Pangea Software's Ultimate Game Programming Guide for Mac OS X

Brian Greenstone

Pangea Software, Inc.
12405 John Simpson Ct.
Austin, TX 78732
www.pangeasoft.net

Printed in the United States of America
04 7 6 5 4 3 2 First Edition

ISBN: 0-9761505-0-6

Grant of License
Purchasers of this book may freely include the source code contained in this book in their own software projects.

Limits of Liability and Disclaimer of Warranty
The Author and Publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the claims and programs to determine their effectiveness. The Author and Publisher make no warranty of any kind, expressed or implied, with regard to these programs or the written documentation contained in this book. The Author and Publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Trademarks
Pangea Software, the Pangea Software logo, Bugdom, Nanosaur, Cro-Mag Rally, Enigmo, Weekend Warrior, and Otto Matic are registered trademarks of Pangea Software, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, Pangea Software, Inc. was aware of a trademark claim. Apple, the Apple logo, Mac, Macintosh, QuickDraw, QuickTime, Sherlock, Carbon, Rendezvous are trademarks of Apple Computer, Inc. registered in the United States and other countries.

Special thanks to the following people who have made my life easier over the years in this business: Chris Bentley and the entire ATI group, George Warner, Eric Klein, Tuncer Deniz, Peter Cohen, Phil Sulak, and many others who I'm sure I'll hear from since I forgot to put their names in this list.

Table of Contents

Introduction **1**

Chapter 1: Development Tools for the Mac **3**

- Xcode 3
- Interface Builder 5
- CHUD 6
- OpenGL Profiler & Shader Builder 8
- Documentation & Sample Code 9

Chapter 2: Choosing a Video Mode **11**

- Getting a List of Valid Display Modes 13
- Letting the User Choose a Video Mode 17

Chapter 3: OpenGL for the Mac **25**

- Initializing OpenGL 26
- Drawing an OpenGL Scene 29
- Working with Text in OpenGL 30
- Displaying Windows in a Full-Screen Context 33

Chapter 4: OpenGL Optimizations **37**

- Macro Optimizations 37
- Caching the OpenGL State 38
- The Transform Hint 43
- Normals 43
- Colors 44
- Reading Pixels 44
- Know when to use glBegin/End or Vertex Arrays 44
- Optimizing VRAM 46

Chapter 5: PowerPC Math Optimizations **49**

- AltiVec for Faster Matrix Multiplies 49
- Fast Vector Normalizing 52

Chapter 6: Vertex Array Range **55**

- Initializing Vertex Array Range 55
- Drawing with VAR 60
- Issues with VAR 62

Chapter 7: Calculating the Frame Rate **65**

Chapter 8: Gamma Fades **69**

CGSetDisplayTransferByFormula 70
CGSetDisplayTransferByTable 73

Chapter 9: Carbon Events 75

Event Loop Timers 75
Menu Bars 78
Preventing Your Game From Going to Sleep 83

Chapter 10: Audio 85

Quicktime for Music Playback 85
The Sound Manager for Playing Effects 98
OpenAL 111

Chapter 11: Simple Input 125

Reading the Keyboard with GetKeys() 125
Reading the Keyboard with Carbon Events 129
Reading the Mouse 133

Chapter 12: Input with the HID Manager 139

The Good, the Bad, and the HID Manager 139
Some HID Manager Terminology 141
Getting the List of HID Devices 142
Getting a Device's Elements 152
Reading Input Data 169
Input.c 171

Chapter 13: Writing a Maya Plug-in 175

Initializing A Maya Plug-In 175
The Plug-In Entry Point 177
Getting the Scene's Layer Info 179
Extracting Geometry Data 184
Extracting Shader Data 189
Installing the Maya Plug-In 195
The BG3D File Format 198
BG3D Linker 199
Loading and Using BG3D Files 201

Chapter 14: Stereo 3D 203

Types of Stereo Glasses 203
Stereo Camera Calculations 204
Rendering for Anaglyph Glasses 209
Anaglyph Color Balancing 212
Rendering for LCD Shutter Glasses 219
Shutter Glasses Hardware 223
Fun with Anaglyphs 224

Chapter 15: Networking **229**

Rendezvous 229
BSD Sockets 238

Chapter 16: Copy Protection **249**

Serial Numbers 249
Hackerproofing 252

Chapter 17: Miscellaneous Mac Tidbits **257**

Setting the Default Directory 257
Finding the Preferences Folder 258
Language Determination 259
Filenames 262
Loading Images with Quicktime 263

Chapter 18: Marketing & Selling **265**

Marketing Your Game 265
Game Demos 267
Selling Retail 269
Selling Online 270

Introduction

Walk into any bookstore or do a search on Amazon.com, and you'll find dozens and dozens of game programming books. Unfortunately, all but a few of these books are written assuming you're on a PC, using Direct 3D, Direct Sound, and all the other Windows-specific technologies. Some of these books have good information about general game programming techniques, but all that information won't do you any good on a Macintosh unless you know how to build a Mac-specific game engine. There are many things you need to know about building a game engine for Mac OS X that are different from the PC, and the goal of this book is to show them to you.

We're going to start from the ground up as we build a small 3D game engine shell to get you started in your Mac game programming endeavors. This book isn't about how to write or design a game, but rather how to build a game engine for Mac OS X. Those other books on game programming can teach you general game programming technique dealing with AI, collision detection, world building, etc., but this book's focus is specifically on the Macintosh technologies that you need to know to get a game engine up and running. If you want to make a career out of developing games for the Mac, this book also covers topics dealing with the business end of things including copy protection, distribution, and marketing.

I have been programming games on the Mac since 1991, and running Pangea Software for even longer. Over those years I've accumulated quite a stockpile of useful techniques to get the best performance out of a game engine on this platform. I've also encountered several bugs with the OS that can often present issues for game programmers, so I'll be sure to point all of those out to make you aware of them.

The CD that comes with this book contains sample projects all written in straight C (none of that C++ gibberish) using Apple's Xcode compiler. These projects include all of the code examples that are discussed in each of the following chapters, and as we progress through these chapters, we'll continue to build upon the game engine. By the time we get to the end of the book you'll have a robust set of functions that you can use to start coding your own games for the Mac.

Chapter 1: Development Tools for the Mac

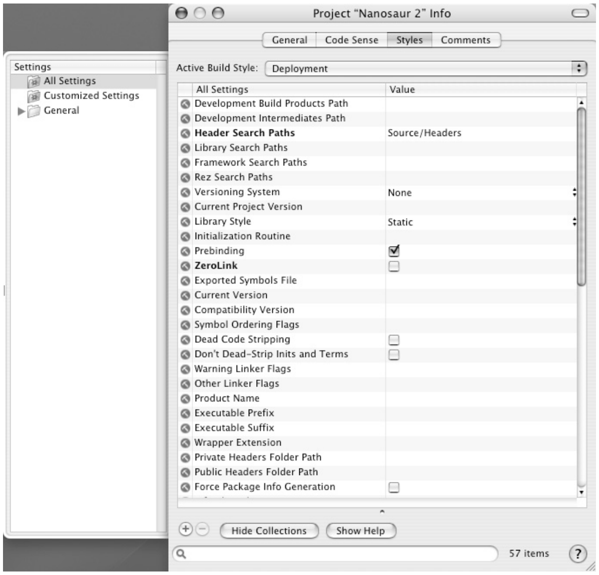
One of the great things about developing for the Macintosh is that all of the basic tools you need to build an application are provided free from Apple, and they're very good and powerful tools:

Xcode

All of the sample projects in this book are built with Apple's new Xcode compiler (version 1.5). Prior to Xcode the compiler of choice for any serious Mac programming was Metrowerks' CodeWarrior, but as the price of CodeWarrior has continued to increase over the years it hasn't evolved enough to justify the expense. Luckily for Mac developers Apple has leapfrogged over CodeWarrior with the totally free Xcode compiler. There are still some things that CodeWarrior is better at than Xcode, but the technology behind Xcode is much more modern and has a great future ahead of it.

Programmers are often afraid of moving from one development environment to another because when things change they're left a bit disoriented, and incompatibilities with existing code can pop up. This was definitely the case when Apple released Project Builder at the onset of OS X. Project Builder was a mess, and no serious game programmer would use it. Apple knew this and wanted to make the transition from CodeWarrior to Xcode a smooth one, so in addition to being able to import CodeWarrior projects, you can also configure Xcode to use the CodeWarrior key shortcuts for just about everything.

The only significant difference between the CodeWarrior and Xcode interfaces is how the project and target settings are handled. In CodeWarrior there was a nice, clean Project Settings dialog that anyone could understand. Xcode has the "Inspector" window that lets you set all sorts of project preferences. Unfortunately, the Inspector window is a bit overwhelming and is poorly documented. It requires more low-level, command line compiler knowledge than most Mac programmers are used to:



Active Build Style: Deployment

All Settings	Value
Development Build Products Path	
Development Intermediates Path	
Header Search Paths	Source/Headers
Library Search Paths	
Framework Search Paths	
Rez Search Paths	
Versioning System	None
Current Project Version	
Library Style	Static
Initialization Routine	
Prebinding	☑
ZeroLink	☐
Exported Symbols File	
Current Version	
Compatibility Version	
Symbol Ordering Flags	
Dead Code Stripping	☐
Don't Dead-Strip Inits and Terms	☐
Warning Linker Flags	
Other Linker Flags	
Product Name	
Executable Prefix	
Executable Suffix	
Wrapper Extension	
Private Headers Folder Path	
Public Headers Folder Path	
Force Package Info Generation	☐

Figure 1-1: The Xcode Inspector window

These are some of the key features that Xcode has over CodeWarrior:

DISTRIBUTED BUILDS

This feature lets Xcode send different source files to multiple processors and/or machines on the network for simultaneous parallel compilation. So, if you're on a dual processor machine, Xcode will compile two .c files simultaneously which obviously speeds up your total project build time. If you have, say, a total of three dual-processor G5's on your local network then Xcode can distribute .c files to all of them, and theoretically be compiling six files simultaneously!

PREDICTIVE COMPILATION

Most of the time when you're working on a project you're just sitting there typing, and the CPU is spinning its wheels doing nothing. But Xcode is smart. Instead of just sitting there doing nothing, it will start compiling files in the background. That way when you're ready to build your project, most of the .c files have already been compiled, and all Xcode needs to do is link.

GCC

While CodeWarrior uses it's own proprietary compiler, Xcode uses GCC which is a much more robust compiler that gives you access to just about every compilation parameter that you could ever want. It also issues far superior compile warnings. For example, take a look at the following code:

```
int TestFunc(void)
{
    int a, b, c;

    c = 0;
    a = 5;

    return(a);
}
```

CodeWarrior would not do a very good job with this function because it would only report a warning that the variable `b` is unused. However, logically both `b` and `c` are not used, and GCC let's us know it. Even though the variable `c` is assigned a value of `0` it is never used to actually do anything. This one feature alone is worth switching from CodeWarrior to Xcode since it really helps eliminate garbage from your code. GCC is smart enough to even detect this on global variables. If you convert an old CodeWarrior project to Xcode and recompile, you'll be amazed at how many "unused variable" warnings come up that you never saw with CodeWarrior.

Xcode is not perfect, however. I won't bother listing all the problems with it since Apple has been rapidly fixing bugs and releasing updates, so any problems I were to list here would likely become invalid in the near future. But suffice to say that Xcode's main flaws deal with stability and ease of use. Xcode's Inspector interface is a nebulous monster compared with CodeWarrior's simple project Settings dialog, so just setting up the parameters of a project with Xcode is a project all in itself and can be quite overwhelming to new users. The best thing to do is to start with one of the sample projects included with this book, and then go from there.

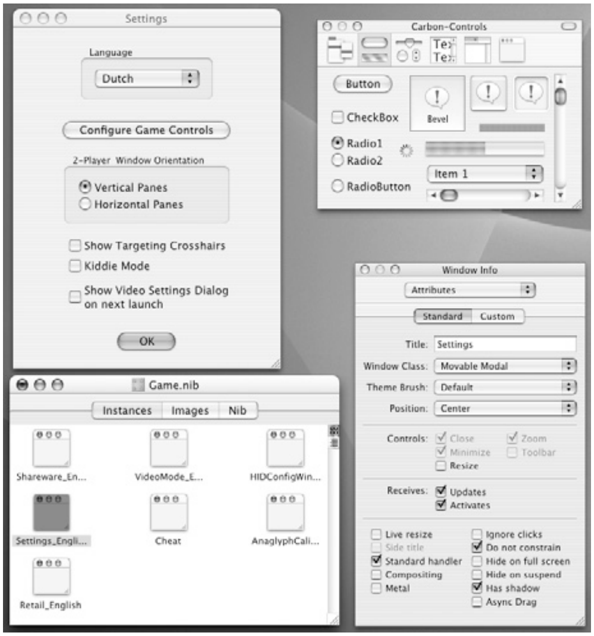
There is a wonderful document on Apple's web site that details the differences between Xcode and CodeWarrior, and it provides a very good guide for transitioning from CodeWarrior to Xcode. The document can be found here:

http://developer.apple.com/tools/switchtoxcode.html

Interface Builder

Interface Builder is a very powerful interface creation tool, but since games usually have their own graphical interfaces for most things, we only need to use this tool to create some basic dialogs and menus. In my games, I usually do the Video Configuration, Input Configuration, Registration, and Settings dialogs as standard Mac OS X dialogs using Interface Builder. The rest of the interface is done as custom in-game screens such as menu screens, high score screens, etc.

For those programmers who are used to using ResEdit or Resourcerer on Mac OS 9, Interface Builder is very similar, yet much more powerful.



Language: Dutch

Configure Game Controls

2-Player Window Orientation
● Vertical Panes
○ Horizontal Panes

☐ Show Targeting Crosshairs
☐ Kiddie Mode
☐ Show Video Settings Dialog on next launch

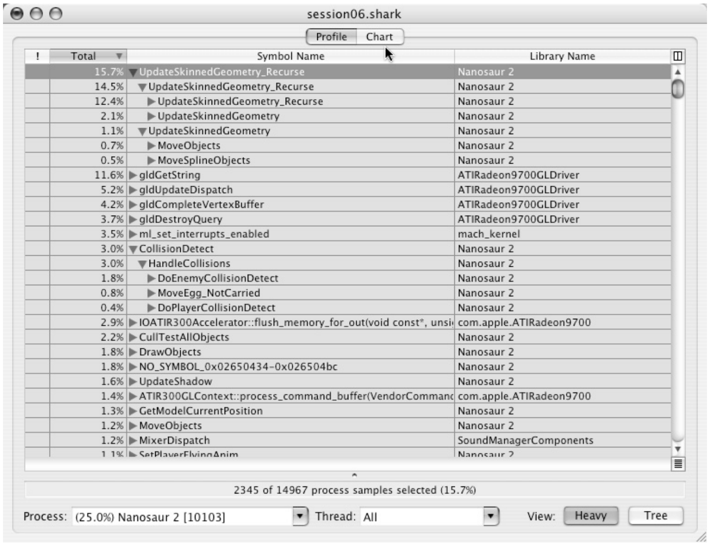
OK

Title: Settings
Window Class: Movable Modal
Theme Brush: Default
Position: Center
Controls: ☑ Close ☑ Zoom ☑ Minimize ☐ Toolbar ☐ Resize
Receives: ☑ Updates ☑ Activates
☐ Live resize
☐ Side title
☑ Standard handler
☐ Compositing
☐ Metal
☐ Ignore clicks
☑ Do not constrain
☐ Hide on full screen
☐ Hide on suspend
☑ Has shadow
☐ Async Drag

Figure 1-2: Interface Builder

CHUD

CHUD is the acronym that Apple uses to refer to its suite of Computer Hardware Understanding Development Tools. A pretty lousy acronym I know, but the tools are incredible! There are many tools that make up CHUD, and the most useful one is called Shark. Shark is a low-level performance optimization tool. It shows you exactly how much time is being spent in each function, on each line of each function, and even each opcode of each line.



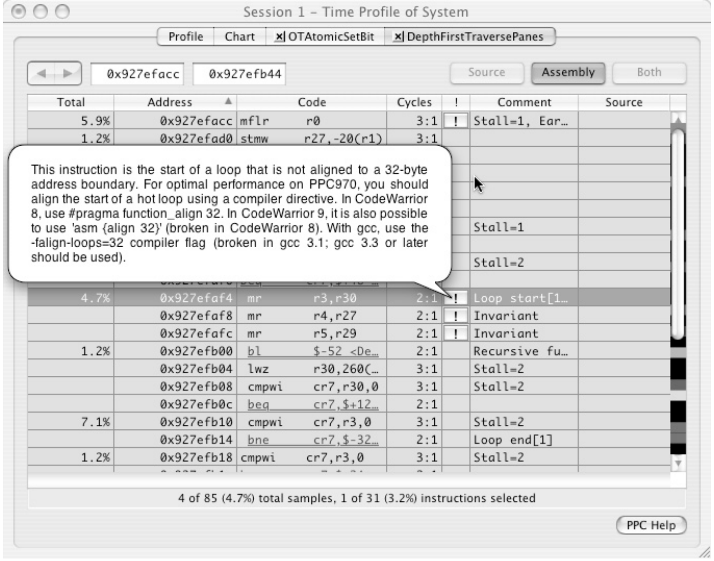
!	Total	Symbol Name	Library Name
	15.7%	▼UpdateSkinnedGeometry_Recurse	Nanosaur 2
	14.5%	▼UpdateSkinnedGeometry_Recurse	Nanosaur 2
	12.4%	▶UpdateSkinnedGeometry_Recurse	Nanosaur 2
	2.1%	▶UpdateSkinnedGeometry	Nanosaur 2
	1.1%	▼UpdateSkinnedGeometry	Nanosaur 2
	0.7%	▶MoveObjects	Nanosaur 2
	0.5%	▶MoveSplineObjects	Nanosaur 2
	11.6%	▶gldGetString	ATIRadeon9700GLDriver
	5.2%	▶gldUpdateDispatch	ATIRadeon9700GLDriver
	4.2%	▶gldCompleteVertexBuffer	ATIRadeon9700GLDriver
	3.7%	▶gldDestroyQuery	ATIRadeon9700GLDriver
	3.5%	▶ml_set_interrupts_enabled	mach_kernel
	3.0%	▼CollisionDetect	Nanosaur 2
	3.0%	▼HandleCollisions	Nanosaur 2
	1.8%	▶DoEnemyCollisionDetect	Nanosaur 2
	0.8%	▶MoveEgg_NotCarried	Nanosaur 2
	0.4%	▶DoPlayerCollisionDetect	Nanosaur 2
	2.9%	▶IOATIR300Accelerator::flush_memory_for_out(void const*, unsi	com.apple.ATIRadeon9700
	2.2%	▶CullTestAllObjects	Nanosaur 2
	1.8%	▶DrawObjects	Nanosaur 2
	1.8%	▶NO_SYMBOL_0x02650434-0x026504bc	Nanosaur 2
	1.6%	▶UpdateShadow	Nanosaur 2
	1.4%	▶ATIR300GLContext::process_command_buffer(VendorCommanc	com.apple.ATIRadeon9700
	1.3%	▶GetModelCurrentPosition	Nanosaur 2
	1.2%	▶MoveObjects	Nanosaur 2
	1.2%	▶MixerDispatch	SoundManagerComponents
	1.1%	▶SetPlayerFlyingAnim	Nanosaur 2

2345 of 14967 process samples selected (15.7%)

Process: (25.0%) Nanosaur 2 [10103] Thread: All View: Heavy Tree

Figure 1-3: A Shark profile of Nanosaur 2

Shark even goes so far as to offer suggestions on how to optimize specific parts of your code. In Figure 1-4 below you can see that under the Comment column Shark indicates problems with the execution performance, and if there is a "!" icon you can click on it to reveal some suggestions for improving performance:



Total	Address	Code	Cycles	!	Comment	Source
5.9%	0x927efacc	mflr r0	3:1	!	Stall=1, Ear…	
1.2%	0x927efad0	stmw r27,-20(r1)	3:1			
					Stall=1	
					Stall=2	
4.7%	0x927efaf4	mr r3,r30	2:1	!	Loop start[1…	
	0x927efaf8	mr r4,r27	2:1	!	Invariant	
	0x927efafc	mr r5,r29	2:1	!	Invariant	
1.2%	0x927efb00	bl $-52 <De…	2:1		Recursive fu…	
	0x927efb04	lwz r30,260(…	3:1		Stall=2	
	0x927efb08	cmpwi cr7,r30,0	3:1		Stall=2	
	0x927efb0c	beq cr7,$+12…	2:1			
7.1%	0x927efb10	cmpwi cr7,r3,0	3:1		Stall=2	
	0x927efb14	bne cr7,$-32…	2:1		Loop end[1]	
1.2%	0x927efb18	cmpwi cr7,r3,0	3:1		Stall=2	

This instruction is the start of a loop that is not aligned to a 32-byte address boundary. For optimal performance on PPC970, you should align the start of a hot loop using a compiler directive. In CodeWarrior 8, use #pragma function_align 32. In CodeWarrior 9, it is also possible to use 'asm {align 32}' (broken in CodeWarrior 8). With gcc, use the -falign-loops=32 compiler flag (broken in gcc 3.1; gcc 3.3 or later should be used).

4 of 85 (4.7%) total samples, 1 of 31 (3.2%) instructions selected

Figure 1-4: Shark giving tips on how to optimize your code

The CHUD tools and documentation can be found on Apple's web site at:

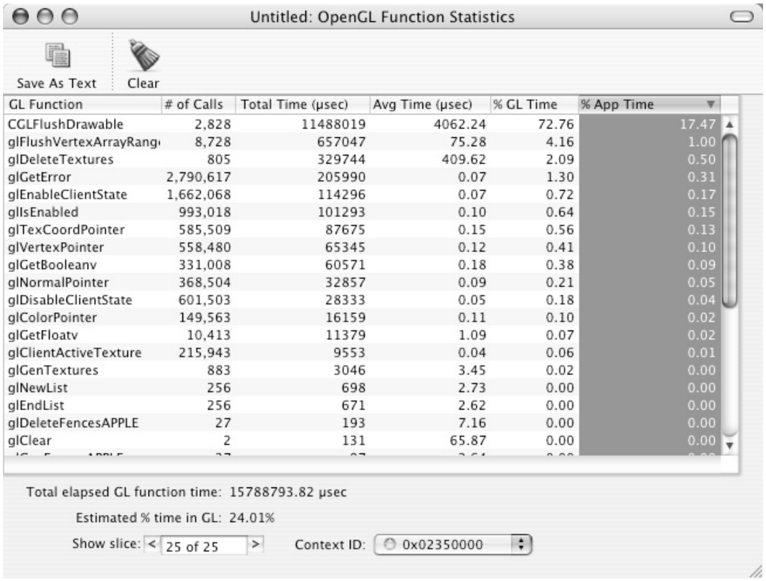
http://developer.apple.com/tools/performance

Once installed, the tools are found in your Developer/Applications/Performance Tools folder of your boot drive. You must download and install these tools yourself because they are not installed by default as part of the OS X Developer Tools.

OpenGL Profiler & Shader Builder

Since all games on the Mac use OpenGL for their 2D and 3D rendering Apple has provided some very useful OpenGL tools:

OpenGL Profiler is similar to the Shark tool in CHUD except that it details the performance of the OpenGL sub-system in your game. As you run your game the OpenGL Profiler tool will track all of the OpenGL calls that you make, and build a statistical database of everything that's going on. This way you can easily determine which OpenGL calls your application is spending its time in.



GL Function	# of Calls	Total Time (µsec)	Avg Time (µsec)	% GL Time	% App Time
CGLFlushDrawable	2,828	11488019	4062.24	72.76	17.47
glFlushVertexArrayRang	8,728	657047	75.28	4.16	1.00
glDeleteTextures	805	329744	409.62	2.09	0.50
glGetError	2,790,617	205990	0.07	1.30	0.31
glEnableClientState	1,662,068	114296	0.07	0.72	0.17
glIsEnabled	993,018	101293	0.10	0.64	0.15
glTexCoordPointer	585,509	87675	0.15	0.56	0.13
glVertexPointer	558,480	65345	0.12	0.41	0.10
glGetBooleanv	331,008	60571	0.18	0.38	0.09
glNormalPointer	368,504	32857	0.09	0.21	0.05
glDisableClientState	601,503	28333	0.05	0.18	0.04
glColorPointer	149,563	16159	0.11	0.10	0.02
glGetFloatv	10,413	11379	1.09	0.07	0.02
glClientActiveTexture	215,943	9553	0.04	0.06	0.01
glGenTextures	883	3046	3.45	0.02	0.00
glNewList	256	698	2.73	0.00	0.00
glEndList	256	671	2.62	0.00	0.00
glDeleteFencesAPPLE	27	193	7.16	0.00	0.00
glClear	2	131	65.87	0.00	0.00

Total elapsed GL function time: 15788793.82 µsec

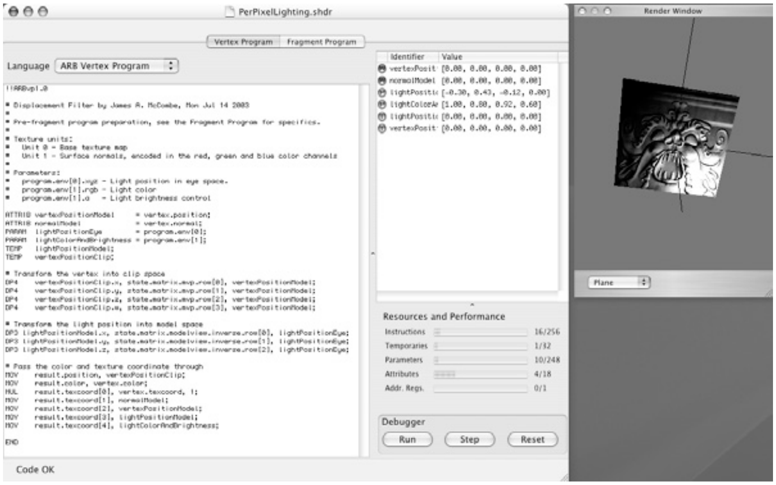
Estimated % time in GL: 24.01%

Show slice: 25 of 25

Context ID: 0x02350000

Figure 1-5: OpenGL Profiler statistics for Nanosaur 2

Shader Builder on the other hand is not a performance tool at all. This is a great little application for quickly writing and testing vertex and fragment shader programs supported by all new 3D video hardware. You use this to quickly develop special effects for your game to do things like bump mapping, cartoon shading, and fur.



```
!!ARBvp1.0

# Displacement Filter by James A. McCombe, Mon Jul 14 2003
#
# Pre-fragment program preparation, see the Fragment Program for specifics.
#
# Texture units:
#   Unit 0 - Base texture map
#   Unit 1 - Surface normals, encoded in the red, green and blue color channels
#
# Parameters:
#   program.env[0].xyz - Light position in eye space.
#   program.env[1].rgb - Light color
#   program.env[1].a   - Light brightness control

ATTRIB vertexPositionModel     = vertex.position;
ATTRIB normalModel             = vertex.normal;
PARAM  lightPositionEye        = program.env[0];
PARAM  lightColorAndBrightness = program.env[1];
TEMP   lightPositionModel;
TEMP   vertexPositionClip;

# Transform the vertex into clip space
DP4    vertexPositionClip.x, state.matrix.mvp.row[0], vertexPositionModel;
DP4    vertexPositionClip.y, state.matrix.mvp.row[1], vertexPositionModel;
DP4    vertexPositionClip.z, state.matrix.mvp.row[2], vertexPositionModel;
DP4    vertexPositionClip.w, state.matrix.mvp.row[3], vertexPositionModel;

# Transform the light position into model space
DP3 lightPositionModel.x, state.matrix.modelview.inverse.row[0], lightPositionEye;
DP3 lightPositionModel.y, state.matrix.modelview.inverse.row[1], lightPositionEye;
DP3 lightPositionModel.z, state.matrix.modelview.inverse.row[2], lightPositionEye;

# Pass the color and texture coordinate through
MOV    result.position, vertexPositionClip;
MOV    result.color, vertex.color;
MUL    result.texcoord[0], vertex.texcoord, 1;
MOV    result.texcoord[1], normalModel;
MOV    result.texcoord[2], vertexPositionModel;
MOV    result.texcoord[3], lightPositionModel;
MOV    result.texcoord[4], lightColorAndBrightness;

END
```

Code OK

Identifier	Value
vertexPosit	[0.00, 0.00, 0.00, 0.00]
normalModel	[0.00, 0.00, 0.00, 0.00]
lightPositi	[-0.30, 0.43, -0.12, 0.00]
lightColorAr	[1.00, 0.80, 0.92, 0.60]
lightPositi	[0.00, 0.00, 0.00, 0.00]
vertexPosit	[0.00, 0.00, 0.00, 0.00]

Resources and Performance

Instructions	16/256
Temporaries	1/32
Parameters	10/248
Attributes	4/18
Addr. Regs.	0/1

Figure 1-6: Shader Builder showing a per-pixel lighting fragment program

Once you've written a shader program with Shader Builder you can save it out to a text file and then your code can read it in with the usual OpenGL Shader Program function `glProgramStringARB()`. For information on how to program and use Shader Programs with OpenGL you should consult the book called "The OpenGL Extensions Guide".

Documentation & Sample Code

Whenever you install any of Apple's Developer tools you also get lots of documentation and sample code. Take a look in the Developer folder on your OS X boot drive and you'll see a folder full of great stuff! All of the tools mentioned above reside in there. The Documentation folder holds a massive amount of information about the tools and also the programming API's for the Mac. Then there's the Examples folder that has tons and tons of sample apps for everything from CHUD to OpenGL Shader Programs to simple Carbon demos.

Chapter 2: Choosing a Video Mode

The first thing we'll want to do to get a game up and running is to either create a window to play the game in, or to take over the whole screen if we want to play full-screen. Your games should always go full-screen since you get the maximum 3D rendering performance that way, and games that go full-screen look a lot more professional than ones that run in a window. However, even if your game normally plays full-screen you will always want to have a stand-by windowed mode for debugging. On OS 9 you could still debug a game that was running full-screen since the CodeWarrior debugger's windows would always appear on top of the game. However, on OS X this is never the case. If you set a breakpoint in your game and the debugger stops on it you'll have no way of seeing it since all of the debugger's windows will be hidden underneath the game. The only way around this is to use multiple monitors or to do remote debugging, but most people don't have such fancy setups, so testing in a windowed mode is the best way to go.

In the days of Mac OS 8 and 9 there was a technology called Game Sprockets. This was a set of libraries that made writing games on the Mac a much easier process than they are today. Unfortunately, Apple didn't bring most of the Game Sprockets to OS X, so we've now got to do a lot of things by hand. One Game Sprocket that they did bring over to OS X is Draw Sprocket, but I don't recommend using it. Instead, we will use Core Graphics API calls to get information about our displays, and to set our screen mode. This gives us much better control over our display than would Draw Sprocket.

Listing 2-1: Switching the Display Mode

```
CGDirectDisplayID   gCGDisplayID;
short               gGameWindowWidth, gGameWindowHeight;

void SwitchDisplayMode(int width, int height, int depth)
{
    CFDictionaryRef     refDisplayMode = 0;


        /* GET THE MAIN DISPLAY & CAPTURE IT */

    gCGDisplayID = CGMainDisplayID();
    CGDisplayCapture(gCGDisplayID);


            /* FIND BEST MATCH FOR WHAT WE WANT */

    refDisplayMode = CGDisplayBestModeForParameters(
                    gCGDisplayID,
                    depth,              // 16 or 32-bit depth
                    width,              // horiz rez
                    height,             // vert rez
                    nil);

    if (refDisplayMode == nil)
        DoFatalAlert("\pCGDisplayBestModeForParameters failed!");


                /* SWITCH TO IT */

    CGDisplaySwitchToMode(gCGDisplayID, refDisplayMode);


        /* GET THE ACTUAL WIDTH/HEIGHT OF OUR SCREEN */

    gGameWindowWidth    = CGDisplayPixelsWide(gCGDisplayID);
    gGameWindowHeight   = CGDisplayPixelsHigh(gCGDisplayID);
}
```

The first thing that `SwitchDisplayMode()` does is to “capture” the main display (the main display is whichever display has the menu bar):

```
    CGDisplayCapture(gCGDisplayID);
```

What capturing does is tell the system that your application wants to reserve this display for it’s own exclusive use. No other application can change this display’s configuration. Then we ask Core Graphics for the display mode that best matches our parameters:

```
    refDisplayMode = CGDisplayBestModeForParameters(
                    gCGDisplayID,
                    depth,              // 16 or 32-bit depth
                    width,              // horiz rez
                    height,             // vert rez
                    nil);
```

Three of the fundamental inputs to `CGDisplayBestModeForParameters()` are the desired bit-depth, horizontal resolution, and vertical resolution. If you’ve asked for a valid resolution supported by your display then this is most likely the mode that you’ll get, but if you’ve asked for a mode that your display cannot do, then Core Graphics will tweak your settings to the closest valid mode allowed by the display. In other words, if you ask for a screen resolution that’s 900x600, odds are that you’ll get 800x600 since no displays that I know of support a 900x600 mode, yet 800x600 is the nearest common resolution.

To cause the video mode switch to occur, we do this:

```
    CGDisplaySwitchToMode (gCGDisplayID, refDisplayMode);
```

Since there's no guarantee that the resolution we got is the one we asked for, we need to extract the actual resolution back out of our display:

```
    gGameWindowWidth    = CGDisplayPixelsWide(gCGDisplayID);
    gGameWindowHeight   = CGDisplayPixelsHigh(gCGDisplayID);
```

Getting a List of Valid Display Modes

Now that we know how to set the display to our desired mode, it would be a good idea to know what video modes are actually supported by the display so that the user can choose a resolution. To do this, we'll need to bring up a dialog with a pop-up menu from which the user can select video modes. Before we display any dialogs, however, we first need to create this list.

Listing 2-2: Creating a List of Display Modes

```
typedef struct
{
    int     rezH,rezV;
}VideoModeType;

short               gNumVideoModes = 0;
VideoModeType       gVideoModeList[MAX_VIDEO_MODES];


void CreateDisplayModeList(void)
{
    int             i, numDeviceModes;
    CFArrayRef      modeList;

    gNumVideoModes = 0;

                /* GET MAIN MONITOR ID */

    gCGDisplayID = CGMainDisplayID();


                /* GET LIST OF MODES FOR THIS MONITOR */

    modeList = CGDisplayAvailableModes(gCGDisplayID);
    if (modeList == nil)
        DoFatalAlert("\pCGDisplayAvailableModes failed!");

    numDeviceModes = CFArrayGetCount(modeList);
```

```
            /*********************/
            /* EXTRACT MODE INFO */
            /*********************/

    for (i = 0; i < numDeviceModes; i++)
    {
        CFNumberRef         numRef;
        CFDictionaryRef     mode;
        Boolean             skip;
        int                 j;
        long                width, height, dep;

                                    // Get the mode out of the list

        mode = CFArrayGetValueAtIndex( modeList, i );
        if (mode == nil)
            DoFatalAlert("\pCFArrayGetValueAtIndex failed!");

                                    // get mode’s width

        numRef = CFDictionaryGetValue(mode, kCGDisplayWidth);
        CFNumberGetValue(numRef, kCFNumberLongType, &width) ;

                                    // get mode’s height

        numRef = CFDictionaryGetValue(mode, kCGDisplayHeight);
        CFNumberGetValue(numRef, kCFNumberLongType, &height);

                                    // get mode’s depth

        numRef = CFDictionaryGetValue(mode, kCGDisplayBitsPerPixel);
        CFNumberGetValue(numRef, kCFNumberLongType, &dep);
        if (dep != 32)              // only look for 32-bit modes
            continue;


                /****************************/
                /* SEE IF ADD TO MODE LIST */
                /****************************/

                /* SEE IF IT'S A VALID MODE FOR US */

        if (CFDictionaryGetValue(mode,
            kCGDisplayModeUsableForDesktopGUI) != kCFBooleanTrue)
            continue;


                    /* CHECK IF ALREADY IN LIST */

        skip = false;
        for (j = 0; j < gNumVideoModes; j++)
        {
            if ((gVideoModeList[j].rezH == width) &&
                (gVideoModeList[j].rezV == height))
            {
                skip = true;
                break;
            }
        }

        if (!skip)
        {
                /* THIS REZ NOT IN LIST YET, SO ADD */

            if (gNumVideoModes < MAX_VIDEO_MODES)
            {
                gVideoModeList[gNumVideoModes].rezH = width;
                gVideoModeList[gNumVideoModes].rezV = height;
                gNumVideoModes++;
            }
        }
    }
}
```

Getting the list of video modes is simple:

```
    modeList = CGDisplayAvailableModes(gCGDisplayID);
```

Then we iterate thru `modeList` finding modes that we want to keep by calling a series of Core Foundation functions that give us information about each mode:

```
    mode = CFArrayGetValueAtIndex(modeList, i);
```

The variable `mode` is what Apple refers to as a "dictionary". It contains a lot of data, and to find the values of specific pieces of data we look them up in the dictionary with a call to `CFDictionaryGetValue()`:

```
    numRef = CFDictionaryGetValue(mode, kCGDisplayWidth);
```

The constant `kCGDisplayWidth` is a string that gets looked up in the dictionary and then that entry's value is returned. Unfortunately, `CFDictionaryGetValue()` does not return an actual integer number that we can use. Rather, it returns a `CFNumberRef` which is an opaque structure that contains our value, so we need to make another function call to get the integer value from it:

```
    CFNumberGetValue(numRef, kCFNumberLongType, &width)
```

In our code we are only using `kCGDisplayBitsPerPixel`, `kCGDisplayWidth`, and `kCGDisplayHeight`, however, there are quite a few additional parameters you can get for each display mode:

```
kCGDisplayMode
kCGDisplayBitsPerSample
kCGDisplaySamplesPerPixel
kCGDisplayRefreshRate
kCGDisplayModeUsableForDesktopGUI
kCGDisplayIOFlags
kCGDisplayBytesPerRow
```

So, if you wanted to determine the refresh rate of the current display mode then do this:

```
numRef = CFDictionaryGetValue(mode, kCGDisplayRefreshRate);
```

However, there is one very important thing to know about refresh rates, and that is that LCD displays will return a value of 0 even though their true refresh rate is typically 60hz. So, you should always special-case your code like this:

```
if (numRef == nil)
    refreshRate = 60;
else
{
    CFNumberGetValue(numRef, kCFNumberLongType, &refreshRate)
    if (refreshRate == 0)
        refeshRate = 60;
}
```

Going back to Listing 2-2, you'll notice that we're skipping any modes that are not 32-bit. The reason for this is that any resolutions that support 32-bits will also support 16-bits, however, there are times when the reverse is not true. So, we only need to track 32-bit modes since we can always make then 16-bit if we want.

It is also important to note that not all modes returned in this list are actually valid. This is a list of all video modes supported by the video card, but not necessarily by the display itself. Therefore, we need call this:

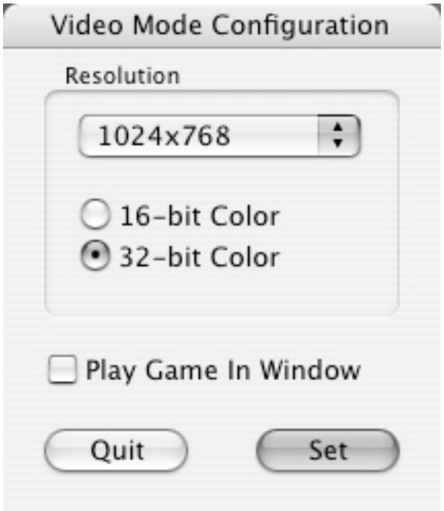
```
CFDictionaryGetValue(mode,  kCGDisplayModeUsableForDesktopGUI)
```

The result from this indicates if the mode is usable by the display. Some video cards can support huge resolutions like 2048x1365, but good luck finding a 17" monitor that can actually go that high. Testing for `kCGDisplayModeUsableForDesktopGUI` lets us know if the resolution is physically possible.

If the video mode checks out, then we add it to our list. Since different modes can have the same resolution, it is important to check for duplicate resolutions to avoid having duplicate information in our list. For example, there may be 5 different video modes at the 1024x768 resolution. The resolutions of these modes are all 1024x768, but there may be different refresh rates or other characteristics that we don't care about.

Letting the User Choose a Video Mode

All the pieces of the puzzle are now in place, so all that's left to do is ask the user what mode they'd like to use. The sample project titled "Video Mode.xcode" on the CD that comes with this book contains all the code listed above, plus the code we're about to cover below. This sample application displays a simple dialog box that lets you choose a video resolution and bit-depth, or you can opt to bring up a window to render into instead.



Resolution: 1024x768

○ 16-bit Color
● 32-bit Color

☐ Play Game In Window

Figure 2-1: The video mode configuration dialog

In the sample project you'll find a file called Game.nib. This file contains the resources for the dialog in Figure 2-1. In order to use this file in your game you need to get a reference to it with these two lines of code:

```
gBundle = CFBundleGetMainBundle();
CreateNibReferenceWithCFBundle(gBundle, CFSTR("Game"), &gNibs);
```

The first line gets a reference to the application's main bundle, and then the next line gets a reference to the nib file named "Game.nib" that's in that bundle. Note that you do not include the ".nib" suffix in the filename. Once you've done this you can easily access any nib resources in your application.

The following code shows how to load and process our dialog resource:

Listing 2-3: Displaying a Screen Mode Dialog

```
void DoScreenModeDialog(void)
{
    OSErr           err;
    EventHandlerRef ref;
    EventTypeSpec   list[] = {{kEventClassCommand,kEventProcessCommand}};
    ControlID       idControl;
    ControlRef      control;
    int             i;
    EventHandlerUPP winEvtHandler;

            /************************/
            /* INITIALIZE THE DIALOG */
            /************************/

            /* BUILD LIST OF VIDEO MODES FOR USER TO CHOOSE FROM */

    CreateDisplayModeList();


            /* CREATE DIALOG WINDOW FROM THE NIB */

    err = CreateWindowFromNib(gNibs, CFSTR("VideoMode"),
                            &gDialogWindow);
    if (err)
        DoFatalAlert("\pCreateWindowFromNib failed!");


            /* SET THE ITEMS IN THE POP-UP MENU */

    BuildResolutionMenu();


            /* CREATE NEW WINDOW EVENT HANDLER */

    winEvtHandler = NewEventHandlerUPP(DoScreenModeDialog_EventHandler);
    InstallWindowEventHandler(gDialogWindow,
                            winEvtHandler,
                            GetEventTypeCount(list),
                            list,
                            0,
                            &ref);


            /**********************/
            /* PROCESS THE DIALOG */
            /**********************/

    ShowWindow(gDialogWindow);
    RunAppModalLoopForWindow(gDialogWindow);


            /*********************/
            /* GET RESULT VALUES */
            /*********************/

            /* GET "PLAY IN WINDOW" CHECKBOX */

    idControl.signature     = 'wind';
    idControl.id            = 0;
    GetControlByID(gDialogWindow, &idControl, &control);
    gPlayInWindow = GetControlValue(control);


            /* GET "16/32" RADIO BUTTONS */

    idControl.signature     = 'bitd';
    idControl.id            = 0;
    GetControlByID(gDialogWindow, &idControl, &control);

    if (GetControlValue(control) == 1)
        gDesiredColorDepth = 16;
    else
        gDesiredColorDepth = 32;


            /* GET RESOLUTION MENU VALUE */

    idControl.signature     = 'rezm';
    idControl.id            = 0;
    GetControlByID(gDialogWindow, &idControl, &control);
    i = GetControlValue(control);

    gDesiredRezH = gVideoModeList[i-1].rezH;
    gDesiredRezV = gVideoModeList[i-1].rezV;


                /***********/
                /* CLEANUP */
                /***********/

    DisposeEventHandlerUPP (winEvtHandler);
    DisposeWindow (gDialogWindow);
```

```
}
```

The function `CreateWindowFromNib()` is all that we need to call to load the dialog's resource data. We simply supply this function with the name of the resource to load, "VideoMode", and a reference to our nib file.

Dialogs created from nibs only need a small amount of maintenance code to function because the OS handles most of the functionality of the dialog's controls. This maintenance is done by a window event handler that is set up with this code:

```
    winEvtHandler = NewEventHandlerUPP(DoScreenModeDialog_EventHandler);
    InstallWindowEventHandler(gDialogWindow, winEvtHandler,
                            GetEventTypeCount(list), list, 0, &ref);
```

We won't spend too much time here discussing the details of processing Macintosh dialogs since there is plenty of other literature that describes this process, but suffice to say that our window event handler doesn't need to do much except check if the user clicks the OK or Quit buttons:

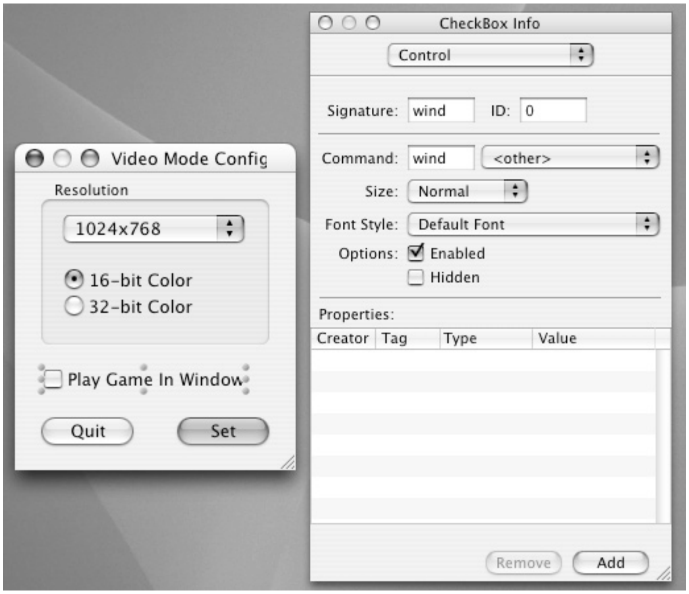
Listing 2-4: The Window Event Handler

```
pascal OSStatus DoScreenModeDialog_EventHandler(
                            EventHandlerCallRef myHandler,
                            EventRef            event,
                            void                *userData)
{
    HICommand       command;

    switch(GetEventKind(event))
    {

            /*******************/
            /* PROCESS COMMAND */
            /*******************/
            //
            // We only care if the user pressed the OK or Quit buttons.
            // Let the OS handle all other events.
            //

        case    kEventProcessCommand:
                GetEventParameter (event,
                                kEventParamDirectObject,
                                kEventParamHICommand,
                                NULL,
                                sizeof(command),
                                NULL,
                                    &command);

                switch(command.commandID)
                {
                            /* "OK" BUTTON */

                    case    'ok  ':
                            QuitAppModalLoopForWindow(gDialogWindow);
                            break;

                            /* "QUIT" BUTTON */

                    case    'quit':
                            ExitToShell();
                            break;
                }
                break;
    }

    return (eventNotHandledErr);
}
```

The checkboxes and radio buttons are handled automatically by OS X, so you don't need to manually deal with them like you would have had to do with the old fashioned resources on Mac OS 9. When the dialog's event loop terminates we've got some code that reads back the control values to determine what the user selected. Reading the value of a control is simple:

```
    idControl.signature     = 'wind';
    idControl.id            = 0;
    GetControlByID(gDialogWindow, &idControl, &control);
    gPlayInWindow = GetControlValue(control);
```

Just remember that the control's signature value is whatever you've assigned to that control in Interface Builder. I like to choose signatures that have some meaning relevant to the function of the control, so in the example above `'wind'` is a logical signature for the "Play Game in Window" checkbox as shown in Figure 2-2 below:



Resolution: 1024x768

● 16-bit Color

○ 32-bit Color

☐ Play Game In Window

Control

Signature: wind ID: 0

Command: wind <other>

Size: Normal

Font Style: Default Font

Options: ☑ Enabled

☐ Hidden

Properties:

Creator	Tag	Type	Value

Figure 2-2: The “Play Game in Window” checkbox has ‘wind’ signature

We already learned how to get a list of valid video modes earlier, so now we use that list of to build our dialog’s pop-up menu:

Listing 2-5: Building the Resolution Pop-Up Menu

```
void BuildResolutionMenu(void)
{
    Str255          menuStrings[MAX_VIDEO_MODES];
    short           i;
    Str32           s,t;
    MenuHandle      hMenu;
    Size            tempSize;
    ControlID       idControl;
    ControlRef      control;

            /* BUILD MENU ITEM STRINGS */

    for (i=0; i < gNumVideoModes; i++)
    {
        NumToString(gVideoModeList[i].rezH, s);     // “width”
        s[s[0]+1] = 'x';                            // “x”
        s[0]++;
        NumToString(gVideoModeList[i].rezV, t);     // “height”
        BlockMove(&t[1], &s[s[0]+1], t[0]);
        s[0] += t[0];
        BlockMove(&s[0], &menuStrings[i][0], s[0]+1);
    }


            /* GET MENU HANDLE FROM DIALOG */

    idControl.signature     = 'rezm';
    idControl.id            = 0;
    GetControlByID(gDialogWindow, &idControl, &control);
    GetControlData(control,kControlMenuPart,
                   kControlPopupButtonMenuHandleTag,
                   sizeof(MenuHandle), &hMenu,
                   &tempSize);

        /* MAKE SURE MENU IS EMPTY */

    DeleteMenuItems(hMenu, 1, CountMenuItems (hMenu));


            /* ADD NEW ITEMS TO THE MENU */

    for (i = 0; i < gNumVideoModes; i++)
        AppendMenu(hMenu, menuStrings[i]);


            /* FORCE UPDATE OF MENU EXTENTS */

    SetControlMaximum(control, gNumVideoModes);
    SetControlValue(control, 1);
}
```

To create a menu on the Mac you simply pass each menu item's Pascal string to the function `AppendMenu()`, and building these is just a matter of converting the width and height resolution values to strings. There are many ways to work with Pascal strings on the Mac, but I've always been a fan of the API calls `NumToString()` and `BlockMove()`. `NumToString()` creates a Pascal string from an integer value, and `BlockMove()` is an easy way to copy one Pascal string into another.

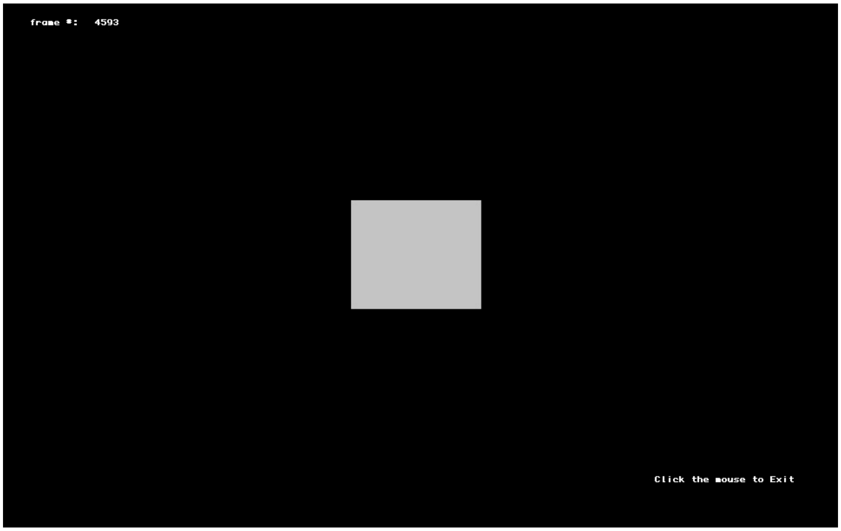
The last line of `BuildResolutionMenu()` calls `SetControlValue()` to tell the menu which menu item is selected by default. In an actual game you would want to set this to the menu item of the matching resolution stored in the game's preferences so that the user doesn't have to keep resetting the resolution each time this dialog comes up.

Chapter 3: OpenGL for the Mac

All games on the Mac now use OpenGL for graphics rendering, even if the game is a 2D sprite-based game. OpenGL provides an extremely fast and efficient way to draw images, sprites, polygons, etc., and Apple is putting a lot into supporting and updating it as new technologies emerge.

This book is not an OpenGL programming tutorial, so I would recommend buying the official “OpenGL Programming Guide” and “OpenGL Reference Manual” along with some other general OpenGL programming books like “The OpenGL Extensions Guide”. What I will primarily be covering are the parts of OpenGL that are specific to the Mac, and techniques for optimizing your OpenGL code for this hardware.

The sample project named “OpenGL Basics.xcode” demonstrates everything presented in this chapter. This sample application builds on what we learned in Chapter 2 by adding a new source file named OpenGL.c. which contains all of the new code. The new functions that we’ll be writing all start with the prefix “OGL_” to indicate that it is part of our new OpenGL support library.



```
frame #:  4593
Click the mouse to Exit
```

Figure 3-1: The OpenGL Basics sample application that draws a simple square

Initializing OpenGL

The Mac-specific subset of OpenGL is called AGL (Apple GL). The function prototypes and constants for AGL are found in the header file agl.h. We use several of these AGL functions to initialize an OpenGL draw context:

Listing 3-1: Initializing an OpenGL Draw Context

```
void OGL_CreateDrawContext(void)
{
    GDHandle        gdevice;
    AGLPixelFormat  fmt;

    GLint       attribWindow[]  =
    {
        AGL_RGBA, AGL_DOUBLEBUFFER, AGL_DEPTH_SIZE, 32,
        AGL_ALL_RENDERERS, AGL_ACCELERATED, AGL_NO_RECOVERY, AGL_NONE
    };
    GLint       attribFullScreen[] =
    {
        AGL_RGBA, AGL_FULLSCREEN, AGL_DOUBLEBUFFER, AGL_DEPTH_SIZE, 32,
        AGL_ALL_RENDERERS, AGL_ACCELERATED, AGL_NO_RECOVERY, AGL_NONE
    };

    gdevice = GetMainDevice();          // get the main display


            /***********************/
            /* CHOOSE PIXEL FORMAT */
            /***********************/

            /* PLAY IN WINDOW */

    if (gPlayInWindow)
        fmt = aglChoosePixelFormat(&gdevice, 1, attribWindow);

            /* FULL-SCREEN */
    else
        fmt = aglChoosePixelFormat(&gdevice, 1, attribFullScreen);

    if ((fmt == nil) || (aglGetError() != AGL_NO_ERROR))
        DoFatalAlert("\pCannot make a draw context!");

            /**********************/
            /* CREATE AGL CONTEXT */
            /**********************/

    gAGLContext = aglCreateContext(fmt, nil);
    if ((gAGLContext == nil) || (aglGetError() != AGL_NO_ERROR))
        DoFatalAlert("\paglCreateContext failed!");
```

```
            /* PLAYING IN A WINDOW */

    if (gPlayInWindow)
    {
        gAGLWin = (AGLDrawable)GetWindowPort(gGameWindow);
        aglSetDrawable(gAGLContext, gAGLWin);
    }

            /* PLAYING FULL-SCREEN */
    else
    {
        gAGLWin = nil;
        aglEnable(gAGLContext, AGL_FS_CAPTURE_SINGLE);
        aglSetFullScreen(gAGLContext, 0, 0, 0, 0);
    }

            /* ACTIVATE THE CONTEXT */

    aglSetCurrentContext(gAGLContext);


            /* NO LONGER NEED PIXEL FORMAT */

    aglDestroyPixelFormat(fmt);
}
```

The first call made is to `aglChoosePixelFormat()` where we pass in an array of constants that define the attributes of the draw context:

```
    fmt = aglChoosePixelFormat(&gdevice, 1, attribFullScreen);
```

A different list of characteristics is passed to `aglChoosePixelFormat()` depending on whether we are rendering full-screen or into a window. It is important to use all of these parameters, and only these parameters to achieve the maximum rendering performance from OpenGL. Here is a brief description of what each AGL attribute does:

AGL_RGBA
This tells OpenGL that we want our draw context to be in RGBA format, however, it should be noted that OpenGL ignores the alpha component even though all of the buffers created for the draw context are 32-bit, and thus, do have an alpha byte.

AGL_FULLSCREEN
This lets OpenGL know that we want to go full-screen so that it can optimize its pipeline to make the best use of that. Full-screen contexts can benefit from hardware page flipping,

thereby avoiding a slower pixel blitting that would otherwise be required to copy the back buffer to the screen.

`AGL_DOUBLEBUFFER`
This lets OpenGL know that we want to render double buffered. That means there are two drawing buffers: one that's currently being draw into and one that is currently being displayed. OpenGL page flips between the two buffers to get flicker-free animation.

`AGL_DEPTH_SIZE`
This constant is followed by a number that is either 16 or 32 depending on the desired bit-depth of the z-buffer. These days it is recommended that you always use a 32-bit z-buffer since VRAM is plentiful, and 16-bit z-buffers often result in drawing artifacts. Only use a 16-bit z-buffer if you're running low on VRAM and need to conserve it.

`AGL_ALL_RENDERERS`
This doesn't do anything particularly useful from the game programmer's point of view, but internally to OpenGL it lets the system know that all rendering engines are acceptable to the application.

`AGL_ACCELERATED`
This eliminates any software-only renderers from being chosen by OpenGL. It insures that only the fast, hardware renderers will be allowed.

`AGL_NO_RECOVERY`
This lets OpenGL follow an optimized pipeline by disabling all failure recovery systems. Typically OpenGL will resort to a software renderer if something were to go wrong with a hardware renderer (such as running out of VRAM), but with this option enabled OpenGL will simply fail instead.

`AGL_NONE`
You must put this at the end of the attributes list to indicate the end of the list.

The pixel format object is now used to create the draw context:

```
gAGLContext = aglCreateContext(fmt, nil);
```

After creating the context we need to activate it, but there are two different ways to create the draw context depending on if it's going to be associated with a full-screen display or with a window. To activate a context for a window we do this:

```
gAGLWin = (AGLDrawable)GetWindowPort(gGameWindow);
aglSetDrawable(gAGLContext, gAGLWin);
aglSetCurrentContext(gAGLContext);
```

To activate the context for a full-screen display we do this:

```
aglEnable(gAGLContext, AGL_FS_CAPTURE_SINGLE);
aglSetFullScreen(gAGLContext, 0, 0, 0, 0);
aglSetCurrentContext(gAGLContext);
```

The input parameters for `aglSetFullScreen()` are actually the display mode parameters width, height, and frequency; however, we pass 0's for all of these since we've already manually set our display (see Chapter 2). I don't recommend setting those display parameters with `aglSetFullScreen()` because each time you create and delete a draw context the screen will switch. By setting the display mode manually the way we did earlier, you can create and destroy OpenGL draw contexts as often as you want in your game, and there won't be any visual "glitching". Additionally, manually switching the display gives you more control over it later should you ever need to do things with it.

