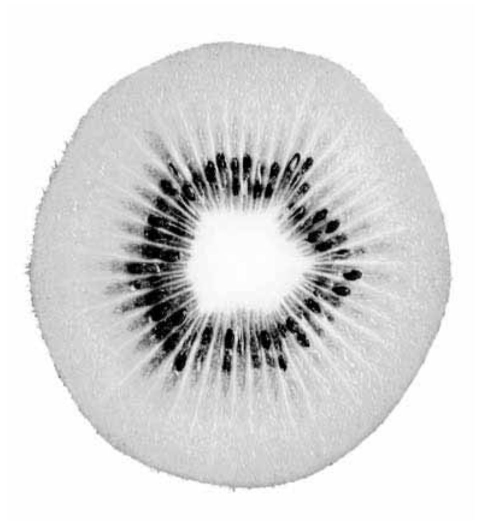
# iPhone Advanced Projects



Dave Mark, Series Editor

Joachim Bondo
Dylan Bruzenak
Steve Finkelstein
Owen Goss
Tom Harrington
Peter Honeder
Ray Kiddy
Noel Llopis
Joe Pezzillo
Florian Pflug
Jonathan Saggau
Ben Britten Smith

Apress®

**iPhone Advanced Projects**

Copyright © 2009 by Dave Mark, Joachim Bondo, Dylan Bruzenak, Steve Finkelstein, Owen Goss, Tom Harrington, Peter Honeder, Ray Kiddy, Noel Llopis, Joe Pezzillo, Florian Pflug, Jonathan Saggau, Ben Britten Smith

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-2403-7

ISBN-13 (electronic): 978-1-4302-2404-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Clay Andres
Technical Reviewer: Glenn Cole
Developmental Editor: Douglas Pundick
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Coordinating Editor: Kelly Moritz
Copy Editor: Kim Wimpsett
Compositor: MacPS, LLC
Indexer: Julie Grady
Artist: April Milne

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 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 http://www.springeronline.com.

For information on translations, please e-mail info@apress.com, or visit http://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 http://www.apress.com/info/bulksales.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com. You will need to answer questions pertaining to this book in order to successfully download the code.

*To my lovely wife, Leonie.*
*—Ben Britten Smith*

*To my wife, Malena, who once again gave me the support I hadn't earned.*
*—Joachim Bondo*

*To everyone I know and to everyone I haven't met yet.*
*—Dylan Bruzenak*

*To all of my family and friends for their support and patience with my demanding schedule. To my loving wife, Michelle, who sustains me and encourages me to take risks. Finally, this one is for my grandmother, Asya; you will live forever in all our hearts.*
*—Steve Finkelstein*

*To the iPhone game developers on Twitter for sharing so much and being such a supportive community.*
*—Noel Llopis (@snappytouch on Twitter)*

*I'm so grateful to so many people I can't possibly hope to name them all individually, so, en masse, let me thank the blessing that is my family (especially my son), the unstoppable geniuses at Apple, the folks at Apress who patiently awaited my writing, the incredibly supportive Mac and iPhone indie developer community, all my clients and customers, my business partners and colleagues, and, of course, the great ineffable spirit of the universe that makes everything possible.*
*Thank you!*
*—Joe Pezzillo*

*To my family, my friends, the island "La Palma," and the one who introduced me to it.*
*—Florian Pflug*

*To Dr. Michele, who doesn't let me call her doctor. Thanks for making me type.*
*—Jonathan Saggau*

# Contents at a Glance

■Contents at a Glance ........ iv
■Contents ........ v
■Foreword ........ xi
■About the Technical Reviewer ........ xii
■Preface ........ xiii
Ben Britten Smith ........ 1
■Everything You Ever Wanted to Know About Particle Systems ........ 3
Joachim Bondo ........ 37
■Chess on the 'Net: Correspondence Gaming with Deep Green ........ 39
Tom Harrington ........ 63
■Audio Streaming: An Exploration into Core Audio ........ 65
Owen Goss ........ 99
■You Go Squish Now! Debugging on the iPhone ........ 101
Dylan Bruzenak ........ 139
■Building Data-Driven Applications with Active Record and SQLite ........ 141
Ray Kiddy ........ 181
■Core Data and Hard-Core Design ........ 183
Steve Finkelstein ........ 209
■Smart In-Application  E-mail with Core Data and Three20 ........ 211
Florian Pflug and Peter Honeder ........ 247
■How iTap Tackles the Challenges of Networking ........ 249
Jonathan Saggau ........ 277
■Fake It 'Til You Make It: Tips and Tricks for Improving Interface Responsiveness ........ 279
Joe Pezzillo ........ 311
■Demystifying the Apple Push Notification Service ........ 313
Noel Llopis ........ 345
■Environment Mapping  and Reflections with OpenGL ES ........ 347
■Index ........ 365

# Contents

■Contents at a Glance .... iv

■Contents .... v

■Foreword .... xi

■About the Technical Reviewer .... xii

■Preface .... xiii

Ben Britten Smith .... 1

■CHAPTER 1: Everything You Ever Wanted to Know About Particle Systems .... 3

- Adding Life to Your Game with Particles .... 5
- Basic Particle Systems and You .... 7
  - Overview of the Sample Code .... 8
  - Basic Game Flow .... 9
  - The Anatomy of a Particle System .... 10
  - Code! Finally! .... 12
  - Slight Tangent About Degenerates .... 15
  - Back to the Code .... 16
  - Random Numbers and Initial Conditions .... 19
  - Emitting Particles .... 20
- Tweaking Your Particle System .... 21
  - May the Force Be with Your Particles .... 25
- Amazing Technicolor Dream Particle .... 28
  - Off on a Tangent: Lerping .... 28
  - Color-Changing Particles .... 30
- Summary .... 35

**Joachim Bondo** 37

■ **Chapter 2: Chess on the 'Net: Correspondence Gaming with Deep Green** 39

Deep Green, an Already Awesome Application 40
The Tasks at Hand 42
 Inviting a Friend to a Game 43
 Accepting the Invitation 43
 Making a Move 43
 Getting Notified 43
The Tools of the Trade 44
Stop Talking, Start Coding! 45
 Installing the Tools 45
 Coding the Web Service 47
 Accepting the Challenge on the Device 54
 Making a Move 57
Summary 61

