3D for iPhone® Apps with Blender and SIO2

Your Guide to Creating 3D Games and More with Open-Source Software

Tony Mullen

Foreword by Romain Marucchi-Foino, author of SIO2

SYBEX® SERIOUS SKILLS.

3D for iPhone® Apps with Blender and SIO2

3D for iPhone® Apps with Blender and SIO2

YOUR GUIDE TO CREATING 3D GAMES AND MORE WITH OPEN-SOURCE SOFTWARE

TONY MULLEN

Wiley Publishing, Inc.

Acquisitions Editor: Mariann Barsolo
Development Editor: Kathi Duggan
Technical Editor: Romain Marucchi-Foino
Production Editor: Rachel McConlogue
Copy Editor: Judy Flynn
Editorial Manager: Pete Gaughan
Production Manager: Tim Tate
Vice President and Executive Group Publisher: Richard Swadley
Vice President and Publisher: Neil Edde
Book Designer: Caryl Gorska
Compositor: Chris Gillespie, Happenstance Type-O-Rama
Proofreader: Nancy Bell
Indexer: Ted Laux
Project Coordinator, Cover: Lynsey Stanford
Cover Design: Ryan Sneed
Cover Image: The squirrel character on the cover is © copyright Blender Foundation | www.bigbuckbunny.org. Game images featured on the cover include images from Cerebrii © 2009 Richard Stoner, Shootin' Annie © 2009 Tony Mullen, Black Sheep © 2009 Rarebyte OG, Dropoly © 2009 Inovaworks, Touch Ski 3D © 2009 Michael Perl, Naquatic, Ivory Tower © 2009 APC LLC (4igames.com), Manic Marble © 2009 Rich Olson, Guns of War © 2009 Dumbgames.net, iPunch © 2009 iDev.com, iPentris © 2009 Rarebyte OG, and Meditation Garden © 2009 SIO2 Interactive.

ISBN: 978-0-470-57492-8

Library of Congress Cataloging-in-Publication Data

Mullen, Tony, 1971-

3D for iPhone apps with Blender and SIO2 : your guide to creating 3D games and more with open-source software / Tony Mullen.

p. cm.

ISBN 978-0-470-57492-8 (paper/website)

1. Computer games—Programming. 2. SIO2 (Electronic resource) 3. iPhone (Smartphone)—Programming. I. Title.

QA76.76.C672M858 2010

794.8'1526—dc22

2009047260

10 9 8 7 6 5 4 3 2 1

For Yuka and Hana

Acknowledgments

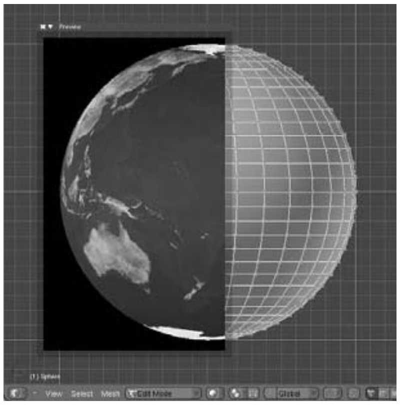
I'm very grateful to everyone involved in the creation of this book and the software it deals with. In particular, I'd like to thank Romain Marucchi-Foino, author of the SIO2 game engine, for his efforts in creating such a useful set of tools for the game-programming community as well as for his great help as tech editor of this book. I'd also like to thank Ton Roosendaal and the Blender developers for their tireless work to make Blender the fantastic piece of software that it is. ■ In addition to the developers, I'd like to thank the many users and game creators who helped me directly or indirectly through their posts on the SIO2 forum. I'm especially grateful to the game creators who allowed me to use images from their games in this book. Their work provides a great showcase for the power of the Blender/SIO2 pipeline. ■ This book wouldn't have been possible without the collaboration of the editorial and production team at Sybex, and I'm very grateful to everyone who had a hand in bringing it to publication, in particular Mariann Barsolo, Pete Gaughan, Kathryn Duggan, and Rachel McConlogue. These are just the people I interacted with most regularly on this project; there are many other people whose contributions I am also grateful for. ■ I'd also like to thank my students and colleagues at Tsuda College, Tokyo, for their support. In particular I'd like to thank my colleague Associate Professor Akihasa Kodate for suggesting I take over teaching his computer graphics class. The deepening of my knowledge of OpenGL that I gained through preparing that class was a great help for me in coming to grips with SIO2 and game development for the iPhone. ■ Finally I'd like to thank my wife, Yuka, and our daughter, Hana, for their love, support, and patience!

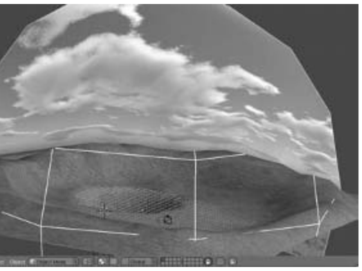
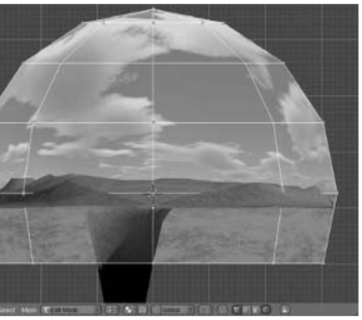
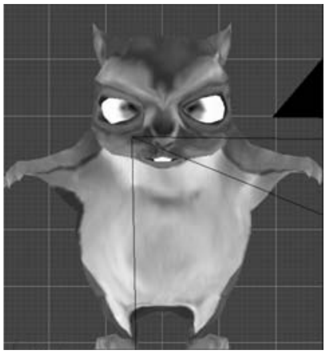
About the Author

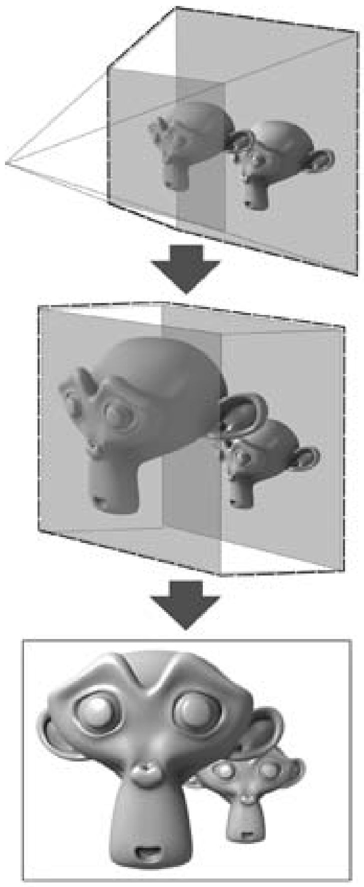
Tony Mullen is a college lecturer, programmer, animator, filmmaker, and writer living in Tokyo. In the past, he has worked as a newspaper cartoonist, graphic designer, and computer science researcher, among other things. Since discovering Blender, he has been involved in CG animation to the point of obsession, but he also maintains a keen interest in stop-motion and traditional animation techniques, notably as the lead animator and codirector of the 16mm film *Gustav Braüstache and the Auto-Debilitator* (winner of the Best Narrative Short award at the New Beijing International Movie Festival in 2007 and Best Filmstock Film at the San Francisco Frozen Films Festival in 2008) and other independent shorts. Along with his filmmaking partner, Rob Cunningham, he was short-listed for Seattle alternative weekly newspaper *The Stranger's* Genius Award for Film in 2008. He is an active member of the Blender community and one of the original members of the Blender Foundation's Trainer Certification Review Board. He is the author of numerous articles and tutorials on Blender and graphics programming for the Japanese magazine *Mac People* and of the Sybex books *Introducing Character Animation with Blender*; *Bounce, Tumble, and Splash! Simulating the Physical World with Blender 3D*; and *Mastering Blender.*

CONTENTS AT A GLANCE

Contents





Foreword

When I first heard about the iPhone and its gaming capacities, I knew right from the start that it was going to be big; Apple has always been revolutionary in all its product lines. Knowing that the device would have support for OpenGL ES and OpenAL technology, I was excited to get the SDK.

I wanted to create a 3D engine built around it and provide a free, flexible, and scalable solution that users could start using out of the box. Being a fan for quite a while, when it came to which 3D editor I should use for the engine integration, Blender was the most obvious choice. Even after almost three years using it on a daily basis, Blender never stops impressing me, and I can't stop praising how much this software can do for the size of its download.

I grabbed my first copy of the iPhone SDK a day after its release and started spending all my free time building a game engine from scratch for the platform. Three months later, I released the first online version of SIO2—a version that ironically was actually never tested on a real device because the iPhone was not accessible in the country I was in. It took more than two revisions of the engine before I was able to test it on the device and the Blender/Xcode/iPhone game development pipeline that I had created.

Now a year later, SIO2 is one of the top 3D game engines used on the App Store, and it wouldn't have been possible without Blender and its community. I'm glad that I chose to use Blender since day one—using it as a world editor in order to provide a WYSIWYG interface for SIO2 was just a perfect match.

When Tony first contacted me, I was excited by the idea of creating this book, packed full of knowledge that will give a good kick start to anybody on their iPhone game creation learning curve with Blender and SIO2. I was honored to be able to participate in the creation of *3D for iPhone Apps with Blender and SIO2*, and I hope you will enjoy it and find it as useful as we do.

— *Romain Marucchi-Foino*

Author of SIO2, the free, open-source 3D game engine for the iPhone and iPod Touch

Lead 3D Programmer at SIO2 Interactive

Shanghai, November 2009

Introduction

Congratulations! By cracking open this book, you have just taken the first step into the exciting, challenging world of interactive 3D content creation for the hottest handheld devices around: the iPhone 3G and its sleek sister, the iPod Touch. If you're coming from a related area of game design, programming, or 3D asset creation, you'll find the information you need here to transfer your skills to the realm of iPhone/iPod Touch development. If you are a complete newcomer to the technology, this book will give you the basis you need to begin creating games, visualizations, virtual worlds, and whatever interactive 3D content you can dream up and get them running on your iPhone or iPod Touch. You'll take advantage of the device's cutting-edge multi-touch interface technology, the physical sensitivity of its built-in accelerometer, and the brilliant clarity of its ultra-high resolution screen. By mixing in a little bit of hard work and ingenuity of your own, you will be able to develop your very own 3D applications to sell on the iTunes App Store, joining the growing number of budding entrepreneurs who are leading the way in innovation for the most exciting new game and application platform around.

What You Will Learn from This Book

This book introduces a powerful, straightforward pipeline for 3D content creation using Blender, the SIO2 game development application programming interface (API), and Apple's own Xcode and iPhone software development kit (SDK). With this combination of tools at your disposal, you'll quickly find yourself pushing the envelope of interactive 3D content creation for the iPhone and iPod Touch.

In this book, you'll learn how to create 3D assets and environments in Blender for use with the SIO2 engine, how to export them, and how to work with them using the SIO2 SDK. You'll learn how to use Blender's texture baking functionality to create convincing surface and lighting effects. You'll learn how to use the multi-touch interface to control

3D objects, camera movement, and characters. By the time you finish reading this book, you'll be in a good position to begin working on a 3D project of your own for the iPhone.

This book is not intended to replace the official tutorials and code samples that accompany the SIO2 SDK download. Those tutorials and code samples, which can be found in their respective project directories in the SIO2 SDK top-level directory, provide an indispensable sample of SIO2 functionality and cover much more material than is dealt with in this book. However, the information in the SIO2 code samples is dense, and diving right into reading the code can mean a steep learning curve for people who are new to the SIO2 engine. This book is an attempt to give a gentler introduction and to include some background information that will help you get more out of the tutorials and code samples. In Appendix C of this book, you'll find an overview of the content provided in the official tutorials. By the time you reach that appendix, you should be able to dive into the code samples with ease.

Why Blender/SIO2?

If you're reading this, you probably don't need to be told why somebody might want to create interactive 3D content for the iPhone or iPod Touch. Just the fact that you *can* is reason enough to want to! But there are a number of options for 3D content creation for the iPhone platform, and it's reasonable to wonder what the Blender/SIO2 pipeline has going for it.

For one thing, it's free. This is nice no matter what your budget is, but it's an even bigger deal when you consider that many of the most exciting and innovative projects available on the iTunes App Store are created by individuals or small studios. The iTunes App Store itself represents something of a revolution in software development and distribution, with many thousands of independent developers making their applications available to many thousands of users for very low prices. The path to success for such independent developers is a combination of solid programming skills, a killer concept, and low development overhead.

Blender is a free and open-source 3D content creation application that rivals applications costing thousands of dollars in terms of functionality and stability. A rapidly growing number of commercial animation and game studios have adopted Blender as their core 3D application for reasons of cost and flexibility. Likewise, SIO2 is free to use with only a minimal requirement of attribution, which is lifted for users of an inexpensive license. Like 3D applications, commercial game engines that offer iPhone compatibility also run into the thousands of dollars. For individual independent developers, the money saved can mean a lot of new hardware and some fancy dinners (or, for the frugal, it could just mean paying a couple of months of rent). For a studio, the savings is multiplied.

But being free isn't the whole story. In fact, the Blender/SIO2 pipeline isn't the only free solution to 3D development for the iPhone. The iPhone platform supports the OpenGL ES graphics API natively, the iPhone SDK comes with OpenGL ES built in, and there are numerous tools for developing and optimizing OpenGL ES code. It is possible to program games and 3D effects directly in OpenGL ES without using any high-level content creation tools at all. However, this is not the easiest or most intuitive way to work.

Modeling and animating are best carried out in a What-You-See-Is-What-You-Get (WYSIWYG) 3D environment such as Blender. The SIO2 API enables you to work directly with assets created in Blender and offers a high-level programming interface to greatly simplify the actual coding you have to do to obtain advanced effects. SIO2 also enables high-level functionality such as Lua scripting and networking.

There are also tools for working with 3D and Blender assets in the iPhone. The Oolong project in particular bears mentioning. Oolong is an open-source API with goals similar to the goals for SIO2. Like SIO2, it uses Bullet physics. It is also well integrated with the cross-platform Gamekit prototyping sandbox, which makes it worth investigating for advanced game programmers who are interested in cross-platform game development. As I write this, development is underway to add functionality to Oolong that will enable it to read Blender `.blend` files directly, which will be an exciting development for open-source game creators. Nevertheless, using Oolong requires a greater degree of game development experience and C++ coding skill to get started, and it is not as well supported by tutorials and code samples. For these reasons, I chose to focus on SIO2 for this book.

All in all, the combination of Blender and the SIO2 engine offers a powerful solution at a negligible fraction of the cost of the big commercial mobile 3D game pipelines while giving you WYSIWYG content creation and an accessible high-level programming environment.

What Else You Need to Know

Although a big part of their appeal is their apparent simplicity, the iPhone and iPod Touch are serious platforms, and the coding you learn about in these pages is serious software development. Creating complete applications for the iPhone and iPod Touch is itself an involved topic. At the same time, 3D programming in general has numerous challenges of its own.

For this reason, any background knowledge you have already about programming in C or OpenGL, computer graphics, 3D content creation, or iPhone development will be of great help to you as you work your way through this book. But because I can't assume you have *all* of this background, I'm going to proceed under the assumption that you have none of it. I'm going to do my best to explain everything in sufficient detail that it should at least make sense to even a completely inexperienced reader. If you find that the book progresses too slowly for you in some places, feel free to skip ahead. Likewise, if you start feeling like you're a bit in over your head reading discussions about computer graphics programming or Blender use, please refer to the appendices in this book, which I hope will function as quick tutorials to get you up to speed on the relevant topics. Throughout the book, I will give tips and references on where to find more in-depth information about a variety of topics. I strongly suggest that you follow these leads and track down as many supplemental resources as you can get your hands on.

The most important requirement for having success with this book, therefore, isn't any specific knowledge but rather an attitude. You may not know C or OpenGL ES, but you must be open to learning at least some of it. You may not be a hotshot at Blender modeling or animation, but you should be willing to be proactive about acquiring these skills. You may not have the slightest idea what a matrix is right now, but you must have enough faith in yourself to believe that you can learn the basic mathematical concepts that 3D game programming demands.

This book will get you well on your way. But it won't be the end of the story. Once you've worked your way through this book, you'll want to find other resources to fill in the gaps that remain in your knowledge. The official tutorials and code samples are an obvious next step. Appendix C offers an overview of what those tutorials contain, so you can get straight to learning the advanced functionality that interests you most.

Who Should Read This Book

This book is for anybody interested in creating 3D applications for the iPhone or iPod Touch. I think you'll find that following the tutorials in this book is the easiest and most direct path to learning what you need to know to create 3D content for the iPhone platform. That's not to say that the book is simple or a "beginner's book." If you don't have computer programming experience, you may find much of this book to be rough going. No single book can take you from 0 to 60 as a mobile game developer on its own; however, this book will at least get your foot on the pedal.

How to Use This Book

The best way to read this book is from beginning to end, all of the chapters in order. Several of the chapters follow explicitly and directly upon the preceding chapter, but there are also more subtle dependencies, and to avoid redundancy, the later chapters were written with the assumption that you have read the previous chapters.

I recommend working through each chapter's content from beginning to end, as it is described in the chapter, taking the SIO2 template project as the launching point. None of the tutorial projects are trivial, and the process of getting your project running based on what you read will give you ample opportunity to debug and double-check your code. At the end of each chapter, the code described in the chapter is printed in the context of the original template file. If you run into problems during the chapter, check this code to see where you might have taken a wrong turn. Finally, you can double-check everything by comparing your project to the corresponding project in the downloadable project archive that accompanies this book.

How This Book Is Organized

As I mentioned, the content of this book is roughly sequential, and concepts introduced early are referred to later. However, there are only a few strict dependencies. The first mid-sized project of the book is split over Chapter 3, Chapter 4, and Chapter 5, so those chapters should be read as a unit. The second mid-sized project is split between Chapter 6 and Chapter 7, and the third and final project is described over the course of Chapter 8 and Chapter 9, so those pairs of chapters should also be regarded as interdependent.

Here is a quick overview of what each chapter and appendix contains:

Chapter 1, "Getting Started with 3D Development for the iPhone," introduces the basics of iPhone development in Xcode and shows you how to build and run a project based on the SIO2 template.

Chapter 2, "Introducing Graphics Programming in SIO2," looks at some fundamentals of graphics programming in OpenGL|ES through the lens of SIO2 and the iPhone SDK.

Chapter 3, "Saying Hello to the Blender/SIO2/iPhone World," walks you through the creation of a simple Blender 3D scene featuring a model of planet Earth for use with SIO2.

Chapter 4, "Going Mobile with SIO2," picks up where Chapter 3 leaves off. This chapter shows you how to import the assets you created in Blender into the SIO2 development environment and add basic interactive functionality.

Chapter 5, "Extending Interactive Feedback with Picking and Text," builds upon the material in Chapter 4, showing you how to add even more sophisticated interactive functionality to your application.

Chapter 6, "Creating an Immersive Environment in SIO2," shows you how to use first-person camera movement and realistic physics to create an immersive 3D world for a player to explore.

Chapter 7, "Props and Physical Objects," builds directly on the project introduced in Chapter 6 by adding a variety of new objects to the scene. Material alpha blending, physics and collisions, and creating billboard objects are all covered here.

Chapter 8, "Animating a Character," turns to the Blender/SIO2 character animation functionality. In this chapter, you learn how to create simple animated actions for a rigged character in Blender and how to activate and control those actions in the SIO2 environment.

Chapter 9, "Working with Widgets," shows you how to use widgets to refine the interface of your app with splash screens and buttons.

Appendix A, "Blender Basics," gives an introduction to basic Blender use, suitable for people who have never used Blender before.

Appendix B, "Key Concepts for Graphics Programming," gives an overview of some key concepts in graphics programming to help deepen your understanding of the book's contents.

Appendix C, "SIO2 Reference," gives information on the official SIO2 tutorials and an overview of the SIO2 file format and functions.

Hardware and Software Considerations

Development for the iPhone platform is fairly restricted. You'll need a Mac running OS X 10.5 (Leopard) or later. You'll also need the iPhone SDK installed, which includes Apple's Xcode integrated development environment (IDE), the iPhone simulator, and other development tools. Getting your hands on the iPhone SDK doesn't cost anything but requires registration with the Apple Developers Connection. However, to make your applications available on the iTunes App Store or to compile your applications onto a physical iPhone or iPod Touch device, you will need to purchase a membership in the iPhone Developers Program, which costs about $100.

Some projects exist for making iPhone development possible on other operating systems. It's doubtful that such efforts will ever be sanctioned by Apple, as welcome as they would be to the developer community at large. The tutorials in this book assume that everything you're doing is carried out on a Mac with the official developer tools and an officially provisioned device. If you have any setup other than this, I wish you the best, but you're on your own.

The Book's Online Project Archive

The projects in this book are available for download in a zip file from the SIO2 website at `http://sio2interactive.com/book/iphoneblendersio2` as well as from this book's Sybex website at `www.sybex.com/go/iphoneblendersio2`. Download and unzip the file, and then put the projects in this file into the main `SIO2_SDK` directory that is part of the official SIO2 package. The official SIO2 package is available for download at `http://sio2interactive.com`.

Contact the Author

You can contact the author at `blender.characters@gmail.com`.

Getting Started with 3D Development for the iPhone

This chapter gives an overview of what you'll need to get started using Blender, SIO2, and Xcode to create interactive 3D content for the iPhone and iPod Touch. It also gives you a heads-up on what you can expect to learn over the course of the rest of the book and tips on where to look for further information on related topics. There's a lot to cover and a few hoops to jump through before you get to the real action, so you'll get right to business by downloading the software you need and setting up your development environment.

- **Getting started**
- **Getting the software**
- **Setting up your SIO2 development environment**

Getting Started

Welcome to the world of interactive 3D graphics programming for the iPhone and iPod Touch using Blender and the SIO2 game engine! I think you'll find that working with these tools is a fun and challenging experience.

Throughout this book, I will assume that you are working on a Mac computer running OS X Leopard (10.5) or later. The official iPhone Software Development Kit (SDK) is designed to run exclusively on Mac. There are some projects underway to create emulators and development environments for doing iPhone development on other platforms, but they are not officially sanctioned by Apple, so if you opt to try to make use of these alternatives, you're on your own. There's no guarantee you're going to be able to install and run your apps on a device or make them available to other iPhone and iPod Touch users.

As mentioned in the introduction, there are a number of areas of background knowledge that will be enormously helpful to you as you work your way through this book. A lot of the necessary information is dealt with in the appendices, but some of it will be up to you to fill in. A good print or online reference for C and C++ syntax will come in handy if you aren't already familiar with these programming languages. As a reference for OpenGL functions, the official *OpenGL Programming Guide (7th Edition)* by Dave Shreiner (Addison Wesley Professional, 2009)—also known as the Red Book—is indispensable.

It is not strictly necessary to have an iPhone or an iPod Touch of your own in order to learn the content of this book. Most (but not all) of the functionality described in this book can be run on your desktop using the iPhone simulator included with the iPhone SDK. However, some functionality, such as the accelerometer, requires the use of an actual device, and if you plan to make your app available to others, it will be necessary to test its performance on an actual device.

Getting the Software

There are a number of software tools you will need to have installed on your computer before you can proceed with this book. Some of what you will need is free and open source, some of it is simply free of charge, and some of it you'll have to pay for (although it won't break the bank). The rest of the chapter will focus on getting what you need and making sure it's working.

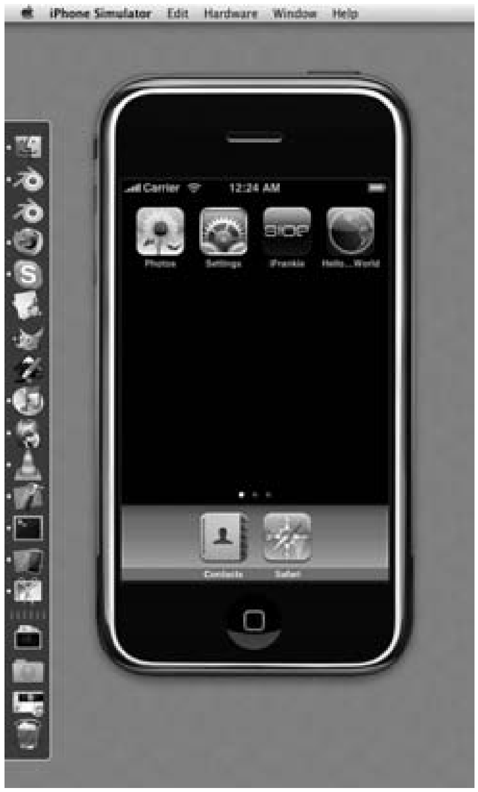
The iPhone SDK

If you're thumbing through this book in the shelves of your local bookstore wondering whether it's right for you, the one thing you should know immediately is that this book (like any book on iPhone development) is for Mac users only. The iPhone SDK 3.0 and development tools are available from Apple for Mac OS X 10.5.7 or greater, and it is not possible to develop for the iPhone on any other platform. If you're running an earlier version of Leopard, you should update your system using the Software Update tool in System Preferences.

The other thing you should be aware of, particularly if you are coming from a background of working with open-source software (as many Blender users are), is that there's nothing open about the iPhone development environment. Apple maintains strict control over how you use the iPhone SDK and how you are able to distribute the products you create with it. This isn't necessarily just because the folks at Apple are control freaks. The era of widely programmable mobile phone handsets has just begun, and there are many open questions about which directions the fledgling industry will take. Apple has erred on the side of caution in terms of security, and the success of the iTunes App Store and many of its contributing developers suggests that Apple is doing something right from a commercial standpoint as well. It remains to be seen how other, more open business models will fare in the arena of programmable handsets.

You can download the iPhone SDK at `http://developer.apple.com/iphone/program/sdk/`. The SDK includes Xcode, Apple's flagship integrated development environment (IDE) that includes a powerful editor and code browser, compilers, and a variety of debugging and testing tools. It also includes the Interface Builder, a separate but tightly integrated development application that enables you to create interfaces in a WYSIWYG manner, and the iPhone simulator, which enables you to test your iPhone software on your own computer via a graphical simulation of the iPhone displayed on your screen, as shown in Figure 1.1.

Figure 1.1
The iPhone simulator



To download all these tools, you will need to register with the Apple Developer Connection (ADC). Basic membership is free of charge, but there are restrictions on what you can do with this level of membership. The most serious restriction on the free membership is that the SDK can compile iPhone software *only* to the simulator. If you want to compile your software for use on an actual physical iPhone or iPod Touch device, you will need to join the iPhone Developer Program, which costs $99. This membership also grants you the right to submit your software for possible inclusion in the iTunes App Store. Membership in the iPhone Developer Program is tightly controlled. Although it is open to anybody to join, there is a period of verification before the certification is issued, and any discrepancies in your application can result in annoying delays while your information is further verified. (Don't mistype your billing address on this one!)

Throughout most of this book, I won't assume that you have iPhone Developer Program membership. The majority of examples in this book can be run on the iPhone simulator. A few features of the hardware are not present in the simulator, such as the accelerometer, which recognizes changes in the angle at which the device is being held. Any places in this book that deal with such functionality will be clearly indicated. If you're new to iPhone development, I recommend that you begin with the simulator. It's free to download, and you can get a good sense of what's involved in programming for the iPhone platform. Once you've decided to get serious, you can spring for the full iPhone Developer Program membership.

Installing the iPhone SDK will install Xcode, the iPhone simulator, Interface Builder, and some other tools on your computer. The installation should be straightforward and self-explanatory. There's a ton of documentation available on the Apple Developer Connection website, and you'll definitely want to delve into it.

Getting Blender

A great thing about mature, user-oriented free software like Blender is the relative ease with which you can download and install it. This book was written to correspond with Blender 2.49, which is the Blender version supported by SIO2 version 1.4.

You can download this version of Blender from the official Blender website. Since the latest Blender version may have changed by the time you read this, please download the software for OS X from the 2.49 archive page at `http://download.blender.org/release/Blender2.49a/`.

Blender should run straight "out of the box." Clicking the Blender application's icon should open a session. If you're new to Blender, now might be a good time to run through Appendix A on the basics of working with it. There are tons of tutorials online as well as a growing number of books available covering a variety of specific topics. Obviously, if you plan to create 3D content in Blender, you're going to want to become as skilled as possible in working with the software.

A Python installation is also required, but you shouldn't have to worry about this because Python 2.5 is installed by default in Leopard.

In OS X, Blender-Python output and errors are displayed in the Console. To read any errors or output from Python scripts, you can run Console (`Applications/Utilities/Console`) before starting up Blender.

Getting SIO2

The centerpiece of this book is the SIO2 engine. SIO2 is a set of software tools for exporting 3D assets from Blender and accessing them from within the Xcode development environment for inclusion in iPhone apps. This book was written to correspond to SIO2 version 1.4. The software is regularly updated and the released version changes regularly, but the version that corresponds to this book (as well as the tutorials used in this book) is available for download at the official SIO2 website: `http://sio2interactive.com/DOWNLOAD.html`.

As mentioned previously, SIO2 is available for free and its use is unrestricted except for one thing: If you use SIO2 to make a game or app available, you are asked to include the SIO2 splash screen at the start of the app. To bypass this restriction and use SIO2 without the splash screen, you are asked to purchase an inexpensive per-game Indie Certificate. The SIO2 Indie Certificate also gives you access to email technical support. SIO2 is not proprietary software, but purchasing the Indie Certificate is a big part of what keeps the project going, so I highly recommend doing so for any serious SIO2 projects. For now, though, simply download the ZIP file in the link and unzip it into a convenient location. I'll refer to this location from now on as your *SIO2_SDK directory*.

Setting Up Your Development Environment

Once you've downloaded the software and followed the steps for installing it, you can test your environment to make sure everything is working. In the following sections, you'll get your first look at the development environment that you'll become very familiar with over the course of the rest of the book. Building SIO2 projects in Xcode should be simple and straightforward, but if you're new to Xcode, there are a few things you might miss. If you hit any snags, skip forward to the troubleshooting section at the end of the chapter.

