support

To develop a professional game engine that is to be taken seriously—even if our goal is to build 2D games rather than 3D games—we will benefit from writing code that is not tied down to any single development tool (which I like to call being *vendor agnostic*). In other words, several different compilers will compile the engine in this book without modification. You might be surprised how "lazy" your C++ coding can become when you get too used to a single vendor's tool (such as Visual C++) and that tool's automatic features. I've run into numerous cases where my Visual C++ code generates warnings—if not errors—in another compiler (such as GCC). Writing code for multiple compilers teaches you to write robust code that is hardened against bugs. One good example is Microsoft's `sprintf_s` function, which does not exist in the standard C library—it's a custom version of `sprintf` in the Microsoft libraries. By being aware of problems like this, you can learn to avoid them altogether. (In this case, I would prefer to use `std::ostringstream` to format a string instead.)

My favorite compiler is not Visual C++. I have nothing against Microsoft—they developed DirectX, after all, which is what we're basing most of our engine on. The problem with Visual C++ is that it is in the midst of a family feud of sorts. There are now several generations of the compiler that disagree with each other—the versions from 1998, 2003, 2005, and 2008, to be specific. While the old Visual C++ 6.0 dating back to 1998 might be considered grossly out of date today, I certainly would expect Visual C++ 7.1 from 2003 to still be in favor. But the reality of the situation is that none of these versions is compatible with each other. If you create a project in one version, it will most likely not work in another. That makes it very difficult to support Visual C++ because most aspiring C++ programmers (namely, beginners) usually assume that the product name is the most important measure of compatibility—if it's called Visual C++, then it will compile Visual C++ code, right? Hmm. One would assume as much, but the situation is complicated by the fact that each new version breaks compatibility with prior versions. Software is complicated enough without throwing these strange problems into the mix. When professional developers struggle with compatibility problems, one can only wonder how beginners fare!

So, what can we do about this problem? The most important thing is that we write code that will compile on all of these compilers. For all intents and purposes, each version of Visual C++ must be treated like a different compiler. It helps to even consider them as the products of different vendors, since Microsoft changes direction with the wind—and there's nothing more frustrating to a C++ programmer than finding his or her two- to three-year old engine or library no longer works with the latest compiler (which is the case today).

Here is a list of compilers that are still relevant today, which should all be able to compile the engine and examples from this book. Specifically, I am focusing on Dev-C++ 5.0 and Visual C++ 2005 SP1 and providing projects for these two compilers on the book's CD. Due to the way the projects are configured, you will find it very easy to add additional compiler support to the existing configurations—as long as a compatible DirectX 9 library is available for your compiler. Note that in the case of the Microsoft compilers, the Professional, Enterprise, and Express (free) editions all function the same.

- Dev-C++ 5.0 (MinGW / CygWin)
- Borland C++ 6.0
- Visual C++ 7.1 (2003)
- Visual C++ 8.0 (2005)
- **Visual C++ 8.0 (2005 SP1)**
- Visual C++ 9.0 (2008)

Advice

Projects for the compilers shown in bold text in the compiler list are available on the CD.

If you are completely new to game development, you will find this book to be a serious challenge because we don't cover the basics here—this is an *Advanced* title. However, if you aren't sure where to begin, I recommend you use Dev-C++ 5.0 (technically, the version is 4.9.9.2 beta). Why? First of all, Dev-C++ is based on the world-class GCC compiler, which is used in all of the professional console development kits (for systems like the Wii and PS3). Secondly, Dev-C++ is *small*. The installer is tiny, and the full installation is only about 120 MB. (Contrast that with Visual C++, which weighs in at five times that size.) Third, Dev-C++ has fewer dependencies, and its binary executable code (built with GCC) does not embed a manifest file like Visual C++ does.

Advice

A *manifest file* describes the runtime libraries required to run an executable program built with Visual Studio (any language). The manifest was supposed to eliminate the DLL dependency problems that developers had to deal with in past versions of Microsoft's development tools. However, "DLL nightmare" has been replaced with "Manifest nightmare," to the extent that many Visual C++ programs will not even run on the same PC they are compiled on.

We want to be able to write advanced 2D games with the least amount of difficulty, which is why I'm making so many strong suggestions this early on. If you use a complex compiler, plan to deal with complex challenges inherent in using such software. But if you don't need feature overload, going with a simpler compiler (such as Dev-C++) will make game development equally simple and painless.

Advice

As of late 2007, Firaxis Games was still using Visual C++ 2003. You can tell by downloading the latest *Civilization IV* SDK (for the *Beyond the Sword* expansion). This is a fairly common situation in game engine "mod" development kits.

DirectX SDK Support

Microsoft's official DirectX SDK can be downloaded from http://msdn.microsoft .com/directx/sdk. The current version at the time of this writing is 9.21.1148, dated November 2007. However, we are not using Direct3D 10—this book does not venture beyond Direct3D 9. If you are using Dev-C++, you do not need Microsoft's DirectX SDK, only the runtime.

Advice

Direct3D is the only DirectX component that has been updated to version 10. None of the other components (DirectSound, DirectInput, and so on) has changed much (if at all) since around 2004. All this means is that DirectInput does what it needs to do just fine and needs no new updates, just as DirectSound supports high-definition audio systems and 3D positional sound without needing to be updated further. However, Direct3D is updated regularly to keep up with the latest graphics hardware.

I recommend you use an older version of DirectX, even if you're using Visual C++. Although the November 2007 and future releases *may work*, there is no guarantee, as Microsoft is not dedicated to preserving backwards compatibility. For instance, the October 2006 release is a good one that I use most often (and this is the version provided on the CD). Just remember this advice when it comes to game development—the latest and greatest tools are not always *preferable* for every game project.

Advice

We do not study the basics of DirectX in this advanced book. If you have never written a line of DirectX code in your life, then you will need a crash course first. I recommend *Beginning Game Programming, 2nd Edition* (Course Technology, 2006), which will teach you all of the basics at a very slow pace. The first four chapters cover Windows programming before even getting into DirectX, and only ambient lighting is covered to keep the examples simple for beginners.

Why Do We Need an Engine?

What is the purpose or advantage of a game engine, as opposed to, say, just writing all the code for a game as needed? Why invest all the time in creating a game engine when you could spend that time just writing the game?

These are valid questions that I have pondered over the years while developing small and large game projects (especially those for college courses). The simple answer is: You don't need an engine to write a game. But that is a loaded answer because it implies that either 1) The game is very simple, or 2) You already have a lot of code from past projects. The first implication is that you can just write a simple game with DirectX or OpenGL code. The second assumes that you have some code already available, perhaps in a game library—filled with functions you've written and reused. A game library saves a lot of time. For instance, it's a given that you will load bitmap files for use in 2D artwork or 3D textures, and once you've written such a function, you do not want to have to touch it again, because it serves a good purpose. Anytime you have to open up a function and modify it, that's a good sign that it was poorly written in the first place. (Then again, it's possible you have gained new knowledge and want to improve your functions, which is valid.)

In my opinion, there are three key reasons why a game engine will help a game development project: teamwork, cross-compiler support, and logistics. Let's examine each issue.

1. Teamwork is much easier when the programmers in a team use a game engine rather than writing their own core game code, because the engine code facilitates standardization across the project. While each programmer has his or her own preferences about how timing should be handled, or how rendering should be done, a game engine with a single high-speed game loop forces everyone on the team to work with the features of the engine. And what of features that are lacking? Usually one or two team members will be the "engine gurus" who maintain the engine based on the team's needs.

2. Cross-compiler support is almost impossible without the use of a game engine. Although many programmers are adept at writing standard C++ code that will build on multiple platforms and compilers, game code usually does not fall into that realm due to its unique requirements (namely, rendering). Cross-compiler support is the ability to compile your game with two or more compilers, rather than just your favorite (such as Visual C++).

Advice

Writing code that builds on compilers from more than one vendor teaches you to write good, *standard* code, without any ties to a specific platform. It's *hard* to write platform-independent code! Be prepared for a serious workout!

3. Logistics in a large game project can be a nightmare without some coordinated way to organize the entities, processes, and behaviors in your game. Logistics is the problem of organizing and supporting a large system, and is often used to describe military operations (for example, the logistics of war—equipping, supplying, and supporting troops). The logistics of a game involve the characters, vehicles, crafts, enemies, projectiles, and scenery—in other words, the "stuff" in a game. Without a system in place to assist with organizing all of these things, the game's source code can become an unmanageable mess.

Let's summarize all of these points in a simple sentence: A game engine makes it easy—sometimes ridiculously easy—to make a game. Contrast that with the problems associated with creating a game from scratch using your favorite APIs, such as Direct3D or OpenGL for graphics, DirectInput for mouse and keyboard support, Winsock for networking, FMOD for audio, and so forth. The logistics of keeping up with the latest updates to all of these libraries alone can be a nightmare for a game developer. But by wrapping all of these libraries and all of your own custom game code into a game engine, you eliminate the headache of maintaining all of those libraries (including their initialization and shutdown) in each game. The best analogy I can come up with is this: "Rolling your own" game code for each game project is like fabricating your own bricks, forging your own nails, and cutting down your own trees in order to build a single house. Why would you do that?

Indeed!

But perhaps the most significant benefit to wrapping an API (such as DirectX) into your own game engine classes is to provide a buffer around that API's unpredictable future revisions. Whenever a change occurs in a library that you regularly use in your games, you can accommodate those changes in your engine classes without having to revise any actual *game code* in the process. Based on my comments already about the problems with compatibility in software today, this is an especially important point to take to heart.

Creating the Engine Project

We are going to create the core game engine project in this chapter and then expand it over the next half-dozen chapters to include all of the features we need to build advanced 2D games. The starting point is the core engine developed in this chapter, which will include WinMain, Direct3D initialization, D3DXSprite initialization, the basic starting game event functions (`game_init()`, and so on), timing and automatic frame-rate maintenance, and of course, a game loop. The great thing about doing all of this right now, at the very beginning, is that we will not have to duplicate any of this code in future chapters—it will already be embedded in the game engine.

Let's get started creating the engine project so that we'll have a foundation with which to discuss the future design of our engine. The `Engine` class is embedded in a namespace called Advanced2D. This namespace will contain all of the engine classes so there will not be any conflicts with other libraries you may need to use in a game. I will go over the project creation for two compilers in detail now and show you which libraries you will need to include in the project, so that you may refer to this chapter again for future projects. We will continue to build on the Engine project (which you are about to create) in future chapters.

Dev-C++ Project

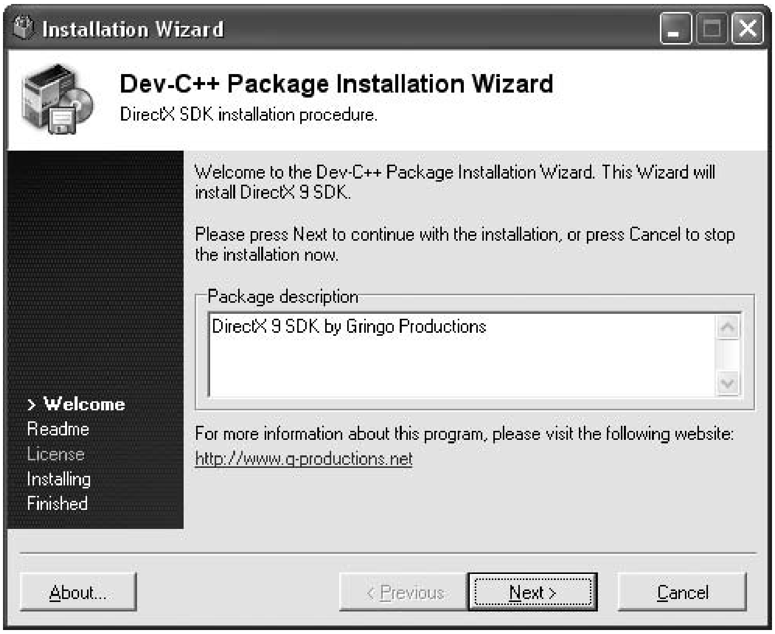
(Note: If you are using Visual C++, you may skip this section.) Dev-C++ 5.0 is a modern C++ compiler based on the GCC MinGW kit for Windows, and it is available for install from the book's CD. The actual revision at the time of this writing is 4.9.9.2. You may check www.bloodshed.net for updates. By default, Dev-C++ includes Win32 compatibility, meaning you can compile Windows code with it using GCC versions of the Windows API. The API is really just a collection of library files (such as gdi32 and winmm). The library file extension for GCC libraries is .a, while the library file extension for MSVC libraries is .lib, and they are not compatible. Since Dev-C++ includes the Win32 API, that really simplifies configuration.

However, DirectX is *not* included, so we must install it. The DirectX SDK distributed by Microsoft does *not* work with Dev-C++ (because of the library file format), but there is a third-party version of the DirectX SDK available for Dev-C++ (and GCC compilers in general), which is available on the book's CD.

First, install Dev-C++ if you have not done so already. The installer is available on the CD-ROM, or you may download it from www.bloodshed.net/dev/devcpp.html. After you have installed Dev-C++, then you can install the DirectX DevPak, either from the CD or from this URL: www.g-productions.net/list.php?c=files_devpak. (Be sure to download the 9.0c version.) The DirectX installer is shown in Figure 1.1. After installation is complete, the DirectX library will show up in the Package Manager, as shown in Figure 1.2.

With the DirectX library now available, we can create a new project. Open Dev-C++ and click File, New, Project. This will bring up the New Project dialog shown in Figure 1.3. Name this new project "Engine" (or whatever you choose).

Make sure you choose Windows Application and C++ Project (from the options on the lower right). Although a DirectX template is available (as you can see on the New Project dialog tabs list), we will not be using the template because it includes too much code, and we want to start off with a simple, empty project. More than likely, the provided DirectX template won't meet your needs anyway


Installation Wizard

Dev-C++ Package Installation Wizard

DirectX SDK installation procedure.

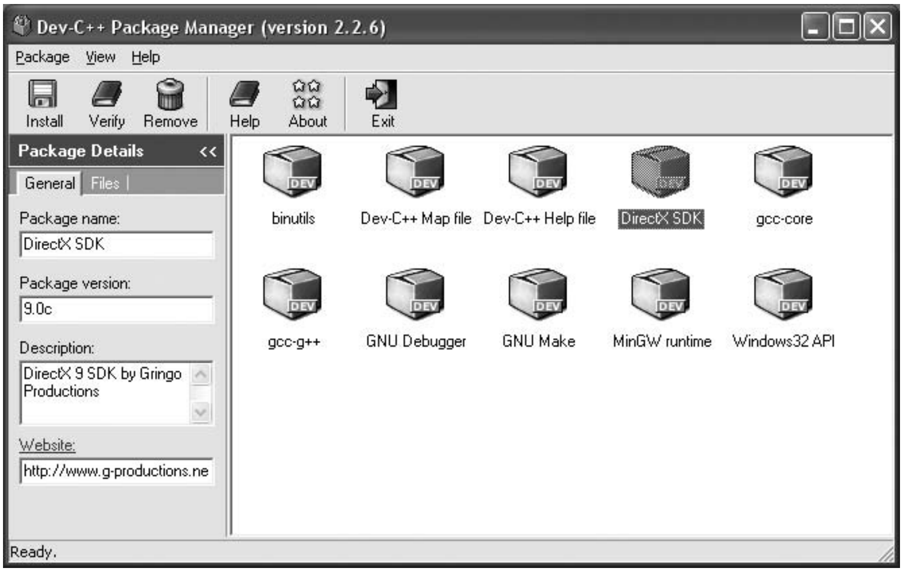
Welcome to the Dev-C++ Package Installation Wizard. This Wizard will install DirectX 9 SDK.

Please press Next to continue with the installation, or press Cancel to stop the installation now.

Package description: DirectX 9 SDK by Gringo Productions

For more information about this program, please visit the following website: http://www.g-productions.net

Figure 1.1
Installing the DirectX 9.0c DevPak.


Dev-C++ Package Manager (version 2.2.6)

Package name: DirectX SDK

Package version: 9.0c

Description: DirectX 9 SDK by Gringo Productions

Website: http://www.g-productions.ne

Ready.

Figure 1.2
The Package Manager shows the packages that have been installed.

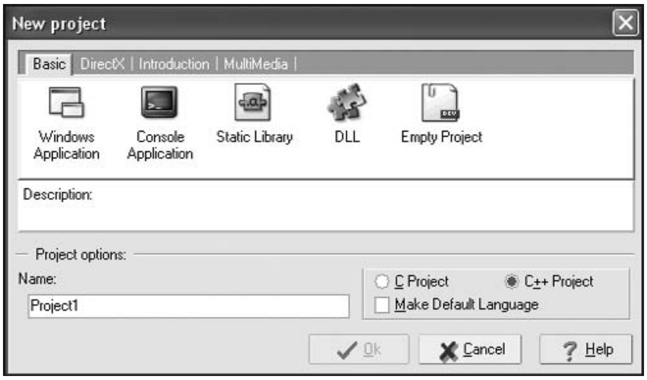


Figure 1.3
Creating a new project in Dev-C++.

because it is a bit out of date, and we will be writing *all* of the DirectX code from scratch on our own!

Dev-C++ does not automatically save the project file when you create the project—it simply starts a new project in memory, and you must save it. Now is a good time to talk about folders, because we will be organizing the engine project into sub-folders for multiple-compiler support.

Creating the New Engine Project

Create a new folder where you would like to store your game engine. I've called my main folder simply "Engine." Now create a new folder inside Engine called devcpp. The folder structure will be Engine\devcpp. This is where you must save the Dev-C++ engine project. I've called my project Engine, and this will be the name of the class, but I saved it to a file called Advanced2D.dev. I encourage you to do the same. Now, if you haven't done so already, save the project. You should find the project file located here if you have saved it correctly: \Engine\devcpp\Advanced2D.dev. If your Dev-C++ project came with a main.cpp file, you may remove it without saving, as we'll be creating our own files.

Now you're going to create six new files in the Engine project. You can create a new file using the File, New, Source File menu option. When you create a new file in Dev-C++, it lets you begin writing code and save it later. You must save these files in the *main engine folder*. I called my folder Engine, so to keep this tutorial simple I recommend you do the same. The source files should be saved in \Engine, not in \Engine\devcpp. Why? It's a matter of logistics, which we discussed earlier. The source code files will be *shared* by all of the compilers, and each compiler will have its own dedicated folder, where it will output all object files and other intermediate files generated during compilation. When prompted to add each of these files to the project, select Yes. This keeps the engine project clean. Just save each of these files to \Engine as you create them:

- Advanced2D.h
- Timer.h
- winmain.h
- Advanced2D.cpp
- Timer.cpp
- winmain.cpp

We'll go over the source code for these files in the upcoming section titled "Engine Source Code." I've created two filters in the project file list: Source Files and Header Files. You may create similar filters for your project and *drag* your source files to the filtered items if you wish, as I have done. The compiler will not create actual *folders*; it will only organize your files. We'll leave the new files empty for a while. The project should look something like Figure 1.4 at this point.

Advice

You can rename a project at any time! Open the Project menu, select Project Options, and you will see a text field where the project name may be edited.

Configuring the New Project

Although the source code files are still empty, we'll just go ahead and configure the project now, while we're on the subject of Dev-C++. Open the Project menu and select Project Options to bring up a dialog of the same name. In the General tab, which comes up first, change the project type to Win32 Static Lib, as shown in Figure 1.5.

Next, click the Build Options tab. Change the field labeled Executable Output Directory to ..\lib. Change the next field, labeled Object File Output Directory, to .\obj, as shown in Figure 1.6. Finally, enable Override Output Filename and enter libAdvanced2D.a as the library filename. These options will cause the compiler to

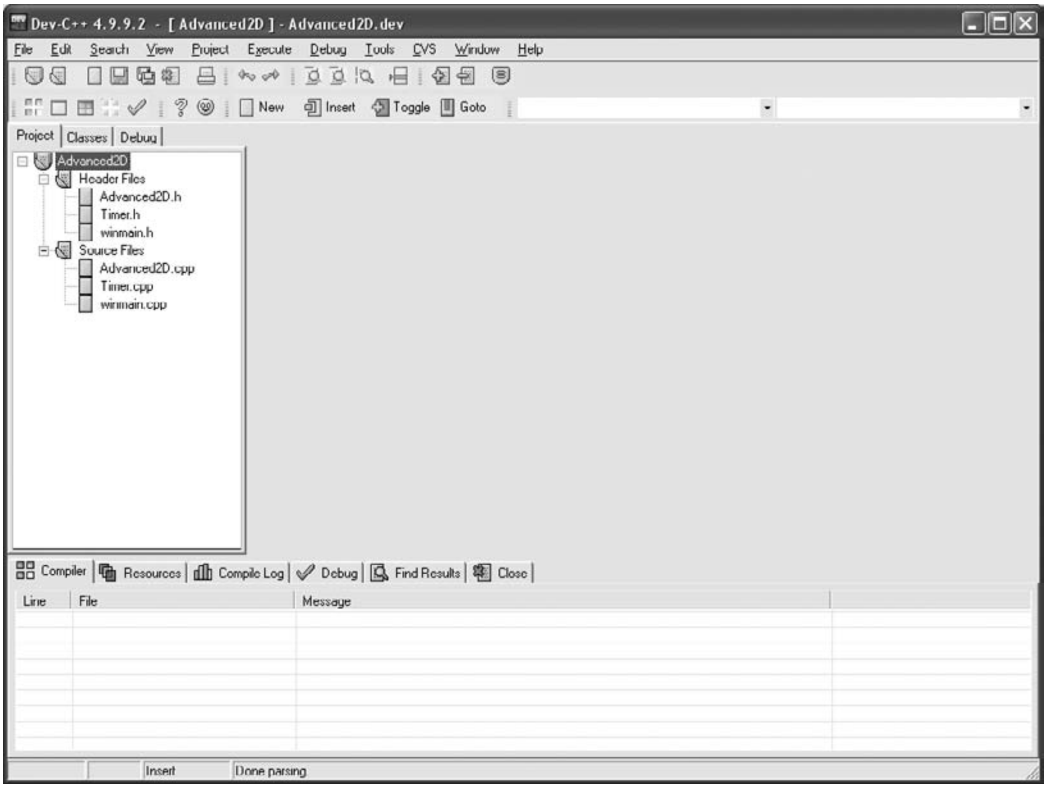
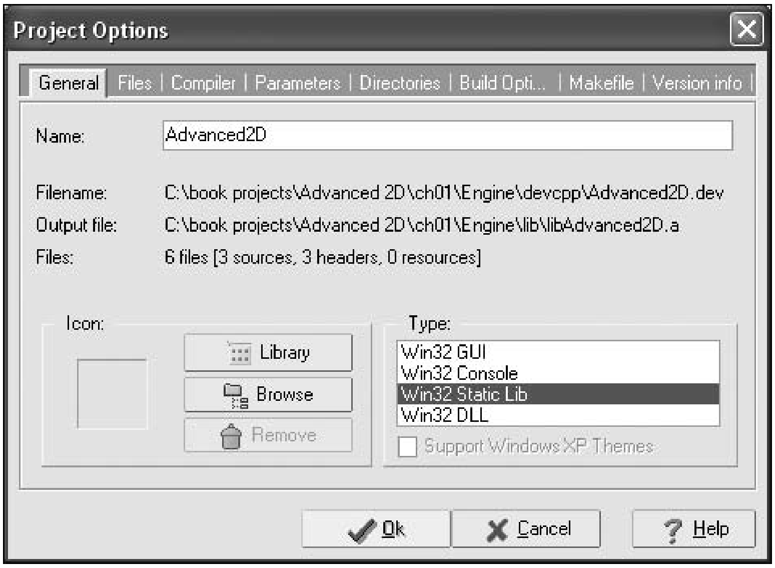
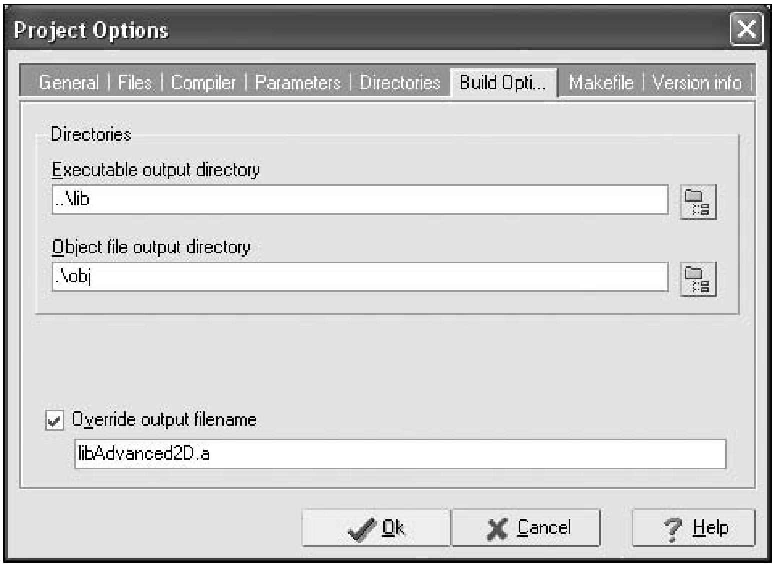


Figure 1.4
The Dev-C++ engine project has files but needs source code!



Name: Advanced2D
Filename: C:\book projects\Advanced 2D\ch01\Engine\devcpp\Advanced2D.dev
Output file: C:\book projects\Advanced 2D\ch01\Engine\lib\libAdvanced2D.a
Files: 6 files [3 sources, 3 headers, 0 resources]
Type: Win32 Static Lib
Support Windows XP Themes [ ]

Figure 1.5
Setting the project type to a static library.


Project Options
General | Files | Compiler | Parameters | Directories | Build Opti... | Makefile | Version info |
Ok
Cancel
Help

Directories

Executable output directory: ..\lib

Object file output directory: .\obj

☑ Override output filename: libAdvanced2D.a

Figure 1.6
Configuring the project output.

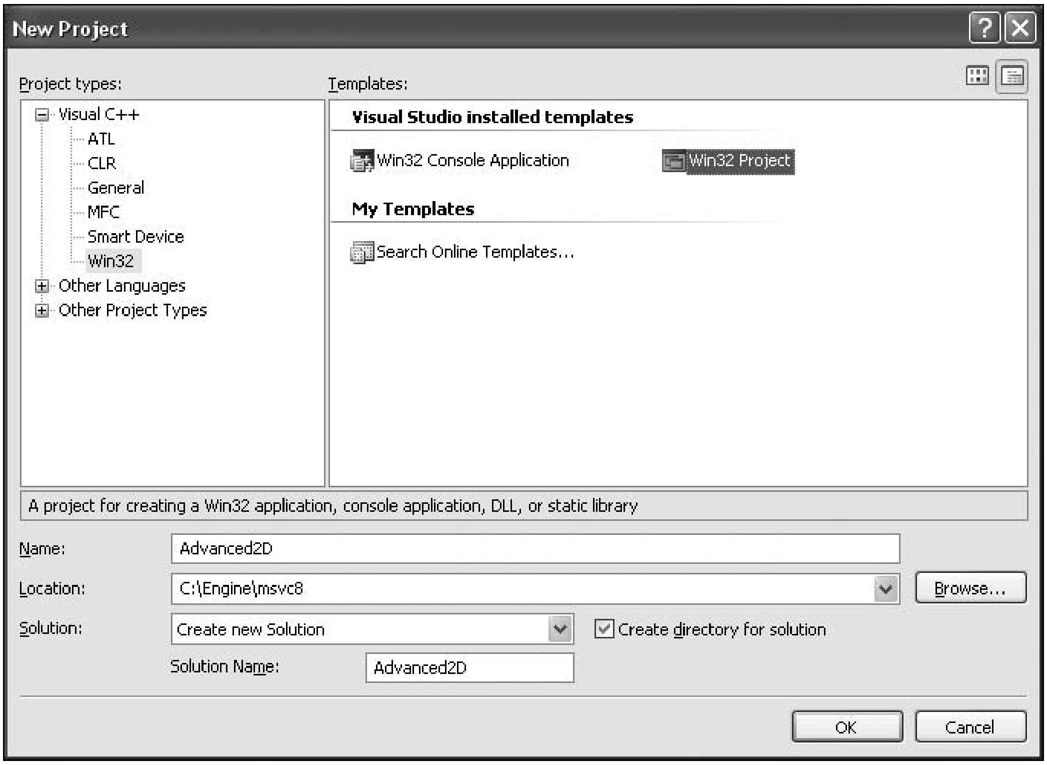
output all object files to a folder called \Engine\devcpp\obj and output the resulting library file to \Engine\lib. Note the library output folder is located under \Engine, rather than \Engine\devcpp. We want the library file (libAdvanced2D.a) to be created in \Engine\lib so it is easy to find.

Visual C++ Project

(Note: If you are using Dev-C++, you may skip this section.) Now we'll create the Visual C++ project for the game engine. I am using Visual C++ 2005 SP1, so that is the format of the projects on the CD. If you are using Visual C++ 2008, the projects will open after they are automatically upgraded to the new project format. Open the File menu and select New, Project to open the New Project dialog shown in Figure 1.7. If you are using an Express edition, you will see fewer items in the list of project templates.

Creating the New Engine Project

This dialog can be pretty convoluted if you aren't familiar with it. I recommend minimizing everything in the tree besides Visual C++. Locate the Win32 section and choose Win32 Project. The project name should be either Advanced2D or



A project for creating a Win32 application, console application, DLL, or static library

Name: Advanced2D
Location: C:\Engine\msvc8
Solution: Create new Solution
[✓] Create directory for solution
Solution Name: Advanced2D

Figure 1.7
Creating a new Win32 project in Visual C++.

Engine. Let's first create a new folder to contain the engine project. I have called the folder \Engine, but you may use any name you wish (as long as you note the difference when referring to figures in this tutorial). The Visual C++ 2005 (that is, MSVC8) project will be stored in a folder called \Engine\msvc8. (If you skipped the previous section, note that we created the Dev-C++ project in \Engine\devcpp.)

The *source code files* will be stored in \Engine, but the *solution and project files* will be stored in \Engine\msvc8. Why? The main reason is to keep the project clean. If you store your project file in the same folder with your sources, then the sources will be cluttered with all of the output files generated by the compiler, not to mention the output folders (Debug or Release). Then there are the program database files (.pdb), object files (.obj), IntelliSense file (.ncb), and so forth. Let's try to keep the project organized, and it will have a more professional feel to it.

Advice

If you ever close Visual C++ and it does not respond for 10 to 15 minutes, that is a known bug with the IntelliSense update process that ignores the user's desire to shut down and continues plodding away. If you want to avoid this bug, one way is to make the project's .ncb file read only.

Advice

When the project wizard appears, you will want to choose Static Library for the project type and uncheck the Precompiled Header option. In addition, be sure to *uncheck* the Create Directory for Solution option. If you forget to disable this option, Visual C++ will create an additional directory inside .\Engine\msvc8, which will not work the way we want.

Now you're going to create six new files in the Engine project. You can create a new file using the Project, Add New menu option, and then choose the type of file you want to have added to the project. When you create a new file in Visual C++, it lets you begin writing code and save it later. (The new file will be called something like Source1.cpp by default.) You must save the source files in the *main engine folder*, \Engine. The source files should be saved in \Engine, not in \Engine\msvc8. Why? It's a matter of logistics, which we discussed earlier. The source code files will be *shared* by all of the compilers, and each compiler will have its own dedicated folder, where it will output all object files and other intermediate files generated during compilation. This keeps the engine project clean. Just save each of these files to \Engine as you create them:

- Advanced2D.h
- Timer.h
- winmain.h
- Advanced2D.cpp
- Timer.cpp
- winmain.cpp

We'll go over the source code for these files in the upcoming section titled "Engine Source Code." We'll leave the new files empty for a while. The project should look something like Figure 1.8 at this point. By default, Visual C++ will also create a new Resources filter in your project file listing, which you may remove or leave as is. (It's irrelevant.)

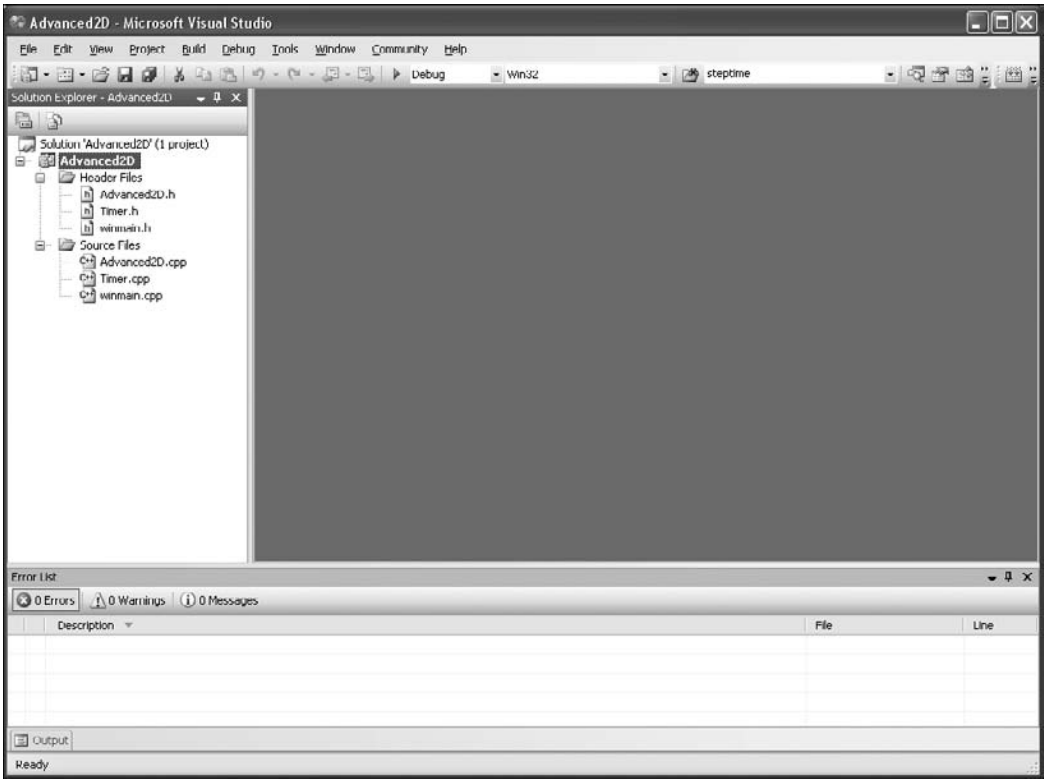


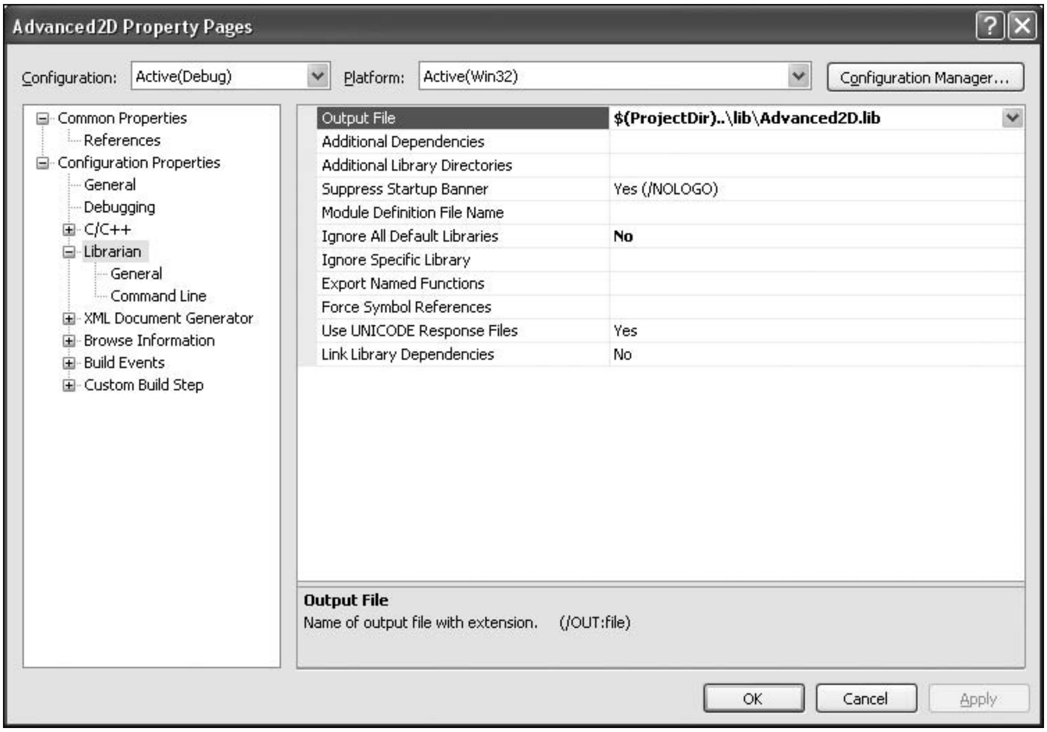
Figure 1.8
The Visual C++ engine project has files but needs source code!

Configuring the New Project

Although the source code files are still empty, we'll just go ahead and configure the project now because the section with the source code (coming up shortly) is applicable to all compilers. Remember: Single source code set, multiple compilers.

Select the project name in the project manager, then right-click and choose Project Properties. (Or you may open the Project menu in Visual C++ and select Properties.) Visual C++ automatically configures new projects to output intermediate files to a folder called either Debug or Release, depending on the configuration currently in use, so we don't need to specify the object folder as we did for Dev-C++. But we *do* need to tell Visual C++ where to output the library file.

Open the Configuration Properties item in the tree view and change the Output File field to $(ProjectDir)..\lib\$(ProjectName).lib, as shown in Figure 1.9. This



Configuration: Active(Debug)
Platform: Active(Win32)

Output File	$(ProjectDir)..\lib\Advanced2D.lib
Additional Dependencies	
Additional Library Directories	
Suppress Startup Banner	Yes (/NOLOGO)
Module Definition File Name	
Ignore All Default Libraries	No
Ignore Specific Library	
Export Named Functions	
Force Symbol References	
Use UNICODE Response Files	Yes
Link Library Dependencies	No

Output File
Name of output file with extension. (/OUT:file)

Figure 1.9
Setting the output file for the Visual C++ static library project.

tells the compiler to send the output file up one folder from the project file (which should be \Engine), and from there go *into* a folder called lib. The entire folder path should be \Engine\lib. This is the same folder where we configured Dev-C++ to output its library file. Note the difference in filenames, though: Advanced2D.lib versus libAdvanced2D.a for Dev-C++ (which I will explain later).

Advice

You will need to make the same configuration changes for both Debug and Release builds; otherwise, the Release build will not be configured properly and will not build when you are ready to create the faster version of the engine. In the settings dialog, make the changes as noted and click Apply, then choose Release in the Configuration drop-down list and do the same.

Advice

I recommend disabling the precompiled header option. Open the Project Settings dialog, choose Configuration Properties, then C/C++, then Precompiled Headers. Set the Create/Use Precompiled Headers option as appropriate.

Advice

Dev-C++ object files have an extension of .o, while for Visual C++ it's .obj. The library files of Dev-C++ have an extension of .a, while for Visual C++ it's .lib. These extensions are just the standard output extensions for GCC-based tools. The resulting executable file still ends up being .exe.

Engine Source Code

Every C++ project in this book will have the same basic folder structure. First, there will be a main folder named after the project (for instance, Alien Invaders). This folder will contain the source code files for the project. Contained within this folder will be subfolders named for the compilers that are supported (in the form of project files—.sln for Visual C++ and .dev for Dev-C++). The Dev-C++ folder is called devcpp, while the Visual C++ 2005 folder is called msvc8. If you have a different compiler, you can just follow the basic instructions in this chapter to create the project for your preferred compiler, and if you have a DirectX library available for it, then the code will be shared. This is the *only* way to build a game engine; don't even bother building one for a single compiler, because that is not practical. Even if you are a diehard Microsoft fan, you still need to make project files available for all the various versions of Visual C++ (because none of them are compatible).

Advice

Visual C++ solution files have an extension of .sln. A solution may have multiple project files, which each have an extension of .vcproj.

Within the main project folder (that is, \Engine), there will be a .\bin folder that will contain the compiled executable for a given project. In this .\bin folder you should put any assets that are needed by your game. Obviously this doesn't apply to the engine itself, only to games you build using the engine. (Consider this a free tip for future reference.) Because the .\bin folder is not a default option, we must set it in the project, and this is not an issue now because we are not yet working on an executable program, just a library project.

Advice

An *asset* in the context of a game includes all media and data files used by the game (such as art assets, model assets, audio assets, and so on), and therefore must be distributed with the game.

Are you ready? The code we'll be going over here in order to build the core engine will require tons of serious mental torque! So, it's time to *downshift* and get your RPMs *way up* as we enter the first corner of the proverbial track toward building this engine.

Advanced2D.h

Here is the source code for the Advanced2D.h header file. The code is the same regardless of whether you are using Dev-C++ or Visual C++. The only problems you may experience (other than the usual typos that must be fixed) are linker errors related to the project configuration. This file describes the core structure of the game engine at this point.

```
// Advanced2D Engine
// Main header file

#ifndef _ADVANCED2D_H
#define _ADVANCED2D_H 1

#include <iostream>
#include <windows.h>
#include <d3d9.h>
#include <d3dx9.h>
#include <dxerr9.h>
#include "Timer.h"

#define VERSION_MAJOR 1
#define VERSION_MINOR 0
#define REVISION 0

//external variables and functions
extern bool gameover;
extern bool game_preload();
extern bool game_init(HWND);
extern void game_update();
extern void game_end();

namespace Advanced2D
{
    class Engine {
    private:
        int p_versionMajor, p_versionMinor, p_revision;
        HWND p_windowHandle;
        LPDIRECT3D9 p_d3d;
        LPDIRECT3DDEVICE9 p_device;
        LPDIRECT3DSURFACE9 p_backbuffer;
        LPD3DXSPRITE p_sprite_handler;
        std::string p_apptitle;
        bool p_fullscreen;
        int p_screenwidth;
        int p_screenheight;
        int p_colordepth;
        bool p_pauseMode;
        D3DCOLOR p_ambientColor;
        bool p_maximizeProcessor;
        Timer p_coreTimer;
        long p_frameCount_core;
        long p_frameRate_core;
        Timer p_realTimer;
        long p_frameCount_real;
        long p_frameRate_real;

    public:
        Engine();
        virtual ~Engine();
        int Init(int width, int height, int colordepth, bool fullscreen);
        void Close();
        void Update();
        void message(std::string message, std::string title = "ADVANCED 2D");
        void fatalerror(std::string message, std::string title = "FATAL ERROR");
        void Shutdown();
        void ClearScene(D3DCOLOR color);
        void SetDefaultMaterial();
        void SetAmbient(D3DCOLOR colorvalue);
        int RenderStart();
        int RenderStop();
        int Release();

        //accessor/mutator functions expose the private variables
        bool isPaused() { return this->p_pauseMode; }
        void setPaused(bool value) { this->p_pauseMode = value; }
        LPDIRECT3DDEVICE9 getDevice() { return this->p_device; }
        LPDIRECT3DSURFACE9 getBackBuffer() { return this->p_backbuffer; }
        LPD3DXSPRITE getSpriteHandler() { return this->p_sprite_handler; }
        void setWindowHandle(HWND hwnd) { this->p_windowHandle = hwnd; }
        HWND getWindowHandle() { return this->p_windowHandle; }
        std::string getAppTitle() { return this->p_apptitle; }
        void setAppTitle(std::string value) { this->p_apptitle = value; }
        int getVersionMajor() { return this->p_versionMajor; }
        int getVersionMinor() { return this->p_versionMinor; }
        int getRevision() { return this->p_revision; }
        std::string getVersionText();
        long getFrameRate_core() { return this->p_frameRate_core; };
        long getFrameRate_real() { return this->p_frameRate_real; };
        int getScreenWidth() { return this->p_screenwidth; }
        void setScreenWidth(int value) { this->p_screenwidth = value; }
        int getScreenHeight() { return this->p_screenheight; }
        void setScreenHeight(int value) { this->p_screenheight = value; }
        int getColorDepth() { return this->p_colordepth; }
        void setColorDepth(int value) { this->p_colordepth = value; }
        bool getFullscreen() { return this->p_fullscreen; }
        void setFullscreen(bool value) { this->p_fullscreen = value; }
        bool getMaximizeProcessor() { return this->p_maximizeProcessor; }
        void setMaximizeProcessor(bool value) { this->p_maximizeProcessor = value;}

    }; //class

}; //namespace

//define the global engine object (visible everywhere!)
extern Advanced2D::Engine *g_engine;

#endif
```

Advanced2D.cpp

The Advanced2D.cpp file contains the source code for the `Engine` class. Note that the `Engine` class is embedded inside a namespace called Advanced2D. This was done to keep the `Engine` and its support classes and functions contained to prevent conflicts with other entities in the global namespace.

Advice

Are you getting lost already with these discussions of namespaces and so forth? This is *basic* C++ programming! If you're struggling with it, you'll need a crash course before proceeding. I recommend *Effective C++, 3rd Edition* (Addison-Wesley Professional, 2005) by Scott Meyers. If you are a complete C++ newbie and you need serious help, then read *C++ Programming for the Absolute Beginner* (Course Technology PTR, 2002) by Dirk Henkemans and Mark Lee.

```
// Advanced2D Engine
// Main source code file

//includes
#include "Advanced2D.h"
#include <cstdlib>
#include <ctime>
#include <string>
#include <sstream>
#include <list>
#include "winmain.h"

namespace Advanced2D
{
    Engine::Engine()
    {
        srand((unsigned int)time(NULL));
        p_maximizeProcessor = false;
        p_frameCount_core = 0;
        p_frameRate_core = 0;
        p_frameCount_real = 0;
        p_frameRate_real = 0;
        p_ambientColor = D3DCOLOR_RGBA(255,255,255, 0);
        p_windowHandle = 0;
        p_pauseMode = false;
        p_versionMajor = VERSION_MAJOR;
        p_versionMinor = VERSION_MINOR;
        p_revision = REVISION;

        //set default values
        this->setAppTitle("Advanced2D");
        this->setScreenWidth(640);
        this->setScreenHeight(480);
        this->setColorDepth(32);
        this->setFullscreen(false);

        //window handle must be set later on for DirectX!
        this->setWindowHandle(0);

}

    Engine::~Engine()
    {
          if (this->p_device) this->p_device->Release();
          if (this->p_d3d) this->p_d3d->Release();
    }

    std::string Engine::getVersionText()
    {
        std::ostringstream s;
        s << "Advanced2D Engine v" << p_versionMajor << "." << p_versionMinor
            << "." << p_revision;
        return s.str();
    }

    void Engine::message(std::string message, std::string title)
    {
        MessageBox(0, message.c_str(), title.c_str(), 0);
    }

    void Engine::fatalerror(std::string message, std::string title)
    {
        this->message(message,title);
        Shutdown();
    }

    int Engine::Init(int width, int height, int colordepth, bool fullscreen)
    {
        //initialize Direct3D
        this->p_d3d = Direct3DCreate9(D3D_SDK_VERSION);
        if (this->p_d3d == NULL) {
            return 0;
        }

        //get system desktop color depth
        D3DDISPLAYMODE dm;
        this->p_d3d->GetAdapterDisplayMode(D3DADAPTER_DEFAULT, &dm);

        //set configuration options for Direct3D
        D3DPRESENT_PARAMETERS d3dpp;
        ZeroMemory(&d3dpp, sizeof(d3dpp));
        d3dpp.Windowed = (!fullscreen);
        d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
        d3dpp.EnableAutoDepthStencil = TRUE;
        d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
        d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_IMMEDIATE;
```

```
        d3dpp.BackBufferFormat = dm.Format;
        d3dpp.BackBufferCount = 1;
        d3dpp.BackBufferWidth = width;
        d3dpp.BackBufferHeight = height;
        d3dpp.hDeviceWindow = p_windowHandle;

        //create Direct3D device
        this->p_d3d->CreateDevice(
            D3DADAPTER_DEFAULT,
            D3DDEVTYPE_HAL,
            this->p_windowHandle,
            D3DCREATE_HARDWARE_VERTEXPROCESSING,
            &d3dpp,
            &this->p_device);

        if (this->p_device == NULL) return 0;

        //clear the backbuffer to black
        this->ClearScene(D3DCOLOR_XRGB(0,0,0));

        //create pointer to the back buffer
        this->p_device->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &this-
>p_ backbuffer);

        //use ambient lighting and z-buffering
        this->p_device->SetRenderState(D3DRS_ZENABLE, TRUE);
        this->p_device->SetRenderState(D3DRS_FILLMODE, D3DFILL_SOLID);
        this->SetAmbient(this->p_ambientColor);

        //initialize 2D renderer
        HRESULT result = D3DXCreateSprite(this->p_device, &this->p_sprite_handler);
        if (result != D3D_OK) return 0;

        //call game initialization extern function
        if (!game_init(this->getWindowHandle())) return 0;
        //set a default material
        SetDefaultMaterial();
        return 1;
    }

    void Engine::SetDefaultMaterial()
    {
        D3DMATERIAL9 mat;
        memset(&mat, 0, sizeof(mat));
        mat.Diffuse.r = 1.0f;
        mat.Diffuse.g = 1.0f;
        mat.Diffuse.b = 1.0f;
        mat.Diffuse.a = 1.0f;
        p_device->SetMaterial(&mat);
    }

    void Engine::ClearScene(D3DCOLOR color)
    {
        this->p_device->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
color, 1.0f, 0);
    }

    void Engine::SetAmbient(D3DCOLOR colorvalue)
    {
        this->p_ambientColor = colorvalue;
        this->p_device->SetRenderState(D3DRS_AMBIENT, this->p_ambientColor);
    }

    int Engine::RenderStart()
    {
        if (!this->p_device) return 0;
        if (this->p_device->BeginScene() != D3D_OK) return 0;
        return 1;
    }

    int Engine::RenderStop()
    {
          if (!this->p_device) return 0;
          if (this->p_device->EndScene() != D3D_OK) return 0;
          if (p_device->Present(NULL, NULL, NULL, NULL) != D3D_OK) return 0;
          return 1;
    }

    void Engine::Shutdown()
    {
        gameover = true;
    }

    void Engine::Update()
    {
        static Timer timedUpdate;
        //calculate core framerate
        p_frameCount_core++;
        if (p_coreTimer.stopwatch(999)) {
            p_frameRate_core = p_frameCount_core;
            p_frameCount_core = 0;
        }

        //fast update with no timing
        game_update();

        //update with 60fps timing
        if (!timedUpdate.stopwatch(14)) {
            if (!this->getMaximizeProcessor())
            {
                Sleep(1);
            }
        }
        else {
            //calculate real framerate
            p_frameCount_real++;
            if (p_realTimer.stopwatch(999)) {
                p_frameRate_real = p_frameCount_real;
                p_frameCount_real = 0;
            }

            //begin rendering
            this->RenderStart();

            //done rendering
            this->RenderStop();
        }
    }

    void Engine::Close()
    {
        game_end();
    }
} //namespace
```

Did the code in the `Engine` class seem like a huge and complex detail that we simply skipped over? Not to worry—you will become familiar with it in future chapters. But our immediate goal is to get the core engine built and working in order to delve into advanced rendering in the next chapter. That calls for a blitzkrieg of code. I apologize if the term invokes negative connotations from World War II, but it is a good term—we need to blitz through the basics and get the core engine built quickly, without stopping to regroup until the goal has been achieved.

Timer.h

The `Timer` class provides quick and easy timing facilities to your games, and to the core engine itself in the form of frame-rate estimation and reporting. The `Timer`'s `stopwatch()` method was designed to be self-contained so that you can repeatedly call `stopwatch()` until the specified amount of time (in milliseconds) has passed—at which point the `Timer` object will reset itself for the next call to `stopwatch()`. It is an elegant design that greatly simplifies timing code.

```
/* Timer class provides timing and stopwatch features to the engine */
#pragma once
#include <time.h>
#include <windows.h>
namespace Advanced2D {
class Timer
{
private:
    DWORD timer_start;
    DWORD stopwatch_start;
public:
    Timer(void);
    ~Timer(void);
    DWORD getTimer();
    DWORD getStartTimeMillis();
    void sleep(int ms);
    void reset();
    bool stopwatch(int ms);
};
};
```

Timer.cpp

The implementation of the `Timer` class' methods is contained in the Timer.cpp file. There are several accessor methods, such as `getStartTimeMillis()` (which returns the number of milliseconds since the program started), in addition to the valuable `stopwatch()` method.

```
#include "Timer.h"
namespace Advanced2D {
Timer::Timer(void)
{
    timer_start = timeGetTime();
    reset();
}

Timer::~Timer(void)
{
}

DWORD Timer::getTimer()
{
    return (DWORD) (timeGetTime());
}

DWORD Timer::getStartTimeMillis()
{
    return (DWORD) (timeGetTime() - timer_start);
}


void Timer::sleep(int ms)
{
    DWORD start = getTimer();
    while (start + ms > getTimer());
}

void Timer::reset()
{
    stopwatch_start = getTimer();
}

bool Timer::stopwatch(int ms)
{
    if ( timeGetTime() > stopwatch_start + ms ) {
          stopwatch_start = getTimer();
          return true;
    }
    else return false;
}
};
```

winmain.h

The winmain header file contains just the few `include` statements needed by the winmain source code file (coming up next).

```
#ifndef _WINMAIN_H
#define _WINMAIN_H 1

#define WIN32_LEAN_AND_MEAN
#define WIN32_EXTRA_LEAN
#include <iostream>
#include <windows.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "Advanced2D.h"
#endif
```

winmain.cpp

```
#include <sstream>
#include "winmain.h"
#include "Advanced2D.h"

//macro to read the key states
#define KEY_DOWN(vk) ((GetAsyncKeyState(vk) & 0x8000)?1:0)

HINSTANCE g_hInstance;
HWND g_hWnd;
int g_nCmdShow;

//declare global engine object
Advanced2D::Engine *g_engine;

bool gameover;

//window event callback function
LRESULT WINAPI WinProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
```

```
{
switch( msg )
{
      case WM_QUIT:
      case WM_CLOSE:
      case WM_DESTROY:
            gameover = true;
            break;
        }
return DefWindowProc( hWnd, msg, wParam, lParam );
}

int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int
nCmdShow)
{
    MSG msg;
    srand((unsigned int)time(NULL));
    g_hInstance = hInstance;
    g_nCmdShow = nCmdShow;
    DWORD dwStyle, dwExStyle;
    RECT windowRect;

    /**
      * Create engine object first!
    **/
    g_engine = new Advanced2D::Engine();

    //let main program have a crack at things before window is created
    if (!game_preload()) {
        MessageBox(g_hWnd, "Error in game preload!", "Error", MB_OK);
        return 0;
    }

    //get window caption string from engine
    char title[255];
    sprintf(title, "%s", g_engine->getAppTitle().c_str());

    //set window dimensions
    windowRect.left = (long)0;
    windowRect.right = (long)g_engine->getScreenWidth();
    windowRect.top = (long)0;
    windowRect.bottom = (long)g_engine->getScreenHeight();
```

```
//create the window class structure
WNDCLASSEX wc;
wc.cbSize = sizeof(WNDCLASSEX);

//fill the struct with info
wc.style = CS_HREDRAW | CS_VREDRAW;
wc.lpfnWndProc = (WNDPROC)WinProc;
wc.cbClsExtra       = 0;
wc.cbWndExtra       = 0;
wc.hInstance       = hInstance;
wc.hIcon = NULL;
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = NULL;
wc.lpszMenuName  = NULL;
wc.lpszClassName = title;
wc.hIconSm = NULL;

//set up the window with the class info
RegisterClassEx(&wc);

//set up the screen in windowed or fullscreen mode?

if (g_engine->getFullscreen())
{
    DEVMODE dm;
    memset(&dm, 0, sizeof(dm));
    dm.dmSize = sizeof(dm);
    dm.dmPelsWidth = g_engine->getScreenWidth();
    dm.dmPelsHeight = g_engine->getScreenHeight();
    dm.dmBitsPerPel = g_engine->getColorDepth();
    dm.dmFields = DM_BITSPERPEL | DM_PELSWIDTH | DM_PELSHEIGHT;

if (ChangeDisplaySettings(&dm, CDS_FULLSCREEN) != DISP_CHANGE_SUCCESSFUL) {
    MessageBox(NULL, "Display mode failed", NULL, MB_OK);
    g_engine->setFullscreen(false);
    }

    dwStyle = WS_POPUP;
    dwExStyle = WS_EX_APPWINDOW;
    ShowCursor(FALSE);
}
```

```
    else {
        dwStyle = WS_OVERLAPPEDWINDOW;
        dwExStyle = WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;
    }

    //adjust window to true requested size
    AdjustWindowRectEx(&windowRect, dwStyle, FALSE, dwExStyle);

    //create the program window
    g_hWnd = CreateWindowEx( 0,
        title, //window class
        title, //title bar
        dwStyle | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
        0, 0, //x,y coordinate
        windowRect.right - windowRect.left, //width of the window
        windowRect.bottom - windowRect.top, //height of the window
        0, //parent window
        0, //menu
        g_hInstance, //application instance
        0);     //window parameters

    //was there an error creating the window?
    if (!g_hWnd)      {
        MessageBox(g_hWnd, "Error creating program window!", "Error", MB_OK);
        return 0;
    }

    //display the window
    ShowWindow(g_hWnd, g_nCmdShow);
    UpdateWindow(g_hWnd);

    //initialize the engine
    g_engine->setWindowHandle(g_hWnd);
    if (!g_engine->Init(g_engine->getScreenWidth(), g_engine->getScreenHeight(),
g_engine->getColorDepth(), g_engine->getFullscreen()))      {
        MessageBox(g_hWnd, "Error initializing the engine", "Error", MB_OK);
        return 0;
    }

    // main message loop
    gameover = false;
    while (!gameover)
    {
        while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        g_engine->Update();
    }

    if (g_engine->getFullscreen()) {
        ShowCursor(TRUE);
    }

    g_engine->Close();
    delete g_engine;

    return 1;
}
```

Compiling the Engine Project

Assuming you have typed the code into the specified files without any mistakes, you should be able to compile the engine project. There should be no dependencies for the engine because the compiler assumes that you will provide the needed libs at link time (when you create an executable using the engine's lib). This is a rather complex issue that we'll examine again over the next several chapters as we enhance the engine with new modules and functionality. You should not see any linker errors, only compiler errors if you have made a mistake while typing in the code. If all else fails and you do not understand how to correct an error, then I suggest you copy the core engine project from the CH01 folder on the CD-ROM to your hard drive and try to compile it.

To compile in Dev-C++, press Ctrl+F9 or use the Execute menu. To compile in Visual C++, press Ctrl+Shift+B or use the Build menu.

If *that* project fails, then you have a compiler configuration problem, not a source code problem. Because this is a very challenging issue, I strongly recommend that you not continue to the next step before fully comprehending what's going on with your engine project and its resulting library file. It's *essential* that you understand what's happening at this stage before moving on.

Advice

You will need to set your Visual C++ project to compile with a multi-byte character set rather than Unicode. To do this, open the Project menu, choose Project Settings, Configuration Properties, General, and set the Character Set field to Use Multi-Byte Character Set.

Testing the Engine

Although this has been a long chapter already, I would be remiss if I didn't provide a test project that will allow you to determine whether you have configured the engine properly. We'll write a short test program and configure it so that it will utilize the Advanced2D.lib (or libAdvanced2D.a for Dev-C++).

The TestEngine Source Code

We're just going to jump right into the source code for the library test project, and then I will show you how to configure Dev-C++ and Visual C++ to build with the game engine library. First, create a new Win32 standard executable project using whichever compiler you prefer. Save the project file at the same folder level where you created the Engine folder, so that TestEngine (the name I have used) is in the same root folder as Engine. The reason for this is that we must tell our test project to look "up" one folder into \Engine in order to locate the library file. Remember we had the engine project output the lib into \Engine\lib? That is where we expect it to be located. In other words, you must have compiled the engine already for this test program to work.

Add a new source code file to the project called Main.cpp. This test program is going to be *very small*! Type the code into Main.cpp using whichever compiler you've chosen to use while working through this book. At a certain point, I will stop going over the project creation and configurations and just present source code for study.

```
#include <iostream>
#include "..\Engine\Advanced2D.h"
bool game_preload()
{
    //display engine version in a message box
    g_engine->message(g_engine->getVersionText(), "TEST ENGINE");
    //return fail to terminate the engine
    return false;
}
```

```
bool game_init(HWND hwnd) { return 0;}
void game_update() {}
void game_end() {}
```

See, I told you it was a short one! We're taking advantage of the Windows API `MessageBox()` function (wrapped into the engine via the `message()` method). This function will display a pop-up message box with any text you want to display. Normally the message box is used to report critical errors in the game, but we'll cheat a bit and use it for output! The alternative is to load up a font, initialize the rendering system, and display text on the program window—which, of course, is ridiculously ambitious at this early stage.

There are four unknown functions in this program:

- `bool game_preload()`
- `bool game_init(HWND hwnd)`
- `void game_update()`
- `void game_end()`

These are the first of many such *game events* that will be called automatically by the game engine at key times during the runtime of the game engine. One such time is at the very beginning of WinMain, when `game_preload()` is called. This is a very fortuitous time for us to initialize the game's screen resolution, depth, fullscreen/windowed mode, and other basic settings. It's important to set these things *before* the program window is created. Thus, `game_preload()` is called near the beginning of WinMain.

The `game_init()` event is called after the program window has been created, Direct3D has been initialized, and the rendering device is available for loading textures and mesh objects and so forth. This event function is where you will load game assets.

