Build awesome 3D apps using the
iOS native OpenGL ES library

Pro OpenGL ES for iOS

Mike Smithwick

Apress®

Pro OpenGL ES for iOS

Mike Smithwick

Apress®

Pro OpenGL ES for iOS

ISBN-13 (pbk): 978-1-4302-3840-9

ISBN-13 (electronic): 978-1-4302-3841-6

President and Publisher: Paul Manning
Lead Editor: Richard Carey
Technical Reviewer: Leila Muhtasib
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Corbin Collins
Copy Editor: Kim Wimpsett
Production Support: Patrick Cunningham
Indexer: BIM Indexing & Proofreading Services
Artist: SPI Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary materials referenced by the author in this text is available to readers at `www.apress.com`. For detailed information about how to locate your book's source code, go to `www.apress.com/source-code`.

To a couple of the greatest parents in the world, who always supported me, never flinching at my wacky requests such as sending me back to see an Apollo launch or buying a telescope.

Contents at a Glance

Contents

About the Author

Mike Smithwick's slow descent into programming computers began when he first got a little 3-bit plastic DigiComp 1 computer in 1963 (`http://en.wikipedia.org/wiki/Digi-Comp_I`). Not too long before that, he got interested in planetariums. Eventually he graduated to programming NASA flight simulator graphics through the 1980s. But what he really wanted to do was become a syndicated cartoonist (really!). Failing to get any syndication deals, he wrote and sold the popular Distant Suns planetarium program for the Commodore Amiga, old-school Mac, and Microsoft Windows while selling himself as a contract programmer on the side, working for Apple, 3DO, Sense-8, and Epyx. Eventually he landed a "real" job at Live365, working on client software Windows and Windows Mobile 6, TiVo, Symbian (ahhh…Symbian…), and iPhone. After 13 short years he decided to go back to the dark side of contracting, writing, and working on Distant Suns for the iPhone after it became modest success in the App Store. Sometimes late at night, he thinks he can hear his Woz-autographed Apple II sobbing for attention from the garage. He may be contacted via `www.distantsuns.com`, lazyastronomer on AIM, and @distantsuns or @lazyastronomer on Twitter.

About the Technical Reviewer

Leila Muhtasib has been passionate about programming since she wrote her first program on MS-DOS. Since then, she's graduated with a Computer Science degree from the University of Maryland, College Park. Fascinated by mobile technology and its increasing ubiquity, she has been programming iPhone applications since the first SDK was released. She is now a Senior Software Engineer and Tech Lead of a mobile development team at Cisco Systems.

Acknowledgments

Thanks to Corbin Collins and Richard Carey, my long-suffering editors, for putting up with a first-time author, someone who clearly needs to read *Writing iOS Books for Beginners*.

And to Leila Muhtasib, my tech editor, who was every bit as good as I thought she would be.

And to Matthew Moodie and Mark Beckner for approving the schedule slippage so I could add in iOS 5 content, ensuring that the book wasn't obsolete on its release day.

And, of course, to Steve Jobs for never compromising and for producing insanely great tools that make work fun and make fun "funner."

Introduction

In 1985 I brought home a new shiny Commodore Amiga 1000, about one week after they were released. Coming with a whopping 512K of memory, programmable colormaps, a Motorola 68K CPU, and a modern multitasking operating system, it had "awesome" writ all over it. Metaphorically speaking, of course. I thought it might make a good platform for an astronomy program, as I could now control the colors of those star-things instead of having to settle for a lame fixed color palette forced upon me from the likes of Hercules or the C64. So I coded up a 24-line basic routine to draw a random star field, turned out the lights, and thought, "Wow! I bet I could write a cool astronomy program for that thing!" Twenty-six years later I am still working on it (I'll get it right one of these days). Back then my dream device was something I could slip into my pocket, pull out when needed, and aim it as the sky to tell me what stars or constellations I was looking at.

It's called the iPhone.

I thought of it first.

As good as the iPhone is for playing music, making calls, or jumping Doodles, it really shines when you get to the 3D stuff. After all, 3D is all around us—unless you are a pirate and have taken to wearing an eye patch, in which case you'll have very limited depth perception. Arrrggghhh.

Plus 3D apps are fun to show off to people. They'll "get it." In fact, they'll get it much more than, say, that mulch buyer's guide app all the kids are talking about. (Unless they show off their mulch in 3D, but that would be a waste of a perfectly good dimension.)

So, 3D apps are fun to see, fun to interact with, and fun to program. Which brings me to this book. I am by no means a guru in this field. The real gurus are the ones who can knock out a couple of NVIDIA drivers before breakfast, 4-dimensional hypercube simulators by lunch, and port Halo to a TokyoFlash watch before the evening's *Firefly* marathon on SyFy. I can't do that. But I am a decent writer, have enough of a working knowledge of the subject to make me harmless, and know how to spell "3D." So here we are.

