HZ BOOKS
华章科技

资深Android开发专家多年实践经验结晶，全面深入讲解OpenGL三维游戏和动态壁纸开发

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

移动开发

OpenGL ES 2 for Android： A Quick-Start Guide

OpenGL ES应用开发实践指南

Android卷

（美） Kevin Brothaler 著　刘力平 苏统华◎译

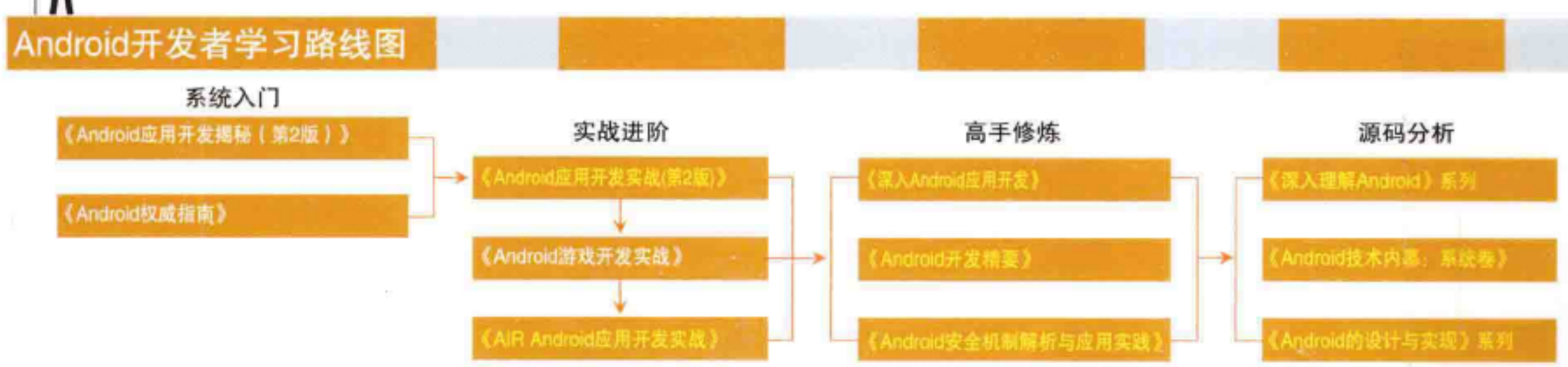
机械工业出版社
China Machine Press

每天几百万部设备的出货量，使Android取得了前所未有的蓬勃发展。这是你学习为Android开发三维游戏和动态壁纸的最佳时机。你会学到所有关于着色器和OpenGL管道的内容，并发现OpenGL ES 2.0的魔力，相比前一个版本，它有很多更加丰富的功能特性。如果你能用Java编程，并且愿意与他人分享你的视觉创意，这本书就是为你而写。

要在Android上创建引人注目的图形，这本书将讲解你所需要的所有知识。通过构建一个简单的空气曲棍球游戏，你会学习到OpenGL的基础内容，在这个开发过程中，你会看到如何初始化OpenGL，以及使用着色器编写图形管道。每一节都是在前一节的基础上构建的，一步步地添加颜色、着色器、三维投影、触控交互等。

然后，你会知道如何把你的想法变成可以在主屏幕上运行的动态壁纸。你将学到更多高级的特效，包括粒子、光照模型和深度缓冲区。当调试程序的时候，你会知道需要从何处下手；当把应用部署到市场上的时候，你会明白需要注意哪些问题。

对新手来说，OpenGL可能有点黑暗艺术的意味。当你阅读这本书时，你将从最基本的原理开始学习每一个新概念。你不仅仅只学会了一个功能特性；你也会理解它是如何工作的，以及为何它的工作方式是那样的。这里所介绍的技术也可应用于其他平台，比如iOS或HTML5 WebGL。



说明：黄色表示已出版

上架指导：计算机/移动开发
ISBN 978-7-111-47782-2
9 787111 477822 >
定价：49.00元

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259
华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

移动开发

OpenGL ES 2 for Android： A Quick-Start Guide

OpenGL ES 应用开发实践指南

Android 卷

（美） Kevin Brothaler 著　刘力平 苏统华◎译

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：Android 卷 /（美）博泽勒尔（Brothaler，K.）著；刘力平，苏统华译．—北京：机械工业出版社，2014.9
（移动开发）
书名原文：OpenGL ES 2 for Android

ISBN 978-7-111-47782-2

I. O… II. ①博… ②刘… ③苏… III. ①图形软件－指南 ②移动终端－应用程序－程序设计－指南 IV. ① TP391.41-62 ② TN929.53-62

中国版本图书馆 CIP 数据核字（2014）第 198717 号

本书版权登记号：图字：01-2014-0324

Kevin Brothaler: *OpenGL ES 2 for Android* (ISBN 978-1-937785-34-5).

Copyright © 2013 The Pragmatic Programmers, LLC.

Simplified Chinese Translation Copyright © 2014 by China Machine Press.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or any information storage and retrieval system, without permission, in writing, from the publisher.

All rights reserved.

本书中文简体字版由 The Pragmatic Programmers, LLC 授权机械工业出版社在全球独家出版发行。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。

OpenGL ES 应用开发实践指南：Android 卷

出版发行：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 责任校对：殷 虹
印 刷：冀城市京瑞印刷有限公司 版 次：2014 年 9 月第 1 版第 1 次印刷
开 本：186mm×240mm 1/16 印 张：15
书 号：ISBN 978-7-111-47782-2 定 价：49.00 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

版权所有 · 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

Foreword 推荐序

游戏是视觉体验。作为游戏开发者，我们要创造出特定的环境和角色，以使玩家能沉浸其中，这就需要使用极其逼真的三维场景或者离奇的、超出现实世界的体验。我们渴望数百万人玩我们的游戏并体验我们的世界，为他们的生活多增加一点儿乐趣。Android 和它的生态系统为我们提供了观众；OpenGL ES 给了我们实现那些梦想中的游戏的技术手段。

OpenGL ES 是移动图形编程方面的事实标准。它是 OpenGL 桌面版本的精简的兄弟，为了保持向后兼容性，它去除了大量冗余的东西。OpenGL ES 有三个主要的版本：版本 1.0 提供了一个不灵活的、固定功能的管道；版本 2.0 推出了一个可编程的管道，这个管道可以提供我们需要的所有功能；最终，诞生了非常年轻、还没有广泛使用的版本 3.0，它在 2.0 标准的基础上添加了新的特性。

虽然 OpenGL ES 几乎被使用 10 年了，但方便初学者查阅的材料还很难找到，尤其是版本 2.0，它推出了一个对新人来说很深奥的主题——着色器。除了设备和平台相关的问题，材料的缺乏设置了相当高的准入门槛。

基于这些原因 Kevin 写作了本书。他削减了所有的陈词滥调，带你浏览了 OpenGL ES 2.0 的实务。Kevin 不仅囊括了支持三维图形的基本的概念，而且记录了你可能遇到的所有特定于 Android 的陷阱；有经验的开发人员在这也能找到几块金子。换句话说："独自上路很危险，带上这本书吧！"。

Mario Zechner[⊖]

⊖ Libgdx 的创建者（http://code.google.com/p/libgdx/），《Beginning Android Games》[Zec 12] 一书的作者。

致 谢 *Acknowledgements*

我非常感谢 Pragmatic Programmers 这个出色的团队给我写这本书的机会，并做了如此伟大的工作帮助我把它完成。在我第一次要开始时，我不确信期望的是什么，但他们做了很细致的工作教会我如何掌握。深深地感谢编辑，Susannah Pfalzer 在整个过程中始终给我非常专业的指导，感谢 Aron Hsiao 帮助我确定了本书务实的写作风格。

我也感谢所有的技术审校者和提供了有价值的反馈的人，包括（排名不分先后）Mario Zechner、Owen Alanzo Hogarth、Sam Rose、Mike Riley、Aaron Kalair、Rene Van der Lende、John Horton、Ed Burnette、Mark Guerra、Maik Schmidt、Kevin Gisi、Brian Schau、Marius Marinescu、Stephen Wolff、Haress Das、Bill Yee、Chad Dumler-Montplaisir、Tibor Simic、Michael Hunter、Jonathan Mischo 和 Stefan Turalski，以及每一个报告勘误或发回反馈的人。你们的反馈和建议为提高本书品质提供了极大的帮助，同时非常感谢你们的鼓励。

如果没有庞大社区的支持和那些愿意分享知识者的慷慨，我无法完成这本书，包括那些看过我的《Learn OpenGL ES》的忠实读者、Khronos 工作组的人、NeHe Productions、Android 团队、John Carmack 和所有可能遗漏的其他巨人，他们数量多到难以记住。我站在他们的肩膀上，没有他们，本书是不可能出版的。

也许我最应该感激我生命中的两个女人，她们让我一直坚持：我的 Oma，让我知道，一点点韧性可以走很长的路；Anne——我的未婚妻，允许我花费大量的业余时间创作本书，并自始至终一直鼓励我。

最后，谢谢你，亲爱的读者，因为你决定拾起这本书并开始阅读。希望本书能使你受益匪浅，并在未来的旅程中能满足你的需要。

Preface 前 言

Android 刚刚经历了一个难以置信的增长周期，全世界的消费者手里有超过 7.5 亿台设备，并且每天都有一百多万台设备被激活[⊖]。和 Apple 一样，在每一台 Android 手机和平板上也有个集中的市场，称为 Google Play。随着这个市场被安装到每台 Android 设备上，对任何一个梦想着发布自己的游戏或者壁纸的人来说，这都是个前所未有的机会。

在 Android 上，与 Apple 的 iOS 及许多其他移动平台一样，开发者可以使用一个跨平台应用编程接口创建二维或者三维图形，这个接口称为 OpenGL；OpenGL 已经用于桌面系统很长时间了，但移动平台上使用的是一个特殊的嵌入式版本，叫做 OpenGL ES。第一个版本的 OpenGL ES 把三维带进了移动平台，它在开发者中非常流行，因为它非常简单易学，同时还包含了一个定义好的特性集合。然而，这个特性集合也很有限，它无法跟上大多数高性能智能手机和平板电脑所提供的先进的、卓越的特性。

发展到了 OpenGL ES 2.0，大多数旧的 API 被淘汰了，取而代之的是新的可编程 API，它更容易增加特殊的效果，也能利用最新的硬件所提供的优势。这些设备现在也能生成其他竞争平台几年前就实现的图形了！然而，为了利用这个能力，我们需要学习 2.0 引进的这些新的 API。在 2012 年 8 月，Khronos 工作组最终确定了下一版本的规范，即 OpenGL ES 3.0，它完全兼容 2.0，只是在其基础上扩展了一些先进的特性。

因此，Android 上的 OpenGL 到底能做什么呢？我们可以创建令人惊艳的动态壁纸（live wallpaper），这些壁纸可以被数百万的用户下载；我们可以创建有竞争力的三维游戏，它们有着栩栩如生的、令人窒息的图形效果。随着硬件成本的降低，以及在线商店的大量普及，是时候开始学习了！

⊖ http://googleblog.blogspot.ca/2013/03/update-from-ceo.html

本书的主要内容

快速浏览一下我们将要讨论的内容：

- 在本书第一部分，会学习如何创建一个简单的空气曲棍球游戏，包括触控、纹理和基本原理。这个项目会教你如何成功地初始化 OpenGL 并将数据发送到屏幕上，以及如何使用基本的向量和矩阵数学创建三维世界。你也会学到 Android 的许多特定细节，比如在 Dalvik 虚拟机和本地环境之间编码数据（marshal data）以及如何在主线程和渲染线程间安全地传送数据。
- 在本书第二部分，你就可以在第一部分学习到的知识的基础上开始搭建三维世界中的效果，比如光照和地形渲染，接下来还会学习如何创建可以运行在 Android 主屏幕（home screen）上的动态壁纸。

本书的读者对象

如果对在 Android 上开发更高级的图形程序感兴趣，本书就是为你所写的。本书假定你有一些编程经验，包括 Java 和 Android 的经验。

Java

如果你以前用过其他托管语言（managed language），比如 C#，转到 Java 就很简单了。如果你有更多关于本地语言（native language）的经验，你就要注意它们的主要区别之一，Android 上的 Java 是运行在虚拟机上的、使用垃圾回收的语言，这既有益处也有代价。

下面这些书籍会更好地帮助你学习 Java：

- 《The Java Programming Language》[AGH05]，作者：Ken Arnold、James Gosling 和 David Holmes
- 《Effective Java》[Blo 08]，作者：Joshua Bloch[⊖]
- 《Thinking in Java》[Eck 06]，作者：Bruce Eckel[⊜]

Android

一旦你熟悉了 Java，要为 Android 开发软件就需要一些相关的库和方法的经验了。为了覆盖所有的基本知识，我推荐阅读 Ed Burnette 的书籍《Hello，Android》[Bur 10]。你也可以看看 Google 的 Android 在线培训中的前两节课：

⊖ 本书中文版已由机械工业出版社引进出版，书号为 978-7-111-25583-3。

⊜ 本书中文版已由机械工业出版社引进出版，书号为 978-7-111-21382-6。

- 构建第一个应用[㊀]
- 管理 Activity 组件的生命周期[㊁]

对于本书的大部分内容，尽管使用模拟器就足够了，但手上有个 Android 设备会更容易些。稍后我们会给出更多细节，参见 1.1 节。

这些应该足够让你通读本书了。我们会从第一个基本原理开始就包括所有的基础知识，这样，你就不需要任何三维图形的编程经验了，并且我们也会在书中讲到一些数学知识，如果你过去学过三角学和线性代数，那你就准备得很充分了！如果没有，不用害怕，讲解过程中，所有的东西都会详细解释的。

如何阅读本书

每一章都是在前一章的基础上讲解的，因此，最好按顺序阅读。所有的示例代码都可以在线下载（参见下面的“在线资源”一节），这样，当你想查看某个特定章节时，就可以下载前一章的整个项目，以此作为基础开始；如果你忘了或者想要从头开始，这会有帮助。

惯例

我们会使用 OpenGL 代指 OpenGL ES 2.0，它是移动平台和网页上的 OpenGL 现代的版本。

在本书的大部分章节中，我们使用 GLES20 类，它是 Android 软件开发包（SDK）的一部分。因为大多数的 OpenGL 常量和方法都在这个类里定义，我一般会忽略类名，而直接提及常量或方法；我们也会在代码中使用静态导入（static import），从而忽略类名（参见 1.5 节）。

在线资源

本书的所有资源都可以在网站 http://pragprog.com/book/kbogla 找到，包括示例代码以及相应的图像和纹理。

你可以访问 Learn OpenGL ES 网站，那有我维护的一个 OpenGL ES 教程的博客[㊂]。

下面是一些很好的在线资源列表，它们由 Khronos 工作组维护[㊃]：

- OpenGL ES 2.0 API 快速参考卡片[㊄]

㊀ http://developer.android.com/training/basics/firstapp/index.html

㊁ http://developer.android.com/training/basics/activity-lifecycle/index.html

㊂ http://www.learnopengles.com/

㊃ http://www.khronos.org/

㊄ http://www.khronos.org/opengles/sdk/docs/reference_cards/OpenGL-ES-2_0-Reference-card.pdf

- OpenGL ES 2.0 参考页[㊀]
- OpenGL ES 着色器语言 (GLSL ES) 参考页[㊁]
- The OpenGL ES 着色器语言[㊂]
- OpenGL ES Common Profile Specification 版本 2.0.25（完整规范）[㊃]

我推荐把参考卡片打印出来放在手边，这样就能在需要时快速参考。Android 使用 EGL（一个本地平台接口）帮助你配置显示设备，你可能发现了 Khronos 的 EGL API 的注册表也很有用[㊄]。

现在开始吧

如今拥有功能强大的手机和平板电脑的人比以前多了，并且市场还在持续增长。Android 的软件工具使得开发者更容易为 Android 开发软件，并且 Google 的 Play Store 使分享应用变得更加容易。让我们向第 1 章前进吧。船起锚了！

㊀ http://www.khronos.org/opengles/sdk/docs/man/

㊁ http://www.khronos.org/opengles/sdk/docs/manglsl/

㊂ http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf

㊃ http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf

㊄ http://www.khronos.org/registry/egl/

Contents 目 录

推荐序

致 谢

前 言

第1章 准备开始 ····· 1

1.1 安装工具 ····· 1

1.2 创建第一个程序 ····· 3

1.3 初始化 OpenGL ····· 5

1.4 创建 Renderer 类 ····· 10

1.5 使用静态导入 ····· 12

1.6 小结 ····· 12

第一部分 一个简单的空气曲棍球游戏

第2章 定义顶点和着色器 ····· 14

2.1 为什么选择空气曲棍球 ····· 14

2.2 不要从头开始 ····· 15

2.3 定义空气曲棍球桌子的结构 ····· 16

2.4 使数据可以被 OpenGL 存取 ····· 18

2.5 引入 OpenGL 管道 ····· 20

2.6 OpenGL 颜色模型 ····· 25

2.7 小结 ····· 26

第3章 编译着色器及在屏幕上绘图 ····· 27

3.1 加载着色器 ····· 27

3.2 编译着色器 ····· 29

3.3 把着色器一起链接进 OpenGL 的程序 ····· 33

3.4 做最后的拼接 ····· 35

3.5 在屏幕上绘制 ····· 39

3.6 小结 ····· 43

3.7 练习 ····· 43

第4章 增加颜色和着色 ····· 44

4.1 平滑着色 ····· 45

4.2 引入三角形扇 ····· 45

4.3 增加一个新的颜色属性 ····· 46

4.4 使用新的颜色属性渲染 ····· 52

4.5 小结 ····· 55

4.6 练习 ····· 55

第5章 调整屏幕的宽高比 ····· 56

5.1 宽高比的问题 ····· 57

5.2 使用虚拟坐标空间 …… 58
5.3 线性代数基础 …… 59
5.4 定义正交投影 …… 62
5.5 加入正交投影 …… 64
5.6 小结 …… 67
5.7 练习 …… 67

第6章 进入第三维 …… 68

6.1 三维的艺术 …… 68
6.2 从着色器到屏幕的坐标变换 …… 69
6.3 添加 w 分量创建三维图 …… 71
6.4 使用透视投影 …… 72
6.5 定义透视投影 …… 73
6.6 在代码中创建投影矩阵 …… 75
6.7 开始使用投影矩阵 …… 76
6.8 增加旋转 …… 79
6.9 小结 …… 81
6.10 练习 …… 82

第7章 用纹理增加细节 …… 83

7.1 理解纹理 …… 84
7.2 把纹理加载进 OpenGL 中 …… 85
7.3 创建新的着色器集合 …… 90
7.4 为顶点数据创建新的类结构 …… 91
7.5 为着色器程序添加类 …… 95
7.6 绘制纹理 …… 99
7.7 小结 …… 101
7.8 练习 …… 101

第8章 构建简单物体 …… 102

8.1 合并三角形带和三角形扇 …… 102
8.2 添加几何图形的类 …… 104
8.3 添加物体构建器 …… 105
8.4 更新物体 …… 112
8.5 更新着色器 …… 113
8.6 集成所有的变化 …… 114
8.7 小结 …… 119
8.8 练习 …… 119

第9章 增加触控反馈：与空气曲棍球游戏交互 …… 120

9.1 为 Activity 增加触控支持 …… 120
9.2 增加相交测试 …… 122
9.3 通过拖动移动物体 …… 130
9.4 增加碰撞检测 …… 132
9.5 小结 …… 136
9.6 练习 …… 137

第二部分 构建三维世界

第10章 用粒子增添趣味 …… 140

10.1 为一个简单的粒子系统创建一组着色器 …… 141
10.2 添加粒子系统 …… 144
10.3 绘制粒子系统 …… 148
10.4 扩散粒子 …… 150
10.5 添加重力 …… 152
10.6 用累加混合技术混合粒子 …… 153
10.7 自定义点的外形 …… 153
10.8 把每个点绘制为一个精灵 …… 154
10.9 小结 …… 156
10.10 练习 …… 156

第11章 增加天空盒 …… 157

11.1 创建天空盒 …… 157

11.2 把立方体贴图加载到 OpenGL 中 …… 159
11.3 创建立方体 …… 161
11.4 增加天空盒着色器程序 …… 163
11.5 在场景中加入天空盒 …… 165
11.6 围绕场景移动相机 …… 167
11.7 小结 …… 169
11.8 练习 …… 169

第12章 增加地形 …… 170

12.1 创建高度图 …… 170
12.2 创建顶点和索引缓冲区对象 …… 171
12.3 加载高度图 …… 173
12.4 绘制高度图 …… 176
12.5 遮罩被隐藏的物体 …… 180
12.6 小结 …… 183
12.7 练习 …… 183

第13章 点亮世界 …… 184

13.1 仿真光照的效果 …… 184
13.2 用朗伯体反射实现方向光 …… 186
13.3 添加点光 …… 192
13.4 小结 …… 199
13.5 练习 …… 199

第14章 创建动态壁纸 …… 200

14.1 实现动态壁纸的 Service 组件 …… 200
14.2 权衡系统的其余部分 …… 207
14.3 小结 …… 209
14.4 练习 …… 209

第15章 采取下一步行动 …… 211

15.1 展望其他平台 …… 211
15.2 了解更高级的技术 …… 213
15.3 与世界分享你的艺术视野 …… 216

附录A 投影背后的矩阵数学 …… 217

附录B 调试 …… 223

参考文献 …… 228

第 1 章 Chapter 1

准备开始

在本章里，我们会直奔主题，创建我们的第一个 Android OpenGL 应用程序。在阅读本书每一章的过程中，你会发现，每一章都会以概述作为开始，然后浏览“游戏计划”，也就是那一章的开发计划。下面就是本章的“游戏计划”：

- 首先，我们会安装和配置开发环境；
- 然后，创建我们的第一个 OpenGL 应用程序，它将初始化 OpenGL，并且处理 Android 的 Activity 组件的生命周期；关于生命周期，后文会做详细介绍。

这些将作为我们需要在屏幕上绘制东西时的基础。

准备好了？出发！

1.1 安装工具

下面是开发 Android OpenGL 所需要的基本工具列表：

- 一台运行 Windows、OS X 或者 Linux 的计算机
- Java 开发包（JDK）
- Android 软件开发包（SDK）
- 集成开发环境（IDE）
- 一个手机、平板电脑或者支持 OpenGL ES 2.0 的模拟器

你需要的第一个工具是一台适合开发的个人计算机；任何较新的计算机都可以，它需要能运行 Windows、OS X 或者 Linux。在这台计算机上，你需要安装 Java 开发包，可以从

Oracle 的网站下载[㊀]；当前，Google 指定 JDK 6 作为 Android 开发的版本，但后面的版本默认情况下应该也可以工作，万一出现问题，要检查编译器的兼容版本是否设置成了 1.6。

你还需要安装 Android 软件开发包，可以从 Android 开发者网站下载[㊁]；这个开发包包括 Android 开发所需要的一切工具，包括支持 OpenGL ES 2.0 的模拟器和集成开发环境；一旦你的计算机已经安装完了 JDK，你就可以继续把 Android SDK 解压到任何选择的文件夹中。

Android SDK 包里附带了 Eclipse，这是一个流行的集成开发环境，也是 Google 的 Android 开发工具集（ADT）官方支持的 IDE[㊂]。本书会一直使用 Eclipse，不过，如果你想使用其他的开发环境，IntelliJ 的 IDEA 社区版（Community Edition）是个不错的选择；Google 最近发布了 Android Studio，它是 IntelliJ 的一个分支，包括了专为支持 Android 开发的新工具和功能[㊃][㊄]。

1.1.1 配置新的模拟器

既然工具安装完成了，就可以使用 Android 虚拟设备（AVD）管理器创建新的虚拟设备。

1. 进入安装 Android SDK 的文件夹。如果是 Windows 系统，运行“SDK Manager.exe”打开 Android SDK 管理器；如果是其他操作系统，运行“sdk/tools/android”。
2. 选择菜单“Tools → Manage AVDs”，打开 Android 虚拟设备管理器。
3. 选择“New”，弹出“Create new Android Virtual Device(AVD)”对话框。
4. 选择“Galaxy Nexus”作为设备（device）。
5. 勾选“Use Host GPU（graphics procession unit）”旁边的复选框。
6. 给虚拟设备命名，其他设置保留默认值；窗口看起来如图 1-1 所示。
7. 单击“OK”按钮，创建新的模拟器映像，如图 1-1 所示。

图 1-1 创建新的 Android 虚拟设备

㊀ www.oracle.com/technetwork/java/javase/downloads/index.html

㊁ http://developer.android.com/sdk/index.html

㊂ ADT 是 Eclipse 中的一个插件。——译者注

㊃ http://www.jetbrains.com/idea/

㊄ http://developer.android.com/sdk/installing/studio.html

现在你可以关闭 AVD 和 SDK 管理器。

使用 x86 模拟器

如果你的计算机支持硬件虚拟化，你可能想尝试一下 x86 模拟器。你需要先下载 Intel x86 Atom 系统映像，它可以从 SDK 管理器中最新的 Android SDK 版本里得到；你还要安装 Intel 的硬件加速执行管理器，展开“Extras”选项，在最底端就能发现那个管理器。

一旦安装完这些软件包，下一步就是配置硬件加速执行管理器。你需要运行一个安装器，它在 Android SDK 安装目录下的“etras/intel/Hardware_Accelerated_Execution_Manager”，运行那个文件夹中的可执行文件，并遵照其安装指导即可。你可能需要确保计算机 BIOS 系统设置里的“Intel Virtualization Technology (VT-x)”选项是使能的。

现在，你只需要按照 1.1.1 节中所描述的步骤配置这个模拟器，只是这次要选择 x86 模拟器而不是 ARM 模拟器。在 Android 开发网站上可以得到更多关于虚拟机（VM）加速的指导[⊖]。

1.1.2 使用模拟器

可以使用 AVD 管理器启动一个模拟器实例，或者，也可以让 IDE 在需要时自动启动；如果系统里没有其他模拟器运行，Eclipse 中的 Android 开发工具插件会启动那个模拟器。让一个模拟器实例持续运行是个不错的主意，这样你就不必每次都等待它启动了。

1.1.3 获取支持 OpenGL ES 2.0 的设备

可以使用模拟器，但是最好有一台真实的设备，因为模拟器不能准确反映真实的性能和结果，而且，即使你的计算机硬件性能很好，模拟器也会显得很慢。Nexus 7 是个很好但不贵的选择，可以从 Google Play 在线购买[⊖]。

1.2 创建第一个程序

我们已经安装并配置了工具，现在来创建第一个 OpenGL 程序。这个程序非常简单：它要做的只是初始化 OpenGL 并不停地清空屏幕；要得到一个可以实际做点什么的 OpenGL 程序，这是我们需要的最少的内容。

⊖ http://developer.android.com/tools/devices/emulator.html#accel-vm

⊖ https://play.google.com/store/devices

Joe 问：为什么要不停地清空屏幕

如果已经在每一帧的屏幕上都绘制了内容，清空屏幕好像是浪费的，那为什么还要这样做呢？

回到一切都用软件渲染的时代，清空屏幕总是一种浪费；开发者总是要假定所有东西都已经绘制好了，并能覆盖前一帧的内容，而不必清空它，这样就能达到优化的目的；可以节省清空屏幕的处理时间。有时，这种优化会导致一些游戏中常见的著名的“镜子大厅”（Hall Of Mirrors）效应，如 Doom：它导致的视觉效果就像站在镜子大厅中间，旧的内容一遍又一遍重复出现[⊖]。

这种优化不再有效了；最新的 GPU 以不同的方式工作，他们使用特殊的渲染技术，如果屏幕是干净的，他们能工作得更快。通过让 GPU 清空屏幕，可以节省帧拷贝浪费的时间。因为 GPU 的这种工作方式，清空屏幕可以帮助避免很多问题，如闪烁或者有物品没有绘制；保留旧的内容容易看到不期望或者不理想的结果。

通过下面的链接，可以学到更多的内容：

- http://developer.amd.com/gpu_assets/gdc2008_ribble_maurice_TileBasedGpus.pdf
- http://www.beyond3d.com/content/articles/38/

如果你想查看源代码，这本书的所有源代码和相关数据都可以从本书的主页上下载[⊖]。

新建项目

接下来按照下面的步骤新建项目：

1. 在 Eclipse 菜单中选择“File → New → Android Application Project”，当对话框弹出时，输入如下信息。

应用程序名（Application Name）：

输入“First OpenGL Project”。

包名（Package Name）：

包名是一个项目独一无二的标识符；按照惯例，要输入一个 Java 样式的包名，因此，让我们输入“com.firstopenglproject.android”。

最低 SDK 版本（Minimum SDK）：

选择“API 10: Android 2.3.3 (Gingerbread)”；这是完全支持 OpenGL ES 2.0 的最低版本号。

2. 其余选项使用默认值，那个表格看起来应该与图 1-2 类似。

⊖ https://en.wikipedia. org/wiki/Noclip_mode

⊖ http://pragprog.com/book/kbogla

3. 单击“Next”按钮，不要勾选“Create custom launcher icon”，并且确认勾选了“Create Activity”；你可以为这个项目选择一个不同的目录。
4. 再单击“Next”按钮就到了创建 Activity 的界面；要保证“Blank Activity”被选中了，再单击“Next”按钮，新的“Blank Activity”配置界面就出现了；设置 Activity 名字为“FirstOpenGLProjectActivity”；你的界面看上去应该与图 1-3 相似。
5. 单击“Finish”按钮，继续并构建这个项目。

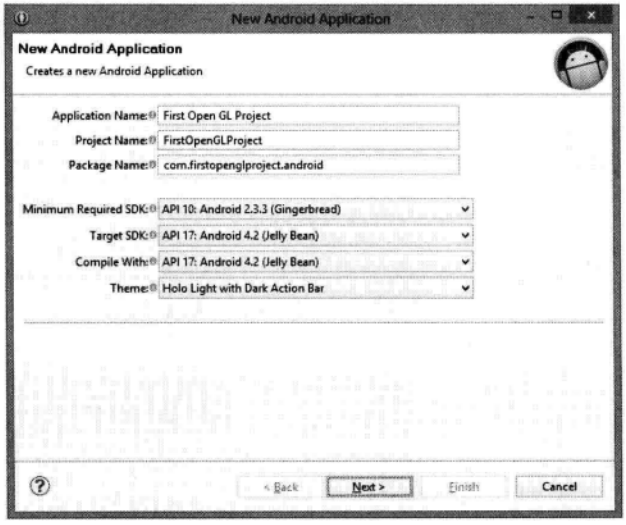


图 1-2 在 Eclipse 中创建新的 Android 项目

当单击“Finish”按钮后，Eclipse 会工作一会儿，然后新项目就创建好了。

1.3 初始化 OpenGL

下一步就是使用一个特殊的类 GLSurfaceView 初始化 OpenGL。GLSurfaceView 会处理 OpenGL 初始化过程中比较基本的操作，如配置显示设备（display）以及在后台线程中渲染；渲染是在显示设备中一个称为“surface”的特定区域完成的，有时也称为视口（viewport）。

GLSurfaceView 类也使得处理标准 Android Activity 生命周期变得容易了；在 Android 里，任何 Activity 都可以创建（create）和销毁（destroy），并且在用户切换到其他 Activity 时暂停（pause），在切换回来时继续（resume）。根据这个生命周期，在 Activity 被暂停的时候，我们需要释放 OpenGL 的资源；GLSurfaceView 为此提供了很多辅助方法。

在 EDBurnette 所写的《Hello, Android》里，你可以学到更多关于 Activity 生命周期的内容。

图 1-3 创建新的 Android 项目：配置 Activity

纹理视图

在幕后，GLSurfaceView 实际上为它自己创建了一个窗口（window），并在视图层次（View Hierarchy）上穿了个“洞”，让底层的 OpenGL surface 显示出来。对于大多数使用情况，这就足够了；但是，GLSurfaceview 与常规视图（view）不同，它没有动画或者变形特效，因为 GLSurfaceView 是窗口（window）的一部分。

从 Android 4.0 Ice Cream Sandwich 开始，Android 提供了一个纹理视图（TextureView），它可以渲染 OpenGL 而不用创建单独的窗口或打洞了，这就意味着，这个视图像一个常规窗口一样，可以被操作，且有动画和变形特效。但是，TextureView 类没有内置 OpenGL 初始化操作，要想使用 TextureView，一种方法是执行自定义的 OpenGL 初始化，并在 TextureView 上运行，另外一种方法是把 GLSurfaceView 的源代码拿出来，把它适配到 TextureView 上。[⊖]

1.3.1 创建 GLSurfaceView 实例

打开前面自动生成的 Activity 类 FirstOpenGLProjectActivity。按下“Ctrl+Shift+T”快

⊖ 更多的信息参见 https://groups.google.com/d/msg/android-developers/U5RXFGpAHPE/IqHeIeGXhr0J 和 http://stackoverflow.com/q/12061419。

捷键，“Open Type”对话框就会弹出来，然后输入“FirstOpenGLProjectActivity”，当那个类显示出来时，就选择它。

Eclipse 键盘快捷键

在 Mac 或者 Linux 平台上，快捷键可能与 Windows 平台不同；比如，快捷键“Ctrl+Shift+O”，在 Windows 平台上，它是用来组织和查找新的 Java 导入（import）的；在 Mac 平台上就是⌘⇧O。可以通过菜单“Help → Key Assist”选择按键辅助功能查看各个平台的键绑定。

现在看起来没有多少代码：

FirstOpenGLProject/src/com/firstopenglproject/android/FirstOpenGLProjectActivity.java

```
package com.firstopenglproject.android;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
public class FirstOpenGLProjectActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_first_open_glproject);
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_first_open_glproject, menu);
        return true;
    }
}
```

我们给这个 Activity 添加一个 GLSurfaceView 实例，这样就可以初始化 OpenGL。让我们在类的顶部加入如下两个新的成员变量：

FirstOpenGLProject/src/com/firstopenglproject/android/FirstOpenGLProjectActivity.java

```
public class FirstOpenGLProjectActivity extends Activity {
    private GLSurfaceView glSurfaceView;
    private boolean rendererSet = false;
```

我们需要导入 GLSurfaceView 类，因此，按“Ctrl+Shift+O”快捷键组织导入，并把这个新类放到代码里；每次加入一个新的类引用，并且它需要被导入时，我们都要做这一步。我们将用 rendererSet 记住 GLSurfaceView 是否处于有效状态。让我们从代码中移除 setContentView() 的调用，并加入如下代码初始化 glSurfaceView：

FirstOpenGLProject/src/com/firstopenglproject/android/FirstOpenGLProjectActivity.java

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    glSurfaceView = new GLSurfaceView(this);
```

1.3.2 检查系统是否支持 OpenGL ES 2.0

因为我们只为2.0版本写代码，我们要做的下一件事就是检查系统是否实际支持OpenGL ES 2.0。让我们添加如下几行代码到 onCreate() 函数中：

FirstOpenGLProject/src/com/firstopenglproject/android/FirstOpenGLProjectActivity.java

```
final ActivityManager activityManager =
    (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);

final ConfigurationInfo configurationInfo =
    activityManager.getDeviceConfigurationInfo();

final boolean supportsEs2 = configurationInfo.reqGlEsVersion >= 0x20000;
```

首先，我们需要 Android ActivityManager 的一个引用，用它获取设备配置信息，然后，取出 reqGlEsVersion 变量检查 OpenGL ES 版本号。如果版本号为 0×20000 或后续版本，我们就可以使用 OpenGL ES 2.0 的 API 了。

但是这段代码实际上不能在模拟器上工作，因为 GPU 模拟部分有缺陷；为了使代码在模拟器上正常工作，我们要按如下代码修改那个检查条件：

FirstOpenGLProject/src/com/firstopenglproject/android/FirstOpenGLProjectActivity.java

```
final boolean supportsEs2 =
    configurationInfo.reqGlEsVersion >= 0x20000
        || (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH_MR1
         && (Build.FINGERPRINT.startsWith("generic")
          || Build.FINGERPRINT.startsWith("unknown")
          || Build.MODEL.contains("google_sdk")
          || Build.MODEL.contains("Emulator")
          || Build.MODEL.contains("Android SDK built for x86")));
```

这段代码测试当前设备是不是模拟器，如果是，就假定它支持 OpenGL ES 2.0。要确保程序能运行，模拟器一定要配置 OpenGL ES 2.0，参见 1.1.1 节。

1.3.3 为 OpenGL ES 2.0 配置渲染表面

下一步就是配置渲染表面（rendering surface），添加如下几行代码：

```
if (supportsEs2) {
    // Request an OpenGL ES 2.0 compatible context.
    glSurfaceView.setEGLContextClientVersion(2);

    // Assign our renderer.
    glSurfaceView.setRenderer(new FirstOpenGLProjectRenderer());
    rendererSet = true;
} else {
    Toast.makeText(this, "This device does not support OpenGL ES 2.0.",
        Toast.LENGTH_LONG).show();
    return;
}
```

如果设备支持 OpenGL ES 2.0，我们就通过调用 setEGLContextClientVersion(2) 配置这

个 surface 视图；然后调用 setRenderer() 传进自定义 Renderer 类的一个新实例，稍后，我们会看到如何创建这个 Renderer 类，同时这段代码通过设置 rendererSet 为 true 记住渲染器（renderer）已经设置过了。当 surface 创建或者发生变化的时候，以及要绘制一幅新帧时，渲染器都会被 GLSurfaceView 调用。

如果设备不支持 OpenGL ES 2.0，怎么办呢？也可以添加一个支持 OpenGL ES 1.0 的回退的渲染器，但是，这种情形现在已经很少见了，可能不值得付出这样的努力。根据 Android 开发者统计板[⊖]，仅有大约 9% 的设备只支持 OpenGL ES 1.1，这个数字还在随着时间不断缩减。如果一个设备不支持 OpenGL ES 2.0，公开发布的应用应该在这个设备的应用程序市场中被隐藏起来，在 14.1.6 节中，我们会学到这点。

我们需要添加另外一个调用把 GLSurfaceView 加入到这个 activity 中，并把它显示到屏幕上，在 onCreate() 函数结尾处，用如下代码替换旧的调用 setContentView()：

FirstOpenGLProject/src/com/firstopenglproject/android/FirstOpenGLProjectActivity.java

```
setContentView(glSurfaceView);
```

1.3.4 处理 Android Activity 生命周期的事件

我们还需要处理 Android Activity 生命周期的事件，否则，如果用户切换到另外一个应用，应用就会崩溃；加入如下方法让这个 Activity 类变得完整：

FirstOpenGLProject/src/com/firstopenglproject/android/FirstOpenGLProjectActivity.java

```
@Override
protected void onPause() {
    super.onPause();

    if (rendererSet) {
        glSurfaceView.onPause();
    }
}

@Override
protected void onResume() {
    super.onResume();

    if (rendererSet) {
        glSurfaceView.onResume();
    }
}
```

这些方法非常重要，有了它们，这个 surface 视图才能正确暂停并继续后台渲染线程，同时释放和续用 OpenGL 上下文。如果它没有做这些，应用程序可能会崩溃，并被 Android 终止；我们还要保证渲染器也设置了，否则调用这些方法也会引起程序崩溃。

⊖ http://developer.android.com/resources/dashboard/opengl.html

1.4 创建 Renderer 类

现在要定义一个渲染器，以便我们开始清空屏幕。

让我们快速浏览一下渲染器接口定义的方法：

- onSurfaceCreated(GL10 glUnused, EGLConfig config)

  当 Surface 被创建的时候，GLSurfaceView 会调用这个方法；这发生在应用程序第一次运行的时候，并且，当设备被唤醒或者用户从其他 activity 切换回来时，这个方法也可能会被调用。在实践中，这意味着，当应用程序运行时，本方法可能会被调用多次。

- onSurfaceChanged(GL10 glUnused, int width, int height)

  在 Surface 被创建以后，每次 Surface 尺寸变化时，这个方法都会被 GLSurfaceView 调用到。在横屏、竖屏来回切换的时候，Surface 尺寸会发生变化。

- onDrawFrame(GL10 glUnused)

  当绘制一帧时，这个方法会被 GLSurfaceView 调用。在这个方法中，我们一定要绘制一些东西，即使只是清空屏幕；因为，在这个方法返回后，渲染缓冲区会被交换并显示在屏幕上，如果什么都没画，可能会看到糟糕的闪烁效果。

怎么会有一个未被使用的参数类型 GL10 呢？它是 OpenGL ES 1.0 的 API（应用程序编程接口）遗留下来的；如果要编写使用 OpenGL ES 1.0 的渲染器，就要使用这个参数；但是，对于 OpenGL ES 2.0，GLES20 类提供了静态方法来存取。

在后台线程中渲染

GLSurfaceView 会在一个单独的线程中调用渲染器的方法。默认情况下，GLSurfaceView 会以显示设备的刷新频率不断地渲染，当然，它也可以配置为按请求渲染，只需要用 GlSurfaceView.RENDERMODE_WHEN_DIRTY 作为参数调用 GLSurfaceView.setRenderMode() 即可。

既然 Android 的 GLSurfaceView 在后台线程中执行渲染，就必须要小心，只能在这个渲染线程中调用 OpenGL，在 Android 主线程中使用 UI（用户界面）相关的调用；两个线程之间的通信可以用如下方法：在主线程中的 GLSurfaceView 实例可以调用 queueEvent() 方法传递一个 Runnable 给后台渲染线程，渲染线程可以调用 Activity 的 runOnUIThread() 来传递事件（event）给主线程。

新建渲染器

在同一个包里创建一个各为 FirstOpenGLProjectRender 的新类，用来实现 Renderer 接口；为了创建这个新类，在包浏览器（package Explorer）里，鼠标右键单击“com.firstopenglproject.android”，接着选择“New → Class”；在弹出的“New Java Class”窗口

中输入“FirstOpenGLProjectRenderer”作为名字，并单击“Finish”按钮。

我们会以下面的代码作为开始，并加入我们的第一个方法 OnSurfaceCreated()：

FirstOpenGLProject/src/com/firstopenglproject/android/FirstOpenGLProjectRenderer.java

```
package com.firstopenglproject.android;

import static android.opengl.GLES20.GL_COLOR_BUFFER_BIT;
import static android.opengl.GLES20.glClear;
import static android.opengl.GLES20.glClearColor;
import static android.opengl.GLES20.glViewport;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class FirstOpenGLProjectRenderer implements Renderer {
    @Override
    public void onSurfaceCreated(GL10 glUnused, EGLConfig config) {
        glClearColor(1.0f, 0.0f, 0.0f, 0.0f);
    }
```

首先，在 onSurfaceCreated() 中调用 glClearColor(1.0f, 0.0f, 0.0f, 0.0f) 设置清空屏幕用的颜色；前三个参数分别对应红色、绿色和蓝色，最后的参数对应一个特殊的分量，称为阿尔法（alpha），它经常用来表示半透明度或透明度。通过把第一个分量设为 1，其余设为 0，我们把红色设置为最大强度，当屏幕被清空时，它就会显示红色。2.6 节会更详细地讨论这个颜色模型。

下一步就是设置视口（viewport）尺寸了，让我们加入如下代码：

FirstOpenGLProject/src/com/firstopenglproject/android/FirstOpenGLProjectRenderer.java

```
@Override
public void onSurfaceChanged(GL10 glUnused, int width, int height) {
    // Set the OpenGL viewport to fill the entire surface.
    glViewport(0, 0, width, height);
}
```

在 onSurfaceChanged() 方法中，调用 glViewPort(0, 0, width, height) 设置了视口尺寸，这就告诉了 OpenGL 可以用来渲染的 surface 的大小。

要用如下代码完成这个渲染器类：

FirstOpenGLProject/src/com/firstopenglproject/android/FirstOpenGLProjectRenderer.java

```
    @Override
    public void onDrawFrame(GL10 glUnused) {
        // Clear the rendering surface.
        glClear(GL_COLOR_BUFFER_BIT);
    }
}
```

在 onDrawFame() 中调用 glClear(GL_COLOR_BUFFER_BIT) 清空屏幕；这会擦除屏幕上的所有颜色，并用之前 glClearColor() 调用定义的颜色填充整个屏幕。

现在测试一下代码，看看会发生什么。按“Ctrl+F11”键运行这个程序。你会看到一个

空白的红色屏幕，如图 1-4 所示。

尝试改变清空的颜色，然后再运行程序看看会发生什么！你应该看到屏幕上的颜色与代码的改动完全匹配。

如果你使用的是模拟器，且程序不工作，而你也检查过模拟器的配置选项“Use Host GPU”已经被勾选了，那就试试在 glSurfaceView.setRenderer() 调用之前加入 glSurfaceView.setEGLConfigChooser（8，8，8，8，16，0）的调用。

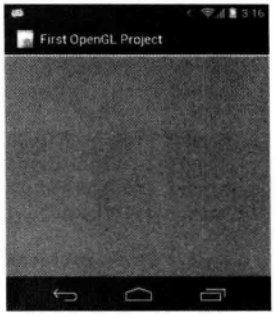


图 1-4 第一个 OpenGL 项目

1.5 使用静态导入

这是我们第一次使用静态导入（static import）指令。我们会在代码中大量地使用它，因为这个指令帮助我们极大地减少了冗长的代码，比如调用 GLE20.glClear (GLE20.GL_COLOR_BUFFER_BIT) 缩减成 glClear(GL_COL_OR_BUFFER_BIT)；当有相当大量的代码使用 OpenGL 和其他实用程序（utility）的时候，这会非常有用。

遗憾的是，Eclipse 对静态导入支持得并不是很好。为了使事情容易些，我推荐你选择菜单“Window → Preferences”，然后选择“Java → Editor → Content Assist → Favorites”，并添加如下类型：

- android.opengl.GLES20
- android.opengl.GLUtils
- android.opengl.Matrix

这对于自动完成代码编写有帮助，但是，这仍然不能修复“组织导入”（Organize Imports）的缺陷。把下面的代码粘贴到类的顶端可以修复它：

```
import static android.opengl.GLES20.*;
import static android.opengl.GLUtils.*;
import static android.opengl.Matrix.*;
```

现在当你组织这些导入时，所有需要的静态导入都会自动被引入；无论何时你需要添加一个对象引用，如果这个对象需要一个新的导入，你就可以容易地修复它，只需到类的顶端，用一个星号（*）替换一个静态导入的结尾字符，并再次运行“组织导入”。

1.6 小结

在本章，我们学习了如何创建一个新的 OpenGL 项目和清空屏幕。我们安装及配置了开发环境、创建了一个新项目、初始化了 OpenGL、响应了 Android Activity 的生命周期，最后清空了屏幕！

我们现在有一个基础了，后面的所有项目都能以此为基础来构建。休息片刻。在下面的两章里，我们会继续在这个基础上构建，学习如何为 GPU 编程，并加入更多特性。准备好了吗？让我们继续学习下一章。

第一部分 *Part 1*

一个简单的空气曲棍球游戏

- 第2章　定义顶点和着色器
- 第3章　编译着色器及在屏幕上绘图
- 第4章　增加颜色和着色
- 第5章　调整屏幕的宽高比
- 第6章　进入第三维
- 第7章　用纹理增加细节
- 第8章　构建简单物体
- 第9章　增加触控反馈：与空气曲棍球游戏交互

Chapter 2 第 2 章

定义顶点和着色器

本章介绍我们的第一个项目：一个简单的空气曲棍球游戏。在我们开发这个项目的过程中，会学习到一些主要的 OpenGL 组件。

作为起始，我们会学习如何使用独立的点集合构建物体，这些点称为顶点；之后，我们会学习怎样使用着色器绘制这些物体，以及告诉 OpenGL 如何绘制这些物体的一些小程序。顶点和着色器这两个概念极其重要，因为每个物体的构建都是通过顶点的聚合形成点、直线和三角形，并且这些基本图形都要使用着色器绘制。

我们会首先学习顶点，这样就可以构建空气曲棍球游戏中用到的桌子，并且使用 OpenGL 的坐标系统定义这个桌子在现实世界中的位置，接下来我们就要创建一套非常基本的着色器在屏幕上绘制这个桌子。在下一章，我们也会学习如何在屏幕上绘制顶点作为点、直线和三角形，并在随后的章节里，学习颜色、平滑着色、纹理和触控交互，以及平行和透视投影。

完成后，我们的空气曲棍球游戏就会如图 2-1 所示。

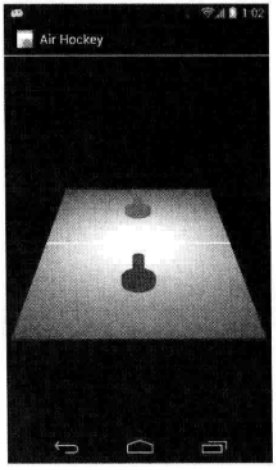


图 2-1　一个简单的空气曲棍球游戏

2.1　为什么选择空气曲棍球

空气曲棍球是一个简单、流行的游戏，经常可以在保龄球馆和酒吧里见到它。尽管简单，但非常容易让人上瘾。在 Android 的应用市场——Google Play 里，一些顶级游戏都是

基于该游戏或者其他令人愉快的游戏的变种。

随着空气曲棍球游戏开发的讲解，我们会学习很多 OpenGL 概念。我们会学习如何定义和绘制一个可以玩游戏的桌子，以及如何添加一些细节，如颜色、着色器和纹理。我们也会学习如何通过执行触屏事件响应用户操作。

游戏规则

要玩空气曲棍球游戏，我们需要一个有两个球门的长方形桌子（每端一个），一个冰球和两个用来击打冰球的木槌。在每个回合开赛前，都把冰球放在桌子中间；每个玩家要尽力把冰球击进对方的球门，同时要防御对方的进攻；第一个射进 7 个球的玩家获胜[⊖]。

作为游戏开发计划的一部分，我们需要做的第一件事情就是要学会如何定义空气曲棍球桌子的结构，以及如何编写可以在屏幕上绘制这张桌子的代码。同时，要构建一个框架作为后续章节的基础。我们暂时让事情简单一点，只定义一个长方形作为桌子，并在桌子中间定义一条分隔线把两边的玩家分开。

我们也需要某种形式用来代表冰球和球门；我们先用独立的点定义它们。在本章结束时，就可以得到所需的结构，只要增加命令就能在屏幕上实际绘制这个桌子了。

2.2　不要从头开始

我们可以重用第 1 章的项目作为起点。

1. 在 Eclipse 里，选择“FirstOpenGLProject”，要确保项目是打开的，然后按“Ctrl+C”快捷键；再按“Ctrl+V”快捷键复制这个项目。
2. 出现提示框时，输入“AirHockey1”作为项目名，你可以自己选择项目目录。
3. 打开新项目，展开“src”文件夹，找到我们在第 1 章中定义的两个类文件：FirstOpenGLProjectActivity.java 和 FirstOpenGLProjectRenderer.java。
4. 重命名我们定义过的这些类，先选择 FirstOpenGLProjectActivity.java，按“Alt+Shift+R”快捷键打开重命名对话框，输入“AirHockeyActivity”（没有 .java 后缀）作为新名字，Eclipse 会自动把后缀附到文件名上；单击“Finish”按钮结束重命名过程。
5. 重复上述步骤把 FirstOpenGLProjectRenderer.java 重命名为“AirHockeyRenderer.java”。
6. 打开“res/values/strings.xml”，把“app_name”定义的字符串值改为“Air Hockey”。
7. 在项目树上展开“src”，并且选择“com.firstopenglproject.android”包，按“Alt+Shift+R”快捷键把它重命名为“com.airhockey.android”。
8. 打开 AndroidManifest.xml，改其包名为“com.airhockey.android”；同时把这个 activity 的属性“android:name”改为“com.airhockey.android.AirHockeyActivity”。

⊖ 详细规则参考 http://en.wikipedia.org/wiki/Air_hockey。——译者注

9. Eclipse 可能把“import com.firstopenglproject.android.R”放在了 AirHockeyActivity 的顶部，并且用下划线标记为错误；如果看到它，就把那行删掉。

我们已经准备好开发这个新项目了。

2.3 定义空气曲棍球桌子的结构

在桌子被绘制到屏幕之前，我们需要告诉 OpenGL 要画什么。开发过程中的第一步是以 OpenGL 能理解的形式定义一个桌子结构。在 OpenGL 里，所有东西的结构都是从一个顶点开始。

2.3.1 介绍顶点

简单来说，一个顶点就是一个代表几何对象的拐角的点，这个点有很多附加属性；最重要的属性就是位置，它代表了这个顶点在空间中的定位。

2.3.2 用顶点构建游戏中的桌子

我们曾说过要暂时保持事物简单，那什么是能表达空气曲棍球桌子结构的最基本的图形呢？我们可以使用长方形。既然一个长方形有 4 个拐角，我们就需要 4 个顶点。长方形是一个二维物体，因此每个顶点都需要一个位置，这个位置在每个维度上都要有一个坐标。

如果我们要把它画到一张图纸上，我们会得到与图 2-2 类似的图形。

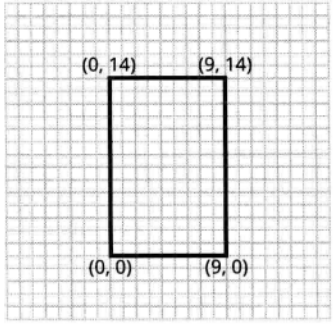


图 2-2 在图纸上绘制一张桌子

2.3.3 在代码中定义顶点

让我们继续，并写一些代码存储这些顶点；这些顶点会表示为一个浮点数列表；因为它们定义在二维坐标系里，所以每个顶点要用两个浮点数进行标记：一个标记 x 轴的位置，另外一个标记 y 轴的位置。

因为一个顶点有两个分量，所以首先创建一个常量用来记住这一事实；打开 AirHockeyRenderer 类，并在它的顶端加入如下常量：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
private static final int POSITION_COMPONENT_COUNT = 2;
```

现在，在 onSurfaceCreated() 的前面加入如下构造函数（constructor）：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
public AirHockeyRenderer() {
            float[] tableVertices = {
```

```
                0f,  0f,
                0f, 14f,
                9f, 14f,
                9f,  0f
            };
}
```

我们采用了浮点数的顺序列表定义顶点数据，因此，我们可以用有小数点的十进制数存储这些位置；这个数组通常称为顶点属性（attribute）数组。到目前为止，只有位置属性储存了，但是，我们稍后也会把颜色及其他属性用同样的概念储存起来。

2.3.4　点、直线及三角形

还记得前面曾说过表示曲棍球桌子的最容易的方式是长方形吗？然而，令人感到混乱的一点是：在 OpenGL 里，只能绘制点、直线以及三角形。

三角形是最基本的几何图形；因为它的结构如此稳定，它随处可见，比如桥梁的结构化构件；它有三条边用来连接它的三个顶点，如果我们拿掉其中一个顶点，剩下的就是一条直线，如果我们再拿掉一个点，就只剩下一个点了。

点和直线可以用于某些效果，但是，只有三角形才能用来构建拥有复杂的对象和纹理的场景。在 OpenGL 里，我们把单独的点放在一个组里构建出三角形，再告诉 OpenGL 如何连接这些点。我们想要构建的所有东西都要用点、直线和三角形定义；如果想构建更复杂的图形，例如拱形，那我们就需要用足够的点拟合这样的曲线。

如果不能使用长方形，我们怎么定义空气曲棍球的桌子呢？事实上，我们可以认为桌子是由两个三角形连接在一起的，如图 2-3 所示。

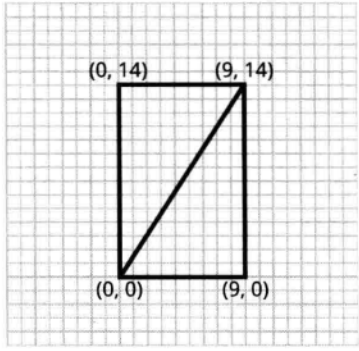


图 2-3　在图纸上绘制一张桌子：两个三角形连接在一起

让我们改变代码以反映这一事实，我们现在使用两个三角形而不是一个长方形：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
float[] tableVerticesWithTriangles = {
        // Triangle 1
        0f,  0f,
        9f, 14f,
        0f, 14f,

        // Triangle 2
        0f,  0f,
        9f,  0f,
        9f, 14f
};
```

三角形的卷曲顺序

你可能注意到了，当我们定义三角形的时候，我们总是以逆时针的顺序排列顶点；这称为卷曲顺序（winding order）。因为在任何地方都使用这种一致的卷曲顺序，可以优化性能：使用卷曲顺序可以指出一个三角形属于任何给定物体的前面或者后面，OpenGL 可以忽略那些无论如何都无法被看到的后面的三角形。

在 12.5 节中我们会学到更多内容。

这个数组表示用六个顶点表示两个三角形；第一个三角形由 (0, 0)、(9, 14) 及 (0, 14) 连接组成，第二个三角形共用了其中的两个点，它由 (0, 0)、(9,0) 及 (9, 14) 构成。

无论何时，如果我们想表示一个 OpenGL 中的物体，都要考虑如何用点、直线及三角形把它组合出来。

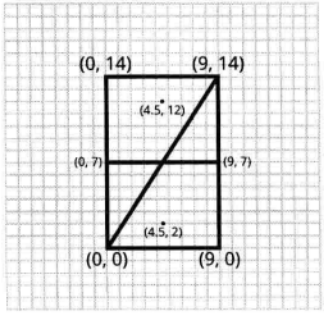


图 2-4 在图纸上绘制一张桌子：加入中间线和两个木槌

2.3.5 添加中间线和两个木槌

我们几乎要完成所有顶点的定义了；我们只需要为中间线和两个木槌添加一些顶点。最后，会得到与图 2-4 相似的图形。

我们将用一条直线定义中间线，并用一个点表示每个木槌。在数组的最后添加一个分号，然后加入下面这些新的顶点：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
// Line 1
0f,  7f,
9f,  7f,

// Mallets
4.5f,  2f,
4.5f, 12f
```

如你所见，由于这个数组由浮点值组合而成，我们可以使用带小数的十进制坐标。为了让 Java 编译器工作，需要在每个数的后面加上小写 f，它告诉 Java 编译器把这个数字解释为浮点数（float），而不是双精度浮点数（double）；双精度浮点数有大约两倍的精度（这是其名字的由来），如果我们不加 f，Java 编译器会把它当作精度丢失的转换，会要求增加显式的类型转换（cast）。

2.4 使数据可以被 OpenGL 存取

我们已经完成顶点的定义了，但是，在 OpenGL 可以存取它们之前，我们仍然需要完成另外一步。主要的问题是这些代码运行的环境与 OpenGL 运行的环境使用了不同的语言，

我们需要理解如下两个主要的概念。

1. 当我们在模拟器或者设备上编译和运行 Java 代码的时候，它并不是直接运行在硬件上的；相反，它运行在一个特殊的环境上，即 Dalvik 虚拟机（Dalvik virtual machine）；运行在虚拟机上的代码不能直接访问本地环境（native environment），除非通过特定的 API。
2. Davik 虚拟机还使用了垃圾回收（garbage collection）机制。这意味着，当虚拟机检测到一个变量、对象或者其他内存片段不再被使用时，就会把这些内存释放掉以备重用；它也能腾挪内存以提高空间使用效率。

本地环境并不是这样工作的，它不期望内存块会被移来移去或者被自动释放。

Android 之所以这样设计，是因为开发者在开发程序的时候不必关心特定的 CPU 或者机器架构，也不必关心底层的内存管理。这通常都能工作得很好，除非要与本地系统交互，比如 OpenGL。OpenGL 作为本地系统库直接运行在硬件上；没有虚拟机，也没有垃圾回收或内存压缩。

2.4.1　从 Java 调用本地代码

Dalvik 方案是 Android 的主要特点之一，但是，如果代码运行在虚拟机内部，那它怎么与 OpenGL 通信呢？有两种技术，第一种技术是使用 Java 本地接口（JNI），这个技术已经由 Android 软件开发包提供了；当调用 android.opengl.GLES20 包里的方法时，软件开发包实际上就是在后台使用 JNI 调用本地系统库的。

2.4.2　把内存从 Java 堆复制到本地堆

第二种技术就是改变内存分配的方式，Java 有一个特殊的类集合，它们可以分配本地内存块，并且把 Java 的数据复制到本地内存。本地内存可以被本地环境存取，而不受垃圾回收器的管控。

我们需要按如图 2-5 所示传输数据；在类的顶部，在构造函数之前加入如下代码：

```
AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java
private static final int BYTES_PER_FLOAT = 4;
private final FloatBuffer vertexData;
```

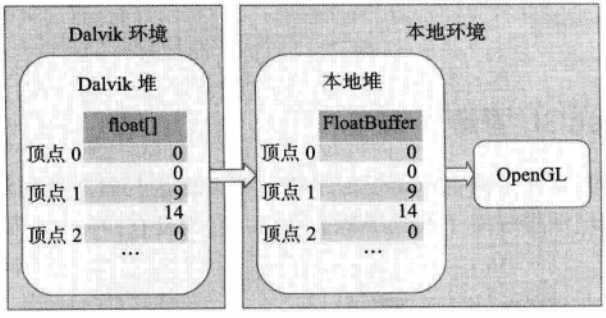


图 2-5　从 Davik 到 OpenGL 传输数据

这里加入了一个整型常量 BYTES_PERFLOAT 和一个 FloatBuffer 类型变量；一个 Java 的浮点数（float）有 32 位（bit）精度，而一个字节（byte）只有 8 位精度；这点可能看起来很明显，每个浮点数都占用 4 个字节；在后面的开发讲解中，很多地方都会提到这点。那个 FloatBuffer 用来在本地内存中存储数据。

让我们加入更多的代码，这次加在构造函数体的结尾处：

```
AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java
vertexData = ByteBuffer
    .allocateDirect(tableVerticesWithTriangles.length * BYTES_PER_FLOAT)
    .order(ByteOrder.nativeOrder())
    .asFloatBuffer();

vertexData.put(tableVerticesWithTriangles);
```

让我们看一下代码的每一个部分。首先，我们使用 ByteBuffer.allocateDirect() 分配了一块本地内存，这块内存不会被垃圾回收器管理。这个方法需要知道要分配多少字节的内存块；因为顶点都存储在一个浮点数组里，并且每个浮点数有 4 个字节，所以这块内存的大小应该是 tableVerticesWithTriangles.length * BYTES_PER_FLOAT。

下一行告诉字节缓冲区（byte buffer）按照本地字节序（native byte order）组织它的内容；本地字节序是指，当一个值占用多个字节时，比如 32 位整型数，字节按照从最重要位到最不重要位或者相反顺序排列；可以认为这与从左到右或者从右到左写一个数类似。知道这个排序并不重要，重要的是作为一个平台要使用同样的排序；调用 order(ByteOrder.nativeOrder()) 可以保证这一点。

最后，我们不愿直接操作单独的字节，而是希望使用浮点数，因此，调用 asFloatBuffer() 得到一个可以反映底层字节的 FloatBuffer 类实例；然后就可以调用 vertexData.put(tableVerticesWithTriangles) 把数据从 Dalvik 的内存复制到本地内存了。当进程结束的时候，这块内存会被释放掉，所以，我们一般情况下不用关心它；但是，如果你在编写代码的时候，创建了很多 ByteBuffer，或者随着程序运行产生了很多 ByteBuffer，你也许想学习一些堆碎片化以及内存管理的技术[⊖]。

为了把数据从 Dalvik 传进 OpenGL，用了这么多步骤，但是在继续讲解之前，理解它是如何工作的是很重要的；就像文化和风俗一样，国与国有很多不同，我们也要知道跨越本地代码边界时的差异。

2.5 引入 OpenGL 管道

现在，我们已经定义了曲棍球桌子的结构，并且把这些数据复制到了 OpenGL 可以存取的本地内存；在把曲棍球桌子画到屏幕上之前，它需要在 OpenGL 的管道（pipeline）中

⊖ http://en.wikipedia.org/wiki/Memory_pool

传递，这就需要使用称为着色器（shader）的子例程（见图 2-6）。这些着色器会告诉图形处理单元（GPU）如何绘制数据。有两种类型的着色器，在绘制任何内容到屏幕之前，需要定义它们。

Joe 问：什么是字节序

字节序（Endianness）是描述一个硬件架构是如何组织位（bit）和字节（byte）的方式，它们在底层组成一个数字。现实中，最常见的就是多字节数，既可以把它们按大头序（big endian order）排列，即把最重要的字节放在前面；或者按小头序（little endian order）排列，即把最不重要的字节放前面。

举个例子，有个十进制数 10 000，如果把它转成二进制数，就是 10011100010000。在大头的架构上，这些位就会排列成：

00100111　00010000

在小头的架构上，它们就会排列为：

00010000　00100111

这次使用十六进制再来看一下。十进制数 10 000 就是十六进制系统中的 2710；因为每两个字符对应一个 8 位字节，在看计算机代码时，十六进制用起来更好些；在大头的架构上，这个数字会另存为：

27 10

而在小头的架构上，同样的数字会另存为：

10 27

正常情况下，我们不需要担心字节序。当使用 ByteBuffer 时，只需要保证它和硬件使用了同样的字节序；否则结果就会出现莫名其妙的错误。在 Wikipedia 上有更多关于字节序的内容[⊖]。

1. 顶点着色器（vertex shader）生成每个顶点的最终位置，针对每个顶点，它都会执行一次；一旦最终位置确定了，OpenGL 就可以把这些可见顶点的集合组装成点、直线以及三角形。
2. 片段着色器（fragment shader）为组成点、直线或者三角形的每个片段生成最终的颜色，针对每个片段，它都会执行一次；一个片段是一个小的、单一颜色的长方形区域，类似于计算机屏幕上的一个像素。

一旦最后的颜色生成了，OpenGL 就会把它们写到一块称为帧缓冲区（frame buffer）的内存块中，然后，Android 会把这个帧缓冲区显示到屏幕上。

⊖ http://en.wikipedia.org/wiki/Endianness

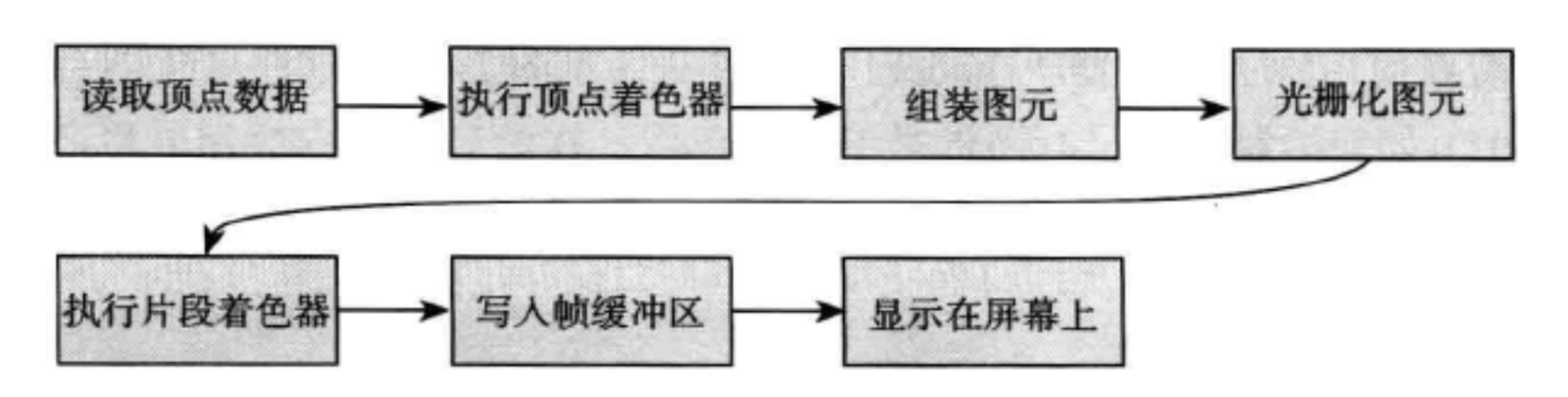


图 2-6 OpenGL 管道概述

Joe 问：为什么使用着色器

在着色器出现之前，OpenGL 只能使用一个固定的方法集合控制很少而有限的事情，比如场景里有多少光线或者加多少雾；这些固定的 API 很容易使用，但是它们很难扩展。你只能实现 API 提供的效果，而且仅此而已；几乎不能添加如卡通着色一样的自定义效果。

随着时间的推移，底层的硬件有了很大提高；设计 OpenGL 的人意识到这些 API 需要演进，并跟上这些变化。在 OpenGL ES 2.0 里，他们使用着色器加入了可编程 API；为了保持简洁，他们把那些固定的 API 完全删除了，因此，用户必须使用着色器。

我们现在用着色器控制每个顶点应该如何画到屏幕上，我们也控制所有点、直线和三角形上的每个片段应该如何绘制；这打开了一个新的、充满了无限可能的新世界。我们现在可以按每个像素实现光照和其他优美的效果，如卡通着色。只要我们可以用着色器语言表达出来，就可以加入任何理想的自定义效果。

作为 OpenGL 和着色器的快速参考，khronos.org 提供了一个很好的快速参考卡片，可以把它打印出来，并且随时查看[⊖]。

2.5.1 创建第一个顶点着色器

让我们创建一个简单的顶点着色器，它会分配在代码中定义的那些位置；为此，首先需要按照下面的步骤为这个着色器创建一个新的文件：

1. 首先，需要创建一个新文件夹；右键单击项目中的“res”文件夹，选择“New”，再选择“Folder”，并把这个新文件夹命名为“raw”。
2. 现在，需要创建一个新文件；右键单击刚刚创建的新文件夹，选择“New”，再选择“File”，并把新文件命名为“simple_vertex_shader.glsl”。

既然着色器的新文件已经创建好了，让我们在其中加入如下代码：

⊖ http://www.khronos.org/opengles/sdk/docs/reference_cards/OpenGL-ES-2_0-Reference-card.pdf

AirHockey1/res/raw/simple_vertex_shader.glsl

```
attribute vec4 a_Position;

void main()
{
    gl_Position = a_Position;
}
```

这些着色器使用 GLSL 定义，GLSL 是 OpenGL 的着色语言；这个着色语言的语法结构与 C 语言相似。更多的信息可以参考前文提到的快速参考卡片或者完整的规范[⊖]。

对于我们定义过的每个单一的顶点，顶点着色器都会被调用一次；当它被调用的时候，它会在 a_Position 属性里接收当前顶点的位置，这个属性被定义成 vec4 类型。

一个 vec4 是包含 4 个分量的向量；在位置的上下文中，可以认为这 4 个分量是 x、y、z 和 w 坐标，x、y 和 z 对应一个三维位置，而 w 是一个特殊的坐标，第 6 章会讲述更多关于 w 的细节。如果没有指定，默认情况下，OpenGL 都是把向量的前三个坐标设为 0，并把最后一个坐标设为 1。

还记得曾经讲过一个顶点会有几个属性，比如颜色和位置么？关键词“attribute”就是把这些属性放进着色器的手段。

之后，可以定义 main()，这是着色器的主要入口点；它所做的就是把前面定义过的位置复制到指定的输出变量 gl_Position；这个着色器一定要给 gl_Position 赋值；OpenGL 会把 gl_Position 中存储的值作为当前顶点的最终位置，并把这些顶点组装成点、直线和三角形。

2.5.2　创建第一个片段着色器

既然已经创建了一个顶点着色器，就有了为每个顶点生成最终位置的子例程；我们仍然需要创建一个为每个片段生成最终颜色的子例程。在此之前，让我们花些时间了解一下什么是片段，以及一个片段是怎么产生的。

光栅化（Rasterization）技术

移动设备的显示屏由成千上百万个小的、独立的部件组成，它们称为像素（pixel）；这些像素中的每一个都有能力显示几百万种不同颜色范围中的一种颜色。然而，这实际上是一种视觉技巧：大多数显示器并不能真正创造几百万种颜色，所以每个像素通常由三个单独的子组件构成，它们发出红色、绿色和蓝色的光，因为每个像素都非常小，人的眼睛会把红色、绿色及蓝色的光混合在一起，从而创造出巨量的颜色范围；把足够多的单独的像素放在一起，就能显示出一页文本或者蒙娜丽莎像。

OpenGL 通过“光栅化”的过程把每个点、直线及三角形分解成大量的小片段，它们可以映射到移动设备显示屏的像素上，从而生成一幅图像。这些片段类似于显示屏上的像素，

⊖ 分别是 http://www.khronos.org/opengles/sdk/docs/reference_cards/OpenGL-ES-2_0-Reference-card.pdf 和 http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf

每一个都包含单一的纯色。为了表示颜色，每个片段都有 4 个分量：其中红色、绿色、蓝色用来表示颜色，阿尔法（alpha）分量用于表示透明度；关于这个颜色模型是如何工作的，我们将在 2.6 节中讨论更多的细节。

在图 2-7 中，可以见到 OpenGL 怎样把一条直线光栅化为一个片段集合。显示系统通常会把这些片段直接映射到屏幕上的像素，结果一个片段就对应一个像素；然而，并不总是这样的：一个超高分辨率的设备可能需要使用较大的片段，以减少 GPU 的工作负荷。

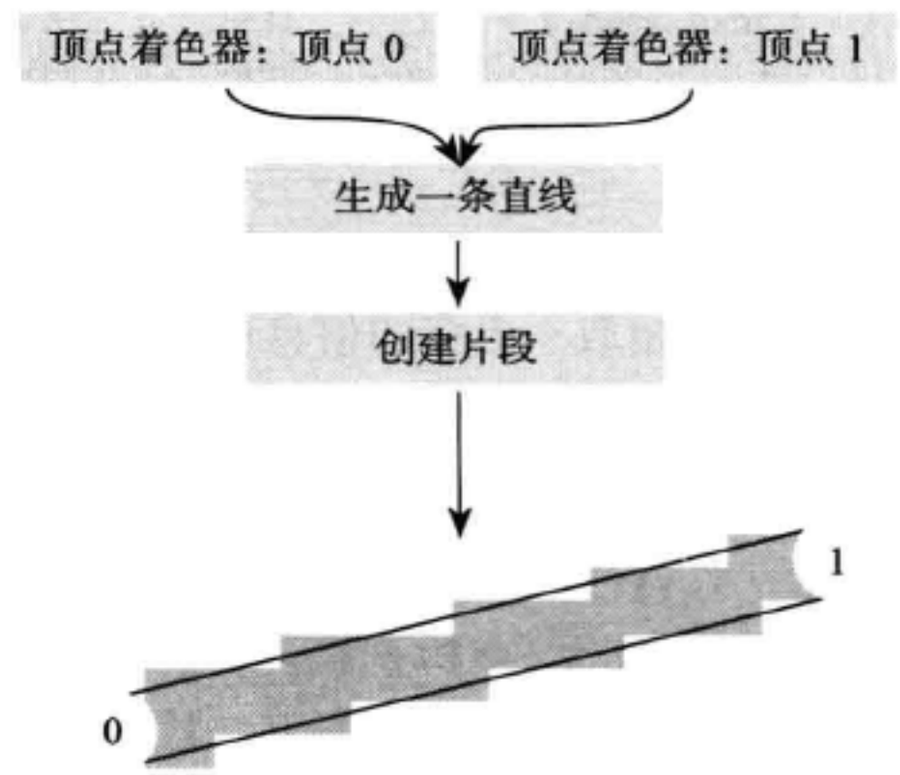


图 2-7 光栅化：生成片段

编写代码

片段着色器的主要目的就是告诉 GPU 每个片段的最终颜色应该是什么。对于基本图元[⊖]的每个片段，片段着色器都会被调用一次，因此，如果一个三角形被映射到 10 000 个片段，片段着色器就会被调用 10 000 次。

让我们继续并编写这个片段着色器；在项目中创建一个新的文件——“/res/raw/simple_fragment_shader.glsl”，并加入如下代码：

AirHockey1/res/raw/simple_fragment_shader.glsl

```
precision mediump float;

uniform vec4 u_Color;

void main()
{
    gl_FragColor = u_Color;
}
```

精度限定符

在这个片段着色器中，文件顶部的第一行代码定义了所有浮点数据类型的默认精度。这就像在 Java 代码中选择浮点数还是双精度浮点数一样。

可以选择 lowp、mediump 和 highp，它们分别对应低精度、中等精度及高精度；然而，只有某些硬件实现支持在片段着色器中使用 highp。

为什么顶点着色器没有定义精度呢？顶点着色器同样可以改变其默认的精度，但是，对于一个顶点的位置而言，精确度是最重要的，OpenGL 设计者决定把顶点着色器的精度默认设置成最高级——highp。

你可能已经猜到了，高精度数据类型更加精确，但是这是以降低性能为代价的；对于片段着色器，出于最大兼容性的考虑，选择了 mediump，这也是基于速度和质量的权衡。

⊖ 这里指点、直线以及三角形。——译者注

生成片段的颜色

这个片段着色器的剩余部分与早前定义的顶点着色器一样。不过这次我们要传递一个 uniform，它叫做 u_Color。它不像属性，每个顶点都要设置一个；一个 uniform 会让每个顶点都使用同一个值，除非我们再次改变它。如顶点着色器中的位置所使用的属性一样，u_Color 也是一个四分量向量，但在颜色的上下文中，这四个分量分别对应红色、绿色、蓝色和阿尔法。

接着我们定义了 main()，它是这个着色器的主入口点，它把我们在 uniform 里定义的颜色复制到那个特殊的输出变量——gl_FragColor。着色器一定要给 gl_GragColor 赋值，OpenGL 会使用这个颜色作为当前片段的最终颜色。

2.6　OpenGL 颜色模型

OpenGL 使用累加 RGB 颜色模型，它只用了三种基本颜色：红色、绿色和蓝色。许多颜色都是通过把这三种基本颜色按不同比例混合在一起而创造的。例如，红色和绿色放在一起会生成黄色，红色和蓝色放在一起可以产生品红色，而蓝色和绿色放在一起就会创造出青色，把红色、绿色和蓝色放在一起，就能看见白色，如图 2-8 所示。

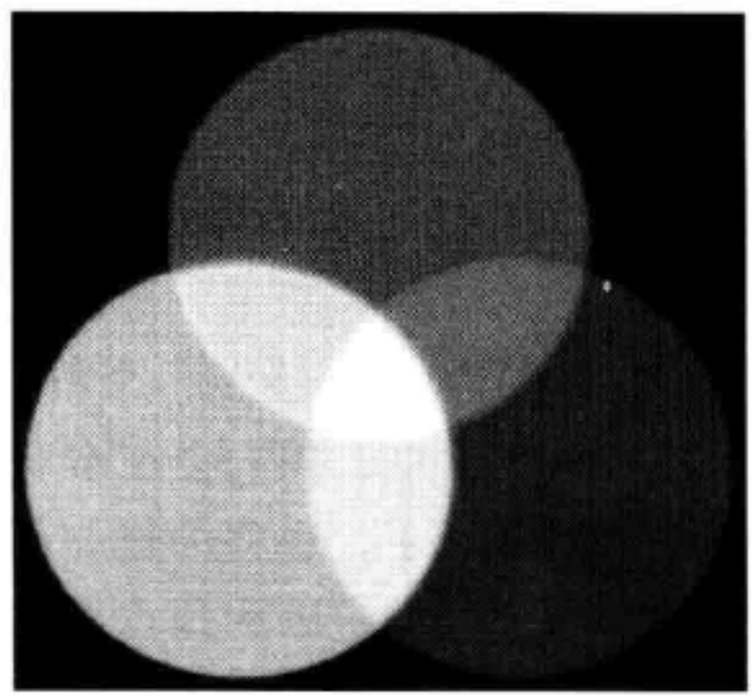


图 2-8　RGB 累加颜色模型

这个模型的工作原理与你可能在学校里学过的减色绘画模型（subtractive paint model）不同：在减色绘画模型里，加入蓝色和黄色制作出绿色，而加入很多颜色会产生黑棕色或者黑色。这是因为颜料不发光，而是吸收光；画上使用的颜色越多，光被吸收得越多，这幅画就会表现得越暗。

累加 RGB 模型遵循光本身的属性，当两柱不同颜色的光线混合在一起时，不会看见更暗的颜色，而是更亮的颜色。一场大雨过后，当我们观察天空中的彩虹时，我们实际上看到了可见光光谱中所有不同的颜色，它们可以合并成白色。

如果感兴趣，Wikipedia 有更多的细节[⊖]。

把颜色映射到显示屏上

OpenGL 假定这些颜色相互之间是线性关系：一个值为 0.5 的红色，其亮度是值为 0.25 的红色的两倍；而一个值为 1 的红色的亮度是值为 0.5 的红色的两倍。这些基本的颜色值被限定在范围 [0, 1] 内，0 代表缺少这个特定的基本颜色，而 1 代表那个颜色的最大强度。

这个颜色模型很好地映射到移动设备使用的显示屏和计算机屏幕上（然而，在 13.3.3

⊖ http://en.wikipedia.org/wiki/RGB_color_model

节中，我们会知道这个映射并不是真的一对一）。这些显示设备几乎总是使用这三种基本颜色：红色、绿色和蓝色（为了有“更纯的黄色”，有些设备可能用黄色作为另外一个基本颜色），0 映射到一个未点亮的像素分量，而 1 映射到那个颜色的全亮度。使用这个颜色模型，人眼能看到的所有颜色几乎都可以被 OpenGL 渲染出来，并显示在屏幕上。

在第 4 章里，我们会学习更多关于颜色使用的内容。

2.7 小结

本章用大量篇幅只介绍了如何定义数据，以及沿着 OpenGL 管道移动数据的着色器。让我们花点儿时间复习一下本章学到的那些关键概念：

- 首先，我们学习了如何定义一个顶点属性数组，并把这个数组复制到本地内存里，以便 OpenGL 存取它。
- 然后，我们写了一个顶点着色器和一个片段着色器；并了解到一个着色器只是可以运行在 GPU 上的一个特殊类型的程序。

在下一章里，我们会在本章的基础上继续开发；在下一章的结尾处，我们就能看到空气曲棍球桌子了，也将为进一步的练习做好准备。我们将以学习如何读入和编译定义过的着色器作为开始。因为顶点着色器和片段着色器总是一起工作，所以我们还要学习如何把这些着色器链接起来形成一个 OpenGL 程序。

一旦我们把这些着色器编译并链接在一起，我们就可以把所有的内容放在一起，并告诉 OpenGL 把第一个版本的空气曲棍球桌子画到屏幕上了。

第 3 章 Chapter 3

编译着色器及在屏幕上绘图

本章会继续上一章开始的工作。作为本章的开发计划，我们首先加载并编译前面定义的着色器，然后把它们链接在一起放在 OpenGL 的一个程序里。我们接下来就可以用着色器程序在屏幕上绘制空气曲棍球桌子了。

打开上一章启动的 AirHockey1 项目，并从那里开始。

3.1 加载着色器

我们既然已经为着色器写了代码，下一步就要把它们加载到内存中；为此，我们首先需要写一个可以从资源文件夹读取那些代码的方法。

3.1.1 从资源中加载文本

在项目中创建一个新的 Java 源代码包，命名为“com.airhockey.android.util”，在这个包中创建一个名为“TextResourceReader”的新类。在类中加入如下代码：

AirHockey1/src/com/airhockey/android/util/TextResourceReader.java