Building the SIO2 Template in Xcode

When you open your SIO2_SDK directory, you'll see a collection of directories. These include the code for the SIO2 engine, documentation of the API and `.sio2` file format, a collection of tutorials in the form of sample projects, supplementary model and texture data for the tutorial projects, and a template for creating new projects. For the purposes of this book, you'll make very heavy use of the template. In fact, the template project will be the starting point for everything you do with SIO2, so it is a good idea to keep a backup copy of the entire directory. Right now you're not going to make any changes to the template—you're only going to build an executable from it to make sure your development environment is properly set up—so it is not necessary to make a copy.

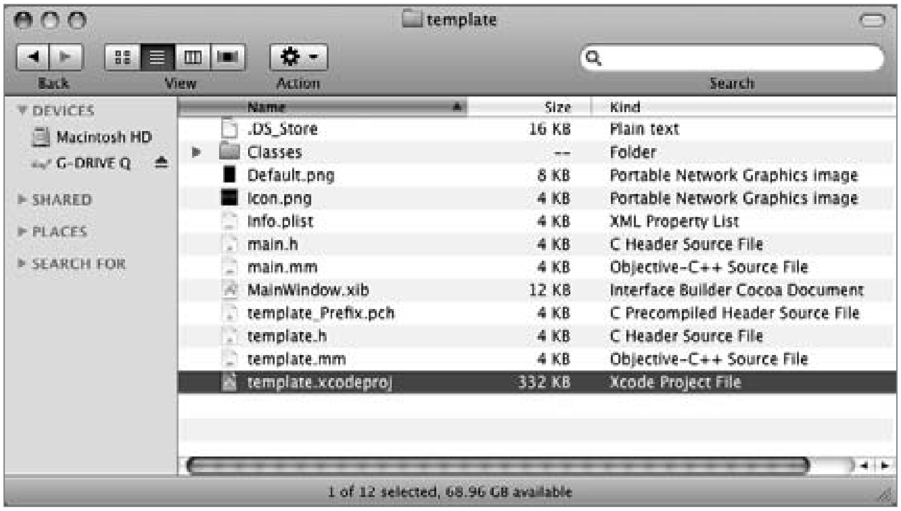
Open the `template` directory and take a look at what's inside. You should see the directory listing shown in Figure 1.2.

Everything that your iPhone app needs resides in this folder. Some of the suffixes are probably familiar to you, but others may not be. Now's not the time to worry about these though. Any code files you need will be dealt with in the Xcode environment. So the only file you really need to bother with here is `template.xcodeproj`. As you might have guessed from the suffix, this file is an Xcode project file. You'll be working a lot with files like these.

Double-click `template.xcodeproj` to open the project in Xcode. The first time you do this, you should see a window something like the one shown in Figure 1.3.

Figure 1.2

The Template project

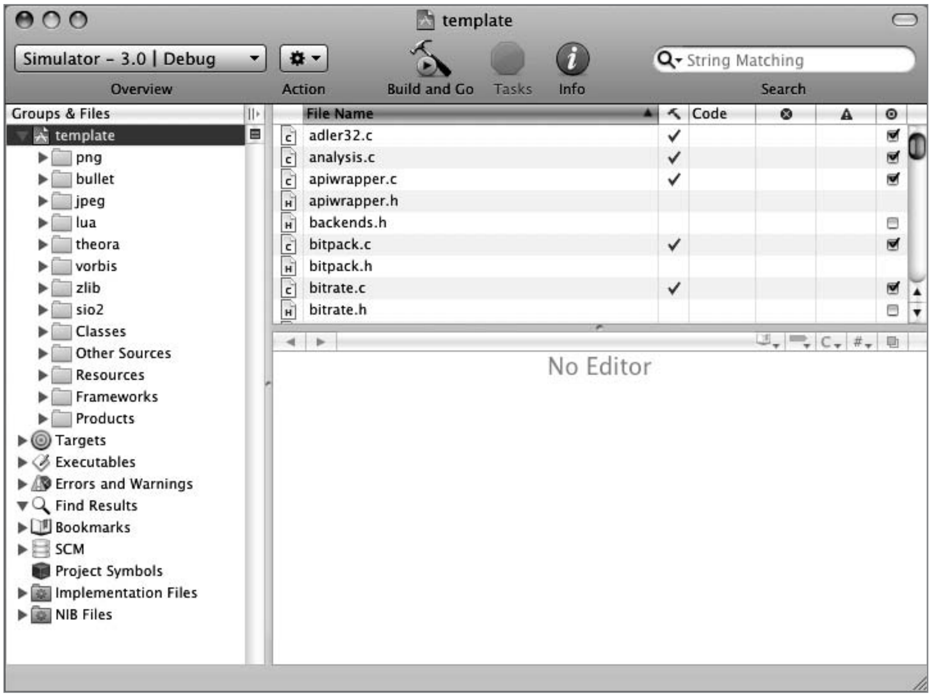


Name	Size	Kind
.DS_Store	16 KB	Plain text
Classes	--	Folder
Default.png	8 KB	Portable Network Graphics image
Icon.png	4 KB	Portable Network Graphics image
Info.plist	4 KB	XML Property List
main.h	4 KB	C Header Source File
main.mm	4 KB	Objective-C++ Source File
MainWindow.xib	12 KB	Interface Builder Cocoa Document
template_Prefix.pch	4 KB	C Precompiled Header Source File
template.h	4 KB	C Header Source File
template.mm	4 KB	Objective-C++ Source File
template.xcodeproj	332 KB	Xcode Project File

1 of 12 selected, 68.96 GB available

Figure 1.3

Opening the Template project in Xcode



File Name		Code			
adler32.c	✓				☑
analysis.c	✓				☑
apiwrapper.c	✓				☑
apiwrapper.h					
backends.h					☐
bitpack.c	✓				☑
bitpack.h					
bitrate.c	✓				☑
bitrate.h					☐

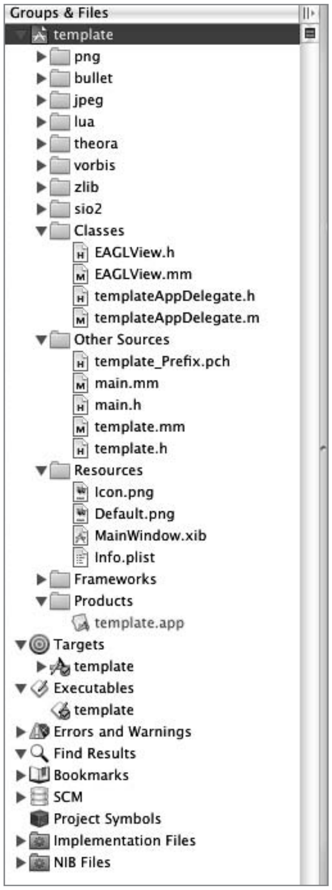
If all has gone smoothly so far, you should now be looking at the Template project in Xcode. This is the integrated development environment (IDE) that you will be working in for all of the coding parts of this book. The main area you see in the lower right (displaying the words *No Editor* in Figure 1.3) is where the code editor will open when a file is selected. Xcode's editor has a lot of powerful features that will help you code more quickly and accurately, and it's worth studying the online documentation available at the ADC website to get fully up to speed with what it has to offer. The window above the editor in the figure gives a listing of files in the project. This window is used for searching and navigating your project quickly.

The drop-down menu in the upper left of the Xcode toolbar is important. This enables you to select the destination for your compiled app. The menu shown in the image is set to Simulator-3.0 | Debug, meaning that the app will be compiled to the iPhone simulator using the 3.0 version of the iPhone OS and with debugging information. Building the application with this setting will automatically start up the iPhone simulator and install and run the app there. If this drop-down menu selection is changed to Device-3.0, Xcode will attempt to install the app on your iPhone or iPod Touch handset. This is possible only if you have registered with the iPhone Developer Program and followed the necessary steps to certify your device.

The tall horizontal pane along the left of the Xcode window is the Groups & Files pane. This gives you a complete overview of everything in your project. Any data or code that your application has access to is listed here. You can click the little triangles to the left of the directory icons to open the directories. Figure 1.4 shows some of the most important files that you'll need to know about as you work your way through this book. Take a close look at those now.

Figure 1.4

The Groups & Files pane



The `Classes` directory lists some standard classes that are typically implemented in iPhone apps. These classes will come up again later in the book, but for now you can regard them as boilerplate code that sets up the viewing environment for the app. You should note, however, that the classes come in pairs of files. Each pair of files includes a header file with a `.h` suffix and a code file with either a `.m` suffix or a `.mm` suffix. The `.m` and `.mm` suffixes indicate Objective-C and Objective-C++ code, respectively. Other source code suffixes you will be likely to see when working with SIO2 and its associated libraries include `.c` for plain C code, `.cpp` for C++ code, and `.cc` for code that can compile as both C and C++.

The `Other Sources` directory includes, as its fiendishly straightforward name suggests, other source code files. The `template_Prefix.pch` file is a precompiled header that you will not need to deal with directly. The `main.h` and `main.mm` files contain the code that makes the top-level function calls for the application. You should take a look at this code, but you will not work much with it directly in this book.

By far, the file that you will work with most in the course of this book will be `template.mm`. This contains most (not all) of the SIO2 API code that accesses the 3D assets and implements the interactive behavior of your app. By the end of this book, you will know this file and files like it inside and out.

The `Resources` directory contains non-code data files that the app needs access to. As you can see, there are two PNG image files currently in the `Resources` directory. One of them is the app icon image, and one of them is the loading screen image.

The `MainWindow.xib` file is created by the Interface Builder application. If you go on to do more iPhone development, you will certainly learn about the Interface Builder and XIB files, but you won't need to deal with them directly for the purposes of this book. All of the official SIO2 tutorials and all of the code in this book are built using the default OpenGL ES template from Xcode, and there is no direct support for building OpenGL ES interfaces with the Interface Builder.

The `Info.plist` file is a property list. This is a table of property values for the app. You can change various things here about your app, such as which image is used for the icon.

The `Resources` directory will also be home to the `.sio2` files created when you export 3D assets from Blender. You'll learn about `.sio2` files in Chapter 3. Note that file extensions are case-sensitive in Mac, and the `.sio2` file extension is always formatted in lowercase.

The `Products`, `Targets`, and `Executables` directories hold the elements of your app. Mostly, you will not need to deal with these directly, except to change their names when creating your own project. The Target system in Xcode enables multiple related applications to be created within a single project, which is useful in large-scale development projects such as client-server applications. For iPhone development, however, it is unlikely you will ever need to deal with more than one target per project, so this functionality can be mostly ignored.

All those directories that I haven't mentioned in the upper half of the Groups & Files pane are important too, but you will mostly not need to access them directly. These are the libraries that provide the functions called in the application. The `sio2` directory contains the actual SIO2 code that implements the functionality you'll be using. The `bullet` directory contains the Bullet Physics Library. To learn about the implementation of these libraries, you can browse these directories.

Now that you've got some idea of what's in your project, you can go ahead and advance to the most anticlimactic part of this whole chapter: building the template app. Do this by clicking the Build And Go button at the top of the Xcode window. Wait a few seconds as Xcode builds the app and installs it on your iPhone simulator. The iPhone simulator should open automatically, the SIO2 loading screen should flash for a split second, and then, if everything has gone smoothly. . .nothing! The screen of your iPhone simulator should go completely black.

Of course, the reason nothing happened is that this is, after all, a template. The whole point of this book will be to teach you how to turn this nothing into something interesting. To stop the current app, click the round button at the base of the iPhone simulator, just as you normally would on your iPhone or iPod Touch to stop an app. You'll return to the buttons screen of the iPhone simulator, as shown in Figure 1.5, where you'll see, sure enough, the button for the template app alongside the other apps installed on the simulator.

If this has all gone as described, then you should be pleased. Your development environment is set up and SIO2 is building smoothly. You're ready to move on to creating actual 3D content for your iPhone or iPod Touch in Chapter 2.

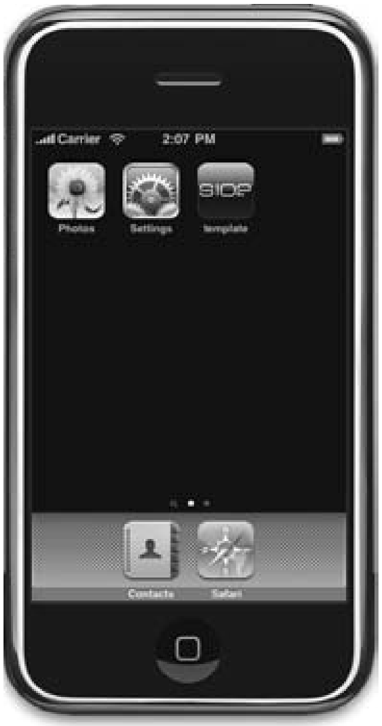


Figure 1.5
The template app installed in your iPhone simulator

Nevertheless, I wouldn't blame you if you felt a little bit gypped after going through a whole chapter without getting to see any actual 3D action on your iPhone simulator. Fortunately, the `SIO2_SDK` directory is packed with ready-made code samples that you can dive into and explore right now. I highly recommend that you take a look at some of them. You can build and run them all in exactly the same way that you did the template, by opening the file with the filename extension `.xcodeproj` in the project's directory and clicking Build And Go. Figure 1.6 shows the results of the `tutorial02` project, featuring every Blender artist's favorite digital monkey. Have fun exploring the other tutorial files. Don't worry if they seem over your head. After you have made your way through this book, you will have the background you need to dive in and pick them apart. Chapter 9 gives an overview of their contents, so you can go straight to the tutorial that has the advanced information you need.

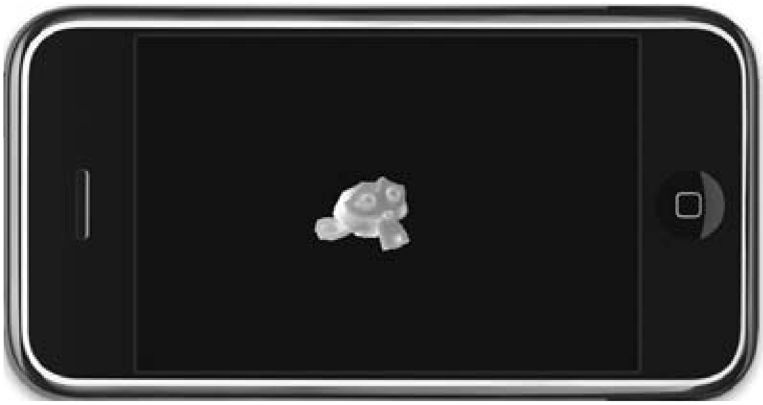


Figure 1.6
Suzanne in your iPhone

Troubleshooting

If Xcode does not open when you double-click a file with the filename extension `.xcodeproj`, it means there is a problem with your Xcode installation. You will need to go back to the instructions at the Apple Developer Connection website and make sure you correctly downloaded and installed the iPhone SDK.

The first time you build an app, it is important to have the build destination and the SDK version set correctly. Make sure the drop-down menu in the upper-left corner of the Xcode window is set to the version of the SDK you are using. Furthermore, make sure the application itself is set to compile using the correct version of the SDK. You can check this in the project settings under Project → Edit Project Settings. If you have trouble, refer to the iPhone Reference Library at `http://developer.apple.com/iphone/library/navigation/index.html` and search for "running applications" for more details about how to set the Project Build settings correctly.

If you are enrolled in the iPhone Developer Program and are compiling to a device, be sure you have read all the relevant documentation on certifying and making your device available to Xcode. If you have done this and have trouble compiling some of the tutorials, it may be due to discrepancies in code signing. If the apps are code-signed to another developer, you will need to change the code-signing value to build them yourself. Code-signing information for a project is found in the Project Settings properties list, and code-signing information for the target application is found in the Active Target properties list, which can be accessed under Project → Edit Active Target *`target_name`* (where *`target_name`* is the name of your target). Code-signing information must be set correctly for both the project and the target. You can find more information about this on the same "Running Applications" iPhone Reference Library page mentioned previously.

You will get a lot of use out of the iPhone Reference Library if you continue with iPhone and iPod Touch programming, so it's a good idea to bookmark it. If you've read this chapter and the pertinent iPhone SDK documents and iPhone Reference Library resources and you're still having problems, go to the SIO2 forum at `http://forum.sio2interactive.com` and run a search on your problem.

In the next chapter, you'll learn more about the fundamentals of OpenGL ES graphics programming in the iPhone.

Introducing Graphics Programming in SIO2

In this chapter, you'll get your first taste of programming graphical content for your iPhone or iPod Touch screen using SIO2. This chapter will also give you an introduction to some of the OpenGL ES graphics programming concepts that you will encounter throughout the book. OpenGL ES is the mobile specification of OpenGL and provides the foundational 3D programming functionality upon which SIO2 is built. SIO2 code works hand in hand with OpenGL ES code to create an exceptionally flexible game programming environment. In this chapter, you'll see how OpenGL ES calls integrate seamlessly with higher-level SIO2 functions. You'll also get a closer look at how easy it is to work with input from the iPhone's sophisticated hardware interface. You won't be working with 3D content quite yet. The fancy 3D stuff will come in the next chapter when you fire up Blender to create your first 3D scene.

- **The SIO2 template**
- **A simple OpenGL demo**
- **Introduction to interactivity**
- **The Complete Code**

The SIO2 Template

The SIO2 engine provides a powerful, high-level programming environment for interactive 3D development on the iPhone. A lot of that power comes from the fact that SIO2 brings together a variety of different libraries and technologies. One of the most important technologies you will be using is the OpenGL ES application programming interface (API) for graphics programming. OpenGL ES is a widely used variant of OpenGL specified for mobile embedded systems such as the iPhone and iPod Touch as well as numerous other mobile platforms. OpenGL ES plays a big role in this chapter and in all subsequent chapters. If you're already conversant with OpenGL, then you will mainly be interested in seeing how its functions are used in the context of the SIO2 environment. If you don't have any experience with OpenGL or graphics programming, then now would be a good time to read through Appendix B, which is an overview of some key concepts that will help you better understand what's going on in the code.

Setting up the SIO2 environment to work within the iPhone SDK framework and to make use of all the necessary tools is not trivial. Fortunately, you don't have to. As you saw in Chapter 1, in addition to a wide array of in-depth, heavily annotated example projects, the freely downloadable SIO2 package comes with a ready-made Xcode template project for you to begin work with right away. All of the SIO2 code in this book takes this template as its starting point.

In Chapter 1, you got an overview of the files in the Template project. Here, you'll take a closer look at the contents of the main files you'll be working with throughout this book. In this chapter, you will be making changes to the code to create some simple graphical content, so you should first make a copy of the entire `template` directory. Do this by right-clicking the directory in the Finder and selecting Duplicate. Rename the new directory project1. Don't move this directory anywhere though. It needs to stay in your SIO2_SDK directory so that the various libraries and resources it uses are accessible. Once you have created the new project directory, go into the directory and open the project in Xcode by double-clicking on the `template.xcodeproj` file.

Although there are several files in the template project, the one that you will spend the most time working with is `template.mm`. This can be found in the `Other Sources` directory in the Groups & Files area of the Xcode interface. Navigate to the file and click on it to bring it up in the Xcode text editor. The rest of this section is devoted to a line-by-line description of the `template.mm` file.

The first few lines contain boilerplate descriptive comments and copyright information:

```
/*
 *  template.mm
 *  template
 *
 *  Created by SIO2 Interactive on 8/22/08.
 *  Copyright 2008 SIO2 Interactive. All rights reserved.
 *
 */
```

Objective-C enables comments to be written in the C style: Multiline comments are delineated by a preceding `/*` and a following `*/`. Anything between these strings is a comment and is ignored by the compiler. Single-line comments can be written using `//` at the beginning of the line.

The next two lines of code ensure that the necessary header files are included:

```
#include "template.h"
#include "../src/sio2/sio2.h"
```

The convention for iPhone development is to have functions' prototypes placed in a header file and their definitions written in a source code file with the same name as the header.

Prototypes tell the compiler about functions that will be defined elsewhere, making it possible to organize your code in a sensible way without causing trouble for the compiler. You don't need to worry too much about that for the purposes of this book as long as you know where things are located.

The `template.h` file contains the prototypes for all the functions defined in the `template.mm` file. You will need to edit that file only if you add new function definitions to `template.mm`. The `../src/sio2/sio2.h` file is the initialization header for the entire SIO2 engine. This file contains `include` directives for all the other SIO2 header files and for those of other necessary directories. In addition to this, the `sio2.h` file defines several important constants, contains prototypes for functions, and defines the structure of the `sio2` object that you will access often when you use the SIO2 engine. There is a single `sio2` object created for an application, and this object acts as a container for all the data you will need to access while the application is running. You'll begin to see the `sio2` object in action later in this chapter. In short, the `sio2.h` file contains the guts of the whole SIO2 engine.

The next chunk of code is where most of the action will occur in your programs. This is the `templateRender` function:

```
void templateRender( void )
```

This function loops as the game progresses, rendering the screen fresh for each frame. Because this function executes repeatedly in rapid succession, it is important to keep it as lean as possible. Be sure not to call initialization functions here, or anything else that can be done just once. This function is only for code that needs to be executed anew for each frame of the game.

The default contents of `templateRender` are minimal, just a few standard OpenGL calls. OpenGL functions can be called directly in the code. The first function called is `glMatrixMode( GL_MODELVIEW )`, which sets the current matrix mode to `GL_MODELVIEW` (see Appendix B to find out what this means if you're not sure). The next function called is `glLoadIdentity()`, which sets the active transformation matrix to the identity transformation (also covered in Appendix B). Finally, the `glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT )` function is called, which clears the depth buffer and the color buffer. It is usually necessary to clear the content of the previous frame before rendering the next frame so that the effect is animation rather than simply a stack of images rendered one on top of the next. This is not always necessary though. For example, when a background or sky box is used so that the screen is redrawn in its entirety each frame, you don't need to clear the color buffer. As with anything, if it's not necessary, you shouldn't do it because it requires time and resources.

The color buffer contains information about the color of each pixel and the depth buffer contains information that is used to calculate which elements should be rendered in front of or behind other elements. This information is used to determine which pixel-sized image fragments are ultimately rendered. As you can see from the comment on the next line, the rendering code specific to your app will generally follow these lines:

```
{
     glMatrixMode( GL_MODELVIEW );
     glLoadIdentity();
     glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );
     // Your rendering code here...
}
```

The next function is the `templateShutdown` function. This is an important function, but it's not something you will usually need to monkey with. This function frees the resources that SIO2 has been using and shuts down the engine:

```
void templateShutdown( void )
{
     // Clean up
     sio2ResourceUnloadAll( sio2->_SIO2resource );
     sio2->_SIO2resource = sio2ResourceFree( sio2->_SIO2resource );
     sio2->_SIO2window = sio2WindowFree( sio2->_SIO2window );
     sio2 = sio2Shutdown();
     printf("\nSIO2: shutdown...\n" );
}
```

The last three functions deal with the interactive functionality of the iPhone/iPod Touch. They are `templateScreenTap`, `templateScreenTouchMove`, and `templateScreenAccelerometer`. Their names are indicative of what they handle, namely screen taps (the state argument distinguishes between the tap down event and the tap up event), moving touches on the screen, and tilting or shaking of the device itself, as sensed by the built-in accelerometer. No functionality is defined for any of these functions:

```
void templateScreenTap( void *_ptr, unsigned char _state )
{
}

void templateScreenTouchMove( void *_ptr )
{
}

void templateScreenAccelerometer( void *_ptr )
{
}
```

That's it. The template is a syntactically correct, complete app that does nothing at all. The necessary functions to render frames and to handle interaction are all defined. In the next sections you'll see how to make changes to this template to make something happen on the screen.

A Simple OpenGL Demo

In the following sections, you'll look more closely at how to run OpenGL code in SIO2. The code introduced here is based on the SIO2 Tutorial 1, which in turn is based upon the iPhone OpenGL ES demo provided by Apple. If you are knowledgeable in OpenGL, this will all be very straightforward. If not, pay close attention and don't be afraid to experiment with variations on the code to get a firsthand feel for how things work. The material in Appendix B is also pertinent here. If you're new to graphics programming, you should definitely read Appendix B. If you're inexperienced with C, you should be prepared to look up some terms online or in a C reference book. These sections and much that follows assume that you are familiar with some common basic concepts in programming, such as what data types are. If you aren't sure what things like floating-point numbers or unsigned chars are, be prepared to spend some time with Google.

Creating Graphical Content in OpenGL ES

The first thing to do is to create a graphical object. Ordinarily, SIO2 will do this automatically, by reading data exported from Blender, but in this example you will create an object by hand by defining a vertex array. The object you'll create is a simple square. The array that will represent its vertices is created like this:

```
const GLfloat squareVertices[] = {
     -100.0f, -100.0f,
     100.0f,  -100.0f,
     -100.0f,  100.0f,
     100.0f,   100.0f,

};
```

This code should be the first few lines in the `templateRender` function, so insert it between lines 16 and 17 in the original template code. The elements of the array represent the *x* and *y* values for each vertex in the square, resulting in a 200×200-pixel square. The name of the array is `squareVertices`, and later you will be able to render the square by creating an OpenGL vertex array using this data. The elements are of type `GLfloat`, which is the OpenGL implementation of floating-point numbers.

You'll color the square vertex by vertex using a color array. In a color array, the first four elements represent the color values for the first vertex, the next four elements represent the color values for the second vertex, and so on, with four array elements for each vertex. Colors on the iPhone are always expressed with four values: R, G, B, and A. Some graphical programming environments also allow colors to be represented with three values, dropping the alpha value, but not the iPhone. The elements represent the red, green, blue, and alpha values. When using unsigned char data types, as in this example, the range of values for each color channel is from 0 to 255. The fourth value, alpha, is used to represent opacity when blending is enabled, but in this example the alpha value is not used. Take a look at the following code and see if you can predict what color each vertex of the square will be. Include this code right after the previous code with the vertex array:

```
const unsigned char squareColors[] = {
     255, 255,   0, 255,
     0,   255, 255, 255,
     0,     0,   0,   0,
     255,   0, 255, 255,
};
```

You've now defined the data for use in a color array and stored it in a variable called *`squareColors`*.

The next chunk of code is for rendering. It should follow the OpenGL lines in the original template, starting at the point where the `your rendering code here` comment is. It will extend to the end of the `templateRender` function. To keep it simple for now, this example is not in 3D, so you'll use SIO2's 2D rendering mode. You'll need to enter 2D mode first and then leave it later. Use the following functions to do that:

```
sio2WindowEnter2D( sio2->_SIO2window, 0.0f, 1.0f );
{
//insert the following code here
}
sio2WindowLeave2D();
```

The arguments to `sio2WindowEnter2D` represent the `SIO2window` object itself and the depth of the ortho projection in GL units.

Between the functions to enter and leave 2D mode, curly brackets delineate a block of code with the comment `insert the following code here`. That's what you'll do. The remaining code described in this section will be inserted between those curly brackets, and will specify how to render the square.

The first line creates the actual vertex array:

```
glVertexPointer( 2, GL_FLOAT, 0, squareVertices );
```

The first argument tells OpenGL how many coordinates are to be used for each vertex. The example is a 2D square with only x- and y-coordinates defined, so the value is `2` in this case. For 3D data, a value of `3` would be appropriate. A value of `4` is also possible, which enables access to the fourth coordinate in the homogeneous coordinate system (see Appendix B for more information on this). The next argument tells OpenGL what type to expect from elements of the array. The array was made up of floating-point numbers, so `GL_FLOAT` is the appropriate label. The next argument is the *stride* of the vertex array, which determines whether the vertex array's elements are densely packed or interleaved. In this example, the stride is `0`, so there is no interleaving. Finally, the array of floating-point numbers previously defined is passed to the function to supply the actual data. The next line of code is required to tell OpenGL to use the vertex array when it comes time to render:

```
glEnableClientState( GL_VERTEX_ARRAY );
```

The next two lines set up and enable the color array:

```
glColorPointer( 4, GL_UNSIGNED_BYTE, 0, squareColors );
glEnableClientState( GL_COLOR_ARRAY );
```

The arguments for `glColorPointer` are analogous to those of `glVertexPointer`. Notice that the size value is `4` because the color values for each vertex are described using four values. You might wonder why four values are used in this example despite the fact that the alpha values are not being used. Indeed, this is an example of a place where your options are somewhat more limited in OpenGL ES than in standard OpenGL. Although you have some options in OpenGL, in OpenGL ES this value is required to be `4`. Don't worry too much about these GL function calls though. When you get to using SIO2, you will generally not have a need to work with these functions directly because they will all be called behind the scenes by SIO2.

In the next few lines of code, the `sio2` object finally makes its appearance!

```
glTranslatef( sio2->_SIO2window->scl->x * 0.5f,
              sio2->_SIO2window->scl->y * 0.5f, 0.0f );
```

In this example, it's being used to place the square at the center of the screen with the `glTranslatef` function. The `glTranslatef` function adds a translate transformation to the `modelview` matrix, determining where the center will be when the scene is drawn. You want this to be in the middle of the screen, so it is necessary to translate the center point half the screen width along the x-axis (from left to right) and half the screen height along the y-axis (from top to bottom). No translation is necessary along the z-axis because this is all taking place in 2D. The way these dimensions are accessed should give you a sense of some of the things that have been initialized automatically with the `sio2` object. The `sio2` object includes a `_SIO2window` object as one of its properties, which in turn has `scl` (scale) values for the x and y axes. The x scale is accessed with `sio2->_SIO2window->scl->x` and the y scale in a corresponding manner. To get half the distance across and down the screen, both values are multiplied by 0.5.

To get a sense of the looping nature of the `templateRender` function, you'll incorporate animation into this example also. A simple example of animation is to rotate the square by a small increment each frame. There are several ways to accomplish this effect, but in this example you will once again turn to the `sio2` object and its `_SIO2window`, which has a property called `d_time` that returns the amount of time that has passed since the application began running, based on an internal chronometer. The next line declares the variable *rotz* and initializes it to `0.0`:

```
static float rotz = 0.0f;
```

The next line of code rotates the transformation matrix:

```
glRotatef( rotz, 0.0f, 0.0f, 1.0f );
```

The `glRotatef` function's first argument is the rotation value, and the next three arguments represent how much influence the rotation value has on each of the three axes. In this case, x- and y-axes are unaffected while z is rotated the full amount of the rotation value. The next line sets the *rotz* value by multiplying the `d_time` value (the change in the chronometer value from the previous frame) by 90.0, so that the *rotz* value increases with each frame:

```
rotz += 90.0f * sio2->_SIO2window->d_time;
```

The static identifier before the declaration of *rotz* ensures that the initialization to 0.0 occurs only once and the value is retained over repeated calls of the function.

