Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

移动开发经典丛书

Game and Graphics Programming for iOS and Android with OpenGL ES 2.0

OpenGL ES 2.0 游戏与图形编程

——适用于iOS和Android

[美] Romain Marucchi-Foino 著
王净 译

清华大学出版社

移动开发经典丛书

OpenGL ES 2.0 游戏与图形编程

——适用于 iOS 和 Android

[美] Romain Marucchi-Foino 著

王 净 译

清华大学出版社

北 京

Romain Marucchi-Foino
Game and Graphics Programming for iOS and Android with OpenGL ES 2.0
EISBN：978-1-119-97591-5

北京市版权局著作权合同登记号 图字：01-2013-7449

图书在版编目(CIP)数据

OpenGL ES 2.0游戏与图形编程——适用于iOS和Android / (美) 马鲁基-弗伊诺(Marucchi-Foino, R.) 著; 王净 译. —北京：清华大学出版社，2014
(移动开发经典丛书)
书名原文：Game and Graphics Programming for iOS and Android with OpenGL ES 2.0
ISBN 978-7-302-35230-3

Ⅰ. ①O… Ⅱ. ①马… ②王… Ⅲ. ①移动终端－游戏程序－程序设计 Ⅳ. ①TN929.53 ②TP311.5

中国版本图书馆 CIP 数据核字(2014)第 014856 号

责任编辑：王　军　韩宏志
装帧设计：牛静敏
责任校对：曹　阳
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　印　　张：18　　字　　数：438 千字
版　　次：2014 年 2 月第 1 版　　印　　次：2014 年 2 月第 1 次印刷
印　　数：1～3000
定　　价：48.00 元

产品编号：

译 者 序

随着智能手机移动嵌入式平台硬件性能的不断提升，3D 游戏应用逐渐普及开来。目前，Android 和 iOS 平台占据了大部分市场份额。如何有效地在这两种平台上开发高质量的游戏已成为很多游戏开发人员追求的目标。OpenGL ES (OpenGL for Embedded Systems) 是 OpenGL 三维图形 API 的子集，专门针对手机、PDA 和游戏主机等嵌入式设备而设计，目前已有越来越多的开发人员选用 OpenGL ES 进行游戏和图形编程。但目前国内关于 OpenGL ES 开发的资料非常有限。相信本书的出版可为 OpenGL 的开发起到推进作用。

本书是一本系统的具备实战性的 OpenGL ES 图形开发指南。由资深的游戏引擎开发专家针对 OpenGL ES 最新版本撰写，不仅系统讲解了 OpenGL ES 的核心概念和技术，讨论 iOS 和 Android 的图形机制，还通过大量案例讲解在 iOS 和 Android 上进行 OpenGL ES 开发的方法和技巧。

全书共分 12 章，涵盖了创建完整游戏所需的所有知识。你将学习创建游戏所涉及的方方面面，比如加载 3D 几何体和纹理，如何处理材质、着色器、声音、相机、剪切、物理、AI、路径发现、骨骼动画等。本书所有章节都彼此相关。每章都以一个实例为基础逐步向读者演示如何掌握所需的技术。如果读者掌握了本书中的所有内容，就可以很快地开始编写实际的游戏和图形程序。

本书的讲解由浅入深，从 Android 和 iOS 平台上游戏和图形开发应用的基础知识到开发大型游戏程序，结构清晰、语言简洁，非常适合进阶开发者阅读参考。

本书第 1 章和第 3 章由范园芳、胡训强负责翻译，第 2 章和第 6 章由彭洁、刘晓昌负责翻译，第 4 章和第 9 章由姜茗瀚负责翻译，第 10 章由晏峰负责翻译，第 5 章、第 7 和第 8 章、第 11 和第 12 章由王净负责翻译。最后，全书由王净进行统稿。在此，我要对各位译者深表谢意。此外，还要感谢我的家人，她们总是无怨无悔地支持我的一切工作，生活在这样美满的家庭，我备感幸福。

译者在翻译过程中，尽量保持原书的特色，并对书中出现的术语和难词难句进行了仔细推敲和研究。但毕竟有少量技术是译者在自己的研究领域中不曾遇到过的，所以疏漏和争议之处在所难免，望广大读者提出宝贵意见。

最后，希望广大读者能多花些时间细细品味这本凝聚作者和译者大量心血的书籍，为将来的职业生涯奠定良好基础。

王 净

作 者 简 介

Romain Marucchi-Foino 是当今流行的移动游戏引擎 SIO2(http://sio2interactive.com)的原作者和创建者。Romain 以前是一名游戏引擎开发人员，自从 iPhone 诞生以来，Romain 就一直致力于使用 OpenGL ES 为移动设备创建先进的游戏引擎。目前，他是 sio2interactive.com 的一名高级 3D 程序员(SIO2 引擎的正式开发人员)，该引擎通过 App Store 和 Android Market 驱动了数以千计的游戏和 3D 应用程序。凭借在移动游戏行业积累的丰富经验，他为许多网上社区、出版物以及博客作出重大贡献。

技术编辑简介

Effie C. Lee 在过去 4 年里一直从事游戏行业，并且是一名自营职业的游戏和图形设计师。在获得了计算机科学专业的学士学位后，他开始对电子游戏和计算机图形学产生了浓厚的兴趣，并参与了多个移动游戏产品的开发。凭借在游戏开发方面的广博知识，他逐渐成为一名专业的 2D 和 3D 图形设计师、游戏程序员(编写脚本)以及游戏网站的 Web 设计师，并且管理游戏质量和本地化。如果你想与他取得联系，可以发邮件到 effiecl@gmail.com。

前　言

欢迎使用本书。本书并不是一本常见的“OpenGL Hello Triangle”书籍——也就是说并不会详细解释“为什么”(你可以从 Google 上找到相关的原因)，而重点介绍“如何做”。将会详细地介绍在编写游戏和图形程序时应该做什么，不应该做什么。

本书包括了 50 多个不同的教程(其中还包括了一些完整的游戏框架)，并且采用了一种简单实用的新颖方法重点介绍需要学习的内容，从而确保完成教程后可以创建一款游戏。

我们将学习使用丰富的 3D 图像创建完整游戏所需的所有元素。如果正在寻找一种好的教学方法来帮助你快速创建梦想中的游戏，那么本书正是你所需要的。

本书读者对象

首先，本书并不是一本入门级图书，而是一本中等层次的书籍，本书假设你已经熟悉线性代数(矩阵、向量和四元数)，具有深厚的 C/C++编程背景，至少接触过基础的 OpenGL 或者 OpenGL ES，以及知道计算机图形的基本工作方式。

如果已经掌握了这些必需的知识，并且想在游戏和图形编程方面取得闪电般的进步，那么本书正是你所需要的。本书主要是为那些想要学习核心知识以便创建功能完备的游戏并在 App Store 和 Android Market 上进行售卖的人所撰写的。

本书主要内容

从根本上讲，本书包括了创建一个完整的游戏所需的所有知识。你将学习创建游戏所涉及的各个方面，如加载 3D 几何体和纹理，如何处理材质、着色器、声音、摄像头、剪切、物理、AI、路径发现和骨骼动画等。

在学完本书后，你将能使用所学的知识并组合所完成的不同教程以创建自己的高级游戏。

本书组织结构

本书的组织结构是所有的章节都彼此关联。每一章都会逐步向读者演示如何掌握所需的技术，以便能够处理和理解下一章的内容。

接下来是所有章的列表，以及每一章所介绍的内容：

- 第 1 章“入门”——我们将学习如何构建开发环境，下载本书的 SDK，导入并重新编译教程以及如何处理全书将要使用的模板项目。
- 第 2 章“设置图形投影”——由于已经拥有了一个正在运行的模板，因此我们将学习如何构建所需的投影矩阵，以便能够处理 2D、2.5D 或 3D。此外还将学习如何在屏幕上绘制简单几何图形以及如何处理摄像头矩阵。
- 第 3 章“处理复杂几何图形”——我们将通过创建一个 Wavefront OBJ 查看器来学习如何从磁盘加载复杂几何图形。还将学习如何加载和创建纹理，如何处理基本光照以及响应触摸事件。
- 第 4 章“构建场景”——该章将对第 3 章学到的知识进行扩展，将解释如何处理更复杂的场景。我们将学习绘图序列以及如何创建可重用的着色器。
- 第 5 章“优化”——该章将介绍可用来优化绘图性能的相关技术。我们将学习纹理压缩和着色器优化的基础知识，学习如何将三角形转换为三角形带以及其他用来获取更佳 FPS 的提示和技巧。
- 第 6 章“实时物理”——由于在该章之前我们已经学习了如何正确地处理场景，因此该章将介绍如何使用 Bullet 向场景添加实时物理行为。我们将首先学习如何创建物理世界和物理实体。然后学习如何在代码中使用不同的技术，以便在碰撞回调中或者根据两个或更多个物理实体之间的接触点添加相关逻辑。
- 第 7 章“摄像头”——该章将重点介绍摄像头。我们将首先学习构建视锥平面，并能从摄像头的角度确定场景中每个对象的可见性。然后学习如何实现多种不同类型的摄像头，包括带有碰撞的完整第一人称和第三人称摄像头(你可以在自己的应用程序中使用这些摄像头)。
- 第 8 章“路径发现”——人工智能(AI)和路径发现功能在游戏中通常扮演重要角色，而这恰好是本章将要介绍的内容。我们将学习如何使用 Recast 和 Detour 库构建导航网格以及让实体在场景中自动移动。该章还将演示如何使用 True Type Font 生成字体纹理以及如何在屏幕上绘制动态文本。
- 第 9 章“音频及其他极佳的游戏编程资料”——该章主要介绍如何使用 OpenAL 来播放音频。我们将学习如何加载 OGG Vorbis 声音文件以及如何实时地从内存中以流的方式读出这些文件或者静态地将它们存储到音频存储器中。此外，还将学习如何创建 3D 定位和环境声源，如何使用加速计以及如何实现纹理动画和创建其他多种效果。
- 第 10 章“高级光照”——该章将介绍如何应用动态光照，这可能是游戏和图形编程中最难掌握的内容。在该章，我们将创建多种不同类型的灯(从定向灯到聚光灯)，并学习如何实时处理这些灯。
- 第 11 章“高级 FX”——该章主要介绍一些特殊效果。我们将学习如何创建全屏的后处理效果、投影纹理和实时阴影，以及如何处理粒子。