```
public static String readTextFileFromResource(Context context,
    int resourceId) {
    StringBuilder body = new StringBuilder();

    try {
        InputStream inputStream =
            context.getResources().openRawResource(resourceId);
        InputStreamReader inputStreamReader =
            new InputStreamReader(inputStream);
```

```
        BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

        String nextLine;

        while ((nextLine = bufferedReader.readLine()) != null) {
            body.append(nextLine);
            body.append('\n');
        }
    } catch (IOException e) {
        throw new RuntimeException(
                "Could not open resource: " + resourceId, e);
    } catch (Resources.NotFoundException nfe) {
        throw new RuntimeException("Resource not found: " + resourceId, nfe);
    }

    return body.toString();
}
```

我们已经定义了一个方法从资源中读取文本，该方法就是 readTextFileFromResource()。它的工作原理就是在程序代码中通过传递 Android 上下文（context）及资源标识符（resource ID）调用 readTextFileFromResource()。为了存取资源标识符，Android 上下文是必需的。例如，要读入片段着色器[⊖]，我们需要用代码 readTextFileFromSource(this.context, R.raw.simple_fragment_shader) 使用这个方法。

我们也在代码中检测了两个可能遇到的一般情形：资源可能不存在，或者读取资源时发生错误。这些情况发生时，要捕捉错误并抛出封装好的异常（exception），这些异常对发生的错误进行了解释。如果代码失败并且抛出了一个异常，通过查看异常的消息和栈轨迹（stack trace），就能更好地知道发生了什么事情。

另外，别忘了按组合键“Ctrl+Shift+O”（在 Mac 上是“⌘⇧O”）把代码中要用的导入（import）加进来。

3.1.2 读入着色器的代码

我们现在就要加入一些调用真正地读取着色器的代码。切换到 AirHockeyRender.java 文件，并在 onSurfaceCreated() 中的 glClearColor() 调用后面加入如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
String vertexShaderSource = TextResourceReader
    .readTextFileFromResource(context, R.raw.simple_vertex_shader);
String fragmentShaderSource = TextResourceReader
    .readTextFileFromResource(context, R.raw.simple_fragment_shader);
```

别忘了加入 TextResourceReader 的导入。因为我们还没有一个 Android 上下文的引用，这段代码还无法编译。在类的顶部加入如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
private final Context context;
```

⊖ 原文为顶点着色器，但是后面代码使用的是片段着色器，原文有误，更正之。——译者注

像下面这样改变构造函数的起始行：

```
AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java
public AirHockeyRenderer(Context context) {
    this.context = context;
```

我们也不得不改变 AirHockeyActivity.java，把 Android 上下文的引用传递进去；打开 AirHockeyActivity.java，按如下方式改变 glSurfaceView.setRenderer() 的调用：

```
AirHockey1/src/com/airhockey/android/AirHockeyActivity.java
glSurfaceView.setRenderer(new AirHockeyRenderer(this));
rendererSet = true;
```

一个 Activity 组件就是 Android 的上下文，因此，把 this 的引用传递进去。

3.1.3　记录发生的一切

当我们开始撰写更复杂的代码时，万一在某处犯了错误，查看事情发生的轨迹会很有帮助。在 Android 里，我们可以使用 Log 类把所有的事情记录在系统日志（System log）中，这些记录可以在 Eclipse 的 Logcat 视图中看到。

我们并不总是想记录所有的事情，因此，我们在 com.airhockey.android.util 包中用如下代码添加了一个称为 LoggerConfig 的新类：

```
AirHockey1/src/com/airhockey/android/util/LoggerConfig.java
package com.airhockey.android.util;

public class LoggerConfig {
    public static final boolean ON = true;
}
```

我们无论何时想要记录一些事情时，就要检查这个常量是真值（true）还是假值（false）。为了打开或关闭这些记录，我们所需要做的就是更新这个常量值并重新编译这个程序。

3.2　编译着色器

现在我们已经把着色器源代码从文件中读出来了，下一步就是编译每个着色器了。我们要创建一个新的辅助类，它可以创建新的 OpenGL 着色器对象、编译着色器代码并且返回代表那段着色器代码的着色器对象。一旦写出样板代码，在未来的项目中就可以重用了。

作为开始，创建一个名为 ShaderHelper 的新类，并在类中添加如下代码：

```
AirHockey1/src/com/airhockey/android/util/ShaderHelper.java
private static final String TAG = "ShaderHelper";

public static int compileVertexShader(String shaderCode) {
    return compileShader(GL_VERTEX_SHADER, shaderCode);
}
```

```
public static int compileFragmentShader(String shaderCode) {
    return compileShader(GL_FRAGMENT_SHADER, shaderCode);
}

private static int compileShader(int type, String shaderCode) {

}
```

这些代码会作为着色器辅助类的基础。与以前一样，不要忘了把导入加进代码中；如果你在使用静态导入时碰到什么问题，请参考 1.5 节；在本书的剩余部分，我们会一直遵循这个样式。

在下一节中，我们会逐步构建 compileShader() 方法。

3.2.1 创建一个新的着色器对象

我们应该做的第一件事就是创建一个新的着色器对象，并且检查这个创建是否成功。在 compileShader() 中加入如下代码：

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java

```
final int shaderObjectId = glCreateShader(type);

if (shaderObjectId == 0) {
    if (LoggerConfig.ON) {
        Log.w(TAG, "Could not create new shader.");
    }

    return 0;
}
```

这里，用 glCreateShader() 调用创建了一个新的着色器对象，并把这个对象的 ID 存入变量 shaderObjectId。这个 type 可以是代表顶点着色器的 GL_VERTEX_SHADER，或者是代表片段着色器的 GL_FRAGMENT_SHADER。剩下的代码也用同样的方式。

记住我们是如何创建对象并检查它是否有效的；这个模式将在 OpenGL 里广泛使用：

1. 首先使用一个如 glCreateShader() 一样的调用创建一个对象，这个调用会返回一个整型值（integer）。
2. 这个整型值就是 OpenGL 对象的引用。无论后面什么时候想要引用这个对象，就要把这个整型值传回 OpenGL。
3. 返回值 0 表示这个对象创建失败，它类似于 Java 代码中返回 null 值。

如果对象创建失败，就给调用代码返回 0。为什么返回 0 而不是抛出一个异常呢？这是因为 OpenGL 内部实际不会抛出任何异常；相反，我们会得到返回值 0，并且 OpenGL 通过 glGetError() 告诉我们这个错误，这个方法可以让我们询问 OpenGL 是不是某个 API 调用导致了错误。我们会一直遵从这个惯例。

附录 B 介绍了更多关于 glGetError() 的知识以及其他调试 OpenGL 代码的方法。

3.2.2　上传和编译着色器源代码

让我们加入如下代码把着色器源代码上传到着色器对象里：

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java

```
glShaderSource(shaderObjectId, shaderCode);
```

一旦有了有效的着色器对象，就可以调用 glShaderSource（shaderObjectId, shaderCode）上传源代码了。这个调用告诉 OpenGL 读入字符串 shaderCode 定义的源代码，并把它与 shaderObjectId 所引用的着色器对象关联起来。然后，可以调用 glCompileShader（shaderObjectId）编译这个着色器：

```
glCompileShader(shaderObjectId);
```

这个调用告诉 OpenGL 编译先前上传到 shaderObjectId 的源代码。

3.2.3　取出编译状态

让我们加入如下代码检查 OpenGL 是否能成功地编译这个着色器：

```
final int[] compileStatus = new int[1];
glGetShaderiv(shaderObjectId, GL_COMPILE_STATUS, compileStatus, 0);
```

为了检查编译是失败还是成功，首先要创建一个新的长度为 1 的 int 数组，称为 compileStatus；然后调用 glGetShaderiv(shaderObjectId, GLES20.GL_COMPILE_STATUS, compileStatus, 0)。这就告诉 OpenGL 读取与 shaderObjectId 关联的编译状态，并把它写入 compileStatus 的第 0 个元素。

这是 Android 平台上的 OpenGL 的另外一个通用模式。为了取出一个值，我们通常会使用一个长度为 1 的数组，并把这个数组传进一个 OpenGL 调用。在同一个调用中，我们告诉 OpenGL 把结果存进数组的第一个元素中。

3.2.4　取出着色器信息日志

当我们获得编译状态的时候，OpenGL 只给出一个简单的是或否的回答。难道没兴趣知道发生了什么错误以及哪里出问题了吗？事实证明，我们可以通过调用 glGetShaderInfoLog (shaderObjectId) 获得一个可读的消息。如果 OpenGL 有什么关于着色器的有用内容，它就会把消息存到着色器的信息日志里。

让我们加入如下代码获取着色器信息日志：

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java

```
if (LoggerConfig.ON) {
    // Print the shader info log to the Android log output.
    Log.v(TAG, "Results of compiling source:" + "\n" + shaderCode + "\n:"
        + glGetShaderInfoLog(shaderObjectId));
}
```

我们把日志输出到 Android 的日志输出中，并把一切都封装在那个检查 LoggerConfg.ON 值的 if 语句里。通过把这个常量赋值为“false”，我们可以很容易地关闭这些日志。

3.2.5 验证编译状态并返回着色器对象 ID

既然我们已经记录了着色器信息日志，就可以查看一下编译是否成功了：

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java

```
if (compileStatus[0] == 0) {
    // If it failed, delete the shader object.
    glDeleteShader(shaderObjectId);

    if (LoggerConfig.ON) {
        Log.w(TAG, "Compilation of shader failed.");
    }

    return 0;
}
```

我们所需要做的就是检查在 3.2.3 节那步的返回值，它是不是 0。如果它是 0，编译就失败了，这种情况下，我们就不再需要着色器对象了，因此告诉 OpenGL 把它删除并返回 0 给调用代码；如果编译成功，着色器对象就是有效的，我们就可以在代码中使用它了。

这就是为了编译一个着色器所需要的全部内容，让我们返回那个新的着色器对象 ID：

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java

```
return shaderObjectId;
```

3.2.6 在 Renderer 类中编译着色器

现在是时候充分利用我们刚刚写过的代码了。切换到 AirHockeyRender.java，并在 onSurfaceCreated() 的结尾处加入如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
int vertexShader = ShaderHelper.compileVertexShader(vertexShaderSource);
int fragmentShader = ShaderHelper.compileFragmentShader(fragmentShaderSource);
```

让我们回顾一下本节所完成的工作。首先，我们创建了一个新类 ShaderHelper，并加入了一个用来创建、编译新着色器对象的方法；我们也创建了 LoggerConfig，一个用来在单一代码行打开或者关闭日志的类。

如果你再看一下 ShaderHelper，就会发现我们实际上定义了三个方法。

- compileShader()：这个 compileShader(type, shaderCode)[⊖]方法使用了着色器源代码和类型；type 可以是代表顶点着色器的 GL_VERTEX_SHADER，或者是代表片段着色器的 GL_FRAGMENT_SHADER。如果 OpenGL 能成功编译这个着色器，这个方法就会给调用代码返回着色器对象的 ID，否则，它就会返回 0。

⊖ 原文为 compileShader(shaderCode)，参数 type 没有被列出。——译者注

- compileVertexShader()：这个 compileVertexShader(shaderCode) 方法是调用 compileShader() 的辅助方法，使用 GL_VERTEX_SHADER 作为着色器类型。
- compileFragmentShader()：这个 compileVertexShder(shaderCode) 方法也是调用 compileShader() 的辅助方法，它使用 GL_FRAGMENT_SHADER 作为着色器类型。

如你所见，这段代码的实质内容都在 compileshader() 中；其他两个方法就是使用 GL_VERTEX_SHADER 或 GL_FRAGMENT_SHADER 调用它。

3.3　把着色器一起链接进 OpenGL 的程序

既然我们已经加载并编译了一个顶点着色器和一个片段着色器，下一步就是把它们绑定在一起放入一个单个的程序（program）里[⊖]。

3.3.1　理解 OpenGL 的程序

简单来说，一个 OpenGL 程序就是把一个顶点着色器和一个片段着色器链接在一起变成单个对象。顶点着色器和片段着色器总是一起工作的。没有片段着色器，OpenGL 就不知道怎么绘制那些组成每个点、直线和三角形的片段；如果没有顶点着色器，OpenGL 就不知道在哪里绘制这些片段。

我们知道顶点着色器计算屏幕上每个顶点的最终位置。我们也知道当 OpenGL 把这些顶点组织成点、直线和三角形并且分解成片段的时候，它就会向片段着色器询问每个片段最后的颜色。顶点和片段着色器一起合作生成屏幕上最终的图像。

虽然顶点着色器和片段着色器总是要一起工作，但并不意味着它们必须是一对一匹配的，我们可以同时在多个程序中使用同一个着色器。

让我们打开 ShaderHelper，并在类的结尾加入如下代码：

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java

```
public static int linkProgram(int vertexShaderId, int fragmentShaderId) {

}
```

与我们构建 compileShader() 所做的一样，我们也将一步一步地构建这个方法。它的大部分代码都在概念上与 compileShader() 相似。

3.3.2　新建程序对象并附上着色器

我们要做的第一件事情就是调用 glCreateProgram() 新建程序对象，并把那个对象的 ID 存进 programObjectId。让我们加入如下代码：

⊖ 这个程序是指 OpenGL 中的一个组件，与一般的应用程序不同，要注意区分。——译者注

```
final int programObjectId = glCreateProgram();

if (programObjectId == 0) {
    if (LoggerConfig.ON) {
        Log.w(TAG, "Could not create new program");
    }

    return 0;
}
```

这个语义同我们早前创建一个新的着色器对象是一样的：返回的整型值是程序对象的引用，如果对象创建失败，我们就会得到返回值 0。

下一步就是附上着色器：

```
glAttachShader(programObjectId, vertexShaderId);
glAttachShader(programObjectId, fragmentShaderId);
```

使用 glAttachShader() 方法把顶点着色器和片段着色器都附加到程序对象上。

3.3.3 链接程序

现在准备把这些着色器联合起来了，为此，将调用 glLinkProgram(programObjectId)：

```
glLinkProgram(programObjectId);
```

为了检查这个链接是成功还是失败，我们会遵循编译着色器时所使用的步骤：

```
final int[] linkStatus = new int[1];
glGetProgramiv(programObjectId, GL_LINK_STATUS, linkStatus, 0);
```

首先创建了一个新的整型数组来容纳这个结果，然后调用 glGetProgramiv(programObjectId, GLES20.GL_LINK_STATUS, linkStatus, 0）把结果存入这个数组。我们也会检查这个程序的信息日志，这样，如果什么地方出错了或者 OpenGL 有些关于这个程序有用的内容，我们就能在 Android 的日志输出中看到：

```
if (LoggerConfig.ON) {
    // Print the program info log to the Android log output.
    Log.v(TAG, "Results of linking program:\n"
        + glGetProgramInfoLog(programObjectId));
}
```

3.3.4 验证链接状态并返回程序对象 ID

现在需要检查链接状态：如果它是 0，就意味着这个链接失败，这个程序对象将无法被使用，因此要把它删除并返回 0 给调用代码：

```
if (linkStatus[0] == 0) {
    // If it failed, delete the program object.
    glDeleteProgram(programObjectId);
    if (LoggerConfig.ON) {
```

```
        Log.w(TAG, "Linking of program failed.");
    }
    return 0;
}
```

我们已经讲了这么多了，如果这个程序链接成功了，就可以在代码中使用它了。现在完成了，因此让我们返回这个新的程序对象给调用代码：

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java

```
return programObjectId;
```

3.3.5　给渲染类加入代码

既然我们已经有代码把这些着色器链接在一起了，让我们继续，从程序中调用它。首先在 AirHockeyRenderer 的顶部加入如下成员变量：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
private int program;
```

这个整型值用来存储那个链接的程序的 ID。在 onSurfaceCreated() 的结尾处加入如下代码把着色器链接起来：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
program = ShaderHelper.linkProgram(vertexShader, fragmentShader);
```

现在也许该拿起一杯咖啡，让你的大脑休息片刻了。在下一节，我们将开始做最后的拼接，并把数据链接到 OpenGL。

3.4　做最后的拼接

我们在前两章中用了很大篇幅为这个应用打下了很好的基础：我们学习了如何使用属性数组定义一个物体的结构，也学习了如何创建着色器、加载并编译它们，以及把它们链接起来形成一个 OpenGL 的程序。

现在是时候在这个基础上开始构建并把它们拼接起来了。在下面的几个步骤里，我们就要把这些部分拼在一起，并准备好把第一个版本的空气曲棍球桌子画到屏幕上。

3.4.1　验证 OpenGL 程序的对象

在开始使用 OpenGL 的程序之前，我们首先应该验证一下它，看看这个程序对于当前的 OpenGL 状态是不是有效的。根据 OpenGL ES 2.0 的文档，它也给 OpenGL 提供了一种方法让我们知道为什么当前的程序可能是低效率的、无法运行，等等[⊖]。

让我们在 ShaderHelper 中加入如下代码：

⊖ http://www.khronos.org/opengles/sdk/docs/man/xhtml/glValidateProgram.xml

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java

```
public static boolean validateProgram(int programObjectId) {
    glValidateProgram(programObjectId);

    final int[] validateStatus = new int[1];
    glGetProgramiv(programObjectId, GL_VALIDATE_STATUS, validateStatus, 0);
    Log.v(TAG, "Results of validating program: " + validateStatus[0]
        + "\nLog:" + glGetProgramInfoLog(programObjectId));

    return validateStatus[0] != 0;
}
```

我们调用 glValidateProgram() 来验证这个程序，然后用 GL_VALIDATA_STATUS 作为参数调用 glGetProramiv() 方法检查其结果。如果 OpenGL 有什么有用的信息要透露，这些信息会显示在程序日志里，因此我们也用 glGetProgramInfoLog () 把日志打印出来。

在开始使用这个程序之前，我们应该验证它，并且我们应该只有在开发或调试应用的时候才去验证它。让我们在 onSurfaceCreated() 的结尾处加入如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
if (LoggerConfig.ON) {
    ShaderHelper.validateProgram(program);
}
```

只有日志需要打开时，它才会调用我们早前定义的验证代码。下一步我们应该做的就是使用曾花费了很大力气创建的 OpenGL 程序。在 onSurfaceCreated() 结尾处加入如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
glUseProgram(program);
```

调用 glUseProgram() 告诉 OpenGL 在绘制任何东西到屏幕上的时候要使用这里定义的程序。

3.4.2 获得一个 uniform 的位置

下一步是获得我们早前在着色器中定义的 uniform 的位置。当 OpenGL 把着色器链接成一个程序的时候，它实际上用一个位置编号把片段着色器[⊖]中定义的每个 uniform 都关联起来了。这些位置编号用来给着色器发送数据，并且我们需要 u_Color 的位置，以便我们可以在要绘画的时候设置颜色。

让我们快速看一下片段着色器：

AirHockey1/res/raw/simple_fragment_shader.glsl

```
precision mediump float;

uniform vec4 u_Color;
```

⊖ 原文为顶点着色器，应该是片段着色器，因为前面 uniform 是在片段着色器代码中使用的，校正之。——译者注

```
void main()
{
    gl_FragColor = u_Color;
}
```

在这个着色器里，我们已经定义了一个称为 u_Color 的 uniform，并在 main() 中把这个 uniform 的值赋给了 gl_FragColor。我们要使用这个 uniform 设置将要绘制的东西的颜色；我们要绘制一张桌子、一个中间分隔线和两个木槌，并且我们要使用不同的颜色绘制它们。

让我们在 AirHockeyRenderer 的顶部加入如下定义：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
private static final String U_COLOR = "u_Color";
private int uColorLocation;
```

我们已经为这个 uniform 的名字创建了一个常量和一个用来容纳它在 OpenGL 程序对象中的位置的变量。uniform 的位置并不是事先指定的，因此，一旦程序链接成功了，我们就要查询这个位置。一个 uniform 的位置在一个程序对象中是唯一的：即使在两个不同的程序中使用了相同的 uniform 名字，也不意味着它们使用相同的位置。

在 onSurfaceCreated() 结尾处加入如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
uColorLocation = glGetUniformLocation(program, U_COLOR);
```

我们调用 glGetUniformLocation() 获取 uniform 的位置，并把这个位置存入 uColorLocation；当我们稍后要更新这个 uniform 值的时候，我们会使用它。

3.4.3　获取属性的位置

像 uniform 一样，在使用属性之前我们也要获得它们的位置。我们可以让 OpenGL 自动给这些属性分配位置编号，或者在着色器被链接到一起之前，可以通过调用 glBindAttribLocation() 由我们自己给它们分配位置编号。我们要让 OpenGL 自动分配这些属性位置，因为它使代码更容易管理。

让我们在 AirHockeyRender 顶部加入如下定义：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
private static final String A_POSITION = "a_Position";
private int aPositionLocation;
```

一旦着色器被链接在一起了，我们就只需要加入一些代码去获取属性位置。在 onSurfaceCreated() 结尾处加入如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
aPositionLocation = glGetAttribLocation(program, A_POSITION);
```

调用 glGetAttribLocation() 获取属性的位置。有了这个位置，就能告诉 OpenGL 到哪里去找到这个属性对应的数据了。

3.4.4 关联属性与顶点数据的数组

下一步是要告诉 OpenGL 到哪里找到属性 a_Position 对应的数据。

在 onSurfaceCreated() 的结尾处加入如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
vertexData.position(0);
glVertexAttribPointer(aPositionLocation, POSITION_COMPONENT_COUNT, GL_FLOAT,
    false, 0, vertexData);
```

回到本章的开始处，我们创建了一个浮点数数组，这些浮点数表示组成空气曲棍球桌子的顶点的位置；我们在本地内存中创建了一个缓冲区，称为 vertexData，并把这些位置复制到这个缓冲区内。

在我们告诉 OpenGL 从这个缓冲区中读取数据之前，需要确保它会从开头处开始读取数据，而不是中间或者结尾处。每个缓冲区都有一个内部的指针，可以通过调用 position(int) 移动它，并且当 OpenGL 从缓冲区中读取时，它会从这个位置开始读取。为了保证它一定从最开头处开始读取，我们调用 position(0) 把位置设在数据的开头处。

然后我们调用 glVertexAttribPointer() 告诉 OpenGL，它可以在缓冲区 vertexData 中找到 a_Position 对应的数据。这是一个非常重要的函数，因此，让我们仔细看一下每个参数传递了什么（见表 3-1）。

表 3-1 glVertexAttribPointer 的参数

glVertexAttribPointer(int index, int size, int type, boolean normalized, int stride, Buffer ptr)	
int index	这个是属性位置，我们传进 aPositionLocation，它指向我们早前在 3.4.3 节中获取的位置
int size	这是每个属性的数据的计数，或者对于这个属性，有多少个分量与每一个顶点相关联。回到 2.3 节，我们决定每个顶点使用两个浮点数：用来表达位置的一个 x 坐标和一个 y 坐标。这就意味着它需要两个分量，我们先前创建了常量 POSITION_COPOMENT_COUNT 标记这一事实，因此我们把这个常量值传递进去。 记住，我们为每个顶点只传递了两个分量，但是在着色器中，a_Position 被定义为 vec4，它有 4 个分量。如果一个分量没有被指定值，默认情况下，OpenGL 会把前 3 个分量设为 0，最后一个分量设为 1
int type	这是数据的类型。我们把数据定义为浮点数的列表，因此把 GL_FLOAT 传递进去
boolean normalized	只有使用整型数据的时候，这个参数才有意义，因此可以暂时把它安全地忽略掉
int stride	第 5 个参数“stride”，只有当一个数组存储多于一个属性时，它才有意义。本章只有一个属性，因此可以忽略这个参数，并暂时把 0 传递进去。我们会在 4.4 节里讨论更多有关“stride”的内容
Buffer ptr	这个参数告诉 OpenGL 去哪里读取数据。不要忘了它会从缓冲区的当前位置读取，如果我们没有调用 vertexData.position(0)，它可能尝试读取缓冲区结尾后面的内容，并使我们的应用程序崩溃

传递不正确的参数给 glVertexAttribPointer() 会导致奇怪的结果，甚至导致程序崩溃。这种崩溃还很难跟踪，因此，我不是言过其实，获得正确的参数是非常重要的。

调用了 glVertexAttribPointer() 之后，OpenGL 就知道在哪里读取属性 a_Position 的数据了。

3.4.5　使能顶点数组

尽管我们已经把数据属性链接起来了，在开始绘制之前，我们还需要调用 glEnableVertexAttribArray() 使能这个属性。在 glVertexAttribPointer() 调用之后加入如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
glEnableVertexAttribArray(aPositionLocation);
```

通过这最后一个调用，OpenGL 现在就知道去哪里寻找它所需要的数据了。

在这一节里，我们取得了 u_Color（uniform）及 a_Position（属性）的位置；每个变量都有一个位置，并且 OpenGL 将使用这些位置，而不是直接使用这些变量的名字；之后，我们调用 glVertexAttribPointer() 告诉 OpenGL，它可以从 vertexData 找到属性 a_Position 的数据。

3.5　在屏幕上绘制

随着完成这些最后的连接，我们现在就准备好开始在屏幕上绘制了！我们将先画桌子，然后画分隔线和木槌。

3.5.1　绘制桌子

在 onDrawFrame() 结尾处，让我们在 glClear() 调用之后加入如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
glUniform4f(uColorLocation, 1.0f, 1.0f, 1.0f, 1.0f);
glDrawArrays(GL_TRIANGLES, 0, 6);
```

我们首先通过调用 glUniform4f() 更新着色器代码中的 u_Color 的值。与属性不同，uniform 的分量没有默认值，因此，如果一个 uniform 在着色器中被定义为 vec4 类型，我们需要提供所有四个分量的值。我们想要以画一张白桌子作为开始，因此，我们把红色、绿色和蓝色的值设置为代表完全亮度的值 1.0f；阿尔法的值无关紧要，但是我们还是要指定它，因为一个颜色有四个分量。

一旦指定了颜色，接下来就可以用 glDrawArrays(GLES20.GL_TRIANGLES, 0, 6) 绘制桌子了；第一个参数告诉 OpenGL，我们想要画三角形。而要画三角形，我们需要给每个三角形传递进去至少三个顶点；第二个参数告诉 OpenGL 从顶点数组的开头处开始读顶点；而第三个参数是告诉 OpenGL 读入六个顶点。因为每个三角形有三个顶点，这个调用最终会画出两个三角形。

让我们快速看一下本章起始处定义的顶点数组：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
float[] tableVerticesWithTriangles = {
        // Triangle 1
        0f,  0f,
```

```
        9f, 14f,
        0f, 14f,

        // Triangle 2
        0f,  0f,
        9f,  0f,
        9f, 14f,

        // Line 1
        0f,  7f,
        9f,  7f,

        // Mallets
        4.5f,  2f,
        4.5f, 12f
};
```

当我们调用 glVertexAttribPointer() 方法的时候，记得我们曾告诉 OpenGL 每个顶点的位置包含两个浮点数分量。glDrawArrays() 调用让 OpenGL 使用前六个顶点绘制三角形，因此，OpenGL 会使用下面这些位置绘制它们：

```
// Triangle 1
0f,  0f,
9f, 14f,
0f, 14f,

// Triangle 2
0f,  0f,
9f,  0f,
9f, 14f,
```

第一个被绘制的三角形由点（0, 0）、（9, 14）和（0, 14）围成，而第二个由点（0, 0）、（9, 0）及（9, 14）围成。

3.5.2 绘制分隔线

下一步是绘制跨越桌子中间的中心分隔线。

在 onDrawFrame() 方法的结尾处加入如下代码：

```
glUniform4f(uColorLocation, 1.0f, 0.0f, 0.0f, 1.0f);
glDrawArrays(GL_LINES, 6, 2);
```

通过传递 1.0f 给第一个分量（红色）及传递 0.0f 给绿色和蓝色，我们把颜色设为红色；但这次需要 OpenGL 画线；我们从六个顶点后的第一个顶点开始读入两个顶点，OpenGL 就可以画出线了。正如 Java 数组一样，这里使用基于 0 的编号规则：0、1、2、3、4、5、6，数字 6 就是六个顶点后的第一个顶点，或者是第七个顶点。因为每条线有两个顶点，最终用这些位置可以画出一条线：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
// Line 1
0f,  7f,
9f,  7f,
```

OpenGL 会从（0, 7）到（9, 7）画一条线。

3.5.3　把木槌绘制为点

最后要做的就是画两个木槌；在 onDrawFrame() 结尾处加入如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
// Draw the first mallet blue.
glUniform4f(uColorLocation, 0.0f, 0.0f, 1.0f, 1.0f);
glDrawArrays(GL_POINTS, 8, 1);

// Draw the second mallet red.
glUniform4f(uColorLocation, 1.0f, 0.0f, 0.0f, 1.0f);
glDrawArrays(GL_POINTS, 9, 1);
```

通过传递 GL_POINTS 给 glDrawArrays() 方法，我们让 OpenGL 绘制点。对于第一个木槌，我们设置其颜色为蓝色，从偏移位置 8 开始，并用一个顶点绘制一个点；对于第二个木槌，我们设置其颜色为红色，以偏移位置 9 开始，并用一个顶点绘制一个点。我们将使用下面这些位置绘制这两个点：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
// Mallets
4.5f,  2f,
4.5f, 12f
```

OpenGL 会在（4.5, 2）绘制第一个点，在（4.5, 12）绘制第二个点。

3.5.4　目前为止我们得到了什么

让我们运行这个程序，看看屏幕上会出现什么。按“Ctrl+F11”键让程序运行，观察设备上或模拟器里会显示什么。你的屏幕看起来应该与图 3-1 相似。如果碰到了什么问题，首先在 Eclipse 里尝试选择“Project->Clean”，清理这个程序。

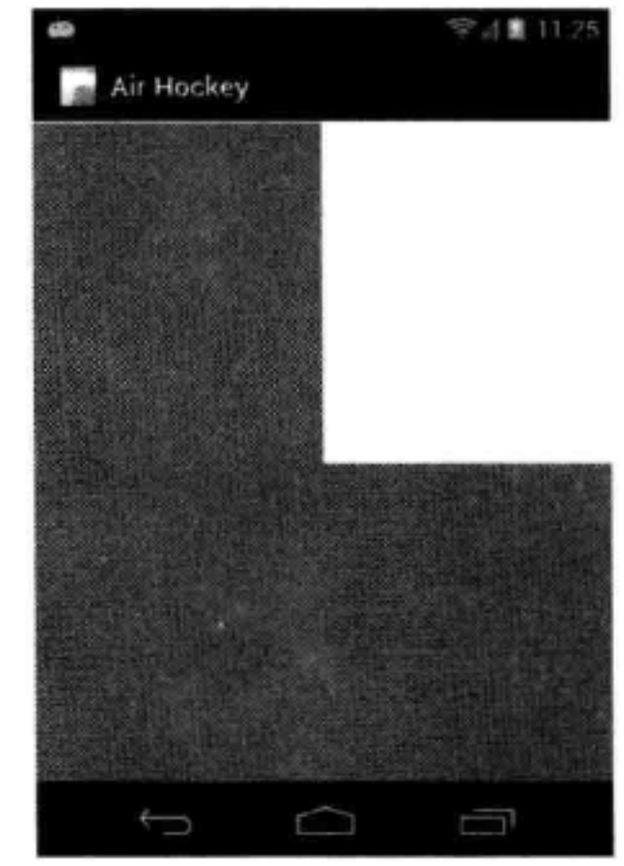


图 3-1　在屏幕上绘图：初次尝试

哦，看起来，哪里不太对！这个背景还是我们在第 1 章中使用的扎眼的红色，对于那个空气曲棍球桌子，我们为什么只看到一个角呢？在讨论这个之前，让我们来修复这个清屏用的颜色。在 onSurfaceCreated() 方法的开始处，找到 glClearColor() 的调用，把它更新成如下代码：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
```

当 glClear() 方法被调用的时候，这会告诉 OpenGL 把屏幕清除成黑色，而不是红色。现在我们已经修复了那个扎眼的红颜色，需要看一下为什么我们只能看见空气曲棍球桌子的一个角了。

3.5.5 OpenGL 如何把坐标映射到屏幕

目前为止，我们还没有解决的一个大问题就是：OpenGL 是怎样把我们已经定义的坐标映射到屏幕上的实际物理坐标的？

这个问题的答案很复杂，随着后面章节的讲解，我们会了解到更多有关的内容；目前，我们只需要知道，无论是 x 还是 y 坐标，OpenGL 都会把屏幕映射到 [−1,1] 的范围内。这就意味着屏幕的左边对应 x 轴的 −1，而屏幕的右边对应 +1；屏幕的底边会对应 y 轴的 −1，而屏幕的顶边就对应 +1，如图 3-2 所示。

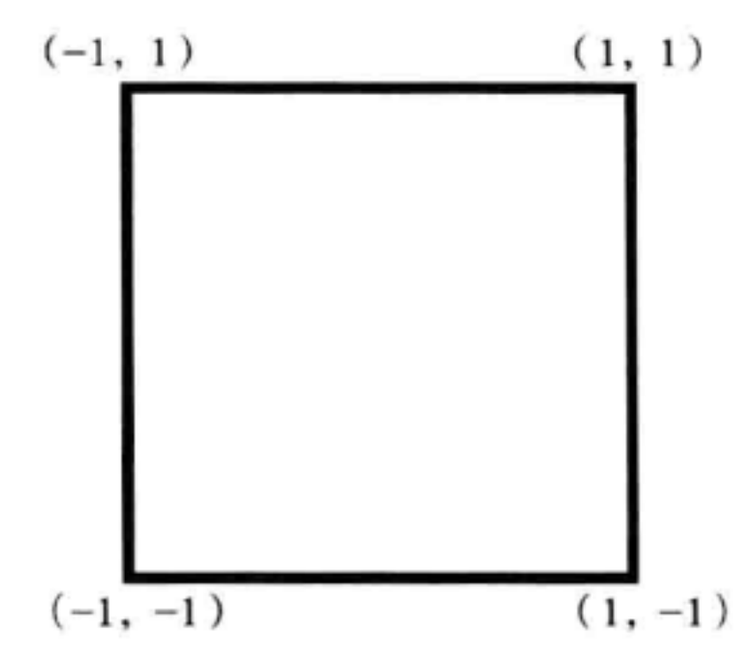


图 3-2 将坐标映射到屏幕

不管屏幕是什么形状和大小，这个坐标范围都是一样的，如果我们需要在屏幕上显示任何东西，都需要在这个范围内绘制它们。让我们回到构造函数，把 tableVerticesWithTriangles 中定义的坐标更新为如下值：

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
float[] tableVerticesWithTriangles = {
    // Triangle 1
    -0.5f, -0.5f,
     0.5f,  0.5f,
    -0.5f,  0.5f,

    // Triangle 2
    -0.5f, -0.5f,
     0.5f, -0.5f,
     0.5f,  0.5f,

    // Line 1
    -0.5f, 0f,
     0.5f, 0f,

    // Mallets
    0f, -0.25f,
    0f,  0.25f
};
```

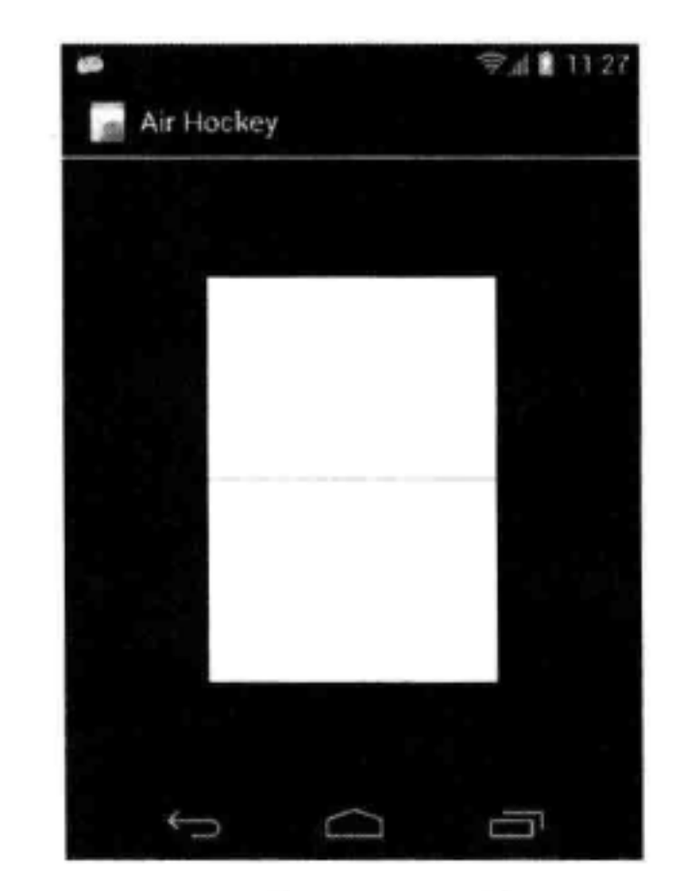


图 3-3 在屏幕上绘图：第二次尝试

让我们再次运行这个应用，我们应该看到与图 3-3 类似的屏幕：

这看起来好多了，但是木槌哪去了？事实证明，对于点来说，OpenGL 需要我们指定在屏幕上所显示的点的大小，但我们还没指定这个大小呢。

3.5.6 指定点的大小

我们可以更新代码，告诉 OpenGL 那些点在屏幕上所显示的大小。在 simple_vertex_shader.glsl 文件里，在 gl_Position 赋值后面，我们可以加入如下代码行指定其大小：

```
AirHockey1/res/raw/simple_vertex_shader.glsl
gl_PointSize = 10.0;
```

通过给另外一个特殊的输出变量 gl_PointSize 赋值，我们告诉 OpenGL 这些点的大小应该是 10。你可能会问，10 个什么？当 OpenGL 把一个点分解为片段的时候，它会生成一些片段，它们是以 gl_Position 为中心的四边形，这个四边形的每条边的长度与 gl_PointSize 相等。gl_PointSize 值越大，屏幕上绘出的点越大。

让我们再运行一次这个应用。我们应该能看到如图 3-4 所示的木槌，每个木槌都被渲染为单个的点。

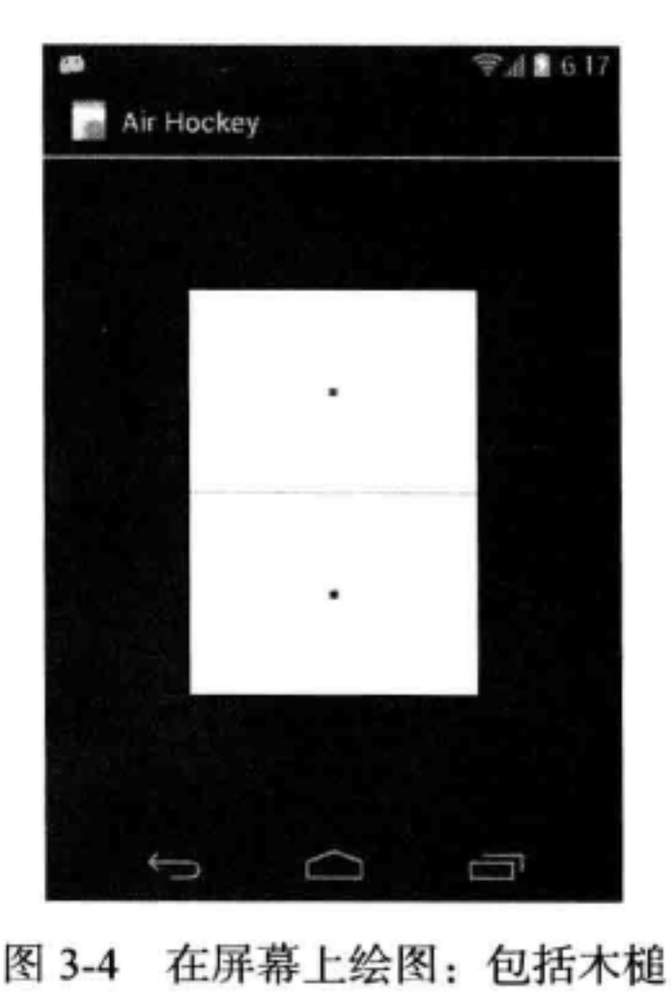


图 3-4　在屏幕上绘图：包括木槌

我们终于把它画出来了！休息一下，坐下来，回忆一下你在本章学过的内容。本章的内容很多，但是我们还是把它讲完了，而且想办法得到了我们要在屏幕上显示的东西。

如果你准备好了，让我们回顾一下学过的内容，再做两个后续的练习。

3.6　小结

在最后把第一个版本的空气曲棍球桌子显示之前，我们不得不撰写了很多样例代码，好消息是，我们可以在后面的项目中重用这些代码。让我们花点儿时间回顾一下本章学过的内容：

- 如何创建和编译着色器。
- 顶点着色器和片段着色器总是一起工作，我们也学到了如何把它们链接到一起形成一个 OpenGL 的程序对象。
- 如何把一个顶点着色器内部的属性变量与顶点属性数组关联起来。

最后，我们把所有的内容放在一起，在屏幕上显示一些东西。

既然我们已经把所有的知识点串连起来了，现在可能是时候回顾一下本章当时可能感觉不太清楚的部分了。你可以在本书的主页获取这些代码[⊖]。

3.7　练习

尝试在桌子的中间画一个冰球。做有点挑战性的事情，看看你是否能给桌子周围添加一个边框。你会怎么做呢？提示一下，看看你是否可以画两个不同颜色的长方形。

一旦你完成了这些练习，让我们开始下一章，学习如何让物体更多姿多彩。

⊖ http://pragprog.com/book/kbogla

Chapter 4 第 4 章

增加颜色和着色

在现实世界里，物体有着各种变化的颜色和着色。如果我们在家中看一下墙面，就会观察到它是用一种单一的颜色喷涂的。同时，墙的某些部分看起来更暗或更亮些，这依赖于墙上的那些部分与其周围的光线所处的方位。这些着色上的细微差别是主要的视觉线索之一，它帮助大脑感知我们所看到的事物；艺术家也自始至终一直在利用这些线索来“糊弄”我们的大脑。在本章，我们将从艺术家身上学习，通过变化的颜色和着色使这个桌子看上去不那么平坦，并且更加现实。

在前面的章节里，把空气曲棍球桌子画到屏幕上花了很多工夫，包括两个木槌和把桌子从中间分开的一条分隔线。我们也做了很多的准备工作，包括写代码加载和编译着色器，以及把它们链接起来成为一个 OpenGL 的程序对象。

对于 OpenGL ES 2.0，较好的方面就是，在一开始做出的许多努力都能作为更好的一个开端。尽管还要写很多代码，但是我们在接下来的章节里能重用这些基础代码。实际上，为我们的场景增加各种变化的颜色和着色是比较容易的，因为我们已经实现了一个基本的框架。

以下是本章的开发计划：

- 首先，我们会学习如何把每个点上的颜色定义为一个顶点属性，而不是整个对象都使用一种单一的颜色。
- 然后，我们会学习如何在构成一个物体的不同顶点之间平滑地混合这些颜色。

作为开始，让我们把第 2 章建立的项目复制到一个新的项目。

1. 在 Eclipse 里，选择上一章的项目 AirHockey1，并按“Ctrl+C”键。然后，按“Ctrl+V”键，复制那个项目。
2. 当有提示的时候，输入“AirHockey2”作为项目名称，如前面的章节一样，项目的位置由你决定。

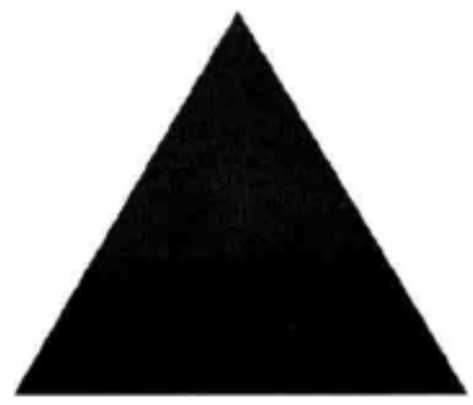
3. 我们可以同以前一样，继续用同样的应用包名和类名。

4.1　平滑着色

在第 2 章中，我们了解到如何在一个 uniform 里用单一的颜色绘制物体，如右图所示。

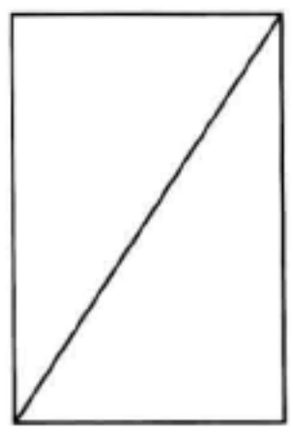
我们已经知道，我们只能画点、直线以及三角形，并且所有物体都以它们为基础构建。既然受限于这三个基本的图元，我们怎样用许多不同的颜色和着色表达一个复杂的场景呢？

我们能使用的一个方法是用上百万个小三角形，每个三角形都有一个不同的颜色。如果使用足够多的三角形，我们就能欺骗观察者，让他们看到一幅美丽的、复杂的、有丰富颜色变化的场景。尽管这在技术上是可行的，但是性能和内存的开销也是非常恐怖的。

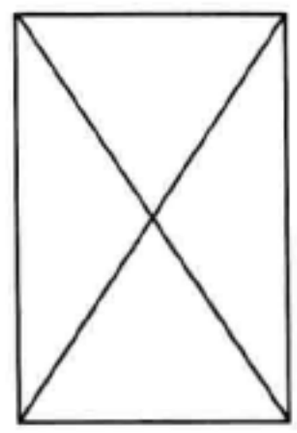


不绘制大量的平面三角形，如果有一个方法可以在同一个三角形中混合不同的颜色，如何？如果在一个三角形的每个点上都有一个不同的颜色，并在三角形的表面上混合这些颜色，我们最终将得到一个平滑着色的三角形，如右图所示。

平滑着色是在顶点之间完成的



OpenGL 给了我们一个方法，它可以平滑地混合一条直线或一个三角形的表面上的每个顶点的颜色。我们要使用这种类型的着色使桌子中心表现得更加明亮，而其边缘处显得比较暗淡，这就好像有一盏灯挂在桌子中间的上方一样。然而，在做这些之前，我们需要更新桌子的结构。我们现在是用两个三角形绘制桌子的，如右图所示。



我们怎样才能让中间显得更明亮呢？在中间并没有点，因此，我们不知道从哪里开始或者向哪里出发去混合颜色。我们需要在中间位置加入一个点，这样，就可以在桌子的中间和边缘之间混合颜色。桌子的结构看上去如右图。

我们需要做的第一件事情就是更新桌子以匹配这个新的结构。

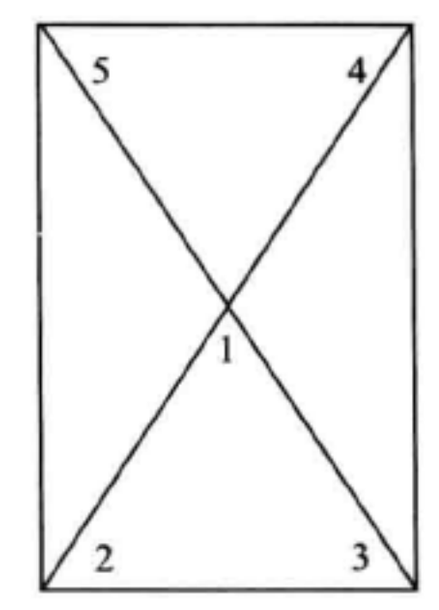
4.2　引入三角形扇

随着桌子中间这个新点的加入，我们最终看到了四个三角形，而不是两个。我们将把这个新点放在中心，其坐标是（0, 0）。让我们打开 AirHockeyRenderer 类，它在本章开头

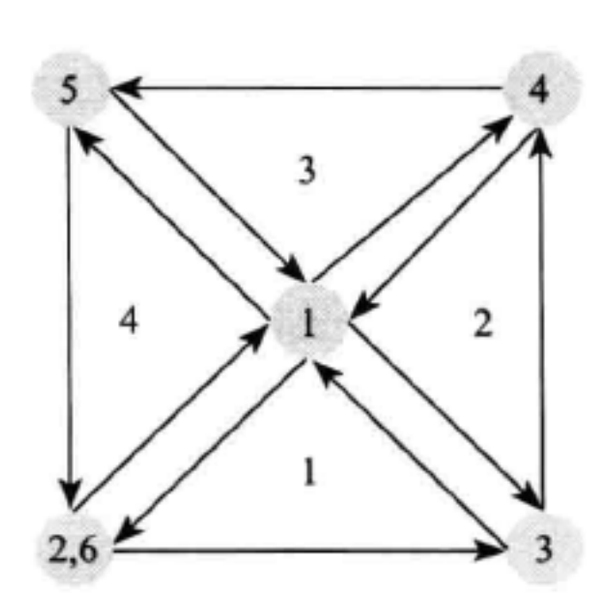
创建的那个新项目中，并把那些三角形更新为如下数据：

AirHockey2/src/com/airhockey/android/AirHockeyRenderer.java

```
// Triangle Fan
   0,     0,
-0.5f, -0.5f,
 0.5f, -0.5f,
 0.5f,  0.5f,
-0.5f,  0.5f,
-0.5f, -0.5f,
```



你可能问的第一个问题就是“为什么我们只定义了六个点？难道不是每个三角形都需要定义三个顶点么？”，确实每个三角形需要三个顶点，但有时候同一个顶点会用于多个三角形。通过给每个独一无二的顶点编号，让我们再看一下这个新的结构，如右图所示。



每一条边上的顶点都被两个三角形使用了，而且，中心的顶点被所有四个三角形使用了！如果我们不得不一次又一次地输入同一个坐标，很快就会感觉厌烦，相反，我们可以告诉 OpenGL 重用这些顶点。通过把这些顶点作为一个三角形扇（triangle fan）绘制，我们可以做到这些。一个三角形扇看起来如右图所示。

一个三角形扇以一个中心顶点作为起始，使用相邻的两个顶点创建第一个三角形，接下来的每个顶点都会创建一个三角形，围绕起始的中心点按扇形展开。为了使这个扇形闭合，我们只需要在最后重复第二个点。

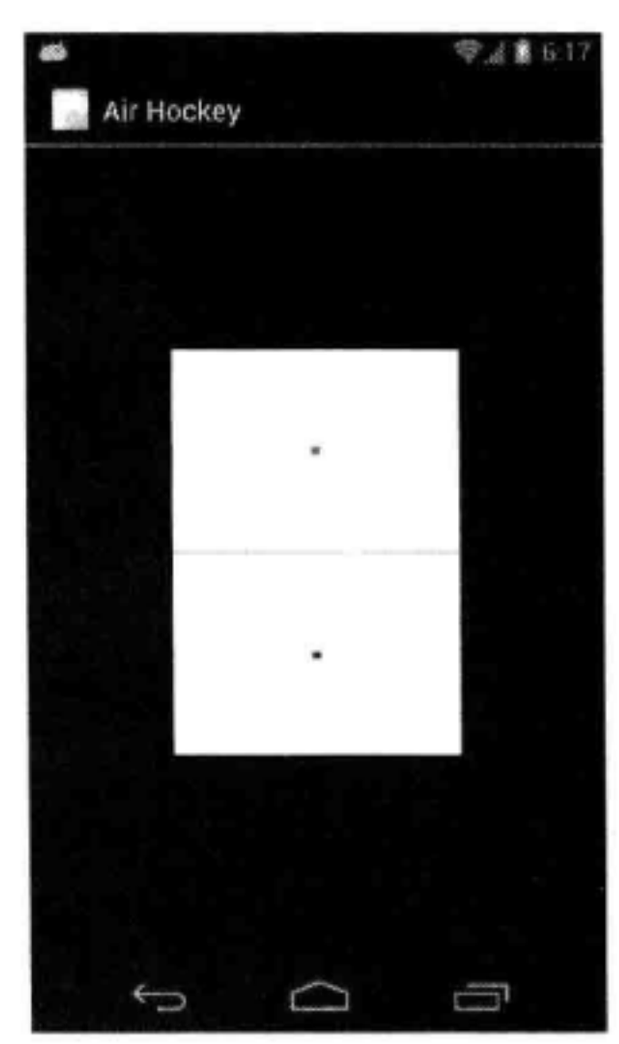


图 4-1 空气曲棍球：增加着色之前

需要更新绘图的调用函数，使得 OpenGL 知道这个数据代表一个三角形扇，在 onDrawFrame() 中，把第一个 glDrawArrays() 的调用更新为如下代码：

AirHockey2/src/com/airhockey/android/AirHockeyRenderer.java

```
glDrawArrays(GL_TRIANGLE_FAN, 0, 6);
```

这会让 OpenGL 使用我们已经定义过的六个新点绘制一个三角形扇。快速运行一下这个应用，看看是否一切还与我们期望的一样。它看上去应该如图 4-1 所示。

既然我们已经用一个中心点重新定义了这个桌子，让我们学习如何给每个顶点添加一个颜色作为第二个属性。

4.3 增加一个新的颜色属性

通过在桌子中心增加一个额外的点，我们已经更新了桌子

的数据结构，现在我们可以给每个点加入一个颜色属性。让我们把整个数据的数组更新如下：

AirHockey2/src/com/airhockey/android/AirHockeyRenderer.java

```
float[] tableVerticesWithTriangles = {
    // Order of coordinates: X, Y, R, G, B

    // Triangle Fan
       0f,    0f,   1f,   1f,   1f,

    -0.5f, -0.5f, 0.7f, 0.7f, 0.7f,
     0.5f, -0.5f, 0.7f, 0.7f, 0.7f,
     0.5f,  0.5f, 0.7f, 0.7f, 0.7f,
    -0.5f,  0.5f, 0.7f, 0.7f, 0.7f,
    -0.5f, -0.5f, 0.7f, 0.7f, 0.7f,

    // Line 1
    -0.5f, 0f, 1f, 0f, 0f,
     0.5f, 0f, 1f, 0f, 0f,

    // Mallets
    0f, -0.25f, 0f, 0f, 1f,
    0f,  0.25f, 1f, 0f, 0f
};
```

如你所见，我们给每个顶点增加了 3 个额外的数字。这些数字代表了红色、绿色和蓝色，它们一起构成那个对应特定顶点的颜色。

使用 Android 的 Color 类转换颜色

当使用浮点属性时，我们需要在范围 0 到 1 之间指定每个颜色分量的值，1 是某个颜色分量的最大值。为某个特定的颜色计算出正确的值可能不是那么明显，但是，使用 Android 的颜色类，我们能轻易地得出正确的值，比如，为了获取绿色对应的各个 OpenGL 颜色分量值可以使用以下语句：

```
float red = Color.red(Color.GREEN) / 255f;
float green = Color.green(Color.GREEN) / 255f;
float blue = Color.blue(Color.GREEN) / 255f;
```

也可以把它应用到网络颜色定义：

```
int parsedColor = Color.parseColor("#0099CC");

float red = Color.red(parsedColor) / 255f;
float green = Color.green(parsedColor) / 255f;
float blue = Color.blue(parsedColor) / 255f;
```

Color.red()、Color.green() 和 Color.blue() 的返回值范围是从 0 到 255，因此，为了把它们转换为 OpenGL 的颜色范围，只需要用每个分量除以 255。

4.3.1 给着色器增加颜色属性

下一步就是从着色器中去掉 uniform 定义的颜色，并用一个属性替换它。接下来会更新 Java 代码以体现这段新的着色器代码。

打开 simple_vertex_shader.glsl，并把它更新如下：

AirHockey2/res/raw/simple_vertex_shader.glsl

```
attribute vec4 a_Position;
attribute vec4 a_Color;

varying vec4 v_Color;

void main()
{
    v_Color = a_Color;

    gl_Position = a_Position;
    gl_PointSize = 10.0;
}
```

我们加入了一个新的属性 a_Color，也加入了一个叫做 v_Color 的新的 varying。你可能会问“varying 究竟是什么呀？”。还记得我们说过我们需要在一个三角形平面上让颜色产生变化（vary）么？就是通过这个被称为 varying 的特殊的变量类型实现的。为了更好地理解一个 varying 是做什么的，让我们回顾一下 OpenGL 是怎样把顶点合并在一起创建物体的，参见图 2-7 。

正如我们在 2.5 节中所学习到的，当 OpenGL 构建一条直线的时候，它会用两个顶点构成这条直线，并为这条直线生成片段；当 OpenGL 构建一个三角形的时候，它会同样使用三个顶点构建这个三角形。然后，对于每一个被生成的片段，片段着色器都会被执行一次。

varying 是一个特殊的变量类型，它把给它的那些值进行混合，并把这些混合后的值发送给片段着色器。使用上面的直线作一个例子，如果顶点 0 的 a_Color 是红色，且顶点 1 的 a_Color 是绿色，然后，通过把 a_Color 赋值给 v_Color，来告诉 OpenGL 我们需要每个片段都接收一个混合后的颜色。接近顶点 0 的片段，混合后的颜色显得更红，而越接近顶点 1 的片段，颜色就会越绿。

关于这种混合是如何实现的，在讨论更多的细节之前，我们把 varying 也加入片段着色器。打开 simple_fragment_shader.glsl，并把它更新如下：

AirHockey2/res/raw/simple_fragment_shader.glsl

```
precision mediump float;
varying vec4 v_Color;

void main()
{
    gl_FragColor = v_Color;
}
```

我们用 varying 变量 v_Color 替换了原来代码中的 uniform。如果那个片段属于一条直

线，那么 OpenGL 就会用构成那条直线的两个顶点计算其混合后的颜色；如果那个片段属于一个三角形，那 OpenGL 就会用构成那个三角形的三个顶点计算其混合后的颜色。

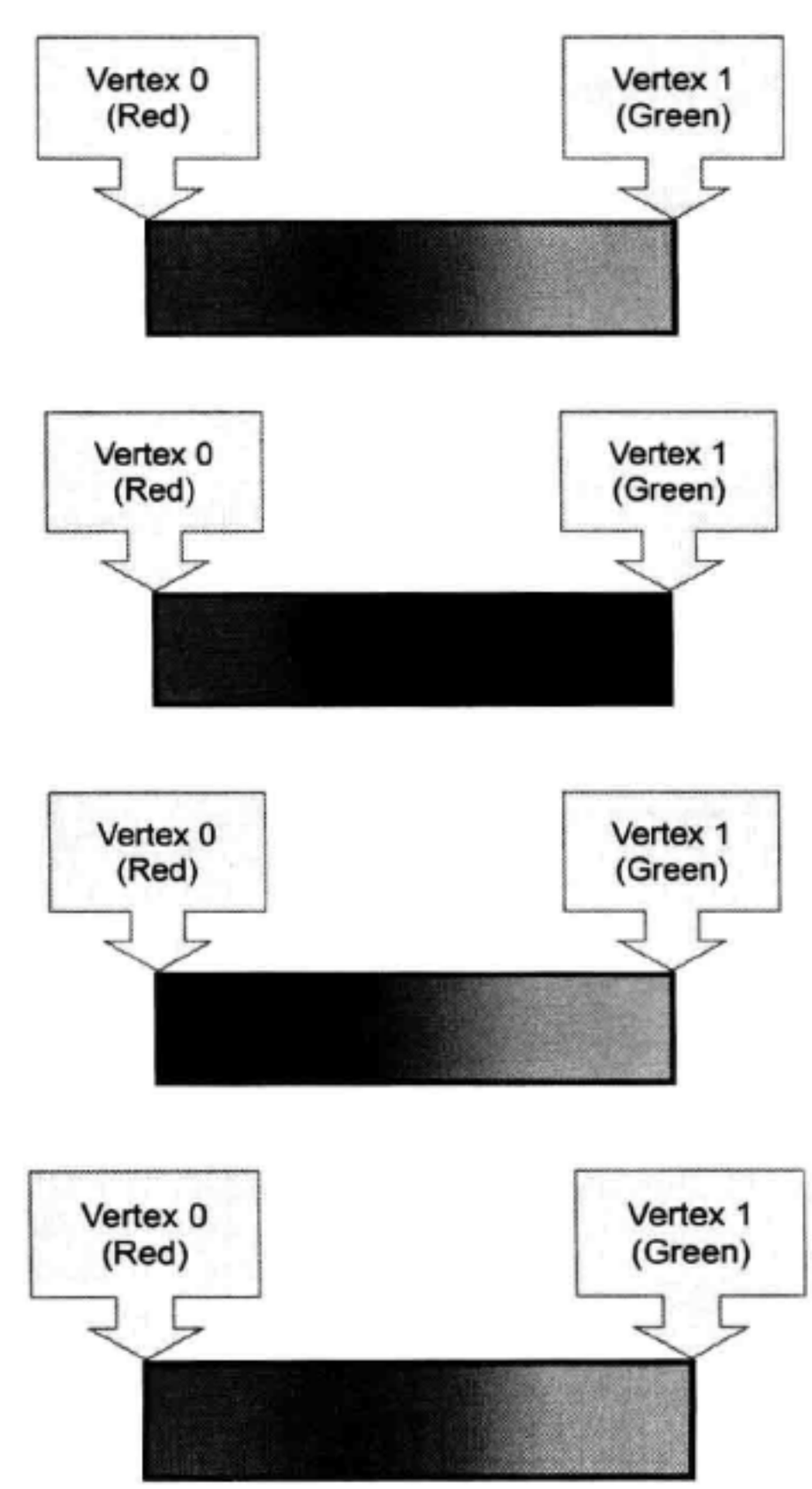
既然已经更新了着色器，我们也需要更新 Java 代码，以便我们传递新的颜色属性给顶点着色器中的 a_Color。在做这些之前，让我们花些时间多了解一下 OpenGL 怎样平滑地从一个点向另外一个点混合颜色。

4.3.2　一个 varying 如何生成每个片段上混合后的颜色

我们刚刚了解到，直线或三角形上的每个片段混合后的颜色可以用一个 varying 生成。我们不仅能混合颜色，还可以给 varying 传递任何值，OpenGL 会选取属于那条直线的两个值，或者属于那个三角形的三个值，并平滑地在那个基本图元上混合这些值，每个片段都会有一个不同的值。这种混合是使用线性插值（linear interpolation）实现的。要了解它是怎么工作的，让我们首先以一条直线为例开始讲解。

沿着一条直线做线性插值

假设有一条直线，它有一个红色顶点和一个绿色顶点，我们要从一个向另外一个混合颜色。混合后的颜色看上去如右图所示。



在这条直线的左边，每个片段的颜色更多地呈红色；随着向右边前进，那些片段的红色分量逐渐减少，在中间处，它们处于红色和绿色之间；随着与绿色顶点越来越近，片段也变得越来越绿了。

我们可以看到每种颜色分量都随着直线长度线性缩放。因为这条线段的左侧顶点是红色，而右侧顶点是绿色，它的左端就是 100% 的红色，中间是 50% 的红色，而右端是 0% 的红色。

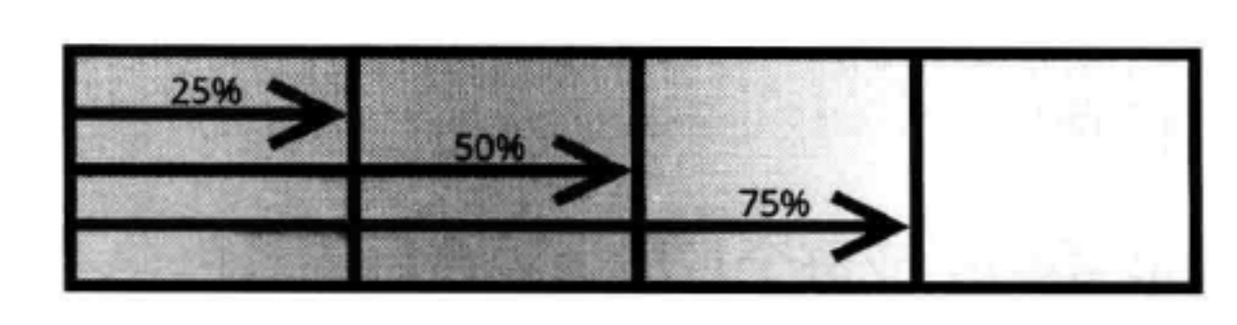
绿色的变化也是一样的。因为左侧顶点是红色的，而右侧顶点是绿色的，这个线段的左端就是 0% 的绿色，中间是 50% 的绿色，而右端就是 100% 的绿色。

一旦我们把这两个颜色叠加在一起，最终就得到一条混合后的直线。

这就是线性插值的基本解释。每种颜色的强度依赖于每个片段与包含那个颜色的顶点的距离。

为了计算这些，我们可以用顶点 0 和顶点 1 的值，计算出当前片段对应的距离比（distance

ratio）。距离比仅仅是 0 到 100 之间的百分比，0% 是左边的顶点，而 100% 就是右边的顶点。当我们从左向右移动，这个距离比例也会从 0% 向 100% 线性增加。这是几个距离比的例子：



要使用线性插值计算实际混合后的值，我们可以使用下面的公式：

blended_value = (vertex_0_value * (100% – distance_ratio)) +
(vertex_1_value * distance_ratio)

这个计算公式是应用于每个分量的，因此，如果我们处理颜色值，这个计算就会分别应用在红色、绿色、蓝色和阿尔法分量上，计算的结果合并成一个新的颜色值。

让我们用这条直线的例子验证一下这个公式。设 vertex_0_value 为红色，它的 RGB 值是（1, 0, 0），设 vertex_1_value 为绿色，它的 RGB 值是（0, 1, 0）。计算一下这条线段上几个位置的颜色（见表 4-1）。

表 4-1 线性插值公式的例子

位置	距离比	公式
最左端	0%	(vertex_0_value * (1 – distance_ratio)) + (vertex_1_value * distance_ratio) = ((1, 0, 0) * (100% – 0%)) + ((0, 1, 0) * 0%) = ((1, 0, 0) * 100%) = (1, 0, 0)（红色）
直线的四分之一处	25%	(vertex_0_value * (1 – distance_ratio)) + (vertex_1_value * distance_ratio) = ((1, 0, 0) * (100% – 25%)) + ((0, 1, 0) * 25%) = ((1, 0, 0) * 75%) + ((0, 1, 0) * 25%) = (0.75, 0, 0) + (0, 0.25, 0) = (0.75, 0.25, 0)（大红）
中间	50%	(vertex_0_value * (1 – distance_ratio)) + (vertex_1_value * distance_ratio) = ((1, 0, 0) * (100% – 50%)) + ((0, 1, 0) * 50%) = ((1, 0, 0) * 50%) + ((0, 1, 0) * 50%) = (0.5, 0, 0) + (0, 0.5, 0) = (0.5, 0.5, 0)（半红半绿）
直线的四分之三处	75%	(vertex_0_value * (1 – distance_ratio)) + (vertex_1_value * distance_ratio) = ((1, 0, 0) * (100% – 75%)) + ((0, 1, 0) * 75%) = ((1, 0, 0) * 25%) + ((0, 1, 0) * 75%) = (0.25, 0, 0) + (0, 0.75, 0) = (0.25, 0.75, 0)（大绿）

（续）

位置	距离比	公式
最右端	100%	(vertex_0_value * (1 – distance_ratio)) + (vertex_1_value * distance_ratio) = ((1, 0, 0) * (100% – 100%)) + ((0, 1, 0) * 100%) = ((1, 0, 0) * 0%) + ((0, 1, 0) * 100%) = (0, 1, 0)（绿色）

要注意到，任何时候两个颜色的权重加起来都是 100%。如果红色是 100%，绿色就是 0%；如果红色是 50%，那绿色就是 50%。

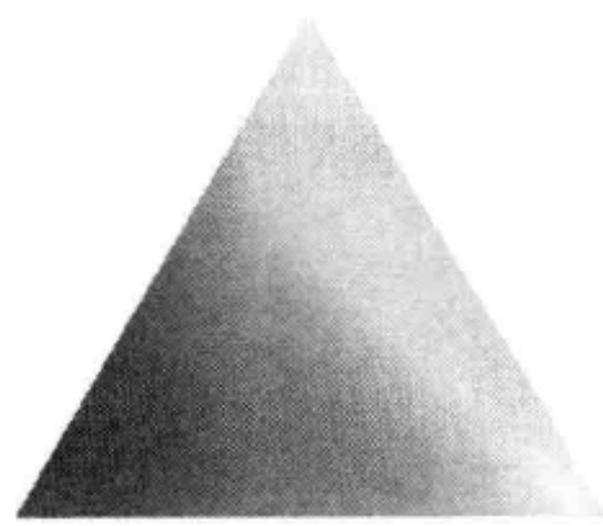
使用一个 varying，我们就可以把任何两种颜色混合在一起。当然，这不只限于颜色：任何其他属性也可以应用插值技术。

既然我们知道了一条直线上的线性插值是如何工作的，让我们继续阅读，看看在一个三角形上它是如何工作的。

在一个三角形表面上混合

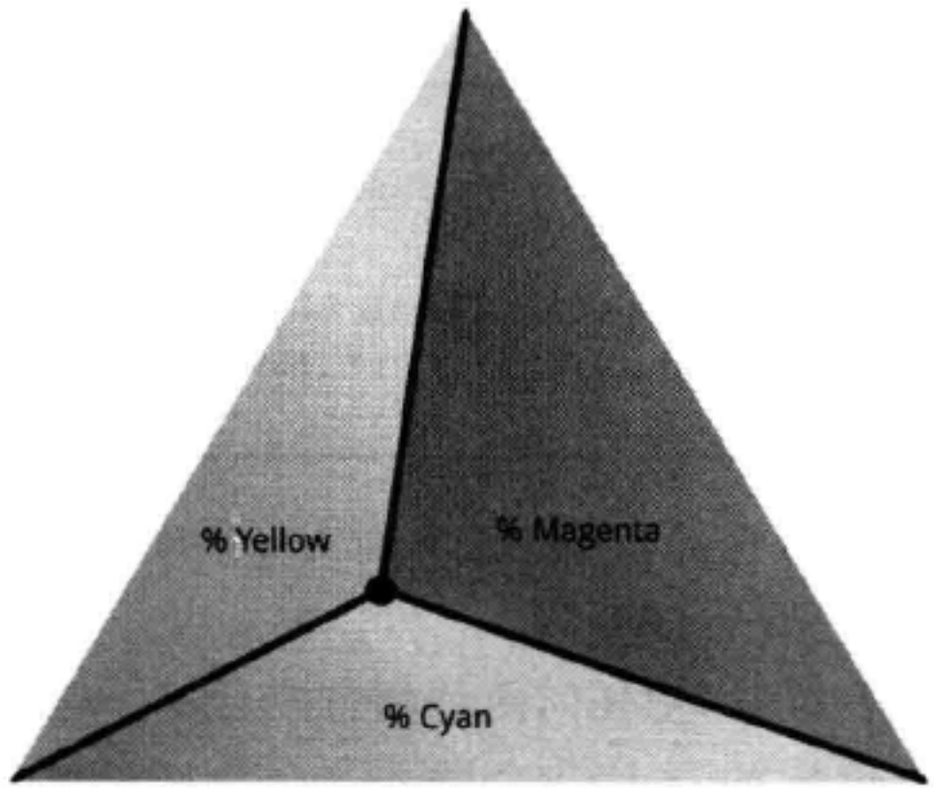
当我们只处理两个点的时候，阐明线性插值是怎么工作的并不困难；我们知道，从某个颜色的一个顶点到另外一个顶点，其比例从 100% 到 0% 缩减，所有按比例缩减的颜色合在一起就得到了最后的颜色。

在一个三角形上的线性插值也是一样的工作原理，但是现在需要处理三个点和三种颜色。让我们看一个直观的例子：



这个三角形与三种颜色有关联：顶端顶点是青色，左端顶点是品红色，右端顶点是黄色。让我们把这个三角形按每个顶点衍生出来的颜色进行分解：

就像那条直线一样，每个颜色在接近它的顶点处都是最强的，向其他顶点移动就会变暗。我们同样用比例确定每种颜色的相对权重，但这次要使用面积的比例，而不是长度。



对于这个三角形内任何给定的点，从那个点向每个顶点所对应的点画一条直线就可以生成三个内部三角形。这三个内部三角形的面积的比例决定了那个点上每种颜色的权重。比如，那个点上黄色的强度就取决于黄色顶点相对的那个内部三角形的面积。距离黄色顶点越近的点，它相对的三角形就越大，在那个点的片段就越显黄。

与直线一样，这些权重之和也总是等于 100%。可以使用下面的公式计算三角形内任何一个点的颜色分量：

```
blended_value = (vertex_0_value * vertex_0_weight)
              + (vertex_1_value * vertex_1_weight)
              + (vetex_2_value * (100% – vertex_0_weight – vertex_1_weight))
```

我们已经理解了它在直线上是怎么工作的，在这种情况下，我们就不需要为此举出具体的例子了。原理是一样的，只是这次要处理三个点而不是两个。

4.4 使用新的颜色属性渲染

因为我们已经给数据增加了一个颜色属性，并且更新了顶点和片段着色器，让它们使用这个属性，下一步就是去掉使用 unifom 传递颜色的旧代码，并告诉 OpenGL 把颜色作为一个顶点属性读入。

4.4.1 更新常量

让我们在 AirHockeyRenderer 的顶部加入如下常量：

AirHockey2/src/com/airhockey/android/AirHockeyRenderer.java

```
private static final String A_COLOR = "a_Color";
private static final int COLOR_COMPONENT_COUNT = 3;
private static final int STRIDE =
    (POSITION_COMPONENT_COUNT + COLOR_COMPONENT_COUNT) * BYTES_PER_FLOAT;
```

我们也需要一个新的成员变量：

AirHockey2/src/com/airhockey/android/AirHockeyRenderer.java
```
private int aColorLocation;
```

现在可以去掉那些旧的常量和与 u_Color 相关的变量了。

你有没有注意到我们增加了一个名为“STRIDE”的特殊常量？因为我们现在在同一个数据数组里面既有位置又有颜色属性，OpenGL 不能再假定下一个位置是紧跟着前一个位置的。一旦 OpenGL 读入了一个顶点的位置，如果它想读入下一个顶点的位置，它不得不跳过当前顶点的颜色数据。我们要用那个跨距（stride）告诉 OpenGL 每个位置之间有多少个字节，这样它就知道需要跳过多少了。

在图 4-2 中，我们可以看到一个直观的例子，它是关于顶点数组当前是怎么存储数据的。那个跨距告诉 OpenGL 每个位置或者每个颜色的间隔。如果不用跨距，我们可以给每个属性用多个顶点数组，如图 4-3 所示。然而，把所有数据放到一个单一数组里通常更加高效，只有当我们想定期更新所有的颜色或所有的位置时，使用多个数组可能才更有意义。

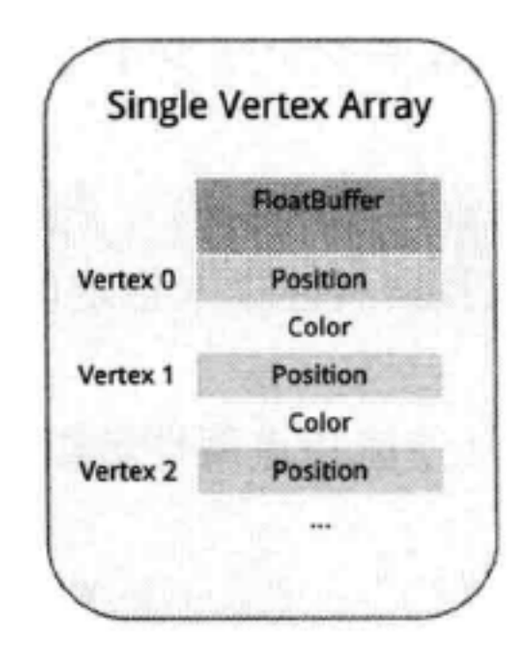


图 4-2　有多个属性的单一顶点数组

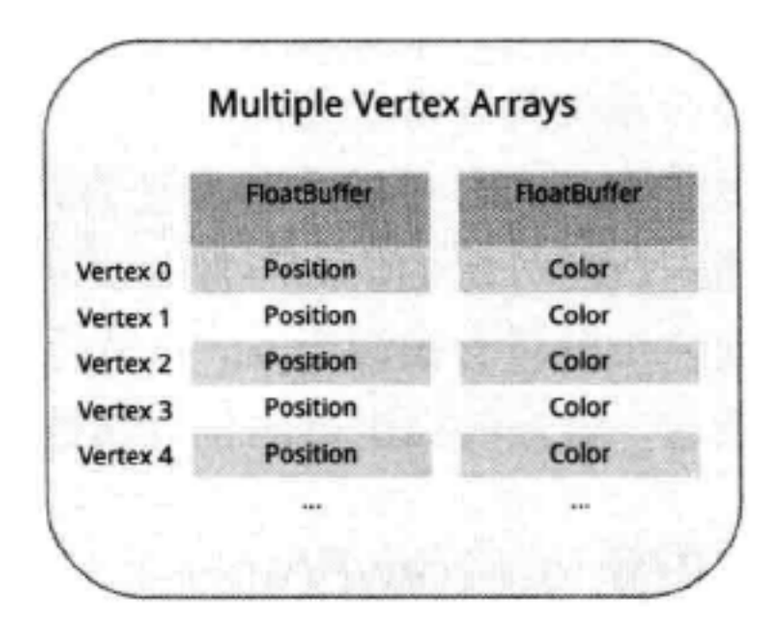


图 4-3　多个顶点数组，其中每个数组都有一个属性

4.4.2　更新 onSurfaceCreated()

下一步是更新 onSurfaceCreated()，让它体现这个新的颜色属性。我们首先需要获取新属性的位置，因此，去掉与 u_Color 相关的代码，并加入如下代码：

AirHockey2/src/com/airhockey/android/AirHockeyRenderer.java
```
aColorLocation = glGetAttribLocation(program, A_COLOR);
```

我们也应该更新 glVertexAttribPointer() 的调用，让它加入那个跨距：

AirHockey2/src/com/airhockey/android/AirHockeyRenderer.java
```
glVertexAttribPointer(aPositionLocation, POSITION_COMPONENT_COUNT, GL_FLOAT,
    false, STRIDE, vertexData);
```

现在，我们可以加入代码告诉 OpenGL 把顶点数据与着色器中的 a_Color 关联起来。在 onSurfaceCreated() 尾部加入如下代码：