The last line renders the scene using the currently enabled arrays with the `glDrawArrays` function. The first argument, `GL_TRIANGLE_STRIP`, tells OpenGL which drawing mode to use:

```
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

With OpenGL ES, the options are limited to drawing modes that work with points, lines, and triangles. Other polygons must be constructed out of triangles. The second argument tells OpenGL which index of the arrays to start with, and the third argument determines how many vertices will be drawn. Using triangle strips is a faster way to render triangles, but there are some cases in which they shouldn't be used. They don't work well when physics simulation is used on an object, which you'll read more about in Chapter 6.

You're ready to build and run your application now. Follow the same steps you followed in Chapter 1 to build and run the example in the section "Building the SIO2

Template in Xcode." You should see a multicolored square appear and rotate counterclockwise on your screen as shown in Figure 2.1.

Looking Closer at Transformations

In this section, you'll go through a few further examples that will help you form a fuller sense of how transformations work. If you are already fluent in OpenGL, you can skim this part. If you're just getting the hang of it though, you'll want to follow this section closely because OpenGL transformations play an important role in programming with SIO2.

To see some more transformations in action, you'll create another square. It's not necessary to use a different vertex array—the one you already created is fine. However, you'll use different colors for the second square. To do this, create another array called `primaryColors`. The 1st vertex will be red, the 2nd vertex blue, the 3rd green, and the 4th white. Remember, the 4th, 8th, 12th, and 16th elements here are basically going to be ignored because alpha blending is not used. Add the following code just after the place where the previous vertex and color data arrays were defined:

```
const unsigned char primaryColors[] = {
     255,   0,    0,    0,
     0,   255,    0,    0,
     0,     0,  255,    0,
     255,  255, 255,   0,
};
```

This square will be a small square that orbits the larger square in a clockwise direction. Because the direction of its rotation is opposite that of the first square, you need to rotate the matrix –2 times the value of *rotz* in order. The next line adds a translation along the x-axis, which will offset the square away from the center of the screen. The next line adds a scaling transformation to reduce the square's size. The last two lines render the square just as before, except using the new color data for the color array:

```
glRotatef(-2.0f*rotz, 0.0f, 0.0f, 1.0f);
glTranslatef(125.0f, 0.0f, 0.0f);
glScalef(0.2f, 0.2f, 0.2f);
glColorPointer(4, GL_UNSIGNED_BYTE, 0, primaryColors);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

If you build and run now you'll see what was just described: a large square with yellow, cyan, maroon, and black corners rotating counterclockwise and the primary-colored smaller square orbiting it clockwise, as shown in Figure 2.2.

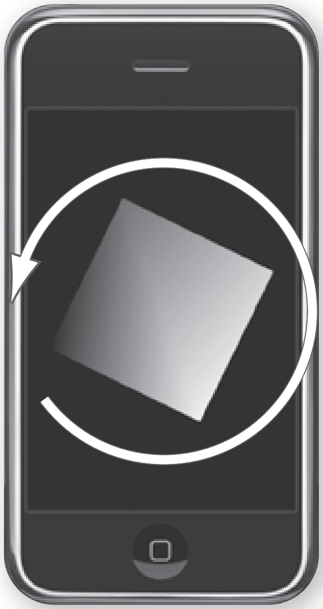


Figure 2.1
A rotating square created in OpenGL

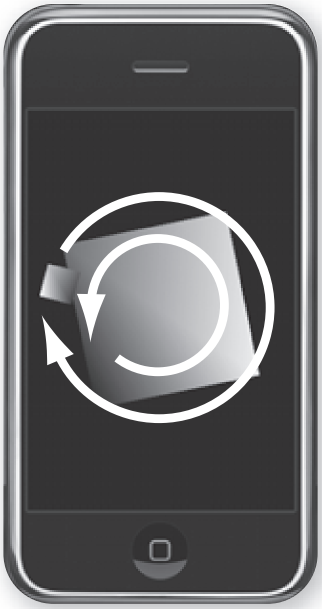


Figure 2.2
A small square orbiting the large square

Transformation Order

The order in which matrix transformations are performed makes a difference. This is due to the fact that matrix multiplication is not commutative (AB≠BA), as described in more detail in Appendix B. You can see that here by reversing the order of the translate and rotate operations. In the preceding code, reverse the order of these lines:

```
glRotatef(-2*rotz, 0.0f, 0.0f, 1.0f);
glTranslatef(125.0f, 0.0f, 0.0f);
```

They then appear like this:

```
glTranslatef(125.0f, 0.0f, 0.0f);
glRotatef(-2*rotz, 0.0f, 0.0f, 1.0f);
```

When you build and run the project, you'll see that the effect has changed. The small square now follows the large square, clinging to one side while rotating around its own center, as shown in Figure 2.3. It shouldn't be too hard to grasp why this is. In the first case, the rotation happens first, so the axis along which the translation happens has already changed direction. In the second case, the translation of the small square occurs before the clockwise rotation (but of course after the original counterclockwise rotation) so the translation is completed first, after which the rotation happens around the new center point.

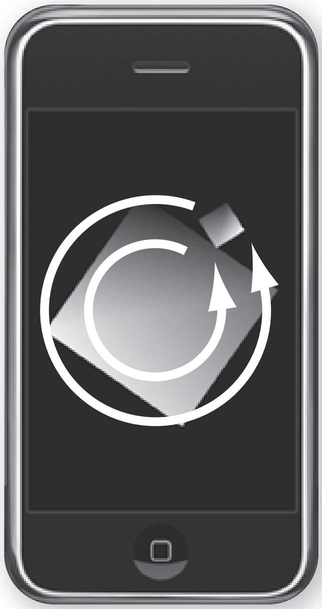


Figure 2.3
The small square's location follows the large square's rotation.

Using the Matrix Stack

Now, what if you want to render both of those small squares simultaneously, one rotating counterclockwise with the large square and the other rotating clockwise in the opposite direction around the large square? You might be tempted to simply write the code for rendering both squares in order, like this:

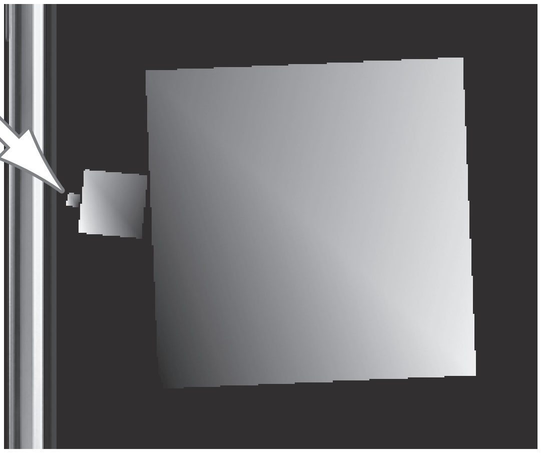
```
glRotatef(-2*rotz, 0.0f, 0.0f, 1.0f);
glTranslatef(125.0f, 0.0f, 0.0f);
glScalef(0.2f, 0.2f, 0.2f);
glColorPointer(4, GL_UNSIGNED_BYTE, 0, primaryColors);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

glTranslatef(125.0f, 0.0f, 0.0f);
glRotatef(-2*rotz, 0.0f, 0.0f, 1.0f);
glScalef(0.2f, 0.2f, 0.2f);
glColorPointer(4, GL_UNSIGNED_BYTE, 0, primaryColors);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

But if you do this, you'll see that it does not yield the results you're looking for. If you use this code, you will find that the second small square is smaller still than the first one, tiny in fact, and it is riding piggyback on the first small square rather than rotating around the large square, as shown in Figure 2.4. The reason for this is clear enough when you recall that all of the transformation operations on the model/view matrix are cumulative (if you're not sure what the model/view matrix is, please refer to Appendix B). You've already scaled down once, so scaling again will make the next square even smaller. Likewise, the rotations and translations acting on the third square are all with respect to where the matrix was when the second square was rendered. Clearly something else is needed.

The solution to this is to use the *matrix stack*. The matrix stack enables you to set a point in the sequence of transformations, add new transformations, and then jump back to the previous point. This is done by *pushing* the matrix onto a stack, adding transformations, doing what you want to do with them, and then *popping* the matrix stack to return to the previous state of transformations. The data structure of a stack is something like a PEZ candy dispenser if you were to load it piece by piece through the character's mouth—as you push more objects onto the stack, the first objects you pushed on are pushed further and further down the stack. To take objects from the stack, you pop them from the top of the stack (like a PEZ dispenser used in the ordinary way). Objects are popped from the stack in first-in/last-out order.

Figure 2.4

A tiny square riding on the small square



In this case, you will want to push the matrix onto the stack before transforming and drawing the first small square, then pop the matrix to return to the transformations of the larger square, and then do the same thing for the second small square, like this:

```
glPushMatrix();
     glRotatef(-2*rotz, 0.0f, 0.0f, 1.0f);
     glTranslatef(125.0f, 0.0f, 0.0f);
     glScalef(0.2f, 0.2f, 0.2f);
     glColorPointer(4, GL_UNSIGNED_BYTE, 0, primaryColors);
     glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glPopMatrix();

glPushMatrix();
     glTranslatef(125.0f, 0.0f, 0.0f);
     glRotatef(-2*rotz, 0.0f, 0.0f, 1.0f);
     glScalef(0.2f, 0.2f, 0.2f);
     glColorPointer(4, GL_UNSIGNED_BYTE, 0, primaryColors);
     glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glPopMatrix();
```

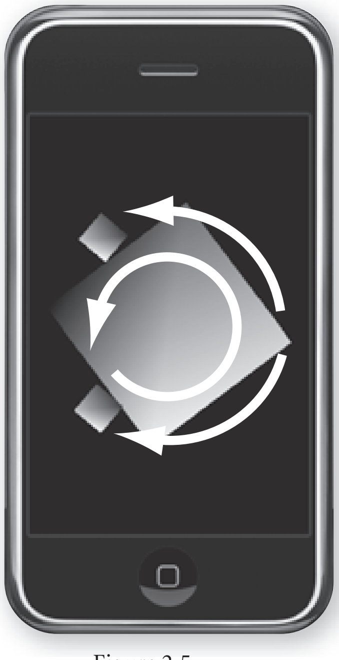


Figure 2.5
The three squares rotating as expected

When you build and run the code now, it will behave as expected, with one small square rotating clockwise around the big square and the other following the counterclockwise rotation of the big square and rotating clockwise around its own center, as shown in Figure 2.5. Experiment with changing these transformations on your own. Can you set it up so that a tiny square orbits the small square? What happens if you call the `glLoadIdentity` function inside the stack?

Introduction to Interactivity

In the final part of this chapter, you'll see how simple it is to work with the iPhone's touch screen and accelerometer interfaces with the SIO2 engine. User interactivity with the iPhone and iPod Touch comes from touching the multitouch screen with one or more fingers and by accessing the accelerator by moving or shaking the device. In this section, you'll look at callback functions for the whole screen, but SIO2 also enables you to have local callbacks on components or part of the screen, so it is not necessary to always put all your code in these functions. You'll read more about local callbacks in Chapter 9 during the discussion of widgets.

There are two basic types of touch input: screen taps and screen touch moves. The first type, taps, can be in one of two states, tap down and tap up. The tap-down event is triggered when a finger initially makes contact with the screen. The tap-up event is triggered when a finger that is touching the screen separates from the surface. Therefore, quickly tapping the screen once will result in two tap events: a tap-down event followed by a tap-up event. Screen-touch move events occur when a finger touching the screen changes location without leaving the screen. All touch move events are preceded by a tap-down event and followed by a tap-up event.

You should recall from the first section of this chapter that there are already functions defined to handle these events. Here, you'll add some code to those functions to access and print out the information they return. First, write the `templateScreenTap` function as follows:

```
void templateScreenTap(void *_ptr, unsigned char _state)
{
     if( sio2->_SIO2window->n_touch )
          {
               printf("templateScreenTap >> state:%d tap:%d x:%f y:%f\n", _
state,
                       sio2->_SIO2window->n_tap,
                       sio2->_SIO2window->touch[0]->x,
                       sio2->_SIO2window->touch[0]->y );
          }
}
```

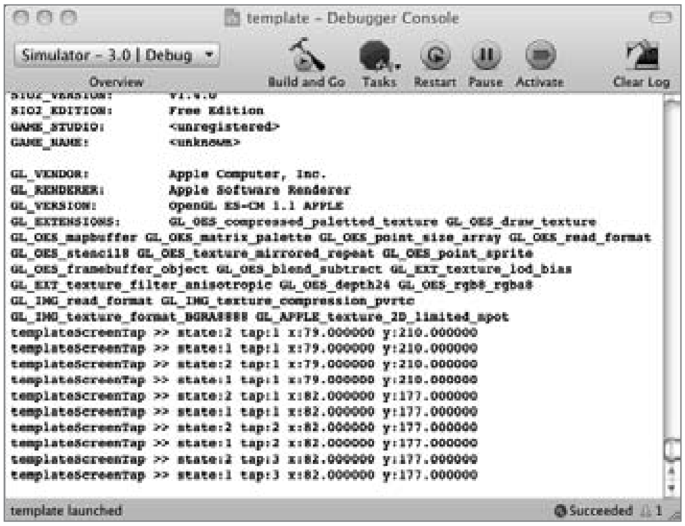
The first line here is simply a conditional to ensure that the screen has been touched in at least one place for the print statement to be executed. Note that once again you are accessing data held by the `sio2` object, and specifically its `_SIO2window` property. The value being accessed is `n_touch`, which holds the number of simultaneous touch points at any given time. There's no point trying to print out touch data if the screen hasn't been touched.

Inside the conditional block is the `printf` function, which is the C command for printing formatted data to the standard output. If what's between the quotes there looks cryptic to you, do a Google search for "printf" to find a primer on what the percent symbols mean. For the moment, suffice it to say that the quoted string in the first argument of `printf` will include the data from the second, third, fourth and fifth arguments, inserted into the appropriate points in the string. So what are the other arguments? The second argument is the `_state` value, which will be either `2` (tap down) or `1` (tap up). The third argument is the `n_tap` value, which tells how many times the screen has been tapped in rapid succession. This is often used to register simple double taps, but it can actually count an arbitrarily high number of quickly struck taps. A pause between taps will reset this value to `0`. The fourth and fifth arguments of `printf` are the `x` and `y` values, respectively, of the first touch point. This simple code doesn't exploit the full multi-touch capacity of the iPhone. You access data from subsequent multi-touch points by the index on the `touch` array: `0` is the first finger to touch the screen, `1` is the second finger to touch the screen simultaneously, `2` is the third, and so on.

So this code will print this data to the standard output. But where is the standard output? That's an important thing to know about when developing anything, and in this case, the standard output is written to Xcode's console. To see the console, select Console from the Run menu. Build and run your project. If you are building on the simulator, you can test the function by clicking your mouse on the simulator's screen. The console should register the touches, as shown in Figure 2.6. If you're building to your device, then touching the screen will have the same effect. Experiment with touching different places on the screen and note what the corresponding `x` and `y` values are. Try tapping quickly and seeing how the tap count changes. Tap down and hold for a while before lifting your finger to see the difference between tap-down and tap-up events.

Figure 2.6

Monitoring touches in the console



```
SIO2_VERSION:          v1.4.0
SIO2_EDITION:          Free Edition
GAME_STUDIO:           <unregistered>
GAME_NAME:             <unknown>

GL_VENDOR:             Apple Computer, Inc.
GL_RENDERER:           Apple Software Renderer
GL_VERSION:            OpenGL ES-CM 1.1 APPLE
GL_EXTENSIONS:         GL_OES_compressed_paletted_texture GL_OES_draw_texture
GL_OES_mapbuffer GL_OES_matrix_palette GL_OES_point_size_array GL_OES_read_format
GL_OES_stencil8 GL_OES_texture_mirrored_repeat GL_OES_point_sprite
GL_OES_framebuffer_object GL_OES_blend_subtract GL_EXT_texture_lod_bias
GL_EXT_texture_filter_anisotropic GL_OES_depth24 GL_OES_rgb8_rgba8
GL_IMG_read_format GL_IMG_texture_compression_pvrtc
GL_IMG_texture_format_BGRA8888 GL_APPLE_texture_2D_limited_npot
templateScreenTap >> state:2 tap:1 x:79.000000 y:210.000000
templateScreenTap >> state:1 tap:1 x:79.000000 y:210.000000
templateScreenTap >> state:2 tap:1 x:79.000000 y:210.000000
templateScreenTap >> state:1 tap:1 x:79.000000 y:210.000000
templateScreenTap >> state:2 tap:1 x:82.000000 y:177.000000
templateScreenTap >> state:1 tap:1 x:82.000000 y:177.000000
templateScreenTap >> state:2 tap:2 x:82.000000 y:177.000000
templateScreenTap >> state:1 tap:2 x:82.000000 y:177.000000
templateScreenTap >> state:2 tap:3 x:82.000000 y:177.000000
templateScreenTap >> state:1 tap:3 x:82.000000 y:177.000000
```

Implement the `templateScreenTouchMove` function as shown here:

```
void templateScreenTouchMove(void *_ptr)
{
     if( sio2->_SIO2window->n_touch )
          {
               printf("templateScreenTouchMove >> x:%f y:%f\n",
                       sio2->_SIO2window->touch[0]->x,
                       sio2->_SIO2window->touch[0]->y );
          }
}
```

This is simpler than the tap function because there's less data to retrieve. There are no states and nothing is counted. The function simply prints the changing `x` and `y` values. You can build and run again to see how this works when you drag your finger around the screen.

The last function can be used only if you have a provisioned device and are a registered member of the iPhone Developer Program because the simulator is not able to work with accelerometer data. The accelerometer senses the position in space of the device, and the 2D interface of the desktop iPhone simulator isn't well suited to mimicking it.

The code here won't cause any problems if you run it on the simulator—it just won't do anything. If you do have a registered device to use as a build destination, then you can access accelerometer data by implementing the `templateScreenAccelerometer` function as shown here:

```
void templateScreenAccelerometer(void *_ptr)
{
     printf("templateScreenAccelerometer >> x:%f y:%f z:%f\n",
             sio2->_SIO2window->touch[0]->x,
             sio2->_SIO2window->touch[0]->y,
             sio2->_SIO2window->touch[0]->z);
}
```

Build and run, and move your device around in the air to see the values printed to the screen. The `x`, `y`, and `z` values here define a 3D vector that represents the direction that the iPhone is pointing in space.

The Complete Code

In case you've run into any snags and need to double-check your work, here is the complete code for the `template.mm` file of this project:

```
/*
 *  template.mm
 *  template
 *
 *  Created by SIO2 Interactive on 8/22/08.
 *  Copyright 2008 SIO2 Interactive. All rights reserved.
 *
 */

#include "template.h"
#include "../src/sio2/sio2.h"

void templateRender( void )
{
     const GLfloat squareVertices[] = {
                    -100.0f, -100.0f,
                     100.0f,  -100.0f,
                    -100.0f,  100.0f,
                     100.0f,   100.0f,
                     };

     const GLubyte squareColors[] = {
                     0,   255,  255,  255,
                     255,   0,  255,  255,
                     255, 255,    0,  255,
                     0,   0,   0,    0,
                     };

     const GLubyte primaryColors[] = {
                    255,   0,   0,   0,
                    0,   255,   0,   0,
                    0,     0, 255,   0,
                    255, 255, 255, 255,
                    };

     glMatrixMode( GL_MODELVIEW );
     glLoadIdentity();

     glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );

     sio2WindowEnter2D( sio2->_SIO2window, 0.0f, 1.0f );
     {
          glVertexPointer( 2, GL_FLOAT, 0, squareVertices );
          glEnableClientState( GL_VERTEX_ARRAY );
          glColorPointer( 4, GL_UNSIGNED_BYTE, 0, squareColors );
          glEnableClientState( GL_COLOR_ARRAY );

          glTranslatef( sio2->_SIO2window->scl->x * 0.5f,
                           sio2->_SIO2window->scl->y * 0.5f, -0.0f );

          static float rotz = 0.0f;
          glRotatef( rotz, 0.0f, 0.0f, 1.0f );
          rotz += 90.0f * sio2->_SIO2window->d_time;

          glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

          glPushMatrix();
               glRotatef(-2*rotz, 0.0f, 0.0f, 1.0f );
               glTranslatef(125.0f, 0.0f, 0.0f);
               glScalef(0.2f, 0.2f, 0.2f);
               glColorPointer( 4, GL_UNSIGNED_BYTE, 0, primaryColors );
               glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
          glPopMatrix();

          glPushMatrix();
               glTranslatef(125.0f, 0.0f, 0.0f);
               glRotatef(-2*rotz, 0.0f, 0.0f, 1.0f );
               glScalef(0.2f, 0.2f, 0.2f);
               glColorPointer( 4, GL_UNSIGNED_BYTE, 0, primaryColors );
               glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
          glPopMatrix();
     }
     sio2WindowLeave2D();
}

void templateShutdown( void )
{
        // Clean up
        sio2ResourceUnloadAll( sio2->_SIO2resource );
        sio2->_SIO2resource = sio2ResourceFree( sio2->_SIO2resource );
        sio2->_SIO2window = sio2WindowFree( sio2->_SIO2window );
        sio2 = sio2Shutdown();
        printf("\nSIO2: shutdown...\n" );
}

void templateScreenTap( void *_ptr, unsigned char _state )
{
     if( sio2->_SIO2window->n_touch )
     {
          printf("templateScreenTap >> state:%d tap:%d x:%f y:%f\n",
                    _state,
                    sio2->_SIO2window->n_tap,
                    sio2->_SIO2window->touch[ 0 ]->x,
                    sio2->_SIO2window->touch[ 0 ]->y );
     }
}
```

```
void templateScreenTouchMove( void *_ptr )
{
     if( sio2->_SIO2window->n_touch )
     {
          printf("templateScreenTouchMove >> x:%f y:%f\n",
                    sio2->_SIO2window->touch[ 0 ]->x,
                    sio2->_SIO2window->touch[ 0 ]->y );
     }
}