The last two event functions—`game_update()` and `game_end()`—are self-explanatory. `game_update()` is called once per frame from the *untimed* portion of the game loop. There is also a *timed* portion of the game loop that tries to achieve a stable 60 FPS. We'll get into rendering in the next chapter.

Dev-C++ Library Test Project

Let's double check the configuration of your Dev-C++ project to get it prepared to link in the game engine and all support libraries required at this point

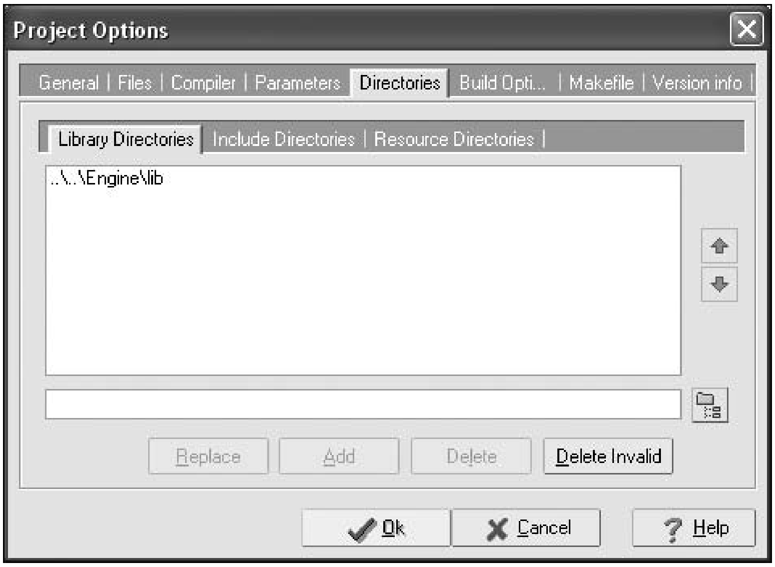

Project Options
General | Files | Compiler | Parameters | Directories | Build Opti... | Makefile | Version info
Library Directories | Include Directories | Resource Directories
..\..\Engine\lib
Replace
Add
Delete
Delete Invalid
Ok
Cancel
Help

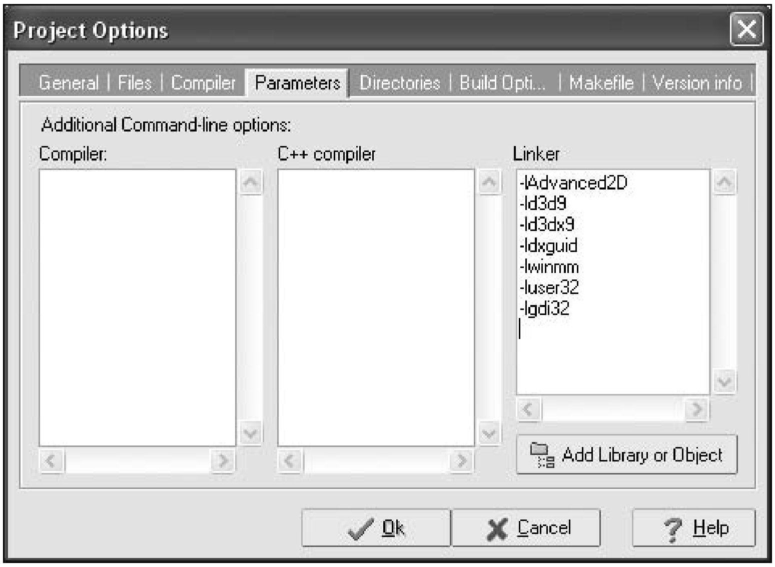
Figure 1.10
Setting the location of the game engine library in the Dev-C++ project.

(including the DirectX libs). Open the Project menu and select Project Options. When you create the project, be sure to add the Main.cpp file located *one folder above* the project folder (that is, \TestEngine\Main.cpp, not \TestEngine\devcpp\Main.cpp, which is incorrect).

First things first. Click the Directories tab and add a new item to the list of Library Directories. We're going to tell Dev-C++ where it can find the engine library. Add an entry with this text: ..\..\Engine\lib (assuming your game engine project is located in the Engine folder—change if needed). See Figure 1.10.

Next, click on the Parameters tab. On the right is a text field labeled Linker. Enter all of the following items into the Linker field:

- -lAdvanced2D
- -ld3d9
- -ld3dx9
- -ldxguid
- -lwinmm


Project Options
General | Files | Compiler | Parameters | Directories | Build Opti... | Makefile | Version info
Additional Command-line options:
Compiler:
C++ compiler
Linker
Add Library or Object
Ok
Cancel
Help

```
-lAdvanced2D
-ld3d9
-ld3dx9
-ldxguid
-lwinmm
-luser32
-lgdi32
```

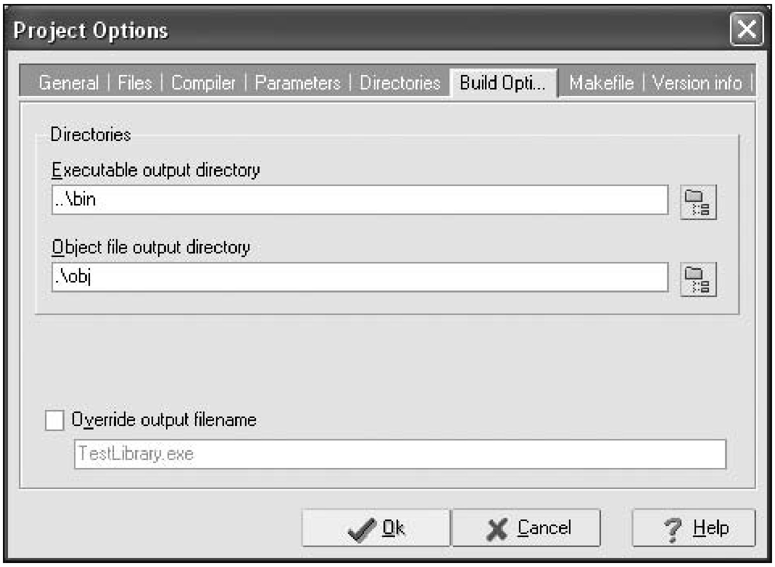
Figure 1.11
The Linker field includes the complete list of libs required by the project.

These are the library files required by the program. The linker is a program that takes all of the object files (.o for GCC) and combines them into a single executable file that's ready to run. See Figure 1.11.

Advice

Although the Dev-C++ library file was specified as libAdvanced2D.a, we do not enter the entire name into the linker options. In GCC-land, the prepended lib and the extension are both assumed. Thus, libAdvanced2D.a is added as a linker option using -lAdvanced2D.

You must make one last setting, and then you can compile the project. We need to tell Dev-C++ where to put the resulting executable file, because the default is not a good location. Do you remember back when you were creating the engine library project and you configured Dev-C++ to send the file to a folder called .\lib? Well, for an executable, we want the output to go to .\bin (which is short for *binary*). The .\bin folder is where you will save game assets that the executable needs to load up, so we might as well get into the habit of doing this for each project now. Open the Project Options again. Click the Build Options tab. In the Executable Output Directory field, enter .\bin. For the Object File Output Directory field, enter .\obj, as shown in Figure 1.12.


Project Options
General | Files | Compiler | Parameters | Directories | Build Opti... | Makefile | Version info
Ok
Cancel
Help

Directories
Executable output directory: ..\bin
Object file output directory: .\obj
Override output filename: TestLibrary.exe

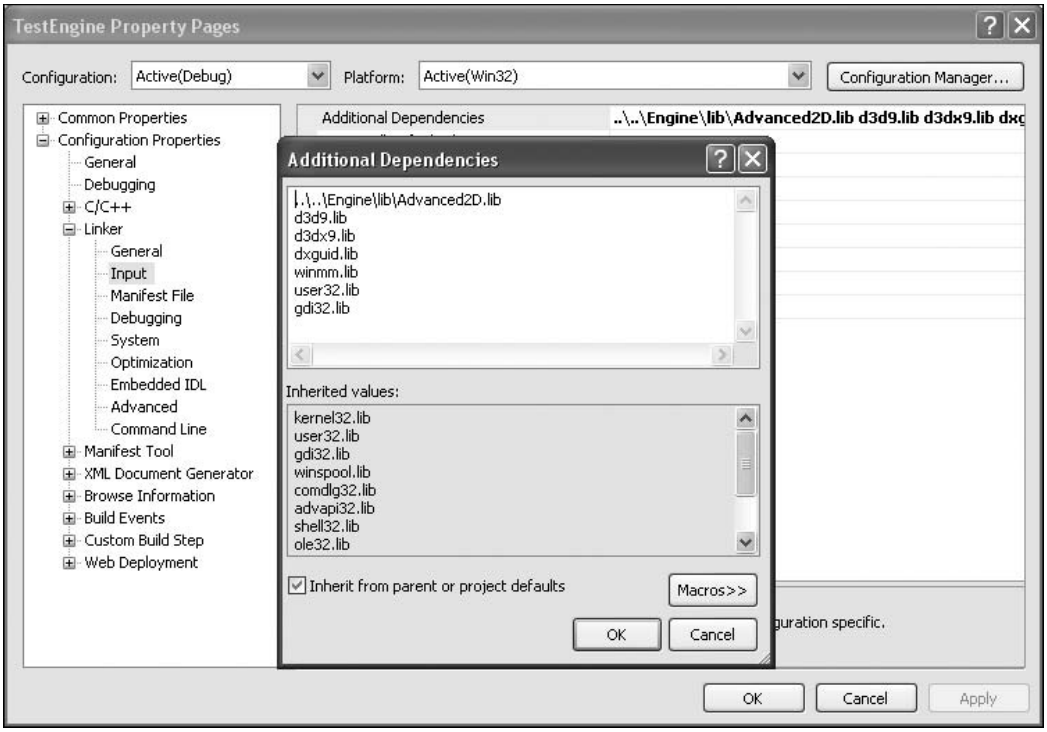
Figure 1.12
Setting the output folders for the project.

Finally, it's time to compile! To compile the program in Dev-C++, press Ctrl+F9 or use the Execute menu. You may also just press F9 to build and run (if there are no compile errors). In order to run the program, you will need to copy the d3dx9.dll file into the same folder as the TestEngine.exe file. This dll is provided on the CD-ROM (duplicated in every chapter's project folders for convenience).

Visual C++ Library Test Project

Visual C++ has the same list of linker files, but the configuration dialog is quite different, so let's see how to do it. We need to tell the linker what libs the project needs in order to run (including the DirectX libs). When you create the project, be sure to add the Main.cpp file located *one folder above* the project folder (that is, \TestEngine\Main.cpp, not \TestEngine\msvc8\Main.cpp, which is incorrect).

Open the Project menu and select Project Options. From the tree-view list, open Linker, Input, as shown in Figure 1.13. In the Additional Dependencies field, add the following list of library files. (You can bring up the mini dialog



Additional Dependencies ..\..\Engine\lib\Advanced2D.lib d3d9.lib d3dx9.lib dxg

Additional Dependencies

```
..\..\Engine\lib\Advanced2D.lib
d3d9.lib
d3dx9.lib
dxguid.lib
winmm.lib
user32.lib
gdi32.lib
```

Inherited values:

```
kernel32.lib
user32.lib
gdi32.lib
winspool.lib
comdlg32.lib
advapi32.lib
shell32.lib
ole32.lib
```

☑ Inherit from parent or project defaults

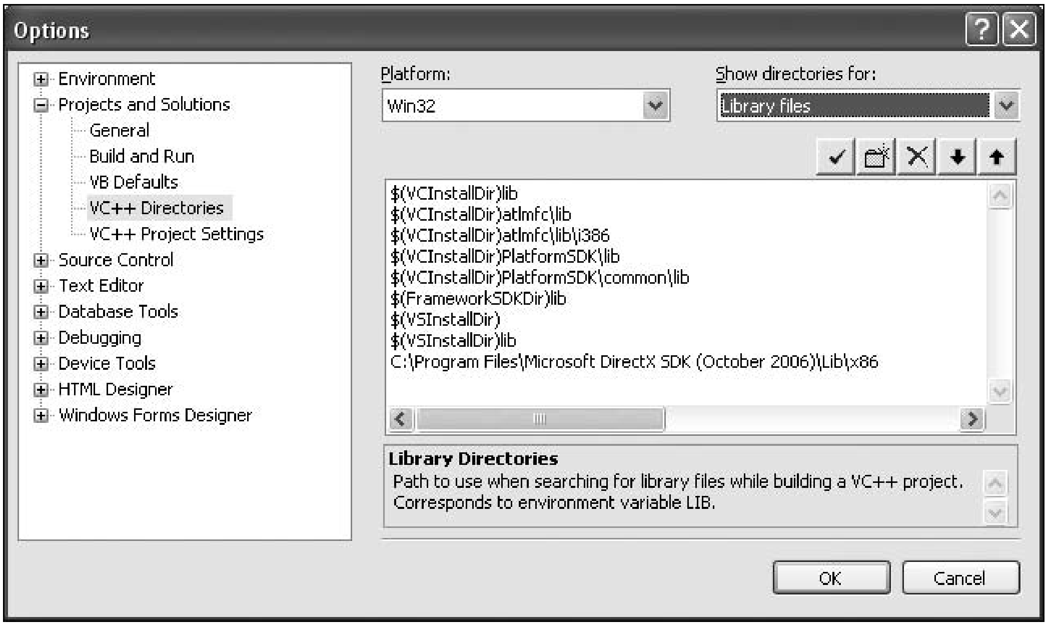
guration specific.

Figure 1.13
Configuring the Visual C++ project's linker dependencies.

shown in the figure by clicking the little ellipsis on the right side of the text field.)

- ..\..\Engine\lib\Advanced2D.lib
- d3d9.lib
- d3dx9.lib
- dxguid.lib
- winmm.lib

The long pathname for Advanced2D.lib is kind of annoying—especially considering that we'll be duplicating this list of libs in every project. You can tell Visual C++ where the lib file is by adding a new folder to the linker search path. This is optional, but it will save time in the long run. However, note that this is a



Platform: Win32

Show directories for: Library files

```
$(VCInstallDir)lib
$(VCInstallDir)atlmfc\lib
$(VCInstallDir)atlmfc\lib\i386
$(VCInstallDir)PlatformSDK\lib
$(VCInstallDir)PlatformSDK\common\lib
$(FrameworkSDKDir)lib
$(VSInstallDir)
$(VSInstallDir)lib
C:\Program Files\Microsoft DirectX SDK (October 2006)\Lib\x86
```

Library Directories
Path to use when searching for library files while building a VC++ project. Corresponds to environment variable LIB.

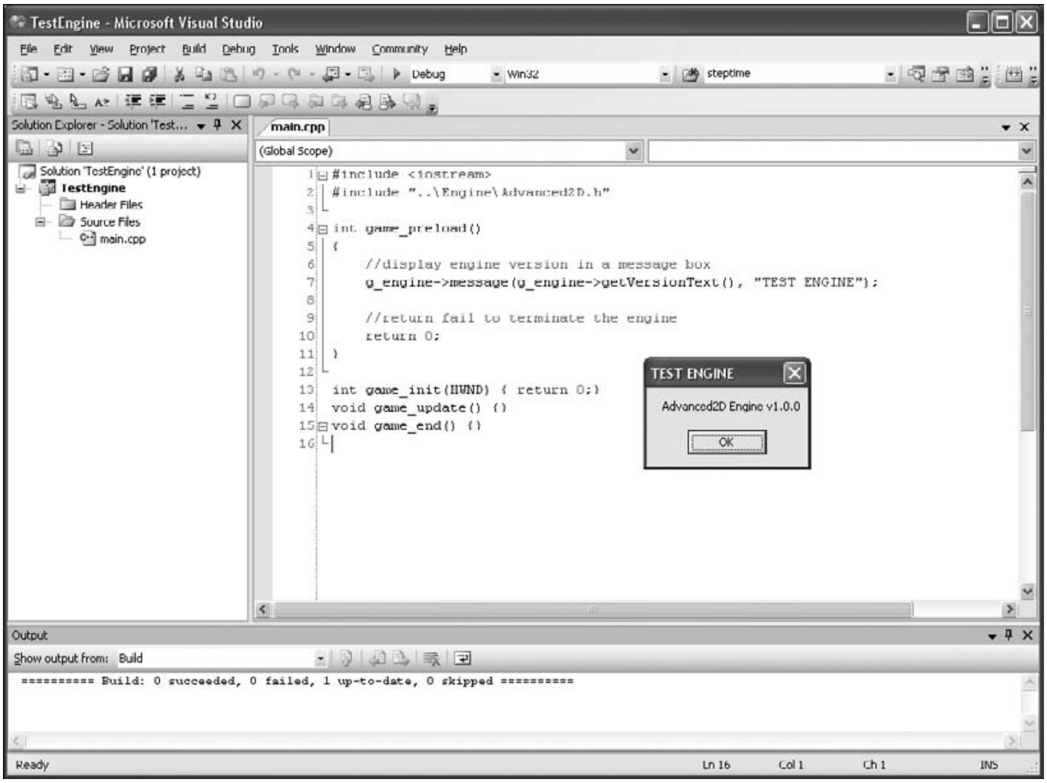
Figure 1.14
Adding the engine's lib folder to the compiler's library search path (optional).

user preference setting, not a project setting, so the new linker path will not be saved with the project file.

Open the Tools menu and select Options. In the tree-view list, expand Projects and Solutions and select VC++ Directories. There is a drop-down list on the right side where you can choose Library Files. Add the folder to this list where your engine's lib file is located, and then you can specify simply Advanced2D.lib in the linker configuration without needing to prepend the relative folder location. See Figure 1.14.

To compile in Visual C++, press Ctrl+Shift+B or use the Build menu. If you have no errors in your code, you can press F5 to build and run the program. If the program compiles without error and runs, you should see the message box pop up, as shown in Figure 1.15.

What you do not see in this simple example is the program window coming up automatically. I've short-circuited the window from appearing by returning 0 in `game_preload()`. When the preload function fails, the game engine shuts down (assuming that something catastrophic failed). If you want to see the program


TestEngine - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Community Help
Debug
Win32
steptime
Solution Explorer - Solution 'Test...
Solution 'TestEngine' (1 project)
TestEngine
Header Files
Source Files
main.cpp
main.cpp
(Global Scope)
TEST ENGINE
OK
Output
Show output from: Build
Ready
Ln 16
Col 1
Ch 1
INS

```
#include <iostream>
#include "..\Engine\Advanced2D.h"

int game_preload()
{
    //display engine version in a message box
    g_engine->message(g_engine->getVersionText(), "TEST ENGINE");

    //return fail to terminate the engine
    return 0;
}

int game_init(HWND) { return 0;}
void game_update() {}
void game_end() {}
```

```
Advanced2D Engine v1.0.0
```

```
========== Build: 0 succeeded, 0 failed, 1 up-to-date, 0 skipped ==========
```

Figure 1.15
It's working! It's working!

window come up with Direct3D rendering (doing nothing but clearing the window, but functioning nonetheless), change the return 0 to return 1 in `game_preload()`.

That's all for now. You should now have a functioning core engine that is eager to start rendering, so let's move on to the next chapter to do just that.

This page intentionally left blank

CHAPTER 2

3D Rendering

Rendering is the process of transforming an entity's data into a visual representation. I hesitate to use the terms "two-dimensional" or "three-dimensional" explicitly because it's possible to render in more ways than what is viewed through a computer monitor. We cannot limit the *theory* to a simple computer monitor because it's now possible to scan a 3D object, as well as *sculpt* a 3D object. This technology is called *3D printing*. For our purposes, though, we'll be learning about rendering graphics on a monitor. In this chapter, you will learn about the rendering system of a game engine, and you will add the rendering module to the Advanced2D engine (created in the first chapter).

We're attempting to tackle a rather large and complex subject in a single chapter, so it might seem a bit overwhelming at first. But if you study the code, you'll see that it just builds on the basic engine project started in the previous chapter. This is called the *iterative process* of software development, and it is a flexible, robust way to write code. (The inflexible, or brittle, method would be to write an entire engine all up front and then work out its subsequent bugs, also all at once.) When you are writing a lot of code, and especially when you are building a complex system such as a game engine, you want to have a *fast* iterative process.

What does iterative mean? Iteration is a repetitive action. When building code, a single step of iteration is to write some code, compile, fix any mistakes, and test. That's right, *test*. A fast iterative process is helpful only if the code you are writing is free of syntax errors as well as logic errors. One of the best ways to test while building is to have a large-scale project in mind for the test subject, such as a graphics demo or a game.

So, what do we need to do to get started? Let's begin by opening the Advanced2D project from the previous chapter, and then add to it. This project is the *game engine,* and you will write a *client* or *test* program that will consume the engine's functionality in order to do something useful. That is the whole point of a game engine, after all—to simplify the front-end code, or rather, the code that you must write for each new game. A game engine separates the programmer from the platform-specific APIs and SDKs (such as Direct3D) as much as possible.

Advice

Everything in this chapter—the text of these pages, the Direct3D source code, the C++ projects, the 3D mesh files, the textures—was developed using completely *free* open-source software—and I'm not talking about any Express software either. Just to prove that it is possible, I installed Windows XP on a new hard drive and neglected to install any Microsoft tools—not even the DirectX SDK. And believe it or not, it *is* entirely possible to develop a high-end Direct3D game using free software, *without* any funky workarounds.

- Words: OpenOffice 2.4 (www.openoffice.org)
- Meshes: Blender (www.blender.org)
- Images: GIMP 2.4 (www.gimp.org)
- Programs: Dev-C++ 5.0 (www.bloodshed.net)
- Graphics: Direct3D 9.0c DevPak (www.g-productions.net)

Of course, if you *want* to use Visual C++ 2005 with Microsoft's official DirectX SDK, you're welcome to—the code is all the same. When we get into subjects such as audio, scripting, and level editing, I'll share with you the free tools used for those purposes as well!

Rendering Basics

I understand your double-take upon reading this chapter title. After all, this is supposed to be a book that teaches advanced 2D graphics programming, right? Yes, indeed it is. But today it's a given that even a retro-style 2D game may have some 3D features. For instance, you can do some really nice special effects with shaders in a pseudo-2D game—that is, an essentially 3D game played in a 2D orientation with only width and height represented (and lacking the third dimension of depth). A good example of this sort of game is *Sid Meier's*

Civilization IV, which is an advanced 3D game with a flat board-game style orientation. The simple fact is that we can't do any advanced 2D rendering (the goal of the book) without at least touching upon the subject of 3D rendering first. After all, the 2D output goes through the 3D system with a fixed camera angle.

Adding Rendering Support

We will not be covering the theory of 3D graphics programming in great detail, although a brief discussion of each major issue is needed. I will just present you with the key concepts, and we'll take it one step at a time. First, we need to add rendering to the game engine that currently doesn't know how to render. As you'll recall from the last chapter, there are currently only four methods in the external game functions:

```
bool game_preload();
bool game_init();
void game_update();
void game_end();
```

Although we can do some initialization and updating, there is no way to render anything with this game engine. The best we were able to do last chapter was display a message in a message box.

Advice

The Advanced2D engine is being presented in a step-by-step format as an aid to learning, but as you might imagine, a game engine grows quite large very quickly, and our engine here is no exception. The "follow along and type in the code" style of learning will not survive beyond this chapter because there is too much code—even though the engine is being developed in parallel with each chapter. Instead, in future examples, we will just go over the engine code with the assumption that you have opened the project available on the CD-ROM.

Do you have the Advanced2D project open? Bring up the winmain.cpp file again and scroll down to the bottom, where the `while` loop is located. This is the primary loop of the game engine. The `PeekMessage`, `TranslateMessage`, and `DispatchMessage` function calls are just part of the Windows "message pump" that is standard in every `winmain` function. What I want you to take notice of is the call to `g_engine->Update()` located in the loop:

```
    // main message loop
    gameover = false;
    while (!gameover)
    {
        while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        g_engine->Update();
    }
```

Although only one function call is located here, this single function call will give power to the entire game engine. Think of this `winmain` loop as the *heart* of the system, and that `Update` function call is the pulse. Or, from the engineering point of view, `winmain` is the distributor, and the `Update` function is the electrical pulses that fire the spark plugs. We will be doing a *lot* of things within that single function. Now you may close the winmain.cpp file because it is complete, and no changes are needed.

Next, let's open the Advanced2D.h and Advanced2D.cpp files. You might recall from the previous chapter that these are the main source code files for the engine itself. Open the Advanced2D.h file and add the following line of code noted in bold. This is the next function we'll add to the engine, giving our client source code file the ability to render something.

```
//external variables and functions
extern bool gameover;

extern bool game_preload();
extern bool game_init(HWND);
extern void game_update();
extern void game_end();
extern void game_render3d();
```

This new function must be added to your game's source code because now the engine expects this function to exist. If you forget to add it to your game, you will get a linker error when you try to build the project. The engine is already capable of rendering 3D objects, and that will be possible as soon as we add the call to `game_render3D` to the engine's `Update` method. Why don't we just do that right now, while we're on the subject?

Scroll down in Advanced2D.cpp until you find the `Engine::Update` method. You'll be adding a single line of code, a call to `game_render3D()`, as discussed previously. By adding this function call *between* the `RenderStart` and `RenderStop` functions, we effectively give the game the ability to do 3D rendering. Here's the whole `Update` method with the new lines highlighted:

```
void Engine::Update()
{
    static Timer timedUpdate;

    //calculate core framerate
    p_frameCount_core++;
    if (p_coreTimer.stopwatch(999)) {
        p_frameRate_core = p_frameCount_core;
        p_frameCount_core = 0;
    }


    //fast update with no timing
    game_update();

    //update with 60fps timing
    if (!timedUpdate.stopwatch(14)) {
        if (!this->getMaximizeProcessor())
        {
            Sleep(1);
        }
    }
    else {
        //calculate real framerate
        p_frameCount_real++;
        if (p_realTimer.stopwatch(999)) {
              p_frameRate_real = p_frameCount_real;
              p_frameCount_real = 0;
        }

        //begin rendering
        this->RenderStart();

        //allow game to render
        game_render3d();

        //done rendering
        this->RenderStop();
    }
}
```

We will come back to this method again in the next chapter, when we add 2D rendering support to the engine.

Adding Camera Support

The two most important considerations when writing a renderer for the first time are the *camera* and the *light source*. If either of these is set improperly, you might see nothing on the screen and incorrectly assume that nothing is being rendered. In fact, something *is* often being rendered even when you don't see anything, but due to the camera's orientation or the lighting conditions of the scene, you might not see anything. Remember those two issues while you peruse the code in this chapter—camera and lighting.

Let's add camera support to the Advanced2D engine. The camera determines what you see on the screen. The camera may be *positioned* anywhere in 3D space, as well as *pointed* in any direction in 3D space. The following Camera.h definition provides support for the projection and view matrices for our engine. Because we aren't truly studying 3D rendering—only using Direct3D as a tool for our upcoming 2D game projects—I will leave the details to another resource. Note that the Camera.h file has already been added to the list of included files in Advanced2D.h. So any new project you create using the Advanced2D engine may simply include Advanced2D.h, rather than all of the individual header files.

Advice

Here are two good reference books that will teach you the ins and outs of Direct3D rendering: *3D Game Engine Programming* (Thomson Course Technology PTR, 2004) by Stefan Zerbst and Oliver Duvel and *Advanced Visual Effects with Direct3D* (Thomson Course Technology PTR, 2005) by Peter Walsh.

```
#pragma once
#include "Advanced2D.h"

namespace Advanced2D {
class Camera
{
private:
    D3DXMATRIX p_matrixProj;
    D3DXMATRIX p_matrixView;
    D3DXVECTOR3 p_updir;

    D3DXVECTOR3 p_position;
    D3DXVECTOR3 p_target;
    float p_nearRange;
    float p_farRange;
    float p_aspectRatio;
    float p_fov;
public:
    Camera(void);
    ~Camera(void);
    void setPerspective(float fov, float aspectRatio, float nearRange,
float farRange);
    float getNearRange() { return p_nearRange; }
    void setNearRange(float value) { p_nearRange = value; }
    float getFarRange() { return p_farRange; }
    void setFarRange(float value) { p_farRange = value; }
    float getAspectRatio() { return p_aspectRatio; }
    void setAspectRatio(float value) { p_aspectRatio = value; }
    float getFOV() { return p_fov; }
    void setFOV(float value) { p_fov = value; }
    void Update();

    D3DXVECTOR3 getPosition() { return p_position; }
    void setPosition(float x, float y, float z);
    void setPosition(D3DXVECTOR3 position);
    float getX() { return p_position.x; }
    void setX(float value) { p_position.x = value; }
    float getY() { return p_position.y; }
    void setY(float value) { p_position.y = value; }
    float getZ() { return p_position.z; }
    void setZ(float value) { p_position.z = value; }

    D3DXVECTOR3 getTarget() { return p_target; }
    void setTarget(D3DXVECTOR3 value) { p_target = value; }
    void setTarget(float x, float y, float z) {
        p_target.x = x; p_target.y = y; p_target.z = z;
    }
};
};
```

Now for the Camera.cpp source code. Because our header file included so many accessors and mutators, the implementation file is rather short in comparison.

```
#include "Camera.h"

namespace Advanced2D {
```

```
Camera::Camera(void)
{
    p_position = D3DXVECTOR3(0.0f,0.0f,10.0f);
    p_updir = D3DXVECTOR3(0.0f,1.0f,0.0f);

    //hard coded to 1.3333 by default
    float ratio = 640 / 480;
    setPerspective(3.14159f / 4, ratio, 1.0f, 2000.0f);
}

Camera::~Camera(void) { }

void Camera::setPerspective(float fov, float aspectRatio, float nearRange,
float farRange)
{
    this->setFOV(fov);
    this->setAspectRatio(aspectRatio);
    this->setNearRange(nearRange);
    this->setFarRange(farRange);
}


void Camera::Update()
{
    //set the camera's perspective matrix
    D3DXMatrixPerspectiveFovLH(&this->p_matrixProj, this->p_fov,
this->p_aspectRatio, this->p_nearRange, this->p_farRange);
    g_engine->getDevice()->SetTransform(D3DTS_PROJECTION, &this->p_matrixProj);

    //set the camera's view matrix
    D3DXMatrixLookAtLH(&this->p_matrixView, &this->p_position,
&this->p_target, &this->p_updir);
    g_engine->getDevice()->SetTransform(D3DTS_VIEW, &this->p_matrixView);
}

void Camera::setPosition(float x, float y, float z)
{
    this->p_position.x = x;
    this->p_position.y = y;
    this->p_position.z = z;
}
```

```
void Camera::setPosition(D3DXVECTOR3 position)
{
    this->setPosition(position.x, position.y, position.z);
}
};
```

Be sure to add both files (Camera.h and Camera.cpp) to your engine project. Or you may just open the Engine project located on the CD-ROM in this chapter's folder.

Adding Mesh Support

Direct3D provides some procedural mesh creation functions that we can use to create a mesh at runtime without having to load a mesh from a file. Arguably, loading a mesh is a simple process because Direct3D can handle that task, too. But, first things first—let's focus on generating and then rendering a simple geometric shape. We're going to quickly go over a `Mesh` class that encapsulates the best Direct3D can give us with regard to mesh loading and rendering support. I will assume that either you are already familiar with this code or you have another reference available because we aren't going to discuss the details. (Refer to the two books I suggested earlier.)

There are three important tasks that I want our `Mesh` class to handle automatically. First, it should be able to quickly and easily load a mesh file (from the .X format). Second, it should handle all transformations internally and provide a mechanism for easily moving, rotating, and scaling the mesh. Finally, it should be able to render itself (via the Direct3D device, of course).

Advice

This is the very same mesh code introduced in *Beginning Game Programming, 2nd Edition* (Muska & Lipman, 2003)! But now it is nicely packaged in a class with additional functionality, such as automatic rotation, scaling, and velocity-based movement.

Now let's see the implementation of the `Mesh` class in the Mesh.cpp file. You will find this class already included in the new Engine project on the CD-ROM. Note that the Mesh.h file has already been added to the list of included files in Advanced2D.h. So any new project you create using the Advanced2D engine may simply include Advanced2D.h, rather than all of the individual header files.

```
#include "Mesh.h"
#include <string>
namespace Advanced2D {
```

```
Mesh::Mesh(void)
{
    mesh = 0;
    materials = 0;
    d3dxMaterials = 0;
    matbuffer = 0;
    material_count = 0;
    textures = 0;
    position = D3DXVECTOR3(0.0f,0.0f,0.0f);
    velocity = D3DXVECTOR3(0.0f,0.0f,0.0f);
    rotation = D3DXVECTOR3(0.0f,0.0f,0.0f);
    scale = D3DXVECTOR3(1.0f,1.0f,1.0f);
}

Mesh::~Mesh(void)
{
    if (materials != NULL) delete[] materials;

    //remove textures from memory
    if (textures != NULL) {
        for( DWORD i = 0; i < material_count; i++)
        {
            if (textures[i] != NULL)
                textures[i]->Release();
        }
        delete[] textures;
    }
    if (mesh != NULL) mesh->Release();
}

int Mesh::GetFaceCount()
{
    return this->mesh->GetNumFaces();
}

int Mesh::GetVertexCount()
{
    return this->mesh->GetNumVertices();
}

bool Mesh::Load(char* filename)
```

```
{
    HRESULT result;

    //load mesh from the specified file
    result = D3DXLoadMeshFromX(
        filename,                       //filename
        D3DXMESH_SYSTEMMEM,             //mesh options
        g_engine->getDevice(),          //Direct3D device
        NULL,                           //adjacency buffer
        &matbuffer,                     //material buffer
        NULL,                           //special effects
        &material_count,                  //number of materials
        &mesh);                           //resulting mesh

    if (result != D3D_OK) {
        return false;
    }

    //extract material properties and texture names from material buffer
    d3dxMaterials = (LPD3DXMATERIAL)matbuffer->GetBufferPointer();
    materials = new D3DMATERIAL9[material_count];
    textures  = new LPDIRECT3DTEXTURE9[material_count];

    //create the materials and textures
    for(DWORD i=0; i < material_count; i++)
    {
        //grab the material
        materials[i] = d3dxMaterials[i].MatD3D;

        //set ambient color for material
        materials[i].Ambient = materials[i].Diffuse;
        //materials[i].Emissive = materials[i].Diffuse;
        materials[i].Power = 0.5f;
        //materials[i].Specular = materials[i].Diffuse;

        textures[i] = NULL;
        if( d3dxMaterials[i].pTextureFilename != NULL &&
            lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
        {
            //load texture file specified in .x file
            result = D3DXCreateTextureFromFile(g_engine->getDevice(),
d3dxMaterials[i].pTextureFilename, &textures[i]);
```

```
            if (result != D3D_OK) {
                return false;
            }
        }
    }

    //done using material buffer
    matbuffer->Release();

    return true;
}


void Mesh::CreateSphere(float radius, int slices, int stacks)
{
    D3DXCreateSphere(g_engine->getDevice(), radius, slices, stacks, &mesh,
NULL);
}

void Mesh::CreateCube(float width, float height, float depth)
{
    D3DXCreateBox(g_engine->getDevice(), width, height, depth, &mesh, NULL);
}

void Mesh::Draw()
{
    if (material_count == 0) {
        mesh->DrawSubset(0);
    }
    else {
        //draw each mesh subset
        for( DWORD i=0; i < material_count; i++ )
        {
            // Set the material and texture for this subset
            g_engine->getDevice()->SetMaterial( &materials[i] );

            if (textures[i])
            {
                if (textures[i]->GetType() == D3DRTYPE_TEXTURE)
                {
                    D3DSURFACE_DESC desc;
                    textures[i]->GetLevelDesc(0, &desc);
                    if (desc.Width > 0) {
                        g_engine->getDevice()->SetTexture( 0, textures[i] );
                    }
                }
            }
            // Draw the mesh subset
            mesh->DrawSubset( i );
        }
    }
}

void Mesh::Transform()
{
    //set rotation matrix
    float x = D3DXToRadian(rotation.x);
    float y = D3DXToRadian(rotation.y);
    float z = D3DXToRadian(rotation.z);
    D3DXMatrixRotationYawPitchRoll(&matRotate, x, y, z);

    //set scaling matrix
    D3DXMatrixScaling(&matScale, scale.x, scale.y, scale.z);

    //set translation matrix
    D3DXMatrixTranslation(&matTranslate, position.x, position.y, position.z);

    //transform the mesh
    matWorld = matRotate * matScale * matTranslate;
    g_engine->getDevice()->SetTransform(D3DTS_WORLD, &matWorld);
}

void Mesh::Rotate(D3DXVECTOR3 rot)
{
    Rotate(rot.x,rot.y,rot.z);
}

void Mesh::Rotate(float x,float y,float z)
{
    rotation.x += x;
    rotation.y += y;
    rotation.z += z;
}

void Mesh::Update()
```

```
{
    position.x += velocity.x;
    position.y += velocity.y;
    position.z += velocity.z;
}

void Mesh::LimitBoundary(float left,float right,float top,float bottom,float
back,float front)
{
    if (position.x < left || position.x > right) {
        velocity.x *= -1;
    }
    if (position.y < bottom || position.y > top) {
        velocity.y *= -1;
    }
    if (position.z < front || position.z > back) {
        velocity.z *= -1;
    }
}
}; //namespace
```

Rendering Meshes

I think it's time for a respite from engine coding for a little while. We need to see whether the new 3D rendering capabilities of our engine are indeed working, so a demo is in order. Let's start simple, with a generic cube. Our cube will be rendered with ambient lighting for now. This is just a case where we want to get *something* up on the screen. It's our first render test, after all! This is the part where all our hard work on the engine begins to pay off. Not only will you *never* have to write any of the previous segments of code again, but it's very likely that you will never even have to *look* at that code again. Instead, we'll just focus on game code, so to speak.

Runtime Cubes

Now let's write the code for the Cube demo program. This will be a *new project.* First we set the basic properties in the `game_preload` event. In `game_init` we create the cube and set the ambient lighting. Finally, in `game_render3D` we clear the scene, set the identity (thus returning the current focus to the origin), and then rotate and render the cube.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;
Camera *camera;
Mesh *mesh;

bool game_preload()
{
    g_engine->setAppTitle("CUBE DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(1024);
    g_engine->setScreenHeight(768);
    g_engine->setColorDepth(32);
    return true;
}

bool game_init(HWND)
{
    //create a cube
    mesh = new Mesh();
    mesh->CreateCube(2.0f, 2.0f, 2.0f);

    //set the camera and perspective
    camera = new Camera();
    camera->setPosition(0.0f, 2.0f, 6.0f);
    camera->setTarget(0.0f, 0.0f, 0.0f);
    camera->Update();

    //set the ambient color
    g_engine->SetAmbient(D3DCOLOR_XRGB(40,40,255));

    return true;
}

void game_update() { }
void game_end()
{
    delete camera;
    delete mesh;
}

void game_render3d()
{
    //clear the scene using a dark blue color
    g_engine->ClearScene(D3DCOLOR_RGBA(30,30,100, 0));

    //return to the origin
    g_engine->SetIdentity();

    //rotate and draw the cube
    mesh->Rotate(2.0f, 0.0f, 0.0f);
    mesh->Transform();
    mesh->Draw();
}
```

In order to compile this program using the Advanced2D engine's new rendering capabilities, you will need to provide your Cube program with the following library files (added to the linker options):

- Advanced2D
- d3d9
- d3dx9
- dxguid
- winmm

For each file listed, prepend the file with -l (lowercase letter L) for Dev-C++, or append .lib for Visual C++. If you aren't sure how to do this, refer to the previous chapter.

All things considered, this is a ridiculously short code listing even if we are just rendering a flat-shaded cube with ambient lighting (see Figure 2.1).

Bouncing Balls

I can think of a few things already, just after a quick perusal of that last source code listing, where we might move even more of the code into the engine. For example, you will *always* need to clear the scene and set the identity. Secondly, I would like to abstract the color system into a more generic RGBA (red-green-blue-alpha) set, rather than using the Direct3D-based color macros—something to consider for a future engine update.

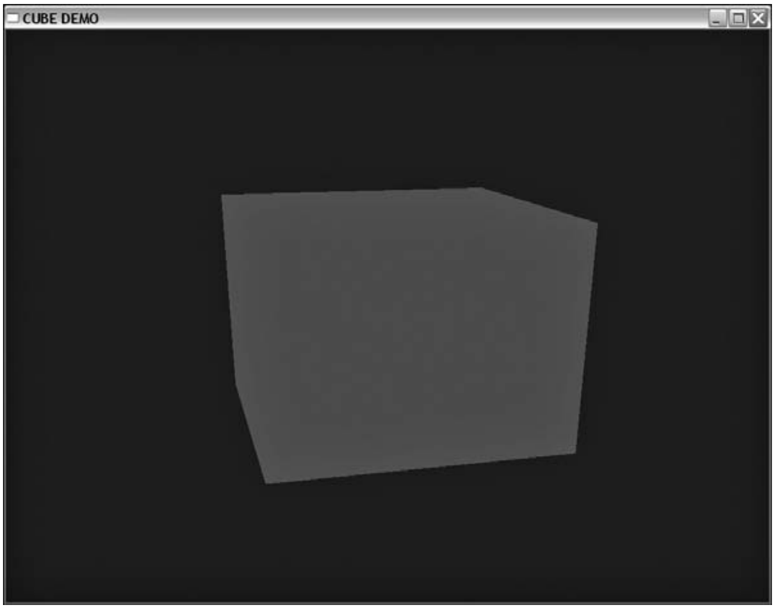


Figure 2.1
The Cube demo renders a flat-shaded cube with ambient lighting.

Advice

A *mesh* is the technical term for a 3D model, but the 3D industry does not use the word *model* because it is not specific. You might have noticed that our `Mesh` class has the ability to load a .X file, which will usually contain references to texture files (included with the .X file but not embedded).

Direct3D is a state-based rendering system, which means it continues to operate as it is currently set until something is changed. When you set the current texture in the Direct3D device, it will obediently use that texture for all rendering output until it is told to change the texture.

If we were building an advanced 3D rendering system, one optimization would be to create a texture cache so that Direct3D's state wouldn't have to be changed so often. Basically, you figure out which textures are shared by all polygons in the scene, and you render all of those polys before changing the texture and rendering all polys that use the new texture, and so on.

Advice

Changing state in a rendering system (such as changing the current texture or shader program) is a very time-consuming process and should be done infrequently. But for the purposes of demonstration and learning, don't be concerned with performance until later.

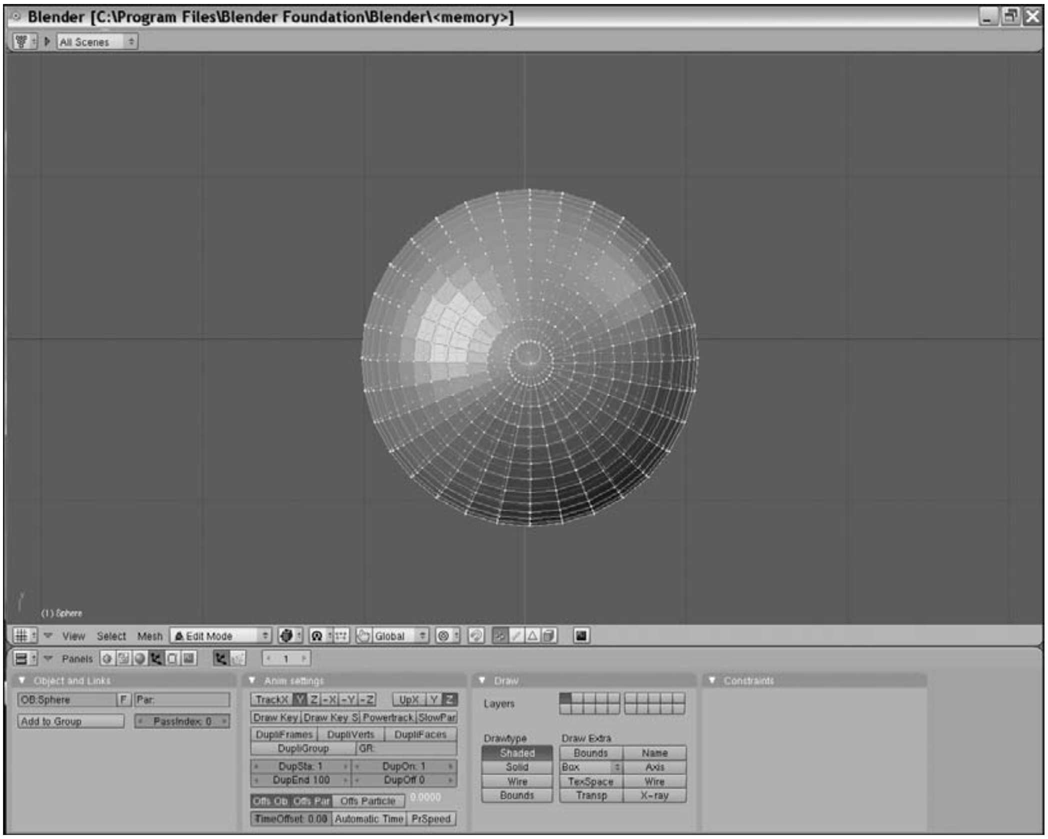


Figure 2.2
Using Blender to create the sphere mesh used in BouncingBalls.

Our BouncingBalls program is a bit of a leap beyond the crude Cube program, as it features a C++ Standard Library container called `vector` to manage the balls (or rather, spheres) in this demonstration program. A vector is a good general-purpose container that mimics a C++ array in many ways, including the ability to index into the "array" using brackets and an index (for instance, `spheres[1]`). Instead of creating the sphere at runtime (as we did previously with the cube), the sphere will be loaded from a .X file. The sphere was created, textured, and exported using Blender, as shown in Figure 2.2.

Advice

If you are not familiar with the C++ Standard Library, I encourage you to pick up a good book on the subject because it's crucial to making a game engine. One good reference is *C++ Standard Library Practical Tips* (Charles River Media, 2005) by Greg Reese.

Since the `Camera` and `Mesh` class header files have already been included in Advanced2D.h, you will not need to include them separately in this or any other new project; you need only include Advanced2D.h. As is the common practice in this book, each new program listing assumes that it is part of a *new project.* If you have not already done so, create a new project (as described in the previous chapter) and name it BouncingBalls. Include the same list of linked library files shown for the previous example program.

```
#include <vector>
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

//we need this keyboard macro to detect Escape key
#define KEY_DOWN(vk) ((GetAsyncKeyState(vk) & 0x8000)?1:0)

//camera object
Camera *camera;

//define the number of spheres
#define SPHERES 100

//create the entity vector and iterator
typedef std::vector<Mesh*>::iterator iter;
std::vector<Mesh*> entities;

bool game_preload()
{
    g_engine->setAppTitle("BOUNCING BALLS");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(1024);
    g_engine->setScreenHeight(768);
    g_engine->setColorDepth(32);
    return true;
}

bool game_init(HWND)
{
    srand(time(NULL));

    //set the camera and perspective
    camera = new Camera();
    camera->setPosition(0.0f, 2.0f, 10.0f);
    camera->setTarget(0.0f, 0.0f, 0.0f);
    camera->Update();

    //create ball meshes
    Mesh *ball;
    for (int n=0; n<SPHERES; n++)
    {
        ball = new Mesh();
        ball->Load("ball.x");
        ball->SetScale(0.3f,0.3f,0.3f);
        ball->SetPosition(0.0f,0.0f,0.0f);
        float x = (float)(rand()%8+1) / 100.0f;
        float y = (float)(rand()%8+1) / 100.0f;
        float z = (float)(rand()%8+1) / 100.0f;
        ball->SetVelocity(x,y,z);
        ball->SetRotation(0.1f,0.2f,0.01f);

        //add this ball to the vector container
        entities.push_back(ball);
    }
    return true;
}

void game_update()
{
    //update entity positions and limit the boundary
    for (iter i = entities.begin(); i != entities.end(); ++i)
    {
        (*i)->Update();
        (*i)->LimitBoundary(-5,5,4,-4,4,-4);
    }

    //escape key will terminate the program
    if (KEY_DOWN(VK_ESCAPE)) g_engine->Close();
}

void game_end()
{
    delete camera;

    //destroy all balls from the vector
    for (iter i = entities.begin(); i != entities.end(); ++i) {
        delete *i;
    }
    //empty the vector
    entities.clear();
}

void game_render3d()
{
    static DWORD start=0;

    //clear the scene using a dark blue color
    g_engine->ClearScene(D3DCOLOR_RGBA(30,30,100,0));

    //return to the origin
    g_engine->SetIdentity();

    //draw entities
    for (iter i = entities.begin(); i != entities.end(); ++i)
    {
        //remember, every entity must be moved individually!
        (*i)->Transform();
        (*i)->Draw();
    }
}
```

This program is far more complex than the Cube demo, and yet the source code listing is only a few lines longer! Why do you suppose that is? First of all, our engine is handling most of the details for us now, but we also benefited from the use of a standard vector, which simplified the code. Figure 2.3 shows the output from the BouncingBalls program. There is no rhyme or reason behind this program; it just sets the balls at random X,Y,Z velocities and lets them go. The `Mesh` class automatically updates the position of each ball based on its position and velocity when the `Update()` method is called.

Direct Lighting

Direct3D supports three types of direct lighting in addition to the ambient level:

- Point light
- Spot light
- Directional light

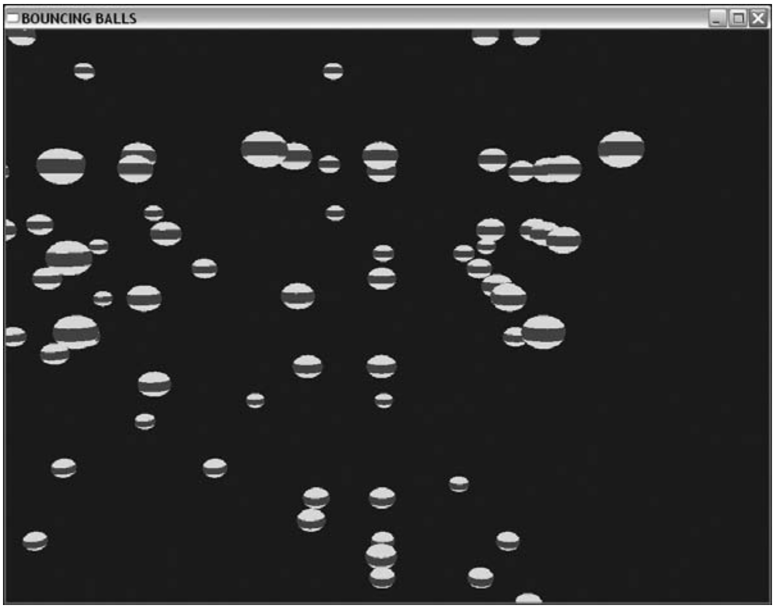


Figure 2.3
The BouncingBalls program demonstrates how to load, manipulate, and render a mesh.

You can use up to eight lights in your game, and can define any one of the lights (0 to 7) as a point, spot, or directional light. Let's take a look at how to create each of these three lights in turn.

Directional Light

Directional lights are peculiar in that you specify the direction of the light, but not its source position. The directional light emits parallel light rays from a distant source that you need not specify. This vector is referred to as a *normal vector* because it points in the desired direction and has a length of 1.0. You can create a normal vector using the `Normalize` function on an existing vector. `Normalize` results in a vector pointing in the same direction with a length of 1.0. Why must the length of a normalized vector be 1.0? Because when that vector is multiplied by another vector or matrix, its value determines direction rather than length (by multiplying those values by 1.0). The following example code creates a directional light above the target (direction vector) pointing downward.

```
//NOTE: This is not a complete program
D3DLIGHT9 light;
light.Type = D3DLIGHT_DIRECTIONAL;
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
light.Diffuse.a = 1.0f;
light.Range = 1000;
D3DXVECTOR3 direction(0.0f,2.0f,0.0f);
D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &direction );
```

Spot Light

Spot lights are quite different from directional lights because they are limited in range and focus on a specific target. You can specify the inner and outer cone of a spot light, where the inner cone is the bright beam of light and the outer cone is the light spillage or *aura* around the beam, much like a flashlight or street lamp. This is the most complicated light to set up. In the following example, `Theta` is the spread of the inner cone, and `Phi` is the spread of the outer cone, while `Falloff` is the decrease in illumination at the outer edge.

```
D3DLIGHT9 light;
light.Type = D3DLIGHT_SPOT;
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
light.Diffuse.a = 1.0f;
light.Position = position;
light.Direction = direction;
light.Range = 200;
light.Theta = 0.5f;
light.Phi = 1.0f;
light.Falloff = 1.0f;
light.Attenuation0 = 1.0f;
```

Point Light

A point light is a single light source that emits in all directions like a lightbulb. You set the position, color, and attenuation, which is the amount of a decrease in light over distance. The range is the maximum distance that the light will illuminate your 3D objects.

```
D3DLIGHT9 light;
light.Type = D3DLIGHT_POINT;
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
light.Position = position;
light.Attenuation0 = 0.1f;
light.Range = range;
```

Creating Lights

After configuring a light, you must assign it to one of the eight hardware lights and then enable it.

```
device->SetLight( 0, &light );
device->LightEnable( 0, TRUE );
```

Now let's package up this light functionality into a reusable class so it will be easy to create one of the three light types, and then add the class to our Advanced2D engine. First up is the header:

```
#pragma once
#include "Advanced2d.h"
namespace Advanced2D {

class Light
{
private:
    D3DLIGHT9 p_light;
    D3DLIGHTTYPE p_type;
    int p_lightNum;
public:
    Light(int lightNum, D3DLIGHTTYPE type, D3DXVECTOR3 position, D3DXVECTOR3
direction, double range);
    ~Light(void);
    void setX(double value) { p_light.Position.x = (float)value; }
    double getX() { return p_light.Position.x; }
    void setY(double value) { p_light.Position.y = (float)value; }
    double getY() { return p_light.Position.y; }
    void setZ(double value) { p_light.Position.z = (float)value; }
    double getZ() { return p_light.Position.z; }
    D3DLIGHTTYPE getType() { return p_type; }
    void setColor(D3DCOLORVALUE color) { p_light.Diffuse = color; };
    D3DCOLORVALUE getColor() { return p_light.Diffuse; }
    void setDirection(D3DXVECTOR3 direction) { this->p_light.Direction = direction; }
    void setDirection(double x,double y,double z) {
        setDirection(D3DXVECTOR3((float)x,(float)y,(float)z));
    }
    D3DXVECTOR3 getDirection() { return this->p_light.Direction; }
    void setPosition(D3DXVECTOR3 pos) { p_light.Position = pos; }
    void setPosition(double x,double y,double z) {
        setPosition(D3DXVECTOR3((float)x,(float)y,(float)z));
    }
    D3DXVECTOR3 getPosition() { return p_light.Position; }
    void Update();
    void Show();
    void Hide();
};
}; //namespace
```

Now for the `Light` class implementation, and then we'll test it.

```
#include "Advanced2D.h"
namespace Advanced2D {

Light::Light(int lightNum, D3DLIGHTTYPE type, D3DXVECTOR3 position,
D3DXVECTOR3 direction, double range)
{
    this->p_lightNum = lightNum;
    ZeroMemory( &p_light, sizeof(D3DLIGHT9) );

    p_light.Diffuse.r = p_light.Ambient.r = 1.0f;
    p_light.Diffuse.g = p_light.Ambient.g = 1.0f;
    p_light.Diffuse.b = p_light.Ambient.b = 1.0f;
    p_light.Diffuse.a = p_light.Ambient.a = 1.0f;

    switch(type)
    {
        case D3DLIGHT_POINT:
            p_light.Type = D3DLIGHT_POINT;
            p_light.Position = position;
            p_light.Attenuation0 = 0.1f;
            p_light.Range = (float)range;
            break;

        case D3DLIGHT_SPOT:
            p_light.Type = D3DLIGHT_SPOT;
            p_light.Position = position;
            p_light.Direction = direction;
            p_light.Range = (float)range;
            p_light.Theta = 0.5f;
            p_light.Phi = 1.0f;
            p_light.Falloff = 1.0f;
            p_light.Attenuation0 = 1.0f;
            break;

    case D3DLIGHT_DIRECTIONAL:
    default:
        p_light.Type = D3DLIGHT_DIRECTIONAL;
        p_light.Range = (float)range;
        //create a normalized direction
        D3DXVec3Normalize( (D3DXVECTOR3*)&p_light.Direction, &direction );
       break;
    }

    //enable the light
    Show();
    Update();
}

Light::~Light(void) { }

void Light::Update()
{
    g_engine->getDevice()->SetLight(p_lightNum, &p_light);
}

void Light::Show()
{
    g_engine->getDevice()->LightEnable(p_lightNum,TRUE);
}

void Light::Hide()
{
    g_engine->getDevice()->LightEnable(p_lightNum,FALSE);
}
}; //namespace
```

Now let's put this class to the test. By adding the Light.h and Light.cpp files to the game engine, it will be possible to load up a mesh and apply direct lighting to it in a scene with only a handful of code. Let's give it a try. The following code is found in the LightingDemo project on the CD-ROM (and it assumes that you're using the version of the `Engine` provided on the CD-ROM with the `Light` class already set up).

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;
//macro to read the keyboard asynchronously
#define KEY_DOWN(vk) ((GetAsyncKeyState(vk) & 0x8000)?1:0)
//game objects
Camera *camera;
Light *light;
Mesh *mesh;

bool game_preload()
{
    g_engine->setAppTitle("LIGHTING DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(1024);
    g_engine->setScreenHeight(768);
    g_engine->setColorDepth(32);
    return true;
}

bool game_init(HWND)
{
    //set the camera and perspective
    camera = new Camera();
    camera->setPosition(0.0f, 2.0f, 40.0f);
    camera->setTarget(0.0f, 0.0f, 0.0f);
    camera->Update();

    //load the mesh
    mesh = new Mesh();
    mesh->Load("cytovirus.x");
    mesh->SetScale(0.1f,0.1f,0.1f);

    //create a directional light
    D3DXVECTOR3 pos(0.0f,0.0f,0.0f);
    D3DXVECTOR3 dir(0.0f,-1.0f,0.0f);
    light = new Light(0, D3DLIGHT_DIRECTIONAL, pos, dir, 100);

    //set a low ambient level
    g_engine->SetAmbient(D3DCOLOR_XRGB(20,20,20));
    return true;
}

void game_update()
{
    //rotate the cytovirus mesh
    mesh->Rotate(-0.1f,0.0f,0.05f);
    //exit when escape key is pressed
    if (KEY_DOWN(VK_ESCAPE)) g_engine->Close();
}

void game_end()
{
    delete camera;
    delete light;
    delete mesh;
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));
    g_engine->SetIdentity();

    mesh->Transform();
    mesh->Draw();
}
```

Figure 2.4 shows the output from the LightingDemo program, as shown in the listing with a directional light.

By changing the key code in the program that creates the light, we can test a spot light fairly easily (demonstrating the usefulness of the `Light` class). This code change will convert the directional light into a spot light, resulting in the output shown in Figure 2.5.

```
D3DXVECTOR3 pos(-10.0f,-20.0f,0.0f);
D3DXVECTOR3 dir(0.0f,2.0f,0.0f);
light = new Light(0, D3DLIGHT_SPOT, pos, dir, 1000);
```

Now let's change the code to produce a point light. The following code produces the output shown in Figure 2.6.

```
D3DXVECTOR3 pos(0.0f,-22.0f,0.0f);
D3DXVECTOR3 dir(0.0f,0.0f,0.0f);
light = new Light(0, D3DLIGHT_POINT, pos, dir, 20);
```

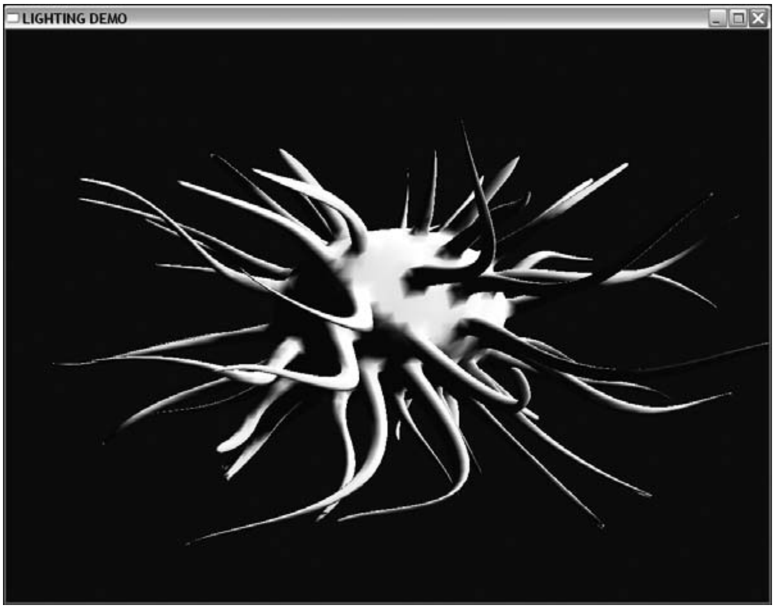


Figure 2.4
The LightingDemo program set to use a directional light.

