



iOS群：335930567 (吹水勿扰)

系统讲解OpenGL ES的核心概念、技术，以及iOS的图形机制，通过大量案例讲解了在iOS上进行OpenGL ES开发的方法和技巧

PEARSON

会员书库

Learning OpenGL ES for iOS
A Hands-On Guide to Modern 3D Graphics Programming

OpenGL ES应用开发实践指南 iOS卷

(美) Erik M. Buck 著
徐学磊 译



机械工业出版社
China Machine Press

华章程序员书库

OpenGL ES 应用开发实践 指南：iOS 卷

*Learning OpenGL ES for iOS:A Hands-On Guide to
Modern 3D Graphics Programming*

(美) Erik M.Buck 著

徐学磊 译

华章图书



机械工业出版社

China Machine Press

图书在版编目 (CIP) 数据

OpenGL ES 应用开发实践指南: iOS 卷 / (美) 巴克 (Buck, E. M.) 著; 徐学磊译. —北京: 机械工业出版社, 2013.6

(华章程序员书库)

书名原文: Learning OpenGL ES for iOS: A Hands-On Guide to Modern 3D Graphics Programming

ISBN 978-7-111-42867-1

I. O… II. ①巴… ②徐… III. ①图形软件 – 指南 ②移动电话机 – 应用程序 – 程序设计 – 指南
IV. ①TP391.41-62 ②TN929.53-62

中国版本图书馆 CIP 数据核字 (2013) 第 126145 号

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

本书版权登记号: 图字: 01-2012-7590

Authorized translation from the English language edition, entitled *Learning OpenGL ES for iOS: A Hands-On Guide to Modern 3D Graphics Programming*, 9780321741837 by Erik M. Buck , published by Pearson Education, Inc., Copyright © 2013.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese Simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2013.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内 (不包括中国台湾地区和中国香港、澳门特别行政区) 独家出版发行。未经出版者书面许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签, 无标签者不得销售。

这是一本系统的具备实战性的 OpenGL ES 3D 图形开发指南。由资深 iOS 开发专家根据 OpenGL ES 最新版本撰写, 不仅详细讲解了 OpenGL ES 与 GLKit 的结合使用, 而且还系统讲解 OpenGL ES 的核心概念、技术, 以及 iOS 的图形机制, 并通过大量案例讲解了在 iOS 上进行 OpenGL ES 开发的方法和技巧。

全书共分 12 章。第 1 章介绍了使用嵌入式图形硬件绘制 3D 图形的最新方法; 第 2 章讲解了如何使用苹果 Xcode 开发工具和 Cocoa Touch 面向对象的框架在 iPhone、iPod Touch 和 iPad 中开发包括 3D 图形的程序; 第 3 章涵盖了纹理的底层概念和常用选项; 第 4 章介绍灯光模拟背后的概念, 以及利用 GLKit 并使用相对简单的应用代码演示灯光效果; 第 5 章讲解并演示从任意视点渲染几何对象的技术; 第 6 章介绍如何制作动画; 第 7 章介绍了如何加载并使用模型; 第 8 章讲解了特效的使用; 第 9 章介绍能够提高 iOS 设备上 OpenGL ES 2.0 渲染性能的优化策略; 第 10 章讲解了地形和拾取; 第 11 章回顾了 3D 渲染所需的常见数学运算; 第 12 章涵盖了一个结合地形渲染、天空盒、粒子系统、动画、变化视点、灯光、模型和碰撞检测技术的实例。

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑 : 陈佳媛

印刷

2013 年 7 月第 1 版第 1 次印刷

186mm×240mm • 17.5 印张

标准书号: ISBN 978-7-111-42867-1

定 价: 69.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzjsj@hzbook.com

译者序

OpenGL ES 是桌面版 OpenGL 的一个子集，是现代移动设备绘图功能的基础。其中“ES”代表嵌入式系统。OpenGL ES 定义了嵌入式 3D 绘图的标准。GLKit 是苹果公司的 iOS 5 中引入的一个软件框架，这个框架简化了很多常用的编程任务，隐藏了 OpenGL ES 版本间的差异，使用这个框架可以提高软件开发的生产率。本书主要讲解的是 iOS 5 中的 GLKit 软件框架，以及 GLKit 所支持的 OpenGL ES 2.0 版本。

本书从基本概念开始，全面介绍了 OpenGL ES 计算机图形的核心概念以及 iOS 的图形机制，包括地形、特效、模型、动画、视点变化、纹理、灯光和优化等。本书例子丰富有趣，一般每章都有多个例子，并且源码结构合理，注释全面。本书会告诉你如何在苹果公司的 iOS 环境中充分地利用好 OpenGL ES。本书专注于最新版 OpenGL ES 的最新使用方法，因此可以让你避开散落在互联网上的不相关的、过时的，或者误导性的方法。本书的第 11 章为数学速查章节，非常方便，如果你是计算机图形的新手，可以先阅读这一章以帮助你更好地理解本书的内容。如果你熟悉桌面版 OpenGL，但是不了解 OpenGL ES 以及 GLKit，那么这本书也非常适合你。

本书的翻译经历了 3 个月，非常感谢华章公司的编辑们在翻译过程中给予我的支持和帮助。对于译者来说，能够给读者带来切实的帮助是我最大的荣幸。虽然在翻译的过程中竭力以信、达、雅为目标，但是由于水平有限，失误和遗漏在所难免，恳请读者批评指正。

译者

前　　言

OpenGL ES 技术是苹果 iOS 设备（iPhone、iPod Touch 以及 iPad）上的用户界面和图形绘制能力的基础。“ES”代表嵌入式系统（Embedded System），这个术语适用于视频游戏机、飞机驾驶员座舱显示器，并且广泛适用于几乎所有生产商的手机。OpenGL ES 是桌面操作系统 OpenGL 版本的一个子集。因此，OpenGL ES 应用通常也适用于桌面系统。

本书介绍了最新图形编程，同时对 iOS 设备中 OpenGL ES 的有效使用做了简洁说明。书中有很多用于演示图形编程概念的例子程序。在网站 <http://opengles.cosmicthump.com/> 上保存着很多例子和相关文章。本书对于从底层位操作到高级主题的图形技术都做了详细的解释。

学习图形编程的重大挑战体现在当你第一次试图整理散落在互联网上的成堆的误导性信息和过时的例子时。最初 OpenGL 是 1992 年的最先进图形工作站中的一个小型软件库。由于图形硬件改进得频繁且更新较快，以至于现在的移动设备已经胜过 OpenGL 刚出现时能够买到的最好的硬件。随着硬件技术的提高，OpenGL 设计者当时所做的一些折中方法和假设已不再有意义了。现在至少存在着 12 种不同的 OpenGL 标准，不过最新的 OpenGL ES 省略了很多对于以前版本中的常见技术的支持。不幸的是，在谷歌的搜索结果中还存在相当多的过时的代码和次优方案，以及几十年来形成的经验。本书将主要关注最新、最高效的方法，以避免分心于过时且不相干的练习上。

读者对象

本书的读者包括学习编程的学生以及精通其他学科又想要学习图形的程序员。读者不需要有计算机图形的经验，但需要熟悉 C 或者 C++ 以及面向对象编程的概念。有 iOS、Objective-C 编程语言和 Cocoa Touch 框架的使用经验是最好的，但不是必需的。在学完本书后，你将有能力在你自己的 iOS 应用中使用高级计算机图形技术。

示例代码

本书提供的很多例子可以用作你自己的工程的起始点。可以从 <http://opengles.cosmictchump.com/learning-opengl-es-sample-code/> 下载本书例子的源代码，这些源码使用 MIT 软件许可协议：<http://www.opensource.org/licenses/mit-license.html>。

这些例子是使用苹果的免费开发者工具建立的，用的是 Objective-C 编程语言，以及苹果的 Cocoa Touch 面向对象的软件框架。OpenGL ES 应用程序编程接口（API）由美国国家标准协会（ANSI）/国际标准化组织（ISO）C 编程语言的数据类型和函数组成。作为 ANSI/ISO C 的一个超集，Objective-C 程序原生地支持与 OpenGL ES 交互。

所有 iOS 应用都或多或少地依赖于 Cocoa Touch 框架，该框架是基于苹果的 Objective-C 语言的。一些开发者通过重用现存的用 C 或者 C++ 写就的跨平台库来最小化自己的应用与 Cocoa Touch 的融合。作为 UNIX 操作系统的一个派生物，iOS 包含了标准 C 库和 UNIX API，这使得移植跨平台代码到苹果设备上变得出人意料的简单。OpenGL ES 自身的一部分也是由跨平台 C 库组成的。尽管如此，不想使用 Cocoa Touch 和 Objective-C 的开发者几乎总是给自己帮倒忙。苹果面向对象的框架史无前例地提升了程序员的生产率。更重要的是，Cocoa Touch 提供了用户对 iOS 应用所期望的平台一体性和精良性。

本书包含了 Objective-C 和 Cocoa Touch。苹果的基于 Objective-C 的 GLKit 框架的强大和简洁是如此令人信服，以至于它明确地指出了图形编程的未来发展方向。如果不使用 GLKit 而只关注操作系统的底层 C 接口和 OpenGL ES，本书几乎无法声称自己教授的是最新技术。

Objective-C

与 ANSI/ISO C 一样，Objective-C 是一个非常小型的语言。有经验的 C 程序员通常会发现，他们可以在至多几个小时内很容易地学会 Objective-C。Objective-C 在开启了一个富有表现力的面向对象的编程风格的同时，最低限度地扩充了 C 语言。本书详述了图形编程，同时在需要的时候会对 Objective-C 语言的特性做适当的介绍。你并不需要精通 Objective-C 或者 Cocoa Touch，但是你需要熟悉 C 或者 C++ 以及面向对象编程的概念。你会发现使用 Objective-C 语言实现应用逻辑是非常容易和简洁的。Cocoa Touch 经常简化应用的设计，尤其是在响应用户输入的时候。

C++

ANSI/ISO C++ 编程语言是 ANSI/ISO C 的一个不是很完美的超集，但它几乎总是

可以自由地与 C 语言混合。OpenGL ES 与 C++ 可以无缝配合，并且 OpenGL 结构审查委员会（ARB）会监督 OpenGL ES 的规范，以保证其未来与 C++ 的兼容性。

C++ 编程语言是用于图形编程的最常见的编程语言之一。但是，C++ 是一个非常大型的编程语言，充满了惯用语法和精妙法则。对于 C++ 语言要达到中等掌握水平可能要花费数年的时间。使用 C++ 做图形编程有许多优势，例如，使用 C++ 操作符重载功能可以让图形程序中数学运算的表达更加简洁。

混合使用 C++ 与 Objective-C 代码并没有任何障碍。苹果开发者工具甚至支持 Objective-C++ 形式，这种形式允许在一个语句中混合使用 C++ 和 Objective-C 代码。但是 Objective-C 是 iOS 的主要编程语言。在苹果和第三方提供的几乎所有 iOS 示例代码中你都可以发现 Objective-C 的代码。如果你想使用 C++ 也是可以的，但这超出了本书讨论的范围。

使用 GLKit 作为导向

本书通过对苹果 GLKit 的探索来讲解图形编程的最新概念。在一些情况下，某些章节会通过实现 GLKit 的部分对象来讲解和演示这些概念。这样做有几个目的：使用 GLKit 来简化启动项目所需的步骤。在第 2 章末尾你会建立起 3 个 OpenGL ES 应用，并在你的 iOS 设备上运行。一章接着一章，逐渐创建彼此关联的很多主题，最终创建一个可重用的知识和代码的基础结构。当付出努力以从零开始创建时，可以帮助你获取一个想要的最终结果的清晰概念。为了获得有价值的最终结果，GLKit 设置了一个高质量的现代基准。

本书会消除你关于怎么使用 OpenGL ES 来实现和扩展 GLKit 的所有疑云。学完本书后，你会成为一个 GLKit 专家，彻底理解 GLKit，同时拥有在你的 iOS 应用中使用 GLKit 的能力。GLKit 演示了当前对于 OpenGL ES 的最好做法，同时如果你需要，甚至可以将其作为你自己的跨平台库的一个模板。

致谢

写一本书需要很多人的支持。首先，感谢我的妻子 Michelle，以及我的孩子 Joshua、Emma 和 Jacob，谢谢你们的理解和支持。其次，感谢出版社的编辑们，他们为本书的写作提供了非常宝贵的建议。最后，感谢那些在学术上、专业上、精神上、道德上和艺术上将影响我一生的人。

目 录

译者序

前言

第1章 使用现代移动图形硬件 / 1

- 1.1 3D 渲染 / 1
- 1.2 为图形处理器提供数据 / 3
 - 1.2.1 缓存：提供数据的最好方式 / 4
 - 1.2.2 帧缓存 / 5
- 1.3 OpenGL ES 的上下文 / 6
- 1.4 一个 3D 场景的几何数据 / 7
 - 1.4.1 坐标系 / 7
 - 1.4.2 矢量 / 9
 - 1.4.3 点、线、三角形 / 11
- 1.5 小结 / 11

第2章 让硬件为你工作 / 12

- 2.1 使用 OpenGL ES 绘制一个 Core Animation 层 / 12
- 2.2 结合 Cocoa Touch 和 OpenGL ES / 14
 - 2.2.1 Cocoa Touch / 14
 - 2.2.2 使用苹果开发者工具 / 15
 - 2.2.3 Cocoa Touch 应用架构 / 15
- 2.3 OpenGL_ES_Ch2_1 示例 / 18
 - 2.3.1 OpenGL_ES_Ch2_1AppDelegate 类 / 18
 - 2.3.2 Storyboards / 19

- 2.3.3 OpenGLES_Ch2_1ViewController 类的 interface / 19
- 2.3.4 OpenGLES_Ch2_1ViewController 类的实现 / 20
- 2.3.5 支持文件 / 30
- 2.4 深入探讨 GLKView 是怎么工作的 / 31
- 2.5 对于 GLKit 的推断 / 40
- 2.6 小结 / 46

第 3 章 纹理 / 48

- 3.1 什么是纹理 / 48
 - 3.1.1 对齐纹理和几何图形 / 49
 - 3.1.2 纹理的取样模式 / 50
 - 3.1.3 MIP 贴图 / 52
- 3.2 OpenGLES_Ch3_1 示例 / 52
- 3.3 深入探讨 GLKTextureLoader 是怎么工作的 / 56
- 3.4 OpenGLES_Ch3_3 示例 / 62
- 3.5 透明度、混合和多重纹理 / 63
 - 3.5.1 在 OpenGLES_Ch3_4 示例中混合片元颜色 / 64
 - 3.5.2 示例 OpenGLES_Ch3_5 中的多重纹理 / 66
 - 3.5.3 在 OpenGLES_Ch3_6 示例中自定义纹理 / 68
- 3.6 纹理压缩 / 70
- 3.7 小结 / 71

第 4 章 散发一些光线 / 72

- 4.1 环境光、漫反射光、镜面反射光 / 73
- 4.2 计算有多少光线照向每个三角形 / 74
- 4.3 使用 GLKit 灯光 / 79
- 4.4 OpenGLES_Ch4_1 示例 / 80
- 4.5 把灯光烘焙进纹理中 / 86
- 4.6 片元计算 / 87
- 4.7 小结 / 88

第 5 章 改变你的视点 / 89

- 5.1 深度渲染缓存 (Depth Render Buffer) / 89

5.2	例子 OpenGLES_Ch5_1 和例子 OpenGLES_Ch5_2 / 91
5.3	深入探讨不用 GLKit 添加深度缓存 / 96
5.4	变换 / 98
5.4.1	基本变换 / 98
5.4.2	顺序很重要 / 101
5.4.3	projectionMatrix 和 modelviewMatrix / 102
5.4.4	textureMatrix / 105
5.5	复合变换手册 / 107
5.5.1	倾斜 / 107
5.5.2	围着一个点旋转 / 107
5.5.3	围着一个点缩放 / 107
5.6	透视和平截头体 / 108
5.7	小结 / 109

第 6 章 动画 / 110

6.1	场景内移动：例子 OpenGLES_Ch6_1 / 111
6.1.1	看向一个特定的 3D 位置 / 111
6.1.2	使用时间 / 113
6.2	动画化顶点数据 / 116
6.2.1	使用索引顶点 / 118
6.2.2	OpenGLES_Ch6_2 示例 / 119
6.3	动画化颜色和灯光：例子 OpenGLES_Ch6_3 / 122
6.4	动画化纹理 / 126
6.4.1	OpenGLES_Ch6_4 示例 / 126
6.4.2	OpenGLES_Ch6_5 示例 / 128
6.5	小结 / 130

第 7 章 加载和使用模型 / 131

7.1	建模工具和格式 / 132
7.2	读取 modelplist 文件 / 136
7.3	OpenGLES_Ch7_1 示例 / 138
7.4	高级模型 / 142
7.4.1	骨骼动画 / 142
7.4.2	蒙皮 / 147

7.4.3 逆动力学和物理模拟 / 150
7.5 小结 / 150

第 8 章 特效 / 151

8.1 天空盒 / 151
8.2 深入探讨 GLKSkyboxEffect 是怎么工作的 / 154
8.3 粒子 / 164
8.4 公告牌 / 170
8.5 小结 / 177

第 9 章 优化 / 178

9.1 尽可能减少渲染 / 178
9.1.1 基于视平截体的剔除 / 179
9.1.2 简化 / 189
9.2 不要猜：解析（Profile）/ 189
9.2.1 工具 OpenGL ES Performance Detective / 190
9.2.2 工具 Instruments / 191
9.3 尽量减少缓存复制 / 192
9.4 尽量减少状态变化 / 192
9.5 小结 / 193

第 10 章 地形和拾取 / 195

10.1 地形的实现 / 195
10.1.1 高度图 / 196
10.1.2 地形瓦片 / 197
10.1.3 地形效果 / 200
10.2 添加模型 / 205
10.2.1 模型放置 / 206
10.2.2 模型效果 / 206
10.3 OpenGL ES 摄像机 / 208
10.4 拾取 / 213
10.5 优化 / 221
10.6 小结 / 228

第 11 章 数学速查 / 229

- 11.1 概述 / 229
- 11.2 解码矩阵 / 230
 - 11.2.1 从平截体获取矩阵 / 233
 - 11.2.2 透视 / 236
 - 11.2.3 矢量的坐标轴分量 / 237
 - 11.2.4 点变换 / 238
 - 11.2.5 转置矩阵和逆矩阵 / 240
- 11.3 四元法 / 241
- 11.4 常用的图形数学 / 242
 - 11.4.1 简单矢量运算 / 242
 - 11.4.2 矢量标量积 / 243
 - 11.4.3 矢量的矢量积 / 243
 - 11.4.4 model-view 矩阵 / 244
 - 11.4.5 投影矩阵 / 245
- 11.5 小结 / 245

第 12 章 理清整体思路 / 246

- 12.1 概述 / 246
- 12.2 一切如故 / 248
 - 12.2.1 控制器子系统 / 249
 - 12.2.2 模型子系统 / 250
 - 12.2.3 视图子系统 / 255
- 12.3 设备动作 / 263
- 12.4 小结 / 265

第 1 章 使用现代移动图形硬件

本章介绍使用嵌入式图形硬件绘制 3 维（3D）图形的最新方法。嵌入式系统涵盖了范围广泛的设备，从飞机驾驶员座舱显示器到自动售货机。绝大多数具有 3D 功能的嵌入式系统都是手持电脑，比如苹果的 iPhone、iPod Touch、iPad，或者是基于谷歌的 Android 操作系统的手机。索尼、任天堂及其他的手持设备也具备强大的 3D 图形能力。

用于嵌入式系统的 OpenGL（OpenGL ES）定义了嵌入式 3D 图形的标准。基于 iOS 5 的 iPhone、iPod Touch 以及 iPad 设备支持的是 OpenGL ES 2.0。苹果的设备也支持旧的 OpenGL ES 1.1 版本。iOS 5 引入了 GLKit 软件框架，这个框架简化了很多常用的编程任务，同时部分隐藏了所支持的这两个 OpenGL ES 版本间的差异。本书主要关注带有 GLKit 的 iOS 5 所支持的 OpenGL ES 2.0 版本。

为了能在 ANSI C 编程语言中使用，OpenGL ES 定义了一个应用程序编程接口。通常用来开发苹果产品的 C++ 和 Objective-C 编程语言可以与 ANSI C 无缝交互。特定的转换层或者粘合层的存在使 OpenGL ES 可以用在 JavaScript 和 Python 中。新兴的 Web 编程标准，如非营利性 Web3D 联盟的 WebGL 标准，也正准备在网页上实现对于 OpenGL ES API 的跨平台标准化访问。本书中讲解的 3D 图形概念适用于所有具有 3D 功能的嵌入式系统。

本章会讲解使用 OpenGL ES 和 iOS 5 来实现 3D 图形的一般方法，但并不会深入特定的编程细节。最新的 3D 图形硬件加速是所有高级移动产品的可视化效果的基础。学习本章是能够从移动硬件中萃取出最好的 3D 图形和可视效果的第一步。

1.1 3D 渲染

图形处理单元（GPU）就是能够结合几何、颜色、灯光和其他数据而产生一个屏幕图像的硬件组件。屏幕只有 2 维，因此显示 3D 数据的技巧就在于产生能够迷惑眼睛使其看到丢失的第 3 维的一个图像，参见图 1-1 中的例子。

用 3D 数据生成一个 2D 图像的过程就叫渲染。在计算机上显示的图片是由矩形的颜色点组成的，这些矩形的颜色点叫做像素。图 1-2 放大了图像中的一部分来显示

单独的像素。如果通过放大镜仔细观察显示器，你会看到每个像素都是由 3 个颜色元素组成的，即一个红点、一个绿点和一个蓝点。图 1-2 还显示了一个进一步放大的像素来描述单独颜色的元素。在一个全彩色的显示器上，一个像素通常含有红、绿、蓝 3 个元素，但这些元素的排列样式可能与图 1-2 中显示的一个挨一个的排列方式不同。



图 1-1 用 3D 数据生成的一个示例图像

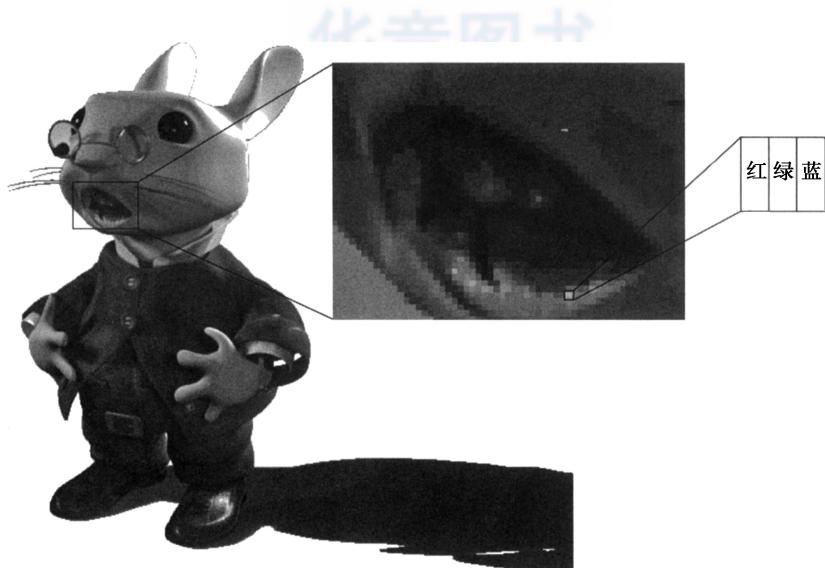


图 1-2 图像是由像素构成的，每个像素有红、绿、蓝 3 个元素

图像是以每个像素至少包含3个值的一个数组存储在电脑的存储器中的。第一个值指定了像素的红色元素的强度，第二个值代表绿色强度，第三个值是蓝色强度。一个包含10000个像素的图像能够以一个拥有30000个强度值的数组的形式存储在存储器中，像素的3个元素每个需要一个值。以不同的强度结合红绿蓝3个值就足以产生彩虹的所有颜色。如果3个元素的强度都是0，结果颜色就是黑色。如果3个颜色都是强度的最大值，结果颜色就是白色。黄色是通过丢掉蓝色并混合红色和绿色得到的。图1-3中的Mac OS X标准颜色面板用户界面包含能够调节相关的红、绿、蓝强度的图像滑块。