void templateScreenAccelerometer( void *_ptr )
{
     printf("templateScreenAccelerometer >> x:%f y:%f z:%f\n",
               sio2->_SIO2window->accel->x,
               sio2->_SIO2window->accel->y,
               sio2->_SIO2window->accel->z );
}
```

You now have a pretty good idea of how the SIO2 template works to quickly get you started creating interactive graphical content on the iPhone and iPod Touch. You've also learned something about how OpenGL code is integrated into the SIO2 programming environment. But you haven't yet seen anything happening in 3D. You can probably tell already, though, that entering floating-point numbers into vertex arrays by hand is not how you want to be creating sophisticated 3D models for your virtual worlds. In the next chapter, you'll put Xcode aside and take a brief detour through the world of creating 3D content the way it ought to be created—with Blender! The chapter after that will bring the threads together and show you how to work with 3D content from Blender directly in SIO2.

Saying Hello to the Blender/SIO2/iPhone World

In this chapter, you'll get down to the business of creating 3D content for the iPhone using Blender. First, you'll take a quick look at where Blender fits into the Blender/SIO2/Xcode game development pipeline. Then you'll get straight to the first part of a relatively simple hands-on example, putting a new 3D spin on the classic Hello World program.

- **The Blender/SIO2/Xcode workflow**
- **An overview of SIO2**
- **Hello 3D World! Creating your world in Blender**
- **Exporting to the SIO2 format**

The Blender/SIO2/Xcode Workflow

Programming games for a specific platform such as the iPhone can be relatively easy or relatively difficult, depending very much on the tools you have at your disposal for game creation and their ability to compile the game to the appropriate platform. On the easy end of the spectrum for game creation, you have graphical game logic editing tools such as those in the Blender Game Engine (BGE), which can be used to create sophisticated interactive environments with minimal programming knowledge. However, if you take this route to create your game, you will be very restricted in terms of how and where you deploy your game. There's currently no way to compile a BGE game directly to the iPhone, or any other mobile platform for that matter. On the other end of the spectrum lies low-level game programming in a language like C using a graphical library like OpenGL ES. If you can do this, you'll have a great deal of flexibility in terms of the platforms you can develop for. However, even for people who know these technologies well, modeling, texturing, animation, and other aspects of content creation are not best approached from a programming standpoint. These are things that should be done with the appropriate content creation program. Furthermore, the logic that underlies graphics libraries like OpenGL ES is quite different from the way high-level game designers tend to think about things. Bridging this conceptual gap can be challenging, and tools that help to make the relationship between game design and game programming more transparent are always welcome. That is exactly what SIO2 is.

An Overview of SIO2

The SIO2 engine comprises several components. There is the `.sio2` file format specification, an exporter written in Python to export 3D data from Blender to the `.sio2` format, and a library of classes and functions written in C and C++ to enable you to access and work with data from the `.sio2` file format in the iPhone SDK environment.

The way these tools work together with Blender and Xcode is illustrated in Figure 3.1. The first step in the process is to create your scene in Blender. In general, a scene created for use with SIO2 is created in the same way as any other Blender scene, but there are a number of crucial differences that will be discussed when they arise throughout this book. Some Blender settings have a specific meaning when exported to the `.sio2` file format, which is not the same as their meaning within Blender. One obvious and drastic example of this is that sound files are exported as texture channels. Notwithstanding a few such quirks, though, the scene you create in Blender is very much like the scene you will eventually be working with in the iPhone development environment.

Once the scene is ready to be exported, the SIO2 exporter script is run from within Blender. You'll see how to do this later in the chapter, but if you've used scripts in Blender before, it will be straightforward. The output of running this script is an `.sio2` file. This file contains all the 3D data that will be accessible to the iPhone app.

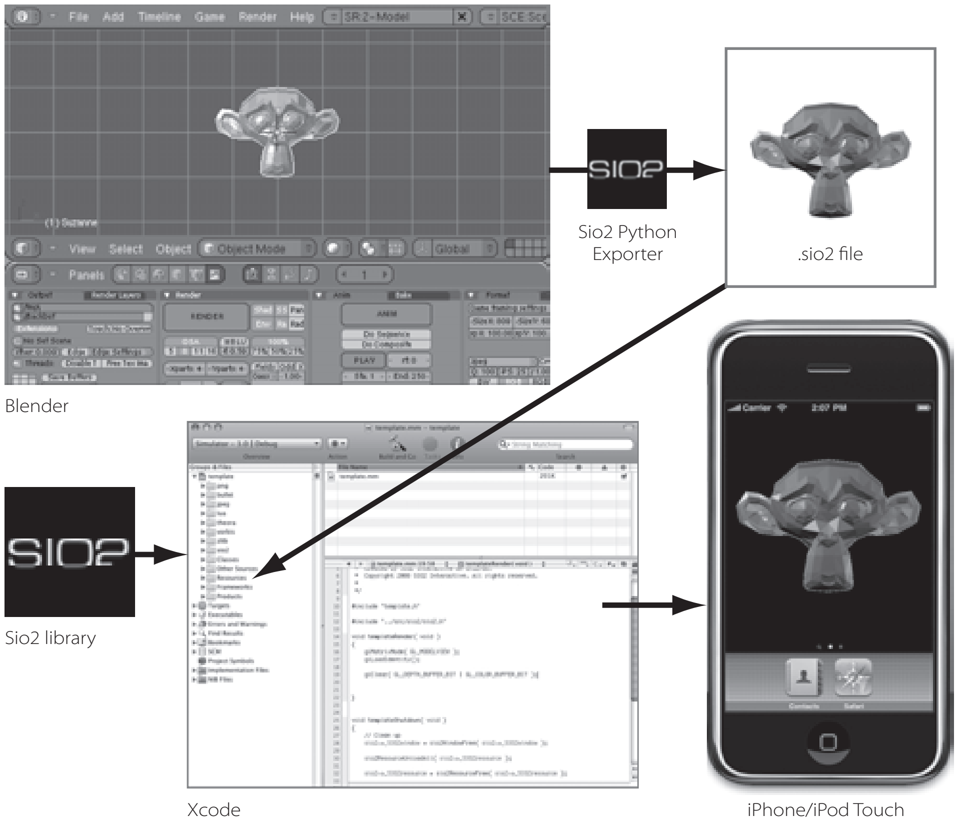


Figure 3.1

An overview of the Blender/SIO2/Xcode workflow

After the `.sio2` file is created, it is placed in the `Resources` directory of your Xcode project so that its data can be available to the app. You'll recall from Chapter 1 that the `Resources` directory is where data other than code is placed so that it can be accessed by the app. Once the `.sio2` file is in the `Resources` directory, you will be able to access the 3D assets within Xcode using the tools provided by the SIO2 library.

As you can see from the diagram of the workflow, the first step toward implementing your 3D application is to create the 3D assets you will need in Blender and export them. The rest of the chapter will be devoted to showing you how to do that. In the next chapter, you'll begin with the 3D content you created here and see how to work with it using SIO2 in Xcode.

Hello 3D World! Creating Your World in Blender

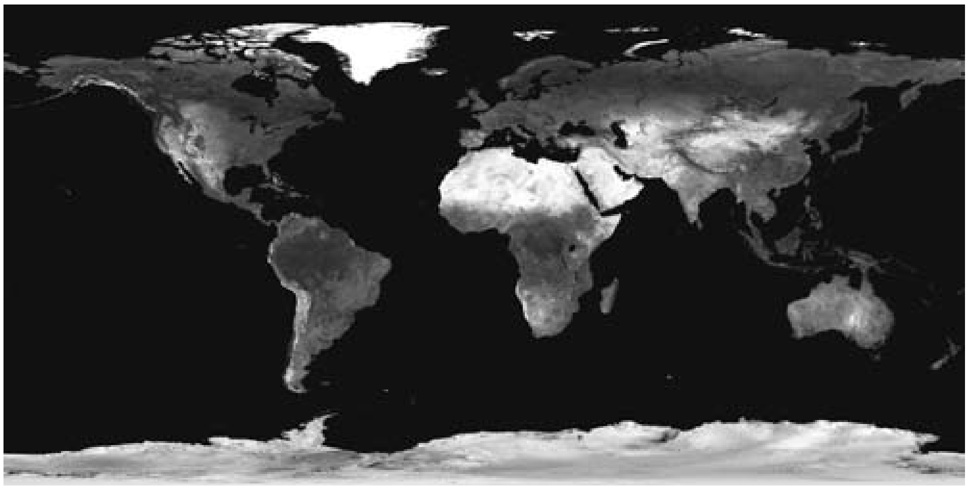
In the following sections, you'll model the planet Earth and a simple background. You'll place the camera and some lights, and export the whole scene for use in your iPhone app later. Although all the steps are described, it's assumed you know the basics of Blender. If you don't feel confident that you do, this would be a good time to take a detour and read through Appendix A. The material in that appendix is the bare minimum you should know in order to make the most of the Blender tutorials in this book.

If you *do* know the basics of Blender, please pay close attention anyway. Creating content for SIO2 in Blender is not identical to other things you might do in Blender. As you'll see in detail in the following sections, there are subtle things you must keep in mind, such as quirks about naming objects and restrictions on texture image size. There are also a number of attributes and settings that have a different meaning to SIO2 than they are taken to mean either in Blender or in the BGE.

Modeling and Texturing the World and Background

Modeling the planet Earth for this exercise will be pretty simple. An unmodified textured sphere will do fine. The texturing will be done simply as well: no bump mapping or normal mapping, no cloud cover or specularity adjustment—just a simple planet Earth color map like the one from NASA's Visible Earth project (`http://visibleearth.nasa.gov`) illustrated in Figure 3.2, which you will find in the downloadable archive that accompanies this book.

Figure 3.2
A color texture for planet Earth



NASA GODDARD SPACE FLIGHT CENTER IMAGE BY RETO STÖCKLI (LAND SURFACE, SHALLOW WATER, CLOUDS). ENHANCEMENTS BY ROBERT SIMMON (OCEAN COLOR, COMPOSITING, 3D GLOBES, ANIMATION). DATA AND TECHNICAL SUPPORT: MODIS LAND GROUP; MODIS SCIENCE DATA SUPPORT TEAM; MODIS ATMOSPHERE GROUP; MODIS OCEAN GROUP. ADDITIONAL DATA: USGS EROS DATA CENTER (TOPOGRAPHY); USGS TERRESTRIAL REMOTE SENSING FLAGSTAFF FIELD CENTER (ANTARCTICA); DEFENSE METEOROLOGICAL SATELLITE PROGRAM (CITY LIGHTS).

There are a few things you'll have to do before the scene is ready for export to SIO2. First, follow these steps to create your planet:

1. Fire up Blender and delete the default cube. Do this by right-clicking the object to select it and then clicking the X key to delete it. Press Enter or left-click on the dialog box shown in Figure 3.3.
2. Add a UV sphere by pressing the spacebar to bring up the floating menu and selecting Add → Mesh → UVsphere, as shown in Figure 3.4.

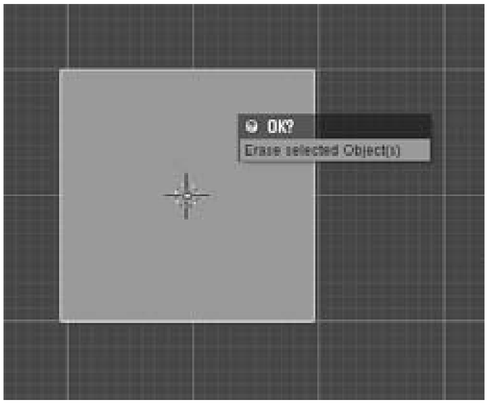


Figure 3.3
Deleting the Default cube

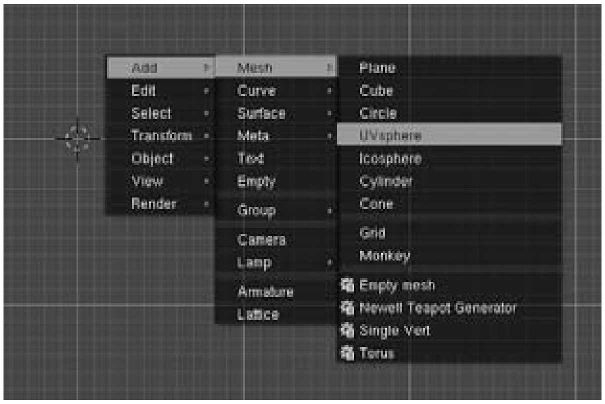


Figure 3.4
Adding the UV sphere

3. Leave the default values at 32 rings and 32 segments, as shown in Figure 3.5, and click OK.
4. The resulting sphere is shown in Figure 3.6. The object's location will be wherever the 3D cursor was when you added the object. In case you added the object somewhere other than the center of the space, press Alt+G to move it to the center of the space. Depending on your user preferences, the object may also be rotated. Press Alt+R to remove any rotations.

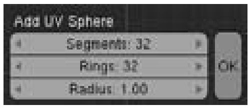
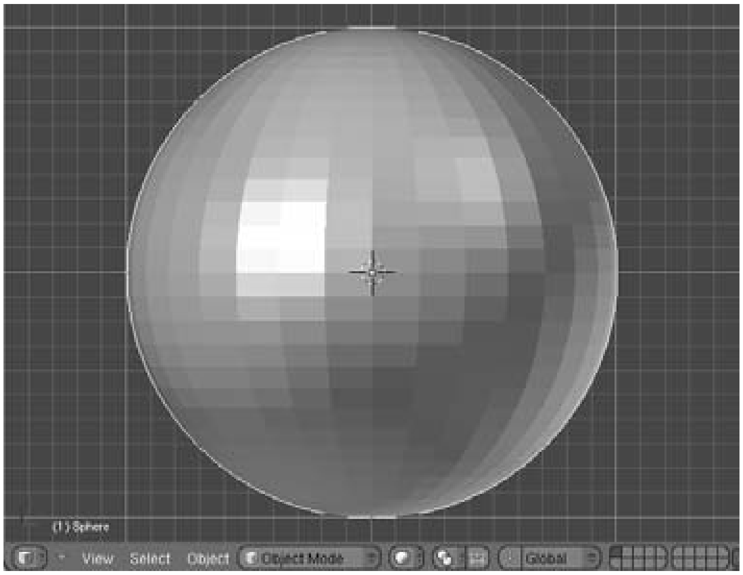


Figure 3.5
Default values for the UV sphere

Note that the default values of 32 rings and 32 segments result in a fairly dense mesh. In a lot of cases, particularly for use in mobile graphics, you will not need such a dense sphere and should use lower values for rings and sections. In this case, however, this denser mesh is appropriate for two reasons. The first is that there will be almost no more vertices in the scene (aside from four more in the background). So a somewhat detailed object isn't going to slow anything down too much. The second reason is that this example will eventually make use of dynamic lighting. Lamps in the scene act on vertices, and the effect is much better on denser meshes. If dynamic lighting is not needed, it's better to use lower poly meshes and get your

Figure 3.6
A UV sphere



lighting effects by using static textures, as you will do in other tutorials later in this book.

5. With the sphere selected, go to the Link And Materials tab, shown in Figure 3.7, select Set Smooth, and then add a material by clicking New in the Material button panel. This material will hold the image texture for the object.
6. Add a texture in the Texture tab of the Material buttons (F5), shown in Figure 3.8, by clicking Add New.

Figure 3.7
Adding a material in the Link And Materials panel

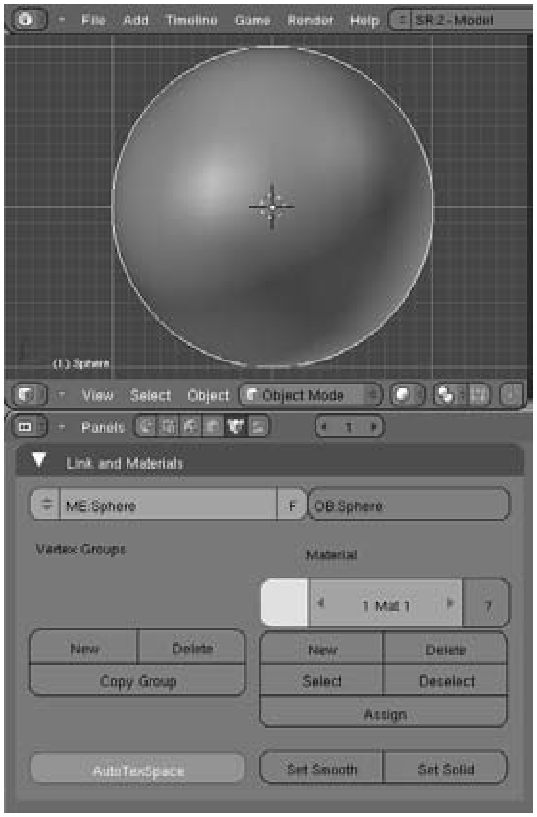
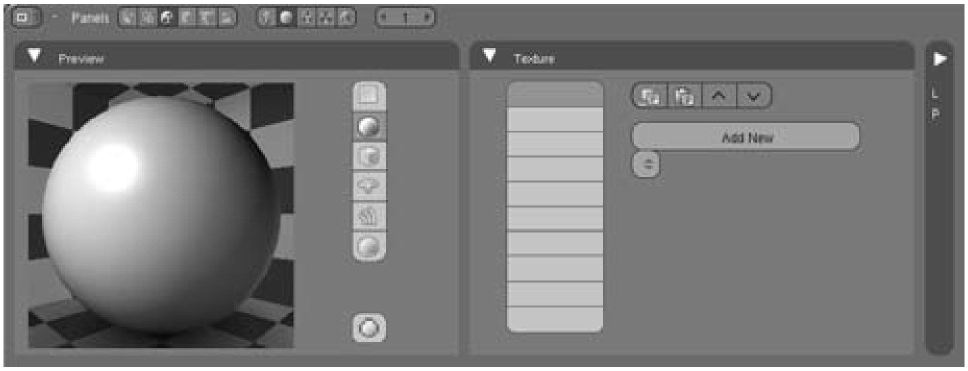


Figure 3.8
Adding a texture to the material



7. In the MA field in the Links And Pipeline panel of the Material buttons area, rename the material **MatEarth.**

Blender has two distinct ways to apply textures to objects. For ordinary animation and rendering, textures are associated with materials on objects. Alternately, you can use only *texture faces* without a material, which can be viewed in the BGE. Texture faces are automatically created whenever an object is UV mapped with a texture, whether a material is present or not. UV textures are present on materials only if they have been explicitly added to a material and set to UV mapping. This can be a confusing distinction, but it is important to understand. When you work with SIO2, the relationship becomes even muddier. SIO2 uses material textures (the first two texture channels of a material, to be specific) but is restricted to using only UV mapped image textures. Furthermore, SIO2 also takes several important settings from the Texture Face panel values. You'll learn more about this in Chapter 6, when you read about using billboard textures.

8. Enter the Texture buttons area (F6) and select Image for the texture type, as shown in Figure 3.9.

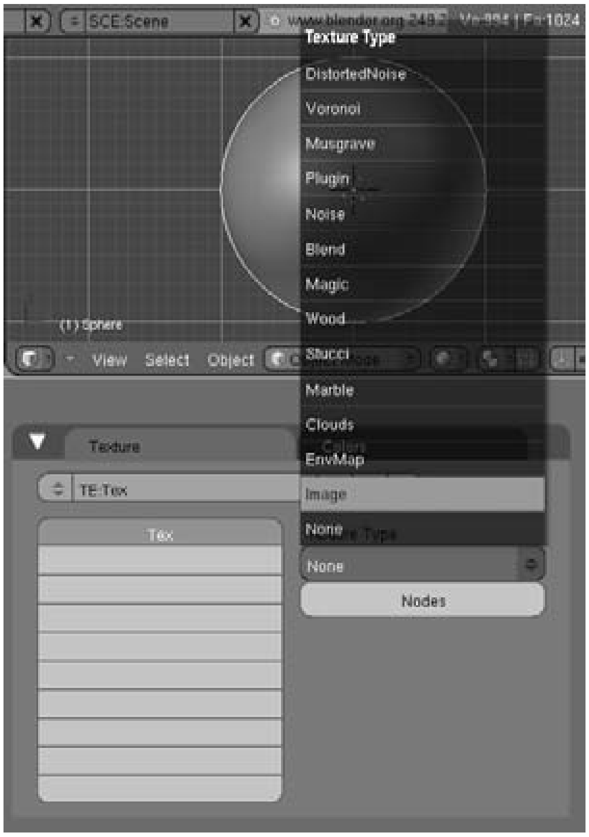
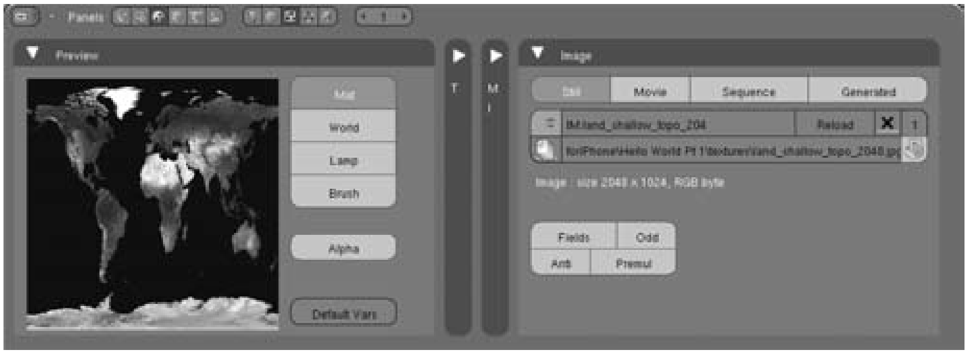


Figure 3.9
Setting an image texture

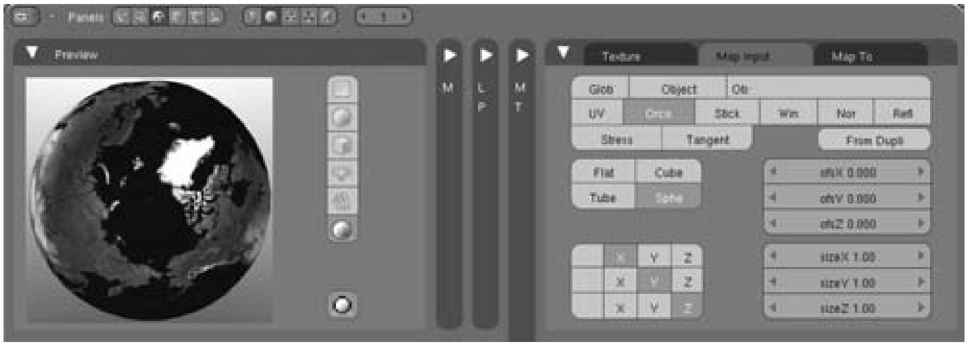
9. Click the Load button on the Image panel to open a file browser and find the planet Earth texture you downloaded previously. Load it so that it appears as shown in Figure 3.10. This kind of rectangular texture can be mapped onto a sphere shape using the Orco mapping coordinates and Sphere mapping.
10. Set those values on the Map Input tab, as shown in Figure 3.11.

Figure 3.10
Loading the Earth surface texture



IM:land_shallow_topo_204
ts\iPhone\Hello World Pt 1\textures\land_shallow_topo_2048.jpg
Image : size 2048 x 1024, RGB byte

Figure 3.11
Sphere mapping the texture



ofsX 0.000
ofsY 0.000
ofsZ 0.000
sizeX 1.00
sizeY 1.00
sizeZ 1.00

11. If you render the image now or preview it in the 3D viewport using Shift+P, you will see that the texture is correctly applied, as shown in Figure 3.12. Note that Shift+P must be pressed in Object mode; after that you can switch to Edit mode to see the mesh structure, as shown in the figure.
12. If you were only interested in rendering the planet Earth with a color texture, this would be sufficient. However, SIO2 requires UV textures, and the mapping for this texture is a spherical texture mapping, not a UV mapping. To get SIO2 to recognize the texture, it is necessary to bake the current texture to a UV map. To do this, you need to unwrap the sphere and create a UV texture on it. Select the edge loop around the equator of the sphere by pressing Shift+Alt and right-clicking on one of the edges in that edge loop (see Figure 3.13).

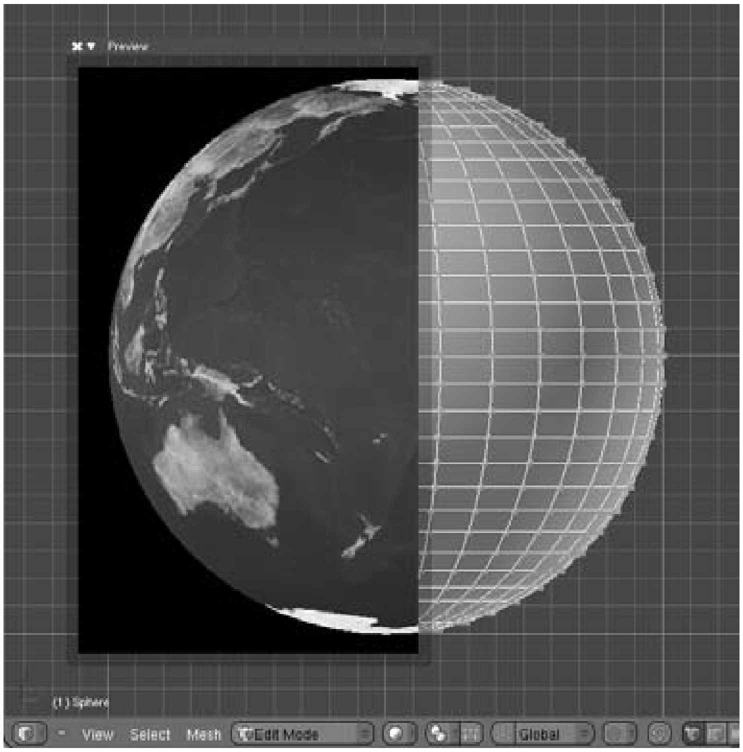


Figure 3.12
The sphere-mapped texture

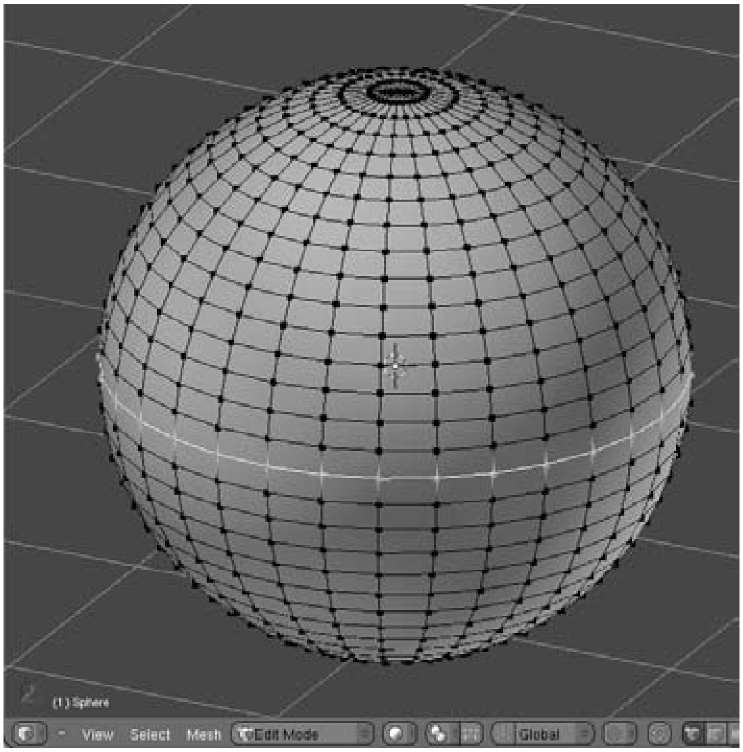
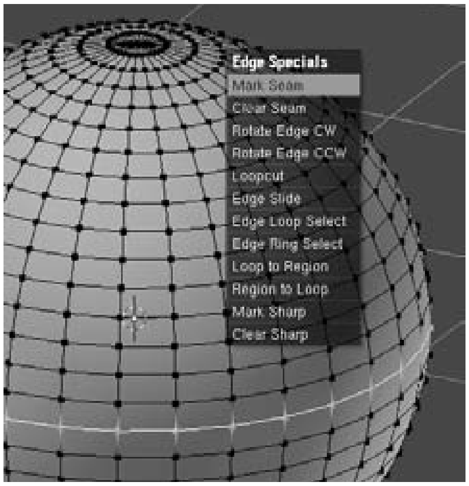


Figure 3.13
The equator edge-loop selected

13. Press Ctrl+E to bring up the Edge Specials menu, and from there choose Mark Seam, as shown in Figure 3.14.
14. Add a UV texture to the mesh by clicking the New button next to the UV Texture label on the Mesh panel in the Editing button area (F9) (Figure 3.15).
15. Select all faces of the object using the A key, and then open up a UV/Image Editor window and press the E key over the mapping area to unwrap the sphere (Figure 3.16). The resulting unwrapped mesh should appear in the UV/Image Editor window as two spiderweb-patterned circles, as shown in Figure 3.17.

Figure 3.15
Adding a UV texture



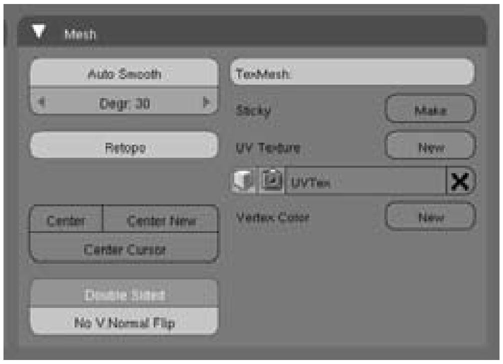
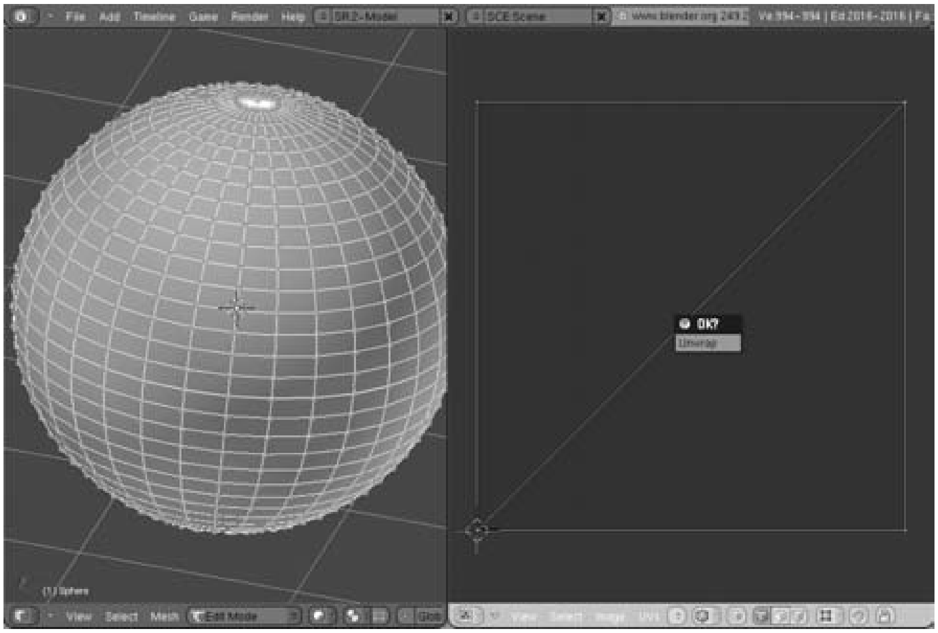


Figure 3.14
Marking the seam

Figure 3.16
Unwrapping the sphere in the UV/Image Editor



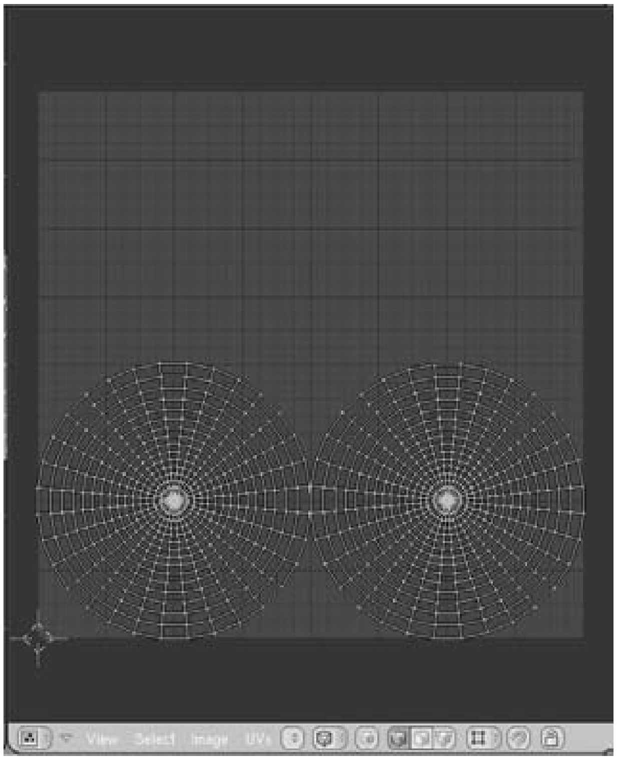
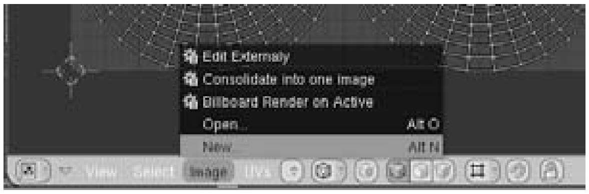


Figure 3.17
The unwrapped mesh pattern

16. Create a new UV texture image by choosing New from under the Image menu in the header of the UV/Image Editor window as shown in Figure 3.18. Leave the Width and Height values at 1024, and select UV Test Grid as shown in Figure 3.19. When you've done this, you can see the texture on the object, as shown in Figure 3.20, by viewing the object using the Textured Draw Type option from the 3D Viewport header.

Figure 3.18
Creating a new UV texture image



