Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Professional WebGL Programming: Developing 3D Graphics for the Web

WebGL高级编程

——开发Web 3D图形

[美] Andreas Anyuru 著

吴文国 译

清华大学出版社

使用一流的硬件加速3D图形开发Web游戏和应用所需掌握的一切知识

希望使用最流行本地应用所具有的高品质、实时3D图形功能来开发Web游戏和应用吗？《WebGL高级编程——开发Web 3D图形》介绍了如何开发并帮助读者快速上手，书中提供了创建可以在大多数Web浏览器上运行的图形丰富的游戏和Web应用所需掌握的知识和技能。包括清晰、循序渐进的指导，专家提示和技能培养练习，详细的补充信息，实践开发实例等所有基础知识。

主要内容

- WebGL理论基础
- WebGL与其他图形技术的关系
- 必备的线性代数知识
- 故障排除和调试技术
- 使用WebGL API绘制图形
- 简化JavaScript库和3D变换
- 编写顶点着色器和片段着色器
- 纹理与光照
- 动画与用户输入
- OpenGL ES着色语言
- WebGL性能优化

从WebGL基础知识到全新地构建令人惊艳的3D图形和动画，本书为想要充分利用这种强大的Web开发技术的读者提供了完整的指导。

作者简介

Andreas Anyuru 是ST-Ericsson公司的资深技术人员，专长于Web技术，他在开发Web图形技术方面富有经验，从事实现和优化WebGL和许多其他基于Linux手机平台的Web技术。

源代码下载及技术支持

http://www.wrox.com

http://www.tupwk.com.cn/downpage

wrox.com

Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

Read More

Find articles, ebooks, sample chapters and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

清华大学出版社数字出版网站

WQBook

www.wqbook.com

Wrox™
An Imprint of
WILEY

ISBN 978-7-302-32183-5

定价：49.00元

WebGL 高级编程

——开发 Web 3D 图形

[美] Andreas Anyuru 著
吴文国 译

清华大学出版社
北 京

Andreas Anyuru
Professional WebGL Programming: Developing 3D Graphics for the Web
EISBN：978-1-119-96886-3
Copyright © 2012 by John Wiley & Sons, Inc.
All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2012-8055

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

WebGL 高级编程——开发 Web 3D 图形/(美)阿尤鲁(Anyuru, A.) 著；吴文国 译. —北京：清华大学出版社，2013.6

书名原文：Professional WebGL Programming: Developing 3D Graphics for the Web
ISBN 978-7-302-32183-5

Ⅰ. ①W… Ⅱ. ①阿… ②吴… Ⅲ. ①网页制作工具—程序设计 Ⅳ. ①TP393.092

中国版本图书馆 CIP 数据核字(2013)第 083362 号

责任编辑：王　军　于　平
装帧设计：牛静敏
责任校对：邱晓玉
责任印制：刘海龙

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：185mm×260mm　　印　　张：19.5　　字　　数：475 千字
版　　次：2013 年 6 月第 1 版　　印　　次：2013 年 6 月第1 次印刷
印　　数：1～3000
定　　价：49.00 元

产品编号：050190-01

作 者 简 介

Andreas Anyuru 精通包括 WebGL 在内的许多 Web 技术，他在移动设备 Web 浏览器集成和优化方面曾有多年的工作经历。Andreas 是 V8 JavaScript 引擎的贡献者之一，后者用在 Google 台式浏览器 Chrome 和 Android 手机中。他设计了几个新教程，并在 Lund 大学的信息和通信工程学院任教。

Andreas 是 ST-爱立信公司的高级技术人员，主要研究领域是 Web 技术。主要工作职责是确保现有的和即将推出的 Web 技术以最优的方式应用于 ST-爱立信基于 Linux 的移动平台上。

技术编辑简介

Paul Brunt 在 HTML5 刚推向市场时就用 JavaScript 开发了几个游戏和应用程序，这些游戏和应用程序广泛应用了 SVG、画布和新一代 JavaScript 引擎等技术。他的工作还包括一个名为“Berts Breakdown”的概念验证平台的游戏演示程序。他对计算机艺术怀有浓厚的兴趣，在 Blender 和实时图形方面具有渊博的知识，WebGL 的发布激发了他开发 GLGE 的欲望。2009 年，当时 WebGL 还处于婴儿期，他就开始开发 GLGE，然后以锐不可挡之势转向在线游戏开发。他对其他 WebGL 架构和项目也有贡献；另外他于 2011 年将 jiglib 物理库移植到 JavaScript 中，并首次在浏览器里演示 3D 物理。

致　　谢

首先要感谢提出 WebGL 规范的 Khronos WebGL 工作小组、浏览器供应商以及实现了 WebGL 规范的开源项目组。没有你们的工作，就没有本书的问世。此外，感谢 WebGL 社区的所有开发人员，你们的讨论、经历和示例使 WebGL 成为一项技术。

感谢 Wiley 出版社的员工，特别是 Jeff Riley、Sara Shlaer、Ellie Scot 和 Chris Webb，感谢你们给我工作上的支持!

感谢 ST-爱立信公司的所有同仁，有了你们，ST-爱立信才成为一个工作愉快的地方；特别感谢 Kent Olsson、Christian Bejram、Marco Cornero、Göran Roth 和 Patrik Aberg，感谢你们给我的信任、支持和指导。

感谢我的家人对我工作的支持和理解，最重要的是，我要感谢我的太太 Jessica。Jessica，你是我的最爱，你是我的一切!

前　言

现在用户将很大一部分时间花在自己的计算机、平板电脑和智能手机上冲浪。用户整天忙于编写文档、发送和接收电子邮件、聊天、观看视频、欣赏音乐、玩游戏或购物。Web浏览器已成为大多数设备上最重要的应用程序。结果是，Web 新技术的发展日新月异。现在 Web 浏览器变得越来越快、越来越稳定、越来越安全，它能够处理每天出现的新技术。

这是一个令 Web 开发人员兴奋的时代。如果想开发一个应用程序，并且希望将它发布给众多用户使用，则 Web 无疑是首选的平台。尽管 Web 技术在最近几年里已快速发展，但是与其他原生应用程序(Native Application)相比，它有一个缺点，即 Web 技术无法创建游戏和其他应用程序所需要的实时 3D 图形。WebGL 的出现改变了这一切。

WebGL 以前所未有的方式允许用户硬件加速 2D 和 3D 图形。用户既可以应用 HTML、CSS 和 JavaScript 提供的全部优点，同时又可以应用功能强大的图形处理单元(Graphics Processor Unit，GPU)提供的好处。本书先向用户介绍开始开发基于 WebGL 的 2D 或 3D 图形应用程序所需的全部基础知识和准备工作。

本书读者对象

本书的主要读者是已对 HTML、CSS 和 JavaScript 有一个基本了解，但是想深入学习如何用 WebGL 为自己的 Web 应用程序或网页创建硬件加速的 2D 或 3D 图形程序的用户。

能够从本书获益的第二类读者包括那些曾经开发过台式 3D API(如 OpenGL 或 Direct3D 等)，但是想在 Web 上把这些知识应用于 WebGL 程序开发的开发人员。

对于学习 3D 图形课程以及想把 WebGL 作为创建原型和测试程序的一个简单工具的学生来说，本书也是十分有用的。因为你们只需要一个支持 WebGL 的 Web 浏览器和一个可以用来编写应用程序的文本编辑器，编写 WebGL 程序的门槛远比其他 3D 图形 API 低(如台式 OpenGL 或 Direct3D)，后者需要一个完整的工具链。

本书主要内容

本书向读者介绍如何开发基于 WebGL 的 Web 应用程序。虽然 WebGL API 可用来硬件加速 2D 图形和 3D 图形，但是它的主要作用是用来创建 3D 图形。3D 图形 API 的一些图书只介绍 API 本身，并没有对 3D 图形或如何使用 API 进行较多的介绍。本书不要求读者

具备任何 3D 图形的理论基础。希望读者通过本书的学习能够掌握 3D 图形基础知识，以及学会用 WebGL API 开发 Web 应用程序。

此外，本书还介绍线性代数的部分基础知识，这有助于读者深入理解 3D 图形和 WebGL 底层的运行机制。掌握了线性代数的基本知识，读者就可以把重点放在线性代数中 3D 图形重要的部分。读者不需要去阅读厚达几百页的通用线性代数教材，这些图书通常以通用和抽象的方式介绍每个专题。如果读者属于只想很快开始编写代码的一类用户，不需要阅读线性代码一节的全部内容(主要是第 1 章的部分内容)。读者可以跳过这部分内容。若后来发现某些问题与线性代数有关，则可以回过头来再仔细阅读相关内容。

必须向读者指出，本书并没有介绍 WebGL 的全部方法。要想得到 WebGL API 的完整参考手册，则至少要拥有一本 Khronos 发布在 www.khronos.org/registry/webgl/specs/latest/ 上的有关最新 WebGL 规范的图书。

本书的组织结构

本书采用一种有利于大多数读者阅读的逻辑顺序。然而，由于读者有不同的背景，因此他们可以根据自己的需要选择最适合自己的顺序阅读本书。下面的概述可能会有助于读者安排好读书计划。

第 1 章介绍理论知识。向读者介绍 WebGL 的历史背景，并把它与其他几个图形技术进行比较。这一章还向读者介绍 WebGL 图形流水线结构，同时也对图形硬件做概括性介绍。此外，本章还介绍线性代数的部分基础知识，这些内容有助于读者更好地理解 3D 图形和 WebGL 的底层工作机制。如果读者认为自己对本章介绍的线性代数有足够的知识，则可以跳过这一部分内容，直接阅读第 2 章。

第 2 章向读者介绍很多实用知识。介绍如何建立第一个完整的 WebGL 应用程序和如何编写一个非常简单的顶点着色器和片段着色器。为了尽可能方便读者后面的学习，本章还介绍几个有用的开发和调试工具。此外，还会介绍如何排除 WebGL 应用程序中的错误。

第 3 章向读者详细介绍 WebGL 绘制的各个不同选项。读者要学习在绘图中可以使用的各个 WebGL 方法和图元。

第 4 章介绍 3 个小型的 JavaScript 图形库，它们常用来执行 WebGL 应用程序中的矢量和矩阵运算。此外，读者还要学习如何用变换运算确定对象在 3D 空间里的位置和朝向。这一章非常重要，如果读者以前没有 3D 图形方面的经历，则要在这一章多花一些时间。

第 5 章介绍 WebGL 的纹理贴图。纹理贴图是使 3D 对象具有真实感的一个重要步骤。此外，本章还介绍如何通过 WebGL 中丢失上下文的处理使自己的程序更加健壮。在开始设计真实应用程序之前一定要仔细阅读这部分的内容，因为它们对健壮性有很高的要求。

第 6 章介绍如何使用动画技术创建 WebGL 场景的动画效果。此外在这一章读者还要学习 JavaScript 的事件处理模式，以及如何利用键盘事件和鼠标事件控制 WebGL 场景。

第 7 章介绍 WebGL 的光照处理，这是一个令人兴奋的主题。读者学习各种光照模型和如何用这些光照模型编写顶点着色器和片段着色器。光照是使 WebGL 场景具有比较真实效果的一项重要技术。

第 8 章包含一些指南、提示和使用技巧，读者利用这些技术尽可能提高自己的 WebGL 应用程序的性能。此外，这一章还分析 WebGL 应用程序的性能问题，如何找到瓶颈位置，如何消除这些瓶颈。该章也会介绍 WebGL 的底层工作机制。这一章还向读者介绍关键软件成分和关键硬件成分。最后本章将介绍移动行业里的一个示例。

阅读本书之前的准备工作

为了运行本书的例子，读者需要一个支持 WebGL 的浏览器。如果读者的系统还没有这样一个浏览器，可以免费下载一个。想知道当前支持 WebGL 的 Web 浏览器列表，请访问 www.khronos.org/webgl/wiki/。在编写本书的时候，以下浏览器支持 WebGL：

- Apple Safari
- Google Chrome
- Mozilla Firefox
- Opera

为了编写自己的 WebGL 应用程序，要使用自己喜欢的文本编辑器。本书还介绍程序调试和程序故障排除的几个十分有用的工具，它们是 Chrome 开发人员工具(Chrome Developer Tools)、Firebug 和 WebGL Inspector。它们都属于开源代码工具，可以免费下载。

源代码

在读者学习本书中的示例时，可以手动输入所有代码，也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从本书合作站点 http://www.wrox.com/或 http://www.tupwk.com.cn/downpage 上下载。登录到站点 http://www.wrox.com/，使用 Search 工具或使用书名列表就可以找到本书。接着单击 Download Code 链接，就可以获得所有的源代码。既可以选择下载一个大的包含本书所有代码的 ZIP 文件，也可以只下载某个章节中的代码。

由于许多图书的标题都很类似，因此按 ISBN 搜索是最简单的，本书英文版的 ISBN 是 978-1-119-96886-3。

在下载代码后，只需用解压缩软件对它进行解压缩即可。另外，也可以进入 http://www.wrox.com/dynamic/books/download.aspx 上的 Wrox 代码下载主页，查看本书和其他 Wrox 图书的所有代码。记住，可以使用书中列出的程序清单的编号容易地找到所要寻找的代码，

如“程序清单 0-1”。

当为大多数可下载的源代码文件命名时，我们会使用这些清单中的数值。对于那些很少的没有用它自己的清单数值命名的程序清单，它们都与文件名匹配，所以很容易就可以在下载的源代码文件中找到它们。

勘误表

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

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

如果您发现的错误在我们的勘误表里还没有出现的话，请登录 www.wrox.com/contact/techsupport.shtml 并完成那里的表格，把您发现的错误发送给我们。我们会检查您的反馈信息，如果正确，我们将在本书的勘误表页面张贴该错误消息，并在本书的后续版本加以修订。

p2p. wrox.com

要与作者和同行讨论，请加入 p2p.wrox.com 上的 P2P 论坛。这个论坛是一个基于 Web 的系统，便于您张贴与 Wrox 图书相关的消息和相关技术，与其他读者和技术用户交流心得。该论坛提供了订阅功能，当论坛上有新的消息时，它可以给您传送感兴趣的论题。Wrox 作者、编辑和其他业界专家和读者都会到这个论坛上来探讨问题。

在 http://p2p.wrox.com 上，有许多不同的论坛，它们不仅有助于阅读本书，还有助于开发自己的应用程序。要加入论坛，可以遵循下面的步骤：

(1) 进入 p2p.wrox.com，单击 Register 链接。

(2) 阅读使用协议，并单击 Agree 按钮。

(3) 填写加入该论坛所需要的信息和自己希望提供的其他可选信息，单击 Submit 按钮。您会收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

不加入 P2P 也可以阅读论坛上的消息，但要张贴自己的消息，就必须先加入该论坛。

加入论坛后，就可以张贴新消息，响应其他用户张贴的消息。可以随时在 Web 上阅读消息。如果要让该网站给自己发送特定论坛中的消息，可以单击论坛列表中该论坛名旁边的 Subscribe to this Forum 图标。

要想了解更多的有关论坛软件的工作情况，以及 P2P 和 Wrox 图书的许多常见问题的解答，就一定要阅读 FAQ，只需在任意 P2P 页面上单击 FAQ 链接即可。

目 录

第 1 章 WebGL 简介 ······1
1.1 WebGL 基础 ······1
1.2 浏览器 3D 图形吸引人的原因 ······2
1.3 设计一个图形 API ······3
1.3.1 即时模式 API ······3
1.3.2 保留模式 API ······3
1.4 图形硬件简介 ······4
1.4.1 GPU ······4
1.4.2 帧缓存 ······5
1.4.3 纹理存储器 ······6
1.4.4 视频控制器 ······6
1.5 WebGL 图形流水线 ······6
1.5.1 顶点着色器 ······8
1.5.2 图元装配 ······11
1.5.3 光栅化 ······12
1.5.4 片段着色器 ······12
1.5.5 逐片段操作 ······15
1.6 WebGL 与其他图形技术的比较 ······16
1.6.1 OpenGL ······16
1.6.2 OpenGL ES 2.0 ······18
1.6.3 Direct3D ······19
1.6.4 HTML5 画布 ······21
1.6.5 可缩放矢量图形 ······25
1.6.6 VRML 与 X3D ······26
1.7 线性代数简介 ······27
1.7.1 坐标系 ······27
1.7.2 点与顶点 ······27
1.7.3 矢量 ······28
1.7.4 矢量的点积或标积 ······29
1.7.5 叉积 ······30
1.7.6 齐次坐标 ······31
1.7.7 矩阵 ······31
1.7.8 仿射变换 ······34
1.8 小结 ······39
第 2 章 创建基本的 WebGL 示例 ······41
2.1 绘制三角形 ······41
2.1.1 创建 WebGL 上下文 ······45
2.1.2 创建顶点着色器和片段着色器 ······47
2.1.3 编译着色器 ······48
2.1.4 创建程序对象和链接着色器 ······48
2.1.5 建立缓冲 ······50
2.1.6 绘制场景 ······51
2.2 了解 WebGL 编码风格 ······52
2.3 调试 WebGL 应用程序 ······53
2.3.1 使用 Chrome 开发人员工具 ······53
2.3.2 Firebug 的使用 ······59
2.3.3 WebGL 的错误处理与错误代码 ······61
2.3.4 WebGL Inspector ······64
2.3.5 WebGL 的故障排除 ······70
2.4 用 DOM API 载入着色器 ······71
2.5 更高级的综合示例 ······73
2.6 小结 ······77
第 3 章 绘制 ······79
3.1 使用 WebGL 绘制图元和绘图方法 ······79

3.1.1 图元 ······ 80
3.1.2 顶点组绕顺序的重要性 ······ 84
3.1.3 WebGL 的绘图方法 ······ 85
3.2 类型化数组 ······ 91
3.2.1 缓冲与视图 ······ 91
3.2.2 WebGL 支持的视图类型 ······ 92
3.3 探讨不同的绘图方法 ······ 93
3.3.1 gl.drawArrays()和 gl.TRIANGLES ······ 94
3.3.2 gl.drawArrays()方法和 gl.TRIANGLE_STRIP 图元 ······ 96
3.3.3 gl.drawElements()方法和 gl.TRIANGLES 图元 ······ 98
3.3.4 gl.drawElements()方法和 gl.TRIANLE_STRIP 图元 ··· 100
3.3.5 总结比较 ······ 102
3.3.6 前期变换顶点缓存和后期变换顶点缓存 ······ 102
3.4 为提高性能交叉存放顶点数据 ······ 104
3.5 使用顶点数组或常量顶点数据 ······ 113
3.6 总结本章的最后一个示例 ······ 114
3.7 小结 ······ 124

第 4 章 小型 JavaScript 库与变换 ······ 127
4.1 JavaScript 中矩阵和向量的操作 ······ 127
4.1.1 Sylvester 库 ······ 128
4.1.2 WebGL-mjs 库 ······ 132
4.1.3 glMatrix 库 ······ 135
4.2 变换运算 ······ 139
4.3 理解完整的变换流水线 ······ 145
4.4 变换的实践 ······ 146
4.4.1 为对象坐标设置缓冲 ······ 147
4.4.2 用 JavaScript 创建变换矩阵并上传给着色器 ······ 148
4.4.3 将变换矩阵上传给 GPU 中的顶点着色器 ······ 148
4.4.4 调用绘图方法 ······ 149
4.5 理解变换顺序的重要性 ······ 150
4.5.1 使用一个固定的全局的坐标系 ······ 150
4.5.2 使用移动的局部的坐标系 ······ 153
4.5.3 变换矩阵的入栈和出栈操作 ······ 154
4.6 一个完整的示例：绘制几个变换后的对象 ······ 157
4.6.1 使用 WebGL 代码创建立方体 ······ 159
4.6.2 视图变换和模型变换的组织 ······ 161
4.7 小结 ······ 162

第 5 章 纹理贴图 ······ 163
5.1 理解丢失上下文 ······ 164
5.1.1 理解解决丢失上下文问题所需要的设置 ······ 164
5.1.2 处理丢失上下文问题时需要考虑的几个因素 ······ 166
5.2 2D 纹理与立方映射纹理 ······ 169
5.3 载入纹理 ······ 170
5.3.1 创建 WebGLTexture 对象 ···· 170
5.3.2 绑定纹理 ······ 171
5.3.3 载入图像数据 ······ 171
5.3.4 将纹理上传到 GPU ······ 173
5.3.5 定义纹理参数 ······ 174
5.3.6 理解载入纹理的完整过程 ······ 175
5.3.7 创建一个纹理对象并载入纹理数据 ······ 177
5.4 定义纹理坐标 ······ 178
5.5 着色器中的纹理处理 ······ 180

5.6 处理纹理过滤……183
5.6.1 纹理伸展……184
5.6.2 纹理收缩……185
5.6.3 Mip 映射纹理……186
5.7 理解纹理坐标包装……188
5.7.1 应用 gl.REPEAT 包装模式……188
5.7.2 应用 gl.MIRRORED_REPEAT 包装模式……190
5.7.3 应用 gl.CLAMP_TO_EDGE 包装模式……191
5.8 一个完整的应用纹理示例……191
5.9 获得用作纹理的图像……194
5.9.1 下载免费纹理……194
5.9.2 用自己拍摄的照片生成纹理……194
5.9.3 绘制图像……195
5.9.4 购买纹理……195
5.10 同域策略与跨域资源共享……195
5.10.1 同域策略应用于一般的图像……196
5.10.2 同域策略应用于纹理……197
5.10.3 跨域资源共享……199
5.11 小结……200

第 6 章 动画与用户输入……203

6.1 创建动画场景……203
6.1.1 setInterval()和 setTimeout()的使用……205
6.1.2 使用 requestAnimationFrame()函数……206
6.1.3 帧频不同引起的运动补偿……209
6.1.4 创建 FPS 计数器测量动画的平稳性……210
6.1.5 用 FPS 作为测量值的缺点……212
6.2 用户交互事件的处理……213
6.2.1 DOM Level 0 基本事件处理……214
6.2.2 DOM Level 2——高级事件处理方法……215
6.2.3 键盘输入……217
6.2.4 鼠标输入……221
6.3 综合应用新知识……223
6.4 小结……229

第 7 章 光照……231

7.1 光源……231
7.2 局部光照模型的工作原理……232
7.3 Phong 反射模型……232
7.3.1 环境反射……233
7.3.2 漫反射……234
7.3.3 镜面反射……236
7.3.4 Phong 反射模型的完整公式和着色器……239
7.3.5 光照效果与纹理相结合……243
7.4 WebGL 光照中需要的 JavaScript 代码……246
7.4.1 为顶点法线设置缓存……247
7.4.2 计算法线矩阵并上传给着色器……249
7.4.3 将光照信息上传给着色器……250
7.5 将不同的插值方法用于着色……250
7.5.1 平面着色……251
7.5.2 Gouraud 着色……252
7.5.3 Phong 着色……253
7.6 矢量必须归一化……256
7.6.1 顶点着色器中的矢量归一化……257
7.6.2 片段着色器的矢量归一化……257

7.7 应用不同类型的光源……258
7.7.1 平行光……258
7.7.2 点光源……259
7.7.3 聚光源……259
7.8 光强衰减……262
7.9 光照映射……265
7.10 小结……267

第 8 章 WebGL 性能优化……269
8.1 WebGL 底层工作机制……269
8.1.1 支持 WebGL 的硬件……270
8.1.2 关键的软件组成……271
8.2 WebGL 性能优化……274
8.2.1 避免初学者的典型错误……274
8.2.2 确定瓶颈位置……275
8.2.3 有关性能的一般性建议……279
8.2.4 改善 CPU 受限的 WebGL 应用程序性能的建议……282
8.2.5 改善顶点受限的 WebGL 应用程序性能的建议……283
8.2.6 改善像素受限的 WebGL 应用程序性能的建议……285
8.3 深入分析融合……286
8.3.1 融合简介……286
8.3.2 设置融合函数……287
8.3.3 绘制顺序与深度缓冲区……290
8.3.4 绘制包含不透明对象和半透明对象的场景……290
8.3.5 修改融合公式中的默认运算符……291
8.3.6 使用预乘 alpha 值……292
8.4 深入讨论 WebGL……292
8.4.1 使用 WebGL 框架……293
8.4.2 发布到 Google Chrome Web Store……293
8.4.3 使用额外资源……293
8.5 小结……294

第 1 章

WebGL 简介

本章主要内容：

- WebGL 基础
- 浏览器 3D 图形吸引人的原因
- 如何使用即时模式 API
- 图形硬件初步
- WebGL 图形流水线
- WebGL 与其他图形技术的比较
- 学习 WebGL 必备的线性代数知识

本章介绍 WebGL、它的一些重要理论以及 3D 图形的一般性知识。通过本章的学习，你可以掌握什么是 WebGL，并对 WebGL 使用的图形流水线有一个基本的理解。

此外，为了使你对 WebGL 有一个更全面的了解，本章还介绍了其他几个相关的图形标准，并且把 WebGL 与这些图形技术进行比较。

最后，本章复习了线性代数的一些基本理论，如果你确实想更深层掌握 WebGL 技术，这些知识是必不可少的。

1.1 WebGL 基础

WebGL 是一个用来在 Web 上生成三维图形效果的应用编程接口。它是以 OpenGL ES 2.0 为基础的。与 OpenGL 一样，它也提供绘制(Rendering(API))功能，但是它应用在 HTML 和 JavaScript 上下文中。WebGL 绘制的曲面本质上是 HTML5 的画布，后者最早是由 Apple 公司引入到 WebKit 开源浏览器引擎中的。引入 HTML5 画布的理由是为了能够在 Dashboard 小部件(Widget)等应用程序中和 Apple Mac OS X 操作系统的 Safari 浏览器中绘制二维图形。

以此画布原理为基础，Mozilla 公司的 Vladimir Vukicevic 开始试验画布的三维图形效果。他称最初的模型为 3D 画布(Canvas 3D)。2009 年，Khronos Group 开始组建新的 WebGL 工作组。现在这个工作组由 Apple、Google、Mozilla 和 Opera 等几个主要浏览器的供应商组成。Khronos Group 是一个非营利的行业同盟，制订了许多开放标准和无版权费的 API。它成立于 2000 年 1 月。除此之外，该小组还开发了许多其他 API 并提出了其他图形技术，如用于嵌入式设备的 OpenGL ES，用于并行程序设计的 OpenCL，用于矢量图形低层加速的 OpenVG，用于加速多媒体部件的 OpenMAX。2006 年，Khronos Group 开始接管并升级了 OpenGL。后者是台式机的 3D 图形 API。

WebGL 1.0 规范最终于 2011 年 3 月确定下来，并且在 Google Chrome、Mozilla Firefox 等浏览器中以及在 Safari 和 Opera 等的开发过程中(在本书编写时)实现对它的支持。

有关不同浏览器对 WebGL 支持的最新消息，请访问 www.khronos.org/webgl/。

1.2 浏览器 3D 图形吸引人的原因

在万维网的早期，网页的内容只包含静态的文本文档，它们由 Web 浏览器读取和显示。近几年中，Web 技术已得到巨大的发展，现在许多网站实际上是全功能的应用程序。它们支持服务器与客户端之间的双向通信，允许用户注册和登录，而且 Web 应用程序提供了丰富的用户界面，如图形、声音和视频。

Web 应用程序的快速发展促使它们成为原生应用程序的一个强有力的竞争者。Web 应用程序包括以下几个优点：

- 可以廉价迅速地传播给大量的用户。用户只需要一个兼容的 Web 浏览器。
- 容易维护。当我们发现应用程序有一个 bug，或者当我们想给应用程序添加有利于用户的新功能时，只需要升级 Web 服务器上的应用程序，用户就可以立刻使用升级后的应用程序。
- 由于应用程序是在 Web 浏览器中执行的，因此相对比较容易得到跨平台的支持(如得到 Windows、Mac OS、Linux 等操作系统的支持)。至少在理论上是如此。

然而，与原生应用程序相比，Web 应用程序也存在有一些局限性。其中之一是 Web 应用程序的用户界面无法做到原生应用程序那样丰富。自从引入了 HTML5 画布标记以来，这种情形已发生很大的变化。有了它，我们可以为 Web 应用程序创建真正高级的 2D 图形。但是早期的 HTML5 画布标记只允许定义 2D 上下文，并不支持 3D 图形。

但是，自从有了 WebGL，开发人员就可以在浏览器内部实现 3D 图形的硬件加速，就可以创建 3D 游戏或者其他高级的 3D 图形应用程序。另一方面，它具有 Web 应用程序的全部优点。WebGL 具有以下吸引人的特性：

- WebGL 是一个开放的标准，任何人都可以使用，不需要支付任何版权费。

- WebGL 利用图形硬件加速图形绘制，这意味着它的速度确实很快。
- WebGL 可以在支持它的本地浏览器上运行，不需要任何插件。
- 由于 WebGL 是以 OpenGL ES 2.0 为基础的，因此对于具有 OpenGL ES 2.0 编程经验的开发人员而言，甚至对于熟悉台式机 OpenGL 开发的人们而言，它是很容易学习的。

WebGL 标准也为学生和其他人员提供了一个快速学习和测试 3D 图形的方法。不像大多数其他 3D API 那样，需要下载和安装一个工具链。用户只需要一个编写代码的文本编辑器，为了查看生成结果，只需要把这个文件载入到支持 WebGL 的浏览器中。

1.3　设计一个图形 API

设计一个图形 API，基本上采用以下两种不同的模式：

- 即时模式 API
- 保留模式 API

WebGL 属于即时模式 API。

1.3.1　即时模式 API

对于即时模式 API(immediate-mode API)，每一帧的场景，不管是否已发生变化，都需要重新绘制。提供 API 的图形库并没有保存需要绘制场景的内部模型，但是应用程序需要在内存中用自己的方法表示场景。这种设计模式大大提高了应用程序的灵活性和控制能力。但是，它需要应用程序执行更多的操作，如跟踪场景的模型、执行初始化和清除操作。图 1-1 用简单的结构说明了即时模式 API 的绘制过程。

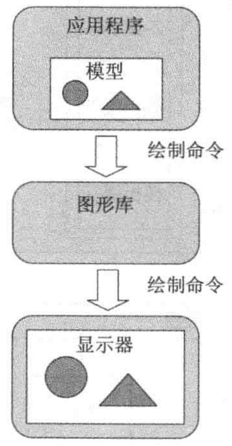


图 1-1　即时模式 API 的工作流程

1.3.2　保留模式 API

提供保留模式 API(retained-mode API)的图形库包含一个内部模型或场景图形。它包含

了所有需要绘制的对象。当应用程序调用保留模式 API 时，需要更新这个内部模型，图形库决定什么时候需要在屏幕执行实际的绘制。这意味着，不需要为绘制每个帧的场景执行绘制命令。在某些方面，保留模式 API 更容易使用，因为图形库可以替用户执行许多操作，因此用户不需要在应用程序中执行这些操作。图 1-2 是保留模式 API 的工作流程。保留模式 API 的一个示例是可缩放矢量图形(Scalable Vector Graphics，SVG)。本章后面将要介绍 SVG 内容。

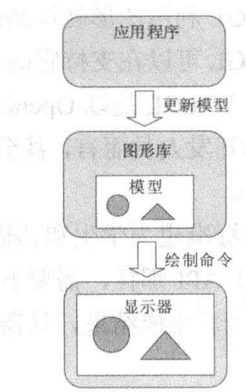


图 1-2　保留模式 API 的工作原理

1.4　图形硬件简介

WebGL 是一个低级 API，而且是建立在 OpenGL ES 2.0 之上的，因此它与实际的图形硬件有密切的关系。为了帮助你更好地理解本书的其余内容，有必要对图形硬件和它的工作原理有一个基本的了解。也许你已经对这方面所有了解，但是为了确保你具有这些必要的知识，本节简单介绍图形硬件的基础知识。

图 1-3 是计算机系统的一个简单结构。应用程序(不管它是运行在 Web 浏览器中的 WebGL 应用程序还是其他应用程序)运行在 CPU 上，并且使用主存(通常简称为 RAM)。为了显示 3D 图形，应用程序需要不断调用底层的驱动程序，后者通过总线把图形数据发送到图形处理单元(Graphics Processing Unit，GPU)。

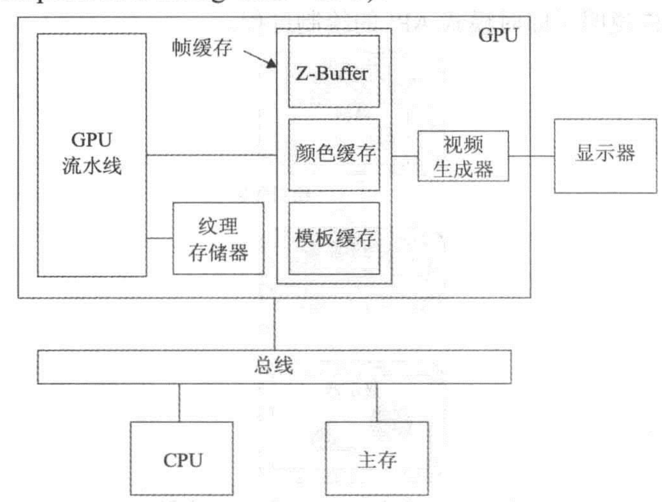


图 1-3　图形硬件以及它与其他硬件关系的简图

1.4.1　GPU

GPU 是专门设计的图形绘制设备，专门用来绘制输出在屏幕上的图形。GPU 通常是

高度并行化的，而且可以快速处理图形数据。GPU 这个术语最早是由 NVIDIA 公司于 1999 年提出的，并推广到市场。当时该公司推出世界上的第一个 GPU，即 GeForce 256。

通常 GPU 采用流水线结构。根据这种结构，数据从流水线的一个阶段(stage)传送到下一个阶段。在本章后面，还要介绍 WebGL 图形流水线的处理步骤。从理论上讲，WebGL 图形流水线也是由多个阶段组成的，它们与 GPU 物理流水线的各个阶段相对应。

1.4.2 帧缓存

图形数据通过整个 GPU 流水线传送后，最后写入到帧缓存中(framebuffer)。帧缓存是一个存储器，它保存了最终显示在屏幕上图像的全部信息。用作帧缓存的物理内存有几种不同的位置。对于简单的图形系统，帧缓存实际上通常就是主存的一部分。但是，最新的图形系统，帧缓存属于 GPU 中专用的快速图形存储器，或者是靠近 GPU 一侧的单独存储芯片。

帧缓存通常至少由以下三个不同的子缓存组成：

- 颜色缓存(color buffer)
- Z-缓存(Z-buffer)
- 模板缓存(stencil buffer)

1. 颜色缓存

颜色缓存是一个矩形阵列的存储器，它以 RGB 或 RGBA 格式保存屏幕上每个像素的颜色。颜色缓存为 RGB 的每个颜色分量分配一定的位数。此外，可能还有一个 alpha 通道，此通道用一定的位数表示帧缓存中每个像素的透明程度(或不透明程度)。一个像素可以使用的全部位数称为帧缓存的颜色深度。例如，颜色深度有：

- 每像素 16 位
- 每像素 24 位
- 每像素 32 位

如果一个帧缓存的颜色深度只有16 位，则它通常只用在小型设备上，如比较简单的移动设备。如果每个像素使用 16 位颜色，常用的分配方案是红颜色 5 位，绿颜色 6 位，蓝颜色 5 位，没有 alpha 通道。这个格式通常称为 RGB565 模式。之所以给绿色多分配一位，是因为，人类眼睛对绿颜色最敏感。16 位颜色系统最多可以显示 2^{16}= 65 536 种不同的颜色。

同样道理，颜色深度为 24 位的帧缓存的常见分配方案是：红色 8 位，绿色 8 位，蓝色 8 位。它可以提供 1600 万种颜色，这样的帧缓存也没有 alpha 通道。

每个像素 32 位的帧缓存的通常分配方案是，红色 8 位，绿色 8 位，蓝色 8 位(分配方案与 24 位颜色深度相同)，另外 8 位用作 alpha 通道。

这里，你可能已经注意到，帧缓存中的 alpha 通道不常用。通常把帧缓存中的 alpha 通道称为目标 alpha 通道，不同于源 alpha 通道。后者代表传入像素的透明度。例如，有一个称为 alpha 融合的运算，它可以生成对象的透明效果。这个运算需要源 alpha 通道，但是不

需要帧缓存中的目标 alpha 通道。

第 8 章将进一步讨论 alpha 融合。

2. Z-缓存

颜色缓存通常保存了在某个时刻 3D 场景中对观察者可见对象的颜色。但是在 3D 场景中，一个对象可能被其他对象遮挡，当整个场景绘制完成时，颜色缓存中不会有属于被隐藏对象的像素信息。

这通常是由图形硬件的 Z-缓存来实现的。Z-缓存通常也称为深度缓存。Z-缓存的单元数量与颜色缓存中的像素数量相同。Z-缓存的每个单元，存储了离观察者最近图元(primitive)的距离。

在本章后面关于深度缓存测试的部分中，将介绍如何利用 Z-缓存处理场景中的深度问题。

3. 模板缓存

除了颜色缓存和 Z-缓存这两个最常用的缓存外，最新的图形硬件还包含模板缓存。模板缓存可以用来控制在颜色缓存的某个位置写入操作。一个实际应用的示例是用它来处理阴影。

1.4.3 纹理存储器

在 3D 图形中，一个重要的操作是把纹理应用于对象的表面。可以把这个过程看成是把图像“粘贴到”几何对象的表面。这些图像就是纹理，需要用缓存保存它们，这样 GPU 可以快速有效地访问它们。通常 GPU 有一个物理的纹理存储器用来存储纹理图像。

第 5 章将进一步讨论纹理处理过程。

1.4.4 视频控制器

视频控制器(也称为视频生成器)以一定的频率逐行扫描颜色缓存，并更新屏幕上的显示内容。对于 LCD 显示器，整个屏幕通常每秒更新 60 次。这就是说，刷新频率为 60Hz。

1.5 WebGL 图形流水线

一个使用 WebGL 技术的 Web 应用程序通常由 HTML、CSS 和 JavaScript 等文件组成，

它们都在 Web 浏览器中运行。此外，除了这些经典的 Web 应用程序内容外，一个 WebGL 应用程序还包含着色器(Shader)的代码和表示 3D(或 2D)对象的数据。

如果浏览器已内置了对 WebGL 的支持，则不需要插件就可以执行 WebGL 程序。JavaScript 代码调用 WebGL API，并把有关 3D 模型的绘制信息传送给 WebGL 流水线。这些信息不仅包含了 WebGL 流水线的两个可编程着色器(即顶点着色器和片段着色器)的源代码，而且还包含了有关 3D 模型如何绘制的信息。

当图形数据通过整个 WebGL 流水线后，GPU 就把结果写入到 WebGL 称之为绘制缓存(drawing buffer)的内存中。我们不妨把绘制缓存看成 WebGL 的帧缓存。它与帧缓存一样，也有一个颜色缓存、一个 Z-缓存和一个模板缓存。但是绘制缓存中的内容在传送到物理帧缓存之前，需要与 HTML 页面中的其他内容进行组合。物理帧缓存中的实际结果直接显示在屏幕上。

在下面几小节中我们介绍 WebGL 流水线的各个阶段，如图 1-4 所示。正如图中所表示的那样，WebGL 流水线有若干个阶段组成。对于 WebGL 程序员来说，最重要的是顶点着色器和片段着色器。

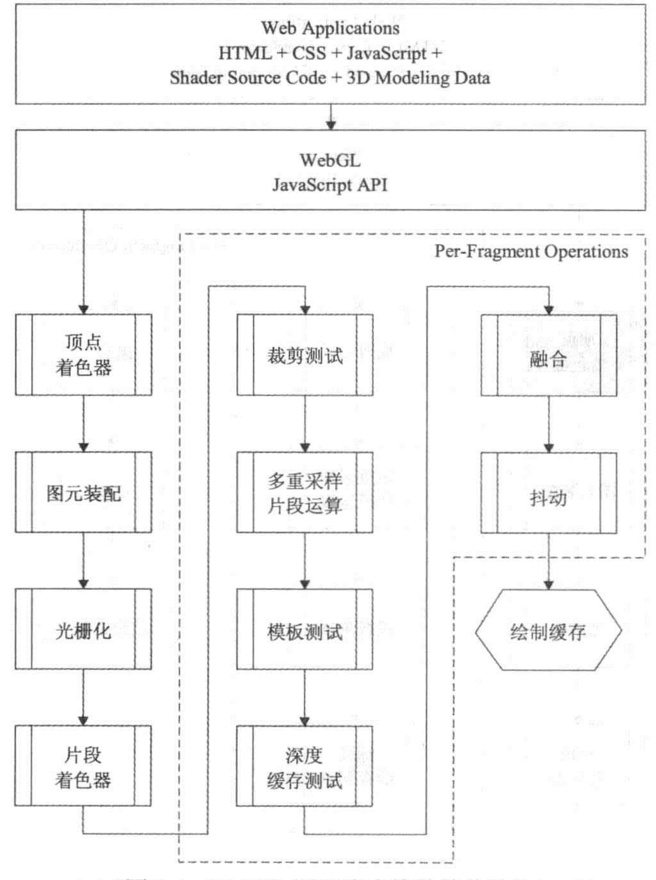


图 1-4 WebGL 图形流水线的总体结构

下面几小节需要引入几个新术语。其中部分术语将在本章论述，其余将在本书后面在几章中介绍。下面这一节概括地介绍 WebGL 图形流水线的结构，这意味着你不需要理解

这一节中的全部细节内容。

1.5.1 顶点着色器

为了得到一个真实感的 3D 场景，仅仅绘制某些位置的对象是不够的，还需要考虑到灯光照射到这些对象时的效果。我们用一个通用的术语表示确定灯光对不同材质效果的整个过程，即着色(shading)。

在 WebGL 中，着色过程分为以下两个阶段：

- 顶点着色器
- 片段着色器

第一阶段是顶点着色器(片段着色器是流水线的后阶段，将在本章的后面介绍)。顶点着色器这个名称来源是这样的事实：我们总是用顶点(vertex)表示几何对象的角点或交点。顶点着色器是流水线中对顶点进行着色的阶段。图 1-5 说明了顶点着色器在 WebGL 图形流水线中的作用。

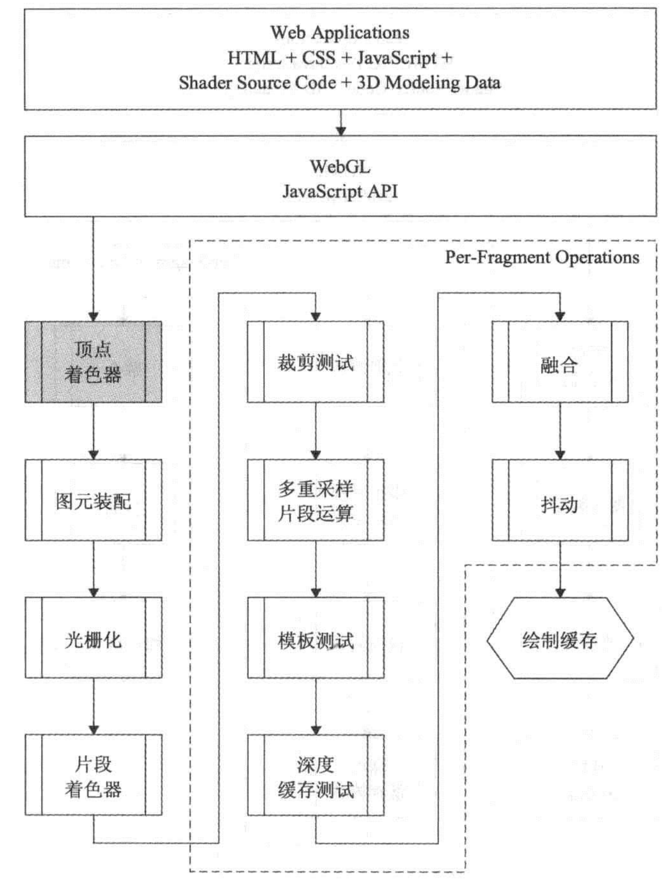


图 1-5 顶点着色器在 WebGL 图形流水线中的作用

顶点着色器是 3D 建模数据(即顶点)经过 JavaScript API 后首先到达的地方。顶点着色器是可编程的，即用户可以自己设计顶点着色器的源代码，并且可以用 JavaScript API 传入

源代码，因此实际上顶点的处理有很多不同的方法。

在着色过程实际开始之前，着色器通常先对顶点进行转变，即乘以一个变换矩阵。把一个对象的全部顶点都乘以一个变换矩阵，其实际作用相当于把每个对象放置在场景中的某个位置。在本章后面和第 4 章将进一步介绍变换矩阵的作用。因此，如果你现在还不能完全理解这些内容，不要太着急。

顶点着色器的输入包括以下内容：

- **顶点着色器的实际源代码。**这些源代码是用 OpenGL ES 着色语言(OpenGL ES Shading Language，GL SL ES)设计的。
- **attribute 变量(属性变量)。**它们是一些用户自定义的变量，它们通常用来包含特定于每个顶点的数据(也有一个名为常量顶点属性(constant vertex attribute)的功能，当我们需要为多个顶点定义同一个属性值时，可以使用这个功能)。顶点位置和顶点颜色就是顶点属性(attribute 变量)的示例。
- **uniform 变量(恒值变量)。**它们用来表示所有顶点都相同的数据。变换矩阵和光源位置都属于 uniform 变量(正如本章后面将指出，在两次 WebGL 绘制调用之间，可以改变 uniform 变量的值。因此，它们只是在一次绘制调用期间保持不变)。

图 1-6 表示了顶点着色器的输出，它包括用户自定义的 varying 变量、一些内置的特殊变量。

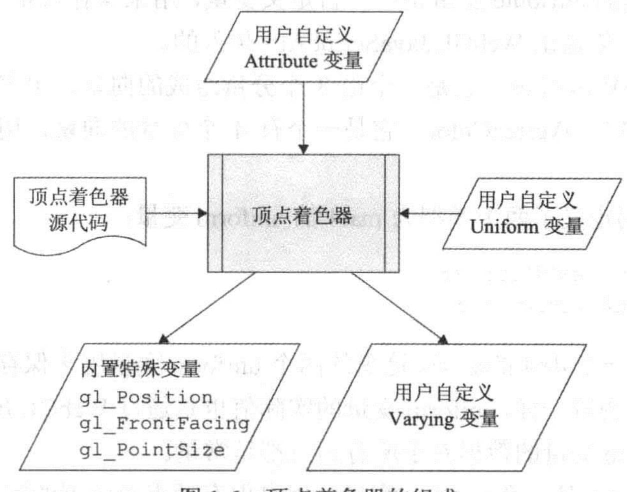


图 1-6　顶点着色器的组成

varying 变量(易变变量)是顶点着色器向片段着色器发送信息的一个手段。本章后面将深入分析这些内置的特殊变量。目前我们暂且只需要知道，内置变量 gl_Position 是最重要的一个变量，而且即使在顶点着色器完成处理之后，它还保存着顶点的位置信息。

以下源代码片段是顶点着色器的一个示例。再次声明，本书后面章节将进一步讨论顶点着色器的内容。这里的源代码只是让你知道顶点着色器的大致结构。

```
attribute vec3 aVertexPos;
attribute vec4 aVertexColor;
```

```
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

varying vec4 vColor;

void main() {
  gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPos, 1.0);

  vColor = aVertexColor;
}
```

正如前面曾提到，这段源代码是用 OpenGL ES 着色语言编写的。正如你将看到，其语法与 C 程序设计语言的语法十分相似。但是也存在某些差别，如两者支持的数据类型不完全一样。但是，如果你以前曾用 C 语言编写过程序，则很多东西看起来非常熟悉。虽然在现在这个特定的时刻，你不需要理解这段程序的每个细节，但是下面将比较深入地介绍这段代码中每条语句的意义。

从程序顶部开始，前两行代码声明了两个属性变量：

```
attribute vec3 aVertexPos;
attribute vec4 aVertexColor;
```

再次说明，这里的 attribute 变量是用户自定义变量，用来保存特定于每个顶点的数据。这些属性变量的实际值是由 WebGL JavaScript API 传入的。

第一个变量是 aVertexPos，它是一个由 3 个分量组成的向量，用来保存一个顶点的位置信息。第二个变量是 aVertexColor，它是一个有 4 个分量的向量，用来保存一个顶点的颜色信息。

接着两行源代码定义了两个类型为 mat4 的 uniform 变量：

```
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
```

类型 mat4 代表一个 4×4 的矩阵。这里的两个 uniform 变量用来保存作用于每个顶点上的变换矩阵。与属性变量一样，uniform 变量的实际值也是通过 WebGL JavaScript API 传入。两者的差别是 uniform 变量的数据对于所有顶点都是常量。

最后，声明 vColor 是一个 varying 变量。用来保存顶点着色器的输出颜色。

```
varying vec4 vColor;
```

顶点着色器用这个 varying 变量把数据传送给片段着色器。

声明了所有变量后，接着是顶点着色器的入口点：

```
void main() {
  gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPos, 1.0);
```

顶点着色器和片段着色器的入口点都是 main()函数。在 main()函数中，第一条语句读出顶点位置(即 aVertexPos 变量中的值)，然后乘以变换矩阵，这样就实现了顶点的变换操

作。把结果写入到一个内置的特殊变量(gl_Position)中，它保存了顶点在顶点着色器中处理完成后的位置信息。顶点着色器最后需要执行的操作是读取由 WebGL JavaScript API 传送来的颜色属性值，并把这个值写到 varying 变量(vColor)中，供后面的片段着色器使用：

```
    vColor = aVertexColor;
}
```

1.5.2　图元装配

在顶点着色器之后是图元装配(primitive assembly)。在这一步操作中，WebGL 流水线需要把已经着色的顶点装配成三角形、线段或点精灵(point sprites)等几何图元。然后，对每个三角形、线段或点精灵，WebGL 需要判断它们在当前时刻是否位于屏幕上可见的 3D 区域中。在大多数情形，我们把这个 3D 可见区域称为视锥体(view frustum)。此视锥体是一个底部为矩形、截去顶部的金字塔。如图 1-7 所示。

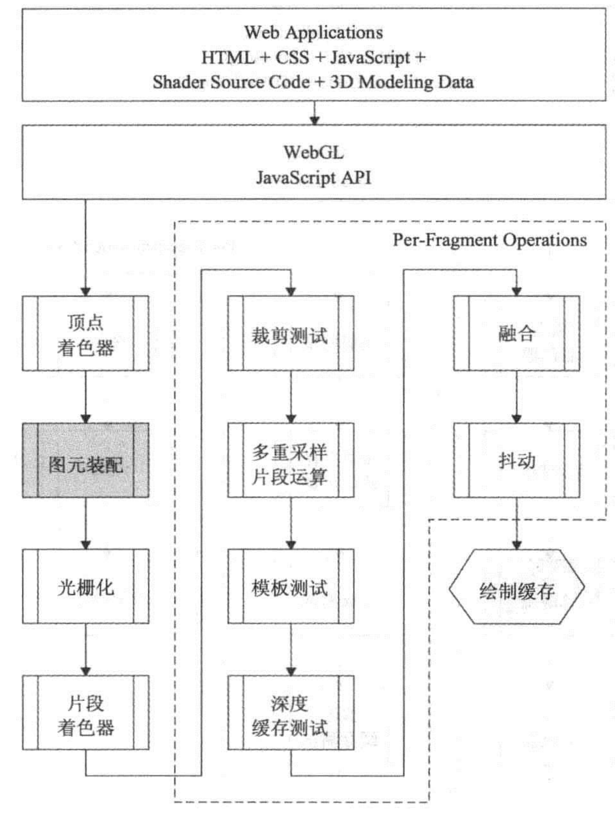


图 1-7　图片装配阶段在 WebGL 流水线中的位置

位于此视锥体内的图元传送到流水线的下一步操作。删除完全处于此视锥体之外的图元，对于部分处于视锥体之外的图元，裁剪掉图元中位于视锥体之外的部分。图 1-8 显示视锥体的一个示例，其中立方体位于这个视锥体之内，而圆柱体位于它之外。构成立方体的图元传送到流水线的下一个阶段，而构成圆柱体的图元在这个阶段中就被丢弃。

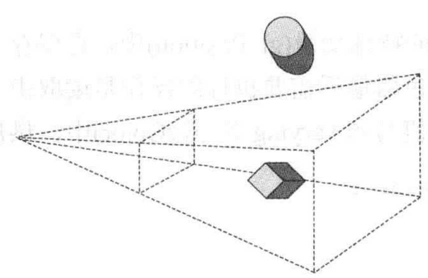


图 1-8　立方体位于其中而圆柱体位于其外的视锥体

1.5.3　光栅化

流水线的下一个阶段是把图元(线段、三角形和点精灵)转换为片段，然后把片段传送给片段着色器。我们可以把片段看成最终绘制在屏幕的一个像素。这个转换过程出现在光栅化操作(见图 1-9)。

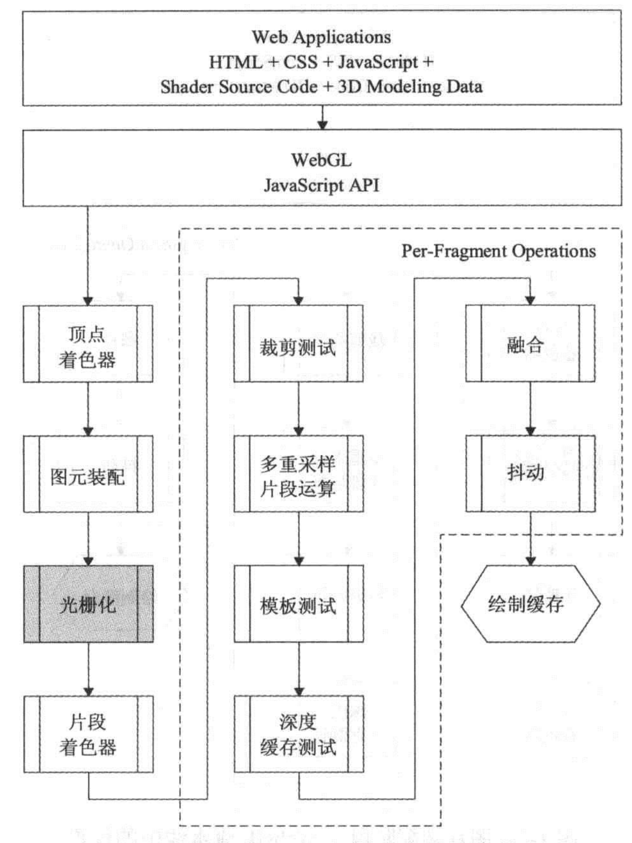


图 1-9　光栅化在 WebGL 图形流水线中的位置

1.5.4　片段着色器

来自光栅化的片段需要发送到流水线的第二个可编程阶段，即片段着色器(见图 1-10)。正如前面曾提到，一个片段实际上对应于屏幕上的一个像素。但是，并非所有的片段都会成为绘制缓存中的像素，因为逐片段操作(后面将要论述)可能会在流水线的最后几个步骤

中丢弃某些片段。因此，WebGL 需要区分片段和像素，把最终能够写入到绘制缓存中的片段称为像素。

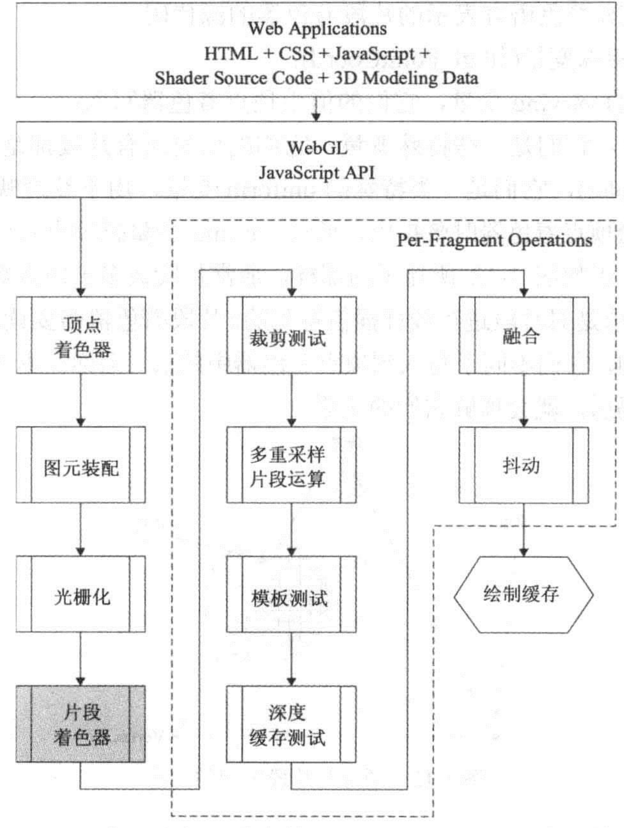


图 1-10　片段着色器在 WebGL 图形流水线中的位置

在其他 3D 绘制 API 中，如微软的 Direct3D，片段着色器实际上也称为像素着色器。图 1-11 说明片段着色器的输入和输出。

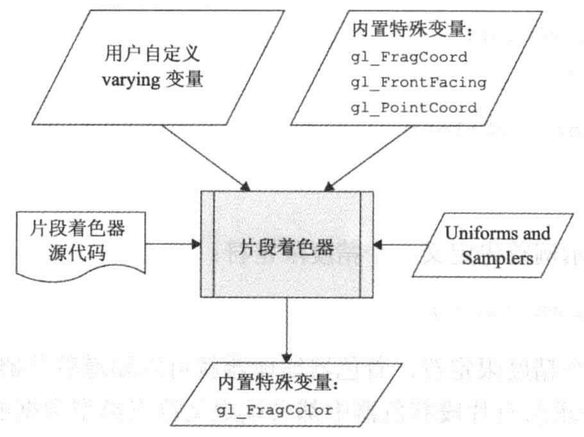


图 1-11　片段着色器的总体结构

片段着色器的输入数据包括以下内容：

- 用 OpenGL ES 着色语言表示的片段着色器的源代码；
- 一些内置的特殊变量(如 gl_PointCoord)；
- 用户自定义的 varying 变量，它们的值由顶点着色器写入；
- uniform 变量，它们是一些特殊变量，它们的值对所有片段都是恒量；
- 采样器(Sampler)，它们是一类特殊的 uniform 变量，用于纹理映射。

正如前面在讨论顶点着色器时曾指出，通过 varying 变量把顶点着色器中的信息传送给片段着色器。然而，正如图 1-12 所显示的那样，通常片段数多于顶点数，写入到顶点着色器的 varying 变量的值是对片段进行线性插值得到的。片段着色器需要读取的 varying 变量值是由线性插值得到的，它们不同于写入到顶点着色器中的值。当你在本书的后面几章接触了较多的着色器源代码后，就会理解它们的关系。

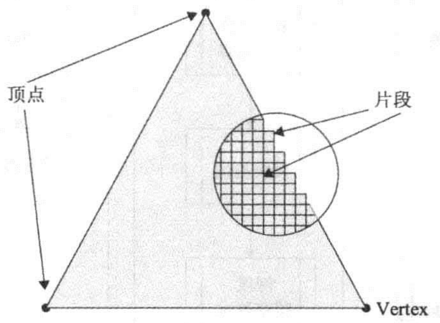


图 1-12　顶点与片段之间的关系

片段着色器通过特殊的内置 gl_FragColor 变量输出结果，片段着色器把片段的颜色写入到这些变量中。下面这段源代码说明一个用 OpenGL ES 着色语言设计的简单片段着色器：

```
precision mediump float;

  varying vec4 vColor;
  void main() {

    gl_FragColor = vColor;

  }
```

这个片段着色器示例首先定义一个精度限定符：

```
precision mediump float;
```

其思想是通过这个精度限定符，着色器编译器就可以知道着色器的变量或数据类型的最小精度。WebGL 要求所有片段着色器中都必须定义浮点类型数据的精度。

在精度限定符之后，片段着色器声明了 vColor varying 变量。在 main()函数中，把 varying 变量 vColor 的值写入到特殊的内置变量 gl_FragColor 中。同样需要注意的是，vColor 变量

的值是由顶点着色器中写入到这个 varying 变量的值线性插值得到的。

1.5.5 逐片段操作

在片段着色器之后，都要把每个片段传送到流水线的下一个阶段，此阶段包含逐片段操作。顾名思义，这步操作实际上包含几个子操作。来自片段着色器的每个片段都可以以不同方式影响绘制缓存中的一个像素，具体取决于逐片段操作的条件和结果。在图 1-13 中，位于右侧虚线边框中的部分属于逐片段操作。为了控制它的行为，可以启动或禁用这些处理。

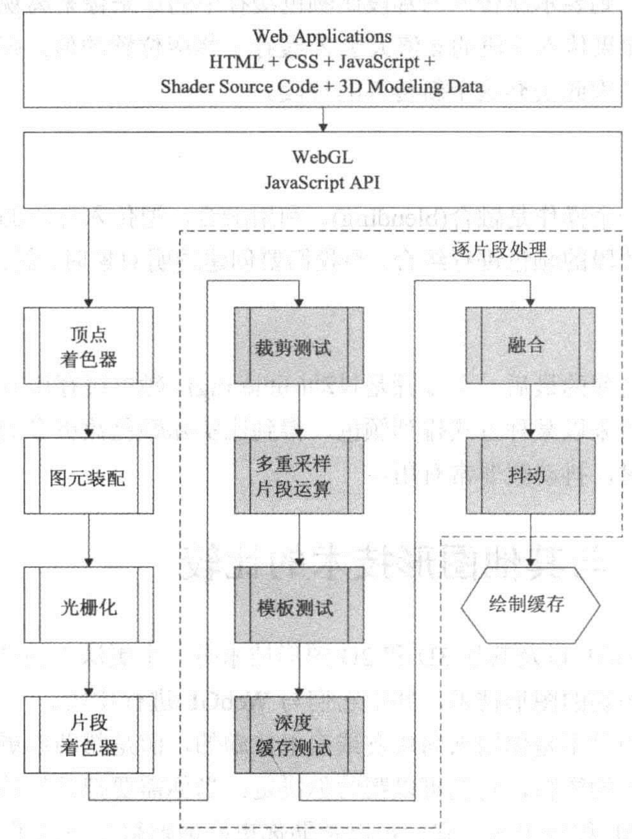


图 1-13 逐片段操作在 WebGL 图形流水线中的位置

1. 裁剪测试

裁剪测试决定片段是否位于裁剪矩形中。此裁剪矩形是由一个左下角的坐标、一个宽度和一个高度决定的。如果片段位于裁剪矩形内，则通过测试，并把片段传递给下一个阶段。如果片段位于裁剪矩形之外，则被丢弃，而且不会到达绘制缓存。

2. 多重采样片段操作

这一步操作修改片段的 alpha 值和覆盖值(coverage)，作为抗锯齿的一个措施。在计算机图形学中，抗锯齿技术是指这样的技术：它用来改善多边形线框的外观使得它们看起来没有锯齿——即在屏幕得到光滑的结果。

3. 深度缓存测试

深度缓存测试根据深度缓存(也称 Z-缓存)的值丢弃输入的部分片段。当我们把一个 3D 场景绘制到 2D 的颜色缓存中时，颜色缓存只存储场景中在某个时刻对观察者可见对象的颜色。有些对象可能会被其他对象遮挡，深度缓存和深度缓存测试决定哪些像素出现在颜色缓存中。对于每个像素，深度缓存都要保存观察者到当前最近图元的距离。

对于传入的片段，把它的 z 值与深度缓存中同一个位置上的 z 值进行比较。如果传入片段的 z 值比较小，则表示新传入的片段比颜色缓存中的原来像素离观察者更近，则传入的片段通过测试。如果传入片段的 z 值大于 Z-缓存中相应位置的值，则表示新片段被绘制缓存中的像素遮挡，因此丢弃这个新传入的片段。

4. 融合

流水线中的下一个操作是融合(blending)。利用融合，把传入片段的颜色与已经在颜色缓存中相应位置的片段的颜色进行组合。当我们要创建透明对象时，就需要使用融合技术。

5. 抖动

位于绘制缓存之前的最后一个步骤是抖动(dithering)。颜色缓存用有限的二进制位数表示每个颜色。抖动用来以某种方式排列颜色，得到比实际颜色数更多的颜色。当颜色缓存可用的颜色数有限时，抖动就非常有用。

1.6 WebGL 与其他图形技术的比较

为了让你对 WebGL 以及其他 3D 和 2D 图形技术有一个更深入的理解，下面几节简单介绍几个与你比较密切的图形技术，并把它们与 WebGL 进行比较。

这些广博的知识并不是你用来向朋友或雇主炫耀的，而是帮助你更好地理解平常看到或听到有关这些技术的资料，它们可以帮助你决定；当你需要把其他技术中的原理或思想引入到自己的 WebGL 程序中时，它们可以帮助你更好地理解这些技术。

1.6.1 OpenGL

长久以来，OpenGL 已成为台式计算机中两个最重要 3D 图形 API 中的一个(另一个是微软的 Direct3D API，下一小节将要介绍)。OpenGL 是一个标准的、跨平台的 3D 图形 API，它可以用在 Linux、Unix 的几个版本、Mac OS X 和 Microsoft Windows 操作系统平台中。

在许多方面，OpenGL 与 WebGL 都非常相似，因此，如果你以前曾有过 OpenGL 的开发经历，则比较容易掌握 WebGL。如果你有过用可编程着色器设计 OpenGL 程序的经历，则更是如此。如果你从来没有使用过 OpenGL，也不要担心，你将在本书中学习如何编写 WebGL 程序。

1. OpenGL 的历史

在 20 世纪 90 年代的早期，SGI(Silicon Graphics)公司是世界上领先的高端图形工作站的制造商。SGI 公司的工作站比起普通的通用计算机具有更好的性能和更多的功能。工作站使用专用的硬件和软件，因此它可以显示复杂的 3D 图形。作为工作站解决方案的一部分，SGI 专门设计了 3D 图形 API，即 IRIS GL API。

随着时间的推移，人们给 IRIS GL API 增加了较多的功能，以至于 SGI 公司想方设法实现向后兼容。IRIS GL API 后来变得越来越难以维护，也越来越难以使用，因此，SGI 公司可能也认识到开发一个开放的标准，使得程序员比较容易建立与他们的图形硬件相兼容的软件会对本公司更为有利。因为，软件是计算机销售的一个重要组成部分。

因此 SGI 逐步淘汰 IRIS，并从头开始设计 OpenGL。OpenGL 是一个开放的、改进的 3D 图形 API。1992 年，SGI 公司推出了 OpenGL 规范的 1.0 版本，并成立一个独立的产业联盟，即 OpenGL 结构评审委员会(OpenGL Architecture Review Board, OpenGL ARB)，由它来决定 OpenGL 的未来。OpenGL ARB 的缔造者包括 SGI、数字设备公司、IBM、Intel 和微软等公司。在以后的几年中，不断有新的成员加入。OpenGL ARB 定期召开会议，给 OpenGL 规范提出修改意见，或者批准修改，或者决定新的发行版本，或者执行性能测试，诸如此类事项。

有时，即使非常成功的企业也会走向衰落。2006 年，SGI 几乎处于破产的边缘。OpenGL 标准的管理事宜从 OpenGL ARB 转交给 Khronos 小组。从那个时候开始，Khronos 小组继续开发和改进 OpenGL。

2. OpenGL 代码示例

自从 1992 年第一个版本发布以后，OpenGL 增加了很多新功能。在开发 OpenGL 新版本时采用的总策略是保证新版本向后兼容。在新版本的规范中，有些功能已标志为“已弃用”。在本节中，我们会看到一段很短的源代码，它可以在屏幕上绘制一个三角形。

在最新的版本中，glBegin()和 glEnd()函数实际上已标志为“已弃用”，但是许多开发人员仍然使用这两个函数，它们出现在许多文献中、现有的 OpenGL 源代码中和 Web 上的 OpenGL 示例中。

```
glClear( GL_COLOR_BUFFER_BIT );// clear the color buffer
glBegin(GL_TRIANGLES); // Begin specifying a triangle
  glVertex3f( 0.0, 0.0, 0.0 ); // Specify the position of the first vertex
  glVertex3f( 1.0, 1.0, 0.0 ); // Specify the position of the second vertex
  glVertex3f( 0.5, 1.0, 0.0 ); // Specify the position of the third vertex
glEnd();                        // We are finished with triangles
```

函数 glClear 清除颜色缓存，这样，颜色缓存不再保存无用的数据或前一帧像素的数据。然后用 glBegin 函数表示开始绘制一个三角形，此三角形用 glBegin 和 glEnd 之间的三个顶点表示。用函数 glVertex3f()确定三个顶点的位置，它们的 z 值都为 0。

有关 OpenGL 的几个要点：

以下是有关 OpenGL 的几个要点：

- OpenGL 是台式机 3D 图形的一个开放标准。
- 现在 OpenGL 规范由 Khronos 组织负责管理和升级，该组织也负责 WebGL 规范。
- OpenGL 主要是一个即时模式 API。
- 升级 OpenGL 使用的总策略是确保新版本向后兼容，因此，通常总是有许多方法实现同一个功能。
- 带有可编程着色器的 OpenGL 与 WebGL 十分相似。当你深入了解 WebGL 后，则应该能够把一些 OpenGL 源代码示例的思想移植到 WebGL 程序中。
- OpenGL 着色器可以用名为 OpenGL 着色语言(OpenGL Shading Language，GLSL)高级语言编写的。

1.6.2 OpenGL ES 2.0

嵌入式 OpenGL (OpenGL for Embedded Systems，OpenGL ES) 是一个以台式 OpenGL 为基础的 3D 图形 API。由于 WebGL 建立在 OpenGL ES 2.0 之上，因此 OpenGL ES 2.0 与 WebGL 最接近。最大的差别是 WebGL 用在 HTML 和 JavaScript 上下文中，而 OpenGL ES 2.0 通常用在 C/C++、Objective C 或 Java 上下文。但是它们的用法也存在一些差别。例如，Khronos 小组在设计 WebGL 规范时决定删除出现在 OpenGL ES 2.0 中的一些功能。

1. OpenGL ES 2.0 简史

OpenGL ES 1.0 是 OpenGL ES 的第一个版本，它是以台式 OpenGL 1.3 规范为基础的。Khronos 小组以 OpenGL 1.3 的功能为基础，删除一些多余的或不适用于嵌入式设备的功能。例如，在台式 OpenGL 1.3 规范中，有这样一个功能：通过调用 glBegin 函数定义需要绘制的图元类型，然后调用 glVertex3f 指定图元的顶点，最后调用 glEnd()函数(你从前一节的代码可以看出)。在 OpenGL ES 中没有这个功能，因为使用顶点数组等原型方法也能够得到同样的结果。

2. OpenGL ES 2.0 代码示例

现在我们介绍一段很短的 OpenGL ES 2.0 源代码。由于 OpenGL ES 2.0 是一个完全基于着色器的图形引擎(与 WebGL 运行模式一样)，即使建立一个简单的示例也需要很多行的源代码。下面这段源代码来自某一个应用程序，此应用程序在屏幕上绘制一个三角形：

```
GLfloat triangleVertices[] = {0.0f, 1.0f, 0.0f,
                             -1.0f, -1.0f, 0.0f,
                              1.0f, -1.0f, 0.0f};

// Clear the color buffer
glClear(GL_COLOR_BUFFER_BIT);
```

```
// Specify program object that contains the linked shaders
glUseProgram(programObject);

// Load the vertex data into the shader
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, triangleVertices);
glEnableVertexAttribArray(0);
glDrawArrays(GL_TRIANGLES, 0, 3);
eglSwapBuffers(myEglDisplay, myEglSurface);
```

在这段程序中，首先定义一个 triangleVertices 数组，这个数组定义了这个三角形的三个顶点。然后调用 glClear()函数清除颜色缓存。这段代码没有说明，如何对顶点着色器和片段着色器进行编译，如何把它们连接成一个程序对象。这些操作必须在调用 glUseProgramObject()函数之前完成。glUseProgramObject()指定绘制对象时需要使用哪个程序对象。程序对象包含已编译的和已连接的顶点着色器和片段着色器。

有关 OpenGL ES 2.0 的几个要点：

以下是有关 OpenGL ES 2.0 的几个要点：

- OpenGL ES 2.0 是一个适用于移动手机等嵌入式设备的、开放的 3D 图形标准。
- OpenGL ES 1.x 和 2.0 是由 Khronos 小组开发和推广的一个规范。
- OpenGL ES 2.0 属于即时模式 API。
- OpenGL ES 2.0 无法与以前的版本实现向后兼容。这是一个不同于台式 OpenGL 的策略。
- OpenGL ES 2.0 与 WebGL 非常相似，因此我们可以把 OpenGL ES 2.0 的源代码和思想移植到 WebGL 中。在 OpenGL ES 2.0 和 WebGL 中都使用 OpenGL ES 着色语言。

1.6.3 Direct3D

DirectX 是微软多媒体和游戏编程 API 的名称。此 API 的一个重要部分是用于 3D 图形编程的 Direct3D。Direct3D 现在应用在许多编程语言中，如 C++、C#、Visual Basic .NET。虽然它得到几个编程语言的支持，但是它只能用在使用 Microsoft Windows 操作系统的计算机中。

从原理上讲，Direct3D 与 OpenGL、OpenGL ES 和 WebGL 有相似之处，因为它也是一个处理 3D 图形的 API。如果你有 Direct3D 方面的经验，则可能已经对图形流水线、Z-缓存、着色器和纹理等有相当的了解。虽然其中部分概念，Direct3D 使用了不同的名称，但是许多原理还是一样的。但是有关 API 的细节方面，Direct3D 与 WebGL 存在很大的差别。

1. Direct3D 简史

1995 年，微软收购了一个名为 RenderMorphics 的公司。这个公司开发了一个名为 Reality Lab 的 3D 图形 API。微软利用 RenderMorphics 的技术开发了 Direct3D 的第一个版本。与此版本一起发行的还有 DirectX 2.0 和 DirectX 3.0。微软决定不采用 OpenGL，但是

想要开发一个专用的 API，即 Direct3D。这个决定导致台式 3D 图形的分崩离析。但是另一方面，它也促使 3D 图形行业的健康竞争，这种竞争最终的产物是 OpenGL 和 Direct3D。

Direct3D 的一个重要的事件是在 8.0 版本中引入可编程着色器。此着色器是用类似汇编语言设计的。在 Direct3D 9.0 中，推出了一个新的着色器编程语言，即高级着色语言(High Level Shading Language，HLSL)。它是微软公司和 NVIDIA 公司联合开发的。HLSL 对于 Direct3D 的意义相当于 OpenGL ES 着色语言对于 WebGL 和 OpenGL ES 2.0 的意义。

2. Direct3D 代码示例

下面这段代码说明了如何用Direct3D代码绘制和显示一个场景。在执行这段代码之前，通常需要先设置和初始化 Direct3D 并建立一个 Direct3D 设备。在这段代码中，我们假设一个全局变量 g_pd3dDevice 保存了一个 Direct3D 设备的指针。通过这个指针访问 Direct3D API 中的函数。

```
void render(void) {

  // clear the back buffer
  g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET,
                      D3DCOLOR_COLORVALUE(0.0f,0.0f,0.0f,1.0f),1.0f, 0 );

  // Signal to the system that rendering will begin
  g_pd3dDevice->BeginScene();

  // Render geometry here...

  // Signal to the systme that rendering is finished
  g_pd3dDevice->EndScene();

  // Show the rendered geometry on the display
  g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

在这段代码中，调用 Clear()函数清除颜色缓存。这个函数看起来比 OpenGL、OpenGL ES 2.0 或 WebGL 中的相应函数要复杂一点，它有很多参数。

前两个参数通常都是 0 和 NULL。我们用这两个参数确定视图的哪部分需要清除。设置 0 和 NULL 表示清除整个颜色缓存。

第三个参数设置为 D3DCLEAR_TARGET，表示要清除颜色缓存。通过这个参数也可可清除 Z-缓存和模板缓存。

第四个参数指定了清除颜色缓存使用的 RGBA 颜色。在这里，把颜色缓存清除为不透明的黑色。

最后两个参数用来设置清除 Z-缓存和模板缓存使用的值。由于本例中，第三个参数表示清除颜色缓存，因此最后两个参数不重要。

在 Direct3D 中，必须调用 BeginScene()，表示开始绘制 3D 场景。调用 EndScene()结

束场景绘制。在这两个函数的调用之间，可以定义所有我们想要绘制的几何对象。在前面这段代码中，没有绘制几何对象的代码。

最后，调用 Present()函数把刚刚绘制得到的后台缓存显示在屏幕上。

有关 Direct3D 的几个要点：

以下是有关 Direct3D 的几个要点：

- Direct3D 是微软专用的 3D 图形标准。
- Direct3D 只适用于使用 Microsoft Windows 操作系统的计算机上。
- Direct3D 使用与 OpenGL、OpenGL ES 2.0 和 WebGL 类似的图形流水线。
- Direct3D 使用 HLSL 语言编写着色器的源代码。这个语言对应于台式 OpenGL 的 GLSL，对应于 OpenGL ES 2.0 和 WebGL 的 OpenGL ES 着色语言。
- Direct3D 的像素着色器对应于 OpenGL、OpenGL ES 2.0 或 WebGL 的片段着色器。

1.6.4 HTML5 画布

HTML5 是 HTML(Hyper Text Markup Language，超文本语言)的第 5 版。HTML5 规范为 Web 开发人员增加了许多新功能。其中最令人感兴趣的是 HTML5 画布(HTML 5 Canvas)。

HTML5 画布是 Web 页面中的一个矩形区域。在这个区域中我们可以用 JavaScript 代码绘制图形。在 WebGL 上下文中 HTML5 画布特别令人感兴趣的原因是，它是 Vladimir Vukićević 最早在 Mozilla 中体验 WebGL 的基础。现在 WebGL 已设计为 HTML5 画布元素的一个绘制上下文(rendering context)。HTML5 也支持早期的 2D 绘制上下文(Canvas-RenderingContext2D 接口)。调用下面的代码可以从画布元素得到 2D 绘制上下文。

```
var context2D = canvas.getContext("2d");
```

同样方法，也可以从画布元素中得到 WebGL 绘制上下文(WebGLRenderingContext 接口)，但是要把其中的 2d 改为 webgl。

```
var contextWebGL = canvas.getContext("webgl");
```

像上面那样提取一个绘制上下文后，就可以通过 context2D 变量调用得到 HTML5 画布(CanvasRenderingContext2D)支持的函数。同样道理，通过 contextWebGL 变量可以调用得到 WebGL 支持的函数(WebGLRenderingContext)。

1. HTML5 画布简史

Apple Mac OS X 操作系统有一个名为 Dashboard 的应用程序。如果你属于 Mac 用户，则可能会熟悉这个程序。Dashboard 实质上是一个包含许多小部件(Widgets)的小型应用程序。这些小部件与 Web 页面一样，它们都是建立在 HTML、CSS 和 JavaScript 等技术之上。

Dashboard 和 Safari Web 浏览器都使用 WebKit 开源浏览器引擎绘制 Web 内容。Apple 公司于 2004 年在 WebKit 中引入了 canvas(画布)标记，从而推出了在这些程序中绘制 2D 图形的新方法。2005 年，它在 Mozilla Firefox 中实现；2006 年，它在 Opera 浏览器中引入这

个标记。后来把 canvas 标记引入到 HTML5 规范中，2011 年微软发布 Internet Explorer 9，它是第一个得到画布支持的 IE 浏览器。

2. 使用 HTML5 画布的代码示例

程序清单 1-1 的代码说明了如何在 JavaScript 脚本中用画布元素绘制一些简单的 2D 图形。整个代码嵌入到一个 HTML 文件中。如果从程序清单的末尾开始看，就会知道，在<body>开始标记与</body>结束标记之间只有一个<canvas>元素。<canvas>元素定义了图形的绘制区域，用 JavaScript 代码调用画布的 API 就可以在这个区域中绘制图形。你还可以看出，onload 事件处理程序定义在<body>标记上，通过它调用 JavaScript 函数 draw()。这意味着，当浏览器载入此文档时，浏览器会自动触发 onload 事件处理程序，然后调用 draw()函数。

再回到程序清单的开头，在<head>开始标记与</head>结束标记之间有 JavaScript 代码。draw()函数的第一个操作是用 document.getElementById()函数读取画布元素的引用，此画布是整个显示区域的一部分。如果调用成功，则利用这个引用及后面的代码得到一个 2D 绘制上下文。

```
var context2D = canvas.getContext("2d");
```

这一行代码已在本节前面讨论过。你在阅读第 2 章的第一个 WebGL 示例时就会明白，这里至此介绍的代码与 WebGL 例中相应的初始化代码非常相似。程序清单 1-1 说明了 HTML5 canvas 标记的使用。

可从 Wrox.com 下载源代码

程序清单 1-1 HTML5 画布标记的使用示例

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <script type="text/javascript">

    function draw() {

      var canvas = document.getElementById("canvas");

      if (canvas.getContext) {

        var context2D = canvas.getContext("2d");

        // Draw a red rectangle
        context2D.fillStyle = "rgb(255,0,0)";
        context2D.fillRect (20, 20, 80, 80);

        // Draw a green rectangle that is 50% transparent
        context2D.fillStyle = "rgba(0, 255, 0, 0.5)";
        context2D.fillRect (70, 70, 80, 100);
```

```
        // Draw a circle with black color and linewidth set to 5
        context2D.strokeStyle = "black";
        context2D.lineWidth = 5;

        context2D.beginPath();
        context2D.arc(100, 100, 90, (Math.PI/180)*0, (Math.PI/180)*360, false);

        context2D.stroke();
        context2D.closePath();

        // Draw some text
        context2D.font = "20px serif";
        context2D.fillStyle = "#ff0000";
        context2D.fillText("Hello world", 40,220);

    }

  }

 </script>

 </head>

<body onload="draw();">

 <canvas id="canvas" width="300" height="300">

  Your browser does not support the HTML5 canvas element.

 </canvas>

</body>

</html>
```

得到了 CanvasRenderingContext2D 对象之后，就可以用它和 API 函数绘制对象。第一个图形是一个红色的矩形。利用 fillStyle 属性把它的填充颜色设置为红色。有好几种方法指定图形的实际颜色，其中包括：

- 使用 rgb()方法，它需要一个 24 位的 RGB 值作为其参数(例如 context2D.fillStyle = rgb(255,0,0);)。
- 使用 rgba()方法，它需要一个 32 位的颜色值作为其参数，其中最后 8 位代表填充颜色的 alpha 通道(例如 context2D.fillStyle=rgba(255,0,0,1);)。
- 使用一个字符串，把填充颜色表示为一个十六进制的数(例如 context2D.fillStyle = "#ff0000")。

对于第一个矩形(即红色矩形)，用前面介绍的第一个方法 rgb()填充它的颜色。然后调用 fillRect()方法实际绘制矩形，并用当前的填充颜色对它进行填充。这个方法需要定义矩

形的左上角的坐标以及矩形的宽度和高度：

```
fillRect(x, y, width, height)
```

下面的代码设置填充颜色，并再次绘制第一个矩形：

```
// Draw a red rectangle
context2D.fillStyle = "rgb(255,0,0)";
context2D.fillRect (20, 20, 80, 80);
```

绘制了第一个矩形之后，用下面的代码绘制第二个矩形：

```
// Draw a green rectangle that is 50% transparent
context2D.fillStyle = "rgba(0, 255, 0, 0.5)";
context2D.fillRect (70, 70, 80, 100);
```

第二个矩形用 rgba()方法设置填充颜色，它的颜色为绿色，而且 50%透明。由于两个矩形部分重叠，我们就会通过绿色矩形看到后面的红色矩形。

第三个图形是一个圆。为了绘制一个圆，首先需要指定路径的起点。路径是指定在一起的一个或多个绘制命令。当所有绘制命令指定之后，我们可以选择是绘制路径的边框还是填充这个路径。在本例中，只需要绘制一个圆，在初学者看来，指定一个路径可能会显得比较复杂，但是如果需要绘制一个复杂的图形，则你就会欣赏 API 这种设计方法。

我们用 beginPath()指定一个路径的开始，然后指定构成这个路径的绘制命令，最后用 closePath()命令结束一个路径。

在本例，路径只有一个命令，即绘制一个圆。但是 canvas API 并没有包含一个显式方法绘制一个圆，而是使用 arc 方法绘制圆。在本例中，使用下面这个方法：

```
arc(x, y, radius, startAngle, endAngle, anticlockwise)
```

x 和 y 是圆的圆心坐标，radius 是圆的半径，startAngle 和 endAngle 指定弧的开始和结束角度，单位为弧度。最后一个参数 anticlockwise 取一个布尔值，它表示弧的方向。

如果在浏览器中载入这段代码，就会看到如图 1-14 所示的结果。

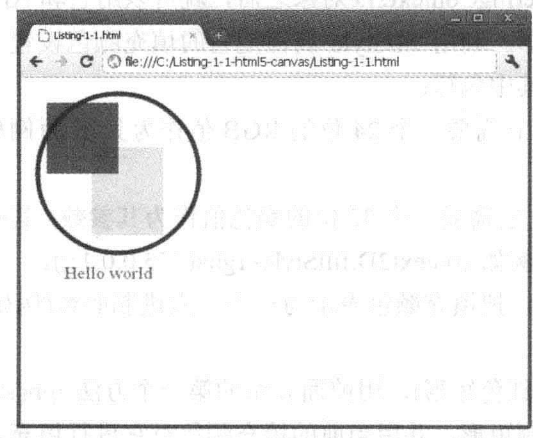


图 1-14　一个简单的使用 HTML5 画布绘制图形的示例

有关 HTML5 画布的几个要点：

以下是有关 HTML5 画布的几个要点：

- HTML5 画布定义一个在 Web 页面上绘制 2D 图形的即时模式 API。
- WebGL 也可以在 HTML5 画布绘制图形，但是它是使用 WebGLRenderingContext 上下文而不是早期的 CanvasRenderingContext2D 上下文。

1.6.5 可缩放矢量图形

可缩放矢量图形(Scalable Vector Graphics，SVG)是一个用 XML 格式描述 2D 图形的语言。顾名思义，它以矢量图形为基础。这意味着，它使用点、线、曲线等几何图元把这些几何图元保存为数学表达式。显示矢量图形的程序(例如 Web 浏览器)根据这些数学表达式生成屏幕上的图像。这样，即使用户放大了图形，用 SVG 生成的图像保持一样的清晰度。在本章介绍的图标标准中，SVG 与 WebGL 关系最不密切。但是你也必须对它有所了解，因为它是 Web 上另一个常用的图形标准，也是属于保留模式 API。

1. SVG 简史

1998 年，微软、Macromedia 和其他一些公司以“建议标准”的形式向 W3C 提供了矢量标志语言(Vector Markup Language，VML)。与此同时，Adobe 和 Sun 公司也提出另一个建议标准，即精密图形标志语言(Precision Graphics Markup Language，PGML)。W3C 对这两个建议进行分析，从 VML 和 PGML 各取一部分组成 SVG 1.0，并于 2001 年成为 W3C 的推荐标准。此后发布了 SVG 1.1 推荐标准，SVG 1.2 推荐标准现在仍处于起草阶段。

除了这些“完全版”SVG 规范外，还有 SVG 移动推荐标准。它包括两个用于移动电话的简化版本。它们是 SVG 微型版(SVG Tiny)和 SVG 基本版(SVG Basic)。SVG 微型版针对非常简单的移动设备，而 SVG 基本版针对高端的移动设备。

2. SVG 代码示例

程序清单 1-2 是一个非常简单的 SVG 代码示例。这段代码在屏幕上绘制一个红色矩形、一个蓝色圆形和一个绿色三角形。

可从 Wrox.com 下载源代码

程序清单 1-2 一个简单的 SVG 代码示例

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">

<rect x="50" y="30" width="300" height="100"
fill="red" stroke-width="2" stroke="black"/>
```

```
<circle cx="100" cy="200" r="40" stroke="black"
stroke-width="2" fill="blue"/>

<polygon points="200,200  300,200  250,100"
 fill="green" stroke-width="2" stroke="black" />