Drawing an OpenGL Scene

With the draw context created and active we are now free to start rendering some graphics into it. This is where a few more AGL functions need to be used.

Listing 3-2: Drawing the Scene

```
void OGL_DrawScene(void)
{
            /* MAKE OUR CONTEXT THE ACTIVE ONE */

    aglSetCurrentContext(gAGLContext);


            /* CLEAR THE DRAW BUFFER & Z-BUFFER */

    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);


                /* DO OUR GEOMETRY DRAWING HERE */

    DrawSceneGeometry();


            /* END RENDER & SWAP THE BUFFER TO MAKE IT VISIBLE */

    aglSwapBuffers(setupInfo->drawContext);
}
```

Most games only have one draw context active at any given time, but it's still a good idea to call `aglSetCurrentContext()` each pass through your render loop just to be safe. Before we start drawing anything we need to make sure that our drawing buffer and z-buffer are both cleared:

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

After this you are safe to start doing all your own OpenGL drawing to render your scene. When you're done submitting your geometry call `aglSwapBuffers()` to cause a page flip to occur.

```
    aglSwapBuffers(setupInfo->drawContext);
```

If you're in full-screen mode this page flip will be a fast hardware toggle, but if you're playing the game in a window then the drawing buffer gets blitted to the window instead.

An important thing to mention about `aglSwapBuffers()` is that the buffers are not necessarily swapped the instant you call this function. Modern video cards will queue up all of your drawing commands including the swap buffer command. Chances are that the video card is still drawing your geometry when you make this call, so it simply puts the swap command into its queue, and as soon as the video card is done drawing it'll execute the swap command. The beauty of this is that your program can proceed to calculating the next frame's data while the previous frame is being drawn by the video card – it's getting two things done simultaneously.

Working with Text in OpenGL

AGL has some built-in utility functions for easily displaying text in an OpenGL context. This is extremely useful for quick debugging and statistical displays. I use this feature in all of my games to display the frames per second, polygon counts, and other debug info. Working with text in AGL is very easy, and you only need to write a handful of functions to support it.

Listing 3-3: Initializing Fonts for OpenGL

```
GLuint  gFontList;

void OGL_InitFont(void)
{
    GLboolean   success;
    gFontList = glGenLists(256);

    success = aglUseFont(gAGLContext,        // our draw context
                        kFontIDMonaco,      // font to use
                        bold,               // style of font
                        9,                  // point size of font
                        0,                  // first character
                        256,                // # characters
                        gFontList);         // list to store font in


    if (!success)
        DoFatalAlert("\paglUseFont failed");
}
```

To create a font for AGL we first create an empty OpenGL "list" to contain bitmaps of all of the font characters:

```
    gFontList = glGenLists(256);
```

Next, our `OGL_InitFont()` function calls `aglUseFont()` which automatically generates the character bitmaps for each character in the requested font. You can specify which font you want to use, along with all the usual font parameters such as style and point size. Be careful about which font you choose to use because not all users may have your desired font installed. I always use Monaco since that's a standard Mac font, however, I've seen many situations where users have uninstalled Monaco from their systems and that causes some of our games to break. My standard reply to them is "Monaco is a system font which is required by many applications, so reinstall it."

To draw a string with this font, there are only three simple calls to make:

```
    glRasterPos2i(x, y);
    glListBase(gFontList);
    glCallLists(stringLength, GL_UNSIGNED_BYTE, cString);
```

Remember, however, that what's actually going on under the hood is that OpenGL is rendering a series of bitmaps using the current rendering state. Therefore, it is important to set the OpenGL state to something consistent as is done in this function:

Listing 3-4: Drawing a String with AGL Fonts

```
void OGL_DrawString(Str255 s, GLint x, GLint y)
{
            /* SET THE DRAWING MATRICES */
```

```
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, 640, 0, 480, -10.0, 10.0);


            /* SET SOME STATE VALUES */

    glDisable(GL_LIGHTING);
    glDisable(GL_TEXTURE_2D);
    glColor4f (1,1,1,1);


        /* SET COORDINATE TO DRAW STRING */

    glRasterPos2i(x, 480-y);


            /* DRAW THE STRING */

    glListBase(gFontList);
    glCallLists(s[0], GL_UNSIGNED_BYTE, &s[1]);
}
```

The most important thing that needs to be done before drawing the string is setting the projection matrix to a nice, flat, orthographic matrix so that we can draw the string to a specified screen coordinate. I like to set my ortho matrix to 640x480. Keep in mind, however, that this is not 640x480 in actual screen pixel coordinates, but rather it is 640x480 in world-space that gets scaled by OpenGL to whatever your true screen size is. An x-coordinate of 320 will always be in the middle of the screen no matter what resolution the screen is.

```
    glOrtho(0, 640, 0, 480, -10.0, 10.0);
```

After this matrix is set, lighting and texturing are turned off, and the current color is set to white. To set the coordinates of the first character in the string do this:

```
    glRasterPos2i(x, 480-y);
```

Drawing the string is done by setting the font list as the current list, and then performing the draw with the `glCallLists()` function:

```
    glListBase(gFontList);
    glCallLists(s[0], GL_UNSIGNED_BYTE, &s[1]);
```

Displaying Windows in a Full-Screen Context

There may be times while your full-screen game is running when you need to bring up a window, say, for a warning message, or maybe for a preferences dialog. These windows cannot be brought up safely when you've got an OpenGL draw context running in full-screen mode. In most cases, these windows will be drawn behind the OpenGL context so you cannot see them at all. It will appear that your application has locked up even though it hasn't.

So, before bringing up any windows or dialogs we need to enter a window-safe mode, and when we're done with that window we can exit the window-safe mode. To do this we'll need two new functions, `Enter2D()` and `Exit2D()`:

Listing 3-5: Entering 2D Window Mode from a Full-Screen Context

```
void Enter2D(void)
{
        /* DON'T DO ANYTHING IF IN WINDOWED MODE OR NO DISPLAY YET */

    if (gPlayInWindow || (gCGDisplayID == -1))
        return;

    InitCursor();                   // make sure there's a cursor


        /* NEED TO UN-CAPTURE THE DISPLAY */

    CGDisplayRelease(gCGDisplayID);

            /* DISABLE GL */

    if (gAGLContext)
    {
        glFlush();
        glFinish();

        aglSetDrawable(gAGLContext, nil);

        glFlush();
        glFinish();
    }
}
```

The `Enter2D()` function does two basic things: it releases the display and it disables OpenGL's full-screen draw context. The function `CGDisplayRelease()` will not change the display's mode at all – it will remain at the current resolution and color depth. The `aglSetDrawable()` call will then make the OpenGL drawing pane disappear so that we can see the Finder's desktop and interact with anything there. We're now free to bring up dialogs, work with menus, run other applications, etc.

Since the video mode stays at its current setting, be sure that your game's dialogs are never larger than the minimum screen resolution that your game supports. For example, if you're playing in 640x480 mode and you try to bring up a dialog that's 800x600 it obviously isn't going to fit.

Once we're ready to go back to the game, we need to exit the 2D mode and return to our full-screen 3D mode by doing the reverse of what we did in Listing 3-7:

Listing 3-6: Returning to the Full-Screen Context

```
void Exit2D(void)
{
        /* DON'T DO ANYTHING IF IN WINDOWED MODE OR NO DISPLAY YET */

    if (gPlayInWindow || (gCGDisplayID == -1))
        return;

    HideCursor();               // make cursor go away

            /* RE-CAPTURE THE DISPLAY */

    CGDisplayCapture(gCGDisplayID);


            /* RE-ENABLE GL */

    if (gAGLContext)
        aglSetFullScreen(gAGLContext, 0, 0, 0, 0);
}
```

The first important place we'll put these calls is in our game engine's `DoAlert()` and `DoFatalAlert()` functions since those can bring up 2D dialogs at any time:

Listing 3-7: Using Enter2D() and Exit2D() to Display Alerts

```
void DoFatalAlert(Str255 s)
{
    SInt16      alertItemHit;

    Enter2D();                  // make sure we can see this

    StandardAlert(kAlertNoteAlert, s, NULL, NULL, &alertItemHit);
    ExitToShell();
}


void DoAlert(Str255 s)
{
    SInt16      alertItemHit;

    Enter2D();                  // make sure we can see this

    StandardAlert(kAlertNoteAlert, s, NULL, NULL, &alertItemHit);

    Exit2D();                   // return to full-screen
}
```

Chapter 4: OpenGL Optimizations

Getting the most out of computer hardware is what game developers have strived for since the dawn of the CPU, and even though computers are blazingly fast these days, we still need to squeeze the performance as tight as we can. There are many things that you can do in OpenGL to improve performance. Some optimizations yield minor results while others will yield huge results.

Macro Optimizations

The smart folks at Apple have given us a way to make our OpenGL function calls just a little bit faster by removing one level of indirection in each call that we make. Normally, when you call an OpenGL function such as `glEnable()`, this `glEnable()` function doesn't actually do any enabling. What it's really doing is looking up the renderer's internal `enable()` function and then it jumps to that. Different renderers will have different internal enable functions, so if you're running on ATI hardware then `glEnable()` is really just looking up and calling the `enable()` function in the ATI driver.

The `AGLContex` structure that defines our draw context contains the entire jump table for every OpenGL function supported by the renderer, so there's no reason that your code can't do that function lookup and jump by itself, thus saving that extra branch to the bogus `glEnable()` function. If you look in the aglmacro.h header file you'll see every OpenGL function call defined as a macro. Each macro takes the draw context and finds the jump vector to the actual OpenGL function in question. For example, the macro for `glEnable()` looks like this:

```
#define glEnable(cap) (*agl_ctx->disp.enable)(agl_ctx->rend, cap)
```

To use the AGL macros you first need to include the aglmacro.h header file at the top of every .c file that you want optimized. Once you include this header in a file, all of the OpenGL calls you make in that file will be done with these macros instead of calling the wrapper functions.

```
#include <AGL/aglmacro.h>
```

Next, at the top of every function that makes an OpenGL call you must include this line of code that assigns your draw context to the `agl_ctx` variable that the macros rely on:

```
AGLContext agl_ctx = gAGLContext;
```

Here is a very simple example of this in action:

Listing 4-1: Example Using AGL Macros

```
#include <AGL/aglmacro.h>

void DoStuff(void)
{
    AGLContext agl_ctx = gAGLContext;

    glDisable(GL_RESCALE_NORMAL);
    glDisable(GL_DITHER);
    glCullFace(GL_BACK);
    glEnable(GL_ALPHA_TEST);
}
```

That's all there is to it! Your OpenGL function calls remain the same as before, they'll just be a little bit faster. All those `glDisable()`, `glEnable()`, etc. calls look like regular function calls, but with aglmacro.h they're really just macros that branch directly to your renderer's internal functions.

We won't be using the macros for the remaining examples in this book since I want to simplify things as much as possible in the code, but when you start building your own games I'd recommend you use this optimization.

Caching the OpenGL State

This next optimization is a bit of a hassle to implement, but it is actually very, very important. Certain video card drivers will suffer a serious drop in performance if you're constantly changing the OpenGL state. Even just calling `glColor()` can cause a serious pipeline stall, so you always want to be sure that you don't modify an OpenGL state value unless that state value has actually changed. For example, if the current color state is already 0,0,0,0 then don't call `glColor4f(0,0,0,0)` again since it isn't needed, and it may trigger a stall in the render pipeline.

So, instead of calling `glColor()` all the time, you'll want to cache the RGBA color values in your own variables, and then test any new color commands with those values to see if they've changed. If and only if they've changed should you proceed and call `glColor()`. The same goes for all `glEnable()` and `glDisable()` calls. Texture unit calls like `glActiveTexture()` and `glClientActiveTexture()` also cause an OpenGL state change, so those should also be cached. As a general rule, cache anything that causes an OpenGL state change.

The sample project "OpenGL State Caching.xcode" has an updated version of the OpenGL.c source file that includes many example state-changing functions. For example, this function is used to enable lighting:

Listing 4-2: Enabling Lighting

```
Boolean     gMyState_Lighting;

void OGL_EnableLighting(void)
{
    if (!gMyState_Lighting)
    {
        glEnable(GL_LIGHTING);
        gMyState_Lighting = true;
    }
}
```

The variable `gMyState_Lighting` is the cached lighting state. If it is already `true` then there's obviously no need to re-enable the lighting since we know that it's already enabled. We only call `glEnable()` if `gMyState_Lighting` is `false`.

The function to disable lighting is almost identical except that it only calls `glDisable()` if `gMyState_Lighting` is `true`:

Listing 4-3: Disabling Lighting

```
void OGL_DisableLighting(void)
{
    if (gMyState_Lighting)
    {
        glDisable(GL_LIGHTING);
        gMyState_Lighting = false;
    }
}
```

Other caching functions are a little more complex, such as the function to set the current color:

Listing 4-4: Setting the Color

```
OGLColorRGBA    gMyState_Color;

void OGL_SetColor4f(float r, float g, float b, float a)
{
    if ((r != gMyState_Color.r) ||
        (g != gMyState_Color.g) ||
        (b != gMyState_Color.b) ||
        (a != gMyState_Color.a))
    {
        glColor4f(r, g, b, a);

        gMyState_Color.r = r;
        gMyState_Color.g = g;
        gMyState_Color.b = b;
        gMyState_Color.a = a;
    }
}
```

Here we cache the RGBA values, and check each of them before we allow `glColor4f()` to be called. This insures that we only call the OpenGL function if one of the color components actually changed.

The important thing to remember when caching the state is that you must be consistent. You cannot choose to call `OGL_SetColor4f()` some of the time, and then other times call `glColor4f()` directly. If your cached state variables lose track of the current values then a cascade of problems will appear as your game runs.

Pushing & Popping the OpenGL State

Very often in your code you will need to preserve the current OpenGL state, modify the state to do some work, and then restore the state as you continue on your way. OpenGL has a built-in way to do this with the `glPushAttrib()` and `glPopAtrib()` functions, but you should *never ever use these functions in your code*! Yes, this is an easy way to save any state information you want, but internally this destroys your pipeline performance for the same reasons that doing any unnecessary state change is bad.

The preferred way to save and restore the current state is to manually do it yourself with the use of your state caching functions. Which state parameters you save is up to you, but the sample function below shows what I like to use in my games:

Listing 4-5: Pushing the State

```
void OGL_PushState(void)
{
    int i;

        /* PUSH MATRIES WITH OPENGL */
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

        /* SAVE OTHER INFO */

    i = gStateStackIndex++;         // get stack index and increment

    if (i >= STATE_STACK_SIZE)
        DoFatalAlert("\pstack overflow");

    gStateStack_Lighting[i]     = gMyState_Lighting;
    gStateStack_CullFace[i]     = gMyState_CullFace;
    gStateStack_DepthTest[i]    = gMyState_DepthTest;
    gStateStack_Normalize[i]    = gMyState_Normalize;
    gStateStack_Texture2D[i]    = gMyState_Texture2D;
    gStateStack_Fog[i]          = gMyState_Fog;
    gStateStack_Blend[i]        = gMyState_Blend;
    gStateStack_Color[i]        = gMyState_Color;
    gStateStack_DepthMask[i]    = gMyState_DepthMask;

    gStateStack_BlendSrc[i]     = gMyState_BlendFuncS;
    gStateStack_BlendDst[i]     = gMyState_BlendFuncD;
}
```

As I said above, using the `glPushAttrib()` function is bad, however using the OpenGL matrix push/pop calls `glPushMatrix()` and `glPopMatrix()` for preserving and restoring the states of the matrices is perfectly fine. Just be aware that there is a limit to the matrix stack's size, and if you exceed that size you will generate a `GL_STACK_OVERFLOW` error. The matrix stacks are large enough in the Mac implementation of OpenGL that I've never had a stack overflow ever occur, but if you want to be safe you can determine the stack sizes for the various matrices by doing this:

```
    GLint modelViewStackDepth;
    GLint projectionStackDepth;

    glGetIntergerv(GL_MODELVIEW_STACK_DEPTH, modelViewStackDepth);
    glGetIntergerv(GL_PROJECTION_STACK_DEPTH, modelViewStackDepth);
```

After pushing the matrices onto the matrix stack we save our current state variables into our own private stacks. Then, to restore the state we pop all the data off of these stacks like so:

Listing 4-6: Restoring the State

```
void OGL_PopState(void)
{
    int     i;

        /* RETREIVE OPENGL MATRICES */

    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);

        /* GET STATE INFO */

    i = --gStateStackIndex;             // dec stack index

    if (i < 0)
        DoFatalAlert("\pstack underflow!");

    if (gStateStack_Lighting[i])        // restore lighting
        OGL_EnableLighting();
    else
        OGL_DisableLighting();


    if (gStateStack_CullFace[i])        // restore backface culling
        OGL_EnableCullFace();
    else
        OGL_DisableCullFace();


    if (gStateStack_DepthTest[i])       // restore depth test
        OGL_EnableDepthTest();
    else
        OGL_DisableDepthTest();

    if (gStateStack_Normalize[i])       // restore normalizing
        OGL_EnableNormalize();
    else
        OGL_DisableNormalize();

    if (gStateStack_Texture2D[i])       // restore texture state
        OGL_EnableTexture2D();
    else
        OGL_DisableTexture2D();

    if (gStateStack_Blend[i])           // restore blending mode
        OGL_EnableBlend();
    else
        OGL_DisableBlend();
    if (gStateStack_Fog[i])             // restore fog state
        OGL_EnableFog();
    else
        OGL_DisableFog();

                                        // restore depth mask state
    OGL_DepthMask(gStateStack_DepthMask[i])

                                        // restore blending mode
    OGL_BlendFunc(gStateStack_BlendSrc[i],
                gStateStack_BlendDst[i]);

                                        // restore color state
    OGL_SetColor4fv(&gStateStack_Color[i]);
}
```

The Transform Hint

Apple has created an OpenGL "hint" state parameter that can improve the speed of transformation calculations, but at the cost of some accuracy:

```
    glHint(GL_TRANSFORM_HINT_APPLE, GL_FASTEST);
```

Since games don't need 100% precise mathematical calculations (we're not trying to put a man on Mars here), you should always have this hint set to `GL_FASTEST`. You'll almost never be able to see any visual difference with this, but on extremely rare occasions a vertex might be out of place by 1 pixel – a fair price to pay for a performance boost.

Normals

When you have a scaling component in your Model-View matrix (such as from a `glScale()` call), any vertex normals will get scaled during the transform calculations. This would cause your lighting to become distorted, so we always want vertex normals to be normalized to a length of 1.0. Luckily, OpenGL can re-normalize any vertex normals after a transformation, but doing so hurts performance. To enable this feature you'd do this:

```
    glEnable(GL_NORMALIZE);
```

All of your 3D model data should have vertex normals that have been pre-normalized. Therefore, if you know that you are not doing any scaling on a model when you render it then you can tell OpenGL to disable the automatic re-normalization code:

```
    glDisable(GL_NORMALIZE);
    glDisable(GL_RESCALE_NORMAL);
```

If it is necessary to scale a model, you can still get some optimization by just enabling `GL_RESCALE_NORMAL`:

```
    glEnable(GL_RESCALE_NORMAL);
```

What `GL_NORMALIZE` does is to entirely re-normalize the vector – a costly calculation, but `GL_RESCALE_NORMAL` simply resizes the normal back to a length of 1.0. This is faster than doing a full-fledged normalize on that vector, but it only works when the scaling is uniform. That means that the x. y, and z components of the scale must all be the same value. If you're doing a non-uniform scale then the only option is to let OpenGL do a full vector normalize by enabling `GL_NORMALIZE`. So, you have to be smart in your code and know when to enable and when to disable these various forms of normalizing states to get the best performance.

Colors

Always be sure to have this code in your draw context initialization:

```
    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
```

This allows you to use `glColor()` to set the state's color instead of using the slower `glMaterial()` function.

Reading Pixels

You should **never ever read data out of the draw buffer or the z-buffer**! Doing this causes the entire rendering pipeline to stall big-time, thus killing your performance. You can literally chop your game's frame rate in half by reading a single pixel from these buffers. Remember that the 3D hardware is still rendering your geometry long after you're done submitting it; therefore, if you try to read a pixel from the frame buffer, OpenGL has to sit there and wait for the scene to finish drawing before it can return the pixel value to you.

Know when to use glBegin/End or Vertex Arrays

Whenever you draw complex models you should always use Vertex Arrays to submit your geometry to OpenGL, however, if you're only drawing small things like sprites then it is still slightly more efficient to use `glBegin()`. The reason for this is that the internal overhead involved in processing Vertex Arrays is disproportionately high if you're only submitting a handful of vertices.

Listing 4-7: Drawing a Sprite with Vertex Arrays

```
void DrawSpriteWithVertexArray(void)
{
    OGLPoint3D      points[4] =
    {
        -1,-1,0,    -1,1,0,     1,1,0,  1,-1,0
    };

    OGLTextureCoord uvs[4] =
    {
        0,0,    0,1,    1,1,    1,0
    };

    GLint   quadVerts[4] = {0,1,2,4};


            /* INIT THE VERTEX ARRAYS */

    glEnableClientState(GL_VERTEX_ARRAY);           // point to points
    glVertexPointer(3, GL_FLOAT, 0, points);

    glTexCoordPointer(2, GL_FLOAT, 0,data->uvs[0]);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);    // enable uv arrays

    glDisableClientState(GL_COLOR_ARRAY);           // no vertex colors
    glDisableClientState (GL_NORMAL_ARRAY);         // no normals

    OGL_SetColor4f(1,1,1,1);


            /* SUBMIT THE VERTEX ARRAYS */

    glDrawElements(GL_QUADS, 4, GL_UNSIGNED_INT, quadVerts);
}
```

Listing 4-8: Drawing a Sprite with glBegin

```
void DrawSpriteWithBeginEnd(void)
{
    OGL_SetColor4f(1,1,1,1);

    glBegin(GL_QUADS);
    glTexCoord2f(0,1);  glVertex2f(-1, -1);
    glTexCoord2f(1,1);  glVertex2f(-1, 1);
    glTexCoord2f(1,0);  glVertex2f(1, 1);
    glTexCoord2f(0,0);  glVertex2f(1, -1);
    glEnd();
}
```

The code for drawing Vertex Arrays in listing 4-7 appears much more complex than the code using `glBegin()` in Listing 4-8, however, if you count the number of OpenGL function calls you'll see that it's actually less in the Vertex Array code. Despite this fact, using `glBegin()` is still more efficient because of the way that OpenGL deals with vertex array data. Internally, there is more overhead with Vertex Arrays.

There is another optimization for Vertex Arrays that is the mother lode of all OpenGL optimizations called Vertex Array Range. However, this optimization is a huge topic so it has it's own chapter in this book. For more information see Chapter 6.

Optimizing VRAM

A well-optimized piece of code may still run unexpectedly slow if you're trying to cram too much texture data into a video card with too little VRAM. If you're rendering a complex scene into a large frame buffer then you might not be leaving enough VRAM for all of the scene's textures to fit. This causes textures to dynamically get paged in and out of VRAM, and paging anything into VRAM is slow.

So, it's important for our game to know how much VRAM it has to work with:

Listing 4-9: Determining a Display's VRAM

```
void CalcDisplayVRAM(void)
{
    io_service_t    port;
    CFTypeRef       classCode;

              /* LOOK UP MAIN DISPLAY'S VRAM */

    port = CGDisplayIOServicePort(CGMainDisplayID());
    classCode = IORegistryEntryCreateCFProperty(port,
                                    CFSTR(kIOFBMemorySizeKey),
                                    kCFAllocatorDefault,
                                    kNilOptions);


                  /* EXTRACT VALUE */

    if (CFGetTypeID(classCode) == CFNumberGetTypeID())
    {
        CFNumberGetValue(classCode, kCFNumberSInt32Type, &gDisplayVRAM);
    }
    else
    {
```

```
                // if failed, then just assume 64MB VRAM to be safe
        gDisplayVRAM = 0x4000000;
    }
}
```

With the value `gDisplayVRAM` you can make tweaks to your game based on various VRAM quantities. For example, in Nanosaur 2 we limit the maximum screen resolution that the user can choose based on this value. If the display only has 32MB of VRAM then we limit the maximum resolution to 1024x768 to insure that there will be enough VRAM remaining to hold textures. Additionally, we avoid loading certain large textures into VRAM at all if the display is below a certain threshold, so things like sky clouds don’t get drawn in low-VRAM situations.

Chapter 5: PowerPC Math Optimizations

The PowerPC chip has some nice capabilities that can speed up certain mathematical calculations for us. If you use Shark to profile your game's performance you may learn about some fairly low level optimizations dealing with the CPU, but in this chapter I'm going to discuss two fairly major optimizations that you should use in your games:

AltiVec for Faster Matrix Multiplies

AltiVec is the vector processing unit on the PowerPC chip, and it has a lot of great uses. Unfortunately, those uses are fairly limited in the world of 3D programming. The one area where AltiVec is easy to implement yet still yields a huge performance boost is in matrix multiplication. Multiplying 4x4 matrices is notoriously slow due to the huge number of calculations and memory read/writes that are needed. AltiVec can do it much, much faster by essentially doing four calculations at a time.

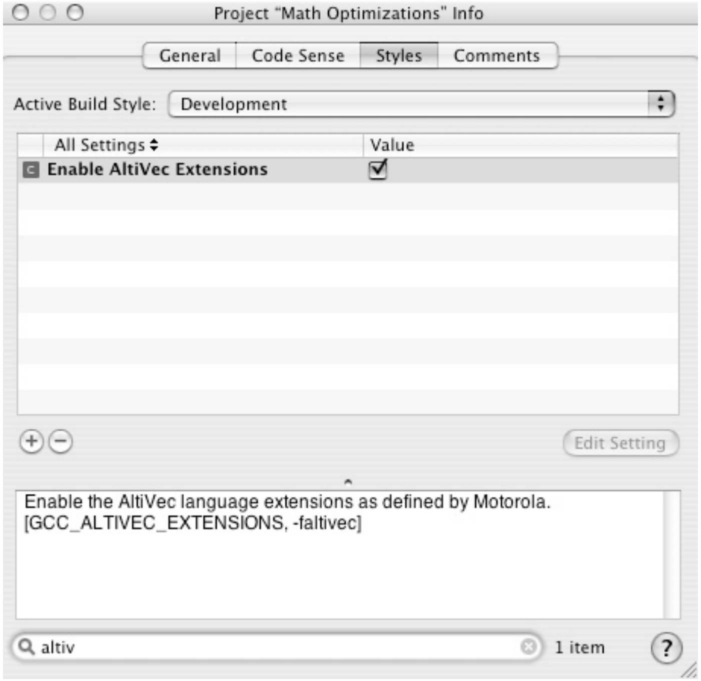
Not all PowerPC chips come with AltiVec, however. The G4 and G5's do, but the G3's and earlier do not, so we must check for the presence of AltiVec when we launch our game and then handle the matrix multiplies in either the traditional way or with AltiVec if it is present. To determine if AltiVec is available we need a function like this:

Listing 5-1: Checking for AltiVec

```
void CheckForAltiVec(void)
{
    long    response;
    u_long  flags;

    gAltiVec = false;           // assume no altivec

    if (!Gestalt(gestaltNativeCPUtype, &response))
    {
                // skip if on G3 or go into this if > G3
        if (response > gestaltCPU750)
        {
                // see if AltiVec available
            Gestalt(gestaltPowerPCProcessorFeatures,(long *)&flags);
            gAltiVec = ((flags & (1 <<
                gestaltPowerPCHasVectorInstructions)) != 0);
        }
    }
}
```

You might think that the first `Gestalt()` test isn’t needed, but it is. Testing for vector instructions on old Macintoshes tends to generate errors or incorrect results, so we first check for PowerPC 750’s which are the G3’s. Only if the CPU is newer can we test for the presence of vector instructions.

Next we need to write our Matrix Multiply function, but first be sure that you have AltiVec enabled in Xcode:



Active Build Style: Development

Enable AltiVec Extensions ☑

Enable the AltiVec language extensions as defined by Motorola.
[GCC_ALTIVEC_EXTENSIONS, -faltivec]

altiv

Figure 5-1: Enabling AltiVec in Xcode

Listing 5-2: The Matrix Multiply Function

```
void OGLMatrix4x4_Multiply(const OGLMatrix4x4   *mA,
                            const OGLMatrix4x4   *mB,
                            OGLMatrix4x4         *result)
{
    if (gAltiVec)
        OGLMatrix4x4_Multiply_AltiVec(mA, mB, result);
    else
        OGLMatrix4x4_Multiply_Float(mA,mB, result);
}
```

We must do the AltiVec and float versions of the matrix multiply in separate functions. This is very important! *Mixing AltiVec and regular floating-point code in the same function causes massive stack thrashing which kills any performance we hope to gain here*. The AltiVec version is shown below:

Listing 5-3: Doing 4x4 Matrix Multiplication with AltiVec

```
static void OGLMatrix4x4_Multiply_AltiVec(const OGLMatrix4x4    *mA,
                                        const OGLMatrix4x4      *mB,
OGLMatrix4x4    *result)
{
    vector float A1,A2,A3,A4, B1, B2, B3, B4;
    vector float zeroF = (vector float) vec_splat_u32(0);
    vector float C1, C2, C3, C4;

    /* LOAD MATRIX A */

    A1 = vec_ld(  0, (float*) mA );
    A2 = vec_ld( 16, (float*) mA );
    A3 = vec_ld( 32, (float*) mA );
    A4 = vec_ld( 48, (float*) mA );
    B1 = vec_ld(  0, (float*) mB );
    B2 = vec_ld( 16, (float*) mB );
    B3 = vec_ld( 32, (float*) mB );
    B4 = vec_ld( 48, (float*) mB );

    //Do the first scalar x vector multiply for each row
    C1 = vec_madd( vec_splat( A1, 0 ), B1, zeroF );
    C2 = vec_madd( vec_splat( A2, 0 ), B1, zeroF );
    C3 = vec_madd( vec_splat( A3, 0 ), B1, zeroF );
    C4 = vec_madd( vec_splat( A4, 0 ), B1, zeroF );

    //Accumulate in the second scalar x vector multiply for each row
    C1 = vec_madd( vec_splat( A1, 1 ), B2, C1 );
    C2 = vec_madd( vec_splat( A2, 1 ), B2, C2 );
    C3 = vec_madd( vec_splat( A3, 1 ), B2, C3 );
    C4 = vec_madd( vec_splat( A4, 1 ), B2, C4 );

    //Accumulate in the third scalar x vector multiply for each row
    C1 = vec_madd( vec_splat( A1, 2 ), B3, C1 );
    C2 = vec_madd( vec_splat( A2, 2 ), B3, C2 );
    C3 = vec_madd( vec_splat( A3, 2 ), B3, C3 );
    C4 = vec_madd( vec_splat( A4, 2 ), B3, C4 );

    //Accumulate in the fourth scalar x vector multiply for each row
    C1 = vec_madd( vec_splat( A1, 3 ), B4, C1 );
    C2 = vec_madd( vec_splat( A2, 3 ), B4, C2 );
    C3 = vec_madd( vec_splat( A3, 3 ), B4, C3 );
    C4 = vec_madd( vec_splat( A4, 3 ), B4, C4 );
    vec_st( C1,  0, (float*) result );
    vec_st( C2, 16, (float*) result );
    vec_st( C3, 32, (float*) result );
    vec_st( C4, 48, (float*) result );
}
```

Knowing how the above code actually works is not really important since it's unlikely that you'll ever use AltiVec for anything else in your game, but if you do want to learn more about AltiVec programming then Apple has lots of information here:

http://developer.apple.com/hardware/ve/

What is important, however, is knowing that the `OGLMatrix4x4` structure must be aligned to 16-bytes since all AltiVec memory accesses are 16-byte aligned. To do this we build the matrix structure as a `union` with an AltiVec `vector` type:

```
typedef union
{
    GLfloat         value[16];
    vector  float   v[4];
}OGLMatrix4x4;
```

With this structure we can still access the individual matrix elements which are 32-bit `float` values, yet the entire structure is guaranteed to always be 16-byte aligned so that we can work with it using AltiVec.

Fast Vector Normalizing

The PowerPC chip has another useful capability in the form of a special opcode called `frsqrte` – Floating point Reciprocal Square Root Estimate. This opcode calculates the value `1.0 / sqrt(n)` in about 1 cycle. To do this calculation without `frsqrte` typically costs around 50 cycles.

The drawback, however, is that `frsqrte` only calculates an *estimate* value which is not particularly precise. Luckily, there is a method for improving the accuracy of the results called Newton-Rhapson refinement, and the following example shows how this all works:

Listing 5-4: Fast Vector Normalizing with frsqrte

```
void OGL_Vector3D_NormalizeFast(float x, float y, float z,
                                OGLVector3D *outV)
{
```

```
float   len;
float   isqrt, temp1, temp2;

            /* CALC LENGTH OF INPUT VECTOR */

    len = (x * x) + (y * y) + (z * z);

        /* CALCULATE 1.0 / SQRT(len) ESTIMATE */

    isqrt = __frsqrte (len);


        /* REFINE THE ESTIMATE WITH NEWTON-RHAPSON */

    temp1 = len * -.5f;
    temp2 = isqrt * isqrt;
    temp1 *= isqrt;
    isqrt *= (float)(3.0/2.0);
    len = isqrt + temp1 * temp2;


            /* RETURN NORMALIZED VECTOR */

    outV->x = x * len;
    outV->y = y * len;
    outV->z = z * len;
}
```

Even with the extra overhead of the Newton-Rhapson refinement, this method of normalizing vectors is up to 16x faster than the more accurate method with the `sqrt()` function. I cannot stress enough, however, that you should not use this for calculations that require high accuracy. For example, when I was developing the game "Enigmo" I couldn't understand why the collision physics were behaving so strangely when water droplets would bounce off of objects. After a few hours of investigating I discovered that the anomaly was a result of using `frsqrte` to normalize the bounce vectors during the collision detection. This caused a domino effect of errors in the physics math. This optimization is best used for normalizing things like vertex normals and other things that don't need high accuracy.

To use the `frsqrte` opcode (or any other PowerPC intrinsic) like we did in Listing 5-4, you've got to include a special header file in your Xcode project:

```
    #include <ppc_intrinsics.h>
```

This wasn't necessary with CodeWarrior since the intrinsics were built into the CodeWarrior compiler, but with Xcode you need to include that header since it's actually a collection of macros that build the intrinsics as assembly code. This file is included in our sample code's precompiled header file, MyPCH.pch.

It should also be noted that the `frsqrte` opcode does not exist on the PowerPC 601, but if you're still writing games to run on a PowerPC 601 then you should seek professional therapy.

Chapter 6: Vertex Array Range

This chapter discusses the mother of all game optimizations on the Mac. If you properly implement the technique that is about to be discussed then you can get anywhere from a 50% to 300% improvement in your game's frame rate!

If you're experienced with OpenGL then you should already be familiar with Vertex Arrays since those are the preferred way to submit geometry meshes for rendering. A small example using Vertex Arrays was shown in Listing 4-7. The new optimization is called "Vertex Array Range", otherwise known as VAR, and the idea behind this optimization is very simple. By marking a Vertex Array's data as "accelerated," any rendering done with that data gets a massive speed boost. It's that simple.

Marking a range of memory as VAR data, speeds up the time it takes OpenGL to transfer that data to the video card for processing. You may not realize it, but a huge amount of time is spent during the rendering process to dump the thousands of points, normals, texture UV's, and vertex colors to the video card. You probably thought that it was the pixel drawing that took so long, but in reality it's the uploading of the raw data that's the speed killer.

The concept behind this optimization is easy, but doing a good implementation of it does take some effort, so we're going to cover the whole enchilada step by step. The sample project named "Vertex Array Range.xcode" contains all of the code in this chapter, and demonstrates how to use it to draw an object on the screen. All of the new code dealing with VAR is in the source file VertexArrayRange.c.

Initializing Vertex Array Range

The first thing we need to do to use the VAR feature is to initialize it:

Listing 6-1: Initializing Vertex Array Range Ranges

```
void OGL_InitVertexArrayRanges(void)
{
    short           i;
    static char     *s;

            /* DETERMINE IF HARDWARE SUPPORTS VAR */

    s = (char *)glGetString(GL_EXTENSIONS);     // get extensions list
    if (strstr(s, "GL_APPLE_vertex_array_range") == nil)
        DoFatalAlert("\pVertex Array Range not supported.");


        /* GENERATE VERTEX ARRAY OBJECTS */
        //
        // Each object represents a chunk of RAM that
        // we’re marking for VAR use.
        //

    glGenVertexArraysAPPLE(NUM_VERTEX_ARRAY_RANGES,
                            &gVertexArrayRangeObjects[0]);


            /* INIT OUR DATA */
            //
            // None of the VAR objects has been assigned to any data yet,
            // so here we just initialize our info.  We'll assign the VAR
            // objects to data later.
            //

    for (i = 0; i < NUM_VERTEX_ARRAY_RANGES; i++)
    {
        gVertexArrayRangeData[i].rangeSize          = 0;
        gVertexArrayRangeData[i].dataBlockPtr       = nil;
        gVertexArrayRangeData[i].forceUpdate        = true;
        gVertexArrayRangeData[i].activated          = false;
    }
}
```

Not all hardware actually supports the VAR feature, so the first thing the initialization function does is check if VAR is supported. To determine this we simply get the current OpenGL Extensions list, and then look for the string “`GL_APPLE_vertex_array_range`”. 3D hardware at least as new as the Nvidia GeForce 2MX or ATI Radeon support VAR, but older cards like the ATI Rage 128 do not.

A list of VAR objects is obtained by calling `glGenVertexArraysAPPLE()`. These object references will be used later when we need to mark blocks of memory as VAR accelerated, but for now all we are doing is initializing everything.

We’re going to need a function that we can use to assign blocks of memory to our VAR engine:

Listing 6-2: Assigning a Vertex Array Range

```
void OGL_AssignVertexArrayRangeMemory(long size, void *data,
                                    Byte whichVAR)
{
    if (whichVAR >= NUM_VERTEX_ARRAY_RANGES)
        DoFatalAlert("\pVAR is out of range!");

    gVertexArrayRangeData[whichVAR].rangeSize       = size;
    gVertexArrayRangeData[whichVAR].dataBlockPtr    = data;
    gVertexArrayRangeData[whichVAR].forceUpdate     = true;
}
```

`OGL_AssignVertexArrayRangeMemory()` is called only to store some information about the block of memory that we want accelerated, but the memory still is not marked as VAR. To actually mark our blocks of memory as VAR, we have an update function that we'll call at the beginning of the game's render loop, before anything gets drawn:

Listing 6-3: Updating the VAR Data

```
void OGL_UpdateVertexArrayRanges(void)
{
    long    size;
    Byte    i;

    for (i = 0; i < NUM_VERTEX_ARRAY_RANGES; i++)
    {
                /* SEE IF THIS VAR IS USED */

        size = gVertexArrayRangeData[i].rangeSize;
        if (size == 0)
            continue;


                /* SEE IF VAR NEEDS UPDATING */

        if (!gVertexArrayRangeData[i].forceUpdate[i])
            continue;


            /* BIND THIS VAR OBJECT SO WE CAN DO STUFF TO IT */

        glBindVertexArrayAPPLE(gVertexArrayRangeObjects[i]);


                /* SEE IF THIS IS THE FIRST TIME IN */

        if (!gVertexArrayRangeData[i].activated)
        {
            glVertexArrayRangeAPPLE(size,
                            gVertexArrayRangeData[i].dataBlockPtr);

            glVertexArrayParameteriAPPLE(
                            GL_VERTEX_ARRAY_STORAGE_HINT_APPLE,
                            GL_STORAGE_SHARED_APPLE);
            // you MUST call this flush to get the data primed

            glFlushVertexArrayRangeAPPLE(size,
                            gVertexArrayRangeData[i].dataBlockPtr);

            glEnableClientState(GL_VERTEX_ARRAY_RANGE_APPLE);

            gVertexArrayRangeData[i].activated = true;
        }

                /* ALREADY ACTIVE, SO JUST UPDATING */

        else
        {
            glFlushVertexArrayRangeAPPLE(size,
                           gVertexArrayRangeData[i].dataBlockPtr);
        }

        gVertexArrayRangeData[i].forceUpdate = false;
    }
}
```

There are lots of important things to discuss about this update function. First it determines if a VAR needs to be updated by checking a flag that we set, and if it does then it calls `glBindVertexArrayAPPLE()` to make that VAR object active:

```
    glBindVertexArrayAPPLE(gVertexArrayRangeObjects[i]);
```

If this is the first time that this VAR is being updated then we have some special initialization to do to it. It's here that we finally get around to telling OpenGL about the block of memory that we wanted accelerated:

```
    glVertexArrayRangeAPPLE(size, gVertexArrayRangeData[i].dataBlockPtr);
```

This tells OpenGL that the currently active VAR object uses this block of memory. The block of memory is defined by the pointer to the start of the memory block and the size to indicate the number of bytes in the block.

The next step is to tell OpenGL what kind of VAR this is: shared or cached. In our case, we set it to shared:

```
    glVertexArrayParameteriAPPLE(GL_VERTEX_ARRAY_STORAGE_HINT_APPLE,
                                 GL_STORAGE_SHARED_APPLE);
```

I'll discuss the differences between shared and cached VAR data in a moment, but first let's see how we complete our update function. After marking our VAR as shared we call `glFlushVertexArrayRangeAPPLE()` to get OpenGL to do it's magic:

```
    glFlushVertexArrayRangeAPPLE(size,
                                  gVertexArrayRangeData[i].dataBlockPtr);
```

This flush call is very important! It basically primes the data that's in our VAR's memory block, and if you don't do this flush you'll likely get a bunch of garbage rendered instead of your model.

After doing this we activate this VAR object with a call to `glEnableClientState()`. That's it! We're ready to start submitting geometry whose data resides in our VAR memory, and it'll get drawn super-fast.

Once we've done all that setup the first time though, we only need to make one call to update the VAR the next time around:

```
    glFlushVertexArrayRangeAPPLE(size,
                                 gVertexArrayRangeData[i].dataBlockPtr);
```

Here, the flush function simply let's OpenGL know that the data in our VAR memory has changed since the last time we rendered with it, so OpenGL can do whatever internal updating it needs to do. Once again, failure to do this flush will result in garbage being drawn, therefore, it is important to always do this update whenever data in your VAR memory block has changed, even if it's only one byte that's different.

Cached Mode vs. Shared Mode

As was stated above, there are two modes you can set a VAR object to: cached or shared. Here's what each one does:

Cached VAR Data

Memory that you mark as cached actually gets copied to VRAM, and is kept there until you dispose of it. Even though the video card processes this data the fastest, it's really not worth the problems it creates. For starters, it uses up precious VRAM that would be better used for storing texture data, not vertex data. Secondly, uploading the data to VRAM is slow, so if you're constantly allocating VAR memory in your game and you suddenly need to upload a large chunk of RAM then you'll probably notice a hiccup in the game as OpenGL uploads all that data to VRAM. Similarly, if you need to change any of the data in that memory block then OpenGL has to re-upload it each time you make a modification, hence, more hiccups in your game.

Shared VAR Data

Memory marked as shared does not get copied to VRAM. Instead it gets tagged as AGP memory. AGP memory is memory with a fast path to the video card. It's like having VRAM in regular RAM, but just a little slower. Shared VAR data is not as fast as cached VAR data, but the speed difference is completely undetectable in a real-world application. I tried both methods when developing Nanosaur 2, and there was absolutely no difference in frame rate between the two. Using shared VAR data in Nanosaur 2 sped the game up by about 70%! Since shared VAR data is not copied to VRAM it is also much easier to make modifications to the information stored there.

So, shared mode is the way to go, but if you do want to experiment with the cached mode you would need to pass `GL_STORAGE_CACHED_APPLE` to `glVertexArrayParameteriAPPLE()` instead of `GL_STORAGE_SHARED_APPLE`. Then, to do an update (which is very expensive in cached mode), you would do this:

```
glVertexArrayRangeAPPLE(0, nil);
glVertexArrayRangeAPPLE(size,
            gVertexArrayRangeData[i].dataBlockPtr);
glFlushVertexArrayRangeAPPLE(size,
            gVertexArrayRangeData[i].dataBlockPtr);
```

If you read Apple's documentation about VAR's you would be lead to believe that updating a cached VAR is exactly the same as updating a shared VAR – just use the `glFlushVertexArrayRangAPPLE()` call. However, this is not true. There was a bug in Mac OS 10.2.x which caused that flush call to basically have no effect on cached VAR memory, so, to be 100% certain that your update works we have to do a hard reset of the VAR's memory allocation. Passing 0 into `glVertexArrayRangeAPPLE()` effectively kills that VAR, and then we reset it with another call to `glVertexArrayRangeAPPLE()`. This is yet another reason to avoid using cached VAR's.

Drawing with VAR

Now it's time to put VAR to some use by accelerating the cube drawing function first used in the Math Optimizations sample project. In that sample project we defined our cube's geometry as an array of points, an array of vertex colors, and an array of triangles. When using VAR it is best to group as much data together as possible so that it can be assigned under a single block of VAR memory. To guarantee that all of our geometry data is blocked together in RAM, we're going to put it all into a single structure:

```
typedef struct
{
    OGLPoint3D      points[8];
    OGLColorRGBA    colors[8];
    GLint           quads[6];
}CubeDataType;
```

To mark the cube's Vertex Array data for VAR use we'll need to assign that block of RAM to our VAR engine:

```
    OGL_AssignVertexArrayRangeMemory(sizeof(CubeDataType),
                                     &gCubeMesh,
                                     0);
```

The final step in this whole process is to bind the VAR object, and submit the geometry. So, before making any of the usual Vertex Array calls, we need to do this:

```
    glBindVertexArrayAPPLE(gVertexArrayRangeObjects[0]);
```

So, let's see how we would go about drawing this cube:

Listing 6-4: Drawing the Cube with VAR

```
static void DrawCube(void)
{
            /* MAKE THE CUBE'S VAR MEMORY ACTIVE */
            // bind to VAR #0

    glBindVertexArrayAPPLE(gVertexArrayRangeObjects[0]);


            /* INIT THE VERTEX ARRAYS */

    glEnableClientState(GL_VERTEX_ARRAY);           // point to points
    glVertexPointer(3, GL_FLOAT, 0, gCubeMesh.points);

    glColorPointer(4, GL_FLOAT, 0, gCubeMesh.colors);
    glEnableClientState(GL_COLOR_ARRAY);            // point to vert colors

    glDisableClientState(GL_TEXTURE_COORD_ARRAY);   // no uv's
    glDisableClientState (GL_NORMAL_ARRAY);         // no normals


            /* SUBMIT THE VERTEX ARRAYS */

    glDrawElements(GL_QUADS, 4*6, GL_UNSIGNED_INT, gCubeMesh.quads);
}
```

You won't be able to notice any performance change with this simple VAR sample application because we're not drawing enough geometry to be able to tell, but the benefits become amazingly obvious once your game starts drawing lots of geometry data in the thousands or tens of thousands of vertices per frame range. Most high-end games on the Mac wouldn't be able to function without using VAR to get massive performance boosts.

Issues with VAR

Ok, now you've heard the good news, so it's time for the bad news. The bad news is that you cannot touch the memory marked as VAR while the video card is still drawing geometry from it. In other words, if we submit our cube geometry and then decide to tweak some vertex colors for the next frame, we cannot do that until the video card is done drawing the cube. If you touch VAR memory while the video card is still using it, all hell will break loose. Even changing just one byte of color data can cause the whole model to render incorrectly. You might see vertices get moved around randomly, faces drawn with incorrect vertices, etc. This usually manifests itself as random flashes in the game when different pieces of geometry data get momentarily corrupted from frame to frame.

Luckily, not all is lost. We can still modify our mesh data via three different methods: `glFinish()`, Fencing, and double-buffering.

glFinish

This is not a viable solution to the problem, but it is useful for debugging. If you need to modify some geometry data, you can always call `glFinish()` since that will wait until the video card is done drawing the scene. At that point you know it's safe to modify the data in the VAR memory. Never ever use this option for anything but debugging since it totally kills any performance gained by using VAR in the first place!

Fencing

OpenGL has a feature called "fencing" which is a way of inserting a marker in the render queue that let's you know when something is done drawing. This is like a localized version of `glFinish()` except that it lets you wait around only until a specific piece of geometry is done instead of the entire scene. In a typical application, fences won't totally kill the performance gained from using VAR's, but they do come close since they still result in a stall. However, there are times when fences are the only solution to the problem.

An OpenGL fences is another type of object just like texture objects or even VAR objects, so they have a `glGen()` call to create them:

```
glGenFencesAPPLE(1, &gMyFenceObject);
```

To use this fence object we need to insert it into the rendering queue immediately after we've submitted the geometry that we're interested in:

```
glSetFenceAPPLE(gMyFenceObject);
```

If we want to modify the geometry data that we just submitted then we make the following call to wait for it to finish rendering:

```
glFinishFenceAPPLE(gMyFenceObject);
```

As you can see, using fences is very easy and it's better than doing a `glFinish()`, but we still have one better option…

Double-Buffering

To get the maximum performance from your game engine using VAR you're going to have to double-buffer any geometry data that you make modifications to on a regular basis. Double-buffering your geometry means that you've got to have two completely separate copies of it, and each copy needs to go into a different VAR object. That way, you can submit copy "A" for rendering while you edit copy "B" for the next frame. Then, on the next frame you submit copy B while you edit copy A.

Obviously, this uses a lot of RAM. The reason that Nanosaur 2 required so much memory to run compared with our older games was simply because of the huge amount of double-buffered geometry data that that game needed. Double-buffering probably increased that game's memory footprint by over 100MB, but the speed increase we got from it was well worth it.

Most of the geometry in your scene is going to be static, meaning that you won't ever be physically modifying it. But there are lots of cases where you may be modifying data such as character animation and particle effects. As you design your game engine you will need to come up with a system for organizing your vertex arrays. My games use a very complex system that allows me to put static geometry into simple VAR's while putting dynamic geometry into double-buffered VAR's. The VAR system presented in the sample project here is a good starting point. You can build on top of that to make a more powerful system to meet your specific needs.

Chapter 7: Calculating the Frame Rate

A fundamental part of every game is calculating the frame rate. This value is needed not only for the pride factor of being able to say "my game runs at a zillion frames per second", but it is an essential value that is needed to regulate the physics in the game. Knowing the frame rate lets us adjust our motion calculations so that everything appears to be moving at the same speed even as the frame rate fluctuates.

Calculating the frame rate is as simple as calling the following function once before every frame of animation in your game loop:

Listing 7-1: Determining the Frames Per Second

```
void CalcFramesPerSecond(void)
{
    AbsoluteTime             currTime,deltaTime;
    static AbsoluteTime      time = {0,0};
    Nanoseconds              nano;

loop:
            /* GET CURRENT TIMER */

    currTime = UpTime();


        /* GET DELTA FROM PREVIOUS TIME */

    deltaTime   = SubAbsoluteFromAbsolute(currTime, time);
    nano        = AbsoluteToNanoseconds(deltaTime);
    time        = currTime;         // reset for next time interval


        /* CONVERT NANOSECONDS TO SECONDS */

    gFramesPerSecond = (float)kSecondScale / (float)nano.lo;


        /* MAKE SURE WE DON'T GO UNDER THE MINIMUM */

    if (gFramesPerSecond < MIN_FPS)
        gFramesPerSecond = MIN_FPS;


        /* MAKE SURE WE DON'T GO OVER THE MAXIMUM */

    if (gFramesPerSecond > MAX_FPS)
```

```
        goto loop;


            /* CALC 1.0 / FPS */

    gOneOverFramesPerSecond = 1.0f/gFramesPerSecond;
}
```

The function `UpTime()` returns the amount of time since your Mac has been booted. To determine how much time has passed since the last time `CalcFramesPerSecond()` was called we simply subtract the previous timer value from the current timer value. A quick call to `AbsoluteToNanoseconds()` will convert the delta value into nanoseconds so that we can easily work with it. Dividing the nanoseconds by `kSecondScale` gives us our frames-per-second value.

We also calculate this:

```
    gOneOverFramesPerSecond = 1.0 / gFramesPerSecond
```

This `gOneOverFramesPerSecond` is what you'll be using almost all the time in any game physics. The reason we calculate this value is because multiplying by a pre-calculated value `1.0/n` is always faster than dividing by `n`. For example, the value of `t` is the same in each line below, but the second line is much more efficient:

```
    t = t / n;              // this has a slow divide
    t = t * oneOverN;       // this has a fast multiply
```

You'll notice another very important thing that we do in this function:

```
    if (gFramesPerSecond < DEFAULT_FPS)
        gFramesPerSecond = DEFAULT_FPS;
```

This limits the minimum frame rate to some default value. The reason for doing this is that in real-world games things can break if the frame rate drops too low. For example, in Enigmo we had to limit the minimum frame rate to about 13fps because at frame rates lower than that the water droplets would move so far from frame to frame that the collision detection and physics response system would start to break down. In other games such as Nanosaur 2, if the frame rate drops too low then objects can move through other solid objects for the same reasons. This is a common problem in almost all games, so the easy solution is to determine what the minimum frame rate is before things start to break and then make sure you never drop below that. In reality the game will still be running at a lower frame rate, but as far as your physics are concerned you're running at that minimum `DEFAULT_FPS` value. Things on the screen will begin to look like they are moving in slow motion, but at least nothing breaks.

Similarly, we also should check for frame rates that are too high. Now you might be wondering "what's wrong with high frame rates? Isn't that what we want?" Well, yes and no. The fact of the matter is that as the frame rate increases the deltas of the game physics values will start to become very small. They can get so small that they cannot accurately be represented in a 32-bit float value. Values approaching 0.00001 will have worse and worse accuracy, and will eventually completely break down and be considered to be 0.0 by the FPU. So, you either have to use floating point `double`'s instead of single precision `float`'s, or you just have to be careful to be sure that no critical motion values get too small.

My favorite example of floating point breakdown is in the game Carmageddon. This game was written in the days of fairly slow hardware - typically in the 180mhz range. The programmers never had anything super-fast to test the game on back when they wrote it, so they had no way of knowing what would happen when faster machines did come out. Well, I found out. When I bought my G4/400mhz I noticed that Carmageddon would exhibit some very strange behavior in the physics: the motion became very erratic and other related visual errors became noticeable. Then, when I bought a G4/1000mhz the super-high frame rate in the game was causing the physics to completely malfunction. Apparently the game had lost all floating-point precision, and it rendered the game completely unplayable.

Since there really isn't any point in having a game run faster than the refresh rate of the display it's running on, I like to limit my games in the range of 85 to 120 fps. To do this limiting we essentially have to slow down our game by just sitting in a loop until it's time to go:

```
if (gFramesPerSecond > MAX_FPS)
    goto loop;
```

The sample project "Frames Per Second.xcode" uses our new frames per second calculation to display the frame rate on the screen, and also to rotate the colored cube at an even speed. In previous projects, this cube would spin at different speeds depending on the speed of your computer because we were simply rotating the cube by 0.1 degrees each frame. By using the `gOneOverFramesPerSecond` value we can adjust that rotation speed to be the same no matter what the frame rate is.

Listing 7-2: Spinning the Cube Based on the FPS

```
static void MoveCube(void)
{
            // spin 300 degrees per second on x-axis

    gCubeRotX += 300.0f * gOneOverFramesPerSecond;

            // spin 100 degrees per second on y-axis

    gCubeRotY += 100.0f * gOneOverFramesPerSecond;
}
```

As you see, we can easily tell the cube to rotate on the x-axis at a rate of 300 degrees per second, and on the y-axis at 100 degrees per second. Multiply any constant value by `gOneOverFramesPerSecond` to get its per-frame value.

Chapter 8: Gamma Fades

The easiest way to transition between scenes in a game is to fade out – fade in. This is easily accomplished with hardware gamma fades. The “Gamma” value of a display is its luminosity curve of the red, green, and blue channels from 0 to 255. Normally, a display has a linear curve like this:

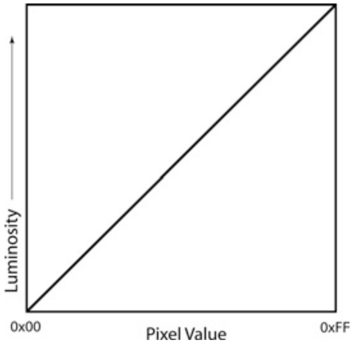


Figure 8-1: The standard gamma curve

To fade the screen 50%, we would change the display’s gamma curve like so:

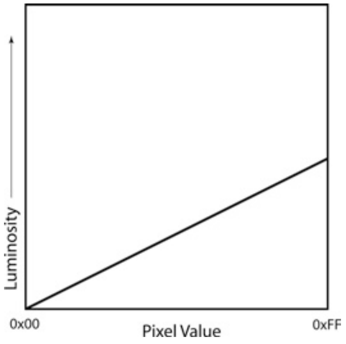


Figure 8-2: 50% faded gamma curve

Implementing gamma fades in your game code is very easy, and there are two ways to do it: You can either use the Core Graphics function `CGSetDisplayTransferByFormula()` or `CGSetDisplayTransferByTable()`.

CGSetDisplayTransferByFormula

This is the easiest way to change the gamma of your display, and the first step is to grab the display's current gamma values so we know from what values to start fading:

Listing 8-1: Get the Initial Gamma Values

```
void InitGammaValues(void)
{
    CGGetDisplayTransferByFormula(gCGDisplayID,
                &gGammaRedMin,
                &gGammaRedMax,
                &gGammaRedGamma,
                &gGammaGreenMin,
                &gGammaGreenMax,
                &gGammaGreenGamma,
                &gGammaBlueMin,
                &gGammaBlueMax,
                &gGammaBlueGamma);

    gGammaBrightness = 1.0;
}
```

All this initialization function does is read the current gamma settings with a single call to `CGGetDisplayTransferByFormula()` which returns the min, max, and gamma value of the red, green, and blue channels. The global variable `gGammaBrightness` holds the current fade value for our game, so we initialize it to 1.0. This value will range from 0.0 to 1.0 depending on how much we want our gamma faded, as you'll see below.