- 第 12 章“骨骼动画”——最后的内容并不一定是最不重要的内容。在该章，我们将学习如何使用 MD5 文件格式处理骨骼动画。该章将首先介绍如何加载和绘制附加到骨骼的网格。然后介绍如何加载动作文件以及如何使用不同类型的混合方法来混合这些动作。

你将会发现，本书并不只是介绍相关理论，还会演示如何将每章所学到的知识应用到实际游戏方案中。

如你所见，本书介绍了在日常游戏或 3D 应用程序中需要使用的有用知识。如果掌握了本书中的所有内容，就可以很快地开始编写实际的游戏和图形程序。

使用本书所需条件

如果打算为 iOS 系统开发应用程序，那么需要一台可以支持 iOS SDK 最新版本的 Mac(如果想了解更多的信息，请访问 http://developer.apple.com)。由于 iOS SDK 提供了 iPhone/iPod Touch 和 iPad 模拟器(可用开发和测试应用程序)，因此 iDevice 是可选的。该模拟器与本书所包含的内容是完全兼容的。

如果打算为 Android 开发应用程序，那么需要一台带有 Android SDK 所支持的操作系统的 Mac 或 PC(更多信息请访问 http://developer.android.com)。同时，还需要一台支持 OpenGL ES 2.0 的 Android 设备，因为 Android SDK 绑定的模拟器仅支持 OpenGL ES 1.0。

此外，本书使用 Blender 作为 3D 建模软件(因为该软件是免费和开源的)。所以，为了测试、调试和重新输出本书 SDK 中使用的所有测试场景，请从 http://blender.org 获取该软件的一份副本。

SDK 源代码

可从 www.wrox.com 下载本书中所使用的官方 SDK(被压缩为一个.zip 文件)。该 SDK 包含了本书中所介绍的所有教程的最终结果。同时，还包含了 SDK 的全部源代码以及教程中使用的所有原始资产，因此你可以访问书中的 2D/3D 场景，并且可以重新编译它们。也可以访问 http://www.tupwk.com.cn/downpage，输入本书中文书名或 ISBN，下载本书源代码。

注意：因为许多书都拥有相类似的标题，所以使用 ISBN 可以更容易地找到本书。本书英文版的 ISBN 为 978-1-119-97591-5。

此外，我个人还在 GFX 3D Engine(一款免费且开源的迷你 3D 游戏和图形引擎，我们

将在本书中使用该引擎，如图 0-1 所示)的官方网站上提供了本书的 SDK(使用了 GIT 版本的控制系统)，可从 http://gfx.sio2interactive.com 下载该引擎。

图 0-1　GFX 3D Engine

读者可以随时从 GFX 3D Engine 网站找到本书 SDK 的最新版本(针对上一个版本的 bug 进行了修复)，因为对于我来说，在该网站上可以更加容易地使用版本控制来更新源代码。而对于 www.wrox.com 上的官方 SDK 来说，更新速度则相对较慢，因为官方 SDK 版本是由发布者提供的，如果愿意，可以耐心地等待官方 SDK 的发布。

此外值得一提的是，在 GFX 3D Engine 网站上(http://gfx.sio2interactive.com)，你可以找到针对本书 SDK 的支持论坛以及 GFX 3D Engine 的最新版本。该网站还提供了其他的与 3D 游戏和图形相关联的演示、教程和其他材料，并且都与本书的 SDK 完全兼容。

勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果你在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

要在网站上找到本书的勘误表，可以登录 http://www.wrox.com，通过 Search 工具或书名列表查找本书，然后在本书的细目页面上，单击 Book Errata 链接。在这个页面上可以查看 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 www.wrox.com/misc-pages/booklist.shtml。

如果在 Book Errata 页面上没有看到你找出的错误，请进入 www.worx.com/contact/techsupport.shtml，填写表单，发电子邮件，我们就会检查你的信息，如果是正确的，就在本书的勘误表中粘贴一个消息，我们将在本书的后续版本中采用。

另外，如果有任何意见和建议，也可以给 wkservice@vip.163.com 发电子邮件。

p2p.wrox.com

P2P 邮件列表是为作者和读者之间的讨论而建立的。读者可以在 p2p.wrox.com 上加入 P2P 论坛。该论坛是一个基于 Web 的系统，用于传送与 Wrox 图书相关的信息和相关技术，

与其他读者和技术用户交流。该论坛提供了订阅功能，当论坛上有新帖子时，会给你发送你选择的主题。Wrox 作者、编辑和其他业界专家和读者都会在这个论坛上进行讨论。

在 http://p2p.wrox.com 上有许多不同的论坛，帮助读者阅读本书，在读者开发自己的应用程序时，也可以从这个论坛中获益。要加入这个论坛，必须执行下面的步骤：

(1) 进入 p2p.wrox.com，单击 Register 链接。

(2) 阅读其内容，单击 Agree 按钮。

(3) 提供加入论坛所需的信息及愿意提供的可选信息，单击 Submit 按钮。

(4) 然后就可以收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

提示：不加入 P2P 也可以阅读论坛上的信息，但只有加入论坛后，才能发送自己的信息。

加入论坛后，就可以发送新信息，回应其他用户的帖子。可以随时在 Web 上阅读信息。如果希望某个论坛给自己发送新信息，可以在论坛列表中单击该论坛对应的 Subscribe to this Forum 图标。

对于如何使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作原理，以及许多针对 P2P 和 Wrox 图书的常见问题解答。要阅读 FAQ，可以单击任意 P2P 页面上的 FAQ 链接。

目　录

第 1 章 入 门

本章内容

- 了解本书使用的软件
- 下载本书的 SDK
- 理解 SDK 的体系结构
- 将相关项目导入到 IDE 中
- 理解本书介绍的模板应用程序
- 学习如何使用模板代码结构

本章将首先创建本书教程和示例使用的开发环境。

然后，快速介绍本书的 SDK 及其下载地址，以及 SDK 中包含的不同目录。随后学习如何将本书中现有的 SDK 项目和模板导入到你所喜欢的 IDE 中，在后面学习本书的不同教程时将需要这么做。

在本章的最后一节，我们将学习跨平台的模板项目。最后，本章提供了一个快速教程，以帮助读者熟悉模板事件以及本书所有教程所使用的语言特点。

1.1 软件需求

本书所有内容都是基于 iOS 5.x+以及 Android 2.x+构建的；在撰写本书时，这是两种移动操作系统最新且最稳定的版本。

1.1.1 针对 iOS 开发人员

对于 iOS 操作系统来说，如果想使用本书，必须从 http://developer.apple.com 中获取最

新 iOS SDK 的副本，并在 Mac 上进行安装。

iOS SDK 提供了一个完全支持 GLES v2 的模拟器，所以即使没有 iOS 设备，或者没有从 Apple 处获取官方的 iOS 开发人员认证，也仍然可以使用本书。

1.1.2 针对 Android 开发人员

对于 Android 操作系统来说，构建环境并不像在 iOS 操作环境中那么简单。首先需要访问 http://developer.android.com/sdk/installing.html，并按照相关指示安装 Android SDK、Eclipse 和 ADT 插件。请注意，本书所使用的 Android SDK 版本是 v2.3.4，当然，也可以使用最新版本。

本书所有代码都使用了 C/C++，这意味着必须安装 Android 本机代码支持。为了完成开发环境的安装，请按照下面的步骤进行操作：

(1) 从 http://developer.android.com/sdk/ndk/index.html 处获取 Android SDK 的一份副本。虽然在撰写本书时使用的版本是 r5c，但本书的所有示例和教程在最新版本中也可以很好地运行。下载 Android NDK 压缩包，并在需要进行读写访问的计算机上解压缩。