**Tom Harrington** 63

■ **Chapter 3: Audio Streaming: An Exploration into Core Audio** 65

Hey, I Could Write an App to Play Music 66
MPMoviePlayerController: Hey, This Is Easy! Right? 66
Finding a Better Approach 68
 The System-Sound Way 69
 AVAudioPlayer: The Not-Available-in-Beta Way 69
Doing It the Cowboy Way with Core Audio 74
 Getting Halfway There: Audio Queue Services 74
 Getting the Rest of the Way There: Audio File Stream Services 81
Putting It All into an App 93
 One More Thing 93
Launch It! 96
iPhone 3.0 and Further Work 96
Summary 97

**Owen Goss** 99

■ **Chapter 4: You Go Squish Now! Debugging on the iPhone** 101

Assumed Knowledge 102
Objective-C vs. C and C++ 104
While You're Writing That Code 105
 Custom Asserts 105
 Custom Logging 107
 Using #define 108
Crash! 109
 Getting a Crash Log from Your Testers 109
 You Have Been Saving Your dSYM Files, Right? 110
 Symbolicating a Crash Log 110

Using atos ....................111
Reproducing Rare Crashes ....................112
Thread ....................112
System ....................113
Race Conditions ....................113
The Scientific Method of Debugging ....................113
Forming a Hypothesis ....................113
Creating a Test for Your Hypothesis ....................114
Proving or Disproving Your Hypothesis ....................115
Increasing the Probability of the Crash ....................115
So, You Have a Call Stack ....................115
Starting Code ....................115
What Is a Memory Stomp? ....................118
Identifying a Mem Stomp ....................122
Tools to Detect Memory Problems ....................123
Watching Variables ....................131
Link Map Files ....................135
Summary ....................137

**Dylan Bruzenak ....................139**

**Chapter 5: Building Data-Driven Applications with Active Record and SQLite ....................141**

A Short Road Off a High Cliff (How I Got Here) ....................141
Ready! Set! Wait, What? (Why I Decided to Write a To-Do Application) ....................142
Data-Driven Applications on the iPhone ....................143
Active Record: A Simple Way of Accessing Data ....................144
Writing a Database Wrapper Around the C API: ISDatabase ....................144
Setting Up the Example Project ....................145
Creating and Initializing the Database ....................148
Opening a Database Connection ....................149
Making Simple Requests ....................152
More Advanced SQL ....................158
Preventing Duplicate Create Statements ....................158
Handling Parameters ....................160
Refactoring and Cleanup ....................162
Grouping Statements into Transactions ....................163
Writing a Simple Active Record Layer: ISModel ....................164
Maintaining the Database Connection ....................165
The Model Object: Grocery Item ....................165
How Groceries Are Mapped ....................166
Saving ....................168
Updating ....................170
Deleting ....................170
Finding Grocery Items ....................171
Putting It All Together ....................174

Simple Migration Handling ........ 176
Alternative Implementations ........ 179
Summary ........ 180
**Ray Kiddy ........ 181**
**■Chapter 6: Core Data and Hard-Core Design ........ 183**
Where Did Core Data Come From? ........ 184
The Client Is King ........ 184
A Very First Core Data App ........ 185
First, Steal Code (Not Music!) ........ 186
A View to an Object, Any Object ........ 187
Our Very First Crash, or Perhaps Not! ........ 193
CoreData Tutorial for iPhone OS: Managing Model Migrations ........ 194
The Easy Migrations Are Easy ........ 194
Adding a New Entity ........ 197
Using Key-Value Coding to Create a Reusable Object ........ 199
Remote Databases: It's All Net! ........ 203
Summary ........ 206
**Steve Finkelstein ........ 209**
**■Chapter 7: mart In-Application E-mail with Core Data and Three20 ........ 211**
Planning a Simple Offline SMTP Client ........ 212
Creating the User Interface ........ 213
Diving into Xcode ........ 213
Setting Up Instance Variables in OfflineMailerAppDelegate.h ........ 215
Initializing the UIApplication Delegate ........ 217
Working with Core Data ........ 218
Understanding the Core Data Stack ........ 221
Adding Three20 ........ 221
Journeying Through the User Interface ........ 224
Managing Top-Level Data with DataManager ........ 226
Diving into Three20 and TTMessageController ........ 228
Composing and Sending Messages ........ 230
Creating the Core Data Model ........ 235
Hacking SKPSMTPMessage to Support Threaded Message Sending ........ 239
Setting Up the NSRunLoop on SKPSMTPMessage ........ 239
Switching the Bits Back to Online Mode ........ 241
Summary ........ 244
**Florian Pflug and Peter Honeder ........ 247**
**■Chapter 8: How iTap Tackles the Challenges of Networking ........ 249**
Meet iTap and iTap Receiver ........ 250
iTap ........ 251
iTap Receiver ........ 251
How the Idea for iTap Emerged and Evolved ........ 252

The Main Challenges ....252
No Physical Buttons on the iPhone ....252
Third-Party Applications Cannot Use USB or Bluetooth ....253
Supporting Both Mac and PC ....254
User-Friendliness Demands Autodiscovery of Computers and Devices....255
WiFi Networking on the iPhone from a Programmer’s Perspective ....255
About the Sample Code....256
Introducing Sockets ....257
Creating a Socket....258
Using CFSocket to React to Networking Events....262
Querying the Network Configuration....264
Contacting All Devices on the Network....267
Detecting WiFi Availability....268
Playing by the Power Management Rules....269
The Networking Subsystem of iTap ....271
To use Bonjour or Not to Use Bonjour ....271
Using Notifications to Communicate Between Components....272
Our Custom Autodiscovery Solution....273
Summary....275

**Jonathan Saggau....277**

**■Chapter 9: Fake It ’Til You Make It: Tips and Tricks for Improving Interface Responsiveness ....279**

Plotting of Historical Stock Prices with AAPLot....280
Storing Data Between Runs ....283
Using Plists to Persist Data ....284
Saving Data to the iPhone Application Sandbox ....285
Shipping AAPLot with Placeholder Data....286
Extending the App for Multiple Stock Graphs: StockPlot ....288
Concurrency ....292
NSOperation, NSOperationQueue, and Blocks ....293
Installing the Plausible Blocks Compiler and Adding It to the Project....294
Using Blocks, NSOperation, and NSOperationQueue in StockPlot ....295
Displaying Large Amounts of Data Efficiently ....298
Zooming a UIScrollView ....300
UIScrollView Zooming Under the Covers....300
Resetting Resolution in a UIScrollView after a Zoom Operation....301
Drawing into an Off-Screen Context ....304
Observations, Tips, and Tricks....309
Summary....310