To fade the screen to black, all we have to do is lower the Max values of the RGB channels, and this is done by calling `CGSetDisplayTransferByFormula()`. For example, to set the gamma to 50% brightness as shown in Figure 8-2, we simply do a call like this:

```
CGSetDisplayTransferByFormula(gCGDisplayID,
                    &gGammaRedMin,
                    &gGammaRedMax * 0.5f,
                    &gGammaRedGamma,
                    &gGammaGreenMin,
                    &gGammaGreenMax * 0.5f,
                    &gGammaGreenGamma,
                    &gGammaBlueMin,
                    &gGammaBlueMax * 0.5,
                    &gGammaBlueGamma);
```

Here's a function that does a fade-out over time using our FPS calculation that we learned in the previous chapter:

Listing 8-2: Doing a Gamma Fade

```
void GammaFadeOut(float fadeDuration)
{
    while (gGammaBrightness > 0.0f)
    {
                /* SET NEW GAMMA */

        SetGammaFade(gGammaBrightness);

                /* DECAY BRIGHTNESS */

        CalcFramesPerSecond();
        gGammaBrightness -= gOneOverFramesPerSecond / fadeDuration;

    }

            /* MAKE SURE WE'RE TOTALLY FADED */

    SetGammaFade (0);
    gGammaBrightness = 0;
}
```

The `GammaFadeOut()` function is a loop that decays our global `gGammaBrightness` value over a specified duration, making use of the `gOneOverFramesPerSecond`. At the end of the function we make sure that the gamma is totally black by forcing a brightness of 0.0. The function that physically sets the gamma brightness is `SetGammaFade()`:

Listing 8-3: Setting the Gamma Fade Brightness

```
void SetGammaFade (float brightness)
{
    float   redMax      = gGammaRedMax * brightness;
    float   greenMax    = gGammaGreenMax * brightness;
    float   blueMax     = gGammaBlueMax * brightness;

    CGSetDisplayTransferByFormula(gCGDisplayID,
                    gGammaRedMin,       redMax,     gGammaRedGamma,
                    gGammaGreenMin,     greenMax,   gGammaGreenGamma,
                    gGammaBlueMin,      blueMax,    gGammaBlueGamma);
}
```

This single function is what all of our other gamma utility functions will ultimately call to make any gamma changes. If we want to do a fade-in while our game is playing, then we'll need a new fade-in function that we can call on each pass through our main loop:

Listing 8-4: Fading In an Animating Scene

```
void GammaFadeInOneFrame(float fadeDuration)
{
    if (gGammaBrightness < 1.0f)
    {
                /* DECAY BRIGHTNESS */

        gGammaBrightness += gOneOverFramesPerSecond / fadeDuration;
        if (gGammaBrightness > 1.0f)            // pin to 1.0
            gGammaBrightness = 1.0f;

                /* SET NEW GAMMA */

        SetGammaFade(gGammaBrightness);
    }
}
```

By calling `GammaFadeInOneFrame()` from our game's animation loop we will get a nice scene fade-in transition when our game starts up. However, since our application doesn't start dimmed out, we've got to turn the gamma brightness all the way down before entering the main animation loop. To instantly darken the gamma we just do this:

```
    gGammaBrightness = 0;
    SetGammaFade(gGammaBrightness);
```

The sample project "Gamma Fades.xcode" contains a full set of these gamma-fading functions in the source file Screen.c. When you run this sample application you'll see the spinning cube fade in as it's animating, and when you click the mouse button, you'll see a fade out done with our `GammaFadeOut()` function.

As you may have realized by now, you can write specialized fade functions to do color fades and not just these black fades. The `CGSetDisplayTransferByFormula()` function takes separate red, green, and blue values, so if you wanted to do a fade to red you would simply decay the green and blue channels leaving only red.

CGSetDisplayTransferByTable

The other way to modify the display's gamma is with `CGSetDisplayTransferByTable()`. This function lets you specify the entire 256-entry gamma table for the RGB channels. What this means is that you can specify an exact gamma curve if you wanted to do some really funky gamma effects. The `CGSetDisplayTransferByFormula()` function that we used earlier essentially does linear gamma curves, so you don't really have much control over it, but the Table method gives you total control. The fact of the matter is, however, that you're unlikely to ever need or have any desire to use your own custom gamma tables unless you've got some bizarre visual effect in mind.

One last thing to note about gamma fades: if your display is faded out then you won't be able to see anything in your debugger. When I know I'm going into a debug session, I always disable my gamma fading functions. For that matter, when I'm debugging I usually run the game in windowed mode, where I don't do any gamma fades at all because a gamma fade affects the whole screen, not just the window that we're rendering into.

Chapter 9: Carbon Events

Carbon Events are what keep everything on the Mac functioning. When you select a window in the Finder, that generates several types of events, and when you click on a checkbox in a dialog, that too generates a series of events. There are handler functions for just about every event that can occur, but luckily, the OS handles most of the basic ones automatically. When you click on a menu bar, the OS automatically handles all of the events involved with navigating that menu and making a selection, or when you drag a window the OS deals with all the updating for that as well.

We can install our own Carbon Event Handlers to manage many things in our game, and in this chapter we're going to learn how to use them to process our game's main loop, and also how to process its main menu. In Chapter 11 we'll see how to use events generated by the keyboard and mouse to do simple forms of input.

Event Loop Timers

Up until now our game engine's main loop has just been a simple `while()` loop, and in the days of Mac OS 9 and before, this was a perfectly acceptable way of doing things. As a matter of fact, this was the *preferred* way to do an event loop because games needed every cycle of CPU power that they could get, and any background tasks running on the Mac would just slow things down. But on OS X background tasks are going to occur whether we like it or not, so it's best to do things correctly using Carbon Events. Failure to do so can actually cause some things to behave incorrectly in your application, and would cause the OS to think that your application has locked up. Using Carbon Events also allow us to have a menu bar in our game if needed, and it lets us easily read the keyboard and extended mouse information as we'll see in Chapter 11.

Installing a Timer Callback

Timers are a type of Carbon Event that trigger a callback function at regular intervals, and we'll use one of these Timer events to call our game's main loop code as quickly as it can. It's like setting the heartbeat of your application.

Listing 9-1: Carbon Event Loop Timers

```
EventLoopTimerRef        gMainLoopTimer  = nil;

void SetMyMainLoopEventTimer(EventLoopTimerProcPtr  myMainLoopFunc)
{
    EventLoopTimerUPP       eventLoopTimerUPP;
    EventLoopRef            mainLoop;

        /* MAKE SURE WE DON'T ALREADY HAVE AN EVENT TIMER GOING */

    if (gMainLoopTimer != nil)
        DoFatalAlert("\p timer  already  set!");


        /* GET A REFERENCE TO OUR APP'S MAIN EVENT LOOP */

    mainLoop = GetMainEventLoop();

            /* CREATE A NEW EVENT LOOP TIMER */
            //
            // This inserts our main loop callback into a
            // new event timer
            //

    eventLoopTimerUPP  =  NewEventLoopTimerUPP(myMainLoopFunc);

    InstallEventLoopTimer(mainLoop,
                kEventDurationNoWait,    // delay before first shot
                kEventDurationMillisecond,  // delay until next shot
                eventLoopTimerUPP,       // which event loop timer to install
                nil,                     // no user data
                &gMainLoopTimer);        // returnedtimer reference

        /* CLEANUP */

    DisposeEventLoopTimerUPP(eventLoopTimerUPP);
}
```

Every application has a Carbon Event "main loop" by default. Don't get confused between the two different loops that we're both calling "main loop." There's the application's Carbon Event Loop which is automatically processed by the OS, and then there's our game's own main loop which is our code that moves objects and draws them. To get a reference to the application's Carbon Event main loop, we do this:

```
    mainLoop = GetMainEventLoop();
```

Next, we create a new Timer event that will trigger a callback to our game's main loop code:

```
    eventLoopTimerUPP  =  NewEventLoopTimerUPP(myMainLoopFunc);
```

To install this Timer into the Carbon Event main loop we call `InstallEventLoopTimer()` which takes several input parameters that define how the Timer will function:

`kEventDurationNoWait`
This tells the Event Manager that we want our Timer event to call our callback function immediately.

`kEventDurationMillisecond`
This tells the Event Manager to trigger the Timer event once every millisecond if possible. If the CPU is still in our main loop callback after one millisecond has expired, then there's no way that another event will get triggered, but as soon as we exit our main loop code and return control to the Event Manager, it will issue another callback to us.

Processing The Main Loop

We need to modify our old main loop code from the previous sample projects to work with our Timer Event callbacks. A typical main loop callback function looks like this:

Listing 9-2: Our Timer Event Callback Function

```
pascal void MyMainLoop (EventLoopTimerRef  theTimer, void* userData)
{
            /* EXIT EVENT LOOP IF BUTTON PRESSED */

    if (Button())
        QuitApplicationEventLoop();


                /* DO 1 PASS OF THE LOOP */

    CalcFramesPerSecond();                    // calc FPS
    GammaFadeInOneFrame(3.0);                 // do 3-second fade-in
    MoveCube();                               // move cube
    OGL_DrawScene();                          // draw scene
}
```

Once we've set all of this up we need to give control of our application to the OS. The Carbon Event Manager will have complete control, and it will issue callbacks to our main loop as our Timer Event fires away. To get the ball rolling we do this:

```
    SetMyMainLoopEventTimer(MyMainLoop);    // install the main loop timer
    RunApplicationEventLoop();              // process all the timer events
```

First we create the main loop timer callback by calling our `SetMyMainLoopEventTimer()` function. Then we enter the application's event loop via `RunApplicationEventLoop()`. As soon as `RunApplicationEventLoop()` is called our Timer event will start calling `MyMainLoop()`. When `MyMainLoop()` returns after processing one frame of animation, control returns to the OS until the Timer fires again.

Once `RunApplicationEventLoop()` is called it does not return until our callback function calls `QuitApplicationEventLoop()`, but our Timer event is still installed, so we need to remove it:

Listing 9-3: Removing the Timer Event

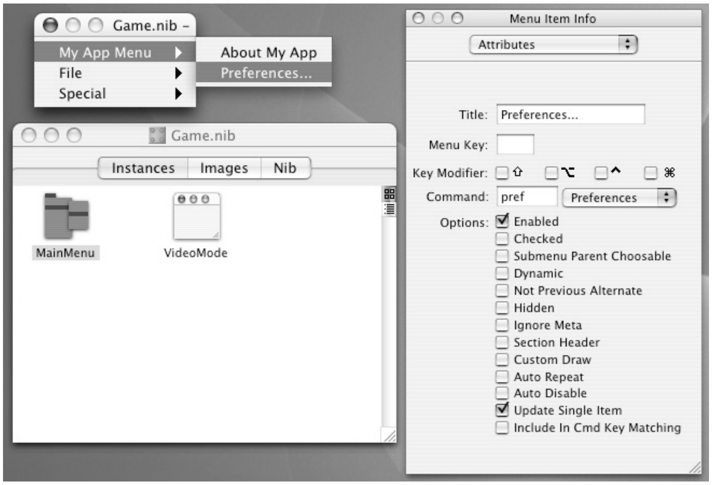
```
void RemoveMyMainLoopEventTimer(void)
{
    if (gMainLoopTimer != nil)
    {
        RemoveEventLoopTimer(gMainLoopTimer);
        gMainLoopTimer = nil;
    }
}
```

The sample project "Carbon Events.xcode" shows the spinning cube that we're familiar with, but this time the animation is running entirely off of one of these Carbon event timers.

Menu Bars

If your game were running full-screen then there obviously wouldn't be any use in having a menu bar since it wouldn't be visible. However, if your game supports a windowed mode then it's a good idea to have a menu bar, and luckily Carbon Events and Interface Builder make supporting menus very easy.

The first step in supporting menus is to build the menu bar in Interface Builder. The Game.nib file in the Carbon Events.xcode project has a new menu bar resource:



Title: Preferences...
Menu Key:
Key Modifier: ☐ ⇧ ☐ ⌥ ☐ ^ ☐ ⌘
Command: pref Preferences
Options:
☑ Enabled
☐ Checked
☐ Submenu Parent Choosable
☐ Dynamic
☐ Not Previous Alternate
☐ Hidden
☐ Ignore Meta
☐ Section Header
☐ Custom Draw
☐ Auto Repeat
☐ Auto Disable
☑ Update Single Item
☐ Include In Cmd Key Matching

Figure 9-1: Our menu resource in Interface Builder

By default, Mac OS X treats the first menu in the list as the "application menu" which means that certain things will happen to it when you see it in the program. In Figure 9-1 you can see that we named the first menu "My App Menu", and it contains just two menu items: "About My App" and "Preferences…", but that is not exactly what will appear in the game:

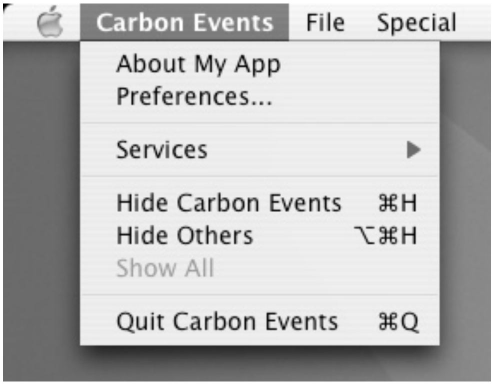


Figure 9-2: The Application Menu is automatically given our application's name

Mac OS X automatically renames the application menu to our application's actual file name regardless of what you've named it in Interface Builder. So, we see "Carbon Events" instead of "My App Menu". Additionally, the OS attaches several default menu items to this application menu. The first two menu items are the ones from our resource, but the additional ones are all default menu items added by Mac OS X, including the Quit menu item.

As you know from Chapter 2, resources built with Interface Builder all operate off of four-character command values, so we assign commands to each menu item that we want to handle. In Figure 9-1 you can see that we have assigned the command ‘pref’ to the menu item for Preferences. The command ‘quit’ is assigned by the OS to the Quit menu item that it created. These commands are handled by a new event handler that we install into our application, but first we need to load our menu bar resource and make it active. This is all done with one call:

```
    SetMenuBarFromNib(gNibs, CFSTR("MainMenu"));
```

The name of the menu resource that we want to load is passed in, and OS X takes care of the rest. The menu bar will appear at the top of the screen, and the user will be able to navigate though it and make selections. This magical, automatic handling of menu bars is one of the things that the Event Manager does for us when we’ve called `RunApplicationEventLoop()` to process our game’s main loop. In addition to handling those Timer events that we installed, it also handles menu clicks and things of that nature.

Even though we can navigate through our menu at this point, selecting a menu item has no effect because we haven’t written any code to handle menu selection events. To do that we need to install an event handler that will receive and process the menu item commands:

Listing 9-4: Installing a Menu Bar and Event Handler

```
void InitMyCommandEventHandler(void)
{
    OSStatus  iErr;

                    // types of events to handle

    EventTypeSpec   events[1] = {kEventClassCommand, kEventCommandProcess};



            /*******************************************/
            /* LOAD AND SET MENU BAR FROM OUR NIB FILE */
            /*******************************************/

    iErr = SetMenuBarFromNib(gNibs, CFSTR("MainMenu"));
    if (iErr != noErr)
        DoFatalAlert("\pSetMenuBarFromNib failed!");


            /************************/
            /* CREATE EVENT HANDLER */
        /***********************/

    gMyEventHandlerUPP = NewEventHandlerUPP(MyEventHandler);

    InstallEventHandler(GetApplicationEventTarget(),
                        gMyEventHandlerUPP,
                        1,
                        events,
                        nil,
                        &gMyEventHandlerRef);
}
```

The OS will handle most of the events that occur in the system, so all we want to do is handle the "command" events – those events that have four-character command signatures. Therefore, we define only one type of event in our `EventTypeSpec` array:

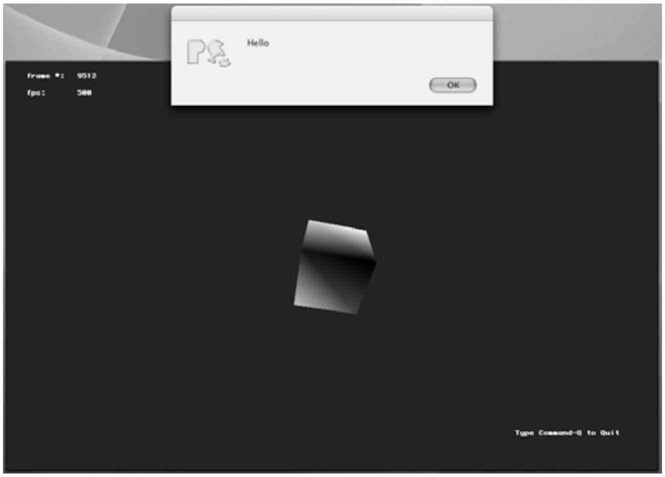
```
    EventTypeSpec   events[1] = {kEventClassCommand, kEventCommandProcess};
```

To install a callback function to handle these command events, the function `InstallEventHandler()` is called with all of the required input parameters. Once we do this, our callback function will get called anytime a menu command needs to be handled:

Listing 9-5: The Menu Command Event Handler

```
pascal OSStatus MyEventHandler(EventHandlerCallRef eventhandler,
                            EventRef event, void *userdata)
{
    OSStatus    result;
    HICommand   command;


        /* EXTRACT COMMAND INFO FROM EVENT */

    GetEventParameter(event, kEventParamDirectObject, typeHICommand,
                        nil, sizeof(HICommand), nil, &command);

        /* HANDLE THE COMMAND */

    switch (command.commandID)
    {
        case    'quit':                         // Quit menu item
                gQuitApplication = true;
                result = noErr;
                break;

        case    'pref':                         // Preferences... menu item
                break;
        case    'helo':                         // Say Hello menu item
                DoAlert("\pHello");
                break;

        default:
                result = eventNotHandledErr;
    }

    return(result);
}
```

One of the cool things about Carbon Events is evident when you select the Say Hello menu item in the Special menu. This brings up a dialog that says "Hello," and even as this dialog is displayed and you manipulate it, the cube continues to spin in the background because our Timer events keep firing.



frame #: 9512
fps: 500

Hello

Type Command-Q to Quit

Figure 9-3: Spinning cube in the background while an Alert dialog comes up

As cool as that is, you'd normally want your game to pause when any dialogs come up like that, so in your games you might make some modifications. In `MyEventHandler()`, do this:

```
        case    'helo':                         // Say Hello menu item
                gPauseGame = true;
                DoAlert("\pHello");
                gPauseGame = false;
                break;
```

Then in `MyMainLoop()` add this to the top of the function:

```
        if (gPauseGame)
            return;
```

The main loop callback will still occur, but it will simply bail out without actually doing anything, thus, giving the illusion that the game has paused.

Preventing Your Game From Going to Sleep

As far as the Mac is concerned, your game is just like any other event-driven application running on it, so unless you tell it otherwise, Mac OS will put your machine to sleep if you leave the game running with no input from the keyboard or mouse – as might happen in a self-running demo mode.

On Mac OS 9 there was a system function named `AutoSleepControl()` that you could call that would prevent the machine from going to sleep. Unfortunately, this call does absolutely nothing on OS X, so if you were thinking about using it, think again. Instead, we have a different way to prevent the Mac from going to sleep on OS X:

```
            UpdateSystemActivity(UsrActivity);
```

Calling `UpdateSystemActivity()` about once every 30 seconds should prevent the computer from going to sleep by tricking it into thinking there was some user activity. I like to insert this code into the `OGL_DrawScene()` function, but you can stick it anywhere inside the game loop that you like. The chunk of code looks like this:

Listing 9-6: Preventing the Mac from Going to Sleep

```
    static float    timer = 0;

    timer -= gOneOverFramesPerSecond;
    if (timer < 0.0f)       // see if time to update system activity
    {
        UpdateSystemActivity(UsrActivity);
        timer = 30.0f;      // reset timer to 30 seconds
    }
```

Chapter 10: Audio

Unfortunately, audio has always been one area that the Macintosh has been deficient. While PC's have had dedicated sound hardware capable of doing surround sound and all sorts of awesome processing effects for over a decade, the Mac has had just a simple built-in DAC which is entirely software driven. Over the years, attempts at doing complex sound effects in software (like with the old Sound Sprocket libraries) resulted in massive performance loss, and the audio quality was never what was achievable with the kind of hardware common in the PC world.

Luckily, the future is starting to look a little brighter now that Apple has formally adopted OpenAL as the high level sound API for OS X. This API is very efficient and very powerful, so game programmers should now be able to start supporting cool audio effects in their games. I'll be discussing OpenAL in detail in the final section of this chapter, but first I'm going to discuss some other ways of producing sound effects and music.

Quicktime for Music Playback

Quicktime is the easiest method of playing back a music file on OS X. The beauty of using Quicktime is that it magically handles just about any standard audio file format around, from MP3 to AIFF to WAV, and even the Dolby AAC format. Additionally, it lets you stream music from a large file without having to load it into RAM (although I recommend pre-loading it into RAM for performance). The only major downside to using Quicktime is that it is lousy at looping a music file. Quicktime likes to pause for a fraction of a second when it "rewinds" a song to loop it, so you cannot have music that seamlessly repeats if you're using Quicktime. I always use Quicktime to play the music in my games, so I've had to tell my musicians to make sure that all the songs have some sort of completion to them that end in silence. That way there will be no noticeable paused during the rewind. In my experience, this rewind delay can be anywhere from 1/20th of a second to a full 1/4 second.

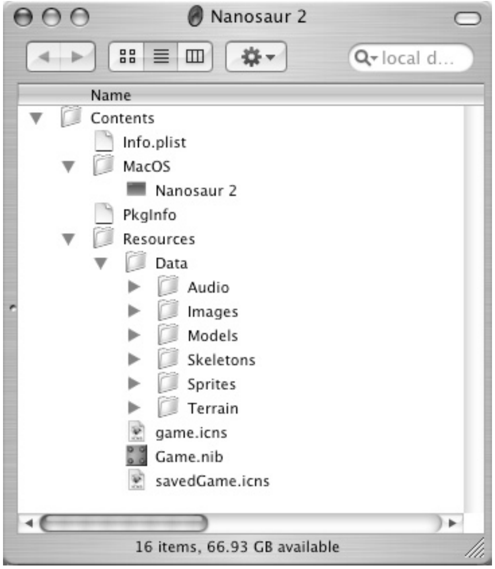
To get started with Quicktime, we need to make one initialization call in our application's startup code:

```
EnterMovies();
```

Before we discuss how to actually load a sound file into Quicktime for playback, we first need to have a short discussion on Files with Mac OS X.

Accessing Data Files in the Resources Folder

All of your data files for your game should reside in your application package's Resources folder. This is where Xcode will copy your data/resource files that are included in the Xcode project:



- Contents
  - Info.plist
  - MacOS
    - Nanosaur 2
  - PkgInfo
  - Resources
    - Data
      - Audio
      - Images
      - Models
      - Skeletons
      - Sprites
      - Terrain
    - game.icns
    - Game.nib
    - savedGame.icns

16 items, 66.93 GB available

Figure 10-1: The Resources folder in the app bundle is where the Data resides

There are at least three ways to access files on the Mac:

- stdio file calls such as `fopen`, `fread`, etc.
- `FSSpec`'s which are the pre-OS X way of getting to files.
- `FSRef`'s which are the OS X preferred way to access files.

Nobody in his or her right mind would actually use stdio in a real Mac application, so we won't even discuss that here. That leaves us with the option of using `FSSpec`'s or `FSRef`'s. Well, the fact of the matter is that we'll want to use both. `FSRef`'s are the most modern way of accessing files since they support Unicode and long filenames, but `FSSpec`'s are still used by much of the OS including calls to Quicktime.

So, to find our application's Resources folder, we call this function:

Listing 10-1:Finding the Application’s Resources Folder

```
FSRef       gMyResourcesFolderFSRef;
FSSpec      gMyResourcesFolderFSSpec;

void GetMyApplicationResourcesFolder(void)
{
    CFBundleRef     appBundle;
    CFURLRef        appResourcesURL;

            /* GET REFERENCES TO APPLICATION’S BUNDLE */

    appBundle = CFBundleGetMainBundle();


            /* GET THE URL OF THE RESOURCES FOLDER */

    appResourcesURL = CFBundleCopyResourcesDirectoryURL(appBundle);
    if (appResourcesURL == nil)
        DoFatalAlert(“\pYou don’t seem to have a Resources folder”);


            /* CONVERT THE URL TO AN FSREF */

    CFURLGetFSRef(appResourcesURL, &gMyResourcesFolderFSRef);


        /* ALSO GET THE FSSPEC WHILE WE’RE HERE */

    FSGetCatalogInfo(&gMyResourcesFolderFSRef,
                    kFSCatInfoNone,             // no catinfo needed
                    nil,
                    nil,
                    &gMyResourcesFolderFSSpec,  // fsspec to save into
                    nil);

        /* A TRICK TO GET THE VOLUME & DIRECTORY ID’S INTO THE FSSPEC */

    iErr = FSMakeFSSpec(gMyResourcesFolderFSSpec.vRefNum,
                        gMyResourcesFolderFSSpec.parID,
                        "\p:Resources:Game.nib",
                        &gMyResourcesFolderFSSpec);

    if (iErr != noErr)
        DoFatalAlert("\perror converting FSRef to FSSpec");

    gMyResourcesFolderFSSpec.name[0] = 0;       // clear name string
}
```

The first part of Listing 10-1 is fairly straightforward. We get the application’s main bundle with a call to `CFBundleGetMainBundle()`, and then we pass that bundle reference to

`CFBundleCopyResourcesDirectoryURL()` which returns a URL to the Resources folder. But we don't' want to work with URL's so the URL is converted into an `FSRef` by `CFURLGetFSRef()`.

This next part is a bit of a trick. When we call `FSGetCatalogInfo()` we get an `FSSpec` back, but it's not quite the `FSSpec` that we want. The `vRefNum` and `parID` (volume and directory ID) fields are not set to the Resources folder. Instead, they are set to the Resources folder's *parent* folder, and the `name` field of the `FSSpec` is "Resources". What we want is for the `vRefNum` and `parID` fields to point directly to the Resources folder. Here's how we can do that:

```
FSMakeFSSpec(gMyResourcesFolderFSSpec.vRefNum,
              gMyResourcesFolderFSSpec.parID,
              "\p:Resources:Game.nib",
              &gMyResourcesFolderFSSpec);
```

When we call `FSMakeFSSpec()` to create a new `FSSpec`, we pass the volume and directory ID's of that parent folder that we found, along with a pathname to a file that we know exists in our Resources folder (in this case Game.nib). Upon return, `gMyResourcesfolderFSSpec` contains the volume and folder ID's of the Resources folder. Mission accomplished!

Starting a Sound File with Quicktime

Now that we know where our game's Resources folder is, we're ready to locate a sound file and give it to Quicktime for playing. The sample project titled "Quicktime Music.xcode" demonstrates how to play a streamed file called "Song.m4a". The `PlaySong()` function looks like this:

Listing 10-2:Playing a Song with Quicktime

```
CGrafPtr    gQTDummyPort = nil;
Movie       gSongMovie = nil;

void PlaySong(Str255    songFileName, Boolean loopFlag)
{
    OSErr       iErr;
    FSSpec      spec;
    short       myRefNum;
    GrafPtr     oldPort;
    TimeValue   timeNow, duration;
    Fixed       playRate;
    Media       theMedia;
```

```
/* CREATE FSSPEC FOR THE SOUND FILE */

iErr = FSMakeFSSpec(gMyResourcesFolderFSSpec.vRefNum,
                    gMyResourcesFolderFSSpec.parID,
                    songFileName,
                    &spec);
if (iErr)
    DoFatalAlert("\pSong file not found");


    /* GOT TO SET A DUMMY PORT FOR QUICKTIME */
    //
    // Even though we’re only playing audio, we must set a
    // dummy port for Quicktime to “render” into.
    //

if (gQTDummyPort == nil)                // create a blank graf port
    gQTDummyPort = CreateNewPort();

GetPort(&oldPort);                      // backup current port
SetPort(gQTDummyPort);                  // set dummy as active port


        /* OPEN THE SOUND FILE AS A QT MOVIE */

iErr = OpenMovieFile(&spec, &myRefNum, fsRdPerm);
if ((myRefNum == 0) || (iErr != noErr))
    DoFatalAlert(“\pError opening movie file”);


    /* CREATE A NEW QT MOVIE FROM THE OPEN FILE */

NewMovieFromFile(&gSongMovie, myRefNum, 0, nil,
                newMovieActive, nil);

CloseMovieFile(myRefNum);


        /* SET MOVIE PLAY HINTS */

SetMoviePlayHints(gSongMovie,
                0,                      // turn these hints off
                hintsUseSoundInterp|hintsHighQuality);

if (loopFlag)                           // turn loop hint on
    SetMoviePlayHints(gSongMovie, hintsLoop, hintsLoop);

        /* PRE-LOAD MOVIE INTO RAM */

timeNow = GetMovieTime(gSongMovie, nil);
duration = GetMovieDuration(gSongMovie);
playRate = GetMoviePreferredRate(gSongMovie);
```

```
    theMedia = GetTrackMedia(GetMovieIndTrack(gSongMovie, 1));

    LoadMovieIntoRam(gSongMovie, timeNow, duration, keepInRam);
    LoadMediaIntoRam(theMedia, timeNow, duration, keepInRam);

    PrePrerollMovie(gSongMovie, timeNow, playRate, nil, nil);
    PrerollMovie(gSongMovie, timeNow, playRate);


            /* SET LOOPING CALLBACK */

    if (loopFlag)
    {
                                        // get timebase of movie
        TimeBase timebase = GetMovieTimeBase(gSongMovie);

                                        // create a new callback ref
        gMovieCallback = NewCallBack(timebase, callBackAtExtremes);

                                        // create callback UPP
        gMovieCallbackUPP = NewQTCallBackUPP(EndOfSongCallback);

        CallMeWhen(gMovieCallback,          // callback ref
                    gMovieCallbackUPP,      // function to callback
                    0,
                    triggerAtStop,          // call when movie stops
                    0,0);
    }


                /* START THE MUSIC PLAYING */

    StartMovie(gSongMovie);

    gSongPlayingFlag = true;
    gLoopSongFlag = loopFlag;

                /* RESTORE THE PORT */

    SetPort(oldPort);
}
```

The first thing that `PlaySong()` does is call `FSMakeFSSpec()` to create a `FSSpec` for the audio file we want to play:

```
iErr = FSMakeFSSpec(gMyResourcesFolderFSSpec.vRefNum,
                    gMyResourcesFolderFSSpec.parID,
                    songFileName,
                    &spec);
```

Here we pass in the volume and directory ID's of our Resources folder that we found earlier along with the filename of the audio file. The result is a new `FSSpec` that we will soon pass to Quicktime.

Even though we are playing an audio file, Quicktime treats everything as a Quicktime "Movie", and since Quicktime movies play into `GrafPorts`, we need to create an empty dummy `GrafPtr` to use. This code will create a new `GrafPtr`, and then it sets it as the current port after backing up the existing port.

```
if (gQTDummyPort == nil)                // create a blank graf port
    gQTDummyPort = CreateNewPort();

GetPort(&oldPort);                      // backup current port
SetPort(gQTDummyPort);                  // set dummy as active port
```

The next step is to open our sound file:

```
OpenMovieFile(&spec, &myRefNum, fsRdPerm);
```

Then we create a new Movie from the file, and then close the file since it's not needed anymore:

```
NewMovieFromFile(&gSongMovie, myRefNum, 0,  nil,
                 newMovieActive, nil);

CloseMovieFile(myRefNum);
```

This doesn't seem like the logical thing to do, does it? Why would we want to close the movie file before we've even played it? Well, the `NewMovieFromFile()` call actually creates a new internal reference to that open file, so when we close it there's still an open reference to the file in the movie object. In other words, the file isn't really completely closed.

Technically, we're ready to play the movie now, but there are several things we can do to try and improve playback performance. Quicktime will normally stream data from the file as it plays it back, but if we pre-load the entire sound file into memory, then the playback will be faster. There are also some hints we can give Quicktime about how to play the movie which may also improve performance, and we'll do those first:

```
    SetMoviePlayHints(gSongMovie, 0,
                    hintsUseSoundInterp | hintsHighQuality);

    if (loopFlag)
        SetMoviePlayHints(gSongMovie, hintsLoop, hintsLoop);
```

The function `SetMoviePlayHints()` can be used to turn various flags on and off. In our code we're turning off sound interpolation and turning off high quality mode. In theory this will improve performance of the sound playback at the cost of some audio quality, but in practice I've never noticed it to make any perceptible difference at all either in sound quality or in performance. Nonetheless, just to be safe I always clear those flags.

Similarly, we use `SetMoviePlayHints()` to then set the `hintsLoop` flag which is supposed to be a hint to Quicktime that this movie is going to loop. Unfortunately, I've never found this to make any difference since with or without this flag there is always a short delay between loops of the movie as Quicktime rewinds it. Regardless, I always set this flag in the off chance that Apple some day makes it work correctly.

We want to pre-load the entire movie into RAM, and to do this we first gather some information about the Quicktime movie that we're playing:

```
    timeNow  = GetMovieTime(gSongMovie, nil);
    duration = GetMovieDuration(gSongMovie);
    playRate = GetMoviePreferredRate(gSongMovie);
    theMedia = GetTrackMedia(GetMovieIndTrack(gSongMovie, 1));
```

This information is used in the following calls to do the pre-load and the pre-roll. Pre-loading the movie places all of the data into RAM while pre-rolling the movie then causes Quicktime to pre-initialize everything it needs to play the movie.

```
    LoadMovieIntoRam(gSongMovie, timeNow, duration, keepInRam);
    LoadMediaIntoRam(theMedia, timeNow, duration, keepInRam);

    PrePrerollMovie(gSongMovie, timeNow, playRate, nil, nil);
    PrerollMovie(gSongMovie, timeNow, playRate);
```

There are two parts of a Quicktime movie: the Movie and the Media. We pre-load both into RAM with the `LoadMovieIntoRam()` and `LoadMediaIntoRam()` calls. You can think of the Movie as the header data, and the Media as the actual sound wave information.

Two different pre-roll functions are called, `PrePrerollMovie()` and `PrerollMovie()`. Calling both of these makes sure that everything that could be initialized *is* initialized.

One last thing that we need to do is to set up a callback function to handle looping. Even though we set the `hintsLoop` flag earlier, this does not actually cause a movie to loop when it reaches the end. Unfortunately, all looping in Quicktime needs to be handled manually, but at least we can ask Quicktime to issue a callback when the movie reaches the end. Four function calls are needed to create this callback and install it into the movie that we're about to start playing:

```
timebase             = GetMovieTimeBase(gSongMovie);
gMovieCallback       = NewCallBack(timebase, callBackAtExtremes);
gMovieCallbackUPP    = NewQTCallBackUPP(EndOfSongCallback);
CallMeWhen(gMovieCallback,
            gMovieCallbackUPP,
            0,
            triggerAtStop,
            0,0);
```

`GetMovieTimeBase()` extracts some timing and identity information out of the movie, and then we pass that to `NewCallBack()` along with the constant `callBackAtExtremes` to define what kind of callback we're creating. Our callback function is defined with `NewQTCallBackUPP()` and then `CallMeWhen()` is used to tell Quicktime to issue the callback when the movie stops. We'll go into the details of this callback in a few pages.

Finally, we're ready to play the music, so we make one last call to Quicktime to get it going:

```
StartMovie(gSongMovie);
```

Updating Quicktime with MoviesTask() Inside a Thread

Unfortunately, Quicktime requires some hand holding to keep it going. For reasons that don't make much sense, you have to continuously call the Quicktime function `MoviesTask()`, otherwise your movie will stop playing. This seems to go against the general philosophy of Mac OS X and the way things work, but it is what it is, and you must call `MoviesTask()` quite often, like 10 times per second!

Calling `MoviesTask()` at such a high rate can be very difficult sometimes, especially if you want your music to keep playing during level loading when you're not in any kind of loop in which you could continuously call it. I used to just insert `MoviesTask()` calls all over the place in my code, so that it would constantly get called during file loading and initialization functions. However, this began to break down when we were working on Nanosaur 2.

Nanosaur 2 used some huge textures, and lots of them. As the textures were read in from the data files, they would get pre-loaded into VRAM by a `glBindTexture()` call, but as VRAM started to fill up it would take OpenGL longer and longer to complete this upload. On machines with very low VRAM it would sometimes take as long as 3 seconds to complete a `glBindTexture()` call. Unfortunately, during this time there was no way to call `MoviesTask()`, so the music would stutter horribly.

After working with Apple for a solid 24 hours on this problem, we came to the only solution which seemed to work: we'd have to create a separate code thread who's sole purpose in life was to call `MoviesTask()` at the constant rate of 10 times per second. Threads are part of the Mac's multi-tasking capabilities that allow different pieces of code run in different threads. To update `MoviesTask()` we are going to create a "pthread" since they're easy to set up and easy to use.

Somewhere in your game's `InitAudio()` function you'll need to put this line of code:

```
    err = pthread_create(&myThread, nil, MySongThread, nil);
```

That simple call to `pthread_create()` is all that is needed to start a new processor thread to our function `MySongThread()`, but things do get a little trickier inside that thread's function:

Listing 10-3:The Song Thread

```
void *MySongThread(void *in)
{
    AbsoluteTime    expTime;

    while(true)                     // loop inside here forever
    {
            /* CALCULATE THE TIME OF NEXT LOOP */
            //
            // Calculate current time + 100 milliseconds.
            // That's when we'll want to loop thru again.
            //

        expTime = AddDurationToAbsolute(100 * kDurationMillisecond,
                                        UpTime());

        /* UPDATE MOVIES TASK */
        //
        // We must be very careful not to call MoviesTask()
        // while some other Quicktime call is in progress!
        //

        if (!gInQuicktimeFunction)
        {
            if (gSongMovie)
            {
                gInMoviesTask = true;
                MoviesTask(gSongMovie, 0);
                gInMoviesTask = false;
            }
        }

            /* WAIT UNTIL IT'S TIME TO LOOP AGAIN */

        MPDelayUntil(&expTime);
    }

    return 0;
}
```

Unlike Timer events that we discussed in Chapter 9, threads are not continuously called by some thread manager somewhere. Instead, this Thread function is called just once, so we must stay inside of it for as long as we want the thread running. The OS will allocate CPU time to each thread running, so just think of the thread as it's own little application running all by itself.

We don't want our thread to use up any more CPU time than is necessary, therefore, rather than just sitting in a tight loop constantly calling `MoviesTask()` we will call it once and then tell the OS to go work on other treads until we need to call it again. `MoviesTask()` only needs to be called about 10 times per second, so to regulate that we essentially set an alarm clock that will wake us up when it's time to call `MoviesTask()` again. To set the alarm's wakeup time, we get the current system clock time with `UpTime()` and then add the equivalent of 1/10th of a second to it:

```
    expTime = AddDurationToAbsolute(100 * kDurationMillisecond, UpTime());
```

The value `expTime` now contains the system clock time when we'll want our alarm to wake us up. `MoviesTask()` is called, and then we turn on our alarm clock by calling `MPDelayUntil()`:

```
    MPDelayUntil(&expTime);
```

This tells the CPU to go off and do other tasks until the system clock reaches our calculated "alarm" time. When that time is reached, `MPDelayUntil()` will exit, and our thread will once again have control and the whole process repeats. If we did not include this delay in our thread loop, the OS would assume that this thread needs a huge amount of CPU time, so it would try to give it all it could. As a result, everything else on the computer would slow down, so calling `MPDelayUntil()` is critically important.

Threads are pieces of code that are running in parallel with other parts of your program, so if we were to call `MoviesTask()` from our thread while the main application thread was making some other Quicktime call, it could result in a crash. We get around this problem by setting flags to indicate that Quicktime is busy, therefore, we shouldn't try to make any other Quicktime calls until it's safe.

In `MySongThread()`, we always check the flag `gInQuicktimeFunction` to be sure that it's ok to call `MoviesTask()`. Similarly, when we call `MoviesTask()` we set another flag, `gInMoviesTask` to let the rest of the application know about it. For example, if we were going to change the volume of our song we would have some code like this:

Listing 10-4:Changing the Song Volume

```
static volatile  Boolean gInQuicktimeFunction = false;
static volatile  Boolean gInMoviesTask = false;


    gInQuicktimeFunction = true;    // dont allow song thread to
                                    // call MoviesTask().

    while(gInMoviesTask);           // wait until thread is done with
                                    // MoviesTask().

    SetMovieVolume(gSongMovie, volume); // change volume

    gInQuicktimeFunction = false;   // all clear
```

First we set the `gInQuicktimeFunction` flag to `true` so that our thread won't call `MoviesTask()` while we're doing this. Then we check if `MoviesTask()` was already being called. If so, then we just sit in a `while` loop until it's done.

Note that these flags must be declared as `volatile` variables. This is critical! These variables are being modified off of an interrupt, so they can change at any moment. By making them `volatile`, the compiler will generate code that insures that the values are re-loaded from memory every time they're referenced.

Looping a Song

Earlier in this chapter we discussed setting up a callback function to handle the looping of our Quicktime movie to cause the song to repeat, so now it's time to talk details. `MoviesTask()` will cause our callback function `EndOfSongCallback()` to be called when it detects that the song has ended. It is important to remember that we're only calling `MoviesTask()` from `MySongThread()`, and not all Mac OS calls are thread-safe. For this reason, we want to limit what we do inside of our callback:

Listing 10-5:The Loop Song Callback

```
pascal void EndOfSongCallback(QTCallBack callback, long refCon)
{
    GoToBeginningOfMovie(gSongMovie);   // rewind song

    CallMeWhen(gMovieCallback,          // reinstate the callback
                gMovieCallbackUPP,
                0,
                triggerAtStop,
                0,0);

    StartMovie(gSongMovie);             // start playing again
}
```

`GoToBeginningOfMovie()` will rewind the song back to the beginning, and then `StartMovie()` will cause it to continue playing. In between those calls we call `CallMeWhen()` to reinstate our callback. Once the callback has occurred, it won't get triggered again unless we reinstate it each time.

Note that we do not need to do any of our safety checks with the `gInQuicktimeFunction` or `gInMoviesTask` flags because we already did this safety check before we called `EnterMovies()`, so we're still safe in our callback which was called from inside `EnterMovies()`.

The nice thing about this method of looping our game music is that it happens automatically since it's running on that dedicated thread. However, our thread is only updating `MoviesTask()` about 10 times per second which means that there could be as much as a 10th of a second from the time a song ends to the time we become aware of that and try to rewind it. This is not the sole cause of the delay that Quicktime experiences when looping audio, but it certainly contributes to it.

Your game will be running at a rate higher than 10 frames per second (hopefully), so you can choose to manually check the movie status in the game's main loop with a test like this:

Listing 10-6:Optional Song Completion Test

```
            /* DO SAFETY CHECK */

    gInQuicktimeFunction = true;
    while(gInMoviesTask);

    /* IF MOVIE IS DONE THEN REWIND */

    if (IsMovieDone(gSongMovie))
    {
        GoToBeginningOfMovie(gSongMovie);
        StartMovie(gSongMovie);
    }

    gInQuicktimeFunction = false;
```

This may trim a fraction of a second off of the delay between loops in your song, but there's still going to be a delay. For this reason, I wouldn't worry about doing this extra test.

The Quicktime Music.xcode sample project demonstrates all of this code, and it includes several more utility functions for working with Quicktime. See the new source file Audio.c.

The Sound Manager for Playing Effects

Since the beginnings of Mac OS the Sound Manager has existed as a way to easily play sound effects. Unfortunately, with OS X, some parts of the Sound Manager got phased out as Apple tried to encourage developers to use Core Audio instead. Core Audio is a totally new sound system for OS X, but it is very, very low level. So low level that it was never really a practical way to do audio in games, because rudimentary things like increasing the pitch of an effect was virtually impossible to implement. So, even though the Sound Manager lost some functionality on OS X, game developers continued to use it as the only practical way of generating sound effects in games. Recently, however, Apple announced support for OpenAL which is an alternative to the Sound Manager, and I'll cover that topic in the last part of this chapter, but the Sound Manager remains the only standard, built-in way to do sound effects on any version of OS X, and it's what every game I've ever written has used.

Sound Channels

The Sound Manager works on the concept of Sound Channels. A channel plays a single digital audio sample, and you can set the pitch, volume, and other parameters of the channel's playback. When we initialize our game, we need to initialize all of the sound channels that we think we'll need. If you think your game will need to play up to 30 simultaneous sound effects, then initialize 30 sound channels. Allocating and initializing a sound channel would normally be a very simple thing to do, but since we're writing a game and we need maximum performance, things get a little more complicated.

Listing 10-7:Initializing Sound Manager Channels

```
void InitSoundManagerChannels(void)
{
    OSErr           iErr;
```

```
short           i;
ExtSoundHeader  sndHdr;
const double    rate = rate44khz;
SndCommand      mySndCmd;
SndChannelPtr   channel;


            /***************************/
            /* MAKE DUMMY SOUND HEADER */
            /***************************/
            //
            // This mimics the header of a sound file with no data.
            // it is needed below to issue sound commands to optimize
            // the channels.
            //

sndHdr.samplePtr            = nil;
sndHdr.sampleRate           = rate44khz;
sndHdr.loopStart            = 0;
sndHdr.loopEnd              = 0;
sndHdr.encode               = extSH;
sndHdr.baseFrequency        = 0;
sndHdr.numFrames            = 0;
sndHdr.numChannels          = 2;
dtox80(&rate, &sndHdr.AIFFSampleRate);
sndHdr.markerChunk          = 0;
sndHdr.instrumentChunks     = 0;
sndHdr.AESRecording         = 0;
sndHdr.sampleSize           = 16;
sndHdr.futureUse1           = 0;
sndHdr.futureUse2           = 0;
sndHdr.futureUse3           = 0;
sndHdr.futureUse4           = 0;
sndHdr.sampleArea[0]        = 0;


        /*********************/
        /* ALLOCATE CHANNELS */
        /*********************/

for (i = 0; i < MAX_CHANNELS; i++)
{
        /* NEW SOUND CHANNEL */

    iErr = SndNewChannel(&channel, sampledSynth,
                    initStereo + initNoInterp,
                    NewSndCallBackUPP(LookSoundCallback));
    if (iErr)
        break;


        /* ISSUE SOME COMMANDS TO OPTIMIZE THE CHANNEL */
```

```
            //
            // The initNoInterp command used above is ignored by
            // the Sound Manager.  To actually cause initNoInterp
            // to work, we must do it this way:
            //

        mySndCmd.cmd        = soundCmd; // install the bogus sound
        mySndCmd.param1     = 0;
        mySndCmd.param2     = (long)&sndHdr;
        SndDoImmediate(channel, &mySndCmd);

        mySndCmd.cmd        = reInitCmd;    // re-init  the channel
        mySndCmd.param1     = 0;
        mySndCmd.param2     = initNoInterp | initStereo;
        SndDoImmediate(channel, &mySndCmd;


            /* INIT SOME OF OUR CHANNEL INFO */

        gChannelInfo[i].channel = channel;      // save channel into list
        gChannelInfo[i].isLooping = false;      // clear looping flag

    }

            /* REMEMBER HOW MANY CHANNELS WE ALLOCATED SUCCESSFULLY */

    gNumChannels = i;
}
```

The first thing this function does is to define a dummy sound header. Sound effects will have headers on them that contain all the important information about the sound such as size, frequency, loop points, etc. We will use this dummy sound header for the sole purpose of issuing some sound commands to initialize certain Sound Channel parameters, but first we create a new sound channel:

```
    SndNewChannel(&channel, sampledSynth, initStereo+initNoInterp,
                    NewSndCallBackUPP(LookSoundCallback));
```

The constant `sampledSynth` tells the Sound Manager that this sound channel plays sampled digital data (as opposed to algorithmically generated sounds). Notice that we pass in the `initStereo+initNoInterp` init flags, but due to a bug in the Sound Manager these flags are usually ignored.

`SndNewChannel()` also takes a pointer to a callback function that is used to loop sound effects. Looping sound effects with the Sound Manager is very similar to looping music with Quicktime as we discussed earlier in this chapter. Luckily, however, with the Sound Manager the audio will loop seamlessly instead of having a short pause between loops. In older versions of the Sound Manager, manual looping was not necessary. The Sound Manager used to recognize the loop data embedded in the sound header, therefore, it would automatically loop any effects that needed it. But something broke in OS 9 that resulted in the need for these callbacks to manually handle looping effects.

After we've allocated our new sound channel, we need to re-initialize it since the init flags we passed to `SndNewChannel()` were ignored. To do this we attach our dummy sound header to the sound channel, and then issue a `reInitCmd` command to set the flags we want. The dummy header is attached to the channel like this:

```
mySndCmd.cmd         = soundCmd;
mySndCmd.param1      = 0;
mySndCmd.param2      = (long)&sndHdr;
SndDoImmediate(channel, &mySndCmd);
```

Then the re-initialize command is issued and we pass in those initialization flags:

```
mySndCmd.cmd         = reInitCmd;
mySndCmd.param1      = 0;
mySndCmd.param2      = initNoInterp | initStereo;
SndDoImmediate(channel, &mySndCmd;
```

Sound channels work by issuing commands to them. There are commands to set the playback frequency, change the volume, stop the channel, start the channel, etc. But for now we just need to issue two commands:

`soundCmd`
This command attaches a sound to the channel. In our case above, we're attaching a dummy sound defined by our dummy header.

`reInitCmd`
This command reinitializes the sound channel with the flags in `param2`. There must be a sound attached to the channel before this command will work. That's why we issue the `soundCmd` first.

Listing 10-8:The Sound Channel Callback

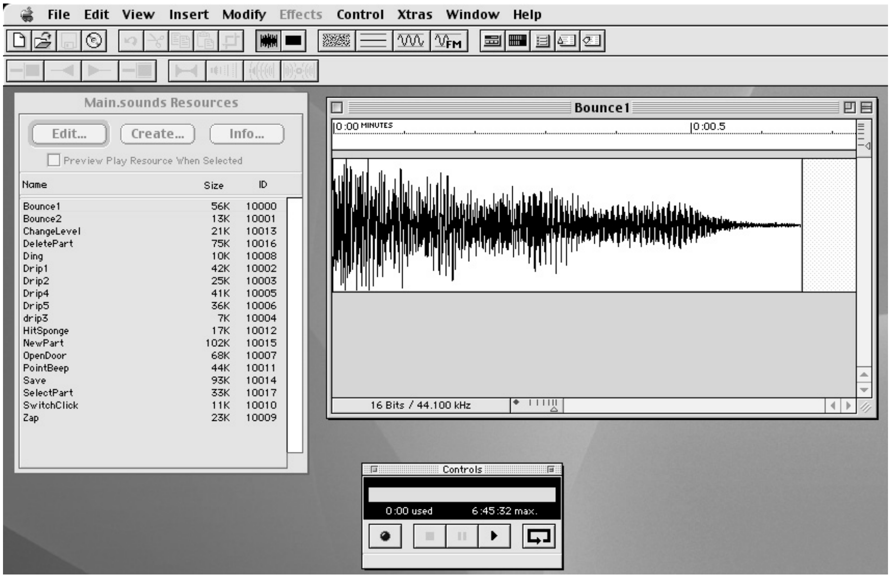
```
pascal void LoopSoundCallback(SndChannelPtr chan,
                              SndCommand *cmd)
{
    SndCommand      theCmd;

    // Play the sound again
    theCmd.cmd      = bufferCmd;
    theCmd.param1   = 0;
    theCmd.param2   = cmd->param2;
    SndDoCommand (chan, &theCmd, true);

    // Just reuse the callBackCmd that got us here in the first place

    SndDoCommand (chan, cmd, true);
}
```

Listing 10-8 shows how our callback function works. It sends a `bufferCmd` to the sound channel to cause the sound to play from the beginning, and then it puts the current command (the command which caused the callback) back into the channel so that we'll get called again the next time the sound needs to loop.

Sound Resources

There are a number of ways to load sound effects for the Sound Manager to use, but the easiest and cleanest method is to use old-fashioned Sound Resources. Apple frowns on modern OS X applications having any files with resource forks, but since they haven't provided us with any better way to store and access the hundreds of sound effects that our games will need, feel free to use them. The only downside to using Sound Resources is that no modern audio application supports them, so to work with them you've got to either write your own utility for copying sound files into resources, or you've got to use something like SoundEdit 16 to do it for you.

SoundEdit 16 is an old but wonderful sound effect editing application. It does pretty much everything that I've ever needed for any sound effects in my games, so I still use it to this day, and it's the only Classic application I ever have to run.



Name	Size	ID
Bounce1	56K	10000
Bounce2	13K	10001
ChangeLevel	21K	10013
DeletePart	75K	10016
Ding	10K	10008
Drip1	42K	10002
Drip2	25K	10003
Drip4	41K	10005
Drip5	36K	10006
drip3	7K	10004
HitSponge	17K	10012
NewPart	102K	10015
OpenDoor	68K	10007
PointBeep	44K	10011
Save	93K	10014
SelectPart	33K	10017
SwitchClick	11K	10010
Zap	23K	10009

Figure 10-2: SoundEdit 16 with the sound resources from Enigmo open.

Using sound resources in our game engine is really easy. When the game starts up we need to pre-load all of the sound resources into memory with a function like this:

Listing 10-9:Loading Sound Resources

```
SndListHandle       gSndHandles[MAX_EFFECTS];
long                gSndOffsets[MAX_EFFECTS];

void LoadSoundResources(FSSpec *spec)
{
    short   refnum, numSoundsInFile,i;


            /* OPEN THE RESOURCE FILE FILE */

    refnum = FSpOpenResFile(spec, fsCurPerm);
    if (refnum == -1)
        DoFatalAlert("\pSound resource file not found!");

    UseResFile(refnum);

            /* COUNT # OF SND RESOURCES IN THE FILE */

    numSoundsInFile = Count1Resources('snd ');
    if (numSoundsInFile > MAX_EFFECTS)
        DoFatalAlert("\pToo many sounds!");
```

```
            /***************************/
            /* LOAD EACH SOUND RESOURCE */
            /***************************/

    for (i=0; i < numSoundsInFile; i++)
    {
                /* GET THE SOUND */

        gSndHandles[i] = (SndListResource **)GetResource('snd ',
                              10000 + i);

        if (gSndHandles[i] == nil)
            DoFatalAlert("\pGetResource failed!");

            /* CONVERT TO A REGULAR HANDLE & LOCK IT */

        DetachResource((Handle)gSndHandles[i]);
        HNoPurge((Handle)gSndHandles[i]);
        HLockHi((Handle)gSndHandles[i]);

                /* GET OFFSET TO SOUND HEADER */

        GetSoundHeaderOffset(gSndHandles[i], &gSndOffsets[i]);
    }

    CloseResFile(refnum);
}
```

The code in Listing 10-8 is pretty standard Resource Manager stuff that you've probably seen a million times before if you're used to programming on older versions of Mac OS. We open the resource fork of the desired file, count the number of 'snd' resources in the file, and then start loading them in with `GetResource()`. It is important that this data doesn't move around, so we detach the handle from the resource and then lock it down and make sure that it's not purgeable:

```
        HNoPurge((Handle)gSndHandles[i]);
        HLockHi((Handle)gSndHandles[i]);
```

The sound resource has some extra data in it that we don't need, so we call `GetSoundHeaderOffset()` to find the offset in the resource to the start of the actual sound header that the Sound Manager needs:

```
        GetSoundHeaderOffset(gSndHandles[i], &gSndOffsets[i]);
```

We'll see how this offset is used later when we actually play one of these sound effects.

Once we've got our sound resources loaded, it's quite easy to play one of them. The first step will be to locate an available sound channel to play it on. We've already initialized our list of sound channels, so the new function `FindSilentChannel()` will scan our list and return the first available channel – a channel that is not currently playing any sound:

Listing 10-10: Finding a Free Sound Channel

```
static short FindSilentChannel(void)
{
    short       c;
    OSErr       iErr;
    SCStatus    theStatus;

    for (c = 0; c < gNumChannels; c++)
    {
            /* LOOPING CHANNELS ARE OBVIOUSLY NOT AVAILABLE */

        if (gChannelInfo[c].isLooping)
            continue;

                /* GET CHANNEL STATUS */

        iErr = SndChannelStatus(gChannelInfo[c].channel,
                                sizeof(SCStatus),
                                &theStatus);
        if (iErr != noErr)
            continue;

                /* SEE IF CHANNEL IS BUSY */

        if (theStatus.scChannelBusy)
            continue;

        return(c);
    }


            /* NO FREE CHANNELS */

    return(-1);
}
```

There are two reasons why we check the `isLooping` flag of each channel: For starters, this is simply faster than calling `SndChannelStatus()`, so if a sound effect is looping then we'll know right away and can avoid one extra call to that function. Additionally, `SndChannelStatus()` can sometimes give incorrect information on channels that have been looped. When we call `SndChannelStatus()` the `scChannelBusy` flag is supposed to be set to `true` if the channel is playing audio, however, this flag will sometimes appear as `false` if an effect has been looped. The flag is always valid on single-shot effects, however.