渲染3D数据为一个2D图像通常发生在几个不同的步骤中，包括设置图像中的每个像素的红、绿、蓝颜色强度的计算。总的来说，本书会讲解怎么让程序在渲染过程的每一步中尽可能地利用好OpenGL ES和图形硬件。第一步是为GPU提供要处理的3D数据。

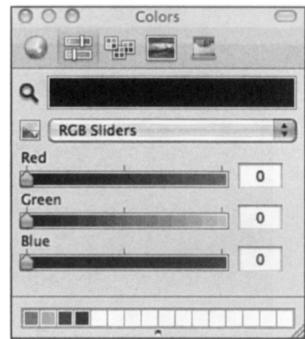


图1-3 用于调节红、绿、蓝颜色元素强度的用户界面

1.2 为图形处理器提供数据

程序会保存3D场景数据到硬件随机存取存储器(RAM)中。嵌入式系统的中央处理单元有专门为其分配的RAM。在图形处理的过程中，GPU也有专门为其分配的RAM。使用现代硬件渲染3D图形的速度几乎完全取决于不同的内存区域被访问的方式。

OpenGL ES是一种软件技术。OpenGL ES部分运行在CPU上，部分运行在GPU上。OpenGL ES横跨在两个处理器之间，协调两个内存区域之间的数据交换。图1-4中的箭头代表了与3D渲染相关的硬件组件之间的数据交换。每个箭头也代表着一个渲染性能的瓶颈。OpenGL ES通常会高效地协调数据交换，但是程序与OpenGL ES的交互方式会明显地增加或者减少所需的数据交换的数量和类型。对于渲染速度，最快的数据交换方式是没有数据交换。

首先，从一个内存区域复制数据到另一

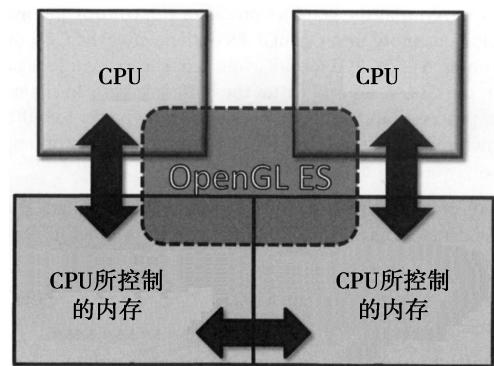


图1-4 硬件组件和OpenGL ES之间的关系

个内存区域速度是相对较慢的。更糟糕的是，除非非常小心，在内存复制发生的时候 GPU 和 CPU 都不能把内存另作它用。因此内存区域之间的数据交换需要尽量避免。

其次，所有的内存访问都是相对较慢的。最新的嵌入式 CPU 可以很容易地完成大约每秒一亿次的运算，但是它只能每秒读写内存 200 万次。这意味着，除非 CPU 能够在每次从内存读取一块数据后有效地运行五个或者更多个运算，否则处理器的性能就处于次优状态，这种状态叫做“数据饥饿”。这种情况对于 GPU 来说更明显，在理想条件下，GPU 能够每秒执行数亿次运算，但是却只能每秒访问内存 2 亿次。GPU 几乎总是受限于内存访问的性能，并且通常需要在每块数据上执行 10 ~ 30 次运算才不会影响整体的图形输出。

概括最新 OpenGL ES 和以前的 OpenGL 版本之间差异的一种方式是，最新的 OpenGL ES 为了支持新改进的方法抛弃了对于旧式的低效的内存复制操作的支持。如果你曾做过老式的桌面 OpenGL 开发，那么现在可以忘掉那些经验了。现在的嵌入式系统不再支持以前的糟糕技术了。OpenGL ES 仍然支持多种为图形处理器提供数据的方式，但只存在一种最好的方式，并且本书会始终使用这种方式。

1.2.1 缓存：提供数据的最好方式

OpenGL ES 为两个内存区域间的数据交换定义了缓存（buffers）的概念。缓存是指图形处理器能够控制和管理的连续 RAM。程序从 CPU 的内存复制数据到 OpenGL ES 的缓存。在 GPU 取得一个缓存的所有权以后，运行在 CPU 中的程序理想情况下将不再接触这个缓存。通过控制独占的缓存，GPU 就能够尽可能以最有效的方式读写内存。图形处理器把它处理大量数据的能力异步同时地应用到缓存上，这意味着在 GPU 使用缓存中的数据工作的同时，运行在 CPU 中的程序可以继续执行。

几乎所有程序提供给 GPU 的数据都应该放入缓存中。缓存存储的到底是几何数据、颜色、灯光效果，还是其他信息并不重要。为缓存提供数据有如下 7 个步骤。

- 1) 生成（Generate）——请求 OpenGL ES 为图形处理器控制的缓存生成一个独一无二的标识符。
- 2) 绑定（Bind）——告诉 OpenGL ES 为接下来的运算使用一个缓存。
- 3) 缓存数据（Buffer Data）——让 OpenGL ES 为当前绑定的缓存分配并初始化足够的连续内存（通常是从 CPU 控制的内存复制数据到分配的内存）。
- 4) 启用（Enable）或者禁止（Disable）——告诉 OpenGL ES 在接下来的渲染中是否使用缓存中的数据。
- 5) 设置指针（Set Pointers）——告诉 OpenGL ES 在缓存中的数据的类型和所有需要访问的数据的内存偏移值。
- 6) 绘图（Draw）——告诉 OpenGL ES 使用当前绑定并启用的缓存中的数据渲染整个场景或者某个场景的一部分。

7) 删除 (Delete) ——告诉 OpenGL ES 删除以前生成的缓存并释放相关的资源。

理想情况下，每个生成的缓存都可以使用一个相当长的时间（可能是程序的整个生命周期）。生成、初始化和删除缓存有时需要耗费时间来同步图形处理器和 CPU。存在这个延迟是因为 GPU 在删除一个缓存之前必须完成所有与该缓存相关的等待中的运算。如果一个程序每秒生成和删除缓存数千次，GPU 可能就没有时间来完成任何渲染了。

OpenGL ES 为一种类型的缓存在使用过程中的每一个步骤的执行定义了下面的 C 语言函数，同时为其他类型的缓存提供了类似的函数。

- glGenBuffers()——请求 OpenGL ES 为图形处理器控制的缓存生成一个独一无二的标识符。
- glBindBuffer()——告诉 OpenGL ES 为接下来的运算使用一个缓存。
- glBufferData() 或者 glBufferSubData()——让 OpenGL ES 为当前绑定的缓存分配并初始化足够的连续内存（通常是从 CPU 控制的内存复制数据到分配的内存）。
- glEnableVertexAttribArray() 或者 glDisableVertexAttribArray()——告诉 OpenGL ES 在接下来的渲染中是否使用缓存中的数据。
- glVertexAttribPointer()——告诉 OpenGL ES 在缓存中的数据的类型和所有需要访问的数据的内存偏移值。
- glDrawArrays() 或者 glDrawElements()——告诉 OpenGL ES 使用当前绑定并启用的缓存中的数据渲染整个场景或者某个场景的一部分。
- glDeleteBuffers()——告诉 OpenGL ES 删除以前生成的缓存并释放相关的资源。

注意 这里提到的 C 函数只是为了描述 OpenGL ES 2.0 API 函数与基础概念的对应方式。全书会用各种例子讲解这类 C 函数，因此现在不用担心记不住它们。

1.2.2 帧缓存

GPU 需要知道应该在内存中的哪个位置存储渲染出来的 2D 图像像素数据。就像为 GPU 提供数据的缓存一样，接收渲染结果的缓冲区叫做帧缓存 (frame buffer)。程序会像任何其他种类的缓存一样生成、绑定、删除帧缓存。但是帧缓存不需要初始化，因为渲染指令会在适当的时候替换缓存的内容。帧缓存会在被绑定的时候隐式开启，同时 OpenGL ES 会自动地根据特定平台的硬件配置和功能来设置数据的类型和偏移。

可以同时存在很多帧缓存，并且可以通过 OpenGL ES 让 GPU 把渲染结果存储到任意数量的帧缓存中。但是，屏幕显示像素要受到保存在前帧缓存 (front frame buffer) 的特定帧缓存中的像素颜色元素的控制。程序和操作系统很少会直接渲染到前帧缓存

中，因为那样会让用户看到正在渲染中的还没渲染完成的图像。相反，程序和操作系统会把渲染结果保存到包括后帧缓存（back frame buffer）在内的其他帧缓存中。当渲染后的后帧缓存包含一个完成的图像时，前帧缓存与后帧缓存几乎会瞬间切换。后帧缓存会变成新的前帧缓存，同时旧的前帧缓存会变成后帧缓存。图 1-5 展示了屏幕显示像素、前帧缓存及后帧缓存三者之间的关系。

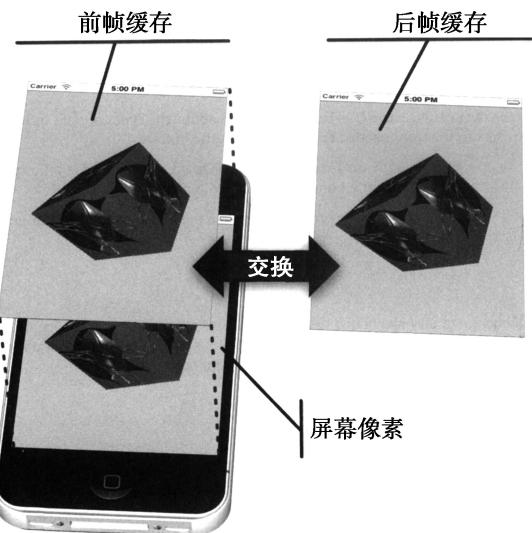


图 1-5 前帧缓存决定了屏幕上显示的像素颜色，同时会与后帧缓存切换

1.3 OpenGL ES 的上下文

用于配置 OpenGL ES 的保存在特定平台的软件数据结构中的信息会被封装到一个 OpenGL ES 上下文（context）中。OpenGL ES 是一个状态机器，这意味着在一个程序中设置了一个配置值后，这个值会一直保持，直到程序修改了这个值。上下文中的信息可能会被保存在 CPU 所控制的内存中，也可能被保存在 GPU 所控制的内存中。OpenGL ES 会按需在两个内存区域之间复制信息，知道何时发生复制有助于程序的优化。第 9 章会介绍性能优化的技术。

OpenGL ES 上下文的内部实现依赖于特定的嵌入式系统以及特定的 GPU 硬件。OpenGL ES 为跟上下文的交互提供了 ANSI C 语言函数，以使程序不需要知道太多与特定系统相关的信息。

OpenGL ES 上下文会跟踪用于渲染的帧缓存。上下文还会跟踪用于几何数据、颜色等的缓存。上下文会决定是否使用某些功能，比如纹理和灯光，分别在第 3 章和第 4 章中进行讲解。上下文还会为渲染定义当前的坐标系统，这个会在第 2 章中讲解。

1.4 一个 3D 场景的几何数据

在为 GPU 提供数据时，很多种数据都可以省略，比如灯光信息和颜色。在渲染一个场景时，OpenGL ES 必须具有的一种数据是用来指定要渲染的图形的几何数据。几何数据是相对于 3D 坐标系定义的。

1.4.1 坐标系

图 1-6 展示的是 OpenGL 的坐标系。坐标系是用于帮助显示空间中的位置之间的关系的参考线的集合。图 1-6 中的每一个箭头叫做一个轴。OpenGL ES 总是开始于一个矩形的笛卡儿坐标系，这意味着任何两个轴之间的角度都是 90 度。空间中的每一个位置被称为一个顶点，每个顶点通过其在 X、Y、Z 轴上的位置被定义。

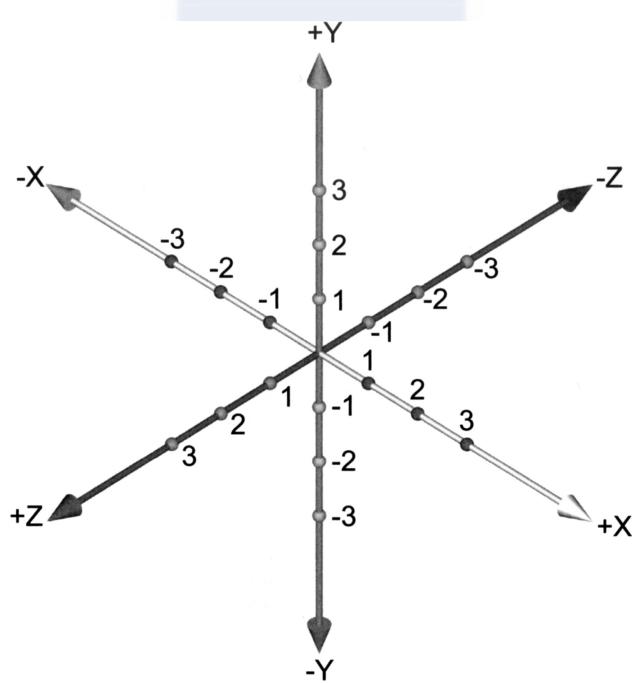


图 1-6 X、Y 和 Z 轴定义了 OpenGL 的坐标系

图 1-7 显示了相对于坐标轴的在 {1.5, 3.0, 0.0} 位置的顶点。这个顶点是通过 X 轴 1.5 的位置，Y 轴 3.0 的位置，Z 轴 0 的位置来定义的。图 1-7 中的虚线显示了顶点是怎么与坐标轴对齐的。

沿着每个轴的位置被称为坐标，确定一个用于 3D 图形的顶点需要 3 个坐标。图 1-8 展示了更多的顶点以及它们在 OpenGL 坐标系中的相对位置。虚线显示了顶点是怎么与坐标轴对齐的。

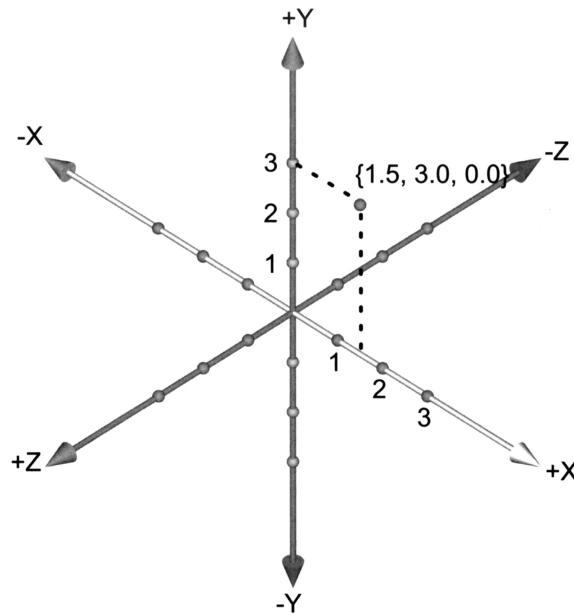
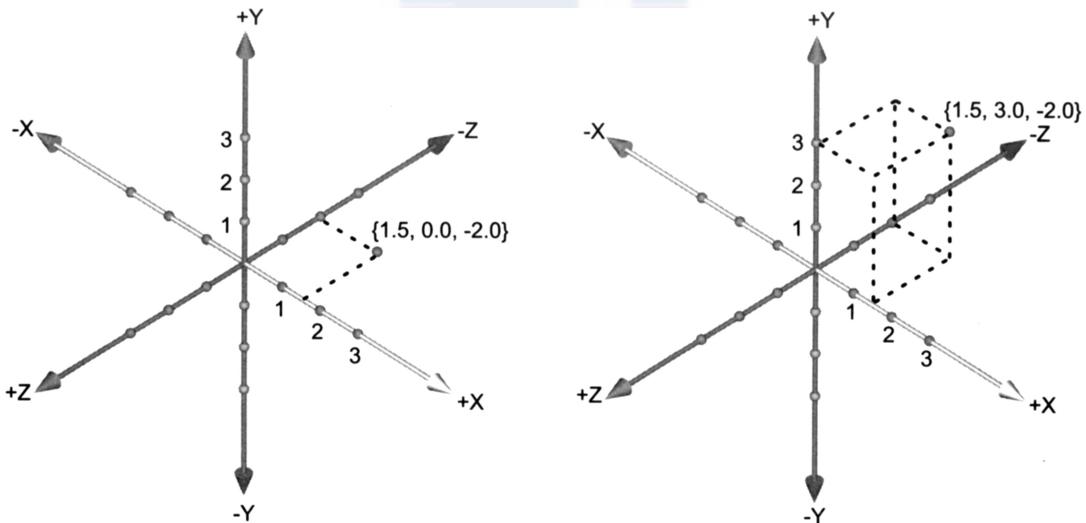
图 1-7 相对于坐标轴的在 $\{1.5, 3.0, 0.0\}$ 位置的顶点

图 1-8 一个坐标系中的顶点的相对位置

OpenGL ES 坐标是以浮点数来存储的。现代 GPU 对浮点运算做了专门的优化，即使是使用其他数据类型的顶点也会被转换成浮点值。

使用和理解坐标系的关键点之一是记住它仅仅是一个假想的数学工具。第 5 章会介绍修改坐标系后的巨大影响。从纯数学意义上讲，可以有很多种非笛卡儿坐标系。例如，极坐标系是通过假想的落在一个使用两个角和一个半径的球的表面上的点来确定 3D 空间中的位置的。现在不用担心非笛卡儿坐标系的数学运算。硬件中的嵌入式 GPU

对于大部分非笛卡儿坐标系都是不支持的，并且本书中没有例子会使用非笛卡儿坐标系。如果你需要在工程中使用，以任何坐标系表达的位置都可以在需要的时候转换成 OpenGL ES 支持的默认坐标系。

OpenGL ES 坐标系没有单位。点 {1, 0, 0} 和点 {2, 0, 0} 之间的距离是沿着 X 轴的 1 单位，但是问下自己，“1 什么？是 1 英寸、1 毫米、1 英里，还是 1 光年？”答案是“这无关紧要”，确切地说这取决于你。你可以自由地假设距离 1 代表厘米还是你的 3D 场景中的其他单位。

注意 对于 3D 图形来说不定义单位可能非常方便，但是这也出现了一个挑战，就是在你想要打印你的 3D 场景的时候。苹果公司的 iOS 系统支持用于 2 维绘图兼容 PDF 的 Quartz 2D，并且按照现实世界的尺寸定义了单位。现实世界的尺寸允许你在绘制几何对象的同时知道将要绘制到与打印分辨率无关的页面上的对象的大小。与此相反，与分辨率无关的定义或者渲染 OpenGL 几何图形的方式是不存在的。

1.4.2 矢量

矢量是图形编程中频繁使用的另一个数学概念。从某种意义上讲，矢量是另一种诠释顶点数据的方式。矢量是既有方向又有距离的一个量。距离也叫大小。所有的顶点都可以用它相对于 OpenGL ES 坐标系原点 ({0, 0, 0}) 的距离和方向来定义。图 1-9 使用一个从原点到顶点 {1.5, 3.0, -2.0} 的实心箭头来描述矢量。虚线显示了顶点是如何与坐标轴对齐的。

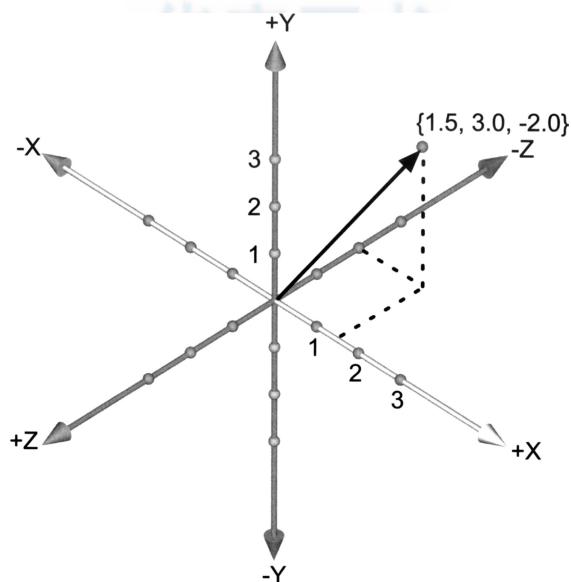


图 1-9 在 3D 坐标系中的一个矢量

可以使用每个顶点的坐标之间的差异来计算任意两个顶点之间的矢量。介于顶点 $\{1.5, 3.0, -2.0\}$ 与原点之间的矢量是 $\{1.5-0.0, 3.0-0.0, -2.0-0.0\}$ 。图 1-10 中的顶点 V1 与顶点 V2 之间的矢量等于 $\{V2.x - V1.x, V2.y - V1.y, V2.z - V1.z\}$ 。

矢量可以加在一起进而产生一个新矢量。介于原点与任意顶点之间的矢量是 3 个轴对齐矢量的和，参见图 1-11。矢量 $A+B+C$ 等于矢量 D（如下式所示），同时确定了在 $\{1.5, 3.0, -2.0\}$ 位置的顶点。

$$\begin{aligned}D.x &= A.x + B.x + C.x = 1.5 + 0.0 + 0.0 = 1.5 \\D.y &= A.y + B.y + C.y = 0.0 + 3.0 + 0.0 = 3.0 \\D.z &= A.z + B.z + C.z = 0.0 + 0.0 + -2.0 = -2.0\end{aligned}$$

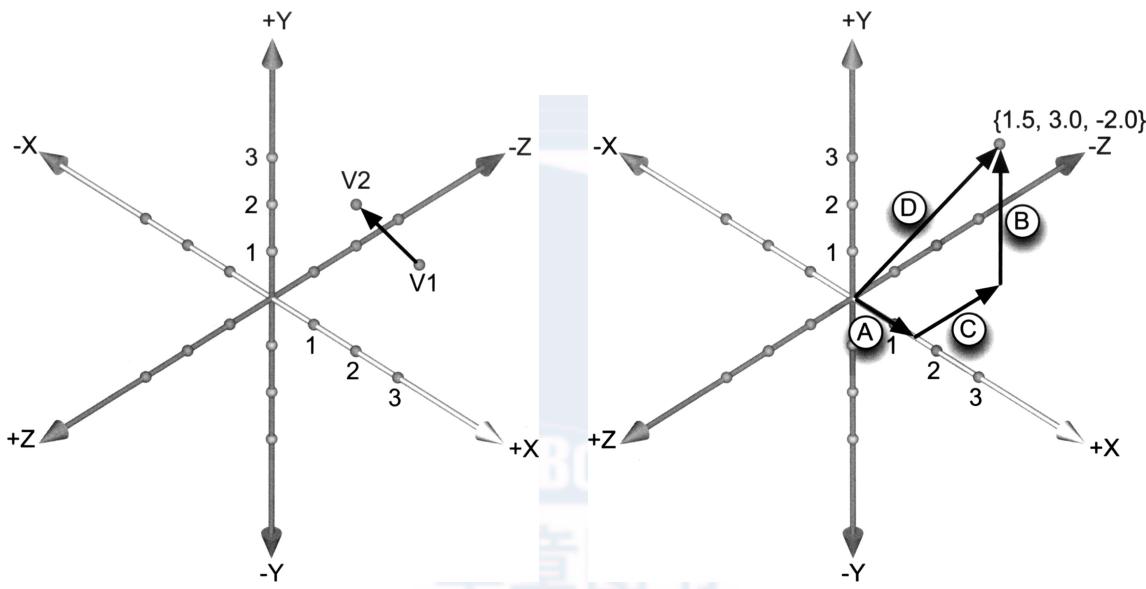


图 1-10 介于顶点 V1 与 V2 之间的矢量

图 1-11 轴对齐矢量的和

矢量是理解现代 GPU 的关键，因为图形处理器就是大规模并行矢量处理器。GPU 能够同时控制多个矢量，并执行用于定义渲染结果的矢量运算。在后面的章节会根据需要适时地解释多个关键的除了加法和减法之外的矢量运算。OpenGL ES 的默认坐标系、顶点和矢量为要渲染的几何数据的定义提供了足够的数学元素。

注意 线性代数的整个领域是处理矢量数学运算。线性代数是关于三角学的，但是它主要使用简单的运算，如加法和乘法来建立和操作复杂的几何图形。计算机图形依赖于线性代数，因为计算机尤其是 GPU 擅长简单的数学运算。本书会根据需要逐渐对线性代数的概念进行讲解。