</svg>
```

本书不打算详细讨论这段代码，但是你可能已经看出，SVG 程序是非常紧凑的。如果你在支持 SVG 的 Web 浏览器上执行这段代码，就会看到类似于图 1-15 的结果。

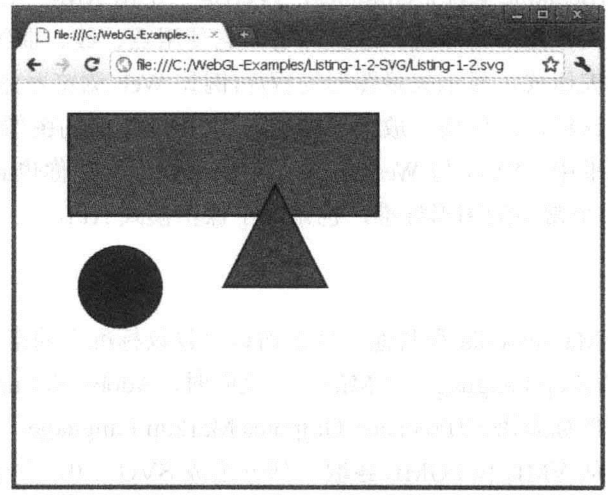


图 1-15　用 SVG 绘制的简单图形

有关 SVG 的几个要点：

以下是有关 SVG 的几个要点：

- SVG 是一个基于 XML 的描述 2D 矢量图形的技术。
- 由于它是一个矢量图形格式，因此 SVG 可以任意缩小和放大，而且不会降低图像的质量。
- 由于 SVG 是基于文本的，因此很容易复制和粘贴图像的部分内容，也很容易用 Web 蜘蛛(Web Crawler)生成搜索索引。
- SVG 完全不同于 WebGL，SVG 是保留模式 API。

1.6.6　VRML 与 X3D

至此，本章已简单地介绍了与 WebGL 上下文关系最密切的几个图形技术。其他两个技术不如前面这些技术有趣，但是它们仍然值得我们去探讨。虚拟现实标志语言(Virtual Reality Markup Language，VRML)和它的后继者 X3D 都是用基于 XML 的技术描述 3D 图形。不管是 VRML 还是 X3D，都没有在任何主要浏览器中实现。如果你想深入了解 VRML 或 X3D，则从 www.web3d.org 和 www.x3dom.org 这两个网站开始。

1.7 线性代数简介

线性代数是高等数学的一部分，它的研究对象是向量和矩阵。为了更好地理解 3D 图形和 WebGL，你最好对线性代数有一个基本的认识。在下面几小节中，我们简单介绍线性代数中有助于理解 3D 图形和 WebGL 图形技术的部分知识。

如果你自认为在这个方面已经具备了相当的知识，则可以跳过这一部分。如果你属于认为高等数学很难或很乏味的那部分人，则作者还是建议你们仔细阅读这里的内容。这里介绍的内容既不像普通的数学课本那样抽象，也没有像高等数学教材那样面面俱到。这里只介绍对理解 3D 图形与 WebGL 图形技术关系十分密切的线性代数知识。

1.7.1 坐标系

为了能够确定在哪个位置绘制 WebGL 对象，需要定义一个坐标系。坐标系有时也称为空间。有很多不同类型的坐标系，但是 WebGL 使用三维、正交、右手坐标系。这听起来有点复杂。

三维表示有三个坐标轴，即通常所说的 x、y、z 轴。正交意味着这三个坐标轴中的任何一个都与其他两个轴垂直，并且它们都归一化为单位长度。右手规则规定了第三个轴的朝向(即 z-轴)。如果 x 轴和 y 轴是正交的，且相交于原点，则有两种方法定位 z 轴的朝向，而且 z 轴都垂直于 x 轴和 y 轴。根据我们的选择，我们称这个坐标系为右手或左手坐标系。

记住右手坐标系的轴方向的一个方法如图 1-16 图示。使用右手，让 x、y、z 轴分别对应于拇指、食指和中指，则拇指指向 x 轴方向，食指指向 y 轴方向，中指指向 z 轴方向。

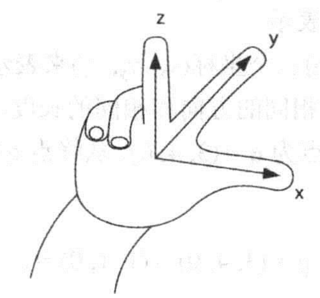


图 1-16 直观观察右手坐标系的一个方法

1.7.2 点与顶点

在 3D 坐标系中的一个点是一个位置，它可以用一个有序的三元组(v_x, v_y, v_z)表示。点的位置是相对于原点的，然后沿 x 轴方向移动 v_x，再沿 y 轴方向移动 v_y，最后沿 z 轴方向移动 v_z 后最后到达的位置。原点的位置(v_x, v_y, v_z) = (0, 0, 0)。

在数学中，点是最基本的构造单元，通过它可以生成其他几何图形。两个点可以定义一个线段，三个点可以定义一个三角形(这里三个点对应于三角形的三个角点)。

当点用来定义 3D 图形中的其他几何图形时，它们通常被称为顶点(vertices，它的单数

形式 vertex)。图 1-17 说明了用三个顶点定义一个三角形的示例。这三个顶点的坐标分别为(1, 1, 0), (3, 1, 0)和(2, 3, 0)，这里坐标系的 z 轴没有画出，但是从纸中垂直穿出。在这个图中，三个顶点用三个填充的小圆表示，这只是为了表示这三个顶点在图中的位置而已，实际上用 WebGL 绘制一个三角形时，这些圆点并不存在。

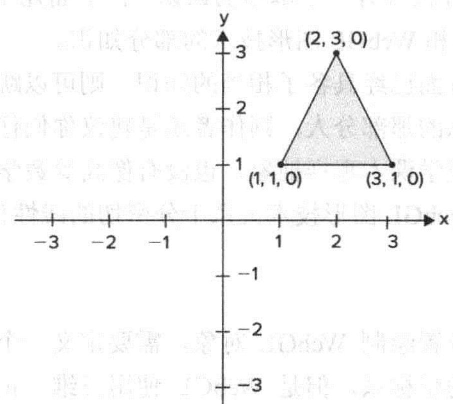


图 1-17　三个顶点定义一个三角形

1.7.3 矢量

许多物理量(如温度、质量或能量)与空间中的任何方向无关，完全由数值大小来定义。这些量称为标量。其他物理量(如速度、加速度或力)需要用一个数值和一个方向才能得到完全的定义。我们称这些物理量为矢量。

矢量是两个点之差，它没有位置，但是有方向和长度。正如图 1-18 的左图所示，一个矢量用一个从起点到终点的箭头表示。

在 3D 空间，一个矢量可以用三个坐标(v_x, v_y, v_z)来表示。这个矢量与以原点为起点以(v_x, v_y, v_z)坐标为终点的矢量具有相同的方向和相同的长度。在图 1-18 的左图中，有一个矢量 u，它的起点为 p = (1, 1, 0) ,终点为 q = (3, 4, 0)。从终点 q 减去起点 p 可以得到这个矢量，如下所示：

$$v = q - p = (3, 4, 0) - (1, 1, 0) = (2, 3, 0)$$

两个矢量可以相加。两个矢量的相加，即把两个矢量的各个分量进行相加，得到一个新的矢量，如下所示：

$$v + u = (v_x + u_x, v_y + u_y, v_z + u_z)$$

两个矢量(u 和 v)的相加运算可以用图形来表示，如图 1-18 右图所示。从图 1-18 可以看出，从矢量 v 的尾端连接到矢量 u 的始端。矢量 w=u+v 是从 u 的尾端连接到 v 的始端。在这个示例中，我们看到两个矢量在 3D 空间中(但是 z = 0)是如何参加相加运算。把 u = (2, 3, 0)加到 v = (1, –1, 0)，得到第三个矢量 w = (3, 2, 0)。

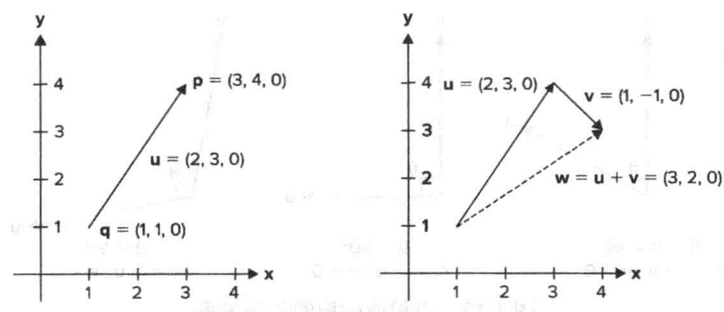


图 1-18　左图是矢量的图形化表示，右图是两个矢量相加运算

矢量也可以乘以一个标量(即一个数值)。结果是一个新矢量，矢量的各个分量都乘以这个标量值。这意味着，对于矢量 u 和标量 k，有以下关系：

$$k\mathrm{u} = (k\mathrm{u}_x, k\mathrm{u}_y, k\mathrm{u}_z)$$

如果一个矢量乘以标量 $k = -1$, 就得到一个与原矢量大小相等、方向相反的矢量。

在 WebGL 中，矢量很重要——例如，用矢量指定光线的方向和视线的方向。使用矢量的另一个示例是表面的法线矢量(简称为法矢量)。法矢量垂直于表面，用法矢量表示表面的朝向。

正如前面曾提到，矢量在基础物理中有很多用途，例如，用矢量表示速度。如果你打算用 WebGL 设计游戏时，矢量就非常有用。假设我们编写一个 3D 游戏。在游戏中，太空飞船的速度为 20m/s，方向向东，与地面平行。则可以用一个矢量表示它：

$$\mathrm{v} = (20, 0, 0)$$

速度的大小就是矢量的长度，太空飞船的方向由矢量的方向表示。

1.7.4　矢量的点积或标积

假如在 3D 空间有两个矢量，它们的相乘有两种方式：

- 点积或标积(scalar product)
- 叉积(cross product)

这一节中我们学习点积或标积。正如标积这个名称所表示的那样，相乘结果是一个标量，不是一个矢量。假设有两个矢量 u 和 v。它们的点积定义为：

$$u \cdot v = |u||v|\cos\theta$$

根据这个定义，点积与两个矢量的长度和它们之间的最小夹角 θ 有关，如图 1-19 所示。由于 cos 90° 等于 0，因此夹角为 90° 的两个矢量的点积等于 0。反之亦然，即，如果两个矢量的点积为零，则这两个矢量肯定是正交的。第 7 章将介绍如何利用矢量的点积计算表面反射的光。

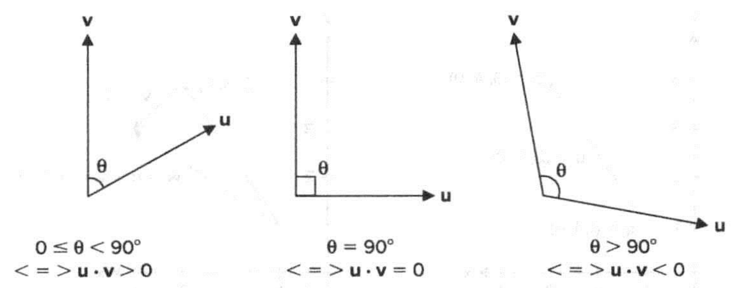


图 1-19　点积或标积的几何关系

在图 1-19 中，左图表示两个矢量的夹角小于 90° 的情况，其点积结果大于 0。在中间图中，夹角为 90° ，它们的点积为 0。在右图中，夹角大于 90° ，它们的点积小于 0。

点积的第二个定义采用代数形式：

$$u \cdot v = u_x v_x + u_y v_y + u_z v_z$$

这个定义与第一个定义是等效的，但是它使用另一种形式。本章的后面将要介绍，如何用点积的这个定义定义矩阵的相乘运算。

1.7.5　叉积

两个矢量的相乘还有另一种形式，即矢量的叉乘(叉积)。两个矢量的叉乘定义为

$$w = u \times v$$

叉积也有一个代数定义：

$$w = u \times v = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x)$$

叉乘的结果是一个新的矢量，这个新矢量具有以下属性：

- $|w| = |u|\ |v| \sin\theta$ (其中，w 是两个矢量之间的最小夹角)。
- w 正交于 u 和 v。
- w 与 u 和 v 符合右手定则。

第一个属性说明了新矢量的长度等于 u 的长度乘以 v 的长度再乘以 $\sin\theta$，这里的 θ 是 u 和 v 的最小夹角。需要指出的是，根据第一个属性，当且仅当 u 与 v 平行时， $\sin 0° = 0$，因此 $u \times v = 0$ 。

第二和第三个属性规定了新矢量 w 的方向。根据第三个属性，新矢量的方向与 u 和 v 的顺序有关。这意味着叉积不满足交换律。它有以下的关系：

$$u \times v = -v \times u$$

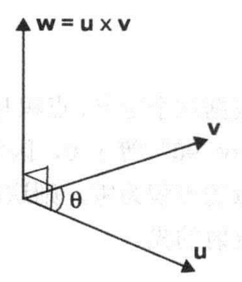


图 1-20　叉积的几何表示

图 1-20 说明了叉积的几何关系。

叉积在 3D 图形中的一个重要应用是计算三角形等

表面的法矢量。在第 7 章中，你将会明白，在 WebGL 中进行光照计算时，法线矢量是一个重要的因素。

1.7.6 齐次坐标

正如本章前面曾提到，在 3D 空间中用三个坐标分量指定一个点或一个矢量。然而，由于点和矢量都使用相同的指定方法，这可能会带来混乱。在齐次坐标中，需要加上第四个坐标(w)。对于矢量，$w = 0$；当 $w \neq 0$ 时，则齐次坐标指定一个点。

我们很容易把一个齐次坐标表示的点(p_x, p_y, p_z, p_w)变换为三个坐标分量表示的点，只需要每个分量都除以 p_w。然后用前三个坐标分量(p_x, p_y, p_z)表示 3D 空间中的点。如果有一个点已经用 3D 坐标表示，则只要在第四个位置加上 1，就可以得到相应的齐次坐标表示的点，即($p_x, p_y, p_z, 1$)。

除了点与矢量之间存在差异外，引入齐次坐标的另一个重要因素是：如果一个点可以用 4 个齐次坐标表示，则可以用 4×4 的矩阵表示点的坐标变换(例如平移、旋转、缩放和剪切)。下面介绍矩阵和变换。

1.7.7 矩阵

矩阵由行和列组成。矩阵的每个数值称为矩阵的元素。一个由 m 行和 n 列组成的矩阵，我们称这个矩阵的维数为 m×n。

在 WebGL 中，最常用的矩阵是 4 行 4 列的矩阵，即它们是 4×4，如下所示：

$$\mathbf{M} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix}$$

只有一列的矩阵(即大小为 m×1)称为列矢量，四个元素的列矢量表示为：

$$\mathbf{v} = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

只有一行的矩阵(即矩阵的维为 1×n)称为行矢量。四个元素的行矢量表示为：

$$\mathbf{v} = \begin{bmatrix} v_0 & v_1 & v_2 & v_3 \end{bmatrix}$$

你下面就会看到，在 WebGL 中，列矢量很常见，它们经常用来表示一个顶点。列矢量乘以一个 4×4 矩阵表示对这个顶点执行某个变换操作。

1. 矩阵的相加和相减

只有维数完全相同的两个矩阵才可以相加或相减。两个矩阵的相加运算是对每个元素

进行相加，与两个矢量的相加运算非常相似。使用的示例是最容易理解两个矩阵的相加运算。假设有两个矩阵 A 和 B，它们的维数相等：

$$\mathbf{A}=\begin{bmatrix}1 & 4 & 2\\ -1 & 0 & 3\end{bmatrix} \qquad \mathbf{B}=\begin{bmatrix}1 & 2 & 5\\ 4 & 1 & 2\end{bmatrix}$$

则它们相加时，得到矩阵 C，如下所示：

$$\mathbf{C}=\mathbf{A}+\mathbf{B}=\begin{bmatrix}2 & 6 & 7\\ 3 & 1 & 5\end{bmatrix}$$

矩阵的相减运算与相加运算的定义相似。对于上面的 A 和 B 矩阵，相减运算得到矩阵 D，如下所示：

$$\mathbf{D}=\mathbf{A}-\mathbf{B}=\begin{bmatrix}0 & 2 & -3\\ -5 & -1 & 1\end{bmatrix}$$

2. 矩阵相乘

矩阵相乘是 3D 图形中的一个非常重要的运算。虽然我们不需要像本节那样用手工进行矩阵相乘运算，但是也有必要知道矩阵相乘的实际过程。矩阵相乘的定义如下：只有当矩阵 A 的列数等于矩阵 B 的行数时，矩阵 A 才可以乘以矩阵 B。如果矩阵 A 的行数为 m，列数为 p(即它为 $m\times p$)，矩阵 B 的行数为 p，列数为 n(即它的维数为 $p\times n$),则矩阵 A 乘以矩阵 B 得到一个 m 行 n 列的矩阵($m\times n$)。这可以用下面的公式来表示:

$$[\mathrm{m}\times\mathrm{p}]\,[\mathrm{p}\times\mathrm{n}]=[\mathrm{m}\times\mathrm{n}]$$

因此，如果矩阵 A 为 $m\times p$，矩阵 B 为 $p\times n$。AB 的结果是一个新矩阵，即 AB 的维数为 $m\times n$。新矩阵 AB 在 ij 位置的元素是矩阵 A 的第 i 行与矩阵 B 的第 j 列的标积，即：

$$a_{i0}b_{0j}+a_{i1}b_{1j}+\cdots+a_{ip-1}b_{p-1j}=\sum_{k=0}^{p-1}a_{ik}b_{kj}$$

分析示例可以帮助我们更好地理解矩阵的相乘运算，假设有两个 M 和 N：

$$\mathbf{M}=\begin{bmatrix}2 & -1\\ -2 & 1\\ -1 & 2\end{bmatrix} \qquad \mathbf{N}=\begin{bmatrix}4 & -3\\ 3 & 5\end{bmatrix}$$

则 M 乘以 N 的结果如下：

$$\mathbf{MN}=\begin{bmatrix}2\times4+(-1)\times3 & 2\times(-3)+(-1)\times5\\ (-2)\times4+1\times3 & (-2)\times(-3)+1\times5\\ (-1)\times4+2\times3 & (-1)\times(-3)+2\times5\end{bmatrix}=\begin{bmatrix}5 & -11\\ -5 & 11\\ 2 & 13\end{bmatrix}$$

关于矩阵相乘运算，有一点必须指出，那就是矩阵的顺序非常重要。不能因为 MN 的

积可以像上面这样定义，就认为可以定义 NM，即使两个积都可以定义，但是它们的结果往往是不相等的，因此有：

$$\mathbf{MN} \neq \mathbf{NM}$$

换一个说法，矩阵相乘不符合交换律。在 WebGL 中，最常用的矩阵相乘是两个 4×4 矩阵相乘运算，或一个 4×4 矩阵与一个 4×1 矩阵相乘运算。

3. 单位矩阵和逆矩阵

对于标量，数值 1 有这样的属性：任何数 x 乘以 1，其结果不变。如果 x 是一个数，则对于任何 x，$1\times x=x$ 都成立。与标量 1 对应的矩阵是单位矩阵。一个单位矩阵是这样一个方阵——它的行数等于它的列数，而且对角位置的元素都为 1，其他位置的元素都为零。下面是一个 4×4 的单位矩阵：

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

矩阵 **M** 乘以单位矩阵 **I**，其结果还是矩阵 M。这说明以下的关系总是成立：

$$\mathbf{MI} = \mathbf{IM} = \mathbf{M}$$

既然你现在知道了单位矩阵是与标量 1 相对应，则可能想知道如何求矩阵的“倒数”矩阵。对于除 0 之外的所有数，它的倒数与它自己相乘的结果为 1。例如，对于任何数 x，则存在另一个数 1/x(它也可以表示为 x^{-1})，则它们的积为 1。同样道理，矩阵 M 的逆矩阵用 M^{-1} 表示。同样存在这样的属性：即矩阵 M 乘以它的逆矩阵，结果为单位矩阵。即因此以下公式总是成立的：

$$\mathbf{MM}^{-1} = \mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$$

注意，只有方阵(行数和列数相等的矩阵)才有逆矩阵，但是并非所有方矩阵都有逆矩阵。

4. 转置矩阵

矩阵 M 转置后得到另一个矩阵，它的行是 M 的列，它的列是 M 的行。矩阵 M 的转置矩阵用 M^T 表示，对于任何 $m\times n$ 矩阵都可以定义它的转置矩阵。由于在 WebGL 中经常使用 4×4 矩阵，我们就以这样一个矩阵为例进行说明。假设矩阵 M 定义如下：

$$\mathbf{M} = \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

则它的转置矩阵 M^T 如下所示：

$$\mathbf{M}^T = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

在本书第 7 章将会介绍，矩阵的转置和求逆运算经常用于法线的变换。

1.7.8 仿射变换

在屏幕上显示 3D 模型时，需要把它们变换到不同的坐标系或空间中。变换是这样一个运算：它以顶点或矢量为运算对象，用某种方法对它进行转换。一个特殊的变换是线性变换，它保持矢量相加运算和标量相乘运算不变性。假如用 f 表示一个变换，且有两个矢量 u 和 v，则只有当以下两个条件都满足时，此变换是线性的：

$$\mathbf{f}(\mathbf{u}) + \mathbf{f}(\mathbf{v}) = \mathbf{f}(\mathbf{u} + \mathbf{v})$$
$$k\,\mathbf{f}(\mathbf{u}) = \mathbf{f}(k\mathbf{u})$$

第一个条件解释如下。如果把一变换应用于两个矢量，然后把变换后的矢量相加，则得到的结果与以下方法得到的结果相等：先把两个矢量相加，然后在它们的和上应用变换。

第二个条件也可以得到同样的解释。先对一个矢量进行变换，然后把结果乘以一个标量 k 等于先把这个矢量乘以标量 k，然后再对其标积进行变换得到的结果。

下面是线性变换的一个示例：

$$\mathbf{f}(\mathbf{u})=3\mathbf{u}$$

现在我们可以验证上述这个变换是满足上述这两个条件的。具体来说，假设有两个矢量(p 和 q)，它们的值如下：

$$\mathbf{p} = [0 \quad 1 \quad 2 \quad 3] \qquad \mathbf{q} = [4 \quad 5 \quad 6 \quad 7]$$

我们先来验证第一个条件。即这两个矢量先乘以 3 然后相加得到的结果等于先相加这两个矢量然后再乘以 3 得到的结果：

$$\begin{aligned}\mathbf{f}(\mathbf{p}) + \mathbf{f}(\mathbf{q}) &= 3\times[0 \quad 1 \quad 2 \quad 3] + 3\times[4 \quad 5 \quad 6 \quad 7] \\ &= [0 \quad 3 \quad 6 \quad 9] + [12 \quad 15 \quad 18 \quad 21] = [12 \quad 18 \quad 24 \quad 30]\end{aligned}$$

$$\begin{aligned}\mathbf{f}(\mathbf{p} + \mathbf{q}) &= 3\times([0 \quad 1 \quad 2 \quad 3] + [4 \quad 5 \quad 6 \quad 7]) \\ &= 3\times[4 \quad 6 \quad 8 \quad 10] = [12 \quad 18 \quad 24 \quad 30]\end{aligned}$$

我们发现这两个情形得到的结果是相等的，因此第一个条件显然是满足的。现在考虑

第二个条件，即 $k\,f(u)=f(ku)$。为了更加具体，使用 $k=2$ 和 $u=p=[0\ 1\ 2\ 3]$，过程如下：

$$k\,\mathbf{f}(\mathbf{u}) = 2\times(3\times[0\quad 1\quad 2\quad 3]) = 6\times[0\quad 1\quad 2\quad 3] = [0\quad 6\quad 12\quad 18]$$

$$\mathbf{f}(k\mathbf{u}) = 3\times(2\times[0\quad 1\quad 2\quad 3]) = 6\times[0\quad 1\quad 2\quad 3] = [0\quad 6\quad 12\quad 18]$$

同样可知，此变换满足第二个条件。因此 f(u)=3u 显然是一个线性变换。

线性变换的示例有缩放、旋转和剪切。在 3D 图形中，点或矢量的线性变换用 3×3 矩阵表示。但是除了这些线性变换外，还有一个非常基本但很重要的变换，即平移变换(translation)。平移变换不能用 3×3 矩阵来表示。

仿射变换是这样一个变换：它先执行一个线性变换，再执行一个平移变换。可以用 4×4 矩阵表示一个仿射变换。如果点或矢量采用齐次坐标表示，则把一个 4×4 变换矩阵乘以列矢量，就可以实现仿射变换。这里的列矢量包含点或矢量。

$$\mathbf{Mv} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} m_{00}v_0 + m_{01}v_1 + m_{02}v_2 + m_{03}v_3 \\ m_{10}v_0 + m_{11}v_1 + m_{12}v_2 + m_{13}v_3 \\ m_{20}v_0 + m_{21}v_1 + m_{22}v_2 + m_{23}v_3 \\ m_{30}v_0 + m_{31}v_1 + m_{32}v_2 + m_{33}v_3 \end{bmatrix}$$

下面几小节介绍四种类型的仿射变换。

1. 平移

平移是指对象的每个顶点都移动一个相同的位移。平移矩阵可以表示如下：

$$\mathbf{T}(t_z, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

当我们用上述的平移矩阵对一个点进行平移变换时，其位移是由矢量(t_x, t_y, t_z)表示。图 1-21 是平移一个三角形的示例，其中的位移矢量为$(t_x, t_y, t_z) = (4, 5, 0)$。

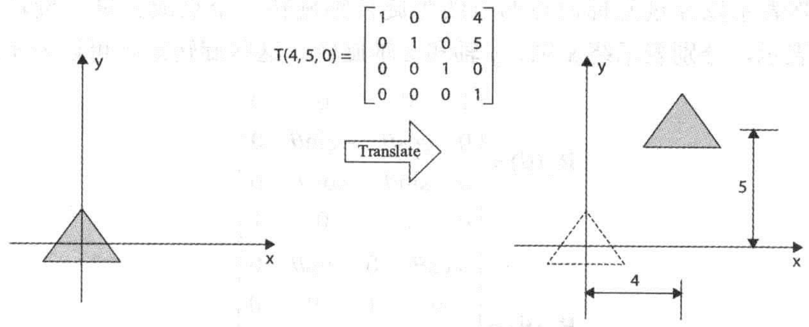


图 1-21　一个三角形的平移变换，沿 x 方向移动 4，沿 y 方向移动 5

左图是三角形在没有应用平移变换之前的位置，右图是三角形沿 x 方向平移 4、y 方

向平移 5 之后的位置。图中没有画出 z 轴，但 z 轴的方向从纸面垂直向外。

现在你知道了矩阵乘法运算，现在就很容易看出，上述的平移矩阵如何作用于一个用齐次坐标表示的点 p 上：

$$\mathbf{p}=\begin{bmatrix}p_x\\p_y\\p_z\\1\end{bmatrix}$$

$$\mathbf{Tp}=\begin{bmatrix}1&0&0&t_x\\0&1&0&t_y\\0&0&1&t_z\\0&0&0&1\end{bmatrix}\begin{bmatrix}p_x\\p_y\\p_z\\1\end{bmatrix}=\begin{bmatrix}p_x+t_x\\p_y+t_y\\p_z+t_z\\1\end{bmatrix}$$

显然可以看出，相乘的结果就是这个点沿着矢量(t_x, t_y, t_z)平移后的位置。

我们注意到，矢量没有位置，但是有方向和大小，因此它不受平移矩阵的影响。记住，齐次坐标表示的矢量的第四个分量为零值。你可证明，矢量 v 乘以平移矩阵后，它的值不受影响。

$$\mathbf{v}=\begin{bmatrix}v_x\\v_y\\v_z\\0\end{bmatrix}$$

$$\mathbf{Tv}=\begin{bmatrix}1&0&0&t_x\\0&1&0&t_y\\0&0&1&t_z\\0&0&0&1\end{bmatrix}\begin{bmatrix}v_x\\v_y\\v_z\\0\end{bmatrix}=\begin{bmatrix}v_x\\v_y\\v_z\\0\end{bmatrix}$$

因此，矢量乘以平移矩阵后，其结果还是原来的矢量，不会产生任何变化，这正是我们所期望的。

2. 旋转

旋转矩阵表示以经过坐标原点的直线为旋转轴旋转一个点或矢量。旋转矩阵通常用 $\mathbf{R}_x$、$\mathbf{R}_y$ 和 $\mathbf{R}_z$ 表示，分别表示绕 x 轴、y 轴和 z 轴旋转。这些旋转矩阵可以表示如下：

$$\mathbf{R}_x(\theta)=\begin{bmatrix}1&0&0&0\\0&\cos\theta&-\sin\theta&0\\0&\sin\theta&\cos\theta&0\\0&0&0&1\end{bmatrix}$$

$$\mathbf{R}_y(\theta)=\begin{bmatrix}\cos\theta&0&\sin\theta&0\\0&1&0&0\\-\sin\theta&0&\cos\theta&0\\0&0&0&1\end{bmatrix}$$

$$\mathbf{R}_z(\theta)=\begin{bmatrix}\cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

正向旋转可以用如图 1-22 所示的右手法则来确定。你想象一下，用右手握着旋转轴，且大拇指的方向指向这个轴的正方向，则其他手指所指的方向就是正向旋转。图 1-22 说明一个绕着 z 轴的旋转，但是这个规则同样适用于任何旋转轴。

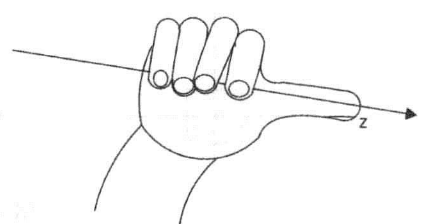


图 1-22　右手法则有助于记住绕某个轴的正向旋转

在图 1-23 中，把旋转矩阵 $\mathbf{R}_z$ 应用于一个三角形，把它绕 z 轴旋转 90°。左图是旋转之前的三角形，右图是应用变换矩阵 $\mathbf{R}_z(90°)$，即绕 z 轴(垂直纸面向外)旋转 90° 之后的三角形。

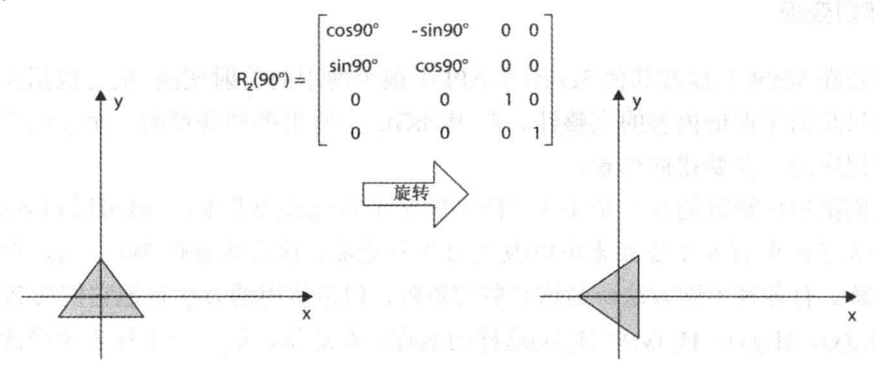


图 1-23　三角形绕 z 轴旋转 90°

可以求出绕任意轴旋转的旋转矩阵，但是作者不想介绍这些细节内容。如果你有兴趣，可以找一本高级的线性代数教材来看，或者在网上查找相关的资料。

3. 缩放

缩放是用来放大或缩小一个对象。下面这个缩放矩阵作用于某一个对象，会使它 x 方向缩放 s_x 倍，y 方向缩放 s_y 倍，z 方向缩放 s_z 倍。

$$\mathbf{S}(s_x,s_y,s_z)=\begin{bmatrix}s_x & 0 & 0 & 0\\ 0 & s_y & 0 & 0\\ 0 & 0 & s_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

如果 $s_x = s_y = s_z$，则这个变换属于均匀缩放。一个均匀缩放会改变对象的大小但是不

会改变它的形状。图 1-24 说明一个正方向，在 x 轴方向放大 2 倍，而 y 轴和 z 轴方向保持不变。

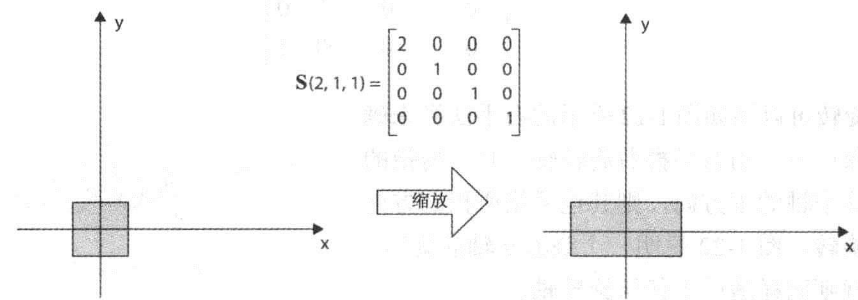


图 1-24 在 x 轴方向放大 2 倍

左图是应用缩放变换之前的正方形，右图是应用缩放矩阵 S(2, 1, 1)之后的图形，可以看出，缩放后变成一个矩形。

4. 剪切变换

剪切是在 WebGL 以及其他 3D 图形 API 中很少使用的仿射变换。把它包括在本书中的主要理由只是为了保证内容的完整性。在 WebGL 中应用剪切变换的一个示例是，在设计一个游戏程序时，需要扭曲场景。

当我们把单位矩阵的左上角 3×3 子阵中的某个零值改为非零值，就可以得到剪切矩阵。左上角 3×3 子阵中有 6 个零元素可以改变为非零元素。这意味着在 3D 空间，有 6 个基本的剪切矩阵。有各种不同方法命名这些剪切矩阵，但是常用的方法是把它们命名为 $\mathbf{H}_{xy}(s)$，$\mathbf{H}_{xz}(s)$，$\mathbf{H}_{yx}(s)$，$\mathbf{H}_{yz}(s)$，$\mathbf{H}_{zx}(s)$和 $\mathbf{H}_{zy}(s)$这样的名称。在这里，第一个下标表示受此矩阵影响的坐标，第二个下标表示执行此剪切的坐标。剪切矩阵 $\mathbf{H}_{xy}(s)$如下所示：

$$\mathbf{H}_{xy}(s)=\begin{bmatrix}1 & s & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}$$

这个剪切矩阵用 y 坐标去改变 x 坐标。如果我们把这个剪切矩阵作用于点 p，就很容易看出这个矩阵如何影响这个点的坐标。

$$\mathbf{H}_{xy}(s)=\begin{bmatrix}1 & s & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}p_x\\p_y\\p_z\\1\end{bmatrix}=\begin{bmatrix}p_x+sp_y\\p_y\\p_z\\1\end{bmatrix}$$

从上述结果更容易看出剪切变换的作用。当 y 值增加时，x 坐标向右切变。图 1-25 是剪切变换的一个示例。

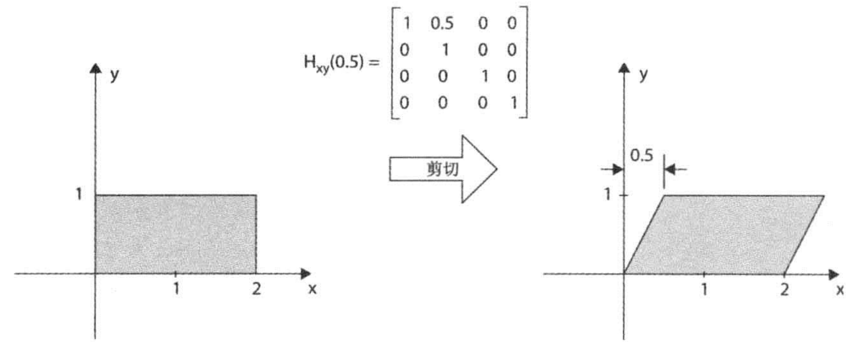


图 1-25　剪切变换的一个示例

1.8　小结

在本章你对 WebGL 有一个初步的了解。通过本章的学习，你掌握了 WebGL 的发展背景以及 WebGL 流水线的组成。此外还对其他 2D 和 3D 图形技术以及它们与 WebGL 的关系有一个基本的理解。

此外，你还学习了线性代数的一些基础知识，它们有助于你更深入理解 WebG 或其他 3D 图形。即使没有理解 3D 图形的细节内容，也能够编写许多基于 WebGL 的应用程序，但是如果你确实掌握了本节的大部分内容，则当遇到问题或需要调试应用程序时，这些知识会对你大有帮助。

第 2 章

创建基本的 WebGL 示例

本章主要内容：

- 创建一个简单但完整的 WebGL 应用程序
- 创建一个 WebGL 上下文
- 编写一个简单的顶点着色器和一个片段着色器，并把它们应用在自己的应用程序中
- 用 WebGL API 载入着色器源代码
- 编译并链接着色器
- 把顶点数据载入 WebGL 缓存，并用缓存绘制场景
- 调试和排除应用程序中的错误
- 用 DOM API 载入着色器

第 1 章只是从理论上介绍 WebGL。本章将向你介绍更多实用知识，且从头到尾介绍一个基本的但是完整的 WebGL 示例。开始时先创建一个尽可能简单的示例。这个示例的全部内容都要在本章中详细介绍，这样你才能够掌握 WebGL 应用程序的绘制过程。

为了使你在本书的剩余部分中比较容易掌握 WebGL，本章还介绍如何调试和排除应用程序中的错误。这里向你提供几个有用的操作技巧，并介绍如何尽快地找出并改正应用程序中的错误。最后将向你介绍如何改进这个初始示例的各部分代码。

2.1 绘制三角形

第一个示例只工作在 2D 图形模式。虽然 WebGL 是一个专门为绘制 3D 图形而设计的 API，但是它可以绘制 2D 图形。第一个简单示例要在黑色的背景上绘制一个白色的三角形，如图 2-1 所示。

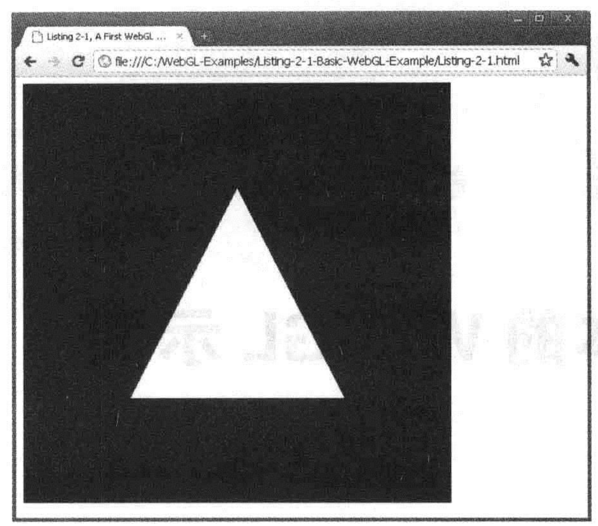


图 2-1　用 WebGL 绘制的一个三角形

在程序清单 2-1 中你会发现，绘制一个三角形需要很多行源代码。一个理由是 WebGL 是完全基于着色器的。这意味着即使在屏幕上绘制一个最基本的对象，也需要一个顶点着色器和一个片段着色器。

编写 WebGL 应用程序时，需要用源代码编写顶点着色器和片段着色器。但是在它们由 GPU 用作着色器程序之前，需要经过运行时的编译和链接。这正是设置和绘制这个简单的示例也需要好几个步骤的原因。下面列出创建一个基本的 WebGL 应用程序所需要的步骤：

(1) 编写一些基本的 HTML 代码，在代码中插入<canvas>标记。<canvas>标记提供了 WebGL 绘制区域。然后编写一些 JavaScript 代码，引用这个画布，这样才能够创建一个 WebGLRenderingContext(WebGL 绘制上下文)对象。

(2) 编写顶点着色器和片段着色器的源代码。

(3) 编写源代码，利用 WebGL API 分别为上述的顶点着色器和片段着色器创建着色器对象。然后把源代码载入到这些着色器对象中，并且编译着色器对象。

(4) 创建一个程序对象并把已编译的着色器对象插入到这个程序对象中。然后链接这个程序对象，再指示 WebGL 如何使用这个程序对象进行绘制。

(5) 设置 WebGL 缓存对象，并把几何对象的顶点数据(本例即这个三角形)载入到顶点缓存。

(6) 指示 WebGL，哪个缓存对应于着色器的属性，最后绘制几何对象(即三角形)。

这些步骤都隐含在程序清单 2-1 中，在后面几小节将向你介绍这个源代码的各个部分组成。

可从
Wrox.com
下载源代码

程序清单 2-1　一个 WebGL 示例，在黑色背景上绘制一个白色三角形

```
<!DOCTYPE HTML>
<html lang="en">
<head>
```

```
<title>Listing 2-1, A First WebGL Example</title>
<meta charset="utf-8">
<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var vertexBuffer;

function createGLContext(canvas) {
  var names = ["webgl", "experimental-webgl"];
  var context = null;
  for (var i=0; i < names.length; i++) {
    try {
      context = canvas.getContext(names[i]);
    } catch(e) {}
    if (context) {
      break;
    }
  }
  if (context) {
    context.viewportWidth = canvas.width;
    context.viewportHeight = canvas.height;
  } else {
    alert("Failed to create WebGL context!");
  }
  return context;
}

function loadShader(type, shaderSource) {
  var shader = gl.createShader(type);
  gl.shaderSource(shader, shaderSource);
  gl.compileShader(shader);

  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert("Error compiling shader" + gl.getShaderInfoLog(shader));
      gl.deleteShader(shader);
      return null;
  }
  return shader;
}

function setupShaders() {
  var vertexShaderSource =
   "attribute vec3 aVertexPosition;                    \n" +
   "void main() {                                      \n" +
   "  gl_Position = vec4(aVertexPosition, 1.0);        \n" +
   "}                                                  \n";

   var fragmentShaderSource =
    "precision mediump float;                          \n"+
```

```
    "void main() {                                        \n"+
    "  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);           \n"+
    "}                                                    \n";

  var vertexShader = loadShader(gl.VERTEX_SHADER, vertexShaderSource);
  var fragmentShader = loadShader(gl.FRAGMENT_SHADER, fragmentShaderSource);

  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);

  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Failed to setup shaders");
  }

  gl.useProgram(shaderProgram);

  shaderProgram.vertexPositionAttribute =
  gl.getAttribLocation(shaderProgram, "aVertexPosition");
}

function setupBuffers() {
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  var triangleVertices = [
         0.0,  0.5,  0.0,
        -0.5, -0.5,  0.0,
         0.5, -0.5,  0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
  gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  vertexBuffer.numberOfItems = 3;
}

function draw() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT);

  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                         vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

  gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
}

function startup() {
  canvas = document.getElementById("myGLCanvas");
```

```
    gl = createGLContext(canvas);
    setupShaders();
    setupBuffers();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    draw();
  }
  </script>

  </head>

  <body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
  </body>

  </html>
```

2.1.1 创建 WebGL 上下文

为了尽可能地简化这个示例，这里把全部的代码都嵌入到一个 HTML 文件中。在这个 HTML 文件中，<head>标记几乎包含了所有的代码。

如果我们先从程序清单 2-1 的源代码末尾开始分析，就会看到 onload 事件处理程序定义在<body>标记上，通过它调用 JavaScript 函数 startup()。具体内容如下所示：

```
<body onload="startup();">
  <canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>

</html>
```

用户输入这个 Web 页面，并且把这个页面的文件和所有外部内容都载入到浏览器时，浏览器触发 onload()事件。当 onload 事件处理程序调用 startup()函数时，我们就知道这个文档已完全载入到浏览器中并且已达到一个稳定状态。因此 startup()函数就是 WebGL 应用程序的入口。

在<body>中，只有一个对象，即 HTML5 的<canvas>标记。给<canvas>标记一个 id，我们就可以在 JavaScript 代码中访问这个画布。<canvas>标记就是 WebGL 使用的绘制区域。这正是 WebGL 图形最终在 Web 页面上绘制图形的地方。

现在来分析<head>中的源代码，就会看到在这个示例中需要编写 JavaScript 代码。在这段 JavaScript 代码的末尾定义了 startup()函数。当 onload 事件触发后并且调用 startup()函数时，这个方法的第一件事情是用 document.getElementById()方法获得到定义在页面正文中<canvas>标记的一个引用：

```
function startup() {
  var canvas = document.getElementById("myGLCanvas");
  var gl = createGLContext(canvas);
```

方法 document.getElementById()不是特定于 WebGL 的，而是属于文档对象模型

(Document Object Model，DOM)API 的一部分。这个 API 定义如何访问文档中的对象，因此浏览器中的 JavaScript 脚本经常调用这个 DOM API 函数。

用 document.getElementById()方法获得画布的引用后，就以它为参数调用自定义 JavaScript 函数 createGLContext()。在这个函数中，通过用一个标准的上下文名称调用方法 canvas.getContext()来创建一个 WebGLRenderingContext 对象。

在 WebGL 规范的演变过程中，使用"experimental-webgl"上下文名称作为 WebGLRendering Context 的临时名称，而在规范最终定稿中用"webgl"名称作为它的正式名称。但是，为了增加用户浏览器能理解这个上下文名称的几率，最好同时使用这两个名称。

WebGLRenderingContext 是一个提供全部 WebGL API 调用的接口。在本书的全部源代码中，把 WebGLRenderingContext 保存在一个名为 gl 的变量中。因此，当你看到以 gl.someMethod()格式调用某个方法时，这表示 someMethod()方法是 WebGLRenderingContext 接口的一部分。或者简单地说，这个方法是 WebGL API 的一部分。

在本书中，当需要讨论 WebGL API 中的某个方法(或者其他任何成员)时，通常在其前面加 gl 前缀。例如，gl.createShader()表示调用 createShader()方法，createShader()方法是 WebGLRendering-Context 的一部分。

初始化 WebGL

本书的示例用一个简单的方法判断是否可以创建一个 WebGL 上下文。“webgl”和“experimental.webgl”这两个上下文名称都需要判断。如果对于这两个上下文名称，canvas.getContext()函数都无法得到一个上下文，则会向用户输出一个 JavaScript 警告信息，提示用户无法创建 WebGL 上下文。

一个稍微复杂的方法是区别创建上下文失败是由浏览器不识别 WebGL 引起的，还是由其他原因引起的。用下面的代码片段可以检查浏览器是否能够识别 WebGL:

```
if (!window.WebGLRenderingContext) {

   // The browser does not know what WebGL is.

   // Present a link to "http://get.webgl.org" where the user
   // can get info about how to upgrade the browser

}
```

如果浏览器可以识别 WebGL，但是调用 canvas.getContext()函数失败，则可以把用户发送到 http://get.webgl.org/troubleshooting。把用户链接到这些 Khronos URL 可以帮助用户获得有关浏览器对 WebGL 支持以及浏览器升级的最新信息。

如果你喜欢自己动手编写代码，则有一个名为 webgl-utils.js 的小型 JavaScript 库，它可以帮助你执行初始化和检查操作，如果初始化失败，它会把用户链接到正确的 URL。Khronos WebGL 维基(www.khronos.org/webgl/wiki)上有这个库。但是，最简单的方法可能还是使用搜索引擎搜索这个库。

2.1.2　创建顶点着色器和片段着色器

所有的 WebGL 程序都必须有一个顶点着色器和一个片段着色器。setupShaders()函数以字符串的形式包含了顶点着色器和片段着色器的源代码。这两个着色器在这个第一个示例中尽量简单化。首先分析顶点着色器：

```
var vertexShaderSource =
  "attribute vec3 aVertexPosition;                 \n" +
  "void main() {                                   \n" +
  "  gl_Position = vec4(aVertexPosition, 1.0);     \n" +
  "}                                               \n";
```

这里把顶点着色器的源代码以字符串的形式赋给 JavaScript 变量 vertexShaderSource。其中的+运算符是 JavaScript 中的字符串连接运算符，它把每行的源代码合并成一个字符串。对于本例这样的小程序，这样做处理是行得通的。但是如果面对一个比较复杂的着色器，则有时用自己的<script>标记在全局级定义一个着色器，然后通过 DOM API 引用它们的源代码，这样做可能会更加方便。

在本章的后面几节中，我们将介绍如何在 WebGL 应用程序中插入着色器源代码，然后利用 DOM API 访问这个着色器。目前只需要知道还有其他方法可以定义着色器源代码。

顶点着色器的第一行代码定义了一个类型为 vec3、名为 aVertexPosition 的变量。vec3 表示一个包含三个分量的向量。在 vec3 类型之前还指定了此变量是属性。属性是一些特殊的输入变量，利用它们把顶点数据从 WebGL API 传送到顶点着色器。

在本例中，利用 aVertexPosition 这个属性把每个顶点的位置传送给顶点着色器，WebGL 根据这些位置绘制一个三角形。为了使顶点顺利通过 API，并最终到达 aVertexPosition 属性中，还需要为顶点设置缓冲区，并将缓冲区绑定到 aVertexPosition 属性上。这两个步骤由后面给出的 setupBuffers()和 draw()函数来实现。

顶点着色器的下一条语句声明 main()函数，它是执行顶点着色器的入口点。main()函数的内容非常简单，它只是把输入的一个顶点位置赋给一个名为 gl_Position 的变量。所有顶点着色器都必须给这个预定义变量赋一个值。当顶点着色器结束处理一个顶点时，这个变量保存了它的位置，并将其传递给 WebGL 流水线的下一个阶段。

本例中的片段着色器也非常简单，具体代码如下：

```
var fragmentShaderSource =
  "precision mediump float;                        \n"+
  "void main() {                                   \n"+
  "  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);      \n"+
  "}                                               \n";
```

这个片段着色器也用一个“+”运算符把各行连接成一个字符串。此片段着色器的第

一行用一个精度限定符声明片段着色器中浮点数的精度，这里使用中等精度。

与顶点着色器一样，在片段着色器中，main()函数定义了它的入口点。在 main()函数的主体中，用 vec4 定义白色，并把这个颜色保存在内置的 gl_FragColor 变量中。这个内置变量是一个包含四个分量的向量，它以 RGBA 格式保存了片段在片段着色器对其进行处理后的输出颜色。

2.1.3 编译着色器

为了创建一个可以载入到 GPU 中且能够绘制几何图形的 WebGL 着色器，首先需要创建一个着色器对象，并把源代码载入到这个对象中，然后编译、链接这个着色器。本节介绍如何载入和编译着色器源代码，下一节介绍如何链接着色器。

自定义的辅助函数 loadShader()创建一个顶点着色器或片段着色器，这要取决于传送给这个函数的参数 type 的值。参数 type 可以设置为 gl.VERTEX_SHADER 或 gl.FRAGMENT_SHADER。函数 loadShader()的代码如下所示：

```
function loadShader(type, shaderSource) {
  var shader = gl.createShader(type);
  gl.shaderSource(shader, shaderSource);
  gl.compileShader(shader);

  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert("Error compiling shader" + gl.getShaderInfoLog(shader));
      gl.deleteShader(shader);
      return null;
  }
  return shader;
}
```

在这段代码中，首先用 gl.createShader()方法创建一个着色器对象。这个方法的参数决定了希望创建的着色器的类型，它可以取 gl.VERTEX_SHADER 或 gl.FRAGMENT_SHADER 值。然后用 gl.shaderSource()方法把源代码载入到着色器对象中。gl.shaderSource()方法的第一个参数表示已经创建的着色器对象，第二个参数表示着色器的源代码。

载入了源代码后，调用 gl.compileShader()方法编译着色器，然后用 gl.getShaderParameter()方法检查编译的状态。如果编译出现错误，则向用户发出一条 JavaScript 警告消息，并删除这个着色器对象，否则返回编译好的着色器。

2.1.4 创建程序对象和链接着色器

setupShaders()函数的第二部分创建一个程序对象，并把编译好的顶点着色器和片段着色器插入到这个对象中，然后把它们链接到一个 WebGL 可以使用的着色器程序。下面是它的源代码：

```
shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
```

```
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);

    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
      alert("Failed to setup shaders");
    }

    gl.useProgram(shaderProgram);

    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
    "aVertexPosition");
  }
```

为了创建程序对象，需要调用一个名为gl.createProgram()的方法，并调用gl.attachShader()方法把编译过的顶点着色器和片段着色器附加到这个程序对象中。然后，调用gl.linkProgram()方法执行链接操作。如果链接成功，就得到一个程序对象，并且可以调用gl.useProgram()方法，告诉WebGL引擎可以用这个程序对象绘制图形。

在链接之后，WebGL实现把顶点着色器使用的属性绑定到通用属性索引上。WebGL实现已为顶点的属性分配了固定数目的“插槽”，通用属性索引就是其中某个插槽的标识符。在顶点着色器中必须知道每一个属性对应的通用属性索引，因为在绘制过程中，就是利用这个索引把包含顶点数据的缓冲与顶点着色器中的属性建立起正确的关联。有两种策略可以获得这个索引：

- 用gl.bindAttribLocation()方法定义在执行链接之前把属性绑定到哪个索引上。定义了这个索引后，就可以在绘制过程中使用这个索引。
- 由WebGL引擎自己决定某个属性要使用哪个索引。在链接完成后，用gl.getAttribLocation()方法得到某个属性的通用属性索引。

本例使用了第二种策略。在连接之后，调用gl.getAttribLocation()方法得到aVertexPosition属性绑定的索引。

把这个索引保存在shaderProgram对象中作为它的一个新属性，并命名为vertexPosition-Attribute。在JavaScript中，一个对象实际上只是一个哈希映射。只要给一个对象的新属性赋一个值就可以创建这个属性。因此，shaderProgram对象并没有一个预先定义的名为vertexPositionAttribute的属性，但是给它赋一个值就可以创建这个属性。后面在draw()函数中，通过保存在这个属性中的索引把包含顶点数据的缓冲绑定到顶点着色器中的aVertexPosition属性上。

很方便给从WebGL API返回的对象(如本节所显示的那样)添加新的属性，你在Web上现有的WebGL代码中也会遇到这种情形。但是如果我们希望得到更健壮的应用程序，它能够处理在WebGL中的丢失上下文，则这不是一个好的策略。第5章将进一步介绍WebGL中的丢失上下文问题，以及如何修改代码以使我们的应用程序更加健壮。在此之前的示例中，我们暂时不考虑丢失上下文问题。

2.1.5 建立缓冲

准备好着色器之后，下一步就是建立缓冲，用来保存顶点数据。在本例中，只需要一个保存三角形顶点位置的缓冲。在名为 setupBuffers()的函数中建立和设置缓冲：

```
function setupBuffers() {
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  var triangleVertices = [
        0.0,  0.5, 0.0,
       -0.5, -0.5, 0.0,
        0.5, -0.5, 0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
  gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  vertexBuffer.numberOfItems = 3;
}
```

这个函数先调用 gl.createBuffer()函数，创建一个 WebGLBuffer 对象，并把它赋给一个全局变量 vertexBuffer，然后把新建的 WebGLBuffer 对象绑定为当前的数组缓冲对象。这相当于告诉 WebGL，从现在开始，这个缓冲对象就是它要使用的对象。三角形的顶点定义在一个名为 triangleVertices 的 JavaScript 数组中。

在本例中，顶点着色器并没有对顶点执行任何的变换操作。你可能还记得，它只是把传入的顶点赋给内置的特殊变量 gl_Position，没有执行任何矩阵乘法运算。

这意味着，仍然使用 WebGL 的默认右手坐标系。这个默认坐标系的坐标原点在(0,0,0)，它位于视口的中央。x 轴沿着水平方向指向右，y 轴方向竖直向上，z 轴方向从屏幕出来指向用户。这三个轴的坐标范围都是从–1 到+1。因此视口的左下角的坐标为 x= –1 和 y= –1，右上角的坐标为 x=1 和 y=1。由于本例绘制的是一个 2D 图形，所有顶点的 z 值都为 0，因此这个三角形绘制在 z=0 的 xy 平面上。

由于定义在数组 triangleVertices 中的全部顶点的 x 和 y 坐标值都在–0.5 和 0.5 之间，因此绘制得到的三角形顶点离视口边缘有点距离。如果修改数组中的坐标值，把–0.5 改为–1.0，把 0.5 改为 1.0，重新载入页面，则会看到三角形的顶点绘制在视口的边缘上。

接着根据包含顶点的 JavaScript 数组创建一个 Float32Array 对象。在第 3 章中，我们将进一步深入讨论这个对象。目前，你只需要知道它用于把顶点数据传递给 WebGL。

调用 gl.bufferData()方法，把顶点数据写入当前绑定的 WebGLBuffer 对象中。这个调用告诉 WebGL，哪些数据保存在用 gl.createBuffer()创建的缓冲对象中。

setupBuffers()函数最后执行的操作是给 vertexBuffer 对象添加两个新的属性并给它们设置适当的初始值。第一个是 itemSize 属性，它定义了每个属性有多少个分量。第二个是 numberOfItems 属性，它定义在这个缓冲中的项或顶点的个数。绘制场景时需要用到这两个属性的值。因此，虽然从上一节中的注解可知，这不是以后解决丢失上下文的最好方法，

但是，在这里给 vertexBuffer 定义属性和值，其接近定义顶点数据的位置。这样，在更新顶点数据的结构时，容易记住需要更新这两个属性。

2.1.6　绘制场景

现在到了最关键的时刻，即绘制所有的对象。我们把场景的实际绘制代码放在一个名为 draw()的函数中：

```
function draw(gl) {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT);

  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                         vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

  gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
}
```

在这个函数中，首先定义一个视口。视口定义了最终绘制的场景在绘制缓冲中的位置。在创建 WebGL 上下文后，程序把视口初始化为一个原点在(0,0)位置的矩形，矩形的宽度和高度为画布的宽度和高度。这意味着调用 gl.viewPort()方法实际上不会影响本例的任何对象。之所以插入这条语句，是因为它是 WebGL 的一个基本方法，你应该熟悉它的用法。

gl.clear()方法通过参数 gl.COLOR_BUFFER_BIT 指示 WebGL 把颜色缓冲清除为事先用 gl.clearColor()函数定义的颜色。本例用 gl.clearColor()把黑色定义为清除颜色。因此，当调用 gl.clear()方法时，颜色缓冲清除为黑色。

在 setupBuffer()函数中，已经创建了一个 WebGLBuffer 对象并通过 gl.bufferData()方法绑定到 gl.ARRAY_BUFFER 目标上。通过 gl.bufferData()方法把顶点数据发送到绑定的缓冲。但是至此还没有告诉 WebGL，顶点着色器中的哪个属性接受来自绑定的缓冲对象的输入数据。在本例中，顶点着色器只有一个缓冲对象和一个属性，但是，通常总是有几个缓存和属性，因此需要为它们创建连接。

WebGL 方法 gl.vertexAttribPointer()把刚刚绑定到 gl.ARRAY_BUFFER 目标上的 WebGL-Buffer 对象赋给一个顶点属性，后者作为一个索引传递给此方法的第一个参数。这个方法的第二个参数表示每个属性的大小或分量数。在本例中，每个属性的分量是 3(因为每个顶点位置用 x、y、z 坐标表示)，在 setupBuffer()方法中已经存储该值，将其作为 vertexBuffer 对象的一个属性(即 itemSize)。

第三个参数表示要把顶点缓冲对象中的值当作浮点数。如果我们传入的数据不是浮点数，则在顶点着色器中使用它之前，必须将其转换为浮点数。第四个参数是规范化标志，它表示是否把非浮点数转化为浮点数。在本例中，缓冲中的数据都是浮点数，因此不需要使用这个参数。第五个参数称为步幅(stride)。当这个参数取值 0 时，表示数据在内存中顺

序存放。第六个参数，即最后一个参数，表示缓冲中的偏移值。由于数据从缓冲的开始位置存放，因此这个参数也设置为 0。

2.2 了解 WebGL 编码风格

如果你是一位经验丰富的程序员，则可能已经知道编码风格指南是怎么回事。例如，编码风格指南包含了如何给变量和函数取名、如何缩进源代码、循环结构必须采用哪种形式、如何使用花括号等规则。有时，其中还包含了一些与应用使用哪些编程语言功能的规则。

其核心思想是如果你(或者你所在的编程小组)在整个编程过程中遵循相同的编码风格，则代码更容易理解，也更容易维护。由于代码更容易理解和维护，因此其中的 bug 也就比较少。

本书并没有打算遵循或强制执行一套完整的编码风格，但是努力在命名约定上尽可能保持一致。然而，本书中的代码示例采用可以反映本书需要(或者你的需要)的编程风格和组织形式，但是这些代码还为没有针对实际的应用程序优化。

例如，本书的 JavaScript 源代码经常直接嵌入 HTML 页面的<head>标记中，而不是把 JavaScript 代码保存在一个外部的 JavaScript 文件中，然后用下面的代码插入这个文件：

```
<script type="text/javascript" src="filename.js"></script>
```

直接在 HTML 页面中插入源代码是为了方便本书内容的介绍，并且在打开一个网页后，在浏览器中选择 View Source Code 命令就可以浏览源代码。但是，你在编写实际的应用程序时最好把 JavaScript 源代码保存一个外部文件中，然后把它包含在 HTML 文件中。

本书使用的命名约定如下：

- 通过调用 canvas.getContext()函数获取的 WebGLRenderingContext 总是保存在名为 gl 的全局变量中。这意味着，如果需要调用 WebGLRenderingContext 的 drawArrays()方法，就直接调用 gl.drawArrays()。
- 通常，采用驼峰式(CamelCase)命名约定来命名 JavaScript 代码和以 OpenGL ES 着色语言编写的着色器源代码中的变量名和函数名。这是指，对于变量名或函数名，第一个字母是小写，然后对于名称中的每个新单词，第一个字母都是大写。作为一个示例，在程序清单 2-1 中有一个设置缓冲的 JavaScript 函数，它使用 setupBuffers()名称。正如你从 gl.drawArrays()名称中可以看出的那样，这个公共的 WebGL API 也遵循驼峰式命名法来命名其方法。
- 在着色器源代码中，属性名使用 a 前缀，可变变量使用 v 前缀，统一变量使用 u 前缀。这样，我们很容易看出，aVertexPosition 表示一个属性，vColor 表示一个可变变量，uMVMatrix 是一个统一变量。

虽然现有的 WebGL 代码使用许多不同的编程风格，但是采用这些命名约定，新的代码就与大量的现有代码相似。当你开始编写自己的代码时，当然可以遵循自己(或者所在团

队)决定的风格。

2.3 调试 WebGL 应用程序

当我们用 C、C++或 Java 等传统的编译语言编写程序时，编译器会告诉我们程序中的错误，如语句的末尾少了一个分号，或者变量名拼写错误。虽然，当今的大多数 JavaScript 实现实际上都在执行 JavaScript 代码之前已把它编译成本地机器码，但是它并没有像传统的编译器那样向最终用户提供有关 JavaScript 代码错误的信息。

这意味着，当我们编写一段 HTML 和 JavaScript 代码并把它载入浏览器中时，即使 JavaScript 代码中的最小错误(使用传统的编译器很容易发现这些错误并通知用户)也会引起这个应用程序表现出与我们的期望完全不同的行为。如果你试图通过手动方式用某个文本编辑器输入程序清单 2-1 的代码，然后把它载入浏览器中，则很可能会遇到这个问题。

幸运的是，这并不是没有解决方法。所有支持 WebGL 的浏览器都提供一些开发工具。如果你以前曾经有过任何类型的 Web 开发的经历，则可能会熟悉这些工具中的一个或多个。下面是一些我们经常使用的开发工具：

- Chrome 开发人员工具——它嵌入到 Google Chrome 浏览器中。
- FireBug——它是 Mozilla Firefox 的一个扩展。
- Web Inspector——它嵌入到 Safari 浏览器中。
- Dragonfly——它嵌入到 Opera 浏览器中。

本书主要介绍 Chrome 开发人员工具的使用，同时也会简单地介绍 Firebug 的用法。如果由于某种原因，你喜欢另一种浏览器或开发工具，那么应该能够在网络上找到相关的资料，然后很容易按照这些资料中的大多数指示进行操作。

2.3.1 使用 Chrome 开发人员工具

Chrome 开发人员工具是一个功能强大的 Web 开发工具，默认情况下集成到 Google Chrome 浏览器中。它是以名为 Web Inspector 的工具为基础的，后者是由 WebKit 开源社区开发的。此 Web Inspector 与 Safari 浏览器内置的 Web 开发工具属于同一个工具。

当我们把网页载入 Chrome 浏览器中时，可以用几种不同的方法启动 Chrome 开发人员工具：

- 在浏览器窗口的右上角，选择扳手菜单，然后选择 Tools | Developer Tools 命令。
- 在 Windows 和 Linux 系统中，可以使用快捷键 Ctrl+Shift+I。在 Mac 计算机中，相应的快捷键为 Command+Option+I。
- 右击网页上的某个元素，在弹出菜单中选择 Inspect Element 命令。

打开 Chrome 开发人员工具后，就会看到如图 2-2 所示的界面。

在图 2-2 中，Chrome 开发人员工具窗口停靠在主窗口的底部，我们可以在背景中看到一些 Web 内容。在 Chrome 开发人员工具窗口的顶部，可以看到一个包含 8 个图标的工具

栏。这些图标对应于 Chrome 开发人员工具中的 8 个面板：

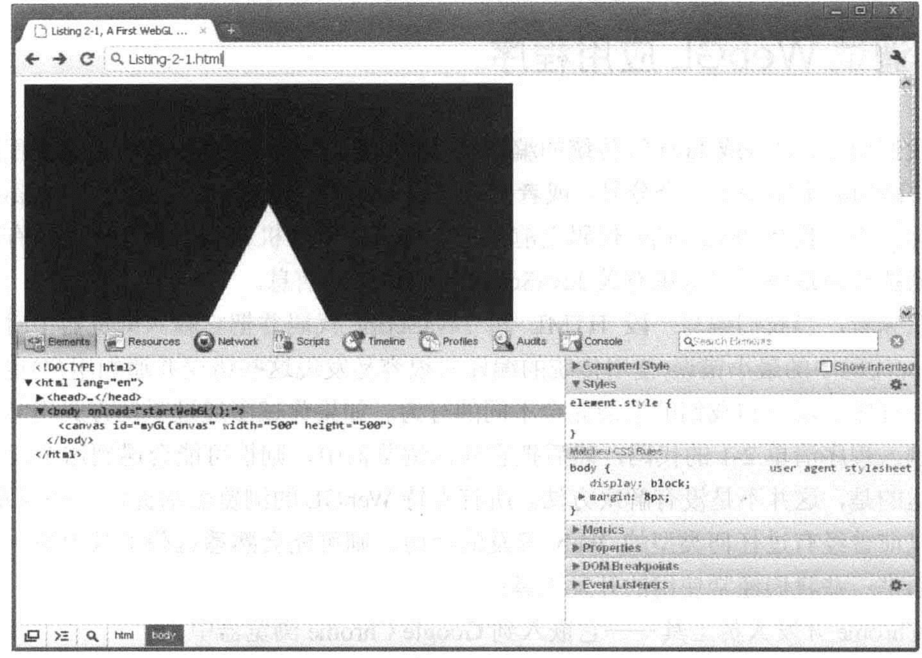


图 2-2　当 Chrome 开发人员工具停靠在主浏览器窗口时的用户界面

- Elements(元素)
- Resources(资源)
- Network(网络)
- Scripts(脚本)
- Timeline(时间线)
- Profiles(统计)
- Audits(审核)
- Console(控制台)

单击一个图标，即可选择相应的面板。在下面几节中，我们将扼要介绍各个面板和它们的作用。

这一节将简单介绍 Chrome 开发人员工具。此外，将把重点放在如何用 Chrome 开发人员工具开发和调试 WebGL 应用程序上。

在 Web 上，Google 公司提供了很多有关 Chrome 开发人员工具工作方式的文档和视频，这些资料都很精彩。你可以阅读这些文档和观看其中一些视频。找到这些信息的最快方法是使用搜索引擎。

1. Elements 面板

Elements 面板以 DOM 树形结构向用户提供了用户的网页所包含的内容。我们可以把

它看成是大多数浏览器提供的 View Source Code 命令的替代品，但是前者比后者更高级。从图 2-3 中可以看到当 Elements 面板取消停靠后的外观。

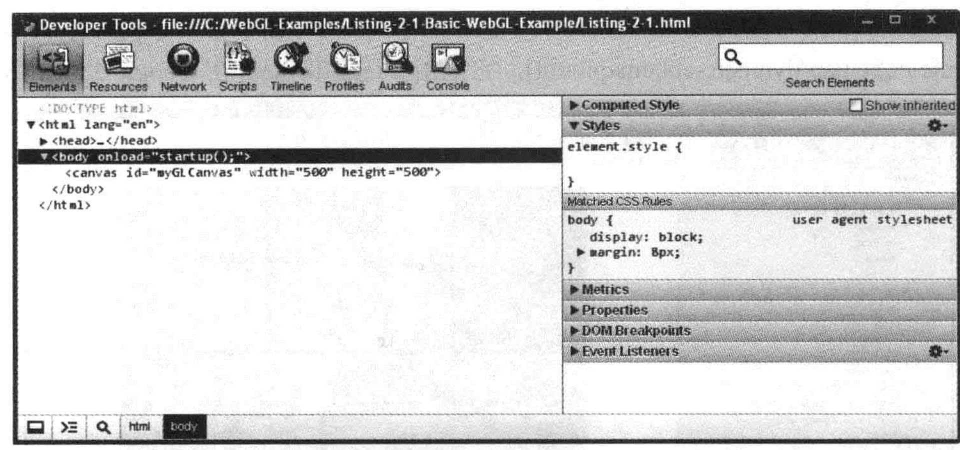


图 2-3　Chrome 开发人员工具中的 Elements 面板

如果我们正在浏览一个网页，并且想知道其中某个元素的细节内容，则单击这个元素，并且选择 Inspect Element 命令。这个命令会打开 Elements 面板，并且此元素处于高亮状态。在图 2-3 中，画布这个元素处于高亮状态。在右边的工具栏中，我们可以找到有关样式、度量、属性和事件监听程序等方面的信息。

WebGL 应用程序并不经常使用这样复杂的 DOM 树。大量的功能出现在 JavaScript 和着色器中。当然，也有例外情形，即使对于包含小型 DOM 树的 Web 应用程序，Elements 面板也会有用。

Elements 面板一个非常有用的功能是我们可以实时修改 HTML、DOM 或 CSS 代码。在第 8 章中，我们将介绍 WebGL 的优化和不同的修改如何影响 WebGL 应用程序的性能。例如，利用 Elements 面板，我们可以实时修改画布的宽度和高度，并且可以看到这些修改如何影响性能。这种测试可以快速给用户提供非常有用的信息，让用户知道自己的 WebGL 应用程序是否存在性能上的瓶颈。

2. Resources 面板

Resources 面板允许我们查看一个 Web 应用程序所包含的全部资源。它允许我们快速看到所有已经使用的资源，包括组成 Web 应用程序的 HTML、JavaScript、CSS 和图像文件。当我们找到某个确实不错的 WebGL 应用程序，并且想知道它是如何构成时，这个面板就非常有用。

利用 Resources 面板，用户很快就可以知道自己的应用程序有多大，是否已使用了 JavaScript 库。Resources 面板也是搜索应用程序中可用资源的一个好方法。例如，当用户想知道一个应用程序如何使用纹理时，可以在 Resources 面板中输入并搜索 texture 单词。找到结果后，按 Enter 键，就可以知道这个单词下一次出现的位置。

在我们浏览左侧栏中的资源时，如果选取一个图像文件，则对应图像显示在面板的右

侧框中。这样，用户就可以知道这个应用程序如何使用不同的纹理。在图 2-4 中，在左侧栏中已经选取了一个图像资源。在右侧框中，我们可以看到这幅图像的实际效果。这个示例来自于 WebGL 的演示程序 Dynamic Cubemap(http://webglsamples.googlecode.com/hg/dynammic-cubemap/dynmaic-cubemap.html)，它是由 Gregg Tavares 在 Google 公司开发的。

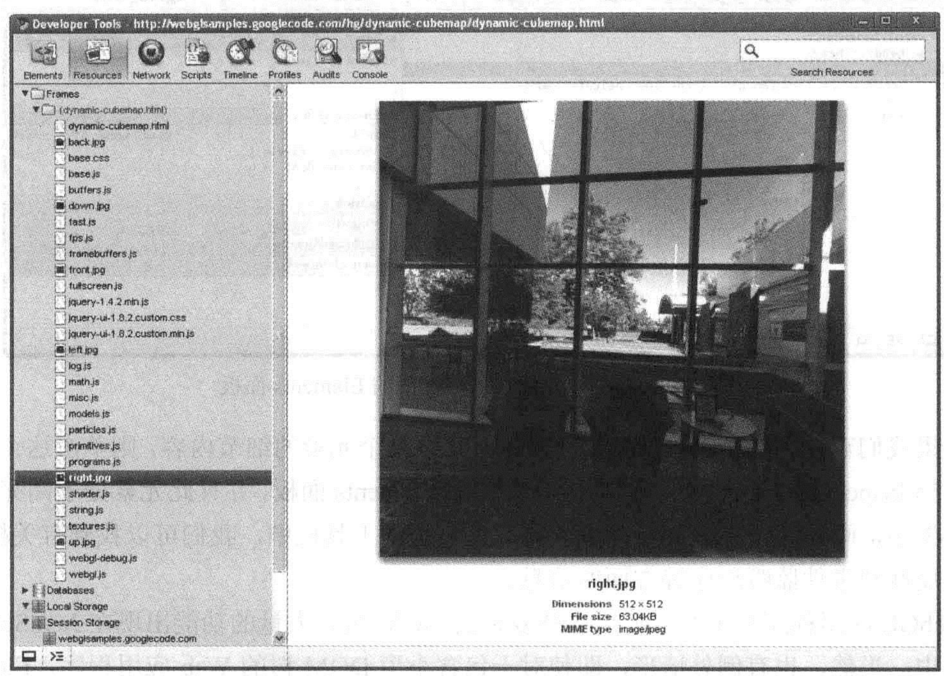


图 2-4　Chrome 开发人员工具的 Resources 面板

3. Network 面板

Network 面板(如图 2-5 所示)允许用户查看哪些资源是通过网络下载的。这个面板可以帮助用户最小化 WebGL 应用程序的启动时间。例如，通过这个面板，我们可以更好地了解资源下载的顺序，以及是否可以加速资源下载的速度。

Developer Tools - http://webglsamples.googlecode.com/hg/aquarium/aquarium.html

Name Path	Method	Status Text	Type	Size Transfer	Time Latency
aquarium.html /hg/aquarium	GET	200 OK	text/html	30.53KB 4.15KB	676ms 491ms
jquery-ui-1.8.2.custom /hg/jquery-ui-1.8.2.custor	GET	200 OK	text/css	30.91KB 5.94KB	522ms 250ms
jquery-1.4.2.min.js /hg/jquery-ui-1.8.2.custor	GET	200 OK	text/javascript	70.48KB 24.03KB	713ms 330ms
socket.io.js /hg/socket.io	GET	404 Not Found	text/html	49B 68B	507ms 323ms
base.js /hg/tdl	GET	200 OK	text/javascript	15.83KB 5.67KB	507ms 323ms
aquarium-common.js /hg/aquarium	GET	200 OK	text/javascript	7.52KB 2.08KB	688ms 505ms
jquery-ui-1.8.2.custom		200		202.14KB	932ms

图 2-5　Chrome 开发人员工具的 Network 面板

4. Script 面板

Script 面板(如图 2-6 所示)为我们调试 JavaScript 代码提供了一个强大的工具。我们可以在代码中设置断点，单步执行，查看栈追踪，查看变量的值。这个面板可能是我们使用 Chrome 开发人员工具开发和调用 WebGL 应用程序时花费最多时间的地方。

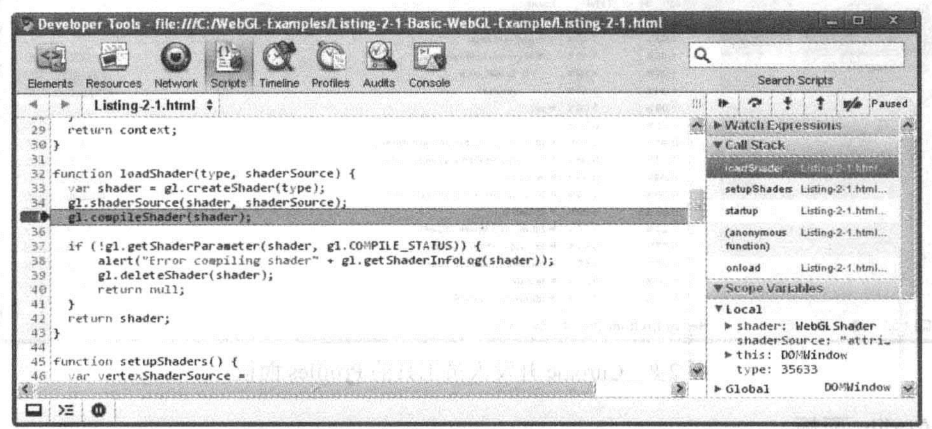


图 2-6　Chrome 开发人员工具的脚本面板

图 2-6 说明应用程序在断点行暂停时 Script 面板的外观。在它的右侧，我们可以看到调用栈、在右上角有以下按钮：继续执行、单步执行函数、单点进入函数、单步退出函数和禁用全部断点。

5. Timeline 面板

Timeline 面板(见图 2-7)为我们提供了载入 Web 应用程序时发生的不同事件的概述。当我们把重点放在开发的 WebGL 部分时，可能不会经常使用这个面板。

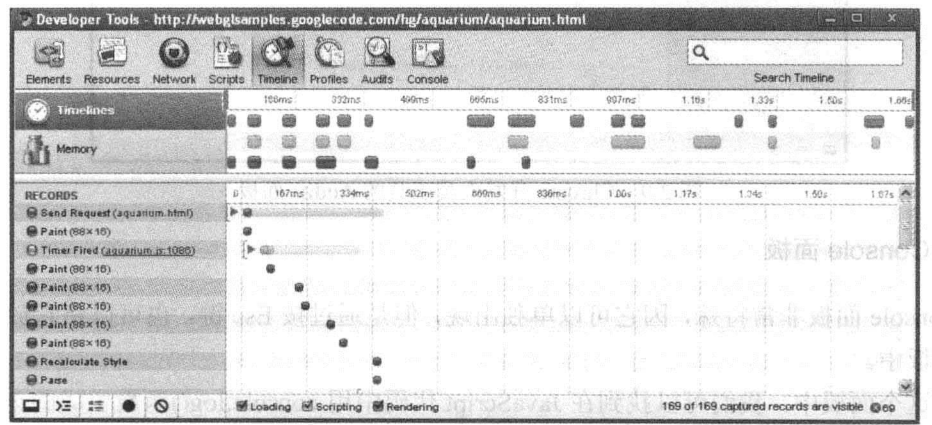


图 2-7　Chrome 开发人员工具的 Timeline 面板

6. Profiles 面板

利用 Profiles 面板，我们可以统计 CPU 和内存使用情况。如果程序需要很多的物理计

算或许多矩阵乘法，或者需要在 CPU 上执行的 JavaScript 逻辑代码，又或者想找出哪些函数占用最多 CPU 时间，则 Profiles 面板可以大显身手。图 2-8 显示了某个时刻统计的结果。

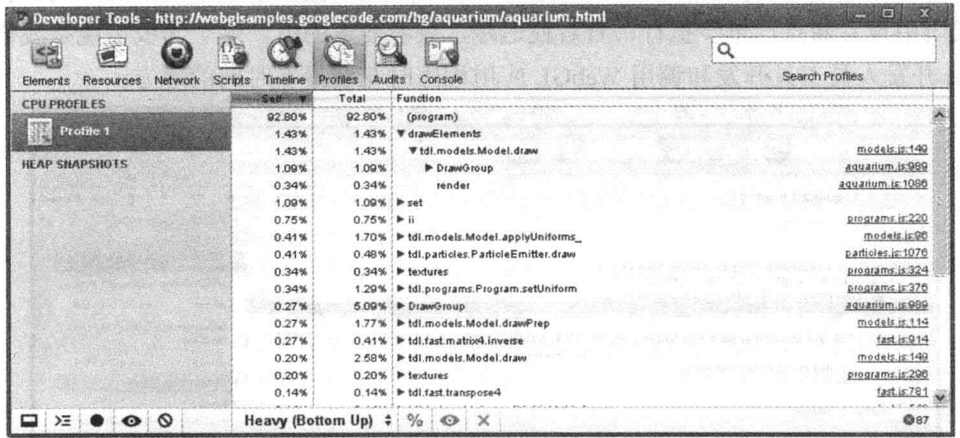


图 2-8 Chrome 开发人员工具的 Profiles 面板

7. Audits 面板

如果用户想改善或优化自己的 Web 程序，但是又不知道如何做，那么 Audits 面板可以帮助您。它向用户提供了哪些地方可以改进的信息，如合并 JavaScript 文件或改进缓存。图 2-9 是启动审核之前的 Audits 面板。

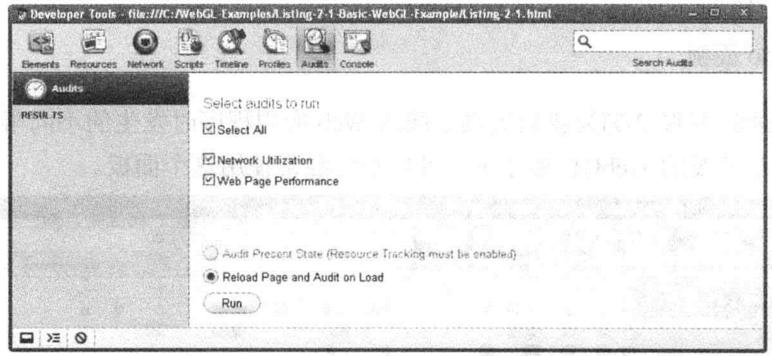


图 2-9 Chrome 开发人员工具的 Audits 面板

8. Console 面板

Console 面板非常特殊，因它可以单独出现，但是通过按 Esc 键，也可以将它放入其他任何面板中。

在这个面板中，我们可以找到在 JavaScript 代码中用 console.log()函数生成的记录，也会获得有关 JavaScript 代码中出现的语法和运行时错误的通知。如果用户正在开发 WebGL 应用程序，并把它载入到浏览器中，但是其没有按预期的要求运行，则 Console 面板是用户首先需要检查的面板之一。最初，用户使用 Console 面板可能是为了查看输出结果，但是它也支持命令行 API，允许用户直接在 Console 面板中缩写命令。如果你对此感兴趣，

可以查看 Internet 上有关 Chrome 开发人员工具的使用文档。

在图 2-10 中，第一行内容是 JavaScript 代码中的 console.log()命令输出的消息。第二行是 JavaScript 异常的结果，这个异常是以下原因引起的：JavaScript 试图调用 gl.drawArray()而非 gl.drawArrays()。在 WebGL API 中没有 gl.drawArray()方法，因此在 Console 面板中出现如图所示的异常。

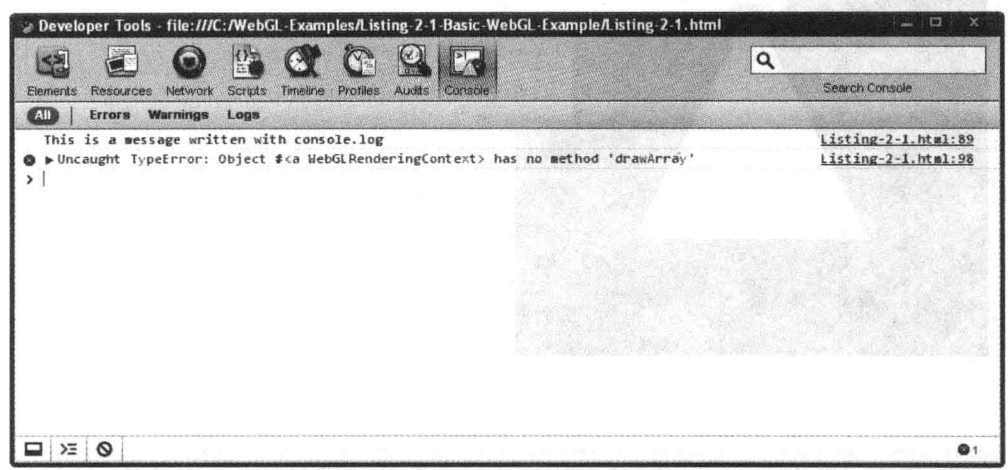


图 2-10　Chrome 开发人员工具的 Console 面板

2.3.2　Firebug 的使用

即使本书的绝大部分示例都是用 Google Chrome 浏览器和 Chrome 开发人员工具开发的，但是本书也会简单介绍如何使用 Firefox 和 Firebug 调试 WebGL 应用程序。毕竟，Firebug 是一个很好的工具，它比 Chrome 开发人员工具更早出现，有很多忠实的用户。

Firebug 是 Firefox 的扩展，它最初是由 Joe Hewitt 开发的。Joe Howitt 也是 Firefox 最早的开发人员之一。它的功能基本上与 Chrome 开发人员工具一样。由于 Firebug 是 Firefox 的一个扩展，因此用户必须单独下载和安装它。用户很容易从 http://getfirebug.com 上下载这个文件。

安装了 Firebug 后，用户就可以用下面的不同方法启动它：

- 利用 Firefox 菜单，即选择 Tools | Firebug | Open Firebug 命令，如图 2-11 所示。
- 用 F12 快捷键打开和关闭 Firebug。
- 右击网页上的某个元素并选择 Inspect Element 命令。

打开 Firebug 后，与 Chrome 开发人员工具界面一样，它也是按不同面板进行组织的。这是因为 Chrome 开发人员工具的开发人员受到 Firebug 的启发。当他们开始开发 Chrome 开发人员工具时，Firebug 已经存在，而且这是一个颇受欢迎且已被证明的好想法。

由于这两个工具具有相似性，因此经常可以互换使用。图 2-12 是 Firebug 的 Scripts 面板，它的作用与 Chrome 开发人员工具的 Scripts 面板一样。用户可以给 JavaScript 代码设置断点、单步执行、分析调用栈等。图 2-12 显示了 JavaScript 在断点处暂停时这个面板的外观。

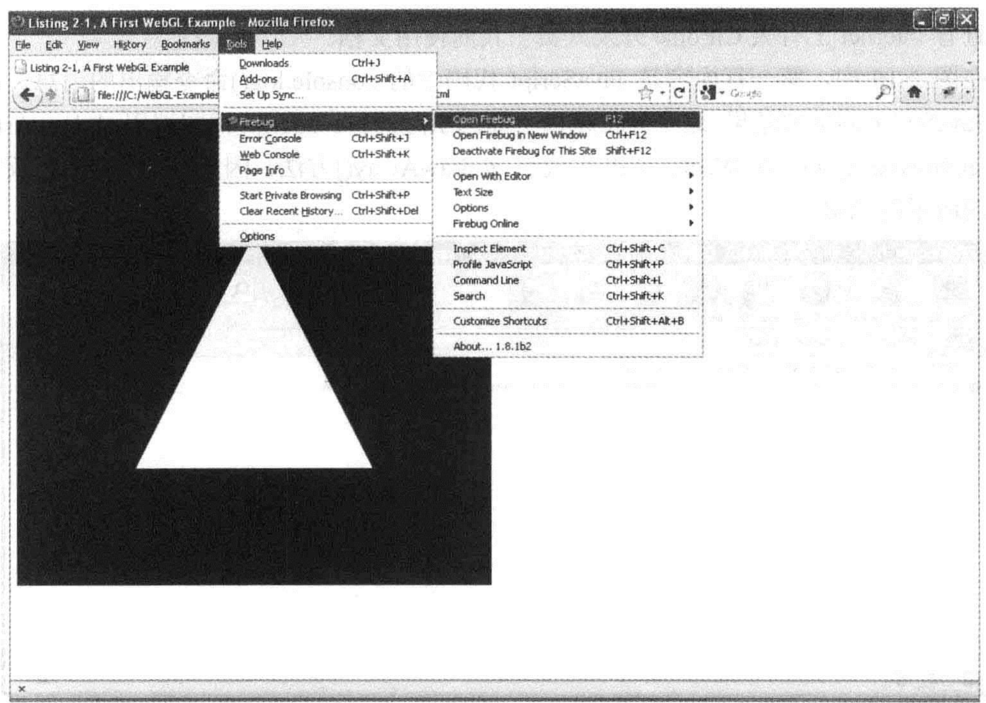


图 2-11　安装了 Firebug 后，从 Firefox 的 Tools 菜单启动它

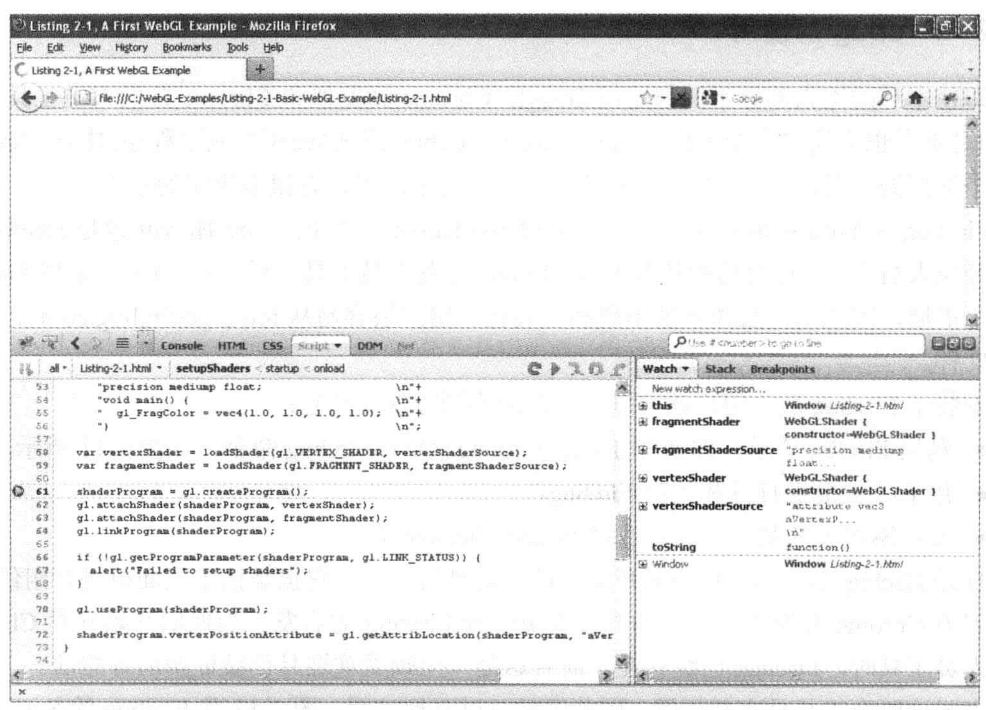


图 2-12　Firebug 的 Scripts 面板

这里需要介绍的最后一个面板是 Firebug 的 Console 面板，它也与 Chrome 开发人员工具的 Console 面板非常相似。图 2-13 就是 Firebug 的 Console 面板。第一行是 console.log()

的输出结果，第二行通知用户在 JavaScript 代码中有一个运行时错误。你可以把图 2-13 与图 2-10 进行比较。在图 2-10 中，Chrome 开发人员工具输出类似的信息。

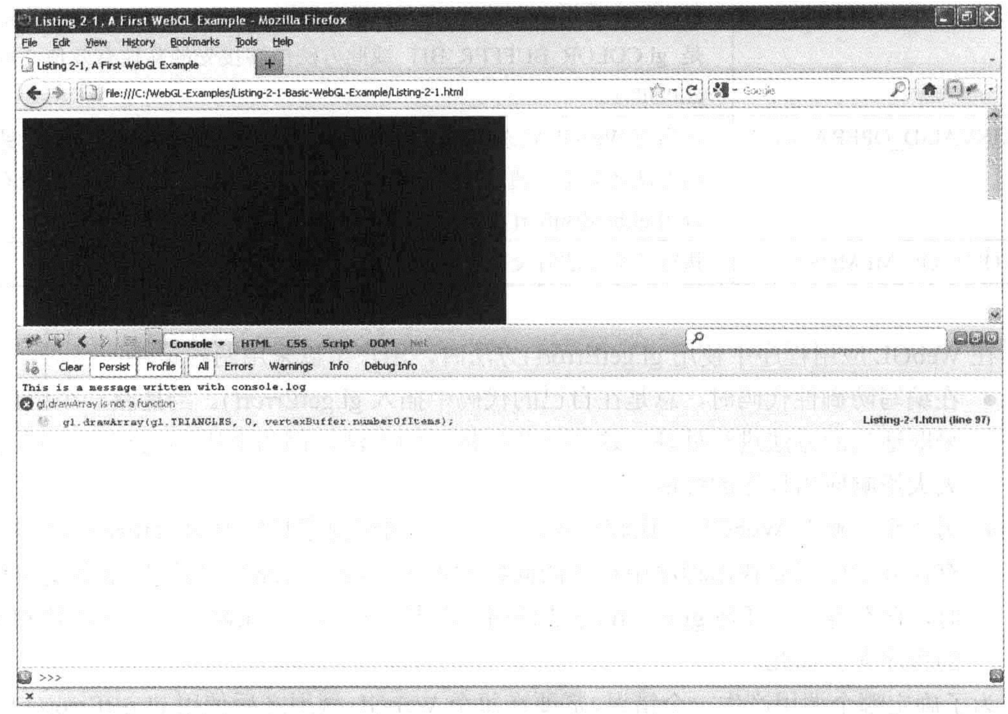


图 2-13 Firebug 的 Console 面板

如果用户掌握了用 Chrome 开发人员工具或 Firebug 工具调试 WebGL 应用程序的技术，则很容易在两者之间互换使用。本书的其余部分使用 Chrome 开发人员工具，这只是因为必须从中选择了一个工具而已，而且在 Chrome 浏览器中默认包括了 Chrome 开发人员工具。

2.3.3 WebGL 的错误处理与错误代码

当 WebGL 检测到代码中的一个错误时，它会生成并记录错误代码。当记录一个错误后，新的错误就不会被记录，直到 WebGL 应用程序调用以下方法为止：