(2) 为使用 Eclipse 编译和调试本机代码，需要安装 Sequoyah 插件。为此，首先从 Eclipse 主菜单中依次选择 Help | Install New Software | Available Software Sites | Sequoyah Metadata Repository，以便启动存储库。然后从 Work With 组合框中选择该条目，一旦加载完存储库数据，就可以选择并安装 Sequoyah Android Native Code Support，如图 1-1 所示。

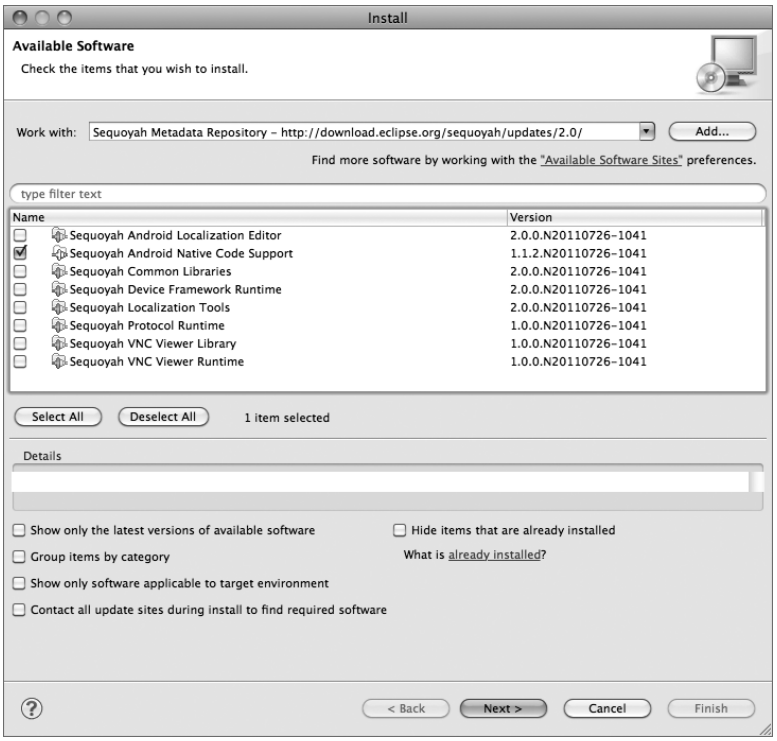

图 1-1 Sequoyah Native Code Support 插件

(3) 安装完 Sequoyah 后，在主菜单中转到 Eclips Preferences | Android | Native Development，指定提取步骤(1)中解压缩的 Android NDK 的位置，如图 1-2 所示。

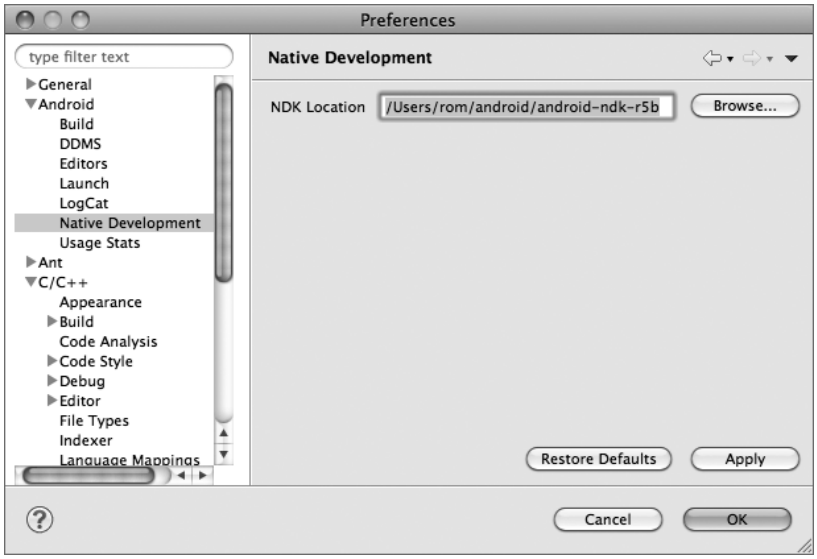

图 1-2　指定 Android NDK 的位置

恭喜你，现在，你的 Android 开发环境已经全部设置完毕！然而请注意，为在 Android 操作系统中使用本书，还需要支持 OpenGL ES 2.0 的实际设备。当前的 Android SDK 所提供的模拟器仅支持 OpenGL ES 1.x，而不支持 OpenGL ES 2.0。所以在 Android 系统上无法执行模拟器上的本地部署；只有使用 GLES 2 时才支持设备部署。

1.2　下载本书的 SDK

一旦构建开发环境，接下来就应该获取本书 SDK 的一份副本。可从 http://www.wrox.com 中下载官方的 SDK。或者如果想通过 GIT 下载 SDK，也可以访问官方的 GFX 3D Engine 网站(http://gfx.sio2interactive.com)，并找到详细的说明。

如果下载了压缩文件，只需在自己拥有读写访问权限的目录中进行解压缩就可以了。而如果是通过 GIT 下载压缩文件，那么在你的设备上就已经可以使用所有的文件和 SDK 体系结构了。

本书 SDK 的体系结构非常简单。如果想了解更多信息，可参阅下面的目录列表。

- _chapter#-#：该目录包含了在完成本书的教程后应该生成的最终结果。在阅读本书的过程中，如果觉得相关的指示不够清楚，或者不确定在什么地方插入代码，又或者想要预先查看某一教程的最终结果，都可以打开该目录。可在根目录中查找教程所使用的源文件(分别被命名为 templateApp.cpp 和 templateApp.h)，也可以访问根目录下的两个子目录，它们分别包含 iOS 和 Android 项目文件。然后将项目加载到 IDE 中并从头开始重建。

- common：该目录包含了创建本书中模板和教程所使用的 GFX 3D Engine 版本的免费和开源代码(在本书中，我们将使用迷你游戏和图形引擎)，以及引擎所依赖的库源。GFX 3D Engine 是一种小型的轻量级图形引擎，它使用了专业引擎的很多内容。该引擎小巧、快速、灵活而且可扩展，所以可使用它在移动设备上渲染高级图形，如图 1-3 所示。

图 1-3　使用 GFX 3D Engine 的 FPS 演示

- data：在该目录中，可以找到每个教程所使用的所有原始资产。这些资产既可以动态地链接到项目中(例如在 iOS 系统中)，也可以简单地复制到每个 Android 教程的 assets 目录中。请注意，所有原始项目 3D 场景都以.blend 形式(Blender 的默认文件扩展名)使用。虽然这并不是强制性的，但强烈建议为你的平台下载 Blender 的副本(可从 http://blender.org 处下载)。这样，就可以学习构建场景的方法以及如何连接资产和输出到 Wavefront OBJ(本书使用的官方 3D 模型交换格式)。
- EULA：在该目录中，可找到本书 SDK 依赖的不同库的所有最终用户许可协议。如果想要发布使用了本书 SDK 的商业应用程序，务必确保该应用程序符合所有这些许可协议。
- glsoptimizerCL：该目录包含了一个简单但功能强大的命令行程序源，你可以使用该程序优化 GLSL 代码(如第 5 章所述)。
- md5_exporter：该目录包含了 Blender 的 Python 脚本(v2.6x)。可使用该脚本将在 Blender 中创建的骨骼动画序列输出为 MD5 版本 10 文件格式(该脚本由 Paul Zirkle 慷慨提供)。
- template：该目录包含了从头创建新项目时所使用的原始模板项目。
- template_chapter#-#：为了加快速度并避免冗余，在阅读本书的教程时，可以复制这些目录，从而可以有一个良好的开始。通过使用默认的模板项目，可以避免从头构建所有内容。

1.3　导入项目

本书包含了 50 多个教程，其中有对简单技术的演示，也有成熟的游戏。为了能够在你的 IDE 中加载并重新生成这些项目，需要执行导入操作。为此，请遵循下面介绍的针对不同类型开发人员的相关指示。

1.3.1　针对 iOS 开发人员

通常对于 iOS 开发人员来说，导入文件非常简单。将项目导入到 XCode 中所需要做的只是双击.xcodeproj 文件。如果想进行编译，只需单击 Build &Run 按钮即可。

1.3.2　针对 Android 开发人员

如果正在使用 Eclipse，那么事情将会有点繁琐。需按下面的过程导入本书的项目。当然，该过程假设你已经正确安装并配置了 Android SDK、Android NDK、Eclipse Classic 以及 ADT 和 Sequoyah Android Native Development 插件(具体安装和配置过程如本章开头所述)。

一旦配置完所有必须预先准备好的文件，就可以遵循下面的步骤导入本书的项目文件：

(1) 从 Eclipse 主菜单中选择 File | New | Android Project。此时应该出现 New Android Project 对话框。

(2) 在 Project name 文本框中输入项目名称。例如，chapter2-1。

(3) 选择 Create project from existing source 选项。

(4) 单击 Browse 按钮，然后选择本章中现有的 Android 目录或模板项目。例如，<*path_to_sdk*>/SDK/_chapter2-1/Android。