**Joe Pezzillo ....311**

**■Chapter 10: Demystifying the Apple Push Notification Service....313**

What Is the Apple Push Notification Service? ....314
What You’ll Need....314

Step 1: Create the Client ....... 314
The Application Delegate ....... 315
Handling Incoming Notifications ....... 317
Sounds ....... 318
Build and Go! Er, Not So Fast... ....... 318
Step 2: Create the Certificate ....... 319
A Walk-Through of the Program Portal Process ....... 319
Back to the Portal ....... 328
Add the Mobile Provisioning File for Code Signing ....... 329
Step 3: Set Up the Server ....... 331
A Walk-Through of What This Script Does ....... 333
Download Server File ....... 334
The Home Stretch ....... 336
Wiring Up the Client ....... 336
Additional Considerations/Advanced Topics ....... 341
Feedback Server ....... 341
SSL Server Connections ....... 342
Moving from Development Sandbox to Production ....... 342
Development vs. Ad Hoc ....... 343
Mobile Provisioning Files ....... 343
Debugging ....... 343
User Experience ....... 343
Open Source Code ....... 344
Hosted Solutions ....... 344
Summary ....... 344

**Noel Llopis ....... 345**

**Chapter 11: Environment Mapping and Reflections with OpenGL ES ....... 347**

The Beginnings ....... 347
First Steps: OpenGL Lighting ....... 349
Turning to Environment Mapping ....... 352
Spherical Environment Mapping Implementation ....... 353
Combining Environment Mapping and Diffuse Textures ....... 356
Per-Pixel Reflections ....... 359
iPhone 3GS ....... 362
Summary ....... 363

**Index ....... 365**

# Foreword

Dear Readers,

We started this series of iPhone Projects books because we recognized that there is a community of iPhone developers all starting from scratch and full of enthusiasm for Apple's iPhone and iPod touch devices. The community has come a long way since we became aware of this phenomenon. For one thing, we're not all starting from scratch anymore, and this book, as does every book in this series, highlights the work of the more experienced among us.

But this enthusiasm remains a defining characteristic, along with an eagerness to learn and a willingness to share. If we were Homeric storytellers, this would be our Trojan War, an image I find particularly apt in this time of renewed gaming interest. And like the ancient poetic bards, we have some compelling stories to tell. Though, rather than warriors with shields and spears, these are tales of developer derring-do.

Our heroes are the quietly toiling, Internet-connected, basement-dwelling developers who are the stuff of iTunes App Store lore. We'll leave the modern-day mythology, Hollywood sound tracks, and CG animation to the finished applications. The chapters in this book are real-life stories of highly caffeinated work, relatively sweat-free code adventurers who dare to push the limits of a cool, little, pocket-sized, life-changing pair of devices known as the iPhone and the iPod touch. It's a dirty job, but somebody has to succeed at it.

I have worked with Dave Mark, the series editor and author of several best-selling Apress books, including *Beginning iPhone 3 Development*, to find developers who produce efficient and bug-free code, design usable and attractive interfaces, and push the limits of the technology. Dave's common-man touch, tell-it-like-it-is sense of reality, and delight at all that's cool and wonderful can be felt throughout the series.

And that brings us back to the unique quality of community among iPhone developers. Every chapter is written by a different developer with their own goals and methods, but they're all willing to share what they've learned with you. And you'll learn many things about the design and implementation of great apps, but you'll also learn that you are not alone. Every developer gets stuck, has a bad day, and experiences delays and frustrations, and the lessons learned from these setbacks are as important as the API calls and algorithms that will be part of your finished products.

And finally, we hope you'll find the apps presented in these chapters and the stories of how they came to be both interesting as human drama and as cool as the iPhone and iPod touch themselves. Happy adventuring, and send us a postcard!

Clay Andres
*Apress Acquisitions Editor, iPhone and Mac OS X*
*clayandres@apress.com*

# About the Technical Reviewer

Glenn Cole has been a professional software developer for nearly three decades, from COBOL and IMAGE on the HP 3000 to Java, Perl, shell scripts, and Oracle on the HP 9000. He is a 2003 alumnus of the Cocoa Bootcamp at the Big Nerd Ranch. In his spare time he enjoys taking road trips, playing frisbee golf, and furthering his technical skills.

# Preface

Getting started with iPhone application development is relatively easy thanks to online tutorials and especially to books like *Beginning iPhone Development* by Dave Mark and Jeff LaMarche. But sometimes, software is just hard.

A year and a half after receiving an iPhone as a birthday present, I am still amazed. It looks so simple and it's so easy to use, but behind it all is a world of complexity.

Apple has worked very hard to document the myriad APIs that make up the iPhone SDK and to provide sample code, but for some of us it's still not enough. Even Apple cannot afford to provide a chapter's worth of explanation for each sample application. Their tutorials can be quite helpful, such as the one on Core Data, but what then?

Enter *iPhone Advanced Projects*.

Ray Kiddy, who worked at Apple for 15 years in various roles, uses Apple's tutorial on Core Data as a starting point and builds from there. More than providing just an introduction, Ray shows what it's like to use Core Data in the real world.

That's the difference between documentation and a book such as this. Of course, it doesn't stop there.

Joachim Bondo, creator of the much-lauded chess application Deep Green, shares his advice and techniques for implementing correspondence gaming.

Noel Llopis, a ten-year veteran of the gaming industry, author of *C++ for Game Programmers*, and instructor of a two-day intensive class in OpenGL programming specifically for the iPhone, lends new meaning to making your application "shine" with a discussion of reflections and environment mapping in OpenGL. I found it to be a fascinating topic.

My knowledge of OpenGL is casual at best, but Ben Britten Smith provides such a clear explanation of particle systems (think smoke and fire) that this was not a hindrance at all. The chapter really was a "blast" to work through.

I've been on a private mailing list with Jonathan Saggau for several years now, and his explanations never fail to impress. Here, he discusses the difficult topic of improving interface responsiveness. (Be sure to have a copy of his sample code handy!)

And that's just the half of it! The projects also include an exploration into Core Audio, a framework for persisting data with SQLite, strategies for networking, techniques for debugging, the Apple Push Notification Service (not for the faint of heart), and intelligent in-app e-mail.

Sometimes, software is hard. With these authors as your guides, it should make your work quite a bit easier.

## Organization

This book is organized roughly in order of challenge, not necessarily according to the complexity of the code as much as the total level of knowledge and effort required.

For example, the Cocoa code that is needed to support the Apple Push Notification Service (APNS) is fairly brief and straightforward, yet the discussion of APNS does not appear until near the end of the book. Why? The primary reason for this is the complexity of the surrounding infrastructure, including working with the iPhone Developer Program Portal and setting up a PHP server appropriately.