First and foremost this book is for experienced iOS programmers who want to at least learn a little of the language of 3D. At least enough to where at the next game programmer's cocktail party you too can laugh at the quaternion jokes with the best of them.

This book covers the basics in both theory of 3D and implementations using the industry standard OpenGL ES toolkit for small devices. While iOS supports both flavors—version 1.x for the easy way, and version 2.x for those who like to get where the nitty-is-gritty—I mainly cover the former, except in the final chapter which serves as an intro to the latter and the use of programmable shaders. And with the release of iOS 5, Apple has offered the 3D community a whole lotta lovin' with some significant additions to the graphics libraries.

Chapter 1 serves as an intro to OpenGL ES alongside the long and tortuous path of the history of computer graphics. Chapter 2 is the math behind basic 3D rendering, whereas Chapters 3 through 8 lead you gently through the various issues all graphics programmers eventually come across, such as how to cast shadows, render multiple OpenGL screens, add lens flare, and so on. Eventually this works its way into a simple (S-I-M-P-L-E!) solar-system model consisting of the sun, earth, and some stars—a traditional 3D exercise. Chapter 9 looks at best practices and development tools, and Chapter 10 serves as a brief overview of OpenGL ES 2 and the use of shaders.

So, have fun, send me some M&Ms, and while you're at it feel free to check out my own app in the Appstore: Distant Suns 3 for both the iPhone and the iPad. Yup, that's the same application that started out on a Commodore Amiga 1000 in 1985 as a 24-line basic program that drew a couple hundred random stars on the screen.

It's bigger now.

Chapter 1

Computer Graphics: From Then to Now

To predict the future and appreciate the present, you must understand the past.

—Probably said by someone sometime

Computer graphics have always been the darling of the software world. Laypeople can appreciate computer graphics more easily than, say, increasing the speed of a sort algorithm by 3 percent or adding automatic tint control to a spreadsheet program. You are likely to hear more people say "Cooooollllll!" at your nicely rendered image of Saturn on your iPad than at a Visual Basic script in Microsoft Word (unless, of course, a Visual Basic script in Microsoft Word can render Saturn, then that really would be cool). The cool factor goes up even more so when said renderings are on a device you can carry around in your back pocket. Let's face it—Steve Jobs has made the life of art directors on science-fiction films very difficult. After all, imagine how hard it must be to design a prop that looks more futuristic than an iPad. (Even before the iPhone was available for sale, the prop department at ABC's *LOST* borrowed some of Apple's screen iconography for use in a two-way radio carried by a helicopter pilot.)

If you are reading this book, chances are you have an iOS-based device or are considering getting one in the near future. If you have one, put it in your hand now and consider what a miracle it is of 21st-century engineering. Millions of man-hours, billions of dollars of research, centuries of overtime, plenty of all-nighters, and an abundance of Jolt-drinking, T-shirt–wearing, comic-book-loving engineers coding into the silence of the night have gone into making that little glass and plastic miracle-box so you could play DoodleJump when *Mythbusters* is in reruns.

Your First OpenGL ES Program

Some software how-to titles will carefully build up the case for their specific topic ("the boring stuff") only to get to the coding and examples ("the fun stuff") by around page 655. Others will jump immediately into some exercises to address your curiosity and save the boring stuff for a little later. This book will be of the latter category.

> **Note** OpenGL ES is a 3D graphics standard based on the OpenGL library that emerged from the labs of Silicon Graphics in 1992. It is widely used across the industry in everything from pocketable machines running games up to supercomputers running fluid dynamics simulations for NASA (and playing really, really fast games). The ES variety stands for *Embedded Systems*, meaning small, portable, low-power devices. Unless otherwise noted, I'll use OpenGL and OpenGL ES interchangeably.

When developing any apps for iOS, it is customary to let Xcode do the heavy lifting at the beginning of any project via its various wizards. With Xcode (this book uses Xcode 4 as reference), you can easily create an example OpenGL ES project and then add on your own stuff to eventually arrive at something someone might want to buy from the App Store.

With Xcode 4 already running, go to **File** ▸ **New** ▸ **New Project**, and you should see something that looks like Figure 1-1.