(5) 单击对话框底部的 Finish 按钮。

图 1-4 演示了每个步骤。

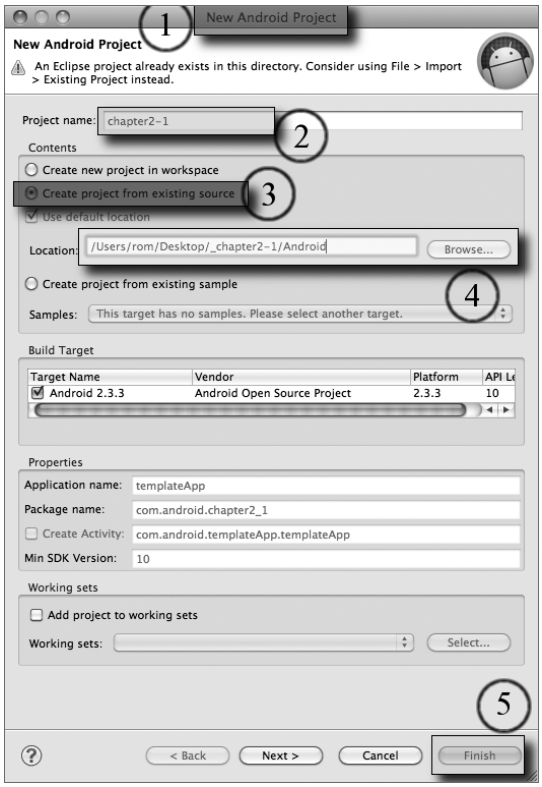

图 1-4　将 Android 项目导入到 Eclipse 中

每次使用 Eclipse 打开现有 Android 项目时，都必须完成上述导入过程。

1.4 模板

如本章前面所示，我们将主要使用本书 SDK 中提供的模板项目。该模板是一个 C/C++跨平台项目，它已经从内部为我们初始化一个可供随时使用的 OpenGL ES 2.0 环境。此外，该模板还提供了 init 和 exit 函数回调，你可将创建和销毁代码插入其中。

同时，该模板还提供了便于使用的回调机制，该机制充当通用 HUB，用来自动处理所有特定于平台的事件。

通过使用该机制，只需链接到一个想要拦截的特定事件的函数回调即可，然后就可以实时地获取该事件的更新信息。该机制包含所有 touche 事件，例如 ToucheBegan、ToucheMoved、ToucheEnded 以及加速计数据。换句话说，该机制为我们设置了一切。我们只需创建本书教程所指示的代码即可，而不必考虑特定于平台的问题。

正如本章标题所示，现在是时候开始入门了！为了熟悉教程的模板和类型，在学习本书时，需要遵循以下指示：

(1) 在 SDK 的根目录复制 template 项目目录，并重命名为 template_test。

(2) 将 template_test 项目加载到 IDE 中(如前所述，根据你自己的平台使用合适的导入方法)，然后打开 templateApp.cpp(对于 iOS 开发人员来说，该文件位于 Project Navigator 内的 templateApp 目录中；而对于 Android 开发人员来说，可在 Project Explorer 面板的 jni 目录下找到该文件)。

(3) 阅读相关的代码注释，了解每个函数的功能。

(4) 从初始代码中注释掉下面的回调(TEMPLATEAPP templateApp ={}: templateApp-ToucheBegan、templateAppToucheMoved 和 templateAppToucheEnded。

(5) 转到 templateAppInit 函数，并在函数的结束括号之前添加以下代码：

```
    /* Use the built-in GFX cross-platform API to print on the
console (XCode) or LogCat (Eclipse) that the execution pointer
passes the templateAppInit function. */
    console_print(
    "templateAppInit, screen size: %dx%d\n", width, height );
```

(6) 在 templateAppDraw 函数回调的结束括号之前添加以下代码块：

```
    /* Specify that you want to use a chili red color to clear the
screen and spice up your app. */
    glClearColor( 1.0f, 0.0f, 0.0f, 1.0f );
    /* Report that the execution pointer was here. */
    console_print( "templateAppDraw\n" );
```

(7) 在 templateAppToucheBegan 函数的结束括号之前添加以下代码：

```
    /* Print that the execution pointer enters the touche began
function and print the touche XY value as well as the number of
taps. */
```

```
console_print( "templateAppToucheBegan,"
               "touche: %f,%f"
               "tap: %d\n", x, y, tap_count );
```

(8) 对 templateAppToucheMoved 和 templateAppToucheEnded 函数重复步骤(7)所示的相同过程，同时使用将要处理的合适回调函数更新 console_print 文本。

(9) 转到已经链接到内置 C 函数 atexit 的 templateAppExit 函数，并在该函数的结束括号之前添加以下代码行：

```
console_print( "templateAppExit...\n" );
```

(10) 生成并运行应用程序。在应用程序运行期间，观察控制台或 LogCat(根据开发所使用的平台不同而定)。触摸屏幕并左右移动手指，在控制台上实时监视事件的触发方式以及触发序列。

1.5　小结

通过本章的学习，我们已经构建了开发环境，并安装了本书的 SDK，还学习了 SDK 的体系结构。

你学习了如何将新的或现有的项目导入 XCode 或 Eclipse 中，本章还详细概述了默认模板为我们完成的相关工作。

现在，我们已经做好准备开始游戏和图形编程中一条极具挑战的旅程。在学习下一章之前，请确保你已经完全理解了本章介绍的所有内容。

第 2 章

设置图形投影

本章内容

- 理解不同类型的投影矩阵的工作方式，以及使用这些矩阵的方式和时机
- 熟悉 SDK 附带的模板应用程序，以及学习如何自定义这些应用程序，从而满足自己的特定需求
- 构建你的第一个实际应用程序——学习如何设置并使用不同类型的投影

开始在屏幕上绘制任何图形之前，首先需要创建一个投影矩阵(projection matrix)。你所使用的图形类型将直接影响该矩阵的创建。不管图形是 2D、2.5D 还是 3D，每种投影矩阵都需要不同的初始化过程，从而使我们能创建满足特定需求的必要透视效果。

在本章，我们将学习现代手机游戏中常用的三种投影类型，并学习如何使用这些投影。

此外，本章还将讲述如何使用本书的 template 项目并完成三个渐进的练习。通过这些练习，我们将分别学习如何操作最常用的图形投影类型以及在屏幕上绘制简单几何体；如何处理顶点和片段着色器以及将它们链接到着色器程序；如何操作顶点属性和 uniform 变量；如何平移、旋转和缩放基本几何体；如何创建简单的摄像头“观察(look-at)”矩阵。

2.1 三种基本的投影类型

当使用 OpenGL ES 绘制图形时，通常需要记住绘制顺序以及想要绘制的透视空间。显然，该绘制顺序将直接影响投影类型以及投影矩阵的创建顺序。

例如，如果要绘制平视显示器(HUD)，并在场景顶部包含字符数据，则首先需要设置2D、2.5D 或 3D 投影(具体是哪一种，取决于将要开发的游戏类型)，然后绘制游戏场景。在颜色缓冲区中渲染场景之后，需要在场景的顶部显示相关的字符。如果只是对屏幕上的HUD 图形进行缩放来适应当前的绘图角度，将破坏整体的纵横比，并最终导致图形扭曲。因此，绘制游戏 HUD 的正确方法是创建“一个单位对应一个像素”比例的投影矩阵。由于 HUD 通常由多个 2D 图形组成，因此在屏幕上恪守该比例是非常重要的。1:1 的 2D 投影将允许我们在屏幕上以一致的方式创建这些图形。

在任何游戏类型中，可以使用三种不同类型的投影：

- **正射 2D 投影**：如前面示例所示，该投影类型可用来绘制任意 HUD。同时，该投影类型还可用于 side-scroller 类型及其他游戏类型，例如经典的 Tetris 以及较早的基于网格的角色扮演游戏。如前所述，该投影类型在屏幕上使用 1:1 的比例，这意味着绘图的大小直接受发送到 GPU(图形处理单元)的正方形(四边形)的像素大小的影响。在更现代的用法中，该投影类型主要用来在屏幕上绘制菜单、文本、HUD 或其他类型的静态或半动态 2D 信息。考虑到无法使用维度的第三个级别，同时因为深度范围被限制为 －1 到 1(－1 表示屏幕上的最近点，而 1 则表示最远点)，所以从根本上讲，深度缓冲区的使用已经过时，应该将其关闭。此外，渲染顺序应该是从后到前，以免覆盖像素。
- **正交(奥托)投影**：该投影类型允许我们在半 2D 环境中绘制图形，并且仍然可以考虑维度的第三个级别。在该投影中，透视完全基于当前的屏幕比。在现代的 2D/2.5D 射击类型游戏以及即时战略游戏中，都可以使用该投影类型。当使用该投影类型时，仍可以访问维度的第三个级别，并充分利用深度缓冲区。图 2-1 显示了一个游戏示例，该游戏结合使用了 2D 正交投影和深度缓冲区。