Now let’s see the code to actually play some sound:

Listing 10-11: Playing a Sound Effect

```
short  PlayEffect(short sndNum, u_long leftVolume, u_long rightVolume,
                u_long rateMultiplier)
{
    SndCommand      mySndCmd;
    SndChannelPtr   chanPtr;
    short           chanNum;
    u_long          loopStart, loopEnd;
    SoundHeaderPtr  sndPtr;

            /* GET POINTER TO THE SOUND HEADER */

    sndPtr = (SoundHeaderPtr)(((long)*gSndHandles[sndNum]) +
    gSndOffsets[sndNum]);


            /* GET A FREE CHANNEL */

    chanNum = FindSilentChannel();
    if (chanNum == -1)                  // no free channels
        return(-1);


            /*********************************/
            /* ISSUE COMMANDS TO THE CHANNEL */
            /*********************************/

    chanPtr = gChannelInfo[chanNum].channel;

        /* FLUSH ANY OLD COMMANDS OUT OF THE CHANNEL */

    mySndCmd.cmd = flushCmd;
    mySndCmd.param1 = 0;
    mySndCmd.param2 = 0;
    SndDoImmediate(chanPtr, &mySndCmd);

    /* MAKE SURE NO SOUND IS PLAYING ON THIS CHANNEL */

    mySndCmd.cmd = quietCmd;
    mySndCmd.param1 = 0;
    mySndCmd.param2 = 0;
    SndDoImmediate(chanPtr, &mySndCmd);

            /* SET LEFT & RIGHT VOLUME FOR CHANNEL */
            //
            // param2 = 32-bit word containing the right
            //      volume in the upper 16 bits, and the
            //      left volume in the lower 16 bits.
            //

    mySndCmd.cmd = volumeCmd;
    mySndCmd.param1 = 0;
    mySndCmd.param2 = (rightVolume  << 16) | leftVolume;
    SndDoCommand(chanPtr, &mySndCmd, true);

            /* START PLAYING THE CHANNEL */
            //
            // param2 points to the sound's header
            //

    mySndCmd.cmd = bufferCmd;
    mySndCmd.param1 = 0;
    mySndCmd.param2 = sndPtr;
    SndDoCommand(chanPtr, &mySndCmd, true);

            /* SET THE PLAYBACK FREQUENCY */
            //
            // This has to be done after bufferCmd above
            //

    mySndCmd.cmd        = rateMultiplierCmd;
    mySndCmd.param1     = 0;
    mySndCmd.param2     = rateMultiplier;
    SndDoImmediate(chanPtr, &mySndCmd);


        /************************************/
        /* SEE IF THIS IS A LOOPING EFFECT */
        /************************************/
        //
        // We look in the sound's header to see if it loops.
        //


    loopStart = sndPtr->loopStart;
    loopEnd = sndPtr->loopEnd;
    if ((loopStart + 1) < loopEnd)
    {
            /* SET THE CALLBACK COMMAND */

        mySndCmd.cmd        = callBackCmd;
        mySndCmd.param1     = 0;
        mySndCmd.param2     = sndPtr;
        SndDoCommand(chanPtr, &mySndCmd, true);

        gChannelInfo[chanNum].isLooping = true;
    }
    else
        gChannelInfo[chanNum].isLooping = false;


            /* SET MY INFO */

    return(chanNum);                            // return channel #
}
```

Our `PlayEffect()` function takes a left and right volume value and something called a rate multiplier value which determines the frequency at which the sound is played. In the Sound.h framework file the constant `kFullVolume` is defined. Passing this value to `PlayEffect()` will cause the left and/or right channels to be played at their maximum volume without over-amplifying it. If you wanted to play the sound at half volume then you would pass in `kFullVolume / 2`. Or, if you did want to over-amplify the effect you could pas in `kFullVolume * 2` which would double the volume of the channel, but it would most likely cause some waveform clipping and distortion since the sound wave is being amplified beyond its limits.

Unfortunately, there are no built-in constants in Sound.h for the rate multiplier, so we've got to make our own:

```
    #define  kNormalRate        0x10000
```

This rate value is a fixed-point multiplier where the lower 16-bits represent the fractional value, so `0x10000` is essentially 1.0. If we wanted to lower the pitch by 50% then we'd pass in `kNormalRate / 2`. Or, to double the pitch, we'd pass `kNormalRate * 2`.

So, to play sound effect #0 with the stereo volume shifted right, but at the default pitch, we call our `PlayEffect()` function like so:

```
    PlayEffect(0, kFullVolume/3, kFullVolume, kNormalRate);
```

Once a sound channel is playing we can continue to send commands to it to alter the playback parameters. Here's a function to change the volume of the channel as it's playing:

Listing 10-12: Changing the Channel Volume

```
void ChangeChannelVolume(short channel, u_long leftVol, u_long rightVol)
{
    SndCommand      mySndCmd;
    SndChannelPtr   chanPtr;

    chanPtr = gChannelInfo[channel].channel;    // get the channel ptr


            /* SEND A VOLUME COMMAND TO THE CHANNEL */

    mySndCmd.cmd = volumeCmd;
    mySndCmd.param1 = 0;
    mySndCmd.param2 = (rightVol << 16) | leftVol;
    SndDoImmediate(chanPtr, &mySndCmd);
}
```

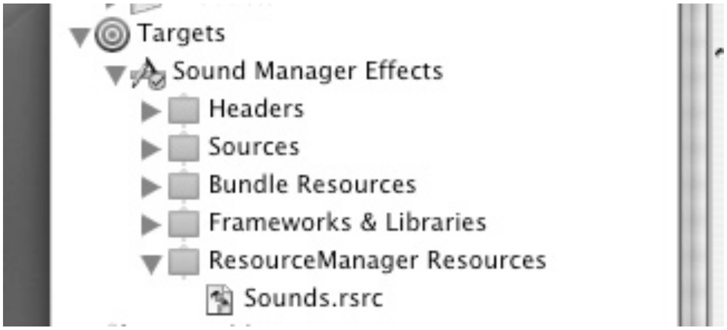
The results of a volume command are pretty much instantaneous thanks to the `SndDoImmediate()` call. You can very smoothly ramp the volume of each channel up and down as you need in your game, and it will sound very even. The same goes for changing the pitch of the sound channel:

Listing 10-13: Changing the Channel Pitch

```
void ChangeChannelRate(short channel, long rateMult)
{
    SndCommand      mySndCmd;
    SndChannelPtr   chanPtr;

    chanPtr = gChannelInfo[channel].channel;    // get the channel ptr

            /* SEND A RATE MULTIPLIER COMMAND */

    mySndCmd.cmd        = rateMultiplierCmd;
    mySndCmd.param1     = 0;
    mySndCmd.param2     = rateMult;
    SndDoImmediate(chanPtr, &mySndCmd);
}
```

Including Sound Resource Files in an Xcode Project

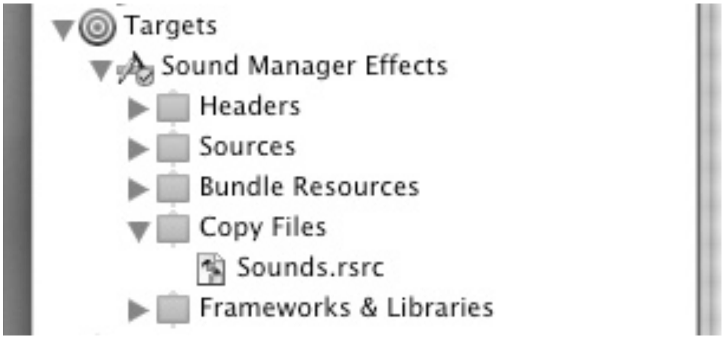
The sample project “Sound Manager Effects.xcode” demonstrates all of these functions, but there is one part of this project that is important to take notice of: Xcode is a little picky about including old fashioned .rsrc files in a project, so we have to do something funky to get it to properly copy and install our Sounds.rsrc file into the game’s Resources folder. Normally when you drag an .rsrc file into an Xcode project it adds that file to the Target's "Resource Manager Resources" build phase (see Figure 10-3):



- Targets
  - Sound Manager Effects
    - Headers
    - Sources
    - Bundle Resources
    - Frameworks & Libraries
    - ResourceManager Resources
      - Sounds.rsrc

Figure 10-3: The Sound.rsrc file in the Resource Manager Resources build phase

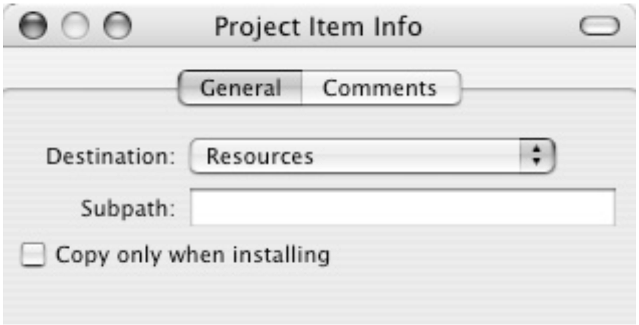
This won't actually work, because Xcode will not properly copy that Sound.rsrc file. The results of this would be a corrupt resource file, an incorrectly named resource file, or both. To force Xcode to copy any old-style resource file, you must add a new "Copy Files" build phase, move the .rsrc file to that phase, and then delete the Resource Manager Resources build phase. It should end up looking like this:



- Targets
  - Sound Manager Effects
    - Headers
    - Sources
    - Bundle Resources
    - Copy Files
      - Sounds.rsrc
    - Frameworks & Libraries

Figure 10-4: A Copy Files build phase is the only way to include .rsrc files

You create a new Build Phase by right-clicking on the Target and then selecting a new Build Phase type from the pop-up menu's list. When you create the Copy Files build phase, be sure to specify that its Destination is set to "Resources":



Destination: Resources

Subpath:

☐ Copy only when installing

Figure 10-5: Set the Copy Files build phase destination to Resources

Compile and run the sample project. Then look in the game's Special menu. You'll see some menu items for playing different sound effects so that you can test sounds from there.

Every game that I have ever written has used the Sound Manager to play effects. Even though Apple has essentially discontinued the Sound Manager, it is still a part of OS X and it works great! But, if you're feeling adventurous you might want to consider using the next technology that I'm going to discuss: OpenAL.

OpenAL

OpenAL is the audio equivalent of OpenGL. It is a standardized audio library that exists on many platforms, and has recently been adopted by Apple as the way of the future for playing sound in applications such as games. One of the nice things about OpenAL is that unlike the Sound Manager which just plays basic waveforms with some easy pitch and volume options, OpenAL supports full 3D spatial placement effects like Doppler shift, reverb, etc. In theory, if any hardware accelerated audio cards ever becomes available for the Mac then this technology should make use of it.

That's the good news. Unfortunately, the first version of OpenAL that will be feature complete is the upcoming version for Mac OS 10.4. The current version of OpenAL at the time of this writing is 1.0.12 and it's not very good. It only does simple attenuated and panned 3D sounds. No Doppler shifting or anything else.

The Pro's of OpenAL

- The API is much simpler yet more powerful than the Sound Manager. There's none of the messy sound command code that we're stuck with when using the Sound Manager.

• Support for full 3D sound including panning, attenuation, Doppler shifts, and even support for multi-speaker sound systems, not just 2-speaker stereo systems.

• If sound hardware ever becomes readily available for the Mac, your old games using OpenAL may just suddenly support it.

• OpenAL is standardized across many platforms, so porting your game will be much easier since you won't have to re-write the entire sound engine.

The Con's of OpenAL

• No built-in way to organize your hundreds of sound effects. The Sound Manager lets you use Sound Resources to organize all of your game's sound effects, but with OpenAL, you've either got to design your own system for grouping sounds into data files, or you're stuck with just having hundreds of little WAV files sitting around that you've got to manually load.

• No way to handle audio in games that have 2-player split-screen modes. OpenAL only has one "Listener" so if you have more than one camera as in a split-screen game like Cro-Mag Rally or Nanosaur 2, you're out of luck. In my games that support a split-screen mode I manually calculate the sound volume such that each sound plays at the loudest volume of either Player 1 or Player 2. This gives the best sound for such a situation, but OpenAL has no means of doing this. It will only play audio based on either Player 1 or Player 2's camera, but not both. I consider this to be a major, major oversight by the OpenAL design committee.

• As of Mac OS 10.3, OpenAL is not actually a part of the OS. If your game supports OpenAL, you've either got to include the OpenAL libraries with your game's installer, or tell the user to go download the latest OpenAL installer from www.openal.org. From a marketing standpoint, this is extremely bad. However, Apple will include OpenAL as part of future versions of Mac OS X.

• No support for non-zero loop-back points. OpenAL supports looping sounds, but it only loops back to the very beginning of the sound; you cannot have a loop point that's elsewhere in the sound like you can with the Sound Manager. Suppose you wanted to have a guitar string effect in your game. The first part of the sound is the "attack" and the second part is the "sustain" - the part that loops. If you looped the whole sound, it would have a "Pling! Pling! Pling! Pling!" sound to it, but if you set a loop-point at the start of the sustain section, it would have a "Plingggggggggg" sound to it which is the correct behavior.

• Faking panning is a hassle. Sometimes you simply want to play an effect, and have it sound like it's coming from the left speaker (such as is often done on menus screens). With the

Sound Manager this is as easy as just passing a volume command to the sound channel. But OpenAL has no manual way to do this. You would actually have to play the sound in 3D space in such a way that it does the desired panning.

Hopefully, over time many of the cons will go away as more features are added to OpenAL, but until then you'll have to consider whether to go with OpenAL or Sound Manager for your immediate game programming needs. If your primary concern is making sure that everyone can play your game, then just stick with the Sound Manager, but if your primary concern is having cool 3D audio then use OpenAL. For the remainder of the sample code in this book we're going to be using OpenAL for playing sound effects, and Quicktime for streaming our music.

Initializing OpenAL

There are four fundamental parts of OpenAL that need to be set up in order to play sounds: the AL Context, Buffers, Sources, and the Listener. Before we proceed, we should define what these terms mean:

AL Context

In the same way that an OpenGL context refers to the window or display that you are drawing graphics into, an OpenAL context refers to the sound system on your computer.

Buffers

A Buffer is your sound wave data plus some state information that determines how that sound should be played.

Sources

This is a point in your 3D world that makes sound – a sound source. If there is an explosion in your game, then you'll create a Source at the coordinate of the explosion.

Listener

There is only one Listener in an OpenAL context. It represents your ears in space, which, in a game is typically the same as the camera location.

First things first. Let's initialize the OpenAL Context:

Listing 10-14: Initializing the OpenAL Context

```
void OAL_Init(void)
{
    ALCcontext  *al_context;
    ALCdevice   *al_device;
    OSErr       iErr;


            /* CREATE AN OPENAL CONTEXT */

    al_device   =   alcOpenDevice(nil);
    al_context  =   alcCreateContext(al_device,0);
    alcMakeContextCurrent(al_context);

    if (alGetError() != AL_NO_ERROR)
        DoFatalAlert("\pcreating OpenAL context failed");


        /* SET SOME STATE INFO */

    alDistanceModel(AL_INVERSE_DISTANCE_CLAMPED);
    alEnable(ALC_CONVERT_DATA_UPON_LOADING);
    alSetInteger(ALC_SPATIAL_RENDERING_QUALITY,
                ALC_SPATIAL_RENDERING_QUALITY_LOW);
    alSetInteger(ALC_RENDER_CHANNEL_COUNT,
                 ALC_RENDER_CHANNEL_COUNT_STEREO);

    iErr= alGetError();
    if (iErr != AL_NO_ERROR)
        DoFatalAlert("\psetting OpenAL state failed");


            /* INIT ALUT */

    alutInit(nil, nil);
}
```

We pass `nil` to `alcOpenDevice()` which tells OpenAL to open a connection to the default audio device. Next we call `alcCreateContext()` to create our audio context reference, and then `alcMakeContextCurrent()` to set it as the current context. That's all there is to it! Much easier than creating an OpenGL context, eh?

Now that we have an active OpenAL context we need to set up some state information just like we would do in OpenGL. The first thing we set is the distance model. The distance model determines how the volume of a sound will decay over distance. The best setting for this is `AL_INVERSE_DISTANCE_CLAMPED`. This causes the correct decay in volume over distance, but also keeps the audio from getting over amplified if the camera gets too close to the sound Source. It's the virtual equivalent of not blowing out your eardrums.

Next, we enable an important feature of OpenAL on the Mac:

```
    alEnable(ALC_CONVERT_DATA_UPON_LOADING);
```

OpenAL on the Mac is built on top of Core Audio, and Core Audio uses floating-point values to represent data in a digital waveform. This is very unusual since most sound data is in the form of 8, 16, or 24-bit integer values. Converting this traditional waveform data on the fly while a sound is playing is expensive, so by enabling `ALC_CONVERT_DATA_UPON_LOADING` we're telling OpenGL to do this integer to float conversion when sounds are loaded.

Games require a lot of CPU power, so sometimes sacrifices must be made in the name of frame rate. Audio processing can be a very CPU intensive thing, especially if there are dozens of sound effects playing simultaneously, therefore, it is wise to set the OpenAL spatial rendering quality to the low setting:

```
    alSetInteger(ALC_SPATIAL_RENDERING_QUALITY,
                 ALC_SPATIAL_RENDERING_QUALITY_LOW);
```

Odds are that nobody will be able to tell much difference between the low and high settings, but the low setting will save some CPU time.

The next parameter that we set is an interesting one. Most Mac users don't have fancy quadraphonic sound systems, but rather they have simple, stereo, 2-speaker systems. We should let OpenAL know this:

```
    alSetInteger(ALC_RENDER_CHANNEL_COUNT,
                  ALC_RENDER_CHANNEL_COUNT_STEREO);
```

In rare cases, however, a user might actually have a multi-speaker setup for doing true surround sound on their computer. In this case you could pass the constant `ALC_RENDER_CHANNEL_COUNT_MULTICHANNEL` to `alSetInteger()`. With this parameter set, OpenAL will correctly handle these systems.

Just like in OpenGL where there is a glut (GL Utility) library, in OpenAL there is an alut (AL Utility) library. Before making any alut calls, we need to initialize it with a call to `alutInit()`. The most important alut call that we'll be using is `alutLoadWAVFile()`.

Loading Sound Files for OpenAL

As was mentioned earlier, OpenAL has no way of organizing sound files like we can do with the Sound Manager. The only built-in support for sound files in OpenAL 1.0 is a function called `alutLoadWAVEFile()` for automatically loading a WAV sound file, however future versions of OpenAL will support other file formats. To load a bunch of sound files into our game we need a function like this:

Listing 10-15: Loading WAV Files

```
void OAL_LoadWAVFiles(short numFiles, char *filenames[])
{
    short       i;
    ALenum      format;
    ALsizei     freq;
    void        *data;
    ALsizei     size;
    OSErr       iErr;
    FSSpec      spec;
    static char fullPathname[MAX_PATHNAME_LEN];

    if (numFiles > MAX_SOUND_BUFFERS)           // check for overflow
        DoFatalAlert("\pnumFiles > MAX_SOUND_BUFFERS");


            /*************************************/
            /* LOAD EACH WAV INTO A SOUND BUFFER */
            /*************************************/
            //
            // Buffers store the information about how a sound should
            // be played and the sound data itself.  So, here we
            // essentially create a buffer and load a WAV file into it.
            //

    for (i = 0; i < numFiles; i++)
    {
                /* CREATE AN FSSPEC FOR THIS WAV FILE */

        iErr = FSMakeFSSpec(gMyResourcesFolderFSSpec.vRefNum,
                        gMyResourcesFolderFSSpec.parID,
                        filenames[i],
                        &spec);

        if (iErr != noErr)
            DoFatalAlert("\pCannot find our WAV file");

                /* CONVERT THE FSSPEC TO A FULL PATHNAME */

        GetFullPathFromFSSpec(&spec, fullPathname, 1000);
```

```
                /* LOAD THE WAV FROM THE PATHNAME */

        alutLoadWAVFile(fullPathname, &format, &data, &size, &freq);


            /* ALLOCATE A BUFFER */

        alGenBuffers(1, &gSoundBuffers[i]);


            /* COPY WAV'S DATA TO THE BUFFER */

        alBufferData(gSoundBuffers[i], format, data, size, freq);


            /* UNLOAD THE WAV FILE */

        alutUnloadWAV(format, data, size, freq);
    }
}
```

The `OAL_LoadWAVFiles()` function takes a list of filenames as input, and then it goes through this list loading each WAV file into an OpenAL Buffer. The process is straightforward. First we need to generate a full pathname to the WAV file we're trying to open. Our WAV files are in the application's Resources folder, so we create an `FSSpec` for the file with a simple call to `FSMakeFSSpec()`. To convert this `FSSpec` to a text pathname we need to write a new utility function:

Listing 10-16: Converting an FSSpec to a Full Pathname

```
void GetFullPathFromFSSpec(const FSSpec *spec, UInt8 *fullpath,
                        int maxBufLen)
{
    OSErr iErr;
    FSRef theRef;

    iErr = FSpMakeFSRef(spec, &theRef);
    if (iErr != noErr)
        DoFatalAlert("\pFSpMakeFSRef failed");

    FSRefMakePath(&theRef, (UInt8 *) fullpath, maxBufLen);
}
```

The process of getting a path from an `FSSpec` is just a matter of first converting our `FSSpec` to an `FSRef` with a call to `FSpMakeFSRef()`, and then we can get the path by calling `FSRefMakePath()`.

We've got the full pathname of the WAV file, so we pass that text string to `alutLoadWAVFile()`:

```
alutLoadWAVFile(fullPathname, &format, &data, &size, &freq);
```

We don't need to know anything about the format of a WAV file since OpenAL automatically takes care of everything. The `format`, `data`, `size`, and `freq` values that are returned contain some information about the WAV file that will be needed to install it into an OpenAL Buffer, but first we need to create the Buffer to hold all this information:

```
alGenBuffers(1, &gSoundBuffers[i]);
```

This generates one Buffer object for us which can now be loaded with data from the WAV file:

```
alBufferData(gSoundBuffers[i], format, data, size, freq);
```

This copies all of the WAV file's data into the Buffer, so the original WAV file is no longer needed, therefore, we unload it:

```
alutUnloadWAV(format, data, size, freq);
```

Playing a Sound

We've now got a Sound Buffer loaded with audio data, and we're almost ready to try playing it, but first we need to tell OpenAL about our Listener. Remember that the Listener is *you*, or more accurately, it's the camera location. Every time we update the OpenGL camera information in our game, we also need to update the OpenAL Listener information:

Listing 10-17: Setting the Listener Information

```
void OAL_SetListenerInfo(OALPoint3D *coord, OALVector3D *aim,
                        OALVector3D *up, OALVector3D *vector)
{
    OALOrientation  orientation;

            /* SET ORIENTATION STRUCT */

    orientation.aim = *aim;
    orientation.up = *up;

    alListenerfv(AL_ORIENTATION, (ALfloat *)&orientation);
```

```
            /* SET POSITION & VELOCITY */

    alListenerfv(AL_POSITION,    (ALfloat *)coord);
    alListenerfv(AL_VELOCITY,    (ALfloat *)vector);
}
```

All OpenAL calls that affect the Listener are made through `alListener()` by passing in a parameter constant followed by the value of that parameter. The most basic parameters are the position, velocity, and orientation of the Listener. The position of the Listener is the camera coordinate in our 3D world, the velocity is a vector representing the speed of the camera in our 3D world, and the orientation is the camera's look-at vector and up-vector.

When we call `alListener()` and pass in `AL_ORIENTATION`, OpenAL expects the input value to be 6 consecutive `floats`. Those `float`'s make up two vectors: the first vector is the aim or look-at vector of the Listener, and the second is the Listener's up-vector. To make things nice and clean we've defined an `OALOrientation` structure:

```
    typedef struct
    {
        OALVector3D aim;
        OALVector3D up;
    }OALOrientation;
```

OpenAL does not automatically calculate the Listener velocity from frame to frame. Instead, we've got to manually calculate that in our camera update code with something like this:

```
    motion.x = gCameraCoord.x - gOldCameraCoord.x;
    motion.y = gCameraCoord.y - gOldCameraCoord.y;
    motion.z = gCameraCoord.z - gOldCameraCoord.z;
```

Motion values are used to calculate Doppler shift, so if you're not planning on supporting Doppler effects in your game then you really don't need to worry about the Listener velocity.

Everything is now in place for us to play some sound, so let's get to it:

Listing 10-18: Playing a Sound Effect

```
ALuint OAL_PlaySound3D(int bufferNum, OALPoint3D *coord,
                    OALVector3D*vector, float pitch,
                    float gain, Boolean loop)
{
    ALuint  theSource;

                /* CREATE A SOUND SOURCE */
    alGenSources(1, &theSource);

    if (alGetError() != AL_NO_ERROR)
        DoFatalAlert("\palGenSources failed");


                /* ASSIGN A SOUND TO THIS SOURCE */

    alSourcei(theSource, AL_BUFFER, gSoundBuffers[bufferNum]);


                /* SET SOURCE'S POSITION & VELOCITY */

    alSourcefv(theSource, AL_POSITION, (ALfloat *)coord);
    alSourcefv(theSource, AL_VELOCITY, (ALfloat *)vector);

                /* SET SOME OTHER PARAMETERS */

                                    // set distance where volume is 100%
    alSourcef(theSource, AL_REFERENCE_DISTANCE, 2.0);

                                    // set volume fall-off rate
    alSourcef(theSource, AL_ROLLOFF_FACTOR, .8);

                                    // set frequency and volume
    alSourcef (theSource, AL_PITCH, pitch);
    alSourcef (theSource, AL_GAIN, gain);

                                    // set looping flag
    alSourcei(theSource, AL_LOOPING, loop);


            /* PLAY THE SOUND */

    alSourcePlay(theSource);

    return(theSource);
}
```

The first step in playing a sound is to create the OpenAL sound Source:

```
    alGenSources(1, &theSource);
```

This allocates one new sound Source object for us (it's kind of like allocating a new Sound Channel with the Sound Manager). Then we need to assign a Buffer to this Source so that OpenAL will know what sound we want this Source to play. Remember that the Buffer is what contains our WAV file's data, so by assigning it to the Source we're telling the Source to play this sound data.

As with the Listener, sound Sources also have a position and velocity value. The position is used by OpenAL to calculate the volume of the sound, and the velocity is used in Doppler shift calculations. The velocity is not needed if you're not doing Doppler shift in your game. All calls that modify Source parameters are done with the `alSource()` function, so the position and velocity are set like this:

```
alSourcefv(theSource, AL_POSITION, (ALfloat *)coord);
alSourcefv(theSource, AL_VELOCITY, (ALfloat *)vector);
```

Next, we need to set some information about how we want our sound's volume to decay with distance from the Listener. In our OpenAL Context initialization function we told OpenAL that we wanted to use the `AL_INVERSE_DISTANCE_CLAMPED` method, but we have the ability to set specific parameters of this calculation on a per-Source basis:

```
alSourcef(theSource, AL_REFERENCE_DISTANCE, 2.0);
alSourcef(theSource, AL_ROLLOFF_FACTOR, .8);
```

The Reference Distance is the distance at which the sound is at 100% volume. If you move the Listener farther from the Source it will start to decay in volume. The Roll-Off Factor regulates this decay. The higher the Roll-Off Factor, the faster the volume will decay as the Listener moves away from the Source. The lower the Roll-Off Factor, the less the volume decays as the Listener moves away. Even though there is a precise mathematical formula describing how these values work with the inverse distance calculation, the best way to make things work in your game is to "play it by ear". In other words, play with these values until you get the desired volume in your game.

If you want to change the pitch or volume of a sound Source, just do this:

```
alSourcef (theSource, AL_PITCH, pitch);
alSourcef (theSource, AL_GAIN, gain);
```

The input values to these calls are multipliers. So, passing a pitch of 1.0 would not modify the sound sample's playback pitch at all, but passing 2.0 would double the frequency of the pitch. The same goes for the Gain value. Passing 0.5 would cut the volume in half, while passing 2.0 would double it.

OpenAL can automatically handle sounds that loop. Another `alSource()` call is all that's needed:

```
alSourcei(theSource, AL_LOOPING, loop);
```

Finally, we've got our Source configured how we want it, so, to make it start playing…

```
alSourcePlay(theSource);
```

Doppler Shift

In the previous pages I've mentioned that OpenAL supports the Doppler shift effect, so now I'll talk a little more about it. For starters, what is Doppler shift? Simply put, it's the effect that causes a freight train to sound higher pitched as it approaches, and then lower pitched as it moves away. Sound travels at a constant speed through the air, but if a sound emitting object is traveling toward you, the sound waves get slightly compressed which causes the frequency to become higher. As the object moves away, the sound waves get slightly stretched, thus, the frequency becomes lower.

OpenGL can simulate this effect by turning it on:

```
alDopplerFactor(1.0);
alDopplerVelocity(1000);
```

The Doppler Factor value gives you a way to tweak the Doppler effect, making it more exaggerated or less exaggerated. The value of 1.0 is the normal value, but higher numbers will increase the effect by making the pitch of the sound raise higher and faster as objects approach, and values below 1.0 will decrease the effect. The default value is 0.0 which tells OpenAL not to do any Doppler shifting effects. Once you change this to a value over 0.0 Doppler calculations will kick in and you'll get the effect with a small CPU cost.

The Doppler Velocity value is the speed of sound in your universe, so the value of 1000 tells OpenAL that sound travels at 1000 units per second. You will probably need to tweak this value quite a bit until you find the sweet spot where the Doppler shift sounds the best in your game.

These Doppler parameters are global state settings, so once you set them it will affect all of the sound Sources played on the current OpenAL context.

There is one major caveat to using Doppler shifts in your game's sound engine: it isn't implemented in OpenAL 1.0. All of the Doppler function calls are there in OpenAL 1.0, but the functions don't actually work. What's worse, they will generate an `AL_INVALID_ENUM` error message even if you try to call them with legitimate values. This should be fixed in the next major release of OpenAL.

The fact is that the problems with Doppler shifts in OpenAL are indicative of the 1.0 library in general. If you just want to play simple, attenuated stereo effects it will work fine, but honestly, you could do the same thing with a little extra coding using the Sound Manager. Like I've said before, if you want to be on the cutting edge of audio on the Mac then dive right into OpenAL, but if you're more concerned about releasing a stable game that every Mac owner can play then use the Sound Manager. When the next major release of OpenAL is available it will likely be more robust and reliable, but the jury is still out.

Chapter 11: Simple Input

There are three different ways to read keyboard and mouse input on OS X, and I'll be covering the two easy methods here in this chapter. There is no simple way to read input from a gamepad or joystick, so that topic is reserved for Chapter 12 which deals with the dreaded HID Manager.

Let's start with the keyboard. The Mac keyboard generally can only read three keys simultaneously, plus modifiers. That means if you hold down the A, B, C, and D keys, only three of those four keys will register a key press. The modifier keys such as Shift, Option, Control, and Command don't count in that three-key limit. So, for example, you can press the Option, E Y, P, and Shift keys all at the same time and they'll all be read correctly. A common mistake made by many Mac gamers is that they will try to assign non-modifier keys to all of their game's controls, and then they don't understand why they can't sometimes fire when they're moving around. You should assign modifier keys to game controls whenever possible since they don't count in that 3-key limit.

Here's a better example of this going awry: Say you have player movement assigned to the arrow keys and the player is moving diagonally by pressing the up and left arrow keys. That's two keys down. Now suppose you've got the duck key set to the D key and the fire key set to F. Well, that's four keys total. Something has got to give, so either the player won't be able to duck, or to shoot, or to move in the direction they wanted. The solution is to assign the duck and shoot keys to modifier keys.

Reading the Keyboard with GetKeys()

If you've been programming the Mac for a long time then you probably know about the old toolbox call `GetKeys()`. This function is the easiest way to read the keyboard on the Mac because it returns a 128-bit value (the `KeyMap`) where each bit represents the state of a key on the keyboard. By testing a specific bit of this value you can instantly determine if a key is pressed or not.

Listing 11-1:Reading the KeyMap with GetKeys()

```
KeyMap gKeyMap, gNewKeys, gOldKeys;

void UpdateKeyMap(void)
{
    GetKeys(gKeyMap);

            /* CALC WHICH KEYS ARE NEW THIS TIME */

    gNewKeys[0] = (gOldKeys[0] ^ gKeyMap[0]) & gKeyMap[0];
    gNewKeys[1] = (gOldKeys[1] ^ gKeyMap[1]) & gKeyMap[1];
    gNewKeys[2] = (gOldKeys[2] ^ gKeyMap[2]) & gKeyMap[2];
    gNewKeys[3] = (gOldKeys[3] ^ gKeyMap[3]) & gKeyMap[3];


            /* REMEMBER AS OLD MAP */

    gOldKeys[0] = gKeyMap[0];
    gOldKeys[1] = gKeyMap[1];
    gOldKeys[2] = gKeyMap[2];
    gOldKeys[3] = gKeyMap[3];
}
```

The type `KeyMap` is defined by the system headers as an array of four 32-bit `long`'s which gives us 128 bits total. The current `KeyMap` is acquired by calling `GetKeys()`:

```
    GetKeys(gKeyMap);
```

Next we determine which key bits are new this time by XOR'ing the old bit-field with the new bit-field, and then AND'ing it with the new bit-field. Did I just confuse you? Let's take a closer look at that logic with this example:

```
    newBits             = 000110010
    oldBits             = 100010000

    new XOR old         = 100100010     // these are the bits that changed
    AND with newBits    = 000100010     // these are the new bits
```

This leaves `1`'s in the bits of keys that are pressed now, but were not pressed previously.

With `gNewKeys` or `gKeyMap` we can easily test the value of any key. The only problem is that we have to know which keys are assigned to which bits in this 128-bit `KeyMap`. There's no rhyme or reason to how this works since it's not in any ASCII or alphabetical order. The easiest way to deal with this is to just use a large list of constants that define the values of all of the commonly used keys:

Listing 11-2:The KeyMap Bits

```
#define KEY_A               0x00
#define KEY_B               0x0b
#define KEY_C               0x08
#define KEY_D               0x02
#define KEY_E               0x0e
#define KEY_F               0x03
#define KEY_G               0x05
#define KEY_H               0x04
#define KEY_I               0x22
#define KEY_J               0x26
#define KEY_K               0x28
#define KEY_L               0x25
#define KEY_M               0x2e
#define KEY_N               0x2d
#define KEY_O               0x1f
#define KEY_P               0x23
#define KEY_Q               0x0c
#define KEY_R               0x0f
#define KEY_S               0x01
#define KEY_T               0x11
#define KEY_U               0x20
#define KEY_V               0x09
#define KEY_W               0x0d
#define KEY_X               0x07
#define KEY_Y               0x10
#define KEY_Z               0x06

#define KEY_1               0x12
#define KEY_2               0x13
#define KEY_3               0x14
#define KEY_4               0x15
#define KEY_5               0x17
#define KEY_6               0x16
#define KEY_7               0x1a
#define KEY_8               0x1c
#define KEY_9               0x19
#define KEY_0               0x1d

#define KEY_K0              0x52
#define KEY_K1              0x53
#define KEY_K2              0x54
#define KEY_K3              0x55
#define KEY_K4              0x56
#define KEY_K5              0x57
#define KEY_K6              0x58
#define KEY_K7              0x59
#define KEY_K8              0x5b
#define KEY_K9              0x5c

#define KEY_PERIOD          0x2f
```

```
#define KEY_QMARK           0x2c
#define KEY_COMMA           0X2b

#define KEY_TAB             0x30
#define KEY_ESC             0x35
#define KEY_CAPSLOCK        0x39
#define KEY_APPLE           0x37
#define KEY_SPACE           0x31
#define KEY_OPTION          0x3a
#define KEY_CTRL            0x3b
#define KEY_UP              0x7e
#define KEY_DOWN            0x7d
#define KEY_LEFT            0x7b
#define KEY_RIGHT           0x7c
#define KEY_SHIFT           0x38
#define KEY_DELETE          0x33
#define KEY_RETURN          0x24
#define KEY_MINUS           0x1b
#define KEY_PLUS            0x18

#define KEY_F1              0x7a
#define KEY_F2              0x78
#define KEY_F3              0x63
#define KEY_F4              0x76
#define KEY_F5              0x60
#define KEY_F6              0x61
#define KEY_F7              0x62
#define KEY_F8              0x64
#define KEY_F9              0x65
#define KEY_F10             0x6d
#define KEY_F11             0x67
#define KEY_F12             0x6f
#define KEY_F13             0x69
#define KEY_F14             0x6b
#define KEY_F15             0x71

#define KEY_TILDE           0x32
#define KEY_HELP            0x72
```

This list of constants tells us which bit to test for each key. The A key is bit #0, the Tab key is bit #0x30, and so on. So, to test our bits we'll write a function like this:

Listing 11-3:Testing KeyMap Bits

```
Boolean GetKeyState(unsigned short key)
{
    unsigned char *keyMap = (unsigned char *)&gKeyMap;
    return ((keyMap[key >> 3] >> (key & 7) ) & 1);
}
```

We get a char pointer to our `gKeyMap` value and then do some fancy shifting and masking on it. The result is either a 0 or a 1 to indicate the bit was set.

That's all there is to it, but there are a few issues to be aware of: Unfortunately, there is a bug in OS X (up to version 10.3 as far as I know) in which `GetKeys()` will simply begin to fail, and the only way to fix it is to reboot the computer. What will happen is that certain keys will fail to set their corresponding bits in the `KeyMap` bit-field. This happens about 1 in 300 times a person plays any game that uses `GetKeys()`. Once the failure starts, it will affect any and all applications that use `GetKeys()` until the machine is rebooted.

The other issue to be aware of is that international keyboards may alter some key mappings. The letters, numbers, and other general keys always stay the same, but some keyboards simply don't have certain keys or do have others. For example, the tilde key doesn't exist on the German keyboard, while the French keyboard has additional keys for their fancy accented letters. These issues are minor and manageable, so I've continued to use `GetKeys()` to some degree in all of my games.

Reading the Keyboard with Carbon Events

Back in Chapter 9 we discussed how to write an Event handler to process a game's menu bar. Well, that same Event hander can be modified to also handle keyboard events. Instead of only looking for `kEventCommandProcess` events, we'll also look for keyboard events:

Listing 11-4:Adding Keyboard Events to our Event Handler

```
void InitMyCommandEventHandler(void)
{
    OSStatus  iErr;
    EventTypeSpec   events[4] =
{
        kEventClassCommand, kEventCommandProcess,
        kEventClassKeyboard, kEventRawKeyDown,
        kEventClassKeyboard, kEventRawKeyUp,
        kEventClassKeyboard, kEventRawKeyModifiersChanged
};


            /*******************************************/
            /* LOAD AND SET MENU BAR FROM OUR NIB FILE */
            /*******************************************/

    iErr = SetMenuBarFromNib(gNibs, CFSTR("MainMenu"));
    if (iErr != noErr)
        DoFatalAlert("\pSetMenuBarFromNib failed!");
```

```
            /************************/
            /* CREATE EVENT HANDLER */
            /************************/

    gMyEventHandlerUPP = NewEventHandlerUPP(MyEventHandler);

    InstallEventHandler(GetApplicationEventTarget(), gMyEventHandlerUPP,
                        4, events, nil, &gMyEventHandlerRef);
}
```

This new version of `InitMyEventHandler()` looks basically the same as that from Listing 9-4. The only difference is that we've added more event types to look for:

`kEventRawKeyDown`
These events occur whenever a key is pressed.

`kEventRawKeyUp`
These events occur whenever a key is released.

`kEventRawKeyModifiersChanged`
These events occur when a modifier key is pressed or released.

Now we need to update our event handler function to process these keyboard events as they come in:

Listing 11-5:Handling Keyboard Events

```
static pascal OSStatus MyEventHandler(EventHandlerCallRef eventhandler,
                                      EventRef event, void *userdata)
{
    OSStatus    result = eventNotHandledErr;
    HICommand   command;
    UInt32      eventClass, eventKind;
    char        charCode;

            /* GET EVENT CLASS & KIND */

    eventClass = GetEventClass(event);
    eventKind  = GetEventKind(event);

    switch (eventClass)
    {
                /************************/
                /* HANDLE COMMAND EVENTS */
                /************************/
```

```
case    kEventClassCommand:

                /* EXTRACT COMMAND INFO FROM EVENT */

        GetEventParameter(event, kEventParamDirectObject,
                        typeHICommand, nil,
                        sizeof(HICommand), nil, &command);

                /* HANDLE THE COMMAND */

        switch (command.commandID)
        {
            case    'quit':         // Quit menu item
                    gQuitApplication = true;
                    result = noErr;
                    break;
        }
        break;


        /*************************/
        /* HANDLE KEYBOARD EVENTS */
        /*************************/

case    kEventClassKeyboard:

        switch(eventKind)
        {
            case    kEventRawKeyDown:
                    GetEventParameter(event,
                                    kEventParamKeyMacCharCodes,
                                    typeChar, nil,
                                    sizeof(charCode), nil,
                                    &charCode);
                    gKeyState[charCode] = true;
                    break;

            case    kEventRawKeyUp:
                    GetEventParameter(event,
                                    kEventParamKeyMacCharCodes,
                                    typeChar, nil,
                                    sizeof(charCode), nil,
                                    &charCode);
                    gKeyState[charCode] = false;
                    break;

            case    kEventRawKeyModifiersChanged:
                    GetEventParameter(event,
                                    kEventParamKeyModifiers,
                                    typeUInt32, nil,
                                    sizeof(gModifiers), nil,
                                    &gModifiers);
                        break;
                }
                break;
        }

    return(result);
}
```

Half of this function looks the same as it did in Chapter 9, but we've added a bunch of new code to handle our new keyboard events. First, we need to determine what kind of event our handler has received:

```
    eventClass = GetEventClass(event);
    eventKind  = GetEventKind(event);
```

The event class and event kind tell us exactly what event occurred. When a key-down or key-up event occurs, we need to find out which key was pressed:

```
    GetEventParameter(event, kEventParamKeyMacCharCodes, typeChar, nil,
                    sizeof(charCode), nil, &charCode);
```

The `charCode` returned is the ASCII character value of the key. So if the G key was pressed, the `charCode` will be "G". To track which keys are pressed and not pressed, we have an array of 256 Boolean flags. The ASCII character is an index into this array.

The modifier keys are handled separately by the `kEventRawKeyModifiersChanged` event, and the value of all of the modifier keys is obtained with another `GetEventParameter()` call:

```
    GetEventParameter(event, kEventParamKeyModifiers, typeUInt32, nil,
                        sizeof(gModifiers), nil, &gModifiers);
```

The `gModifiers` variable will contain a bit-mask where each bit represents a modifier key; similar to how the `KeyMap` bits worked earlier. To test the value of a modifier key, there is a list of constants in the system header Events.h that we can mask against:

```
    cmdKeyBit
    shiftKeyBit
    alphaLockBit
    optionKeyBit
    controlKeyBit
    rightShiftKeyBit
    rightOptionKeyBit
    rightControlKeyBit
```

So, for example, if we wanted to see if the Option key was pressed or not, we do this:

```
    if (gModifiers & optionKeyBit)
        printf(“option key is pressed”);
    else
        printf(“option key is not pressed”);
```

This method of reading the keyboard is quite common in Mac games, but as you can see it is a lot more work than just using `GetKeys()`. The benefit, however, is that you get the actual ASCII value of a key which is nice for things like High Scores screens where the user is actually typing something.

Reading the Mouse

If you only need to know the window coordinates and the status of the button on the mouse, then life is pretty easy:

Listing 11-6:Reading the Mouse Coordinates

```
void GetMouseScreenCoord(int *x, int *y)
{
    Point   pt;
    GrafPtr oldPort;


        /* FULL SCREEN MODE */

    if (!gPlayInWindow)
    {
        GetMouse(&pt);

        *x = pt.h;
        *y = pt.v;
    }

        /* WINDOWED MODE */

    else
    {
        GetPort(&oldPort);
        SetPort(gGameWindowGrafPtr);

        GetMouse(&pt);

        SetPort(oldPort);

        *x = pt.h;
        *y = pt.v;
    }
}
```

The call `GetMouse()` will return the mouse coordinates within the current `GrafPort`. If we're in full-screen mode then we just call this and we're done, but if we're playing in a window then we need a few more lines to get the window-relative coordinates. This is done by setting the port to our window before calling `GetMouse()`.

Determining the status of the mouse button is as simple as this:

```
    mouseButtonDown = Button();
```

There is other mouse data that you may want to receive such as mouse delta values, or additional buttons and scroll wheels on complex mice devices. All desktop Mac's ship with that lousy one-button mouse, but there are a lot of people out there who toss that mouse as soon as they unpack their new computer. There are much better mice out there that have scroll wheels and additional buttons. If you're not already using one of those mice, I highly recommend switching to one. Reading the delta values, scroll wheel values, and additional buttons is simply a matter of adding more Carbon Events to our event handler:

Listing 11-7:Including Mouse Events in the Event Handler

```
EventTypeSpec   events[9] =
{
    kEventClassCommand, kEventCommandProcess,
    kEventClassKeyboard, kEventRawKeyDown,
    kEventClassKeyboard, kEventRawKeyUp,
    kEventClassKeyboard, kEventRawKeyModifiersChanged};

    kEventClassMouse, kEventMouseMoved,
    kEventClassMouse, kEventMouseDragged,
    kEventClassMouse, kEventMouseUp,
    kEventClassMouse, kEventMouseDown,
    kEventClassMouse, kEventMouseWheelMoved
};
```

These five new event types let us find out all sorts of great information about the mouse:

`kEventMouseMoved`
This event occurs whenever the mouse moves.

`kEventMouseDragged`
This event occurs whenever the mouse moves while the button is held down.

kEventMouseDown
This event occurs when any button on the mouse is pressed.

kEventMouseUp
This event occurs when any button on the mouse is released.

kEventMouseWheelMoved
This event occurs when the scroll wheel on the mouse is spun.

We need to add some code to `MyEventHandler()` to take care of processing these mouse events:

Listing 11-8:Processing Mouse Events

```
    case    kEventClassMouse:
            switch(eventKind)
            {
                        /* MOUSE MOVED */

                case    kEventMouseMoved:
                case    kEventMouseDragged:

                                        // get mouse delta
                        GetEventParameter(event,
                                        kEventParamMouseDelta,
                                        typeQDPoint, nil,
                                        sizeof(mouseDelta), nil,
                                        &mouseDelta);

                        gMouseDeltaX = mouseDelta.h;
                        gMouseDeltaY = mouseDelta.v;

                                        // get mouse coord
                        GetEventParameter(event,
                                        kEventParamMouseLocation,
                                        typeQDPoint, nil,
                                        sizeof(mouseCoord), nil,
                                        &mouseCoord);

                        gMouseCoordX = mouseCoord.h;
                        gMouseCoordY = mouseCoord.v;
                            break;

                        /* MOUSE WHEEL MOVED */

                case    kEventMouseWheelMoved:

                        GetEventParameter(event,
                                        kEventParamMouseWheelDelta,
                                        typeSInt32, nil,
                                        sizeof(gMouseWheelDelta), nil,
                                        &gMouseWheelDelta);
                        break;


                        /* MOUSE BUTTON DOWN */

                case    kEventMouseDown:
                                        // which button was pressed?

                        GetEventParameter(event,
                                        kEventParamMouseButton,
                                        typeMouseButton,
                                        nil ,sizeof(whichButton), nil,
                                        &whichButton);

                        switch(whichButton)
                        {
                                        // left button?
                            case    kEventMouseButtonPrimary:
                                    gMouseLeftButtonDown = true;
                                    break;

                                        // right button?
                            case    kEventMouseButtonSecondary:
                                    gMouseRightButtonDown = true;
                                    break;

                                        // middle button?
                            case    kEventMouseButtonTertiary:
                                    gMouseMiddleButtonDown = true;
                                    break;
                        }
                        break;


                        /* MOUSE BUTTON UP */

                case    kEventMouseUp:
                                        // which button was released?
                        GetEventParameter(event,
                                        kEventParamMouseButton,
                                        typeMouseButton,
                                        nil ,sizeof(whichButton), nil,
                                        &whichButton);

                        switch(whichButton)
                        {
                                        // left button?
                            case    kEventMouseButtonPrimary:
                                    gMouseLeftButtonDown = false;
```

```
                                break;

                                    // right button?
                        case    kEventMouseButtonSecondary:
                                gMouseRightButtonDown = false;
                                break;

                                    // middle button?
                        case    kEventMouseButtonTertiary:
                                gMouseMiddleButtonDown = false;
                                break;
                    }
                    break;
            }
            break;
```

The function `GetEventParameter()` is used extensively here. When there is a mouse moved or dragged event we call `GetEventParameter()` to get the mouse deltas and coordinates. The same goes for getting the scroll wheel delta.

Technically, OS X can handle mice with up to 32 buttons, but most standard mice have between one and three buttons, so the CarbonEvents.h header file defines constants for the first three buttons:

```
kEventMouseButtonPrimary      = 1,
kEventMouseButtonSecondary    = 2,
kEventMouseButtonTertiary     = 3
```

We use these constants to determine which button generated the mouse up or mouse down event, however, you can also just pass in the numbers 1 through 32 to specify a button. How the buttons from 4 to 32 map to any particular mouse is not defined, but for the first three buttons the primary button is always the left button, the secondary button is always the right one, and the tertiary is always the middle.

Some really funky mice may have more than one scroll wheel, so if you want to support this you can find out which scroll wheel triggered the event with another call to `GetEventParameter()`:

```
EventMouseWheelAxis axis;

GetEventParameter(event, kEventParamMouseWheelAxis,
                typeMouseWheelAxis, nil, sizeof(axis),
                nil, &axis);
```

Two axis constants are defined in CarbonEvents.h:

```
    kEventMouseWheelAxisX         = 0,
    kEventMouseWheelAxisY         = 1
```

It is important to mention that the delta values returned for the mouse and scroll wheel movement are only relative to the last time that a delta event occurred. You're responsible for figuring out what any given delta value really means to your application. Faster machines may be able to detect mouse movement faster, thus, there will be more `kEventMouseMoved` events, and so the delta reported at each event will be smaller. Also, it is very important to understand that your global delta variables will never go to zero because they'll always hold the delta value from the most recent event. If the user stops moving the mouse, no event will be generated, so, the mouse delta variables will still have whatever value was in it previously. The best way to handle this is to process your delta values from inside the event handler function. In other words, if you get a `kEventMouseWheelMoved` event, then read the delta event and immediately process it. If the scroll wheel zooms your camera in your game, then do that zooming now -don't even bother saving that delta in a global variable.

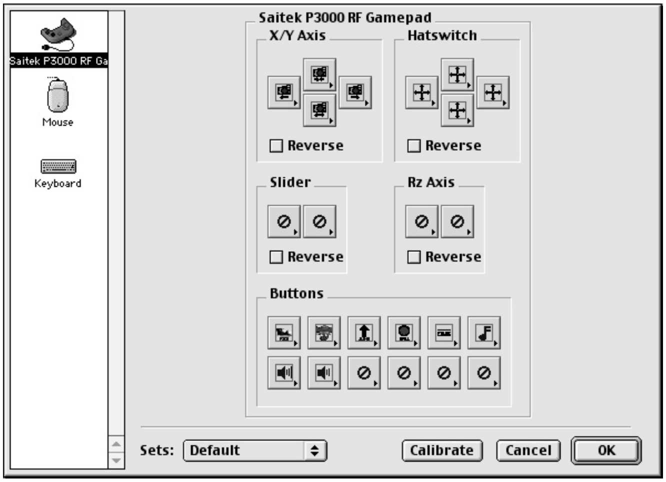
The sample project "Simple Input.xcode" contains all of this sample code and displays mouse deltas and scroll wheel deltas on the screen.

Chapter 12: Input with the HID Manager

Welcome to the special corner in Hell that's been reserved for input devices on OS X. Unfortunately, this is the one part of OS X which has always been a disaster for game programmers, so prepare yourself, it's going to get ugly.

The Good, the Bad, and the HID Manager

I've mentioned before that in the days of Mac OS 8 and 9 there was something called Game Sprockets that was a collection of libraries for doing game related stuff. One of these Sprockets was Input Sprocket. Input Sprocket was a wonderful thing because it made it incredibly easy for a game programmer to support just about any mouse, keyboard, gamepad or joystick out there, and it was 100% reliable. It also had a standard configuration dialog for assigning the controls of keyboards, mice, gamepads, joysticks, etc. Every game had the same dialog, so users immediately understood what to do to set up their devices. The only real issue that anyone ever had with Input Sprocket was that this configuration dialog was rather ugly. Functional, but ugly:



Sets: Default

Figure 12-1: The ugly yet functional Input Sprocket Configuration Dialog

So, rather than just fixing the "ugly" issue with Input Sprocket and moving it over to OS X, Apple decided to kill the technology altogether. For a few years this left us with basically no way of doing input on OS X. The answer to Input Sprocket was supposed to be this new thing called the HID Manager (HID is an acronym for "Human Interface Device"). Unfortunately, it wasn't until Mac OS 10.2 that the HID Manager became usable, but even then, it was so loaded with bugs that using it was difficult.

Saying that using the HID Manager is difficult is something of an understatement because even to this day it's still a nightmare for both game developers and users. The HID Manager constantly loses devices, and gives incorrect information about devices. In some cases this misinformation is the fault of the device itself, but Input Sprocket somehow always handled these devices 100% correctly. The HID Manager's success rate is probably only around 80%. For example, I've yet to find a gamepad who's D-Pad gets correctly recognized by the HID Manager as a D-Pad, and I've only found one type of joystick who's hat switch gets correctly identified as a hat switch. The bottom line is that the data you'll get about devices from the HID Manager is extremely unreliable, but it's all we've got to work with on OS X. This is the main reason why so many games on OS X simply don't support input devices. It's just not worth the trouble, and it generates a huge number of tech support calls from users who cannot get their gamepads to work as they should. If you choose to support input devices on OS X, you can expect to be flooded with emails like this one that I coincidentally just received as I was writing this chapter:

> Dear President of Pangea Software, Inc.:
>
> I am having a problem with my iShock II Gamepad when I play Bugdom 2 on Mac OS X. All of the action buttons (A, B, C, D) make Skip jump. The D-Pad button control Skip (I use them for camera controls). I checked my gamepad configuration dialog but the controls are just the way I set them. The controls worked fine on the day I got the gamepad. The same thing happened with a different game on my computer. I think it is a bug on my computer. It can't be the gamepad because I recently got it. Is there any way to fix this?
>
> Sincerely,
> Seth

Luckily, I've been using the HID Manager for quite some time now, and I've discovered ways to work around many of these bugs with code hacks and physical coercion. For example, I know that sometimes just plugging a device into a different USB port can fix problems like the one Seth was seeing with his gamepad. Other times just rebooting makes devices work again. The file Input.c in the "HID Manager Input.xcode" project contains an entire input system using the HID Manager, and it's whopping 3000 lines of code with lots of special cases to handle the various bugs in the HID Manager. The HID Manager is a low-level API, so, we're essentially writing our own version of Input Sprocket from the ground up.

Some HID Manager Terminology

Before we get to any code, we need to go over the definitions of some terms that the HID Manager uses:

Device

A Device is any HID compliant input device such as a gamepad, keyboard, mouse, joystick, steering wheel, etc.

Element

An Element is generally any control on the Device which causes input. This can be a key on a keyboard, the x-axis of a joystick, the button on a gamepad, etc. These types of Elements are represented by the following constants:

```
kIOHIDElementTypeInput_Misc
kIOHIDElementTypeInput_Button
kIOHIDElementTypeInput_Axis
```

Elements can also be groups or collections of other elements. These types of Elements, `kIOHIDElementTypeCollection`, don't actually have any physical control associated with them, but rather they're just a logical grouping of other controls. This makes things rather confusing when trying to get the list of actual control Elements on a Device as you'll see later.

Usage Page

The Usage Page of a Device is just a weird way to indicate the genre or class of a Device. This is an area where the HID Manager is poorly designed because every gamepad and joystick that I've ever plugged in has come up with the Usage Page of type `kHIDPage_GenericDesktop`. This happens despite the fact that there are other Usage Page types defined in IOHIDUsageTables.h that would seem to be much more game-device-related, including one specifically called `kHIDPage_Game`:

```
kHIDPage_Simulation
kHIDPage_VR
kHIDPage_Sport
kHIDPage_Game
kHIDPage_Arcade
```

Usage

The Usage specifies a sub-type of the Usage Page. There is a huge list of Generic Desktop Usage constants in IOHIDUsageTables.h, but it's the following constants that we'll primarily be using to identify our gamepads, joysticks and such:

```
    kHIDUsage_GD_Mouse
    kHIDUsage_GD_Joystick
    kHIDUsage_GD_GamePad
    kHIDUsage_GD_Keyboard
    kHIDUsage_GD_MultiAxisController
```

Getting the List of HID Devices

The first thing we want to do is get a list of all eligible HID Devices. In our game engine that we're building, we only care about keyboard, gamepad, and joystick devices, so we've got to write an initialization function that creates this list:

Listing 12-1:The Function to Get a List of HID Devices

```
mach_port_t     gHID_MasterPort;

void InitMyHIDManagerStuff(void)
{
    short           i;
    io_iterator_t   hidObjectIterator   = nil;
    IOReturn        ioReturnValue       = kIOReturnSuccess;


            /*******************************/
            /* BUILD A LIST OF HID DEVICES */
            /*******************************/

    gNumHIDDevices = 0;


        /* GET A PORT TO INIT COMMUNICATION WITH I/O KIT */

    ioReturnValue = IOMasterPort(bootstrap_port, &gHID_MasterPort);
    if (ioReturnValue != kIOReturnSuccess)
        DoFatalAlert("\pIOMasterPort failed!");


            /* CREATE AN ITERATION LIST OF HID DEVICES */

    FindHIDDevices(gHID_MasterPort, &hidObjectIterator);
    if (hidObjectIterator == nil)
```

```
        DoFatalAlert("\pMyInitHID:  no HID Devices found!");


            /* PARSE ALL THE HID DEVICES IN THE ITERATION LIST */
            //
            // This builds our master list of desired HID devices and
            // all associated elements.
            //

    ParseAllHIDDevices(hidObjectIterator);


            /* DISPOSE OF THE ITERATION LIST */

    IOObjectRelease(hidObjectIterator);


            /* SET THE DEFAULT CONTROL SETTINGS */

    ResetAllDefaultControls();
}
```

This function starts off by creating a connection to the I/O Kit in the Mac OS X:

```
        IOMasterPort(bootstrap_port, &gHID_MasterPort);
```

The `gHID_MasterPort` reference is then passed to this function:

Listing 12-2:Generating the Device List

```
void FindHIDDevices(mach_port_t masterPort,
                    io_iterator_t *hidObjectIterator)
{
    CFMutableDictionaryRef  hidMatchDictionary;
    IOReturn                    ioReturnValue = kIOReturnSuccess;


        /* CREATE A DICTIONARY OF HID DEVICES */

    hidMatchDictionary = IOServiceMatching(kIOHIDDeviceKey);


        /* FILL THE DICTIONARY WITH THE HID DEVICES */

    ioReturnValue = IOServiceGetMatchingServices(masterPort,
                            hidMatchDictionary, hidObjectIterator);


                /* VERIFY THAT WE FOUND ANYTHING */
    if ((ioReturnValue != kIOReturnSuccess) ||
        (*hidObjectIterator == nil))
    {
        DoFatalAlert("\pFindHIDDevices: No matching HID devices
                    found!");
    }
}
```

The function `IOServiceMatching()` essentially creates a blank `CFDictionary` to contain HID Device references:

```
    hidMatchDictionary = IOServiceMatching(kIOHIDDeviceKey);
```

Next, we fill that Dictionary with the list of HID devices:

```
    ioReturnValue = IOServiceGetMatchingServices(masterPort,
                                hidMatchDictionary, hidObjectIterator);
```

You'll notice that we check both the `ioReturnValue` and `hidObjectIterator` to make sure we actually got something. You'd be amazed how often this fails because the HID Manager will suddenly stop functioning, and won't find your mouse or keyboard or anything else plugged in. If this happens, the only solution is to reboot your computer. I get customers emailing me with support questions about this all the time, so when it happens I just tell them that everything should work after a reboot. However, rebooting *does not always* fix the problem. Sometimes you have to physically unplug the keyboard, and plug it into a different USB port before the device will be seen again. A customer of mine had this happen just the other day, and he was lucky because the USB swapping worked for him. But there are times when rebooting and swapping USB ports still doesn't work. I have yet to find a solution to this rare case, but users who experience this have reported that later in the day or the next day, the devices will start to be seen by the HID again.