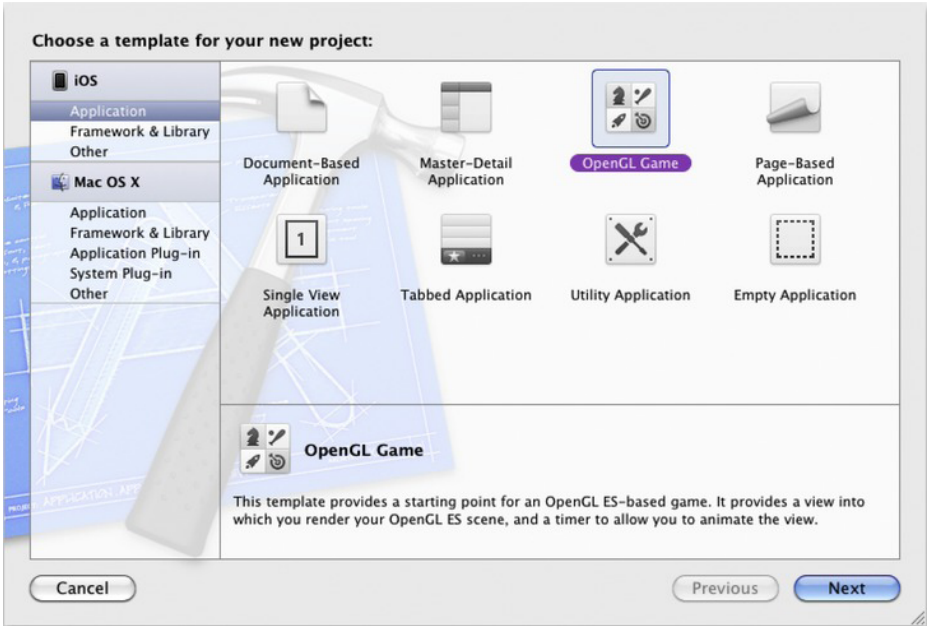

Figure 1-1. *Xcode project wizard*

Select the *OpenGL Game* template, and fill in the needed project data. It doesn't matter whether it is for the iPhone or iPad.

Now compile and run, making sure you have administrative privileges. If you didn't break anything by undue tinkering, you should see something like Figure 1-2.

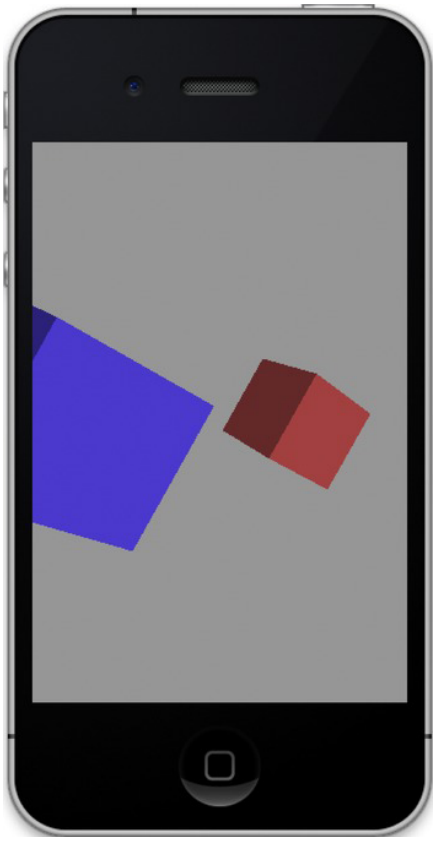

Figure 1-2. *Your first OpenGL ES project. Give yourself a high five.*

The code will be examined later. And don't worry, you'll build stuff fancier than a couple of rotating cubes. The main project will be to construct a simple solar-system simulator based on some of the code used in Distant Suns 3. But for now, it's time to get to the boring stuff: where computer graphics came from and where it is likely to go.

A Spotty History of Computer Graphics

To say that 3D is all the rage today is at best an understatement. Although forms of "3D" imagery go back to more than a century ago, it seems that it has finally come of age. First let's look at what 3D is and what it is not.

3D in Hollywood

In 1982 Disney released *Tron*, the first movie to widely use computer graphics depicting life inside a video game. Although the movie was a critical and financial flop (not unlike the big-budget sequel released in 2011), it would eventually join the ranks of cult favorites right up there with *Showgirls* and *The Rocky Horror Picture Show*. Hollywood had taken the bite out of the apple, and there was no turning back.

Stretching back to the 1800s, what we call "3D" today was more commonly referred to as *stereo vision*. Popular Victorian-era *stereopticons* would be found in many parlors of the day. Consider this technology an early Viewmaster. The user would hold the stereopticon up to their face with a stereo photograph slipped into the far end and see a view of some distant land, but in stereo rather than a flat 2D picture. Each eye would see only one half of the card, which carried two nearly identical photos taken only a couple of inches apart.

Stereovision is what gives us the notion of a depth component to our field of view. Our two eyes deliver two slightly different images to the brain that then interprets them in a way that we understand as depth perception. A single image will not have that effect. Eventually this moved to movies, with a brief and unsuccessful dalliance as far back as 1903 (the short *L'arrivée du Train* is said to have had viewers running from the theater to avoid the train that was clearly heading their way*)* and a resurgence in the early 1950s, with *Bwana Devil* being perhaps the best known.

The original form of 3D movies generally used the "anaglyph" technique that required the viewers to wear cheap plastic glasses with a red filter over one eye and a blue one over the other. Polarizing systems were incorporated in the early 1950s and permitted color movies to be seen in stereo, and they are still very much the same as today. Afraid that television would kill off the movie industry, Hollywood needed some gimmick that was impossible on television in order to keep selling tickets, but because both the cameras and the projectors required were much too impractical and costly, the form fell out of favor, and the movie industry struggled along just fine.