图 2-1　通过 SIO2 交互的 Ragdoll 发射器

- **透视投影**：在所有精美的 3D 游戏中(在这些游戏中，需要在屏幕上渲染实时动态且逼真的 3D 世界)，都需要使用该投影类型。通过使用该投影类型，我们可以模拟从真正的人眼角度看到的 3D 世界。除了屏幕纵横比之外，这种投影类型还需要考虑观看场景的摄像头的视场，图 2-2 显示了 iOS 中的超前汽车游戏。

图 2-2　通过 SIO2 交互的 Sky 比赛

2.2　正射 2D 投影

接下来，让我们学习构建 2D 正射投影矩阵所需的代码。

在本节，我们将从头创建一个使用了本书 SDK 中 template 项目的简单程序。该程序将在绝对像素坐标中使用屏幕投影在屏幕上绘制一个可缩放的彩色四边形。

在开始编写代码之前，首先需要复制位于 SDK 根目录中的 template 项目。为此，请右击该目录，并创建该目录的本地副本，然后重命名为 chapter2-1。在将该项目加载到你所喜欢的 IDE 中后，找到 templateApp.cpp 源文件，并在代码编辑器中将其打开。

本章教程的第一步是通过删除本章所有练习中不需要的回调函数，调整新创建的 template 项目。

我们将重点介绍 templateAppInit 和 templateAppDraw 函数回调，所以可以删除其他的回调和代码注释，从而可以在开始时拥有一个干净的模板。请修改代码，以便得到下面所示的结果：

```
#include "templateApp.h"

TEMPLATEAPP templateApp = { templateAppInit,
                            templateAppDraw };

void templateAppInit( int width, int height ) {
   atexit( templateAppExit );
   GFX_start();
   glViewport( 0, 0, width, height );
}

void templateAppDraw( void ) {
   glClear( GL_COLOR_BUFFER_BIT );
}

void templateAppExit( void ) {
}
```

2.2.1 程序和项目初始化

下面所示的步骤将引导我们完成一个必要的过程，以便设置程序所使用的全局变量。此外，我们将学习本书的 SDK 所提供的不同类型的结构，这些结构可帮助我们操作程序的不同方面，例如着色器以及从磁盘加载资产等。还将学习如何使用 SDK 中提供的 API 以创建第一个 2D 屏幕投影矩阵。

(1) 在 templateApp.cpp 资源文件的顶部(在 templateAppInit 函数声明之前)定义顶点和片段着色器文件名，如下所示：

```
#define VERTEX_SHADER ( char * )"vertex.glsl"
#define FRAGMENT_SHADER ( char * )"fragment.glsl"
```

(2) 声明一个标志来启用(ON(1))或禁用(OFF(0))着色器的调试功能。没有启用调试功能通常意味着着色器的编译速度更快，但此时也不会报告任何错误，所以如果产生错误，最终的结果将无法确定。因此，在开发应用程序时应该保持该标志为 ON(1)。

```
#define DEBUG_SHADERS 1
```

(3) 创建用来管理所有着色器程序的空白 PROGRAM 结构，如下所示：

```
PROGRAM *program = NULL;
```

在全书中，我们都将使用 PROGRAM 结构来处理着色器程序。在 SDK/common/program.cpp 和 program.h 源文件中可找到该实现的完整源代码。此外，与 PROGRAM 结构相链接的 SHADER 结构的代码也可在 shader.cpp 和 shader.h 源文件中找到(这两个文件同样位于本书源代码包的 SDK/common 目录中)。从根本上讲，该预设结构(即 PROGRAM 结构)可以处理顶点和片段着色器与主着色器程序之间的所有交互。这样，我们就可以编译自己的着色器，并且自动将它们链接到另一个着色器程序。此外，该结构还提供了一个简单易行的方法来访问由 GPU 自动赋值的 uniform 变量和顶点属性，还提供了一种易用的回调机制，允许我们实时地访问、设置和修改这些 uniform 变量。

(4) 声明 MEMORY 结构指针，如下所示：

```
MEMORY *m = NULL;
```

该对象也是 SDK 的一部分。从根本上讲，除了该对象的所有工作都在内存中完成之外，它的行为类似于 C 语言中的 FILE。在移动设备上，系统内存是处理数据的最快捷方法。在本练习中，我们将使用 MEMORY 结构从磁盘中读取着色器文件。作为一般性规则，我们应尽可能地避免磁盘访问。通过使用该结构，在加载资产时，将获得更高的加载速度以及更大的灵活性。

(5) 在该步骤中，我们将根据需要修改 templateAppInit 的内容。考虑到读者是第一次使用该函数，所以将详细解释一些内容。首先从首次函数调用开始。该调用使用标准的 atexit 函数，这样当应用程序退出时可得到反馈。然后编写必要的代码，以便刷新内存中

的内容。对于本书的每个教程和练习来说，请确保始终将 templateAppExit 与 atexit 相链接，如下所示：

```
void templateAppInit( int width, int height ) {
   atexit( templateAppExit );
```

(6) 使用 GFX 辅助函数(该函数也是本书 SDK 的一部分)开启 GLES 初始化过程：

```
GFX_start();
```

该代码行对所有的 OpenGL ES 标准计算机状态进行初始化，以便确保正确地设置一切参数，而不管当前使用的设备的驱动程序是什么。如果想了解关于 GFX_start 函数的更多信息，可参阅位于 SDK/common/gfx.cpp 中的源代码。

如果仔细查看 GFX 的实现过程，你会发现它还提供了 GLES2 所不具备的矩阵操作功能，并且模仿了 GLES1 和 GL 桌面实现过程所提供的矩阵机制。

通过使用 GFX 辅助函数，可以非常容易地对矩阵执行 push、pop、加载以及乘法运算，还可以像在 OpenGL ES 和 OpenGL 较早的版本中那样直接访问模型视图、投影、法线以及纹理矩阵。

(7) 可使用标准的 glViewport 命令，用当前的屏幕尺寸设置 GL 视区：

```
glViewport( 0, 0, width, height );
```

(8) 现在，使用 GFX_set_matrix 模式功能告诉 GFX 实现过程重点关注投影矩阵，以便建立 2D 投影：

```
GFX_set_matrix_mode( PROJECTION_MATRIX ); {
```

(9) 声明两个用来保存屏幕宽度和高度一半值的临时 float 变量，如下所示：

```
float half_width = ( float )width * 0.5f,
      half_height = ( float )height * 0.5f;
```

(10) 接下来，需要确保当前投影矩阵是干净的。为此，使用下面的函数调用来加载单位矩阵：

```
GFX_load_identity();
```

(11) 现在，我们已经做好了设置 2D 屏幕投影的准备。为此，需要使用 GFX_set_orthographic_2d 函数，并在参数中传入屏幕尺寸的一半(以视区矩阵的原点为基准的正方向和负方向，包括左、宽、右和高)。该操作将使用 GL 单元与屏幕像素之间 1:1 的比例，在屏幕的中心定位投影矩阵的原点(如果愿意，也可以定位支点)。

```
GFX_set_orthographic_2d( -half_width,
                          half_width,
                         -half_height,
                         half_height );
```

(12) 接下来，为了符合 OGL，需要将该矩阵转换到屏幕的左下角，如下所示：

```
GFX_translate( -half_width, -half_height, 0.0f );
```

你可能已经知道，OpenGL 使用颜色缓冲区的左下角作为(0,0)坐标。而我们所插入的 GFX_translate 函数调用将矩阵的默认位置转换为与 GL 颜色缓冲区坐标相对齐，从而确保所有图形都是相对于屏幕左下角绘制的。

(13) 如本章前面所述，当使用正射 2D 投影类型时，并不需要使用深度缓冲区，因为大多数情况下，使用该类型时只需重写颜色缓冲区即可。所以关闭深度缓冲区，如下所示：

```
glDisable( GL_DEPTH_TEST );
```

(14) 由于深度缓冲区被关闭，因此还需要关闭深度掩码：

```
    glDepthMask( GL_FALSE );
}
```

2.2.2 顶点和片段着色器

因为本书旨在向读者展示一种实现游戏和图形引擎不同元素的简单方法，所以并不介绍 GLSL ES 语言的细节信息。如果想了解关于 GLSL ES 的更多信息，可以免费访问 http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf。

在开始介绍 templateAppInit 函数中加载所需顶点和片段着色器的核心代码之前，首先需要创建顶点和片段着色器。

由于着色器完全是基于文本的，因此可使用任何文本编辑器来编辑它们。对于本例来说，需要创建两个空白着色器文件。请将这两个文件分别命名为 vertex.glsl 和 fragment.glsl，并保存在当前练习目录的根目录下(SDK/chapter2-1/)。