Ok, we've now got the master list of all HID devices available on this computer, so now we need to scan this list, and keep only the Devices that we want our game to use:

Listing 12-3:Parsing the Device List

```
void ParseAllHIDDevices(io_iterator_t hidObjectIterator)
{
    io_object_t             hidDevice;
    IOReturn                ioReturnValue;
    kern_return_t           result;
    CFMutableDictionaryRef  properties;
    CFTypeRef               object;
    long                    usagePage, usage, locationID;
    Boolean                 allowThisDevice;
    /*****************************************************/
    /* ITERATE THRU ALL OF THE HID OBJECTS IN THE LIST */
    /*****************************************************/

        /* GET THE HID DEVICE IN THE WHILE LOOP */

while ((hidDevice = IOIteratorNext(hidObjectIterator)))
{
    /* CONVERT THE DEVICES PROPERTIES INTO A CF DICTIONARY */

    result = IORegistryEntryCreateCFProperties(hidDevice,
                        &properties,
                        kCFAllocatorDefault,
                        kNilOptions);

    if ((result != KERN_SUCCESS) || (properties == nil))
        continue;


        /* GET THE “USAGE PAGE” OF THIS HID DEVICE */

    object = CFDictionaryGetValue(properties,
                                CFSTR(kIOHIDPrimaryUsagePageKey));
    if (!object)
    {
            // on 10.3 this is probably a legit error, but on 10.2 it's
            // probably the keyboard

        if (gPantherOrBetter)
            goto error;
    }
    else
    {
        if (!CFNumberGetValue(object, kCFNumberLongType, &usagePage))
        {
            if (gPantherOrBetter)
                goto error;
            else
                usagePage = kHIDPage_GenericDesktop;
        }
    }

        /* GET THE “USAGE” OF THIS DEVICE */

    object = CFDictionaryGetValue(properties,
                                    CFSTR(kIOHIDPrimaryUsageKey));
    if (!object)
    {
        if (gPantherOrBetter)
            goto error;
    }
```

```
else
{
    if (!CFNumberGetValue(object, kCFNumberLongType, &usage))
    {
        if (gPantherOrBetter)
            goto error;
        else
            usage = kHIDUsage_GD_Keyboard;
    }
}


        /* GET LOCATION ID OF THE USB DEVICE */

object = CFDictionaryGetValue(properties,
                             CFSTR(kIOHIDLocationIDKey));
if (!object)
    locationID = 0xdeadbeef;
else
{
    if (!CFNumberGetValue(object, kCFNumberLongType, &locationID))
    {
        locationID = 0xdeadbeef;
    }
}

        /**********************************************/
        /* SEE IF THIS IS A DEVICE WE'RE INTERESTED IN */
        /**********************************************/

allowThisDevice = false;        // assume we don't want this device

switch(usagePage)
{
    case    kHIDPage_GenericDesktop:
            switch(usage)
            {
                case    kHIDUsage_GD_Joystick:
                case    kHIDUsage_GD_GamePad:
                case    kHIDUsage_GD_Keyboard:
                        allowThisDevice = true;
                        break;
            }
            break;
}

        /***********************************/
        /* ADD THIS HID DEVICE TO OUR LIST */
        /***********************************/

if (allowThisDevice)
{
```

```
            AddHIDDeviceToList(hidDevice, properties, usagePage,
                               usage, locationID);
        }


            /* RELEASE THE HID OBJECT */
error:
        CFRelease(properties);          // free the CF dictionary
        IOObjectRelease(hidDevice);
    }
}
```

This function is riddled with error checking and hacks to handle the bugs in older versions of the HID Manager on OS 10.2.8 and earlier. You'll notice that we check the variable `gPantherOrBetter` quite often. This variable is set in our application's initialization function, and it is set to `true` if we're running on Mac OS 10.3 or later. This is because many HID Manager bugs were fixed in 10.3, but we still have to work around these bugs in 10.2.

To scan through our list of Devices we use the `IOIteratorNext()` function which returns a device reference in the form of an `io_object_t`. With this device reference we can easily generate a Core Foundation Dictionary that contains a list of all of the parameters describing the device:

```
    IORegistryEntryCreateCFProperties(hidDevice,
                        &properties, kCFAllocatorDefault, kNilOptions);
```

It's really easy to extract all sorts of information about each device by making calls to `CFDictionaryGetValue()`. The first thing we want to do is get the Usage Page of this device:

```
    object = CFDictionaryGetValue(properties,
                                CFSTR(kIOHIDPrimaryUsagePageKey));
```

The value `kIOHIDPrimaryUsagePageKey` is defined in IOHIDKeys.h along with lots of other "keys" that you can use to extract information:

```
    kIOHIDTransportKey
    kIOHIDVendorIDKey
    kIOHIDVendorIDSourceKey
    kIOHIDProductIDKey
    kIOHIDVersionNumberKey
    kIOHIDManufacturerKey
    kIOHIDProductKey
    kIOHIDSerialNumberKey
    kIOHIDLocationIDKey
    kIOHIDDeviceUsageKey
    kIOHIDDeviceUsagePageKey
    kIOHIDDeviceUsagePairsKey
    kIOHIDPrimaryUsageKey
    kIOHIDPrimaryUsagePageKey
    kIOHIDMaxInputReportSizeKey
    kIOHIDMaxOutputReportSizeKey
    kIOHIDMaxFeatureReportSizeKey
```

If the value returned is `nil` then that means that `CFDictionaryGetValue()` was unable to find that key anywhere in the device's Dictionary. Unfortunately, there was a major bug in the version of the HID Manager prior to Mac OS 10.3 where the keyboard device might be missing all or some of its standard key values, including the critical information such as the Usage Page and Usage – you know, that little thing that helps us determine what the device actually is.

If we're on Mac OS 10.2 and we get an error from one of these `CFDictionaryGetValue()` calls, it's safe to guess that this device is actually a keyboard. There's no way to know for sure, but it probably is. So, when an error occurs we just fake it with a hack to force the Usage Page and Usage to `kHIDPage_GenericDesktop` and `kHIDUsage_GD_Keyboard` respectively.

Another piece of information that we extract from the Device is its Location ID via `kIOHIDLocationIDKey`. The Location ID uniquely identifies this device's location on the USB chain. The reason this value is important is because the user might have two or more totally identical devices plugged in, and we need a way to differentiate between them. For example, I have two identical Saitek gamepads plugged in. They both have 100% identical information such as Page Usage, Page, Device Name, Manufacturer Name, etc. The only way to tell them apart is by their location ID. This is useful when saving and restoring user settings for the different devices.

Once we've gotten the Usage Page and Usage values, we check them to see if they're what we're looking for. In our engine we're only looking for Generic Desktop devices which are keyboards, gamepads, or joysticks. If the device meets our criteria then we add it to our list of devices that we're interested in. Otherwise, we skip it.

Listing 12-4:Add a Device to our List

```
static void AddHIDDeviceToList(io_object_t hidDevice,
                            CFMutableDictionaryRef properties,
                            long usagePage,
                            long usage, long locationID)
{
    CFTypeRef   object;
```

```
const char  *deviceName;
const char  defaultName[] = "Unnamed Device";
short       d;
long        vendorID, productID;

if (gNumHIDDevices >= MAX_HID_DEVICES)
    return;

d = gNumHIDDevices;


    /*********************************/
    /* GATHER SOME INFO THAT WE NEED */
    /*********************************/

        /* GET PRODUCT NAME */

object = CFDictionaryGetValue(properties, CFSTR(kIOHIDProductKey));
if (!object)
    deviceName = defaultName;
else
    deviceName = GetCFStringFromObject(object);


        /* GET VENDOR & PRODUCT ID'S */

object = CFDictionaryGetValue(properties, CFSTR(kIOHIDVendorIDKey));
if (!object)
    vendorID = d;                           // set some bogus vendorID
else
if (!CFNumberGetValue(object, kCFNumberLongType, &vendorID))
    vendorID = d;                           // set some bogus vendorID

object = CFDictionaryGetValue(properties, CFSTR(kIOHIDProductIDKey));
if (!object)
    productID = d;                          // set some bogus product ID
else
if (!CFNumberGetValue(object, kCFNumberLongType, &productID))
    productID = d;                          // set some bogus product ID


        /************************/
        /* SAVE INFO INTO LIST */
        /************************/

gHIDDeviceList[d].usagePage     = usagePage;    // keep usage Page
gHIDDeviceList[d].usage         = usage;        // keep usage
gHIDDeviceList[d].vendorID  = vendorID;         // keep vendor ID
gHIDDeviceList[d].productID = productID;        // keep product ID
gHIDDeviceList[d].locationID    = locationID;   // keep location ID
gHIDDeviceList[d].numElements    = 0;           // no elements yet
```

```
    strncpy(gHIDDeviceList[d].deviceName,       // copy device name string
            deviceName,
            DEVICE_NAME_MAX_LENGTH);

            /* ONLY KEYBOARDS ARE ACTIVE BY DEFAULT */

    if  ((usagePage == kHIDPage_GenericDesktop) &&
        (usage == kHIDUsage_GD_Keyboard))
        gHIDDeviceList[d].isActive = true;
    else
        gHIDDeviceList[d].isActive = false;


            /****************************/
            /* RECURSIVELY ADD ELEMENTS */
            /****************************/

    RecurseDictionaryElement(properties, CFSTR(kIOHIDElementKey));



                /* OPEN AN INTERFACE TO THIS DEVICE */

    MyOpenHIDDeviceInterface(hidDevice,
                        &gHIDDeviceList[d].hidDeviceInterface);

    gNumHIDDevices++;
}
```

Once again, this function is loaded with hacks to work around the various bugs in the HID Manager. The function starts out by getting the name of the device:

```
    object = CFDictionaryGetValue(properties, CFSTR(kIOHIDProductKey));
```

Unfortunately, there will be random times when the name string simply isn’t in the device’s property list. It’s rare, but it does happen. Also, some older USB devices don’t have product names in their ROM’s, so this will also fail in those cases. Such a device is the Ariston Ares joystick. So, if we do get an error trying to extract the name string from the device, then we just set it to some default value like “Unnamed Device.”

Next, we get some additional information about the device that is useful for identifying it when we save and restore the user’s configuration settings. However, these values will randomly fail, especially in older versions of the HID Manager. The Vendor ID and Product ID values seem to be especially prone to failure if the device happens to a PowerBook keyboard. So, we check for these failures, and if one occurs we just set the ID’s to some arbitrary value.

After saving all of the device's information into our `gHIDDeviceList` array, the next step is to recursively scan the device for all of its control Elements by calling our `RecurseDictionaryElements()` function. We'll discuss this in a moment, but for now lets look at the next line of code from Listing 12-4:

```
    CreateHIDDeviceInterface(hidDevice,
                            &gHIDDeviceList[d].hidDeviceInterface);
```

Creating an interface to the device is a fancy way of saying that we're simply connecting to that device, thus, making it available to our application. In more complex terms, the device interface provides jump pointers to functions that your application can use to access it.

Listing 12-5:Creating and Opening an Interface to the Device

```
static void MyOpenHIDDeviceInterface(io_object_t hidDevice,
                                IOHIDDeviceInterface ***hidDeviceInterface)
{
    IOCFPlugInInterface     **plugInInterface;
    HRESULT                 plugInResult;
    SInt32                  score = 0;
    IOReturn                ioReturnValue;

            /* CREATE AN INTERMEDIATE INTERFACE TO THE DEVICE */

    ioReturnValue = IOCreatePlugInInterfaceForService(hidDevice,
                                    kIOHIDDeviceUserClientTypeID,
                                    kIOCFPlugInInterfaceID,
                                    &plugInInterface, &score);
    if (ioReturnValue != kIOReturnSuccess)
        DoFatalAlert("\pIOCreatePlugInInterfaceForService failed!");


    /* QUERY THE INTERMEDIATE INTERFACE TO GET THE ACTUAL INTERFACE */

    plugInResult = (*plugInInterface)->QueryInterface(plugInInterface,
                        CFUUIDGetUUIDBytes(kIOHIDDeviceInterfaceID),
                        (LPVOID)hidDeviceInterface);
    if (plugInResult != S_OK)
        DoFatalAlert("\pCouldn't create HID class device interface");


                /* RELEASE THE INTERMEDIATE INTERFACE */

    (*plugInInterface)->Release(plugInInterface);


            /* OPEN THE INTERFACE */
    ioReturnValue = (**hidDeviceInterface)->open(*hidDeviceInterface, 0);
    if (ioReturnValue != kIOReturnSuccess)
        DoFatalAlert("\pCouldn’t open device interface");
}
```

This function is perhaps the most cryptic function in our entire input engine, and frankly it’s not important that you understand what it’s doing. Just know that it’s opening up a connection to the Device so that we can read data from it later.

Getting a Device’s Elements

Ok, back to getting our Device’s Element list. The database of Elements in a Device is more complicated than it probably needs to be. It’s not just a simple list of Elements, no, that would be too easy. Instead it’s a series of groups of Elements in a hierarchy of data that includes arrays of sub-elements. In other words… it’s a mess. The diagram below shows a very basic example of such a hierarchy:

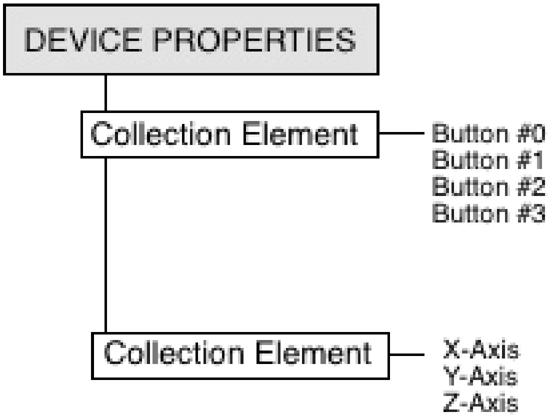


Figure 12-2: Hierarchy of a fictional HID Device’s Elements

To extract a list of elements from the device we’ve got to parse through it recursively:

Listing 12-6:Recursing Through a Device’s Elements

```
void RecurseDictionaryElement(CFDictionaryRef dictionary,
                                CFStringRef key)
{
    CFTypeRef   object;
    CFTypeID    type;

            // get the desired key value from the dictionary

    object = CFDictionaryGetValue(dictionary, key);
    if (object)
    {
        type = CFGetTypeID(object);         // get the type of element

        if (type == CFArrayGetTypeID())     // it's an array of elements
            MyCFArrayParse(object);
        else
        if (type == CFDictionaryGetTypeID())    // it's a sub-dictionary
            VerifyAndAddHIDElement(object);
    }
}
```

One of two things happens in this function: If the Element passed in is an Array then we need to parse that array for all of the Elements inside it. Otherwise, if the Element is a Dictionary, then we try to add that Element to our list of Elements for the Device. The physical control Elements that we’re looking for are always Dictionary Elements.

Here’s how we parse an Array Element:

Listing 12-7:Parsing an Array of Elements

```
void MyCFArrayParse(CFArrayRef object)
{
    CFRange range;

    range.location = 0;
    range.length = CFArrayGetCount(object);

    CFArrayApplyFunction(object, range, MyCFArrayCallback, 0);
}

void MyCFArrayCallback (const void * object, void * parameter)
{
    if (CFGetTypeID(object) == CFDictionaryGetTypeID())
    VerifyAndAddHIDElement(object);
}
```

Core Foundation does the parsing of the array somewhat automatically for us. We simply pass it the array object, the number of elements in the array to parse, and a pointer to a callback function. Then Core Foundation will parse the array for us, and call the `MyCFArrayCallback()` function for each element it finds in the array. Since we know that only Dictionary elements are actual input device controls, we toss out anything else, but if we come across a Dictionary we try to add it to our list.

Listing 12-8:Verifying an Element and adding it to our List

```
static void VerifyAndAddHIDElement(CFDictionaryRef dictionary)
{
    CFTypeRef                object;
    IOHIDElementCookie       cookie;
    long                     elementType, usagePage, usage;
    long                     min,max,scaledMin,scaledMax;
    const char               *elementName;
    short                    d, e;

        /*****************************************************/
        /* FIRST DETERMINE IF THIS IS AN ELEMENT WE CARE ABOUT */
        /*****************************************************/

                /* GET THE TYPE OF THIS ELEMENT */

    object = CFDictionaryGetValue(dictionary, CFSTR(kIOHIDElementTypeKey));
    if (!object)
        goto skip_element;

    elementType = GetCFNumberFromObject(object);


                /* SKIP ANY TYPES THAT WE DON’T CARE ABOUT */

    switch(elementType)
    {
        case    kIOHIDElementTypeInput_Misc:
        case    kIOHIDElementTypeInput_Button:
        case    kIOHIDElementTypeInput_Axis:
                break;

        default:
                goto skip_element;
    }


        /***********************************************************/
        /* THIS IS AN ELEMENT WE LIKE, SO EXTRACT THE IMPORTANT DATA */
        /***********************************************************/
        /* GET THE ELEMENT'S COOKIE */

object = CFDictionaryGetValue(dictionary,
                            CFSTR(kIOHIDElementCookieKey));
cookie = (IOHIDElementCookie)GetCFNumberFromObject(object);

            /* GET ELEMENT’S USAGE PAGE */

object = CFDictionaryGetValue(dictionary,
                            CFSTR(kIOHIDElementUsagePageKey));
usagePage = GetCFNumberFromObject(object);

            /* THROW OUT CERTAIN ONES */

if (usagePage == kHIDPage_PID)
    goto skip_element;
if (usagePage == kHIDPage_LEDs)
    goto skip_element;


        /* GET ELEMENT’S USAGE */

object = CFDictionaryGetValue(dictionary, CFSTR(kIOHIDElementUsageKey));
usage = GetCFNumberFromObject(object);


        /* GET MIN/MAX VALUES OF THIS CONTROL */

object = CFDictionaryGetValue(dictionary, CFSTR(kIOHIDElementMinKey));
min = GetCFNumberFromObject(object);

object = CFDictionaryGetValue(dictionary, CFSTR(kIOHIDElementMaxKey));
max = GetCFNumberFromObject(object);

object = CFDictionaryGetValue(dictionary,
                            CFSTR(kIOHIDElementScaledMinKey));
scaledMin = GetCFNumberFromObject(object);

object = CFDictionaryGetValue(dictionary,
                            CFSTR(kIOHIDElementScaledMaxKey));
scaledMax = GetCFNumberFromObject(object);


        /* GET NAME STRING */

object = CFDictionaryGetValue(dictionary, CFSTR(kIOHIDElementNameKey));
if (object)
    elementName = GetCFStringFromObject(object);

if ((object == nil) || (elementName == nil))
{
        elementName = CreateElementNameString(usagePage, usage);
        if (elementName == nil)
            goto skip_element;
    }

        /*********************/
        /* SAVE ELEMENT INFO */
        /*********************/

    d = gNumHIDDevices;
    e = gHIDDeviceList[d].numElements;

    if (e >= MAX_HID_ELEMENTS)
        goto skip_element;

    gHIDDeviceList[d].elements[e].elementType   =   elementType;
    gHIDDeviceList[d].elements[e].cookie            =   cookie;
    gHIDDeviceList[d].elements[e].usagePage     =   usagePage;
    gHIDDeviceList[d].elements[e].usage         =   usage;
    gHIDDeviceList[d].elements[e].min           =   min;
    gHIDDeviceList[d].elements[e].max           =   max;
    gHIDDeviceList[d].elements[e].scaledMin     =   scaledMin;
    gHIDDeviceList[d].elements[e].scaledMax     =   scaledMax;

    strncpy(gHIDDeviceList[d].elements[e].name, elementName,
            ELEMENT_NAME_MAX_LENGTH);           // copy device name string


                /* SET DEFAULT CALIBRATION VALUES */

    SetElementDefaultCalibration (d, e);


    gHIDDeviceList[d].numElements++;


            /**********************************/
            /* TRY TO SUB-RECURSE THE ELEMENT */
            /**********************************/

skip_element:
    RecurseDictionaryElement(dictionary, CFSTR(kIOHIDElementKey));
}
```

That's a pretty huge chunk of code, so here goes the explanation. Before we can add the Element to our list, we've got to make sure it's an element that we care about, so we get the type out of the Dictionary:

```
    object = CFDictionaryGetValue(dictionary, CFSTR(kIOHIDElementTypeKey));
```

Then we check it to see if it's something that we want to keep or toss:

```
switch(elementType)
{
    case    kIOHIDElementTypeInput_Misc:
    case    kIOHIDElementTypeInput_Button:
    case    kIOHIDElementTypeInput_Axis:
            break;

    default:
            goto skip_element;
}
```

Obviously we want to keep all Button and Axis control elements, but we also want to keep the Misc types too because those tend to be joystick hat switches.

Next we gather all sorts of information about the Element in a similar manner to how we gathered information about Devices earlier. And just like Devices, Elements also have their own Usage Page and Usage values. The Usage Page for an Element identifies the specific type of control that it is. The ones we care about are:

```
kHIDPage_GenericDesktop
kHIDPage_KeyboardOrKeypad
kHIDPage_Button
```

For unknown reasons, the HID gods didn't add a Usage Page type to identify axes, sliders, hat switches, start buttons, D-Pads, etc. Instead they just grouped all of those into the `kHIDPage_GenericDesktop` which doesn't really make any sense (welcome to the HID Manager).

Anyway, there are a few Usage Page types that we need to specifically look for and eliminate:

```
if (usagePage == kHIDPage_PID)
    goto skip_element;
if (usagePage == kHIDPage_LEDs)
    goto skip_element;
```

These tend to come up for keyboard devices and cause all sorts of problems if you don't skip them. Yes, even the LED's on your keyboard are considered HID elements, and if you don't toss them here they'll appear to be keys.

Now we gather some additional information that tells us about the minimum and maximum values of the control. Different joysticks, for example, will have different ranges of their axis values. Some may go from 0 to 255 as you move the joystick left to right, while higher-end ones may go from –1024 to +1024. So, it's important to know the range so that we can calibrate the controls, and scale them to numbers our game can use later.

The next step is important, and it is one of the big headaches of the HID Manager. That step is the act of trying to figure out the name of the Element.

```
    object = CFDictionaryGetValue(dictionary, CFSTR(kIOHIDElementNameKey));
```

Only in *very rare* cases does the HID Manager successfully return the names of the controls on any given device, so, we've got to manually assign them in a new function, `CreateElementNameString()`. We pass the Usage Page and Usage values to this function, and based on those values it will return a string containing the name to give the control.

Listing 12-9:Naming a Control Element

```
const char *CreateElementNameString(long usagePage, long usage)
{
    const char *c = "Unnamed Element";
    const char *buttonNames[30] =
    {
    "Button 1",     "Button 2",     "Button 3",     "Button 4",
    "Button 5",     "Button 6",     "Button 7",     "Button 8",
    "Button 9",     "Button 10",    "Button 11",    "Button 12",
    "Button 13",    "Button 14",    "Button 15",    "Button 16",
    "Button 17",    "Button 18",    "Button 19",    "Button 20",
    "Button 21",    "Button 22",    "Button 23",    "Button 24",
    "Button 25",    "Button 26",    "Button 27",    "Button 28",
    "Button 29",    "Button 30",
    };

    switch(usagePage)
    {
            /**************************/
            /* GENERIC DESKTOP DEVICE */
            /**************************/

        case    kHIDPage_GenericDesktop:
                switch(usage)
                {
                    case    kHIDUsage_GD_X:
                            c = "X-Axis";
                            break;

                    case    kHIDUsage_GD_Y:
```

```
        c = "Y-Axis";
        break;

case    kHIDUsage_GD_Z:
        c = "Z-Axis";
        break;

case    kHIDUsage_GD_Rx:
        c = "Rotate X-Axis";
        break;

case    kHIDUsage_GD_Ry:
        c = "Rotate Y-Axis";
        break;

case    kHIDUsage_GD_Rz:
        c = "Rotate Z-Axis";
        break;

case    kHIDUsage_GD_Slider:
        c = "Slider";
        break;

case    kHIDUsage_GD_Dial:
        c = "Dial";
        break;

case    kHIDUsage_GD_Wheel:
        c = "Wheel";
        break;

case    kHIDUsage_GD_Hatswitch:
        c = "Hat Switch";
        break;

case    kHIDUsage_GD_Start:
        c = "Start";
        break;

case    kHIDUsage_GD_Select:
        c = "Select";
        break;

case    kHIDUsage_GD_DPadUp:
        c = "D-Pad Up";
        break;

case    kHIDUsage_GD_DPadDown:
        c = "D-Pad Down";
        break;

case    kHIDUsage_GD_DPadRight:
```

```
                    c = "D-Pad Right";
                    break;

                case    kHIDUsage_GD_DPadLeft:
                        c = "D-Pad Left";
                        break;

            }
            break;


        /***********/
        /* BUTTONS */
        /***********/

    case    kHIDPage_Button:
            if (usage < 30)
                c = buttonNames[usage-1];
            else
                c = "Button";
            break;

        /*******************/
        /* KEYBOARD DEVICE */
        /*******************/

    case    kHIDPage_KeyboardOrKeypad:

            switch(usage)
            {
                case    kHIDUsage_KeyboardA:
                        c = "A";
                        break;
                case    kHIDUsage_KeyboardB:
                        c = "B";
                        break;
                case    kHIDUsage_KeyboardC:
                        c = "C";
                        break;
                case    kHIDUsage_KeyboardD:
                        c = "D";
                        break;
                case    kHIDUsage_KeyboardE:
                        c = "E";
                        break;
                case    kHIDUsage_KeyboardF:
                        c = "F";
                        break;
                case    kHIDUsage_KeyboardG:
                        c = "G";
                        break;
                case    kHIDUsage_KeyboardH:
```

```
        c = "H";
        break;
case    kHIDUsage_KeyboardI:
        c = "I";
        break;
case    kHIDUsage_KeyboardJ:
        c = "J";
        break;
case    kHIDUsage_KeyboardK:
        c = "K";
        break;
case    kHIDUsage_KeyboardL:
        c = "L";
        break;
case    kHIDUsage_KeyboardM:
        c = "M";
        break;
case    kHIDUsage_KeyboardN:
        c = "N";
        break;
case    kHIDUsage_KeyboardO:
        c = "O";
        break;
case    kHIDUsage_KeyboardP:
        c = "P";
        break;
case    kHIDUsage_KeyboardQ:
        c = "Q";
        break;
case    kHIDUsage_KeyboardR:
        c = "R";
        break;
case    kHIDUsage_KeyboardS:
        c = "S";
        break;
case    kHIDUsage_KeyboardT:
        c = "T";
        break;
case    kHIDUsage_KeyboardU:
        c = "U";
        break;
case    kHIDUsage_KeyboardV:
        c = "V";
        break;
case    kHIDUsage_KeyboardW:
        c = "W";
        break;
case    kHIDUsage_KeyboardX:
        c = "X";
        break;
case    kHIDUsage_KeyboardY:
        c = "Y";
```

```
                break;
        case    kHIDUsage_KeyboardZ:
                c = "Z";
                break;

        case    kHIDUsage_Keyboard1:
                c = "1";
                break;
        case    kHIDUsage_Keyboard2:
                c = "2";
                break;
        case    kHIDUsage_Keyboard3:
                c = "3";
                break;
        case    kHIDUsage_Keyboard4:
                c = "4";
                break;
        case    kHIDUsage_Keyboard5:
                c = "5";
                break;
        case    kHIDUsage_Keyboard6:
                c = "6";
                break;
        case    kHIDUsage_Keyboard7:
                c = "7";
                break;
        case    kHIDUsage_Keyboard8:
                c = "8";
                break;
        case    kHIDUsage_Keyboard9:
                c = "9";
                break;
        case    kHIDUsage_Keyboard0:
                c = "0";
                break;

        case    kHIDUsage_KeyboardReturnOrEnter:
                c = "Return";
                break;
        case    kHIDUsage_KeyboardEscape:
                c = "ESC";
                break;
        case    kHIDUsage_KeyboardDeleteOrBackspace:
                c = "Delete";
                break;
        case    kHIDUsage_KeyboardTab:
                c = "Tab";
                break;
        case    kHIDUsage_KeyboardSpacebar:
                c = "Spacebar";
                break;
        case    kHIDUsage_KeyboardHyphen:
```

```
        c = " - ";
        break;
case    kHIDUsage_KeyboardEqualSign:
        c = "=";
        break;
case    kHIDUsage_KeyboardOpenBracket:
        c = "[";
        break;
case    kHIDUsage_KeyboardCloseBracket:
        c = "]";
        break;

case    kHIDUsage_KeyboardBackslash:
        c = "Backslash";
        break;
case    kHIDUsage_KeyboardNonUSPound:
        c = "Pound";
        break;
case    kHIDUsage_KeyboardSemicolon:
        c = ";";
        break;
case    kHIDUsage_KeyboardQuote:
        c = "Quote";
        break;
case    kHIDUsage_KeyboardGraveAccentAndTilde:
        c = "~ (tilde)";
        break;

case    kHIDUsage_KeyboardComma:
        c = ",";
        break;
case    kHIDUsage_KeyboardPeriod:
        c = ".";
        break;
case    kHIDUsage_KeyboardSlash:
        c = "/";
        break;
case    kHIDUsage_KeyboardCapsLock:
        c = "CAPSLOCK";
        break;

case    kHIDUsage_KeyboardF1:
        c = "F1";
        break;
case    kHIDUsage_KeyboardF2:
        c = "F2";
        break;
case    kHIDUsage_KeyboardF3:
        c = "F3";
        break;
case    kHIDUsage_KeyboardF4:
        c = "F4";
```

```
                break;
        case    kHIDUsage_KeyboardF5:
                c = "F5";
                break;
        case    kHIDUsage_KeyboardF6:
                c = "F6";
                break;
        case    kHIDUsage_KeyboardF7:
                c = "F7";
                break;
        case    kHIDUsage_KeyboardF8:
                c = "F8";
                break;
        case    kHIDUsage_KeyboardF9:
                c = "F9";
                break;
        case    kHIDUsage_KeyboardF10:
                c = "F10";
                break;
        case    kHIDUsage_KeyboardF11:
                c = "F11";
                break;
        case    kHIDUsage_KeyboardF12:
                c = "F12";
                break;
        case    kHIDUsage_KeyboardF13:
        case    kHIDUsage_KeyboardPrintScreen:
                c = "F13";
                break;
        case    kHIDUsage_KeyboardF14:
        case    kHIDUsage_KeyboardScrollLock:
                c = "F14";
                break;
        case    kHIDUsage_KeyboardF15:
        case    kHIDUsage_KeyboardPause:
                c = "F15";
                break;

        case    kHIDUsage_KeyboardInsert:
                c = "Insert";
                break;
        case    kHIDUsage_KeyboardHome:
                c = "Home";
                break;
        case    kHIDUsage_KeyboardPageUp:
                c = "Page Up";
                break;
        case    kHIDUsage_KeyboardDeleteForward:
                c = "Del";
                break;
        case    kHIDUsage_KeyboardEnd:
                c = "End";
```

```
                    break;
            case    kHIDUsage_KeyboardPageDown:
                    c = "Page Down";
                    break;

            case    kHIDUsage_KeyboardRightArrow:
                    c = "Right Arrow";
                    break;
            case    kHIDUsage_KeyboardLeftArrow:
                    c = "Left Arrow";
                    break;
            case    kHIDUsage_KeyboardDownArrow:
                    c = "Down Arrow";
                    break;

// ** for some reason kHIDUsage_KeyboardRightControl appears to really
// be the down arrow, but only on 10.2.6, not 10.3

            case    kHIDUsage_KeyboardRightControl:
                    if (gPantherOrBetter)
                        c = nil;
                    else
                        c = "Down Arrow";
                    break;

            case    kHIDUsage_KeyboardUpArrow:
                    c = "Up Arrow";
                    break;

            case    kHIDUsage_KeypadNumLock:
                    c = "Num Lock / Clear";
                    break;
            case    kHIDUsage_KeypadSlash:
                    c = "Keypad Slash /";
                    break;
            case    kHIDUsage_KeypadAsterisk:
                    c = "Keypad Asterisk *";
                    break;
            case    kHIDUsage_KeypadHyphen:
                    c = "Keypad Hyphen -";
                    break;
            case    kHIDUsage_KeypadPlus:
                    c = "Keypad Plus +";
                    break;
            case    kHIDUsage_KeypadEnter:
                    c = "Keypad Enter";
                    break;

            case    kHIDUsage_Keypad1:
                    c = "Keypad 1";
                    break;
            case    kHIDUsage_Keypad2:
                    c = "Keypad 2";
                    break;
            case    kHIDUsage_Keypad3:
                    c = "Keypad 3";
                    break;
            case    kHIDUsage_Keypad4:
                    c = "Keypad 4";
                    break;
            case    kHIDUsage_Keypad5:
                    c = "Keypad 5";
                    break;
            case    kHIDUsage_Keypad6:
                    c = "Keypad 6";
                    break;
            case    kHIDUsage_Keypad7:
                    c = "Keypad 7";
                    break;
            case    kHIDUsage_Keypad8:
                    c = "Keypad 8";
                    break;
            case    kHIDUsage_Keypad9:
                    c = "Keypad 9";
                    break;
            case    kHIDUsage_Keypad0:
                    c = "Keypad 0";
                    break;

            case    kHIDUsage_KeypadPeriod:
                    c = "Keypad Period .";
                    break;
            case    kHIDUsage_KeyboardNonUSBackslash:
                    c = "Keypad Backslash";
                    break;

            case    kHIDUsage_KeypadEqualSign:
                    c = "Keypad Equal =";
                    break;
            case    kHIDUsage_KeyboardHelp:
                    c = "Help";
                    break;
            case    kHIDUsage_KeypadComma:
                    c = "Keypad Comma ,";
                    break;

            case    kHIDUsage_KeyboardReturn:
                    c = "Return";
                    break;

            case    kHIDUsage_KeyboardLeftControl:
                    c = "CTRL";
                    break;
            case    kHIDUsage_KeyboardLeftShift:
```

```
                    c = "Shift";
                    break;
            case    kHIDUsage_KeyboardLeftAlt:
                    c = "Option";
                    break;
            case    kHIDUsage_KeyboardLeftGUI:
                    c = "Apple/Command";
                    break;

            default:
                    c = nil;        // key not supported so pass
                                    // back nil so that we skip the
                                    // element
        }
        break;

    }

    return(c);
}
```

Wow, that function was even more gigantic than the previous one, eh? It may be big, but it doesn't really do anything magical. It's just a giant `switch` statement that sets the Element's name string based on its `usagePage` and `usage` variables.

At the top of the function we have a large table of button names used to name the buttons on a device:

```
    const char *buttonNames[30] =
    {
    "Button 1",     "Button 2",     "Button 3",     "Button 4",
    "Button 5",     "Button 6",     "Button 7",     "Button 8",
    ...
    ...
```

We use this to assign names to any buttons on a device that don't report their own name string to the HID Manager. This is unfortunate, because the user is not going to have any idea which button on his gamepad is the one named "Button 1". If the HID Manager were smart like Input Sprocket, then it would return the correct name strings like "Red Button", or "C Button" to correctly identify it. This is one of the many fundamental problems in trying to support these devices on OS X. Even Apple's own devices like the keyboard don't even return key names! That's what most of `CreateElementNameString()` is doing – it's manually naming every kind of key that we have a `case` for. Good luck supporting non-English keyboards! This is a disaster.

The only way around this naming problem is to go out and buy every kind of input device on the market, plug them in, and run your HID code to find out which button or key does what. Then write a massive table into your code that assigns the correct name string to the various devices. Since you're not likely to go to that much trouble, we'll stick with just using the generic button names.

Another thing to note in our `CreateElementNameString()` function is that there is a hack in there to fix a HID Manager keyboard bug:

```
case    kHIDUsage_KeyboardRightControl:
        if (gPantherOrBetter)
            c = nil;
        else
            c = "Down Arrow";
        break;
```

Prior to Mac OS 10.3 (Panther) there was a bug that would cause the Down Arrow key to show up as the Right Control key on some systems. Even though the usage value is `kHIDUsage_KeyboardRightControl`, that element is actually the down arrow key! There would be another element with the usage value of `kHIDUsage_KeyboardDownArrow`, but that's not really the down arrow. That element appears to be totally bogus since it doesn't seem to map to any key at all. This would happen 100% of the time on every Mac that I own, however, there were certain models of Macs that seemed to work fine (according to Apple). So, we've put in a hack here that checks if we're on 10.3 or later, and if so we just toss the Right Control key element. But if we're on 10.2 then we set our name string to "Down Arrow".

You should also be aware that the keyboard does not work with the HID Manager at all on Mac OS 10.1 or earlier. You must have 10.2.6 or later for the HID to be at all useful. Preferably, your game should require 10.3 or later since that version of the HID is significantly more reliable than the older versions.

The IOHIDUsageTables.h header file defines constants for the modifier keys as this:

```
kHIDUsage_KeyboardLeftControl
kHIDUsage_KeyboardLeftShift
kHIDUsage_KeyboardLeftAlt
kHIDUsage_KeyboardLeftGUI
kHIDUsage_KeyboardRightControl
kHIDUsage_KeyboardRightShift
kHIDUsage_KeyboardRightAlt
kHIDUsage_KeyboardRightGUI
```

You'll notice that the modifiers all have a right and left version. Well, even though most keyboards do have different left and right modifier keys, these values are bogus in the world of HID. None of the right-side modifier keys will respond at all! This is probably another bug in the HID Manager, or maybe it's just a "feature", but either way if you see an Element with a Usage value of `kHIDUsage_KeyboardRightGUI` it won't work. The user can hammer on that right Command key to their heart's content, but the HID Manager will never register key press on that key. Only the left modifier values work.

Reading Input Data

Congratulations! You've made it through the minefield of bugs trying to get a list of all your device and control data. Now it's time to actually read some data! There are two different methods for reading data from a HID Device: You can poll the data directly, or create event queues that receive updates as they happen. There are pro's and con's to each method:

Polling

Polling data is the process of manually reading the value of each control element that our game needs. The nice thing about polling is that the code is very simple and clean, but the downside is that we're constantly reading the values of lots of elements, and this is considered by some to be inefficient. However, most games only have a small number of control needs (move, jump, fire, duck, etc), so the overhead of polling the controls for each of those really isn't bad. It costs you maybe 1:10,000th of a second at worst.

Queuing

Queuing is more efficient than polling since you're not constantly reading data from your USB devices, but the complexity of setting up a queue for every Element you're using adds a lot more mess to an already messy API. Queuing does have one other small advantage: you won't miss any button presses. When polling, it's possible that you'll miss a button press if the user presses the button and releases it in the same frame before you've had a chance to read the Element's state. With queuing, every press of every button gets added to the queue. However, in the real world this just doesn't matter since games run at 30+ frames per second, and I challenge anyone to try pressing and releasing the Fire button on a gamepad in 1/30th of a second or faster.

In my humble opinion, polling is the way to go since it saves a lot of messy code, and the downsides really aren't a concern. So, in the game engine that we're building here we'll be sticking with polling, but in case you do want to set up queues for your Elements, here's how you would do it:

Listing 12-10: Initializing a Queue and Reading Data from It

```
#define QUEUE_SIZE  10  // maximum # of values in queue before oldest
                        // elements in queue get purged

IOHIDQueueInterface **CreateQueueForHIDElements(
                IOHIDDeviceInterface    **hidDeviceInterface,
                long                    numElementsToQueue,
                IOHIDElementCookie      *elementCookies)
{
    IOHIDQueueInterface     **queue;
    long                    i;


            /* CALL OUR DEVICE INTERFACE TO MAKE A QUEUE */

    queue = (*hidDeviceInterface)->allocQueue(hidDeviceInterface);
    if (queue)
    {
        (*queue)->create(queue, 0, QUEUE_SIZE);


                /* ADD ELEMENTS TO THE QUEUE */
                //
                // We put all of the elements into the same queue
                //

        for (i = 0; i < numElementsToQueue; i++)
        (*queue)->addElement(queue, elementCookies[i], 0);


                /* START DATA DELIVERY TO QUEUE */

        (*queue)->start(queue);
    }

    return(queue);
}

void ReadDataFromDeviceQueue(IOHIDQueueInterface **queue)
{
    AbsoluteTime        zeroTime = {0,0};
    IOHIDEventStruct    event;
    HRESULT             result;

    do
    {
                /* GET THE NEXT EVENT (IF ANY) OUT OF THE QUEUE */
        result = (*queue)->getNextEvent(queue, &event, zeroTime, 0);
        if (!result)
        {
            /* PASS THE ELEMENT COOKIE AND THE VALUE TO SOME HANDLER */

            MyHandleQueueEvent(event.elementCookie, event.value);
        }
    }while(!result);
}
```

Now to put things into perspective, let's see how we read an Element's value with polling instead:

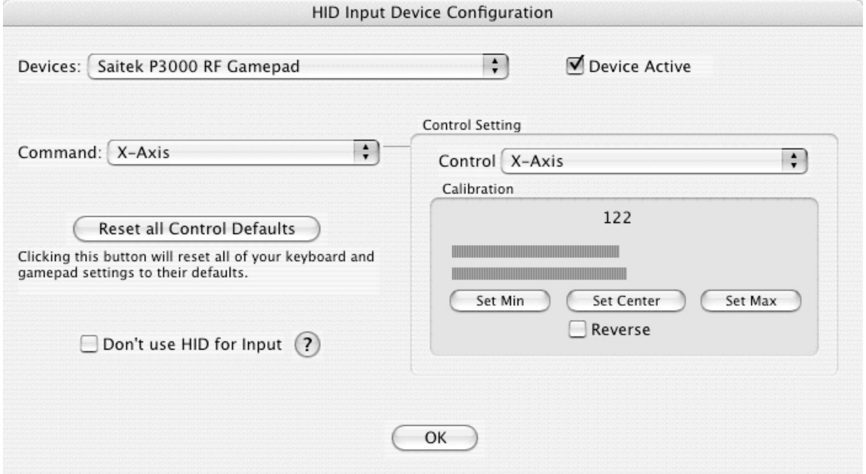
Listing 12-11: Polling an Element

```
    result = (*hidDeviceInterface)->getElementValue(hidDeviceInterface,
                                            cookie, &hidEvent);
    if (result != noErr)
        value = hidEvent.value = 0;
    else
        value = hidEvent.value;
```

Much simpler than queuing, don't you think? There is only one thing to be aware of, and it's another "feature" of the HID Manager. Notice that we check the result value returned from `getElementValue()`. This is critical because the HID will sometimes malfunction and spit out an error code when you call this. It's very rare, but it happens now and then. So, if an error occurs just assume that the value of the element is 0.

Input.c

The "HID Manager Input.xcode" project has a full implementation of everything we've talked about in this Chapter. This is heavily based on the input system that I used in Nanosaur 2, and it includes a configuration dialog for letting the user configure their devices based on a needs list. It also has code for saving and restoring the configuration of all the devices along with some calibration functions for joystick axes.



HID Input Device Configuration

Devices: Saitek P3000 RF Gamepad

☑ Device Active

Command: X-Axis

Control Setting

Control X-Axis

Calibration

122

☐ Reverse

Clicking this button will reset all of your keyboard and gamepad settings to their defaults.

☐ Don't use HID for Input

Figure 12-3: Our HID Configuration Dialog

If you're familiar with how Input Sprocket used to work, then you'll understand how my little HID system in Input.c works since it is also based on a "needs" list. A Need is a structure that defines an action that the game performs based on input from a device. For example, a standard game will have a Need for the fire weapon action, a Need for the turn left action, a Need for the jump action, etc. The `InputNeedType` structure that we're using is defined like this:

```
typedef struct
{
    char                    name[64];
    short                   defaultKeyboardKey;
    NeedElementInfoType     elementInfo[MAX_HID_DEVICES];
    long                    value;
    long                    oldValue;
    Boolean                 newButtonPress;
}InputNeedType;
```

`name`

This is a 64-character text string containing the name of the action to be performed such as "Jump" or "Turn Left". This is what the user will see in the configuration dialog.

`defaultKeyboardKey`

This is the default keyboard element usage value to assign to this action. For example, to assign the right arrow key to the Turn Right action, you would set it to `kHIDUsage_KeyboardRightArrow`.

`elementInfo`

You can assign multiple device elements to the same Need. For example, the user might have assigned the spacebar to the Fire action, and also assigned the A button on his gamepad to the Fire action. So, for each device there may be an element assigned to this Need, and the `elementInfo` will tell us the status of each of those elements.

```
typedef struct
{
    short   elementNum;
    long    elementCurrentValue;
}NeedElementInfoType;
```

The `elementNum` value is just an index into the Element list for each device, and `elementCurrentValue` is the most recently polled value of that Element.

`value`

This is the final value of the Need. We get the value by scanning each Element assigned to this Need. The largest value is the one we keep. So, if the Fire button on the keyboard is not pressed (giving a value of 0), but the Fire button on the gamepad is pressed (giving a value of 1), our engine will take the 1 and put that in the `value` field.

`oldValue`

Every time we update all of our Needs by polling all the elements assigned to them, it's a good idea to keep a copy of the previous value in case anything in the game needs to know old info (say, for determining state changes).

`newButtonPress`

This gives our code an easy way to tell if a button press is new on this frame. It's calculated simply by comparing `value` with `oldValue`. If `oldValue` was 0 and `value` is 1 then obviously this is a new button press.

Toward the top of the Input.c file all of the Needs for our sample application are defined:

Listing 12-12: The Sample Project's Needs List

```
InputNeedType   gControlNeeds[NUM_CONTROL_NEEDS] =
{
    {                                   // kNeed_TurnLeft_Button
        "Turn Left Button",
        kHIDUsage_KeyboardLeftArrow,    // keyboard default = left arrow
    },

    {                                   // kNeed_TurnRight_Button
        "Turn Right Button",
        kHIDUsage_KeyboardRightArrow,   // keyboard default = right arrow
    },

    {                                   // kNeed_Forward_Button
        "Forward Button",
        kHIDUsage_KeyboardUpArrow,      // keyboard default = up arrow
    },

    {                                   // kNeed_Backward_Button
        "Backward Button",
        kHIDUsage_KeyboardDownArrow,    // keyboard default = down arrow
    },

    {                                   // kNeed_XAxis
        "X-Axis",
        0,
    },

    {                                   // kNeed_YAxis
        "Y-Axis",
        0,
    },

    {                                   // kNeed_Fire
        "Fire Button",
        kHIDUsage_KeyboardSpacebar, // keyboard default = spacebar
    },
};
```

This is pretty straightforward stuff. We're just filling the array with some of the basic fields for each Need. We set the name followed by the default keyboard key. You'll notice that we don't set a default key for the X-Axis and Y-Axis Needs since there's no way to simulate an axis on a keyboard. If you wanted, you could assign a key here, but the axis will only range from 0 to 1 as you press and release the key.

There you have it. That's the basic introduction to the HID Manager, and you have my condolences. Unless your game will really benefit from gamepad or joystick input, my recommendation is to steer clear of the HID Manager, and just stick with the other input methods for reading the keyboard and mouse. It'll save you weeks of coding and debugging, and years of your life since the stress involved with doing tech support for the HID Manager will turn your hair gray in no time.

Chapter 13: Writing a Maya Plug-in

We've now got a pretty huge chunk of our game engine written, but there's one really important thing missing: 3D model data. Up until now we've just been spinning a colored cube whose geometry is generated in the code. Getting 3D model data into an application takes some work, so we'll need a whole chapter to cover it.

The de-facto standard in 3D modeling applications is Maya. There are plenty of other 3D modeling programs on the Mac, but odds are that nobody will take you seriously unless you're using Maya. So, if you haven't done so already, you should shell out the $2000 and go buy a copy along with some of the many Maya books that are available.

On the CD for this book you'll see a project called "bg3dExporter.xcode". This project contains a full, working exporter plug-in for Maya 6. In this chapter I'm going to discuss the fundamental things you need to know about writing an exporter plug-in for Maya, and I'll talk a bit about the BG3D file format that our game engine is going to use in all of the sample code to follow.

Initializing A Maya Plug-In

Maya has a very powerful plug-in architecture, but unfortunately it is written in C++ instead of straight C, and because of the way they did their API you can only call their functions from C++. In the bg3dExporter.xcode project you'll find the Maya.cpp source file. This file contains all of the code that interfaces directly with the Maya API calls.

There are several things that must be set up just right in order for Maya to recognize a plug-in:

Listing 13-1:The Plug-In Command Definition

```
class bg3d : public MPxCommand
{
public:
                    bg3d();
    virtual         ~bg3d();

    MStatus         doIt(const MArgList& args);
    static          MSyntax newSyntax();
    static void*    creator();
private:
    void            printType(const MObject& node,
                            const   MString& prefix);
    bool            quiet;
};
```

This cryptic C++ code is essentially telling Maya that our command “bg3d” should call the function `doIt()` whenever we execute the command from the Maya command line.

We also need an initialization function like this:

Listing 13-2:Initializing the Plug-In

```
MStatus initializePlugin(MObject obj)
{
    MStatus     status;
    MFnPlugin   plugin( obj, "Alias", "3.0", "Any" );


    status = plugin.registerCommand( "bg3d",
                                    bg3d::creator,
                                    bg3d::newSyntax );
    if (!status)
    {
        status.perror("registerCommand");
        return status;
    }

    return status;
}
```

This initialization changes a little every time there’s a new version of Maya. For Maya 6 we need to set this up exactly as shown. When a new version of Maya comes out you’ll need to check the sample code that comes with it to see what the new version requires on this line:

```
    MFnPlugin   plugin( obj, "Alias", "3.0", "Any" );
```

The `plugin()` call creates a plug-in object that we can use to call Maya’s plug-in API calls. The first API call that we make is used to register our plug-in’s command name with Maya:

```
    status = plugin.registerCommand( "bg3d",
                                    bg3d::creator,
                                    bg3d::newSyntax );
```

By passing “bg3d” to `registerCommand()`, Maya will know that when the user types “bg3d” on the command line, it should invoke this plug-in.

The Plug-In Entry Point

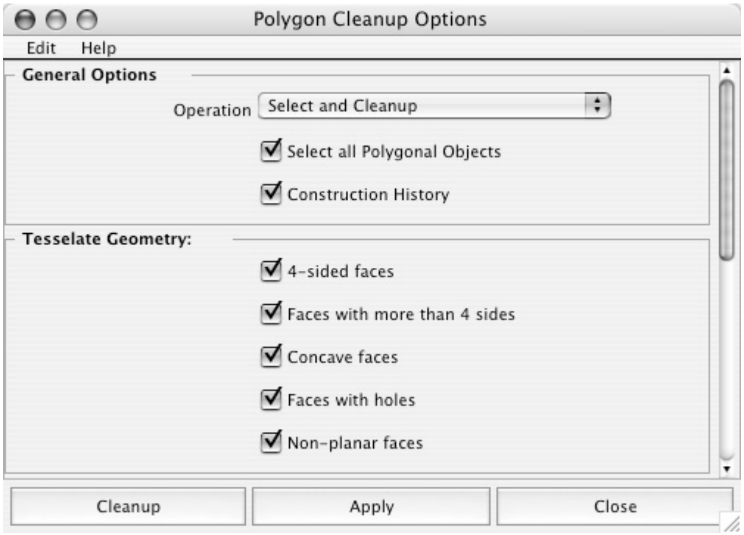
We’ve registered our plug-in with Maya, so if the user types “bg3d” it causes Maya to call the `doIt()` function. This function triangulates the entire scene, and then calls our main exporter code that decomposes the scene and writes out the data. When that is done, we undo the triangulation before returning back to Maya.

Listing 13-3:The doIt() Plug-In entry Function

```
MStatus bg3d::doIt( const MArgList& args )
{

            /* ISSUE COMMAND TO TRIANGULATE EVERYTHING */

    MString cmd = "polyCleanupArgList 3 {
            \"1\",\"1\",\"1\",\"1\",\"1\",\"1\",\"1\",\"1\",\"1\",\"1e-
            05\",\"0\",\"1e-    05\",\"0\",\"1e-05\",\"0\",\"-1\",\"0\" };";
    MGlobal::executeCommand(cmd, true, true);
    MGlobal::executeCommand(cmd, true, true);

            /* PROCESS PLUGIN */

    MayaDisplayMessage("Calling PluginEntry()");
    PluginEntry();


            /* ISSUE MEL COMMAND TO UNDO TRIANGULATE */

    MString cmd2 = "Undo";
    MGlobal::executeCommand(cmd2, true, true);
    MGlobal::executeCommand(cmd2, true, true);


    return MS::kSuccess;
}
```

It is possible to issue commands to Maya from inside a plug-in. The first thing we do in the `doIt()` function above is to issue a `polyCleanupArtList` command which will triangulate the entire scene for us. The gibberish after the command is actually a list of parameters to send to the command. I have no idea what that gibberish actually means, and I don’t even need to know because I got all of that text from Maya’s Script Editor. Here’s how it’s done:

Run Maya and go to the Polygon menu where you’ll see a menu item called Cleanup. Select the Cleanup option box to get to the Polygon Cleanup Options dialog:



General Options

Operation: Select and Cleanup

☑ Select all Polygonal Objects

☑ Construction History

Tesselate Geometry:

☑ 4-sided faces

☑ Faces with more than 4 sides

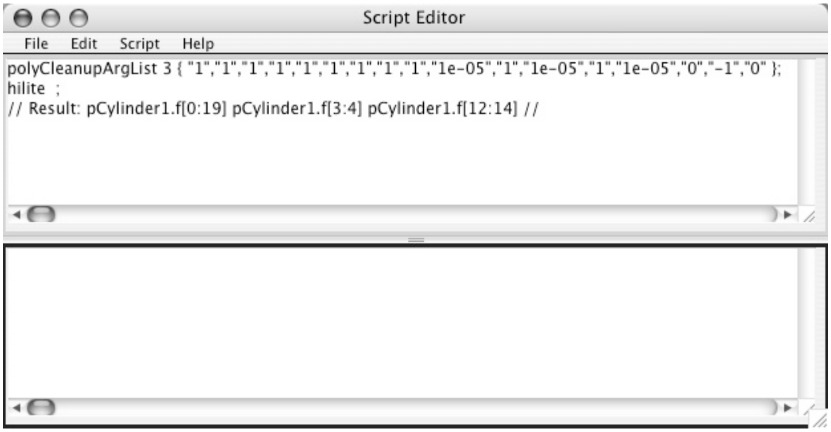
☑ Concave faces

☑ Faces with holes

☑ Non-planar faces

Figure 13-1: The Polygon Cleanup Options dialog

Make sure all of the tessellation options are checked. Then click the Apply button to cause the scene to get tessellated into triangles. Now look in the Maya Script Editor window where you'll see the full command that was issued:



```
polyCleanupArgList 3 { "1","1","1","1","1","1","1","1","1","1e-05","1","1e-05","1","1e-05","0","-1","0" };
hilite  ;
// Result: pCylinder1.f[0:19] pCylinder1.f[3:4] pCylinder1.f[12:14] //
```

Figure 13-2: The polyCleanupArgList command that was issued

This command's text is exactly what we've copied into the `doIt()` function, so by issuing that command in our plug-in we're doing the exact same thing that the Polygon Cleanup

Options dialog did for us. This can be done for any command in Maya. Just see what the command text is in the Script Editor, and then copy-paste it into your code.

You will notice that we issue the `polyCleanupArgList` command twice:

```
MGlobal::executeCommand(cmd, true, true);
MGlobal::executeCommand(cmd, true, true);
```

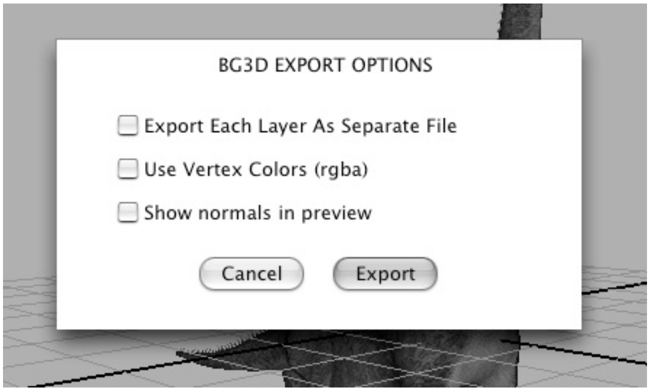
The reason for this is that this Maya command is a little buggy. It doesn't always tessellate everything down to triangles on the first pass, but doing it twice will ensure that everything has been triangulated.

Once our plug-in is done exporting all of that triangle data, we need to return Maya to its original state by issuing two `Undo` commands to undo both of our `polyCleanupArgLlst` commands.

```
MString cmd2 = "Undo";
MGlobal::executeCommand(cmd2, true, true);
MGlobal::executeCommand(cmd2, true, true);
```

Getting the Scene's Layer Info

Polygon meshes in Maya can be grouped into Layers. A Layer is a very useful way to organize multiple meshes into single Objects. We can use layers as a way of having multiple objects in a single file, and our exporter can export the data in each layer as a separate BG3D file:



BG3D EXPORT OPTIONS

☐ Export Each Layer As Separate File

☐ Use Vertex Colors (rgba)

☐ Show normals in preview

Figure 13-3: Our plug-in's options dialog

For example, below is a screenshot showing the Maya file that has all of the rock models used in Nanosaur 2. The BG3D exporter has the option to save each rock model as a different file whose name is the same as the Layer name. So, the layer “tall_rock_1” will be exported as “tall_rock_1.bg3d”.