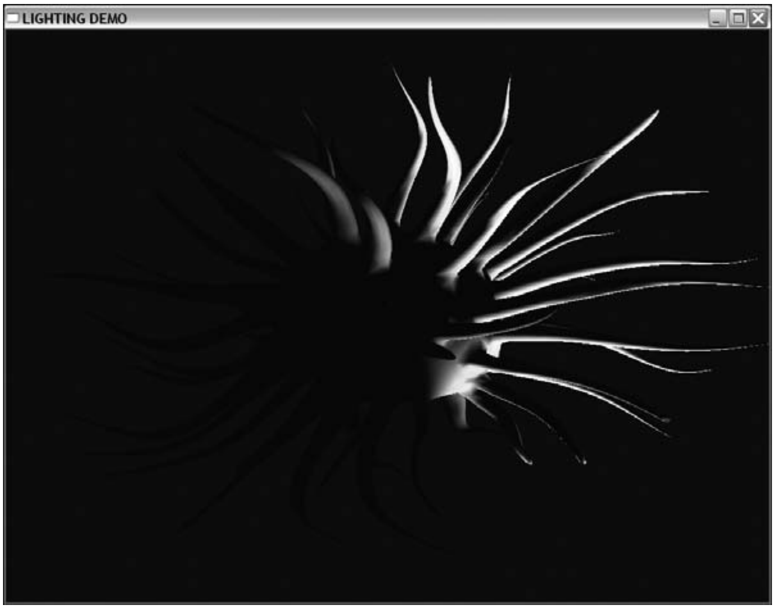


Figure 2.5
The LightingDemo program set to use a spot light.

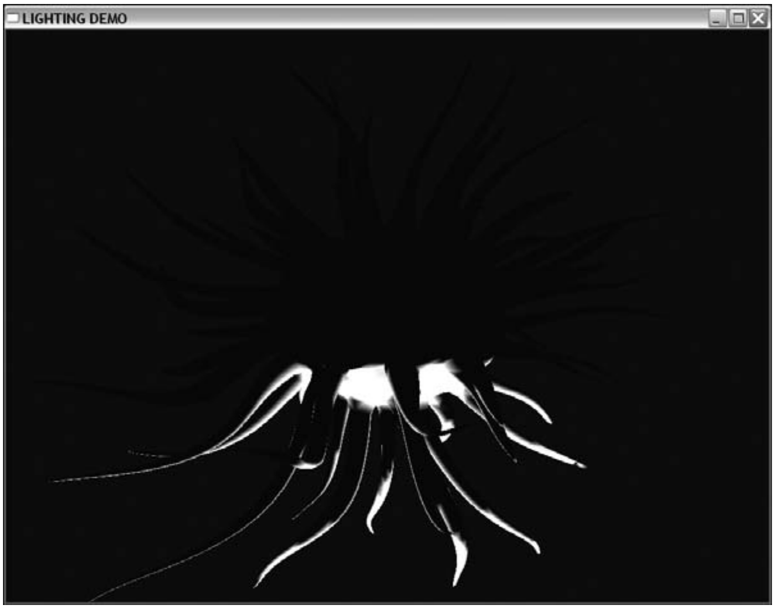


Figure 2.6
The LightingDemo program set to use a point light.

Although you can change the position, direction, range, and other properties for a light source after it has been created, you cannot change the type of light source. Of course, you could just delete the `Light` object and create a new light at any time, or you could selectively turn off and on some lights based on the situation in your game.

Well, I think that's enough of a feature set for the engine's 3D rendering capabilities! We do not have any advanced rendering ability in this engine, but then 3D is not our focus so what we can do here is more than enough for a 2D renderer—which is the focus of the next chapter.

CHAPTER 3

2D Rendering

We have made good progress on the Advanced2D engine in short time, and we have a few more key features to add before we'll have a viable engine for building the games we need to explore later in this book. It goes without saying that we need 2D rendering support! Even the most advanced 3D engine today needs to support 2D rendering for things such as text output and a graphical user interface.

There are two ways to render 2D objects in Direct3D. First, you can create a quad (or rectangle) comprised of two triangles with a texture representing the 2D image you wish to draw. This technique works with and even supports transparency, responds to lighting, and can be moved in the Z direction. The second method available in Direct3D for rendering 2D objects is with sprites—and this is the method we will focus on in this chapter. A *sprite* is a 2D representation of a game entity that usually must interact with the player in some way. A tree or rock might be rendered in 2D and interact with the player by simply getting in the way, stopping the player by way of collision physics.

We must also deal with game characters that directly or indirectly interact with the player's character (which might be a spaceship, an Italian plumber, or a spiky-haired hedgehog). The types of sprites that interact with the player might be an enemy ship or a laser in a space combat game—I could go on and on with examples.

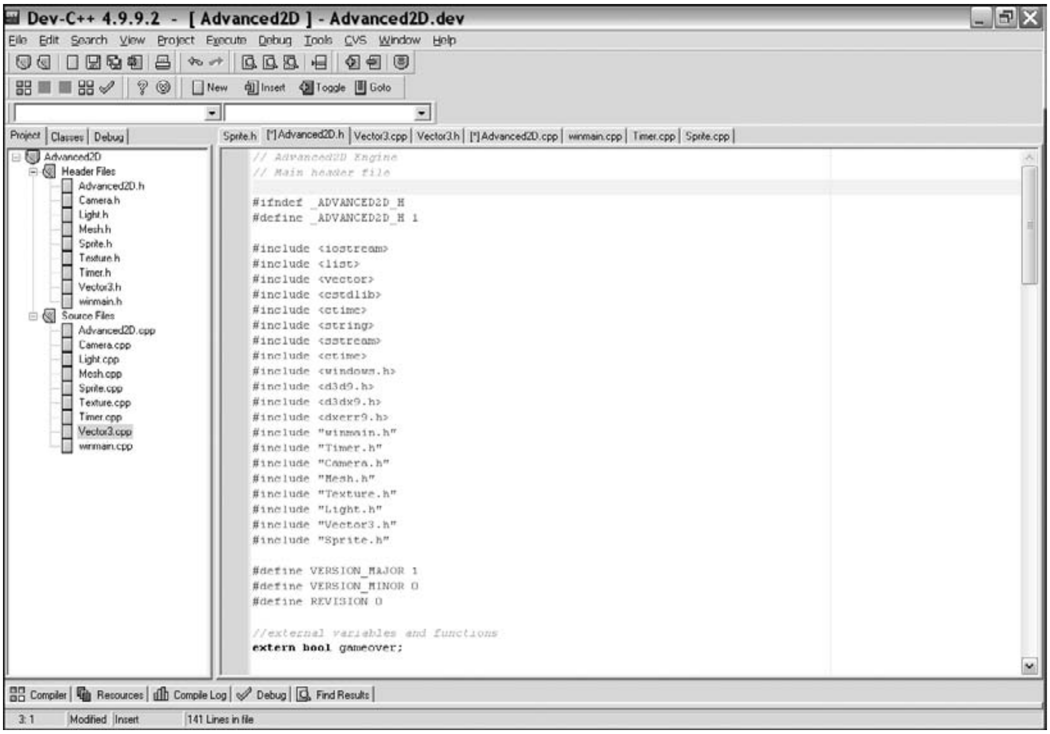
Advice

The Advanced2D engine will evolve from one chapter to the next as it gains new classes and capabilities. This is normal for software development—nothing is set in stone yet! Every aspect of the engine will change over time to accommodate new features and needs that are identified through the development of demo programs. By presenting the engine in lock-step fashion as you see here, you are able to watch the engine develop from its early stages into the fully featured version used in the final chapters.

Basic 2D Rendering

We will get to sprite rendering soon, but first we need to add basic 2D rendering support to the Advanced2D engine. We'll then use this basic support to develop more complex rendering techniques (such as a sprite system) later in this chapter.

Open the Advanced2D engine project from the CD-ROM in the folder for this chapter. The engine project is called Engine.dev for Dev-C++ or Engine.sln for Visual C++. We aren't going to update or maintain the engine project here; I'm just going to show you the new features of the engine in this chapter (and likewise in the chapters to come). Open the Advanced2D.h file, as shown in Figure 3.1.


Dev-C++ 4.9.9.2 - [Advanced2D] - Advanced2D.dev
File Edit Search View Project Execute Debug Tools CVS Window Help
New Insert Toggle Goto
Project Classes Debug
Sprite.h [*]Advanced2D.h Vector3.cpp Vector3.h [*]Advanced2D.cpp winmain.cpp Timer.cpp Sprite.cpp
Advanced2D
Header Files
Advanced2D.h
Camera.h
Light.h
Mesh.h
Sprite.h
Texture.h
Timer.h
Vector3.h
winmain.h
Source Files
Advanced2D.cpp
Camera.cpp
Light.cpp
Mesh.cpp
Sprite.cpp
Texture.cpp
Timer.cpp
Vector3.cpp
winmain.cpp
Compiler Resources Compile Log Debug Find Results
3: 1 Modified Insert 141 Lines in file

```
// Advanced2D Engine
// Main header file

#ifndef _ADVANCED2D_H
#define _ADVANCED2D_H 1

#include <iostream>
#include <list>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <string>
#include <sstream>
#include <ctime>
#include <windows.h>
#include <d3d9.h>
#include <d3dx9.h>
#include <dxerr9.h>
#include "winmain.h"
#include "Timer.h"
#include "Camera.h"
#include "Mesh.h"
#include "Texture.h"
#include "Light.h"
#include "Vector3.h"
#include "Sprite.h"

#define VERSION_MAJOR 1
#define VERSION_MINOR 0
#define REVISION 0

//external variables and functions
extern bool gameover;
```

Figure 3.1
The Advanced2D engine project in Dev-C++.

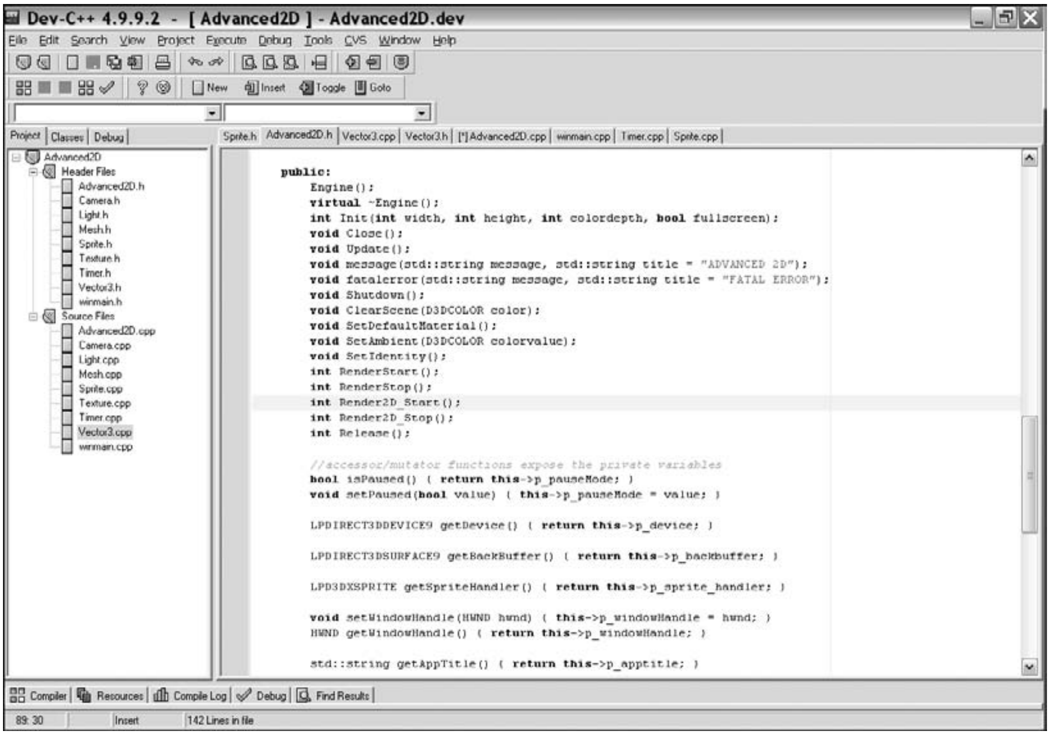
Now, scroll down a little ways to where the public methods are declared in the `Advanced2D::Engine` class. Note the two new lines highlighted in bold. These are two new methods added to the class. Although they are declared as public, they will not need to be called from outside the class.

```
public:
             Engine();
     virtual ~Engine();
     int Init(int width, int height, int colordepth, bool fullscreen);
     void Close();
     void Update();
     void message(std::string message, std::string title = "ADVANCED 2D");
     void fatalerror(std::string message, std::string title = "FATAL ERROR");
     void Shutdown();
     void ClearScene(D3DCOLOR color);
     void SetDefaultMaterial();
     void SetAmbient(D3DCOLOR colorvalue);
     int RenderStart();
     int RenderStop();
     int Render2D_Start();
     int Render2D_Stop();
      int Release();
```

All the existing code was part of the project from the previous chapter, and we've just added the two new methods to the class here. Figure 3.2 shows the code in Dev-C++.

Now let's review the code for these method definitions. Switch to the Advanced2D.cpp file and scroll down to the `Engine::RenderStart` and `Engine::RenderStop` definitions where the new methods have been added. I'm showing you the code here only for reference—there's no need to type it into the engine project, which has already been modified.

```
int Engine::Render2D_Start()
{
     if (p_sprite_handler->Begin(D3DXSPRITE_ALPHABLEND) != D3D_OK)
        return 0;
     else
        return 1;
}
int Engine::Render2D_Stop()
{
     p_sprite_handler->End();
     return 1;
}
```


Dev-C++ 4.9.9.2 - [Advanced2D] - Advanced2D.dev
File Edit Search View Project Execute Debug Tools CVS Window Help
New Insert Toggle Goto
Project Classes Debug
Sprite.h Advanced2D.h Vector3.cpp Vector3.h [*] Advanced2D.cpp winmain.cpp Timer.cpp Sprite.cpp
Advanced2D
Header Files
Advanced2D.h
Camera.h
Light.h
Mesh.h
Sprite.h
Texture.h
Timer.h
Vector3.h
winmain.h
Source Files
Advanced2D.cpp
Camera.cpp
Light.cpp
Mesh.cpp
Sprite.cpp
Texture.cpp
Timer.cpp
Vector3.cpp
winmain.cpp
Compiler Resources Compile Log Debug Find Results
89: 30 Insert 142 Lines in file

```
public:
    Engine();
    virtual ~Engine();
    int Init(int width, int height, int colordepth, bool fullscreen);
    void Close();
    void Update();
    void message(std::string message, std::string title = "ADVANCED 2D");
    void fatalerror(std::string message, std::string title = "FATAL ERROR");
    void Shutdown();
    void ClearScene(D3DCOLOR color);
    void SetDefaultMaterial();
    void SetAmbient(D3DCOLOR colorvalue);
    void SetIdentity();
    int RenderStart();
    int RenderStop();
    int Render2D_Start();
    int Render2D_Stop();
    int Release();

    //accessor/mutator functions expose the private variables
    bool isPaused() { return this->p_pauseMode; }
    void setPaused(bool value) { this->p_pauseMode = value; }

    LPDIRECT3DDEVICE9 getDevice() { return this->p_device; }

    LPDIRECT3DSURFACE9 getBackBuffer() { return this->p_backbuffer; }

    LPD3DXSPRITE getSpriteHandler() { return this->p_sprite_handler; }

    void setWindowHandle(HWND hwnd) { this->p_windowHandle = hwnd; }
    HWND getWindowHandle() { return this->p_windowHandle; }

    std::string getAppTitle() { return this->p_apptitle; }
```

Figure 3.2
Adding new functionality to the Advanced2D engine.

The `sprite_handler` variable was defined in the Advanced2D.h file like so:

```
LPD3DXSPRITE p_sprite_handler;
```

This sprite handler object is created in `Engine::Init` (located in Advanced2D .cpp):

```
//initialize 2D renderer
HRESULT result = D3DXCreateSprite(this->p_device, &this->p_sprite_handler);
if (result != D3D_OK) return 0;
```

After initializing the sprite handler as part of the engine's startup process, we can then use this `D3DXSprite` object to render 2D graphics. The rendering step for 2D must be done within the 3D rendering process, as you'll see in a moment.

The `Engine::Update()` function needs some modification to enable 2D rendering within the game loop. The new function calls are highlighted in bold text.

```
void Engine::Update()
{
      static Timer timedUpdate;

      //calculate core framerate
      p_frameCount_core++;
      if (p_coreTimer.stopwatch(999)) {
          p_frameRate_core = p_frameCount_core;
          p_frameCount_core = 0;
      }

      //fast update with no timing
      game_update();

      //update with 60fps timing
      if (!timedUpdate.stopwatch(14)) {
          if (!this->getMaximizeProcessor()) {
              Sleep(1);
          }
      }
      else {
          //calculate real framerate
          p_frameCount_real++;
          if (p_realTimer.stopwatch(999)) {
              p_frameRate_real = p_frameCount_real;
              p_frameCount_real = 0;
          }

          //begin rendering
          this->RenderStart();

          //let game do it's own 3D
          game_render3d();
          //2D rendering
          Render2D_Start();
          game_render2d();
          Render2D_Stop();

          //done rendering
          this->RenderStop();
      }
}
```

Raising Happy Sprites

It's one thing to know how to render a sprite—even a complex sprite with transparency and animation—but it's quite another matter to do something *useful* with it. Some software engineers cannot see beyond the specifications, are unable to design creative gameplay, and, as a result, focus their time on mechanics of the game. What we're doing now is managing the *logistics* of 2D games by building this game engine and providing support facilities within the engine to simplify the *engineering* side of 2D game development. There are literally hundreds of game engines at repositories such as SourceForge, but they are mostly the result of failed game projects. When you design an engine from the outset with reuse and multi-genre support in mind, then you will more likely finish the game you have planned, as well as end up with a useful engine out of the deal.

We need to build a sprite engine that is powerful enough to support a myriad of game genres—from fixed-screen arcade-style games, to scrolling shooters, to board games, and so on. In other words, our 2D rendering system must be robust, fully featured, and versatile. That calls for some iterative programming!

Advice

Iterative programming is a development methodology in which a system (such as a game) is built in small stages and is more like the growth of a life form than the construction of a building (a common analogy in software engineering theory). The term "iterative" comes from the edit-compile-test process that is repeated over and over until the code functions as desired.

This contrasts sharply with other methodologies that call for extensive preparation and design of every facet of a system in advance. In my experience, a large software project is more comparable to a life form than to an inanimate object, and iterative development seems to produce better results in many cases.

The interesting thing about iterative development is that it is often adopted when a project runs over budget or misses its completion dates, at which point the team will resort to iterative programming to finish the project. Why spend most of the time with a faulty methodology when the most effective one is only reserved to complete a failed development plan? Game developers have been writing iterative code for decades!

Now that we have a 2D rendering subsystem available, we need to code up some functionality that actually does something useful with sprites. You could just load up an image and draw it with `D3DXSprite`. But to what end? We need the ability to manipulate *game entities* that will move on the screen in interesting ways and interact with each other, not to mention animate themselves. (We'll explore entity management in more detail in Chapter 7, "Entity Management.")

Creating Vectors

The most useful concept we will need in order to manipulate sprites effectively is a vector. A *vector* is a mathematical construct that represents two things at once—a point as well as a direction. A vector is not merely a point, nor is it merely a direction; otherwise, we would use one term or the other to describe it. However, we *can* use a vector to represent simple points, or positions, for game entities such as sprites and meshes.

A vector with its own built-in functionality will be incredibly helpful to a `Sprite` class. We will be able to give a sprite properties, such as position, direction, and velocity, as well as calculate the trajectory to a target, the normal angle of a polygon, and other helpful functions (some of which we may not need but which are available nonetheless). We will use the `Vector3` class (listed in the following section) to do the "heavy lifting" for the upcoming `Sprite` class.

Advice

I found the following URL to be a helpful reference for the math behind computer graphics concepts such as points, lines, vectors, and matrices: http://programmedlessons.org/VectorLessons/vectorIndex.html.

Vector3.h

Here is the `Vector3` class definition as it appears in the Vector3.h file:

```
#include "Advanced2D.h"
#pragma once
namespace Advanced2D {
     class Vector3 {
     private:
          double x, y, z;

     public:
          Vector3();
          Vector3(const Vector3& v);
          Vector3(double x, double y, double z);
          Vector3(int x, int y, int z);
          void Set(double x1,double y1,double z1);
          void Set(const Vector3& v);
          double getX() { return x; }
          void setX(double v) { x = v; }
          double getY() { return y; }
```

```
            void setY(double v) { y = v; }
            double getZ() { return z; }
            void setZ(double v) { z = v; }
            void Move( double mx,double my,double mz);
            void operator+=(const Vector3& v);
            void operator-=(const Vector3& v);
            void operator*=(const Vector3& v);
            void operator/=(const Vector3& v);
            bool operator==( const Vector3& v ) const;
            bool operator!=( const Vector3& p ) const;
            Vector3& operator=( const Vector3& v);
            double Distance( const Vector3& v );
            double Length();
            double DotProduct( const Vector3& v );
            Vector3 CrossProduct( const Vector3& v );
            Vector3 Normal();
        }; //class
}; //namespace
```

Vector3.cpp

Here is the Vector3 class implementation. Most of the code in Vector3 will not be used immediately, but I want to provide the complete class right now rather than modifying it later with new functionality, even if some of its methods are unknown to you.

```
#include "Advanced2D.h"
namespace Advanced2D {
    Vector3::Vector3()
        {
            x = y = z = 0;
        }

        Vector3::Vector3( const Vector3& v )
        {
            *this = v;
        }

        Vector3::Vector3( double x, double y, double z )
        {
            Set( x, y, z );
        }
```

```
Vector3::Vector3( int x, int y, int z)
{
    Set((double)x,(double)y,(double)z);
}

void Vector3::Set( double x1,double y1,double z1 )
{
    x=x1; y=y1; z=z1;
}

void Vector3::Set( const Vector3& v)
{
    x=v.x; y=v.y; z=v.z;
}

void Vector3::Move( double mx,double my,double mz)
{
    x+=mx; y+=my; z+=mz;
}

void Vector3::operator+=(const Vector3& v)
{
    x+=v.x; y+=v.y; z+=v.z;
}

void Vector3::operator-=(const Vector3& v)
{
    x-=v.x; y-=v.y; z-=v.z;
}

void Vector3::operator*=(const Vector3& v)
{
    x*=v.x; y*=v.y; z*=v.z;
}

void Vector3::operator/=(const Vector3& v)
{
    x/=v.x; y/=v.y; z/=v.z;
}

//equality operator comparison includes double rounding
bool Vector3::operator==( const Vector3& v ) const
{
    return (
        (((v.x - 0.0001f) < x) && (x < (v.x + 0.0001f))) &&
        (((v.y - 0.0001f) < y) && (y < (v.y + 0.0001f))) &&
        (((v.z - 0.0001f) < z) && (z < (v.z + 0.0001f))) );
}

//inequality operator
bool Vector3::operator!=( const Vector3& p ) const
{
    return (!(*this == p));
}

//assign operator
Vector3& Vector3::operator=( const Vector3& v)
{
    Set(v);
    return *this;
}

//distance only coded for 2D
double Vector3::Distance( const Vector3& v )
{
    return sqrt((v.x-x)*(v.x-x) + (v.y-y)*(v.y-y));
}

//Vector3 length is distance from the origin
double Vector3::Length()
{
    return sqrt(x*x + y*y + z*z);
}

//dot/scalar product: difference between two directions
double Vector3::DotProduct( const Vector3& v )
{
    return (x*v.x + y*v.y + z*v.z);
}

//cross/Vector product is used to calculate the normal
Vector3 Vector3::CrossProduct( const Vector3& v )
{
    double nx = (y*v.z)-(z*v.y);
    double ny = (z*v.y)-(x*v.z);
```

```
        double nz = (x*v.y)-(y*v.x);
        return Vector3(nx,ny,nz);
    }

    //calculate normal angle of the Vector
    Vector3 Vector3::Normal()
    {
        double length;
        if (Length() == 0)
            length = 0;
        else
            length = 1 / Length();
        double nx = x*length;
        double ny = y*length;
        double nz = z*length;
        return Vector3(nx,ny,nz);
    }
}
```

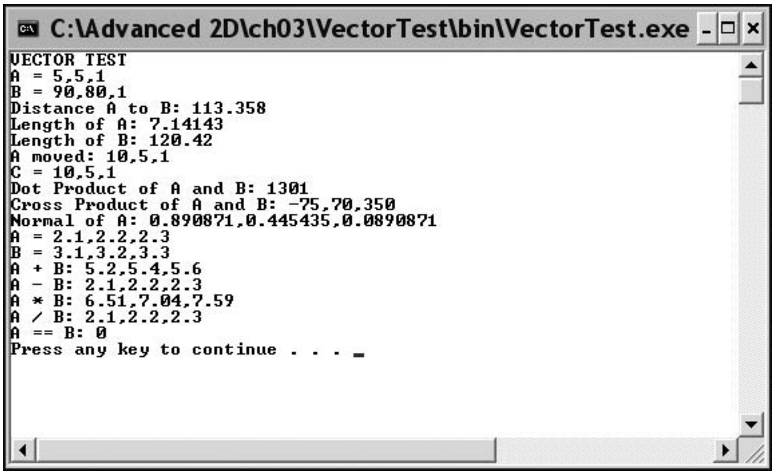
Testing Vector3

Because the `Vector3` class is so complicated (and important!), I want to test its functionality before plugging it into the upcoming `Sprite` class. On the CD is a project called VectorTest, which is a Win32 Console project that does not need the whole game engine, just the `Vector3` class. The Vector3.h and Vector3.cpp files had to be modified a bit to allow the class to compile on its own. (The Advanced2D.h `include` and `namespace` lines were commented out.) As a console program, we need only a main function and will use `iostream` for output. Among other things, the VectorTest program demonstrates how you can manipulate a vector using operators such as +, —, +=, and —=. See Figure 3.3.

```
#include <iostream>
#include "Vector3.h"
using namespace std;

int main(int argc, char *argv[])
{
    cout << "VECTOR TEST" << endl;

    Vector3 A(5,5,1);
    cout << "A = " << A.getX() << ","
        << A.getY() << "," << A.getZ() << endl;
```

```
Vector3 B(90,80,1);
cout << "B = " << B.getX() << ","
    << B.getY() << "," << B.getZ() << endl;

cout << "Distance A to B: "
    << A.Distance( B ) << endl;

cout << "Length of A: " << A.Length() << endl;
cout << "Length of B: " << B.Length() << endl;

A.Move(5, 0, 0);
cout << "A moved: " << A.getX() << ","
    << A.getY() << "," << A.getZ() << endl;

Vector3 C = A;
cout << "C = " << C.getX() << "," << C.getY()
    << "," << C.getZ() << endl;

cout << "Dot Product of A and B: "
    << A.DotProduct(B) << endl;


Vector3 D = A.CrossProduct(B);
cout << "Cross Product of A and B: " << D.getX()
    << "," << D.getY() << "," << D.getZ() << endl;
D = A.Normal();
cout << "Normal of A: " << D.getX() << ","
    << D.getY() << "," << D.getZ() << endl;

A.Set(2.1,2.2,2.3);
B.Set(3.1,3.2,3.3);
cout << "A = " << A.getX() << "," << A.getY()
    << "," << A.getZ() << endl;
cout << "B = " << B.getX() << "," << B.getY()
    << "," << B.getZ() << endl;

A += B;
cout << "A + B: " << A.getX() << "," << A.getY()
    << "," << A.getZ() << endl;

A -= B;
cout << "A - B: " << A.getX() << "," << A.getY()
```

```
        << "," << A.getZ() << endl;

    A *= B;
    cout << "A * B: " << A.getX() << "," << A.getY()
        << "," << A.getZ() << endl;

    A /= B;
    cout << "A / B: " << A.getX() << "," << A.getY()
        << "," << A.getZ() << endl;

    cout << "A == B: " << (A == B) << endl;

    system("pause");
    return 0;
}
```



```
VECTOR TEST
A = 5,5,1
B = 90,80,1
Distance A to B: 113.358
Length of A: 7.14143
Length of B: 120.42
A moved: 10,5,1
C = 10,5,1
Dot Product of A and B: 1301
Cross Product of A and B: -75,70,350
Normal of A: 0.890871,0.445435,0.0890871
A = 2.1,2.2,2.3
B = 3.1,3.2,3.3
A + B: 5.2,5.4,5.6
A - B: 2.1,2.2,2.3
A * B: 6.51,7.04,7.59
A / B: 2.1,2.2,2.3
A == B: 0
Press any key to continue . . . _
```

Figure 3.3
The VectorTest program tests the features of the `Vector3` class.

Creating a Reusable Sprite Class

The `Vector3` class will greatly simplify the code in the `Sprite` class, which otherwise would have to calculate things such as velocity on its own. In some cases, we'll use `Vector3` just for a simple x,y position. This might seem wasteful when the `Vector3` class is so powerful, but it keeps our code uniform and predictable. I'm a firm believer in writing *maintainable* code rather than *slightly faster* code.

What do we want to do with sprites? When it comes right down to it, the answer is *almost everything!* Sprites are at the very core of 2D games, which is, of course, our focus here. We need to load and draw simple sprites (with no animation, just a single image), as well as the more complex animated sprites (with frames of animation). There is a need for both static and animated sprites in every game. In fact, most game objects are animated, which begs the questions how do we create an animation, and how do we animate a sprite? We'll get to the first question later, when we create some complete game projects. We'll get to the second question in just a moment.

Sprite.h

To answer the second question requires a bit of work. Let's take a look at the `Sprite` class header first. This class is feature rich, meaning that it is loaded with features we haven't even gone over yet and will not go over until we use some of these features in future chapters (for instance, collision detection, which is not covered until Chapter 9, "Physics").

```
#include "Advanced2D.h"
#pragma once
namespace Advanced2D {
    enum CollisionType {
        COLLISION_NONE = 0,
        COLLISION_RECT = 1,
        COLLISION_DIST = 2
    };

    class Sprite {
    private:
        bool visible;
        bool alive;
    int lifetimeLength;
        Timer lifetimeTimer;
        int objecttype;

        Vector3 position;
        Vector3 velocity;
        bool imageLoaded;
        int state;
        int direction;
     protected:
         Texture *image;
         int width,height;
         int animcolumns;
         int framestart,frametimer;
         int movestart, movetimer;
         bool collidable;
         enum CollisionType collisionMethod;
         int curframe,totalframes,animdir;
         double faceAngle, moveAngle;
         int animstartx, animstarty;
         double rotation, scaling;
         D3DXMATRIX matRotate;
         D3DXMATRIX matScale;
         void transform();
         D3DCOLOR color;

     public:
         //screen position
         Vector3 getPosition() { return position; }
         void setPosition(Vector3 position) { this->position = position; }
         void setPosition(double x, double y) { position.Set(x,y,0); }
         double getX() { return position.getX(); }
         double getY() { return position.getY(); }
         void setX(double x) { position.setX(x); }
         void setY(double y) { position.setY(y); }

         //movement velocity
         Vector3 getVelocity() { return velocity; }
         void setVelocity(Vector3 v) { this->velocity = v; }
         void setVelocity(double x, double y) { velocity.setX(x); velocity.setY(y); }

         //image size
         void setSize(int width, int height) { this->width = width; this->height =
 height; }
         int getWidth() { return this->width; }
         void setWidth(int value) { this->width = value; }
         int getHeight() { return this->height; }
         void setHeight(int value) { this->height = value; }

         bool getVisible() { return visible; }
         void setVisible(bool value) { visible = value; }
         bool getAlive() { return alive; }
         void setAlive(bool value) { alive = value; }

         int getState() { return state; }
         void setState(int value) { state = value; }

         int getDirection() { return direction; }
         void setDirection(int value) { direction = value; }

         int getColumns() { return animcolumns; }
         void setColumns(int value) { animcolumns = value; }

         int getFrameTimer() { return frametimer; }
         void setFrameTimer(int value) { frametimer = value; }

         int getCurrentFrame() { return curframe; }
         void setCurrentFrame(int value) { curframe = value; }

         int getTotalFrames() { return totalframes; }
         void setTotalFrames(int value) { totalframes = value; }

         int getAnimationDirection() { return animdir; }
         void setAnimationDirection(int value) { animdir = value; }

         double getRotation() { return rotation; }
         void setRotation(double value) { rotation = value; }
         double getScale() { return scaling; }
         void setScale(double value) { scaling = value; }
         void setColor(D3DCOLOR col) { color = col; }

         int getMoveTimer() { return movetimer; }
         void setMoveTimer(int value) { movetimer = value; }

         bool isCollidable() { return collidable; }
         void setCollidable(bool value) { collidable = value; }
         CollisionType getCollisionMethod() { return collisionMethod; }
         void setCollisionMethod(CollisionType type) { collisionMethod = type; }

     public:
         Sprite();
         virtual ~Sprite();
         bool loadImage(std::string filename, D3DCOLOR transcolor = D3DCOLOR_
XRGB(255,0,255));
```

```
        void setImage(Texture *);
        void move();
        void animate();
        void draw();

    }; //class
}; //namespace
```

Sprite.cpp

That was a large header file, I'll admit, but it was jam-packed with features that we'll need later, and—as I mentioned back at the `Vector3` listing—I prefer to give you the complete listing for a reusable class rather than modifying it. Here is the `Sprite` class implementation:

```
#include "Advanced2D.h"
namespace Advanced2D {
    Sprite::Sprite()
    {
        this->image = NULL;
        this->imageLoaded = false;
        this->setPosition(0.0f,0.0f);
        this->setVelocity(0.0f,0.0f);
        this->state = 0;
        this->direction = 0;
        this->width = 1;
        this->height = 1;
        this->curframe = 0;
        this->totalframes = 1;
        this->animdir = 1;
        this->animcolumns = 1;
        this->framestart = 0;
        this->frametimer = 0;
        this->animcolumns = 1;
        this->animstartx = 0;
        this->animstarty = 0;
        this->faceAngle = 0;
        this->moveAngle = 0;
        this->rotation = 0;
        this->scaling = 1.0f;
        this->color = 0xFFFFFFFF;
        this->movetimer = 16;
        this->movestart = 0;
        this->collidable = true;
        this->collisionMethod = COLLISION_RECT;
    }


    Sprite::~Sprite() {
        if (imageLoaded)
            delete image;
    }


    bool Sprite::loadImage(std::string filename, D3DCOLOR transcolor)
    {
        //if image already exists, free it
        if (imageLoaded && image != NULL) delete image;

        //create texture and load image
        image = new Texture();
        if (image->Load(filename,transcolor))
        {
            this->setSize(image->getWidth(),image->getHeight());
            imageLoaded = true;
            return true;
        }
        else
            return false;
    }


    void Sprite::setImage(Texture *image)
    {
        this->image = image;
        this->setWidth(image->getWidth());
        this->setHeight(image->getHeight());
        this->imageLoaded = false;
    }

    void Sprite::transform()
    {
        D3DXMATRIX mat;
        D3DXVECTOR2 scale((float)scaling,(float)scaling);
        D3DXVECTOR2 center((float)(width*scaling)/2, (float)(height*scaling)/2);
```

```
        D3DXVECTOR2 trans((float)getX(), (float)getY());
        D3DXMatrixTransformation2D(&mat,NULL,0,&scale,&center,(float)rotation,
&trans);
        g_engine->getSpriteHandler()->SetTransform(&mat);
    }

    void Sprite::draw()
    {
        //calculate source frame location
        int fx = (this->curframe % this->animcolumns) * this->width;
        int fy = (this->curframe / this->animcolumns) * this->height;
        RECT srcRect = {fx,fy, fx+this->width, fy+this->height};

        //draw the sprite frame
        this->transform();
        g_engine->getSpriteHandler()->Draw(
            this->image->GetTexture(),&srcRect,NULL,NULL,color);
    }

    void Sprite::move()
    {
        if (movetimer > 0) {

            if (timeGetTime() > (DWORD)(movestart + movetimer)) {
                //reset move timer
                movestart = timeGetTime();

                //move sprite by velocity amount
                this->setX(this->getX() + this->velocity.getX());
                this->setY(this->getY() + this->velocity.getY());
            }
        }
        else {
            //no movement timer--update at cpu clock speed
            this->setX(this->getX() + this->velocity.getX());
            this->setY(this->getY() + this->velocity.getY());
        }
    }

    void Sprite::animate()
    {
        //update frame based on animdir
        if (frametimer > 0) {
            if (timeGetTime() > (DWORD)(framestart + frametimer)) {
                //reset animation timer
                framestart = timeGetTime();
                curframe += animdir;

                //keep frame within bounds
                if (curframe < 0) curframe = totalframes-1;
                if (curframe > totalframes-1) curframe = 0;
            }
        }
        else {
            //no animation timer--update at cpu clock speed
            curframe += animdir;
            if (curframe < 0) curframe = totalframes-1;
            if (curframe > totalframes-1) curframe = 0;
        }
    }
}
```

This is a pretty awesome sprite implementation, if I do say so myself. We have here the ability to render a sprite to any desired scale, at any angle of rotation, with timed animation, using alpha channel transparency and timer-based movement. Whew! Obviously, the `Sprite` class has a lot of functionality that we won't be able to take advantage of for a while, but when we do, it will greatly simplify our future code!

Rendering Sprites with Transparency

We need to put the new `Vector3` and `Sprite` classes through some minor tests to determine how well they function together. We'll explore transparency while testing the `Sprite` class.

`D3DXSprite` doesn't care whether your sprites' source images use a color key or an alpha channel for transparency—it just renders the image as requested. If you have an image with an alpha channel—for instance, a 32-bit targa—then it will be rendered with alpha, including translucent blending with the background if your image has partial alpha ranges defined. But if your image has no alpha because you are using a background color key for transparency—for instance, a 24-bit bitmap—then it will be drawn without the color-keyed pixels.

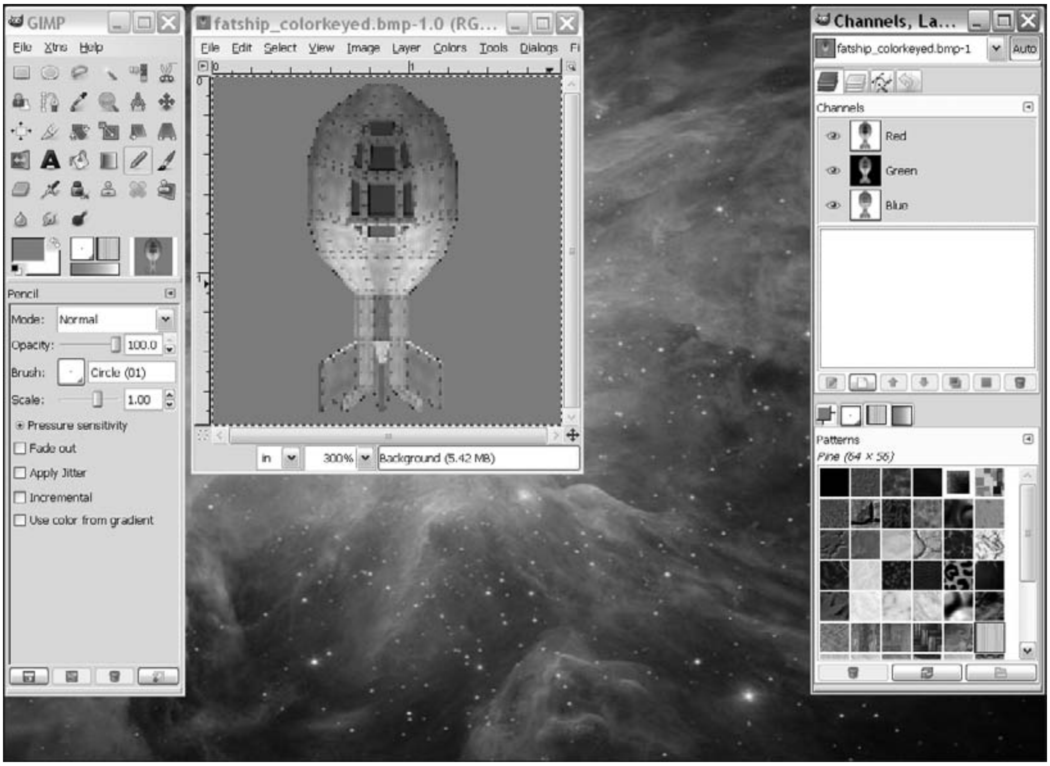


Figure 3.4
This sprite (being edited in GIMP) will be rendered with color-keyed transparency.

Color Key Transparency

Looking at the sprite functionality at a lower level, you can tell the sprite renderer (`D3DXSprite`) what color you want to use for the color key; our `Sprite` class defines magenta (with an RGB of 255, 0, 255) as the default transparent color key. Figure 3.4 shows the sprite we're using in the upcoming demo. This example program is called ColorkeyDemo and is ready to go on the CD-ROM.

Advice

The rocket ship featured here was modeled by UAT graduate Nathan Cox, who—after a stint with EA on *Medal of Honor: Airborne*—now works for Nihilistic Software.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;
Sprite *sprite;
bool game_preload()
{
     g_engine->setAppTitle("SPRITE COLOR KEY DEMO");
     g_engine->setFullscreen(false);
     g_engine->setScreenWidth(640);
     g_engine->setScreenHeight(480);
     g_engine->setColorDepth(32);
     return 1;
}
bool game_init(HWND)
{
     //load sprite
     sprite = new Sprite();
     sprite->loadImage("fatship_colorkeyed.bmp");
     return true;
}

void game_update()
{
     //exit when escape key is pressed
     if (KEY_DOWN(VK_ESCAPE)) g_engine->Close();
}
void game_end()
{
     delete sprite;
}
void game_render3d()
{
     g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));
     g_engine->SetIdentity();
}
void game_render2d()
{
     //calculate center of screen
     int cx = g_engine->getScreenWidth() / 2;
     int cy = g_engine->getScreenHeight() / 2;

     //calculate center of sprite
     int sx = sprite->getWidth() / 2;
     int sy = sprite->getHeight() / 2;

     //draw sprite centered
     sprite->setPosition(cx-sx,cy-sy);
     sprite->draw();
}
```

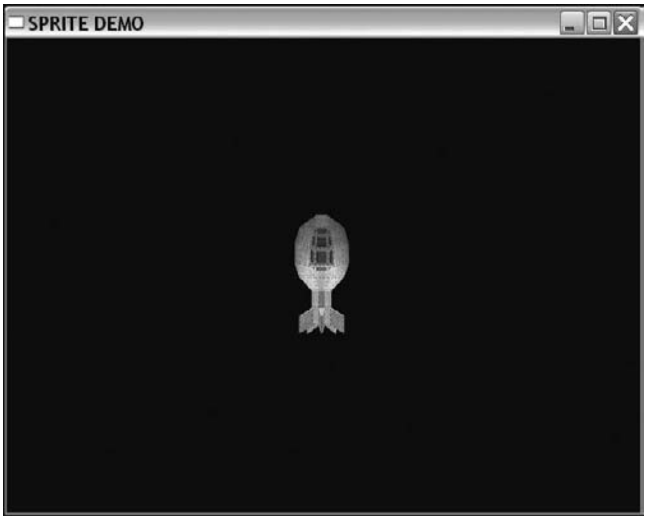


Figure 3.5
The SpriteDemo program demonstrates sprite transparency.

I've called this program ColorkeyDemo, and it is available on the CD. Figure 3.5 shows the output from the code just listed.

Alpha Channel Transparency

You can make an entire game using just color-keyed transparency, but there is a limitation on the quality when using this technique because you must have discrete pixels in such an image unless some sort of render-time blending is performed. Although it is possible to do alpha blending at runtime, it's not a good way to develop a game—it's best to prepare your artwork in advance.

The preferred method for rendering with transparency (especially among artists) is using an alpha channel. One great advantage to alpha-blended images is support for partial transparency—that is, translucent blending. Rather than using a black border around a color-keyed sprite (the old-school way of highlighting a sprite), an artist will blend a border around a sprite's edges using an alpha level for partial translucency, which looks fantastic in comparison! To do that, you must use a file format that supports 32-bit RGBA images. Targa is a good choice, and PNG files work well, too. Let's take a look at the spaceship sprite again—this time with an alpha channel rather than a color-keyed background. Note the checkerboard pattern in the background; this is a common way of

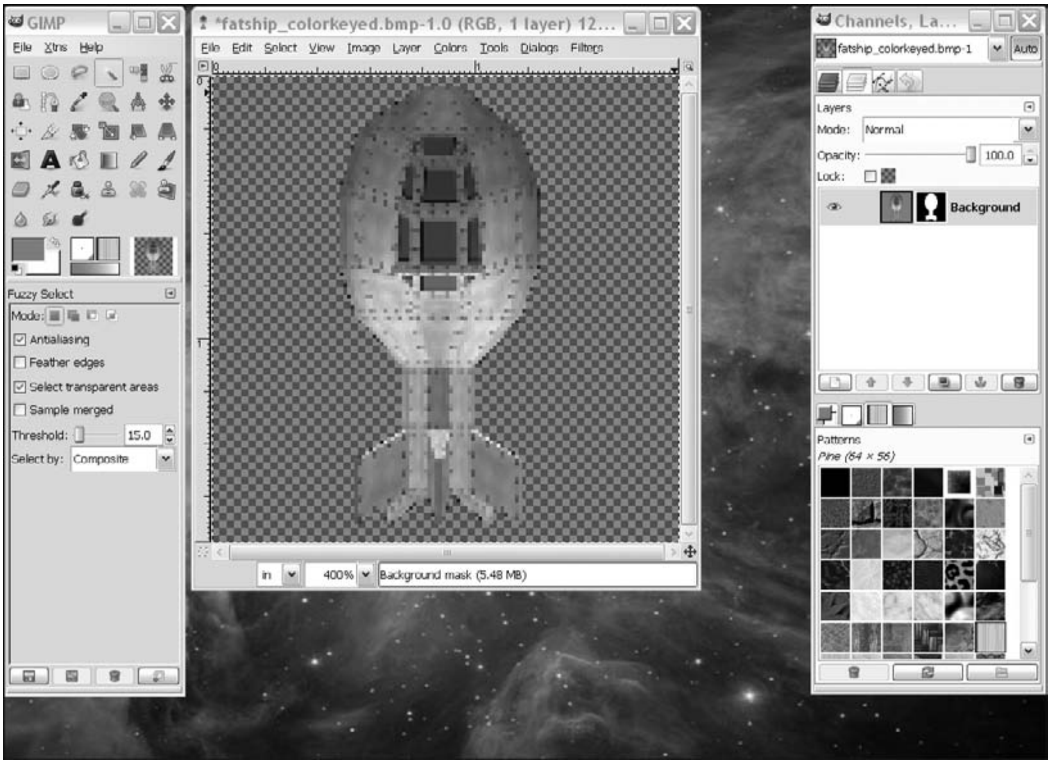


Figure 3.6
This sprite will be rendered with alpha channel transparency.

showing the alpha channel in graphic editors. Figure 3.6 shows the fatship sprite with a new alpha channel.

Advice

Oddly enough, the latest version of the Windows Bitmap format now supports 32-bit RGBA color with an alpha channel, too!

The AlphaDemo program (shown in Figure 3.7) is nearly identical to the ColorkeyDemo program, so I won't list it here again. The only changes that have been made are the program's title and the filename (using the fatship_alpha.tga file instead of fatship_colorkeyed.bmp). The output from the program is the same as the previous one, but this program now renders the sprite with an alpha—we're not using it to its full potential, but we will in due time.

Speaking of potential, these are *very* basic sprite demos. We can't see movement, rotation, or animation, so what was the point of all this? It's important, in my

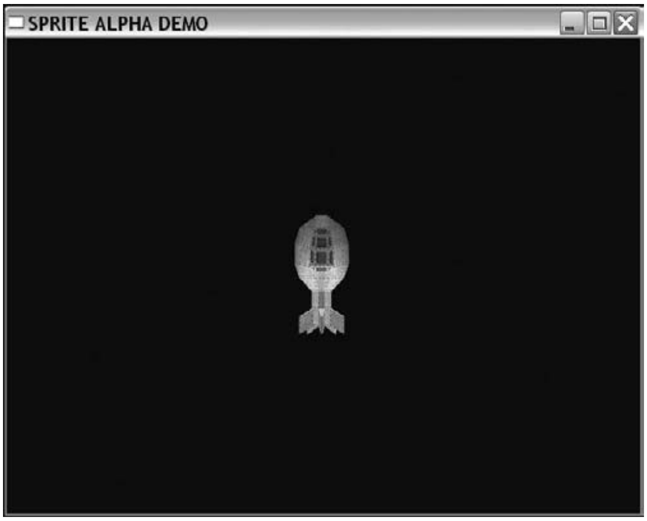


Figure 3.7
The AlphaDemo program demonstrates sprite rendering with an alpha channel.

opinion, to focus on the technology first, and then when that is covered, to focus on use or implementation. We have just added some quite advanced vector and sprite support to the Advanced2D engine and verified that 2D rendering is working (with both color-keyed and alpha transparency). Next, we need to test the more advanced features of the `Sprite` class, which we'll do in the next chapter.

This page intentionally left blank

Chapter 4

Animation

Let's talk about sprite animation. The simplest way to animate a sprite is to load up several images representing an animation sequence and draw those images, one at a time, with some timing. The most obvious problem with this is that you must deal with many image files for each animation sequence—and the typical animated sprite has 30 or so frames! Imagine if you had just a dozen sprites! Obviously, that is not a good way to go about it.

A better solution is to use a sprite sheet. A *sprite sheet* is an image containing many frames for an animation sequence laid out in tiles that are arranged into rows and columns, as shown in Figure 4.1. In this sprite sheet, each horizontal row represents a direction that the sprite can face, with 8 frames of animation and 8 directions, for a total of 64 frames in all.

Using the `Sprite` class developed in the previous chapter, we could create a dragon sprite with code like this:

```
Sprite *dragon = new Sprite();
dragon->loadImage("dragon.tga");
```

This is as far as we've gotten up to this point, but the `Sprite` class supports animation too—we just need to tap into it. Each sprite can have its own individual properties for animation, such as the total frames, number of columns (in the sprite sheet), and animation timing. Let's see how those might be set for the dragon sprite. First, we have to tell the `Sprite` class how large each frame is, because it sets the width and height to the full size of the image by default. The

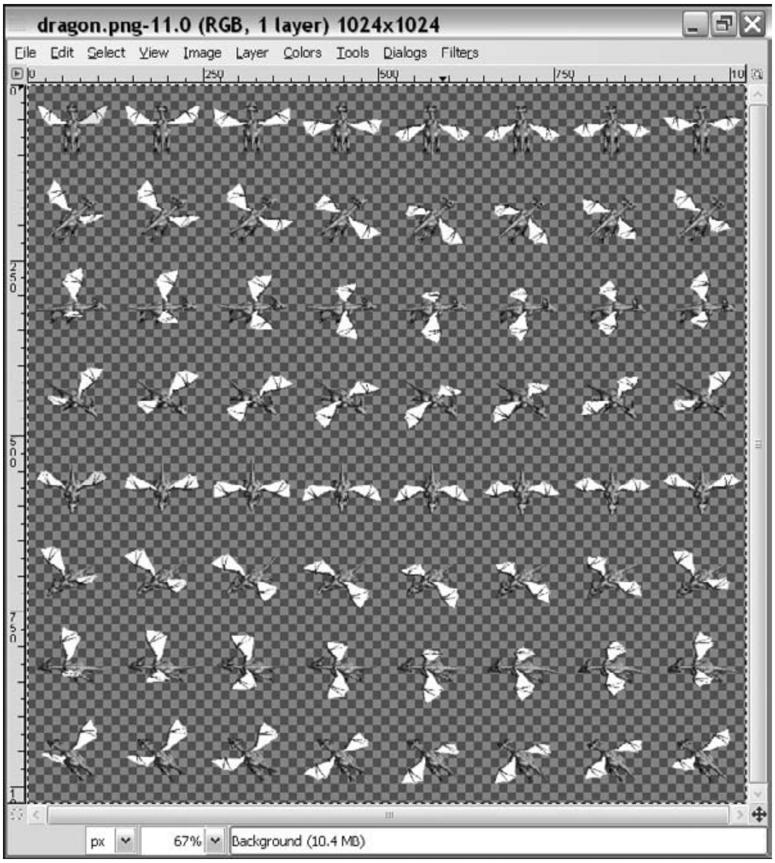

dragon.png-11.0 (RGB, 1 layer) 1024x1024
File Edit Select View Image Layer Colors Tools Dialogs Filters
px
67%
Background (10.4 MB)

Figure 4.1
Animated dragon sprite stored in a sheet of rows and columns.

image size is the size of the whole sprite sheet, while the frame size is the size of each cell of animation.

```
dragon->setSize(96,96);
```

The total number of frames of animation must be set as well. When we're doing animation, the range of valid frame numbers (which are zero-based) will be 0 to `totalFrames` minus 1. The following line of code will cause the animation system to animate the dragon based on frames 0 to 63, and then auto-wrap around to 0.

```
dragon->setTotalFrames(64);
```

To turn on the animation system, you must set the frame timer. This is a millisecond timer that can be used to animate a sprite based on a constant timer regardless of the game's frame rate (60 fps or otherwise). There are additional properties in the `Sprite` class that we will use in some of the future game projects. For instance, the lifetime property can be used to automatically terminate a sprite after a fixed number of milliseconds has passed. (The lifetime of a sprite can be set to cause that sprite to automatically terminate after a fixed duration, which is useful for such things as bullets and explosions.)

Once a sprite is configured with the desired properties, you can animate and draw a sprite using the `Sprite::animate()` and `Sprite::draw()` methods. The `animate()` method looks like this:

```
void Sprite::animate()
{

//update frame based on animdir
if (frametimer > 0) {
    if (timeGetTime() > (DWORD)(framestart + frametimer)) {
        //reset animation timer
        framestart = timeGetTime();
        curframe += animdir;
        //keep frame within bounds
        if (curframe < 0) curframe = totalframes-1;
        if (curframe > totalframes-1) curframe = 0;
    }
}
    else {
        //no animation timer--update at cpu clock speed
        curframe += animdir;
        if (curframe < 0) curframe = totalframes-1;
        if (curframe > totalframes-1) curframe = 0;
    }
}}
```

Note that if you do not set the animation timer, then the animation will run at the full processor clock speed. Since the time is specified in milliseconds, the value you use will be based on the desired frame rate for the sprite. Average rates vary from around 20 to 50 milliseconds per frame.

Animation Demo

Now we will create an example program to demonstrate a single animated sprite. By keeping the demos short and simple, it's my belief that the code is easier to

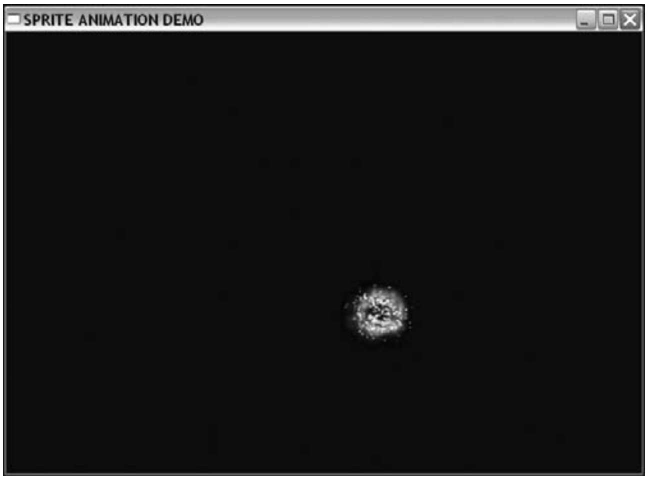


Figure 4.2
The AnimationDemo program draws an animated explosion.

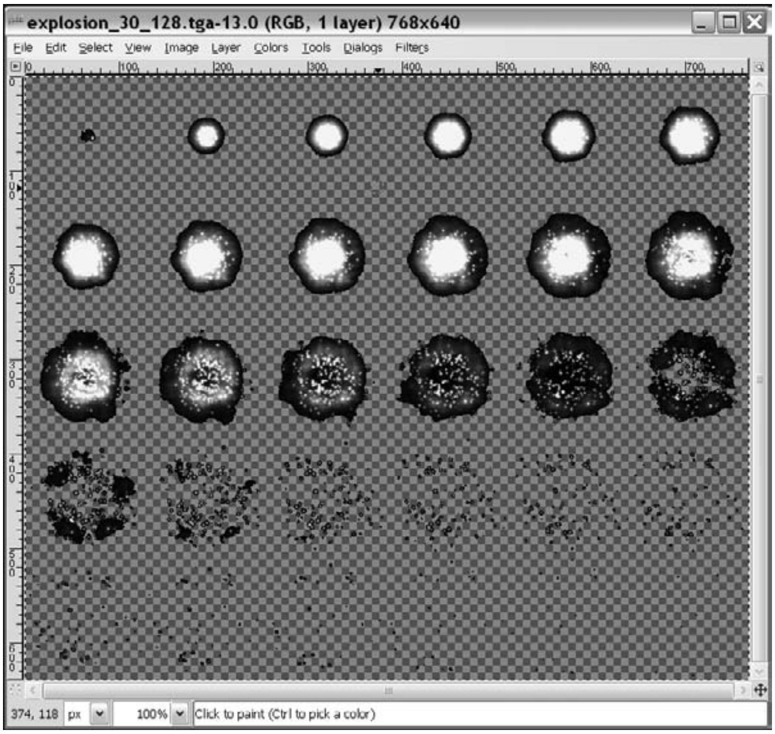

explosion_30_128.tga-13.0 (RGB, 1 layer) 768x640
File Edit Select View Image Layer Colors Tools Dialogs Filters
374, 118 px
100%
Click to paint (Ctrl to pick a color)

Figure 4.3
The sprite sheet for an animated explosion.

understand and learn from. This short program will animate a single explosion, rendering an alpha-transparent targa image at random locations around the screen, as shown in Figure 4.2.

The explosion is composed of 30 128 × 128 sprite frames in a sheet with six columns. Figure 4.3 shows the sprite sheet of the explosion; note the effective use of alpha to produce transparent regions as the explosion dissipates. This shows just how much better alpha is versus the older color-key technology. You can also very easily cause sprites to fade in or out to produce effects such as cloaking or shielding (in the case of a spaceship, for instance). Another popular trick with alpha is to cause a sprite to flicker on and off repeatedly after a collision. One of my favorite tricks is to cycle a sprite's alpha through the red color component when the sprite "dies."

Now for the code.

Advice

The explosion animation was provided courtesy of Reiner Prokein and is available at www.reinerstileset.de.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

Sprite *explosion;

bool game_preload()
{
     g_engine->setAppTitle("SPRITE ANIMATION DEMO");
     g_engine->setFullscreen(false);
     g_engine->setScreenWidth(800);
     g_engine->setScreenHeight(600);
     g_engine->setColorDepth(32);
     return true;
}

bool game_init(HWND)
{
     explosion = new Sprite();
     explosion->loadImage("explosion_30_128.tga");
     explosion->setTotalFrames(30);
     explosion->setColumns(6);
     explosion->setSize(128,128);
     explosion->setFrameTimer(40);
```

```
    return true;
}

void game_update()
{
    int cx,cy;

    //animate the explosion sprite
    explosion->animate();
    if (explosion->getCurrentFrame() == explosion->getTotalFrames() - 1)
    {
        //set a new random location
        cx = rand()%(g_engine->getScreenWidth()-128);
        cy = rand()%(g_engine->getScreenHeight()-128);
        explosion->setPosition(cx,cy);
    }

    //exit when escape key is pressed
    if (KEY_DOWN(VK_ESCAPE)) g_engine->Close();
}

void game_end()
{
    delete explosion;
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));

}

void game_render2d()
{
    //draw the current frame of the explosion
    explosion->drawframe();
}
```

Advice

I have moved the `SetIdentity()` function call directly into the engine's main loop because it was redundant in the game code—since it must be set every frame anyway when rendering in 3D. It's all part of the engine's evolution!