为对这两个着色器进行打包，并可以在应用程序包中访问，需要将它们链接到项目。

如果你是一位 XCode 用户，可首先使用 Finder 选择这两个.glsl 文件，然后将它们直接拖放到项目树的 Resources 目录中，并确认该操作。

如果你是一位 Eclipse 用户，可首先选择这两个.glsl 文件，然后将它们复制/粘贴到项目的 assets 目录。

对于 iOS 和 Android 系统来说，着色器文件将在运行时与应用程序捆绑并供访问。此外，请注意，从现在开始，每当需要将资产链接到项目时，都会引用该过程，因为对于模型、纹理、声音、物理文件来说，链接过程都是相同的。

1. vertex.glsl

对于本例来说，我们仅在屏幕上绘制一个彩色正方形。首先，需要编写必要的代码来转换顶点，然后获取每个顶点的颜色并将其发送到片段着色器，以便进行像素处理。为此，请打开 vertex.glsl 文件，并完成下面的步骤：

(1) 在顶点着色器的第一行定义一个 uniform 变量(这意味着可在 C/C++代码中操作该

变量的值)。为了对向下发送到着色器的每个顶点进行转换，以便在屏幕上进行显示，需要将当前的投影矩阵乘以当前模型视图矩阵，而该 uniform 变量将保存最终相乘的结果。

```
uniform mediump mat4 MODELVIEWPROJECTIONMATRIX;
```

(2) 然后，定义一个变量来保存顶点着色器当前正在处理的顶点位置。为处理此类变量，需要使用 attribute 说明符来声明该变量。因为我们处理的是顶点位置，所以将该变量命名为 POSITION，如下所示：

```
attribute mediump vec4 POSITION;
```

(3) 同时，该正方形还将接收与每个顶点位置相关联的颜色。所以使用 attribute 关键字声明另一个用来定义每个顶点颜色的变量，并命名为 COLOR：

```
attribute lowp vec4 COLOR;
```

(4) 如前所述，顶点着色器主要处理顶点。所以，为将 COLOR 变量传递给片段着色器，以便进行像素处理，必须使用一个中间变量将该颜色发送到片段着色器。此类任务所使用的说明符是 varying，由于前面已将大写的 COLOR 声明为 attribute，因此此时可声明为小写的 color：

```
varying lowp vec4 color;
```

(5) 接下来，需要插入着色器的 main 函数。与 C/C++中一样，为了确定执行指针的默认入口点，每个着色器都需要有一个 main 函数：

```
void main( void ) {
```

(6) 每当需要处理一个顶点并将其显示在屏幕上时，都必须使用内置的 gl_Position 变量。为了能够在屏幕上看到一个顶点，必须将该顶点的位置(投影矩阵和模型视图矩阵的相乘结果)赋予该变量，如下所示：

```
gl_Position = MODELVIEWPROJECTIONMATRIX * POSITION;
```

(7) 设置顶点着色器的最后一个至关重要的操作是将顶点颜色发送到片段着色器。为此，只需将值 COLOR 赋予 varying 变量 color，从而使该值与片段着色器中类似的变量(稍后将创建)相关联。

```
   color = COLOR;
} /* Close the main function */
```

(8) 保存文件。

2. 关于精度限定符的一些说明

在开始介绍片段着色器代码之前，你可能已经注意到，当在 GLSL ES 中声明任何变量时，都需要使用精度限定符。

尤其是在 GLSL ES 中使用精度限定符时，可以控制浮点精度(包括向量和矩阵)以及整数变量。当大量使用精度限定符时，这些限定符可以极大地提升着色器程序的性能并提高执行速度。

执行着色器程序是所占用的 GPU 时间越少，可添加的 GL 指令也就越多，从而有机会在屏幕上渲染更复杂的绘图，同时保持可接受的帧率。

表 2-1 列出了相关的精度关键字以及浮点数和整数的范围。

表 2-1　精度限定符表

精　　度	浮点数范围	整 数 范 围
highp	−2^62～2^62	−2^16～2^16
mediump	−2^14～2^14	−2^10～2^10
lowp	−2.0～2.0	−2^8～2^8

3. fragment.glsl

接下来是时间编写本练习的片段着色器了。你可能已经猜到，片段着色器主要处理基于像素的操作。在第一个示例中，我们并不会在片段着色器中编写太多的代码；然而，在本书的后续章节中，代码的复杂程度将会显著增加。

使用自己喜欢的文本编辑器打开 fragment.glsl 文件，然后按照下面的步骤创建第一个片段着色器：

(1) 使用与 vertex.glsl 文件中相同的语法声明名为 COLOR 的 varying 变量。由于这两个变量完全相同，且都使用 varying 说明符进行声明，因此着色器编译器会自动将它们关联起来，并将在顶点着色器中设置的值发送到片段着色器，以便进行相应处理。

```
varying lowp vec4 color;
```

(2) 现在，通过插入 main 函数，创建片段着色器的主入口点：

```
void main( void ) {
```

(3) 要指定在顶点着色器中指定的颜色，只需将 varying 变量颜色赋予内置的 gl_FragColor 变量：

```
    gl_FragColor = color;
}
```

(4) 保存文件。

2.2.3　链接着色器程序

接下来，让我们看看将顶点和片段着色器链接到 PROGRAM 结构所需的代码。

由于这是本书的第一个示例，同时在后续章节中将逐步使用 PROGRAM 结构及其相关

功能，因此本节将介绍完整的初始化过程。

现在，让我们返回到 templateApp.cpp，或更准确地讲，是返回到 templateAppInit 函数。

(1) 首先，必须初始化 program 变量指针，所以请添加下面的代码行：

```
program = PROGRAM_init( ( char * )"default" );
```

该函数自动分配所需的内存，将整个结构设置为空白并做好接收其他命令的准备，以便成功链接着色器程序。该函数的最后一个参数主要是为了使用方便。在本练习中，我们将仅处理一个 PROGRAM 结构。但随着阅读深入，将会处理多个 PROGRAM 结构，所以该函数能将一个名称与这些结构相关联，以便进行识别。

(2) 接下来，通过调用相关联的_init 函数，创建一个新的顶点和片段 SHADER 指针，如下所示：

```
program->vertex_shader = SHADER_init( VERTEX_SHADER,
                                      GL_VERTEX_SHADER );
program->fragment_shader = SHADER_init( FRAGMENT_SHADER,
                                        GL_FRAGMENT_SHADER );
```

SHADER_init 函数的第一个参数表示着色器使用的内部名称，第二个参数表示类型，所以 GLES 可以相应地将它们联系起来。

如你所见，我们对 PROGRAM 结构的两个 SHADER 指针进行了初始化。或者，也可以将它们作为单独的 SHADER 指针进行初始化，然后手动将它们附加到 PROGRAMS，以便进行重用。

(3) 为了能够从磁盘中加载所创建的着色器，需要在内存中加载相关的内容。为此，需要像下面所示的那样使用 MEMORY 结构：

```
m = mopen( VERTEX_SHADER, 1 );
```

其中，第一个参数是文件名称，而第二个参数指定路径是否相对于应用程序(1 表示是，0 表示否)。

(4) 为安全起见，也为了进一步了解 mopen 函数背后的加载机制，对 m 变量进行一次指针检查，如下所示：

```
if( m ) {
```

注意，如果文件加载失败，该指针将为 NULL。非空指针可以确保文件被成功加载，并且驻留在内存中。

(5) 现在，我们需要对包含在 MEMORY 缓冲区指针(m->buffer)中的顶点着色器代码进行编译，为此，将代码传递到 SHADER_compile 函数，如下所示：

```
if( !SHADER_compile( program->vertex_shader,
                     ( char * )m->buffer,
                     DEBUG_SHADERS ) ) exit( 1 );
```

显而易见，该函数的第一个参数是一个有效的 SHADER 结构指针，它表示将对其代码进行编译的着色器。第二个参数表示着色器源(可从内存中访问该源)。最后一个参数允许启用或禁用调试功能。

从上面的代码可以看到，如果 SHADER_compile 函数编译代码失败，该函数将返回 0。而在本次练习中，如果失败，将会触发退出操作并调用 templateAppExit 函数。

(6) 请输入下面所示的代码，以便释放 MEMORY 结构指针 m，因为在下一步将重新使用该指针来加载片段着色器。

```
}
m = mclose( m );
```

(7) 接下来加载片段着色器并进行编译。为此，需要重复使用为顶点着色器使用的相同加载代码结构，但此时必须指定 FRAGMENT_SHADER 文件，如下所示：

```
m = mopen( FRAGMENT_SHADER, 1 );
if( m ) {
    if( !SHADER_compile( program->fragment_shader,
                         ( char * )m->buffer,
                         DEBUG_SHADERS ) ) exit( 2 );
}
m = mclose( m );
```

到目前为止，我们已经成功编译顶点和片段着色器，并做好链接到着色器 PROGRAM 的准备。注意，如果想成功链接到着色器程序，至少需要一个有效且已编译的顶点着色器以及一个有效且已编译的片段着色器；否则，着色器编译器将会失败，并且会在系统控制台报告错误。

(8) 为了完成顶点和片段着色器的最终链接阶段，可调用 PROGRAM_link 函数，如下所示：

```
if( !PROGRAM_link( program, DEBUG_SHADERS ) ) exit( 3 );
```