With the advent of digital projection systems in the 1990s and fully rendered films such as *Toy Story*, stereo movies and eventually television finally became both practical and affordable enough to move it beyond the gimmick stage. In particular, full-length animated features (*Toy Story* being the first) made it a no-brainer to convert to stereo. All one needed to do was simply rerender the entire film but from a slightly different viewpoint. This is where stereo and 3D computer graphics merge.

The Dawn of Computer Graphics

One of the fascinating things about the history of computer graphics, and computers in general, is that the technology is still so new that many of the giants still stride among us. It would be tough to track down whoever invented the buggy whip, but I'd know whom to call if you wanted to hear firsthand how to program the Apollo Lunar Module computers from the 1960s.

Computer graphics (frequently referred to as CG) come in three overall flavors: 2D for user interface, 3D in real time for flight or other forms of simulation as well as games, and 3D rendering where quality trumps speed for non-real-time use.

MIT

In 1961, an MIT engineering student named Ivan Sutherland created a system called Sketchpad for his PhD thesis using a vectorscope, a crude light pen, and a custom-made Lincoln TX-2 computer (a spin-off from the TX-2 group would become DEC). Sketchpad's revolutionary graphical user interface demonstrated many of the core principles of modern UI design, not to mention a big helping of object-oriented architecture tossed in for good measure.

> **Note** For a video of Sketchpad in operation, go to YouTube and search for *Sketchpad* or *Ivan Sutherland*.

A fellow student of Sutherland's, Steve Russell, would invent perhaps one of the biggest time sinks ever made, the computer game. Russell created the legendary game of Spacewar in 1962, which ran on the PDP-1, as shown in Figure 1-3.

Figure 1-3. *The 1962 game of Spacewar resurrected at the Computer History Museum in Mountain View, California, on a vintage PDP-1. Photo by Joi Itoh, licensed under the Creative Commons Attribution 2.0 Generic license (*`http://creativecommons.org/licenses/by/2.0/deed.en`*).*

By 1965, IBM would release what is considered the first widely used commercial graphics terminal, the 2250. Paired with either the low-cost IBM-1130 computer or the IBM S/340, the terminal was meant largely for use in the scientific community.

Perhaps one of the earliest known examples of computer graphics on television was the use of a 2250 on the CBS news coverage of the joint Gemini 6 and Gemini 7 missions in December 1965 (IBM built the Gemini's onboard computer system). The terminal was used to demonstrate several phases of the mission on live television from liftoff to rendezvous. At a cost of about $100,000 in 1965, it was worth the equivalent of a very nice home. See Figure 1-4.

Figure 1-4*. IBM-2250 terminal from 1965. Courtesy NASA.*

University of Utah

Recruited by the University of Utah in 1968 to work in its computer science program, Sutherland naturally concentrated on graphics. Over the course of the next few years, many computer graphics visionaries in training would pass through the university's labs.

Ed Catmull, for example, loved classic animation but was frustrated by his inability to draw—a requirement for artists back in those days as it would appear. Sensing that computers might be a pathway to making movies, Catmull produced the first-ever computer animation, which was of his hand opening and closing. This clip would find its way into the 1976 film *Future World*.

During that time he would pioneer two major computer graphics innovations: texture mapping and bicubic surfaces. The former could be used to add complexity to simple forms by using images of texture instead of having to create texture and roughness using discrete points and surfaces, as shown in Figure 1-5. The latter is used to generate algorithmically curved surfaces that are much more efficient than the traditional polygon meshes.

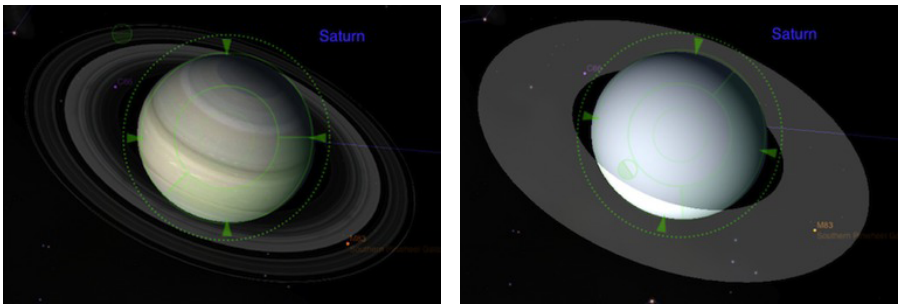

Figure 1-5. *Saturn with and without texture*

Catmull would eventually find his way to Lucasfilm and, later, Pixar and eventually serve as president of Disney Animation Studios where he could finally make the movies he wanted to see. Not a bad gig.