```
AirHockey2/src/com/airhockey/android/AirHockeyRenderer.java
vertexData.position(POSITION_COMPONENT_COUNT);
glVertexAttribPointer(aColorLocation, COLOR_COMPONENT_COUNT, GL_FLOAT,
    false, STRIDE, vertexData);

glEnableVertexAttribArray(aColorLocation);
```

这些代码比较重要，因此让我们花点时间仔细地理解每一行代码：

1. 首先，我们把 vertexData 的位置设为 POSITION_COMPONENT_COUNT，这个值被设为 2。为什么执行这一步呢？这是因为，当 OpenGL 开始读入颜色属性时，我们要它从第一个颜色属性开始，而不是第一个位置属性。

   当需要跳过第一个位置时，我们就要把位置分量的大小计算在内；把那个位置设置为 POSITION_COMPONENT_COUNT，缓冲区的指针就被调整到第一个颜色属性的位置了。相反，如果我们把位置设为 0，OpenGL 就会把位置当作颜色读进来了。

2. 接下来，我们调用 glVertexAttribPointer() 把颜色数据和着色器中的 a_Color 关联起来。那个跨距告诉 OpenGL 每个颜色之间有多少个字节，这样，当需要读入所有顶点的颜色时，它就知道要读取下一个顶点的颜色需要跳过多少个字节了。跨距以字节为单位是非常重要的。

   尽管 OpenGL 中的一种颜色有四个分量（红色、绿色、蓝色和阿尔法），我们并不必指定所有分量的值。不像 uniform，OpenGL 会用默认值替换属性中未指定值的分量：前三个分量会被设为 0，而最后一个分量会被设为 1。

3. 最后，就像前面讲过的位置属性一样，我们要为颜色属性使能顶点属性数组。

4.4.3 更新 onDrawFrame

我们就只剩下一件事情要做了：更新 onDrawFrame()。我们所需要做的就是删除 glUniform4f() 调用，因为我们不再需要它们了。既然我们已经把顶点数据和 a_Color 关联起来了，只需要调用 glDrawArrays() 即可，OpenGL 会自动从顶点数据里读入颜色属性。

4.4.4 把代码整合起来

运行这个程序，看一下我们得到了什么；它看起来应该如图 4-4 所示。

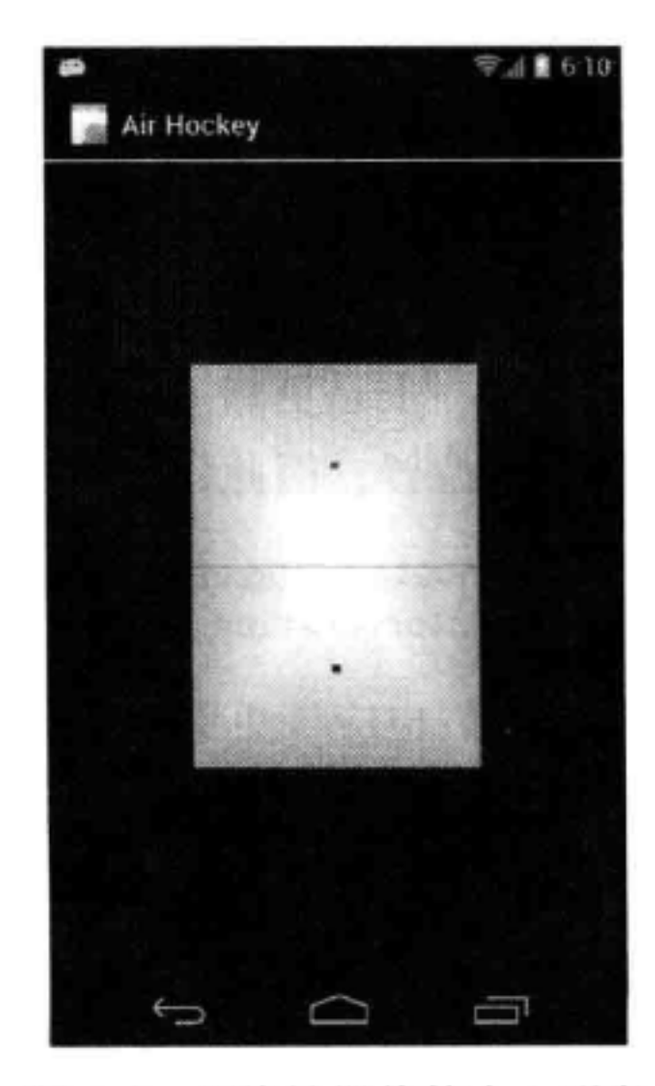


图 4-4 用线性插值使中心明亮

这个空气曲棍球桌子看起来比以前更好了，而且我们也的确看到了桌子中心比边缘处更加明亮。可是，我们也让每个三角形的形状变得突出了。之所以会这样，是因为线性插值的方向是顺着三角形的，因此当三角形内部看起来平滑的

时候，有时候我们会看到在一个三角形结束的地方接上另外一个三角形了。

为了减少或消除这种效果，我们可以使用更多的三角形或者使用一个光照算法并以每个片段为基础计算颜色。我们会在第 13 章中学习更多关于光照算法的内容。

4.5　小结

因为我们已经有了一个基本的框架，给每个顶点增加颜色并不太困难。为此，我们给顶点数据和顶点着色器增加了一个新的属性，并且告诉 OpenGL 如何使用跨距读入数据。接着我们学习了如何使用一个 varying 在三角形平面上进行插值。

一个需要记住的重要内容：当传递属性数据时，我们要确保给分量计数和跨距传递正确的值。如果它们错了，我们最终可能会看到一个混乱的屏幕，甚至崩溃。

4.6　练习

看看你是否能给跨越屏幕中间的那条线加入一些颜色插值。还有一个有点挑战性的练习，如何改变构成空气曲棍球桌子的三角形？使它们的边缘不要太明显。提示：可尝试给那个三角形扇增加更多的三角形。

一旦你完成了这些练习，我们将开始学习向量（vector）和矩阵（matrix），以及如何修复一个令人讨厌的问题，这个问题在屏幕从竖屏转到横屏时就会出现。

Chapter 3

第 5 章

调整屏幕的宽高比

你可能还没注意到，这个空气曲棍球桌子当前的宽高比有问题。为了看看发生了什么，现在打开前一章开发的项目，继续在设备或模拟器上运行那个应用。一旦运行起来了，把你的设备从竖屏旋转到横屏（如果使用的是模拟器，按“Ctrl+F12”键）。

在竖屏模式下，那个应用程序看起来应该如图 5-1 所示，图 5-2 是其横屏模式的情况。

这个桌子在横屏模式情况下被压扁了！这之所以会发生，是因为我们之前直接把坐标传递给 OpenGL 时，没有考虑屏幕的宽高比。每个二维和三维应用程序都有一个共同的大问题：怎样决定在屏幕上显示什么以及如何针对屏幕尺寸做出调整？这个问题有一个通用的解决方案：在 OpenGL 里，我们可以使用投影把真实世界的一部分映射到屏幕上，以这种方式映射会使它在不同的屏幕尺寸或方向上看起来总是正确的。考虑到有如此大量的设备，能够适应这些设备是很重要的。

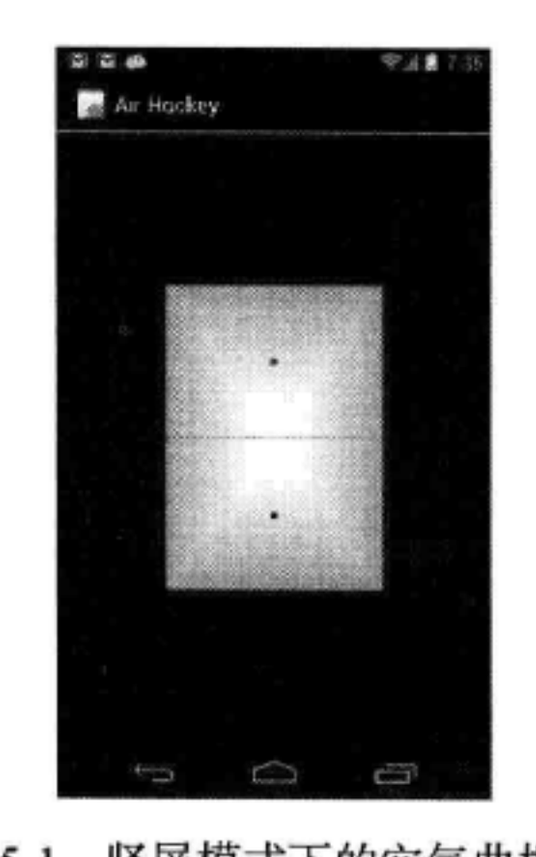


图 5-1　竖屏模式下的空气曲棍球

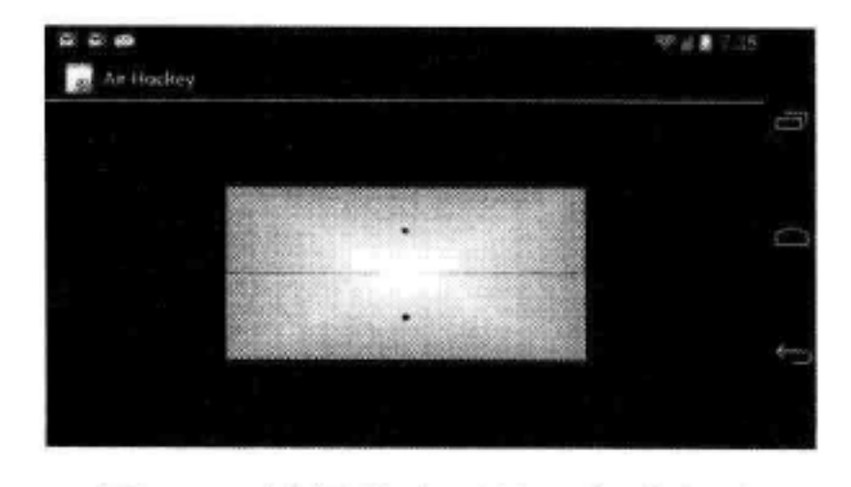


图 5-2　横屏模式下的空气曲棍球

在本章，我们要了解为什么这个桌子看起来被压扁了，以及怎样用投影修复这个问题。下面是本章的开发计划。

- 首先，我们要复习一些基本的线性代数内容，

并学习矩阵（matrix）与向量（vector）是如何相乘的。

- 然后，要学习如何通过矩阵定义和使用投影，这能让我们弥补屏幕方向变化的影响，这样桌子看起来就不会是扁的。

如上一章一样，让我们复制上一章的项目作为一个新项目，并以此开始。我们把这个新项目称为“AirHockeyOrtho”。如果你需要快速地复习一下，请遵循 2.2 节的那些操作顺序。

5.1　宽高比的问题

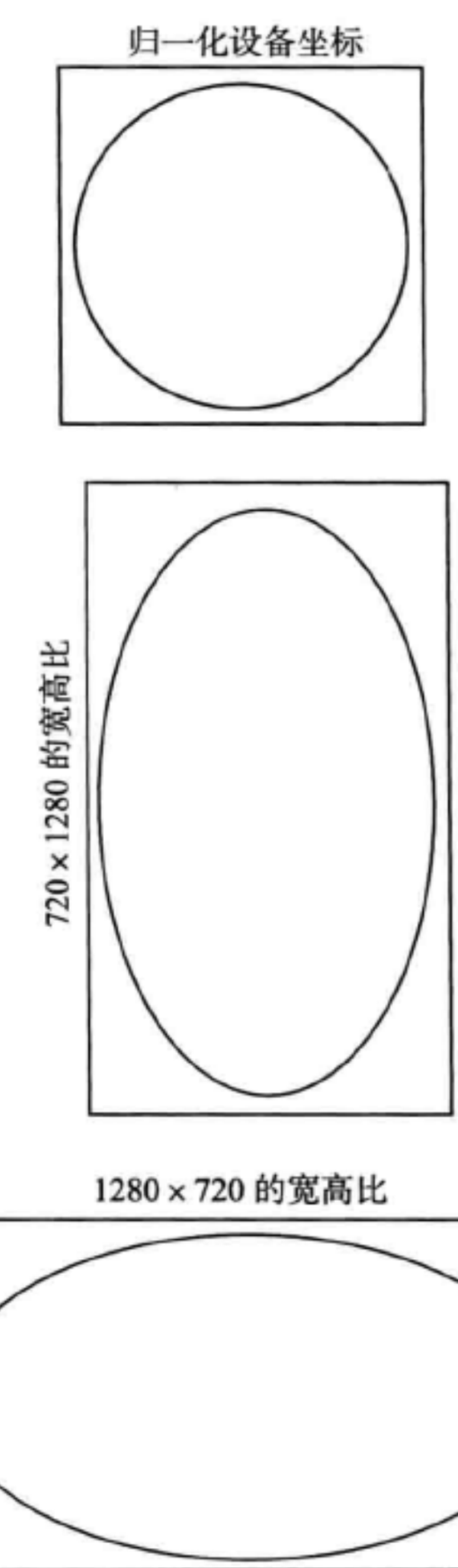
我们现在相当熟悉这样一个事实，在 OpenGL 里，我们要渲染的一切物体都要映射到 x 轴和 y 轴上 [−1, 1] 的范围内，对 z 轴也是一样的。这个范围内的坐标被称为归一化设备坐标，其独立于屏幕实际的尺寸或形状。

不幸的是，因为它们独立于实际的屏幕尺寸，如果直接使用它们，我们就会遇到问题，例如在横屏模式情况下被压扁的桌子。

假设实际的设备分辨率以像素为单位是 1280 × 720，这在新的 Android 设备上是一个常用的分辨率。为了使讨论更加容易，让我们也暂时假定 OpenGL 占用整个显示屏。

如果设备是在竖屏模式下，那么 [−1, 1] 的范围对应 1280 像素高，却只有 720 像素宽。图像会在 x 轴显得扁平，如果在横屏模式，同样的问题也会发生在 y 轴上。

归一化设备坐标假定坐标空间是一个正方形，如右图所示。



然而，因为实际的视口（viewport）可能不是一个正方形，图像就会在一个方向上被拉伸，在另外一个方向上被压扁。在一个竖屏设备上，归一化设备坐标上定义的图像看上去就是在水平方向上被压扁了：

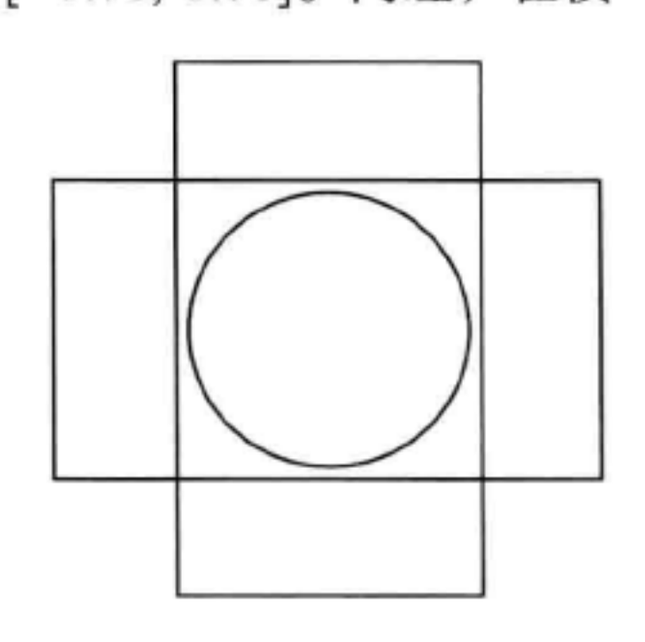
在横屏模式下，同样的图像就在另一个方向上看起来是压扁的。

适应宽高比

我们需要调整坐标空间，以使它把屏幕的形状考虑在内，可行的一个方法是把较小的

范围固定在 [−1, 1] 内，而按屏幕尺寸的比例调整较大的范围。

举例来说，在竖屏情况下，其宽度是 720，而高度是 1280，因此我们可以把宽度范围限定在 [−1, 1]，并把高度范围调整为 [−1280/720, 1280/720] 或 [−1.78, 1.78]。同理，在横屏模式情况下，把高度范围设为 [−1.78, 1.78]，而把高度范围设为 [−1, 1]。



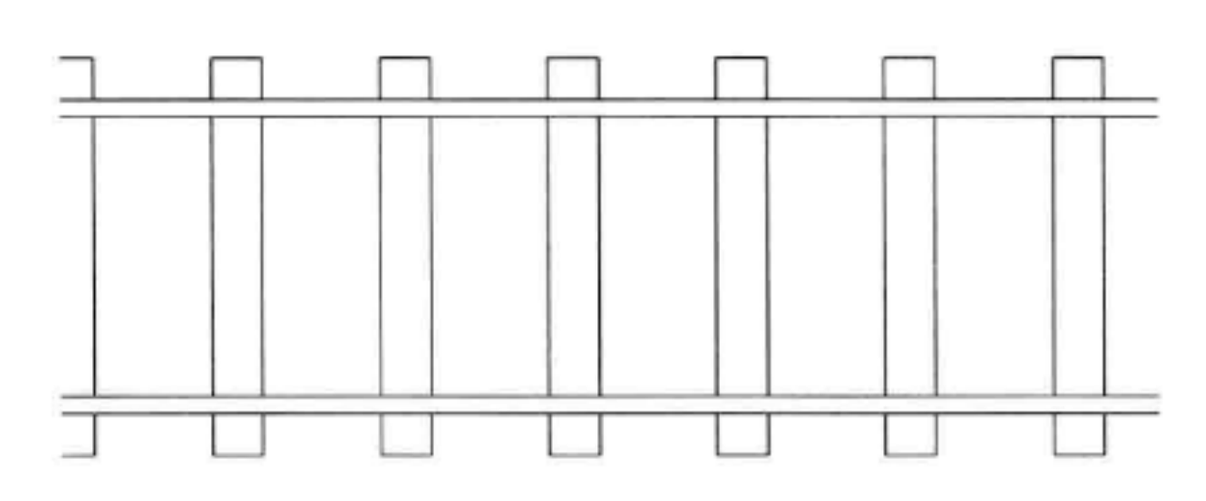
通过调整已有的坐标空间，最终会改变我们可用的空间。

通过这个方法，不论是竖屏模式还是横屏模式，物体看起来就都一样了。

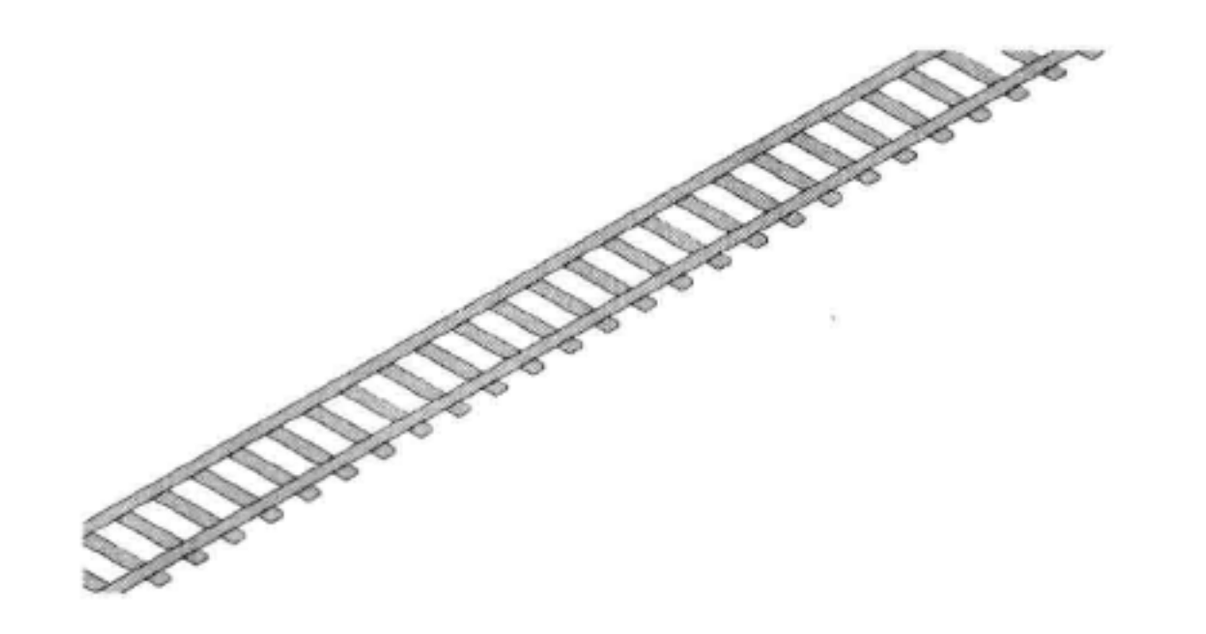
5.2 使用虚拟坐标空间

调整坐标空间，以便我们把屏幕方向考虑进来，我们需要停止直接在归一化设备坐标上工作，而开始在虚拟坐标空间里工作。接下来，我们需要找到某种可以把虚拟空间坐标转换回归一化设备坐标的方法，让 OpenGL 可以正确地渲染它们。这种转换应该把屏幕方向计算在内，以使空气曲棍球桌子在竖屏模式和横屏模式情况下看上去都是正确的。

我们想要进行的操作叫作正交投影（orthographic projection）。使用正交投影，不管多远或多近，所有的物体看上去大小总是相同的。为了更好地理解这种投影是做什么的，想象一下，在我们的场景中有一个火车轨道，直接从空中俯瞰，这些轨道看起来可能是这样的：



还有一种特殊类型的正交投影，被称为等轴测投影（isometric projection），它是从侧角观察的一种正交投影。正如在一些城市模拟和策略游戏中看到的，这种类型的投影能用来重新创建一个经典的三维角。



从虚拟坐标回到归一化设备坐标

当我们使用正交投影把虚拟坐标变换回归一化设备坐标时，实际上定义了三维世界内部的一个区域。在这个区域内的所有东西都会显示在屏幕上，而区域外的所有东西都会被裁剪掉。在下面的图像中，我们能在一个封闭的立方体内看到一个简单的场景（见图 5-3 ）。

当我们使用一个正交投影矩阵把这个立方体映射到屏幕上时，就会看到如图 5-4 所示的正交投影。

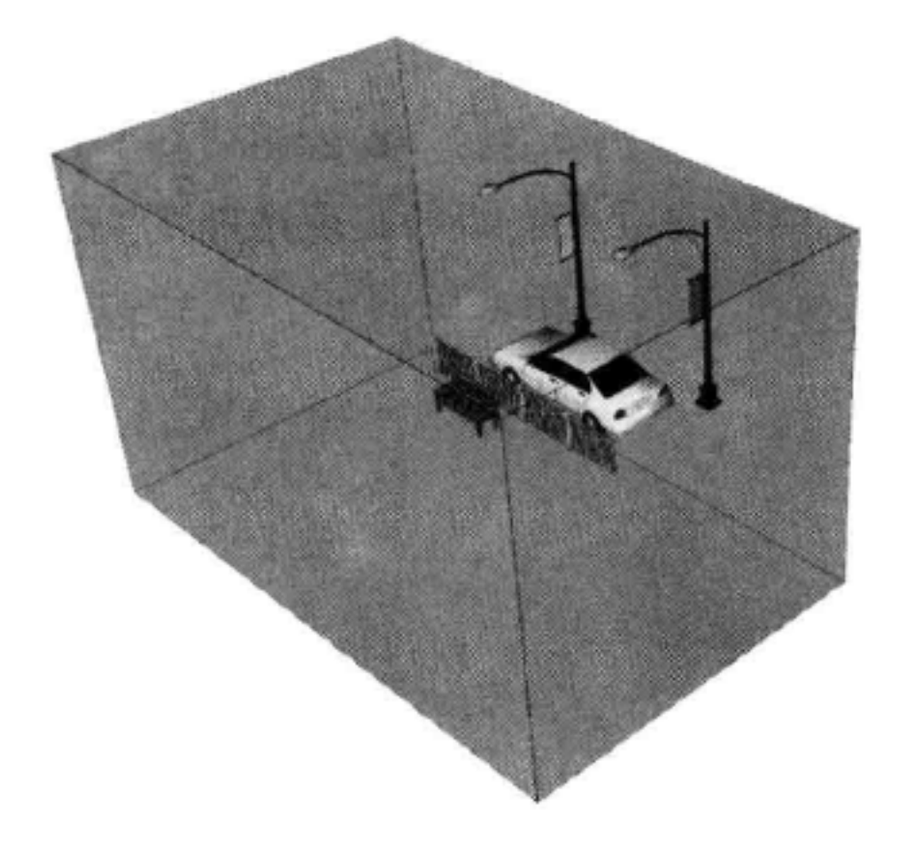


图 5-3　一个正交立方体内的场景

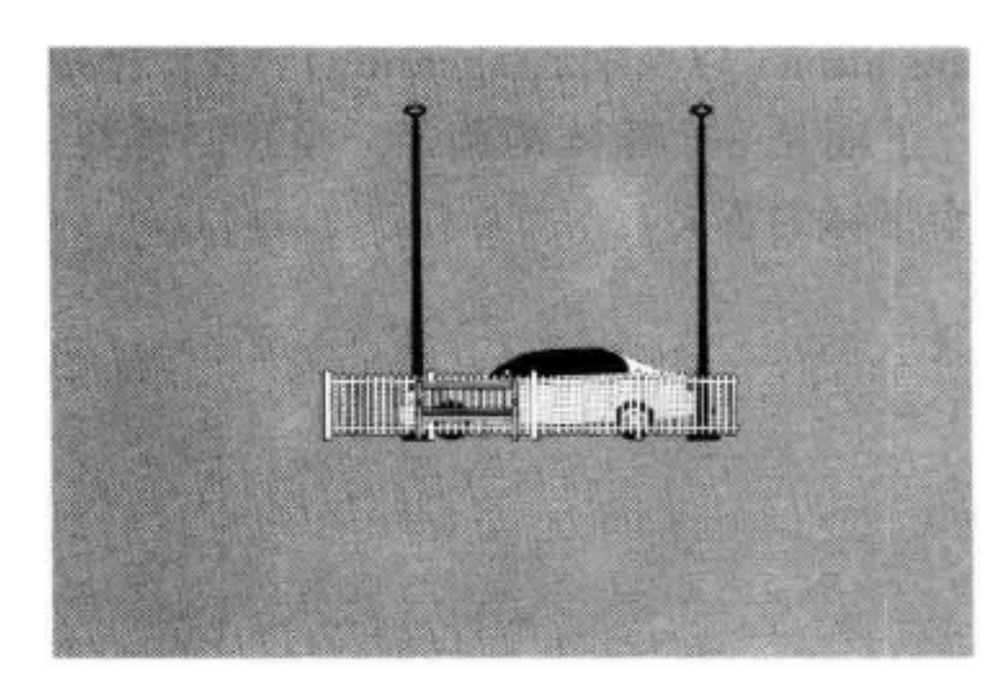


图 5-4　一个正交投影

利用正交投影矩阵改变立方体的大小，以使我们可以在屏幕上看到或多或少的场景。我们也能改变立方体的形状弥补屏幕的宽高比的影响。

5.3　线性代数基础

OpenGL 大量地使用了向量和矩阵，矩阵的最重要的用途之一就是建立正交和透视投影。其原因之一是，从本质上来说，使用矩阵做投影只涉及对一组数据按顺序执行大量的加法和乘法，这些运算在现代 GPU 上执行得非常快。

让我们回到你在高中或者大学的时代，复习一下线性代数的基本知识。如果你不记得了、忘得差不多了或者从来没上过这门课，也不需要担心，我们会一起温习这些基础数学。一旦我们理解了这些基础数学，就可以开始学习如何用矩阵做正交投影。

5.3.1　向量

一个向量（vector）是一个有多个元素的一维数组。在 OpenGL 里，一个位置通常是一个四元素向量，颜色也是一样。我们使用的大多数向量一般都有四个元素。在下面的例子中，我们可以看到一个位置向量，它有一个 x、一个 y、一个 z 和一个 w 分量。

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

我们会在第 6 章中详细解释这个 w 分量。

5.3.2 矩阵

一个矩阵（Matrix）是一个有多个元素的二维数组。在 OpenGL 里，我们一般使用矩阵作向量投影，如正交或者透视投影，并且也用它们使物体旋转（rotation）、平移（translation）以及缩放（scaling）。我们把矩阵与每个要变换的向量相乘即可实现这些变换。

下面是一个矩阵的例子。当我们看到一个矩阵如何与一个向量相乘的时候，就会明白那个下标的意义。

$$\begin{bmatrix} x_x & x_y & x_z & x_w \\ y_x & y_y & y_z & y_w \\ z_x & z_y & z_z & z_w \\ w_x & w_y & w_z & w_w \end{bmatrix}$$

5.3.3 矩阵与向量乘法

要让矩阵乘以一个向量，我们把矩阵放在左边，向量放在右边。接下来我们从矩阵的第一行开始，那一行的第一个分量乘以那个向量的第一个分量，那一行的第二个分量乘以那个向量的第二个分量，以此类推；然后，我们把那一行的所有结果相加得出其最终结果的第一个分量。

下面是一个完整的矩阵与向量乘法的例子：

$$\begin{bmatrix} x_x & x_y & x_z & x_w \\ y_x & y_y & y_z & y_w \\ z_x & z_y & z_z & z_w \\ w_x & w_y & w_z & w_w \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x_x x + x_y y + x_z z + x_w w \\ y_x x + y_y y + y_z z + y_w w \\ z_x x + z_y y + z_z z + z_w w \\ w_x x + w_y y + w_z z + w_w w \end{bmatrix}$$

对于第一行，我们让 x_x 和 x 相乘、x_y 和 y 相乘、x_z 和 z 相乘以及 x_w 和 w 相乘，然后把所有四个结果加起来得到这个结果的 x 分量。

因为我们看到了矩阵下标的作用，它应该显得更有意义了。矩阵第一行的所有四个分量都影响了那个结果 x，第二行的所有 4 个分量都影响了那个结果 y，以此类推。在矩阵的每一行内，第一个分量与向量的 x 相乘，第二个分量也与向量的 y 相乘，以此类推。

5.3.4　单位矩阵

我们看一个有实际数值的例子。我们从一个简单的矩阵开始，它被称为单位矩阵（identity matrix）。单位矩阵如下图所示：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

它之所以被称为单位矩阵，是因为这个矩阵乘以任何向量总是得到与原来相同的向量。就像把任何数字乘以 1 都会得到原来的数字一样。

例如，用单位矩阵乘以一个包含 1、2、3 和 4 的向量：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1\times1 + 0\times2 + 0\times3 + 0\times4 \\ 0\times1 + 1\times2 + 0\times3 + 0\times4 \\ 0\times1 + 0\times2 + 1\times3 + 0\times4 \\ 0\times1 + 0\times2 + 0\times3 + 1\times4 \end{bmatrix}$$

对于第一行，我们把向量的第一个分量乘以 1，并忽略其他与 0 相乘的分量；对于第二行，同样这么做，但我们保留了这个向量的第二个分量。这样计算的结果证明了这个答案与原来的向量是一模一样的。

让我们简化那些乘法，并把这些结果加在一起。计算结果为：

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

5.3.5　使用平移矩阵

既然理解了单位矩阵，让我们看一个非常简单的矩阵类型——平移矩阵（translation matrix），它在 OpenGL 里十分常用。使用这种类型的矩阵，我们可以把一个物体沿着指定的距离移动。这个矩阵看起来与单位矩阵差不多，但在右侧指定了三个额外的元素。

$$\begin{bmatrix} 1 & 0 & 0 & x_{\text{translation}} \\ 0 & 1 & 0 & y_{\text{translation}} \\ 0 & 0 & 1 & z_{\text{translation}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

让我们看一个位置（2, 2）的例子，这个位置 z 的默认值是 0，w 的默认值是 1。我们要把这个向量沿 x 轴平移 3，沿 y 轴也平移 3，因此，把 $x_{\text{translation}}$ 赋值为 3，$y_{\text{translation}}$ 也赋值为 3。

结果是这样的：

$$\begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1\times2 + 0\times2 + 0\times0 + 3\times1 \\ 0\times2 + 1\times2 + 0\times0 + 3\times1 \\ 0\times2 + 0\times2 + 1\times0 + 0\times1 \\ 0\times2 + 0\times2 + 0\times0 + 1\times1 \end{bmatrix}$$

简化这些乘法之后，我们得到这个：

$$\begin{bmatrix} 2 + 0 + 0 + 3 \\ 0 + 2 + 0 + 3 \\ 0 + 0 + 0 + 0 \\ 0 + 0 + 0 + 1 \end{bmatrix}$$

把这些结果加在一起得到最终的答案：

$$\begin{bmatrix} 5 \\ 5 \\ 0 \\ 1 \end{bmatrix}$$

这个位置正是我们所期望的（5, 5）。

这个矩阵之所以可行，是原因它构建在单位矩阵之上，所以，它做的第一件事就复制原来的向量。因为平移分量与 w 相乘，并且我们通常把一个位置的 w 分量指定为 1（记住，如果我们没有指定 w 分量，OpenGL 会默认把它设为 1），这个平移分量只是被加到结果中。

这里要注意的是，w 的影响是非常重要的。在下一章，我们会学习透视投影，在这样的投影操作之后，一个坐标的 w 分量的值可能不是 1。如果我们对一个坐标投影以后，其 w 分量的值不是 1 了，对其做平移或其他类型的变换时，我们会遇到麻烦，物体会被扭曲。

以上关于向量和矩阵数学的内容帮我们打好了基础；可以继续学习如何定义正交投影了。

5.4 定义正交投影

要定义正交投影，我们将使用 Android 的 Matrix 类，它在 android.opengl 包中。这个类有一个称为 orthoM() 的方法，它可以为我们生成一个正交投影。正如我们在 5.1.1 节中讨论的，我们将使用这个投影调整坐标空间。我们也会马上看到，正交投影与平移矩阵是非常相似的。

我们看一下 orthoM() 的所有参数：

```
orthoM(float[] m, int mOffset, float left, float right, float bottom, float top, float near, float far)
```

- float[] m：目标数组，这个数组的长度至少有 16 个元素，这样它才能存储正交投影矩阵；

- int mOffset：结果矩阵起始的偏移值；
- float left: x 轴的最小范围；
- float right: x 轴的最大范围；
- float bottom: y 轴的最小范围；
- float top：y 轴的最大范围；
- float near：z 轴的最小范围；
- float far：z 轴的最大范围。

当我们调用这个方法的时候，它应该产生下面的正交投影矩阵：

$$\begin{bmatrix} \frac{2}{\text{right}-\text{left}} & 0 & 0 & -\frac{\text{right}+\text{left}}{\text{right}-\text{left}} \\ 0 & \frac{2}{\text{top}-\text{bottom}} & 0 & -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} \\ 0 & 0 & \frac{-2}{\text{far}-\text{near}} & -\frac{\text{far}+\text{near}}{\text{far}-\text{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

不要让这些分数把你吓倒：这与 5.3.5 节中看到的平移矩阵非常相似。这个正交投影矩阵会把所有在左右之间、上下之间和远近之间的事物映射到归一化设备坐标中从 −1 到 1 的范围，在这个范围内的所有事物在屏幕上都是可见的。

主要的区别就是 z 轴有一个负值符号，它的效果是反转 z 坐标。这就意味着，物体离得越远，z 坐标的负值会越来越小。之所以这样完全是历史和传统的缘故。

左手与右手坐标系统

为了更好地理解 z 轴的问题，我们需要理解左手坐标系统与右手坐标系统之间的区别。想知道一个坐标系统是左手的还是右手的，你拿出一只手，把大拇指指向 x 轴的正值方向，然后把食指指向 y 轴的正值方向。

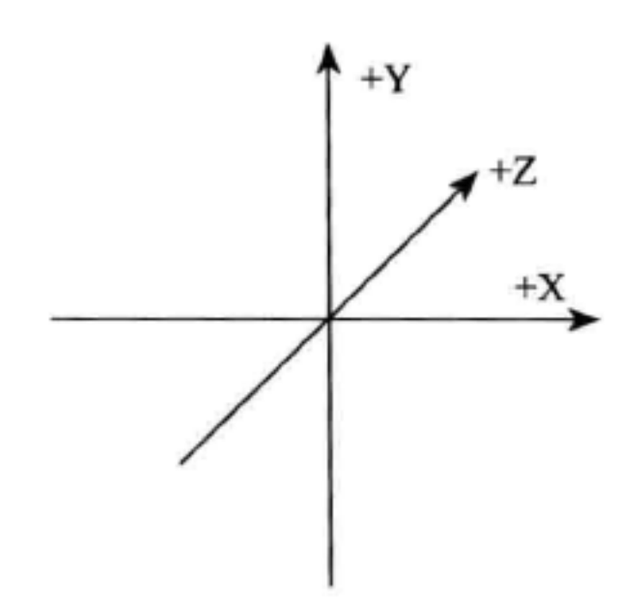


图 5-5　左手坐标系统

现在，把你的中指指向 z 轴。如果你需要用左手做这些，那你看到的就是一个左手坐标系统；如果你需要用右手，那你看到的就是一个右手坐标系统。

按照图 5-5 和图 5-6 试一下，把你的中指指向 z 轴，记住要把大拇指指向 x 轴正值方向、食指指向 y 轴正值方向。

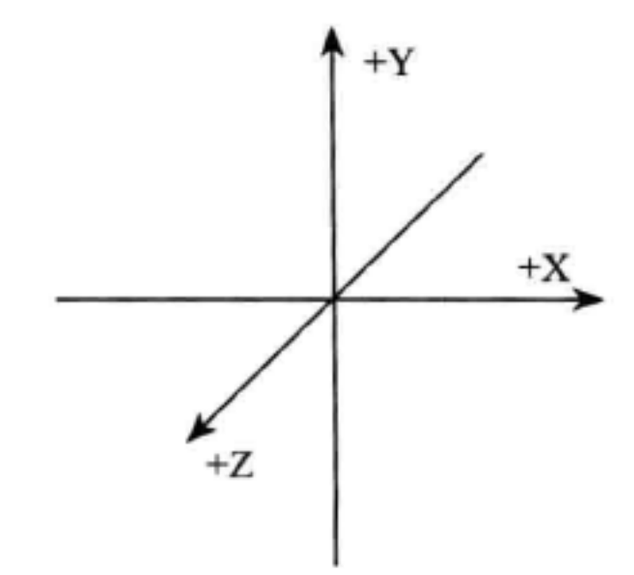


图 5-6　右手坐标系统

其实，左手还是右手的选择都没关系，只是一个简单的转换。归一化设备坐标使用的是左手坐标系统，而在 OpenGL 的早期版本，默认使用的却是右手系统，其使用 z 的负值增加表

示距离增加。这就是为什么 Android 的 Matrix 会默认生成反转 z 的矩阵。

如果你偏好在所有的地方都使用左手坐标系统，而不只是在归一化设备坐标系统里，那你就可以取消 orthoM() 在 z 轴上所做的转换。

现在我们对矩阵数学有了一个基本的体会，已经准备好在代码中添加正交投影了。如果你愿意看一些特殊的例子，你可以参考 A1.1 节的内容，一旦你准备好继续下一节，再回到这里。

5.5 加入正交投影

让我们更新代码加入一个正交投影并修复那个被压扁的桌子。

5.5.1 更新着色器

我们需要做的第一件事就是更新着色器，以便它用矩阵变换位置。打开 AirHockeyOrtho 项目（我们在本章开始处创建了这个项目）。打开 simple_vertex_shader.glsl 并更新如下：

AirHockeyOrtho/res/raw/simple_vertex_shader.glsl

```
uniform mat4 u_Matrix;

attribute vec4 a_Position;
attribute vec4 a_Color;

varying vec4 v_Color;

void main()
{
    v_Color = a_Color;

    gl_Position = u_Matrix * a_Position;
    gl_PointSize = 10.0;
}
```

我们添加了一个新的 uniform 定义“u_Matrix”，并把它定义为一个 mat4 类型，意思是这个 uniform 代表一个 4 × 4 的矩阵。更新位置赋值那一行：

AirHockeyOrtho/res/raw/simple_vertex_shader.glsl

```
gl_Position = u_Matrix * a_Position;
```

我们没有传递数组中定义的位置，而是传递那个位置与一个矩阵的乘积。这与我们在 5.3.3 节中讨论的数学运算是一样的。它也意味着顶点数组不用再被翻译为归一化设备坐标了，其将被理解为存在于这个矩阵所定义的虚拟坐标空间中。这个矩阵会把坐标从虚拟坐标空间变换回归一化设备坐标。

5.5.2 添加矩阵数组和一个新的 uniform

打开 AirHockeyRenderer，在类的顶部添加如下定义：

AirHockeyOrtho/src/com/airhockey/android/AirHockeyRenderer.java
```
private static final String U_MATRIX = "u_Matrix";
```

它保存顶点着色器中定义的那个新的 uniform 的名字；

我们还需要一个顶点数组存储那个矩阵：

AirHockeyOrtho/src/com/airhockey/android/AirHockeyRenderer.java
```
private final float[] projectionMatrix = new float[16];
```

我们也需要一个整型值用于保存那个矩阵 uniform 的位置：

AirHockeyOrtho/src/com/airhockey/android/AirHockeyRenderer.java
```
private int uMatrixLocation;
```

然后，我们只需要在 onSurfaceCreated() 中加入如下代码：

AirHockeyOrtho/src/com/airhockey/android/AirHockeyRenderer.java
```
uMatrixLocation = glGetUniformLocation(program, U_MATRIX);
```

5.5.3 创建正交投影矩阵

下一步就是更新 onSurfaceChanged()。在 glViewport() 调用后面加入如下代码行：

AirHockeyOrtho/src/com/airhockey/android/AirHockeyRenderer.java
```
final float aspectRatio = width > height ?
    (float) width / (float) height :
    (float) height / (float) width;

if (width > height) {
    // Landscape
    orthoM(projectionMatrix, 0, -aspectRatio, aspectRatio, -1f, 1f, -1f, 1f);
} else {
    // Portrait or square
    orthoM(projectionMatrix, 0, -1f, 1f, -aspectRatio, aspectRatio, -1f, 1f);
}
```

这段代码会创建一个正交投影矩阵，这个矩阵会把屏幕的当前方向计算在内。它会建立我们在 5.1.1 节中描述的那样的虚拟坐标空间。在 Android 里不只有一个 Matrix 类，因此你要确保导入了 android.opengl.Matrix。

我们首先计算了宽高比，它使用宽和高中的较大值除以宽和高中的较小值。不管是竖屏还是横屏，这个值都是一样的。

我们接下来调用 orthoM(float[] m, int mOffset, float left, float right, float bottom, float top, float near, float far)。如果在横屏模式下，我们会扩展宽度的坐标空间，这样它的取值范围就不是从 −1 到 1，而是从 -aspectRatio 到 aspectRatio；同时高度的取值范围保持从 −1 到 1。如果在竖屏模式，我们就反过来扩大高度的取值范围，保持宽度的取值范围为从 −1 到 1。

5.5.4 传递矩阵给着色器

在 AirHockeyRenderer 里做的最后的更改是给着色器传递那个正交投影矩阵。为此，在

onDrawFrame() 之中，我们在 glClear() 调用之后加入如下代码：

AirHockeyOrtho/src/com/airhockey/android/AirHockeyRenderer.java

```
glUniformMatrix4fv(uMatrixLocation, 1, false, projectionMatrix, 0);
```

5.5.5 观察我们的改动

我们现在可以运行这个应用程序，看看得到了什么？它在竖屏模式下应该如图 5-7 所示，在横屏模式下如图 5-8 所示。

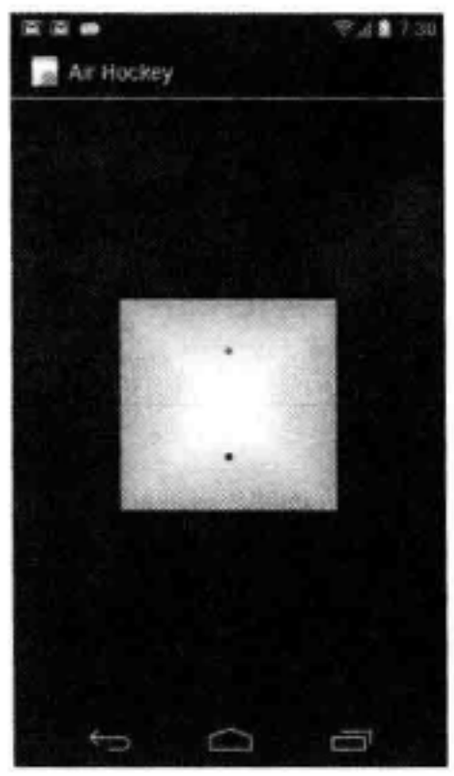


图 5-7 竖屏模式下使用正交投影的空气曲棍球

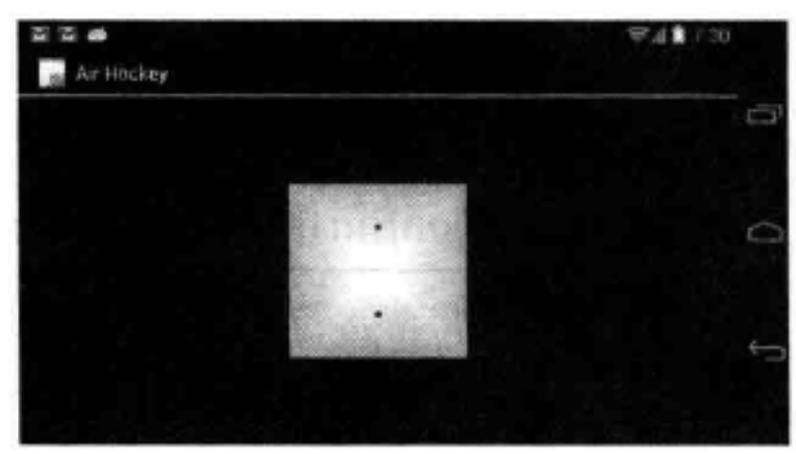


图 5-8 横屏模式下使用正交投影的空气曲棍球

5.5.6 更新桌子的数据

现在，这个桌子在竖屏和横屏模式下看上去都一样了，这是我们所期望的；但是它看起来还是有点不太对劲儿。发生了什么？即使我们现在看到的桌子不会再被压扁了，但这个原来被压扁的桌子在竖屏模式下看上去很窄。

让我们更新桌子的结构，让它高点（如图 5-9 和图 5-10 所示）；只更新 tableVerticesWithTriangles 中的 y 位置（第二列）来匹配那些数据，如下代码所示：

AirHockeyOrtho/src/com/airhockey/android/AirHockeyRenderer.java

```
float[] tableVerticesWithTriangles = {
    // Order of coordinates: X, Y, R, G, B

    // Triangle Fan
       0f,    0f,   1f,   1f,   1f,
    -0.5f, -0.8f, 0.7f, 0.7f, 0.7f,
     0.5f, -0.8f, 0.7f, 0.7f, 0.7f,
     0.5f,  0.8f, 0.7f, 0.7f, 0.7f,
    -0.5f,  0.8f, 0.7f, 0.7f, 0.7f,
    -0.5f, -0.8f, 0.7f, 0.7f, 0.7f,
```

```
    // Line 1
    -0.5f, 0f, 1f, 0f, 0f,
     0.5f, 0f, 1f, 0f, 0f,

    // Mallets
    0f, -0.4f, 0f, 0f, 1f,
    0f,  0.4f, 1f, 0f, 0f
};
```

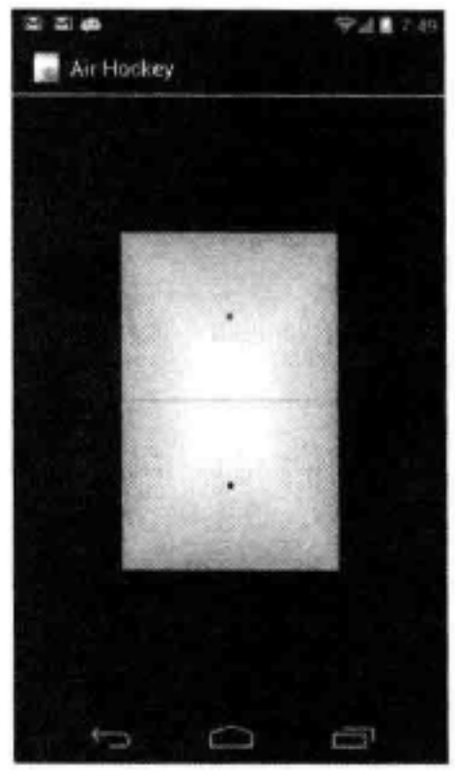


图 5-9　竖屏模式下有点高的空气曲棍球

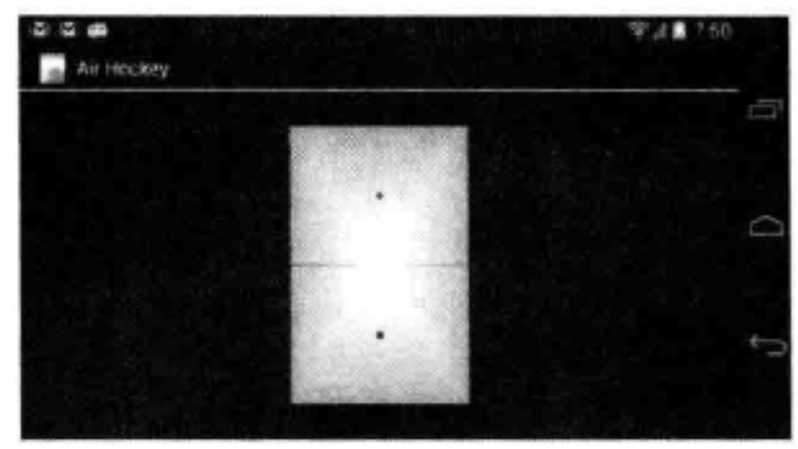


图 5-10　横屏模式下有点高的空气曲棍球

现在，它看上去更像我们期望的空气曲棍球桌子的形状了，并且在竖屏和横屏模式下都保持同样的形状。

5.6　小结

我们花费时间学习了一些线性代数背后的基础，并用它去理解矩阵与向量相乘的结果。接着，我们学习了如何定义正交投影矩阵，它允许我们重新定义坐标空间，我们使用这个矩阵修复了扭曲的问题，当我们从竖屏旋转到横屏模式时，这个问题总是发生。

如果有任何关于矩阵数学不清楚的地方，你可能要去复习 5.3 节；从这里到本书结束，我们将在向量和矩阵上花费越来越多的时间！

5.7　练习

尝试调整正交矩阵，让桌子看起来更大或者更小，并在屏幕上平移。为了实现这些，你需要调整传递给 orthoM() 的左、右、顶和底的值。

一旦你完成了这些练习，准备好在椅子上坐稳了，因为我们就要进入第三维了。

Chapter 6 第 6 章

进入第三维

想象一下你在一个游戏室里，站在一个空气曲棍球的前面，面向你的对手观察这个桌子。从你的视角看，这个桌子看上去是什么样的？你那端的桌子会显得较大，因为你是从一个角度向下观察这个桌子的，而不是直接从空中俯视。毕竟，没有人玩空气曲棍球游戏的时候站在桌子的上面并直接向下看。

OpenGL 很擅长渲染二维事物，但是当我们给这个组合加入第三维，这个世界确实变得灿烂了。在本章，我们会学习如何进入第三维，让我们获得从桌子一边盯着对手的视觉感受。

这就是本章的开发计划：

- 首先，我们会学习关于 OpenGL 的透视除法（perspective division）的内容，以及如何在二维屏幕上使用 w 分量创造三维的幻象。
- 一旦理解了 w 分量，我们就会学习如何设置透视投影，让我们可以见到三维形式的桌子。

作为开始，我们把上一章的项目复制到一个新的项目中，并命名为“AirHockey3D”。

6.1 三维的艺术

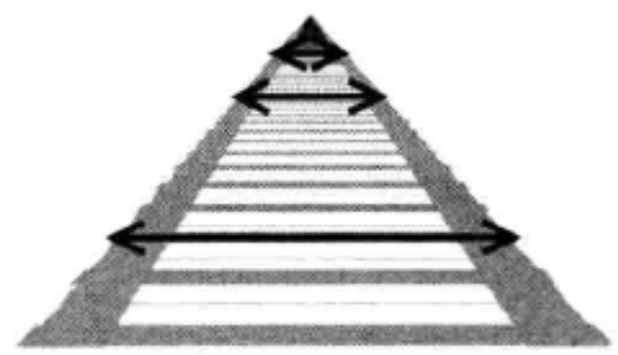
几个世纪以来，艺术家一直在欺骗人们把一个平面二维绘图视为一个完整的三维场景。他们使用的技巧之一被称为线性投影（linear projection），它的工作原理是，在一个想象中的消失点处把并行线段聚合在一起，从而创建出立体化的幻象。

我们可以看到这种效果的一个经典的例子，当站在一对笔直的铁轨上时，向铁轨的远处看，它们看起来越来越近了，直到它们看起来消失在地平线上的一个单个点上。

那些铁路枕木离我们越远，它们看上去也越小。如果我们测量每个铁路枕木的外观尺寸，被测量出来的尺寸将按照我们的眼睛与其之间的距离成比例递减。

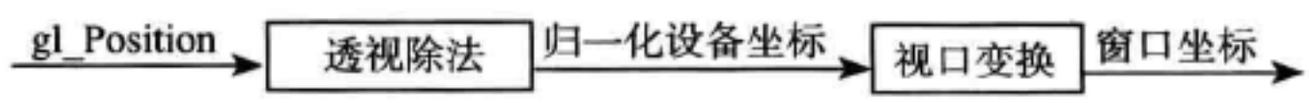
在右图中，观察一下每个铁路枕木测量的宽度是怎样随着距离递减的。

这个技巧就是创造真实的三维投影所需要的；让我们继续学习 OpenGL 实际是怎样做的。



6.2　从着色器到屏幕的坐标变换

我们现在熟悉了归一化设备坐标，并且知道要让一个顶点显示在屏幕上，它的 x、y 和 z 分量都要在范围 [−1, 1] 内取值。让我们看看下面的流程图，复习一下怎样把一个在顶点着色器上的原始 gl_Position 坐标变换为最终的屏幕坐标。



这里涉及两个变换步骤和三个不同的坐标空间。

6.2.1　剪裁空间

当顶点着色器把一个值写到 gl_Position 的时候，OpenGL 期望这个位置是在剪裁空间（clip space）中的。剪裁空间背后的逻辑非常简单：对于任何给定的位置，它的 x、y 以及 z 分量都需要在那个位置的 −w 和 w 之间。比如，如果一个位置的 w 是 1，那么其 x、y 以及 z 分量都需要在 −1 和 1 之间。任何这个范围外的事物在屏幕上都是不可见的。

一旦我们了解了透视除法，它为什么依赖于那个位置的 w 分量的原因就显而易见了。

6.2.2　透视除法

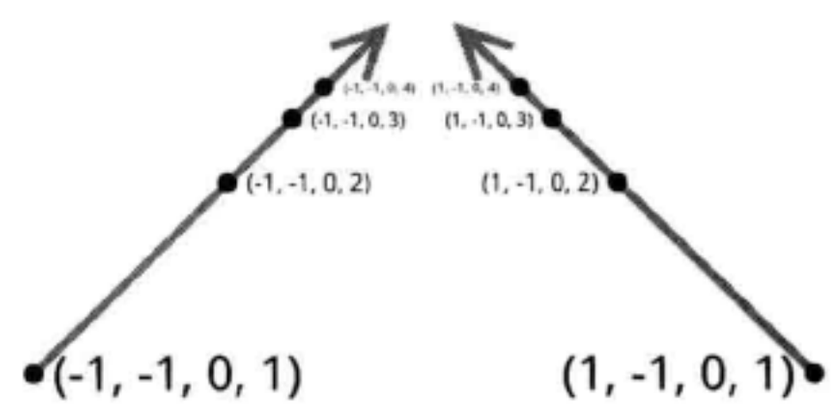
在一个顶点位置成为一个归一化设备坐标之前，OpenGL 实际上执行了一个额外的步骤，它被称为透视除法（perspective division）。透视除法之后，那个位置就在归一化设备坐标中了，不管渲染区域的大小和形状，对于其中的每个可视坐标，其 x、y 和 z 分量的取值都位于 [−1, 1] 的范围内。

为了在屏幕上创建三维的幻象，OpenGL 会把每个 gl_Position 的 x、y 和 z 分量都除以它的 w 分量。当 w 分量用来表示距离的时候，就使得较远处的物体被移动到距离渲染区域中心更近的地方，这个中心的作用就像一个消失点。OpenGL 正是用这种方法欺骗我们，使

我们看见一个三维场景的，它使用了已经被艺术家们使用了几个世纪的技巧。

举个例子，假设一个物体，它有两个顶点，每个顶点都在三维空间中的同一个位置，它们有同样的 x、y 和 z 分量，但是 w 分量却不同。假设这两个坐标是 (1, 1, 1, 1) 和 (1, 1, 1, 2)。在 OpenGL 把这些作为归一化设备坐标使用之前，它会做透视除法，其前三个分量都除以 w；这两个坐标被整除后变为 (1/1, 1/1, 1/1) 和 (1/2, 1/2, 1/2)。这个除法之后，归一化设备坐标就是 (1, 1, 1) 和 (0.5, 0.5, 0.5)。有较大 w 值的坐标被移动到距离 (0, 0, 0) 更近的位置了，(0, 0, 0) 就是归一化设备坐标里渲染区域的中心。

在下面的图像中，我们可以在一个实际的例子中看到这个影响，有同样 x、y 和 z 值的坐标，随着其 w 值的增加，它距离中心也越来越近。



在 OpenGL 里，三维效果是线性的，并且是沿着直线完成的。在现实生活中，事物更加复杂（想象一下鱼眼镜头），这种线性投影只是一种合理的近似。

同质化坐标

因为透视除法，剪裁空间中的坐标经常被称为同质化坐标（homogenous coordinate）[⊖]，它是 1827 年由 August Ferdinand Möbius 引入的。它们被称为同质化的原因是因为剪裁空间中的几个坐标可以映射到同一个点。例如，考虑以下几个点：

(1, 1, 1, 1), (2, 2, 2, 2), (3, 3, 3, 3), (4, 4, 4, 4), (5, 5, 5, 5)

透视除法之后，所有这些点都映射到归一化设备坐标中的 (1, 1, 1)。

除以 w 的优势

你可能想知道我们为什么不简单地除以 z 呢？毕竟，如果我们把 z 解释为距离，并且有 2 个坐标，分别是 (1, 1, 1) 和 (1, 1, 2)，我们就可以通过除以 z 得到两个归一化坐标 (1, 1) 和 (0.5, 0.5)。

尽管这行得通，但增加 w 作为第四个分量有另外一个优势。我们可以把投影的影响与实际的 z 坐标解耦，以便我们可以在正交投影和透视投影之间切换。保留 z 分量作为深度缓冲区（depth buffer）还有一个好处，我们会在 12.5.1 节中讲述。

6.2.3 视口变换

在我们看到最后的结果之前，OpenGL 需要把归一化设备坐标的 x 和 y 分量映射到屏幕上的一个区域内，这个区域是操作系统预留出来用于显示的，被称为视口（viewport）；这些被映射的坐标被称为窗口坐标（window coordinate）。除了要告诉 OpenGL 怎样映射之外，我们不需要太关心这些坐标。在当前的代码中，我们已经在 onSurfaceChanged() 中用 glViewport() 的调用告诉 OpenGL 了。

⊖ http://en.wikipedia.org/wiki/Homogeneous_coordinates

当 OpenGL 做映射的时候，它会把从 (−1, −1, −1) 到 (1, 1, 1) 的范围映射到那个为显示而预留的窗口上。这个范围之外的归一化设备坐标会被裁剪掉。正如我们在 5.1.1 节中学到的，不管视口的宽度或高度，这个范围总是一样的。

6.3　添加 w 分量创建三维图

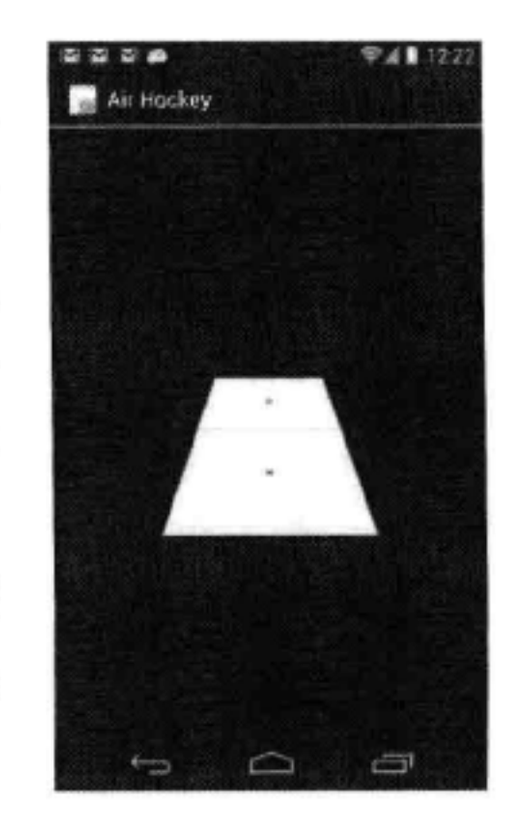
如果我们实际看一下 w 分量的使用，更容易理解它的影响，因此，让我们把它加入桌子的顶点数据中，看看会发生什么。因为我们现在要指定一个位置的 x、y、z 和 w 分量，作为开始，让我们把 POSITION_COMPONENT_COUNT 更新如下：

```
AirHockey3D/src/com/airhockey/android/AirHockeyRenderer.java
private static final int POSITION_COMPONENT_COUNT = 4;
```

对于所使用的一切，我们都必须给 OpenGL 传递正确的分量计数；否则就可能出现花屏、什么都不显示，或者应用程序崩溃。

下一步是更新所有的顶点数据：

```
AirHockey3D/src/com/airhockey/android/AirHockeyRenderer.java
float[] tableVerticesWithTriangles = {
    // Order of coordinates: X, Y, Z, W, R, G, B

    // Triangle Fan
       0f,    0f, 0f, 1.5f,   1f,   1f,   1f,
    -0.5f, -0.8f, 0f,   1f, 0.7f, 0.7f, 0.7f,
     0.5f, -0.8f, 0f,   1f, 0.7f, 0.7f, 0.7f,
     0.5f,  0.8f, 0f,   2f, 0.7f, 0.7f, 0.7f,
    -0.5f,  0.8f, 0f,   2f, 0.7f, 0.7f, 0.7f,
    -0.5f, -0.8f, 0f,   1f, 0.7f, 0.7f, 0.7f,

    // Line 1
    -0.5f, 0f, 0f, 1.5f, 1f, 0f, 0f,
     0.5f, 0f, 0f, 1.5f, 1f, 0f, 0f,

    // Mallets
    0f, -0.4f, 0f, 1.25f, 0f, 0f, 1f,
    0f,  0.4f, 0f, 1.75f, 1f, 0f, 0f
};
```

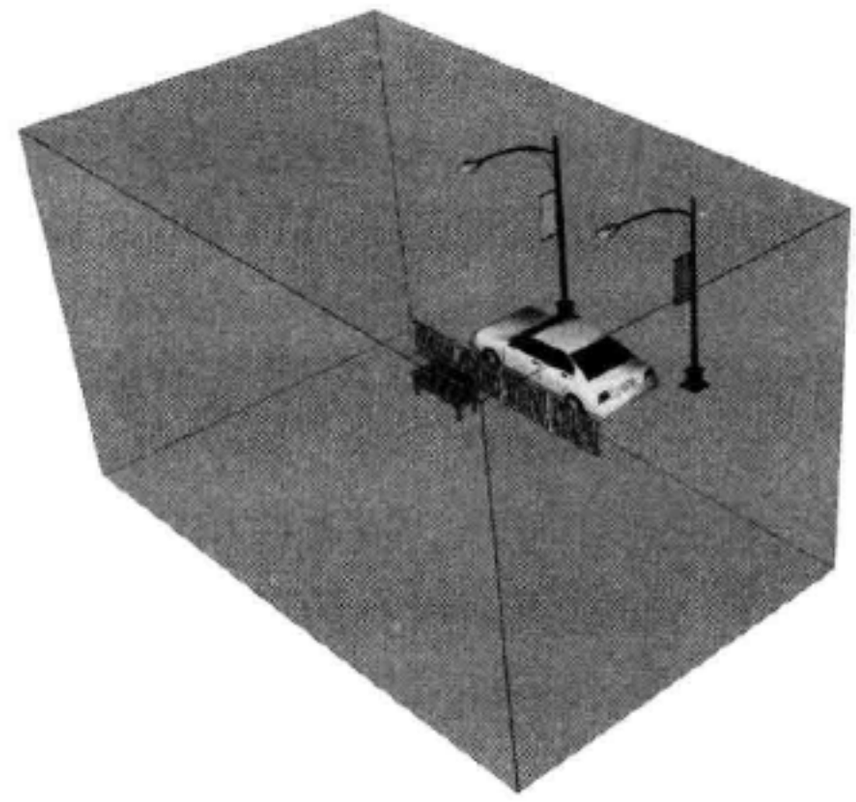


我们给顶点数据加入了一个 z 和一个 w 分量。我们已经更新了所有的顶点，其中接近屏幕底部的顶点的 w 设为 1，而那些接近屏幕顶部的顶点的 w 设为 2；我们也把那条直线和木槌的 w 更新为从 1 到 2 之间的小数。应该产生这样的效果：桌子的顶部看上去比底部小，好像我们正向远处观看一样。我们把所有顶点的 z 分量都设为 0，因为我们不需要 z 上有任何值就能获得立体效果。

OpenGL 会自动使用我们指定的 w 值做透视除法，我们现在的正交投影只是把这些 w 值复制过去；继续运行这个项目，看看它看起来是什么样的。它看上去应该与右图类似。

它们开始看起来更像三维了！我们只是放进了一个自己指定的 w 就做到了这些。然而，如果我们想让这些物体更动态，比如改变桌子的角度或者缩小、放大，怎么办呢？不要使用硬编码指定 w 的值，我们要用矩阵来生成这些值。把我们刚刚做的改动去掉，恢复原来的代码；在下一节，我们要学习如何用透视投影矩阵自动生成 w 的值。

6.4 使用透视投影



在进入透视投影背后的矩阵数学之前，让我们在视觉层次上讨论一下。在前一章，我们用一个正交投影矩阵来弥补屏幕的宽高比，它通过调整显示区域的宽度和高度使之变换为归一化设备坐标。

在右图中，我们把一个正交投影可视化为一个包围整个场景的立方体，它代表 OpenGL 最终在视口上渲染的内容，也是我们能看到的内容。

图 6-1 “被投影的场景”从一个不同的视角显示了同样的场景。

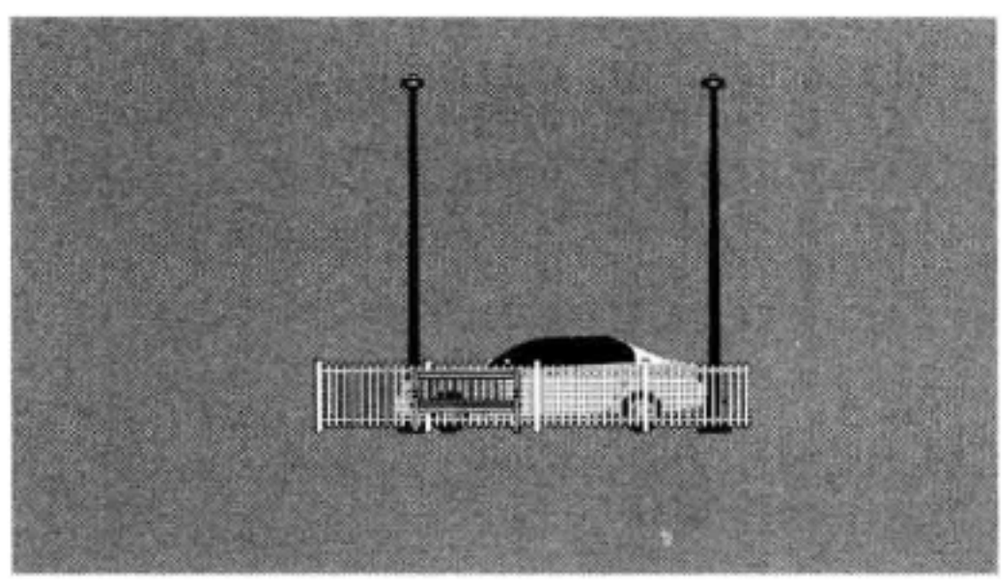


图 6-1 被投影的场景

视椎体

一旦我们使用了投影矩阵，场景中的并行线就会在屏幕上的一个消失点处汇聚在一起，当物体离得越来越远，它们会变得更小。不使用立方体，我们可以看到的那个空间区域将如图 6-2 所示。

这个形状叫做视椎体（frustum）[⊖]，这个观看空间是由一个透视投影矩阵和投影除法创建的。简单来说，视椎体只是一个立方体，其远端比近端大，从而使其变成一个被截断的金字塔。两端的大小差别越大，观察的范围越宽，我们能看到的也越多。

⊖ http://en.wikipedia.org/wiki/Frustum

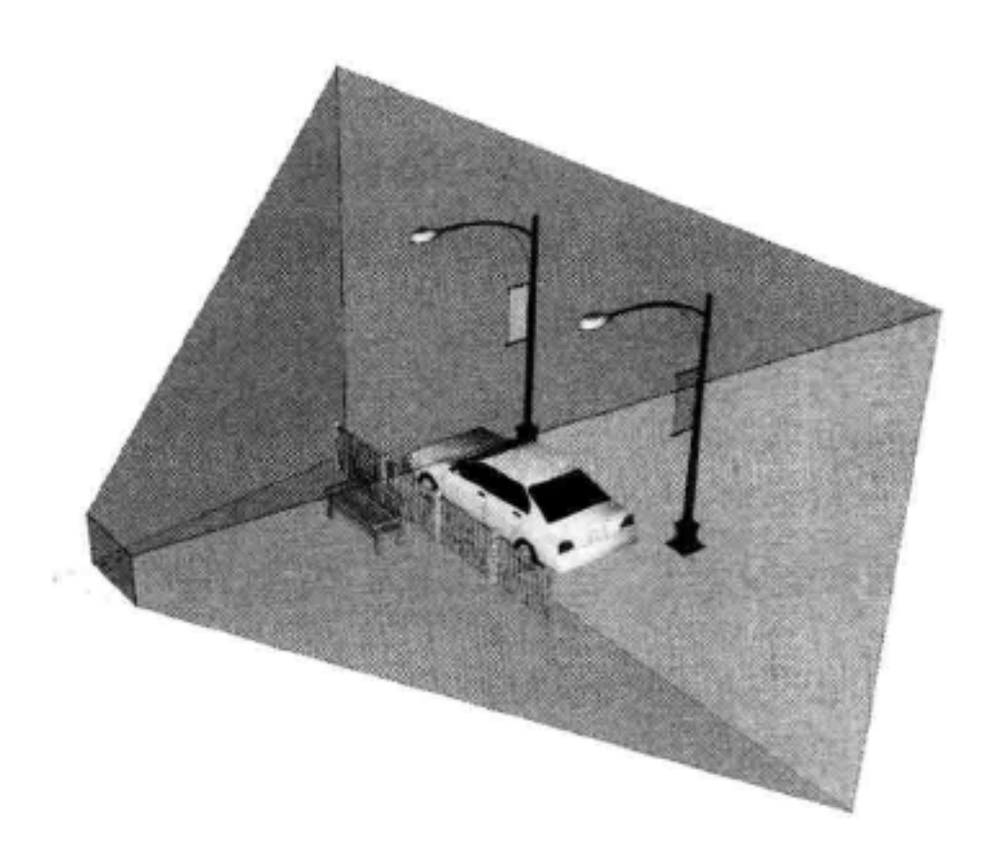
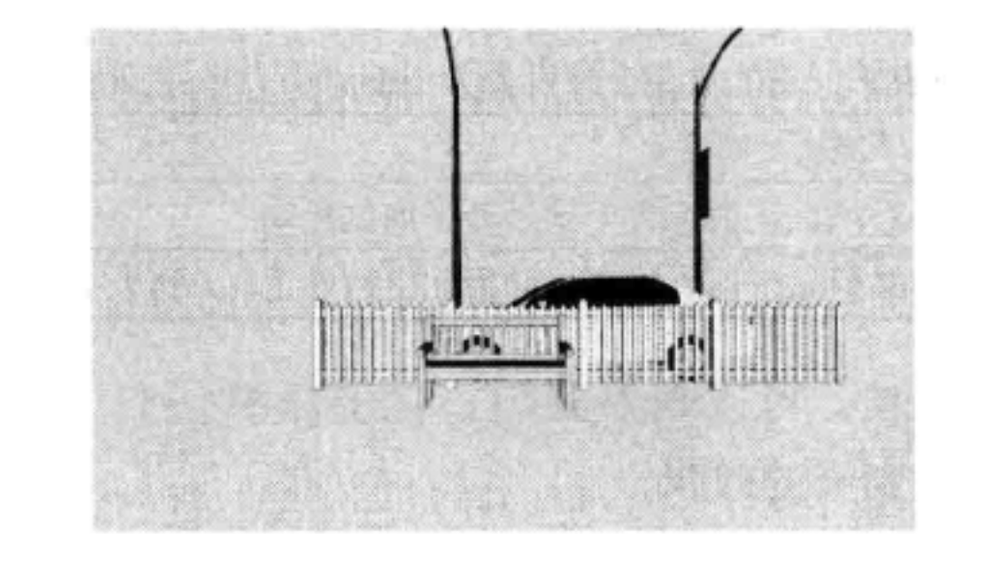


图 6-2　通过视椎体投影

一个视椎体有一个焦点（focal point）。这个焦点可以这样得到，顺着从视椎体较大端向较小端扩展出来的那些直线，一直向前通过较小端直到它们汇聚到一起。当你用透视投影观察一个场景的时候，那个场景看上去就像你的头被放在了焦点处。焦点和视椎体小端的距离被称为焦距（focal length），它影响视椎体小端和大端的比例，以及其对应的视野。

在下面的图像中，我们可以在视椎体内部看一下这个场景，这是从焦点处所观察的：



焦点还有另外一个有趣的性质，从焦点看上去视椎体两端在屏幕上占据了同样大小的空间。视椎体的远端虽然较大，但是它也比较远，它仍然占据同样大小的空间。这与我们看到的日食是一样的原理：月亮虽然比太阳小很多，但因为它也近了很多，使它看上去很大，足够掩盖住太阳的光辉！这些都依赖于这个有利的位置。

6.5　定义透视投影

为了重新创造三维的魔法，透视投影矩阵需要和透视除法一起发挥作用。投影矩阵不能自己做透视除法，而透视除法需要某些东西才能起作用。

如果一个物体向屏幕中心移动，当它离我们越来越远时，它的大小也越来越小，因此，投影矩阵最重要的任务就是为 w 产生正确的值，这样，当 OpenGL 做透视除法的时候，远

处的物体看起来就会比近处的物体小。能实现这些的方法之一是利用 z 分量，把它作为物体与焦点的距离并且把这个距离映射到 w。这个距离越大，w 值越大，所得物体越小。

这里不会讨论更多关于数学的内容，但是如果你想知道更多的细节，可以直接跳到 A1.2 节，等你准备好继续下一节时回到这里。

对宽高比和视野进行调整

让我们看一个更加通用的投影矩阵，它允许我们调整视野以及屏幕的宽高比。

$$\begin{bmatrix} \frac{a}{aspect} & 0 & 0 & 0 \\ 0 & a & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

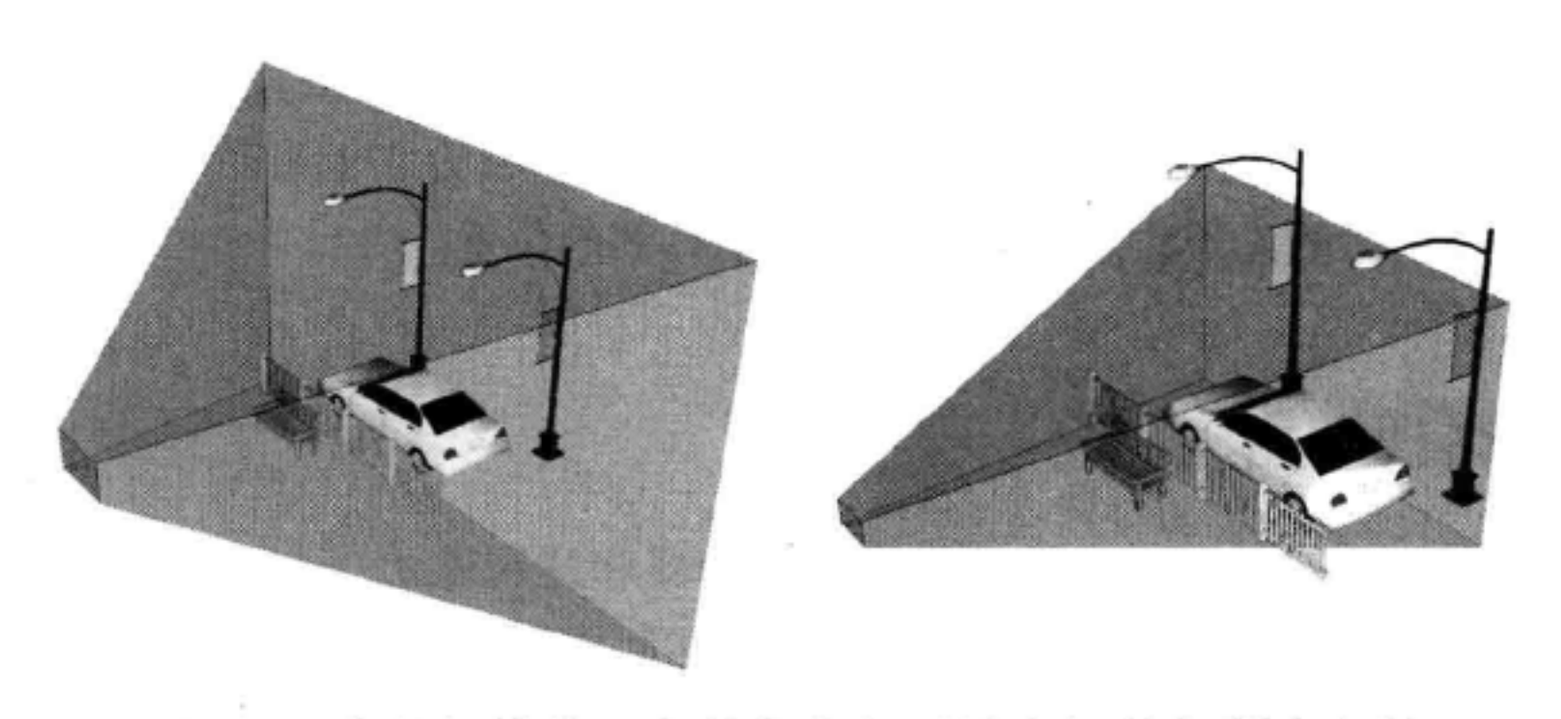
表 6-1 是这个矩阵中定义的变量的一个快速解释。

表 6-1 投影矩阵变量

a	如果我们想象一个相机拍摄的场景，这个变量就代表那个相机的焦距。焦距是由 1/tan(视野 /2)（tan 表示正切函数）计算得到的。这个视野必须小于 180 度。 比如，一个 90 度的视野，它的焦距会被设置为 1/tan(90° /2)，也就是 1/1 或者 1
aspect	屏幕的宽高比，它等于宽度 / 高度
f	到远处平面的距离，必须是正值且大于到近处平面的距离
n	到近处平面的距离，必须是正值。比如，如果此值被设为 1，那近处平面就位于一个 z 值为 −1 处

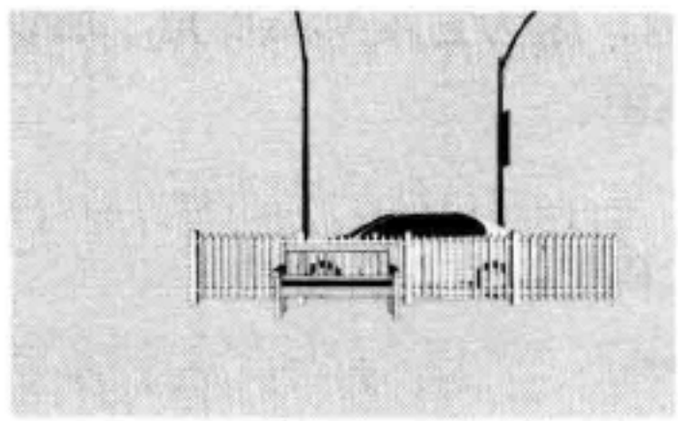
随着视野变小，焦距变长，可以映射到归一化坐标中 [−1, 1] 范围内的 x 和 y 值的范围就越小。这会产生使视椎体变窄的效果。

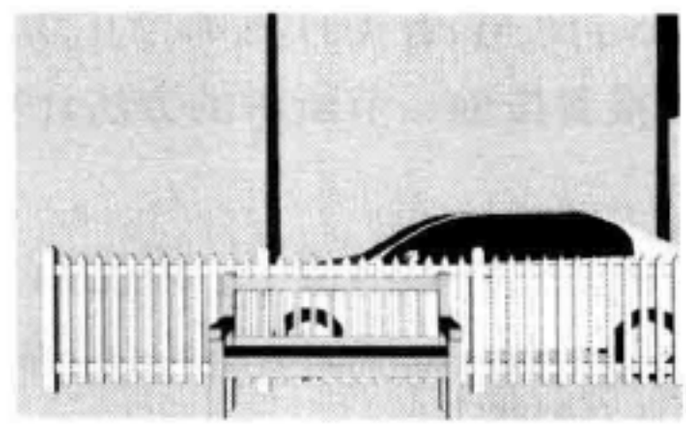
在下图中，左边的视椎体有 90 度的视野，而右边的视椎体只有 45 度的视野。



你可以看到，对于 45 度的视椎体，它的焦点与近端之间的焦距有点长。

看看下面的图像，它们是在相同的视椎体内分别从它们的焦点处所看到的图像。





较窄的视野通常很少有扭曲的问题。反过来说，随着视野变宽，最终的图像的边缘看起来会扭曲得更加严重。在现实生活中，较宽的视野会让一切看上去都是弯曲的，就像在照相机上使用鱼眼镜头观察到的效果。因为 OpenGL 使用的是沿着直线的线性投影，最终的图像反而会被拉伸。

6.6　在代码中创建投影矩阵

我们现在准备好在代码中添加透视投影了。Android 的 Matrix 类为它准备了两个方法——frustumM() 和 perspectiveM()。不幸的是，frustumM() 有个缺陷，它会影响某些类型的投影[⊖]，而 perspectiveM() 只是从 Android 的 ICS（Ice Cream Sandwich，冰淇淋三明治）版本开始才被引入，在早期的 Android 版本里并没有这个方法。我们可以简单地只支持 ICS 及其以上版本，但是这样会丢掉很大一部分市场，一些用户仍然在运行早期的 Android 版本。

作为替代，我们可以创建我们自己的方法来实现前一节定义的矩阵。打开我们在本章开始处创建的项目，给包 com.airhockey.android.util 添加一个新类，命名为 MatrixHelper。我们会实现这样一个方法，它与 Android 的 Matrix 类中的 perspectiveM() 非常相似。

创建自己的 perspectiveM

在 MatrixHelper 的开始处加入如下方法签名（method signature）：

AirHockey3D/src/com/airhockey/android/util/MatrixHelper.java

```
public static void perspectiveM(float[] m, float yFovInDegrees, float aspect,
    float n, float f) {
```

计算焦距

我们要做的第一件事情就是计算焦距，这将基于在 y 轴上的视野。就在方法签名之后加入如下代码：

AirHockey3D/src/com/airhockey/android/util/MatrixHelper.java

```
final float angleInRadians = (float) (yFovInDegrees * Math.PI / 180.0);

final float a = (float) (1.0 / Math.tan(angleInRadians / 2.0));
```

⊖ http://code.google.com/p/android/issues/detail?id=35646

我们使用 Java 的 Math 类计算那个正切函数，因为它需要弧度角，所以我们把视野从度转换为弧度。接着按前一节所讲的方法计算焦距。

输出矩阵

我们现在可以写出矩阵的值了。增加如下代码把这个方法完成：

AirHockey3D/src/com/airhockey/android/util/MatrixHelper.java

```
    m[0] = a / aspect;
    m[1] = 0f;
    m[2] = 0f;
    m[3] = 0f;

    m[4] = 0f;
    m[5] = a;
    m[6] = 0f;
    m[7] = 0f;

    m[8] = 0f;
    m[9] = 0f;
    m[10] = -((f + n) / (f - n));
    m[11] = -1f;
    m[12] = 0f;
    m[13] = 0f;
    m[14] = -((2f * f * n) / (f - n));
    m[15] = 0f;
}
```

这就把矩阵数据存到了参数 m 定义的浮点数组中，这个数组需要至少 16 个元素。OpenGL 把矩阵数据按照以列为主的顺序存储，这就意味着我们一次写一列数据，而不是一次写一行。前四个值是第一列，下一组四个数是第二列，以此类推。

我们现在已经完成了自己的 perspectiveM()，并准备好在代码中使用它了。这个方法与 Android 源代码自带的方法非常相似[⊖]，我们只做了一点小改动，以使它更加可读。

6.7 开始使用投影矩阵

我们现在将转而使用那个透视投影矩阵了。打开 AirHockeyRenderer，并从 onSurfaceChanged() 去掉所有的代码，只保留 glViewPort() 调用。加入如下代码：

AirHockey3D/src/com/airhockey/android/AirHockeyRenderer.java

```
MatrixHelper.perspectiveM(projectionMatrix, 45, (float) width
    / (float) height, 1f, 10f);
```

这会用 45 度的视野创建一个透视投影。这个视椎体从 z 值为 −1 的位置开始，在 z 值为 −10 的位置结束。

加入 MatrixHelper 的导入后，继续运行这个程序。你可能注意到空气曲棍球桌子不见

⊖ http://grepcode.com/file_/repository.grepcode.com/java/ext/com.google.android/android/4.0.4_r1.2/android/opengl/Matrix.java/?v=source

了！因为我们没有给桌子指定 z 的位置，默认情况下它处于 z 为 0 的位置。因为这个视椎体是从 z 值为 −1 的位置开始的，除非把它移到那个距离内，否则我们无法看到桌子。

不要硬编码 z 的值，在使用投影矩阵进行投影之前，让我们使用一个平移矩阵把桌子移出来。依照惯例，我们把这个矩阵称为模型矩阵（model matrix）。

利用模型矩阵移动物体

在类的顶部加入如下矩阵定义：

AirHockey3D/src/com/airhockey/android/AirHockeyRenderer.java

```
private final float[] modelMatrix = new float[16];
```

我们要使用这个矩阵把空气曲棍球桌子移到那个距离内。在 onSurfaceChanged() 结尾处，加入如下代码：

AirHockey3D/src/com/airhockey/android/AirHockeyRenderer.java