Of course, every developer has their own ideas about what is difficult or challenging and what is not, so the chapter sequence is intended only as a rough guide. Each chapter is independent of the others, so feel free to jump straight to your projects of interest.

## What's in the Book

The book opens with Ben Britten Smith discussing particle systems using OpenGL. Although it's not a tutorial on OpenGL per se, Ben provides enough background and detail so that the code makes sense at a conceptual level even to those of us with only minimal experience in that area. Take your time in understanding this chapter and the sample code behind it, and the effort will be well rewarded. Besides, it's great fun!

Chapter 2 finds Joachim Bondo demonstrating how to implement correspondence gaming such as with his chess application Deep Green. You'll see the power of Python in Google App Engine, understand RESTful web services, implement a custom URL scheme (to support a URL beginning with `chess://`), and use Django's template engine to take advantage of a plist with embedded logic and variable substitution. It's a mouthful, but Joachim makes it look easy.

Audio is one of those topics that's just plain hard. Different requirements mean different APIs; it doesn't take much to become overwhelmed by the complexity. In Chapter 3, Tom Harrington shares the results of his investigation into processing audio streams, starting with the Media Player framework and moving to System Sound Services and the AV Foundation framework before settling on Core Audio. Audio is hard; take advantage of Tom's guidance.

Every iPhone developer who has written a nontrivial application has experienced a difficult-to-find bug. In Chapter 4, Owen Goss provides advice that goes well beyond using `NSLog()` and stepping through the debugger. You'll want to work through this chapter more than once to be sure you recognize which tools to use and when.

Dylan Bruzenak tackles data-driven applications in Chapter 5 with SQLite and the Active Record design pattern. Enterprise and cross-platform developers in particular will benefit from this, as will anyone who wants to keep fine-grained control over the data in their application.

Core Data is new to the iPhone with OS 3.0. It takes the task of data persistence to a seemingly magical level. (At least that's how I first experienced it on the Mac side.) In Chapter 6, Ray Kiddy guides us from Apple's tutorial on Core Data to its proper use in the real world, highlighting issues that can occur along the way and showing how to avoid them. Core Data is a big deal; you'll want to work through this chapter more than once.

In Chapter 7, Steve Finkelstein combines two open source projects with Core Data to build an intelligent offline email client. It recognizes when the network status changes and uses `NSInvocationOperation` to keep the user interface responsive while performing other operations. When sending e-mail, control stays within the application.

Peter Honeder and Florian Pflug get down to the socket level for networking in Chapter 8. In addition to discussing the ins and outs of communicating with devices on the network, they also discuss both power management and the trade-offs between using `SCNetworkReachability` for detecting a Wi-Fi network vs. rolling their own autodetection code.

An unresponsive user interface is one of the most frustrating behaviors an application can exhibit. In Chapter 9, Jonathan Saggau demonstrates techniques that can be used to address this. From `NSOperation`/`NSOperationQueue` to "blocks" (part of Snow Leopard but currently available on the iPhone only via Plausible Blocks) to drawing into an off-screen context and more, this chapter is very enlightening.

Joe Pezzillo provides step-by-step guidance for setting up APNS in Chapter 10. As Joe notes, the process is not particularly difficult, but it is lengthy and involved, and that's just for the creation of the distribution certificate. The Cocoa code is almost anticlimactic.

The book concludes with a fascinating chapter by Noel Llopis on environment mapping and reflections using OpenGL. You'll get more out of the chapter if you first brush off your linear algebra text, but there is still much to be learned even without it. This is the kind of polish that iPhone users love to see.

You can see that this book is packed with projects that are both relevant and interesting. Take advantage of the authors' knowledge to help your application stand above the rest!

Glenn Cole

# Ben Britten Smith

Company: http: // benbritten.com

Location: Melbourne, Australia

Former Life As a Developer: I have been writing software in one form or another since gradeschool. Back then I wrote in BASIC and Logo. Over the intervening quarter century or so I have ranged all over the map, from writing low level assembly for embedded systems through all the major (and not so major) languages settling now and again on the big ones, like C, C++, Perl, Smalltalk, Obj C, PHP, etc.

Somewhere along the way I got involved with a visual effects company called Spydercam, and wrote their industrial motion control system. This system is still in heavy use and is used on many feature films. Then in 2005, Spydercam's lead hardware designer, lead mechanical engineer and I were awarded an Academy Award for Technical Achievement for our efforts in 3D motion control. Some interesting trivia: the system we designed is the only one that I am aware of that runs on a mac, written entirely in native Cocoa/Obj-C.

I am also active in the Multi-touch surface open source community. I wrote an open source tracker called BBTouch and an open source OSC implementation called BBOSC.

Life as an iPhone DevelooperMore recently I have relocated from New York City to live in Melbourne with my wife Leonie. Here I have started offering my services as a freelance cocoa developer, and once the SDK became public, the market for iPhone work exploded. I have worked on a half dizen apps that are on the store now for various clients, titles like SnowDude, Blackout and aSleep. More recently I have begun collaborating on games of my own design, we just finished one: SnowFerno. I am currently in development on a follow-on from

***SnowDude called SkateDude, and a third as yet unnamed Dude project.  After those are done I have two more collaboration projects that are in pre-production, both games and both 2D platformers.***

***Key Technologies:* *Three or four key technologies discussed:***

- OpenGL
- Texture Atlases
- Particle Systems
- Cool Stuff

Chapter 1

# Particle Systems: More Fun and Easier Than You Think

When I was hired to write SnowDude, my employers, the Lycette Bros., and I set out a simple goal: we wanted a nice, clean, simple game that was easy to pick up and fun to play. There was not a big budget, so simplicity was the rule of the day.

I initially built the game using Core Animation, thinking that would be the quickest and easiest route to getting our 2D graphics onto the screen. In our early prototypes, this worked great; however, as we began adding the background elements and all the little graphic bits that made the game come alive, our performance crashed. I was forced at this point to reengineer the game model with OpenGL as the rendering API. This gave us all the performance we needed, and that micro game engine became the basis for many future projects in OpenGL on the iPhone.

SnowDude was a successful project in our eyes; it didn't break any App Store sales records, but the game was stable, clean, simple, and fun. (Go buy it!) The game was a lateral move for all the parties involved. I had built simple games in the past, but the bulk of my experience is in real-time motion control systems for feature films. The Lycette Bros. came from the world of Flash games and developing apps for other mobile platforms, so SnowDude was not just a game app but a way for everyone involved to dip their toes into a new platform.

Since then, I have gone on to develop a dozen or so apps for various clients and have released my first personal project to the app store: SnowFerno, which is a puzzle game where you take on the persona of a snowball trying to roll its way through hell.