1.4.3 点、线、三角形

OpenGL ES 使用顶点数据来定义点、线段和三角形。一个顶点会定义坐标系中的一个点的位置，两个顶点会定义一个线段，三个顶点会定义一个三角形。OpenGL ES 只渲染顶点、线段和三角形。图 1-12 显示了使用很多三角形建立的几何对象有多复杂。

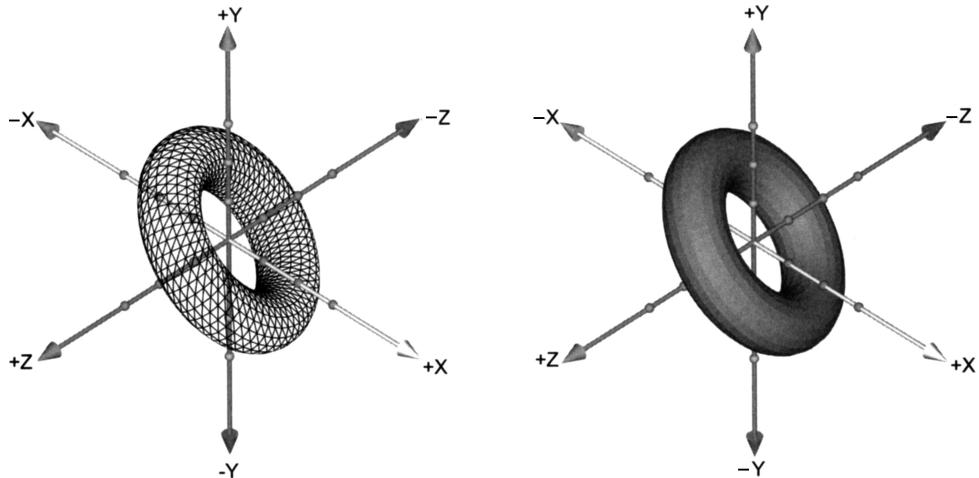


图 1-12 渲染为三角形和线段的顶点数据

1.5 小结

OpenGL ES 是访问类似 iPhone 和 iPad 的现代嵌入式系统的 3D 图形加速硬件的标准。把程序提供的几何数据转换为屏幕上的图像的过程叫做渲染。GPU 控制的缓存是高效渲染的关键。容纳几何数据的缓存定义了要渲染的点、线段和三角形。OpenGL ES 3D 的默认坐标系、顶点和矢量为几何数据的描述提供了数学基础。渲染的结果通常保存在帧缓存中。有两个特别的帧缓存，前帧缓存和后帧缓存，它们控制着屏幕像素的最终颜色。OpenGL ES 的上下文保存了 OpenGL ES 的状态信息，包括用于提供渲染数据的缓存地址和用于接收渲染结果的缓存地址。

第 2 章会介绍一个使用苹果 Xcode 开发工具和 Cocoa Touch 面向对象的框架在 iPhone、iPod Touch 和 iPad 中绘制 3D 图形的程序。第 2 章的例子是本书后面例子的基础。

第 2 章 让硬件为你工作

本章会讲解怎么在 iOS 5 应用中建立和使用 OpenGL ES 图形。一个初始例子程序会利用第 1 章中的图形概念让嵌入式硬件渲染一个图像。初始例子会被延伸成另外两个版本以解释苹果的 iOS 5 引入的 GLKit 技术与基础的 OpenGL ES 函数之间的关系。

2.1 使用 OpenGL ES 绘制一个 Core Animation 层

第 1 章介绍了 OpenGL ES 的帧缓存。iOS 操作系统不会让应用直接向前帧缓存或者后帧缓存绘图，也不会让应用直接控制前帧缓存和后帧缓存之间的切换。操作系统为自己保留了这些操作，以便它可以随时使用 Core Animation 合成器来控制显示的最终外观。

Core Animation 包含层的概念。同一时刻可以有任意数量的层。Core Animation 合成器会联合这些层并在后帧缓存中产生最终的像素颜色，然后切换缓存。图 2-1 显示的是合并两个层来产生后帧缓存中的颜色数据的过程。

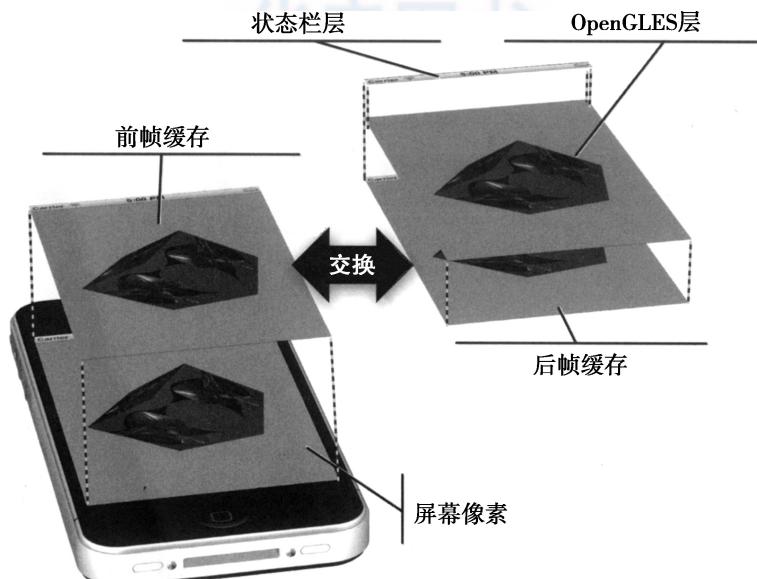


图 2-1 合并 Core Animation 层来产生后帧缓存中的颜色数据

一个应用提供的层与操作系统提供的层混合起来可以产生最终的显示外观。例如，在图 2-1 中，OpenGL ES 层显示了一个应用生成的旋转的立方体，但是在显示器顶部的显示状态栏的层是由操作系统生成和控制的。大部分的应用使用多个层。每一个 iOS 原生用户界面对象都有一个对应的 Core Animation 层，因此一个显示多个按钮、文本域、图像等的应用会自动使用多个层。

层会保存所有绘制操作的结果。例如，iOS 为有效地在层上绘制视频提供了软件对象。有用类似淡入淡出的特殊效果显示图像的层。层内容可以使用苹果的用于 2D 的支持富文本的 Core Graphics 框架来绘制。类似本章中的例子，应用会使用 OpenGL ES 来渲染层内容。

注意 苹果的 Core Animation 合成器使用 OpenGL ES 来尽可能高效地控制 GPU、混合层和切换帧缓存。图形程序员经常使用术语混合（composite）来描述混合图像以形成一个合成结果的过程。所有显示的图画都是通过 Core Animation 合成器来完成的，因此最终都涉及 OpenGL ES。

帧缓存会保存 OpenGL ES 的渲染结果，因此为了渲染到一个 Core Animation 层上，程序需要一个连接到某个层的帧缓存。简言之，每个程序用足够的内存配置一个层来保存像素颜色数据，之后创建一个使用层的内存来保存渲染的图像的帧缓存。图 2-2 介绍了 OpenGL ES 的帧缓存与层之间的关系。

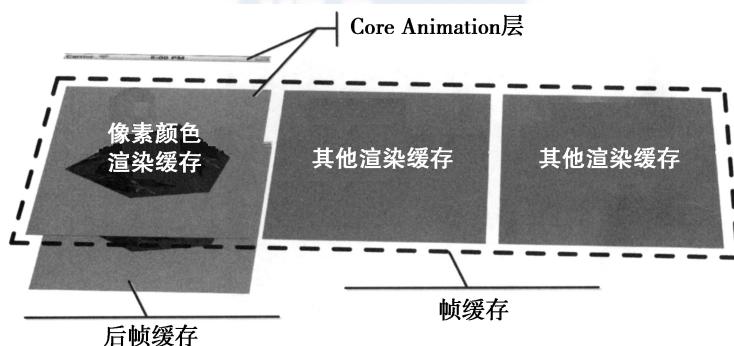


图 2-2 帧缓存可以与层分享像素存储器

图 2-2 显示了一个像素颜色渲染缓存（pixel color render buffer）和另外两个标识为其他渲染缓存（other render buffer）的缓存。除了像素颜色数据，OpenGL ES 和 GPU 有时会以渲染的副产品的形式产生一些有用的数据。帧缓存可以配置多个叫做渲染缓存的缓存来接收多种类型的输出。与层分享数据的帧缓存必须要有一个像素颜色渲染缓存。其他的渲染缓存是可选的，但是本章中不会使用。图 2-2 为完整起见显示了其他的渲染缓存，因为大部分不平凡的 OpenGL ES 程序会使用至少一个额外的渲染缓存，具体内容会在第 5 章中解释。

2.2 结合 Cocoa Touch 和 OpenGL ES

OpenGL ES Ch2_1 是本章的第一个示例应用，为本书的示例提供了起始点。这个程序设置 OpenGL ES 渲染一个图像到一个 Core Animation 层。之后由 iOS Core Animation 合成器自动地把渲染的层内容与其他层结合起来，产生保存在后帧缓存中的像素颜色数据，并最终显示到屏幕上。

图 2-3 显示了 OpenGL ES Ch2_1 渲染出来的图像。虽然只有一个三角形，但是执行 OpenGL ES 渲染的步骤与更复杂场景的步骤是相同的。这个例子使用了苹果的 Cocoa Touch 技术以及 Xcode 集成开发环境（IDE）。包含在 Xcode 中的苹果 iOS 开发者工具是 iOS 软件开发工具包（SDK）的一部分，其官方下载网址为 <http://developer.apple.com/technologies/ios/>。

2.2.1 Cocoa Touch

Cocoa Touch 由可重用的用于创建和运行应用的函数及软件对象组成。苹果的 iOS 由一个类似于 Mac OS X 的近乎完全与 UNIX 相似的操作系统组成，Mac OS X 是运行在苹果的 Macintosh 系列电脑上的一个操作系统。谷歌的基于 Linux 的 Android 操作系统也与 UNIX 类似。Cocoa Touch 建立在 UNIX 系统的基础之上，同时整合了许多不同的功能，从网络连接到 Core Animation，再到用户用来开始、停止、观察、与应用交互的图形用户界面。不使用 Cocoa Touch 而只使用 ANSI C 编程语言、UNIX 命令行工具，以及 UNIX 应用程序编程接口编写一个 iOS 程序在技术上是可能的。但是，大部分用户不会这么写程序，因为苹果不接受这样的应用发布到苹果的 App Store 上。

Cocoa Touch 主要是由 Objective-C 编程语言实现的。Objective-C 向 ANSI C 编程语言添加了少量的语法元素和一个面向对象的运行时系统。Cocoa Touch 提供了对于包括 OpenGL ES 的底层 ANSI C 技术的入口，但即使是最简单的像 OpenGL ES Ch2_1 这样的例子应用也需要 Objective-C。Cocoa Touch 提供了很多用于 iOS 应用的标准功能，以及让开发者专注于让应用变得独一无二的功能。每一个 iOS 程序员都受益于对 Cocoa Touch 和 Objective-C 的学习和使用。但是，本书主要关注 OpenGL ES，所以仅仅简要介绍 Cocoa Touch 技术。

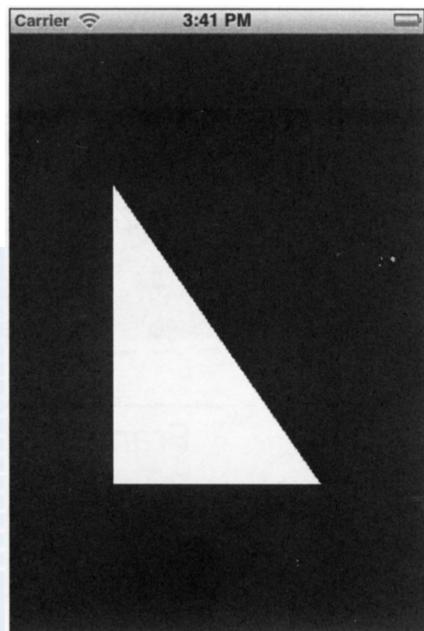


图 2-3 OpenGL ES Ch2_1 示例所产生的最终屏幕结果

2.2.2 使用苹果开发者工具

Xcode 运行在 Mac OS X 上，包含可识别语法的代码编辑器、编译器、调试器、性能工具，以及一个文件管理用户界面。Xcode 支持使用 ANSI C、C++、Objective-C 和 Objective-C++ 语言做开发，同时它可以与各种外部源代码管理系统一起工作。苹果使用 Xcode 开发自己的软件。更多信息可登录：http://developer.apple.com/iphone/library/referencelibrary/GettingStarted/URL_Tools_for_iPhone_OS_Development/index.html 查询。

图 2-4 显示的是 Xcode，并且加载了用 OpenGLE_S_Ch2_1.xcodeproj 配置文件定义的建立 OpenGLES_Ch2_1 例子所需要的资源。Xcode 有与大部分其他的 IDE 相似功能，例如开源 Eclipse IDE 和微软的 Visual Studio IDE。图 2-4 中左边栏的列表指定了需要编译并链接到最终应用中的文件。在 Xcode 窗口顶部的工具栏提供了用于组建和运行开发中的应用的按钮。剩下的用户界面主要是由源代码编辑器组成。

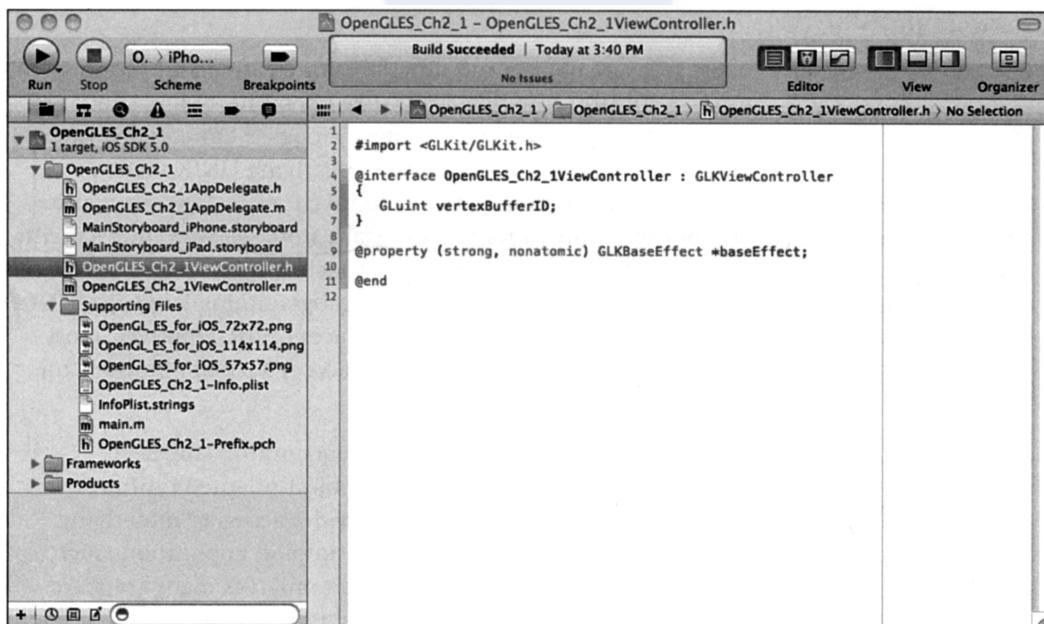


图 2-4 Xcode 工程示例

2.2.3 Cocoa Touch 应用架构

图 2-5 列出了所有使用 OpenGL ES 的最新 Cocoa Touch 应用的主要软件组件。箭头指示了组件之间的典型的信息流。在图 2-5 中 Cocoa Touch 提供了灰色的组件，应用通常使用不需修改的灰色组件。图 2-5 中的白色组件对于每一个应用来说都是独一无二的。较复杂的应用会包含专属的软件组件。不要被图 2-5 的复杂所迷惑。大部分情况下，只有应用委托（application delegate）和根视图控制器（root view controller）这两个白色组件需要所有程序员的干预，其他组件是提供标准的 iOS 应用行为的基础结构的。

一部分，均不需要程序员做任何干预。

操作系统控制对硬件组件的访问，发送用户的屏幕触摸事件给基于 Cocoa Touch 的应用。Cocoa Touch 实现了标准的图形组件，包括触摸键盘和状态栏，因此单独的应用不需要重新创建这些组件。苹果提供了一个图表以说明常用的 Cocoa Touch 应用组件，图表地址为：<http://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhone-OSProgrammingGuide/AppArchitecture/AppArchitecture.html>。为简单起见，苹果的图表省略了 OpenGL ES 和 Core Animation 层，但是却为 Cocoa Touch 应用设计提供了额外的依据。

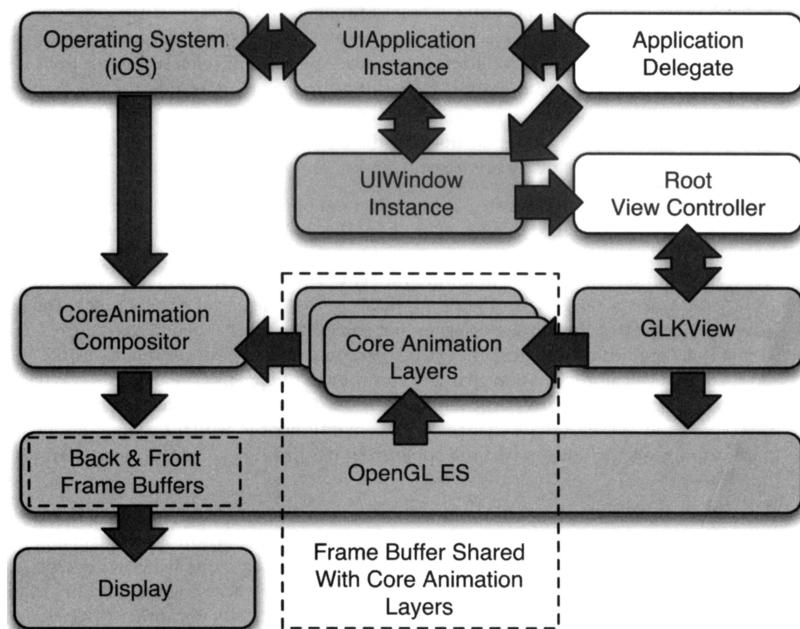


图 2-5 Cocoa Touch OpenGL ES 应用的软件架构

图 2-5 中显示了第 1 章介绍的 OpenGL ES 和帧缓存组件的作用。本章介绍了 Core Animation 层和 Core Animation 合成器。图 2-5 中剩下的组件实现了标准的 Cocoa Touch 行为。

- **UIApplication**：每个应用都包含一个单一的 UIApplication 类的实例。UIApplication 是一个 Objective-C Cocoa Touch 对象，它提供了应用与 iOS 之间的双向通信。应用向 iOS 请求服务，系统为正在运行的应用提供信息，例如显示器的当前方向。UIApplication 会与一个或更多个 Cocoa Touch UIWindow 实例通信，还会与一个用于路由用户输入事件到正确的对象的委托（delegate）通信。
- **应用委托 (application delegate)**：委托对象提供了一个响应另一个对象的变化或者影响另一个对象的行为的机会。基本思路是两个对象协调解决一个问题。对象

(如 UIApplication) 是非常常见并可以在各种情况下重用的。它保存了一个对于另一个对象（它的委托）的引用，并且在关键时刻发送消息给委托。可能只是通知委托发生了一些事情并给予委托一个执行额外处理的机会的消息，或者是要求委托返回能够控制发生什么的关键信息的消息。委托通常是一个特定的应用所特有的自定义对象。应用委托会接收关于 Cocoa Touch 应用所运行的环境的所有重要改变，包括应用完成启动和结束这样的改变。

- UIWindow：Cocoa Touch 应用总是有至少一个自动创建的覆盖整个屏幕的 UIWindow 实例。UIWindow 实例控制屏幕的矩形区域，并且它们能够被重叠和分层以便一个窗口覆盖另一个窗口。除了覆盖整个屏幕的窗口，Cocoa Touch 应用很少直接访问窗口。Cocoa Touch 会按需自动使用其他的 UIWindows 来向用户显示警告和状态信息。UIWindows 会包含一个或多个提供窗口图形内容的 UIView 实例。应用架构内的窗口的一个重要的作用是从 UIApplication 实例收集用户的输入事件，然后根据实际情况把这些事件重新发送给正确的 UIView 实例。例如，UIWindow 确定哪一个 UIView 实例被用户触摸，然后直接发送合适的事件到那个实例。
- 根视图控制器：每一个窗口都有一个可选的根视图控制器。视图控制器是 Cocoa Touch UIViewController 类的实例并把大部分 iOS 应用的设计联系起来。视图控制器会调节一个相关的视图的外观并支持在设备方向变化时旋转视图。根视图控制器指定填充整个窗口的 UIView 实例。UIViewController 类的默认行为控制了 iOS 应用的标准可视化效果。GLKViewController 类是支持 OpenGL ES 特有的行为和动画计时的 UIViewController 的内建子类。OpenGL_ES_Ch2_1 例子创建了一个 GLKViewController 的子类 OpenGL_ES_Ch2_1ViewController 来提供所有例子所特有的行为。
- GLKView：这是 Cocoa Touch UIView 类的内建子类。GLKView 简化了通过用 Core Animation 层来自动创建并管理帧缓存和渲染缓存共享内存所需要做的工作。GLKView 相关的 GLKViewController 实例是视图的委托并接收当视图需要重绘时的消息。创建你自己的 UIView 或者 GLKView 的子类来实现应用特有的绘图是可能的，但是 OpenGL_ES_Ch2_1 采用了简单的方法，使用未经修改的 GLKView。OpenGL_ES_Ch2_1 例子实现了所有应用特有的行为，包括在 OpenGL_ES_Ch2_1ViewController 类中的绘画。

注意 GLKView 和 GLKViewController 类名字中的 GLK 前缀表明这些类是 iOS 5 引入的 GLKit 框架的一部分。框架是指被编译后的代码所使用的编译后的代码、接口声明，以及类似图像和数据的资源文件的集合。框架高效地组织可重用共享库并且在一些情况下可能包含库的多个版本。GLKit 为简化 iOS 中 OpenGL ES 的使用提供了类和函数。GLKit 是 Cocoa Touch 以及多个其他的框架（包含 UIKit）的一部分，UIKit 包含许多类如 UIApplication、UIWindow 和 UIView。

2.3 OpenGLES_Ch2_1 示例

你可以从网址 <http://opengles.cosmichump.com/learning-opengl-es-sample-code/> 上下载关于本书的示例代码。Xcode 工程以及所有创建本书的示例所需要的文件都包含在里面。一个叫做 OpenGLES_Ch2_1.xcodeproj 的文件保存了关于工程自身的信息。在电脑上正确地安装了苹果的 iOS 5 软件开发工具包和 Xcode 之后，双击 OpenGLES_Ch2_1.xcodeproj 文件来开启 Xcode 并加载工程。在加载完成后，点击 Xcode 工具栏的 Run 按钮来编译并连接工程中的文件，然后会启动苹果的 iPhone 模拟器来运行 OpenGLES_Ch2_1 应用。

注意 本书的所有例子都是用苹果的 ARC 技术来为 Objective-C 对象管理内存的。在新创建的 iOS Xcode 工程中会默认开启 ARC。使用 ARC 避免了手动为对象管理内存并简化了例子代码。

图 2-6 列出了例子中构建和连接的文件。本节的其余部分介绍了每个文件的内容和目的。

OpenGLES_Ch2_1 工程是使用 Xcode 的标准单视图应用（Single View Application）模板创建的。这个模板设置新工程以创建一个由一个单独的填满整个屏幕的 UIView（或者其子类）实例组成的简单应用。其他模板为其他类型的 iOS 应用提供了一个起始点。单视图应用模板生成了一个命名为 OpenGLES_Ch2_1AppDelegate 的自定义应用委托类，一个命名为 OpenGLES_Ch2_1ViewController 的自定义视图控制器类。

2.3.1 OpenGLES_Ch2_1AppDelegate 类

OpenGLES_Ch2_1AppDelegate.h 文件是在工程刚创建时由 Xcode 自动生成的。OpenGLES_Ch2_1AppDelegate.h 文件包含对 OpenGLES_Ch2_1AppDelegate 类的 Objective-C 声明。OpenGLES_Ch2_1 例子可以不用做任何修改使用生成的类。