```
setIdentityM(modelMatrix, 0);
translateM(modelMatrix, 0, 0f, 0f, -2f);
```

这就把模型矩阵设为单位矩阵，再沿着 z 轴平移 −2。当我们把空气曲棍球桌子的坐标与这个矩阵相乘的时候，那些坐标最终会沿着 z 轴负方向移动 2 个单位。

相乘一次还是相乘两次

我们现在要做一个选择：我们仍然需要把这个矩阵应用于每个顶点，因此第一个选项是给顶点着色器新增一个额外的矩阵。我们把每个顶点都与这个模型矩阵相乘，让它们沿着 z 轴负方向移动 2 个单位，接下来把每个顶点与投影矩阵相乘。这样，OpenGL 就可以做透视除法，并把这些顶点变换到归一化设备坐标了。

如果不想这么麻烦，还有一个更好的方式：我们可以把模型矩阵与投影矩阵相乘，得到一个矩阵，然后把这个矩阵传递给顶点着色器。通过这种方式我们就可以在着色器中仅保留一个矩阵。

矩阵乘法

矩阵与矩阵乘法的原理与矩阵与向量的乘法很相似。

举个例子，假定有两个通用的矩阵，如下所示：

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

为了得到其结果的第一个元素，我们把第一个矩阵的第一行与第二个矩阵的第一列相乘，并加在一起，得到的结果就是：

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31}+a_{14}b_{41} & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix}$$

然后计算其结果的第二个元素，我们把第一个矩阵的第二行与第二个矩阵的第一列相乘，并把它们相加，得到的结果就是：

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} ? & ? & ? & ? \\ a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31}+a_{24}b_{41} & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix}$$

以此类推，继续得出其结果矩阵的每个后续元素。

乘法顺序

既然我们知道如何把两个矩阵相乘了，我们需要小心，要确保它们按正确的顺序相乘。相乘时，我们既可以把投影矩阵放在左边、模型矩阵放在右边；也可以把模型矩阵放在左边、投影矩阵放在右边。

与常规乘法不同，这个顺序很重要！如果我们把顺序弄错了，物体可能看起来很怪异，甚至我们可能什么都看不到！下面就是两个矩阵按其中一个特定顺序相乘的例子：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 20 & 13 \end{bmatrix}$$

下面是同样两个矩阵按相反顺序相乘的结果：

$$\begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 13 & 20 \\ 5 & 8 \end{bmatrix}$$

顺序不同，结果也不同。

选择适当的顺序

为了弄清楚我们应该使用那种顺序，让我们看一下只使用投影矩阵的数学运算：

$$\text{vertex}_{\text{clip}} = \text{ProjectionMatrix} * \text{vertex}_{\text{eye}}$$

$\text{vertex}_{\text{eye}}$ 代表场景中的顶点在与投影矩阵相乘之前的位置。我们一旦加入模型矩阵来移动那个桌子，这个数学运算看起来就像这样：

$$\text{vertex}_{\text{eye}} = \text{ModelMatrix} * \text{vertex}_{\text{model}}$$

$$\text{vertex}_{\text{clip}} = \text{ProjectionMatrix} * \text{vertex}_{\text{eye}}$$

$\text{vertex}_{\text{model}}$ 代表顶点在被模型矩阵放进场景中之前的位置。把这两个表达式合在一起，最后得到如下公式：

$$vertex_{clip} = ProjectionMatrix * ModelMatrix * vertex_{model}$$

为了用一个矩阵替换这两个矩阵，我们不得不把投影矩阵乘以模型矩阵，就是把投影矩阵放在左边，把模型矩阵放在右边。

更新代码使用一个矩阵

让我们把这个新的矩阵代码封装一下，把下面的代码加到 onSurfaceChanged() 中的 translateM() 调用后面：

```
AirHockey3D/src/com/airhockey/android/AirHockeyRenderer.java
final float[] temp = new float[16];
multiplyMM(temp, 0, projectionMatrix, 0, modelMatrix, 0);
System.arraycopy(temp, 0, projectionMatrix, 0, temp.length);
```

不论什么时候把两个矩阵相乘，都需要一个临时变量来存储其结果。如果尝试直接写入这个结果，这个结果将是未定义的！

我们首先创建一个临时的浮点数组用来存储其临时结果；然后调用 multiplyMM() 把投影矩阵和模型矩阵相乘，其结果存进这个临时数组。下一步，我们调用 System.arraycopy() 把结果存回 projectionMatrix，它现在包含模型矩阵与投影矩阵的组合效应。

如果我们现在运行这个应用程序，结果应该如图 6-3 所示。把那个空气曲棍球桌子推动到那个距离内，这个距离足够把它放在我们的视椎体内，但是桌子还是直立的。一番快速回顾以后，我们会学习如何旋转这个桌子，以便能从某个角度观察它，而不是正上方。

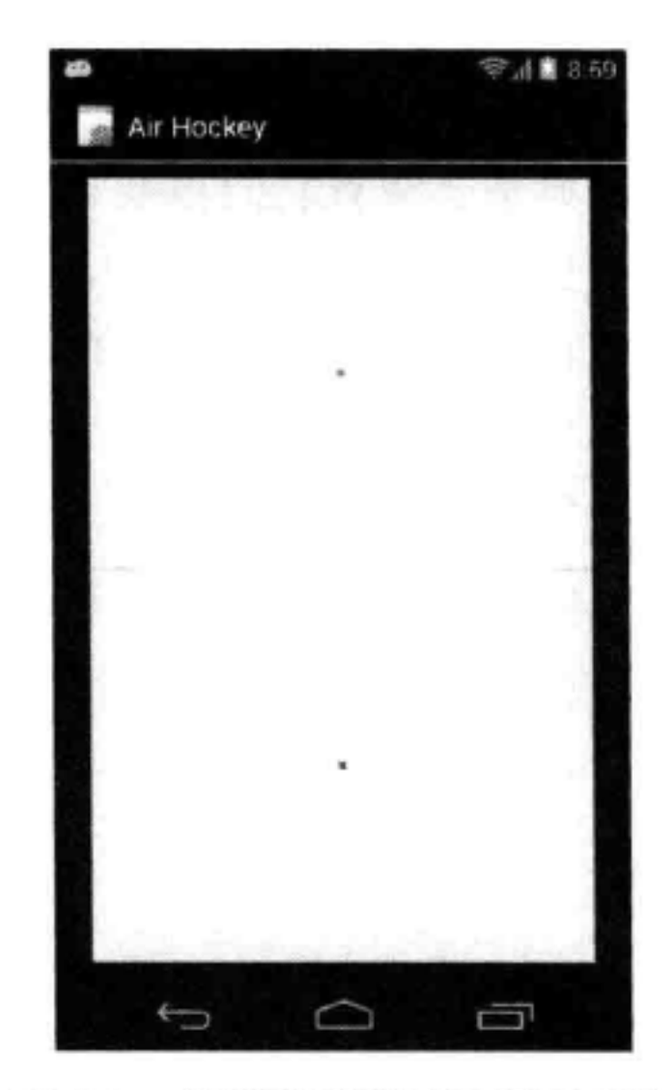


图 6-3　使用投影矩阵和模型矩阵

快速回顾

我们快速回顾一下上面几小节讲到的内容：

- 我们学习了在空气曲棍球桌子被传递给投影矩阵之前如何使用一个额外的矩阵把它移动到屏幕内；
- 我们又学习了如何让两个矩阵相乘；
- 接着，我们学习了如何把投影矩阵和模型矩阵合并，这样我们就不必修改顶点着色器，它不用接受另外一个矩阵并且每次都要与两个矩阵相乘。

6.8　增加旋转

既然我们已经有了一个配置好的投影矩阵和一个可以移动桌子的模型矩阵，那么我们需要做的就是旋转这张桌子，以便可以从某个角度观察它。如果使用旋转矩阵，我们只需

一行代码就可以做到。我们还从来没有用过旋转，现在花一些时间了解一下这些旋转是怎么工作的。

6.8.1 旋转的方向

需要弄清楚的第一件事情是我们需要围绕哪个轴旋转以及旋转多少度。让我们再看一下图 5-6。要搞清楚一个物体怎样围绕一个给定的轴旋转，我们将使用右手坐标规则：伸出你的右手，握拳，让大拇指指向正轴的方向。倘若是一个正角度的旋转，蜷曲的手指会告诉你一个物体是怎样围绕那个轴旋转的。观察图 6-4，当你把大拇指指向 x 轴的正方向时，看看旋转的方向是怎样跟随手指的绕轴线卷曲的。

分别用 x 轴、y 轴和 z 轴试一下这个旋转。如果绕着 y 轴旋转，桌子会绕着它的顶端和底端水平旋转。如果绕着 z 轴旋转，桌子会在一个圆圈内旋转。我们想要做的是让桌子绕着 x 轴向后旋转，因为这会让桌子看起来更有层次。

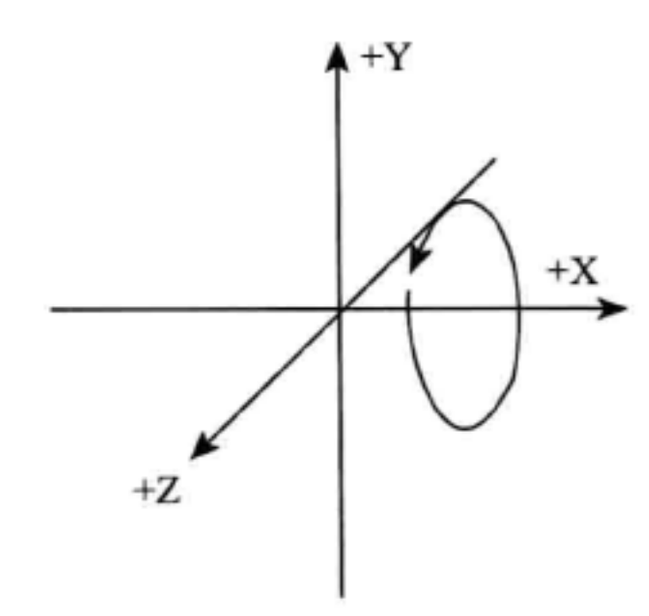


图 6-4 绕 x 轴旋转

6.8.2 旋转矩阵

我们将使用一个旋转矩阵去做实际的旋转。旋转矩阵使用正弦和余弦三角函数把旋转角转换成缩放因子。下面就是绕 x 轴旋转所用矩阵的定义：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & -\sin(a) & 0 \\ 0 & \sin(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

然后是绕 y 轴旋转所用的矩阵：

$$\begin{bmatrix} \cos(a) & 0 & \sin(a) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(a) & 0 & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

最后，还有一个绕 z 轴旋转所用的矩阵：

$$\begin{bmatrix} \cos(a) & -\sin(a) & 0 & 0 \\ \sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

把所有这些矩阵合并为一个通用的旋转矩阵，使其可以基于任意一个角度和向量旋转[⊖]，这也是可能的。

作为一个测试，让我们试试绕 x 轴的旋转。我们从一个点开始，它在原点上面一个单位，也就是 y 值是 1。把它绕 x 轴旋转 90 度。首先，让我们准备这个旋转矩阵：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(90) & -\sin(90) & 0 \\ 0 & \sin(90) & \cos(90) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

让我们把这个矩阵与这个点的位置向量相乘，看看得到什么：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

这个点从 (0, 1, 0) 被移动到了 (0, 0, 1)。如果我们回过头看一下图 6-4，并对 x 轴使用右手规则，我们可以看到正向旋转是如何把一个点沿着一个绕 x 轴的圈移动的。

6.8.3　在代码中加入旋转

我们现在准备好把这个旋转加入代码了。回到 onSurfaceChanged()，调整那个平移矩阵，并加入一个旋转矩阵，如下：

AirHockey3D/src/com/airhockey/android/AirHockeyRenderer.java

```
translateM(modelMatrix, 0, 0f, 0f, -2.5f);
rotateM(modelMatrix, 0, -60f, 1f, 0f, 0f);
```

我们把这张桌子放得更远点儿，因为我们一旦把它旋转了，它的底部会距离我们更近。我们接着把它绕着 x 轴旋转 −60 度，这会让桌子处于一个很好的角度，就像我们站在它前面一样。

这张桌子现在看起来应该如图 6-5 所示。

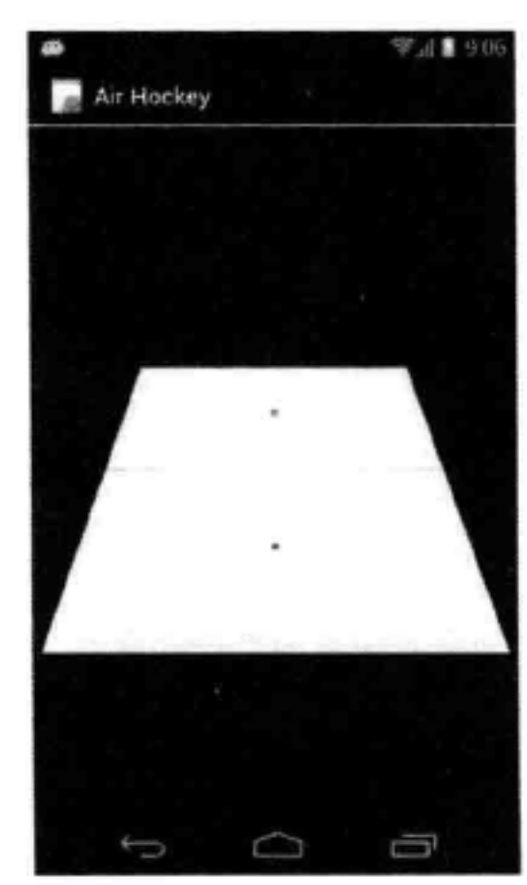


图 6-5　绕 x 轴旋转 −60 度的空气曲棍球桌子

6.9　小结

本章的信息量很大。当我们学习投影矩阵以及它们怎样与 OpenGL 的透视除法一起工作的时候，引入了更多有关矩阵数

⊖ http://en.wikipedia.org/wiki/Rotation_matrix#Rotation_matrix_from_axis_and_angle

学的内容。接着我们学习了如何使用第二个矩阵移动和旋转那张桌子。

好消息是我们不需要深入理解这些投影和旋转背后的数学原理及理论，因为我们只是要使用它们。只要基本理解了什么是视椎体以及矩阵如何帮我们移动东西，你就会发现今后用 OpenGL 开发更加容易。

6.10 练习

试试调整一下视野，观察其应用在这张空气曲棍球桌子上的效果。你也可以尝试用不同的方法移动这张桌子。

一旦你完成了这些练习，就可以开始把这张桌子变得更美观。下一章我们将开始使用纹理。

第 7 章 Chapter 7

用纹理增加细节

我们已经设法只用简单的图形和颜色完成了很多工作。但是还缺了点什么：如果我们想在这些图形上绘画并加入精致的细节呢？像艺术家一样，我们可以从基本的图形和颜色开始，再使用纹理（texture）在其表面上加入额外的细节。简单来说，纹理就是一个图像或照片，它们可以被加载进 OpenGL 中。

我们可以用纹理加入大量令人难以置信的细节。想想你最近可能玩过的一个精美的三维游戏。与任何其他三维程序一样，那个游戏的本质也只是使用了点、直线和三角形。然而，通过纹理添加的细节和一个熟练的艺术家的加工，这些三角形可以用纹理构建一个精美的三维场景。

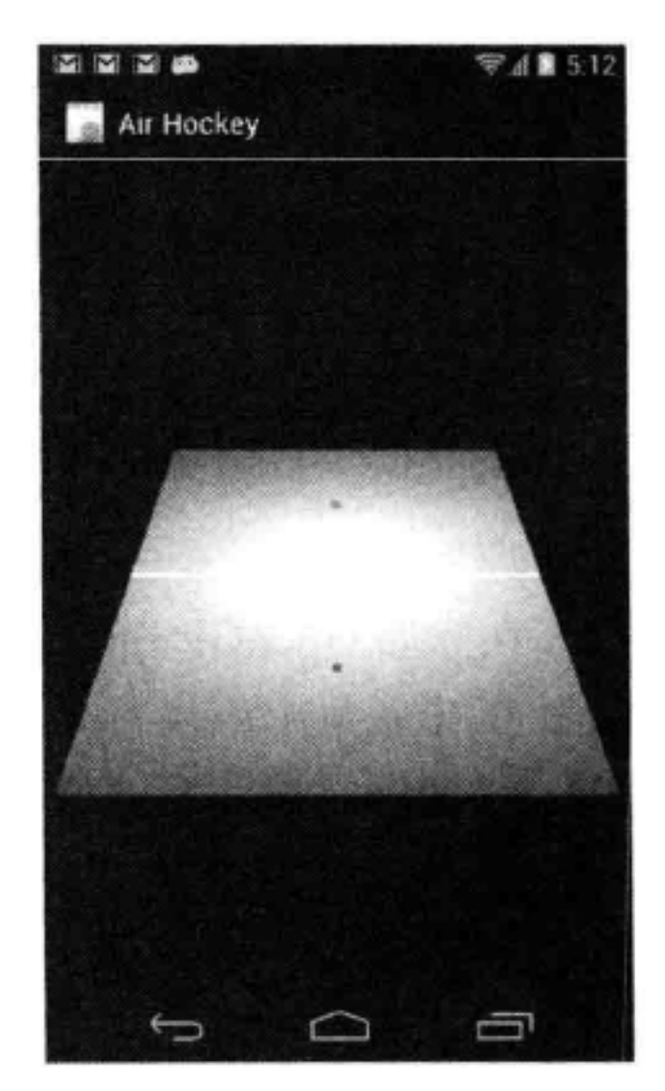


图 7-1 使用一个过滤后的纹理的空气曲棍球桌子

一旦我们开始使用纹理，就要开始使用多个着色器的程序了。为了使其更容易管理，我们要学习如何让代码适应这些变化，以便使用多个着色器程序和顶点数据源，并在它们之间来回切换。

下面是本章的开发计划：

- 我们将以纹理的介绍作为开始，然后撰写把纹理加载进 OpenGL 中的代码；
- 我们要学习如何显示那个纹理，调整代码，使其支持多个着色器程序；
- 我们也会讲到不同的纹理过滤模式以及它们的应用场景。

当这些都完成了，空气曲棍球桌子应该与图 7-1 类似。我们把上一章的项目复制成一个新的项目，命名为

“AirHockeyTextured”，以此作为开始。

7.1 理解纹理

OpenGL 中的纹理可以用来表示图像、照片、甚至由一个数学算法生成的分形数据。每个二维的纹理都由许多小的纹理元素（texel）组成，它们是小块的数据，类似于我们前面讨论过的片段和像素。要使用纹理，最常用的方式是直接从一个图像文件加载数据。

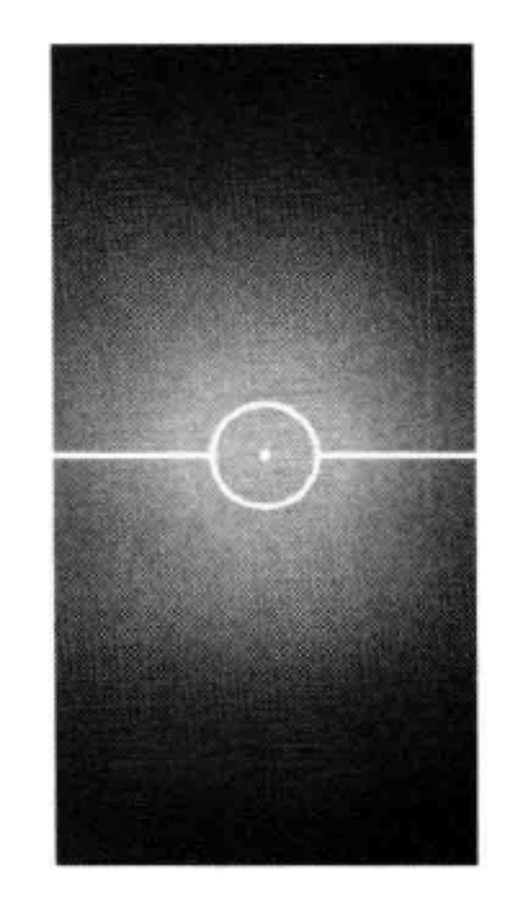


图 7-2 表面图像

我们要使用图 7-2 中的图像作为新的空气曲棍球桌子的表面，并把它加载进来作为纹理。

代码中使用到的所有图像都可以从本书的主页下载。我推荐把纹理图像存储到项目的“/res/drawable-nodpi/”文件夹中[⊖]。

每个二维的纹理都有其自己的坐标空间，其范围是从一个拐角的 (0, 0) 到另一个拐角的 (1, 1)。按照惯例，一个维度叫做 S，而另一个称为 T。当我们想要把一个纹理应用于一个三角形或一组三角形的时候，我们要为每个顶点指定一组 ST 纹理坐标，以便 OpenGL 知道需要用那个纹理的哪个部分画到每个三角形上。这些纹理坐标有时也会被称为 UV 纹理坐标，如图 7-3 所示。

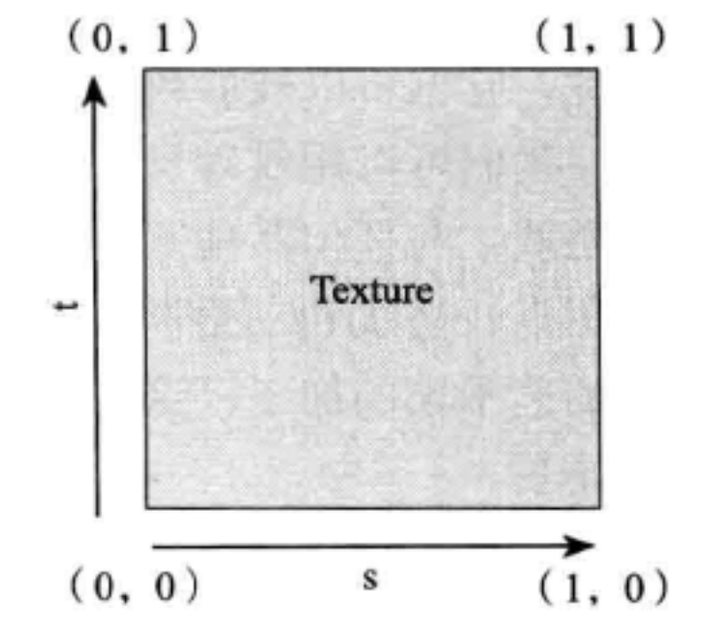


图 7-3 OpenGL 二维纹理坐标

对一个 OpenGL 纹理来说，它没有内在的方向性，因此我们可以使用不同的坐标把它定向到任何我们喜欢的方向上。然而，大多数计算机图像都有一个默认的方向，它们通常被规定为 y 轴向下（如图 7-4 所示），y 的值随着向图像的底部移动而增加。只要我们记住，如果想用正确的方向观看图像，那纹理坐标就必须要考虑这点，这就不会给我们带来任何麻烦。

在标准 OpenGL ES 2.0 中，纹理不必是正方形，但是每个维度都应该是 2 的幂（POT）。这就意味着每个维度都是这样的一个数字，如 128、256、512 等。这样规定的原因在于非 POT 纹理可以被使用的场合非常有限，而 POT 纹理适用于各种情况。

纹理的尺寸也有一个最大值，它根据不同的实现而变化，但是通常都比较大，比如 2048 × 2048。

⊖ http://pragprog.com/book/kbogla

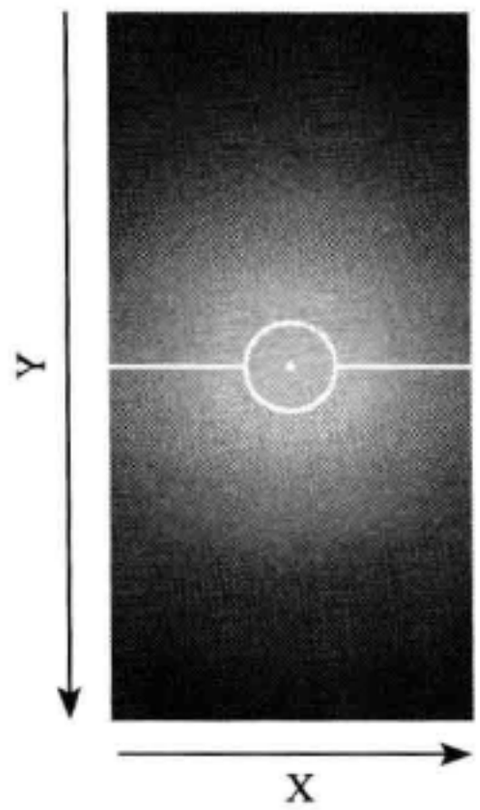


图 7-4　计算机图像：y 轴向下

7.2　把纹理加载进 OpenGL 中

我们的第一个任务是把一个图像文件的数据加载到一个 OpenGL 的纹理中。

作为开始，让我们在 com.airhockey.android.util 包里创建一个名为“TextureHelper”的新类。我们将以下面的方法签名开始：

AirHockeyTextured/src/com/airhockey/android/util/TextureHelper.java

```
public static int loadTexture(Context context, int resourceId) {
```

这个方法会把 Android 上下文（Context）和资源 ID 作为输入参数，并返回加载图像后的 OpenGL 纹理的 ID。开始时，我们会使用创建其他 OpenGL 对象时一样的模式生成一个新的纹理 ID：

AirHockeyTextured/src/com/airhockey/android/util/TextureHelper.java

```
final int[] textureObjectIds = new int[1];
glGenTextures(1, textureObjectIds, 0);

if (textureObjectIds[0] == 0) {
    if (LoggerConfig.ON) {
        Log.w(TAG, "Could not generate a new OpenGL texture object.");
    }
    return 0;
}
```

通过传递 1 作为第一个参数调用 glGenTextures(1, texttureObjectIds, 0)，我们就创建了一个纹理对象。OpenGL 会把那个生成的 ID 存储在 textureObjectIds 中。我们也检查了 glGenTextures() 调用是否成功，如果结果不等于 0 就继续，否则记录那个错误并返回 0。因为 TAG 还没有被定义，让我们在类的顶部为它加入如下定义：

AirHockeyTextured/src/com/airhockey/android/util/TextureHelper.java

```
private static final String TAG = "TextureHelper";
```

7.2.1 加载位图数据并与纹理绑定

下一步是使用 Android 的 API 读入图像文件的数据。OpenGL 不能直接读取 PNG 或者 JPEG 文件的数据，因为这些文件被编码为特定的压缩格式。OpenGL 需要非压缩形式的原始数据，因此，我们需要用 Android 内置的位图解码器把图像文件解压缩为 OpenGL 能理解的形式。

让我们继续实现 loadTexture()，把那个图像解压缩为一个 Android 位图：

AirHockeyTextured/src/com/airhockey/android/util/TextureHelper.java

```
final BitmapFactory.Options options = new BitmapFactory.Options();
options.inScaled = false;

final Bitmap bitmap = BitmapFactory.decodeResource(
    context.getResources(), resourceId, options);

if (bitmap == null) {
    if (LoggerConfig.ON) {
        Log.w(TAG, "Resource ID " + resourceId + " could not be decoded.");
    }

    glDeleteTextures(1, textureObjectIds, 0);
    return 0;
}
```

首先创建一个新的 BitmapFactory.Options 的实例，命名为“options”，并且设置 inScaled 为“false”。这告诉 Android 我们想要原始的图像数据，而不是这个图像的缩放版本。

接下来调用 BitmapFactory.decodeResource() 做实际的解码工作，把我们刚刚定义的 Android 上下文、资源 ID 和解码的 options 传递进去。这个调用会把解码后的图像存入 bitmap，如果失败就会返回空值（null）。我们检查了那个失败，如果位图是空值，那个 OpenGL 纹理对象会被删除。如果解码成功，就继续处理那个纹理。

在可以使用这个新生成的纹理对象做任何其他事之前，我们需要告诉 OpenGL 后面的纹理调用应该应用于这个纹理对象。我们为此使用一个 glBindTexture() 调用：

AirHockeyTextured/src/com/airhockey/android/util/TextureHelper.java

```
glBindTexture(GL_TEXTURE_2D, textureObjectIds[0]);
```

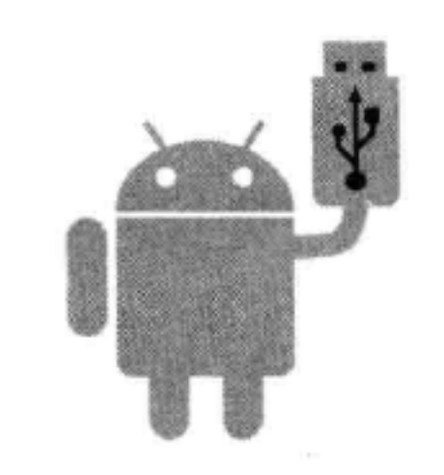
第一个参数 GL_TEXTURE_2D 告诉 OpenGL 这应该被作为一个二维纹理对待，第二个参数告诉 OpenGL 要绑定到哪个纹理对象的 ID。

7.2.2 理解纹理过滤

当纹理大小被扩大或者缩小时，我们还需要使用纹理过滤（texture filtering）明确说明会发生什么。当我们在渲染表面上绘制一个纹理时，那个纹理的纹理元素可能无法精确地映射到 OpenGL 生成的片段上。有两种情况：缩小和放大。当我们尽力把几个纹理元素挤进一个片段时，缩小就发生了；当我们把一个纹理元素扩展到许多片段时，放大就发生了。

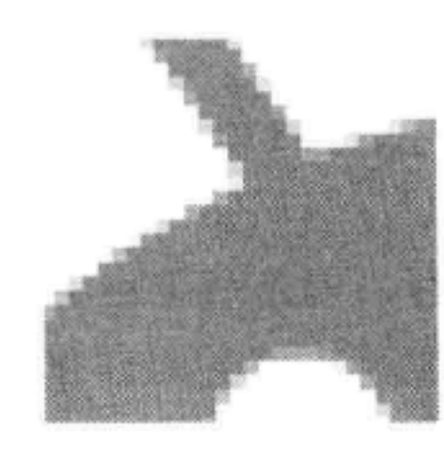
针对每一种情况，我们都可以配置 OpenGL 使用一个纹理过滤器。

首先，讲述两个基本的过滤模式：最近邻过滤（nearest-neighbor filtering）和双线性插值（bilinear interpolation）。还有其他的过滤模式，随后会详细讲解。我们会使用下面的图像阐述每一种过滤模式。



最近邻过滤

这个方式为每个片段选择最近的纹理元素。当我们放大纹理时，它的锯齿效果看起来相当明显，如下图所示。

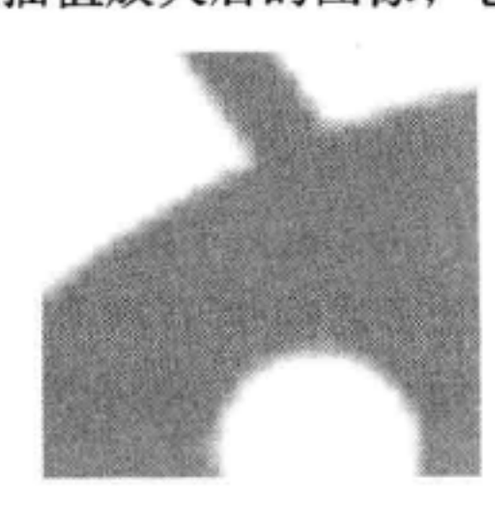


每个纹理单元都清楚地显示为一个小方块。

当我们缩小纹理时，因为没有足够的片段用来绘制所有的纹理单元，许多细节将会丢失。

双线性过滤

双线性过滤使用双线性插值平滑像素之间的过渡，而不是为每个片段使用最近的纹理元素，OpenGL 会使用四个邻接的纹理元素，并在它们之间用一个线性插值算法做插值，这个算法与我们在 4.3.2 节中讨论过的算法一样。我们之所以叫它双线性，是因为它是沿着两个维度插值的。下面是使用双线性插值放大后的图像，它采用的纹理与前面的相同。



这个纹理现在看起来比以前平滑多了。但还是有些锯齿显现出来，因为我们把这个纹理扩展得太多了，但是锯齿不像使用最近邻过滤时那么明显。

MIP 贴图

尽管双线性过滤很适合处理放大，但是对于缩小到超过一定的大小时，它就不好用了。一个纹理在渲染表面所占大小减少得越多，就会有越多的纹理元素拥挤到每一个片段上。因为 OpenGL 的双线性过滤只给每个片段使用四个纹理元素，我们将会失去很多细节。因为每一帧都要选择不同的纹理元素，这还会引起噪音以及移动中的物体的闪烁。

为了克服这些缺陷，可以使用 MIP 贴图（mipmapping）技术，它可以用来生成一组优化过的不同大小的纹理。当生成这组纹理的时候，OpenGL 会使用所有的纹理元素生成每个级别的纹理，当过滤纹理时，还要确保所有的纹理元素都能被使用。在渲染时，OpenGL 会根据每个片段的纹理元素数量为每个片段选择最适合的级别。

图 7-5 是一组 MIP 贴图的纹理，把它们合并在单个图上是为了方便对比。

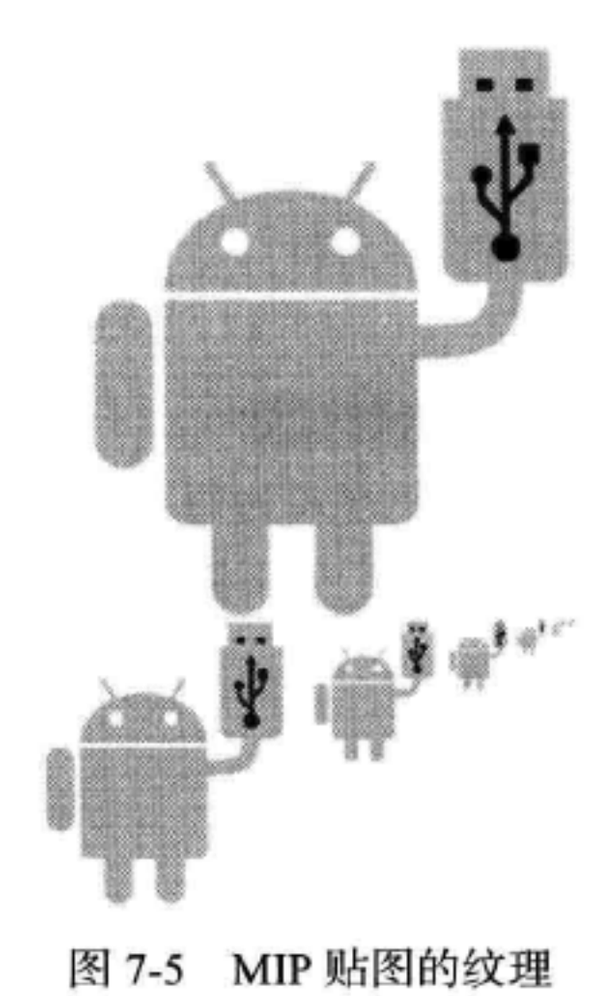


图 7-5 MIP 贴图的纹理

使用 MIP 贴图，会占用更多的内存，但是渲染也会更快，这是因为较小级别的纹理在 GPU 的纹理缓存中占用较少的空间。

为了更好地理解 MIP 贴图是如何提高缩小情况下的质量，我们比较一下那个可爱的机器人，使用双线性过滤把纹理元素尺寸缩小到其原来的 12.5%，如图 7-6 所示。

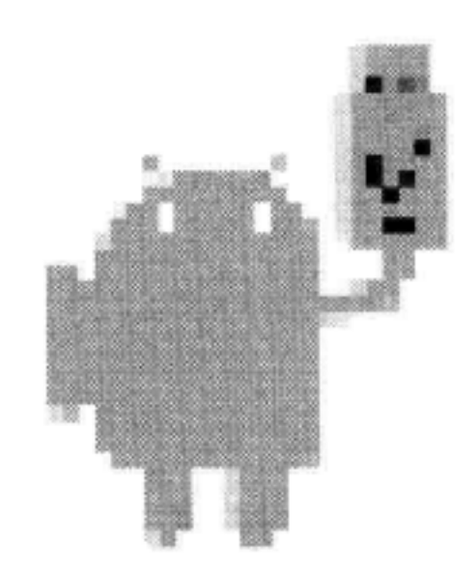


图 7-6 使用双线性过滤缩小

就这种质量，可能还不如用最近邻过滤。看一下当我们加入 MIP 贴图时会得到什么，如图 7-7 所示。

随着 MIP 贴图的使用，OpenGL 将选择最适合的纹理级别，然后用优化过的纹理做双线性插值。每个级别的纹理都是用来自所有纹理元素的信息构建的，因此得到的图形看起来更好些，保留了更多的细节。

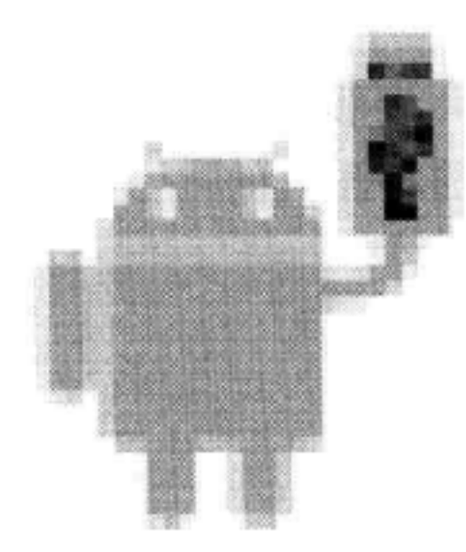


图 7-7 使用 MIP 贴图缩小

三线性过滤

如果 OpenGL 在不同的 MIP 贴图级别之间来回切换，当我们用双线性过滤来使用 MIP 贴图时，在其渲染的场景中，在不同级别的 MIP 贴图切换时，我们有时候能看到明显的跳跃或者线条。我们可以切换到三线性过滤（trilinear filtering），告诉 OpenGL 在两个最邻近的 MIP 贴图级别之间也要插值，这样，每个片段总共要使用 8 个纹理元素插值。这有助于消除每个

MIP 贴图级别之间的过渡，并且得到一个更平滑的图像。

7.2.3　设置默认的纹理过滤参数

既然已经了解了纹理过滤，我们继续 loadTexture()，并加入如下代码：

AirHockeyTextured/src/com/airhockey/android/util/TextureHelper.java

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

我们用一个 glTexParameteri() 调用设置每个过滤器：GL_TEXTURE_MIN_FILTER 是指缩小的情况，而 GL_TEXTURE_MAG_FILTER 指放大的情况。对于缩小的情况，我们选择 GL_LINEAR_MIPMAP_LINEAR，它告诉 OpenGL 使用三线性过滤；我们设置放大过滤器为 GL_LINEAR，它告诉 OpenGL 使用双线性过滤。

表 7-1 和表 7-2 揭示了可能的选项以及缩小和放大情况下有效的选项。

表 7-1　OpenGL 纹理过滤模式

GL_NEAREST	最近邻过滤
GL_NEAREST_MIPMAP_NEAREST	使用 MIP 贴图的最近邻过滤
GL_NEAREST_MIPMAP_LINEAR	使用 MIP 贴图级别之间插值的最近邻过滤
GL_LINEAR	双线性过滤
GL_LINEAR_MIPMAP_NEAREST	使用 MIP 贴图的双线性过滤
GL_LINEAR_MIPMAP_LINEAR	三线性过滤（使用 MIP 贴图级别之间插值的双线性过滤）

表 7-2　每种情况下允许的纹理过滤模式

缩小	GL_NEAREST GL_NEAREST_MIPMAP_NEAREST GL_NEAREST_MIPMAP_LINEAR GL_LINEAR GL_LINEAR_MIPMAP_NEAREST GL_LINEAR_MIPMAP_LINEAR
放大	GL_NEAREST GL_LINEAR

7.2.4　加载纹理到 OpenGL 并返回其 ID

我们现在可以用一个简单的 GLUtils.texImage2D() 调用加载位图数据到 OpenGL 里了：

AirHockeyTextured/src/com/airhockey/android/util/TextureHelper.java

```
texImage2D(GL_TEXTURE_2D, 0, bitmap, 0);
```

这个调用告诉 OpenGL 读入 bitmap 定义的位图数据，并把它复制到当前绑定的纹理对象。

既然这些数据已经被加载进 OpenGL 了，我们就不再需要持有 Android 的位图了。正常情况下，释放这个位图数据也会花费 Dalvik 的几个垃圾回收周期，因此我们应该调用 bitmap 对象的 recycle() 方法立即释放这些数据：

AirHockeyTextured/src/com/airhockey/android/util/TextureHelper.java

```
bitmap.recycle();
```

生成 MIP 贴图也是一件容易的事情。我们用一个快速的 glGenerateMipmap() 调用告诉 OpenGL 生成所有必要的级别：

AirHockeyTextured/src/com/airhockey/android/util/TextureHelper.java

```
glGenerateMipmap(GL_TEXTURE_2D);
```

既然我们完成了纹理的加载，一个很好的实践就是解除与这个纹理的绑定，这样我们就不会用其他纹理方法调用意外地改变这个纹理：

AirHockeyTextured/src/com/airhockey/android/util/TextureHelper.java

```
glBindTexture(GL_TEXTURE_2D, 0);
```

传递 0 给 glBindTexture() 就与当前的纹理解除绑定了。最后一步是返回纹理对象 ID：

AirHockeyTextured/src/com/airhockey/android/util/TextureHelper.java

```
return textureObjectIds[0];
```

我们现在有一个方法了，它可以从资源文件夹读入图像文件，并把图形数据加载进 OpenGL。我们也取回一个纹理 ID，它可被用做这个纹理的引用，如果加载失败，我们会得到 0。

7.3 创建新的着色器集合

在把纹理绘制到屏幕之前，我们不得不创建一套新的着色器，它们可以接收纹理，并把它应用在要绘制的片段上。这些新的着色器与我们目前为止使用过的着色器相似，只是为了支持纹理做了一些轻微的改动。

7.3.1 创建新的顶点着色器

在项目中的 /res/raw/ 目录下新建一个文件，命名为“texture_vertex_shader.glsl”，并加入如下内容：

AirHockeyTextured/res/raw/texture_vertex_shader.glsl

```
uniform mat4 u_Matrix;

attribute vec4 a_Position;
attribute vec2 a_TextureCoordinates;

varying vec2 v_TextureCoordinates;

void main()
{
    v_TextureCoordinates = a_TextureCoordinates;
    gl_Position = u_Matrix * a_Position;
}
```

这个着色器的大多数代码看上去应该都比较熟悉：我们已经为矩阵定义了一个 uniform，并且也为位置定义了一个属性。我们使用这些去设置最后的 gl_Position。而对于这些新的东西，我们同样给纹理坐标加了一个新的属性，它叫“a_TextureCoordinates”。因为它有两个分量：S 坐标和 T 坐标，所以被定义为一个 vec2。我们把这些坐标传递给顶点着色器被插值的 varying，称为 v_TextureCoordinates。

7.3.2　创建新的片段着色器

在同样的目录，创建一个叫做“texture_fragment_shader.glsl”的新文件，并加入如下内容：

AirHockeyTextured/res/raw/texture_fragment_shader.glsl

```
precision mediump float;

uniform sampler2D u_TextureUnit;
varying vec2 v_TextureCoordinates;

void main()
{
    gl_FragColor = texture2D(u_TextureUnit, v_TextureCoordinates);
}
```

为了把纹理绘制到一个物体上，OpenGL 会为每个片段都调用片段着色器，并且每个调用都接收 v_TextureCoordinates 的纹理坐标。片段着色器也通过 uniform——u_TextureUnit 接收实际的纹理数据，u_TextureUnit 被定义为一个 sampler2D，这个变量类型指的是一个二维纹理数据的数组。

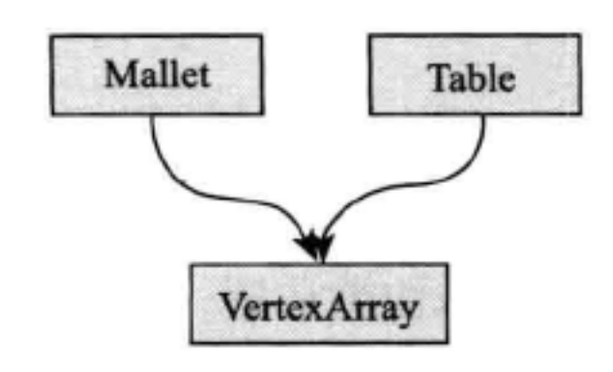
被插值的纹理坐标和纹理数据被传递给着色器函数 texture2D()，它会读入纹理中那个特定坐标处的颜色值。接着通过把结果赋值给 gl_FragColor 设置片段的颜色。

接下来的两节将稍微复杂些：我们要创建一套新的类集合，并把这些现有的桌子数据和着色器程序的代码放到这些类里。然后，在运行时切换它们。

7.4　为顶点数据创建新的类结构

首先，我们将把顶点数据分离到不同的类中，每个类代表一个物理对象的类型。我们将为桌子创建一个新类，并为木槌创建另一个类。因为纹理上已经有一条直线了，所以我们不需要给那个分隔线创建新类。

为了减少重复，我们也会创建独立的类，用于封装实际的顶点数组。新的类结构看上去如右图所示。



我们会创建 Mallet 类管理木槌的数据，以及 Table 类管理桌子的数据；并且每个类都会有一个 VertexArray 类的实例，它用来

封装存储顶点矩阵的 FloatBuffer。

我们将从 VertexArray 类开始。在你的项目中创建一个新的包，命名为“com.airhockey.android.data”，并在那个包中创建一个新类，命名为“VertexArray”。在该类内加入如下代码：

AirHockeyTextured/src/com/airhockey/android/data/VertexArray.java

```
private final FloatBuffer floatBuffer;

public VertexArray(float[] vertexData) {
    floatBuffer = ByteBuffer
        .allocateDirect(vertexData.length * BYTES_PER_FLOAT)
        .order(ByteOrder.nativeOrder())
        .asFloatBuffer()
        .put(vertexData);
}

public void setVertexAttribPointer(int dataOffset, int attributeLocation,
    int componentCount, int stride) {
    floatBuffer.position(dataOffset);
    glVertexAttribPointer(attributeLocation, componentCount, GL_FLOAT,
        false, stride, floatBuffer);
    glEnableVertexAttribArray(attributeLocation);

    floatBuffer.position(0);
}
```

这段代码包含一个 FloatBuffer，如我们在 2.4 节中解释的，它是用来在本地代码中存储顶点矩阵数据的。这个构建器取用一个 Java 的浮点数据数组，并把它写进这个缓冲区。

我们也创建了一个通用的方法把着色器中的属性与这些数据关联起来。它遵循我们在 3.4.3 节中解释过的同样的模式。

因为我们最终要在几个类中都使用 BYTES_PER_FLOAT，我们需要给它找个新地方。要做到这点，我们要在 com.airhockey.android 包中创建一个名为“Constants”的新类，并加入如下代码：

AirHockeyTextured/src/com/airhockey/android/Constants.java

```
package com.airhockey.android;

public class Constants {
    public static final int BYTES_PER_FLOAT = 4;
}
```

接下来，我们可以更新 VertexArray 类，让它使用这个新的常量。

7.4.1 加入桌子数据

现在我们将定义一个存储桌子数据的类，这个类会存储桌子的位置数据；我们还会加入纹理坐标，并把这个纹理应用于这个桌子。

添加类常量

创建一个名为“com.airhockey.android.objects”的新包；在这个新包中，创建一个名为

“Table”的新类，并在类内部加入如下代码：

AirHockeyTextured/src/com/airhockey/android/objects/Table.java

```
private static final int POSITION_COMPONENT_COUNT = 2;
private static final int TEXTURE_COORDINATES_COMPONENT_COUNT = 2;
private static final int STRIDE = (POSITION_COMPONENT_COUNT
    + TEXTURE_COORDINATES_COMPONENT_COUNT) * BYTES_PER_FLOAT;
```

如你所见，我们定义了位置分量计数、纹理坐标分量计数以及跨距。

添加顶点数据

下一步是用如下代码定义顶点数据：

AirHockeyTextured/src/com/airhockey/android/objects/Table.java

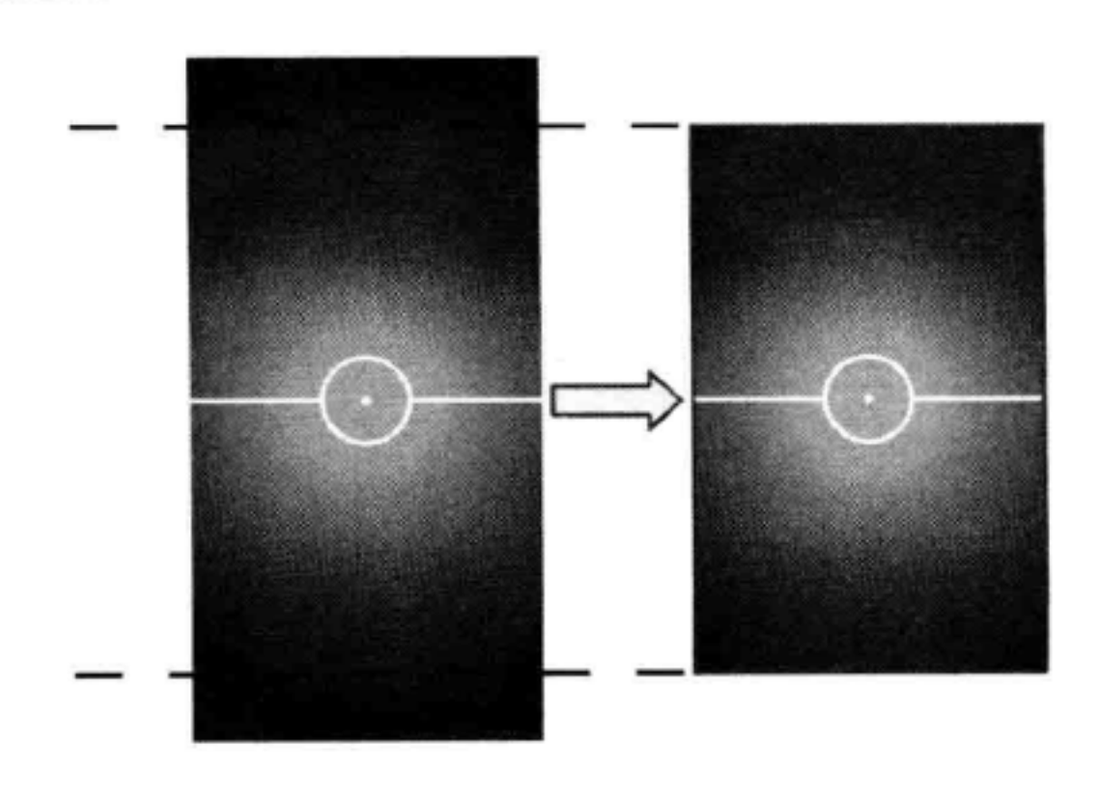
```
private static final float[] VERTEX_DATA = {
    // Order of coordinates: X, Y, S, T

    // Triangle Fan
       0f,    0f, 0.5f, 0.5f,
    -0.5f, -0.8f,   0f, 0.9f,
     0.5f, -0.8f,   1f, 0.9f,
     0.5f,  0.8f,   1f, 0.1f,
    -0.5f,  0.8f,   0f, 0.1f,
    -0.5f, -0.8f,   0f, 0.9f };
```

这个数组包含了空气曲棍球桌子的顶点数据。我们也定义了 x 和 y 的位置，以及 S 和 T 纹理坐标。你可能注意到了那个 T 分量正是按那个 y 分量相反的方向定义的。之所以会这样，如我们在 7.1 节中解释的，图像的朝向是右边向上的。当我们使用一个对称的纹理时，这一点实际上没有关系，但是在其他情况下，这就会有问题了，因此一定要记住这个规则。

裁剪纹理

我们还使用了 0.1f 和 0.9f 作为 T 坐标。为什么？这个桌子是 1 个单位宽、1.6 个单位高，而纹理图像是 512 × 1024 像素，因此，如果它的宽度对应 1 个单位，那纹理的高实际就是 2 个单位。为了避免把纹理压扁，我们使用范围 0.1 到 0.9 裁剪它的边缘，而不是用 0.0 到 1.0，并且只画它的中间部分。

下图阐释了这个概念：



即使不使用裁剪，我们还可以坚持使用从 0.0 到 1.0 的纹理坐标，把纹理预拉伸，这样被压扁到空气曲棍球桌子之后，它看上去就是正确的了。采用这种方法，那些无法显示的纹理部分就不会占用任何内存了（见图 7-8）。

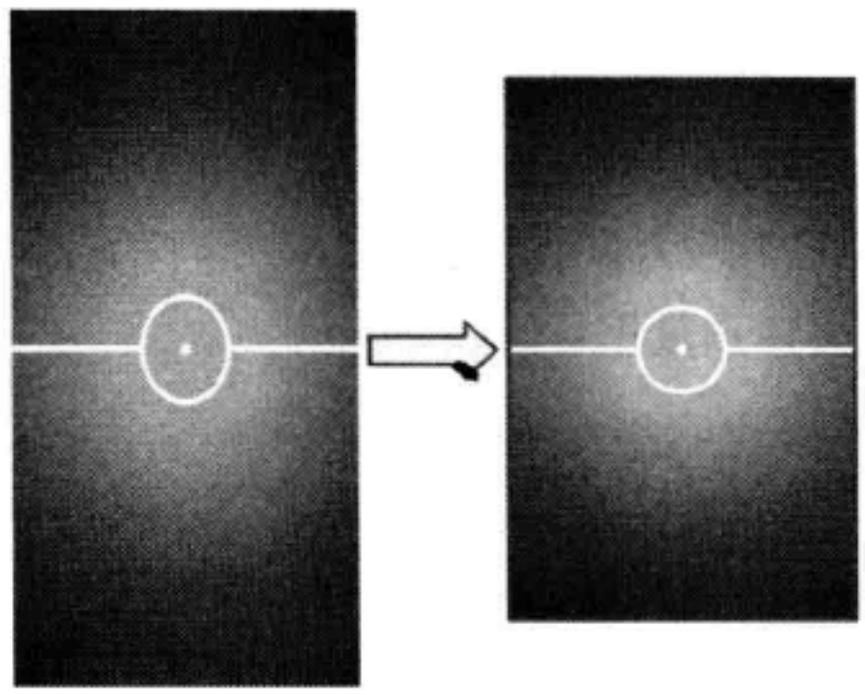


图 7-8 预拉伸

初始化和绘制数据

现在为 Table 类创建一个构造函数。这个构造函数会使用 VertexArray 把数据复制到本地内存中的一个 FloatBuffer。

AirHockeyTextured/src/com/airhockey/android/objects/Table.java

```
private final VertexArray vertexArray;

public Table() {
    vertexArray = new VertexArray(VERTEX_DATA);
}
```

添加一个方法把顶点数组绑定到一个着色器程序上：

AirHockeyTextured/src/com/airhockey/android/objects/Table.java

```
public void bindData(TextureShaderProgram textureProgram) {
    vertexArray.setVertexAttribPointer(
        0,
        textureProgram.getPositionAttributeLocation(),
        POSITION_COMPONENT_COUNT,
        STRIDE);

    vertexArray.setVertexAttribPointer(
        POSITION_COMPONENT_COUNT,
        textureProgram.getTextureCoordinatesAttributeLocation(),
        TEXTURE_COORDINATES_COMPONENT_COUNT,
        STRIDE);
}
```

这个方法为每个顶点调用了 setVertexAttribPointer()，并从着色器程序获取每个属性的位置。它通过 getPositionAttributeLocation() 把位置数据绑定到被引用的着色器属性上，并通过 getTextureCoordinatesAttributeLocation() 把纹理坐标数据绑定到被引用的着色器属性上。当我们创建着色器的类时，会定义这些方法。

我们只需要加入最后一个方法就可以画出这张桌子了：

AirHockeyTextured/src/com/airhockey/android/objects/Table.java

```
public void draw() {
    glDrawArrays(GL_TRIANGLE_FAN, 0, 6);
}
```

7.4.2　加入木槌数据

在同一个包中创建另一个类，命名为“Mallet”。在这个类中加入如下代码：

AirHockeyTextured/src/com/airhockey/android/objects/Mallet.java

```
private static final int POSITION_COMPONENT_COUNT = 2;
private static final int COLOR_COMPONENT_COUNT = 3;
private static final int STRIDE =
    (POSITION_COMPONENT_COUNT + COLOR_COMPONENT_COUNT)
    * BYTES_PER_FLOAT;
private static final float[] VERTEX_DATA = {
    // Order of coordinates: X, Y, R, G, B
    0f, -0.4f, 0f, 0f, 1f,
    0f,  0.4f, 1f, 0f, 0f };
private final VertexArray vertexArray;

public Mallet() {
    vertexArray = new VertexArray(VERTEX_DATA);
}

public void bindData(ColorShaderProgram colorProgram) {
    vertexArray.setVertexAttribPointer(
        0,
        colorProgram.getPositionAttributeLocation(),
        POSITION_COMPONENT_COUNT,
        STRIDE);
    vertexArray.setVertexAttribPointer(
        POSITION_COMPONENT_COUNT,
        colorProgram.getColorAttributeLocation(),
        COLOR_COMPONENT_COUNT,
        STRIDE);
}

public void draw() {
    glDrawArrays(GL_POINTS, 0, 2);
}
```

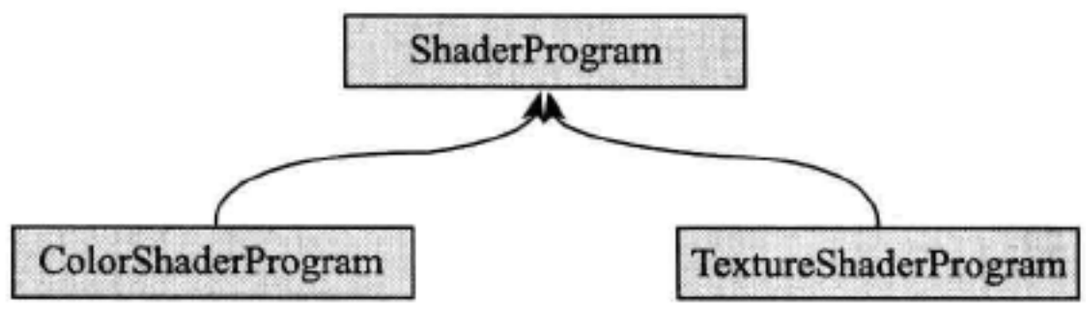
它遵循与 Table 类一样的模式，与之前一样，我们还是把木槌画为点。

顶点数据现在被定义好了：我们有一个类表示桌子数据，另一个表示木槌数据，第三个类使得更容易管理顶点数据本身。下一步是为着色器程序定义类。

7.5　为着色器程序添加类

在本节中，我们会为纹理着色器程序创建一个类，并为颜色着色器程序创建另一个类；

我们会用纹理着色器绘制桌子，用颜色着色器绘制木槌。我们也会创建一个基类作为它们的公共函数。我们不用再担心那条直线，因为它是纹理的一部分。



我们开始给 ShaderHelper 加入一个辅助函数。打开那个类，在其尾部加入如下方法：

AirHockeyTextured/src/com/airhockey/android/util/ShaderHelper.java

```
public static int buildProgram(String vertexShaderSource,
    String fragmentShaderSource) {
    int program;

    // Compile the shaders.
    int vertexShader = compileVertexShader(vertexShaderSource);
    int fragmentShader = compileFragmentShader(fragmentShaderSource);

    // Link them into a shader program.
    program = linkProgram(vertexShader, fragmentShader);

    if (LoggerConfig.ON) {
        validateProgram(program);
    }

    return program;
}
```

这个辅助函数会编译 vertexShaderSource 和 fragmentShaderSource 定义的着色器，并把它们链接在一起成为一个程序。如果日志是打开的，它也会验证这个程序。我们会使用这个辅助函数组成我们的基类。

创建一个名为“com.airhockey.android.programs”的包，并在包中创建一个名为“ShaderProgram”的新类。在类内部加入如下代码：

AirHockeyTextured/src/com/airhockey/android/programs/ShaderProgram.java

```
// Uniform constants
protected static final String U_MATRIX = "u_Matrix";
protected static final String U_TEXTURE_UNIT = "u_TextureUnit";

// Attribute constants
protected static final String A_POSITION = "a_Position";
protected static final String A_COLOR = "a_Color";
protected static final String A_TEXTURE_COORDINATES = "a_TextureCoordinates";

// Shader program
protected final int program;
protected ShaderProgram(Context context, int vertexShaderResourceId,
    int fragmentShaderResourceId) {
    // Compile the shaders and link the program.
    program = ShaderHelper.buildProgram(
```

```
        TextResourceReader.readTextFileFromResource(
            context, vertexShaderResourceId),
        TextResourceReader.readTextFileFromResource(
            context, fragmentShaderResourceId));
}

public void useProgram() {
    // Set the current OpenGL shader program to this program.
    glUseProgram(program);
}
```

我们通过定义一些公用的常量作为这个类的开始。在构造函数中，我们调用刚刚定义过的辅助函数，其使用指定的着色器构建了一个 OpenGL 着色器程序。我们用 useProgram() 作为结束，其调用 glUseProgram() 告诉 OpenGL 接下来的渲染要使用这个程序。

7.5.1　加入纹理着色器程序

我们现在将定义一个类来建立和表示纹理着色器程序。

创建一个名为 TextureShaderProgram 的新类，其继承自 ShaderProgram，并在该类内部加入如下代码：

AirHockeyTextured/src/com/airhockey/android/programs/TextureShaderProgram.java

```
// Uniform locations
private final int uMatrixLocation;
private final int uTextureUnitLocation;

// Attribute locations
private final int aPositionLocation;
private final int aTextureCoordinatesLocation;
```

我们加入了四个整型用于保存那些 uniform 和属性的位置。

初始化着色器程序

下一步是创建用于初始化着色器程序的构造函数。我们加入如下代码：

AirHockeyTextured/src/com/airhockey/android/programs/TextureShaderProgram.java

```
public TextureShaderProgram(Context context) {
    super(context, R.raw.texture_vertex_shader,
        R.raw.texture_fragment_shader);

    // Retrieve uniform locations for the shader program.
    uMatrixLocation = glGetUniformLocation(program, U_MATRIX);
    uTextureUnitLocation = glGetUniformLocation(program, U_TEXTURE_UNIT);

    // Retrieve attribute locations for the shader program.
    aPositionLocation = glGetAttribLocation(program, A_POSITION);
    aTextureCoordinatesLocation =
        glGetAttribLocation(program, A_TEXTURE_COORDINATES);
}
```

这个构造函数会用我们选择的资源调用其父类的构造函数，其父类会构造着色器程序。我们将读入并保存那些 uniform 和属性的位置。

设置 uniform 并返回属性位置

传递矩阵和纹理给它们的 uniform。加入如下代码：

AirHockeyTextured/src/com/airhockey/android/programs/TextureShaderProgram.java

```
public void setUniforms(float[] matrix, int textureId) {
    // Pass the matrix into the shader program.
    glUniformMatrix4fv(uMatrixLocation, 1, false, matrix, 0);

    // Set the active texture unit to texture unit 0.
    glActiveTexture(GL_TEXTURE0);

    // Bind the texture to this unit.
    glBindTexture(GL_TEXTURE_2D, textureId);

    // Tell the texture uniform sampler to use this texture in the shader by
    // telling it to read from texture unit 0.
    glUniform1i(uTextureUnitLocation, 0);
}
```

第一步是传递矩阵给它的 uniform，这足够简单明了。下一部分就需要更多的解释了。当我们在 OpenGL 里使用纹理进行绘制时，我们不需要直接给着色器传递纹理。相反，我们使用纹理单元（texture unit）保存那个纹理。之所以这样做，是因为一个 GPU 只能同时绘制数量有限的纹理。它使用这些纹理单元表示当前正在被绘制的活动的纹理。

如果需要切换纹理，我们可以在纹理单元中来回切换纹理，但是，如果我们切换得太频繁，可能会拖慢渲染的速度。也可以同时用几个纹理单元绘制多个纹理。

通过调用 glActiveTexture() 把活动的纹理单元设置为纹理单元 0，我们以此开始，然后通过调用 glBindTexture() 把纹理绑定到这个单元。接着，通过调用 glUniform1i(uTextureUnitLocation, 0) 把被选定的纹理单元传递给片段着色器中的 u_TextureUnit。

我们几乎已经完成了这个纹理着色器类；只需要一种方法来获取属性的位置，以便可以把它们绑定到正确的顶点数组数据矩阵。加入如下代码完成这个类：

AirHockeyTextured/src/com/airhockey/android/programs/TextureShaderProgram.java

```
public int getPositionAttributeLocation() {
    return aPositionLocation;
}

public int getTextureCoordinatesAttributeLocation() {
    return aTextureCoordinatesLocation;
}
```

7.5.2 加入颜色着色器程序

在同一个包中创建另一个类，并命名为“ ColorShaderProgram”。这个类应该也继承自 ShaderProgram，它也遵循与 TextureShaderProgram 一样的模式：有一个构造函数、一个设置 uniform 的方法和获取属性位置的方法。在此类内部加入如下代码：

AirHockeyTextured/src/com/airhockey/android/programs/ColorShaderProgram.java

```
// Uniform locations
```

```
private final int uMatrixLocation;

// Attribute locations
private final int aPositionLocation;
private final int aColorLocation;

public ColorShaderProgram(Context context) {
    super(context, R.raw.simple_vertex_shader,
        R.raw.simple_fragment_shader);
    // Retrieve uniform locations for the shader program.
    uMatrixLocation = glGetUniformLocation(program, U_MATRIX);
    // Retrieve attribute locations for the shader program.
    aPositionLocation = glGetAttribLocation(program, A_POSITION);
    aColorLocation = glGetAttribLocation(program, A_COLOR);
}
public void setUniforms(float[] matrix) {
    // Pass the matrix into the shader program.
    glUniformMatrix4fv(uMatrixLocation, 1, false, matrix, 0);
}

public int getPositionAttributeLocation() {
    return aPositionLocation;
}

public int getColorAttributeLocation() {
    return aColorLocation;
}
```

我们会使用这个项目绘制木槌。

通过把这些着色器程序与这些程序要绘制的数据进行解耦，就很容易重用这些代码了。比如，我们可以通过这个颜色着色器程序用一个颜色属性绘制任何物体，而不仅仅是木槌。

7.6　绘制纹理

既然我们已经把顶点数据和着色器程序分别放于不同的类中了，现在就可以更新渲染类，使用纹理进行绘制了。打开 AirHockeyRenderer，删掉所有代码，只保留 onSurfaceChanged()，这是我们唯一不会改变的方法。加入如下成员变量和构造函数：

AirHockeyTextured/src/com/airhockey/android/AirHockeyRenderer.java

```
private final Context context;

private final float[] projectionMatrix = new float[16];
private final float[] modelMatrix = new float[16];

private Table table;
private Mallet mallet;

private TextureShaderProgram textureProgram;
private ColorShaderProgram colorProgram;
```

```
private int texture;

public AirHockeyRenderer(Context context) {
    this.context = context;
}
```

我们保留了上下文和矩阵的变量，并添加了顶点数组、着色器程序和纹理的变量。这个构造函数被简化为只保存一个 Android 上下文的引用。

7.6.1 初始化变量

加入如下的 onSurfaceCreated() 初始化这些新的变量：

AirHockeyTextured/src/com/airhockey/android/AirHockeyRenderer.java

```
@Override
public void onSurfaceCreated(GL10 glUnused, EGLConfig config) {
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

    table = new Table();
    mallet = new Mallet();

    textureProgram = new TextureShaderProgram(context);
    colorProgram = new ColorShaderProgram(context);

    texture = TextureHelper.loadTexture(context, R.drawable.air_hockey_surface);
}
```

我们把清屏颜色设置为黑色，初始化顶点数组和着色器程序，并用在 7.2 节中定义的辅助函数加载纹理。

7.6.2 使用纹理进行绘制

不再赘述 onSurfaceChanged()，因为它还是保持不变的。加入如下 onDrawFrame() 绘制桌子和木槌：

AirHockeyTextured/src/com/airhockey/android/AirHockeyRenderer.java

```
@Override
public void onDrawFrame(GL10 glUnused) {
    // Clear the rendering surface.
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw the table.
    textureProgram.useProgram();
    textureProgram.setUniforms(projectionMatrix, texture);
    table.bindData(textureProgram);
    table.draw();

    // Draw the mallets.
    colorProgram.useProgram();
    colorProgram.setUniforms(projectionMatrix);
    mallet.bindData(colorProgram);
    mallet.draw();
}
```

我们清空了渲染表面，接下来，我们做的第一件事是绘制桌子。我们首先调用 textureProgram.useProgram() 告诉 OpenGL 使用这个程序，然后通过调用 textureProgram.setUniforms() 把那些 uniform 传递进去。下一步是通过调用 table.bindData() 把顶点数组数据和着色器程序绑定起来。最后调用 table.draw() 绘制桌子。

我们重复同样的调用顺序，用颜色着色器程序绘制了木槌。

7.6.3 运行程序并查看结果

运行这个程序，它看起来应该与图 7-1 类似。即使没有相应的 OpenGL 错误，你也许注意到一个错误出现在你的 logcat 调试日志中，如“E/IMGSRV(20095): :0: HardwareMipGen: Failed to generate texture mipmap levels (error=3)”。这也许意味着，当调用 glGenerateMipMap() 时，你的设备不支持非正方形纹理。

对于这个错误，一个容易的修复是压扁纹理，以使它变为正方形。因为纹理是被应用在一个长方形表面的，它会被拉伸，并且看上去还是与以前一样。

7.7 小结

现在我们知道如何加载纹理并把它显示在空气曲棍球桌子上；我们也学习了如何重新组织我们的程序，以便在代码中可以容易地在多个着色器程序和顶点数组之间切换。我们调整纹理来适应它们将要被绘制的形状，既可以通过调整纹理坐标，也可以通过拉伸或压扁纹理本身来实现。

纹理不会被直接绘制，它们要被绑定到纹理单元，然后把这些纹理单元传递给着色器。通过在纹理单元中把纹理切来切去，我们还可以在场景中绘制不同的纹理，但是过分的切换可能使性能下降。我们也能同时使用多个纹理单元绘制几个纹理。

7.8 练习

尝试加载一个不同的图像，并用另外一个纹理单元把这个图像与当前的图像混合（blend）起来。当你在片段着色器中把那些值赋给 gl_FragColor 时，你可以尝试把它们加在一起或者乘在一起。

一旦你完成了这些练习，我们将开始学习如何提高木槌的外观。

Chapter 8 第 8 章

构建简单物体

我们的空气曲棍球项目已经取得了很大进展：桌子现在放在了一个好的角度，并且由于使用了纹理，更加好看了。然而，因为我们当前只是用一个点绘制每个木槌，它们实际上看起来还不像一个真正的木槌。你能想象用一个小圆点作为木槌玩空气曲棍球么？许多应用都是合并简单的图形构建更复杂的物体，接下来就将学习如何那样做，以使我们可以构建一个更好的木槌。

我们还缺少一个容易的方法在场景中平移、旋转和来回移动，许多三维应用程序使用一个视图（view）矩阵实现这些；对这个矩阵所做的改动会影响整个场景，就像我们从一个移动相机中观察事物一样。我们将添加一个视图矩阵使它更容易旋转和来回移动。

浏览一下本章的开发计划：

- 首先，我们会学习如何把三角形组织成为三角形带和三角形扇，并把它们合并在一起成为一个单个物体；
- 我们也会学习如何定义视图矩阵，并把它集成到我们的矩阵层次结构中。

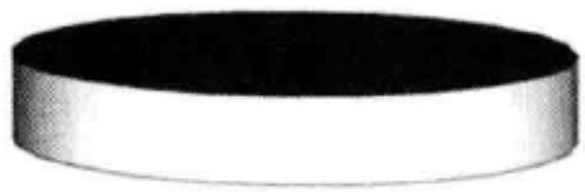
一旦完成了这些任务，只用一行代码就能移动这个场景了，而且我们也会构建更好看的木槌，它们看起来可以实际用来击打冰球了。说到冰球，场景中现在还没有，因此我们也要添加它。

首先，从前一章复制一个新的项目，并命名为“AirHockeyWithBetterMallets”。

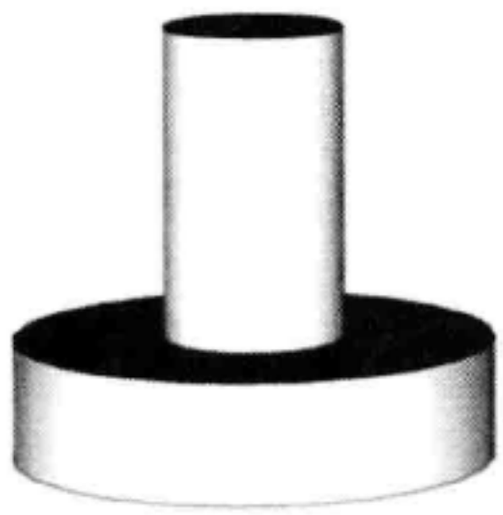
8.1 合并三角形带和三角形扇

对于要构建一个木槌，或者一个冰球这样的事情。让我们首先尝试在一个较高的层次

上想象一下它的形状。一个冰球可以被表示为一个扁平的圆柱体，如下图所示：



木槌就有点复杂了，它可以被表示为两个圆柱体，其中一个在另一个的上面，如下图所示：

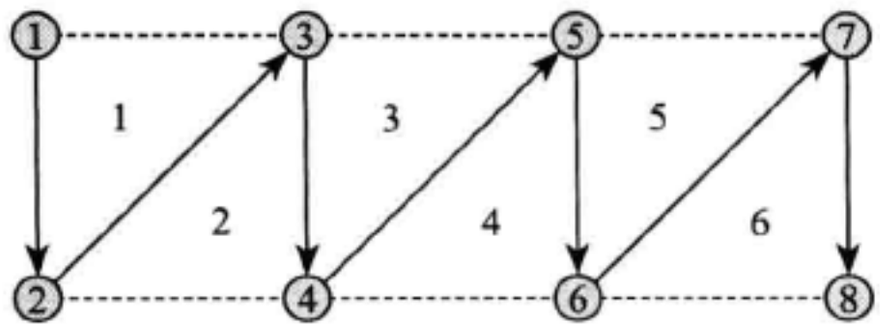


我知道你正在想什么：一个真实的空气曲棍球木槌不仅仅只是两个粘在一起的圆柱体。忍耐一下，一旦我们学会了这些基础，你就会感觉很有信心去构建一个更真实的木槌了。

为了弄清楚如何在 OpenGL 中构建这些物体。让我们想象一下怎样用纸张来构建它们：首先，我们要剪出一个圆形作为圆柱的顶；然后，取出一张平面的纸，把它剪成适合的大小，并卷成一个管子。要组成这个圆柱体，接下来就把那个圆放在这个管子上。我们需要一个圆柱体做冰球，两个做木槌。

结果证明在 OpenGL 中相当容易实现。要构建圆，我们可以使用一个三角形扇。我们第一次讲到三角形扇是在 4.2 书中，那时我们通过它用六个顶点和四个三角形构建了一张桌子。我们也能用一个三角形扇表示一个圆；我们只需要使用大量的三角形，并按圆形排列外面的顶点。

要构建圆柱的侧面，我们可以用一个相关的概念，它被称为三角形带（Triangle Strip）。与三角形扇一样，三角形带让我们定义多个三角形而不用一遍又一遍地重复那些三角形中共有的点；但是它不是绕着一个圆扇形展开的，三角形带像大桥横梁一样排列，那些三角形彼此相邻放置，如下图所示：



正如三角形扇一样，一个三角形带的前三个顶点定义了第一个三角形。这之后的每个额外的顶点都定义了另外的一个三角形。为了使用三角形带定义这个圆柱体的侧面，我们

只需要把这个带卷绕成一个管子，并确保最后两个顶点与最前面的两个顶点一致。

Joe 问：为什么不直接使用库

对 Android 来说，已经有很多合适的三维库了，从简单的开源封装——libgdx[⊖]，再到比较高级的商业化框架，比如 Unity3D[⊖]。这些库能帮助你提高生产率，但只有当你对 OpenGL、三维渲染，以及底层是如何把这些东西拼在一起的有了基本的理解后才能体会到；否则这些库没有任何意义，你也许感觉像使用黑箱魔法一样。举个例子，如 Spring 和 Hibernate 等框架使 Java 效率更高，但是如果都不知道 Java 是如何工作的，就开始使用它们，是不是为时尚早。

研究这些库有助于学习如何开发自己的组件。Java3D 和 jMonkeyEngine 是 Java 桌面版上广泛使用的框架；因为它们的文档很容易从网上得到[⊖]，它们是很好的学习起点。

8.2 添加几何图形的类

要构建冰球和木槌，我们现在很清楚需要什么了：对于冰球，我们需要一个三角形扇做顶和一个三角形带做侧面；对于木槌，我们需要两个三角形扇和两个三角形带。为了更容易地构建这些物体，我们将定义一个几何图形的类，它可以容纳一些基本的图形定义，以及一个用来做实际构建工作的 ObjectBuilder。

让我们以这个几何物体的类作为开始。在 com.airhockey.android.util 包中创建一个叫做“Geometry”的新类。在类内部加入如下代码：

AirHockeyWithImprovedMallets/src/com/airhockey/android/util/Geometry.java

```
public static class Point {
    public final float x, y, z;
    public Point(float x, float y, float z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public Point translateY(float distance) {
        return new Point(x, y + distance, z);
    }
}
```

我们已经添加了一个类用来表示三维场景中的一个点，其中有一个辅助函数用于把这

⊖ http://code.google.com/p/libgdx/

⊖ http://unity3d.com/

⊖ http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138252.html 和 http://jmonkeyengine.com/

个点沿着 y 轴平移。我们也需要给圆一个定义；在 Point 类后面加入如下代码：

AirHockeyWithImprovedMallets/src/com/airhockey/android/util/Geometry.java

```
public static class Circle {
    public final Point center;
    public final float radius;

    public Circle(Point center, float radius) {
        this.center = center;
        this.radius = radius;
    }

    public Circle scale(float scale) {
        return new Circle(center, radius * scale);
    }
}
```

我们同样有一个辅助函数用于缩放圆的半径。最后是给圆柱体一个定义：

AirHockeyWithImprovedMallets/src/com/airhockey/android/util/Geometry.java

```
public static class Cylinder {
    public final Point center;
    public final float radius;
    public final float height;

    public Cylinder(Point center, float radius, float height) {
        this.center = center;
        this.radius = radius;
        this.height = height;
    }
}
```

一个圆柱体就像一个扩展的圆，它有一个中心、一个半径和一个高度。

你可能注意到了我们已经把这几个几何物体的类定义为不可变的（immutable）；无论任何时候改动它，都会返回一个新的对象。这有助于使代码更容易使用和理解，但是当你需要提高性能时，你也许想一直用简单的浮点数组，并用静态函数改变它们。

8.3 添加物体构建器

我们现在可以开始写物体构建器类了。在 com.airhockey.android.objects 包中创建一个名为“ObjectBuilder”的类。在类内部，以下面的代码作为开始：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
private static final int FLOATS_PER_VERTEX = 3;
private final float[] vertexData;
private int offset = 0;

private ObjectBuilder(int sizeInVertices) {
    vertexData = new float[sizeInVertices * FLOATS_PER_VERTEX];
}
```

目前为止，没有什么太复杂的。我们定义了一个常量用来表示一个顶点需要多少浮点

数，一个数组用于保存这些顶点，以及一个变量用于记录数组中下一个顶点的位置。这个构造函数基于需要的顶点数量初始化了数组。

我们不久将定义两个静态方法用来生成冰球和木槌。这些静态方法会使用正确的大小创建一个新的 ObjectBuilder 实例，调用 ObjectBuilder 实例中的方法给 vertexData 添加顶点，并把生成的数据返回给调用者。

对于这个物体构建器应该怎样工作，我们给出了一些需求：

- 调用者可以决定物体应该有多少个点。点越多，冰球和木槌看上去就越平滑。
- 物体将被包含在一个浮点数组中。物体被创建后，调用者将有一个绑定到 OpenGL 的数组和一个绘制物体的命令。
- 物体将以调用者指定的位置为中心，并平放在 x-z 平面上，换句话说，物体的顶部要指向正上方。

我们要用一个计算圆柱体顶部顶点数量的方法作为开始：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
private static int sizeOfCircleInVertices(int numPoints) {
    return 1 + (numPoints + 1);
}
```

一个圆柱体的顶部是一个用三角形扇构造的圆；它有一个顶点在圆心，围着圆的每个点都有一个顶点，并且围着圆的第一个顶点要重复两次才能使圆闭合。

下面计算圆柱体侧面顶点的数量：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
private static int sizeOfOpenCylinderInVertices(int numPoints) {
    return (numPoints + 1) * 2;
}
```

一个圆柱体的侧面是一个卷起来的长方形，由一个三角形带构造，围着顶部圆的每个点都需要两个顶点，并且前两个顶点要重复两次才能使这个管闭合。

8.3.1 用圆柱体创建冰球

我们现在能创建一个静态方法来生成一个冰球了。继续创建一个名为 createPuck() 的新方法，做如下定义：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

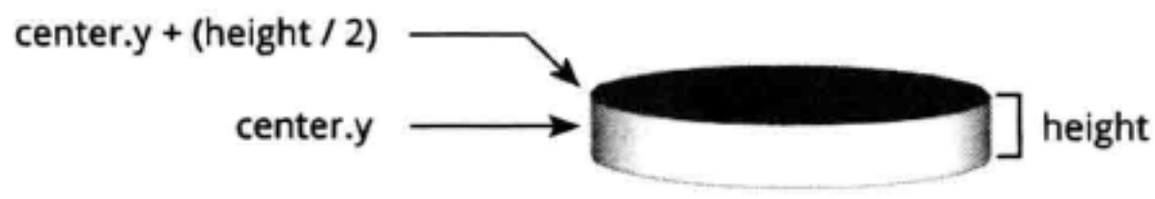
```
static GeneratedData createPuck(Cylinder puck, int numPoints) {
    int size = sizeOfCircleInVertices(numPoints)
             + sizeOfOpenCylinderInVertices(numPoints);

    ObjectBuilder builder = new ObjectBuilder(size);

    Circle puckTop = new Circle(
        puck.center.translateY(puck.height / 2f),
        puck.radius);
```

```
        builder.appendCircle(puckTop, numPoints);
        builder.appendOpenCylinder(puck, numPoints);

        return builder.build();
    }
```

我们所做的第一件事情是找出需要多少个顶点表示这个冰球，然后用那个数量实例化一个新的 ObjectBuilder。一个冰球由一个圆柱体的顶部（就是一个圆）和一个圆柱体的侧面构成，因此所有顶点的数量就等于 sizeOfCircleInVertices(numPoints) + sizeOfOpenCylinderInVertices(numPoints)。

然后，我们要计算冰球的顶部应该放在哪里，并调用 appendCircle() 创建它。通过调用 appendOpenCylinder()，我们也生成了冰球的侧面，之后通过返回 build() 的结果返回生成的数据。这些方法都不存在，因此我们需要创建它们。

我们为什么要把冰球的顶部移动 puck.height/2f 呢？看一下下图：



这个冰球垂直方向以 center.y 为中心，因此在那放置圆柱体的侧面没有问题。然而，圆柱体顶部需要被放在冰球的顶部，要想这样做，我们需要把它向上移动冰球整体高度的一半。