And now, a bit less than a year after the original SnowDude was released, there is interest in a spin-off (or two), and we are starting to build the first one: SkateDude.

SnowDude was ultimately a fast-paced maze game. You are a snowboarder, and your goal is to get as far as you can down the “slope,” avoiding various obstacles along the way. You can avoid the obstacles by either jumping over them or boarding around them. If you make it to the checkpoint, you get some bonus time, and you can play for a higher score.

As far as programming complexity, SnowDude was not very. It consists of just a handful of textured quads, some clever use of the accelerometer, simple collisions, and some game logic.

When we all came to the table to start talking about SkateDude, we wanted to make it be a more active game experience. We wanted the obstacle avoidance to be only a small part of the game play. We decided to add tricks that you can do while in the air and a more robust control system. We added many more options to earn points, such as grinding along hand rails or park benches and doing multipart tricks like jumping onto a rail, grinding along it, and then jumping off and doing a trick before landing. All of these options add a sense of excitement and give the players an opportunity to feel the thrill of conquering the challenges.

One thing that we hadn’t nailed down in the early development meetings was how to visually enhance the game. We didn’t know how we would use the stunning graphics that the artist was generating to help bring the challenges alive and add a sense of accomplishment to the game play.

We started playing around with adding particle systems to the game. At first, I just added some very subtle sparks that shot out from under the skateboard when the player was grinding across something. This encouraged me to add a few more things. And then I added a few more systems and then a few more. I added a particle system to the controls so that if you hit a big jump, the button exploded in a shower of stars. I added a bunch of sparks that shot off the place where you touched the screen to do a jump. I added particles everywhere! Well, that was great and added lots of exciting elements, but I did go a bit far, and we ultimately scaled back to a few simple systems that added some fun and encouraged the players to want to grind and do tricks by rewarding them not only with points but with a fun visual system where a bubble with point values would shoot out from under the board like sparks and float up to join the score at the top of the screen.

This made the game much more visceral. Now, when you jump and grind across the various surfaces and edges in the game, you can visually see the points you are racking up, and the faster you grind or the higher your trick, the more points you get, so the particle systems that are shooting point bubbles out are exploding at the higher levels. Figure 1-1 is an early development screenshot of SkateDude; you can see the sparks coming off the skateboard trucks as well as the point indicators shooting out as you grind.

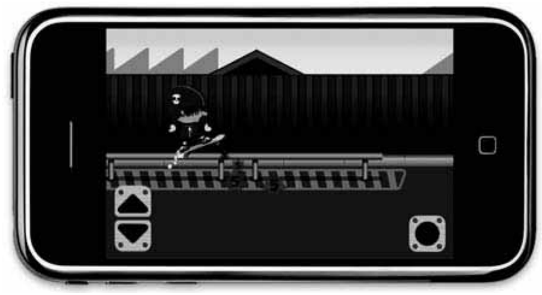


**Figure 1-1.** *An early development screenshot from the game SkateDude by the Lycette Bros. This shot shows two of the particle systems I added to make the game more exciting and visceral.*

## Adding Life to Your Game with Particles

For the rest of the chapter, I'll go over particles and how you can use them in subtle and not-so-subtle ways to add life to your games. I'll show you how to build your own particle emitter system in OpenGL and incorporate it into your own projects.

First, what is a particle system, and why would you want to use it? A *particle system* is a large collection of small graphics (you guessed it, those are the particles) that when taken as a whole can simulate effects that would otherwise be very hard to render. Things like smoke and fire are good examples. Particles are particularly good at simulating systems that are inherently dynamic and ever-changing.

Fire is a good example. You can simulate an OK fire with an animation, but it will always have a cartoonish look. If you want a fairly decent simulation of fire, you will want to use particle systems.

SnowFerno is a good example. Given that you are a snowball in hell, we mostly use particles to simulate just fire and smoke effects (see Figures 1-2 and 1-3). But fire and smoke are not the only things you should think about simulating with particle systems.

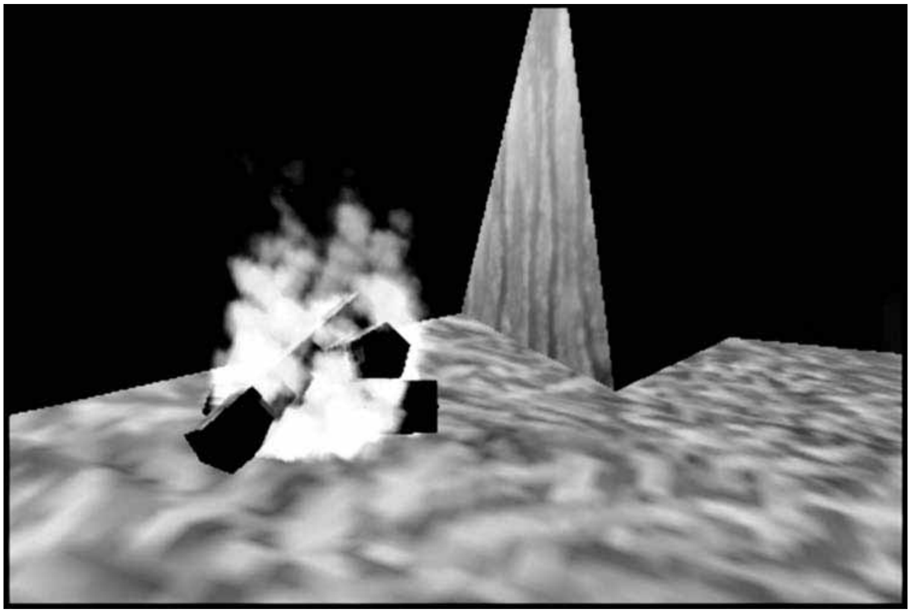


**Figure 1-2.** *A simple fire and smoke effect using particles. This is one of the particle systems in SnowFerno.*

**Figure 1-3.** *SnowFerno was set in Dante's Inferno, so we had plenty of opportunities to use fire effects.*

Particles are often associated with 3D games where the environments are immersive and players expect things such as realistic weather effects and smoke and fire and splattering blood and explosions. The list goes on and on. You can achieve all of these effects with particles.

However, it is also good to think about particles when designing your 2D apps as well, and not just 2D action games either. I often play some puzzle games to pass the time, such as Drop7 and AuroraFeint. Both of these use particles to add a bit of excitement and life to the game. In Figure 1-4, you can see the block-smashing effect in Aurora Feint.