OpenGLES_Ch2_1AppDelegate.m 文件也是在工程刚创建时由 Xcode 自动生成的。OpenGLES_Ch2_1AppDelegate.m 文件包含对 OpenGLES_Ch2_1AppDelegate 类的 Objective-C 实现代码。生成的代码包含多个一般由应用的委托实现的方法的存根实现。OpenGLES_Ch2_1 例子可以不用做任何修改使用生成的类。

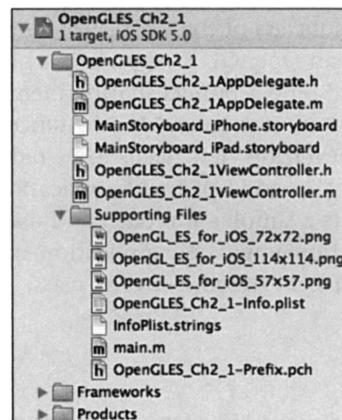


图 2-6 在 OpenGLES_Ch2_1 Xcode 工程中的文件

2.3.2 Storyboards

MainStoryboard_iPhone.storyboard 和 MainStoryboard_iPad.storyboard 文件也是由 Xcode 的单视图应用模板生成的。在 Xcode 中可以可视化地编辑这些文件以设定用户界面。iPad 与 iPhone 的屏幕大小不同，因此会有不同的用户界面。当例子运行时，Open-GLES_Ch2_1 实例会自动读取与当前设备相适合的 Storyboard。Storyboards 会把 UIViewController 实例以及与其相关联的 UIView 实例联系起来。Storyboards 会指定视图控制器之间的过渡，自动化应用的大部分用户交互设计，潜在地消除了原本需要编写的代码。但是，OpenGLES_Ch2_1 这个例子是如此简单以至于它只使用了一个视图控制器——OpenGLES_Ch2_1ViewController 类的一个实例。

2.3.3 OpenGLES_Ch2_1ViewController 类的 interface

OpenGLES_Ch2_1ViewController.h 文件原本是在工程刚创建的时候由 Xcode 生成的，但是在这个例子中做了一些修改。OpenGLES_Ch2_1ViewController.h 包含修改过的用于 OpenGLES_Ch2_1ViewController 类的 Objective-C 声明。粗体的代码是对所生成代码的主要修改。

```
//  
//  OpenGLES_Ch2_1ViewController.h  
//  OpenGLES_Ch2_1  
  
#import <GLKit/GLKit.h>  
  
@interface OpenGLES_Ch2_1ViewController : GLKViewController  
{  
    GLuint vertexBufferID;  
}  
  
@property (strong, nonatomic) GLKBaseEffect *baseEffect;  
  
@end
```

文件以一段注释开始。Cocoa Touch GLKit 框架接口是通过 #import 编译指令导入的，这与 ANSI C 的 #include 指令类似。两个指令都把特定文件的内容插入到包含这些指令的编译文件中。当相同的文件内容被导入到每个编译文件的次数超过一次时，Objective-C 的 #import 指令就会自动将其阻止，这里首选 #import，虽然 Objective-C 也支持 #include。

OpenGLES_Ch2_1ViewController 是 GLKViewController 类的一个子类并从 GLKViewController 继承了很多基本功能，GLKViewController 又从它的超类 UIViewController 继承了很多

功能。特别地，GLKViewController 会自动地重新设置 OpenGL ES 和应用的 GLKView 实例以响应设备方向的变化并可视化过渡效果，例如淡出和淡入。

在 OpenGL_ES_Ch2_1ViewController 的接口中声明的 vertexBufferID 变量保存了用于盛放本例中用到的顶点数据的缓存的 OpenGL ES 标识符。OpenGL_ES_Ch2_1ViewController 类的实现代码解释了缓存标识符的初始化和使用。

OpenGL_ES_Ch2_1ViewController 接口中的 baseEffect 属性声明了一个 GLKBaseEffect 实例的指针。Objective-C 属性声明了与实例变量相似的值。Objective-C 对象的属性可以使用“点符号”访问，例如，someObject.baseEffect；或者使用方法访问，比如按照 -set<PropertyName> 命名约定的用于设定属性值的方法和按照 -<propertyName> 命名约定的用于返回属性值的方法。这些特别命名的方法叫做访问器（accessor）。-baseEffect 访问器用于返回 OpenGL_ES_Ch2_1ViewController 的 baseEffect 属性的值。用于设置这个属性值的访问器是“-setBaseEffect:”。属性通常并不一定是以实例变量的形式实现的，它们的值可能是按需计算的或者是从数据库里加载的。Objective-C 的属性语法提供了对象不需要暴露类声明中的值是怎么保存的就能提供值的声明方式。当点符号被用来获取或设置一个属性的值时，Objective-C 编译器会自动地替换成对于适当命名的访问器方法的访问。Objective-C 还提供了一个自动生成访问器方法的实现代码的方法，这个我们会在 OpenGL_ES_Ch2_1ViewController 的实现代码中讲到。

GLKBaseEffect 是 GLKit 提供的另一个内建类。GLKBaseEffect 的存在是为了简化 OpenGL ES 的很多常用操作。GLKBaseEffect 隐藏了 iOS 设备支持的多个 OpenGL ES 版本之间的差异。在应用中使用 GLKBaseEffect 能减少需要编写的代码量。在 OpenGL_ES_Ch2_1ViewController 的实现中详细解释了 GLKBaseEffect。

2.3.4 OpenGL_ES_Ch2_1ViewController 类的实现

OpenGL_ES_Ch2_1ViewController.m 文件是原先那个工程刚建立时 Xcode 自动生成的文件，不过为这个例子做了一些修改。OpenGL_ES_Ch2_1ViewController.m 文件包含了针对 OpenGL_ES_Ch2_1ViewController 类的 Objective-C 的实现。在实现代码中只定义了 3 个方法：-viewDidLoad、-glkView:drawInRect: 和 -viewDidUnload。本节会详细解释这 3 个方法。本章和后面章节中的例子会在这个例子的源码之上建立。以下就是实现代码的开始部分：

```
//  
// OpenGL_ES_Ch2_1ViewController.m  
// OpenGL_ES_Ch2_1  
  
#import "OpenGL_ES_Ch2_1ViewController.h"
```

```
@implementation OpenGLES_Ch2_1ViewController
@synthesize baseEffect;
```

`@synthesize baseEffect`；表达式让 Objective-C 编译器为 `baseEffect` 属性自动地生成访问器方法。相对于使用 `@synthesize` 表达式，另一种方法是在代码中显式实现恰当命名的访问器方法。对于这个例子来说没有显式实现其访问器的任何理由，因为标准的访问器行为就足够了。只有在属性的存储需要特殊处理时或者属性值的变化需要调用自定义应用逻辑时才需要显式地编写访问器。

在 `OpenGLES_Ch2_1ViewController.m` 中的接下来一段代码定义了一个 C 结构体 `SceneVertex`，用来保存一个 `GLKVector3` 类型的成员 `positionCoords`。回顾一下第 1 章，顶点位置可以用一个起始于坐标系原点的矢量来表示。`GLKit` 的 `GLKVector3` 类型保存了 3 个坐标：X、Y 和 Z。

`vertices` 变量是一个用顶点数据初始化的普通 C 数组，这个变量用来定义一个三角形。

The `vertices` variable is as an ordinary C array initialized with vertex data to define a triangle.

```
///////////////////////////////
// This data type is used to store information for each vertex
typedef struct {
    GLKVector3 positionCoords;
}
SceneVertex;

///////////////////////////////
// Define vertex data for a triangle to use in example
static const SceneVertex vertices[] =
{
    {{-0.5f, -0.5f, 0.0}}, // lower left corner
    {{ 0.5f, -0.5f, 0.0}}, // lower right corner
    {{-0.5f,  0.5f, 0.0}} // upper left corner
};
```

这个例子的顶点位置坐标是挑选出来的，因为默认的用于一个 OpenGL 上下文的可见坐标系是分别沿着 X、Y、Z 轴从 -1.0 延伸到 1.0 的。例子三角形的坐标把它置于可见坐标系的中央并且与 X 和 Y 轴所形成的平面对齐。图 2-7 显示了 `vertices[]` 定义的在一个代表默认 OpenGL ES 坐标系的可见部分的立方体内的三角形。

1. -viewDidLoad

接下来的 `-viewDidLoad` 方法为 OpenGL ES 提供三角形的顶点数据。`-viewDidLoad` 方法继承自 `GLKViewController` 类，当与 `GLKViewController` 相关联的应用的 `GLKView` 实例被从 `storyboard` 文件加载时会自动调用这个方法。`OpenGLES_Ch2_1ViewController`

提供了它自己的 -viewDidLoad 方法的实现，在这个实现中会首先调用它的超类的实现：

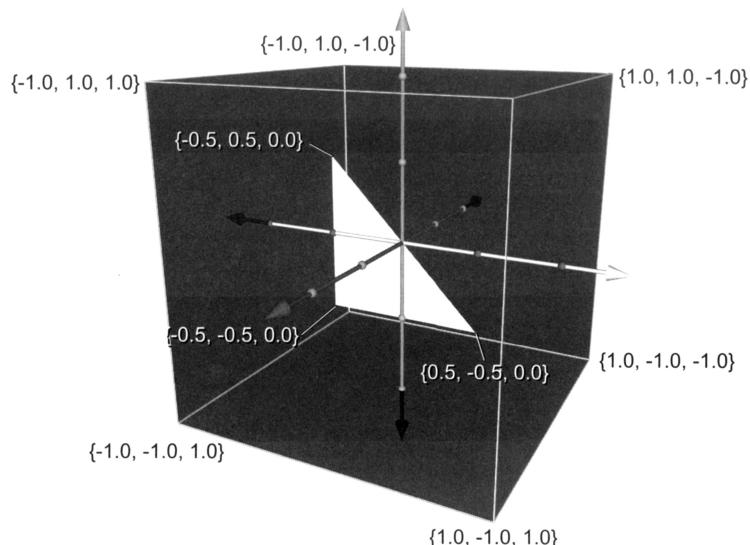


图 2-7 在默认的 OpenGL ES 坐标系中的三角形顶点

```
///////////////////////////////
// Called when the view controller's view is loaded
// Perform initialization before the view is asked to draw
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Verify the type of view created automatically by the
    // Interface Builder storyboard
    GLKView *view = (GLKView *)self.view;
    NSAssert([view isKindOfClass:[GLKView class]],
        @"View controller's view is not a GLKView");

    // Create an OpenGL ES 2.0 context and provide it to the
    // view
    view.context = [[EAGLContext alloc]
        initWithAPI:kEAGLRenderingAPIOpenGLES2];

    // Make the new context current
    [EAGLContext setCurrentContext:view.context];

    // Create a base effect that provides standard OpenGL ES 2.0
    // Shading Language programs and set constants to be used for
    // all subsequent rendering
    self.baseEffect = [[GLKBaseEffect alloc] init];
```

```

self.baseEffect.useConstantColor = GL_TRUE;
self.baseEffect.constantColor = GLKVector4Make(
    1.0f, // Red
    1.0f, // Green
    1.0f, // Blue
    1.0f); // Alpha

// Set the background color stored in the current context
glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // background color

// Generate, bind, and initialize contents of a buffer to be
// stored in GPU memory
glGenBuffers(1, // STEP 1
    &vertexBufferID);
glBindBuffer(GL_ARRAY_BUFFER, // STEP 2
    vertexBufferID);
glBufferData( // STEP 3
    GL_ARRAY_BUFFER, // Initialize buffer contents
    sizeof(vertices), // Number of bytes to copy
    vertices, // Address of bytes to copy
    GL_STATIC_DRAW); // Hint: cache in GPU memory
}

```

-viewDidLoad 方法会将它继承的 view 属性的值转换为 GLKView 类型。类似 OpenGL_ES_Ch2_1ViewController 的 GLKViewController 的子类只能与 GLKView 实例或者是 GLKView 子类的实例一起正确工作。但是，这个例子的 storyboard 文件定义了哪一个是与应用的 GLKViewController 实例相关联的视图。使用 NSAssert() 函数的一个运行时检查会验证在运行时从 storyboard 加载的视图是否确实是正确的类型。如果验证的条件为 false，那么 NSAssert() 会向调试器或 iOS 设备控制台发送一个错误消息。NSAssert() 还会生成一个如果不做处理就停止应用的 NSInternalInconsistencyException。在这个例子中，无法从一个加载自 storyboard 的错误视图还原应用的界面，因此在运行时监测到错误的时候最好先停止应用。

如在第 1 章介绍的，OpenGL ES 的上下文不仅会保存 OpenGL ES 的状态，还会控制 GPU 去执行渲染运算。OpenGL_ES_Ch2_1ViewController 的 -viewDidLoad 方法会分配并初始化一个内建的 EAGLContext 类的实例，这个实例会封装一个特定于某个平台的 OpenGL ES 上下文。苹果还没有说明开头的 EAGL 前缀代表什么，但是它可能代表的是“Embedded Apple GL”。苹果 iOS 中的 OpenGL ES 框架一般是以 EAGL 为前缀来声明 Objective-C 类和函数的。

在任何其他的 OpenGL ES 配置或者渲染发生之前，应用的 GLKView 实例的上下文属性都需要设置为当前。EAGLContext 实例既支持 OpenGL ES 1.1，又支持 OpenGL ES 2.0。本书中的例子使用的是 2.0 版本。下面的代码行在为视图的上下文属性赋值之

前，分配了一个新的 EAGLContext 的实例，并用粗体标注的常量将它初始化为 OpenGL ES 2.0。

```
view.context = [[EAGLContext alloc]
    initWithAPI:kEAGLRenderingAPIOpenGLES2];

// Make the new context current
[EAGLContext setCurrentContext:view.context];
```

一个应用可以使用多个上下文。EAGLContext 的方法 “+setCurrentContext:” 会为接下来的 OpenGL ES 运算设置将会用到的上下文。“+setCurrentContext:” 方法前面的加号表明 “+setCurrentContext:” 是一个类方法。在 Objective-C 中，类方法是在即使没有这个类的实例时仍然可以被类自身调用的方法。

苹果的 OpenGL ES framework 为 EAGLContext 的 “-initWithAPI:” 方法声明了常量 kEAGLRenderingAPIOpenGLES2。同时，还存在着一个 kEAGLRendering-APIOpenGLES1 常量。OpenGL ES 2.0 标准与以前的版本有很大的不同。尤其是 OpenGL ES 2.0 省略了很多特性和在上一个标准中定义的应用支持基础结构。作为替代，OpenGL ES 2.0 提供了一个新的更灵活的可编程 GPU 概念。为了灵活性，苹果建议我们在新应用中使用 OpenGL ES 2.0。在介绍苹果的 GLKit 之前，OpenGL ES 2.0 还需要一些前期工作以编程 GPU 并重新创建一些 2.0 版本丢失的、1.1 版本默认包含的便利功能。GLKit 现在替换了大部分 OpenGL ES 1.1 的基础结构，同时让 OpenGL ES 2.0 与 OpenGL ES 1.1 使用起来一样容易。

-viewDidLoad 方法接着设置 OpenGL_ES_Ch2_1ViewController 的 baseEffect 属性为一个新分配并初始化的 GLKBaseEffect 类型的实例，同时设置 GLKBaseEffect 实例的一些属性为比较适合这个例子的值。

```
// Create a base effect that provides standard OpenGL ES 2.0
// Shading Language programs and set constants to be used for
// all subsequent rendering
self.baseEffect = [[GLKBaseEffect alloc] init];
self.baseEffect.useConstantColor = GL_TRUE;
```

GLKBaseEffect 类提供了不依赖于所使用的 OpenGL ES 版本的控制 OpenGL ES 渲染的方法。OpenGL ES 1.1 跟 OpenGL ES 2.0 的内部工作机制是非常不同的。2.0 版本执行为 GPU 专门定制的程序。如果没有 GLKit 和 GLKBaseEffect 类，完成这个简单的例子就需要用 OpenGL ES 2.0 的 “Shading Language” 编写一个小的 GPU 程序。GLKBaseEffect 会在需要的时候自动地构建 GPU 程序并极大地简化本书中的例子。

控制渲染像素颜色的方式有很多种。这个应用的 GLKBaseEffect 实例使用一个恒定不变的白色来渲染三角形。这就意味着在三角形中的每一个像素都有相同的颜色。下面

的代码使用在 GLKit 中定义的用于保存 4 个颜色元素值的 C 数据结构体 GLKVector4 来设置这个恒定的颜色：

```
self.baseEffect.constantColor = GLKVector4Make(
    1.0f, // Red
    1.0f, // Green
    1.0f, // Blue
    1.0f); // Alpha

// Set the background color stored in the current context
glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // background color
```

前三个颜色元素是第 1 章图 1-2 中介绍的红、绿、蓝。第 4 个值为透明度，它决定了像素是半透明还是不透明。透明度元素会在第 3 章做详细介绍。设置红、绿、蓝为满值 1.0，以设置成白色。设置透明度为满值以使颜色为完全不透明。红、绿、蓝和透明度值统称为一个 RGBA 颜色。GLKVector4Make() 函数返回一个用指定的值初始化的 GLKit GLKVector4 结构体。

glClearColor() 函数设置当前 OpenGL ES 的上下文的“清除颜色”为不透明黑色。清除颜色由 RGBA 颜色元素值组成，用于在上下文的帧缓存被清除时初始化每个像素的颜色值。

第 1 章介绍了用于在 CPU 控制的内存和 GPU 控制的内存之间交换数据的缓存的概念。用于定义要绘制的三角形的顶点位置数据必须要发送到 GPU 来渲染。创建并使用一个用于保存顶点数据的顶点属性数组缓存。前 3 个步骤如下：

- 1) 为缓存生成一个独一无二的标识符。
- 2) 为接下来的运算绑定缓存。
- 3) 复制数据到缓存中。

下面来自 -viewDidLoad 方法的实现的代码执行了前 3 步：

```
// Generate, bind, and initialize contents of a buffer to be
// stored in GPU memory
 glGenBuffers(1, // STEP 1
             &vertexBufferID);
 glBindBuffer(GL_ARRAY_BUFFER, // STEP 2
             vertexBufferID);
 glBufferData(GL_ARRAY_BUFFER, // Initialize buffer contents
             sizeof(vertices), // Number of bytes to copy
             vertices, // Address of bytes to copy
             GL_STATIC_DRAW); // Hint: cache in GPU memory
```

在第 1 步中，glGenBuffers() 函数的第一个参数用于指定要生成的缓存标识符的数量，第二个参数是一个指针，指向生成的标识符的内存保存位置。在当前情况下，一个

标识符被生成，并保存在 vertexBufferID 实例变量中。

在第 2 步中，glBindBuffer() 函数绑定用于指定标识符的缓存到当前缓存。OpenGL ES 保存不同类型的缓存标识符到当前 OpenGL ES 上下文的不同部位。但是，在任意时刻每种类型只能绑定一个缓存。如果在这个例子中使用了两个顶点属性数组缓存，那么在同一时刻它们不能都被绑定。

glBindBuffer() 的第一个参数是一个常量，用于指定要绑定哪一种类型的缓存。OpenGL ES 2.0 对于 glBindBuffer() 的实现只支持两种类型的缓存，GL_ARRAY_BUFFER 和 GL_ELEMENT_ARRAY_BUFFER。GL_ELEMENT_ARRAY_BUFFER 将会在第 6 章详细解释。GL_ARRAY_BUFFER 类型用于指定一个顶点属性数组，例如本例中三角形顶点的位置。glBindBuffer() 的第二个参数是要绑定的缓存的标识符。

注意 缓存标识符实际上是一个无符号整型。0 值表示没有缓存。用 0 作为第二个参数调用 glBindBuffer() 函数来配置当前上下文的话，没有指定类型的缓存会被绑定。缓存标识符在 OpenGL ES 文档中又叫做“names”。

在第 3 步中，glBufferData 函数复制应用的顶点数据到当前上下文所绑定的顶点缓存中。

```
glBufferData(           // STEP 3
    GL_ARRAY_BUFFER, // Initialize buffer contents
    sizeof(vertices), // Number of bytes to copy
    vertices,        // Address of bytes to copy
    GL_STATIC_DRAW); // Hint: cache in GPU memory
```

glBufferData() 的第一个参数用于指定要更新当前上下文中所绑定的是哪一个缓存。第二个参数指定要复制进这个缓存的字节的数量。第三个参数是要复制的字节的地址。最后，第 4 个参数提示了缓存在未来的运算中可能将被怎样使用。GL_STATIC_DRAW 提示会告诉上下文，缓存中的内容适合复制到 GPU 控制的内存，因为很少对其进行修改。这个信息可以帮助 OpenGL ES 优化内存使用。使用 GL_DYNAMIC_DRAW 作为提示会告诉上下文，缓存内的数据会频繁改变，同时提示 OpenGL ES 以不同的方式来处理缓存的存储。

2. -glkView:drawInRect:

每当一个 GLKView 实例需要被重绘时，它都会让保存在视图的上下文属性中的 OpenGL ES 的上下文成为当前上下文。如果需要的话，GLKView 实例会绑定与一个 Core Animation 层分享的帧缓存，执行其他的标准 OpenGL ES 配置，并发送一个消息来调用 OpenGL_ES_Ch2_1ViewController 的 -glkView:drawInRect: 方法。-glkView:drawInRect: 是 GLKView 类的委托方法。作为 GLKViewController 的子类，OpenGL_ES_Ch2_1ViewController 会自动成为从 storyboard 文件加载的关联视图的委托。

下面委托方法的实现告诉 baseEffect 准备好当前 OpenGL ES 的上下文，以便为使用 baseEffect 生成的属性和 Shading Language 程序的绘图做好准备。接着，调用 OpenGL ES 的 glClear() 函数来设置当前绑定的帧缓存的像素颜色渲染缓存中的每一个像素的颜色为前面使用 glClearColor() 函数设定的值。正如 2.1 节所描述的，帧缓存可能有除了像素颜色渲染缓存之外的其他附加的缓存，并且如果其他的缓存被使用了，它们可以通过在 glClear() 函数中指定不同的参数来清除。glClear() 函数会有效地设置帧缓存中的每一个像素的颜色为背景色。

```
///////////////////////////////
// GLKView delegate method: Called by the view controller's view
// whenever Cocoa Touch asks the view controller's view to
// draw itself. (In this case, render into a Frame Buffer that
// shares memory with a Core Animation Layer)
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect
{
    [self.baseEffect prepareToDraw];

    // Clear Frame Buffer (erase previous drawing)
    glClear(GL_COLOR_BUFFER_BIT);

    // Enable use of currently bound vertex buffer
    glEnableVertexAttribArray(          // STEP 4
        GLKVertexAttribPosition);

    glVertexAttribPointer(           // STEP 5
        GLKVertexAttribPosition,
        3,                      // three components per vertex
        GL_FLOAT,               // data is floating point
        GL_FALSE,               // no fixed point scaling
        sizeof(SceneVertex), // no gaps in data
        NULL);                 // NULL tells GPU to start at
                               // beginning of bound buffer

    // Draw triangles using the first three vertices in the
    // currently bound vertex buffer
    glDrawArrays(GL_TRIANGLES,      // STEP 6
        0, // Start with first vertex in currently bound buffer
        3); // Use three vertices from currently bound buffer
}
```

在帧缓存被清除以后，是时候用存储在当前绑定的 OpenGL ES 的 GL_ARRAY_BUFFER 类型的缓存中的顶点数据绘制例子中的三角形了。使用缓存的前三步已经在 -viewDidLoad 方法中被执行了。正如第 1 章所描述的，OpenGL_ES_Ch2_1ViewController 的 “glkView:drawInRect:” 方法会执行剩下的几个步骤：

- 4) 启动。
- 5) 设置指针。
- 6) 绘图。

在第 4 步中，通过调用 `glEnableVertexAttribArray()` 来启动顶点缓存渲染操作。OpenGL ES 所支持的每一个渲染操作都可以单独地使用保存在当前 OpenGL ES 上下文中的设置来开启或关闭。

在第 5 步中，`glVertexAttribPointer()` 函数会告诉 OpenGL ES 顶点数据在哪里，以及怎么解释为每个顶点保存的数据。在这个例子中，`glVertexAttribPointer()` 的第一个参数指示当前绑定的缓存包含每个顶点的位置信息。第二个参数指示每个位置有 3 个部分。第三个参数告诉 OpenGL ES 每个部分都保存为一个浮点类型的值。第四个参数告诉 OpenGL ES 小数点固定数据是否可以被改变。本书中没有例子会使用小数点固定的数据，因此这个参数值是 `GL_FALSE`。