Sprite Rotation and Scaling

We can rotate and scale a sprite with relative ease thanks to the D3DX library. If we want to draw a single-frame sprite, draw a single cell from a sprite sheet, or do full-blown animation, we can use the same multipurpose `Sprite::draw()` function, which looks like this:

```
void Sprite::draw()
{

     //calculate source frame location
     int fx = (this->curframe % this->animcolumns) * this->width;
     int fy = (this->curframe / this->animcolumns) * this->height;
     RECT srcRect = {fx,fy, fx+this->width, fy+this->height};
     //draw the sprite frame
     this->transform();
     g_engine->getSpriteHandler()->Draw( this->image->GetTexture(),
         &srcRect,NULL,NULL,color);
}
```

The `draw()` method calls on `transform()` to perform the translation, rotation, and scaling operations on the sprite before it is rendered.

```
void Sprite::transform()
{
```

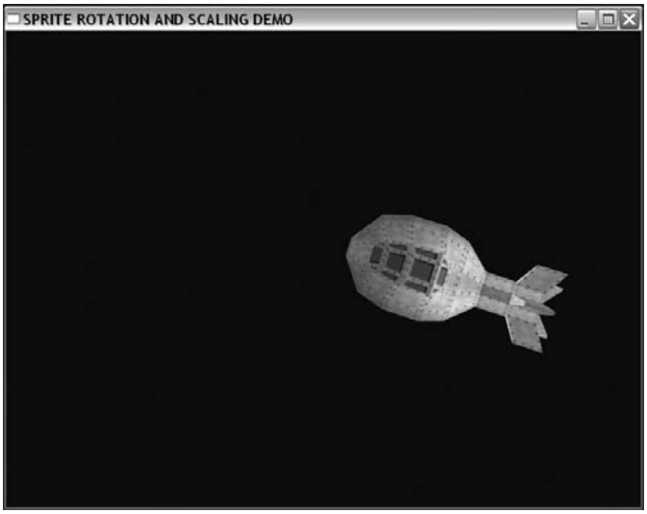


Figure 4.4
The RotateScaleDemo program draws a sprite with rotation and scaling.

```
    3DXMATRIX mat;
    D3DXVECTOR2 scale((float)scaling,(float)scaling);
    D3DXVECTOR2 center((float)(width*scaling)/2, (float)(height*scaling)/2);
    D3DXVECTOR2 trans((float)getX(), (float)getY());
    D3DXMatrixTransformation2D(&mat,NULL,0,&scale,&center,(float)rotation,&trans);
    g_engine->getSpriteHandler()->SetTransform(&mat);
}
```

The D3DX library does it all for us with a single function call, `D3DXMatrixTransformation2D`. This single function creates a matrix with scaling, rotation, and translation all combined. Let me show you what you can do with it. Following is an example called RotateScaleDemo, and a screenshot is shown in Figure 4.4.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

Sprite *ship;

bool game_preload()
{
    g_engine->setAppTitle("SPRITE ROTATION AND SCALING DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(800);
    g_engine->setScreenHeight(600);
    g_engine->setColorDepth(32);
    return true;
}

bool game_init(HWND)
{
    //load sprite
    ship = new Sprite();
    ship->loadImage("fatship.tga");

    return true;
}

void game_update()
{
    static float scale = 0.01f;
    float r,s;

    //set position
    ship->setPosition(400,300);
    //set rotation
    ship->setRotation(timeGetTime()/600.0f);

    //set scale
    s = ship->getScale() + scale;
    if (s < 0.01 || s > 2.5f) scale *= -1;
    ship->setScale(s);

    //exit when escape key is pressed
    if (KEY_DOWN(VK_ESCAPE)) g_engine->Close();
}

void game_end()
{
    delete ship;
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));

}

void game_render2d()
{
    ship->draw();
}
```

Animation with Transforms

We can apply this functionality to animation as well. Since `D3DXSprite` is used to draw single- or multi-frame sprites, you can use the same transformation to rotate and scale a sprite regardless of whether it's animated. Figure 4.5 shows a sprite sheet containing frames from an animated space rock or asteroid.

This program uses the same transforms that were applied to the fatship sprite in the previous example; the difference now is that we're dealing with an animated sprite. What's the difference? As far as Direct3D is concerned, there is none. Our `Sprite::draw()` method handles single "static" sprites as well as sprites with animation.

Let's give animation with rotation and scaling a try. Figure 4.6 shows the output from the RotateAnimDemo program, with the code listing to follow.

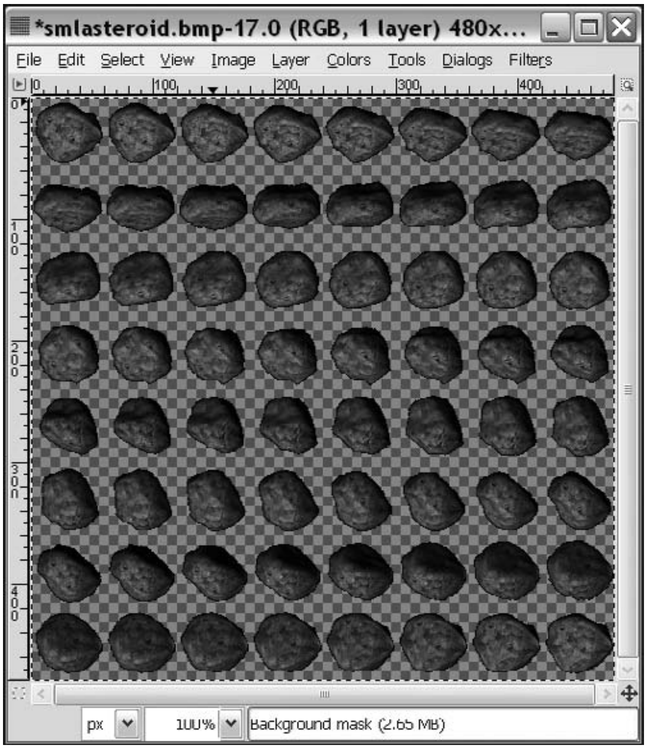

*smlasteroid.bmp-17.0 (RGB, 1 layer) 480x...
File Edit Select View Image Layer Colors Tools Dialogs Filters
px
100%
Background mask (2.65 MB)

Figure 4.5
A 64-frame animated asteroid.

Figure 4.6
The RotateAnimDemo program draws an animated sprite with rotation and scaling.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

Sprite *asteroid;

bool game_preload()
{
     g_engine->setAppTitle("SPRITE ANIMATE/ROTATE/SCALE DEMO");
     g_engine->setFullscreen(false);
     g_engine->setScreenWidth(800);
     g_engine->setScreenHeight(600);
     g_engine->setColorDepth(32);
     return true;
}

bool game_init(HWND)
{
     //load sprite
     asteroid = new Sprite();
     asteroid->loadImage("asteroid.tga");
     asteroid->setTotalFrames(64);
     asteroid->setColumns(8);
     asteroid->setSize(60,60);
     asteroid->setFrameTimer(30);

     return true;
}

void game_update()
{
     static float scale = 0.005f;
     float r,s;

     //set position
     asteroid->setPosition(400,300);

     //set rotation
     asteroid->setRotation(timeGetTime()/600.0f);

     //set scale
     s = asteroid->getScale() + scale;
     if (s < 0.25 || s > 5.0f) scale *= -1;
     asteroid->setScale(s);
```

```
    //exit when escape key is pressed
    if (KEY_DOWN(VK_ESCAPE)) g_engine->Close();
}

void game_end()
{
    delete asteroid;
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));

}

void game_render2d()
{
    asteroid->animate();
    asteroid->draw();
}
```

Particles

Particles are tiny sprites that are rendered with about 50-percent alpha transparency so that they seem to glow. The key to creating a particle system—that is, an emitter or other special effect—is to start with a good source particle image. Figure 4.7 shows an enlarged view of a 16 × 16 particle sprite. Note the amount of alpha transparency in the image—only the central white portion is fully opaque, while the rest will blend with whatever background the particle is rendered over.

Sprite-based particles differ significantly from shader-based particles rendered by the 3D hardware. Three-dimensional particles can emit light (emissive) or reflect light (reflective) and can be used to simulate *real* smoke and fog. Sprite-based particles can be used to generate smoke trails behind missiles and spaceships, among other things.

A so-called *particle system* is a managed list of particles that are rendered in creative ways. That list takes the form of an `std::vector`—something we have demonstrated before, but have not yet fully utilized in the game engine. (That is the subject of Chapter 7, "Entities."). An `std::vector` will work *slightly* faster

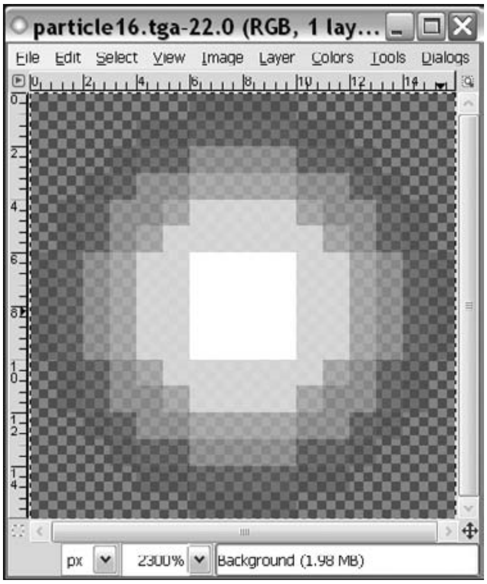

particle16.tga-22.0 (RGB, 1 lay...
File Edit Select View Image Layer Colors Tools Dialogs
px
2300%
Background (1.98 MB)

Figure 4.7
Source particle image.

than an `std::list` when your list does not need to change very often. Our particle emitter will create particles but not remove any (until the object is destroyed, that is). An `std::list` would be preferred if you needed to add and remove items regularly, but it's not quite as fast as an `std::vector` when it comes to sequential iteration.

To make working with particles more reasonable, we'll code up the most obvious functionality into a class. Following is the definition for the `ParticleEmitter` class. This class uses an `std::vector` filled with `Sprite` objects to represent the entities in the emitter. The class is otherwise completely self contained and can handle most types of particle systems that I have seen over the years. Basically, a great particle system works in such a way that the player shouldn't notice that it's a particle at all. When a spaceship is cruising through space, it can emit a flame and smoke with the use of two particle emitters, for example. This code belongs in the ParticleEmitter.h file.

```
#include "Advanced2D.h"
#pragma once
namespace Advanced2D {
    class ParticleEmitter
    {
```

```
    private:
        typedef std::vector<Sprite*>::iterator iter;
        std::vector<Sprite*> particles;
        Texture *image;
        Vector3 position;
        double direction;
        double length;
        int max;
        int alphaMin,alphaMax;
        int minR,minG,minB,maxR,maxG,maxB;
        int spread;
        double velocity;
        double scale;
    public:
        void setPosition(double x, double y) { position.Set(x,y,0); }
        void setPosition(Vector3 vec) { position = vec; }
        Vector3 getPosition() { return position; }
        void setDirection(double angle) { direction = angle; }
        double getDirection() { return direction; }
        void setMax(int num) { max = num; }
        void setAlphaRange(int min,int max);
        void setColorRange(int r1,int g1,int b1,int r2,int g2,int b2);
        void setSpread(int value) { spread = value; }
        void setLength(double value) { length = value; }
        void setVelocity(double value) { velocity = value; }
        void setScale(double value) { scale = value; }

        ParticleEmitter();
        virtual ~ParticleEmitter();
        bool loadImage(std::string imageFile);
        void draw();
        void update();
        void add();
    }; //class
}; //namespace
```

Following is the implementation file for the `ParticleEmitter` class. I'll explain how it works at the end of the code listing.

```
#include "Advanced2D.h"
namespace Advanced2D {
    ParticleEmitter::ParticleEmitter()
    {
```

```
    //initialize particles to defaults
    image = NULL;
    max = 100;
    length = 100;
    direction = 0;
    alphaMin = 254; alphaMax = 255;
    minR = 0; maxR = 255;
    minG = 0; maxG = 255;
    minB = 0; maxB = 255;
    spread = 10;
    velocity = 1.0f;
    scale = 1.0f;
}

bool ParticleEmitter::loadImage(std::string imageFile)
{
    image = new Texture();
    return image->Load(imageFile);
}


ParticleEmitter::~ParticleEmitter()
{
    delete image;

    //destroy particles
    for (iter i = particles.begin(); i != particles.end(); ++i)
    {
        delete *i;
    }
    particles.clear();
}

void ParticleEmitter::add()
{
    static double PI_DIV_180 = 3.1415926535 / 180.0f;
    double vx,vy;

    //create a new particle
    Sprite *p = new Sprite();
    p->setImage(image);
    p->setPosition(position.getX(), position.getY());
```

```
        //add some randomness to the spread
        double variation = (rand() % spread - spread/2) / 100.0f;

        //set linear velocity
        double dir = direction - 90.0;
        vx = cos( dir * PI_DIV_180) + variation;
        vy = sin( dir * PI_DIV_180) + variation;
        p->setVelocity(vx * velocity,vy * velocity);

        //set random color based on ranges
        int r = rand()%(maxR-minR)+minR;
        int g = rand()%(maxG-minG)+minG;
        int b = rand()%(maxB-minB)+minB;
        int a = rand()%(alphaMax-alphaMin)+alphaMin;
        p->setColor(D3DCOLOR_RGBA(r,g,b,a));

        //set the scale
        p->setScale( scale );

        //add particle to the emitter
        particles.push_back(p);

    }

    void ParticleEmitter::draw()
    {
        //draw particles
        for (iter i = particles.begin(); i != particles.end(); ++i)
        {
            (*i)->draw();
        }
    }

    void ParticleEmitter::update()
    {
        static Timer timer;

        //do we need to add a new particle?
        if ((int)particles.size() < max)
        {
            //trivial but necessary slowdown
            if (timer.stopwatch(1)) add();
        }
        for (iter i = particles.begin(); i != particles.end(); ++i)
        {
            //update particle's position
            (*i)->move();

            //is particle beyond the emitter's range?
            if ( (*i)->getPosition().Distance(this->position) > length)
            {
                //reset particle to the origin
                (*i)->setX(position.getX());
                (*i)->setY(position.getY());
            }
        }
    }

    void ParticleEmitter::setAlphaRange(int min,int max)
    {
        alphaMin=min;
        alphaMax=max;
    }

void ParticleEmitter::setColorRange(int r1,int g1,int b1,int r2,int g2,int b2)
    {
        minR = r1; maxR = r2;
        minG = g1; maxG = g2;
        minB = b1; maxB = b2;
    }
}
```

Using the `ParticleEmitter` class is very easy; you just have to supply the source image. That image can be any reasonably nice-looking circle on a bitmap, or perhaps a simple square image if you want to produce a blocky effect. I have created a circle on a 16 × 16 bitmap with several shades of alpha built into the image. Combined with the color and alpha effects we'll apply when drawing the image, this will produce the particles in our emitter. However, you can produce quite different particles using a different source image—something to keep in mind!

Here is how you can create a simple emitter. This example code creates a new particle emitter using the particle16.tga image; sets it at screen location 400,300; sets the angle to 45 degrees; sets a maximum of 1,000 particles; sets an alpha range of 0 to 100 (which is faint); sets the random spread from the given angle to 30 pixels; and sets the range to 250 pixels.

```
ParticleEmitter *p new ParticleEmitter();
p->loadImage("particle16.tga");
p->setPosition(400,300);
p->setDirection(45);
p->setMax(1000);
p->setAlphaRange(0,100);
p->setSpread(30);
p->setLength(250);
```

After creating the emitter, you need to give it a chance to update its particles and draw itself. The `ParticleEmitter::update()` method should be called from your `game_update()` function, while `ParticleEmitter::draw()` should be called from your `game_render2d()` function.

Following is an example program called ParticleDemo that demonstrates just a few of the possibilities! This example creates two normal emitters, a rotation pattern, and then two emitters together, rotating in a circle, generating a smoke-like effect. Figure 4.8 shows the output with a white background. To really

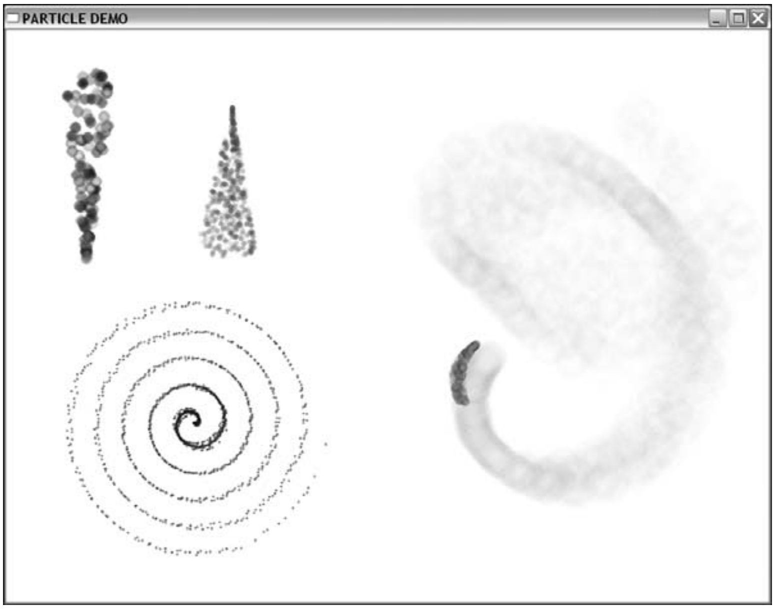


Figure 4.8
Particle demonstration with a white background.

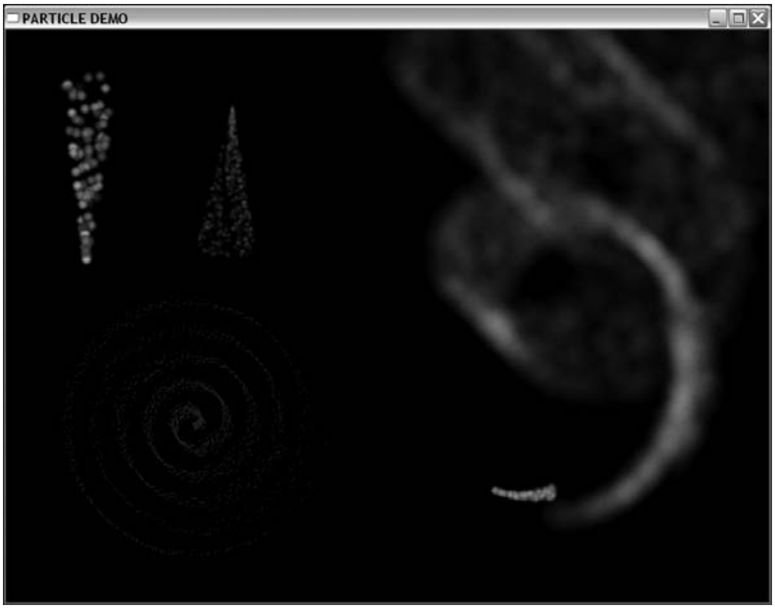


Figure 4.9
Particle demonstration with a black background.

appreciate the alpha blending taking place here, you must see it with a dark background, as shown in Figure 4.9.

There's a lot of code in the ParticleTest program, which is listed below. But most of this is setup code to configure the many particle emitters demonstrated in the program. Once the emitters are configured, the rest of the program listing is fairly short, with just calls to update and draw each emitter.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

ParticleEmitter *pa;
ParticleEmitter *pb;
ParticleEmitter *pc;
ParticleEmitter *pd;
ParticleEmitter *pe;

bool game_preload()
{
    g_engine->setAppTitle("PARTICLE DEMO");
    g_engine->setFullscreen(false);
```

```
        g_engine->setScreenWidth(1024);
        g_engine->setScreenHeight(768);
        g_engine->setColorDepth(32);
        return 1;
}

bool game_init(HWND)
{
        g_engine->setMaximizeProcessor(true);

        pa = new ParticleEmitter();
        pa->loadImage("particle16.tga");
        pa->setPosition(100,300);
        pa->setDirection(0);
        pa->setMax(500);
        pa->setAlphaRange(100,255);
        pa->setSpread(30);
        pa->setVelocity(2.0);
        pa->setLength(250);

        pb = new ParticleEmitter();
        pb->loadImage("particle16.tga");
        pb->setPosition(300,100);
        pb->setDirection(180);
        pb->setScale(0.6);
        pb->setMax(500);
        pb->setAlphaRange(0,100);
        pb->setColorRange(200,0,0,255,10,10);
        pb->setVelocity(2.0);
        pb->setSpread(40);
        pb->setLength(200);

        pc = new ParticleEmitter();
        pc->loadImage("particle16.tga");
        pc->setPosition(250,525);
        pc->setDirection(0);
        pc->setScale(0.5);
        pc->setMax(2000);
        pc->setAlphaRange(100,150);
        pc->setColorRange(0,0,200,10,10,255);
        pc->setVelocity(0.2);
        pc->setSpread(5);
```

```
    pc->setLength(180);
    pd = new ParticleEmitter();
    pd->loadImage("particle16.tga");
    pd->setPosition(750,650);
    pd->setScale(0.75);
    pd->setMax(10);
    pd->setAlphaRange(50,100);
    pd->setColorRange(210,50,0,255,255,1);
    pd->setVelocity(2.0);
    pd->setDirection(0);
    pd->setSpread(40);
    pd->setLength(100);

    pe = new ParticleEmitter();
    pe->loadImage("particle16.tga");
    pe->setPosition(730,575);
    pe->setScale(4.0f);
    pe->setMax(1000);
    pe->setAlphaRange(1,20);
    pe->setColorRange(250,250,250,255,255,255);
    pe->setVelocity(2.0);
    pe->setDirection(0);
    pe->setSpread(80);
    pe->setLength(800);

    return true;
}

void game_update()
{
    //move particles
    pa->update();
    pb->update();

    //update the circular emitter
    float dir = pc->getDirection() + 0.2f;
    pc->setDirection(dir);
    pc->update();

    //update the rotating emitter
    static double unit = 3.1415926535 / 36000.0;
    static double angle = 0.0;
    static double radius = 150.0;
    angle += unit;
    if (angle > 360) angle = 360 - angle;
    float x = 750 + cos(angle) * radius;
    float y = 500 + sin(angle) * radius;
    pd->setPosition(x,y);
    pd->update();

    //update smoke emitter
    pe->setPosition(x,y);
    pe->update();

    //exit when escape key is pressed
    if (KEY_DOWN(VK_ESCAPE)) g_engine->Close();
}

void game_end()
{
    delete pa;
    delete pb;
    delete pc;
    delete pd;
    delete pe;
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,0));
}

void game_render2d()
{
    pa->draw();
    pb->draw();
    pc->draw();
    pd->draw();
    pe->draw();
}
```

There is so much more potential for particles than what you've seen here! In Chapter 10, "Math," you will learn to calculate the angle between two sprites, which will make it possible to emit a particle stream from the rear direction of a spaceship or rocket and have that stream point in the right direction. You could also create a slow particle emitter to simulate smoke and have it appear as if a ship, aircraft, or other type of game object is damaged!

Now it feels as if we've been dealing with sprites for a long time now, so let's take a break! The next two chapters cover device input and game audio, which will be a nice change of pace.

This page intentionally left blank

CHAPTER 5

Input

Getting input from the user is as important as rendering, but this subject does not often get as much attention because, frankly, it just doesn't change very often. We're going to use DirectInput to get input from the keyboard and the mouse in this chapter. DirectInput hasn't changed in many years and is still at version 8.1. Contrast that with the huge changes taking place with Direct3D every year! Although we can use a joystick, it's such a non-standard device for the PC that it's not worth the effort. Granted, if you're working on a game that would benefit from a joystick, by all means support it! Just note that most PC gamers prefer a keyboard and mouse. If you want to support an Xbox 360 controller, you can look into the XInput library, which is now packaged with the DirectX SDK (as well as included with XNA Game Studio).

Keyboard Input

The keyboard is the standard input device, even for games that don't specifically use it, so it is a given that your games must support a keyboard one way or another. If nothing else, you should allow the user to exit your game or at least bring up an in-game menu by pressing the Escape key. (That's the standard.) The primary DirectInput object is called `IDirectInput8`; you can reference it directly or use the `LPDIRECTINPUT8` pointer. Why is the number "8" attached to these interfaces? Because DirectInput has not changed version 8.1. The DirectInput library file is called dinput8.lib (libdinput8.a for Dev-C++), so be sure to add this file to the list of linked files for your project. I'll assume that you read the previous chapters and learned how to set up a project to support DirectX.

DirectInput Device

Here is how to scan the keyboard for key presses. You will want to first define the primary DirectInput object used by your program, along with the device:

```
LPDIRECTINPUT8 dinput;

LPDIRECTINPUTDEVICE8 dinputdev;
```

After defining these objects, you can then call `DirectInputCreate8` to initialize DirectInput. This function creates the primary DirectInput object. The first parameter is the instance handle for the current program. A convenient way to get the current instance when it is not immediately available (normally this is only found in `Winmain`) is by using the `GetModuleHandle` function. The second parameter is the DirectInput version, which is always passed as `DIRECTINPUT_VERSION`, defined in dinput.h. The third parameter is a reference identifier for the version of DirectInput that you want to use, which is usually `IID_IDirectInput8`. The fourth parameter is a pointer to the DirectInput object pointer (yes, that's a pointer to a pointer), and the fifth parameter is always `NULL`. Here is an example of how you might call this function:

```
HRESULT result = DirectInput8Create(
    GetModuleHandle(NULL),
    DIRECTINPUT_VERSION,
    IID_IDirectInput8,
    (void**)&p_dinput,
    NULL);
```

After initializing the object, you can then use it to create a DirectInput device for a specific input device (usually just a keyboard or a mouse) by calling the `CreateDevice` function on the returned `DirectInput` object.

The first parameter is a value that specifies the type of object you want to create, which should be either `GUID_SysKeyboard` or `GUID_SysMouse`.

The second parameter is your device pointer that receives the address of the DirectInput device. The third parameter is always `NULL`. Here is how you might call this function:

```
result = p_dinput->CreateDevice(GUID_SysKeyboard, &dikeyboard, NULL);
```

Initializing the Keyboard

Once you have created a DirectInput keyboard device, you can then initialize the keyboard handler to prepare it for input. The next step is to set the keyboard's data format, which instructs DirectInput how to pass the data back to your program. It is abstracted in this way because there are hundreds of input devices on the market with myriad features, so there has to be a uniform way to read them all.

Setting the Data Format

The `SetDataFormat` function specifies which data format will be used.

The single parameter to this function specifies the device type. For the keyboard, you want to pass the value of `c_dfDIKeyboard` as this parameter. The constant for a mouse would be `c_dfDIMouse`. Here, then, is a sample function call:

```
HRESULT result = dikeyboard->SetDataFormat(&c_dfDIKeyboard);
```

Note that you do not need to define `c_dfDIKeyboard` yourself; it is defined in dinput.h.

Setting the Cooperative Level

The next step is to set the cooperative level, which determines how much of the keyboard DirectInput will give your program by way of priority. To set the cooperative level, you call the `SetCooperativeLevel` function. The first parameter is the window handle. The second parameter is the interesting one—it specifies the priority that your program will have over the keyboard or mouse. The most common values to pass when working with the keyboard are `DISCL_EXCLUSIVE` and `DISCL_FOREGROUND` (in either full-screen or windowed mode). When you gain exclusive use of the keyboard, DirectInput will still allow key combinations such as Alt+Tab and Ctrl+Alt+Delete because those key combinations are detected at a lower level within Windows. So, here is how you might call the function:

```
HRESULT result = dikeyboard->SetCooperativeLevel( hwnd,
    DISCL_NONEXCLUSIVE | DISCL_FOREGROUND );
```

There is a valid argument to be made for acquiring the keyboard and mouse in nonexclusive mode in order to support screen captures and minimizing the window, among other reasons. While developing a game, I like to work in nonexclusive mode so I can do these things without having to restart a program using DirectInput. If the program window loses focus, then DirectInput loses the devices—requiring a manual re-acquire on our part. That being the case, using exclusive mode does not really cause any problems. I recommend nonexclusive for the keyboard and exclusive for the mouse. Give it a try, in both windowed and full-screen modes, and determine which level of control you prefer.

Acquiring the Device

The last step in the process is to acquire the keyboard device using the `Acquire` function. If the function returns a positive value (`DI_OK`), then you have successfully acquired the keyboard and you are ready to start checking for key presses.

Remember to always release the keyboard when you are done using it, or you could leave the keyboard handler in an unknown state. You cannot rely on Windows or DirectInput to clean up after you. Each DirectInput device has an `Unacquire` function.

Reading Key Presses

Somewhere in your game loop you need to poll the keyboard to update its key values. We need to define the array of keys that are to be populated with the key states:

```
char keys[256];
```

You must poll the keyboard to fill in this array of characters, and to do that you call the `GetDeviceState` function. This function is used for all devices regardless of type, so it is standard for all input devices. The first parameter is the size of the device state buffer to be filled with data, and the second parameter is a pointer to the data. Here is how you can poll the keyboard state:

```
dikeyboard->GetDeviceState(sizeof(keys), (LPVOID)&keys);
```

You can then check the `keys` array for values corresponding to the DirectInput key codes, which are all listed in the dinput.h header file.

To check the state of a key, you need to perform a simple bit-mask comparison with one of the DirectInput key codes. Here is how to check the state of the Escape key:

```
if (keys[DIK_ESCAPE] & 0x80) . . .
```

By putting this sort of comparison inside a loop that scans all keys, you can automatically detect key presses within the game engine and pass on the key-press events to the game in more of an event-based system (rather than a polled system). We'll get into that later in the chapter, after going over the mouse-specific code.

Mouse Input

After you have written a handler for the keyboard, the mouse is a bit easier to deal with because the DirectInput object will already exist. The code to initialize and poll the mouse is very similar to the keyboard code. First, define the mouse device variable:

```
LPDIRECTINPUTDEVICE8 dimouse;
```

Next, create the mouse device:

```
result = p_dinput->CreateDevice(GUID_SysMouse, &dimouse, NULL);
```

Initializing the Mouse

Assuming the DirectInput object is already initialized, the next step is to set the data format for the mouse, which instructs DirectInput how to pass the data back to your program.

Setting the Data Format

We'll use the `SetDataFormat` function for the mouse as well as for the keyboard. The single parameter to this function specifies the device type, which for the mouse should be the predefined `c_dfDIMouse`. Here is an example:

```
HRESULT result = dimouse->SetDataFormat(&c_dfDIMouse);
```

Note, again, that you do not need to define `c_dfDIMouse`, because it is defined in dinput.h.

Setting the Cooperative Level

The next step (again, like the keyboard interface) is to set the cooperative level, which determines how much priority over the mouse DirectInput will give your program. To set the cooperative level, you call the `SetCooperativeLevel` function with the window handle and the mouse priority. Common values are DISCL_EXCLUSIVE and DISCL_FOREGROUND (which has the added benefit of hiding the stock Windows cursor from view). If your game is running full screen, then you may consider gaining exclusive access to the input devices. I often use non-exclusive mode in my code. Here is an example:

```
HRESULT result = dimouse->SetCooperativeLevel( hwnd,
    DISCL_EXCLUSIVE | DISCL_FOREGROUND);
```

Acquiring the Device

The last step is to acquire the mouse device using the Acquire function. If the function returns DI_OK, then you have successfully acquired the mouse and you are ready to start checking for movement and button presses. As with the keyboard device, you must unacquire the mouse and release the mouse device after you are finished using it.

Reading the Mouse

Somewhere in your game loop you need to poll the mouse to update the mouse position and button status using the GetDeviceState function (again, the same as with the keyboard). You will use the DIMOUSESTATE struct to poll the mouse:

```
typedef struct DIMOUSESTATE {
    LONG lX;
    LONG lY;
    LONG lZ;
    BYTE rgbButtons[4];
} DIMOUSESTATE;
```

To fill the DIMOUSESTATE struct, call the GetDeviceState function:

```
DIMOUSESTATE mouse_state;
dimouse->GetDeviceState(sizeof(mouse_state), (LPVOID)&mouse_state);
```

There is an alternate struct available for your use when you want to support complex mouse devices with more than four buttons, in which case the button array is doubled in size but the struct is otherwise the same:

```
typedef struct DIMOUSESTATE2 {
    LONG lX;
    LONG lY;
    LONG lZ;
    BYTE rgbButtons[8];
} DIMOUSESTATE2;
```

Because multi-button mouse devices are rare and cannot be relied upon for standard input in a game, it's usually best to support only two or three mouse buttons, at least for standard input. You might support additional mouse buttons as a shortcut or macro for common game functions (such as grouping units or doing a double jump or something similar).

After polling the mouse, you can then check the `mouse_state` struct for x and y motion and button presses. You can check for mouse movement, also called *mickeys,* using the `lX` and `lY` member variables. What are mickeys? *Mickeys* represent motion of the mouse rather than an absolute position, so you must keep track of the old position if you want to use these mouse-positioning values to draw your own pointer. Mickeys are a convenient way of handling mouse motion because you can continue to move in a single direction, and the mouse will continue to report movement, even if the "pointer" would have reached the edge of the screen.

As you can see from the struct, the `rgbButtons` array holds the result of button presses. If you want to check for a specific button (starting with 0 for button 1), here is how you might do that:

```
button_1 = obj.rgbButtons[0] & 0x80;
```

A more convenient method of detecting button presses is by using a macro definition in code:

```
#define BUTTON_DOWN(obj, button) (obj.rgbButtons[button] & 0x80)
```

By using the macro definition, you can check for button presses like this:

```
button_1 = BUTTON_DOWN(mouse_state, 0);
```

Engine Modifications

You now have all the information you need to add keyboard and mouse support to your games. But let's transform this code into something more reusable and avoid having to write any DirectInput code again in the future. To maximize code reuse, I've written a helper class called `Input` that encapsulates the keyboard and mouse. This class will be utilized by the Advanced2D engine to automatically provide keyboard and mouse events to a game.

Input Class

Here is the definition of the `Input` class:

```
#include "Advanced2d.h"
#pragma once
namespace Advanced2D {
    class Input {
    private:
        HWND window;
        IDirectInput8 *di;
        IDirectInputDevice8 *keyboard;
        char keyState[256];
        IDirectInputDevice8 *mouse;
        DIMOUSESTATE mouseState;
        POINT position;
    public:
        Input( HWND window );
        virtual ~Input();
        void Update();
        bool GetMouseButton( char button );

        char GetKeyState(int key) { return keyState[key]; }
        long GetPosX() { return position.x; }
        long GetPosY() { return position.y; }
        long GetDeltaX() { return mouseState.lX; }
        long GetDeltaY() { return mouseState.lY; }
        long GetDeltaWheel() { return mouseState.lZ; }
    };
};
```

Now we'll take a look at the implementation of the `Input` class, which is also rather short. This class just wraps a DirectInput keyboard and device without providing any additional processing of input data.

```
#include "Advanced2D.h"
namespace Advanced2D {
    Input::Input( HWND hwnd )
    {
        //save window handle
        window = hwnd;

        //create DirectInput object
        DirectInput8Create( GetModuleHandle(NULL), DIRECTINPUT_VERSION,
            IID_IDirectInput8, (void**)&di, NULL );
        //initialize keyboard
        di->CreateDevice(GUID_SysKeyboard, &keyboard, NULL);
        keyboard->SetDataFormat( &c_dfDIKeyboard );
        keyboard->SetCooperativeLevel( window,
            DISCL_FOREGROUND | DISCL_NONEXCLUSIVE );
        keyboard->Acquire();

        //initialize mouse
        di->CreateDevice(GUID_SysMouse, &mouse, NULL);
        mouse->SetDataFormat(&c_dfDIMouse);
        mouse->SetCooperativeLevel(window,DISCL_FOREGROUND|DISCL_NONEXCLUSIVE);
        mouse->Acquire();
    }

    Input::~Input()
    {
        di->Release();
        keyboard->Release();
        mouse->Release();
    }

    void Input::Update()
    {
        //poll state of the keyboard
        keyboard->Poll();
        if (!SUCCEEDED(keyboard->GetDeviceState(256,(LPVOID)&keyState)))
        {
            //keyboard device lost, try to re-acquire
            keyboard->Acquire();
        }

        //poll state of the mouse
        mouse->Poll();
        if (!SUCCEEDED(mouse->GetDeviceState(sizeof(DIMOUSESTATE),&mouse-
State)))
        {
            //mouse device lose, try to re-acquire
            mouse->Acquire();
        }

        //get mouse position on screen
        GetCursorPos(&position);
        ScreenToClient(window, &position);
    }

    bool Input::GetMouseButton( char button )
    {
        return ( mouseState.rgbButtons[button] & 0x80 );
    }
};
```

Engine Changes

We are not trying to build the engine in a step-by-step fashion in this book because, as mentioned previously, it's too difficult to keep track of the changes. Instead, I'll show you the key code and classes added to the engine whenever a modification takes place, and I'll recommend that you open the Engine project in the chapter to examine the code that has been added.

I mentioned earlier that we want to encapsulate keyboard and mouse input and transform it from a polled system to an event-based system. This means that we don't want to poll the keyboard and mouse every frame—we want the game engine to do that and just tell us when input has been detected.

In the Advanced2D.h file, the following new external functions have been added. These are the functions that will be called in the game's code, so these functions must all be defined in your game to avoid a linker error. (Even if you don't need them all, they must all be included.) We have events for key press and release, mouse buttons, mouse relative movement, mouse position, and mouse wheel!

```
extern void game_keyPress(int key);
extern void game_keyRelease(int key);
extern void game_mouseButton(int button);
extern void game_mouseMotion(int x,int y);
extern void game_mouseMove(int x,int y);
extern void game_mouseWheel(int wheel);
```

Also in the Advanced2D.h header, we add an object variable from the `Input` class and two helper functions: `UpdateKeyboard()` will poll and process key events, while `UpdateMouse()` will handle mouse events.

```
Input *p_input;
void UpdateKeyboard();
void UpdateMouse();
```

In addition to these definitions, we must include the dinput.h file in Advanced2D.h so the DirectInput library is available for use in the game engine. Note that the `Input` object is a private property; there's no need to expose it. Over in the Advanced2D.cpp file we have a few things to do. First, the keyboard and mouse must be initialized by creating a new instance of the `Input` class. Examine the `Engine::Init()` method where the following code has been added:

```
//initialize DirectInput
p_input = new Input(this->windowHandle);
```

Next, scrolling down to the `Engine::Update()` method, look for the section of code that performs the frame rate calculation, and you will find the following new code that updates the keyboard and mouse states:

```
//update input devices
p_input->Update();
this->UpdateKeyboard();
this->UpdateMouse();
```

Finally, there are the two class methods mentioned earlier that must be implemented in the Advanced2D.cpp file, as follows:

```
void Engine::UpdateMouse()
{
    static int oldPosX = 0;
    static int oldPosY = 0;
    int deltax = p_input->GetDeltaX();
    int deltay = p_input->GetDeltaY();

    //check mouse buttons 1-3
    for (int n=0; n<4; n++) {
        if (p_input->GetMouseButton(n))
            game_mouseButton(n);
    }

    //check mouse position
    if (p_input->GetPosX() != oldPosX || p_input->GetPosY() != oldPosY) {
        game_mouseMove(p_input->GetPosX(), p_input->GetPosY() );
        oldPosX = p_input->GetPosX();
        oldPosY = p_input->GetPosY();
    }
    //check mouse motion
    if (deltax != 0 || deltay ) {
        game_mouseMotion(deltax,deltay);
    }

    //check mouse wheel
    int wheel = p_input->GetDeltaWheel();
    if (wheel != 0)
        game_mouseWheel(wheel);
}

void Engine::UpdateKeyboard()
{
    static char old_keys[256];
    for (int n=0; n<255; n++)
    {
        //check for key press
        if (p_input->GetKeyState(n) & 0x80) {
            game_keyPress(n);
            old_keys[n] = p_input->GetKeyState(n);
        }
        //check for release
        else if (old_keys[n] & 0x80) {
            game_keyRelease(n);
            old_keys[n] = p_input->GetKeyState(n);
        }
    }
}
```

Testing Keyboard and Mouse Input

That's a lot of code, and we've burned through the keyboard and mouse input subject quickly in this chapter! A demo is called for to test the functionality of the new input routines in the engine, as well as the new event functions added to the exports. I've got a great idea: We'll create a demo that uses the mouse to draw particles! I've wanted to work more with the `ParticleEmitter` class again, and this is a good way to play with it while simultaneously testing input. The new DirectInput keyboard events will replace the old `KEY_DOWN` macro for the purpose of exiting the program via the Escape key. The mouse is used to draw particles with random velocities. See Figure 5.1.

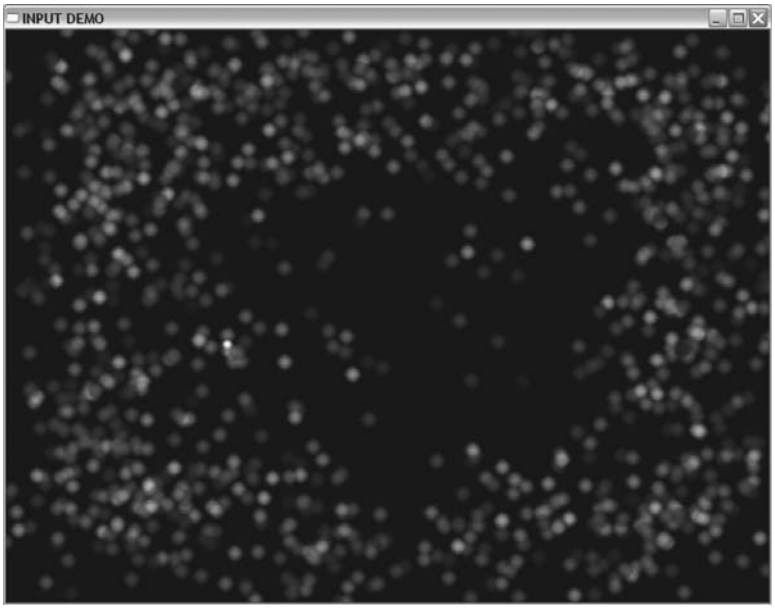


Figure 5.1
The InputDemo program demonstrates keyboard and mouse input.

Although the InputDemo program runs fine in a window, I recommend running it in full-screen and windowed mode to see for yourself how the exclusive or nonexclusive mode affects the mouse and keyboard. If you're using exclusive mode, the mouse will be completely trapped by the program unless you Alt+Tab to another window. Because the `Input` class intelligently handles the loss of devices, it will re-acquire the keyboard and mouse when you return to the demo.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

ParticleEmitter *p;
Sprite *cursor;

bool game_preload()
{
    g_engine->setAppTitle("INPUT DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(1024);
    g_engine->setScreenHeight(768);
    g_engine->setColorDepth(32);
    return 1;
}
```

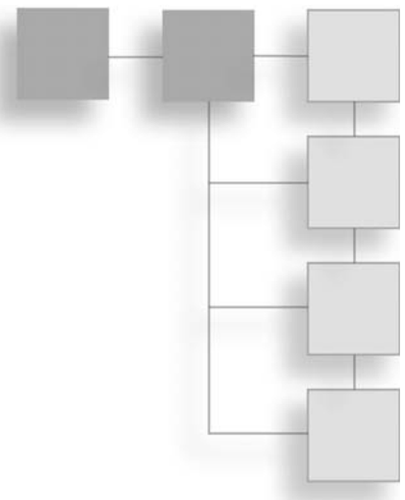
```
bool game_init(HWND)
{
    p = new ParticleEmitter();
    p->loadImage("particle16.tga");
    p->setMax(0);
    p->setAlphaRange(50,200);
    p->setDirection(0);
    p->setSpread(270);
    p->setScale(1.5f);
    p->setLength(2000);

    //load cursor
    cursor = new Sprite();
    cursor->loadImage("particle16.tga");

    return true;
}

void game_update()
{
    p->update();
}

void game_keyPress(int key) { }
void game_keyRelease(int key)
{
    if (key == DIK_ESCAPE) g_engine->Close();
}

void game_mouseButton(int button)
{
    switch(button) {
        case 0: //button 1
            p->setVelocity( (rand() % 10 - 5) / 500.0f );
            p->add();
            break;
    }
}

void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y)
```

```
{
    float fx = (float)x;
    float fy = (float)y;
    cursor->setPosition(fx,fy);
    p->setPosition(fx,fy);
}
void game_mouseWheel(int wheel) { }

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,0));
}

void game_render2d()
{
    p->draw();
    cursor->draw();
}

void game_end()
{
    delete p;
    delete cursor;
}
```

That's it for input. You can load up the current implementation of the Advanced2D engine and build a game with the tools developed so far. But we still have much ground to cover! For instance, we have no way to support sound effects or music yet. Good thing that is coming up in the next chapter.

CHAPTER 6

Audio



Good audio is an absolutely crucial aspect of a successful game. Sound can literally make or break a game! Can you think of a game you've played at one time or another that had such dreadful sound effects or music that you could not continue playing? Conversely, have you ever played a game with such dramatic sound or music that it drew you further into the game than would have otherwise been possible with just the graphics? I remember the first time I played *Halo: Combat Evolved* for the first time in the fall of 2001. Sure, it's part of pop culture now, but back then it was a niche game that was relatively unknown—at least, for the first few weeks! I remember the graphics and incredibly realistic physics, but it was the *music soundtrack* (composed by Martin O'Donnell and Michael Salvatori) and *sound effects* that really sold me on the incredible quality of the game. Even a small game, such as a hobby project, should have carefully chosen or composed audio.

Advice

Killer Tracks is a company that specializes in providing licensable audio for video games. See their offerings at www.killertracks.com.

Designing an Audio System

So, we know that audio is important in our games. The real question is this: How do we write an audio system for a game? I'll bet you were expecting me to bring up DirectSound at about this time. Am I right? On the contrary, DirectSound is a terrible library that cannot even load a WAV file on its own, let alone any other audio format. DirectSound is a very low-level (almost device driver–level) interface to the sound hardware with a mixer. We are not going to waste our time with it. Instead, we're going to use a professional audio system called FMOD.

What Is FMOD?

FMOD is a professional audio engine used in most commercial games today, including those for PC and the major consoles, with support for Nintendo Wii, Microsoft Xbox 360, and Sony PS3. Most surprisingly, FMOD is available *free* for noncommercial use. This is an extraordinarily generous gesture by Firelight Technologies, the developers of FMOD. Beyond the library's pedigree, it is just plain easy to use and it works *great*. The website for FMOD is www.fmod.org, and this is where you will want to visit to download the latest version. The current version of FMOD available at the time of this writing (officially labeled FMOD Ex 4.12) is on the CD-ROM in the \libraries folder. Let's talk about what makes FMOD tick.

FMOD can load and play many different audio files, but the two formats we're concerned with are Windows .wav (WAV) files and Ogg Vorbis .ogg (Ogg) files. You can use WAV files for sound effects and Ogg files just for music, or you can use Ogg files for both sound and music. Due to the compression rate and file size, I recommend against using WAV files for music.

Advice

We can't use MP3 because it's a licensed, proprietary format. To use the MP3 format, you must secure a license! For more information, visit www.mp3licensing.com. Due to this licensing limitation, the alternative Ogg Vorbis format is preferred for distributed games. According to www.vorbis.com, "Ogg Vorbis is a completely open, patent-free, professional audio encoding and streaming technology with all the benefits of Open Source."

If you want to convert audio files to Ogg (for instance, music files), I recommend the free but awesome Audacity sound editing program available at audacity .sourceforge.net. Audacity is not as powerful as a commercial tool, but it gets the job done quickly and easily. Among its many features is the ability to convert MP3s to Ogg files.

Using the FMOD SDK

If you're new to FMOD, I recommend using the version included with the book to ensure compatibility, since a high-quality book like this one will remain on the market for many years. When you are familiar with FMOD, visit Firelight Technologies' website at www.fmod.org and download the latest version.

FMOD is composed of a library file, a DLL, and these header files:

- fmod.h
- fmod.hpp
- fmod_codec.h
- fmod_dsp.h
- fmod_errors.h
- fmod_output.h

If you're writing code in C, then use the fmod.h header, but if you're using C++, use the fmod.hpp header. You only need to include the main header, because it includes the others. An FMOD library file is available for Visual C++ and Dev-C++ and is included in the example project for this chapter (a program called AudioTest that we'll go over later). Here are the library files:

- Visual C++: fmodex_vc.lib
- Dev-C++: libfmodex.a

Of course, these files from the FMOD SDK are only needed to compile your game with FMOD support. Once your game is built, you no longer need these files. Instead, you will need the FMOD runtime file (which is the same for both compilers):

- fmodex.dll

Advice

FMOD cannot be distributed freely unless you abide by the terms of use specified by Firelight Technologies at the www.fmod.org website. If you release a freeware game, you must include Firelight's official copyright notice! I secured permission to include the FMOD SDK files with this book.

The best way to learn how to use the FMOD SDK is to see an example. Instead of a demo program, I'm going to show you a pair of C++ classes that encapsulate FMOD into convenient properties and methods. Don't get me wrong—FMOD is not difficult to use. But rather than going over the initialization and usage of

FMOD and its support functions, we'll instead examine the code used to build the classes.

Audio Classes

As usual when we've come up with reusable code, it needs to be packaged and added to the Advanced2D game engine so that it's available to any game that uses the engine, and the audio system is no exception. What we need to do is write a wrapper for FMOD. Although FMOD already comes with a C++ implementation, we're just using the C function library version of FMOD because it is fully supported on both Visual C++ and Dev-C++ (while the C++ version is not available for GCC at the time of this writing).

But, I have an idea to make the process of loading and playing audio even more interesting. Instead of just loading up a sample and playing it, what if we were to create an audio manager that would make it possible to load up a sample and store it in an `std::vector` by name? It should then be possible to play any previously loaded sample by just using the sample name. This sounds better than having a bunch of global audio sample objects in the game's source code file.

We're going to need two classes to make this work. First, a generic `Sample` class that just wraps up an `FMOD_SOUND` object and an `FMOD_CHANNEL` object for playback, in addition to keeping track of its own name. Next, we'll write an `Audio` class that simplifies the FMOD library. Although FMOD is already easy to use, we want the *game engine* to automatically handle the mundane tasks, such as updating the audio system.

Sample Class

The `Sample` class is not useful without the `Audio` class, so `Sample` is stored in the same class definition (Audio.h) and implementation (Audio.cpp) files with the `Audio` class. Following is the `Sample` class definition. Note the names of the FMOD objects in this listing: `FMOD_SOUND` and `FMOD_CHANNEL`. `FMOD_SOUND` is a sound buffer or sample, while `FMOD_CHANNEL` is a playback channel. Although you may reuse a single channel when playing multiple samples, it is simpler to use one channel per sample in our classes. (Note: The `Sample` class is embedded in the Audio.h and Audio.cpp files along with the `Audio` class.)

```
class Sample
{
private:
    std::string name;
public:
    FMOD_SOUND *sample;
    FMOD_CHANNEL *channel;
    Sample(void);
    ~Sample(void);
    std::string getName() { return name; }
    void setName(std::string value) { name = value; }
};
```

I'm not overly concerned about accessors and mutators in the `Sample` class. It is more convenient to expose the `sample` and `channel` pointers publicly to simplify the code in the `Audio` class.

And now for the `Sample` class implementation, which is stored in the Audio.cpp file with the `Audio` class implementation. The `Sample` class' most important responsibility is to free the memory used by an audio sample using the `FMOD_Sound_Release` function.

```
Sample::Sample()
{
    sample = NULL;
    channel = NULL;
}

Sample::~Sample()
{
    if (sample != NULL) {
        FMOD_Sound_Release(sample);
        sample = NULL;
    }
}
```

Audio Class

The `Audio` class is the real workhorse of the two classes, providing numerous methods to load and play samples. The `Audio` class provides a built-in sound manager that can store samples internally so you need not maintain global sample objects in your game's code listing. You can load and play a sound from the `Audio` class in a "fire and forget" style of programming that is very easy to implement and use. On the other hand, there may be cases when you want to manage your own samples, and the `Audio` class supports that as well, with the ability to both load and play a sample using a `Sample` object that you provide.

There are some interesting methods in this class. You can play and stop a sample by name. (Yes, that's right, using a string rather than an object!) The class also has the ability to stop all samples in the output buffer. During testing, I had a specific but unusual need to stop playback of all sounds except for one, and that spawned the `StopAllExcept()` method! Another interesting method is `FindSample()`, which will search through its internal list of samples (by name) and return a pointer to the `Sample` object. (Just remember that it points to the object in the audio manager, so you should not delete any sample retrieved in this way.)

Now, let's see the definition for the `Audio` class found in Audio.h:

```
class Audio
{
private:
    FMOD_SYSTEM *system;
    typedef std::vector<Sample*> Samples;
    typedef std::vector<Sample*>::iterator Iterator;
    Samples samples;
public:
    Audio();
    ~Audio();
    FMOD_SYSTEM* getSystem() { return system; }
    bool Init();
    void Update(); //must be called once per frame
    bool Load(std::string filename, std::string name);
    Sample* Load(std::string filename);
    bool Play(std::string name);
    bool Play(Sample *sample);
    void Stop(std::string name);
    void StopAll();
    void StopAllExcept(std::string name);
    bool IsPlaying(std::string name);
    bool SampleExists(std::string name);
    Sample *FindSample(std::string name);
};
```

Finally, we come to the implementation of the `Audio` class, which does all of the real work. This class will make it very easy to load and play a sample. There's a lot of C++ Standard Library code in here, so if you are not familiar with it, you might feel a bit lost. We'll be using STL constructs even more in Chapter 7, "Entities," while building an entity manager.

```
Audio::Audio()
{
    system = NULL;
}

Audio::~Audio()
{
    //release all samples
    for (Iterator i = samples.begin(); i != samples.end(); ++i)
    {
        (*i) = NULL;
    }
    FMOD_System_Release(system);
}

bool Audio::Init()
{
    if (FMOD_System_Create(&system) != FMOD_OK) {
        return false;
    }
    if (FMOD_System_Init(system,100,FMOD_INIT_NORMAL,NULL) != FMOD_OK) {
        return false;
    }
    return true;
}

void Audio::Update()
{
    FMOD_System_Update(system);
}

Sample* Audio::Load(std::string filename)
{
    if (filename.length() == 0) return false;

    Sample *sample = new Sample();
    FMOD_RESULT res;
    res = FMOD_System_CreateSound(
        system,              //FMOD system
        filename.c_str(),     //filename
        FMOD_DEFAULT,         //default audio
        NULL,                 //n/a
        &sample->sample);    //pointer to sample
    if (res != FMOD_OK) {
        sample = NULL;
    }
    return sample;
}

bool Audio::Load(std::string filename, std::string name)
{
    if (filename.length() == 0 || name.length() == 0) return false;
    Sample *sample = new Sample();
    sample->setName(name);
    FMOD_RESULT res;
    res = FMOD_System_CreateSound(
        system,               //FMOD system
        filename.c_str(),     //filename
        FMOD_DEFAULT,         //default audio
        NULL,                 //n/a
        &sample->sample);     //pointer to sample
    if (res != FMOD_OK) {
        return false;
    }
    samples.push_back(sample);
    return true;
}

bool Audio::SampleExists(std::string name)
{
    for (Iterator i = samples.begin(); i != samples.end(); ++i)
    {
        if ((*i)->getName() == name) {
            return true;
        }
    }
    return false;
}

bool Audio::IsPlaying(std::string name)
{
    Sample *samp = FindSample(name);
    if (samp == NULL) return false;

    int index;
    FMOD_Channel_GetIndex(samp->channel, &index);
    // FMOD returns 99 if sample is playing, 0 if not
    return (index > 0);
}

Sample *Audio::FindSample(std::string name)
{
    Sample *sample = NULL;
    for (Iterator i = samples.begin(); i != samples.end(); ++i)
    {
        if ((*i)->getName() == name) {
            sample = (*i);
            break;
        }
    }
    return sample;
}


bool Audio::Play(std::string name)
{
    FMOD_RESULT res;
    Sample *sample = FindSample(name);
    if (sample->sample != NULL) {
        //sample found, play it
        res = FMOD_System_PlaySound(
            system,
            FMOD_CHANNEL_FREE,
            sample->sample,
            true,
            &sample->channel);

        if (res!= FMOD_OK) return false;
        FMOD_Channel_SetLoopCount(sample->channel, -1);
        FMOD_Channel_SetPaused(sample->channel, false);
    }
    return true;
}

bool Audio::Play(Sample *sample)
{
    FMOD_RESULT res;
    if (sample == NULL) return false;
    if (sample->sample == NULL) return false;
    res = FMOD_System_PlaySound(
        system,
        FMOD_CHANNEL_FREE,
        sample->sample,
        true,
        &sample->channel);

    if (res!= FMOD_OK) return false;
    FMOD_Channel_SetLoopCount(sample->channel, -1);
    FMOD_Channel_SetPaused(sample->channel, false);
    return true;
}

void Audio::Stop(std::string name)
{
    if (!IsPlaying(name)) return;
    Sample *sample = FindSample(name);
    if (sample = = NULL) return;
    FMOD_Channel_Stop(sample->channel);
}

void Audio::StopAll()
{
    for (Iterator i = samples.begin(); i != samples.end(); ++i)
    {
        FMOD_Channel_Stop( (*i)->channel );
    }
}

void Audio::StopAllExcept(std::string name)
{
    for (Iterator i = samples.begin(); i != samples.end(); ++i)
    {
        if ((*i)->getName() != name) {
            FMOD_Channel_Stop( (*i)->channel );
        }
    }
}
```

Adding FMOD to the Game Engine

The `Audio` class is very easy to use, so we don't need to abstract it any further inside the game engine; it will suffice to just add a public `Audio` object to the engine so it's available via the global `g_engine`. The object will be called audio, and you will be able to access it via g_engine->audio. Let's take a look at the minor changes made to the engine to accommodate the new audio system.

In the Advanced2D.h header, an Audio class instance is defined as public:

```
//simplified public Audio object
Audio *audio;
```

Over in the Advanced2D.cpp engine implementation file, scroll down to the Engine::Init method and you will find the audio initialization:

```
//create audio system
audio = new Audio();
if (!audio->Init()) return 0;
```

Scrolling down a ways in the file, locate the Engine::Update method and the following lines of code:

```
//update audio system
audio->Update();
```

The FMOD system is not multi-threaded, so it does not automatically update the audio stream in the background—we must call a function to give FMOD an opportunity to update audio playback.

The last change made is to the destructor, where we need to wipe the audio object from memory:

```
Engine::~Engine()
{

        audio->StopAll();
        delete audio;
        delete p_input;
        if (this->p_device) this->p_device->Release();
        if (this->p_d3d) this->p_d3d->Release();}
```

Audio Test

"Testing 1, 2, 3 . . . ." What's the point of writing this fancy new audio system when we haven't heard anything out of it yet? Let's take the audio system for a spin. Following is an example program called AudioTest. Since there is nothing displayed by this program, I will skip a screenshot this time. You can use the mouse buttons to play sound effects by clicking the program window.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

//independent sample
Sample *wobble;

bool game_preload()
{
    g_engine->setAppTitle("AUDIO TEST");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(640);
    g_engine->setScreenHeight(480);
    g_engine->setColorDepth(32);
    return true;
}

bool game_init(HWND)
{
    g_engine->message("Press mouse buttons to hear sound clips!");

    //load sample into audio manager
    if (!g_engine->audio->Load("gong.ogg", "gong")) {
        g_engine->message("Error loading gong.ogg");
        return false;
    }

    //load sample into audio manager
    if (!g_engine->audio->Load("explosion.wav", "explosion")) {
        g_engine->message("Error loading explosion.wav");
        return false;
    }

    //load independent sample
    wobble = new Sample();
    wobble = g_engine->audio->Load("wobble.wav");
    if (!wobble) {
        g_engine->message("Error loading wobble.wav");
        return false;
    }

    return true;
}
```

```
void game_keyRelease(int key)
{
    if (key == DIK_ESCAPE) g_engine->Close();
}

void game_mouseButton(int button)
{
    switch(button) {
        case 0:
            //play gong sample stored in audio manager
            g_engine->audio->Play("gong");
            break;
        case 1:
            //play explosion sample stored in audio manager
            g_engine->audio->Play("explosion");
            break;
        case 2:
            //play woggle sample stored independently
            g_engine->audio->Play(wobble);
            break;
    }
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(100,0,0));
}


void game_end()
{
    g_engine->audio->StopAll();
    delete wobble;
}


//unused game events
void game_update() { }
void game_keyPress(int key) { }
void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y) { }
void game_mouseWheel(int wheel) { }
void game_render2d() { }
```

Table 6.1 Linker Options

Visual C++	Dev-C++
Advanced2D.lib	-lAdvanced2D
d3d9.lib	-ld3d9
d3dx9.lib	-ld3dx9
dxguid.lib	-ldxguid
dinput8.lib	-ldinput8
winmm.lib	-lwinmm
fmodex_vc.lib	-lfmodex

To compile the AudioTest program you will need to add the FMOD static library to your project's linker options. At this point in the engine's development, the linker options described in Table 6.1 are required.

That about wraps up the audio system. This has been a pretty fast romp through some heavy code for a third-party library. But, would it surprise you to learn that this is the norm rather than the exception in game development?

CHAPTER 7

Entities

The difference between a real game engine and an SDK wrapper is the level of abstraction evident in the game code. Up to this point, we have merely been writing more convenient C++ classes for key game library components, such as rendering, input, audio, and so forth. Those classes do not make a game engine, they are merely tools. A true engine must *run*, for one thing! Imagine it this way: You have a block, crankshaft, heads, camshafts, pistons, spark plugs, a fuel injection intake, and a throttle body; do these parts individually produce power? An engine performs work. Every component is crucial to the correct running of the engine, but the engine is far more than just the sum of its parts. Let's follow the same analogy when thinking about our game engine, and then work on putting the components together, from individual pieces to a whole machine that can produce work.

An entity can be just about any *thing* you can imagine, but in the context of game development, an entity is usually an instance of an encapsulated system that performs some function. For instance, you might think of a sound effect as an entity, and that might be a valid description, but it doesn't quite fit. I think of a sound effect as a result of some action performed by an entity, not as an entity itself. What types of objects in a game are likely to perform actions or interact in some way? Most likely, only a mesh or a sprite is likely to interact in a game. So, let's imagine that mesh and sprite objects share at least one behavior—they are both *entities* in a game. By sharing basic properties, such as position and velocity, we can manipulate both sprites and meshes using a single call to shared function names. This takes the form of virtual methods in the class definition. Pure virtuals are methods declared with `= 0` in the definition, which is equivalent to setting the function pointer to null. On the technical side of C++, class function names are actually pointers to the shared function code in memory, and a pure virtual means that the subclasses override the base class' function names. I'm using the word "function" to describe the process, while "method" is the proper name for functions defined within a class.

Building an Entity Manager

The entity management system in a game engine shouldn't care what type of object you add to it, as long as that object is derived from a base entity. You should be able to subclass an entity into as many different entity types as you want to use in your game! For our game engine, we'll modify the `Sprite` and `Mesh` classes (created in previous chapters) so they can be used as entities. You probably wouldn't want to treat things such as lights and cameras as entities because those are part of the props and equipment of a game, not entities. But, if you're interested in experimenting, you *could* technically add those types of objects to the entity manager—I'm just not sure how useful that would be.

An entity should provide base properties and methods that will be shared by all entities (regardless of its actual functionality in the game). We want to be able to add an entity by name or identifier number, among other things, and the entity class should provide these facilities.

An entity manager will automatically process the entities and then report the results to the game (or rather, to *you,* the programmer). This will only work if the entities are properly initialized before they are added. The properties will affect how each entity is drawn, moved, animated, and so forth. If we set an entity's properties a certain way, it should automatically move and animate. In the future, we may want to add behavior to game entities so they interact with their environment in an even higher level of automation (which is the subject of A.I.). Before that will be possible, however, the entity manager must be programmed with the basic logistics of managing entities.

The entity manager should make it easy to manipulate entities once they're in the system. We need functionality that makes it possible to add, find, and delete entities from the game code. In the engine itself, we need to automatically move, animate, and draw entities based on their properties. This is the part where game programming really starts to get fun, because at this point we're working at a higher level, more in the realm of designing gameplay than doing low-level stuff like rendering. This automated functionality is possible through the use of the Standard Template Library; specifically, an `std::list`. We could use a `std::vector`, which is faster at consecutive iteration. In other words, when the entity manager goes through the group of entities and processes each one, in a sequential manner, a vector is faster than a list. You may not notice any difference until there are a few tens of thousands of entities, and really it's a matter of preference. In the end, we need to use an `std::list` because it is better at deleting items, which is more challenging with an `std::vector` (which tends to complain quite a bit if you remove an item while it's iterating through its members).

Advice

If your STL knowledge is a bit rusty, I recommend *C++ Standard Library Practical Tips* (Thomson Course Technology PTR, 2005) by Greg Reese.

The Entity Class

Let's start with a new class called `Entity`. This simple class is more of a placeholder with a few minor properties used to identify the type of entity being subclassed. Some of the methods in `Entity` are declared as pure virtual, meaning you *must* subclass `Entity` into a new class; you cannot use an `Entity` alone. The properties are all important and are used by the entity manager to process the entities. Actually, the manager doesn't really care whether your entity is a sprite, a mesh, or a Hobgoblin; it will just process the virtual methods and use the properties you provide it.

Although a strongly typed engine might define specific entity types with an enumeration or some constant values (such as `ObjectType`), I did not want to regulate the engine too much—it's up to *you* to set the properties when you create your entity objects and add them to the manager, and then write the code to respond to the events based on object type. One very interesting property is lifetime (composed of two variables—`lifetimeLength` and `lifetimeTimer`). Using this property, you can set an entity to auto-expire after a fixed amount of time (measured in milliseconds). If you want an entity to participate in the game for only 10 seconds, you can set its lifetime to 10000, and it will be automatically removed when the time expires. This can be extremely handy for many types of games in which you would otherwise have to add logic to terminate things such as bullets and explosions manually.

However, there is one property that we *must* set in order to perform the correct type of rendering, either 2D or 3D. You cannot render 2D and 3D objects together because 2D sprites must be rendered by D3DXSprite within the 3D rendering pipeline. The `Entity` class, defined in a moment, includes an enumeration called `RenderType` that also falls inside the overall Advanced2D namespace (so it's visible to the `Entity` class). We need to use this simple enumeration to determine whether an entity should be rendered in 2D or 3D. That will be done by modifying the `Sprite` and `Mesh` constructors later. The `Entity` class has a constructor with a mandatory parameter that is called by the constructors of `Sprite` and `Mesh` with the appropriate 2D or 3D setting. It's automatic once the classes are defined.