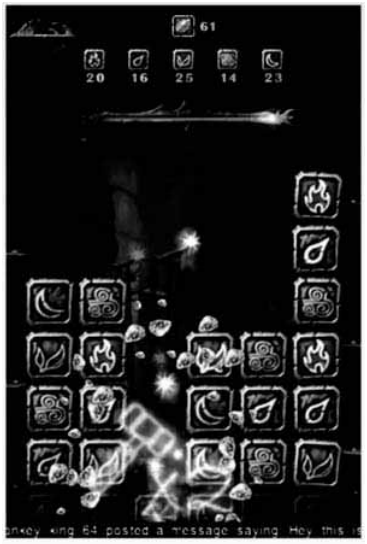


**Figure 1-4.** *Aurora Feint uses particles to make its block smashing exciting.*

Particles do not need to be big flashy things; they don't have to be grand explosions or giant fireballs. You can add subtle fun touches to your game interface with some simple effects as well. Drop7 does this well; when you "crack" one of the unknown numbers, it breaks open with a simple particle effect. It is so subtle that you might not even notice it, but it adds that bit of life and personality that makes the game fun. When you set up a nice long chain reaction, all those little particle explosions really make it that much more satisfying.

## Basic Particle Systems and You

OK, now you know where you can add particle effects to your games, so now let's talk about how to add them.

First, I will presume you have some familiarity with OpenGL. If you don't know OpenGL, that is fine; you can still do particles in Core Animation and Core Graphics, so much of the conceptual stuff will be applicable. However, OpenGL excels at things like particle systems because it is so good at moving textures onto the screen very fast. In a Core Animation particle implementation, you might be able to get a particle system with a few dozen particles, maybe even 100 for a short while. With OpenGL, you can generate thousands of particles at once, even on the iPhone.

## Overview of the Sample Code

The sample project, called Particles, started its life as a generic OpenGL project template from Apple. I have added a simple game harness around Apple's template code. Originally this code was written for the *Beginning Game Development for iPhone*, and the chapters I wrote in that book go into great detail about this code base. Most of the implementation details are not that important to the discussion of particle systems, but I will do a brief overview anyway.

Let's take a look at the basic design:

- `EAGLView`: This is a modified version of the `EAGLView` you get when you start a new Xcode OpenGL iPhone project. It is responsible for OpenGL buffer swapping as well as most of the boilerplate OpenGL initialization stuff. This is the main view for the application.
- `SceneObject`: This is the base class for anything in the game. It has the basic instance vars that most everything that needs to be rendered needs. All rendered objects inherit from this class.
- `SceneController`: This is the main controller for the game. It handles the game loop. It has a single `SceneObject` that is the root of all objects in the current scene. It is a singleton.
- `InputViewController`: Since the input and the main view are basically the same thing, this view controller handles the `EAGLView` as well as wrangling the touch events. The input controller has its own list of scene objects that get rendered last, in a heads-up display style.
- `RenderController`: This object deals with rendering all the scene objects. It performs simple culling. The render controller uses a `SceneObject`'s mesh to render that object. The mesh is basically the collection of all the vertex data for a particular model.
- `MaterialController`: This object handles the loading of textures into OpenGL. It can handle single textures or atlases when accompanied with a `.plist` file describing the atlas contents.
- `GameTypes`: This is just a big collection of structs and inline functions that come in handy. The two types I use the most in the sample code are `BBPoint`, an xyz point struct, and `BBRange`, a range of floats.

The reason that I am not just showing how to build a stand-alone particles project is that I think it is important to think about how these things fit into the bigger picture. Although the sample program does little more than show off some particle effects, it is important to think of these concepts in the context of a larger application.

The Particles sample project is not a fully realized game engine by any stretch, but it is a good place to start, and it has much of what you would need to build a simple 3D application in OpenGL. This makes it a good platform for you to explore the concepts of particle systems.

## Basic Game Flow

Figure 1-5 shows the flow for the game harness. It follows the basic game design pattern that you are probably familiar with.

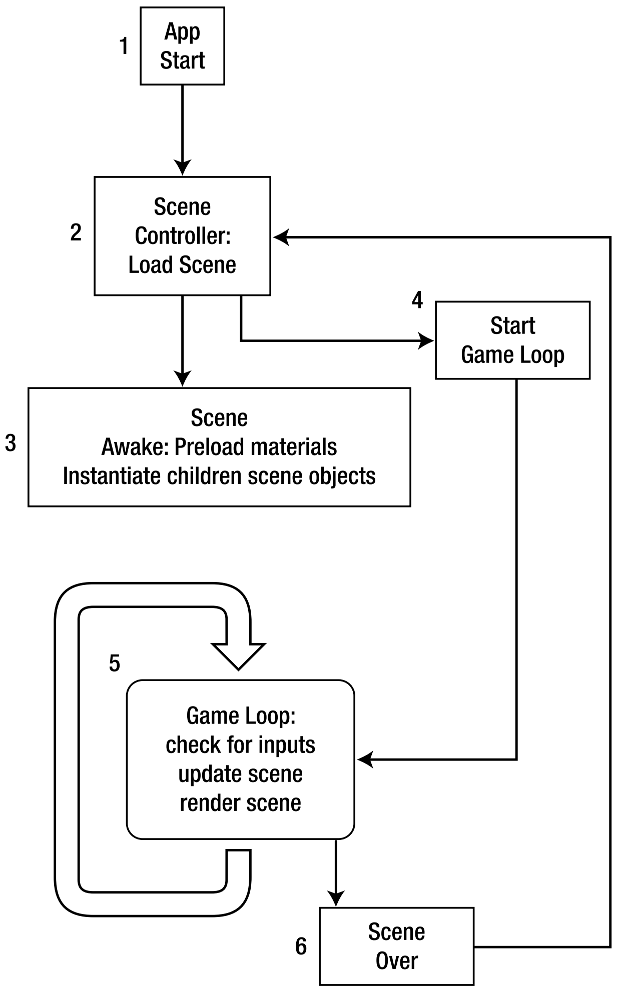

1
App Start
2
Scene Controller: Load Scene
4
Start Game Loop
3
Scene Awake: Preload materials Instantiate children scene objects
5
Game Loop: check for inputs update scene render scene
6
Scene Over

**Figure 1-5.** *This is the basic flow for the game harness.*

After the app starts up and everything is loaded from the xib files and you are ready to go, the `SceneController` is called upon to load the first scene. This scene is simply a `SceneObject` that is the parent of all the objects you want to have interact for this scene.

After the scene is "alloced," the method `awake` is called on it, and that is where the scene will call out to the other support objects, like the material controller, to make sure that all the resources for this scene are loaded. (In this case, this will generally just be textures, but in the broader case, this might include sound files or game data of some sort.)