Many others of the top names in the industry would likewise pass through the gates of University of Utah and the influence of Sutherland:

- John Warnock, who would be instrumental in developing a device-independent means of displaying and printing graphics called PostScript and the Portable Document Format (PDF) and would be cofounder of Adobe.
- Jim Clark, founder of Silicon Graphics (SGI), which would supply Hollywood with some of the best graphics workstations of the day and create the 3D software development framework now known as OpenGL. After SGI, he co-founded Netscape Communications, which would lead us into the land of the World Wide Web.
- Jim Blinn, inventor of both bump mapping, which is an efficient way of adding true 3D texture to objects, and environment mapping, which is used to create really shiny things. Perhaps he would be best known creating the revolutionary animations for NASA's Voyager project, depicting their flybys of the outer planets, as shown in Figure 1-6 (compare that with Figure 1-7 using modern devices). Of Blinn, Sutherland would say, "There are about a dozen great computer graphics people, and Jim Blinn is six of them." Blinn would later lead the effort to create Microsoft's competitor to OpenGL, namely, Direct3D.

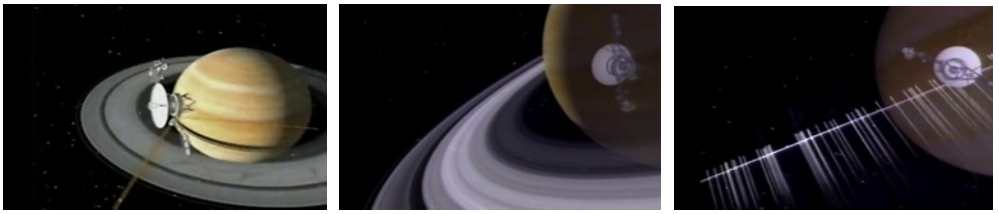

Figure 1-6. *Jim Blinn's depiction of Voyager II's encounter with Saturn in August of 1981. Notice the streaks formed of icy particles while crossing the ring plane. Courtesy NASA.*

Figure 1-7. *Compare Figure 1-6, using some of the best graphics computers and software at the time, with a similar view of Saturn from Distant Suns 3 running on a $500 iPad.*

Coming of Age in Hollywood

Computer graphics would really start to come into their own in the 1980s thanks both to Hollywood and to machines that were increasingly powerful while at the same time costing less. For example, the beloved Commodore Amiga that was introduced in 1985 cost less than $2,000, and it brought to the consumer market an advanced multitasking operating system and color graphics that had been previously the domain of workstations costing upwards of $100,000. See Figure 1-8.

Figure 1-8. *Amiga 1000, circa 1985. Photo by Kaivv, licensed under the Creative Commons Attribution 2.0 Generic license (*`http://creativecommons.org/licenses/by/2.0/deed.en`*).*

Compare this to the original black-and-white Mac that was released a scant 18 months earlier for about the same cost. Coming with a very primitive OS, flat file system, and 1-bit display, it was fertile territory for the "religious wars" that broke out between the various camps as to whose machine was better (wars that would also include the Atari ST).

> **Note** One of the special graphics modes on the original Amiga could compress 4,096 colors into a system that would normally max out at 32. Called Hold and Modify (HAM mode), it was originally included on one of the main chips for experimental reasons by designer Jay Miner. Although he wanted to remove the admitted kludge that produced images with a lot of color distortion, the results would have left a big empty spot on the chip. Considering that unused chip landscape was something no self-respecting engineer could tolerate, he left it in, and to Miner's great surprise, people started using it.

A company in Kansas called NewTek pioneered the use of Amigas for rendering high-quality 3D graphics when coupled with its special hardware named the Video Toaster. Combined with a sophisticated 3D rendering software package called Lightwave 3D, NewTek opened up the realm of cheap, network-quality graphics to anyone who had a few thousand dollars to spend. This development opened the doors for elaborate science-fiction shows such as *Babylon 5* or *Seaquest* to be financially feasible considering their extensive special effects needs.

During the 1980s, many more techniques and innovations would work their way into common use in the CG community:

- Loren Carpenter developed a technique to generate highly detailed landscapes algorithmically using something called *fractals*. Carpenter was hired by Lucasfilm to create a rendering package for a new company named Pixar. The result was REYES, which stood for Render Everything You Ever Saw.
- Turner Whitted developed a technique called *ray tracing* that could produce highly realistic scenes (at a significant CPU cost), particularly when they included objects with various reflective and refractive properties. Glass items were common subjects in various early ray-tracing efforts, as shown in Figure 1-9.
- Frank Crow developed the first practical method of *anti-aliasing* in computer graphics. Aliasing is the phenomenon that generates jagged edges because of the relatively poor resolution of the display. Crow's method would smooth out everything from lines to text, producing far more natural and pleasing imagery. Note that one of Lucasfilm's early games was called *Rescue on Fractalus*. The bad guys were named *jaggies* (another term for anti-aliasing).
- *Star Trek II: The Wrath of Khan* brought with it the first entirely computer-generated sequence used to illustrate how a device called the Genesis Machine could generate life on a lifeless planet. That one simulation was called "the effect that wouldn't die" because of its groundbreaking techniques in flame and particle animation, along with the use of fractal landscapes.