8.3.2　用三角形扇构造圆

下一步是写代码用一个三角形扇构造这个冰球的顶部。我们会把数据写进 vertexData，并使用 offset 记录我们写到数组的哪个位置了。

创建一个名为 appendCircle() 的新方法，并加入如下代码：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
private void appendCircle(Circle circle, int numPoints) {
    // Center point of fan
    vertexData[offset++] = circle.center.x;
    vertexData[offset++] = circle.center.y;
    vertexData[offset++] = circle.center.z;

    // Fan around center point. <= is used because we want to generate
    // the point at the starting angle twice to complete the fan.
    for (int i = 0; i <= numPoints; i++) {
        float angleInRadians =
              ((float) i / (float) numPoints)
            * ((float) Math.PI * 2f);

        vertexData[offset++] =
              circle.center.x
            + circle.radius * FloatMath.cos(angleInRadians);
        vertexData[offset++] = circle.center.y;
        vertexData[offset++] =
              circle.center.z
```

```
                + circle.radius * FloatMath.sin(angleInRadians);
        }
    }
```

要构建三角形扇，我们首先在 circle.center 定义一个圆心顶点，接着，我们围绕圆心的点按扇形展开，并把第一个点绕圆周重复两次考虑在内。我们接下来使用三角函数和单位圆（见图 8-1）的概念生成那些点。

为了生成沿一个圆周边的点，我们首先需要一个循环，它的范围涵盖从 0 到 360 度的整个圆，或者 0 到 2π 弧度。要找到圆周上的一个点的 x 的位置，我们要调用 cos(angle)，要找到它的 z 的位置，我们调用 sin(angle)；我们用圆的半径缩放这两个位置。

因为这个圆将被平放在 x-z 平面上，单位圆的 y 分量就会映射到 y 的位置。

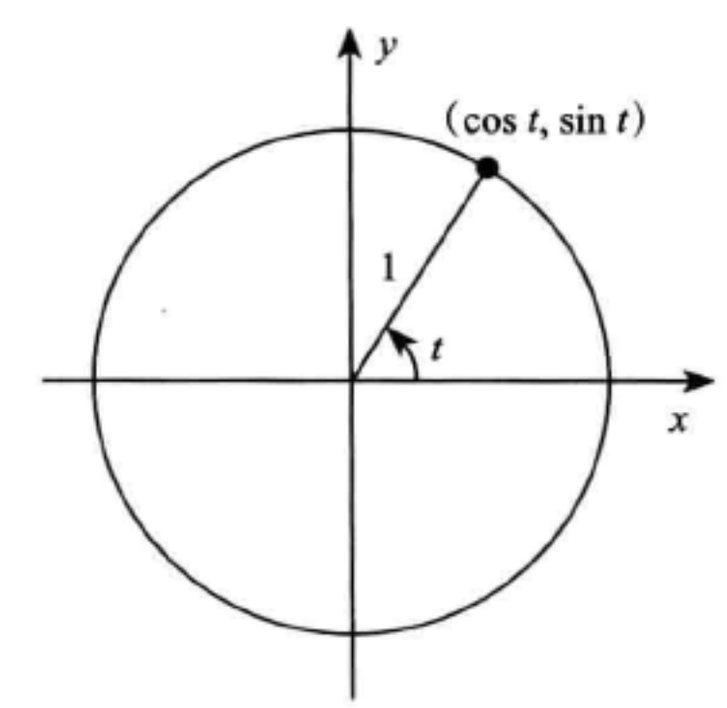


图 8-1 单位圆

8.3.3 为三角形扇添加一个绘画命令

我们还需要告诉 OpenGL 如何绘制这个冰球的顶部。因为一个冰球是由两个基本图元构造的，一个构造顶部的三角形扇，另一个构造侧面的三角形带，所以我们需要一种方法，它可以把这些绘画命令合并在一起，以便在后面只需要调用 puck.draw() 即可。能实现这些的一个方法是把每一个绘画命令都加入一个绘画命令列表（list）中。

我们要创建一个接口（interface）表示单个绘画命令。在 ObjectBuilder 顶部加入如下代码：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
static interface DrawCommand {
    void draw();
}
```

我们也需要一个实例变量用于保存那些整理好的绘画命令。

在 vertexData 后面加入如下定义：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
private final List<DrawCommand> drawList = new ArrayList<DrawCommand>();
```

我们现在可以为三角形扇添加一个绘画命令了。在 appendCircle() 顶部修改代码，加入如下变量定义：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
final int startVertex = offset / FLOATS_PER_VERTEX;
final int numVertices = sizeOfCircleInVertices(numPoints);
```

因为我们只使用一个数组保存所有物体的数据，我们需要告诉 OpenGL 每个绘画命令对应的正确的顶点偏移值。计算偏移值和长度，并把它们存储在 startVertex 和 numVertices。现在可以在 appendCircle() 尾部加入如下代码：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
drawList.add(new DrawCommand() {
    @Override
    public void draw() {
        glDrawArrays(GL_TRIANGLE_FAN, startVertex, numVertices);
    }
});
```

通过这些代码，我们就创建了一个内部类，它调用 glDrawArrays()，并且我们还把这个内部类添加进了绘画命令列表。后面要绘制这个冰球时，我们只需要执行列表中的每个 draw() 方法。

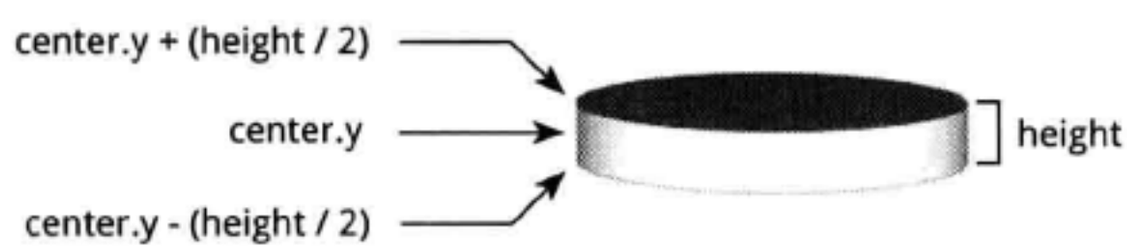
8.3.4 用三角形带构造圆柱体侧面

下一步是用一个三角形带构造这个冰球的侧面。首先，我们在 appendCircle() 后面加入如下代码：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
private void appendOpenCylinder(Cylinder cylinder, int numPoints) {
    final int startVertex = offset / FLOATS_PER_VERTEX;
    final int numVertices = sizeOfOpenCylinderInVertices(numPoints);
    final float yStart = cylinder.center.y - (cylinder.height / 2f);
    final float yEnd = cylinder.center.y + (cylinder.height / 2f);
```

正如以前一样，我们要计算出起始顶点和顶点数量，以便于在绘画命令中使用它们。我们也要搞清楚这个冰球从哪里开始到哪里结束，这些位置应该如下图所示：



加入如下代码生成实际的三角形带：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
for (int i = 0; i <= numPoints; i++) {
    float angleInRadians =
          ((float) i / (float) numPoints)
        * ((float) Math.PI * 2f);

    float xPosition =
          cylinder.center.x
        + cylinder.radius * FloatMath.cos(angleInRadians);

    float zPosition =
          cylinder.center.z
        + cylinder.radius * FloatMath.sin(angleInRadians);

    vertexData[offset++] = xPosition;
    vertexData[offset++] = yStart;
    vertexData[offset++] = zPosition;
```

```
        vertexData[offset++] = xPosition;
        vertexData[offset++] = yEnd;
        vertexData[offset++] = zPosition;
    }
```

我们使用了同前面生成圆周顶点一样的算法，只是这次我们为圆周上的每个点生成了两个顶点，一个是圆柱顶部，另一个是圆柱底部。前面两个点的位置重复两次以使这个圆柱体闭合。

加入如下代码完成这个方法：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
    drawList.add(new DrawCommand() {
        @Override
        public void draw() {
            glDrawArrays(GL_TRIANGLE_STRIP, startVertex, numVertices);
        }
    });
}
```

我们使用 GL_TRIANGLE_STRIP 告诉 OpenGL 绘制一个三角形带。

8.3.5 返回生成的数据

要使 createPuck() 工作，我们只需要定义 build() 方法。我们要使用它返回 GeneratedData 对象内部生成的数据。我们还没有定义这个类，因此，在 ObjectBuilder 的顶部，就在 DrawCommand 之后加入如下类：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
static class GeneratedData {
    final float[] vertexData;
    final List<DrawCommand> drawList;

    GeneratedData(float[] vertexData, List<DrawCommand> drawList) {
        this.vertexData = vertexData;
        this.drawList = drawList;
    }
}
```

这只是一个 holder 类，以便我们可以用单个对象返回顶点数据和绘画命令列表。现在我们只需要定义 build()：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
private GeneratedData build() {
    return new GeneratedData(vertexData, drawList);
}
```

要 createPuck() 正常工作，这些就是我们所需的全部。让我们花费片刻复习一下这个流程：

- 首先，我们从这个类的外部调用静态方法 createPuck()。这个方法创建一个新的 ObjectBuilder 对象，它用合适大小的数组保存这个冰球的所有数据。它也创建了一个显示列表，以便后续绘制这个冰球。

- 在 createPuck() 内部，我们调用 appendCircle() 和 appendOpenCylinder() 分别生成冰球的顶部和侧面。每个方法都给 vertexData 添加了它的数据，并给 drawList 添加了一个绘画命令。
- 最后，我们调用 build() 返回生成的数据。

8.3.6　用两个圆柱体构造一个木槌

现在可以用我们学过的内容构造一个木槌了。一个木槌可以由两个圆柱体构造，因此构造一个木槌几乎与构造两个不同大小的冰球一样。我们将以某种方式定义木槌，如图 8-2 所示。

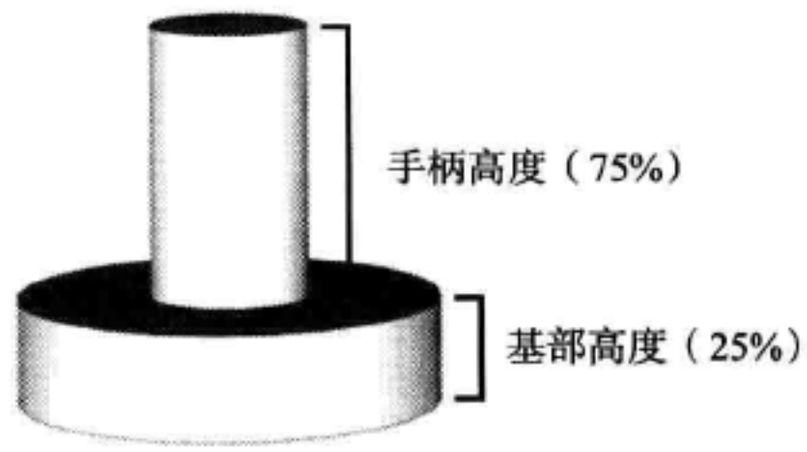


图 8-2　木槌定义

手柄的高度占整体高度的 75%，而基部的高度占整体高度 25%。我们也可以看出手柄的宽度占整体宽度的三分之一。有了这些定义，我们就能计算在哪里放置组成木槌的这两个圆柱体了。

当把这些定义写出来时，拿出一张纸并把物体画出来，然后以图表形式画出这些物体相对中心的位置以及它们的侧面，这也是有帮助的。要创建木槌，我们需要计算出每个圆柱体顶部的 y 的位置以及每个圆柱体中心的位置。

在 createPuck() 之后加入名为“createMallet”的方法。我们以下面的代码开始：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
static GeneratedData createMallet(
    Point center, float radius, float height, int numPoints) {
    int size = sizeOfCircleInVertices(numPoints) * 2
            + sizeOfOpenCylinderInVertices(numPoints) * 2;

    ObjectBuilder builder = new ObjectBuilder(size);

    // First, generate the mallet base.
    float baseHeight = height * 0.25f;

    Circle baseCircle = new Circle(
        center.translateY(-baseHeight),
        radius);
    Cylinder baseCylinder = new Cylinder(
        baseCircle.center.translateY(-baseHeight / 2f),
        radius, baseHeight);

    builder.appendCircle(baseCircle, numPoints);
    builder.appendOpenCylinder(baseCylinder, numPoints);
```

我们创建了一个有正确大小的新 ObjectBuilder 对象，然后生成木槌的基部。这段代码与我们在 createPuck() 中所写的十分相似。

当你优化导入（import）时，Eclipse 可能会让你做出选择，一个是我们自己定义的 Point 类“com.airhockey.android.util.Geometry.Point”，或者 Android SDK 中自带的一个不同的 Point 类“android.Graphics.Point”。你要确保选择了我们自己定义的 Point 类。

让我们加入下面的代码生成手柄：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
float handleHeight = height * 0.75f;
float handleRadius = radius / 3f;

Circle handleCircle = new Circle(
    center.translateY(height * 0.5f),
    handleRadius);
Cylinder handleCylinder = new Cylinder(
    handleCircle.center.translateY(-handleHeight / 2f),
    handleRadius, handleHeight);

builder.appendCircle(handleCircle, numPoints);
builder.appendOpenCylinder(handleCylinder, numPoints);
```

我们遵循与以前一样的步骤，但是使用了不同的坐标和不同的大小。让我们加入对 build() 的调用来完成这个方法：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/ObjectBuilder.java

```
    return builder.build();
}
```

这就是 ObjectBuilder 类的全部了！我们现在能生成冰球和木槌了；当我们要绘制它们时，只需要把顶点数据绑定到 OpenGL，并调用 object.draw()。

8.4 更新物体

既然我们有了一个物体构建器，就不用再把木槌画成点了，需要更新 Mallet 类。我们也需要添加一个新的 Puck 类。让我们以冰球开始。在同一个包中创建一个叫做“Puck”的新类，并给此类加入如下代码：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/Puck.java

```
private static final int POSITION_COMPONENT_COUNT = 3;

public final float radius, height;

private final VertexArray vertexArray;
private final List<DrawCommand> drawList;

public Puck(float radius, float height, int numPointsAroundPuck) {
    GeneratedData generatedData = ObjectBuilder.createPuck(new Cylinder(
        new Point(0f, 0f, 0f), radius, height), numPointsAroundPuck);
    this.radius = radius;
    this.height = height;

    vertexArray = new VertexArray(generatedData.vertexData);
    drawList = generatedData.drawList;
}
```

当一个新冰球被创建的时候，它会生成那个物体的数据，用 vertexArray 把顶点存储在一个本地缓冲区中，并把绘画命令列表存储在 drawList 中。

让我们完成这个类：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/Puck.java

```
public void bindData(ColorShaderProgram colorProgram) {
    vertexArray.setVertexAttribPointer(0,
        colorProgram.getPositionAttributeLocation(),
        POSITION_COMPONENT_COUNT, 0);
}
public void draw() {
    for (DrawCommand drawCommand : drawList) {
        drawCommand.draw();
    }
}
```

第一个方法 bindData() 遵循的模式与 Table 和 Mallet 类所遵循的是一样的：它把顶点数据绑定到着色器程序定义的属性上。第二个方法 onDraw() 只是遍历 ObjectBuilder.createPuck() 创建的显示列表。

我们也需要更新 Mallet 类，用下面的代码替换掉该类中的所有代码：

AirHockeyWithImprovedMallets/src/com/airhockey/android/objects/Mallet.java

```
private static final int POSITION_COMPONENT_COUNT = 3;

public final float radius;
public final float height;

private final VertexArray vertexArray;
private final List<DrawCommand> drawList;

public Mallet(float radius, float height, int numPointsAroundMallet) {
    GeneratedData generatedData = ObjectBuilder.createMallet(new Point(0f,
        0f, 0f), radius, height, numPointsAroundMallet);

    this.radius = radius;
    this.height = height;

    vertexArray = new VertexArray(generatedData.vertexData);
    drawList = generatedData.drawList;
}
public void bindData(ColorShaderProgram colorProgram) {
    vertexArray.setVertexAttribPointer(0,
        colorProgram.getPositionAttributeLocation(),
        POSITION_COMPONENT_COUNT, 0);
}
public void draw() {
    for (DrawCommand drawCommand : drawList) {
        drawCommand.draw();
    }
}
```

这些代码遵循与 Puck 类一样的模式。

8.5 更新着色器

我们还需要更新颜色着色器。我们用每个顶点的位置而不是每个顶点的颜色定义了冰

球和木槌，因此我们不得不把颜色作为一个 uniform 传递进去。所有的改动里第一件要做的就是给 ShaderProgram 加入一个新的常量：

AirHockeyWithImprovedMallets/src/com/airhockey/android/programs/ShaderProgram.java

```
protected static final String U_COLOR = "u_Color";
```

下一步是更新 ColorShaderProgram。继续移除所有 aColorLocation 的引用，包括 getColorAttributeLocation()，然后加入如下 uniform 位置的定义：

AirHockeyWithImprovedMallets/src/com/airhockey/android/programs/ColorShaderProgram.java

```
private final int uColorLocation;
```

更新构造函数设置 uniform 的位置：

```
uColorLocation = glGetUniformLocation(program, U_COLOR);
```

要完成这些改动，setUniforms() 更新如下：

```
public void setUniforms(float[] matrix, float r, float g, float b) {
    glUniformMatrix4fv(uMatrixLocation, 1, false, matrix, 0);
    glUniform4f(uColorLocation, r, g, b, 1f);
}
```

我们还需要更新实际的着色器。更新 simple_vertex_shader.glsl 的内容如下代码所示：

AirHockeyWithImprovedMallets/res/raw/simple_vertex_shader.glsl

```
uniform mat4 u_Matrix;
attribute vec4 a_Position;
void main()
{
    gl_Position = u_Matrix * a_Position;
}
```

更新 simple_fragment_shader.glsl 如下代码所示：

AirHockeyWithImprovedMallets/res/raw/simple_fragment_shader.glsl

```
precision mediump float;

uniform vec4 u_Color;

void main()
{
    gl_FragColor = u_Color;
}
```

我们的着色器现在就是最新的了。

8.6 集成所有的变化

本章最困难的部分已经完成了。我们学习了如何用简单的几何形状构造冰球和木槌，并更新了着色器用以反映这些变化。所有剩下的就是把这些变化集成到 AirHockeyRenderer；同时，我们还要学习通过加入视图矩阵添加一个相机的概念。

那么我们为什么想添加另一个矩阵呢？当我们最早开始空气曲棍球项目的时候，根本没使用任何矩阵。我们首先添加了一个正交矩阵调整宽高比，接着切换到一个透视矩阵获得一个三维投影。之后添加了一个模型矩阵开始来回移动物体。视图矩阵实际只是模型矩阵的扩展；它的使用是出于同样的目的，只不过它平等地应用于场景中的每一个物体。

8.6.1　简单的矩阵层次结构

让我们花点儿时间复习一下，把一个物体放到屏幕上的三种主要的矩阵类型。

- 模型矩阵

  模型矩阵是用来把物体放在世界空间（world-space）坐标系的。比如，我们可能有冰球模型和木槌模型，它们初始的中心点都在（0，0，0）。没有模型矩阵，这些模型就会卡在那里：如果我们想要移动它们，就不得不自己更新每个模型的每个顶点。如果不想这样做，我们可以使用一个模型矩阵，把那些顶点与这个矩阵相乘来变换它们。如果我们想把冰球移动到（5，5），我们只需要准备一个模型矩阵，它会为我们完成的。

- 视图矩阵

  视图矩阵是出于同模型矩阵一样的原因被使用的，但是它平等地影响场景中的每一个物体。因为它影响所有的东西，它在功能上等同于一个相机：来回移动相机，你会从不同的视角看见那些东西。

  使用另外一个矩阵的优势是它让我们预先把许多变换处理成单个矩阵。举个例子，想象一下我们要来回旋转一个场景，并把它移动一定量的距离。能实现这些的一种方式是把同样的旋转和平移调用应用于每一个单个的物体。尽管那样可行，但如果只把这些变换存到另外一个矩阵，并把这个矩阵应用于每个物体，会更容易实现。

- 投影矩阵

  最后，说说投影矩阵。这个矩阵帮助创建三维的幻象，通常只有当屏幕变换方位时，它才会变化。

  让我们也复习一下一个顶点如何从它原来的位置变换到屏幕的。

- $vertex_{model}$

  这是模型坐标系中的一个顶点。例如，被包含在桌子顶点内部的位置。

- $vertex_{world}$

  这是在世界空间中用模型矩阵定位过的一个顶点。

- $vertex_{eye}$

  这是与我们的眼睛或相机相对的一个顶点。我们使用一个视图矩阵让所有的顶点在世界空间中绕着我们当前的观察位置移动。

- $vertex_{clip}$

  这是被投影矩阵处理过的一个顶点。下一步就是做透视除法，正如我们在 6.2.2 节中

解释的一样。

- $vertex_{ndc}$

  这是归一化设备坐标系中的一个顶点。一旦一个顶点落在这个坐标中，OpenGL 就会把它映射到视口，你就能从屏幕上看到它了。

  这个链条看上去如下所示：

  $vertex_{clip}$ = ProjectionMatrix * $vertex_{eye}$

  $vertex_{clip}$ = ProjectionMatrix * ViewMatrix * $vertex_{world}$

  $vertex_{clip}$ = ProjectionMatrix * ViewMatrix * ModelMatrix * $vertex_{model}$

  要得到正确的结果，我们需要应用本章中的每个矩阵。

8.6.2 给空气曲棍球桌子添加新的物体

让我们继续给 AirHockeyRender 添加一个视图矩阵，同时，我们还要处理新的木槌和冰球。我们将首先在类的顶部添加一些新的矩阵定义：

AirHockeyWithImprovedMallets/src/com/airhockey/android/AirHockeyRenderer.java

```
private final float[] viewMatrix = new float[16];
private final float[] viewProjectionMatrix = new float[16];
private final float[] modelViewProjectionMatrix = new float[16];
```

我们把视图矩阵存储到 viewMatrix，其他两个矩阵用于保存矩阵乘积的结果。我们也在 Mallet 后面给这个新的 puck 添加一个定义：

AirHockeyWithImprovedMallets/src/com/airhockey/android/AirHockeyRenderer.java

```
private Puck puck;
```

不要忘记更新导入（import）。下一步是初始化 Mallet 和 Puck 对象。我们将在 onSurface-Created() 中用一个指定的大小创建它们，如下代码所示：

AirHockeyWithImprovedMallets/src/com/airhockey/android/AirHockeyRenderer.java

```
mallet = new Mallet(0.08f, 0.15f, 32);
puck = new Puck(0.06f, 0.02f, 32);
```

冰球与木槌的半径和高度被设置为一个固定的大小，以使它们看上去与桌子成比例。每个物体都是用围着圆的 32 个点创建的。

8.6.3 初始化新矩阵

下一步是更新 onSurfaceChanged()，并初始化视图矩阵。更新 onSurfaceChanged() 匹配如下内容：

AirHockeyWithImprovedMallets/src/com/airhockey/android/AirHockeyRenderer.java

```
@Override
public void onSurfaceChanged(GL10 glUnused, int width, int height) {
    // Set the OpenGL viewport to fill the entire surface.
    glViewport(0, 0, width, height);
    MatrixHelper.perspectiveM(projectionMatrix, 45, (float) width
```

```
        / (float) height, 1f, 10f);
    setLookAtM(viewMatrix, 0, 0f, 1.2f, 2.2f, 0f, 0f, 0f, 0f, 1f, 0f);
}
```

这个方法的第一部分相当标准：设置视口，并建立投影矩阵。下一部分是新内容：调用 setLookAtM()（参数见表 8-1）创建一个特殊类型的视图矩阵。

setLookAtM(float[] rm, int rmOffset, float eyeX, float eyeY, float eyeZ, float centerX, float centerY, float centerZ, float upX, float upY, float upZ)

表 8-1　setLookAtM() 参数

float[] rm	这是目标数组。这个矩阵的长度应该至少容纳 16 个元素，以便它能存储视图矩阵
int rmOffset	setLookAtM() 会把结果从 rm 的这个偏移值开始存进 rm
float eyeX, eyeY, eyeZ	这是眼睛所在的位置。场景中的所有东西看起来都像是从这个点观察它们一样
float centerX, centerY, centerZ	这是眼睛正在看的地方；这个位置出现在整个场景的中心
float upX, upY, upZ	要是我们刚才正在讨论你的眼睛，那么这是你的头指向的地方。upY 的值为 1 意味着你的头笔直指向上方

我们调用 setLookAtM() 时，把眼睛（eye）设为（0，1.2，2.2），这意味着眼睛的位置在 x-z 平面上方 1.2 个单位，并向后 2.2 个单位。换句话说，场景中的所有东西都出现在你下面 1.2 个单位和你前面 2.2 个单位的地方。把中心（center）设为（0，0，0），意味着你将向下看你前面的原点，并把指向（up）设为（0，1，0），意味着你的头是笔直指向上面的，这个场景不会旋转到任何一边。

8.6.4　更新 onDrawFrame

在运行程序之前，只剩下最后几个改动了，看看这些新的改动。在 glClear() 调用后面把下面的代码加入 onDrawFrame()：

AirHockeyWithImprovedMallets/src/com/airhockey/android/AirHockeyRenderer.java

```
multiplyMM(viewProjectionMatrix, 0, projectionMatrix, 0, viewMatrix, 0);
```

这会把投影和视图矩阵乘在一起的结果缓存到 viewProjectionMatrix。用如下代码替换 onDrawFrame() 的余下部分：

AirHockeyWithImprovedMallets/src/com/airhockey/android/AirHockeyRenderer.java

```
positionTableInScene();
textureProgram.useProgram();
textureProgram.setUniforms(modelViewProjectionMatrix, texture);
table.bindData(textureProgram);
table.draw();

// Draw the mallets.
positionObjectInScene(0f, mallet.height / 2f, -0.4f);
colorProgram.useProgram();
colorProgram.setUniforms(modelViewProjectionMatrix, 1f, 0f, 0f);
mallet.bindData(colorProgram);
```

```
mallet.draw();

positionObjectInScene(0f, mallet.height / 2f, 0.4f);
colorProgram.setUniforms(modelViewProjectionMatrix, 0f, 0f, 1f);
// Note that we don't have to define the object data twice -- we just
// draw the same mallet again but in a different position and with a
// different color.
mallet.draw();

// Draw the puck.
positionObjectInScene(0f, puck.height / 2f, 0f);
colorProgram.setUniforms(modelViewProjectionMatrix, 0.8f, 0.8f, 1f);
puck.bindData(colorProgram);
puck.draw();
```

这段代码的大部分与上一章相同，但是有一些关键的区别。第一个不同之处是我们在绘制那些物体之前调用了 positionTableInScene() 和 positionObjectInScene()。我们在绘制木槌之前还更新了 setUniforms()，并且添加了绘制冰球的代码。

你还注意到了么？我们用同一个木槌数据绘制了两个木槌。如果需要，我们可以用同一个顶点集合绘制成百上千个物体：我们需要做的就是在绘制每个物体之前更新模型矩阵。

让我们添加 positionTableInScene() 的定义：

AirHockeyWithImprovedMallets/src/com/airhockey/android/AirHockeyRenderer.java

```
private void positionTableInScene() {
    // The table is defined in terms of X & Y coordinates, so we rotate it
    // 90 degrees to lie flat on the XZ plane.
    setIdentityM(modelMatrix, 0);
    rotateM(modelMatrix, 0, -90f, 1f, 0f, 0f);
    multiplyMM(modelViewProjectionMatrix, 0, viewProjectionMatrix,
        0, modelMatrix, 0);
}
```

这个桌子原来是以 x 和 y 坐标定义的，因此要使它平放在地上，我们需要让它绕 x 轴向后旋转 90 度。注意，不像前面的课程，我们不需要把桌子平移一定的距离，因为我们想要桌子在世界坐标里保持在位置（0，0，0），并且视图矩阵已经想办法使桌子对我们可见了。

最后一步是通过把 viewProjectionMatrix 和 modelMatrix 相乘将所有矩阵合并在一起，并把结果存储到 modelViewProjectionMatrix，它接着会被传递给着色器程序。

让我们把 positionObjectInScene() 的定义也添加进来：

AirHockeyWithImprovedMallets/src/com/airhockey/android/AirHockeyRenderer.java

```
private void positionObjectInScene(float x, float y, float z) {
    setIdentityM(modelMatrix, 0);
    translateM(modelMatrix, 0, x, y, z);
    multiplyMM(modelViewProjectionMatrix, 0, viewProjectionMatrix,
        0, modelMatrix, 0);
}
```

木槌和冰球已经被定义好，并被水平放在 x-z 平面上了，因此不需要旋转它们。我们要根据传递进来的参数平移它们，将它们放在桌子上方正确的位置。

运行这个程序。如果一切按计划顺利进行，那它看起来应该如图 8-3 所示。新的木槌和

冰球看起来应该是放在桌子中间的。木槌也许显得太坚实；我们会在第 13 章中学习如何改进这点。

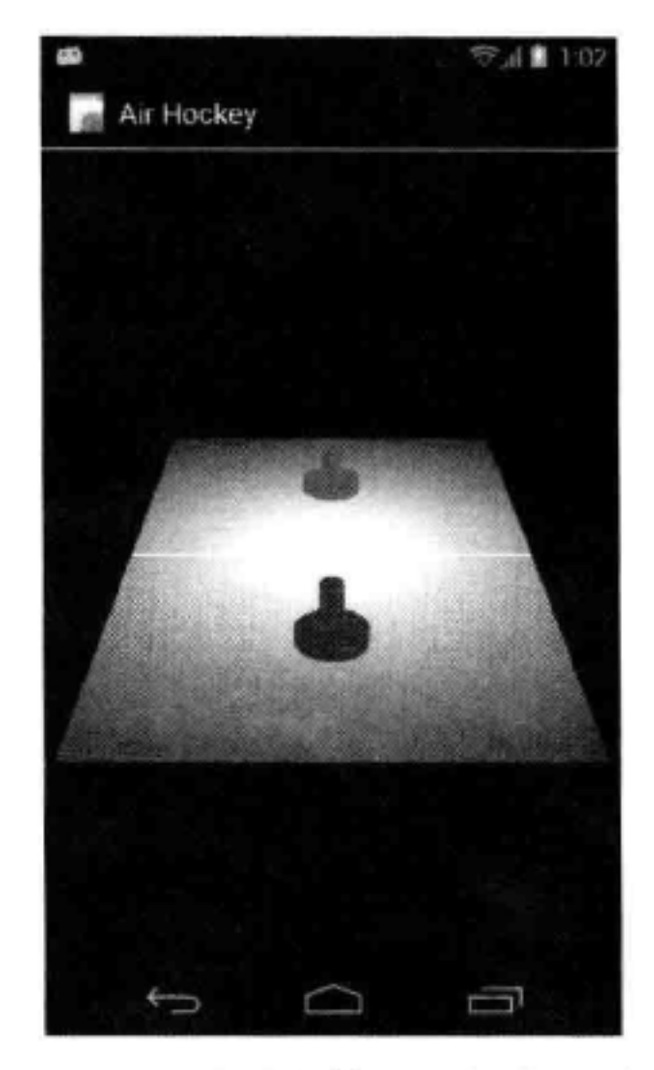


图 8-3　改进木槌的空气曲棍球

8.7　小结

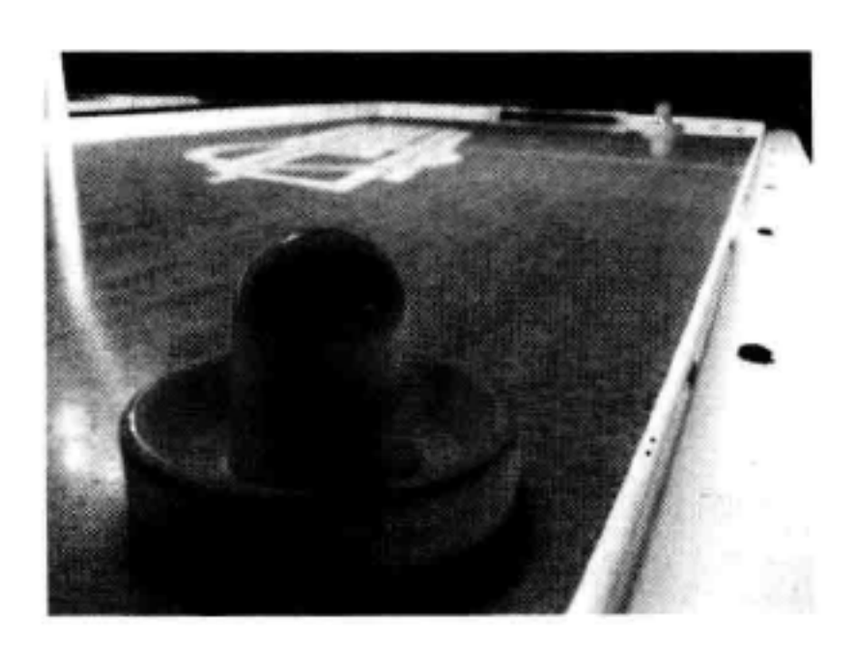
恭喜你又完成了一个紧凑的章节！我们学习了所有关于生成三角形带和三角形扇的内容，以及如何把它们放在一起形成物体。当构造这些物体的时候，我们也学习了如何封装那些绘画调用，以便我们可以容易地把它们一起绑在一个单个命令里。

我们也介绍了矩阵层次结构的概念，一个矩阵用于投影，一个用于相机，一个用于在世界空间移动物体。以这种方式把事物分开，使操作场景和移动物体变得更加容易。

8.8　练习

作为你的第一个练习，通过给 onDrawFrame() 添加一个方法调用绕着桌子慢慢旋转你的视点（viewpoint）。

一个更具挑战性的练习，看一下下面的图：



你要怎样更新木槌生成器呢？使它看上去更接近这种类型的木槌。你还可以使用简单的图形构造这个木槌。

这也许是你想到的解决问题的方法。

- 两个常规的圆柱体侧面：一个给手柄，一个给基部的外侧面；
- 两个圆环：一个用于基部的顶部，一个用于基部的内部；
- 一个斜的圆柱体侧面用于连接基部内部的两个圆环；
- 一个半球盖在木槌的上面。

当然，想象力是无限的，你可以用任何你想到的创意。当你准备好了，我们将在下一章学习如何用手指来回移动木槌。

Chapter 9 第 9 章

增加触控反馈：与空气曲棍球游戏交互

良好的、有触控支持的用户交互是许多游戏和应用的基石；它可以给用户一种与真实事物交互的感觉，即使他们只是在盯着屏幕上的那些像素。有些移动游戏已经非常流行，只是因为它们使用了一种新的触控模式；大量的游戏一直在使用这个技术。

这个空气曲棍球游戏现在已经有了长相更好看的木槌，但是，如果我们能实际使用它们，不是很好吗？在本章，我们就会通过添加触控支持使这个程序更具交互性。我们要学习如何使用三维相交测试和碰撞检测，以便我们可以抓住木槌并在屏幕上来回拖动它。

以下是本章的开发计划：

- 我们以空气曲棍球项目添加触控交互作为开始。我们会复习一些数学知识及其演算过程，它们是使交互可以工作所需要的知识。
- 接下来我们会学习如何让木槌和冰球交互，并停留在边界内。

当我们完成本章的时候，我们就能用木槌击打冰球，并观看它在桌子上来回反弹了！作为开始，让我们把前一章的项目另存为一个新的项目，并命名为“AirHockeyTouch”。

9.1 为 Activity 增加触控支持

我们从介绍 Android 触控事件系统开始。我们要做的第一件事是要保存渲染器的一个引用，因此，打开 AirHockeyActivity，并按如下代码修改 setRenderer() 调用：

AirHockeyTouch/src/com/airhockey/android/AirHockeyActivity.java

```
final AirHockeyRenderer airHockeyRenderer = new AirHockeyRenderer(this);

if (supportsEs2) {
    // ...
    glSurfaceView.setRenderer(airHockeyRenderer);
```

我们会在触控处理程序（handler）中使用这个引用通知渲染器新的触控事件。

9.1.1　监听触控事件

让我们开始写触控处理程序吧，在调用 setContentView() 之前加入如下代码：

AirHockeyTouch/src/com/airhockey/android/AirHockeyActivity.java

```
glSurfaceView.setOnTouchListener(new OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        if (event != null) {
            // Convert touch coordinates into normalized device
            // coordinates, keeping in mind that Android's Y
            // coordinates are inverted.
            final float normalizedX =
                (event.getX() / (float) v.getWidth()) * 2 - 1;
            final float normalizedY =
                -((event.getY() / (float) v.getHeight()) * 2 - 1);
```

在 Android 里，我们可以通过调用 setOnTouchListener() 监听视图（view）的触控事件。当用户触碰了那个视图，我们就会收到一个调用 onTouch()。

我们需要做的第一件事是检查是否有事件要处理。在 Android 里，触控事件是发生在视图的坐标空间中的，视图的左上角映射到（0，0），右下角映射到的坐标等于视图的大小。举个例子，如果视图是 480 个像素宽、800 个像素高，那右下角就映射到（480，800）。

我们在着色器中需要使用归一化设备坐标（参考 5.1 节），因此我们需要把触控事件坐标转换回归一化设备坐标，这需要把 y 轴反转，并把每个坐标按比例映射到范围 [−1，1] 内。

9.1.2　转发触控事件给渲染器

让我们完成以下触控处理程序：

AirHockeyTouch/src/com/airhockey/android/AirHockeyActivity.java

```
            if (event.getAction() == MotionEvent.ACTION_DOWN) {
                glSurfaceView.queueEvent(new Runnable() {
                    @Override
                    public void run() {
                        airHockeyRenderer.handleTouchPress(
                            normalizedX, normalizedY);
                    }
                });
            } else if (event.getAction() == MotionEvent.ACTION_MOVE) {
                glSurfaceView.queueEvent(new Runnable() {
                    @Override
                    public void run() {
                        airHockeyRenderer.handleTouchDrag(
                            normalizedX, normalizedY);
                    }
                });
            }
```

```
                return true;
            } else {
                return false;
            }
        }
    });
```

检查一下这个事件是按压（press）还是拖拽（drag）事件，因为我们需要按不同方式处理每种情况。按压对应 MotionEvent.ACTION_DOWN，而拖拽对应 MotionEvent.ACTION_MOVE。

重要的是，要记住，Android 的 UI 运行在主线程，而 OpenGL 的 GLSurfaceView 运行在一个单独的线程中，因此我们需要使用线程安全的技术在两个线程间通信。我们使用 queueEvent() 给 OpenGL 线程分发调用，对于按压事件，调用 airHockeyRenderer.handleTouchPress()，对于拖拽事件，调用 airHockeyRenderer.handleTouchDrag()。这些方法此刻并不存在，因此我们不久就将创建它们。

在完成这个处理程序时，返回“true”告诉 Android 我们已经处理了这个触控事件。如果事件是“null”，那就返回“false”。

打开 AirHockeyRenderer，并为 handleTouchPress() 和 handleTouchDrag() 加入桩代码：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
public void handleTouchPress(float normalizedX, float normalizedY) {

}

public void handleTouchDrag(float normalizedX, float normalizedY) {

}
```

作为一个小练习，在这里添加一些日志记录，运行程序，当你触碰屏幕时，看看发生了什么。

9.2 增加相交测试

我们已经在归一化设备坐标里得到了屏幕的被触碰的区域，就需要决定这个被触碰的区域是否涵盖木槌。我们需要执行相交测试，这是开发三维游戏和应用时一个非常重要的操作。以下是我们需要做的：

1. 首先，我们要把二维屏幕坐标转换到三维空间，并看看我们触碰到了什么。要做到这点，我们要把被触碰的点投射到一条射线上，这条射线从我们的视点跨越那个三维场景。
2. 然后，我们需要检查看看这条射线是否与木槌相交。为了使事情简单些，我们假定那个木槌实际上是一个差不多同样大小的包围球，然后测试那个球。

让我们以创建两个新的成员变量作为开始：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
private boolean malletPressed = false;
private Point blueMalletPosition;
```

我们要用 malletPressed 跟踪木槌当前是否被按到了。我们也把木槌的位置存储在 blueMalletPosition，它需要被初始化为一个默认值，因此给 onSurfaceCreated () 添加如下代码：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
blueMalletPosition = new Point(0f, mallet.height / 2f, 0.4f);
```

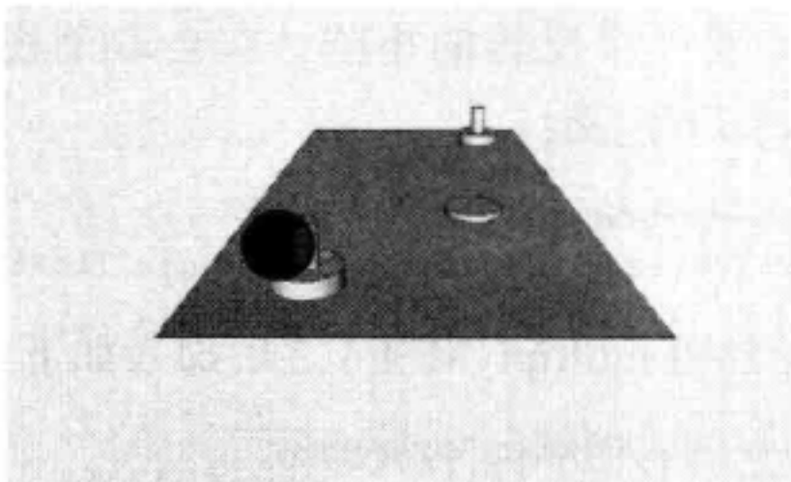
接下来，按如下代码更新 handleTouchPress()：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
public void handleTouchPress(float normalizedX, float normalizedY) {
    Ray ray = convertNormalized2DPointToRay(normalizedX, normalizedY);

    // Now test if this ray intersects with the mallet by creating a
    // bounding sphere that wraps the mallet.
    Sphere malletBoundingSphere = new Sphere(new Point(
            blueMalletPosition.x,
            blueMalletPosition.y,
            blueMalletPosition.z),
        mallet.height / 2f);

    // If the ray intersects (if the user touched a part of the screen that
    // intersects the mallet's bounding sphere), then set malletPressed =
    // true.
    malletPressed = Geometry.intersects(malletBoundingSphere, ray);
}
```

要计算被触碰的点是否与木槌相交，我们首先要把被触控的点投射到一条射线上，用一个包围球封装木槌，然后测试一下看看那条射线是否与球相交。

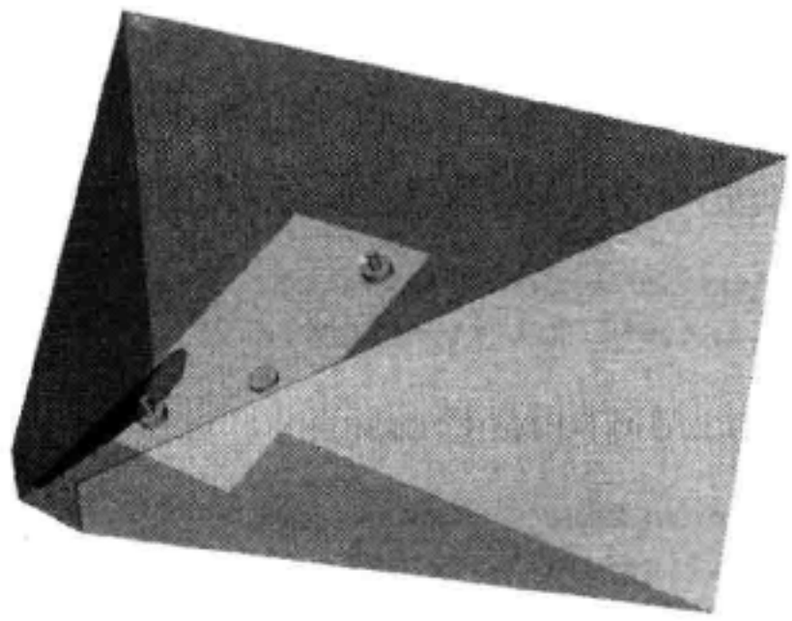
如果我们从视觉上看看这些东西，可能更加直观。让我们想象一个空气曲棍球的场景，有桌子、冰球和两个木槌，并想象在下图黑圈的位置触摸屏幕：



我们清楚地触碰了其中一个木槌。然而，被触碰的区域是在二维空间中，而木槌是在三维空间中。我们怎么测试被触碰的点是否与那个木槌相交呢？

要测试这个，我们首先要把那个二维的点转换为两个三维的点：一个在三维视椎体的近端，另一个在三维视椎体的远端（如果“视椎体”（frustum）这个单词让你一头雾水，现在也是时候回到 6.4.1 节，花些时间复习一下）。然后，我们在这两个点之间画一条直线来创建一条射线。如果我们从侧面看一下这个场景，下图示意了这条射线怎样与那个三维场

景相交的：



为了使数学计算容易些，当我们做测试的时候，假定这个木槌是一个球。

首先，定义 convertNormalized2DPointToRay()，解决这个谜题的第一部分：把被触碰的点转换为一个三维射线。

9.2.1 二维点扩展成三维直线

通常，当我们把一个三维场景投递到二维屏幕的时候，我们使用透视投影和透视除法把顶点坐标变换为归一化设备坐标（参考 6.2.2 节）。

现在我们想按相反方向变换：我们有被触碰点的归一化设备坐标，我们要计算出在三维世界里那个被触碰的点与哪里相对应。为了把被触碰的点转换为一个三维射线，实质上我们需要取消透视投影和透视除法。

我们当前有被触碰点的 x 和 y 坐标，但我们还不知道它应该在多远或多近的地方。要解决这个模糊性，我们把被触碰的点映射到三维空间的一条直线：直线的近端映射到我们在投影矩阵中定义的视椎体的近平面，直线的远端映射到视椎体的远平面。

要实现这个转换，我们需要一个反转的矩阵，它会取消视图矩阵和投影矩阵的效果。让我们给矩阵定义的列表添加如下定义：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
private final float[] invertedViewProjectionMatrix = new float[16];
```

在 onDrawFrame() 里，在调用 multiplyMM() 之后加入如下一行代码：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
invertM(invertedViewProjectionMatrix, 0, viewProjectionMatrix, 0);
```

这个调用会创建一个反转的矩阵，我们可以用它把那个二维被触碰的点转换为两个三维坐标。如果场景可以移来移去，它就会影响场景的哪一部分在手指下面，因此我们也把视图矩阵考虑在内。我们通过反转合并后的视图和投影矩阵实现这些。

反转透视投影和透视除法

我们现在可以开始定义 convertNormalized2DPointToRay()。让我们以下面的代码开始：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
private Ray convertNormalized2DPointToRay(
    float normalizedX, float normalizedY) {
    // We'll convert these normalized device coordinates into world-space
    // coordinates. We'll pick a point on the near and far planes, and draw a
    // line between them. To do this transform, we need to first multiply by
    // the inverse matrix, and then we need to undo the perspective divide.
    final float[] nearPointNdc = {normalizedX, normalizedY, -1, 1};
    final float[] farPointNdc =  {normalizedX, normalizedY,  1, 1};

    final float[] nearPointWorld = new float[4];
    final float[] farPointWorld = new float[4];

    multiplyMV(
        nearPointWorld, 0, invertedViewProjectionMatrix, 0, nearPointNdc, 0);
    multiplyMV(
        farPointWorld, 0, invertedViewProjectionMatrix, 0, farPointNdc, 0);
```

为了把被触碰的点映射到一条射线，我们在归一化设备坐标里设置了两个点：其中一个点是 z 值为 -1 的点，而另一个点是 z 值为 +1 的点。我们分别把这两个点存储在 nearPointNdc 和 farPointNdc。因为我们不知道 w 分量应该被设为多大，暂且把它们的 w 分量设为 1。接下来，我们把每个点都与 invertedViewProjectionMatrix 相乘，得到世界空间中的坐标。

我们也需要撤销透视除法的影响。反转的视图投影矩阵有一个有趣的属性：把顶点和反转的视图投影矩阵相乘以后，nearPointWorld 和 farPointWorld 实际上就含有了反转的 w 值。这是因为，通常情况下，投影矩阵的主要意义就是创建不同的 w 值，以便透视除法可以施加它的魔法；因此，如果我们使用一个反转的投影矩阵，我们就会得到一个反转的 w。我们所需要做的就是把 x、y 和 z 除以这些反转的 w，这样就撤销了透视除法的影响。

让我们继续定义 convertNormalized2DPointToRay()：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
divideByW(nearPointWorld);
divideByW(farPointWorld);
```

我们也需要定义 divideByW()：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
private void divideByW(float[] vector) {
    vector[0] /= vector[3];
    vector[1] /= vector[3];
    vector[2] /= vector[3];
}
```

定义一条射线

我们现在已经成功地把一个被触碰的点转换为世界空间中的两个点了。我们现在可以用这两个点定义一个跨越那个三维场景的射线了。让我们完成 convertNormalized2DPointToRay()：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
    Point nearPointRay =
        new Point(nearPointWorld[0], nearPointWorld[1], nearPointWorld[2]);

    Point farPointRay =
        new Point(farPointWorld[0], farPointWorld[1], farPointWorld[2]);

    return new Ray(nearPointRay,
                   Geometry.vectorBetween(nearPointRay, farPointRay));
}
```

我们同样需要增加 Ray 类的定义。让我们给 Geometry 类加入如下代码：

AirHockeyTouch/src/com/airhockey/android/util/Geometry.java

```
public static class Ray {
    public final Point point;
    public final Vector vector;

    public Ray(Point point, Vector vector) {
        this.point = point;
        this.vector = vector;
    }
}
```

一条射线包含一个起点和一个表示射线方向的向量。为了得到这个向量，我们调用 vectorBetween() 创建了一个从近点指向远点的向量。

让我们在 Geometry 类中为 Vector 类加入一个基本的定义：

AirHockeyTouch/src/com/airhockey/android/util/Geometry.java

```
public static class Vector  {
    public final float x, y, z;

    public Vector(float x, float y, float z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

我们还需要定义 vectorBetween()：

AirHockeyTouch/src/com/airhockey/android/util/Geometry.java

```
public static Vector vectorBetween(Point from, Point to) {
    return new Vector(
        to.x - from.x,
        to.y - from.y,
        to.z - from.z);
}
```

我们现在已经完成了这个拼图的第一部分：把一个被触碰的点转换为一条三维射线。现在我们需要加入相交测试了。

9.2.2 执行相交测试

我们稍早前提到过，如果我们假定木槌是一个球体，相交测试就会相当容易。实际上，

如果我们回过头去看看 handleTouchPress()，我们定义了一个与木槌大小相当的包围球：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java
```
Sphere malletBoundingSphere = new Sphere(new Point(
        blueMalletPosition.x,
        blueMalletPosition.y,
        blueMalletPosition.z),
    mallet.height / 2f);
```

我们还没有定义 Sphere 类，因此，让我们继续并给 Geometry 类加入如下代码：

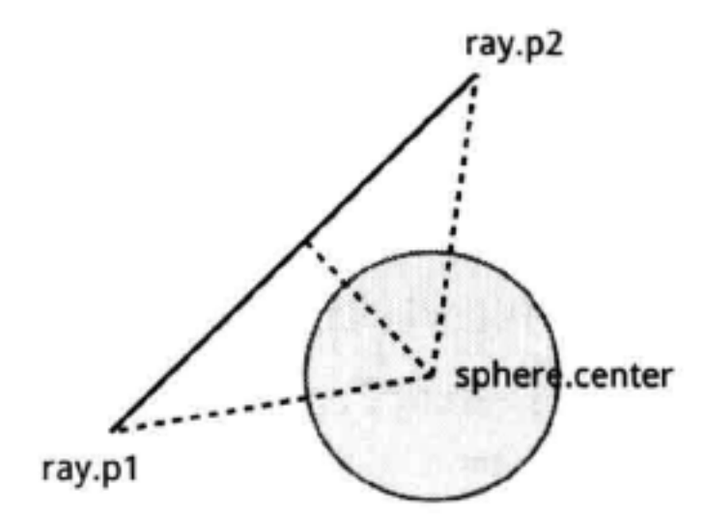
AirHockeyTouch/src/com/airhockey/android/util/Geometry.java
```
public static class Sphere {
    public final Point center;
    public final float radius;

    public Sphere(Point center, float radius) {
        this.center = center;
        this.radius = radius;
    }
}
```

再看一眼 handleTouchPress()，我们还需要加入相交测试：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java
```
malletPressed = Geometry.intersects(malletBoundingSphere, ray);
```

使用三角形计算距离

在我们为此编写代码之前，让我们把这个相交测试可视化，因为这会使得它更容易理解：



为了执行这个测试，我们需要遵循以下步骤：

1. 我们需要计算出球体与射线之间的距离。要得到这个距离，我们首先定义射线上的两个点：起始点和结束点，结束点是由起始点与射线的向量相加而得。接下来，在这两个点与球体的中心点之间创建一个虚拟三角形，最后，通过计算三角形的高就得到了这个距离。
2. 接下来我们比较那个距离和球体的半径。如果那个距离比半径小，那么射线就与球体相交。

关于这个算法的更详细的解释可以参考 Wolfram MathWorld[⊖]。

让我们开始编写代码吧。给 Geometry 类加入如下方法：

AirHockeyTouch/src/com/airhockey/android/util/Geometry.java
```
public static boolean intersects(Sphere sphere, Ray ray) {
    return distanceBetween(sphere.center, ray) < sphere.radius;
}
```

这个方法会决定球心与射线的距离，并检查那个距离是否小于球体半径。如果是，那球体就与射线相交。

⊖ http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html

用向量计算距离

继续并为 distanceBetween() 写出如下代码：

AirHockeyTouch/src/com/airhockey/android/util/Geometry.java

```
public static float distanceBetween(Point point, Ray ray) {
    Vector p1ToPoint = vectorBetween(ray.point, point);
    Vector p2ToPoint = vectorBetween(ray.point.translate(ray.vector), point);
    // The length of the cross product gives the area of an imaginary
    // parallelogram having the two vectors as sides. A parallelogram can be
    // thought of as consisting of two triangles, so this is the same as
    // twice the area of the triangle defined by the two vectors.
    // http://en.wikipedia.org/wiki/Cross_product#Geometric_meaning
    float areaOfTriangleTimesTwo = p1ToPoint.crossProduct(p2ToPoint).length();
    float lengthOfBase = ray.vector.length();

    // The area of a triangle is also equal to (base * height) / 2. In
    // other words, the height is equal to (area * 2) / base. The height
    // of this triangle is the distance from the point to the ray.
    float distanceFromPointToRay = areaOfTriangleTimesTwo / lengthOfBase;
    return distanceFromPointToRay;
}
```

这个方法看起来有点复杂，但是它只是执行我们刚刚提到过的三角形方法而已。

我们首先定义了两个向量：一个是从射线的第一个点到球心，另一个是从射线的第二个点到球心。这两个向量一起定义了一个三角形。

要得到这个三角形的面积，我们首先需要计算这两个向量的交叉乘积（cross product）[⊖]。计算这个交叉乘积会得到第三个向量，它垂直于前两个向量，但是对我们更重要的特性是，这个向量的长度恰好是前两个向量定义的三角形的面积的两倍。

一旦得到了三角形的面积，就可以使用三角形公式计算三角形的高了，这个高就是射线与球体中心点的距离。这个高度等于 (area*2)/lengthOfBase。我们在 areaOfTriangleTimesTwo 中存储了 area*2，并利用 ray.vector 的长度计算三角形底边的长度。要计算三角形的高度，我们只需要把一个除以另外一个。一旦我们有了这个距离，就可以把它与球体的半径作比较了，看看这个射线是否与球体相交。

要是这行得通，我们需要给 Point 类加入一个新的方法：

AirHockeyTouch/src/com/airhockey/android/util/Geometry.java

```
public Point translate(Vector vector) {
    return new Point(
            x + vector.x,
            y + vector.y,
            z + vector.z);
}
```

我们也需要给 Vector 类加入两个额外的方法：

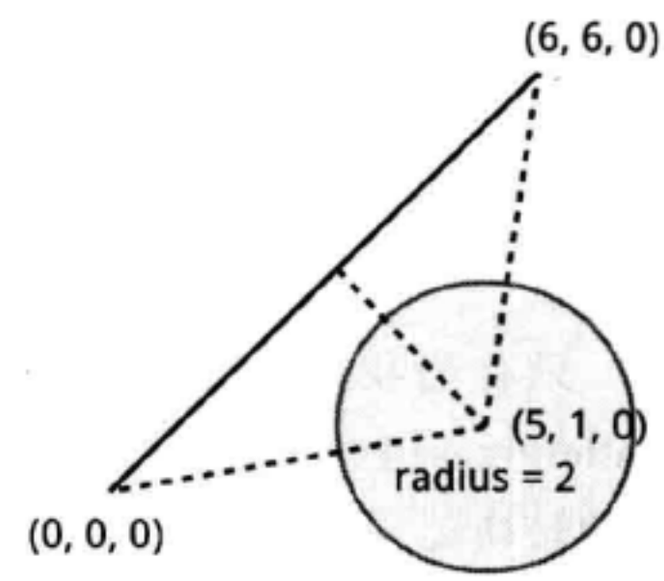
⊖ http://en.wikipedia.org/wiki/Cross_product

AirHockeyTouch/src/com/airhockey/android/util/Geometry.java

```
public float length() {
    return FloatMath.sqrt(
        x * x
      + y * y
      + z * z);
}

// http://en.wikipedia.org/wiki/Cross_product
public Vector crossProduct(Vector other) {
    return new Vector(
        (y * other.z) - (z * other.y),
        (z * other.x) - (x * other.z),
        (x * other.y) - (y * other.x));
}
```

第一个方法——length() 利用勾股定理返回向量的长度[⊖]。第二个方法——crossProduct 计算两个向量的交叉乘积[⊜]。

一个射线与球体相交测试的例子



让我们用一个实际的例子运行一遍代码，使用右图中的坐标点：

这里有一条射线，起点位于（0,0,0），其向量是（6，6，0）。因为我们需要两个点，所以把向量与起始点相加得到第二个点，它位于（6，6，0）。还有一个球体，它的半径是 2，中心点位于（5，1，0）。

我们要做的第一件事情是分配向量 p1ToPoint 和 p2ToPoint。对于 p1ToPoint，我们把它设置为射线的起点和球体的中心之间的向量，因此它被设为（5,1,0）-（0,0,0）=（5,1,0）。对于 p2ToPoint，我们把它设置为射线的第二个点与球体中心之间的向量，因此它被设为（5，1，0）-（6，6，0）=（-1，-5，0）。

下一步是获得三角形的高度。我们首先利用 p1ToPoint 和 p2ToPoint 的交叉乘积的长度获得其两倍的面积。我不想列出所有中间的步骤，但是如果你顺着那些数学公式，你最后会得到一个交叉乘积（0，0，24），其长度为 24。换句话说，这个三角形的面积等于 24 除以 2。

要获得三角形的高度，我们需要把面积乘以 2 并除以底边长。面积乘以 2 等于 24，因此，现在我们只需要利用射线向量（6，6，0）的长度获得底边的长度。这个向量的长度大约是 8.49，因此，如要解决这个高度，只需要用 24/8.49 = 2.82。

我们现在可以做最后的测试了。2.82 大于半径值 2，因此，这条射线一定不与这个球体相交。

现在完成了 handleTouchPress() 的定义。运行这个应用，并加入一些调试语句看看，当你的手指碰到木槌时会发生什么。

⊖ http://en.wikipedia.org/wiki/Pythagorean_theorem

⊜ http://en.wikipedia.org/wiki/Cross_product

9.3 通过拖动移动物体

既然可以测试木槌是否被触碰了，我们将努力解决拼图的下一部分：当我们来回拖动木槌的时候，它要去哪里？我们可以用这种方式考虑事情：木槌平放在桌子上面，当我们来回移动手指的时候，木槌应该随着手指移动并继续平放在桌子上。我们可以通过执行射线－平面（ray-plane）相交测试计算出它的正确位置。

让我们完成 handleTouchDrag() 的定义：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
public void handleTouchDrag(float normalizedX, float normalizedY) {
    if (malletPressed) {
        Ray ray = convertNormalized2DPointToRay(normalizedX, normalizedY);
        // Define a plane representing our air hockey table.
        Plane plane = new Plane(new Point(0, 0, 0), new Vector(0, 1, 0));
        // Find out where the touched point intersects the plane
        // representing our table. We'll move the mallet along this plane.
        Point touchedPoint = Geometry.intersectionPoint(ray, plane);
        blueMalletPosition =
            new Point(touchedPoint.x, mallet.height / 2f, touchedPoint.z);
    }
}
```

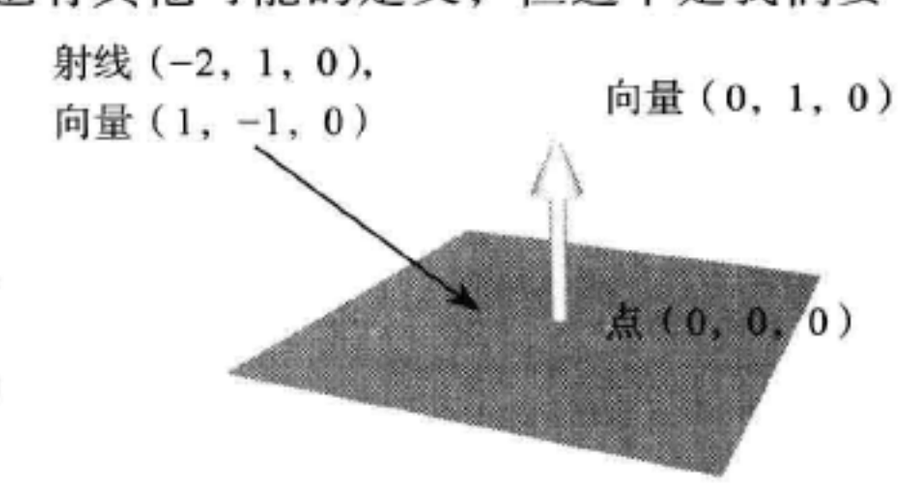
只有当我们开始就用手指按住那个木槌时，我们才想要拖动它，因此，首先检查 malletPressed 是否为 true，如果是，那我们就做射线转换，它与我们在 handleTouchPress() 中的转换是一样的。一旦我们有了表示被触碰点的射线，我们就要找出这条射线与表示空气曲棍球桌子的平面在哪里相交了，然后，把木槌移动到那个点。

在 Geomitry 类中为 Plane 类加入代码：

AirHockeyTouch/src/com/airhockey/android/util/Geometry.java

```
public static class Plane {
    public final Point point;
    public final Vector normal;

    public Plane(Point point, Vector normal) {
        this.point = point;
        this.normal = normal;
    }
}
```

平面的定义非常简单：它包含一个法向向量（normal vector）和平面上的一个点；法向向量仅仅是一个垂直于那个平面的一个向量。平面也有其他可能的定义，但这个是我们要使用的定义。



在右图中，我们可以看到平面的一个例子，它位于（0，0，0），有一个法向向量（0，1，0）：

这里还画了一条射线，位于（-2，1，0），且有一个向量（1，-1，0）。我们要使用这个平面和射线解释相交测试。让我们加入如下代码计算那个交点：

AirHockeyTouch/src/com/airhockey/android/util/Geometry.java

```
public static Point intersectionPoint(Ray ray, Plane plane) {
    Vector rayToPlaneVector = vectorBetween(ray.point, plane.point);

    float scaleFactor = rayToPlaneVector.dotProduct(plane.normal)
                      / ray.vector.dotProduct(plane.normal);

    Point intersectionPoint = ray.point.translate(ray.vector.scale(scaleFactor));
    return intersectionPoint;
}
```

要计算这个交点，我们需要计算出射线的向量要缩放多少才能刚好与平面相接触；这就是缩放因子（scaling factor）。接下来我们用这个被缩放的向量平移射线的点来找出这个相交点。

要计算这个缩放因子，我们首先创建一个向量，它在射线的起点和平面上的一个点之间。然后计算那个向量与平面的法向向量之间的点积（dot product）[⊖]。

这两个向量的点积与它们之间的余弦直接相关（尽管通常不相等）。举个例子，如果有两个平行的向量，（1，0，0）和（1，0，0），那它们之间的角度就是 0 度，这个角的余弦就是 1。如果有两个相互垂直的向量，（1，0，0）和（0，0，1），那它们之间的角度就是 90 度，这个角的余弦就是 0。

为了计算这个缩放量，我们可以用射线到平面（ray-to-plane）的向量与平面法向向量之间的点积除以射线向量与平面法向向量的点积。这就得到了我们需要的缩放因子。

当射线与平面平行时，一个特殊的情况就会发生：在这种情况下，射线与平面是不可能有交点的。射线会与平面的法向向量相垂直，它们之间的点积将为 0，当我们想计算缩放因子的时候，会得到一个除以 0 的除法。最终，我们会得到一个“相交点”，所有分量都是浮点数——NaN，它是“不是一个数字”（Not a Number）的缩写。

如果你不理解所有的细节，也不用担心。重要的是它可以工作；对于数学上的好奇，在 Wikipedia 上有一个很好的解释，你可以从那里学到更多内容[⊜]。

我们还需要给 Vector 类加入如下的代码来填充那些余下的空白：

AirHockeyTouch/src/com/airhockey/android/util/Geometry.java

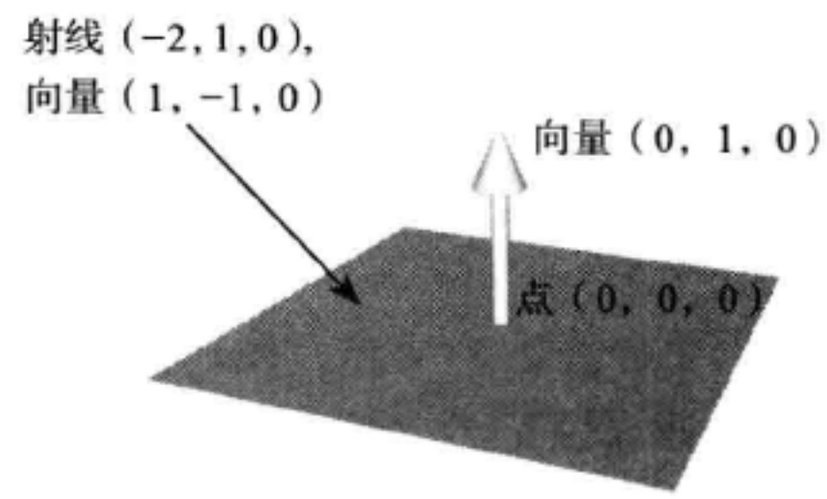
```
public float dotProduct(Vector other) {
    return x * other.x
         + y * other.y
         + z * other.z;
}

public Vector scale(float f) {
    return new Vector(
        x * f,
        y * f,
        z * f);
}
```

⊖ http://en.wikipedia.org/wiki/Dot_product

⊜ http://en.wikipedia.org/wiki/Line-plane_intersection

第一个方法——dotProduct() 计算两个向量之间的点积[⊖]。第二个方法——scale() 用那个缩放量均匀地缩放向量的每个分量。

一个射线与平面相交测试的例子



与以前一样，我们也举一个例子看看这些数字是怎样得出来的。让我们用右图的射线和平面作为例子。

这里有一个平面，位于（0，0，0），其法向向量是（0，1，0），并且还有一个射线，位于（-2，1，0），其向量为（1，-1，0）。如果我们把这个向量扩展得足够远，这个射线会在哪里碰到平面呢？让我们运用数学公式找到答案。

首先，我们需要把平面与射线之间的向量赋值给 rayToPlaneVector。它应该被设为（0，0，0）-（-2，1，0）=（2，-1，0）。

然后，下一步是计算 scaleFactor。一旦计算出那两个点积，这个公式就简化为 -1/-1，将得到值为 1 的缩放因子。

要得到相交点，我们只需要把射线点按那个被缩放的射线向量平移。射线向量被缩放 1 倍，因此，我们可以只把那个向量加到射线点上，得到（-2，1，0）+（1，-1，0）=（-1，0，0）。这就是射线与平面相交的位置。

为了使 handleTouchDrag() 工作，我们已经加入了所有需要的代码。只剩下一个部分：我们需要回到 AirHockeyRenderer，当绘制那个蓝色木槌时，实际使用这个新的点。让我们更新 onDrawFrame()，并把第二个 positionObjectInScene() 调用更新为如下代码：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
positionObjectInScene(blueMalletPosition.x, blueMalletPosition.y,
    blueMalletPosition.z);
```

运行一下程序；现在你应该能在屏幕上来回拖动那个木槌了，它会随着你的指尖移动！

9.4 增加碰撞检测

虽然你有机会找到点乐趣了，可以来回拖动木槌，但你可能已经注意到了第一个问题：木槌可以走到边界外面去，正如图 9-1 所示。在本节中，我们要添加一些基本的碰撞检测，让木槌待在它应该在的地方。我们也会运用一些基本的物理原理，让我们可以在桌子上击打木槌。

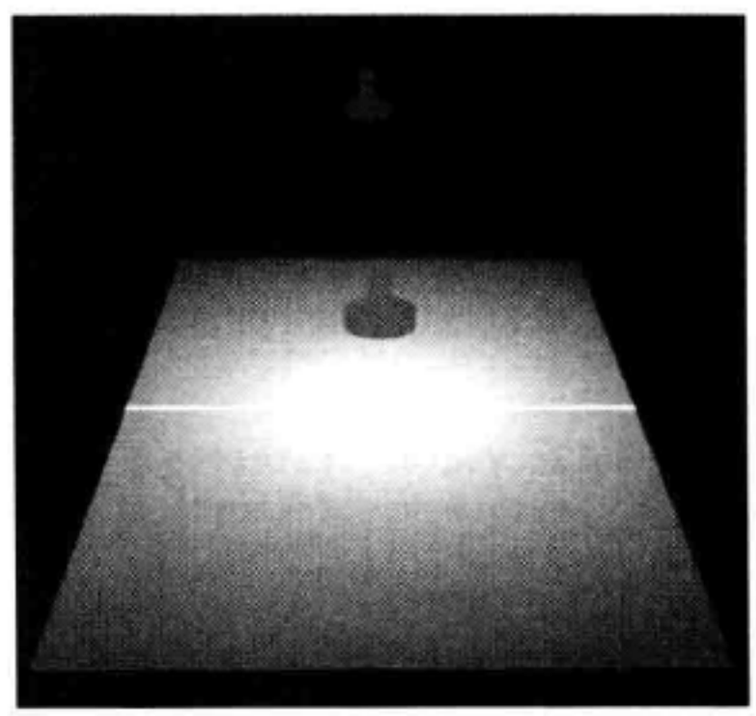


图 9-1 边界外的木槌

⊖ http://en.wikipedia.org/wiki/Dot_product

9.4.1　让玩家的木槌保持在边界内

首先，给 AirHockeyRenderer 加入如下的边界定义：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
private final float leftBound = -0.5f;
private final float rightBound = 0.5f;
private final float farBound = -0.8f;
private final float nearBound = 0.8f;
```

这些定义与空气曲棍球桌子的四边相对应。现在我们可以更新 handleTouchDrag()，并用下面的代码替换 blueMalletPosition 的赋值：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
blueMalletPosition = new Point(
    clamp(touchedPoint.x,
          leftBound + mallet.radius,
          rightBound - mallet.radius),
    mallet.height / 2f,
    clamp(touchedPoint.z,
          0f + mallet.radius,
          nearBound - mallet.radius));
```

如果我们回顾一下 handleTouchDrag()，就会记得 touchedPoint 代表一个相交点，它是我们触碰屏幕的点与空气曲棍球桌子所在平面的相交点。木槌就想移动到这个点上。

要保持木槌不超出桌子的边界，我们把 touchedPoint 限定在桌子的边界内。木槌不能跨过桌子的任何一条边。通过使用 0f，而不是 farbound，我们也把桌子的分隔线考虑进来，这样玩家就不会跨到另一边了，并且我们还把木槌的半径计算在内了，这样木槌的边就不能越过玩家边界的边了。

我们还需要为 clamp() 加入定义：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
private float clamp(float value, float min, float max) {
    return Math.min(max, Math.max(value, min));
}
```

继续并再次运行这个应用。你现在应该发现那个蓝色木槌拒绝移到边界外了。

9.4.2　增加速度和方向

现在，我们可以添加一些代码来用木槌击打冰球了。要知道冰球应该如何作出反应，我们需要回答两个问题：

- 木槌要移动多快？
- 木槌要向那个方向移动？

要回答这两个问题，我们需要随着时间变化持续跟踪木槌是如何移动的。我们要做的第一件事就是给 AirHockeyRenderer 加入一个新的成员变量 previousBlueMalletPosition：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
private Point previousBlueMalletPosition;
```

我们要给它赋一个值，在给 blueMalletPosition 赋值之前，在 handleTouchDrag() 中加入如下代码：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java
```
previousBlueMalletPosition = blueMalletPosition;
```

下一步是为冰球存储它的位置、速度和方向。为 AirHockeyRenderer 类加入如下成员变量：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java
```
private Point puckPosition;
private Vector puckVector;
```

我们将使用向量存储冰球的速度和方向。我们需要初始化这些变量，因此让给 onSurfaceCreated() 加入如下代码：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java
```
puckPosition = new Point(0f, puck.height / 2f, 0f);
puckVector = new Vector(0f, 0f, 0f);
```

我们现在可以在 handleTouchDrag() 结尾处加入如下的碰撞检测代码，要确保这段代码被放在语句“if (malletPressed) {”内：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java
```
float distance =
    Geometry.vectorBetween(blueMalletPosition, puckPosition).length();

if (distance < (puck.radius + mallet.radius)) {
    // The mallet has struck the puck. Now send the puck flying
    // based on the mallet velocity.
    puckVector = Geometry.vectorBetween(
        previousBlueMalletPosition, blueMalletPosition);
}
```

这段代码首先检查蓝色木槌和冰球之间的距离，然后，它会看那个距离是否小于它们的半径之和。如果是，那木槌就击中了冰球，并且我们用前一个木槌的位置和当前木槌的位置给冰球创建了一个方向向量。木槌移动得越快，那个向量就会越大，冰球也会移动得越快。

我们需要更新 onDrawFrame()，以便冰球可以在每一帧上移动。让我们在 onDrawFrame() 的开始处加入如下代码：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java
```
puckPosition = puckPosition.translate(puckVector);
```

作为最后一步，在我们绘制冰球之前，也需要更新 positionObjectInScene()，如下代码所示：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java
```
positionObjectInScene(puckPosition.x, puckPosition.y, puckPosition.z);
```

再次运行这个程序，当你用木槌击打冰球时，看看会发生什么。

9.4.3　加入对边界的反射

我们现在有了另外一个问题：冰球可以移动，但是正如图 9-2 所示，它只是一直不停地向前移动。

为了修复这个问题，我们也不得不给冰球增加边界检查，并且，无论何时当它撞到桌子边缘，都要把它从桌子边缘弹开。

在 onDrawFrame() 中，我们可以在 puckPosition.translate() 调用之后加入如下代码：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
if (puckPosition.x < leftBound + puck.radius
 || puckPosition.x > rightBound - puck.radius) {
    puckVector = new Vector(-puckVector.x, puckVector.y, puckVector.z);
}
if (puckPosition.z < farBound + puck.radius
 || puckPosition.z > nearBound - puck.radius) {
    puckVector = new Vector(puckVector.x, puckVector.y, -puckVector.z);
}
// Clamp the puck position.
puckPosition = new Point(
    clamp(puckPosition.x, leftBound + puck.radius, rightBound - puck.radius),
    puckPosition.y,
    clamp(puckPosition.z, farBound + puck.radius, nearBound - puck.radius)
);
```

我们首先检查冰球是否向左移动得过远或是向右移动得过远了。如果是，那我们就通过反转那个向量的 x 分量变换它的方向。

我们接着检查冰球是否越过了桌子的近边或远边。在这种情况下，我们通过反转那个向量的 z 分量变换它的方向。不要对 z 的检查感到困惑，因为负的 z 值指的是距离，东西离得越远，z 值越小。

最后，通过把冰球限定在桌子的边界内，我们把它放回桌子的范围内了。如果我们再试一下这个程序，冰球现在应该在桌子内来回弹了，而不是飞离它的边界。

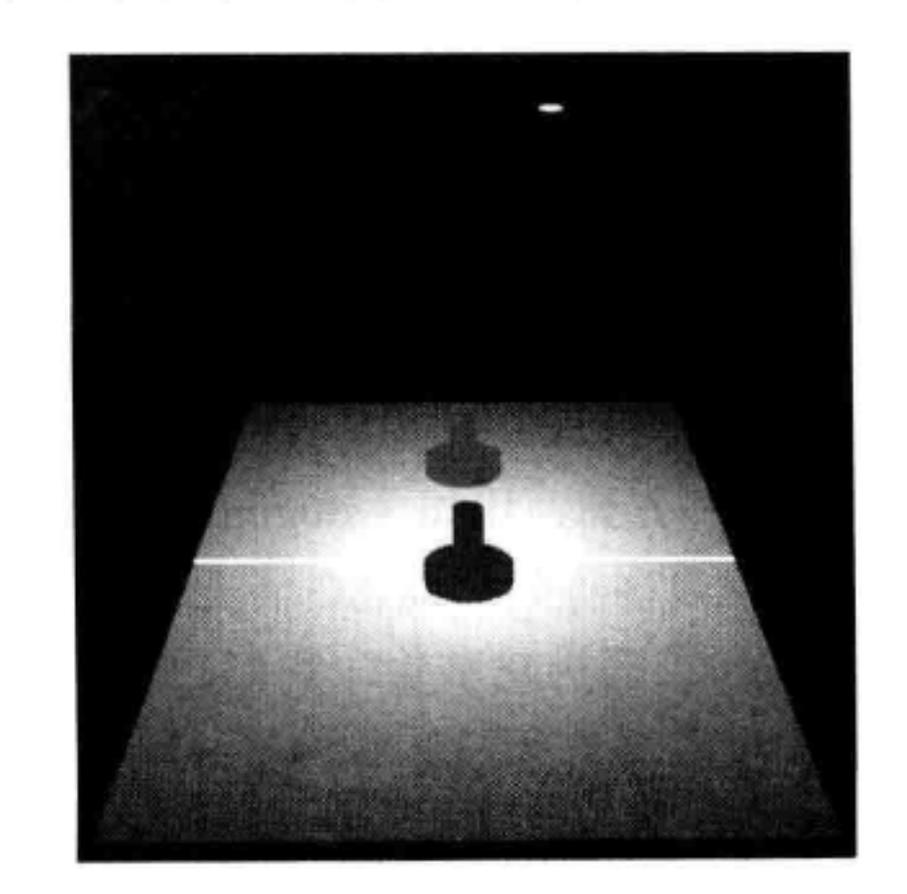


图 9-2　边界外的冰球

9.4.4 增加摩擦

冰球的移动方式还有一个大问题：它永远不会慢下来！这看起来非常不现实，因此，我们要加入一些模拟阻尼的代码使冰球渐渐地慢下来。要使冰球在每一帧中慢下来，在 onDrawFrame() 中，冰球相关的代码结尾处加入如下代码：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
puckVector = puckVector.scale(0.99f);
```

如果再运行一次这个程序，我们就会看到冰球慢下来了，并最终停了下来。通过给桌子的反弹加入额外的阻尼，我们可以使这个游戏更加真实。一旦落入每次反弹检查的条件体内，加入两遍下面的代码：

AirHockeyTouch/src/com/airhockey/android/AirHockeyRenderer.java

