HZ BOOKS
华章IT

[PACKT]
PUBLISHING

首部全面讲解Three.js开发的实用指南，循序渐进讲解Three.js的各种功能

通过大量交互式示例，深入探索使用开源的Three.js库创建绚丽三维图形的实用方法和技巧

Learning Three.js

The JavaScript 3D Library for WebGL

[美] Jos Dirksen 著

李鹏程 译

Three.js 开发指南

机械工业出版社
China Machine Press

Three.js开发指南

[美] Jos Dirksen 著
李鹏程 译

Learning Three.js

The JavaScript 3D Library for WebGL

机械工业出版社

图书在版编目（CIP）数据

Three.js 开发指南 /（美）德克森（Dirksen, J.）著；李鹏程译 . —北京：机械工业出版社，2015.1
（Web 开发技术丛书）
书名原文：Learning Three.js: The JavaScript 3D Library for WebGL

ISBN 978-7-111-48832-3

I. T… II. ① 德… ② 李… III. Java 语言—程序设计—指南 IV. TP312-62

中国版本图书馆 CIP 数据核字（2014）第 293435 号

本书版权登记号：图字：01-2014-2030

Three.js 开发指南

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

责任编辑：陈佳媛　　责任校对：董纪丽

印　　刷：三河市宏图印务有限公司　　版　　次：2015 年 3 月第 1 版第 1 次印刷

开　　本：186mm × 240mm　1/16　　印　　张：17

书　　号：ISBN 978-7-111-48832-3　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991　88361066　　投稿热线：（010）88379604

购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

Preface 前　言

在过去的两年里，浏览器变得更加强大了，而且能够作为平台来运行复杂的应用和图形。但大部分的图形都是二维的。现在大部分的浏览器已经实现了 WebGL，它们不仅可以创建二维图形和应用，还可以充分利用 GPU，创建漂亮的、高性能的三维应用。

直接使用 WebGL 编程非常复杂。你需要了解 WebGL 的内部细节，学习复杂的着色器语法，才能用好 WebGL。Three.js 库提供了一套基于 WebGL 的、非常易用的 JavaScript API，这样你不必学习 WebGL 的细节就可以创建漂亮的三维图形。

Three.js 库提供了大量功能和 API，你可以通过它们直接在浏览器中创建三维场景。本书提供了大量的交互式示例和代码，从中你可以学习到 Three.js 提供的所有 API。

本书内容

第 1 章讲述使用 Three.js 库的基本步骤。到这章结束的时候，你就可以创建出第一个 Three.js 场景。你可以直接在浏览器中创建三维场景，并制作动画。

第 2 章将会讲解 Three.js 库使用过程中你所需要理解的基础组件。你将会学习使用光源、网格、几何体、材质和相机。通过这章你会对 Three.js 库提供的各种光源及场景中使用的相机有一个大体的了解。

第 3 章进一步讲解场景中的各种光源。该章会展示几个例子，并解释如何使用各种光源，该章所介绍的光源包括：SpotLight、DirectionLight、AmbientLight、PointLight、HemisphereLight，以及 AreaLight。另外，该章还展示如何在光源中应用 LensFlare 效果。

第 4 章讨论 Three.js 库中可以用于网格的各种材质。这一章将会展示所有可以在材质中设置的属性，每种属性都有其特定的用途。并且这章还提供交互式的例子，通过它们你可以试验 Three.js 库中各种可用的材质。

第 5 章开始探讨 Three.js 库中的各种几何体。在这一章里你会学习如何创建、设置 Three.js 里的几何体，而且你还可以通过交互式的示例来试验，该章介绍的几何体包括：平

面、圆、自定义形状、方块、球体、圆柱体、圆环、环面纽结和多面体。

第 6 章继续第 5 章未讲完的内容，展示如何设置、使用 Three.js 库中的高级几何体，例如凸面体和扫描体。该章将会介绍如何从二维几何体中拉出三维几何体，以及如何使用二元运算将几何体组合起来。

第 7 章讲解如何使用 Three.js 库中的粒子系统。你将会学习如何从零开始创建粒子系统，以及如何基于已有的几何体创建粒子系统。你还会学习如何使用精灵和粒子材质来改变粒子的外观。

第 8 章展示如何从外部源导入网格和几何体。你将会学习如何使用 Three.js 库的 JSON 格式来存储几何体和场景。这一章还解释了如何从 OBJ、DAE、STL、CTM 和 PLY 格式的文件中加载模型。

第 9 章探讨各种动画效果，使场景活动起来。你将会学习如何结合 Tween.js 库使用 Three.js，以及基于变形和骨骼的动画效果。

第 10 章进一步展开讨论第 4 章引入的材质。这一章会深入讲解材质的细节。该章介绍了各种可用的材质，以及应用到网格上时如何控制材质。另外，在这一章你还会看到如何将 HTML5 视频的输出和 HTML5 画布中的元素作为纹理。

第 11 章探讨如何使用 Three.js 库来对已经渲染过的场景进行后期处理。通过后期处理，你可以对已经渲染过的场景应用某些特效，如模糊、移轴、陈旧。另外，你还会学习如何创建自定义的后期处理效果，以及创建自定义顶点和片段着色器。

第 12 章讲解如何在 Three.js 的场景中添加物理效果。有了物理效果后，你就可以检测物体之间的碰撞，使它们能够对重力产生反应，并产生摩擦。这一章将会展示如何使用 JavaScript 的物理库来达到上述效果。

阅读之前的准备

阅读本书你要准备的只是一个文本编辑器（例如 Sublime 文本编辑器），用来编辑示例，以及一个现代的浏览器，用来显示这些示例。有些示例需要一个本地的 Web 服务器，但是在第 1 章，你将会学习如何搭建一个非常轻量的 Web 服务器来使用本书中的示例。

在我们开始之前，先对书中示例可能出现的问题简单提一下。第 1 章大致介绍了一下支持 WebGL 的浏览器，这是运行 Three.js 所必需的。现在的浏览器，诸如 Chrome、Firefox 和 Internet Explorer 可以很好地支持 HTML5 标准。但是仍然有一些细节要注意。当浏览器有新的版本时，它们有可能会不支持 WebGL 的某些特性。例如，在本书完成的时候，Windows 7 上的 Chrome 和 Firefox 在运行第 11 章的例子时会有问题。所以在尝试这些示例时，请确保升级到最新版的 Chrome 和 Firefox。

读者对象

本书对所有已经了解 JavaScript 并开始想在浏览器里创建三维图形的人都会有很大的帮助。你不必去了解任何高级的算法或 WebGL，只要具有一般的 JavaScript 和 HTML 知识即可。所有示例代码可以登录华章网站（www.hzbook.com）免费下载，本书中用到的工具也都是开源的。所以，如果你想要学习如何创建漂亮的、交互的三维图形，并且可以在任何现代的浏览器上运行，那么这本书就是你想要的。

致　谢 *Acknowledgements*

写书是一件耗时且艰苦的事情。如果没有很多人的支持和帮助，我不可能完成。我想在这里感谢他们。

首先是我的项目协调人，Leena Purkait，要不是她帮助我协调所有的细节，从而让我可以专注于内容的创作，我根本不可能按时完成所有的章节。我还要感谢 Ritika Singh 和 Anita Nayak，她们花了很多时间和精力来完善本书。

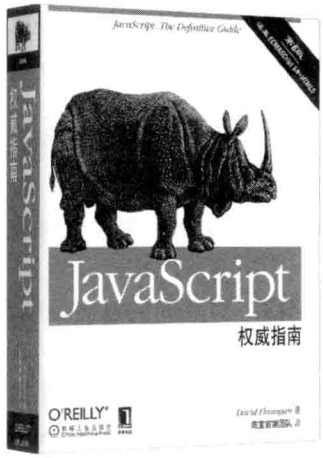
还有来自 Packt 出版社的好多人，他们在我写书、审校期间给了我很多帮助，保证整个过程顺利进行。干得不错，伙计们！

当然，我还要感谢 Ricardo Cabello，即 Mr.dò_ób，Three.js 正是他的杰作。

非常感谢本书的各位审校人。感谢你们那些非常好的反馈和建议，对本书的改进帮助很大。正是你们的积极评论才成就了本书。

我还没有提到最重要的人。我想感谢我的太太 Brigitte，她一再地容忍我在周末和晚上将大量的时间花费在笔记本电脑上。我要感谢我的女儿 Sophie，她经常把我从键盘上拉开，逗我开心。还有我的女儿 Amber，尽管只有几周大，但是她让我对生命中真正重要的事情和时刻充满感激。

推荐阅读



JavaScript权威指南（原书第6版）

从1996年以来，本书已经成为 JavaScript 程序员的《圣经》。

第6版涵盖HTML5和ECMAScript 5。很多章节完全重写，以便跟得上当今的最佳Web开发实践。该版本的新增章节描述了jQuery和服务器端JavaScript。

对于那些希望学习Web编程语言的有经验的程序员和希望精通JavaScript的当前JavaScript程序员，本书最适合不过了。

深入理解PHP：高级技巧、面向对象与核心技术（原书第3版）

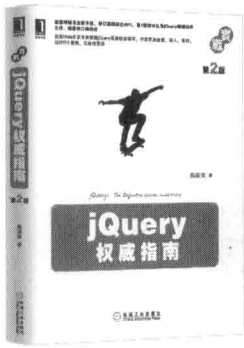
国际知名Web开发专家和技术畅销书作家最新力作，PHP领域经典著作。

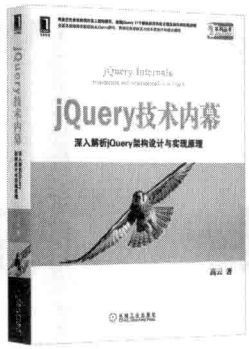
从编程技巧、面向对象和扩展三个角度系统讲解和总结了成为中高级PHP程序员应该具备的技术和技能，包含大量实用案例，极具实践指导意义。

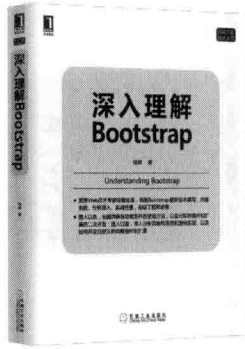
如果你已经具备一定的PHP编程基础，想使开发效率更高，想把应用做得更好，那么这本书应该是你需要阅读的。本书旨在为想修炼成为高级PHP程序员的中初级PHP程序员提供实用的方法和建议。

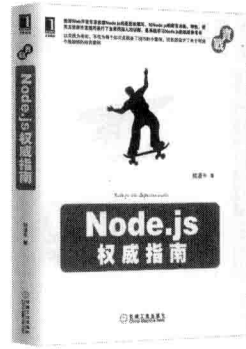
Web前端开发&设计经典

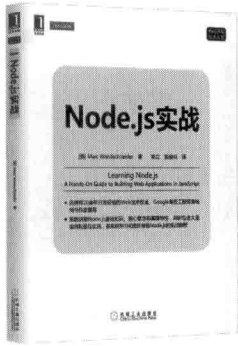
框架篇

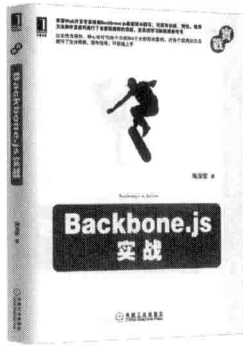


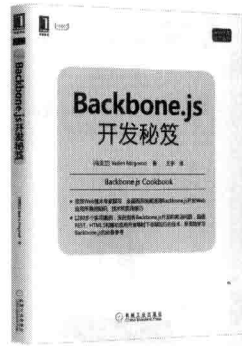






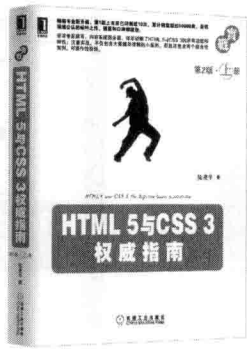


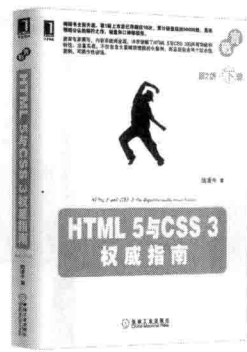


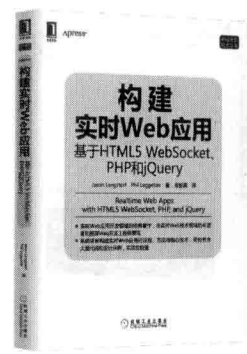


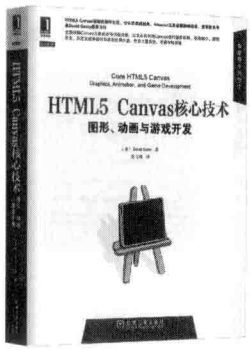
Web前端开发&设计经典

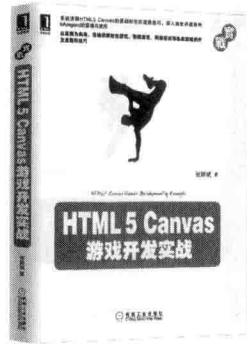
HTML5&CSS3篇

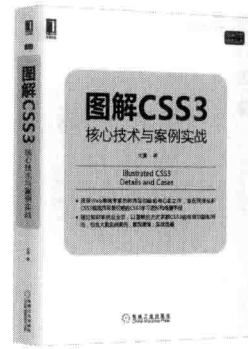












Contents 目　录

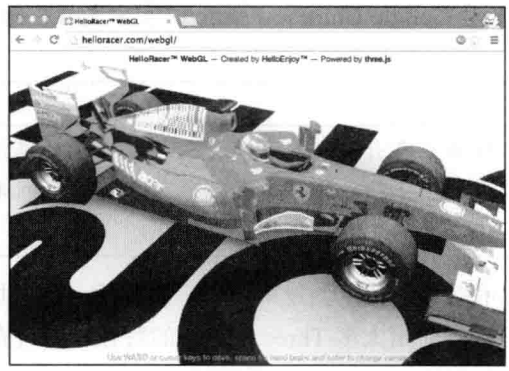
第 1 章 Chapter 1

用 Three.js 创建你的第一个三维场景

如今浏览器的功能越来越强大，而且这些功能可以通过 JavaScript 直接调用。你可以用 HTML5 标签轻松地添加视频和音频，而且可以在 HTML5 画布上创建各种交互组件。现在这个功能集合里又有了一个新成员，即支持 WebGL。通过 WebGL，你可以直接使用显卡的计算资源，创建高性能的二维和三维计算机图形，然后在 JavaScript 里直接使用 WebGL 编程，创建三维场景并生成动画，这个过程非常复杂，而且容易出错。Three.js 库可以简化这个过程。Three.js 可以帮助我们的地方如下：

- 创建简单的和复杂的三维图形
- 在三维场景中生成动画、移动物体
- 在物体上应用纹理和材质
- 从三维建模软件中加载图形
- 创建基于样条曲线的二维图形

只要几行代码，你就可以从简单的三维图形创建出逼真的、实时的场景来，如下图所示：



在第 1 章中我们会直接开始探究 Three.js，通过创建几个例子来展示 Three.js 是如何工作的。你可以自己亲手试验一下这几个例子。但我们并不会深入探究所有的技术细节，这些细节我们将会在后面的章节中讲解。本章会探讨下面几方面的内容：

- 使用 Three.js 时要用到的工具
- 下载本书附带的源码和示例
- 创建你的第一个 Three.js 场景
- 应用材质、光线和动画来改进你创建的场景
- 引入几个辅助库，用于统计和控制场景

我们首先会简单介绍一下 Three.js，然后立即开始讲解第一个示例及其代码。在开始之前我们先来快速浏览一下现在市面上几个最负盛名的浏览器，以及它们对 WebGL 的支持情况。

现在支持 Three.js 的浏览器如下表：

浏 览 器	支 持 情 况
Mozilla Firefox	4.0 版以后开始支持
Google Chrome	第 9 版以后开始支持
Safari	5.1 版开始支持，以及那些新安装在 Mac OS X 的山狮版、狮子版和雪豹版操作系统上的版本。确保在 Safari 中打开了 WebGL 选项。为此你可以找到 Preferences | Advanced 菜单，选中 Show develop menu in menu bar。然后选中菜单项 Develop | Enable WebGL
Opera	12.00 版以后开始支持。要让 Opera 支持 WebGL，你还需要打开 opera:config 文件，设置 WebGL，并将 Enable Hardware Acceleration 置为 1。然后重启浏览器
Internet Explorer	Internet Explorer 是长期以来唯一一个不支持 WebGL 的主流浏览器。从 IE11 开始，微软才开始支持 WebGL

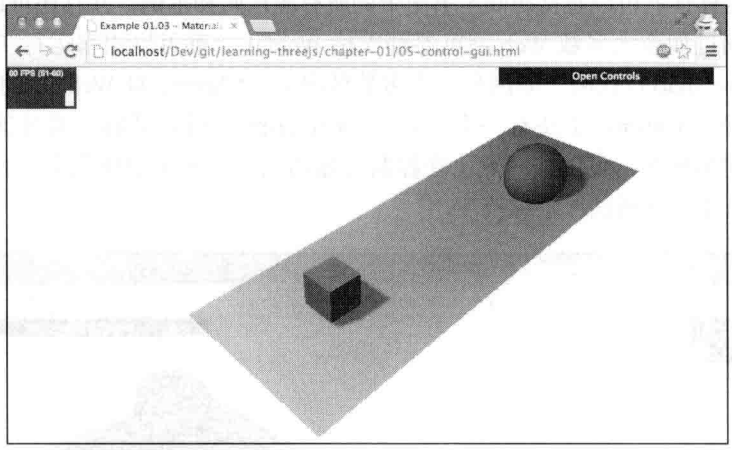
基本上 Three.js 可以在任何流行的浏览器上运行，除了 IE 的大部分版本。所以如果你想用旧版本的 IE，你有两个选择：你可以用 Google Chrome Frame 来支持 WebGL，可以从如下网址下载：https://developers.google.com/chrome/chrome-frame/。除了 Google Chrome Frame，你也可以使用 iewebgl 插件，下载网址是 http://iewebgl.com/。在 IE 上安装这个插件，使其支持 WebGL。

Google 已经停止了对 Google Chrome Frame 的官方支持，而且 Google Chrome Frame 也不支持 IE10。

现在 Three.js 的工程师正在开发一个遵从 CSS-3D 规范的新渲染器，而这个规范大多数浏览器都支持（包括 IE10）。除了桌面浏览器之外，手机和平板电脑上的浏览器也都支持 CSS-3D。

通过本章的学习，你可以直接开始创建第一个三维场景，这个场景应该可以在上述任意一个浏览器里运行。我们不会介绍太多 Three.js 复杂的特性，不过在本章结束的时候你应该

可以建立一个 Three.js 场景，如下图所示：



通过创建这个场景，你可以学习到 Three.js 的一些基础知识，还可以创建第一个动画。在开始学习这个示例之前，我们先介绍几个工具，这些工具可以帮助你更方便地使用 Three.js。我们还要学习如何下载本书中展示的那些示例。

1.1　使用 Three.js 的前提条件

Three.js 是一个 JavaScript 库，所以当你用 Three.js 创建一个 WebGL 应用时，你所需的仅仅是一个文本编辑器和一个支持 WebGL 的浏览器（用来渲染结果）。我非常诚恳地向你推荐三个 JavaScript 编辑器，在过去的几年里我一直在使用它们。

- WebStorm：这是一个由 JetBrains 公司开发的编辑器，对于编写 JavaScript 程序支持得非常好。它提供代码补全、自动部署等功能，还可以直接在编辑器里调试 JavaScript。除此之外，WebStorm 还能很好地支持 Git（以及其他的版本控制系统）。你可以从网址 http://www.jetbrains.com/webstorm/ 获取一个试用版。
- **Notepad++**：这是一个通用型的编辑器，支持很多的编程语言。用它可以很方便地排版和格式化 JavaScript 代码。

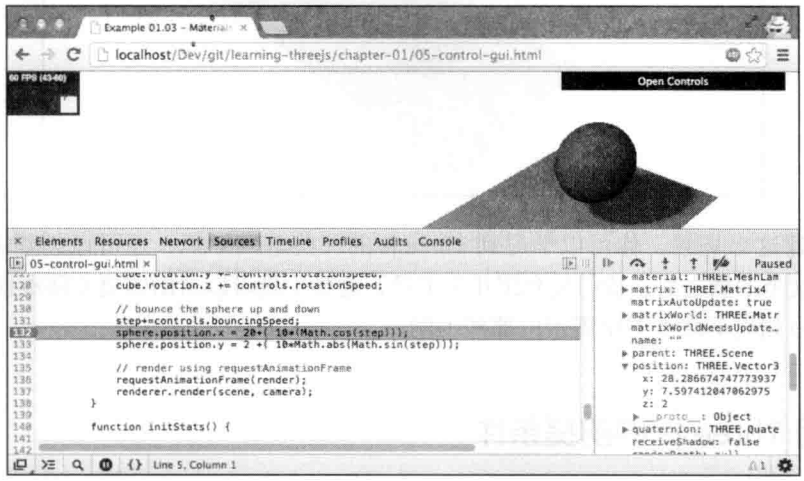
Notepad++ 只有 Windows 版。

- **Sublime Text Editor**：这是一个非常好用的小型编辑器，对于编写 JavaScript 代码支持良好。除此之外，它还提供了很多很有用的选择和编辑功能，一旦你习惯了这些功能，那它就是一个真正不错的 JavaScript 编辑环境。Sublime Text Editor 也有免费试用版，你可以从网址 http://www.sublimetext.com/ 下载。

即使你不用这三个编辑器，依然有很多编辑器可供选择，开源的、商业的都有。它们都可以用来编辑 JavaScript 代码，创建 Three.js 项目。网上有一个开源项目有点儿意思，你或

许可以试一试，网址是 http://c9.io。这是一个云端的 JavaScript 代码编辑器，而且可以登录你的 GitHub 账户。这样你就可以直接访问本书附带的所有源代码和示例，并且还可以试验它们。

前面我提到过如今大多数 Web 浏览器都支持 WebGL，都可以用来运行 Three.js 的示例。通常我在 Chrome 里运行代码。原因是，大多数情况下，Chrome 对 WebGL 的支持是最好的，性能也不错，而且 Chrome 里还有一个很好的 JavaScript 代码调试器。通过这个调试器你可以很快地找到问题所在，可以加断点、在控制台输出信息。在本书里我会给你一些该调试器的使用建议，以及一些调试提示和技巧。



有关 Three.js 的介绍到此为止，接下来让我们获取源代码并开始创建第一个场景。

1.2 获取源代码

本书所有的源代码都可以从 GitHub（https://github.com）下载。GitHub 是一个在线的基于 Git 的代码仓库，你可以通过它存储、访问源代码，并管理源代码的版本。你可以用两种方式获取源代码：

- 克隆（clone）Git 代码仓库
- 下载代码压缩包，然后解压

在后面的章节中我们会详细探讨这两种方式。

1.2.1 用 Git 克隆代码仓库

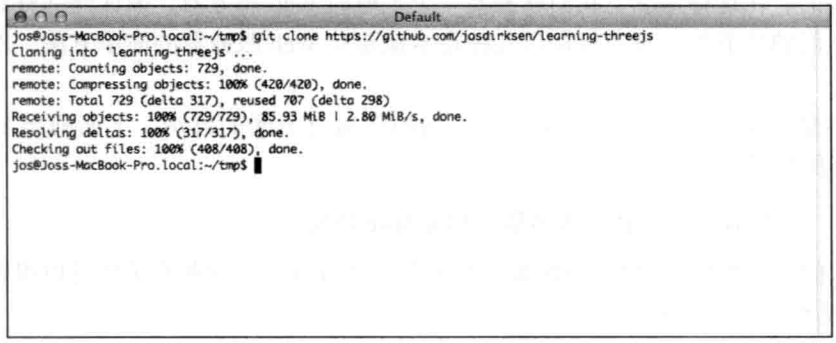
Git 是一个开源的、分布式版本控制系统。在写这本书的时候我就是用 Git 创建并管理书中示例的代码。这就是我会用 GitHub 这个免费的、在线 Git 代码仓库的原因。你可以通过访问 https://github.com/josdirksen/learning-threejs 以查看代码仓库中的内容。

你可以使用 git 命令行工具来克隆整个代码仓库，从而获取所有示例的代码。为此，你

首先要下载一个适合你的操作系统的 Git 客户端。对于现在大多数的操作系统，都可以从 http://git-scm.com 下载相应的客户端，或者你也可以使用 GitHub 自己提供的客户端（Mac 版和 Windows 版）。安装好 Git 之后你就可以用它来克隆本书的代码仓库。打开一个命令提示窗口，转到一个你想保存源代码的目录。然后在这个目录里运行如下的命令：

```
https://github.com/josdirksen/learning-threejs
```

运行该命令即可开始下载所有的示例，如下图所示：



现在，本书所有的示例都将包含在文件夹 learning-three.js 里。

1.2.2　下载并解压压缩包

如果你不想用 Git 直接从 GitHub 下载源代码，你也可以下载一个压缩包。访问 https://github.com/josdirksen/learningthreejs，并点击下载链接，如下图所示：

将压缩包解压到一个目录，这样你就拥有了所有的示例。

1.2.3 测试示例

现在你已经下载了或克隆了所有的源代码，让我们来快速检查一下，看看是不是所有的东西都准备好了，并熟悉一下文件夹的结构。源代码和示例是按照章节来组织的。查看这些例子有两种方法。你可以直接在浏览器中打开解压的或克隆的目录，并运行某个示例；或者你也可以在本地安装一个 Web 服务器。对于大多数的基础性的示例，第一个方法就可以了。但是当我们要加载一些外部资源时，例如三维模型和纹理图片，仅仅打开一个 HTML 文件是不够的。在这种情况下我们就需要一个本地的 Web 服务器，确保外部资源能够正确加载。在下面的几节里，我们将探讨几种在本地建立 Web 服务器的简单方法，以方便我们测试。

建立本地的 Web 服务器非常简单，不同的环境有不同的方法。下面列出了几个建立 Web 服务器的例子。

1.2.3.1 用 Python——适合大多数 UNIX/Mac 系统

大多数的 UNIX/Linux/Mac 系统都已经安装了 Python。在这些系统中可以很容易地建立本地 Web 服务器：

```
> python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

在你签出或下载的源代码目录下运行上述命令。

1.2.3.2 用 NPM——如果你已经安装了 Node.js

如果你已经用过 Node.js，那么很可能你已经安装了 NPM。使用 NPM 时你有两种选择，它们都可以快速建立用于测试的本地 Web 服务器。第一种是使用 **HTTP Server** 模块：

```
 > npm install -g http-server
 > http-server
Starting up http-server, serving ./ on port: 8080
Hit CTRL-C to stop the server
```

你也可以用 Simple HTTP Server：

```
> npm install -g simple-http-server
> nserver
simple-http-server Now Serving: /Users/jos/git/Physijs at http://
localhost:8000/
```

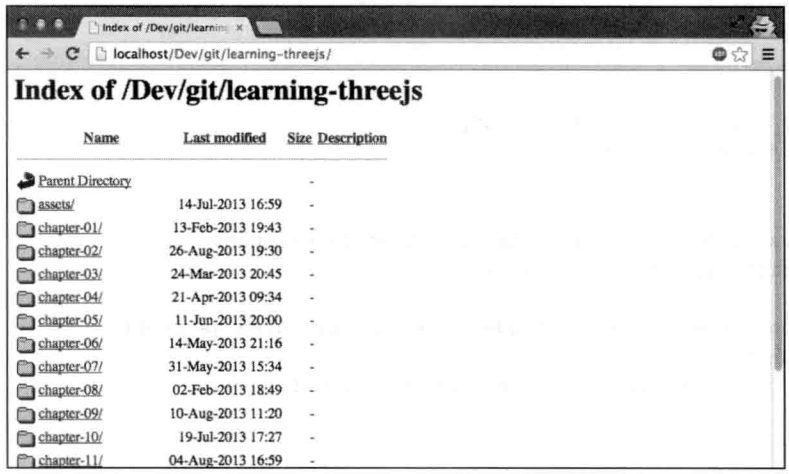
但是第二种方式有个缺点，它不能自动地列出文件夹里的东西，而第一种方式就可以。

1.2.3.3 用 Mongoose——在 Mac/Windows 上可以移植

如果你没有安装 Python 或者 NPM，那么你还可以用一个简单的、可移植的 Web 服务器，Mongoose。首先，从 https://code.google.com/p/mongoose/downloads/list 下载特定平台的

二进制文件。如果是 Windows 平台，把下载好的文件复制到示例所在的目录，双击它，即可启动一个 Web 浏览器，并显示该目录中的内容。

对于其他操作系统，你也必须将这个可执行程序复制到目标目录，但不能双击，只能通过命令行来启动。在这两种情况下，一个本地的 Web 服务器都会启动，端口是 8080。



点击各章的条目，我们就可以展示并访问这一章的示例。在本书中，当我提到某个示例时，我会指出示例的名称和它所在的文件夹，这样你就可以直接测试它，以及查看源代码。

到此为止，你应该已经安装好了一个编辑器，并且可以访问所有的源代码。现在是时候去创建我们的第一个场景了。

1.3　创建 HTML 页面框架

首先我们要创建一个 HTML 页面的框架，这样就可以在后面所有的示例中将它作为基础模板。这个 HTML 页面的框架如下所示：

```
<!DOCTYPE html>

<html>

  <head>
    <title>Example 01.01 - Basic skeleton</title>
    <script type="text/javascript"
            src="../libs/three.js"></script>
    <script type="text/javascript"
            src="../libs/jquery-1.9.0.js"></script>
    <style>
        body{
```

```
            /* set margin to 0 and overflow to hidden,
               to use the complete page */

            margin: 0;
            overflow: hidden;
        }
    </style>
  </head>
  <body>

<!-- Div which will hold the Output -->
<div id="WebGL-output">
</div>

<!-- Javascript code that runs our Three.js examples -->
    <script type="text/javascript">

    // once everything is loaded, we run our Three.js stuff.
    $(function () {
          // here we'll put the Three.js stuff
    });

    </script>
  </body>
</html>
```

从上述代码可以看出这个框架是一个非常简单的 HTML 页面，上面只有寥寥几个元素。在 <head> 元素里我们会加载本书示例中用到的外部 JavaScript 库。本书中的所有示例至少会用到上述代码中列出的两个 JavaScript 库：Three.js 和 jquery-1.9.0.js。在 <head> 元素里我们还添加了几行 CSS 代码。当我们创建一个全页面的 Three.js 场景时，这几行 CSS 代码可以移除所有的滚动条。在这个页面的 <body> 里只有一个 元素。当我们写 Three.js 程序代码时，会把 Three.js 渲染器的输出指向这个元素。在前面的代码片段里，你会看到已经有几行 JavaScript 代码了。这一小段代码的作用是在页面加载结束时，使用 jQuery 调用一个匿名的 JavaScript 函数。我们所有的 Three.js 程序代码都写在这个匿名函数里。

Three.js 有两个版本：

- Three.min.js：通常在互联网上部署网站的时候用的就是这个库。这是用 UglifyJS 创建的 Three.js 库的压缩版，其尺寸只有原来 Three.js 库的一半。本书所有示例和代码都基于 Three.js r60 这个项目，这个项目是 2013 年 8 月发布的。
- Three.js：这是正常的 Three.js 库。本书中的示例用的就是这个库。如果你能够读懂并理解 Three.js 的源代码，那么调试起来会简单很多。

如果在浏览器里打开这个网页，那么结果并不令人吃惊。正如所料，你看到的只是一个空白的网页。

下一节你将会学习如何添加两个三维物体，并将渲染的结果输出到在 HTML 框架页面中定义的 元素里。

1.4　渲染并展示三维对象

现在你要创建第一个场景，添加几个物体和一个相机。我们的第一个例子包含下面几个对象：

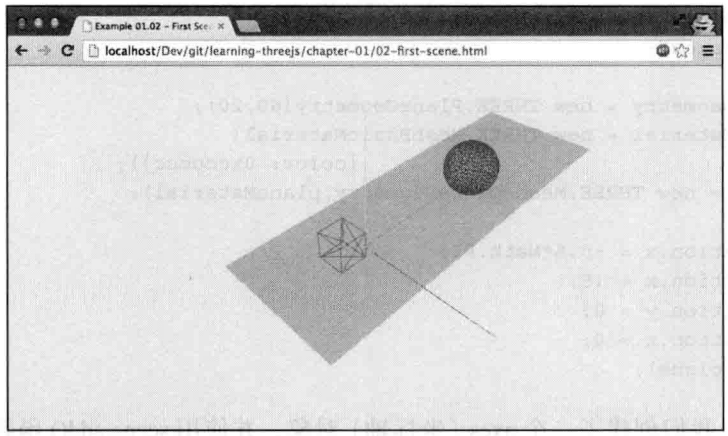
对　象	描　述
Plane（平面）	作为地面的二维矩形。渲染的结果是屏幕中央的一个灰色的矩形
Cube（方块）	一个三维立方体，渲染为红色
Sphere（球体）	一个三维球体，渲染为蓝色
Camera（相机）	相机决定着你所能看到的输出结果
Axes（轴）	x、y 和 z 轴。这是一个有助于调试的工具，通过它可以找到物体渲染的位置

我先给你看看代码（带注释的代码文件可以在文件夹 chapter-01 下找到，文件是 02-first-scene.html），然后我会解释一下代码的功能：

```
<script type="text/javascript">


    $(function () {
        var scene = new THREE.Scene();

        var camera = new THREE.PerspectiveCamera(45
                            , window.innerWidth / window.innerHeight
                            , 0.1, 1000);

        var renderer = new THREE.WebGLRenderer();
        renderer.setClearColorHex(0xEEEEEE);
```

```
        renderer.setSize(window.innerWidth, window.innerHeight);

        var axes = new THREE.AxisHelper( 20 );
        scene.add(axes);

        var planeGeometry = new THREE.PlaneGeometry(60,20,1,1);
        var planeMaterial = new THREE.MeshBasicMaterial(
                                            {color: 0xcccccc});
        var plane = new THREE.Mesh(planeGeometry,planeMaterial);

        plane.rotation.x=-0.5*Math.PI;
    plane.position.x = 15;
    plane.position.y = 0;
    plane.position.z = 0;
    scene.add(plane);

    var cubeGeometry = new THREE.CubeGeometry(4,4,4);
    var cubeMaterial = new THREE.MeshBasicMaterial(
                    {color: 0xff0000, wireframe: true});
    var cube = new THREE.Mesh(cubeGeometry, cubeMaterial);

    cube.position.x = -4;
    cube.position.y = 3;
    cube.position.z = 0;

    scene.add(cube);

    var sphereGeometry = new THREE.SphereGeometry(4,20,20);
    var sphereMaterial = new THREE.MeshBasicMaterial(
                        {color: 0x7777ff, wireframe: true});
    var sphere = new THREE.Mesh(sphereGeometry,sphereMaterial);

    sphere.position.x = 20;
    sphere.position.y = 4;
    sphere.position.z = 2;

    scene.add(sphere);

    camera.position.x = -30;
    camera.position.y = 40;
    camera.position.z = 30;
    camera.lookAt(scene.position);

    $("#WebGL-output").append(renderer.domElement);
    renderer.render(scene, camera);
});
```

如果在浏览器中打开这个示例，那么看到的结果就跟我们的目标有些类似，但仍然还有很长的路要走：



在开始美化之前，逐行解析一下这些代码，这样你就会明白这些代码在干什么：

```
var scene = new THREE.Scene();

var camera = new THREE.PerspectiveCamera(45
                          , window.innerWidth / window.innerHeight
                          , 0.1, 1000);

var renderer = new THREE.WebGLRenderer();
renderer.setClearColorHex(0xEEEEEE);
renderer.setSize(window.innerWidth, window.innerHeight);
```

在本示例一开始，我们定义了 scene（场景）、camera（相机）和 renderer（渲染器）。变量 scene 是一个容器，用来保存并跟踪所有我们想渲染的物体。我们想要渲染的球体和方块会在本例后面添加到这个 scene 变量中。在这个例子中我们还创建了一个 camera 变量。camera 变量定义了我们能够在渲染好的 scene 里看到什么。在第 2 章中你会学到更多可以传递给相机的参数。接下来，我们定义了一个 renderer 对象。renderer 对象负责计算指定相机角度下，浏览器中 scene 的样子。在这个示例里我们创建了一个 WebGLRenderer 对象，使用计算机上的显卡来渲染场景。

如果你看过 Three.js 的源代码和文档，你会发现除了基于 WebGL 的渲染器外，还有一些其他的渲染器。有基于画板（Canvas）的渲染器，甚至还有基于 SVG 的渲染器。尽管它们可以工作，也能渲染简单的场景，但我并不推荐，因为它们十分耗费 CPU 资源，也缺乏某些功能，例如支持复杂材质和阴影。

我们在代码里还通过调用 setClearColorHex() 函数将 renderer 的背景色设置为接近白色的颜色（0XEEEEEE），并通过 setSize() 函数来告诉 renderer 将 scene 渲染成多大尺寸。

到目前为止，我们有了一个基本上是空白的 scene、一个 renderer 和一个 camera。但是还没有要渲染的东西。下面的代码用来添加辅助的坐标轴和平面：

```
var axes = new THREE.AxisHelper( 20 );
scene.add(axes);

var planeGeometry = new THREE.PlaneGeometry(60,20);
var planeMaterial = new THREE.MeshBasicMaterial(
                                        {color: 0xcccccc});
var plane = new THREE.Mesh(planeGeometry,planeMaterial);

plane.rotation.x = -0.5*Math.PI;
plane.position.x = 15;
plane.position.y = 0;
plane.position.z = 0;
scene.add(plane);
```

可以看到，我们创建了一个 axes（坐标轴）对象，并使用 scene.add() 函数将这些坐标轴添加到我们的场景中。现在我们要创建平面。这可以在两步内完成。首先我们通过新代码 THREE.PlaneGeometry（60，20）定义平面的尺寸。在本例中该平面宽 60、高 20。我们还要告诉 Three.js 这个平面的外观（例如颜色和透明度）。在 Three.js 里，我们可以通过创建一个材质对象来达到这个目的。在本例中我们创建了一个基本的材质（使用 MeshBasicMaterial() 方法），颜色是 0xcccccc。接下来我们把这两个对象合并到一个名为 plane（平面）的 Mesh（网格）对象中。在将这个 plane 变量添加到场景之前，我们还要将它放置到合适的位置；我们先将它绕着 x 轴旋转 90 度，然后使用它的 position（位置）属性定义它在场景中的位置。如果你已经开始对 Mesh 对象的细节感兴趣了，那么你可以看一下第 2 章的示例 06-mesh-properties.html，该示例展示并解释旋转和定位。最后，我们要把这个 plane 添加到 scene，就像我们添加 axes 那样。

可以依照同样的方法添加 cube（方块）和 sphere（球体），但是要把 wireframe（线框）属性设为 true。接下来是本示例的最后部分：