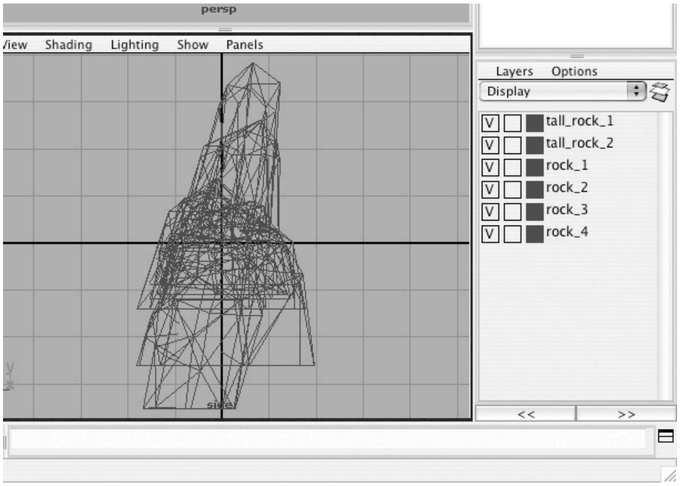


Figure 13-4: Six different rock models in one Maya file.

Our plug-in simply has to scan Maya’s Layer List, find the mesh data in each layer, and then output that mesh data to a BG3D file.

Listing 13-4:Get Maya’s Layer List

```
OSErr Maya_GetLayersInfo(void)
{
    int             i, j;
    MStatus         stat;
    bool            visible;
    MStringArray    layerMemberString;

    gNumLayers = 0;             // init layer count


        /************************************************************/
        /* SCAN THRU ALL DISPLAY LAYER DG NODES TO GET ATTRIB INFO */
        /************************************************************/

            /* START ITERATION LOOKING ONLY FOR DISPLAY LAYERS */
```

```
MItDependencyNodes dgIter(MFn::kDisplayLayer, &stat);


            /* ITERATE THRU ALL LAYERS */

for ( ; !dgIter.isDone(); dgIter.next())
{
    MObject     dgItem = dgIter.item(); // get this layer's MObject
    MFnDependencyNode   fnNode(dgItem); // get the node of Object


        /* GET THE LAYER'S NAME & CONVERT TO PASCAL STRING */

    MString layerName = fnNode.name();  // get name from the node
    c2pstrcpy(gLayerInfoList[gNumLayers].name, layerName.asChar());


            /* SEE IF THIS LAYER IS VISIBLE */

                        // get visibility attribute of the node
    MPlug visPlug = fnNode.findPlug("visibility", &stat);

                        // get value of the attribute
    stat = visPlug.getValue(visible);

    gLayerInfoList[gNumLayers].isVisible = visible;

    gNumLayers++;
}


/*******************************************************************/
/* FIND THE NAMES OF ALL OF THE OBJECTS THAT BELONG TO EACH LAYER */
/*******************************************************************/

        /* SCAN EACH LAYER WE GOT ABOVE */

for (i = 0; i < gNumLayers; i++)
{
    char        layerName[255];
    long        numMembers;

        /* ISSUE COMMAND TO GET DISPLAY LAYER MEMBERS */
        //
        // First build the command w/o the layer’s name, and
        // then append the layer name to the command string
        //

    MString cmd = "editDisplayLayerMembers -q ";
    p2cstrcpy(layerName, gLayerInfoList[i].name);
    cmd += layerName;
```

```
        MGlobal::executeCommand(cmd, layerMemberString, false, false);

                    /* COUNT # OF STRINGS RETURNED */

        numMembers = layerMemberString.length();
        gLayerInfoList[i].numMembers = numMembers;


                /* SAVE THE NAMES OF ALL THOSE OBJECTS INTO OUR LIST */

        for (j = 0; j < numMembers; j++)
        {
            const char *memberName = layerMemberString[j].asChar();
            c2pstrcpy(gLayerInfoList[i].memberNames[j], memberName);
        }
    }

    return(noErr);
}
```

This function has two sections: the first scans the layers and determines which ones are visible (we don't want to export any layers that are hidden), and then the second section determines which objects are assigned to each layer.

A scene in Maya is built out of "DG Nodes" which are all linked together to form a hierarchy. Each node contains data of different types. There are layer nodes, texture nodes, geometry nodes, etc. In this case, we're looking for Layer nodes, so we start iterating through our scene's nodes with this call:

```
    MItDependencyNodes dgIter(MFn::kDisplayLayer, &stat);
```

This creates an iteration object that will iterate through only Display Layers. To increase the iteration to the next layer, you invoke the iteration's `next()` function like this:

```
    dgIter.next();
```

Then, to determine if we've reached the end of the list we test this function:

```
    dgIter.isDone();
```

This is the method for iterating through any kind of data in Maya, so you'll see it elsewhere in our exporter's code. The iteration object has all sorts of functions that can be called to get information about the node. The next thing we need to do is to get a reference to the data "Object" in this node:

```
    MObject     dgItem = dgIter.item();
```

As we scan through these node objects, we get the `MFnDependencyNode` for each one:

```
MFnDependencyNode   fnNode(dgItem);
```

This dependency node contains all of the function pointers relevant to this specific type of object. This is a Layer object, so all of the functions in `fnNode` will be Layer-related. There are two things we need to know about the Node: its name and its visibility flag. Getting the name is easy:

```
MString     layerName = fnNode.name();
```

The `MString` variable `layerName` is a Maya string object, so to make it useful we need to call a sub-function to get the pointer to the actual C string text:

```
layerName.asChar();
```

Next we want to determine if this Layer is visible or not. Layers in Maya can be hidden or shown by toggling the Visibility checkbox in the Layer Pane:

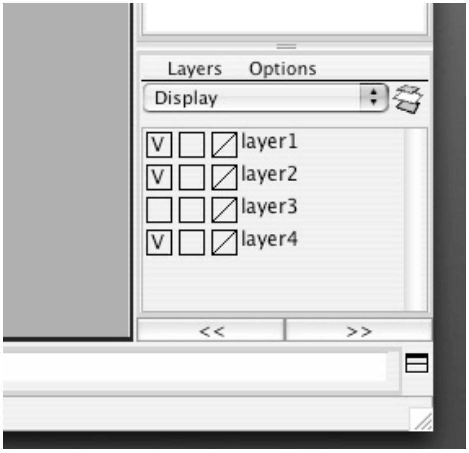


Figure 13-5: Layer pane showing layer 3 is hidden

Getting the Visibility attribute of a Layer is different from getting the name since there isn't an explicit `getVisibility()` function in the Layer Object. Instead, we have to search for an attribute named “visibility” in the node by doing this:

```
MPlug visPlug = fnNode.findPlug("visibility", &stat);
stat = visPlug.getValue(visible);
```

The `findPlug()` function will look for an attribute named "visibility" in the Layer object. Then, to get the value of the visibility attribute, we call `getValue()` which returns a Boolean value in the variable `visible`.

At this point we've got a list of all of the Layers in our scene, so now our `Maya_GetLayersInfo()` function needs to determine which geometry objects are assigned to each layer. This is a bit tricky because there's no easy way to find this out. You'd think that there would be a function call to get the referenced objects for each layer, but there isn't. Instead we've got to do this the hard way by issuing a command to Maya which will spit out a list of the names of all of the objects associated with a layer:

```
MString cmd = "editDisplayLayerMembers -q ";
p2cstrcpy(layerName, gLayerInfoList[i].name);
cmd += layerName;
```

This code builds a command string that looks something like `"editDisplayLayerMembers –q layer1`", and then we execute the command in the usual way:

```
MGlobal::executeCommand(cmd, layerMemberString, false, false);
```

The variable `layerMemberString` will contain an array of `MString` objects upon return from executing this command. The number of members (i.e. geometry meshes) found is determined by counting the number of strings returned:

```
numMembers = layerMemberString.length();
```

The C string pointers are extracted from the `MString` objects the same way as we did for the Layer names earlier:

```
const char *memberName = layerMemberString[j].asChar();
```

When the `Maya_GetLayersInfo()` function completes we'll have a list of Layers, and for each Layer we'll have a list of the names of the objects associated with that Layer. This member list contains more than just the names of the geometry meshes in that layer. It may also contain the names of transform objects and shapes as well. We'll be tossing that data out later once we identify those objects, but for now it's all in each Layer's member name list.

Extracting Geometry Data

The meat of any plug-in is the part that actually gets geometry data, and does something with it. In our case, we need to find all of the vertices, polygons, and materials associated with all of the meshes in our scene, and then save them out to a BG3D file. We can iterate through all of the meshes in our scene much in the same way that we iterated through all of the layers earlier, but things get much more complicated here. The basic flow of things goes like this:

```
Loop Through All Layers
    • Iterate through all Meshes
    • If Mesh's name is in our Layer's member list then this mesh is in this layer.
            Find the materials assigned to this mesh
                    - find the mesh faces assigned to this material
                            - get face vertex points, uv's, normals, and colors
```

Iterating through the Meshes

In Maya.cpp the function that performs this task is `Maya_ExtractFacesAndMaterials()`. This function is large and contains a lot of code that isn't of much interest here, so rather than listing the whole thing and discussing it line by line, I'm going to discuss the fundamental parts of the extraction process. For starters, to iterate through the Meshes in the scene we need to create an iteration object:

```
MItDag dagIter( MItDag::kBreadthFirst, MFn::kMesh, &stat );
```

Iterating through meshes is exactly the same as iterating through layers, so we use the `next()` and `isDone()` functions to move from the current mesh to the next. To get each mesh's data we first must create a path to its node:

```
dagIter.getPath( dagPath );
```

Then we can extract the function pointers for the node like this:

```
MFnDagNode dagNode(dagPath, &stat);
```

The `dagNode` variable is now what we can use to access the Mesh's functions. This seems like a long way to go to do this, and it is. If you've ever used the plug-in API for other 3D modeling applications such as Lightwave then this probably seems overly complicated.

Even though we requested only Meshes when we initialized our iteration object `dagIter`, it is still wise to verify the node since sometimes Maya will give us data that we don't want:

```
if ( dagNode.isIntermediateObject())
    continue;
if ( !dagPath.hasFn( MFn::kMesh ))
    continue;
```

```
if ( dagPath.hasFn( MFn::kTransform ))
    continue;
```

There are three tests that are performed here. First, we test to see if the Node is an "intermediate object". We don't want those, so skip them. Second, we double-check that the node has function pointers to access Mesh data. If not then it's an invalid mesh, so skip it. Finally, if the node has transformation function pointers then something isn't right, so skip it. After this, we can be assured that the node really is a Mesh object.

Our iteration is going through every mesh in the scene, but we only want meshes for the current layer that we're exporting, so, we need to get the name of this mesh and compare it to the list of object names in our Layer. For each mesh, we get its name like so:

```
MString         meshPathMString = dagNode.partialPathName();
const char      *meshPathNamePtr = meshPathMString.asChar();
```

If that name string isn't in the current layer's list then we skip it and iterate to the next mesh, but if there is a match then it's time to extract some data out of it.

Iterate Through the Material's Triangle List

Here's where things get a little funky. The way we extract polygons and vertices out of a mesh isn't by directly extracting that data from our `dagNode`. Nope, instead what we do is find the Material assigned to this Mesh, and then find the polygons associated with that Material. To get the list of all material objects connected to the Mesh we do this:

```
MObjectArray    sets;
MObjectArray    comps;

MFnMesh fnMesh(dagPath);

unsigned instanceNumber = dagPath.instanceNumber();

fnMesh.getConnectedSetsAndMembers(instanceNumber, sets,
                                comps, true);
```

A mesh may have multiple materials assigned to it, but for simplicity we're only going to look for the first material, so we get the first element out of the material array:

```
MObject set = sets[0];              // first material in list
MObject comp = comps[0];
```

Maya has many types of materials, but the only kinds we're interested in are Surface Shaders since those are what we can easily export to the BG3D file and OpenGL can use. Determining if a material is a Surface Shader is similar to finding the Visibility attribute in a layer, but this time we look for an attribute named “surfaceShader”:

```
MFnSet              fnSet(set);
MFnDependencyNode   dnSet(set);

MObject             ssAttr = dnSet.attribute(
                                MString("surfaceShader"));
MPlug               ssPlug(set, ssAttr);

MPlugArray          srcPlugArray;
ssPlug.connectedTo(srcPlugArray, true, false);

if (srcPlugArray.length() == 0)
    continue;
```

Once again, it’s a little cryptic what’s going on here thanks to C++, but all that is happening is that we’re making sure that this material object has a `surfaceShader` attribute. If it does, then we can extract the Surface Shader’s node:

```
MObject     shaderObject = srcPlugArray[0].node();
```

Iterate Through a Polygon’s Vertices

Later, we’ll see how to extract color and texture information from the shader object, but right now we’re concerned with extracting the mesh data. We need to set up another iteration object so that we can iterate through all of the polygons associated with the current material. This should be looking familiar by now:

```
MItMeshPolygon piter(dagPath, comp);
```

Now things get easier because extracting polygon data is simple and logical. First we’ll get the number of vertices in this polygon:

```
numPoints = piter.polygonVertexCount();
```

Since we tessellated the entire scene earlier, `numPoints` should always come up as 3, but it’s always a good idea to verify it just in case. To get the coordinates of each of the three vertices we do this:

```
for (i = 0; i < 3; i++)
{
    float   pointtemp[4];

    MPoint  mpoint = piter.point(i, MSpace::kWorld);
```

```
        mpoint.get(pointtemp);

        x[i] = pointtemp[0];
        y[i] = pointtemp[1];
        z[i] = pointtemp[2];
    }
```

Note that we are asking for the coordinates in world-space by passing `kWorld` into the `point()` function. This causes the coordinates to be returned as they appear in Maya, but the scale may be different since the API always returns coordinates in the millimeter scale. In other words, if you're in meter mode when you're modeling in Maya and then you export your model, the coordinates will be 100x what you'd expect. So, suppose you're in meter mode and you have a vertex at the coordinate 45, 100, -17. Well, the API is going to give you the value 4500, 10000, -1700 because that's what it is in millimeters. This means you have a choice of either always working in millimeter mode in Maya to keep things even, or if you prefer working in meters then you'll need to divide each coordinate x,y,z by 100 to scale it down. I prefer to work in meters in Maya, so the BG3D Exporter project does divide all the coordinates to keep them at the proper scale. You may need to modify this depending on what unit scale mode you prefer to work with in Maya.

Extracting the polygon's vertex indices is very easy, and these will be in counter-clockwise order so that backfaces will be correct:

```
    vertexIndices[0] = piter.vertexIndex(0);
    vertexIndices[1] = piter.vertexIndex(1);
    vertexIndices[2] = piter.vertexIndex(2);
```

Getting the polygon vertex normals, UV's, and colors is just as easy as getting the coordinates. When reading in the vertex normals it's always a smart idea to make sure that they're normalized:

```
    for (i = 0; i < 3; i++)
    {
        double      normaltemp[3];
        Mvector     mvec;

        piter.getNormal(i, mvec, MSpace::kWorld);
        mvec.get(normaltemp);

        triNormals[i].x = normaltemp[0];
        triNormals[i].y = normaltemp[1];
        triNormals[i].z = normaltemp[2];

        OGLVector3D_Normalize(&triNormals[i], &triNormals[i]);
    }
```

Reading UV's goes like this:

```
for (i = 0; i < 3; i++)
{
    float2 tempUV;

    piter.getUV(i, tempUV);

    triUVs[i].u = tempUV[0];
    triUVs[i].v = tempUV[1];
}
```

Reading vertex colors is equally as easy, but we do need to be careful about one thing: Calling the iteration's `getColor()` function will return black (0,0,0) if no vertex color has explicitly been set in Maya, so it is important to verify that the vertex actually has a color before trying to get it:

```
    for (i = 0; i < 3; i++)
    {
        if (piter.hasColor(i, &stat))   // does this vertex have color?
        {
            MColor  mcolor;
            piter.getColor(mcolor, i);

            triColors[i].r = mcolor.r;
            triColors[i].g = mcolor.g;
            triColors[i].b = mcolor.b;
            triColors[i].a = mcolor.a;
        }
        else
        {
            triColors[i].r =
            triColors[i].g =
            triColors[i].b =
            triColors[i].a = 1.0;
        }
}
```

Extracting Shader Data

Earlier, we found a Shader Object assigned to a Mesh, so, now let's see how to extract information from it. The first piece of information we should get is the texture map (if any). The pathname to the shader's texture file can be found with this code:

Listing 13-5:Getting a Texture's Pathname

```
MPlug colorPlug = MFnDependencyNode(shaderObj).findPlug("color",
                                                        &status);
if (status != MS::kFailure)
{
    MItDependencyGraph dgIt(colorPlug, MFn::kFileTexture,
                            MItDependencyGraph::kUpstream,
                            MItDependencyGraph::kBreadthFirst,
                            MItDependencyGraph::kNodeLevel, &status);

    if (status != MS::kFailure)
    {
        dgIt.disablePruningOnFilter();

        if (!dgIt.isDone())                 // true == no texture
        {
            const   char *filePath;
            FSSpec  spec;
            OSErr   iErr;
            Str255  hfsPath;
            MString textureName;
            char    newPath[500];

            MObject textureNode = dgIt.thisNode();
            MPlug filenamePlug =
                        MFnDependencyNode(textureNode).findPlug(
                        "fileTextureName");

            filenamePlug.getValue(textureName);

                        /* GET UNIX PATH */

            filePath = textureName.asChar();
            strcpy (newPath, filePath);         // copy to our buffer

                        /* CONVERT TO HFS PATH */

            ConvertFileRepresentation(newPath, kCFURLPOSIXPathStyle,
                                    kCFURLHFSPathStyle);
            c2pstrcpy(hfsPath, newPath);


                    /* MAKE IT INTO AN FSSPEC THAT WE CAN USE */

            iErr = FSMakeFSSpec(0,0, hfsPath, &spec);
            if (iErr)
            {
                MGlobal::displayError("Cannot find texture file!”);
                return(-1);
            }
```

```
                    /* GET THE FILE AND DEAL WITH IT */

                GetTextureMap(m, &spec, &matData);

                hasTexture = true;
            }
        }
    }
```

Once again, we start by looking for an attribute in an object. This time we're looking for the "color" attribute in our shader object:

```
MPlug colorPlug = MFnDependencyNode(shaderObj).findPlug("color",
                                  &status);
```

Then we make another iteration object, this time of type `kFileTexture`, but we have no intention of actually iterating through this because the first instance contains the data we're looking for. We test the iteration's `isDone()` function, and if it returns `true` then this shader object doesn't have any texture map. Otherwise, we know there is a texture to be found. To get the pathname of the texture map's source file, we look for yet another attribute:

```
MPlug filenamePlug = MFnDependencyNode(textureNode).findPlug(
                    "fileTextureName");
filenamePlug.getValue(textureName);
filePath = textureName.asChar();
```

This returns a UNIX file path that is pretty much useless to us, so we call a new function that uses some Core Foundation calls to convert the UNIX path to a nice HFS path. Then we're free to load the texture map, and do whatever we want with it.

```
strcpy (newPath, filePath);               // copy path into a buffer

ConvertFileRepresentation(newPath,
                    kCFURLPOSIXPathStyle,
                    kCFURLHFSPathStyle);    // convert UNIX to HFS path
c2pstrcpy(hfsPath, newPath);              // make Pascal string

FSMakeFSSpec(0,0, hfsPath, &spec);        // get FSSpec of texture
```

The function `ConvertFileRepresentation()` is the heart of the UNIX to HFS conversion, and it looks like this:

Listing 13-6:Converting file paths from one type to another

```
Boolean ConvertFileRepresentation (char *fileName, short inStyle,
                                short outStyle)
{
    CFStringRef     rawPath;
    CFURLRef        baseURL;
    CFStringRef     newURL;
    char            newPath[500];

    if (fileName == nil)
        return (false);

    if (inStyle == outStyle)
        return (true);

    rawPath = CFStringCreateWithCString (nil, fileName,
                                            kCFStringEncodingUTF8);
    if (rawPath == nil)
        return (false);

    baseURL = CFURLCreateWithFileSystemPath (nil, rawPath,
                                            (CFURLPathStyle)inStyle, false);
    CFRelease (rawPath);
    if (baseURL == nil)
        return (false);

    newURL = CFURLCopyFileSystemPath (baseURL, (CFURLPathStyle)outStyle);
    CFRelease (baseURL);
    if (newURL == nil)
        return (false);

    CFStringGetCString (newURL, newPath, MAXPATHLEN, kCFStringEncodingUTF8);
    CFRelease (newURL);
    strcpy (fileName, newPath);
    return (true);
}
```

Personally, I like to use Quicktime's Image Importer features to load these texture maps since Quicktime supports virtually every file format under the sun. We'll talk more about importing images with Quicktime in Chapter 17.

In addition to texture maps there are also color parameters associated with a surface shader. Extracting the transparency and diffuse color information out of a shader object is really easy, but to do it we first need to know what type of shader it is. Just as there are several types of materials there are also several types of Surface Shaders. The two types we support in our plug-in are the Phong shader and the Lambert shader. To find out which type of shader this is, we call the shader's `apiType()` function:

```
MFn::Type   apiType = shaderObj.apiType();
```

Then we have some code that will extract the transparency and diffuse color information out of it:

Listing 13-7:Getting color info from a Phong or Lambert Shader

```
MColor              diffuseColor;
MColor              transColor;

switch(apiType)
{
            /************************/
            /* EXTRACT PHONG SHADER */
            /************************/

    case    MFn::kPhong:
                                // get functions for phong shaders
            MFnPhongShader  phong(shaderObj);


                        /* GET DIFFUSE COLOR */
                        //
                        // in Maya, the textures override diffuse
                        // color, so assume white, otherwise
                        // we'll end up with black.
                        //

            if (!hasTexture)

            {
                diffuseColor    = phong.color() * phong.diffuseCoeff();
                diffuseColor.get(tempColor);

                materialColor.r = tempColor[0];
                materialColor.g = tempColor[1];
                materialColor.b = tempColor[2];
            }
            else
            {
                materialColor.r =
                materialColor.g =
                materialColor.b = 1.0f;
            }

                        /* GET TRANSPARENCY */

            transColor = phong.transparency();
            materialColor.a = 1.0f - transColor.r;
            break;


            /**************************/
            /* EXTRACT LAMBERT SHADER */
            /**************************/

        case    MFn::kLambert:
                                        // get functions for phong shaders
            MFnLambertShader    lambert(shaderObj);


                        /* GET DIFFUSE COLOR */

            if (!hasTexture)
            {
                diffuseColor    = lambert.color() *
                                  lambert.diffuseCoeff();
                diffuseColor.get(tempColor);
                materialColor.r = tempColor[0];
                materialColor.g = tempColor[1];
                materialColor.b = tempColor[2];
            }
            else
            {
                materialColor.r =
                materialColor.g =
                materialColor.b = 1.0f;
            }


                        /* GET TRANSPARENCY */

            transColor      = lambert.transparency();
            matData.diffuseColor.a = 1.0f - transColor.r;
            break;


            /***************************/
            /* UNSUPPORTED SHADER TYPE */
            /***************************/

        default:
            MGlobal::displayError("unsupported shader type");
            return(-1);
    }
```

In Maya the diffuse color of a material is ignored if the material has a texture map. OpenGL, however, does not. OpenGL will apply the diffuse color to the material, essentially filtering it. For example, if we have a sphere mapped with a baseball texture, but we also have a red diffuse color, then OpenGL will render the baseball tinted red. In Maya, however, that diffuse color would be ignored, and a regular white baseball would be drawn. Therefore, if we know that we have a texture map associated with this shader, then we need to set the diffuse color to white so that it will render the same in OpenGL as it does in Maya. But if there's no texture then we can extract the diffuse color from the shader object.

To correctly calculate the material's color we need to multiply the diffuse color value by the diffuse coefficient. These are separate values in Maya that affect how an object is displayed.

```
diffuseColor    = phong.color() * phong.diffuseCoeff();
```

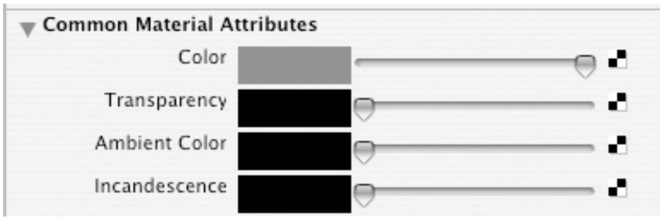


Figure 13-6: The color and transparency material settings

In Figure 13-6 you can see the material attributes for a shader that has an RGB color value of (1,0,0), and the slider next to the color indicates the diffuse coefficient. Here it is set all the way up to 1.0. Below the color setting is the transparency slider. Here it is set to 0.0, so the material is totally opaque. Note that a transparency value of 0.0 is opaque, and a value of 1.0 is totally transparent. This is the opposite of how alpha values work, so, when we read the transparency value from the shader, we have to invert it to make it into an alpha value:

```
transColor             = lambert.transparency();
matData.diffuseColor.a = 1.0f - transColor.r;
```

Installing the Maya Plug-In

Unfortunately, the Maya folks goofed with their plug-in architecture. The shared libraries that Maya loads must have a file extension of ".lib", but when Xcode builds a shared library it always attaches the extension ".dylib". If you install the file bg3dExporter.dylib in your Maya plug-ins folder, Maya won't see it. You've got to manually rename the file with a ".lib" extension. Once you've done this you should put the plug-in in the Users/Shared/Alias/maya/plug-ins folder:

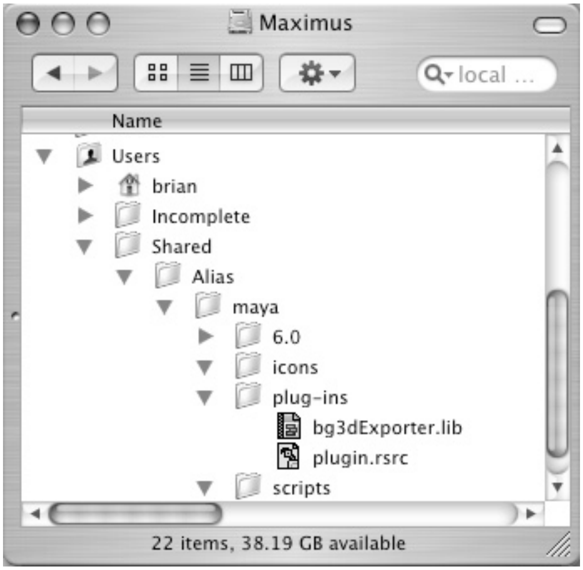


Figure 13-7: This is where the plug-in files go.

Note, however, that this file path changes almost every time that a new version of Maya is released. This is bound to change for the next version of Maya since Maya is no longer owned by Alias, hence, the Alias folder in the pathname will likely be renamed.

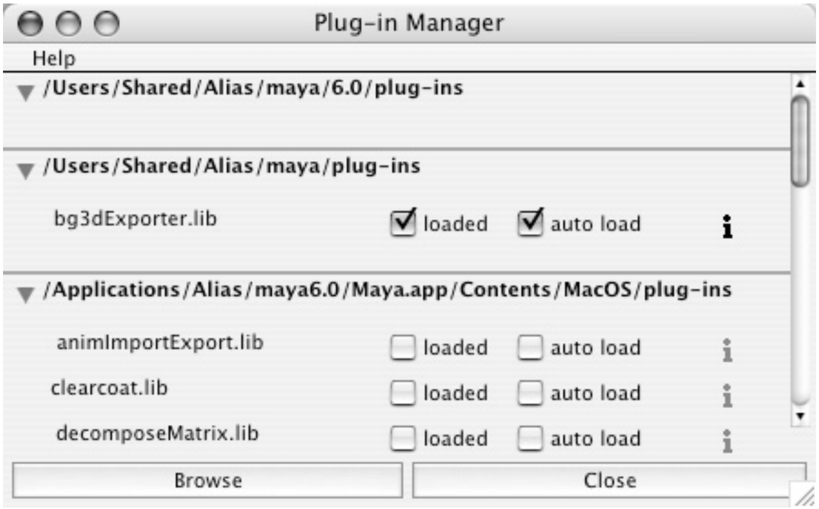
In addition to the inconvenience of manually renaming the file, you've also got to manually copy the plugin.rsrc file as well. Maya won't recognize a bundled shared library with .nib files, so to make life much easier for us it's best to use old-style .rsrc files for our dialogs. This .rsrc file also needs to be in the same plug-in's folder as the shared library itself. This way we'll be able to locate it with the following code:

Listing 13-8:Finding our Resource File

```
        /* FIND THE USERS FOLDER */

    theResult = FindFolder( kOnSystemDisk, kUsersFolderType, false,
                        &tmpVRef, &tmpDirID );
    if (theResult != noErr)
        StandardAlert(kAlertStopAlert, "\pCould not find Users folder!",
                        NULL, NULL, &alertItemHit);

            /* FIND PATH FROM USERS TO THE RSRC FILE */

    theResult = FSMakeFSSpec(tmpVRef, tmpDirID, "\p:Shared:Alias:maya:plug-
                            ins:plugin.rsrc", &gSharedLibSpec);
    if (theResult != noErr)
```

```
        StandardAlert(kAlertStopAlert, "\pcould not find plugin.rsrc!",
                      NULL, NULL, &alertItemHit);
```

If you've correctly renamed the shared library with the .lib extension and you've put it into the correct plug-ins folder along with the .rsrc file then Maya will see the plug-in, but you still have to tell Maya that you want to load it. Under the Window menu in Maya, select the "Settings/Preferences…" item and then the "Plug-In Manager…" item. This will bring up the Plug-in Manager dialog:



/Users/Shared/Alias/maya/6.0/plug-ins

/Users/Shared/Alias/maya/plug-ins

bg3dExporter.lib ☑ loaded ☑ auto load

/Applications/Alias/maya6.0/Maya.app/Contents/MacOS/plug-ins

animImportExport.lib ☐ loaded ☐ auto load

clearcoat.lib ☐ loaded ☐ auto load

decomposeMatrix.lib ☐ loaded ☐ auto load

Figure 13-8: The Maya Plug-in Manager dialog

Toward the top you'll see the bg3dExporter.lib plug-in. Be sure to check both the "loaded" and "auto load" checkboxes. This way, the plug-in will be installed automatically every time you run Maya. Whenever you want to export models in your scene, just type the command "bg3d" on Maya's command line, press Return, and the plug-in will get called.

This pretty much covers all of the information you need to write a Maya file exporter plug-in. The bg3dExporter.xcode project on the CD is fully functional, and it's what we'll be using to generate the models used by the sample code in the rest of this book. There is a lot of additional code in the project that deals with exporting the 3D model files in the format we want, but you may choose to export the data however you deem necessary for your game. Should you decide to use this tool as-is then you'll need to know a little about the files it spits out: the BG3D files which are discussed next.

The BG3D File Format

Back in the days of OS 9 Apple had their Quickdraw 3D technology. One of the great things about Quickdraw 3D was that it had a built-in 3D model file format called 3DMF and it had all the API functions needed to magically read and write these files. But when Quickdraw 3D bit the dust, so did 3DMF. I used to use the 3DMF files for all of my games back then including Weekend Warrior, Nanosaur, and Bugdom, so when I switched my game engine over to OpenGL for Cro-Mag Rally I had to figure out a substitute for 3DMF.

No existing 3D file format did what I needed for my games, so I decided to design my own file format that all of my games and tools now use. This new 3D file format is called BG3D (Brian Greenstone 3D), and it is a metafile consisting of tags followed by data. The tags you'll find in a BG3D file are as follows:

`BG3D_TAGTYPE_GEOMETRY`
This indicates that the data to follow is the header for a new geometry object. This header contains the type of geometry, the number of materials assigned to the texture along with the texture number. It also has basic geometry information such as the number of points and triangles in the geometry.

`BG3D_TAGTYPE_VERTEXARRAY`
This indicates that the data to follow consists of x, y, z vertex coordinates for the current geometry object.

`BG3D_TAGTYPE_NORMALARRAY`
This indicates that the data to follow consists of x, y, z vertex normals for the current geometry object.

`BG3D_TAGTYPE_UVARRAY`
This indicates that that data to follow consists of uv vertex texture coordinates for the current geometry object.

`BG3D_TAGTYPE_COLORARRAY`
This indicates that the data to follow consists of RGBA vertex color values for the current geometry object.

`BG3D_TAGTYPE_TRIANGLEARRAY`
This indicates that the data to follow consists of triangle vertex indices for the current geometry object.

BG3D_TAGTYPE_MATERIALFLAGS
This indicates that the data to follow is a simple 32-bit value that contains information about the next material we're going to load.

BG3D_TAGTYPE_MATERIALDIFFUSECOLOR
This indicates that the data to follow is an RGBA color value representing the current material's color.

BG3D_TAGTYPE_TEXTUREMAP
This indicates that the data to follow is a header plus texture map data. The header has the width and height of the texture along with the texture format of the texture. The data after the header will be of variable length depending on the size of the texture defined in that header.

BG3D_TAGTYPE_GROUPSTART
This indicates that any geometry to follow is part of a group. This is helpful for organizing multiple pieces of geometry that belong to a single object; like wheels being part of a car model.

BG3D_TAGTYPE_GROUPEND
This indicates the end of a group.

BG3D_TAGTYPE_ENDFILE
This indicates the end of the BG3D file.

In all of the sample projects to come you'll see a new source file called BG3D.c. This file contains all of the code needed to parse one of these BG3D files. Nothing particularly interesting happens in this code. It just opens the BG3D file and starts reading the tags and data out of it to build geometry objects in the form of OpenGL Vertex Arrays. You may wish to change this implementation or design your own file format for your games. In my games, the BG3D files have a few additional tags for other pieces of data such as bounding boxes and compressed textures, and that's the beauty of using a metafile for your model data since you can encapsulate anything you want in there depending on your needs.

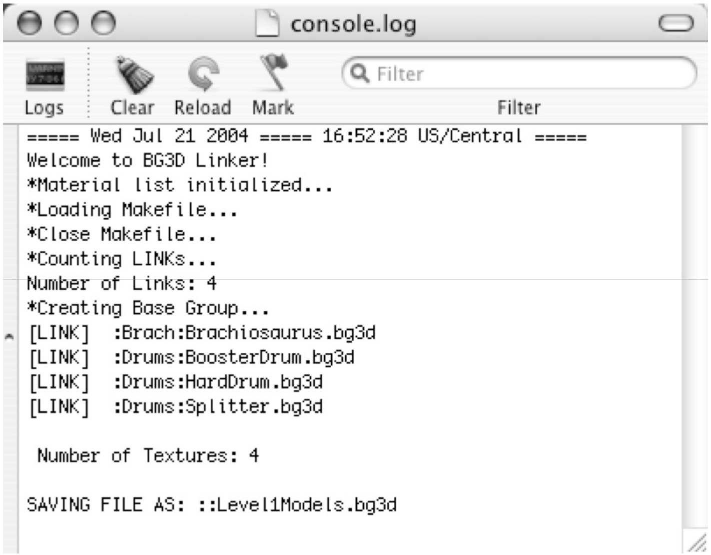
BG3D Linker

In your game you'll have hundreds of different models, so rather than having hundreds of individual BG3D files to load, it makes much more sense to group lots of BG3D files into a single, large BG3D file. That way, different models can share the same textures too. So, you'll find another useful tool on the book's CD called "BG3D Linker". You can use this tool to merge multiple BG3D files together using a makefile that looks something like this:

```
LINK :Brach:Brachiosaurus.bg3d
LINK :Drums:BoosterDrum.bg3d
LINK :Drums:HardDrum.bg3d
LINK :Drums:Splitter.bg3d

%::Level1Models.bg3d

end
```

The BG3D Linker tool will scan this makefile for all of the `LINK` keywords, and merge each of the referenced .bg3d files into the output file indicated by a `%` sign. So, in the above example four different .bg3d files will be merged into a single BG3D file named `Level1Models.bg3d`. If different models share the same texture map, BG3D Linker removes the duplicates so that the final file is small and efficient.

When you run BG3D Linker, select **Execute Makefile** from the **File** menu, and then select your makefile. As the makefile gets processed you'll see the output in the Console window:



```
===== Wed Jul 21 2004 ===== 16:52:28 US/Central =====
Welcome to BG3D Linker!
*Material list initialized...
*Loading Makefile...
*Close Makefile...
*Counting LINKs...
Number of Links: 4
*Creating Base Group...
[LINK]  :Brach:Brachiosaurus.bg3d
[LINK]  :Drums:BoosterDrum.bg3d
[LINK]  :Drums:HardDrum.bg3d
[LINK]  :Drums:Splitter.bg3d

 Number of Textures: 4

SAVING FILE AS: ::Level1Models.bg3d
```

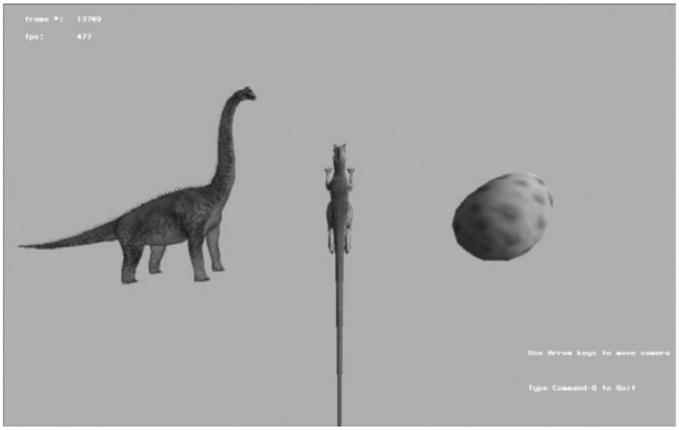
Figure 13-9: Console output from BG3D Linker

At this point you're probably asking yourself "isn't this going to fuse all of my geometry together into a massive blob of triangles and vertices?" No, because the BG3D file format has Groups, so this tool puts each separate model into its own group so that when we read this BG3D file back into our game we can separate each model out. Every 3D game I've ever written for the Mac has used one version or another of this linker utility. I like to have one large .bg3d file for each level's unique models, and then I also have a global.bg3d file that contains models that are used throughout the game.

Loading and Using BG3D Files

The sample project "BG3D.xcode" contains a new source file named BG3D.c that has all of the code needed to load and extract data from a .bg3d model file. The project has another new source file called "MetaObjects.c", and this file has all of the code needed to manage our 3D model objects. If you're familiar with Apple's old Quickdraw 3D API then this will seem very familiar because I based my object system on the one used in Quickdraw 3D. The basic idea here is that there are "objects" that contain data. The data may be a matrix, geometry data, or even groups. This sample project has a very simplified version of the object management system that I use in my games.

In the BG3D.xcode project there are two .bg3d model files that it loads in: Dinosaurs.bg3d and Eggs.bg3d. The Dinosaurs.bg3d file contains two different dinosaur models, and the Eggs.bg3d file contains three different egg models. When you run the project, you'll see several objects spinning on the screen:



```
frame #:  13709
fps:      477

Use Arrow keys to move camera

Type Command-Q to Quit
```

Figure 13-10: Three models spinning

First, the code loads the .bg3d files like this:

```
ImportBG3D("\pDinosaurs.bg3d", MODEL_GROUP_DINOSAURS);
ImportBG3D("\pEggs.bg3d", MODEL_GROUP_EGGS);
```

The `ImportBG3D()` function in bg3d.c will parse the input .bg3d file and process all of the tags to build a database of textures and meshes. All of the vertex and triangle information for each mesh is stored into a contiguous block of memory so that when we're done we can mark that block of memory for use by Vertex Array Range as discussed in Chapter 6.

To create a "meta-object" representing one of the models that we want to draw, we follow these steps: First, we create a Group Object to contain all of our object's info:

```
baseGroupObj = MO_CreateNewObjectOfType(MO_TYPE_GROUP, 0, nil);
```

After that, we want to create a transformation Matrix Object, and then put it in our Group:

```
matrixObj = MO_CreateNewObjectOfType(MO_TYPE_MATRIX, 0, &matrix);
MO_AppendToGroup(baseGroupObj, matrixObj);
```

When the .bg3d file was read in, all of the materials and meshes were stored into meta-objects, and we kept references to them in `gBG3DGroupList[]`, so to add one of those models to our Group Object we simply call `MO_AppendToGroup()` again, and pass in a reference to the model we want:

```
MO_AppendToGroup(baseGroupObj, gBG3DGroupList[bg3dGroup][bg3dModel]);
```

To draw any object, all we have to do is pass it to `MO_DrawObject()`. Our code will parse the meta-data, and process the matrix and mesh that it finds in there.

```
MO_DrawObject(gRaptorObject);
```

The details of what all this code is doing really are not important since you'll probably want to write your own object handling functions, but for the sake of getting something on the screen I wanted to give you this brief explanation of how I like to do things in my games. The code in BG3D.c and MetaObjects.c is pretty straightforward and well commented, so you should be able to read through it and figure out how everything works if you're interested.

Chapter 14: Stereo 3D

In the mid-1990's the big buzzword was "Virtual Reality", remember that? You could go to your local mall and pay $5 to put on a 20 pound headset to watch some ultra-low-rez things flying around a room for a few minutes, or, pay $20 for a 15 minute ride in a flight simulator on hydraulics. If you really had nothing better to do with your money, you could shell out a few hundred bucks on a VR headset and then pray that you could find a game that would run on it. For better or for worse, VR finally died a painful death as most of these Virtual Reality companies bit the dust. The fact is that 3D processing horsepower was nowhere near where it needed to be to drive stereo 3D in those days. The technology at the time could barely render a single frame of a scene at a decent speed, let along render two frames – one for each eye.

Types of Stereo Glasses

But now, a decade later, we have the horsepower, and maybe even a little to spare, so doing stereo 3D games is now a realistic possibility (let's just hope the term "Virtual Reality" stays dead, though). With the Mac and OpenGL there are two different ways that we can support stereo 3D: with anaglyph glasses and with shutter glasses.

Anaglyph Glasses

These are the old fashioned red-blue glasses from the sci-fi B-movies of the 1950's. There have been some advances in this field since 1950, and we now like to use red-cyan glasses instead of red-blue. Even though there's quite a bit of color distortion with these glasses, your eyes do adjust to some degree, and they're a very economical way to do stereo rendering on a home computer. The old red-blue type are still commonly found, so be sure that you are using red-cyan's since the red-blue type will not work well for what we're going to do.

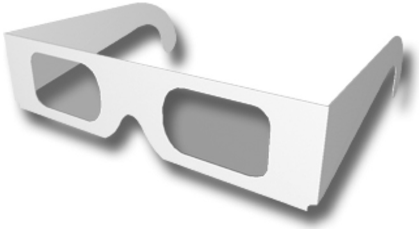


Figure 14-1: Red-Cyan Anaglyph Glasses

Shutter Glasses

These were popular in the heyday of Virtual Reality, and can still be found today. They're basically a lightweight headset that has two LCD shutter panels; one over each eye. They flicker rapidly in synchronization with the monitor to reveal the left and right views. Unfortunately, these only work on CRT displays with high refresh rates, and since most computer users are switching to LCD flat-panel displays there isn't much use for these now. That being said, the effect is pretty awesome since there's no color distortion like you get with anaglyph glasses, and Apple has put some special support into OpenGL for such hardware.

Figure 14-2: LCD Shutter Glasses

Stereo Camera Calculations

Before doing anything else we need to get a grasp of the concept behind stereo rendering. The basic idea is that we render the scene twice each frame: once for the left eye, and once for the right eye. However, doing this "correctly" is more complicated than people usually think. Everyone's first instinct is to mimic reality by having two cameras that are separated by a small distance, and have them look at a focal point. This is often called the "toe-in' method, and is shown in Figure 14-3:

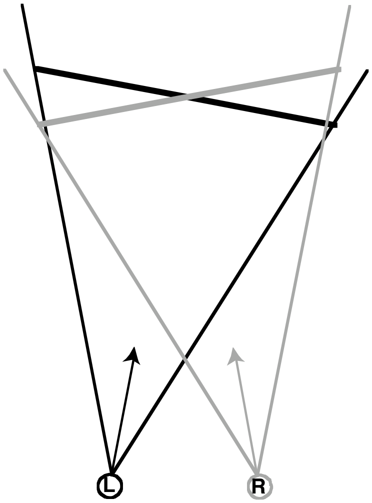


Figure 14-3: Two cameras looking at focal point

Even thought the toe-in method mimics reality (since that's what our eyes actually do in the real world), this method has serious problems in the realm of projected 3D math. The problem is evident by looking at the projection planes in the figure above. Notice that they're not parallel, but instead they form an X. This is bad because it will result in vertical parallax distortions in the rendered image. For example, suppose that there is a sphere on the left side of the scene as shown in Figure 14-4:

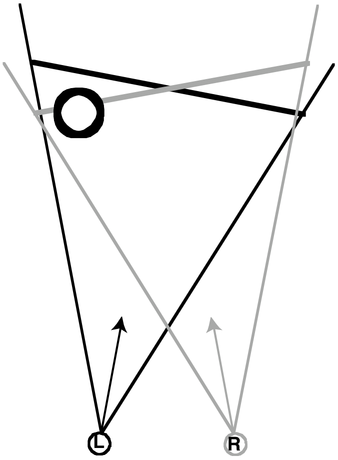


Figure 14-4: The sphere is closer to the right eye's projection plane than the left eye's

The sphere is closer to the right eye's projection plane than the left eye's projection plane, so, when the scene is rendered, the objects will be drawn at slightly different sizes. The scaling effect gets worse toward the edges of the screen. The rendered scene will look something like this:

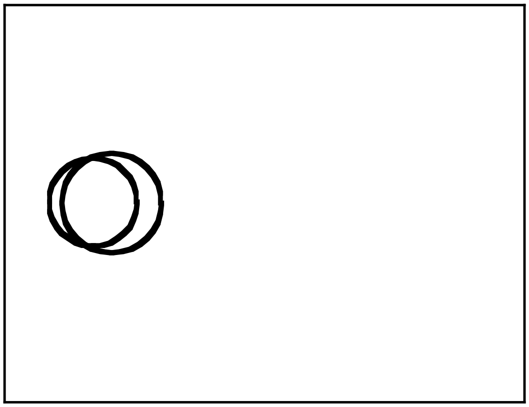


Figure 14-5: The spheres suffer vertical parallax and don't line up well

If you were to view this scene with stereo glasses you would see a 3D effect, but it would be difficult on the eyes, especially near the edges of the screen. What we want to see are two spheres which have no parallax scaling, just horizontal shifting like this:

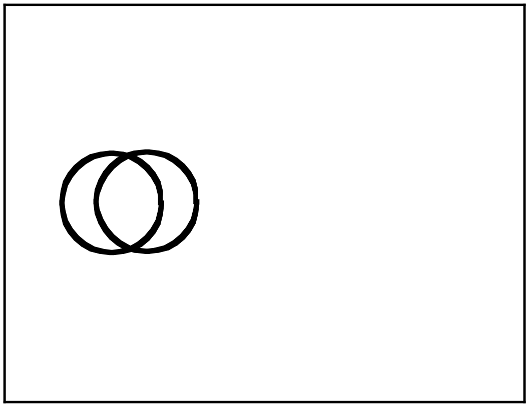


Figure 14-6: The kind of stereo projection we want to achieve

In order to achieve this we need to modify the view frustums like so:

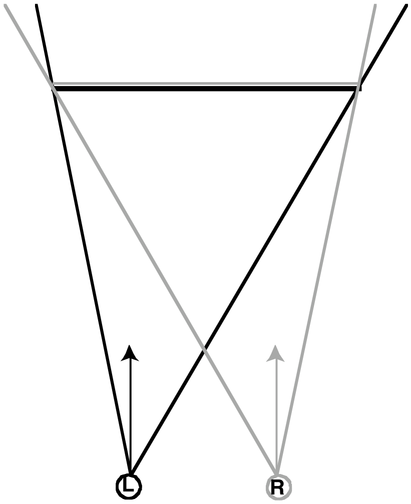


Figure 14-7: Modified view frustums to achieve a flat projection plane

As funky as those skewed view frustums look, they actually will yield excellent results! Scenes drawn like this will have very "pain-free" stereo images that look great. This is no longer a true perspective camera, so we cannot use `gluPerspective()` to easily set up the camera's projection matrix. We've now got to do some calculations to manually set our camera's frustum, so instead of this…

```
    gluPerspective (CAMERA_FOV,             // fov
                    aspect,                 // aspect
                    CAMERA_HITHER,          // hither
                    CAMERA_YON);            // yon
```

… we need to do this:

Listing 14-1:Calculating a Projection Matrix for Stereo Cameras

```
static void CalcStereoCameraFrustum(Boolean isLeftCamera)
{
    float   left, right, a, b;
    float   fov;

            /* DO FORMULA CALCULATIONS */

    fov = CAMERA_FOV / 180.0f * M_PI;       // convert FOV to radians

    aspect = (float)gGameWindowWidth/(float)gGameWindowHeight;

    a   = CAMERA_HITHER * tan(fov * .5f);
    b   = CAMERA_HITHER / STEREO_FOCAL_LENGTH;

    if (isLeftCamera)                       // left camera
    {
        left  = - aspect * a + (STEREO_CAMERA_SEPARATION * .5f) * b;
        right =   aspect * a + (STEREO_CAMERA_SEPARATION * .5f) * b;
    }
    else                                    // right camera
    {
        left  = - aspect * a - (STEREO_CAMERA_SEPARATION * .5f) * b;
        right =   aspect * a - (STEREO_CAMERA_SEPARATION * .5f) * b;
    }

            /* SET THE PROJECTION MATRIX FRUSTUM */

    glFrustum(left, right, -a, a, CAMERA_HITHER, CAMERA_YON);
}
```

There are a few new parameters introduced in this function:

STEREO_FOCAL_LENGTH
This value determines the distance from the camera where objects appear to intersect our display. Anything closer than this will appear to be popping out of the screen, and anything farther than this distance will appear to be behind the screen. As cool as it might seem to have your 3D objects hovering in front of the screen, it's usually best to have most of your scene appear behind it. Objects projected in front tend to cause a little more eye strain, and can cause that "cross-eyed" effect that you've probably experienced if you've ever been to a 3D movie.

STEREO_CAMERA_SEPARATION
This value determines how far the left and right cameras are separated. The farther apart they are, the more exaggerated the stereo 3D effect will be, but since separating the cameras farther and farther is akin to making your head bigger and bigger in the universe, the objects in the scene will seem to look like toys, and take on a surreal appearance. Moving the cameras closer together will lessen the stereo effect, but it will also make the objects seem larger since, in effect, our head is shrinking in the virtual universe.

There's a general rule in stereo imaging that says that the camera separation should be 1/30th of the distance to the focal plane. So, if your focal plane is 300 units away then your camera separation should be about 10 units. This rule is not written in stone by any means, but it is a good ratio to start with as you tweak things. Personally, I like to exaggerate the stereo 3D effect in my games, so I tend to use ratios that are more like 1:20 rather than 1:30.

This `STEREO_CAMERA_SEPARATION` parameter gets used again to calculate the coordinates of the left and right eye. Remember that we're going to be drawing each frame twice: once for the left eye and once for the right eye, and when we do this we need to actually offset our camera for each location:

Listing 14-2:Offsetting the Camera for the Left or Right Eye

```
static void OffsetStereoCameraCoord(Boolean isLeftCamera)
{
    OGLVector3D     aim;
    OGLVector3D     xaxis;
    float           sep = STEREO_CAMERA_SEPARATION * .5f;

    if (isLeftCamera)               // negate separation for left eye
        sep = -sep;

            /* CALC CAMERA'S X-AXIS */

    aim.x = gCamera_LookAt.x - gCamera_Coord.x;
    aim.y = gCamera_LookAt.y - gCamera_Coord.y;
    aim.z = gCamera_LookAt.z - gCamera_Coord.z;
    OGLVector3D_Normalize(&aim, &aim);
    OGLVector3D_Cross(&aim, &gCamera_UpVec, &xaxis);


                /* OFFSET CAMERA COORD */

    gStereoCamera_Coord.x = gCamera_Coord.x + (xaxis.x * sep);
    gStereoCamera_Coord.y = gCamera_Coord.y + (xaxis.y * sep);
    gStereoCamera_Coord.z = gCamera_Coord.z + (xaxis.z * sep);

                /* OFFSET CAMERA LOOKAT */

    gStereoCamera_LookAt.x = gCamera_LookAt.x + (xaxis.x * sep);
    gStereoCamera_LookAt.y = gCamera_LookAt.y + (xaxis.y * sep);
    gStereoCamera_LookAt.z = gCamera_LookAt.z + (xaxis.z * sep);
}
```

There's nothing particularly amazing about this code. We're simply calculating an x-axis vector for the camera, and then offsetting the camera along that vector. If it's the left camera we're dealing with then we offset it left, and if it's the right camera then we offset it to the right.

Rendering for Anaglyph Glasses

Ok, we've covered the basic math behind stereo cameras, so now let's put it to use. The sample project "AnaglyphStereoRendering.xcode" shows how to fully implement support for anaglyph glasses using the code above plus a few more items of interest. The first item of interest we need to discuss is how to actually generate an image for anaglyph viewing.

Anaglyph glasses have a red filter over the left eye, and a cyan (a mix of blue and green) filter over the right eye. When we render our scene, we need to draw everything for the left eye using only the color red, and draw everything for the right eye with only blue and green. Luckily, this is very easy to do with OpenGL because OpenGL has a simple way to turn on and off color channels during rendering. The `glColorMask()` function lets you set the state of the Red, Green, and Blue channels, so when we go to render the left eye we turn off the Green and Blue channels, leaving only the Red channel active:

```
    glColorMask(GL_TRUE, GL_FALSE, GL_FALSE, GL_TRUE);
```

The cool thing about `glColorMask()` is that when we render the right eye (which needs only the Green/Blue channels), OpenGL will not erase the Red channel that's already in the frame buffer. This means it is not necessary to draw the left eye into one buffer and the right eye into another and then composite them manually.

So, our new `OGL_DrawScene()` function looks like this:

Listing 14-3:Drawing a Scene with a Stereo Camera

```
void OGL_DrawScene(void)
{

            /* MAKE OUR CONTEXT THE ACTIVE ONE */

    aglSetCurrentContext(gAGLContext);

    OGL_UpdateVertexArrayRange();                  // update VAR memory

            /*********************/
            /* CLEAR BACK BUFFER */
            /*********************/

                            // make sure clearing Red/Green/Blue channels
    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

                            // clear back buffer
    glClear(GL_COLOR_BUFFER_BIT);


        /************************************************/
        /* DRAW SCENE IN TWO PASSES (LEFT & RIGHT EYE) */
        /************************************************/
    for (gStereoPass = 0; gStereoPass < 2; gStereoPass++)
    {
        glClear(GL_DEPTH_BUFFER_BIT);   // clear the z-buffer on each pass!

                /* SET COLOR MASK */

        if (gStereoPass == 0)
        {
                                        // red-only for left eye
            glColorMask(GL_TRUE, GL_FALSE, GL_FALSE, GL_TRUE);
        }
        else
        {
                                        // green-blue for right eye
            glColorMask(GL_FALSE, GL_TRUE, GL_TRUE, GL_TRUE);
        }

            /* OFFSET CAMERA FOR LEFT/RIGHT EYE */

        OffsetStereoCameraCoord(gStereoPass == 0);


            /* SET FRUSTUM AND OTHER CAMERA MATRICES */

        SetCameraMatrices();

                /* DRAW SCENE */

        DrawSceneGeometry();                        // draw stuff
    }


            /* END RENDER & SWAP THE BUFFER TO MAKE IT VISIBLE */

    aglSwapBuffers(gAGLContext);
}
```

When we start to draw the frame, we turn on all of the color channels so that when we clear the back buffer, it'll clear the whole thing:

```
    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
    glClear(GL_COLOR_BUFFER_BIT);
```

Then we enter a loop to cause the entire scene to be drawn twice, once for each eye. Each time we draw the scene we need to clear the z-buffer:

```
    glClear(GL_DEPTH_BUFFER_BIT);
```

Then we set the color mask to either red or cyan depending on which eye we're rendering for. On the first pass we do the left eye, so we only turn on the Red channel:

```
glColorMask(GL_TRUE, GL_FALSE, GL_FALSE, GL_TRUE);
```

And then for the second pass we do the right eye, so we draw only into the Green and Blue channels for cyan:

```
glColorMask(GL_FALSE, GL_TRUE, GL_TRUE, GL_TRUE);
```

Next, we move the camera to either the left or right eye position:

```
OffsetStereoCameraCoord(gStereoPass == 0);
```

Then we set the Projection and Model View matrices as needed for either the left or right eye camera:

```
SetCameraMatrices();
```

With everything set, we draw the scene and do the next pass if needed.

Anaglyph Color Balancing

Everything we've covered so far is all that is needed to render a stereo 3D world, but if we left it at that, our images wouldn't look too good because there's one other major issue that needs to be addressed when dealing with 3D anaglyph images: color balancing. When we render our scene, we're filtering the left eye with red, and the right eye with cyan, but what happens if we're drawing a red object such as an apple? Well, it'll look fine in the left eye since the red filter won't have much effect on the object, but it'll appear almost black in the right eye because the cyan filter will block out all of the red color in the apple. Any rendered pixel that isn't a shade of gray is going to suffer from this problem to some degree.

There is also a physiological problem in that the human eye is much more sensitive to green than any other color, so the image seen through the cyan filter will look much brighter than the image seen through the red filter. The eye is so sensitive to green that you will often see "ghosting" in the left eye as some of the green bleeds through the red filter. If you've got a good pair of anaglyph glasses and a good display then your cyan filter should block almost 100% of the red light, but there will always be a small amount of green bleeding through your red filter. Keeping this to a minimum is important, but to do so requires a darker red filter thus dimming the brightness even more.

When I made the custom anaglyph glasses for Nanosaur 2, I had our manufacturer send me gel samples for the different shades of red and cyan that they had. I picked red and a cyan gels that worked best on Apple's LCD displays, and I've been very happy with the results. Don't be mistaken. There is a *huge* difference in the quality of various anaglyph glasses. I've seen some that had so much color bleeding in both eyes that it was impossible to see any 3D images at all. You've got to be sure you've got glasses with properly tuned filters.

Additionally, the color accuracy of the phosphors on a CRT display is nowhere near as good as you get with an LCD. LCD colors are fairly pure, meaning that the red channel won't have much green or blue in it, but CRT's tend to emit stray color wavelengths in each color channel, so the red phosphor may be spewing out a fair amount of green or blue that you cannot see until you look at it through a cyan filter. Anaglyph images are best viewed on LCD displays since the color accuracy is so good, but you can use a CRT as long as you are aware that there will be more ghosting.