```
puckVector = puckVector.scale(0.9f);
```

现在，当冰球从桌子边缘弹开时，我们将看到它变得更慢了。

9.5 小结

我们在本章讲述了一些有趣的主题：首先，我们学习了如何用手指抓住并移动木槌，然后，我们学习了如何让冰球在桌子内来回反弹。在这个开发过程中，你可能已经注意到了重叠（overlap）的问题；我们会在 12.5.1 节中学习如何解决它。

有些数学可能超出了我们的理解，但重要的部分是在较高的层次上理解那些概念，以便我们知道如何使用它们。有许多很好的库可以使事情更简单些，比如 Bullet physics 和 JBox2D[⊖]。

有很多方式可以扩展我们所学的知识：比如，可以创建一个保龄球游戏，一个球被玩家扔出去，同时观察那个球沿着球道向前滚动，并撞击远处的木瓶。触控交互是真正使移动平台不同的地方，它带来了丰富的想象空间。

我们现在已经到了空气曲棍球项目的结尾了。花点儿时间坐下来，回忆一下我们学过的所有内容，因为我们确实走了很长的路。我们距离拥有一个完整的游戏已经不远了；我们所需的就是一些声音、有基本人工智能（AI）的对手、一个菜单和一些这样或那样的特效，我们已经准备好继续了。有些库可以解决其中一些工作，比如 libgdx[⊜]。你也可以通过一本书更详细地研究游戏开发的这些方面，比如《Beginning Android Games》。

在开发过程中，我们也学习了很多重要的概念。首先，我们搞清楚了着色器是如何工作的，以及通过学习颜色、矩阵和纹理构建事物，我们甚至学习了如何构建简单的物体，

⊖ 分别是：http://bulletphysics.org/ 和 http://www.jbox2d.org/。

⊜ https://code.google.com/p/libgdx/

并用手指移动它们。对你已经学到的和完成的内容感到骄傲吧，因为我们是只依靠我们自己通过直接在底层使用 OpenGL 走到这的。

9.6 练习

在学习本书的下一部分之前，让我们花些时间完成下面的练习：

- 因为我们没有在每一帧上改变视图矩阵，你能做些什么来优化 viewProjectionMatrix 和 invertedViewPorjectionMatrix 的更新呢？
- 当前，不管木槌是直接击中冰球还是它的侧面，它都作出同样的反应。更新碰撞检测的代码，把撞击角度考虑进去；图 8-1 可能会给你如何解决这个问题的建议。
- 更新碰撞检测的代码，当冰球自己移动的时候，以便它可以与木槌交互；还有个悬赏分，使它的移动帧速率独立。提示：存储移动向量作为每秒钟的单位，计算出每帧之间逝去的时间，并以此计算出那一帧的移动增量。

对于进一步的参考，花些时间在线研究一下交叉乘积和点积也是个不错的主意，看一下那些视觉化的例子有助于更好地理解它们是怎么工作的。

一旦你完成了这些练习，我们就可以继续学习下一章了。

第二部分 *Part 2*

构建三维世界

- 第10章　用粒子增添趣味
- 第11章　增加天空盒
- 第12章　增加地形
- 第13章　点亮世界
- 第14章　创建动态壁纸
- 第15章　采取下一步行动

Chapter 10 第 10 章

用粒子增添趣味

我们将要改变开发方向，并开始探索更多 OpenGL 背后的艺术。在下面几章里，我们要开始用高山、云雾以及一些有趣的特效构建一幅风景。

在这个项目结束时，我们会把这幅风景变成一个三维动态壁纸，它可以运行在你的 Android 设备的主屏幕上，以此圆满结束。与三维游戏一样，三维动态壁纸也已经迅速普及了，因为 Android 设备越来越快，而且其屏幕越来越大、越来越明亮。从某种程度来说，这些动态壁纸也比较容易创建，因为其关注点更多的在于艺术和审美，通常不需要关心游戏的逻辑。

美存在于观察者的眼中，帮助编程的技巧与成为艺术家所需要的技巧是不一样的。不管怎样，我们都要学习其中的很多技术，使用这些技术可以使事物看起来漂亮且真实。我们将学习如何使用光照、混合和更多的技术，以及如何让它们一起工作创建一个场景。到我们完成本书的这部分时，这个项目看起来将如图 10-1 所示。

图 10-1　一个简单的场景：夜景中的点光源

在本章，我们会以探索粒子（particles）的世界作为新项目的开始，粒子就是被画作一组点的简单物体。通过粒子，我们可以用几个基本物理现象的组合渲染出一些相当优美的效果。我们可以创建向空中喷射水滴的喷泉，并看着它们回落到地上。我们可以模拟下雨的效果，还可以创建爆炸和烟花的效果。粒子背后的数学并不复杂，使它们容易添加到任何三维场景中。

在开始之前，让我们先预览一下本章的开发计划：

- 首先，我们要讲述建立一个粒子系统需要什么。
- 然后，我们要添加一些喷泉，让它们向空中喷射一些粒子。
- 我们也会学习如何让粒子更好看，这需要一些技术，比如，混合和点精灵（point sprite）。

我们可以重用 AirHockeyTouch 项目中的大部分代码，但是我们需要先做一些清理工作。作为项目的开始，把 AirHockeyTouch 复制到一个名为“Particles”的新项目，然后，按照下面的这些步骤修整一下：

1. 把包重命名为“com.particles.android”。你需要更新 Java 包以及 AndroidManifest.xml 中的引用。
2. 把 Activity 组件重命名为“ParticlesActivity”，并把着色器的类重命名为“ParticlesRenderer”。
3. 移除所有 com.particles.android.objects 中的类。
4. 从 com.particles.android.programs 中移除 ColorShaderProgram 及 TextureShaderProgram。
5. 移除 ParticlesRenderer 的内容，只留下构造函数和空白的 Renderer 接口方法的定义，在项目的开发过程中，我们会再把这些内容添加进来。
6. 从 ParticlesActivity 中移除触摸监听器（touch listener）。
7. 通过修改 string.xml 中的 app_name 的值把应用名字改为“Particles”。
8. 最后，移除“/res/raw/”和“/res/drawable-nodpi/”中的所有内容。

现在我们就有了一个干净的基础了，将以此作为开始。

10.1　为一个简单的粒子系统创建一组着色器

作为开始，让我们增加一个非常简单的粒子系统，它代表一个喷泉。我们可以把这想象为一个光照下的喷水泉，或者我们在烟花表演中看到的一个烟花喷泉。为了让这个喷泉启动，我们需要处理一些技术细节。

首先，我们需要某种方式，它可以在内存中表示所有的粒子。我们可以为此使用一个 Java 对象的数组，但是在运行时创建、删除大量的对象消耗可能太大了，而且没有一种简单的方法把数据传递给 OpenGL。作为替代，我们可以把所有的粒子数据内嵌到一个固定大小的数组中，如图 10-2 所示。要添加一个粒子，我们只需要增加粒子的计数，把数据写进粒子数组，并把变动的内容复制到本地缓冲区。当空间不够时，我们可以通过在其开头处重新开始以便回收这个数组。

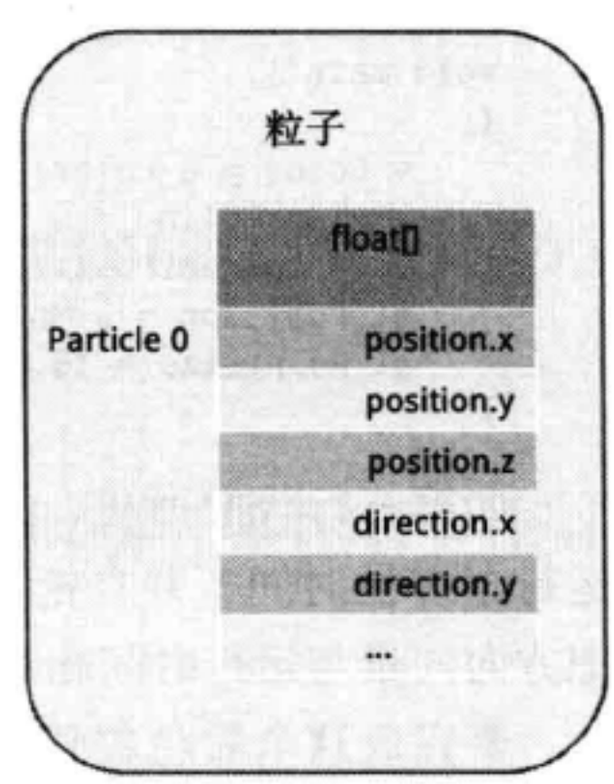


图 10-2　粒子在内存中的布局

我们也需要一个方法来绘制每个粒子。我们可以把每个粒

子表示为一个顶点，并把这些顶点绘制为一组点，每个点都有一个唯一的位置和颜色。

最后，我们还需要某种更新这些粒子的方法。通过把这些逻辑放进一个着色器程序中可以让 GPU 做部分的更新工作。我们要为每个粒子存储一个方向向量和一个创建时间；用这个创建时间，我们可以计算出粒子自被创建出来之后运行了多少时间，然后，可以用这个运行时间、方向向量和位置推算出粒子当前的位置。我们将使用一个浮点数储存时间，并用 0.0 表示粒子系统开始运行的时间。

有了这些基本的需求，我们可以为着色器程序总结出一套初始的规范了。我们首先需要一个为投影矩阵所定义的 uniform 和一个为当前时间定义的 uniform，以便着色器可以计算出每个粒子自从被创建后运行了多长时间。我们还需要与粒子属性对应的四个属性：位置、颜色、方向向量和创建时间。

10.1.1 给着色器添加代码

让我们开始给着色器添加代码。继续并在“/res/raw/”文件夹内创建一个名为“particle_vertex_shader.glsl”的新顶点着色器。首先以如下定义作为开始：

Particles/res/raw/particle_vertex_shader.glsl

```
uniform mat4 u_Matrix;
uniform float u_Time;

attribute vec3 a_Position;
attribute vec3 a_Color;
attribute vec3 a_DirectionVector;
attribute float a_ParticleStartTime;

varying vec3 v_Color;
varying float v_ElapsedTime;
```

这些定义满足了我们的粒子着色器的需求。我们在片段着色器中也需要使用颜色和运行时间，因此我们也为这两个变量创建了两个 varying。让我们完成这个顶点着色器：

Particles/res/raw/particle_vertex_shader.glsl

```
void main()
{
    v_Color = a_Color;
    v_ElapsedTime = u_Time - a_ParticleStartTime;
    vec3 currentPosition = a_Position + (a_DirectionVector * v_ElapsedTime);
    gl_Position = u_Matrix * vec4(currentPosition, 1.0);
    gl_PointSize = 10.0;
}
```

如第三行代码所见，我们首先把颜色发送给片段着色器，接着计算粒子从被创建之后运行了多少时间，并且把那个时间也发送给片段着色器。为了计算粒子的当前位置，我们把方向向量与运行时间相乘，并与 a_Position 相加。运行时间越长，粒子走得越远。

要完成这个着色器代码，我们把粒子用那个矩阵进行投影，而且，因为我们把粒子渲染为一个点，所以把点的大小设成了 10 个像素。

当我们做数学运算时，要确保不会意外地把 w 分量弄混乱，这是很重要的。因此，我们将用 3 分量向量表示位置和方向，只有需要把它与 u_Matrix 相乘时，才把它转换为完全的 4 分量向量。这确保上面的数学运算只影响 x、y 和 z 分量。

现在我们可以继续并添加片段着色器了。在与顶点着色器相同的地方创建一个被称为“particle_fragment_shader.glsl”的新文件，并加入如下代码：

Particles/res/raw/particle_fragment_shader.glsl

```
precision mediump float;
varying vec3 v_Color;
varying float v_ElapsedTime;
void main()
{
    gl_FragColor = vec4(v_Color / v_ElapsedTime, 1.0);
}
```

通过把颜色除以运行时间，这个着色器会使年轻的粒子明亮，而使年老的粒子暗淡。如果发生除以 0 的情况怎么办？根据规范，这会导致一个不明确的结果，但不会导致着色器程序终止。要一个更加可预测的结果，你可以总是给分母加上一个很小的数[⊖]。

10.1.2　用一个 Java 类封装着色器

着色器代码完成了，我们现在可以用一个 Java 类封装着色器，它使用我们在本书第一部分所使用的模式。让我们首先给 ShaderProgram 加入一些新的常量：

Particles/src/com/particles/android/programs/ShaderProgram.java

```
protected static final String U_TIME = "u_Time";

protected static final String A_DIRECTION_VECTOR = "a_DirectionVector";
protected static final String A_PARTICLE_START_TIME = "a_ParticleStartTime";
```

这些新的常量被定义后，我们可以继续在 com.particles.android.programs 包中加入一个名为“ParticleShaderProgram”的新类，它继承自 ShaderProgram，并以如下代码作为这个类的开始：

Particles/src/com/particles/android/programs/ParticleShaderProgram.java

```
// Uniform locations
private final int uMatrixLocation;
private final int uTimeLocation;

// Attribute locations
private final int aPositionLocation;
private final int aColorLocation;
private final int aDirectionVectorLocation;
private final int aParticleStartTimeLocation;
```

让我们完成这个类的定义：

⊖ http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf

Particles/src/com/particles/android/programs/ParticleShaderProgram.java

```
public ParticleShaderProgram(Context context) {
    super(context, R.raw.particle_vertex_shader,
        R.raw.particle_fragment_shader);

    // Retrieve uniform locations for the shader program.
    uMatrixLocation = glGetUniformLocation(program, U_MATRIX);
    uTimeLocation = glGetUniformLocation(program, U_TIME);

    // Retrieve attribute locations for the shader program.
    aPositionLocation = glGetAttribLocation(program, A_POSITION);
    aColorLocation = glGetAttribLocation(program, A_COLOR);
    aDirectionVectorLocation = glGetAttribLocation(program, A_DIRECTION_VECTOR);
    aParticleStartTimeLocation =
        glGetAttribLocation(program, A_PARTICLE_START_TIME);
}

public void setUniforms(float[] matrix, float elapsedTime) {
    glUniformMatrix4fv(uMatrixLocation, 1, false, matrix, 0);
    glUniform1f(uTimeLocation, elapsedTime);
}
public int getPositionAttributeLocation() {
    return aPositionLocation;
}
public int getColorAttributeLocation() {
    return aColorLocation;
}
public int getDirectionVectorAttributeLocation() {
    return aDirectionVectorLocation;
}
public int getParticleStartTimeAttributeLocation() {
    return aParticleStartTimeLocation;
}
```

它使用了与第一部分相同的模式封装了这个类。

10.2 添加粒子系统

我们现在可以开始创建粒子系统了。让我们在 com.particles.android.objects 包中创建一个名为“ParticleSystem”的新类，这个类以如下代码开始：

Particles/src/com/particles/android/objects/ParticleSystem.java

```
private static final int POSITION_COMPONENT_COUNT = 3;
private static final int COLOR_COMPONENT_COUNT = 3;
private static final int VECTOR_COMPONENT_COUNT = 3;
private static final int PARTICLE_START_TIME_COMPONENT_COUNT = 1;

private static final int TOTAL_COMPONENT_COUNT =
    POSITION_COMPONENT_COUNT
  + COLOR_COMPONENT_COUNT
  + VECTOR_COMPONENT_COUNT
  + PARTICLE_START_TIME_COMPONENT_COUNT;

private static final int STRIDE = TOTAL_COMPONENT_COUNT * BYTES_PER_FLOAT;
```

目前为止，我们只有一些基本定义，它们是为分量计数和粒子之间的跨距添加的。让我们继续构建这个类：

Particles/src/com/particles/android/objects/ParticleSystem.java

```
private final float[] particles;
private final VertexArray vertexArray;
private final int maxParticleCount;

private int currentParticleCount;
private int nextParticle;

public ParticleSystem(int maxParticleCount) {
    particles = new float[maxParticleCount * TOTAL_COMPONENT_COUNT];
    vertexArray = new VertexArray(particles);
    this.maxParticleCount = maxParticleCount;
}
```

我们现在有了一个存储粒子的浮点数组和一个 VertexArray，VertexArray 用来代表我们要发送给 OpenGL 的数据，因为数组的大小是固定的，我们还定义了保存最大粒子数的 maxParticleCount。我们将使用 currentParticleCount 和 nextParticle 持续记录数组中的粒子。

让我们开始构建一个名为 addParticle() 的新方法：

Particles/src/com/particles/android/objects/ParticleSystem.java

```
public void addParticle(Point position, int color, Vector direction,
    float particleStartTime) {
    final int particleOffset = nextParticle * TOTAL_COMPONENT_COUNT;

    int currentOffset = particleOffset;
    nextParticle++;

    if (currentParticleCount < maxParticleCount) {
        currentParticleCount++;
    }

    if (nextParticle == maxParticleCount) {
        // Start over at the beginning, but keep currentParticleCount so
        // that all the other particles still get drawn.
        nextParticle = 0;
    }
```

要创建新的粒子，首先要传入位置、颜色、方向和粒子创建时间。颜色作为 Android 的颜色定义被传递进来，我们需要用 Android 的 Color 类把这个颜色分解为单独的分量。

在给数组加入新的粒子之前，我们需要计算它需要存在哪里。因为所有的粒子都一起存在我们的数组中，它类似于一种无定形的 blob（二进制大对象）。为了计算正确的偏移值，我们使用 nextParticle 存储下一个粒子的编号，其中第一个粒子从 0 开始编号。然后，我们可以通过把 nextParticle 与每个粒子的分量计数相乘得到偏移值。我们把这个偏移值存储在 particleOffset 和 currentOffset 中；我们使用 particleOffset 记住新粒子从哪里开始，而用 currentOffset 记住新粒子的每个属性的位置。

一个新粒子每次被添加进来时，我们就给 nextParticle 增加 1，当到了数组的结尾处，

我们就从 0 开始以便回收最旧的粒子。我们也需要记录有多少个粒子需要被绘制出来，为此，一个新粒子每次被加入的时候，我们都增加 currentParticleCount，但要把它限制在最大粒子数内。

我们已经解释了这些操作，接下来让我们把这个新的粒子数据写到数组中：

Particles/src/com/particles/android/objects/ParticleSystem.java

```
particles[currentOffset++] = position.x;
particles[currentOffset++] = position.y;
particles[currentOffset++] = position.z;

particles[currentOffset++] = Color.red(color) / 255f;
particles[currentOffset++] = Color.green(color) / 255f;
particles[currentOffset++] = Color.blue(color) / 255f;

particles[currentOffset++] = direction.x;
particles[currentOffset++] = direction.y;
particles[currentOffset++] = direction.z;

particles[currentOffset++] = particleStartTime;
```

首先存进的是位置，然后是颜色（使用 Android 的 Color 类分解出每个分量），紧接着是方向向量，最后是粒子创建时间。Android 的 Color 类返回的分量范围是从 0 到 255，而 OpenGL 期望的范围是从 0 到 1，因此，通过用每个分量都除以 255，我们把它从 Android 的范围转换为 OpenGL 期望的范围（更多细节请参考 4.3 节）。

我们还需要把这个新的粒子复制到本地缓冲区，以便 OpenGL 可以存取这些新的数据，因此，让我们用下面的方法调用完成 addParticle()：

Particles/src/com/particles/android/objects/ParticleSystem.java

```
vertexArray.updateBuffer(particles, particleOffset, TOTAL_COMPONENT_COUNT);
```

我们只想复制这些新数据，这样就不会浪费时间去复制那些没有改变的数据，因此我们传递进起始偏移值和那个计数。由于 updateBuffer() 作为 VertexArray 内部的一个新方法，我们还需要加入它的定义。为此，我们在类的结尾处加入如下代码：

Particles/src/com/particles/android/data/VertexArray.java

```
public void updateBuffer(float[] vertexData, int start, int count) {
   floatBuffer.position(start);
   floatBuffer.put(vertexData, start, count);
   floatBuffer.position(0);
}
```

现在，我们可以返回 ParticleSystem，并加入一个绑定函数：

Particles/src/com/particles/android/objects/ParticleSystem.java

```
public void bindData(ParticleShaderProgram particleProgram) {
    int dataOffset = 0;
    vertexArray.setVertexAttribPointer(dataOffset,
        particleProgram.getPositionAttributeLocation(),
        POSITION_COMPONENT_COUNT, STRIDE);
    dataOffset += POSITION_COMPONENT_COUNT;
```

```
vertexArray.setVertexAttribPointer(dataOffset,
    particleProgram.getColorAttributeLocation(),
    COLOR_COMPONENT_COUNT, STRIDE);
dataOffset += COLOR_COMPONENT_COUNT;

vertexArray.setVertexAttribPointer(dataOffset,
    particleProgram.getDirectionVectorAttributeLocation(),
    VECTOR_COMPONENT_COUNT, STRIDE);
dataOffset += VECTOR_COMPONENT_COUNT;

    vertexArray.setVertexAttribPointer(dataOffset,
        particleProgram.getParticleStartTimeAttributeLocation(),
        PARTICLE_START_TIME_COMPONENT_COUNT, STRIDE);
}
```

这又只是一段样板式的代码，它遵循与前面几章一样的模式，在着色器程序中把顶点数据与正确的属性绑定，并小心遵守我们在 addParticle() 中所使用的顺序。如果我们把颜色和方向向量的顺序搞混了，或者犯了此类错误，当我们绘制那些粒子时，我们将看到相当滑稽的结果。

让我们加入一个 draw() 函数来结束这个类：

Particles/src/com/particles/android/objects/ParticleSystem.java

```
public void draw() {
    glDrawArrays(GL_POINTS, 0, currentParticleCount);
}
```

我们现在就实现了一个粒子系统。这个系统可以让我们添加一定量的粒子，回收旧的粒子，并在内存中有效地定位那些彼此相邻的粒子。

添加一个粒子喷泉

有了这个粒子系统，我们现在需要某个程序，它可以为我们实际生成一些粒子，并把它们添加到粒子系统中。让我们创建一个粒子喷泉，在与 ParticleSystem 相同的包中创建一个名为“ParticleShooter”的新类，并加入如下代码：

Particles/src/com/particles/android/objects/ParticleShooter.java

```
private final Point position;
private final Vector direction;
private final int color;

public ParticleShooter(Point position, Vector direction, int color) {
    this.position = position;
    this.direction = direction;
    this.color = color;
}
```

我们已经给定了这个粒子发射器的位置、方向和颜色；当我们创建新的粒子时，我们只要把这些直接传递给粒子系统。让我们继续编写这个粒子发射器：

Particles/src/com/particles/android/objects/ParticleShooter.java

```
public void addParticles(ParticleSystem particleSystem, float currentTime,
    int count) {
```

```
        for (int i = 0; i < count; i++) {
            particleSystem.addParticle(position, color, direction, currentTime);
        }
    }
```

在 addParticles() 中，我们传进了粒子系统、要添加多少粒子以及粒子系统的当前时间。我们现在已经有了所有的组件，只需要在 Renderer 类中加入一些调用把这一切粘合起来。

10.3 绘制粒子系统

只需要给 ParticlesRenderer 加入一些代码，就能最终看到这些粒子了。让我们以下面的定义开始：

Particles/src/com/particles/android/ParticlesRenderer.java

```
private final Context context;

private final float[] projectionMatrix = new float[16];
private final float[] viewMatrix = new float[16];
private final float[] viewProjectionMatrix = new float[16];

private ParticleShaderProgram particleProgram;
private ParticleSystem particleSystem;
private ParticleShooter redParticleShooter;
private ParticleShooter greenParticleShooter;
private ParticleShooter blueParticleShooter;
private long globalStartTime;

public ParticlesRenderer(Context context) {
    this.context = context;
}
```

这里定义了 Android 的 Context 和那些矩阵的标准变量，以及粒子着色器、粒子系统和三个粒子发射器的变量。我们还定义了全局启动时间的变量和一个标准构造函数。

接下来，让我们定义 onSurfaceCreated() 的内容：

Particles/src/com/particles/android/ParticlesRenderer.java

```
@Override
public void onSurfaceCreated(GL10 glUnused, EGLConfig config) {
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

    particleProgram = new ParticleShaderProgram(context);
    particleSystem = new ParticleSystem(10000);
    globalStartTime = System.nanoTime();

    final Vector particleDirection = new Vector(0f, 0.5f, 0f);

    redParticleShooter = new ParticleShooter(
        new Point(-1f, 0f, 0f),
        particleDirection,
        Color.rgb(255, 50, 5));

    greenParticleShooter = new ParticleShooter(
```

```
        new Point(0f, 0f, 0f),
        particleDirection,
        Color.rgb(25, 255, 25));

    blueParticleShooter = new ParticleShooter(
        new Point(1f, 0f, 0f),
        particleDirection,
        Color.rgb(5, 50, 255));
}
```

我们把清屏的颜色设为黑色，初始化粒子着色器程序，并用一万个粒子的最大限量初始化一个新的粒子系统，然后，我们使用 System.nanoTime() 获得当前系统时间，并用它设置全局启动时间。我们想在粒子系统中使用浮点数时间，因此，当粒子系统被初始化时，当前时间将是 0.0，在此时创建的粒子，其创建时间将是 0.0。5 秒以后，新粒子的创建时间是 5.0。为此，我们可以用当前系统时间与 globalStartTime 的差值作为创建时间，因为 System.nanoTime() 返回以纳秒为单位的时间，我们只要把这个差值除以一万亿就可以转换为秒。

这个方法的下半部分建立了三个粒子喷泉。每个喷泉由一个粒子发射器表示，每个发射器都将按照 particleDirection 定义的方向或者沿 y 轴垂直向上发射它的粒子。我们把这三个喷泉按照从左到右的顺序排列，并且设置了它们的颜色：第一个是红色，第二个是绿色，第三个是蓝色。

让我们加入 onSurfaceChanged() 的定义：

Particles/src/com/particles/android/ParticlesRenderer.java

```
@Override
public void onSurfaceChanged(GL10 glUnused, int width, int height) {
    glViewport(0, 0, width, height);

    MatrixHelper.perspectiveM(projectionMatrix, 45, (float) width
        / (float) height, 1f, 10f);

    setIdentityM(viewMatrix, 0);
    translateM(viewMatrix, 0, 0f, -1.5f, -5f);
    multiplyMM(viewProjectionMatrix, 0, projectionMatrix, 0,
        viewMatrix, 0);
}
```

这是一个标准定义，它用一个常规的透视投影和视图矩阵把物体放在正确的空间中。让我们为 onDrawFrame() 加入如下定义完成这个渲染器：

Particles/src/com/particles/android/ParticlesRenderer.java

```
@Override
public void onDrawFrame(GL10 glUnused) {
    glClear(GL_COLOR_BUFFER_BIT);

    float currentTime = (System.nanoTime() - globalStartTime) / 1000000000f;

    redParticleShooter.addParticles(particleSystem, currentTime, 5);
    greenParticleShooter.addParticles(particleSystem, currentTime, 5);
    blueParticleShooter.addParticles(particleSystem, currentTime, 5);
```

```
        particleProgram.useProgram();
        particleProgram.setUniforms(viewProjectionMatrix, currentTime);
        particleSystem.bindData(particleProgram);
        particleSystem.draw();
    }
```

每次一个新帧被绘制时，我们都计算当前时间并把它传给着色器。它会告诉着色器粒子自从被创建后移动了多远。我们也为每个喷泉生成了 5 个新的粒子，然后用粒子着色程序绘制这些粒子。

运行一下这个程序，几秒钟后，它应该看上去如图 10-3 所示。

我们现在有了一个能工作的系统，随着粒子运行时间的变化，我们会看到它由明变暗的效果，但是它看起来还是很奇怪，不是吗？当粒子向上移动时，它们不应该扩散开么？因为重力的作用，它们不应该向下回落么？接下来我们要解决这些问题及其他更多的问题。

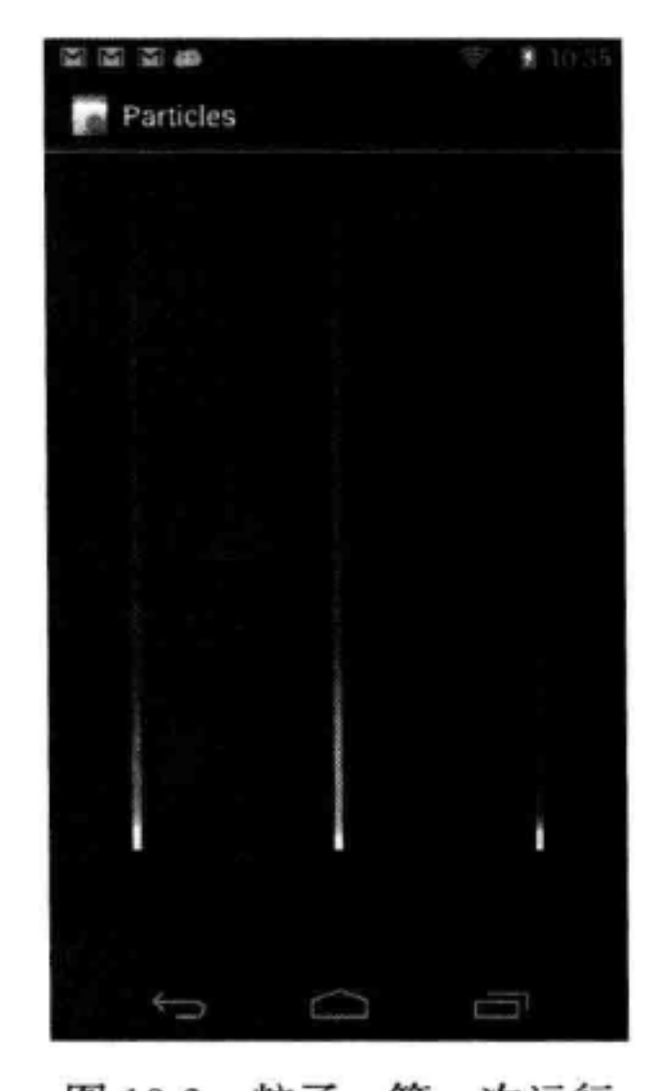


图 10-3 粒子：第一次运行

10.4 扩散粒子

我们要做的第一件事是扩散粒子，我们也将变化每个粒子的速度，让每个粒子喷泉有更多的变化。让我们回到 ParicleShooter 类，为其加入如下成员变量：

Particles/src/com/particles/android/objects/ParticleShooter.java

```
private final float angleVariance;
private final float speedVariance;

private final Random random = new Random();

private float[] rotationMatrix = new float[16];
private float[] directionVector = new float[4];
private float[] resultVector = new float[4];
```

每个发射器都将有一个角度变化量，用来控制粒子的扩散，以及一个速度变化量，用来改变每个粒子的速度。我们也有一个矩阵和两个向量，因此，我们可以使用 Android 的 Matrix 类做些数学运算。

按如下代码更新构造函数的签名：

Particles/src/com/particles/android/objects/ParticleShooter.java

```
public ParticleShooter(
    Point position, Vector direction, int color,
    float angleVarianceInDegrees, float speedVariance) {
```

接下来加入如下代码给这些新的成员变量赋值：

Particles/src/com/particles/android/objects/ParticleShooter.java

```
this.angleVariance = angleVarianceInDegrees;
this.speedVariance = speedVariance;

directionVector[0] = direction.x;
directionVector[1] = direction.y;
directionVector[2] = direction.z;
```

现在，我们只需要更新 addParticles()，应用角度和速度的变化量。

按如下代码更新 for 循环的内容：

Particles/src/com/particles/android/objects/ParticleShooter.java

```
setRotateEulerM(rotationMatrix, 0,
    (random.nextFloat() - 0.5f) * angleVariance,
    (random.nextFloat() - 0.5f) * angleVariance,
    (random.nextFloat() - 0.5f) * angleVariance);

multiplyMV(
    resultVector, 0,
    rotationMatrix, 0,
    directionVector, 0);

float speedAdjustment = 1f + random.nextFloat() * speedVariance;

Vector thisDirection = new Vector(
    resultVector[0] * speedAdjustment,
    resultVector[1] * speedAdjustment,
    resultVector[2] * speedAdjustment);

particleSystem.addParticle(position, color, thisDirection, currentTime);
```

要改变发射角度，我们使用 Android 的 Matrix.setRotateEulerM() 创建一个旋转矩阵，它会用 angleVariance 的一个随机量改变发射角度，其单位是度。接下来我们把这个矩阵与方向向量相乘得到一个角度稍小的旋转向量。要调整速度，我们把方向向量的每一个分量都与相同的 speedVariance 的随机调整值相乘。一旦完成这些，我们就通过调用 particleSystem.addParticle() 添加这个新的粒子。

有了这些改动，就不再需要 direction 成员变量了，所以把它删掉。现在需要更新 ParticlesRenderer，调整新的构造函数参数。修改 onSurfaceCreated()，以便粒子发射器可以按如下方式创建出来：

Particles/src/com/particles/android/ParticlesRenderer.java

```
final float angleVarianceInDegrees = 5f;
final float speedVariance = 1f;

redParticleShooter = new ParticleShooter(
    new Point(-1f, 0f, 0f),
    particleDirection,
    Color.rgb(255, 50, 5),
    angleVarianceInDegrees,
    speedVariance);

greenParticleShooter = new ParticleShooter(
```

```
        new Point(0f, 0f, 0f),
        particleDirection,
        Color.rgb(25, 255, 25),
        angleVarianceInDegrees,
        speedVariance);

    blueParticleShooter = new ParticleShooter(
        new Point(1f, 0f, 0f),
        particleDirection,
        Color.rgb(5, 50, 255),
        angleVarianceInDegrees,
        speedVariance);
```

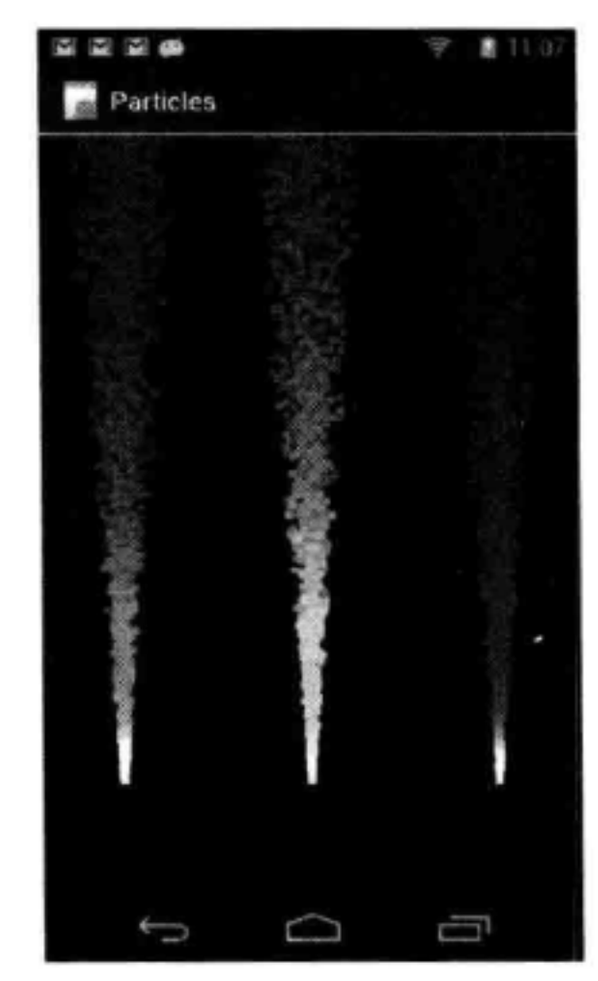


图 10-4 有发射角和发射速度变化量的粒子

我们的系统已经准备好了，每个粒子喷泉都有一个 5 度的发射角变化量以及一个 1 单位的发射速度变化量。继续运行这个应用看看我们这次看到了什么。它看起来应该如图 10-4 所示。

看起来效果好多了！现在我们需要添加重力把那些粒子拉回地面。

10.5 添加重力

任何上升到空中的东西都会掉下来因为地球有吸引力。牛顿受到苹果从树上落下的启发而发现了重力效应，他也因此而出名；如果我们也给粒子加入重力效应，效果看起来会更好。

在地球上，每个人都感受到 $9.8m/s^2$ 的加速度，如果我们忽略风阻的影响，某个东西下落的时间越长，它的速度就越快。通过给顶点着色器加入一个重力调整因子，我们可以很容易地在代码中实现这个效果。打开 particle_vertex_shader.glsl，在 v_ElapsedTime 赋值语句之后加入如下代码行：

Particles/res/raw/particle_vertex_shader.glsl

```
float gravityFactor = v_ElapsedTime * v_ElapsedTime / 8.0;
```

它通过应用重力加速度公式和粒子运行时间的平方值计算得到一个加速重力因子；我们还把它除以 8 以弱化这个效果。数值 8 是被任意使用的：可以使用其他任何数值，同样也能获得不错的效果。现在需要给当前的位置应用这个重力因子，在 currentPosition 赋值语句之后给顶点着色器加入如下代码：

Particles/res/raw/particle_vertex_shader.glsl

```
currentPosition.y -= gravityFactor;
```

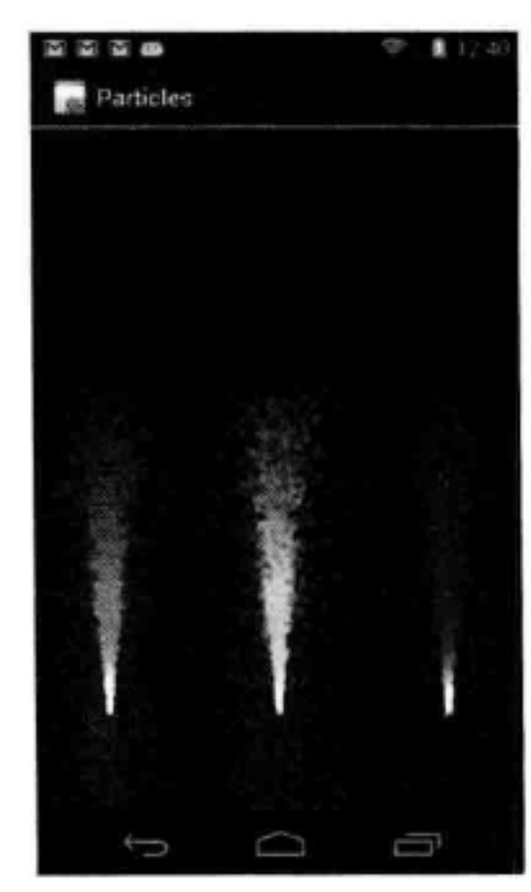


图 10-5 为粒子增加重力效应

让我们再次运行这个应用，看看会发生什么。它看起来应该如图 10-5 所示。

现在我们看到每个向上移动的粒子都渐渐慢下来，并最终向地面回落。我们还可以让它更好看：一些较暗的粒子覆盖了那些较明亮的，这看起来有点奇怪。

10.6　用累加混合技术混合粒子

当我们在 OpenGL 中实现各种效果时，我们不得不经常回想我们要努力仿制的那个效果。如果把这三个粒子流想象成一个喷射的烟花，就像我们在烟花表演上看到的一样，那我们应该期望这些粒子可以发光；粒子越多，它们就应该越亮。模仿这个效果的方法之一是使用累加混合技术。

让我们使能混合技术，在 onSurfaceCreated() 方法里为 ParticlesRenderer 类加入如下代码：

```
Particles/src/com/particles/android/ParticlesRenderer.java
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
```

就是它！我们首先要使能混合技术，然后设置混合模式为累加混合。为了更好地理解这是怎么工作的，让我们看一下 OpenGL 的默认混合公式：

输出 =（源因子 * 源片段）+（目标因子 * 目标片段）

在 OpenGL 里，混合技术的工作原理是把片段着色器的结果和已经在帧缓冲区中的颜色进行混合。源片段的值来自于片段着色器，目标片段的值就是已经在帧缓冲区中的值，源因子和目标因子的值是通过调用 glBlendFunc() 配置的。在我们刚刚添加的代码中，调用 glBlendFunc() 时，我们把每个因子都设为 GL_ONE，它把混合公式变为如下的公式：

输出 =（GL_ONE * 源片段）+（GL_ONE * 目标片段）

通过这个混合模式，来自片段着色器的片段就被累加到已经存在于屏幕上的片段了，这就是为什么它叫累加混合技术。还有其他更多的混合模式，可以在 Khronos 网站上在线查看[⊖]。

我们的粒子现在看上去被点亮了，因为它们混合在一起了。我们还要记住 OpenGL 限定每一个颜色分量的值，因此，如果我们给纯绿色加上纯绿色，得到的还是纯绿色。然而，如果我们只把一小点的红色累加到那个绿色上足够多次，实际上也会偏移它的色调，最终得到黄色；再累加一点蓝色到那个黄色足够多的次数，我们就会看到白色。

考虑到 OpenGL 的限值特性，我们可以实现一些优美的效果。比如，在图 10-6 中，我们的红色烟花喷泉在它的最亮处实际上有点发黄，这是因为我们给它的基色上加入了一点绿色和稍少一点的蓝色。

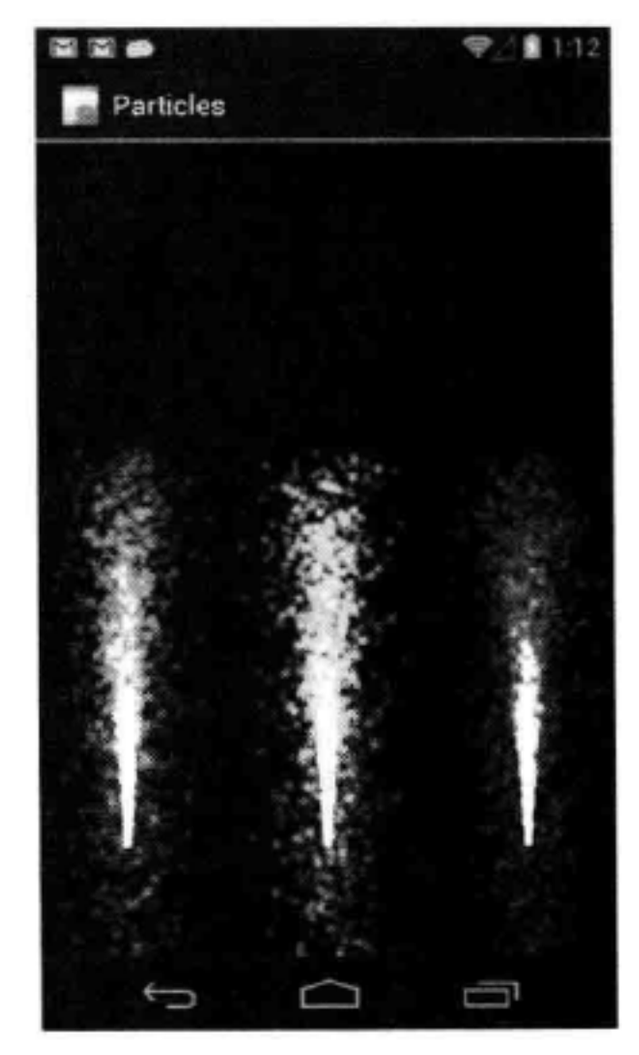


图 10-6　使用累加混合技术混合粒子

10.7　自定义点的外形

你可能已经注意到了，这些点被渲染为小的四方块，其每条边上的像素数量都等于

⊖ http://www.khronos.org/opengles/sdk/docs/man/xhtml/glBlendFunc.xml

gl_PointSize 的值。我们实际上可以使用另一个特殊的 OpenGL 变量 gl_PointCoord 自定义这些点的外形。对于每个点，当调用片段着色器时，我们都会得到一个二维的 gl_PointCoord 坐标空间，在每个轴上，其分量的范围都是从 0 到 1，其取值依赖于点上的哪个片段当前正在被渲染。

要知道这是怎么工作的，我们首先要使用 gl_PointCoord 把片段绘制为圆，而不是方形。我们怎么做呢？渲染每个点时，相对于 gl_PointCoord 上的每个轴来说，其片段的位置范围都是从 0 到 1，因此，我们把点的圆心放在 (0.5, 0.5)，其每条边都有 0.5 单位的空间。换句话说，点的半径是 0.5。要绘制一个圆，我们所要做的就是只绘制那些位于半径内的片段。

首先让我们调高点的大小，以便使它更容易被看到。将 particle_vertex_shader.glsl 中的 gl_PointSize 更新为如下值：

Particles/res/raw/particle_vertex_shader.glsl

```
gl_PointSize = 25.0;
```

现在将片段着色器更新为如下代码：

Particles/res/raw/particle_fragment_shader.glsl

```
precision mediump float;
varying vec3 v_Color;
varying float v_ElapsedTime;
void main()
{
    float xDistance = 0.5 - gl_PointCoord.x;
    float yDistance = 0.5 - gl_PointCoord.y;
    float distanceFromCenter =
        sqrt(xDistance * xDistance + yDistance * yDistance);

    if (distanceFromCenter > 0.5) {
        discard;
    } else {
        gl_FragColor = vec4(v_Color / v_ElapsedTime, 1.0);
    }
}
```

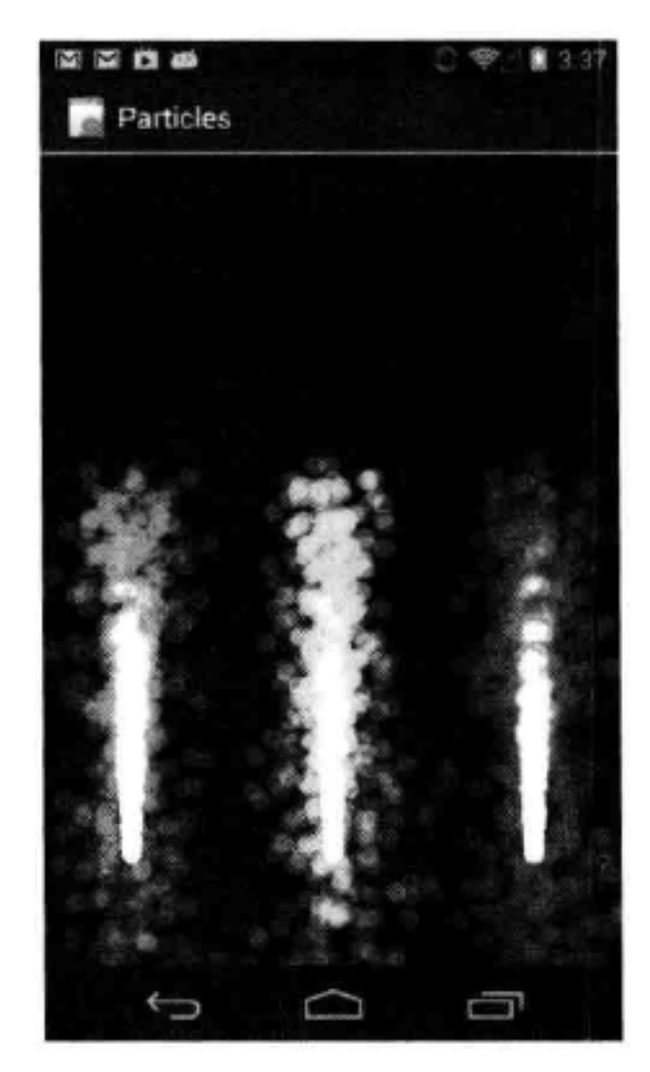


图 10-7 被绘制为圆的粒子

用这种方法把一个点绘制为圆，其开销有点大，但是它能工作。它的工作原理是：对于每个片段，我们使用勾股定理计算其与圆心的距离[⊖]。如果那个距离大于 0.5 单位的半径，那么当前的片段就不是圆的一部分，我们还使用了特殊的关键词“discard”告诉 OpenGL 丢掉这个片段；否则，我们还像以前一样绘制这个片段。

让我们再次运行这个应用；它看上去应该如图 10-7 所示。

10.8 把每个点绘制为一个精灵

我们刚刚学过的技术是可行的，但是有时候纹理更有效。使用相同的 gl_PointCoord 和

⊖ http://en.wikipedia.org/wiki/Pythagorean_theorem

纹理，实际上可以把每个点绘制为一个点精灵。对于每个粒子，我们将改动粒子着色器，让它使用图 10-8 所示的纹理。

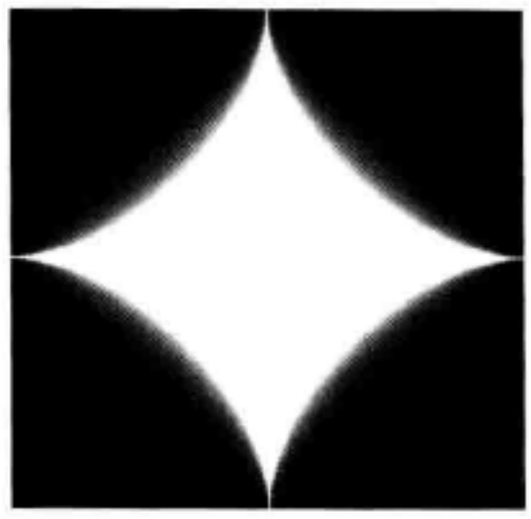


图 10-8　粒子使用的纹理

这个纹理（particle_texture.png）及其他后续章节中用到的所有纹理都可以从本书的主页上下载。我推荐把它们存储在项目的“/res/drawable-nodpi/”文件夹中[⊖]。

要实现本项目中的纹理，要使用在第 7 章学到的知识。首先更新片段着色器，加入如下 uniform：

Particles/res/raw/particle_fragment_shader.glsl

```
uniform sampler2D u_TextureUnit;
```

去掉我们在前一节加入的画圆的逻辑，按如下代码更新 gl_FragColor 的赋值语句：

Particles/res/raw/particle_fragment_shader.glsl

```
gl_FragColor = vec4(v_Color / v_ElapsedTime, 1.0)
             * texture2D(u_TextureUnit, gl_PointCoord);
```

它会使用 gl_PointCoord 作为纹理坐标在每个点上绘制一个纹理。纹理的颜色会与点的颜色相乘，这样就可以用与以前一样的方式将这些点染上颜色。

我们现在需要给 ParticleShaderProgram 加入新的 uniform。首先加入如下成员变量：

Particles/src/com/particles/android/programs/ParticleShaderProgram.java

```
private final int uTextureUnitLocation;
```

在构造函数的结尾处加入如下代码：

Particles/src/com/particles/android/programs/ParticleShaderProgram.java

```
uTextureUnitLocation = glGetUniformLocation(program, U_TEXTURE_UNIT);
```

我们需要把 setUniforms() 的签名更新为如下代码：

Particles/src/com/particles/android/programs/ParticleShaderProgram.java

```
public void setUniforms(float[] matrix, float elapsedTime, int textureId) {
```

我们也需要在 setUniforms() 的结尾处加入如下代码：

Particles/src/com/particles/android/programs/ParticleShaderProgram.java

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textureId);
glUniform1i(uTextureUnitLocation, 0);
```

关于纹理绑定，在 7.5.1 节中的“设置 uniform 并返回属性位置”一节中解释了更多的细节。

我们只需要加载这个新的纹理，就准备好运行这个应用程序了。打开 ParticlesRenderer，并加入下面的新成员变量：

Particles/src/com/particles/android/ParticlesRenderer.java

```
private int texture;
```

在 onSurfaceCreated() 的结尾处加入如下代码：

⊖ http://pragprog.com/book/kbogla

```
Particles/src/com/particles/android/ParticlesRenderer.java
texture = TextureHelper.loadTexture(context, R.drawable.particle_texture);
```

现在我们可以在 onDrawFrame() 中更新调用 particleProgram.setUniforms() 了，如下代码所示：

```
Particles/src/com/particles/android/ParticlesRenderer.java
particleProgram.setUniforms(viewProjectionMatrix, currentTime, texture);
```

再运行这个应用程序；它们看起来应该与图 10-9 类似。

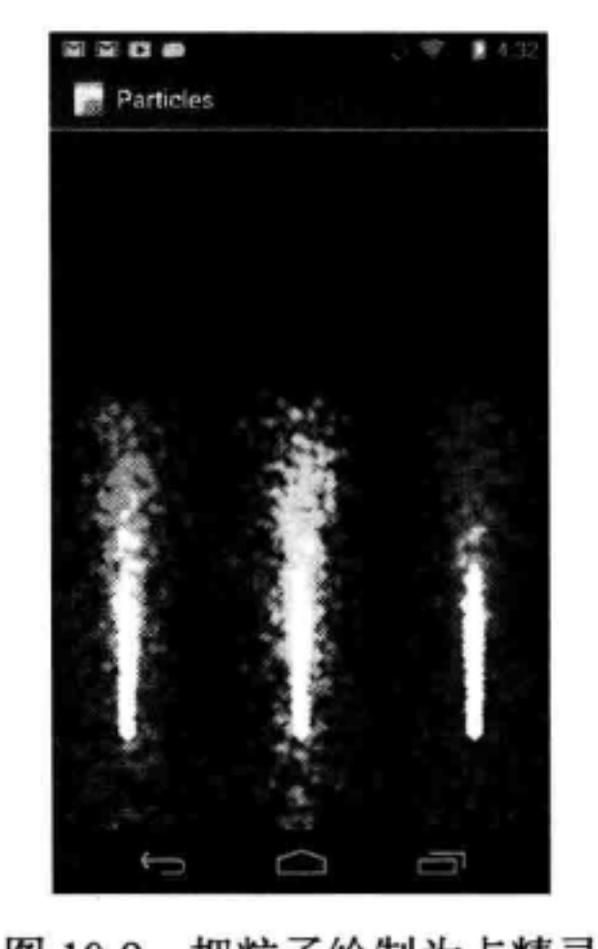


图 10-9 把粒子绘制为点精灵

10.9 小结

我们已经讲述了粒子系统的一些基本内容，但事实上我们只触及了一些皮毛而已。粒子可以实现很多效果，包括逼真的火焰、雨和雪等。我们学习了如何用 OpenGL 的 GL_POINTS 模式绘制粒子，我们还进一步讨论了如何用 gl_PointCoord 及 discard 关键词自定义点的形状；也可以使用纹理更进一步定制点的外形。有了这些能力，我们可以实现一些相当优美的效果。

10.10 练习

既然我们已经讲述了粒子系统的基础，让我们更进一步，创建一些在空中炸开的烟花。基于你实现这个效果所用的方法，你的应用可能看起来如图 10-10 所示。

当你做这个练习时，你要考虑如下几个有挑战性的问题：

- 怎样创建那些烟花的轨迹呢？你要在 Java 代码中更新粒子的位置，并用那个位置创建新的粒子吗？或者你能修改粒子的创建时间，并用它创造出这个效果么？
- 需要权衡粒子的数量、大小以及设备的能力。在某一时刻，太多的粒子会拖慢帧速率。
- 这个烟花应该创造烟雾的效果么？

图 10-10 一个粒子烟花的例子

作为一个研究性的练习，多阅读一些关于粒子系统及如何在 CPU 和 GPU 上生成粒子的内容。使用你喜欢的搜索引擎，查阅这些粒子如何应用于许多不同的效果，包括火焰、雨、雪和烟雾。

当你准备好了，我们将向下一章出发！

第 11 章 Chapter 11

增加天空盒

我们曾从一个漆黑的虚空开始，接下来，给那个场景添加三个粒子喷泉从而加入一些可见的内容。然而，我们的世界大部分还是漆黑、虚无的。我们的喷泉漂浮在黑暗之中，没有任何东西能防止那些粒子落入深渊。

首先，我们要学习如何为场景加入一个背景，它有助于提供视觉方面的感受。许多游戏和壁纸都使用二维艺术和三维技术的结合，做出它们自己的背景，在本章中，我们将学习如何使用天空盒（skybox）把喷泉置于一个天空的背景下，这是一种可以用来添加 360 度全景的技术。这些天空盒在 20 世纪 90 年代后期的许多游戏中首次出现，直至今天，它们还在继续使用。它们也在游戏世界之外被大量使用；当使用 Google 的街景（street view）时，我们可以看到 360 度的全景，这只是其中一个例子。

让我们看一下本章的开发计划：

- 我们要学习如何使用立方体贴图（cube mapping）定义一个天空盒，用这个方法可以把 6 个不同的纹理一起拼接到一个立方体上。
- 我们还会第一次接触索引（index）数组，这是减少顶点重复的一个方法，它只把那些独一无二的顶点存储在一个顶点数组中，并用指向那些顶点的索引值把它们联系在一起，这就避免了顶点数据的重复。索引数组可以减少内存的使用，当有很多顶点可以重用时，它也可以提高性能。

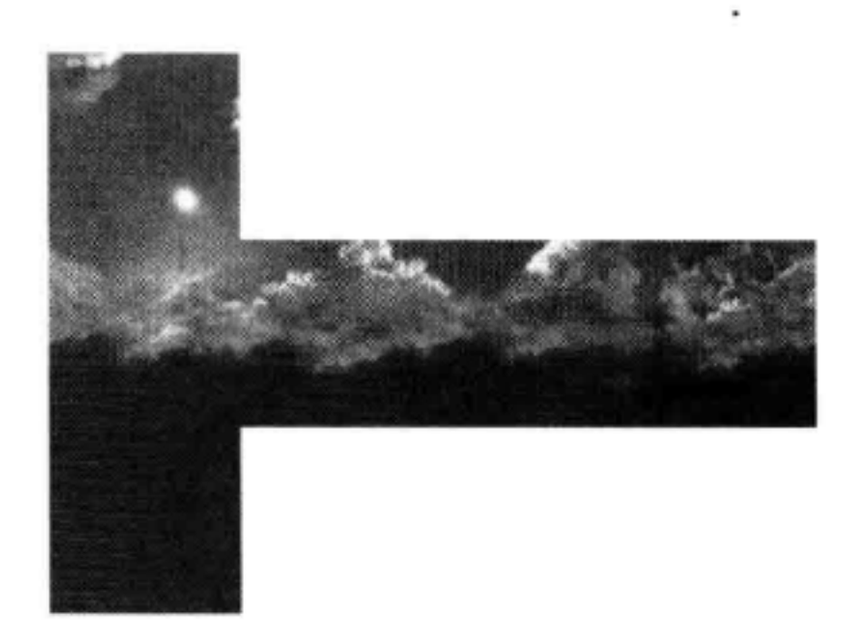
我们要以上一章的项目为基础。把它复制到一个新项目，并命名为“Skybox”。

11.1 创建天空盒

天空盒是表达三维全景的一个方法，不管你的头转向哪边，这个全景在任何方向上都

能被看见。渲染天空盒最经典的方法之一是使用一个环绕观察者的立方体，并在其每个面上都附上一个细致的纹理；这也称为立方体贴图。在绘制天空盒的时候，我们只需要保证它出现在场景中所有其他物体的后面。

当我们使用天空盒的时候，用一个立方体表示天空，因此，我们通常会期望天空就像一个巨大的立方体，它的每个面之间都有非常明显的连接处。然而，我们有可以使立方体的边缘消失的技巧：立方体的每个面都是用 90 度视野和平面投影拍摄的一幅画，我们期望这个投影与 OpenGL 本身用的是同一类型[⊖]。我们把观察者置于立方体的中间，并使用一个常规视野的透视投影，此时，一切都排列得恰到好处，立方体的边缘消失了，我们得到了一个无缝的全景图，如右图所示。



尽管这些图像本身是用 90 度视野创建的，但 OpenGL 场景自身可以使用的角度范围很广，比如 45 度或者 60 度。因为 OpenGL 透视投影的直线性质会引起边缘周围的扭曲程度增加[⊖]，只要不使用过大的角度，这个技巧就能工作得很好。

在下面的图中，我们可以看到一个例子，一个立方体的六个面的贴图彼此相邻排列，它显示了每个面如何融入其相邻的面。

天空盒技术的优点和缺点

尽管立方体贴图十分适合于实现一个简单的天空盒，但它也有简单所带来的一些缺点。如果天空盒是从立方体的中心被渲染的，这个效果才会很好，因此，对于天空盒中的任何部分，不管它们离得多远，从观察者的角度来说，它们总是处于同样的距离。因为天空盒是由预先渲染的纹理组成的，这个场景也必然是静态的，其中的云并不移动。根据观察者的视野的不同，要想让这些立方体贴图的纹理在屏幕上看起来不错，它们也许需要非常高的分辨率，这会消耗很多纹理内存和带宽。

许多现代的游戏和应用程序通常会变通地解决这些限制，它们使用自己的动态元素和云的独立的三维场景实现传统的天空盒技术。这个独立的三维场景还是在所有其他物体后面被渲染出来，但是相机可以围绕这个独立的场景移动，而且场景内部的元素可以有动画效果，给观察者处于一个巨大的三维世界中的幻象。

要在场景中创建天空盒，我们要把立方体的每个面都存储在一个单独的纹理中，然后，告诉 OpenGL 把这些纹理拼接成立方体贴图，并把它应用在一个立方体上。当 OpenGL 使

⊖ http://en.wikipedia.org/wiki/Planar_projection

⊖ http://en.wikipedia.org/wiki/Rectilinear_lens

用线性插值渲染这个立方体时，许多 GPU 实际上会把相邻面之间的纹理单元混合起来，以使立方体的边缘看不出缝隙。事实上，OpenGL ES 3.0 版本保证支持无缝立方体贴图。

11.2 把立方体贴图加载到 OpenGL 中

在随附的在线资源中，你会找到本章的立方体贴图的例子，在“/res/drawable-nodpi/”文件夹中有六个独立的纹理，其中每个都对应立方体的一个面。让我们给 TextureHelper 类加入一个新方法，把这些纹理加载到一个 OpenGL 的立方体贴图上。这个方法被称为 loadCubeMap()，让我们从下面的代码开始：

Skybox/src/com/particles/android/util/TextureHelper.java

```
public static int loadCubeMap(Context context, int[] cubeResources) {
    final int[] textureObjectIds = new int[1];
    glGenTextures(1, textureObjectIds, 0);

    if (textureObjectIds[0] == 0) {
        if (LoggerConfig.ON) {
            Log.w(TAG, "Could not generate a new OpenGL texture object.");
        }
        return 0;
    }
    final BitmapFactory.Options options = new BitmapFactory.Options();
    options.inScaled = false;
    final Bitmap[] cubeBitmaps = new Bitmap[6];
```

当调用这个方法时，我们传递进去 6 个图像资源，其中每一个都对应立方体的一个面。这个顺序有很大影响，因此我们将以一个标准顺序使用这些图像，它的顺序可以在这个方法的注释中记录下来。开始，就像加载正常的纹理一样，我们创建了一个 OpenGL 纹理对象。当我们从资源中加载图像并把它们传输给 OpenGL 时，我们用了 6 个位图对象临时在内存中保存这些图像资源数据。

让我们把这些图像加载进位图数组：

Skybox/src/com/particles/android/util/TextureHelper.java

```
for (int i = 0; i < 6; i++) {
    cubeBitmaps[i] =
        BitmapFactory.decodeResource(context.getResources(),
            cubeResources[i], options);

    if (cubeBitmaps[i] == null) {
        if (LoggerConfig.ON) {
            Log.w(TAG, "Resource ID " + cubeResources[i]
                + " could not be decoded.");
        }
        glDeleteTextures(1, textureObjectIds, 0);
        return 0;
    }
}
```

我们把所有 6 个图像都解码到内存中，加载过程中要确保每次加载都成功了。下一步，

我们要配置纹理过滤器：

Skybox/src/com/particles/android/util/TextureHelper.java

```
glBindTexture(GL_TEXTURE_CUBE_MAP, textureObjectIds[0]);

glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

因为每个立方体贴图的纹理都总是从同一个视点被观察，不大有必要使用 7.2.2 节中所描述的 MIP 贴图技术，所以我们只使用常规的双线性过滤，也能节省纹理内存。如果立方体贴图的分辨率比设备分辨率高很多，在把纹理加载到 OpenGL 之前，我们也可以把它们都缩小。

加入如下代码，把每张图像与其对应的立方体贴图的面关联起来：

Skybox/src/com/particles/android/util/TextureHelper.java

```
texImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, cubeBitmaps[0], 0);
texImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, cubeBitmaps[1], 0);

texImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, cubeBitmaps[2], 0);
texImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, cubeBitmaps[3], 0);

texImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, cubeBitmaps[4], 0);
texImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, cubeBitmaps[5], 0);
```

当我们调用这个方法时，我们要以左、右、下、上、前和后的顺序传递立方体的面。只要这个方法的调用者也按同样的顺序传递那些面，这里就没有问题。

我们首先把左面和右面的纹理贴到 x 轴的负面和正面，把下面和上面的纹理贴到 y 轴的负面和正面，再把前面的纹理贴到 z 轴的负面，后面的纹理贴到 z 轴的正面，我们似乎是在使用左手坐标系统。立方体贴图的惯例是：在立方体内部使用左手坐标系统，而在立方体外部使用右手坐标系统[⊖]。当我们渲染天空盒时，这很重要，如果我们搞混了这两个惯例，天空盒看起来就像翻转了 z 轴。

让我们用下面的代码完成这个方法：

Skybox/src/com/particles/android/util/TextureHelper.java

```
    glBindTexture(GL_TEXTURE_2D, 0);

    for (Bitmap bitmap : cubeBitmaps) {
        bitmap.recycle();
    }

    return textureObjectIds[0];
}
```

我们与纹理解除了绑定，回收了所有位图对象，并返回 OpenGL 的纹理对象 ID 给调用者。

当处理大的立方体贴图纹理时，这种加载纹理的方式会占用大量的内存，因此，如果你知道程序要运行在内存受限的设备上，就应该用另外一个方式，一次只加载一个位图，

⊖ http://www.opengl.org/registry/specs/ARB/texture_cube_map.txt

并把它加载到对应的立方体面上，每一个接下来的立方体面都可以重用这个位图对象。

11.3 创建立方体

下一个任务是为天空盒创建一个新的立方体对象，这需要创建我们的第一个索引数组。因为一个立方体只有 8 个相互独立的顶点，它很适合使用索引技术。每个顶点有 3 个位置分量，要存储这些顶点，我们需要 24 个浮点数。

假设我们决定每个面用 2 个三角形来绘制这个立方体，我们就总共需要 12 个三角形。每个三角形有 3 个顶点，如果我们只用一个顶点数组绘制这个立方体，我们将需要 36 个顶点或者 108 个浮点数，其中很多数据是重复的。通过使用一个索引数组，就不用重复所有的顶点数据了。相反，只需要重复那些索引值。这允许我们减少数据的整体大小。

让我们在 com.particles.android.objects 下创建一个名为“Skybox”的新类，在其内部用下面的代码作为开始：

Skybox/src/com/particles/android/objects/Skybox.java

```
private static final int POSITION_COMPONENT_COUNT = 3;
private final VertexArray vertexArray;
private final ByteBuffer indexArray;

public Skybox() {
    // Create a unit cube.
    vertexArray = new VertexArray(new float[] {
        -1,  1,  1,     // (0) Top-left near
         1,  1,  1,     // (1) Top-right near
        -1, -1,  1,     // (2) Bottom-left near
         1, -1,  1,     // (3) Bottom-right near
        -1,  1, -1,     // (4) Top-left far
         1,  1, -1,     // (5) Top-right far
        -1, -1, -1,     // (6) Bottom-left far
         1, -1, -1      // (7) Bottom-right far
    });
```

这个 VertexArray 保存我们的顶点数组，不久我们将使用 ByteBuffer 保存索引数组。用下面的代码完成这个构造函数：

Skybox/src/com/particles/android/objects/Skybox.java

```
    indexArray =  ByteBuffer.allocateDirect(6 * 6)
            .put(new byte[] {
                // Front
                1, 3, 0,
                0, 3, 2,

                // Back
                4, 6, 5,
                5, 6, 7,

                // Left
                0, 2, 4,
                4, 2, 6,
```

```
            // Right
            5, 7, 1,
            1, 7, 3,

            // Top
            5, 1, 4,
            4, 1, 0,

            // Bottom
            6, 2, 7,
            7, 2, 3
        });
    indexArray.position(0);
}
```

这个索引数组使用索引偏移值指向每个顶点。比如，0 指向顶点数组中的第一个顶点，且 1 指向第二个顶点。通过这个索引数组，我们把所有顶点分别绑定成三角形组，每个组有立方体上每个面的 2 个三角形。

让我们只研究一下这个立方体的前面，仔细看看有什么不同：

Skybox/src/com/particles/android/objects/Skybox.java

```
1, 3, 0,
0, 3, 2,
```

现在，让我们看看，如果只使用顶点数组，应该如何定义这个面：

```
 1,  1,  1,          // Top-right near
 1, -1,  1,          // Bottom-right near
-1,  1,  1,          // Top-left near
-1,  1,  1,          // Top-left near
 1, -1,  1,          // Bottom-right near
-1, -1,  1,          // Bottom-left near
```

通过一个索引数组，我们可以用位置指向每个顶点，而不用一遍又一遍地重复同一个顶点数据。

加入如下代码来完成这个类：

Skybox/src/com/particles/android/objects/Skybox.java

```
public void bindData(SkyboxShaderProgram skyboxProgram) {
    vertexArray.setVertexAttribPointer(0,
        skyboxProgram.getPositionAttributeLocation(),
        POSITION_COMPONENT_COUNT, 0);
}

public void draw() {
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, indexArray);
}
```

bindData() 方法是标准的，我们只剩下要定义天空盒的着色器和编译着色器的类还没有完成了。要绘制立方体，我们需要调用 glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, indices)，它告诉 OpenGL 绘制我们在 bindData() 中绑定的顶点，它使用了 indices 所定义的索引数组，并把这个数组解释为无符号字节数（unsigned byte）。在 OpenGL ES 2 中，indices 需要是无符号字节数（范围为 0 ～ 255 的 8 位整型数）或者无

符号短整型数（范围为 0 ～ 65535 的 16 位整型数）。

之前已经把索引数组定义为 ByteBuffer 了，因此我们要告诉 OpenGL 把这个数据理解为无符号字节数据流。Java 的字节类型（byte）实际上是有符号的（signed），意味着它的范围是从 −128 到 127，但是只要我们一直使用这个范围的正值部分，我们就不会碰到问题。

11.4 增加天空盒着色器程序

让我们继续为天空盒创建一个顶点着色器。在你的“/raw”资源文件夹中创建一个新的文件，命名为“skybox_vertex_shader.glsl”，并加入如下内容：

Skybox/res/raw/skybox_vertex_shader.glsl

```
uniform mat4 u_Matrix;
attribute vec3 a_Position;
varying vec3 v_Position;

void main()
{
    v_Position = a_Position;
    v_Position.z = -v_Position.z;

    gl_Position = u_Matrix * vec4(a_Position, 1.0);
    gl_Position = gl_Position.xyww;
}
```

如 main() 内第一行代码所示，我们首先把顶点的位置传递给片段着色器，接着在下一行反转其 z 分量；这个传递给片段着色器的位置就是立方体上每个面之间将被插值的位置，这使我们以后可以使用这个位置查看天空盒的纹理上正确的部分。其 z 分量被反转了，使得我们可以把世界的右手坐标空间转换为天空盒所期望的左手坐标空间。如果我们跳过这一步，天空盒仍然能正常工作，但是它的纹理看上去会是反的。

通过用 a_Position 乘以矩阵把那个位置投影到剪裁空间坐标之后，要用下面的代码把其 z 分量设置为与其 w 分量相等的值：

Skybox/res/raw/skybox_vertex_shader.glsl

```
gl_Position = gl_Position.xyww;
```

这是一种技巧，它确保天空盒的每一部分都将位于归一化设备坐标的远平面上以及场景中的其他一切后面。这个技巧能够奏效，是因为透视除法把一切都除以 w，并且 w 除以它自己，结果等于 1。透视除法之后，z 最终就在值为 1 的远平面上了。

这个技巧目前可能看起来没有必要，因为，如果我们想要天空盒出现在其他一切物体的后面，可以先把它画出来，然后再在它上面绘制其他的物体。但这个技巧的背后还有性能的考虑，我们会在 12.5.1 节中讲述更多的细节。

继续编写片段着色器。加入一个名为“skybox_fragment_shader.glsl”的新文件，并加入如下代码：

有符号与无符号数据类型之间的转换

有时候，使用有符号数据类型的取值范围是不够用的。当我们使用无符号数时，我们可以存储从 0 到 255 的数值。然而，Java 通常只让我们用 −128 到 127 范围内的数值，255 绝对在这个范围之外了。尽管我们不能写这样的代码“byte b = 255;”，但还是有方法欺骗 Java，让它把这个数值解释为一个有符号数。

让我们看一下 255 的二进制值：在二进制中，我们会把这个数值表示为“11111111”。我们只需要找到一种方法，告诉 Java 把这些位放到其字节中。事实证明，我们可以做到这一点，我们可以使用 Java 的位掩码技术屏蔽掉数值 255 的最后 8 位，看看下面的代码：

```
byte b = (byte) (255 & 0xff);
```

这个方法有效，因为 Java 会把数字 255 解释为一个 32 位的整型值，这个整型值足以容纳数值 255。然而，一旦我们把 255 赋值给字节变量，Java 实际上会把这个字节变量的值当作 −1，而不是 255，因为对 Java 来说，字节是有符号的，它会把这个值用补码表示[⊖]。然而，OpenGL 或任何其他使用无符号数值的地方都会把这个字节变量的值看作 255。

要把这个值在 Java 中读回来，我们不能直接就去读它，因为 Java 把它当作 −1，我们需要较大的数据类型来容纳这个结果。举个例子，我们可以用如下的代码做这个转换：

```
short s = (short) (b & 0xff);
```

我们不能只使用“short s = b;”，因为 Java 会做符号扩展（sign extension），s 的值还是 −1。通过告诉 Java 屏蔽最后 8 位，我们就隐式地把它 b 转换为一个整型值，然后，Java 会正确地把这个整型值的最后 8 位解释为 255。跟我们期望的一样，s 也被设为 255 了。

Skybox/res/raw/skybox_fragment_shader.glsl

```
precision mediump float;

uniform samplerCube u_TextureUnit;
varying vec3 v_Position;

void main()
{
        gl_FragColor = textureCube(u_TextureUnit, v_Position);
}
```

为了使用立方体纹理绘制这个天空盒，我们调用 textureCube() 时，把那个被插值的立方体面的位置作为那个片段的纹理坐标。

让我们加入相匹配的 Java 类来封装这个着色器程序，在 com.particles.android.programs

⊖ http://en.wikipedia.org/wiki/Two's_complement

包中加入一个名为“SkyboxShaderProgram”的新类，这个类继承自 ShaderProgram，并在其中加入如下代码：

Skybox/src/com/particles/android/programs/SkyboxShaderProgram.java

```
private final int uMatrixLocation;
private final int uTextureUnitLocation;
private final int aPositionLocation;

public SkyboxShaderProgram(Context context) {
    super(context, R.raw.skybox_vertex_shader,
        R.raw.skybox_fragment_shader);

    uMatrixLocation = glGetUniformLocation(program, U_MATRIX);
    uTextureUnitLocation = glGetUniformLocation(program, U_TEXTURE_UNIT);
    aPositionLocation = glGetAttribLocation(program, A_POSITION);
}

public void setUniforms(float[] matrix, int textureId) {
    glUniformMatrix4fv(uMatrixLocation, 1, false, matrix, 0);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureId);
    glUniform1i(uTextureUnitLocation, 0);
}

public int getPositionAttributeLocation() {
    return aPositionLocation;
}
```

到目前为止，我们已经很熟悉这个着色器程序封装的类了，它也非常简单。因为我们使用了一个立方体贴图纹理，所以用 GL_TEXTURE_CUBE_MAP 绑定这个纹理。

11.5　在场景中加入天空盒

既然我们已经有了立方体模型，也写好了着色器代码，让我们给这个场景加入天空盒。打开 ParticlesRenderer，并在类的顶部加入如下成员变量：

Skybox/src/com/particles/android/ParticlesRenderer.java

```
private SkyboxShaderProgram skyboxProgram;
private Skybox skybox;
private int skyboxTexture;
```

同时，把已经存在的那个成员变量“texture”重命名为“particleTexture”。加入任何遗漏的导入，然后，在 onSurfaceCreated() 中，用如下代码初始化这些新的变量：

Skybox/src/com/particles/android/ParticlesRenderer.java

```
skyboxProgram = new SkyboxShaderProgram(context);
skybox = new Skybox();
skyboxTexture = TextureHelper.loadCubeMap(context,
    new int[] { R.drawable.left, R.drawable.right,
                R.drawable.bottom, R.drawable.top,
                R.drawable.front, R.drawable.back});
```

因为使用了天空盒，我们不想把平移矩阵应用到这个场景上，也不想把它应用到天空盒上。出于这个原因，我们需要为天空盒和粒子使用一个不同的矩阵，因此，去掉 onSurfaceChanged() 中的 perspectiveM() 调用之后的所有代码行；相反，我们会在 onDrawFrame() 中设置这些矩阵。把 onDrawFrame() 更新为如下代码：

Skybox/src/com/particles/android/ParticlesRenderer.java

```
@Override
public void onDrawFrame(GL10 glUnused) {
    glClear(GL_COLOR_BUFFER_BIT);
    drawSkybox();
    drawParticles();
}
```

首先画出天空盒，然后在天空盒上绘制粒子。让我们加入绘制天空盒的方法：

Skybox/src/com/particles/android/ParticlesRenderer.java

```
private void drawSkybox() {
    setIdentityM(viewMatrix, 0);
    multiplyMM(viewProjectionMatrix, 0, projectionMatrix, 0, viewMatrix, 0);
    skyboxProgram.useProgram();
    skyboxProgram.setUniforms(viewProjectionMatrix, skyboxTexture);
    skybox.bindData(skyboxProgram);
    skybox.draw();
}
```

我们将以 (0, 0, 0) 为中心绘制天空盒，以使我们站在天空盒中间观察，从视觉上看，一切都是正确的。

让我们加入绘制粒子的代码：

Skybox/src/com/particles/android/ParticlesRenderer.java

```
private void drawParticles() {
    float currentTime = (System.nanoTime() - globalStartTime) / 1000000000f;

    redParticleShooter.addParticles(particleSystem, currentTime, 1);
    greenParticleShooter.addParticles(particleSystem, currentTime, 1);
    blueParticleShooter.addParticles(particleSystem, currentTime, 1);

    setIdentityM(viewMatrix, 0);
    translateM(viewMatrix, 0, 0f, -1.5f, -5f);
    multiplyMM(viewProjectionMatrix, 0, projectionMatrix, 0, viewMatrix, 0);

    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ONE);

    particleProgram.useProgram();
    particleProgram.setUniforms(viewProjectionMatrix, currentTime, particleTexture);
    particleSystem.bindData(particleProgram);
    particleSystem.draw();

    glDisable(GL_BLEND);
}
```

这与上一章是相似的，粒子喷泉被移到可视的距离内，只是现在是在 drawParticles 内部更新矩阵，并在这个方法内打开和关闭混合功能。之所以这样做，是因为我们不想在绘制天空盒的时候打开混合功能。

因为只有绘制粒子时才启用和停用混合功能，所以把 onSurfaceCreated() 中的 glEnable (GL_BLEND) 和 glBlendFunc() 调用去掉。

让我们测试一下这个新的天空盒，看看怎么样。如果一切顺利，它看起来应该如图 11-1 所示。

我们现在有了一个暴风雨天空的背景衬托这个场景了！看到那些粒子喷泉飘浮在空中，还是有点奇怪，我们将在下章中通过添加一些地形来解决这一问题。

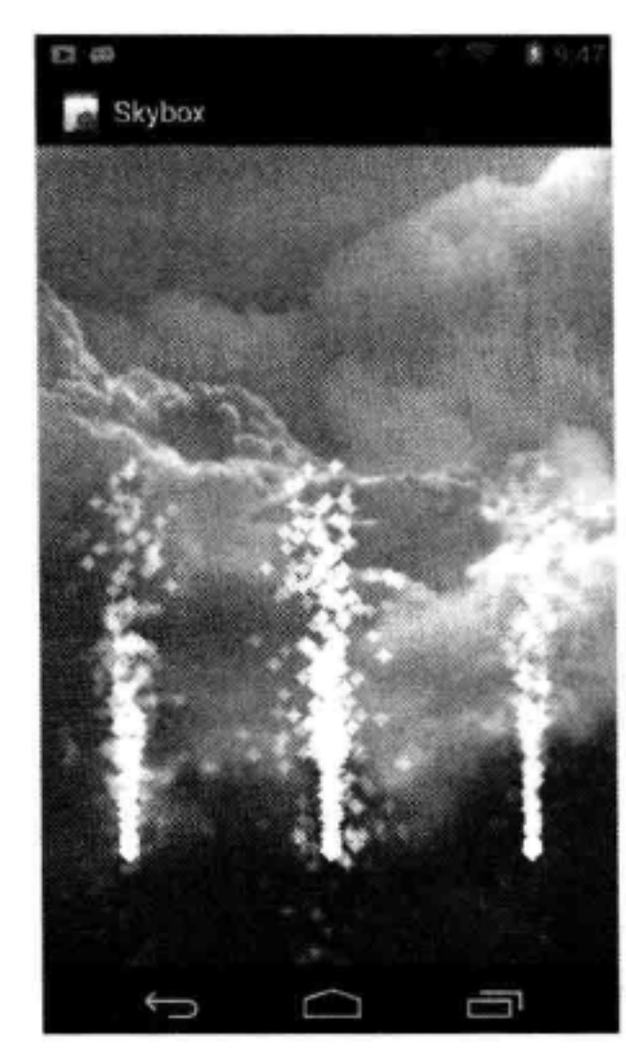


图 11-1　一个暴风雨的天空盒

11.6　围绕场景移动相机

当前，我们只能在屏幕上看到很小一部分的天空盒。如果我们能来回移动并能看到天空盒的其余部分，是不是很好呢？

通过监听触控事件，并用那些触控事件一起旋转天空盒和场景，我们可以很容易地实现。

打开 ParticlesActivity，在 setContentView() 调用之前加入如下代码：

Skybox/src/com/particles/android/ParticlesActivity.java

```
glSurfaceView.setOnTouchListener(new OnTouchListener() {
    float previousX, previousY;

    @Override
    public boolean onTouch(View v, MotionEvent event) {
        if (event != null) {
            if (event.getAction() == MotionEvent.ACTION_DOWN) {
                previousX = event.getX();
                previousY = event.getY();
            } else if (event.getAction() == MotionEvent.ACTION_MOVE) {
                final float deltaX = event.getX() - previousX;
                final float deltaY = event.getY() - previousY;
               previousX = event.getX();
               previousY = event.getY();

               glSurfaceView.queueEvent(new Runnable() {
                   @Override
                   public void run() {
                       particlesRenderer.handleTouchDrag(
                           deltaX, deltaY);
                   }
               });
            }

            return true;
        } else {
            return false;
        }
    }
});
```

这段代码定义了一个触控监听器，它会测量你的手指在每个连续的 onTouch() 调用之间滑动了多远。当你第一次触摸屏幕时，当前触摸的位置就被记录在 previousX 和 previousY 中。

当你在屏幕上滑动手指时，你会得到一串拖动事件，并且每次拖动都会这样，首先会计算新位置与旧位置的差，并把它存储到 deltaX 和 deltaY 中，然后你要更新 previousX 和 previousY。通过调用 handleTouchDrag()，这个差值会被传递给粒子渲染器。我们需要去创建这个方法，因此，回到 ParticlesRenderer 类，并在类中加入如下代码：

Skybox/src/com/particles/android/ParticlesRenderer.java

```
private float xRotation, yRotation;

public void handleTouchDrag(float deltaX, float deltaY) {
    xRotation += deltaX / 16f;
    yRotation += deltaY / 16f;

    if (yRotation < -90) {
        yRotation = -90;
    } else if (yRotation > 90) {
        yRotation = 90;
    }
}
```

这个方法会采用用户在每个方向上拖动的距离，并把它加到 xRotation 和 yRotation 上，这两个变量表示以度为单位的旋转。我们不想让触控过于灵敏，所以用 16 缩减了拖动的效果，我们也不想上下旋转的角度过大，因此把 y 旋转限制在 +90 度和 −90 度之间。

我们把旋转的效果应用到天空盒和场景上。首先更新 drawSkybox()，用下面的代码替换当前矩阵相关的代码：

Skybox/src/com/particles/android/ParticlesRenderer.java

```
setIdentityM(viewMatrix, 0);
rotateM(viewMatrix, 0, -yRotation, 1f, 0f, 0f);
rotateM(viewMatrix, 0, -xRotation, 0f, 1f, 0f);
multiplyMM(viewProjectionMatrix, 0, projectionMatrix, 0, viewMatrix, 0);
```

现在，我们只需要旋转粒子就可以了：

Skybox/src/com/particles/android/ParticlesRenderer.java