注意 小数点固定类型是 OpenGL ES 支持的对于浮点类型的一种替代。小数点固定类型用牺牲精度的方法来节省内存。所有现代 GPU 都对浮点数的使用做了优化，并且小数点固定数据在使用之前最终都会被转换成浮点数。因此坚持使用浮点数可以减少 GPU 的运算量并提高精度。

第五个参数叫做“步幅”，它指定了每个顶点的保存需要多少个字节。换句话说，步幅指定了 GPU 从一个顶点的内存开始位置转到下一个顶点的内存开始位置需要跳过多少字节。`sizeof(GLKVector3)` 指示在缓存中没有额外的字节，即顶点位置数据是密封的。在一个顶点缓存中保存除了每个顶点位置的 X、Y、Z 坐标之外的其他数据也是可能的。图 2-8 中的顶点数据内存模型显示了顶点存储器的一些选项。第一个图显示的是每个顶点的 3D 顶点位置坐标都紧密地保存在 12 字节中，就像 `OpenGL_ES_Ch2_1` 那个例子一样。第二个图显示的是用于每个顶点的存储的额外字节，因此在内存中在一个顶点与下一个顶点的位置坐标之间有缺口。

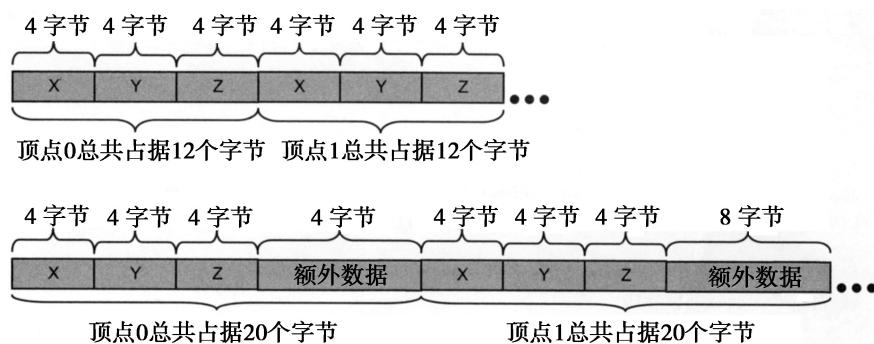


图 2-8 在顶点数组缓存内存中的顶点数据的一些可能的排列方式

`glVertexAttribPointer()` 的最后一个参数是 `NULL`，这告诉 OpenGL ES 可以从当前绑定的顶点缓存的开始位置访问顶点数据。

在第 6 步中，通过调用 `glDrawArrays()` 来执行绘图。`glDrawArrays()` 的第一个参数会告诉 GPU 怎么处理在绑定的顶点缓存内的顶点数据。这个例子会指示 OpenGL ES 去渲染三角形。`glDrawArrays()` 的第二个参数和第三个参数分别指定缓存内的需要渲染的第一个顶点的位置和需要渲染的顶点的数量。至此，在图 2-3 中显示的场景已经被完全地渲染出来或者至少在 GPU 处理完成后它就会被完全地渲染出来。请记住 GPU 运算与 CPU 运算是异步的。在这个例子中的所有代码都是运行在 CPU 上的，然后在需要进一步处理的时候向 GPU 发送命令。GPU 可能也会处理发送自 iOS 的 Core Animation 的命令，因此在任何给定的时刻 GPU 总共要执行多少处理并不一定。

3. -viewDidLoad

`OpenGL ES _Ch2_1ViewController` 实现的最后一个方法是 `-viewDidLoad` 方法。和 `-viewDidLoad` 在与视图控制器相关的视图被加载时会被自动调用一样，`-viewDidUnload` 方法会在视图最终被卸载时调用。卸载的视图将不再被绘制，因此任何只是在绘制时需要的 OpenGL ES 缓存都可以被安全地删除。

第 7 步是删除不再需要的顶点缓存和上下文。设置 `vertexBufferID` 为 0 避免了在对应的缓存被删除以后还使用其无效的标识符。设置视图的上下文属性为 `nil` 并设置当前上下文为 `nil`，以便让 Cocoa Touch 收回所有上下文使用的内存和其他资源。

```
///////////
// Called when the view controller's view has been unloaded
// Perform clean-up that is possible when you know the view
// controller's view won't be asked to draw again soon.
- (void)viewDidUnload
{
    [super viewDidUnload];

    // Make the view's context current
    GLKView *view = (GLKView *)self.view;
    [EAGLContext setCurrentContext:view.context];

    // Delete buffers that aren't needed when view is unloaded
    if (0 != vertexBufferID)
    {
        glDeleteBuffers (1,           // STEP 7
                        &vertexBufferID);
        vertexBufferID = 0;
    }

    // Stop using the context created in -viewDidLoad
    ((GLKView *)self.view).context = nil;
    [EAGLContext setCurrentContext:nil];
}

@end
```

2.3.5 支持文件

在图 2-6 中显示的组内的 png 文件是 OpenGL_ES_Ch2_1 应用的图标。操作系统会根据应用所运行的设备是 iPhone、iPod Touch 还是 iPad 而自动地选择正确的图标。OpenGL_ES_for_iOS_72x72.png 文件包含 iPad 要用到的图标图像。OpenGL_ES_for_iOS_114x114.png 和 OpenGL_ES_for_iOS_57x57.png 包含 iPod Touch 和 iPhone 要用到的图标图像。扩展名 .png 代表 Portable Network Graphics (PNG)。iOS 设备原生支持 PNG 文件并且会按照国际标准化组织的标准存储图像。

OpenGL_ES_Ch2_1-Info.plist 文件是在新工程创建的时候由 Xcode 自动生成的。OpenGL_ES_Ch2_1-Info.plist 保存诸如应用的版本号、使用的 storyboard 文件的名字、图标文件的名字等配置信息。添加应用特有的信息到这个文件是可能的，但是 OpenGL_ES_Ch2_1 应用不需要任何非标准的信息。配置文件只会在应用每次启动的时候由应用读取一次。

InfoPlist.strings 文件是在新工程创建的时候由 Xcode 自动生成的。InfoPlist.strings 文件包含在 OpenGL_ES_Ch2_1-Info.plist 文件中使用的字符串的本地化版本。本地化是指为不同的语言和文化团体提供专门的文本和图像。InfoPlist.strings 提供了一个根据用户的地区为相同的应用提供不同的图标和用户界面的方式。

1. main() 函数

main.m 文件是工程创建的时候由 Xcode 自动生成的并且包含应用的 main() 函数的实现。main() 函数被操作系统调用来开启 ANSI C、C++ 和 Objective-C 程序的运行。生成的 main.m 文件包含如下代码：

```
//  
//  main.m  
//  OpenGL_ES_Ch2_1  
  
  
#import <UIKit/UIKit.h>  
  
#import "OpenGL_ES_Ch2_1AppDelegate.h"  
  
int main(int argc, char *argv[])  
{  
    @autoreleasepool  
    {  
        return UIApplicationMain(argc, argv, nil,  
                               NSStringFromClass([OpenGL_ES_Ch2_1AppDelegate class]));  
    }  
}
```

`main()` 函数使用 Objective-C 的 `@autoreleasepool` 关键字来开启自动引用计数[⊖] (Automatic Reference Counting)。`main()` 函数还会调用 `UIApplicationMain()` 函数，这个函数创建了包含 `UIApplication` 实例在内的应用的关键对象，同时开始处理用户事件。`NSStringFromClass([OpenGLES_Ch2_AppDelegate class])` 表达式指定了与新创建的 `UIApplication` 实例一起使用的应用委托类的名字。`UIApplication` 创建了一个充满整个显示屏的 `UIWindow`，同时加载了一个或者多个 `storyboard` 文件来构建应用的用户界面。`UIApplicationMain()` 函数不会干涉 `UIApplication` 的执行，直到应用退出才会返回。

2. 预编译头文件

`OpenGLES_Ch2_1-Prefix.pch` 文件是由 Xcode 自动生成的并且改善了这个例子中应用的编译速度。扩展名 `.pch` 代表预编译头文件 (pre-compiled header)。

3. Frameworks 和 Products

`Frameworks` 是 iOS 应用用到的系统库和资源。在 Xcode 文件列表中的 `Frameworks` 文件夹指定了应用使用了哪些框架。`OpenGLES_Ch2_1` 例子使用了 `OpenGL.framework`、`QuartzCore.framework`、`GLKit.framework`、`UIKit.framework`、`Foundation.framework` 和 `CoreGraphics.framework`。

`Products` 文件夹包含编译 Xcode 工程中指定的应用时产生的结果。

2.4 深入探讨 GLKView 是怎么工作的

在介绍 iOS 5 的 `GLKit` 之前，对于每个开发者来说创建一个类似于 `GLKView` 的 `UIView` 的子类是必要的。苹果没有提供 `GLKit` 类的源代码，但是根据 `OpenGL ES` 可以推断出类似 `GLKView` 这样的类的主要功能可能的实现方式。本节剩下的部分和 `OpenGLES_Ch2_2` 这个例子会介绍 `AGLKView` 类以及它对于 `GLKView` 的部分重新实现。`AGLKView` 类不应该用在产品代码中，它仅仅是为了消除对于 `GLKView`、`Core Animation` 和 `OpenGL ES` 之间的交互的神秘感。不管从哪个方面来说，苹果优化的、测试的、面向未来的对于 `GLKit` 的实现都是最好的。如果你对于深入了解 `GLKView` 没有兴趣，可以跳到下一节。

注意 如果你不想在你的应用中使用苹果的 `GLKit` 类，你就需要使用 `AGLKView` 示例或者自己再创建一个相似的类。本书剩下的例子都假设使用了 `GLKit`。因此 `AGLKView` 不会出现在本书其他的任何例子中。

[⊖] 一个在 Cocoa Touch 应用中提供自动对象内存管理的编译器级语言功能。

在 OpenGL_ES_Ch2_2 例子中的 AGLKView 类继承自 Cocoa Touch 的 UIView 类，下面是这个类的接口声明，与苹果的 GLKView 类接口相似。

```

//  

//  AGLKView.h  

//  OpenGL_ES_Ch2_1  

//  

#import <UIKit/UIKit.h>  

#import <OpenGLES/ES2/gl.h>  

#import <OpenGLES/ES2/glext.h>  

@class EAGLContext;  

@protocol AGLKViewDelegate;  

/////////////////////////////  

// This subclass of the Cocoa Touch UIView class uses OpenGL ES  

// to render pixel data into a Frame Buffer that shares pixel  

// color storage with a Core Animation Layer.  

@interface AGLKView : UIView  

{  

    EAGLContext      *context;  

    GLuint           defaultFrameBuffer;  

    GLuint           colorRenderBuffer;  

    GLint            drawableWidth;  

    GLint            drawableHeight;  

}  

@property (nonatomic, weak) IBOutlet id <AGLKViewDelegate> delegate;  

@property (nonatomic, retain) EAGLContext *context;  

@property (nonatomic, readonly) NSInteger drawableWidth;  

@property (nonatomic, readonly) NSInteger drawableHeight;  

- (void)display;  

@end  

#pragma mark - AGLKViewDelegate  

@protocol AGLKViewDelegate <NSObject>  

@required  

- (void)glkView:(AGLKView *)view drawInRect:(CGRect)rect;  

@end

```

AGLKViewDelegate 协议指定了一个任何 AGLKView 的委托都必须实现的方法。如果 AGLKView 实例的委托属性不等于 nil，每个 AGLKView 实例都会向它的委托发送 “-glkView:drawInRect:” 消息。

AGLKView 的实现比较简单易懂，但重写了来自 UIView 的多个方法并添加了一些用于支持 OpenGL ES 绘图的方法。

```
//  
//  AGLKView.m  
//  OpenGLES_Ch2_1  
  
#import "AGLKView.h"  
#import <QuartzCore/QuartzCore.h>  
  
@implementation AGLKView  
  
@synthesize delegate;  
@synthesize context;  
  
/////////////////////////////  
// This method returns the CALayer subclass to be used by  
// CoreAnimation with this view  
+ (Class)layerClass  
{  
    return [CAEAGLLayer class];  
}
```

每一个 UIView 实例都有一个相关联的被 Cocoa Touch 按需自动创建的 Core Animation 层。Cocoa Touch 会调用 “+layerClass” 方法来确定要创建什么类型的层。在这个例子中，AGLKView 类重写了继承自 UIView 的实现。当 Cocoa Touch 调用 AGLKView 实现的 “+layerClass” 方法时，它被告知要使用一个 CAEAGLLayer 类的实例，而不是原先的 CALayer。CAEAGLLayer 是 Core Animation 提供的标准层类之一。CAEAGLLayer 会与一个 OpenGL ES 的帧缓存共享它的像素颜色仓库。

接下来的代码块实现了 “-initWithFrame:context:” 方法并重写了继承来的 “-initWithCoder:” 方法。“-initWithFrame:context:” 方法初始化了通过代码手动分配的实例。“-initWithCoder:” 方法是 Cocoa Touch 用于初始化对象的标准方法之一。Cocoa Touch 会自动调用 “-initWithCoder:” 方法，这是反归档先前归档入一个文件的对象的过程的一部分。归档和反归档在其他流行的面向对象的框架中（比如 Java 和微软的 .NET）叫做串行化和反串行化。当 OpenGLES_Ch2_2 应用启动时，在这个例子中使用的 AGLKView 实例会自动地从应用的 storyboard 文件中加载（又叫做反归档）。

之后的两个方法的代码几乎是相同的。每个实例初始化的时候只有一个方法会被调

用。两个方法首先都会给超类 UIView 一个执行其初始化的机会，然后再执行这个例子需要的 Core Animation 和 OpenGL ES 上下文的一次性初始化。第一步是初始化视图的 Core Animation 层的本地指针，具体代码如下：

```
CAEAGLLayer *eaglLayer = (CAEAGLLayer *)self.layer;
```

需要 C 语言的类型转换 (CAEAGLLayer *)，这是因为 UIView 的 “-layer” 方法返回的是一个 CALayer 实例的指针。在 AGLKView 类的实现中，真正使用的是 CAEAGLLayer 类型，因此强制编译器接受 CAEAGLLayer 类型的这个转换是安全的。

```
///////////
// This method is designated initializer for the class
- (id)initWithFrame:(CGRect)frame context:(EAGLContext *)aContext;
{
    if ((self = [super initWithFrame:frame]))
    {
        CAEAGLLayer *eaglLayer = (CAEAGLLayer *)self.layer;

        eaglLayer.drawableProperties =
            [NSDictionary dictionaryWithObjectsAndKeys:
                [NSNumber numberWithBool:NO],
                kEAGLDrawablePropertyRetainedBacking,
                kEAGLColorFormatRGBA8,
                kEAGLDrawablePropertyColorFormat,
                nil];

        self.context = aContext;
    }

    return self;
}

///////////
// This method is called automatically to initialize each Cocoa
// Touch object as the object is unarchived from an
// Interface Builder .xib or .storyboard file.
- (id)initWithCoder:(NSCoder *)coder
{
    if ((self = [super initWithCoder:coder]))
    {
        CAEAGLLayer *eaglLayer = (CAEAGLLayer *)self.layer;

        eaglLayer.drawableProperties =
            [NSDictionary dictionaryWithObjectsAndKeys:
                [NSNumber numberWithBool:NO],
                kEAGLDrawablePropertyRetainedBacking,
```

```

        kEAGLColorFormatRGBA8,
        kEAGLDrawablePropertyColorFormat,
        nil];
}

return self;
}

```

每个 AGLKView 的初始化方法会使用一个临时的 NSDictionary 实例来设置 eaglLayer 的 drawableProperties 属性。Dictionary 会保存键值对的集合。每个值都可以用对应的键快速获取。NSDictionary 是一个 Cocoa Touch 类，在这里被 CAEAGLLayer 类实例使用是为了保存层中用到的 OpenGL ES 的帧缓存类型的信息。

```

eaglLayer.drawableProperties =
[NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithBool:NO],
    kEAGLDrawablePropertyRetainedBacking,
    kEAGLColorFormatRGBA8,
    kEAGLDrawablePropertyColorFormat,
    nil];

```

这个示例设置 kEAGLDrawablePropertyRetainedBacking 键的值为 NO 并设置 kEAGLDrawablePropertyColorFormat 键的值为 kEAGLColorFormatRGBA8。不使用“保留背景”的意思是告诉 Core Animation 在层的任何部分需要在屏幕上显示的时候都要绘制整个层的内容。换句话说，这段代码是告诉 Core Animation 不要试图保留任何以前绘制的图像留作以后重用。RGBA8 颜色格式是告诉 Core Animation 用 8 位来保存层内的每个像素的每个颜色元素的值。

两个手动实现的访问器方法用于设置和返回视图的特定于平台的 OpenGL ES 上下文。因为 AGLKView 实例需要创建和配置一个帧缓存和一个像素颜色渲染缓存来与视图的 Core Animation 层一起使用，所以设置上下文会引起一些副作用。由于上下文保存缓存，因此修改视图的上下文会导致先前创建的所有缓存全部失效，并需要创建和配置新的缓存。

会受缓存操作影响的上下文是在调用 OpenGL ES 函数之前设定为当前上下文的。请注意在下面的代码中，创建帧缓存和渲染缓存会遵循一些适用于其他类型的缓存的相同的步骤，包括在 OpenGL_ES_Ch2_1 例子中的顶点数组缓存。一个新的步骤会调用 glFramebufferRenderbuffer() 函数来配置当前绑定的帧缓存以便在 colorRenderBuffer 中保存渲染的像素颜色。

```

///////////////////////////////
// This method sets the receiver's OpenGL ES Context. If the
// receiver already has a different Context, this method deletes
// OpenGL ES Frame Buffer resources in the old Context and the

```

```

// recreates them in the new Context.
- (void)setContext:(EAGLContext *)aContext
{
    if(context != aContext)
    { // Delete any buffers previously created in old Context
        [EAGLContext setCurrentContext:context];

        if (0 != defaultFrameBuffer)
        {
            glDeleteFramebuffers(1, &defaultFrameBuffer); // Step 7
            defaultFrameBuffer = 0;
        }

        if (0 != colorRenderBuffer)
        {
            glDeleteRenderbuffers(1, &colorRenderBuffer); // Step 7
            colorRenderBuffer = 0;
        }
    }

    context = aContext;

    if(nil != context)
    { // Configure the new Context with required buffers
        context = aContext;
        [EAGLContext setCurrentContext:context];

        glGenFramebuffers(1, &defaultFrameBuffer); // Step 1
        glBindFramebuffer( // Step 2
            GL_FRAMEBUFFER,
            defaultFrameBuffer);

        glGenRenderbuffers(1, &colorRenderBuffer); // Step 1
        glBindRenderbuffer( // Step 2
            GL_RENDERBUFFER,
            colorRenderBuffer);

        // Attach color render buffer to bound Frame Buffer
        glFramebufferRenderbuffer(
            GL_FRAMEBUFFER,
            GL_COLOR_ATTACHMENT0,
            GL_RENDERBUFFER,
            colorRenderBuffer);
    }
}
}

///////////////////////////////
// This method returns the receiver's OpenGL ES Context
- (EAGLContext *)context

```

```
{
    return context;
}
```

下面的“-display”方法设置视图的上下文为当前上下文，告诉OpenGL ES让渲染填满整个帧缓存，调用视图的“-drawRect:”方法来实现用OpenGL ES函数进行真正的绘图，然后让上下文调整外观并使用Core Animation合成器把帧缓存的像素颜色渲染缓存与其他相关层混合起来。

```
///////////////////////////////
// Calling this method tells the receiver to redraw the contents
// of its associated OpenGL ES Frame Buffer. This method
// configures OpenGL ES and then calls -drawRect:
- (void)display;
{
    [EAGLContext setCurrentContext:self.context];
    glViewport(0, 0, self.drawableWidth, self.drawableHeight);

    [self drawRect:[self bounds]];

    [self.context presentRenderbuffer:GL_RENDERBUFFER];
}
```

glViewport()函数可以用来控制渲染至帧缓存的子集，但是在这个例子中使用的是整个帧缓存。

如果视图的委托属性不是nil，“-drawRect:”方法会调用委托的“-glkView:drawInRect:”方法。没有委托，AGLKView什么都不会绘制。AGLKView的子类可以通过重写继承的“-drawRect:”实现来绘图，即使是没有指定委托。“-glkView:drawInRect:”的参数是一个要被绘制的视图和一个覆盖整个视图范围的矩形。

```
/////////////////////////////
// This method is called automatically whenever the receiver
// needs to redraw the contents of its associated OpenGL ES
// Frame Buffer. This method should not be called directly. Call
// -display instead which configures OpenGL ES before calling
// -drawRect:
- (void)drawRect:(CGRect)rect
{
    if(self.delegate)
    {
        [self.delegate glkView:self drawInRect:[self bounds]];
    }
}
```

任何在接收到视图重新调整大小的消息时，Cocoa Touch都会调用下面的-layoutSubviews方法。视图附属的帧缓存和像素颜色渲染缓存取决于视图的尺寸。视图会自

动地调整相关层的尺寸。上下文的“-renderbufferStorage:fromDrawable:”方法会调整视图的缓存的尺寸以匹配层的新尺寸。

```
///////////
// This method is called automatically whenever a UIView is
// resized including just after the view is added to a UIWindow.
- (void)layoutSubviews
{
    CAEAGLLayer *eaglLayer = (CAEAGLLayer *)self.layer;

    // Initialize the current Frame Buffer's pixel color buffer
    // so that it shares the corresponding Core Animation Layer's
    // pixel color storage.
    [context renderbufferStorage:GL_RENDERBUFFER
        fromDrawable:eaglLayer];

    // Make the Color Render Buffer the current buffer for display
    glBindRenderbuffer(GL_RENDERBUFFER, colorRenderBuffer);

    // Check for any errors configuring the render buffer
    GLenum status = glCheckFramebufferStatus(
        GL_FRAMEBUFFER) ;

    if(status != GL_FRAMEBUFFER_COMPLETE)
    {
        NSLog(@"failed to make complete frame buffer object %x",
              status);
    }
}
```

“-drawableWidth”和“-drawableHeight”方法是它们各自的属性的访问器。它们被实现用来通过OpenGL ES的glGetRenderbufferParameteriv()方法获取和返回当前上下文的帧缓存的像素颜色渲染缓存的尺寸。

```
///////////
// This method returns the width in pixels of current context's
// Pixel Color Render Buffer
- (NSInteger)drawableWidth;
{
    GLint          backingWidth;

    glGetRenderbufferParameteriv(
        GL_RENDERBUFFER,
        GL_RENDERBUFFER_WIDTH,
        &backingWidth);

    return (NSInteger)backingWidth;
}
```

```
///////////
// This method returns the height in pixels of current context's
// Pixel Color Render Buffer
- (NSInteger)drawableHeight;
{
    GLint          backingHeight;

    glGetRenderbufferParameteriv(
        GL_RENDERBUFFER,
        GL_RENDERBUFFER_HEIGHT,
        &backingHeight);

    return (NSInteger)backingHeight;
}
```

最后，在一个对象可以被回收时 Cocoa Touch 会自动地调用“-dealloc”方法，然后它的资源就会返回给操作系统。AGLKView 实现“-dealloc”方法是为了确保视图的上下文不再是当前上下文，其次是为了设置上下文属性为 nil。如果在属性变成 nil 之后，视图的上下文不再被使用了，那这个上下文也会被自动回收。

```
///////////
// This method is called automatically when the reference count
// for a Cocoa Touch object reaches zero.
- (void)dealloc
{
    // Make sure the receiver's OpenGL ES Context is not current
    if ([EAGLContext currentContext] == context)
    {
        [EAGLContext setCurrentContext:nil];
    }

    // Deletes the receiver's OpenGL ES Context
    context = nil;
}

@end
```

关于 AGLKView 类就介绍这些内容。例子 OpenGLES_Ch2_2 还包含一个与 GLKit 的 GLKViewController 类相似的 AGLKViewController 类。除了在显示 GLKit 类的运行机制时使用了 AGLKView 和 AGLKViewController 类而不是 GLKit 类之外，OpenGLES_Ch2_2 和 OpenGLES_Ch2_1 是相同的。