```
#include "Advanced2D.h"
#pragma once
namespace Advanced2D {

    enum RenderType {
        RENDER2D = 0,
        RENDER3D = 1
    };

    class Entity {
    private:
        int id;
        std::string name;
        bool visible;
        bool alive;
        enum RenderType renderType;
        int objectType;
        int lifetimeLength;
        Timer lifetimeTimer;

    public:
        Entity(enum RenderType renderType);
        virtual ~Entity() { };
        virtual void move() = 0;
        virtual void animate() = 0;
        virtual void draw() = 0;
        void setID(int value) { id = value; }
        int getID() { return id; }
        void setRenderType(enum RenderType type) { renderType = type; }
        enum RenderType getRenderType() { return renderType; }
        std::string getName() { return name; }
        void setName(std::string value) { name = value; }
        bool getVisible() { return visible; }
        void setVisible(bool value) { visible = value; }
        bool getAlive() { return alive; }
        void setAlive(bool value) { alive = value; }

        int getLifetime() { return lifetimeLength; }
        void setLifetime(int milliseconds) {
            lifetimeLength = milliseconds; lifetimeTimer.reset();
        }
        bool lifetimeExpired() {
            return lifetimeTimer.stopwatch(lifetimeLength);
        }
        int getObjectType() { return objectType; }
        void setObjectType(int value) { objectType = value; }
    };
};
```

Here is the Entity class implementation. All we need here is the constructor to initialize the property variables; otherwise, the Entity class is mostly made up of accessor and mutator methods in the header. Note that Entity does not have a default constructor, only one with the RenderType parameter. You *must* tell an entity whether it should be rendered in 2D or 3D, and this takes care of that requirement.

```
#include "Advanced2D.h"
namespace Advanced2D {
    Entity::Entity(enum RenderType renderType)
    {
        this->renderType = renderType;
        this->id = -1;
        this->name = "";
        this->visible = true;
        this->alive = true;
        this->objectType = 0;
        this->lifetimeLength = 0;
        this->lifetimeTimer.reset();
    }
};
```

The three pure virtual methods are Entity:move(), Entity::animate(), and Entity::draw(), which means these three must, at minimum, be implemented in a subclass. We'll get to that in a bit, with the `Sprite` and `Mesh` classes. First, we need to make some changes to the core engine.

Modifying the Engine

The engine will need to be modified to support entity management. This is the part where we begin to take all of the components (`Sprite`, `Mesh`, and so on) and begin assembling them into a functional engine. We'll begin with some changes to the `Advanced2D` class with the following new game event functions:

```
extern void game_entityUpdate(Advanced2D::Entity*);
extern void game_entityRender(Advanced2D::Entity*);
```

These functions, which must be present in the game's code file, will receive entity-related events. Specifically, `game_entityUpdate` is called whenever an entity is manipulated in some way (moved, animated, and so on), while `game_entityRender` is called after an entity is rendered. Why would this be necessary, you may be wondering? It's helpful if you want to quickly and easily add something to the rendering pipeline, such as a manually rendered force field or a special effect drawn over a specific entity. There *is* some overhead involved with these function calls. A possible future optimization would be a flag that determines whether these events are called by the engine. After all, if you don't ever plan to use an event, it's wasteful to have it called several thousand times per frame (dependent, of course, on the number of entities in your game).

I've mentioned "entity manager" quite a bit in the chapter so far, but I haven't really explained what it is. The manager is not a class; it's just some new functionality in the engine, in the form of new methods that automatically handle the entities. We need to define the entity list in the Advanced2D.h private section:

```
std::list<Entity*> p_entities;
```

This is template-based code. When the `std::list` class is used to create the instance called `p_entities`, we must tell the container what type of object it will contain. The `std::list` is a container for other objects. When this code is compiled, the C++ compiler creates a new class based on a container of `Sprite` objects.

Also in the private section of the `Engine` class are three management methods used internally by the engine to update, draw, and delete entities.

```
void UpdateEntities();
void DrawEntities();
void BuryEntities();
```

That odd-sounding `BuryEntities` method is actually quite descriptive, because its job is to remove all "dead" entities from the list. But how does an entity die, you wonder? Very simply, by setting its "alive" property to false.

Advice

The magnificent thing about the entity manager is that you can dynamically add new entities to your game, and it then *automatically* updates and draws them. And, if you set the lifetime property, the entity manager will even terminate your game's entities automatically.

Let's jump over to the Advanced2D.cpp class implementation file in order to add the functional code for the entity manager.

New `Engine::Update`

Scrolling down in the Advanced2D.cpp file, locate the `Engine::Update()` method. Here is the complete source code for the method, with the new entity manager code highlighted in bold.

```
void Engine::Update()
{
    static Timer timedUpdate;
    //calculate core framerate
    p_frameCount_core++;
    if (p_coreTimer.stopwatch(999)) {
        p_frameRate_core = p_frameCount_core;
        p_frameCount_core = 0;
    }
    //fast update with no timing
    game_update();

    //update entities
    if (!p_pauseMode) UpdateEntities();

    //update with 60fps timing
    if (!timedUpdate.stopwatch(14)) {
        if (!this->getMaximizeProcessor()) { Sleep(1); }
    }
    else {
        //calculate real framerate
        p_frameCount_real++;
        if (p_realTimer.stopwatch(999)) {
            p_frameRate_real = p_frameCount_real;
            p_frameCount_real = 0;
        }
        //update input devices
        p_input->Update();
        this->UpdateKeyboard();
        this->UpdateMouse();
        //update audio system
        audio->Update();
        //begin rendering
        this->RenderStart();
        game_render3d();
        //render 3D entities
        if (!p_pauseMode) Draw3DEntities();
        //render 2D entities
        Render2D_Start();
        game_render2d();
        //render 2D entities
        if (!p_pauseMode) Draw2DEntities();
        //done rendering
        Render2D_Stop();
        this->RenderStop();
    }
    //remove dead entities from the list
    BuryEntities();
}
```

Engine::UpdateEntities

The `UpdateEntities` method is called from `Engine::Update` to process everything in the entity list. *Process* here means to move, animate, and check the lifetime of each entity, and call the game event functions for each entity that is updated (but rendering is done elsewhere). If you want to add functionality to the entity manager, this is where you will want to do that because this code runs at the core clock speed—not the slow framerate speed. This is where we will add some physics code in the near future.

```
void Engine::UpdateEntities()
{
    std::list<Entity*>::iterator iter;
    Entity *entity;
    iter = p_entities.begin();
    while (iter != p_entities.end())
    {
        //point local sprite to object in the list
        entity = *iter;

        //is this entity alive?
        if ( entity->getAlive() ) {
            //move/animate entity
            entity->move();
            entity->animate();

            //tell game that this entity has been updated
            game_entityUpdate( entity );

            //see if this entity will auto-expire
            if ( entity->getLifetime() > 0)
            {
                if ( entity->lifetimeExpired() ) {
                    entity->setAlive(false);
                }
            }
        }
        ++iter;
    }
}
```

`Engine::Draw3DEntities`

The `Engine::Draw3DEntities` method is called from `Engine::Update` to process all 3D entities (if any). The entire entity list is iterated through; any entities with a `RenderType` of `RENDER3D` have their `draw()` method called. Any other entities are ignored.

```
void Engine::Draw3DEntities()
{
    Entity *entity;
    std::list<Entity*>::iterator iter = p_entities.begin();
    while (iter != p_entities.end())
     {
        //temporary pointer
        entity = *iter;
        //is this a 3D entity?
        if ( entity->getRenderType() == RENDER3D ) {
            //is this entity in use?
            if ( entity->getAlive() && entity->getVisible() ) {
                entity->draw();
                game_entityRender( entity );
            }
        }
        ++iter;
    }
}
```

Advice

Do your instincts tell you that it's wasteful to iterate through the entire entity list *twice* to process the 3D and 2D entities separately? That means you are anticipating how the machine will run your code, which is a good thing. However, processors are extremely good at doing loops today, with their multiple pipeline architectures and cache memory, so don't worry about duplicating loops for different processes. In the end, the only code that takes clock cycles is the code in called functions, while the code in the loop is pipelined and probably would not even show up in profiling. As it turns out, we cannot combine these loops anyway because the 2D and 3D rendering must be done at different times.

Engine::Draw2DEntities

Like the `Draw3DEntities` method, `Draw2DEntities` also iterates through the entity list and picks out objects with a `RenderType` of `RENDER2D` and calls the `draw()` method for each one.

```
void Engine::Draw2DEntities()
{
    Entity *entity;
    std::list<Entity*>::iterator iter = p_entities.begin();
    while (iter != p_entities.end()) {
        //temporary pointer
        entity = *iter;
        //is this a 2D entity?
        if ( entity->getRenderType() == RENDER2D ) {
            //is this entity in use?
            if ( entity->getAlive() && entity->getVisible() ) {
                entity->draw();
                game_entityRender( entity );
            }
        }
```

```
            ++iter;
        }
}
```

Advice

Don't worry about slowing down your game by drawing too many sprites, because Direct3D *batches* sprite rendering and does it extremely quickly in the 3D hardware. What you *should* be concerned with is code that updates sprites that may slow down the game.

Engine::BuryEntities

The last of the private entity manager support methods is BuryEntities. This method iterates through the entity list (p_entities), looking for any objects that are "dead" (where the alive property is false). Thus, to delete an object from the entity manger, just call setAlive(false), and it will be removed at the end of the frame update loop. Although you will create a new entity on the heap (with new) and then add it to the entity manager, you will not need to remove entities because the list::erase method automatically calls delete for each object as it is destroyed. As a result, we can use a "fire and forget" policy with our entities and trust that the container is cleaning up afterward.

```
void Engine::BuryEntities()
{
    std::list<Entity*>::iterator iter = p_entities.begin();
    while (iter != p_entities.end()) {
        if ( (*iter)->getAlive() == false ) {
            iter = p_entities.erase( iter );
        }
        else iter++;
    }
}
```

Now that the entity manager has been added to the engine for internal processing, we need to add the public access methods that allow the game to access the entity manager. Following are methods for adding and locating entities by either object type or by name. If you want to truly give entities their own unique identifier, you must be sure to assign each one a distinct object type number in order to locate it later (if that's even necessary). You may also locate entities by name, but again it's up to you to give each one a unique name. We have no way to return multiple entities by object type or name (although perhaps a FindNext method would be helpful...).

`Engine::addEntity`

The `Engine::addEntity` method is used by the game to add an entity to the manager. First, you must create a new object from a class derived from `Entity`, instantiate the class, set its properties, and then add it to the list.

```
void Engine::addEntity(Entity *entity)
{
        static int id = 0;
        entity->setID(id);
        p_entities.push_back(entity);
        id++;
    }
```

`Engine::findEntity`

There are two `Engine::findEntity` methods available for searching the entity list. The first one searches by object type (via an integer parameter) and is user-definable, so this is a property that you must set in your game object if you want to search for it by object type. Every entity *must* have some form of identification, or you will not be able to respond to it in your game's update event (for instance, to detect when a bullet hits an enemy ship). The entity manager automatically assigns a sequential ID value to each new entity that you may read with the `getID()` method, but this does not help identify the *type* of entity unless you manually set it with `Entity::setObjectType()`.

```
Entity *Engine::findEntity(int objectType)
{
    std::list<Entity*>::iterator i = p_entities.begin();
    while (i != p_entities.end())
    {
        if ((*i)->getAlive()==true && (*i)->getObjectType()==objectType)
            return *i;
        else ++i;
    }
    return NULL;
}
```

The second form of `Engine::findEntity` searches by name and is also based on the name property that you set in the object prior to adding it to the entity manager.

```
Entity *Engine::findEntity(std::string name)
{
    std::list<Entity*>::iterator i = p_entities.begin();
    while (i != p_entities.end()) {
        if ( (*i)->getAlive() == true && (*i)->getName() == name )
            return *i;
        else ++i;
    }
    return NULL;
}
```

Modifying the Sprite Class

Let's see what must be changed in the `Sprite` class to support the entity manager. As it turns out, we only need to change the class name definition by adding `Entity` as the parent class, and no other changes are needed to either the definition or the implementation file.

```
class Sprite : public Entity {
    //sprite class code omitted
};
```

Testing Sprites as Entities

Now let's see how the entity management system works in a real example. On the CD is a project called SpriteEntityDemo that you may open and run. Figure 7.1 shows the output from the program.

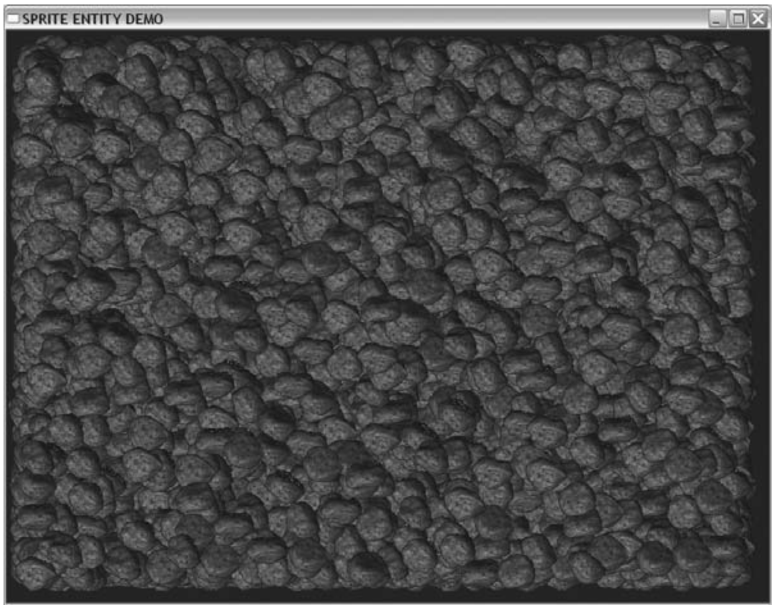


Figure 7.1
This program automatically moves, animates, and renders 10,000 sprite entities.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

#define OBJECT_SPRITE 100
#define MAX 10000

Texture *image;

bool game_preload()
{
    g_engine->setAppTitle("SPRITE ENTITY DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(1024);
    g_engine->setScreenHeight(768);
    g_engine->setColorDepth(32);
    return 1;
}

bool game_init(HWND)
{
    Sprite *asteroid;
    image = new Texture();
    image->Load("asteroid.tga");
    for (int n=0; n < MAX; n++) {
        //create a new asteroid sprite
        asteroid = new Sprite();
        asteroid->setObjectType(OBJECT_SPRITE);
        asteroid->setImage(image);
        asteroid->setTotalFrames(64);
        asteroid->setColumns(8);
        asteroid->setSize(60,60);
        asteroid->setPosition( rand() % 950, rand() % 700 );
        asteroid->setFrameTimer( rand() % 100 );
        asteroid->setCurrentFrame( rand() % 64 );
        if (rand()%2==0) asteroid->setAnimationDirection(-1);
        //add sprite to the entity manager
        g_engine->addEntity(asteroid);
    }
    std::ostringstream s;
    s << "Entities: " << g_engine->getEntityCount();
    g_engine->message(s.str());
    return true;
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));
}

void game_keyRelease(int key)
{
    //exit when escape key is pressed
    if (key == DIK_ESCAPE) g_engine->Close();
}

void game_entityUpdate(Advanced2D::Entity* entity)
{
    //type-cast Entity to a Sprite
    Sprite* sprite = (Sprite*)entity;

    //this is where you can update sprite properties
}

void game_entityRender(Advanced2D::Entity* entity)
{
    //engine automatically renders each entity
    //but we can respond to each render event here
}

void game_end()
{
    delete image;
}

void game_update() { }
void game_render2d() { }
void game_keyPress(int key) { }
void game_mouseButton(int button) { }
void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y) { }
void game_mouseWheel(int wheel) { }
```

Advice

If you see a linker warning that refers to "uuid.lib", you may ignore it. This is an unavoidable warning caused by a "#pragma comment" statement in one of the DirectX header files. The use of `#pragma` to add a library reference to a project is a bad coding practice because it bypasses the normal `makefile`-based build process.

Modifying the Mesh Class

The `Mesh` class was first introduced way back in Chapter 2 and provided basic 3D rendering support to the engine (a feature that will get little use in actual practice, since we're focusing our attention on 2D games). Here is the only change that is needed to bring the `Mesh` class into the `Entity` family.

```
class Mesh : public Entity
{
    //entity class code omitted
};
```

Testing Meshes as Entities

Let's see whether the entity manager can handle `Mesh` objects. I'm using a high-poly mesh (the cytovirus.x file from the DirectX SDK examples), so the load time is lengthy if you increase the number of entities in this demo. To enable the program to start up in a reasonable time, it defaults to only 10 mesh objects. (You could optionally use a low-poly mesh such as the ball.x file featured in the BouncingBalls demo back in Chapter 2.) In addition to demonstrating how mesh entities are handled, this program also highlights the weaknesses in the 3D rendering portion of the engine.

Let's face it, we're building a 2D engine here, we have ignored advanced 3D rendering and optimization issues completely, and we are rendering each mesh subset individually. Since Direct3D is a state-based renderer, it must change its state every time a new mesh subset is rendered, which is extremely slow. This engine *needs* a vertex buffer—badly. But we aren't going to create one. There are plenty of books about 3D engine development, but this is not one of them, and it does not pretend to be.

Figure 7.2 shows the output from the MeshEntityDemo program. The entity-based code in the following listing is highlighted in bold.

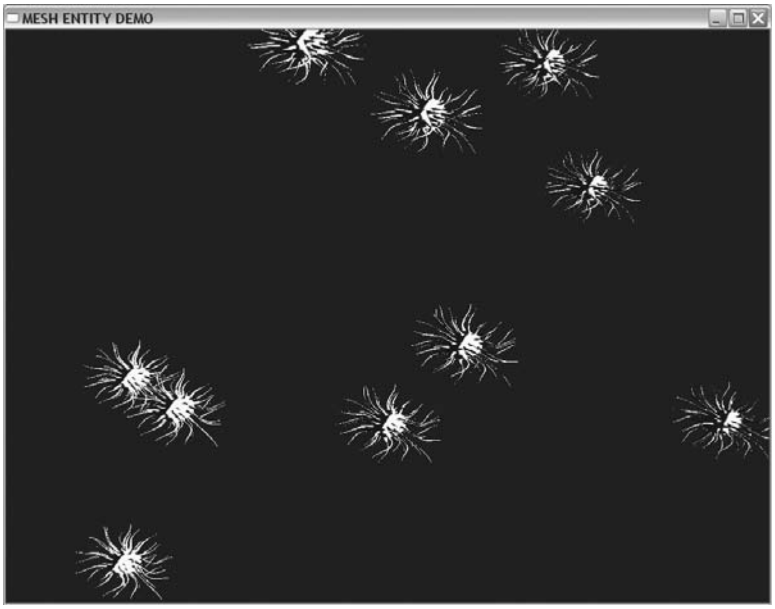


Figure 7.2
This program automatically updates and renders mesh entities.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;
#define MAX 10
Camera *camera;
Light *light;

bool game_preload()
{
    g_engine->setAppTitle("MESH ENTITY DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(1024);
    g_engine->setScreenHeight(768);
    g_engine->setColorDepth(32);
    return 1;
}

bool game_init(HWND)
{
    //set the camera and perspective
    camera = new Camera();
    camera->setPosition(0.0f, 2.0f, 50.0f);
    camera->setTarget(0.0f, 0.0f, 0.0f);
    camera->Update();

    //create a directional light
    D3DXVECTOR3 pos(0.0f,0.0f,0.0f);
    D3DXVECTOR3 dir(1.0f,0.0f,0.0f);
    light = new Light(0, D3DLIGHT_DIRECTIONAL, pos, dir, 100);
    light->setColor(D3DXCOLOR(1,0,0,0));
    g_engine->SetAmbient(D3DCOLOR_RGBA(0,0,0,0));

    //load meshes
    Mesh *mesh;
    for (int n=0; n<MAX; n++) {
        mesh = new Mesh();
        mesh->Load("cytovirus.x");
        mesh->SetScale(0.02f,0.02f,0.02f);
        float x = rand() % 40 - 20;
        float y = rand() % 40 - 20;
        float z = rand() % 10 - 5;
        mesh->SetPosition(x,y,z);
        //add mesh to entity manager
        g_engine->addEntity(mesh);
    }

    return 1;
}

void game_update()
{
    //nothing to update!
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_RGBA(0,0,60,0));
    g_engine->SetIdentity();
}

void game_keyRelease(int key)
{
    if (key == DIK_ESCAPE) g_engine->Close();
}
```

```
void game_entityUpdate(Advanced2D::Entity* entity)
{
    if (entity->getRenderType() == RENDER3D) {
        //type-cast Entity to a Mesh
        Mesh* mesh = (Mesh*)entity;
        //perform a simple rotation
        mesh->Rotate(0,0.2f,0);
    }
}

void game_entityRender(Advanced2D::Entity* entity)
{
    //type-cast Entity to a Mesh
    Mesh* mesh = (Mesh*)entity;

    //engine automatically renders each entity
    //but we can respond to each render event here
}

void game_end()
{
    delete camera;
    delete light;
}

void game_render2d() { }
void game_keyPress(int key) { }
void game_mouseButton(int button) { }
void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y) { }
void game_mouseWheel(int wheel) { }
```

That wraps up entity management, at least for the time being. We'll come back to the subject again two chapters from now, when we add physics-related features to the engine. But first, let's spend some more time in sprite animation code in the next chapter and build a font system. Not only will that permit text output in a game, but more importantly, it will let us print out debugging info on the screen.

CHAPTER 8

Fonts

One of the most crucial features of a game engine is the ability to display text on the screen, also called *font output*. This is a challenging problem because font output has the potential to bring a game engine to its knees if it is not implemented properly. The font system included with the DirectX SDK (CD3DXFont) is a good example of how *not* to render text, because it is very slow! Why is it slow? Because CD3DXFont renders text to a scratch texture and then blits the image to the screen, and that rendering is done with Windows GDI functions. All of the professional game engines use what is known as *bitmapped fonts*. A bitmapped font is a font stored on a bitmap. (How witty is that?) Figure 8.1 shows a bitmapped font.

Rendering a bitmapped font is extremely fast because we can just use our `Sprite` class to render text! As you learned earlier, D3DXSprite batches sprite drawing so that it is extremely efficient in the 3D hardware.

Creating a Font

I've included several bitmapped fonts on the CD for your use. These fonts were created with a very useful tool called Bitmap Font Builder by Thom Wetzel, Jr. (www.lmnopc.com), which is included on the CD. You can use Bitmap Font Builder (shown in Figure 8.2) to create a bitmapped font from any TrueType font installed on your Windows system. The font shown in the figure is 10-point Verdana.

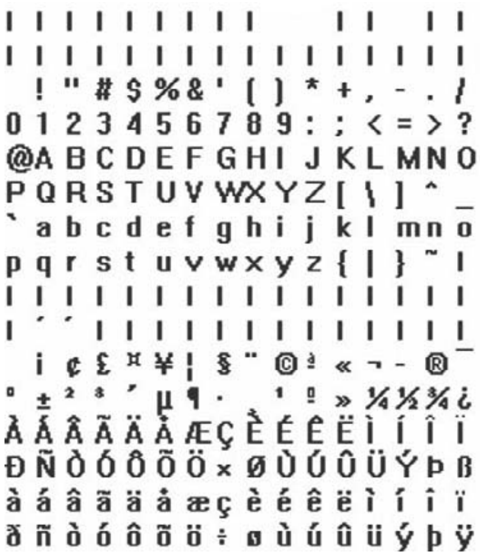


Figure 8.1
This is the System 12-point font in ASCII order.

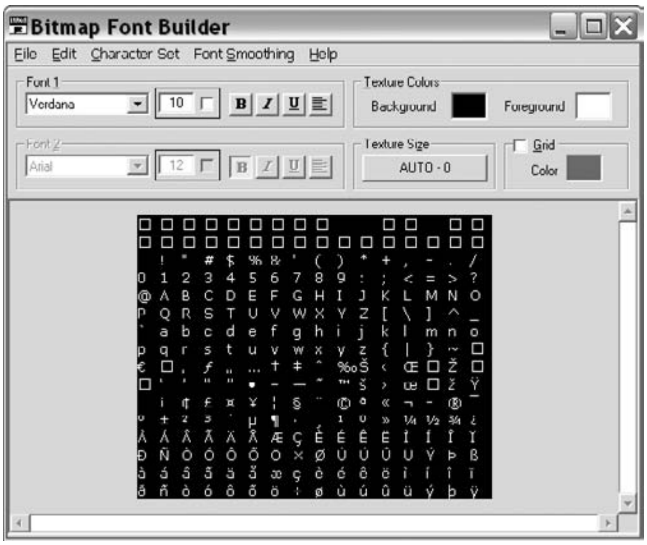


Figure 8.2
Bitmap Font Builder is used to render a TrueType font onto a bitmap.

The settings are important. I recommend setting the Texture Size field to Auto with 0-pixel spacing for best results. If the Character Set is configured to render two fonts, change it using the menu to a single ASCII font, as shown in the figure. Although you will never use most of those unusual ASCII characters, you never

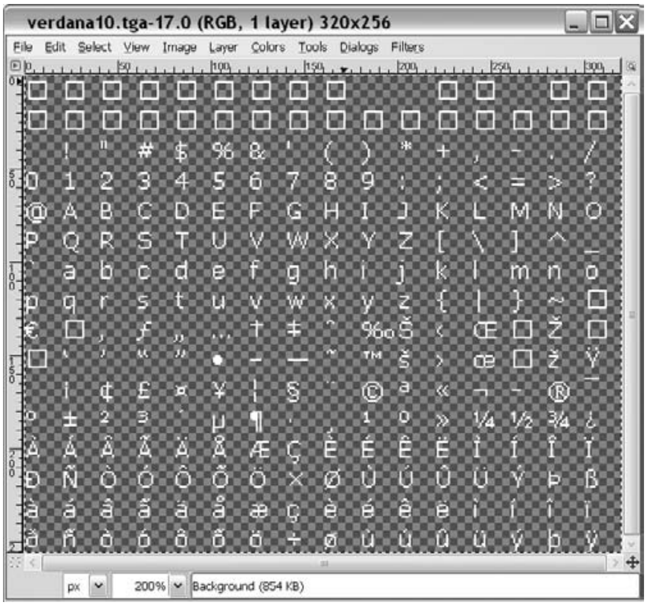

verdana10.tga-17.0 (RGB, 1 layer) 320x256
File Edit Select View Image Layer Colors Tools Dialogs Filters
px
200%
Background (854 KB)

Figure 8.3
Bitmap Font Builder automatically generates an alpha channel for transparency.

know, and the code for rendering the font is simpler when you are using a font with characters numbered 0–255.

When you have configured the font you want to produce, open the File menu, choose Save 32-bit TGA (RGBA), and enter a filename. This will save a new 32-bit Targa file with an alpha channel. Saving the 10-point Verdana font produces a Targa file shown in Figure 8.3. You can experiment with different fonts to come up with one you like for your games. When you are setting up a font, note that it will look sharper in your game than it looks in the BFB preview; although you may be tempted to output a font in bold, that usually is not needed.

After you have saved the font to a Targa image, you will need to export the font width data, which will be used to render the font proportionally. BFB makes this very easy by exporting the width data into a simple binary data file that you can read and use when rendering a font (using an animated sprite).

Open the File menu in BFB and choose Save Font Widths (Byte Format). You will be prompted for a filename. I find it makes sense to use the same filename that I used for the font, but append a .dat extension. This data file will be composed of 256 font width values stored in binary format for a total of 512 bytes (two bytes per ASCII character width).

Loading and Rendering a Font

You *could* load a bitmapped font into a Sprite object and render it by treating each character as a frame in the font "animation" sheet. In fact, this is exactly what we will do. But there is too much configuration and custom code to be duplicated that way. Instead, a new subclass of Sprite will do nicely. The new class will be called Font and will inherit its basic functionality from Sprite and add some of its own new features.

Font Class

Let's take a look at the new Font class, which is now available in the Engine project on the CD (under this chapter's folder). Here's the header file:

```
#include "Advanced2D.h"
#pragma once
namespace Advanced2D {
    class Font : public Sprite {
    private:
        int widths[256];
    public:
        Font();
        virtual ~Font(void) { }
        void Print(int x,int y,std::string text,int color = 0xFFFFFFFF);
        int getCharWidth() { return this->width; }
        int getCharHeight() { return this->height; }
        void setCharWidth(int width) { this->width = width; }
        void setCharSize(int width, int height) {
            setCharWidth( width );
            this->height = height;
        }
        bool loadWidthData(std::string filename);
    };
};
```

Now let's take a look at the implementation file Font.cpp. There are just two methods in the implementation file, with the most important method being Print, which actually displays text on the screen. The Print method accepts four parameters that are self-explanatory: x, y, text, and color. The code in Print goes through each character of the string and prints out a character from the font image based on the ASCII code of the character (from 0 to 255). This is *very easy* by just setting the sprite's current frame to the ASCII code! When that's done, presto—the character corresponding to that "animation frame" will be rendered. Furthermore, because BFB saved the Targa with an alpha channel, we have automatic transparency support built in.

The second method, aside from the constructor, loads the proportional font width data. An `std::ifstream` reads 512 bytes at once and then copies out the width data from every other byte in the buffer. The end result is an array called `widths` that contains custom proportional values for each character in the bitmapped font.

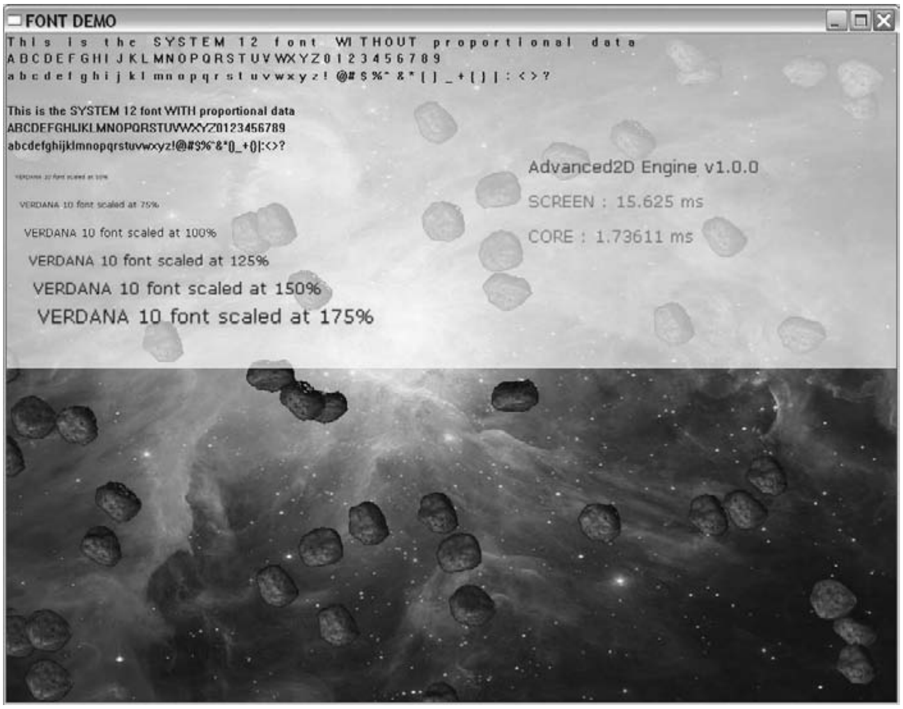
```
#include "Advanced2D.h"
namespace Advanced2D {
    Font::Font() : Sprite()
    {
        //set character widths to default
        memset(&widths, 0, sizeof(widths));
    }

    void Font::Print(int x, int y, std::string text, int color)
    {
        float fx = (float)x;
        float fy = (float)y;

        //set font color
        this->setColor( color );

        //draw each character of the string
        for (unsigned int n=0; n<text.length(); n++) {
            int frame = (int)text[n];
            this->setCurrentFrame( frame );
            this->setX( fx );
            this->setY( fy );
            this->draw();
            //use proportional width if available
            if (widths[frame] == 0) widths[frame] = this->width;
            fx += widths[frame] * this->scaling;
        }
    }

    bool Font::loadWidthData(std::string filename)
    {
        unsigned char buffer[512];
        //open font width data file
        std::ifstream infile;
        infile.open(filename.c_str(), std::ios::binary);
        if (!infile) return false;

        //read 512 bytes (2 bytes per character)
        infile.read( (char *)(&buffer), 512 );
        if (infile.bad()) return false;
        infile.close();

        //convert raw data to proportional width data
        for (int n=0; n<256; n++) {
            widths[n] = (int)buffer[n*2];
        }
        return true;
    }
};
```

Using the New Font Class

We now have a multipurpose bitmapped font class that can load and render proportional fonts, so let's put it to the test and see how fast it is! Figure 8.4 shows the FontDemo program included on the CD. This program demonstrates proportional as well as non-proportioned font output to show the difference, which is significant. Without the font width data, we would have to condense the font by hard-coding the width data inside the program because non-proportioned text just looks too unprofessional. As you can see from the figure, we can display any TrueType font once it has been converted using a tool such as BFB. As a bonus, we have all of the features of the sprite renderer available, too. That means you can print text in any color with rotation and scaling support.

Advice

Direct3D colors can be a bit confusing because there are many support functions and macros defined by the SDK to assist with color conversion. I find it easier at times to just define a color using a hexadecimal code. The format is 0xAARRGGBB, where AA is the 16-bit alpha component, RR is the 16-bit red component, and likewise for green and blue. White can be coded as 0xFFFFFFFF (note that the first two "FF" characters are alpha). If you want to draw a sprite with 50-percent transparency, the color would be 0x99FFFFFF. If you want to create a simpler RGB color, you may use one of the macros, such as D3DCOLOR_XRGB(r,g,b).



```
FONT DEMO
This is the SYSTEM 12 font WITHOUT proportional data
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
abcdefghijklmnopqrstuvwxyz!@#$%^&*[]_+{}|:<>?

This is the SYSTEM 12 font WITH proportional data
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
abcdefghijklmnopqrstuvwxyz!@#$%^&*()_+{}|:<>?

VERDANA 10 font scaled at 50%
VERDANA 10 font scaled at 75%
VERDANA 10 font scaled at 100%
VERDANA 10 font scaled at 125%
VERDANA 10 font scaled at 150%
VERDANA 10 font scaled at 175%

Advanced2D Engine v1.0.0
SCREEN : 15.625 ms
CORE : 1.73611 ms
```

Figure 8.4
The FontDemo program demonstrates proportional bitmapped font rendering.

Here is the listing for the FontDemo program. It's quite lengthy compared to most of the examples we've seen so far! This is due to the amount of output in this demo, not a result of the `Font` class being difficult to configure. In addition, this program is paving the way for a console that will be added to the game engine soon. As you saw in the last figure, the font output was being rendered on top of a transparent panel overlay, with a background and moving sprites underneath. This panel is hard-coded into the FontDemo program at this point as a proof of concept, but it will soon find its way into the engine as a reusable component. So, here is the listing. I recommend you open and run this program now because it looks very cool.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

#define SCREENW 1024
#define SCREENH 768
#define OBJECT_BACKGROUND 1
#define OBJECT_SPRITE 100
#define MAX 50
```

```
Sprite *panel;
Texture *asteroid_image;
Font *system12;
Font *nonprop;
Font *verdana10;

bool game_preload()
{
    g_engine->setAppTitle("FONT DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(SCREENW);
    g_engine->setScreenHeight(SCREENH);
    g_engine->setColorDepth(32);
    return 1;
}

bool game_init(HWND)
{
    //load background image
    Sprite *background = new Sprite();
    background->loadImage("orion.bmp");
    background->setObjectType(999);
    g_engine->addEntity(background);

    //load asteroid image
    asteroid_image = new Texture();
    asteroid_image->Load("asteroid.tga");

    //create asteroid sprites
    Sprite *asteroid;
    for (int n=0; n < MAX; n++)
    {
        //create a new asteroid sprite
        asteroid = new Sprite();
        asteroid->setObjectType(OBJECT_SPRITE);
        asteroid->setImage(asteroid_image);
        asteroid->setTotalFrames(64);
        asteroid->setColumns(8);
        asteroid->setSize(60,60);
        asteroid->setPosition( rand() % SCREENW, rand() % SCREENH );
        asteroid->setFrameTimer( rand() % 100 );
        asteroid->setCurrentFrame( rand() % 64 );
        if (rand()%2==0) asteroid->setAnimationDirection(-1);
        asteroid->setVelocity( (float)(rand()%10)/10.0f, (float)(rand()%10)/10.0f );
        //add asteroid to the entity manager
        g_engine->addEntity(asteroid);
    }

    //load the panel
    panel = new Sprite();
    panel->loadImage("panel.tga");
    float scale = SCREENW / 640.0f;
    panel->setScale(scale);
    panel->setColor(0xBBFFFFFF);

    //load the System12 font
    system12 = new Font();
    if (!system12->loadImage("system12.tga")) {
        g_engine->message("Error loading system12.tga");
        return false;
    }
    system12->setColumns(16);
    system12->setCharSize(14,16);
    if (!system12->loadWidthData("system12.dat")) {
        g_engine->message("Error loading system12.dat");
        return false;
    }

    //load System12 without proportional data
    nonprop = new Font();
    nonprop->loadImage("system12.tga");
    nonprop->setColumns(16);
    nonprop->setCharSize(14,16);

    //load the Verdana12 font
    verdana10 = new Font();
    if (!verdana10->loadImage("verdana10.tga")) {
        g_engine->message("Error loading verdana10.tga");
        return false;
    }
    verdana10->setColumns(16);
    verdana10->setCharSize(20,16);
    if (!verdana10->loadWidthData("verdana10.dat")) {
        g_engine->message("Error loading verdana10.dat");
        return false;
    }

    return true;
}


void game_update() { }
void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));
}

void game_render2d()
{
    std::ostringstream os;
    std::string str;

    panel->draw();

    nonprop->Print(1,1,
        "This is the SYSTEM 12 font WITHOUT proportional data",
        0xFF111111);
    nonprop->Print(1,20,
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789",
        0xFF111111);
    nonprop->Print(1,40,
        "abcdefghijklmnopqrstuvwxyz!@#$%^&*()_+{}|:<>?",
        0xFF111111);

    system12->setScale(1.0f);
    system12->Print(1,80,
        "This is the SYSTEM 12 font WITH proportional data",
        0xFF111111);
    system12->Print(1,100,
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789",
        0xFF111111);
    system12->Print(1,120,
        "abcdefghijklmnopqrstuvwxyz!@#$%^&*()_+{}|:<>?",
        0xFF111111);
    for (float s=0.5f; s<2.0f; s+=0.25f) {
        verdana10->setScale( s );
        int x = (int)(s * 20);
        int y = (int)(100 + s * 120);
        os.str("");
        os << "VERDANA 10 font scaled at " << s*100 << "%";
        verdana10->Print(x,y, os.str(), 0xFF111111);
    }

    verdana10->setScale( 1.5f );
    verdana10->Print(600,140, g_engine->getVersionText(), 0xFF991111);

    os.str("");
    os << "SCREEN : " << (float)(1000.0f/g_engine->getFrameRate_real()) << " ms";
    verdana10->Print(600,180, os.str(), 0xFF119911);

    os.str("");
    os << "CORE : " << (float)(1000.0f/g_engine->getFrameRate_core()) << " ms";
    verdana10->Print(600,220,os.str(),0xFF119911);
}

void game_entityUpdate(Advanced2D::Entity* entity)
{
    switch(entity->getObjectType())
    {
        case OBJECT_SPRITE:
            Sprite* spr = (Sprite*)entity;
            if (spr->getX() < -60) spr->setX(SCREENW);
            if (spr->getX() > SCREENW) spr->setX(-60);
            if (spr->getY() < -60) spr->setY(SCREENH);
            if (spr->getY() > SCREENH) spr->setY(-60);
            break;
    }
}


void game_keyRelease(int key)
{
    if (key == DIK_ESCAPE) g_engine->Close();
}

void game_end()
{
    delete panel;
    delete asteroid_image;
    delete system12;
    delete nonprop;
    delete verdana10;
}

void game_keyPress(int key) { }
void game_mouseButton(int button) { }
void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y) { }
void game_mouseWheel(int wheel) { }
void game_entityRender(Advanced2D::Entity* entity) { }
```

The background used in this chapter should be credited. The Orion nebula was technically created by God, but the photo was snapped by the Hubble Space Telescope, courtesy of NASA and our tax dollars. However, I've looked and looked and have never seen anything other than white dots in the sky. It must be up there somewhere—HST was tax revenue well spent!

Chapter 9

Physics

This chapter covers some basic physics features that will improve the capabilities of the game engine. We will explore two different ways to detect collisions between entities. Real games have sprites that interact, with bullets and missiles that hit enemy ships and cause them to explode, sprites that must navigate a maze without going through walls, and sprites that can run and jump over crates and land on top of enemy characters (such as how Mario jumps onto turtles in *Super Mario World* to knock them out). All of these situations require the ability to detect when two sprites have collided, or touched each other. Sprite collision opens up the world of game programming and makes it possible for you to build a real game! The key to collision testing is to identify where two sprites are on the screen, and then compare their bounding rectangles. That is why this type of collision testing is called *bounding rectangle collision detection*. We will also consider circular collision based on the distance between two entities.

Collision Detection

The only real "physics" we're going to deal with in this chapter concerns the detection of collisions between entities and the response to those collision events. The two types of collision testing we will utilize are *bounding rectangle* and *distance*.

If you know the location of two sprites and you know the width and height of each, then it is possible to determine whether the two sprites are intersecting. Bounding rectangle collision detection describes the use of a sprite's boundary for collision testing. You can get the upper-left corner of a sprite by merely looking at its X and Y values. To get the lower-right corner, add the width and height to the X and Y values. Collectively, these values may be represented as *left, top, right,* and *bottom.*

Automated Collision Detection

The game engine is capable of handling collision detection automatically using its internal entity list. What we want the engine to do is automatically perform collision detection, but then *notify* the game when a collision occurs. In the Advanced2D.h file is a new external function definition:

```
extern void game_entityCollision(Advanced2D::Entity*,Advanced2D::Entity*);
```

The `game_entityCollision` function will be called (and is therefore required) in your game's source code file. Also in the engine header file is the definition of four new support functions used internally by the engine to perform collision testing:

```
bool collision(Sprite *sprite1, Sprite *sprite2);
bool collisionBR(Sprite *sprite1, Sprite *sprite2);
bool collisionD(Sprite *sprite1, Sprite *sprite2);
void TestForCollisions();
```

Advice

Although we have an opportunity to support collision with other types of entities, the code here is written specifically for `Sprite` entities. If you want to support collision detection with other types of entities, you can duplicate this code and adapt it.

Over in the engine implementation file, Advanced2D.cpp, the `Engine::Update` method now includes the call to `TestForCollisions`. (You can ignore the timer code; we'll go over it later in the chapter.)

```
void Engine::Update()
{
    //calculate core framerate
    p_frameCount_core++;
    if (p_coreTimer.stopwatch(999)) {
        p_frameRate_core = p_frameCount_core;
        p_frameCount_core = 0;
    }
    //fast update with no timing
    game_update();

    //update entities
    if (!p_pauseMode) UpdateEntities();

    //perform global collision testing
    if (!p_pauseMode && collisionTimer.stopwatch(50)) {
    TestForCollisions();
}
```

Just as the `UpdateEntities` call includes logic to support pausing the game, so too is there logic for pausing the game included when calling `TestForCollisions`. Also in this logic is a timer that limits the collision detection to 20 Hz (every 50 ms). This is a *critical* section of code in the engine! Collision testing is a time-consuming process. If you allow it to run all out without a timer slowing it down, it will *kill* your engine's performance (by a factor of one hundred or worse—that's right, one percent of its full potential). Timing is *very* important in the game loop. An engine might be capable of awesome performance and frame rates, but one little mistake in timing could give one the impression that the engine isn't very good. So, be careful with such details!

Now, down a bit further in the Advanced2D.cpp file, we will find the newly added `TestForCollisions` method. This rather complex function goes through the entity list and performs several conditional tests before actually calling on the `collision` support function to perform a collision test. First, the `RenderType` of the entity is tested because we are currently only concerned with collisions between sprites, not meshes (which is an entirely different process not particularly suited to 2D games). When the entity has been verified to be a sprite, then its *alive, visible,* and *collidable* properties are examined—and the sprite is skipped if any of them are false. If a sprite jumps all of these hurdles, then it becomes the focus of attention for a while—as all *other* sprites are compared to *this* sprite to determine whether a collision has occurred. For every *other* sprite in the list, the same set of comparisons is made. Although it seems that this is a lot of logic that only slows down the collision process, note that Boolean logic is optimized by the compiler *and* Boolean logic is very predictable—so it will be highly pipelined in the processor.

```
void Engine::TestForCollisions()
{
    std::list<Entity*>::iterator first;
    std::list<Entity*>::iterator second;
    Sprite *sprite1;
    Sprite *sprite2;

    first = p_entities.begin();
    while (first != p_entities.end() )
    {
        //we only care about sprite collisions
        if ( (*first)->getRenderType() == RENDER2D )
        {
            //point local sprite to sprite contained in the list
            sprite1 = (Sprite*) *first;

            //if this entity is alive and visible...
            if ( sprite1->getAlive() && sprite1->getVisible() && sprite1->
isCollidable() )
            {
                //test all other entities for collision
                second = p_entities.begin();
                while (second != p_entities.end() )
                {
                    //point local sprite to sprite contained in the list
                    sprite2 = (Sprite*) *second;

                    //if other entity is active and not same as first entity...
                    if ( sprite2->getAlive() && sprite2->getVisible() &&
sprite2->isCollidable() && sprite1 != sprite2 )
                    {
                        //test for collision
                        if ( collision(sprite1, sprite2 ) ) {
                            //notify game of collision
                            game_entityCollision( sprite1, sprite2 );
                        }
                    }
                    //go to the next sprite in the list
                    second++;
                }
            }
            //go to the next sprite in the list
            first++;
        }//render2d
    } //while
}
```

Now let's check out the collision methods that do all the real work of performing a collision test. We need two methods for the two collision tests supported by the engine—bounding rectangle and distance-based. In addition, one method merely called `collision` will determine what type of collision the sprite is configured to use and then call the bounding rectangle or distance version to test for the collision. The `Sprite` class defines `COLLISION_RECT` (located in `enum CollisionType`) by default. The other two types of collision that can be set are `COLLISION_DIST` and `COLLISION_NONE` (to skip collision detection—this is equivalent to setting the `collidable` property to false).

```
bool Engine::collision(Sprite *sprite1, Sprite *sprite2)
{
    switch (sprite1->collisionMethod) {
        case COLLISION_RECT:
            return collisionBR(sprite1,sprite2);
            break;
        case COLLISION_DIST:
            return collisionD(sprite1,sprite2);
            break;
        case COLLISION_NONE:
        default:
            return false;
    }
}
```

Here's the bounding rectangle method:

```
bool Engine::collisionBR(Sprite *sprite1, Sprite *sprite2)
    {
        bool ret = false;

        Rect *ra = new Rect(
            sprite1->getX(),
            sprite1->getY(),
            sprite1->getX() + sprite1->getWidth()*sprite1->getScale(),
            sprite1->getY() + sprite1->getHeight()*sprite1->getScale());

        Rect *rb = new Rect(
            sprite2->getX(),
            sprite2->getY(),
            sprite2->getX() + sprite2->getWidth()*sprite2->getScale(),
            sprite2->getY() + sprite2->getHeight()*sprite2->getScale());

        //are any of sprite b's corners intersecting sprite a?
        if (ra->isInside( rb->getLeft(), rb->getTop() ) ||
            ra->isInside( rb->getRight(), rb->getTop() ) ||
            ra->isInside( rb->getLeft(), rb->getBottom() ) ||
            ra->isInside( rb->getRight(), rb->getBottom() ))
                ret = true;

        delete ra;
        delete rb;
        return ret;
}
```

The `collisionBR` method made use of a non-existent class called `Rect`. What gives? Well, the class is new in this chapter, after all, so you might not have met it yet. Here's the definition:

```
class Rect {
public:
    double left,top,right,bottom;
public:
        Rect(int left,int top,int right,int bottom);
        Rect(double left,double top,double right,double bottom);
        virtual ~Rect() { }
        double getLeft() { return left; }
        double getTop() { return top; }
        double getRight() { return right; }
        double getBottom() { return bottom; }
        bool isInside(Vector3 point);
        bool isInside(int x,int y);
        bool isInside(double x,double y);
    };
```

And now for the `Rect` implementation:

```
Rect::Rect(int left,int top,int right,int bottom)
{
    this->left = (double)left;
    this->top = (double)top;
    this->right = (double)right;
    this->bottom = (double)bottom;
}
```

```
Rect::Rect(double left,double top,double right,double bottom)
{
    this->left = left;
    this->top = top;
    this->right = right;
    this->bottom = bottom;
}

bool Rect::isInside(Vector3 point)
{
    return this->isInside(point.getX(), point.getY());
}

bool Rect::isInside(int x,int y)
{
    return this->isInside((double)x, (double)y);
}

bool Rect::isInside(double x,double y)
{
    return (x > left && x < right && y > top && y < bottom);
}
```

The second collision method uses the distance between two sprites to determine whether they are colliding based on their radii—where the radius is the width or height divided by two. Fortunately, we already have the `Vector3` class with its own `Distance` method that will handle this nicely. (The `Vector3` class was first introduced in Chapter 3.) Most of the code in the `collisionD` method is setup code to load the two `Vector3` objects before calculating the distance between them. If the distance is less than the radius of each sprite, then they are overlapping and a collision has occurred!

```
bool Engine::collisionD(Sprite *sprite1, Sprite *sprite2)
{
    double radius1, radius2;

    //calculate radius 1
    if (sprite1->getWidth() > sprite1->getHeight())
```

```
        radius1 = (sprite1->getWidth()*sprite1->getScale())/2;
    else
        radius1 = (sprite1->getHeight()*sprite1->getScale())/2;

    //point = center of sprite 1
    double x1 = sprite1->getX() + radius1;
    double y1 = sprite1->getY() + radius1;
    Vector3 vector1(x1, y1, 0.0);

    //calculate radius 2
    if (sprite2->getWidth() > sprite2->getHeight())
        radius2 = (sprite2->getWidth()*sprite2->getScale())/2;
    else
        radius2 = (sprite2->getHeight()*sprite2->getScale())/2;

    //point = center of sprite 2
    double x2 = sprite2->getX() + radius2;
    double y2 = sprite2->getY() + radius2;
    Vector3 vector2(x2, y2, 0.0);

    //calculate distance
    double dist = vector1.Distance( vector2 );

    //return distance comparison
    return (dist < radius1 + radius2);
}
```

Bounding Rectangle Collision Test

Collision testing is fast for your average, reasonable game. But what if you have several hundred or a thousand or more sprites on the screen at once, and each one needs to be included in collision testing? The number of collision tests that must be performed is equal to the square of the number of sprites. So, if there are 100 sprites, there will be 10,000 collision tests. A quick and fairly easy optimization of the collision testing system is possible here. Rather than testing every sprite with every other sprite, *twice* through the loop, it would be better to run two loops and compare every even sprite with every odd sprite in the list. This is something to keep in mind if you create a game that will require a lot of collision tests (which is not the norm).

The CollisionDemo (bounding rectangle version) is really fascinating to watch because it draws translucent boxes over the sprites when collisions occur. Figure 9.1 shows the program running, while Figure 9.2 shows some stats. The program does not run very well with a large number of sprites, mainly due to the boxes being rendered each time. If you have 1,000 sprites—with a corresponding 1,000,000 collision tests—then there could be upwards of 20,000 or so collision events per frame. The boxes are automatically removed after a few milliseconds, but they still slow down the program. But, it does run nicely with 100 or so sprites (which is far more than what you will usually find in a game during a single frame).

Let's see the source code for the CollisionDemo_BR program (which focuses on the bounding rectangle method). We're getting a bit ahead of ourselves here by using the unknown Console class, but I wanted you to see the results of the collision test without interrupting the subject at hand. The source code for Console is provided in the next chapter.

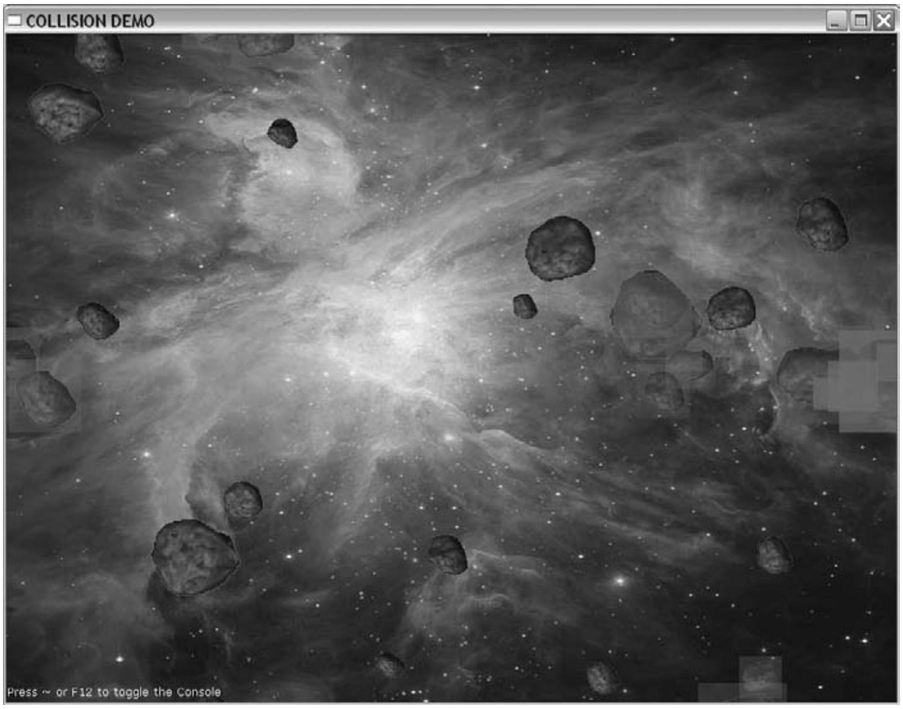
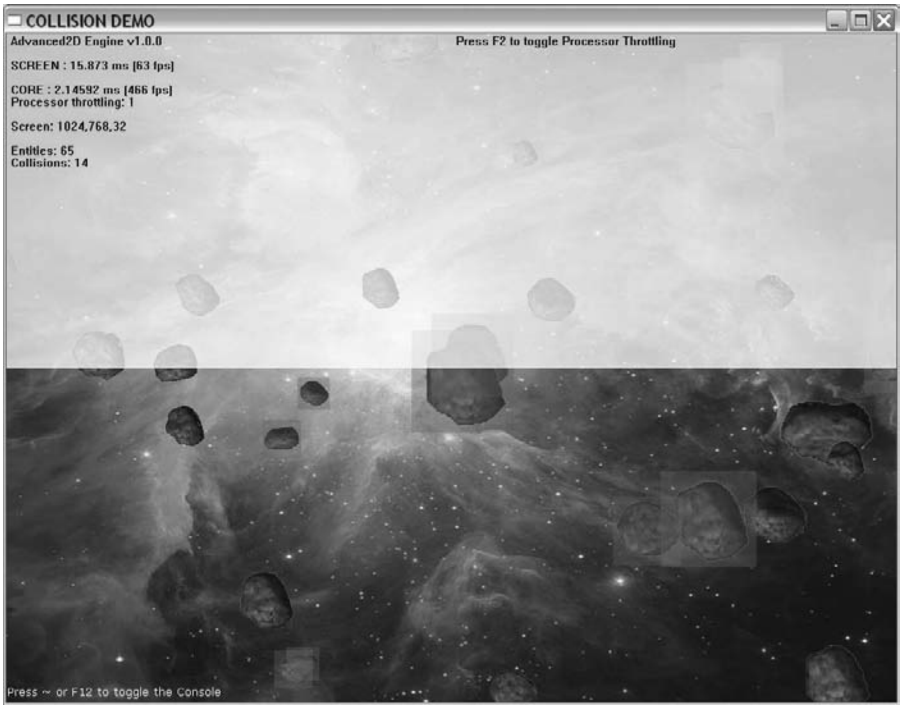


Figure 9.1
The CollisionDemo program demonstrates bounding rectangle collision detection using translucent collision boxes.



```
Advanced2D Engine v1.0.0                    Press F2 to toggle Processor Throttling

SCREEN : 15.873 ms [63 fps]

CORE : 2.14592 ms [466 fps]
Processor throttling: 1

Screen: 1024,768,32

Entities: 65
Collisions: 14

Press ~ or F12 to toggle the Console
```

Figure 9.2
The stats of the CollisionDemo program show that the translucent boxes really hurt performance!