```
gl.getErrors()
```

这个方法用来查询记录下来的错误代码，它返回当前的错误代码(参见表 2-1)，并把当前错误复位为 gl.NO_ERROR。此后，可以重新记录新的错误。

表 2-1

WebGL 错误代码	说　明
gl.NO_ERROR	自上次 gl.getError()调用以来，没有记录到新的错误
gl.INVALID_ENUM	GLenum 类型的参数超出了范围。一个示例是当调用 gl.drawArrays()时，它的第一个参数不是用 gl.TRIANGLES 值或其他此方法可以接受的枚举值，而是用 gl.COLOR_BUFFER_BIT

(续表)

WebGL 错误代码	说　明
gl.INVALID_VALUE	数字型参数超出范围。一个示例是当调用 gl.clear()方法时，传给它的不是 gl.COLOR_BUFFER_BIT 或此方法能够接受的其他有效值，而是 gl.POINTS
gl.INVALID_OPERATION	在当前 WebGL 状态中调用一个不允许使用的方法。当调用生成此错误的方法之前忘记调用正确的方法就会产生此错误。一个示例是没有先调用 gl.bindBuffer()就直接调用 gl.bufferData()方法
gl.OUT_OF_MEMORY	执行某个方法时没有足够的内存

在 WebGL 应用程序中使用 gl.getError()方法时，用户可以采用两种策略：

- 在编写防御性代码时，总是在自己的代码中插入 gl.getError()。当捕获一个错误时，采取适当的方法进行处理。这不是值得推荐的方法，因为调用 gl.getError()方法会大大影响应用程序的性能。
- 另一个策略在 WebGL 中比较常见，它只是在调试程序时使用 gl.getError()方法。只有在开发中想要找出程序中存在的问题时使用 gl.getError()。当程序准备交付使用时，它不再包含任何 gl.Error()方法调用。本书推荐第二个策略，对于性能特别重要的程序尤其如此。

为了确定哪个调用产生一个错误，需要在每个 WebGL 调用之后使用 gl.getError()方法，这个方法可能会大大增加系统的负担，而且代码会变得难以理解。另一个方法是使用这样一个库：它封装了 WebGLRenderingContext，并在每次调用后在后台调用 gl.getError()函数。Chromium(这是一个开源 Web 浏览器项目，Google Chrome 的许多源代码来自这个项目)的作者已经为我们开发一个小型的 JavaScript 库。这个库尽管只有一个文件，但是它非常有用。为了在自己的 WebGL 应用程序使用这个库，用户必须：

(1) 从 https://cvs.hronos.org/svn/repos/registrty/trunk/public/webg/sdk/debug/webgl-debug.js 下载这个 JavaScript 库。

(2) 把它与自己的 WebGL 源代码放在同一个文件夹中，并用下面的代码包含这个文件；

```
<script src="webgl-debug.js"></script>
```

(3) 在创建 WebGLRenderingContext 对象时，调用以下的函数，把它封装到这个 JavaScript 库中：

```
gl=WebGLDebugUtils.makeDebugContext(canvas.getContext("webgl"));
```

执行这些步骤后，所有可以用 gl.getError()方法捕获的 WebGL 错误都输出到 JavaScript 控制台上。

为了让你对 webgl-debug.js 的使用有一个更全面的了解，下面的代码段是程序清单 2-1 中使用了 JavaScript 库 webgl-debug.js 之后的部分代码。

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Snippet demonstrating webgl-debug library</title>
<meta charset="utf-8">
<script src="webgl-debug.js"></script>
<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var vertexBuffer;

function createGLContext(canvas) {
  var names = ["webgl", "experimental-webgl"];
  var context = null;
  for (var i=0; i < names.length; i++) {
    try {
      context = canvas.getContext(names[i]);
    } catch(e) {}
    if (context) {
      break;
    }
  }
  if (context) {
    context.viewportWidth = canvas.width;
    context.viewportHeight = canvas.height;
  } else {
    alert("Failed to create WebGL context!");
  }
  return context;
}
  . . .
function startup() {
  canvas = document.getElementById("myGLCanvas");
  gl = WebGLDebugUtils.makeDebugContext(createGLContext(canvas));
  setupShaders();
  setupBuffers();
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  draw();
}
```

为了测试这个调试库，我们故意删除调用 gl.bindBuffer()方法的语句，让 WebGL 应用程序生成一个错误。图 2-14 就是在这种情形下 Chrome 开发人员工具的 Console 面板显示的内容。如果没有调用 gl.bindBuffer()方法，则调用 gl.bufferData()方法时 WebGL 会生成一个 gl.INVALID_OPERATION 错误。这个调试库调用了 gl.getError()方法，它清空了记录的错误，并且继续其执行，但是没有把任何数据载入到缓冲中。这意味着，在 Console 面板上会看到两个与 gl.INVALID_OPERATION 有关的记录：一个是当调用 gl.vertexAttribPointer()时产生的，另一个是调用 gl.drawArrays()时产生的。

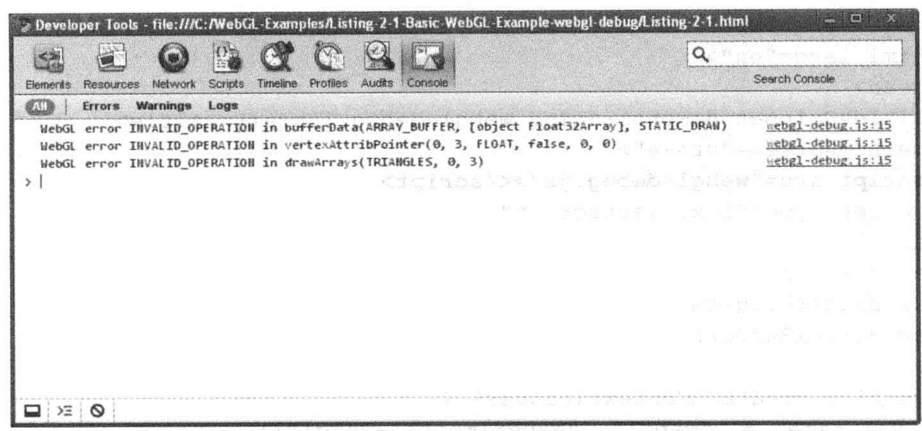


图 2-14　当用 webgl-debug.js 调试库记录 3 个 WebGL 错误时 Chrome 开发人员工具的 Console 面板

当我们使用这个调试库时，在 Console 面板中与 WebGL 错误对应的每行右侧都有一个行号。但是，它并非生成此错误的调用方法语句所在的行号，而是 webgl-debug.js 中 console.log()调用语句所在的行号。如果用户想知道生成此错误的方法调用所在的行，则也不难。打开脚本，在执行 console.log()语句所在的行设置一个断点，然后再次运行这个应用程序。当我们到达这个断点时，我们只需要查看 Scripts 面板中的调用栈就很容易看出哪个 WebGL 方法产生此错误。

注意，由于 webgl-debug.js 调试库使用了 gl.getError()方法，而且这个方法对程序的性能会产生负面的影响，因此在生产代码中不应该使用 webgl-debug.js。

2.3.4 WebGL Inspector

Chrome 开发人员工具和 Firebug 都是很不错的工具，它们对于我们开发和调试 WebGL 应用程序非常有用。但是，它们都是通用的 Web 开发工具，并不是专为 WebGL 设计的。

一个非常有用且专门为 WebGL 设计的工具是 WebGL Inspector。它是一个 WebGL 调试程序，它与 WebGL 的关系就像 PIX 与 Direct3D 的关系以及 gDEBugger 与 OpenGL 的关系，为其提供了许多类似的功能。WebGL Inspector 是一个开源程序，它作为 Chrome 的一个扩展而出现，最初是由 Ben Vanik 开发的。用户可以从 http://benvanik.github.com/WebGL-Inspector 下载这个程序。

安装了 WebGL Inspector 后，把浏览器导航到包含 WebGL 内容的网页。一个带有 GL 文本的红色小图标显示在 Google Chrome 地址栏(有时称为 omnibox)的最右侧。

在本节中，我们利用 Gregg Tavares 在 Google 公司开发的演示程序 Dynamic Cubemap(http://webglsamples.googlecode.com/hg/dynamic-cubemap/dynamic-cubemap.html)说明 WebGL Inspector 的工作方式。

当用户单击 GL 小图标时，有两个名为 Capture 和 UI 的按钮出现在网页的右上角，它们就在 GL 图标的下面。如果单击 UI 按钮，则 WebGL Inspector 的完整 UI 界面出现在 Google Chrome 的底部，我们看到的结果与图 2-15 相似。初始时，UI 界面没有包含任何记录的信息。如果再次单击 UI 按钮，则完整 UI 界面就会消失，屏幕上只有 Capture 和 UI 按钮。

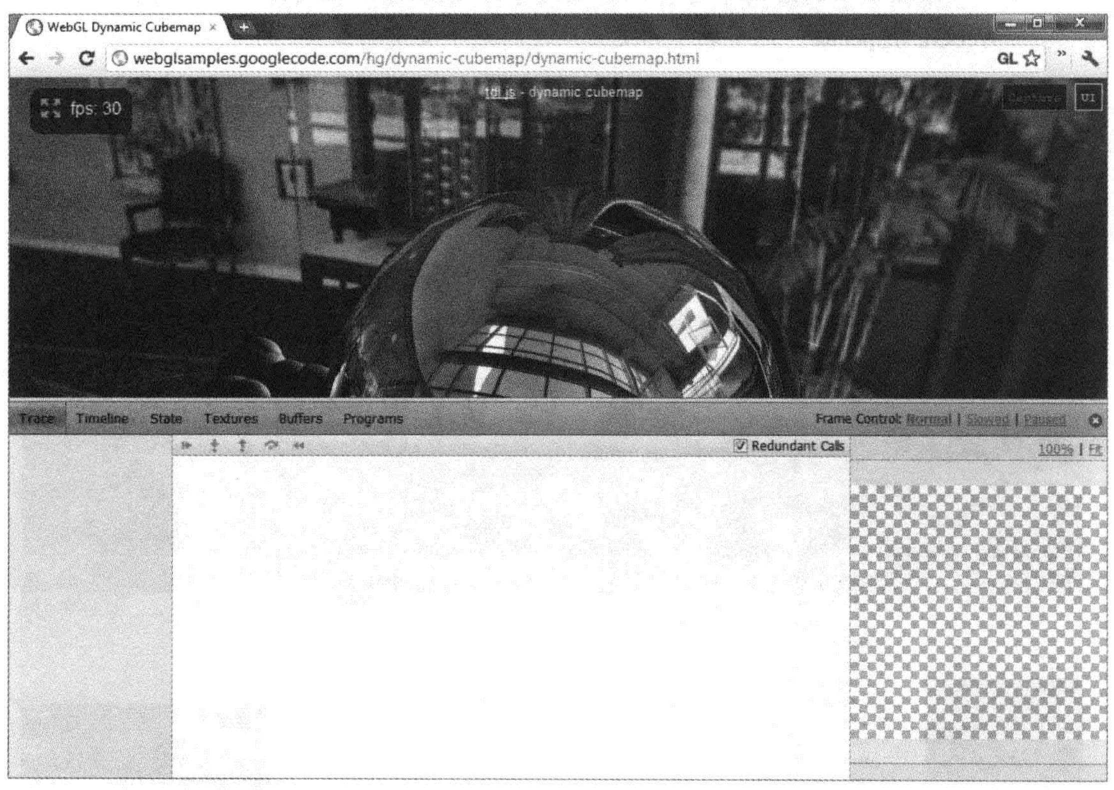


图 2-15　WebGL Inspector 在捕获到任何信息之前的用户界面

当用户单击 Capture 按钮时，系统就记录下 WebGL 应用程序的一帧内容。记录了一帧内容后，我们就可以利用 WebGL Inspector 分析许多与正在调试的 WebGL 应用程序有关的信息。

WebGL Inspector 在其窗口顶部有一个工具栏，里面有 6 项，它们是：

- Trace(跟踪)
- Timeline(时间线)
- State(状态)
- Textures(纹理)
- Buffers(缓冲)
- Programs(程序)

每个工具栏项对应一个面板，单击一项就会出现一个相应的面板。因此，它的界面组织与 Chrome 开发人员工具和 Firebug 十分相似。WebGL Inspector 包含许多有用的功能，深入学习这个工具的最好方法是亲自动手实验。在以下几节中会有选择地介绍其中几个功能。

当用户使用 WebGL Inspector 时，它提供的一些信息用户可能目前还无法理解。在以下几节的讨论中提到的一些概念你可能还没有学过。

你在本书的后面中还有机会深入学习 WebGL，在阅读本书时应该使用 WebGL Inspector。这有助于你更好地理解这个有用的工具。

1. Trace 面板

Trace 面板(见图 2-16)显示了当前捕获帧中所有记录下来的 WebGL 调用。可以逐个执行这些记录中的命令。当单步执行这些记录中的命令时，在 Trace 面板的右侧可以看到 WebGL 场景的创建过程。

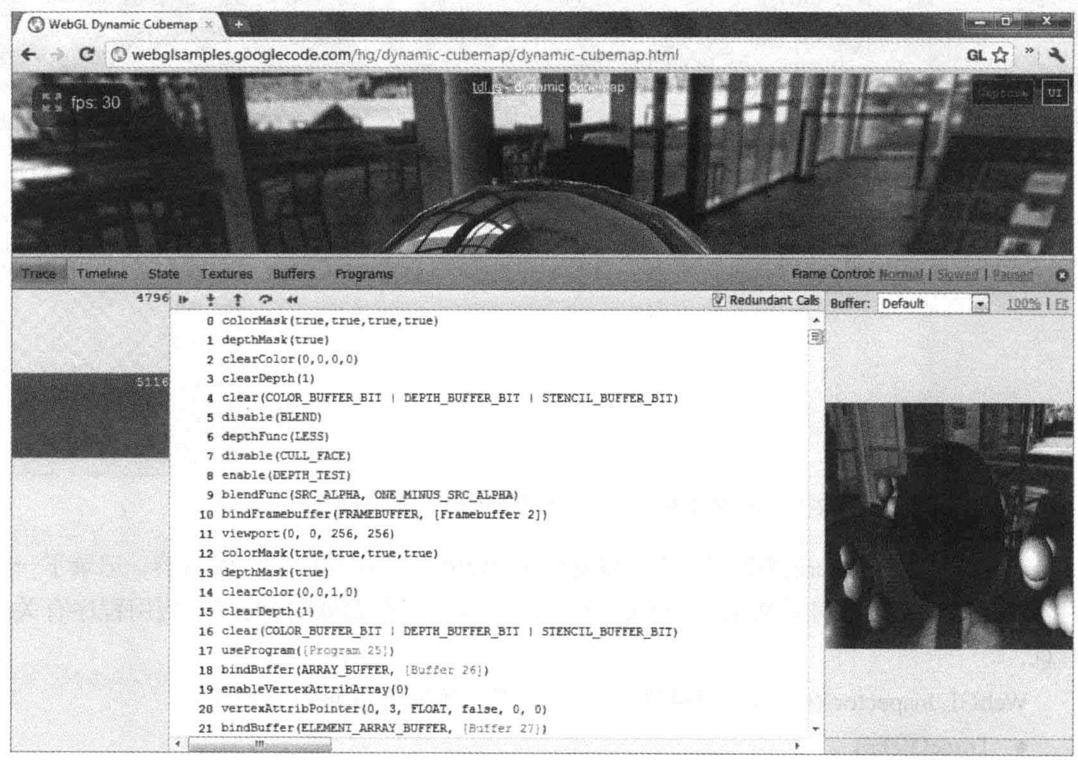


图 2-16　WebGL Inspector 的 Trace 面板

为了单步执行这些记录的命令，可以使用 Trace 面板顶部的小箭头按钮，或者使用功能快捷键。以下是这些功能快捷键及其作用：

- F8——单步执行(向前)
- F9——播放整个场景
- F6——单步后退一个调用

- F7——跳到下一个绘图调用
- F10——从每一帧的开始重新启动

当 WebGL 应用程序绑定了某个缓冲或者载入一个着色器程序后，就会出现 Trace 面板。当用户单击这个缓冲或着色器程序时，就会出现一个新的面板，它显示用户单击对象的更多信息。

Trace 面板的一个非常有用的功能是 WebGL Inspector 可以用黄色高亮显示冗余的调用。所谓冗余调用是指 WebGL API 的调用不会对 WebGL 的状态产生任何有意义的变化。例如，给统一变量赋一个与原本相同的值被认为是冗余调用，它就会在 Trace 面板中高亮显示出来。当用户需要优化 WebGL 应用程序的性能时，用户可以通过调试看看程序是否存在可以优化的冗余调用。

2. Timeline 面板

Timeline 面板显示有关 WebGL 应用程序的实时统计数据。图 2-17 是 Timeline 面板的外观。

图 2-17　WebGL Inspector 的 Timeline 面板

3. State 面板

State 面板显示 WebGL 应用程序的状态快照。图 2-18 是 State 面板的外观。

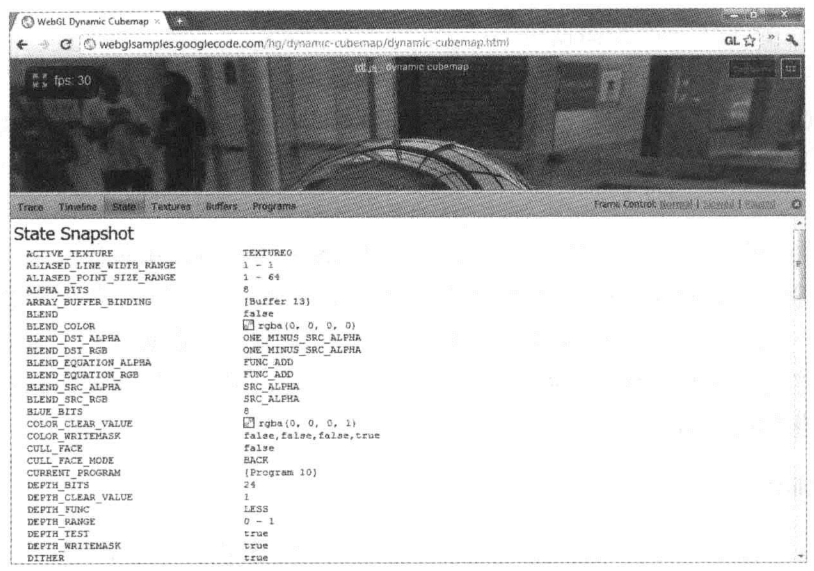


图 2-18 WebGL Inspector 的 State 面板

4. Textures 面板

Textures 面板(见图 2-19)提供许多有用的信息，这些信息都与 WebGL 应用程序中使用的纹理有关。在这个面板中，可以看到纹理图像，以及 2D 纹理和立方体贴图纹理的标识信息。这个面板也提供有关纹理的封装方式、纹理过滤以及从哪些 URL 可以载入纹理的信息。第 5 章将进一步讨论纹理。

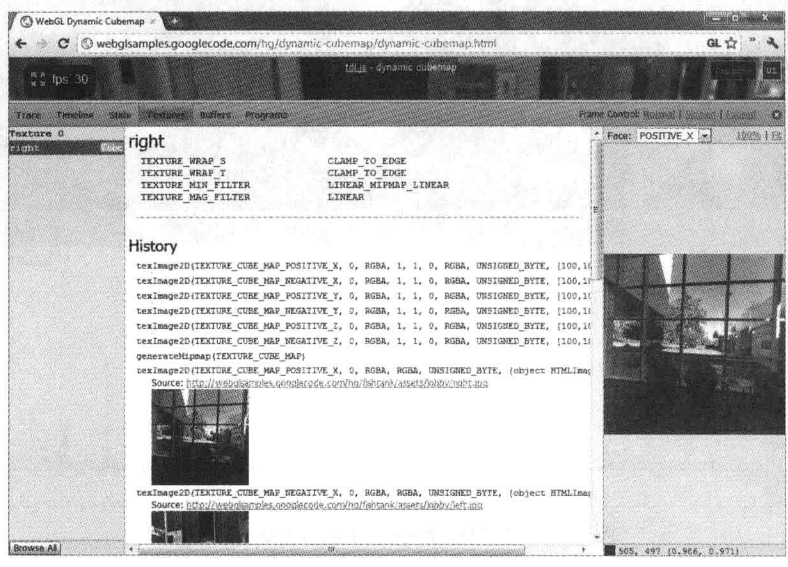


图 2-19 WebGL Inspector 的 Textures 面板

5. Buffers 面板

Buffers 面板显示 WebGL 应用程序中使用的各种缓冲的信息。面板的左侧显示 WebGL

程序中使用的各个缓冲的信息，包括缓冲的内容、缓冲的大小以及不同缓冲绑定的对象。面板的右侧显示选择的缓冲对象对应的几何对象。

在图 2-20 中，选择的缓冲有一个对应的球体几何对象。用户可以用鼠标旋转或缩放这个几何对象，对于比较复杂的几何对象，这个面板特别有用。

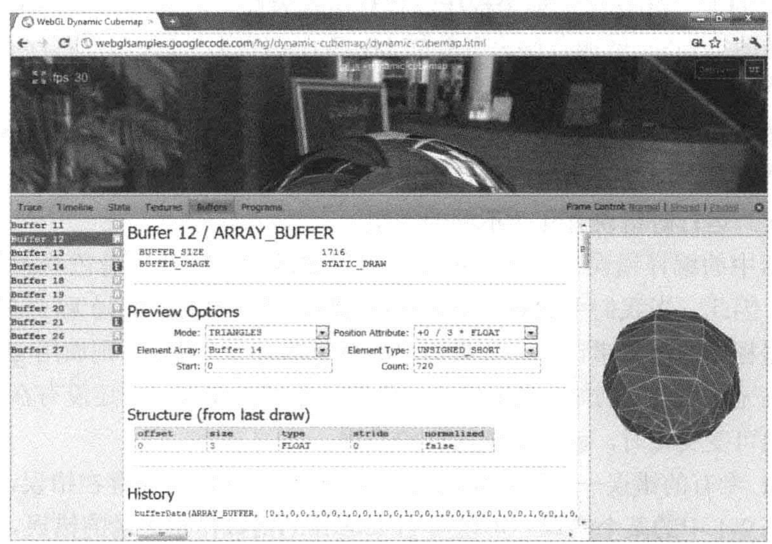


图 2-20　WebGL Inspector 的 Buffers 面板

6. Programs 面板

Programs 面板向用户提供当前 WebGL 应用程序使用的所有着色器程序信息。用户可以看到程序中用到的统一体变量和属性，也会看到顶点着色器和片段着色器的源代码。图 2-21 显示 Programs 面板的一部分。

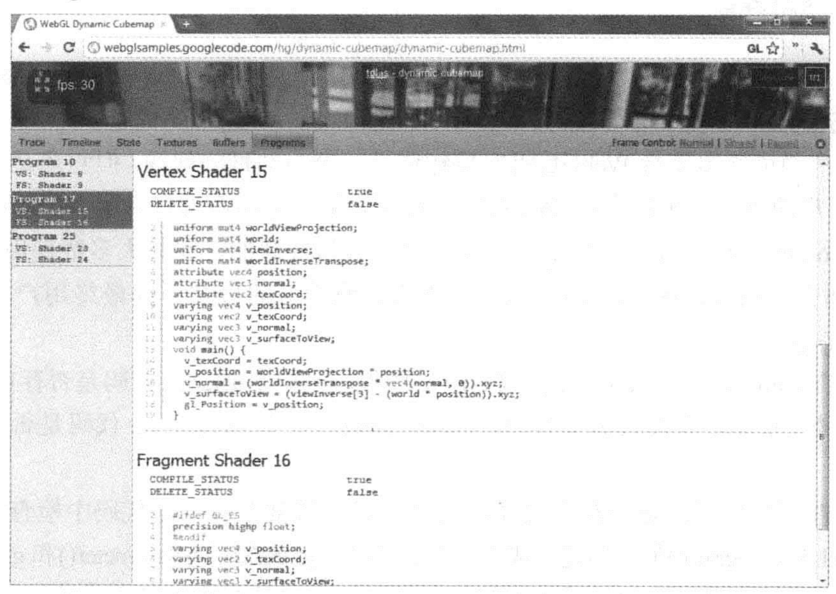


图 2-21　WebGL Inspector 的 Programs 面板

2.3.5 WebGL 的故障排除

WebGL 应用程序可能会出现许多错误。为了能够尽快而方便地找出并修正应用程序中的问题，了解这些问题对用户会有很大的帮助。下面是用户可能遇到的各种不同错误类型：

- JavaScript 语法错误——如果程序包含打印或类似的错误，则 JavaScript 引擎不会认为这样的 JavaScript 源代码为有效的 JavaScript 程序。如果应用程序存在语法错误，则这个应用程序根本无法执行。
- 运行时错误——程序存在一个错误，在程序开始运行之前，JavaScript 引擎无法发现这个错误。但是，当用户执行自己的 JavaScript 程序时，就会出现问题，触发一个错误。运行时错误有几个不同的类型。
- 着色器中的编译错误——用户的顶点着色器源代码或片段着色器源代码存在某个错误，因此，当我们用 gl.compileShader()编译着色器时，编译无法通过。
- 着色器程序的链接错误——如果用户的着色器有一个错误，则调用 gl.linkProgram()方法就无法通过。例如，当用户的片段着色器试图使用一个还没有在顶点着色器中定义的可变变量时，就会出现这样的错误。
- WebGL 专有的错误——用户调用某个 WebGL 方法时可能存在错误，因此 WebGL 生成表 2-1 中的某个错误。可以用 gl.getError()函数检索记录的错误。
- 逻辑错误——在 JavaScript 或 OpenGL ES 着色语言源代码中存在一个逻辑错误。用户的 WebGL 应用程序是由有效的源代码组成的，但是由于存在逻辑错误，应用程序没有按用户期望的方式执行。

现在你已经知道了当把自己的 WebGL 应用程序载入浏览器时可能出现的各种错误，另外，你也掌握了不同错误之间的区别。

故障排除检查表

因此，当用户把 WebGL 应用程序载入浏览器，但是没有得到所预期的结果时，应该怎么办？需要按以下顺序进行检查：

(1) 用户的浏览器支持 WebGL 吗？虽然这是一个很幼稚的问题，但它还是有可能会出现的。当用户的浏览器不能支持 WebGL 或者因为某些原因而没有启用 WebGL 时，调用 canvas.getContext()就会返回 null 值。通常这是很容易发现的，因为大多数使用 WebGL 的应用程序都会利用 JavaScript 的 alert()函数或其他类似的机制，向最终用户显示不支持 WebGL 的信息。

(2) 检查 Chrome 开发人员工具的 Console 面板，看看 JavaScript 代码是否存在语法错误。

(3) 检查 Chrome 开发人员工具的 Console 面板，看看 JavaScript 代码是否存在运行时错误。

(4) 检查着色器代码是否存在编译或链接错误。通常在着色器代码中检查 gl.compileShader()和 gl.linkProgram()的调用是否成功，方法是调用 gl.getShaderParameter()和 gl.getProgramParameter()。然后使用 alert()，或者用其他方法通知用户是否编译错误或链接错误。

(5) 检查任何 WebGL 调用是否成功，是否产生一个可以用 gl.getError()方法检索的错误。如果用户用 webgl-debug.js 等调试库并通过 console.log()方法记录下这些错误，则在 Console 面板中可以看到这些错误信息。先修正这些错误中的第一个，因为其他错误可能是第一个错误引起的。

(6) 检查 WebGL 虚拟照相机是否指向正确的方向。WebGL 没有内置支持照相机，但是你将在第 4 章中学习如何用变换操作模拟照相机(之所以在这里包括这些信息，是因此当你在 WebGL 应用程序中模拟一个照相机时，可以对照这里的错误处理方法)。

(7) 当用户已经检查并且修正以上所有错误，但是应用程序还是不能如所希望的那样运行时，则现在需要启动 Chrome 开发人员工具的 Scripts 面板对代码进行调试。有些错误用户只需要检查源代码就可以找出来，但是有些错误就很难找到，因此利用调试器可以比较容易找出这样的错误：

- 检查对象属性名是否有拼写错误。例如，如果用户给 WebGLBuffer 对象添加新的属性，用来保存缓冲中的 itemSize 和 numberOfItems(如程序清单 2-1 所示)，然后在 gl.drawArrays()或 gl.drawElements()调用中使用这些属性，那么必须保证不要写入属性 numberOfItems 并在随后读取属性 nbrOfItems。之所以仔细检查属性名，是因为 JavaScript 不允许读取未初始化的变量，但是允许读取一个对象的未初始化属性。这样的 bug 是很难发现的。
- 检查 WebGLRenderingContext 对象的全部属性是否都拼写正确。这与前一点非常相似。例如，在调用 gl.drawArrays()时，把 gl.TRIANGLE 而非 gl.TRIANGLES 作为参数传递给它。JavaScript 并不会发出警告信息，因为在 JavaScript 看来，程序只是读取对象的一个未初始化属性而已。

当然，在 WebGL 开发过程中，可能会遇到更多的错误。但是，按照这个顺序进行检查可以帮助用户较快地找到并改正一些难以发现的错误。

2.4　用 DOM API 载入着色器

在第一个 WebGL 示例中(见程序清单 2-1)，顶点着色器和片段着色器都以字符串的形式插入到 JavaScript 代码中，并用 JavaScript 的+运算符把字符串合并成着色器代码，如下所示：

```
var vertexShaderSource =
    "attribute vec3 aVertexPosition;              \n" +
    "void main() {                                \n" +
    "  gl_Position = vec4(aVertexPosition, 1.0); \n" +
    "}
```

在前面讨论程序清单 2-1 时曾指出，还有一个更简便的方法在 WebGL 应用程序中插入用户的着色器。下面是相同的顶点着色器源代码，但是它插入<script>标记中。这里并没有

使用 JavaScript 的+运算符合并字符串：

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  void main() {
    gl_Position = vec4(aVertexPosition, 1.0);
  }
</script>
```

对于这个小示例，你可能认为这两种方法没有多大区别。但是，当我们遇到一个源代码很长的大型着色器时，则使用第二种方法会更加简洁，而且可读性会更好。

此外，由于 WebGL 的着色器是用 OpenGL ES 着色语言编写的，因此 Internet 上和很多文献中有很多的着色器示例都是用 OpenGL ES 2.0 编写的，它们也可以用于 WebGL。也有一些着色器开发工具，如 AMD 公司的 RenderMonkey，它可以输出 OpenGL ES 着色语言。如果使用第二个方法，则很容易在自己的 WebGL 应用程序中，在<script>标记之间插入这些着色器。

你还需要记住一件事，在<script>标记之间编写的着色器源代码不是用 JavaScript 语言编写的。由于这里的源代码是用 OpenGL ES 着色语言编写的，因此不可以使用 JavaScript 语法，也不可以调用 JavaScript 方法。

当用户在<script>标记之间包含着色器源代码，而不是直接把它赋给一个 JavaScript 变量时，则需要用一种方法检索它，并把它合并到一个 JavaScript 字符串中，然后通过 WebGL API 传入。下面这段代码介绍现有的许多 WebGL 程序经常使用的一种模式。很难确定这种想法最初来自哪里，但是，来自 Mozilla 公司的早期 WebGL 示例使用了这种模式。

与以下源代码稍微不同的一个版本出现在 http://learningwebgl.com/cookbook/index.php/Loading_shaders_from_HTML_script_tags 中，它是由 Valdimir Vukićević 提供的。

```
function loadShaderFromDOM(id) {
  var shaderScript = document.getElementById(id);

  // If we don't find an element with the specified id
  // we do an early exit
  if (!shaderScript) {
    return null;
  }

  // Loop through the children for the found DOM element and
  // build up the shader source code as a string
  var shaderSource = "";
  var currentChild = shaderScript.firstChild;
  while (currentChild) {
```

```
      if (currentChild.nodeType == 3) { // 3 corresponds to TEXT_NODE
        shaderSource += currentChild.textContent;
      }
      currentChild = currentChild.nextSibling;
    }

    var shader;
    if (shaderScript.type == "x-shader/x-fragment") {
      shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (shaderScript.type == "x-shader/x-vertex") {
      shader = gl.createShader(gl.VERTEX_SHADER);
    } else {
      return null;
    }

    gl.shaderSource(shader, shaderSource);
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert(gl.getShaderInfoLog(shader));
      return null;
    }
    return shader;
  }
```

函数 loadShaderFromDOM()需要接收一个 id 属性参数，它利用这个 id 属性在 DOM 树中找到相应的元素。例如，如果我们给插入着色器源代码的<script>标记设置 id 为 shader-vs，则需要把 shader-vs 作为参数传给 loadShaderFromDOM()函数。

loadShaderFromDOM()使用 DOM API 函数并根据传入的 id 找到相应的元素，然后遍历这个元素的全部子元素，生成一个表示源代码的文本字符串。

最后，这个函数检查所找到元素的类型属性，并根据类型创建一个顶点着色器或片段着色器。它把着色器的源代码载入创建的着色器对象中，并对它进行编译。如果编译成功，把这个着色器对象返回给调用这个函数的程序。

2.5 更高级的综合示例

至此，你已经知道了如何利用 DOM API 的 document.getElement()方法载入自己的着色器，也知道了如何调试自己的 WebGL 应用程序，如何排除应用程序中的错误。现在来看看一个完整的示例，它应用了这里介绍的新知识。

程序清单 2-2 是以本章前面的程序清单 2-1 为基础的，但是它利用 DOM API 载入顶点着色器和片段着色器。这些着色器包括在页面的<header>元素的<script>标记中。此外，本例使用了本章前面介绍的 webgl-debug.js 调试库。

如果你已经仔细阅读本章前面的内容，则这里的源代码应该不会陌生，应该说很容易理

解。如果你注意到代码中的高亮部分，就知道它们是顶点着色器和片段着色器。在两个着色器之后，插入 webgl-debug.js 调试库，它是一个外部 JavaScript 文件。函数 createGLContext()与程序清单 2-1 中没有任何变化。

下一个高亮的代码段包含 loadShaderFromDOM()函数，你应该能够识别它。在 setupShaders()函数中，现在没有了包含着色器源代码的 JavaScript 字符串。现在这个函数调用 loadShaderFromDOM()函数，创建一个顶点着色器对象；再调用一次 loadShaderFromDOM()函数，创建一个片段着色器。最后是 startup()函数，我们看到，为了使用调试库，在 WebGLDebugUtils.makeDebugContext()函数中调用 createGLContext()方法。

可从
Wrox.com
下载源代码

程序清单 2-2　一个用 DOM API 载入着色器并包含 webgl-debug.js 调试库的完整示例

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Listing 2-2, Load Shaders From DOM</title>
<meta charset="utf-8">
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;

  void main() {
    gl_Position = vec4(aVertexPosition, 1.0);
  }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump fl oat;

  void main() {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
  }
</script>

<script src="webgl-debug.js"></script>
<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var vertexBuffer;

function createGLContext(canvas) {
  var names = ["webgl", "experimental-webgl"];
  var context = null;
  for (var i=0; i < names.length; i++) {
    try {
      context = canvas.getContext(names[i]);
    } catch(e) {}
    if (context) {
```

```
        break;
      }
    }
    if (context) {
      context.viewportWidth = canvas.width;
      context.viewportHeight = canvas.height;
    } else {
      alert("Failed to create WebGL context!");
    }
    return context;
}

function loadShaderFromDOM(id) {
    var shaderScript = document.getElementById(id);

    // If we don't fi nd an element with the specifi ed id
    // we do an early exit
    if (!shaderScript) {
      return null;
    }

    // Loop through the children for the found DOM element and
    // build up the shader source code as a string
    var shaderSource = "";
    var currentChild = shaderScript.fi rstChild;
    while (currentChild) {
      if (currentChild.nodeType == 3) { // 3 corresponds to TEXT_NODE

        shaderSource += currentChild.textContent;
      }
      currentChild = currentChild.nextSibling;
    }

    var shader;
    if (shaderScript.type == "x-shader/x-fragment") {
      shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (shaderScript.type == "x-shader/x-vertex") {
      shader = gl.createShader(gl.VERTEX_SHADER);
    } else {
      return null;
    }

    gl.shaderSource(shader, shaderSource);
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert(gl.getShaderInfoLog(shader));
      return null;
    }
    return shader;
```

```
}

function setupShaders() {
  vertexShader = loadShaderFromDOM("shader-vs");
  fragmentShader = loadShaderFromDOM("shader-fs");

  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);

  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Failed to setup shaders");
  }

  gl.useProgram(shaderProgram);

  shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
  "aVertexPosition");
  }

function setupBuffers() {
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  var triangleVertices = [
         0.0,  0.5,  0.0,
        -0.5, -0.5,  0.0,
         0.5, -0.5,  0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
  gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  vertexBuffer.numberOfItems = 3;
}

function draw() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT);

  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                         vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

  gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
}

function startup() {
  canvas = document.getElementById("myGLCanvas");
  gl = WebGLDebugUtils.makeDebugContext(createGLContext(canvas));
  setupShaders();
```

```
    setupBuffers();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    draw();
}
</script>

</head>

<body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>

</html>
```

实验代码

如果你还没有修改过自己的代码，并且还没有实验这个程序的功能，则现在就应该执行这些操作。修改自己的代码可以帮助你更好地理解程序，也帮助你记住本章介绍的概念。以下是几点与修改代码有关的建议。如果你掌握了本章的内容，则应该知道如何修改这个程序：

- 把三角形的颜色从白色改为黑色。
- 把背景的颜色从黑色改为绿色。
- 改变三角形顶点的 z 坐标，使得三角形从屏幕上消失。三角形为什么会消失？
- 不改变顶点数据，如何使三角形看起来比较小(提示：试试视口)。
- 在内部修改代码，插入以下错误，用 Chrome 开发人员工具或 Firebug 分析这些错误：
    - JavaScript 语法错误。
    - JavaScript 运行时错误。
    - WebGL 的 gl.INVALID_OPERATION 错误。

2.6　小结

本章分析了一个基本的 WebGL 示例，它在屏幕上绘制一个 2D 三角形。本章从头到尾完整地介绍了这个程序的所有源代码。通过这个示例，你掌握了创建一个 WebGL 应用程序所需的全部基本步骤。

本章也介绍了 WebGL 应用程序的调试和故障排除方法。介绍了 Chrome 开发人员工具以及它对于 WebGL 的作用。也简单介绍了 Firebug 的用法，并且介绍了这两个工具在许多方面存在的相似性。此外，本章还介绍了 WebGL Inspector 的用法和它在调试 WebGL 应用程序时的作用。

最后，本章还介绍了如何在<script>标记中插入顶点着色器和片段着色器，并利用 DOM API 检索着色器。利用这个技术，我们更容易重用以 OpenGL ES 2.0 编写或生成的代码。

第 3 章

绘　图

本章主要内容

- 如何用 WebGL 绘制三角形、线和点精灵
- 掌握不同图元的绘制方法
- 认识三角形组绕顺序的重要性
- 如何使用 WebGL 提供的绘图方法
- 知道什么是类型化数组以及它们在 WebGL 中的使用方式
- 了解用户可以使用的绘图方法以及如何得到最优性能
- 如何在同一个数组中交叉使用不同类型的顶点数据

在第 2 章中，我们学习了如何使用 WebGL 绘制一个简单的三角形，还使用了 gl.drawArrays()，并把 gl.TRIANGLES 作为它的第一个参数，以告诉 WebGL 用这些顶点定义一个三角形。正如你所料，这不是 WebGL 中的唯一绘图方法。本章将向你介绍使用 WebGL 绘制图形的其他可用方法。

由于 WebGL 中的绘制是以定义在 WebGLBuffer 对象中的顶点数据为基础的，因此我们将深入分析在创建和组织顶点数据时可以使用的各种不同选项。本章将以一个示例作为总结，其中将使用一些您学到的新知识。

3.1　使用 WebGL 绘制图元和绘图方法

gl.drawArrays()方法是绘制图元的两个可用方法之一。另一个方法是 gl.drawElements()。通过本章的学习，你对这两个方法的用法将有更深的理解。但是，首先我们来讨论用顶点可以定义哪些图元。下面几节介绍的图元是可以作为 gl.drawArrays()或 gl.drawElements()方法的第一个参数传入的图形对象。

3.1.1 图元

可以用 WebGL 建立复杂的 3D 模型，但是，这些 3D 模型都是由以下的 3 种基本几何图元构建的：

- 三角形
- 线
- 点精灵

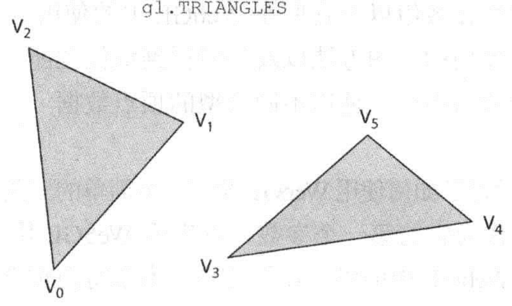
以下几节将详细介绍这些图元。

1. 三角形

在第 1 章中，你已经知道了点是用来构建其他几何对象的最基本构建块。从数学角度来看也确实如此，但是从 3D 图形硬件的角度来看，实际上三角形才是最基本的构建块。一个 3D 的点当然很重要，因为定义三角形需要 3 个顶点。但是当今大多数 3D 图形硬件经过高度优化处理，可以快速绘制三角形。WebGL 中可以使用的各个不同三角形图元是 WebGLRenderingContext 的一部分：

- gl.TRIANGLES
- gl.TRIANGLE_STRIP
- gl.TRIANGLE_FAN

图 3-1 说明了这些图元可能的形状。这些三角形图元的详细信息将在以下几节中介绍。



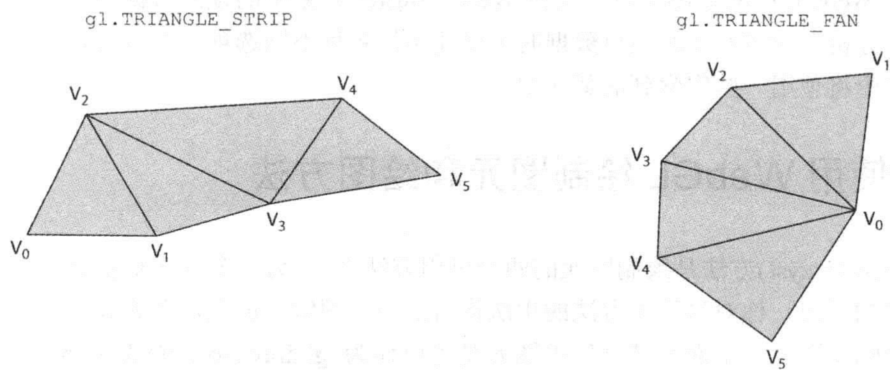


图 3-1 WebGL 中的各种三角形图元

独立的三角形

gl.TRIANGLES 是最基本的三角形图元。使用 gl.drawArrays()函数时，必须为每个三角形定义 3 个顶点，利用它们可以绘制独立的三角形(即一个三角形的顶点不会被另一个三角形重用)。

在图 3-1 的顶部，用顶点(V_0,V_1,V_2)定义一个三角形，用顶点(V_3,V_4,V_5)定义另一个三角形。以下公式决定所绘制三角形的个数：

$$绘制的三角形数量 = \frac{count}{3}$$

这里的 count 表示使用 gl.drawArrays()或 gl.drawElements()函数时分别传入的顶点数或索引数。

三角形带

对于许多几何对象，我们需要绘制几个连接在一起的三角形。gl.TRIANGLE_STRIP 绘制由连接的三角形构成的带状图形。如果有很多相互连接的三角形，则使用三角形带图元相比于使用独立三角形图元的优点是可重用三角形之间的顶点。因此在构建几何模型的三角形时，可以定义较少的顶点。较少的顶点意味着需要处理的数据量较少，这也意味着从内存传送到 GPU 的数据量也较少。

在图 3-1 的左下角中，三角形带的第一个三角形由顶点(V_0,V_1,V_2)定义，下一个顶点是 V_3，它与前两个顶点一起定义第二个三角形，因此第二个三角形由顶点(V_2,V_1,V_3)定义，注意这个三角形的顶点顺序。对于三角形带(对于后面马上就要介绍的三角扇也是如此)，组成带的其余三角形的顶点顺序必须遵循第一个三角形的顶点顺序。

由于第一个三角形的顶点要按逆时针顺序定义，第二个、第三个三角形也必须按逆时针顺序定义。这意味着，第二个三角形的顶点顺序必须是(V_2,V_1,V_3)，第三个三角形的顺序为(V_2,V_3,V_4)，第四个三角形为(V_4,V_3,V_5)。

所有三角形带上都遵循这种模式。在第一个三角形之后，每个新顶点会与前面的两个顶点构成一个新的三角形。对于这个新的三角形，需要添加一个顶点，移除另外一个顶点(相对于前一个三角形而言)。对于每一个新的三角形，保留前一个三角形的后两个顶点。

为了更容易看出三角形顶点的组合模式，用表列出所有的三角形和它们对应的顶点。表 3-1 显示三角形带的每个三角形和它们的顶点。在“顶点 3”这一列中，我们很容易看出，为每个三角形添加一个新顶点。同时也能看出，对于第二个和第四个三角形，第一个顶点和第二个顶点的顺序颠倒——即对于第二个三角形，V_2 在 V_1 之前；对于第四个三角形，V4 在 V_3 之前。

表 3-1 三角形带的顶点组合模式

三角形	顶点 1	顶点 2	顶点 3
1	V_0	V_1	V_2
2	V_2	V_1	V_3

(续表)

三角形	顶点 1	顶点 2	顶点 3
3	V_2	V_3	V_4
4	V_4	V_3	V_5

gl.drawArrays()方法用 gl.TRIANGLE_STRIP 参数绘制的三角形数等于定义的顶点个数减 2(如果使用 gl.drawElements()方法，则绘制的三角形数等于索引数减 2)。这样我们得到一个简单的公式：

绘制的三角形数量＝count – 2

这里的 count 是使用 gl.drawArrays()时的顶点数，或者是使用 gl.drawElements()时的索引数。

三角扇

绘制几个相互连接的三角形的另一个方法是用 gl.TRIANGLE_FAN 绘制三角扇。

对于一个三角扇，前 3 个顶点构成第一个三角形，其中第一个顶点用做扇形的原点，在第一个三角形之后，新增的一个顶点与前一个顶点和原点构成一个新的三角形。

在图 3-1 的右下角中可以看出这种模式。顶点(V_0,V_1,V_2)构成第一个三角形，顶点(V_0,V_2,V_3)构成第二个三角形，顶点(V_0,V_3,V_4)构成第三个三角形，顶点(V_0,V_4,V_5)构成第四个三角形。

按照这种模式生成相同的三角形，则三角扇比起独立的三角形需要较少的顶点(对于 gl.drawElements()则是较少的索引)。使用 gl.TRIANGLE_FAN 生成的三角形数量由以下公式得到：

绘制的三角形数量=count – 2

这里的 count 是传入 gl.drawArrays()方法的顶点数或传入 gl.drawElements()方法的索引数。

2. 线

虽然 3D 图形主要与三角形绘制有关，但是有时也需要绘制线。下面是我们在 WebGL 中经常使用的 3 种不同线图元：

- gl.LINES
- gl.LINE_STRIP
- gl.LINE_LOOP

图 3-2 说明了 3 个不同类型的线图元，下面几节将详细介绍每个这些图元。

当我们用前一节介绍的三角形图元之一绘制三角形时，默认得到一个填充的三角形。如果由于某些原因而想绘制一个没有填充效果的几何对象(有时称它为线框模型)，则最好使用本节介绍的线图元之一。

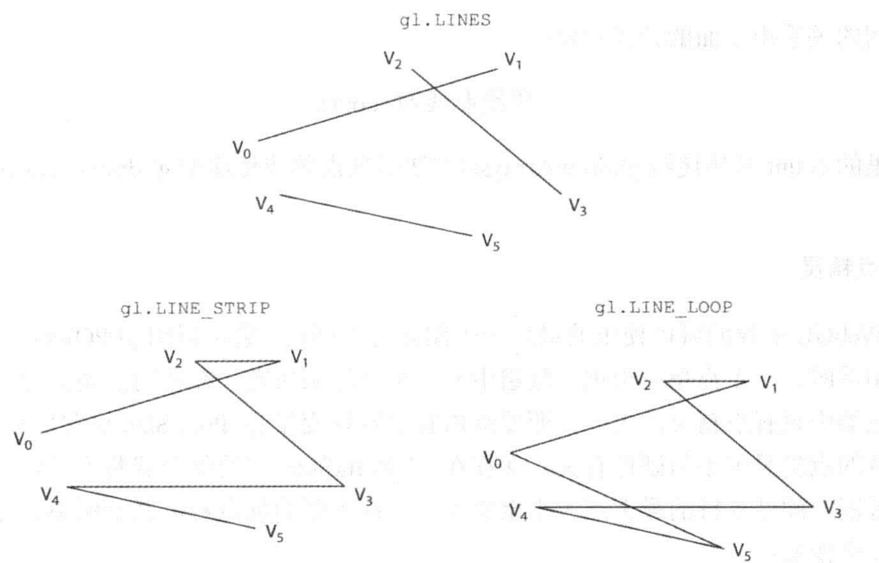


图 3-2 WebGL 中的各种线图元

独立线

用 gl.drawArrays()绘制的独立三角形不会重用两个三角形的任何顶点，同样的道理，独立线也不会重用线的任何顶点。用 gl.LINES 绘制独立线段。图 3-2 的顶部是用(V_0,V_1)、(V_2,V_3)、(V_4,V_5)绘制得到的 3 条独立的线。

每个线需要两个顶点，因此所绘制线数由以下公式给出：

$$所绘制线数 = count/2$$

这里的 count 是传递给 gl.drawArrays()方法的顶点数或传递给 gl.drawElements()方法的索引数。

线带

如果我们想同时绘制几条头尾相连的线，则可以使用 gl.LINE_STRIP 绘制线带。在图 3-2 的左下角，在(V_0, V_1)、(V_1, V_2)、(V_2, V_3)、(V_3, V_4)顶点之间各绘制一条线。所绘制线数由以下公式决定：

$$所绘制线数 = count - 1$$

这里的 count 是传递给 gl.drawArrays()方法的顶点数或传递给 gl.drawElements()方法的索引数。

线环

线环是用 gl.LINE_LOOP 绘制得到的，它的绘制方式与线带非常相似。唯一差别是在线环中，有一条线从最后一个顶点到指定的第一个顶点。在图 3-2 的右下角，在(V_0, V_1)、(V_1, V_2)、(V_2, V_3)、(V_3, V_4)、(V_4, V_0)顶点之间各绘制一条线。对于线环，所绘制线数与顶

点数之间的关系由下面的公式决定：

所绘制线数=count

这里的 count 是传递给 gl.drawArrays()方法的顶点数或传递给 gl.drawElements()方法的索引数。

3. 点精灵

在 WebGL 中我们可以使用的最后一个图元是点精灵，它可以用 gl.POINTS 绘制得到。绘制点精灵时，一个点精灵由顶点数组中的一个坐标来决定。在使用点精灵时，还需要在顶点着色器中设置点精灵的大小，即要给内置的特殊变量 gl_PointSize 设置像素大小。

支持的点精灵大小与硬件有关，保存在 gl_PointSize 中的值会截断为硬件支持的某个点大小范围。硬件支持的最大点大小至少为 1。在下面的顶点着色器示例中，点大小设置为直径 5 个像素：

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPos;

  void main() {
    gl_Position = vec4(aVertexPos, 1.0);

    gl_PointSize = 5.0;
  }

</script>
```

在 WebGL 中，点精灵经常用来呈现粒子效果。粒子效果常用来实时模拟真实的自然现象，这些现象包括爆炸、大火、烟雾、灰尘等。从这些示例可以看出，粒子效果常用在 3D 图形游戏中。本书不打算介绍粒子效果，但是如果你对此有兴趣，在 Web 上搜索可以找到有关这方面的很多有趣的材料，你可以从这些材料开始学习。

3.1.2 顶点组绕顺序的重要性

在 WebGL 中，三角形的一个重要属性是顶点组绕顺序。三角形的顶点组绕顺序是逆时针(CounterClockWise，CCW)或者顺时针(ClockWise，CW)。当三角形按顶点的逆时针顺序构建时，我们称它的组绕顺序为逆时针，如图 3-3 左图所示。当三角形按顶点的顺时针顺序构建时，我们称它的组绕顺序为顺时针，如图 3-3 右图所示。

组绕顺序之所以很重要，是因为它决定了三角形的面是否朝向观察者。朝向观察者的三角形为正面三角形，否则为背面三角形。在许多情形中，WebGL 不需要对背面三角形进行光栅化处理。例如，如果在某个场景中观察者看不见一些对象的背面，则可以指示 WebGL 剔除这些无法看到的面。很容易调用下面的 3 个方法来实现这一点：

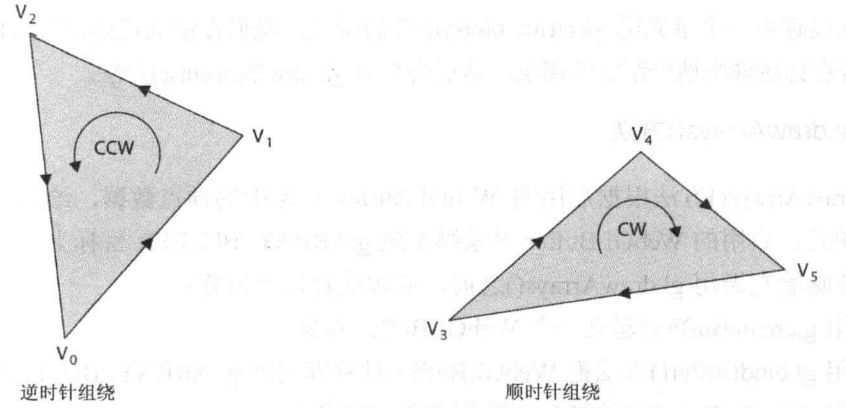


图 3-3　三角形的组绕顺序是逆时针(CCW)或者顺时针(CW)

```
gl.frontFace(gl.CCW);
gl.enable(gl.CULL_FACE);
gl.cullFace(gl.BACK);
```

第一个方法告诉 WebGL 采用逆时针组绕的三角形是正面三角形。即使不调用 gl.frontFace()方法，WebGL 默认也会采用这种处理方式。第二个方法激活面剔除功能，默认情况下这个功能处于禁用状态，因此需要调用 gl.enable(gl.CULL_FACE)方法启动它。第三个方法告诉 WebGL 剔除背面三角形。即使没有调用 gl.cullFace()方法，背面剔除也是 WebGL 的默认处理方式。

这些方法有不同的使用方式。例如，可以用下面的命令告诉 WebGL，使用顺时针组绕的三角形是正面三角形：

```
gl.frontFace(gl.CW);
```

此外，用下面的代码指示 WebGL 剔除正面三角形：

```
gl.cullFace(gl.FRONT);
```

如果一个场景由许多对象组成，它们的背面对用户是不可见的，则最好激活背面剔除功能。这可以改善 WebGL 应用程序的性能，因为 GPU 不需要对不可见的三角形进行光栅化处理。

3.1.3　WebGL 的绘图方法

在 WebGL 中，有 3 个方法可以用来更新绘图缓冲：

- gl.drawArrays()
- gl.drawElements()
- gl.clear()

但是在绘制几何对象时，必须使用前两个方法中的一个。第三个方法 gl.clear()只是用来把

全部像素设置为一个事先用 gl.clearColor()定义的颜色。我们在前面已经用过 gl.drawArrays()方法，现在比较详细地介绍它的用法。然后会介绍 gl.drawElements()方法。

1. gl.drawArrays()方法

gl.drawArrays()方法根据启用的 WebGLBuffer 对象中的顶点数据，绘制由第一个参数定义的图元。启用的 WebGLBuffer 对象绑定到 g.ARRAY_BUFFER 目标上。

这意味着在调用 gl.drawArrays()之前，必须执行以下操作：

- 用 g.createBuffer()建立一个 WebGLBuffer 对象。
- 用 gl.bindBuffer()方法把 WebGLBuffer 对象绑定到 gl.ARRAY_BUFFER 目标。
- 用 gl.bufferData()方法把顶点数据载入到缓冲中。
- 用 gl.enableVertexAttribArray()方法激活通用顶点属性。
- 调用 gl.vertexAttribPointer()方法把顶点着色器的属性连接到 WebGLBuffer 对象中的正确数据。

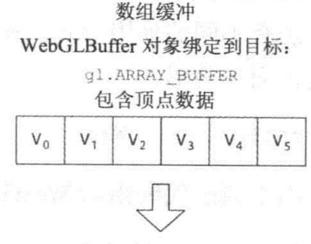
gl.drawArrays()方法的原型如下：

```
void drawArrays(GLenum mode, Glint first, GLsizei count);
```

下面介绍这些参数的意义：

- mode 定义了所要渲染的图元的类型。它可以取以下值之一：
  - gl.POINTS
  - gl.LINES
  - gl.LINE_LOOP
  - gl.LINE_STRIP
  - gl.TRIANGLE
  - gl.TRIANGLE_STRIP
  - gl.TRIANGLE_FAN
- first 参数定义顶点数据数组中的哪个索引用作第一个索引。
- count 定义了需要使用的顶点数。

归纳起来，这个方法的 mode 参数定义了要绘制的图元类型，count 参数定义连续顶点的个数，用 first 参数定义了第一个顶点在数组中的索引位置。图 3-4 说明了 gl.drawArrays()方法工作方式的概念视图。



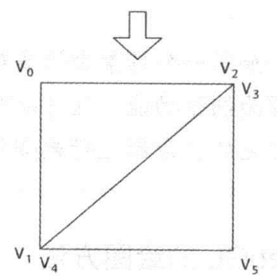


图 3-4 gl.drawArrays()方法工作方式的概念视图

gl.drawArrays()方法的设计要求表示图元的顶点必须按正确的顺序进行绘制。如果顶点之间不存在共享，则使用这种方法既简单又快速。但是，如果有一个对象的表面由一个三角形网格组成，而且每个顶点都由此网格上的多个三角形共享，则使用 gl.drawElements()方法可能会更好。

2. gl.drawElements()方法

你现在已经知道，当存在多个共享顶点时，使用 gl.TRIANGLE_STRIP 和 gl.TRIANGLE_FAN 图元比起使用 gl.TRIANGLE 图元可以增加顶点的重用程度。

gl.drawElements()方法有时也称为索引绘图，它可以进一步提高顶点的重用程度。从前面可知，gl.drawArrays()直接使用一个或多个数组缓冲(即绑定到目标 gl.ARRAY_BUFFER 的 WebGLBuffer 对象)，这些数组缓冲以正确的顺序包含顶点数据。gl.drawElements()方法也利用包含顶点数据的数组缓冲，但是它还使用一个元素数组缓冲(即绑定到 gl.ELEMENT_ARRAY_BUFFET 目标上的 WebGLBuffer 对象)。这个元素数组缓冲包含了带有顶点数据的数组缓冲的索引。

这意味着，顶点数据在数组缓冲中可以是任何顺序，因为在元素数组缓冲对象中的索引决定 gl.drawElements()方法使用的顶点的顺序(但是，你在本章后面将会看到，从程序的性能来看，应尽可能按顺序读取顶点数据)。此外，如果要实现顶点共享，则这种方法比较容易实现，即只要把元素数组缓冲中的某些项指向数组缓冲中的同一个索引。图 3-5 说明了 gl.drawElements()工作方式的概览。

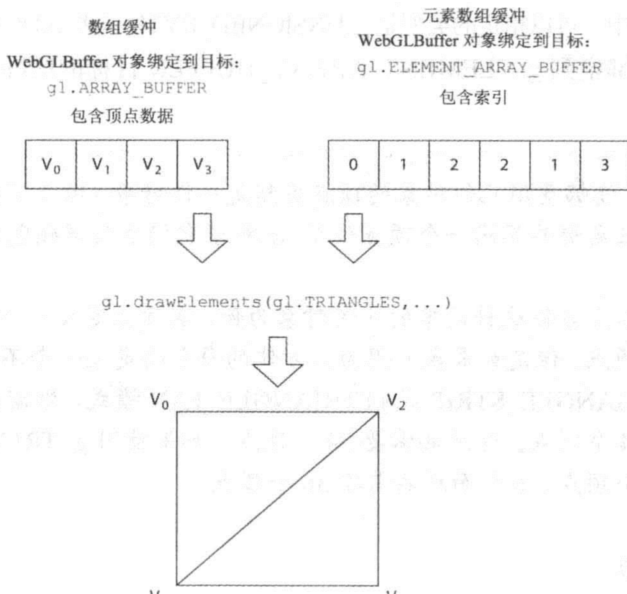


图 3-5 gl.drawElements()工作方式的概念视图

在调用 gl.drawElements()方法之前，也要像本章前面介绍过的 gl.drawArrays()那样设置数组缓冲。此外，还需要设置元素数组缓冲，具体过程如下：

- 用 gl.createBuffer()创建一个 WebGLBuffer 对象。
- 用 gl.bindBuffer()方法把这个 WebGLBuffer 对象绑定到 gl.ELEMENT_ARRAY_ BUFFER 目标上。

用 gl.bufferData()把决定顶点数据使用顺序的索引载入到此缓冲中。

gl.drawElements()方法的原型如下：

```
void drawElements(GLenum mode, GLsizei count, GLenum type,GLintptr offset);
```

该方法的各个参数意义如下：

- mode 定义了要渲染的图元的类型。与 gl.drawArrays()方法的 mode 参数一样，它可以取以下值：
  - gl.POINTS
  - gl.LINES
  - gl.LINE_LOOP
  - gl.LINE_STRIP
  - gl.TRIANGLE
  - gl.TRIANGLE_STRIP
  - gl.TRIANGLE_FAN
- count 定义了绑定到 gl.ELEMENT_ARRAY_BUFFER 目标上的缓冲中的索引数。
- type 定义了元素索引的类型，元素索引存储在绑定到 gl.ELEMENT_ARRAY_BUFFER 目标上的缓冲中。可以指定的类型是 gl.UNSIGNED_BYTE 或者 gl.UNSIGNED_SHORT。
- offset 定义绑定到 gl.ELEMENT_ARRAY_BUFFER 目标的缓冲中的偏移量，索引从此处开始。

通常，能够重用几何对象的顶点当然是一件好事，但是不能因为几何模型中的一些三角形共享同一个顶点位置，就希望它们也共享颜色或法线等其他顶点数据。

我们以立方体这种简单的几何对象为例，实际上定义一个立方体只需要 8 个不同的顶点。但是如果我们想为立方体的每个面定义一个不同的颜色，并且使用 gl.TRIANGLE_STRIP 或 gl.TRIANGLE_FAN 模式，则需要为立方体的每个面定义 4 个顶点。这总共需要 24 个顶点。如果使用 gl.TRIANGLE，则每个面需要 6 个顶点，6 个面总共需要 36 个顶点。

3. 退化三角形

从性能的角度来看，gl.drawArrays()或 gl.drawElements()函数的调用次数越少越好。例如，如果一个顶点数组包含 200 个三角形，则一次调用 gl.drawArrays()或 gl.drawElements()比 100 次调用绘图函数且每次绘制两个三角形的效率要高许多。

如果使用独立三角形图元(即 gl.TRIANGLES)，这很容易实现。但是，如果使用 gl.TRIANGLE_STRIP 图元，则当三角形带之间存在不连续性时，就不那么容易组合不同的三角形带。

这种不连续性或在两个三角形带间存在跳转的解决方法是插入额外的索引(如果使用 gl.drawArrays()函数，则插入额外的顶点)，这样就得到退化三角形。退化三角形是指三角

形至少有两个索引(或顶点)是相同的。因此，存在面积为 0 的三角形。这样的三角形很容易被 GPU 检测并删除。

在 gl.drawElements()和 gl.drawArrays()方法中都可以使用退化三角形。但是在 gl.drawArrays()方法中使用退化三角形需要复制顶点数据。无论从内存或性能的角度来看，其成本都比在元素数组缓冲中复制索引要大许多。

这是因为，顶点数据比元素索引要占用更多的内存。此外，复制顶点的想法很不好，这是因为在 GPU 中，变换后的顶点缓存通常只会缓存顶点索引。这意味着，每次出现在三角形带中时，复制的顶点都需要经过顶点着色器的变换处理。

连接两个三角形带需要增加额外的索引，额外索引的数量取决于第一个三角形带所使用的索引数。这是因为，三角形的组绕顺序起决定作用。假设我们想使第一个和第二个三角形带都使用相同的组绕顺序，则需要考虑两种情况：

- 如果第一个三角形带包含偶数个三角形，则为了连接第二个三角形带，需要增加两个额外的索引。
- 如果第一个三角形带包含奇数个三角形，则为连接第二个三角形带，则为了保证组绕顺序不变而需要增加 3 个额外的索引。

图 3-6 是第一个情形的一个示例(即第一个三角形带包含偶数个三角形)。第一个三角形带由(V_0,V_1,V_2)和(V_2,V_1,V_3)这两个三角形组成，它们对应于元素数组缓冲中的元素索引(0,1,2,3)。第二个三角形带由(V_4,V_5,V_6)和(V_6,V_5,V_7)这两个三角形组成，它们对应于元素数组缓冲中的元素索引(4,5,6,7)。

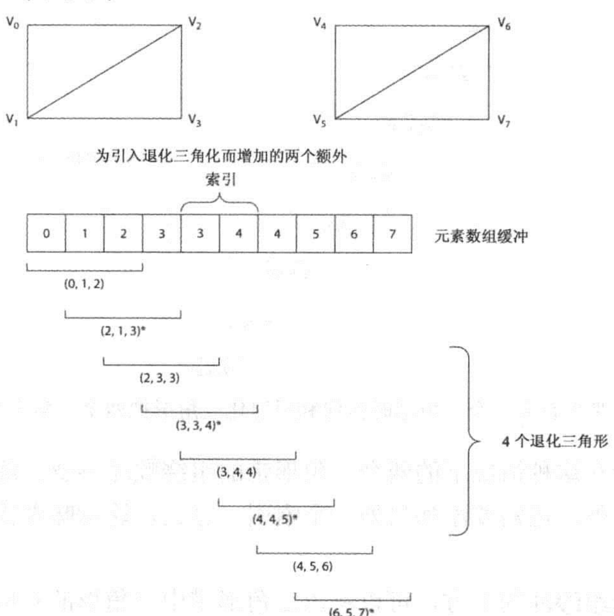


图 3-6 第一个三角形带含有偶数个三角形时如何利用退化三角形把两个三角形带连接在一起的图示

为了连接这两个三角形带并建立所需要的退化三角形，需要在这两个三角形带之间添加两个额外的索引。这两个索引就是图 3-6 中所示的索引 3 和 4。这里使用的规则是第一个三角形带的最后一个顶点和最后一个三角形带的第一个顶点重复。

在图 3-6 中元素数组缓冲的下方，可以看出缓冲中的索引如何构成三角形带的三角形。每个第二个三角形的括号后面都有一个星号(*)，表示此三角形是由调换前两个顶点的顺序得到的。这正是我们在本章前面有关三角形带的一节中学到的内容。

首先是来自第一个三角形带的两个三角形，它们的索引分别为(0,1,2)和(2,1,3)。然后是 4 个退化三角形，它们将会被 GPU 丢弃。最后是来自后一个三角形带的两个三角形，它们的索引分别为(4,5,6)和(6,5,7)。

图 3-7 说明了第一个三角形带由奇数个三角形组成的情况。在这个示例中，第一个三角形带由(V_0,V_1,V_2)、(V_2,V_1,V_3)、(V_2,V_3,V_4)这 3 个三角形组成，它们与元素数组缓冲中的元素索引(0,1,2,3,4)相对应。第二个三角形带由(V_5,V_6,V_7)、(V_7,V_6,V_8)这两个三角形组成，它们与元素数组缓存中最后位置的元素索引(5,6,7,8)相对应。

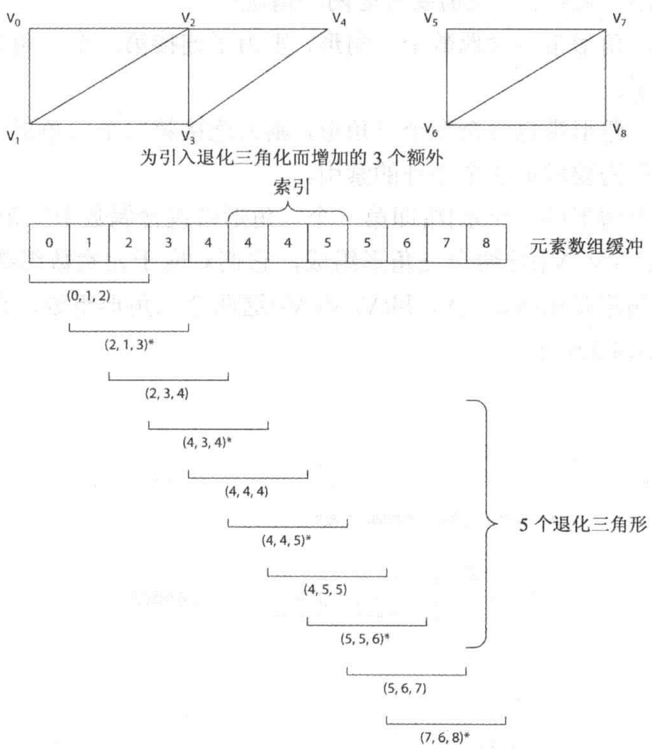


图 3-7 第一个三角形带含有奇数个三角形时如何利用退化三角形把两个三角形带连接在一起的图示

如果我们想保持在这种情况下的两个三角形带的组绕顺序不变，除了像前面的偶数情形那样添加两个索引外，还需要添加另外一个索引。总之，这意味着需要添加索引 4、4、5，如图 3-7 中所示。

在图 3-7 元素数组缓冲的下方，可以看出三角形带中三角形的对应索引。首先是第一个三角形带的 3 个三角形，对应的索引分别为(0,1,2)、(2,1,3)和(2,3,4)。然后是 5 个退化三

角形，它们最终会被 GPU 丢弃。最后是后一个三角形带中的两个三角形，它们的索引分别为(5,6,7)和(7,6,8)。

3.2 类型化数组

在到目前为止本书提供的 WebGL 完整示例中，都用 Float32Array 类型表示由 WebGL API 传递在到缓冲中的顶点数据，其代码类似于以下：

```
vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
var triangleVertices = [
        0.0,   0.5,  0.0,
       -0.5,  -0.5,  0.0,
        0.5,  -0.5,  0.0
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
  gl.STATIC_DRAW);
```

在本节中我们将介绍 Float32Array 到底是什么，以及为什么要用这个类型。

在 C 和 C++等程序设计语言中，需要处理二进制数据的情形并不少见，在这些语言中完全支持二进制数据的处理。但是在 JavaScript 语言中，二进制数据的处理并不常见，因此 JavaScript 语言并没有内置二进制数据的处理功能。开发人员在需要处理二进制数据时所采用的方法是把它们表示为字符串，然后用 JavaScript 的 charCodeAt()方法和位操作运算符(&、|、<<和>>)进行处理。

charCodeAt()方法返回位于 index 位置的字符的 Unicode 码，index 就是这个方法的参数。下面这个示例说明如何获得用字符串表示的数据中某个位置的特定字节：

```
var oneByte =  str.charCodeAt(index) & 0xFF;
```

利用这种技术的组合，我们可以读取二进制数据，并把它们转换为 32 位的整数或浮点数。但是在 WebGL 中，这个过程的开销太大。而且如果速度是一个关键因素，这通常不是一个切实可行的解决方法。因此，引入了 JavaScript 类型化数组类型，它提供一个比较有效的二进制处理方法。

3.2.1 缓冲与视图

为了处理二进制数据，类型化数组规范定义了缓冲和一个或多个缓冲视图等概念。缓冲是一个固定长度的二进制数据存储区，由类型 ArrayBuffer(数组缓冲)表示。例如，用下面的代码创建一个 8 字节的缓冲：

```
var buffer = new ArrayBuffer(8);
```

执行这条语句后，我们就得到一个 8 字节的缓冲。但是，无法直接对这个缓冲中的数

据进行处理。为此，需要创建 ArrayBuffer 的一个视图。从 ArrayBuffer 可以创建若干个不同的视图，前面我们看到的 Float32Array 就是这样一个视图。用下面的语句从名为 buffer 的数组缓冲创建一个 Float32Array 视图：

```
var viewFloat32 = new Float32Array(buffer)。
```

如果需要，也可以给同一个缓冲创建更多的视图。例如，我们可以同一个缓冲创建另外两个视图 Uint16Array 和 Uint8Array：

```
var viewUint16 = new Uint16Array(buffer);
var viewUint8 = new Uint8Array(buffer);
```

如果创建了一个 ArrayBuffer，并且以它为基础又创建了一个 Float32Array、一个 Uint16Array、一个 Uint8Array 视图，我们就得到如图 3-8 所示的对应关系。从中可以看出，在不同的视图中可以用不同的索引访问数组缓冲中的不同字节。例如，viewFloat32[0]代表 ArrayBuffer 中的字节 0~ 3，它表示一个 32 位的浮点数。viewUint[2]表示数组缓冲中的第 4 和第 5 个字节，它是一个 16 位的无符号整数。

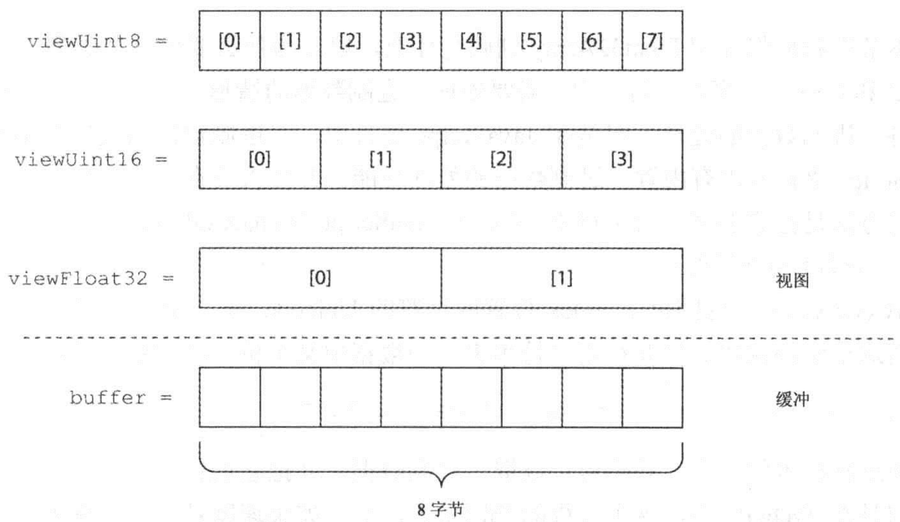


图 3-8　8 字节的数组缓冲与 3 个不同视图之间的对应关系

3.2.2　WebGL 支持的视图类型

前面介绍了类型化数组的 3 类不同视图。表 3-2 列出了 WebGL 中的全部视图类型以及相应的大小(即字节数)。

表 3-2　不同类型的类型化数组视图

视图类型	说明	元素大小(单位为字节)
Uint8Array	8 位无符号整数	1 字节
Int8Array	8 位有符号整数	1 字节
Uint16Array	16 位无符号整数	2 字节

(续表)

视图类型	说明	元素大小(单位为字节)
Int16Array	16 有符号整数	2 字节
Uint32Array	32 位无符号整数	4 字节
Int32Array	32 位有符号整数	4 字节
Float32Array	32 位 IEEE 浮点数	4 字节
Float64Array	64 位 IEEE 浮点数	8 字节

所有的视图类型都使用相同的构造函数，具有相同的属性、常量和方法。在类型化数组的规范中用通用术语 TypedArray 表示表 3-2 中的某个视图类型。

本节以 Float32Array 为例，但是其他视图类型也有相同的成员。在前一节(讨论缓冲与视图)的示例中，先创建一个缓冲，然后显式地创建几个视图。然而，你应该还记得通过 WebGL API 上传顶点的代码，如下所示：

```
vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
var triangleVertices = [
        0.0,   0.5,  0.0,
       -0.5,  -0.5,  0.0,
        0.5,  -0.5,  0.0
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
  gl.STATIC_DRAW);
```

这个示例使用 Float32Array 的构造函数，它以 JavaScript 数组为参数。

这个构造函数创建一个新的 ArrayBuffer 对象，且它的空间足以存放 JavaScript 数组 triangleVeritces 的内容。但是，这段代码也直接建立一个 Float32Array 视图，此视图绑定到这个缓冲上。triangleVertices 中的顶点数据上传到 ArrayBuffer 中。图 3-9 显示了 Float32Array 和相应的 ArrayBuffer，用前面的代码段把 ArrayBuffer 传入 gl.bufferData()方法。

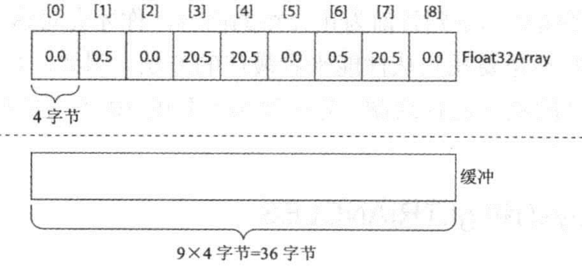


图 3-9 Float32Array 和相应的 ArrayBuffer

3.3 探讨不同的绘图方法

至此你已经对不同图元(即 gl.TRIANGLES、gl.TRIANGLE_STRIP 等)和两个方法(即

gl.drawArrays()和 gl.drawElements())有了一个初步了解。现在我们对几何对象的各种绘图方法进行比较。

在 WebGL 中，最常绘制的图元可能是三角形，绘制三角形可以使用 gl.drawArrays() 或 gl.drawElements()方法。通常我们使用 gl.TRIANGLES 或 gl.TRIANGLE_STRIP 图元。gl.TRINAGLE_FAN 图元比较少见，在绘制通用的三角形网格时很少使用它，因此这里的比较并没有把它包括在内。

假设我们要绘制如图 3-10 所示的网格，它由 50 个三角形组成。使用下面的方法与图元的组合，它们之间有什么不同呢？

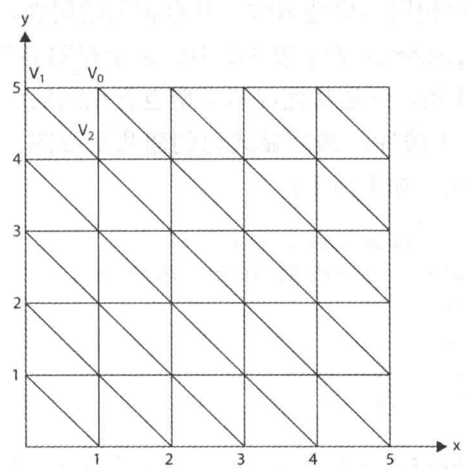


图 3-10　一个由 50 个三角形组成的简单网格

- gl.drawArrays()和 gl.TRIANGLES
- gl.drawArrays()和 gl.TRIANLGE_STRIP
- gl.drawElements()和 gl.TRIANGLE
- gl.drawElements()和 gl.TRIANGLE_STRIP

但是如果想知道需要用到哪些数组缓冲和元素数组缓冲以及要使用多少内存处理这些缓冲中的数据，则需要阅读本章到目前为止介绍的内容，你才会对这个问题有更好的理解。

注意，这种比较并不是要找到从性能来看最佳的方法。例如，对于呈现图 3-10 中的网格，并不是按效率最高的顺序进行绘制。先绘制第一列的 10 个三角形，然后绘制第二列的 10 个三角形，依此类推。

3.3.1　gl.drawArrays()和 gl.TRIANGLES

你可能会认为，使用 gl.drawArrays()和 gl.TRIANGLES 是最简单的方法。因为使用了 gl.drawArrays()，就不需要使用一个元素数组缓冲。对于 gl.TRIANGLE 图元，每个三角形需要 3 个顶点。顶点总数由以下公式决定：

$$需要的顶点数=3\times三角形数=3\times50=150\ 个顶点$$

因此，需要定义一个能够存放150个顶点的数组缓冲。如果只考虑顶点位置，并且用Float32Array表示位置，则存储此顶点数据所需要的内存由以下公式决定：

所需要的内存=150个顶点×3×4(字节/顶点)=1800字节

因此，150个顶点需要乘3(每个顶点有x、y、z三个坐标分量)，然后乘4，因为Float32Array的每个元素需要4字节。结果就是1800字节。

下面这段代码说明如何建立一个缓冲，以及如何用这个方法绘制网格。你可以看出，有些顶点可能会重复出现。例如，V_3与V_2的位置相同，而V_4与V_1位置相同。

```
function setupBuffers() {

  meshVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

  var meshVertexPositions = [
    1.0,  5.0,  0.0, //v0
    0.0,  5.0,  0.0, //v1
    1.0,  4.0,  0.0, //v2

    1.0,  4.0,  0.0, //v3 = v2
    0.0,  5.0,  0.0, //v4 = v1
    0.0,  4.0,  0.0, //v5

    1.0,  4.0,  0.0, //v6 = v3 = v2
    0.0,  4.0,  0.0, //v7 = v5
    1.0,  3.0,  0.0, //v8

    1.0,  3.0,  0.0, //v9 = v8
    0.0,  4.0,  0.0, //v10 = v7 = v5
    0.0,  3.0,  0.0, //v11

    ...

    5.0, 0.0,  0.0, //v148
    4.0, 1.0,  0.0, //v149
    4.0, 0.0,  0.0  //v150


  ];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(meshVertexPositions),
   gl.STATIC_DRAW);
  meshVertexPositionBuffer.itemSize = 3;
  meshVertexPositionBuffer.numberOfItems = 150;

  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

}
```

```
...

function draw() {
  ...

  gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);


  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
      meshVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);

  gl.drawArrays(gl.TRIANGLES, 0, meshVertexPositionBuffer.numberOfItems);
}
```

3.3.2 gl.drawArrays()方法和 gl.TRIANGLE_STRIP 图元

这种模式也需要调用 gl.drawArrays()方法，因此不需要元素数组缓冲。然而，由于使用了 gl.TRIANGLE_STRIP，因此与独立三角形相比，这种模式中的每个三角形需要较少的顶点。回顾本章前面有关三角形带的论述，其中给出如下公式：

所绘制三角形数=count – 2

这里的 count 是 gl.drawArrays()使用的顶点数。这意味着，50 个三角形所需要的顶点数由以下公式决定：

需要的顶点数=所绘制三角形数+2=50+2=52

因此我们需要定义一个可以保存 52 个顶点的数组缓冲。然而，由于这里使用 gl.TRIANGLE_STRIP 图元，因此绘制这个三角形网格需要调用 gl.drawArrays()五次。从性能角度来看，gl.drawArrays()方法的调用次数越少越好。解决的方法是使用退化三角形。在 1、2、3、4 列的末尾需要跳转到下一列。每次跳转需要增加两个顶点，这意味着对于退化三角形需要增加 8 个顶点。在前面的结果上增加 8 个顶点，因此一共需要 60 个顶点。如果使用 60 个顶点和 gl.TRIANGLE_STRIP 图元，则绘制这个网格只需要调用一次 gl.drawArrays()方法。

可以用下面的公式计算 60 个顶点所需要的内存大小：

需要的内存大小=60 个顶点×3×4(字节/顶点)=720 字节

该公式表示，60 个顶点乘 3(每个顶点有 x、y、z 坐标)，再乘 4(每个 Float32Array 的元素为 4 个字节)，这样就得到 720 个字节。从这里可以看出，使用 gl.TRIANGLE_STRIP 图元比使用 gl.TRIANGLES 图元节省内存，即使对于这样一个相对比较简单的网格也是如此。

下面是用这种方法绘制三角形网格的部分所选代码片段：

```
function setupBuffers() {

  meshVertexPositionBuffer = gl.createBuffer();
```

```
gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

var meshVertexPositions = [
    1.0,  5.0,  0.0, //v0
    0.0,  5.0,  0.0, //v1
    1.0,  4.0,  0.0, //v2
    0.0,  4.0,  0.0, //v3
    1.0,  3.0,  0.0, //v4
    0.0,  3.0,  0.0, //v5
    1.0,  2.0,  0.0, //v6
    0.0,  2.0,  0.0, //v7
    1.0,  1.0,  0.0, //v8
    0.0,  1.0,  0.0, //v9
    1.0,  0.0,  0.0, //v10
    0.0,  0.0,  0.0, //v11

    // The 2 vertices below create the jump
    // from column 1 to column 2 of the mesh
    0.0,  0.0,  0.0, //v12, degenerate
    2.0,  5.0,  0.0, //v13 degenerate

    2.0,  5.0,  0.0, //v14
    1.0,  5.0,  0.0, //v15
    2.0,  4.0,  0.0, //v16

      ...