与 GLKit 的 GLKViewController 类相似，AGLKViewController 使用一个 Core Animation CADisplayLink 对象来调度和执行与控制器相关联的视图的周期性的重绘。CADisplayLink 本质上是一个用于显示更新的同步计时器，它能够被设置用来在每个显示更新或者其他更新时发送一个消息。CADisplayLink 计时器的周期是以显示更新来计量的。

显示更新率通常是由嵌入设备的硬件决定的，它代表一个帧缓存的内容每秒最多能够被在屏幕上通过的像素显示出来的次数。因此来自 CADisplayLink 的消息为重新渲染一个场景提供了理想的触发器。渲染速度如果快于显示刷新率就是一种浪费，因为用户永远看不到两次显示刷新之间的额外的帧缓存的更新。

2.5 对于 GLKit 的推断

GLKit 类封装并简化了 Cocoa Touch 应用和 OpenGL ES 之间的常见交互。就像推测已存的 GLKit 类可能是怎么实现的是一种指导一样，遵循苹果的用来构建 GLKit 的范例来创建新类也是可能的。

本章的 OpenGLES_Ch2_1 例子使用了 GLKit，并且为了实现两个目的添加了对于 OpenGL ES 函数的直接调用，这两个目的是：清除帧缓存，以及使用一个顶点数组缓存来绘图。就像 GLKit 的 GLKView 封装了帧缓存和层管理，例子 OpenGLES_Ch2_3 重构 OpenGLES_Ch2_1 的可重用 OpenGL ES 代码为两个新类：AGLContext 和 AGLKVertexAttribArrayBuffer。本书使用 AGLK 作为类和函数的前缀来代表是对于 GLKit 类的追加。GLKit 将来的版本可能包含与 AGLK 类相似的类。苹果通常不会预告对于其框架的增强，并且即使苹果最终实现了与本书中的 AGLK 类具有相似功能的类，也无法保证苹果会按相同的方式实现它。

AGLContext 类是一个在例子 OpenGLES_Ch2_1 中使用的内建的 EAGLContext 类的简单子类。对于本例来说，AGLContext 仅仅添加了一个 clearColor 属性和一个用来告诉 OpenGL ES 去设置在上下文的帧缓存中的每个像素颜色为 clearColor 的元素值的“-clear:”方法。

```
//  
//  GLKContext.h  
//  OpenGLES_Ch2_3  
  
#import <GLKit/GLKit.h>  
  
@interface AGLKContext: EAGLContext  
{  
    GLKVector4 clearColor;  
}  
@property (nonatomic, assign) GLKVector4  
    clearColor;  
  
- (void)clear:(GLbitfield)mask;  
  
@end
```

AGLContext 的实现实现了“-clear:”方法和用于 clearColor 属性的访问器。通过调用一个 OpenGL ES 的 glClearColor() 函数来设置 clearColor 属性。“-clear:”方法通过调用 glClear() 方法来实现。

```
//  
//  GLKContext.m  
//  OpenGLES_Ch2_3  
//  
  
#import "AGLContext.h"  
  
@implementation AGLContext  
  
/////////////////////////////  
// This method sets the clear (background) RGBA color.  
// The clear color is undefined until this method is called.  
- (void)setClearColor:(GLKVector4)clearColorRGBA  
{  
    clearColor = clearColorRGBA;  
  
    NSAssert(self == [[self class] currentContext],  
        @"Receiving context required to be current context");  
  
    glClearColor(  
        clearColorRGBA.r,  
        clearColorRGBA.g,  
        clearColorRGBA.b,  
        clearColorRGBA.a);  
}  
  
/////////////////////////////  
// Returns the clear (background) color set via -setClearColor:.  
// If no clear color has been set via -setClearColor:, the  
// return clear color is undefined.  
- (GLKVector4)clearColor  
{  
    return clearColor;  
}  
  
/////////////////////////////  
// This method instructs OpenGL ES to set all data in the  
// current Context's Render Buffer(s) identified by mask to  
// colors (values) specified via -setClearColor: and/or  
// OpenGL ES functions for each Render Buffer type.  
- (void)clear:(GLbitfield)mask  
{  
    NSAssert(self == [[self class] currentContext],  
        @"Receiving context required to be current context");
```

```
    glClear(mask);
}
```

```
@end
```

GLKVertexAttribArrayBuffer 类封装了使用 OpenGL ES 2.0 的顶点属性数组缓存（或者简称“顶点缓存”）的所有 7 个步骤。这个函数减少了应用需要调用的 OpenGL ES 函数的数量。在后续章节的例子中重用 AGLKVertexAttribArrayBuffer 会在尽可能减少与 7 个缓存管理步骤相关的错误的同时减少编写的代码量。

```
//
// GLKVertexAttribArrayBuffer.h
// OpenGLES_Ch2_3
//

#import <GLKit/GLKit.h>

@class AGLKElementIndexArrayBuffer;

@interface AGLKVertexAttribArrayBuffer : NSObject
{
    GLsizeiptr    stride;
    GLsizeiptr    bufferSizeBytes;
    GLuint        glName;
}

@property (nonatomic, readonly) GLuint
glName;
@property (nonatomic, readonly) GLsizeiptr
bufferSizeBytes;
@property (nonatomic, readonly) GLsizeiptr
stride;

- (id)initWithAttribStride:(GLsizeiptr)stride
numberOfVertices:(GLsizei)count
data:(const GLvoid *)dataPtr
usage:(GLenum)usage;

- (void)prepareToDrawWithAttrib:(GLuint)index
numberOfCoordinates:(GLint)count
attribOffset:(GLsizeiptr)offset
shouldEnable:(BOOL)shouldEnable;

- (void)drawArrayWithMode:(GLenum)mode
startVertexIndex:(GLint)first
numberOfVertices:(GLsizei)count;

@end
```

下面的实现在三个方法中封装了 7 个缓存管理步骤。AGLKVertexAttribArrayBuffer 类包含一些错误检查代码，但除此之外还包含一个对于例子 OpenGL_ES_Ch2_1 中的缓存管理代码的简单重用和重构。除了在类接口中声明的 3 个方法之外，还实现了一个“-dealloc”方法来删除一个相关联的 OpenGL ES 缓存标识符。

```

//  

//  GLKVertexAttribArrayBuffer.m  

//  OpenGL_ES_Ch2_3  

//  

#import "AGLKVertexAttribArrayBuffer.h"  

@interface AGLKVertexAttribArrayBuffer ()  

@property (nonatomic, assign) GLsizeiptr  

    bufferSizeBytes;  

@property (nonatomic, assign) GLsizeiptr  

    stride;  

@end  

@implementation AGLKVertexAttribArrayBuffer  

@synthesize glName;  

@synthesize bufferSizeBytes;  

@synthesize stride;  

//  

// This method creates a vertex attribute array buffer in  

// the current OpenGL ES context for the thread upon which this  

// method is called.  

- (id)initWithAttribStride:(GLsizeiptr)aStride  

    numberOfRowsInSection:(GLsizei)count  

    data:(const GLvoid *)dataPtr  

    usage:(GLenum)usage;  

{  

    NSParameterAssert(0 < aStride);  

    NSParameterAssert(0 < count);  

    NSParameterAssert(NULL != dataPtr);  

    if(nil != (self = [super init]))  

    {  

        stride = aStride;  

        bufferSizeBytes = stride * count;

```



```

///////////
// Submits the drawing command identified by mode and instructs
// OpenGL ES to use count vertices from the buffer starting from
// the vertex at index first. Vertex indices start at 0.
- (void)drawArrayWithMode:(GLenum)mode
    startVertexIndex:(GLint)first
    numberOfVertices:(GLsizei)count
{
    NSAssert(self.bufferSizeBytes >=
        ((first + count) * self.stride),
        @"Attempt to draw more vertex data than available.");

    glDrawArrays(mode, first, count); // Step 6
}

///////////
// This method deletes the receiver's buffer from the current
// Context when the receiver is deallocated.
- (void)dealloc
{
    // Delete buffer from current context
    if (0 != glName)
    {
        glDeleteBuffers (1,           // STEP 7
                        &glName);
        glName = 0;
    }
}
@end

```

仔细检查例子 OpenGLES_Ch2_3 中的 OpenGLES_Ch2_3ViewController 类的实现，并把它与例子 OpenGLES_Ch2_1 中的 OpenGLES_Ch2_1ViewController 类的实现做下对比。在 OpenGLES_Ch2_3ViewController 中没有对于 OpenGL ES 函数的直接调用。另外，你可能会注意到一些小的代码风格的不一致。例如 OpenGLES_Ch2_1 使用一个 GLKVector4 结构体来设置恒定的颜色，但是用来设置清除颜色的对于 OpenGL ES 函数的调用可以直接接收 RGBA 颜色元素值：

```

self.baseEffect.constantColor = GLKVector4Make(
    1.0f, // Red
    1.0f, // Green
    1.0f, // Blue
    1.0f); // Alpha

// Set the background color stored in the current context
glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // background color

```

AGLKVertexAttribArrayBuffer 和 AGLKContext 类扩展了 GLKit 引入的代码样式，因此 OpenGL_ES_Ch2_3 例子会使用一个一贯的样式：

```
self.baseEffect.constantColor = GLKVector4Make(
    1.0f, // Red
    1.0f, // Green
    1.0f, // Blue
    1.0f); // Alpha

// Set the background color stored in the current context
((AGLKContext *)view.context).clearColor = GLKVector4Make(
    0.0f, // Red
    0.0f, // Green
    0.0f, // Blue
    1.0f); // Alpha
```

细小的样式一致性上的改进会让应用代码更整洁，但更重要的是，在 Objective-C 类中封装 OpenGL ES 代码会让代码重用更容易。减少缓存被创建并使用时出现程序错误的机会。同时，可让使用多个缓存的应用需要编写、测试和调试的代码更少。

2.6 小结

所有的图形 iOS 应用都包含 Core Animation 层。所有的绘图都发生在层之上。Core Animation 合成器会混合当前应用与操作系统的层，从而在 OpenGL ES 的后帧缓存中产生最终的像素颜色。之后 Core Animation 合成器切换前后缓存以便把合成的内容显示到屏幕上。

原生 iOS 应用使用的是苹果的 Cocoa Touch 框架，并且由 Xcode IDE 组建而成。由 Xcode 生成的 main.m 文件包含建立 Cocoa Touch 应用结构和从 storyboard 文件中加载应用的用户界面的代码。一个标准的 Cocoa Touch 对象集合会提供标准 iOS 应用的所有功能。特定于某一个应用的功能是通过修改在标准应用结构中的应用委托或者根视图控制器对象来实现的。复杂的应用会向 Xcode 工程添加很多的额外文件。

使用 OpenGL ES 的 Cocoa Touch 应用要么使用 Core Animation 层配置一个与一个 OpenGL ES 的帧缓存分享内存的自定义 UIView，要么使用内建的 GLKView 类。苹果的 GLKit 框架提供了 GLKView 以处理细节，因此开发者很少会重新创建 UIView 的子类。一个 OpenGL ES 的上下文会存储当前 OpenGL ES 的状态，并控制 GPU 硬件。每个 GLKView 实例都需要一个上下文。

本章中的 OpenGL_ES_Ch2_1 例子为接下来的例子奠定了基础。所有本例专有的代码都是在 OpenGL_ES_Ch2_1ViewController 类中实现的。应用的 storyboard 文件指定了 GLKView 实例，这个实例使用根视图控制器作为委托。委托会接收来自其他对象的消息。

息并在响应中执行特定应用的操作或者控制其他对象。

例子中 `OpenGLES_Ch2_1ViewController` 类的实现包含 3 个方法：一个会在视图从 storyboard 文件中加载时被自动调用，一个会在每次视图需要被重绘时被自动调用，还有一个会在视图卸载时被自动调用。这三个方法分别通过创建必要的上下文和顶点缓存来初始化 OpenGL ES，使用上下文和顶点缓存在 GLKView 的与 Core Animation 层分享内存的帧缓存中绘图，并删除上下文和顶点缓存。

`GLKBaseEffect` 类隐藏了 iOS 所支持的 OpenGL ES 版本之间的很多不同。当使用 OpenGL ES 2.0 时，`GLKBaseEffect` 会生成直接在 GPU 上运行的 Shading Language 程序。`GLKBaseEffect` 使程序员专注于应用的功能和图形概念，而无须学习 Shading Language。

本章中的 `OpenGLES_Ch2_3` 例子把 `OpenGLES_Ch2_1` 例子中的 OpenGL ES 函数调用重构为 `AGLKVertexAttribArrayBuffer` 类和 `AGLContext` 类。

第 3 章会扩展 `OpenGLES_Ch2_1` 示例以使三角形渲染更加有趣和强大。



第3章 纹理

第2章的例子演示了怎么使用OpenGL ES 2.0渲染和显示一个白色的三角形。现实中的对象很少仅由单一的白色或者其他固定的颜色组成。OpenGL ES提供了一个为几何图形中的每个顶点设置不同颜色的方法，不过实际上即使是拥有多重颜色顶点的三角形也无法在渲染的场景中提供逼真的感觉。本章会介绍一个叫做纹理的图形技术，它可以控制一个渲染的三角形中的每个像素的颜色。

纹理是计算机图形中最复杂的话题之一。本章会介绍纹理的底层概念和常用选项。多个例子会逐步说明这些技术。

3.1 什么是纹理

纹理是一个用来保存图像的颜色元素值的OpenGL ES缓存。图3-1显示的是本章例子中的一个纹理使用的一个图像。纹理可以使用任何图像，包括树木、面孔、砖块、云彩，或者机器。当把纹理应用到几何图形中后，会使渲染的场景显得更自然，会使三角形的复杂组合像是真实的物体而不只是有颜色的面。

在纹理的缓存中保存的颜色值可能要耗费很多的内存。所有的嵌入式系统都为纹理设定了内存的最大尺寸限制。所有的iPhone和iPad Touch版本都可以支持由 1024×1024 像素的图像生成的纹理。iPad和最新一代iPhone、iPod Touch所支持的尺寸甚至更大。但是，由于嵌入式系统的可用内存相对较小，应尽量使用最小的图像来产生可以接受的渲染结果。

当用一个图像初始化一个纹理缓存之后，在这个图像中的每个像素变成了纹理中的一个纹素(texel)。与像素类似，纹素保存颜色数据。像素和纹素之间的差别很微妙：像素通常表示计算机屏幕上的一个实际的颜色点，因此，像素通常被用来作为一个测量单位。说一个图像有256像素宽，256像素高是正确的。与此相反，纹素存在于一个虚拟

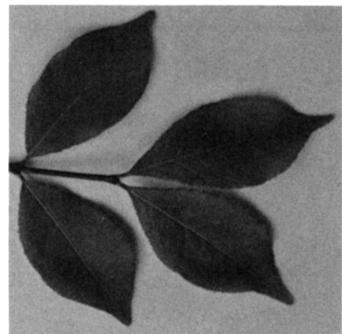


图3-1 一个用作OpenGL ES纹理的图像

的没有尺寸的数学坐标系中。图 3-2 显示了 OpenGL ES 纹理坐标系中的一个纹理。

纹理坐标系有一个命名为 S 和 T 的 2D 轴。在一个纹理中无论有多少个纹素，纹理的尺寸永远是在 S 轴上从 0.0 到 1.0，在 T 轴上从 0.0 到 1.0。从一个 1 像素高 64 像素宽的图像初始化来的纹理会沿着整个 T 轴有 1 纹素，沿着 S 轴有 64 纹素。

注意 本书中的例子使用的是 2D 纹理，但是一些 OpenGL 实现还会支持 1D 和 3D 纹理。一个 1D 的纹理就相当于沿着 T 轴只有 1 纹素的 2D 纹理，因此一个 64 纹素的 1D 纹理与从尺寸为 64×1 的图像初始化来的 2D 纹理是相同的。3D 纹理就像一个层饼，一个在拥有 R、S 和 T 轴的坐标系中沿着 R 轴堆叠的多个 2D 纹理的层饼。1D 和 3D 纹理对于某些特定的应用来说是非常方便的，但是 2D 纹理是迄今为止最常见的。

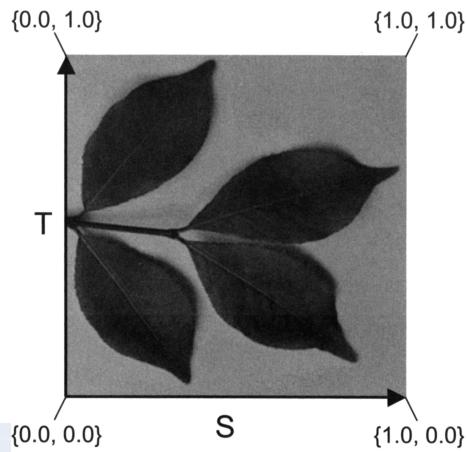


图 3-2 OpenGL ES 纹理坐标系

3.1.1 对齐纹理和几何图形

我们需要告诉 OpenGL ES 如何使用一个纹理来给几何图形对象着色。图 3-3 显示的是第 2 章例子中的简单三角形，但是这次它是用一个纹理绘制的。在图 3-3 中需要注意的第一件事是绘制的三角形的高度要比宽度大。这个三角形是用 $\{-0.5f, -0.5f, 0.0\}$ 、 $\{0.5f, -0.5f, 0.0\}$ 、 $\{-0.5f, 0.5f, 0.0\}$ 这三个顶点定义的，这使得三角形的高度和宽度在纯数学的 OpenGL ES 坐标系中是相等的。但是这个例子的帧缓存是按像素来匹配屏幕尺寸的。在渲染时，GPU 会转换纯数学 OpenGL ES 坐标系中的每个顶点的 X、Y、Z 坐标为帧缓存中所对应的真实像素位置。帧缓存中的像素位置叫做视口（viewport）坐标。第 2 章曾简短地提到过视口。转换为视口坐标的结果是所有绘制的几何图形都被拉伸以适合屏幕大小，也就是说高比宽大了。第 5 章会讲解怎么控制 GPU 的这个转换。

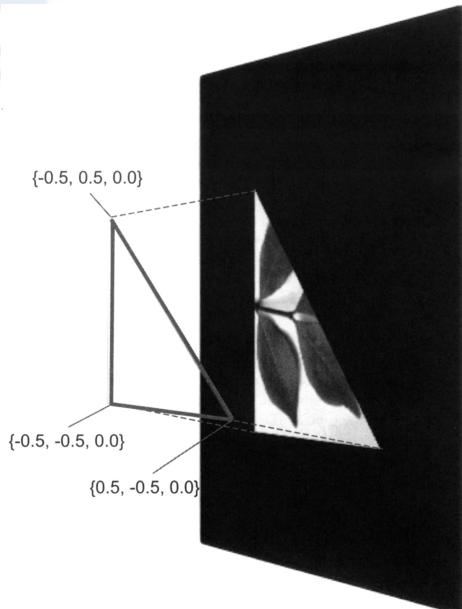


图 3-3 一个渲染进帧缓存的纹理三角形

注意 一个场景中的每个几何坐标的转换都要耗费一打的矢量数学运算，但是 GPU 经常会在每秒执行 10 亿次矢量运算。与内存读取相比这个转换是如此之快，以至于这个转换在实际的应用中几乎不会产生性能影响。

在每个顶点的 X、Y、Z 坐标被转换成视口坐标后，GPU 会设置转换生成的三角形内的每个像素的颜色。转换几何形状数据为帧缓存中的颜色像素的渲染步骤叫做点阵化（rasterizing），每个颜色像素叫做片元（fragment）。当 OpenGL ES 没有使用纹理时，GPU 会根据包含该片元的对象的顶点的颜色来计算每个片元的颜色。当设置了使用纹理后，GPU 会根据在当前绑定的纹理缓存中的纹素来计算每个片元的颜色。

程序需要指定怎么对齐纹理和顶点，以便让 GPU 知道每个片元的颜色由哪些纹素决定。这个对齐又叫做映射（mapping），是通过扩展为每个顶点保存的数据来实现的：除了 X、Y、Z 坐标，每个顶点还给出了 U 和 V 坐标值。每个 U 坐标会映射顶点在视口中的最终位置到纹理中的沿着 S 轴的一个位置。V 坐标映射到 T 轴。图 3-4 形象化了 3.2 节将介绍的 OpenGL_ES_Ch3_1 例子中演示的纹理映射。

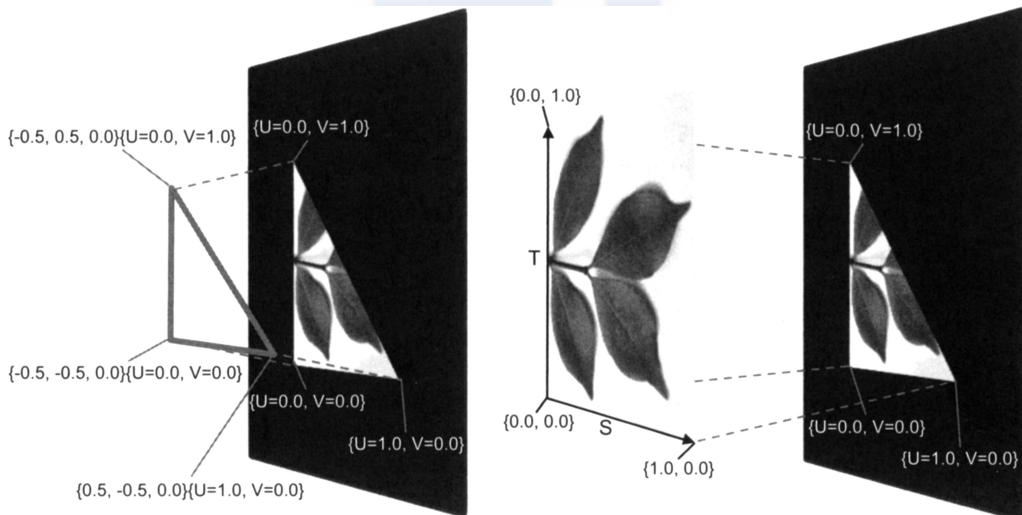


图 3-4 纹理映射到转换后的顶点

3.1.2 纹理的取样模式

每个顶点的 U 和 V 坐标会附加到每个顶点在视口坐标中的最终位置。然后 GPU 会根据计算出来的每个片元的 U、V 位置从绑定的纹理中选择纹素。这个选择过程叫做取样（Sampling）。取样会把纹理的 S 和 T 坐标系与每个渲染的三角形的顶点的 U、V 坐标匹配起来。如图 3-4 所示，在 S 和 T 坐标系中与 {0, 0} 位置最近的纹素会被映射到拥有 {0, 0} 的 U、V 坐标的顶点所对应的片元上。每个随后的片元位置对应于一个沿

着 S 和 T 轴的与该片元在 U、V 坐标中的位置等比例的位置。例如，一个 U、V 坐标为 {0.5,0.5} 的片元会被当前绑定的纹理中最接近中间位置的纹素所着色。

注意 {S,T} 坐标和 {U,V} 坐标通常是由程序员和 3D 设计师设定的。两者之间只有一个实际的差异：{U,V} 坐标可能会超出 0.0 到 1.0 的这个范围。栅格化会基于一个本节后面要介绍的可配置的“循环模式”映射 {S, T} 坐标为超出 0.0 到 1.0 范围的 {U,V} 坐标。

渲染过程中的取样可能会导致纹理被拉伸、压缩，甚至翻转。例子 OpenGL_ES_Ch3_3 会演示一些可能会产生的纹理变形。

OpenGL ES 支持多个不同的取样模式：考虑一下当一个三角形产生的片元少于绑定的纹理内的可用纹素的数量时会发生什么。一个拥有大量纹素的纹理被映射到帧缓存内的一个只覆盖几个像素的三角形中，这种情况会在任何时间发生。相反的情况也会发生。一个包含少量纹素的纹理可能会被映射到一个在帧缓存中产生很多个片元的三角形。程序会使用如下的 glTexParameter() 函数来配置每个绑定的纹理，以便使 OpenGL ES 知道怎么处理可用纹素的数量与需要被着色的片元的数量之间的不匹配。