So, what does this all mean to us as programmers? Well, it means that we need to color balance all of the textures in our game so that there are no pixels that are severely red or cyan shifted like the red apple would be. We also need to reduce any green peaks since our eyes are too sensitive to that color.

If you do some research on the web, you'll find all sorts of complex formulas for doing anaglyph color balancing, and there are even some methods which people have patented. There is a research paper from the Computer Science Department at North Carolina State University that goes into more gritty details of anaglyph color balancing algorithms than you could ever want:

`http://research.csc.ncsu.edu/stereographics/ei03.pdf`

The fact is, however, that you can get away with some pretty simple code to do basic color balancing, and how you choose to color balance your textures is very dependent on how much color distortion you can accept. If you were to do 100% color balancing you'd end up with a grayscale texture, so you've really got to play with your code to get the look that you want. In Bugdom 2 the texture colors were so saturated that converting the textures to grayscale usually yielded better results than any attempt at color balancing, but in Nanosaur 2 the colors were less saturated, so, it was much easier to get good color balance without making everything go gray.

The following function is very similar to what was used in Nanosaur 2. It's a simple hack, but it's effective:

Listing 14-4:Doing RGB Color Balancing

```
#define RED_RATIO_ADJ       .6f
#define GREEN_RATIO_ADJ     .4f
#define BLUE_RATIO_ADJ      .8f


static void ColorBalanceRGBForAnaglyph(u_long *rr, u_long *gg, u_long *bb)
{
    long    r,g,b;                  // 8-bit color component values
    float   fr,fg,fb;               // float color component values
    float   d;
    float   lumR, lumGB, ratio;

            /* GET UNBALANCED INPUT RGB VALUES */

    fr = r = *rr;                   // get both integer and float versions
    fg = g = *gg;
    fb = b = *bb;


                /* CALC LUMINOSITY OF RED AND CYAN EYES */
                //
                // The NTSC luminance calculation is = .299r + .587g + .114b
                //

    lumR    = fr * .299f;               // red luminosity = r*.299
    lumGB   = fg * .587f + fb * .114f;  // cyan luminosity = g*.587 + b*.114


        /* BALANCE RED */

    ratio = lumGB / lumR;               // calc ratio of cyan to red
    d = fr * ratio * RED_RATIO_ADJ;     // d = adjusted red value
    if (d > fr)                         // only adjust up, not down
    {
        r = d;
        if (r > 0xff)                   // make sure doesn't overflow 8-bits
            r = 0xff;
    }


        /* BALANCE GREEN */

    ratio = lumR / lumGB;               // calc ratio of red to cyan
    d = fg * ratio * GREEN_RATIO_ADJ;   // d = adjusted green value
    if (d > fg)
    {
        g = d;
        if (g > 0xff)
            g = 0xff;
    }
```

```
        /* BALANCE BLUE */

    d = fb * ratio * BLUE_RATIO_ADJ;    // d = adjusted blue value
    if (d > fb)
    {
        b = d;
        if (b > 0xff)
            b = 0xff;
    }

    *rr = r;                            // return the new RGB values
    *gg = g;
    *bb = b;
}
```

This function takes as input the Red, Green, and Blue component values of a texture's pixel, and outputs the color-balanced values. The first thing we do is to calculate the luminosity of the Red and Cyan channels. The standard formula for calculating luminosity is:

$$Luminosity = .299r + .587g + .114b$$

So, to calculate the luminosity of Red and Cyan:

```
    lumR    = fr * .299f;
    lumGB   = fg * .587f + fb * .114f;
```

These luminosity values indicate how well balanced each eye is. If `lumR` is greater than `lumGB` then we know that this pixel will look brighter in the left eye than the right eye. What we want to do is to try and balance out the RGB components based on these luminosities, so we start with Red by calculating the luminosity ratio of Cyan to Red:

```
    ratio = lumGB / lumR;
```

We use this ratio to amplify the red component:

```
    d = fr * ratio * RED_RATIO_ADJ;
```

The constant `RED_RATIO_ADJ` is a value that we've set based on what looks good in our game. You'll want to tweak these values to make things look best for your own game. If the scene in your game has a lot of greenery like trees and bushes, then you may need to bump the `RED_RADIO_ADJ` value up to make the red channel better balanced against all that green. However, if your scene contains a lot of red lava and fire, then you might need to bump up the `GREEN_RATIO_ADJ` value.

Since Anaglyph images tend to be dim due to all the filtering, we really want to avoid lowering a pixel's brightness, so if your calibrated component value is less than the original value we don't do anything. Otherwise, we update it:

```
    if (d > fr)
    {
        r = d;
        if (r > 0xff)
            r = 0xff;
    }
```

This is repeated for the Green and Blue channels.

Black & White

In all of my games that support stereo rendering I give the user the option of playing in black & white mode where all of the grayscale textures are perfectly color balanced. This yields the sharpest possible images for anaglyphs, but obviously there's no color, therefore, things just don't look quite as interesting. Additionally, the loss of color tends to make the objects in the scene a bit less distinguishable and darker. Your bright red power-ups no longer stand out against the green grass.

The function to convert textures to grayscale looks like this:

Listing 14-5:Converting Textures to Grayscale

```
static void ConvertTextureToGrey(void *imageMemory, short width,
                                short height)
{
    long    x,y;
    float   r,g,b;
    u_long  a,lum;
    u_long  *pix32 = (u_long *)imageMemory;

    for (y = 0; y < height; y++)
    {
        for (x = 0; x < width; x++)
        {
                /* CALC LUMINANCE OF THE RGB CHANNELS */

            r = (float)((pix32[x] >> 16) & 0xff) / 255.0f * .299f;
            g = (float)((pix32[x] >> 8) & 0xff) / 255.0f * .586f;
            b = (float)(pix32[x] & 0xff) / 255.0f * .114f;
            a = (pix32[x] >> 24) & 0xff;

                /* CALC LUMINANCE OF THIS PIXEL*/
            lum = r + g + b;            // calc total luminance
            lum *= 255.0;               // convert to 8-bit byte
            if (lum > 0xff)
                lum = 0xff;


                /* CONVERT THRU OUR BRIGHTNESS CURVE */

            lum = gAnaglyphGreyTable[lum];


                /* SAVE LUMINOSITY INTO RGB CHANNELS TO MAKE GREY */

            pix32[x] = (a << 24) | (lum << 16) | (lum << 8) | lum;

        }
        pix32 += width;
    }
}
```

Once again we use the standard NTSC luminance calculation to get the luminosity values of the RGB components:

```
    r = (float)((pix32[x] >> 16) & 0xff) / 255.0f * .299f;
    g = (float)((pix32[x] >> 8) & 0xff) / 255.0f * .586f;
    b = (float)(pix32[x] & 0xff) / 255.0f * .114f;
```

Then we add all of these luminosities together to get the pixel's full luminosity value:

```
    lum = r + g + b;
```

At this point we could simply use this luminosity value as the RGB component values and be done with it, but a trick that works well to brighten up the scene is to amplify the luminosity on a curve table. This table works just like a gamma table. It's 256 entries, one for each possible value for the RGB components. We use the current luminosity as an index into the table which gives us the new luminosity:

```
    lum = gAnaglyphGreyTable[lum];
```

Listing 14-6:Generating a Grayscale Amplification Curve

```
static void BuildGreyCurve(void)
{
    short   i;
    float   f;
    f = 0;
    for (i = 0; i < 255; i++)
    {
        gAnaglyphGreyTable[i] = sin(f) * 255.0f;
        f += (M_PI / 2.0) / 255.0f;
    }
}
```

The curve that gets generated looks like this:

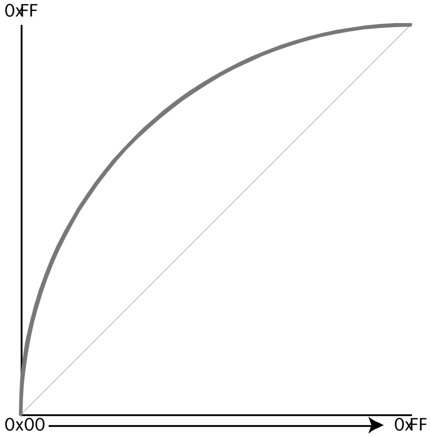


Figure 14-8: A simple grayscale amplification curve

As you can see from the curve, all of the values are bumped up in luminosity since the straight diagonal line represents the non-amplified luminosity values. The only problem with amplifying the luminosity like this is that it washes out the contrast a little, but it's the lesser of two evils.

There are other tweaks you can make to the grayscale code if you like. For example, instead of making it a true grayscale with equal Red, Green, and Blue values, you might want to lower the Green values to reduce ghosting and balance the left and right eye luminosities. If you wanted to totally eliminate the ghosting you could just set the green channel to 0x00 and then pump up the blue to compensate.

Where to Buy Anaglyph Glasses

When we shipped Nanosaur 2 we decided to include two pairs of red-cyan anaglyph glasses in each box. The total cost on this was about 60 cents per box which isn't bad considering what a good selling point this was. As mentioned earlier, we had the manufacturer send us gel samples for the different shades of red and cyan filters that they had. Once we found a pair of gels that worked best on an Apple LCD display we had them custom print up several thousand pairs for us. There are many companies out there who sell anaglyph glasses, but prices can vary wildly. The company we like to use is Rainbow Symphony:

www.rainbowsymphony.com

Even if you're only looking for a few pairs to play around with you can get our custom made glasses from them since they sell the Nanosaur 2 glasses on their site:

www.rainbowsymphony.com/nano3d.html

Figure 14-9: The custom built Nanosaur 2 anaglyph glasses

The best way to test a pair of anaglyph glasses is to do this: Load up Photoshop and draw a pure green square on the left side of the image, and then draw a pure red square on the other side. Look through the anaglyph glasses and close your right eye. Observe how much green is showing through the left filter. It should be only a little – the less the better. Then close your left eye and see if any red is visible in the right eye. There should be no red visible at all if your glasses are of good quality.

If you're on a CRT display you'll see much more ghosting of both colors in each eye, but on an Apple LCD display you should see no red at all in the right eye, and only a small amount of green in the left eye.

Rendering for LCD Shutter Glasses

Supporting LCD Shutter Glasses is not much different than supporting anaglyph glasses. The big difference is that it's difficult to find the right type of shutter glasses that are compatible with OpenGL on the Mac. The way that shutter glasses work is that the LCD panel over each eye will rapidly flicker from clear to opaque allowing the left eye to see the left image for a fraction of a second, and then the right eye to see the right image. The trick is in synchronizing the shutter glasses with the display. The method that Apple implemented for OpenGL is called "Blue-Line Sync". It's a pretty crude, yet effective method. A blue line is drawn at the bottom of the frame buffers for the left and right eyes. A short blue line for the left buffer, and a long blue line for the right buffer. The Shutter Glasses' hardware detects these lines, and uses them as triggers to open and close the LCD panels.

We're still going to draw each frame twice, once for the left eye and once for the right eye, but this time we have to draw them into two separate frame buffers. OpenGL will automatically page flip between the two frame buffers, and the shutter glasses will detect the blue-line sync signal as this happens.

To tell OpenGL that we will be doing stereo 3D with shutter glasses, we modify our Draw Context attribute list:

```
GLint   attrib32bitStereo[]     = {AGL_RGBA, AGL_FULLSCREEN, AGL_STEREO,
                                  AGL_DOUBLEBUFFER, AGL_DEPTH_SIZE, 32,
                                  AGL_ALL_RENDERERS, AGL_ACCELERATED,
                                  AGL_NO_RECOVERY, AGL_NONE};
```

This looks the same as the attribute list we've used in all of the other sample projects except that one new attribute has been added: `AGL_STEREO`. When OpenGL sees this attribute it will know to create two frame buffers, one for the left eye and one for the right eye. It should also be noted that you *must* play full-screen to use `AGL_STEREO`. You cannot play in a window, so the sample project "Shutter Glasses.xcode" doesn't have a windowed option.

The extra frame buffer needed for `AGL_STEREO` uses up a significant amount of VRAM, so, you should be very careful about trying to activate this mode on systems with very little VRAM. Back in Chapter 4 we learned how to determine how much VRAM was available on the main display, and this is a good place to use that function. If the amount of VRAM is less than 16MB then you should probably not allow `AGL_STEREO`.

Luckily, we don't need to worry about ghosting or color balancing when using shutter glasses, so we don't need to have any crazy texture whacking functions. We leave all of our textures as they are, and when we render the images we don't need to mess with any color channels. The only thing special we have to do is tell OpenGL which buffer we're drawing into:

```
        if (gStereoPass == 0)
            glDrawBuffer(GL_BACK_LEFT);
        else
            glDrawBuffer(GL_BACK_RIGHT);
```

The rest of our `OGL_DrawScene()` function looks pretty typical:

Listing 14-7:Drawing A Scene for Shutter Glasses

```
void OGL_DrawScene(void)
{

            /* MAKE OUR CONTEXT THE ACTIVE ONE */

    aglSetCurrentContext(gAGLContext);
    OGL_UpdateVertexArrayRange();               // update VAR memory


        /**********************************************/
        /* DRAW SCENE IN TWO PASSES (LEFT & RIGHT EYE) */
        /**********************************************/

    for (gStereoPass = 0; gStereoPass < 2; gStereoPass++)
    {
            /* TELL OPENGL WHICH BUFFER WE'RE WORKING WITH */

        if (gStereoPass == 0)
            glDrawBuffer(GL_BACK_LEFT);
        else
            glDrawBuffer(GL_BACK_RIGHT);

                /* CLEAR FRAME & Z BUFFERS */

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


            /* OFFSET CAMERA FOR LEFT/RIGHT EYE */

        OffsetStereoCameraCoord(gStereoPass == 0);


            /* SET FRUSTUM AND OTHER CAMERA MATRICES */

        SetCameraMatrices();

                /* DRAW SCENE */

        DrawSceneGeometry();                        // draw stuff
    }
```

```
            /* DRAW THE BLUE LINE */

    DrawBlueLineSync();


            /* END RENDER & SWAP THE BUFFER TO MAKE IT VISIBLE */

    aglSwapBuffers(gAGLContext);
}
```

You'll notice that once we're done drawing our scene we call a new function `DrawBlueLineSync()` which takes care of drawing correct blue-line sync signals into each frame buffer:

Listing 14-8:Drawing the Blue-Line Sync

```
static void DrawBlueLineSync(void)
{
    short   buffer;
    short   w = gGamePrefs.screenWidth;
    short   h = gGamePrefs.screenHeight;

            /* SET A SAFE STATE FOR DRAWING A BLUE LINE */

    OGL_PushState();
    OGL_DisableTexture2D();
    OGL_DisableBlend();
    OGL_DisableLighting();
    OGL_DisableFog();
    OGL_DisableDepthTest();


            /* DRAW A DIFFERENT BLUE LINE INTO THE LEFT & RIGHT BUFFERS */

    for (buffer = GL_BACK_LEFT; buffer <= GL_BACK_RIGHT; buffer++)
    {
        GLint matrixMode, vp[4];

                /* SET THE BUFFER TO WORK WITH */

        glDrawBuffer(buffer);

                /* SET VIEWPORT & MATRICES */

        glGetIntegerv(GL_VIEWPORT, vp);
        glViewport(0, 0, w, h);

        glGetIntegerv(GL_MATRIX_MODE, &matrixMode);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glScalef(2.0f / w, -2.0f / h, 1.0f);
        glTranslatef(-w / 2.0f, -h / 2.0f, 0.0f);


                /* DRAW THE SYNC LINE */

        OGL_SetColor4f(0.0f, 0.0f, 0.0f, 1);    // set color to black
        glBegin(GL_LINES);                      // Erase the line to black
        glVertex3f(0.0f, h - 0.5f, 0.0f);
        glVertex3f(w, h - 0.5f, 0.0f);
        glEnd();

        OGL_SetColor4f(0.0f, 0.0f, 1.0f, 1);    // set color to blue
        glBegin(GL_LINES);
        glVertex3f(0.0f, h - 0.5f, 0.0f);
        if(buffer == GL_BACK_LEFT)
        {
                    // draw a short blue line for the left buffer
            glVertex3f(w * 0.30f, h - 0.5f, 0.0f);
        }
        else
        {
                    // draw a longer blue line for the right buffer
            glVertex3f(w * 0.80f, h - 0.5f, 0.0f);
        }
        glEnd();
    }

    OGL_PopState();
}
```

One other issue to be aware of is that you should not do gamma fades when in `AGL_STEREO` mode. If the gamma is dark, then the blue line will obviously not be very blue, thus, the shutter glasses won't have anything to sync from. You've got to keep the gamma at full brightness so that the blue line is always visible to the hardware.

Shutter Glasses Hardware

The effectiveness of the shutter glasses is entirely dependant on the refresh rate of your monitor. LCD displays are typically only 60hz so they cannot be used with LCD glasses. The flicker would be unbearable. The minimum refresh rate before you go insane from the flicker is around 75hz, but realistically you want to be at 100hz to 120hz where the flicker is almost completely undetectable. Since no LCD does this, you've got to have a good CRT display.

There are many brands of shutter glasses out there, and most of them are very inexpensive (you can get a pair for under $100). However, none of those $100 pairs will work on the Mac because they do not do blue-line syncing. Some pairs require special drivers and special coding to trigger the shutters, while others do really lame things like interlacing to sync everything. The only company that currently manufactures and sells shutter glasses with controllers that on with blue-line sync is the Stereo Graphics Corporation:

www.stereographics.com

Their Crystal Eyes shutter glasses will cost you well over $1000 for a single pair! Ouch! It's unlikely that you're going to go out and drop that kind of money for a gimmick like stereo 3D gaming, but that's the only easy option right now. If you're a technically inclined individual, however, there is another option, but it's ugly. Someone has figured out how to build their own blue-line sync controller that will work with cheap shutter glasses, and they've posted an article to the web:

http://aberco.free.fr/3d.html

You've got to speak French, have a lab and an electrical engineering degree to figure out how to do this, but if you're looking for a challenge… there ya go. You can probably build the whole thing for less than $50, and I've been told that it works great with our game, Nanosaur 2, which supports `AGL_STEREO`.

Fun with Anaglyphs

Ok, to wrap up this chapter I wanted to quickly talk about something that's a bit off the topic of video games, but still on the topic of stereo 3D images. If you think that the stereo 3D effects in OpenGL are cool, there are other stereo things that you can do with your Macintosh that are even cooler! It's actually quite easy to make your own stereo 3D photographs using just a regular digital camera and a freeware utility called "AnaBuilder". AnaBuilder is a totally awesome utility that will take left and right digital photos, and merge them together to build anaglyph images. It can be downloaded here:

http://anabuilder.free.fr/download/MacOsX/

Taking Stereo Photos

To use this tool you first have to take some stereo photos with your digital camera. All you've got to do is pick a subject that you want to make 3D, and snap a picture of it. Then move the camera to the right just a few inches, and snap another picture while being careful not to change the orientation of the camera. This is a good time to use that 1 to 30 ratio that we discussed earlier. Only move the camera 1/30th of the distance to the nearest object in your scene. So, if the closest object you are photographing is, say, a tree that is 30' away from you, then move the camera 1' to the right when taking the second picture. If the subject is closer like, say, the stapler on your desk which is only 2' away, then only move the camera an inch.

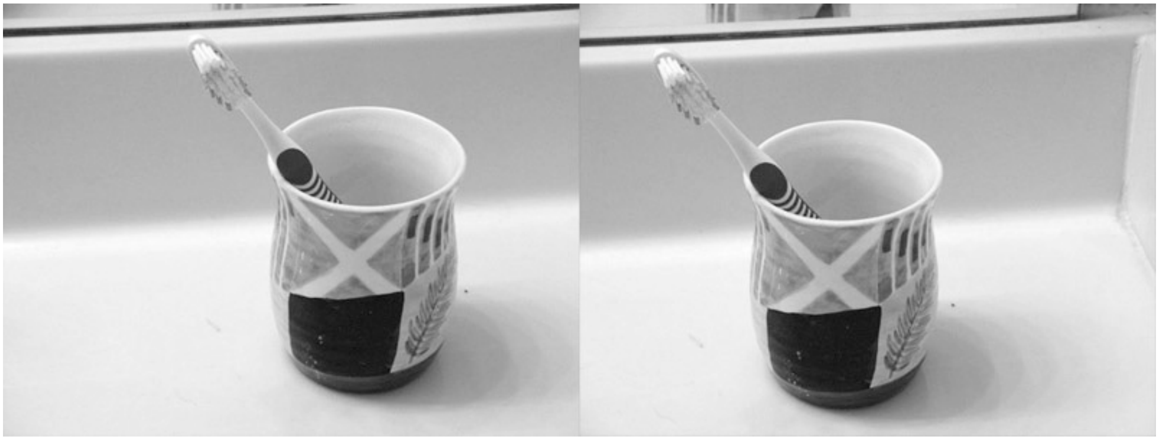
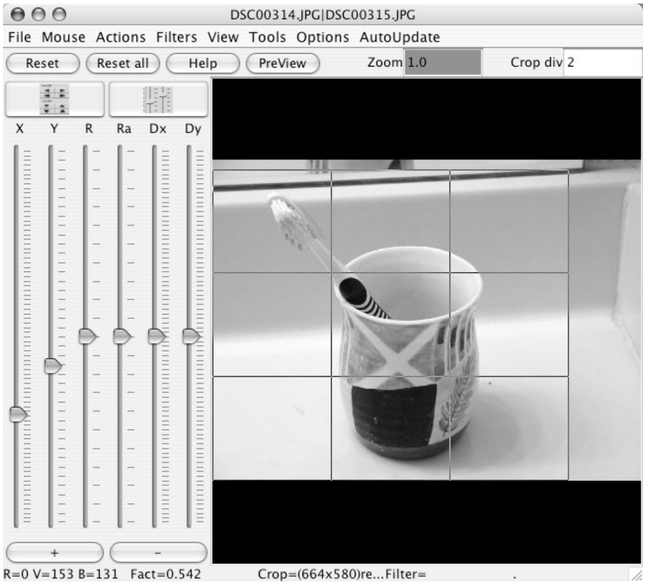


Figure 14-10: Left and right photos of a toothbrush

Next, load the two images into Photoshop and shrink them down to a reasonable size like 1024x768. AnaBuilder is a Java application, so it's a little sluggish. Trying to load a giant 5 mega-pixel image into it is just asking for trouble. After you've done this, launch AnaBuilder and load the left and right images from AnaBuilder's File menu. Note that all of this utility's menus are in the AnaBuilder window, not in the usual menu bar.

Aligning the Photos in AnaBuilder

There are countless options in AnaBuilder for aligning the images, applying filters, altering the gamma, etc, but the two most important things are the alignment controls and the color balancing window. First, we need to align the two images to form an anaglyph, and the best way to do it is with the slider controls. Below the Reset button in the AnaBuilder window you'll see a button with an icon representing slider controls on it. Click it to bring up the slider controls, and then start moving things around to get the two images lined up how you want:



Zoom 1.0

Crop div 2

R=0 V=153 B=131 Fact=0.542 Crop=(664x580)re...Filter=

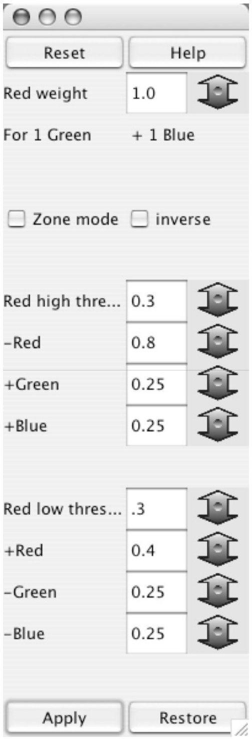
Figure 14-11: Drag the alignment sliders to get the left & right images lined up

You'll need to be wearing your red-cyan anaglyph glasses as you do this in order to see how things are looking.

When you moved the camera from left to right to take these two pictures, odds are that you wobbled a little and that the pictures are slightly out of rotation. There are slider bar controls to adjust the rotation, but one of the nicest features of AnaBuilder is the AutoFit feature. Once you've got the images roughly lined up, you can select AutoFit from the Actions menu. It may take a little while (15 to 30 seconds) to perform its calculations, but this feature will usually do a very good job of making the two images line up nicely. Once it's done performing the AutoFit, you can tweak the X slider to change the depth of the stereo image.

Color Balancing in AnaBuilder

The next step is to tweak the color balancing. I've found that the default color filter settings on AnaBuilder are way too heavily shifted toward red. The red eye is always way too bright compared with the cyan eye, but you can tweak this in the Red/Cyan Desaturation Filter which is found in the Filters menu:



Red weight	1.0
For 1 Green	+ 1 Blue
☐ Zone mode	☐ inverse
Red high thre...	0.3
-Red	0.8
+Green	0.25
+Blue	0.25
Red low thres...	.3
+Red	0.4
-Green	0.25
-Blue	0.25

Figure 14-12: The AnaBuilder Red/Cyan Desaturation Filter

There's no real rhyme or reason to how you should tweak these values. Just start playing with them until the left and right eyes seem well balanced, and the 3D image is clear. I usually lower the "Red weight" value, and raise the "-Red" value to equalize things a bit, but varies for each image I'm working with. For example, if your scene has a lot of green trees in it, then you may need to boost the red instead of decreasing it since the scene may be too saturated with green. You can spend 20 minutes playing with these filter values before you're 100% satisfied with the results.

Slide Bar Attachments for Your Tripod

If you really get into the art of making anaglyphs you will probably want to invest in a Slide Bar. A slide bar is an attachment for a tripod which lets you easily slide a camera from side to side. This makes life a lot easier since the camera stays perfectly oriented as you take your pictures. I've got a small 8" slide bar that cost me about $30 and it works for just about everything except for large panoramic shots which need more separation than 8". Slide bars are very easy to find on the web, and they come in all sizes. Just do a search for “anaglyph slide bar” and you’ll get lots of results.

You can also just do a search on the web for “anaglyph pictures,” and you’ll be amazed at what comes up. There are some truly spectacular homemade anaglyph images out there. I highly recommend checking them out. Here’s a short list of some worthy sites:

www.marsunearthed.com

www.fotocommunity.de/pc/pc/pcat/41027

www.alpix.com/nice/htmlen/pictstereo.htm

www.geocities.com/Paris/Parc/4239/lesGrands.htm

Chapter 15: Networking

I've done a few networked games in my career, and after the last one I vowed never to do another one again. That was back in the days of OS 9 when networking was significantly more difficult than it is today on OS X since we had Net Sprocket to deal with (not one of my favorite Sprockets). These days we have two separate technologies that make networking a much easier task, but the fact remains that networking is a tricky thing, and you should expect a lot of tech support incidents if your games support it.

The "Networking.xcode" sample project has a simple networking implementation that lets you either Host a network game, or Join a network game. When the game starts, each player can move a colored sphere around the screen and it'll all be networked together. It's simple, but it works, and it uses all of the basic OS X networking technologies that I am going to discuss in this chapter. The two technologies I'm referring to are Rendezvous and BSD Sockets. Rendezvous is used to allow players to find each other on the network. It's like announcing to the world "Hey I'm here! Anyone want to play?" BSD Sockets are what we'll use to actually transfer data between players in a network game. Sockets are part of OS X's UNIX core, so the function calls are a bit archaic, but there's a plethora of information that you can find on the web dealing with it.

Rendezvous

As previously stated, Rendezvous is what we use so that players can locate each other on a network. It provides an easy way for a Host to advertise a game, and for Clients to locate that Host. When you start a networked game you need to decide if you're going to be the Host or if you want to be a Client. The Host is the player who announces a new game to the world, and is responsible for managing all of the players. The Client is a player who locates a Host's game and joins it. As a general rule, the person with the fastest computer should always be the Host since most Host implementations require a little extra processing.

Hosting a Game

To advertise a game as a Host we'll need a new function:

Listing 15-1:Using Rendezvous to Advertise a Game

```
static void AdvertiseOurGame(void)
{
    CFNetServiceClientContext   context = {0, nil, nil, nil, nil };
    CFStreamError               error;
    Boolean                     registered;

                /* CREATE A NET SERVICE */
                //
                // Pass pass "" as the name so that OS X will use the
                // computer's name, and if there is a name collision
                // it will automatically handle it.
                //

    gMyRVService = CFNetServiceCreate(kCFAllocatorDefault,
                                CFSTR(""),          // use default domain
                                kServiceType,       // service type
                                CFSTR(""),          // use computer's name
                                MY_PORT_NUM);       // port #

    if (gMyRVService == nil)
        DoFatalAlert("\pCFNetServiceCreate() failed!");


                /* REGISTER/PUBLISH THE SERVICE ASYNCHONOUSLY */

    CFNetServiceSetClient(gMyRVService, AdvertisingCallback, &context);
    CFNetServiceScheduleWithRunLoop(gMyRVService,
                                    CFRunLoopGetCurrent(),
                                    kCFRunLoopCommonModes);

    registered = CFNetServiceRegister(gMyRVService, &error);

                /* HANDLE ERRORS */

    if (!registered)
    {
        CFNetServiceUnscheduleFromRunLoop(gMyRVService,
                                        CFRunLoopGetCurrent(),
                                        kCFRunLoopCommonModes);
        CFNetServiceSetClient(gMyRVService, nil, nil);
        CFRelease(gMyRVService);

        DoFatalAlert("\pCFNetServiceRegister() failed!");
    }
}
```

Rendezvous is the trademark name of this so-called “zero-configuration” network technology, but the function calls are all `CFNetService` calls. The first call we make will create a new network service for us:

```
    gMyRVService = CFNetServiceCreate(kCFAllocatorDefault,
                                CFSTR(""),          // use default domain
                                kServiceType,       // service type
                                CFSTR(""),          // use computer's name
```

```
                                     MY_PORT_NUM);        // port #
```

The parameters we pass to `CFNetServiceCreate()` describe our game to the network. The constant `kServiceType` is a string defined like this:

```
#define kServiceType    CFSTR("_mygamename._tcp.")
```

This is the most important parameter because this is the string that other Clients will search for when looking for games to join. The first part of the string identifies our game, and it must be preceded with an underscore character as in the example above. The second part of the string is the connection type - in this case it's a TCP/IP connection. We could this to a UDP network connection by putting "_udp." at the end, but for the examples in this book we're going to stick with just TCP/IP connections.

The fourth parameter to `CFNetServiceCreate()` is the name that we want other players to see when they join the game. In practice, you would want the user to be able to name their games, like "Brian's Battle Arena," but in our example we pass an empty string. The empty string tells the OS to assign the computer's default name to the service. The default name is whatever you've named your computer in the Sharing Preferences pane. If the name you've chosen is already in use, then you'll get a "name collision error", but by letting the OS assign the default computer name you'll avoid this because the OS detects any name collisions and appends a number to the end of the name to make it unique. For example, if there are two computers named "My Computer" that are trying to host a net game, then the second one to start hosting will automatically be renamed to "My Computer.1".

The final parameter sent to `CFNetServiceCreate()` is the port number that we want to use to play over. This value is completely arbitrary, but to be safe you should use values over 20,000 since the OS reserves many lower port numbers.

After we've created this new Network Service object, we need to register it so that it will start advertising itself:

```
CFNetServiceSetClient(gMyRVService, AdvertisingCallback, &context);
CFNetServiceScheduleWithRunLoop(gMyRVService,
                                CFRunLoopGetCurrent(),
                                kCFRunLoopCommonModes);

registered = CFNetServiceRegister(gMyRVService, &error);
```

The `CFNetServiceSetClient()` call is used to set a callback function that will handle any errors that occur. Typically, the only error you're going to ever get would be a name collision error, and those will only occur if you decided not to let the OS set the Service name as explained above.

Next, we install the Service into our application's run loop. This way, as we sit in our Host Game dialog, the OS can be updating this service as it needs. The final call to `CFNetServiceRegister()` gets it all going, so now we just sit back and wait for other players to find us.

Locating Hosts

If we're looking to join a game as a Client, then we need to set up Rendezvous to locate any games being advertised on the network. To do this, we create a Network Browser Service with the following function:

Listing 15-2:Using Rendezvous to Locate Hosts Being Advertised

```
static void StartBrowsingForHosts(void)
{
    CFNetServiceClientContext clientContext = { 0, nil, nil, nil, nil };
    CFStreamError   error;
    Boolean         result;
    OSStatus        err = noErr;


    gNumHostServices = 0;               // clear the list of hosts


                /* CREATE A NETWORK BROWSER OBJECT */

    gServiceBrowserRef = CFNetServiceBrowserCreate(kCFAllocatorDefault,
                                            HostBrowserCallback,
                                            &clientContext);
    if (gServiceBrowserRef == nil)
        DoFatalAlert("\pCFNetServiceBrowserCreate() failed!");


                /* INSTALL THE BROWSER */

    CFNetServiceBrowserScheduleWithRunLoop(gServiceBrowserRef,
                                        CFRunLoopGetCurrent(),
                                        kCFRunLoopCommonModes);


            /* START SEARCHING THE DOMAIN FOR SERVICES */

    result = CFNetServiceBrowserSearchForServices(
                            gServiceBrowserRef,
                                CFSTR(""),      // in default domain
                                kServiceType,   // look for our service type
                                &error);

                /* DID SOMETHING GO WRONG? */

    if (!result)
    {
        CFNetServiceBrowserUnscheduleFromRunLoop(gServiceBrowserRef,
                                                CFRunLoopGetCurrent(),
                                                kCFRunLoopCommonModes);
        CFRelease(gServiceBrowserRef);
        gServiceBrowserRef = nil;
    }
}
```

Once again, we start by creating a Network Service object, but this time we're creating a Browser service:

```
    gServiceBrowserRef = CFNetServiceBrowserCreate(kCFAllocatorDefault,
                                                HostBrowserCallback,
                                                &clientContext);
```

`CFNetServiceBrowserCreate()` installs a callback function that will be called whenever a Host is located or if a Host leaves the network. We'll discuss that callback function in a moment, but first let's continue setting up our Browser service:

```
    result = CFNetServiceBrowserSearchForServices(
                                gServiceBrowserRef,
                                CFSTR(""),      // in default domain
                                kServiceType,   // look for our service type
                                &error);
```

Here we pass our `kServiceType` string, which, as you remember, contains the name of the Service that the Host registered earlier. The Browser object will start searching the network for Net Services that match that string. When something is found, our callback is called:

Listing 15-3:The Browser Callback

```
static void HostBrowserCallback(CFNetServiceBrowserRef browser,
                                CFOptionFlags   flags,
                                CFTypeRef       domainOrService,
                                CFStreamError*  error,
                                void*           info)
{
    if (flags & kCFNetServiceFlagIsDomain)
        return;
    if (flags & kCFNetServiceFlagRemove)
        LostAHost((CFNetServiceRef)domainOrService);
    else
        FoundAHost((CFNetServiceRef)domainOrService);
}
```

Here, we determine if the callback occurred because a new network Host was located, or if one that was already located has gone away. To do this, we simply check the `flags` variable and see if `kCFNetServiceFlagRemove` is set. If not set then we add this Host to our list of available Hosts, or if it is set then we need to remove that Host from the list.

Here is the `FoundAHost()` function that will extract the Host's name and IP address, and then store it into our list:

Listing 15-4:Extracting Information About a Host Service

```
static void FoundAHost(CFNetServiceRef service)
{
    Boolean     status;

            /* SEE IF WE'VE GOT TOO MANY HOSTS */

    if (gNumHostServices >= MAX_HOSTS_TO_FIND)
        return;


                /* GET THE SERVICE'S NAME */

    CFStringRef name = CFNetServiceGetName(service);

    CFStringGetCString(name,
                        gHostList[gNumHostServices].name,
                        MAX_NAME_SIZE,
                        kCFStringEncodingMacRoman);


            /* GET THE SERVICE'S PORT ID & IP ADDRESS */

    gHostList[gNumHostServices].portNum =
        GetIPAddressOfHost(service, gHostList[gNumHostServices].ipAddress);

    gNumHostServices++;


            /* UPDATE JOIN-GAME DIALOG'S HOST MENU */

    BuildHostListMenu();
}
```

The function `CFNetServiceGetName()` easily extracts the name string for the Host, but getting the IP address and port # are a little more complicated, so we need another new function:

Listing 15-5:Getting the IP Address & Port Number of a Host

```
int GetIPAddressOfHost(CFNetServiceRef service, char *outString)
{
    int             portNum;
    CFArrayRef      addrList;
    CFDataRef       addrDataRef;
    struct sockaddr *theIPAddr;

                /* RESOLVE THE ADDRESS */

    CFNetServiceResolve(service, nil);


                /* GET THE ADDRESS DATA */
                //
                // This actually returns an array of addresses, but since
                // we know that we're only looking for a single IP address
                // we just check the first array entry.
                //

    addrList = CFNetServiceGetAddressing(service);

                                    // get data from 1st element in array
    addrDataRef = CFArrayGetValueAtIndex(addrList, 0);

                /* GET A POINTER TO THE IP ADDRESS */
                //
                // The IP address is in the sockaddr structure.
                // The port ID # is in the first two bytes of the sa_data
                // list, and the remaining 4 bytes contain the IP address
                // 2.3.4.5
                //

                    // get ptr to the sockaddr data in the CDDataRef object
    theIPAddr = (void *)CFDataGetBytePtr(addrDataRef);


                /* CONVERT THE IP NUMBER TO A C STRING */

    ConvertIPAddressToString(theIPAddr, outString);


                /* CALCULATE THE PORT ID # */
    portNum = (u_long)(theIPAddr->sa_data[0]) << 8;
    portNum |= (u_long)(theIPAddr->sa_data[1]);


    return(portNum);
}
```

The first step in getting the IP address from the Host is to resolve the Service's address:

```
    CFNetServiceResolve(service, nil);
```

Next we can get the address data out of it:

```
    addrList = CFNetServiceGetAddressing(service);
```

This function actually returns a `CFArray`, but it's an array with only one element, so we get the address like so:

```
    addrDataRef = CFArrayGetValueAtIndex(addrList, 0);
```

Different network connection types will return different address formats. We set up our network as a TCP/IP network by putting "_.tcp." at the end of our Service Type string. This means that we know we're getting an IP address here, and IP addresses are stored in a `sockaddr` structure. The `CFDataGetBytePtr()` function will return a pointer to the `sockaddr` value:

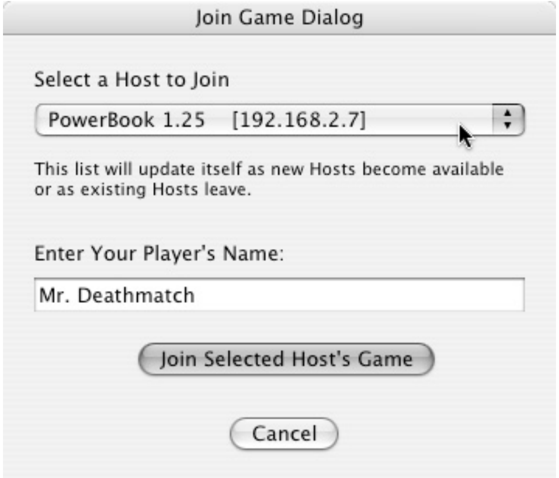
```
    theIPAddr = (void *)CFDataGetBytePtr(addrDataRef);
```

The `sockaddr` structure contains an array of bytes where the first two bytes are the socket number, and the next four bytes are the four values that make up an IP address, such as 168.0.1.4. We'd like to convert those four bytes into an actual IP address string, so we've written yet another function to do this:

Listing 15-6:Converting an IP Address into a String

```
void ConvertIPAddressToString(struct sockaddr *ipAddr, char *outString)
{
    int j,i;

    j = 0;
    for (i = 2; i <= 5; i++)             // the IP addr is in bytes 2,3,4,5
    {
        Str32   s;
        UInt8   num = ipAddr->sa_data[i];   // get this # of the IP address
        NumToString(num, s);                // convert # to Pascal string
        p2cstrcpy(&outString[j], s);        // convert & copy to C string
        j += s[0];
        if (i == 5)
            outString[j] = 0x00;            // add 0x00 to end of C string
        else
            outString[j++] = '.';           // insert a "."
    }
}
```

Even if you only have two computers on your network, it's possible that the `HostBrowserCallback()` will get called multiple times, each time appearing to have found a new Host. This happens if there are multiple connection pathways on your network, for example, you might have both a wireless Airport connection and an Ethernet connection. Or, there may even be a Firewire network connection as well. When you've got a setup like this you may see the same Host multiple times, however, each one is going to have a different IP address. Most network applications would look for these duplicates and eliminate them, however, I prefer to show all of them to the user and let him or her choose which connection to use. The Host pop-up menu in our sample project's Join Game dialog shows the name of the Host's game along with the Host's IP address:



Select a Host to Join

PowerBook 1.25 [192.168.2.7]

This list will update itself as new Hosts become available or as existing Hosts leave.

Enter Your Player's Name:

Mr. Deathmatch

Figure 15-1: The Host's IP address is shown in the list

Showing the IP address may assist the player in choosing the fastest network connection type. If you've got a gigabit Ethernet connection and an Airport connection, which would you want to choose? The trick is that you've got to be able to recognize IP addresses. If it starts with the number "10" then it's probably an Airport network.

BSD Sockets

That's all there is to advertising your game to other players, and having those players locate you. Now we need to actually set up a connection between the Host and the Clients, and start sending data between them. All communication is done with Sockets, and a Socket is basically like a telephone. You've got a telephone in your house, and your friend has a telephone in his house. If your friend knows your telephone number, he can dial you up. Once you pick up on your end, a connection has been established, and you can start talking. This is exactly how BSD Sockets work.

Listening for Connections

At the same time that we call `AdvertiseOurGame()` to announce our game to the network using Rendezvous, we also need to create a Listener Socket that will listen for incoming calls from any Clients who've seen us and wish to join our game.

Listing 15-7:Creating a Listener Socket

```
static void CreateHostsListenerSocket(void)
{
    struct sockaddr_in  socketAddr;

            /* CREATE SOCKET */

    gHostListenerSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (gHostListenerSocket < 0)
        DoFatalAlert("\psocket() failed!");


        /* SET SOCKET ADDR INFO */

    bzero( &socketAddr, sizeof(socketAddr) );       // clear addr

    socketAddr.sin_family   = AF_INET;              // internet address
    socketAddr.sin_port     = htons(MY_PORT_NUM);   // arbitrary port ID #


                // allow connection on any type of network:
                // (firewire, ethernet, airport, etc)

    socketAddr.sin_addr.s_addr  = htonl(INADDR_ANY);


            /* BIND THE SOCKET */
            //
            // If the port ID is already being used then bind will fail.
            //
    if (bind(gHostListenerSocket, (struct sockaddr *)&socketAddr,
    sizeof(socketAddr)) < 0)
    {
        close(gHostListenerSocket);
        DoFatalAlert("\pbind() failed.");
    }


            /* MAKE SOCKET LISTEN-ONLY */

    listen(gHostListenerSocket, MAX_PLAYERS);   // max # of queued connects
}
```

All Sockets are created with a call to `socket()`:

```
    gHostListenerSocket = socket(AF_INET, SOCK_STREAM, 0);
```

Here we pass `AF_INET` which tells the Socket to use Internet addressing, and then `SOCK_STEAM` which indicates that it's a streaming socket.

To assign this Socket to our network port, we first set up some addressing information and then call `bind()` to bind the Socket to a port. You may notice that we use some strange looking functions when we set certain values in the `socketAddr` structure. The `htons()` and `htonl()` functions are used to make sure the "endian" order of the bytes is correct. Remember that the Mac uses big-endian ordering for values, yet the PC uses little-endian ordering. Calling this function ensures that the bytes making up these address values are in the correct order.

Once the Socket has been bound to the port, we call `listen()` which will cause the Socket to start checking for incoming connection requests. This is like plugging in the telephone, and now you're waiting for someone to call you. In a little while we'll discuss how a Client would "dial up" a Host, but for now let's just assume that there's someone out there "dialing" our IP address, asking for a connection. How do we pick up the phone? In our Host Game dialog, we continuously call the following function to see if there is anyone calling us:

Listing 15-8:Checking for Connection Requests

```
int CheckForConnectionRequestFromClient(void)
{
    int connectionSocket;

            /* SEE IF THERE'S A CONNECTION REQUEST WAITING FOR US */

    if (!IsDataWaitingToBeRead(gHostListenerSocket))
```

```
        return(-1);


            /* ACCEPT A CONNECTION (IF ANY) */

    connectionSocket = accept(gHostListenerSocket, nil, nil);

    return(connectionSocket);
}
```

To accept a connection all we need to do is call `accept()`, however, this is a “blocking” function. That means that if we call it, it will just sit there until a connection request is received. That would be bad. So, before calling `accept()` we need to determine if there is a connection request actually waiting for us on the Socket:

Listing 15-9:Determine if a Socket has Data Waiting to be Read

```
Boolean IsDataWaitingToBeRead(int socket)
{
    struct fd_set       connectionSet;
    struct timeval      timeout;
    int                 result;


            /* SET TIMEOUT & SET VALUES */

    timeout.tv_sec          = 0;
    timeout.tv_usec         = 0;            // don't block at all

    FD_ZERO(&connectionSet);                // use macro to clear the set
    FD_SET(socket, &connectionSet);         // use macro to set socket


        /* CALL SELECT TO SEE IF THERE'S DATA WAITING ON THE SOCKET */


    result = select(socket+1,
                    &connectionSet,
                    nil,
                    nil,
                    &timeout );             // pass timeout info


            /* CHECK RESULTS */

    if (result < 0)
        return(false);

    if (result > 0)                     // was there an event on the socket?
    {
```

```
                                        // is there data waiting to be read?

        if (FD_ISSET(socket, &connectionSet))
            return(true);
        else
            return(false);
    }

    return(false);
}
```

This function is a bit cryptic, but what it does is quite simple. We call the `select()` function to determine if there is data on a Socket, but it takes a few interesting input parameters. First, we have to create a "connection set" that contains a list of Sockets that we want to test. In our case, we only want to test one Socket, so we set up `connectionSet` like this:

```
    FD_ZERO(&connectionSet);                // use macro to clear the set
    FD_SET(socket, &connectionSet);         // use macro to set socket
```

These are two macros used to first zero out the set, and then to put a value in it. The one value being set is our `socket` value.

Note that the first parameter sent to `select()` is `socket + 1`. Instead of setting this value to the number of entries in the connection set, this value is always set to your Socket number plus one. Strange, I know, but that's how some genius decided this should work.

The other parameter sent to `select()` is a `timeout` value. This indicates how long we want the `select()` function to wait for any data to arrive. In our case, we don't want it to wait at all, so we set the `tv_sec` and `tv_usec` values to 0.

With those parameters set, `select()` can now check if there's data waiting for us on our Socket. If the result returned is negative, then an error occurred. If the result is 0 then no data is waiting us, but if it's greater than 0 that's still no guarantee that there is data. Well, that's not totally true. In our case a return value of 1 would probably indicate that there's data waiting on the Socket, but to be totally sure, we call another macro to test it:

```
        if (FD_ISSET(socket, &connectionSet))
            return(true);
        else
            return(false);
```

If `IsDataWaitingToBeRead()` returns `true` then we know there's a connection request waiting for us on our Socket, so it's safe to call `accept()`. The `accept()` call returns a new communication Socket to us, and this is the Socket that we'll use from now on to send and receive data between the Host and the Client. The Listener Socket is no longer needed once all of the Clients have joined the Host's game and you're ready to play. That is, of course, unless you want to allow new Clients to join the game even while it's playing. You can close down any Socket by calling `close()`.

Requesting a Connection

So, we know how to pick up the phone, but how do we place the call? The first step was locating the Host and extracting its IP address, which is what we did earlier. That was like looking in the Yellow Pages for a plumber that you like. The IP Address that we got for the Host was like getting that plumber's phone number. Now the Client needs to create a Socket, and issue a connection request to the Host.

Listing 15-10: Sending a Connection Request to the Host

```
static void SendConnectionRequestToHost(char *ipAddrString)
{
    struct sockaddr_in  hostAddr;


        /* CREATE A CONNECTION SOCKET */
        //
        // This creates a TCP/IP streaming socket.
        //

    gConnectionToHost = socket(AF_INET, SOCK_STREAM, nil);


        /* SET ADDRESS INFO OF HOST THAT WE WANT TO CONNECT TO */

    bzero(&hostAddr, sizeof(hostAddr));     // clear address info

    hostAddr.sin_family = AF_INET;          // internet
    hostAddr.sin_port = htons(MY_PORT_NUM); // port #

    inet_pton(AF_INET, ipAddrString,        // convert IP string to in_addr
            &hostAddr.sin_addr);


        /* ISSUE THE CONNECTION REQUEST */
        //
        // connect() only issues the connect request to the server, but
        // it's up to the host to accept() it.
        //

    if (connect(gConnectionToHost, (struct sockaddr *)&hostAddr,
             sizeof(hostAddr)) < 0)
        DoFatalAlert("\pconnect() failed!  The host vanished!");


            /* SEND A MESSAGE TO THE HOST TO TELL IT WHAT OUR NAME IS */
            //
            // gMyName is a c-string which we acquired in
            // the Join Game dialog
            //

    if (WriteNetData(gConnectionToHost, gMyName, MAX_NAME_SIZE) <= 0)
        DoFatalAlert("\pWriteNetData() failed!");
}
```

We create a connection Socket with a call to `socket()`:

```
    gConnectionToHost = socket(AF_INET, SOCK_STREAM, nil);
```

Next, we set the address that we want this socket to connect to:

```
    hostAddr.sin_family = AF_INET;          // internet
    hostAddr.sin_port = htons(MY_PORT_NUM); // port #

    inet_pton(AF_INET, ipAddrString,        // convert IP string to in_addr
                &hostAddr.sin_addr);
```

The address's family is set to `AF_INET` since it's an Internet connection that we're making, and then we set the port number to `MY_PORT_NUM`. We could also set the port number to the value that was returned by `GetIPAddressOfHost()` earlier, but since we're hard-coding our port numbers in our sample project, we might as well just use the constant.

The IP address is set with the cryptic function `inet_pton()`. This converts the IP address string into the format that `hostAddr` needs.

To issue the connection request we simply call `connect()`. Even though the Host hasn't had time to `accept()` that request yet, we can go ahead and start sending data which will get queued up on the Host's end. The first thing that we want to send the Host is our player name. This is just a C string containing the name that the player wants to be seen as.

Sending Data

To send data from a Client to the Host or from the Host to a Client, we'll want to call this new function:

Listing 15-11: Sending Data

```
int WriteNetData(int socket, void *buffer, int numBytes)
{
    Ptr     bytes = buffer;         // Ptr to buffer
    int     count, n;

    n = count = 0;

    while (count < numBytes)        // loop until we've sent all the bytes
    {
        n = send(socket, bytes, numBytes - count, 0);   // write data

        if (n > 0)                  // we wrote some bytes
        {
            count += n;             // inc byte count
            bytes += n;             // inc ptr to buffer
        }
        else
        if (n < 0)                  // error
            return(-1);
    }

    return(count);
}
```

There are several Socket functions that we can use to send data, but the one we've chosen to use here is `send()`. There is also a `write()` function that does the exact same thing as `send()` except that it does not have the flags parameter. Use whichever function you prefer, but be aware that calling `send()` or `write()` does not guarantee that all of the bytes you requested got sent. These functions return a byte count indicating how many bytes were actually sent, so we sit in a `while` loop to make sure that everything goes out.

Remember how we needed a function to determine if a Socket had data waiting to be read on it before we called `accept()`? Well, the same thing goes for writing data. We need a function that checks the Socket to let us know if it's safe to write data to it. This function is nearly identical to `IsDataWaitingToBeRead()` since it works the same way:

Listing 15-12: Determining if a Socket is to Send Data

```
static Boolean  IsSocketReadyForWrite(int socket)
{
    struct fd_set       connectionSet;
    struct timeval      timeout;
    int                 result;
```

```
        /* SET TIMEOUT & SET VALUES */

    timeout.tv_sec          = 0;
    timeout.tv_usec         = 0;            // don't block at all

    FD_ZERO(&connectionSet);                // use macro to clear the set
    FD_SET(socket, &connectionSet);         // use macro to set socket


      /* CALL SELECT TO SEE IF THERE'S DATA WAITING ON THE SOCKET */


    result = select(socket+1,
                    nil,
                    &connectionSet,
                    nil,
                    &timeout );


        /* CHECK RESULTS */

    if (result < 0)
        return(false);

    if (result > 0)                     // was there an event on the socket?
    {
        if (FD_ISSET(socket, &connectionSet))   // is socket ready?
            return(true);
        else
            return(false);
    }

    return(false);
}
```

The only difference between this function and `IsDataWaitingToBeRead()` is that the `connectionSet` is passed in as the 3rd parameter, not the 2nd parameter. This configures the call so that we're checking the Socket's output queue instead of its input queue. If the output queue is clear then the result will be 1, so we're safe to send data over the Socket.

It is not absolutely necessary that you call `IsSocketReadyForWrite()` before sending any data. If you do call `send()` and the Socket wasn't ready for it, then `send()` will just block until the Socket is ready. So, if we want to prevent our application from stalling it's a good idea to check the write status of the Socket before trying to `send()` data through it. As you'll see in the sample project, I only call `IsSocketReadyForWrite()` when passing game data during gameplay. I don't bother with it when I'm sending setup information before the game starts.

Unfortunately, when sending data over a TCP/IP connection there is one big problem that pops up all the time. The data we're sending between players is generally pretty small, and the OS doesn't immediately send network data unless there's a lot of it. If we try sending say 100 bytes, that 100 bytes goes into an output queue where it sits until one of two things happen: either more data comes in forcing the queue to flush, or some time has passed (around 1/5th of a second) in which case the OS goes ahead and sends the data.

One way to get around this problem is to use UDP connections instead of TCP/IP connections. UDP data is always sent immediately, however, it is not guaranteed to ever get to its destination. TCP/IP data is guaranteed to get where it's going. You can fiddle with whatever connection types you want to try for your game, but if you want to make sure that TCP/IP is flushing its queue every time you send data then there's a little trick that you can do: send a lot of data! In the sample project, you'll see this call in a few places:

```
    WriteNetData(gConnectionToHost, &gPadding, PADDING_SIZE);
```

I call this after I've sent the important information that I want the other players to get. What this does is send a block of 3500 bytes to the Socket, thereby causing the output queue to flush and send everything immediately. This is a total hack and will cause problems if you're on a slow network, but for the purposes of our sample project it actually works quite well. You just need to remember to read all that garbage padding on the receiving end.

Receiving Data

To receive data from a remote Socket we have a function that should look rather familiar:

Listing 15-13: Reading Data

```
int ReadNetData(int socket, void *buffer, int numBytes)
{
    Ptr     bytes = buffer;         // Ptr to buffer
    int     count, n;

    n = count = 0;

    while (count < numBytes)        // loop until we've got all the bytes
    {
        n = recv(socket, bytes, numBytes - count, 0);   // read data

        if (n > 0)                  // we got some bytes
        {
            count += n;             // inc byte count
            bytes += n;             // inc ptr to buffer
        }
        else
        if (n < 0)                      // error
            return(-1);
    }

    return(count);
}
```

The `ReadNetData()` function looks virtually identical to the `WriteNetData()` function. The only difference is that we call `recv()` instead of `send()`. We could have also used the function `read()` instead of `recv()` since it's completely identical minus the flags parameter. Just like when writing data, reading data is not guaranteed to get all the bytes you requested, so `ReadNetData()` sits in a loop until all of the needed data has been received.

Now you know the basics of networking on OS X, but believe me, there is a lot more to networking if you want there to be. I recommend reading up on BSD Sockets as there are lots of resources on the Internet dealing with this technology.

Chapter 16: Copy Protection

There is one basic rule in sales that says “Thou shall not disgruntle thy customer,” but many game publishers seem to have forgotten this rule, and they’ve implemented copy protection schemes that are a nuisance to their paying customers. I’m primarily talking about games that require you to keep the CD in the drive when playing the game. This form of copy protection is completely ineffective and totally unacceptable since it’s a major hassle for legitimate paying customers. So, don’t even think about doing something like that! There are much more effective ways to reduce piracy that won’t irritate your customers, thus, leaving them happy and loyal.

Serial Numbers

Using unique serial numbers is the first thing that all of your games should do whether they’re downloadable or come on a CD. If your games are downloadable and the user buys the serial numbers on your web site, then you can write an algorithm that embeds their credit card number into the serial numbers that they purchase. This is highly recommended, and if you do it you should publicize it. If a person knows that their credit card number is encoded in their serial number, then they’ll be much less likely to share that serial number with anyone else.

You’ll need to spend a little bit of time each week on the internet looking for serial numbers that people have posted to pirate boards and other places. When you find a serial number, you can add it to your list of known voided serial numbers so that they won’t work anymore. This technique is becoming very popular in all sorts of applications, not just games. The trick is in figuring out a good way to void those serial numbers on user’s machines after they’ve already started using them.

Your game should have an internal copy of this pirate serial number list, and when a user enters their serial number it should get verified against it. Every time you post an update to your game it should include an updated list. One trick that I do is to update this list constantly, and re-upload the files, but I keep the game’s version number the same. This totally confuses pirates. Someone might post a serial number to a pirate board and say that it works with version 1.0.5, but once I see that and update the list, any more people downloading 1.0.5 will be totally confused why that serial doesn’t seem to work with their copy.

Phone Home

Not everyone is going to download updates, however, so we need to find another scheme for dynamically updating the pirate serial numbers list on a user's computer. This is done by "phoning home" every few days. In other words, every so often you can have your game read the current pirate serial number list off of your web site, and then compare the user's existing serial number to those in the pirate list. If there is a match, then the user is using a pirated serial number. This is actually quite easy to implement, and all of the games that Pangea Software has released since around 1999 do this.

You should never dial out a modem since this would violate the "customer aggravation" credo, so before trying to read a file off of a web site see if there is currently a live connection to the Internet.