All graphics that you plan to use in SIO2 need to be sized according to powers of two. Images can have 64, 128, 256, 512, or 1,024 pixels per side, but images with dimensions that are not powers of two will not be rendered by OpenGL ES in the iPhone. This is an easy mistake to make, and no error will be raised at any point; your objects will simply fail to appear. For this reason, it's a good thing to bear in mind and to double-check if you have a problem viewing your objects in the iPhone. You should always use the smallest image size that you can without sacrificing the quality of the final image.

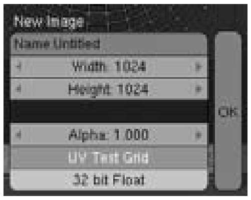


Figure 3.19
Values for the UV texture image

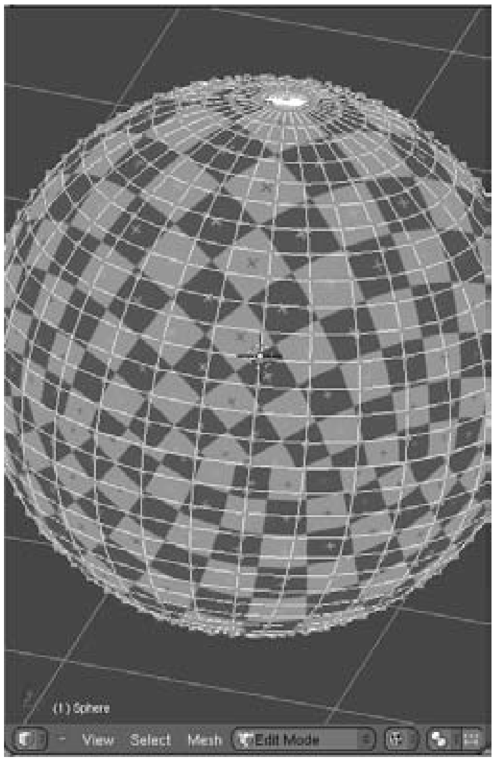


Figure 3.20
Viewing the textured faces with the Textured Draw Type option

17. You are now working with two textures. One of them is the sphere mapped texture on the material you created previously; this is the texture that will render if you render the scene. The other is the UV map on the texture face, which is visible in Textured Draw Type view mode. This is the texture that can *receive* rendered information in the form of texture baking. To see texture baking in action, go to the Bake tab in the Scene buttons area (F10) and set the bake value to Textures, as shown in Figure 3.21. Then go ahead and click the BAKE button. When you do this, you will see the UV test grid image transform before your eyes into the UV-mapped Earth surface texture shown in Figure 3.22. Be sure to *save the image* by selecting the Save As option from the Image menu in the UV/Image Editor header. Baked images are not automatically saved to distinct image files, and if you close Blender without saving the baked image, it will be discarded when you reopen Blender. Save the file as `earthUV.jpg`. Select Jpeg from the drop-down menu on the File Browser header when you save. JPEG files do not have an alpha channel, so it is not possible for them to carry transparency information. In this example, you won't be using transparency, so the smaller size of a JPEG is a good option. Saving video memory is critical when programming for mobile devices, so any way you can conserve is worthwhile.

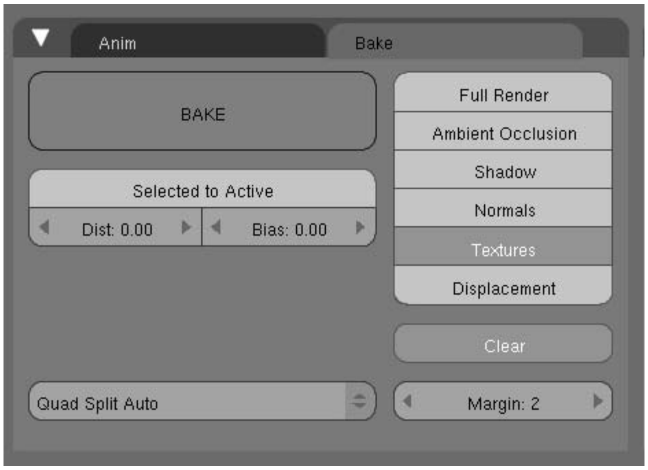


Figure 3.21
The Bake tab

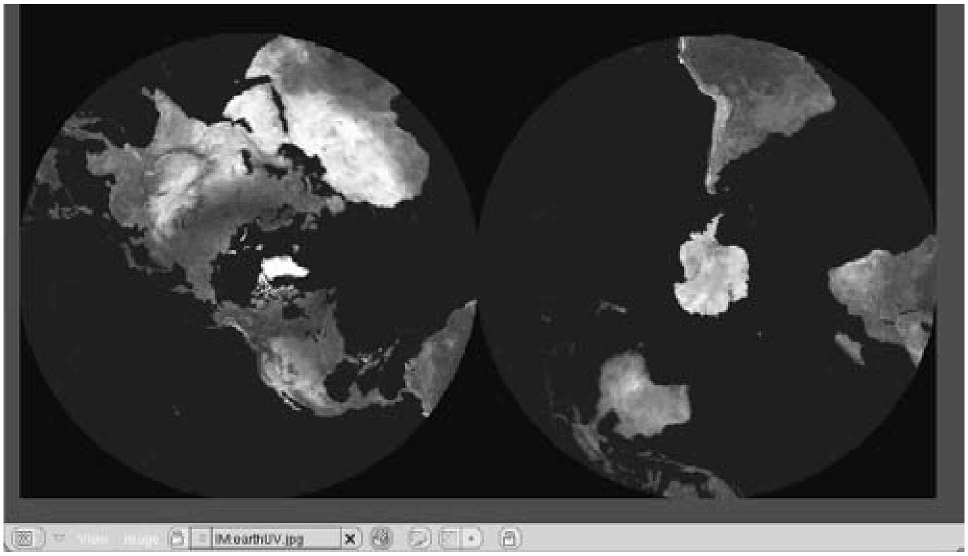


Figure 3.22
The baked UV-mapped Earth surface texture

18. Now you have a properly UV-mapped Earth texture. You must now replace the sphere-mapped texture on the material with the UV-mapped texture you just created. First, you need to replace the image associated with the image texture by selecting the new image from the drop-down menu in the Image tab in the Texture buttons area, as shown in Figure 3.23. Then you need to change the Map Input values in the Material buttons area from Orco and Sphere to UV and Flat, as shown in Figure 3.24. If you preview the render now, using Shift+P, you will see that the UV-mapped image is now set as the image to render, as shown in Figure 3.25.

Figure 3.23

Selecting earthUV .jpg as the texture image

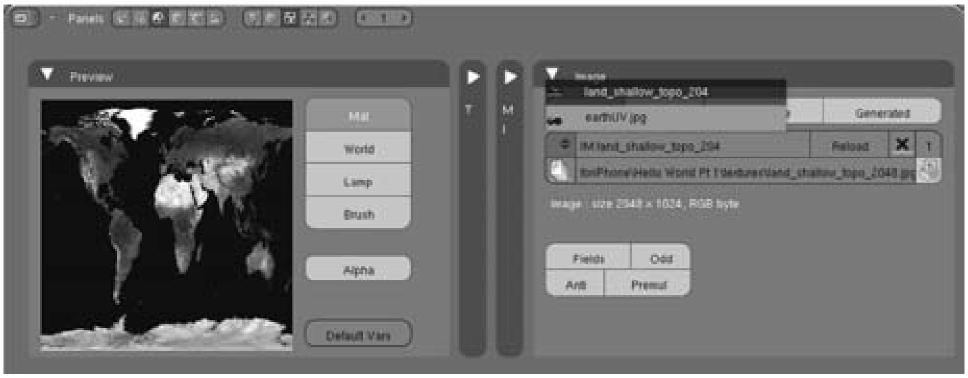


Figure 3.24

Changing the Map Input values to UV mapping

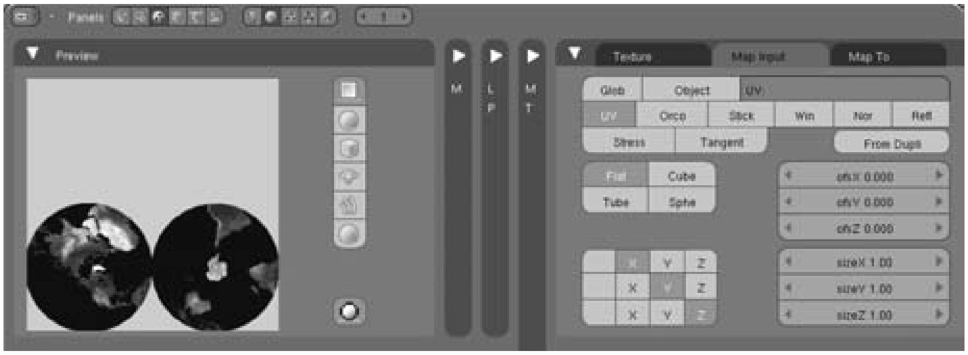
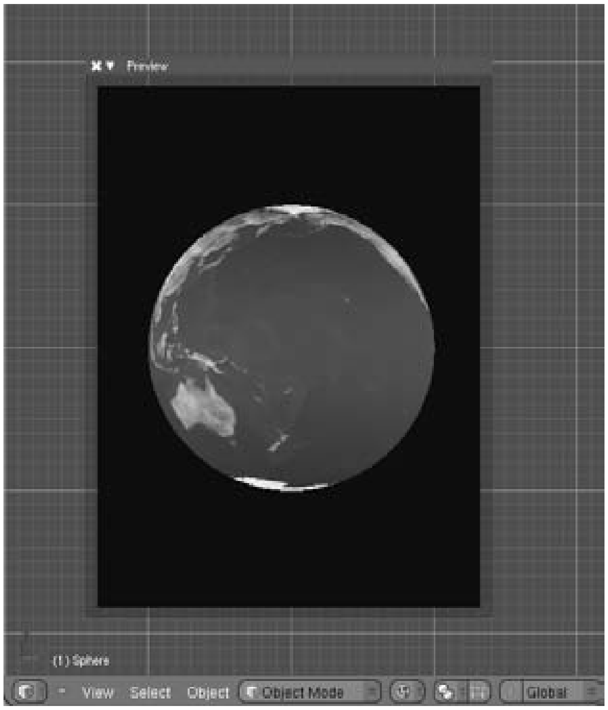


Figure 3.25

Previewing the UV mapped model



19. Aside from UV textures, the settings on the Blender materials themselves have minimal effect on how the object will look in SIO2. There are a few places where they do. Changing the specular and diffuse reflection colors will change them in the SIO2 file also. The Hard value on the Shaders tab of the Material buttons will affect the shininess value of the SIO2 file output. For this example, the results look better with a low shininess value, so set the value of Hard to 5.

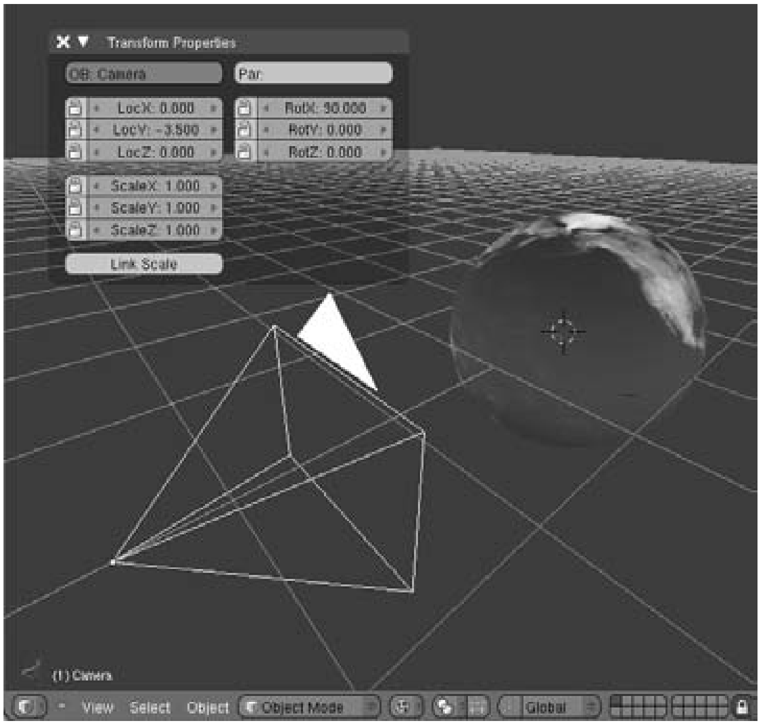
You're now finished with modeling and texturing the planet Earth for this example, but the scene is not quite finished. You still need to set up the camera, and adding some lights and background will make the scene look nicer.

Lights, Camera, and Background

As in the BGE, using lights and real-time shading is optional. You can create a scene that uses no lights at all and have all of your shadows and lighting effects represented by textures. In many cases, this is the best way to go for static objects because real-time lighting requires processor resources to calculate and because meshes lit with real-time lights require a higher vertex density to look good. In this example, however, you'll use real-time lights. For one thing, the scene complexity is limited, and a reasonable mesh density can be used on the sphere. For another thing, you will later make the sphere spin, and dynamic objects cannot be lit convincingly using static textures only.

Placing the camera is simple. Select the camera in Object mode and press the N key over the 3D viewport to bring up the Transform Properties window. Set the values as shown in Figure 3.26. The location along the x- and z-axes is **0**, and the location along the y-axis is **–3.5**. Rotation around Y and Z is **0**, and rotation around X is **90**. There's nothing special about these values—they simply place the camera in a position to get a nice straight shot of the planet Earth model you created. You don't need to make any other changes to the camera for this exercise. The camera won't be moving in this example, so you don't need to worry about gimbal lock, which can arise when an animated camera points straight down with 0 rotation on all axes.

Figure 3.26
Transform properties for the camera



Transform Properties

OB: Camera
Par:

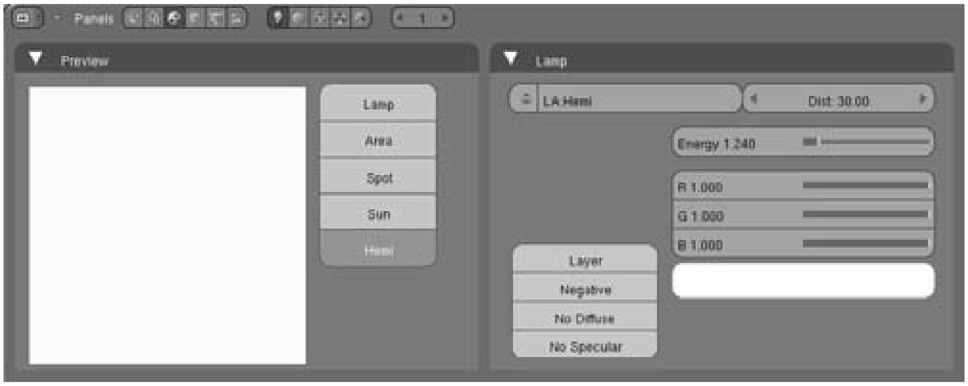
LocX: 0.000
LocY: -3.500
LocZ: 0.000
RotX: 90.000
RotY: 0.000
RotZ: 0.000

ScaleX: 1.000
ScaleY: 1.000
ScaleZ: 1.000

For this example there will be two lights: one for key lighting and one for a slight backlight from the opposite direction. By default there is a single light of the omnidirectional Lamp type. Select this lamp and change it to a Hemi light in the Lamp buttons area, as shown in Figure 3.27. Give it an Energy value of **1.24**. Finally, change its Object name to **Hemi**.

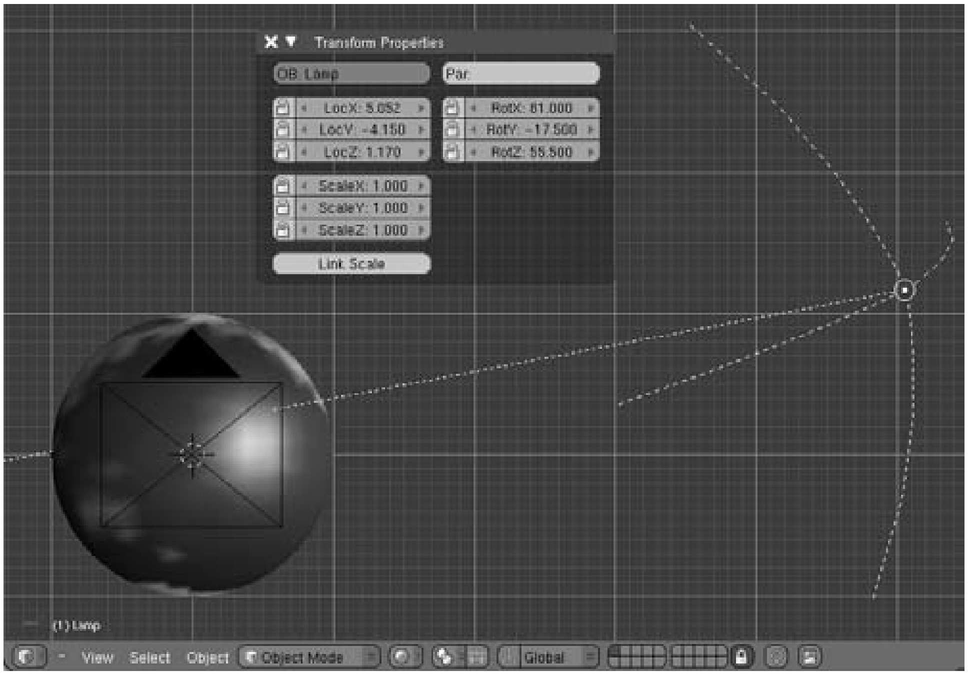
Once again, in the 3D viewport press the N button to bring up the Transform Properties window, if it isn't already up. Enter the rotation and location values shown in Figure 3.28 (LocX: **5.052**, LocY: **–4.15**, LocZ: **1.17**, RotX: **81**, RotY: **–17.5**, RotZ: **55.5**).

Figure 3.27
Lamp values for the Hemi lamp



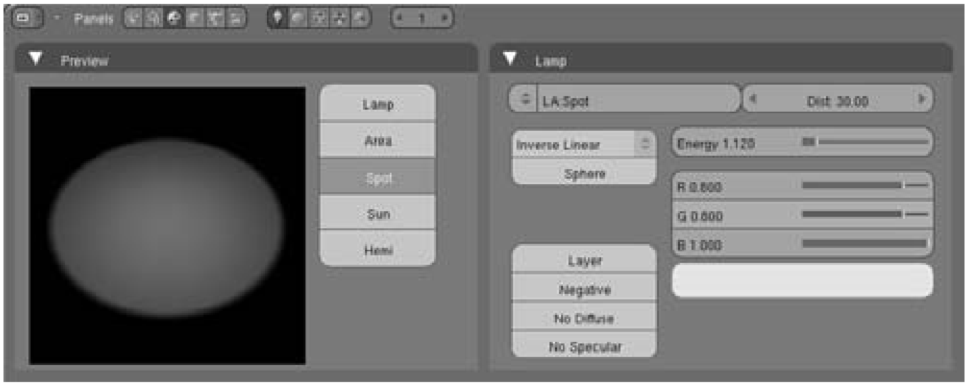
LA:Hemi
Dist: 30.00
Energy 1.240
R 1.000
G 1.000
B 1.000

Figure 3.28
Transform values for the Hemi lamp



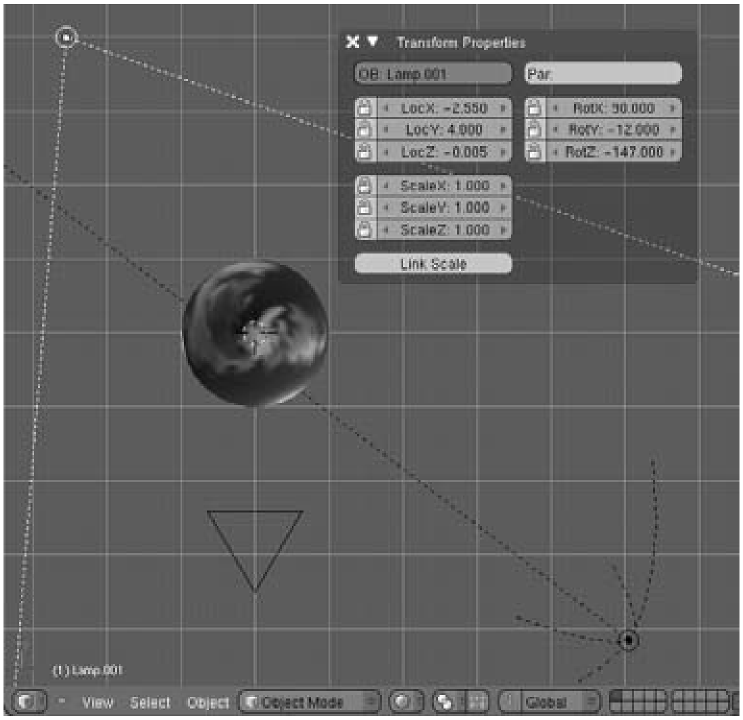
OB: Lamp
Par:
LocX: 5.052
LocY: -4.150
LocZ: 1.170
RotX: 81.000
RotY: -17.500
RotZ: 55.500
ScaleX: 1.000
ScaleY: 1.000
ScaleZ: 1.000

Create the second lamp by copying the Hemi lamp with Shift+D. Switch the lamp type to Spot and rename the object **Spot**. Renaming objects when you copy them is important because the `.001` object suffix format has a specific meaning in SIO2. It indicates pseudo-instancing, which you'll learn about later. This is a convenient feature when you want it, but it means that you must be careful not to accidentally have object names with that suffix. Set the Spot light's values as shown in Figure 3.29. Specifically, set the Energy at around 1.12 and push the blue value up to 1 while leaving red and green at 0.8. Set the Transform values for the Spot light in the same way you did for the camera and the Hemi light, with values as shown in Figure 3.30 (LocX: **–2.55**, LocY: **4**, LocZ: **–0.005**, RotX: **90**, RotY: **–12**, RotZ: **–147**).



LA:Spot
Dist: 30.00
Inverse Linear
Sphere
Energy 1.120
R 0.800
G 0.800
B 1.000

Figure 3.29
Lamp values for the spot light



OB: Lamp.001
Par:
LocX: -2.550
LocY: 4.000
LocZ: -0.005
RotX: 90.000
RotY: -12.000
RotZ: -147.000
ScaleX: 1.000
ScaleY: 1.000
ScaleZ: 1.000
(1) Lamp.001

Figure 3.30
Transform values for the spot light

Now that you've got the lights and camera set up as they should be, follow these steps to create a simple, glowing background that will appear as a kind of halo or aura around the planet:.

1. In Object mode, press the spacebar and select Add → Mesh → Plane to create a new plane, as shown in Figure 3.31. Depending on how you have your defaults set, you may automatically enter Edit mode. If you are in Edit mode, reenter Object mode. In case you added the plane somewhere other than the center (origin) of the 3D space, press Alt+G to clear the location of the object, and then press Alt+R to clear the object's rotation. The result should look like Figure 3.32.

Figure 3.31
Adding a plane

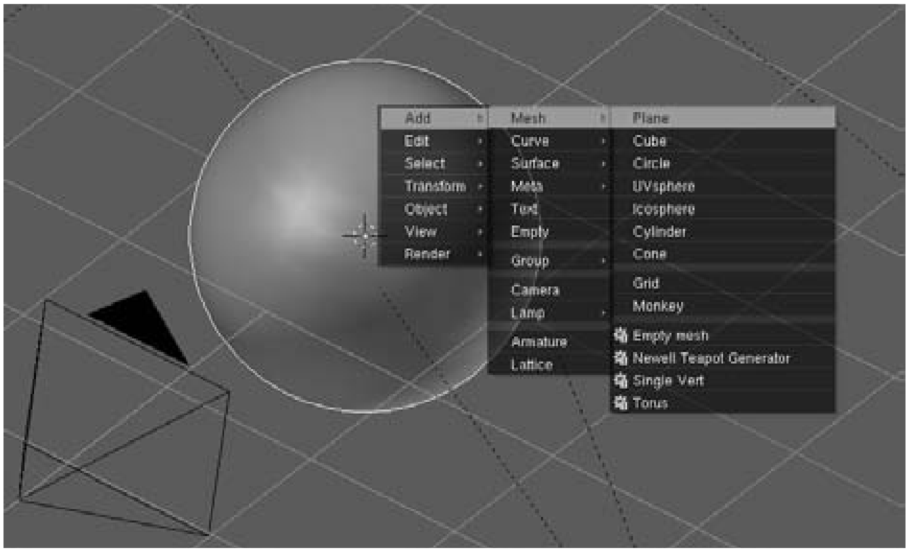
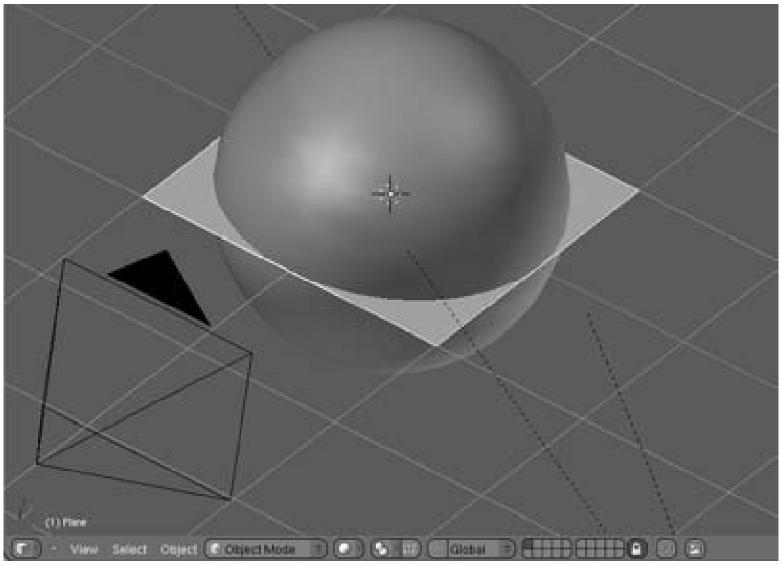
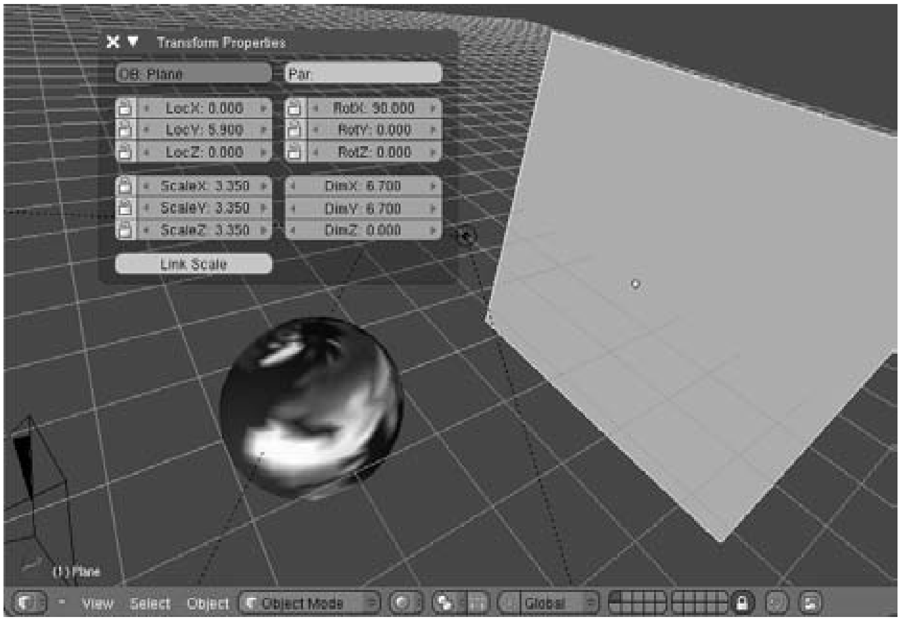


Figure 3.32
The plane at the origin



2. Set the location, rotation, and scale as shown in Figure 3.33. All the axes are scaled up by a factor of 3.35. The object is moved 5.9 units along the y-axis and rotated 90 degrees around the x-axis. Later, once you've seen the results, you can adjust these values to see how the background effect changes. The important thing is that the face normal must be pointing toward the camera; otherwise the face will not be visible by default.



OB: Plane | Par

LocX: 0.000	RotX: 90.000
LocY: 5.900	RotY: 0.000
LocZ: 0.000	RotZ: 0.000
ScaleX: 3.350	DimX: 6.700
ScaleY: 3.350	DimY: 6.700
ScaleZ: 3.350	DimZ: 0.000

Link Scale

Figure 3.33
Positioning the plane

3. Add a new material to the plane and add a texture to the material. Make the texture a blend type texture with values as shown in Figure 3.34. Set the type to Halo and use the Colorband on the Colors tab to create a gradation from black (R: **0**, G: **0**, B: **0**) to white (R: **1**, G: **1**, B: **1**) with both colors having an Alpha (A) value of **1**. Set the Map Input values as shown in Figure 3.35. The mapping type should be set to Orco and the XYZ texture mapping matrix at the lower-left corner of the Map Input tab should be set as shown.
4. Just as you did for the sphere mapping of the planet Earth model, you will bake this blend texture to a UV mapped image texture. Follow the same steps to add a UV texture to the plane, and then create a new image with the UV Test Grid pattern, except that this time you should create the new image to be 512×512 pixels. Unwrapping isn't necessary this time because the object is just a single face. In Texture Draw Type view, the plane will look like it does in Figure 3.36.

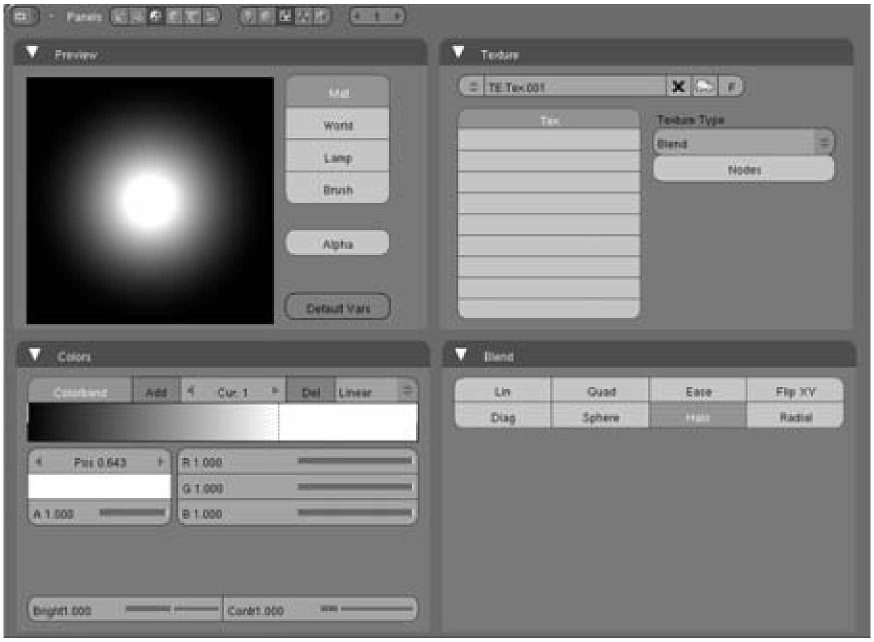


Figure 3.34
A blend texture for the plane

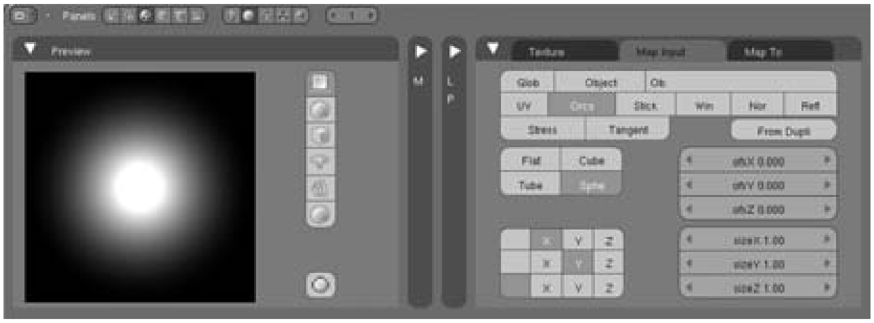


Figure 3.35
Mapping the blend texture

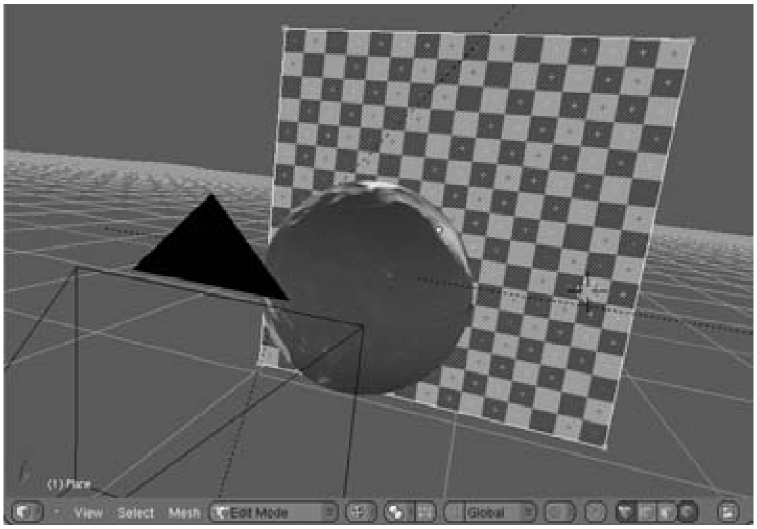


Figure 3.36
The new UV image texture on the plane

5. With the plane selected, go to the Bake panel in the Scene buttons area (F10) and once again select the Textures option and click the BAKE button. The halo blend texture will be baked onto an image, as shown in Figure 3.37. As you did with the planet Earth model in step 9 of the previous procedure, go back and replace the current material texture (in this case the Blend texture) with an Image texture using this newly baked image. Just as you did with the planet Earth model, select UV in the Map Input tab. Don't forget to save the image as a JPEG file!
6. This background isn't intended to reflect light, so turn the specularity and hardness values way down on the material. When you look at the result in the preview, the textured plane should provide a glow around the edge of the Earth model as shown in Figure 3.38.

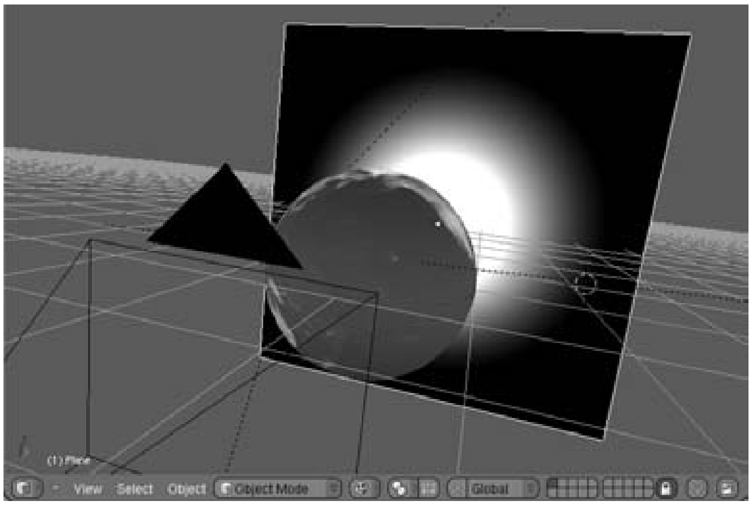


Figure 3.37
The baked UV texture on the plane

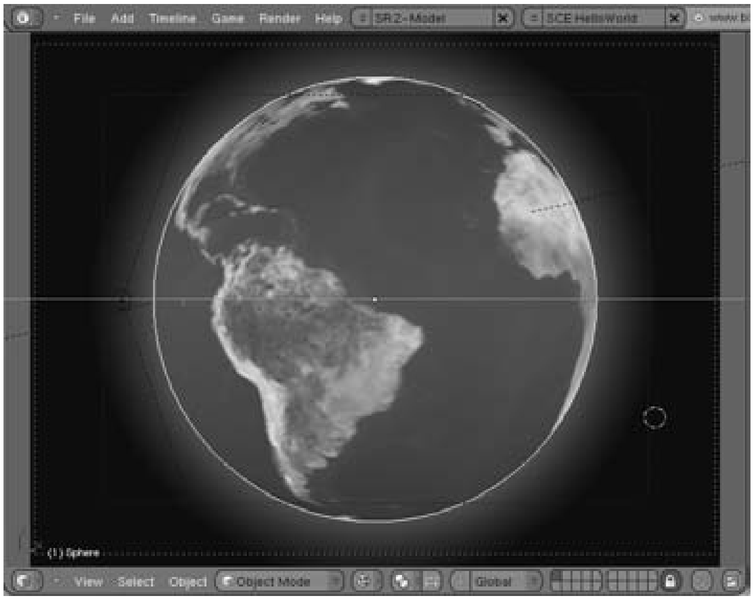


Figure 3.38
The final scene in preview

You're finished with the main modeling and texturing of the scene. It is now necessary to export the scene to the SIO2 file format using the Python export script bundled with SIO2. Before you do that, there's one more thing you should do with both of the mesh objects you've created. You should convert them both from quads to triangles. This step is optional because the SIO2 export script also has an option to convert from quads to triangles, but it is a good idea so that you don't forget or get confused about the state of your meshes. To convert a mesh from quads to triangles, select the mesh, enter Edit mode by pressing Tab, and select the entire mesh by pressing the A key. Finally, press Ctrl+T to convert the entire mesh to triangles. Your UV mapping values will be preserved. If you don't convert your quads to triangles, and forget to choose that option in the exporter, your quads will all show up with one blacked-out triangular half when you run your application on the iPhone.

Exporting to the SIO2 File Format

To make your 3D objects accessible to the SIO2 library in the Xcode environment, you must export them to the SIO2 file format. This is simple to do, but there are a few important points to keep in mind and some subtle details to be aware of that will help things go more smoothly.

The Exporter Script

For exporting Blender assets, a Python exporter is included with the standard SIO2 distribution. You can find this in the `SIO2_SDK/exporter` subdirectory. The script itself is called, intuitively enough, `sio2_exporter.py`. You run `sio2_exporter.py` in Blender just as you would any other standard Python script. You can either run it directly from the Blender Text Editor or install it in the `.blender/scripts` directory. If you install it, you will be able to access it from the main File → Export menu. Installing scripts in the `.blender/scripts` directory is not as straightforward on Mac OS X as it is on other platforms due to OS X's tendency to conceal important files from the user in the GUI. If you want to install scripts for Blender in OS X, you will need to copy them into the appropriate directory via the terminal command line. If you have installed Blender in the ordinary way in your applications directory, the location to copy the script to is `/Applications/Blender 2.49/blender.app/Contents/MacOS/.blender/scripts`. It's beyond the scope of this book (and unnecessary for the book's purposes) to get into the details of how to navigate the file system and move files around via the command line. However, bear in mind that the contents of `blender.app` can be accessed only by the command line and that the `.blender` directory is a hidden "." directory, as distinct from the `blender` executable found in the same subdirectory. You can find basic Unix command-line documentation on the Web. For the time being, it is fine to simply run the exporter directly from the Text Editor.

To do this, you need to open up a new Text Editor window area as shown in Figure 3.39. From the Text menu, select Open and navigate to the location of the `sio2_exporter.py` file on your hard drive, as shown in Figure 3.40. Select the file and click Open Text File.

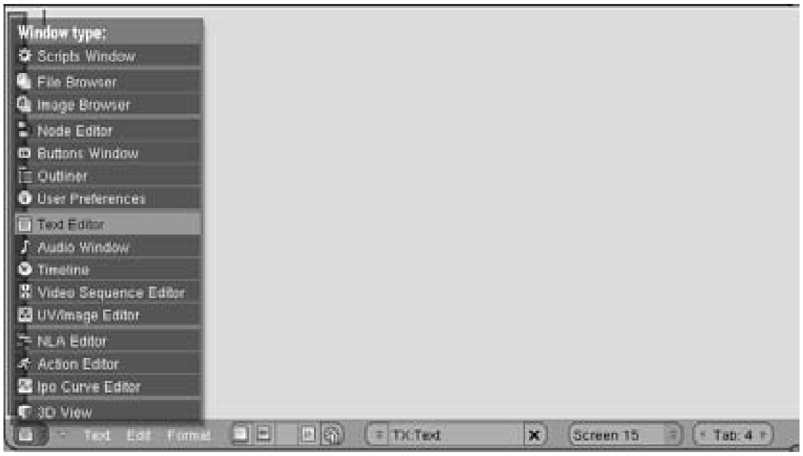
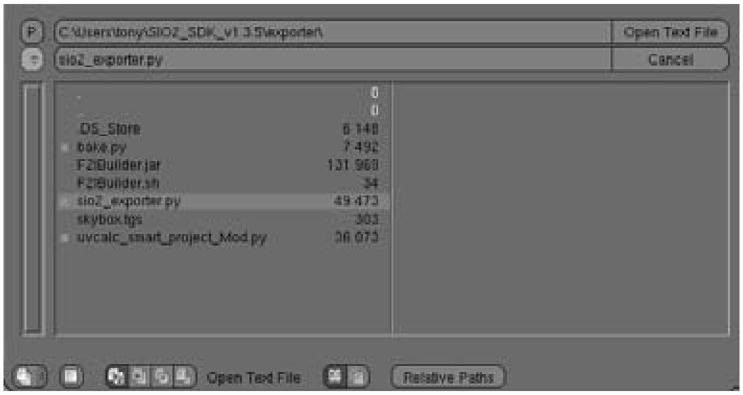


Figure 3.39
A Text Editor window area