    4.0,  1.0,  0.0, //v58
    5.0,  0.0,  0.0, //v59
    4.0,  0.0,  0.0  //v60
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(meshVertexPositions),
 gl.STATIC_DRAW);
meshVertexPositionBuffer.itemSize = 3;
meshVertexPositionBuffer.numberOfItems = 60;

gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

}

...

function draw() {
  ...

  gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);



  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
```

```
        meshVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);

    gl.drawArrays(gl.TRIANGLE_STRIP, 0, meshVertexPositionBuffer.numberOfItems);

}
```

3.3.3 gl.drawElements()方法和 gl.TRIANGLES 图元

如果使用 gl.drawElements()方法，则需要定义一个数组缓冲保存顶点数据和一个元素数组缓冲保存索引。对于数组缓冲，由于实际上此网格只有 36 个不同的顶点，因此只需要在数组缓冲中保存这 36 个顶点，所需要的内存大小可以下面的公式得到：

数组缓冲所需的内存大小=36 个顶点×3×4(字节/顶点)=432 个字节

对于此示例，还需要为元素数组缓冲中的索引分配内存，由于使用 gl.TRIANGLES 图元，每个三角形需要 3 个索引，因此有以下公式：

需要的索引数=3×所绘制三角形数=3×50=150 个索引

如果索引保存在类型为 Uint16Array 的元素数组缓冲中，则每个元素需要 2 个字节，因此所需要的内存由以下公式决定：

元素数组缓冲所需的内存大小=150 个索引×2(字节/索引)=300 字节

只要把上述两个公式的结果相加就可以得到所需要的总内存大小，即 432 字节加上 300 字节，因此需要的总内存为 732 字节。

下面是用这种方法绘制网格的代码片段：

```
function setupBuffers() {
  meshVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

  var meshVertexPositions = [
    1.0,  5.0,  0.0, //v0
    0.0,  5.0,  0.0, //v1
    1.0,  4.0,  0.0, //v2
    0.0,  4.0,  0.0, //v3
    1.0,  3.0,  0.0, //v4
    0.0,  3.0,  0.0, //v5
    1.0,  2.0,  0.0, //v6
    0.0,  2.0,  0.0, //v7
    1.0,  1.0,  0.0, //v8
    0.0,  1.0,  0.0, //v9
    1.0,  0.0,  0.0, //v10
    0.0,  0.0,  0.0, //v11

    ...
```

```
    4.0,  1.0,  0.0, //v34
    5.0,  0.0,  0.0, //v35
    4.0,  0.0,  0.0  //v36
  ];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(meshVertexPositions),
   gl.STATIC_DRAW);
  meshVertexPositionBuffer.itemSize = 3;
  meshVertexPositionBuffer.numberOfItems = 36;

  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

  meshIndexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, meshIndexBuffer);

  var meshIndex = [
    0, 1, 2,
    2, 1, 3,
    2, 3, 4,
    4, 3, 5,
    4, 5, 6,
    6, 5, 7,

    ...

    35, 34, 36

  ];

  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(meshIndex),
   gl.STATIC_DRAW);
  meshIndexBuffer.itemSize = 1;
  meshIndexBuffer.numberOfItems = 150;
}

...

function draw() {

...

  gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
    meshVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, meshIndexBuffer);

  gl.drawElements(gl.TRIANGLES,
                  meshIndexBuffer.numberOfItems,
                  gl.UNSIGNED_SHORT, 0);
}
```

3.3.4 gl.drawElements()方法和 gl.TRIANLE_STRIP 图元

由于这种情形也要用到 gl.drawElements()方法，因此需要一个数组缓冲和一个元素数组缓冲。数组缓冲的内容与前一个示例(即 gl.drawElements()和 gl.TRIANGLES 图元)使用的数组缓冲完全一样。这个网格需要 36 个不同的顶点位置，每个顶点位置由 x、y、z 三个坐标分量来确定，每个分量占用 4 个字节，因此所需要的内存为：

数组缓冲所需的内存大小=36×3×4=432 字节

不同之处在于元素数组缓冲的索引。现在再次利用三角形数与顶点数的关系：

所绘制三角形数=count – 2

这里的 count 表示使用 gl.drawElements()方法时所需要的索引数。50 个三角形所需要的索引数由下面的公式决定：

所需要的索引数=所绘制三角形数+2=50+2=52 个索引

但是我们希望只需要调用 gl.drawElements()一次，所以必须使用退化三角形，在相邻列之间建立跳转需要增加 8 个索引。增加这 8 个索引后，元素数组缓冲中一共有 60 个索引。如果元素数组缓冲是由 Uint16Array 元素组成的，则元素数组缓冲所占用的内存大小为：

元素数组缓冲所需的内存大小=60 个索引×2(字节/索引)=120 字节

在这种情形下所需要的总内存是 432 字节的数组缓冲和 120 字节的元素数组缓冲，一共是 552 字节。

这个示例的代码如下所示：

```
function setupBuffers() {
  meshVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

  var meshVertexPositions = [
    1.0,  5.0,  0.0, //v0
    0.0,  5.0,  0.0, //v1
    1.0,  4.0,  0.0, //v2
    0.0,  4.0,  0.0, //v3
    1.0,  3.0,  0.0, //v4
    0.0,  3.0,  0.0, //v5
    1.0,  2.0,  0.0, //v6
    0.0,  2.0,  0.0, //v7
    1.0,  1.0,  0.0, //v8
    0.0,  1.0,  0.0, //v9
    1.0,  0.0,  0.0, //v10
    0.0,  0.0,  0.0, //v11
    // start of column 2
    2.0,  5.0,  0.0, //v12
```

```
    1.0,  5.0,  0.0, //v13
    2.0,  4.0,  0.0, //v14

    ...

    4.0,  1.0,  0.0, //v34
    5.0,  0.0,  0.0, //v35
    4.0,  0.0,  0.0  //v36
  ];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(meshVertexPositions),
   gl.STATIC_DRAW);
  meshVertexPositionBuffer.itemSize = 3;
  meshVertexPositionBuffer.numberOfItems = 36;

  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

  meshIndexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, meshIndexBuffer);

  var meshIndex = [
    0, 1,  2,
    3, 4,  5,
    6, 7,  8,
    9, 10, 11,
    11, 12,   // indices for degenerate triangles
    12, 13, 14,

    ...

    34, 35, 36
  ];

  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(meshIndex),
   gl.STATIC_DRAW);
  meshIndexBuffer.itemSize = 1;
  meshIndexBuffer.numberOfItems = 60;
}

...

function draw() {

  ...

  gl.bindBuffer(gl.ARRAY_BUFFER, meshVertexPositionBuffer);

  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
     meshVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
```

```
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, meshIndexBuffer);

    gl.drawElements(gl.TRIANGLE_STRIP,
                    meshIndexBuffer.numberOfItems,
                    gl.UNSIGNED_SHORT, 0);
}
```

3.3.5 总结比较

前面分析了全部 4 种情况。表 3-3 归纳了前面的分析结果。从表中可以看出，当共享顶点很多时，gl.drawElements()方法与 gl.TRIANGLE_STRIP 图元的组合比 gl.drawArrays()方法与 gl.TRIANGLES 图元的组合可以节省很多的内存。

表 3-3 各种不同绘图方法的比较

方法与图元的组合	需要的总内存	是否需要元素数组缓冲	需要退化三角形吗？
gl.drawArrays()和 gl.TRIANGLES	1800	不需要	不需要
gl.drawArrays()和 gl.TRIANGLE_STRIP	720	不需要	需要
gl.drawElements()和 gl.TRIANGLES	720	需要	不需要
gl.drawElements()和 gl.TRIANGLE_STRIP	552	需要	需要

你还需要知道，这是一个只包含 50 个三角形的小网格，而且只考虑顶点位置。由于只考虑顶点的位置，因此数组缓冲的每个元素占用 12 个字节的内存。如果面对的是一个很大的网格，而且还需要包括法线和纹理坐标，则使用 gl.drawElements()方法和 gl.TRIANGLE_STRIP 图元的组合可以节省更多的内存。

重要的是，不要毫无必要地浪费内存，但是从性能角度来看，内存并不是唯一重要的因素。从性能角度来看，三角形绘制的顺序以及顶点数据的组织形式也是很重要的因素。下一节将介绍其理由。

3.3.6 前期变换顶点缓存和后期变换顶点缓存

在第 1 章中曾提到，通过 WebGL API 传递的顶点在到达 WebGL 流水线之前需要经过顶点着色器的变换处理。到达流水线之后，顶点要经过图元装配、光栅化、片段着色、逐片段处理、最后作为像素出现在绘图缓冲中。同一个顶点可能经常出现在网格的几个不同的三角形中。这意味着如果已经对某一个顶点进行变换，则之后对同一个顶点的变换都是多余的。

为了解决这个问题，现代的 GPU 都内置了一个缓存，它通常称为后期变换顶点缓存。它的使用是为了避免顶点着色器对同一个顶点多次处理。一个顶点经过顶点着色器变换后就保存在后期变换顶点缓存中，这个缓存通常比较小，而且采用先进先出模式(FIFO)，这意味着，它只缓存最近变换的顶点的结果。

这意味着，如果三角形的顶点按随机顺序传送，则许多顶点即使在前面已经过变换也

可能会未命中此缓存。为了表示后期变换顶点缓存的效率，我们采用平均缓存未命中率(Average Cache Miss Ratio，ACMR)这个概念。ACMR 定义为缓存未命中次数除以所绘制三角形数，即采用以下公式：

$$ACMR=\frac{\text{缓存未命中次数}}{\text{所绘制三角形数}}$$

显然，ACMR 越小越好。对于一个很大的网格，三角形数接近于顶点数的两倍。这在理论上意味着，ACMR 在最优情形下可以取最小值 0.5，即每个顶点都只变换一次。但实际上最坏的情形是，每个三角形的 3 个顶点都未命中缓存，此时 ACMR 的值为 3.0。

如果网格的三角形按某一个方法进行组织，使得后期变换顶点缓存能够得到充分利用，则从性能角度来看可以获得许多好处。人们已经在这个领域中进行了许多研究，而且开发了一些工具(如 Stripe，它由 Stony Brook 大学开发)，利用这些工具，我们可以以最优顺序组织网格的顶点。

除了后期变换顶点缓存外，有些 GPU 还通常提供前期变换顶点缓存。如果在前期变换顶点缓存中没有找到一个顶点，则需要读取这个顶点并把它传送给顶点着色器进行变换。

当需要读取一个顶点时，通常同时把这个顶点附近的一块比较大的连续顶点数据块读入缓存中。这意味着，即使在使用 gl.drawElements()的索引绘图模式时，顶点数据在顶点数组缓冲中也可以采用任何顺序，但是尽量使顶点的索引顺序与顶点的使用顺序相一致。

图 3-11 显示这方面的一个示例。在上方，数组缓冲中的顶点数据在内存中是连续存放的，当读取 V_0 时，很可能同时也把后续的一些顶点读入前期变换顶点缓存中。因此，当后面需要 V_1、V_2、V_3 时，很可能它们都已经在缓存中，不需要从慢速的内存中读取。

在图 3-11 的下方，顶点没有连续存放，采用这种组织方式，不大可能命中前期变换顶点缓存。

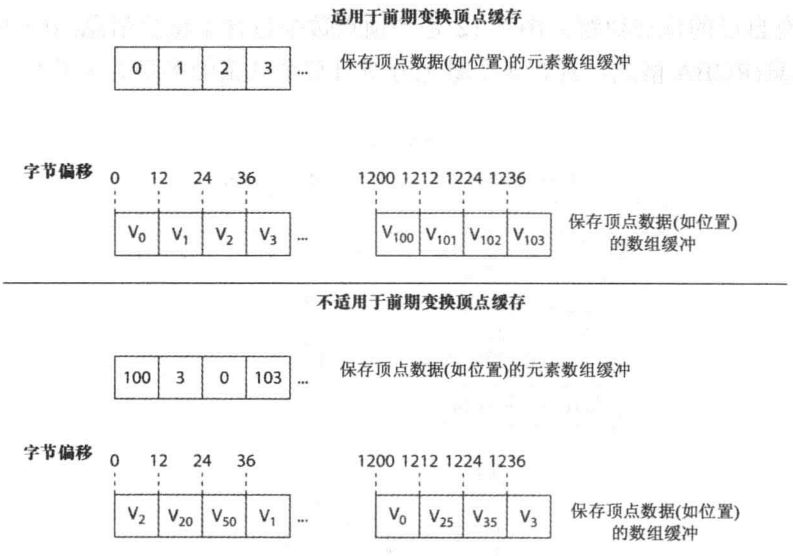


图 3-11 在上方，数组缓冲中的数据在内存中是连续存放的，这可能会提高前期变换顶点缓存的命中率。在下方，数组缓冲中的数据在内存中不是连续存放的，因此前期变换顶点缓存的命中率可能会很低

三角形绘制的顺序以及顶点在内存中的顺序对性能是有影响的。你很有必要了解这一点，但是无须对此过于担心，不要急于找到三角形的最优绘制顺序和顶点在内存中的最优存放顺序，当出现性能问题并且确信这个问题是与 WebGL 应用程序中的顶点有关时，再考虑这些因素。

这就是说，当用户知道你的图形对象有很多共享顶点时，一个经验规则是不要用 gl.drawArrays()方法，而是使用 gl.drawElements()的索引绘图方法，这至少可以提高后期变换顶点缓存的命中率。如果用户还没有以后期变换顶点缓存的最优方式排列三角形的顺序，则 gl.TRIANGLE_STRIP 图元通常会给 WebGL 应用程序提供最佳性能，前提是不要为了连接不同的三角形带而为退化三角形插入太多额外的顶点。

3.4 为提高性能交叉存放顶点数据

在前面的大多数示例中，顶点数据只包含顶点位置信息。然而，在实际的 WebGL 应用程序中，顶点数据通常还包含更多的信息。除了顶点位置信息，顶点数据还包括顶点法线、顶点颜色和纹理坐标。

当顶点包含多种数据时，可以采用以下两种方法组织这些数据：

- 把每类顶点数据保存在 WebGLBuffer 对象的单独数组中，这意味着，除了顶点位置数组外，可能还有其他数组，如法线数组。这通常称为数组结构。
- 把所有类型的数据都保存在 WebGLBuffer 对象的一个数组中。这意味着，需要把不同类型的数据交叉保存在同一个数组中。这通常称为结构数组。

一般来说，第一个方法(数组结构)是建立缓冲并把数据载入缓冲的最简单方法。每类顶点数据都有自己的顶点数组。图 3-12 是当顶点数据包含了位置信息(用坐标 x、y、z 表示)和颜色信息(RGBA 格式，具有 4 个颜色分量且每个分量的类型为无符号字节)时的数组结构。

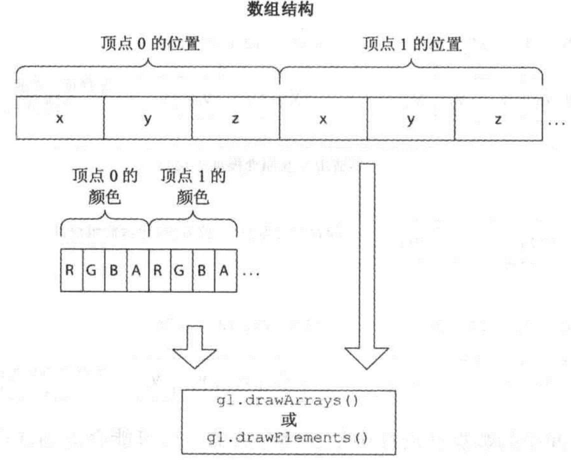


图 3-12　非交叉顶点数组(数组结构)工作方式的概念图

如果使用结构数组，则在建立缓冲和载入数据时需要更多的操作。图 3-13 显示了如何把位置数据和颜色数据交叉保存在同一个顶点数组中。

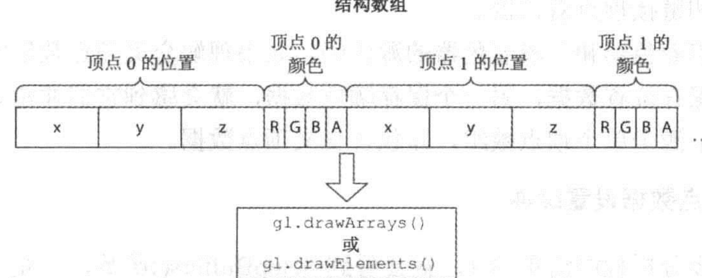


图 3-13　交叉顶点数组(结构数组)工作方式的概念图

从性能角度来看，把顶点数据按交叉模式保存在一个结构数组中是首选的顶点数据组织方式。这是因为它提供顶点数据的更好的内存局部性。当顶点着色器需要某个顶点的位置数据时，在同一个时刻，它也可能需要同一个顶点的法线数据和纹理数据。如果把这些数据保存在内存相近位置，则读取其中一个数据很可能同时读取同一个块中的其他数据，因此当顶点着色器需要它们时，它们已经出现在前期变换顶点缓存中。

结构数组的使用

理解结构数组的原理后，使用结构数组实际上并不困难。但是第一次使用时确实需要一点技巧，因此本节提供一个完整的示例介绍它的用法，这样你就可以把它载入浏览器，并且自己可以进行尝试。

我们尽量简化程序清单 3-1 中的几何模型，这样可以把重点放在交叉的顶点数组上。当用户把源代码载入浏览器时，可以看到如图 3-14 所示的一个彩色三角形。由于本书不是彩色排版，因此你无法看到其颜色。你最好亲自把源代码载入浏览器，看看运行结果。

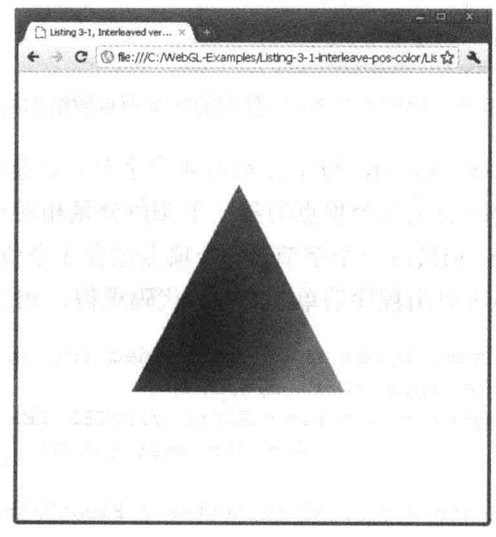


图 3-14　把程序清单 3-1 中的源代码载入浏览器，就会看到一个彩色三角形

你应该熟悉这个程序清单的大部分内容。你应该还记得，在第 2 章中曾提到，当在浏览器中载入页面时触发 JavaScript 函数 startup()。你应该还记得如何创建 WebGL 上下文，如何载入、编译和链接顶点着色器。

仔细分析顶点着色器和片段着色器的源代码。如果理解交叉顶点数组实际就是有两个属性变量，一个保存位置数据，另一个保存颜色数据，就会感到它们非常有意思。这两个属性变量的数据来源于单个顶点数组，即包含交叉顶点数据。

1. 为交叉顶点数据设置缓冲

如果你进一步分析程序清单 3-1，就会看到 setupBuffers()函数，它是本例中最重要的内容。在本例中使用该函数创建缓冲和在其中载入数据。

首先，用 gl.createBuffer()方法创建一个 WebGLBuffer 对象，并用 gl.bindBuffer()方法把它绑定到目标缓冲上。然后用标准的 JavaScript 数组定义顶点数据。这里按交叉模式排列顶点数据，即先是 3 个位置元素(x、y、z)，然后是 4 个颜色元素(即 r、g、b、a)，接下来又是 3 个位置元素和 4 个颜色元素，依此类推。本例首先用 JavaScript 数组保存顶点数据，这主要是为了让你对顶点数据的保存方式有一个比较清晰的概念。然后从这个 JavaScript 数组读取数据并载入一个类型化数组中。顶点数据也可以从外部文件或其他资源载入。

前面当我们需要把顶点数据载入 WebGLBuffer 对象时，需要创建一个类型化数组，并直接以一个 JavaScript 数组作为输入源。在这个示例中，由于交叉顶点数据包含不同类型的数据，因此现在的情况稍微复杂。本例创建的 WebGLBuffer 对象的结构如图 3-15 所示。

交叉的顶点数据

顶点 0 的位置　顶点 0 的颜色　顶点 1 的位置　顶点 1 的颜色

x | y | z | R | G | B | A | x | y | z | R | G | B | A | …

4 字节　4 字节　4 字节　1 字节

图 3-15　程序清单 3-1 中使用的交叉顶点数组的格式

首先是 3 个位置元素(x、y、z)，每个元素占 4 个字节。这意味着，3 个位置元素总共需要占 12 个字节，这同时也是这个顶点的第一个颜色分量相对于缓冲的起始位置的偏移值。每个颜色分量(r、g、b、a)只占一个字节。一个顶点包含 3 个位置分量和 4 个颜色分量，因此一个顶点的数据大小可以由程序清单 3-1 中的代码求得，相应的代码片段如下所示：

```
// Calculate how many bytes that are needed for one vertex element
// that consists of (x,y,z) + (r,g,b,a)
var vertexSizeInBytes = 3 * Float32Array.BYTES_PER_ELEMENT +
                        4 * Uint8Array.BYTES_PER_ELEMENT;

var vertexSizeInFloats = vertexSizeInBytes / Float32Array.BYTES_PER_ELEMENT;
```

很容易看出，vertexSizeInBytes 的值为 16，vertexSizeInFloats 的值是 4。得到了这些值后，分配一个 ArrayBuffer，保存实际的顶点数据。为了访问位置元素，要把 Float32Array 映射到这个 ArrayBuffer。此外，为了访问颜色元素，还需要把 Uint8Array 映射到这个 ArrayBuffer。

```
// Allocate the buffer
var buffer = new ArrayBuffer(nbrOfVertices * vertexSizeInBytes);

// Map the buffer to a Float32Array view to access the position
var positionView = new Float32Array(buffer);

// Map the same buffer to a Uint8Array to access the color
var colorView = new Uint8Array(buffer);
```

然后是一个循环，用于读取 JavaScript 数组的每个元素值并填充 ArrayBuffer：

```
// Populate the ArrayBuffer from the JavaScript Array
var positionOffsetInFloats = 0;
var colorOffsetInBytes = 12;
var k = 0; // index to JavaScript Array
for (var i = 0; i < nbrOfVertices; i++) {
  positionView[positionOffsetInFloats] = triangleVertices[k];      // x
  positionView[1+positionOffsetInFloats] = triangleVertices[k+1]; // y
  positionView[2+positionOffsetInFloats] = triangleVertices[k+2]; // z
  colorView[colorOffsetInBytes] = triangleVertices[k+3]; // R
  colorView[1+colorOffsetInBytes] = triangleVertices[k+4];        // G
  colorView[2+colorOffsetInBytes] = triangleVertices[k+5];        // B
  colorView[3+colorOffsetInBytes] = triangleVertices[k+6];        // A

  positionOffsetInFloats +=vertexSizeInFloats;
  colorOffsetInBytes +=vertexSizeInBytes;
  k +=7;
}
```

变量 positionOffsetInFloats 表示 Float32Array 的位置，这个位置就是每个顶点的坐标 x 分量写入的位置。第一次迭代循环时，它的值为 0。然后每次迭代增加 vertexSizeInFloats(值为 4)。变量 colorOffsetInBytes 是红色分量的 Uint8Array 中的位置。第一次迭代循环时，它的值为 12，然后每次迭代时，它的值增加 vertexSizeInBytes(值为 16)。

2. 基于交叉顶点数据绘图

为交叉顶点数据创建了正确的缓冲后，现在需要告诉 WebGL 其中的数据是如何组织的，然后根据这些数据进行绘图。

```
// Bind the buffer containing both position and color
gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);

// Describe how the positions are organized in the vertex array
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
```

```
        triangleVertexBuffer.positionSize, gl.FLOAT, false, 16, 0);

// Describe how colors are organized in the vertex array
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
        triangleVertexBuffer.colorSize, gl.UNSIGNED_BYTE, true, 16, 12);

// Draw the triangle
gl.drawArrays(gl.TRIANGLES, 0, triangleVertexBuffer.numberOfItems);
```

首先需要绑定包含交叉顶点数据的缓冲。然后针对顶点着色器中的每个属性，用 gl.vertexAttribPointer()方法连接到绑定缓冲中的数据上。在上述这段代码中，此方法的第一条调用语句指示 WebGL 绑定的 WebGLBuffer 对象中的位置数据是如何组织的。这个方法中各个参数的意义如下：

- 第一个参数告诉 WebGL，绑定的缓冲应该用作哪个通用属性索引的输入。这里定义的索引对应于顶点着色器中的 aVertexPosition 属性。
- 第二个参数定义每个位置信息要用多少个元素来表示。位置信息由 x、y、z 三个分量组成，因此 trianlgeVertexBuffer.positionSize 的值设置为 3，这个值通过第二个参数传递给 vertexAttribPointer()方法。
- 第三个参数设置为 gl.FLOAT，它表示位置元素是浮点类型的值。
- 第四个参数是规范化标志。它说明了如何处理非浮点类型的数据项。由于位置数据是浮点类型的，因此这个标志是否设置为 false 对位置数据并不重要。
- 第五个参数是跨度。它定义了一个元素的开始位置与下一个同类型元素的开始位置之间的间隔。从图 3-15 中可以看出，本例中的跨度为 16 字节。在前一个示例中，顶点数据没有采用交叉模式，因此跨度为 0。
- 第六个参数是本调用定义的顶点数据类型中第一个元素的偏移值。由于位置数据定义在颜色数据之前，因此位置的偏移值为 0。

掌握了第一条调用语句的用法后，第二条调用语句就容易理解了。在第二条调用语句中，第三个参数设置为 gl.UNSIGNED_BYTE，因为每个颜色分量用 ArrayBuffer 中的一个字节来表示。

第四个参数(规范化标志)现在很重要，因为现在的顶点数据是 gl.UNSIGNED_BYTE 而非 gl.FLOAT。所有供顶点着色器处理的顶点属性都是单精度浮点数，如果传入的顶点数据是另外一个类型，则在顶点着色器处理这些数据之前需要把它们转换为浮点数。规范化标志就定义如何进行这种转换。

如果把这个规范化标志设置为 true，则把所有无符号数值映射为[0.0,1.0]范围内的值，所有带符号数值都映射为[-1.0,1.0]范围内的一个值。由于顶点颜色数据定义为 gl.UNSIGNED_BYTE，因此所有的颜色数据都除以 255，映射为[0.0,1.0]中的值。

最后，必须注意第二次调用 gl.vertexAttribPointer()中的最一个参数，它是第一个颜色分量的偏移值。这个参数设置为 12，因为它的前面有 3 个浮点位置元素。

在 WebGL 论坛中经常发现的一个错误是，当初学者在交叉顶点数据上应用 gl.vertexAttribPointer()时，总是不能正确设置跨度参数。他们不是把这个参数设置为一个顶点元素的开始位置与下一个同类型顶点元素的开始位置之间的字节数，而是把它设置为一个顶点元素的结尾位置与下一个同类型顶点元素的开始位置之间的字节数。这个设置当然是错误的，肯定会产生问题。

可从 Wrox.com 下载源代码

程序清单 3-1　应用交叉顶点数组的一个示例

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Listing 3-1, Interleaved vertex data</title>
<meta charset="utf-8">
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;
  varying vec4 vColor;

  void main() {
    vColor = aVertexColor;
    gl_Position = vec4(aVertexPosition, 1.0);
  }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;

  varying vec4 vColor;
  void main() {
    gl_FragColor = vColor;
  }
</script>

<script src="webgl-debug.js"></script>

<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var triangleVertexBuffer;

function createGLContext(canvas) {
  var names = ["webgl", "experimental-webgl"];
  var context = null;
  for (var i=0; i < names.length; i++) {
    try {
```

```
        context = canvas.getContext(names[i]);
      } catch(e) {}
      if (context) {
        break;
      }
    }
    if (context) {
      context.viewportWidth = canvas.width;
      context.viewportHeight = canvas.height;
    } else {
      alert("Failed to create WebGL context!");
    }
    return context;
  }

  function loadShaderFromDOM(id) {
    var shaderScript = document.getElementById(id);

    // If we don't find an element with the specified id
    // we do an early exit
    if (!shaderScript) {
      return null;
    }

    // Loop through the children for the found DOM element and
    // build up the shader source code as a string
    var shaderSource = "";
    var currentChild = shaderScript.firstChild;
    while (currentChild) {
      if (currentChild.nodeType == 3) { // 3 corresponds to TEXT_NODE
        shaderSource += currentChild.textContent;
      }
      currentChild = currentChild.nextSibling;
    }

    var shader;
    if (shaderScript.type == "x-shader/x-fragment") {
      shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (shaderScript.type == "x-shader/x-vertex") {
      shader = gl.createShader(gl.VERTEX_SHADER);
    } else {
      return null;
    }

    gl.shaderSource(shader, shaderSource);
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert(gl.getShaderInfoLog(shader));
      return null;
```

```
  }
  return shader;
}

function setupShaders() {
  vertexShader = loadShaderFromDOM("shader-vs");
  fragmentShader = loadShaderFromDOM("shader-fs");

  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);

  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Failed to setup shaders");
  }

  gl.useProgram(shaderProgram);

  shaderProgram.vertexPositionAttribute =
  gl.getAttribLocation(shaderProgram, "aVertexPosition");

  shaderProgram.vertexColorAttribute =
  gl.getAttribLocation(shaderProgram, "aVertexColor");

  // We enable vertex attrib arrays for both position and color attribute
  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
  gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);
}

function setupBuffers() {

  triangleVertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);
  var triangleVertices = [
     //( x     y     z )  (r    g    b    a )
     // --------------------------------------
       0.0,  0.5,  0.0,   255,   0,    0,  255, // V0
       0.5, -0.5,  0.0,     0, 250,    6,  255, // V1
      -0.5, -0.5,  0.0,     0,   0,  255,  255  // V2
  ];

  var nbrOfVertices = 3;

  // Calculate how many bytes that are needed for one vertex element
  // that consists of (x,y,z) + (r,g,b,a)
  var vertexSizeInBytes = 3 * Float32Array.BYTES_PER_ELEMENT +
    4 * Uint8Array.BYTES_PER_ELEMENT;

  var vertexSizeInFloats = vertexSizeInBytes / Float32Array.BYTES_PER_ELEMENT;
```

```
    // Allocate the buffer
    var buffer = new ArrayBuffer(nbrOfVertices * vertexSizeInBytes);

    // Map the buffer to a Float32Array view to access the position
    var positionView = new Float32Array(buffer);

    // Map the same buffer to a Uint8Array to access the color
    var colorView = new Uint8Array(buffer);

    // Populate the ArrayBuffer from the JavaScript Array
    var positionOffsetInFloats = 0;
    var colorOffsetInBytes = 12;
    var k = 0; // index to JavaScript Array
    for (var i = 0; i < nbrOfVertices; i++) {
      positionView[positionOffsetInFloats] = triangleVertices[k];     //x
      positionView[1+positionOffsetInFloats] = triangleVertices[k+1]; //y
      positionView[2+positionOffsetInFloats] = triangleVertices[k+2]; //z
      colorView[colorOffsetInBytes] = triangleVertices[k+3];          //R
      colorView[1+colorOffsetInBytes] = triangleVertices[k+4];        //G
      colorView[2+colorOffsetInBytes] = triangleVertices[k+5];        //B
      colorView[3+colorOffsetInBytes] = triangleVertices[k+6];        //A

      positionOffsetInFloats +=vertexSizeInFloats;
      colorOffsetInBytes +=vertexSizeInBytes;
      k +=7;
    }

    gl.bufferData(gl.ARRAY_BUFFER, buffer, gl.STATIC_DRAW);
    triangleVertexBuffer.positionSize = 3;
    triangleVertexBuffer.colorSize = 4;
    triangleVertexBuffer.numberOfItems = 3;
  }

  function draw() {
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Bind the buffer containing both position and color
    gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);

    // Describe how the positions are organized in the vertex array
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
             triangleVertexBuffer.positionSize, gl.FLOAT, false, 16, 0);

    // Describe how colors are organized in the vertex array
    gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
        triangleVertexBuffer.colorSize, gl.UNSIGNED_BYTE, true, 16, 12);

    // Draw the triangle
    gl.drawArrays(gl.TRIANGLES, 0, triangleVertexBuffer.numberOfItems);
```

```
}

function startup() {
  canvas = document.getElementById("myGLCanvas");
  gl = WebGLDebugUtils.makeDebugContext(createGLContext(canvas));
  setupShaders();
  setupBuffers();
  gl.clearColor(1.0, 1.0, 1.0, 1.0);

  draw();
}
</script>

</head>

<body onload="startup();">
  <canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>
</html>
```

3.5 使用顶点数组或常量顶点数据

在到目前为止介绍的源代码中，顶点着色器中属性变量的输入值都来自于 WebGLBuffer 对象的一个顶点数组。顶点数组保存了几何对象中每个顶点专用的数据。为了让一个属性从一个顶点数组读取数据，需要进行以下设置：

- 用 gl.enableVertexAttribArray()方法激活对应顶点着色器中的属性的通用属性索引。
- 创建一个 WebGLBuffer 对象，把它绑定到顶点缓冲上，并把顶点数据载入到顶点缓冲。
- 调用 gl.vertexAttribPointer()方法，把顶点着色器中某个属性相对应的通用属性索引连接到绑定的 WebGLBuffer 对象上。

然而，如果顶点数据对于一个图元的所有顶点都是常量，则不需要把这个值复制到顶点数组的每个元素中。

相反，我们可以禁用与希望传入常量数据的顶点着色器中这个属性相对应的通用属性索引，为此要调用 gl.disableVertexAttribArray()方法。然后将所有顶点的数据设置为这个常量值。例如，为了给类型 vec4 的属性设置常量顶点数据，需要调用下面的方法：

```
gl.vertexAttrib4f(index,r,g,b,a)
```

在这条调用语句中，index 参数是需要设置顶点数据的通用属性索引。r、g、b、a 参数是要给所有顶点设置的 4 个颜色分量。还有相应的方法设置 3 个、两个或一个浮点数：

```
gl.vertexAttrib3f(index, x, y, z);
gl.vertexAttrib2f(index, x, y);
gl.vertexAttrib1f(index, x);
```

图 3-16 说明了 gl.enableVertexAttribArray()和 gl.disableVertexAttribArray()方法的工作方式。

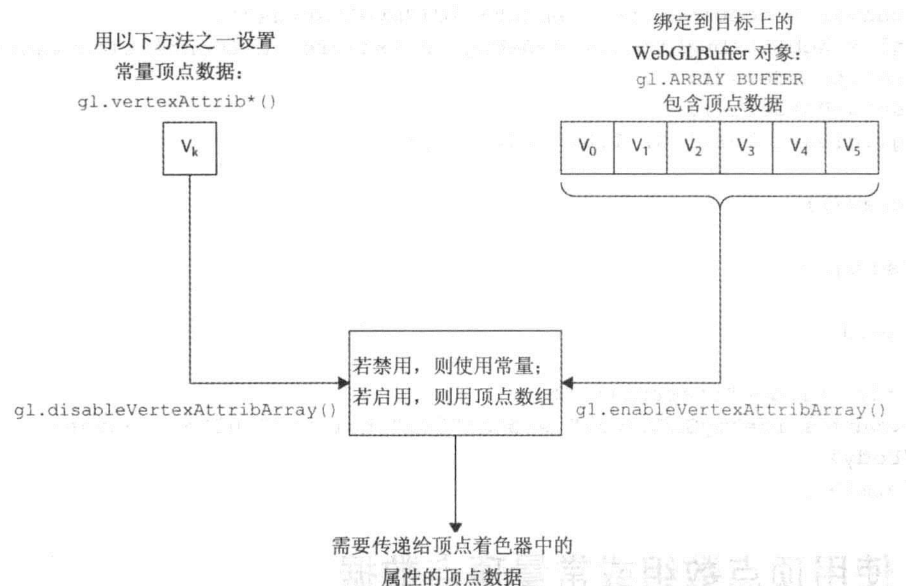


图 3-16　gl.enableVertexAttribArray()方法和 gl.disableVertexAttribArray()方法工作方式的概念图

3.6　总结本章的最后一个示例

你已经学习了 WebGL 的几种绘图方法以及绘制过程中可以使用的各个选项，现在介绍本章的最后一个示例，它应用了本章前面介绍的几个新概念。程序清单 3-2 是这个示例的源代码，它用 gl.drawArrays()和 gl.drawElements()方法以及线和三角形图元绘制一个多边形、一个独立三角形和一个三角形带。如果将程序清单 3-2 中的代码载入浏览器，则会看到如图 3-17 所示的结果。虽然你只需要对照图 3-17 就能看懂后面的说明，但是如果亲自把源代码载入浏览器中并分析运行结果，则会更有意义。

先分析着色器源代码，因为它最容易理解。可以看出，这个顶点着色器有两个属性变量，它们都从 WebGL API 获得输入值。你在后面的代码中将会看到，aVertexPostion 从绑定的、已激活的 WebGLBuffer 对象获得输入值，而另一个属性变量的值分别来自一个绑定的 WebGLBuffer 对象和一个由 WebGL API gl.vertexAttrib4f()方法传递过来的常量颜色。

aVertexPosition 现在扩展为 vec4 齐次坐标，然后直接赋给内置变量 gl_Position，而没有经过任何的变换处理。

片段着色器接受可变变量 vColor 的值，并把它赋给特殊的内置变量 gl_FragColor。有必要指出，在片段着色器中，这个可变变量 vColor 是由顶点着色器中通过 vColor 发送过来的值的线性插值得到的。这可以由图 3-17 右上角的独立三角形看出。这个三角形的 3 个顶点分别为红色、绿色和蓝色，位于 3 个顶点之间的像素的颜色是由这三个颜色的值进行线

性插值得到的(为了对这个三角形的颜色有更好的了解，在自己的浏览器中载入这段源代码)。

图 3-17 对应程序清单 3-2 的 WebGL 几何对象

如果继续往下查看源代码，找到 JavaScript 开始的位置，则会看到几个全局变量：

```
var gl;
var canvas;
var shaderProgram;
var hexagonVertexBuffer;
var triangleVertexBuffer;
var triangleVertexColorBuffer;
var stripVertexBuffer;
var stripElementBuffer;
```

这里的最后 5 个变量都用来存放各个 WebGLBuffer 对象。hexagonVertexBuffer 用来保存六边形的顶点位置，triangleVertexBuffer 用来存放独立三角形的 3 个顶点位置，ttiangle-VertexColorBuffer 用来存放三角形的顶点颜色。stripVertexBuffer 保存了三角带的顶点位置。三角形带的绘制采用了索引与 gl.drawElements()相结合的方法，因此除了数组缓冲外，它还需要元素数组缓冲，即 stripElementBuffer。

六边形和三角形带的颜色对于所有顶点都是一个常量，因此它们的颜色用 gl.vertex-Attrib4f()方法设置。因此，没有必要为这两个对象的颜色定义缓冲。

继续往下查看，直到下一个高亮的代码段为止。这段代码就是前面介绍的缓冲创建和设置代码。图 3-18 显示了本例中各个对象的顶点位置。

需要注意的是，由于在顶点着色器中不需要进行变换，因此只使用 WebGL 的默认坐

标系。这个默认的坐标系有一个更好的名称，即裁剪坐标系或裁剪空间。相应的坐标称为裁剪坐标，这些坐标值正是 gl_Position 的输入源。在第 4 章中，我们还要讨论各个不同的坐标系以及它们之间的转换。

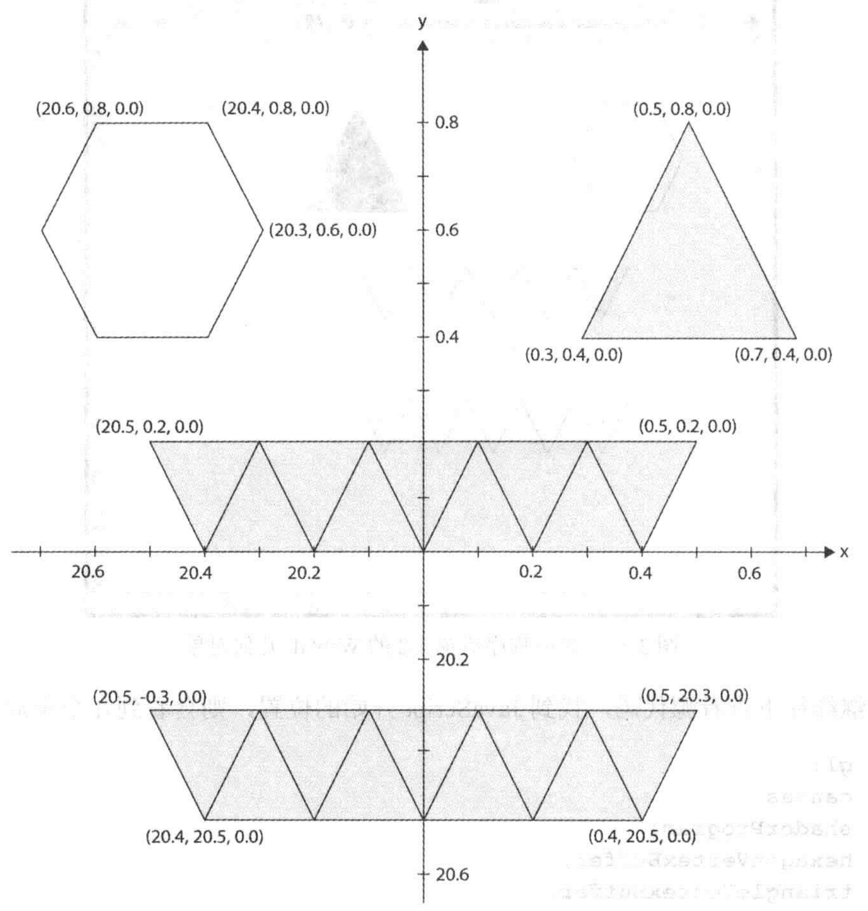


图 3-18　几何对象在裁剪坐标系中的位置，这些坐标值正是 gl_Position 变量所需要的值

在裁剪坐标中，视口左下角的位置为 x=-1 和 y=-1，右上角的位置为 x=1 和 y=1。因此，如果几何对象需要显示在这个视口中，则它们的 x 和 y 坐标值必须位于这个范围之内。由于本例只绘制 2D 对象，因此 z 坐标可以设置为 0。

创建和载入顶点的源代码你现在应该不会陌生。但是，你必须格外注意如何创建三角形带的缓存和如何给它设置顶点数据。如图 3-17 所示，这个三角形带实际上是由两个分离的三角形带组成的，它们之间并没有几何对象连接。但是，从源代码中可以看出，只有一个 WebGLBuffer 对象用于保存这个三角形带的全部顶点。这里用退化三角形把这两个三角形带连接成一个。

由于第一个三角形带包含奇数个三角形，因此需要插入 3 个额外的索引，才可以建立从第一个三角形带到第二个三角形带的连接。这些额外的索引分别为 10、10 和 11，这可以从下面的 indices 数组的第二行看出。

```
stripElementBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, stripElementBuffer);
  var indices = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                10, 10, 11, // 3 extra indices for the degenerate triangles
                11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21];

  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
    gl.STATIC_DRAW);

stripElementBuffer.numberOfItems = 25;
```

本例中另一个有趣的部分是绘图。首先用下面的代码段绘制这个六边形：

```
// Draw the hexagon
// We disable the vertex attrib array since we want to use a
// constant color for all vertices in the hexagon
gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);
gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 0.0, 0.0, 0.0, 1.0);

gl.bindBuffer(gl.ARRAY_BUFFER, hexagonVertexBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                      hexagonVertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

gl.drawArrays(gl.LINE_STRIP, 0, hexagonVertexBuffer.numberOfItems);
```

在调用gl.drawArrays()之前，需要先正确设置数据。首先调用g.disableVertexAttribArray()方法，将顶点的颜色设置为一个常量，在此使用 gl.vertexAttrib4f()方法将这个颜色设置为不透明的黑色，而不是使用来自颜色数组中的值。然后，将这个缓冲绑定到gl.ARRAY_BUFFER上，并调用gl.vertexAttribPointer()方法把顶点着色器中的aVertexPosition链接到hexagonVertexPosition，后者保存了六边形的顶点位置。最后，调用gl.drawArrays()方法绘制这个六边形。

下一个需要绘制的对象是前面提到的独立三角形，下面是它的绘制代码：

```
// Draw the independent triangle
// For the triangle we want to use per-vertex color so
// we enable the vertexColorAttribute again
gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);

gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
   triangleVertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
   triangleVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);

gl.drawArrays(gl.TRIANGLES, 0, triangleVertexBuffer.numberOfItems);
```

关于这段代码需要注意的是，现在必须调用gl.enableVertexAttribArray()方法激活顶点

属性数组，这是由于三角形带从一个顶点数组而不是一个常量顶点属性读取颜色值。

下面是绘制这个三角形带的代码：

```
// draw triangle-strip
// We disable the vertex attribute array for the vertexColorAttribute
// and use a constant color again.
gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);
gl.bindBuffer(gl.ARRAY_BUFFER, stripVertexBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                   stripVertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 1.0, 1.0, 0.0, 1.0);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, stripElementBuffer);

gl.drawElements(gl.TRIANGLE_STRIP, stripElementBuffer.numberOfItems,
 gl.UNSIGNED_SHORT, 0);
gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 0.0, 0.0, 0.0, 1.0);
```

这里再次禁用顶点属性数组。需要指出的是，数组缓冲和元素数组缓冲都需要绑定，因为这里采用了 gl.drawElements()进行索引的绘制方法。

最后是两次调用函数 gl.drawArrays()的代码，它们用来绘制两个三角形带之间的辅助线：

```
// Draw help lines to easier see the triangles
// that build up the triangle-strip
gl.drawArrays(gl.LINE_STRIP, 0, 11);
gl.drawArrays(gl.LINE_STRIP, 11, 11);
```

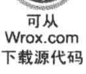
第一条调用语句绘制一个线带，它从顶点位置 0 开始，总共使用了 11 个顶点。第二条调用语句绘制的线带从顶点位置 11 开始，总共使用 11 个顶点。

在程序清单 3-2 的末尾插入了 startup()函数，在这个函数中，以下 3 个调用用来激活背面剔除操作：

```
gl.frontFace(gl.CCW);
gl.enable(gl.CULL_FACE);
gl.cullFace(gl.BACK);
```

在这个示例中，实际上不需要进行背面剔除，因为所有的三角形都采用逆时针组绕顺序，gl.cullFace()指定只对背面三角形进行剔除操作。但是，激活背面剔除处理可以确保为退化三角形正确增加额外顶点。记住，取决于这些额外顶点的增加方法，我们可能会改变第二个三角形带的组绕方向，因此需要进行背面剔除处理。

程序清单 3-2　绘制一个六边形、一个三角形和一个三角形带的示例




```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Listing 3-2, Lines, Triangle and Triangle Strip</title>
```

```
<meta charset="utf-8">
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;
  varying vec4 vColor;

  void main() {
    vColor = aVertexColor;
    gl_Position = vec4(aVertexPosition, 1.0);
  }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;

  varying vec4 vColor;
  void main() {
    gl_FragColor = vColor;
  }
</script>

<script src="webgl-debug.js"></script>

<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var hexagonVertexBuffer;
var triangleVertexBuffer;
var triangleVertexColorBuffer;
var stripVertexBuffer;
var stripElementBuffer;

function createGLContext(canvas) {
  var names = ["webgl", "experimental-webgl"];
  var context = null;
  for (var i=0; i < names.length; i++) {
    try {
      context = canvas.getContext(names[i]);
    } catch(e) {}
    if (context) {
      break;
    }
  }
  if (context) {
    context.viewportWidth = canvas.width;
    context.viewportHeight = canvas.height;
  } else {
    alert("Failed to create WebGL context!");
  }
```

```
  return context;
}

function loadShaderFromDOM(id) {
  var shaderScript = document.getElementById(id);

  // If we don't find an element with the specified id
  // we do an early exit
  if (!shaderScript) {
    return null;
  }

  // Loop through the children for the found DOM element and
  // build up the shader source code as a string
  var shaderSource = "";
  var currentChild = shaderScript.firstChild;
  while (currentChild) {
    if (currentChild.nodeType == 3) { // 3 corresponds to TEXT_NODE
      shaderSource += currentChild.textContent;
    }
    currentChild = currentChild.nextSibling;
  }

  var shader;
  if (shaderScript.type == "x-shader/x-fragment") {
    shader = gl.createShader(gl.FRAGMENT_SHADER);
  } else if (shaderScript.type == "x-shader/x-vertex") {
    shader = gl.createShader(gl.VERTEX_SHADER);
  } else {
    return null;
  }

  gl.shaderSource(shader, shaderSource);
  gl.compileShader(shader);

  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(shader));
    return null;
  }
  return shader;
}

function setupShaders() {
  vertexShader = loadShaderFromDOM("shader-vs");
  fragmentShader = loadShaderFromDOM("shader-fs");

  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);
```

```
  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Failed to setup shaders");
  }

  gl.useProgram(shaderProgram);

  shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
   "aVertexPosition");
  shaderProgram.vertexColorAttribute = gl.getAttribLocation(shaderProgram,
   "aVertexColor");

  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
}

function setupBuffers() {
  hexagonVertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, hexagonVertexBuffer);
  var hexagonVertices = [
       -0.3,  0.6,  0.0, //v0
       -0.4,  0.8,  0.0, //v1
       -0.6,  0.8,  0.0, //v2
       -0.7,  0.6,  0.0, //v3
       -0.6,  0.4,  0.0, //v4
       -0.4,  0.4,  0.0, //v5
       -0.3,  0.6,  0.0, //v6
  ];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(hexagonVertices),
   gl.STATIC_DRAW);
  hexagonVertexBuffer.itemSize = 3;
  hexagonVertexBuffer.numberOfItems = 7;

  triangleVertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);
  var triangleVertices = [
       0.3,  0.4,  0.0, //v0
       0.7,  0.4,  0.0, //v1
       0.5,  0.8,  0.0, //v2
  ];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
   gl.STATIC_DRAW);
  triangleVertexBuffer.itemSize = 3;
  triangleVertexBuffer.numberOfItems = 3;

  triangleVertexColorBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
  var colors = [
          1.0, 0.0, 0.0, 1.0, //v0
          0.0, 1.0, 0.0, 1.0, //v1
```

```
            0.0, 0.0, 1.0, 1.0 //v2
        ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
    triangleVertexColorBuffer.itemSize = 4;
    triangleVertexColorBuffer.numberOfItems = 3;


    stripVertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, stripVertexBuffer);
    var stripVertices = [
        -0.5,  0.2,  0.0, //v0
        -0.4,  0.0,  0.0, //v1
        -0.3,  0.2,  0.0, //v2
        -0.2,  0.0,  0.0, //v3
        -0.1,  0.2,  0.0, //v4
         0.0,  0.0,  0.0, //v5
         0.1,  0.2,  0.0, //v6
         0.2,  0.0,  0.0, //v7
         0.3,  0.2,  0.0, //v8
         0.4,  0.0,  0.0, //v9
         0.5,  0.2,  0.0, //v10
         // start second strip
        -0.5, -0.3,  0.0, //v11
        -0.4, -0.5,  0.0, //v12
        -0.3, -0.3,  0.0, //v13
        -0.2, -0.5,  0.0, //v14
        -0.1, -0.3,  0.0, //v15
         0.0, -0.5,  0.0, //v16
         0.1, -0.3,  0.0, //v17
         0.2, -0.5,  0.0, //v18
         0.3, -0.3,  0.0, //v19
         0.4, -0.5,  0.0, //v20
         0.5, -0.3,  0.0  //v21
    ];

    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(stripVertices),
     gl.STATIC_DRAW);
    stripVertexBuffer.itemSize = 3;
    stripVertexBuffer.numberOfItems = 22;

    stripElementBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, stripElementBuffer);
    var indices = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                   10, 10, 11, // 3 extra indices for the
     degenerate triangles
                  11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21];

    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
     gl.STATIC_DRAW);
```

```
    stripElementBuffer.numberOfItems = 25;
  }

  function draw() {
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Draw the hexagon
    // We disable the vertex attrib array since we want to use a
    // constant color for all vertices in the hexagon
    gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);
    gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 0.0, 0.0, 0.0, 1.0);

    gl.bindBuffer(gl.ARRAY_BUFFER, hexagonVertexBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                 hexagonVertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

    gl.drawArrays(gl.LINE_STRIP, 0, hexagonVertexBuffer.numberOfItems);

    // Draw the independent triangle
    // For the triangle we want to use per-vertex color so
    // we enable the vertexColorAttribute again
    gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);

    gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                 triangleVertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

    gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
                 triangleVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);

    gl.drawArrays(gl.TRIANGLES, 0, triangleVertexBuffer.numberOfItems);

    // draw triangle-strip
    // We disable the vertex attribute array for the vertexColorAttribute
    // and use a constant color again.
    gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);
    gl.bindBuffer(gl.ARRAY_BUFFER, stripVertexBuffer);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                  stripVertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

    gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 1.0, 1.0, 0.0, 1.0);
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, stripElementBuffer);

    gl.drawElements(gl.TRIANGLE_STRIP, stripElementBuffer.numberOfItems,
     gl.UNSIGNED_SHORT, 0);
    gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, 0.0, 0.0, 0.0, 1.0);

    // Draw help lines to easier see the triangles
```

```
    // that build up the triangle-strip
    gl.drawArrays(gl.LINE_STRIP, 0, 11);
    gl.drawArrays(gl.LINE_STRIP, 11, 11);
  }

  function startup() {
    canvas = document.getElementById("myGLCanvas");
    gl = WebGLDebugUtils.makeDebugContext(createGLContext(canvas));
  setupShaders();
    setupBuffers();
    gl.clearColor(1.0, 1.0, 1.0, 1.0);

    gl.frontFace(gl.CCW);
    gl.enable(gl.CULL_FACE);
    gl.cullFace(gl.BACK);

    draw();
  }
  </script>

  </head>

  <body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
  </body>

  </html>
```

实验示例

现在你已经遍览所有源代码，也阅读了这个源代码的说明。接下来应该对这个示例进行实验，尝试进行一些修改。你不妨试试以下操作：

- 改变退化三角形的索引，从而只用两个额外的索引(即 10 和 11)。此外还要更新 stripElementBuffer.numberOfItem 的值。注意下方的三角形带如何消失。为什么会消失？
- 保持上面的修改不变(即只用 10 和 11 两个额外的顶点索引建立退化三角形)，把 gl.frontFace()的参数从 gl.CCW 改为 gl.CW，看看有什么结果并了解其原因。
- 删除激活背面剔除的语句，看看有什么结果。
- 删除调用 gl.drawArrays()的两行代码，这两行代码用于绘制三角形带中三角形之间的辅助线。看看会有什么结果。

3.7 小结

本章介绍了 WebGL 的各个绘图选项。此外还介绍了可以与 gl.drawArrays()和 gl.drawElements()这两个绘图方法结合使用的 7 个不同图元：

- gl.TRIANGLES

- gl.TRIANGLE_STRIP
- gl.TRIANGLE_FAN
- gl.LINES
- gl.LINE_STRIP
- gl.LINE_LOOP
- gl.POINTS

现在你已经知道三角形图元是 WebGL 中可以使用的最基本构建块。你还知道了如何利用三角形的组绕方向确定这个三角形是否面向用户。此外，你还了解了如何剔除背面朝向用户的三角形。

本章也介绍了什么是类型化数组以及它在 WebGL 中的应用，即通过它们向 WebGL API 传递顶点数据，以及如何进一步向 GPU 传送数据。

最后，你现在应该具有足够的基础知识，了解在未来的 WebGL 应用程序中选择哪个图元以及使用哪种绘图方法。你还知道，前期变换顶点缓存和后期变换顶点可能对 WebGL 应用程序的性能有影响。

第 4 章

小型 JavaScript 库与变换

本章主要内容：

- 如何利用 JavaScript 库处理 JavaScript 中的向量和矩阵运算
- 如何利用变换运算确定对象在 3D 场景中的位置和方向
- 概述 WebGL 中几个重要的变换运算
- 介绍顶点在到达屏幕之前经历的各个坐标系或坐标空间
- 如何用一个变换矩阵栈让不同的对象具有不同的变换
- 如何在 WebGL 中确定虚拟照相机的位置和方向

第 1 章介绍了 WebGL 中经常用到的一些线性代数基本知识，还特别介绍了一些重要变换的数学理论。本章介绍如何在 WebGL 中应用这些变换。但是首先将介绍 3 个 JavaScript 库，它们可以用来处理 JavaScript 中的向量和矩阵运算。它们在 WebGL 应用程序中非常有用。

此外，本章还介绍顶点在通过 WebGL API 传入并在 WebGL 流水线中到达屏幕之前经历的各个坐标系或空间。在将阅读了本章之后，你将能够利用这些变换确定对象在 3D 空间场景中的位置和朝向。

4.1 JavaScript 中矩阵和向量的操作

JavaScript 内部并没有支持矩阵或向量的运算。与矩阵或向量最接近的 JavaScript 数据类型是内置的 Array 对象。你可以自己动手用 JavaScript 语言编写函数实现矩阵和向量运算，或者使用现有的某个开源库。下面 3 个库都可以在 Web 上找到，而且经常用在 WebGL 应用程序中：

- Sylvester
- WebGL-mjs

- glMatrix

这 3 个库都相对比较小，而且它们的功能很相似。另一个共同之处是它们并没有脱离 WebGL API 细节内容。另外有几个大型库，它们建立在 WebGL 抽象层之上。本书将这些大型库称为 WebGL 框架。

这些 WebGL 框架包含很多功能，但是它们经常通过提供一个 API 来隐藏许多与 WebGL 有关的细节。使用 WebGL 时需要学习这些细节。在实际绘制对象之前，需要处理许多细节。例如，需要创建着色器对象，将源代码载入着色器对象中，编译着色器，链接着色器等。这些任务经常由 WebGL 框架在内部处理，用户的应用程序不需要考虑所有这些细节。

虽然用户在许多方面欣赏这样的功能，但是目前你可能想掌握 WebGL 的这些细节。因此利用几个小型库之一实现矩阵和向量运算并能够看到 WebGL API 的处理细节是比较好的选择。

在学习这些库之前，有一个实际的细节必须有所了解。为了最小化下载时间，大多数发布在 Web 服务器上并且可以在实际 Web 应用程序中应用的 JavaScript 库都采用某种压缩方法。压缩过程通常会删除其中的空格和注释，这些内容并不是 JavaScript 引擎执行 JavaScript 程序所必需的。

有时，JavaScript 库以两种不同的形式出现：

- 压缩版本，当用户的应用程序完成开发后且准备发布到 Web 服务器上时，需要用到这种版本。
- 非压缩版本，这种格式方便用户阅读源代码。调试自己的应用程序，理解它的工作方式。

有时，包含非压缩版本库的文件用它的名称说明其版本。例如，Sylvester 库的非压缩版本使用了 sylvester.src.js 文件名，而 sylvester.js 文件包含了这个库的压缩版本。但是，有时情况正好与此相反。包含库的压缩版本的文件用文件名说明它是一个压缩版本或是一个最小化库。例如，对于 JavaScript 的 glMatrix 库，它的压缩版本文件名为 glMatrix-min.js，而它的非压缩版本文件名为 glMatrix.js。

但是，你不必过多考虑这些文件名。当用文本编辑器打开这些文件时，很容易看出哪个文件包含压缩版本，哪个文件包含非压缩版本。在本书的示例中，使用了库的非压缩版本，目的是为了方便你阅读源代码和调试库。

现在介绍这 3 个 JavaScript 库。如果你用自己常用的搜索引擎搜索 Internet，就可以找到这些库的最新源代码和文档。

4.1.1 Sylvester 库

Sylvester 库是由 James Coglan 开发的，它是为 JavaScript 设计的一个通用向量和矩阵运算库。它并不是专为 WebGL 开发的，因此缺少出现在 WebGL-mjs 库和 glMatrix 库中的某些功能。相反，它比较通用，可以处理任意大小的矩阵和向量。此外，它也可以处理 3D 空间的线和平面，这些功能正是其他两个库所没有的。

现在仔细分析 Sylvester 库的功能和表示方法。为了新建一个向量，使用下面的函数：

```
Vector.create(elements)
```

这个函数根据 elements 参数创建一个新向量，elements 是一个 JavaScript 数组。这个函数有一个简化的别名：

```
$V(elements);
```

这意味着，如果想建立一个包含 3 个元素([3、4、5])的向量时，可以直接用下面的语句：

```
var v=Vector.create([3,4,5]);
```

或者采用简化形式：

```
v = $V([3,4,5]);
```

假设有两个向量，则可以用 Sylvester 库方便地求得它们的和、点积或叉积：

```
var u = Vector.create([1,2,3]);
var v = Vector.create([4,5,6]);

var s = u.add(v); // s = [5,7,9]
var d = u.dot(v); // d = 1*4+2*5+3*6 = 32
var c = u.cross(v); // c = [-3,6,-3]
```

现在介绍矩阵运算的表示方法。用下面的代码建立一个新矩阵：

```
Matrix.create(elements)
```

与 Vector.create()一样，Matrix.create()方法也有一个简化的别名：

```
$M(elements)
```

这里的 elements 是一个嵌套的数组，这正是 Sylvester 与其他两个库的不同之处，后者用一维数组填充一个矩阵。对于 Sylvester，顶层数组包含行，每一行本身是一个元素数组。因此，为了创建一个 3×2 的矩阵，我们要用下面的代码：

```
var M = Matrix.create([[ 2, -1],   // first row
                       [-2, 1],    // second row
                       [-1, 2]]);  // third row
```

或者使用简化的表示方法：

```
var M = $M([[ 2, -1], // first row
            [-2, 1],  // second row
            [-1, 2]]); // third row
```

假设有两个矩阵(用 M 和 N 表示)，则它们的积 MN 可以表示为：

```
var MN=M.multiply(N)
```

程序清单 4-1 说明了如何在嵌套 JavaScript 的 HTML 文件中使用 Sylvester 库的部分功

能。除了前面介绍的方法外，这个示例还介绍了 inspect()的用法。在 Sylvester 库中，这个方法用于向量和矩阵。它返回一个对象的字符串表示，是一个非常有用的代码调试工具。

可从 Wrox.com 下载源代码

程序清单 4-1　用 Sylvester JavaScript 库实现向量和矩阵的数学运算

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Test of Sylvester JavaScript Library</title>
<meta charset="utf-8">
<script type="text/javascript" src="sylvester.src.js"></script>
<script type="text/javascript">

function testSylvesterJsLibrary() {
  var u = Vector.create([1,2,3]);
  var v = Vector.create([4,5,6]);

  var s = u.add(v);
  alert(s.inspect());                // s = [5, 7, 9]

  var d = u.dot(v); // d = 1*4+2*5+3*6 = 32
  alert(d);          // will alert 32

  var c = u.cross(v);
  alert(c.inspect()); // Will alert [-3, 6, -3]

  var M = Matrix.create([[ 2, -1], // first row
                         [-2, 1],  // second row
                         [-1, 2]]); // third row

var N = Matrix.create([[4, -3],      // first row
                       [3, 5]]); // second row

var MN = M.multiply(N);

alert(MN.inspect()); // will alert [ 5, -11]
                     //            [-5, 11]
                     //            [ 2, 13]


var I = Matrix.I(4); // create identity matrix
alert(I.inspect());  // will alert [1, 0, 0, 0]
                     //            [0, 1, 0, 0]
                     //            [0, 0, 1, 0]
                     //            [0, 0, 0, 1]

}

</script>
</head>
<body onload="testSylvesterJsLibrary();">
  Simple web page to test Sylvester JavaScript library. <br />
  The page shows a couple of alerts with the result
```

```
    from vector and matrix maths. <br />
    You need to read the source code with view source or or similar
    to understand the results of the alerts.
  </body>
  </html>
```

如果使用 Chrome 开发人员工具(或 Firebug)，并在程序清单 4-1 中的矩阵创建和初始化语句后面设置一个断点，就会看到 Sylvester 中的矩阵是用 JavaScript 中的嵌套数组表示的，如图 4-1 所示。

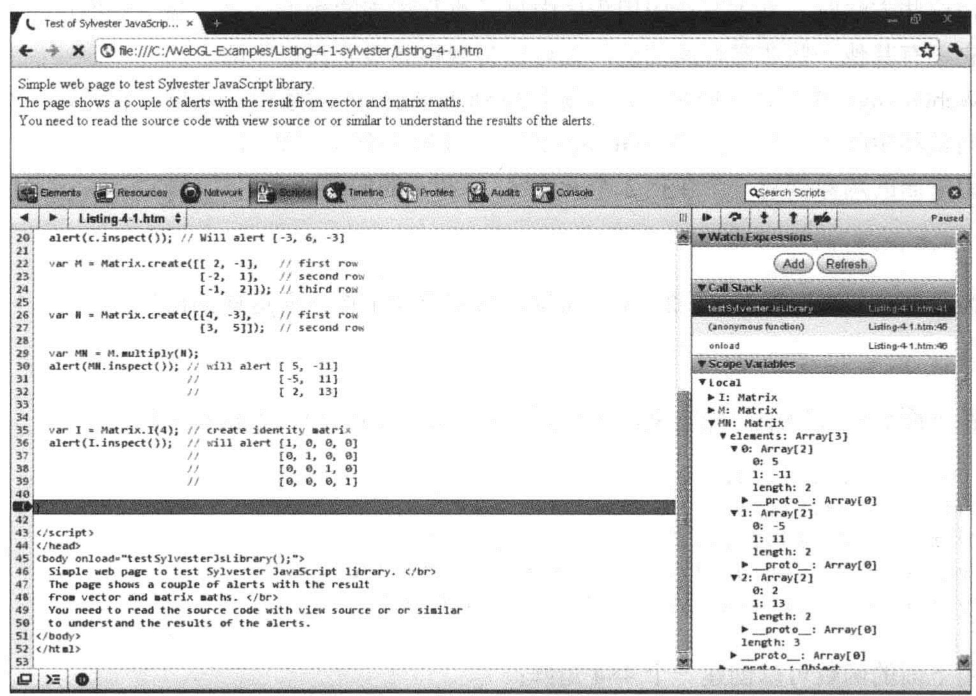


图 4-1　在 Sylvester 库中，矩阵是用嵌套数组表示的

例如，如果查看程序清单 4-1 中的 MN 矩阵，就会发现它的外层数组有 3 个元素，这是因为这个矩阵有 3 行。每一行由两个元素的内层数组表示，这是因为这个矩阵有两列。

以下是几个有关 Sylvester 的要点：

- Sylvester 不是专为 WebGL 开发的，因此当我们在 WebGL 中使用这个库时，必须与其他小型库一起使用，或者自己为 WebGL 特有的功能编写几个函数，这些功能建立投影矩阵或视图变换。
- Sylvester 使用面向对象的范例。
- Sylvester 支持向量、矩阵、线和平面。
- 在 Sylvester 中，矩阵表示为嵌套的 JavaScript 数组。这意味着，它们与线性代数中矩阵的一般表示法非常接近。相反，嵌套的 JavaScript 数组需要转换为一维数组，然后才可以通过 WebGL API 传递给顶点着色器。

4.1.2 WebGL-mjs 库

WebGL-mjs 主要是由 Vladimir Vuićević 开发的。Vladimir Vuićević 不仅因为这个精心设计的库而出名，而且因为他对 WebGL 技术的贡献而出名。与 Sylvester 不同，WebGL-mjs 是专为 WebGL 设计的，它只能处理 4×4 矩阵和 3 个分量的向量。

由于 WebGL-mjs 是专为 WebGL 设计的，因此它也包含了几个非常有用的方法(即 makeScale()、makeTranslate()和 makeRotate())，用户通过这几个方法很容易实现变换运算。当我们使用这些方法时，不需要在应用程序中显式执行矩阵的相乘运算。除了变换运算的功能外，它还有其他一些非常好的功能，如可以为视锥体设置投影矩阵。

WebGL-mjs 并没有将向量和矩阵等集成到 JavaScript 对象中。相反，它通过一组函数提供向量和矩阵的运算。在 WebGL-mjs 库中，矩阵实际上用数组表示。

用下面的函数创建一个包括 3 个元素 x、y 和 z 的向量：

```
V3.$(x,y,z)
```

为了创建一个分量为 1、2、3 的向量，只需要使用像下面这样的语句：

```
var v = V3.$(1,2,3)
```

在下面这段代码中，先定义两个向量，然后求它们的和、点积和叉积；

```
var u = V3.$(1,2,3);
var v = V3.$(4,5,6);
var s = V3.add(u,v);     // s = [5,7,9]
var d = V3.dot(u,v);     // d = 1*4+2*5+3*6 = 32
var c = V3.cross(u,v);   // c = [-3,6,-3]
```

用下面的函数可以创建一个 4×4 矩阵：

```
M4x4.$(m00, m01, m02, m03, // first column
       m04, m05, m06, m07, // second column
       m08, m09, m10, m11, // third column
       m12, m13, m14, m15) // fourth column
```

这里，m00..m03 是这个矩阵中第一列的元素，m04..m07 是第二列的元素，依此类推。

当矩阵元素按列相加和保存时，这样的矩阵通常称为列优先矩阵。

需要注意它与性线代数中矩阵排列顺序的区别，还要注意它与 Sylvester 中矩阵表示法的不同，后者通过嵌套数组定义矩阵。

假设有两个矩阵 M 和 N，则它们的积为：

```
var MN  = M4x4.mul(M,N);
```

程序清单 4-2 是一个完整的示例，它用 WebGL-mjs 库创建向量和矩阵以及基于它们进

行运算。从这个示例和前面的代码片段可以看出，WebGL-mjs 的函数总是通过其参数接受向量和矩阵，并对这些参数进行运算。与 Sylvester 相比，WebGL-mjs 的表示方法不那么面向对象。

可从
Wrox.com
下载源代码

程序清单 4-2　用 WebGL-mjs JavaScript 库实现向量和矩阵的数学运算

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Test of WebGL-mjs JavaScript Library</title>
<meta charset="utf-8">
<script type="text/javascript" src="mjs.js"></script>
<script type="text/javascript">

function convertToString(array) {
   var stringRep = '';
   for (var i=0; i<array.length; i++) {
     var stringRep = stringRep + array[i]+' ';
   }
   return stringRep;
}

function testMjsJsLibrary() {
   var u = V3.$(1,2,3);
   var v = V3.$(4,5,6);

   var s = V3.add(u,v);          // s = [5,7,9]
   alert(convertToString(s));    // will alert [5,7,9]

   var d = V3.dot(u,v);     // d = 1*4+2*5+3*6=32
   alert(d);                // will alert 32

   var c = V3.cross(u,v);
   alert(convertToString(c));    //will alert [-3,6,-3]

   var M = M4x4.$(1,0,0,0,       // first column
                  0,1,0,0,       // second column
                  0,0,1,0,       // third column
                  2,3,4,1);      // fourth column

   var I = M4x4.$(1,0,0,0,       // first column
                  0,1,0,0,       // second column
                  0,0,1,0,       // third column
                  0,0,0,1);      // fourth column

   var MI = M4x4.mul(M,I);
   alert(convertToString(MI)); // will alert 1,0,0,0
                               //            0,1,0,0,
                               //            0,0,1,0,
                               //            2,3,4,1
```

```
    var T = M4x4.makeTranslate3(2,3,4);
    alert(convertToString(T));  // will alert 1,0,0,0
                                //            0,1,0,0,
                                //            0,0,1,0,
                                //            2,3,4,1
}

</script>
</head>

<body onload="testMjsJsLibrary();">
    Simple web page to test WebGL-mjs JavaScript library. <br />
    The page shows a couple of alerts with the result
    from vector and matrix maths. <br />
    You need to read the source code with view source or or similar
    to understand the results of the alerts.
</body>

</html>
```

图 4-2 说明了用 Chrome 开发人员工具在程序清单 4-2 中矩阵创建和初始化代码后的位置设置一个断点所看到的结果。我们发现，在 WebGL-mjs 中一个矩阵表示为 16 个元素的、类型为 Float32Array 的数组，元素的索引为 0~15。如果仔细分析程序清单 4-2 中矩阵 T 的 Float32Array 表示，就会明白，索引为 13、14、15 的元素分别对应为 x 方向、y 方向、z 方向的平移。

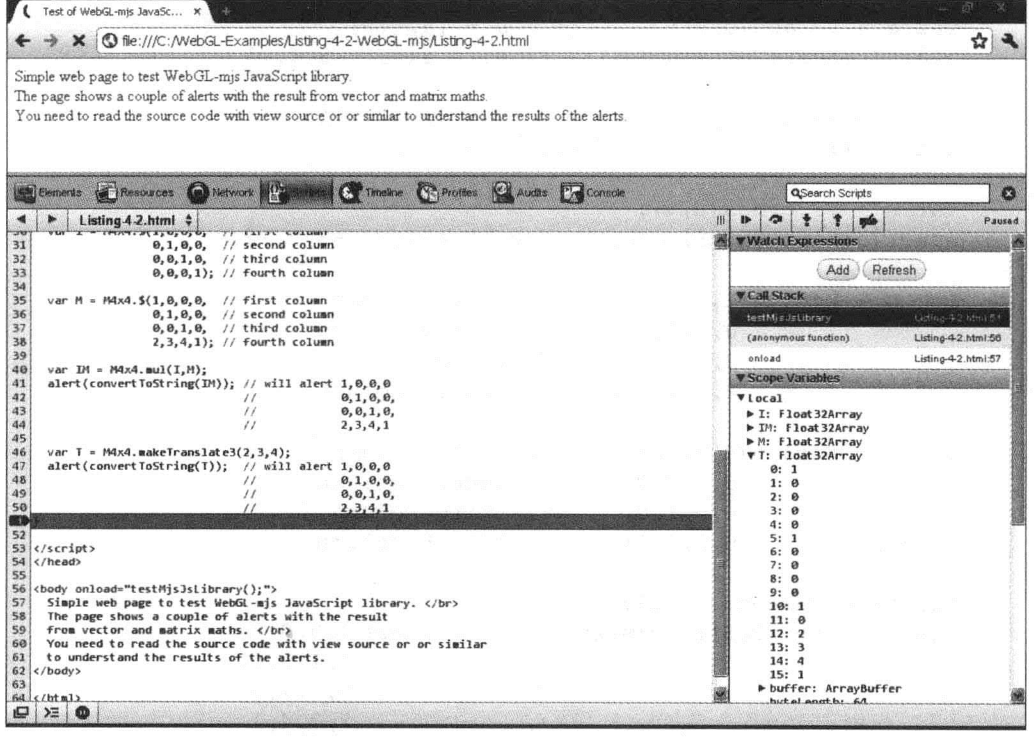


图 4-2　在 WebGL-mjs 中，矩阵表示为 16 个元素的 Float32Array 数组

以下是几个有关 WebGL-mjs 的要点：

- WebGL-mjs 是专为 WebGL 设计的，因此它有一些非常好的功能，如变换运算、投影矩阵和视图变换。
- WebGL-mjs 不采用面向对象模式，函数需要处理的向量和矩阵要通过参数传递给它。
- WebGL-mjs 只能处理 4×4 矩阵和 3 个分量的向量。
- WebGL 中的矩阵表示 16 个元素的 Float32Array 数组。其中前 4 个元素代表矩阵的第 1 列，接着 4 个元素代表第 2 列，其他元素依此类推。
- 由于矩阵采用 Float32Array 表示，因此可以直接通过 WebGL API 向顶点着色器传递矩阵。

4.1.3　glMatrix 库

JavaScript 库 glMatrix 是由 Brandon Jone 开发的。它主要是为 WebGL 设计的，实际上它与 WebGL-mjs 十分相似。它支持 3 个元素的向量和 3×3、4×4 矩阵。但是，在函数参数的定义和可选参数的处理上它们存在一些差别。在分析这些可选参数之前，先理解一个基本运算。为了创建 3 个元素的向量，要用下面的函数：

```
vec3.create(vec)
```

这里 vec 是一个可选的数组，它包含 3 个初始元素。这意味着，如果创建一个元素为 1、2、3 的 3 分量向量，则要编写下面的代码：

```
var u = vec3.create([1,2,3]);
```

也可以不用初始值直接创建一个 3 分量向量：

```
var u=vec3.create();
```

glMatrix 库中大多数执行向量和矩阵运算的函数都有一个参数表示此运算针对向量还是矩阵。作用于向量的函数采用以下形式：

```
vec3.opeation(srcVec,otherOperands,destVec(optional));
```

如果指定了 destVec 参数，则运算结果写入 destVec 中，并将它作为函数值返回。然而，如果没有指定 destVec 参数，则运算的结果写入 srcVec 中并返回。这意味着，如果没有指定 destVec，则 srcVec 的内容会被修改。因此，如果我们本来不希望 srcVec 被改变，则这会产生错误。

在下面的代码段中，我们可以看出当指定可选目标参数(即 destVec)后函数的工作方式。首先定义并初始化两个向量 u 和 v，然后建立第三个向量，用来保存前两个向量的相加结果。由于第三个向量只用来保存运算结果，因此不需要给它赋初始值。最后，对这两个向量进行相加运算，并将结果写入目标向量中。

```
var u = vec3.create([1,2,3]);
var v = vec3.create([4,5,6]);
var s = vec3.create();
vec3.add(u,v,s);            // s = [5,7,9] and u is unchanged
```

由于指定了可选目标向量，因此运算结果保存到这个向量中，并且只有这个参数受到影响。

现在分析下面这段代码，看看不指定目标向量会有什么结果：

```
var u = vec3.create([1,2,3]);
var v = vec3.create([4,5,6]);
var s = vec3.add(u,v); // s = [5,7,9] and u = [5,7,9]
```

因此，如果在使用 glMatrix 库时没有指定可选的目标向量，则结果写入第一个操作数中。

与向量一样，矩阵的运算基本上也遵循同样的模式。调用下面的函数可以创建一个 4×4 的矩阵：

```
mat4.create(mat)
```

其中，mat 是一个可选数组，它包含了这个矩阵的 16 个初始元素。与 WebGL-mjs 一样，前 4 个元素对应此矩阵的第一列，接下来 4 个元素对应于矩阵的第二列，依此类推。如果需要指定可选的初始数组，则此函数的调用代码如下所示：

```
var N = mat4.create([0,1,2,3,    // first column
                     4,5,6,7,    // second column
                     8,9,0,1,    // third column
                     2,3,4,5]);  // fourth column
```

假设有两个矩阵 M 和 N，要求出它们的积 MN，则需要调用以下代码：

```
var MN = mat4.multiply(M,N);
```

需要注意的是，在相乘运算中没有指定可选的目标矩阵，因此相乘结果写入 M 矩阵中。

程序清单 4-3 是一个完整示例的源代码，说明如何使用这个 glMatrix JavaScript 库。除了通常的向量和矩阵运算外，这个示例还使用 vec3.str()和 mat4.str()函数，它们将一个向量或矩阵转换为一个字符串。同时也需要注意在程序清单 4-3 末尾如何调用 mat4.translate()函数。它创建一个平移变换矩阵，这个平移变换沿 x 轴方向移动 2 个单位，y 轴方向移动 3 个单位，z 轴方向移动 4 个单位。表示平移值的元素的索引分别为 13、14、15。

程序清单 4-3　用 glMatrix JavaScript 库实现向量和矩阵的数学运算

可从 Wrox.com 下载源代码

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Test of glMatrix JavaScript Library</title>
<script type="text/javascript" src="glMatrix.js"></script>
<script type="text/javascript">
```

```
function testglMatrixJsLibrary() {
var u = vec3.create([1,2,3]);
var v = vec3.create([4,5,6]);
var r = vec3.create();
var t = vec3.create([1,2,3]);
// In the add below the optional receiver vec3 is specified as 3:rd
// argument so the first two arguments are not modified
var s = vec3.add(u,v,r); // s = r = [5,7,9] as expected, u not changed
alert(vec3.str(s)); // will alert [5,7,9]
alert(vec3.str(r)); // will alert [5,7,9]
alert(vec3.str(u)); // will alert [1,2,3]
// In the add below the optional receiver vec3 is not specified
// so the the result is written to the first argument u in addition to s2
var s2 = vec3.add(u,v) // s2 = [5,7,9] as expected,
// but now also u = [5,7,9]
alert(vec3.str(s2)); // will alert [5,7,9]
alert(vec3.str(u)); // will alert [5,7,9]
var d = vec3.dot(t,v); // d = 1*4+2*5+3*6 = 32
alert(d); // will alert 32
var c = vec3.cross(t,v,r); // c = r = [-3,6,-3]
alert(vec3.str(r)); // will alert [-3,6,-3]
var I = mat4.create([1,0,0,0, // first column
0,1,0,0, // second column
0,0,1,0, // third column
0,0,0,1]); // fourth column
var M = mat4.create([1,0,0,0, // first column
0,1,0,0, // second column
0,0,1,0, // third column
2,3,4,1]); // fourth column
var IM = mat4.create();
mat4.multiply(I, M, IM);
alert(mat4.str(IM)); // will alert [1,0,0,0
// 0,1,0,0,
// 0,0,1,0,
// 2,3,4,1]
var T = mat4.create();
mat4.translate(I, [2,3,4],T);
alert(mat4.str(T)); // will alert [1,0,0,0
// 0,1,0,0,
// 0,0,1,0,
// 2,3,4,1]
}
</script>
</head>
<body onload="testglMatrixJsLibrary();">
Simple web page to test glMatrix JavaScript library. <br />
The page shows a couple of alerts with the result
from vector and matrix maths. <br />
You need to read the source code with view source or or similar
to understand the results of the alerts.
</body>
</html>
```

图 4-3 说明了在程序清单 3-4 的矩阵创建和初始化代码之后设置一个断点时会看到的结果。与 WebGL-mjs 一样，你也会看到，glMatrix 的矩阵表示为 16 个元素的 Float32Array 数组，元素的索引为 0~15。如果仔细分析程序清单 4-3 中代表矩阵 T 的 Float32Array 数组，就会明白这个数组中索引为 13、14 和 15 的元素分别对应于 x 轴、y 轴和 z 轴的平移值。

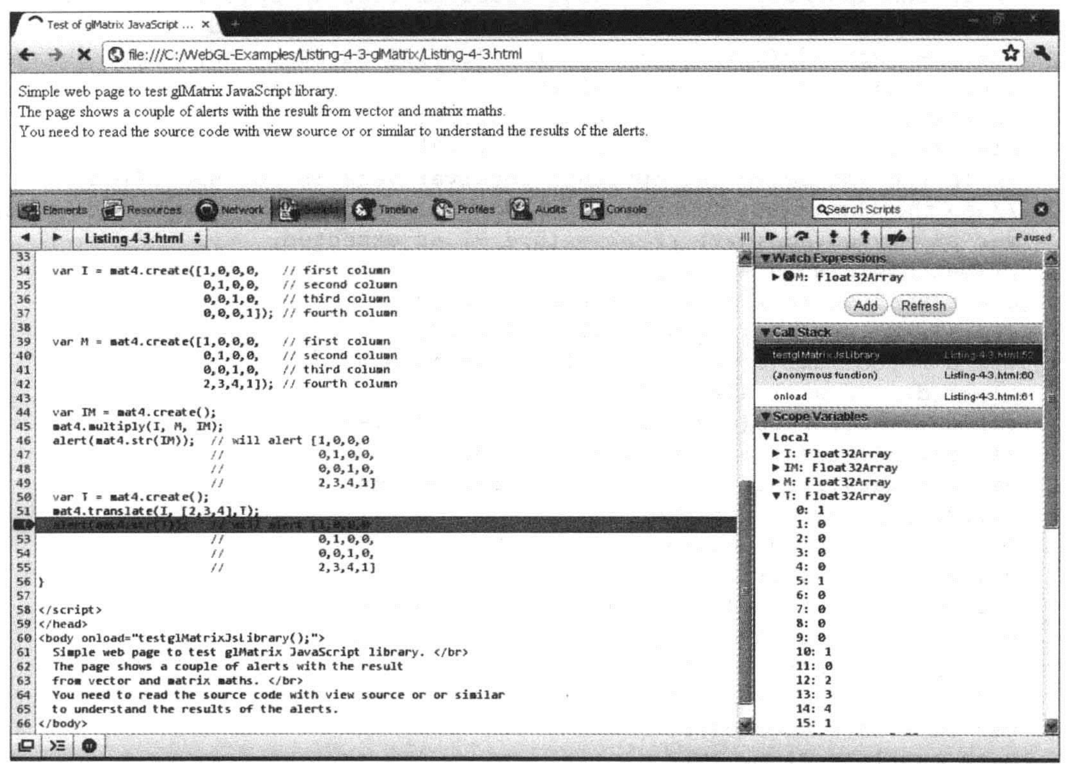


图 4-3　在 glMatrix 中，矩阵表示为包含 16 个元素的 Float32Array 数组

以下是几个有关 glMatrix 的要点：

- 与 WebGL-mjs 一样，glMatrix 也是专为 WebGL 设计的，因此它有一些非常优秀的功能，如执行变换运算、设置投影矩阵、执行视图变换。
- glMatrix 也没有采用面向对象范例，而函数需要处理的向量和矩阵是通过参数传入的。
- WebGL-mjs 可以处理 3×3、4×4 矩阵和 3 个分量的向量。
- 如果没有指定 WebGL 函数的最后一个可选参数，则 glMatrix 函数总是修改第一个参数的值。
- 在 WebGL-mjs 中，矩阵用 16 个元素的 Float32Array 数组来表示。其中后者的前 4 个元素代表矩阵的第一列，接着 4 个元素代表矩阵的第二列，依此类推。
- 采用包含 16 个元素的 Float32Array 数组表示矩阵后，就可以通过 WebGL API 直接向顶点着色器传递矩阵。

你已经对这 3 个小型库有一个初步的了解，因此现在就可以从中选择一个库作为自己开发 WebGL 项目的基础。也许你可能会找到一个这里没有介绍的更好的库。如果本书中的其余示例需要这种类型的小型库，就选择使用 glMatrix 库。

4.2　变换运算

在第 1 章中，我们已经介绍了变换的一些线性代数理论。前面曾提到，掌握这些基本知识是非常重要的。然而，在前几节中介绍 JavaScript 库时你可能已经注意到，在 WebGL 代码中执行旋转、平移和缩放等操作实际上不需要手动执行矩阵相乘运算。如果你愿意，可以使用 JavaScript 库。如果你愿意，可以以第 1 章的线性代数理论为基础自己动手设计一个 JavaScript 库。

有时，在 WebGL 中执行变换操作的最麻烦部分不是执行矩阵相乘运算。实际上，大多数 3D 图形的新手总认为最困难的是为了定位场景中的对象和照相机的位置，需要执行一系列的不同变换。但是你不要担心，在以后几节中将学到有关变换的基本操作。

变换的使用方式

第 3 章的程序清单 3-2 说明了如何在屏幕上绘制一个六边形、一个三角形和一个三角形带。在这个示例中，顶点着色器并没有执行任何变换操作，因此载入 WebGLBuffer 对象的顶点坐标必须与它们在视口中的最终坐标相对应。更具体一些，所有保存在 WebGLBuffer 对象中的坐标都必须是裁剪坐标，因为它们正是顶点着色器在结束处理后其内置变量 gl_Position 所需要的数据。

在 WebGL 以及一般的 3D 图形中，一种比较聪明的做法是在用户坐标系中建立模型，然后经过一系列不同的变换，将它们正确地放置在一个世界坐标系中(正确的位置和方向)。在 WebGL 中，我们经常提到以下类型的变换：

- 模型变换
- 视图变换
- 模型视图变换
- 投影变换
- 透视除法(perspective division)
- 视口变换

在最低级别，前述的前 4 个变换通常可以通过顶点着色器中的矩阵相乘运算实现。透视除法和视口变换稍微有点不同。这两个变换不属于矩阵相乘运算，也不是由顶点着色器来处理的，而是由 WebGL 流水线的图元装配阶段来处理的(有关 WebGL 流水线的图元装配阶段的更多信息，请阅读第 1 章的内容)。

1. 模型变换

通过 WebGL API 传入且保存在 WebGLBuffer 对象中的顶点初始坐标称为对象坐标。

模型变换用来确定模型在世界坐标系统中的位置和方向。3D 场景中的每个对象都有自己的模型变换。当所有的模型都变换到世界坐标系中后，它们就共存于同一个坐标系中。模型变换通常由以下变换组成：

- 平移
- 旋转
- 缩放

如果仔细查看 JavaScript 库 glMatrix 中的功能，就会发现这些变换对应于以下函数：

- mat4.translate()
- mat4.rotate()
- mat4.scale()

假设我们用 WebGL 编写一个 3D 游戏，场景中有一艘飞船。我们首先在对象坐标系中构建这个飞船。然后利用模型变换将它平移到世界坐标系中的某个位置，再将它旋转到正确的方向，最后可能还需要对它应用缩放变换，使它的大小适合于这个游戏。

幸运的是，只需要一个 4×4 矩阵就可以保存平移、旋转和缩放的全部组合。这意味着，为了将对象的顶点从对象坐标系变换到世界坐标系，只需要将顶点坐标乘上这个 4×4 矩阵。在顶点着色器中很容易实现矩阵相乘运算。

2. 视图变换

在 3D 场景中，WebGL 只对照相机(或称观察者)能够看到的对象进行绘制。视图变换用来确定对象在“虚拟”照相机中的位置和方向。这里之所以使用虚拟一词，是因为 WebGL 并没有显式方法用来移动场景中的照相机。照相机总是位于原点位置，而且视线朝 z 轴负方向。视图变换的作用实际上是设置一个矩阵，它对场景中顶点的变换效果正好与变换照相机的效果相反。阅读了下一节内容后，你才会对此有更好的理解。

视图变换通常由以下变换组成：

- 平移
- 旋转

从前一节中可以知道，这些变换可以由以下的 glMatrix 函数来实现：

- mat4.translate()
- mat4.rotate();

顶点经过视图变换后，它们就处在眼睛坐标系中，因此就称它们为眼睛坐标。

设置视图变换矩阵的一种策略是 mat4.translate()函数和 mat4.rotate()函数的调用组合。另一个经常使用的比较简单的策略是调用 mat4.lookAt()实用工具函数。这个函数根据传入的参数(即输入照相机的位置、视线方向和一个用来定义“向上方向”的向量)创建一个视图变换矩阵。在后面几个示例中我们将看到 mat4.lookAt()函数的用法。

3. 模型视图变换

在讨论最终场景在屏幕上的显示时，模型变换和视图变换实际上是同一回事。重要的是场景中的照相机与对象之间的关系。想象一下，假设有一个对象和一个照相机都位于原点位置。这个照相机的视线沿着 z 轴负方向。为了看到对象，有两种方法可供选择：

- 将照相机向后移动——由于默认时照相机是朝 z 轴负方向的，因此照相机向后移动实际上就是沿正 z 轴移动。
- 将对象向前移动到场景中——这是指将对象沿着 z 轴负方向移动。

这两种方法实际上都是同一回事，它们只是思考方式不一样。将照相机沿着一个方向移动，或者将所有对象沿着相反方向移动。

假设当前照相机和对象都位于原点，但是我们希望照相机与对象之间有 10 单位的距离。不管我们从哪个角度思考这个问题，结果都是用一个矩阵乘上这个对象的顶点。在这个矩阵中，在 z 轴方向的平移值为-10。利用 glMatrix 这个 JavaScript 库，可以建立这个矩阵：