When everything is ready, the game loop is started.

The game loop first checks for inputs, and then it calls `update:` on the scene. The scene object will update all of its children recursively until the entire scene model has had a chance to update its state. Finally, the game loop passes the root scene object to the renderer to be rendered. Then it starts all over again.

At some point in the scene, the update portion of the loop will generate an end-of-scene notification. (Maybe your character died, you ran out of time, or you hit a button to move on to the next scene...whatever) The current scene is unloaded, and the next scene is loaded.

This is a fairly standard game engine design. The big component that's missing here is a collision detection system. You will do some simple collision stuff with the particle systems but nothing too complicated.

## The Anatomy of a Particle System

Just in case you have never come in contact with a particle system, I will start with the basics: what exactly constitutes a particle?

Particles can be any texture and are usually rendered as a textured quad (two triangles). Depending on the effect you are going for, your particle textures might be semitransparent like the simple white particle in Figure 1-6. Soft semitransparent particles will yield "fuzzy" effects quite well. This makes a nice effect because particles in a high concentration will be brighter and more intense, whereas out on the edges where there may be only a few particles, the overall effect is dimmer and "blurry."

**Figure 1-6.** *A simple particle texture. This is about the simplest semitransparent texture you can get. It is just a white blur, 25 X 25 pixels.*

That said, you can get some great effects from fully opaque or hard-edged particles as well, such as things like marbles rolling across a floor or leaves falling.

Each particle in the system has its own state, and each particle will get its own initial conditions and then behave based on a set of rules. All of this ordered chaos—a multitude of particles that are all slightly different but similar—can create some amazing fluid, living, organic effects.

That is the particle. You also need  something that generates the particles, and that is known as the *emitter*. The emitter's job is to build new particles at some predetermined rate. It has to assign each particle an initial state that meets the requirements for that particular effect. These are things such as starting position, size, life span, speed, and direction. After a particle has been created, the emitter then has to keep track of each particle, and for every rendered frame, it needs to collect all the vertex and UV and any other rendering data for each particle and build some big arrays to send off to the renderer.

In many particle effects, each particle has a life span, and once that span is over, the emitter needs to collect those particles and remove them from the scene.

So, basically the emitter itself has a mini game loop going on. Every time it gets updated, it needs to create some new particles and add them to its currently active particle list. Then it goes through all the active particles and moves them or rotates them or whatever. Then it needs to check to see whether any particles have reached the end of their life, and if so, it removes them from the active list. Finally, it needs to make the data arrays from all the particle states.

Here are a few things to keep in mind:

- The particle system needs to be able to go through thousands of particles in a single frame, so you need to find efficient ways to handle all of the particles and keep them updated.
- The emitter may need to emit a few hundred particles every frame, possibly even a few thousand, so you also need to be very efficient about creating particles. Allocing objects is a costly process, so you want to avoid it at all costs.
- Hundreds of particles can expire at the same frame, so you need to also be clever about how you clean up your particles. Memory cleanup is slow and can affect performance, so you need to be careful about releasing a zillion particles all at once.
- Dynamically mallocing vertex array memory is expensive. You want to avoid changing the size of your vertex data arrays.

How do you solve these problems?

When your particle emitter is first created, you will need to build a big reserve of prealloced particle objects. Similarly, you will malloc a big chunk of memory for your vertex data arrays, big enough to hold the maximum number of particles.

Then during the update loop, when you emit new particles, you just grab them out of the pool and assign them their initial state. This is so much faster than allocing new objects on the fly. This becomes especially important for effects such as explosions where you need to emit lots of particles all at once.

When you build your data arrays for each frame, you just use as much of the vertex data space as you need and leave the rest as reserve.

Similarly, at the end of the particle life, when you clear them out of the active list, you simply return the particle objects to the pool.

Figure 1-7 shows this life cycle. Also of note: I used a particle system to generate both the spark shower and the pool.

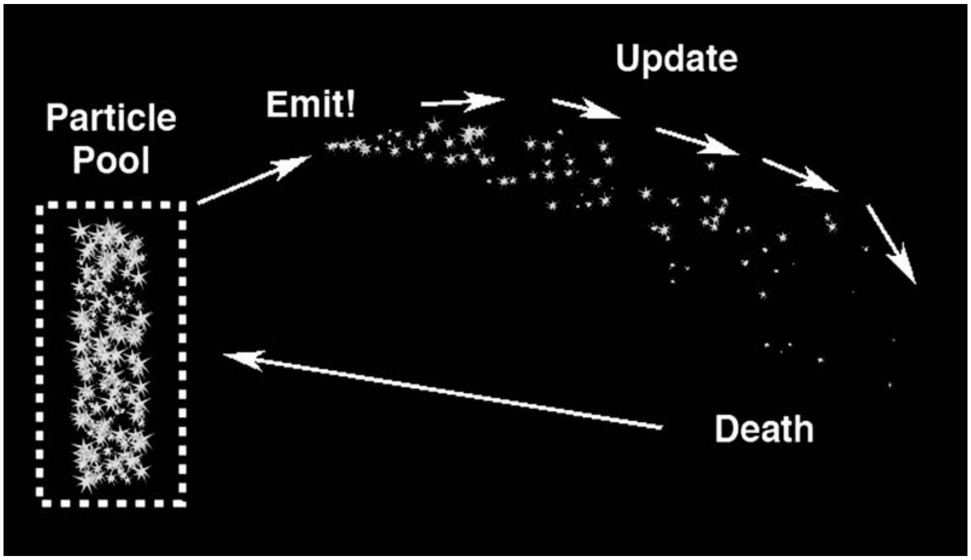


**Figure 1-7.** *The particle life cycle. Nonactive particles start in the pool. They are pulled out of the pool and given some initial state when they are emitted. They live out their exciting particle life until they finally die. They are then collected and returned to particle limbo to await resurrection.*

The downside to this method is that it can be very memory consuming, and the setup time can be significant if you have many particle systems. The secret is to tune the max particles for the type of effect you are creating. A blizzard of falling snow might require a few thousand particles, whereas a subtle foreground of falling leaves may require only a few dozen.

## Code! Finally!

OK, I have rambled on for quite a few pages about the whats and whys of particles. It is time to get your hands dirty with some code.