```
setIdentityM(viewMatrix, 0);
rotateM(viewMatrix, 0, -yRotation, 1f, 0f, 0f);
rotateM(viewMatrix, 0, -xRotation, 0f, 1f, 0f);
translateM(viewMatrix, 0, 0f, -1.5f, -5f);
multiplyMM(viewProjectionMatrix, 0, projectionMatrix, 0, viewMatrix, 0);
```

这个旋转首先应用 y 轴的旋转矩阵，然后应用 x 轴的旋转矩阵，这叫“FPS 样式”旋转（FPS 代表 First Person Shooter（第一人称发射者）），向上或向下旋转总是让你向头上看或者向脚下看，向左或向右旋转总是让你围着以你的脚为中心的圆来回旋转。

现在，如果你再次运行这个应用程序，并在屏幕上滑动你的手指，你就可以来回移动相机，并看到天空盒的其他部分了，如下图所示：

11.7 小结

在本章中，我们学习了如何添加一个天空盒，它使用了一个立方体，并用立方体贴图技术把天空盒的纹理应用到这个立方体上。我们也学习了索引数组，当一个物体有很多公用的顶点时，索引数组可以减少内存的使用。

尽管我们在本章中为天空盒使用了立方体贴图，但是它也可以被用在其他效果上。一个立方体贴图也可以用作一个物体的环境贴图（environment map），它使物体看上去反光或发亮。当处理更复杂的物体和场景时，使用索引数组用会十分得心应手，但是衡量一下性能总是最好的，比较它们的结果，选择性能最好的那一个。

11.8 练习

作为你的第一个练习，给这个场景添加一个小立方体，并把立方体贴图应用到这个小立方体上。一个更高级的练习，给相机添加移动性（比如，你可以使用音量键向前或向后移动），移动整个场景，而不移动天空盒。

当你准备好了，让我们进入下一章。

Chapter 12 第 12 章

增加地形

现在，我们已经有了一个不错的背景，这个背景在地平线上有些乌云，可以给场景添加一些内容了。在本章中，我们将学习如何使用高度图（height map）给场景添加地形（terrain）。当我们实现这些内容时，我们会接触到许多 OpenGL 的新领域，并学习如何使用深度缓冲区（depth buffer）防止超量绘制，为了提高性能，我们也将学习如何在 GPU 上直接存储顶点和索引数组。

高度图是给场景添加地形的简便方法之一，可以很容易地使用常见的绘画程序生成或编辑高度图。深度缓冲区本身是 OpenGL 的一个重要部分，它帮助我们很容易地渲染比较复杂的场景，而不用太关心那个场景是怎样被拼在一起的。

本章的开发计划如下：

- 首先，我们要看一下如何创建一个高度图，并用顶点缓冲区对象和索引缓冲区对象把它加载到我们的应用程序中。
- 然后，学习剔除（culling）和深度缓冲区，这两个技术用于遮罩（occluding）隐藏的物体。

让我们继续上一章的项目，把那些代码复制到一个新项目，并命名为“Heightmap”。

12.1 创建高度图

简单来说，高度图是表示高度的一个二维地图，很像你在一个地图集中看到的地形图。一个创建高度图的简单方法是使用灰度图，亮的区域表示高地，而暗的区域表示低地。因为它只是一幅图像，我们可以用任何常见的绘图程序绘制自己的高度图，最终得到像下面

这幅图一样的图像。你甚至可以从网上下载真正的地形高度图，比如，National Evevation Dataset[⊖]。

我们将使用前面的图作为这个项目的高度图，你可以从本书的网站上下载[⊖]。把这个图像放到项目中的“/res/drawable-nodpi/”文件夹；下一步，我们将把这个高度图数据加载进来。

图 12-1 一个高度图的例子

12.2 创建顶点和索引缓冲区对象

为了加载高度图，我们要使用两个新的 OpenGL 对象：一个顶点缓冲区对象和一个索引缓冲区对象。这两个对象类似于我们在前一章中所使用的顶点数组和索引数组，只是图形驱动器可以选择把这两个对象直接放进 GPU 的内存中。对于那些一经创建就不经常变化的对象来说，比如高度图，这可以带来更好的性能。因为这些缓冲区对象并不总是比较快，所以比较一下这两个选项总是值得的。

12.2.1 创建一个顶点缓冲区对象

要加载这些缓冲区对象，我们需要加入一些配套代码。在 com.particles.android.data 包中，创建一个名为“VertexBuffer”的新类，并加入下面的成员变量和构造函数：

Heightmap/src/com/particles/android/data/VertexBuffer.java

```
private final int bufferId;

public VertexBuffer(float[] vertexData) {
    // Allocate a buffer.
    final int buffers[] = new int[1];
    glGenBuffers(buffers.length, buffers, 0);
    if (buffers[0] == 0) {
        throw new RuntimeException("Could not create a new vertex buffer object.");
    }
    bufferId = buffers[0];

    // Bind to the buffer.
    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);

    // Transfer data to native memory.
    FloatBuffer vertexArray = ByteBuffer
        .allocateDirect(vertexData.length * BYTES_PER_FLOAT)
        .order(ByteOrder.nativeOrder())
        .asFloatBuffer()
        .put(vertexData);
```

⊖ http://ned.usgs.gov/

⊖ http://pragprog.com/book/kbogla/

```
        vertexArray.position(0);

        // Transfer data from native memory to the GPU buffer.
        glBufferData(GL_ARRAY_BUFFER, vertexArray.capacity() * BYTES_PER_FLOAT,
            vertexArray, GL_STATIC_DRAW);

        // IMPORTANT: Unbind from the buffer when we're done with it.
        glBindBuffer(GL_ARRAY_BUFFER, 0);
    }
```

要把顶点数据发送到一个顶点缓冲区对象，首先用 glGenBuffers() 创建了一个新的缓冲区对象。这个方法需要传递一个数组，因此，我们创建了一个新的、只有一个元素的数组，用它来存储这个新缓冲区的 ID。然后，用 glBindbuffer() 调用绑定到这个缓冲区，传递 GL_ARRAY_BUFFER 告诉 OpenGL 这是一个顶点缓冲区。

就像我们使用 VertexArray 时所做的一样，要把数据复制到缓冲区对象，我们不得不首先把它传到本地内存中。一旦完成了，我们就可以用 glBufferData() 调用把数据传到缓冲区对象了。让我们更详细地看一下这个方法的参数（参见表 12-1）。

表 12-1 glBufferData 的参数

glBufferData (int target, int size, Buffer data, int usage)	
int target	顶点缓冲区对象应该设 GL_ARRAY_BUFFER，索引缓冲区对象应该设 GL_ELEMENT_ARRAY_BUFFER
int size	数据的大小（以字节为单位）
Buffer data	由 allocateDirect() 创建的一个缓冲区 Buffer 对象
int usage	告诉 OpenGL 对这个缓冲区对象所期望的使用模式。以下是它的可用选项： GL_STREAM_DRAW 这个对象只会被修改一次，并且不会被经常使用。 GL_STATIC_DRAW 这个对象将被修改一次，但是会经常使用。 GL_DYNAMIC_DRAW 这个对象将被修改和使用很多次。 这些只是提示，而不是限制，所以 OpenGL 可以根据需要做任何优化。大多数情况下，我们都使用 GL_STATIC_DRAW

当我们已经完成往缓冲区内加载数据的时候，要确保与缓冲区解除绑定，这通过把 0 作为缓冲区 ID 传递给 glBindBuffer() 调用完成；否则，在我们的代码中，其他地方的一些函数调用可能无法正常工作，比如 glVertexAttribPointer()。

我们也需要给 glVertexAttribPointer() 封装一下，就像之前构建 VertexArray 类时一样，加入一个名为“setVertexAttribPointer()”的方法：

Heightmap/src/com/particles/android/data/VertexBuffer.java

```
public void setVertexAttribPointer(int dataOffset, int attributeLocation,
    int componentCount, int stride) {
    glBindBuffer(GL_ARRAY_BUFFER, bufferId);
    glVertexAttribPointer(attributeLocation, componentCount, GL_FLOAT,
        false, stride, dataOffset);
```

```
    glEnableVertexAttribArray(attributeLocation);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

这里的主要区别是，在调用 glVertexAttribPointer() 之前，需要绑定到缓冲区，而且使用了一个稍有不同的 glVertexAttribPointer()，它的最后一个参数为 int 类型，而不是 Buffer 对象。这个整型参数告诉 OpenGL 当前属性对应的以字节为单位的偏移值；对于第一个属性，它可能是 0，对于其后续的属性，它就是一个指定的字节偏移值。

与以前一样，在从这个方法返回前，要确保与缓冲区解除绑定。

12.2.2 创建一个索引缓冲区对象

我们还需要一个索引缓冲区的封装类，因此，在 VertexBuffer 所在的包中，继续创建一个名为“IndexBuffer”的新类。复制 VertexBuffer 中的成员变量和构造函数，并对其做如下的改动：

- 使用 short[] 和 ShortBuffer 作为类型。
- 使用 GL_ELEMENT_ARRAY_BUFFER，而不是 GL_ARRAY_BUFFER。
- 要获得以字节为单位的大小，在 Constants 类中加入值为 2 的常量 BYTES_PER_SHORT，并在你调用 glBufferData() 时使用那个常量，而不是 BYTES_PER_FLOAT。

当我们使用它绘制时，需要使用其缓冲区 ID，为此加入下面的存取函数：

Heightmap/src/com/particles/android/data/IndexBuffer.java

```
public int getBufferId() {
    return bufferId;
}
```

既然配套代码已经到位了，让我们把高度图加载进来。

12.3 加载高度图

要把高度图加载进 OpenGL，我们需要加载图像数据，并把它转换为一组顶点，每个顶点对应一个像素。每个顶点都有一个基于其所在图像中的位置和一个基于像素亮度的高度。一旦所有的顶点都被加载进来，就可以使用索引缓冲区把它们组成 OpenGL 能绘制的三角形。

12.3.1 生成顶点数据

让我们在 com.particles.android.objects 包中创建一个新类，命名为“Heightmap”，并在类中加入如下代码：

Heightmap/src/com/particles/android/objects/Heightmap.java

```
private static final int POSITION_COMPONENT_COUNT = 3;

private final int width;
```

```
    private final int height;
    private final int numElements;
    private final VertexBuffer vertexBuffer;
    private final IndexBuffer indexBuffer;

    public Heightmap(Bitmap bitmap) {
        width = bitmap.getWidth();
        height = bitmap.getHeight();

        if (width * height > 65536) {
            throw new RuntimeException("Heightmap is too large for the index buffer.");
        }
        numElements = calculateNumElements();
        vertexBuffer = new VertexBuffer(loadBitmapData(bitmap));
        indexBuffer = new IndexBuffer(createIndexData());
    }
```

我们把一个 Android 的 Bitmap 对象传递进去，把数据加载进一个顶点缓冲区，并为那些顶点创建了一个索引缓冲区。加入 loadBitmapData() 的定义如下代码所示：

Heightmap/src/com/particles/android/objects/Heightmap.java

```
    private float[] loadBitmapData(Bitmap bitmap) {
        final int[] pixels = new int[width * height];
        bitmap.getPixels(pixels, 0, width, 0, 0, width, height);
        bitmap.recycle();

        final float[] heightmapVertices =
            new float[width * height * POSITION_COMPONENT_COUNT];
        int offset = 0;
```

为了有效地读入所有位图数据，我们首先用 getPixels() 调用提取所有的像素，然后，回收 bitmap，因为我们不需要保存它了。既然每个像素都对应一个顶点，就用那个位图的宽和高为那些顶点创建一个新的数组。

让我们加入一些代码，把位图像素转换为高度图的数据：

Heightmap/src/com/particles/android/objects/Heightmap.java

```
        for (int row = 0; row < height; row++) {
            for (int col = 0; col < width; col++) {
                final float xPosition = ((float)col / (float)(width - 1)) - 0.5f;
                final float yPosition =
                    (float)Color.red(pixels[(row * height) + col]) / (float)255;
                final float zPosition = ((float)row / (float)(height - 1)) - 0.5f;

                heightmapVertices[offset++] = xPosition;
                heightmapVertices[offset++] = yPosition;
                heightmapVertices[offset++] = zPosition;
            }
        }
        return heightmapVertices;
    }
```

要生成高度图的每一个顶点，我们首先要计算顶点的位置；高度图在每个方向上都是 1 个单位宽，且其以 x-z 平面上的位置 (0, 0) 为中心，因此，通过这些循环，位图的左上角将被映射到 (−0.5, −0.5)，右下角会被映射到 (0.5, 0.5)。

我们假定这个图像是灰度图，因此，读入其对应像素的红色分量，并把它除以 255，得到高度。一个像素值 0 对应高度 0，而一个像素值 255 对应高度 1。一旦我们计算完了位置和高度，就可以把这个新的顶点写入数组了。

在继续之前，让我们仔细地看一下这个循环。为什么一行一行地读取这个位图，并从左向右扫描每一列呢？我们为什么不一列一列地读取这个位图呢？一行一行地读取位图的原因在于其在内存中的布局方式就是这样的，当 CPU 按顺序缓存和移动数据时，它们更有效率。

记住存取像素的方式也是很重要的。当我们用 getPixels() 提取像素时，Android 给我们返回一个一维数组。那么我们怎么知道去哪里读入像素呢？可以用下面的公式计算其正确的位置：

像素偏移值 = 当前行 * 高度 + 当前列

通过这个公式，我们可以用两个循环读入一个一维数组，就好像它是一个二维位图一样。

12.3.2 生成索引数据

回到构造函数，我们通过调用 calculateNumElements() 得出需要多少个索引元素，并把其结果存入 numElements。现在，让我们继续实现这个方法：

Heightmap/src/com/particles/android/objects/Heightmap.java

```
private int calculateNumElements() {
    return (width - 1) * (height - 1) * 2 * 3;
}
```

其工作原理是，针对高度图中每 4 个顶点构成的组，会生成 2 个三角形，且每个三角形有 3 个索引，总共需要 6 个索引。通过用 (width−1) 乘以 (height−1)，计算出我们需要多少组，然后，我们只需要用组数乘以每组 2 个三角形和每个三角形的 3 个元素就得出所有的元素个数。比如，一个 3 × 3 的高度图有 (3 −1) × (3−1) = 2 × 2 = 4 个组，以及每个组需要 2 个三角形和每个三角形需要 3 个元素，总共得到 24 个元素。

用下面的代码生成这组索引：

Heightmap/src/com/particles/android/objects/Heightmap.java

```
private short[] createIndexData() {
    final short[] indexData = new short[numElements];
    int offset = 0;

    for (int row = 0; row < height - 1; row++) {
        for (int col = 0; col < width - 1; col++) {
            short topLeftIndexNum = (short) (row * width + col);
            short topRightIndexNum = (short) (row * width + col + 1);
            short bottomLeftIndexNum = (short) ((row + 1) * width + col);
            short bottomRightIndexNum = (short) ((row + 1) * width + col + 1);

            // Write out two triangles.
            indexData[offset++] = topLeftIndexNum;
            indexData[offset++] = bottomLeftIndexNum;
```

```
                indexData[offset++] = topRightIndexNum;

                indexData[offset++] = topRightIndexNum;
                indexData[offset++] = bottomLeftIndexNum;
                indexData[offset++] = bottomRightIndexNum;
            }
        }

        return indexData;
    }
```

这个方法根据需要的大小创建了一个 short 类型的数组，然后，通过行和列的循环为每 4 个顶点构成的组创建三角形。我们甚至不需要实际的像素数据就能做到这一点；我们所需要的只是宽度和高度。在 11.3 节中第一次提到了索引，这段代码遵循与其相同的模式。

如果你尝试存储一个大于 32 767 的索引值，一些有趣的事情就会发生：short 类型的转换会使这个数值回绕成一个负数。然而，因为补码的原因，当 OpenGL 把它们作为无符号值读入的时候，这些负数也会被当作正确的值（参见 11.4 节）。只要索引数组的元素数不超过 65 535，就不会有问题。

12.3.3 提示与陷阱

当使用缓冲区对象时，有几点需要注意。从技术上讲，Android 从其 2.2 版本（Froyo）开始支持 OpenGL ES 2，但不幸的是其 Java 与本地的绑定被破坏了，如果不写一个自定义的 Java 本地接口（JNI）的绑定，顶点和索引缓冲区是无法在 Java 层使用的。

好消息是这些绑定已经在 Android 的 Gingerbread 版本中修复了，在编写本书时，市场上只有 9% 的设备还在运行 Froyo 版本，所以这个问题不是什么大不了的事，没有过去那么严重。

就像 Java 的 ByteBuffers 一样，不正确地使用 OpenGL 缓冲区对象会导致本地应用程序崩溃，它很难调试。如果你的应用程序突然消失了，而且你在 Android 的日志中看到诸如“Segmentation fault”的内容，我建议你仔细检查所有涉及缓冲区的调用，尤其是 glVertexAttribPointer() 调用。

12.4 绘制高度图

既然已经把高度图加载进来了，让我们把它绘制到屏幕上。

在项目的“/res/raw”文件夹中创建一个新的文件，命名为“heightmap_vertex_shader.glsl”，并加入如下代码：

Heightmap/res/raw/heightmap_vertex_shader.glsl

```
uniform mat4 u_Matrix;
attribute vec3 a_Position;
varying vec3 v_Color;
```

```
void main()
{
    v_Color = mix(vec3(0.180, 0.467, 0.153),    // A dark green
                  vec3(0.660, 0.670, 0.680),    // A stony gray
                  a_Position.y);

    gl_Position = u_Matrix * vec4(a_Position, 1.0);
}
```

这个顶点着色器使用了一个新的着色器函数“mix()”用来在两个不同的颜色间做平滑插值。我们配置了高度图，使其高度处于 0 和 1 之间，并使用这个高度作为两个颜色之间的比例。高度图在接近底部的地方呈现绿色，在接近顶部的地方显示灰色。

加入一个匹配的名为“heightmap_fragment_shader.glsl”的片段着色器：

Heightmap/res/raw/heightmap_fragment_shader.glsl

```
precision mediump float;

varying vec3 v_Color;

void main()
{
    gl_FragColor = vec4(v_Color, 1.0);
}
```

要在 Java 中封装这个着色器，在 com.particles.android.programs 包中创建一个名为“HeightmapShaderProgram”的新类，它使用与包中其他类同样的模式。一旦完成了，回到 Heightmap，并在类的结尾处加入如下方法：

Heightmap/src/com/particles/android/objects/Heightmap.java

```
public void bindData(HeightmapShaderProgram heightmapProgram) {
    vertexBuffer.setVertexAttribPointer(0,
        heightmapProgram.getPositionAttributeLocation(),
        POSITION_COMPONENT_COUNT, 0);
}

public void draw() {
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer.getBufferId());
    glDrawElements(GL_TRIANGLES, numElements, GL_UNSIGNED_SHORT, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}
```

我们将使用 bindData() 告诉 OpenGL 当调用 draw() 时去哪里获取数据。在 draw() 中，我们告诉 OpenGL 使用索引缓冲区绘制数据。这个调用使用的 glDrawElements() 与上一章使用的那个有点不同：就像用 glVertexAttribPointer() 一样，最后一个参数使用了一个 int 类型的偏移，而不是 Buffer 对象的引用，这用来告诉 OpenGL 从哪个索引开始读取数据。

与以前一样，我们也会在使用之前绑定缓冲区，并确保在使用之后解除绑定。

12.4.1 为 ParticlesRenderer 加入高度图和新矩阵

因为所有的组件都准备好了，让我们前往 ParticlesRenderer 把它们整合在一切。首先需

要在类的顶部加入两个新的矩阵，把矩阵列表更新为如下代码：

Heightmap/src/com/particles/android/ParticlesRenderer.java
```
private final float[] modelMatrix = new float[16];
private final float[] viewMatrix = new float[16];
private final float[] viewMatrixForSkybox = new float[16];
private final float[] projectionMatrix = new float[16];

private final float[] tempMatrix = new float[16];
private final float[] modelViewProjectionMatrix = new float[16];
```

还需要为高度图和高度图着色器程序加入两个新的成员变量：

Heightmap/src/com/particles/android/ParticlesRenderer.java
```
private HeightmapShaderProgram heightmapProgram;
private Heightmap heightmap;
```

在 onSurfaceCreated() 中初始化它们：

Heightmap/src/com/particles/android/ParticlesRenderer.java
```
heightmapProgram = new HeightmapShaderProgram(context);
heightmap = new Heightmap(((BitmapDrawable)context.getResources()
    .getDrawable(R.drawable.heightmap)).getBitmap());
```

为了减少矩阵代码复制和粘贴的使用量，需要做更多的改动。为此，我们使用一个矩阵代表相机，它应用于所有的物体，再为天空盒定义一个矩阵，它只表示旋转。

在类中加入一个名为“updateViewMatrices()”的新方法，如下代码所示：

Heightmap/src/com/particles/android/ParticlesRenderer.java
```
private void updateViewMatrices() {
    setIdentityM(viewMatrix, 0);
    rotateM(viewMatrix, 0, -yRotation, 1f, 0f, 0f);
    rotateM(viewMatrix, 0, -xRotation, 0f, 1f, 0f);
    System.arraycopy(viewMatrix, 0, viewMatrixForSkybox, 0, viewMatrix.length);

    // We want the translation to apply to the regular view matrix, and not
    // the skybox.
    translateM(viewMatrix, 0, 0, -1.5f, -5f);
}
```

通过这段代码，我们就可以使用 viewMatrix 一起旋转和平移高度图和粒子了，我们也可以使用 viewMatrixForSkybox 旋转天空盒。

我们曾在 8.6.1 节介绍过矩阵的层次结构，这里也遵循同样的概念。我们要使用这个新的方法，在 handleTouchDrag() 的结尾处加入 updateViewMatrices() 调用。在 onSurfaceChanged() 中，我们也需要用另外一个 updateViewMactrices() 调用替换 perspectiveM() 调用之后的代码，如下代码所示：

Heightmap/src/com/particles/android/ParticlesRenderer.java
```
@Override
public void onSurfaceChanged(GL10 glUnused, int width, int height) {
    glViewport(0, 0, width, height);
    MatrixHelper.perspectiveM(projectionMatrix, 45, (float) width
        / (float) height, 1f, 10f);
    updateViewMatrices();
}
```

我们还需要两个新的辅助方法把这些矩阵相乘形成一个最终合并在一起的模型 – 视图投影矩阵，这两个方法分别用于绘制天空盒和其他物体。在类中加入如下两个方法：

Heightmap/src/com/particles/android/ParticlesRenderer.java

```
private void updateMvpMatrix() {
    multiplyMM(tempMatrix, 0, viewMatrix, 0, modelMatrix, 0);
    multiplyMM(modelViewProjectionMatrix, 0, projectionMatrix, 0, tempMatrix, 0);
}
private void updateMvpMatrixForSkybox() {
    multiplyMM(tempMatrix, 0, viewMatrixForSkybox, 0, modelMatrix, 0);
    multiplyMM(modelViewProjectionMatrix, 0, projectionMatrix, 0, tempMatrix, 0);
}
```

我们需要使用一个临时矩阵存储其中间结果，因为如果使用同一个矩阵既作为最终的结果又作为一个操作数会弄乱这个矩阵。现在可以用下面的代码替换掉 drawSkybox() 中现有的与矩阵相关的代码了：

Heightmap/src/com/particles/android/ParticlesRenderer.java

```
setIdentityM(modelMatrix, 0);
updateMvpMatrixForSkybox();
```

对于 drawParticles()，也用下面的代码替换掉其现有的与矩阵相关的代码：

Heightmap/src/com/particles/android/ParticlesRenderer.java

```
setIdentityM(modelMatrix, 0);
updateMvpMatrix();
```

一旦完成了这些，就可以更新天空盒和粒子的 setUniforms() 调用了，用 modelViewProjectionMatrix 替换掉那些遗漏的 viewProjectionMatrix 引用。现在要处理相机的旋转，并用视图矩阵把物体平移放进场景中，因此，我们不再需要为每个物体复制和粘贴与矩阵设置相关的代码了。通过调用 setIdentityM(modelMatrix, 0)，我们要把模型矩阵重置为标准矩阵，实际上这个标准矩阵什么都没做，因此，当我们在 updateMvpMatrix() 中把所有矩阵乘在一起时，只有视图矩阵和投影矩阵起作用。

12.4.2 绘制高度图

搞清楚了这些矩阵，让我们继续绘制高度图！

在 onDrawFrame() 中，就在调用 glClear() 之后加入调用 drawHeightmap()。加入的这个方法的内容如下：

Heightmap/src/com/particles/android/ParticlesRenderer.java

```
private void drawHeightmap() {
    setIdentityM(modelMatrix, 0);
    scaleM(modelMatrix, 0, 100f, 10f, 100f);
    updateMvpMatrix();
    heightmapProgram.useProgram();
    heightmapProgram.setUniforms(modelViewProjectionMatrix);
    heightmap.bindData(heightmapProgram);
    heightmap.draw();
}
```

因为我们不想让这个山太突兀，我们用模型矩阵使高度图在 x 和 z 方向上变宽 100 倍，而在 y 方向上只变高 10 倍。等等，我们知道着色器中的颜色插值依赖于顶点所在位置的 y 值，这不会扰乱它吗？不会的，因为着色器读入顶点位置的时间是在我们把它与矩阵相乘之前。

我们还需要更新投影矩阵给出足够的空间，因此，在 onSurfaceChanged() 中，把 perspectiveM() 的最后一个参数改为 100f；这会设置投影，以使我们可以在被远平面剪裁之前的最大 100 个单位空间上绘制东西。

12.5 遮罩被隐藏的物体

现在，一切都准备就绪了，让我们运行这个应用程序，看看会发生什么。你可能会惊讶地发现地形图竟然没有显示出来！检查一下 onDrawFrame()，看看哪里出错了：

Heightmap/src/com/particles/android/ParticlesRenderer.java

```
glClear(GL_COLOR_BUFFER_BIT);
drawHeightmap();
drawSkybox();
drawParticles();
```

我们首先绘制了高度图，然后就在其后面绘制了天空盒，这个天空盒擦除了所有之前绘制的东西。切换这两个绘图调用函数，使天空盒先画出来，然后再让高度图绘制出来，看看会发生什么？

地形图现在应该显示出来了，但是，你可能看见一些奇怪的效果，如图 12-2 所示。之所以会这样，是因为地形图本身也有重复绘制的问题：没有考虑地形的哪些部分与观察者更近，其不同的部分彼此重叠绘制了。你可能也注意到了，粒子仍然掉进了“地里”，这也不太合理了。

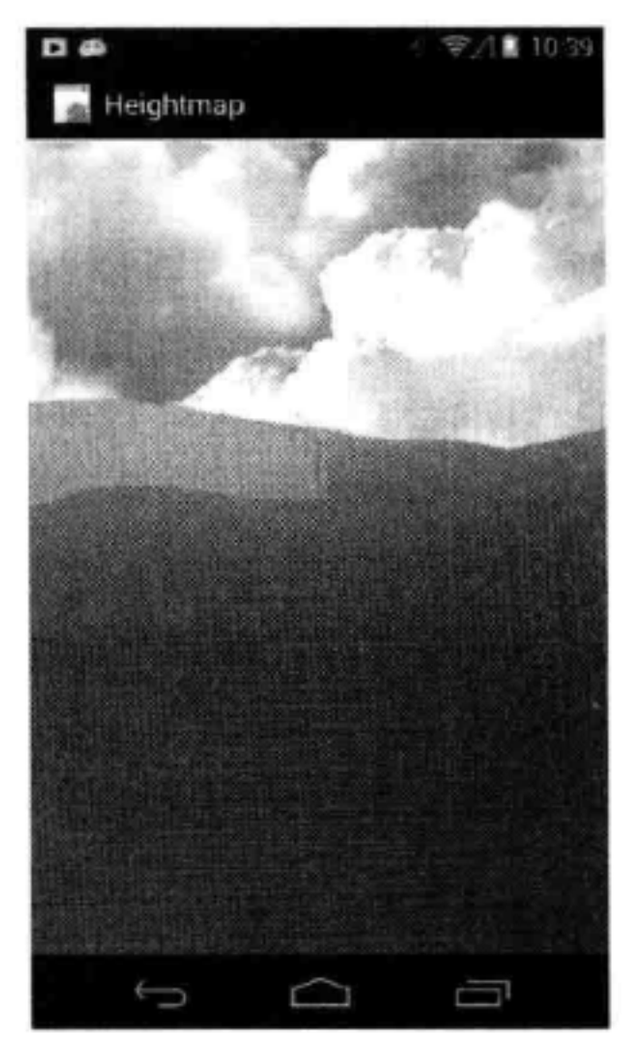


图 12-2 有奇怪效果的地形图

12.5.1 用深度缓冲区消除隐藏面

我们能否把所有的三角形排序呢？以便可以按照从后向前的顺序绘制东西，使它们看起来与期望的一样。这是一个可能的解决方案，但是会碰到两大问题：首先，绘制的顺序依赖于当前的观察点，使它的计算很复杂；其次，这个方案也浪费，因为将花费大量的时间绘制那些从来不会被看见的东西，它们会被近处的东西覆盖。

OpenGL 用深度缓冲区技术为我们提供了一个更好的解决方案，它是一个特殊的缓冲区，用于记录屏幕上每个片段的深度。当这个缓冲区打开时，OpenGL 会为每个片段执行深度测试算法：如果片段比已经存在的片段更近，就绘制它；否则，就丢掉它。

让我们反转 onDrawFrame() 的绘制顺序，以使高度图首先被绘制，然后，在 onSurfaceCreated() 内，让我们在调用 glClearColor() 后面添加调用 glEnable(GL_DEPTH_TEST)，打开深度缓冲区功能。我们也需要在 onDrawFrame() 中把 glClear() 调用更新为 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)。这告诉 OpenGL 在每个新帧上也要清空深度缓冲区。

让我们再次运行这个应用，看看能得到什么。地形现在没问题了，粒子也不再落入地里了，但是，如我们在图 12-3 中看到的，天空盒现在相当混乱。

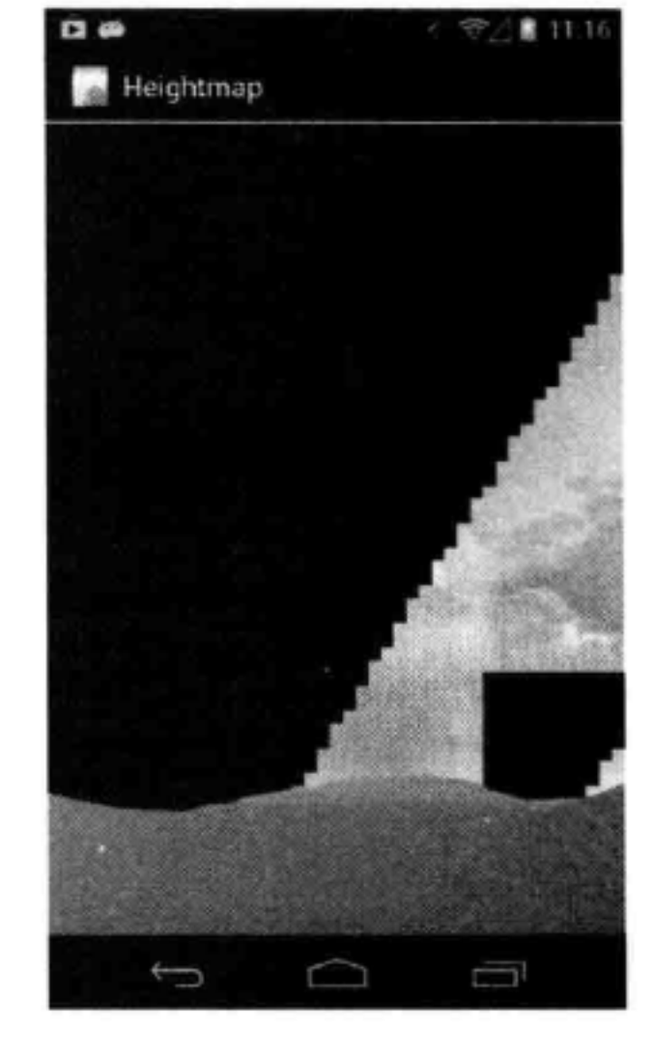


图 12-3 渲染天空盒时出现的问题

深度测试

回到 11.4 节，我们曾经配置了天空盒着色器程序，以使天空盒被绘制在远平面上。现在，深度测试功能被打开了，这也给我们带来一个问题，因为 OpenGL 默认情况下只绘制那些比其他片段更近或者比远平面更近的片段。我们根本不应该看见天空盒的任何部分，但是由于浮点数的精度问题，它还是让天空盒的某些部分显示出来了。

要修复这个问题，我们可以改变天空盒着色器，让它绘制得稍微近点，或者我们可以改变深度测试算法，让那些片段通过测试。编辑 drawSkybox()，按如下代码改变深度测试算法：

```
glDepthFunc(GL_LEQUAL);
«draw»
glDepthFunc(GL_LESS);
```

默认情况下，深度测试被设置为使用 GL_LESS，它意味着：如果新片段比任何已经存在那里的片段近或者比远平面近，就让它通过测试。我们为天空盒把 GL_LESS 改为 GL_LEQUAL，这改变了它的含义：如果新片段与已经存在那里的片段相比较近或者二者在同等的距离处，就让它通过测试。绘制完天空盒，我们把这个测试重置为默认算法，使其他一切依然按预期的方式绘制出来。

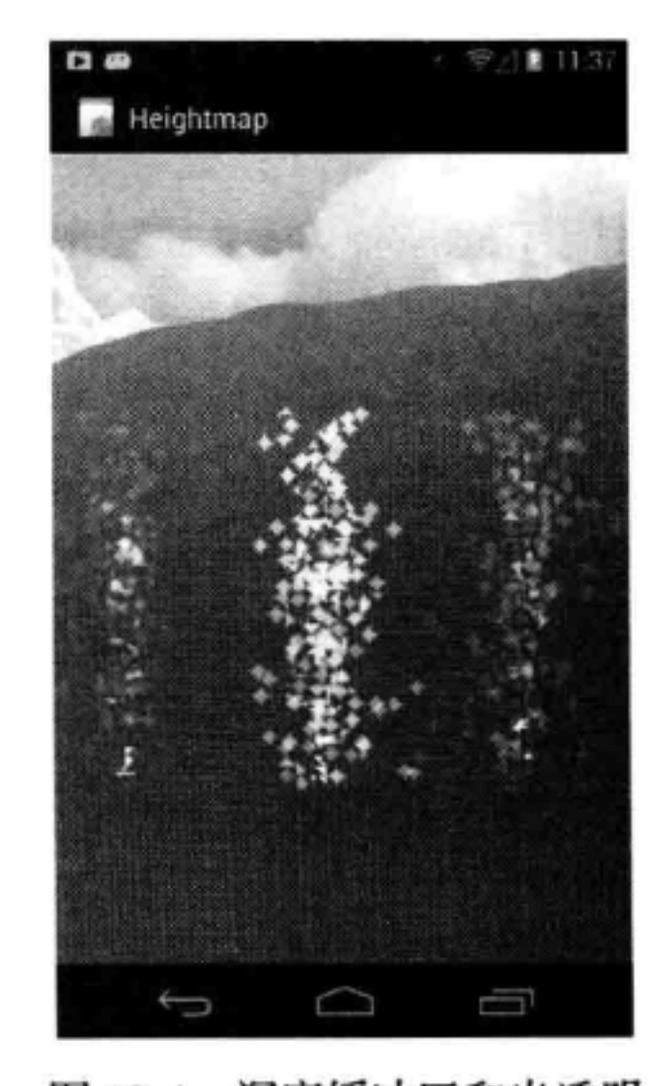


图 12-4 深度缓冲区和半透明物体的问题

深度缓冲区和半透明物体

如果我们再次运行这个应用程序，天空盒看起来应该不错了，但是，如我们在图 12-4 中所看到的，粒子还有个奇怪的问题。它们现在被地面裁剪了，而且还似乎彼此遮罩了！这不合理，因为我们希望粒子是半透明的，且相互混合。在粒子接触到地面的地方，我们需要一种方法使

它们彼此不被阻挡，同时还要裁剪它们。

在 OpenGL 中，这是可能的，在保持测试功能开启的同时禁用深度更新。这意味着粒子将针对地面进行测试，但是，其测试结果不会被写入深度缓冲区，这样它们就不会彼此阻挡了。因为我们最后才绘制粒子，因此这个方法行得通。

按如下代码更新 drawParticles() 关闭深度测试的写操作：

```
glDepthMask(false);
«draw»
glDepthMask(true);
```

再次运行这个应用；现在，一切看起来都应该和期望的一样了。

深度缓冲区和透视除法

深度缓冲区存储透视除法加工后的深度值，这在深度值和距离之间创建了一种非线性关系。它有这样一个效果，在与近平面接近的地方，深度的精度很高，而随着距离的增加，其精度也越来越低，这可能导致一些缺陷。由于这个原因，透视投影的近平面和远平面之间的比率不应该大于当前场景所需要的比率（也就是说，其近平面处的值为 1，且其远平面处的值为 100，平面可能没有问题，但是，其近平面处的值为 0.001，而其远平面处的值为 100 000，就会出现问题）。

12.5.2 剔除

OpenGL 为我们提供了另一种提高性能的方法，通过使能剔除（culling）技术消除隐藏面。默认情况下，OpenGL 把所有的多边形表面当作两面渲染。你可以很容易地看到这个效果，编辑 updateViewMatrices()，把调用 translateM() 的最后一个参数改为 15f；这会把我们的视点放在高度图的正下方。此时看看四周，一切看起来都相当奇怪，我们从来都没有期望“从内向外”观察地形，如图 12-5 所示。

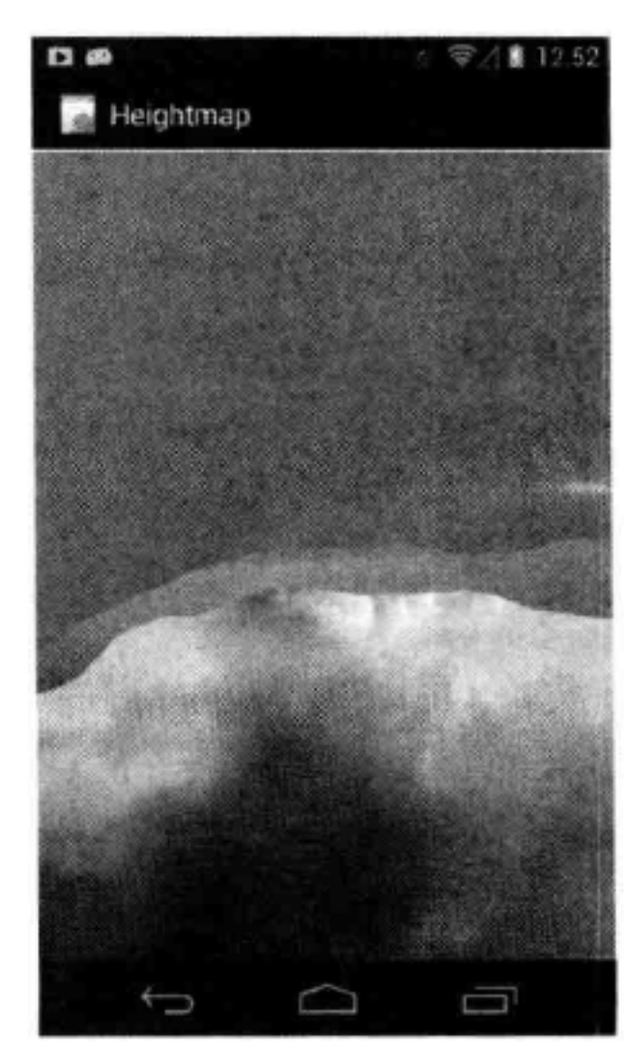


图 12-5 高度图的下方

渲染地形图的下方从来都没有什么用处，因此，我们可以告诉 OpenGL 关闭两面绘制，从而削减绘制开销。为此，我们回到 onSurfaceCreated()，添加一个调用 glEnable(GL_CULL_FACE)。OpenGL 就会查看每个三角形的卷曲顺序，它就是我们定义顶点的顺序。从观察点上看，如果这个卷曲顺序是逆时针的，这个三角形将被绘制出来，否则，它就会被丢掉。

我当时没有指出这一点，但是，在定义物体时，实际上已经非常小心了，从观察者的角度来说，它们的卷曲顺序都是正确的，一旦我们使能了剔除功能，它就会工作，不需要任何进一步的努力。

12.6 小结

在本章中，我们学习了如何从一个文件加载高度图，并把它转化为 OpenGL 可以绘制的形式，其使用了顶点缓冲区对象和索引缓冲区对象。高度图是用来表示地形的一种很好的方式，但是，因为它们是基于二维图的，所以不容易表示地形上的“洞”，比如洞穴或拱形结构。高度图还有尺寸固定的分辨率，这基于源图像的宽度、高度和每个像素的精度。

我们还探讨了深度测试和剔除，这两个技术使我们更容易正确地渲染物体，并同时提高性能。当我们在 11.4 节中创建天空盒着色器的程序时，我们提到过，把天空盒放在远平面处，并把它绘制在其他物体后面，这会使得性能得到提升。之所以这么说的原因之一是当我们使用深度缓冲区时，GPU 会丢掉不可见的片段，从而节省绘制时间。另外一个原因是很多移动设备的 GPU 使用了基于图块（tile-based）的渲染，它允许这些 GPU 快速地丢掉大面积的隐藏几何体[⊖]。

利用硬件剔除技术把如天空盒一样的东西绘制在所有其他不透明的物体后面，这也是一个不错的主意，如此，GPU 能够看到大部分的天空盒都被地形遮罩了，跳过那些部分可以节省时间。

12.7 练习

我们在本章讲述了很多新特性，因此，让我们花点时间详细地探索一下它们：

- 阅读关于 glCullFace() 和 glFrontFace() 的内容。然后做一个实验，在 onSurfaceCreated() 中加入一个 glFrontFace(GL_CW) 的调用。
- 改变地形的片段着色器，让它用片段深度作为颜色渲染地形（提示：你可以使用 gl_FragCoord.z 取得当前的深度值，其范围是从 0 到 1；别忘了，这个值是非线性的）。
- 给高度图添加一个纹理。要让事情更加有趣，根据高度的不同，混合两个不同的纹理。
- 一旦以上工作完成了，加入一个水平的水面盖住地形底部。
- 改变高度图的实现，使用三角形带和退化三角形（你可以使用你最喜欢的搜索引擎了解如何做到这一点，或者浏览“Learn OpenGL ES”的网站[⊜]）。不要担心，这些退化不是坏事，它们实际上是一个很好的方法，以一种更加紧凑和高效的形式存储一个高度图。

作为一个研究性练习，你还可以阅读关于三维像素（voxel）的内容，它是高度图的一个替代，它把地形存储在一个三维矩阵中。Minecraft 是使用此技术的一个著名的例子。

当你准备好了，让我们前往下一章！

⊖ http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLESPlatforms/OpenGLESPlatforms.html（译者注：此处链接失效，读者可搜索“tile-based GPU”寻找相关内容，这是一种 GPU 架构）。

⊜ http://www.learnopengles.com/android-lesson-eight-an-introduction-to-index-buffer-objects-ibos/

Chapter 13 第 13 章

点亮世界

光照的仿真已经成为计算机图形领域的一个主要研究课题，我们可以看到这个研究的影响，这不仅反映在逐步提升的游戏视觉上，而且还反映在电影和电视节目中的电脑成像(CGI，Computer Generated Imagery) 等领域。

光照算法的使用可以使一个场景的外观以一个非常不同的方式呈现。白天可以过渡到晚上，某些部分可能落入阴影中，一座山可以显示山峰、峡谷和裂隙。二维场景甚至也可以利用光照的特性实现一些效果，比如视觉深度、爆炸等。在本章中，我们要学习如何使用一些简单的光照算法给我们的山体加入对比，然后，使其变暗到晚上背景，使这些粒子喷泉焕发光彩。

让我们预览一下本章的开发计划：

- 首先，我们将学习如何使用方向光源实现漫反射（diffuse reflection），使那个场景看起来就像被天空盒中的太阳照射一样，然后，我们会学习如何加入一些环境光（ambient light）使黑暗的阴影最小化。
- 然后，我们要关上天空盒以得到一个较暗的天空盒，并调低亮度，我们会学习如何使用点光源点亮每个粒子喷泉。

让我们开始这场表演，把前一章的项目复制为一个新的项目，并命名为“Lighting”。

13.1 仿真光照的效果

当我们观察周围的世界时，实际上看到的是上万亿的微小的粒子累积的效果，这些粒子叫作光子。光子是由能量源发射的，比如太阳；在长距离的旅行之后，它们会从一些物

体弹开并产生折射，直到最后撞到我们眼睛后面的视网膜上，我们的眼睛和大脑会从那里吸收它们，并重新构建所有这些光子的活动形成我们所看到的世界。

过去计算机图形学通过这些模拟实际光子的行为或者用简化操作虚构其行为来仿真光照的效果。模拟实际光子的行为的一个方法是使用射线跟踪器（ray tracer）。射线跟踪器通过向场景中发射射线模拟光子，并计算这些射线如何与场景中的物体交互。这个技术功能十分强大，它本身非常善于实现反射、折射和其他特殊的效果，如焦散（当光线穿过水面时，就能看到这个效果）。

不幸的是，对于实时渲染来说，使用射线跟踪器开销太大。作为替代，大多数游戏和应用程序都会使事情简化，在较高的层次上近似于光线的工作方式，而不是直接模拟它。简单的算法可以被广泛使用，也有很多方法用来虚构反射、折射和更多的效果。这些技术可以让 OpenGL 把大部分负载放在 GPU 上，即使在一个移动平台上，它们也运行得非常快。

在 OpenGL 中使用光照

要给一个 OpenGL 的场景添加光照，我们通常可以把不同的光源分为下面的组：

环境光（Ambient light）

环境光看上去来自四面八方，场景中的一切被照亮的程度都一样。这近似于我们从大的、平等的光源获取的光照，比如天空，环境光也能用于虚构在光线到达我们的眼睛之前从许多物体弹开的效果，因此阴影从来不会被漆成黑色。

方向光（Directional light）

方向光看上去似乎来自一个方向，光源好像处于极其远的地方。这与我们从太阳或月亮获取的光照相似。

点光（Point light）

点光看上去是从附近某处投射的光亮，而且光的密度随着距离而减少。这适于表示近处的光源，其把它们的光投射到四面八方，向一个灯泡或蜡烛一样。

聚光（Spot light）

聚光与点光类似，只是加了一个限制，只能向一个特定的方向投射。这是我们从手电筒或者聚光灯（顾名思义）所获得的光照类型。

我们也可以把光线在物体表面反射的方式分为两类：

漫反射（Diffuse reflection）

漫反射是指光线平等地向所有方向蔓延，适于表示没有抛光表面的材质，比如地毯或外面的混凝土墙。这些类型的表面似乎有很多不同的观察点一样。

镜面反射（Specular reflection）

镜面反射在某个特定的方向上反射更加强烈，适于被抛光的或者闪亮的材质，比如光滑的金属或者刚刚打过蜡的汽车。

许多材质都结合了两方面的特质：用沥青路面作为例子。一个路面通常从许多方向上

看起来都是一样的，但是，当太阳在低空时，并且条件合适的情况下，某个方向上可能反射足够的太阳光，以至于让司机在那个点上致盲，从而引起交通事故。

13.2 用朗伯体反射实现方向光

要实现漫反射，我们可以使用一个简单的技术，即朗伯体反射（Lambertian reflectance），它是以 Johann Heinrich Lambert 命名的，他是生活在 18 世纪的一位瑞士数学家和天文学家。朗伯体反射描述了这样一个表面，它会反射所有方向上打到它的光线，以使它在任何观察点上看起来都是一样的。它的外观只依赖于它与光源所处的方位及其距离。

让我们看一个例子，这个例子使用一个水平表面和单个的、不随距离而减弱的方向光源，因此，唯一有影响的就是这个表面相对于该光源所处的方位。在图 13-1 中，我们可以看到这个表面垂直正对光源。在这个角度，它捕捉并反射尽可能多的光线。在图 13-2 中，表面相对光源旋转了 45 度，这样，它就不能再捕捉和反射那么多的光线了。

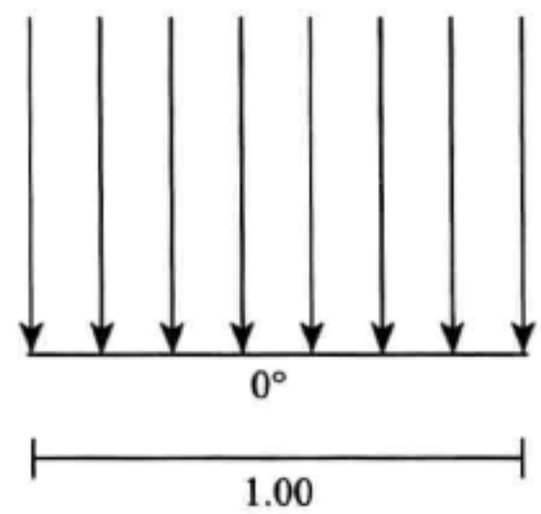


图 13-1 直接面对光源的表面

在 45 度时，这个表面反射的光线少了多少呢？如果我们测量一下它的横截面，就会发现它只是表面没有旋转之前的宽度的 0.707 倍。这与旋转角的余弦有关（参见图 8-1）；要计算出一个表面接收了多少光线，我们所需要做的就是计算出它直接面向光源时接收了多少光线，再把结果乘以那个角度的余弦值。

例如：如果一个朗伯体表面正常情况下从与其成 0 度角的方向光源接收了 5 流明光线，那么，当它相对于那个光源处于 45 度方位时，它就会反射约 3.5 流明的光线 (5 × cos45°)。要理解朗伯体反射，就需要理解这个关系。

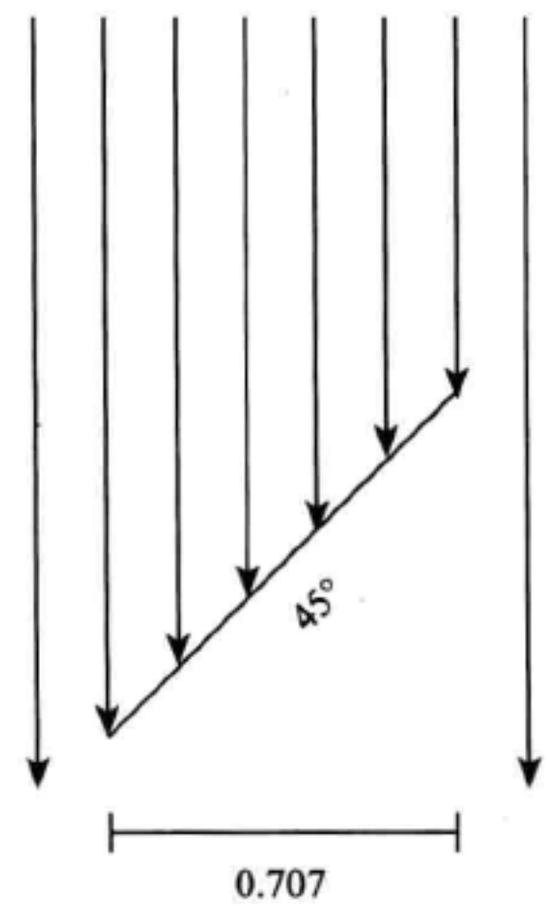


图 13-2 与光源成一定角度的表面

13.2.1 计算高度图的方位

在给高度图添加朗伯体反射之前，我们需要某个方法获知其表面的方位是什么。因为高度图不是一个水平的表面，我们需要计算高度图上每个点的方位。我们可以用表面法线（surface normal）表示方位，它是一个特殊类型的向量，它垂直于表面，且有一个值为 1 的单位长度；我们第一次了解到这些还是在 9.3 节中。

因为表面法线是应用于表面的而不是点，在我们计算每个点的法线时，把这个点的邻

接点合并在一起创建一个平面。我们将用两个向量来表示这个平面：一个从右侧的点指向左侧的点，另外一个从上面的点指向下面的点。如果我们计算这两个向量的叉积，就能得到一个垂直于这个平面的向量，然后，我们可以归一化那个向量以得到中间点的表面法线[⊖]。

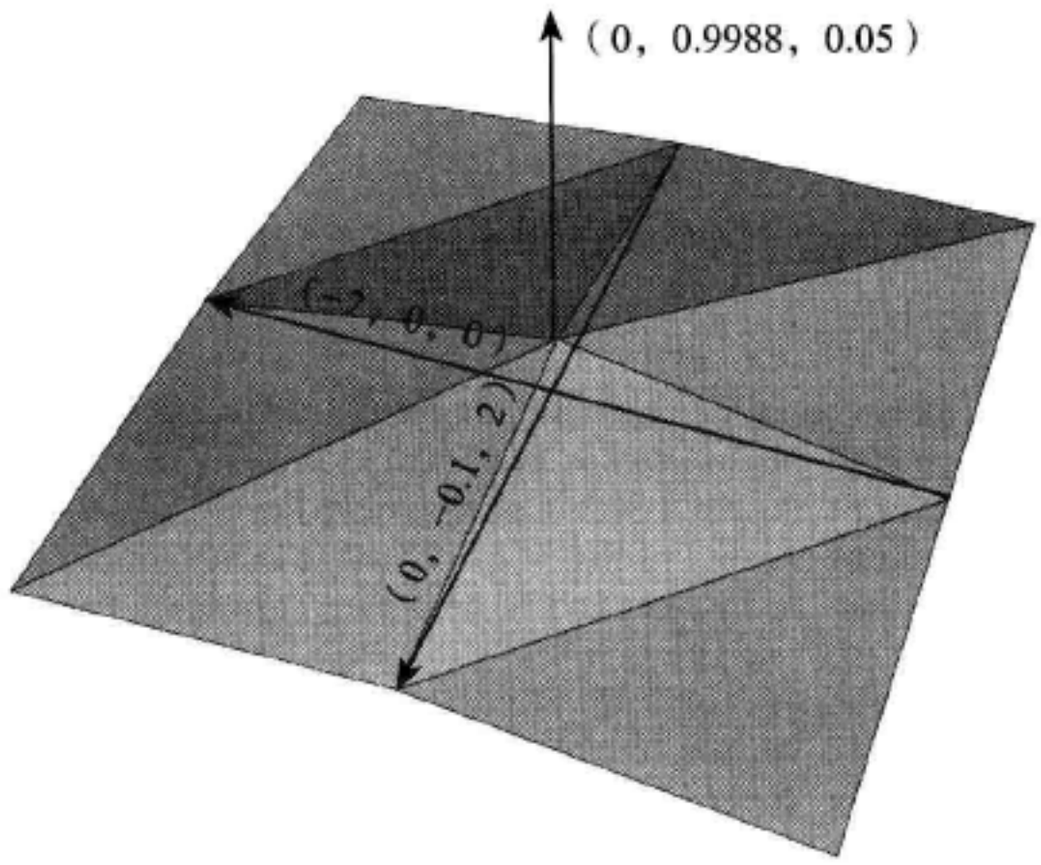


图 13-3 为高度图中的一个点生成表面法线

看看图 13-3 中的例子。

我们假定每个点占用一个单位部分，x 值向右增加，z 值向下增加。其上边、左边、右边和下边的点的高度分别是 0.2、0.1、0.1 和 0.1。要计算从右向左的向量，我们用右边的点减去左边的点得到值为 (−2, 0, 0) 的向量，再用上边和下边的点做同样的计算得到值为 (0, −0.1, 2) 的向量。一旦有了这两个向量，我们就可以计算它们的叉积得到值为 (0, 4, 0.2) 的向量，我们再把这个向量归一化得到值为 (0, 0.9988, 0.05) 的表面法线。

我们为什么使用从右向左的向量，而不是从左向右呢？因为我们想要这个表面法线指向上方，离开高度图，所以我们用叉积的右手规则计算出每个向量需要指向的方向，以使这个叉积就是正确的方向[⊖]。

现在我们已经知道了要做什么，让我们打开 HeightMap 类，开始改写代码。首先，加入一些新的常量：

Lighting/src/com/particles/android/objects/Heightmap.java
```
private static final int NORMAL_COMPONENT_COUNT = 3;
private static final int TOTAL_COMPONENT_COUNT =
    POSITION_COMPONENT_COUNT + NORMAL_COMPONENT_COUNT;
private static final int STRIDE =
    (POSITION_COMPONENT_COUNT + NORMAL_COMPONENT_COUNT) * BYTES_PER_FLOAT;
```

我们将改变顶点缓冲区，以使它存储位置和法线，为此，我们需要知道全部的分量计数和跨距。让我们更新 loadBitmapData() 中 heightmapVertices 的赋值语句，为法线增加一些空间：

Lighting/src/com/particles/android/objects/Heightmap.java
```
final float[] heightmapVertices =
    new float[width * height * TOTAL_COMPONENT_COUNT];
```

这可以保证有足够的空间留给位置和法线。还是在 loadBitmapData() 内部，在我们生成位置的那个循环体内，把它更新为如下代码：

⊖ http://en.wikipedia.org/wiki/Cross_product

⊖ http://en.wikipedia.org/wiki/Cross_product#Definition

Lighting/src/com/particles/android/objects/Heightmap.java

```
final Point point = getPoint(pixels, row, col);

heightmapVertices[offset++] = point.x;
heightmapVertices[offset++] = point.y;
heightmapVertices[offset++] = point.z;
```

我们已经把位置的生成提取为一个独立的方法了，不久就会添加它的定义。让我们再添加些代码获取当前点的邻接点，并为其生成表面法线：

Lighting/src/com/particles/android/objects/Heightmap.java

```
final Point top = getPoint(pixels, row - 1, col);
final Point left = getPoint(pixels, row, col - 1);
final Point right = getPoint(pixels, row, col + 1);
final Point bottom = getPoint(pixels, row + 1, col);

final Vector rightToLeft = Geometry.vectorBetween(right, left);
final Vector topToBottom = Geometry.vectorBetween(top, bottom);
final Vector normal = rightToLeft.crossProduct(topToBottom).normalize();

heightmapVertices[offset++] = normal.x;
heightmapVertices[offset++] = normal.y;
heightmapVertices[offset++] = normal.z;
```

为了生成这个法线，我们遵循了前面总结的算法：首先，获取其邻接点；然后，用这些点创建代表其平面的两个向量；最后，采用这两个向量的叉积，并把它归一化以得到其表面法线。我们还没有定义 normalize()，因此，让我们打开 Geometry 类，找到 Vector 的定义，按如下代码加入这个方法：

Lighting/src/com/particles/android/util/Geometry.java

```
public Vector normalize() {
    return scale(1f / length());
}
```

让我们继续编写 Heightmap 类，加入 getPoint() 的定义：

Lighting/src/com/particles/android/objects/Heightmap.java

```
private Point getPoint(int[] pixels, int row, int col) {
    float x = ((float)col / (float)(width - 1)) - 0.5f;
    float z = ((float)row / (float)(height - 1)) - 0.5f;

    row = clamp(row, 0, width - 1);
    col = clamp(col, 0, height - 1);

    float y = (float)Color.red(pixels[(row * height) + col]) / (float)255;

    return new Point(x, y, z);
}

private int clamp(int val, int min, int max) {
    return Math.max(min, Math.min(max, val));
}
```

这段代码的工作与之前其在循环体内的代码是一样的，但是，它现在对邻接点超出边界的情况做了限制。比如，当我们为 (0, 0) 生成法线时，要取其上边和左边的点，这些点实

际上在高度图中是不存在的，这种情况下，我们假定它们存在，并给它们赋予与中间顶点一样的高度。这样，我们还可以为那个顶点生成一个表面法线。

让我们完成所有的改变，把 bindData() 更新为如下代码：

Lighting/src/com/particles/android/objects/Heightmap.java

```
public void bindData(HeightmapShaderProgram heightmapProgram) {
    vertexBuffer.setVertexAttribPointer(0,
        heightmapProgram.getPositionAttributeLocation(),
        POSITION_COMPONENT_COUNT, STRIDE);

    vertexBuffer.setVertexAttribPointer(
        POSITION_COMPONENT_COUNT * BYTES_PER_FLOAT,
        heightmapProgram.getNormalAttributeLocation(),
        NORMAL_COMPONENT_COUNT, STRIDE);
}
```

因为在同一个顶点缓冲区对象中既存储了位置，又存储了法线数据，我们现在不得不把跨距传递给调用 glVertexAttribPointer() 的辅助函数 setVertexAttribPointer()，以使 OpenGL 知道在每个元素之间需要跳过多少字节。第二个 setVertexAttribPointer() 调用也非常重要，因为我们也为法线指定了以字节为单位的起始偏移值；否则，OpenGL 就会读入一部分位置和一部分法线，并把那个值当作法线，这看起来会非常怪异。

13.2.2 给着色器加入方向光

既然高度图包括了法线，我们的下一个任务就是更新高度图着色器，为其加入支持方向光的代码。先给 heightmap_vertex_shader.glsl 加入一个新的 uniform，代码如下：

Lighting/res/raw/heightmap_vertex_shader.glsl

```
uniform vec3 u_VectorToLight;
```

这个向量将存储指向方向光源的归一化向量。我们还需要一个用于高度图法线的新属性：

Lighting/res/raw/heightmap_vertex_shader.glsl

```
attribute vec3 a_Normal;
```

现在，准备就绪了，在 v_Color 的第一个赋值后面，给着色器的主体加入如下代码：

Lighting/res/raw/heightmap_vertex_shader.glsl

```
vec3 scaledNormal = a_Normal;
scaledNormal.y *= 10.0;
scaledNormal = normalize(scaledNormal);
```

你可能记得，当我们绘制高度图的时候，使用 scaleM() 扩展了它，使它变成 10 倍高和 100 倍宽，换句话说，高度图现在宽于它的高度 10 倍。以这种方式缩放改变了高度图的形状，意味着，我们预先产生的法线也不正确了。为了弥补这一点，我们按相反的方向缩放法线，使法线高于它的宽度 10 倍。重新归一化法线之后，它现在就会匹配这个新的几何形状了。

这个工作原理涉及一些高等数学，因此，就目前而言，你只需接受这个结果。稍后，我们会在 13.3.2 节中看一个更通用的调整法线的方法。

我们已经调整了表面法线，接下来计算朗伯体反射：

Lighting/res/raw/heightmap_vertex_shader.glsl
```
float diffuse = max(dot(scaledNormal, u_VectorToLight), 0.0);
v_Color *= diffuse;
```

要计算表面与光线之间夹角的余弦值，我们要计算指向光源的向量与表面法线的点积。它的工作原理是，当两个向量都是归一化的向量时，那两个向量的点积就是它们之间夹角的余弦，这恰恰就是我们计算朗伯体反射所需要的[⊖]。

为了避免出现负的结果，我们用 max() 把最小余弦值限制为 0，然后，应用这个光线，把当前顶点的颜色与余弦值相乘。余弦值在 0 和 1 之间，因此，最终的颜色将是处于黑色和原色之间的某个颜色。

13.2.3 更新着色器封装类代码

我们现在需要更新封装类以反应这些新的改变。首先，为 ShaderProgram 加入如下这些新的常量：

Lighting/src/com/particles/android/programs/ShaderProgram.java
```
protected static final String U_VECTOR_TO_LIGHT = "u_VectorToLight";
protected static final String A_NORMAL = "a_Normal";
```

切换到 HeightmapShaderProgram，为方向光的 uniform 的位置和法线属性的位置加入新的成员变量：

Lighting/src/com/particles/android/programs/HeightmapShaderProgram.java
```
private final int uVectorToLightLocation;
private final int aNormalLocation;
```

在构造函数的结尾处加入如下代码，为这些新位置赋值：

Lighting/src/com/particles/android/programs/HeightmapShaderProgram.java
```
uVectorToLightLocation = glGetUniformLocation(program, U_VECTOR_TO_LIGHT);
aNormalLocation = glGetAttribLocation(program, A_NORMAL);
```

我们需要更新 setUniforms()，以便可以更新这个新的 uniform，如下代码所示：

Lighting/src/com/particles/android/programs/HeightmapShaderProgram.java
```
public void setUniforms(float[] matrix, Vector vectorToLight) {
    glUniformMatrix4fv(uMatrixLocation, 1, false, matrix, 0);
    glUniform3f(uVectorToLightLocation,
        vectorToLight.x, vectorToLight.y, vectorToLight.z);
}
```

让我们完成这个类的改变，给法线属性的位置加入一个新的存取函数：

Lighting/src/com/particles/android/programs/HeightmapShaderProgram.java
```
public int getNormalAttributeLocation() {
    return aNormalLocation;
}
```

一切就绪，在渲染器类中将它们结合在一起，看看新光照的作用。

⊖ http://en.wikipedia.org/wiki/Dot_product

13.2.4 看看方向光的作用

打开 ParticlesRenderer，给我们的光源定义实际的向量：

Lighting/src/com/particles/android/ParticlesRenderer.java
```
private final Vector vectorToLight = new Vector(0.61f, 0.64f, -0.47f).normalize();
```

这个向量大约指向天空盒中的太阳。你可以用下面这些步骤得出一个相似的结果：

1. 创建一个指向 (0, 0, −1) 的向量，也就是指向正上方。
2. 按场景旋转方向的反向旋转这个向量。
3. 加入日志语句打印这个向量当前的方向，然后运行这个应用，旋转场景直到太阳处于屏幕的中间。

我们同样归一化这个向量，以便将它传递给着色器，并用它计算朗伯体反射。让我们将 drawHeightmap() 中的 heightmapProgram.setUniforms() 调用更新为下面的代码，把这个向量传递给着色器：

Lighting/src/com/particles/android/ParticlesRenderer.java
```
heightmapProgram.setUniforms(modelViewProjectionMatrix, vectorToLight);
```

就是这些！让这个应用运行一次，看看我们得到了什么；它看上去应该与图 13-4 相似：

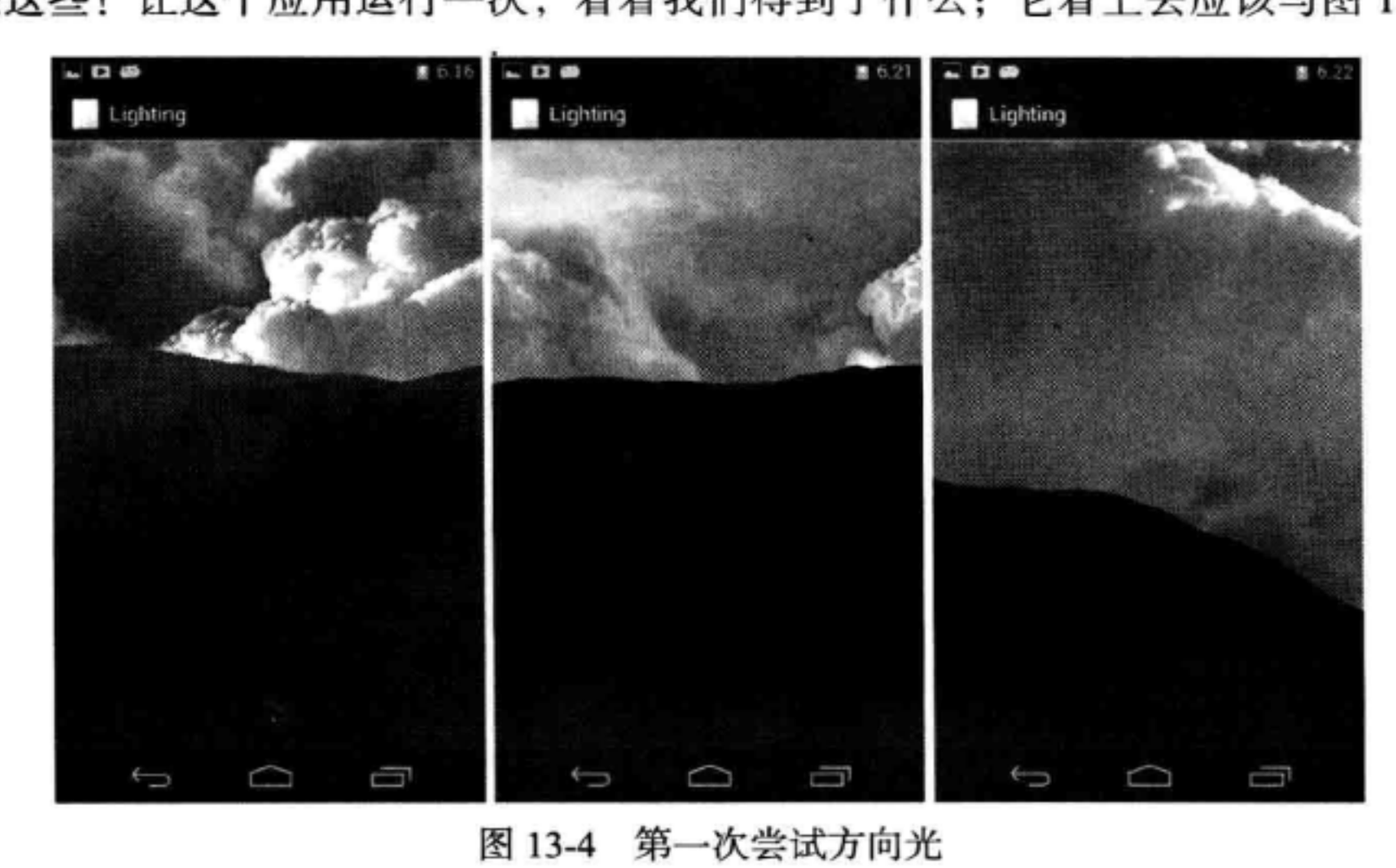


图 13-4 第一次尝试方向光

我们现在可以看到山的形状和形式了，但是你可能注意到了，黑暗的区域太黑了。问题是由于我们没有全局照明；在实际生活中，光在到达眼睛前通过天空漫射并在许多物体上发生了反射，因此，太阳投下的阴影没有一处是近乎漆黑的。我们可以在场景中加入环境光伪造这个现象，环境光平等地照射到所有物体上。让我们回到 heightmap_vertex_shader.glsl，在 vColor 和漫反射相乘的那行代码后面加入如下代码：

Lighting/res/raw/heightmap_vertex_shader.glsl
```
float ambient = 0.2;
v_Color += ambient;
```

这给整个高度图加入一个照明的基础量，这样，就没有东西会呈现得太暗。再看一下这个效果，看看我们得到了什么。如图 13-5 所示，阴影现在看起来更加合理了。

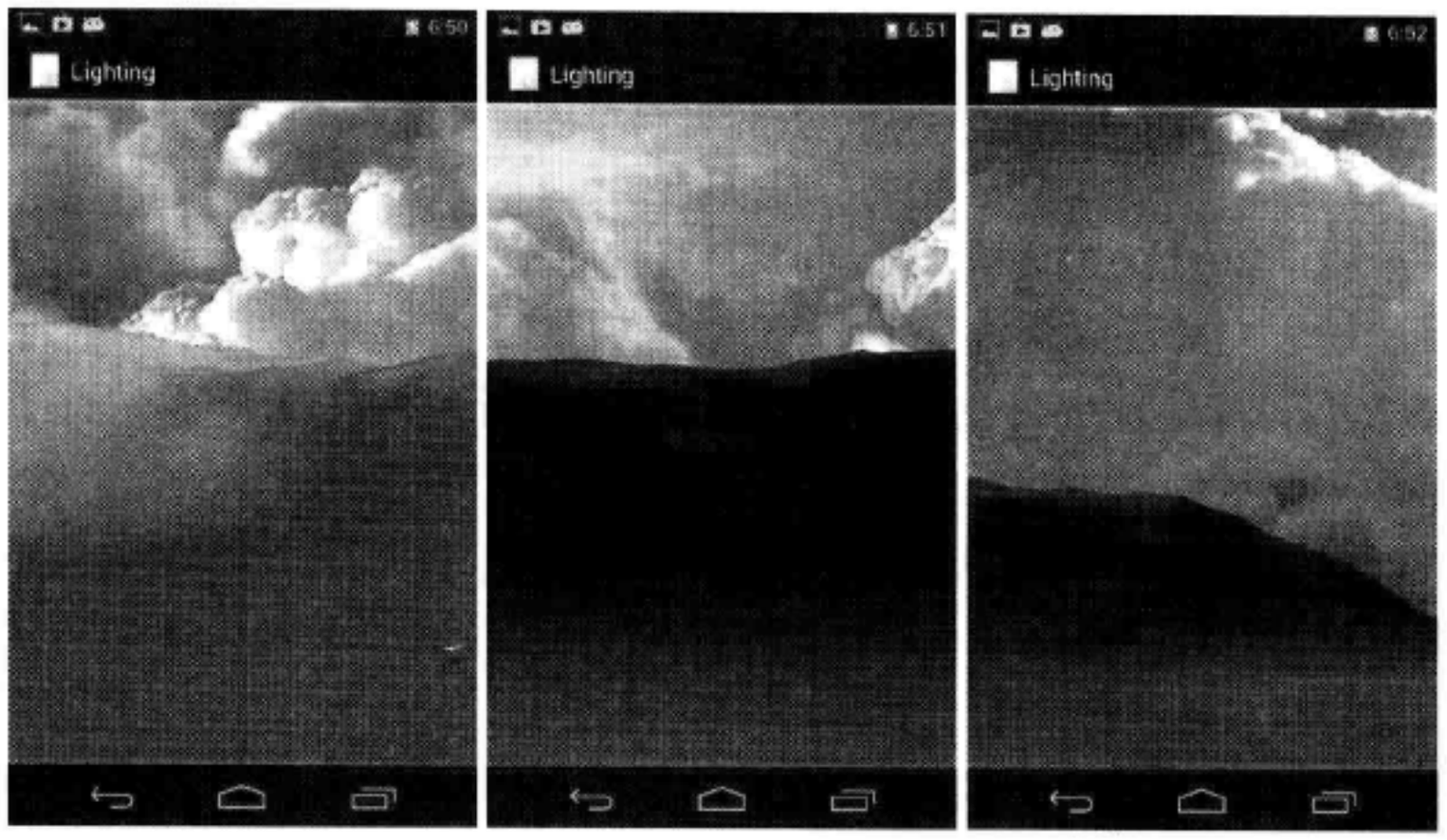


图 13-5 加入环境光

13.3 添加点光

现在我们准备好给场景增加一些点光源了，这样就可以让粒子喷泉发光了。在一个明亮的背景下，我们无法很好地看到这个效果，因此，我们要做的第一件事就是切换到夜晚的天空盒。你可以从网站上下载这个夜晚的天空盒，并把它放在项目的“/res/drawable-nodpi”文件夹中[⊖]。

要切换到夜晚的天空盒，让我们回到 ParticlesRenderer 类，并按如下代码更新 skyboxTexture 的赋值语句：

Lighting/src/com/particles/android/ParticlesRenderer.java

```
skyboxTexture = TextureHelper.loadCubeMap(context,
new int[] { R.drawable.night_left, R.drawable.night_right,
            R.drawable.night_bottom, R.drawable.night_top,
            R.drawable.night_front, R.drawable.night_back});
```

然后，把指向光源的向量更新为如下代码：

Lighting/src/com/particles/android/ParticlesRenderer.java

```
private final Vector vectorToLight = new Vector(0.30f, 0.35f, -0.89f).normalize();
```

这个新的向量指向天空盒中的月亮。我们还需要在着色器中调低光的强度，因此，让我们返回 heightmap_vertex_shader.glsl，在漫反射与 v_Color 相乘之前，给它加入如下的调整量：

⊖ http://pragprog.com/book/kbogla/

Lighting/res/raw/heightmap_vertex_shader.glsl
```
diffuse *= 0.3;
```

我们还应该调低环境光：

Lighting/res/raw/heightmap_vertex_shader.glsl
```
float ambient = 0.1;
```

如果你现在继续运行这个应用，应该看到一个夜晚的背景和相应变暗的高度图。

13.3.1 理解点光源

我们一直在讲述漫反射和朗伯体反射模型，点光与方向光在数学上都是相似的，然而，我们需要记住两个关键的不同点：

- 对于方向光，我们只存储指向其光源的向量，因为这个向量对于场景中的所有点都是一样的；对于点光则相反，我们将存储其光源的位置，用这个位置计算场景中每个点指向这个光源的向量。
- 在实际生活中，点光源的亮度会随着距离的平方降低；这叫作平方反比定律（The Inverse Square Law）[⊖]。我们将使用点光源的位置计算出其与场景中每个点的距离。

13.3.2 给着色器加入点光

要实现点光，我们需要改变着色器，借助这次机会我们将采用一个更加结构化且通用的方法在着色器中加入光照。让我们看一看最重要的一些改变：

- 我们将把位置和法线放入眼空间（eye space），在这个空间里，所有的位置和法线都是相对于照相机的位置和方位；这样做是为了使我们可以在同一个坐标空间中比较所有事物的距离和方位。我们为什么使用眼空间而不是世界空间呢？因为镜面光也依赖于照相机的位置，即使我们在本章中没有使用镜面光，学习如何使用眼空间还是一个好主意，这样，我们在以后就可以直接使用它了。
- 要把一个位置放进眼空间中，我们只需要让它与模型矩阵相乘，把它放入世界空间中，然后再把它与视图矩阵相乘，这样就把它放入眼空间了。为了简化操作，我们可以把视图矩阵与模型矩阵相乘得到一个单一的矩阵，称为模型视图矩阵，再用这个矩阵把我们的位置放入眼空间中。
- 如果模型视图矩阵只包含平移或旋转，这对法线也是有用的，但是，如果我们缩放了一个物体，会怎么样？如果缩放在所有方向上都是一样的，我们只需要重新归一化法线，使它的长度保持为 1，但是如果物体在某个方向上被压扁了，那么我们要补偿那一点。

当我们加入方向光时，我们确切地知道高度图缩放了多少，因此，可以直接补偿。

⊖ http://en.wikipedia.org/wiki/Inverse-square_law

这不是一个灵活的方案，有一个通用的方法可以实现这些，倒置模型视图矩阵，转置这个倒置的矩阵，让法线与那个矩阵相乘，然后归一化其结果。这个方法可行的原因涉及一些高等数学；如果你感兴趣，提供两个很好的解释链接，它们讲述了很多详细的内容[⊖]。

首先，用下面的内容替换 heightmap_vertex_shader.glsl：

Lighting/res/raw/heightmap_vertex_shader.glsl

```
uniform mat4 u_MVMatrix;
uniform mat4 u_IT_MVMatrix;
uniform mat4 u_MVPMatrix;

uniform vec3 u_VectorToLight;             // In eye space
uniform vec4 u_PointLightPositions[3];    // In eye space
uniform vec3 u_PointLightColors[3];

attribute vec4 a_Position;
attribute vec3 a_Normal;

varying vec3 v_Color;

vec3 materialColor;
vec4 eyeSpacePosition;
vec3 eyeSpaceNormal;

vec3 getAmbientLighting();
vec3 getDirectionalLighting();
vec3 getPointLighting();
```

我们现在将使用 u_MVMatrix 表示模型视图矩阵，使用 u_IT_MVMatrix 表示那个倒置矩阵的转置，并用 u_MVPMatrix 表示合并后的模型视图投影矩阵，正如我们以前使用 u_Matrix 一样。

方向光向量还与以前一样，不同之处在于，我们现在希望它被定义在眼空间中。我们用 u_PointLightPositions 传递点光源的位置，它也被定义在眼空间中，并用 u_PointLightColors 传递颜色。最后两个 uniform 被定义为数组，以便我们可以通过一个 uniform 传递多个向量。

对于属性，我们现在用一个 vec4 类型表示其位置，以减少 vec3 和 vec4 之间的转换次数。我们不需要改变顶点数据，因为 OpenGL 会用默认值 1 设置第 4 个分量，但是还要小心：uniform 就不一样了，它们必须要指定所有分量的值。

varying 还与以前一样；varying 定义之后，我们加入了几个新的变量，我们要使用它们计算光照，我们还有三个新函数的声明，稍后会在着色器中定义它们。

继续在着色器中加入下面的代码：

⊖ 分别是 http://arcsynthesis.org/gltut/index.html 和 http://www.cs.uaf.edu/2007/spring/cs481/lecture/01_23_matrices.html

Lighting/res/raw/heightmap_vertex_shader.glsl

```
void main()
{
    materialColor = mix(vec3(0.180, 0.467, 0.153),    // A dark green
                        vec3(0.660, 0.670, 0.680),    // A stony gray
                        a_Position.y);
    eyeSpacePosition = u_MVMatrix * a_Position;

    // The model normals need to be adjusted as per the transpose
    // of the inverse of the modelview matrix.
    eyeSpaceNormal = normalize(vec3(u_IT_MVMatrix * vec4(a_Normal, 0.0)));

    v_Color = getAmbientLighting();
    v_Color += getDirectionalLighting();
    v_Color += getPointLighting();

    gl_Position = u_MVPMatrix * a_Position;
}
```

在着色器的主方法内，我们像以前一样给材质的颜色赋值，并在眼空间中计算当前的位置和法线。然后，我们计算每一种光的类型，并把其结果颜色累加到 v_Color 上，最后，像以前一样投影其位置。

继续编写如下代码：

Lighting/res/raw/heightmap_vertex_shader.glsl

```
vec3 getAmbientLighting()
{
    return materialColor * 0.1;
}

vec3 getDirectionalLighting()
{
    return materialColor * 0.3
         * max(dot(eyeSpaceNormal, u_VectorToLight), 0.0);
}
```

正如我们之前所做的一样，这两个函数计算环境光和方向光。让我们用下面为点光定义的函数结束这个着色器：

Lighting/res/raw/heightmap_vertex_shader.glsl

```
vec3 getPointLighting()
{
    vec3 lightingSum = vec3(0.0);

    for (int i = 0; i < 3; i++) {
        vec3 toPointLight = vec3(u_PointLightPositions[i])
                          - vec3(eyeSpacePosition);
        float distance = length(toPointLight);
        toPointLight = normalize(toPointLight);

        float cosine = max(dot(eyeSpaceNormal, toPointLight), 0.0);
        lightingSum += (materialColor * u_PointLightColors[i] * 5.0 * cosine)
                       / distance;
    }

    return lightingSum;
}
```

介绍一下它的工作原理，我们循环计算每个通过的点光，计算其每一个的光照，并把其结果加入 lightingSum。这段代码用朗伯体反射计算其光照量，就像之前计算方向光一样，但还是有些很重要的不同点：

- 对于每个点光源，我们都要计算当前位置到那个光源的向量，也要计算当前位置与那个光源的距离。
- 一旦得到了归一化的向量，我们就可以计算其朗伯体反射。然后，把那个材质的颜色与点光的颜色相乘，并把其结果颜色应用于当前的顶点。我们用 5 放大这个结果，使其显得更明亮一点，并把它与其余弦值相乘，计算其朗伯体反射。
- 在把其结果累加到 lightingSum 之前，用前一步的结果除以其距离，以使其光照密度随距离而减少。

最后的计算完成了，我们的着色器也写完了。

13.3.3 显示设备的非线性本质

由于 OpenGL 和显示设备处理光照和颜色的方法不同，要在 OpengGL 中获得正确的光照和颜色有时候是非常棘手的。对于 OpenGL 来说，颜色是落在线性频谱上的，因此，一个值为 1.0 的颜色的亮度是值为 0.5 的颜色的两倍。然而，由于许多显示设备的非线性本质，对于显示器上的亮度，其实际的区别可能要比这个大多了。

其以这种方式工作的原因，部分是由于历史的原因。曾几何时，大家都用过大的、笨重的 CRT 显示器作为主要的显示设备，而这些显示器是通过把电子束射击到屏幕上工作的。这些荧光粉往往有一个指数响应，而不是线性响应，就使得 1.0 的颜色的亮度比 0.5 的颜色大两倍多。出于兼容性等原因，许多显示设备直到今天还维持着类似的行为[⊖]。

这种非线性的行为会搞乱我们的光照，使事物比实际要暗。通常情况下，计算光照的衰减是用其密度除以其距离的平方，但为了不让点光衰减得太快，我们去掉了指数，只除以其距离。

13.3.4 更新着色器的封装代码

我们现在需要更新着色器封装代码以匹配这个新的着色器。

打开 ShaderProgram，并加入一些新的常量：

Lighting/src/com/particles/android/programs/ShaderProgram.java

```
protected static final String U_MV_MATRIX = "u_MVMatrix";
protected static final String U_IT_MV_MATRIX = "u_IT_MVMatrix";
protected static final String U_MVP_MATRIX = "u_MVPMatrix";
protected static final String U_POINT_LIGHT_POSITIONS =
```