```
camera.position.x = -30;
camera.position.y = 40;
camera.position.z = 30;
camera.lookAt(scene.position);

$("#WebGL-output").append(renderer.domElement);
renderer.render(scene, camera);
```

现在所有要渲染的部分都已经添加到场景中的正确位置了。我之前提到过相机定义哪些东西可以被渲染。在这段代码中，我们使用 x、y 和 z 这三个 position 属性指定相机的位置，即悬挂在场景的上方。为确保相机能够拍摄到这些物体，我们使用了 lookAt() 函数指向场景的中心。剩下要做的就是将 renderer 的输出挂接到 HTML 页面框架中的 元素；我们使用 jQuery 来选择正确的输出元素，并告诉 renderer 用我们提供的相机来渲染 scene。

在下面的几节里我们会添加灯光、使用更多的材质，甚至使用动画，让这个场景变得更加好看。

1.5　添加材质、灯光和阴影

在 Three.js 里添加新的材质和灯光非常简单，做法跟我们在前一节所解释的基本一样。我们首先在场景里添加一个光源（完整代码请参见示例 03-example-light.html）：

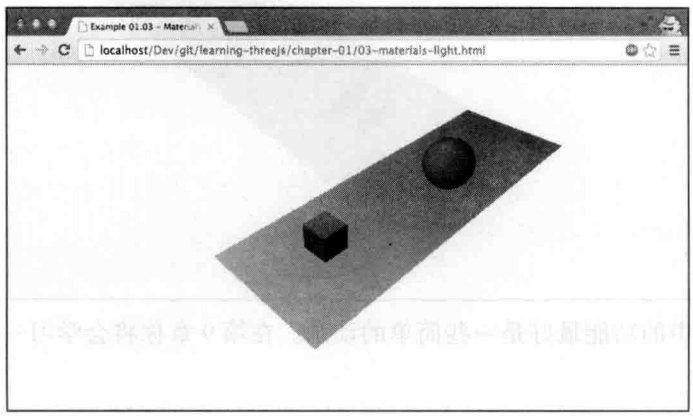
```
var spotLight = new THREE.SpotLight( 0xffffff );
spotLight.position.set( -40, 60, -10 );
scene.add( spotLight );
```

通过 SpotLight() 方法创建一个光源，并从（spotLight.position.set(–40, 60, –10)）位置处照射我们的场景。如果我们现在就渲染场景，那么结果跟之前的场景没什么区别。原因是不同的材质对光源的反应并不相同。我们在前一个示例中用的基础材质（调用 MeshBasicMaterial() 方法）不会对场景中的光源产生反应，而只会以指定的颜色渲染物体。所以我们不得不改变 plane、sphere 和 cube 的材质，如下所示：

```
var planeGeometry = new THREE.PlaneGeometry(60,20);
var planeMaterial = new THREE.MeshLambertMaterial(
                                        {color: 0xffffff});
var plane = new THREE.Mesh(planeGeometry,planeMaterial);
...
var cubeGeometry = new THREE.CubeGeometry(4,4,4);
var cubeMaterial = new THREE.MeshLambertMaterial(
                                        {color: 0xff0000});
var cube = new THREE.Mesh(cubeGeometry, cubeMaterial);
...
var sphereGeometry = new THREE.SphereGeometry(4,20,20);
var sphereMaterial = new THREE.MeshLambertMaterial(
                                        {color: 0x7777ff});
var sphere = new THREE.Mesh(sphereGeometry,sphereMaterial);
```

在这段代码里，我们将场景中各个物体的材质属性改成了 MeshLambertMaterial。Three.js 里有两种材质可以对光源产生反应：MeshLambertMaterial 和 MeshPhongMaterial。

渲染结果如下图所示，但是这仍然不是我们所追求的效果：



我们快要完成了，球体和方块看起来好多了。不过还没有添加阴影。

渲染阴影要花费大量的计算资源，因此默认情况下 Three.js 并不生成阴影。但是生成阴影还是很简单的。要生成阴影我们不得不修改几处代码，如下面的代码片段所示：

```
renderer.setClearColorHex(0xEEEEEE, 1.0);
renderer.setSize(window.innerWidth, window.innerHeight);
renderer.shadowMapEnabled = true;
```

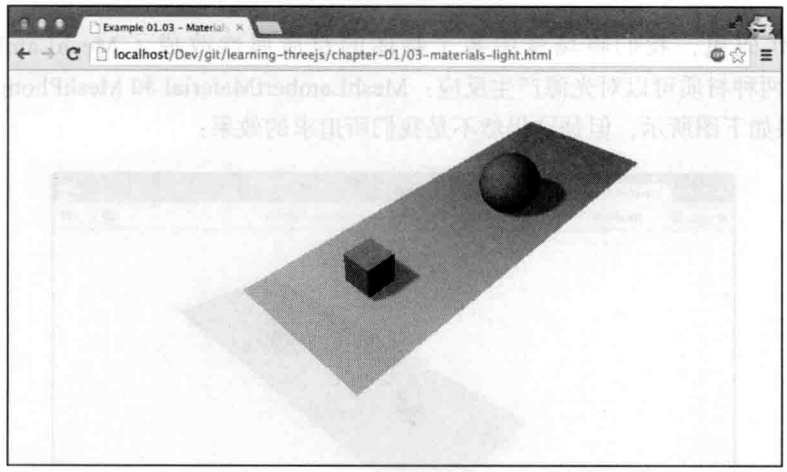
首先要改的是告诉 renderer 我们需要阴影。为此可以将 shadowMapEnabled（允许阴影映射）属性设为 true（真）。如果只是看这个修改的渲染结果，那么还不会有任何区别。原因是我们需要明确定义哪个物体投射阴影，哪个物体接受阴影。在我们的例子里，我们想让球体和方块将阴影投射到地面上。为了达到这个目的，你可以将那些物体的相应属性设为 true，如下代码所示：

```
plane.receiveShadow = true;
...
cube.castShadow = true;
...
sphere.castShadow = true;
```

现在要实现阴影效果，只剩下一件事要做。我们要定义场景中的哪个光源可以产生阴影。不是所有的光源都可以产生阴影（你将会在下一章学到更多关于这个问题的知识），在这个示例中用 SpotLight() 方法生成的光源可以产生阴影。我们只要设置正确的属性，阴影就会渲染出来：

```
spotLight.castShadow = true;
```

这样，该场景中就有了由我们的光源产生的阴影，如下图所示：



添加到场景中的功能最好是一些简单的动画。在第 9 章你将会学习一些更高级的动画选项。

1.6　用动画扩展你的首个场景

如果要在场景中加入动画，那么首先要找到一种方法，使得场景可以以一定的时间间隔进行渲染。在 HTML5 和相应的 JavaScript API（应用开发接口）出来之前，这个方法是使用 setInterval (function, interval) 方法。例如，我们可以通过 setInterval() 方法指定函数每 100 毫秒调用一次。这个函数的问题是它并不考虑浏览器中发生的事情。如果你正在浏览其他页面，这个函数仍然会每隔几毫秒就会被调一次。除此之外，setInterval() 方法并没有跟显示器的重画同步。这可能会导致较高的 CPU 使用率，降低系统效率。

1.6.1　引入 requestAnimationFrame() 方法

幸运的是，现在的浏览器已经有了解决 setInterval() 函数问题的方法，即 requestAnimationFrame() 函数。通过 requestAnimationFrame() 你可以指定一个函数，按照浏览器定义的时间间隔调用。你可以在这个指定的函数里执行所有必要的绘画操作，而浏览器则会尽可能保证绘画过程平滑、高效。使用方法非常简单（完整的源代码可以在 04-materials-light-animation.html 里找到），你只要写一个负责渲染的函数即可：

```
function renderScene() {
  requestAnimationFrame(renderScene);
  renderer.render(scene, camera);
}
```

在 renderScene() 函数里，我们又调了一次 requestAnimationFrame()，目的是让这个动画持续运行。代码中我们唯一需要改的是：在整个场景创建完毕后，调用一次 renderScene() 函数来启动动画，而不是调用 renderer.render()：

```
...
  $("#WebGL-output").append(renderer.domElement);
  renderScene();
```

如果现在运行上述代码，那么跟之前的例子相比你还不能看出差别，因为我们还没有加入任何动画。在添加动画之前，我想先引入一个小的辅助库，这个辅助库可以检测出动画运行的帧频。这个库的作者也是 Three.js 库的作者。当我们显示动画时，该库可以在一个小图片里显示每秒显示的帧数（FPS）。

要显示这个统计图形，我们首先要在 HTML 页面的 <header> 标签里引入这个库：

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

然后添加一个 元素，用来显示统计图形：

```
<div id="Stats-output"></div>
```

剩下要做的就是初始化统计对象，并添加到 元素里，如下所示：

```
function initStats() {
            var stats = new Stats();
            stats.setMode(0);
            stats.domElement.style.position = 'absolute';
            stats.domElement.style.left = '0px';
            stats.domElement.style.top = '0px';
            $("#Stats-output").append( stats.domElement );
            return stats;
        }
```

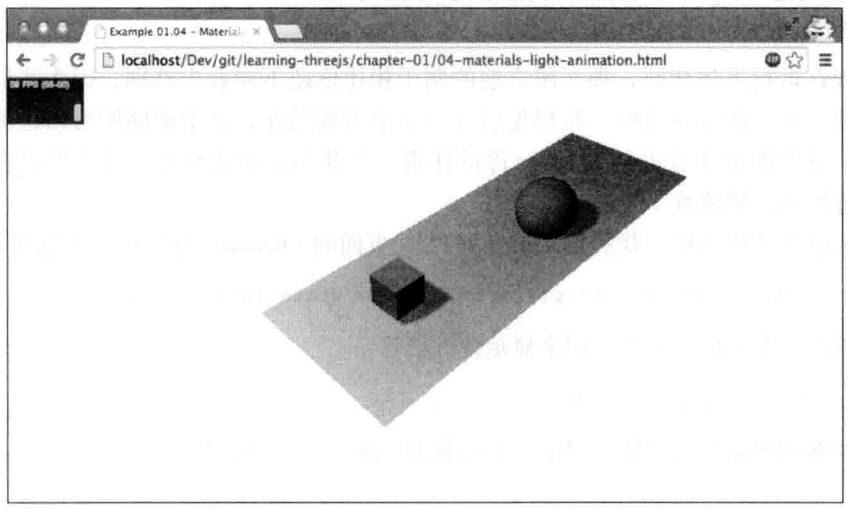
该函数初始化统计对象。值得注意的是 setMode() 函数。如果设置为 0，那么我们监测的是 FPS；如果设置为 1，那么监测的是渲染时间。对于本例，我们感兴趣的是 FPS，所以设置的是 0。在匿名函数的一开始调用该函数，使得场景拥有了统计的功能：

```
$(function () {

        var stats = initStats();
        ...
}
```

剩下的就是要告诉 stats 对象每次渲染开始的时间。为此我们可以在 render() 函数里调用 stats.update() 方法，如下所示：

```
function render() {
          stats.update();
          ...
          requestAnimationFrame(render);
          renderer.render(scene, camera);
      }
```

添加上述代码之后再次运行，你就会在左上角看到一个统计图形，如下图所示：



1.6.2　转动方块

引入了 requestAnimationFrame() 方法并配置好统计对象之后，我们就可以开始写控制动画的代码了。本节我们要扩展 render() 函数：绕坐标轴旋转红色的方块。代码如下：

```
function render() {
            ...
            cube.rotation.x += 0.02;
            cube.rotation.y += 0.02;
            cube.rotation.z += 0.02;
            ...
            requestAnimationFrame(render);
            renderer.render(scene, camera);
        }
```

看起来很简单，是不是？我们做的只是在每次调用 render() 函数时，在每个坐标轴的 rotation 属性上增加 0.02，其效果就是使得这个方块绕所有的轴旋转。让那个蓝色的球体反弹起来并不难。

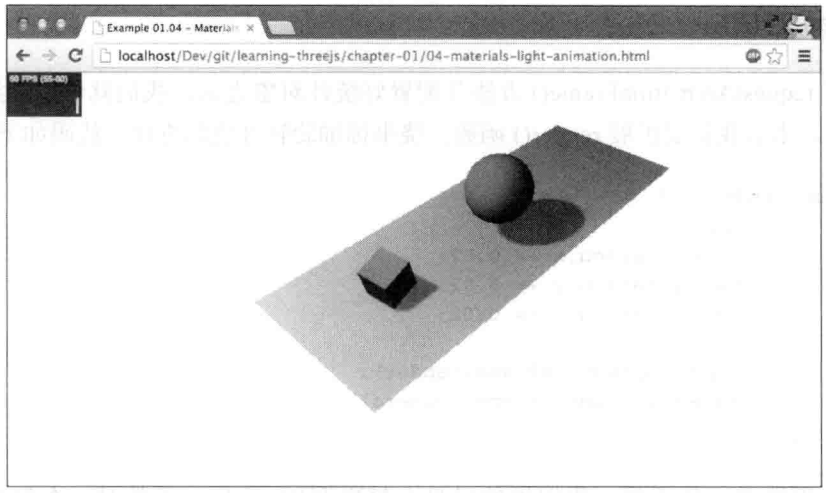
1.6.3　弹跳球

同样，要让球体跳起来只要在 render() 函数里加几行代码就可以了：

```
var step=0;
function render() {
            ...
            step+=0.04;
            sphere.position.x = 20+( 10*(Math.cos(step)));
            sphere.position.y = 2 +( 10*Math.abs(Math.sin(step)));
            ...
            requestAnimationFrame(render);
            renderer.render(scene, camera);
        }
```

对于方块，我们修改的是 rotation 属性；对于球体，我们要改的是它的 position 属性。我们的目的是让球体依照一条好看的、光滑的曲线，从一个地方跳到另一个地方。

为此，我们要修改它在 x 轴上的 position（位置）和 y 轴上的 position（位置）。使用 step 变量并在函数 Math.cos() 和 Math.sin() 的帮助下，我们可以创建出一条光滑的轨迹。我不想在这里详细解释它是如何工作的。现在你只要知道 step ＋＝ 0.4 定义的是球的弹跳速度即可。在第 8 章我会讲述这些函数控制动画的细节，我会解释所有事情。下图展示的就是带有动画的场景：



在结束本章之前，我想在这个基础的场景上再添加几个元素。在创建类似的三维场景、动画、颜色和属性时，通常需要尝试几次才能找到合适的颜色和速度。如果有一个简单的GUI（图形界面）让我们可以在运行时修改这些东西，那么事情就会简单很多。幸运的是，的确有这样的工具。

1.7 使用 dat.GUI 库简化试验

Google 的几个人创建了一个名为 dat.GUI 的库（查找相关文档请访问 http://code.google.com/p/dat-gui/），在它的帮助下，你很容易就可以创建出一个简单的界面组件，用以修改代码中的变量。我们将在本节使用 dat.GUI 库在示例中添加一个用户界面，使得我们可以：

- 控制球体的弹跳速度
- 控制方块的旋转

就像添加统计对象一样，我们首先要在 HTML 页面的 <header> 元素里添加这个库，代码如下所示：

```
<script type="text/javascript" src="../libs/dat.gui.js"></script>
```

接下来我们要定义一个 JavaScript 对象，用来保存我们想要通过 dat.GUI 库修改的那些变量。在 JavaScript 代码的主体里添加如下 JavaScript 对象：

```
var controls = new function() {
            this.rotationSpeed = 0.02;
            this.bouncingSpeed = 0.03;
        }
```

在这个 JavaScript 对象里，我们会定义两个属性（this.rotationSpeed 和 this.bouncingSpeed）以及它们的默认值。接下来我们会把这个对象传递给 dat.GUI 对象，并定义这两个属性的取

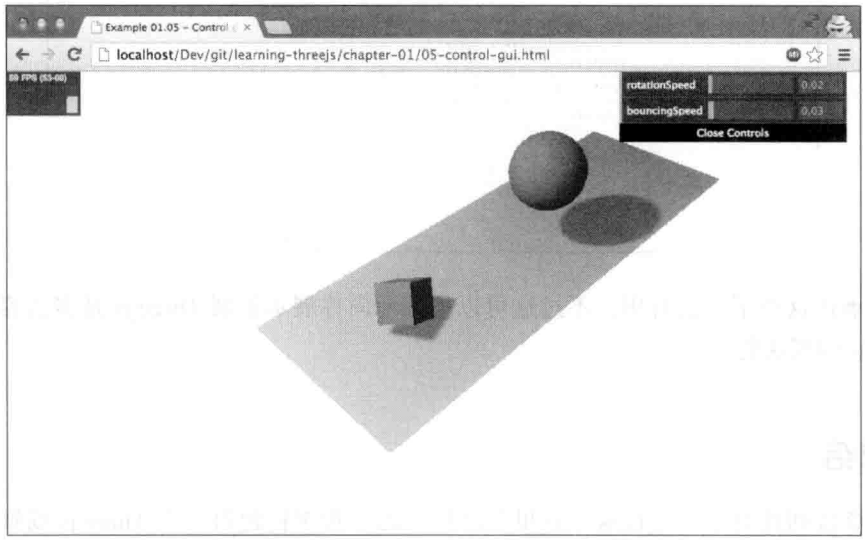
值范围，如下所示：

```
var gui = new dat.GUI();
gui.add(controls, 'rotationSpeed',0,0.5);
gui.add(controls, 'bouncingSpeed',0,0.5);
```

属性 rotationSpeed 和 bouncingSpeed 的取值范围都是从 0 到 0.5。现在要做的是保证在 render 循环里直接引用这两个属性。这样当我们在 dat.GUI 用户界面里修改时可以直接影响物体的旋转速度和弹跳速度。如下所示：

```
function render() {
            ...
          cube.rotation.x += controls.rotationSpeed;
             cube.rotation.y += controls.rotationSpeed;
             cube.rotation.z += controls.rotationSpeed;
    step+=controls.bouncingSpeed;
    sphere.position.x = 20+( 10*(Math.cos(step)));
    sphere.position.y = 2 +( 10*Math.abs(Math.sin(step)));
    ...
}
```

现在运行这个示例（05-control-gui.html）时，你就会看到一个简单的用户界面，通过它可以控制物体的弹跳和旋转速度：



1.8　使用 ASCII 效果

本章我们都在使用先进浏览器的功能来创建好看的三维渲染效果。Three.js 还有几个有意思的功能，可以用来改变渲染的效果。在结束本章之前，我想介绍其中一种效果：

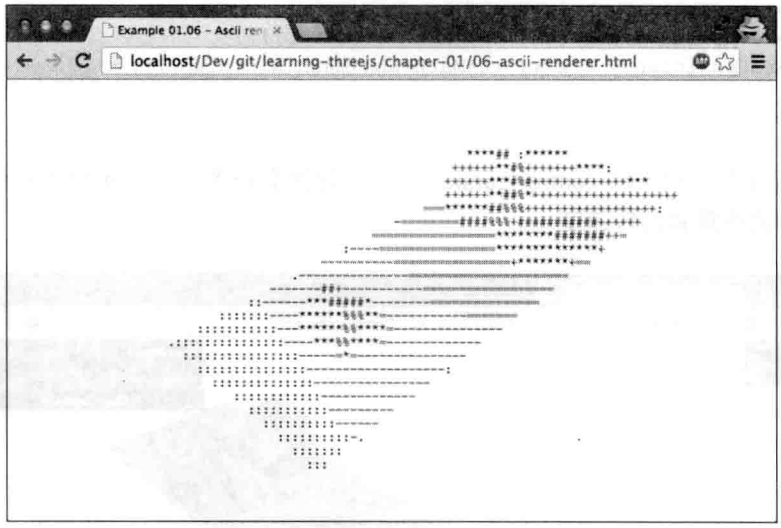
ASCII 效果（文本画）。使用 ASCII 效果，你可以把漂亮的动画场景变成一个复古的、文本画风格的动画，而这只需要几行代码。你只要将 JavaScript 代码主体部分的最后几行从：

```
$("#WebGL-output").append(renderer.domElement);
```

改成：

```
var effect = new THREE.AsciiEffect( renderer );
effect.setSize( window.innerWidth, window.innerHeight );
$("#WebGL-output").append(effect.domElement);
```

还要稍微改一下 render 循环。将 renderer.render（scene，camera）替换成 effect.render（scene，camera）。应用该效果之后，输出结果如下图所示：



必须承认这并不怎么有用，不过这可以很好地向你展示扩展 Three.js 是多么容易，而这都源自于它的模块化。

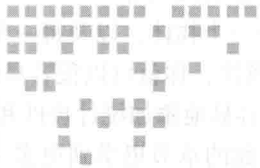
1.9 总结

第 1 章就到此为止了。在这一章里你已经学习了很多构建每一个 Three.js 场景的基础概念，这为我们学习下一章开了一个好头。本章中应当记住的内容如下所示：

- 你可以从网上找到本章和其他章示例的源代码。最好的学习方法就是摆弄一下这些示例，做些试验。
- 在这个 Three.js 项目里，你创建了一个场景，添加了一些需要渲染的物体（带有材质的图形）。

- 你所用的材质定义了物体的样子。每种材质对光源的反应都不一样。
- 渲染阴影的开销非常大，并且需要在渲染器、每一个物体，以及每一个光源上打开。
- 通过修改场景中物体的 position 属性和 rotation 属性，你就可以很容易地做出动画。
- 使用两个辅助库和几行 JavaScript 代码，可以很容易地添加统计控件和定制控件。

下一章我们将会进一步扩展这个示例。你将会在后面的章节里学到更多 Three.js 里非常重要的构建场景的模块。



Chapter 2 第 2 章

使用构建 Three.js 场景的基本组件

前一章你学习了一些 Three.js 库的基础知识。我们看了几个示例，你也创建了第一个完整的 Three.js 场景。本章我们将稍微深入一下 Three.js 库，讲解一些构成 Three.js 场景的基础组件。通过本章你将探索以下主题：

- Three.js 场景中使用哪些组件
- 你能用 THREE.Scene() 对象做什么
- 几何图形和网格是如何关联的
- 正投影相机和透视相机的区别

我们首先来看一看如何创建场景并添加物体。

2.1 创建场景

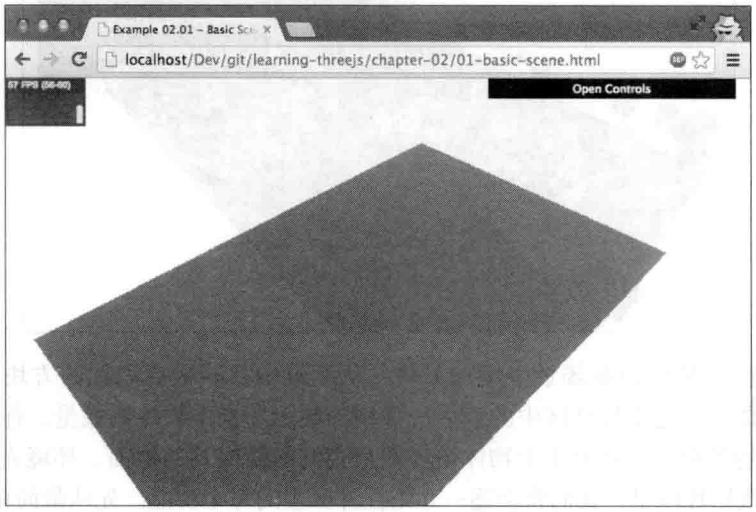
在前一章你已经创建了一个 THREE.Scene() 对象，所以你已经了解了 Three.js 库的基础。我们已经看到，一个场景要想显示任何东西需要三种类型的组件：

组　件	描　述
相机	决定哪些东西将要在屏幕上渲染
光源	它们会对材质如何显示，以及生成阴影时材质如何使用（细节部分在第 3 章讨论）产生影响
物体	它们是在相机透视图里主要的渲染对象：方块、球体等

THREE.Scene() 对象就像是所有这些对象的容器。这个对象本身并没有太多可选项和函数。

2.1.1　场景的基本功能

探索场景功能的最好方式就是看示例。在本章（chapter-02）的源代码里你可以找到这个示例，01-basic-scene.html。我会用这个示例来解释一个场景所拥有的各种函数和选项。当我们在浏览器里打开这个示例时，其输出差不多如下图所示：

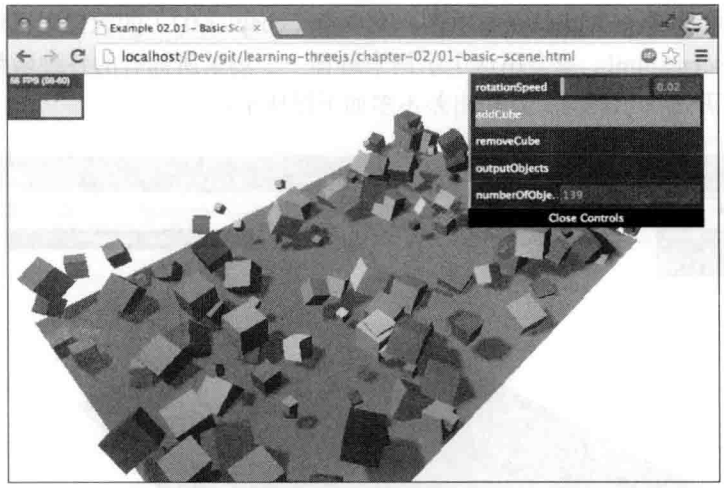


这跟我们在前一章看到的那个例子非常像。尽管这个场景看上去有点儿空荡荡，但它已经包含了几个对象。通过下面的源代码我们可以看到：已经用 THREE.Scene() 对象的 Scene.add（object）函数添加了一个 THREE.mesh 对象（plane，你看到的那个地面）、一个 THREE.SpotLight（聚光灯光源）对象和一个 THREE.AmbientLight（环境光）对象。场景渲染的时候 THREE.Camera 对象会自动添加进来，但是如果你喜欢，也可以手工添加。

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(45,
window.innerWidth / window.innerHeight, 0.1, 1000);
...
var planeGeometry = new THREE.PlaneGeometry(60,40,1,1);
var planeMaterial = new THREE.MeshLambertMaterial({color: 0xffffff});
var plane = new THREE.Mesh(planeGeometry,planeMaterial);
...
scene.add(plane);
var ambientLight = new THREE.AmbientLight(0x0c0c0c);
scene.add(ambientLight);
...
var spotLight = new THREE.SpotLight( 0xffffff );
...
scene.add( spotLight );
```

在我们深入讲解 THREE.Scene() 对象之前，我要首先说明一下你可以在这个示例中做些什么，然后我们再来看一些代码。在浏览器中打开这个示例，看一下右上角的那些控件。如

下图所示：



使用这些控件你可以在场景中添加方块、从场景中移除最后添加的方块，以及显示场景中的所有物体。在这个控件区中的最后一项用于显示场景中物体的数量。你可能会发现场景启动的时候场景中就已经有 4 个物体了。就是我们前面提到的地面、环境光、点光源和相机。在下面的代码片段里，我们将会逐一介绍控件区里的每个功能。先从最简单的 addCube() 函数开始：

```
this.addCube = function() {

  var cubeSize = Math.ceil((Math.random() * 3));
  var cubeGeometry = new
        THREE.CubeGeometry(cubeSize,cubeSize,cubeSize);
  var cubeMaterial = new THREE.MeshLambertMaterial(
        {color: Math.random() * 0xffffff });
  var cube = new THREE.Mesh(cubeGeometry, cubeMaterial);
  cube.castShadow = true;
  cube.name = "cube-" + scene.children.length;
  cube.position.x=-30 + Math.round(
            (Math.random() * planeGeometry.width));
  cube.position.y= Math.round((Math.random() * 5));
  cube.position.z=-20 + Math.round((Math.random() *
              planeGeometry.height));

  scene.add(cube);
  this.numberOfObjects = scene.children.length;
};
```

现在这段代码应该很容易读懂。这里也没有引入很多新概念。当你点击 addCube 按钮时，一个新的 THREE.CubeGeometry 实例就会创建出来，其尺寸是 0 到 3 之间的一个随机数。除了随机大小尺寸之外，这个方块在场景中的颜色和位置也是随机的。

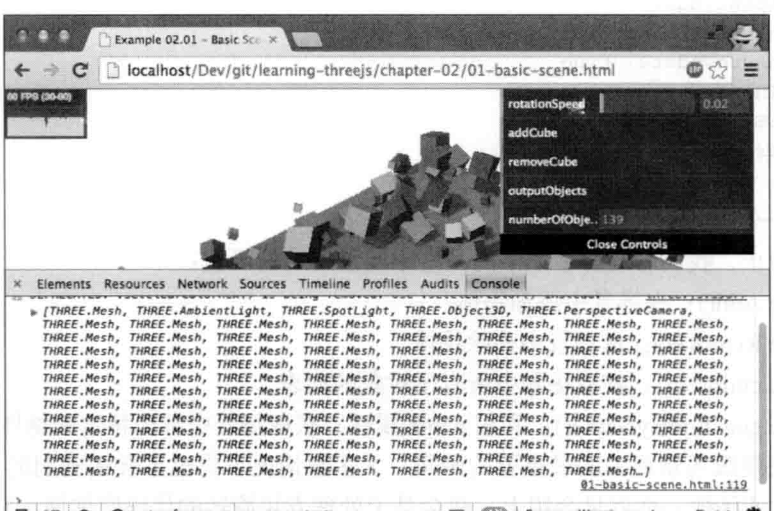
这段代码里的一个新东西是我们使用 name 属性为这个方块指定了一个名字。方块的名字是在 cube- 后面加上当前场景中对象的数量（即显示的 scene.children.length 属性）。所以你将会得到诸如 cube-1、cube-2、cube-3 等类似的名字。名字在调试的时候很有用，但是也可以直接用来在场景中查找对象。如果使用 Scene.getChildByName（name）函数，你可以直接获取指定的对象，然后可以执行一些操作，例如改变它的位置。你或许想知道前面那段代码的最后一行的目的是什么。我们的控制界面使用 numberOfObjects 变量显示场景中的对象数量。所以无论什么时候添加或删除对象，我们都要将这个变量设置为更新后的数量。

我们可以在控制界面上调用的下一个函数是 removeCube。顾名思义，点击这个按钮就可以把最后添加的方块从场景中移除。下面的代码片段即该函数的定义：

```
this.removeCube = function() {
  var allChildren = scene.children;
  var lastObject = allChildren[allChildren.length-1];
  if (lastObject instanceof THREE.Mesh) {
      scene.remove(lastObject);
      this.numberOfObjects = scene.children.length;
  }
}
```

在场景中添加对象我们使用 add() 函数，而从场景中移除一个对象，我们就要使用 remove() 函数。在这段代码里我们通过 THREE.Scene() 对象的 children 属性获取最后添加的对象。我们还需要检查一下这个对象是不是一个 Mesh 对象，以防移除相机和光源。删掉这个对象之后，我们将再一次更新控制界面上那个表示场景中对象数量的属性。

控制界面上的最后一个按钮的标签是 outputObjects。你大概已经点过它了，而且什么也没有发生。这个按钮的功能是打印出场景中的所有对象，并将结果输出到网络浏览器的 Console（控制台），如下图所示：



往 Console 输出信息用的是内置的 console 对象，如下所示：

```
this.outputObjects = function() {
  console.log(scene.children);
}
```

这对调试来讲非常有利；特别是当你为物体命名之后，这对查找场景中跟某个特定物体相关的问题或错误非常有用。例如，物体 cube-17 的属性输出结果如下所示：

```
__webglActive: true
__webglInit: true
_modelViewMatrix: THREE.Matrix4
_normalMatrix: THREE.Matrix3
_vector: THREE.Vector3
castShadow: true
children: Array[0]
eulerOrder: "XYZ"
frustumCulled: true
geometry: THREE.CubeGeometry
id: 20
material: THREE.MeshLambertMaterial
matrix: THREE.Matrix4
matrixAutoUpdate: true
matrixRotationWorld: THREE.Matrix4
matrixWorld: THREE.Matrix4
matrixWorldNeedsUpdate: false
name: "cube-17"
parent: THREE.Scene
position: THREE.Vector3
properties: Object
quaternion: THREE.Quaternion
receiveShadow: false
renderDepth: null
rotation: THREE.Vector3
rotationAutoUpdate: true
scale: THREE.Vector3
up: THREE.Vector3
useQuaternion: false
visible: true
__proto__: Object
```

到目前为止，我们已经看了如下的一些跟场景相关的函数：

- Scene.Add()：在场景中添加物体
- Scene.Remove()：从场景中移除物体
- Scene.children()：获取场景中所有子对象的列表
- Scene.getChildByName()：利用 name 属性，获取场景中某个特定的物体

这些函数是跟场景相关的最重要的函数，大多数情况下你也不会用到别的函数。但是还有几个辅助性的函数，你迟早会用上。我会基于处理方块旋转的代码向你展示它们。

正如你在第 1 章所看到的，我们使用了一个 render 循环来渲染场景。让我们来看一段代码，它可以实现同样的功能：

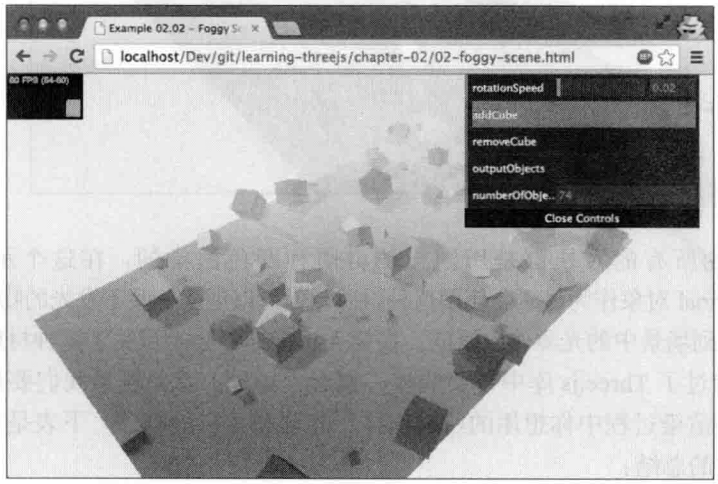
```
function render() {
  stats.update();
    scene.traverse(function(e) {
      if (e instanceof THREE.Mesh && e != plane ) {
        e.rotation.x+=controls.rotationSpeed;
        e.rotation.y+=controls.rotationSpeed;
        e.rotation.z+=controls.rotationSpeed;
      }
    });

  requestAnimationFrame(render);
  renderer.render(scene, camera);
}
```

我们可以看到这里用到了 THREE.Scene.traverse() 函数。我们可以将一个函数作为参数传递给 traverser() 函数。这个传递来的函数将会在场景的每一个子对象上调用一次。在 render() 函数里，我们将使用 traverse() 函数类更新每个方块的旋转弧度（我们特意忽略了表示地面的 plane 对象）。我们也可以使用 for 循环遍历 children 这个属性数组来达到同样的目的。

在我们深入 Mesh（网格）和 Geometry（几何体）对象的细节之前，我想向你展示一下可以在 Scene（场景）对象上设置的两个有趣属性：fog（雾化）和 overrideMaterial（材质覆盖）。

2.1.2　在场景中添加雾化效果

通过 fog 属性可以为整个场景添加一种雾化效果。一个物体离得越远，就越模糊。下面这个屏幕截图展示的就是打开雾化效果的样子：



在 Three.js 库里打开雾化效果很简单。只要在定义完场景后加上如下一行代码即可：

```
scene.fog=new THREE.Fog( 0xffffff, 0.015, 100 );
```

我们在这里定义了一个白色的雾化效果（0xffffff）。后面的两个属性用来调节雾的显示。0.015 是 near（近处）属性的值，而 100 设置的是 far（远处）属性的值。通过这两个属性你可以决定雾从什么地方开始，以及浓度加深的程度。还有另外一个方法可以设置场景中雾的浓度；你可以使用如下的定义：

```
scene.fog=new THREE.FogExp2( 0xffffff, 0.015 );
```

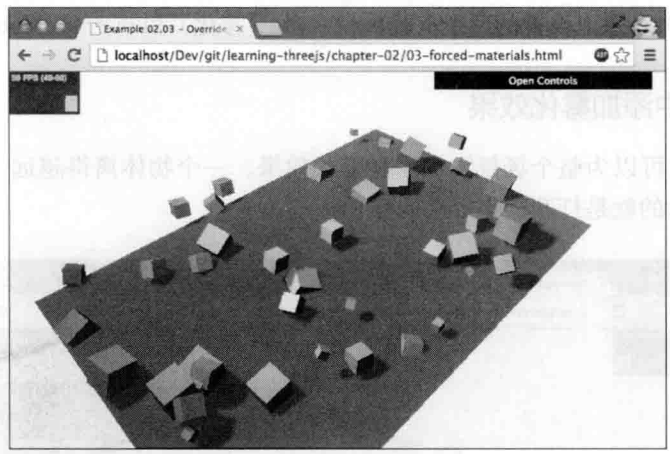
这次我们不指定 near 属性和 far 属性，只给出颜色和浓度。要想获得理想的效果，你最好亲手试验一下这些属性。

2.1.3 使用材质覆盖属性

我们要探讨的属性是场景的 overrideMaterial 属性，该属性用来设置所有物体的材质。当你像下面这段代码那样使用这个属性时，所有添加到场景中的物体都会使用同样的材质：

```
scene.overrideMaterial = new
      THREE.MeshLambertMaterial({color: 0xffffff});
```

场景的渲染结果如下图所示：



你可以看到所有的方块都是用同样的材质和颜色渲染的。在这个示例中，我们用 MeshLambertMaterial 对象作为材质。使用这种材质，你可以创建一些不发光的物体，而且这些物体可以对你添加到场景中的光源作出反应。在第 4 章中你将会学习关于这种材质更多的内容。

本节我们看过了 Three.js 库中首要的核心概念：场景。有关场景我们要记住的最重要的是：它基本上是渲染过程中你想用的所有物体、光源和相机的容器。下表是对场景对象最重要的函数和属性的总结：

函数/属性	描　述
add（object）	在场景中添加对象。你可以像我们后面看到的那样，创建对象组
children	返回一个场景中所有对象的列表，包括相机和光源
getChildByName（name）	创建对象时，可以通过 name 属性为它指定一个独一无二的名字。场景对象提供了一个方法，你可以使用该方法根据名字直接返回这个对象
remove（object）	如果你在场景中引用了一个对象，那么你也可以用这个函数从场景中删除它
traverse（funcion）	children 属性返回场景中所有子对象的列表。通过 traverse() 函数，我们同样可以传入一个回调函数访问这些子对象
fog	通过该属性可以设置场景的雾化效果。它可以渲染出一层雾气，隐藏远处的物体
overrideMaterial	通过这个属性，你可以强制场景中的所有物体都使用相同的材质