```
C:\Users\tony\SIO2_SDK_v1.3.5\exporter\                Open Text File
sio2_exporter.py                                        Cancel

 .                                   0
 ..                                  0
 .DS_Store                       6 148
 bake.py                         7 492
 FZIBuilder.jar                131 960
 FZIBuilder.sh                      34
 sio2_exporter.py               49 473
 skybox.tga                        303
 uvcalc_smart_project_Mod.py    36 073

Open Text File    Relative Paths
```

Figure 3.40
Opening the exporter script

The script will appear in the Text Editor window as text. To run the script, you can select Text → Run Python Script from the header menu, or simply press Alt+P with the mouse over the Text Editor area. When you run the script, the Text Editor area will be replaced by the Scripts Window area and the script's interface will appear in that area, as shown in Figure 3.41.

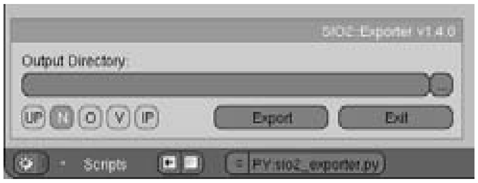


Figure 3.41
The interface of the exporter script

The Exit button halts the script, and the Export button executes the export. Before exporting, you must select the destination for your `.sio2` file in the Output Directory field. The ellipses (…) button to the right of the field will enable you to browse the file system to select the directory you want. Keep in mind that this field is for the directory *into which* the `.sio2` file should be placed. In general, you will probably want to choose your project directory so that you can keep all the necessary files for your project in one place. Because you haven't yet created a project directory for this project, you can export to whatever directory you choose—just be sure it's someplace you'll remember. Another important thing to keep in mind is that the exporter will also create a new directory with the same name as the `.sio2` file (you'll read more about this in the next section). When you have to export multiple times, be sure you don't accidentally start exporting into this subdirectory. A few confusing things can happen when you export repeatedly (as you will no doubt do, because you will want to continue to work with the Blender content as you develop your application). Normally, when you export multiple times, the same output directory will appear by default, so you won't have to set this by hand afresh each time you export. However, when you export repeatedly and delete multiple export files by moving them to the trash, the exporter sometimes gets confused and defaults to the trash directory. In these cases, you'll have to set the output directory again by hand.

The five buttons along the left have the following meanings:

UP updates a previously exported `.sio2` file. This speeds up the process of exporting but should be used only when updating objects that have already all been exported. If you are exporting for the first time, or if you have added objects to the scene that have not yet been exported for a first time, you should leave this button unselected.

N enables the export of normal direction information. In order for real-time lighting to be calculated, SIO2 must know about the direction of the normals of all the faces. If you do not select this option, real-time lighting will not be calculated correctly. In the present Hello3DWorld example, you need to export with this option.

O enables mesh optimization. This option automatically splits quads into triangles, converts triangles to triangle strips, and removes doubles. If you did not split the quad faces into triangles in Blender in advance, you should use this option.

V bakes the current lighting conditions onto the mesh by setting the appropriate vertex colors. Vertex colors are one way to get simple, static lighting effects on a mesh.

IP ensures that all Ipo curves are exported rather than just the ones that are linked to objects that are exported.

Exporting the World

To export a scene, you need to take the following steps:

1. Make sure all the objects you want to export are on Layer 1 of the Blender layers. If you created all the objects on Layer 1, you don't need to do this, but if you don't know, it can't hurt to be sure. To do this, select all objects with the A key in Object

mode, press the M key, select the farthest left square on the top row of the layer buttons (see Figure 3.42), and then click OK.

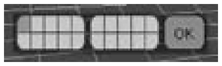


Figure 3.42
Sending all the objects to Layer 1

2. Make sure the scene name is the name of the file you want to export. Do this by editing the name in the SCE field of the User Preferences header at the top of the default Blender screen, as shown in Figure 3.43. The default scene name is Scene, and this is fine to use, but it will become confusing to have all of your `.sio2` files named `Scene.sio2`. It's better to give the scene a name. For this example, name the scene **Hello3DWorld**.

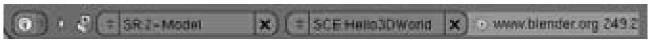


Figure 3.43
The SCE field with the name of the scene

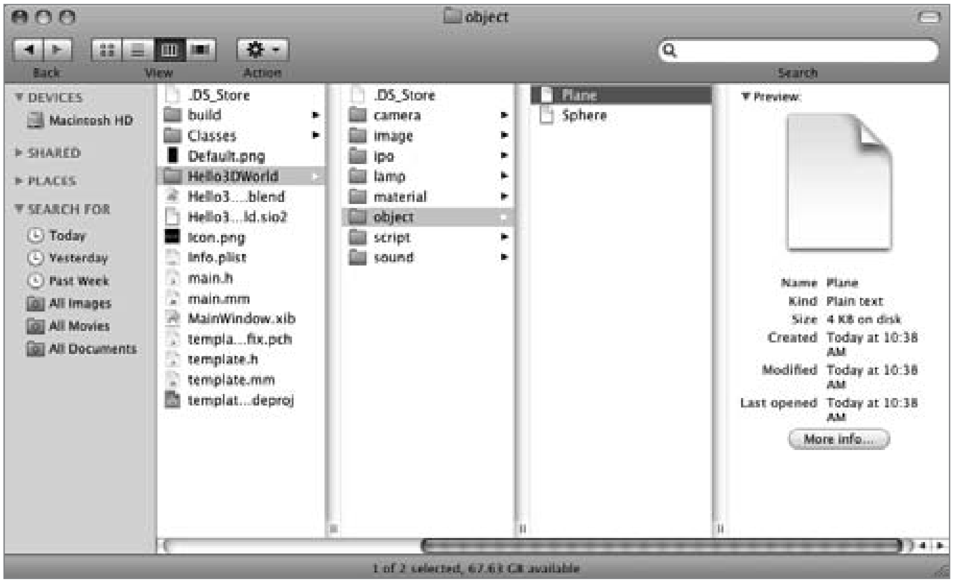
3. Make sure all the objects you want to export are selected in Object mode by pressing the A key if necessary. Select the N option on the exporter script interface to export normal directions, and click Export. Check your output directory to make sure that the file `Hello3DWorld.sio2` and a new directory with the same name (minus the `.sio2` suffix) have been created there.

The *.sio2* File Format

You have now finished exporting your Blender scene to a format that can be read and used by the SIO2 library in the Xcode environment. Before you go on and actually start working with it directly in Chapter 4, it's a good idea to take a closer look at what you've just created.

When you export your Blender scene using the SIO2 exporter script, the 3D data is saved as a file with the suffix `.sio2`. This is the file format that the SIO2 library reads. In fact, you can read it too. The `.sio2` file format is simply a zip file containing a directory tree with plain-text files to encode the 3D data. If you want to, you can uncompress the `.sio2` file yourself by simply changing the suffix to `.zip` and uncompressing the archiving in the same way you would uncompress any other ZIP file. This isn't usually necessary though, because by default the exporter writes the `.sio2` file and an uncompressed version of the data to the same directory. The name of the directory, like the name of the `.sio2` file, is the same as the name of the Blender scene that was exported. If you open this directory, you will see eight subdirectories: `camera`, `image`, `ipo`, `lamp`, `material`, `object`, `script`, and `sound`. These subdirectories contain the data you would expect. In each subdirectory are files corresponding to the 3D assets from the Blender scene. Each asset, such as a mesh object, is represented by a separate text file with the same name as the object itself. You can see the subdirectories of the exporter output in Figure 3.44. The top-level output was to the `template` directory introduced in Chapter 1. The `Hello3DWorld` directory is the uncompressed content of `Hello3DWorld.sio2`. The exported scene includes the Sphere object and the plane, which are found in the `object` directory. The files that represent the meshes themselves are plain-text files.

Figure 3.44
The contents of `Hello3DWorld.sio2`



The text of these files gives SIO2 the information it needs to build the assets in OpenGL ES. As an example, here are the contents of the `Camera` file created in the `camera` directory that was generated by the script:

```
camera( "camera/Camera")
{
        l( 0 -3.5 0 )
        r( 0.0 0 0 )
        d( 0 1 0 )
        f( 49.134 )
        cs( 0.1 )
        ce( 100 )
}
```

The syntax used here is similar to the C syntax, and the information is mostly straightforward, although some details are different from what you would find in Blender. The `l` attribute is directly from the corresponding location value in Blender. The `r` attribute is the rotation value, which is also directly from Blender. The `d` attribute, however, is the normalized direction vector of the camera, which is not an explicit Blender value but rather is calculated from the camera's position and rotation. Likewise, the `f` (field of view) represents the angle of view and is calculated from the Blender camera lens value. The lower the lens value, the wider the `f` value. The `cs` and `ce` values are the start and end clipping values, respectively, taken directly from the Blender values. These values determine the area within the field of view that the camera can see. Things farther away from the camera than the `cend` value are clipped from the scene and therefore not visible to the camera, and things closer than the `cstart` value are also clipped.

The kind of information contained in the SIO2 file about each type of 3D asset is specific to the type of object. The `Camera` file, therefore, contains information about the field of view and clipping values, whereas the file for a Mesh object contains information pertinent to meshes. You can see a simple example of this in the following code, which was generated to describe a simple UV-textured plane that has been divided into two triangles (the text to describe a cube is already fairly long to be displayed on the page of a book, so an example with fewer vertices is more suitable here):

```
object( "object/Plane" )
{
         l( 0 5.9 0 )
         r( 90.0 0 0 )
         s( 3.351 3.351 3.351 )
         ra( 4.739 )
         di( 1 1 0 )
         vb( 128 0 48 96 0 )
         v( 1 1 0 )
         v( -1 1 0 )
         v( -1 -1 0 )
         v( 1 -1 0 )
         n( 0 0 1 )
         n( 0 0 1 )
         n( 0 0 1 )
         n( 0 0 1 )
         u0( 0 1 )
         u0( 1 1 )
         u0( 1 0 )
         u0( 0 0 )
         ng( 1 )
         g( "null" )
         mt( "material/Material" )
         ni( 6 )
         i( 0 1 2 )
         i( 0 2 3 )

}
```

The first line, `l`, is the same as for other objects. The object's pivot point is located at the origin in Blender, so the values are `0`. The `ra` and `di` values are used in physics simulations to determine the collision boundaries of the object. The `vb` attribute determines the offset value for the OpenGL vertex buffer object that will hold the vertex data for this 3D object. (For more information about vertex buffer objects in OpenGL ES, see Appendix B.) The `v` attributes show the x, y, and z values for each vertex, and the `u0` values represent the UV mappings for each vertex for the first UV layer (two are possible). The `n` attributes represent the direction of the normals. These were exported because you had the N option selected in the exporter. The `ng` value is the number of vertex groups defined on the mesh, and the `g` value contains their names. Finally, the `i` value represents the number of indices necessary to assemble the full polygon mesh from the vertices, and the `ind` attributes list the indices in the order necessary to do that. Depending on the mesh and its features, other attributes may be present. In general, you will not need to work directly with this data because the SIO2 library provides an API for accessing the data in a convenient way. Nevertheless, it is good to have a basic idea of what information the data files contain and how they are formatted. The complete SIO2 file format specification is included in the SIO2 installation directory and is reprinted in Appendix C.

Now you've created and exported the scene, and you have a basic knowledge of what is in the files you'll be working with from here on. In the next chapter, you'll finally get down to putting these files to use to create content that you'll be able to run directly on your iPhone!

CHAPTER 4

Going Mobile with SIO2

In this chapter you'll use the SIO2 library to access and work with the 3D data you created in Chapter 3. You'll see how assets are loaded and how data is accessed through the `sio2` object. You'll also take the next step in creating interactive content, using touch input to interact directly with the 3D scene. By the end of the chapter you will have compiled and run your Hello 3D World application on your iPhone simulator or device, and you'll know all the basics of how to get your 3D content from Blender to your iPhone or iPod Touch screen.

- **Initializing SIO2 and loading assets**
- **Rendering**
- **Touch screen interaction**
- **The Complete Code**

Initializing SIO2 and Loading Assets

In Chapter 3 you created a 3D scene in Blender and exported it to an `.sio2` file using the SIO2 Python exporter. You also took a brief look at the contents of that file to get an idea of the kind of data that the SIO2 engine will have at its disposal to work with. Now, the scene you created will serve as the basis of an SIO2/OpenGL ES 3D environment for your iPhone.

To make this happen, first create a new Xcode project. In your `SIO2_SDK` directory, right-click the `template` directory and select Duplicate, as you did in Chapter 2. Rename the directory **project2**. To keep things organized, move the `Hello3DWorld.sio2` file that you created in Chapter 3 into this directory also. Fire up Xcode by double-clicking the `template.xcodeproj` file in this directory.

Headers and Prototypes

As discussed previously, most of your SIO2 programming will be focused on the `template.mm` file. There are some occasions where other files also need to be modified, and this is one of them. For this project, you're going to create a new function in the template that will handle one-time loading and initialization tasks to avoid a lot of heavy resource use inside the render loop. This function will be called `templateLoading` and it will be defined, like the other functions, in `template.mm`. However, like the other functions, it must be prototyped in the corresponding header, `template.h`.

Find `template.h` in the `Other Sources` directory in Groups & Files and click it to bring it up in the text editor. Below the comments, the file looks like this:

```
#ifndef TEMPLATE_H
#define TEMPLATE_H


void templateRender( void );
void templateShutdown( void );
void templateScreenTap( void *_ptr, unsigned char _state );
void templateScreenTouchMove( void *_ptr );
void templateScreenAccelerometer( void *_ptr );

#endif
```

The `#ifndef`, `#define`, and `#endif` preprocessor commands ensure that the content of this file is included only once. The code between the conditionals consists of function prototypes. You probably recognize the names of these functions from Chapter 2. These are the functions defined in `template.mm`. Add another line to this list someplace after `#define` and before `#endif` to prototype the `templateLoading` function, like this:

```
void templateLoading( void );
```

That's all that needs to be done in `template.h`, but there's still one more file that needs to be edited slightly. In the `Classes` directory in the Groups & Files list in Xcode, find the file `EAGLView.mm`. Line 129 of that file looks like this:

```
129        sio2->_SIO2window->_SIO2windowrender = templateRender;
```

This executes the `templateRender` function. However, because you will be adding a different function to handle the initialization, you will need to ensure that the initialization function is executed first. To do that, change the function called here to `templateLoading`, like this:

```
sio2->_SIO2window->_SIO2windowrender = templateLoading;
```

Later, within the code of `templateLoading` itself, you will assign `templateRender` back to the `SIO2windowrender` callback function and the rendering will continue as it was originally intended.

You can now turn your attention to `template.mm`. This of course is the default template source file that was explained in its entirety in Chapter 2. Recall that the `.mm` suffix indicates that the file will be interpreted by the compiler as potentially containing C, C++, or Objective C code. In general, `template.mm` will almost always contain C/C++ code, although you will see an example in Chapter 6 where a small amount of Objective-C code is also used in this file.

The *templateLoading* Function

Now that you've added the prototype for the `templateLoading` function, you'll define what the function actually does. In `template.mm`, add the function definition after the end of the `templateRender` function definition. For now, you can just add an empty definition, like this:

```
void templateLoading ( void )
{
}
```

The remaining code in this section will go between these two curly brackets.

The purpose of the `templateLoading` function is to load assets from the `.sio2` file into the SIO2 iPhone programming environment. This entails converting the `.sio2` file assets into OpenGL ES v1.1 content, although this is somewhat behind the scenes. SIO2 does this by associating an `.sio2` file with an `SIO2resource` object, which in turn is associated with the top-level `sio2` object.

The main `SIO2resource` structure will hold the contents of the `.sio2` file in a dictionary data structure so that they can be accessed and used to populate other structures. To initialize this dictionary structure, use the `sio2ResourceCreateDictionary` function, with the `SIO2resource` structure belonging to the `sio2` structure as the function's argument, like this:

```
    sio2ResourceCreateDictionary( sio2->_SIO2resource );
```

`void` **`sio2ResourceCreateDictionary`**`( SIO2resource *`*`resource`*` )`

This function initializes the dictionary structure of the `SIO2resource` object *resource* with the keys `object`, `material`, `lamp`, `camera`, `ipo`, and `action`.

You then need to open up the appropriate `.sio2` file and associate it with the `SIO2resource` object. This is done as follows:

```
        sio2ResourceOpen( sio2->_SIO2resource, "Hello3DWorld.sio2", 1);
```

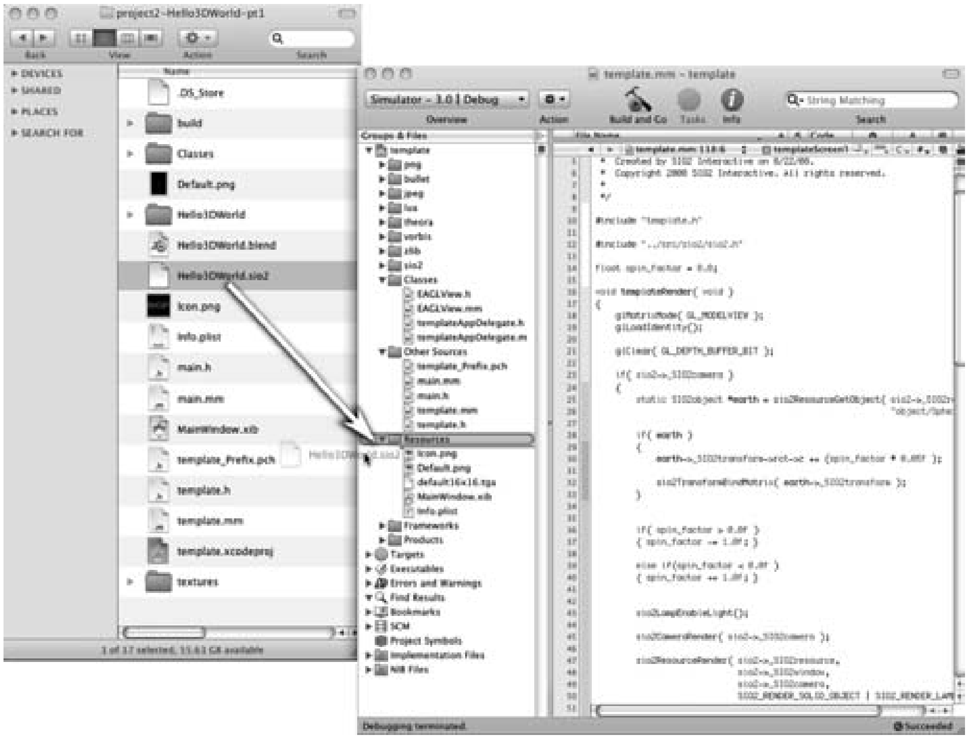
`void` **`sio2ResourceOpen`**`( SIO2resource *`*`resource`*`, const char *`*`fname`*`, unsigned char `*`rel`*` )`

This function opens the file named *`fname`* and associates it with `SIO2resource` *`resource`*. The *`rel`* value determines whether the relative path is used. If *`rel`* is nonzero, it is assumed the file is located in the project directory; otherwise an absolute path must be given.

Note that `Hello3DWorld.sio2` is the name of the `.sio2` file you exported from Blender in Chapter 3. At the beginning of this chapter, you placed the file in the same directory with the rest of the project files, but this is not quite enough to make the file accessible to the project in Xcode. Any files used by an iPhone application must be placed into the project itself from within Xcode. Do this now by dragging the icon for the file `Hello3DWorld.sio2` from the Finder into the Groups & Files area in Xcode, as shown in Figure 4.1. This is very important. If you try to run a program that accesses files that haven't been placed into the project in this way, you will get an error. When you have placed the file in Groups & Files, you'll see the dialog shown in Figure 4.2. Check the box next to "Copy items into destination group's folder (if needed)."

Figure 4.1

Dragging the `.sio2` file from the Finder to Groups & Files

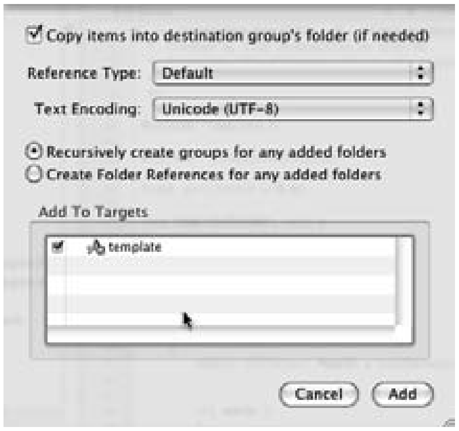


```
 *  Created by SIO2 Interactive on 8/22/08.
 *  Copyright 2008 SIO2 Interactive. All rights reserved.
 *
 */

#include "template.h"

#include "../src/sio2/sio2.h"

float spin_factor = 0.0;

void templateRender( void )
{
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    glClear( GL_DEPTH_BUFFER_BIT );

    if( sio2->_SIO2camera )
    {
        static SIO2object *earth = sio2ResourceGetObject( sio2->_SIO2r
                                                          "object/Sphe

        if( earth )
        {
            earth->_SIO2transform->rot->z += (spin_factor * 0.05f );

            sio2TransformBindMatrix( earth->_SIO2transform );
        }


        if( spin_factor > 0.0f )
        { spin_factor -= 1.0f; }

        else if(spin_factor < 0.0f )
        { spin_factor += 1.0f; }


        sio2LampEnableLight();

        sio2CameraRender( sio2->_SIO2camera );

        sio2ResourceRender( sio2->_SIO2resource,
                            sio2->_SIO2window,
                            sio2->_SIO2camera,
                            SIO2_RENDER_SOLID_OBJECT | SIO2_RENDER_LAM
```

The next step is to extract the contents of the SIO2resource object.