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

#define SCREENW 1024
#define SCREENH 768
#define OBJECT_BACKGROUND 1
#define OBJECT_SPRITE 100
#define MAX 40

Texture *asteroid_image;
Font *font;
Console *console;
std::ostringstream ostr;
Texture *collisionBox;
int collisions;

bool game_preload()
{
    g_engine->setAppTitle("COLLISION DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(SCREENW);
    g_engine->setScreenHeight(SCREENH);
    g_engine->setColorDepth(32);
    return 1;
}

bool game_init(HWND)
{
    //load background image
    Sprite *background = new Sprite();
    if (!background->loadImage("orion.bmp")) {
        g_engine->message("Error loading orion.bmp");
        return false;
    }
    background->setObjectType(OBJECT_BACKGROUND);
    background->setCollidable(false);
    g_engine->addEntity(background);

    //create the console
    console = new Console();
    if (!console->init()) {
        g_engine->message("Error initializing console");
        return false;
    }

    //load asteroid image
    asteroid_image = new Texture();
    if (!asteroid_image->Load("asteroid.tga")) {
        g_engine->message("Error loading asteroid.tga");
        return false;
    }

    //create asteroid sprites
    Sprite *asteroid;
    for (int n=0; n < MAX; n++)
    {
        //create a new asteroid sprite
        asteroid = new Sprite();
        asteroid->setObjectType(OBJECT_SPRITE);
        ostr.str(""); ostr << "ASTEROID" << n;
        asteroid->setName(ostr.str());
        asteroid->setImage(asteroid_image);
        asteroid->setScale( (float)(rand() % 150 + 50) / 100.0f );
        //set animation properties
        asteroid->setTotalFrames(64);
        asteroid->setColumns(8);
        asteroid->setSize(60,60);
        asteroid->setPosition( rand() % SCREENW, rand() % SCREENH );
        asteroid->setFrameTimer( rand() % 90 + 10 );
        asteroid->setCurrentFrame( rand() % 64 );
        if (rand()%2==0) asteroid->setAnimationDirection(-1);

        //set movement properties
        float vx = (float)(rand()%10 - 5)/10.0f;
        float vy = (float)(rand()%10 - 5)/10.0f;
        asteroid->setVelocity( vx, vy );

        //collision toggle
        asteroid->setCollidable(true);

        //movement timer keeps sprite consistent at any framerate
        asteroid->setMoveTimer( 16 );

        //add asteroid to the entity manager
        g_engine->addEntity(asteroid);
    }

    //load the Verdana10 font
    font = new Font();
    if (!font->loadImage("verdana10.tga")) {
        g_engine->message("Error loading verdana10.tga");
        return false;
    }
    font->setColumns(16);
    font->setCharSize(20,16);
    if (!font->loadWidthData("verdana10.dat")) {
        g_engine->message("Error loading verdana10.dat");
        return false;
    }

    //load highlight image used to show collisions
    collisionBox = new Texture();
    if (!collisionBox->Load("highlight.tga")) {
        g_engine->message("Error loading highlight.tga");
        return false;
    }

    return true;
}
```

```
void updateConsole()
{
    int y = 0;
    console->print(g_engine->getVersionText(), y++);
    y++;

    ostr.str("");
    ostr << "SCREEN : " << (float)(1000.0f/g_engine->getFrameRate_real())
        << " ms (" << g_engine->getFrameRate_real() << " fps)";
    console->print(ostr.str(), y++);
    y++;

    ostr.str("");
    ostr << "CORE : " << (float)(1000.0f/g_engine->getFrameRate_core())
        << " ms (" << g_engine->getFrameRate_core() << " fps)";
    console->print(ostr.str(), y++);

    ostr.str("");
    ostr << "Processor throttling: " << g_engine->getMaximizeProcessor();
    console->print(ostr.str(), y++);
    y++;

    ostr.str("");
    ostr << "Screen: " << g_engine->getScreenWidth() << ","
        << g_engine->getScreenHeight() << "," << g_engine->getColorDepth();
    console->print(ostr.str(), y++);
    y++;

    ostr.str("");
    ostr << "Entities: " << g_engine->getEntityCount();
    console->print(ostr.str(), y++);

    ostr.str("");
    ostr << "Collisions: " << collisions;
    console->print(ostr.str(), y++);
    y++;


    ostr.str("");
    ostr << "Press F2 to toggle Processor Throttling";
    console->print(ostr.str(), 27);
}
```

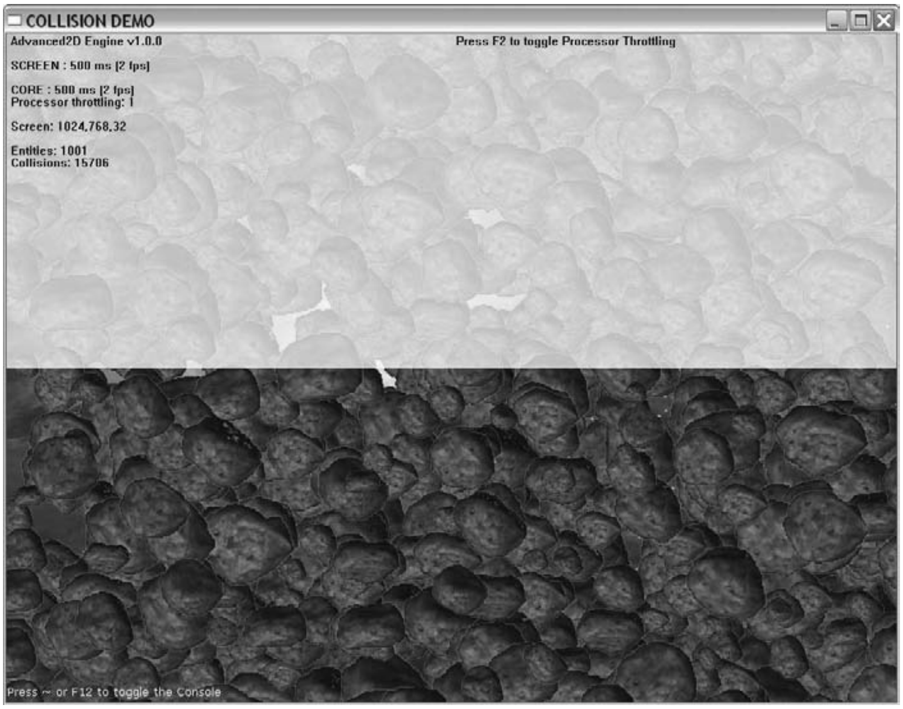
```
void game_update()
{
    updateConsole();
    collisions = 0;
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));
    g_engine->SetIdentity();
}

void game_render2d()
{
    font->Print(1,SCREENH-20,"Press ~ or F12 to toggle the Console");
    //draw console
    if (console->isShowing()) console->draw();
}

void game_keyRelease(int key)
{
    switch (key) {
        case DIK_ESCAPE:
            g_engine->Close();
            break;
        case DIK_F12:
        case DIK_GRAVE:
            console->setShowing( !console->isShowing() );
            break;
        case DIK_F2:
            g_engine->setMaximizeProcessor(!g_engine->getMaximizeProcessor());
            break;
    }
}

void game_end()
{
    delete console;
    delete asteroid_image;
    delete font;
}
```

```
void game_entityUpdate(Advanced2D::Entity* entity)
{
    switch(entity->getObjectType())
    {
        case OBJECT_SPRITE:
            Sprite* spr = (Sprite*)entity;
            if (spr->getX() < -60) spr->setX(SCREENW);
            if (spr->getX() > SCREENW) spr->setX(-60);
            if (spr->getY() < -60) spr->setY(SCREENH);
            if (spr->getY() > SCREENH) spr->setY(-60);
            break;
    }
}

void game_entityCollision(Advanced2D::Entity* entity1,Advanced2D::Entity* entity2)
{
    Sprite *box;
    Sprite *a = (Sprite*)entity1;
    Sprite *b = (Sprite*)entity2;

    if (a->getObjectType() == OBJECT_SPRITE && b->getObjectType() == OBJECT_SPRITE)
    {
        collisions++;

        //add first collision box
        box = new Sprite();
        box->setColor(0x33DD4444);
        box->setImage(collisionBox);
        box->setPosition( a->getPosition() );
        box->setScale( a->getWidth() * a->getScale() / 100.0f );
        box->setLifetime(100);
        g_engine->addEntity( box );

        //add second collision box
        box = new Sprite();
        box->setColor(0x33DD4444);
        box->setImage(collisionBox);
        box->setPosition( b->getPosition() );
        box->setScale( b->getWidth() * b->getScale() / 100.0f );
        box->setLifetime(100);
        g_engine->addEntity( box );
    }
}
```

```
void game_keyPress(int key) { }
void game_mouseButton(int button) { }
void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y) { }
void game_mouseWheel(int wheel) { }
void game_entityRender(Advanced2D::Entity* entity) { }
```

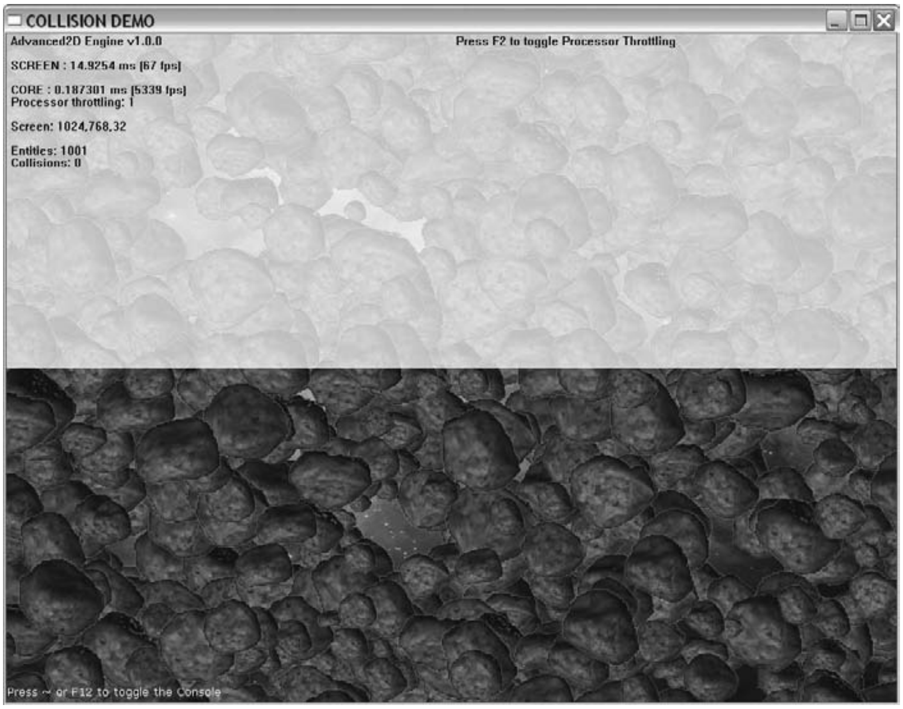
One of the most obvious performance improvements that can be made to a 2D game is in the collision detection department. Figure 9.3 shows the same CollisionDemo program running, but this time there are 1,000 sprites with bounding rectangle collision detection turned on. This demo performs 1,000,000 collision tests, each of which requires a call to the `isInside` function—so, one conditional and four function calls, times 1,000,000, every frame. The framerate has dropped to 2 fps, which is slideshow rate.

Although one may argue that 1,000 sprites is a gross exaggeration of what will ever be found in a real game, the exaggeration helps to identify bottlenecks that slow down the engine. Here's a good question: What if you just want to *draw* a



```
Advanced2D Engine v1.0.0                    Press F2 to toggle Processor Throttling

SCREEN : 500 ms [2 fps]

CORE : 500 ms [2 fps]
Processor throttling: 1

Screen: 1024,768,32

Entities: 1001
Collisions: 15706

Press ~ or F12 to toggle the Console
```

Figure 9.3
This sprite collision demo is a performance punishment test!



```
Advanced2D Engine v1.0.0                    Press F2 to toggle Processor Throttling

SCREEN : 14.9254 ms [67 fps]

CORE : 0.187301 ms [5339 fps]
Processor throttling: 1

Screen: 1024,768,32

Entities: 1001
Collisions: 0

Press ~ or F12 to toggle the Console
```

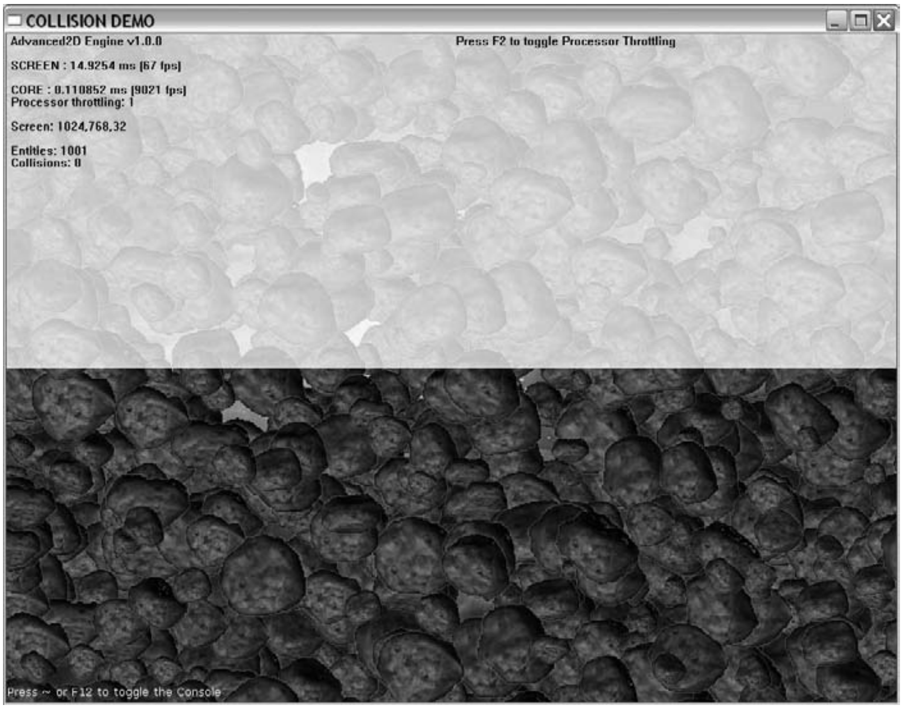
Figure 9.4
The engine supports sprite rendering without collision detection.

thousand or so sprites, without concern for collisions? That's a good point. The `Sprite` class has a property called `collidable` (with support methods `isCollidable` and `setCollidable`). Turning off collision testing for the asteroid sprites results in a very healthy improvement, as shown in Figure 9.4.

What this illustrates is a need for optimization. Before attempting to speed up the code, let's start with the compiler. If you're using Visual C++, change the configuration in both the Engine and CollisionDemo projects from Debug to Release build (and perform a Rebuild All), which will automatically optimize the project. If you're using Dev-C++, open the Project Options and choose Compiler, Optimization, and Best Optimization.

Advice

Dev-C++ supports multiple compiler configurations too, but not by default. Open the Tools menu and choose Compiler Options, and you can create new configurations using the dialog that appears (including Debug and Release) and set each configuration using the Settings tab.



```
Advanced2D Engine v1.0.0                    Press F2 to toggle Processor Throttling

SCREEN : 14.9254 ms [67 fps]

CORE : 0.110852 ms [9021 fps]
Processor throttling: 1

Screen: 1024,768,32

Entities: 1001
Collisions: 0

Press ~ or F12 to toggle the Console
```

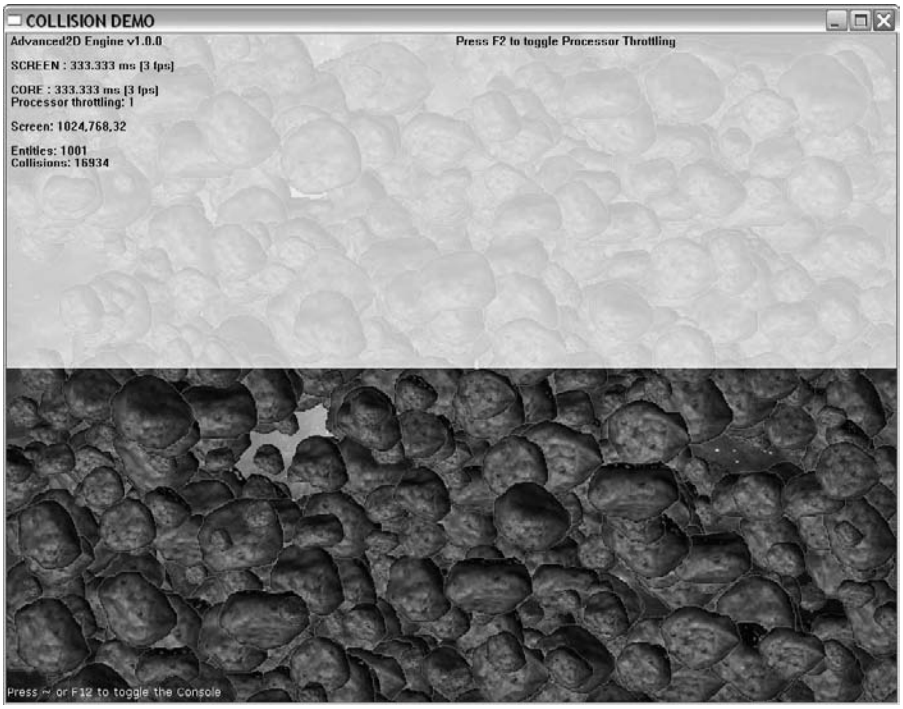
Figure 9.5
Changing from Debug to Release build nearly doubled performance!

With compiler optimizations turned on for a Release build, the new non-collision results are shown in Figure 9.5. As you can see, the framerate nearly doubled!

But how will the Release build improve the framerate with collisions turned back on? Let's go back into the `game_init` function and turn collision back on for all asteroid sprites. When run, the output results in 3 fps, as shown in Figure 9.6. The same 90-percent improvement is seen here, but the small framerate is rounded down (and is really closer to 4 fps).

Still not satisfied with the result? Sixty million collision tests per second (one million per frame) is some very heavy processing—a heavy load for a game. There is a simpler improvement that can be made here—reducing the number of times collision testing is performed. In a perfect world, where everyone owns a quantum computer that's capable of performing nearly instantaneous calculations, we could just throw as many collision tests as we want at the engine, and it would handle them with ease.

We would not want to drop the rate to once per second because that would result in noticeable artifacts (such as clear hits going undetected in a high-speed arcade



```
Advanced2D Engine v1.0.0                    Press F2 to toggle Processor Throttling

SCREEN : 333.333 ms [3 fps]

CORE : 333.333 ms [3 fps]
Processor throttling: 1

Screen: 1024,768,32

Entities: 1001
Collisions: 16934

Press ~ or F12 to toggle the Console
```

Figure 9.6
With collisions turned back on, the framerate improved by the same amount (percentage-wise).

game). In your typical arcade-style shooter, bullets are the fastest sprites, capable of crossing the screen in about one second. That's a velocity of about 1,000 pixels per second, or one pixel per millisecond—which is extremely fast. Taking that into account and considering that the average sprite is 64 pixels wide, we come up with a figure of 60 fps (~16 ms)—the screen refresh rate. That's the ideal rate in order to catch all collisions *immediately* when they occur. But it is simply overkill for a sprite-based game and is an inefficient use of cycles.

A cleaner number is more like 10 to 20 times per second. If we sample collisions at 10 Hz, we can catch fast-moving sprites at a granularity of 100 pixels (at most). Sampling at 20 Hz cuts it down to 50-pixel granularity—that is, the position of the sprite every 50 ms, based on the assumption that the screen contains at least 1,000 pixels in one direction (for instance, horizontally). But remember, these are extremes! We're just estimating based on the extreme cases, and it's rare for a sprite to move that fast in practice. At any rate, we should reduce the collision testing to 20 Hz. (Note: This was done retroactively in `Engine::Update` earlier in the chapter.)

So, what solution can we come up with to improve collision processing to an acceptable level? In addition to the even/odd optimization mentioned earlier, another way to optimize the sprite collision system is by dividing the screen up into partitions or squares and only testing sprites that exist in the same area of the screen (like a 2D version of binary space partitioning, an optimization used to improve 3D games).

Distance-Based Collision Test

The game engine also supports collision detection using the distance between the centers of two sprites. Using this method, the center point of each sprite is determined, and then the distance function is used to calculate the distance between the two centers. Taking the radius of each sprite and accounting for the scaling factor (if any), this method then determines whether the distance between the two sprites is short enough for them to collide.

Figure 9.7 shows the CollisionDemo_D program output. This version is running with 50 sprites. There is a new art set used in this demo to differentiate it from the

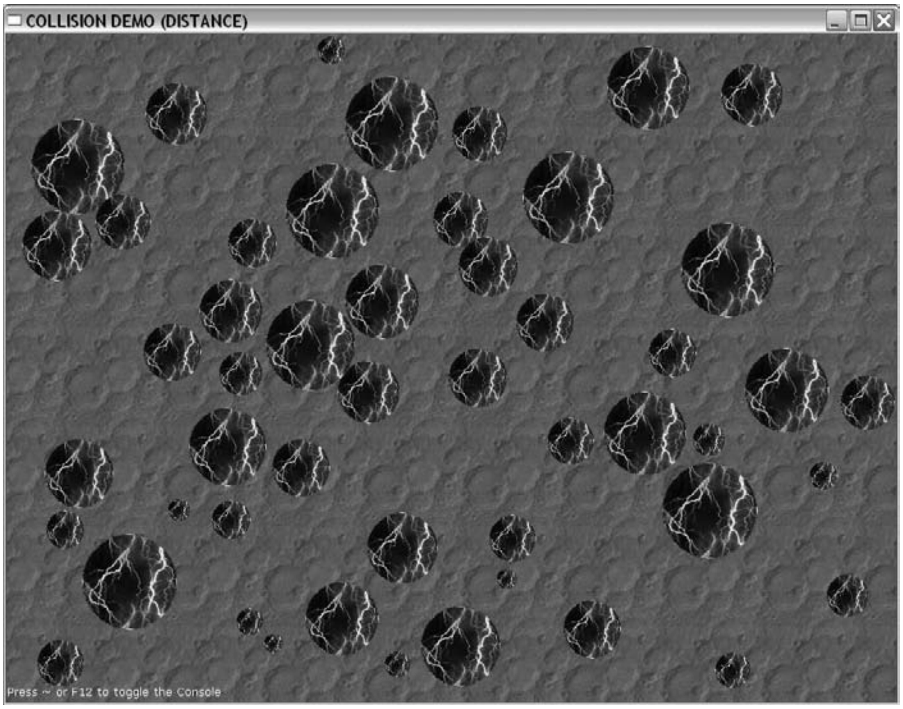
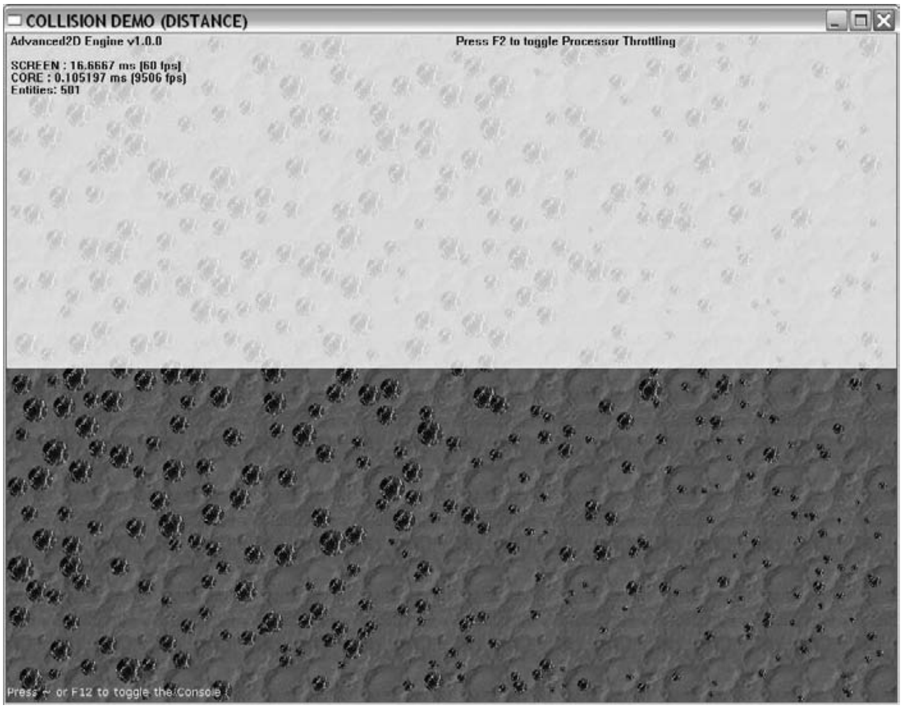

COLLISION DEMO (DISTANCE)
Press ~ or F12 to toggle the Console

Figure 9.7
The distance-based collision demo with 50 sprites.

bounding rectangle demo. Distance-based collision detection results in some very accurate-looking collision response when dealing with circular-shaped sprites, such as the images used in this example. One interesting aspect of this program is the way it responds to sprite collisions. Rather than just showing a collision box, it actually causes the sprites to bounce off of each other. (See `game_entityUpdate` for the response code.)

Figure 9.8 shows another version of the program running. This time there are 500 sprites, but the core is still achieving a tenth of a millisecond (approximately 9,000 fps). What is the main difference between this program and the previous one, which seemed to have performance problems? The real slowdown was due to the hundreds of translucent boxes being added to the entity manager every frame and then deleted shortly thereafter. So, this situation begs the question, what kind of performance would we see using bounding rectangle collision—but without the boxes? I'll let you explore the possibility!


COLLISION DEMO (DISTANCE)

```
Advanced2D Engine v1.0.0                    Press F2 to toggle Processor Throttling

SCREEN : 16.6667 ms (60 fps)
CORE : 0.105197 ms (9506 fps)
Entities: 501
```

Press ~ or F12 to toggle the Console

Figure 9.8
The distance-based collision demo with 50 sprites.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

#define SCREENW 1024
#define SCREENH 768
#define OBJECT_BACKGROUND 1
#define OBJECT_SPRITE 100
#define MAX 50
#define SCALE 70

Texture *ball_image;
Font *font;
Console *console;
std::ostringstream ostr;
int collisions;

bool game_preload()
{
    g_engine->setAppTitle("COLLISION DEMO (DISTANCE)");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(SCREENW);
    g_engine->setScreenHeight(SCREENH);
    g_engine->setColorDepth(32);
    return 1;
}

bool game_init(HWND)
{
    //load background image
    Sprite *background = new Sprite();
    if (!background->loadImage("craters.tga")) {
        g_engine->message("Error loading craters.tga");
        return false;
    }
    background->setObjectType(OBJECT_BACKGROUND);
    background->setCollidable(false);
    g_engine->addEntity(background);

    //create the console
    console = new Console();
    if (!console->init()) {
        g_engine->message("Error initializing console");
        return false;
    }
    //load asteroid image
    ball_image = new Texture();
    if (!ball_image->Load("lightningball.tga")) {
        g_engine->message("Error loading lightningball.tga");
        return false;
    }

    //create sprites
    Sprite *sprite;
    for (int n=0; n < MAX; n++)
    {
        //create a new sprite
        sprite = new Sprite();
        sprite->setObjectType(OBJECT_SPRITE);
        sprite->setImage(ball_image);
        sprite->setSize(128,128);
        sprite->setScale( (float)(rand() % SCALE + SCALE/4) / 100.0f );
        sprite->setPosition( rand() % SCREENW, rand() % SCREENH );
        sprite->setCollisionMethod(COLLISION_DIST);

        //set velocity
        float vx = (float)(rand()%30 - 15)/10.0f;
        float vy = (float)(rand()%30 - 15)/10.0f;
        sprite->setVelocity( vx, vy );

        //add sprite to the entity manager
        g_engine->addEntity(sprite);
    }

    //load the Verdana10 font
    font = new Font();
    if (!font->loadImage("verdana10.tga")) {
        g_engine->message("Error loading verdana10.tga");
        return false;
    }
    if (!font->loadWidthData("verdana10.dat")) {
        g_engine->message("Error loading verdana10.dat");
        return false;
    }
    font->setColumns(16);
    font->setCharSize(20,16);

    return true;
}
```

```
void updateConsole()
{
    int y = 0;
    console->print(g_engine->getVersionText(), y++);
    y++;
    ostr.str("");
    ostr << "SCREEN : " << (float)(1000.0f/g_engine->getFrameRate_real())
        << " ms (" << g_engine->getFrameRate_real() << " fps)";
    console->print(ostr.str(), y++);
    ostr.str("");
    ostr << "CORE : " << (float)(1000.0f/g_engine->getFrameRate_core())
        << " ms (" << g_engine->getFrameRate_core() << " fps)";
    console->print(ostr.str(), y++);
    ostr.str("");
    ostr << "Entities: " << g_engine->getEntityCount();
    console->print(ostr.str(), y++);
    ostr.str("");
    ostr << "Press F2 to toggle Processor Throttling";
    console->print(ostr.str(), 27);
}

void game_update()
{
    updateConsole();
    collisions = 0;
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));

}

void game_render2d()
{
    font->Print(1,SCREENH-20,"Press ~ or F12 to toggle the Console");
    //draw console
    if (console->isShowing()) console->draw();
}

void game_keyRelease(int key)
{
    switch (key) {
        case DIK_ESCAPE:
            g_engine->Close();
            break;
        case DIK_F12:
        case DIK_GRAVE:
            console->setShowing( !console->isShowing() );
            break;
        case DIK_F2:
            g_engine->setMaximizeProcessor(!g_engine->getMaximizeProcessor());
            break;
    }
}

void game_end()
{
    delete console;
    delete ball_image;
    delete font;
}

void game_entityUpdate(Advanced2D::Entity* entity)
{
    switch(entity->getObjectType())
    {
        case OBJECT_SPRITE:
            Sprite* spr = (Sprite*)entity;
            float w = (float)spr->getWidth() * spr->getScale();
            float h = (float)spr->getHeight() * spr->getScale();
            float vx = spr->getVelocity().getX();
            float vy = spr->getVelocity().getY();

            if (spr->getX() < 0) {
                spr->setX(0);
                vx = fabs(vx);
            }
            else if (spr->getX() > SCREENW-w) {
                spr->setX(SCREENW-w);
                vx = fabs(vx) * -1;
            }
            if (spr->getY() < 0) {
                spr->setY(0);
                vy = fabs(vy);
            }
            else if (spr->getY() > SCREENH-h) {
                spr->setY(SCREENH-h);
                vy = fabs(vy) * -1;
            }

            spr->setVelocity(vx,vy);
            break;
    }
}

void game_entityCollision(Advanced2D::Entity* entity1,Advanced2D::Entity* entity2)
{
    Sprite *box;
    Sprite *a = (Sprite*)entity1;
    Sprite *b = (Sprite*)entity2;
    if (a->getObjectType() == OBJECT_SPRITE && b->getObjectType() == OBJECT_SPRITE)
    {
        collisions++;

        //get position of both sprites
        double x1 = a->getX();
        double y1 = a->getY();
        double x2 = b->getX();
        double y2 = b->getY();

        //get velocity of both sprites
        double vx1 = a->getVelocity().getX();
        double vy1 = a->getVelocity().getY();
        double vx2 = b->getVelocity().getX();
        double vy2 = b->getVelocity().getY();

        //compare sprite orientation toward each other
        if (x1 < x2) {
            vx1 = fabs(vx1) * -1;
            vx2 = fabs(vx1);
        }
        else if (x1 > x2) {
            vx1 = fabs(vx1);
            vx2 = fabs(vx2) * -1;
        }
        if (y1 < y2) {
            vy1 = fabs(vy1) * -1;
            vy2 = fabs(vy2);
        }
        else {
            vy1 = fabs(vy1);
            vy2 = fabs(vy2) * -1;
        }

        //set new velocities
        a->setVelocity(vx1,vy1);
        b->setVelocity(vx2,vy2);
    }
}
void game_keyPress(int key) { }
void game_mouseButton(int button) { }
void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y) { }
void game_mouseWheel(int wheel) { }
void game_entityRender(Advanced2D::Entity* entity) { }
```

That wraps up physics for our game engine. Because the collision detection implementations are built into the core of the engine (as an early design goal), we do not need to invoke a `Physics` class to make use of it. When you create a sprite object, you can set its `collisionMethod` property to `COLLISION_RECT` or `COLLISION_DIST`, and the engine will take care of the rest. That's good! Collision response is then done in the `game_entityCollision` event function in your main code file—and *that* is where all the real action is to be found! As with any solution to a programming problem, there are alternatives and even better ways of doing things. As we discussed in this chapter, there are ways to optimize sprite collision algorithms. You should consider optimizing the collision system to work best with the type of game you're building at any particular time.

This page intentionally left blank

CHAPTER 10

Math

This chapter covers some basic math functions that will improve the support library within the game engine. First we will look at linear velocity, then we'll examine a more advanced technique for calculating the angle between two points (which is helpful when targeting an enemy in a game or for moving a sprite along a path set by waypoints). Note that this chapter is not about the *theory* behind any of these math functions, nor does this text attempt to derive any of the math functions—we are simply coding some of the more common math functions into our game engine.

The versatile `Vector3` class, introduced in Chapter 3, already has many commonly used math functions built in. You may want to review the `Vector3` class because it provides the following:

- Distance between two vectors
- Length of a vector
- Dot product
- Cross product
- Normalized vector

Because this assortment of math functions is already very useful as contained within `Vector3`, you may use them when convenient, but I believe it is helpful to provide, in the engine, more generic versions of these and the new math functions we develop this chapter. More specifically, we need a `Math` class embedded in the game engine (like `g_engine->audio` for the audio system).

Math Class

The `Math` class will be added to the latest version of the game engine in this chapter on the CD-ROM. The first step is to incorporate some of the more useful math functions from the `Vector3` class that may be helpful in a more generic context (although having those functions embedded in `Vector3` is still a good idea). The `Math` class will include some overloaded versions of these functions that work with `double` data type parameters (as well as `Vector3` parameters) and some new functions introduced in this chapter:

- Distance
- Vector length
- Dot product
- Cross product
- Normalized vector
- Converting radians to degrees
- Converting degrees to radians
- X velocity of an angle
- Y velocity of an angle
- Angle to target vector

Math Class Header

Here is the header for the `Math` class with some constants predefined for convenience:

```
#include "Advanced2D.h"
#pragma once
```

```
namespace Advanced2D {
    const double PI = 3.1415926535;
    const double PI_over_180 = PI / 180.0f;
    const double PI_under_180 = 180.0f / PI;

    class Math {
    public:
        double toDegrees(double radian);
        double toRadians(double degree);
        double wrapAngleDegs(double degs);
        double wrapAngleRads(double rads);
        double LinearVelocityX(double angle);
        double LinearVelocityY(double angle);
        Vector3 LinearVelocity(double angle);
        double AngleToTarget(double x1,double y1,double x2,double y2);
        double AngleToTarget(Vector3& source,Vector3& target);
        double Distance( double x1,double y1,double x2,double y2 );
        double Distance( Vector3& v, Vector3& vec2 );
        double Length(Vector3& vec);
        double Length(double x,double y,double z);
        double DotProduct(double x1,double y1,double z1,
            double x2,double y2,double z2);
        double DotProduct(Vector3& vec1, Vector3& vec2);
        Vector3 CrossProduct(double x1,double y1,double z1,
            double x2,double y2,double z2);
        Vector3 CrossProduct(Vector3& vec1, Vector3& vec2);
        Vector3 Normal(double x,double y,double z);
        Vector3 Normal(Vector3& vec);
    };
};
```

Math Class Implementation

Now we can go over the code for the Math implementation file. The `Math` class includes the angular velocity and angle to target functions, which I will explain in detail in subsequent sections of the chapter.

```
#include "Advanced2D.h"
namespace Advanced2D {
    double Math::toDegrees(double radians)
    {
        return radians * PI_under_180;
    }
```

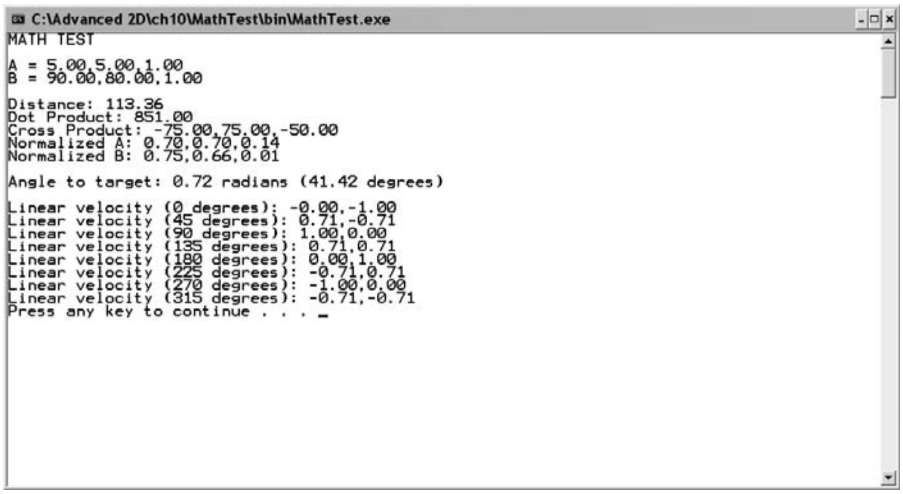
```
double Math::toRadians(double degrees)
{
    return degrees * PI_over_180;
}

double Math::wrapAngleDegs(double degs)
{
    double result = fmod(degs, 360.0);
    if (result < 0) result += 360.0f;
    return result;
}

double Math::wrapAngleRads(double rads)
{
    double result = fmod(rads, PI * 2.0);
    if (result < 0) result += PI * 2.0;
    return result;
}

double Math::LinearVelocityX(double angle)
{
    angle -= 90;
    if (angle < 0) angle = 360 + angle;
    return cos( angle * PI_over_180);
}

double Math::LinearVelocityY(double angle)
{
    angle -= 90;
    if (angle < 0) angle = 360 + angle;
    return sin( angle * PI_over_180);
}

Vector3 Math::LinearVelocity(double angle)
{
    double vx = LinearVelocityX(angle);
    double vy = LinearVelocityY(angle);
    return Vector3(vx,vy,0.0f);
}

double Math::AngleToTarget(double x1,double y1,double x2,double y2)
{
    double deltaX = (x2-x1);
```

```
        double deltaY = (y2-y1);
        return atan2(deltaY,deltaX);
    }

    double Math::AngleToTarget(Vector3& source,Vector3& target)
    {
        return AngleToTarget(source.getX(),source.getY(),target.getX(),
target.getY());
    }

    double Math::Distance( double x1,double y1,double x2,double y2 )
    {
        double deltaX = (x2-x1);
        double deltaY = (y2-y1);
        return sqrt(deltaX*deltaX + deltaY*deltaY);
    }

    double Math::Distance( Vector3& vec1, Vector3& vec2 )
    {
        return Distance(vec1.getX(),vec1.getY(),vec2.getX(),vec2.getY());
    }

    double Math::Length(double x,double y,double z)
    {
        return sqrt(x*x + y*y + z*z);
    }

    double Math::Length(Vector3& vec)
    {
        return Length(vec.getX(),vec.getY(),vec.getZ());
    }

    double Math::DotProduct(double x1,double y1,double z1,
        double x2,double y2,double z2)
    {
        return (x1*x2 + y1*y2 + z1*z2);
    }

    double Math::DotProduct( Vector3& vec1, Vector3& vec2 )
    {
        return DotProduct(vec1.getX(),vec1.getY(),vec1.getZ(),
            vec2.getX(),vec2.getY(),vec2.getZ());
    }
    Vector3 Math::CrossProduct( double x1,double y1,double z1,
        double x2,double y2,double z2)
    {
        double nx = (y1*z2)-(z1*y2);
        double ny = (z1*y2)-(x1*z2);
        double nz = (x1*y2)-(y1*x2);
        return Vector3(nx,ny,nz);
    }

    Vector3 Math::CrossProduct( Vector3& vec1, Vector3& vec2 )
    {
        return CrossProduct(vec1.getX(),vec1.getY(),vec1.getZ(),
            vec2.getX(),vec2.getY(),vec2.getZ());
    }

    Vector3    Math::Normal(double x,double y,double z)
    {
        double length = Length(x,y,z);
        if (length != 0) length = 1 / length;
        double nx = x*length;
        double ny = y*length;
        double nz = z*length;
        return Vector3(nx,ny,nz);
    }

    Vector3 Math::Normal(Vector3& vec)
    {
        return Normal(vec.getX(),vec.getY(),vec.getZ());
    }
};
```

Now that you have the `Math` class available, you can begin exploring its features in a more convenient way (as opposed to writing examples with C++ functions, and then porting them to the class afterward—you can now just defer to the class directly).

Math Test

Before getting into the new math functions, let's run the `Math` class through a few tests to make sure it's working as expected. This is always a good idea before plugging a new module or class into the engine (and assuming it works without testing). Figure 10.1 shows the output of the MathTest program. Note that this program is using the same values that were used in the VectorTest program back in Chapter 3, but the output has been changed. (Since we know that the `Vector3` class is working as expected, the property tests have been removed.) The



```
C:\Advanced 2D\ch10\MathTest\bin\MathTest.exe
MATH TEST

A = 5.00,5.00,1.00
B = 90.00,80.00,1.00

Distance: 113.36
Dot Product: 851.00
Cross Product: -75.00,75.00,-50.00
Normalized A: 0.70,0.70,0.14
Normalized B: 0.75,0.66,0.01

Angle to target: 0.72 radians (41.42 degrees)

Linear velocity (0 degrees): -0.00,-1.00
Linear velocity (45 degrees): 0.71,-0.71
Linear velocity (90 degrees): 1.00,0.00
Linear velocity (135 degrees): 0.71,0.71
Linear velocity (180 degrees): 0.00,1.00
Linear velocity (225 degrees): -0.71,0.71
Linear velocity (270 degrees): -1.00,0.00
Linear velocity (315 degrees): -0.71,-0.71
Press any key to continue . . . _
```

Figure 10.1
This program demonstrates the functionality of the `Math` class.

calculations are now being performed by the `Math` class rather than by `Vector3`'s methods. Pay particular attention to the outputs for "Angle to target" and "Linear velocity," which demonstrate these functions that will be covered next.

```
#include <iostream>
#include <iomanip>
#include "Math.h"
using namespace std;

int main(int argc, char *argv[])
{
    Math math;
    float angle,x,y;
    cout << "MATH TEST" << endl << endl;
    cout.setf(ios::fixed);
    cout << setprecision(2);

    Vector3 A(5,5,1);
    cout << "A = " << A.getX() << "," << A.getY() << "," << A.getZ() << endl;

    Vector3 B(90,80,1);
    cout << "B = " << B.getX() << "," << B.getY() << "," << B.getZ() << endl;
    cout << endl << "Distance: " << math.Distance( A, B ) << endl;
    cout << "Dot Product: " << math.DotProduct( A, B ) << endl;

    Vector3 D = math.CrossProduct( A,B );
    cout << "Cross Product: " <<
        D.getX() << "," << D.getY() << "," << D.getZ() << endl;
    D = math.Normal( A );
    cout << "Normalized A: " << D.getX() << ","
        << D.getY() << "," << D.getZ() << endl;
    D = math.Normal( B );
    cout << "Normalized B: " << D.getX() << ","
        << D.getY() << "," << D.getZ() << endl << endl;

    angle = math.AngleToTarget( A, B );
    cout << "Angle to target: " << angle << " radians (";
    cout << math.toDegrees(angle) << " degrees)" << endl << endl;

    for (angle=0; angle<360; angle+=45) {
        x = math.LinearVelocityX(angle);
        y = math.LinearVelocityY(angle);
        cout << "Linear velocity (" <<
            setprecision(0) << angle << " degrees): ";
        cout << setprecision(2) << x << "," << y << endl;
    }
    system("pause");
    return 0;
}
```

Linear Velocity

Have you ever wondered how some shooter-style games are able to fire projectiles (be they bullets, missiles, plasma bolts, phaser beams, or what have you) at any odd angle away from the player's ship, as well as at any angle from enemy sprites? These projectiles are moving using velocity values (for X and Y) that are based on the object's direction (or angle) of movement. Given any angle, we can calculate the velocity needed to move in precisely that direction. This applies to aircraft, sea vessels, spacecraft, as well as projectiles, missiles, lasers, plasma bolts, or any other object that needs to move at a given angle (presumably toward a target).

The X velocity of a game entity can be calculated for any angle, and that value is then multiplied by the speed at which you want the object to move in the given direction. The `LinearVelocityX` function (following) automatically orients the angle to quadrant four of the Cartesian coordinate system and converts the angle from degrees to radians. Since the cosine function gives us the *horizontal* value of a point on a circle, we use cosine to calculate X velocity as if we were drawing a circle based on a small radius.

```
float Math::LinearVelocityX(float angle)
{
    angle -= 90;
    if (angle < 0) angle = 360 + angle;
    return cos( angle * PI_over_180);
}
```

Likewise for the Y velocity value, we use the Y position on the edge of a circle (based on radius) for the calculation using the sine function.

```
float Math::LinearVelocityY(float angle)
{
    angle -= 90;
    if (angle < 0) angle = 360 + angle;
    return sin( angle * PI_over_180);
}
```

So, as it turns out, the "velocity" of an object based on an angle—that is, its linear velocity—is simply the same pair of X,Y values that would be calculated when tracing the boundary of a circle (based on a radius). The example program called VelocityDemo (shown in Figure 10.2) demonstrates how you can use these math functions in a game.

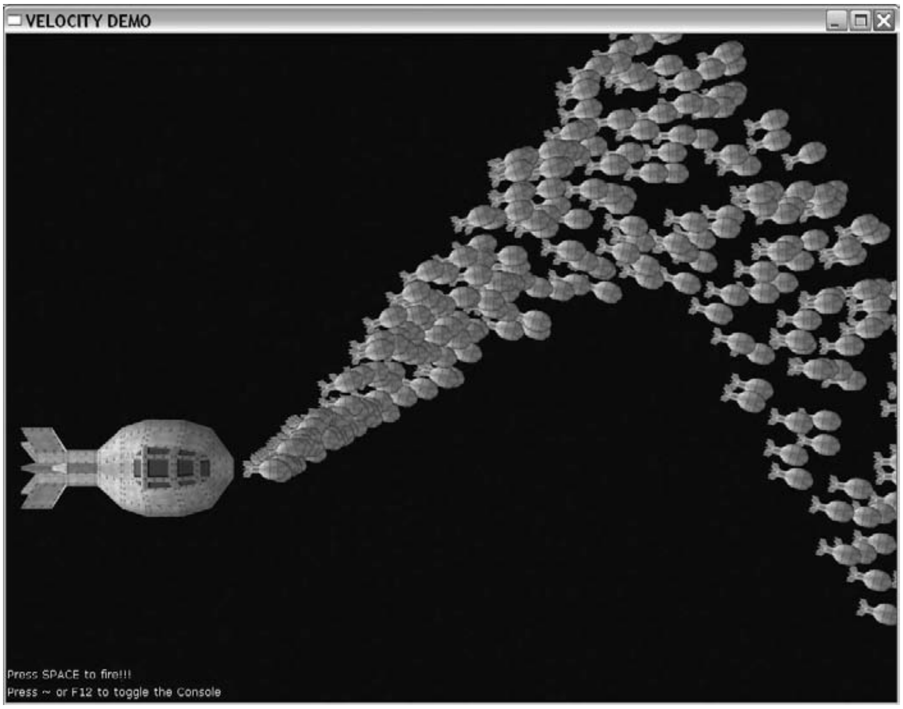


Figure 10.2
The VelocityDemo program shows how to move an object in any direction.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

#define SCREENW 1024
#define SCREENH 768
#define VELOCITY 0.0001
#define ROCKETVEL 3.0
#define OBJECT_SHIP 100
#define OBJECT_ROCKET 200

Font *font;
Console *console;
Sprite *ship;
Vector3 velocity;
Texture *rocket_image;

bool game_preload()
{
    g_engine->setAppTitle("VELOCITY DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(SCREENW);
    g_engine->setScreenHeight(SCREENH);
    g_engine->setColorDepth(32);
    return 1;
}

bool game_init(HWND)
{
    //create the console
    console = new Console();
    if (!console->init()) {
        g_engine->message("Error initializing console");
        return false;
    }

    //create ship sprite
    ship = new Sprite();
    ship->setObjectType(OBJECT_SHIP);
    ship->loadImage("fatship256.tga");
    ship->setRotation( g_engine->math->toRadians(90) );
    ship->setPosition( 10, SCREENH/2-ship->getHeight()/2 );
    g_engine->addEntity(ship);
    //load rocket image
    rocket_image = new Texture();
    rocket_image->Load("fatrocket64.tga");

    //load the Verdana10 font
    font = new Font();
    if (!font->loadImage("verdana10.tga")) {
        g_engine->message("Error loading verdana10.tga");
        return false;
    }
    if (!font->loadWidthData("verdana10.dat")) {
        g_engine->message("Error loading verdana10.dat");
        return false;
    }
    font->setColumns(16);
    font->setCharSize(20,16);

    //maximize processor
    g_engine->setMaximizeProcessor( !g_engine->getMaximizeProcessor() );

    return true;
}

void updateConsole()
{
    std::ostringstream ostr;
    int y = 0;
    console->print(g_engine->getVersionText(), y++);
    ostr.str("");
    ostr << "REFRESH : " << (float)(1000.0f/g_engine->getFrameRate_core())
        << " ms (" << g_engine->getFrameRate_core() << " fps)";
    console->print(ostr.str(), y++);
    ostr.str("");
    ostr << "Entities: " << g_engine->getEntityCount();
    console->print(ostr.str(), y++);
}

void game_update()
{
    updateConsole();
}
```

```
void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));
    g_engine->SetIdentity();
}

void game_render2d()
{
    font->Print(1,SCREENH-20,"Press ~ or F12 to toggle the Console");
    font->Print(1,SCREENH-40,"Press SPACE to fire!!!");

    //draw console
    if (console->isShowing()) console->draw();
}

void game_keyRelease(int key)
{
    switch (key) {
        case DIK_ESCAPE:
            g_engine->Close();
            break;
        case DIK_F12:
        case DIK_GRAVE:
            console->setShowing( !console->isShowing() );
            break;
    }
}
void game_end()
{
    delete console;
    delete font;
    delete ship;
}

void game_entityUpdate(Advanced2D::Entity* entity)
{
    float y;
    Sprite *ship, *rocket;
    Vector3 position;
    switch(entity->getObjectType())
    {
        case OBJECT_SHIP:
            ship = (Sprite*)entity;
            position = ship->getPosition();
            y = position.getY() + velocity.getY();
            if (y < 0) {
                y = 0;
                velocity.setY(0);
            }
            if (y > SCREENH-128) {
                y = SCREENH-128;
                velocity.setY(0);
            }
            position.setY(y);
            ship->setPosition( position );
            break;

        case OBJECT_ROCKET:
            rocket = (Sprite*)entity;
            if (rocket->getX() > SCREENW)
                rocket->setAlive(false);
            break;
    }
}

void game_entityCollision(Advanced2D::Entity* entity1,
    Advanced2D::Entity* entity2) { }

void firerocket()
{
    Sprite *ship = (Sprite*)g_engine->findEntity(OBJECT_SHIP);
    if (!ship)
    {
        g_engine->message("Error locating ship in entity manager!","ERROR");
        g_engine->Close();
    }

    Sprite *rocket = new Sprite();
    rocket->setObjectType(OBJECT_ROCKET);
    rocket->setImage(rocket_image);
    rocket->setMoveTimer(1);
    rocket->setCollidable(false);
    float randrot = rand() % 40 - 20;
    float angle = 90 + randrot;
    rocket->setRotation( g_engine->math->toRadians(angle) );
    float x = ship->getX() + ship->getWidth();
    float y = ship->getY()+ship->getHeight()/2-rocket->getHeight()/2;
    rocket->setPosition(x,y);
    float vx = g_engine->math->LinearVelocityX(angle) * ROCKETVEL;
    float vy = g_engine->math->LinearVelocityY(angle) * ROCKETVEL;
    rocket->setVelocity(vx, vy);
    g_engine->addEntity(rocket);
}


void game_keyPress(int key)
{
    float y;
    switch(key)
    {
        case DIK_UP:
        case DIK_W:
            y = velocity.getY() - VELOCITY;
            if (y < -3.0) y = -3.0;
            velocity.setY(y);
            break;
        case DIK_DOWN:
        case DIK_S:
            y = velocity.getY() + VELOCITY;
            if (y > 3.0) y = 3.0;
            velocity.setY(y);
            break;
        case DIK_SPACE:
        case DIK_LCONTROL:
            firerocket();
            break;
    }
}

void game_mouseButton(int button) { }
void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y) { }
void game_mouseWheel(int wheel) { }
void game_entityRender(Advanced2D::Entity* entity) { }
```

Angle to Target

Calculating the angle from one point to another (as in the case where one sprite is targeting another) is extremely useful (if not crucial) in most games. Imagine you are working on a real-time strategy game. You must program the game so that the player can select units with the mouse and right-click a target location where the unit must move to. Even a simple process like that requires a calculation—between the unit's location and the selected target location in the game. In the space shooter genre, in order to fire at the player's ship, enemies must be able to face the player to fire in the correct direction. I could provide you with many more examples, but I suspect you get the point. The key to this important need is a calculation that I like to call *angle to target*.

The calculation is very simple—about as simple as calculating angular velocity, which is much simpler than the `Distance` function. We need to use another trigonometry function this time: `atan2()`. This is a standard C math library function that calculates the arctangent of two deltas—first the Y delta, then the X delta. A *delta* is the difference between two values. For our purposes here, we need to get the delta of both X and Y for two points. For instance, if Point A is located at X1,Y1, and Point B is located at X2,Y2, then we can calculate the delta of the two points like so:

```
deltaX = X2 - X1
deltaY = Y2 - Y1
```

The `atan2()` function requires the `deltaY` first, then the `deltaX` parameter. Here is the `AngleToTarget` method as it appears in the `Math` class:

```
float Math::AngleToTarget(float x1,float y1,float x2,float y2)
{
    float deltaX = (x2-x1);
    float deltaY = (y2-y1);
    return atan2(deltaY,deltaX);
}
```

I have coded an overloaded version of this function so you can pass `Vector3` values:

```
float Math::AngleToTarget(Vector3& src,Vector3& tgt)
{
    return AngleToTarget(src.getX(),src.getY(),tgt.getX(),tgt.getY());
}
```

See, it's like I said, fairly simple. But, wow, is this unassuming function useful! I would be remiss by not providing a demo program that shows off this newfound tool. The TargetingDemo program (shown in Figure 10.3) is one of my favorite demos! It's the reverse of what you usually find in a video game. In this demo,

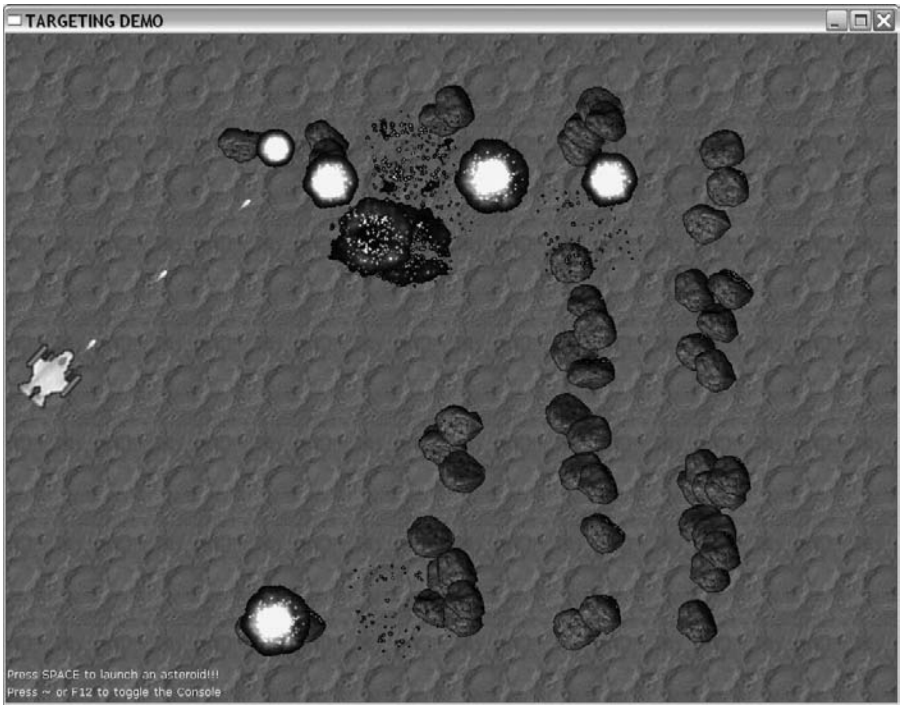

TARGETING DEMO
Press SPACE to launch an asteroid!!!
Press ~ or F12 to toggle the Console

Figure 10.3
The TargetingDemo program demonstrates the utility of angle to target.

you (the player) are in control of the asteroids, and the *computer* has to shoot them! The program first needs to figure out which asteroid is closest. Then it must calculate the angle to that target asteroid. Finally, it can fire a bullet at the target. It's really quite fun watching the computer frantically shoot down asteroids—and get confused when there is a large cluster of asteroids close by!

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

#define SCREENW 1024
#define SCREENH 768
#define BULLET_VEL 3.0
#define ASTEROID_VEL 3.0

#define OBJECT_BACKGROUND 1
#define OBJECT_SHIP 10
#define OBJECT_BULLET 20
#define OBJECT_ASTEROID 30
#define OBJECT_EXPLOSION 40
```

```
Font *font;
Console *console;
Texture *bullet_image;
Texture *asteroid_image;
Texture *explosion_image;
Vector3 ship_position;
Vector3 nearest_asteroid;
Vector3 target_lead;
float ship_angle = 90;
float nearest_distance;
Timer fireTimer;

bool game_preload()
{
    g_engine->setAppTitle("TARGETING DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(SCREENW);
    g_engine->setScreenHeight(SCREENH);
    g_engine->setColorDepth(32);
    return 1;
}


bool game_init(HWND)
{
    //create the background
    Sprite *background = new Sprite();
    background->setObjectType(OBJECT_BACKGROUND);
    if (!background->loadImage("craters.tga")) {
        g_engine->message("Error loading craters.tga");
        return false;
    }
    g_engine->addEntity( background );

    //create the console
    console = new Console();
    if (!console->init()) {
        g_engine->message("Error initializing console");
        return false;
    }

    //create ship sprite
    Sprite *ship = new Sprite();
```

```
ship->setObjectType(OBJECT_SHIP);
if (!ship->loadImage("spaceship80.tga")) {
    g_engine->message("Error loading spaceship.tga");
    return false;
}
ship->setRotation( g_engine->math->toRadians(90) );
ship->setPosition( 10, SCREENH/2-32 );
g_engine->addEntity(ship);

//load bullet image
bullet_image = new Texture();
if (!bullet_image->Load("plasma.tga")) {
    g_engine->message("Error loading plasma.tga");
    return false;
}

//load asteroid image
asteroid_image = new Texture();
if (!asteroid_image->Load("asteroid.tga")) {
    g_engine->message("Error loading asteroid.tga");
    return false;
}

//load the explosion image
explosion_image = new Texture();
if (!explosion_image->Load("explosion_30_128.tga")) {
    g_engine->message("Error loading explosion");
    return false;
}

//load the Verdana10 font
font = new Font();
if (!font->loadImage("verdana10.tga")) {
    g_engine->message("Error loading verdana10.tga");
    return false;
}
if (!font->loadWidthData("verdana10.dat")) {
    g_engine->message("Error loading verdana10.dat");
    return false;
}
```

```
    font->setColumns(16);
    font->setCharSize(20,16);

    //load sound effects
    if (!g_engine->audio->Load("fire.wav","fire")) {
        g_engine->message("Error loading fire.wav");
        return false;
    }

    if (!g_engine->audio->Load("boom.wav","boom")) {
        g_engine->message("Error loading boom.wav");
        return false;
    }

    //maximize processor
    g_engine->setMaximizeProcessor( !g_engine->getMaximizeProcessor() );

    return true;
}

void updateConsole()
{
    std::ostringstream ostr;
    int y = 0;
    console->print(g_engine->getVersionText(), y++);
    ostr.str("");
    ostr << "REFRESH : " << (float)(1000.0f/g_engine->getFrameRate_core())
        << " ms (" << g_engine->getFrameRate_core() << " fps)";
    console->print(ostr.str(), y++);

    ostr.str("");
    ostr << "Entities: " << g_engine->getEntityCount();
    console->print(ostr.str(), y++);

    ostr.str("");
    ostr << "Nearest asteroid: " << nearest_asteroid.getX() << "," << nearest_
asteroid.getY();
    console->print(ostr.str(), y++);

    ostr.str("");
    ostr << "Nearest distance: " << nearest_distance;
    console->print(ostr.str(), y++);

    ostr.str("");
    ostr << "Leading target: " << target_lead.getX() << "," << target_lead.getY();
    console->print(ostr.str(), y++);

    ostr.str("");
    ostr << "Angle to target: " << ship_angle;
    console->print(ostr.str(), y++);
}

void addAsteroid()
{
    //add an asteroid
    Sprite *asteroid = new Sprite();
    asteroid->setObjectType(OBJECT_ASTEROID);
    asteroid->setVelocity(-ASTEROID_VEL, 0);
    asteroid->setPosition(SCREENW,50+rand()%(SCREENH-150));
    asteroid->setImage(asteroid_image);
    asteroid->setTotalFrames(64);
    asteroid->setColumns(8);
    asteroid->setSize(60,60);
    asteroid->setFrameTimer( rand() % 100 );
    asteroid->setCurrentFrame( rand() % 64 );
    if (rand()%2==0) asteroid->setAnimationDirection(-1);
    g_engine->addEntity( asteroid );
}

void firebullet()
{
    //get the ship from the entity manager
    Sprite *ship = (Sprite*)g_engine->findEntity(OBJECT_SHIP);
    if (!ship)
    {
        g_engine->message("Error locating ship in entity manager!","ERROR");
        g_engine->Close();
    }

    //create bullet sprite
    Sprite *bullet = new Sprite();
    bullet->setObjectType(OBJECT_BULLET);
    bullet->setImage(bullet_image);
    bullet->setMoveTimer(1);
    bullet->setLifetime(5000);
    //set bullet equal to ship's rotation angle
    float angle = g_engine->math->toRadians(ship_angle);
    bullet->setRotation( angle );

    //set bullet's starting position
    float x = ship->getX() + ship->getWidth()/2;
    float y = ship->getY() + ship->getHeight()/2-8;
    bullet->setPosition(x,y);

    //set bullet's velocity
    float vx = g_engine->math->LinearVelocityX(ship_angle) * BULLET_VEL;
    float vy = g_engine->math->LinearVelocityY(ship_angle) * BULLET_VEL;
    bullet->setVelocity(vx, vy);

    //fire bullet
    g_engine->addEntity(bullet);
    g_engine->audio->Play("fire");
}


void targetNearestAsteroid(Sprite *asteroid)
{
    //get asteroid's position
    Vector3 target = asteroid->getPosition();

    //calculate distance to target
    float dist = ship_position.Distance( target );
    if (dist < nearest_distance) {
        nearest_asteroid = target;
        nearest_distance = dist;

        //lead the target for better accuracy
        target_lead.setX(asteroid->getVelocity().getX() * 0.01f);
        target_lead.setY(asteroid->getVelocity().getY() * 0.01f);
        nearest_asteroid.setX(nearest_asteroid.getX() + target_lead.getX());
        nearest_asteroid.setY(nearest_asteroid.getY() + target_lead.getY());

        //calculate angle to target
        ship_angle = g_engine->math->AngleToTarget(ship_position,nearest_
asteroid);
        ship_angle = 90 + g_engine->math->toDegrees( ship_angle );

    }
    //is there a target to shoot at?
    if (nearest_distance < 1200) {
        if (fireTimer.stopwatch(100)) {
            firebullet();
        }
    }
}

void game_update()
{
    updateConsole();
}

void game_render2d()
{
    font->Print(1,SCREENH-20,"Press ~ or F12 to toggle the Console");
    font->Print(1,SCREENH-40,"Press SPACE to launch an asteroid!!!");
    if (console->isShowing()) console->draw();
    nearest_distance = 999999;
}

void game_end()
{
    delete console;
    delete font;
    delete bullet_image;
    delete asteroid_image;
    delete explosion_image;
}

void game_entityUpdate(Advanced2D::Entity* entity)
{
    float y;
    Sprite *ship, *bullet, *asteroid;
    Vector3 position;

    switch(entity->getObjectType())
    {
        case OBJECT_SHIP:
            ship = (Sprite*)entity;
            ship_position = ship->getPosition();
            ship->setRotation( g_engine->math->toRadians(ship_angle) );
            break;
        case OBJECT_BULLET:
            bullet = (Sprite*)entity;
            if (bullet->getX() > SCREENW)
                bullet->setAlive(false);
            break;
        case OBJECT_ASTEROID:
            asteroid = (Sprite*)entity;
            if (asteroid->getX() < -64) asteroid->setX(SCREENW);
            targetNearestAsteroid( asteroid );
            break;
    }
}


void game_entityCollision(Advanced2D::Entity* entity1,Advanced2D::Entity*
entity2)
{
    if (entity1->getObjectType() == OBJECT_ASTEROID)
    {
        Sprite *asteroid = (Sprite*)entity1;

        if (entity2->getObjectType() == OBJECT_BULLET)
        {
            //create an explosion
            Sprite *expl = new Sprite();
            expl->setObjectType(OBJECT_EXPLOSION);
            expl->setImage(explosion_image);
            expl->setColumns(6);
            expl->setCollidable(false);
            expl->setSize(128,128);
            float x = asteroid->getPosition().getX();
            float y = asteroid->getPosition().getY();
            expl->setPosition(x-32,y-32);
            expl->setTotalFrames(30);
            expl->setFrameTimer(40);
            expl->setLifetime(1000);
            g_engine->addEntity( expl );

            //remove the asteroid
            entity2->setAlive(false);
```

```
                //remove the bullet
                entity1->setAlive(false);

                //play explosion sound
                g_engine->audio->Play("boom");
            }
        }
    }

void game_keyPress(int key)
{
    switch (key) {
        case DIK_SPACE:
            addAsteroid();
            break;
    }
}

void game_keyRelease(int key)
{
    switch (key) {
        case DIK_ESCAPE:
            g_engine->Close();
            break;
        case DIK_F12:
        case DIK_GRAVE:
            console->setShowing( !console->isShowing() );
            break;
    }
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));
}

void game_mouseButton(int button) { }
void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y) { }
void game_mouseWheel(int wheel) { }
void game_entityRender(Advanced2D::Entity* entity) { }
```

Drop-Down Console

The drop-down console featured in the example programs in this and the previous chapter is in need of some explanation because we've just ignored it so far. The first crude console was introduced two chapters ago, while the current one you see here was actually built in the previous chapter; I reserved a study of the `Console` class until now. The `Console` class has nothing to do with math, but we need to go over it now for reference.