下一节会详细介绍那些可以添加到场景中的对象。

2.2　使用几何和网格对象

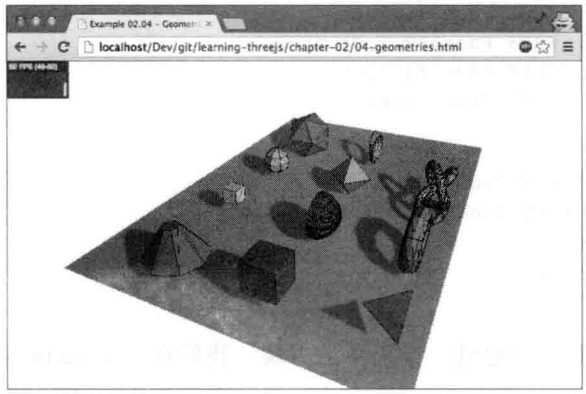
迄今为止，在前面的每个示例里你都可以看到我们使用了几何对象和网格对象。例如，要在场景中添加一个球体，我们是这么做的：

```
var sphereGeometry = new THREE.SphereGeometry(4,20,20);
var sphereMaterial = new THREE.MeshBasicMaterial({color: 0x7777ff);
var sphere = new THREE.Mesh(sphereGeometry,sphereMaterial);
```

我们定义了该对象的形状、几何结构、外观、材质，并把所有这些跟一个可以添加到场景中的网格对象结合在一起。本节将进一步介绍几何对象与网格对象。我们将从几何对象开始。

2.2.1　几何对象的属性和函数

Three.js 库附带了很多现成的几何体，你可以在三维场景中使用它们。只要加上材质、创建一个 mesh 变量，基本上就算完成了。下面是示例 04-geometries.html 的屏幕截图，其中展示的是 Three.js 库中几个标准的几何体：



我们将在第 5 章探索一下 Three.js 库中提供的所有基础的和高级的几何体对象。现在，我们将详细讲解一下什么是 geometry 变量。

Three.js 库中的 geometry 和其他大多数三维库中的一样，基本上是三维空间中的点集，以及一些将这些点连接起来的面。举例来说，一个方块：

- 一个方块有 8 个角。每个角都可以定义为 x、y 和 z 坐标的一个组合。所以每个方块都是三维空间中的 8 个点。在 Three.js 库中，这些点称为顶点（vertice）。
- 一个方块有 6 个侧面，每个角有一个顶点。在 Three.js 库里，每个侧面称为面（face）。

当你使用 Three.js 库提供的这些几何体时，你不必亲自定义所有的这些顶点和面。对于一个方块来讲，你只要定义长宽高即可。Three.js 库会利用这些信息，在正确的位置创建一个拥有 8 个顶点的几何体，并用正确的面连接起来。尽管你可以使用 Three.js 库提供的几何体，或者自动生成，但是你仍然可以通过定义顶点和面，手工创建几何体。创建方法可以参考下面的代码片段：

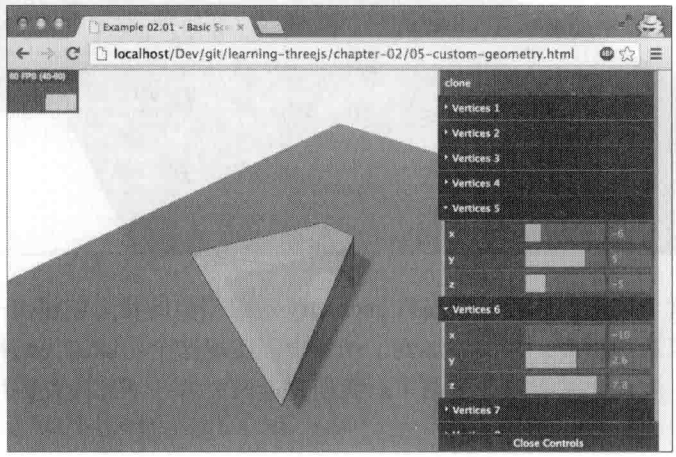
```
var vertices = [
            new THREE.Vector3(1,3,1),
            new THREE.Vector3(1,3,-1),
            new THREE.Vector3(1,-1,1),
            new THREE.Vector3(1,-1,-1),
            new THREE.Vector3(-1,3,-1),
            new THREE.Vector3(-1,3,1),
            new THREE.Vector3(-1,-1,-1),
            new THREE.Vector3(-1,-1,1)
        ];

var faces = [
            new THREE.Face3(0,2,1),
            new THREE.Face3(2,3,1),
            new THREE.Face3(4,6,5),
            new THREE.Face3(6,7,5),
            new THREE.Face3(4,5,1),
            new THREE.Face3(5,0,1),
            new THREE.Face3(7,6,2),
            new THREE.Face3(6,3,2),
            new THREE.Face3(5,7,0),
            new THREE.Face3(7,2,0),
            new THREE.Face3(1,3,4),
            new THREE.Face3(3,6,4),
        ];

var geom = new THREE.Geometry();
geom.vertices = vertices;
geom.faces = faces;
geom.computeCentroids();
geom.mergeVertices();
```

这段代码展示的是如何创建一个简单的方块。我们在一个 vertices 数组里定义了构成这

个方块的点。将这些点连接起来，创建三角面片，并保存在 faces 数组里。例如，元素 new THREE.Face3（0，2，1）就是用 vertices 数组里的点 0、2 和 1 创建的一个三角面片。

在这个示例中，我们用 THREE.Face3 元素来定义方块的 6 个侧面，也就是说一个面由两个三角面片构成。在 Three.js 库以前的版本里，你也可以用四边形而不是三角形。定义一个侧面时四边形使用 4 个顶点而不是 3 个。在三维建模领域里，到底是使用四边形好还是用三角形好存在比较大的争论。基本上，在建模的时候使用四边形是大家经常采用的方式，因为它比三角形更容易增强和平滑。但是对于渲染和游戏引擎来讲，使用三角形更容易，因为任意一个形状都可以渲染成多个三角形。

我们现在可以用顶点和面来构建自己的几何体，并用它来创建网格。我已经创建了一个例子，你可以尝试改一下顶点的位置。在示例 05-custom-geometry.html 里，你可以修改这个方块的所有顶点。如下图所示：



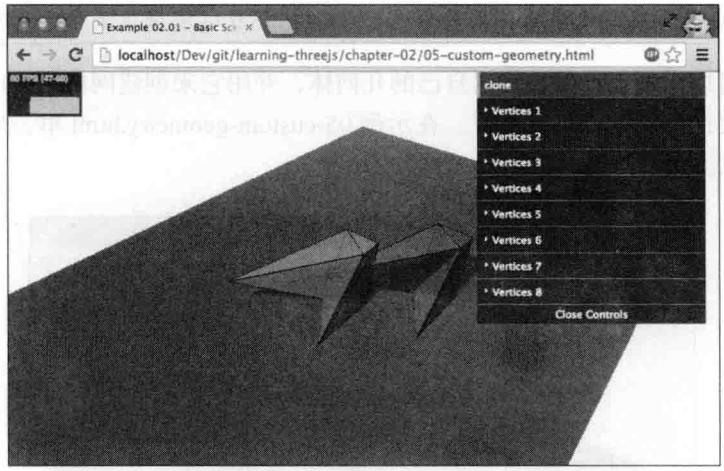
这个示例的框架跟我们其他示例的框架一样，都有一个 render 循环。对于一个顶点，无论何时修改了该顶点下拉列表框里的属性，这个方块都会按照修改后的位置正确地渲染出来。但这需要一些额外的设置。出于效率方面的考虑，Three.js 库假设一个网格的几何体在其生命周期内不会改变。为了使我们的示例能够工作，我们需要将下面的代码添加到 render 循环里：

```
mesh.geometry.vertices=vertices;
mesh.geometry.verticesNeedUpdate=true;
mesh.geometry.computeFaceNormals();
```

在上述这段代码的第一行里，我们将界面上这个 mesh 的 vertices 属性指向一个更新后的顶点数组。我们不需要重新配置侧面，因为它们跟以前一样仍然连接到那些点。设置好这些更新的顶点之后，我们需要告诉 geometry 对象，这些顶点需要更新。为此我们可以将 geometry 的 verticesNeedUpdate 属性设为 true。最后我们需要通过调用 computeFaceNormals()

函数重新计算侧面，从而完成整个模型的更新。

我们将要介绍的最后一个 geometry 函数是 clone() 函数（克隆、复制函数）。之前我们说过 geometry 对象定义的是物体的形式、形状，赋予一定材质之后，我们就可以创建出一个可以添加到场景中并由 Three.js 库渲染的物体。从名字我们就可以猜出，通过 clone() 函数我们可以创建出一个 geometry 对象的副本。赋予不同的材质后，我们就可以用这个副本创建出一个不同的网格对象。在同一个示例里（05-custome-geometry.html）你可以在控制界面的顶部找到一个 **clone** 按钮，如下图所示：



如果你点击这个按钮，你就可以按照 geometry 对象当前的状态创建出一个副本，而且这个新对象被赋予了新的材质，并被添加到了场景中。实现这个功能的代码相当简单，但由于我之前用的材质，实现起来有点儿复杂了。我们先退回去看一下那段我们用来为方块创建绿色材质的代码：

```
var materials = [
  new THREE.MeshLambertMaterial( { opacity:0.6,
                                   color: 0x44ff44,
                                   transparent:true } ),
  new THREE.MeshBasicMaterial( { color: 0x000000,
                                 wireframe: true } )
];
```

正如你所看到的，我使用的不是一个单一的材质，而是有两个元素的材质数组。原因是除了显示一个绿色透明的方块之外，我还想显示一个线框。因为使用线框的话，更容易找出顶点和面的位置。当然，在创建网格的时候，Three.js 库也支持使用多种材质。你可以使用 SceneUtils.createMultiMaterialObject() 函数来达到这个目的，如下所示：

```
var mesh = THREE.SceneUtils.createMultiMaterialObject(
                geom,materials);
```

在这个函数里，Three.js 库创建的不是一个 THREE.Mesh 实例，而是为每个你指定的材质创建一个实例，并把这些实例存放在一个组里。你可以像使用 Scene 对象那样使用这个组。你可以添加网格、通过名字查找网格等。例如，要为组中所有子对象添加阴影，我们可以这么做：

```
mesh.children.forEach(function(e) {e.castShadow=true});
```

现在回到我们之前讨论的 clone() 函数：

```
this.clone = function() {

 var cloned = mesh.children[0].geometry.clone();
 var materials = [
    new THREE.MeshLambertMaterial( { opacity:0.6,
                                     color: 0xff44ff,
                                     transparent:true } ),
    new THREE.MeshBasicMaterial({ color: 0x000000,
                                  wireframe: true } )
              ];

  var mesh2 =
     THREE.SceneUtils.createMultiMaterialObject(cloned,materials);
  mesh2.children.forEach(function(e) {e.castShadow=true});
  mesh2.translateX(5);
  mesh2.translateZ(5);
  mesh2.name="clone";
  scene.remove(scene.getChildByName("clone"));
  scene.add(mesh2);
}
```

这段 JavaScript 代码会在点击 **clone** 按钮时调用。这里我们复制的是方块的第一个子对象，geometry。请记住，mesh 变量有两个子对象：一个 mesh 用的是 MeshLambertMaterial 材质，另一个 mesh 用的是 MeshBasicMaterial 材质。基于这个复制出来的几何体，我们将创建一个新的网格，并命名为 mesh2。我们可以使用 translate() 函数移动（更多内容请参见第 5 章）这个新建的网格，删除之前的副本（如果可见的话），并把这个副本添加到场景中。

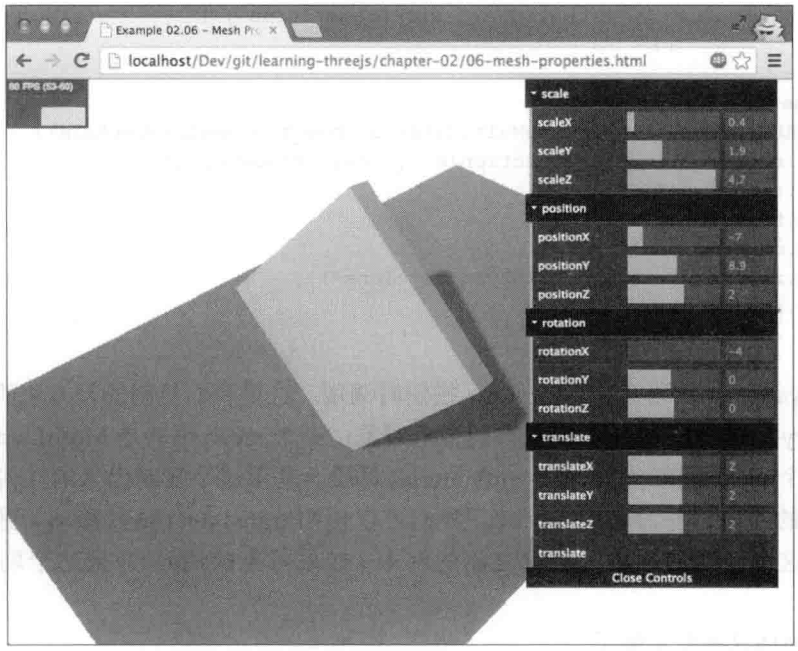
有关几何体这些就足够了。

2.2.2　网格对象的函数和属性

我们已经知道，创建一个网格需要一个几何体，以及一个或多个材质。网格创建好之后，我们就可以把它添加到场景中，它就可以被渲染了。你可以利用网格的几个属性来改变它在场景中的位置，及其显示效果。在下面的示例里，我们会看一下如下的属性和函数：

函数/属性	描　述
position（位置）	决定该对象相对其父对象的位置。多数情况下，一个对象的父对象是 THREE.Scene() 对象
rotation（旋转）	通过这个属性你可以设置对象绕任何一个轴的旋转弧度
scale（比例）	通过这个属性你可以沿着 x、y 和 z 轴缩放对象
translateX（amount）(X 轴平移)	沿 x 轴将对象平移指定的距离
translateY（amount）(Y 轴平移)	沿 y 轴将对象平移指定的距离
translateZ（amount）(Z 轴平移)	沿 z 轴将对象平移指定的距离

照例，我们给你准备好了一个例子，你可以改一下这些属性。如果你在浏览器中打开例子 06-mesh-properties.html，你会看到一个下拉菜单，通过这个菜单你可以修改这些属性，而且立即就可以看到修改的结果，如下图所示：



我会为你逐一讲解这些属性和函数；先从 position 属性开始。我们已经看到过这个属性好几次了，所以我们会讲快一点儿。通过这个属性你可以设置对象的 x、y 和 z 轴坐标。一个对象的位置是相对于其父对象讲的，而父对象则通常是你向其中添加对象的那个场景。我们将在第 5 章讨论对象组合时再来讨论这个问题。我们可以用三种方式设置对象的位置；直接设置坐标，如下所示：

```
cube.position.x=10;
cube.position.y=3;
```

```
cube.position.z=1;
```

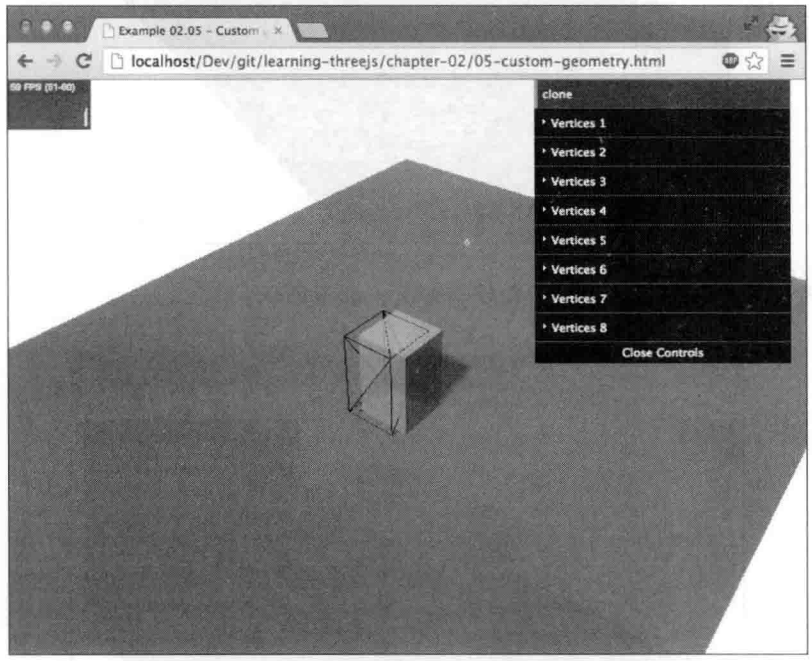
但我们也可以一次性地设置它们：

```
cube.position.set(10,3,1);
```

还有第三种方法。position 属性是一个 THREE.Vector3 对象。这意味着我们可以像下面这样设置对象：

```
cube.postion=new THREE.Vector3(10,3,1)
```

在看网格的其他属性之前，我想先稍微额外提一下。之前我说过设置对象的位置时是相对于其父对象的位置设置的。在前一节讲述 THREE.Geometry 对象时，我们用 THREE.SceneUtils.createMultiMaterialObject 函数创建了一个多材质对象。我也解释过这个函数返回的不是一个单独的网格对象，而是一个对象组，其中每个网格的几何体都是一样的，但材质不一样。在本例里，它包含两个网格。如果我们改变其中一个网格的位置，你会清楚地看到两个独立的对象。但是，如果我们移动这个对象组，那么它们的偏移量就是一样的。如下图所示：

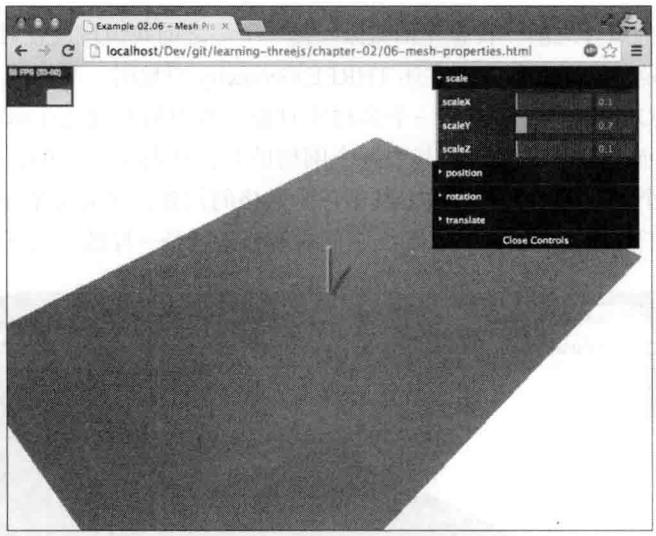


我们将在第 8 章中深入讨论这种父子关系，以及对象组是如何影响变换（例如缩放、旋转和平移）的。好了，这个列表中的下一项是 rotation（旋转）属性。在本章和前一章你已经看到这个属性被用了好几次。通过这个属性，你可以设置对象绕轴旋转的弧度。你可以像我们设置 position 属性那样设置这个属性的值。你可能还记得数学课上讲过的，旋转一周的弧度是 PI 的两倍。下面的代码片段展示的是如何设置旋转：

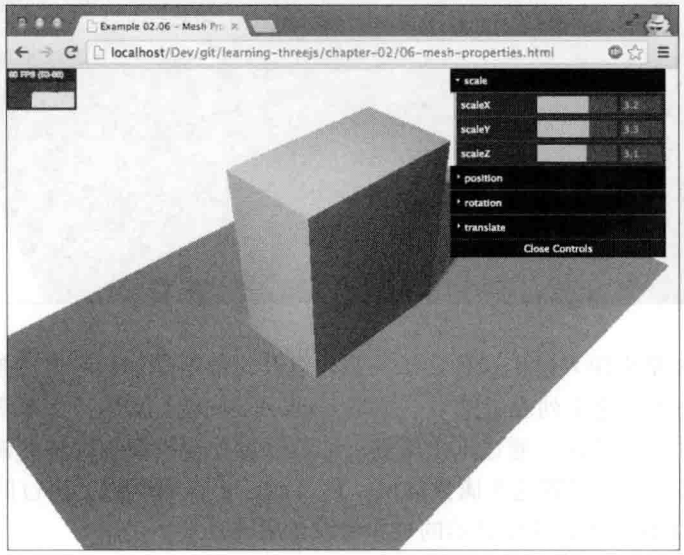
```
cube.rotation.x=0.5*Math.PI;
cube.rotation.set(0.5*Math.PI,0,0);
cube.rotation = new THREE.Vector3(0.5*Math.PI,0,0);
```

你可以在示例 06-mesh-properties.html 中尝试修改这个属性。

列表中的下一项是我们还没有讨论过的 scale（缩放）。这个名字已经很好地说明你可以用它来做什么了。你可以沿着指定轴缩放对象。如果你设的缩放值小于 1，这个对象就会缩小，如下所示：



如果这个值大于 1，那么这个对象就会变大，如下所示：



我们在本章将要讨论网格的最后部分是 translate 函数。使用 translate 函数你也可以改变对象的位置，但不是定义你想要将对象放在哪里的绝对位置，而是定义相对当前位置对象移动的量。假设你已经在场景中添加了一个球体，位置是（1，2，3）。下一步我们想让这个对象沿着 X 轴平移，translateX（4）。它的位置现在就是（5，2，3）。如果我们想让这个对象恢复到它原来的位置，可以调用 translateX（-4）。在例子 06-mesh-properties.html 里有一个 translate 菜单项。通过它你可以试验这个功能。只要设置 x、y 和 z 轴方向平移的距离，点击 translate 按钮，你就会看到这个对象会依照这三个值移动到一个新位置。

在第 5 章和第 7 章里，你可以找到更多关于网格和几何体的信息，以及如何使用这些对象。

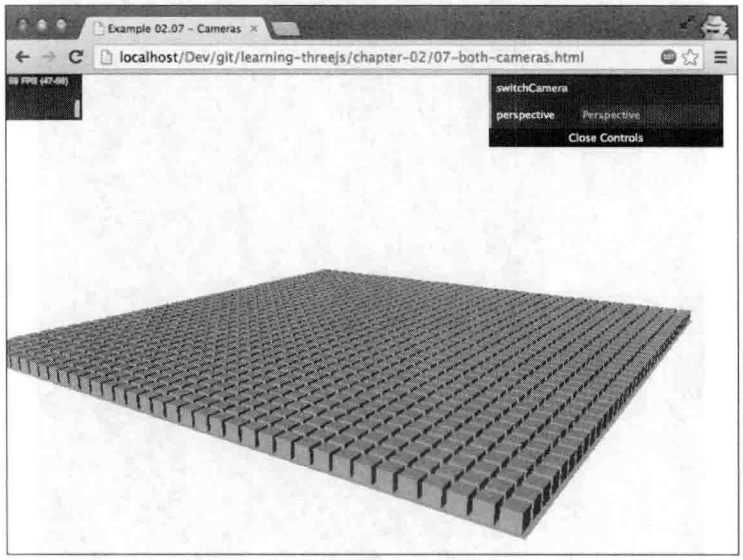
2.3　选择合适的相机

Three.js 库里有两种不同的相机：正投影相机和透视相机。我们将在第 3 章详细看一下如何使用这些相机，所以本章我只会讲一些比较基础的内容。要解释这两种相机的不同之处，最好的方法就是看几个例子。

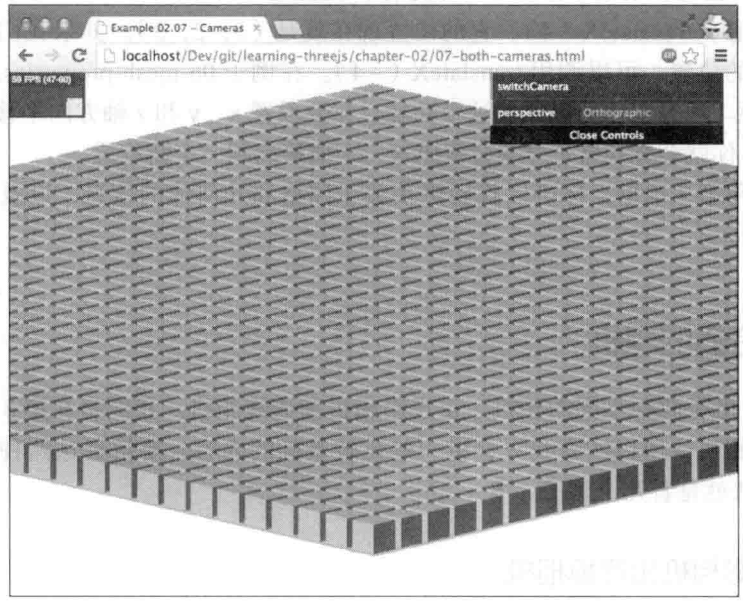
2.3.1　正投影相机和透视相机

本章的例子里有一个名为 07-both-cameras.html 的例子。当你打开这个示例时，你会看到类似下图所示的结果：

这是一个透视视图，也是最自然的视图。正如你所看到的，距离相机越远的方块，被渲染得越小。



如果我们把这个相机改为 Three.js 库支持的另外一种相机，即正投影相机，那么对于同一个场景你会看到如下的视图：



使用正投影相机的话，所有方块渲染出来的尺寸都一样；对象和相机之间的距离不会影响渲染结果。这种相机通常用在二维游戏中，例如《模拟城市 4》和早期版本的《文明》，如下图所示：

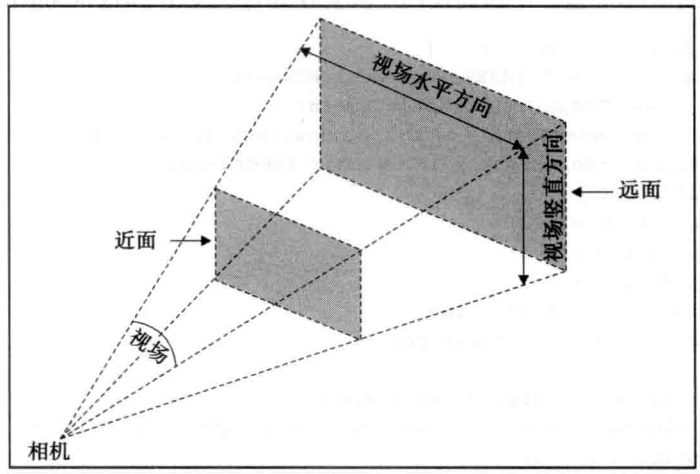
在我们的示例里，应尽量使用透视相机，因为它最贴近真实世界。换相机非常简单。在示例 07-both-cameras.html 里，下面的代码片段会在你每次点击 switchCamera 按钮时调用：

```
this.switchCamera = function() {
  if (camera instanceof THREE.PerspectiveCamera) {
    camera = new THREE.OrthographicCamera(
          window.innerWidth / - 16, window.innerWidth / 16,
          window.innerHeight / 16, window.innerHeight / - 16,
          -200, 500 );
    camera.position.x = 2;
    camera.position.y = 1;
    camera.position.z = 3;
    camera.lookAt(scene.position);
    this.perspective = "Orthographic";
  } else {
    camera = new THREE.PerspectiveCamera(45,
            window.innerWidth / window.innerHeight, 0.1, 1000);
    camera.position.x = 120;
    camera.position.y = 60;
    camera.position.z = 180;

    camera.lookAt(scene.position);
    this.perspective = "Perspective";
    }
};
```

在上面这个代码片段里你可以看到，我们创建 THREE.PersepectiveCamera 的方法跟创建 THREE.OrthographicCamera 的方法有些不一样的地方。我们先来看一下 THREE.PerspectiveCamera。它接受如下这些参数：

参　数	描　述
fov（视场）	fov 表示视场（field of view）。这是从相机位置能够看到的部分场景。例如，人类有差不多 180 度的视场，而一些鸟类差不多会有一个完整的、360 度的视场。 但是由于普通的计算机显示器不能完全显示我们看到的景象，所以一般会选择一块儿较小的区域。对于游戏来讲，大多数情况下会用 60 度到 90 度左右的视场。 推荐默认值：45
aspect（长宽比）	这是渲染结果输出区的横向长度和纵向长度的比值。在我们的例子中，由于我们会使用整个窗口作为输出界面，所以会使用这个窗口的长宽比。这个长宽比决定了水平视场和垂直视场之间的比例关系，正如你在下图所看到的那样。 推荐默认值：window.innerWidth/window.innerHeight
near（近面）	near 属性定义的是 Three.js 库从距离相机多近的地方开始渲染场景。通常情况下我们会为这个属性设一个很小的值，从而可以渲染从相机位置可以看到的所有物体。 推荐默认值：0.1
far（远面）	far 属性定义的是相机可以从它所处的位置看多远。如果我们将这个值设得太低，那么场景中的一部分可能不会被渲染；如果设得太高，在某些情况下会影响渲染的效率。 默认值：1000

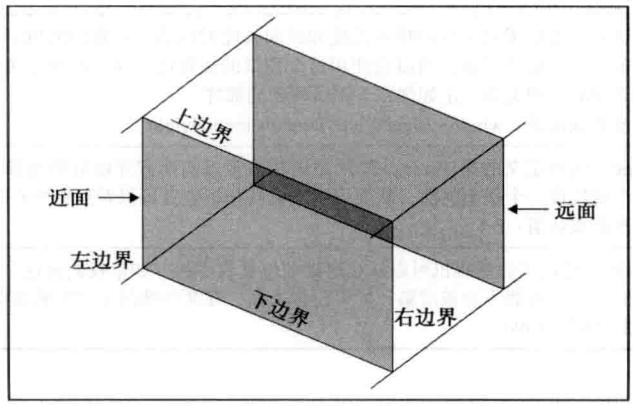
下图很好地展示了这些属性是如何结合在一起，从而决定你能看到什么：



要配置正投影相机，我们得使用其他一些属性。正投影不关心使用什么样的长宽比，或者以什么样的视角观察场景。所有对象渲染的尺寸都一样。对于一个正投影相机来讲，你需要定义一个需要渲染的方块。下表中的 OrthographicCamera 对象的属性反应了这一点：

参　数	描　述
left（左边界）	在 Three.js 文档里该属性是可视范围的左平面。你可以将它当做是可渲染部分的左侧边界。如果我们把这个值设为 -100，那么你就不会看到任何比这个左侧边界更远的对象
right（右边界）	跟 left 属性一样，不过这次是界面的另外一侧。比这个右侧边界更远的对象不会被渲染
top（上边界）	可被渲染空间的最上面
bottom（下边界）	可被渲染空间的最下面
near（近面）	基于相机所在的位置，从这一点开始渲染场景
far（远面）	基于相机所在的位置，一直渲染到场景中的这一点

所有这些属性都可以在下图中找到：

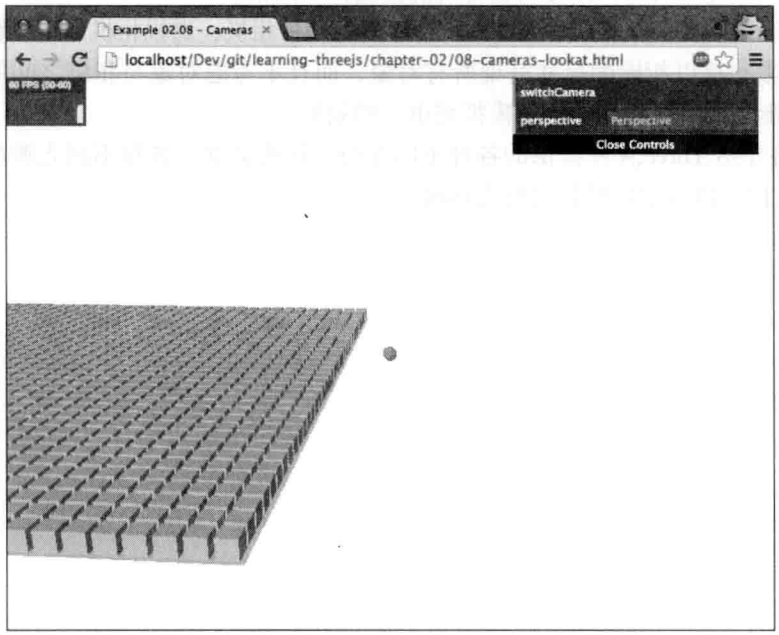


2.3.2　让相机在指定点上聚焦

到目前为止我们已经看了如何创建相机，以及各个参数的含义。在前一章，我们也看到过你需要将相机放在场景中的某个位置，以及从相机到场景中心的视图将会被渲染。一般来讲，相机会指向场景的中心，用坐标来表示就是 position（0，0，0）。但是我们也可以轻松改变相机所看的位置，如下代码所示：

```
camera.lookAt(new THREE.Vector3(x,y,z));
```

我已经加了一个例子，在这个例子中，相机可以移动，而且它所指向的位置用一个红点标识了出来，如下图所示：



如果你打开例子 08-cameras-lookat.html，你会看到场景正在从左向右移动。场景并不是真的在移动。而是相机在拍摄不同的地方（屏幕中央那个红点），其效果就是场景在从左向右移动。你也可以在这个例子里将相机换成正投影相机。你会看到改变相机拍摄地方的效果跟用透视相机时有所不同。

2.4　总结

本章我们讨论了很多东西。而这应该会让你对什么是场景，以及场景中最重要的组成部分有所了解。在下面几章里，我们会深入介绍 Three.js 库的细节。下面是本章中你应该记住的几点：

- ❑ 场景是 Three.js 库中的主要容器。你可以将你想要渲染的对象添加到场景中。
- ❑ 场景并没有很多特殊的选项和属性。它最重要的功能是允许你添加对象、移除对象，以及处理场景的 children 属性。
- ❑ 你可以通过配置 Fog 对象为场景添加 fog 属性。
- ❑ 几何体和网格关系紧密。几何体定义对象的外观，赋予材质后你可以用它来创建网格。Three.js 库可以渲染网格。
- ❑ Three.js 库提供了很多标准几何体。但是你也可以自己创建一些。但是如果不使用算法的话，这需要完成很多工作。
- ❑ 你也可以编程控制 mesh 的 position、rotation 和 scale 属性。
- ❑ 通过 translate 属性，你可以相对当前位置移动网格。
- ❑ 渲染场景需要一个相机。在 Three.js 库里有两种相机：透视相机和正投影相机。
- ❑ 正投影相机以相同的尺寸渲染所有对象，而且不考虑对象与相机之间的距离。使用正投影相机可以达到类似《模拟城市》的效果。

下一章将介绍 Three.js 库提供的各种不同光源。你将会学习各种不同光源的行为，如何创建和配置它们，以及它们对特定材质影响。

第 3 章 Chapter 3

使用 Three.js 里的各种光源

在第 1 章里你已经学习过了 Three.js 库的一些基础知识，在前一章里我们深入地学习了场景中最重要的部分：几何体、网格和相机。你可能也注意到了我们在前一章里没有讲光源，尽管它们是每个 Three.js 场景的重要组成部分。如果没有光源，我们就不可能看到任何渲染结果。由于 Three.js 库提供了很多种光源，每一种都有特殊的用途，因此我们将会用整整一章来讲解各种光源的细节，并为下一章使用材质做好准备。在本章中，你将会学到下列主题：

- Three.js 库里有哪些可用的光源
- 什么时候用什么样的光源
- 如何调整和配置各种光源的行为
- 如何创建镜头眩光

在前面这些章节里，我们已经拥有很多例子了，你可以用它们来试验光源的行为。本章的例子可以在随书源代码的 chapter-03 文件夹下找到。

3.1 探索 Three.js 库提供的光源

Three.js 库提供了一系列光源，而且每种光源都有特定的行为和用途。本章将要讨论下面这些光源：

光源名称	描　述
AmbientLight（环境光）	这是一种基础光源，它的颜色会添加到整个场景和所有对象的当前颜色上
PointLight（点光源）	空间中的一点，朝所有的方向发射光线
SpotLight（聚光灯光源）	这种光源有聚光的效果，类似台灯、天花板上的吊灯，或者手电筒

（续）

光源名称	描　述
DirectionalLight（方向光）	也称作是无限光。从这种光源发出的光线可以看作是平行的。例如，太阳光
HemisphereLight（半球光）	这是一种特殊光源，可以用来创建更加自然的室外光线，模拟反光面和光线微弱的天空
AreaLight（面光源）	使用这种光源可以指定散发光线的平面，而不是空间中的一个点
LensFlare（镜头眩光）	这不是一种光源，但是通过 LensFlare 可以为场景中的光源添加眩光效果

本章分为两个主要部分。首先，我们会看一下各种基础光源：AmbientLight、PointLight、SpotLight 和 DirectionalLight。这些光源比较简单，只需要很少的设置，可以用来重建大多数场景所需的光照效果。然后，我们会看几个用于特殊目的的光源和效果：HemisphereLight、AreaLight 和 LensFlare。只有在特殊场合下你才会用到它们。

3.2 学习基础光源

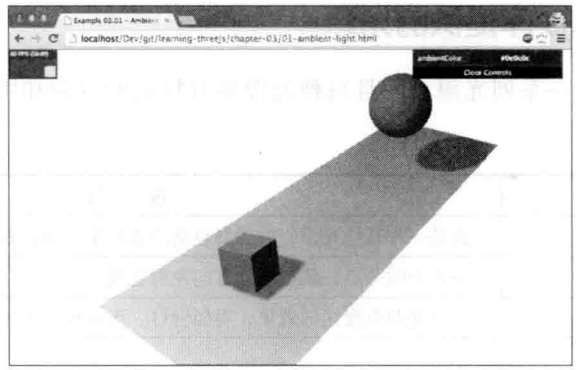
我们从最基础的光源（AmbientLight，环境光）开始。

3.2.1 AmbientLight——影响整个场景的光源

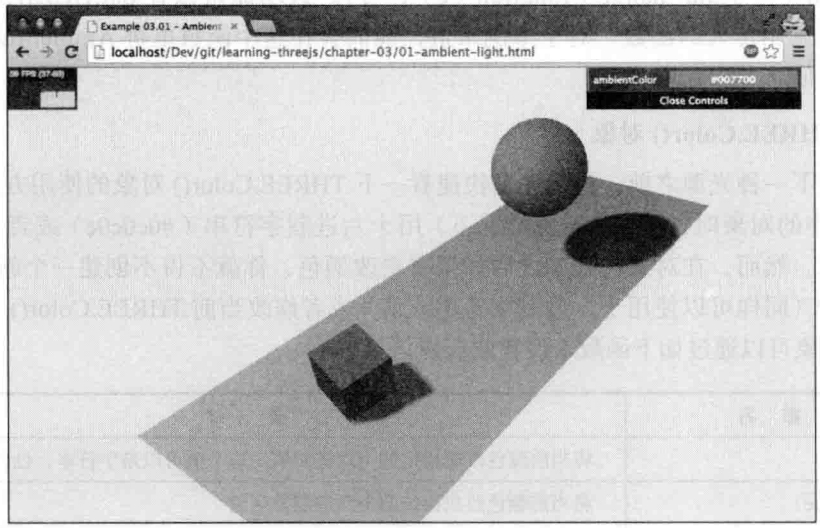
AmbientLight 光源的颜色会影响整个场景。AmbientLight 的光线没有特定的来源，而且这个光源也不会影响阴影的生成。你不能将 AmbientLight 作为场景中的唯一光源。在使用其他光源（例如 SpotLight 和 DirectionalLight）的同时使用 AmbientLight，目的是弱化阴影或添加一些颜色。要理解这一点最简单的办法就是看一下 chapter-03 文件夹下的例子 01-ambient-light.html。在这个例子里，可以使用一个简单的用户界面来修改添加到场景中的 AmbientLight 光源。