注意，与前面的 SHADER_compile 函数相类似，PROGRAM_link 函数的最后一个参数确定是否使用调试功能。如前所述，如果产生错误，该函数将返回 0，并在系统控制台报告错误消息。

2.2.4 绘图代码

为让应用程序运行起来，下一步处理 templateAppDraw 函数。在该函数中，我们将插入一些必要的代码，以便在屏幕上实际显示一些内容。此外，我们还将设置不同的顶点属性并控制应用程序中的 uniform 变量。

1. 准备框架

在能够实际绘制图形之前，首先必须准备在单个帧中顺序渲染所需的数据。此时，在屏幕上显示任何帧之前，还缺少一些重要信息。我们需要声明所有顶点、顶点颜色以及可

视转换(练习中的正方形将使用该转换，以便在屏幕上显示)。为此，请按照下列步骤操作：

(1) 首先声明一些 2D 位置(顶点)，然后将这些顶点链接在一起，从而在屏幕上绘制一个正方形。为声明这些顶点，在 templateAppDraw 开始括号之后插入下面的声明语句：

```
static const fl oat POSITION[ 8 ] = {
0.0f, 0.0f, // Down left (pivot point)
1.0f, 0.0f, // Up left
0.0f, 1.0f, // Down right
1.0f, 1.0f // Up right
};
```

(2) 接下来，需要声明顶点颜色，并将它们与步骤(1)中创建的顶点位置数组相关联。为此，必须按照 POSITION 数组中对应的顶点顺序声明顶点颜色。其结果是，第一个顶点将与第一个 RGBA 颜色相关联，第二个顶点与第二个 RGBA 颜色相关联，以此类推。输入下面的代码：

```
static const fl oat COLOR[ 16 ] = {
1.0f /* R */, 0.0f /* G */, 0.0f /* B */, 1.0f /* A */, /* Red */
0.0f, 1.0f, 0.0f, 1.0f, /* Green */
0.0f, 0.0f, 1.0f, 1.0f, /* Blue */
1.0f, 1.0f, 0.0f, 1.0f /* Yellow */
};
```

(3) 现在已经拥有了绘图需要的所有值，可以开始构建应用程序的渲染循环了。首先指定用来清理颜色缓冲区的颜色值(此时为淡灰色)，在 glClear 函数调用之前插入以下代码：

```
glClearColor( 0.5f, 0.5f, 0.5f, 1.0f );
```

(4) 为了避免覆盖像素，每次调用 templateAppDraw 函数时都需要告诉 OpenGL 清理颜色缓冲区，如下所示：

```
glClear( CL_COLOR_BUFFER_BIT );
```

(5) 在步骤(1)中，我们实际上创建了一个 1×1 像素的正方形，这在屏幕上是很难看到的。为在屏幕上看到这个正方形，需要添加下面的代码，以便在 X 和 Y 轴上对其进行缩放：

```
   /* Select the model view matrix. */
   GFX_set_matrix_mode( MODELVIEW_MATRIX );
   /* Reset it to make sure you are going to deal with a clean
identity matrix. */
   GFX_load_identity();
   /* Scale the quad to be 100px by 100px. */
   GFX_scale( 100.0f, 100.0f, 0.0f );
```

2. 绘制正方形

接下来的逻辑步骤是调用必要的 API，从而告诉 GPU 在屏幕上绘制一些内容。为此，请按照下列步骤操作：

(1) 首先，添加下面的 if 语句，从而确保拥有有效的着色器程序 ID：

```
if( program->pid ) {
```

你可能已经知道，GLES 索引通常从 1 开始。所以，如果某个着色器编译失败，那么 program->pid 将为 0，后续代码将不会执行。

(2) 声明两个用来保存顶点属性和统一位置的临时变量，如下所示：

```
char attribute, uniform;
```

为了能够架起应用程序数据与 GPU 数据之间的桥梁，需要使用这些统一位置。

(3) 现在，必须告诉 GPU 你将使用哪个程序来绘图；否则，GPU 将不知道如何处理发送来的数据。为此，请调用下面的函数：

```
glUseProgram( program->pid );
```

(4) 输入下面的代码，从视频存储器中检索 uniform 变量位置：

```
uniform =
PROGRAM_get_uniform_location( program,
( char * )"MODELVIEWPROJECTIONMATRIX" );
```

注意，我们想要检索位置的变量名必须与着色器中声明的变量名完全相同。

(5) 在检索 uniform 变量的位置后，接下来更新 GPU 中的数据。为此输入下面的代码：

```
      glUniformMatrix4fv(
      /* The location value of the uniform. */
      uniform,
      /* How many 4x4 matrix */
      1,
      /* Specify to do not transpose the matrix. */
      GL_FALSE,
      /* Use the GFX helper function to calculate the result of the
current model view matrix multiplied by the current projection matrix. */
      ( float * )GFX_get_modelview_projection_matrix() );
```

(6) 现在，我们以相同的方式处理顶点属性。请输入下面的代码，检索顶点位置：

```
attribute =
PROGRAM_get_vertex_attrib_location( program,
( char * )"POSITION" );
```

(7) 一旦获取 POSITION 属性位置，就可以通过使用 glEnableVertexAttribArray 函数，告诉 GLES 可以使用顶点属性位置了，如下所示：

```
glEnableVertexAttribArray( attribute );
```

请记住，OpenGL ES 是基于计算机状态实现的，所以务必确保所有想要使用的内容都被启用，而不需要使用的内容则都被禁用。

(8) 还需要告诉 GLES 需要使用该属性的哪些数据。为此，请使用 glVertexAttribPointer 函数，如下所示：

```
    glVertexAttribPointer(
    /* The attribute location */
    attribute,
    /* How many elements; XY in this case, so 2. */
    2,
    /* The variable type. */
    GL_FLOAT,
    /* Do not normalize the data. */
    GL_FALSE,
    /* The stride in bytes of the array delimiting the elements,
in this case none. */
    0,
    /* The vertex position array pointer. */
    POSITION );
```

(9) 请输入下面的代码，对 COLOR 数组执行相同的操作：

```
    attribute =
    PROGRAM_get_vertex_attrib_location( program,
                                        ( char * )"COLOR" );
    glEnableVertexAttribArray( attribute );
    glVertexAttribPointer( attribute,
                           4,
                           GL_FLOAT,
                           GL_FALSE,
                           0,
                           COLOR );
```

(10) 现在调用 glDrawArrays 函数，告诉 GPU 使用特定模式(从数组的指定索引开始)绘制相应的数据，并告诉使用多少数据：

```
          glDrawArrays(
          /* The drawing mode. */
          GL_TRIANGLE_STRIP,
          /* Start at which index. */
          0,
          /* Start at which index. */
          4 );
    } /* Close the program->pid check. */
```

(11) 最后，作为一种安全措施，还需要调用 GFX_error 辅助函数。该函数将检查在绘制当前帧时是否产生 GL 错误(如果产生错误，那么对于 XCode 用户来说，将会在控制前显示错误消息，而对于 Eclipse 用户来说，则在 LogCat 中显示)。

```
    GFX_error();
```

3. 清理

转到 templateAppExit 函数，并按照下面的步骤来清理已分配的内存：

(1) 在该函数的开始括号之后插入一条简单的 printf 语句，以便通知应用程序已退出：

```
printf("templateAppExit...\n");
```

(2) 然后，对 MEMORY 结构变量 m 进行一次指针检查，因此当执行指针到达 printf 语句时，该变量可能仍然被赋予了值，如果是，就通过 mclose 函数使用该变量：

```
if( m ) m = mclose( m );
```

(3) 添加以下代码，从而完成与步骤(2)类似的检查，但此次需要释放两个 SHADER 结构以及一个 PROGRAM 结构：

```
if( program && program->vertex_shader )
   program->vertex_shader = SHADER_free( program->vertex_shader );
if( program && program->fragment_shader )
   program->fragment_shader = SHADER_free( program->fragment_shader );
if( program )
   program = PROGRAM_free( program );
```

现在，我们已经做好运行该程序的准备，所以请单击生成和运行按钮。此时，应该看到如图 2-3 所示的运行结果。

图 2-3　你的第一个正射 2D 投影

关于 2D 正射投影的内容到此为止。在阅读完本节后，你将能够使用 2D 正射投影(1 个单元等于 1 个像素)绘制简单的彩色正方形。随着学习后续内容，你将发现可以重复使用该实现过程在屏幕上绘制游戏的 HUD 和菜单。

2.3　正交投影

在本节，我们将学习如何创建可缩放的正交投影，通过该投影，可以使用屏幕长宽比来建立简单的透视视图。为了详细研究正交投影的可能功能，我们将首先修改前一节中创建的代码。然后向现有的坐标系统添加一个新维度。最后，使用观察矩阵创建一个简单摄像头，该摄像头允许我们在三维场景中指定位置和观察者的视角。