Figure 1-9. *Sophisticated images such as this are within the range of hobbyists with programs such as the open source POV-Ray. Photo by Gilles Tran, 2006.*

The 1990s brought the T1000 "liquid metal" terminator in *Terminator 2: Judgment Day*, the first completely computer-generated full-length feature film of *Toy Story*, believable animated dinosaurs in *Jurassic Park*, and James Cameron's *Titanic*, all of which helped solidified CG as a common tool in the Hollywood director's arsenal.

By the decade's end, it would be hard to find any films that didn't have computer graphics as part of the production in either actual effects or in postproduction to help clean up various scenes. New techniques are still being developed and applied in ever more spectacular fashion, as in Disney's delightful *Up!* or James Cameron's beautiful *Avatar*.

Now, once again, take out your i-device and realize what a little technological marvel it is. Feel free to say "wow" in hushed, respectful tones.

Toolkits

All of the 3D wizardry referenced earlier would never have been possible without software. Many CG software programs are highly specialized, and others are more general purpose, such as OpenGL ES, the focus of this book. So, what follows are a few of the many toolkits available.

OpenGL

Open Graphics Library (OpenGL) came out of the pioneering efforts of SGI, the maker of high-end graphics workstations and mainframes. Its own proprietary graphics framework, IRIS-GL, had grown into a de-facto standard across the industry. To keep customers as competition increased, SGI opted to turn IRIS-GL into an open framework so as to strengthen their reputation as the industry leader. IRIS-GL was stripped of non-graphics-related functions and hardware-dependent features, renamed OpenGL, and released in early 1992. As of this writing, version 4.1 is the most current one available.

As small handheld devices became more common, OpenGL for Embedded Systems (OpenGL ES) was developed, which was a stripped-down version of the desktop version. It removed many of the more redundant API calls while simplifying other elements. making it run efficiently on lower-power CPUs. As a result, it has been widely adopted across many platforms, such as Android, iOS, Nintendo 3DS, and BlackBerry (OS 5.0 and newer).

There are two main flavors of OpenGL ES, 1.*x* and 2.*x*. Many devices support both. 1.*x* is the higher-level variant, based on the original OpenGL specification. Version 2.*x* (yes, I know it's confusing) is targeted toward more specialized rendering chores that can be handled by programmable graphics hardware.

Direct3D

Direct3D (D3D) is Microsoft's answer to OpenGL and is heavily oriented toward game developers. In 1995, Microsoft bought a small company called RenderMorphics that

specialized in creating a 3D framework named RealityLab for writing games. RealityLab was turned into Direct3D and first released in the summer of 1996. Even though it was proprietary to Windows-based systems, it has a huge user base across all of Microsoft's platforms: Windows, Windows 7 Mobile, and even Xbox. There are constant ongoing debates between the OpenGL and Direct3D camps as to which is more powerful, flexible, and easier to use. Other factors include how quickly hardware manufacturers can update their drivers to support new features, ease of understanding (Direct3D uses Microsoft's COM interface that can be very confusing for newcomers), stability, and industry support.

The Other Guys

While OpenGL and Direct3D remain at the top of the heap when it comes to both adoption and features, the graphics landscape is littered with numerous other frameworks, many which are supported on today's devices.

In the computer graphics world, graphics libraries come in two very broad flavors: low-level rendering mechanisms represented by OpenGL and Direct3D and high-level systems typically found in game engines that concentrate on resource management with special extras that extend to common gameplay elements (sound, networking, scoring, and so on). The latter are usually built on top of one of the former for the 3D portion. And if done well, the higher-level systems might even be abstracted enough to make it possible to work with both GL and D3D.

QuickDraw 3D

An example of a higher-level general-purpose library is QuickDraw 3D (QD3D). A 3D sibling to Apple's 2D QuickDraw (used in pre-OS-X days), QD3D had an elegant means of generating and linking objects in an easy-to-understand hierarchical fashion (*a scene-graph*). It likewise had its own file format for loading 3D models and a standard viewer and was platform independent. The higher-level part of QD3D would calculate the scene and determine how each object and, in turn, each piece of each object would be shown on a 2D drawing surface. Underneath QD3D there was a very thin layer called RAVE that would handle device-specific rendering of these bits.