在这个例子里，使用 SpotLight 光源来照亮物体，并生成阴影。

从下面这个屏幕截图里你可以看到，我们使用的是第 1 章中的那个场景，而且 AmbientLight 光源的颜色是可调的：



我们在这个场景里使用的标准颜色是 #0c0c0c。这是该颜色的十六进制表示形式。如果你对用十六进制表示颜色的方式不太熟悉，可以参考维基百科上的内容：http://en.wikipedia.org/wiki/Web_colors#Hex_triplet。在这个例子里我们将使用一个非常暗淡的浅灰色，用来弱化网格对象在地面（plane 对象）上生硬的投影。如果你想把这个颜色改成比较明亮的绿色（#007700），你可以在右上角的菜单上完成；各个对象就会笼罩在一片明亮的绿光下，如下图所示：



正如上图所示，绿色应用到了所有的物体上，并在整个场景中投下一片绿色光辉。使用这种光源时你应该记住的是：用色应该尽量保守。如果你指定的颜色过于明亮，那么你很快就会发现画面颜色过于饱和了。

既然我们已经看过 AmbientLight 能做什么，接下来学习如何创建和使用 AmbientLight 光源。下面几行代码展示了如何创建 AmbientLight 光源，以及如何将该光源与 dat.GUI 控制菜单关联起来：

```
var ambiColor = "#0c0c0c";
var ambientLight = new THREE.AmbientLight(ambiColor);
scene.add(ambientLight);
...

var controls = new function() {
   this.ambientColor = ambiColor  ;
}

var gui = new dat.GUI();
gui.addColor(controls,.'ambientColor').onChange(function(e) {
  ambientLight.color = new THREE.Color(e);
});
```

创建 AmbientLight 光源非常简单。由于 AmbientLight 光源不需要指定位置，因此只需要使用 new THREE.AmbientLight（ambiColor）指定颜色（十六进制）。然后将该光源添加到场景中即可。在这个例子里将 AmbientLight 光源的颜色绑定到控制菜单。为此，可以使用跟前面章节相同的配置方法。唯一需要改变的是将调用 gui.addColor（…）函数，而不是 gui.add（…）函数。该方法会在控制菜单里添加一个选项，在这个选项里可以直接改变传入的颜色变量。在代码中你可以看到我们使用了 dat.GUI 控制菜单的 onChange 功能，gui.addColor（…）.onChange（function（e）{…}）。该功能可以让 dat.GUI 在每次颜色值改变的时候调用传入的函数。对于本例来讲，我们会在这个函数里将 AmbientLight 光源的颜色设为新值。

使用 THREE.Color() 对象

在讲述下一种光源之前，我们先来快速看一下 THREE.Color() 对象的使用方法。当构造 Three.js 库中的对象时，可以（通常情况下）用十六进制字符串（#0c0c0c）或者十六进制数值指定颜色。然而，在对象构造好之后如果要更改颜色，你就不得不创建一个新的 THREE.Color() 对象（同样可以使用十六进制字符串或值）或者修改当前 THREE.Color() 对象的内部属性。该对象可以通过如下函数来设置或获取信息：

函 数 名	描 述
set（value）	将当前颜色改成指定的十六进制值。这个值可以是字符串，也可以是数字
setHex（value）	将当前颜色改成指定的十六进制数字值
setRGB（r，g，b）	根据提供的 RGB 值设置颜色。参数的范围从 0 到 1
setHSV（h，s，v）	根据提供的 HSV 值设置颜色。参数的范围从 0 到 1
setStyle（style）	根据 CSS 值设置颜色
copy（color）	从提供的颜色对象复制颜色值到当前对象
copyGamaToLinear（color）	通常在内部使用： 根据提供的颜色设置物体的颜色。颜色参数先从伽马色彩空间转换到线性色彩空间。伽马色彩空间用的也是 RGB 值，但是会带有一个额外的比例系数，而在线性色彩空间中这个系数是 1
CopyLinearToGamman（color）	主要在内部使用： 基于提供的颜色设置物体的颜色。该颜色首先从线性色彩空间转换到伽马色彩空间
convertGamaToLinear()	将当前颜色从伽马色彩空间转换到线性色彩空间
convertLinearToGamma()	将当前颜色从线性色彩空间转换到伽马色彩空间
getHex()	以十六进制值形式从颜色对象中获取颜色值
getHexString()	以十六进制字符串形式从颜色对象中获取颜色值
getStyle()	以 CSS 值的形式从颜色对象中获取颜色值
getHSV()	以 HSV 形式从颜色对象中获取颜色值

（续）

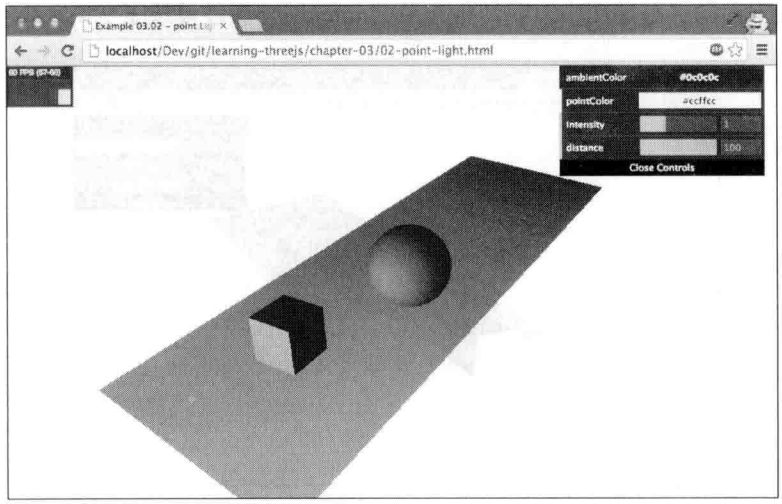
函 数 名	描　述
add（color）	将提供的颜色添加到当前颜色上
addColors（color1，color2）	通常在内部使用： 将提供的各个颜色添加到当前颜色
addScalar（s）	通常在内部使用： 在当前颜色的 RGB 分量上添加值
multiply（color）	通常在内部使用： 将当前颜色与提供的颜色相乘
multiplyScalar（s）	通常在内部使用： 将当前光源与提供的值相乘
lerp（color，alpha）	通常在内部使用： 找到当前颜色和指定颜色之间的颜色值，并将该颜色值与提供的 alpha 值相乘
clone()	复制当前颜色

从这张表里你可以看出改变当前颜色有很多方法。这些方法中有很多是 Three.js 库内部使用的，但也有一些方法可以轻松修改光源和材质的颜色。

接下来学习 PointLight（点光源）。

3.2.2　PointLight——照射所有方向的光源

Three.js 库中的 PointLight（点光源）是一种单点发光，照射所有方向的光源。夜空中的照明弹就是一个很好的点光源例子。跟所有光源一样，我们有一个专门的例子，你可以通过这个例子来试验一下 PointLight。如果你打开 chapter-03 文件夹下的例子 02-point-light.html，你会看到一个点光源正在绕场景移动，如下图所示：



这还是第 1 章中那个例子的场景，只是这次有一个点光源绕场景移动。为了能清楚看到这个点光源在哪儿，我们让一个橙色的小球（sphere 对象）沿着相同的轨迹移动。随着光源的移动，你将看到红色的方块和蓝色的球被照亮。

你可能会注意到在这个例子中看不到任何阴影。在 Three.js 库中点光源不会产生投影。原因是点光源会朝所有方向发射光线，在这种情况下计算阴影对 GPU 来讲是个沉重的负担。

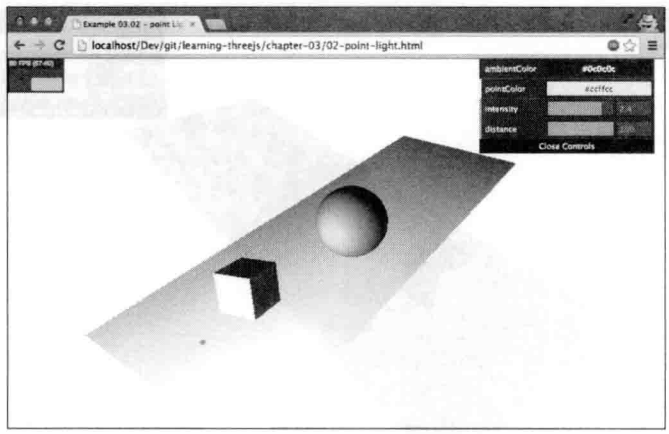
跟前面的 AmbientLight（环境光）光源一样，你所要做的只是设置颜色，并将光源添加到场景中即可。但对于点光源来说，我们还可以设置如下的几个属性：

属　性	描　述
color（颜色）	光源颜色
intensity（强度）	光照的强度。默认值是 1
distance（距离）	光源照射的距离
position（位置）	光源所在的位置
visible（是否可见）	如果设为 true（真），该光源就会打开；如果设为 false（假），该光源就会关闭

我们将会在接下来的几个例子和屏幕截图中解释这些属性。首先我们来看看如何创建点光源：

```
var pointColor = "#ccffcc";
var pointLight = new THREE.PointLight(pointColor);
pointLight.distance = 100;
scene.add(pointLight);
```

这里没有什么新东西。我们使用指定的颜色创建了一个光源，设置好位置并添加到场景中。首先要看的属性是 intensity。通过该属性你可以设置光线的亮度。如果设成 0，那么你将什么都看不到。设成 1，你得到的将是默认的亮度。如果设成 2，你得到的将是两倍的亮度。例如，在下面的屏幕截图中，我们将光源的 intenisity 属性设成了 2.4：



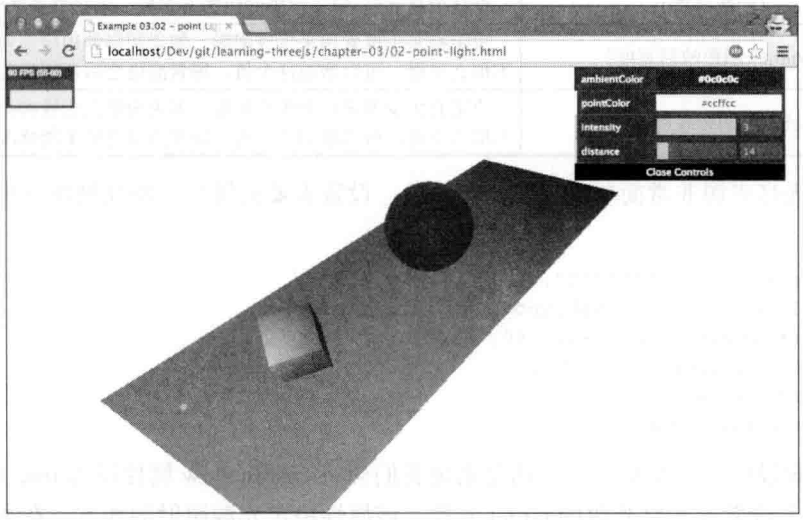
改变光源的 intensity 属性只须添加如下的代码：

```
pointLight.intensity = 2.4;
```

你可以参考如下的代码，使用 dat.CUI 监听器：

```
var controls = new function() {
  this.intensity = 1;
}
var gui = new dat.GUI();
   gui.add(controls, 'intensity', 0, 3).onChange(function (e) {
     pointLight.intensity = e;
   });
```

点光源的最后一个属性很有意思，我们最好用一个例子来解释它。在下面的屏幕截图中你看到的还是同一个场景，只是这次的 intensity 值很高（一个非常明亮的光源），但距离很短。



点光源的 distance（距离）属性决定的是光线可以照射多远。你可以像这样设置该属性：pointLight.distance = 4；。在我们的例子中，光照的亮度（intensity）在 14 的距离（distance）内慢慢降到 0. 这就是为什么在这个例子中我们依然可以看到一个被照亮的方块（cube），但是却看不到这个蓝色的球（sphere），正如早前你在屏幕截图中所看到的那样。这个 distance 属性的默认值是 0，也就是说光线的亮度不会随着距离的增加而递减。

3.2.3　SpotLight——具有锥形效果的光源

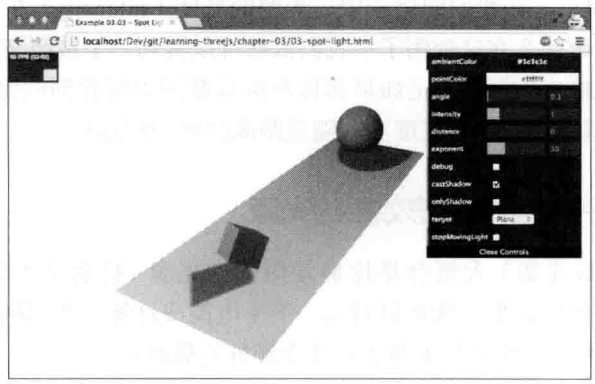
SpotLight（聚光灯光源）大概会是你最常用到的光源（特别是当你想要生成阴影时）。SpotLight 光源有一种锥形效果。你可以对比一下手电筒或灯笼。我们刚看过的点光源的属性也可以应用到聚光灯光源。聚光灯光源也有几个额外的属性：

属　性	描　述
castShadow（投影）	如果设为 true，这个光源就会生成阴影
shadowCameraNear（投影近点）	从距离光源的哪一点开始可以生成阴影
shadowCameraFar（投影远点）	到距离光源的哪一点为止可以生成阴影
shadowCameraFov（投影视场）	用于生成阴影的视场有多大（参见 2.3.1 节）
target（目标）	决定光照的方向
shadowBias（阴影偏移）	用来偏置阴影的位置
angle（角度）	光源射出的光柱有多宽。单位是弧度，默认值是 Math.PI/3
exponent（光强衰减指数）	光照指向特定目标。在这个方向上距离光源越远，则光照强度递减得越快。这个值决定光照强度递减得有多快
onlyShadow（仅阴影）	如果设为 true，那么这个光源只会生成阴影，而不会在场景中添加任何光照
shadowCameraVisible（投影方式可见）	如果设为 true，你就可以看到光源在哪里以及如何生成阴影（参考下一节的例子）
shadowDarkness（阴影暗度）	默认值是 0.5。定义渲染的阴影有多黑。场景渲染之后不能修改
shadowMapWidth（阴影映射宽度）	决定有多少像素用来生成阴影。如果阴影的边缘参差不齐或看上去不那么平滑，可以增加这个值。场景渲染之后不能修改
shadowMapHeight（阴影映射高度）	决定有多少像素用来生成阴影。如果阴影的边缘参差不齐或看上去不那么平滑，可以增加这个值。场景渲染之后不能修改

创建聚光灯光源非常简单。只要指定颜色、设置需要的属性、添加到场景中即可，如以下代码所示：

```
var pointColor = "#ffffff";
var spotLight = new THREE.SpotLight(pointColor);
spotLight.position.set(-40, 60, -10);
spotLight.castShadow = true;
spotLight.target = plane;
scene.add(spotLight);
```

跟点光源没什么大差别。唯一的差别是我们会将 castShadow 属性设为 true（因为我们想要阴影），以及设置聚光灯光源的 target 属性。该属性决定光源照射的地方。在本例中我们将会指向地面对象 plane 的中心。当运行例子 03-spot-light.html 时，你会看到如下场景：



在这个例子里，可以设置一些聚光灯光源独有的属性。target 属性是其中一个。如果把这个属性设为那个蓝色的小球（sphere 对象），那么这个光源将会一直瞄准这个球的中心，即使它会在场景中移动。我们创建这个光源时，瞄准的是地面（plane）对象。在本例中，也可以让它指向其他两个对象。但是如果你不想让光源瞄准某个特定的对象，而是让它指向空间中任意一点该怎么办？要达到这个目的，可以创建一个空的 THREE.Object3D() 实例，如下所示：

```
var target = new THREE.Object3D();
target.position = new THREE.Vector3(5, 0, 0);
```

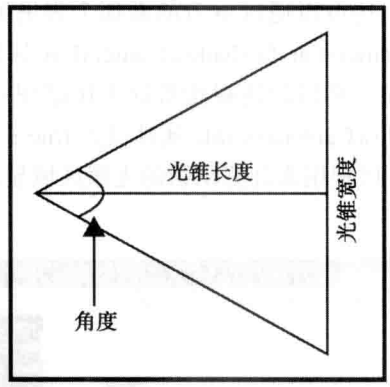
并按如下代码设置聚光灯光源的 target 属性：

```
spotlight.target = target
```

本节前面的那张表列出了几个可以用来控制光线如何从光源发散出来的属性。distance 属性和 angle 属性定义了光锥的形状。angle 属性定义了光锥的宽度；distance 属性定义的是光锥的长度。如果探究一下 Three.js 的源代码，我们就可以看到这究竟是如何定义的：

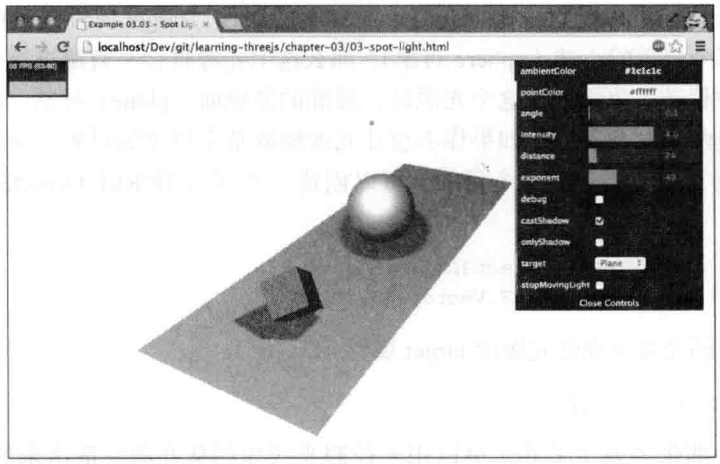
```
var coneLength = light.distance ? light.distance : 10000;
var coneWidth = coneLength * Math.tan( light.angle * 0.5 ) * 2;
```

我们来看一下第二个语句，但我们不会深入探讨三角函数。正切函数（Math.tan()）可以用来计算对边长度（光锥宽度）和邻边长度（光锥长度）之比。下图展示的就是角度和光锥长度是如何决定光锥宽度的。



在 Three.js 的代码里，光源的发射角度会先除以 2（参考前面的示意图），因为正切函数应该在直角三角形里使用。要获取光锥宽度，函数 Math.tan() 的结果（比值）要乘以光锥长度。这样得到的是光锥宽度的一半，我们还要乘以 2 才能得到最终的光锥宽度。

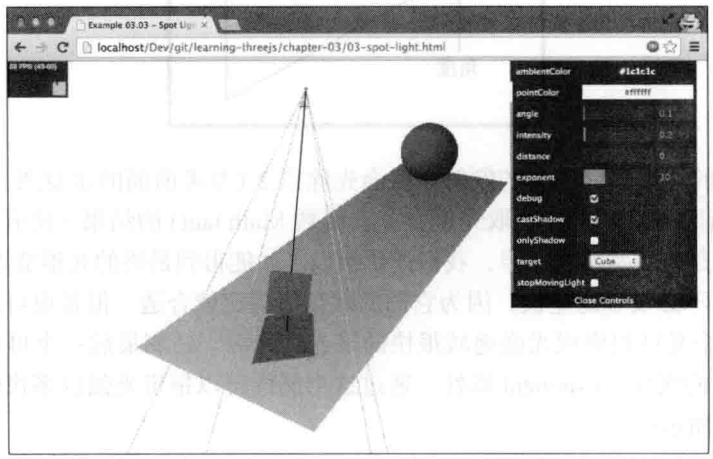
一般情况下不必设置这些值，因为它们的默认值都比较合适。但是也可以用这些属性来创建（例如）一个光柱很窄或光强递减很快的聚光灯光源。还剩最后一个可以用来改变聚光灯光源渲染方式的属性，exponent 属性。通过这个属性可以指定光强以多快的速度从中心开始衰减，如下图所示：



我们有一个非常明亮的光束（intensity 值很高），离中心越远光强衰减得越快（exponent 值很高）。也可以使用小 exponent 值和小 angle 来创建同样的聚光效果。

如果角度很小，那它很快就会导致光线渲染失真（这个术语在图形学中是用来描述意料之外的变形或屏幕上渲染得比较奇怪的地方）。

在讲述下一个光源之前，我们先快速看一下聚光灯光源提供的几个跟阴影相关的属性。我们已经知道要生成阴影可以将聚光灯光源的 castShadow 属性设为 true。Three.js 库也允许你微调阴影渲染的方法。这可以通过本节前面那个表里解释的几个属性来完成。通过 shadowCameraNear、shadowCameraFar 和 shadowCameraFov 属性，可以控制光线如何投影、在哪里投影。其工作原理跟前一章讲的透视相机的工作原理一致。要看看这些是如何发生的，最简单的方法就是将 shadowCameraVisible 属性设为 true；可以通过构造菜单上的 debug（调试）复选框来设置。这样可以把用来计算阴影的光照区域显示出来，正如你在下图看到的那样：



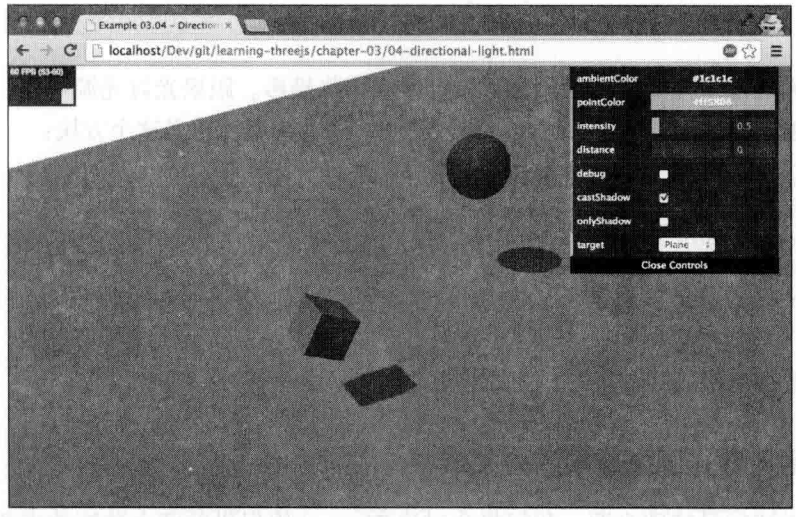
下面针对使用阴影你可能会遇到的几个问题列出了几个提示：

- 启用 shadowCameraVisible 属性。这样会把受光源影响、生成阴影的区域显示出来。
- 如果阴影看上去有点儿模糊，可以增加 shadowMapWidth、shadowMapHeight 属性的值，或者保证用于生成阴影的区域紧密包围着对象。可以使用 shadowCameraNear、shadowCameraFar 和 shadowCameraFov 属性来配置这个区域。
- 记住，你不仅要告诉光源生成阴影，你还要通过 castShadow 和 receiveShadow 属性告诉每个几何体是否接收和（或）投射阴影。

3.2.4　DirectinalLight——模拟远处类似太阳的光源

我们要看的最后一个基础光源是 DirectionalLight（方向光）光源。方向光光源可以看做是距离很远的光源。这个光源发出的所有光线都是相互平行的。方向光源的一个范例是太阳。太阳是如此遥远，以至于到达地球的光线都成了平行光。方向光光源和我们之前看过的聚光灯光源之间主要的差别是：方向光不像聚焦光那样离目标越远越暗淡。被方向光光源照亮的整个区域接收到的光强是一样的。

你可以在例子 04-directional-light.html 里看到实际效果，如下图所示：

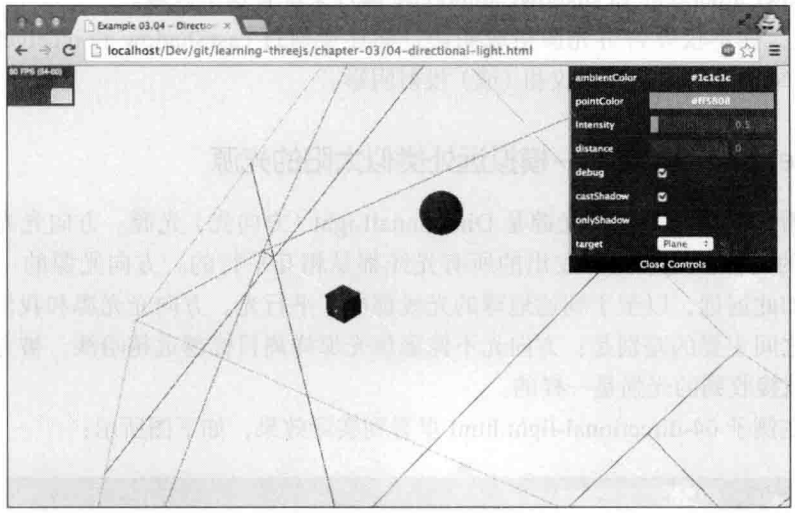


正如你所看到的，这个场景里没有那种锥形的光线。所有对象接收到的都是相同光强的光。在这种光里只用 direction（方向）、color（颜色）和 intensity（强度）属性计算颜色和阴影。

跟聚光灯光源一样，可以设置几个属性来控制光照强度和投影方法。方向光光源和聚光灯光源有很多属性是相同的，例如 position、target、intensity、distance、castShadow、onlyShadow、shadowCameraNear、shadowCameraFar、shadowDarkness、shadowCameraVisible、shadowMapWidth、shadowMapHeight 和 shadowBias。有关这些属性更多的信息，可以参照

3.2.3 节的表格。下面几段只讨论方向光光源特有的属性。

如果你研究一下那个聚光灯光源的例子，你会发现我们不得不定义生成阴影的光锥。然而，对于方向光光源，由于所有光线都是相互平行的，所以不会有光锥；取而代之的是一个方块，如下图所示：



在这个方块范围内的所有对象都可以投影或接收投影。跟聚光灯光源一样，包围对象的空间定义得越紧密，投影的效果越好。可以使用下面几个属性定义这个方块：

```
directionalLight.shadowCameraNear = 2;
directionalLight.shadowCameraFar = 200;
directionalLight.shadowCameraLeft = -50;
directionalLight.shadowCameraRight = 50;
directionalLight.shadowCameraTop = 50;
directionalLight.shadowCameraBottom = -50;
```

你可以把这个跟 2.3 节定义正投影相机的方法比较一下。

3.2.5 使用特殊光源生成高级光照效果

这一节讲述的是特殊光源，我们将会讨论 Three.js 库提供的两个独特的光源。首先要讨论的是 HemisphereLight（半球光光源），这种光源可以为室外场景创建更加自然的光照效果。然后我们会看一看 AreaLight（平面光光源），它可以从一个很大的平面发射光线，而不是从单个点。最后我们会展示一下如何在场景中添加镜头眩光效果。

3.2.5.1 半球光光源

我们将要看的第一个特殊光源是 THREE.HemisphereLight（半球光光源）。使用半球光光源，可以创建出更加贴近自然的光照效果。如果不使用这种光源，要模拟室外光照，可以添

加一个方向光光源来模拟太阳，或者再添加一个环境光光源，为场景提供基础色。但是，这样看上去不怎么自然。当你在室外的时候，并不是所有的光照都来自上方；很多是来自空气的散射、地面的反射，以及其他物体的反射。Three.js 库中的 HemisphereLight（半球光光源）就是为这种情形创建的。它提供了一种简单的方法，用以获取比较自然的光照效果。可以浏览示例 05-hemisphere-light.html。这是第一个需要本地 Web 服务器的例子。如果你还没有，可以参考第 1 章，创建一个本地的 Web 服务器。下图展示的场景使用的就是半球光光源：

在这个例子中，可以打开或关闭半球光光源，设置颜色（color）和光强（intensity）。创建半球光光源非常简单，如以下代码片段所示：

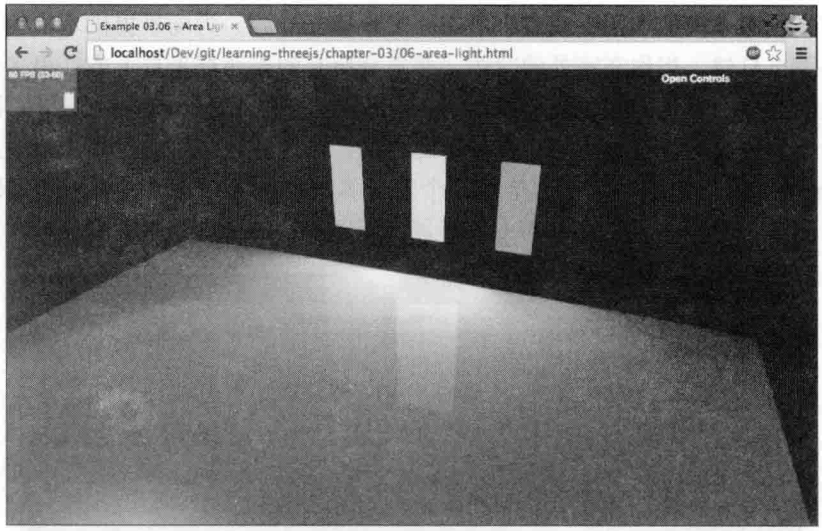
```
var hemiLight = new THREE.HemisphereLight(0x0000ff, 0x00ff00, 0.6);
hemiLight.position.set(0, 500, 0);
scene.add(hemiLight);
```

你只要指定来自上方（光线）的颜色（接收自天空的颜色）、接收自地面的颜色，以及它们的光照强度。之后如果想修改这些属性，可以使用如下属性：

属　性	描　述
groundColor	从地面发出的光线的颜色
Color	从天空发出的光线的颜色
intensity	光线照射的强度

3.2.5.2　平面光光源

我们最后要看的光源是 AreaLight（平面光光源）。使用平面光光源，可以定义一个发光的矩形。AreaLight 并不在标准 Three.js 库中，而是在它的扩展库中，所以在使用之前我们要完成几个额外的步骤。在深入细节之前先来看一下我们追求的结果（例子 06-area-light.html）：



如果你要使用平面光光源，那么就不能再用之前我们一直在示例中使用的 THREE.WebGLRenderer 对象了。因为平面光光源是一种非常复杂的光源；它会对 THREE.WebGLRenderer 对象造成非常严重的性能损失。而 THREE.WebGLDeferredRenderer（WebGL 的延迟渲染器）对象在渲染场景时使用了一种不同的方法，可以处理复杂的光照（或者光源很多的情况）。

要使用 THREE.WebGLDeferedRenderer 对象，必须引入 Three.js 提供的额外的几段 JavaScript 源代码。在 HTML 页面框架的头部，加上如下这些 <script/> 源代码：

```
<head>
    <title>Example 03.07 - Area Light</title>
    <script type="text/javascript" src="../libs/three.js"></script>
    <script type="text/javascript"
          src="../libs/jquery-1.9.0.js"></script>
    <script type="text/javascript" src="../libs/stats.js"></script>
    <script type="text/javascript" src="../libs/dat.gui.js"></script>
    <script type="text/javascript"
          src="../libs/WebGLDeferredRenderer.js"></script>
    <script type="text/javascript"
          src="../libs/ShaderDeferred.js"></script>
    <script type="text/javascript"
          src="../libs/RenderPass.js"></script>
    <script type="text/javascript"
          src="../libs/EffectComposer.js"></script>
    <script type="text/javascript"
          src="../libs/CopyShader.js"></script>
    <script type="text/javascript"
          src="../libs/ShaderPass.js"></script>
    <script type="text/javascript"
          src="../libs/FXAAShader.js"></script>
    <script type="text/javascript"
```

```
        src="../libs/MaskPass.js"></script>
</head>
```

包含了这些库之后，就可以使用 THREE.WebGLDeferredRenderer 对象了；这个渲染器的使用方法，跟我们在其他示例里使用的那个差不多。它只是多了几个参数而已，如下面的代码片段所示：

```
var renderer = new THREE.WebGLDeferredRenderer({
    width: window.innerWidth,
    height: window.innerHeight,
    scale: 1, antialias: true,
    tonemapping: THREE.FilmicOperator, brightness: 2.5 });
```

现在不用太在意这些属性的含义。第 10 章会深入讲解 WebGL 的延迟渲染器（THREE.WebGLDefferredRenderer）对象，到时再来解释。在添加正确的 JavaScript 库和不同的渲染器之后，就可以开始添加 Three.AreaLight 对象，并修改它的属性。

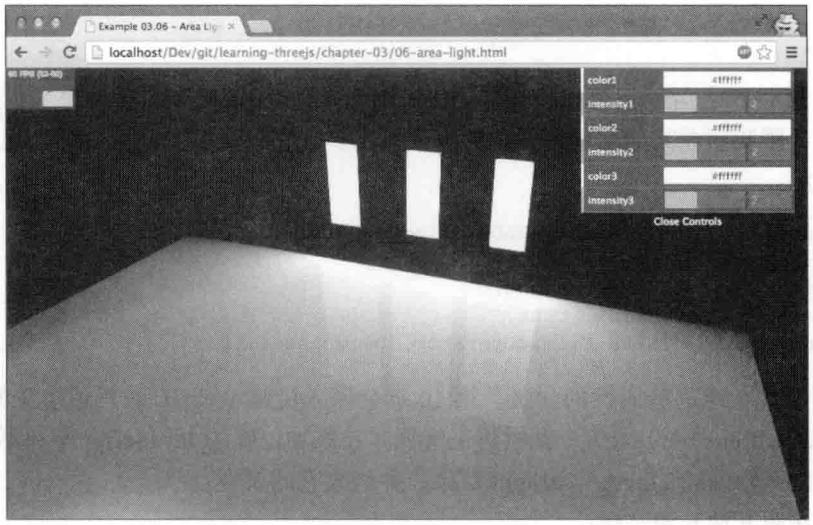
处理方法跟其他光源一样：

```
var areaLight1 = new THREE.AreaLight(0xff0000, 3);
areaLight1.position.set(-10, 10, -35);
areaLight1.rotation.set(-Math.PI / 2, 0, 0);
areaLight1.width = 4;
areaLight1.height = 9.9;
scene.add(areaLight1);
```

在这个示例里，创建了一个 THREE.AreaLight 实例。这个光源的颜色为 0xff0000，光强是 3。跟其他光源一样，可以使用 position 属性设置该光源在场景中的位置。在创建 THREE.AreaLight 实例的同时也会创建出一个水平面。在这个例子里，创建出的平面光光源是竖立的，因此要绕 x 轴将这个光源旋转 -Math.PI/2。最后，使用 width 和 height 属性设置平面光光源的尺寸，并把它添加到场景中。当你自己第一次尝试该光源的时候，你可能会觉得奇怪为什么在你放置光源的地方什么都看不到呢。这是因为你不能看到光源本身，而只能看到它发出的光，而且只有当这些光照射到某个物体上时才能看到。如果你想复制出我在例子中所展示的场景，你可以在相同的位置（由 position 属性指定）处放置一个平面或方块，用来模拟平面光光源，如下所示：

```
var planeGeometry1 = new THREE.CubeGeometry(4, 10, 0);
var planeGeometry1Mat = new THREE.MeshBasicMaterial({color: 0xff0000})
var plane1 = new THREE.Mesh(planeGeometry1, planeGeometry1Mat);
plane1.position = areaLight1.position;
scene.add(plane1);
```

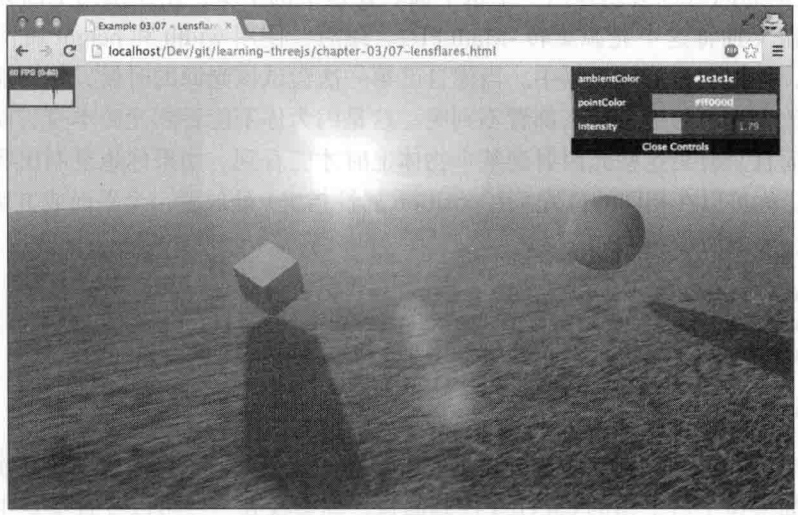
可以用 THREE.AreaLight 实例创建出非常漂亮的效果，但是可能要试验几次才能获得想要的效果。如果拉下右上角的 dat.GUI 控制面板，你会找到一些可以设置场景中三个光源颜色和光强的控件，如下图所示：



3.2.5.3 镜头眩光

本章将要探讨的最后一个主题是镜头眩光。你可能已经对镜头眩光比较熟悉了。例如，当你直接朝着太阳拍照时就会出现镜头眩光。大多数情况下需要避免这种情形，但是对于游戏和三维图片来说，它提供了一种很好的效果，让场景看上去更加真实。

Three.js 库也支持镜头眩光，而且在场景中添加眩光很简单。本节中我们将在场景中添加一个镜头眩光，创建出如下图所示的输出效果。可以打开示例 07-lensflares.html，查看这个效果。



可以通过实例化 THREE.LensFlare 对象来创建镜头眩光。首先要做的就是创建这个对象。THREE.LensFlare 对象接受如下参数：

```
THREE.LensFlare=function(texture, size, distance, blending,
color);
```