获取正交投影

首先复制前面的项目 chapter2-1(位于 SDK 目录中)，并重命名为 chapter2-2。然后修改该项目，使其集成正交投影。请按照下列步骤操作：

(1) 在 templateAppInit 函数中找到 GFX_set_matrix_mode 函数调用，并删除花括号{和}之间的代码块。

(2) 在花括号{和}之间插入下面所示的代码行，从而使用正交投影初始化替换前面的屏幕投影：

```
    /* Clean the projection matrix by loading an identity matrix. */
    GFX_load_identity();
    GFX_set_orthographic(
    /* The screen ratio. */
    ( float )height / ( float )width,
    /* The scale of the orthographic projection; the higher to 0,
the wider the projection will be. */
    5.0f,
    /* The aspect ratio. */
    ( float )width / ( float )height,
    /* The near clipping plane distance. */
    1.0f,
    /* The far clipping plane distance. */
    100.0f,
    /* The rotation angle of the projection. */
    0.0f );
```

注意远端和近端的剪切面——从根本上讲，这些剪切面控制观察者可以从多远看到该几何体。如果转换后的顶点远于远端的剪切面或者近于近端的剪切面，那么你将无法看到这些顶点，因为它们都被裁剪掉了。

GFX_set_orthographic 函数的最后一个参数允许控制投影的旋转角度。大多数现代设备都可以根据当前的设备方向翻转屏幕。也可以使用该参数根据设备方向调整场景的方向。

(3) 在正交投影设置调用的后面添加如下代码：

```
glDisable( GL_CULL_FACE );
```

该函数告诉 GLES 不要剪切背面。为提高运行速度，大多数现代 GPU 会自动分析三角形，以确定它们是否面对观察者。如果不是，则会在早期舍弃它们，以免执行额外的计算。

在本教程的后面，我们将动态显示该正方形，所以此时必须确保在屏幕上正确地绘制了该正方形，而不管旋转角度如何。即使该正方形被反转了，我们也不希望 GPU 舍弃屏幕上的任何顶点数据。

(4) 使用下面的代码重写 templateAppDraw 函数中的 POSITION 数组声明：

```
    static const fl oat POSITION[ 12 ] = {
    -0.5f, 0.0f, -0.5f, // Bottom left
     0.5f, 0.0f, -0.5f,
    -0.5f, 0.0f, 0.5f,
     0.5f, 0.0f, 0.5f // Top right
};
```

注意，上述代码向顶点位置数据添加了第三个维度。由于此时我们将从 XY 屏幕坐标系统转移到 XYZ 坐标系统(其中的 Z 轴代表向上的向量)，因此需要将正方形的支点转换为围绕 X:0、Y:0 和 Z:0 进行旋转。

(5) 由于需要完全激活深度缓冲区和深度掩码(这与前一个示例不同)，因此必须告诉 OpenGL ES 每一帧都需要清理深度缓冲区：

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

(6) 为创建摄像头矩阵，找到 templateAppDraw 函数中的 GFX_scale 函数，并用下面的变量声明进行替换：

```
     /* The eye position in world coordinates. */
vec3 e = { 0.0f, -3.0f, 0.0f },
     /* The position in world space where the eye is looking. */
     c = { 0.0f, 0.0f, 0.0f },
     /* Use the positive Z axis as the up vector. */
     u = { 0.0f, 0.0f, 1.0f };
```

(7) 接下来调用 GFX_look_at 函数，并使用 e、c 和 u 作为参数：

```
GFX_look_at( &e, &c, &u );
```

该 GFX_look_at 调用将创建所需的观察矩阵。如果想了解该函数背后的更多数学知识，可参考 SDK/common/gfx.cpp 中的源代码。

(8) 现在是时候在屏幕上动态显示该正方形了！首先需要创建一个静态的 float 变量，该变量将动态增加，以便动态显示正方形支点的 Y 位置。为此输入下面的代码：

```
static float y = 0.0f;
      /* Increment the Y location value every frame. */
      y += 0.1f;
```

(9) 使用下面的 GFX_translate 函数在 Y 轴上转换正方形的支点：

```
GFX_translate( 0.0f /* X */,
               y,
               0.0f /* Z */ );
```

(10) 由于已经定义了变量 y，因此在使用 GFX_rotate 函数时重用该变量来动态显示正方形的旋转：

```
GFX_rotate( /* Boost the angle a bit by multiplying it. */
            y * 50.0f,
            /* X axis */
            1.0f,
            /* Y axis */
            1.0f,
            /* Z axis */
            1.0f );
```

该函数接收 4 个参数：第 1 个参数表示旋转角度(以度为单位)，其他 3 个参数则分别表示受旋转角度影响的各轴。

(11) 在步骤(4)中，我们修改了各个 POSITION 组件的大小。为了能够让 GLES 使用新的三维顶点位置数组正确绘制正方形，需要将 glVertexAttribPointer 的参数从 2(XY)调整为 3(XYZ)，如下所示：

```
glVertexAttribPointer( attribute,
                       /* 2 */ 3,
                       GL_FLOAT,
                       GL_FALSE,
                       0,
                       POSITION );
```

生成并运行该程序!应该可以看到该正方形在 X、Y、Z 轴上进行旋转(如图 2-4 所示)，当 Y 位置超过 100 一段时候后，正方形将会消失，这也就意味着它被远端的剪切面剪切。

图 2-4　正交正方形

恭喜你！你已经成功地设置了一个正交投影，使用了深度缓冲区以及创建了一个完整的 3D 对象(虽然这只是一个非常简单的对象—但却是很好的开始)。通过使用转换和旋转，我们在屏幕上动态显示了正方形，还设置了一个基本观察矩阵来模拟现实空间中的摄像头。对初学者来说，上述内容已经非常多了！

你可能已经注意到，当正方形在屏幕上旋转时，投影看上去似乎不正确——当正方形旋转到一个很大的角度时尤其如此。这是因为该投影完全基于屏幕和纵横比，而不是基于真正的视角。然而，许多 3D 编辑软件都使用这种模式类型，因为通过它可以直接看到观察者所面对的几何体。同时，该模式类型对于某些 2D 或 2.5D(在绘制时仍然可以使用深度缓冲区)游戏来说也是非常理想的。特别在 2.5D 游戏中，通常有一个静态摄像头角度向下查看左右上下平移的场景，从而

提供一种逼真的 2.5D 体验。

关于正交投影的介绍到此为止。下一节中将添加一个视场，从而解决正交投影中出现的视角问题。该视场表示观察者的眼睛角度，从而创建逼真的 3D 投影矩阵。

2.4 透视投影

在本节，我们将学习如何设置真实的 3D 透视视图。这也是本书所讲授的最后一种投影。在本书几乎所有的示例和练习中，都会使用该类型的投影。

同样，在深入研究代码之前，首先复制上一节中的项目(chapter2-2)，为保持一致性，将复制的项目文件夹重命名为 chapter2-3。

由于我们已经编写了所有必需的代码，因此将正交投影改为逼真的 3D 透视投影非常简单！只需找到 templateAppInit 中的 GFX_set_orthgraphic 函数，并用下面的代码替换函数调用：

```
GFX_set_perspective(
/* Field of view angle in degree. */
45.0f,
/* The screen aspect ratio. */
( float )width / ( float )height,
/* The near clipping plane. */
0.01f,
/* The far clipping plane. */
100.0f,
/* The device screen orientation in angle to use. */
0.0f );
```

可以看到，除第一个参数视场(简称 FOV)外，该函数与 GFX_set_orthographic 函数几乎完全相同。当提到 FLV 时，角度越大，视角范围也就越宽广；相反，角度越小，投影也就越窄。

或者，也可在测试应用程序之前，删除或注释掉 GFX_translate 调用(位于 templateAppDraw 中)，以便在正方形旋转时能够真实地观察投影矩阵上的3D透视效果。

现在，生成并运行应用程序！应该可以看到图 2-5 所示的结果。

当运行应用程序时，可以看到由 GFX_set_perspective 函数创建的投影矩阵与前面创建的投影矩阵是不一样的。主要区别是不管从哪个角度看，正方形看起来都非常不错，这是因为透视矩阵计算包括了视场，从而以真实世界中观察正方形的方式进行计算。

图 2-5 使用真实的 3D 透视旋转

2.5　小结

本章介绍了如何对直接影响绘图角度的三种主要投影矩阵进行设置。通过使用这些投影，可创建任何类型的 2D、2.5D 或 3D 游戏。

同时，还分别学习了如何在屏幕上设置和绘制基本形状，如何使用深度缓冲区，如何操作顶点属性和 uniform 变量以及如何创建观察矩阵来控制观察者眼睛的位置和方向。

这是不错的开头，现在你已经掌握了创建一些逼真游戏和 3D 应用程序需要的所有基础知识。

在学习下一章之前，我建议你回顾一下本章的练习，并确保完全理解练习中所演示的内容。此外，还应该尝试修改一些参数，以便更加熟悉整体程序结构和可能的功能。

作为额外的练习，可尝试将 templateAppInit 函数中的投影矩阵创建代码移到 templateAppDraw 函数中，从而将屏幕投影与正交或 3D 透视投影混合起来。