```
        unsigned int i = 0;
        while( i != sio2->_SIO2resource->gi.number_entry )
        {
              sio2ResourceExtract( sio2->_SIO2resource, NULL
    );
              ++i;
        }
```

Recall that the sio2 file is basically a ZIP file, so its contents need to be uncompressed before use. The details of this block are not really important to understand completely, but you should know that it's a necessary step for putting the content of the SIO2resource into usable form.



☑ Copy items into destination group's folder (if needed)

Reference Type: Default

Text Encoding: Unicode (UTF-8)

◉ Recursively create groups for any added folders

○ Create Folder References for any added folders

Add To Targets

☑ template

Figure 4.2

Dialog for adding items to Groups & Files

```
void sio2ResourceExtract( SIO2resource *resource, char *password )
```

This function extracts the archived contents of the SIO2resource *resource*. The *password* value is NULL by default.

Now that the content of the .sio2 file has been extracted, you can close the file. Do that with the next line of code:

```
        sio2ResourceClose( sio2->_SIO2resource );
```

```
void sio2ResourceClose( SIO2resource *resource )
```

This function closes the .sio2 file associated with the SIO2resource object *resource*.

Now you need to bind the resources. In OpenGL terminology, to "bind" an asset or property means to set it to be used in subsequent function calls. This is what you are doing here. Binding the images, materials, and matrices makes them active as the program runs. This is done with the following lines of code:

```
        sio2ResourceBindAllImages( sio2->_SIO2resource );
        sio2ResourceBindAllMaterials( sio2->_SIO2resource );
        sio2ResourceBindAllMatrix( sio2->_SIO2resource );
```

```
void sio2ResourceBindAllImages( SIO2resource *resource )
void sio2ResourceBindAllMaterials( SIO2resource *resource )
void sio2ResourceBindAllMatrix( SIO2resource *resource )
```

These functions set the images, materials, and matrices associated with the SIO2resource *resource* to be affected by relevant subsequent function calls.

The next line generates OpenGL ES ID values of the appropriate type for all of the assets in the `SIO2resource` object (ID generation is discussed in Appendix B):

```
sio2ResourceGenId( sio2->_SIO2resource );
```

The next two lines are some basic initialization code for the `sio2` root object. The first of the two lines resets the state of the `sio2` root object. This involves setting the `sio2` root object's 3D property objects to null values. This does not change the `sio2->_SIO2window` value. The next line assigns the content of the `templateRender` function to the actual render function, which itself is a property of the `SIO2window` object (recall how you changed a similar line in the original template's `EAGLView.mm` file):

```
sio2ResetState();
sio2->_SIO2window->_SIO2windowrender = templateRender;
```

void **sio2ResetState**(void)

This function sets the `SIO2camera`, `SIO2object`, `SIO2lamp`, `SIO2vertexgroup`, `SIO2material`, `SIO2font`, `SIO2ipo`, `SIO2ipocurve`, `SIO2action`, and all `SIO2image` structures associated with the `sio2` root object to `NULL`.

The camera asset from `SIO2resource` is assigned to the `SIO2camera` value of the `sio2` root object like this:

```
sio2->_SIO2camera = sio2ResourceGetCamera( sio2->_SIO2resource,
                                                       "camera/Camera");
```

Now the project has a camera. With the camera information from the `.sio2` file and the scale information from `sio2->_SIO2window`, you can build the perspective view volume (as described in Appendix B) using the `sio2Perspective` function. This function was not called in the example in Chapter 2 because 3D rendering was not used in that example. To adjust the projection matrix, the function needs the camera's field of view (`sio2->_SIO2camera->fov`), which represents the camera's lens angle measured in degrees, the aspect ratio of the window (calculated by dividing the x scale of the window by the y scale: `sio2->_SIO2window->scl->x / sio2->_SIO2window->scl->y`), and minimum and maximum clip values (`cstart` and `cend`, respectively).

```
sio2Perspective( sio2->_SIO2camera->fov,
                 sio2->_SIO2window->scl->x / sio2->_SIO2window->scl->y,
                 sio2->_SIO2camera->cstart,
                 sio2->_SIO2camera->cend );
```

void **sio2Perspective**(”oat fovy, ”oat *aspect*, ”oat *zNear*, ”oat *zFar*)

This function adjusts the projection matrix with size and angles determined by the arguments to the function. The *fovy* argument is the camera's field of view. The *aspect* argument is the width over height ratio of the view window. The *zNear* and *zFar* values are the front and back clipping panes, respectively; that is, the nearest and farthest z-axis values to be rendered.

This wraps up the loading and initialization portion of the program. You can now turn your attention to setting up the render loop.

Rendering

Now it's time to write the render loop, which is the code that will be executed for each frame as your application runs. As you should recall from Chapter 2, this code goes in the function `templateRender`. The default template implementation of that function looks like this:

```
void templateRender( void )
{
     glMatrixMode( GL_MODELVIEW );
     glLoadIdentity();
     glClear( GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT );
     //Your rendering code here...
}
```

For the most part, this section of the function will remain unchanged, except for one small thing. If you remember the scene you created in Blender, you'll recall that the background of the scene is a solid, opaque textured plane. Each frame that renders will completely overwrite all pixels in the window. Because of this, there's no need to clear the color buffer for each frame. On mobile platforms such as the iPhone, processing resources are at a premium, and it's best to eliminate all unnecessary processing steps. To stop the color buffer from being cleared each frame, delete the `GL_COLOR_BUFFER_BIT` label from the `glClear` function call, changing the line to this:

```
     glClear( GL_DEPTH_BUFFER_BIT );
```

The depth buffer still needs to be cleared to properly calculate hidden face removal when faces move in relation to each other, so you can't remove this line entirely.

The remaining render code should execute only after the `sio2` root object has been populated with a camera, which happens in `templateLoading`, as you saw previously. To ensure this has happened, use a conditional statement to check that the `sio2->_SIO2camera` structure is pointing to an existing handle, like this:

```
     if( sio2->_SIO2camera )
     {
     }
```

The rest of the rendering code should all be placed between the two curly brackets of that conditional. Because you'll eventually be using real-time lighting, you need to declare a variable to represent the color of the ambient light. You want a weak uniform light for this, so declare the variable `ambient_color` like this:

```
          static vec4 ambient_color = { 0.1f, 0.1f, 0.1f, 1.0f };
```

Making the World Go 'Round

To access a 3D object for the purpose of transformations, you have to assign the appropriate resource to an `SIO2object` structure. In this example, the model of the planet Earth will rotate, so you must create a variable of the type `SIO2object` and use `sio2ResourceGetObject` to assign the `Sphere` object from the Blender scene to the variable. I've chosen the name *`earth`* for the variable name. Assign it the appropriate value like this:

```
            static SIO2object *earth = sio2ResourceGetObject( sio2->_
      SIO2resource,

      "object/Sphere" );
```

The next block of code will rotate the world around the z-axis:

```
if( earth )
 {
   earth->_SIO2transform->rot->z += 15.0f*sio2->_SIO2window->d_time;
   sio2TransformBindMatrix( earth->_SIO2transform );
 }
```

First check to make sure the object is pointing to a valid handle to avoid errors that can arise from trying to access properties of uninstantiated objects. Objects of type `SIO2object` have `_SIO2transform` properties, which in turn have rotation values for each axis. The x-axis rotation value for this object is accessed with *`earth->_SIO2transform->rot->z`*. The first line in the conditional statement increases this value with each frame. You should never use a hard-coded value for animation on the iPhone. Using the `sio2->_SIO2window->d_time` as a factor for animation values makes them independent of the frame rate because it associates them directly with the delta time value of the window.

The next line binds the transformation matrix to ensure that the transformation is actually applied at render time.

`void` **`sio2TransformBindMatrix`**`( SIO2transform *`*`transform`*` )`

This function sets the transform function to be calculated for use in subsequent functions in the program. This function recalculates the new matrix based on the location, rotation, and scale transformations in the corresponding `SIO2transform` structure. It is necessary to call this before rendering an object that has been operated on with an `SIO2transform`.

On the following line, add a comment to indicate where to include the interactive spin factor code:

```
//spin factor code
```

This will be discussed in the next section, on touch screen interactivity, and you'll want to come back to this point in the program to include the pertinent code.

Before you code in the touch screen interactivity, you'll finish the basic render code so that you can run the application and see some preliminary 3D results. To do this, the only thing missing now is light.

And Then There Was Light

The last few lines of the render code enable lighting and then complete the rendering process. Set the ambient light value and then enable real-time lighting with the following function calls:

```
sio2LampSetAmbient( &ambient_color );;
sio2LampEnableLight();
```

void **sio2LampSetAmbient**(vec4 color)

This function sets the ambient lighting to the color described by the *color* variable. Assigning full white color will yield maximum ambient lighting.

void **sio2LampEnableLight**(void)

This function enables the OpenGL ES states required for lighting to be calculated.

The next few lines of code call the `sio2CameraRender` function, which calculates the placement of the camera, and the `sio2ResourceRender` function, which renders the resources indicated by the bit mask in the fourth argument. In this case, `SIO2_RENDER_SOLID_OBJECT` and `SIO2_RENDER_LAMP` tell the SIO2 engine to render solid 3D objects and lights:

```
sio2CameraRender( sio2->_SIO2camera );
sio2ResourceRender( sio2->_SIO2resource,
                    sio2->_SIO2window,
                    sio2->_SIO2camera,
                   SIO2_RENDER_SOLID_OBJECT | SIO2_RENDER_LAMP);
```

void **sio2CameraRender**( )

This function calculates the placement of the camera.

```
void sio2ResourceRender( SIO2resource *resource, SIO2window *window, SIO2camera
*camera, int mask  )
```

This function renders the resources associated with `SIO2resource` *resource* to the `SIO2 window` *window*. The camera information is passed via the *camera* parameter. The types of resources rendered depend on the value or values of the bit mask *mask*. The possible values are `SIO2_RENDER_IPO`, `SIO2_RENDER_LAMP`, `SIO2_RENDER_SOLID_OBJECT`, `SIO2_RENDER_TRANSPARENT_OBJECT`, `SIO2_RENDER_ALPHA_TESTED_OBJECT`, `SIO2_RENDER_CLIPPED_OBJECT`, `SIO2_RENDER_EMITTER`, `SIO2_EVALUATE_SENSOR`, `SIO2_EVALUATE_TIMER`, and `SIO2_UPDATE_SOUND_STREAM`. Multiple *mask* values can be entered, separated by the vertical pipe symbol, representing Boolean disjunction (or).

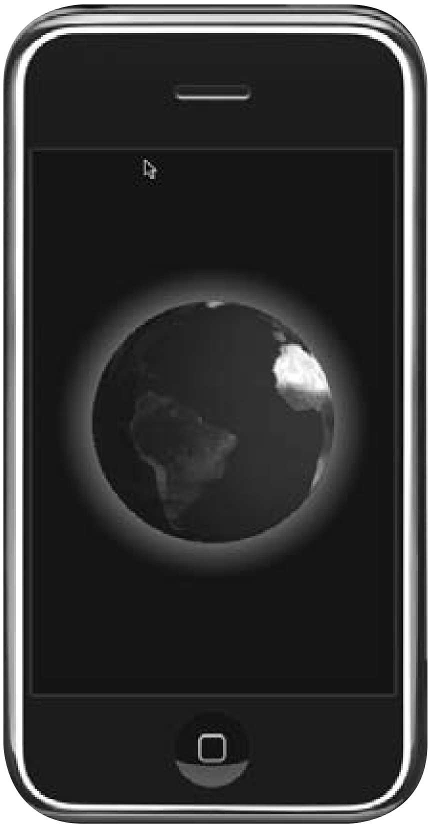


Figure 4.3
Running the application on the simulator

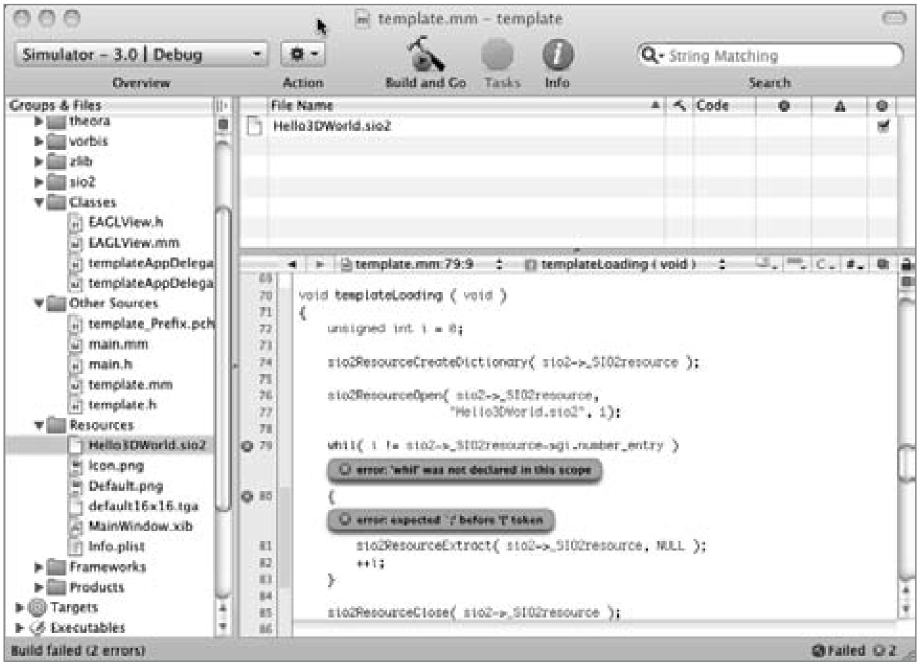
At this point you can build and run the application. You should see the planet Earth appear on your screen with a black background and a white glow, as shown in Figure 4.3, just as you saw in your Blender render. You should also see that the planet is rotating slowly around the z-axis.

It's good to be aware that real-time lighting in the iPhone is extremely demanding on the graphics processing unit and can have a serious effect on performance. The SIO2 engine allows a maximum of eight lights to be set up, but you should normally not try to light a whole scene using GL lighting. In most cases, real-time lighting is not necessary, as you will see in Chapter 6 when you learn to bake lighting patterns to textures in Blender. Doing that can enable you to get well-lit scenes in your iPhone with no real-time lighting at all.

Troubleshooting

As is always the case with programming, there are a virtually infinite number of places where small things can go wrong and cause problems for successfully executing your code. If you have managed to get this far in the book without errors, my hat is off to you! If you do run into problems at this point, here are some general tips for tracking them down.

Syntax errors are the most common problems you'll have. You've forgotten a semicolon somewhere or mistyped a keyword. Xcode's code completion and syntax highlighting is a great help in preventing many potential syntax errors, but errors get past it. In these cases, the application will usually not compile successfully. If there are errors compiling, Xcode will highlight the line where the error occurred with a red icon with a little white *x* in it, as shown in Figure 4.4. Always focus your attention on the first error that comes up because subsequent errors are often a result of the first one.



```
void templateLoading ( void )
{
    unsigned int i = 0;

    sio2ResourceCreateDictionary( sio2->_SIO2resource );

    sio2ResourceOpen( sio2->_SIO2resource,
                      "Hello3DWorld.sio2", 1);

    whil( i != sio2->_SIO2resource->gi.number_entry )
error: 'whil' was not declared in this scope
    {
error: expected ';' before '{' token
        sio2ResourceExtract( sio2->_SIO2resource, NULL );
        ++i;
    }

    sio2ResourceClose( sio2->_SIO2resource );
```

Figure 4.4
Syntax error highlighting

There are more insidious bugs that can occur at runtime, after a successful compile. For an inexperienced C/C++/Objective-C programmer, collecting diagnostic information from the debugger can be a challenge, and it's beyond the scope of this book to get into hard-core C debugging. In any case, if you're sticking with code from this book for the time being, this shouldn't be necessary. If you're having runtime crashes, a very likely cause of the problem is that data files are missing from the project's resources. Any files accessed by the code *must* be included in the Groups & Files area. Check the `Resources` directory here to make sure everything is in place. Double-check the filenames to make sure they are the same as the ones you're calling in the code.

If your screen is simply black when you run the application, it's likely the problem has to do with either the `.blend` file or the `.sio2` file you exported from it. Check the `.blend` file again to make sure the normals are pointed outward and all the lamps and objects you want to render are on a selected layer (see Appendix A if you're not sure how to do these things). Try reexporting the `.sio2` file to the project directory, being careful to have all the objects that you want to export selected in the Blender 3D view. Objects that are not selected will not be exported!

When you're reexporting, the export directory will be removed and re-created if the UP toggle button is not toggled. However, note that if you are updating an export with UP toggled and you change an object's name, the previously exported object information will not be removed, so if you ever need to delete or rename exported objects, delete the previous `.sio2` file and the corresponding directory. This is not always strictly necessary, but sometimes it is, and it's important to know that not deleting a previous `.sio2` file and directory can be the source of some problems. If you are having persistent problems, you should also try deleting the `build` directory that the compiler places in your project directory. This will force a completely fresh build.

You can run your program with the debugger by starting the application with Run → Debug.

Touch Screen Interaction

Now that you've run the application and seen the 3D content in action in your iPhone/iPod Touch or on your iPhone simulator, it's time to add some interactive functionality and take advantage of what makes the iPhone platform so exciting in the first place!

Spin Control

The goal here will be to control the spin of the planet using a swipe of your finger on the touch screen rather than simply watching it go round and round. To do this, you'll modify the speed it spins in an interactive way. This will require a variable. You'll use a floating-point number for this, and the variable will be called *`spin_factor`*. Declare it at the beginning of the program, before the definition of `templateRender`, like this:

```
float spin_factor = 0.0;
```

This variable is going to be used as a multiplier for the change in rotation. In `templateRender`, find the line where the change in rotation is determined:

```
earth->_SIO2transform->rot->z += 15.0f * sio2->_SIO2window->d_time;
```

Then change the line to include the spin factor value, like this:

```
earth->_SIO2transform->rot->z += (spin_factor * sio2->_SIO2window->d_time );
```

If you build and run the application now, the globe will be completely still because *`spin_factor`* is initialized as zero and has not been changed. You'll deal with giving the variable a nonzero value shortly. But first, there are a few more lines to add to `templateRender`. In the case where *`spin_factor`* is not zero, you want to gradually bring it back to zero so that when you spin the planet it eventually slows down and stops spinning, just as a real globe would do. The following code does this by checking to see if the globe is spinning in a positive or negative direction and, if it is, slowing it down incrementally. Place this code at the same point where you inserted the `//spin factor code` comment in the previous section:

```
 if( spin_factor > 0.0f )
     { spin_factor -= 1.0f; }
else if(spin_factor < 0.0f )
     { spin_factor += 1.0f; }
```

Handling the Screen Tap and Touch Move

The `templateScreenTap` and `templateScreenTouchMove` functions are where the touch screen magic really happens. But before you get into defining the functions themselves, you need to declare another variable. This variable will be used inside the functions, but it must be defined outside of the functions so that it will be global and its value will be available across function calls. You can declare the variable anywhere in the code prior to the function definitions, but to minimize confusion, it's a good idea to declare it close to the functions where it will be used. For that reason, declare the variable right above the definition of `templateScreenTap`. The variable is of type `vec2`, which represents a two-dimensional vector. Call the variable *`start`* and declare it like this:

```
vec2 start;
```

The next thing to do is to fill in the definition for `templateScreenTap`:

```
void templateScreenTap( void *_ptr, unsigned char _state )
{
     if( _state == SIO2_WINDOW_TAP_DOWN ){
          start.x = sio2->_SIO2window->touch[ 0 ]->x;
          start.y = sio2->_SIO2window->touch[ 0 ]->y;
     }
}
```

All that needs to be done here is to register the x and y positions of the point where your finger makes contact with the screen. That way, when you swipe your finger, SIO2 can calculate which direction the swipe went. Because you're only interested in the tap down case, include a conditional to ensure that the state of the tap is `SIO2_WINDOW_TAP_DOWN`. Then assign the *x* and *y* values of the 0 touch (the first finger to hit the screen) to the *x* and *y* values of the *`start`* vector.

The `templateScreenTouchMove` will finally give *`spin_factor`* its nonzero value:

```
void templateScreenTouchMove( void *_ptr )
{
     if( sio2->_SIO2window->n_touch )
     {
          spin_factor = 2*(sio2->_SIO2window->touch[ 0 ]->x - start.x);
      }
}
```

This is calculated depending on the difference between the current position of your finger and the start position so that a longer swipe will increase the speed of the spin more than a short swipe. This gives a pretty good approximation of the kind of behavior you'd expect when spinning an object with your finger.

You won't be using the accelerometer in this example, so the default empty function definition of `templateScreenAccelerometer` remains unchanged. You can build and run your application now and you'll see that you can spin the globe by swiping your finger to the right or left.

The Complete Code

Here is the complete code for `template.mm`:

```
/*
 *  template.mm
 *  template
 *
 *  Created by SIO2 Interactive on 8/22/08.
 *  Copyright 2008 SIO2 Interactive. All rights reserved.
 *
 */

#include "template.h"

#include "../src/sio2/sio2.h"

float spin_factor = 0.0;
void templateRender( void )
{
     glMatrixMode( GL_MODELVIEW );
     glLoadIdentity();

     glClear( GL_DEPTH_BUFFER_BIT );

     if( sio2->_SIO2camera )
     {
          static vec4 ambient_color = { 0.1f, 0.1f, 0.1f, 1.0f };
          static SIO2object *earth =
                        sio2ResourceGetObject( sio2->_SIO2resource,
                        "object/Sphere" );

          if( earth )
          {
                earth->_SIO2transform->rot->z +=
                        (spin_factor * sio2->_SIO2window->d_time );
                sio2TransformBindMatrix( earth->_SIO2transform );
          }

           if( spin_factor > 0.0f )
               { spin_factor -= 1.0f; }
          else if(spin_factor < 0.0f )
               { spin_factor += 1.0f; }

           sio2LampSetAmbient( &ambient_color );
           sio2LampEnableLight();
          sio2CameraRender( sio2->_SIO2camera );
          sio2ResourceRender( sio2->_SIO2resource,
                              sio2->_SIO2window,
                              sio2->_SIO2camera,
                              SIO2_RENDER_SOLID_OBJECT |
                              SIO2_RENDER_LAMP);

     }
}

void templateShutdown( void )
{
     // Clean up
     sio2ResourceUnloadAll( sio2->_SIO2resource );
     sio2->_SIO2resource = sio2ResourceFree( sio2->_SIO2resource );
     sio2->_SIO2window = sio2WindowFree( sio2->_SIO2window );
     sio2 = sio2Shutdown();
     printf("\nSIO2: shutdown...\n" );
}

void templateLoading ( void )
{
     unsigned int i = 0;
     sio2ResourceCreateDictionary( sio2->_SIO2resource );
     sio2ResourceOpen( sio2->_SIO2resource,
                       "Hello3DWorld.sio2", 1);

     while( i != sio2->_SIO2resource->gi.number_entry )
     {
           sio2ResourceExtract( sio2->_SIO2resource, NULL );
           ++i;
     }

     sio2ResourceClose( sio2->_SIO2resource );
     sio2ResourceBindAllImages( sio2->_SIO2resource );
     sio2ResourceBindAllMaterials( sio2->_SIO2resource );
     sio2ResourceBindAllMatrix( sio2->_SIO2resource );
     sio2ResourceGenId( sio2->_SIO2resource );
     sio2ResetState();

     sio2->_SIO2camera = sio2ResourceGetCamera( sio2->_SIO2resource,
                                                  "camera/Camera");
     sio2Perspective( sio2->_SIO2camera->fov,
                      sio2->_SIO2window->scl->x /
                              sio2->_SIO2window->scl->y,
                      sio2->_SIO2camera->cstart,
                      sio2->_SIO2camera->cend );
     sio2->_SIO2window->_SIO2windowrender = templateRender;
}

vec2 start;

void templateScreenTap( void *_ptr, unsigned char _state )
{
     if( _state == SIO2_WINDOW_TAP_DOWN ){
          start.x = sio2->_SIO2window->touch[ 0 ]->x;
          start.y = sio2->_SIO2window->touch[ 0 ]->y;
     }
}

void templateScreenTouchMove( void *_ptr )
{
     if( sio2->_SIO2window->n_touch )
     {
          spin_factor = 2*(sio2->_SIO2window->touch[ 0 ]->x - start.x);
      }
}