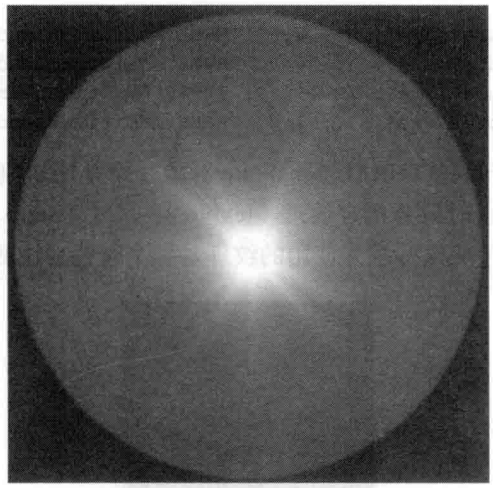
这些参数的含义列在下表中：

参　数	描　述
texture（纹理）	参数 texture 用作眩光的材质。决定眩光的样子
size（尺寸）	可以指定眩光应该多大。这个尺寸的单位是像素。如果指定的值是 -1，那就使用纹理本身的尺寸
distance（距离）	光源（0）到相机（1）的距离
blending（融合）	我们可以为眩光提供多种材质。融合模式决定它们将如何融合在一起。默认的融合模式是 THREE.AdditiveBlending，它可以提供一种半透明的眩光。更多内容将在下一章讲解
color（颜色）	眩光的颜色

让我们看看创建这个对象的代码（示例 07-lensflares.html）：

```
var textureFlare0 = THREE.ImageUtils.loadTexture
      ("../assets/textures/lensflare/lensflare0.png");

var flareColor = new THREE.Color(0xffaacc);
var lensFlare = new THREE.LensFlare(textureFlare0, 350, 0.0, THREE.
AdditiveBlending, flareColor);

lensFlare.position = spotLight.position;
scene.add(lensFlare);
```

我们需要先加载一个纹理。在这个示例里，我用的是 Three.js 库例子中的纹理，如下图所示：



如果你比较一下这个纹理和本节此前展示的那个截图，你会发现它定义的是眩光的样

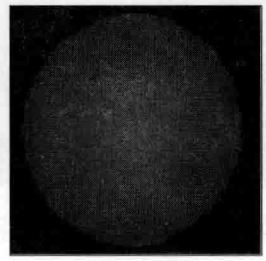
子。接下来，要使用 THREE.Color（0xffaacc）定义眩光的颜色。这将使眩光泛着红色。有了这两个对象之后，就可以创建 THREE.LensFlare 对象了。在这个示例里，把眩光的尺寸（size）设为 350，距离（distance）设为 0.0（就在光源处）。

创建完 THREE.LensFlare 对象之后，可以将它放在光源处，添加到场景中，如下图所示：

这看起来已经很好了，但是如果你把这张截图跟本章前面的截图比较一下，你会发现页面中央少了一些小的圆形失真图形。我们会使用创建眩光的方法来创建它们。

```
var textureFlare3 = THREE.ImageUtils.loadTexture
      ("../assets/textures/lensflare/lensflare3.png");

lensFlare.add(textureFlare3, 60, 0.6, THREE.AdditiveBlending);
lensFlare.add(textureFlare3, 70, 0.7, THREE.AdditiveBlending);
lensFlare.add(textureFlare3, 120, 0.9, THREE.AdditiveBlending);
lensFlare.add(textureFlare3, 70, 1.0, THREE.AdditiveBlending);
```

这次并没有创建一个新的 THREE.LensFlare 对象，而是用了刚创建的 THREE.LensFlare 对象的 add() 函数。调用它时只要指定纹理（texture）、尺寸（size）、距离（distance）和融合（blending）模式即可。为生成这些眩光使用的纹理是一个颜色很浅的圆，如下所示：



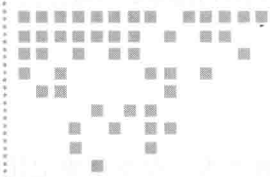
如果你再来看这个场景，就会发现失真图形会出现在用 distance 参数指定的地方。

3.3　总结

本章讲述的是 Three.js 库提供的各种光源，信息量非常大。而本章要记住的最重要的东西列在下面：

- 配置光源、颜色和阴影不是严谨的科学。需要不断试验；使用 dat.GUI 控件可以微调配置。
- 环境光光源（AmbientLight）的颜色可以附加到场景中的每一种颜色上。它没有位置的概念。通常用这种光源来柔化那些硬生生的颜色和阴影。
- 点光源（PointLight）并不生成阴影，而且可以朝所有方向发射光线。可以把它跟夜空中的照明弹相比对。
- 聚光灯光源（SpotLight）类似手电筒。发射出的光线是一个锥形，而且可以配置它随着距离的增大而逐渐变弱。聚光灯光源可以设定为生成阴影。
- 聚光灯光源跟方向光光源一样，都有一个 debug（微调）开关，可以用来微调阴影相机的配置。
- 方向光光源（DirectoionalLight）可以跟很远地方的光源（例如太阳）相对比。由于距离很远，所以其光线都是相互平行的。光线离指定的目标越远，则光强衰减得越多。
- 如果你想要一个更加自然的室外效果，可以使用半球光光源（HemisphereLight）。它可以将天空的光照和来自地面的反射光计算在内。
- 如果你想用平面光光源（AreaLight），你要记住用 WebGL 延迟渲染器（WebGLDeferredRenderer）对象。如果你有大量的光源，而且性能也有问题，你就应该考虑使用 WebGLDeferredRenderer 对象，而不是 WebGLRenderer。
- 要实现模拟照片中的眩光效果，可以使用 Three.js 库中的 LensFlare 组件，以便在场景中的光源上添加眩光效果。

到本章为止，我们已经介绍了几种材质，而且在本章你也可以看到各种材质对于光照的反应各不相同。下一章将会概述 Three.js 库中的各种材质。



Chapter 4 第 4 章

使用 Three.js 的材质

前面的章节简单提了一下材质。你已经知道了材质结合几何体可以构成网格。材质就像是物体的皮肤，决定几何体外表的样子。例如，皮肤可以决定一个几何体看起来是否像金属、透明与否，以及是否显示成线框。然后这个网格才可以添加到场景中，并由 Three.js 库来渲染。到目前为止，我们还没有很详细地讨论过材质。本章将深入探讨一下 Three.js 库提供的所有材质，你也将会从中学到如何运用这些材质来创建漂亮的三维对象。本章将要探讨的材质列在下表中：

名　称	描　述
MeshBasicMaterial（网格基础材质）	基础材质，可以用它赋予几何体一种简单的颜色，或者显示几何体的线框
MeshDepthMaterial（网格深度材质）	根据网格到相机的距离，这种材质决定如何给网格染色
MeshNormalMaterial（网格法向材质）	这是一种简单的材质，根据物体表面的法向向量计算颜色
MeshFaceMaterial（网格面材质）	这是一个容器，可以在这个容器里为物体的各个表面指定不同的颜色
MeshLambertMaterial（网格朗伯[⊖]材质）	这种材质会考虑光照的影响，可以用来创建颜色暗淡的、不光亮的物体
MeshPhongMaterial（网格 Phong[⊖]式材质）	这种材质也会考虑光照的影响，而且可以用来创建光亮的物体
ShaderMaterial（着色器材质）	这种材质允许使用自定义的着色器程序，直接控制顶点的放置方式，以及像素的着色方式

⊖ Lambert，朗伯，一种亮度单位。以德国数学家、天文学家和物理学家 John Heinrich Lambert（1728—1777 年）的名字命名。——译者注

⊖ Phong，一种着色方式，由美国越南裔学者裴祥风发明，于 1973 年的博士论文中首度发表。——译者注

（续）

名　称	描　述
LineBasicMaterial（直线基础材质）	这种材质可以用于 THREE.Line（直线）几何体，从而创建着色的直线
LineDashedMaterial（虚线材质）	这种材质跟直线基础材质一样，不过可以用来创建出一种虚线效果

这些材质有一些共同的属性，所以在我们讨论第一个材质（MeshBasicMaterial）之前，我们先来看一下所有材质都具有的属性。

4.1　理解共有属性

你可以快速看一下哪些属性是所有材质共有的。Three.js 提供了一个材质基类，THREE.Material，这个类列出了所有共有属性。我们已经把这些共有属性分成了三类，如下所示。

- 基础属性：这些属性是最常用到的。通过这些属性可以控制物体的透明度、是否可见或如何引用物体（通过 ID 或自定义名称）。
- 融合属性：每个物体都有一系列的融合属性。这些属性决定物体如何与背景融合。
- 高级属性：有一些高级属性可以控制底层 WebGL 上下文渲染物体的方法。大多数情况下，不会用到这些属性。

我们将从这个列表中的第一项开始：基础属性。

4.1.1　基础属性

THREE.Material 类的基础属性列在下表中。你可以在 4.2.1 节中看看如何运用这些属性。

属　性	描　述
ID（标识符）	用来标识材质，在创建时赋值
name（名称）	可以通过这个属性赋予材质名称
opacity（透明度）	定义物体有多透明。与属性 transparent 一起使用。该属性的取值范围是 0 ～ 1
transparent（是否透明）	如果设为 true，Three.js 库就会根据 opacity 的值来渲染物体。如果是 false，这个物体就不透明，只是着色明亮一些。
overdraw（过度描绘）	如果你用 THREE.CanvasRenderer（画布渲染器）对象，多边形会被渲染得稍微大一点儿。当你用这个渲染器画出来的物体有缝隙时，可以将这个属性设为 true
visible（是否可见）	定义该材质是否可见。如果将它设为 false，那么在场景中就看不到该物体
side（侧面）	通过这个属性，可以决定在几何体的哪一面应用这个材质。默认值是 THREE.FrontSide（前面），这可以将材质应用到物体的前（外）面。也可以将它设为 THREE.BackSide（后面），这可以将材质应用到物体的后（内）侧。或者也可以将它设为 THREE.DoubleSide（双侧），这样就可以将材质应用到物体的内外两侧
needsUpdate（要否刷新）	对于材质的某些修改，你需要告诉 Three.js 库材质已经改变了。如果这个属性设为 true，Three.js 就会使用新的材质属性刷新它的缓存

对于每个材质还可以设置一些融合属性。

4.1.2 融合属性

材质有几个跟融合相关的一般属性。稍后在讨论合并材质的时候，会再次探讨这个主题，但我们不会涉及很多细节。

名　称	描　述
blending（融合）	决定物体上的材质如何跟背景融合。融合模式一般是 NormalBlending，在这种模式下只显示材质的上层
blendsrc（融合源）	除了使用标准融合模式之外，还可以通过指定 blendsrc、blenddst 和 blendequation 属性来创建自定义的融合模式。该属性指定物体（源）如何跟背景（目标）相融合。默认值是 SrcAlphaFactor，即使用 alpha（透明度）通道进行融合
blenddst（融合目标）	该属性定义融合时如何使用背景（目标），默认值是 OneMinusSrcAlphaFactor，其含义是：目标也使用源的 alpha 通道进行融合，只是用的值是 1（源的 alpha 通道值）
blendingequation（融合公式）	指定如何使用 blendsrc 和 blenddst 的值。默认方法是 AddEquation，即将两个颜色值相加。使用这三个属性，就可以创建自定义的融合模式

最后一组属性大多在内部使用，用来控制 WebGL 渲染场景时的细节。

4.1.3 高级属性

我们不会深入探讨这些属性的细节。它们跟 WebGL 内部如何工作相关。如果你想更多地了解这些属性，那么 OpenGL 规范是个很好的起点。可以在下面这个网址找到此规范：http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf。

名　称	描　述
depthTest（深度测试）	这是一个高级 WebGL 属性。使用这个参数可以打开或关闭 GL_DEPTH_TEST 参数。该参数决定像素深度是否用来计算新的像素值。通常情况下不必修改这个属性。更多信息可以在前面提到的那个 OpenGL 规范中找到
depthWrite	这是另外一个内部属性。可以用来决定这个材质是否影响 WebGL 的深度缓存。如果你将一个物体用作二维贴图时（例如一个套子），你应该将这个属性设为 false。但是一般来讲，你不应该修改这个属性
polygonOffset、polygonOffsetFactor 和 polygonOffsetUnits	通过这些属性，可以控制 WebGL 的 POLYGON_OFFSET_FILL 功能。一般不需要使用它们。有关该功能的解释，可以参考 OpenGL 规范
alphaTest	可以给这个属性指定一个值（从 0 到 1）。如果某个像素的 alpha 值小于这个值，那么该像素就不会显示出来

现在让我们来看看所有可用的材质，你会看到这些属性对渲染结果的影响。

4.2　从简单的网格材质（基础、深度和面）开始

本节介绍几种简单的网格材质：MeshBasicMaterial、MeshDepthMaterial、MeshNormalMaterial 和 MeshFaceMaterial。我们将从 MeshBasicMaterial 开始。

4.2.1　简单表面的 MeshBasicMaterial

MeshBasicMaterial 是一种非常简单的材质，这种材质不考虑光照的影响。使用这种材质的网格会被渲染成一些简单的平面多边形，而且你也有机会显示几何体的线框。除了上一节提及的那些共有属性之外，还可以设置下面这些属性：

名　称	描　述
color	设置材质的颜色
wireframe	设置这个属性可以将材质渲染成线框。对调试非常有利
wireframeLinewidth（线框线宽）	如果已经打开了 wireframe，这个属性可以定义线框中线的宽度
wireframeLinecap（线框线端点）	这个属性定义线框模式下顶点间线段的端点如何显示。可选值包括 butt（平）、round（圆）和 square(方)。默认值是 round。在实际使用中，这个属性的修改结果很难看出来。WebGLRenderer 对象不支持该属性
wireframeLinejoin（线框线段连接点）	定义线段的连接点如何显示。可选的值包括 round、bevel（斜角）和 miter（尖角）。默认值是 round。如果你在一个使用低透明度和很大 wireframeLinewidth 值的例子里靠近观察，你就可以看到这个属性的效果。WebGLRenderer 不支持该属性
shading（着色）	该属性定义如何着色。可选的值是 THREE.SmoothShading 和 THREE.FlatShading。该属性在这个材质的例子里没有设置。可以在 4.2.4 节中找到相应的例子
vertexColors（顶点颜色）	可以通过这个属性为每一个顶点定义不同的颜色。该属性在使用 CanvasRenderer 时不起作用，但可以在使用 WebGLRenderer 时起作用。关于该属性的例子，可以参考 LineBasicMaterial 的例子。在这个例子里使用该属性为线段的不同部分赋予不同的颜色
fog（雾化）	该属性指定当前材质是否会受全局雾化效果设置的影响。我们在例子中没有用到该属性。不过要是设成 false，那么我们在第 2 章看到的全局雾化效果设置就不会影响当前对象的渲染

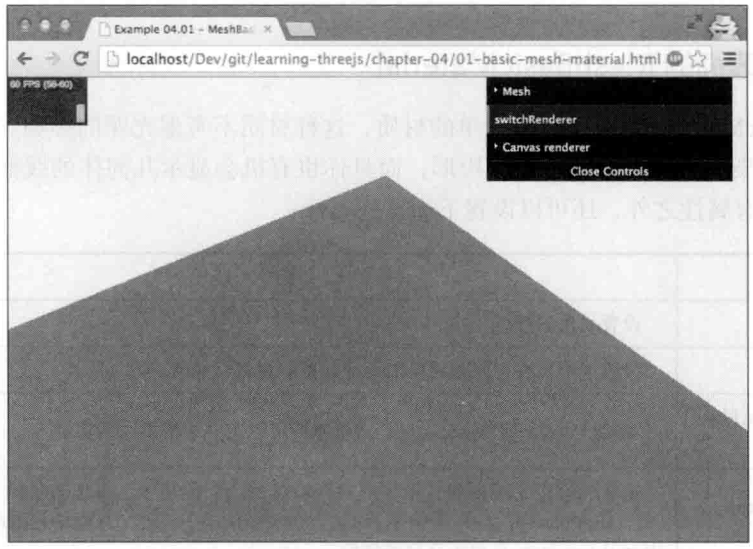
在前面的章节里我们已经看到如何创建材质，并把它们赋给某个对象。我们会像下面这样设置 MeshBasicMaterial：

```
var meshMaterial = new THREE.MeshBasicMaterial({color: 0x7777ff});
```

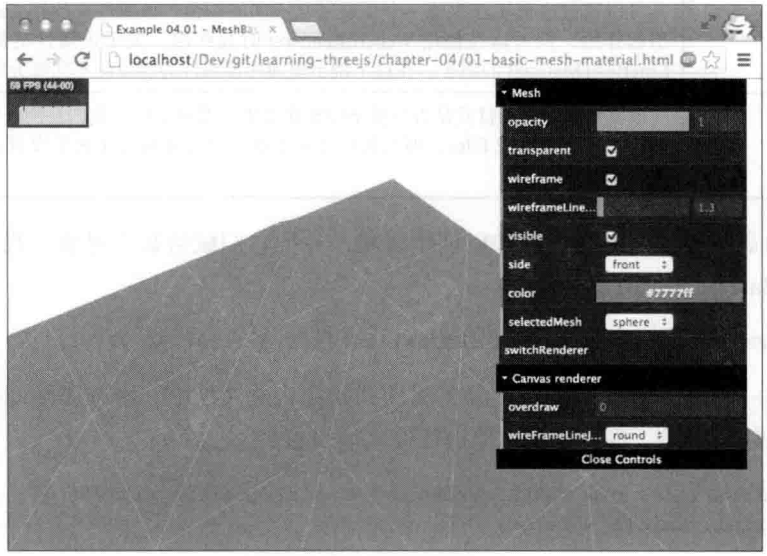
这会创建一种新的材质，并将其颜色属性初始化为 0x7777ff。所有属性即可以在像上面这样在构造函数里传递，也可以在材质创建好之后设置：

```
var meshMaterial = new THREE.MeshBasicMaterial({color: 0x7777ff});
meshMaterial.visible = false;
```

我已经添加了一个例子，你可以试验一下 MeshBasicMaterial 的属性，以及我们在上一节里说到的那些属性。打开文件夹 chapter-04 下面的例子 01-basic-mesh-material.html，你会看到一个旋转的方块，如下图所示：



这是一个非常简单的物体。可以通过右上角的菜单试验这些属性，选择各种类型的网格（甚至还可以修改渲染器）。假设有一个球体，opacity 设为 0.2，transparent 设为 true，wireframe 设为 true，wireframeLinewidth 是 9，并且使用 CanvasRenderer 对象，那么渲染结果如下所示：



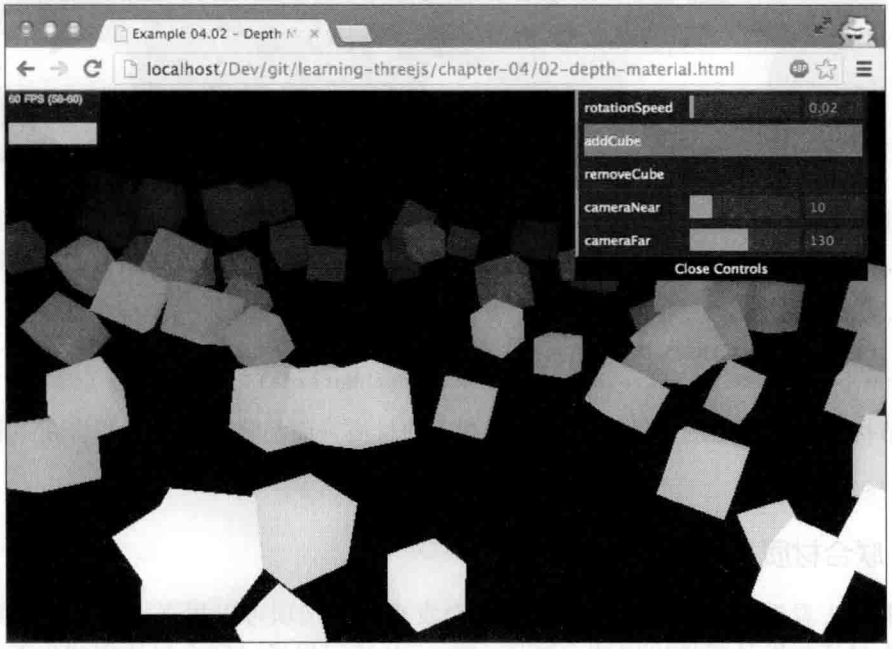
在这个例子里有一个可以设置的属性是 side。通过这个属性可以指定在几何体的哪一面上应用材质。如果选择的是 plane（平面）网格，就可以检验该属性是如何起作用的。由于材质往往是应用在物体前面的面上，所以在平面旋转时会有一半的时间看不到它（其实显示的是它的背面）。如果你把 side 属性设为 double，那么由于几何体的两面都有材质，所以这个平面始终都可以看见。

4.2.2　基于深度着色的 MeshDepthMaterial

列表中的下一种材质是 MeshDepthMaterial。使用这种材质的物体，其外观不是由光照或某个材质属性决定的；而是由物体到相机的距离决定的。可以将这种材质与其他材质相结合，从而很容易地创建出逐渐消失的效果。这种材质只有两个控制线框的属性：

名　称	描　述
wireframe	是否显示线框
wireframeLinewidth	线框线的宽度

为了展示该材质，我们修改了第 2 章中那个方块的例子，如下图所示（文件夹 chapter-04 中的 02-depth-material.html）。记住，必须点击 **addCube** 按钮才能在场景中添加方块。



尽管这种材质没有多少属性可以控制物体的渲染效果，但我们依然可以控制物体消失的速度。在这个例子里，将相机的 near 属性和 far 属性暴露出来。正如你所记得的，我们在第

2 章通过这两个属性控制相机可见的区域。所有跟相机距离小于 near 属性的物体不会显示出来，而所有跟相机距离大于 far 属性的物体也都落在相机的可见范围之外。

相机 near 属性和 far 属性之间的差距决定场景的亮度和物体消失的速度。如果这个差距非常大，那么当物体远离相机时，只会稍微消失一点。如果这个差距非常小，那么物体消失的效果将会非常明显，如下图所示：

创建 MeshDepthMaterial 非常简单。这个对象没有多少参数。在这个例子里，我们使用了 scene.overrideMaterial 属性，以保证场景中的所有物体都会使用该材质，如下面这段代码所示：

```
var scene = new THREE.Scene();
scene.overrideMaterial = new THREE.MeshDepthMaterial();
```

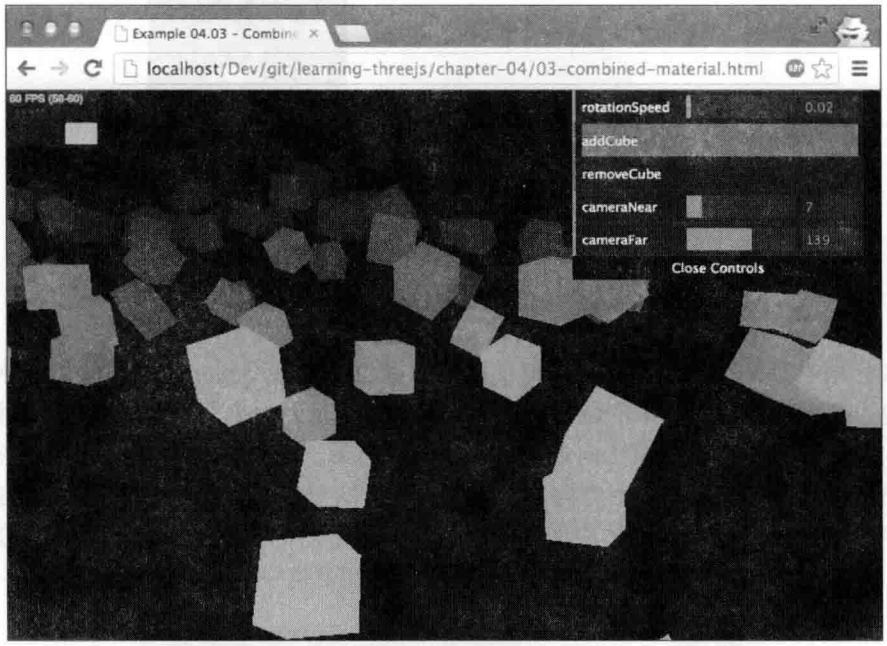
本节接下来要谈论的主题并不是一种真正的材质，而是将各种材质联合在一起的一种方法。

4.2.3 联合材质

如果回头看看 MeshDepthMaterial，你会发现没有选项可以用来设置方块的颜色。一切都是由材质的默认属性决定的。但是，Three.js 库可以通过联合材质创建出新效果（这也是材质融合起作用的地方）。如果使用下面的代码，可以把 MeshDepthMaterial 材质赋予方块：

```
var cubeMaterial = new THREE.MeshDepthMaterial();
var colorMaterial = new THREE.MeshBasicMaterial({color: 0x00ff00,
            transparent: true, blending: THREE.MultiplyBlending})
var cube = new THREE.SceneUtils.createMultiMaterialObject(
            cubeGeometry, [colorMaterial, cubeMaterial]);
cube.children[1].scale.set(0.99, 0.99, 0.99);
```

然后就可以获得下图所示的绿色方块，这些方块可以从 MeshDepthMaterial 对象获得渐变效果，从 MeshBasicMaterial 对象获得颜色（参照例子 08-combined-material.html）。

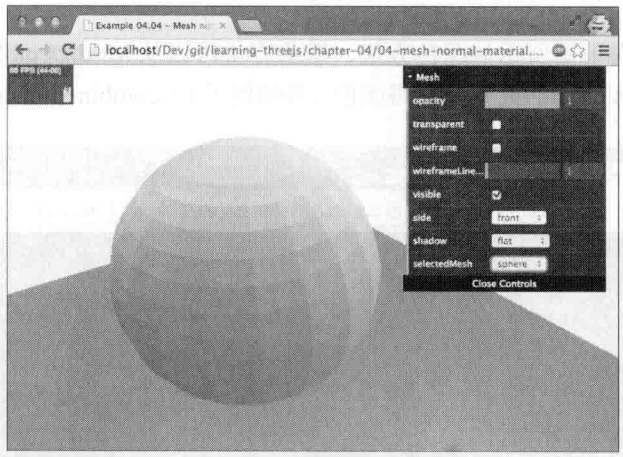


让我们来看看要达到这种效果需要采取的步骤。首先要创建两种材质。对于 MeshDepthMaterial 没有什么特别要做的；但是对于 MeshBasicMaterial，要把 transparent 属性设为 true，并指定融合模式。如果不把 transparent 属性设为 true，就只会得到一些纯绿色的物体，因为 Three.js 库不会执行任何融合操作。如果将 transparent 属性设为 true，Three.js 就会检查 blending 属性，来看看这个绿色的 MeshBasicMaterial 材质如何跟背景相互作用。这里所说的背景是用 MeshDepthMaterial 渲染的方块。第 9 章会详细讨论各种可以选用的融合模式。在这个例子里用的是 THREE.MultiplyBlending 对象。这种模式会把前景的颜色跟背景的颜色相乘，给你想要的结果。上述代码片段的最后一行也很重要。当调用 createMultiMaterialObject() 函数创建一个网格的时候，几何体会被复制，返回一个网格组，里面两个网格完全相同。如果没有最后一行，那么在渲染的时候画面会有闪烁，因为它们会直接在彼此的上面渲染。通过缩小带有 MeshDepthMaterial 材质的网格，就可以避免这种现象。

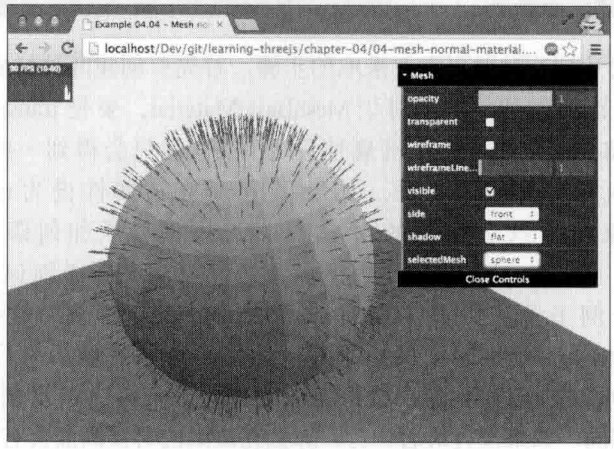
下一种材质也不会对渲染时使用的颜色有任何影响。

4.2.4 计算法向颜色的 MeshNormalMaterial

要理解这种材质的功能，最简单的方法是看一个例子。打开 chapter-04 文件夹下的例子 03-mesh-normal-material.html。如果选择 sphere（球体）网格，你就会看到类似下面的屏幕截图：



正如你所看到的，网格上的每一面在渲染时颜色都稍有不同，而且即便在球体旋转时，这些颜色也基本上保持在原来的位置。之所以是这样，是因为每一个面的颜色是从该面向外指的法向量计算得到的。所谓法向量是与面垂直的向量。法向量在 Three.js 库中有很广泛的应用。它可以用来决定光的反射方向，在三维物体上映射材质时起辅助作用，还可以在计算光照、阴影时提供信息，为物体表面像素上色。但是幸运的是，Three.js 库会处理这些向量的计算，而且在库内部使用。下面这个屏幕截图显示的就是一个法向量的例子：



法向量所指的方向决定每个面从 MeshNormalMaterial 材质获取的颜色。由于球体各面的法向量都不相同，所以在这个例子里我们看到的是一个色彩斑斓的球。

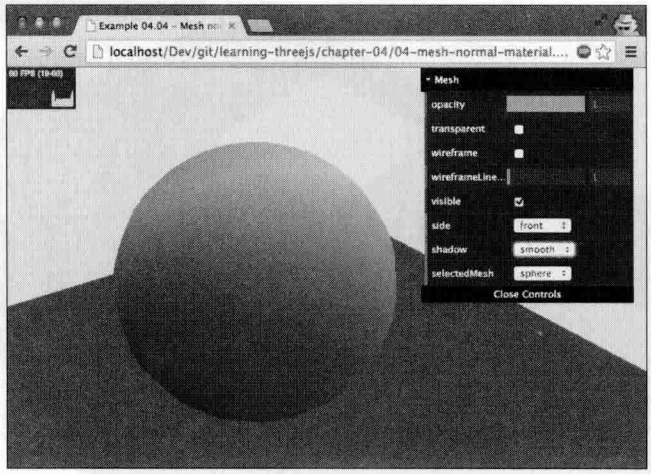
要添加这些表示法向量的箭头，可以使用 THREE.ArrowHelper 对象，如以下代码片段所示：

```
for (var f = 0, fl = sphere.geometry.faces.length; f <
fl; f++) {
    var face = sphere.geometry.faces[ f ];
    var arrow = new THREE.ArrowHelper(
            face.normal,
            face.centroid,
            2,
            0x3333FF);
    sphere.add(arrow);
}
```

这段代码在球体的每个面上添加一个长度为 2、颜色为 0x3333ff 的箭头，表示该面的法向量。MeshNormalMaterial 对象还有几个属性可以设置，如下表所示：

名　称	描　述
wireframe	该属性指定是否显示线框
wireframeLinewidth	指定线框线的宽度
shading	该属性用来设置着色方法：THREE.FlatShading 表示平面着色，THREE.SmoothShading 表示平滑着色

我们已经见识过 wireframe 和 wireframeLinewidth 属性了，但是在关于 MeshBasicMaterial 的那个例子里，我们略过了 shading 属性。通过 shading 属性，我们可以告诉 Three.js 库如何渲染物体。如果使用 THREE.FlatShading 对象，那么每个面是什么颜色就会渲染成什么颜色（正如你在前面那个截图中看到的那样），或者你也可以使用 THREE.SmoothShading 对象，这样就可以使物体的表面变得光滑些。举例来说，如果我们使用 THREE.SmoothShading 对象来渲染球体，得到的结果如下图所示：



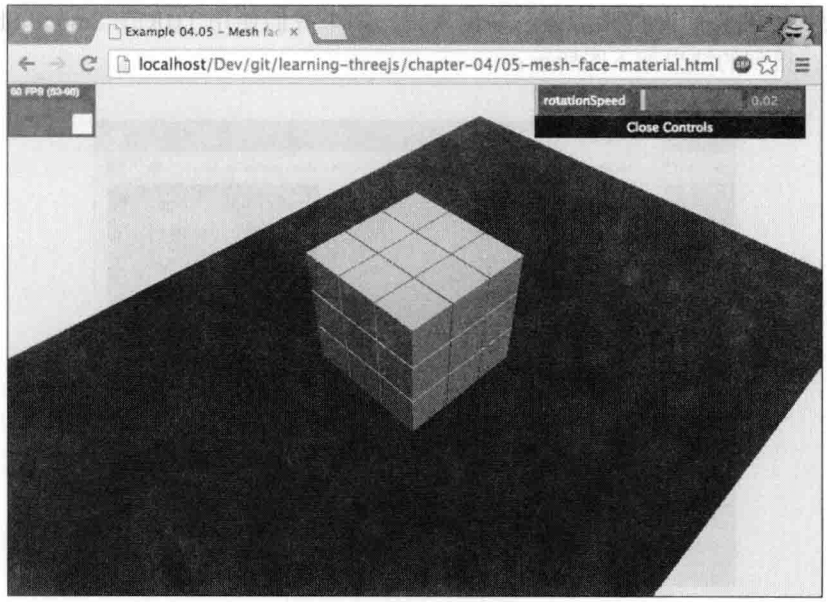
简单材质快要讲完了，最后一种是 MeshFaceMaterial。

4.2.5 为每个面指定材质的 MeshFaceMaterial

最后这种基础材质并不是真正的材质，更像是一种材质容器。通过 MeshFaceMaterial 可以为几何体的每一个面指定不同的材质。假如你有一个方块，上面有六个面，你可以用这种材质来为每个面指定一种材质（例如不同的颜色）。这种材质使用起来很简单，如下所示：

```
var matArray = [];
matArray.push(new THREE.MeshBasicMaterial( { color: 0x009e60 }));
matArray.push(new THREE.MeshBasicMaterial( { color: 0x0051ba }));
matArray.push(new THREE.MeshBasicMaterial( { color: 0xffd500 }));
matArray.push(new THREE.MeshBasicMaterial( { color: 0xff5800 }));
matArray.push(new THREE.MeshBasicMaterial( { color: 0xC41E3A }));
matArray.push(new THREE.MeshBasicMaterial( { color: 0xffffff }));

var faceMaterial = new THREE.MeshFaceMaterial(materialArray);

var cubeGeom = new THREE.CubeGeometry(3,3,3);
var cube = new THREE.Mesh(cubeGeom, faceMaterial);
```

我们先创建了一个名为 matArray 的数组，用来保存各个材质。接下来我们为每个面创建一种不同颜色的材质（本例用的是 THREE.MeshBasicMaterial）。然后用这个数组来实例化 THREE.MeshFaceMaterial，再结合方块的几何体来构建一个网格。让我们来深入研究一下这段代码，来看看要再现下面的例子（一个魔方）还需要做些什么。可以在文件夹 chapter-04 里找到这个例子（04-mesh-face-material.html）：



魔方由一些小方块组成：沿 x 轴 3 个，沿 y 轴 3 个，沿 z 轴 3 个，代码如下：

```
var group = new THREE.Mesh();
// add all the rubik cube elements
var mats = [];
mats.push(new THREE.MeshBasicMaterial({ color: 0x009e60 }));
mats.push(new THREE.MeshBasicMaterial({ color: 0x0051ba }));
mats.push(new THREE.MeshBasicMaterial({ color: 0xffd500 }));
mats.push(new THREE.MeshBasicMaterial({ color: 0xff5800 }));
mats.push(new THREE.MeshBasicMaterial({ color: 0xC41E3A }));
mats.push(new THREE.MeshBasicMaterial({ color: 0xffffff }));
var faceMaterial = new THREE.MeshFaceMaterial(mats);

for (var x = 0; x < 3; x++) {
    for (var y = 0; y < 3; y++) {
        for (var z = 0; z < 3; z++) {
            var cubeGeom = new THREE.CubeGeometry(2.9, 2.9, 2.9);
            var cube = new THREE.Mesh(cubeGeom, faceMaterial);
            cube.position =
                new THREE.Vector3(x * 3 - 3, y * 3, z * 3 - 3);
            group.add(cube);
        }
    }
}
```

在这段代码里先创建了一个 MeshFaceMaterial。接着会创建三个循环，以保证创建出正确数目的方块。在循环里会创建每一个方块，赋予材质，定位，并把它们添加到一组（group）中。你应该记住的是，方块的位置是与这个组之间的相对位置。如果移动或旋转这个组，所有方块都会随着它移动和旋转。有关如何使用组的更多信息，请参考第 8 章。

如果在浏览器中打开这个例子，你会看到整个魔方在旋转（各个单独方块并没有旋转）。这是因为我们在渲染循环里添加了如下代码：

```
group.rotation.y=step+=0.01;
```

这会使得整个组绕着它的中心（0,0,0）旋转。我们在定位每个方块时，确保它们放在这个中心点的周围。这就是你为什么会在代码（cube.position=new THREE.Vector3（x*3 – 3, y*3, z*3 – 3））中看到 –3 这个偏置。

MeshFaceMaterial 是最后一种基础材质。下一节介绍 Three.js 库里的高级材质。

4.3　学习高级材质

这一节介绍 Three.js 库提供的高级材质。我们先看一下 MeshPhongMaterial，然后是 MeshLabmertMaterial。这两种材质会对光源做出反应，可以分别用来创建光亮的材质和暗淡的材质。本节还会介绍一种最通用但也最难用的材质：ShaderMaterial。通过 ShaderMaterial，你可以创建自己的着色程序，定义材质和物体如何显示。对于本节最后一个关于高级材质的

主题，我们会来看一看如何创建一个拥有多种材质的网格，而不是仅仅拥有一种材质。

4.3.1 用于暗淡、不光亮表面的 MeshLambertMaterial

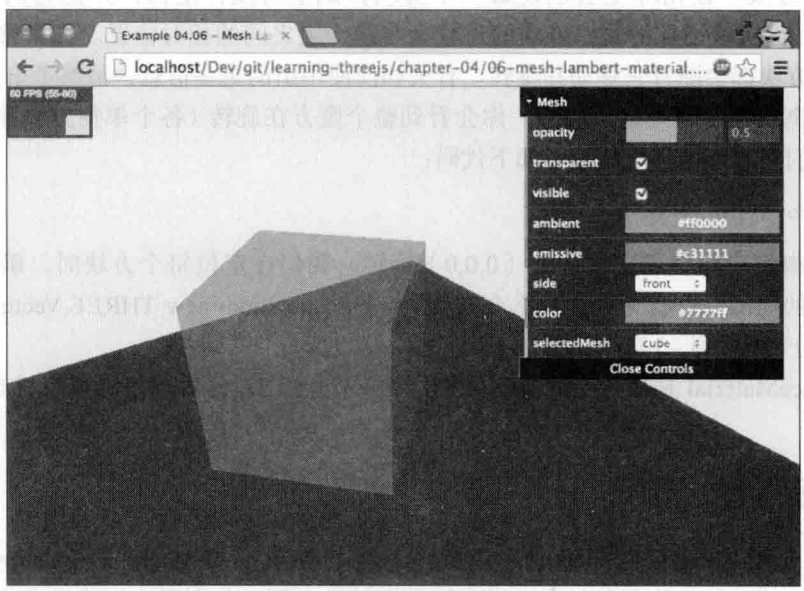
这种材质可以用来创建看上去暗淡的并不光亮的表面。该材质非常易用，而且会对场景中的光源产生反应。可以在这个材质上配置前面提到的几个属性：color、opacity、shading、blending、depthTest、depthWrite、wireframe、wireframeLinewidth、wireframeLinecap、wireframeLinejoin、vertexColors 以及 fog。我们不会解释这些属性的细节，但是会重点讨论这种材质独有的一些属性。剩下的属性就只有下面两个了：