Users could go with the standard version of RAVE, which would render the scene as expected. But more ambitious users could write their own that would display the scene in a more artistic fashion. For example, one company generated the RAVE output so as to look like their objects were hand-painted on the side of a cave. It was very cool when you could take this modern version of a cave drawing and spin it around. The plug-in architecture also made QD3D highly portable to other machines. When potential users balked at using QD3D since it had no hardware solution on PCs, a version of RAVE was produced that would use the hardware acceleration available for Direct3D by actually using its competitor as its rasterizer. Sadly, QD3D was almost immediately killed on the second coming of Steve Jobs, who determined that OpenGL should be the 3D standard for Macs in the future. This was an odd statement because QD3D was not a competitor to the other but an add-on that made the lives of programmers much easier. After Jobs

refused requests to make QD3D open source, the Quesa project was formed to re-create as much as possible the original library, which is still being supported at the time of this writing. And to nobody's surprise, Quesa uses OpenGL as its rendering engine.

A disclaimer here: I wrote the RAVE/Direct3D layer of QD3D only to have the project canceled a few days after going "gold master" (ready to ship).

OGRE

Another scene-graph system is Object-oriented Rendering Engine (OGRE). First released in 2005, OGRE can use both OpenGL and Direct3D as the low-level rasterizing solution, while offering users a stable and free toolkit used in many commercial products. The size of the user community is impressive. A quick peek at the forums shows more than 6,500 topics in the General Discussion section alone at the time of this writing.

OpenSceneGraph

Recently released for iOS devices, OpenSceneGraph does roughly what QuickDraw 3D did, by providing a means of creating your objects on a higher level, linking them together, and performing scene management duties and extra effects above the OpenGL layer. Other features include importing multiple file formats, text support, particle effects (used for sparks, flames, or clouds), and the ability to display video content in your 3D applications. Knowledge of OpenGL is highly recommended, because many of the OSG functions are merely thin wrappers to their OpenGL counterparts.

Unity3D

Unlike OGRE, QD3D, or OpenSceneGraph, Unity3D is a full-fledged game engine. The difference lies in the scope of the product. Whereas the first two concentrated on creating a more abstract wrapper around OpenGL, game engines go several steps further, supplying most if not all of the other supporting functionality that games would typically need such as sound, scripting, networked extensions, physics, user interface, and score-keeping modules. In addition, a good engine will likely have tools to help generate the assets and be platform independent.

Unity3D has all of these so would be overkill for many smaller projects. Also, being a commercial product, the source is not available, and it is not free to use, costing a modest amount (compared to other products in the past that could charge $100,000 or more).

And Still Others

Let's not ignore A6, Adventure Game Studio, C4, Cinder, Cocos3d, Crystal Space, VTK, Coin3D, SDL, QT, Delta3D, Glint3D, Esenthel, FlatRedBall, Horde3D, Irrlicht,

Leadwerks3D, Lightfeather, Raydium, Panda3D (from Disney Studios and CMU), Torque (available for iOS), and many others. Although they're powerful, one drawback of using game engines is that more often than not, your world is executed in their environment. So if you need a specific subtle behavior that is unavailable, you may be out of luck. That brings me back to the topic of this book.

Back to the Waltz of the Two Cubes

Up through iOS4, Apple saw OpenGL as more of a general-purpose framework. But starting with iOS5, they wanted to emphasize it as a perfect environment for game development. That is why, for example, the project icon in the wizard is titled "OpenGL Game," where previously it was "OpenGL ES Application." That also explains why the example exercise pushes the better performing—but considerably more cumbersome—OpenGL ES 2 environment, while ignoring the easier version that is the subject of this book.

> **Note** Also starting with iOS5, Apple has added a number of special helper-objects in their new GLKit framework that take over some of the common duties developers had to do themselves early on. These tasks include image loading, 3D-oriented math operations, creating a special OpenGL view, and managing special effects.

With that in mind, I'll step into 2.0-land every once in a while, such as via the example app described below, because that's all we have for now. Detailed discussions of 2.0 will be reserved for the last chapter, because it really is a fairly advanced topic for the scope of this book.

A Closer Look

The wizard produces six main files not including those of the plist and storyboards. Of these, there are the two for the view controller, two for the application delegate, and two mysterious looking things called shader.fsh and shader.vsh.

The shader files are unique to OpenGL ES 2.0 and are used to fine-tune the look of your scenes. They serve as small and very fast programs that execute on the graphics card itself, using their own unique language that resembles C. They give you the power to specify exactly how light and texture should show up in the final image. Unfortunately, OpenGL ES 2.0 requires shaders and hence a somewhat steeper learning curve, while the easier and more heavily used version 1.1 doesn't use shaders, settling for a few standard lighting and shading effects (called a "fixed function" pipeline). The shader-based applications are most likely going to be games where a visually rich experience is as important as anything else, while the easier 1.1 framework is just right for simple games, business graphics, educational titles, or any other apps that don't need to have perfect atmospheric modeling.

The application delegate has no active code in it, so we can ignore it. The real action takes place in the viewController via three main sections. The first initializes things using some of the standard view controller methods we all know and love, the second serves to render and animate the image, and the third section manages these shader things. Don't worry if you don't get it completely, because this example is merely intended to give you a general overview of what a basic OpenGL ES program looks like.