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

使用值 GL_LINEAR 来指定参数 GL_TEXTURE_MIN_FILTER 会告诉 OpenGL ES 无论何时出现多个纹素对应一个片元时，从相配的多个纹素中取样颜色，然后使用线性内插法来混合这些颜色以得到片元的颜色。产生的片元颜色可能最终是一个纹理中不存在的颜色。例如，一个纹理是由交替的黑色和白色纹素组成的，线性取样会混合纹素的颜色，因此片元最终会是灰色的。使用 GL_NEAREST 值来指定 GL_TEXTURE_MIN_FILTER 参数会产生一个不同的结果。与片元的 U、V 坐标最接近的纹素的颜色会被取样。如果一个纹理是由交替的黑色和白色纹素组成的，GL_NEAREST 取样模式会拾取其中的一个纹素或者另一个，并且最终的片元要么是白色的，要么是黑色的。下一节（包括图 3-5）会展示线性插值和最邻近差值取样模式之间的效果对比。

glTexParameter() 的 GL_TEXTURE_MAG_FILTER 参数用于在没有足够的可用纹素来唯一地映射一个或者多个纹素到每个片元上时配置取样。在这种情况下，GL_LINEAR 值会告诉 OpenGL ES 混合附近纹素的颜色来计算片元的颜色。GL_LINEAR 值会有一个放大纹理的效果，并会让它模糊地出现在渲染的三角形上。GL_TEXTURE_MAG_FILTER 的 GL_NEAREST 值仅仅会拾取与片元的 U、V 位置接近的纹素的颜色，并放大纹理，这会使它有点像素化地出现在渲染的三角形上。

除了减小和放大过滤选项，当 U、V 坐标的值小于 0 或者大于 1 时，程序会指定要发生什么。有两个选择，要么尽可能多地重复纹理以填满映射到几何图形的整个 U、V 区域，要么每当片元的 U、V 坐标的值超出纹理的 S、T 坐标系的范围时，取样纹理边缘的纹素。纹理的循环模式是为 S 和 T 轴分别设置的，参考下面的代码：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

本章的例子 OpenGLES_Ch3_3 会演示一些由这些纹理参数值产生的效果。

3.1.3 MIP 贴图

MIP 贴图是与取样密切相关的。根据维基百科，MIP 代表拉丁短语 *Multum In Parvo*，意思是“放置很多东西的小空间”。回忆一下，内存存取是现代图形处理的薄弱环节。当有多个纹素对应一个片元时，线性取样会导致 GPU 仅仅为了计算一个片元的最终颜色而读取多个纹素的颜色值。MIP 贴图是一个为纹理存储多个细节级别的技术。高细节的纹理会沿着 S 轴和 T 轴存储很多的纹素。低细节的纹理沿着每个轴存储很少的纹素。最低细节的纹理只保存一个纹素。多个细节级别增加了在 S、T 轴上的纹素和每个片元的 U、V 坐标之间有紧密的对应关系的可能性。当存在一个紧密的对应关系时，GPU 会减少取样纹素的数量，进而会减少内存访问的次数。

使用 MIP 贴图通常会通过减少 GPU 取样的数量来提高渲染的性能，但是 MIP 贴图使每个纹理所需要的内存增加了 1/3，在网页 <http://en.wikipedia.org/wiki/Mipmap> 上做了解释。在 iOS 设备上，内存限制可能优先于渲染性能。通常最好的策略是逐个在使用和不使用 MIP 贴图的情况下测试你的应用，以决定哪种方式的效果最好。

3.2 OpenGLES_Ch3_1 示例

纹理涉及很多潜在的复杂选项，但是苹果的 GLKit 极大地简化了常见纹理的配置。本例是基于第 2 章的 OpenGLES_Ch2_3 示例建立的。在 OpenGLES_Ch3_1 中的一个单独的文件，OpenGLES_Ch3_1ViewController 类的实现文件，包含了对 OpenGLES_Ch2_3 的所有修改代码。下面的代码块用粗体标识了变化部分。首先，纹理坐标被加入到 SceneVertex 类型的声明中：

```
///////////////////////////////
// This data type is used to store information for each vertex
typedef struct {
    CLKVector3 positionCoords;
```

```

    GLKVector2 textureCoords;
}
SceneVertex;

```

纹理坐标定义了几何图形中的每个顶点的纹理映射。为例子保存顶点数据的 vertices 数组初始化了纹理坐标和位置坐标。

```

///////////////////////////////
// Define vertex data for a triangle to use in example
static const SceneVertex vertices[] =
{
    {{-0.5f, -0.5f, 0.0f}, {0.0f, 0.0f}}, // lower left corner
    {{ 0.5f, -0.5f, 0.0f}, {1.0f, 0.0f}}, // lower right corner
    {{-0.5f,  0.5f, 0.0f}, {0.0f, 1.0f}}, // upper left corner
};

```

“-viewDidLoad” 方法扩展了示例 OpenGLES_Ch2_3 中的代码来影响 GLKit 提供的 GLKTextureLoader 类，这个类用于将一个纹理图像加载到一个 OpenGL ES 纹理缓存中。

```

///////////////////////////////
// Called when the view controller's view is loaded
// Perform initialization before the view is asked to draw
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Verify the type of view created automatically by the
    // Interface Builder storyboard
    GLKView *view = (GLKView *)self.view;
    NSAssert([view isKindOfClass:[GLKView class]],
        @"View controller's view is not a GLKView");

    // Create an OpenGL ES 2.0 context and provide it to the
    // view
    view.context = [[AGLContext alloc]
        initWithAPI:kEAGLRenderingAPIOpenGLES2];

    // Make the new context current
    [AGLContext setCurrentContext:view.context];

    // Create a base effect that provides standard OpenGL ES 2.0
    // shading language programs and set constants to be used for
    // all subsequent rendering
    self.baseEffect = [[GLKBaseEffect alloc] init];
    self.baseEffect.useConstantColor = GL_TRUE;
    self.baseEffect.constantColor = GLKVector4Make(
        1.0f, // Red
        1.0f, // Green

```

```

    1.0f, // Blue
    1.0f); // Alpha

    // Set the background color stored in the current context
    ((AGLContext *)view.context).clearColor = GLKVector4Make(
        0.0f, // Red
        0.0f, // Green
        0.0f, // Blue
        1.0f); // Alpha

    // Create vertex buffer containing vertices to draw
    self.vertexBuffer = [[AGLKVertexAttribArrayBuffer alloc]
        initWithAttribStride:sizeof(SceneVertex)
        numberOfVertices:sizeof(vertices) / sizeof(SceneVertex)
        data:vertices
        usage:GL_STATIC_DRAW];

    // Setup texture
    CGImageRef imageRef =
        [[UIImage imageNamed:@"leaves.gif"] CGImage];

    GLKTextureInfo *textureInfo = [GLKTextureLoader
        textureWithCGImage:imageRef
        options:nil
        error:&NULL];

    self.baseEffect.texture2d0.name = textureInfo.name;
    self.baseEffect.texture2d0.target = textureInfo.target;
}

```

CGImageRef 是一个在苹果的 Core Graphics 框架中定义的 C 数据类型。Core Graphics 包含很多强大的 2D 图像处理和绘制函数。UIImage 的 “+imageNamed:” 方法会返回一个初始化自图形文件的 UIImage 实例。很多不同的图像文件格式都支持。命名的图像必须被包含为应用的一部分，以便 “+imageNamed:” 可以找到它。

GLKTextureLoader 的 “-textureWithCGImage:options:error:” 方法会接受一个 CGImageRef 并创建一个新的包含 CGImageRef 的像素数据的 OpenGL ES 纹理缓存。这个方法看似强大。接受一个 CGImageRef 使得图像数据的源可以是任何 Core Graphics 支持的形式，从一个电影的单个帧到由一个应用绘制的自定义 2D 图像，再到一个图像文件的内容。“options:” 参数接受一个存储了用于指定 GLKTextureLoader 怎么解析加载的图像数据的键值对的 NSDictionary。可用选项之一是指示 GLKTextureLoader 为加载的图像生成 MIP 贴图。

GLKTextureLoader 会自动调用 glTexParameter() 方法来为创建的纹理缓存设置 OpenGL ES 取样和循环模式。如果使用了 MIP 贴图，并且 GL_TEXTURE_MIN_FILTER 被设置成 GL_LINEAR_MIPMAP_LINEAR，这会告诉 OpenGL ES 使用与被取样的 S、T 坐标最接近的纹素的线性插值取样两个最合适 MIP 贴图图像尺寸

(细节级别)。然后,来自MIP贴图的两个样本被线性差值来产生最终的片元颜色。GL_LINEAR_MIPMAP_LINEAR过滤器通常会产生高质量的渲染输出,但是会比其他模式需要更多的GPU计算。如果没有使用MIP贴图,GLKTextureLoader会自动设置GL_TEXTURE_MIN_FILTER为GL_LINEAR。GL_TEXTURE_WRAP_S和GL_TEXTURE_WRAP_T都会被设置为GL_CLAMP_TO_EDGE。

第2章例子中引入的GLKBaseEffect对象提供了对于使用纹理做渲染的内建的支持。在OpenGL ES纹理缓存被创建以后,这个例子设置baseEffect的texture2d0属性使用一个新的纹理缓存。

GLKTextureInfo类封装了与刚创建的纹理缓存相关的信息,包括它的尺寸以及它是否包含MIP贴图,但是这个例子只需要缓存的OpenGL ES标识符、名字和用于纹理的OpenGL ES目标。OpenGL ES上下文会为各种缓存分别保存配置信息。帧缓存会被单独从顶点属性数组缓存或者纹理缓存配置。事实上,OpenGL ES的上下文支持多种纹理缓存。例如,一种纹理缓存会包含普通的2D图像数据,然后另一种会保存一个用于特殊效果的特别形状的图像,这些内容会在第8章中介绍。GLKTextureInfo的target属性指定被配置的纹理缓存的类型。一些OpenGL ES实现还会为1D和3D纹理保持独立的纹理缓存目标。

```
///////////
// GLKView delegate method: Called by the view controller's view
// whenever Cocoa Touch asks the view controller's view to
// draw itself. (In this case, render into a frame buffer that
// shares memory with a Core Animation Layer)
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect
{
    [self.baseEffect prepareToDraw];

    // Clear back frame buffer (erase previous drawing)
    [(AGLContext *)view.context clear:GL_COLOR_BUFFER_BIT];

    [self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribPosition
        numberOfCoordinates:3
        attribOffset:offsetof(SceneVertex, positionCoords)
        shouldEnable:YES];
    [self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribTexCoord0
        numberOfCoordinates:2
        attribOffset:offsetof(SceneVertex, textureCoords)
        shouldEnable:YES];

    // Draw triangles using the first three vertices in the
    // currently bound vertex buffer
    [self.vertexBuffer drawArrayWithMode:GL_TRIANGLES
        startVertexIndex:0
        numberOfVertices:3];
}
```

在 OpenGL ES Ch2_3 例子中创建并在这里重用的 AGLKVertexAttribArrayBuffer 类隐式地支持每个顶点的属性的任意组合。OpenGL ES Ch3_1ViewController 的 “-glkView:drawInRect:” 方法的实现首先告诉 vertexBuffer 让 OpenGL ES 为渲染顶点位置做好准备（就像例子 OpenGL ES Ch2_3），然后添加第二个对于 “-prepareToDrawWithAttrib:numberOfCoordinates:attribOffset:shouldEnable:” 方法的调用。第二个调用告诉 vertexBuffer 让 OpenGL ES 为每个顶点的两个纹理坐标的渲染做好准备。编译器会用纹理坐标开始的每个 SceneVertex 结构体内的内存偏移来代替 ANSI C 的 offsetof() 宏。

在一个纹理被赋给 baseEffect，同时 OpenGL ES 为使用位置和纹理坐标属性做好准备之后，调用 AGLKVertexAttribArrayBuffer 的 “-drawArrayWithMode:startVertexIndex:numberOfVertices:” 方法来指示 OpenGL ES 去渲染有纹理的三角形。

例子 OpenGL ES Ch3_1 对于 “-viewDidLoad” 方法的实现与例子 OpenGL ES Ch2_3 的实现保持不变。

3.3 深入探讨 GLKTextureLoader 是怎么工作的

OpenGL ES 的纹理缓存与第 1 章讨论过的其他的缓存具有相同的步骤。首先使用 glGenTextures() 函数生成一个纹理缓存标识符；然后使用 glBindTexture() 函数将其绑定到当前上下文；接下来，通过使用 glTexImage2D() 函数复制图像数据来初始化纹理缓存的内容。例子 OpenGL ES Ch3_2 显示了所有的步骤。

大部分的 OpenGL ES 实现要么需要，要么受益于使用尺寸为 2 的幂的纹理。图 3-1 中的图像为 256×256 像素。这个尺寸符合 OpenGL ES 的要求，因为 256 是 2 的幂。2 的幂包括 $2^0 = 1$ 、 $2^1 = 2$ 、 $2^2 = 4$ 、 $2^3 = 8$ 、 $2^4 = 16$ 、 $2^5 = 32$ 、 $2^6 = 64$ 、 $2^7 = 128$ 、 $2^8 = 256$ 和 $2^9 = 512$ 。一个 4×64 的纹理是有效的，一个 128×128 的纹理可以工作良好，一个 1×64 的纹理也可以。一个 200×200 的纹理要么不工作，要么根据使用的 OpenGL ES 版本在渲染时导致效率低下。限制纹理的尺寸通常不会引起任何问题。例子 OpenGL ES Ch3_2 显示了怎么生成纹理缓存，按需重新调整图像的尺寸为 2 的幂，使用图像初始化纹理缓存。

示例 OpenGL ES Ch3_2 中的 AGLKTextureLoader 是苹果的 GLKit 的 GLKTextureLoader 类的部分实现。AGLKTextureLoader 类不会用在产品代码中，它仅仅是为了消除关于 GLKTextureLoader、Core Graphics 和 OpenGL ES 之间交互的神秘感。

使用 AGLKTextureLoader 代替 GLKit 的 GLKTextureLoader 是例子 OpenGL ES Ch3_2 中的 OpenGL ES Ch3_2ViewController 与例子 OpenGL ES Ch3_1 中的对应的视图控制器之间的唯一不同。在例子 OpenGL ES Ch3_2 中的 AGLKTextureLoader 类的接口和实现提供了 GLKit 提供的功能的一个子集。具体来说，GLKit 的 GLKTextureLoader

类支持异步纹理加载，MIP 贴图生成，以及比简单的 2D 平面更加吸引人的纹理缓存类型。AGLKTextureLoader 只复制了在例子 OpenGL_ES_Ch3_1 中用到的 GLKTextureLoader 的功能。

例子 OpenGL_ES_Ch3_2 中的 AGLKTextureLoader.h 文件声明了两个类：AGLKTextureInfo 和 AGLKTextureLoader。

```
//  
//  AGLKTextureLoader.h  
//  OpenGL_ES_Ch3_2  
  
  
#import <GLKit/GLKit.h>  
  
#pragma mark -AGLKTextureInfo  
  
@interface AGLKTextureInfo : NSObject  
{  
    @private  
        GLuint name;  
        GLenum target;  
        GLuint width;  
        GLuint height;  
}  
  
@property (readonly) GLuint name;  
@property (readonly) GLenum target;  
@property (readonly) GLuint width;  
@property (readonly) GLuint height;  
  
@end  
  
  
#pragma mark -AGLKTextureLoader  
@interface AGLKTextureLoader : NSObject  
  
+ (AGLKTextureInfo *)textureWithCGImage:(CGImageRef)cgImage  
options:(NSDictionary *)options  
error:(NSError **)outError;  
  
@end
```

AGLKTextureInfo 是一个封装了纹理缓存的有用信息的简单类，例如相应的 OpenGL ES 纹理缓存的标识符以及纹理的图像尺寸。AGLKTextureLoader 只声明了一个方法——“+textureWithCGImage:options:error:”。

AGLKTextureLoader 的实现展现了 Core Graphics 和 OpenGL ES 的整合，提供了与 GLKit 的 GLKTextureLoader 相似功能。在“+textureWithCGImage:options:error:”方

法中对于 OpenGL ES 函数的调用完成了标准的缓存管理步骤，包括生成、绑定和初始化一个新的纹理缓存，参见下面粗体显示的代码：

```
///////////////////////////////
// This method generates a new OpenGL ES texture buffer and
// initializes the buffer contents using pixel data from the
// specified Core Graphics image, cgImage. This method returns an
// immutable AGLKTextureInfo instance initialized with
// information about the newly generated texture buffer.
// The generated texture buffer has power of 2 dimensions. The
// provided image data is scaled (re-sampled) by Core Graphics as
// necessary to fit within the generated texture buffer.
+ (AGLKTextureInfo *)textureWithCGImage:(CGImageRef)cgImage
    options:(NSDictionary *)options
    error:(NSError **)outError;
{
    // Get the bytes to be used when copying data into new texture
    // buffer
    size_t width;
    size_t height;
    NSData *imageData = AGLKDataWithResizedCGImageBytes(
        cgImage,
        &width,
        &height);
}

// Generation, bind, and copy data into a new texture buffer
GLuint      textureBufferID;

glGenTextures(1, &textureBufferID);           // Step 1
 glBindTexture(GL_TEXTURE_2D, textureBufferID); // Step 2

glTexImage2D(                                // Step 3
    GL_TEXTURE_2D,
    0,
    GL_RGBA,
    width,
    height,
    0,
    GL_RGBA,
    GL_UNSIGNED_BYTE,
    [imageData bytes]);

// Set parameters that control texture sampling for the bound
// texture
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
```

```

// Allocate and initialize the AGLKTextureInfo instance to be
// returned
AGLKTextureInfo *result = [[AGLKTextureInfo alloc]
    initWithName:textureBufferID
    target:GL_TEXTURE_2D
    width:width
    height:height];

return result;
}

```

`glGenTextures()` 和 `glBindTexture()` 函数与用于顶点缓存的命名方式相似的函数的工作方式相同。但是，`glTexImage2D(GLenum target, GLint level, GLint internalFormat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *data)` 函数是 OpenGL ES 标准中最复杂的函数之一。它复制图片像素的颜色数据到绑定的纹理缓存中。`glTexImage2D()` 函数的第一个参数是用于 2D 纹理的 `GL_TEXTURE_2D`。第二个参数用于指定 MIP 贴图的初始细节级别。如果没有使用 MIP 贴图，第二个参数必须是 0。如果开启了 MIP 贴图，使用第二个参数来明确地初始化每个细节级别，但是要小心，因为从全分辨率到只有一纹素的每个级别都必须被指定，否则 GPU 将不会接受这个纹理缓存。

`glTexImage2D()` 的第三个参数是 `internalFormat`，用于指定在纹理缓存内每个纹素需要保存的信息的数量。对于 iOS 设备来说，纹素信息要么是 `GL_RGB`，要么是 `GL_RGBA`。`GL_RGB` 为每个纹素保存红、绿、蓝三种颜色元素。`GL_RGBA` 保存一个额外的用于指定每个纹素透明度的透明度元素。

`glTexImage2D()` 函数的第四个和第五个参数用于指定图像的宽度和高度。高度和宽度需要是 2 的幂。`border` 参数一直是用来确定围绕纹理的纹素的一个边界的大，但是在 OpenGL ES 中它总是被设置为 0。第七个参数 `format` 用于指定初始化缓存所使用的图像数据中的每个像素所要保存的信息，这个参数应该总是与 `internalFormat` 参数相同。其他的 OpenGL 版本可能在 `format` 和 `internalFormat` 参数不一致时自动执行图像数据格式的转换。

倒数第二个参数用于指定缓存中的纹素数据所使用的位编码类型，可以是下面的符号值之一：`GL_UNSIGNED_BYTE`、`GL_UNSIGNED_SHORT_5_6_5`、`GL_UNSIGNED_SHORT_4_4_4_4` 和 `GL_UNSIGNED_SHORT_5_5_5_1`。使用 `GL_UNSIGNED_BYTE` 会提供最佳色彩质量，但是它每个纹素中每个颜色元素的保存需要一字节的存储空间。结果是每次取样一个 RGB 类型的纹素，GPU 都必须最少读取 3 字节（24 位），每个 RGBA 类型的纹素需要读取 4 字节（32 位）。其他的纹素格式使用多种编码方式来把每个纹素的所有颜色元素的信息保存在 2 字节（16 位）中。`GL_UNSIGNED_SHORT_5_6_5` 格式把 5 位用于红色，6 位用于绿色，5 位用于蓝色，但是没有透明度部分。`GL_UNSIGNED_SHORT_4_4_4_4` 格式平均为每个纹素的颜色元素使用 4 位。`GL_UNSIGNED_SHORT_5_5_5_1` 格式为红、绿、蓝各使用 5 位，但透明度只使用 1 位。使

用 GL_UNSIGNED_SHORT_5_5_5_1 格式会让每个纹素要么完全透明，要么完全不透明。

不管为每个颜色元素保存的位数量是多少，颜色元素的强度最终都会被 GPU 缩放到 0.0 到 1.0 的范围内。一个强度为满值的颜色元素（所有那个颜色元素的位都是 1）对应于一个 1.0 的强度。透明度颜色元素强度为 1.0 表示完全不透明，透明度强度为 0.5 表示 50% 透明度，透明度强度为 0.0 则表示完全透明。

glTexImage2D() 的最后一个参数是一个要被复制到绑定的纹理缓存中的图片的像素颜色数据的指针。

“+textureWithCGImage:options:error:” 方法使用 AGLKDataWithResizedCGImageBytes() 函数来获取用于初始化纹理缓存的内容的字节。AGLKDataWithResizedCGImageBytes() 是在 AGLKTextureLoader.m 中实现的，并且包含转换一个 Core Graphics 图像为 OpenGL ES 可用的合适字节的代码。

```
///////////
// This function returns an NSData object that contains bytes
// loaded from the specified Core Graphics image, cgImage. This
// function also returns (by reference) the power of 2 width and
// height to be used when initializing an OpenGL ES texture buffer
// with the bytes in the returned NSData instance. The widthPtr
// and heightPtr arguments must be valid pointers.
static NSData *AGLKDataWithResizedCGImageBytes(
    CGImageRef cgImage,
    size_t *widthPtr,
    size_t *heightPtr)
{
    NSParameterAssert(NULL != cgImage);
    NSParameterAssert(NULL != widthPtr);
    NSParameterAssert(NULL != heightPtr);

    size_t originalWidth = CGImageGetWidth(cgImage);
    size_t originalHeight = CGImageGetHeight(cgImage);

    NSCAssert(0 < originalWidth, @"Invalid image width");
    NSCAssert(0 < originalHeight, @"Invalid image width");

    // Calculate the width and height of the new texture buffer
    // The new texture buffer will have power of 2 dimensions.
    size_t width = AGLKCalculatePowerOf2ForDimension(
        originalWidth);
    size_t height = AGLKCalculatePowerOf2ForDimension(
        originalHeight);

    // Allocate sufficient storage for RGBA pixel color data with
    // the power of 2 sizes specified
    NSMutableData *imageData = [NSMutableData dataWithLength:
        height * width * 4]; // 4 bytes per RGBA pixel
```

```

NSAssert(nil != imageData,
@"Unable to allocate image storage");

// Create a Core Graphics context that draws into the
// allocated bytes
CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
CGContextRef cgContext = CGBitmapContextCreate(
    [imageData mutableBytes], width, height, 8,
    4 * width, colorSpace,
    kCGImageAlphaPremultipliedLast);
CGColorSpaceRelease(colorSpace);

// Flip the Core Graphics Y-axis for future drawing
CGContextTranslateCTM (cgContext, 0, height);
CGContextScaleCTM (cgContext, 1.0, -1.0);

// Draw the loaded image into the Core Graphics context
// resizing as necessary
CGContextDrawImage(cgContext, CGRectMake(0, 0, width, height),
    cgImage);

CGContextRelease(cgContext);

*widthPtr = width;
*heightPtr = height;
return imageData;
}

```

Core Graphics 函数会把指定的 `cgImage` 拖入 `imgaeData` 提供的字节中。Core Graphics 把 `cgImage` 拖入一个适当大小的 Core Graphics 上下文中，这个过程的一个副作用是把图像的尺寸调整为了 2 的幂。图像在被绘制的时候还被翻转了。翻转 Y 轴是必须的，因为 Core Graphics 是以原点在左上角同时 Y 轴向下增大的形式来实现 iOS 中的图片保存的。OpenGL ES 的纹理坐标系会放置原点在左下角，同时 Y 值向上增大。翻转 Y 轴确保了图像字节拥有适用于纹理缓存的正确的方向。