First build a particle:

```
@interface BBParticle : NSObject {
  BBPoint position;
    BBPoint velocity;
    CGFloat life;
    CGFloat size;
    CGFloat grow;
    CGFloat decay;
}
```

```
@property (assign) BBPoint position;
@property (assign) BBPoint velocity;
@property (assign) CGFloat life;
@property (assign) CGFloat size;
@property (assign) CGFloat grow;
@property (assign) CGFloat decay;
```

This is a very basic particle. The basic state is `position`, `life`, and `size`. `velocity`, `grow`, and `decay` are the state changers. Particles can be far more complicated than this, and you will add some more stuff to your particle later, but for now let's keep it simple.

Next you look inside your particle implementation:

```
@implementation BBParticle

@synthesize position,velocity;
@synthesize life,size,grow,decay;

-(void)update:(NSTimeInterval)deltaTime
{
    position.x += velocity.x * deltaTime;
    position.y += velocity.y * deltaTime;
    position.z += velocity.z * deltaTime;

    life -= decay  * deltaTime;
    size += grow  * deltaTime;
    if (size < 0.0) size = 0.0;
}
```

Very simple. You have a time-based update. You take all of your state and change it by a fraction equal to the amount of time for this frame. Finally, you check your size. You don't want to go into negative size because that will just flip your particle over and make it grow.

That's it! You have a nice simple model object with a single data manipulator method.

Next, let's build a simple particle emitter object. This one is a bit more complicated than the particle:

```
@interface BBParticleSystem : BBSceneObject {
    NSMutableArray * childrenParticles;

    GLfloat * uvCoordinates;
    GLfloat * vertexes;

    NSMutableArray * unusedParticles;

    NSInteger vertexIndex;

    BOOL emit;
    CGFloat emitCounter;

    BBRange emissionRange;
    BBRange sizeRange;
    BBRange growRange;
    BBRange xVelocityRange;
    BBRange yVelocityRange;
    BBRange zVelocityRange;

    BBRange lifeRange;
    BBRange decayRange;

    CGFloat minU;
    CGFloat maxU;
    CGFloat minV;
    CGFloat maxV;

    CGFloat particleRemainder;
}
```

Wow, that is a fair few instance variables! One thing that you will learn quickly (or may already know if you have played with emitters before) is that a good particle emitter will be very flexible, and that requires lots of inputs to tweak to get just the right effect. Lots of inputs means lots of instance variables.

Let's get into the implementation:

```
- (id) init
{
    self = [super init];
    if (self != nil) {
        [self preload];
    }
    return self;
}
```

That was a simple `init` method. Basically, you just call `preload`, which is where you, well, preload all your particles and memory allocations:

```
-(void)preload
{
    if (childrenParticles == nil) childrenParticles = [[NSMutableArray alloc] init];
    unusedParticles = [[NSMutableArray alloc] initWithCapacity:kMaxParticles];
    NSInteger count = 0;
    for (count = 0; count < kMaxParticles; count++) {
        BBParticle * p = [[BBParticle alloc] init];
        [unusedParticles addObject:p];
        [p release];
    }
```

First you create your particle limbo and fill it with particles ready to be jettisoned into life to burn brightly for a few moments and then be pulled back into the land of the inactive.

```
    // remember 6 vertexes per particle + UVs
    vertexes = (CGFloat *) malloc(2 * 6 * kMaxParticles * sizeof(CGFloat));
    uvCoordinates = (CGFloat *) malloc(2 * 6 * kMaxParticles * sizeof(CGFloat));
}
```

Don't forget to malloc some room for the vertexes and UV coordinates.

I'll now go off on a tangent momentarily and talk about `GL_TRIANGLES` vs. `GL_TRIANGLE_STRIP`.

## Slight Tangent About Degenerates

You are going to be drawing a whole slew of textured quads onto the screen. However, generally a quad is only four vertexes. So, what is up here?

You are going to be rendering all your particles in the same draw call, and they are not connected, so you will need to figure out a good way to draw them all.

If you use `GL_TRIANGLES`, then you are basically just draw each triangle individually. Every quad is just two triangles and six vertexes. This has the advantage of being very simple to program.

You could also use `GL_TRIANGLE_STRIP` and connect each quad with degenerate triangles. A *degenerate triangle* is a triangle where the three points lie on a line. You can see in Figure 1-8 how this works. A triangle with colinear points has no area, so the renderer will throw it out. The easiest way to connect two meshes with a degenerate triangle is to just duplicate the last vertex of the first mesh and the first vertex of the second mesh and then add them together. This basically inserts two colinear triangles into the strip so that the rendered effect is two separate quads. This means, on average, each quad requires six vertexes, just like the `GL_TRIANGLES` method.

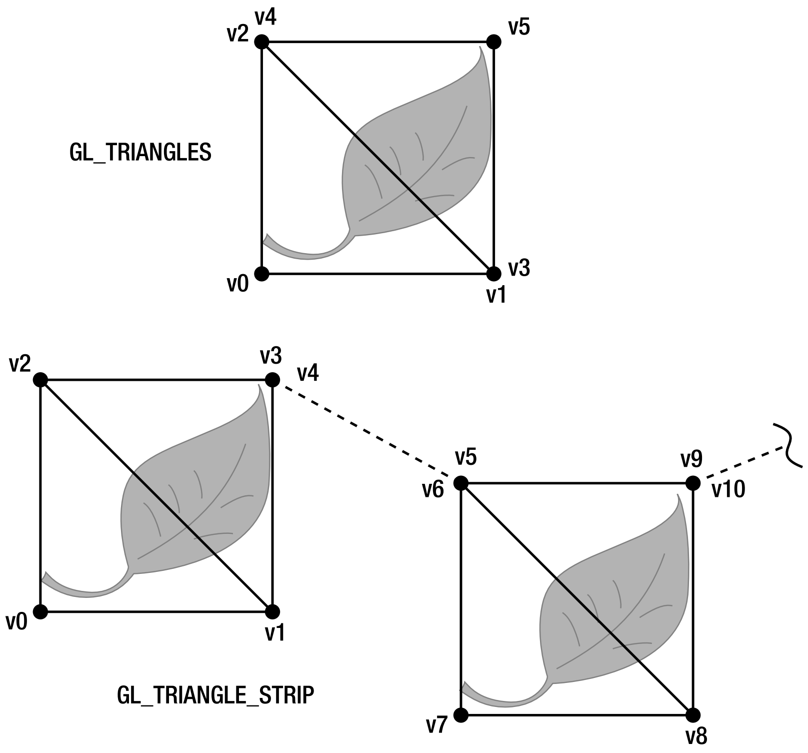


**Figure 1-8.** *With `GL_TRIANGLES`, you have two separate polygons drawn individually. With `GL_TRAINGLE_STRIP`, all the polygons are connected, so you have to basically put two degenerate triangles in between the two separate quads.*