void templateScreenAccelerometer( void *_ptr )
{
}
```

Extending Interactive Feedback with Picking and Text

In this chapter, you'll pick up from where you left off with the previous chapter and explore several more ways of adding interactivity to your application. You'll begin by taking an in-depth look at how SIO2 enables you to easily implement 3D picking so the user can select objects simply by touching the screen. After you've seen how to incorporate picking into your Hello3DWorld application, you'll learn how to implement text-based feedback on the iPhone screen. Finally, you'll see how to package your application with its own name and icon.

- **Object picking**
- **Working with text and fonts**
- **Using multi-touch functionality**
- **Packaging your app**
- **The Complete Code**

Object Picking

The word *picking* refers to identifying a point in a virtual 3D environment based on a 2D point indicated by the user. With picking, a user can select an object in the 3D world with a mouse click or a touch screen tap. This is a crucial function for many 3D games and interactive environments because it grants the user a high degree of interactivity with the virtual world.

There is more than one way to do picking. You can find several common approaches using OpenGL functions in OpenGL literature online and in books such as the *OpenGL SuperBible* by Richard S. Wright and Benjamin Lipchak (3rd edition, Sams, 2004) and *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1* (often referred to as "The Red Book") by Dave Shreiner and the Khronos OpenGL ARB Working Group (7th edition, Addison-Wesley Professional, 2009). In SIO2 specifically, there are also several ways to do picking. One method that I will not discuss in depth in this book involves using the Bullet physics library to calculate a ray from the view plane to the 3D object being picked. You can learn about this method by studying Tutorial 6.1 in the SIO2 package. The method I cover in this chapter follows the approach taken in Tutorial 6.2 and is both computationally less expensive and a little bit easier to understand than the physics-based method.

The method of picking here works by first assigning each object a unique solid color and then rendering a frame to the OpenGL color buffer with the objects in these colors, without lighting or materials. When the frame has been rendered, an OpenGL function is called to return the color value of the pixel that has been tapped. Because each object has been rendered in a solid, unique color, the color of the pixel is all that's needed to pick the correct object.

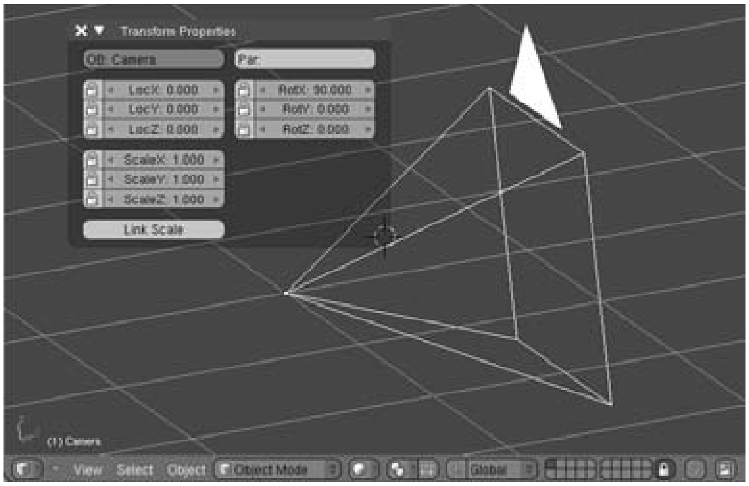
A Simple Picking Example in Blender

Before you apply picking to your Hello3DWorld application, it's a good idea to look at a more stripped-down example so you can easily see exactly how the picking code is working. This example will give you a little bit more practice creating 3D scenes for SIO2 and will show you how to dig deeper into the SIO2 code to find out how things work on your own. Being able to quickly navigate from function calls to function definitions is a fundamental skill in Xcode that you'll find comes in very handy when debugging.

To begin, create a new project by duplicating and renaming the `template` directory, just as you have done in previous chapters. Give it the name **`picking`**. Now fire up Blender and save the `.blend` file in the project directory. Follow these steps to create a simple scene for use in this project:

1. Select the cube and the lamp objects and delete them both by pressing the X key. For information about navigating the 3D space and selecting objects, refer to Appendix A.

2. Select the camera and press Alt+G to clear the location. Press the N key over the 3D viewport to bring up the Transform Properties floating panel (Figure 5.1). Input the rotation values RotX: **90**, RotY: **0**, and RotZ: **0** as shown in the figure.



OB: Camera | Par:

LocX: 0.000 | RotX: 90.000
LocY: 0.000 | RotY: 0.000
LocZ: 0.000 | RotZ: 0.000

ScaleX: 1.000
ScaleY: 1.000
ScaleZ: 1.000

Link Scale

Figure 5.1

Setting the location and rotation of the camera

3. Press the spacebar and choose Add → Mesh → Plane to add a plane, as shown in Figure 5.2. If you are using default settings, the plane should appear as shown in Figure 5.3, at the origin of the space facing upward. If it looks different from this, tab into Object mode and press Alt+G and Alt+R to clear the rotation and location.

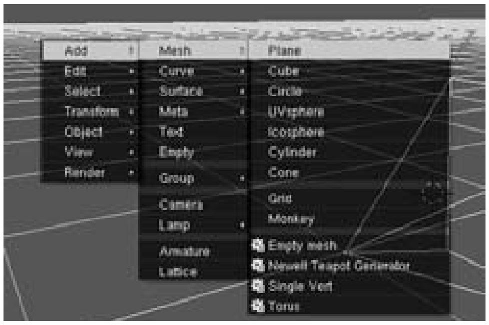


Figure 5.2

Adding a plane

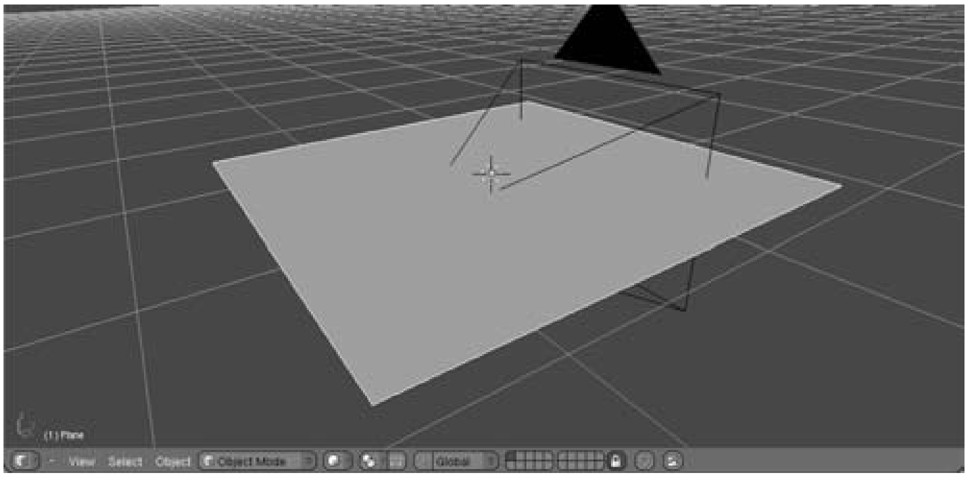
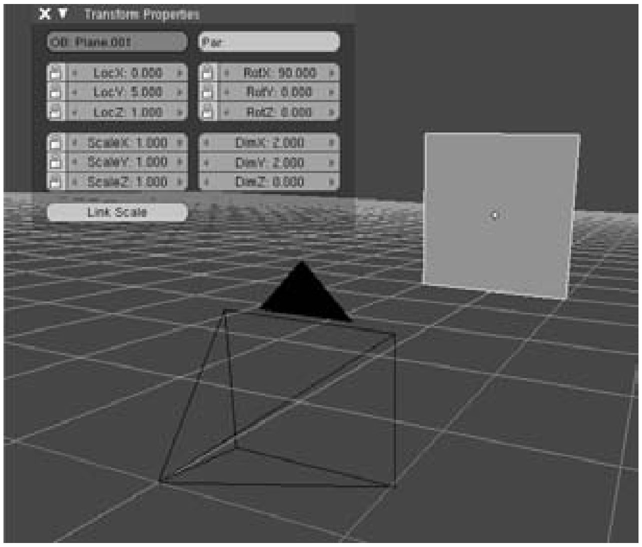


Figure 5.3

The resulting plane

4. Look at the Transform Properties panel (N key) and enter the values LocX: **0**, LocY: **5**, LocZ: **1**, RotX: **90**, RotY: **0**, and RotZ: **0** for the location and rotation of the plane, as shown in Figure 5.4.



Transform Properties

OB: Plane.001 | Par

LocX: 0.000 | RotX: 90.000
LocY: 5.000 | RotY: 0.000
LocZ: 1.000 | RotZ: 0.000

ScaleX: 1.000 | DimX: 2.000
ScaleY: 1.000 | DimY: 2.000
ScaleZ: 1.000 | DimZ: 0.000

Link Scale

Figure 5.4
Setting location and rotation for the plane

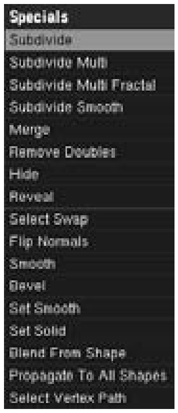


Figure 5.5
Subdividing

5. Tab into Edit mode. Make sure the mesh is entirely selected using the A key. Press the W key and select Subdivide from the menu, as shown in Figure 5.5. Scale the plane vertically down slightly by pressing the S key followed by the Z key (Figure 5.6).

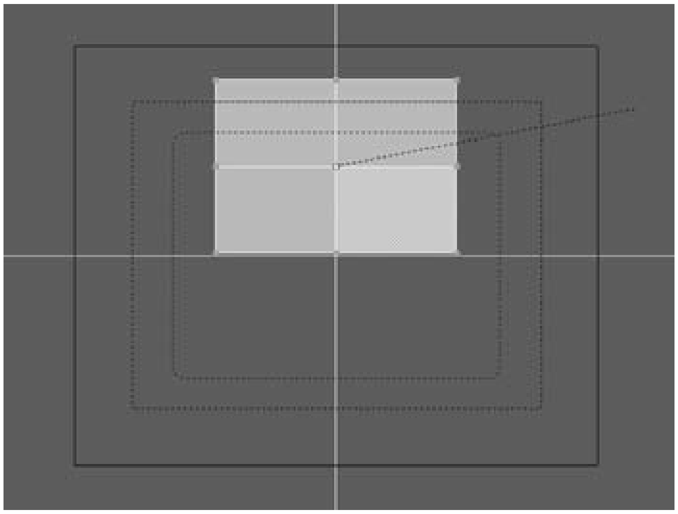


Figure 5.6
Scaling vertically

6. From the Mesh menu, execute the script for converting the quads to triangles: Mesh → Faces → Convert Quads To Triangles, as shown in Figure 5.7. The resulting

triangle mesh will look like they do in Figure 5.8, but this isn't quite what you want. The triangles in the upper-left and lower-right quad should be flipped. Select the lower-right quad and execute the flip triangle edges script by choosing Mesh → Faces → Flip Triangle Edges, as shown in Figure 5.9. Do the same with the upper-left quad to get the topology shown in Figure 5.10.

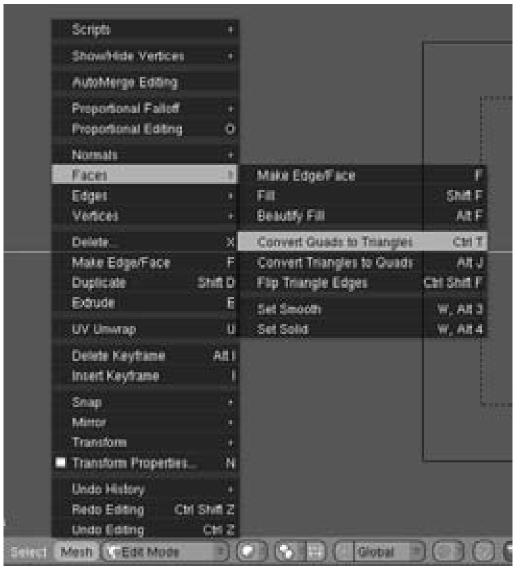


Figure 5.7
Converting quads to triangles

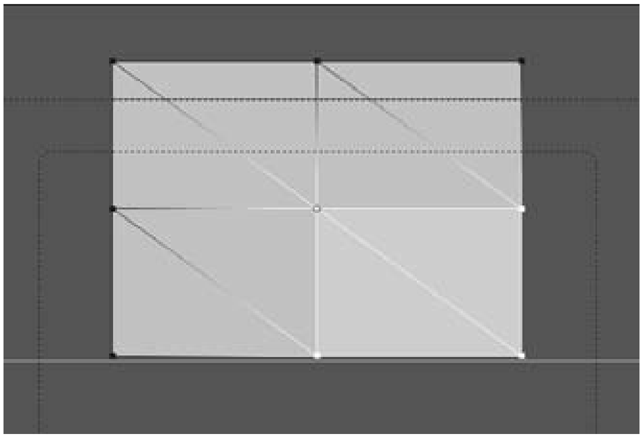


Figure 5.8
Resulting triangles

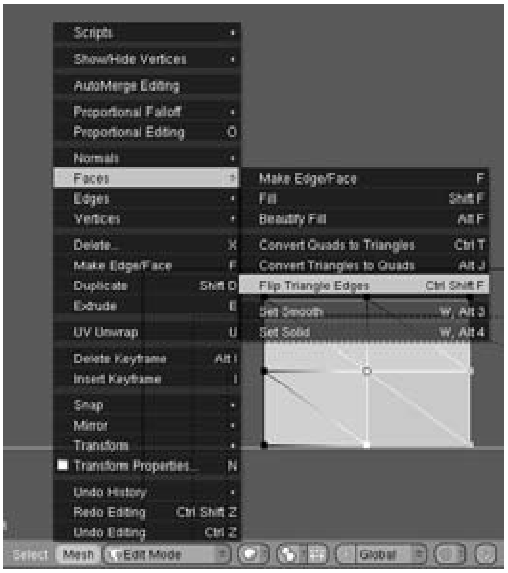


Figure 5.9
Flipping triangles

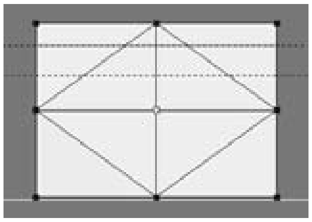


Figure 5.10
The finished triangle mesh

7. Tab into Object mode and duplicate the mesh by pressing Ctrl+D and then pressing Z to constrain the new object to move vertically. Move the second plane down below the first one, as shown in Figure 5.11.
8. Rename the new mesh object **Plane2**. Enter the new name in the OB field in the Link And Materials panel, as shown in Figure 5.12. This is important. By default, duplicated objects in Blender are given names ending with numbers, starting with .001. These numerical suffixes are used by SIO2 as the basis of *pseudo-instancing*, which you will read more about in Chapter 7. All you need to know now is that object names ending with such automatically generated suffixes will be basically ignored by SIO2 in the present example. So renaming this object is a requirement.

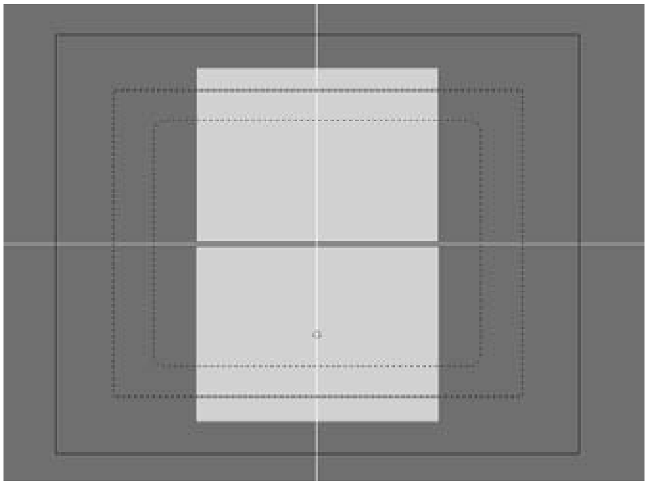


Figure 5.12

Renaming the object

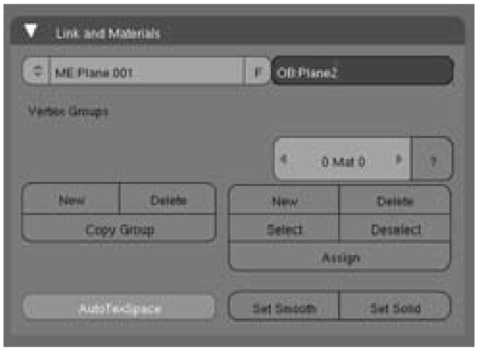


Figure 5.11

Duplicating the mesh

9. Now it's time to add some color to the scene. In this simple example, you won't need any materials or textures. Instead, you will use the simplest way to color a mesh that SIO2 can understand, namely vertex painting. Enter Vertex Paint mode with the header drop-down menu as shown in Figure 5.13. Adjust the vertex paint color and opacity using the panel shown in Figure 5.14. Paint each vertex of each plane a different color. Stick with primary colors like red (two RGB values set at 0 and one value set at 1) and secondary colors like yellow (two RGB values set at 1 and one value set at 0) as well as black (all three RGB values set at 0) and white (all three RGB values set at 1). You will be able to see the vertex colors in Textured display mode. The resulting scene, from the camera view, is shown in Figure 5.15 (unfortunately reprinted in grayscale, so you'll have to use your imagination with regard to the colors).

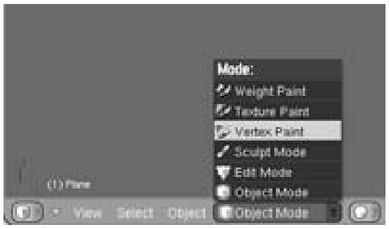


Figure 5.14
The Paint panel

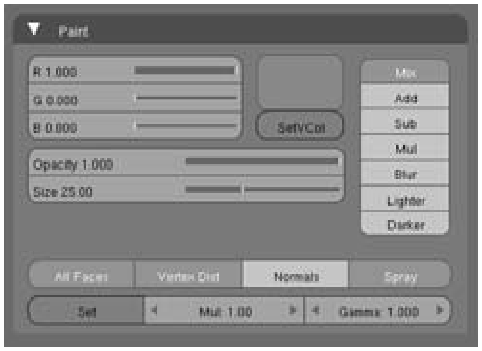


Figure 5.13
Entering Vertex Paint mode

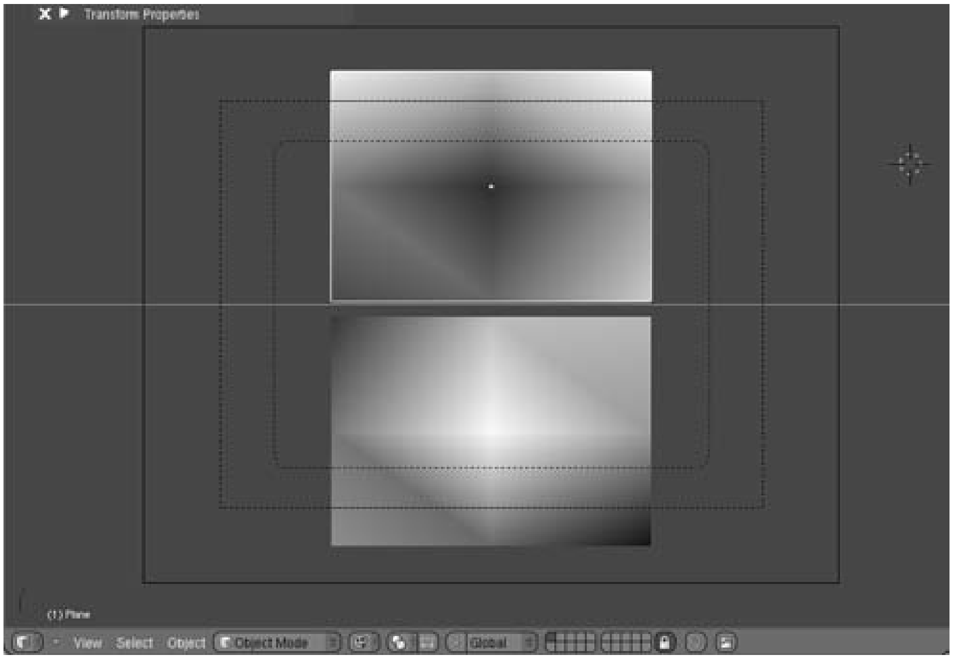


Figure 5.15
The finished scene from the camera view, in Textured display mode

10. Change the scene name to **PickingScene** in the SCE field in the top header. Now open the SIO2 export script and run it exactly as you did in Chapter 3. You don't need to worry about exporting normals or triangulating the mesh. Simply make sure all objects are selected and export the contents of the file to an `.sio2` file in the project directory.

Now you've got your 3D content and you're ready to do some experimenting with picking.

Implementing Color-Based Picking in SIO2

In your newly created project, open `EAGLView.mm` and change the render callback function initialization, just as you did in Chapter 4. Find the line

```
sio2->_SIO2window->_SIO2windowrender = templateRender
```

and change it to

```
sio2->_SIO2window->_SIO2windowrender = templateLoading;
```

Then add the declaration of `templateLoading` to `template.h`, among the other function declarations, as you also did in Chapter 4.

Drag and drop the file you just exported, `PickingScene.sio2`, from the Finder window into the `Resources` directory in the Xcode Groups & Files list.

Finally, modify `template.mm` as shown in the following code. In the explanations that follow, I'll be referring to specific lines of code, so make sure your line numbers correspond with what's printed here. This is the full code for `template.mm`:

```
10  #include "template.h"
11  #include "../src/sio2/sio2.h"
12
13  vec2 t;
14  unsigned char tap_select = 0;
15  SIO2object *selection = NULL;
16
17  void templateRender( void )
18  {
19       glMatrixMode( GL_MODELVIEW );
20       glLoadIdentity();
21
22       glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );
23
24       sio2->_SIO2camera = sio2ResourceGetCamera( sio2->_SIO2resource,
25                                                 "camera/Camera");
26
27
28       if(sio2->_SIO2camera){
29            sio2Perspective( sio2->_SIO2camera->fov,
30                             sio2->_SIO2window->scl->x /
                                          sio2->_SIO2window->scl->y,
31                             sio2->_SIO2camera->cstart,
32                             sio2->_SIO2camera->cend );
33
34            sio2CameraRender( sio2->_SIO2camera );
35            sio2ResourceRender( sio2->_SIO2resource,
36                                 sio2->_SIO2window,
37                                 sio2->_SIO2camera,
38                                 SIO2_RENDER_SOLID_OBJECT );
```

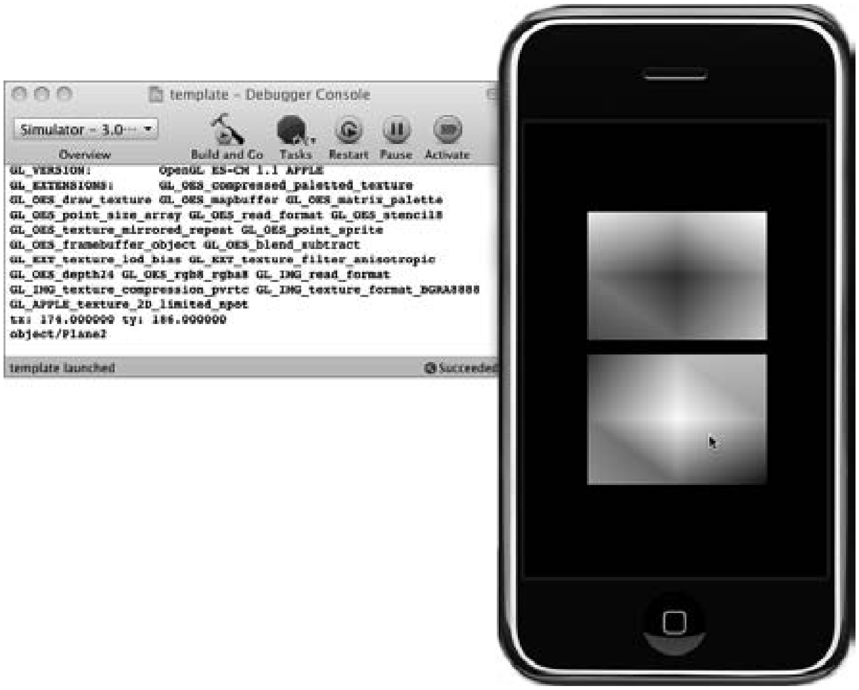
```

          if(tap_select){
               tap_select = 0;

               printf("tx: %f ty: %f \n", t.x, t.y);

               selection = sio2ResourceSelect3D( sio2->_SIO2resource,
                                                  sio2->_SIO2camera,
                                                  sio2->_SIO2window,
                                                  &t);

               printf( "%s\n", selection->name );
          }
     }
}

void templateLoading( void )
{
     unsigned int i = 0;

     sio2ResourceCreateDictionary( sio2->_SIO2resource );

     sio2ResourceOpen( sio2->_SIO2resource,
                       "PickingScene.sio2", 1);

     while( i != sio2->_SIO2resource->gi.number_entry )
     {
          sio2ResourceExtract( sio2->_SIO2resource, NULL );
          ++i;
     }
     sio2ResourceClose( sio2->_SIO2resource );
     sio2ResourceBindAllMatrix( sio2->_SIO2resource );
     sio2ResourceGenId( sio2->_SIO2resource );
     sio2ResetState();

     sio2->_SIO2window->_SIO2windowrender = templateRender;
}

void templateScreenTap( void *_ptr, unsigned char _state )
{
     if( _state == SIO2_WINDOW_TAP_DOWN ){
          t.x = sio2->_SIO2window->touch[ 0 ]->x;
          t.y = sio2->_SIO2window->scl->y -
                    sio2->_SIO2window->touch[ 0 ]->y;
          tap_select = 1;
     }
}
```

```

void templateScreenTouchMove( void *_ptr )
{
}

void templateScreenAccelerometer( void *_ptr )
{
}
```

`SIO2object` **`sio2ResourceSelect3D`**`(SIO2resource` *`*resource`*`, SIO2camera` *`*camera`*`, SIO2window` *`*window`*`, vec2` *`*v`*`)`

This function handles picking objects in a 3D space via screen touch. The function takes as arguments a resource manager *resource*, a camera object *camera*, a window object *window*, and a 2D vector *v* representing the point on the screen that was touched. It returns the selected SIO2object.

If you build and run this code as it is, you should see the two colorful planes on your simulator or device screen. They don't do much though. In order to see the functionality, you'll need to run the console. From the Xcode Run menu, choose Console. Now build and run your code again. You should now be able to pick the planes by clicking or tapping on them. The selected object's name will be printed to the console as shown in Figure 5.16. If you aren't able to get this working, go back and make sure you've got the code typed in correctly, or check the troubleshooting tips in Chapter 4.



```
GL_VERSION:         OpenGL ES-CM 1.1 APPLE
GL_EXTENSIONS:      GL_OES_compressed_paletted_texture
GL_OES_draw_texture GL_OES_mapbuffer GL_OES_matrix_palette
GL_OES_point_size_array GL_OES_read_format GL_OES_stencil8
GL_OES_texture_mirrored_repeat GL_OES_point_sprite
GL_OES_framebuffer_object GL_OES_blend_subtract
GL_EXT_texture_lod_bias GL_EXT_texture_filter_anisotropic
GL_OES_depth24 GL_OES_rgb8_rgba8 GL_IMG_read_format
GL_IMG_texture_compression_pvrtc GL_IMG_texture_format_BGRA8888
GL_APPLE_texture_2D_limited_npot
tx: 174.000000 ty: 186.000000
object/Plane2
```

Figure 5.16
Running your code in the iPhone simulator with the console

So how is this code working? Much of the code you typed in is the same as code you've seen in previous chapters. The resource loading code is pretty much identical to what you saw in Chapter 4. The new stuff here is related to the picking functionality. Lines 13 to 15 declare relevant variables. The first one, `t` of type `vec2`, should be somewhat familiar from Chapter 4. This is a two-dimensional vector that will hold the x- and y-coordinate values of the touch point. Note that the y-coordinate must be reversed by subtracting the touched coordinate from the total scale of the window along the y-axis. This is because the iPhone's touch screen y-coordinate is counted from the bottom corner up, whereas the OpenGL ES color buffer's y-coordinate is counted from the top corner down. The next variable, `tap_select`, will serve as a flag indicating whether a tap has landed. The `selection` variable holds a pointer to an `SIO2object`. As its name implies, this will be used to store a pointer to the 3D object that is selected (tapped) by the user.

The `templateScreenTap` function beginning at line 77 is where the tap is registered. Two things happen in this function: the x- and y-coordinate values of `t` are set according to the touch position, and `tap_select` is set to `1`.

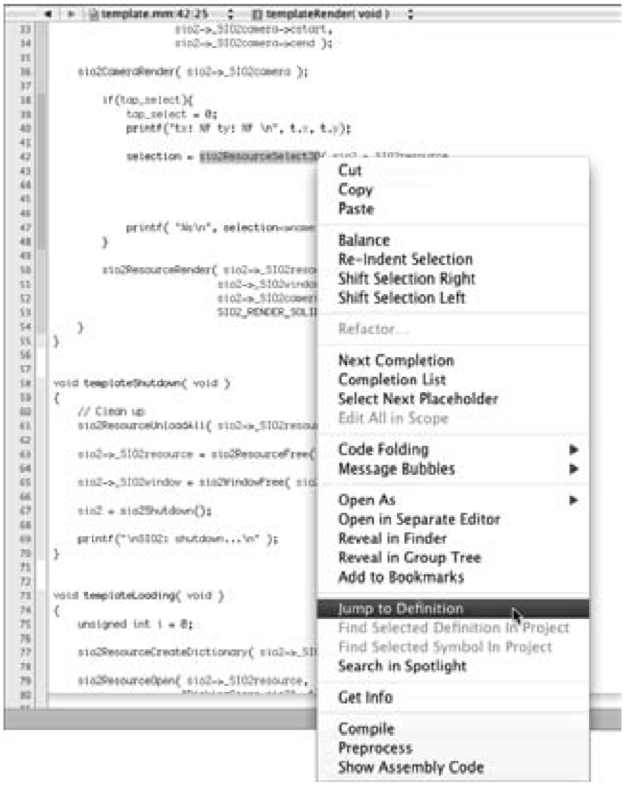
When `tap_select` is set to `1`, the picking code in `templateRender` is executed according to the conditional statement on line 40. The first thing that happens is that `tap_select` is reset to `0` so that future taps can be registered separately. The `printf` statement on line 53 prints the x- and y-coordinates to the console.

Line 45 is where `selection` receives its value from the `sio2ResourceSelect3D` function. This is where the picking action happens. The function takes the resource manager, the camera, the window object, and the touch coordinate vector `t` and returns the selected object. Line 50 prints the name of the selected object to the console.

A Closer Look at the Picking Code

It's worthwhile to take a closer look at `sio2ResourceSelect3D`. The fact is, it's worthwhile to take a closer look at any function you use because understanding how a function does its job will give you insights into when it's appropriate and how to get the most out of using it. Understanding the workings of the built-in SIO2 functions will also be enormously valuable when debugging your own code. It's a good idea to be careful when looking at files in the `src` directory though. Any edits you make to these files will affect all of your projects because they all share a single `src` directory. In general, you won't want to modify these files, but you might occasionally want to print out some diagnostic messages as you explore the functionality. Just be sure to leave source files as you found them when you're done.

Xcode provides numerous tools for navigating and debugging code, and the more you can learn about these tools, the faster and more efficiently you'll be able to write code. In order to drill into the `sio2ResourceSelect3D` function definition itself, you'll use one of these tools now. Right-click over the function call in line 45. When you do, you'll see the name of the function highlighted in blue and a menu will appear. From the menu, select Jump To Definition as shown in Figure 5.17. Voilà! You are immediately taken directly to the function definition in `sio2_resource.cc`.



```
                        sio2->_SIO2camera->cstart,
                        sio2->_SIO2camera->cend );

    sio2CameraRender( sio2->_SIO2camera );

        if(tap_select){
            tap_select = 0;
            printf("tx: %f ty: %f \n", t.x, t.y);

            selection = sio2ResourceSelect3D(

            printf( "%s\n", selection->

        }

        sio2ResourceRender( sio2->_SIO2reso
                            sio2->_SIO2windo
                            sio2->_SIO2camer
                            SIO2_RENDER_SOLI
    }
}


void templateShutdown( void )
{
    // Clean up
    sio2ResourceUnloadAll( sio2->_SIO2resou

    sio2->_SIO2resource = sio2ResourceFree(

    sio2->_SIO2window = sio2WindowFree( sio

    sio2 = sio2Shutdown();

    printf("\nSIO2: shutdown...\n" );
}


void templateLoading( void )
{
    unsigned int i = 0;

    sio2ResourceCreateDictionary( sio2->_SI

    sio2ResourceOpen( sio2->_SIO2resource,
```

Figure 5.17

Jumping to a function definition

The code for the `sio2ResourceSelect3D` function looks like this:

```
SIO2object *sio2ResourceSelect3D( SIO2resource *_SIO2resource,
                                  SIO2camera   *_SIO2camera,
                                  SIO2window   *_SIO2window,
                                  vec2         *_v )
{
    unsigned int i = 0,
    selection = 0;

    col4 col;

    while( i != _SIO2resource->n_object )
    {
        SIO2object *_SIO2object =
               ( SIO2object * )_SIO2resource->_SIO2object[ i ];

2563            sio2GenColorIndex( selection, _SIO2object->col );
2564
2565            if( sio2ObjectRender( _SIO2object,
2566                                  _SIO2window,
2567                                  _SIO2camera,
2568                                  0, 1 ) )
2569            { ++selection; }
2570            else
2571            { _SIO2object->col->a = 0; }
2572
2573            ++i;
2574       }
2575
2576       glReadPixels( ( int )_v->x,
2577                     ( int )_v->y,
2578                     1, 1,
2579                     GL_RGBA, GL_UNSIGNED_BYTE, &col );
2580
2581       if( !col.a )
2582       { return NULL; }
2583
2584       i = 0;
2585       while( i != _SIO2resource->n_object )
2586       {
2587            SIO2object *_SIO2object =
                     ( SIO2object * )_SIO2resource->_SIO2object[ i ];
2588
2589            if( !memcmp( &col, _SIO2object->col, 4 ) )
2590            { return _SIO2object; }
2591
2592            ++i;
2593       }
2594
2595       return NULL;
2596  }
```

`void` **`sio2GenColorIndex`**`(unsigned int *`*`index`*`, col4 *`*`col`*`)`

This function generates an array of unique RGBA colors. The *index* value determines how many colors are generated, and the generated colors are stored in the color array col.

After declaring the necessary variables, the function begins looping through all of the objects in the scene on line 2559 using the `SIO2resource->n_object` value from the resource manager. This value holds the number of objects in the scene, so the `i` value in this loop will iterate through all the objects. The `sio2GenColorIndex` function assigns a unique color to each object, replacing the object's original color. You can dig into this function the same way you did before, but for the purpose of this description, it's enough to know generally what's being done. The first color has R, G, and B values of `0`, and subsequent colors increment the B value by `1`.

The `sio2ObjectRender` function on line 2565 renders the frame to the OpenGL color buffer. The conditional here is to ensure that objects intended to be invisible are given *`_SIO2object->col->a`* values of `0` so that they will not be selected later.

On line 2576, `glReadPixels` is called. This is an OpenGL ES function that takes the coordinate values of a location on the screen, in this case the touch point, and assigns the RGBA values of the point to a variable. In this case, the *`col`* variable receives the color information.

Finally, the loop beginning at 2587 iterates through the objects, comparing their temporary color with the color of the current pixel, as assigned by `glReadPixels`. This comparison happens on line 2589. The `memcmp` C function compares two chunks of memory and returns zero if they are identical. The result of doing this is to return the object whose temporary color is identical to the pixel being touched. In this way, picking is accomplished without the need for any fancy spatial calculations other than the ones that come for free with rendering.

To see the `glReadPixels` and color assignment more vividly, try reading the pixels and printing out their output *before* the generation of the color index on line 2563 by inserting the following code before that line:

```
glReadPixels( ( int )_v->x,
            ( int )_v->y,
            1, 1,
            GL_RGBA, GL_UNSIGNED_BYTE, &col );
printf("r:%u, g:%u, b%:u", col.r, col.g, col.b)
```

This will print the original vertex colors of the planes. You can repeat the `printf` line after `glReadPixels` is called the second time to see how the values change. Be sure to delete these additions to the source code when you've gotten the idea.

Adding Picking to Hello3DWorld

Now you'll return to the `Hello3DWorld` project from Chapter 4 and add picking functionality to that application. If you followed the previous section, then adding picking to `Hello3DWorld` will be pretty straightforward, but there are two main complications. These are lighting and materials. Both lighting and materials influence the color of individual pixels in a way that the generated color assignment does not control. The way this needs to be dealt with is by disabling lighting and materials just before entering the picking