So, what is the console all about anyway?

A *console window* is a non-intrusive way to communicate with the developer—or the game's player. The console may be toggled on or off with a key or key combination. Some consoles have an input prompt that allows the user to type in commands to interact with the engine—to change settings, run debugging scripts, load game levels, and so forth.

Our console does not have an input prompt, although you are welcome to add one. At this stage, the console only needs to display information, so that is all it does. The console does have its own list of strings that may be used to print text with the System12 font. These text lines may be printed sequentially, with auto line feed, or a specific line may be printed manually.

Console Class

Here is the header file Console.h that has been included in the Engine project:

```
class Console {
private:
    bool showing;
    Sprite *panel;
    Font *font;
    int currentLine;
    std::vector<std::string> textlines;
    std::vector<std::string>::iterator iter;
public:
    Console();
    virtual ~Console();
    bool init();
    void draw();
    void clear();
    void print(std::string text, int line = -1);
    bool isShowing() { return this->showing; }
    void show() { this->showing = true; }
    void hide() { this->showing = false; }
    void setShowing(bool value) { this->showing = value; }
};
```

The implementation file Console.cpp is next. Note how the console automatically loads an image that is rendered with alpha transparency onto the game screen. This adds a dependency to all games that use the engine—the panel.tga file must be included. But that is to be expected; once an engine begins to incorporate new features, additional dependencies must be provided (such as font image and data files).

```
#include "Advanced2D.h"
namespace Advanced2D {
    Console::Console()
    {
        showing = false;
        currentLine = 0;
        clear();
    }

    Console::~Console()
    {
        delete font;
        delete panel;
    }

    bool Console::init()
    {
        //load the panel image
        panel = new Sprite();
        if (!panel->loadImage("panel.tga")) return false;
        double scale = g_engine->getScreenWidth() / 640.0f;
        panel->setScale(scale);
        panel->setColor(0x99FFFFFF);

        //load the font
        font = new Font();
        if (!font->loadImage("system12.tga")) return false;
        font->setColumns(16);
        font->setCharSize(14,16);
        if (!font->loadWidthData("system12.dat")) return false;
        return true;
    }

    void Console::draw()
    {
        int x = 5, y = 0;
        if (!showing) return;
        //draw panel background
        panel->draw();
        //draw text lines
        for (unsigned int n = 0; n < textlines.size(); n++)
        {
            font->Print(x,y*14, textlines[n], 0xFF000000);
            y += 1;
            if (y > 26) {
                if (x > 10) break;
                 x = g_engine->getScreenWidth()/2 + 5;
                 y = 0;
            }
        }
    }

    void Console::print(std::string text, int line)
    {
        if (line > -1) currentLine = line;
        textlines[currentLine] = text;
        if (currentLine++ > 52) currentLine = 0;
    }

    void Console::clear()
    {

        for (int n=0; n<55; n++)
            textlines.push_back("");
    }
};
```

Console Test

The ConsoleDemo program shows the console over a simulation of—you guessed it—asteroids. Figure 10.4 shows the program running normally without the console, demonstrating how the console can be used to reduce the clutter of debug text that normally fills a game screen during development. Figure 10.5 shows the console displayed over the ‘‘game’’ screen. Note the alpha blending with the sprite objects underneath. The console can be rendered with more or less alpha by just modifying the color used to render the console image.

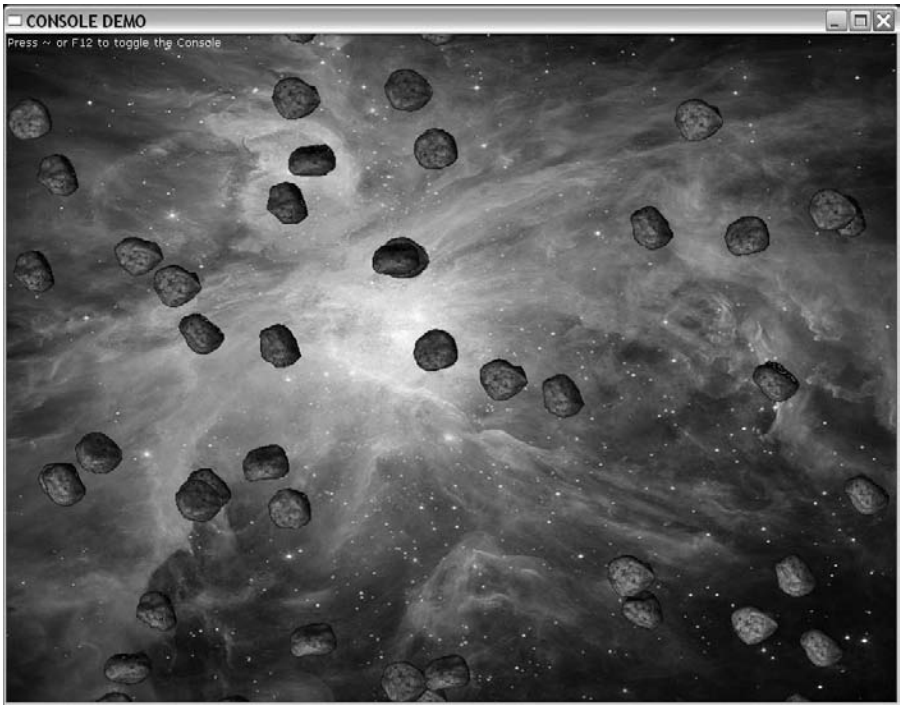
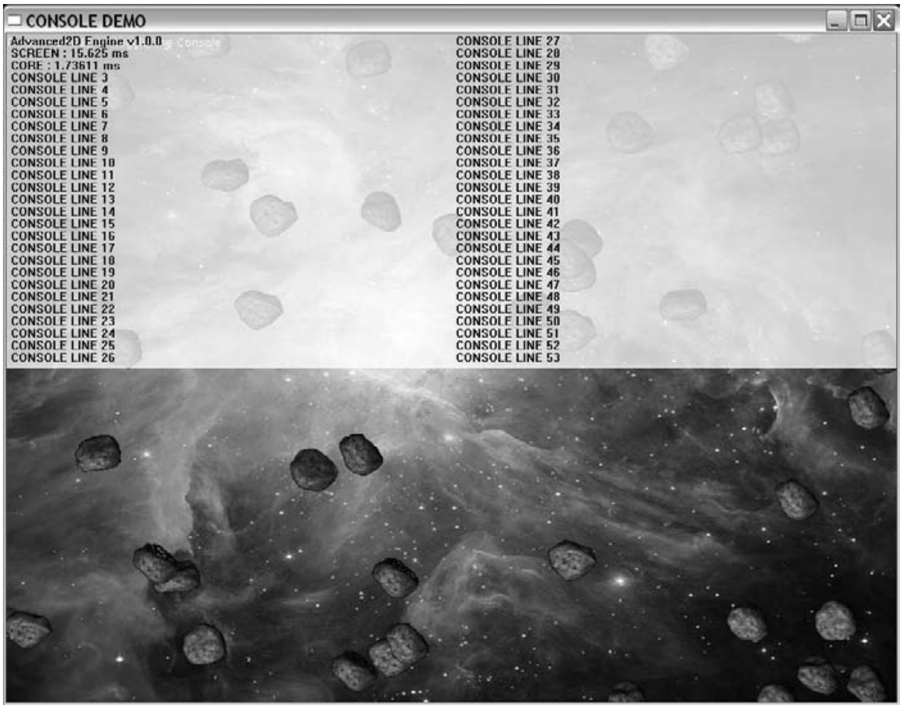


Figure 10.4
The game screen that is normally seen.



```
Advanced2D Engine v1.0.0        CONSOLE LINE 27
SCREEN : 15.625 ms              CONSOLE LINE 28
CORE : 1.73611 ms               CONSOLE LINE 29
CONSOLE LINE 3                  CONSOLE LINE 30
CONSOLE LINE 4                  CONSOLE LINE 31
CONSOLE LINE 5                  CONSOLE LINE 32
CONSOLE LINE 6                  CONSOLE LINE 33
CONSOLE LINE 7                  CONSOLE LINE 34
CONSOLE LINE 8                  CONSOLE LINE 35
CONSOLE LINE 9                  CONSOLE LINE 36
CONSOLE LINE 10                 CONSOLE LINE 37
CONSOLE LINE 11                 CONSOLE LINE 38
CONSOLE LINE 12                 CONSOLE LINE 39
CONSOLE LINE 13                 CONSOLE LINE 40
CONSOLE LINE 14                 CONSOLE LINE 41
CONSOLE LINE 15                 CONSOLE LINE 42
CONSOLE LINE 16                 CONSOLE LINE 43
CONSOLE LINE 17                 CONSOLE LINE 44
CONSOLE LINE 18                 CONSOLE LINE 45
CONSOLE LINE 19                 CONSOLE LINE 46
CONSOLE LINE 20                 CONSOLE LINE 47
CONSOLE LINE 21                 CONSOLE LINE 48
CONSOLE LINE 22                 CONSOLE LINE 49
CONSOLE LINE 23                 CONSOLE LINE 50
CONSOLE LINE 24                 CONSOLE LINE 51
CONSOLE LINE 25                 CONSOLE LINE 52
CONSOLE LINE 26                 CONSOLE LINE 53
```

Figure 10.5
The game screen with the console drawn over the top with alpha blending and text output.

That wraps up our math chapter! You now have the tools you need to make a complete 2D game using Direct3D and the other libraries we've explored in the book thus far. Next we'll get into a bit of theory by studying multi-threading in the next chapter, and after that, scripting.

This page intentionally left blank

CHAPTER 11

Threading

Today's modern processors come with multiple cores, each of which runs independently to run programs and significantly increase the throughput compared to a single-core processor. The clock speed is no longer the most important factor, because a quad-core processor will outperform most dual-core processors even if there is a clock speed discrepancy. In this chapter you will learn how to create threads using the POSIX Threads library—a popular cross-platform library that is part of the core Linux operating system (and available for Windows as well).

A thread is a set of instructions, usually in a loop, that runs in parallel with other sets of instructions (or threads) in a program. In a multitasking operating system, every program has at least one thread—itself—because the operating system breaks down every running process into one or more threads that may take advantage of dual-core or multiple processors in a computer system.

Introducing the POSIX Threads Library

Every modern operating system uses threads for essential and basic operation and would not be nearly as versatile without threads. A *thread* is best described as a function that runs within the memory space of a program but is executed in parallel. This section will provide a short overview of multi-threading and how it can be used (fairly easily) to enhance a game. I will not go into the vast details of threaded programming because the topic is too huge and unwieldy to fully explain in only a few pages. Instead, I will provide you with enough information and example code that you will be able to start using threads.

To be multi-threaded, a program will create at least one thread that will run in addition to that program's main loop. Any time a program uses more than one thread, you must take extreme caution when working with data that is potentially shared between threads. It is generally safe for a program to share data with a single thread (although it is not recommended), but when more than one thread is in use, you must use a protection scheme to protect the data from being manipulated by two threads at the same time.

To protect data, you can make use of mutexes that will lock data inside a single thread until it is safe to unlock the data for use in the main program or in another thread. The locking and unlocking must be done inside a loop that runs continuously inside the thread callback function. Note that if you do not have a loop inside your thread function, it will run once and terminate. The idea is to keep the thread running—doing something—while the main program is doing the delegating work. You should think of a thread as a new employee who has been hired to alleviate the amount of work done by the program (or rather, by the main thread).

We disseminate the subject as if it's just another C function, but threads were at one time an extraordinary achievement that was every bit as exciting as the first connection in ARPAnet in 1969 or the first working version of UNIX. In the 1980s, parallel programming was as hip as virtual reality, but like the latter, it was not to be a true reality until the early 1990s. *Multi-threaded programming* is the engineers' term for parallel processing and is a solution that has been proven to work. The key to parallel processing came when software engineers realized that the processor is not the focus; rather, software design is. In the words of Agent Smith from *The Matrix,* "We lacked a programming language with which to construct your world."

A single-processor system should be able to run multiple threads. Once that goal was realized, adding two or more processors to a system provided the ability to delegate those threads, and this was a job for the operating system. No longer tasked with designing a parallel-processing architecture, engineers in both the electronics and software fields abstracted the problem so the two were not reliant upon each other. A single program can run on a motherboard with four CPUs and push all of those processors to the limit—if that single program invokes multiple threads. As such, the programs themselves were treated as single threads. And yet, there can be many non-threaded programs running on our fictional quad-processor system, and it might not be taxed at all. It depends on what each program is doing.

Math-intensive processes, such as 3D rendering, can eat a CPU for breakfast. But with the advent of threading in modern operating systems, programs such as 3D Studio Max, Maya, LightWave, and Photoshop can invoke threads to handle intense processes, such as scene rendering and image manipulation. Suddenly, that dual-G5 Mac is able to process a Photoshop image in four seconds, whereas it took 45 seconds on your G3 Mac! Why? Threads and multiple processor cores.

However, just because a single program is able to share four CPUs, that doesn't mean each thread is an independent entity. Any global variables in the program (main thread) can be used by the invoked threads as long as care is taken that data is not damaged. Imagine 10 children grasping for an ice cream cone at the same time and you get the picture. What your threaded program must do is isolate the ice cream cone for each child, and only make the ice cream cone available to the others after that child has released it. Get the picture?

How does this concept of threading relate to processes? As you know, modern operating systems treat each program as a separate process, allocating a certain number of milliseconds to each process. This is where you get the term *multitasking;* many processes can be run at the same time using a time-slicing mechanism. A process owns a heap (the thing used for global variables and dynamically allocated memory via `new` or `malloc`). The process heap is shared by all the threads in the process. Each thread in the process (the main thread and any additional worker threads you start) gets its own stack (used for local variables).

The vast majority of Linux and UNIX operating system flavors will already have the pthread library installed because it is a core feature of the kernel. Other systems might not be so lucky. Windows uses its own multi-threading library. Of course, a primary goal of this book is to keep this code 100-percent portable. So what you need is a pthread library that is compatible with the POSIX systems. After all, that is what the "p" in pthread stands for—POSIX threads. An important thing you should know about the Windows implementation of pthread is that it abstracts the Windows threading functionality, molding it to conform to pthread standards.

Installing POSIX Threads

Although Red Hat's pthread library is open source, I have chosen not to distribute it with the book and have only included the libs, dlls, and key headers. You can download the pthread library and find extensive documentation at http://sources.redhat.com/pthreads-win32. I encourage you to browse the site and get the latest version of Pthreads-Win32 from Red Hat. The pthreads library is composed of three header files, a library file, and a DLL runtime file (which must be distributed with the program's executable). The three header files are:

- pthread.h
- sched.h
- semaphore.h

These files should be copied to your compiler's .\include folder for best results. Optionally, you can add a folder to your compiler's include path so that it can find the pthread headers (wherever you have copied them to your hard drive). Due to the way the pthread headers are defined, you cannot include them locally—they must be referenced globally by your compiler.

Second, you must copy the pthread library file into your compiler's .\lib folder, or add a library path to your compiler. For Dev-C++, the library file is called libpthreadGC.a, and you will add it as a linker option using `-lpthreadGC`. For Visual C++, the library file is called pthreadVC.lib, and you will add it to the list of additional dependencies by its filename: pthreadGC.lib.

Third, to run a program compiled with the pthread library, you must include the runtime DLL file. For Dev-C++, the file is pthreadGC.dll. For Visual C++, the file is pthreadVC.dll. The library file just provides a link between your program and the DLL, where the compiled pthread code is located and linked into your program at runtime. You will need to distribute the pthreadxx.dll file just as you must include the fmodex.dll file for FMOD audio support. If you're a Dev-C++ user, you must also provide the custom d3dx9.dll for Direct3D support too (although this file is installed with the normal DirectX runtime when you build with Visual C++).

Advice

On the CD-ROM under \libraries, you will find a folder called .\pthreads that includes ready-to-use headers, libs, and DLLs for Dev-C++ and Visual C++. You may want to install these files in your compiler's install folder in order to use the Pthreads-Win32 library.

Using POSIX Threads

There are two ways to use threads to offload processing from your game loop. The first method is to write a thread function that runs once and then returns.

The second method is to write a thread function with its own `while` loop that runs continually in parallel with your game loop. There are advantages and drawbacks to both methods. The single-run function method uses more processor cycles because the thread function is being called many times per second, but it will result in fewer mutex waits (which happen when the thread is locked by another process). The continually running function with its own `while` loop is more efficient because it is only called once in order to run in parallel, but the drawbacks are less versatility and more instances of mutex waits. Neither method is better than the other, as both types of thread function will be useful in a game. I tend to favor the single-run thread functions over the embedded loop function method, if only because it allows for smaller, more mission-specific functions. There's no reason why you cannot write *many* single-purpose thread functions that run once depending on the conditions in the game.

Let me give you some examples to help you visualize both scenarios. First, you have a game that creates a thread before launching its own main loop. Inside the thread function you have programmed it to update all of the sprites in the entity manager. Since the entities are created on the heap with *new,* each entity in the list is really just a pointer. Thus, iterating through the entity list means we go through a list of pointers to gain access to each mesh or sprite object in the list. The thread function runs in a tight loop with no timing whatsoever, so it runs *really* fast. In your game loop, however, each time an entity is updated, there is a call to the `game_entityUpdate` function in your game's source code, and a pointer to the entity is passed to this function each time. Now, if your tightly written thread loop tried to read data in a specific sprite while the `game_entityUpdate` function was *writing* data to the same sprite, that would crash the whole program—or at best, lock up the thread due to the mutex, which would cause the game to freeze as if it were paused.

Let's take a look at this scenario from the single-run thread function point of view. In the engine, the entity list is iterated and a thread function is called to update each entity (with movement, animation, collision detection, and so forth). But *now,* most of that processing is being called from the game loop, not from the thread loop. Whenever we need to update a sprite, a thread function is called, and when that update is finished, the thread is terminated. Our game experiences quite a bit of overhead with all of the function calls, but the advantage is that now we can update an entity in `game_entityUpdate` or `game_entityCollision` without causing a mutex lock. How? If, inside one of the single-run thread functions, we experience a thread lock, that thread will wait until the lock is released, and it will then finish its processing and kill the thread. However, in a tight thread loop, the mutex in the *main program* could be locked instead! This could potentially lock up the game loop. Although the engine's threads would continue to run just fine (hogging the system, so to speak), the game loop that communicates by way of our event functions (`game_keyPress` and so forth) would be interrupted. We can predict this because the game loop has *timing code* in it. That timing code means the game loop can be easily interrupted. The thread loop will have no such timing code, because it is designed to run as fast as possible.

As you can imagine, a lot of thought must be put into a multi-threaded game engine before we just haphazardly create a tight thread function at engine startup and then *assume* that, with our newfound threading power, the engine will run faster. In reality, the thread locks are probably *slowing it down*!

Returning to the subject of single-run thread functions, there is another drawback that I have not mentioned yet. When you create a new thread, the point in the program where the thread was created *continues* to the next statement. The program doesn't need to wait until the thread function returns before it executes the next line. This means we can actually create many threads simultaneously, with each one updating a single object in memory without conflict. This is inevitably faster than a monolithic thread function *if* we have a multi-core processor, since a looping thread will only utilize a single core. Our multiple-thread function call theory would utilize multiple cores, since the operating system decides where threads are run and balances them among all available processor cores.

The drawbacks seem to outweigh the advantages to the single-run threads. Since the threads are being created and destroyed thousands of times per second, the overhead will be high, outweighing any advantages we would otherwise gain by supporting any number of processor cores. In addition, creating and destroying threads repeatedly can cause some instability in the framerate of the game loop, making it difficult to maintain a smooth and reliable core. Due to these issues, we will use a single thread function—but we can create more than one if we need to.

Programming POSIX Threads

I am going to cover the key functions in this section and let you pursue the full extent of multi-threaded programming on your own using the references I have suggested.

Creating a New Thread

First of all, how do you create a new thread? New threads are created with the `pthread_create` function.

```
int pthread_create (
    pthread_t *tid,
    const pthread_attr_t *attr,
    void *(*start) (void *),
    void *arg);
```

Yeah! That's what I thought at first, but it's not a problem. Here, let me explain. The first parameter is a `pthread_t` struct variable. This struct is large and complex, and you really don't need to know about the internals to use it. ("Ignorance is bliss," to quote Cipher from *The Matrix*.) If you want more details, I encourage you to pick up Butenhof's book *Programming with POSIX Threads* (Addison-Wesley Professional, 1997) as a reference.

The second parameter is a `pthread_attr_t` struct variable that usually contains attributes for the new thread. This is usually not used, so you can pass null to it.

The third parameter is a pointer to the thread function used by this thread for processing. This function should contain its own loop, but should have exit logic for the loop when it's time to kill the thread. (I use a `done` variable.)

The fourth parameter is a pointer to a numeric value for this thread to uniquely identify it. You can just create an `int` variable and set it to a value before passing it to `pthread_create`.

Here's an example of how to create a new thread:

```
int id;
pthread_t pthread0;
int threadid0 = 0;
id = pthread_create(&pthread0, NULL, thread0, (void*)&threadid0);
```

So you've created this thread, but what about the callback function? Oh, right. Here's an example:

```
void* thread0(void* data)
{
    int my_thread_id = *((int*)data);
    while(!done)
    {
        //do something!
    }
    pthread_exit(NULL);
    return NULL;
}
```

Killing a Thread

This brings us to the `pthread_exit` function, which terminates the thread. Normally you'll want to call this function at the end of the function, after the loop has exited. Here's the definition for the function:

```
void pthread_exit (void *value_ptr);
```

You can get away with just passing null to this function because `value_ptr` is an advanced topic for gaining more control over the thread.

Mutexes: Protecting Data from Threads

At this point you can write a multi-threaded program with only the `pthread_ create` and `pthread_exit` functions, knowing how to create the callback function and use it. That is enough if you only want to create a single thread to run inside the process with your program's main thread. But more often than not, you will want to use two or more threads in a game to delegate some of the workload. Therefore, it's a good idea to use a mutex for all your threads. Recall the ice cream cone analogy. Are you sure that new thread won't interfere with any globals? Have you considered timing? What if you are using a thread for rendering while another thread is writing to the back buffer? Most memory chips cannot read and write data at the same time. It is very likely is that you'll update a small portion of the buffer (by drawing a sprite, for instance) while the buffer is being blitted to the screen. The result is some unwanted flicker—yes, even when using a double buffer. What you have here is a situation that is similar to a vertical refresh conflict, only it is occurring in memory rather than directly on the screen. What I am trying to point out is that threads can step on each other's toes, so to speak, if you aren't careful to use a mutex.

A *mutex* is a block used in a thread function to prevent other threads from running until that block is released. Assuming, of course, that all threads use the same mutex, it is possible to use more than one mutex in your program. The easiest way is to create a single mutex, and then block the mutex at the start of each thread's loop, unblocking at the end of the loop. Creating a mutex doesn't require a function; rather, it requires a struct variable. In our simplistic approach here, I'm using only a single mutex for the entire program, but in practice you would want to use many mutexes.

```
//create a new thread mutex to protect variables
pthread_mutex_t threadsafe = PTHREAD_MUTEX_INITIALIZER;
```

This line of code will create a new mutex called `threadsafe` that, when used by all the thread functions, will prevent data read/write conflicts. You must destroy the mutex before your program ends; you can do so using the `pthread_mutex_destroy` function.

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Here is an example of how it would be used:

```
pthread_mutex_destroy(&threadsafe);
```

Next, you need to know how to lock and unlock a mutex inside a thread function. The `pthread_mutex_lock` function is used to lock a mutex.

```
int pthread_mutex_lock (pthread_mutex_t * mutex);
```

This has the effect of preventing any other threads from locking the same mutex, so any variables or functions you use or call (respectively) while the mutex is locked will be safe from manipulation by any other threads. Basically, when a thread encounters a locked mutex, it waits until the mutex is available before proceeding. (It uses no processor time; it simply waits.)

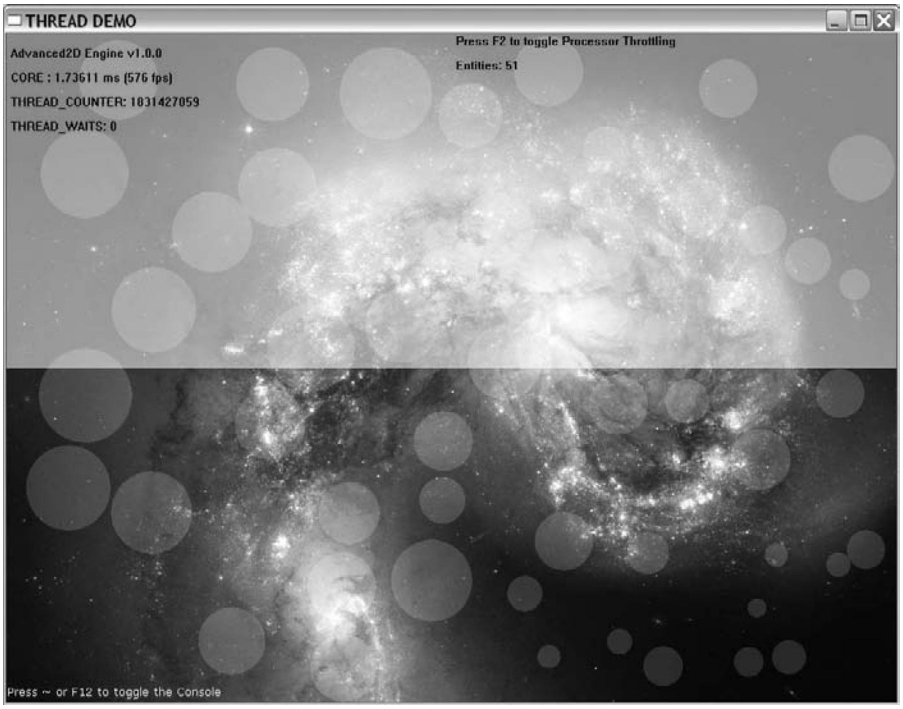
Here is the unlock function:

```
int pthread_mutex_unlock (pthread_mutex_t * mutex);
```

The two functions just shown will normally return zero if the lock or unlock succeeded immediately; otherwise, a non-zero value will be returned to indicate that the thread is waiting for the mutex. This should not happen for unlocking, only for locking. If you have a problem with `pthread_mutex_unlock` returning non-zero, it means the mutex was locked while that thread was supposedly in control over the mutex—a bad situation that should never happen. But when it comes to game programming, bad things do often happen while you are developing a new game, so it's helpful to print an error message for any non-zero return.

ThreadDemo Program

We need an example to see how threading works and to see how the pthread library is used. The ThreadDemo program (shown in Figure 11.1) includes all of



```
Advanced2D Engine v1.0.0
CORE : 1.73611 ms (576 fps)
THREAD_COUNTER: 1831427059
THREAD_WAITS: 0
Press F2 to toggle Processor Throttling
Entities: 51
Press ~ or F12 to toggle the Console
```

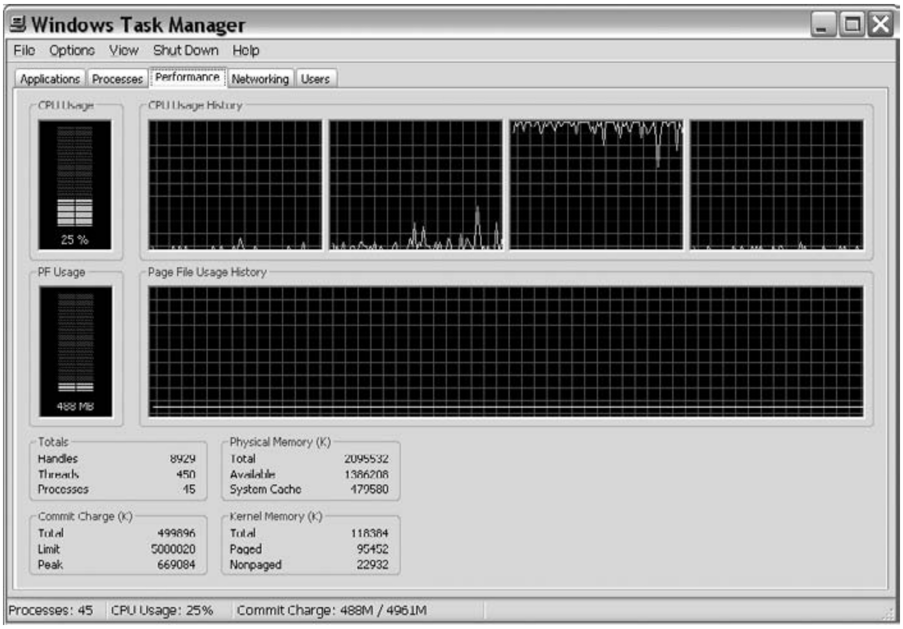
Figure 11.1
The ThreadDemo program increments a variable in a separate thread.

the thread code directly, outside of the engine, so you can learn from it. This program draws translucent sprites over the background just as a visual cue that the game loop is not being affected by the threads. We want to make sure threading does not screw up the framerate, and these circle sprites help in that regard. Afterward, we will incorporate threads into the engine and run some punishing tests (by creating thousands of sprites) to see whether there's a performance gain.

Advice

A game built with our Advanced2D library will normally use seven threads already due to its libraries. Use this number as a baseline when you examine your program running in Task Manager.

Figure 11.2 shows the Task Manager while the ThreadDemo program is running. Surprisingly enough, the game engine is actually running on core #2 with the engine set to minimal processor usage (with a core update time of 1.7 ms). The third core, which is maxed out, is where the thread function is running! The



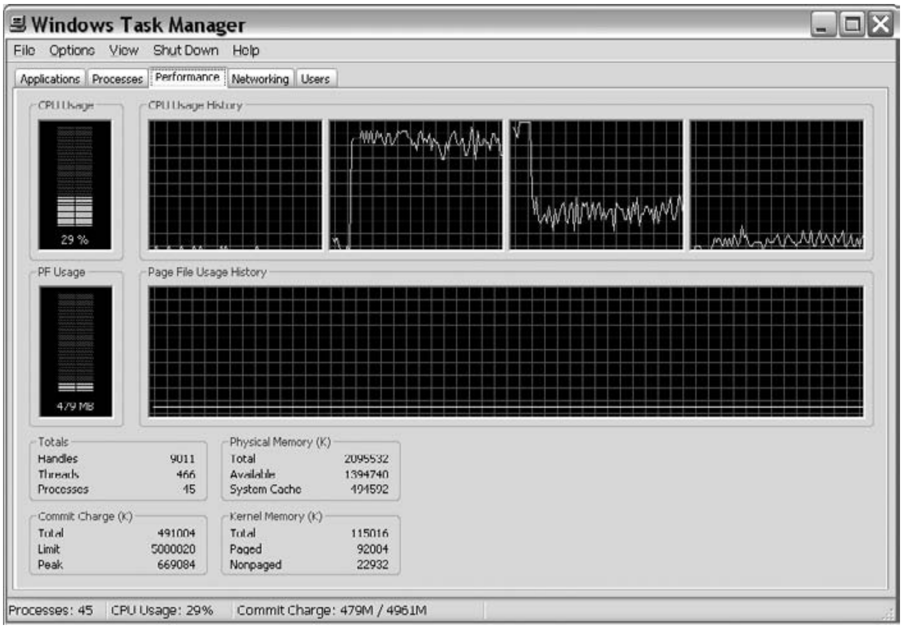
```
Totals                      Physical Memory (K)
Handles          8929       Total          2095532
Threads           450       Available      1386208
Processes          45       System Cache    479580

Commit Charge (K)           Kernel Memory (K)
Total          499896       Total           118384
Limit         5000020       Paged            95452
Peak           669084       Nonpaged         22932

Processes: 45 | CPU Usage: 25% | Commit Charge: 488M / 4961M
```

Figure 11.2
Processor core usage in Task Manager while ThreadDemo is running.

thread is running ludicrously fast because all it has to do is increment a variable. If you want to really see a better balance between the two cores, you would need to add some load to the thread function (by performing some higher math calculations, such as `Distance` or `AngleToTarget`).

By pressing F2 to turn off processor throttling, we can maximize the speed of the core game loop—up from 1.7 ms to 0.03 ms (a six-fold increase that results from skipping the `Sleep(1)` line in the core)—which has been built into the engine all along, so that's nothing new. The interesting thing is that the thread counter does not speed up or slow down based on the core throttling—it just continues to increment at a steady (but insanely fast) rate. Our engine core is running at about 33,000 fps, but the rate is only that fast because we are doing next to nothing in this demo—just moving a few sprites around. Add some mesh objects and lighting and a few hundred entities, and the rate would drop significantly.

Now let's take a look at the Task Manager while the ThreadDemo program is running with a maximized core. As Figure 11.3 shows, the second core is now under load, as that is where our engine core is now running at full speed. The third



```
Totals                    Physical Memory (K)
Handles         9011      Total          2095532
Threads          466      Available      1394740
Processes         45      System Cache    494592

Commit Charge (K)         Kernel Memory (K)
Total         491004      Total           115016
Limit        5000020      Paged            92004
Peak          669084      Nonpaged         22932

Processes: 45  CPU Usage: 29%  Commit Charge: 479M / 4961M
```

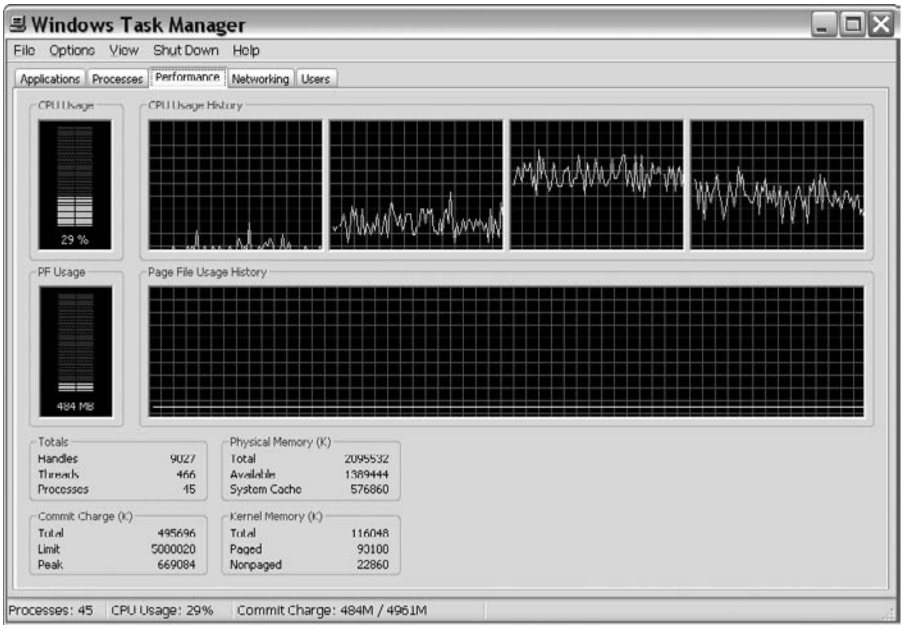
Figure 11.3
The second core is under load, and the third core has dropped to about 25 percent.

core, though, has dropped down to about 25 percent. What could account for the drop? I might have expected both cores to be running at peak now. There are no mutex locks occurring because the program keeps track of any locks (and it is showing up at zero).

Advice

The background image used in the ThreadDemo program is titled "The Antennae Galaxies/NGC 4038-4039," photographed by Hubble Space Telescope. This image was made available by NASA-STScI at www.hubblesite.org.

This is really just an artifact of the operating system assigning tasks to the processor. If you create more threads, you will see that they are balanced more equitably on a multi-core processor, as shown in Figure 11.4. In this example, processor throttling is turned off, and the engine core is running at 0.04 ms (about 40 microseconds—millionths of a second).



Totals
Handles 9027
Threads 466
Processes 45

Physical Memory (K)
Total 2095532
Available 1389444
System Cache 576860

Commit Charge (K)
Total 495696
Limit 5000020
Peak 669084

Kernel Memory (K)
Total 116048
Paged 93100
Nonpaged 22860

Processes: 45 CPU Usage: 29% Commit Charge: 484M / 4961M

Figure 11.4
More threads are balanced more equitably on a multi-core processor.

Here is the source code for the ThreadDemo program, which is on the CD-ROM in the \sources\ch11 folder.

```
#include <pthread.h>
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

#define SCREENW 1024
#define SCREENH 768
#define OBJECT_BACKGROUND 1
#define OBJECT_SPRITE 100
#define MAX 50
#define SCALE 70

Texture *circle_image;
Font *font;
Console *console;
std::ostringstream ostr;
int collisions;
```

```
//THREAD STUFF
#define MAXTHREADS 2
bool done = false;
unsigned long thread_counter = 0;
void* thread_function(void* data);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int thread_waits = 0;

void* thread_function(void* data)
{
    while(!done)
    {
        //lock the mutex
        if (pthread_mutex_lock(&mutex) != 0) thread_waits++;

        //increment thread counter
        thread_counter++;

        //do something more intensive
        Vector3 vector1(100,200,300);
        Vector3 vector2(400,500,600);
        double dist = g_engine->math->Distance(vector1,vector2);
        double dist_squared = dist*dist;
        double square_root_of_dist = sqrt(dist);
        double answer = square_root_of_dist;

        //unlock the mutex
        if (pthread_mutex_unlock(&mutex) != 0) thread_waits++;
    }
    pthread_exit(NULL);
    return NULL;
}

bool game_preload()
{
    g_engine->setAppTitle("THREAD DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(SCREENW);
    g_engine->setScreenHeight(SCREENH);
    g_engine->setColorDepth(32);
    return 1;
}
```

```
bool game_init(HWND)
{
    int n;
    //load background image
    Sprite *background = new Sprite();
    if (!background->loadImage("galaxies.tga")) {
        g_engine->message("Error loading galaxies.tga");
        return false;
    }
    background->setObjectType(OBJECT_BACKGROUND);
    background->setCollidable(false);
    g_engine->addEntity(background);

    //create the console
    console = new Console();
    if (!console->init()) {
        g_engine->message("Error initializing console");
        return false;
    }
    console->setShowing(true);

    //load sprite image
    circle_image = new Texture();
    if (!circle_image->Load("circle.tga")) {
        g_engine->message("Error loading circle.tga");
        return false;
    }

    //create sprites
    Sprite *sprite;
    for (n=0; n < MAX; n++)
    {
        //create a new sprite
        sprite = new Sprite();
        sprite->setObjectType(OBJECT_SPRITE);
        sprite->setImage(circle_image);
        sprite->setColor(D3DCOLOR_RGBA(255,255,255,50));
        sprite->setSize(128,128);
        sprite->setScale( (float)(rand() % SCALE + SCALE/4) / 100.0f );
        sprite->setPosition( rand() % SCREENW, rand() % SCREENH );
        sprite->setCollisionMethod(COLLISION_DIST);

        //set velocity
        float vx = (float)(rand()%30 - 15)/10.0f;
        float vy = (float)(rand()%30 - 15)/10.0f;
        sprite->setVelocity( vx, vy );

        //add sprite to the entity manager
        g_engine->addEntity(sprite);
    }

    //load the Verdana10 font
    font = new Font();
    if (!font->loadImage("verdana10.tga")) {
        g_engine->message("Error loading verdana10.tga");
        return false;
    }
    if (!font->loadWidthData("verdana10.dat")) {
        g_engine->message("Error loading verdana10.dat");
        return false;
    }
    font->setColumns(16);
    font->setCharSize(20,16);

    //create the thread(s)
    int id;
    for (n = 0; n < MAXTHREADS; n++) {
        pthread_t mythread;
        int mythread_id = n;
        id = pthread_create(&mythread, NULL, thread_function, (void*)&mythread_ id);
    }

    return true;
}

void updateConsole()
{
    int y = 0;
    console->print(g_engine->getVersionText(), 1);
    ostr.str("");
    ostr << "CORE : " << (float)(1000.0f/g_engine->getFrameRate_core())
         << " ms (" << g_engine->getFrameRate_core() << " fps)";
    console->print(ostr.str(), 3);
    ostr.str("");
    ostr << "THREAD_COUNTER: " << thread_counter;
    console->print(ostr.str(), 5);
    ostr.str("");
    ostr << "THREAD_WAITS: " << thread_waits;
    console->print(ostr.str(), 7);
    console->print("Press F2 to toggle Processor Throttling", 27);
    ostr.str("");
    ostr << "Entities: " << g_engine->getEntityCount();
    console->print(ostr.str(), 29);
}

void game_update()
{
    //any code that touches thread variables must be wrapped in a mutex
    pthread_mutex_lock(&mutex);
    updateConsole();
    collisions = 0;
    pthread_mutex_unlock(&mutex);
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));
}

void game_render2d()
{
    font->Print(1,SCREENH-20,"Press ~ or F12 to toggle the Console");
    if (console->isShowing()) console->draw();
}

void game_keyRelease(int key)
{
    switch (key) {
        case DIK_ESCAPE:
            g_engine->Close();
            break;
        case DIK_F12:
        case DIK_GRAVE:
            console->setShowing( !console->isShowing() );
            break;
        case DIK_F2:
            g_engine->setMaximizeProcessor( !g_engine->getMaximizeProcessor() );
            break;
    }
}
```

```
void game_end()
{
    //kill the persistent thread
    done = true;
    pthread_mutex_destroy(&mutex);

    //delete objects
    delete console;
    delete circle_image;
    delete font;
}
void game_entityUpdate(Advanced2D::Entity* entity)
{
    switch(entity->getObjectType())
    {
        case OBJECT_SPRITE:
            Sprite* spr = (Sprite*)entity;
            float w = (float)spr->getWidth() * spr->getScale();
            float h = (float)spr->getHeight() * spr->getScale();
            float vx = spr->getVelocity().getX();
            float vy = spr->getVelocity().getY();
            if (spr->getX() < 0) {
                spr->setX(0);
                vx = fabs(vx);
            }
            else if (spr->getX() > SCREENW-w) {
                spr->setX(SCREENW-w);
                vx = fabs(vx) * -1;
            }
            if (spr->getY() < 0) {
                spr->setY(0);
                vy = fabs(vy);
            }
            else if (spr->getY() > SCREENH-h) {
                spr->setY(SCREENH-h);
                vy = fabs(vy) * -1;
            }

            spr->setVelocity(vx,vy);
            break;
    }
}
```

```
void game_entityCollision(Advanced2D::Entity* entity1,Advanced2D::Entity*
entity2)
{
    Sprite *box;
    Sprite *a = (Sprite*)entity1;
    Sprite *b = (Sprite*)entity2;

    if (a->getObjectType() == OBJECT_SPRITE && b->getObjectType() == OBJECT_
SPRITE)
    {
        collisions++;
        float x1 = a->getX();
        float y1 = a->getY();
        float x2 = b->getX();
        float y2 = b->getY();

        float vx1 = a->getVelocity().getX();
        float vy1 = a->getVelocity().getY();
        float vx2 = b->getVelocity().getX();
        float vy2 = b->getVelocity().getY();

        if (x1 < x2) {
            vx1 = fabs(vx1) * -1;
            vx2 = fabs(vx1);
        }
        else if (x1 > x2) {
            vx1 = fabs(vx1);
            vx2 = fabs(vx2) * -1;
        }
        if (y1 < y2) {
            vy1 = fabs(vy1) * -1;
            vy2 = fabs(vy2);
        }
        else {
            vy1 = fabs(vy1);
            vy2 = fabs(vy2) * -1;
        }
        a->setVelocity(vx1,vy1);
        b->setVelocity(vx2,vy2);
    }
}
```

```
void game_keyPress(int key) { }
void game_mouseButton(int button) { }
void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y) { }
void game_mouseWheel(int wheel) { }
void game_entityRender(Advanced2D::Entity* entity) { }
```

Threading the Game Engine

We should not add threads to the game engine unless there is a solid reason for doing so. In a simple demo program, additional threads might slow things down a bit. But in a large, complex game or a demo with many entities, the addition of a thread or two (if done carefully) should reap a significant performance boost. In other words, we want to see the core timing improve or remain steady under load by utilizing additional processor cores.

One issue that crops up when making the paradigm shift to a threaded engine is the problem of lists and iteration, especially when the engine is shutting down. There are many iterations going on in the engine core: moving, animating, drawing, collision testing. Each of these processes involves an iteration at various stages with function calls to game events. If you assume that the game is shutting down and you destroy a mutex while that mutex is in use, it will crash or hang the game. To resolve this problem, every iterative loop must have an added condition that causes it to break out when the game is shutting down. This is done by checking the global `gameover` flag in every engine function that iterates through the entity list.

Threaded Garbage Collection

Although we could move quite a bit of the engine into one or more threads, I do not want to add a level of instability to the engine just as we're nearly finished with it and ready to start building some game examples. So, while it is feasible, we will not thread the entity update or collision detection methods in the engine. Instead, I have moved the garbage collection system into a thread function. As you may recall, we added entity management to the engine back in Chapter 7, and included with that new functionality was a function called `BuryEntities`. The purpose of this function is to destroy any entities that have been disabled (by having their "alive" property set to false).

The new thread is created in `Engine::Init`:

```
mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_t thread_bury_entities;
int threadid = 1;
int id = pthread_create(&thread_bury_entities, NULL, thread_function_bury_
entities, (void*)&threadid);
```

The thread callback function is defined as a prototype in Advanced2D.h:

```
void* thread_function_bury_entities(void* data);
```

Here is the entire function as it appears in Advanced2D.cpp. Note that this function is not part of the Advanced2D class, but it is contained in the Advanced2D *namespace*.

```
void* thread_function_bury_entities(void* data)
{
    static Timer timer;
    std::list<Entity*>::iterator iter;
    while(!gameover)
    {
        if (timer.stopwatch(2000))
    {
            pthread_mutex_lock(&g_engine->mutex);

        //iterate through entity list
        iter = g_engine->getEntityList()->begin();
        while (iter != g_engine->getEntityList()->end())
        {
            if ( (*iter)->getAlive() == false )
            {
                delete (*iter);
                iter = g_engine->getEntityList()->erase( iter );
            }
            else {
                iter++;
                if (gameover) break;
            }
        }
        if (gameover) break;
        pthread_mutex_unlock(&g_engine->mutex);
        } //if
    } //gameover
    pthread_exit(NULL);
    return NULL;
}
```

Over in the `Engine::Update` method, we need to add some thread security to the many function calls in this core method using mutex locks. I will let you open the Engine project from \sources\ch11 on the CD-ROM. Here is the first section of code in the method, just to give you an example. Note how the `game_update()` call has been wrapped inside a pair of functions that lock and unlock the mutex. This is common in a threaded application, and our game engine is no exception. Without mutex protection at key areas of the game loop that are accessed in multiple places, we could end up with a very unstable and crash-prone engine.

```
void Engine::Update()
{
    //calculate core framerate
    p_frameCount_core++;
    if (p_coreTimer.stopwatch(999)) {
        p_frameRate_core = p_frameCount_core;
        p_frameCount_core = 0;
    }
    //fast update with no timing
    pthread_mutex_lock(&mutex);
    game_update();
    pthread_mutex_unlock(&mutex);
```

Testing the Newly Threaded Engine

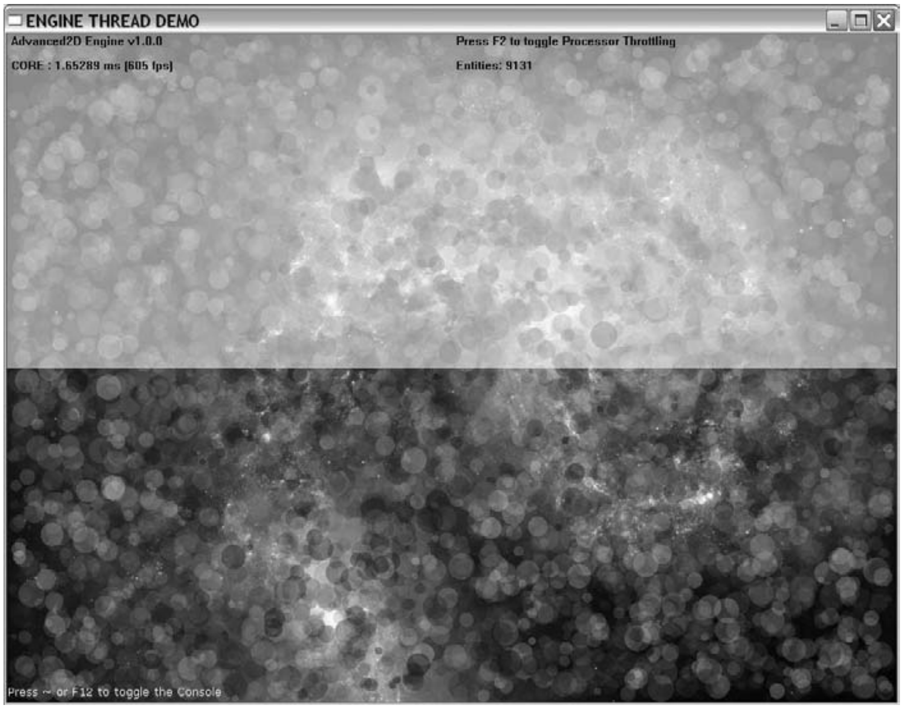
We'll now take the new engine for a spin with a new test program. The source code for this example is very similar to the ThreadDemo program earlier in the chapter. But in this example, all of the thread code has been removed (since it's now in the game engine), and there are *many* more sprites in this version! The chapter files on the CD-ROM are located under two main folders:

- Non-threaded engine
- Threaded engine

The Engine project in the non-threaded engine folder is the pre-thread version of the engine used to build the ThreadDemo program earlier in the chapter. The threaded engine folder contains another copy of the Engine project, but this one now has the thread code built into it, and this version of ThreadDemo utilizes the engine's thread support. By separating the sources into these two folders you can open the Engine project in both cases and rebuild it as needed.

The goal of the new ThreadDemo program is to give the engine's threaded garbage collection system a solid performance test with thousands of entities being added and destroyed very quickly for a long period of time. This should not only test the performance of the garbage collector, but also test the stability of the engine with its new thread support. Stability is a real challenge when you are first beginning to work with threading. You have to be careful about protecting data that could be accessed by multiple threads at the same time.

Think of your thread variables as antimatter. If your main game loop touches any variables that are being manipulated in a thread at the same time, it's like combining matter and antimatter—you'll get a huge explosion! The new ThreadDemo program is shown in Figure 11.5.



```
Advanced2D Engine v1.0.0                     Press F2 to toggle Processor Throttling
CORE : 1.65289 ms (605 fps)                  Entities: 9131
Press ~ or F12 to toggle the Console
```

Figure 11.5
The new ThreadDemo program demonstrates the engine's new garbage collection system.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;

#define SCREENW 1024
#define SCREENH 768
#define OBJECT_BACKGROUND 1
#define OBJECT_SPRITE 100
#define MAX 5000
#define SCALE 20

Texture *circle_image;
Font *font;
Console *console;
std::ostringstream ostr;

bool game_preload()
{
    g_engine->setAppTitle("ENGINE THREAD DEMO");
    g_engine->setFullscreen(false);
    g_engine->setScreenWidth(SCREENW);
    g_engine->setScreenHeight(SCREENH);
    g_engine->setColorDepth(32);
    return 1;
}

void add_sprite()
{
    Sprite *sprite = new Sprite();
    sprite->setObjectType(OBJECT_SPRITE);
    sprite->setImage(circle_image);
    D3DCOLOR color = D3DCOLOR_RGBA(0,rand()%255,rand()%255,rand()%100);
    sprite->setColor(color);
    sprite->setSize(128,128);
    sprite->setScale( (float)(rand() % SCALE + SCALE/4) / 100.0f );
    sprite->setPosition( rand() % SCREENW, rand() % SCREENH );
    sprite->setCollidable(false);
    sprite->setLifetime( rand() % 30000 );
    //set velocity
    float vx = (float)(rand()%30 - 15)/10.0f;
    float vy = (float)(rand()%30 - 15)/10.0f;
    sprite->setVelocity( vx, vy );
    //add sprite to the entity manager
    g_engine->addEntity(sprite);
}
```

```
bool game_init(HWND)
{
    int n;
    //load background image
    Sprite *background = new Sprite();
    if (!background->loadImage("galaxies.tga")) {
        g_engine->message("Error loading galaxies.tga");
        return false;
    }
    background->setObjectType(OBJECT_BACKGROUND);
    background->setCollidable(false);
    g_engine->addEntity(background);

    //create the console
    console = new Console();
    if (!console->init()) {
        g_engine->message("Error initializing console");
        return false;
    }
    console->setShowing(true);

    //load sprite image
    circle_image = new Texture();
    if (!circle_image->Load("circle.tga")) {
        g_engine->message("Error loading circle.tga");
        return false;
    }


    //create sprites
    for (n=0; n < MAX; n++)
 {
        add_sprite();
    }

    //load the Verdana10 font
    font = new Font();
    if (!font->loadImage("verdana10.tga")) {
        g_engine->message("Error loading verdana10.tga");
        return false;
    }
    if (!font->loadWidthData("verdana10.dat")) {
        g_engine->message("Error loading verdana10.dat");
        return false;
    }
    font->setColumns(16);
    font->setCharSize(20,16);

    //maximize processor
    g_engine->setMaximizeProcessor(true);

    return true;
}

void updateConsole()
{
    static Timer timer;
    if (!timer.stopwatch(50)) return;

    console->print(g_engine->getVersionText(), 0);
    ostr.str("");
    ostr << "CORE : " << (float)(1000.0f/g_engine->getFrameRate_core())
        << " ms (" << g_engine->getFrameRate_core() << " fps)";
    console->print(ostr.str(), 2);
    console->print("Press F2 to toggle Processor Throttling", 27);
    ostr.str("");
    ostr << "Entities: " << g_engine->getEntityCount();
    console->print(ostr.str(), 29);
}

void game_update()
{
    //increase size of entity list
    add_sprite();
    //update console info
    updateConsole();
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));
}

void game_render2d()
{
    font->Print(1,SCREENH-20,"Press ~ or F12 to toggle the Console");
    if (console->isShowing()) console->draw();
}
```

```
void game_keyRelease(int key)
{
    switch (key) {
        case DIK_ESCAPE:
            g_engine->Close();
            break;
        case DIK_F12:
        case DIK_GRAVE:
            console->setShowing( !console->isShowing() );
            break;
        case DIK_F2:
            g_engine->setMaximizeProcessor( !g_engine->getMaximizeProcessor() );
            break;
    }
}

void game_end()
{
    delete console;
    delete circle_image;
    delete font;
}

void game_entityUpdate(Advanced2D::Entity* entity)
{
    switch(entity->getObjectType())
    {
        case OBJECT_SPRITE:
            Sprite* spr = (Sprite*)entity;
            float w = (float)spr->getWidth() * spr->getScale();
            float h = (float)spr->getHeight() * spr->getScale();
            float vx = spr->getVelocity().getX();
            float vy = spr->getVelocity().getY();
            if (spr->getX() < 0) {
                spr->setX(0);
                vx = fabs(vx);
            }
            else if (spr->getX() > SCREENW-w) {
                spr->setX(SCREENW-w);
                vx = fabs(vx) * -1;
            }
            if (spr->getY() < 0) {
                spr->setY(0);
                vy = fabs(vy);
            }
            else if (spr->getY() > SCREENH-h) {
                spr->setY(SCREENH-h);
                vy = fabs(vy) * -1;
            }

            spr->setVelocity(vx,vy);
            break;
    }
}

void game_keyPress(int key) { }
void game_mouseButton(int button) { }
void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y) { }
void game_mouseWheel(int wheel) { }
void game_entityRender(Advanced2D::Entity* entity) { }
void game_entityCollision(Advanced2D::Entity* entity1,Advanced2D::Entity*
entity2) { }
```

Writing a multi-threaded game is now within your grasp! The result from this chapter is a slightly threaded version of the game engine and an example that uses threads in the game code instead. These two different ways of approaching threading will give you the ability to choose which way you prefer to go with threads. For the sake of simplicity, I will continue using the non-threaded engine in the chapters to come.

CHAPTER 12

Scripting

A script language for a game engine is a programming language that can be used to write script programs that do not need to be compiled. A script program can be edited after a game has been compiled, thus changing the properties and functionality of the game. High-level scripting languages allow for rapid development, content creation, and interactive events, and they drive many of today's most powerful game engines and tools. Used for both game logic and automation tools, scripting language has become a mainstay in game production.

When a game is "scripted," it means the game supports a script language. We can use a script language in many ways. The simplest use of a script is to define basic properties for a game (such as screen dimensions, full-screen mode, and color depth). We call such properties *script globals.* Script programs can also contain functions. These functions can accept parameters, perform calculations, and return results to the calling program (that is, your game). On the converse, a script program can call on C++ functions in your game. When a game has been *scripted,* by adding support for a script language and providing facilities within the game to support it, then the game takes on a whole new dimension for the game designer. No longer forced to go through programmers for gameplay changes, a game's designer can make changes to scripts to modify the game's look, feel, and other aspects.

Advice

This chapter does not teach the Lua language beyond a quick overview, as our main concern is enhancing the game engine with scripting support. The online Lua manual is an excellent resource at www.lua.org/manual/5.1/. Several books have been written about Lua, including Alex Varanese's *Game Scripting Mastery* (Course Technology PTR, 2002) and Tom Gutschmidt's *Game Programming with Python, Lua, and Ruby* (Course Technology PTR, 2003).

Introducing Lua

I have chosen to use Lua for my game projects, and therefore will share this wonderful script language with you. Lua is distributed from www.lua.org under MIT's "liberal license," which is far more open than even GPL or LGPL. You can download and freely distribute Lua, port it to new systems, and do whatever you want with it without license. Lua (pronounced LOO-ah) means "moon" in Portuguese. As such, it is neither an acronym nor an abbreviation, but a noun.

Lua is a name—the name of the Earth's moon and the name of the language. Like most names, it should be written in lowercase with an initial capital—that is, Lua. Lua is a powerful, fast, lightweight, embeddable scripting language. It combines a procedural syntax with an associative array-based system for variables and has extensible semantics. Lua is dynamically typed and runs by interpreting byte-codes for a register-based virtual machine (similar to the way in which Java programs work). Lua has automatic memory management with incremental garbage collection. It is a proven and robust language, used in industrial equipment and embedded systems, and it is the leading scripting language used in games.

Lua is fast! It is most likely the fastest language in the realm of interpreted scripting languages because it is based on the standard C library. Lua is portable, distributed in a small package that can be compiled for any platform that has an ANSI/ISO C compiler. Lua runs on all flavors of UNIX and Windows and on many mobile devices, such as BREW, Symbian, and Pocket PC. It also runs on embedded microprocessors, such as ARM. Lua is free, distributed under a liberal license, and can be used for any purpose (including commercial) at no cost. Lua was born and raised at Tecgraf, the Computer Graphics Technology Group of PUC-Rio (Pontificia Universidad Catolica de Rio de Janeiro in Brazil).

Running Lua from the Command Prompt

Most game developers test their Lua scripts from the command prompt before plugging them into a game—to verify that the script runs as expected without bugs. You can use the Lua interpreter to do that. The Lua binaries include lua5.1.exe (the interpreter) and luac5.1.exe (the script bytecode compiler). To simplify running the programs from the command line, I recommend renaming them to lua.exe and luac.exe. lua.exe (as I will refer to it from now on) is the interpreter, and you will use it to run Lua script files. luac.exe (as it will be known from now on) is the compiler, which converts a script into a bytecode file. You do not need to compile a Lua script in order to run it. You may compile it into bytecode if you don't want anyone to edit or copy your scripts when you release a program to the public.

Let's run a script from the command prompt. First, open a command prompt. In Windows, click Start, All Programs, Accessories, Command Prompt. Optionally, you may click Start, Run and type cmd into the text field (then hit Enter).

The first thing you need to do is change the current directory to where your project files are or will be stored. All of the script files demonstrated in this chapter are available on the book's CD-ROM in the .\sources\ch12\sample scripts folder.

You can use the `cd` command to change the current directory. You can type in absolute paths (such as C:\Program Files\Microsoft Visual Studio 2005) as well as relative paths (such as .\projects). The goal is to get into the directory where you have extracted the Lua interpreter (lua.exe).

Advice

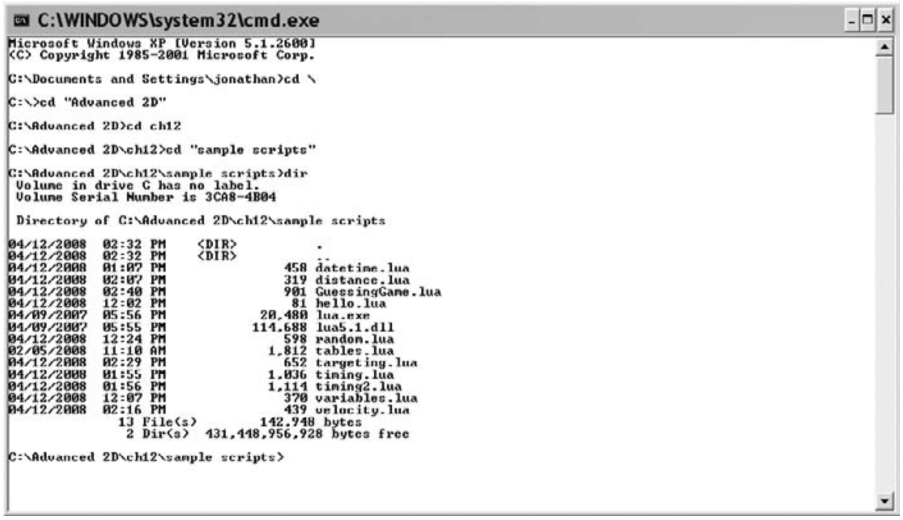
After you type in a command, such as `cd`, when you are about to type in a directory or file name, you may type in the first few letters and hit the Tab key (once or multiple times) to cycle through the files and/or folders that begin with the characters you have entered.

If you do not know where the files are located, try copying everything from \sources\ch12\sample scripts into the root folder of C:\ or in a subfolder called C:\Advanced2D (just as an example). Once you have used `cd` to reach the folder where the files are located, you can proceed.

Command-prompt programming is how users and programmers used to interact with the operating system in the old days, and it was not user friendly or easy to learn. Most Linux gurus still prefer their beloved shell rather than a fancy GUI because typing is usually much faster than clicking and moving a mouse. Here's a helpful command:

```
dir
```

This gives you a listing of the current directory. But you can also pass a folder name to `dir` to list the contents of any folder on your system. Figure 12.1 illustrates.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\jonathan>cd \

C:\>cd "Advanced 2D"

C:\Advanced 2D>cd ch12

C:\Advanced 2D\ch12>cd "sample scripts"

C:\Advanced 2D\ch12\sample scripts>dir
 Volume in drive C has no label.
 Volume Serial Number is 3CA8-4B04

 Directory of C:\Advanced 2D\ch12\sample scripts

04/12/2008  02:32 PM    <DIR>          .
04/12/2008  02:32 PM    <DIR>          ..
04/12/2008  01:07 PM               458 datetime.lua
04/12/2008  02:07 PM               319 distance.lua
04/12/2008  02:40 PM               901 GuessingGame.lua
04/12/2008  12:02 PM                81 hello.lua
04/09/2007  05:56 PM            20,480 lua.exe
04/09/2007  05:55 PM           114,688 lua5.1.dll
04/12/2008  12:24 PM               598 random.lua
02/05/2008  11:10 AM             1,812 tables.lua
04/12/2008  02:29 PM               652 targeting.lua
04/12/2008  01:55 PM             1,036 timing.lua
04/12/2008  01:56 PM             1,114 timing2.lua
04/12/2008  12:07 PM               370 variables.lua
04/12/2008  02:16 PM               439 velocity.lua
              13 File(s)        142,748 bytes
               2 Dir(s)  431,448,956,928 bytes free

C:\Advanced 2D\ch12\sample scripts>
```

Figure 12.1
Using the command prompt to navigate the file system.