```
mat4.translate(modelViewMatrix,[0,0,-10],modelViewMatrix);
```

模型视图变换这个术语是指，模型变换和视图变换已合并成一个模型视图矩阵。在编写 WebGL 应用程序时，我们很容易会将模型变换和视图变换看成是两个独立的变换，但是实际上，它们可以合并成模型视图矩阵。当我们给顶点乘上一个合并后的模型视图矩阵时，它表示顶点直接从对象坐标系转换为眼睛坐标系。图 4-4 说明了模型变换与视图变换如何合并成模型视图变换。

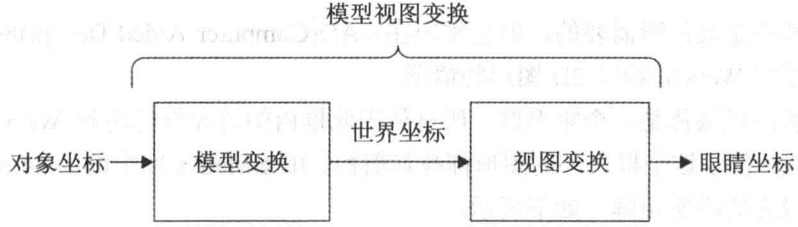


图 4-4　模型视图变换是模型变换和视图变换的合并

沿着某一个方向移动虚拟的照相机本质上相当于沿着相反方向移动全部对象。OpenGL ES 1.x 和桌面 OpenGL 的固定函数版本并没有单独的模型矩阵和单独的视图矩阵。事实上，它们提供了一个合并的模型视图矩阵。

WebGL 和 OpenGL ES 2.0 都是基于着色器的，用户可以自己决定使用哪个矩阵，以及如何在顶点着色器中应用变换。但是，通常使用合并的模型视图矩阵比较容易。

4. 投影变换

投影变换应用在模型视图变换之后。利用它可以确定如何将 3D 场景投影到屏幕上。

投影变换也决定了视域体的外观。投影变换实际上是将视域体变换成一个单位立方体，这个单位立方体在 3 个坐标轴上的取值范围都是-1~+1。

gl_Position 变量是众多来自顶点着色器的输出之一，它要求所有显示在屏幕上的顶点都要位于这个单位立方体内。为了能够处理位于此单位立方体外的坐标，需要利用顶点着色器中的一个投影矩阵对顶点的位置进行变换，使得它们在写入 gl_Position 变量之前都位于此单位立方体内。位于此单位立方体之外的顶点和图元都要被裁剪掉。

在至此介绍的所有示例中还没有用到投影变换，因此用户不得不将自己的图形对象设置为在这个单位立方体内才可以显示在屏幕上。

在 3D 图形中经常使用一个类比，即把变换作用与用照相机拍摄部分对象的过程相类比。在这种情形下，模型变换相当于在场景中定位对象，视图变换相当于定位和对准照相机，投影变换相当于为照相机选择一个镜头。镜头决定了视野，某种程度上也决定了对象的效果。有两类投影变换，它们是：

- 正交投影
- 透视投影

这两个投影都会在后面几节中得到论述。

正交投影

正交投影有时也称为平行投影，这种投影的一个属性是平行线经过投影后仍然平行。这种投影的一个重要特性是它不会因对象离观察者远或近而影响对象的大小。场景中的所有对象经过投影都保持它们之间的相对大小。当我们需要一个场景看起来有真实感时，则这种投影通常不是我们所需要的，但它常用在 CAD(Computer Aided Design)应用程序中，或者用于想要用 WebGL 绘制 2D 图形的情形。

正交投影的视域体是一个矩形框。所有位于此框内的图元都传递到 WebGL 流水线中的下一个阶段，位于这个框之外的图形都被裁剪掉。用 glMatrix 库中的 mat4.ortho()函数可以设置正交投影的投影矩阵，如下所示：

```
mat4.ortho(left,right,bottom,top,near,far,projectionMatrix);
```

这个函数创建一个投影矩阵，它相当于如图 4-5 所示的长方体视域体。这个视域体的左边界由 left 参数确定，右边界由 right 参数决定，底边界由 bottom 参数决定，顶边界由 top 参数决定，近平面由 near 参数决定，远平面由 far 参数决定。观察者在这个长方体的正前方的无限远处。创建得到的投影矩阵保存在最后一个参数(即 projectionMatrix)中。

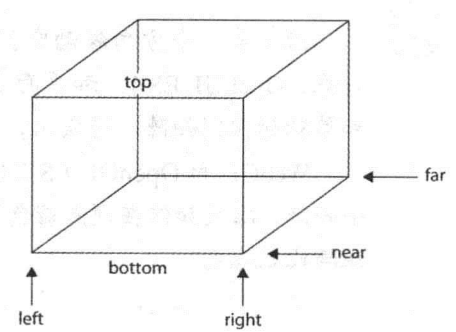


图 4-5　正交投影的视域体

透视投影

当我们使用透视投影时，离观察者较远的对象看起来比较小，而离观察者较近的对象

看起来比较大。与平行投影相比，这种投影可以提供比较真实的场景，因为它比较接近于人眼的视觉原理。这种投影的视域体是一个锥台(因此称为视锥体，它相当于去掉顶部的金字塔)。

在 WebGL 中，有两个函数可以用来设置透视投影的矩阵：

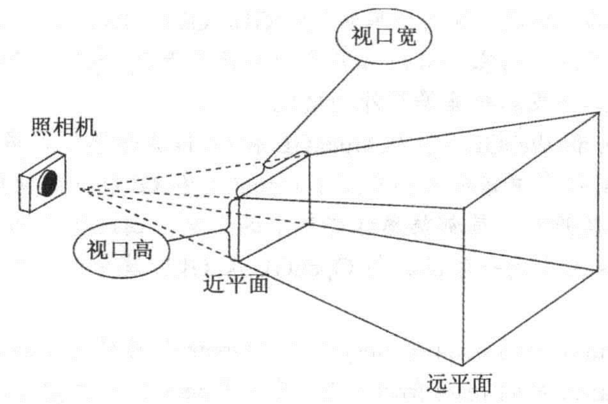
- mat4.perspective()
- mat4.frustum()

第一个函数 mat4.perspective()的原型如下：

```
mat4.perspective(fvoy,aspect,near,far,projectionMatrix);
```

这里的 fvoy 表示视域的垂直范围，ascpect 是纵横比(视口的宽/视口的高)，纵横比用来确定视域的水平范围。参数 near 是视域体的近平面离观察者的距离，参数 far 是视域体的远平面离观察者的距离。调用此方法得到的结果是一个投影矩阵，它保存在最后一个参数 projectionMatrix 中。

图 4-6 说明一个透视投影的视域体，其下方是视域体的侧面视图。从图中可以清楚看出视域垂直范围的定义。



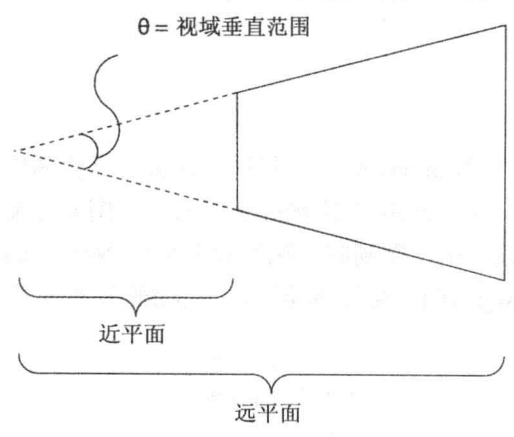


图 4-6　透视投影的视域体

我们也可以用第二个函数 mat4.frustum()定义一个透视投影矩阵，如下所示：

```
mat4.frustum(left,right,bottom,top,near,far,projectionMatrix);
```

这些参数的含义如下：

- left 是近平面的左边界。
- right 是近平面的右边界。
- bottom 是近平面的底边界。
- top 是近平面的顶边界。
- near 是近平面的位置。
- far 是远平面的位置。
- projectionMatrix 设置操作的目标，即生成的投影矩阵。

mat4.frustum()函数比 mat4.perspective()函数更加通用，而且在 glMatrix 库(以及其他几个有此方法的库)中，mat4.perspective()是由 mat4.frustum()来实现的。

在 glMatrix 和 WebGL-mjs JavaScript 库中，许多函数并不是由这些库的开发人员发明的。相反，这两个库从OpenGL、GLU(OpenGL实用工具库)等其他图形库借鉴了许多内容。GLU 库是专为补充桌面OpenGL库而设计的，它提供创建视图矩阵和投影矩阵等额外的功能。

glMatrix 和 WebGL-mjs 从 OpenGL 和 GLU 库借鉴了许多内容，这当然是一件好事。有很多书籍和文档介绍了 OpenGL 和 GLU 库的使用。3D 图形论坛中的绝大多数开发人员都熟悉这些库中的函数，因此很容易找到所需要的帮助。glMatrix 库中的一些函数与 OpenGL 及 GLU 库中的相应函数的对应关系如下：

- glMatrix 中的 mat4.rotate()对应于 OpenGL 库的 glRotate()。
- glMatrix 中的 mat4.frustum()对应于 OpenGL 库的 glFrustum()。
- glMatrix 中的 mat4.perspective()对应于 GLU 库的 gluPerspective()。
- glMatrix 中的 mat4.lookAt()对应于 GLU 库中的 gluLookAt()。

5. 透视除法

当顶点着色器将坐标写入 gl_Position 变量时，在裁剪坐标系中进行该操作，而且它们是用包含 4 个分量(x_c、y_c、z_c、w_c)的齐次坐标表示的。在图元装配期间，顶点要经过透视除法，即将所有的坐标除以 w_c，得到归一化的设备坐标(Normalized Device Coordinate，NDC)。裁剪坐标(x_c,y_c,z_c,w_c)与归一化设备坐标(x_c,y_d,z_d)的关系可以表示为：

$$\begin{bmatrix} x_d \\ y_d \\ z_d \end{bmatrix} = \begin{bmatrix} x_c / w_c \\ y_c / w_c \\ z_c / w_c \end{bmatrix}$$

可以看出，这一步并没有采用矩阵相乘运算。此外，它也不同于前面的步骤，因为这不是一个用户能显式影响结果的操作。第四个裁剪坐标 w_c 通常为 1，通常在代码中不需要修改。

6. 视口变换

与透视除法一样，视口变换也没有采用矩阵相乘运算。虽然视口变换是图元装配操作的一部分，但是用户可以调用以下方法影响视口变换：

```
gl.viewPort(x, y, w, h);
gl.depthRange(n,f);
```

你在前面已经看到过 gl.viewPort()方法的调用语句，它用视口左下角位置坐标(x,y)、宽度(w)和高度(h)定义一个视口。

gl.depthRange()方法用来定义用户希望达到的景深范围。这里的参数 n 代表近平面，f 代表远平面。n 和 f 值压缩为在范围[0.0,1.0]内，而且不允许将 n 设置为大于 f 的值，否则 WebGL 实现将产生一个 gl.INVALID_OPERATION 错误。

此外还要注意没有调用这个方法时景深的默认值。默认情况下，n 的值为 0.0，f 的值为 1.0。

从归一化设备坐标(x_d,y_d,z_d)变换为窗口坐标(x_w,y_w,z_w)要使用下面的公式：

$$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = \begin{bmatrix} (w/2)x_d + o_x \\ (h/2)y_d + o_y \\ ((f-n)/2)z_d + (f+n)/2 \end{bmatrix}$$

这里的 w 和 h 代表视口的宽度和高度，单位为像素，用于 gl.viewPort()方法。o_x 和 o_y 代表视口的中心坐标，它们定义为 $o_x=(x+w)/2$ 和 $o_y=(y+h)/2$。

4.3　理解完整的变换流水线

至此你对模型变换、视图变换、模型视图变换、投影等重要概念已经有一个基本的理解，因此现在就要讨论整个 WebGL 变换流水线的处理过程。图 4-7 说明了整个变换流水线的结构。

前面曾提到，WebGL 是完全基于着色器的，这意味着用户可以完全控制顶点着色器中的任何一个变换矩阵，至少在理论上是如此。然而，如图 4-7 所示，实际情形是需要分解模型视图矩阵，并且经常用到一个投影矩阵。

虽然根据前面几节的论述，你已经大概掌握了这个流水线的处理过程，但是这一节还是会对它作全面的介绍，确保你能够完全理解它的处理过程。

在图 4-7 的左侧，用对象坐标表示的顶点保存在 WebGLBuffer 对象中，顶点着色器从这个对象读取数据，并将顶点坐标乘上模型视图矩阵。乘上模型视图矩阵后，顶点处于眼

睛坐标中。如果需要在顶点着色器中执行光照处理，则通常是在眼睛坐标中进行。

在顶点着色器里完成

对象坐标(来自于 WebGLBuffer 对象)

x y z w

模型视图矩阵

眼睛坐标

投影矩阵

裁剪坐标(写入 gl_Position)

在图元装配阶段完成

模型视图矩阵

归一化设备坐标

投影矩阵

窗口坐标

图 4-7　变换流水线

顶点着色器然后将顶点坐标乘上投影矩阵，得到裁剪坐标。顶点着色器将顶点坐标写入 gl_Position 变量中，gl_Position 变量的值代表裁剪坐标。

在顶点着色器将顶点写入 gl_Position 后，顶点要经过透视除法，即用第四分量 w_c 除以裁剪坐标(x_c,y_c,z_c,w_c)得到归一化设备坐标。最后，归一化设备坐标映射通过视口变换映射到实际屏幕坐标。

4.4 变换的实践

希望你已经对各个变换以及它们在 WebGL 中的应用有一个基本的理解。你已经看到，JavaScript 库 glMatrix 中的一些函数可以用来创建变换矩阵，此外你也掌握了整个变换流水线的处理过程。现在从更加实用的角度说明整个变换链的实际应用。下面从更高层次分析这个变换链的过程：

(1) 创建 WebGLBuffer 对象，保存场景中对象的对象坐标。

(2) 在调用任何绘图方法之前，创建模型视图矩阵和投影矩阵，通常是使用 JavaScript 库提供的函数。

(3) 在 JavaScript 代码中创建的变换矩阵需要上传到 GPU 中的顶点着色器。通常的做法是在顶点着色器中用 gl.uniformMatrix4fv()方法定义一个统一体。

(4) 调用 gl.drawArrays()或 gl.drawElements()方法绘制场景。

(5) 现在为场景中的每个顶点执行顶点着色器。顶点着色器从 WebGLBuffer 对象获得数据，WebGLBuffer 对象保存了对象坐标。顶点着色器引用统一体(它保存了上传的变换矩阵)，并进行矩阵相乘操作，实现变换。

4.4.1　为对象坐标设置缓冲

在将顶点数据赋给缓冲之前，通常需要创建并初始化 WebGL 对象，如创建 WebGL 上下文以及载入、编译和链接顶点着色器。现在你应该已经熟悉这些操作。当用户需要创建 WebGLBuffer 对象时，唯一的好消息是，用户不需要根据它们在最终场景中的放置确定对象顶点坐标。

当我们还没有执行任何变换操作时，对象坐标必须与裁剪坐标相等。这意味着，在 WebGLBuffer 对象中的坐标必须在立方体内，即在[-1,-1,-1]~[1,1,1]范围内以不被裁剪。此外，如果我们想要某个对象在另一个对象的左侧，则在给顶点位置赋值时必须格外小心。

现在，利用变换操作，我们拥有更多的灵活性，而且模型的对象坐标通常总是以原点为中心，不管它们在最终场景中的位置如何。下面这段代码只是缓冲的通常设置方法，对你来说应该比较熟悉：

```
function setupBuffers() {
  cubeVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexPositionBuffer);

  var cubeVertexPosition = [
      // Front face
      1.0, 1.0, 1.0, //v0
     -1.0, 1.0, 1.0, //v1
     -1.0, -1.0, 1.0, //v2
      1.0, -1.0, 1.0, //v3

...

     -1.0, -1.0, -1.0, //v22
     -1.0, -1.0, 1.0, //v23
  ];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(cubeVertexPosition),
                gl.STATIC_DRAW);
  cubeVertexPositionBuffer.itemSize = 3;
  cubeVertexPositionBuffer.numberOfItems = 24;

  cubeVertexIndexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeVertexIndexBuffer);
  var cubeVertexIndices = [
            0, 1, 2, 0, 2, 3, // Front face

            ...

            20, 22, 21, 20, 23, 22 // Bottom face
      ];
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(cubeVertexIndices),
  gl.STATIC_DRAW);
  cubeVertexIndexBuffer.itemSize = 1;
```

```
    cubeVertexIndexBuffer.numberOfItems = 36;
}
```

4.4.2 用 JavaScript 创建变换矩阵并上传给着色器

创建了缓冲后，还需要创建变换矩阵，在调用 gl.drawArrays()或 gl.drawElements()方法执行实际绘图之前需要将它们上传给着色器。第一步通常是用 JavaScript 代码创建模型视图矩阵和投影矩阵，通常这需要调用诸如 glMatrix 这类 JavaScript 库中的函数。下面的代码段示例说明这些操作的步骤：

```
modelViewMatrix = mat4.create();
projectionMatrix = mat4.create();

function draw() {

    ...

    mat4.perspective(60, gl.viewportWidth / gl.viewportHeight,
                     0.1, 100.0, projectionMatrix);
    mat4.identity(modelViewMatrix);
    mat4.lookAt([8, 5, 10],[0, 0, 0], [0, 1, 0], modelViewMatrix);

    mat4.translate(modelViewMatrix, [0.0, 3.0, 0.0], modelViewMatrix);
```

首先，用 mat4.create 方法创建一个 4×4 的投影矩阵，然后用 mat4.perspective()方法建立一个投影矩阵，此投影矩阵的视域体垂直范围为 60°、其纵横比由视口的宽度除以视区的高度决定，近平面位于离观察者 0.1 单位的位置，远平面位于离观察者 100.0 单位的位置。

然后用 mat4.identity()方法将单位矩阵载入模型视图矩阵中。这样做的目的是在将场景的模型视图变换应用于模型视图矩阵之前对它进行初始化。如果不将这个模型视图矩阵设置为单位矩阵，则它的元素值可能是来自前一帧的内容，这很可能不是我们想要的值。

通常，在执行模型变换之前根据模型视图矩阵建立视图变换。在这段代码中，视图变换要用 mat4.lookAt()方法来设置。mat4.lookAt()方法的第一个参数定义了观察者位于(8,5,10)位置。第二个参数定义视图方向是指向原点。第三个参数定义照相机向上的方向为 y 轴正方向。最后调用 mat4.translate()方法，将沿 y 轴正方向移动 3 个单位的平移变换添加到这个模型视图矩阵上。

4.4.3 将变换矩阵上传给 GPU 中的顶点着色器

用 JavaScript 创建了变换矩阵之后，我们必须在将它作用于顶点着色器中的顶点之前将其上传到 GPU 中。这实际上就是用 gl.uniformMatrix4fv()方法将 16 个浮点数传递给一个统一体。下面这段代码说明两个辅助函数，用它们将模型视图矩阵和投影矩阵传递给顶点着色器：

```
function uploadModelViewMatrixToShader() {
  gl.uniformMatrix4fv(shaderProgram.uniformMVMatrix,
```

```
                    false, modelViewMatrix);
}

function uploadProjectionMatrixToShader() {
  gl.uniformMatrix4fv(shaderProgram.uniformProjMatrix,
                    false, projectionMatrix);
}
```

gl.uniformMatrix4fv()方法的第一个参数定义了接收数据的统一体，第二个参数表示是否转置上传的列，在 WebGL 中，这个参数必须设置为 false，否则就会产生 gl.INVALID_VALUE 错误。

在 WebGL 中(在 OpenGL ES 2.0 中也是如此)，gl.uniformMatrix()方法(类似于本节调用的 gl.uniformMatrix4fv()方法)的转置参数必须设置为 false。之所以在这个方法的定义中还包含此参数，是为了与桌面 OpenGL 规范保持一致。不要忘记，WebGL 是以 OpenGL ES 2.0 为基础的，后者则是以桌面 OpenGL 2.0 为基础的。

由于 glMatrix 库将一个矩阵表示为一个包含 16 个元素且按列优先存放的 Float32Array 数组，而 OpenGL ES 着色语言中的 mat4 类型也将矩阵表示为按列存放的数组，因此这两个矩阵表示法完全相符，这个转置参数不应设置为除 false 之外的任何值。最后一个参数是矩阵的实际数据。当我们用 glMatrix 创建和处理矩阵时，这个参数必须是一个包含 16 个元素的 Float32Array 数组。

4.4.4　调用绘图方法

在正确创建了变换矩阵并上传给 GPU 中的顶点着色器之后，现在就可以调用 gl.drawArrays()和 gl.drawElements()方法执行绘图操作。这个步骤应该不是新内容，你应该比较熟悉。

在顶点着色器中执行矩阵相乘运算

调用 gl.drawArrays()或 gl.drawElements()方法时，需要为绑定到 WebGLBufer 对象中的顶点执行顶点着色器。在下面这个代码段中，会看到一个顶点着色器。在这个顶点着色器中，顶点在对象坐标系中的位置保存在 aVertexPosition 属性中。模型视图矩阵保存在名为 uMVMatrix 的统一体(类型为 mat4)中，投影矩阵保存在相同类型的 uPMatrix 统一体中。可以看出顶点如何从对象坐标(在 aVertexPosition 中)变换到裁剪坐标，即将对象坐标乘上模型视图矩阵，再乘上投影矩阵，且最后的裁剪坐标保存在 gl_Position 中：

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;
```

```
    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;

    varying vec4 vColor;

    void main() {
      gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);

      vColor = aVertexColor;
    }

</script>
```

4.5 理解变换顺序的重要性

在第 1 章中学习矩阵相乘运算时，我们知道矩阵相乘的顺序很重要。一般来说，对于矩阵 M 和 N，它们的相乘结果与它们的相乘顺序有关，即通常有：

$$\mathbf{MN} \neq \mathbf{NM}$$

你还需要知道，在顶点着色器中执行的变换操作实际上就是矩阵相乘运算。除了在顶点着色器中执行矩阵相乘运算外，在将变换矩阵上传到着色器之前也需要在 JavaScript 代码中执行矩阵相乘运算。例如，在 JavaScript 中建立模型视图矩阵时需要执行几个变换运算(例如旋转之后执行的变换)，这也属于这种情况。

这意味着，执行变换操作的顺序非常重要。考虑平移变换和旋转变换，这两个变换对于模型视图变换非常重要。先执行平移变换后执行旋转变换，这与先执行旋转变换后执行平移变换的结果完全不一样。当需要执行一系列变换时，有两种不同的方法看待这些变换：

- 使用一个固定的全局的坐标系。
- 使用一个移动的局部的坐标系。

它们就是思考这些变换的两种不同方法。不管我们使用哪种方法，最终的结果都是一样的。在 4.5.1 节中，我们讨论如何用固定的全局的坐标系看待这些变换；在 4.5.2 节中，我们讨论如何用一个移动的局部的坐标系看待这些变换。

4.5.1 使用一个固定的全局的坐标系

一个固定的全局的坐标系表示当我们执行各个变换时，这个坐标系是固定的，不会移动。这样的坐标系正是我们通常所需要的。为了理解使用固定的全局的坐标系的这种思维方式，我们分析执行一系列变换操作时矩阵相乘运算的细节。

作为一个示例，假设我们正在使用 JavaScript 库 glMatrix，在下面的代码段中用 mat4.rotate()方法执行旋转变换，用 mat4.translate()方法执行平移变换：

```
mat4.identity(modelViewMatrix);
mat4.rotate(modelViewMatrix, Math.PI/4, [0,0,1], modelViewMatrix);
mat4.translate(modelViewMatrix, [5,0,0], modelViewMatrix);
```

分析每次调用这些函数后对模型视图矩阵中的值的影响。在调用 mat4.identity()方法后，模型视图矩阵成为一个单位矩阵 I。调用 mat4.rotate()方法之后，模型矩阵的值是单位矩阵后乘旋转矩阵 R，这意味着现在的模型视图矩阵的值为 IR。调用 mat4.translate()之后，模型视图矩阵后乘平移矩阵 T，结果是 IRT。

调用 glMatrix 中的这些函数之后得到模型视图矩阵，通过调用 gl.uniformMatrix4fv()方法将它上传给 GPU 中的顶点着色器。在顶点着色器中，正如前面看到的那样，顶点坐标都要乘上这个模型视图矩阵：

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;

  varying vec4 vColor;

  void main() {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);

    vColor = aVertexColor;
}

</script>
```

这意味着，如果顶点位置用向量 v 表示，则为了将顶点从对象坐标变换到视坐标，要执行以下相乘运算：

$$眼睛坐标中的坐标=IRTv$$

可以看出，顶点的变换过程就如同顶点先乘上矩阵 T，再乘上矩阵 R，最后乘上矩阵 I。为了将这个顺序与 glMatrix 库的函数调用顺序相比较，我们重写这段代码如下：

```
mat4.identity(modelViewMatrix);
mat4.rotate(modelViewMatrix, Math.PI/4, [0,0,1], modelViewMatrix);
mat4.translate(modelViewMatrix, [5,0,0], modelViewMatrix);
```

可以看出，顶点变换的顺序正好与代码中的函数调用顺序相反。图 4-8 说明了这个过程。图中有一个位于原点的立方体，正如前面所述，先调用 mat4.rotate()，然后调用 mat4.translate()。在左边，此立方体位于原点。从图 4-8 的中间可以看出，这个立方体已经沿着 x 轴正方向平移。最后，在右边，这个立方体已经绕 z 轴旋转 PI/4 角度。最终，立方体变换到 z=0 的平面上，位于 x 轴正方向与 y 轴正方向之间的某个位置。

在代码中的调用顺序

```
mat4.rotate()
mat4.translate()
```

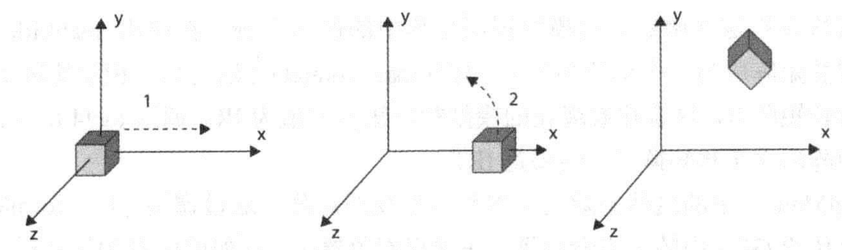


图 4-8　在一个固定的全局的坐标系中，虽然 mat4.rotate()调用是在 mat4.translate()调用之前，但是实际上，平移变换发生在旋转变换之前

如果我们改变 mat4.rotate()和 mat4.translate()的调用顺序，则最终的代码段如下：

```
mat4.identity(modelViewMatrix);
mat4.translate(modelViewMatrix, [5,0,0], modelViewMatrix);
mat4.rotate(modelViewMatrix, Math.PI/4, [0,0,1], modelViewMatrix);
```

这意味着，顶点要乘上模型视图矩阵 ITR(这里的 I 表示单位矩阵，T 是平移矩阵，R 是旋转矩阵)。现在，顶点在眼睛坐标中的位置可以由以下公式得到：

眼睛坐标中的顶点=ITRv

可以看出，这意味着，旋转变换在平移变换之前发生。图 4-9 说明一个位于原点的立方体如何经历这样的变换过程。在左边，立方体位于坐标系的原点位置。在中间，立方体已经绕 z 轴旋转了 PI/4 角度。在右边，立方体沿着 x 轴正方向平移。结果是在 x 轴正方向中的某个位置有一个 PI/4 角度的立方体。

在代码中的调用顺序

```
mat4.translate()
mat4.rotate()
```

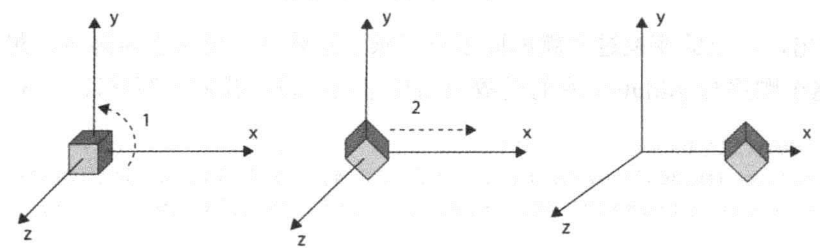


图 4-9　在一个固定的全局的坐标系中，虽然 mat4.translate()调用是在 mat4.rotate()调用之前，但是实际上，旋转变换发生在平移变换之前

当用固定的全局的坐标系看待变换时，变换实际发生的顺序与代码中的调用顺序正好相反。

4.5.2　使用移动的局部的坐标系

在思考一系列变换时，除了使用固定的全局的坐标系外，另一种方法是使用移动的局部的坐标系。这个坐标系捆绑到进行变换的对象上，当我们将某个变换作用于一个对象上时，这个对象的局部坐标系跟随这个对象。相对于此局部坐标系执行所有随后的变换。如果采用这种思维模式，则变换的发生顺序与代码中的调用顺序是一致的。再次考虑相同的代码段：

```
mat4.identity(modelViewMatrix);
mat4.rotate(modelViewMatrix, Math.PI/4, [0,0,1], modelViewMatrix);
mat4.translate(modelViewMatrix, [5,0,0], modelViewMatrix);
```

然后，我们可以这样想象：绕 z 轴旋转这个对象和它的局部坐标系 PI/4 角度，然后沿着 x 轴(现在 x 轴也已经旋转，指向 x 轴正方向与 y 轴正方向之间的 PI/4 角度)平移。图 4-10 说明了应用这种移动的局部的坐标系执行变换的思维模式。

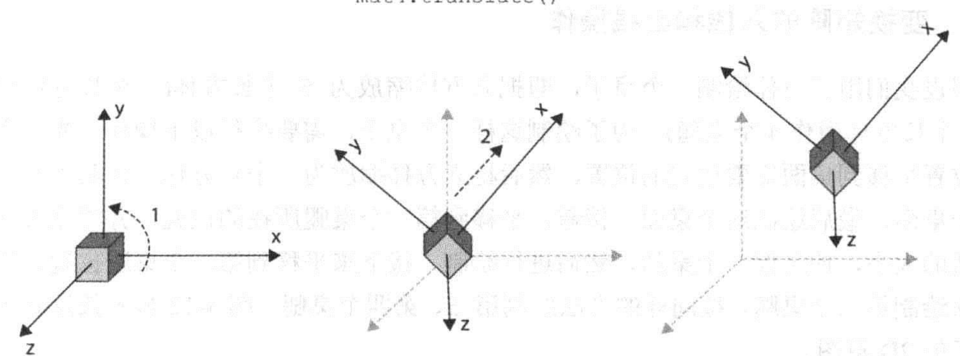


图 4-10　在一个移动的局部的坐标系中，平移变换之前的旋转变换与在源代码中调用 mat4.translate()之前先调用 mat4.rotate()的顺序相一致

前面曾提到，不管使用移动的局部坐标系还是固定的全局坐标系，得到的结果和源代码都是一样的，只是思考问题的方式不同。

为了保持一致，查看在执行 mat4.rotate()之前执行 mat4.translate()且使用一个局部坐标系时是如何思考变换的。因此假设有以下的代码段：

```
mat4.identity(modelViewMatrix);
mat4.translate(modelViewMatrix, [5,0,0], modelViewMatrix);
mat4.rotate(modelViewMatrix, Math.PI/4, [0,0,1], modelViewMatrix);
```

接着，我们想象对这个对象作平移操作，且它的局部坐标系沿着 x 轴移动 5 个单位，然后，以原点为中心作旋转变换操作(这个原点也跟着平移 5 个单位，仍然位于对象的中心)。图 4-11 说明了如何看待这些使用移动的局部的坐标系的变换。

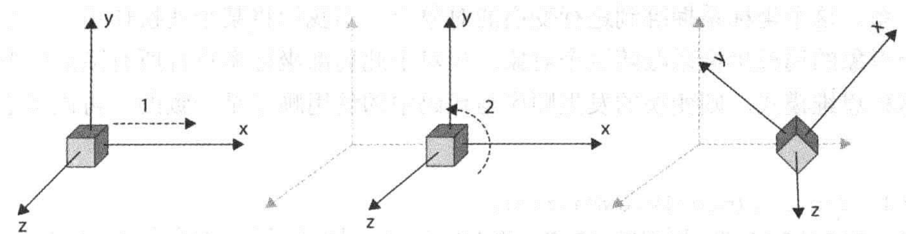


图 4-11　在一个移动的局部的坐标系中，平移操作发生在旋转操作之前，在代码中与此对应的是先调用 mat4.translate()，然后调用 mat4.rotate()

当我们利用局部坐标系思考变换时，变换的执行顺序与代码中的变换调用顺序相一致。局部坐标系跟随变换的对象。

4.5.3　变换矩阵的入栈和出栈操作

假设我们用立方体绘制一个桌子，即把立方体缩放为 5 个长方体(一个长方体用作桌面，4 个长方体用作 4 个桌腿)。为了绘制这样一个桌子，需要执行以下操作：先从桌子的原点位置平移到桌面需要放置的位置，然后将立方体缩放为一个长方体，使其大小合适作为一个桌面，最后绘制这个桌面。接着，平移到第一个桌腿所在的位置，并将立方体缩放为合适的大小，使之像一个桌腿，然后进行绘制。接下来平移到第二个桌腿位置，用同样的方法绘制第二个桌腿。按同样的方法绘制第三、第四个桌腿。图 4-12 显示按这种方法绘制桌子的 2D 草图。

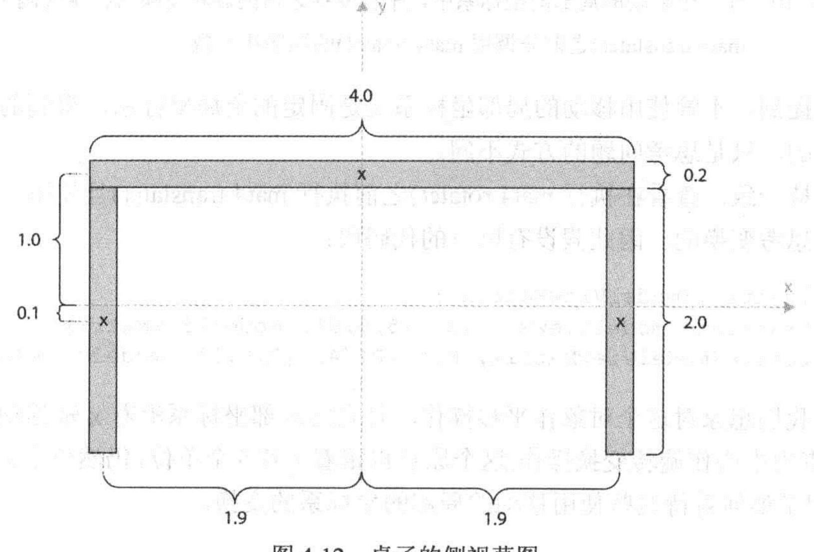


图 4-12　桌子的侧视草图

从性能角度来说，越少调用 gl.drawArrays()和 gl.drawElements()函数越好。这意味着用 5 个长方体绘制一个桌子并不是最有效的方法。然而，本章使用这种方法来说明平移变换以及模型视图矩阵栈的入栈和出栈操作。

用这样的方法绘制一个桌子是可行的，但其过程也相当麻烦。将第一个立方体缩放为桌面后，对第一个桌腿的平移变换必须考虑到这个缩放操作对之后创建的对象的影响。当需要绘制这样的组合对象时，实际上我们通常使用分层变换技术，将变换状态保存在层次上的某个位置。

如果我们用第二种方法绘制这个桌子，则从桌子的原点位置开始。在将桌面平移到合适位置之前，先将当前的模型视图矩阵保存起来，然后将立方体平移到桌面所在的位置，并将它缩放为一个像桌面的长方体，最后绘制桌面。

桌面绘制完成后，不是直接从当前桌面位置平移第一个桌腿，而是先恢复在桌面平移和缩放之前保存的模型视图矩阵。这意味着现在回到了桌面的原点位置，即模型视图矩阵没有缩放效果。

现在，我们再次保存当前模型视图矩阵，然后平移到第一个桌腿位置，将立方体缩放为像一个桌腿的长方体，并绘制它。当第一个桌腿绘制完成后，恢复保存的变换矩阵。然后用同样的方法继续绘制第二、第三、第四个桌腿。

保存这种层级结构的变换矩阵的最好方法是使用变换矩阵栈。当我们想要保存一个变换矩阵时，就将它推入到栈顶；当我们想恢复这个矩阵时，只需要将它从栈顶弹出。图 4-13 说明了模型视图矩阵栈的概念。模型视图矩阵栈的思想对于模型视图矩阵是很有用的，但是在某些情形下，用一个投影矩阵栈也是比较有用的。一个示例是我们在一个 WebGL 应用程序中同时使用透视投影和正交投影。当我们需要在透视投影和正交投影之间切换时，就要用到投影矩阵栈。

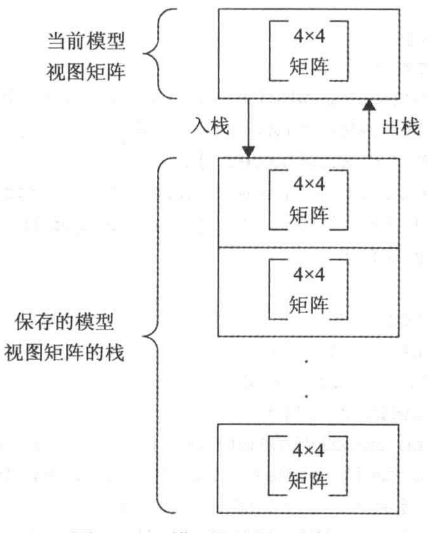


图 4-13 模型视图矩阵栈

JavaScript 的数组数据类型都有 push()和 pop()方法，利用这两个方法可以将数组当作栈来处理。基于 JavaScript 的数组，很容易实现模型视图矩形栈，如下面的代码所示：

```
modelViewMatrix = mat4.create();
modelViewMatrixStack = [];

function pushModelViewMatrix() {
  var copyToPush = mat4.create(modelViewMatrix);
  modelViewMatrixStack.push(copyToPush);
}

function popModelViewMatrix() {
  if (modelViewMatrixStack.length == 0) {
   throw "Error popModelViewMatrix() - Stack was empty ";
  }
  modelViewMatrix = modelViewMatrixStack.pop();
}
```

在 WebGL 应用程序中，当需要保存模型视图矩阵时就调用 pushModelViewMatrix()函数，然后在需要恢复这个矩阵时调用 popModelViewMatrix()函数。

现在假设有一个函数，它可以绘制一个还没有经过变换且两个端点在(-1,-1,-1)和(1,1,1)位置的立方体：

```
drawCube(r,g,b,a)
```

这个函数用(r,g,b,a)颜色绘制一个立方体，这个立方体在还没变换之前的边长为 2。然而，它的效果要考虑到当前的模型视图变换矩阵。

现在为了用 drawCube()函数和前面介绍的模型视图矩阵栈绘制图 4-12 中的桌子，可以使用以下的代码段：

```
function drawTable(){
  // draw table top
  pushModelViewMatrix();
  mat4.translate(modelViewMatrix, [0.0, 1.0, 0.0], modelViewMatrix);
  mat4.scale(modelViewMatrix, [2.0, 0.1, 2.0], modelViewMatrix);
  uploadModelViewMatrixToShader();
  // draw the actual cube (now scaled to a cuboid) in brown color
  drawCube(0.72, 0.53, 0.04, 1.0); // argument sets brown color
  popModelViewMatrix();

  // draw table legs
  for (var i=-1; i<=1; i+=2) {
   for (var j= -1; j<=1; j+=2) {
     pushModelViewMatrix();
     mat4.translate(modelViewMatrix, [i*1.9, -0.1, j*1.9], modelViewMatrix);
     mat4.scale(modelViewMatrix, [0.1, 1.0, 0.1], modelViewMatrix);
     uploadModelViewMatrixToShader();
     drawCube(0.72, 0.53, 0.04, 1.0); // argument sets brown color
```

```
            popModelViewMatrix();
        }
      }
    }
```

进入 drawTable()函数后，第一条语句就是调用 pushModelViewMatrix()函数，保存当前模型视图矩阵。然后沿 y 轴平移 1 个单位，到达我们想要放置桌面的位置。接着调用 mat4.scale()将立方体沿 x 轴方向和 z 轴方向放大 2 倍，沿 y 轴方向放大 0.1 倍。然后调用 uploadModelMatrix- ToShader()函数将这个模型视图矩阵上传到顶点着色器，再调用 drawCube()函数绘制桌面。最后，调用 popModelViewMatrix()函数将模型视图矩阵恢复为调用 mat4.translate()和 mat4.scale()方法之前的值。

桌面绘制完成后，接着是两个嵌套的循环，用于绘制四个桌腿。在循环体内，先用 mat4.translate()方法将立方体平移到桌腿位置，然后缩放立方体，使它沿 x 轴方向和 z 轴方向放大 0.1 倍，沿 y 轴方向放大 1 倍，绘制每个桌腿后，调用 popModelViewMatrix()函数恢复当前模型视图矩阵。

假设渲染这样一个场景，场景中的每个对象都有自己的变换。如果要绘制这样的场景，我们可以使用模型视图变换矩阵栈。我们将对象的变换函数、绘图函数与模型视图矩阵栈的入栈和出栈操作合并在一起，如下所示：

```
pushModelViewMatrix();
mat4.translate(modelViewMatrix, [xPos, yPos, zPos],
    modelViewMatrix);
drawObject();
popModelViewMatrix();
```

4.6　一个完整的示例：绘制几个变换后的对象

现在分析一个完整的示例，它应用本章前面介绍的许多知识。程序清单 4-4 是从本书的伴随网站 wrox.com 上下载的，仔细阅读这段源代码(由于这段源代码太长，因此没有将它包括在本书中)。程序清单 4-4 绘制这样一个场景：在地面上放置一个桌子，桌子上放一个盒子。如果将程序清单 4-4 中的代码载入浏览器，则会看到如图 4-14 所示的结果。

对于你来说，程序清单 4-4 中的绝大部分代码应该是熟悉的。先从程序清单 4-4 顶部的 JavaScript 代码开始，这段代码如下：

```
var floorVertexPositionBuffer;
var floorVertexIndexBuffer;
var cubeVertexPositionBuffer;
var cubeVertexIndexBuffer;

var modelViewMatrix;
```

```
var projectionMatrix;
var modelViewMatrixStack;
```

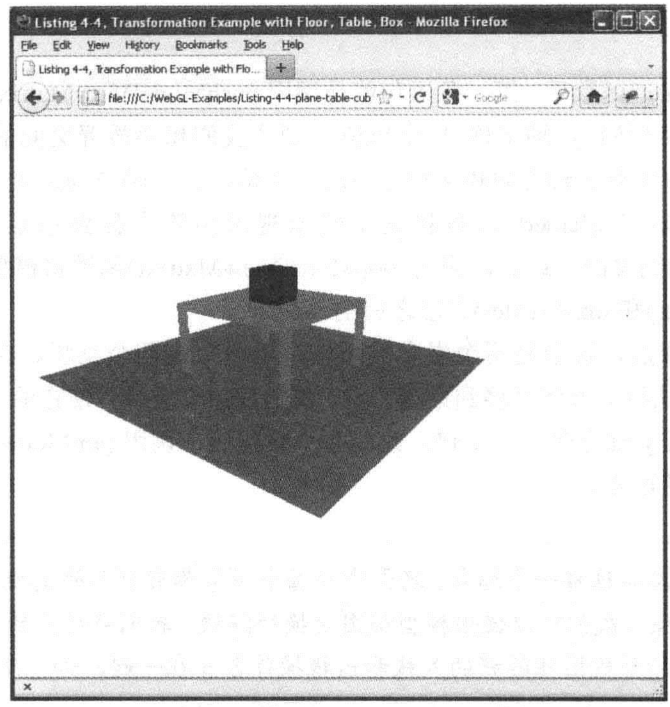


图 4-14　使用程序清单 4-4 绘制的场景

在这段代码中，定义了几个变量来表示地板和立方体的顶点位置和元素索引。此外，这里也定义了几个变量来表示模型矩阵、投影矩阵和模型视图矩阵栈。

下一个需要分析的代码段位于 setupShaders()函数的末尾。在这段代码中创建模型视图矩阵、投影矩阵和模型视图矩阵栈并将它们赋给对应的全局变量：

```
  modelViewMatrix = mat4.create();
  projectionMatrix = mat4.create();
  modelViewMatrixStack = [];
}
```

紧随这段代码之后，有两个辅助函数，它们推入模型视图矩阵栈或者从栈中弹出：

```
function pushModelViewMatrix() {
  var copyToPush = mat4.create(modelViewMatrix);
  modelViewMatrixStack.push(copyToPush);
}

function popModelViewMatrix() {
  if (modelViewMatrixStack.length == 0) {
    throw "Error popModelViewMatrix() - Stack was empty ";
  }
  modelViewMatrix = modelViewMatrixStack.pop();
}
```

在这个示例中，缓冲的创建操作分为两个函数。一个函数是 setupFloorBuffers()，它设置地板的缓冲，另一个函数是 setupCubeBuffers()，它设置立方体的缓冲，此立方体用来保存桌子和桌子上的盒子。函数 setupFloorBuffers()非常简单，它仅仅建立代表一个平面的顶点位置。这个平面位于 y=0 位置，顶点的 x 和 z 值分别为-5.0 和 5.0，如以下代码所示：

```
function setupFloorBuffers() {
  floorVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, floorVertexPositionBuffer);

var floorVertexPosition = [
    // Plane in y=0
     5.0, 0.0,  5.0,  //v0
     5.0, 0.0, -5.0,  //v1
    -5.0, 0.0, -5.0,  //v2
    -5.0, 0.0,  5.0]; //v3
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(floorVertexPosition),
               gl.STATIC_DRAW);
```

在这段代码之后是函数 setupCubeBufffers()，它定义立方体的顶点。如何为这个立方体的顶点创建缓冲的详细过程将在下一节中讨论。

4.6.1 使用 WebGL 代码创建立方体

使用 WebGL 创建立方体并不困难。与平常一样，先创建顶点缓冲。第一次创建立方体时，保证全部三角形的顶点组绕顺序都正确确实有点困难。三角形组绕顺序已经在第 3 章中讨论过，但在这里，我们只是将它当作 2D 对象。在 3D 场景中绘制对象时，为绘制任何实体对象的外表面，只需要正面的三角形。

在下面的代码段中，我们再次看到 setupCubeBuffer()函数。这段代码中的顶点对应于如图 4-15 所示的立方体。对象坐标的原点位立方体的中心，从下面这段代码可以看出，这个立方体的边长为 2 个单位。

```
function setupCubeBuffers() {
  cubeVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexPositionBuffer);

  var cubeVertexPosition = [
       // Front face
       1.0,  1.0, 1.0, //v0
      -1.0,  1.0, 1.0, //v1
      -1.0, -1.0, 1.0, //v2
       1.0, -1.0, 1.0, //v3

       // Back face
       1.0,  1.0, -1.0, //v4
      -1.0,  1.0, -1.0, //v5
```

```
            -1.0, -1.0, -1.0, //v6
             1.0, -1.0, -1.0, //v7

             // Left face
            -1.0,  1.0,  1.0, //v8
            -1.0,  1.0, -1.0, //v9
            -1.0, -1.0, -1.0, //v10
            -1.0, -1.0,  1.0, //v11

             // Right face
             1.0,  1.0, 1.0, //12
             1.0, -1.0, 1.0, //13
             1.0, -1.0, -1.0, //14
             1.0,  1.0, -1.0, //15

             // Top face
             1.0,  1.0,  1.0, //v16
             1.0,  1.0, -1.0, //v17
            -1.0,  1.0, -1.0, //v18
            -1.0,  1.0,  1.0, //v19

             // Bottom face
             1.0, -1.0,  1.0, //v20
             1.0, -1.0, -1.0, //v21
            -1.0, -1.0, -1.0, //v22
            -1.0, -1.0,  1.0, //v23
    ];

    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(cubeVertexPosition),
                  gl.STATIC_DRAW);
    cubeVertexPositionBuffer.itemSize = 3;
    cubeVertexPositionBuffer.numberOfItems = 24;

    cubeVertexIndexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeVertexIndexBuffer);
    var cubeVertexIndices = [
              0, 1, 2,     0, 2,  3,    // Front face
              4, 6, 5,     4, 7,  6,    // Back face
              8, 9, 10,    8, 10, 11,   // Left face
              12, 13, 14,  12, 14, 15,  // Right face
              16, 17, 18,  16, 18, 19,  // Top face
              20, 22, 21,  20, 23, 22   // Bottom face
         ];
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
                  new Uint16Array(cubeVertexIndices),gl.STATIC_DRAW);
    cubeVertexIndexBuffer.itemSize = 1;
    cubeVertexIndexBuffer.numberOfItems = 36;
}
```

从图 4-15 可以看出，立方体每个角点的附近都标示有 3 个顶点，这 3 个顶点都对应同一个位置点。一个立方体通常只有 8 个不同的顶点位置。在本例中，用单一颜色绘制这个立方体，不需要定义法线，因此 8 个顶点就足够了。但是，对于通用对象，立方体的每个面要用不同的颜色绘制，或者每个面都需要法线，因此每个角点需要 3 个顶点表示，因为每个角点共享 3 个不同的面。这个示例使用了 24 个顶点，这样可以看出通用对象的表示方法，如果需要为立方体的每个面定义不同的颜色，则比较容易修改代码。

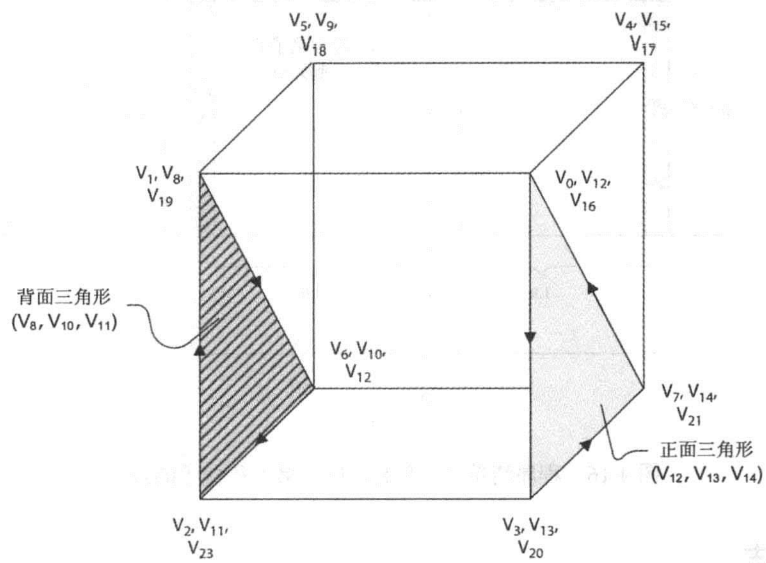


图 4-15　立方体示意图，图中标示了一个正面三角形和一个背面三角形

图 4-15 也显示，立方体左侧面的一个三角形使用顺时针组绕，它是一个背面三角形。右侧的带阴影三角形使用逆时针组绕，它是一个正面三角形。

4.6.2　视图变换和模型变换的组织

程序清单 4-4 的其余代码很容易理解，它与本章前面介绍的代码段非常相似。图 4-16 显示了这个示例中的地板、桌子和盒子的位置及尺寸。

需要注意的是代码中的视图变换和模型变换的建立顺序。首先用 mat4.lookAt()设置视图变换，然后将不同对象的模型变换添加到模型视图矩阵中。这样，对象的顶点先乘上模型变换矩阵，再乘上视图变换矩阵。这正符合本章前面提到的 WebGL 变换流水线的处理顺序。

有一个方法可能帮助你真正掌握变换的概念，只是这个方法有点传统，要用到笔和纸。设想有一个对象，如立方体，我们想从不同的方向看这个立方体。写下所需要的变换，修改程序清单 4-4 中的代码，进行测试，直到成功为止。反复做这样的练习，直到你真正掌握变换为止。

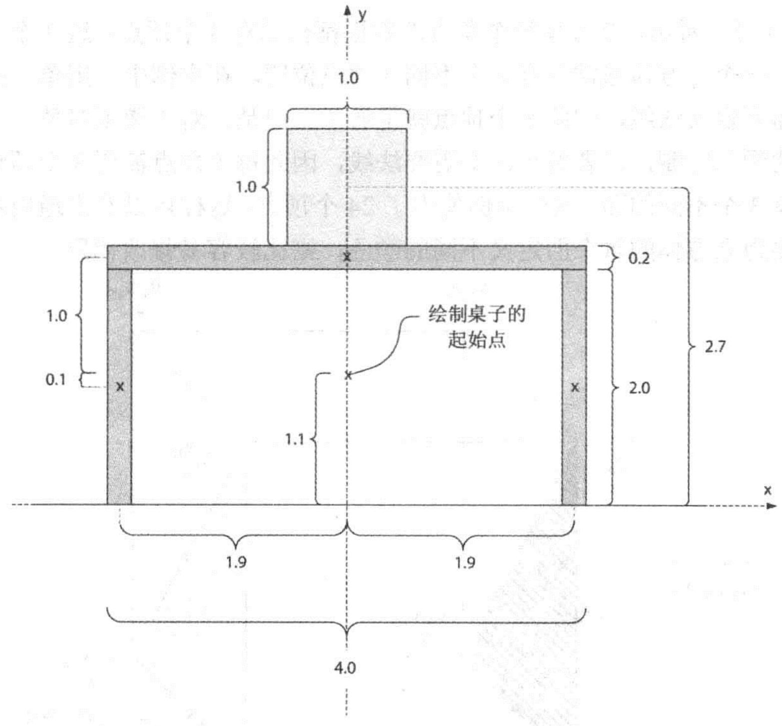


图 4-16　程序清单 4-4 中的地板、桌子和盒子的尺寸

4.7　小结

本章介绍了 3 个小型但十分有用的 JavaScript 库，它们可以帮助我们实现 JavaScript 中所需要的矩阵和向量运算。它们是：

- Sylvester
- WebGL-mjs
- glMatrix

你已经知道，WebGL-mjs 库与 glMatrix 库非常相似，编写这两个库时都已经考虑到 WebGL。此外，我们还介绍了许多与变换有关的内容：

- 模型变换
- 视图变换
- 模型视图变换
- 投影变换
- 透视除法
- 视口变换

你还了解到，模型视图变换是模型变换和视图变换的组合。此外，你还知道了变换顺序的重要性，以及如何从固定的全局坐标系或跟对象一起的移动的局部坐标系角度思考一系列变换。

第 5 章

纹理贴图

本章主要内容：

- 了解纹理贴图的基本概念以及它在 3D 图形中的应用
- 如何通过解决丢失上下文问题提高应用程序的健壮性
- 了解 2D 纹理与立方映射纹理之间的差别
- 如何在 WebGL 中载入纹理数据
- 掌握纹理的伸展(magnification)和收缩(minification)模式
- 掌握纹理的过滤和 Mip 映射
- 在 WebGL 中创建自己的纹理的需要考虑的几个因素
- 为什么同源策略对于 WebGL 中的纹理很重要
- 理解跨域资源共享以及它在 WebGL 中有什么作用

在对象的表面应用纹理是 WebGL 的一个重要操作。其思想是用一幅彩色图像表示对象的细节，这些细节在对象的几何模型中是不存在的。用简单的话来说，纹理贴图过程就是把纹理图像映射到几何对象上的过程。

例如考虑一堵砖墙，它的模型很容易用两个橙色三角形表示。可以想象，这样一堵墙看起来不是很真实，但是由于只需要几个顶点就可以表示它，因此绘制速度非常快。与此相反的是，如果我们要使这堵墙看起来比较真实，则需要用大量的顶点表示这个墙的几何改变和颜色变化。用这种方法可以得到比较真实的模型，但是建立这样的模型却有点困难，而且计算机需要渲染较长的时间。

有了纹理贴图，我们只需要用一个很大的多边形表示墙的几何结构，然后把墙的彩色图像应用到这个多边形上。只要观察者不是离这堵墙很近，就能够用很有限的几何细节得到比较有真实感的墙。

本章的主要内容都与纹理有关，但是首先要了解到目前为止一直被忽略的重要问题：

丢失上下文。丢失上下文会影响应用程序的稳定性和健壮性。

5.1 理解丢失上下文

GPU 是一个共享资源，除了 WebGL 应用程序，还有其他客户端使用 GPU 资源。WebGL 存在一个丢失上下文的问题，这是指如果当前应用程序被剥夺了 GPU 资源，则它会丢失上下文。

造成 WebGL 应用程序丢失上下文有好几个原因。一个情形是绘图过程调用占用 CPU 很长时间，以使用户的系统不能响应或者被挂起。即使对于桌面 OpenGL，也可能存在这样的问题，但是由于 WebGL 运行非信任代码，这些代码通过顶点着色器和片段着色器对 GPU 进行低级访问，这个问题对于 WebGL 比桌面 OpenGL 更加严重。

有些操作系统和设备驱动程序解决这个问题的一个方法用驱动程序检测 GPU 是否处于无响应状态。如果是，它就复位 GPU，并通过 webglcontestlost 事件告诉 WebGL 应用程序，它已丢失上下文。

在本书到目前为止介绍的示例中，并没有考虑到应用程序丢失上下文这个问题。应用程序中并没有侦听 webglcontextlost 事件。如果我们希望当丢失上下文时应用程序仍然能“健壮”地运行，这些示例使用的结构都不值得推荐。当丢失上下文时，它隐含了以下事情：

- gl.isContextLost()方法返回 true。
- 声明为 void 的 WebGL 方法立刻返回。
- 把 null 传递给可以返回 null 的 WebGL 方法。
- gl.getAttribLocation()方法不是返回顶点着色器中属性的有效位置，而是返回–1。
- 当丢失上下文时，第一次调用 gl.getError()方法会返回 gl_CONTEXT_LOST_WEBGL 信息，之后，它返回 gl.NO_ERROR，直到上下文恢复为止。
- gl.checkFramebufferStatus()方法返回 gl.FRAMEBUFFER_UNSUPPLIED。
- 所有以 is 开头的 WebGL 方法(如 isFinished()、isProgram()等)都返回 false。

这些情况对所有第一次看到这个问题的读者可能有点复杂，如果我们希望能够正确处理丢失上下文问题，必须用新的概念考虑这个问题。

5.1.1 理解解决丢失上下文问题所需要的设置

当 WebGL 应用程序发生丢失上下文问题后，默认的行为是这个应用程序无法重新得到上下文。为了继续使用 WebGL 应用程序，用户必须用手动方法将这个应用程序重新载入浏览器中，并且希望这个上下文现在可以再次使用。

注册两个事件侦听程序就可以重写这种默认的行为：其中一个侦听程序在出现上下文丢失时会通知用户的应用程序，另一个侦听程序在发现上下文恢复后会通知用户的应用程序。

为了注册能够侦听 webglconextlost 事件的侦听程序，要用下面的代码：

```
canvas = document.getElementById("myGLCanvas");
canvas.addEventListener('webglcontextlost',
                        handleContextLost, false);
```

addEventListener()方法的第一个参数是需要侦听的事件名称。第二个参数是当事件发生时需要调用的事件处理程序。第三个参数是一个布尔值，表示在事件传播的捕获阶段中事件处理程序是否捕获事件。在本例中，在捕获阶段不需要捕获任何事件，因此把这个参数设置为 false。你在第 6 章将学习有关事件处理程序和事件传播等内容，那时你才会明白这个参数的意义。

为了注册一个恢复上下文时可以调用的事件侦听程序，要使用下面的代码：

```
canvas.addEventListener('webglcontextrestored',
                        handleContextRestored, false);
```

在调用 webglcontextlost 事件侦听程序时，需要阻止默认的行为模式，即上下文不可恢复模式。为此，需要调用此事件的 preventDefault()方法，如下所示：

```
function handleContextLost(event) {
  event.preventDefault();

  cancelRequestAnimFrame(pwgl.requestId);

  ...

}
```

此外，通常还需要终止绘制循环。这看起来十分奇怪，因为到目前为止，在本书所有的代码示例中绘图方法只执行一次。但是，对象通常可能在一个场景中到处移动。在这种情况下，用户的绘图程序，即这些示例中的 draw()函数，必须多次重复调用。有关这方面的知识本章后面将作进一步介绍，但是目前你只需要知道，调用 cancelRequestAnimFrame()能够终止对 draw()函数的调用。

当恢复上下文后，系统就调用用户注册的事件侦听程序。此时，程序已经把 WebGL 的状态复位到默认状态。此外，所有由 WebGL 分配的资源都变成无效。

这意味着，我们不得不重新设置所需要的状态，重新创建纹理、缓冲、着色器、程序以及其他资源。这通常意味着，还需要像第一次启动应用程序那样执行同样的设置和初始化操作。恢复上下文后需要处理的最后一件事情是重新启动绘图过程。为此，要调用 requestAnimFrame()函数。下面是处理 webglcontextrestore 事件的完整函数示例：

```
function handleContextRestored(event) {
  setupShaders();
  setupBuffers();
  setupTextures();
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  pwgl.requestId = requestAnimFrame(draw,canvas);
}
```

模拟上下文丢失事件

如果已经在 WebGL 应用程序中实现了 webglcontextlost 事件和 webglcontextrestore 事件的处理程序，那么必须经过测试以确保它们真正起作用。在这个时候，一个非常有用的资源就是前面第 2 章介绍过的调试库 webgl-debug.js。在第 2 章中我们曾提到，它可以把 WebGL 错误输出到 JavaScript 控制台上。

在处理丢失上下文问题时，可以用 webgl-debug.js 库模拟 webglcontextlost 事件和 webglcontextrestore 事件。为了使用这个库，如同使用其他任何外部 JavaScript 代码一样，首先必须在 WebGL 应用程序中包含 webgl-debug.js 文件：

```
<script src="webgl-debug.js"> <script>
```

然后用一个简单方法引用这个库，即在应用程序的创建和初始化代码段中添加类似于下面的代码：

```
canvas = document.getElementById("myGLCanvas");
canvas = WebGLDebugUtils.makeLostContextSimulatingCanvas(canvas);

canvas.addEventListener('webglcontextlost',
                        handleContextLost, false);

canvas.addEventListener('webglcontextrestored',
                        handleContextRestored, false);

...

window.addEventListener('mousedown', function() {
  canvas.loseContext();
});
```

如果调用 WebGLDebugUtils.makeLostContextSimulatingCanvas()方法，则会创建原始画布的包装，从而可以模拟 webglcontextlost 事件和 webglcontextrestore 事件。然后只需要添加这两个事件的侦听程序，最后添加鼠标按下(mousedown)事件的侦听程序。在 mousedown 事件的处理程序中，调用 canvas.loseContext()方法(它是新建的包装画布的一部分)可以模拟 webglcontextlost 事件和触发器，从而可以调用用户注册的事件处理程序。

第 2 章曾提到，你可以从 Khronos 储存库下载 webgl-debug.js 库：https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/sdk/debug/webgl-debug.js

webgl-debug.js 文件包含一些注释，它们进一步解释如何使用这个库的其他功能。

5.1.2 处理丢失上下文问题时需要考虑的几个因素

本书前面的示例都没有考虑到丢失上下文。当然在实际的应用程序中，建议用户考虑这个问题。即使这个问题并不经常发生，但是创建应用程序的结构应该考虑到这种情况。

本节讨论当上下文丢失时，为了使应用程序更加健壮而需要考虑哪些因素。

Khronos WebGL 维基(www.khronos.org/webgl/wiki/HandlingContextLost)网页包含一些丢失上下文问题的有用信息和说明。这些信息最初是由 Gregg Tavares 收集和整理的，Gregg Tavares 是众多对 WebGL 技术做出巨大贡献的人士之一。

1. 不要给 WebGL 资源对象添加新的属性

本书前面的示例总是给不同的 WebGL 资源对象添加新的属性，这些资源对象是通过 WebGL API 创建的。例如，经常给新建的 WebGLBuffer 对象添加 itemSize 和 numberOfItems 属性(这可以从下面的代码段看出)：

```
floorVertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, floorVertexPositionBuffer);

var floorVertexPosition = [
   // Plane in y=0
   5.0, 0.0,  5.0, //v0
   5.0, 0.0, -5.0, //v1
  -5.0, 0.0, -5.0, //v2
  -5.0, 0.0,  5.0]; //v3

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(floorVertexPosition),
              gl.STATIC_DRAW);

// THE TWO LINES BELOW ARE ARE BAD IF THE CONTEXT IS LOST!
fl oorVertexPositionBuffer.itemSize = 3;
fl oorVertexPositionBuffer.numberOfItems = 4;
```

虽然这也许是组织代码的一种简便方法，而且在许多真实的 WebGL 应用程序中或许会看到类似的代码，但是当上下文丢失时，这不是代码的最好构建方法。当上下文丢失时，gl.createBuffer()返回 null，如果我们给值为 null 的 floorVertexPositionBuffer 对象添加 itemSize 和 numberOfItems 属性，则肯定会出现异常。同样的道理，我们不应该给其他 WebGL 对象(如纹理对象、着色器对象、程序对象等)添加属性。

代码还有许多其他组织方法。一个方法是给另一个不是由 WebGL API 创建的全局对象添加属性。在下面这段代码中，一个名为 pwgl(这是 Professional WebGL 的缩写)的对象用来保存相关信息：

```
pwgl.floorVertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.floorVertexPositionBuffer);

var floorVertexPosition = [
    // Plane in y=0
     5.0, 0.0,  5.0,  //v0
     5.0, 0.0, -5.0,  //v1
    -5.0, 0.0, -5.0,  //v2
    -5.0, 0.0,  5.0]; //v3
```

```
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(floorVertexPosition),
                gl.STATIC_DRAW);

pwgl.FLOOR_VERTEX_POS_BUF_ITEM_SIZE = 3;
pwgl.FLOOR_VERTEX_POS_BUF_NUM_ITEMS = 4;
```

2. 在检查着色器编译结果的同时检查丢失上下文

在调用 gl.getShaderParameter()方法检查着色器的编译结果时，也必须调用 gl.isContextLost()方法检查上下文是否丢失。这样做的理由是防止丢失上下文导致编译失败。

下面这段代码只检查编译状态，而没有检查上下文是否丢失：

```
gl.shaderSource(shader, shaderSource);
gl.compileShader(shader);

if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
  alert(gl.getShaderInfoLog(shader));
  return null;
}
```

只需要对这段代码做一个简单的修改，即只要添加对 gl.isContextLost()方法的调用，就可以使应用程序在上下文丢失时仍然能够健壮地运行。这从下面的代码段可以看出：

```
gl.shaderSource(shader, shaderSource);
gl.compileShader(shader);

if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS) &&
    !gl.isContextLost()) { // To avoid alert() when context is lost
  alert(gl.getShaderInfoLog(shader));
  return null;
}
return shader;
```

3. 在检查着色器链接状态的同时检查上下文是否丢失

前面提到过，在检查着色器编译状态时需要检查上下文有没有丢失。同样的道理，我们也要在检查程序链接状态的同时检查上下文是否丢失。原始代码并没有在检查链接状态时检查上下文是否丢失，以下就是这样的代码段：

```
shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);

if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
  alert("Failed to setup shaders");
}
```

以下是更新后的代码，它检查上下文是否丢失：

```
var shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);

if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS) &&
    !gl.isContextLost()) {
  alert("Failed to setup shaders");
}
```

5.2 2D纹理与立方映射纹理

WebGL中最常见的纹理格式是2D纹理。2D纹理的最基本形式是使用单幅图像作为纹理，如前面示例中的砖墙纹理。虽然图像是最常用的纹理，但是在WebGL中也可以使用画布元素、视频元素、ImageData对象和类型化数组中的原始数据作为2D纹理的输入数据。

不管2D纹理的输入数据属于哪一类，2D纹理都需要一个坐标系，这样才可以定义从纹理上的哪个位置采样纹理元素(简称纹素)。纹素(texel)只是纹理的一个像素，但是为了把它与屏幕上的像素区分开来，我们通常称它为纹理元素。纹理采样是指从纹理上提取一个像素。

图5-1说明了如何在一个2D纹理上定义纹理坐标。2D纹理的纹理坐标通常都用(s,t)或(u,v)表示。纹理的左下角定义为原点，它的坐标值为(0.0,0.0)。不管纹理是不是一个正方形，纹理右上角的坐标都为(1.0,1.0)。s轴是水平轴且指向右方，t轴是垂直轴且指向上方。

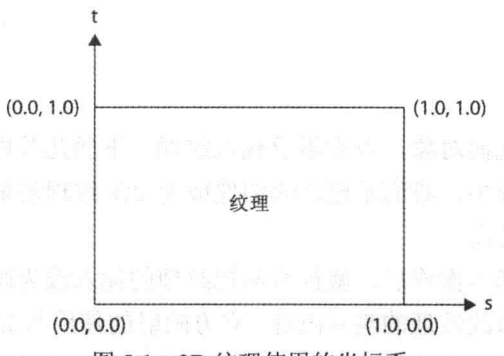


图5-1 2D纹理使用的坐标系

除了通常的2D纹理外，WebGL还支持名为立方映射纹理的功能。一个立方映射纹理虽然属于单个纹理对象，但是它由6个正方形纹理组成，每个正方形对应于正方体的一个面。立方映射纹理常用于环境映射，即它用来在细小对象上产生环境反射效果。

图5-2说明用立方纹理实现环境映射的一个示例，这正是我们在第2章中介绍WebGL Inspector时使用的同一个示例。为了真正看到这个演示示例的结果，最好使用浏览器观看这个演示程序的在线版本。这个演示程序也称为“Dynamic Cubemap”(动态的立方映射)，

它是由 Gregg Tavares 开发的，从 http://webglsamples.googlecode.com 上可以下载。

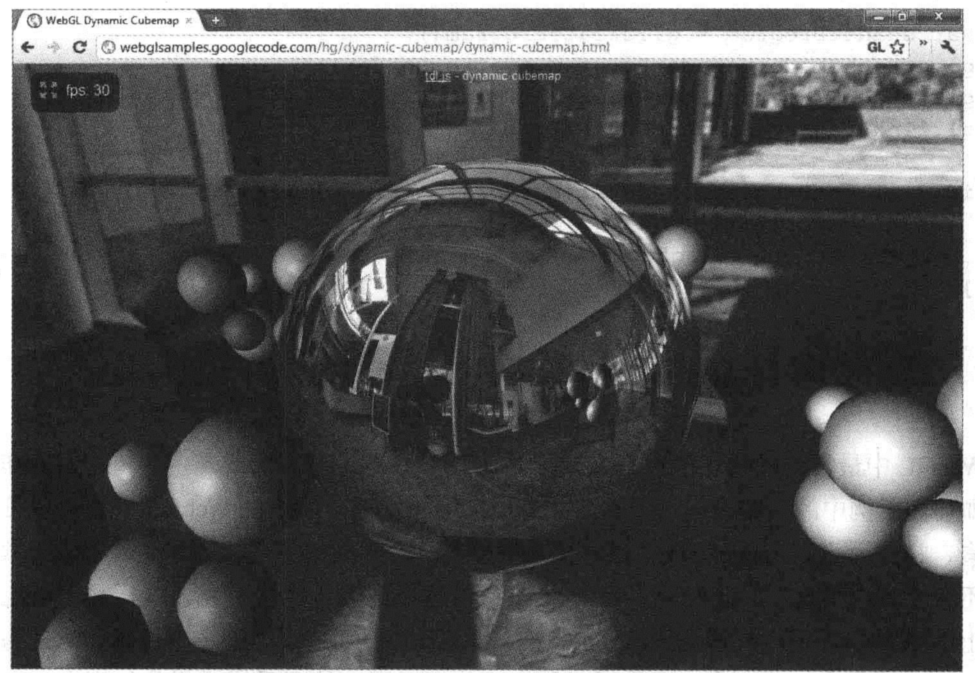


图 5-2 Gregg Eavares 开发的演示程序“Dynamic Cubemap”是一个用立方映射生成环境映射的优秀示例

立方映射可以用来创建天空盒(skybox)，后者是 3D 游戏中一个常用的技术。此技术的思想是，遥远的对象，如天空、远山或远方的建筑，都是映射到立方对象表面的纹理的一部分。当这个立方对象足够大时，它就使人产生远距离 3D 环境的印象。

5.3 载入纹理

为了把纹理应用于几何对象，首先需要载入纹理。下面几节内容介绍如何把普通图像文件的纹理载入纹理对象中，我们希望这幅图像成为 2D 纹理的输入数据。图像文件可以是 PNG、JPEG 或 GIF 格式。

当你掌握了纹理的基本操作后，就很容易把纹理的输入改为画布元素或视频元素。这个示例向你说明了如何修改纹理的输入内容。立方映射的使用与 2D 纹理的使用非常相似，因此在这种情形下，最好的方法是先学习 2D 纹理的使用，然后掌握立方映射工作方式与 2D 纹理的不同之处。

5.3.1 创建 WebGLTexture 对象

在 WebGL 中使用纹理的第一个步骤是为每一个纹理创建一个 WebGLTexture 对象。创建纹理对象要使用下面的代码：

```
var texture =gl.createTexture()
```

WebGLTexture 是一个容器对象，它可以作为纹理的引用，通过它访问与此纹理有关的处理参数。如果你曾经编写过 OpenGL 或 OpenGL ES 程序，则 WebGL 中的 gl.createTexture()方法与 OpenGL 和 OpenGL ES 中的 glGenTextures()函数相对应。

还有一个方法可以显式地删除一个 WebGLTexture 对象。例如，如果要删除一个名为 texture 的纹理对象，则可以使用以下的代码：

```
gl.deleteTexture(texture);
```

注意，当结束使用纹理时，并不需要调用 gl.deleteTexture()方法。JavaScript 垃圾收集在销毁 WebGLTexture 对象时会自动删除相应的纹理对象。这个方法只是给用户提供更灵活的控制权，控制何时销毁纹理对象。

5.3.2 绑定纹理

在对新创建的纹理对象做任何操作之前，首先需要把它绑定为当前纹理对象。例如，为了把一个名为 texture 的纹理对象绑定为一个 2D 纹理对象，要用下面的代码：

```
gl.bindTexture(gl.TEXTURE_2D,texture);
```

调用 gl.bindTexture()可告诉 WebGL，这就是从现在起需要操作的纹理对象。这个方法与 WebGLTexture 纹理对象的关系就如同 gl.bindBuffer()方法与 WebGLBuffer 缓冲对象的关系。

5.3.3 载入图像数据

绑定了纹理对象后，我们就可以把图像数据载入纹理对象中。这意味着，把纹理数据上传到 GPU(或 GPU 可以访问的内存)。把纹理数据上传给 GPU 要使用 gl.texImage2D()方法，它的用法将在下一节介绍。这个方法可以接受各种不同格式的纹理数据，但是当纹理是普通图像文件(PNG、GIF 或 JPEG)时，则它通常可以接受一个 HTML DOM 类型 Image 对象。因此，在调用 gl.texImage2D()方法之前，需要把数据保存在一个 Image 对象中。

一个 Image 对象是由 HTML 文档中的<img>标记显式创建得到的元素。但是调用下面的函数也可以显式创建一个 Image 对象：

```
image = new Image();
```

新建的 Image 对象是一个空对象，还没有载入任何图像数据。为了把图像数据载入 Image 对象中，需要把 Image 对象的 src 属性设置为需要载入图像的 URL。只要把图像的 URL 赋给这个 src 属性，系统就会按异步方式载入这个图像。

为了知道图像何时载入结束，可以使用 onload 事件。当图像载入结束时会立刻引发这个事件。下面这个代码段创建一个 Image 对象，并载入图像数据：

```
function loadImageForTexture(url, texture) {
  var image = new Image();
  image.onload = function() {
    pwgl.ongoingImageLoads.splice(pwgl.ongoingImageLoads.indexOf(image), 1);
```

```
        textureFinishedLoading(image, texture);
    .}
    pwgl.ongoingImageLoads.push(image);
    image.src = url;
}
```

载入图像且 onload 事件触发匿名函数时，调用 textureFinishedLoading()函数把处理过程转入下一个步骤。在下一个步骤中把图像数据上传给 GPU。

在用 JavaScript 代码把图像数据载入一个 Image 对象时，一个常用的方法是把 onload 事件处理程序赋给 Image 对象。当我们想把一个 Image 对象作为 WebGL 中某个纹理的输入数据时，就是使用这种方法。但是当我们想在一个 HTML5 2D 画布上绘制 Image 对象时，类似的代码也很常见。

本例的 loadImageForTexture()函数也有两行代码专门用来解决 WebGL 中的丢失上下文问题。需要跟踪请求发送给哪个图像，以防在图像载入结束之前或者 onload 事件触发之前丢失上下文。在把图像的 URL 赋给 Image 对象的 src 属性之前，为了记住这幅图像，要把图像添加到一个数组中，如下所示：

```
pwgl.ongoingImageLoads.push(image);
```

当 onload 事件触发后，就可以用下面的代码删除数组中的这幅图像：

```
pwgl.ongoingImageLoads.splice(pwgl.ongoingImageLoads.indexOf(image), 1);
```

如果出现丢失上下文，则可以循环访问当前已载入图像的数组，删除每个元素的 onload 事件处理程序，如下所示：

```
function handleContextLost(event) {
  event.preventDefault();
  cancelRequestAnimFrame(pwgl.requestId);

   // Ignore all ongoing image loads by removing
   // their onload handler
   for (var i = 0; i < pwgl.ongoingImageLoads.length; i++) {
     pwgl.ongoingImageLoads[i].onload = undefined;
  }
  pwgl.ongoingImageLoads = [];
}
```

纹理大小必须是 2 的 n 次方

在学习如何载入图像数据时，必须知道图像可接受的大小。开发人员经常选择宽度和高度都是 2 的 n 次方的图像(即图像的宽度和高度为 1、2、4、8、16、32、64、128 等值)。另一种表示法是纹理图像必须是 $2^m \times 2^n$ 的格式，这里的 m 和 n 都非负的整数。

采用这种方法的理由之一是老式的 GPU 只支持纹理的宽度和高度都是 2 的 n 次方。桌面 OpenGL 2.0 及之后的版本实际上可以支持非 2 的 n 次方(NPOT)的纹理。在 OpenGL ES 2.0 和 WebGL 中，允许使用 NPOT 纹理，但是存在以下限制：

- 如果使用 NPOT 纹理，则不能使用 Mip 映射贴图。
- 唯一允许使用的重复模式是 gl.CLAMP_TO_EDGE。

当在本章后面阅读有关Mip映射和纹理坐标包装后，就会明白这两个限制意味着什么。

5.3.4 将纹理上传到 GPU

为了把纹理上传到 GPU，需要调用 gl.texImage2D()方法。这个方法有多个不同的版本，它们各有不同的参数，具体用法取决于用作纹理的数据类型。这个方法的 3 个以前的原型如下所示：

```
void texImage2D(GLenum target, GLint level, GLenum internalformat,
GLenum format, GLenum type, HTMLImageElement image) raises (DOMException)

void texImage2D(GLenum target, GLint level, GLenum internalformat,
GLenum format, GLenum type, HTMLCanvasElement canvas) raises (DOMException)

void texImage2D(GLenum target, GLint level, GLenum internalformat,
GLenum format, GLenum type, HTMLVideoElement video) raises (DOMException)
```

对于第一个版本，纹理数据是一个 HTML DOM 类型的 Image 对象。第二个版本接受一个 HTML5 画布元素作为纹理的输入数据。最后一个版本接受一个视频元素作为纹理的输入数据。

接受 HTML Image 对象的版本可能会是我们最经常使用的方法。为了将 Image 对象上传到 GPU，要像下面的代码那样调用 gl.texImage2D()方法：

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
              gl.UNSIGNED_BYTE, image);
```

第 1 个参数表示目标是一个 2D 纹理，第 2 个参数指定了 Mip 映射级别，Mip 映射纹理将在本章后面讨论。目前暂且只需要知道，这里必须把这个参数设置为 0。

第 3 个参数表示内部格式，第 4 个参数表示格式。在 WebGL 中，这两个参数必须相同。在本例中，用 gl.RGBA 表示此纹理的每个纹素都有红、绿、蓝和 alpha 通道这 4 个分量。

第 5 个参数定义了每个纹素数据的存储类型。用 gl.UNSIGNED_BYTE 表示用一个字节保存红、绿、蓝和 alpha 的每个通道信息。这意味着，每个纹素需要占用 4 个字节的内存。

最后一个参数表示 HTML DOM 类型的 Image 对象，此对象已载入图像数据。从这个函数的原型可以看出，这个参数也可以是 HTML5 画布元素或视频元素。

在下面这个示例中，gl.texImage2D()函数使用 gl.RGBA 格式和 gl.UNSIGNED_BYTE 数据类型：

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
              gl.UNSIGNED_BYTE, image);
```

表 5-1 说明第 3 个参数、第 4 个参数和第 5 个参数的各种组合情况。

表 5-1 格式参数和数据类型参数的各种组合

格　式	类　型	每个纹素的字节数
gl.RGBA	gl.UNSIGNED_BYTE	4
gl.RGB	gl.UNSIGNED_BYTE	3
gl.RGBA	gl.UNSIGNED_SHORT_4_4_4_4	2
gl.RGBA	gl.UNSIGNED_SHORT_5_5_5_1	2
gl.RGB	gl.UNSIGNED_SHORT_5_6_5	2
gl.LUMINANCE_ALPHA	gl.UNSIGNED_BYTE	2
gl.LUMINANCE	gl.UNSIGNED_BYTE	1
gl.ALPHA	gl.UNSIGNED_BYTE	1

第一列表示 gl.texImage2D()方法的第三个和第四个参数使用的格式或内部格式。前面已经解释过，gl.RGBA 表示每个纹素都有红、绿、蓝和 alpha 通道。格式 gl.RGB 表示每个纹素都有红、绿、蓝通道，但是没有 alpha 通道。格式 gl.LUMINANCE_ALPHA 表示有亮度通道和 alpha 通道，而格式 gl. LUMINANCE 表示只有亮度通道，格式 gl.ALPHA 表示只有 alpha 通道。

第二列表示 gl.texImage2D()的第五个参数的类型。当类型为 gl.UNSIGNED_BYTE 时，表示格式中的每个通道都占用一个字节。第三列表示每个纹素占用的内存大小。

第二列中的其他类型为纹素数据提供了一种紧凑的表示法。类型 gl.UNSIGNED_SHORT_4_4_4_4 表示 RGBA 格式中的每个通道占用 4 位。gl.UNSIGNED_ SHORT_5_5_5_1 类型表示红、绿、蓝都占用 5 位，但是 alpha 通道只占用一位。最后，gl.UNSIGNED_SHORT_5_6_5 类型要与 gl.RGB 格式一起使用，它表示红色用 5 位、绿色用 6 位，蓝色用 5 位表示。

把纹理上传给 GPU 时，就是把纹理保存在 GPU 可以用于纹理的内存中。这是一个特殊的视频内存，GPU 访问它的速度比访问普通的系统内存要快许多，但是对于某些系统，它是专供 GPU 使用的普通系统内存的一部分。在纹理上传过程中，经常执行一个名为纹理重组(swizzling)的进程。纹理重组是指为了提高缓存的性能而对内存中纹理的字节进行重新排列的技术。

5.3.5 定义纹理参数

有几个参数需要用户来设置，它们会影响绘图期间的纹理处理。这些参数都是用 gl.texParameteri()方法设置的。本节介绍这个方法和其参数，但是在本章的其余部分中你将逐步了解这些参数的意义。

这个方法的原型如下：

```
void texParameteri(GLenum target, GLenum pname, GLint param)
```

该方法的参数可以取以下值：

- target 可以取 gl.TEXTURE_2D 或 gl.TEXTURE_CUBE_MAP 值。
- pname 定义我们想要设置的目标参数。它可以取以下值：
  - gl.TEXTURE_MAG_FILTER
  - gl.TEXTURE_MIN_FILTER
  - gl.TEXTURE_WRAP_S
  - gl.TEXTURE_WRAP_T
- param 定义了目标参数的值。param 可以设置的值取决于第二个参数 pname 的定义。
  如果 pname 取 gl.TEXTURE_MAG_FILTER，则 param 可以取以下值之一：
  - gl.NEAREST
  - gl.LINEAR

  如果 pname 是 gl.TEXTURE_MIN_FILTER，则 param 可以取以下值之一：
  - gl.NEAREST
  - gl.LINEAR
  - gl.NEAREST_MIMAP_NEAREST
  - gl.LINEAR_MIMAP_NEAREST
  - gl.NEAREST_MIMAP_LINEAR
  - gl.LINEAR_MIMAP_LINEAR

  如果 pname 是 gl.TEXTURE_WRAP_S 或 gl.TEXTURE_WRAP_T，则 param 可以取以下值之一：
  - gl.REPEAT
  - gl.CLMAP_TO_EDGE
  - gl.MIRROR_REPEAT

前面曾提到，这些参数的具体意义不在这里介绍，在本章中介绍相关主题时，再来讨论这个方法的用法。

5.3.6 理解载入纹理的完整过程

现在分析一个完整的源代码段，它创建一个纹理对象和一个 Image 对象，然后把图像数据载入 Image 对象中。下面的源代码段有几行代码前面还没有介绍过，接下来就将解释它们的作用：

```
function textureFinishedLoading(image, texture) {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
                image);