> **Note** All of these exercises are available on the Apress site, including additional bonus exercises that may not be in the book.

You will notice that throughout all of the listings, various parts of the code are marked with a numbered comment. The numbers correspond to the descriptions following the listing and that highlight various parts of the code.

Listing 1-1. *The initialization of the wizard-generated view controller.*

```
#import "TwoCubesViewController.h"

#define BUFFER_OFFSET(i) ((char *)NULL + (i))

// Uniform index.
Enum                                                                    //1
{
    UNIFORM_MODELVIEWPROJECTION_MATRIX,
    UNIFORM_NORMAL_MATRIX,
    NUM_UNIFORMS
};
GLint uniforms[NUM_UNIFORMS];

// Attribute index.
enum
{
    ATTRIB_VERTEX,
    ATTRIB_NORMAL,
    NUM_ATTRIBUTES
};

GLfloat gCubeVertexData[216] =                                          //2
{
    // Data layout for each line below is:
    // positionX, positionY, positionZ,     normalX, normalY, normalZ,
    0.5f, -0.5f, -0.5f,        1.0f, 0.0f, 0.0f,
    0.5f, 0.5f, -0.5f,         1.0f, 0.0f, 0.0f,
    0.5f, -0.5f, 0.5f,         1.0f, 0.0f, 0.0f,
    0.5f, -0.5f, 0.5f,         1.0f, 0.0f, 0.0f,
    0.5f, 0.5f, 0.5f,          1.0f, 0.0f, 0.0f,
    0.5f, 0.5f, -0.5f,         1.0f, 0.0f, 0.0f,

    0.5f, 0.5f, -0.5f,         0.0f, 1.0f, 0.0f,
    -0.5f, 0.5f, -0.5f,        0.0f, 1.0f, 0.0f,
    0.5f, 0.5f, 0.5f,          0.0f, 1.0f, 0.0f,
    0.5f, 0.5f, 0.5f,          0.0f, 1.0f, 0.0f,
```

```
    -0.5f, 0.5f, -0.5f,        0.0f, 1.0f, 0.0f,
    -0.5f, 0.5f, 0.5f,         0.0f, 1.0f, 0.0f,

    -0.5f, 0.5f, -0.5f,        -1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f,       -1.0f, 0.0f, 0.0f,
    -0.5f, 0.5f, 0.5f,         -1.0f, 0.0f, 0.0f,
    -0.5f, 0.5f, 0.5f,         -1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f,       -1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, 0.5f,        -1.0f, 0.0f, 0.0f,

    -0.5f, -0.5f, -0.5f,       0.0f, -1.0f, 0.0f,
    0.5f, -0.5f, -0.5f,        0.0f, -1.0f, 0.0f,
    -0.5f, -0.5f, 0.5f,        0.0f, -1.0f, 0.0f,
    -0.5f, -0.5f, 0.5f,        0.0f, -1.0f, 0.0f,
    0.5f, -0.5f, -0.5f,        0.0f, -1.0f, 0.0f,
    0.5f, -0.5f, 0.5f,         0.0f, -1.0f, 0.0f,

    0.5f, 0.5f, 0.5f,          0.0f, 0.0f, 1.0f,
    -0.5f, 0.5f, 0.5f,         0.0f, 0.0f, 1.0f,
    0.5f, -0.5f, 0.5f,         0.0f, 0.0f, 1.0f,
    0.5f, -0.5f, 0.5f,         0.0f, 0.0f, 1.0f,
    -0.5f, 0.5f, 0.5f,         0.0f, 0.0f, 1.0f,
    -0.5f, -0.5f, 0.5f,        0.0f, 0.0f, 1.0f,

    0.5f, -0.5f, -0.5f,        0.0f, 0.0f, -1.0f,
    -0.5f, -0.5f, -0.5f,       0.0f, 0.0f, -1.0f,
    0.5f, 0.5f, -0.5f,         0.0f, 0.0f, -1.0f,
    0.5f, 0.5f, -0.5f,         0.0f, 0.0f, -1.0f,
    -0.5f, -0.5f, -0.5f,       0.0f, 0.0f, -1.0f,
    -0.5f, 0.5f, -0.5f,        0.0f, 0.0f, -1.0f
};

@interface TwoCubesViewController () {
    GLuint _program;

    GLKMatrix4 _modelViewProjectionMatrix;                                    //3
    GLKMatrix3 _normalMatrix;
    float _rotation;

    GLuint _vertexArray;
    GLuint _vertexBuffer;
}
@property (strong, nonatomic) EAGLContext *context;
@property (strong, nonatomic) GLKBaseEffect *effect;

- (void)setupGL;
- (void)tearDownGL;

- (BOOL)loadShaders;
- (BOOL)compileShader:(GLuint *)shader type:(GLenum)type file:(NSString *)file;
- (BOOL)linkProgram:(GLuint)prog;
- (BOOL)validateProgram:(GLuint)prog;
@end
```