Text Output

Before running the Lua interpreter, we need a script file. You can use Notepad, but while we're in the command prompt, let's do it the keyboard way. Type this:

notepad hello.lua.

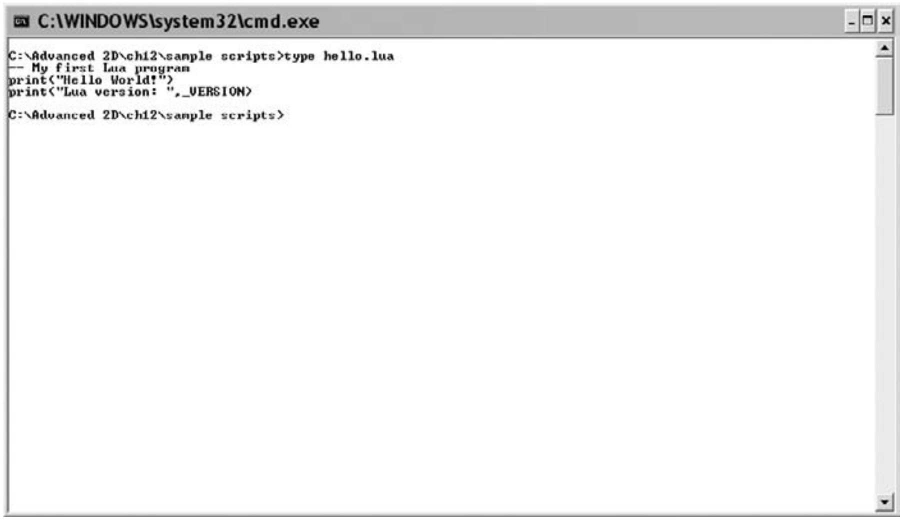
Notepad will open with a new file called hello.lua, ready for your use. If the file already exists, Notepad will open the file. Here's a sample script you can type into the hello.lua file:

```
-- My first Lua program
print("Hello World!")
print("Version: ",_VERSION)
```

Let's verify that the script file has been created by using another useful command:

```
type hello.lua
```

Figure 12.2 shows the output.



```
C:\Advanced 2D\ch12\sample scripts>type hello.lua
-- My first Lua program
print("Hello World!")
print("Lua version: ",_VERSION)

C:\Advanced 2D\ch12\sample scripts>
```

Figure 12.2
Viewing the contents of a file at the command prompt.

Advice

If you are serious about Lua script programming, you will need a better editor. I recommend Notepad++, a free editor that provides colored syntax highlighting and a tabbed interface. Notepad++ is provided on the CD-ROM and is also available for download at http://notepad-plus.sourceforge.net.

The Lua interpreter is so small that you can just copy it into your current project folder so that you can run it from the command line to test your script files. (Just be sure to copy the lua5.1.dll library file with Lua.exe since it's a dependency.)

Advice

If you run Lua.exe without a script file, it will go into command mode. Just press Ctrl+C to exit.

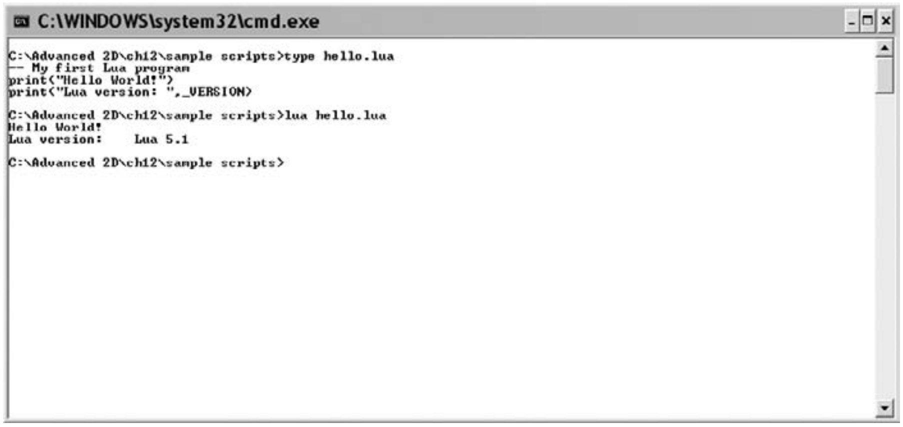
Assuming you're in the right folder, you have copied lua.exe and lua5.1.dll into that folder, and you have created your first script file (whew!), let's give it a test run:

```
lua hello.lua
```

Figure 12.3 shows the output from the hello.lua program.

Variables

Let's talk about how Lua handles variables. In Lua, *everything* is an object. But since "object" is such a generic term these days, to be more specific, Lua uses a



```
C:\Advanced 2D\ch12\sample scripts>type hello.lua
-- My first Lua program
print("Hello World!")
print("Lua version: ",_VERSION)

C:\Advanced 2D\ch12\sample scripts>lua hello.lua
Hello World!
Lua version:    Lua 5.1

C:\Advanced 2D\ch12\sample scripts>
```

Figure 12.3
Running the hello.lua script program.

dictionary-style container for all variables (where a dictionary stores each data item with a lookup value). Here is an example script showing how variables are declared and used.

```
-- Doing variables in Lua
--simple variables
myinteger = 100
mydouble = 3.1415926535
mystring = "some string"
print("myinteger = ", myinteger)
print("mydouble = ", mydouble)
print("mystring = ", mystring)

--complex variable (table)
Person = {
  age = 30,
  name = "John"
}
print("Person's name: ", Person.name)
print("Person's age: ", Person.age)
```

This program produces the following output:

```
myinteger =     100
mydouble =     3.1415926535
mystring =     some string
Person's name:     John
Person's age:     30
```

As you can see from the output, when using the `print()` function, multiple parameters can be separated by a comma, which inserts a tab character into the output stream. If you want to just append text, you must use Lua's unusual text concatenation operator, `..` (double dot), like so:

```
print("mystring = " .. mystring)
```

Random Numbers

Lua has a weak random-number generator because it cannot be easily initialized with a random seed. Although Lua provides a function called `math.randomseed()`, it does not work properly because Lua does not provide an *epoch*-based millisecond timer. Lua initializes its timer when the script begins to run, which means the best we can do is send 0 to `math.randomseed()`. We need to mix up the random-number generator (`math.random()`) with some large numbers and the use of modulus (`math.mod()`) to produce a pseudo-random result. It's messy. But it can be put into a reusable function.

```
function gen_random(max)
    local temp = math.mod((42 * math.random() + 29573), 139968)
    local ret = math.mod( ((100 * temp)/139968) * 1000000, max)
    return round(ret + 0.5)
end
```

This function has a dependency—a function called `round`. Here is the `round` function (which supports rounding to any number of decimal places):

```
function round(num, places)
  local mult = 10^(places or 0)
  return math.floor(num * mult + 0.5) / mult
end
```

Those two functions work pretty well to produce a random number up to the passed maximum value parameter. But more often than not, we need to generate a random number within a fixed range (such as 100 to 150). That's easy enough:

```
function random_range(min,max)
  return gen_random(max-min) + min
end
```

We can now create random numbers with pretty good consistency. Here is an example:

```
math.randomseed( os.time() )
output = ""
```

```
for n = 1,10 do
  a = random_range(1,1000)
  output = output .. tostring(a) .. ","
end
print(output)
```

The output from this program is a list of 10 random numbers in the range of 1 to 1,000:

```
63,52,806,319,700,189,617,981,852,15,
```

Arrays and Tables

Lua variables can contain complex data types, similar to a C++ struct (and even a class through some convoluted code). To define a table, set a variable name equal to an empty set of brackets:

```
grades = {}
```

After defining the table, you can fill it in with data like so:

```
grades[1] = 90.5
grades[2] = 78.3
grades[3] = 85.8
grades[4] = 76.2
grades[5] = 68.1
grades[100] = 50.3
grades[200] = 100.0
```

Did you notice that the last two elements in the grades table are out of order (100 and 200)? Lua allows you to do that, but it will not fill in the missing elements leading up to those numbers—they will all be `nil` unless they are defined (which is true of any variable in Lua). Just remember that every variable is an entry in Lua's dictionary container. In that case, `grades[100]` is more of a name than an array element (though that's a simplification). Let's examine a more complex type of Lua table—one containing multiple named items:

```
persons = {}
persons[1] = { name = "John", age = 30, weight = 180, IQ = 120 }
persons[2] = { name = "John", age = 18, weight = 150, IQ = 113 }
persons[3] = { name = "Sue", age = 19, weight = 110, IQ = 125 }
persons[4] = { name = "Dave", age = 20, weight = 160, IQ = 110 }
persons[5] = { name = "Laura", age = 24, weight = 100, IQ = 118 }
```

```
persons[6] = { name = "Don", age = 18, weight = 130, IQ = 122 }
persons[7] = { name = "Julie", age = 22, weight = 120, IQ = 105 }
persons[8] = { name = "Craig", age = 21, weight = 180, IQ = 112 }
persons[9] = { name = "Sarah", age = 20, weight = 115, IQ = 130 }
```

The `persons` table contains four properties: name, age, weight, and IQ. You can legally violate the syntax of any one element in the table, and Lua will not complain—although you could wind up with a program crash. For instance, I could redefine `persons[8]` like so:

```
persons[8] = { something = 1, something_else = "blah" }
```

and Lua will accept it. But, although you *can* do this, it makes no sense to do so because the most common use for a table is for iteration.

Lua provides a library called `table` that you can use to get information about one of your tables. The function `table.getn()` will tell you how many items are contained in your table. For instance:

```
print( table.getn( persons ) )
```

will display the number of elements in the `persons` table (which is 9). Odd as it may seem to your C++ training, `table` is a Lua library, and your own custom-defined tables are *not* objects with methods such as size or length. You must pass the name of your table to `table.getn()` instead. With this new information, we can print out the data in the `persons` table. Note in this example that you can access properties in a table using two different formats (interchangeably)—either the property name as an index or the property name with the dot operator.

```
print("PERSONS")
size = table.getn(persons)
for n = 1, size do
  print( "Person #" .. n )
  print( "    Name   = " .. persons[n]["name"] )
  print( "    Age    = " .. persons[n].age )
  print( "    Weight = " .. persons[n]["weight"] )
  print( "    IQ     = " .. persons[n].IQ )
end
```

This produces the output:

```
PERSONS
Person #1
```

```
    Name   = John
    Age    = 30
    Weight = 180
    IQ     = 120
Person #2
    Name   = John
    Age    = 18
    Weight = 150
    IQ     = 113
Person #3
    Name   = Sue
    Age    = 19
    Weight = 110
    IQ     = 125
Person ...
```

Timing

Lua provides functions to retrieve the current date and time. The `os.date()` function returns the current date with a default format that includes the time (and you may modify the format if you wish):

```
Date: 04/12/08 13:04:24
```

The `os.time()` function returns the number of seconds that have passed since January 1, 1970 (on most systems—though this start date may vary from system to system):

```
Time: 1208030664
```

Dividing this number by 60 produces the minutes since the beginning of the epoch. Dividing by another 60 results in hours. Then, dividing by 24 hours and again by 360 days, you will calculate the number of years. Now let's look into more specific timing features of the Lua language.

There are times when you may wish to profile or benchmark your script code to see whether it's slowing down the game much (if at all). We can use Lua's `os` library to get the current time in a number of ways. I've mentioned already that Lua does a poor job of seeding the random number generator due to its lack of an *epoch*-based millisecond timer. But once the program starts up, you *can* retrieve milliseconds in order to profile your script code. Here is a function called

`Stopwatch()` that I find useful for slowing down the output of profiling code (for instance, limiting the print calls to once per second):

```
start_time = 0
function Stopwatch(ms)
  if Timer() > start_time + ms then
    start_time = Timer()
    return true
  else
    return false
  end
end
```

`Stopwatch()` just returns true or false depending on whether the desired number of milliseconds has passed. To profile a function in Lua, as in C++, you must call it *many* times, getting a baseline time value, and then divide that time by the number of iterations. This is the only way to get the actual time taken to call a function since the processor can perform a function call in a matter of microseconds (millionths of a second) or nanoseconds (one billionth of a second), while a millisecond is only one thousandth of a second.

The `Stopwatch()` function has a dependency on another function we have not yet seen. The `Timer()` function returns the number of milliseconds since the program started with the help of `os.clock()`. This function normally returns just *seconds,* but it also provides a decimal value containing the milliseconds as well. By multiplying `os.clock()` by 1,000, we can convert floating-point seconds into integer milliseconds.

```
function Timer()
  return os.clock() * 1000
end
```

Here is an example that prints out the raw clock value and the converted millisecond value once every second:

```
repeat
  if Stopwatch(1000) then
    print("os.clock(): " .. os.clock())
    print("Timer(): " .. Timer())
  end
until false
```

That script program produces the following output:

```
os.clock(): 1.015
Timer(): 1015
os.clock(): 2.015
Timer(): 2015
os.clock(): 3.031
Timer(): 3031
os.clock(): 4.046
Timer(): 4046
```

The millisecond timer is indeed returning milliseconds, as you can see in this output, because the data is being printed out once per second.

The last thing I want to go over with you regarding timing is a function profiling program. The purpose of this program is to demonstrate how to test the runtime of a single function. To slow down the function call, we will calculate square root—which is notoriously difficult for most processors to calculate. First, we'll get a random number, then get the square root of the number using `math.sqrt()`, and return the value for good measure.

```
function SlowMathFunction()
  r = random_range(1,999999)
  num = math.sqrt(r)
  return num
end
```

Why do we set the square root value equal to a variable first, before returning it? The Lua interpreter might be smart enough to optimize code like this, so we need to physically store the result of the calculation in a variable to actually use a processor cycle.

This program uses a `for` loop that runs the function a couple million times in order to calculate the time taken to run the function *once,* and the results are printed out.

```
print("Profiling function...")
TOTAL = 2000000

start = Timer()
for n = 1,TOTAL,1 do
  var = SlowMathFunction()
end
finish = Timer()
```

```
delta = finish-start
print("Total run time: " .. delta .. " ms" )
print("Function run time: ")
milli = round(delta / TOTAL, 8)
micro = round(milli * 1000, 8)
nano = round(micro * 1000, 8)
print(" milliseconds: " .. milli )
print(" microseconds: " .. micro )
print(" nanoseconds : " .. nano )
```

Here is some sample output. As you can see, it takes about one and a half microseconds to run the `SlowMathFunction()` just once.

```
Profiling function...
Total run time: 3218 ms
Function run time:
  milliseconds: 0.001609
  microseconds: 1.609
  nanoseconds : 1609
```

Just out of curiosity, let's see how much of that 1.6 microsecond figure is taken up in the function calls (including the `random` function). Here's a revised version of the program that has the math calculation embedded in the `for` loop:

```
Total run time: 2796 ms
Function run time:
  milliseconds: 0.001398
  microseconds: 1.398
  nanoseconds : 1398
```

Look at that! There's a difference of 211 nanoseconds. Interestingly, this program calls the `SlowMathFunction()` two million times, so there seems to be a noticeable but infinitesimally small amount of overhead in each function call. The value will differ from one system to the next and will depend on processor speed, but I calculated one-tenth of a picosecond (which is, for all practical purposes, too small to be relevant).

Distance

In general, you will want to code your math functions in C++, rather than in Lua, because despite Lua's solid performance, it is an *interpreted* language and it cannot compete with a compiled binary for performance. But there are times when a designer may wish to just perform some range tests or gameplay tests to see whether a game is doing what it's supposed to do in unusual situations. One common calculation is to derive the distance between two points. A point can represent the position of any game entity. The following program prints out the distance between two points:

```
function distance(x1,y1,x2,y2)
  return math.sqrt( (x2-x1)^2 + (y2-y1)^2 )
end
x1 = 100
y1 = 150
x2 = 780
y2 = 620
print("Point 1: " .. x1 .. "," .. y1)
print("Point 2: " .. x2 .. "," .. y2)
print("Distance = " .. distance(x1,y1,x2,y2) )
```

Feel free to change the point locations. The output using these points is:

```
Point 1: 100,150
Point 2: 780,620
Distance = 826.6196223173
```

Velocity

We explored linear velocity and incorporated velocity calculations in the `Math` class back in Chapter 10. We can code these functions in Lua as well. Here are the Lua versions of `VelocityX()` and `VelocityY()` with some test code:

```
function VelocityX(angle)
  return math.cos( (angle-90) * math.pi/180)
end
function VelocityY(angle)
  return math.sin( (angle-90) * math.pi/180)
end
ang = 120
vx = round(VelocityX(ang),2)
vy = round(VelocityY(ang),2)
print("Velocity(" .. ang .. ") = " .. vx .. "," .. vy)
```

This example script produces the following output:

```
Velocity(120) = 0.87,0.5
```

Targeting

For good measure, I'll throw in one last math function that has been converted from its C++ equivalent in the `Math` class: calculating the angle between two points. Like the C++ `atan2()` function, Lua's `math.atan2()` calculates an angle based on delta Y and delta X for two points.

```
function target_angle(x1,y1,x2,y2)
  deltaX = x2-x1
  deltaY = y2-y1
  return math.atan2( deltaY, deltaX )
end
x1 = 100
y1 = 100
x2 = 900
y2 = 600
print("Point 1 = " .. x1 .. "," .. y1 )
print("Point 2= " .. x2 .. "," .. y2 )
rangle = target_angle(x1,y1,x2,y2)
rangle = round(rangle,4)
dangle = math.deg( target_angle(x1,y1,x2,y2) )
dangle = round(dangle,4)
print("Target Angle in radians = " .. rangle )
print("Target Angle in degrees = " .. dangle )
```

(Just grab a copy of the `round()` function from one of the other script listings for use in this program—it was covered earlier in the chapter.) Here's the output:

```
Point 1 = 100,100
Point 2= 900,600
Target Angle in radians = 0.5586
Target Angle in degrees = 32.0054
```

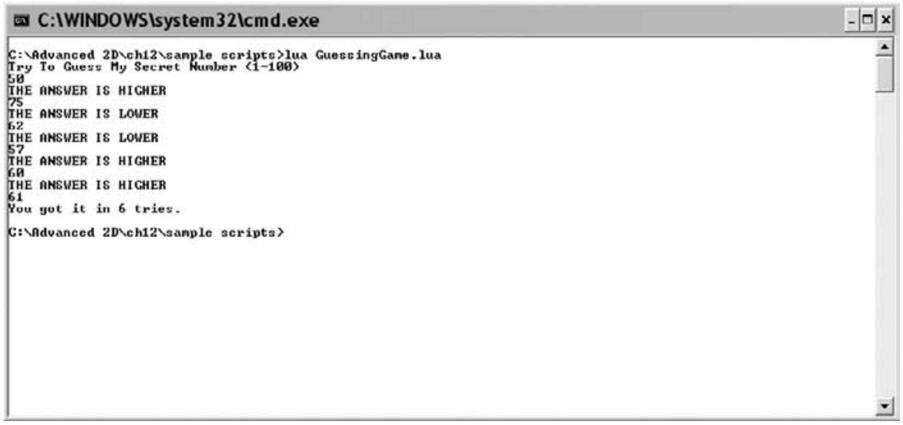
Guessing Game

Now let's make a simple game entirely in Lua script! You can run this game from a command prompt using the Lua interpreter. The guessing game script first generates a random number from 1 to 100, and then asks the user to continue guessing until he or she gets the answer right. Each time a number is entered, the game tells the player whether the answer is higher or lower.

Here is the source code for the game. Not shown are the common functions we've gone over previously: `round()`, `gen_random()`, and `random_range()`.

```
function GetInput()
  return io.stdin:read("*l")
end
print "Try To Guess My Secret Number (1-100)"
math.randomseed( os.time() )
answer = random_range(1,100)
guess = 0
total = 0
repeat
  input = GetInput()
  guess = tonumber(input)
  if guess > answer then
    print("THE ANSWER IS LOWER")
  elseif guess < answer then
    print("THE ANSWER IS HIGHER")
  end
  total = total + 1
until guess == answer
print("You got it in " .. total .. " tries.")
```

Figure 12.4 shows the output of the guessing game script in a command prompt.



```
C:\Advanced 2D\ch12\sample scripts>lua GuessingGame.lua
Try To Guess My Secret Number (1-100)
50
THE ANSWER IS HIGHER
75
THE ANSWER IS LOWER
62
THE ANSWER IS LOWER
57
THE ANSWER IS HIGHER
60
THE ANSWER IS HIGHER
61
You got it in 6 tries.

C:\Advanced 2D\ch12\sample scripts>
```

Figure 12.4
The Guessing Game script program.

Lua and C++

Now we will plug Lua into the game engine to provide scripting support for our future game projects. Distributed with the Lua library are the header files and library file that must be added to a game project. In our case, we'll be adding these files to the Advanced2D engine project. Here are the headers:

- lua.hpp
- lua.h
- luaxlib.h
- luaconf.h
- lualib.h

You may copy these files into your compiler's .\include folder so that they will always be available for future projects, or you may copy the headers into your project's source code folder. Either way, you will need to include the lua.hpp file. This extension (.hpp) is technically the proper file extension for a C++ header file, but .h is so common and familiar that most programmers still use it. Although you need all of the Lua headers, you need only include lua.hpp as:

```
#include "lua.hpp"
```

Or, if referring to the file in your compiler's .\include folder, like so:

```
#include <lua.hpp>
```

This single header includes the others so you need only include this one file.

Lua can be compiled into a binary executable directly, but the more common way of including Lua support in a C++ program is with a library file. Included in the Lua distribution are the full sources with makefiles for various compilers. I have provided the precompiled library files for Dev-C++ and Visual C++ on the CD-ROM under the folder for this chapter.

The Dev-C++ library file is called liblua.a and is added to the linker options with -llua. The Visual C++ library file is called lua5.1.lib and is added to the linker options with the whole filename.

Lua Script Class

I have written a simple `Script` class that encapsulates basic Lua scripting support, which will keep the game code free of the somewhat messy Lua function calls. The `Script` class does not support function parameters or return values. To keep the class as simple and easy to use as possible, it only works with globals. If you want to pass a parameter to a function, you can just set a global, call the function, then retrieve the result (also a global). This simplicity may have limits as you begin to gain experience with script programming—in which case you will be able to add new functionality to the class.

Setting and retrieving globals is accomplished with functions. Here are the string functions:

```
std::string getGlobalString(std::string name);
void setGlobalString(std::string name, std::string value);
```

To set a global string to be used by the Lua script, call `setGlobalString`. Likewise, to retrieve a global string, use `getGlobalString`. Here is an example:

```
std::string Name = script.getGlobalString("NAME");
Name = "New Name String";
script.setGlobalString("NAME", Name);
```

There are also functions for working with global numbers and Booleans that work in a similar fashion.

```
double getGlobalNumber(std::string name);
void setGlobalNumber(std::string name, double value);
bool getGlobalBoolean(std::string name);
void setGlobalBoolean(std::string name, bool value);
```

The `Script` class can run functions defined in your Lua script using the `runFunction` method, which accepts as its single parameter the name of the function. No parameters or return values are supported, although you can accomplish the same thing using global variables.

```
void runFunction(std::string name);
```

Now let's see the source code for the `Script` class, and afterward we'll plug it into the game engine. Here is the header file for the `Script` class:

```
#include "Advanced2D.h"
#pragma once
namespace Advanced2D {
    class Script
    {
    private:
        lua_State *luaState;
    public:
        Script();
        Script(std::string scriptfile);
        virtual ~Script();

        bool loadScript(std::string scriptfile);

        std::string getGlobalString(std::string name);
        void setGlobalString(std::string name, std::string value);

        double getGlobalNumber(std::string name);
        void setGlobalNumber(std::string name, double value);

        bool getGlobalBoolean(std::string name);
        void setGlobalBoolean(std::string name, bool value);

        void runFunction(std::string name);
    };
};
```

Next up is the `Script` class implementation.

```
#include "Script.h"
namespace Advanced2D {
    Script::~Script()
    {
        lua_close(luaState);
    }

    Script::Script()
    {
        luaState = lua_open();
        luaL_openlibs(luaState);
    }
    Script::Script(std::string scriptfile) : Script()
    {
        loadScript(scriptfile);
    }

    bool Script::loadScript(std::string scriptfile)
    {
        bool value = true;
        try {
```

```
        luaL_dofile(luaState, scriptfile.c_str());
    }
    catch(...) {
        value = false;
    }
    return value;
}

std::string Script::getGlobalString(std::string name)
{
    std::string value = "";
    try {
        lua_getglobal(luaState, name.c_str());
        value = lua_tostring(luaState, -1);
        lua_pop(luaState, 1);
    }
    catch(...) {
    }
    return value;
}

void Script::setGlobalString(std::string name, std::string value)
{
    lua_pushstring(luaState, value.c_str());
    lua_setglobal(luaState, name.c_str());
}

double Script::getGlobalNumber(std::string name)
{
    double value = 0.0;
    try {
        lua_getglobal(luaState, name.c_str());
        value = lua_tonumber(luaState, -1);
        lua_pop(luaState, 1);
    }
    catch(...) {
    }
    return value;
}

void Script::setGlobalNumber(std::string name, double value)
{
        lua_pushnumber(luaState, (int)value);
        lua_setglobal(luaState, name.c_str());
    }

    bool Script::getGlobalBoolean(std::string name)
    {
        bool value = 0;
        try {
            lua_getglobal(luaState, name.c_str());
            value = (bool)(int) lua_toboolean(luaState, -1);
            lua_pop(luaState, 1);
        }
        catch(...) {
        }
        return value;
    }

    void Script::setGlobalBoolean(std::string name, bool value)
    {
        lua_pushboolean(luaState, (int)value);
        lua_setglobal(luaState, name.c_str());
    }

    void Script::runFunction(std::string name)
    {
        //call script function, 0 args, 0 retvals
        lua_getglobal(luaState, name.c_str());
        lua_call(luaState, 0, 0);
    }
};
```

Advice

This `Script` class began life in a real game project. *Starflight: The Lost Colony* (www.starflightgame.com) uses Lua scripts extensively, for everything from the Starport, to the player's ship physics, to alien encounters. In particular, the encounter system features a dozen or so script files with about 10,000 lines of script code!

Linking with the Lua Library

While the Advanced2D engine has no linked files (because it is a library itself, not an object file), we do need to add the required linked files to our game projects.

At this stage in the engine's development, our example programs and games must include the following linker options for Dev-C++:

- -lAdvanced2D
- -ld3d9
- -ld3dx9
- -ldinput8
- -ldxguid
- -lwinmm
- -lfmodex
- -llua

If you're using Visual C++, the linker options will need to include these files:

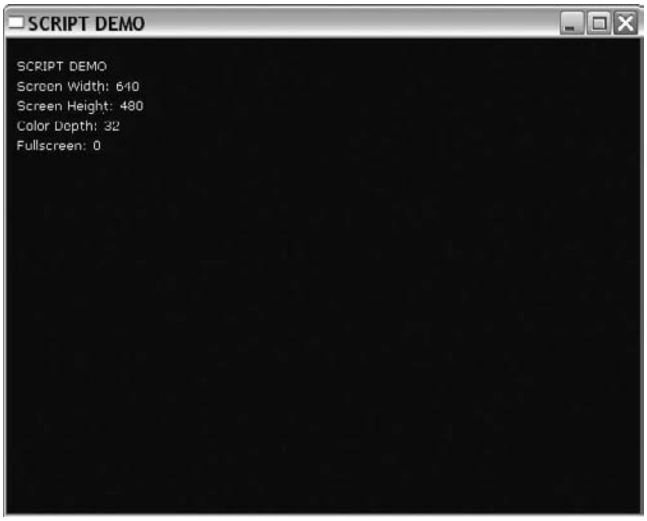
- ..\..\Engine\lib\Advanced2D.lib
- d3d9.lib
- d3dx9.lib
- dinput8.lib
- dxguid.lib
- winmm.lib
- fmodex_vc.lib

Script Test

The ScriptDemo program is a very simple program that demonstrates how to use a Lua script to configure the screen and program title. The following properties are stored in the script file:

```
PROGRAMTITLE = "SCRIPT DEMO"
FULLSCREEN = false
SCREENWIDTH = 640
SCREENHEIGHT = 480
COLORDEPTH = 32
```

Figure 12.5 shows the ScriptDemo program window. You may experiment with the properties in the script.lua file used by this program without needing to recompile the program! Change any property you want to see how the change affects the program when it loads up. This is the power of scripting—being able to make dramatic changes to a program without recompiling its code.



```
SCRIPT DEMO
Screen Width: 640
Screen Height: 480
Color Depth: 32
Fullscreen: 0
```

Figure 12.5
The ScriptDemo program uses a configuration script.

```
#include "..\Engine\Advanced2D.h"
using namespace Advanced2D;
Font *font;
Script script;
std::string title;
int width;
int height;
int depth;
bool fullscreen;

bool game_preload()
{
    script.loadScript("script.lua");
    title = script.getGlobalString("PROGRAMTITLE");
    width = (int)script.getGlobalNumber("SCREENWIDTH");
    height = (int)script.getGlobalNumber("SCREENHEIGHT");
    depth = (int)script.getGlobalNumber("COLORDEPTH");
    fullscreen = script.getGlobalBoolean("FULLSCREEN");
    g_engine->setAppTitle(title);
    g_engine->setScreenWidth(width);
    g_engine->setScreenHeight(height);
    g_engine->setColorDepth(depth);
    g_engine->setFullscreen(fullscreen);
    return true;
}

bool game_init(HWND)
{
    //load the Verdana10 font
    font = new Font();
    if (!font->loadImage("verdana10.tga")) {
        g_engine->message("Error loading verdana10.tga");
        return false;
    }
    if (!font->loadWidthData("verdana10.dat")) {
        g_engine->message("Error loading verdana10.dat");
        return false;
    }
    font->setColumns(16);
    font->setCharSize(20,16);
    return true;
}

void game_render2d()
{
    std::ostringstream ostr;
    font->Print(10,20,title);
    ostr << "Screen Width: " << width;
    font->Print(10,40,ostr.str());
    ostr.str("");
    ostr << "Screen Height: " << height;
    font->Print(10,60,ostr.str());
    ostr.str("");
    ostr << "Color Depth: " << depth;
    font->Print(10,80,ostr.str());
    ostr.str("");
    ostr << "Fullscreen: " << fullscreen;
    font->Print(10,100,ostr.str());
}
```

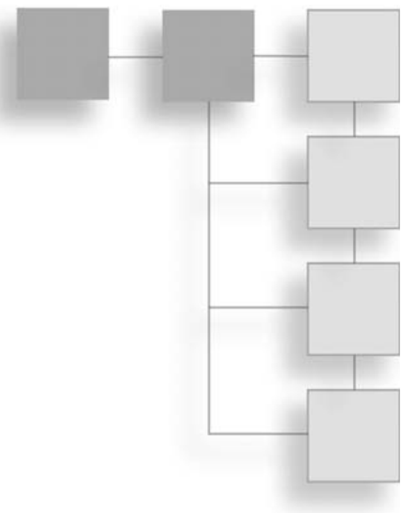
```
void game_end()
{
    delete font;
}

void game_keyRelease(int key)
{
    switch (key) {
        case DIK_ESCAPE:
            g_engine->Close();
            break;
    }
}

void game_render3d()
{
    g_engine->ClearScene(D3DCOLOR_XRGB(0,0,80));
}

void game_update() { }
void game_keyPress(int key) { }
void game_mouseButton(int button) { }
void game_mouseMotion(int x,int y) { }
void game_mouseMove(int x,int y) { }
void game_mouseWheel(int wheel) { }
void game_entityRender(Advanced2D::Entity* entity) { }
void game_entityUpdate(Advanced2D::Entity* entity) { }
void game_entityCollision(Advanced2D::Entity* entity1,
    Advanced2D::Entity* entity2) { }
```

That wraps up scripting support. The game engine has now been dramatically improved as a result of this powerful new feature, and we will continue to utilize it in the chapters to come.

This page intentionally left blank

CHAPTER 13

Games



This final chapter of the book demonstrates the example game projects provided on the CD-ROM that make use of the Advanced2D engine. It's been a long haul since the first chapter, and we've created a lot of fascinating code along the way. Now we need to put it to good use. There are two important points that I want to communicate to you before we take a look at the examples.

The first point that I want to drive home is to help you to understand the nature of the Advanced2D engine: It is an ever-evolving engine that continues to see improvement day by day. The example games in this chapter were developed as prototypes *before* the Advanced2D engine was developed and indeed were used as a trial run for the earliest entity management and collision detection tests. Now that the engine is fully developed, these two games were upgraded to take advantage of new features that I did not imagine originally. The result was a feedback loop that manifested while upgrading to the latest engine specs. While doing so, I found a need for new features (class properties and methods) in the engine, either to simplify the front-end code or to improve performance.

The second point follows up on the first: A game engine is never *finished;* there are only levels of functionality. Early in the project—back around Chapter 3—it was possible to create a game with animation and crude keyboard input. It *did* function, and the `Sprite` class worked. However, over the next 10 chapters, that class (as well as others) saw continual improvements, bug fixes, and optimizations. Like a novelist or a film director, a game developer may feel that his or her work is never truly finished, but due to deadlines and the fact that life must go on, one must find a good stopping point. There, at that snapshot in time, you deliver what outsiders might consider a finished product—but you feel that it is never truly finished. I feel that I am nowhere *near* finished with the Advanced2D engine! But what is in print need not stymie the game's further development online.

As you may learn from exploring the book's CD-ROM, I have provided examples of research that I was conducting just prior to the book going to print—code that might have become another engine class or example game. But the important thing is that the engine works and is rock solid. Which brings me back to a topic of iterative programming—a technique that resulted in a good measure of stability in this engine. If at any time you introduce a feature to a game that is *really cool* but ultimately makes the program unstable, you *must* find a way to make it stable or you must remove that feature. Such was the case with the threaded garbage collector introduced into the engine back in Chapter 11.

In theory, threading provides enormous performance gains for a game engine (just as more pistons in an automobile engine result in more power). However, there are alternatives to the brute-force approach to performance. In the automobile-racing realm, a turbocharger provides more power than additional pistons, but it's a complex technology that requires maintenance. For a one-shot drag-racing engine, that's a great solution because the engine will be rebuilt after every run down the quarter mile. But for a NASCAR race, teams need moderate power with *great* reliability, so they build race engines with hardened internals but no power adders.

The thread-based code works great on its own, but more research will need to be done to determine how to best support multiple-core processors while maintaining stability in the engine.

What approach do you need to take for your engine? That will depend on your design goals. I prefer strong stability over raw performance. There are many game studios producing stable, high-quality games, but the one studio that stands out every time is Blizzard Entertainment (developers of the *Warcraft* and *Starcraft* series). Dating back to the early 1990s, Blizzard's games have always maintained an exceptionally high standard of quality, both in their code and artwork, and the success of the company's products is an obvious result. By focusing on quality and stability first and basing your games on that foundation, you can introduce new performance and visual upgrades later. I won't single out any case examples, but there are many games released today for both PC and console that look terrific, with all the latest fancy buzzwords built in (which marketing people *love!*), and yet some games are riddled with bugs. Consider these issues while developing your own games. What would improve the gameplay more—a normal mapping shader or a more fluid animation system?

Scrolling Example

One of the additional examples provided on the CD-ROM with this chapter is a research project in bitmap-based side scrolling. This technique is not new, of course—side-scrolling games have been around for decades. But it will be a new feature in the engine in due time. Bitmap scrolling layers along with tiled scrolling layers will make possible some interesting new games, such as classic *Mario*-style side scrollers with both bitmap and tiled layers rendered with parallax perspective and alpha blending. Now I know what a film director goes through during the final editing process. I would love to put the book on hold another month to fully develop this subject! Alas, it will have to be continued online.

Figure 13.1 shows a prototype bitmap scroller class that provides the ability to add multiple independently scrolling layers with alpha. In this example, two layers are being drawn transparently over a starry background. Work will continue on this example.

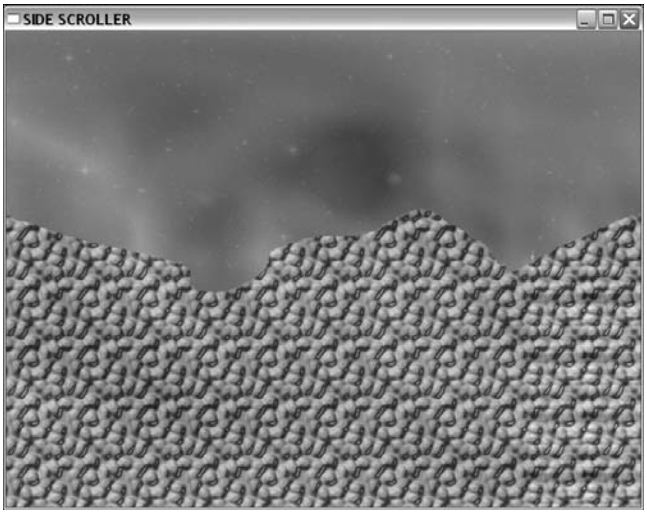
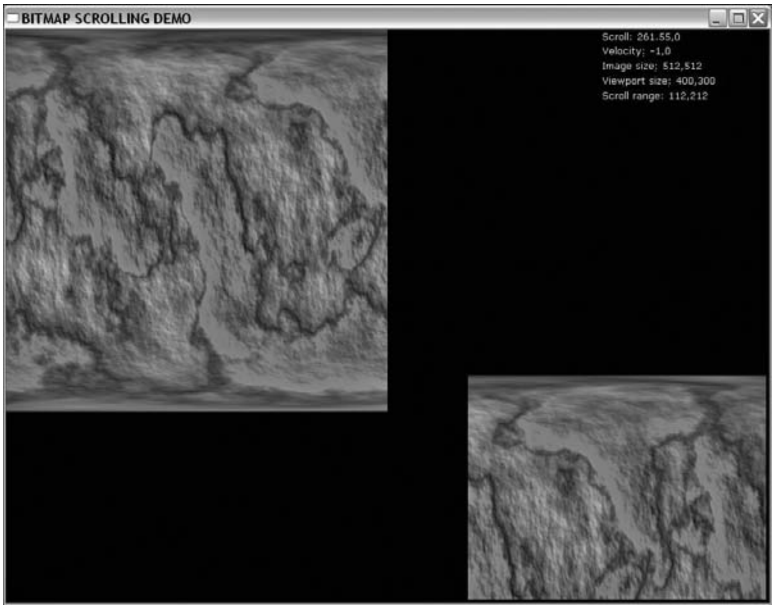


Figure 13.1
This example demonstrates layered side scrolling with transparency.



```
Scroll: 261.55,0
Velocity: -1,0
Image size: 512,512
Viewport size: 400,300
Scroll range: 112,212
```

Figure 13.2
This example shows the algorithm that makes side scrolling work.

The second screenshot, Figure 13.2, shows a working example of bitmap scrolling in a small viewport on the screen. The texture was generated by a program called TextureGenerator, which is based on the Perlin random noise library and is provided in the \sources\bonus folder of the CD. We used this code to generate real-time random planet textures in *Starflight*. The result: Every planet has a unique textured surface that need not be stored in a bitmap file, and the planet surface tiled scroller engine uses the texture for the tilemap! I've used the same texture generator to produce a molten planet texture for use as an example layer in this program.

Bitmap-based side scrolling is just the beginning of this future engine upgrade, which will include scrolling in any direction and support for tiled layers. I have covered these subjects in other books such as *Beginning Game Programming, 2nd Edition* (the prequel to this book in many ways) and *Game Programming All In One, 3rd Edition,* so the existing theory and example code will make it very easy to add this old but fun technique to the Advanced2D engine. Here's an intriguing challenge: How would you integrate a scrolling layer system into the core engine so that layers are automatically updated and rendered, while maintaining mesh and sprite rendering at the same time?

Blocks Game

Popularized by classic games such as *Breakout* and *Arkanoid,* the ball-and-paddle block-bashing game genre is a fun exercise in game programming because it requires fast-paced input, sprite control, animation, and collision detection—all the nutrients you need for a healthy game in a little package. This is what I consider the marquee demo game for the Advanced2D engine because it provides a good overview of the engine's core features.

The source code is about as tight as I could make it without resorting to complex algorithms, and yet it is still 870 lines—or 30 pages of laid-out text (which is why I'm not listing the code here). And yet, this is a *tiny* game by most standards. (In contrast, most commercial games have more than 800 lines of code just for the title screen.) Figure 13.3 shows a screenshot of the Blocks game.

From this figure, you can see that this is no ordinary *Breakout*-style game. First of all, the ball is a light source that casts a shadow when it nears the paddle. Or is this just a rendering trick? You can explore the source code to find out! Second, the ball leaves a trail behind it as it moves. In addition, the blocks themselves are rendered in varying levels of alpha, and you must hit them repeatedly to eliminate them. The ball's power level increases as long as you don't let it hit the floor.

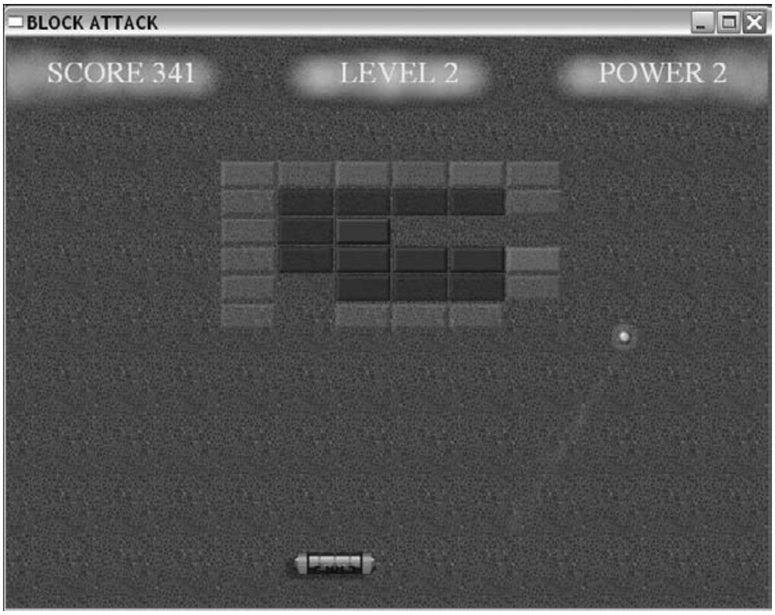


Figure 13.3
The Blocks game.

Unlike most games of this genre, this game doesn't have "lives" in the normal sense; instead, if you let the ball hit the floor, the power level drops back down to one, and you have to build it back up again! The status information on top is highlighted with smoky particles!

You may add new levels to the game by modifying the levels.h file. Here's a challenge: Modify the game so it uses Lua script to define the levels. By doing this, anyone will be able to custom-design new block layouts and try them out without recompiling the game.

Alien Invaders

The Alien Invaders game is a very old concept but has a lot of validity for a game programming exercise because it demonstrates even more of the complexity found in most games, such as advanced collision response. This game demonstrates the flexibility of the engine with a trivial example that is slightly smaller than Blocks, at just about 700 lines of code. It's shorter because it doesn't use as many special effects, such as particles (although it's ripe for such enhancement!). However, unlike Blocks, this game *does* feature a 3D background scene (of a rotating Earth) instead of a fixed background image. See Figure 13.4.

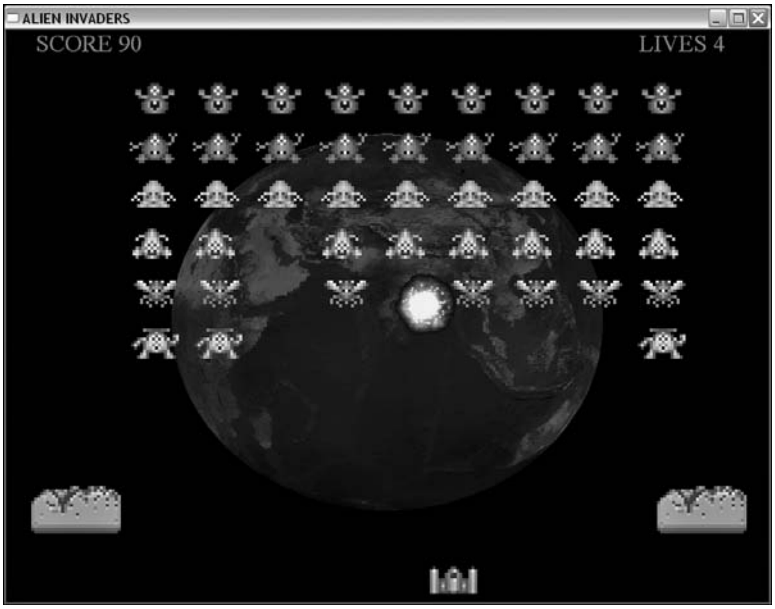


Figure 13.4
The Alien Invaders game.

After much experimentation I decided to detach the invader sprites from the engine's entity manager because the code to move the invaders was too convoluted with those sprites being embedded. Although the entity manager does a lot of work for us, in some cases (such as this) it actually becomes a hindrance. No matter; it's just as feasible handling the invader sprites in the front-end game by responding to bullet update events instead of collision events. Of course, things like bullets and explosions are still best left to the entity manager.

Epilogue

I would be remiss if I didn't leave you with some ideas for further study. I want to mention a significant game project that I have been involved with for the last year and a half, because this project led to many of the C++ classes presented in this book. *Starflight: The Lost Colony* is an official sequel to the original *Starflight* series, developed by Binary Systems (owned by Rod McConnell) and published by Electronic Arts. Many older gamers who remember playing *Starflight* in the late 1980s and early 1990s (in particular, with the Sega Genesis port) consider this to be the most enjoyable adventure sci-fi game ever made. Our goal with *The Lost Colony* was to re-create the atmosphere of exploration and adventure in the original games, while upgrading the gameplay to modern standards. Visit www.starflightgame.com for more details about this freeware game and chat with the game's developers about their experiences.

Here's a quick perusal of the game. The artwork is truly what makes this game so engaging! (Art credit: Andrew Chason and Ronald Conley.) Figure 13.5 shows the title screen, which is animated.

Figure 13.6 shows the Starport commons, which contain various modules, such as crew assignments and ship upgrades.

Figure 13.7 shows an alien encounter taking place. Be careful in your attitude toward aliens, or you could wind up at the business end of a missile! Of course, the bread and butter of *Starflight* has always been about space exploration, and *The Lost Colony* delivers, as you can see in Figure 13.8.

Finally, we come to another major module of the game—planet surface exploration. See Figure 13.9. This is perhaps the most enjoyable part of the game—landing on planets, collecting interesting life forms (or running away from them!), mining for valuable minerals, and trading with alien civilizations. The tiled scroller used for the planet surface module evolved from the one used for space travel in order to

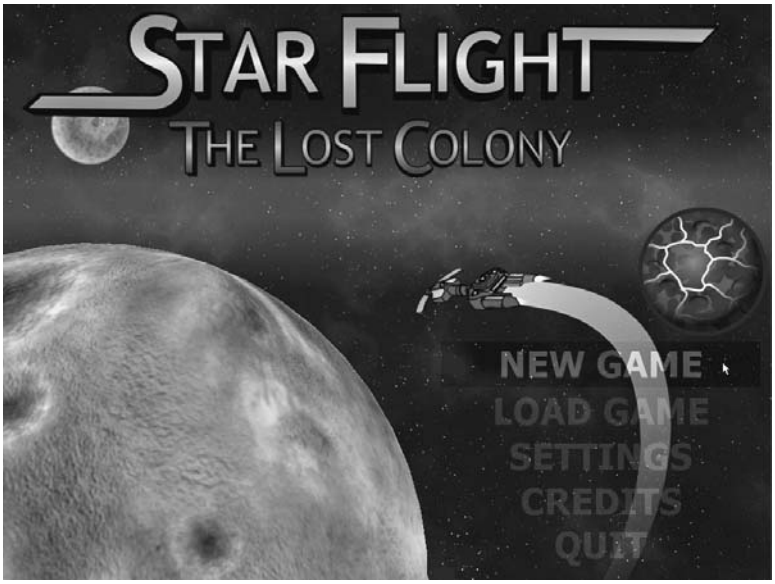


Figure 13.5
The title screen of *Starflight*.

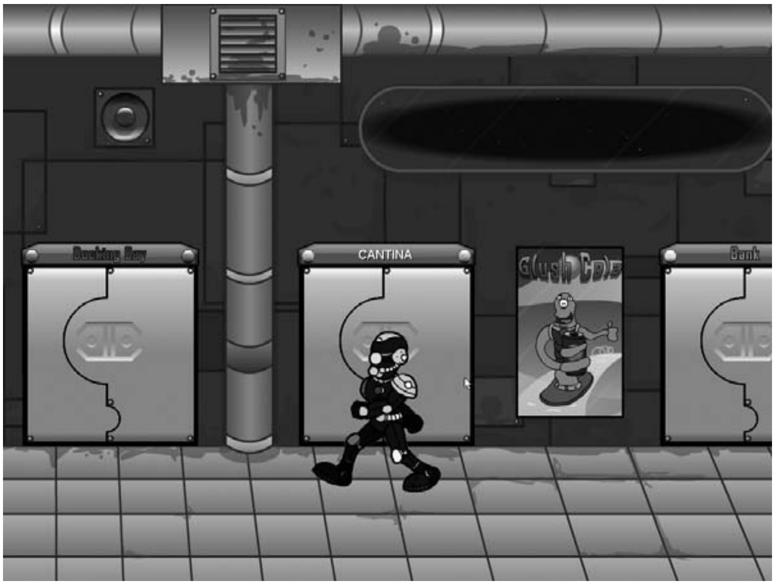
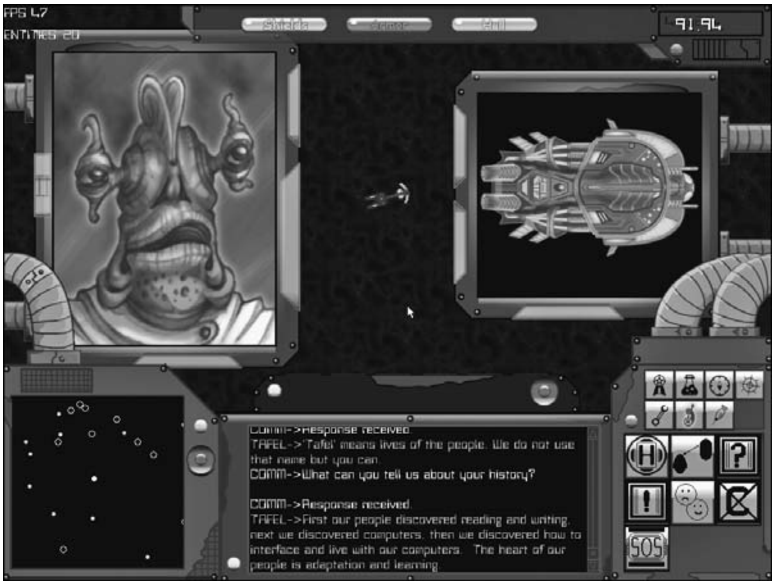
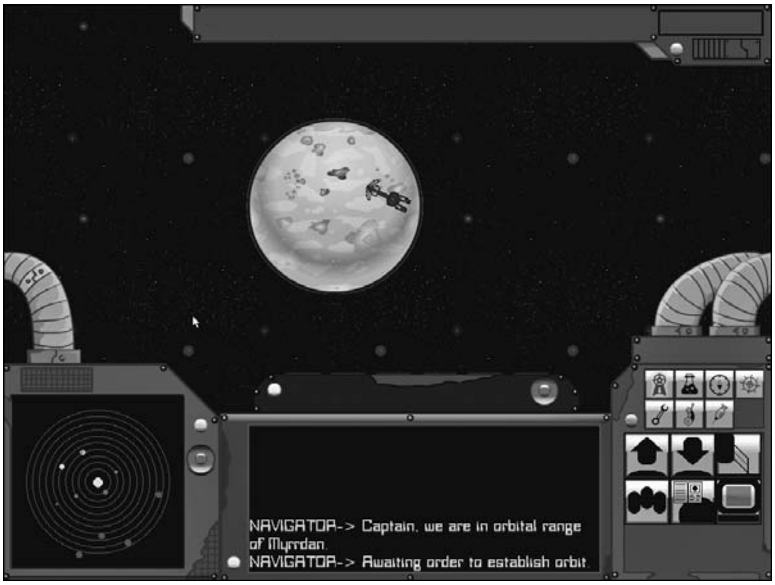


Figure 13.6
The Starport is the center of action for human space travelers.



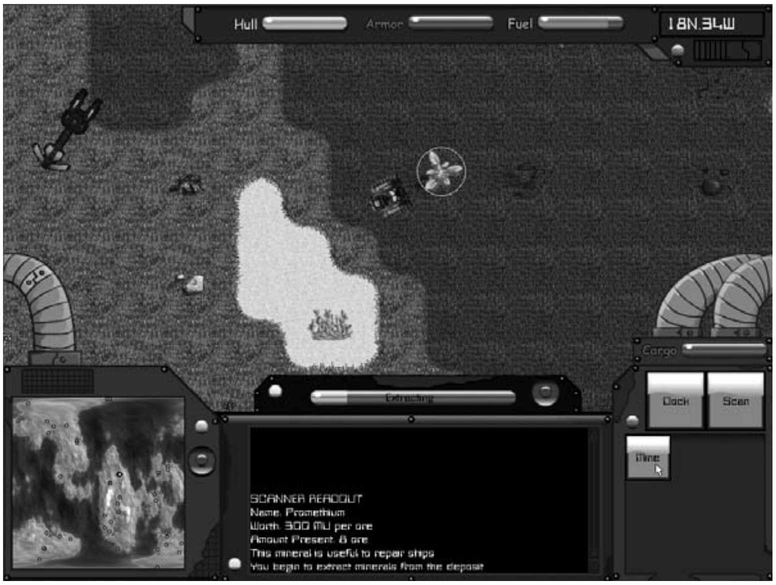
```
COMM->Response received.
TAFEL->'Tafel' means lives of the people. We do not use
that name but you can.
COMM->What can you tell us about your history?

COMM->Response received.
TAFEL->First our people discovered reading and writing,
next we discovered computers, then we discovered how to
interface and live with our computers.  The heart of our
people is adaptation and learning.
```

Figure 13.7
Engaging alien civilizations.



```
NAVIGATOR-> Captain, we are in orbital range
of Myrrdan.
NAVIGATOR-> Awaiting order to establish orbit.
```

Figure 13.8
Exploring star systems is what it's all about.



```
SCANNER READOUT
Name: Promethium
Worth: 300 MU per ore
Amount Present: 8 ore
This mineral is useful to repair ships
You begin to extract minerals from the deposit
```

Figure 13.9
Exploring a planet surface.

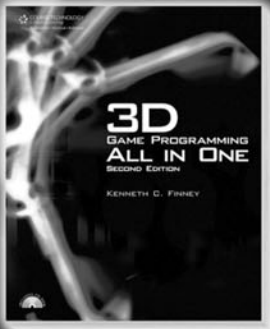
provide curved edges and two layers of overlay to improve its appearance. The tile map is generated from the random planet texture (the same one viewed from planet orbit).

Advice

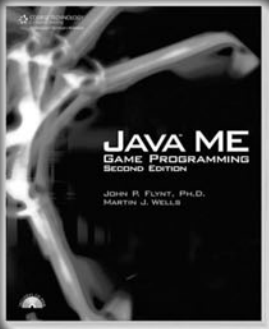
To continue the adventure, come join the game programming forum at www.jharbour.com/forum, where we discuss new features in the Advanced2D engine, hold programming contests, and talk about new demos and games. This forum is where I post all new updates to the engine and game examples. Best of all, this forum is not advertiser driven. I look forward to chatting with you online!

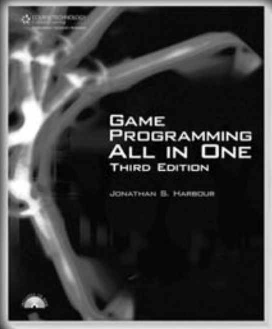
INDEX

GOT GAME?


3D
Game Programming
All in One
Second Edition
Kenneth C. Finney

XNA
Game Studio Express
Developing Games for Windows and the Xbox 360


Java ME
Game Programming
Second Edition
John P. Flynt, Ph.D.
Martin J. Wells


Game
Programming
All in One
Third Edition
Jonathan S. Harbour