Listing 16-1:Determining if there is an Internet Connection

```
Boolean IsInternetAvailable(void)
{
    OSErr               iErr;
    InetInterfaceInfo   info;

    iErr = OTInetGetInterfaceInfo(&info, kDefaultInetInterface);

    if (iErr)
        return(false);

    return(true);
}
```

All this function does is make a single call to `OTInetGetInterfaceInfo()` which returns `true` if the Open Transport IP protocol stack is loaded. If there's no IP stack loaded then there's no Internet connection. Downloading a data file is equally as simple:

Listing 16-2:Downloading a File from a URL

```
OSStatus DownloadURL(const char *urlString, Handle buffer)
{
    OSStatus    err;

        /* MAKE SURE THERE'S A BUFFER TO LOAD INTO */

    if (buffer == nil)
        DoFatalAlert("\pDownloadURL: buffer == nil");

            /* DOWNLOAD DATA */
    err = URLSimpleDownload(
                        urlString,      // pass the URL string
                        nil,            // download to buffer, not FSSpec
                        buffer,         // handle to buffer
                        0,              // no flags
                        nil,            // no callback eventProc
                        nil);           // no userContex

    return(err);
}
```

To call `DownloadURL()` we would first need to allocate a handle to hold the data we're reading, and then pass a string containing the full URL of the file we're looking for. You can read any data from a web page with this function. For example, here's how to read the Pangea Software homepage:

```
    buffer = AllocHandle(100000);
    DownloadURL("http://www.pangeasoft.net/index.html", buffer);
```

As long as we're going through all of this trouble to download a data file, why not make it do more than just obtain a list of known pirated serial numbers? I like to embed all sorts of useful information in these data files including version information and messages. The version information can let the user know if there is an update of your game available, and putting messages in there is a great way to let customers know about new products and things of that nature. The data files that my games download look something like this:

```
#$#$
#BVER
2.0.0
#BSER
DJCDIADPOOOT
JDCQWWOBVLKA
ARBIICOSLNCP
*
#NOTE0003
Our new game, Nanosaur 2: Hatchling is now available!  Get the free demo at
www.pangeasoft.net/nano2
#TEND
```

This is a metafile that contains tags identifying different pieces of data:

#$#$
This tag indicates the start of the file. It's important to look for this because your buffer might have been filled with a "404 File Not Found" message from the server if the data file

wasn't found. `DownloadURL()` won't return an error if the server kicks out a message like that. As far as the Mac is concerned it read valid data, so we've got to check that the data is actually our file by looking for this tag.

`#BVER`
This is short for "Best Version", and the data to follow is the version number of the most recent update to the game. The game compares this version number with its own version number, and if it's newer then it will display a message to the user informing them that an update is available.

`#BSER`
This is short for "Bad Serials". The data following this tag is a list of known pirate serial numbers. The end of the list is indicated by an asterisk.

`#NOTE`
This tag also has a number after it indicating the Note ID #. We track these ID numbers so that we know what messages have already been displayed to the user. Obviously, we don't want to display the same message every time the user runs the game, so we track which messages have already been viewed. In the example above, the message is #3 so the following text will only be displayed once, but if we later change the message to #4 then whatever text follows it will be displayed once.

`#TEND`
This marks "The End" of the data file.

In the sample project "Copy Protection.xcode" you'll see a new source file called "Internet.c". This file contains all of the code needed to load this type of metafile from a web site and then parse the tags. This sample code is fairly stripped down since it doesn't really do anything with the serial number data that it extracts out of the metafile. It does notify you about newer versions, and it will display the embedded message, but all it does with the pirate serial number list is print it to the Console. Additionally, the embedded message will be displayed every time you run the sample application because we're not tracking the Note ID numbers here.

Hackerproofing

Hackers will do everything in their power to figure out your copy protection schemes and find ways around them. It is possible, however, to make your game such a pain to hack that most people won't bother. That's what I've done with most of my games, and as a result they are very difficult to pirate on a mass scale.

One of the first things that hackers do is try to work around your phone-home code. There is a free utility that they use called "Little Snitch". What Little Snitch does is notify the user whenever an application attempts to access the internet, and then it lets the user choose to let it happen or to deny it. If the user denies the access, then `URLSimpleDownload()` will return an error as though the web site was not found. Unless you take action on this problem, hackers can quite easily get around your phone-home calls, so what I've chosen to do in my games is to simply not let the game run if Little Snitch is found. In other words, I've intentionally made my games incompatible with that utility.

To detect if Little Snitch (or any other process of interest is running), we do this:

Listing 16-3:Check to see if Little Snitch is running

```
Boolean CheckForLittleSnitch(void)
{
    ProcessSerialNumber psn = {kNoProcess, kNoProcess};
    ProcessInfoRec  info;
    short           i;
    OSErr           iErr;
    Str255          s;
    const char      snitch[] = "\pLittleSnitchDaemon";

            /* SET A PROCINFO REC */

    info.processName        = s;            // process name will go here
    info.processInfoLength  = sizeof(ProcessInfoRec);
    info.processAppSpec     = nil;


    /* SCAN ALL OF THE PROCESSES RUNNING & LOOK FOR LITTLE SNITCH */

    while(GetNextProcess(&psn) == noErr)
    {
        iErr = GetProcessInformation(&psn, &info);  // get process's info
        if (iErr)                                   // if err then no more processes
            break;

        if (s[0] == snitch[0])                      // see if name matches
        {
            for (i = 1; i <= s[0]; i++)
            {
                if (s[i] != snitch[i])
                    goto next_process;
            }
        }
        else
            goto next_process;
            /* IF GETS HERE THEN LITTLE SNITCH IS RUNNING */

        return(true);
next_process:;
    }

    return(false);
}
```

It's up to you to decide how to handle things if Little Snitch is found, but my advice is to not bring up a dialog saying "Warning, this game is not compatible with Little Snitch". Anything that obvious is only going to tell hackers what to look for when hacking your game. In my games, I just quietly set a flag, and then I check that flag later when I load a level. If the flag is set then I just quit the game – no explanation given.

In your games you'll need to try hard to find ways to make your code hackerproof. Another tactic that hackers always use is to scan through your application's binary looking for the hard-coded list of pirate serial numbers. If they can find the list, then they'll usually clear it out to a bunch of 0's. What I've done in my games to avoid this problem is to checksum the list of serial numbers. Every time the game launches it'll re-calculate the checksum, and if it doesn't match my pre-calculated checksum then I bring up a message saying "This application appears to be corrupt", and then bail. Additionally, never store the raw serial number data in a list. You should always encode it somehow so that it's more difficult for hackers to see. Even just XOR'ing each character with some value like 0xC5 will make it much harder to find.

Another thing that I've learned to do is to rename my anti-piracy functions. That will make it harder for hackers to locate the functions in question. For example, it would be unwise to actually leave the `CheckForLittleSnitch()` function named like that because hackers will see that, and immediately go to work on it. Instead, rename that function to something totally misleading like "`AudioInitChannels()`".

Even if a hacker does locate some of your critical copy protection functions, it's still easy to make their lives difficult. Sometimes hackers will just put a "return" opcode at the front of your critical functions which will cause them to be skipped. You can get around this by making sure that flags are set inside each critical function, and then later in the game you randomly check these flags to see if they were set. If not, then your copy protection code was never called.

Never just set simple True/False flags either because a good hacker might figure this out. Always set the flags to some sort of cookie value that will be much more difficult for a hacker to understand. For example, in your serial number verification function you might have a flag called `gSerialWasVerified` that gets set to indicate that verification did occur. Set that flag to some crazy value like `0x1FA394` to indicate a False response, and maybe `0x42EA901` to indicate a True response. Things like that will drive hackers crazy!

One other important thing that I do with my serial number code is to put the functions in a header file, and make them inline. Then I call the verification functions from several different places in my game. This will cause your verification code to be duplicated in many places, so even if a hacker thinks they've worked around your functions they might not realize that there are actually multiple copies of each function scattered all over the place.

No matter what you do, you'll never be able to stop 100% of the piracy of your game. That's ok because all you need to do is stop the "mass" piracy, and that's pretty easy to do if you take the precautions described above. Don't worry about little Jimmy giving his friend Stephen a copy of your game – that's an issue, but at least one of those two people actually did buy the game. When your game is on a pirate board, however, the gloves come off because nobody there is paying anything for anything.

Chapter 17: Miscellaneous Mac Tidbits

There are a whole bunch of other Mac OS X specific things that every game programmer should know about, but they don't really fall under any specific category, so I'm going to discuss them all here.

Setting the Default Directory

On Mac OS 9 and earlier, the "Default Directory" was always set to the directory of the application that was running. This made it easy to load data files that resided in the application's folder. For example, in Bugdom, if I wanted to load the file Forest.aiff as shown here…

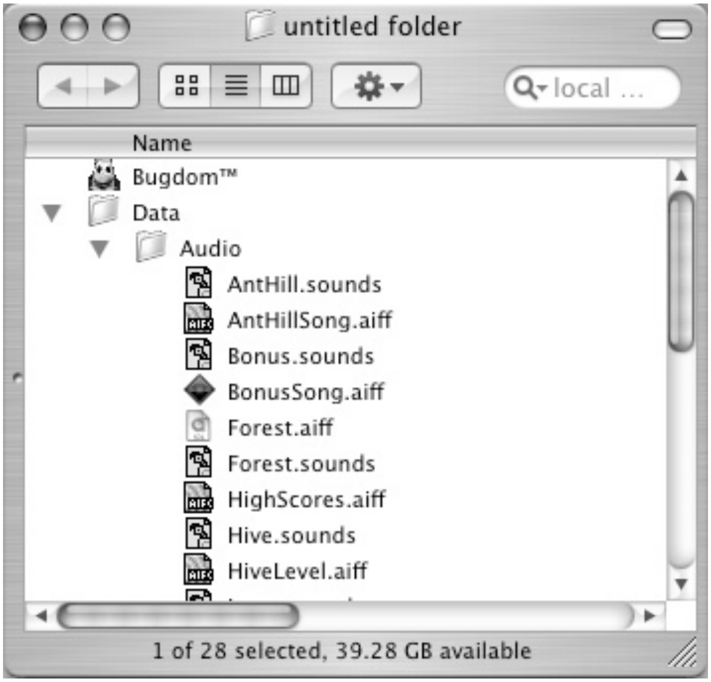


Figure 17-1: The directory listing of Bugdom

… I would make a simple call like this:

```
    FSMakeFSSpec(0, 0, "\p:Data:Audio:Forest.aiff", &spec);
```

The volume and directory ID's are set to 0 to indicate that we're giving a path from the default directory, and the default directory back on OS 9 was always the running application's folder. On OS X, however, the default directory is set to garbage when your application launches. You need to manually set it with this code:

Listing 17-1:Setting the Default Directory

```
void SetDefaultDirectory(void)
{
    ProcessSerialNumber serial;
    ProcessInfoRec      info;
    FSSpec              app_spec;
    WDPBRec             wpb;

        /* GET OUR PROCESS'S INFO */

    serial.highLongOfPSN    = 0;
    serial.lowLongOfPSN     = kCurrentProcess;

    info.processInfoLength = sizeof(ProcessInfoRec);
    info.processName = NULL;
    info.processAppSpec = &app_spec;

    GetProcessInformation(&serial, & info);


            /* EXTRACT VOL/DIR INFO */

    wpb.ioVRefNum = app_spec.vRefNum;
    wpb.ioWDDirID = app_spec.parID;
    wpb.ioNamePtr = NULL;


            /* SET DEFAULT DIRECTORY */

    PBHSetVolSync(&wpb);
}
```

Finding the Preferences Folder

When you save a user's settings you'll want to write a file into their Preferences folder. To do that you've got to know the Volume and Directory ID's of that folder.

```
    iErr = FindFolder(kOnSystemDisk,
                    kPreferencesFolderType,
                    kDontCreateFolder,          // only locate the folder
                    &gPrefsFolderVRefNum,
                    &gPrefsFolderDirID);
```

The constant `kPreferencesFolderType` indicates that we're looking for the Preferences folder, but you can use this same call with other constants to locate all sorts of other system folders. A complete list is found in the CarbonCore framework file named Folders.r, but some of the more commonly used ones are:

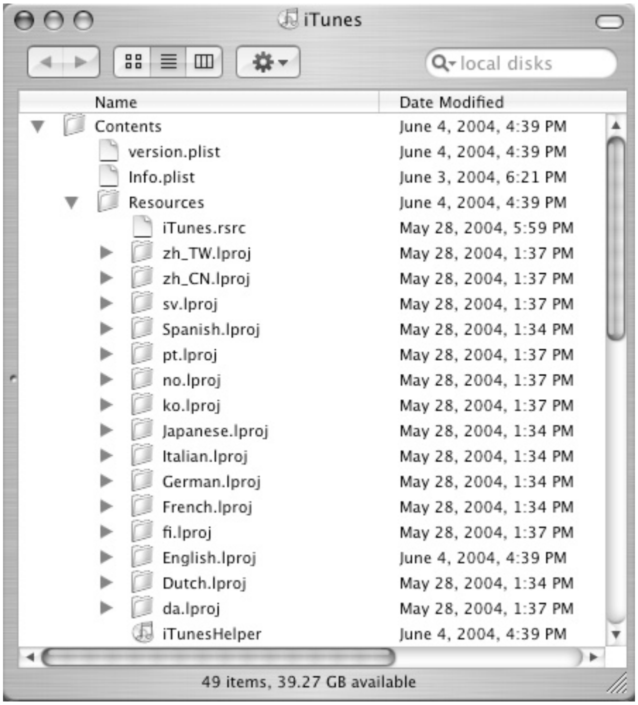
```
#define kSystemFolderType               'macs'
#define kDesktopFolderType              'desk'
#define kApplicationsFolderType         'apps'
#define kDocumentsFolderType            'docs'
#define kCurrentUserFolderType          'cusr'
```

Once you've located the Preferences folder you'll usually want to create a folder for all of your game's preference files:

```
    DirCreate(gPrefsFolderVRefNum,gPrefsFolderDirID,
                "\pMyGame",
                &createdDirID);
```

Language Determination

The "correct" way to handle localization on OS X is to have localized (translated) resources for each language that you support, and put them into the appropriate folders in the application bundle. For example, here's a directory listing for iTunes showing all of the languages that it supports:



Name	Date Modified
Contents	June 4, 2004, 4:39 PM
version.plist	June 4, 2004, 4:39 PM
Info.plist	June 3, 2004, 6:21 PM
Resources	June 4, 2004, 4:39 PM
iTunes.rsrc	May 28, 2004, 5:59 PM
zh_TW.lproj	May 28, 2004, 1:37 PM
zh_CN.lproj	May 28, 2004, 1:37 PM
sv.lproj	May 28, 2004, 1:37 PM
Spanish.lproj	May 28, 2004, 1:34 PM
pt.lproj	May 28, 2004, 1:37 PM
no.lproj	May 28, 2004, 1:37 PM
ko.lproj	May 28, 2004, 1:37 PM
Japanese.lproj	May 28, 2004, 1:34 PM
Italian.lproj	May 28, 2004, 1:34 PM
German.lproj	May 28, 2004, 1:34 PM
French.lproj	May 28, 2004, 1:34 PM
fi.lproj	May 28, 2004, 1:37 PM
English.lproj	June 4, 2004, 4:39 PM
Dutch.lproj	May 28, 2004, 1:34 PM
da.lproj	May 28, 2004, 1:37 PM
iTunesHelper	June 4, 2004, 4:39 PM

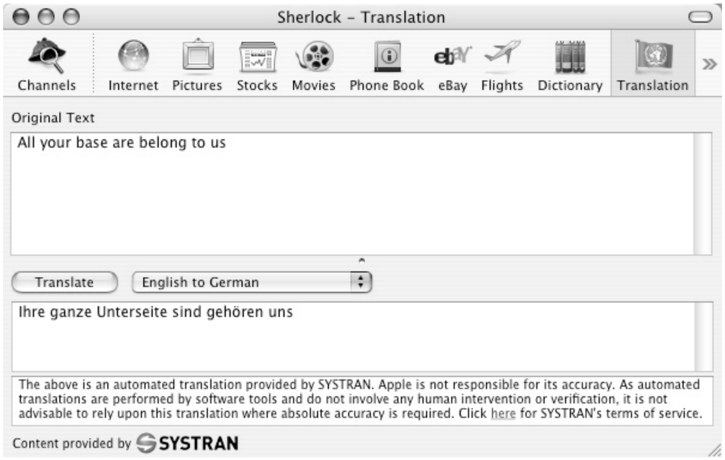
Figure 17-2: Resource folders for iTunes

The OS will automatically load the correct resources based on the current language settings for the computer. This is all fine and dandy except that games often don't work quite that way. Sometimes we really need to know what language is set so that we can manually handle it in our game. To find this out we do this:

Listing 17-2:Determining the Language Settings

```
long DetermineLanguage(void)
{
    long        keyboardScript, languageCode;

        /* FIND OUT WHAT LANGUAGE COMPUTER IS SET TO */

    keyboardScript = GetScriptManagerVariable(smKeyScript);
    languageCode = GetScriptVariable(keyboardScript, smScriptLang);

            /* DO SOMETHING  WITH RESULTS */

    switch(languageCode)
    {
        case    langFrench:
                printf(“We support French!”);
                break;

        case    langGerman:
                printf(“We support German!”);
                break;

        case    langSpanish:
                printf(“We support Spanish!”);
                break;

        case    langItalian:
                printf(“We support Italian!”);
                break;

        case    langSwedish:
                printf(“We support Swedish!”);
                break;

        case    langDutch:
                printf(“We support Dutch!”);
                break;

        default:
                printf(“Unsupported language, so using English by default”);
    }

    return(languageCode);
}
```

Determining what language the computer is using is the easy part. Doing the actual localization of your game is not. If you’ve got no budget to pay a professional translator to localize the text in your game then you can always revert to using Sherlock. Sherlock translations tend to be quite horrible, but I’ve used it on many occasions.



Original Text

All your base are belong to us

English to German

Ihre ganze Unterseite sind gehören uns

The above is an automated translation provided by SYSTRAN. Apple is not responsible for its accuracy. As automated translations are performed by software tools and do not involve any human intervention or verification, it is not advisable to rely upon this translation where absolute accuracy is required. Click here for SYSTRAN's terms of service.

Figure 17-3: Using Sherlock to do localization

Even though Sherlock translations are bad, you'll end up getting emails from users who offer better translations that you can put into the next update. Even when I hire professional translators to localize my games, there are always some errors, and my customers always catch them. So, when version 1.0 of a game comes out, the localization is awful, but within a week or so you'll have enough free feedback from your international customers that you can update all the text and release version 1.0.1.

Should you choose to make a concerted effort to do good localization, then you can do a search on the web for "software localization service" and you'll have many companies to choose from. Send your English resources and instruction manual to several companies for a bid, and then go with the one you like the most. You'd be surprised how different the bids for this will be. For example, when we bid out Nanosaur 2 we got bids ranging from $1700 to $5000. Obviously, we didn't go with the $5000 option.

Filenames

The standard file system on OS X is HFS, and HFS is not a case sensitive system. As far as the OS is concerned, the names "myfile.c" and "MyFile.c" are exactly the same. Because of this, many game programmers have been lazy about capitalization in their filenames and the matching strings in the code. In Figure 17-1 you'll notice that all of the data files for Bugdom were capitalized. However, in the code I wasn't very careful, and would often just use all lower-case in the filename strings. I've probably done this in every game I've ever written, but I've really got to stop myself from doing this because it causes problems on OS X.

Some people have formatted their drives with UFS (Unix File System) which actually *is* case-sensitive. When they install any of my games on their UFS drives and then try to run them, they eventually get a File Not Found error. There are other issues with running a true Mac application on a UFS volume, so it's not recommended either way, but to avoid problems with future versions of HFS you should really try to be consistent with your filenames and name strings in the code.

Loading Images with Quicktime

In Chapter 10 we talked about how Quicktime can be used to play pretty much any kind of Audio file around. Well, the same goes for image formats. Quicktime can be used to load anything from JPEG's, TIFF's, PICT's, and even Photoshop files. The following code will load the image file indicated in the `FSSpec` into a `GWorld`:

Listing 17-3:Loading an Image with Quicktime

```
void DrawPictureIntoGWorld(FSSpec *myFSSpec, GWorldPtr *theGWorld)
{
    OSErr                       iErr;
    GraphicsImportComponent     gi;
    Rect                        r;
    ComponentResult             result;
    PixMapHandle                hPixMap;


            /* PREP IMPORTER COMPONENT */

                                        // load importer for this image file
    result = GetGraphicsImporterForFile(myFSSpec, &gi);
    if (result != noErr)
    {
        DoFatalAlert("\pGetGraphicsImporterForFile failed!  One of
                    Quicktime's importer components is missing.  You should
                    reinstall OS X to fix this.");
    }

            /* GET DIMENSIONS OF IMAGE */

    if (GraphicsImportGetBoundsRect(gi, &r) != noErr)
        DoFatalAlert("\pGraphicsImportGetBoundsRect failed!");


            /* MAKE GWORLD */

    iErr = NewGWorld(theGWorld, 32, &r, nil, nil, 0);       // try app mem
    if (iErr)
```

```
        DoFatalAlert("\pDrawPictureIntoGWorld: NewGWorld failed");

    hPixMap = GetGWorldPixMap(*theGWorld);  // get gworld's pixmap
    (**hPixMap).cmpCount = 4;               // we want full 4-component argb
                                            // (defaults to only rgb)

            /* DRAW INTO THE GWORLD */

    DoLockPixels(*theGWorld);

                                        // set the gworld to draw image into
    GraphicsImportSetGWorld(gi, *theGWorld, nil);

                                        // set import quality
    GraphicsImportSetQuality(gi,codecLosslessQuality);

    result = GraphicsImportDraw(gi);    // draw into gworld
    CloseComponent(gi);                 // cleanup
    if (result != noErr)
        DoFatalAlert("\pGraphicsImportDraw failed!");
}
```

The call to `GetGraphicsImporterForFile()` tells Quicktime to figure out what kind of image file this is and to load the appropriate importer for it. If it is a file type that Quicktime does not understand you'll get an error.

Next, we determine the size of the image by calling `GraphicsImportGetBoundsRect()`, and then we use the returned `Rect` to allocate a new `GWorld`. It's into this `GWorld` that we'll draw the image, and there is a very important hack that we do here:

```
    (**hPixMap).cmpCount = 4;
```

When you allocate a 32-bit `GWorld` the component count (`cmpCount`) is actually set to 3. This would tell Quicktime to only recognize the image's RGB channels, but not the alpha channel. We want to preserve any alpha channel information in the image, so we've got to manually set the `cmpCount` to 4.

Before drawing the image into the `GWorld`, it is a good idea to call `GraphicsImportSetQuality()` to make sure we get the best image we can. For most standard image formats there is no difference in import quality, but to be safe we'll always set this to `codecLosslessQuality`.

To draw the image, we simply call `GraphicsImportDraw()`, and Quicktime takes care of everything.

Chapter 18: Marketing & Selling

Once you've made a game for the Mac, it sure would be nice to make some money off of it, wouldn't it? My motto is "You've got to spend money to make money", so if you want your game to be a commercial success you've got to spend the time and money on marketing and sales. You could take the easy way out and try to get one of the existing Mac game publishers to publish your game for you, but I recommend against that in most cases. You should consider finding a publisher if you just don't have the resources to publish it yourself, but if you think you've got a hit game then you really don't want to dilute its value by putting a publisher in the mix. Besides, you should always try to develop your company's brand name so that you'll get more and more successful in the future.

When I tell people that I make games for the Mac, I usually get a funny look, but the fact is that the Mac is an excellent platform for developing games. While we may only have 5% of the worldwide computer market, it's better to be a big fish in a small pond than a small fish in a big pond, and it doesn't take much effort to become a big fish in Apple's tiny pond. The PC game market is so overcrowded that you'd have a better chance of making ice cream in Hell than starting a successful game company on that platform. The Mac, however, is an easy market to break into and get attention. As long as you've got a good game, you can make good money.

Marketing Your Game

The great thing about developing for the Mac is that marketing to Mac users is much easier than it is for any other gaming platform. Advertisements in Mac magazines and on Mac web sites cost a fraction of what it would be for the same thing in other magazines, say for PC, Playstation, etc. Unlike the PC market which is totally flooded with games, the Mac market has plenty of room in it; so, smaller games can still get a lot of good publicity.

Here is a list of the best Mac Marketing Resources you should pursue:

Inside Mac Games

www.insidemacgames.com

This is the premier web site for gaming on the Mac, and it has been around since 1993. You should always send any press releases to IMG since they are always happy to announce new game information. If they like you enough they'll usually do previews and reviews of your games.

You can also take out banner ads for a very reasonable price, and they'll be seen by your core gaming audience, so it's well worth it. To get information about current advertising rates contact tuncer@insidemacgames.com

Mac Game Files

www.macgamefiles.com

This site is actually run by Inside Mac Games, so there is some cross-linking between the two sites. This is the main repository for Mac game downloads. Just about every demo, update, and downloadable game can be found here along with user reviews and comments. To submit your game to MGF click on the "submit a file" link at the top of the homepage.

While MGF is a good place to host your files, it should not be your only host. The file transfer speeds on MGF are generally quite slow. I usually get between 2k/sec and 12k/sec. Not exactly high-speed bandwidth, but it's free so don't complain.

Mac Gamer

www.macgamer.com

This is another one of the big Mac gaming web sites. Always send your Press Releases to these guys as well.

MacCentral

www.maccentral.com

This is not a game-specific web site, but it is the premier Mac news site and one of their primary writers, Peter Cohen, is a big fan of games, so be sure to keep these guys in the loop as your game nears release. MacCentral is actually operated by Macworld Magazine, so you can get cross-promotional advertising rates. Ads here are a bit pricey for a small game publisher, but they will reach a wide audience so if you can work out a good deal then go for it.

VersionTracker

www.versiontracker.com

This is a hugely popular site that lists software updates and new releases every day. It's a good marketing resource because just by having your game listed in their daily updates you'll get a ton of traffic to your site. Be sure to always let them know when you've released an update to your game since that's just more free publicity.

Macworld Magazine

www.macworld.com

Macworld is the oldest major Mac magazine still around, and they have a very large readership base. Advertising in this magazine is very expensive, and you usually have to commit to a series of ads, but I've always found this to be an effective way to get the word out.

MacAddict Magazine

www.macaddict.com

This magazine has really increased in popularity over the last few years, and it probably caters more to the gamer crowd than Macworld magazine does. The other nice thing about MacAddict is that they ship a CD with every issue, and if you're lucky, they'll ask you if they can put your game demo on there. This is a very huge deal! Whenever MacAddict has put one of our games on their CD, our sales of that game will typically triple during that month! You can also pay to have your game put on their CD.

Apple's Game Site

www.apple.com/games

Apple claims to get a massive number of hits on their games page every month, but honestly, I've never found any marketing here to make much difference in sales. They'll often do articles about new games, and they'll host movie trailers, but advertising here doesn't seem to pay off – at least it didn't for us when we tried it. You might have better luck.

Game Demos

The single most important thing you can do to market your game is to release a demo. People don't like to shell out hard earned money for something they've never seen, so a free demo is the best selling gimmick you can do, and they're easy to make. It usually only takes me about 2 hours to build demo versions of any of my games. If you're just porting a PC game that's already well known, then a demo probably isn't too important, but if you're developing an original title then it is critical.

In the old days of video games (and by old I mean like 1986) most game demos would expire after a certain amount of use. For some reason, people stopped doing this, but I highly recommend that all game demos have an expiration time bomb in them. While it's true that an expiring demo will annoy some people, the fact is that for every one person that gets angry with you for doing that, there will be ten other people buying the game who might not have done so otherwise. Besides, if someone doesn't know if they want to buy your game after playing it for an hour then it's unlikely they will ever buy it.

You would not believe how many times I've picked up the phone and heard a woman telling me that she needed to order a game right away, and needed it shipped Next Day because her son was having a tantrum since he couldn't play the demo anymore. I could always hear a crying kid in the background. I'm not making this up! This happens all the time, so trust me, expiring demos are a great motivation to get people to buy your game. Small children, especially, will just play a demo forever since they don't know better, but if that demo expires you bet you'll be hearing from the parents with a credit card number in hand.

The other important thing about game demos is that you should never give too much away. You'll always be tempted to show the player all the cool stuff in your game, but resist! You need to keep the demo downloads small, and you should always leave the customer wanting more. Smaller demos are faster to download, thus, more people will try it out. On a similar note, *never ever* release a game demo before a game has shipped! You need to get every impulse buy that you can, and if the game isn't available when the user tries out your demo, they're going to blow it off, and you've just lost a sale.

My favorite example of a demo gone wrong is the one for the old EA game called "Future-Cop." This was a really cool game, but they released the demo about 2 months before the game actually shipped, and the demo had way too much stuff in it. When I played the demo I really wanted to go buy the game, but by the time that FutureCop finally shipped several months later, I was tired of it. I had played the demo to death, and had no desire for more.

One of the troublesome issues with game demos is getting them hosted for people to download them. Our game demos are usually 20-30 Megs in size, and if you've got thousands of people downloading it every month, the bandwidth gets pretty heavy. As I mentioned above, you can submit your game files to MacGameFiles.com which is free, but their bandwidth is very poor, so you need to provide a faster way for people to get the files. You can occasionally find sites that offer unlimited bandwidth for $29.95 month, but those are always garbage. Your downloads will be even slower than MacGameFiles, and there's usually fine print that says that “unlimited bandwidth” really means 30gigs per month. Anything over that costs you $5 per gig or something ridiculous like that.

There is no magic bullet solution to the bandwidth problem, however, there is one service that I can recommend which has been amazingly reliable:

www.fileburst.com

FileBurst charges about $1.00 per gigabyte of bandwidth, with discounts over 100 gigs. The download speeds are wicked fast too. I’ve never gotten less than 250k/sec off of their site even during high traffic time periods. We use FileBurst to host all of our files including game demos, updates, and full versions.

Statistically speaking, I’ve found that about 1 in 20 people who download a game demo will end up buying it (5% sell-through rate). If the demo is around 100 Megs (like Billy Frontier was), then that means it costs about 10 cents in bandwidth per download. If only 1 in 20 people end up buying it then that means you’ve paid about $2.00 in bandwidth fees to make the sale. You always need to work this into your business plan. I always assume $1.00 to $2.00 in costs for bandwidth on each sale.

Selling Retail

Many people are going to think I’m crazy when I say this, but here it goes… There’s no money to be made in retail on the Mac. There I said it. To start with, there are only a few stores selling Mac software, and only two really big ones with significant sales volume: CompUSA and the Apple Store. The only way to get into either of these stores is to use one of the big distributors like Ingram Micro or Navarre. Without going into a long tirade, the bottom line is that the middle men take so much money from you that there’s no way you can actually make a decent profit. Getting paid by a distributor is worse than pulling teeth, and you should consider yourself lucky if you get paid at all. If I were to add up all of the money owed to us from various distributors who simply stiffed us over the years, it would come close to $20,000!

You don’t want to mess with the mail order catalogs either. In order to get picked up by a mail order catalog you have to agree to buy a huge number of expensive ads in their mailers, and the sales are always very low. For example, we spent about $12,000 on an advertising campaign for Cro-Mag Rally in one of the big Mac mail order catalogs. Guess how many copies they sold… just guess… It was around 30. Yes, we spent $12,000 of our marketing budget to sell 30 copies of a game that had a profit margin of around $10. That was a fair trade don’t you think?

Now I should point out that it's not all bad. As long as you deal with the small guys, your life can be much better. We still do some retail distribution, but now we use a nice little distributor called Visco Entertainment. These guys are honorable and always pay their bills on time, so I recommend inquiring with them about distribution:

www.viscoent.com

Selling direct to the smaller retail stores is usually a pleasant experience as well. The small companies are run by caring human beings who aren't out to screw you, so you can feel safe doing business with them.

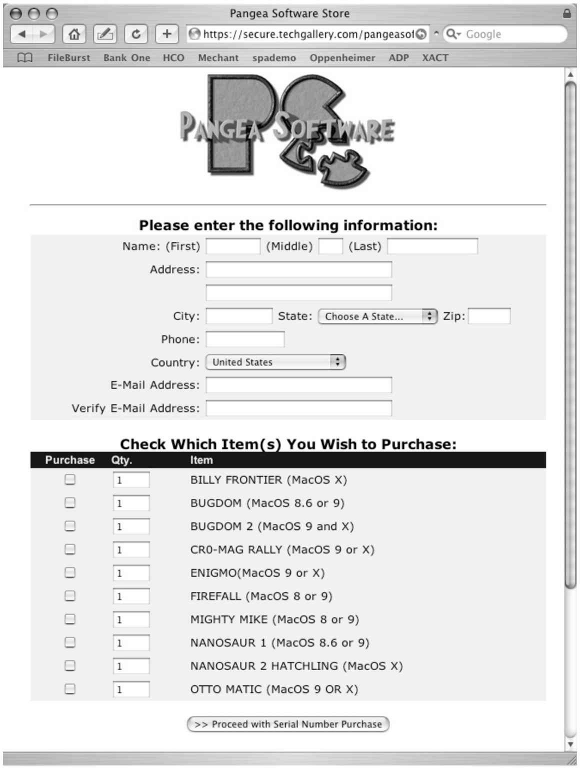
Selling Online

Online sales are the future of software distribution, and the future is now. About two years ago Pangea Software switched its business model from retail sales to online sales, and it was the best business decision I've ever made. Our profits skyrocketed since we'd cut out all of the middlemen. Plus, we got immediate payment from the credit card companies since all of these sales were direct-to-end-user sales. On a $40 game we would be lucky to have a $10 profit from that when we went through the big distribution channels, because by the time you deduct the manufacturing and shipping costs, discounts, rebates, returns, and lack of payments, that was about all that remained. But with direct online sales a $40 game sells for $40 and our only costs are the $1 to $2 in bandwidth fees plus the 3% that the credit card company takes. The rest is all profit.

While it is true that you'll sell more units in retail than you can online, the profit margin is so much higher with online sales that it more than makes up the difference. We make over 3x as much profit on an online sale as we did with retail sales. So, even if we only sell half as many copies with online distribution, we're still making a whole lot more profit. Profit aside, think about all the stress that you don't have to deal with when doing online distribution. You don't have to worry about when or *if* you'll ever get paid, and you don't have to deal with buyers or any of those people who are out to squeeze you for every penny.

The Online Store

Around 75% of all of our online sales are for the downloadable versions of our games, with the other 25% buying the boxed CD versions. During the Christmas buying season it's an especially good idea to offer a boxed version of your games since downloads don't make the best gifts. To handle all of this we've had a custom order page built that handles orders for both the CD versions of the games and the downloadable versions:



Please enter the following information:

Name: (First) (Middle) (Last)
Address:
City: State: Choose A State... Zip:
Phone:
Country: United States
E-Mail Address:
Verify E-Mail Address:

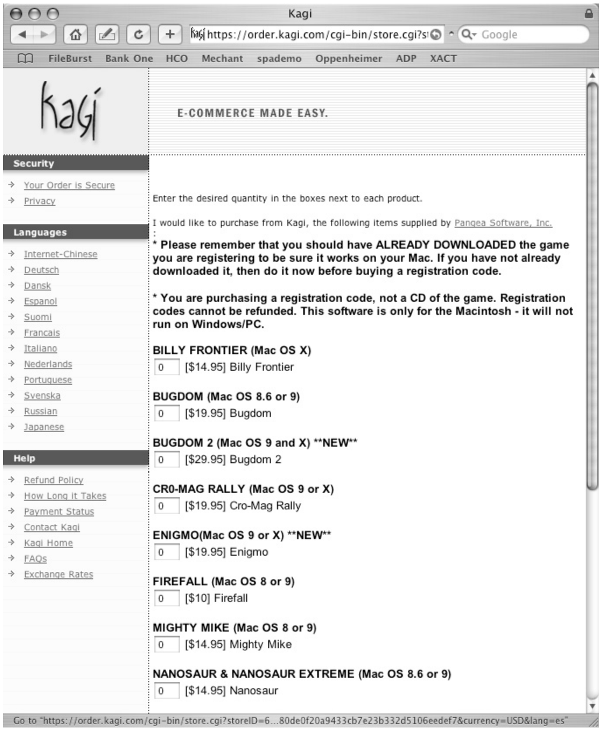
Check Which Item(s) You Wish to Purchase:

Purchase	Qty.	Item
☐	1	BILLY FRONTIER (MacOS X)
☐	1	BUGDOM (MacOS 8.6 or 9)
☐	1	BUGDOM 2 (MacOS 9 and X)
☐	1	CR0-MAG RALLY (MacOS 9 or X)
☐	1	ENIGMO(MacOS 9 or X)
☐	1	FIREFALL (MacOS 8 or 9)
☐	1	MIGHTY MIKE (MacOS 8 or 9)
☐	1	NANOSAUR 1 (MacOS 8.6 or 9)
☐	1	NANOSAUR 2 HATCHLING (MacOS X)
☐	1	OTTO MATIC (MacOS 9 OR X)

Figure 18-1: The Pangea Software store front for downloaded games

Once the user purchases a serial number for a downloaded game they will be emailed the serial number.

It will cost you a few thousand dollars to get a store like this up and running, but if you cannot afford that then there is another option open to you. There is a well-known company named Kagi that, for many years, has offered a shareware payment collection service, and they too can spit out serial numbers for these kinds of purchases. The two biggest downsides to using Kagi are that they charge a pretty big fee for each sale (usually around $2.50), and it takes a while to get paid. If you make a sale on January 1st, you will likely not get paid for that sale until around February 24th. In contrast, a sale on a custom order page might only cost you $1.00 in credit card fees, and the money will be deposited into your account the next business day.



Enter the desired quantity in the boxes next to each product.

I would like to purchase from Kagi, the following items supplied by Pangea Software, Inc.
:

*** Please remember that you should have ALREADY DOWNLOADED the game you are registering to be sure it works on your Mac. If you have not already downloaded it, then do it now before buying a registration code.**

*** You are purchasing a registration code, not a CD of the game. Registration codes cannot be refunded. This software is only for the Macintosh - it will not run on Windows/PC.**

BILLY FRONTIER (Mac OS X)
0 [$14.95] Billy Frontier

BUGDOM (Mac OS 8.6 or 9)
0 [$19.95] Bugdom

BUGDOM 2 (Mac OS 9 and X) **NEW**
0 [$29.95] Bugdom 2

CR0-MAG RALLY (Mac OS 9 or X)
0 [$19.95] Cro-Mag Rally

ENIGMO(Mac OS 9 or X) **NEW**
0 [$19.95] Enigmo

FIREFALL (Mac OS 8 or 9)
0 [$10] Firefall

MIGHTY MIKE (Mac OS 8 or 9)
0 [$14.95] Mighty Mike

NANOSAUR & NANOSAUR EXTREME (Mac OS 8.6 or 9)
0 [$14.95] Nanosaur

Go to "https://order.kagi.com/cgi-bin/store.cgi?storeID=6...80de0f20a9433cb7e23b332d5106eedef7&currency=USD&lang=es"

Figure 18-2: The Kagi storefront

Even though we have our own custom store to handle all of our orders, I still keep my Kagi account up and running in case there are malfunctions with my store and I need to point customers elsewhere. It doesn't cost anything to have a Kagi account, so it's good to always have it as a standby.

We've actually used a few different custom store technologies over the years, but by far the best one is the one that we use today that XACT Commerce built for us:

www.xactcommerce.com

They've designed a fantastic order page and processing system, and they are very quick to fix any problems that come up. If you're going to sell your games online then I highly recommend contacting these folks to have them build your store for you.

Manufacturing & Order Fulfillment

The other part of this equation is manufacturing. You'll need boxed CD versions of your games for any retail sales, plus, you need something in a box for people to buy as gifts. Don't underestimate the power of Christmas. During a typical Christmas buying season our sales will be 3-4x higher than normal.

You've got a lot of options open to you when it comes to manufacturing, but remember that a penny saved is a penny earned, so don't go overkill on the packaging for your game. In the old days when game boxes were really large, it would cost us about $2.50 per unit. These days we ship everything in standard DVD style cases, and that has reduced the cost to about $1.50 per unit. In addition to that, our shipping costs have gone way down because a DVD case fits quite nicely in a standard UPS letter shipper. The old boxes required larger packing envelopes, and UPS charged us accordingly.

Remember that this is the Mac we're talking about, so sales volumes are not going to be in the hundreds of thousands. If you're also offering a download version of the game then assume that most of your sales are going to be from that, not the CD version. What I'm trying to say is that you don't want to have 10,000 units of your game manufactured since you'll probably never sell all of those. Do small builds of 2,500 units at a time. These smaller builds will cost a little more per unit, but at least you won't have excess inventory to use as firewood for the next eight years.

There are many companies out there who will manufacture your packages for you, but you want to use a manufacturer who is also an Order Fulfillment company. An Order Fulfillment company receives the orders from your order page, and processes them in their warehouse. You don't have to worry about stuffing games into envelopes and all that mess. They do it for you… for a fee, of course. The typical fulfillment fee on a package will cost you around $3.50 – more for international orders.

The company that we have used for the last 5 years is Software Packaging Associates in Cincinnati, OH.

www.softpack.com

In addition to the fact that we like working with these people, there is one other thing that makes them a good fulfillment option: location. Ohio is centrally located in the US which means that Ground shipping to anywhere in the country is only a few days. If you choose a fulfillment company in California, then any of your customers on the East Coast are going to have to wait a very long time for their packages to arrive. It is always a good idea to find a fulfillment company that is geographically central.

The other great thing about Software Packaging Associates is that they're in DHL's hub city: Cincinnati. We do all of our international shipping with DHL because they have the best international rates, and since we're shipping from their hub city we've managed to get an incredibly good deal from them. If you were shipping from, say Phoenix, DHL would charge you more because every package still has to make its way to Cincinnati. So, it's a good idea to use a fulfillment company that's located in a DHL, UPS, or Fed-Ex hub city.

One final note about shipping: make these companies fight for your business. Don't just call up UPS and ask for an account. Ask them what kind of discount you can get since you're going to "be using UPS a lot to ship a lot of packages." Tell them you're also talking to the other guys, and you want the best deal they can give you. I kid you not when I say that you can get discounts of 30-50% right off the bat if you try hard enough.

Index

AbsoluteToNanoseconds, 65, 66
AGL, 26-31, 37, 38, 220, 223, 224
aglChoosePixelFormat, 26, 27
AGLContex, 37
aglCreateContext, 26, 28
aglEnable, 27, 29
aglSetCurrentContext, 27-30, 210, 221
aglSetDrawable, 27, 28, 33
aglSetFullScreen, 27, 29, 34
aglSwapBuffers, 29, 30, 211, 222
aglUseFont, 31
AGP, 60
AIFF, 85
alBufferData, 117, 118
alcCreateContext, 114
alcMakeContextCurrent, 114
alcOpenDevice, 114
alDistanceModel, 114
alEnable, 114, 115
alGenBuffers, 117, 118
alGenSources, 120
alListenerfv, 118, 119
alpha, 27, 195, 264
alSetInteger, 114, 115
alSourcef, 120, 121
alSourcefv, 120, 121
alSourcei, 120, 121
alSourcePlay, 120, 122
AltiVec, 49-52
alutInit, 114, 115
alutLoadWAVEFile, 116
alutLoadWAVFile, 115, 117, 118
alutUnloadWAV, 117, 118
AnaBuilder, 224-227
Anaglyph Glasses, 203, 209, 218
AppendMenu, 23
ATI, 37, 56
audio, 85-123
AutoSleepControl, 83
Axis, 141, 154, 157-159, 174

backfaces, 188
bandwidth, 266, 268-270
BG3D, 175, 179, 180, 185, 186, 188, 197-202
bit-depth, 12, 17, 28
black & white mode, 216
BlockMove, 22, 23
blue-line sync, 220-224
Browser, 232, 233
BSD Sockets, 229, 238, 247
Buffers, 113, 116
Bugdom, 2, 140, 198, 213, 257, 262
BuildGreyCurve, 217
BuildResolutionMenu, 18, 22
bump mapping, 9
button names, 167, 168

cache, 38
Cached Mode, 59
CalcDisplayVRAM, 46
CalcFramesPerSecond, 65, 66, 71, 77
calibrate, 158
callback, 75-78, 81, 83, 90, 93, 96, 97, 100, 102, 154, 231, 233, 234, 251
CallMeWhen, 90, 93, 97
capturing, 12
Carbon Events, 75-82, 129, 134
Carmageddon, 67
cartoon shading, 9
CFArrayGetCount, 13, 153
CFArrayGetValueAtIndex, 14, 15, 235, 236
CFBundleCopyResourcesDirectoryURL, 87, 88
CFBundleGetMainBundle, 17, 87
CFDictionaryGetValue, 14-16, 145-150, 153-158
CFNetServiceBrowserCreate, 232, 233
CFNetServiceBrowserScheduleWithRunLoop, 232

CFNetServiceBrowserSearchForServices, 232, 233
CFNetServiceCreate, 230, 231
CFNetServiceRegister, 230, 231, 232
CFNetServiceScheduleWithRunLoop, 230, 231
CFNetServiceSetClient, 230, 231
CFNumberGetValue, 14-16, 46, 145, 146, 149
CFURLGetFSRef, 87, 88
CGDisplayAvailableModes, 13, 15
CGDisplayBestModeForParameters, 11, 12
CGDisplayCapture, 11, 12, 34
CGDisplayIOServicePort, 46
CGDisplayRelease, 33
CGDisplaySwitchToMode, 12, 13
CGGetDisplayTransferByFormula, 70
CGMainDisplayID, 11, 13, 46
CGSetDisplayTransferByFormula, 69, 70, 71, 72, 73
CGSetDisplayTransferByTable, 69, 73
ChangeChannelRate, 109
ChangeChannelVolume, 109
CheckForAltiVec, 49
CheckForConnectionRequestFromClient, 239
CHUD, 6, 8, 9
Classic, 102
Client, 229, 232, 239, 242, 243
CodeWarrior, 3-5, 11, 53
Color Balancing, 212, 214, 226
ColorBalanceRGBForAnaglyph, 214
computer name, 231
copy protection, 1, 249, 252, 254
Core Audio, 98, 115
Core Foundation, 15, 147, 154, 191
Core Graphics, 11, 12, 69
CreateQueueForHIDElements, 170
CreateWindowFromNib, 18, 20
Cro-Mag Rally, 2, 112, 198, 269
CRT, 204, 213, 219, 224
Crystal Eyes, 224

debugging, 11, 30, 62, 73, 174
default directory, 258
Demos, 267
Device, 139, 141, 143, 144, 148-153, 169
DHL, 274
dictionary, 15, 144, 147, 148, 153-156, 158
diffuse coefficient, 195
diffuse color, 192-195
digital camera, 224, 225
distance model, 114
Distributed Builds, 4
distributors, 269
Dolby AAC, 85
Doppler Factor, 122
Doppler shift, 111, 119, 121, 122
Doppler Velocity, 122
DoScreenModeDialog, 18, 20
Double-buffering, 63
DownloadURL, 250-252
D-Pad, 140, 159, 160
draw buffer, 44
draw context, 26-31, 33, 37, 44
DrawBlueLineSync, 222
DrawPictureIntoGWorld, 263
DrawSpriteWithBeginEnd, 45
DrawSpriteWithVertexArray, 45

Element, 141, 152-154, 156-158, 169, 171
endian, 239
EndOfSongCallback, 90, 93, 96, 97
Enigmo, 2, 53, 66, 103
Enter2D, 33-35
event handler, 20, 80, 130, 134, 138
event loop, 21, 75, 76, 78
Event Loop Timers, 75, 76
Exit2D, 33-35
exporter plug-in, 175, 197

Fencing, 62
FileBurst, 269
filter gels, 213
FindHIDDevices, 142, 143
FindSilentChannel, 105, 106
flicker, 28, 204, 220, 223
floating point precision, 67
flushing, 246
font, 31, 32
for Generic Desktop devices, 148
FoundAHost, 234

fragment shader, 9
frame rate, 44, 55, 60, 65-67, 115, 203
frames per second calculation, 67
framework, 108, 259
frsqrte, 52-54
FSGetCatalogInfo, 87, 88
FSMakeFSSpec, 87-91, 116, 117, 190, 191, 196, 258
FSRef, 86-88, 117
FSSpec, 86-91, 103, 116, 117, 190, 191, 251, 258, 263
full-screen, 11, 27-30, 33-35, 78, 134, 220

game demos, 268, 269
Game Sprockets, 11, 139
gamepad, 125, 139-142, 167, 169, 173, 174
gamma fades, 69, 73, 223
GammaFadeInOneFrame, 72, 77
GammaFadeOut, 71, 72
GCC, 4
Gestalt, 49, 50
GetControlByID, 19, 21, 23
GetControlData, 23
GetControlValue, 19, 21
GetEventClass, 130, 132
GetEventKind, 20, 130, 132
GetEventParameter, 20, 81, 131, 132, 135, 136, 137
GetFullPathFromFSSpec, 116, 117
GetIPAddressOfHost, 234, 235, 243
GetKeys, 125, 126, 129, 133
GetMainEventLoop, 76
GetMouse, 133, 134
GetMouseScreenCoord, 133
GetMovieDuration, 89, 92
GetMoviePreferredRate, 89, 92
GetMovieTime, 89, 92
GetMovieTimeBase, 90, 93
GetMyApplicationResourcesFolder, 87
GetTrackMedia, 90, 92
ghosting, 212, 213, 218, 219, 220
glBindTexture, 93
glBindVertexArrayAPPLE, 57, 58, 61
glCallLists, 31, 32
glClear, 29, 30, 210, 211, 221
glColor, 38, 44
glColor4f, 32, 40
glColorMask, 210-212
glDisable, 32, 38, 39, 43
glDisableClientState, 45, 61
glDrawElements, 45, 61
glEnable, 37-39, 43, 44
glEnableClientState, 45, 58, 59, 61
glFinish, 33, 62, 63
glFlush, 33
glFlushVertexArrayRangeAPPLE, 58-60
glGenLists, 31
glGenVertexArraysAPPLE, 56
glGetString, 55
glHint, 43
glListBase, 31, 32
glOrtho, 32
glPopAtrib, 40
glProgramStringARB, 9
glPushAttrib, 40, 41
glRasterPos2i, 31, 32
glTexCoordPointer, 45
gluPerspective, 207
glVertexArrayParameteriAPPLE, 57, 58, 60
glVertexArrayRangeAPPLE, 57, 58, 60
glVertexPointer, 45, 61
GrafPtr, 88, 91, 133

hackers, 253, 254, 255
hat switch, 140
HFS, 190, 191, 262, 263
HID Manager, 125-174
Host, 229, 232-239, 242, 243

InitGammaValues, 70
InitMyCommandEventHandler, 80, 129
InitMyHIDManagerStuff, 142
InitSoundManagerChannels, 98
input devices, 139, 140
Input Sprocket, 139-141, 167, 172
Inside Mac Games, 265, 266
InstallEventLoopTimer, 76, 77
InstallWindowEventHandler, 18, 20
Interface Builder, 5, 6, 21, 78-80
Internet addressing, 239

IOCreatePlugInInterfaceForService, 151
IOIteratorNext, 145, 147
IOMasterPort, 142, 143
IORegistryEntryCreateCFProperties, 145, 147
IORegistryEntryCreateCFProperty, 46
IOServiceGetMatchingServices, 143, 144
IOServiceMatching, 143, 144
IP address, 234-237, 239, 242, 243
IsDataWaitingToBeRead, 239-241, 244, 245
IsInternetAvailable, 250
IsSocketReadyForWrite, 244, 245

joystick, 125, 139-142, 150, 157, 158, 171, 174

Kagi, 271, 272
keyboard, 75, 83, 125, 129, 130, 132, 133, 139, 141, 142, 144, 145, 148, 150, 157, 167, 168, 172, 173, 174
keyboard events, 129
key-down, 132
KeyMap, 125, 126, 127, 128, 129, 132
key-up, 132

Lambert shader, 192
Layers, 179, 182-185
Lightwave, 185
Listener, 112, 113, 118, 119, 121, 238, 242
Little Snitch, 253, 254
LoadMediaIntoRam, 90, 92
LoadMovieIntoRam, 90, 92
LoadWAVFiles, 116, 117
localization, 259, 261, 262
loopback points, 112
looping, 85, 93, 96, 97, 100, 105, 112, 120
LoopSoundCallback, 101
luminosity, 69, 214, 215, 217, 218

Mac Game Files, 266
Mac Gamer, 266
MacAddict Magazine, 267
MacCentral, 266
MacGameFiles, 268
Macro Optimizations, 37
Macworld Magazine, 266, 267
mail order catalogs, 269
makefile, 199, 200
manufacturing, 270, 273
marketing, 1, 112, 265, 267, 269
Material, 186
Matrix Multiplication, 51
Maya, 175-197
Menu Bars, 78
meshes, 55, 179, 184-186, 202
modifiers, 125, 169
mouse delta values, 134
MoviesTask, 93-97
MP3, 85
MPDelayUntil, 95
music, 85, 93, 94, 97, 100, 113
MyCFArrayParse, 153
MyEventHandler, 81, 82, 130, 135

Nanosaur 2, 7, 8, 47, 60, 63, 66, 93, 112, 171, 180, 213, 218, 219, 224, 251, 262
Net Sprocket, 229
networking, 229, 247
NewCallBack, 90, 93
NewEventHandlerUPP, 18, 20, 81, 130
NewEventLoopTimerUPP, 76, 77
NewMovieFromFile, 89, 91
Newton-Rhapson refinement, 52, 53
nib, 17, 18, 20, 78, 81, 87, 88, 196
NumToString, 22, 23, 237
Nvidia, 56

OAL_Init, 114
OAL_PlaySound3D, 119
OAL_SetListenerInfo, 118
OGL_AssignVertexArrayRangeMemory, 56, 57, 61
OGL_CreateDrawContext, 26
OGL_DisableLighting, 39, 42, 222
OGL_DrawScene, 29, 77, 83, 210, 221
OGL_DrawString, 31
OGL_EnableLighting, 39, 42
OGL_InitFont, 30
OGL_InitVertexArrayRanges, 55
OGL_PopState, 42, 223
OGL_PushState, 40, 222

OGL_UpdateVertexArrayRanges, 57
OGLMatrix4x4_Multiply, 50
OGLMatrix4x4_Multiply_AltiVec, 51
online sales, 270
OpenAL, 85, 98, 111-123
OpenGL Profiler, 8
OpenMovieFile, 89, 91
optimizations, 37, 46, 49, 55
Order Fulfillment, 273

packaging, 273
ParseAllHIDDevices, 143, 144
Phong shader, 192
phoning home, 250
piracy, 249, 254, 255
PlayEffect, 106, 108
PlaySong, 88, 90
Polling, 169, 171
polygons, 25, 184, 186, 187
port, 46, 89, 91, 134, 140, 142-144, 230, 231, 235, 238, 239, 242, 243
PowerPC, 49, 50, 52, 53, 54
precompiled header, 53
Predictive Compilation, 4
Preferences folder, 258, 259
PrerollMovie, 90, 92
Project Builder, 3
projection matrix, 32, 207
pthread_create, 94

Queuing, 169
Quickdraw 3D, 198, 201
Quicktime, 85-100, 113, 192, 263, 264

Rainbow Symphony, 219
ReadNetData, 246, 247
RecurseDictionaryElement, 150, 153, 156
recv(), 247
rendering, 8, 11, 25, 27-29, 31, 44, 46, 55, 63, 73, 115, 203, 204, 210, 212, 216
Rendezvous, 229-238
ResEdit, 6
resolution, 12, 13, 16, 17, 23, 32, 33, 34, 47
resource, 18, 20, 78, 79, 80, 86, 102, 103, 104, 110, 267
Resource Manager, 104, 110
Resourcerer, 6
Resources Folder, 86, 87
retail, 269
retail stores, 270
reverb, 111
RunApplicationEventLoop, 78, 80
RunAppModalLoopForWindow, 19

sample code, 9
scroll wheel, 134, 135, 137, 138
select(), 241
send(), 244-247
serial numbers, 249-254, 271
SetControlValue, 23
SetGammaFade, 71, 72
SetMenuBarFromNib, 80, 129
SetMoviePlayHints, 89, 92
SetMyMainLoopEventTimer, 76-78
Shader Builder, 9
shared libraries, 195
Shared Mode, 59
shareware, 271
Shark, 6, 7, 8, 49
Sherlock translations, 261, 262
Shutter Glasses, 204, 219, 220, 221, 223
Sleep, 83
slide bar, 227
SndChannelStatus, 105
SndDoImmediate, 100, 101, 106, 107, 109
SndNewChannel, 99, 100, 101
Socket, 238-246
socket(), 239, 243
Software Packaging Associates, 273
Song Volume, 96
Sound Channels, 98
sound interpolation, 92
Sound Manager, 98-123
Sound Resources, 102, 103, 112
SoundEdit 16, 102, 103
Sources, 113, 121, 122
split-screen, 112
StartMovie, 90, 93, 97, 98
stereo 3D, 203, 208, 212, 220, 224
Stereo Graphics Corporation, 224

Surface Shaders, 186, 192

TCP/IP, 231, 236, 242, 246
textures, 46, 47, 93, 193, 199, 202, 213, 216, 220
the menu item commands, 80
Threads, 94, 95
thread-safe, 96
Timer events, 75, 80, 82, 95
toe-in, 204, 205
Transform Hint, 43
transparency, 192-195
triangulation, 177

UDP, 231, 246
UNIX, 190, 191, 229
UpdateKeyMap, 126
updates, 5, 169, 249, 250, 267, 269
UpdateSystemActivity, 83
UPS, 273, 274
UpTime, 65, 66, 94, 95
URL, 87, 88, 250, 251
Usage Page, 141, 142, 147, 148, 157, 158
USB, 140, 144, 146, 148, 150, 169

VAR, 55-63, 210, 221
vec_ld, 51
vec_madd, 51
vec_splat, 51
vec_st, 52
vector, 37, 44, 49-52, 118-121, 209
VerifyAndAddHIDElement, 153, 154
VersionTracker, 266
Vertex Array Range, 46, 55, 56, 202
Vertex Arrays, 44-46, 55, 63, 199
vertex colors, 45, 55, 60, 62, 189
vertex normals, 43, 53, 188, 198
vertical parallax, 205, 206
video mode, 12, 17, 34
Virtual Reality, 203, 204
Visco Entertainment, 270
volume, 88-91, 96, 98, 101, 107-114, 120, 121, 258, 263, 269
VRAM, 28, 46, 47, 59, 60, 93, 220

WAV, 85, 112, 116-118, 120
WAV sound file, 116
Weekend Warrior, 2, 198
world-space, 32, 188

XACT Commerce, 272
Xcode, 1, 3-5, 50, 53, 86, 109, 110, 195

z-buffer, 28, 30, 44, 211