  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
```

```
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

    gl.bindTexture(gl.TEXTURE_2D, null);
  }

  function loadImageForTexture(url, texture) {
    var image = new Image();
    image.onload = function() {
      pwgl.ongoingImageLoads.splice(pwgl.ongoingImageLoads
                                      .indexOf(image), 1);
      textureFinishedLoading(image, texture);
    }
    pwgl.ongoingImageLoads.push(image);
    image.src = url;
  }

  function setupTextures() {
    // Texture for the table
    pwgl.woodTexture = gl.createTexture();
    loadImageForTexture("wood_128x128.jpg", pwgl.woodTexture);
  }
```

运行这段代码时，首先执行 3 个函数中的第一个，即 setupTextures()。我们通常在 WebGL 应用程序的设置和初始化部分调用这个函数。在这个函数调用中，用 gl.createTexture()方法创建一个纹理对象。然后调用 loadImageForTexture()函数，并且将纹理图像的相对 URL 作为第一个参数传递给它，把创建的纹理对象作为它的第二个参数。

loadImageForTexture()函数创建一个 Image 对象，它用来保存纹理数据。在这个函数中设置 onload 事件处理程序，并且为了解决丢失上下文问题，使用 ongoingImageLoads 数组跟踪当前正在载入的是哪个图像文件。

图像载入结束时会触发 Image 对象的 onload 事件，并且以 Image 对象为第一个参数、以纹理对象为第二个参数来调用 textureFinishedLoading()函数。在这个函数中，第一件事情是调用 gl.bindTexture()方法将此纹理对象绑定为当前纹理。然后，在把此纹理数据载入纹理对象之前，调用一个前面还没有介绍过的方法，如下所示：

```
  gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
```

gl.pixelStorei()方法会影响后续对 gl.texImage2D()的调用方式。如果 gl.pixelStorei()的第一个参数设置为 gl.UNPACK_FLIP_Y_WEBGL，第二个参数设置为 true，则用 gl.texImage2D()函数读取纹理图像时，图像会绕水平轴翻转。

翻转图像的原因是在 WebGL 中纹理使用的坐标系(所有 OpenGL 版本都如此)不同于 Image 对象使用的坐标系。在 WebGL 中，纹理的原点(0.0,0.0)在纹理的左下角，水平轴指向右方，垂直轴指向上方。

对于 Image 对象，原点(0.0,0.0)位于图像的左上角，水平轴指向右方，但是垂直轴指向下方。图 5-3 说明这两个坐标系。左边是 HTML DOM 类型的 Image 对象使用的坐标系，

右边是WebGL纹理使用的坐标系。

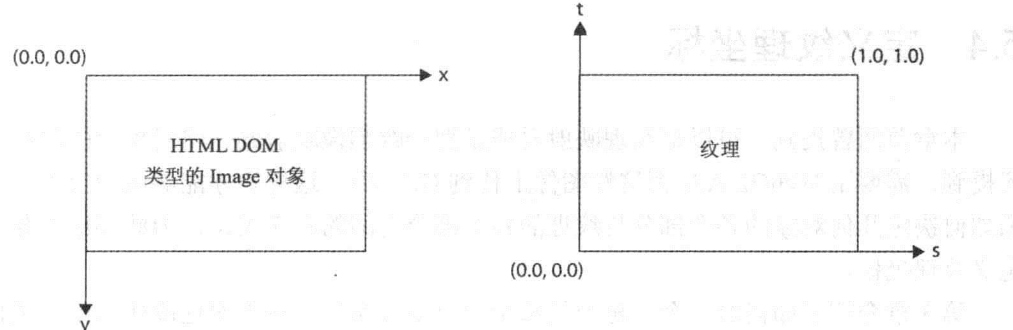


图5-3 HTML DOM Image对象和WebGL纹理的坐标系

在定义了纹理数据是否需要翻转之后，调用gl.texImage2D()把纹理数据上传到GPU。最后两次调用gl.texParameteri()方法，参数分别设置为gl.TEXTURE_MAG_FILTER和gl.TEXTURE_MIN_FILTER，如下所示：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

以上两条调用语句定义了当前纹理的纹素与屏幕上的像素没有一对一映射关系时纹理的映射方式。第一条调用语句定义纹理放大时(或伸展时)的纹理映射方式，第二条调用语句定义了纹理缩小时(或收缩时)的纹理映射方式。

在本章的后面将进一步介绍有关纹理伸展和收缩的内容。

5.3.7 创建一个纹理对象并载入纹理数据

下面归纳创建一个纹理对象并载入图像数据的步骤：

(1) 用gl.createTexture()方法创建一个WebGLTexture对象。

(2) 用new Image()方法创建一个HTML DOM类型的Image对象。

(3) 给Image对象的onload事件定义事件处理程序，这样就可以调用一个函数，它可以绑定刚创建的纹理对象，并且可把纹理数据上传到GPU。

(4) 把Image对象的src属性设置为希望绑定到纹理的图像的URL。

(5) 把图像数据载入Image对象的过程结束时，触发onload事件，此时我们就可以用gl.bindTexture()方法绑定这个纹理对象。

(6) 如果我们希望这幅图像在用作纹理时不要上下颠倒，则调用下面的代码：

```
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
```

(7) 绑定纹理对象后，就可以调用gl.texImage2D()方法把图像数据传给GPU。

(8) 用gl.texParameteri()方法定义纹理参数。具体地说，如果我们不想载入整个Mip映射图像链，则要把最小化过滤器(gl.TEXTURE_MIN_FILTER)设置为gl.NEAREST或gl.LINEAR。在下面的代码中，我们把它设置为gl.NEAREST：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

5.4 定义纹理坐标

本章前面曾提到，可以把纹理映射看成是把一幅图像映射到几何对象上的过程。此外还提到，需要用 WebGL API 把这幅图像上传到 GPU 中，这样它才能够成为纹理。本节介绍如何决定几何对象的各个部分与纹理的各个部分之间的对应关系，为此要给对象的顶点定义纹理坐标。

第 3 章介绍了如何给一个三角形的每个顶点定义颜色。在下面这段代码中，我们再次看到如何给 WebGLBuffer 对象设置顶点颜色(与第 3 章的代码相比，这段代码的内容已经更新，它可以处理丢失上下文问题)。

```
// This code snippet is listed here so you can
// comparare with setting up colors.

pwgl.triangleVertexColorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.triangleVertexColorBuffer);
var colors = [
          1.0, 0.0, 0.0, 1.0, //v0
          0.0, 1.0, 0.0, 1.0, //v1
          0.0, 0.0, 1.0, 1.0 //v2
      ];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors),
              gl.STATIC_DRAW);
pwgl.TRIANGLE_VERTEX_COLOR_BUF_ITEM_SIZE = 4;
pwgl.TRIANGLE_VERTEX_COLOR_BUF_NUM_ITEMS = 3;
```

除了给 WebGLBuffer 对象设置颜色外，这段代码还用 gl.bindBuffer()方法绑定缓冲，并且在调用 gl.drawArrays()和 gl.drawElements()方法之前调用 gl.vertexAttribPointer()方法，把缓冲连接到顶点着色器中的正确属性。这段代码已经在第 3 章中介绍过，在这里插入这段代码只是为了方便阅读：

```
// This code snippet is listed here so you can
// comparare with setting up colors.

gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.triangleVertexColorBuffer);
gl.vertexAttribPointer(pwgl.vertexColorAttributeLoc,
                       pwgl.TRIANGLE_VERTEX_COLOR_BUF_ITEM_SIZE,
                       gl.FLOAT, false, 0, 0);

gl.drawArrays(gl.TRIANGLES, 0, pwgl.TRIANGLE_VERTEX_COLOR_BUF_NUM_ITEMS);
```

在创建缓冲时我们曾经给顶点定义颜色，采用同样的方式，创建缓冲时我们可以给每个顶点定义一个纹理坐标。由于你已经知道如何给顶点定义颜色，因此如何给顶点定义纹理坐标就容易理解了。

下面这段代码设置纹理坐标：

```
// Setup buffer with texture coordinates
pwgl.triangleVertexTextureCoordinateBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.triangleVertexTextureCoordinateBuffer);
var textureCoordinates = [
  0.0, 0.0, //v0
  1.0, 0.0, //v1
  0.5, 1.0, //v2
];

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoordinates),
              gl.STATIC_DRAW);
pwgl.TRIANGLE_VERTEX_TEX_COORD_BUF_ITEM_SIZE = 2;
pwgl.TRIANGLE_VERTEX_TEX_COORD_BUF_NUM_ITEMS = 3;
```

这段代码与设置顶点颜色的代码几乎完全一样。WebGL 并不知道缓冲中的内容是颜色、纹理坐标还是其他数据。它只知道缓冲中包含数值。可以看出，将纹理坐标保存在名为 textureCoordinate 的数组中，然后把它上传给 WebGLBuffer 对象。

这段代码中并没有包含顶点位置缓冲，但是你可以想象，三角形有 3 个顶点位置，它们与纹理坐标的对应关系如图 5-4 所示。

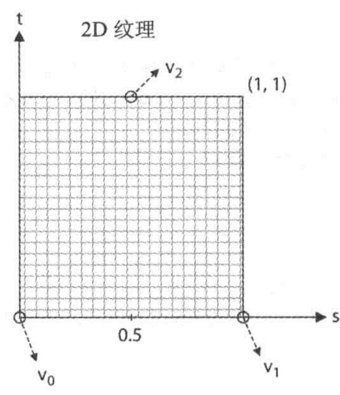


图 5-4 三角形的坐标纹理

创建纹理坐标缓冲的代码与创建颜色缓冲的代码之间存在一个微小的差别，这个差别就是它们使用的变量名不同，这样做只是为了方便你理解。还有其他微小的差别，如在本例中，每个顶点的颜色用 4 个浮点数来定义，而每个顶点的纹理用两个浮点数来定义。

在根据缓冲中的纹理坐标执行任何绘图之前，需要绑定纹理坐标缓冲，并调用 gl.vertexAttribPointer()方法定义纹理数据的组织方式，规定纹理坐标缓冲连接到顶点着色器中的哪个属性。此外，还需要确保已经用 gl.bindTexture()方法建立正确的纹理绑定。以下代码通常是某个绘图函数的一部分：

```
gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.tringleVertexTextureCoordinateBuffer);
gl.vertexAttribPointer(pwgl.vertexTextureAttributeLoc,
                       pwgl.TRIANGLE_VERTEX_TEX_COORD_BUF_ITEM_SIZE,
```

```
                        gl.FLOAT, false, 0, 0);

gl.bindTexture(gl.TEXTURE_2D, pwgl.woodTexture);

gl.drawArrays(gl.TRIANGLES, 0,
              pwgl.TRIANGLE_VERTEX_TEX_COORD_BUF_NUM_ITEMS);
```

在这段代码中，变量 pwgl.triangleVertexTextureCoordinateBuffer 表示 WebGLBuffer 纹理对象，它保存了每个顶点的纹理坐标(如本节前面所示)。gl.vertexAttribPointer()方法的第一个参数 pwgl.vertexPositionAttribLoc 包含了着色器程序中的属性位置，这个位置通常可以用 gl.getAttribLocation()方法获得，这在前面已经提到过多次。但是，这个方法的调用并没有出现在这段代码中。

5.5 着色器中的纹理处理

前面我们已经介绍如何创建纹理对象以及如何把纹理数据上传给 GPU。此外，我们还介绍如何在 WebGLBuffer 对象中定义纹理坐标。

在这一节中，我们要介绍如何在着色器中处理纹理。这里先重复介绍只处理普通颜色的顶点着色器和片段着色器：

```
<script id="shader-vs" type="x-shader/x-vertex">
  // vertex shader that uses colors, NOT textures
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;
  varying vec4 vColor;

  void main() {
    vColor = aVertexColor;
    gl_Position = vec4(aVertexPosition, 1.0);
  }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
  // Fragment shader that uses colors, NOT textures
  precision mediump float;

  varying vec4 vColor;
  void main() {
    gl_FragColor = vColor;
  }
</script>
```

顶点着色器从名为 aVertexColor 的属性得到颜色值，aVertexColor 属性来自 WebGLBuffer 对象，后者从 JavaScript 代码载入颜色值。顶点着色器通常只把线性属性赋给一个可变变量，然后在 WebGL 流水线中对可变变量的值进行线性插值运算，插值结果传送给片段着色器。

在片段着色器中，将可变变量 vColor 的值赋给一个特殊变量 gl_FragColor，这个变量保存了在片段着色器处理完成后的片段颜色。

回顾了前面的知识后，现在就介绍如何在顶点着色器和片段着色器中处理纹理，以下是相应的代码：

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec2 aTextureCoordinates;

  varying vec2 vTextureCoordinates;

  void main() {
    gl_Position = vec4(aVertexPosition, 1.0);
    vTextureCoordinate = aTextureCoordinates;
  }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  varying vec2 vTextureCoordinates;
  uniform sampler2D uSampler;
  void main() {
    gl_FragColor = texture2D(uSampler, vTextureCoordinates);
  }
</script>
```

从这段代码可以看出，使用纹理的顶点着色器代码与前面使用颜色的顶点着色器非常相似。但也存在些微的差别，即用来保存颜色信息的属性现在改为一个名为 aTextureCoordinates 的属性，后者的类型为 vec2，它的输入数据是纹理坐标。此外还有一个名为 vTextureCoordinates 的可变变量，它把纹理坐标传递给片段着色器。

在片段着色器中，读取可变变量 vTextureCoordinates 的值，然后从纹理对象读取纹素，并声明一个类型为 sampler2D 的特殊统一体变量 uSampler。这个采样器用来表示需要访问哪个纹理图像单元(在下一节中将进一步介绍纹理图像单元)。

采样器统一体的值来自 JavaScript 代码，用 gl.uniformli()方法设置。它的值必须与绑定纹理的纹理图像单元相对应。例如，假设有纹理绑定到纹理图像单元 gl.TEXTURE0 上，则要用 gl.uniformli()方法把 uSampler 统一体变量设置为 0。下面这段代码有助于你理解采样器与纹理图像单元之间的关系：

```
// Get the location of the uniform uSampler
pwgl.uniformSamplerLoc = gl.getUniformLocation(shaderProgram,
                                               "uSampler");
...

// Bind the texture to texture unit gl.TEXTURE0
gl.activeTexture(gl.TEXTURE0);
```

```
gl.bindTexture(gl.TEXTURE_2D, texture);

...

// Set uSampler in fragment shader to have the value 0
// so it matches the texture unit gl.TEXTURE0
gl.uniform1i(pwgl.uniformSamplerLoc, 0);
```

图 5-5 是一个概念图，说明了纹理化如何处理不同纹理单元。从图中可以看出，通过 gl.uniformlli()方法给采样器统一体变量 uSampler 设置值，从而可以选择应该使用哪个纹理图像单元。

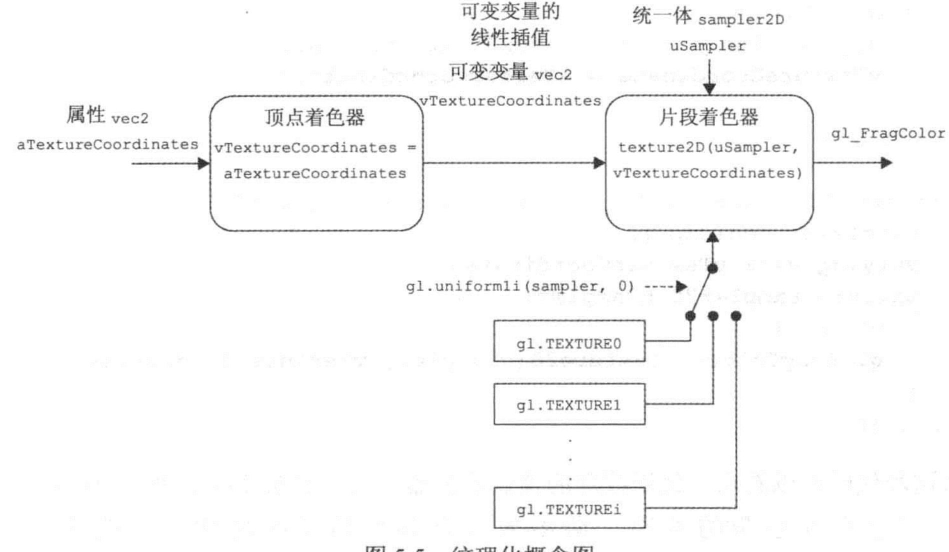


图 5-5　纹理化概念图

操作纹理图像单元

WebGL 可以有多个纹理图像单元(有时简称为纹理单元)，它们使用 gl.TEXTURE0、gl.TEXTURE1 和 gl.TEXTURE2 这样的名称。

系统支持的纹理单元数取决于 WebGL 的实现，但是用 gl.getParameter()方法和 gl.MAX_TEXTURE_IMAGE_UNITS 参数可以查询系统支持的纹理单元数，如下所示：

```
var nbrOfTextureImageUnits = gl.getParameter(gl.MAX_TEXTURE_IMAGE_UNITS);
```

在操作纹理时，总是选取其中一个纹理图像单元。调用 gl.activeTexture()方法可以设定要操作的纹理单元。例如，如果要操作 gl.TEXTURE1 纹理图像单元，则使用下面的代码：

```
gl.activeTexture(gl.TEXTURE1);
```

如果不调用 gl.activeTexture()方法设置活动纹理图像单元，默认时使用 gl.TEXTURE0 纹理图像单元。

调用以下函数可以判断哪个是活动纹理图像单元：

```
var activeTextureUnit = gl.getParameter(gl.ACTIVE_TEXTURE);
```

这里需要指出，这个函数的返回值并不是对于 gl.TEXTURE0 为 0，对于 gl.TEXTURE1 为 1，以此类推。相反，根据 WebGL 规范，它们的值定义如下：

```
/* TextureUnit */
const GLenum TEXTURE0                = 0x84C0;
const GLenum TEXTURE1                = 0x84C1;
const GLenum TEXTURE2                = 0x84C2;
const GLenum TEXTURE3                = 0x84C3;
const GLenum TEXTURE4                = 0x84C4;
const GLenum TEXTURE5                = 0x84C5;
const GLenum TEXTURE6                = 0x84C6;
const GLenum TEXTURE7                = 0x84C7;
const GLenum TEXTURE8                = 0x84C8;
const GLenum TEXTURE9                = 0x84C9;
const GLenum TEXTURE10               = 0x84CA;
const GLenum TEXTURE11               = 0x84CB;

...

const GLenum TEXTURE29 = 0x84DD;
const GLenum TEXTURE30 = 0x84DE;
const GLenum TEXTURE31 = 0x84DF;
```

对用户来说，不同纹理单元的确切值并不重要。但重要的是，如果想知道哪个纹理图像单元是活动的，然后根据这个结果执行相应的代码，则用户必须逐一测试这些符号值(如 gl.TEXTURE0、gl.TEXTURE1 等)，如下所示：

```
if (activeTextureUnit == gl.TEXTURE0) {

  // Do something

}
else if (activeTextureUnit == gl.TEXTURE1) {

  // Do something else

}
```

5.6 处理纹理过滤

假设有一个大小为 512×512 纹素的纹理，再假设将此纹理应用于一个正方形，这个正方形投影到屏幕上，它的大小与纹理的大小一样。则在这个情形下，应用纹理后的正方形看起来与原图非常相似。

在一般的 3D 场景中，对象离照相机距离有时很近，有时很远。根据透视投影，这意

味着当对象离照相机较近时，其在屏幕上的投影比较大；当对象离照相机较远时，其在屏幕上的投影较小。

如果投影对象在屏幕上占据的区域大于纹理原始图像的大小，则需要拉伸纹理，这就是所谓的纹理伸展(magnification)。与此相反，如果投影对象在屏幕上占据的区域小于纹理原始图像的大小，则需要减小纹理的大小，这就是所谓的纹理收缩(minification)。

我们用纹理过滤表示纹理伸展或收缩时计算片段颜色的一般过程。调用 gl.texParameteri() 方法可以控制纹理的过滤方式。前面已经介绍过这个方法的原型，为了方便起见，现在重复绘出这个方法的原型：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
```

为了定义纹理的伸展过滤器，需要把这个方法的第二个参数设置为 gl.TEXTURE_MAG_FILTER；为了定义纹理的收缩过滤器，需要把它的第二个参数设置为 gl.TEXTURE_MIN_FILTER。在下面几节中，我们将进一步介绍纹理的伸展和收缩以及其他可以使用的过滤器。

5.6.1 纹理伸展

图 5-6 说明了使用伸展过滤模式时像素与纹素之间的关系。纹理已经伸展或拉伸，因此一个纹素对应屏幕上的几个像素。

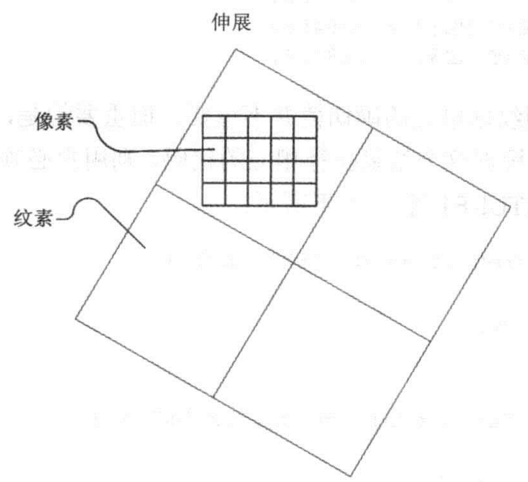


图 5-6 使用伸展过滤时，一个纹素对应多个像素

纹理的过滤过程实质上就是确定像素(实际上就是片段)颜色的过程。伸展纹理时，可以选择两种过滤模式。最基本的过滤模式是最近相邻模式，用 gl.NEAREST 名称指定。用以下代码把伸展过滤设置为最近相邻模式：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

最近相邻过滤模式表示把最接近纹素的颜色值赋给每个纹理坐标。这种过滤模式速度快，因为每个纹理坐标只需要使用一个纹素作为输入。

如果纹理伸展很多，这种过滤模式会产生块状效果(blocky appearance)，这有时也称为

像素化效果。仔细分析图 5-6 就会看出，其中许多像素落在同一个纹素中，因此许多像素显示同一种颜色。

伸展过滤的另一个过滤模式称为线性过滤或双线性过滤，用 gl.LINEAR 名称定义。根据线性过滤模式，每个纹理坐标的颜色不是来自单个纹素，而是它周围的 4 个纹素的加权平均值。为了把伸展过滤设置为线性过滤模式，需要调用 gl.texParameteri()方法，如下所示：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
```

线性过滤不会像最近相邻过滤模式那样产生块状效果，但线性过滤的特点是，当纹理拉伸时，会产生模糊效果。

5.6.2　纹理收缩

图 5-7 说明纹理收缩的一个示例。几个纹素对应于屏幕上的一个像素。这意味着，几个纹素影响屏幕上一个像素的颜色，但是实际上，很难确定影响每个像素的全部纹素。

对于纹理收缩，与纹理伸展一样，也可以选择同样的两个过滤模式：最近相邻过滤和线性过滤(在下一节中你就会知道，还有其他基于 Mip 映射的过滤模式)。

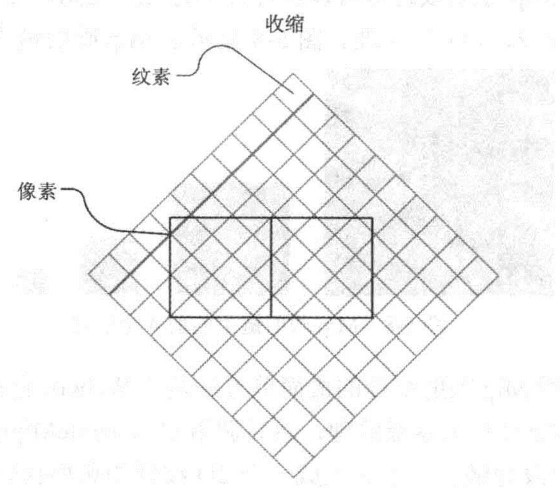


图 5-7　对于收缩纹理，每个像素对应多个纹素

用下面的调用语句设置纹理收缩的最近相邻过滤模式：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

最近相邻过滤模式的颜色计算过程与纹理伸展的最近相邻过滤模式相同。纹理坐标的颜色由最接近纹素的颜色决定。当每个像素对应的区域覆盖很多纹素时，这种过滤模式会产生锯齿现象。

纹理收缩可以使用的另一个基本过滤模式是线性过滤。下面的代码为收缩纹理设置线性过滤模式：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
```

纹理收缩线性过滤与纹理伸展线性过滤的处理过程完全相同。将离纹理坐标最近的 4 个纹素的加权平均值作为此纹理坐标的颜色。线性过滤得到的效果略好于最近相邻过滤，但是当一个像素对应不止 4 个纹素时，这种过滤模式也会产生锯齿问题。

5.6.3 Mip 映射纹理

从前一节的讨论可知，当一个像素对应很多纹素时，不管是最近相邻过滤还是线性过滤都会产生锯齿问题。这个问题的一个解决方法是使用一个较小尺寸的纹理，这样每个像素不会对应很多纹素。但是对于场景中靠近观察者的对象，为了避免出现纹理伸展现象，你可能希望纹理具有较高的分辨率，纹理伸展会使纹理对象看起来有块状效果或模糊效果。

Mip 映射是这个问题的解决方法。除了基础纹理(即零级纹理)外，Mip 映射使用一系列的较小纹理。每个新纹理都是系列中前一个纹理的一半大小。这一系列纹理形成一个纹理链，即为 Mip 映射纹理链。与只应用基础纹理的情形相比，一个完整的 Mip 映射链大约要多占用 1/3 的内存空间。

纹理不必是正方形，但是纹理链上的纹理必须一直继续到最后一个纹理的大小为 1×1 为止。作为一个示例，Mip 映射纹理链可以包含大小为 256×256、128×128、64×64、32×32、16×16、8×8、4×4、2×2、1×1 的纹理。图 5-8 显示了 Mip 映射链上的前 4 幅图像。

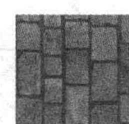


图 5-8　Mip 映射链上的前 4 个纹理

在 WebGL 中应用 Mip 映射纹理的最简单方法是让 WebGL 自动生成 Mip 映射纹理链。我们只需要载入基础纹理作为零级纹理，然后调用 gl.generateMipmap()方法，WebGL 会自动生成整个 Mip 映射纹理链。以下是生成一个 2D 纹理的典型代码：

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
              image);
gl.generateMipmap(gl.TEXTURE_2D);
```

另一个方法是用一个图像编辑工具或专门为生成 Mip 映射纹理而设计的纹理工具，离线生成 Mip 映射链上的全部纹理图像。然后，在生成 Mip 映射链上的全部图像后，用 gl.texImage2D()方法载入每幅图像，这个方法的第二个参数定义了图像的 Mip 映射级。以下代码载入 Mip 映射 0 级至 Mip 映射 4 级：

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
              imageLevel_0);

gl.texImage2D(gl.TEXTURE_2D, 1, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
```

```
                    imageLevel_1);

    gl.texImage2D(gl.TEXTURE_2D, 2, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
                    imageLevel_2);

    gl.texImage2D(gl.TEXTURE_2D, 3, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
                    imageLevel_3);

    gl.texImage2D(gl.TEXTURE_2D, 4, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
                    imageLevel_4);
```

可以看出，第二个方法需要较多的操作。但是，如果对Mip映射链上的低分辨率图像有特别的要求，则会知道这种方法是很有用的。

为了使用Mip映射纹理，基础纹理的长和宽都必须是2的次方(即1、2、4、8、16、…、512、1024等)。

1. 如何选择纹理过滤模式

当我们为自己的纹理创建一个完整的Mip映射纹理链后，除了gl.NEAREST和gl.LINEAR这两个普通的过滤模式外，我们还可以为纹理收缩选择4种新过滤模式。下面列出所有可以使用的过滤模式并对它们逐一作详细的介绍：

- gl.NEAREST——将最近相邻过滤模式应用于基础纹理，这是指使用最接近纹素的颜色作为纹理坐标的颜色。
- gl.LINEAR——将线性过滤(也称为双线性过滤)应用于基础纹理，这是指纹理坐标的颜色是由相邻4个纹素的颜色的加权平均值得到的。
- gl.NEAREST_MIPMAP_NEAREST——选择最近的Mip映射级，并且在这个Mip映射级中使用最近相邻过滤。
- gl.NEAR_MIPMAP_LINEAR——选择两个最近的Mip映射级，然后在每个Mip映射级中使用最近相邻过滤得到两个中间结果，对这两个中间结果进行线性插值得到纹理坐标的最终颜色。
- gl.LINEAR_MIPMAP_NEAREST——选择最近的Mip映射级，然后在每个Mip映射级中选择线性过滤。
- gl.LINEAR_MIPMAP_LINEAR——选择两个最近的Mip映射级，在这两个Mip映射级中，用线性过滤得到两个中间结果，对这两个结果使用线性插值，得到最终值。这个过滤模式在所有过滤中生成最好的过滤效果。有时也称为三线性过滤。

可以看出，4个Mip映射纹理过滤模式的名称都采用gl.A_MIPMAP_B的形式，其中A和B为NEAREST或LINEAR。这里的A定义在一种Mip映射级中使用纹素的方式。如果A为NEAREST，则表示在Mip映射级中使用最近相邻过滤。如果A为LINEAR，则表

示在 Mip 映射级中使用线性插值计算。

这里的 B 规定是使用单个 Mip 映射级还是使用两个最近的 Mip 映射级。如果 B 是 NEAREST，则使用最近的 Mip 映射级。如果 B 是 LINEAR，则表示使用两个最近的 Mip 映射级，在这两个 Mip 映射级中，由 A 规定过滤模式。然后把两个 Mip 映射级的结果进行线性插值得到最终的结果。

2. 应用 Mip 映射纹理可以得到更好的性能

虽然使用 Mip 映射纹理的主要理由是它的视觉效果，但是使用这种纹理还可以得到更好的性能。原因是，当使用 Mip 映射纹理时，一般情形下缓存的利用率会比较高，因为使用纹理收缩并使用比较“粗糙”的纹理时，纹素的读取往往发生在内存中比较靠近的单元中。

如果不用 Mip 映射纹理，则在收缩纹理时，需要从内存中相隔很远的单元读取纹素。这会降低需要读取的纹素已经出现在靠近 GPU 的缓存中的可能性。

如果用 gl.LINEAR_MIPMAP_LINEAR 参数定义三线性过滤模式，则比起 gl.LINEAR_MIPMAP_NEAREST 等过滤模式，GPU 需要执行更多的运算，需要从内存中读取更多的纹素。最好在不同的设备上测试不同的过滤模式，选择一种性能和质量都比较好的过滤模式。

5.7 理解纹理坐标包装

前面已经提到，纹理坐标是这样定义的：不管纹理是否是正方形，纹理的左下角的坐标为(0,0)，右上角的坐标为(1.1)。但是，实际上用户可以为这个范围之外的 4 个几何对象定义纹理坐标。

如果用户在这个范围之外定义纹理，则 WebGL 根据纹理包装模式处理纹理。利用 gl.texParameter()方法，可以分别为 s 方向和 t 方向定义纹理的包装模式。这个方法的第二个参数可以取 gl.TEXTURE_WRAP_S 或 gl.TEXTURE_WRAP_T 两个值之一。该方法的第三个参数定义包装模式，它们是：

- gl.REPEAT
- gl.MIRRORED_REPEAT
- gl.CLMAP_TO_EDGE

下面几节将详细讨论这些包装模式。

5.7.1 应用 gl.REPEAT 包装模式

如果没有显式定义一个包装模式，则 WebGL 的默认包装模式是 gl.REPEAT。在这个模式中，WebGL 在纹理坐标的每个整数边界上重复这个纹理。在 WebGL 采样纹理之前，它删除这个纹理坐标的整数部分。如果纹理应用的几何对象比纹理的输入图像大好几倍，则这种包装模式就非常有用，因为如果拉伸纹理以使它覆盖整个几何对象，则生成的效果可能会很差。包装模式利用称为平铺的方案把纹理应用于几何对象上。

平铺是计算机图形中一个的通用术语，它常用来表示把一个小纹理重复地作用于一个大对象的模式。其中一个示例是一块很大的草地。假如纹理的大小只占整块草地的一小部分，则重复使用这个纹理，使整个草地都被纹理覆盖。

为了得到比较好的效果，我们通常希望纹理的边界很好地融合在一起，这样不同纹理之间存在的缝隙尽可能不会显示出来。

图 5-9 说明把一个小型 2D 纹理应用于一个较大几何对象的示例。在上方，几何顶点都给出了纹理坐标，它们是(0,0)、(1,0)、(1,1)、(0,1)。结果是纹理被伸展，因此当小纹理被拉伸时，几何对象表面的纹理效果看起来与原来的纹理相差很大。在下方，几何顶点的旁边给出了纹理坐标(0,0)、(3,0)、(3,3)和(0,3)。在这种情形下，应用于几何对象上的纹理没有受到拉伸。

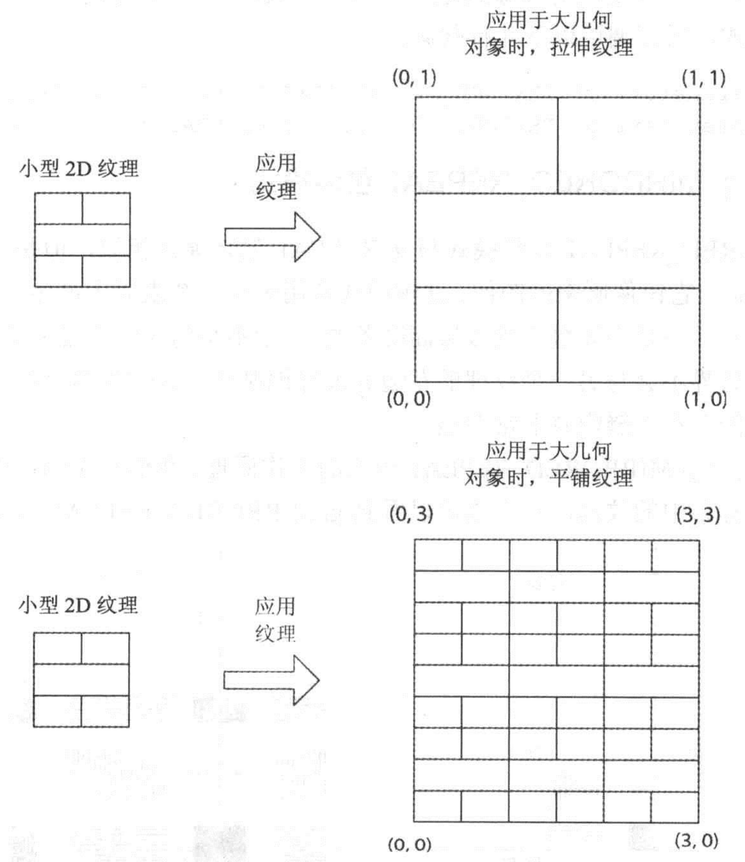


图 5-9 在[0,1]范围之外定义纹理坐标的效果

图 5-10 更详细说明了 gl.REPEAT 模式。图的左侧是在没有发生包装时在一个坐标中的纹理。在图的右侧可以看到 gl.REPEAT 模式的效果。

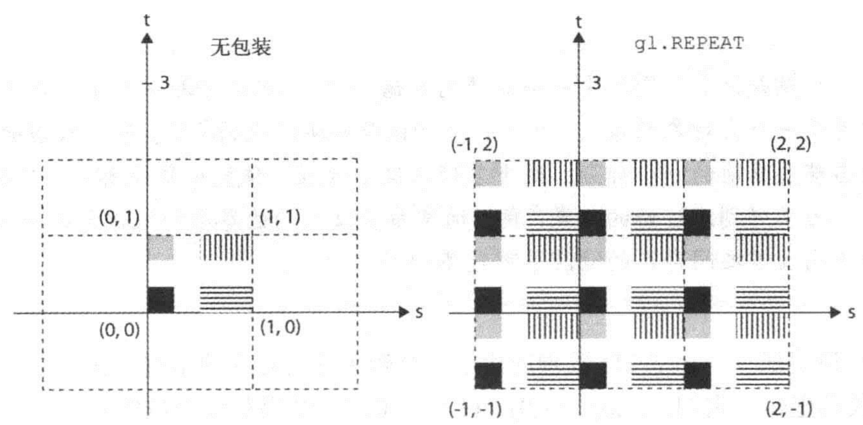


图 5-10 左侧的纹理没有包装模式，右侧是同一个纹理具有包装模式时的结果，其中采用 gl.REPEAT 包装模式

为了对当前纹理绑定的对象显式地设定包装模式，且在 s 方向和 t 方向上都设置为 gl.REPEAT 模式，需要调用以下几行代码：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
```

5.7.2 应用 gl.MIRRORED_REPEAT 包装模式

gl.MIRRORED_REPEAT 包装模式与 gl.REPEAT 包装模式相似，但是当纹理坐标的整数部分是偶数时，它镜像原来的图像。这种模式常用来在一个表面上产生无缝的平铺效果，而这个纹理原来并不是为无缝平铺效果而准备的。所谓不是为无缝平铺效果而准备的纹理，是指纹理的右边界不会与另一个纹理的左边界很好地融合，因此当两个纹理并排放置在一起时，它们之间存在的缝隙就非常明显。

图 5-11 说明 gl.MIRRORED_REPEAT 模式的工作原理。我们在左侧可以看到没有包装模式时纹理坐标系中的纹理，在右侧可以看到 gl.MIRRORED_REPEAT 效果。

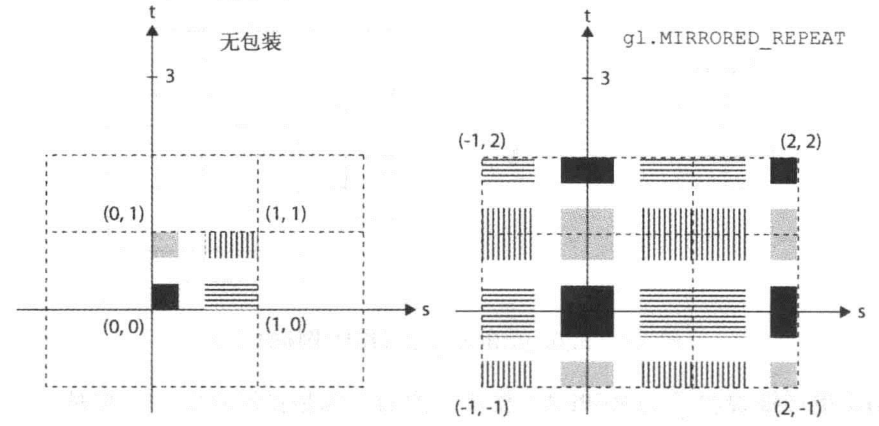


图 5-11 左侧的纹理没有包装模式，右侧是同一个纹理在具有包装模式时的结果，其中采用 gl.MIRRORED_REPEAT 包装模式

为了对当前纹理绑定的对象显式地设定包装模式，且在 s 方向和 t 方向上设置为 gl.REPEAT 模式，需要调用以下几行代码：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.MIRRORED_REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.MIRRORED_REPEAT);
```

5.7.3 应用 gl.CLAMP_TO_EDGE 包装模式

应用 gl.CLAMP_TO_EDGE 包装模式时，所有纹理坐标都嵌在[0,1]范围内。在这个范围之外的纹理坐标将由纹理的最接近边界采样得到。

图 5-12 说明 gl.CLAMP_TO_EDGE 包装模式的工作原理。在左侧，我们看到在纹理坐标系中的纹理没有包装模式。在右侧，我们看到 gl.CLMAP_TO_EDGE 包装模式的效果。

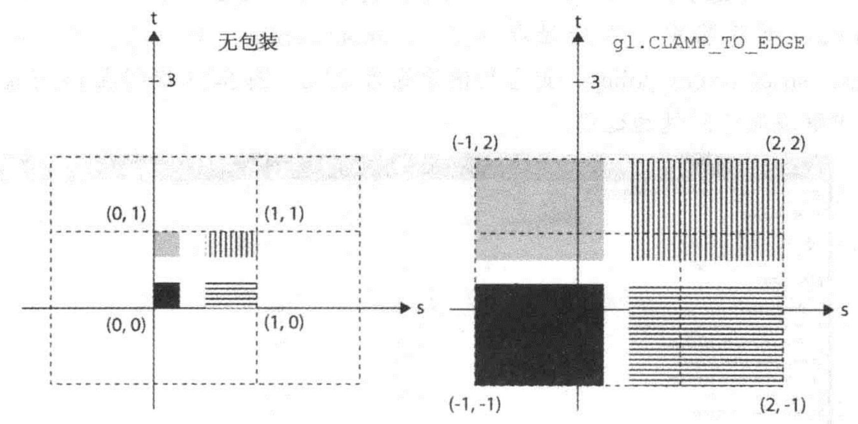


图 5-12 左侧的纹理没有包装模式，右侧是同一个纹理具有包装模式时的结果，其中采用 gl.CLAMP_TO_EDGE 包装模式

为了对当前纹理绑定的对象显式地设置包装模式，且在 s 方向和 t 方向上设置为 gl.CLAMP_TO_EDGE 模式，需要调用以下几行代码：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
```

5.8 一个完整的应用纹理示例

既然你已经知道了 WebGL 的纹理处理方法，现在就利用这些知识创建一个应用纹理的完整示例。程序清单 5-1 中的代码是以第 4 章的程序清单 4-1 为基础的，但是这里使用纹理而非基本颜色。

此外，为了解决丢失上下文问题，程序清单 5-1 在结构上稍微做了调整。这意味着没有给 WebGL 对象添加任何属性，相反给一个名为 pwgl 的全局对象添加所需要的属性。此外，还要仔细观察如何注册 webglcontextlost 事件和 webglcontextrestored 事件。这些事件的注册是在程序清单 5-1 末尾的 startup()函数中完成的。在 startup()函数中，我们还看到如何

应用来自 webgl-debug.js JavaScript 库的功能，因此可以模拟 webglcontextlost 事件和 webglcontextrestored 事件，本章开头对此进行过介绍。

默认时，大多数浏览器严格限制 JavaScript 对本地文件的操作权限。如果用户想通过在浏览器中打开一个本地 WebGL 应用程序的方法执行这个应用程序，则必须记住，大多数浏览器不会允许用户从文件系统中上传纹理图像。当你开发一个包含纹理处理的 WebGL 应用程序，并且要在本地对它进行测试时，就会遇到这个问题。如果你想用“另存为”保存在 Web 上找到的一个 WebGL 应用程序，然后运行这个应用程序，即从文件系统打开它，那么也会遇到这个问题。

如果你遇到这个问题，一个解决方法是修改本地文件的默认处理方式。在 Firefox 浏览器中，在地址栏中输入 about:config，然后导航到 security-fileuri.strict_origin_policy，把它的值设置为 false。图 5-13 说明在 Firefox 浏览器中解决这个问题的过程。

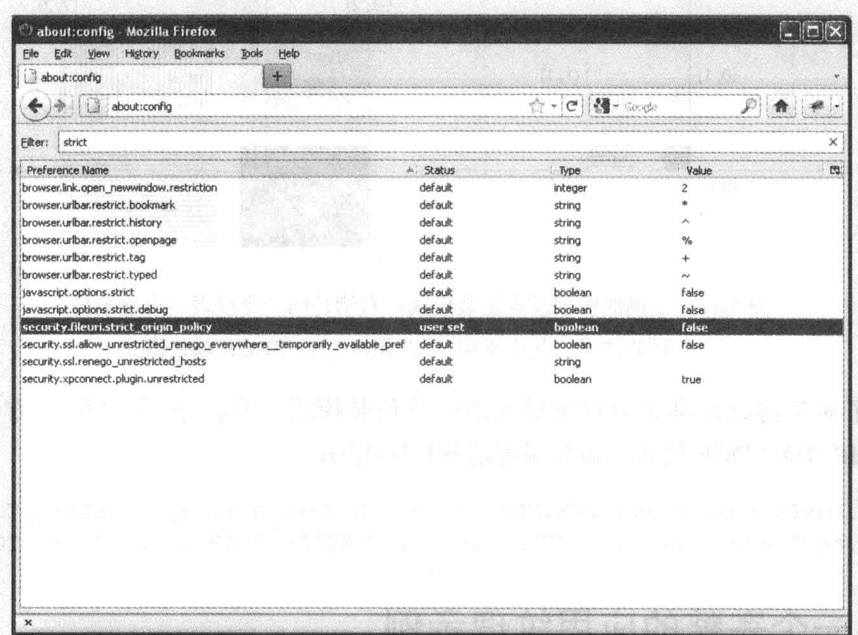


图 5-13　在 Firefox 地址栏中输入 about:config，可以修改浏览器的设置，它允许我们从本地文件系统载入一幅纹理图像。

在 Chrome 浏览器中，用下面的命令行开关启动 Chrome:

```
--allow-file-access-from-files
```

图 5-14 显示把程序清单 5-1 载入浏览器后的结果。把图 5-14 与图 4-14 进行比较，就会发现在应用纹理后，其结果比起程序清单 4-14 生成的结果更加有真实感。

程序清单 5-1 使用一个以前未曾介绍过的小型 JavaScript 实用工具库。从 Wrox.com 网

站下载程序清单 5-1，仔细分析这个程序(由于这个程序清单太长，本书没有给出它)。这个实用工具库的名称是 webgl-utils.js，它用来实现绘图循环。这个库提供了 requestAnimFrame() 和 cancelRequestAnimFrame()函数。

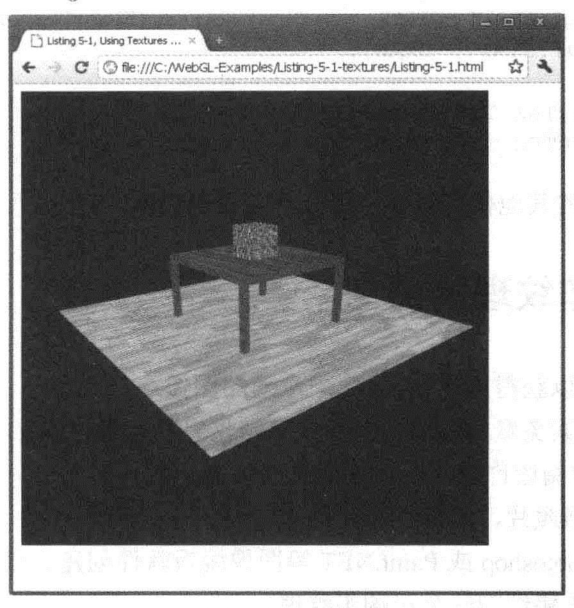


图 5-14　对应于程序清单 5-1 的场景

第 6 章将深入讨论绘图循环和这个库的作用，其中将详细介绍动画技术。虽然这个程序中没有动画对象，但是本章之所以引入这个库，是因为这里需要载入纹理图像。

由于当我们把一个 URL 赋给 Image 对象的 src 属性时，此图像数据会以异步方式载入这个 Image 对象中，因此在绘制场景时就会存在一个竞态条件。如果这个场景是在纹理图像载入之前绘制的(即执行 draw()函数)，则这个纹理不会正常显示。

竞态条件是指某个过程的输出结果(在这里即是场景绘制)取决于两个或两个以上事件的顺序或时序。这里的两个事件分别是图像数据载入结束和 draw()函数调用。

另一个需要考虑的问题是如何给地板定义纹理坐标。只有处于[0.0,1.0]范围之外的纹理坐标才会使用纹理包装。这意味着，地板纹理不需要放大很多。如果把 floorVertexTextureCoordinates 缓冲中的纹理坐标从 2.0 改为 1.0 并用浏览器观看其结果，就会看到地板的纹理需要进一步放大：

```
pwgl.floorVertexTextureCoordinateBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.floorVertexTextureCoordinateBuffer);
var floorVertexTextureCoordinates = [
    2.0, 0.0,
    2.0, 2.0,
```

```
        0.0, 2.0,
        0.0, 0.0
    ];

    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(floorVertexTextureCoordinates),
                  gl.STATIC_DRAW);

    pwgl.FLOOR_VERTEX_TEX_COORD_BUF_ITEM_SIZE = 2;
    pwgl.FLOOR_VERTEX_TEX_COORD_BUF_NUM_ITEMS = 4;
```

程序清单 5-1 中的其他代码对你来说并不会很难理解。

5.9 获得用作纹理的图像

有好几个方法可以获得用作 WebGL 中纹理的图像：

- 在 Web 上搜索免费的纹理。有很多网站收集了大量的纹理，供用户免费使用。但是，需要仔细阅读许可，明白使用这些纹理的条件。
- 用照相机拍摄照片，然后把照片用作纹理。
- 用 GIMP、Photoshop 或 Paint.NET 等图像编辑软件创建自己的图像。
- 向一些专门销售纹理的公司购买纹理。

5.9.1 下载免费纹理

Web 上有很多网站允许我们免费下载纹理。这些网站经常提供一些很不错的纹理图像供用户下载，允许用户将其用在自己的项目中。但是，必须仔细阅读许可，明白使用这些纹理的条件。

这些许可规定通常允许用户在自己的 2D 或 3D 模型中使用纹理而无须支付任何费用。然而，通常这些网站也规定，不允许用户把这些纹理以独立文件的形式销售或发行，这些单独的纹理文件可能会对原网站构成竞争威胁。

如果你要查找这些免费的纹理，有两个网站值得一看：

- www.cgtextures.com
- www.texturing.com

另外，查找免费纹理的最好方法可能是在 Web 上使用搜索引擎。

本书使用的纹理都来自 www.cgtextures.com 网站，这个网站也有一些非常不错的纹理，这些纹理的分辨率远远高于本书使用的纹理的分辨率。

5.9.2 用自己拍摄的照片生成纹理

即使从免费网站上可以下载大量各种类型的纹理，但是用户的项目可能需要某些特殊

的纹理，也许用户想使用以前没有人想到过的纹理，或者只想用自己的照片告诉朋友，这些纹理是你自己创建的。

不管是哪个理由，本节将介绍如何用数码照相机拍摄用作纹理的照片：

- 把照相机直接对准拍摄对象的表面。通常，将作为纹理的图像不需要透视效果。
- 不要将镜头调得太近，否则能够拍摄到的目标区域会很小，很小的区域使照片平铺变得很困难。
- 避免在很暗的环境中拍摄照片。如果拍摄时光线不足，则拍摄得到的照片会模糊，从而很难用作纹理。
- 避免拍摄阳光下的金属对象。金属具有很高的反射率，在阳光下会产生很强的高光。平铺这样拍摄到的照片纹理会很难看。

5.9.3 绘制图像

如果用户曾经用过 GIMP、Photoshop、Paint.NET 等图像编辑工具，或者想学习它们的用法，则用这些工具绘制纹理是值得一试的作法。用户可以大胆地尝试，在绘制图像时可以尽情发挥自己的想象力，或者从 Web 上下载这些工具的使用教程。有许多操作教程，详细介绍了这些软件的使用，帮助您绘制纹理图像。

5.9.4 购买纹理

除了下载免费的纹理外，用户还可以购买纹理。有些网站既提供免费纹理，也向用户出售纹理。用户可以这些网站下载几个纹理，或者成为其会员，可以一次性下载大量的纹理。

用户可能会问，既然有许多网站可以免费下载纹理，为什么还要花钱购买纹理呢？有时我们购买到的纹理分辨率更高，质量更好。用户需要决定是否值得为自己的项目购买纹理。

5.10 同域策略与跨域资源共享

同域策略(same-origin policy)是一个与 Web 应用程序有关的重要安全概念。简而言之，在浏览器中运行的一个脚本不允许从另一个域获取数据。对于两个资源(它们可以是脚本、文档、图像等)，如果包含这两个资源的文档满足以下条件，则可以认为它们是属于同域资源：

- 两个资源都源自于同一个域名(例如，两个资源都来自 example.com)。
- 两个资源都用同一种模式访问(如 http)。
- 两个资源通过同一个端口号访问(HTTP 的默认端口号为 80)。

这 3 个条件必须都满足，它们才被认为是来自同域的资源。表 5-2 给出了几个示例，进一步说明同域资源的概念。第一列是几个 URL，第二列说明这些 URL 是否与 www.example.com/page-1.html 属于同一个域。

表 5-2 同域策略的示例(表中的 URL 与 www.example.com/page-1.html 相比较)

包含第二个资源的文档的 URL	是否同域
http://www.example.com/page-2.html	是
http://www.example.com/dir/page-2.html	是
http://www.domain2-example.com/page-1.html	不是，域不同
https://www.example.com/page-1.html	不是，访问的方式不同。注意，https 的访问方式不同于 http
http://www.example.com:8080/page-2.html	不是，端口不同

5.10.1 同域策略应用于一般的图像

同域策略适用于几类不同的资源。本节介绍它对图像的影响，因为图像属于与纹理有关的资源。但是，首先会介绍它对一般图像的影响，然后具体介绍它对在 HTML5 画布上绘制的图像的影响。

来自一个域的 HTML 页面可以显示来自另一个域的图像，其 HTML 代码如下：

```
<img src="http://otherdomain.com/image.gif" />
```

但是，不允许 JavaScript 代码从另一个非 JavaScript 域中读取图像的像素。例如，假设有一个 JavaScript 程序，它创建一个 Image 对象，从另一个域(不同于脚本所在的域)读取图像数据并放入这个 Image 对象，最后把这个 Image 对象绘制在 HTML5 画布上。

根据同域策略，这是允许的行为，但是这幅图像在用 context2D.drawImage()方法绘制在画布上后，会对这个画布作一个标志，从而浏览器不允许脚本用 canvas.toDataURL()方法对画布中的数据做序列化处理。下面这段代码说明了这个思想：

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Example of Same Orgin Policy</title>
<script type="text/javascript">
// Assume that this document is loaded from http://example-domain.com
function draw() {
  var canvas = document.getElementById("canvas");

  if (canvas.getContext) {
    var context2D = canvas.getContext("2d");

    // Create an Image element
    var img = new Image();
    img.onload = function(){
      context2D.drawImage(img, 0, 0);
```

```
            // The call to canvas.toDataURL() is not allowed and
            // will generate a security error if the domain that the
            // image.jpg (see below) originates from is different from
            // the domain that this document originates from.
            var pixelData = canvas.toDataURL();
         }
      }

      img.src = 'http://other-domain.com/image.jpg';
   }

</script>
</head>
<body onload="draw();">
   <canvas id="canvas" width="300" height="300">

      Your browser does not support the HTML5 canvas element.

   </canvas>
</body>
</html>
```

canvas.toDataURL()方法可以把画布中的数据序列化为一幅图像。如果不用任何参数调用这个方法，则默认情况下，画布中的数据序列化为PNG格式。这个方法以data:URL模式返回一个地址，PNG图像以base64的编码字符串内联编码到这个地址中(data:URL模式是在RFC 2397标准中定义的)。base64编码方法把二进制数据编码成一个ASCII格式的字符串。这表示，如果以下调用成功：

```
pixelData = canvas.toDataURL();
```

则pixelData包含了如下格式的字符串：

```
"data:image/png;base64,iVBORw0KGgoAAA...ErkJggg=="
```

base64编码的数据是以“ivBORw”开始的。这个字符串可以赋给<img>标记的src属性，通过它显示在浏览器中。但是由于字符串中保存有图像数据，因此这个JavaScript代码成功调用canvas.toDataURL后也可以访问这幅图像的像素数据。

5.10.2 同域策略应用于纹理

在前一节中已经看到，可以把一幅跨域的图像读入一个Image对象中，并把它绘制在一个画布中。如果图像的域不是画布的域，则浏览器对这个画布作一个标志，表示从此之后不可能从画布读取像素。

如果把这个模式与 WebGL 进行比较，则相应的策略改为：允许跨域读取图像数据并放入一个纹理，然后在 WebGL 画布上绘制这个纹理。然而，在将跨域图像绘制在画布上后，不允许从画布读取图像的像素。这是最初的 WebGL 规范的规定，也是最初在浏览器中的实现模式。

然而，在这个规范的完善过程中，安全专家 Steve Baker 指出：当用户访问一个恶意网站时，专门编写一个特殊的片段着色器并上传给 GPU，就可以推理出纹理的内容。这个片段着色器通过测试纹素的颜色值测量运行的时间，根据这个时间可以计算此纹理中像素的亮度。为了说明现实中这种攻击是存在的，我们还专门设计了一个概念证明程序。

结果是，一般情况下，WebGL 不允许用 gl.texImage2D()或 gl.texSubImage2D()方法把跨域图像或视频上传为纹理。下面这段代码说明一个从 www.otherdomain.com 域载入图像的示例。如果这个域不同于用户下载画布的域(即来自 WebGLRenderingContext 的画布对象)，调用 gl.texImage2D()就会产生一个安全错误：

```
function textureFinishedLoading(image, texture) {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

// The call to texImage2D() will generate a security error
// if the image is loaded from another domain.

gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
              image);

  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

  gl.bindTexture(gl.TEXTURE_2D, null);
}

function loadImageForTexture(url, texture) {
  var image = new Image();
  image.onload = function() {
    pwgl.ongoingImageLoads.splice(pwgl.ongoingImageLoads.indexOf(image), 1);
    textureFinishedLoading(image, texture);
  }
  pwgl.ongoingImageLoads.push(image);
  image.src = url;
}

function setupTextures() {
  pwgl.woodTexture = gl.createTexture();

  // Load an image to be used as a texture from another domain
  loadImageForTexture("www.otherdomain.com/wood_texture.jpg",
                      pwgl.woodTexture);
}
```

如果需要使用保存在另一个域上的图像或视频，则利用跨域资源共享(Cross-Origin Resource Sharing，CORS)与保存有图像和视频的服务器进行协作，引用这些图像和视频作为纹理。有关CORS的更多内容将在下一节中介绍。

5.10.3 跨域资源共享

如果我们想把保存在另一个域中的图像用作 WebGL 应用程序中的纹理，这完全是可行的。如果浏览器和保存媒体的服务器都支持 CORS，则可以用跨域资源共享实现浏览器与服务器之间的协作。

CORS 是不同域资源之间实现可控通信的一个通用方法，这个规范不是专门针对 WebGL 的，但它是跨域纹理的一个很好的解决方案。它基于以下事实：当浏览器想要通过脚本使用来自另一个域上的服务器中的资源时，它就在给服务器的请求中发送一个特殊的 HTTP 报头(名为 origin)。如果服务器允许，则服务器在响应中插入中另一个 HTTP 报头(即 Access-Control-Allow-Origin)并表示允许访问这个资源。

WebGL 的开发人员不难发现，可以用 CORS 请求服务器上的图像或视频。简单地说，在前面的代码中只需要添加一行代码就可以载入图像或视频。下面这段代码说明如何载入另一个域中的图像，这里使用的方法是在给 Image 对象的 src 属性设置图像的 URL 之前，把 Image 对象的 crossOrigin 属性设置为 anonymous(匿名)。这里高亮显示为实现 CORS 访问而添加的一行代码：

```
function textureFinishedLoading(image, texture) {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
  // The call to texImage2D() will now be allowed
  // provided that the server permitted the CORS request
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
                image);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

  gl.bindTexture(gl.TEXTURE_2D, null);
}

function loadImageForTexture(url, texture) {
  var image = new Image();
  image.onload = function() {
    pwgl.ongoingImageLoads.splice(pwgl.ongoingImageLoads.indexOf(image), 1);
    textureFinishedLoading(image, texture);
  }
  pwgl.ongoingImageLoads.push(image);
  image.crossOrigin = "anonymous";
  image.src = url;
}

function setupTextures() {
```

```
    pwgl.woodTexture = gl.createTexture();

    // Load an image to be used as a texture from another domain
    loadImageForTexture("www.otherdomain.com/wood_texture.jpg",
                        pwgl.woodTexture);
}
```

需要指出的是，当我们用 CORS 机制请求图像数据时，绘制纹理的 WebGL 画布并没有受到影响。这意味着，除了能够把 Image 对象作为纹理上传给 GPU 外，在把纹理绘制到画布后，还需要用 canvas.toDataURL()方法读取画布中的像素。

虽然 CORS 不是专为 WebGL 设计的，但是用户有必要知道，CORS 的支持可以帮助通过 canvas.toDataURL()方法读取画布上的像素。下面这段代码说明如何从一个画布读取数据，在该画面中已经绘制来自另一个域的图像：

```
var canvas = document.getElementById("canvas");

  if (canvas.getContext) {
    var context2D = canvas.getContext("2d");

    // Create an Image element
    var img = new Image();
    img.onload = function(){
      context2D.drawImage(img, 0, 0);

      // The call to canvas.toDataURL() is now allowed provided that
      // the server permitted the CORS request
      var pixelData = canvas.toDataURL();
    }
  }
img.crossOrigin = "anonymous";
img.src = "http://other-domain.com/image.jpg";
```

5.11 小结

本章介绍了什么是纹理以及它在 WebGL 中的作用。此外，本章还介绍了如何通过解决上下文丢失问题提高 WebGL 应用程序的健壮性。

本章还介绍了在 WebGL 中绘图时使用纹理的几个步骤。

本章介绍了纹理过滤，以及什么是纹理伸展和纹理收缩，还介绍了当纹理收缩时 Mip 映射技术如何改善场景的质量和绘制性能。你已经掌握了纹理的 3 种包装模式：

- gl.REPEAT
- gl.MIRRORED_REPEAT
- gl.CLAMP_TO_EDGE

你现在知道当纹理坐标超出[0,1.0]范围时这些包装模式如何影响纹理的应用。

此外，本章还介绍了在 WebGL 中使用纹理时同域策略的重要性，以及它对纹理的应用存在的限制。本章也介绍了什么是 CORS，如何在 WebGL 应用程序中应用 CORS 跨域纹理。

纹理和丢失上下文是 WebGL 中两个非常重要的问题。第 6 章将介绍如何在场景中创建动画。

第6章 动画与用户输入

本章主要内容：

- 如何在 WebGL 中创建动画
- 掌握 requestAnimationFrame()、setInterval()和 setTimeout()这三个函数之间的不同
- 如何在 WebGL 应用程序中测量帧频
- 掌握事件处理细节
- 如何处理键盘输入
- 如何处理鼠标输入

到目前为止，在本书介绍的示例中，场景中的所有对象都位于同一个位置。本章介绍如何利用 WebGL 的动画技术在 3D 场景中创建运动效果。

此外，本章还介绍如何让用户控制 WebGL 场景，即如何处理用户的按键动作或鼠标事件。由于在 Web 应用程序中用户的输入是根据 JavaScript 事件处理模式来处理的，因此为了让你对事件处理模式有一个比较深入的理解，本章会先详细介绍事件处理模式。

6.1 创建动画场景

动画是一种通过快速显示一系列图像(或帧)模拟运动的技术。一般说来，在 Web 上主要有以下两类动画：

- 声明式动画(declarative animation)，如层叠样式表(Cascading Style Sheets, CSS)动画或 SVG 的<animate>标记。这些动画都不需要脚本来处理。用户只需要说明元素如何运动，不需要自己生成动画的每一帧内容。
- 基于脚本的动画(script-based animation)，这种动画的每一帧内容都是由 JavaScript 脚本来更新的，而且是某个回调函数的执行结果。这类动画包括通过更新 DOM 元

素的样式对象、在 HTML5 2D 画布上绘制动画对象和用 WebGL 绘制动画图形而创建的动画。

除这两类动画外，还有其他类型的动画，如 GIF 动画、Flash 动画。但是，它们都是属于其他类型的动画，因为构成动画的各幅图像是包含 GIF 图像或 Flash 动画的文件的一部分。

如果你具有 Web 开发背景且熟悉 CSS 技术，则可以跳过这个注解。如果你还不了解 CSS，本注解会简单介绍这方面的内容。

CSS 是一个标准，用来描述一个文档的显示。它经常与 HTML 文档一起使用(但是它也可以与 XML 文档一起使用)。其思想是，HTML 描述文档的内容和结构，而 CSS 描述文档的显示，如文档的外观和格式化。

一个 CSS 样式表由一组规则组成，这些规则规定文档如何显示。每个规则都使用以下格式：

```
selector {
  property1: value1;
  property2: value2;
  ...
}
```

这里的 selector 说明此规则应用于 HTML 文档中的哪个元素或哪些元素。在 selector 之后是属性-值对列表，它们描述元素的显示方式。

在 CSS 样式表中，关于规则的一个非常简单的示例是：

```
h1 {
  color: red;
  text-align: center;
}
```

这个示例说明在<h1>标记中的文本必须采用红色，且必须居中。用这种方式可以定义不同的 DOM 元素出现在页面上的外观和位置。

通过 style 对象，也可以用 JavaScript 代码更新这些 DOM 元素的样式。这个 style 对象是所有 DOM 元素的一部分。作为一个示例，用下面的几行 JavaScript 语句可以更新一个 DOM 元素的位置：

```
e = document.getElementById('id');
e.style.left = parseInt(e.style.left)+10+'px';
```

创建基于 DOM 样式的动画的基本思想是更改 DOM 元素的样式对象。只需要重复执行上述的第二行代码，就可以使这个元素向右移动。通常，更新像上面这样的样式对象会立刻改变 DOM 对象的位置。但是，对于 CSS 过渡效果和 CSS 动画，我们可以定义一个对象如何实现光滑的过渡效果，即如何从一个初始位置运动到最终位置，而不需要借助于任何 JavaScript 代码。

> CSS 的内容很丰富，有很多图书专门介绍 CSS 技术。如果你想真正掌握这种技术(强烈建议你如此做)，必须认真阅读这些图书之一，或者参考网络上很多与这个主题有关的免费在线资源。可以从以下两个非常好的在线资源开始学习：
>
> - www.w3.org/Style/css/
> - www.w3schools.com/css/

为了使模型在 WebGL 场景中移动，需要重复绘制场景，而且在两次绘制场景间隔，需要更新动画模型的位置。这与 HTML5 2D 画布动画使用的技术完全一样，但是对于 2D 动画，在绘制每帧内容之前，我们通常只需要更新动画对象的 x 轴和 y 轴位置。

传统上，基于脚本的动画一直以来都是用 setTimeout()和 setInterval()这两个 JavaScript 方法之一创建的。这两个方法用来定义回调函数，我们希望在未来规定的时间(单位为毫秒)内调用这些回调函数。在回调函数中更新动画，然后再次绘制。重复地执行此过程就可以生成一个动画。用 setTimeout()和 setInterval()方法创建一个动画的详细过程将在下一节中介绍。

6.1.1　setInterval()和 setTimeout()的使用

如前所述，传统上基于脚本的动画是采用以下两个 JavaScript 方法之一创建的：

- setTimeout(codeToCall, timeoutInMilliseconds)
- setInterval(codeToCall, timeoutInMilliseconds)

这两个方法都是 JavaScript 窗口对象的一部分，这意味着它们是全局方法。setTimeout()方法在规定的时间后(单位为毫秒，由它的第二个参数确定)调用第一个参数表示的代码或函数。这个方法有一个返回值，如果把这个返回值作为参数传给 clearInterval()函数，则可以取消对调度函数或代码的调用。

setInterval()方法与 setTimeout()方法相似，但是它每隔一定的时间(由第二个参数定义)反复调用第一个参数中的代码或函数。此外，setInterval()方法也返回一个值，如果把这个值作为参数送给 clearInterval()函数，则可以取消调度函数或代码的执行。

作为一个示例，以下是一个非常简单的绘图循环结构，它建立在 setInterval()之上：

```
function draw() {

  // 1. Update the positions of the objects in your scene

  // 2. Draw the current frame of your scene

}
```

```
function startup() {
 // Do your usual setup and initialization
 setInterval(draw, 16.7);
}
```

startup()函数通常是在载入文档后调用的(或者在我们希望启动动画的任何时刻调用)。在该函数中，首先对应用程序进行一般性的设置和初始化，然后调用 setInterval()，并规定在一定的时间间隔内(如本例中的 16.7 毫秒)调用 draw()函数。在这个示例中之所以选择 16.7 毫秒，是因为通常 LCD 显示器的刷新频率不会大于 60Hz。这意味着，动画更新的速度快于每秒 60 次没有多大意义。根据这个值就可以得到帧时间的近似值，即 1/60Hz=0.0167 秒=16.7 毫秒。

6.1.2 使用 requestAnimationFrame()函数

虽然 Web 开发人员很久以前就可以用 setInterval()和 setTimeout()函数创建动画，但是最近人们推荐另一个实现基于脚本的动画的方式，即使用 requestAnimationFrame()函数。与 setInterval()和 setTimeout()函数一样，后者也是 HTML DOM 窗口对象的一部分。

强烈建议你使用 requestAnimationFrame()函数而不是 setInterval()和 setTimeout()函数创建 WebGL 场景的动画。

1. requestAnimationFrame()函数的优点

requestAnimationFrame()函数是专门为创建脚本式动画而设计的。有了一个专用的动画创建方法，浏览器供应商可以努力优化这个方法，使得它比传统的 setInterval()和 setTimeout()方法更适合于创建动画。

用 setInterval()或 setTimeout()方法创建动画时，需要确定动画更新的最佳频率。但是这个最佳频率对于动画的设计人员来说是很难确定的。

但是，浏览器是比较容易确定这个最佳帧频的。浏览器上可能会同时运行多个动画，这可能会影响这个帧频。在这种情形下，浏览器会降低所有动画的帧频，这样它们就以流畅但稍低的频率执行动画。

浏览器也可以减速更新或暂停更新在某个不可见选项卡中正在运行的动画。这样，用这个方法不仅可以改善动画的性能，还可以节省电能，后者对于移动设备尤为重要。电池寿命通常是移动设备的一个重要因素。

如果 WebGL 应用程序包含了网络通信或其他与定时控制有关但与绘图无关的任务，则用户可能希望这种频率控制建立在另一个定时控制机制之上，而不是基于 requestAnimationFrame()。其理由是，如果这个选项卡不可见且浏览器减速运行或者暂停调用回调函数，则用户可能仍然会希望网络活动或用户应用程序中的其他周期性活动继续下去。

2. 使用 requestAnimationFrame()方法

requestAnimationFrame()方法很容易使用。只需要把绘制回调函数作为它的第一个参数传入，就可以调用这个回调函数。假设回调函数是 draw()，则它的调用方法如下：

```
requestAnimationFrame(draw) ;
```

当 WebGL 应用程序调用这个回调函数时，就把当前时间作为参数传给它。当前时间在动画循环中很有用，因为它常用来计算绘制上一帧绘制后已经过去了多少时间，然后补偿由于帧频的波动而受影响对象的运动。在本章后面我们还将进一步学习这些内容。

下面这段代码在较高层面上说明，用户如何利用 requestAnimationFrame()方法构建动画循环：

```
function draw(currentTime) {

   // 1. Reqeuest a new call to draw the next frame before
   //    you actually start drawing the current frame.
   requestAnimationFrame(draw);

   // 2. Update the positions of the moving objects in your scene


   // 3. Draw your scene

}

function startup() {
   // Do your usual setup and initialization
   canvas = document.getElementById("myGLCanvas");
   gl = createGLContext(canvas);
   setupShaders();
   setupBuffers();
   setupTextures();

   draw();
}
```

注意 draw()函数的定义，它有一个表示当前时间的参数 currentTime，但是当从 startup()函数内部调用这个 draw()函数时，并没有给它传递任何实参。在许多其他编程语言中，这

是不允许的。但是在 JavaScript 中，可以用任意实参来调用一个函数，不管这个函数在定义时有多少个形参。

万维网联盟(W3C)已发布了一个工作草案，规定了方法 requestAnimationFrame()的应用细节。这个工作草案的官方名称是“Timing control for script-based animations(基于脚本动画的定时控制)”。如果你对这个规范感兴趣，可以在 www.w3.org/TR/animation-timing/网页上找到该规范。

工作草案是 W3C 标准的第一级成熟标志。一个工作草案意味着，这个标准已向用户发布，它可以供“社区”范围内查阅。另外，这也意味着，并非这个标准中的全部内容都已确定，可能某些内容需要修改。

至于 requestAnimationFrame()方法的实现，出现了一个问题，因为不同的浏览器供应商已经使用稍微不同的名称。一个实际的解决方法是使用 JavaScript 实用工具库 webgl-utils.js 中的一个小程序(shim，一种小型包装程序)。这个实用工具库最早由 Google 公司的开发人员编写的，在网络上可以找到这个库，例如：

```
https://cvs.khronos.org/svn/repos/registry/trunk/public/
webgl/sdk/demos/common/webgl-utils.js
```

在 webgl-utils.js 中，requestAnimationFrame()的支持跨平台的小程序采用如下格式：

```
/**
 * Provides requestAnimationFrame in a cross browser way.
 */
window.requestAnimFrame = (function() {
 returnwindow.requestAnimationFrame ||
        window.webkitRequestAnimationFrame ||
        window.mozRequestAnimationFrame ||
        window.oRequestAnimationFrame ||
        window.msRequestAnimationFrame ||
        function(/* function FrameRequestCallback */ callback,
                 /*
           DOMElement Element */ element) {
         return window.setTimeout(callback, 1000/60);
        };
})();
```

可以看出，如果没有找到其他方法，则使用 setTimeout()回退函数。需要注意的是，使用这个回退函数时，不需要把当前时间作为参数传递给一个已注册为回调函数的函数。另外还要注意，这个实用工具库命名该方法为 requestAnimFrame 而非 requestAnimationFrame()。其原因是，此标准只是 W3C 的工作草案，还需要进一步修订。

6.1.3　帧频不同引起的运动补偿

不管是 setTimeout()、setInterval()还是 requestAnimationFrame()，在 WebGL 应用程序中生成动画的最简单方法是更新每个运动模型的位置，即在新创建的一帧中设置每个对象的不同位置。这种方法在许多情况下是可行的，但是在某些情况下也存在一些缺点，例如：

- 我们移动对象的速度与帧频有关。这意味着在一些慢速帧中，对象移动的速度比其在快速帧中要慢。如果开发一个赛车游戏，我们当然不希望汽车跑得快只是由于这个游戏是在某个高速设备运行的，而跑得慢是由于此游戏是在低端移动设备上等较慢系统上运行的。
- 假设由于 CPU、GPU 负载以及内存总线的原因而引起帧频波动，则每一帧中对象移动的距离也会随之波动。这样会得到非常奇怪的动画。

下面这段代码说明，只要考虑每个新创建帧的绘图频率，就可以得到比较平稳的动画：

```
function draw(currentTime) {
   // 1. Reqeuest a new call to draw the next frame before
   //    you actually start drawing the current frame.
   requestAnimationFrame(draw);

   // 2. Calculate how to compensate for varying frame rate
   //    and then update the position of the moving objects.
   if (currentTime === undefined) {
     currentTime = Date.now();
   }
   if (pwgl.animationStartTime === undefined) {
     pwgl.animationStartTime = currentTime;
   }
   if (pwgl.y < 5) {
     // Move your object. In this specific example the movement is just
     // moving a box vertically from the position where y = 2.7 to y = 5
     // The movement should take 3 seconds.
     pwgl.y = 2.7 + (currentTime - pwgl.animationStartTime)/3000 * (5.0-2.7);
   }

     // 3. Draw your scene


}

function startup() {
   // Do your usual setup and initialization

   canvas = document.getElementById("myGLCanvas");
   gl = createGLContext(canvas);
   setupShaders();
   setupBuffers();
   setupTextures();
   gl.clearColor(0.0, 0.0, 0.0, 1.0);
```

```
    gl.enable(gl.DEPTH_TEST);

    pwgl.x = 0.0;
    pwgl.y = 2.7;
    pwgl.z = 0.0;
    pwgl.animationStartTime = undefined;

    draw();
}
```

6.1.4 创建 FPS 计数器测量动画的平稳性

当 WebGL 应用程序中有一个动画场景时，通常我们希望这个动画看起来越平稳越好。定义每秒绘制多少帧(Frames Per Second，FPS)可以衡量动画的性能或动画的平稳程度。较高的 FPS 意味着较好的性能，即一般所谓的动画比较平稳。记住，通常我们不需要比每秒 60 帧更快的 FPS，因为大多数 LCD 显示器的更新频率为 60Hz。

基本上，60Hz 的频率相当于 60 FPS。Hz 单位常用来定义硬件(如显示器)的频率，而当我们谈论应用程序时，常用 FPS 单位。

理论上，FPS 值比较容易计算。我们只需要测量绘制每帧所需要的时间。例如，在一帧的开始时检查时间并保存下来。在下一帧中，回到代码中的同一个地方，再次检查时间。第二次时间与第一次时间之差就是帧时间。FPS 值是帧时间的倒数。这表示有以下的基本公式：

$$FPS = \frac{1}{\text{帧时间}}$$

如果需要在应用程序中执行这个计算，并且想要在屏幕上显示计算结果，则有两个方案供选择：

- 实时显示 FPS 值，即先测量绘制每帧所需要的时间，然后直接计算帧时间的倒数值，得到 FPS 值并显示。
- 显示某个形式的 FPS 平均值。求平均值也有好几种方法。

每更新一帧就实时计算 FPS 值的缺点是它可能变化太频繁。假设应用程序的运行频率为 30 FPS，这意味着这个显示值每秒要改变 30 次。如果我们每帧都更新屏幕的 FPS 值，则得到闪烁的显示效果，用户很难看清楚 FPS 的实际值。

因此，通常较好的方法是计算并显示某种类型的平均值。例如，可以使用移动平均值。另一个更好的方法是计数每秒内绘制的帧，当一秒时间过去后，则把显示的帧数作为 FPS 值并显示。然后重新开始计数帧，又经过一秒钟后，再显示这个计数值，以此类推。

下面这段代码说明了如何在场景绘制函数的开头和结束添加一些代码以实现此功能：

```
function draw(currentTime) {
  pwgl.requestId = requestAnimFrame(draw);
  if (currentTime === undefined) {
    currentTime = Date.now();
  }

  // Update FPS if a second or more has passed since last FPS update
  if(currentTime - pwgl.previousFrameTimeStamp >= 1000) {
    pwgl.fpsCounter.innerHTML = pwgl.nbrOfFramesForFPS;
    pwgl.nbrOfFramesForFPS = 0;
    pwgl.previousFrameTimeStamp = currentTime;
  }

// Draw the scene
...
// When a frame is completed, update the number of drawn
// frames to be able to calculate the FPS value
pwgl.nbrOfFramesForFPS++;
}
```

在 1 秒钟内，利用 pwgl.fpsCounter 的 innerHTML 属性把 FPS 的值设置为在过去一秒钟时间内绘制的帧数。

下面这段代码显示如何用 startup()函数初始化计算 FPS 所需要的几个变量。另外，在 HTML 代码中添加了一个<span>元素，它有一个唯一的 id。通过它来显示 FPS 值，即利用 document.getElementById()函数得到这个<span>的引用，然后利用 innerHTML 属性设置 FPS 值，如下所示。

```
<script type="text/javascript">

  ...

function startup() {

  ...

  pwgl.nbrOfFramesForFPS = 0;
  pwgl.previousFrameTimeStamp = Date.now();
  pwgl.fpsCounter = document.getElementById("fps");
draw();
}
</script>

  </head>

<body onload="startup();">
  <canvas id="myGLCanvas" width="500" height="500"></canvas>
  <div id="fps-counter">
```

```
        FPS: <span id="fps">--</span>
      </div>
    </body>

    </html>
```

6.1.5 用 FPS 作为测量值的缺点

即使 FPS 是测量图形性能的最常用方法，但是如果用户不加以注意，使用这种测量方法就会存在一个缺点，即 FPS 与帧绘制所需的时间不成线性关系。

假设，你有一个 WebGL 应用程序，它的帧频为 60 FPS。你修改了代码(例如，在场景中添加一些图形对象)，使这个 FPS 值降到 50 FPS，这会带来什么问题呢？显然，这个应用程序丢失了 10 个 FPS，但是绘制这个场景中添加的图形对象会增加多少时间？由下面的计算可以得到答案：

$$\frac{1}{50}-\frac{1}{60}\approx 0.02-0.0167=0.0033s=3.3ms$$

因此，在这个示例中，程序修改完毕后，一个帧需要增加 3.3ms 的额外绘制时间。

但是，假设这个 WebGL 应用程序从一开始的帧频就为 30 FPS，修改这个应用程序后，它的帧频降为 20 FPS。在这种情况下，绘制这个场景中添加的图形对象会增加多少时间？

$$\frac{1}{30}-\frac{1}{20}\approx 0.05-0.033=0.017s=17ms$$

在上述第二个示例中，修改应用程序后会增加 17ms 的绘制时间。从前面这两个示例可以看出，帧频都下降 10 FPS，但是从 60 FPS 下降到 50 FPS 只会对帧时间产生微小的影响，在这种情况下，只会增加 3.3ms 的额外绘制时间。另一方面，如果帧频从 30 FPS 下降为 20 FPS，则绘制时间要增加 17ms。

如果我们在开始时有一个较高的帧频，则帧时间的微小变化会引起 FPS 值的很大变化(见图 6-1)。图 6-1 反映了 FPS 值随帧绘制时间的变化。

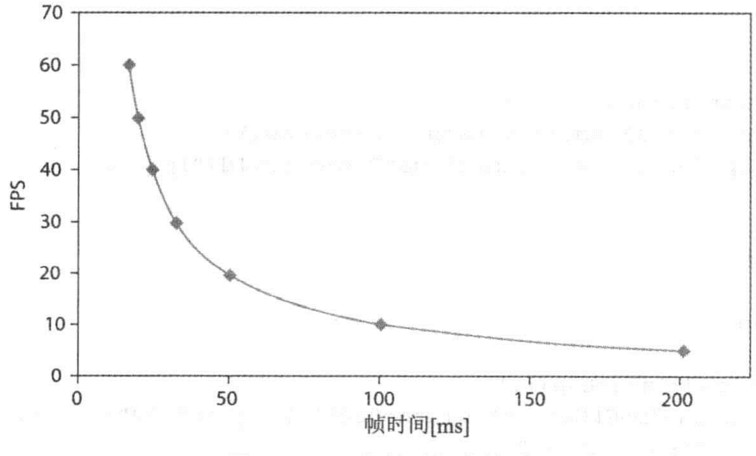


图 6-1 FPS 与帧时间的关系

表 6-1 列出几组 FPS 与帧时间的关系示例

表 6-1 FPS 与帧时间的几组实际值

帧时间	FPS=1/(帧时间)
17ms	60
20ms	50
25ms	40
33ms	30
50ms	20
100ms	10
200ms	5

假设有一个很大的 FPS 值，这意味着它位于图 6-1 中图形的左侧，从图中可以看出帧时间的微小变化就会引起 FPS 的很大变化。

相反，假设有一个很小的 FPS 值，这意味着它位于此图形的右侧，因此同样大小的帧时间变化量会产生较小的 FPS 值变化，因为右侧的图形较平坦。

6.2 用户交互事件的处理

为了在 WebGL 应用程序中处理用户的输入，需要掌握 JavaScript 的事件处理机制。这类事件处理并不是专门用于 WebGL 的，因此如果你有 Web 开发背景，则可能已经熟悉这方面的内容。

但是，由于事件处理非常重要，它能够使用户的 WebGL 应用程序具有交互功能——例如，应用程序可以响应键盘操作或鼠标单击——因此本节详细介绍事件处理的基本知识，确保你掌握所需要的知识。

事件处理的基本思想是当浏览器中发生了某个重要的事情时，浏览器就创建并发送一个事件。用户的应用程序可以用事件处理程序侦听这些事件。

浏览器可以创建很多不同类型的事件。实际上，你在前面的几章中已经看到几个事件的示例，尽管其中没有对事件处理做详细的讨论或介绍。例如，在第 5 章中，我们使用了以下事件，尽管它们与用户的输入没有关系：

- load——资源载入结束。利用这个事件，我们就能够知道什么时候纹理的图像资源载入结束。
- webglcontextlost——WebGL 上下文已丢失。
- webglcontextrestored——WebGL 上下文已恢复。

与用户输入有关的事件有许多。比较有用的事件示例有：

- keydown

- keypress
- keyup
- mousedown
- mouseup
- mousemove

处理 Web 浏览器的事件有两种主要方法，它们是：

- DOM Level 0 基本事件处理
- DOM Level 2 高级事件处理

此外，IE 浏览器的旧版本使用它们特有的方法，即混合使用这两种方法。这种方法不在本书的讨论范围内。其他两种方法将在以下几节中讨论。

6.2.1 DOM Level 0 基本事件处理

DOM Level 0 事件处理是指旧式的事件处理方法，凡是支持 JavaScript 的浏览器都支持这种事件处理方法。根据这种事件处理模型，通过 HTML 元素的属性来定义事件处理程序，或者通过把 JavaScript 的一个值赋给相应的属性来定义事件处理程序。DOM Level 0 使用的事件处理名称都是以 on 开头的，如 onload、onmousedown、onmouseup 等。

1. 事件处理程序作为 HTML 元素的一个属性

前面已提到，在 HTML 元素的属性中可以定义事件处理程序。为了让你对此有一个了解，以下是一个示例。你可能熟悉这段代码，它来自本书前面的几个示例：

```
<!DOCTYPE HTML>
  <html lang="en">
  <head>

  ...