名　称	描　述
ambient （环境色）	这是该材质的环境色。跟上一章讲过的 AmbientLight 光源一起使用。这个颜色会与 AmbientLight 光源的颜色相乘。默认值是白色
emissive （发射的）	这是该材质发射的颜色。它其实并不像一个光源，只是一种纯粹的、不受其他光照影响的颜色。默认值是黑色

这个材质的创建跟其他材质都一样，如下所示：

```
var meshMaterial = new THREE.MeshLambertMaterial({color:0x7777ff});
```

关于这个材质的例子，可以参考 05-mesh-lambert-material.html；你会看到类似下面截屏的结果：



从截图中你可以看到，这个材质看上去比较暗淡。我们可以用另一种材质创建比较光亮的物体表面。

4.3.2　用于光亮表面的 MeshPhongMaterial

通过 MeshPhongMaterial，可以创建一种光亮的材质。可以使用的属性跟创建暗淡材质的 MeshLambertMaterial 基本一样。我们照样略过那些基础的、已经讨论过的属性：color、opacity、shading、blending、depthTest、depthWrite、wireframe、wireframeLinewidth、wireframeLinecap、wireframeLinejoin 和 vertexColors。

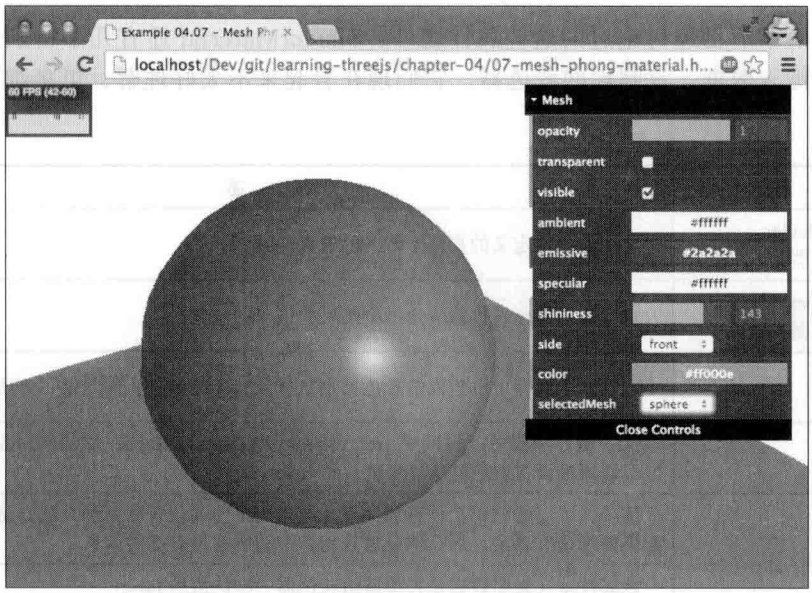
对于这种材质我们感兴趣的属性列在下表中：

名　称	描　述
ambient	这是该材质的环境色。跟上一章讲过的 AmbientLight 光源一起使用。这个颜色会与 AmbientLight 光源的颜色相乘。默认是白色
emissive	这是该材质发射的颜色。它其实并不像一个光源，只是一种纯粹的、不受其他光照影响的颜色。默认值是黑色
specular（镜面的）	该属性指定该材质的光亮程度及其高光部分的颜色。如果将它设置成跟 color 属性相同的颜色，将会得到一种更加类似金属的材质。如果设置成 grey（灰色），材质将变得更像塑料
shininess	该属性指定高光部分的亮度。默认值是 30

MeshPhongMaterial 对象的初始化跟我们看过的其他所有材质都一样：

```
var meshMaterial = new THREE.MeshPhongMaterial({color: 0x7777ff});
```

为了方便比较，我们为这个材质创建了一个跟 MeshLambertMaterial 那个例子一样的例子。可以通过控制界面来试验一下这个材质。例如，下图中的设置可以创建出一种看起来像是塑料的材质。可以打开 06-mesh-phong-material.html 来看看这个例子。



最后一种高级材质（ShaderMaterial）是最复杂的。

4.3.3 用 ShaderMaterial 创建自己的着色器

THREE.ShaderMaterial 是 Three.js 库中功能最丰富、最复杂的一种材质。通过它，可以使用自己定制的着色器，直接在 WebGL 环境中运行。着色器可以将 Three.js 中的 JavaScript 对象转换为屏幕上的像素。通过这些自定义的着色器，你可以明确指定你的对象如何渲染和遮盖，或者修改 Three.js 库中的默认值。本节不会涉及如何定制着色器的细节，更多信息可以参考第 11 章。现在我们只看一个非常基础的示例，展示一下如何设置这种材质。

ShaderMaterial 有几个属性可以设置，我们已经看过的如下表所示：

名　称	描　述
wireframe	设置这个属性可以将材质渲染成线框。对调试非常有利
wireframeLinewidth（线框线宽）	如果已经打开了 wireframe 属性，这个属性可以定义线框中线的宽度
shading	该属性定义如何着色。可选值为 THREE.SmoothShading 和 THREE.FlatShading。该属性在这个材质的例子里没有设置。可以在 4.2.4 节中找到相应的例子
vertexColors	可以通过这个属性为每一个顶点定义不同的颜色。该属性在使用 CanvasRenderer 时不起作用，但可以在使用 WebGLRenderer 时起作用。关于该属性的例子，可以参考 LineBasicMaterial 的例子。我们在这个例子里使用该属性为线段的不同部分赋予不同的颜色
fog	该属性指定当前材质是否会受全局雾化效果设置的影响。我们在例子中没有用到该属性。不过要是设成 false，那么我们在第 2 章看到的全局雾化效果设置就不会影响当前对象的渲染

除了前面章节已经讨论过的这些属性之外，ShaderMaterial 还有几个特别属性，使用它们你可以传入数据，定制你的着色器。它们现在看起来不太好理解，更多细节可以参考第 9 章。

名　称	描　述
fragmentShader（像素着色器）	这个着色器定义的是每个传入的像素的颜色
vertexShader（顶点着色器）	这个着色器允许你修改每一个传入的顶点的位置
uniforms（统一值）	通过这个属性可以向你的着色器发信息。同样的信息会发到每一个顶点和片段
defines	这个属性的值可以转换成 vertexShader 和 fragmentShader 里的 #define 代码。该属性可以用来设置着色器程序里的一些全局变量
attributes	该属性可以修改每个顶点和片段。通常用来传递位置数据和与法向量相关的数据。如果要用这个属性，那么你要为几何体中的所有顶点提供信息
lights	该属性定义光照数据是否传递给着色器。默认值是 false

在我们看例子之前，我们先简要解释一下 ShaderMaterial 里最重要的部分：要使用这个材质，必须要传入两个不同的着色器。

- vertexShader：vertexShader 会在几何体的每一个顶点上执行。可以用这个着色器通过改变顶点的位置来对几何体进行变换。
- fragmentShader：fragmentShader 会在几何体的每一个像素上执行。在 vertexShader 里，我们会返回这个特定像素应该显示的颜色。

到目前为止，对于本章讨论过的所有材质，Three.js 库都提供了 framentShader 和 vertexShader，所以你不必担心。

本节会给出一个简单的例子，这个例子会创建一种动态材质，其中会用到一个非常简单的 vertexShader，用来修改一个方块各个顶点的 x、y、z 坐标，还会用到一个 fragmentShader，使用来自 glsl.heroku.com 的着色器，创建连续变化的材质。

接下来你会看到我们使用 vertexShader 的完整代码。

着色器不是用 JavaScript 编写的。只能用类似 C 的 GLSL 语言来写着色器。

```
<script id="vertex-shader" type="x-shader/x-vertex">
    uniform float time;

    void main()
    {
    vec3 posChanged = position;
    posChanged.x = posChanged.x*(abs(sin(time*1.0)));
    posChanged.y = posChanged.y*(abs(cos(time*1.0)));
    posChanged.z = posChanged.z*(abs(sin(time*1.0)));

    gl_Position = projectionMatrix
                  * modelViewMatrix
                  * vec4(posChanged,1.0);
    }
</script>
```

我们不会在这里解释太多细节，只关注一下这段代码中最重要的部分。为了能够在 JavaScript 中跟着色器通信，我们会使用所谓的统一值（uniform）。在这个例子里使用语句 "uniform float time;" 传入外部数据。根据这个数据，我们会改变传入顶点的 x、y、z 坐标（通过 position 变量传入）：

```
vec3 posChanged = position;
posChanged.x = posChanged.x*(abs(sin(time*1.0)));
posChanged.y = posChanged.y*(abs(cos(time*1.0)));
posChanged.z = posChanged.z*(abs(sin(time*1.0)));
```

现在向量 posChanged 包含的就是顶点的新坐标，根据传入的 time 变量计算得到。我们要做的最后一步是将这个新坐标传回给 Three.js 库，如下所示：

```
gl_Position = projectionMatrix * modelViewMatrix
* vec4(posChanged,1.0);
```

gl_Position 是一个特殊变量，用来返回最终的位置。

接下来要创建一个 shaderMaterial，并传入这个 vertexShader。出于这个目的，创建了一个简单的辅助函数：

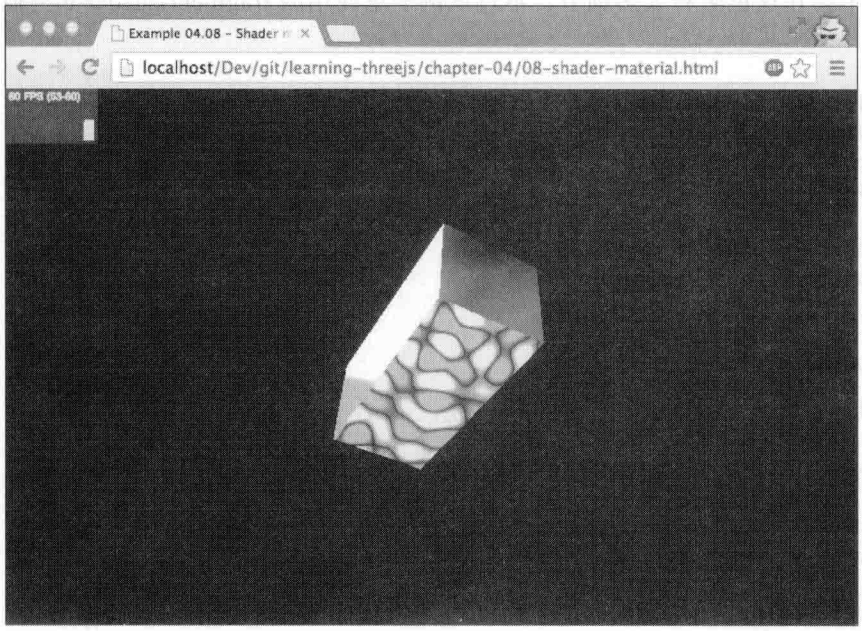
```
function createMaterial(vertexShader, fragmentShader) {
    var vertShader =
            document.getElementById(vertexShader).innerHTML;
    var fragShader =
            document.getElementById(fragmentShader).innerHTML;

    var attributes = {};
    var uniforms = {
        time: {type: 'f', value: 0.2},
        scale: {type: 'f', value: 0.2},
        alpha: {type: 'f', value: 0.6},
        resolution: { type: "v2", value: new THREE.Vector2() }
    };

    uniforms.resolution.value.x = window.innerWidth;
    uniforms.resolution.value.y = window.innerHeight;

    var meshMaterial = new THREE.ShaderMaterial({
        uniforms: uniforms,
        attributes: attributes,
        vertexShader: vertShader,
        fragmentShader: fragShader,
        transparent: true

    });
    return meshMaterial;
}
```

这个函数的用法是这样的：var meshMaterial1 = createMaterial ("vertex-shader", "fragment-shader-1");。这些参数所指的是 HTML 页面中脚本元素的 ID。在这里你也可以看到我们创建了一个 uniforms 变量。这个变量是用来从我们的渲染器中向着色器传递信息的。这个例子的完整渲染循环如下所示：

```
function render() {
    stats.update();

    cube.rotation.y = step += 0.01;
    cube.rotation.x = step;
    cube.rotation.z = step;

    cube.material.materials.forEach(function (e) {
```

```
        e.uniforms.time.value += 0.01;
    });

    // render using requestAnimationFrame
    requestAnimationFrame(render);
    renderer.render(scene, camera);
}
```

你可以看到这个渲染循环每执行一次变量 time 就会增加 0.01。这个信息会传递给我们的 vertexShader，用来计算方块顶点的新位置。现在打开例子 07-shader-material.html，你会看到这个方块沿着它的轴收缩、膨胀，如下图所示：



在这个例子里你可以看到方块的每一个面都在不断变化。正是每个面上的 fragementShader 造就了这种变化。正如你所猜想的，这里用到了 MeshFaceMaterial，如下所示：

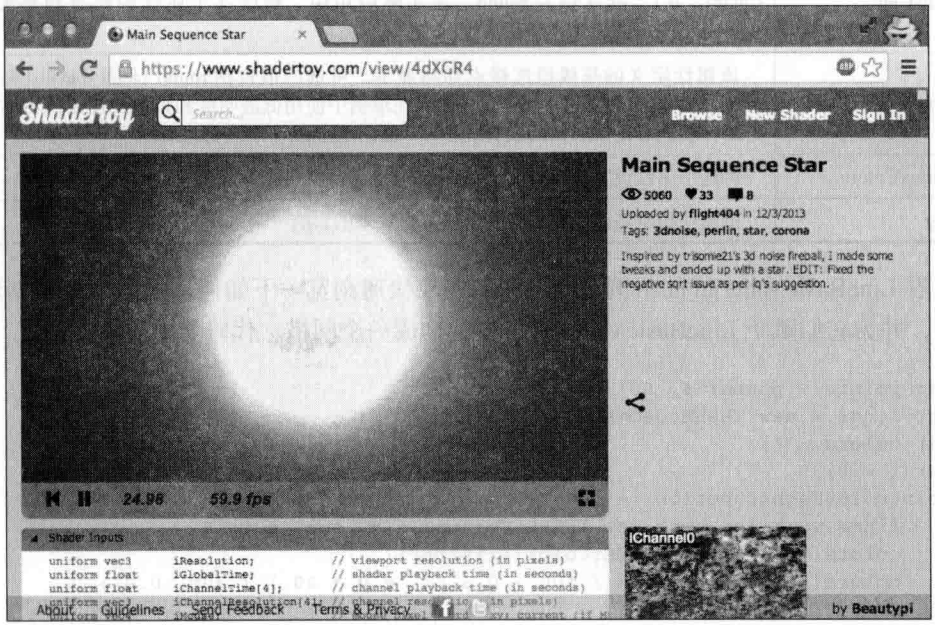
```
var cubeGeometry = new THREE.CubeGeometry(20, 20, 20);

var meshMaterial1 = createMaterial("vertex-shader", "fragment-
shader-1");
var meshMaterial2 = createMaterial("vertex-shader",
                                   "fragment-shader-2");
var meshMaterial3 = createMaterial("vertex-shader",
                                   "fragment-shader-3");
var meshMaterial4 = createMaterial("vertex-shader",
                                   "fragment-shader-4");
var meshMaterial5 = createMaterial("vertex-shader",
```

```
                                        "fragment-shader-5");
var meshMaterial6 = createMaterial("vertex-shader",
                                        "fragment-shader-6");

var material = new THREE.MeshFaceMaterial([meshMaterial1,
                    meshMaterial2, meshMaterial3, meshMaterial4,
                    meshMaterial5, meshMaterial6]);

var cube = new THREE.Mesh(cubeGeometry, material);
```

唯一还没有解释的部分是 fragmentShader。在这个例子里，所有像素着色器都是从 http://glsl.heroku.com 复制的。这个网站提供了一个试验环境，可以在这里创建并分享着色器。我不会在这里解释太多细节，不过该例子里使用的 fragment-shader-6 类似下面的代码段：

```
<script id="fragment-shader-6" type="x-shader/x-fragment">
    #ifdef GL_ES
    precision mediump float;
    #endif

    uniform float time;
    uniform vec2 resolution;


    void main( void )
    {

    vec2 uPos = ( gl_FragCoord.xy / resolution.xy );

    uPos.x -= 1.0;
    uPos.y -= 0.5;

    vec3 color = vec3(0.0);
    float vertColor = 2.0;
    for( float i = 0.0; i < 15.0; ++i ) {
    float t = time * (0.9);

    uPos.y += sin( uPos.x*i + t+i/2.0 ) * 0.1;
    float fTemp = abs(1.0 / uPos.y / 100.0);
    vertColor += fTemp;
    color += vec3( fTemp*(10.0-i)/10.0
                  ,fTemp*i/10.0, pow(fTemp,1.5)*1.5 );
    }

    vec4 color_final = vec4(color, 1.0);
    gl_FragColor = color_final;
    }
</script>
```

最终传回给 Three.js 库的颜色是通过语句 "gl_FragColor = color_finial;" 设置的。要获得像素着色器的更多感性认识，可以研究一下 http://glsl.heroku.com 上提供的那些着色器，并在你自己的项目中应用这些代码。在讨论下一种材质之前，再来看一个例子，看看自定义的顶点着色器（https://www.shadertoy.com/view/4dXGR4）能做些什么：



更多关于像素着色器和顶点着色器的内容可以在第 11 章中找到。

4.4　线段几何体的材质

我们将要看的最后两种材质只能用于特定的几何体：THREE.Line（线段）。顾名思义，线段上只有顶点，不包含任何面。Three.js 库提供了两个用于线段的不同材质，如下所示。

- LineBasicMaterial：通过线段基础材质可以设置线段的颜色、宽度、端点和连接点属性。
- LineDashedMaterial：跟 LineBasicMaterial 的属性一样，但是通过指定短划线和空格的长度，可以创建出虚线效果。

我们从基础的变体开始，然后再来看看虚线的变体。

4.4.1　LineBasicMaterial

这种用于 THREE.Line 的材质非常简单。下表列的是这种材质中可用的属性：

名　称	描　述
color	指定线的颜色。如果指定 vertexColors，这个属性就会被忽略
linewidth	该属性定义线的宽度
LineCap	该属性定义顶点间的线段端点如何显示。可选的值包括 butt（平）、round（圆）和 square（方）。默认值是 round。在实际应用中，修改这个属性的结果很难看出来。WebGLRenderer 也不支持该属性
LineJoin	该属性定义的是线段连接点如何显示。可选的值包括 round（圆）、bevel（斜切）和 miter（尖角）。默认值是 round。如果在示例中使用低透明度和很粗的线条，当你靠得很近时你就能看见该属性的设置效果。WebGLRenderer 不支持该属性
vertexColors	将这个属性设置成 THREE.VertexColors 值，就可以为每个顶点指定一种颜色
fog	该属性指定当前物体是否受全局雾化效果的影响

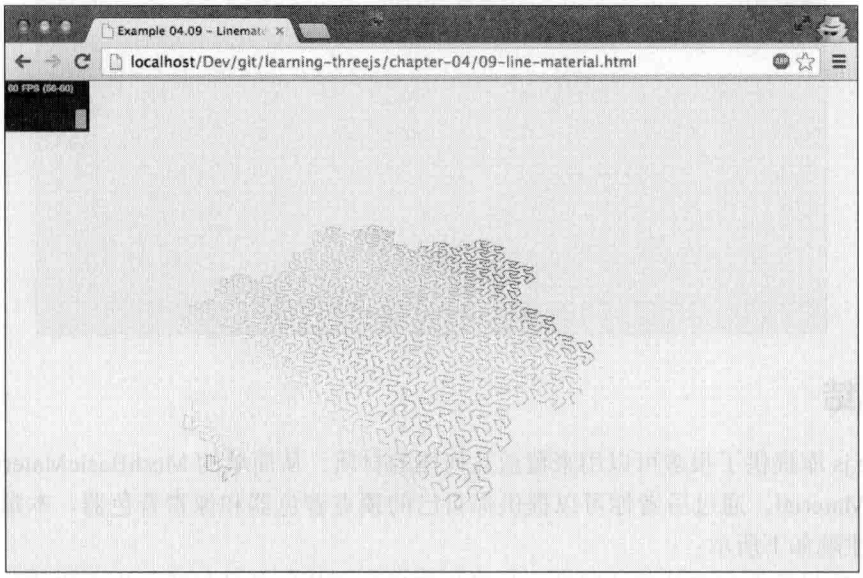
在看 LineBasicMaterial 的示例之前，我们先来快速浏览一下如何从顶点集合中创建出一些线段，并为它们赋予 LineBasicMaterial 材质，构成一个网格。代码如下所示：

```
var points = gosper(4, 60);
var lines = new THREE.Geometry();
var colors = [];
var i = 0;
points.forEach(function (e) {
    lines.vertices.push(new THREE.Vector3(e.x, e.z, e.y));
    colors[ i ] = new THREE.Color(0xffffff);
    colors[ i ].setHSL(e.x / 100 + 0.5, (  e.y * 20 ) / 300, 0.8);
    i++;
});

lines.colors = colors;
var material = new THREE.LineBasicMaterial({
    opacity: 1.0,
    linewidth: 1,
    vertexColors: THREE.VertexColors });

var line = new THREE.Line(lines, material);
```

这段代码的第一行 "var points = gosper（4，60）;" 作为示例，获取一组 x、y 坐标。这个函数返回的是一个 Gosper 曲线（更多信息，可以参考 http://en.wikipedia.org/wiki/Gosper_curve），这是一种填充二维空间简单的算法。接下来我们要做的是创建一个 THREE.Geometry 实例，为每个坐标创建一个顶点，并把它们放进该实例的线段属性中。对于每个坐标我们还会计算一个颜色值，用来设置 colors 属性。

在这个例子里用 setHSL() 方法设置颜色。使用 HSL，要提供色调（hue）、饱和度（saturation）和亮度（lightness），而不是红、绿、蓝的值。用 HSL 比用 RGB 更加直观，而且也更容易创建出匹配的颜色集合。你可以在 CSS3 规范里找到关于 HSL 的解释，网址是 http://www.w3.org/TR/2003/CRcss3-color-20030514/#hsl-color。

既然几何体已经准备好了，我们就可以创建一个 LineBasicMaterial，结合这个几何体可以创建 THREE.Line 网格。你可以在示例 09-line-material.html 看到结果，如下图所示：



本章将要讨论的下一个也是最后一个材质跟 LineBasicMaterial 稍有不同。通过 LineDashedMaterial，可以给线段上色，还可以添加一种虚线效果。

4.4.2 LineDashedMaterial

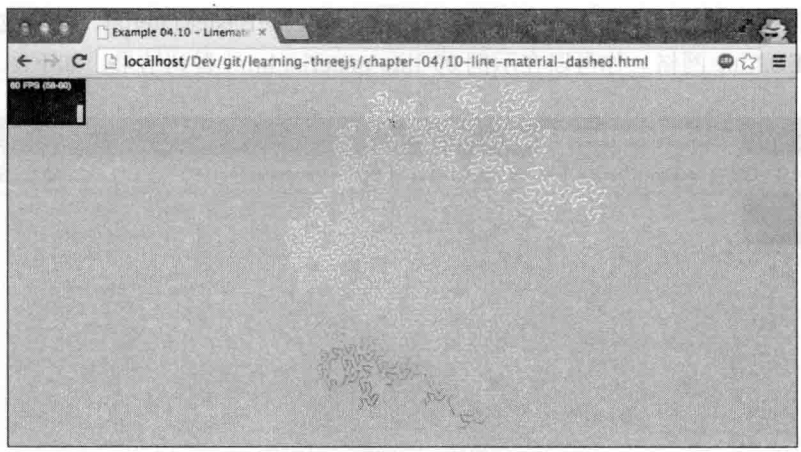
这种材质有跟 LineBasicMaterial 一样的属性，以及几个额外的、可以用来定义短划线长度和短划线中间空格长度的属性，如下所示：

名　称	描　述
scale （缩放比例）	缩放 dashSize 和 gapSize。如果 scale 小于 1，dashSize 和 gapSize 就会增大；如果 scale 大于 1，dashSize 和 gapSize 就会缩小
dashSize	短划线的长度
gapSize	间隔的长度

这个属性的用法跟 LineBasicMaterial 基本一样，如下所示：

```
lines.computeLineDistances();
var material = new THREE.LineDashedMaterial({ vertexColors: true,
        color: 0xffffff, dashSize: 10, gapSize: 1, scale: 0.1 });
```

唯一的区别是必须调用方法 computeLineDistances()。如果不调用，间隔就不会显示出来。你可以在 10-line-material-dashed.html 找到这种材质的例子，运行效果如下图所示：



4.5 总结

Three.js 库提供了很多可以用来覆盖几何体的材质，从简单的 MeshBasicMaterial 到复杂的 ShaderMaterial。通过后者你可以提供你自己的顶点着色器和像素着色器。本章讨论过的最重要的主题如下所示：

- 各种材质有很多共同的属性。如果你知道如何使用某种材质，你也应该知道如何使用其他材质。
- 并不是所有材质都会对场景中的光源做出反应。如果你希望一种材质计算光照的影响，你应该使用 MeshPhongMaterial 或者 MeshLambertMaterial。
- 如果要创建一种透明的材质，仅仅设置 opacity 属性是不够的，还需要将 transparent 属性设成 true。
- 材质的大部分属性都可以在运行时修改，但是有一些属性（例如 side）不可以在运行时修改。如果你要修改这些属性的值，应该将 needsUpdate 属性设为 true。要了解运行时哪些属性可以修改，哪些不可以修改，可以参考网页：https://github.com/mrdoob/three.js/wiki/Updates。
- 可以为一个几何体赋予多种材质。但是要记住，这么做会复制几何体，从而创建出多个网格。
- THREE.Line 几何体不可以用普通材质覆盖。因此只能使用 THREE.LineBasicMaterial 或 THREE.LineDashedMaterial。
- 如果想要一个光亮的物体，可以使用 MeshPhongMaterial；如果想要一个暗淡的物体，可以使用 MeshLambertMaterial。
- 使用 dat.GUI 来试验各种材质属性。在开发过程中猜测正确的材质属性值是非常困难的。

在本章和前一章，我们提到过一些几何体，并用在了一些示例里，而且还探究过其中几个。下一章将会介绍几何体的方方面面，以及如何使用它们。

第 5 章 Chapter 5

学习使用几何体

在前面几章我们学了 Three.js 库的很多用法。你知道如何创建基础场景、添加光照、以及为网格配置材质。在第 2 章我们接触了 Three.j 库提供的几何体，可以用这些几何体创建三维物体，但我们并没有真正深入这些几何体的细节。本章和下一章将展示 Three.js 提供的各种几何体。本章涉及的几何体包括：

- PlaneGeometry（平面）
- CircleGeometry（圆形）
- ShapeGeometry（塑形）
- CubeGeometry（立方体）
- SphereGeometry（球体）
- CylinderGeometry（圆柱）
- TorusGeometry（圆环）
- TorusKnotGeometry（环面纽结）
- PolyhedronGeometry（多面体）
- IcosahedronGeometry（二十面体）
- OctahedronGeometry（八面体）
- TetraHedronGeometry（四面体）

下一章将介绍下面几个复杂几何体：

- ConvexGeometry（凸面体）
- LatheGeometry（扫描面）
- ExtrudeGeometry（拉伸几何体）
- TubeGeometry（管状体）

- ParametricGeometry（参数几何体）
- TextGeometry（文本几何体）

让我们来看看 Three.js 提供的所有基础几何体。

5.1 Three.js 提供的基础几何体

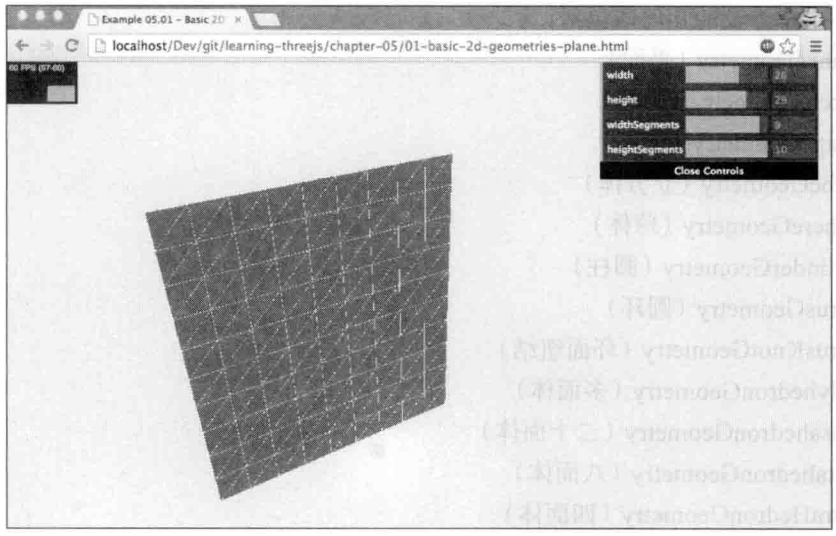
Three.js 中有几种几何体可以创建二维网格，但是大部分还是用来创建三维网格。本节先介绍一些二维几何体：CircleGeometry、PlaneGeometry 和 ShapeGeometry。然后再介绍所有可用的基础三维几何体。

5.1.1 二维几何体

二维几何体看上去是扁平的，顾名思义，它们只有两个维度。列表中的第一个二维几何体是 PlaneGeometry（平面）。

5.1.1.1 PlaneGeometry

PlaneGeometry 可以用来创建非常简单的二维矩形。通过本章的示例 01-basic-2d-geometries-plane.html，你可以找到这种几何体的例子。



创建这种几何体非常简单，如下所示：

```
new THREE.PlaneGeometry(width, height, widthSegments, heightSegments);
```

在这个 PlaneGeometry 例子中，可以修改它的属性，并立即查看修改的结果。关于这些属性的解释列在下表中：

属　性	必　须	描　述
width（宽度）	是	该属性指定矩形的宽度
height（高度）	是	该属性指定矩形的高度
widthSegments（宽度段数）	否	该属性指定矩形的宽度应该划分成几段
heightSegments（高度段数）	否	该属性指定矩形的高度应该划分成几段

正如你所看到的，这并不是一个复杂的几何体。你只要指定尺寸就可以了。如果你想创建更多的面（比如，一种拥有多个方格的样式），你可以使用 widthSegments 属性和 heightSegments 属性，将这个矩形分成多个小面。

在我们讲述下一个材质之前，先对该示例中用到的材质（也是我们在本章其他示例中使用的材质）做一个简短的说明。我们使用如下的方法在这些几何体上创建网格：

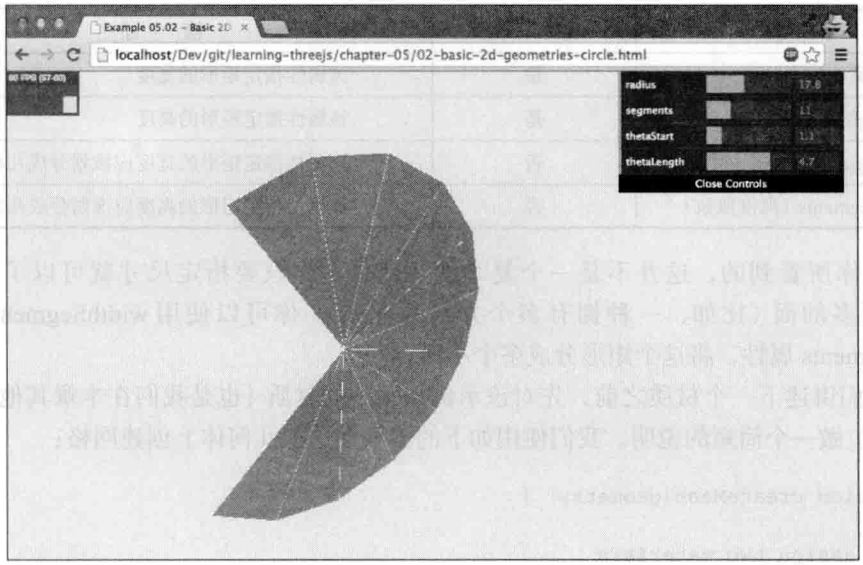
```
function createMesh(geometry) {

  // assign two materials
  var meshMaterial = new THREE.MeshNormalMaterial();
  meshMaterial.side = THREE.DoubleSide;
  var wireFrameMaterial = new THREE.MeshBasicMaterial();
  wireFrameMaterial.wireframe = true;

  // create a multimaterial
  var mesh = THREE.SceneUtils.createMultiMaterialObject(
    geometry, [meshMaterial,wireFrameMaterial]);
  return mesh;
}
```

我们在这个函数里基于传入的几何体创建了一个拥有多个材质的网格。第一个材质是 MeshNormalMaterial。正如你在前一章所看到的，这种材质基于物体表面的法向向量（方向）来计算它们的颜色。我们还把这种材质设置成双面的（THREE.DoubleSide）。如果我们不这么做，那么当物体的背面朝向相机的时候，就会看不到它。我们还添加了一个 MeshBasicMatreial 材质，通过它来打开线框效果。这样我们就可以看到该物体在三维空间中的形状，以及该几何体创建的面。

5.1.1.2　CircleGeometry

你可能已经猜到 CircleGeometry 能创建出什么。通过这个几何体你可以创建出简单的二维圆（或部分圆）。我们先看看这个例子：02-basic-2d-geometries-circle.html。在下面的截图中，我们创建了一个简单的 CircleGeometry 几何体，其 thetaLength[⊖]（弧度）属性小于 2*Pi。

⊖　从字面上来看该参数像是表示弧长，但实际含义是弧度。——译者注



在这个示例里你可以观察、控制用 CircleGeometry 创建的网格。当你创建 CircleGeometry 对象时，你可以指定几个属性，从而决定这个圆的样子。

属　性	是否必须	描　述
radius（半径）	是	该属性定义圆的半径，从而决定圆的大小。所谓半径指的是从圆心到圆弧的距离
segments（分段）	否	该属性定义创建圆所用面的数量。最少 3 个，如果没有指定则默认 8 个。值越大，创建出的圆越光滑
thetaStart（起始角）	否	该属性定义从哪儿开始画圆。取值范围是 0 到 2*Pi
thetaLength（角度）	否	该属性定义圆要画多大。如果没有指定，则默认为 2*Pi（整圆）。如果你指定的是 0.5*Pi，那么得到的将是一个四分之一圆。结合 thetaStart 属性和该属性即可定义圆的形状

从代码角度来说，你可以用以下的代码片段创建出一个完整的圆：

```
new THREE.CircleGeometry(3, 12);
```

如果你想创建一个半圆，那么你可以使用如下的代码：

```
new THREE.CircleGeometry(3, 12, 0, Math.PI);
```

在进入下一个几何体之前，我们先对 Three.js 创建二维图形（PlaneGeometry、CircleGeometry 和 ShapeGeometry）时所用的方向做一个简要说明。Three.js 创建这些对象时只使用 x 轴和 y 轴，所以它们都是“直立”的。这很符合逻辑，因为它们是二维对象。但是一般来讲，特别是用 PlaneGeometry 时，你会希望这些几何体躺下来，

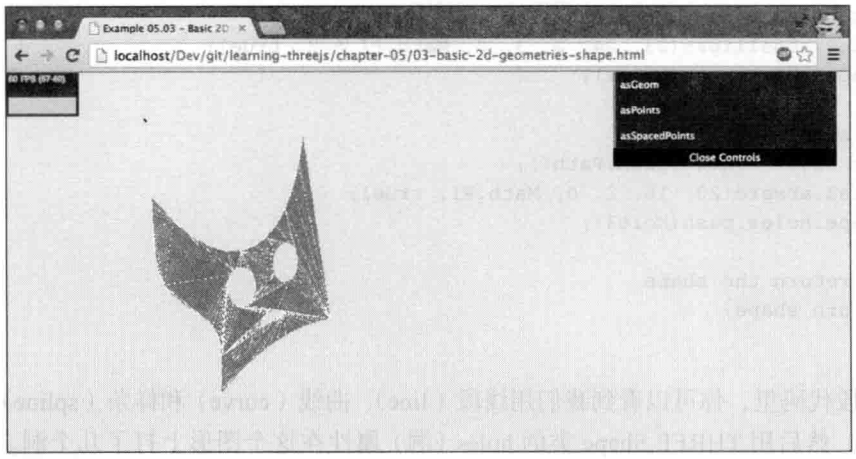
以便构建一种地面，好把其他对象放在上面。创建一个水平放置而不是竖直的二维对象最简单的方法就是：将这个网格沿 x 轴向后旋转四分之一圈（-pi/2）。

```
mesh.rotation.x = -Math.PI/2;
```

这就是 CircleGeometry 的全部内容，接下来是最后一个二维图形：ShapeGeometry。

5.1.1.3　ShapeGeometry

PlaneGeometry 和 CircleGeometry 只有有限的方法来定制它们的外观。如果你想创建一个自定义的二维图形，你可以使用 ShapeGeometry。通过 ShapeGeometry 你可以调用几个函数来创建你自己的图形。你可以将该功能与 HTML 画布和 SVG 中的 path 功能相比较。我们先从一个示例开始，然后我们会向你展示一下如何用这些函数来绘制你自己的图形。这个示例（03-basic-2d-geometries-shape.html）可以在本章的源代码中找到。



在这个例子里你可以看到一个定制的二维图形。先不必描述该几何体的属性，让我们来看看用来创建该图形的代码：

```
function drawShape() {

  // create a basic shape
  var shape = new THREE.Shape();

  // startpoint
  shape.moveTo(10, 10);

  // straight line upwards
  shape.lineTo(10, 40);

  // the top of the figure, curve to the right
  shape.bezierCurveTo(15, 25, 25, 25, 30, 40);
```

```
    // spline back down
    shape.splineThru(
      [new THREE.Vector2(32, 30),
      new THREE.Vector2(28, 20),
      new THREE.Vector2(30, 10),
    ])

    // curve at the bottom
    shape.quadraticCurveTo(20, 15, 10, 10);

    // add 'eye' hole one
    var hole1 = new THREE.Path();
    hole1.absellipse(16, 24, 2, 3, 0, Math.PI * 2, true);
    shape.holes.push(hole1);

    // add 'eye hole 2'
    var hole2 = new THREE.Path();
    hole2.absellipse(23, 24, 2, 3, 0, Math.PI * 2, true);
    shape.holes.push(hole2);

    // add 'mouth'
    var hole3 = new THREE.Path();
    hole3.absarc(20, 16, 2, 0, Math.PI, true);
    shape.holes.push(hole3);

    // return the shape
    return shape;
}
```