⊖ 参考 http://stackoverflow.com/questions/6397817/color-spaces-gamma-and-image-enhancement 和 http://http.developer.nvidia.com/GPUGems3/gpugems3_ch24.html

```
    "u_PointLightPositions";
protected static final String U_POINT_LIGHT_COLORS = "u_PointLightColors";
```

我们还需要改动 HeightmapShaderProgram。去掉 uMatrixLocation 和与其相关的代码，并加入如下这些新成员：

```
private final int uMVMatrixLocation;
private final int uIT_MVMatrixLocation;
private final int uMVPMatrixLocation;
private final int uPointLightPositionsLocation;
private final int uPointLightColorsLocation;
```

我们还需要更新其构造函数：

```
uMVMatrixLocation = glGetUniformLocation(program, U_MV_MATRIX);
uIT_MVMatrixLocation = glGetUniformLocation(program, U_IT_MV_MATRIX);
uMVPMatrixLocation = glGetUniformLocation(program, U_MVP_MATRIX);

uPointLightPositionsLocation =
    glGetUniformLocation(program, U_POINT_LIGHT_POSITIONS);
uPointLightColorsLocation =
    glGetUniformLocation(program, U_POINT_LIGHT_COLORS);
```

要完成这些改变，需要去掉当前的 SetUniforms()，并构造一个新的，如下代码所示：

```
public void setUniforms(float[] mvMatrix,
                        float[] it_mvMatrix,
                        float[] mvpMatrix,
                        float[] vectorToDirectionalLight,
                        float[] pointLightPositions,
                        float[] pointLightColors) {
    glUniformMatrix4fv(uMVMatrixLocation, 1, false, mvMatrix, 0);
    glUniformMatrix4fv(uIT_MVMatrixLocation, 1, false, it_mvMatrix, 0);
    glUniformMatrix4fv(uMVPMatrixLocation, 1, false, mvpMatrix, 0);
```

我们现在传递了几个矩阵，以及方向光和点光的位置与颜色。这个方法体的前三行代码把所有矩阵传递给着色器。让我们用下面的代码传递光照数据来完成这个方法：

```
    glUniform3fv(uVectorToLightLocation, 1, vectorToDirectionalLight, 0);

    glUniform4fv(uPointLightPositionsLocation, 3, pointLightPositions, 0);
    glUniform3fv(uPointLightColorsLocation, 3, pointLightColors, 0);
}
```

第一行传递方向光的向量给着色器，接下来的两行把点光的位置和颜色也传递给着色器。我们把着色器中的最后两个 uniform 定义为有三个向量的数组，因此，对于每个 uniform，我们调用 glUniform*fv() 时，都把第二个参数设为 3，3 是其计数。这告诉 OpenGL 需要从数组中为那个 uniform 读入 3 个向量。

13.3.5 更新 Renderer 类

我们现在只需要更新 ParticlesRenderer 以便定义和传递这些新的 uniform。首先，我们需要在类的顶部定义两个新的矩阵：

Lighting/src/com/particles/android/ParticlesRenderer.java

```
private final float[] modelViewMatrix = new float[16];
private final float[] it_modelViewMatrix = new float[16];
```

更新 updateMvpMatrix() 来设置这两个新的矩阵：

```
private void updateMvpMatrix() {
    multiplyMM(modelViewMatrix, 0, viewMatrix, 0, modelMatrix, 0);
    invertM(tempMatrix, 0, modelViewMatrix, 0);
    transposeM(it_modelViewMatrix, 0, tempMatrix, 0);
    multiplyMM(
        modelViewProjectionMatrix, 0,
        projectionMatrix, 0,
        modelViewMatrix, 0);
}
```

这段代码把 modelViewMatrix 设置为合并后的模型视图矩阵，并把 it_modelViewMatrix 设置为那个反转矩阵的倒置。回到类的顶部，我们还需要为那些新的光加入一些新成员：

```
final float[] vectorToLight = {0.30f, 0.35f, -0.89f, 0f};

private final float[] pointLightPositions = new float[]
    {-1f, 1f, 0f, 1f,
      0f, 1f, 0f, 1f,
      1f, 1f, 0f, 1f};

private final float[] pointLightColors = new float[]
    {1.00f, 0.20f, 0.02f,
     0.02f, 0.25f, 0.02f,
     0.02f, 0.20f, 1.00f};
```

这个新的 vectorToLight 的定义要替换掉前面的定义；你不久就会明白我们为什么要把它存到一个普通的浮点数组中。我们也要把每个点光源的位置和颜色存到它们各自的数组中，这些位置和颜色与我们为每个粒子发射器设定的位置和颜色大致匹配。其主要的区别在于每个点光源都被放在它的粒子发射器上方一个单位处，因为地形是绿色的，绿色的光也稍微变暗些，这样它就不会压制住红光和蓝光。

现在，我们只需要用下面的代码替换 drawHeightmap() 中的 setUniforms() 调用：

Lighting/src/com/particles/android/ParticlesRenderer.java

```
// Put the light positions into eye space.
final float[] vectorToLightInEyeSpace = new float[4];
final float[] pointPositionsInEyeSpace = new float[12];
multiplyMV(vectorToLightInEyeSpace, 0, viewMatrix, 0, vectorToLight, 0);
multiplyMV(pointPositionsInEyeSpace, 0, viewMatrix, 0, pointLightPositions, 0);
multiplyMV(pointPositionsInEyeSpace, 4, viewMatrix, 0, pointLightPositions, 4);
multiplyMV(pointPositionsInEyeSpace, 8, viewMatrix, 0, pointLightPositions, 8);

heightmapProgram.setUniforms(modelViewMatrix, it_modelViewMatrix,
    modelViewProjectionMatrix, vectorToLightInEyeSpace,
    pointPositionsInEyeSpace, pointLightColors);
```

我们需要把方向光的向量和点光源的位置放入眼空间中，为此，我们使用 Android 的 Matrix 类把它们乘以视图矩阵。那些位置已经在世界空间中了，因此，不必事先把它们与模型矩阵相乘。一旦完成了这些，我们就可以用 heightmapProgram.setUniforms() 调用把所

有数据传递给着色器。

让我们来看一眼！如果一切顺利，你的屏幕看上去应该与图 13-6 相似。

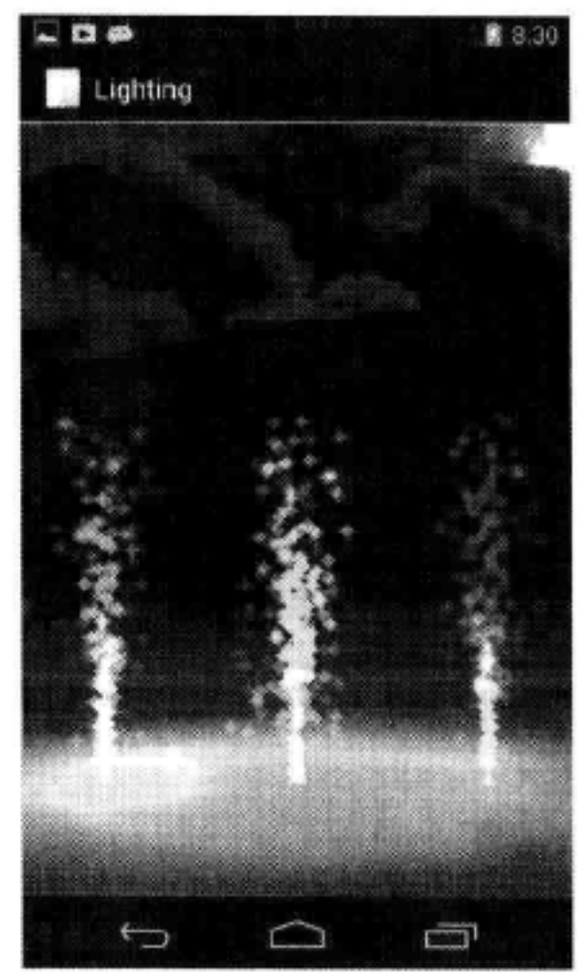


图 13-6　夜景衬托下的点光

13.4　小结

我们在本章讲述了很多内容，学习了所有关于环境光、方向光和点光的知识，我们还学习了如何用朗伯体反射实现漫反射。这些光照的公式可以更好地帮助我们实现酷炫优美的游戏和动态壁纸。

这里实现的框架也可以用来扩展成更复杂的光照类型，比如镜面反射。随着计算越来越复杂，我们通常会在之前的基础上开始构建，正如点光是在计算方向光的基础上构建的。

13.5　练习

这里有一些练习，帮助你更深入、更详细地了解这些概念：

- 把方向光的计算移出着色器，并把其结果作为每个顶点中的一个额外的浮点数存储到高度图的顶点缓冲区中，这样，我们就只需为每个顶点计算一次。
- 调整光照，使其变成动态的。实现它的一个方法是通过打开、关闭粒子发射器，然后，根据地面上剩余的粒子数量调整每个发射器的亮度。如果要更有创意一些，你甚至可以根据节拍调整它们。
- 把光照的计算移到片段着色器中，以使光照的计算针对每个片段，而不是针对每个顶点。当你把数据传递给片段着色器时，要记住，对于可以合理地线性插值的数据，你应该只使用 varying 类型；否则，你不会真正得到什么。你可以安全地传递顶点法线，但是，你需要在片段着色器中重新把它们归一化。

我们还有两个研究性的练习，用来跟进我们已经学过的内容：

- 看看是否有方法可以优化高度图的法线生成算法，去掉冗余的步骤。像 Wolfram Alpha 一样的工具可以帮助我们自动地实现这个算法[⊖]。
- 找找其他的方法，我们可以对事物进行网格化，并应用高度图提高它的视觉外观。提供两个资源的链接，你可以从此开始[⊜]。

当你准备好了，让我们前往下一章，把这个光照项目变成一个动态壁纸，它可以在运行于 Android 设备的主屏！

⊖ http://www.wolframalpha.com/ 或者 http://www.flipcode.com/archives/Calculating_Vertex_Normals_for_Height_Maps.shtml

⊜ http://web.eecs.umich.edu/~sugih/courses/eecs494/fall06/lectures/workshop-terrain.pdf 和 http://mtnphil.wordpress.com/2012/10/15/terrain-triangulation-summary/

Chapter 14 第 14 章

创建动态壁纸

从 Android 的早期版本开始，用户就可以用动态壁纸（live wallpaper）替换他们的设备的主屏幕背景，它是一个特殊类型的应用程序，可以绘制动态的、有活力的背景。这些动态壁纸已经被证明是十分流行的，因为它们的下载量大多达到百万！在本章中，你将学习如何用 OpenGL 和我们一直在使用的 GLSurfaceView 实现你自己的动态壁纸。

让我们看一下动态壁纸的开发计划：

- 首先，我们要学习如何用动态壁纸的 Service 组件和一个自定义的 GLSurfaceView 封装上一章的光照项目。
- 然后，我们要学习如何优化其性能和电池消耗。

作为开始，我们将继续上一章的光照项目，把它另存为一个新的项目，命名为“LiveWallPaper”。当我们完成时，我们的动态壁纸看起来将与图 14-1 相似。

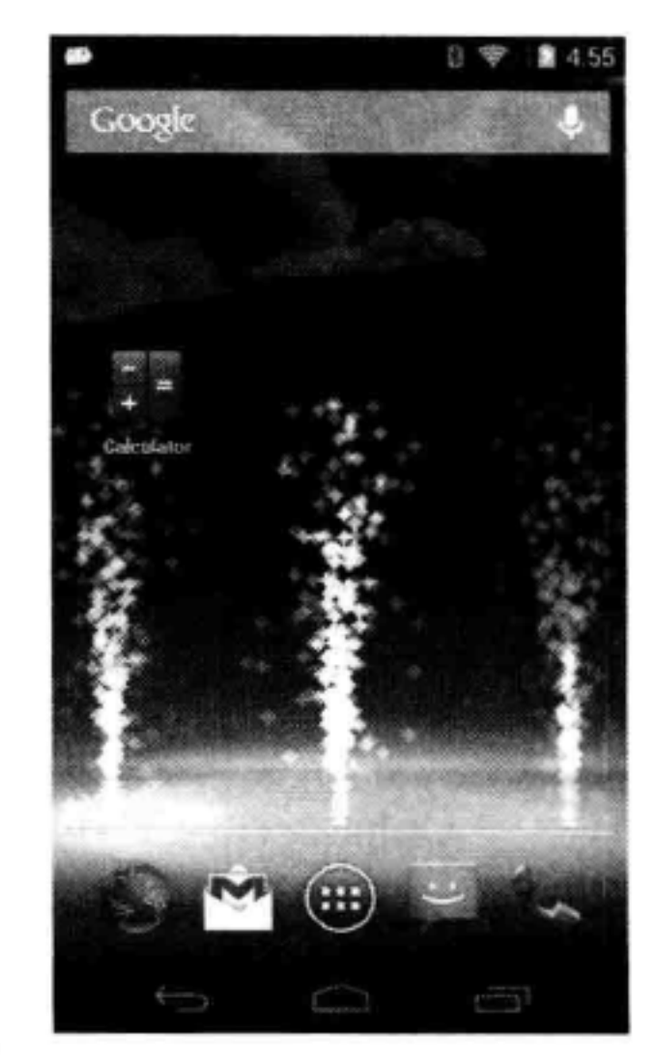


图 14-1 运行在主屏幕上的动态壁纸

14.1 实现动态壁纸的 Service 组件

要实现动态壁纸，我们将使用一个 Android 的 Service 组件，它是一种特殊类型的应用程序组件，可以用来为系统的其他部分提供特性。Android 用 WallpaperService 提供了基本的动态壁纸的实现，要创建动态壁纸，我们所需要做的就是用我们自己的实现继承这个基类[⊖]。

⊖ 分别是：http://developer.android.com/reference/android/service/wallpaper/WallpaperService.html 和 http://androiddevelopers.blogspot.ca/2010/02/live-wallpapers.html

14.1.1 继承 WallpaperService

首先，让我们创建一个新的名为“com.particles.android.wallpaper”的包。在这个新包中，让我们创建一个新类，称之为“GLWallpaperService”，代码如下：

LiveWallpaper/src/com/particles/android/wallpaper/GLWallpaperService.java

```
public class GLWallpaperService extends WallpaperService {
    @Override
    public Engine onCreateEngine() {
        return new GLEngine();
    }
}
```

这个 GLWallpaperService 本身是非常简单的：它只重载了 onCreateEngine() 方法，要实现一个动态壁纸，我们所需要做的就是重载那个方法，并返回 WallpaperService.Engine 的一个新的实例。这个引擎（engine）会包含动态壁纸的具体实现，它的生命周期事件与我们在第 1 章中所学过的 Activity 组件十分相似。

14.1.2 继承 WallpaperService.Engine

让我们为这个引擎加入一个基本的实现，并让它作为 GLWallpaperService 的一个内部类：

LiveWallpaper/src/com/particles/android/wallpaper/GLWallpaperService.java

```
public class GLEngine extends Engine {
    @Override
    public void onCreate(SurfaceHolder surfaceHolder) {
        super.onCreate(surfaceHolder);
    }

    @Override
    public void onVisibilityChanged(boolean visible) {
        super.onVisibilityChanged(visible);
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
    }
}
```

这些生命周期事件会如下面这样被 Android 调用：

onCreate(SurfaceHolder surfaceHolder)

当 WallpaperService 被系统创建时，它会创建它自己的渲染表面（Surface），让应用程序在上面绘制。然后，它会通过调用 onCreate() 初始化 WallpaperService 的 Engine，并把这个表面传递进去作为 SurfaceHolder，它是 Surface 类的一个抽象接口。

onVisibilityChanged(boolean visible)

无论动态壁纸可见还是被隐藏起来，它都会被调到。这个行为类似于 Activity 的回调方法 onPause() 和 onResume()，当动态壁纸被隐藏起来时我们需要暂停渲染，当它又可见时，

需要继续渲染。

onDestroy()

当动态壁纸被销毁时，它会被调用到。和其他 Android 组件一样，当进程被杀掉时，系统极有可能不会调用 onDestroy()。

要让我们的动态壁纸正确地运行，我们只需要实现这些生命周期回调方法，处理它们的方法很大部分都与我们在 ParticlesActivity 中所做的一样。

让我们从 onCreate() 开始，用我们在 ParticlesActivity 中所使用的方式初始化 OpenGL；让我们给 GLEngine 加入两个新的成员变量，并更新 onCreate()，如下所示：

LiveWallpaper/src/com/particles/android/wallpaper/GLWallpaperService.java

```
private WallpaperGLSurfaceView glSurfaceView;
private boolean rendererSet;

@Override
public void onCreate(SurfaceHolder surfaceHolder) {
    super.onCreate(surfaceHolder);
    glSurfaceView = new WallpaperGLSurfaceView(GLWallpaperService.this);

    // Check if the system supports OpenGL ES 2.0.
    ActivityManager activityManager =
        (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
    ConfigurationInfo configurationInfo = activityManager
        .getDeviceConfigurationInfo();

    final boolean supportsEs2 =
        configurationInfo.reqGlEsVersion >= 0x20000
            // Check for emulator.
            || (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH_MR1
             && (Build.FINGERPRINT.startsWith("generic")
              || Build.FINGERPRINT.startsWith("unknown")
              || Build.MODEL.contains("google_sdk")
              || Build.MODEL.contains("Emulator")
              || Build.MODEL.contains("Android SDK built for x86")));

    final ParticlesRenderer particlesRenderer =
        new ParticlesRenderer(GLWallpaperService.this);

    if (supportsEs2) {
        glSurfaceView.setEGLContextClientVersion(2);
        glSurfaceView.setRenderer(particlesRenderer);
        rendererSet = true;
    } else {
        Toast.makeText(GLWallpaperService.this,
            "This device does not support OpenGL ES 2.0.",
            Toast.LENGTH_LONG).show();
        return;
    }
}
```

这段代码或多或少与我们在每章中一直在使用的 OpenGL 初始化代码相同；主要的不同在于我们要在一个动态壁纸的 Service 中使用它，而不是在一个 Activity 中，因为 GLSurfaceView 主要是用来与 Activity 一起工作，我们需要用一些小改动创建一个自定义的版本。

14.1.3 创建一个自定义的 GLSurfaceView

为 GLEngine 加入一个自定义的 GLSurface，并作为其内部类：

LiveWallpaper/src/com/particles/android/wallpaper/GLWallpaperService.java

```
class WallpaperGLSurfaceView extends GLSurfaceView {
    WallpaperGLSurfaceView(Context context) {
        super(context);
    }
    @Override
    public SurfaceHolder getHolder() {
        return getSurfaceHolder();
    }
    public void onWallpaperDestroy() {
        super.onDetachedFromWindow();
    }
}
```

正常情况下，GLSurfaceView 是被加入到一个 Activity 的视图层次中的，并且在那个 Activity 内的一个平面上进行渲染。因为动态壁纸使用它的方式有点不同，我们需要改变这个行为。GLSurfaceView 会调用 getHolder() 存取其 Activity 内的表面，因此，我们只需要重载 getHolder()，让它返回动态壁纸的渲染表面；我们通过调用 [WallpaperService.Engine].getSurfaceHolder() 实现这一点。

我们也需要改变清理 GLSurfaceView 的方式。GLSurfaceView 通常会在与其关联的窗口分离时清理它自己，正常情况下，当 Activity 的视图层次调用 onDetachedFromWindow() 时，它就知道该清理了。既然我们不会把 GLSurfaceView 添加到 Activity 中，我们就创建一个新的方法，称为 onWallpaperDestroy()，当壁纸被销毁时，我们就会调用这个函数，让 GLSurfaceView 知道是时候清理它自己了。

14.1.4 完成壁纸的引擎

我们自定义的 GLSurfaceView 已经就绪了，让我们完成这个壁纸引擎的剩余部分：

```
@Override
public void onVisibilityChanged(boolean visible) {
    super.onVisibilityChanged(visible);
    if (rendererSet) {
        if (visible) {
            glSurfaceView.onResume();
        } else {
            glSurfaceView.onPause();
        }
    }
}
@Override
public void onDestroy() {
    super.onDestroy();
    glSurfaceView.onWallpaperDestroy();
}
```

根据壁纸是否可见暂停和继续 GLSurfaceView 的执行，并在 onDestroy() 中进行清理。

现在，一切就绪了，我们的动态壁纸 Service 准备好接受测试了。

14.1.5 添加一个 XML 描述符

现在，我们需要添加一个 XML 文件，它会为 Android 系统描述我们的动态壁纸。为此，让我们在“/res”文件夹中创建一个名为“xml”的新文件夹，在这个新文件夹内部，创建一个新文件，命名为“wallpaper.xml”，并加入如下内容：

LiveWallpaper/res/xml/wallpaper.xml

```
<?xml version="1.0" encoding="utf-8"?>
<wallpaper xmlns:android="http://schemas.android.com/apk/res/android"
    android:thumbnail="@drawable/ic_wallpaper" />
```

这个 XML 文件为 Android 系统提供了有关这个动态壁纸的额外信息，比如，在动态壁纸列表中要显示什么样的图标。那个“drawable/ic_wallpaper”可以从本书的网站上下载，并放在“/res/drawablexhdpi”中[⊖]。

14.1.6 更新 Android Manifest 和从不支持的设备中移除

我们只需要给 AndroidManifest.xml 做两个变动，然后，就能测试这个新的动态壁纸了！打开 AndroidManifest.xml，并在 <uses-sdk> 后面加入如下代码：

LiveWallpaper/AndroidManifest.xml

```
<uses-feature android:name="android.software.live_wallpaper" />
<uses-feature android:glEsVersion="0x00020000" android:required="true" />
```

这两个标签让 Android 知道这个应用包含一个动态壁纸，并且它还需要 OpenGL ES 2.0 或以上版本。像 Google Play 一样的应用程序商店会用这些标签过滤搜索结果，并从不支持的设备中隐藏这些应用。

我们还需要加入那个动态壁纸 Service 的引用。在 <applicaton> 标签内，让我们在 <activity> 标签后面加入如下内容：

LiveWallpaper/AndroidManifest.xml

```
<service
    android:name=".wallpaper.GLWallpaperService"
    android:label="@string/app_name"
    android:permission="android.permission.BIND_WALLPAPER" >
    <intent-filter>
        <action android:name="android.service.wallpaper.WallpaperService" />

    </intent-filter>
    <meta-data
        android:name="android.service.wallpaper"
        android:resource="@xml/wallpaper" />
</service>
```

当我们的应用程序被安装到一个设备上的时候，这个 <service> 标签和它的内容会告

⊖ http://pragprog.com/book/kbogla/

诉 Android，那个 GLWallpaperService 是一个动态壁纸的 Service，而且它可以从 /res/xml/wallpaper.xml 中取出有关那个 Service 的额外信息。一旦一个应用程序问世，<service> 标签的属性“ android:name=”的值就不应该被改变了，因为，当应用程序升级时，改变其名字会导致用户的动态壁纸可能被重置[⊖]。

14.1.7　体验新的动态壁纸

既然我们已经完成了所有的实现部分，让我们构建并运行这个应用程序，看看实际的效果。当我们在 Eclipse 中运行应用程序时，会默认打开 Activity。但是，我们想检验动态壁纸，因此，让我们按下面这些步骤把这个新的动态壁纸添加到主屏幕上：

1. 首先，切换到设备的主屏幕上，然后，按住屏幕上的空白部分，直到弹出一个菜单。
2. 选择“Live Wallpapers”，然后，从弹出的列表中选择我们自己的动态壁纸。
3. 选择“ Set wallpaper”，应该可以看到那个动态壁纸出现在你的主屏幕上了，如图 14-1 所示。

图 14-2 显示了这些步骤的顺序。

图 14-2　添加动态壁纸的步骤

试着切换一下主屏幕，进入应用抽屉（所有已安装应用程序的列表），然后切回主屏幕，看看发生了什么！你有没有注意它的性能和表现如何？

14.1.8　随主屏幕滚动背景

你注意到的第一件事情可能是，当你在主屏幕上的不同页面间来回滑动时，动态壁纸没有移动。通过在 GLEngine 中实现 onOffsetsChanged() 方法，我们可以实现壁纸随主屏幕

⊖ http://android-developers.blogspot.ca/2011/06/things-that-cannot-change.html

滚动的效果，具体代码如下所示[⊖]：

LiveWallpaper/src/com/particles/android/wallpaper/GLWallpaperService.java
```
@Override
public void onOffsetsChanged(final float xOffset, final float yOffset,
    float xOffsetStep, float yOffsetStep, int xPixelOffset, int yPixelOffset)
    glSurfaceView.queueEvent(new Runnable() {
        @Override
        public void run() {
            particlesRenderer.handleOffsetsChanged(xOffset, yOffset);
        }
    });
}
```

当用户滚动主屏幕时，onOffsetChanged() 会被调用，并使用 x 和 y 的偏移值，其中每个值都在 0 和 1 之间。我们会把这些偏移值传递给渲染器，在渲染器中，我们会调用 handleOffsetsChanged()，并使用这些偏移值移动这个场景。这个方法还没有实现，因此，我们稍后就会实现它。

首先，把 onCreate() 中的 particlesRenderer 提升为 GLEngine 的一个成员变量，以便我们可以在 onOffsetsChanged() 中存取它。一旦完成了，让我们切换到 ParticlesRenderer，加入两个新的成员变量用来保存偏移值：

LiveWallpaper/src/com/particles/android/ParticlesRenderer.java
```
private float xOffset, yOffset;
```

让我们加入 handleOffsetsChanged() 的实现：

LiveWallpaper/src/com/particles/android/ParticlesRenderer.java
```
public void handleOffsetsChanged(float xOffset, float yOffset) {
    // Offsets range from 0 to 1.
    this.xOffset = (xOffset - 0.5f) * 2.5f;
    this.yOffset = (yOffset - 0.5f) * 2.5f;
    updateViewMatrices();
}
```

当 handleOffsetsChanged() 被调用时，我们会调整其偏移值，以使它们的范围变为从 −2.5 到 +2.5，而不是从 0 到 1，如此，我们可以从左边的粒子喷泉滚动到右边的粒子喷泉。通过调整 updateViewMatrices() 内部的 translateM() 调用，我们可以用这些偏移值更新视图矩阵，如下代码所示：

LiveWallpaper/src/com/particles/android/ParticlesRenderer.java
```
translateM(viewMatrix, 0, 0 - xOffset, -1.5f - yOffset, -5f);
```

我们可以把 translateM() 当作从一边向另一边移动整个场景的操作，因此，当用户向左移动主屏幕时，我们想让视点也向左移动，这意味着我们想要整个场景向右移动。我们用 translateM() 调用中的 x 和 y 部分分别减去 xOffset 和 yOffset 来实现这一点。

再运行一次这个动态壁纸，试着在主屏幕的不同页面之间切换一下。你现在应该看到，

⊖ http://developer.android.com/reference/android/service/wallpaper/WallpaperService.Engine.html#onOffsetsChanged(float, float, float, float, int, int)

背景和粒子喷泉也跟着移动了。

14.2 权衡系统的其余部分

用户往往对动态壁纸有相当高的期望，不仅使主屏幕更好看，还要节约设备资源。如果一个动态壁纸耗光了一半的电量，或者它使设备变慢了，用户常常会十分懊恼。他们还期望，当动态壁纸被放在后台时不要消耗电量，当他们回到主屏幕时，壁纸可以快速地恢复启动。

14.2.1 限制帧率

默认情况下，Android 会要求 OpenGL 渲染帧的速度与显示设备的刷新率一致，通常大概是 60 帧每秒。这会很快耗光电池的电量，因此，要减少对电池和系统性能的影响，我们可以加入一个帧率限制器，每秒钟渲染较少的帧数，从而减少电池的消耗。让我们回到 ParticlesRenderer，并在其顶部加入一个新的成员变量：

LiveWallpaper/src/com/particles/android/ParticlesRenderer.java

```
private long frameStartTimeMs;
```

这个变量以毫秒为单位记录每帧之间的间隔时间。让我们将 onDrawFrame() 的开始处修改为如下代码：

LiveWallpaper/src/com/particles/android/ParticlesRenderer.java

```
@Override
public void onDrawFrame(GL10 glUnused) {
    limitFrameRate(24);
```

每次渲染一个新帧时，我们都会调用 limitFrameRate() 把帧率限制为 24 帧每秒。让我们加入这个新方法的定义：

LiveWallpaper/src/com/particles/android/ParticlesRenderer.java

```
private void limitFrameRate(int framesPerSecond) {
    long elapsedFrameTimeMs = SystemClock.elapsedRealtime() - frameStartTimeMs;
    long expectedFrameTimeMs = 1000 / framesPerSecond;
    long timeToSleepMs = expectedFrameTimeMs - elapsedFrameTimeMs;

    if (timeToSleepMs > 0) {
        SystemClock.sleep(timeToSleepMs);
    }
    frameStartTimeMs = SystemClock.elapsedRealtime();
}
```

调用这个方法时，我们首先计算自从上一帧被渲染后逝去了多少时间，然后，计算在需要渲染下一帧之前还剩下多少时间。之后，我们让线程暂停那些时间，这样，我们就让它的速度降低到了需要的帧率。

比如，假定一个帧通常需要 20 毫秒去渲染，也就是 50 帧每秒，我们想要每帧降低到

40 毫秒，或者 25 帧每秒。当我们在 onDrawFrame() 中调用 limitFrameRate(25) 时，我们会发现，elapsedFrameTimeMs 大约是 20 毫秒，因为我们想要它降到 25 帧每秒，expectedFrameTimeMs 将是 40 毫秒。一旦我们用期望的帧时间减去其运行时间，我们就会得到 20 毫秒的 timeToSleepMs，然后，调用 SystemClock.sleep(timeToSleepMs) 休眠，直到需要绘制下一帧。

为了记录实际的帧率，让我们给类加入一个常量和两个另外的成员变量：

LiveWallpaper/src/com/particles/android/ParticlesRenderer.java

```
private static final String TAG = "ParticlesRenderer";
private long startTimeMs;
private int frameCount;
```

我们要在 onDrawFrame() 的开始处调用一个名为“logFrameRate()”的新方法记录帧率，将其加在 limitFrameRate() 调用的后面：

LiveWallpaper/src/com/particles/android/ParticlesRenderer.java

```
@Override
public void onDrawFrame(GL10 glUnused) {
    limitFrameRate(24);
    logFrameRate();
```

现在，我们只需要加入 logFrameRate() 的定义：

LiveWallpaper/src/com/particles/android/ParticlesRenderer.java

```
private void logFrameRate() {
    if (LoggerConfig.ON) {
        long elapsedRealtimeMs = SystemClock.elapsedRealtime();
        double elapsedSeconds = (elapsedRealtimeMs - startTimeMs) / 1000.0;

        if (elapsedSeconds >= 1.0) {
            Log.v(TAG, frameCount / elapsedSeconds + "fps");
            startTimeMs = SystemClock.elapsedRealtime();
            frameCount = 0;
        }
        frameCount++;
    }
}
```

这个方法会测量和记录每秒的帧数。

试试看，你最终得到了多少帧率。

14.2.2 保留 EGL 上下文

你可能已经注意到了，当我们离开主屏幕，稍后返回时，整个场景会被重新加载，粒子也会被重新初始化，并且经常能看到明显的延迟。这是因为当我们暂停 GLSurfaceView 时，默认情况下，它会释放所有的 OpenGL 资源，当我们稍后恢复运行时，渲染表面会被重新创建，又将调用 onSurfaceCreated()，它会要求我们重新加载所有的数据。

实际上，GLSurfaceView 在后台使用了一个 API 去调用 EGL 本地平台图形接口。当我们用 GLSurfaceView 初始化 OpenGL 时，一个 EGL 上下文就会被创建，而且 OpenGL 可能

用到的纹理、缓冲区和其他资源都链接到这个上下文。在较早的设备上，这些上下文是有限的资源[⊖]，因此，无论何时我们让 GLSurfaceView 暂停时，它通常都会释放这个上下文，并清除所有的 OpenGL 资源。

许多较新的设备有能力让多个上下文同时存在，从 Android 3.0 Honeycomb 版本开始，通过调用 GLSurfaceView.setPreserveEGLContextOnPause(true)，我们可以保留 EGL 上下文。如果上下文可以被保留，那么，我们的动态壁纸就能立即恢复运行，而不必重新加载所有的 OpenGL 数据。

让我们更新 GLWallpaperService.GLEngine 中的 onCreate() 方法，在 glSurfaceView.setEGLContextClientVersion() 调用之后加入如下代码：

```
LiveWallpaper/src/com/particles/android/wallpaper/GLWallpaperService.java
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    glSurfaceView.setPreserveEGLContextOnPause(true);
}
```

这段代码首先检查 Android 的当前版本是 Honeycomb 还是其后续的版本，然后，我们让 GLSurfaceView 在暂停时保留 EGL 上下文。只要我们用版本检查封装了这段代码，即使这段代码运行在早期的 Android 版本上，我们也不必使用 Java 的反射来调用那个方法。

再次运行这个动态壁纸，试着切换到应用抽屉，再返回主屏幕。现在，你应该发现壁纸恢复得更快了，粒子也没有被重置。

14.3 小结

恭喜你，又完成了另外一个项目，并创建了你的第一个动态壁纸！我们学习了如何把粒子场景放到主屏幕上，还了解了如何通过降低帧率减少资源和电池消耗，同时，通过在暂停和恢复运行时保留 EGL 上下文以保持其效率。

当我们构建这个项目时，也讲述了许多 OpenGL 的核心内容，比如，如何使用顶点和索引缓冲区，如何用深度缓冲区消除隐藏面，以及如何使用混合和光照技术增添效果。你并不总需要使用所有这些技术，而事实上，Android 市场上很多流行的游戏只使用简单又漂亮的二维图形。不过，现在你有这些技术在手，就能以此为基础了解更多的技术，同时，灵活地利用这些技术和相应的工具能帮助你实现你的艺术创想。

14.4 练习

我们已经在本章中讲述了许多动态壁纸的基础，但是还有许多唾手可得的内容需要学习，让我们从下面的练习开始：

⊖ http://www.khronos.org/egl

- 为了优化性能，让我们改变初始化粒子系统的方法，把总粒子数从 10 000 减少到一个低很多的数值。一旦完成了，让我们做进一步的优化，去掉落到地形下面的粒子，实现的方法之一是在数组中用最后激活的粒子替换每个要被去掉的粒子。
  当你做这些改动时，尝试更新它的行为，以使粒子系统在 onSurfaceCreated() 被第二次调用的情况下不会被重置。
- Android 动态壁纸支持一个额外的设置（Settings）Activity，它可以这样被配置，给 /res/xml/wallpaper.xml 中的 <wallpaper> 标签添加一个 android:settingsActivity 属性。用一个简单的方法添加一个设置 Activity，改变动态壁纸每秒钟的帧渲染率[⊖]。
- Android 4.2 Jelly Bean 版本有个新特性，Android 支持 Daydream，这是设备空闲或充电时的一个交互式屏幕保护器。尝试把这个烟花的 OpenGL 场景实现为一个 Daydream[⊖]。

还有另外一个用 OpenGL 实现动态壁纸的方法，重新实现 GLSurfaceView 的大部分内容，而不是继承它。你可以通过下面链接的在线资源了解更多有关这个方法的内容，并实现它[⊜]。

当你准备好了，让我们前往下一章，采取下一步行动。

⊖ http://www.learnopengles.com/how-to-use-opengl-es-2-in-an-android-live-wallpaper/

⊖ http://android-developers.blogspot.ca/2012/12/daydream-interactive-screen-savers.html

⊜ http://www.rbgrn.net/content/354-glsurfaceview-adapted-3d-live-wallpapers
https://github.com/GLWallpaperService/GLWallpaperService/blob/master/GLWallpaperService/src/net/rbgrn/android/glwallpaperservice/GLWallpaperService.Java
http://www.learnopengles.com/how-to-use-opengl-es-2-in-an-android-live-wallpaper/

第 15 章 Chapter 15

采取下一步行动

我们终于到了最后一章！首先，让我们跳出 Android 看两个方法。我们要浏览两个跨平台的框架和使用它们的方法，同时评估每个方法的利与弊。

我们也会花些时间浏览一些 OpenGL 的较高级的内容，它们超出了本书所讲的范围，包括多重纹理（multitexturing）、各向异性过滤（anisotropic filtering）和反走样（antialiasing, AA），以及在二维游戏和应用中相当有用的技术，比如广告牌（billboarding）和纹理图集（texture atlas）。我们还会快速阅览 OpenGL ES 3.0 会有哪些新增内容。

当我们学习这些内容时，你也可以参考这些技术的一些样例实现，下载本书的代码文件，导入 TakingTheNextStep 文件夹中的项目[⊖]。

15.1 展望其他平台

如果某一天你不想仅限于 Android 平台，要扩大你的影响力，那么你一定要考虑一下可移植性和跨平台编码的问题。很多主要的平台上都完美支持 OpenGL ES 2，包括 iOS 和使用 WebGL 的网络；然而，这些平台通常需要用其他编程语言。在 iOS 上，Objective C 是主要的开发语言；而在网络上，JavaScript 是主要的开发语言。

如果为每个平台都重写代码将需要更多的资源，所以跨平台开发就非常受欢迎了。一个极端，你可以用一种特有的环境，在所有的平台上使用同一种语言，并且为你抽象所有的东西；另一个极端，你也可以自己做所有的事情，构建你自己的框架。而占领中间地带的是开源框架，例如 libgdx（由 Mario Zechner 创建）和 Cocos2d-x（由 Zhe Wang 创立），它

⊖ http://pragprog.com/book/kbogla/

们允许你支持多个平台，而不用重写太多的代码[⊖]。

当考虑跨平台开发时，有两个要点需要记住：第一，你要确保它花费的工作不比为每个平台维护一套独立的代码更多；第二，你要避免落入陷阱，把一切都按最低的共同标准裁剪，导致在所有的平台上都体验不佳。

我们要快速地看一下如何使用 libgdx 和 Cocos2d-x 作为代码的框架，以及每种方法的利弊。

15.1.1 利用 Libgdx 在 Java 上构建

Libgdx 是一个流行的开源游戏开发库，它使用 Java，并且支持 OpenGL ES 1 和 OpenGL ES 2，除了对 Android 的大力支持，它也支持其他平台，它为每个平台使用一个不同的后端以及一个翻译 Java 代码的翻译层。为了支持网页，它使用一个 Google 网页工具套件把 Java 编译为 JavaScript，为了支持 iOS，它使用一个名为 ikvmc 的工具把 Java 编译为 .NET 中间语言，然后，通过使用一个 .NET 编译器，比如 Xamarin.iOS，它会被变为一个 iOS 库。

由于使用 Java 作为主要的编程语言，你可以避免手动的内存管理、栈和堆崩溃、未定义的行为以及其他相关的问题，在你使用其他编程语言时，如 C 或 C++，这些问题都可能出现。另一方面，当涉及垃圾回收和多个翻译层时，也比较难以控制其性能。

15.1.2 利用 Cocos2d-x 在 C 和 C++ 上构建

不使用 Java，你也可以使用 C 或者 C++，几乎所有的平台都有它们的编译器。Cocos2d-x 是一个流行的游戏开发框架，它使用 C++，支持 OpenGL ES 2，目标是为了用最少的抽象和翻译层支持 Android、iOS 和其他平台；正如其名字所暗示的，Cocos2d-x 也为二维游戏做了优化。因为 C 和 C++ 直接编译为机器代码，这个方法在那些寻求最佳性能和控制的游戏开发者中非常流行（Java 也有不俗的表现，但 Dalvik 虚拟机在生成快速 JIT 代码方面不如桌面 JVM 好）。

这个方法的主要缺点之一是，Android 对 C 和 C++ 的支持肯定不如对 Java 的支持好，同时，你也打开了潘多拉的魔盒，如栈和堆崩溃、未定义的行为以及其他隐晦的问题，尽管最新的 GCC 和 Clang 的版本有很好的调试工具帮助你克服这些，但它们仍会降低程序的稳定性。目前为止，C 和 C++ 仍然很难移植到网页上，尽管可能并不总是这样 :Google 正在用本地客户端（Native Client）帮助人们把本地语言放在网页上，甚至也有编译器可以把 C++ 编译为 JavaScript，然后，就可以在浏览器上运行 C++ 了。

⊖ 分别是 https://code.google.com/p/libgdx/ 和 http://www.cocos2d-x.org/

15.2　了解更高级的技术

现在，我们已经讲了这么多了，让我们花点时间浏览一些在 OpenGL 中渲染事物的更高级的方法。

15.2.1　多重纹理

回到第 7 章，我们了解到，我们可以很容易地应用纹理技术，只要指定一组纹理坐标，并使用纹理单元把实际的纹理传递进去。OpenGL ES 2.0 实际上提供多个纹理单元的支持，大多数 GPU 支持至少两个。这可以被用来在两个文件间进行混合，从而产生各种各样的效果。在样例项目中，我们使用多重纹理（multitexturing）绘制了地形，它在低海拔处用了草地纹理，在高海拔处用了石头纹理，并把二者混合。

15.2.2　各向异性过滤

在同一章中，我们还学习到，OpenGL 支持许多纹理过滤模式，包括最近邻过滤、没有贴图的双线性过滤和有贴图的双线性及三线性过滤。使用贴图技术提高了性能和质量，但标准贴图的一个问题是，它针对正对面的纹理渲染进行了优化，而没有针对任意角度优化。如果我们看看样例项目中的地面，就可以看到，它在远处被过分地涂抹了。我们可以一直关闭贴图，但是，这不仅降低其性能，而且使得场景中的远处区域变得非常跳跃和嘈杂。

通过使用各向异性过滤（Anisotropic Filtering），我们可以提高任意角度的纹理质量，它是一种考虑了观察角度的纹理过滤类型。在图 15-1 中，你可以看到一个三线性过滤和 15x 的各向异性过滤的对比，15x 是 Nexus 7 的 Tegra 3 芯片支持的最大值。通过各向异性过滤，从一个倾斜的观察角度看，纹理看上去好多了，少了很多的涂抹；需要权衡的是，其性能可能会降低。较高层次的各向异性过滤会过滤更多的倾斜表面，从而提高质量。

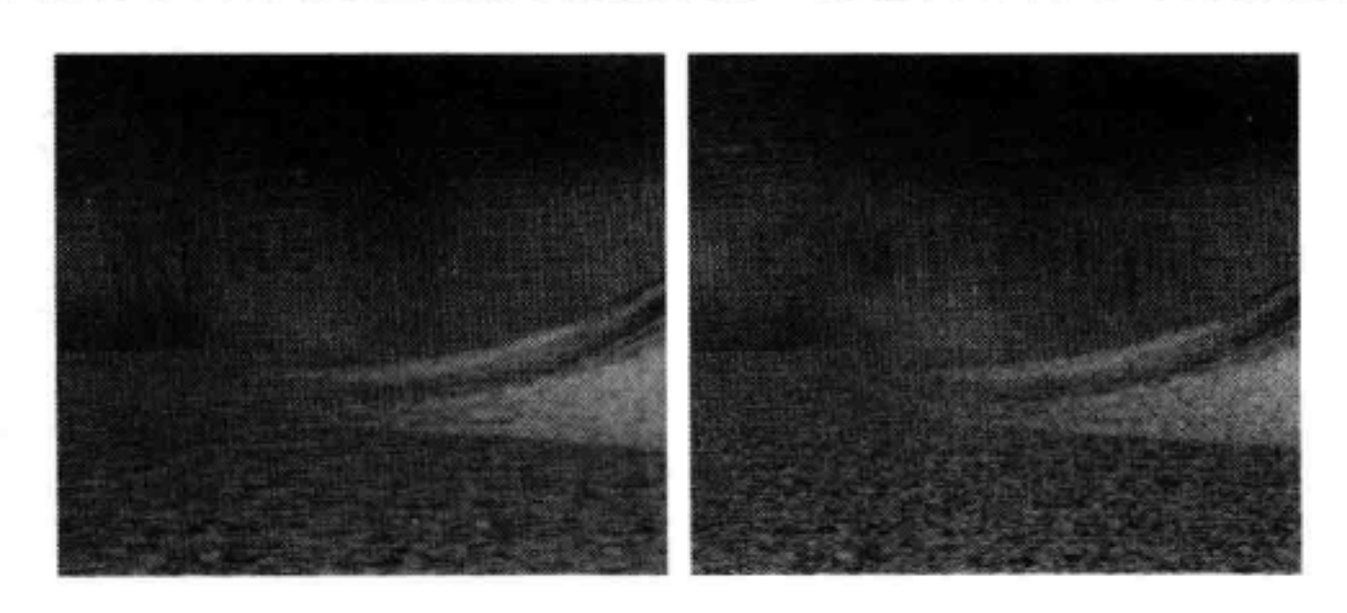


图 15-1　三线性过滤与 15x 各向异性过滤的对比

OpenGL 把实际的实现留给硬件来决定，而且，并不是所有的 GPU 都支持各向异性过滤。在样例项目中，你会看到怎样检查其是否支持，如果支持，如何使能各向异性过滤的最大量。

15.2.3 反走样

反走样（antialiasing）是一种用来减少走样的渲染技术，它在三维场景中常常表现为“锯齿”和三角形的边的形式[⊖]。有很多平滑处理一个场景的方法，每个方法都有其自身的优缺点。

在过去，应用程序常常这样简单地消除场景中的锯齿，接连多次渲染同一个场景，每次渲染都用一个稍稍不同的透视投影。通过把这些渲染混合在一起，反走样的目的就达到了。在 OpenGL ES 中，一些 GPU 可以采用多重采样反走样（multisample antialiasing）技术通过 OpenGL 本身进行反走样，有些 NVIDIA 芯片组也支持覆盖反走样（coverage antialiasing）。在图 15-2 中，你可以看到一个常规渲染与使用 2x 覆盖模式的 NVIDIA 的覆盖反走样的对比的例子。

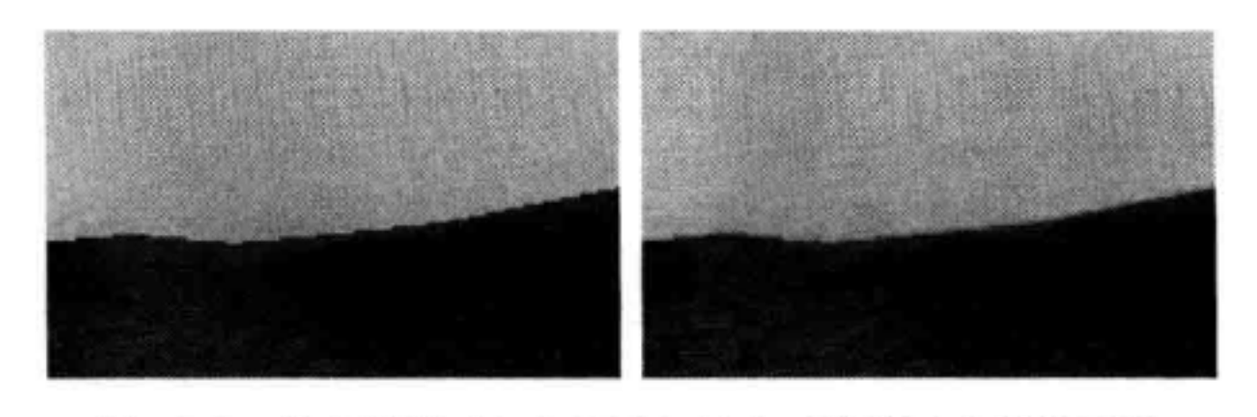


图 15-2 常规渲染与 NVIDIA 的 2x 覆盖反走样的对比

随着高分辨率和小像素在移动设备上的盛行，反走样已经不像过去在桌面平台那么重要了。而且，反走样消耗了许多额外的资源，却在图像质量上收效甚微。然而，在某些应用程序里，它是有意义的。在样例项目中，我们采用了 Google 的游戏开发者会议（Game Developers Conference，GDC）2011 演示中的代码（一个很好的例子，采用了不同的 OpenGL ES 2 技术）向你展示了如何在你自己的项目中是启用反走样技术，参见图 15-3[⊖]。

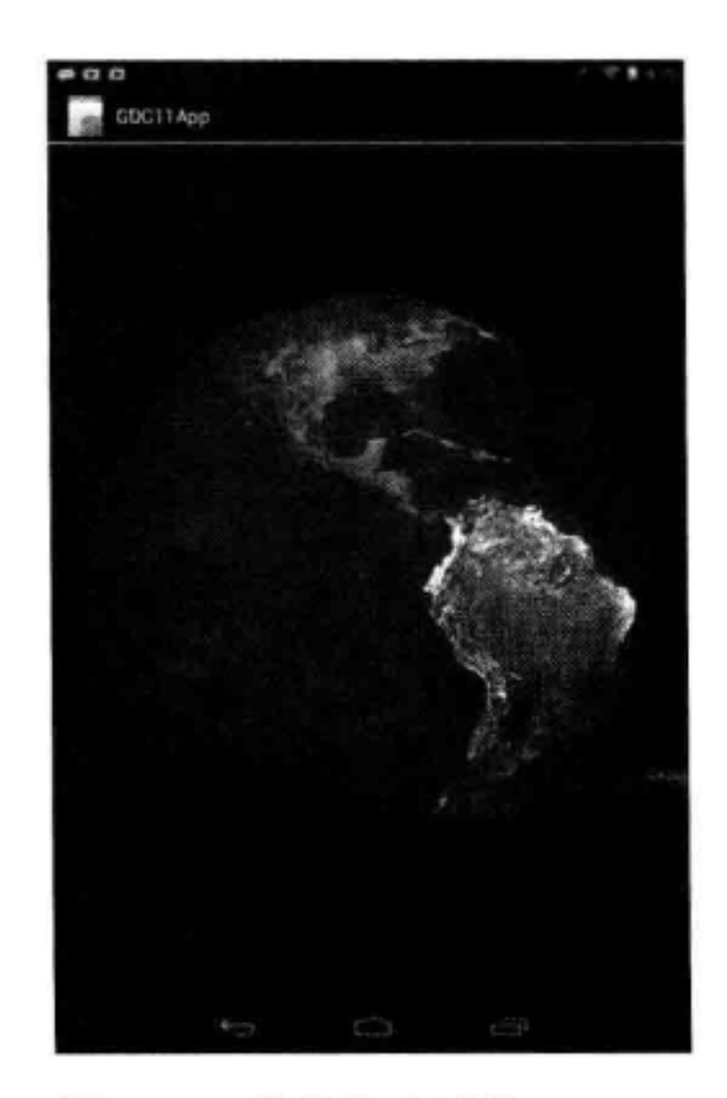


图 15-3 稍微修改过的 GDC 11 演示程序

15.2.4 公告牌和其他二维技术

在第 10 章中，我们学习了如何使用点精灵把粒子渲染为二维纹理。公告牌（billboarding）是另一种在三维场景中渲染二维精灵的方法，它的工作原理是，绘制两个三角形组成的矩形四边形，并在这个有透明区域的四边形上渲染纹理，不管相机面向哪个方向，要始终让这个四方形朝向相

⊖ http://en.wikipedia.org/wiki/Aliasing

⊖ https://code.google.com/p/gdc2011-android-opengl/

机。这对同样类型的事物很有效，比如，我们使用点精灵的那些粒子，其优势是没有点大小的限制。

对于其他一些事物，比如草丛，一个较好的技术是使用同样的二维四边形，但是，不要改变它们的朝向去面对相机，相反，我们要保持它们的朝向不变，我们把它们中的几个放在一起，以使它们相互重叠。这可以在游戏中看到良好的效果，比如 Assassin 的 Creed 3，在丛林中穿梭的时候。这个技术对于今天的移动电话和平板可能有点沉重，但是，随着像 Ouya（一个基于 Android 的游戏机）一样的设备的出现，我们可以期望这类技术会用得越来越多。

15.2.5 使用纹理图集

切换纹理代价很高，因此，为什么不把许多不同的纹理放进一个巨大的纹理中，然后为不同的物体使用这个纹理的不同部分呢？这个巨大的纹理叫做纹理图集（texture atlas），对于特定类型的游戏，使用它可以使性能得到提升，尤其是基于二维精灵和基于图块的游戏。对于游戏中的自定义字体的渲染，使用纹理图集也能提升其性能。

纹理图集也有一些不足：MIP 贴图会把邻接的子纹理混合在一起，并且有时候把所有纹理拼装到纹理图集上也很痛苦。有很多工具可以帮助你克服这点，我们曾在 15.1.1 节中讨论过一个跨平台框架——libgdx，它内置了其中一个工具[⊖]。

15.2.6 介绍 OpenGL ES 3.0

在 2012 年 8 月，Khronos 工作组宣布了 OpenGL ES 3.0 规范的发布，它是构建在 OpenGL ES 2.0 上的一个迭代提高。这个新的规范还没有来得及铺开，且目前还没有出现在 Android 设备的统计板上，但迟早，设备将开始支持 OpenGL ES 3.0。下面是我们期望的一些新特性[⊖]：

- 纹理技术得到了极大的提高，支持许多新的纹理类型和特性。高质量的 ETC2/EAC 纹理压缩被作为标准特性内置了，立方体贴图现在也保证是无缝的[⊜]。
- 渲染管道现在支持多个渲染目标以及高级特性，如遮挡查询和实例化绘制。
- 这个规范包括一套扩充的纹理和缓冲区格式，减少了不同实现之间的差异。

这些新特性提高了 OpenGL ES 2 的能力，引入了一些先进功能，使开发者更容易从桌面 OpenGL 移植代码。默认支持或者通过 OpenGL 扩展，你的设备设置可能已经支持其中一些特性了（比如无缝的立方体贴图）。

⊖ https://code.google.com/p/libgdx/wiki/TexturePacker

⊖ http://www.khronos.org/news/press/khronos-releases-opengl-es-3.0-specification

⊜ http://en.wikipedia.org/wiki/Ericsson_Texture_Compression#ETC2_and_EAC

15.3 与世界分享你的艺术视野

有很多内容我们还没有讲述。有各种各样高级的效果可以通过 OpenGL ES 2 的功能实现，比如镜面光、雾、反射、阴影和各种后期处理效果，如高动态范围照明和景深。

当前在移动世界里还没有看到太多这些比较高级的效果，一部分原因是现有的大部分 GPU 还没有足够的能力，另一部分原因是它们对于那些典型的测卷轴游戏或三维跑步游戏是大材小用了。这会随着时间改变的，正如我们看到的，Android 在其他领域被越来越多地使用，比如游戏机，另外，电话和平板本身也越来越强大。OpenGL ES 2 带来了很多能量，我们只是触及了皮毛。

我们现在已经一起到达了冒险之旅的终点。OpenGL ES 2.0 是一个有很大潜力的、功能强大的图形库，而且你现在有工具可以在其上进行绘制了。当你依靠自己继续这个旅程时，要勇于冒险、保持好奇心，与世界分享你的艺术视野。

附录 A Appendix A

投影背后的矩阵数学

在 OpenGL 中有两种主要的投影类型：正交和透视。这两个投影都是用来把坐标从虚拟世界坐标变换为归一化设备坐标，归一化设备坐标的每个轴的取值范围是从 -1 到 1。投影定义了什么能被看到，以及会以多大呈现。

在继续学习之前，你可能想再次读一读 5.3 节。

A.1 正交投影背后的数学

我们将从正交投影开始，因为它们比较容易理解。正交投影把三维世界的部分映射到屏幕上，它是以这样一种方式实现这些的，不管物体是远还是近，它们看上去的大小都是一样的。基于这个原因，这种投影类型非常适于实现二维游戏和美术。

A.1.1 创建单位正交投影

下面是正交投影矩阵的基本定义：

$$\begin{bmatrix} \frac{2}{\text{right}-\text{left}} & 0 & 0 & -\frac{\text{right}+\text{left}}{\text{right}-\text{left}} \\ 0 & \frac{2}{\text{top}-\text{bottom}} & 0 & -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} \\ 0 & 0 & \frac{-2}{\text{far}-\text{near}} & -\frac{\text{far}+\text{near}}{\text{far}-\text{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

使用这个给定的矩阵，所有左与右之间、上和下之间以及远和近之间的坐标都会被映射到归一化设备坐标，并且在这个范围内的一切都在屏幕上可见。

让我们实际看一下这个矩阵：作为我们的第一个例子，我们将创建一个单位正交投影。让我们构建这个矩阵，把 −1 传递给左、下和近，把 +1 传递给右、上和远。经过简单替换，我们得到下面的矩阵：

$$\begin{bmatrix} \frac{2}{1--1} & 0 & 0 & -\frac{1+-1}{1--1} \\ 0 & \frac{2}{1-1} & 0 & -\frac{1+-1}{1--1} \\ 0 & 0 & \frac{-2}{1+1} & -\frac{1+-1}{1--1} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

经过简单的负值转换后，我们得到下面的结果：

$$\begin{bmatrix} \frac{2}{1+1} & 0 & 0 & -\frac{1-1}{1+1} \\ 0 & \frac{2}{1+1} & 0 & -\frac{1-1}{1+1} \\ 0 & 0 & \frac{-2}{1+1} & -\frac{1-1}{1+1} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

下一步是运用加减运算：

$$\begin{bmatrix} \frac{2}{2} & 0 & 0 & -\frac{0}{2} \\ 0 & \frac{0}{2} & 0 & -\frac{0}{2} \\ 0 & 0 & \frac{-2}{2} & -\frac{0}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

最后，我们简化这些分数，得到最终的矩阵：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

它看起来几乎与单位矩阵完全一样！之所以是这样，是因为归一化设备坐标在每个轴上的范围是从 −1 到 1，因此，当我们把 −1 和 1 也传递进去作为我们的范围时，实际上要

求一个保持其坐标不变的正交投影，就像单位矩阵一样。主要的区别是，它的 z 轴是反转的，这完全是由于惯例，正如我们在 5.4.1 节中解释的。

A.1.2 创建常规正交投影

作为下一个例子，让我们创建一个有实际用途的正交投影。如果我们不喜欢负数，想把所有的坐标指定在范围 [0, 1] 内，而不是 [−1, 1]，怎么办？这意味着，左、下和近都是 0，而右、上和远都是 1。

当我们调用 orthoM() 时，我们要表达的就是，需要一个矩阵，对于其 x、y 和 z 分量，它会把 [0, 1] 映射到范围 [−1, 1] 上。这就是初始矩阵看起来的样子：

$$\begin{bmatrix} \frac{2}{1-0} & 0 & 0 & -\frac{1+0}{1-0} \\ 0 & \frac{2}{1-0} & 0 & -\frac{1+0}{1-0} \\ 0 & 0 & \frac{2}{1-0} & -\frac{1+0}{1-0} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

这一次我们不会遍历所有的简化步骤。让我们看看最后的结果：

$$\begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 2 & 0 & -1 \\ 0 & 0 & -2 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

要证明这个矩阵的确把范围 [0, 1] 内的坐标变换为范围 [−1, 1]（记住 z 轴的那个特殊反转），让我们用几个不同的向量实验一下。首先，我们从一个最小范围的向量开始，其所有分量都是 0：

$$\begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 2 & 0 & -1 \\ 0 & 0 & -2 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$$

正如我们所期望的，所有分量都映射到归一化设备坐标的 −1 了。现在，我们将试一下中间的向量：

$$\begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 2 & 0 & -1 \\ 0 & 0 & -2 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0.5 \\ 0.5 \\ -0.5 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

结果也是在归一化设备坐标的中间，其 x、y 和 z 都变为 0 了。现在，让我们试一下最大范围为 1 的向量：

$$\begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 2 & 0 & -1 \\ 0 & 0 & -2 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

这次，所有分量也都在归一化设备坐标的最大范围上了。

记住，在把向量与矩阵相乘之前，我们在向量中指定了一个负的 z 值，而不是一个正的 z 值；尽管我们曾指定近值为 0，远值为 1，但实际上不得不传递从 0 到 −1 的范围。这只是为那个反转的 z 轴所做的调整，正如我们在 5.4.1 节中所描述的。

A.2 透视投影背后的数学

透视投影的工作有点不同。它们也把虚拟世界的部分映射到屏幕上，但它们使用透视实现这点：物体离得越远，它在屏幕上呈现得越小。这个投影矩阵不能由它自己实现这些，因此，它使用第四个分量——w，并与透视除法一起使用。你可以在 6.2.2 节中阅读更多的内容。

下面是一个非常基本的透视投影矩阵。这个矩阵假定一个右手坐标空间（参见 5.4.1 节），其近平面从 z 值为 −1 处开始。作为一个快速练习，看看你是否能猜到远端位于哪里，记住，OpenGL 会把每个分量都除以 w：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & -2 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

让我们看看这个矩阵内发生了什么：

- 这个矩阵的前两行只是简单地复制 x 和 y 分量，而没有修改它们。
- 这个矩阵的第三行复制 z 分量，同时把它反转。第四列的 −2 会乘以一个向量的 w 值，它默认是 1，因此，矩阵的这行最终会把最后的 z 设为 −1 * z − 2。比如，值为 −1 的 z 会被设为 −1，值为 −2 的 z 会被设为 0，值为 −3 的 z 会被设为 1，等等。我们为什么使用 −2，而不是其他数值呢？简单来说，第三列中的 −1 和第四列中的 −2 就是把视椎体的近平面映射到归一化设备坐标的近平面处的数值（−1 会映射到 −1）；并且，因为同样的值为 −1 的 z 会创建一个值为 1 的 w，在透视除法之后 z 值仍保持 −1。

作为另外一个练习，我们希望为此得到一个数学上的正式证明。关于这些值是怎样

演算出来的，songho.ca 有个很好的总结[⊖]。然而，要理解如何使用投影矩阵，正式的证明是不需要的。

- 第四列把最后的 w 值设为负 z。一旦 OpenGL 执行了透视除法，这会起到把较远处的物体缩小的效果。

当我们把这个矩阵与视椎体的近端上的一个点相乘时，让我们看看会发生什么：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & -2 \\ 0 & 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix}=\begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

这个点的 z 值为 −1，其结果的 z 值为 −1，w 值为 1。让我们再看看两个点，每个都比前一个远：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & -2 \\ 0 & 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} -1 \\ -1 \\ -2 \\ 1 \end{bmatrix}=\begin{bmatrix} -1 \\ -1 \\ 0 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & -2 \\ 0 & 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} -1 \\ -1 \\ -3 \\ 1 \end{bmatrix}=\begin{bmatrix} -1 \\ -1 \\ 1 \\ 3 \end{bmatrix}$$

随着点变得越来越远，它们的 z 和 w 也变得原来越大。

除以 w

我们已经探讨了透视投影魔法的第一部分：定义了一个矩阵，它会创建一个随距离增加而增加的 w 值。然而，还剩下一步：透视除法。OpenGL 把每个分量都除以 w 之后，最终会得到下面的结果：

$$\begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix}=\begin{bmatrix} -1/1 \\ -1/1 \\ -1/1 \end{bmatrix}=\begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

$$\begin{bmatrix} -1 \\ -1 \\ 0 \\ 2 \end{bmatrix}=\begin{bmatrix} -1/2 \\ -1/2 \\ 0/2 \end{bmatrix}=\begin{bmatrix} -0.5 \\ -0.5 \\ 0 \end{bmatrix}$$

⊖ http://www.songho.ca/opengl/gl_projectionmatrix.html

$$\begin{bmatrix}-1\\-1\\1\\3\end{bmatrix}=\begin{bmatrix}-1/3\\-1/3\\1/3\end{bmatrix}=\begin{bmatrix}-0.3\overline{3}\\-0.3\overline{3}\\0.3\overline{3}\end{bmatrix}$$

让我们用一些更远的点试一下。下面是 z 值为 −100 的点：

$$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&-1&-2\\0&0&-1&0\end{bmatrix}\begin{bmatrix}-1\\-1\\-100\\1\end{bmatrix}=\begin{bmatrix}-1\\-1\\98\\100\end{bmatrix}=\begin{bmatrix}-1/100\\-1/100\\98/100\end{bmatrix}=\begin{bmatrix}-0.01\\-0.01\\0.98\end{bmatrix}$$

这是 z 值为 −10 000 的点：

$$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&-1&-2\\0&0&-1&0\end{bmatrix}\begin{bmatrix}-1\\-1\\-10000\\1\end{bmatrix}=\begin{bmatrix}-1\\-1\\9998\\10000\end{bmatrix}=\begin{bmatrix}-1/10000\\-1/10000\\9998/10000\end{bmatrix}=\begin{bmatrix}-0.0001\\-0.0001\\0.9998\end{bmatrix}$$

通过这种类型的投影矩阵，视椎体的远端实际上处于无限远处。Eric Lengyel 把这称为“无限投影矩阵（infinite projection matrix）”[⊖]。由于硬件精度的限制，不管 z 多远，在归一化设备坐标中，它都是接近而不能完全匹配远平面的 1。

你可能也注意到了，OpenGL 把一切都除以 w 之后，输入 z 和输出 z 之间的关系不再是线性的了。随着输入 z 值从 −1 变到 −2，输出 z 从 −1 增加到了 0，但当输入 z 值从 −10000 变为 −20000 时，其对输出 z 值的影响就很小了。当我们在场景中不同距离处渲染物体的时候，这种非线性关系具有重要意义，我们曾在 12.5.1 节中讲述了更多的细节。

⊖ http://www.terathon.com/gdc07_lengyel.pdf

附录 B Appendix B

调　试

当我们开发 OpenGL 应用程序时，迟早都会碰到一些奇怪的问题需要我们去调试。也许物体没有按我们期望的方式被渲染；或者应用由于段错误而崩溃[⊖]。在本附录中，你会接触到许多不同的解决这些问题的方法，以及一些需要小心的常见陷阱。

B.1 用 glGetError 调试

使用 OpenGL，排除问题的方法之一就是通过检查我们是否执行了 OpenGL 不期望的调用，或者传递了错误的参数。为此，我们可以通过调用 glGetError() 获得 OpenGL 的错误标记状态。以下是一些最常用的错误码：

GL_INVALID_ENUM

我们给一个函数传入一个值，而这个值是不被接受的。比如，我们可能传递了一个 GL_FLOAT 类型的值给 glEnable()，或者其他对于那个函数没有意义的值。

GL_INVALID_VALUE

我们给一个函数传递的其中一个数值参数超出了其取值范围。

GL_INVALID_OPERATION

针对当前给定的 OpenGL 状态，我们尝试执行一个无效操作。

除了 glGetError() 之外，我们还有更多特殊的方法可以用来查看错误，比如我们在 3.2.3 节中学过的方法。为了在接近错误发生的地方捕捉到它们，经常调用 glGetError() 是

⊖ http://en.wikipedia.org/wiki/Segmentation_fault

个不错的主意。因为一次可以记录多个错误标识，OpenGL 帮助手册也推荐在一个循环中调用它，直到它返回 GL_NO_ERROR。

让我们用一个例子测验一下这个方法。打开本书的其中一个项目，并在 onSurfaceCreated() 的开始处加入一个 glEnable(GL_TEXTURE_2D) 调用。在 glEnable() 调用之后，加入另一个 glGetError() 调用，并检查其返回值。当你运行这个应用程序时，应该看到 glGetError() 会返回 GL_INVALID_ENUM，这是因为 GL_TEXTURE_2D 不是 OpenGL ES 2.0 中的 glEnable() 能接受的常量之一。

在 OpenGL ES 1 中，通过在渲染器回调函数中提供的封装的接口，我们可以让 GLSurfaceView 封装每一个 OpenGL 的调用。不幸的是，这不适用于 OpenGL ES 2.0，因为我们直接调用了 GLES20 类中的静态方法。我们可以用一种变通的方法解决这个问题，创建我们自己的封装类，封装每个 OpenGL 的函数，并在每个函数后面循环调用 glGetError()。如果我们使用那个类，当 OpenGL 错误发生时，我们就知道它很接近问题的根源。

B.2 使用 Tracer for OpenGL ES

从 Android 4.1 开始，OpenGL 应用程序也能使用工具“Tracer for OpenGL ES”[⊖]。这个工具可以捕捉发送给每个被渲染的帧的 OpengGL 命令流。你可以按下面这些步骤运行跟踪器：

- 切换到跟踪器视图，在 Eclipse 里，选择“Window->Open Perspective->Other”，然后选择“Tracer for OpenGL ES”。
- 单击工具条上的跟踪捕捉按钮。
- 一个跟踪选项窗口会出现，如图 B-1 所示。选择要调试的设备或模拟器实例（记住，在设备上跟踪的结果会更准确和中肯），输入要启动的包名和 Activity 组件，然后输入目标跟踪文件。
- 接下来，你会看到一个显示跟踪进度的窗口。一旦跟踪完成，你要选择“Stop Tracing”。你会看到一个有很多详细内容的窗口，显示所有的跟踪信息，如图 B-2 所示。

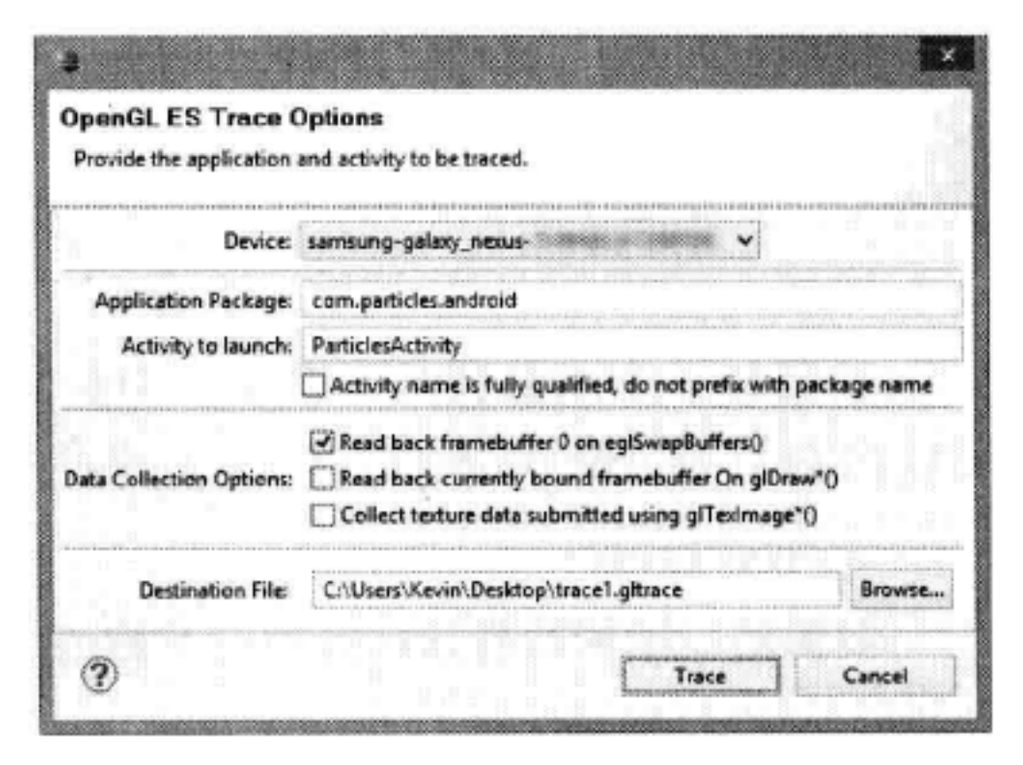


图 B-1 OpenGL ES 跟踪选项

⊖ http://developer.android.com/tools/help/gltracer.html 和 http://www.teamblubee.com/2012/09/22/androidopenggland-slow-rendering/

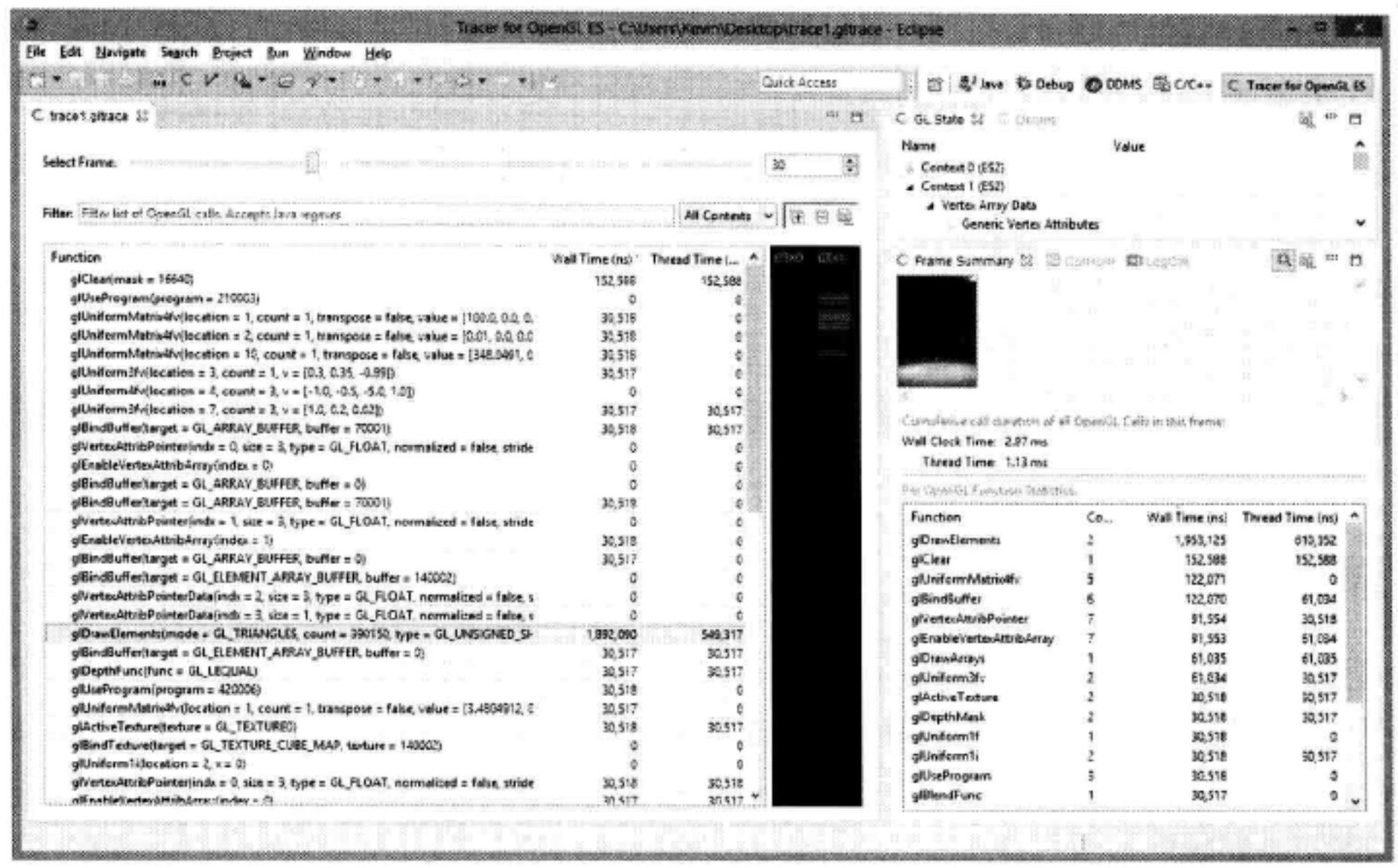


图 B-2 OpenGL ES 跟踪结果

跟踪结果包含一个帧选择器，它用来查看对于一个给定帧的所有信息。一旦一个帧被选中了，你就能看到有关那一帧的许多细节，包括 OpenGL 调用的列表、每个调用发生的时间、帧的快照、每个不同类型的 OpenGL 调用所花的时间和当前的 OpenGL 状态。你也许不能相信那个执行时间，因为我注意到它们有时会被严重低估。

除了 Android SDK 所带的 OpenGL 跟踪器，许多 GPU 供货商也提供他们自己的工具；例如，NVIDIA 为 Tegra GPU 提供 PerfHUD ES[⊖]，PowerVR 为 PowerVR GPU 提供 PVRTrace[⊜]。这些工具通常可以提供更强大的功能；比如，PerfHUD ES 可以提供具体的着色器编译信息，而且让你实时编辑并重新编译着色器，它甚至可以一步一步地向你展示每一帧是怎样被绘制的[⊜]。

Joe 问：为什么我的跟踪器不工作

只要你的设备运行 Android 4.1 或者后续版本，跟踪器就应该工作；然而，如果你有一台运行 Android 4.2.2 的 Nexus 设备，那很不幸，你可能会碰上问题，因为跟踪器被破坏了。如果这发生了，查阅一下下面链接的缺陷报告，因为读到这些的时候，问题可能

⊖ https://developer.nvidia.com/nvidia-perfhud-es

⊜ http://www.imgtec.com/powervr/insider/pvrtrace.asp

⊜ http://www.curious-creature.org/2012/12/01/android-performance-case-study/

已经被修复了[㊀]。

就现在而言，你也可以变通解决这个问题，把你的设备降级到 Android 4.1。为此，你将需要从 Google 的开发者网站为你的 Nexus 设备下载一个旧的工厂映像[㊁]，然后，你要按照指导用那个旧的工厂映像刷新你的设备[㊂]。请千万记住，刷新设备不适于胆小的人，因为你会失去质保，如果哪里出错了，你的设备很可能变成一个昂贵的镇纸器。

B.3 需要小心的缺陷

有时候问题并不明显：glGetError() 没有返回任何不寻常的内容，跟踪也表现得很好。其他时候，应用可能由于一个不期望的位置处出现的段错误而崩溃，剥夺了我们设置断点，查出问题的机会。当这些问题发生时，我们不得不回到标准的调试技术，缩小问题查找范围，直到发现问题[㊃]。

最容易出问题的方法之一是 glVertexAttribPointer()（参见 3.4.3 节），它的使用出错经常会导致奇怪的显示问题和段错误。作为一个例子，让我们假定有一个顶点数组表示一个物体，这个物体有位置属性、颜色属性和法线属性，每个属性有三个浮点值，换句话说，总共有 9 个浮点值。为了绑定每个属性，我们加入下面的代码：

```
private static final int POSITION_COUNT = 3;
private static final int COLOR_COUNT = 3;
private static final int NORMAL_COUNT = 3;
private static final int STRIDE = POSITION_COUNT + COLOR_COUNT + NORMAL_COUNT;

«body»

floatBuffer.position(0);
glVertexAttribPointer(positionAttributeLocation, POSITION_COUNT,
        GL_FLOAT, true, STRIDE, floatBuffer);

floatBuffer.position(POSITION_COUNT);
glVertexAttribPointer(colorAttributeLocation, COLOR_COUNT,
        GL_FLOAT, false, STRIDE, floatBuffer);

floatBuffer.position(COLOR_COUNT);
glVertexAttribPointer(normalAttributeLocation, NORMAL_COUNT,
        GL_SHORT, false, STRIDE, floatBuffer);
```

㊀ https://code.google.com/p/android/issues/detail?id=52446 和 https://code.google.com/p/android/issues/detail?id=53426

㊁ https://developers.google.com/android/nexus/images

㊂ http://forums.androidcentral.com/nexus-4-rooting-roms-hacks/223923-guide-nexus-4-factory-imagerestore.html

㊃ http://blog.regehr.org/archives/849

你能找出其中的缺陷么？这是其中两个：

- 我们忘了指定以字节为单位的跨距。
- 当我们为法线设置浮点缓冲区起始位置时，错误地把它设置为 COLOR_COUNT，它是 3。我们实际上是想把它设为 POSITION_COUNT + COLOR_COUNT，因为法线是从第六个浮点值开始的。

你还看到了什么其他的问题？还有很多其他可能出错的地方；例如，我们可能调用 floatBuffer.position(COLOR_COUNT * BYTES_PER_FLOAT)，这是不对的，因为那里的参数是以浮点为单位，而不是字节。当我们实际想传递 GL_FLOAT 时，传进了 GL_SHORT 也是不对的。出错时，你要检查的第一件事就是你的 glVertexAttribPointer() 调用。

这里是一些其他建议，用来解决很多你可能遇到的更常见的缺陷：

- 检查着色器编译情况，看看着色器日志中是否有警告或错误消息。有时即使着色器编译没有问题，其所用的计算也可能是不正确的或者无用的。通过用硬编码数值替换其计算结果，然后运行程序看看它的表现，你可以缩小问题范围。

  另一个调试着色器的方法是设置检查条件，然后，如果条件是真值，我们用一种颜色绘制片段；如果条件是假值，就用另一种颜色。
- 也很容易忘记与 OpenGL 对象解除绑定或者重置某个特定状态。例如，如果通过加入调用 glBindBuffer(GL_ARRAY_BUFFER, 0) 修复了问题，那你可能忘记与顶点缓冲区对象解除绑定。然后，你可以回到那个调用的地方，看看在哪里放置解除绑定的调用。
- 如果一个物体没有显示出来，有时你可以通过禁用深度检测或剔除技术发现问题所在。如果它显示出来了，那问题就可能是它在场景中的位置，或者也可能是剔除技术配置的问题或物体的三角形卷曲顺序不对（参见 12.5.2 节）。
- 如果纹理的尺寸不是 2 的幂（参见 7.1 节），一些 GPU 就不能显示它们。如果你使用的图形格式有表示透明度的阿尔法通道，而且你的着色器代码没有考虑这一点，纹理有时可能会完全变成黑色或白色。

GPU 本身或者其驱动也很有可能存在缺陷，因此，在不同设备之间或者设备与模拟器之间进行比较有时也是有帮助的。通过不断地搜索和缩小问题的范围，总能找出问题原因，并修复缺陷。

参考文献

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 4th, 2005.

[Blo08] Joshua Bloch. *Effective Java*. Addison-Wesley, Reading, MA, 2008.

[Bur10] Ed Burnette. *Hello, Android: Introducing Google's Mobile Development Platform, Third Edition*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2010.

[Eck06] Bruce Eckel. *Thinking in Java*. Prentice Hall, Englewood Cliffs, NJ, Fourth, 2006.

[Zec12] Mario Zechner. *Beginning Android Games*. Apress, New York City, NY, 2012.

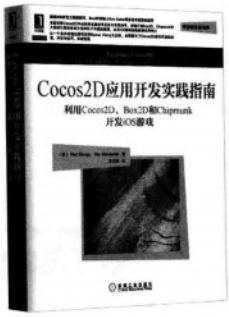
推荐阅读

HTML5游戏开发实战

作者：Makzan ISBN：978-7-111-39176-0 定价：59.00元

游戏设计师修炼之道：数据驱动的游戏设计

作者：Michael Moore ISBN：978-7-111-40087-5 定价：69.00元



Cocos2D应用开发实践指南：利用Cocos2D、Box2D和Chipmunk开发iOS游戏

作者：Rod Strougo 等 ISBN：978-7-111-42507-6 定价：99.00元

移动游戏开发精要

作者：Kimberly Unger 等 ISBN：978-7-111-43413-9 定价：59.00元

推荐阅读

Android
入门经典

Android
达人
活用宝典

Android
开发宝典

Android
嵌入式系统程序开发
(基于Cortex-A8)

Android
应用开发攻略
O'REILLY

Android游戏开发
实践指南

Android
程序设计
O'REILLY

精彩绝伦的Android UI设计

Android取证实战
调查、分析与移动安全