  </head>

  <body onload="startup();">
    <canvas id="myGLCanvas" width="500" height="500"></canvas>
  </body>
</html>
```

在<body>元素的 onload 属性中设置一个事件处理程序，此事件处理程序在文档载入后执行。注意，在本例中，事件处理程序可以是 JavaScript 代码的任意字符串。实际上，只要用引号分隔语句，就可以把多个 JavaScript 语句直接赋给一个事件处理程序。但是如果事件处理程序由几条语句组成，则从结构角度来看，最好是调用一个能够将这些语句集合在一起的函数。

2. 事件处理程序作为 JavaScript 对象的属性

在 Web 文档中，每个 HTML 元素都对应于一个 DOM 元素。DOM 元素在 JavaScript 程序中被当作一个 JavaScript 对象进行访问。这个 JavaScript 对象的属性与 HTML 元素的属性相对应。

这意味着，把事件处理程序赋给一个 HTML 对象的另一个方法是把一个代表事件处理程序的函数直接赋给 JavaScript 对象的一个属性。有一点需要指出，虽然 HTML 事件处理程序属性的值必须是一个字符串，但是赋给 JavaScript 对象属性的值必须是一个函数。

在第 5 章中，我们已用过这种技术，即把一个函数赋给 JavaScript 对象的一个属性。在其中，我们把一个匿名函数赋给图像的 onload 事件处理程序，这样就可以知道何时图像数据载入结束。下面这段代码就说明了这个方法：

```
var image = new Image();
image.onload = function() { // Assign event handler as anonymous function
   // handle the loaded image
   ...
}

image.src = url;
```

用普通的函数而非匿名函数也可实现同样的功能。在本例中，我们把一个函数赋给一个事件处理程序，如下所示：

```
function imageLoadHandler() {
   // handle the loaded image
   ...
}
var image = new Image();
image.onload = imageLoadHandler; // Assign the event handler
image.src = url;
```

注意，在这种情况下，作为事件处理程序的函数的名称后面必须有括号。

实际上在某些情形下，像本节这样，利用 JavaScript 属性把一个值赋给事件处理程序而不是给 HTML 属性定义一个包含 JavaScript 代码的字符串反而有好处。喜欢使用 JavaScript 属性的 Web 开发人员经常提到两个优点，它们是：

- 把 HTML 代码与 JavaScript 代码相分离时，程序更清晰。这样，程序更容易模块化，且更容易维护。
- 如果有必要，我们把可以把事件处理函数设置为动态形式，需要时把它们赋给 JavaScript 对象的属性，不需要时就删除事件处理程序。当用户需要一个复杂的应用程序时，这个方法就很有用。

6.2.2　DOM Level 2——高级事件处理方法

DOM Level 2 事件处理模型不同于前一节介绍的 DOM Level 0 事件处理模型。从表面上

看，两个模型只是在注册事件处理程序的方式上有不同。对于 DOM Level 0 模型，可以把一个 JavaScript 代码字符串赋给 HTML 元素的属性，也可以把一个函数赋给 JavaScript 属性。

对于 DOM Level 2 模型，我们可以用 addEventListener()方法注册一个元素的事件侦听程序。在本书第 5 章中，你已经看到如何用这个方法注册 webglcontextlost 和 webglcontextrestored 事件的处理程序。

```
canvas.addEventListener('webglcontextlost', handleContextLost, false);
canvas.addEventListener('webglcontextrestored', handleContextRestored,
                         false);
```

此外，DOM Level 2 模型也不同于 DOM Level 0，其事件处理程序的名称没有前缀 on。例如，DOM Level 0 中的事件处理程序 onload 在 DOM Level 2 中只对应 load 事件处理程序。

然而，DOM Level 0 模型与 DOM Level 2 模型的差别不仅仅局限于事件处理程序的注册方式和命名上。更大的差别在于事件传播机制不同。

事件传播

对于 DOM Level 0 模型，事件发送给在其上发生此事件的文档元素。如果这个元素已注册了一个事件处理程序，就执行这个事件处理程序。这个简单的方法在很多情况下都十分有效。

但是，DOM Level 2 模型要复杂得多。它有一个事件传播过程，此过程有 3 个阶段：

- 事件捕获阶段
- 执行位于目标结点上的处理程序
- 事件冒泡阶段

这 3 个阶段可用图 6-2 来说明。

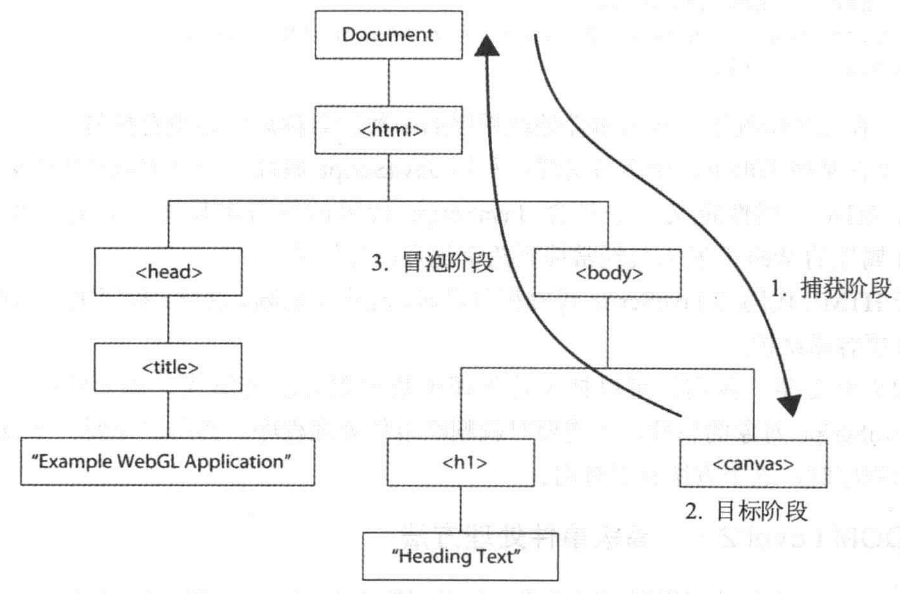


图 6-2　DOM Level 2 模型的事件传播过程

在第一个阶段，即事件捕获阶段，事件从 DOM 树的顶端结点开始，即从文档对象开始，然后往下向目标结点传播。如果目标结点的任何祖先结点已经注册一个事件处理程序且激活了它的捕获功能，就会在事件的捕获阶段执行这个处理程序。调用 addEventListerner()函数并把它的最后一个参数设置 true，就可以注册一个事件处理程序并激活它的捕获功能。

第二个阶段执行位于目标结点上的事件处理程序，这与 DOM Level 0 事件模型的行为相似。

第三个阶段(最后一个阶段)是事件冒泡阶段。从某种意义上讲，它与捕获阶段相反。在事件冒泡阶段，事件向上传播直到 DOM 树的顶部，即到文档对象为止。需要指出的是，并非所有事件都向上冒泡。然而在一般情况下，低层事件，如 mousedown、mouseup 和 mousemove，都要向上冒泡。在冒泡阶段，任何注册在目标结点的祖先结点上的事件处理程序都会被触发。

现在你已经知道了 Web 应用程序的事件处理过程。此外，你完全掌握了 addEventListener()方法的第三个参数的用法。在前面的示例中，这个参数设置为 false，这是因为我们对捕获阶段不感兴趣。

```
canvas.addEventListener('webglcontextlost', handleContextLost, false);
```

6.2.3 键盘输入

键盘输入的事件处理程序因浏览器供应商和操作系统不同而有所区别。但是有些基本的内容对于绝大多数浏览器具有共性。一般而言，当按下一个字母数字键时，会引发 3 个键盘事件：

- keydown
- keypress
- keyup

当按下一个字母数字键时，首先引发一个 keydown 事件，紧随其后是 keypress 事件。当释放这个键时，引发一个 keyup 事件。

实际上，keydown 和 keyup 事件不同于 keypress 事件。如果我们能够分清键和字符的区别，则比较容易理解它们之间的差别。我们可以把一个键看成设备上的物理键，而字符是我们按下一个物理键后显示的符号(如 A、a、b、@、$、%等)。

一般而言，keydown 和 keyup 事件表示物理键被按下或释放。而另一方面，keypress 事件代表输入的字符。当用户想知道按下哪个键或按下的键对应哪个字符时，则键盘事件有两个属性会令用户感兴趣，它们是：

- keyCode——它表示一个虚拟键码(简称虚拟码)，给我们提供了用户按下哪个键的信息。虚拟键码对应键的大写字母版本的 ASCII 码。例如，如果按下标有 A 的键，则生成 65 的虚拟键码，而不管是否按下大写字母锁定键。
- charCode——代表最后得到的字符的 ASCII 码。

通过一个示例可以帮助你更好地理解这些内容。假设我们已经为 keydown、keyup 和

keypress 事件添加相应的事件处理程序，如下所示：

```
document.addEventListener('keydown', handleKeyDown, false);
document.addEventListener('keyup', handleKeyUp, false);
document.addEventListener('keypress', handleKeyPress, false);
```

再进一步假设这 3 个事件处理程序用 console.log()方法向控制台输出一个信息，此信息包含了 charCode 和 keyCode，如下所示：

```
function handleKeyDown(event) {
  console.log("keydown - keyCode=%d, charCode=%d",
              event.keyCode, event.charCode);
}

function handleKeyUp(event) {
  console.log("keyup - keyCode=%d, charCode=%d",
              event.keyCode, event.charCode);
}

function handleKeyPress(event) {
  console.log("keypress - keyCode=%d, charCode=%d",
              event.keyCode, event.charCode);
}
```

按下标有 A 的键时(没有按下大写字母锁定键)，则在 Google Chrome 终端上看到以下结果：

```
keydown - keyCode=65, charCode=0
keypress - keyCode=97, charCode=97
keyup - keyCode=65, charCode=0
```

在 Firefox 浏览器中按下标有 A 的键时(没有按下大写字锁定键)会生成以下的结果：

```
keydown - keyCode=65, charCode=0
keypress - keyCode=0, charCode=97
keyup - keyCode=65, charCode=0
```

可以看出，针对这个数字字符键，这两个浏览器都会引发 3 个事件——keydown、keypress 和 keyup。在这两个浏览器中，keydown 事件和 keyup 事件生成的虚拟键码都一样，都是 65。这正是字符'A'的 ASCII 码。

但是，对于 keypress 事件，Google Chrome 浏览器得到的虚拟键码是 97，这对应于字符'a'的 ASCII。在 Firefox 浏览器中，得到的 keyCode 是 0，进一步的信息需要读取 keypress 事件返回的 charCode 码才能确定。

对不可显示的字符——如 Esc、箭头键或 F1~F12 键——不管是 Chrome 还是 Firefox 浏览器，都会引发一个 keydown 事件和一个 keyup 事件。然而，Firefox 浏览器还引发一个 keypress 事件，而 Chrome 则没有。作为一个示例，当我们在 Chrome 浏览器中按下 Esc 键(同样使用输出 keyCode 和 charCode 信息的事件处理程序)时，则得到以下结果：

```
keydown - keyCode=27, charCode=0
keyup - keyCode=27, charCode=0
```

如果在 Firefox 中按下 Esc 键，则得到以下结果：

```
keydown - keyCode=27, charCode=0
keypress - keyCode=27, charCode=0
keyup - keyCode=27, charCode=0
```

键盘事件处理因浏览器的不同而表现出很大的差别。这意味着，如果用户在自己的 Web 应用程序中使用键盘事件，则必须在想要支持的各种浏览器上测试自己的应用程序。

从这些简单的示例可以看出，处理键盘输入事件时，就会遇到不同浏览器具有不同行为模式这个问题。然而，当利用键盘输入控制 WebGL 场景中的对象时，更重要的是要知道按下的是哪个键，而不是这个键对应的字符。例如，在一款赛车游戏中，通常我们需要知道用户按的是油门还是刹车。在飞行模拟游戏中，我们需要知道用户是否按发射火箭的键。

如果你仔细观察本节的示例，则可能已经发现，在所有示例中 keydown 和 keyup 事件的 keyCode 属性都具有相同的行为。最终人们发现，在 WebGL 应用程序中，对于 keydown 和 keyup 事件，使用它们的 keyCode 属性是一种优秀策略。表 6-2 列出一些有用键的 keyCode 值。

表 6-2　JavaScript 中一些有用的 keyCode

键	keyCode	键	keyCode
Escape	27	G	71
向左箭头	37	H	72
向上箭头	38	I	73
向右箭头	39	J	74
向下箭头	40	K	75
0	48	L	76
1	49	M	77
2	50	N	78
3	51	O	79
4	52	P	80
5	53	Q	81
6	54	R	82
7	55	S	83
8	56	T	84
9	57	U	85
A	65	V	86

(续表)

键	keyCode	键	keyCode
B	66	W	87
C	67	X	88
D	68	Y	89
E	69	Z	90
F	70		

如果你在 Web 上输入“JavaScript keyCode”或者“JavaScript keyCode test”关键字，就会找到几个网页，它们有一个文本输入字段和一段简短的 JavaScript 代码。如果用户按下键盘上的某个键，这个 JavaScript 程序就会输出相应的 keyCode。

处理单键按下和多键同时按下事件

至此你已经学习了有关键盘事件的处理方法，并且知道如何基于单个 keydown、keyup 或 keypress 事件进行处理。在许多情形下这已经是很不错的处理方式。但是，有时我们希望能够在自己的应用程序中同时使用两个键或多个键组合实现某个功能。这在游戏中很常见。

假设我们正在用 WebGL 设计一款赛车游戏，我们希望用一个键表示油门，用其他两个键控制方向盘。更进一步假设这款赛车游戏很不一般，它允许游戏玩家向对手发射导弹，因此还需要用另一个键发射导弹。此外，游戏的开发人员可能还希望游戏玩家在操控方向盘的同时能够按下油门，或者能够发射导弹。

在下面这段代码中，事件处理程序 handleKeyDown()是为 keydown 事件注册的，handleKeyUp()是为 keyup 事件注册的。

按下键盘上的 M 键就立即引发一个操作，即发射一个导弹。在 handleKeyDown()函数中，调用一个静态方法 String.fromCharCode()，把 keyCode 的 ASCII 码转换为一个字符。直接把 keyCode 值与 M 的 ASCII 码(它为 77)相比较也是完全可以的，但是利用 String.from Char- Code()，我们在读写这段代码时就不需要知道 M 的 ASCII 值。

为了能够处理游戏玩家同时按下油门和转弯操作，开发人员需要记住哪些键会被同时按下。这只需要跟踪变量 pwgl.listOfPressedKeys 的值。然后把这个列表(即 pwgl.listOfPressed-Keys)用在 handlePressDownKeys()函数中。一般而言，这个函数需要每帧调用一次，通常是在场景绘制之前调用。在这个函数中，要检查当前哪几个键被同时按下，根据检查结果采取不同的操作。注意在这个简单的示例中，在 handlePressedDownKeys()函数中，处理油门按键事件实际上会得到一个加速度，加速度实际上依赖于帧频。为了避免这种情况，可以应用 6.1.3 节学到的技术：

```
function handleKeyDown(event) {
  // When you get a keydown you first handle any immediate actions
  // that are relevant.
  if (String.fromCharCode(event.keyCode) == "M") {

    // Fire missile since M key was pressed down
    fireMissile();
  }

  // Store information about which key has been pressed
  // so we can check this in the function handlePressedDownKeys().
  // This strategy let you have several keys pressed down simultaneously
  pwgl.listOfPressedKeys[event.keyCode] = true;
}

function handleKeyUp(event) {
  // Update list of keys that are pressed down,
  // by setting the released key to false;
  pwgl.listOfPressedKeys[event.keyCode] = false;
}

...

function handlePressedDownKeys() {
  if (pwgl.listOfPressedKeys[38]) {
    // Arrow up, the user pressed the gas pedal.
    speed += 0.5;
  }
  if (pwgl.listOfPressedKeys[40]) {
    // Arrow down, the user pressed the brake.
    speed -= 0.5;
  }
  if (pwgl.listOfPressedKeys[37]) {
    // Arrow left, the user wants to turn left.
    turnLeft();
  }
  if (pwgl.listOfPressedKeys[39]) {
    // Arrow right, the user wants to turn right.
    turnRight();
  }

}
```

6.2.4 鼠标输入

既然你已经知道了如何在 WebGL 应用程序中处理键盘输入，现在简单介绍鼠标输入处理，它是 WebGL 应用程序获取用户输入的另一个重要方法。虽然存在几个不同的鼠标事件，但是有 3 个事件在 WebGL 应用程序中特别有用：

- mousemove

- mousedown
- mouseup

当把鼠标移动到某个对象之上时就会引发 mousemove 事件，在某个元素之上按下鼠标按键时就会引发 mousedown 事件，在某个对象之上释放鼠标按键时就会引发 mouseup 事件。这 3 个事件都包含几个非常有用的属性。这 3 个事件都有 clientX 和 clientY 属性，这两个属性包含了鼠标指针相对于浏览器视口的左上角的位置信息。

mousedown 和 mouseup 也包含一个 button 属性，它决定鼠标上的哪个按键被按下或释放。如果 button 值为 0，则表示此事件与鼠标的左按键有关；如果 button 值为 1，则此事件与鼠标的中间按键有关；如果 button 值为 2，则事件与鼠标的右按键有关。

我们可以为这些事件注册一个事件处理程序，如下面几行代码所示：

```
document.addEventListener('mousemove', handleMouseMove, false);
document.addEventListener('mousedown', handleMouseDown, false);
document.addEventListener('mouseup', handleMouseUp, false);
```

下面这段代码包含 3 个函数，它们是这 3 个事件的处理程序。在本例中，事件处理函数只执行一个事情，即用 console.log()方法把有关信息输出到浏览器窗口上。这些信息包含了生成的事件、clientX 属性值和 clientY 属性值。对于 mousedown 和 mouseup 事件，也把 button 属性的值输出到浏览器窗口上。

```
function handleMouseMove(event) {
  console.log("mousemove - clientX=%d, clientY=%d",
              event.clientX, event.clientY);
  }

function handleMouseDown(event) {
  console.log("mousedown - clientX=%d, clientY=%d, button=%d",
              event.clientX, event.clientY, event.button);
}

function handleMouseUp(event) {
  console.log("mouseup - clientX=%d, clientY=%d, button=%d",
              event.clientX, event.clientY, event.button);
}
```

设置了这 3 个事件处理程序后，如果我们把鼠标指针从左上角向下移动到坐标 x=5,y=5 的窗口位置，则在浏览器的控制台中输出以下消息：

```
mousemove - clientX=0, clientY=0
mousemove - clientX=1, clientY=0
mousemove - clientX=2, clientY=0
mousemove - clientX=2, clientY=1
mousemove - clientX=2, clientY=2
mousemove - clientX=2, clientY=3
mousemove - clientX=3, clientY=3
mousemove - clientX=3, clientY=4
```

```
mousemove - clientX=4, clientY=4
mousemove - clientX=5, clientY=5
```

同样道理，如果我们在窗口坐标 x=5 和 y=5 的位置按下并释放鼠标的左按键，则会看到如下的输出消息：

```
mousedown - clientX=5, clientY=5, button=0
mouseup - clientX=5, clientY=5, button=0
```

如果在同一个位置按下并释放鼠标的右按键，则会看到如下的消息：

```
mousedown - clientX=5, clientY=5, button=2
mouseup - clientX=5, clientY=5, button=2
```

6.3　综合应用新知识

现在把本章学到的知识应用到一个较大的示例中，这个示例也是针对第 5 章中的同一个场景的。但是在这里，我们想让这个场景动起来，即让桌面上原先静止的盒子在桌面上面沿着一个圆的轨迹运动。此外，用向上和向下箭头键改变盒子沿着其运动的圆的半径。图 6-3 显示这个场景的效果。

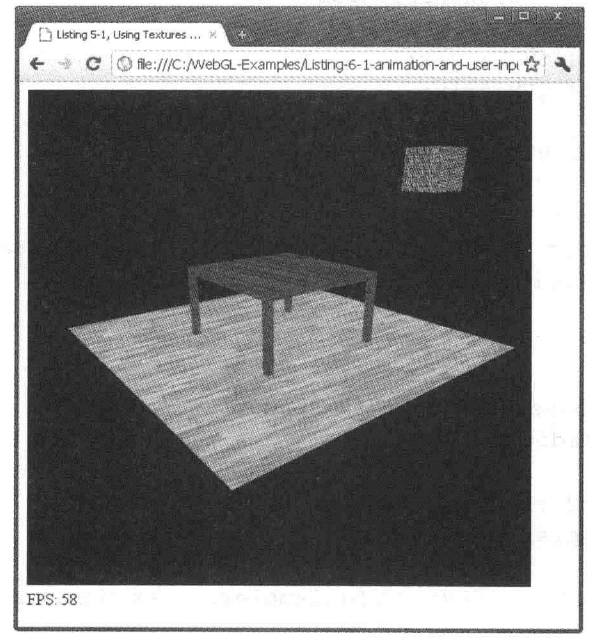


图 6-3　由程序清单 6-1 生成的动画场景

程序清单 6-1 是与这个场景对应的源代码。无须过多的解释，你应该能够看懂绝大部分内容，因为这个示例主要以本书到目前为止介绍的内容为基础。

程序清单 6-1 的完整源代码可以从 wrox.com 网站上下载，但是由于你已经开始对执行以下操作的代码非常熟悉：创建 WebGL 上下文、载入着色器、编译着色器、链接着色器、

建立顶点缓冲等，因此这里并没有列出这个示例的全部代码。你在前面的几章中已经好几次看到过这些代码。下面的内容只提供程序清单 6-1 中的部分代码段，并且只对你比较陌生的内容进行说明。

首先还是需要分析 HTML 文档的开头部分。在几行初始 HTML 代码之后，这段代码包含一个顶点着色器和一个片段着色器，现在它们对你而言应该非常熟悉了。

可从 Wrox.com 下载源代码

程序清单 6-1　一个运动的 WebGL 场景

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Listing 6-1, Animated WebGL Scene</title>
<script src="webgl-debug.js"></script>
<script type="text/javascript" src="glMatrix.js"></script>
<script src="webgl-utils.js"></script>

<meta charset="utf-8">

<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec2 aTextureCoordinates;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;

  varying vec2 vTextureCoordinates;

  void main() {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vTextureCoordinates = aTextureCoordinates;
  }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;

  varying vec2 vTextureCoordinates;
  uniform sampler2D uSampler;
  void main() {
    gl_FragColor = texture2D(uSampler, vTextureCoordinates);
  }
</script>
```

下一步分析位于 HTML 文件末尾的代码。在 HTML 文件的末尾有<body>标记，后者包含<canvas>标记和<div>标记，<div>标记用于 FPS 计数器。此外，<body>标记中还有 startup()函数，当 onload 事件触发时调用此函数。

startup()函数首先用 document.getElementById()获得一个画布的引用，然后调用 webgl-

debug.js 库中的 WebGLDebugUtils.makeLostContextSimulatingContext()方法，以原画布为基础创建一个包装画布，这样就可以模拟 webglcontextlost 事件和 webglcontextrestored 事件。此外，还需要指出的是，有 3 行代码模拟丢失上下文，在这个函数中，这 3 行内容已被注释掉。为了能够用鼠标释放模拟丢失上下文，需要取消这几行的注释。给这个画布添加事件侦听程序，然后调用 createGLContext()函数创建 WebGLRenderingContext 对象。

接着调用 init()函数，这个函数设置和初始化用户应用程序第一次启动时以及处理上下文丢失和上下文恢复事件所需要的变量。这样，既可以在应用程序启动时从 startup()函数中调用 init()函数，也可以在恢复丢失的上下文时从 handleContextRestored()函数中调用 init()函数。最后，当一切准备就绪后，startup()函数调用 draw()函数开始绘制场景。

```
function startup() {
  canvas = document.getElementById("myGLCanvas");
  canvas = WebGLDebugUtils.makeLostContextSimulatingCanvas(canvas);

  canvas.addEventListener('webglcontextlost', handleContextLost, false);
  canvas.addEventListener('webglcontextrestored',
                            handleContextRestored, false);
  document.addEventListener('keydown', handleKeyDown, false);
  document.addEventListener('keyup', handleKeyUp, false);
  document.addEventListener('keypress', handleKeyPress, false);
  document.addEventListener('mousemove', handleMouseMove, false);
  document.addEventListener('mouseup', handleMouseUp, false);

  gl = createGLContext(canvas);
  init();

  pwgl.fpsCounter = document.getElementById("fps");

  // Uncomment the three lines of code below to be able to test lost context
  // window.addEventListener('mousedown', function() {
  //    canvas.loseContext();
  // });

  // Draw the complete scene
  draw();
}
</script>

</head>

<body onload="startup();">
  <canvas id="myGLCanvas" width="500" height="500"></canvas>
  <div id="fps-counter">
    FPS: <span id="fps">--</span>
  </div>
</body>

</html>
```

现在讨论 init()函数的内容，目的是让你对这个函数的作用有一个大致的了解。可以看出，这个函数确保着色器、顶点缓冲和纹理等都已设置完毕。此外，这个函数还包含其他一些初始化代码：

```
function init() {
  // Initialization that is performed during first startup and when the
  // event webglcontextrestored is received is included in this function.
  setupShaders();
  setupBuffers();
  setupTextures();
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.enable(gl.DEPTH_TEST);

   // Initialize some varibles for the moving box
  pwgl.x = 0.0;
  pwgl.y = 2.7;
  pwgl.z = 0.0;
  pwgl.circleRadius = 4.0;
  pwgl.angle = 0;
  // Initialize some variables related to the animation
  pwgl.animationStartTime = undefined;
  pwgl.nbrOfFramesForFPS = 0;
  pwgl.previousFrameTimeStamp = Date.now();
}
```

这就是 init()函数的完整内容。现在分析上下文丢失和上下文恢复事件的处理程序。先介绍 handleContextLost()函数，它调用 event.preventDefault()方法阻止丢失上下文时的默认行为。其理由是，根据这种默认的行为，丢失的上下文绝对不会得到恢复，而这不是此时所希望发生的事情。此外，这个函数会取消动画，并循环访问所有正在进行的图像加载操作，把它们的 onload 处理程序设置为 undefined，从而保证忽略这些事件。

```
function handleContextLost(event) {
   event.preventDefault();
   cancelRequestAnimFrame(pwgl.requestId);

   // Ignore all ongoing image loads by removing
   // their onload handler
   for (var i = 0; i < pwgl.ongoingImageLoads.length; i++) {
     pwgl.ongoingImageLoads[i].onload = undefined;
   }
   pwgl.ongoingImageLoads = [];
}
```

当恢复上下文时，调用 handleContextRestored()函数。在这个函数中，调用 init()函数，再次初始化应用程序的状态，然后启动动画：

```
function handleContextRestored(event) {
   init();
```

```
    pwgl.requestId = requestAnimFrame(draw,canvas);
  }
```

接着介绍键盘事件处理程序。handleKeyDown()和 handleKeyUp()函数用来更新一个列表，此列表保存了当前按下的键。在每个帧中用 handlePressedDownKeys()函数读取这个列表一次。如果用户按下向上箭头键，场景中的盒子沿着其运动的圆的半径增加 0.1；如果用户按下向下方向键，这个圆的半径减少 0.1。函数 handleKeyPress()只包含注释掉的代码，这些代码把结果写入 console.log()中。

```
function handleKeyDown(event) {
  pwgl.listOfPressedKeys[event.keyCode] = true;

  // If you want to have a log for keydown you can
  // uncomment the two lines below.
  // console.log("keydown - keyCode=%d, charCode=%d",
  //             event.keyCode, event.charCode);
}

function handleKeyUp(event) {
  pwgl.listOfPressedKeys[event.keyCode] = false;

  // If you want to have a log for keyup you can
  // uncomment the two lines below.
  // console.log("keyup - keyCode=%d, charCode=%d",
  //             event.keyCode, event.charCode);
}

function handleKeyPress(event) {
  // If you want to have a log for keypress you can
  // uncomment the two lines below.
  // console.log("keypress - keyCode=%d, charCode=%d",
  //             event.keyCode, event.charCode);
}

function handlePressedDownKeys() {
  if (pwgl.listOfPressedKeys[38]) {
    // Arrow up, increase radius of circle
    pwgl.circleRadius += 0.1;
  }
  if (pwgl.listOfPressedKeys[40]) {
    // Arrow down, decrease radius of circle
    pwgl.circleRadius -= 0.1;
    if (pwgl.circleRadius < 0) {
      pwgl.circleRadius = 0;
    }
  }
}
```

最后分析 draw()函数，这个函数负责整个场景的绘制任务。draw()函数先调用 requestAnimFrame()函数，请求动画的下一次绘图调用。然后它调用 handlePressedDownKeys()函数，该函数正是这里要介绍的内容。当键盘按键事件处理完毕后，draw()函数设置变换并在其他辅助函数的帮助下绘制场景。

```
function draw(currentTime) {
  pwgl.requestId = requestAnimFrame(draw);
  if (currentTime === undefined) {
    currentTime = Date.now();
  }

  handlePressedDownKeys();

  // Update FPS if a second or more has passed since last FPS update
  if(currentTime - pwgl.previousFrameTimeStamp >= 1000) {
    pwgl.fpsCounter.innerHTML = pwgl.nbrOfFramesForFPS;
    pwgl.nbrOfFramesForFPS = 0;
    pwgl.previousFrameTimeStamp = currentTime;
  }

  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
  mat4.perspective(60, gl.viewportWidth / gl.viewportHeight,
                   1, 100.0, pwgl.projectionMatrix);
  mat4.identity(pwgl.modelViewMatrix);
  mat4.lookAt([8, 5, 10],[0, 0, 0], [0, 1,0], pwgl.modelViewMatrix);

  uploadModelViewMatrixToShader();
  uploadProjectionMatrixToShader();
  gl.uniform1i(pwgl.uniformSamplerLoc, 0);

  drawFloor();

  // Draw table
  pushModelViewMatrix();
  mat4.translate(pwgl.modelViewMatrix, [0.0, 1.1, 0.0], pwgl.modelViewMatrix);
  uploadModelViewMatrixToShader();
  drawTable();
  popModelViewMatrix();

  // Calculate the position for the box that is initially
  // on top of the table but will then be moved during animation
  pushModelViewMatrix();
  if (currentTime === undefined) {
    currentTime = Date.now();
  }
  if (pwgl.animationStartTime === undefined) {
    pwgl.animationStartTime = currentTime;
  }
```

```
        // Update the position of the box
        if (pwgl.y < 5) {
          // First move the box vertically from its original position on top of
          // the table (where y = 2.7) to 5 units above the floor (y = 5).
          // Let this movement take 3 seconds
          pwgl.y = 2.7 + (currentTime - pwgl.animationStartTime)/3000 * (5.0-2.7);
        }
        else {
          // Then move the box in a circle where one revolution takes 2 seconds
          pwgl.angle = (currentTime - pwgl.animationStartTime)/
                       2000*2*Math.PI % (2*Math.PI);

          pwgl.x = Math.cos(pwgl.angle) * pwgl.circleRadius;
          pwgl.z = Math.sin(pwgl.angle) * pwgl.circleRadius;
        }

        mat4.translate(pwgl.modelViewMatrix,
         [pwgl.x, pwgl.y, pwgl.z], pwgl.modelViewMatrix);
        mat4.scale(pwgl.modelViewMatrix, [0.5, 0.5, 0.5], pwgl.modelViewMatrix);
        uploadModelViewMatrixToShader();
        drawCube(pwgl.boxTexture);
        popModelViewMatrix();

        // Update number of drawn frames to be able to count fps
        pwgl.nbrOfFramesForFPS++;
      }
```

6.4 小结

本章介绍了如何让自己的 WebGL 场景动起来，从而得到运动的感觉。我们先分析了传统的动画技术，即用 setInterval()或 setTimeout()函数构建绘制循环；此外，还介绍了使用新方法 requestAnimationFrame()的优点。你学习了如何测量动画的 FPS 值，如何将结果显示在屏幕上。此外你还知道了用非线性 FPS 衡量 WebGL 应用程序性能的结果。

你在本章中学习了事件处理方法。具体来说，就是如何处理用户输入。你现在还知道了 DOM Level 0 事件处理模型与 DOM Level 2 事件处理模型的差别。最后，你还看到键盘输入处理是一个比较复杂的事件处理，因为需要考虑不同浏览器的不同行为。

在第 7 章中，你将要学习另一个令人兴奋的主题，即如何在 WebGL 应用程序中模拟光照。当我们想要创建比较具有真实感的 WebGL 场景，光照就是一个重要的组成部分。

第 7 章

光　照

本章主要内容：

- 局部光照模型与全局光照模型之间的区别
- 如何把 Phong 反射模型用于光照计算，以及如何用 OpenGL ES 着色语言(Shading Language)实现这个模型
- 环境反射、漫反射和镜面反射之间的区别
- 光照模型和插值技术之间的区别
- 平面着色、Gouraud 着色和 Phong 着色之间的区别
- 如何用 OpenGL ES 着色语言实现平行光、点光源和聚光灯
- 如何模拟光照衰减效果

创建具有真实感的 3D 场景时，光照是一个重要的因素。在真实世界中，灯光大大地影响了我们对周围环境的观看。例如，只要对比在晴空万里时看到大海的效果与在阴云密布下看到的效果，就会明白光照的作用。本章将学习如何在 WebGL 中处理光照。

7.1　光源

上学时，在物理课本中学过光是电磁辐射。人眼能够觉察到的光的波长处于 400nm~700nm。波长在 400nm 附近的光人眼感觉为紫色，波长在 700nm 附近的光在人眼感觉为红色。

在 3D 图形中，通常需要确定不同的光源(如太阳或白炽灯)如何影响场景中不同的表面。准确确定光源的效果非常困难，因为光与不同对象和材质之间的作用非常复杂。

即使只有一个光源，它也可以向不同的方向发射光线。光沿着直线传播，直到被一个对象反射或吸收为止。光线经过对象多次反射，反射的光照亮其他对象。

光也会发生折射现象。例如，当光照射到水面时，或者当光照射到一个具有散射作用

的对象时，它会继续沿着许多不同的方向照射。为了在 3D 图形中模拟真实世界的灯光，显然需要一个比真实世界中光源简单许多的模型。

7.2 局部光照模型的工作原理

在 3D 图形中模拟灯光时，可以使用以下两种不同类型的光照模型。

- 全局光照模型
- 局部光照模型

图 7-1 说明了全局光照模型与局部光照模型之间的区别。

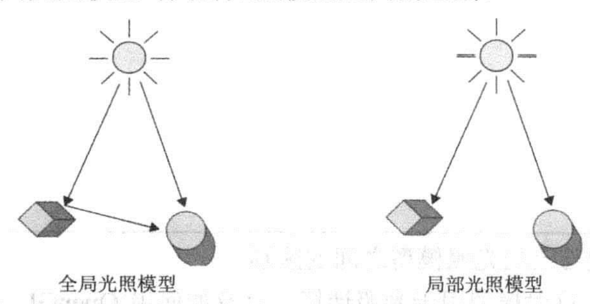


图 7-1 全局光照模型与局部光照模型之间的区别

全局光照模型会利用没有被光源直接照射到的对象的信息。例如，该模型考虑了来自对象反射的光源的光，以及反射光又照射另一对象的光。

全局光照模型的一个例子是光线跟踪(ray tracing)。这一技术试图模拟光的复杂行为。它可以产生极为真实的场景，但是由于它需要大量的计算机资源，因此该方法经常用于在显示场景之前渲染场景，例如用于静止图像或影片。全局光照模型的另一个例子是热辐射法，但是即使是热辐射法，也是实时 3D 图形中不常使用的算法。

局部光照模型只考虑直接从已选定光源发射的光。其结果是，模型中的对象不会被其他对象反射的光所照射。局部光照模型的另一个重要属性是对象不会阻挡照射到它的光线。这意味着，在局部光照模型中不会自动创建阴影。

由于 WebGL 通常用于实时 3D 图形中，因此在 WebGL 也常使用某些类型的局部光照模型。如果需要在局部光照模型中生成阴影，则可以用“阴影映射”(Shadow mapping)技术创建阴影。在 WebGL 中一个常用的局部光照模型是 Phong 反射模型，稍后介绍。

7.3 Phong 反射模型

Phong 反射模型(Phong reflection model)是以计算机图形学的研究者 Bui Tuong Phong 的名字命名的，他于 1975 年开发并发表了该模型。为了理解这个模型，我们需要知道，真实世界对象的颜色是由离开这个对象表面的光的颜色决定的。例如，一个看起来为红色的对象，主要反射(或发射)的是红色光。

注意，Phong 反射模型不同于 Phong 着色模型，尽管 Phong 着色模型也由同一个人(即 Bui Tuong Phong)开发和发表的。本章后面还要介绍 Phong 着色模型。

在 Phong 反射模型中，一个点(即一个顶点或片段)的最终颜色是由 3 个不同的反射分量组成，即

- 环境光
- 漫射光
- 镜面光

这意味着，一个顶点或一个片段的颜色可以描述为顶点或片段的总反射。

总反射 = 环境反射 + 漫反射 + 镜面反射

为了计算 3 个反射分量中的任何一个，该模型指定了场景中的全部光源都有 3 个分量，即环境光、漫反射光和镜面光。

更进一步，场景中的每个材质也相应地都有环境光反射率、漫反射率和镜面反射率这 3 个属性。材质除了具有这 3 个属性外，还具有第四个属性，即光泽度(shininess)。它可以用来计算镜面反射分量。

前面公式中的“环境反射”、“漫反射”和“镜面反射”是光与相应分量的材质属性相互作用的结果。在后面几节中，还要进一步讨论这种相互作用。

有一点必须搞清楚：环境光是表现出环境反射的光，漫射光是表现出漫反射的光，镜面光是表现出镜面反射的光。这意味着，在讨论 Phong 模型的情景中，人们(包括作者本人在内)总是把“环境光”与“环境反射”、“漫射光”与“漫反射”，“镜面光”与“镜面反射”互换使用。

Phong 反射模型有时也称为 ADS 光照模型，ADS 是 3 个反射分量——即环境光(Ambient)、漫射光(Diffuse)和镜面光(Specular)3 个单词的首字母组成的。

7.3.1 环境反射

环境光是在场景中反射多次以至于无法确定来自某个方向的光。结果是，一个对象的各个侧面都会被环境光照射。假设环境光用 I_a 表示(包含一个 RGB)，材质环境属性为 k_a(它对应于红光、绿光和蓝光分别有相应的材质属性)，则从该材质表面反射的环境反射 I 由以下方程决定。

$$I=k_aI_a$$

因此，你要做的是基于各分量的 k_a 和 I_a 的乘法。把材质属性的红色分量乘以光的红色分量，材质属性的绿色分量乘以光的绿色分量，材质属性的蓝色分量乘以光的蓝色分量。

用 OpenGL ES 着色语言表示，相应地有以下代码。

```
uniform vec3 uAmbientMaterial;
uniform vec3 uAmbientLight;

...

vec3 ambientReflection = uAmbientMaterial * uAmbientLight;
```

从这段着色代码和上述的方程($I=k_aI_a$)可以看出，环境光并没有考虑光的位置和方向，也没有考虑观察方向。

Phong 反射模型是局部光照模型，这意味着它没有直接考虑到反射的光会照射其他对象。环境光补偿了这一点，因为环境光可以看成是已被多次反射的光。

没有环境光，则对象没有被光源直接照射到的表面会全是黑的。例如，处于室内时，通常周围的墙壁、天花板和其他对象会把一些光反射到其他没有被光源直接照射到的表面上。因此在室内场景，如果没有光线直接照射到的对象表面完全是黑的，看上去会不真实。

7.3.2 漫反射

在计算反射光强度时，漫射光考虑到入射光线的方向。垂直入射到表面的光线会反射较多的光，因此比起入射方向与表面的法线有夹角的光线，它的亮度比较大。图 7-2 说明漫反射的一个简单情形。

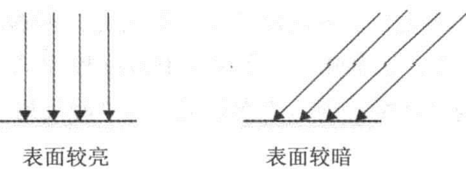


图 7-2 漫反射的例子，左图的光线垂直照射表面，因此表面的亮度要比右图(以一定的入射角照射表面)强

对于漫反射，光线沿各个方向均匀反射。这意味着视线的方向不重要。粗糙表面或暗色材质(如粉笔或泥土)总是表现出漫反射效果。

阅读其他有关 3D 图形文献时，记住，有时称为漫反射为朗伯反射(Lambertian Reflection)。

计算漫反射的方程类似于计算环境反射的方程，但是除了漫反射材质属性 k_d 和漫射光分量 I_d 外，漫反射还要考虑 $\cos\theta$ 因子。正是这个因子考虑到入射光的方向。因此可以用两种不同的方法表示漫反射的公式。第一个方法是：

$$I=k_dI_d\max(\cos\theta,0)$$

角度 θ 定义为表面法线 n 与入射光线方向 l 的最小夹角。公式中的 max()函数指定将

$\cos\theta$ 的负值限制为 0(即把负值设置为 0)。继续往下看，就会明白原因。

图 7-3 所示为漫反射的光照几何模型。从漫反射的方程可以看出，当光线与法线的夹角 θ 为 0° 时，反射光最强，因为 $\cos 0°=1$。当 θ 为 90° 时，则表示光是从侧边发射过来，只接触到表面。由于 $\cos 90° = 0$，因此没有光反射。

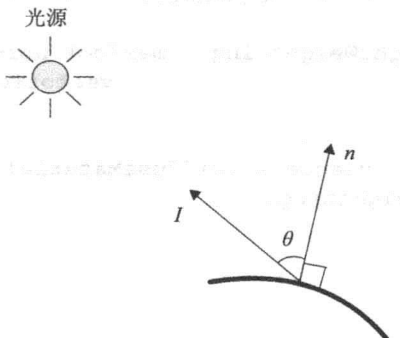


图 7-3　漫反射的光照几何模型

注意，当 $\cos\theta$ 这一项的值为负时，表示角度 θ 大于 90°，这意味着，光是从表面背面发射过来的。这些光线不会照射表面，因此只考虑 $\cos\theta$ 的非负值情形。这正是通过 max() 函数把 $\cos\theta$ 一项的最小值限制为 0 的原因。

第 1 章介绍了 3D 图形中使用的线性代数知识，其中曾提到点积或标量积可以用于 WebGL 的光照计算。现在介绍如何使用。首先，复习两个矢量 $\boldsymbol{u}$ 和 $\boldsymbol{v}$ 的点积的定义。

$$\boldsymbol{u} \cdot \boldsymbol{v} = |\boldsymbol{u}||\boldsymbol{v}|\cos\theta$$

这里的 θ 是 $\boldsymbol{u}$ 和 $\boldsymbol{v}$ 两个矢量之间的最小夹角。如果利用点积的这个定义，并结合漫反射公式，就可以看出，当法线 $\boldsymbol{n}$ 和光线方向 I 都为单位值，漫反射公式可以写成第二个形式：

$$I = k_{\mathrm{d}} I_{\mathrm{d}} \max(\boldsymbol{n} \cdot \boldsymbol{I}, 0)$$

同样道理，这个公式把点积的负值限制为 0。由于在 WebGL 中光照计算通常是由顶点着色器或片段着色器执行的，而且在 OpenGL ES 着色语言中已内置了一个函数 dot()来计算点积，因此漫反射的第二个定义公式往往是 WebGL 应用程序中最方便使用的形式。下面这段着色器源代码用来计算漫反射。

```
uniform vec3 uDiffuseMaterial;
uniform vec3 uDiffuseLight;
uniform vec3 uLightPosition;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;

main() {

   // Get vertex position in eye coordinates
   vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
```

```
    vec3 vertexPositionEye3 = vertexPositionEye4.xyz / vertexPositionEye4.w;

    // Calculate vector (l) to the light source
    vec3 vectorToLightSource = normalize(uLightPosition - vertexPositionEye3);

    // Transform the normal (n) to eye coordinates
    vec3 normalEye = normalize(uNMatrix * aVertexNormal);

    fl oat diffuseLightWeightning = max(dot(normalEye,
                                          vectorToLightSource), 0.0);


    vec3 diffuseRefl ectance = uDiffuseMaterial * uDiffuseLight *
      diffuseLightWeightning;

    ...

}
```

注意，这段代码的开始部分计算计算漫反射所需要的输入值(如在视坐标系中的法线矢量和光线矢量)。以到目前为止所掌握的知识来看，这段代码不难理解，但是目前暂时不详细讨论这段代码，等到本章后面分析 Phong 反射模型的完整着色器代码时再介绍。目前暂且只考虑代码段中高亮显示部分，它们对应于漫反射的计算。

如果希望简化 WebGL 中的光照计算，则只需要用漫反射和环境反射代替完整的 Phong 反射模型，结果仍相当不错。

7.3.3 镜面反射

如果有一个表面光滑的对象——如由抛光的金属制成的对象——则光照射到这个对象时，会在其上产生明亮的斑点或强光。镜面反射就是用来模拟 Phong 反射模型中的这一行为。

你已经看到，环境反射和漫反射都不考虑视线的方向。但是镜面光的反射类似于光线在平面镜上的反射。大部分光按特定的方向反射，因此视线方向非常重要。镜面反射的几何原理如图 7-4 所示。

在图 7-4 中，来自某个方向的光线用矢量 I 表示。注意，该矢量是指向光源的。表面的法线用 n 表示。对于光滑的对象，通常所有光线都沿着 r 方向反射。对于不太光滑的对象，则反射光分散在矢量 r 周围。

矢量 v 指向观察者。当 v 与 r 之间的夹角 θ 为 0 时，绝大部分光都被朝向观察者反射。反射矢量 r 与观察者矢量 v 的夹角 θ 越大，反射到观察者的光就越少。

通常这个量的下降快慢与 $\cos\theta$ 的 α 次方成正比，这里的 α 是材质的光泽度。因此计算镜面反射光 I 的公式可以表示为：

$$I = k_s I_s \max(\cos\theta, 0)^{\alpha}$$

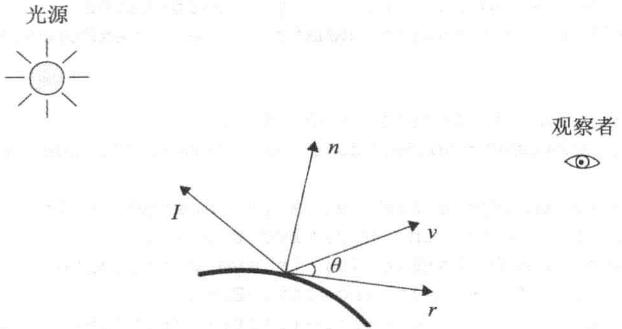


图 7-4　镜面反射的几何原理

这里的 k_s 和 I_s 与环境反射和漫反射计算公式中的因子相似。这意味着镜面材质属性用 k_s 表示，传入光的镜面光分量用 I_s 表示。正如前面曾提到，角度 θ 是指矢量 v 和 r 之间的夹角，α 表示材质的光泽度(shininess)(后面将介绍光泽度这个概念)。

采用与漫反射类似的方法，用点积定义重写镜面反射公式。假设矢量 r、v、n 和 l 都是单位矢量，镜面反射可以表示成第二种形式：

$$I = k_s I_s \max(r \cdot v, 0)^{\alpha}$$

计算反射矢量 r 的数学公式可以表示为

$$r = 2(l \cdot n)n - l$$

然而，并不需要手工计算反射矢量。OpenGL ES 着色语言内置了一个非常有用的函数 reflect()，它可以根据矢量 l 和法线 n 计算反射矢量 r。然而，你必须小心。reflect()函数假定矢量 l 指定光的方向是从光源指向表面，这正好与定义的 l 的方向相反(如图 7-4 所示)。在下面这段 OpenGL ES 着色语言的代码中，可以看到，在顶点着色器中如何计算镜面反射。

```
uniform vec3 uSpecularMaterial;
uniform vec3 uSpecularLight;
uniform vec3 uLightPosition;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;

const float shininess = 32.0;

main() {

   // Get vertex position in eye coordinates
   vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
   vec3 vertexPositionEye3 = vertexPositionEye4.xyz / vertexPositionEye4.w;

   // Calculate the vector (l) to the light source
   vec3 vectorToLightSource = normalize(uLightPosition - vertexPositionEye3);
```

```
// Transform the normal (n) to eye coordinates
vec3 normalEye = normalize(uNMatrix * aVertexNormal);


// Calculate the reflection vector (r)
vec3 reflectionVector = normalize(reflect(-vectorToLightSource, normalEye));

// The camera in eye coordinates is located in the origin
// and pointing along the negative z-axis.
// Calculate viewVectorEye (v) in eye coordinates as
// (0.0, 0.0, 0.0) - vertexPositionEye3
vec3 viewVectorEye = -normalize(vertexPositionEye3);

fl oat rdotv = max(dot(refl ectionVector, viewVectorEye), 0.0);
fl oat specularLightWeight = pow(rdotv, shininess);

vec3 specularRefl ection = uSpecularMaterial * uSpecularLight *
  specularLightWeight;

...

}
```

与前面漫反射的着色器代码示例一样，在这段计算镜面反射的代码中，也有几行代码用来计算镜面反射所需要的输入，它们将在后面介绍。代码中高亮显示部分与镜面反射计算的计算公式相对应。

光泽度

如前所述，镜面反射的光泽度(shininess)由镜面反射公式中的α来表示。α值较大，表明光泽度高，意味着反射光的强度随矢量r与矢量v之间夹角θ增大而快速降低。α值较小，表明光泽度低，意味着反射光强度随矢量r与矢量v之间夹角θ的增大而缓慢降低。

为了更好地理解光泽度如何影响反射光，图 7-5 显示了因子$\cos(\theta)^\alpha$(这个因子直接影响镜面反射光的光强)如何在不同α下随角度θ的变化而变化。表 7-1 列出与此图对应的数值。

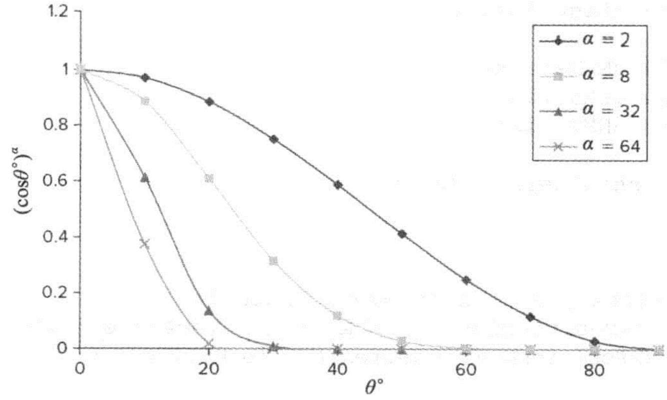


图 7-5　影响镜面反射光强度的$\cos(\theta)^\alpha$的示意图。本图为$\cos(\theta)^\alpha$在光泽度α取不同值时随角度θ(视线方向v与反射方向r之间的夹角)的不同而变化的曲线

表 7-1 光强因子 $\cos(\theta)^{\alpha}$ 随角度 θ 和光泽度 α 的变化所产生的变化

角度 θ°	α=2	α=8	α=32	α=64
0°	1	1	1	1
10°	0.97	0.88	0.61	0.38
20°	0.88	0.61	0.14	0.02
30°	0.75	0.32	0.01	0.00
40°	0.59	0.12	0.00	0.00
50°	0.41	0.03	0.00	0.00
60°	0.25	0.00	0.00	0.00
70°	0.12	0.00	0.00	0.00
80°	0.03	0.00	0.00	0.00
90°	0.00	0.00	0.00	0.00

从图 7-5 的最上面开始，第一条曲线反映了当光泽度 α=2 时(与光泽度较低的情形相对)，$\cos(\theta)^{\alpha}$ 因子随角度 θ 的变化。在这情形下，角度 θ 增大时光强的下降速度却比较缓慢。例如，反射矢量与视线方向的夹角为 30° 时，光强因子只下降到原来的 0.75。

自上而下的第二条曲线说明在 α=8 时光强因子的变化，第三条曲线反映 α=32 时的变化，最下面的曲线反映 α=64 时的变化。当光泽度 α 大到 64 时，可以看出，即使角度只有很小的 10° ，光强因子已下降到原来的 0.38。

关于光泽度还有一件事情需要明白，它只影响光强随 r 与 v 之间夹角增大而下降的速度。这意味着，它只影响镜面的强光范围的大小，而不会影响其强度(即光有多亮)。当视线矢量(v)与反射矢量(r)重合时(即 v=r)，不管光泽度 α 为多少，光强因子 $\cos(\theta)^{\alpha}$ 的值总是 1。

7.3.4 Phong 反射模型的完整公式和着色器

至此，已经介绍了 Phong 反射模型的各个反射分量，现在把它们组合在一起。Phong 反射模型的完整公式可以表示为：

$$I_{\text{phong}} = k_{\text{a}} I_{\text{a}} + k_{\text{d}} I_{\text{d}} \max(n \cdot l, 0) + k_{\text{s}} I_{\text{s}} \max(r \cdot v, 0)^{\alpha}$$

其中，计算反射矢量 r 的数学公式可表示为：

$$r = 2(l \cdot n)n - 1$$

公式中各个变量的含义已经在前几节中详细讨论过，但是为了方便理解，这里再次简单介绍一下。

- I_{phong}——最终得到的光强。
- k_{a}——环境材质属性(反射系数)，该值越大表明反射光越强。
- I_{a}——环境光分量。
- k_{d}——漫反射材质(系数)，该值越大表明反射光越强。

- I_d——漫射光分量。
- n——表面法线的单位矢量。
- l——光线的单位矢量，方向指向光源。
- k_s——镜面反射材质属性(系数)，该值越大表明反射光越强。
- I_s——镜面光分量。
- r——反射光单位矢量。
- v——视线单位矢量，指向观察者。
- α——材质的光泽度。

可以看出，计算最终的光照强度需要考虑很多参数。

必须注意，如果有多个光源，则需要把它们各自的贡献添加到总光强中。有时，Phong 反射模型的公式经常写成∑形式，表示要把所有光源的各个分量累加在一起。

程序清单 7-1 给出了一个完整的顶点着色器，它利用 Phong 反射模型计算光照。与 Phong 反射模型计算公式以及前面各节介绍环境反射、漫反射和镜面反射时使用的代码相比，这段代码稍微简单些。

如果实际的 Phong 反射模型是在着色器代码中实现的，有时需要使用这种简化处理。其思想是，光的各个分量与材质的各个分量已经预先相乘在一起。这意味着，着色器中 uniform 变量 uAmbientLightColor 对应于 k_aI_a，uDiffuseLightColor 对应于 k_dI_d，uSpecularLightColor 对应于 k_sI_s。

如果用户没有打算为不同的顶点定义不同的材质属性，则这种优化处理很有意义。

对于光照，很多重要的代码都出现在顶点着色器和片段着色器中。因此，本章的几个代码段只显示着色器源代码。也会提及 JavaScript 与此有关的代码，但是为了节省篇幅以及不使人感到乏味，不会将它们全部列出。

然而，执行着色器需要完整的源代码(包括 JavaScript 代码)，因此要从 www.wrox.com 网站上下载用到的全部源代码。

注意，从 www.wrox.com 上下载一个 WebGL 程序时，它通常对应本章中的两部分代码，即一个顶点着色器和一个片段着色器。例如，下载本例的源代码时，位于名为 Listing7-1-and-7-2-per-vertex-lighting-Phong-reflection-model-no-textures 文件夹中的 WebGL 程序对应于本章的程序清单 7-1(顶点着色器)和程序清单7-2(片段着色器)。除了顶点着色器和片段着色器外，下载的源代码文件还包含 JavaScript 代码，以便可把完整的 WebGL 程序加载到 Web 浏览器中测试着色器。

程序清单 7-1 采用逐顶点着色模式实现 Phong 反射模型的顶点着色器

```
<script id="shader-vs" type="x-shader/x-vertex">
  // Vertex shader implemented to perform lighting according to the
  // Phong reflection model.

  attribute vec3 aVertexPosition;
  attribute vec3 aVertexNormal;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;
  uniform mat3 uNMatrix;
  uniform vec3 uLightPosition;
  uniform vec3 uAmbientLightColor;
  uniform vec3 uDiffuseLightColor;
  uniform vec3 uSpecularLightColor;

  varying vec3 vLightWeighting;

  const float shininess = 32.0;

  void main() {
    // Get the vertex position in eye coordinates
    vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
    vec3 vertexPositionEye3 = vertexPositionEye4.xyz / vertexPositionEye4.w;

    // Calculate the vector (l) to the light source
    vec3 vectorToLightSource = normalize(uLightPosition - vertexPositionEye3);

  // Transform the normal (n) to eye coordinates
  vec3 normalEye = normalize(uNMatrix * aVertexNormal);

  // Calculate n dot l for diffuse lighting
  float diffuseLightWeightning = max(dot(normalEye,
                                   vectorToLightSource), 0.0);

  // Calculate the reflection vector (r) that is needed for specular light
  vec3 reflectionVector = normalize(reflect(-vectorToLightSource,
                                            normalEye));

  // The camera in eye coordinates is located in the origin and is pointing
  // along negative z-axis. Calculate viewVectorEye (v)
  // in eye coordinates as:
  // (0.0, 0.0, 0.0) - vertexPositionEye3
  vec3 viewVectorEye = -normalize(vertexPositionEye3);

  float rdotv = max(dot(reflectionVector, viewVectorEye), 0.0);

  float specularLightWeightning = pow(rdotv, shininess);
```

```
    // Sum up all three reflection components and send to the fragment shader
    vLightWeighting = uAmbientLightColor +
                      uDiffuseLightColor * diffuseLightWeightning +
                      uSpecularLightColor * specularLightWeightning;

    // Finally transform the geometry
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    }
</script>
```

程序清单 7-1 中执行的光照计算部分代码应该容易理解，因为现在已经知道 Phong 反射模型各分量的计算方法。

重要的是要记住，所有光照计算都必须在同一个坐标系中完成，通常在视坐标系中完成。这意味着，顶点着色器需要先把顶点位置和法线变换到视坐标系中。

顶点变换通常只需要把顶点位置乘以模型视图矩阵，如下所示。

```
vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
```

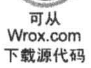
法线通常不可以用模型视图矩阵进行变换。假如用一个矩阵 M 变换某个表面上的一个顶点，则需要用 M 的逆转置矩阵 M^{-T} 乘以表面的法线。另一个方法是使用 M 的上角 3×3 的转置逆矩阵。在这段代码中，使用了第二种方法。这可能会让人困惑，但本章后，你还要进一步学习有关法线变换的内容。

着色器中的 uniform 变量 uNMatrix 代表了 uMatrix 的左上角 3×3 逆转置矩阵。用于变换法线的矩阵是在 JavaScript 代码中计算的(这里没有介绍 JavaScript，将在本章后面介绍)，并把它作为一个 uniform 变量发送给着色器。然后在着色器中用下面的代码未变换法线。

```
vec3 normalEye = normalize(uNMatrix * aVertexNormal);
```

如果是在顶点着色器中进行光照计算并且没有使用纹理，则片段着色器就比较简单。在程序清单 7-2 中，片段着色器与程序清单 7-1 中的顶点着色器一起工作。它要做好是把 varying 变量 vLightWeighting 扩展为齐次坐标，再把它赋给内置变量 gl_FragColor。

程序清单 7-2　在顶点着色器中实现 Phong 反射模型时所用的片段着色器




```
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  varying vec3 vLightWeighting;

  void main() {
    gl_FragColor = vec4(vLightWeighting.rgb, 1.0);
  }
</script>
```

图 7-6 所示为这个基本的 WebGL 程序所用的程序清单 7-1 中顶点着色器和程序清单 7-2 中片段着色器的绘制结果。这个 WebGL 程序绘制没有应用纹理的地板、桌子和盒子。桌

面和地板几乎是纯白色，因为灯光位于它们的正上方。桌腿是深灰色。因为只有微弱的光照射到它们。在 7.3.5 小节中，将会知道，如何方便地利用这些着色器，以便把光照效果作用到纹理对象。

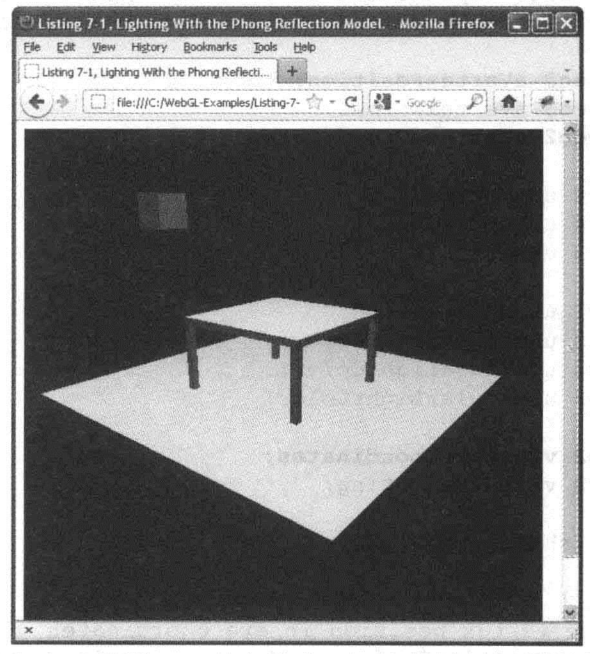


图 7-6　有光照效果但未使用纹理的 WebGL 场景

7.3.5　光照效果与纹理相结合

在第 5 章中已学过纹理处理方法，并知道将纹理用于对象时对象看起来会更真实。现在自然会希望将光照与纹理结合起来。

如果熟悉纹理的处理过程和光照的计算，则很容易将两者结合在一起。可以像通常那样执行光照计算，然后用片段着色器把采样到的纹素颜色与光照计算得到的颜色结合在一起。

最常用的方法是按分量逐个把光照计算得到的值与纹理采样到的颜色值相乘。这种分量逐个相乘的模式有时也称为调制模式，如下所示。

```
gl_FragColor = vec4(vLightWeighting.rgb * texelColor.rgb, texelColor.a);
```

很快就有机会看到完整的片段着色器，但是首先需要看一下顶点着色器。在顶点着色器中，与纹理处理有关的唯一操作是把纹理坐标作为属性(attribute)变量的输入，并把这个属性变量写入 varying 变量中。因此，程序清单 7-3 中的顶点着色器与程序清单 7-1 中的顶点着色器几乎完全一样。仅有的几处不同(都与纹理坐标有关)都已在代码中用粗体显示。由于这段代码有很多注释，同时大部分内容已经在程序清单 7-1 中详细介绍过，因此程序清单 7-3 中顶点着色器的代码不需要赘述。

可从
Wrox.com
下载源代码

程序清单 7-3　Phong 反射模型的顶点着色器

```
<script id="shader-vs" type="x-shader/x-vertex">
  // Vertex shader implemented to perform lighting according to the
  // Phong reflection model. Forwards texture coordinates to fragment
  // shader.
  attribute vec3 aVertexPosition;
  attribute vec3 aVertexNormal;
  attribute vec2 aTextureCoordinates;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;
  uniform mat3 uNMatrix;

  uniform vec3 uLightPosition;
  uniform vec3 uAmbientLightColor;
  uniform vec3 uDiffuseLightColor;
  uniform vec3 uSpecularLightColor;

  varying vec2 vTextureCoordinates;
  varying vec3 vLightWeighting;

  const float shininess = 32.0;

  void main() {
    // Get the vertex position in eye coordinates
    vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
    vec3 vertexPositionEye3 = vertexPositionEye4.xyz / vertexPositionEye4.w;

    // Calculate the vector (l) to the light source
    vec3 vectorToLightSource = normalize(uLightPosition - vertexPositionEye3);

    // Transform the normal (n) to eye coordinates
    vec3 normalEye = normalize(uNMatrix * aVertexNormal);

    // Calculate n dot l for diffuse lighting
    float diffuseLightWeightning = max(dot(normalEye,
                                     vectorToLightSource), 0.0);

    // Calculate the reflection vector (r) that is needed for specular light
    vec3 reflectionVector = normalize(reflect(-vectorToLightSource,
                                              normalEye));

    // The camera in eye coordinates is located in the origin and is pointing
    // along the negative z-axis. Calculate viewVectorEye (v)
    // in eye coordinates as:
    // (0.0, 0.0, 0.0) - vertexPositionEye3
    vec3 viewVectorEye = -normalize(vertexPositionEye3);

    float rdotv = max(dot(reflectionVector, viewVectorEye), 0.0);

    float specularLightWeightning = pow(rdotv, shininess);

    // Sum up all three reflection components and send to the fragment shader
```

```
    vLightWeighting = uAmbientLightColor +
                      uDiffuseLightColor * diffuseLightWeightning +
                      uSpecularLightColor * specularLightWeightning;

    // Finally transform the geometry
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vTextureCoordinates = aTextureCoordinates;
  }
</script>
```

程序清单 7-4 所示为结合了纹理处理与光照的片段着色器。这从这段代码可以看出，纹理处理先经过采样，然后把采样值与光照计算结果相乘。光照计算结果是作为 varying 变量 vLightWeighting 传入的。实际的光照计算是由相应的顶点着色器完成的(见程序清单 7-3)。

程序清单 7-4 结合了纹理和光照的片段着色器

可从 Wrox.com 下载源代码

```
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;

  varying vec2 vTextureCoordinates;
  varying vec3 vLightWeighting;
  uniform sampler2D uSampler;

  void main() {
    vec4 texelColor = texture2D(uSampler, vTextureCoordinates);

    gl_FragColor = vec4(vLightWeighting.rgb * texelColor.rgb, texelColor.a);
  }
</script>
```

图 7-7 所示为基本 WebGL 应用程序中应用了程序清单 7-3 中顶点着色器和程序清单 7-4 的片段着色器的绘制结果。它绘制了应用了纹理的地板、桌子和盒子。

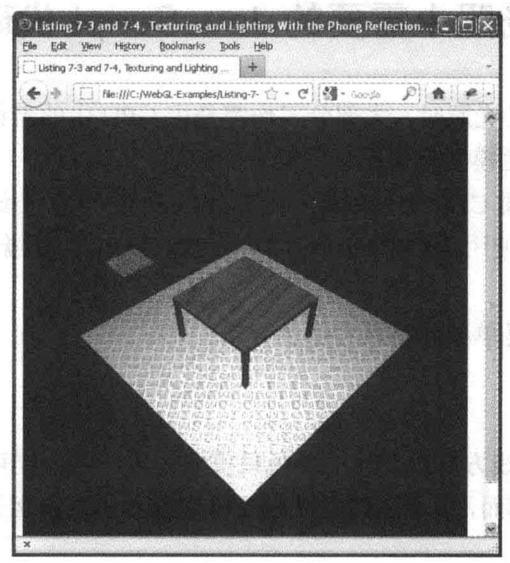


图 7-7 光照作用于纹理对象的结果

用迟后添加调制模式创建镜面高光效果

程序清单 7-4 使用了基本调制模式的一个变异形式，它把计算的镜面光与环境光和漫射光的计算分离开来。只把环境光和漫射光相结合的结果与采样的纹理颜色一起调制，如下所示。

```
void main() {
   vec4 texelColor = texture2D(uSampler, vTextureCoordinates);

   gl_FragColor = vec4(vLightWeighting.rgb * texelColor.rgb +
                       vSpecularLightWeighting.rgb, texelColor.a);
 }
```

在这段代码中，假设 varying 变量 vLightWeighting 包含已经计算得到的环境光和漫射光，变量 vSpecularLightWeighting 包含已经计算得到的镜面光。可以看出，镜面光是在调制之后添加的。这就是所谓的“迟后添加调制模式”(modulation with a late add)。

使用这种方式的理由是：在 OpenGL ES 着色器语言中和 WebGL 中，有效的颜色值是在 0.0 和 1.0 之间。这意味着，至少在理论上，环境光、漫射光和镜面光的最大值是白色，即表示为(1.0,1.0,1.0,1.0)。如果将采样的纹素颜色乘以该值，就得到原始的纹素颜色。这意味着,没有一个有效的颜色表示可以与采样的纹素颜色相乘进而得到在镜面高亮处的颜色。这正是使用第二种方法的原因。在调制之后添加镜面反射项的颜色，就可以添加更明亮的镜面高亮效果。

然而在实际上，环境光、漫射光和镜面光相加所达到的值，其每个颜色通道值会大于 1.0。这意味着，不必单独保留镜面反射项并在调制之后添加这一项的值。但是有必要知道着色器代码是否按这种模式编写。

7.4 WebGL 光照中需要的 JavaScript 代码

在处理 WebGL 中的光照时，大部分工作都是由 GPU 中的着色器完成的。正如本章的前几节所述，Phong 反射模型就是在着色器中实现的。

在 JavaScript 中需要 CPU 处理的操作通常仅限于向 GPU 传送光照计算所需要的输入数据。例如，通常需要使用 JavaScript 代码向着色器传送以下数据。

- 光源位置或方向
- 灯光颜色(或许是单独的材质和颜色分量)
- 顶点法线
- 法线矩阵

如果光源位置或光线方向以及光的颜色只需要设置一次，则可以在着色器中通过硬编码设置(即直接赋值)。然而，顶点法线和法线矩阵则通常需要由运行在 CPU 中的 JavaScript 代码来传送。

从 JavaScript 代码发送顶点法线，是为了计算通用网格上的顶点法线，通常还需要使用相邻顶点，以确定表面的朝向。但是顶点着色器通常不包含关于相邻顶点的信息，因此通常要从 JavaScript 传送法线矢量。理论上，法线矩阵可以在着色器中计算，但是实际上通常它是在 CPU 中由 JavaScript 计算的，然后作为 uniform 变量传递。

在 JavaScript 中光照计算需要以下步骤：

(1) 设置包含顶点法线的缓存。

(2) 根据模型视图矩阵计算法线矩阵，并将其作为 uniform 变量传送给着色器。

(3) 向顶点着色器发送光源信息(光源位置、光照方向及光的颜色)。

下面几小节将更详细地讨论这些步骤。

7.4.1　为顶点法线设置缓存

假设 3D 对象是一个立方体。对于这种简单情形，无需计算就很容易得到立方体每个面的法线。

- 正面的法线为(0.0,0.0,1.0)。
- 背面的法线为(0.0,0.0,-1.0)。
- 左侧面的法线为(-1.0,0.0,0.0)。
- 右侧面的法线为(1.0,0.0,1.0)。
- 顶面的法线为(0.0,1.0,0.0)。
- 底面的法线为(0.0,-1.0,0.0)。

根据前面的描述，图 7-8 显示了一个立方体 6 个面的表面法线。注意，在给一个立方体或 WebGL 其他任何对象指定法线时，通常需要为每个顶点定义法线，而不是像图 7-8 那样为每个面定义法线。图 7-8 的例子只是用来说明立方体各个法线的方向。

立方体每个面的法线与坐标系指向相同的方向。如果对象不是这样的简单立方体，则不经过计算很难确定每个面的法线。但是，由于已经在第 1 章中掌握了 3D 图形的线性代数知识，因此知道大多数情形下可以利用矢量叉乘计算三角形理论上的法线。图 7-9 说明如何计算顶点为 V_0、V_1、V_2 的三角形的法线。

有关矢量叉乘的更多内容，可以参考本书的第 1 章。

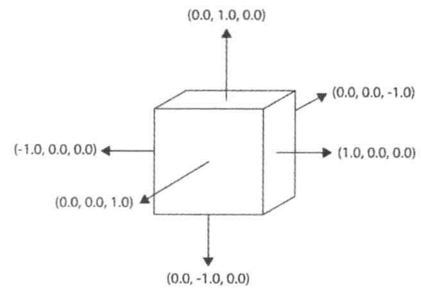


图 7-8　立方体各面的法线

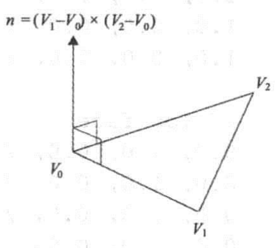


图 7-9　用矢量叉乘计算三角形的法线

如果想在 JavaScript 计算法线，一种方法是使用 JavaScript 的 glMatrix 库中的叉乘计算函数：

```
vec3.cross(vector1, vector2, normal);
```

这里需要注意的是，在把法线正确应用于本章前面介绍的光照计算之前必须对得到的法线进行规范化。

如果对象是用 Blender 或 Maya 等 3D 建模程序创建的，则这些程序通常会自动生成顶点的法线。把这样的模型导入到自己的 WebGL 应用程序时，需要解析顶点数据(顶点位置和顶点法线)，然后再将其加载到缓存中。

为顶点法线创建和绑定缓存然后把数据加载到缓存的代码，与把其他顶点数据(位置数据和纹理数据)加载到缓存的代码非常相似。在下面这段 JavaScript 代码中，可以看到如何把一个立方体的顶点法线加载到缓存。

```
pwgl.cubeVertexNormalBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.cubeVertexNormalBuffer);

var cubeVertexNormals = [
       // Front face
       0.0, 0.0, 1.0, //v0
       0.0, 0.0, 1.0, //v1
       0.0, 0.0, 1.0, //v2
       0.0, 0.0, 1.0, //v3

       // Back face
       0.0, 0.0, -1.0, //v4
       0.0, 0.0, -1.0, //v5
       0.0, 0.0, -1.0, //v6
       0.0, 0.0, -1.0, //v7

       // Left face
      -1.0, 0.0, 0.0, //v8
      -1.0, 0.0, 0.0, //v9
      -1.0, 0.0, 0.0, //v10
      -1.0, 0.0, 0.0, //v11

       // Right face
       1.0, 0.0, 0.0, //12
       1.0, 0.0, 0.0, //13
       1.0, 0.0, 0.0, //14
       1.0, 0.0, 0.0, //15

       // Top face
       0.0, 1.0, 0.0, //v16
       0.0, 1.0, 0.0, //v17
       0.0, 1.0, 0.0, //v18
       0.0, 1.0, 0.0, //v19
```

```
        // Bottom face
        0.0, -1.0, 0.0, //v20
        0.0, -1.0, 0.0, //v21
        0.0, -1.0, 0.0, //v22
        0.0, -1.0, 0.0, //v23
];

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(cubeVertexNormals),
              gl.STATIC_DRAW);

pwgl.CUBE_VERTEX_NORMAL_BUF_ITEM_SIZE = 3;
pwgl.CUBE_VERTEX_NORMAL_BUF_NUM_ITEMS = 24;
```

7.4.2 计算法线矩阵并上传给着色器

法线的变换可能会让人困惑。因为，对法线的变换不同于对顶点或其他矢量的变换。但是法线变换的基本规则还是很容易掌握的。

更让人困惑的是，掌握了基本规则后，可能会遇到一些并未遵守规则的代码示例或其他人的建议。法线变换有 3 种主要选择。

- 通常，不能把应用于对象顶点变换的模型视图矩阵应用于对象表面法线变换。如果用矩阵 M 变换某个表面上的一个顶点，则变换这个表面的法线要用这个 M 的转置逆矩阵，即 M^T。这个基本规则适用于大多数情形。
- 由于法线是矢量，因此可以对这个规则做第一次优化。假设变换顶点的矩阵为 M，则可以用 M 的左上角 3×3 矩阵的转置逆矩阵来变换法线。
- 在某些情形，变换还可以进一步优化。如果知道矩阵 M 只是由以下变换组成(这种情况并不少见)，则可以直接使用原始矩阵：
  - 平移
  - 旋转
  - 均匀缩放(没有拉伸或挤压)

如果已经掌握用矩阵的转置逆矩阵变换法线并看到一些使用其他方法的代码，就会产生困惑。但是，掌握了这些规则后，就不难看懂或编写法线变换的代码。

本书中使用第二种方法：用 M 左上角 3×3 矩阵的转置逆矩阵变换法线。这个方法得到 JavaScript 库 glMatrix 的支持，本书的绝大部分示例都使用这个方法。glMatrix 库有一个名为 mat4.toInverseMat3()的方法，它作用于一个原始的 4×4 矩阵得到其左上角 3×3 矩阵。之后，3×3 矩阵的转置变换要调用 mat3.transpose()方法。下面这段代码说明了从模型视图矩阵计算法线矩阵并把法线矩阵上传着色器的 uniform 变量的过程。

```
var normalMatrix = mat3.create();
mat4.toInverseMat3(pwgl.modelViewMatrix, normalMatrix);
mat3.transpose(normalMatrix);
gl.uniformMatrix3fv(pwgl.uniformNormalMatrixLoc, false, normalMatrix);
```

7.4.3 将光照信息上传给着色器

如果还没有在着色器中用硬编码方法设置光照信息，则需要从 JavaScript 中将该数据上传给着色器。假设要通过 uniform 变量上传光源数据，则使用的方法与将其他数据上传给 uniform 变量一样。首先需要确定相关的 uniform 变量在当前程序对象中的位置。这可以用 gl.getUniformLocation()方法来实现，具体如下所示。

```
pwgl.uniformLightPositionLoc =
  gl.getUniformLocation(shaderProgram, "uLightPosition");

pwgl.uniformAmbientLightColorLoc =
  gl.getUniformLocation(shaderProgram, "uAmbientLightColor");

pwgl.uniformDiffuseLightColorLoc =
  gl.getUniformLocation(shaderProgram, "uDiffuseLightColor");

pwgl.uniformSpecularLightColorLoc =
  gl.getUniformLocation(shaderProgram, "uSpecularLightColor");
```

然后，必须把这些 uniform 变量的值上传给 GPU。在这里，所有 uniform 变量都包含 3 个分量的矢量，因此要用 gl.uniform3fv()方法。具体代码如下所示。

```
function setupLights() {
  gl.uniform3fv(pwgl.uniformLightPositionLoc, [0.0, 15.0, 5.0]);
  gl.uniform3fv(pwgl.uniformAmbientLightColorLoc, [0.2, 0.2, 0.2]);
  gl.uniform3fv(pwgl.uniformDiffuseLightColorLoc, [0.7, 0.7, 0.7]);
  gl.uniform3fv(pwgl.uniformSpecularLightColorLoc, [1.0, 1.0, 1.0]);
}
```

7.5 将不同的插值方法用于着色

在第 1 章中首次介绍顶点着色器和片段着色器时，主要介绍了 3D 图形中着色的实际含义。现在要更详细地介绍着色计算。

着色是决定一个顶点、图元(如三角形)或对象的颜色的过程，例如：

- 光源的位置
- 离光源的距离
- 光的颜色
- 法线矢量(或着色位置的朝向)
- 材质属性

当然，最常见的情形是在 3D 场景不只有一个顶点。即使在本书的第一个示例中，也至少需要绘制一个三角形，在实际的 WebGL 应用程序中，通常会有由许多三角形组成的更高级的模型。如果着色计算是指决定一个三角形或由很多三角形组成的对象的颜色的过程，则着色过程通常由以下两个不同的部分组成。

- 光照模型
- 插值技术

光照模型定义了如何利用光的颜色、材质和不同角度计算指定点的颜色。换句话说，光照模型确定了某个点颜色的计算公式。例如，已经学习了如何用 Phong 反射模型计算某个点的颜色。

插值技术是一个如何根据顶点位置的颜色确定对象上其他位置的颜色的过程。在本章前面的例子中，已经计算了每个顶点的颜色。已知顶点的颜色后，片段的颜色要通过顶点颜色的线性插值确定。但是计算顶点的颜色然后用线性插值确定片段的颜色并不是唯一的解决方案。

着色使用以下 3 种方法进行插值计算。

- 平面着色
- Gouraud 着色
- Phong 着色

有时这些方法是指着色的不同方式。一般来说，平面着色最简单，结果最差；Gouraud 着色需要很多的计算，结果比较好；Phong 着色是这 3 种方法中最高级的，结果最好。以下几小节将详细介绍这些方法。

7.5.1 平面着色

平面着色根据每个三角形的法线计算着色效果。根据它的法线，着色计算只执行一次，整个三角形都采用计算结果的颜色。必须指出，在 WebGL(或 OpenGL ES 2.0)中，平面着色通常并不吸引人，因为在 WebGL(OpenGL ES 2.0)中，通常为每个顶点定义一个法线。然而，如果在 WebGL 中使用原始的 gl.TRIANGLES 并且为每个三角形的 3 个顶点提供相同法线，得到的结果就是平面着色。

这种着色模式可以用图 7-10 来说明。此图表示一个由 3 个三角形的可见边 T_0、T_1、T_2 组成的表面(既然每个三角形只看到一条边，因此就用边表示相应的三角形)。此外，也会看到每个三角形的两条法线。因为三角形 T_0 的法线指向一个方向，三角形 T_1 的法线指向另一个方向，T_2 的法线指向第三个方向，所以，经过着色计算后，这 3 个三角形的颜色不同。但是，如果光源和观察者远离这个平面，则各三角形的颜色都是一个常量，这就是平面着色的效果。

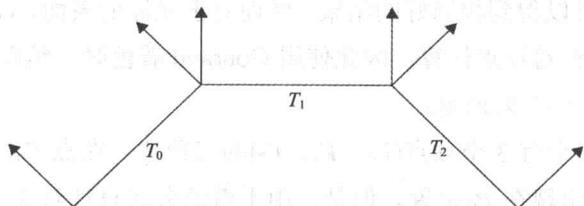


图 7-10　由 3 个三角形组成的表面，法线朝向不同因此得到的结果像是平面着色

为了理解平面着色的局限性，必须考虑平面着色的前提。在以下这些情形，使用平面着色是可接受的。

- 光源无限远时，入射光线的角度在整个三角形上是一样的(无限远的光源通常称为平行光，本章后面还将进一步介绍)。
- 观察者无限远时，三角形上任何位置到观察者的角度都相同。
- 三角形代表一个平面(如立方体的一个面)时，而不是曲面的近似表示时。

通常如果这些条件中的一个或多个不能成立，因而平面着色得到不满意的效果。特别当三角形是曲面近似表示(如球)的一部分时，结果尤其如此，这种情况下，平面着色的结果像是表面由多个小面片组成，而不是一个预期的光滑表面。三角形越大，曲面的平面着色结果越差。幸好在 WebGL 中没有理由用平面着色近似表示曲面。

7.5.2 Gouraud 着色

Gouraud 着色是以法国计算机科学家 Henri Gouraud 的名字命名的，他于 1971 年发明了该方法并且公开发表了他的发现。这个方法有时也称为逐顶点着色(或逐顶点光照处理)，这是因为着色是针对每个顶点计算的，而后对每个顶点的结果颜色进行线性插值得到片段的颜色。Gouraud 着色的代码用于程序清单 7-1 和程序清单 7-3 中，已在 Phong 反射模型中介绍过。

如果一个三角形是曲面(如球)近似表示的一部分，则每个顶点的法线不应该是这个三角形纯数学上的精确法线。而是要使用一个考虑到要建模的基层曲面的朝向的法线(如图 7-11 所示)。如果采用平均法线，则从一个三角形到相邻三角形的颜色过渡过程会比较光滑，因此三角形之间的缝隙就不会像平面着色那样明显。

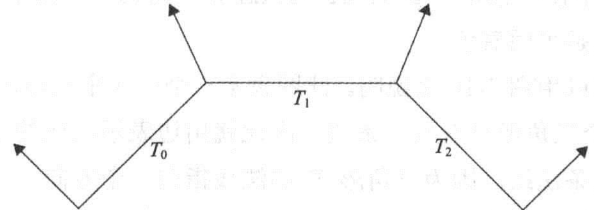


图 7-11　由 3 个三角形组成的表面，顶点的法线是相邻三角形的数学上精确法线的平均值

因此，对于曲面，Gouraud 着色产生的结果比平面着色好。通常，对于大多数粗糙的表面，Gouraud 着色可以得到相当好的结果。然而对于光泽的表面，Gouraud 着色会生成伪象。由于着色公式只针对顶点计算，因此使用 Gouraud 着色时，镜面高光会在顶点之间减弱。其原因可以用图 7-12 来说明。

图 7-12 所示为一个有 3 个顶点(V_0、V_1、V_2)的三角形。在点 P_0，反射矢量 r 和视线 v 重合，因此镜面高光出现在 P_0 位置。但是，由于着色公式只针对 3 个顶点(V_0、V_1、V_2)进行计算，因此会忽略镜面高光。

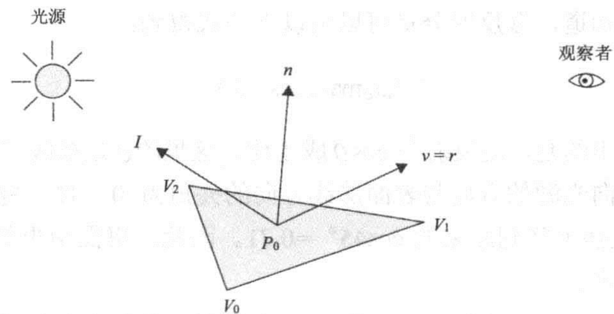


图 7-12 使用 Gouraud 着色模型时，不会出现落在三角形内部的镜面高光

7.5.3 Phong 着色

Phong 着色是以科学家 Bui Tuong Phong 的名字命名的(提出 Phong 反射模型的同一人)，它对三角形的每个片段进行着色计算。有时，这个方法也称为逐片段着色(或逐片段光照)，后者实际上更好地描述了该方法如何工作。

由于颜色是按片段着色的，因此得到的结果比此前介绍的方法要好，用于光亮表面尤其如此。使用 Phong 着色时，对于光亮对象，不会像 Gourand 着色一样创建伪象。也不会忽略三角形内镜面光照射到的位置。

即使不考虑镜面高光，Phong 着色得到的结果通常也比 Gouraud 着色好。图 7-13 表示一个桌面的草图，它在本书前面的一些例子中出现过。但是现在假设在桌子上方挂着一盏灯，它离桌面很近。图中显示了这盏灯的 3 条光线。在顶点 V_0 和 V_1，光线与表面法线的夹角均为 45°。在桌面正中间的 P_0 位置，光线与法线的夹角为 0°。

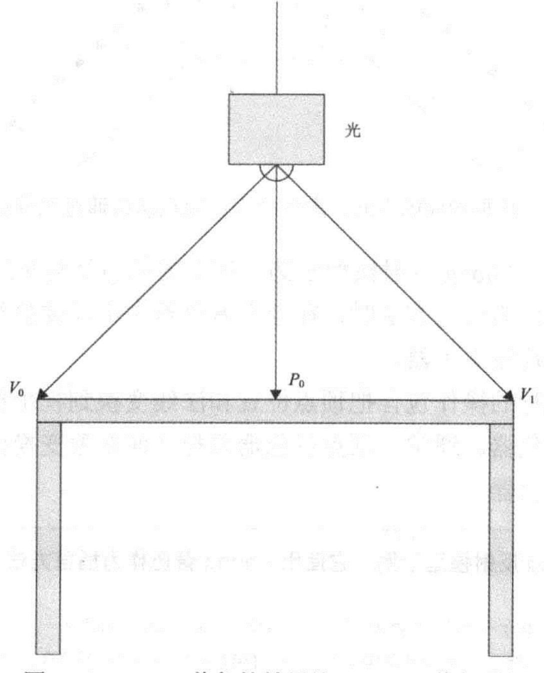


图 7-13　Phong 着色的效果比 Gouraud 着色更好

从本章前面已经知道，漫反射分量可以由以下公式得到：

$$I=k_dI_d\max(\cos\theta,0)$$

这里需要特别指出的是，漫反射与 $\cos\theta$ 成正比，这里的 θ 是法线与指向光源方向的夹角。这意味着，当指向光源的方向与表面法线方向的夹角为 0° 时，漫反射最大。当该夹角为 45° 时，反射光强下降到原来的 cos45° =0.71。因此，桌面中央的亮度比 V_0 和 V_1 位置边缘的亮度要大许多。

但是，如果使用 Gouraud 着色，桌面正中央 P_0 位置的亮度不会比其他位置更亮。因为着色公式只计算顶点 V_0 和 V_1 的情况，因此桌面的这些位置看起来比较暗。然后，对结果按片段进行线性插值，因此在 P_0(桌面正中间)的光强也比较暗。

如果使用 Phong 着色，则传送给顶点着色器的法线作为 varying 变量传送给片段着色器。知道，所有 varying 变量都要经过线性插值计算，并利用线性插值得到的法线在片段着色器中计算每个片段的光照。则在本例中，这意味着，对桌面正中间 P_0 位置也要进行光照计算，因此该位置要比边缘 V_0 和 V_1 位置亮。

本例假设灯光是一个点光源，即光源向四周发射光。看完本章内容后就会知道，如果本例的光源建模为聚光灯且考虑到衰减因素，Phong 着色与 Gouraud 着色之间的差别会更大。但是目前不必考虑这些问题，看完本章内容时就会明白。

为了完整起见，图 7-14 在同一个表面上同时显示平面着色和 Gouraud 着色的法线。4 条实线代表位于顶点的法线，它们作为属性(attribute)传递给顶点着色器。虚线则是片段着色器中线性插值得到的法线。

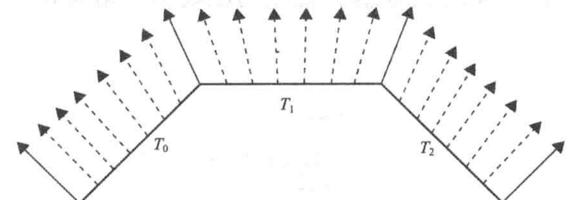


图 7-14　由 3 个三角形构成的表面，虚线代表由顶点法线通过线性插值得到的法线

程序清单 7-5 所示为 Phong 反射模型示例。该例的插值计算采用 Phong 着色模型。可以看出，以 Phong 着色作为插值方法时，在顶点着色器中不需要执行多少操作，因为实际的光照计算现已转移到片段着色器。

顶点着色器需要执行的操作包含把顶点位置和法线变换到视坐标，并通过 varying 变量把它们传送给片段着色器。通常，顶点着色器要把几何对象变换到裁剪坐标系，也要把纹理坐标传递给片段着色器。

程序清单 7-5　Phong 反射模型示例，它使用 Phong 着色作为插值方法

```
<script id="shader-vs" type="x-shader/x-vertex">
   // Vertex shader implemented to perform lighting according to the
   // Phong reflection model. The interpolation method that is used is
```

```
    // Phong shading (per-fragment shading) and therefore the actual
    // lighting calculations are performed in the fragment shader.
    attribute vec3 aVertexPosition;
    attribute vec3 aVertexNormal;
    attribute vec2 aTextureCoordinates;

    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;
    uniform mat3 uNMatrix;

    varying vec2 vTextureCoordinates;
    varying vec3 vNormalEye;
    varying vec3 vPositionEye3;

    void main() {
    // Get vertex position in eye coordinates and send to the fragment shader
    vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
    vPositionEye3 = vertexPositionEye4.xyz / vertexPositionEye4.w;

    // Transform the normal to eye coordinates and send to fragment shader
    vNormalEye = normalize(uNMatrix * aVertexNormal);

    // Transform the geometry
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vTextureCoordinates = aTextureCoordinates;
    }
</script>
```

对于Phong着色(逐片段着色)而言，绝大部分光照计算都在片段着色器中完成(见程序清单7-6)。实际的源代码应该不难理解，因为在前面的顶点着色器中，已经用几乎完全相同的代码实现了Phong反射模型的计算。

注意，由于需要在片段着色器中完成光照计算，因此也需要在片段着色器中定义光照计算所需要的uniform变量。光照计算结束后，将计算的结果与采样纹素的颜色相乘，就会把纹理与光照效果结合起来。

可从Wrox.com下载源代码

程序清单7-6 用Phong着色(逐片段着色)实现Phong反射模型时的片段着色器

```
<script id="shader-fs" type="x-shader/x-fragment">
    // Fragment shader implemented to perform lighting according to the
    // Phong reflection model. The interpolation method that is used is
    // Phong shading (per-fragment shading) and therefore the actual
    // lighting calculations are implemented here in the fragment shader.
    precision mediump float;

    varying vec2 vTextureCoordinates;
    varying vec3 vNormalEye;
    varying vec3 vPositionEye3;
```

```
    uniform vec3 uLightPosition;
    uniform vec3 uAmbientLightColor;
    uniform vec3 uDiffuseLightColor;
    uniform vec3 uSpecularLightColor;

    uniform sampler2D uSampler;

    const float shininess = 32.0;

    void main() {

      // Calculate the vector (l) to the light source
      vec3 vectorToLightSource = normalize(uLightPosition - vPositionEye3);

      // Calculate n dot l for diffuse lighting
      float diffuseLightWeighting = max(dot(vNormalEye,
                                      vectorToLightSource), 0.0);

      // Calculate the reflection vector (r) that is needed for specular light
      vec3 reflectionVector = normalize(reflect(-vectorToLightSource,
                                           vNormalEye));

      // Camera in eye space is in origin pointing along the negative z-axis.
      // Calculate viewVector (v) in eye coordinates as
      // (0.0, 0.0, 0.0) - vPositionEye3
      vec3 viewVectorEye = -normalize(vPositionEye3);

      float rdotv = max(dot(reflectionVector, viewVectorEye), 0.0);

      float specularLightWeighting = pow(rdotv, shininess);

      // Sum up all three reflection components
      vec3 lightWeighting = uAmbientLightColor +
                          uDiffuseLightColor * diffuseLightWeighting +
                          uSpecularLightColor * specularLightWeighting;
      // Sample the texture
      vec4 texelColor = texture2D(uSampler, vTextureCoordinates);
      // modulate texel color with lightweigthing and write as final color
      gl_FragColor = vec4(lightWeighting.rgb * texelColor.rgb, texelColor.a);
    }
  </script>
```