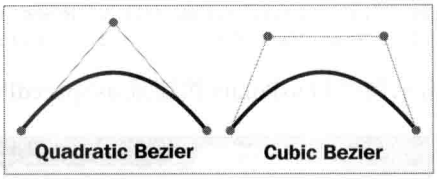
在这段代码里，你可以看到我们用线段（line）、曲线（curve）和样条（spline）创建出图形的轮廓。然后用 THREE.Shape 类的 holes（洞）属性在这个图形上打了几个洞。但是本节讨论的是 THREE.ShapeGeometry 对象，而不是 THREE.Shape 对象。要从 Shape 创建出一个几何体，我们需要编写如下代码：

```
new THREE.ShapeGeometry(drawShape());
```

这个函数的调用结果是一个用来创建网格的几何体。ShapeGeometry 类没有别的选项可以用来定制图形了。所以还是让我们来看看用来创建图形（Shape）的绘图函数：

名　称	描　述
moveTo(x, y)	该函数将绘图点移动到指定的 x、y 坐标处
lineTo(x, y)	该函数从当前位置（例如由 moveTo 设定的位置）画一条线到指定的 x、y 坐标处
quadricCurveTo(aCPx, aCPy, x, y)（二次曲线）	你可以用两种方法来定义曲线。你可以用 quadraticCurveTo 函数，或者用 bezierCurveTo 函数（下一项）。这两个函数的不同在于它们指定曲线曲率的方法。下图是这两种方法的示意图。

（续）

名　称	描　述
quadricCurveTo(aCPx, aCPy, x, y)（二次曲线）	 对于二次曲线，我们要额外指定一个点（使用 aCPx 和 aCPy 参数），当然还要指定端点（x 和 y 参数）。对于三次曲线（由 bezierCurveTo 函数绘制），你需要多指定两个点才能定义曲线。起始点是路径的当前点
bezierCurveTo(aCPx1, aCPy1, aCPx2 和 aCPy2, x, y)	根据提供的参数画一条曲线。相关说明可以参考前一行的内容。该曲线的绘制是基于两个定义曲线的坐标（aCPx1、aCPy1、aCPx2 和 aCPy2）以及终点坐标（x 和 y）。起始点是路径的当前位置
splineThru(pts)	该函数沿着提供的坐标（点）集合绘制一条光滑曲线。这个参数应该是一个 THREE.Vector2 对象数组。起始点是路径的当前位置
arc(ax, ay, aRadius, aStartAngle, aEndAngle, aClockwise)	画圆（或者一段圆弧）。圆弧起始于路径的当前位置。ax 和 ay 用来指定圆心与当前位置之间的偏移量。aRadius 设置圆的大小，而 aStartAngle 和 aEndAngle 则用来定义圆弧要画多长。布尔属性 aClockwise 决定这段圆弧是顺时针画还是逆时针画
absArc(aX, aY,aRadius, aStart-Angle,aEndAngle, AClockwise)	参考 arc 的描述。其位置是绝对位置，而不是相对当前位置
ellipse(aX, aY,xRadius, yRadius, aS-tartAngle,aEndAngle, aClockwise)	参考 arc 的描述。作为补充，通过 ellipse 函数我们可以分别指定 x 轴半径和 y 轴半径
absEllipse(aX, aY,xRadius, yRa-dius,aStartAngle,aEndAngle, aClockwise)	参考 ellipse 的描述。其位置是绝对位置，而不是与当前位置的相对值

最后一个我们需要说明的 Shape 对象属性是 holes（孔洞）属性。通过往这个属性中添加 THREE.Shape 对象（参考本节开始的代码示例），你可以在图形中打几个洞（例如本例中的 eye 对象）。

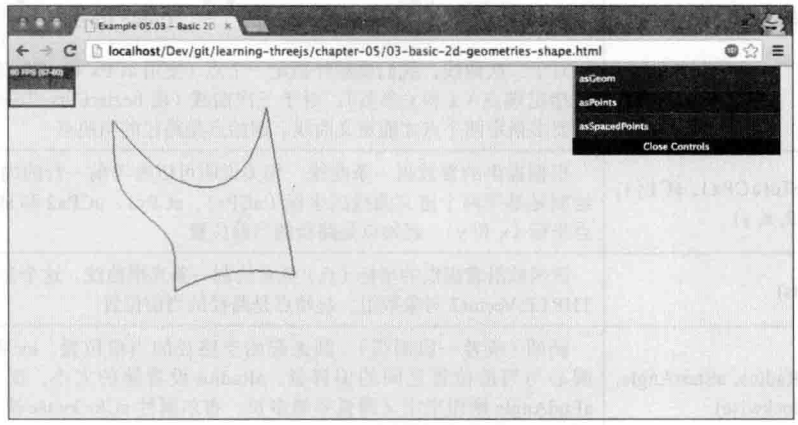
在这个例子里，我们已经讨论过使用 THREE.ShapeGeometry（drawShape()）函数，从 Shape 对象创建一个 ShapeGeometry。Shape 对象本身也有几个辅助函数，你可以用来创建几何体。

名　称	描　述
makeGeometry	该函数从 Shape 对象返回一个 ShapeGeometry 对象
createPointsGeometry (divisions)	该函数将图形转换成一个点集。属性 divisions（分段数）定义返回点的数目。这个值越高，返回的点越多，最终的曲线也就越平滑。这个 divisions 会分别应用到路径中的每一部分
createSpacedPointsGeometry (divisions)	该函数也是将图形转换成一个点集，但是这一次，分段数是一次性地应用到整个路径上

如果你用 createPointsGeometry 函数或 createSpacedPointsGeometry 函数创建一个点集，你可以用这些点来创建线段：

```
new THREE.Line( shape.createPointsGeometry(10), new
  THREE.LineBasicMaterial( { color: 0xff3333, linewidth: 2 } ) );
```

当你点击示例中的 asPoints 按钮或 asSpacedPoints 按钮时，你会看到如下图所示的结果：



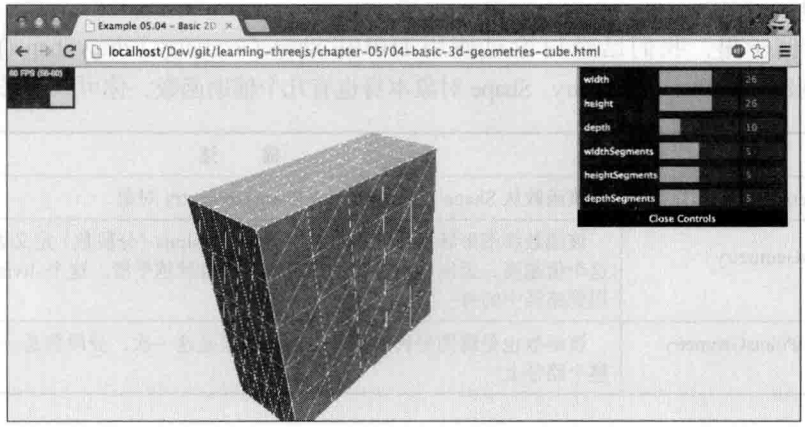
这就是所有的二维图形了。接下来将要展示和说明的是基础三维图形。

5.1.2 三维几何体

本节将讨论三维几何体，我们将从一个已经看到过多次的几何体（CubeGeometry）开始。

5.1.2.1 CubeGeometry

CubeGeometry 是一种非常简单的三维几何体，你只要指定宽度、高度和深度即可创建出一个方块。我们已经添加了一个示例（04-basic-3d-geometries-cube.html），你可以在上面试验这些属性。参考下图：



正如你在图中看到的，通过改变 CubeGeometry 对象的 width（宽度）、height（高度）和 depth（深度），你就可以控制网格的尺寸。这三个属性也是在创建一个方块时必须要提供的，如下所示：

```
new THREE.CubeGeometry(10,10,10);
```

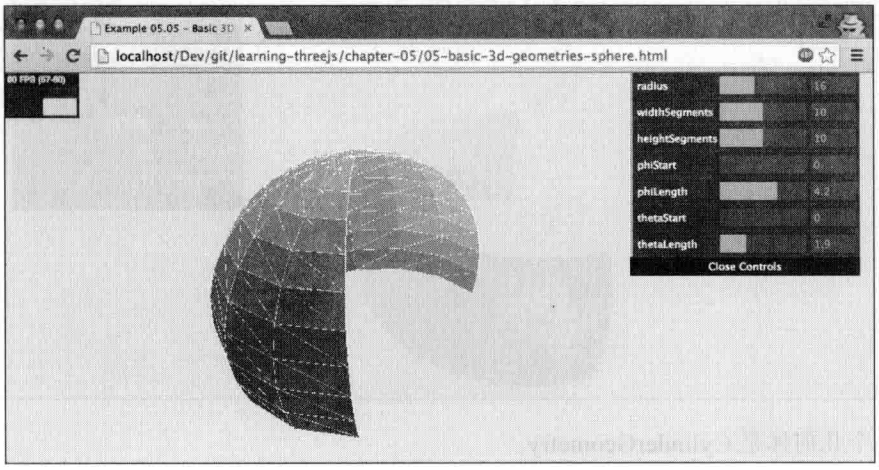
在这个例子里可以看到，你还可以定义方块的其他属性。下表是所有属性的说明：

属　性	是否必须	描　述
width	是	该属性定义方块的宽度。所谓宽度是方块沿 x 轴方向的长度
height	是	该属性定义方块的高度。所谓宽度是方块沿 y 轴方向的长度
depth	是	该属性定义方块的深度。所谓宽度是方块沿 z 轴方向的长度
widthSegments	否	该属性定义的是沿方块的 x 轴方向，将面分成多少份。默认值是 1
heightSegments	否	该属性定义的是沿方块的 y 轴方向，将面分成多少份。默认值是 1
depthSegments	否	该属性定义的是沿方块的 z 轴方向，将面分成多少份。默认值是 1

通过增加各个分段（segment）属性，你可以将方块的 6 个大面分成很多小面。这在你用 MeshFaceMaterial 为方块的不同部分设置特定材质属性时比较有用。CubeGeometry 是一个非常简单的几何体。另一个简单几何体是 SphereGeometry。

5.1.2.2　SphereGeometry

通过 SphereGeometry 你可以创建一个三维球体。我们直接看例子：05-basic-3d-geometries-sphere.html。参考下图：

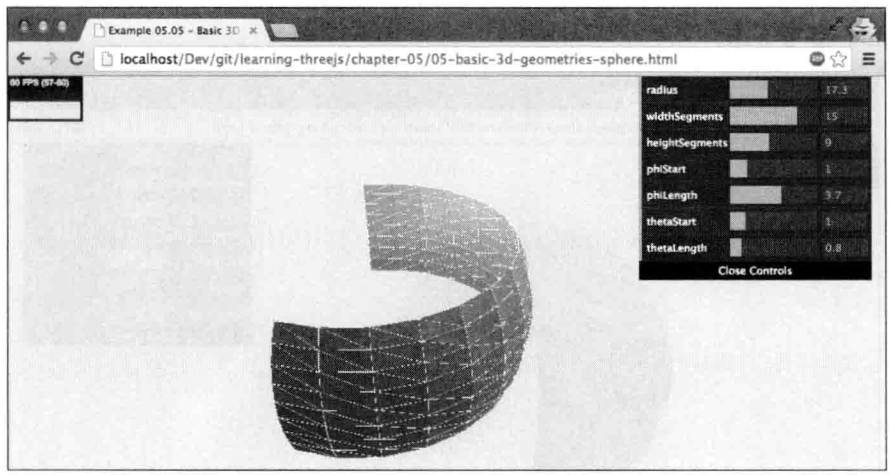


在前面这个截图里，我们展示的是半个打开的球，是基于 SphereGeometry 创建的。这个几何体非常灵活，可以用来创建所有跟球体相关的几何体。一个基础的 SphereGeometry 可以简单地通过 new THREE.SphereGeometry 来创建。下表中的属性可以用来调节结果网格的

外观：

属　性	是否必须	描　述
radius	否	该属性设置球体的半径。决定最终网格有多大。默认是 50
widthSegments	否	该属性指定竖直方向上的分段数。段数越多，球体的表面越光滑。默认值是 8，最小值是 3
heightSegments	否	该属性指定水平方向上的分段数。段数越多，球体的表面越光滑。默认值是 6，最小值是 2
phiStart	否	该属性用来指定从 x 轴的什么地方开始绘制。取值范围是 0 到 2*Pi。默认是 0
phiLength	否	该属性用来指定从 phiStart 开始画多少。2*Pi 是整球，0.5*Pi 画的是一个打开的四分之一球
thetaStart	否	该属性用来指定从 y 轴的什么地方开始绘制。取值范围是 0 到 Pi。默认是 0
thetaLength	否	该属性用来指定从 thetaStart 开始画多少。Pi 是整球，0.5*Pi 只会绘制上半球

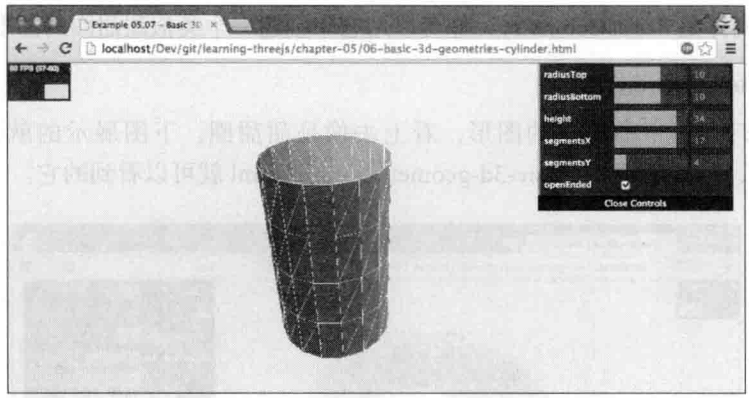
属性 radius、widthSegments 和 heightSegments 应该很清楚了。我们已经在别的例子里看过这些属性了。如果没有例子，属性 phiStart、phiLength、thetaStart 和 thetaLength 有点儿难懂。幸运的是，你可以通过示例 05-basic-3d-geometries-sphere.html 的菜单试验这些属性，创建一些比较有趣的图形，如下所示：



下一个几何体是 CylinderGeometry。

5.1.2.3 CylinderGeometry

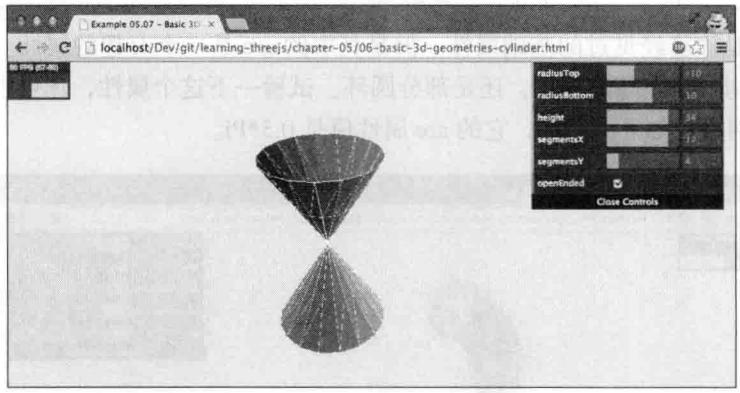
通过这个几何体我们可以创建圆柱和类似圆柱的物体。跟其他几何体一样，我们也有一个例子（06-basic-3d-geometries-cylinder.html），你可以在上面试验该几何体的属性。如下图所示：



创建 CylinderGeometry 时，没有必须要提供的参数。所以你只要调用 new THREE.CylinderGeometry() 即可创建出一个圆柱。正如你在示例中所看到的，你也可以提供几个参数，用来修改圆柱的外观。这些属性列在下表中：

属　性	是否必须	描　述
radiusTop	否	该属性设置圆柱顶部的尺寸，默认是 20
radiusBottom	否	该属性设置圆柱底部的尺寸，默认是 20
height	否	该属性设置圆柱的高度，默认是 100
segmentsX	否	该属性设置沿 x 轴分成多少段，默认是 8。这个数字越大，圆柱越光滑
segmentsY	否	该属性设置沿 y 轴分成多少段，默认是 1。分段越多，意味着面越多
openEnded	否	该属性指定网格的顶部和底部是否封闭。默认是 false

这些都是配置圆柱体的基础属性。但是有趣的是你可以在顶部（或底部）使用值为负数的半径。如果这么设置，你就可以用这个几何体创建一个类似沙漏的图形，如下图所示。需要注意的是：正如你从颜色中所看出的，圆柱的上半部分内外翻转了。如果你用的材质不是设置成 THREE.DoubleSide，你就看不到上半部分。

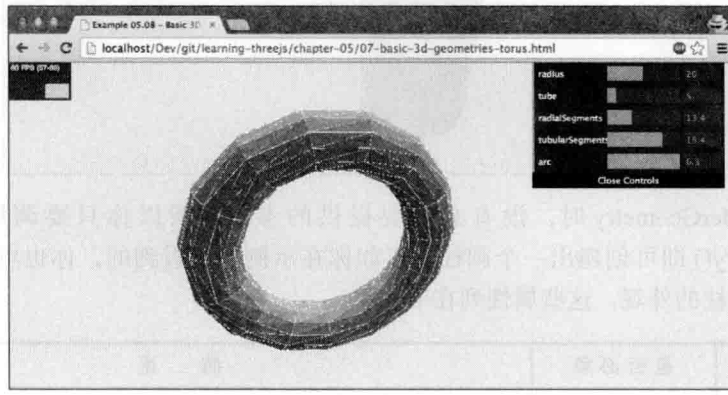


Create an hourglass shape by setting the top radius to a negative value

下一个几何体是 TorusGeometry，你可以用它来创建一个类似甜甜圈的图形。

5.1.2.4 TorusGeometry

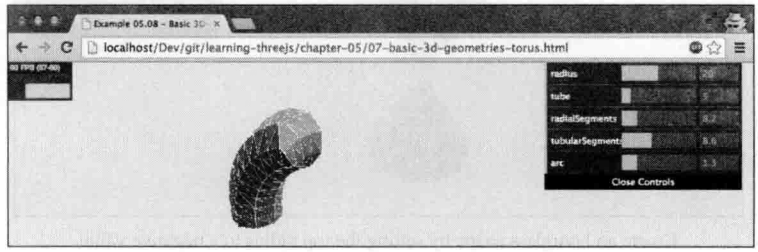
Torus（圆环）是一种简单的图形，看上去像是甜甜圈。下图展示的就是一个真实的 TorusGeometry，打开示例 07-basic-3d-geometries-torus.html 就可以看到的它。



跟大多数简单几何体一样，创建 TorusGeometry 时没有必须提供的参数。下表列出的是创建这个几何体时可以指定的参数：

属　性	是否必须	描　述
radius	否	这个参数设置的是完整圆环的尺寸，默认值是 100
tube	否	这个参数设置的是管子（真正的甜甜圈）的半径。该属性的默认值是 40
radialSegments	否	这个参数设置的是沿圆环长度方向分成的段数。默认是 8。可以在示例中看看修改该属性的结果
tubularSegments	否	这个参数设置的是沿圆环宽度方向分成的段数。默认是 6。可以在示例中看看修改该属性的结果
arc	否	通过该属性的值，你可以控制是否绘制一个完整的圆环。默认值是 2*Pi（完整圆环）

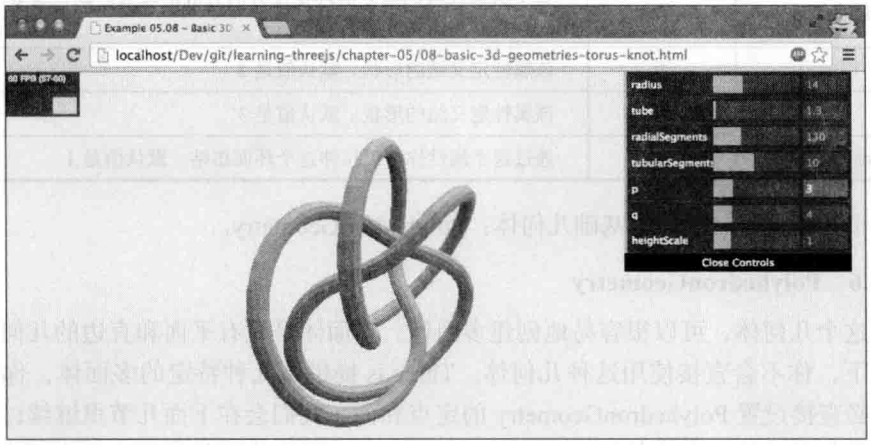
大部分都是你已经见过的基础属性。但是其中的 arc 是一个有趣的属性。通过这个属性你可以指定绘制一个完整的圆环，还是部分圆环。试验一下这个属性，你可以用它来创建出一些很有趣的网格，如下图所示，它的 arc 属性值是 0.5*Pi。



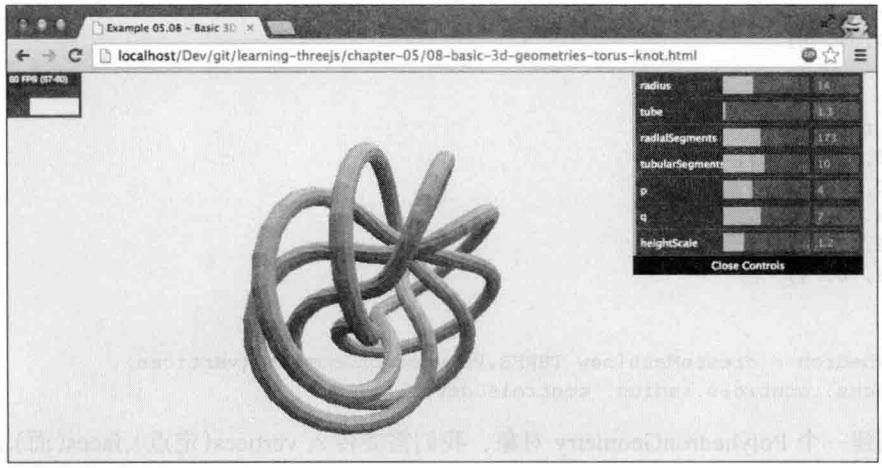
TorusGeometry 非常简单明了。下一节我们将要看的几何体是 TorusKnotGeometry，虽然名字跟 TorusGeometry 差不多，但是没有那么简洁。

5.1.2.5　TorusKnotGeometry

通过 TorusKnotGeometry 你可以创建一个环面纽结。环面纽结是一种比较特别的结，看上去就像是一根管子绕着它自己转了几圈。要想解释这一点，最好是看一个例子：08-basic-3d-geometries-torus-knot.html。



如果你打开这个例子，修改属性 p 和 q，你就可以创建出各种各样漂亮的几何体。p 属性定义该结绕其轴多久旋转一次。p 属性定义该结绕其内部旋转多少次。如果这听上去很空洞，你也不必担心。你不需要理解这些属性也能创建出漂亮的结（如果有人对细节感兴趣，Wikipedia 上关于这个主题有篇很不错的文章，网址是 http://en.wikipedia.org/wiki/Torus_knot）。



在这个几何体的例子里，你可以试验下面这些属性，看看不同的 p、q 组合对几何体有

什么样的影响，如上图所示。

属　性	是否必须	描　述
radius	否	这个参数设置的是完整圆环的尺寸，默认值是 100
tube	否	这个参数设置的是管子（真正的甜甜圈）的半径。该属性的默认值是 40
radialSegments	否	这个参数设置的是沿圆环长度方向分成的段数。默认是 8。可以在示例中看看修改该属性的结果
tubularSegments	否	这个参数设置的是沿圆环宽度方向分成的段数。默认是 6。可以在示例中看看修改该属性的结果
p	否	该属性定义结的形状。默认值是 2
q	否	该属性定义结的形状。默认值是 3
heightScale	否	通过这个属性你可以拉伸这个环面纽结。默认值是 1

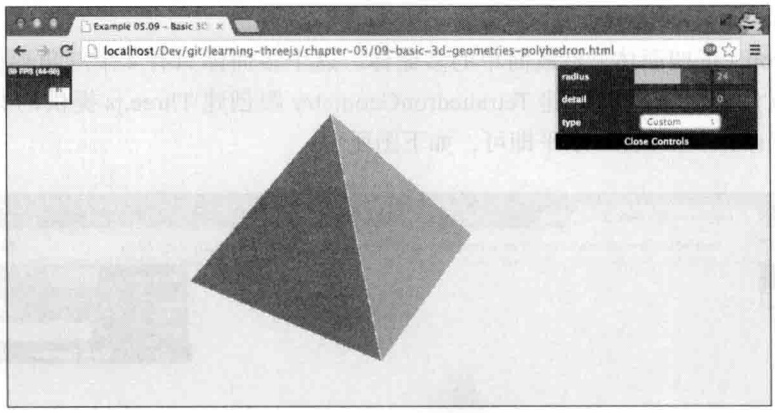
下一个几何体是最后一个基础几何体：PolyhedronGeometry。

5.1.2.6 PolyhedronGeometry

使用这个几何体，可以很容易地创建多面体。多面体是只有平面和直边的几何体。但是多数情况下，你不会直接使用这种几何体。Three.js 提供了几种特定的多面体，你可以直接使用，不必直接设置 PolyhedronGeometry 的定点和面。我们会在下面几节里继续讨论这些多面体。如果你的确想要直接使用 PolyhedronGeometry，那么你不得不指定各个顶点和面（就像我们在第 3 章使用方块时那样）。例如，我们要创建一个类似金字塔的多面体：

```
var vertices = [
  [1, 0, 1],
  [1, 0, -1],
  [-1, 0, -1],
  [-1, 0, 1],
  [0, 1, 0]
];

var faces = [
  [0, 1, 2, 3],
  [0, 1, 4],
  [1, 2, 4],
  [2, 3, 4],
  [3, 0, 4]
];

polyhedron = createMesh(new THREE.PolyhedronGeometry(vertices,
  faces, controls.radius, controls.detail));
```

要创建一个 PolyhedronGeometry 对象，我们需要传入 vertices（定点）、faces（面）、radius（半径）和 detail 参数（稍后进一步讨论）。这个 PolyhedronGeometry 的结果可以在示例 09-basic-3d-geometries-polyhedron.html 里看到（type 选择 Custom）。



当你创建多面体时，可以传入下面的四个属性：

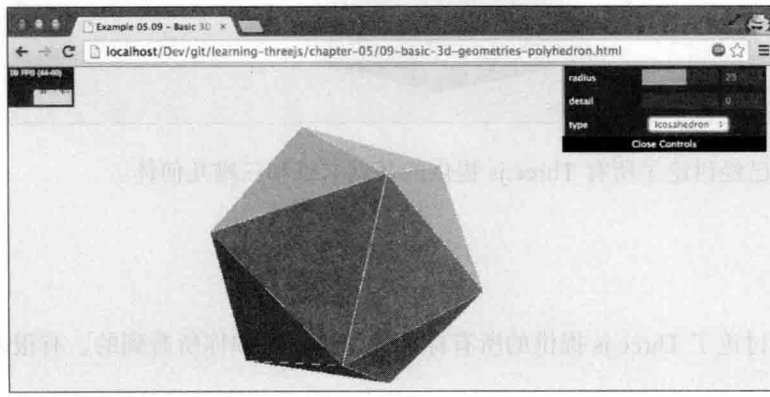
属　性	是否必须	描　述
vertices	是	该属性设置构成多面体的顶点
faces	是	该属性指定由 vertices 创建出的面
radius	否	该属性指定多面的大小。默认值是 1
detail	否	通过这个属性你可以给这个多面体添加额外的细节。如果设为 1，这个多面体上的每个三角形都会分成 4 个小三角形。如果设为 2，那么那些 4 个小三角形中的每一个都会继续分成 4 个小三角形，依次类推

在本节的开始，我们提到过 Three.js 提供了几个开箱即用的多面体。在下面的几个小节里，我们将会快速浏览一下这些多面体。

所有这些多面体都可以在示例 09-basic-3d-geometries-polyhedron.html 里找到。

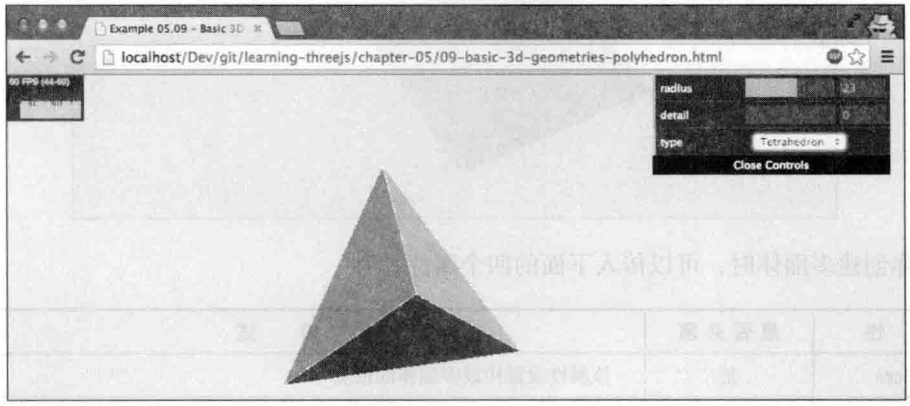
1. IcosahedronGeometry

通过 IcosahedronGeometry（正 20 面体）可以创建出一个有 20 个相同三角形面的多面体，这些三角形面是从 12 个顶点创建出来的。创建这个多面体时，你要做的只是指定 radius 和 detail 水平，如下图所示：



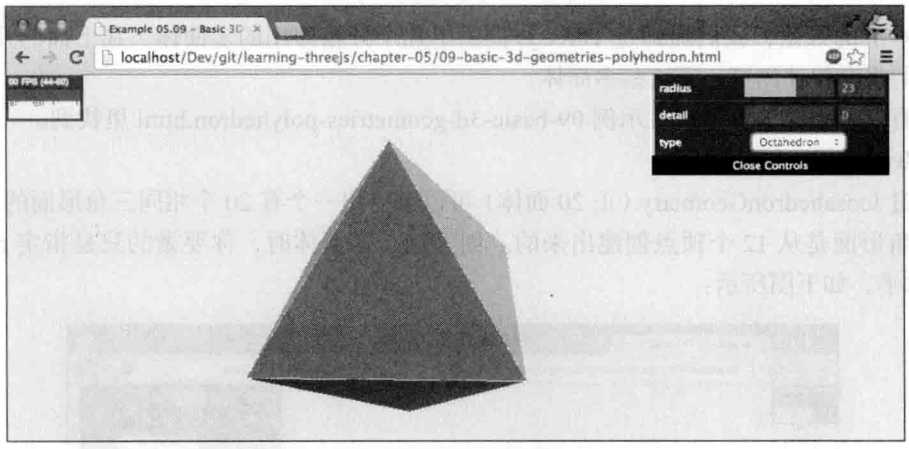
2. TetrahedronGeometry

Tetrahedron（正四面体）是最简单的多面体。这个多面体只有 4 个三角形面，而这些面是从 4 个顶点创建出来的。创建 TetrahedronGeometry 跟创建 Three.js 提供的其他多面体一样，只要指定 radius 和 detail 水平即可，如下图所示：



3. Octahedron

Three.js 提供的最后一个多面体是 Octahedron 正八面体。顾名思义，这个多面体有 8 个面。这些面是从 6 个顶点中创建出来的。下图展示的就是这样一个多面体。



至此我们已经讨论了所有 Three.js 提供的基础二维和三维几何体。

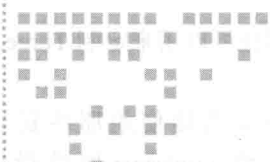
5.2 总结

本章我们讨论了 Three.js 提供的所有标准几何体。正如你所看到的，有很多几何体可以

开箱即用。本章需要记住的最重要的主题如下所示：

- 试验几何体是可能的。使用本章的示例来了解那些可以用来定制 Three.js 提供的标准几何体的属性。
- 开始创建几何体时选择一种合适的材质。不要直接使用那些复杂的材质，可以从简单的 MeshBasicMaterial 材质开始，并将 wireframe 属性设为 true，或者 MeshNormalMaterial 材质也可以。那样你就可以对几何体的真实形状有一个更好的了解。
- 记住，创建二维几何体时，z 轴没有考虑。如果你想拥有一个水平的二维图形，那么你必须将这个网格绕 x 轴旋转 –0.5*Pi。
- 如果你要旋转一个二维图形，或者一个开放的三维图形（例如圆柱或者管子），记住要将材质设置成 THREE.DoubleSide。如果你不这么做，那么该几何体的内侧或背面将会不可见。

本章我们聚焦在简单、易懂的网格。Three.js 也提供了一些方法可以用来创建复杂几何体。下一章你将会学习如何创建它们。



Chapter 6 第 6 章

使用高级几何体和二元操作

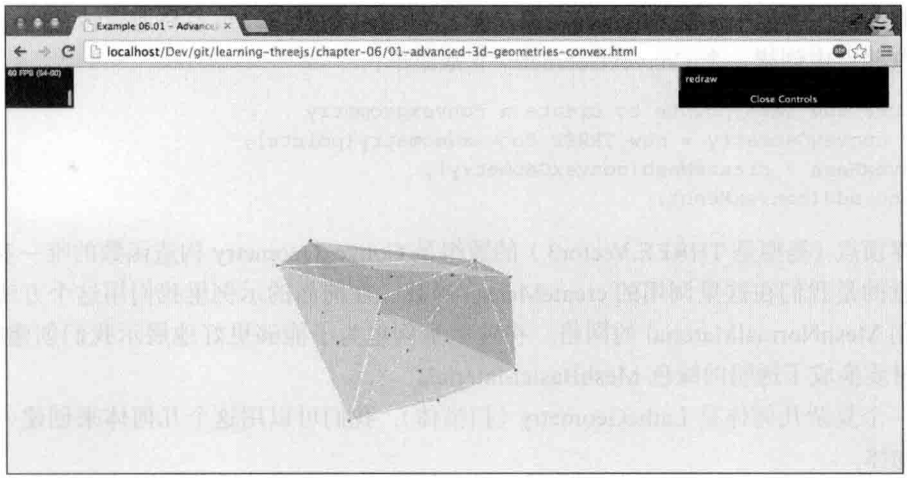
在前一章我们向你展示了所有 Three.js 库提供的基础几何体。除了这些基础几何体，Three.js 还提供了一些比较高级而且特别的对象。本章我们将向你展示这些高级几何体，并涵盖下面这些内容：

- 你将会学习如何使用高级几何体，例如 ConvexGeometry（凸面体）、LatheGeometry（扫描体）和 TubeGeometry（管状几何体）。
- 我们会展示如何使用 ExtrudeGeometry（拉伸几何体）从二维图形生成三维图形。画这个二维图形我们用的是 Three.js 提供的功能。我们还会基于一个从外部加载的 SVG 图片创建出一个三维图形。
- 如果你想自己创建一些可以定制的图形，可以使用本章或前一章讨论的方法。不过 Three.js 库也提供了一个 ParamtericGeometry 对象，通过这个对象你可以基于一组公式来创建几何体。
- 最后，我们来看看如何使用 TextGeometry 来创建三维文字。
- 另外我们还会展示一下如何使用二元操作从已有的几何体中创建出新几何体。二元操作是由 Three.js 的扩展 THREEBSP 提供的功能。

我们从上述列表中的第一项 ConvexGeometry（凸面体）开始。

6.1 ConvexGeometry

通过 ConvexGeometry 我们可以在一组点的外面建立一个凸包。所谓凸包就是包围这组点的最小图形。理解这个概念的最好方法就是看一个例子。如果你打开示例 01-advanced-3d-geometries-convex.html，你就会看到一个包围一组随机点的凸包。



我们在这个例子里随机生成了一组点，然后在这组点的基础上创建了一个 ConvexGeometry。你可以在这个示例里点击 redraw（重新绘制）按钮，这可以生成 20 个新点，并绘制凸包。我们还为每个点添加了一个小的 SphereGeometry（球体），这样可以更好地观察凸包是如何工作的。下面这段代码展示的是这些点是如何创建并添加到场景中的：

```
function generatePoints() {
  // add 10 random spheres
  var points = [];
  for (var i = 0; i < 20; i++) {
    var randomX = -15 + Math.round(Math.random() * 30);
    var randomY = -15 + Math.round(Math.random() * 30);
    var randomZ = -15 + Math.round(Math.random() * 30);
    points.push(new THREE.Vector3(randomX, randomY, randomZ));
  }

  spGroup = new THREE.Object3D();
  var material = new THREE.MeshBasicMaterial(
    {color: 0xff0000, transparent: false});
  points.forEach(function (point) {
    var spGeom = new THREE.SphereGeometry(0.2);
    var spMesh = new THREE.Mesh(spGeom, material);
    spMesh.position = point;
    spGroup.add(spMesh);
  });

  // add the points as a group to the scene
  scene.add(spGroup);
}
```

正如你在代码中所看到的，我们创建了 20 个随机点（THREE.Vector3），并保存在一个数组中。接着我们遍历这个数组并创建 SphereGeometry，并把它们定位到这些点上。所有这

些点都添加到一个组中（更多信息参考第 7 章），这样我们就可以很容易地旋转它们。

从这些点中创建一个 ConvexGeometry 非常简单：

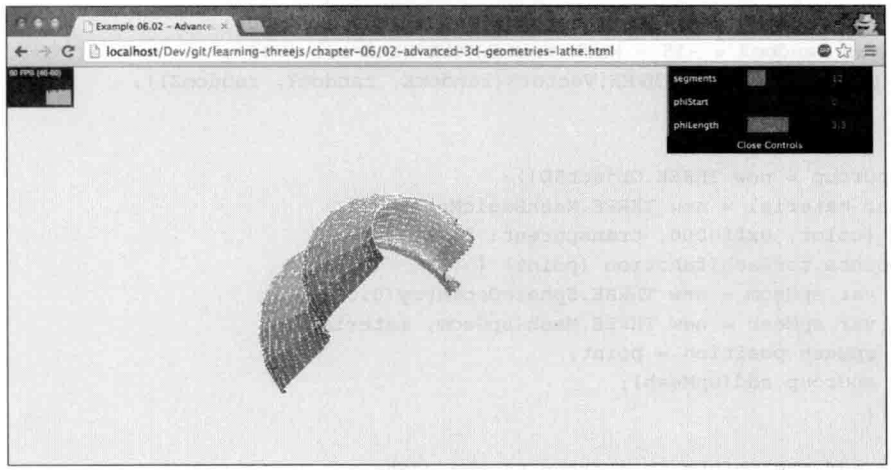
```
// use the same points to create a convexgeometry
var convexGeometry = new THREE.ConvexGeometry(points);
convexMesh = createMesh(convexGeometry);
scene.add(convexMesh);
```