注意 Mac OS X 版 Core Graphics 保存图像的方式跟 OpenGL ES 一样都是以原点在左下角，同时 Y 值向上增大的方式来保存的。在建立 iOS 时，苹果修改了 Core Graphics 的实现，导致了 Mac OS X 上不存在的与 OpenGL ES 的轻微不兼容。

在 `cgImage` 被拖入 `imageData` 提供的字节之后，函数会返回 `imgaeData` 和数据对应的高度和宽度。

AGLKTextureLoader.m 文件内剩下的代码是不言自明的。只有一个细节的实现有点生疏，就是为用于初始化 AGLKTextureInfo 类的一个方法的实现和声明所使用的一个 Objective-C 类别 (category)。下面代码块中粗体标注的就是类别语句。

```
//////////  
// Instances of AGLKTextureInfo are immutable once initialized  
@interface AGLKTextureInfo : AGLKTextureLoader  
  
- (id)initWithName:(GLuint)aName  
    target:(GLenum)aTarget  
    width:(size_t)aWidth  
    height:(size_t)aHeight;  
  
@end
```

Objective-C 类别会向现存的类添加方法，并使跨多个文件实现一个类成为可能。在这个例子中，类别只是为了使“-initWithName:target:width:height:”方法的声明与类的主接口相分离。在 AGLKTextureLoader.m 文件而不是 .h 文件中声明类别表示这个方法应该只用在 AGLKTextureLoader.m 文件中。这个类别的名字可以随意，但是命名它为 AGLKTextureLoader 进一步强调了添加的这个方法是要在 AGLKTextureLoader 类中使用的。

3.4 OpenGLES_Ch3_3 示例

摆弄一下例子 OpenGLES_Ch3_3 呈现的用户界面。它演示了各种效果，包括纹理的取样模式、纹理的循环模式，以及当一个纹理被映射到顶点并修改在视口中的位置时的图像失真。图 3-5 演示了几种效果。

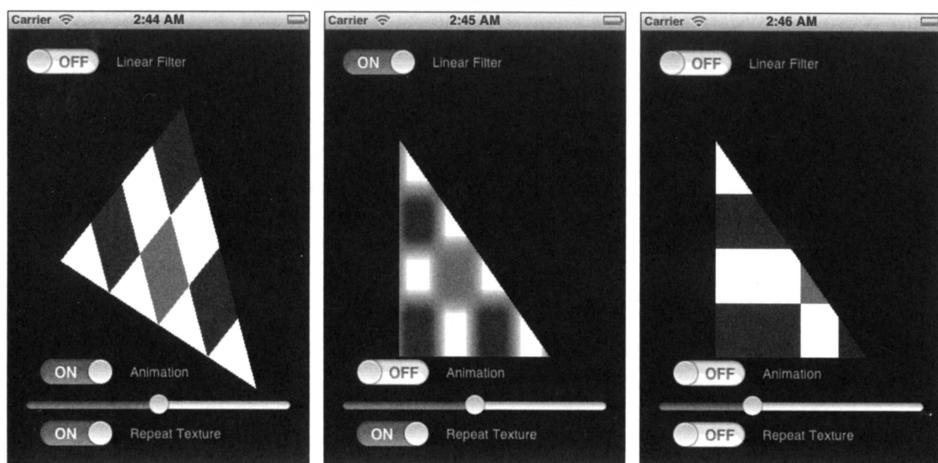


图 3-5 OpenGLES_Ch3_3 示例的截图显示了失真、过滤器和循环模式

图 3-5 中间的截图演示了一个 GL_LINEAR 过滤器放大一个小纹理来填充整个三角形的效果。另外两个截图显示的是 GL_NEAREST 过滤器的效果。

例子 OpenGLES_Ch3_3 向前一个例子创建的 AGLKVertexAttribArrayBuffer 类添加了一个“-reinitWithAttribStride:numberOfVertices:data:”方法。定期地用变化了的顶点位置重新初始化顶点数组缓存的内容以产生一个意在突出纹理失真的简单动画，这个失真是自然发生的几何纹理映射的一部分。

纹理取样模式和纹理循环模式被保存在 OpenGL ES 上下文的每个纹理的标识符中。换句话说，在上下文中的每个纹理缓存都有它自己相应的不依赖于任何其他的纹理缓存的模式。模式是通过 glTexParameter() 函数来设置的，并且通常在一个纹理缓存被初始化以后就不会再更改它的模式了。但是，例子 OpenGLES_Ch3_3 修改了模式以响应用户的输入。这个例子使用下面的类别扩展了 GLKit 的 GLKEffectPropertyTexture 类。

```
@implementation GLKEffectPropertyTexture (AGLKAdditions)

- (void)aglkSetParameter:(GLenum)parameterID
    value:(GLint)value;
{
    glBindTexture(self.target, self.name);

    glTexParameter(
        self.target,
        parameterID,
        value);
}

@end
```

方法“-aglkSetParameter:value:”是使用 aglk 前缀命名的，这样避免了任何未来的方法命名冲突的可能性。如果添加的方法简单地被叫做“-setParameter:value:”，那么如果苹果有一天用相同的名字添加了一个方法到 GLKEffectPropertyTexture，这些方法就会发生冲突。Objective-C 在运行时最终会选择其中的一个方法，但是对于程序员来说可能就很难分辨出运行的到底是哪一个实现了。

3.5 透明度、混合和多重纹理

OpenGL ES 提供了如此多的选项用来保存、映射，以及配置纹理，以至于这个主题常常会压垮程序员。关于纹素颜色格式、纹理环境函数、取样模式，以及细节级别存在一个天文数字的组合和排列。本节甚至会介绍更多的选项，因此为了保持这个主题在可管理范围内，专注于现代最常用的用例是必要的。

可以使用包含透明度元素的 GL_RGBA 纹理格式来指定每个纹素的透明度。图 3-6 显示了一个拥有透明度颜色元素值的纹理，因此棋盘格图案可以透过半透明的纹素显示出来。通常一个或者更多个纹素会结合灯光和顶点颜色来决定每个片元的最终颜色和透

明度。第 4 章会介绍灯光。每个片元产生的透明度都会影响片元怎么与一个帧缓存内的现存内容相混合。

当纹理计算出来一个完全不透明的最终片元颜色时，这个片元颜色会简单地替换任何在帧缓存的像素颜色渲染缓存内现存的对应的像素颜色。还有更有意思的混合方式。如果计算出来的片元颜色部分透明或者全透明，OpenGL ES 会使用一个混合函数来混合片元颜色与像素颜色渲染缓存内对应的像素。

通过调用 glEnable(GL_BLEND) 函数来开启混合。然后通过调用 glBlendFunc(sourceFactor, destinationFactor) 来设置混合函数。sourceFactor 参数用于指定每个片元的最终颜色元素是怎么影响混合的。destinationFactor 参数用于指定在目标帧缓存中已经存在的颜色元素会怎么影响混合。最常用的混合函数配置是设置 sourceFactor 为 GL_SRC_ALPHA，设置 destinationFactor 为 GL_ONE_MINUS_SRC_ALPHA，如下代码所示：

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

GL_SRC_ALPHA 用于让源片元的透明度元素挨个与其他的片元颜色元素相乘。GL_ONE_MINUS_SRC_ALPHA 用于让源片元的透明度元素（1.0）与在帧缓存内的正被更新的像素的颜色元素相乘。结果是，如果片元的透明度值为 0，那么没有片元的颜色会出现在帧缓存中。如果片元的透明度值为 1，那么片元的颜色会完全替代在帧缓存中的对应的像素颜色。介于 0.0 到 1.0 之间的透明度值意味着片元颜色的一部分会被添加到帧缓存内对应的像素颜色的一部分中来产生一个混合的结果。当使用 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) 时，在帧缓存中的最终颜色是用下面的方程式计算的：

$$\begin{aligned} \text{Red}_{\text{final}} &= \text{Alpha}_{\text{fragment}} * \text{Red}_{\text{fragment}} + (1.0 - \text{Alpha}_{\text{fragment}}) * \text{Red}_{\text{Frame Buffer}} \\ \text{Green}_{\text{final}} &= \text{Alpha}_{\text{fragment}} * \text{Green}_{\text{fragment}} + (1.0 - \text{Alpha}_{\text{fragment}}) * \text{Green}_{\text{Frame Buffer}} \\ \text{Blue}_{\text{final}} &= \text{Alpha}_{\text{fragment}} * \text{Blue}_{\text{fragment}} + (1.0 - \text{Alpha}_{\text{fragment}}) * \text{Blue}_{\text{Frame Buffer}} \\ \text{Alpha}_{\text{final}} &= \text{Alpha}_{\text{fragment}} + (1.0 - \text{Alpha}_{\text{fragment}}) * \text{Alpha}_{\text{Frame Buffer}} \end{aligned}$$

3.5.1 在 OpenGL ES Ch3_4 示例中混合片元颜色

使用 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) 会与 iOS Core Graphics 的“正常混合模式”产生相同的结果。图 3-7 是例子 OpenGL ES Ch3_4 产生的显示结果。图 3-7 中的树叶纹理与在帧缓存的像素颜色渲染缓存中的黑色像素混合，

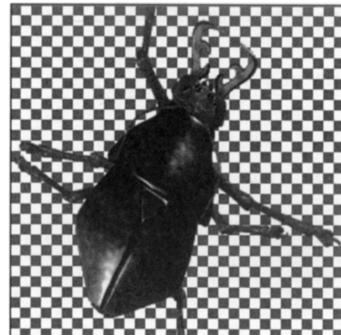


图 3-6 含有透明度颜色元素的纹理
会在半透明纹素的位置显示图案

因此在树叶纹理包含透明纹素的每个地方黑色像素都保持不变。然后使用一个虫子图片的第二个纹理会与像素颜色渲染缓存相混合。每当虫子纹理的纹素是透明的时候，在帧缓存内的已存的颜色就保持不变。最终的渲染结果为虫子纹理在树叶纹理之上，树叶纹理在黑色背景之上。

注意 在图 3-7 中显示的层效果揭示了 iOS Core Animation 技术的基本原理。每个 Core Animation 层使用一个对应的 OpenGL ES 的像素颜色渲染缓存来保存像素颜色数据。每个层的像素颜色数据作为一个 OpenGL ES 纹理缓存，并且纹理缓存会使用 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) 函数来与后帧缓存混合在一起。

例子 OpenGLES_Ch3_4 在例子 OpenGLES_Ch3_3 中的“-viewDidLoad”方法的实现中添加了下面的粗体代码。该代码会加载第二个纹理并开启与像素颜色渲染缓存的混合。

```
// Setup texture0
CGImageRef imageRef0 =
    [[UIImage imageNamed:@"leaves.gif"] CGImage];

self.textureInfo0 = [GLKTextureLoader
    textureWithCGImage:imageRef0
    options:[NSDictionary dictionaryWithObjectsAndKeys:
        [NSNumber numberWithBool:YES],
        GLKTextureLoaderOriginBottomLeft, nil]
    error:NULL];

// Setup texture1
CGImageRef imageRef1 =
    [[UIImage imageNamed:@"beetle.png"] CGImage];

self.textureInfo1 = [GLKTextureLoader
    textureWithCGImage:imageRef1
    options:[NSDictionary dictionaryWithObjectsAndKeys:
        [NSNumber numberWithBool:YES],
        GLKTextureLoaderOriginBottomLeft, nil]
    error:NULL];

// Enable fragment blending with Frame Buffer contents
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

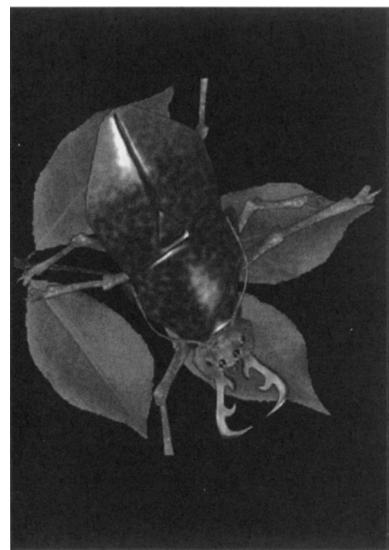


图 3-7 一个虫子纹理在树叶纹理之上，树叶纹理在黑色背景之上

相比上一个例子的另一个小变化是从图像加载纹理时应用了一个 NSDictionary 对象来设定选项。在这个例子中，GLKTextureLoaderOriginBottomLeft 键与布尔值 YES 搭配是为了命令 GLKit 的 GLKTextureLoader 类垂直翻转图像数据。这个翻转可以抵消图像的原点与 OpenGL ES 标准原点之间的差异。

```
[NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithBool:YES],
    GLKTextureLoaderOriginBottomLeft, nil]
```

例子 OpenGLES_Ch3_4 把相同的几何图形渲染了两次：第一次使用了一个纹理，第二次使用了另一个。混合发生在每次被一个纹理着色的一个片元与在像素颜色渲染缓存中已存的像素颜色混合的时候。下面来自 OpenGLES_Ch3_4 的 “-glkView:drawInRect:” 实现的摘录显示了相关的代码：

```
self.baseEffect.texture2d0.name = self.textureInfo0.name;
self.baseEffect.texture2d0.target = self.textureInfo0.target;
[self.baseEffect prepareToDraw];

// Draw triangles using the first three vertices in the
// currently bound vertex buffer
[self.vertexBuffer drawArrayWithMode:GL_TRIANGLES
    startVertexIndex:0
    numberOfVertices:6];

self.baseEffect.texture2d0.name = self.textureInfo1.name;
self.baseEffect.texture2d0.target = self.textureInfo1.target;
[self.baseEffect prepareToDraw];

// Draw triangles using the first three vertices in the
// currently bound vertex buffer
[self.vertexBuffer drawArrayWithMode:GL_TRIANGLES
    startVertexIndex:0
    numberOfVertices:6];
```

GLKit 的 baseEffect 是由第一个纹理设定的，同时 vertexBuffer 被绘制。然后 baseEffect 由第二个纹理设定，同时 vertexBuffer 被再次绘制。这两个过程中也伴随着与像素颜色渲染缓存的混合。绘图的顺序决定了哪一个纹理会出现在另一个之上。在当前情况下是虫子在树叶的上面。纹理绘制的顺序颠倒过来的话会把树叶置于虫子之上。

3.5.2 示例 OpenGLES_Ch3_5 中的多重纹理

很多有用的可视效果可以通过把片元颜色与在像素颜色渲染缓存中现存的颜色相混合来实现，但是这个技术有两个主要的缺点：每次显示更新时几何图形必须要被渲染一到更多次，混合函数需要从像素颜色渲染缓存读取颜色数据以便与片元颜色混合。然后

结果被写回帧缓存。当带有透明度数据的多个纹理如图 3-7 所示层叠时，每个纹理的像素颜色渲染缓存的颜色会被再次读取、混合、重写。如例子 OpenGLES_Ch3_4 所示通过多次读写像素颜色渲染缓存来创建一个最终的渲染像素的过程叫做多通道渲染。如往常一样，内存访问限制了性能，因此多通道渲染是次优的。接下来将介绍的多重纹理方法避免了多通道渲染的大部分缺陷。

所有的现代 GPU 都能够同时从至少两个纹理缓存中取样像素。GLKit 的 GLKBaseEffect 类同时支持两种纹理。执行像素取样和混合的硬件组件叫做一个纹理单元或者一个取样器。如果你的应用需要超过两个纹理单元，在确定一个单独的通道中可以结合多少个纹理之前，请使用下面的代码：

```
GLint iUnits;
glGetIntegerv(GL_MAX_TEXTURE_UNITS, &iUnits);
```

例子 OpenGLES_Ch3_5 产生的输出与 OpenGLES_Ch3_4 相同，除了树叶和虫子纹理被混合在了一个通道中，而且与像素颜色渲染缓存的内容混合来产生结果片元颜色的过程只会在每次显示更新中发生一次。

多重纹理引入了另一个组合组配置选项。为了帮助降低复杂性，iOS 5 中的 GLKit 的 GLKEffectPropertyTexture 类操作了 3 种常见的多重纹理模式：GLKTextureEnvModeReplace、GLKTextureEnvMode Modulate，以及 GLKTextureEnvModeDecal。GLKEffectPropertyTexture 默认使用 GLKTextureEnvModeModulate 模式，这种模式几乎总是产生最好的结果。GLKTextureEnvModeModulate 模式会让所有的为灯光和其他效果计算出来的颜色与从一个纹理取样的颜色相混合。

GLKEffectPropertyTexture 的 envMode 属性用于配置混合模式。OpenGLES_Ch3_5 与 OpenGLES_Ch3_4 加载相同的两个纹理，但是不再需要明确地启动与帧缓存的像素颜色渲染缓存的混合。相反，baseEffect 的第二个纹理属性 texture2D1 被设置为使用 GLKTextureEnvModeDecal 模式，这种模式会使用一个与 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) 类似的方程式来混合第二个与第一个纹理。

```
:self.baseEffect.texture2D1.envMode = GLKTextureEnvModeDecal;
```

需要为所有的纹理单元的使用提供纹理坐标。多重纹理支持为每个纹理使用不同的纹理坐标。在本书例子中的 SceneVertex 数据类型可以被扩展来保存每个顶点的纹理坐标的多重集合，但是 OpenGLES_Ch3_5 配置 OpenGL ES 2.0 为两个纹理使用相同的纹理坐标。下面的来自例子 OpenGLES_Ch3_5 的“-glkView:drawInRect:”方法的完整实现显示了具体步骤。粗体的代码突出了每个纹理单元的纹理坐标的设定：

```
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect
{
    // Clear back frame buffer (erase previous drawing)
```

```

[(AGLKContext *)view.context clear:GL_COLOR_BUFFER_BIT];

[self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribPosition
    numberOfRowsInSection:3
    attribOffset:offsetof(SceneVertex, positionCoords)
    shouldEnable:YES];

[self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribTexCoord0
    numberOfRowsInSection:2
    attribOffset:offsetof(SceneVertex, textureCoords)
    shouldEnable:YES];

[self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribTexCoord1
    numberOfRowsInSection:2
    attribOffset:offsetof(SceneVertex, textureCoords)
    shouldEnable:YES];

[self.baseEffect prepareToDraw];

// Draw triangles using the first three vertices in the
// currently bound vertex buffer
[self.vertexBuffer drawArrayWithMode:GL_TRIANGLES
    startVertexIndex:0
    numberOfRowsInSection:6];
}

```

3.5.3 在 OpenGL_ES_Ch3_6 示例中自定义纹理

多重纹理的强大和灵活性在使用自定义的利用 OpenGL ES Shading Language 的 OpenGL ES 2.0 片元程序时会变得更加明显。一个额外的例子，在 OpenGL_ES_Ch3_6 中，首先用一个由 GLKit 的 GLKBaseEffect 在后台自动生成的 Shading Language 程序绘制一个立方体，然后使用下面的自定义顶点和片元的 Shading Language 程序绘制第二个立方体。现在不用担心没有学过 Shading Language，但是如果你很好奇，可以摆弄一下 OpenGL_ES_Ch3_6，看看它是怎么工作的。

```

//
// Shader.vsh
// TestShaders
//
//

attribute vec4 aPosition;
attribute vec3 aNormal;
attribute vec2 aTextureCoord0;
attribute vec2 aTextureCoord1;

varying lowp vec4 vColor;

```

```

varying lowp vec2 vTextureCoord0;
varying lowp vec2 vTextureCoord1;

uniform mat4 uModelViewProjectionMatrix;
uniform mat3 uNormalMatrix;

void main()
{
    vec3 eyeNormal = normalize(uNormalMatrix * aNormal);
    vec3 lightPosition = vec3(0.0, 0.0, 1.0);
    vec4 diffuseColor = vec4(0.7, 0.7, 0.7, 1.0);

    float nDotVP = max(0.0, dot(eyeNormal, normalize(lightPosition)));

    vColor = vec4((diffuseColor * nDotVP).xyz, diffuseColor.a);
    vTextureCoord0 = aTextureCoord0.st;
    vTextureCoord1 = aTextureCoord1.st;

    gl_Position = uModelViewProjectionMatrix * aPosition;
}

```

一个片元着色器是一个由 GPU 执行的，用来完成计算当前渲染缓存中的每个片元的最终颜色所需要的运算的简短程序。包含片元着色器程序的文件通常使用“.fsh”文件扩展名。

```

//
// Shader.fsh
// TestShaders
//

uniform sampler2D uSampler0;
uniform sampler2D uSampler1;
varying lowp vec4 vColor;
varying lowp vec2 vTextureCoord0;
varying lowp vec2 vTextureCoord1;

void main()
{
    // first texture Modulate, second texture Decal
    lowp vec4 color0 = texture2D(uSampler0, vTextureCoord0);
    lowp vec4 color1 = texture2D(uSampler1, vTextureCoord1);
    gl_FragColor = mix(color0, color1, color1.a) * vColor;
}

```

相比其他的纹理混合配置方法，GL Shading Language 程序往往是既简短又更加自文档化的。但是，OpenGL ES Shading Language 是一个复杂到值得用整本书来论述的主题，例如 OpenGL Shading Language（第3版），由 Randi J. Rost、Bill Licea-

Kane、Dan Ginsburg、John M. Kessenich、Barthold Lichtenbelt、Hugh Malan 以及 Mike Weiblen 所著。本书会逐步地讲解一些概念同时在几个章节中做例子演示，但是本书不是一本完整的参考书。

3.6 纹理压缩

苹果建议使用 GPU 所支持的纹理压缩方式。纹理压缩最小化了当要保存相对高细节级别的纹理时保存纹理缓存所需要的内存的数量。截止到 iOS 5，所有的 iOS 设备都支持 PowerVR 纹理压缩（PVRTC）格式。但是，这无法保证将来的苹果硬件还会支持 PVRTC 格式。因此，在使用它之前，程序有必要检查非标准 OpenGL ES 的 GL_IMG_texture_compression_pvrtc 扩展的存在性。苹果建议使用下面的 C 函数来检查这个扩展：

```
BOOL CheckForExtension(NSString *searchName)
{
    // For best results, extensionsNames should be stored so that it
    // does not need to be recreated on each invocation.
    NSString *extensionsString = [NSString
        stringWithCString:glGetString(GL_EXTENSIONS) encoding:
        NSASCIIStringEncoding];
    NSArray *extensionsNames = [extensionsString
        componentsSeparatedByString:@" "];

    return [extensionsNames containsObject:searchName];
}
```

一个程序可以使用下面形式的代码来检查是否支持纹理压缩。

```
if( CheckForExtension(@"GL_IMG_texture_compression_pvrtc") )
{
    // The extension is available.
}
```

作为 iOS SDK 的一部分，苹果提供了命令行实用工具 texturetool。texturetool 工具会将 “.png” 图像转换为能够直接加载到压缩的纹理缓存的格式，这个网址：http://developer.apple.com/iphone/library/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TextureTool/ TextureTool.html 做了介绍。GLKit 的 GLKTextureLoader 类也支持压缩的纹理。

用来初始化压缩纹理的图像的尺寸需要是 2 的幂，同时它们还需要是正方形。一个 256×256 像素的图像可以用在一个压缩纹理缓存中，但是一个 64×128 像素的图像却不能。

苹果为使用压缩纹理提供了示例代码，地址为：<http://developer.apple.com/iphone/>

library/samplecode/PVRTextureLoader/。

3.7 小结

本章几乎提供了实现一个 2D 硬件加速的基于 sprite 的游戏所需的全部信息。一个 sprite 仅仅是一个可以移动的包含透明像素的图像，例子 OpenGL_ES_Ch3_3 甚至演示了在显示器上来回移动纹理的一个方法，同时间接演示了由纹理取样产生的失真效果。苹果在 <http://developer.apple.com/iphone/library/samplecode/GLImageProcessing> 网址提供的示例代码中显示了如何用多重纹理来实现强大的图像处理效果，例如对图像进行亮度、对比度、饱和度、色度和锐度调整。这些例子提示了商业图像处理应用之下的苹果硬件加速 Core Image 技术的实现。

截止到现在，所有的例子程序都是使用 3D 技术绘制简单的 2D 三角形。第 2 章介绍了在一个 iOS 设备上显示任何基于 OpenGL ES 的图像所需的所有步骤。第 3 章扩展了第 2 章的例子实现在三角形上显示多重纹理。第 4 章会在整个三维上深入探讨渲染。灯光是在一个二维屏幕上显示出三维假象的关键。