7.6 矢量必须归一化

已经知道，如果两个矢量都是单位矢量，则它们的点积可以用来计算它们夹角的余弦。在着色器中经常利用这一技术。矢量必须是单位长度这个要求也正是在顶点着色器和片段着色器中经常使用 OpenGL ES 着色语言内置函数 normalize()的原因。在下面几节中，将介

绍何时矢量需要归一化，何时矢量不需要归一化。

7.6.1 顶点着色器中的矢量归一化

作为属性传递给顶点着色器的矢量通常需要在顶点着色器中进行变换处理，矢量经过变换处理后需要归一化。现在以法线矢量为例。

```
// Transform the normal to eye coordinates and send to fragment shader
vNormalEye = normalize(uNMatrix * aVertexNormal);
```

通常，调用 normalize()函数有两个理由。

- 作为属性从 JavaScript 传入的法线可能不是单位矢量。
- 作为属性从 JavaScript 代码传入的法线是单位矢量，经过变换处理后，它也可能会改变其长度。

如果从 JavaScript 代码传送过来的矢量是单位矢量而且变换矩阵没有改变它的长度，则可以跳过前面示例中的归一化处理。只包含旋转和平移的变换矩阵不会改变矢量的长度，且如果发送给顶点着色器的矢量是单位矢量，则经过这样的变换后，它仍然是单位矢量。

如果使用 mat4.lookAt()方法设置模型视图矩阵，而且除了旋转和平移，没有应用其他任何变换，则模型视图变换矩阵只包含旋转变换和平移变换。如果确保变换的矢量是来自 JavaScript 代码的单位矢量，则在这种情形，实际上可以在顶点着色器中跳过归一化步骤。

7.6.2 片段着色器的矢量归一化

通常，作为 varying 变量从顶点着色器传递给片段着色器的矢量，即使在顶点着色器中已经归一化，在片段着色器中也需要再次归一化。这是因为，线性插值通常会改变矢量的长度(如图 7-15 所示)。

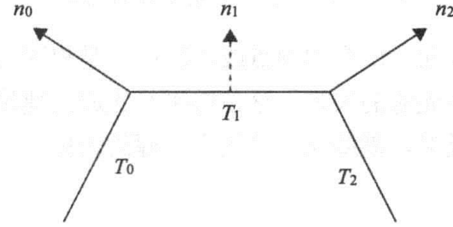


图 7-15 法线矢量 n_0 和 n_2 线性插值后得到的矢量 n_1 比单位矢量短

图 7-15 说明，两个法线单位矢量 n_0 和 n_1，经过插值后得到法线矢量 n_1。在该例中，n_1 比单位矢量短，因此需要在片段着色器中进行归一化处理，使其为单位矢量。

还必须注意，如果顶点的法线不是单位矢量，则不仅在片段着色器中得到的法线长度不正确，通常方向也不正确。这意味着，为了使归一化后的法线有正确的方向，即使法线矢量只用于片段着色器中的计算，在顶点着色器中也需要归一化顶点的法线。

也有一个例外情形，不需要归一化片段着色器中的矢量。这是指当所有作为 varying 变量从顶点着色器传送过来的矢量都是单位矢量，而且所有顶点的法线方向相同时。这些

矢量经过插值，得到的矢量都是单位矢量，而且它们的方向与顶点法线的方向相同。

这种例外情形的一个示例是平行光照射到表面。平行光来自无限远处的光源，因此对于所有顶点而言，光线的方向相同(7.7 节将进一步介绍平行光)。

7.7 应用不同类型的光源

前面已经介绍了不同的反射(环境反射、漫射反射和镜面反射)和光在 Phong 反射模型中的分量。此外，还介绍了着色使用的不同插值技术。

但是至此还没有讨论光源的各种类型和它们的发射方式。在前面的例子中，已经看到光源被放在指定的位置，并且向各个方向发射光。这类光源称为点光源(point light)。在下面几小节中，将分析点光源与另外两类常见的光源平行光(directional Light)和聚光(spot light)的区别。

7.7.1 平行光

平行光(Directional Light)的光源离被照射表面无限远。由于它离表面无限远，入射到表面的光线彼此平行，而且光线的方向对于表面上的顶点或片段都相同(如图 7-16 所示)。这意味着是用方向而非位置来定义这样的光源。

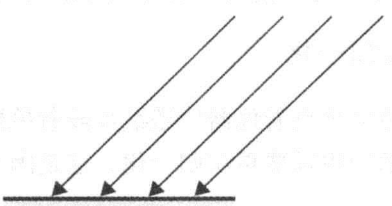


图 7-16 平行光发射平行的光线

由于光线的方向对于表面上所有的顶点或片段而言是相同的，因此不需要像点光源那样为每个顶点或片段计算到光源的矢量。这使得平行光源的建模比点光源更简单。

为了计算点光源的漫反射，需要先计算到点光源的矢量，然后利用点积计算漫反射，具体过程如下所示。

```
// Calculate the vector (l) to the light source. This is NOT needed for
// for a directional light since you typically already have
// vectorToLightSource.
vec3 vectorToLightSource = normalize(uLightPosition - vPositionEye3);

// Calculate n dot l for diffuse lighting
float diffuseLightWeighting = max(dot(vNormalEye,
                                      vectorToLightSource), 0.0);
```

对于平行光，第一步计算不需要，可以直接计算法线矢量与指向光源的矢量的点积。一个常见的示例是用平行光模拟照射地球的太阳光。

7.7.2 点光源

点光源位于某个特定位置且向各个方向发射光(如图 7-17 所示)。

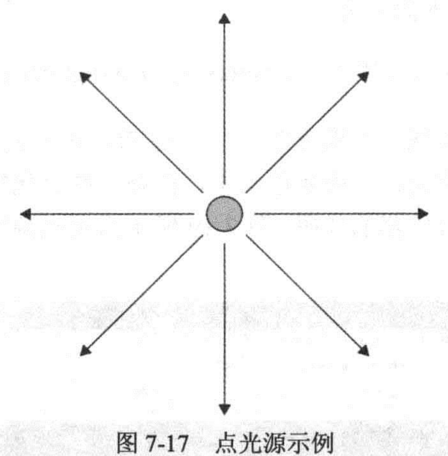


图 7-17 点光源示例

前面已经介绍了一些使用点光源的程序示例，因此本小节不再特别介绍使用点光源的代码。但是，有一点必须记住，有时点光源使用衰减模型。这个问题有待讨论。本章后面将要介绍光源的衰减问题。

7.7.3 聚光源

聚光源(Spot light)在某个方向按圆锥形状发射光。真实世界中一个可以模拟为聚光源的例子是汽车前灯。聚光源可以用不同方法建模。图 7-18 的几何结构可用于理解本书的聚光源。

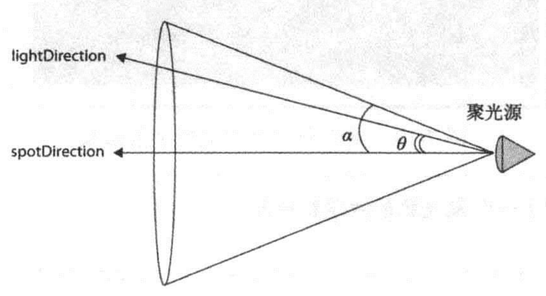


图 7-18 聚光源的几何模型

在图 7-18 中，spotDirection 表示聚光源的朝向。它沿着光锥的轴方向。lightDirection 表示聚光源到点(片段或顶点)的方向，光照计算就是针对这个方向的。

图 7-18 还显示了两个角度(θ和α)的作用。θ表示 spotDirection 与 lightDirection 之间的夹角。α表示点光源的截面角度。当θ大于α时，执行光照计算的目标点位于光锥之外，没有被聚光源照射到。

另一方面，如果点位于光锥之内，则在 spotDirection 方向光强最大，然后随着θ值的增大逐渐衰减。常用的方法(这里也是使用这种方法)是用$(\cos\theta)^{spotExponent}$表示它的衰减过程。

当 spotExponent 值很大时，随着 spotDirection 和 lightDirection 之间夹角 θ 的增大，光强衰减的速度就比较快。可能会发现镜面反射公式曾使用类似方式建模。在光锥中光强衰减效果用 spotEffect 表示，则有下面的公式：

$$\text{spotEffect} = (\cos\theta)^{\text{spotExponent}} = (\text{spotDirection} \cdot \text{lightDirection})^{\text{spotExponent}}$$

如果在顶点着色器中实现一个聚光源并且应用 Phong 着色模式(逐片段着色)，则它与点光源没有任何区别。变化是在片段着色器中，但是这些变化很微小。程序清单 7-7 给出了实现一个聚光源的片段着色器的代码。图 7-19 所示为聚光源照射到应用了纹理的地板时所使用的着色器的结果。

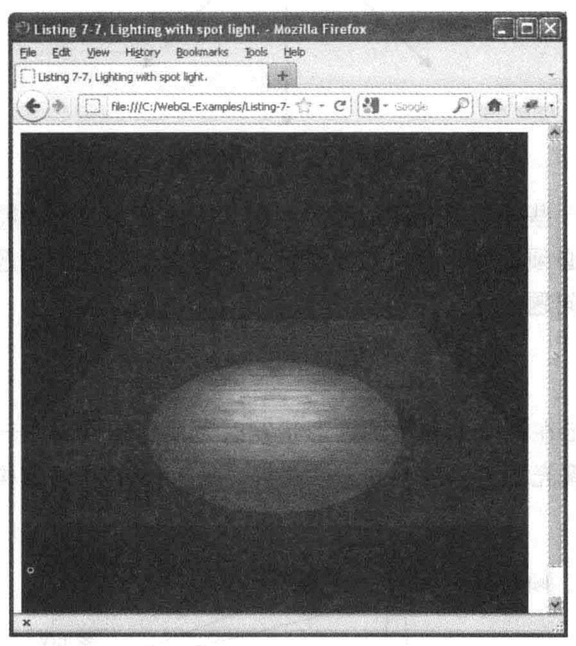


图 7-19 聚光源在纹理地板上的效果

可从 Wrox.com 下载源代码

程序清单 7-7 实现了一个聚光源的片段着色器

```
<script id="shader-fs" type="x-shader/x-fragment">
  // Fragment shader implemented to perform lighting according to the
  // Phong reflection model with a spot light. The interpolation method
  // that is used is Phong shading (per-fragment shading) and therefore the
  // actual lighting calculations are implemented here in the fragment shader.
  precision mediump float;

  varying vec2 vTextureCoordinates;
  varying vec3 vNormalEye;
  varying vec3 vPositionEye3;

  uniform vec3 uAmbientLightColor;
  uniform vec3 uDiffuseLightColor;
```

```
uniform vec3 uSpecularLightColor;
uniform vec3 uLightPosition;
uniform vec3 uSpotDirection;
uniform sampler2D uSampler;

const float shininess = 32.0;
const float spotExponent = 40.0;
const float spotCosCutoff = 0.97; // corresponds to 14 degrees

vec3 lightWeighting = vec3(0.0, 0.0, 0.0);

void main() {

  // Calculate the vector (l) to the light source
  vec3 vectorToLightSource = normalize(uLightPosition - vPositionEye3);

  // Calculate n dot l for diffuse lighting
  float diffuseLightWeighting = max(dot(vNormalEye,
                                    vectorToLightSource), 0.0);

  // We only do spot and specular light calculations if we
  // have diffuse light term.
  if (diffuseLightWeighting > 0.0) {
    float spotEffect = dot(normalize(uSpotDirection),
                           normalize(-vectorToLightSource));

    // Check that we are inside the spot light cone
    if (spotEffect > spotCosCutoff) {
      spotEffect = pow(spotEffect, spotExponent);

      // Calculate the reflection vector (r) needed for specular light
      vec3 reflectionVector = normalize(reflect(-vectorToLightSource,
                                                 vNormalEye));

      // Camera in eye space is in origin pointing along the negative z-axis.
      // Calculate viewVector (v) in eye coordinates as
      // (0.0, 0.0, 0.0) - vPositionEye3
      vec3 viewVectorEye = -normalize(vPositionEye3);

      float rdotv = max(dot(reflectionVector, viewVectorEye), 0.0);

      float specularLightWeighting = pow(rdotv, shininess);

      lightWeighting =
        spotEffect * uDiffuseLightColor * diffuseLightWeighting +
        spotEffect * uSpecularLightColor * specularLightWeighting;
    }
  }

  // Always add the ambient light
```

```
    lightWeighting += uAmbientLightColor;

    vec4 texelColor = texture2D(uSampler, vTextureCoordinates);
    // modulate texel color with lightweigthing and write as final color
    gl_FragColor = vec4(lightWeighting.rgb * texelColor.rgb, texelColor.a);
  }
</script>
```

7.8 光强衰减

光的强度通常随着光在介质(如空气)中的传播距离衰减。在真实世界中，光强衰减与 $1/r^2$ 成正比，这里的 r 是离光源的距离。用来描述这种衰减过程的函数有时也称为距离衰减函数(distance falloff function)。

在计算机图形学中，常用的距离衰减函数不同于 $1/r^2$。一个原因是，用 $1/r^2$ 函数模拟计算机图形学中的光照会造成亮度的变化太大。下面的衰减函数经常用在台式 OpenGL、DirectX 等固定功能的流水线中，也用在 WebGL 中。

$$\frac{1}{consantAtt + linearAtt \times r + quadraticAtt \times r^2}$$

这个公式中各个参数的意义如下所示。

- *constantAtt*——衰减常量。
- *linearAtt*——线性衰减。
- *quadratticAtt*——二次方衰减。
- *r*——距光源的距离。

图 7-20 反映这个衰减函数的曲线。x 轴代表 r，即距光源的距离，y 轴是衰减量。下方的曲线(即 fdist1)是所有常量为 1 的衰减函数。

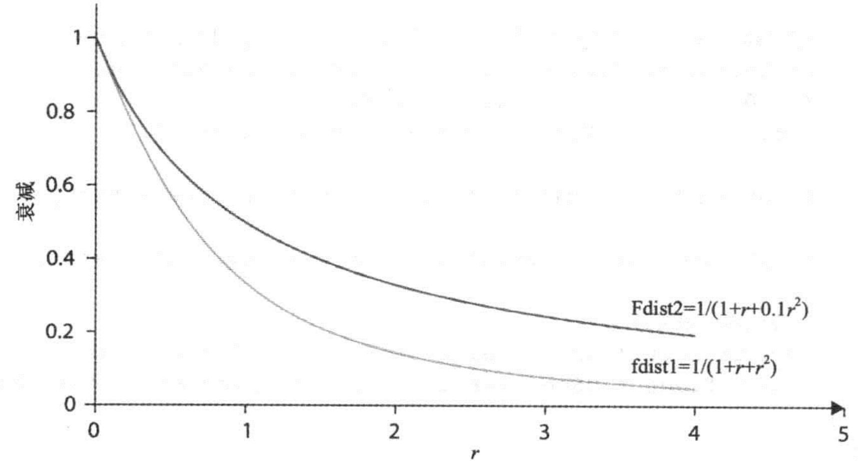


图 7-20　带有两组常量的衰减函数示例。x 轴表示距光源的距离，y 轴表示衰减量

这意味着可以得到以下衰减函数。

$$\frac{1}{1+r+r^2}$$

上方的曲线(名为 fdist2)所示为二次方衰减常量为 0.1、其他两个常量为 1 时的衰减曲线。这意味着可以得到以下衰减函数。

$$\frac{1}{1+r+0.1r^2}$$

可以看出，二次方常量从 1.0 减少到 0.1 时，相当于向上提升了曲线，因此在这种情形下，距离增大时，衰减比较缓慢。

基于该衰减函数在着色器中实现衰减不需要多少新代码。程序清单 7-8 所示为一个完整的着色器，它实现了有衰减效果的点光源。程序中计算衰减的代码高亮显示。

可从 Wrox.com 下载源代码

程序清单 7-8 带衰减效果的点光源片段着色器

```
<script id="shader-fs" type="x-shader/x-fragment">
  // Fragment shader implemented to perform lighting according to the
  // Phong reflection model with a point light and attenuation.
  precision mediump float;

  varying vec2 vTextureCoordinates;
  varying vec3 vNormalEye;
  varying vec3 vPositionEye3;
  varying vec3 vLightPositionEye3;

  uniform vec3 uAmbientLightColor;
  uniform vec3 uDiffuseLightColor;
  uniform vec3 uSpecularLightColor;
  uniform vec3 uLightPosition;
  uniform sampler2D uSampler;

  const float shininess = 32.0;
  // Attenuation constants
  const fl oat constantAtt = 1.0;
  const fl oat linearAtt = 0.1;
  const fl oat quadraticAtt = 0.05;

  vec3 lightWeighting = vec3(0.0, 0.0, 0.0);

  void main() {
    float att = 0.0; // Attenuation

    // Calculate the vector (l) to the light source
    vec3 vectorToLightSource = normalize(vLightPositionEye3 - vPositionEye3);
```

```
        // Calculate n dot l for diffuse lighting
        float diffuseLightWeighting = max(dot(vNormalEye,
                                         vectorToLightSource), 0.0);
        // Only do spot calculations if diffuse term is available
        if (diffuseLightWeighting > 0.0) {
          // Calculate attenuation
          fl oat distance = length(vec3(vLightPositionEye3 - vPositionEye3));

          att = 1.0/(constantAtt+linearAtt * distance +
                     quadraticAtt * distance * distance);

        // Calculate the reflection vector (r) needed for specular light
        vec3 reflectionVector = normalize(reflect(-vectorToLightSource,
                                                  vNormalEye));

        // Camera in eye space is in origin pointing along the negative z-axis.
        // Calculate viewVector (v) in eye coordinates as
        // (0.0, 0.0, 0.0) - vPositionEye3
        vec3 viewVectorEye = -normalize(vPositionEye3);

        float rdotv = max(dot(reflectionVector, viewVectorEye), 0.0);

        float specularLightWeighting = pow(rdotv, shininess);

        lightWeighting =
          att * uDiffuseLightColor * diffuseLightWeighting +
          att * uSpecularLightColor * specularLightWeighting;
        }

        // Always add the ambient light
        lightWeighting += uAmbientLightColor;

        vec4 texelColor = texture2D(uSampler, vTextureCoordinates);
        // modulate texel color with lightweigthing and write as final color
        gl_FragColor = vec4(lightWeighting.rgb * texelColor.rgb, texelColor.a);
      }
    </script>
```

虽然程序清单 7-8 中只有几行新代码，但考虑了如何计算光强衰减效果，这很重要。首先，由于光强衰减不是与距离成线性关系，因此先在顶点着色器中计算衰减光强，然后在片段着色器中使用线性插值得到的衰减光强是不正确的。通常，在顶点着色器中计算距离，并把该值发送到片段着色器，然后在片段着色器中计算衰减光强。

但是如果有这样一个比较“粗糙”的模型，其顶点之间相隔很远(如本书前面几次出现的地板和桌面)，这种方法可能会得到错误的结果。其原因是，计算顶点的距离，然后根据这些进行插值得到片段的距离。对于“粗糙”模型，特别当光源离模型很近时，顶点的距离远远大于相应三角形最近片段的距离。

这很容易用图 7-21 来说明。在图中，用两个三角形表示一个地板，三角形的顶点位置

如下所示。

```
var floorVertexPosition = [
    // Plane in y=0
     5.0, 0.0,  5.0,  //v0
     5.0, 0.0, -5.0,  //v1
    -5.0, 0.0, -5.0,  //v2
    -5.0, 0.0,  5.0]; //v3
```

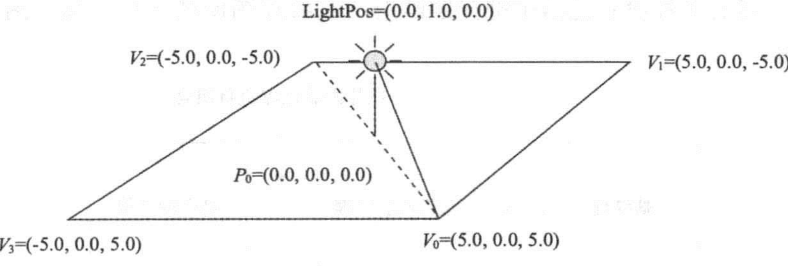


图 7-21 在顶点着色器中计算衰减需要的距离时，“粗糙”的模型如何表明问题的示例

假设有一个点位于坐标(0.0,1.0,0.0)，它离地板中心(P_0，坐标为 0.0,0.0,0.0)的距离为 1。则光源到顶点 V_0 的距离为

$$\|(5.0,0.0,5.0)-(0.0,1.0,0.0)\|\sqrt{5.0^2+(-1.0)^2+5.0^2}=\sqrt{51}=7.1$$

在着色器中，不需要像这样手工计算距离。OpenGL ES 着色语言内置了 length()函数，可用于计算矢量的长度。

由于全部 4 个顶点距光源的距离相同，这意味，对距离进行线性插值时，则在 P_0 片段也会得到相同的距离，这显然是错误的，因为 P_0 到光源的距离为 1.0，而根据 length()的计算，这个距离为 7.1。由于这个距离太大，因此在 P_0 附近的片段看起来比较暗。

这正是把距离计算放在程序清单 7-8 的片段着色器中的一个原因。另一个方法是使用“较细”的模型，并在顶点着色器中进行距离计算。

注意，通常在平行光中不考虑衰减。因为平行光的光源无限远，根据距光源的距离计算光强衰减毫无意义。

7.9 光照映射

至此，应该知道如何用 Phong 反射模型、Gouraud 着色或 Phong 着色模拟真实世界中的光照。此外，掌握光照映射(light map)也很重要。它与模拟光照稍微不同，在 3D 图形游戏中尤为常见的。最先使用该技术的 3D 图形游戏是 Quake(雷神之锤)。

光照映射的思想是：如果场景中有静止的对象，可以创建包含预先计算好的光照信息

的纹理，来模拟这些对象上的光照。这样，可以用全局光照模型(如辐射着色)计算得到高质量的光照效果，但是现在不需要大量的实时计算。光照计算可以用 Blender 或 Maya 等 3D 建模型软件事先离线完成，然后把结果保存在纹理中，并应用在自己的 WebGL 应用程序中。

在理论上，可用同一个纹理保存基纹理图像(如砖墙)和光照信息。但是，更好的办法是使用另外一个名为光照映射(light map)的单独纹理(光照映射这个词通常表示只包含光照信息的纹理)。然后在片段着色器中把光照映射与基纹理图像结合在一起。图 7-22 说明了它的基本思想。

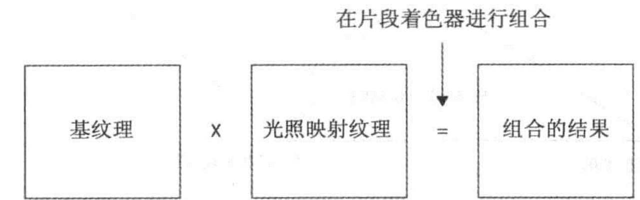


图 7-22　应用光照映射的一个例子，在基图像纹理中结合了光照映射信息

把光照映射与基纹理分离开，光照效果可以重用于几个不同的基纹理上。如果愿意可更容易修改基纹理上的光照信息，例如，可以用光照映射模拟手电筒关闭时。

此外，为了节省内存，砖墙等基纹理经常使用拼贴(tiled)模式。通常按拼贴模式计算光照是没有意义的。

由于已经掌握了如何在 WebGL 中应用一般的纹理，则光照映射一旦创建后就不难使用。为了了解光照映射的用法，下面这段代码演示了包含两个不同采样程序的片段着色器。其中一个采样程序 uSamplerBase 对基映射进行采样，uSampleLight 采样程序则对光照映射进行采样。

```
precision mediump float;
varying vec2 vTextureCoordinates;
uniform sampler2D uSamplerBase;
uniform sampler2D uSamplerLight;

void main() {
  vec4 baseColor = texture2D(uSamplerBase, vTexCoordinates);
  vec4 lightValue = texture2D(uSamplerLight, vLightCoordinates);
  gl_FragColor = baseColor * lightvalue;
}
```

为了使该片段着色器能运行，还需要增加以下的 JavaScript 代码。

```
// Get the location of the uniform uSamplerBase for the base map
pwgl.uniformSamplerBaseLoc = gl.getUniformLocation(shaderProgram,
                                                    "uSamplerBase");

// Get the location of the uniform uSamplerLight for the light map
pwgl.uniformSamplerLightLoc = gl.getUniformLocation(shaderProgram,
```

```
                                        "uSamplerLight");
...

// Bind the base texture to texture unit gl.TEXTURE0
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, textureBase);

// Bind the light texture to texture unit gl.TEXTURE1
gl.activeTexture(gl.TEXTURE1);
gl.bindTexture(gl.TEXTURE_2D, textureLight);

...

// Set base map sampler to the value 0
// so it matches the texture unit gl.TEXTURE0
gl.uniform1i(pwgl.uniformSamplerBaseLoc, 0);

// Set light map sampler to the value 1
// so it matches the texture unit gl.TEXTURE1
gl.uniform1i(pwgl.uniformSamplerLightLoc, 1);
```

7.10 小结

本章介绍了许多与 WebGL 中光照有关的内容。通过本章的学习，你应该掌握了全局光照模型与局部光照模型的区别，掌握了一个非常流行的局部光照模型—— Phong 反射模型。此外还知道了 Phong 反射模型中 3 个不同的反射分量：

- 环境反射
- 漫反射
- 镜面反射

此外，还熟悉了在 3D 图形学经常用于着色的 3 种基本的插值技术：

- 平面着色
- Gouraud 着色(逐顶点着色)
- Phong 着色(逐片段着色)

以及以下 3 种不同类型的光：

- 平行光
- 点光源
- 聚光源

现在知道了这些光的差别，也能够用 OpenGL ES 着色语言编写代码基于这些光源对光照建模。此外，还知道，衰减会使点光源或聚光源具有更真实的效果。最后，还介绍了如何用光照映射模拟 WebGL 中的灯光。

第 8 章

WebGL 性能优化

本章主要内容：

- 实际支持 WebGL 的硬件
- 支持 WebGL 的重要软件组成
- 如何确定 WebGL 图形流水线存在的性能瓶颈
- 如何优化不同的 WebGL 瓶颈
- WebGL 的常见性能知识
- WebGL 的融合机制以及如何用它实现透明效果

在某些情形下，可能需要开发一个只显示相对静止图形的 WebGL 应用程序，因此性能也许不是太重要。然而，大多数情况下，性能非常重要。在前面几章中，已经讨论了应该从性能角度考虑的几个因素。但是由于性能非常重要，因此本章大部分内容都与 WebGL 性能优化有关。

然而，在开始讨论与性能有关的问题之前，先简单地介绍 WebGL 底层的工作机制——硬件组成和软件组成。熟悉这些组成有助于更好地理解和分析与性能有关的问题。

本章的结尾，还要介绍融合技术，以及如何利用这种技术实现透明效果。最后，本章提供一些可以有助于在读完本书之后继续学习 WebGL 的资源。

8.1 WebGL 底层工作机制

本节介绍几个支持 WebGL 设备的真实硬件和软件。由于智能手机和平板电脑等移动设备的使用越来越普遍，许多用户希望使用移动设备运行自己开发的 WebGL 应用程序。因此(也因为作者在移动行业积累了大量的经验)，本节介绍的例子都来自移动行业。但应该注意到，当前移动设备的硬件和软件组成在许多方面与 PC 行业的组件相似，因此这里

介绍的内容具有普遍性。

8.1.1 支持 WebGL 的硬件

当前，智能手机和平板电脑采用两种不同的技术解决方案。

- 把应用程序处理器和蜂窝调制解调器(cellular modem)集成到同一个芯片上。这种解决方案的优点是降低成本和电源功耗。
- 将调制解调器和应用程序处理器分开来，在特定产品中将这两者相结合可以给设备制造商带来更大的灵活性。

ST-爱立信有 3 个产品系列可供设备生产商用于生产智能手机和平板电脑。

- NovaThor 是一个集成的、完备移动平台，它把程序处理器和蜂窝调制解调器集成到同一个芯片上。
- Nova 系列具有独立的应用程序处理器，不包含蜂窝调制解调器。
- Thor 蜂窝调制解调器为集成设备提供了移动宽带访问。

下面介绍 NovaThor U8500 和 NovaThor L9540。

1. NovaThor U8500

NovaThor U8500 采用双核 ARM Cortex-A9 处理器，最高频率为 1GHz。图 8-1 是此芯片用于移动平台的硬件模块图。这里未介绍图中的每个模块，重点介绍从 WebGL 角度看来最重要的硬件模块。

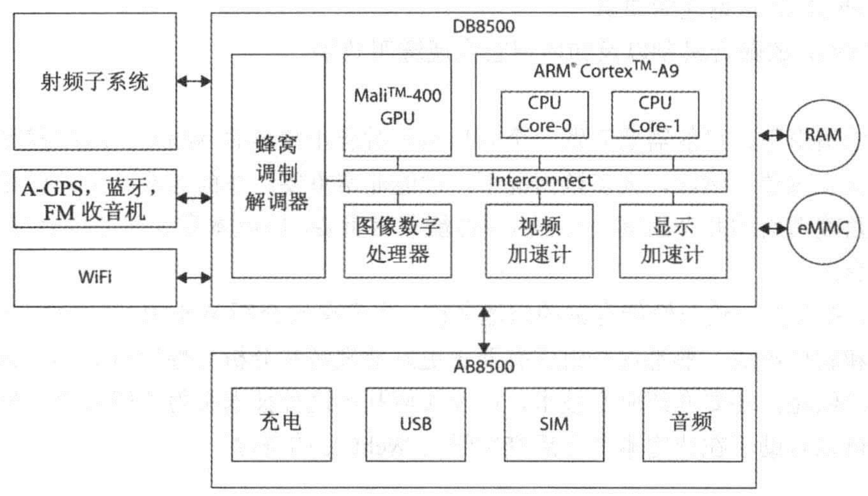


图 8-1 NovaThor U8500 简化的模块图

双核 ARM Cortex-A9 处理器是大多数软件运行的 CPU。例如，Linux 内核和安卓都是在 Cortex-A9 处理器上执行的。这也是 Webkit 和 V8 JavaScript 等引擎的 Web 浏览器和它的支持库运行的地方。因为 NovaThor U8500 使用双核 CPU，所以不同的线程和进程可以在两个不同的核上同时运行。

Interconnect 模块是一个高性能的 on-chip 总线。它把 Cortex-A9 处理器连接到其他 on-chip

模块，如 ARM 公司的 Mali-400 GPU、图像数字处理器、视频加速计、显示加速计和蜂窝调制解调器等。

从 WebGL 角度来看，Mali-400 GPU 显然是一个十分重要的硬件。GPU 是以基于小块的延迟绘制技术(Tile-Based Deferred Rendering，TBDR)为基础的。在移动设备的 GPU 中，该技术尤为常见，且它基于把帧缓存分成小块这一思路。对于 Mali-400 来说，小块的尺寸是 16×16 像素。由于这样的小块(tile)不需要太大的内存，因此 GPU 当前正在处理的小块可以保存在速度非常快的 on-chip 内存中。这正是这种体系结构的主要优点。当小块被光栅化，并且小块的全部片段都处理完后，它的内容(颜色和深度)会被写入位于外部内存上的完整帧缓存中。

虽然有关 WebGL 性能的讨论通常集中在 GPU 上，但是 CPU 或者 GPU 与 CPU 之间的总线以及网络连接也会影响用户体验到的总体性能。如果网络连接速度很快，则 WebGL 应用程序就很快加载到浏览器中，这也是很重要的。需要为自己的程序下载很大的 3D 模型时，这尤为重要。

现在有两种方法把 WebGL 应用程序下载到基于 NovaThor U8500 的设备上。使用 WLAN 或者结合使用蜂窝调制解调器和射频(RF)子系统。若为后者，蜂窝调制解调器用数字信息调制载波信号，然后用 RF 子系统传送。

2. NovaThor L9540

NovaThor L9540 由应用程序处理器和移动宽带调制解调器两部分组成。前者基于双核 ARM Cortex A9 处理器，后者是 ST-爱立信生产的 Thor M7400。应用程序处理器包含一个由 Imagination Technologies 公司生产的 PowerVR 系列 5 GPU，它比 NovaThor U8500 中使用的 Mali-400 GPU 功能要强大许多。此外，与 NovaThor U8500 相比，用于 NovaThor L9540 的 Cortex-A9 处理器的主频和高速缓存都增加了不少。

虽然 WebGL 在 NovaThor U8500 平台上能很好地工作，但是在 NovaThor L9540 中的高效宽带调制解调器 M7400、新型的 PowerVR 系列 5 GPU 和改进的 Cortex-A9 处理器使得它成为比 NovaThor U8500 更加强大的运行 WebGL 的平台。

> 如果你是这样一类用户：喜欢使用新型的嵌入设计的原型，并且想控制自己设备上的全部软件，则不妨看看 IGLOO 开源社区(www.igloocommunity.org)。这个开源社区以 Snowball 为主题。Snowball 是一个低成本的单主板计算机。Snowball 又是建立在 ST-爱立信 Nova A9500 处理器之上。该处理器是 NovaThor L9540 的祖先之一。

8.1.2 关键的软件组成

在智能手机、平板电脑和支持 WebGL 的 PC 机中，有大量不同的软件。图 8-2 简单说

明了从 WebGL 角度看一些最重要的软件组成。

浏览器
WebKit
V8 JavaScript 引擎
用户空间
OpenGL ES 库
HTTP 栈
libc
Linux 内核
GPU 驱动程序
网络设备驱动程序
内核空间
硬件

图 8-2　支持 WebGL 的设备的关键软件

最上层是 Web 浏览器。Web 浏览器通常控制显示 WebGL 图形和其他 Web 内容的窗口。Web 浏览器还负责地址栏、收藏夹、访问历史及浏览器中可用的其他菜单。

当前，所有重要的 Web 浏览器都是建立在某个类型的浏览器引擎之上的。引擎执行繁重而复杂的任务：分析 HTML、构建 DOM 树、生成 Web 页面布局等。例如，Google Chrome 和苹果的 Safari 都使用开源浏览器引擎 WebKit，而 Firefox 使用 Gecko 引擎(也是开源程序)。

V8 JavaScript 是一个现代的、高效的 JavaScript 引擎，用来执行 JavaScript。这个引擎主要是由 Google 公司的开发人员开发的，但由于它是开源代码，因此其他公司和个人对它也有贡献。V8 分析 JavaScript 源代码并构建抽象语法树(Abstract Syntax Tree，AST)。在 AST 的辅助下，它可以把 JavaScript 代码编译成适用于特定目标体系结构(如 IA-32、x64、ARM 或 MIPS)的二进制可执行代码。然后 CPU 执行二进制代码。

GPU 需要的软件驱动程序通常分为两类：用户空间库(User Space library，图 8-2 中的 OpenGL ES 库)和内核空间驱动程序(Kernel Space driver)。在 Linux 环境中，这样分类的一个理由是 Linux 内核(包括内核驱动程序)是在 GPLv2 下发布的，它要求所有驱动程序的源代码要公开。

然而，大多数 GPU 厂商并不想公开自己的驱动程序的全部源代码。他们把大部分的功能放入用户空间库中，从而保密自己的源代码，用户空间库是专属的，不需要公开发布。内核空间驱动程序获得 GPLv2 许可，但是它只是一个非常小的模块，用户空间库利用它与硬件通信，因此它并没公开很多真正有用的功能。

用户空间库有时也称为用户模式驱动程序(User Mode Driver，UMD)，在Linux环境中与共享库(.so文件)相对应，在微软Windows环境中与动态链接库(.dll文件)相对应。

图8-2也包含了HTTP栈，WebKit利用它向Web服务器请求Web内容。HTTP栈用套接字(Socket)导出Linux 内核中TCP/IP栈的功能。Linux内核中的TCP/IP栈又利用网络驱动程序把IP数据包发送给蜂窝调制解调器或WLAN。

libc代表标准的库功能。它是浏览器、WebKit、JavaScript引擎运行的基础。

索尼爱立信(现在是索尼移动通信)是最早发布在安卓浏览器中支持的WebGL的供货商。它把源代码发布为开源代码，如果对这方面感兴趣或者想亲自体验它，可在https://github.com/sonyericssondev/WebGL网页找到相关内容。

WebGL应用示例

为了帮助理解不同组成部分之间的相互作用，这里介绍一个场景，描述下载WebGL应用程序并在Web浏览器中显示时的交互。以下步骤都已经相当简化。

(1) 用户在Web浏览器的地址栏中输入含有WebGL内容的Web页面的地址。浏览器把这个URL地址发送给WebKit，WebKit用HTTP栈建立一个HTTP请求消息，请求Web页面。

(2) HTTP请求消息通过套接字发送给TCP/IP栈，TCP/IP栈利用传输控制协议(TCP)和IP协议封装这个HTTP请求消息。IP数据包由网络设备驱动程序存放在缓冲区中并发送给蜂窝调制解调器或WLAN。

(3) 当HTTP请求消息到达Web服务器时，Web服务器会用HTTP响应消息做出响应，并把请求的内容发回给用户的设备。为了简化讨论，假设全部内容(包含HTML、JavaScript和着色器源代码)都包含在一个文件中。

(4) 请求的内容到达用户的设备，并经过网络驱动程序、TCP/IP栈、HTTP栈到达WebKit。经过WebKit分析后，创建一个内部DOM树。WebKit还创建了绘制树(render tree)，绘制树是一个对不同元素如何绘制的内部描述。然后把JavaScript代码发送给V8 JavaScript引擎，后者分析JavaScript代码并把它编译成二进制可执行文件，再由CPU执行。

(5) 当JavaScript代码包含对WebGL API的调用时，这些调用语句返回到WebKit，WebKit会调用OpenGL ES 2.0 API。顶点着色器和片段着色器的源代码(用OpenGL ES着色语言2.0库编写的)经过OpenGL ES 2.0库编译后生成二进制代码，然后会将库代码经过内核模型驱动程序上传给GPU。

(6) 最后，创建了纹理、顶点缓冲区和uniform变量并把它们发送给GPU后，绘制过

程开始。

8.2 WebGL 性能优化

本书大部分例子相对比较简单，以方便阐述 WebGL 的某些特定功能。这意味着，从性能角度来看，应用程序采用什么样的代码结构并不重要。大多数 WebGL 实现运行这些示例都不会有任何问题。但是，在实际工作中开始编写包含大量对象和大量复杂着色器的大型应用程序时，最终还是要考虑性能问题。本节将介绍如何处理这些情况。

8.2.1 避免初学者的典型错误

要衡量一个只在 CPU 上运行的应用程序各部分的性能时，基本策略是添加代码启动一个时钟，执行一些任务，然后停止时钟。这样就可以知道应用程序执行这些任务要花多长时间。

如果把这个想法应用于 WebGL 应用程序，则很自然会写出如下的伪代码。

```
// This code does NOT measure how long
// it takes to do the actual drawing.

startClock();

gl.drawElements(gl.TRIANGLE_STRIP, ...);

stopClock();
```

但是这段代码并不像 WebGL 初学者所期望的那样工作。调用 gl.drawElements()函数不会衡量实际绘制三角带所用的时间。

这是因为，图形的流水线以异步方式工作。调用 gl.drawElements()或者其他任何 WebGL 方法时，低层的函数通常把命令放置在一个命令缓冲区中后就会返回。之后这些命令由 GPU 来执行。这意味着，前面这段代码只能衡量到把命令放置到命令缓冲区中需要多长时间，这不能真正代表程序的绘制性能。

gl.flush()和 gl.finish()方法

WebGL 提供两个方法，可以用前面介绍的方法衡量绘制时间。

- gl.flush()
- gl.finish()

调用 gl.flush()方法时，它指明所有先前发送到 WebGL 的命令都必须在有限的时间内完成运行。显然，这个方法没多大帮助，尽管该命令看起来很吸引人。

gl.finish()方法强制指定所有前面的 Web 命令都必须完成，在帧缓冲区全部更新之前，该方法不会返回。虽然，gl.finish 方法看起来可以解决这个问题，但是实际上，许多 GPU

厂商为了获得额外的性能，并没有使 gl.Finish()(它是 gl.finish()依赖的 OpenGL ES 函数)真正实现同步。

这意味着，无法完全保证以下这段代码能够衡量到绘制一个三角形带所需要的时间。

```
// This code does NOT measure how long
// it takes to do the actual drawing.

startClock();

gl.drawElements(gl.TRIANGLE_STRIP, ...);

gl.finish();

stopClock();
```

8.2.2　确定瓶颈位置

第 1 章曾提到，WebGL 的绘制很大程度上采用流水线形式。这意味着，对于指定时间的特定设备，WebGL 应用程序在流水线的某个阶段出现性能瓶颈。当一个阶段成为性能瓶颈时，就意味着，它是流水线中执行速度最慢的部分。瓶颈阶段成为整个程序绘制性能的限制。

为了改善整个绘制性能，需要优化瓶颈阶段。然而必须指出，在一个帧期间，瓶颈可能会在流水线的不同阶段之间移动。因此，确定流水线的某个特定阶段是瓶颈时，只意味着这个阶段在帧的大部分时间中是瓶颈。

Web 应用程序的一个主要优点是它可以应用在许多不同的设备、操作系统和浏览器上。但是需要确定 WebGL 应用程序的瓶颈位置时，这也是真正的挑战。对于不同的设备，瓶颈可能出现在不同阶段。

虽然这听起来很显而易见，但是重要的是能记住。为了找出应用程序针对某个设备存在的瓶颈位置，需要在设备上进行测试。然而，在所有支持 WebGL 的设备上测试几乎是不可能的。因此，必须有选择地测试某些设备，然后希望应用程序在其他设备上也能正常运行。一般说来，可以假定，如果一个应用程序在低端移动设备能正常运行，则也能在高端设备上正常运行。

WebGL 应用程序的性能瓶颈分为 3 大类(如图 8-3 所示)。

- CPU 受限(CPU-limited)
- 顶点受限(Vertex-limited)
- 像素受限(Pixel-limited)

图 8-3 所示的流水线是真实 WebGL 流水线和第 1 章介绍的理论 WebGL 流水线的简化

表示。图中包含了一些方块(模块)，以便于理解性能瓶颈的位置。此外，还有很多不同的方法可以对 WebGL 图形流水线中的瓶颈进行分类。例如，根据图 8-3 和前面介绍的宽泛分类，与纹理有关的瓶颈被归为像素受限。也可以将其归为纹理受限。

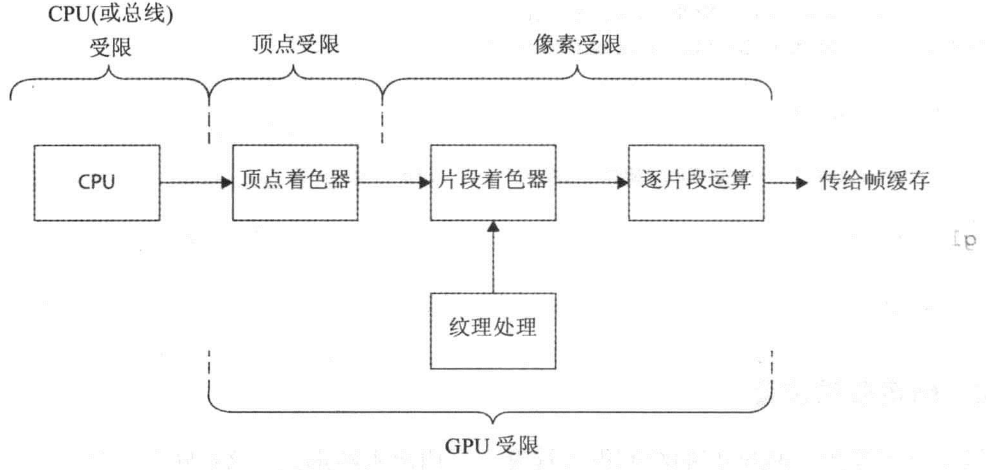


图 8-3　显示性能瓶颈的 WebGL 流水线

如果想改善 WebGL 应用程序的性能，第一步是确定当前瓶颈所在的位置。为此，需要执行一系列的测试，在某个特定的阶段改变工作负荷。

- 增加某个阶段必须执行的工作。如果性能下降，就找到了当前瓶颈。
- 在流水线的某个特定阶段减少工作。如果减少特定阶段必须完成的工作，且性能改善，则找到瓶颈的位置。

另一有助于确定瓶颈位置的相关技术，就是改变 CPU 或 GPU 的频率。如果用户所处的环境容易改变频率，则可以迅速发现 WebGL 应用程序是否是 CPU 受限的。

下面将介绍几个用来确定瓶颈位置的测试方法。不必按特定的顺序执行这些测试。每个测试都会提供有关瓶颈位置的信息。如果在某个设备上一些测试比另一些测试容易完成，则先完成容易的测试是合理的。在同一个阶段执行不同的测试会对有关瓶颈结论的正确性更有信心。

最新的 CPU 有时也支持统一着色器体系结构(Unified Shader Architecture)。这意味着所有类型的着色器都用相同的硬件系统执行指令，而且驱动程序在运行时实现负载平衡，并且决定应该用 GPU 的哪个硬件模块来执行顶点着色器或片段着色器。

对于使用统一着色器体系结构的 GPU 来说，很难用本章的测试方法确定 GPU 中的瓶颈位置。在这种情况下，只能说程序是 CPU 受限或 GPU 受限。如果想进一步了解这方面的信息，大多数 GPU 厂商也提供不同的测试工具。可以用这些工具执行详细的性能分析。

1. CPU受限WebGL应用程序的测试

CPU受限是指WebGL应用程序受CPU速度的限制。然而，这个术语有时有更广泛的意义，即WebGL受到除GPU之外其他硬件的限制。在这种情况下，应用程序受限实际上是受总线的带宽、CPU高速缓存的大小或可用内存大小的限制。

在WebGL应用程序中通常需要由CPU完成的任务取决于用JavaScript编写的代码，它包含以下处理。

- 用户输入的处理
- 物理模拟
- 对象变换所需要的矩阵相乘运算
- 碰撞检测
- 人工智能

在开始考虑自己的WebGL应用程序是否属于CPU受限之前，首先确保绘制循环有没有包含任何会降低绘制速度的代码。如果JavaScript代码在绘制循环中访问文件系统API、本地存储、索引的数据，则会严重影响系统的性能。另外，也不要在绘制循环中包含网络访问和着色器源代码的编译。

如果自己编写整个WebGL应用程序，则可以控制何时访问哪个API。但是，如果重用其他人编写的代码或库，则最好记住这一点。对于复杂的代码，最好用Chrome开发人员工具或Firebug切实查看应用程序运行过程。

可以用以下的测试确定程序是否是CPU受限的。

- 用所用平台中的程序查看CPU使用率。例如，在Linux系统中使用top程序，在Windows系统中使用任务管理器(Task Manager)，在Mac OS系统中使用活动监视器(Activity Monitor)。如果应用程序的CPU使用率接近100%，则意味着这个应用程序是CPU受限。但是，必须谨慎对待测试得出的结论。也可能存在这样的情况：应用程序中用一个忙等待结构等待CPU资源，例如，在自旋锁情形(spinlock)，测试结果显示为很高的CPU使用率，而实际情形可能是GPU瓶颈。此外，还需要注意，即使这些工具显示很低的CPU使用率，也不一定就说明瓶颈问题出现在GPU。总线的速度、RAM、网络的活动状态以及其他因素也经常成为性能的受限因素。
- 降低CPU的时钟频率，看看性能是否发生变化(这是一个更为直接的测试)。如果CPU时钟下降x，性能也下降x，则可能属于CPU受限情况。然而，如果CPU时钟频率下降x，而性能并没有下降，则很可能不是受CPU频率的限制。
- 如果可能，改变目标测试设备上的GPU频率。这与改变CPU频率相互补充。在PC机中，BIOS有时允许修改GPU的时钟频率，对于某些GPU，还有专门的实用工具软件(如ATITool，CoolBits和PowerStrip)可用来改变GPU的时钟频率。如果降低GPU时钟频率，程序的性能也随着下降，则很可能属于GPU受限情况。

- 如果 JavaScript 代码容易修改，则修改代码，在 CPU 上少执行一些逻辑工作和其他任务，但绘制的场景不变，则可以得到指示，说明应用程序是否属于 CPU 受限。如果 CPU 少处理一些任务，而性能提高了，很可能属于 CPU 受限。

如果怀疑应用程序属于 CPU 受限，就可以使用 Chrome 开发人员工具或 Firebug 工具帮助了解 JavaScript 代码的运行细节。正如前面曾提到，这些工具可以提供有关网络活动状态或文件系统执行情况等有用信息。这些会限制 WebGL 应用程序的性能。

如果想知道在 CPU 一方整个系统的行为，包括底层的软件和硬件，可以使用 OProfile 或 perf 等工具。这些工具允许用户使用原生代码，因此看到在哪个原生函数(C/C++)所用的运行时间最多。所有的现代 CPU 都有性能监视单元(Performance Monitor Unit，PMU)，可以统计和报告以下低级事件。

- 数据和指令高速缓存的未命中率
- TLB 未命中率
- 分支预测未命中率
- 运行周期(elapsed cycle)
- 指令执行数

Perf 属于 Linux 内核的一部分，需要侦听低级事件时尤为有用。

如果用户是应用程序的开发人员，而且只对优化 WebGL 应用程序感兴趣，则这些原生性能分析工具可能不是首选使用的工具。但是，如果希望性能的改善贡献给 WebKit、Chromium、Geeko 或 FireFox 等开源软件，这些工具就非常有用。

2. 顶点受限的 WebGL 应用程序的测试

顶点受限(有时也称为几何受限)是指 WebGL 应用程序的整体速度受限于顶点着色器读取顶点的速度或在顶点着色器中处理顶点的速度。顶点着色器实际执行的操作包括以下两种。

- 顶点变换。
- 逐顶点光照处理。

为了测试应用程序是否属于顶点受限，可以执行以下测试。

- 如果使用带用很多顶点的 3D 模型，则不妨使用顶点较少的模型。
- 减少逐顶点光源的数量，或者为光照使用比较简单的着色计算。
- 看看是否可以去掉顶点着色器中的某些变换或其他操作。

如果对于上述任何一个测试，性能都会提高，则属于顶点受限。当然也可以反其道而行，例如在顶点着色器中增加额外的处理代码，如果性能下降，则顶点着色器是应用程序的瓶颈。为了检查瓶颈是否由顶点数据读取操作引起，可以添加额外的虚拟数据(dummy data)作为顶点数据。如果性能下降，则程序的瓶颈出现在顶点数据读取操作，必须考虑减少顶点数据量或者尝试重新组织顶点数据，以更好地利用预变换的顶点高速缓存和变换后的顶点高速缓存。

3. 像素受限 WebGL 应用程序的测试

像素受限的应用程序受到在顶点着色器之后执行的操作的限制。这些操作基本上是指片段着色器中的光栅化和片段处理，以及纹理处理或逐顶点处理。

下面的方法可以测试应用程序在这个阶段是否遇到瓶颈。

- 减小<canvas>标记的 width 和 height 属性值。例如，对于像<canvas width="1024" height="1024" >这样的代码，不妨把它修改为<canvas width="512" height="512" >，如果性能提高，则可能是像素受限。注意，画布的 width 和 height 属性不同于 CSS 中的 width 和 height 属性，虽然后者也可以作用于画布。本章后面还要进一步介绍这方面的内容。
- 改变片段着色器中每个片段的处理量，例如，去掉光源或者使用比较简单的光照计算。
- 禁用纹理或者使用低分辨率的纹理，看看是否在纹理处理过程遇到瓶颈。此外还可以修改纹理过滤模式。

8.2.3 有关性能的一般性建议

虽然在遇到性能瓶颈时，推荐的解决办法是找出当前瓶颈的位置，但是如果在设计或开发早期就遵循有关性能的一般性准则，通常十分有效。这会降低出现性能问题的几率。本小节介绍几个通用的性能建议，记住这些会十分有用。

除了阅读本章介绍的性能建议，还可以查阅 Google 工程师们提供的一些演讲。

- Gregg Tavares 在 Google I/O 2011 年会上发表“Techniques and Performance”的演讲。
- Ben Vanik 和 Kenneth Russhel 在 New Game Conference 2011 发表有关“Debugging and Optimizing WebGL Applications”的演讲。

在 YouTube 上可以找到这两个视频(或者输入“WebGL Techniques and Performance”和“Debuggin and Optimizing WebGL Application”关键词搜索)，本章的一些建议就是以这两个演讲为基础的。

1. 减少绘制调用的次数

任何对 WebGL API 的调用都会带来开销。每个调用都会要求 CPU 进行额外的处理和数据复制，这会占用时间并要求 CPU 做一些额外工作。通常，如果 GPU 接收到大批可并行处理的数据，运行效率会提高许多。

提高 WebGL 性能的一个最重要建议是在每帧处理期间使调用 gl.drawArrays()和 gl.drawElements()的次数最少。这通常被称为批处理(batching)，例如，某个场景的一部分

需要绘制 1000 个三角形，则调用两次 gl.drawArrays()或 gl.drawElements()每次绘制 500 个三角形，比调用 100 次每次绘制 10 个三角形更好。

第 3 章简短介绍过这个问题。其中介绍了如何用退化三角形把两个不相邻的三角形带连接成一个三角形带，从而只用一次调用 gl.drawArrays()或 gl.drawElements()方法就可绘制它们。如果遇到一个比较复杂的 WebGL 应用程序，代码有许多地方需要调用 gl.drawArrays()或 gl.drawElements()，则在 WebGL Inspector 程序中查看会帮上大忙，因为利用它可以轻易看出每帧中绘制调用的次数。

2. 避免绘制时从 GPU 读回数据或状态

如果不从 GPU 读回数据，则其运行状态会最好。所有 read 和 get 调用，如 gl.readPixels()和 gl.getError()，都会切实降低程序的性能，因为 GPU 中的流水线经常需要刷新。

从性能角度来看，更好的做法是初始化时读取一个值，然后把 JavaScript 代码中需要的值保存在缓冲区中，而不是在绘制场景时通过调用 WebGL API 请求这些值。

3. 从生产代码中删除错误检查代码和调试库

在开发过程中使用错误检查和调试库是十分有必要的，但是在开发阶段之后，应该从生产代码中删除 webgl-debug.js 等库。这是因为在生产代码中保留需要执行的额外的 JavaScript 代码确实不是好想法。此外，这些库通常包含对 gl.getError()的调用，正如已经介绍过的，如果想得到很好的性能，则从 GPU 读取数据不是好主意。

4. 用 WebGL Inspector 找出冗余的调用

第 2 章曾介绍了 WebGL Inspector 有多有用。由于 JavaScript 调用并不是免费的，特别是修改 WebGL 流水线状态的代价十分昂贵，因此应该尽量避免对 WebGL API 的冗余调用。正如第 2 章的内容所讲，WebGL Inspector 的 Trace(跟踪)面板可以高亮显示冗余调用，如图 8-4 所示。

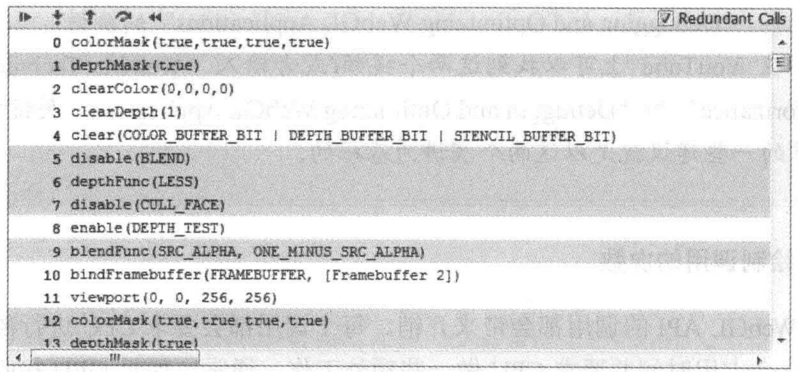


图 8-4　WebGL Inspector 的 Trace 面板可用于找到对 WebGL API 的冗余调用

要记住 WebGL 的一个优点，即其状态是跨帧持续的。例如，如果想在所有时间进行深度测试，只需要在代码的初始设置时调用下面的函数。

```
gl.enable(gl.DEPTH_TEST);
```

不需要在每帧的开头调用 gl.enable(gl.DEPTH_TEST);

5. 整理模型以避免状态改变

通常，应该避免对 GPU 的状态进行没有必要的改变。执行状态改变时，CPU 总要刷新其流水线，这会降低性能。如果场景中绘制的一些对象有大量实例，则最好是对对象进行分组排序，以便可以设置一组对象的共同属性(如融合的状态与深度测试；纹理、属性和 uniform 属性；以及使用的着色器程序)。然后绘制这些对象，接着设置下一组对象的共同属性，再绘制这组对象，依此类推。

绘制场景时，在需要绘制成千上万个对象时，通常要避免绘制每个对象都需要修改 WebGL 的状态。因此与其使用类似于下面的代码。

```
for (var i = 0; i < numberOfModels; i++) {
  var currentModel = listOfModels[i];
  gl.useProgram(currentModel.program);
  currentModel.setupAttributes();
  currentModel.setupUniforms();
  gl.drawElements();
}
```

不如把这些对象进行分类，跟踪最近绘制的模型是否与当前需要绘制的模型相同。如果是，则不需要调用 WebGL API 进行大量的更新处理。这意味着，使用类似下面的代码会更好。

```
var previousModelType = null;
for (var i = 0; i < numberOfModels; i++) {
  var currentModel = listOfModels[i];
  if (currentModel.type != previousModelType) {
    gl.useProgram(currentModel.program);
    currentModel.setupAttributes();
    currentModel.setupUniforms();
    previousModel.type = currentModelType;
  }
  gl.drawElements();
}
```

如果需要对这些对象按优先顺序进行排列，则尽可能把修改或更新成本最大的对象排在前面。例如，通常调用 gl.uniform3fv()来更新 uniform 变量比调用 gl.useProgram()修改整个着色器更便宜。因此，把需要用同一个着色器程序绘制的对象组合在一起，比把需要修改 uniform 变量的对象组合在一起更为重要。

这是 Gregg Tavares 在 Google I/O 2011 演讲中介绍的技术之一。该例和其他例子的源代码可以从 http://webglsamples.googlecode.com/hg/google-io/2011/index.html 下载。

8.2.4 改善 CPU 受限的 WebGL 应用程序性能的建议

如果用户的 WebGL 应用程序属于 CPU 受限类型，则改善 CPU 性能的明显方法是减少 CPU 的工作量。为了减少 CPU 的工作量，可以简化应用程序，以减少物理计算，减少游戏逻辑或 JavaScript 的其他操作。如果用户不想简化 WebGL 应用程序，可以试试下面介绍的方法。

1. 从绘制循环中移走一切可以移走的代码

实际上这在前面已经提到过，但是它非常重要，因此这里再加以说明。应该从绘制循环中移走一切可以移走的代码，在绘制循环之前初始化所有允许的操作。特别是要尽可能避免编译和链接着色器、加载纹理、执行网络访问，或者执行文件系统的访问。如果在绘制循环中还有内循环，则把内循环中的一些共用内容移到外面可以改善程序的性能。例如，对于下面的程序：

```
for (var i=0; i++; i< 10000) {
   setupCommon();
   doSomeSpecificWork();
}
```

改为以下的代码会更好。

```
setupCommon();
for (var i=0; i++; i< 10000) {
   doSomeSpecificWork();
}
```

如你所知，JavaScript 利用垃圾收集功能(Garbage Collection，GC)清除堆中无用的对象。即使 JavaScript 引擎不断改善，执行垃圾收集操作也仍需要时间。如果希望每秒绘制 60 帧(fps)，则完成一帧大约需要 16ms，因此如果希望平滑地运行动画，则在绘制期间没有时间执行垃圾收集操作。

很难控制 JavaScript 引擎何时执行垃圾收集操作，但有一种方法可以减少垃圾收集操作，就是最小化新对象的分配。这在 JavaScript 中并不容易，因为很多东西需要分配对象。然而，如果有一个真正严格的循环，而且你知道需要分配很大的内存块，则最好在进入循环之前分配内存。循环内部分配的内存越小，可执行的垃圾收集会相对比较快。

2. 将 CPU 执行的操作移到 GPU

如果用户的应用程序属于 CPU 受限类型，则可以尝试把某些操作从 CPU 移到 GPU。即使 JavaScript 引擎越来越快，如果在绘制循环中需要为每个帧执行一些矩阵相乘操作，仍然会提高程序的性能。

如果只绘制少数几个对象，这并不很重要，因为 JavaScript 引擎速度已经足够快。但是如果在每帧中需要绘制几千个对象，每个对象需要执行几次矩阵相乘运算，在用户属于

CPU 受限的情况下，则这会很快使帧的速率恶化。

另一方面，注意只将操作移到 GPU 中，并假设性能会提高。通常，场景中的一个对象包含多个顶点，甚至包含更多的片段。因此，如果不谨慎把操作从 CPU 移到 GPU，可能会使 CPU 中每个对象的一次运算可能在 GPU 中变成每个对象的多次运算。在 GPU 执行运算会改善程序的性能，但是这一切都取决于瓶颈出现在哪里。

3. 使用分类数组代替 JavaScript 数组对象

如果用户怀疑自己的应用程序属于 CPU 受限类型，则值得查看下，把 JavaScript 中的数组改为分类数组是否会改善性能。

有关分类数组的更多信息，参阅第 3 章内容。

因为发明分类数组就是为了实现对性能敏感的功能，另外因为分类数组的总大小和元素大小是固定的，从理论上讲，对分类数组的元素的读写访问会更快。因此，与其像下面的代码那样创建一个数组：

```
myArray = new Array(size);
```

不如用下面的代码创建一个分类数组：

```
myArray = new Float32Array(size);
```

然而，使用分类数组带来的潜在速度改善取决于用户如何使用数组，例如是否对这个数组进行大量的读写操作，以及何时创建数组等。

这也是典型的优化方法，它在不同的浏览器或同一个浏览器的不同版本中有不同的表现。如果考虑使用这个优化方法，在有关的论坛和博客中搜索最新的讨论是个好主意，因为 JavaScript 引擎和 Web 浏览器发展很快。

曾在本书许多例子中使用过的 JavaScript 库 glMatrix 可以检查浏览器是否支持分类数组 Float32Array。如果是，则它可以用于所有数组。JavaScript 库 Sylvester(本书第 4 章介绍过它的用法)只用标准的 JavaScript Array 对象实现这个功能。

8.2.5 改善顶点受限的 WebGL 应用程序性能的建议

下面介绍应用程序属于顶点受限类型时，一些可以改善程序性能的优化方法。有些内容第 3 章中已经介绍过，这些内容涉及 WebGL 中不同的绘制方法。如果忘记了某些细节，可以参考第 3 章的内容。

1. 使用三角形带和三角形扇形

如果不用独立三角形(gl.TRIANGLES)而是用三角带(gl.TRIANGLE_STRIP)或三角扇形(gl.TRIANGLE_FAN)，则对于同样数量的三角形，必须定义较少的顶点(详细内容，参阅第 3 章)。

如果用较少的顶点构建模型，则顶点着色器需要读取和处理的顶点也较少。如果程序属于顶点受限型，这会改善程序的性能。但是，还必须记住一点：尽可能减少绘制函数的调用。因此，即使使用了三角形带，也应该尽可能增加退化三角形，使三角形带越长越好。

2. 使用 gl.drawElements()的索引绘制

当模型很大而且有很多共享顶点时，应该使用 gl.drawElements()方法进行索引绘制，而不是使用 gl.drawArrays()方法。索引绘制法可以避免重复同一顶点数据。此外，变换后顶点高速缓存通常只用于索引绘制。

3. 使用交叉顶点数据

如果用一个数组交叉地保存顶点数据，而不是用独立的顶点数组保存不同的属性，通常会得到更好的性能，因为顶点数组具有更好的局部内存。例如，把顶点位置读入到变换前顶点缓存时，很可能会把该顶点的法线信息也读入到变换前的顶点缓存中，在需要时供顶点着色器使用。

4. 用细节层次简化模型

如果属于顶点受限类型，但又不想降低 3D 模型的复杂度，可以使用细节层次(Level Of Detail，LOD)技术。这意味着当对象离开观察者的视线时可以降低模型的复杂度。实际上，通常为模型创建不同的版本，在运行时进行改变。当然，还要记住，在 JavaScript 代码中实现 LOD 逻辑会增加 CPU 的工作量，因此存在这样的风险：如果不谨慎，可能会把瓶颈从 GPU 移到 CPU。

5. 避免在顶点数组中重复使用常量数据

在某些情形，有的模型会包含对于所有顶点而言相同的数据。在理论上，当然可以在顶点数组中重复这个常量数据，使每个顶点都有这个相同值。

更好的办法是为该数据使用顶点常量属性或 uniform 变量。调用 gl.disableVertexAttribArray()方法禁用与顶点着色器中的属性相对应的原型属性索引，就可以使用顶点常量属性。然后，调用下面的方法，给这个顶点常量属性设置值。

```
gl.vertexAttrib4f(index, r, g, b, a);
```

这个方法允许把 r、g、b 和 a 这 4 个值设置为浮点值，此外还有相应的方法设置 3 个、两个或一个浮点值。

在本书的许多例子中，给 uniform 变量设置值。例如，假如 uniform 变量由 4 个浮点数

组成，则用下面的方法给它设置值。

```
gl.uniform4f(location, r, g, b, a);
```

8.2.6　改善像素受限的 WebGL 应用程序性能的建议

如果应用程序属于像素受限类型，则考虑下面介绍的解决方法。注意，本书也把纹理受限包含在这一类中。

1. 扩展画布

如果片段着色器的运算量很大而且用户怀疑它存在一个瓶颈，则可以让 GPU 中的片段着色器处理一个较小的区域，然后让浏览器中的合成器(compositor)把它缩放到较大的区域。这之所以可行，是因为有两种不同的方法指定画布的宽度和高度，而且这两种方法有不同的意义。

首先，可以像本书所有例子那样，指定 canvas 元素的 width 和 height 属性，如下所示。

```
<canvas id="myGLCanvas" width="500" height="500"></canvas>
```

这指定了 GPU 中的绘制区域为 500×500 像素。

此外，还可以将 width 和 height 指定为 style(样式)属性，或者外部 css 文件指定。如果把画布的 width 和 height 属性与作为 style 属性的 width 和 height 结合在一起，则可以使用如下代码：

```
<canvas id="myGLCanvas" width="500" height="500"
  style="width:100%;height:100%"></canvas>
```

在这段代码中，所绘制的画布的大小仍然设置为 500×500 像素，但是显示大小设置为包围盒(containing box)的 100%。假设浏览器窗口为包围盒而且大于 500×500 像素，则画布要放大到这个较大的窗口。图 8-5 简单说明了这一技术。内层矩形代表画布的大小，外层矩形代表包围盒，浏览器中的合成器(compositor)把画布缩放到后者的大小。片段着色器的工作量越大，这个方法越有效。

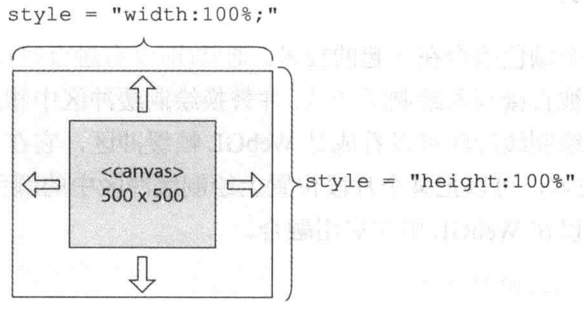


图 8-5　像素受限时，为了得到更好的性能可以扩展画布

2. 将计算移到顶点着色器

如果怀疑片段着色器中存在瓶颈，则可以把某些可以线性化的计算移到顶点着色器中，然后把结果通过 varying 变量传送到片段着色器。例如，在第 7 章中曾提到，同样的光照计算，既可以在顶点着色器中也可以在片段着色器中完成。

通常，在 WebGL 流水线较早阶段执行计算会提高性能。理由是越早执行计算意味着需要执行计算的次数越少。顶点的个数通常少于片段的个数，因此这也意味着，如果在顶点着色器中计算，则需要执行的计算次数就更少。但是，改善性能的前提是应用程序不属于顶点受限。

3. 将 Mip 映射应用于纹理贴图

通常在纹理中使用 Mip 映射是个好主意。当纹理缩小并且可以使用 Mip 映射时，通常会提高高速缓存的利用率，因为读取的纹素在内存中的位置比没有使用 Mip 映射更近。有关 Mip 映射和纹理过滤，可以参考第 5 章内容。

8.3 深入分析融合

第 1 章曾提到片段离开片段着色器之后在 WebGL 流水线中会发生什么。此外，还介绍了这些操作更常用的名字是“逐片段运算”。为了帮助回忆这些内容，现在列出逐片段运算。

- 裁剪测试
- 多采样片段操作
- 模板测试
- 深度缓存测试
- 融合
- 抖动

在以下几小节中，将深入介绍融合技术，它与深度测试一样，也是逐片段运算中最重要也最有用的操作。

8.3.1 融合简介

融合是把两个颜色组合在一起的技术。通常(即没有融合时)，通过 WebGL 图形流水线深度测试的片段被直接写入绘制缓冲区，并替换绘制缓冲区中相应的像素(正如可能还记得第 1 章的内容，绘制缓冲区可以看成是 WebGL 帧缓冲区，它在显示之前与 HTML 页面组合)。利用融合技术，可以把某个片段位置上绘制缓冲区中的颜色与传入的片段颜色组合。用下面的调用可以在 WebGL 中在启用融合。

```
gl.enable(gl.BLEND);
```

传入片段的颜色称为源颜色(source color)，绘制缓冲区中原来片段的颜色称为目标颜色(destination color)。

这两个颜色的组合是按照融合公式进行的，这个公式是高度可定制的。融合公式的通用表示如下所示。

$$color_{final} = factor_{source} \times color_{source} \ op \ factor_{dest} \times color_{dest}$$

这个公式中各个参数的作用如下所示。

- $color_{final}$——对片段执行融合操作后最终得到的颜色。
- $factor_{source}$——与传入片段的颜色相乘的缩放因子。
- $color_{source}$——传入片段的颜色。
- op——数学运算符，用于把传入片段的颜色和目标片段的颜色分别乘上相应的缩放因子后再进行组合。
- $factor_{dest}$——与目标片段的颜色相乘的缩放因子。
- $color_{dest}$——目标片段的颜色。

8.3.2 设置融合函数

通用融合公式中的缩放因子 $factor_{source}$ 和 $factor_{dest}$ 通过调用 WebGL 方法 gl.blendFunc()或 gl.blendFuncSeparate()设置。这两个方法都是 WebGLRenderingContext 的一部分，而且它们的定义如下所示。

```
void blendFunc(GLenum sfactor, GLenum dfactor);


void blendFuncSeparate(GLenum srcRGB, GLenum dstRGB,
                       GLenum srcAlpha, GLenum dstAlpha);
```

这两个方法的参数用来设置融合函数，这些参数的可能值列在表 8-1 的第一列中。该表的细节将在本章后面介绍。

最简单的方法是使用 gl.blendFunc()方法，它只需要两个参数。参数 sfactor 指定 RGB 值和 alpha 值的源融合函数，参数 dfactor 指定 RGB 和 alpha 值的目标融合函数。调用 gl.blendFunc()函数的常见方法如下所示。

```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

回忆一下，8.3.1 小节中通用融合函数的定义如下所示。

$$color_{final} = factor_{source} \times color_{source} \ op \ factor_{dest} \times color_{dest}$$

假设在通用融合公式中默认的运算符为加(+)。调用 gl.blendFunc()将 $factor_{source}$ 设置为 gl.SRC_ALPHA，将 $factor_{dest}$ 设置为 gl.ONE_MINUS_SRC_ALPHA 时，则上述融合公式可以表示为更具体的形式。

$$color_{final} = \alpha_{source} \times color_{source} + (1 - \alpha_{source}) \times color_{dest}$$

这是融合公式常用的形式，它经常在 WebGL 和许多其他类似的计算机图形中，用来实现半透明效果。式中的 α_{source} 代表半透明对象的透明度(实际上它表示不透明度，因为 α=0.0 表示完全透明，α=1.0 表示完全不透明)。为了更好地理解这个公式，可以看一下下面的源片段如何影响最终的颜色值 $color_{final}$。

- 不透明的源片段(α_{source}=1.0)，则结果是 $color_{final}=color_{source}$(源片段的颜色覆盖了目标片段的颜色)。
- 完全透明的源片段(α_{source}=0.0)，则结果是 $color_{final}=color_{dest}$(源片段对最终的颜色没有任何贡献，最终的颜色等于目标片段的颜色)。
- 半透明源片段(α_{source}=0.5)，则结果是 $color_{final}=0.5\times color_{source}+0.5\times color_{dest}$(最终的颜色是源颜色和目标颜色的相混合)。

源 RGB 和源 alpha 函数的默认融合函数是 gl.ONE，目标 RGB 和目标 alpha 函数的默认融合函数是 gl.ZERO，这意味着，为了实际启用融合功能，仅调用以下函数是不够的：

```
gl.enable(gl.BLEND);
```

在实际看到融合效果之前，还必须调用 gl.blendFunc()或 gl.blendFuncSeparate()方法，把融合函数设置为合适的参数。

现在来看看相对高级的 gl.blendFuncSeparate()方法的用法，定义如下所示。

```
void blendFuncSeparate(GLenum srcRGB, GLenum dstRGB,
                       GLenum srcAlpha, GLenum dstAlpha);
```

在有些情形下，单独为源和目标的 RGB 和 alpha 值指定融合函数很有用。gl.blendFuncSeparate()方法有 4 个参数，因此可以为源颜色和目标颜色的 RGB 和 alpha 值分别设置融合函数。参数 srcRGB 和 dstRGB 分别指定源和目标的 RGB 融合函数，srcAlpha 和 dstAlpha 分别指定源和目标的 alpha 融合函数。下面这个例子说明如何调用 gl.blendFuncSeparate()方法。

```
gl.blendFuncSeparate(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA,
                     gl.ZERO, gl.ONE);
```

这里对带参数的 gl.blendFuncSeparate()方法的调用设置了融合，因此 RGB 值的融合方式与前面例子中的 gl.blendFunc()方法一样。但是，为 alpha 值设置融合函数的方式会导致目标 alpha 不受融合的影响。这是十分有用的，例如用目标 alpha 保存一些与透明度无关的信息。

现在，应该进一步了解表 8-1，即使其中一些融合函数很少使用。正如前面已经提到，

第一列为融合函数，可以作为 gl.blendFunc()和 gl.blendFuncSeparate()函数的参数使用。第二列和第三列指定哪些缩放因子(或融合因子)用于指定的融合函数。在表中，(R_s、G_s、B_s、A_s)代表源片段的颜色分量(即传入片段)，而(R_d、G_d、B_d、A_d)代表目标片段的颜色分量。此外，表中还有(R_c、G_c、B_c、A_c)，它表示常量颜色分量，它可以用下面的调用语句来设置。

```
void blendColor(GLclampf red, GLclampf green,
                GLclampf blue, GLclampf alpha);
```

参数 red、green、blue 和 alpha 定义常量融合颜色的(R_c、G_c、B_c、A_c)成分。

表 8-1　融合函数

融合函数	RGB 融合因子	alpha 融合因子
gl.ZERO	(0,0,0)	0
gl.ONE	(1,1,1)	1
gl.SRC_COLOR	(R_s,G_s,B_s)	A_s
gl.ONE_MINUS_SRC_COLOR	$(1,1,1) - (R_s,G_s,B_s)$	$1- A_s$
gl.DST_COLOR	(R_d,G_d,B_d)	A_d
gl.ONE_MINUS_DST_COLOR	$(1,1,1) - (R_d,G_d,B_d)$	$1- A_d$
gl.SRC_ALPHA	(A_s,A_s,A_s)	A_s
gl.ONE_MINUS_SRC_ALPHA	$(1,1,1) - (A_s,A_s,A_s)$	$1- A_s$
gl.DST_ALPHA	(A_d,A_d,A_d)	A_d
gl.ONE_MINUS_DEST_ALPHA	$(1,1,1) - (A_d,A_d,A_d)$	$1-A_d$
gl.CONSTANT_COLOR	(R_c,G_c,B_c)	A_c
gl.ONE_MINUS_CONSTANT_COLOR	$(1,1,1) - (R_c,G_c,B_c)$	$1-A_C$
gl.CONSTANT_ALPHA	(A_c,A_c,A_c)	A_c
gl.ONE_MINUS_CONSANT_ALPHA	$(1,1,1) - (A_c,A_c,A_c)$	$1-A_c$
gl.SRC_ALPHA_STATURATE	(f,f,f)	1

在讨论常量颜色的使用时，应该提到 WebGL 对常量融合函数如何组合为 gl.blendFunc()函数和 gl.blendFuncSeparate()函数进行强制限制。常量颜色与常量 alpha 值不能同时用作融合函数的源因子和目标因子。例如，如果调用 gl.blendFunc()方法时，把其中一个因子设置为 gl.CONSTANT_COLOR 或 gl.ONE_MINUS_CONSTANT_COLOR，把另一个设置为 gl.CONSTANT_ALPHA 或 gl.ONE_MINUS_CONSTANT_ALPHA，就会产生 gl.INVALID_ OPERATION 错误。

在表 8-1 的最后一行中，融合函数为 gl.SRC_ALPAH_SATURATE，这是一个特殊值，

因为它只允许源 RGB 和 alpha 值，在最后一行中第二列的 f 表示以下函数。

$$f = \min(A_s, 1 - A_d)$$

8.3.3 绘制顺序与深度缓冲区

正如前面曾提到，融合可以用来实现对象的半透明效果，方法是使用以下调用设置融合函数。

```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

但是，为了得到这个效果，半透明对象需要按从后向前的顺序绘制(即从最远到最近的顺序)。理由是，用 gl.blendFunc()方法设置的融合公式是与顺序有关的。一个特殊情况是只有两个重叠的半透明的表面，且两个 alpha 值都为 0.5。只有在这种情形下，对象的顺序才不重要。

在使用融合时，另一个需要记住的重要内容是深度缓冲区如何和工作。如果绘制一个半透明的表面，然后在它的背后绘制另一个启用了深度测试的表面，则第二个表面不会显示。这是因为，第二个表面的片段已被深度测试去掉。因此，一般说来，在场景中要在半透明对象之前绘制非透明对象。8.3.4 小节将进一步讨论这方面的内容。

8.3.4 绘制包含不透明对象和半透明对象的场景

如果在一个场景中，同时有不透明对象和半透明对象，通常要启用深度测试，已删除不透明对象背后的对象。如果半透明对象比不透明对象离观察者更近，则需要把半透明对象与不透明对象融合。

已经知道，为了使半透明对象正确融合，需要从后向前的顺序绘制它们。然而，有时很难把所有的半透明对象按从后向前的顺序排序。特别当对象相互渗透时尤其困难。在这种情况下，即使按对象的重心位置排列对象的顺序，也无法保证所有的表面都正确。

无法保证场景中每个表面都正确排序时，会希望确保这些对象的表面至少是可见的。如果是这样，则需要把深度缓冲区设置为只读，但是仍然启用深度测试。可以在 WebGL 应用程序中用下面的调用实现。

```
gl.depthMask(false)
```

调用之后，深度测试仍然可执行，这意味着，如果深度缓冲区中的值显示当前片段被绘制缓冲区中的像素遮挡住，则该片段会被丢弃。这里重要的区别是深度缓冲区是只读的，不会被新值更新。这样，深度缓冲区保存了通过调用 gl.depthMask(false)方法禁止写入深度缓冲区时的值。

如果在深度测试已启用而深度缓冲区设置为只读模式之后绘制半透明对象，则至少可以保证所有的半透明对象都是可见的，不会因为另一个半透明对象处于它们之前而被丢弃。

为了再次启用深度缓冲区的可写入模式，只需要再次调用 gl.depthMask()方法，并且将其参数设置为 true。

作为小结，下面这段代码说明了同时包含不透明对象和半透明对象的场景如何绘制的一般思路。

```
// 1. Enable depth testing, make sure the depth buffer is writable
//    and disable blending before you draw your opaque objects.
gl.enable(gl.DEPTH_TEST);
gl.depthMask(true);
gl.disable(gl.BLEND);

// 2. Draw your opaque objects in any order (preferably sorted on state)

// 3. Keep depth testing enabled, but make depth buffer read-only
//    and enable blending
gl.depthMask(false);
gl.enable(gl.BLEND);

// 4. Draw your semi-transparent objects back-to-front

// 5. If you have UI that you want to draw on top of your
//    regular scene, you can finally disable depth testing
gl.disable(gl.DEPTH_TEST);

// 6. Draw any UI you want to be on top of everything else.
```

8.3.5　修改融合公式中的默认运算符

下面再来分析通用的融合公式：

$$color_{final} = factor_{source} \times color_{source} \ op \ factor_{dest} \times color_{dest}$$

已经介绍过如何配置这个融合公式，即如何用 gl.blendFunc()和 gl.blendFuncSeparate()设置 $factor_{source}$ 和 $factor_{dest}$。此外，还知道源与目标相融合使用的默认数学运算符是加法。如果想修改，可以使用 gl.blendEquation()和 gl.blendEquationSeparate()方法，它们的定义如下所示。

```
void blendEquation(GLenum mode);

void blendEquationSeparate(GLenum modeRGB, GLenum modeAlpha);
```

这两个方法的参数可以取以下 3 个值。

- gl.FUNC_ADD——把源运算量加到目标运算量中，这是默认值。
- gl.FUNC_SUBTRACT——从源运算量中减去目标运算量。
- gl.FUNC_REVERSE_SUBTRACT——从目标运算量中减去源运算量。

与这 3 个运算符对应的通用融合公式如下所示。

$$color_{final} = factor_{source} \times color_{source} + factor_{dest} \times color_{dest}$$
$$color_{final} = factor_{source} \times color_{source} - factor_{dest} \times color_{dest}$$
$$color_{final} = factor_{dest} \times color_{dest} - factor_{source} \times color_{source}$$

8.3.6 使用预乘 alpha 值

为了根据 alpha 通道值进行融合，需要保证颜色值包含 alpha 通道。如果颜色来自顶点的属性，则必须指定颜色的全部 4 个分量。如果颜色来自于纹理，则纹理格式必须包含 alpha 通道。PNG 图像就是支持 alpha 通道的一个例子。

有两种方法在应用计算机图形中 alpha 通道。

- 预乘 alpha(Premultiplied alpha，也称关联 alpha)
- 非预乘 alpha(也称非关联 alpha)

预乘 alpha 是指 RGB 值在存储之前已与 alpha 通道相乘。这意味着，半透明红色的 RGBA 值可以表示为(0.5,0.0,0.0,0.5)。

非预乘 alpha 是指 RGB 值在存储之前还没有与 alpha 通道相乘。这意味着，半透明红色的 RGBA 值可以表示为(1.0,0,0,0.5)。PNG 图像通常使用非预乘 alpha。

用 WebGL 和融合时，如准备使用预乘 alpha 或非预乘 alpha，必须事先计划好。虽然 PNG 图像使用非预乘 alpha，但是用 gl.texImage2D()或 gl.texSubImage2D()以纹理形式加载图像时，也可以指定 alpha 值与 RGB 值相乘。用下面的方法调用 WebGL API 可以指定预乘 alpha 模式。

```
gl.pixelStorei(gl.UNPACK_PREMULTIPLY_ALPHA_WEBGL, true);
```

注意，如果已经有预乘 alpha 值，则经常用于半透明效果的默认融合公式不能再表示为以下形式：

$$color_{final} = \alpha_{source} \times color_{source} + (1 - \alpha_{source}) \times color_{dest}$$

由于 RGB 值已与 alpha 值相乘，因此对于预乘 alpha 值，相应的融合公式表示为：

$$color_{final} = colorPreMult_{source} + (1 - \alpha_{source}) \times color_{dest}$$

上式中，$colorPreMult_{source}$代表源预乘颜色，融合公式可以用下面的调用来设置。

```
gl.blendFunc(gl.ONE, gl_ONE_MINUS_SRC_ALPHA);
```

8.4 深入讨论 WebGL

虽然本书已经介绍很多有关 WebGL 的内容，但是与大多数令人感兴趣的技术一样，总是还有更多的内容需要学习。下面将介绍哪里可以找到与 WebGL 有关的资源。

8.4.1 使用 WebGL 框架

本书主要介绍 WebGL，并且用一个比较小的 JavaScript 库(如 glMatrix)确保将注意力集中到 WebGL 的学习上。这种方法的优点是可以在低层控制 GPU 和操作的执行方式。此外，由于 WebGL 是一个开放的标准，与 OpenGL 和 OpenGL ES 非常相似，因此用户很容易在 WebGL 应用程序中重用这两个技术的思想或代码。关于 OpenGL，现在有大量的图书、技术文章和其他文档，这些资料可以帮助你更容易掌握 WebGL。

然而，如果不喜欢 WebGL 作为一个低层 API 这个事实，想用更高层的东西编写应用程序，也存在许多建立在 WebGL 之上的高层库和框架，如下所示。

- GLGE
- PhiloGL
- SceneJS
- SpiderGL
- Three.js

这些框架的目的是通过抽象 WebGL API 的许多细节来加快应用程序的开发速度。

找到这些框架最容易的方法是使用 google 搜索引擎。此外，也可以参考 Khronos WebGL 维基页面，里面有一部分专门介绍用户的贡献，其中提到几个不同的 WebGL 框架。

8.4.2 发布到 Google Chrome Web Store

Google Chrome Web Store 允许用户发布自己的 Web 应用程序，这样 Google Chrome 用户可以很容易找到它们。如果正在开发支持 WebGL 的应用程序，可能想把它们发布到 Chrome Web Store 中。

除了用户可以更容易找到自己的应用程序这个事实外，Chrome Web Store 还允许向使用用户自己应用程序的其他用户收费。Chrome Web Store 的在线文档有详细的介绍，还有一些入门性质的视频，因此很容易入门。

8.4.3 使用额外资源

学习 WebGL 的最好办法是自己动手开发 WebGL 应用程序。此外，分析其他人的应用程序也可以学到很多知识。Chrome 开发人员工具(Chrome Developer Tool)和 Firebug 使这一切都变得容易，因为它允许研究其他人开发 WebGL 应用程序时使用的技术和思想。

下面列出一些有用的在线资源，它们包含了大量的有关 WebGL 的信息。

- Khronos 专门为 WebGL 提供的主页(www.khronos.org/webgl/)包含了许多到 WebGL 资源的有用链接。在这个主页中我们可以找到以下链接。
  - 最新的 WebGL 规范
  - Khronos WebGL 维基
  - 一些优秀的 WebGL 演示程序
  - WebGL 论坛

- Giles Thomas 的网站(http://learningwebgl.com/blog) 包含一些有关 WebGL 的教程，以及 Web 上有关 WebGL 的最新消息。
- Brandon Jones 的博客(http://blog.tojicode.com/)包含一些关于 WebGL 的文章，以及用 WebGL 开发游戏时需要考虑的问题。
- Gregg Tavares、Kenneth Russell、Ben Vanik 及其他开发人员提供的会议演示文稿。找到这些演示文档最容易的方法是用 Google 搜索引擎。其中绝大多数都可以通过 Khronous WebGL 维基的链接访问。
- 来自 Mozilla 的 WebGL 教程(https://developer.mozilla.org/en/WebGL)。
- Chrome 体验(www.chromeexperiments.com)中包括许多有趣的 WebGL 演示程序，从这些程序可以学到新的技术。注意，这些演示程序并非都是基于 WebGL 的。
- HTML5 Rocks(www.html5rocks.com)提供了许多有用的 HTML5 资源。如果想把 WebGL 与其他 HTML5 功能结合起来，需要访问这个非常不错的网站。虽然 WebGL 不是 W3C HTML5 规范的一部分，但是该网站提供了许多有关 WebGL 的有用资料。

这里列出了一些有用的资源，在你通向 WebGL 专家的道路上可能非常有用。然而，事情总是变化得很快，因此为了能跟上最新的技术，最好在 Web 中搜索最新的和令人感兴趣的 WebGL 资料。

8.5 小结

本章从较高层次介绍了 WebGL 的底层实现细节。还介绍了一些重要的硬件和软件组成。

本章最重要的部分是有关 WebGL 的性能优化。本章介绍如何执行不同的测试，如何利用这些测试判断程序中瓶颈的位置。本章也提供了一些有关 WebGL 性能的一般性建议和针对性建议，利用这些建议，用户可以知道出现性能瓶颈的位置。

最后，本章还介绍了融合技术，以及绘制一个包含非透明和半透明的场景时需要考虑的因素。