保存顶点（类型是 THREE.Vector3 ）的数组是 ConvexGeometry 构造函数的唯一参数。最后要注意的是我们在这里调用的 createMesh() 函数。在前面的示例里我们用这个方法创建了一个使用 MeshNormalMaterial 的网格。在这个示例里为了能够更好地展示我们创建的凸包，我们把材质换成了透明的绿色 MeshBasicMaterial。

下一个复杂几何体是 LatheGeometry（扫描体），我们可以用这个几何体来创建一个类似花瓶的物体。

6.2 LatheGeometry

通过 LatheGeometry 你可以从一条光滑曲线开始创建图形。这条曲线是通过指定一些点（也叫节点）来定义的，而这条曲线通常也被称作样条曲线。当这条样条曲线绕一个固定点旋转时就构成了一个类似花瓶或铃铛的图形。同样，要理解 LatheGeometry 的功能最好的方法就是来看一个例子。示例 02-advanced-3d-geometries-lathe.html 展示就是这样一个几何体：



在这个屏幕截图里你可以看到这条样条曲线是由一组红色小球来表示的。这些小球的位置连同其他一些参数一同传递给 LatheGeometry 的构造函数。在这个例子里我们将这条样条曲线转了半圈，如此就构成了你所看到的图形。在看那些参数之前，我们先来看看创建样条曲线的代码，以及 LatheGeometry 如何使用这条样条曲线：

```
function generatePoints(segments, phiStart, phiLength) {
  // add 10 random spheres
  var points = [];
  var height = 5;
  var count = 30;
  for (var i = 0; i < count; i++) {
    points.push(new THREE.Vector3((Math.sin(i * 0.2)
      + Math.cos(i * 0.3)) * height + 12,
      0, ( i - count ) + count / 2));
  }

  ...

  // use the same points to create a convexgeometry
  var latheGeometry = new THREE.LatheGeometry
    (points, segments, phiStart, phiLength);
  latheMesh = createMesh(latheGeometry);
  scene.add(latheMesh);
}
```

在这段 JavaScript 代码里你可以看到我们生成了 30 个点，这些点的 x 坐标是正弦函数和余弦函数结果的组合，z 坐标则是基于变量 i 和 count。通过这些点即可创建出样条曲线，而这些点则用一些红点显示在我们之前看过的那个屏幕截图上。

基于这些点我们就可以创建 LatheGeometry。除了这个顶点数组，LatheGeometry 还接受其他几个参数。下表所列的就是所有这些参数：

属　性	是否必须	描　述
points	是	该属性指定构成样条曲线的点，然后基于这条样条曲线生成类似铃铛或花瓶的图形
segments	否	该属性指定创建图形时所用的分段数目。这个数字越高，最终的图形越光滑。默认值是 12
phiStart	否	该属性指定创建图形时从圆的何处开始。取值范围是 0 到 2*Pi。默认值是 0
phiLength	否	该属性指定创建出的图形有多完整。例如四分之一图形就是 0.5*Pi。默认值是完整的 360 度或 2*Pi

在本章开始的时候我们展示了几个二维图形。在下一节里我们将学习如何从这些二维图形中创建出三维图形，这就是所谓的拉伸。

6.3　通过拉伸创建几何体

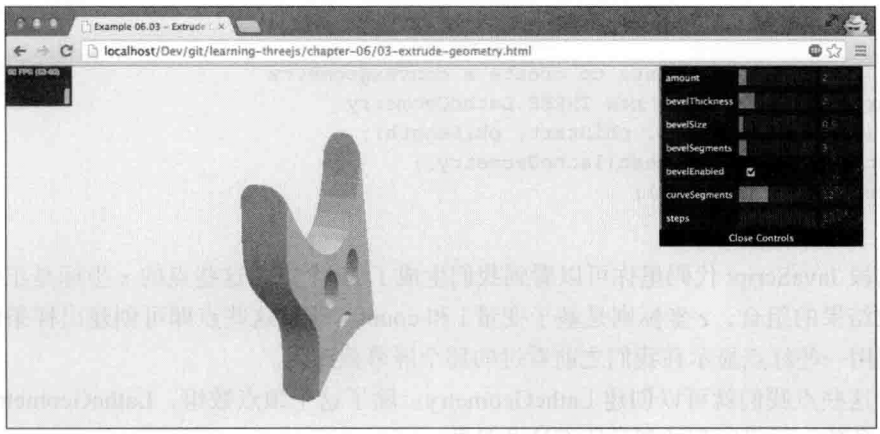
Three.js 提供了几个方法让我们可以把一个二维图形拉伸成三维图形。所谓拉伸指的是先画一个图形的二维轮廓，然后沿着 z 轴将它转换成三维图形。例如，如果我们拉伸一

个 THREE.CircleGeometry 对象，我们就会得到一个类似圆柱体的图形；如果我们拉伸一个 THREE.PlanceGeometry 对象，我们就会得到一个类似方块的图形。

最通用的拉伸图形的方法是使用 THREE.ExtrudeGeometry 对象。

6.3.1 ExtrudeGeometry

通过 ExtrudeGeometry 你可以从一个二维图形创建出一个三维图形。在我们深入到这个几何体的细节之前，我们先来看一个示例，03-extrude-geometry.html，如下图所示：



在这个示例里我们使用本章之前创建的二维图形，并使用 ExtrudeGeometry 将它转换成三维图形。正如你在这个截图中所看到的，这个图形沿着 z 轴拉伸，最终形成一个三维图形。创建 ExtrudeGeometry 的代码非常简单：

```
var options = {
  amount: 10,
  bevelThickness: 2,
  bevelSize: 1,
  bevelSegments: 3,
  bevelEnabled: true,
  curveSegments: 12,
  steps: 1
};

shape = createMesh(new THREE.ExtrudeGeometry(drawShape(),
  options));
```

在这段示例代码里我们跟前一章一样，使用 drawShape() 函数创建图形。然后将这个图形连同 options（选项）对象一起传递给 THREE.ExtrudeGeometry 的构造函数。通过 options 你可以明确定义图形应该怎样拉伸。下表是对可以传递给 THREE.ExtrudeGeometry 的各个选项的解释：

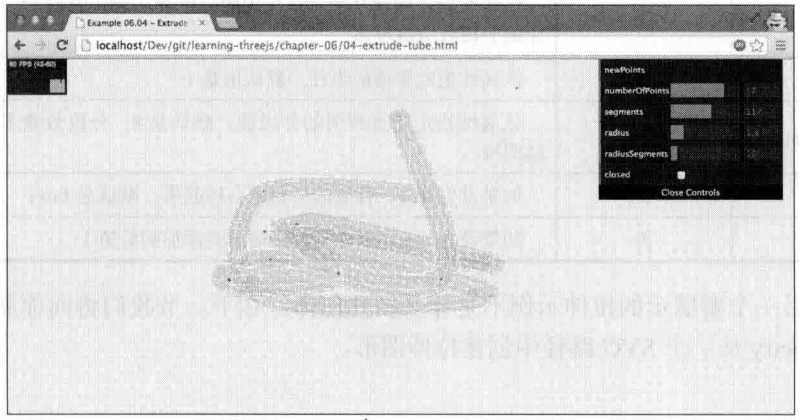
属　性	是否必须	描　述
amount（数量）	否	该属性指定图形可以拉多高。默认值是 100
bevelThickness（斜角厚度）	否	该属性指定斜角的深度。斜角是前后面和拉伸体之间的倒角。默认值是 6
bevelSize（斜角尺寸）	否	该属性指定斜角的高度。默认值是 bevelThickness-2
bevelSegments（斜角分段数）	否	该属性定义的是斜角的分段数。段数越多，斜角越光滑。默认值是 3
bevelEnabled（是否用斜角）	否	如果设为 true，就会有斜角。默认值是 true
curveSegments（曲线分段数）	否	该属性指定拉伸图形时曲线分成多少段。段数越多，曲线越光滑。默认值是 12
steps（拉伸体段数）	否	该属性定义拉伸体被分成多少段。默认值是 1
extrudePath（拉伸路径）	否	该属性指定图形沿着什么路径拉伸。如果没有指定，图形就会沿着 z 轴拉伸
material（材质）	否	该属性定义的是前后面所用材质的索引。用函数 THREE.SceneUtils.createMultiMaterialObject 创建网格
extrudeMaterial（拉伸材质）	否	该属性指定斜角和拉伸体所用材质的索引。用函数 THREE.SceneUtils.createMultiMaterialObject 创建网格

你可以使用示例 12-extrude-geometry.html 里的菜单来试验这些选项。

在这个例子中我们沿着 z 轴拉伸图形。正如你在选项列表里所看到的，你也可以沿着一条路径拉伸图形。在下一个几何体 TubeGeometry 里我们就会那么做。

6.3.2　TubeGeometry

TubeGeometry 沿着一条三维样条曲线拉伸出一根管子。你可以通过指定顶点来定义路径，然后 TubeGeometry 就可以创建这根管子。你可以在本章的源代码路径下找到一个可以试验的例子：13-extrude-tube.html。如下图所示：



在这个例子里你可以看到，我们随机生成一些点，然后用这些点来画管道。通过右上角的控件，我们可以定义管道的外观，或者点击 newPoints 按钮生成新管道。创建管道的代码很简单：

```
var points = [];
for (var i = 0 ; i < controls.numberOfPoints ; i++) {
  var randomX = -20 + Math.round(Math.random() * 50);
  var randomY = -15 + Math.round(Math.random() * 40);
  var randomZ = -20 + Math.round(Math.random() * 40);

  points.push(new THREE.Vector3(randomX, randomY, randomZ));
}

var tubeGeometry = new THREE.TubeGeometry(
  new THREE.SplineCurve3(points),
    segments, radius, radiusSegments, closed);

var tubeMesh = createMesh(tubeGeometry);
scene.add(tubeMesh);
```

我们首先要做的是获取一组顶点，类型是 THREE.Vector3。跟我们在 THREE.Convex-Geomtry 和 THREE.LatheGeometry 示例里做的一样。但是，在使用这些点创建管道之前，我们先要把这些点转换成 THREE.SplineCurve3 类。换言之，我们需要用这些点来定义一条光滑曲线。做法很简单，只要把顶点数组传递给 THREE.SplineCurve3 构造函数即可。有了样条曲线，以及其他一些参数（稍后解释），我们就可以创建管道，并添加到场景中。

除了 THREE.SplineCurve3 对象，TubeGeometry 构造函数还接受别的参数。下表列出的就是 TubeGeometry 的所有参数：

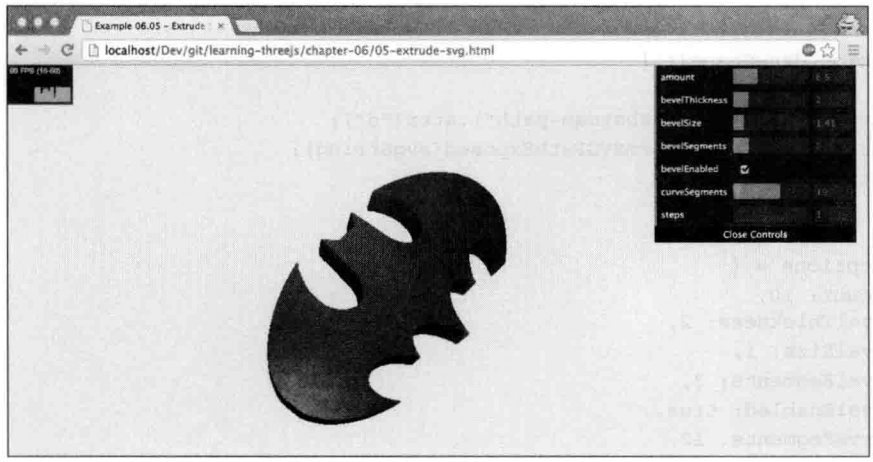
属　性	是否必须	描　述
path	是	该属性用一个 THREE.SplineCurve 对象来指定管道应当遵循的路径
segments	否	该属性指定构建这个管道所用的分段数。默认值是 64。路径越长，指定的分段数应该越多
radius	否	该属性指定管道的半径。默认值是 1
radiusSgements	否	该属性指定管道圆周的分段数。默认是 8。分段数越多，管道看上去越圆滑
closed	否	如果设为 true，管道的头和尾会连起来。默认是 false
debug	否	如果设为 true，额外的调试信息会添加到管道上

本章最后一个要展示的拉伸示例不是什么新几何体。在下一节我们将向你展示如何使用 ExtrudeGeometry 从一个 SVG 路径中创建拉伸图形。

6.3.3　从 SVG 拉伸

我们在讨论 ShapeGeometry 时曾经提到过 SVG 和曲线创建图形的方式基本相同。特别是 SVG 有一个跟 Three.js 处理图形相同的方式。本节我们将来看看如何使用来自 https://github.com/asutherland/d3-threeD 的小型库，将 SVG 路径转换成 Three.js 图形。

作为一个例子，我们使用 ExtrudeGeometry 将一个蝙蝠侠标识符的 SVG 图案转换成三维图形。你可以在本章的源码里找到一个示例（05-extrude-svg.html）来试验：



首先我们来看看原始 SVG 的样子（可以在本例的源代码里找到）：

```
<svg version="1.0" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
width="1152px" height="1152px" xml:space="preserve">
<g>
<path  id="batman-path" style="fill:rgb(0,0,0);" d="M 261.135
114.535 C 254.906 116.662 247.491 118.825 244.659 119.344 C
229.433 122.131 177.907 142.565 151.973 156.101 C 111.417 177.269
78.9808 203.399 49.2992 238.815 C 41.0479 248.66 26.5057 277.248
21.0148 294.418 C 14.873 313.624 15.3588 357.341 21.9304 376.806 C
29.244 398.469 39.6107 416.935 52.0865 430.524 C 58.2431 437.23
63.3085 443.321 63.3431 444.06 ... 261.135 114.535 "/>
</g>
</svg>
```

除非你是 SVG 专家，否则这对你来说就毫无意义。但是基本上你看到的是一组绘图指令。例如，C 277.987 119.348 279.673 116.786 279.673 115.867 是告诉浏览器画一条三次 Bezier 曲线，而 L 489.242 111.787 则是告诉我们应该在指定位置画一条线。幸运的是我们不必自己写代码来解析。使用 d3-threeD 库，可以自动转换这些指令。该库原是跟优秀的 D3.js 库一同开发、使用的，但是做了一点儿小小的调整，以便我们能够单独使用其中部分功能。

SVG 的含义是 Scalabel Vector Graphics（可缩放矢量图）。这是一个基于 XML 的标准，用来在网页上创建二维矢量图。该标准是一个开放的标准，大部分现代浏览器都支持。但是直接使用 SVG，通过 JavaScript 来操纵它并不直观。幸运的是有几个开源 JavaScript 库使得使用 SVG 时简单很多。D3.js 和 Raphael.js 就是其中最好的两个。

下面的代码展示的是如何加载你之前看到的 SVG，并转换成 ExtrudeGeometry，显示在屏幕上：

```
function drawShape() {

  var svgString = $("#batman-path").attr("d");
  var shape = transformSVGPathExposed(svgString);
  return shape;
}

var options = {
  amount: 10,
  bevelThickness: 2,
  bevelSize: 1,
  bevelSegments: 3,
  bevelEnabled: true,
  curveSegments: 12,
  steps: 1
};

shape = createMesh(new THREE.ExtrudeGeometry(drawShape(),
  options));
```

在这段代码里你会看到有一个 transformSVGPathExposed 函数的调用。该函数是 d3-ThreeD 库提供的，接受一个 SVG 字符串作为参数。我们使用表达式 $（"#batman-path"）.attr（"d"）直接从 SVG 元素中获取 SVG 字符串。SVG 元素中的属性 d 包含的就是用来绘制图形的路径表达式。添加上好看的、闪亮的材质，再加上一个聚光灯光源，你就可以重建该示例。

本节我们最后要讨论的几何体是 ParametricGeometry。通过该几何体，你可以指定几个函数，自动创建几何体。

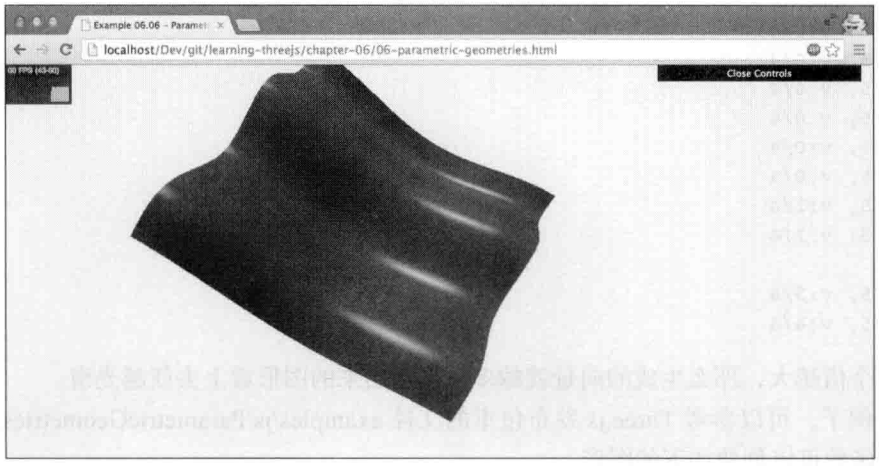
6.3.4 ParametricGeometry

通过 ParametricGeometry，你可以创建基于等式的几何体。在看我们的例子之前，最好先来看一下 Three.js 提供的例子。在你下载的 Three.js 发布包里，有这样的文件：examples/js/ParametricGeometries.js。你可以在这个文件里找到几个公式的例子，并在 ParametricGeometry 中使用它们。最基础的例子是一个创建平面的函数：

```
function plane(u, v) {
  var x = u * width;
  var y = 0;
  var z = v * depth;
  return new THREE.Vector3(x, y, z);
}
```

ParametricGeometry 会调用这个函数。u 和 v 的取值范围是 0 到 1，而且针对 0 到 1 之间的所有值该函数还会被调用很多次。在这个例子里，u 值用来确定向量的 x 坐标，v 值用来确定 z 坐标。当这个函数被调用的时候，你就会得到一个宽为 width，深为 depth 的基础平面。

我们的例子所做的事情跟这差不多。但创建的不是一个平面，而是一个类似波浪的东西，你可以在示例 06-parametric-geometries.html 看到如下的结果：



要创建这样的图形，我们要将如下的函数传递给 ParametricGeometry：

```
radialWave = function (u, v) {
    var r = 50;

  var x = Math.sin(u) * r;
  var z = Math.sin(v / 2) * 2 * r;
  var y = (Math.sin(u * 4 * Math.PI)
    + Math.cos(v * 2 * Math.PI)) * 2.8;

  return new THREE.Vector3(x, y, z);
}

var mesh = createMesh(new THREE.ParametricGeometry(radialWave,
  120, 120, false));
```

正如你在示例中所看到的，只要几行代码，我们就可以创建出有趣的几何体。你还可以在这个示例里看到传递给 ParametricGeometry 的参数。下表是对这些参数的解释：

属　性	是否必须	描　述
function	是	该参数是一个函数，以 u、v 值（0 到 1）作为参数，返回值是一个 Vector3 类型的对象，作为图形上点的坐标
slices	是	定义 u 值应该分成多少份
stacks	是	定义 v 值应该分成多少份
useTris	否	默认是 false。如果设为 true，那么该几何体创建时将会使用三角面片。如果设为 false，使用的将是四边形

在进入本章的下一节之前，我想最后对 slices 和 stacks 属性的使用做一下说明。我们曾经提到过 u、v 属性会传递给由 function 属性指定的函数，而且这两个属性的取值范围是 0 到 1。通过 slices 和 stacks 属性，我们可以指定 function 函数会被调用多少次。假如我们将 slices 设为 5，stacks 设为 4，那么在调用这个函数时将会使用如下参数：

```
u:0/5, v:0/4
u:1/5, v:0/4
u:2/5, v:0/4
u:3/5, v:0/4
u:4/5, v:0/4
u:5/5, v:0/4
u:0/5, v:1/4
u:1/5, v:1/4
...
u:5/5, v:3/4
u:5/5, v:4/4
```

这两个值越大，那么生成的向量就越多，创建出来的图形看上去就越光滑。

更多例子，可以参考 Three.js 发布包里的文件 examples/js/ParametricGeometries.js。该文件包含的函数可以创建如下的图形：

- 克莱因瓶
- 平面
- 二维莫比乌斯带
- 三维莫比乌斯带
- 管道
- 环面纽结
- 球体

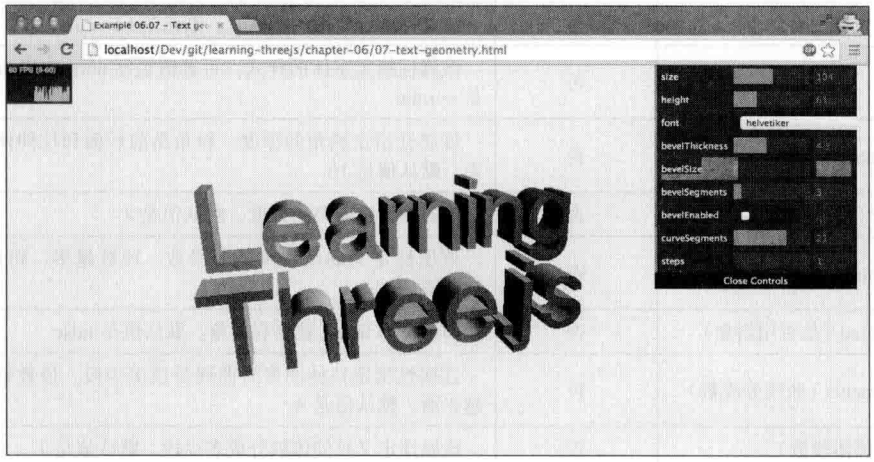
本章的下一部分讲述的是如何创建三维文本对象。

6.4 创建三维文本

本节我们来快速看一下如何创建出三维文本效果。首先，我们将会学习如何使用 Three.js 提供的字体渲染文本，然后学习如何使用你自己定义的字体。

6.4.1　渲染文本

在 Three.js 中渲染文本非常简单。你所要做的只是指定想用的字体，以及基本的拉伸属性（就是我们在讨论 ExtrudeGeometry 时见过的那些属性）。下面这个截图展示的就是一个在 Three.js 里如何渲染文本的例子：07-text-geometry.html：



显示这几行文本的代码如下所示：

```
var options = {
  size: 90,
  height: 90,
  weight: 'normal',
  font: 'helvetiker',
  style: 'normal',
  bevelThickness: 2,
  bevelSize: 4,
  bevelSegments: 3,
  bevelEnabled: true,
  curveSegments: 12,
  steps: 1
};


text1 = createMesh(new THREE.TextGeometry("Learning", options));
text1.position.z = -100;
text1.position.y = 100;
scene.add(text1);

text2 = createMesh(new THREE.TextGeometry("Three.js", options));
scene.add(text2);
};
```

让我们来看一看可以在 TextGeometry 里指定的属性：

属　性	是否必须	描　述
size	否	该属性指定文本的大小。默认是 100
height	否	该属性指定拉伸的长度。默认是 50
weight	否	该属性指定字体的权重。可选的值是 normal 和 bold。默认是 normal
font	否	该属性指定要用的字体名。默认是 helvetiker
style	否	该属性指定字体的样式。可选值包括 normal 和 italic。默认是 normal
bevelThickness（斜角厚度）	否	该属性指定斜角的深度。斜角是前后面和拉伸体之间的倒角。默认值是 10
bevelSize（斜角尺寸）	否	该属性指定斜角的高度。默认值是 8
bevelSegments（斜角分段数）	否	该属性定义的是斜角的分段数。段数越多，斜角越光滑。默认值是 3
bevelEnabled（是否用斜角）	否	如果设为 true，就会有斜角。默认值是 false
curveSegments（曲线分段数）	否	该属性指定拉伸图形时曲线分成多少段。段数越多，曲线越光滑。默认值是 4
steps（拉伸体段数）	否	该属性定义拉伸体被分成多少段。默认值是 1
extrudePath（拉伸路径）	否	该属性指定图形沿着什么路径拉伸。如果没有指定，图形就会沿着 z 轴拉伸
material（材质）	否	该属性定义的是前后面所用材质的索引。用函数 THREE.SceneUtils.createMultiMaterialObject 创建网格
extrudeMateria（拉伸材质）	否	该属性指定斜角和拉伸体所用材质的索引。用函数 THREE.SceneUtils.createMultiMaterialObject 创建网格

Three.js 所包含的字体也被添加到本书的源码中了。你可以在文件夹 the assets/fonts 中找到它们。

如果你想渲染一组二维文字，例如用作材质的纹理，那么你不应该使用 TextGeometry。TextGeometry 和 JavaScript 字体引入了很多操作。对于简单的二维字体渲染，最好使用 HTML5 画布。通过 context.font 属性，你可以设置要用的字体，通过 context.fillText 属性你可以将文本输出到画布上。

在这个几何体中也可以使用其他字体，不过你首先要把它们转换成 JavaScript。如何转换下节将会说明。

6.4.2 添加自定义字体

Three.js 提供了几种可以在场景中使用的字体。这些字体的基础是由 typeface.js（http://typeface.neocracy.org）提供的字体。typeface.js 是一个可以将 TrueType 和 OpenType 字

体转换成 JavaScript 的库。转换出来的 JavaScript 文件可以包含在你的页面中，然后即可在 Three.js 中使用。

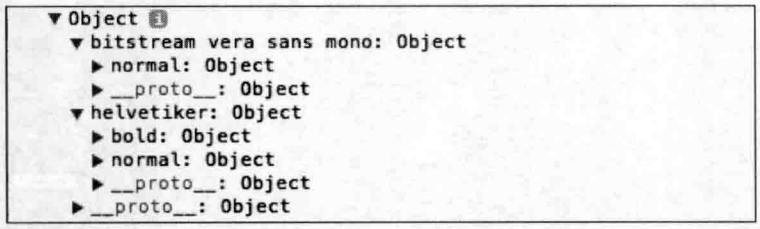
要转换已有的 OpenType 或 TrueType 字体，可以使用网页 http://typeface.neocracy.org/fonts.html。你可以在这个网页上上传一个字体，然后它就会帮你把该字体转换成 JavaScript。要包含这个字体，只要在你的 HTML 页面顶部加上如下几行代码即可：

```
<script type="text/javascript"
  src="../assets/fonts/bitstream_vera_sans_mono_roman.typeface.js">
</script>
```

这样即可加载该字体，并在 Three.js 中使用。如果你想知道字体的名字（用在 font 属性上），你可以使用如下的 JavaScript 代码，将字体缓存输出到控制台上：

```
console.log(THREE.FontUtils.faces);
```

这将输出如下图所示的内容：



从上图可以看出字体 helvetiker 能够使用权重 bold 和 normal。字体 bitstream vera sans mono 能够使用权重 normal。另外一种查看字体名的方法是看该字体的 JavaScript 源码。在这个文件的结尾你会找到一个名为 familyName 的属性。该属性中也包含字体的名字：

```
"familyName":"Bitstream Vera Sans Mono"
```

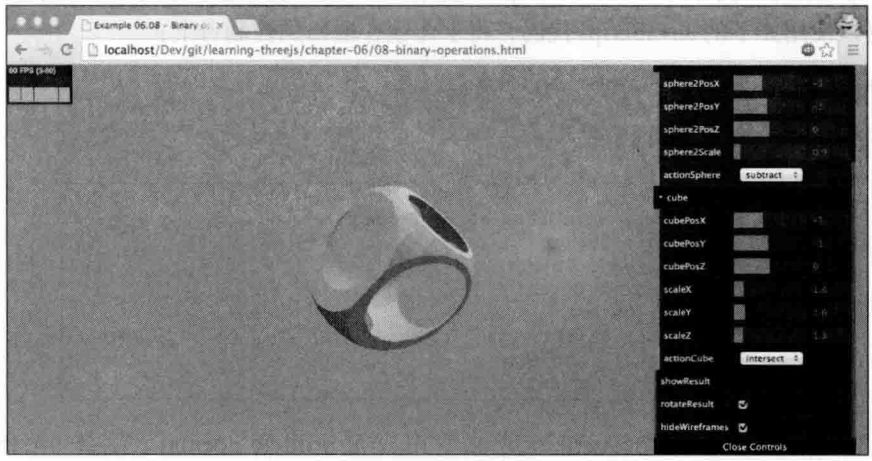
本章的下一部分我们将会介绍 THREEBSP 库，该库使用二元操作相交、相减和联合可以创造出非常有趣的几何体。

6.5　使用二元操作组合网格

本节我们将会看到一种特别的创建几何体的方法。在本章之前的小节，以及前一章里，我们一直使用 Three.js 默认提供的几何体来创建几何体。通过默认的属性集合，你可以创建出漂亮的模型，但是你所能做的也会受制于 Three.js 所提供的内容。在本节，我们将向你展示如何将各种标准几何体组合在一起创建出新的几何体。为此，我们将使用 Three.js 的扩展库 THREEBSP。你可以在网上找到这个库，网址是：https://github.com/skalnik/ThreeBSP。这个扩展库提供如下三个函数：

名　称	描　述
intersect（相交）	使用该函数可以在两个几何体的交集上创建出新的几何体。两个几何体相互交叠的地方就是新的几何体
union（联合）	union 函数可以将两个几何体联合在一起创建出新的几何体。你可以将这个函数与我们将会在第 8 章讨论的 mergeGeometry 函数相比较
subtract（相减）	subtract 函数与 union 函数相反。通过这个函数你可以在第一个几何体中减去两个几何体交叠的部分，从而创建出新的几何体

在下面的几节里我们将会详细介绍每个函数的细节。



上图所示的例子是依次使用 union 和 subtract 功能所可能创建出来的几何体。使用这个库需要把它包含在我们的网页中。该库是用 coffee-script（咖啡脚本）写的，这是一种对用户更加友好的 JavaScript 脚本的变体。要使用这个库我们有两个选择。我们可以添加 coffee-script 文件，并在运行时编译，或者将它预先编译成 JavaScript 文件，然后直接包涵编译后的文件。对于第一种方法，我们需要做的是：

```
<script type="text/javascript"
  src="../libs/coffee-script.js"></script>
<script type="text/coffeescript"
  src="../libs/ThreeBSP.coffee"></script>
```

ThreeBSP.coffee 文件中包含我们所需的功能，而 coffee-script.js 文件则可以解析 ThreeBSP 所用的 coffee 语言。最后一步是要保证在我们使用 ThreeBSP 功能之前，ThreeBSP.coffee 文件已经解析完毕。为此我们要在页面文件底部添加如下代码：

```
<script type="text/coffeescript">
  onReady();
</script>
```

然后将我们原先的匿名函数改名为 onReady：

```
function onReady() {
  // Three.js code
}
```

如果我们使用 coffee-script 的命令行工具预先将 coffee-script 编译成 JavaScript，我们就可以在网页中直接包含编译好的 JavaScript 文件。要将它编译成 JavaScript 代码可以使用如下的命令行：

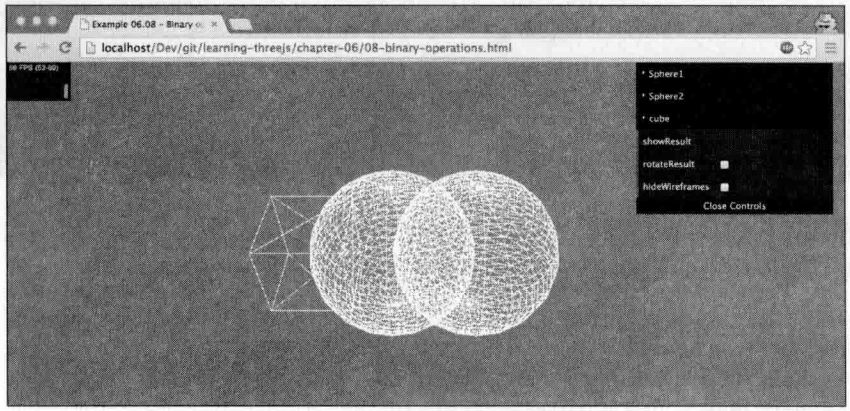
```
coffee --compile ThreeBSP.coffee
```

该命令可以创建出一个 ThreeBSP.js 文件，我们可以将它包含在示例文件中，就像我们使用其他 JavaScript 文件一样。在我们的示例里使用的是第二种方法，因为相比在加载网页时再编译 coffee-script 文件，这种方式加载起来比较快。

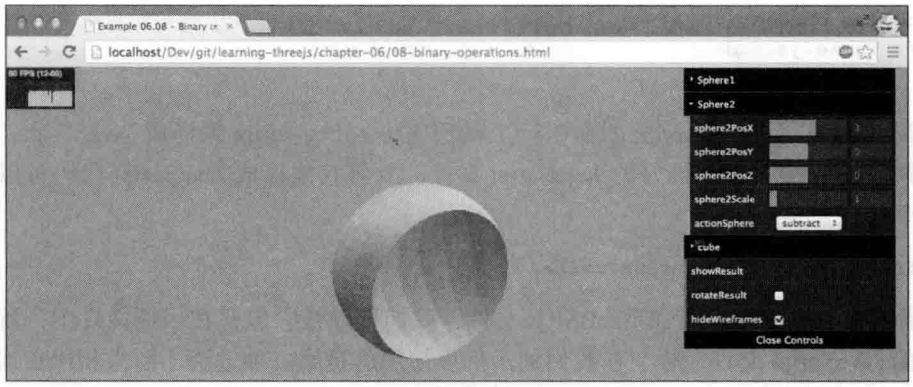
6.5.1　subtract 函数

在我们讲解 subtract 函数之前，有一个重要的步骤你需要记住。这三个函数在计算时使用的是网格的绝对位置。所以如果你在应用这些函数之前将网格组合在一起或者使用多种材质，你可能会得到一些奇怪的结果。为了得到最好的、可预测的结果，应当确保使用未经组合的网格。

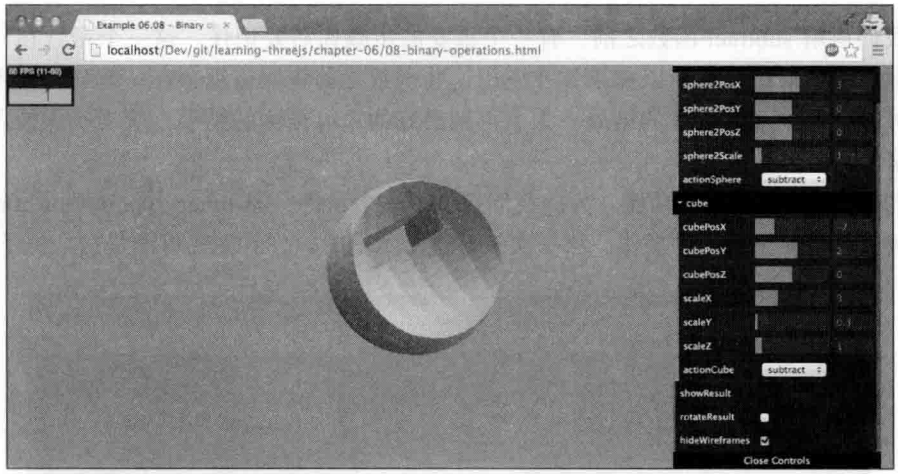
我们先来看看 subtract 功能。为此我们提供了一个示例：08-binary-operations.html。通过这个示例你可以使用这三种操作。第一次打开这个示例时，你看到的可能是这个样子：



该场景中有三个线框：一个方块和两个球体。Sphere1 是中间那个球，所有操作都会在这个对象上执行；Sphere2 是右边那个球，cube 是左边的方块。在 Sphere2 和 Cube 上可以指定四种操作中的一种：subtract、union、intersect 和 none（无操作）。这些操作都是基于 Sphere1 的。如果把 Sphere2 的操作设为 subtract，并选择 showResult，（隐藏线框）其结果就是从 Sphere1 中减去 Sphere1 和 Sphere2 交叠的区域。需要说明的是，点击 showResult 按钮后要过几秒钟这些操作才能完成。



在这个示例里，定义在 Sphere2 上的操作先执行，然后执行定义在 cube 上的操作。所以如果我们要减去 Sphere2 和 cube（沿 x 轴拉长了一些），将会得到如下结果：



理解相减操作的最好方式就是试验一下这个例子。完成该操作的 ThreeBSP 代码很普通，在本例中使用 redrawResult 函数实现。该函数在点击 showResult 按钮时调用：

```
function redrawResult() {
  scene.remove(result);
  var sphere1BSP = new ThreeBSP(sphere1);
  var sphere2BSP = new ThreeBSP(sphere2);
  var cube2BSP = new ThreeBSP(cube);

  var resultBSP;

  // first do the sphere
  switch (controls.actionSphere) {
    case "subtract":
      resultBSP = sphere1BSP.subtract(sphere2BSP);
      break;
```

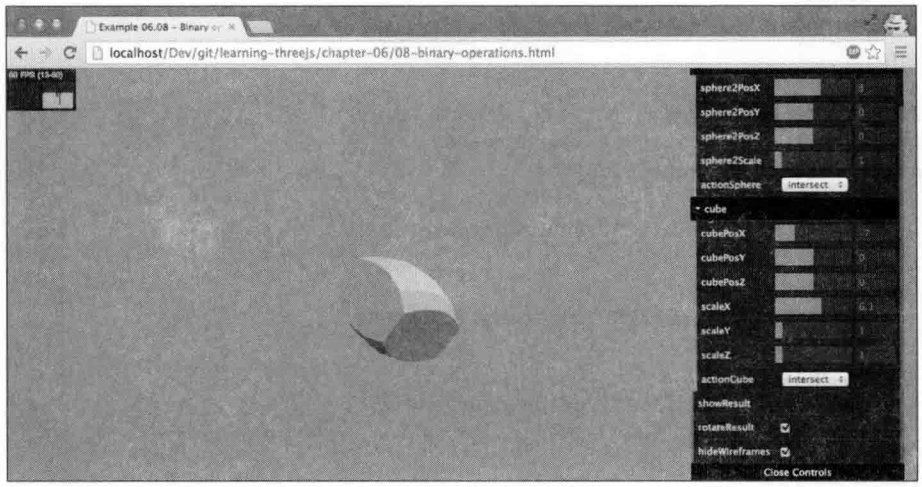
```
    case "intersect":
      resultBSP = sphere1BSP.intersect(sphere2BSP);
      break;
    case "union":
      resultBSP = sphere1BSP.union(sphere2BSP);
      break;
    case "none": // noop;
  }

  // next do the cube
  if (!resultBSP) resultBSP = sphere1BSP;
  switch (controls.actionCube) {
    case "subtract":
      resultBSP = resultBSP.subtract(cube2BSP);
      break;
    case "intersect":
      resultBSP = resultBSP.intersect(cube2BSP);
      break;
    case "union":
      resultBSP = resultBSP.union(cube2BSP);
      break;
   case "none": // noop;
  }


  if (controls.actionCube === "none"
    && controls.actionSphere === "none") {
    // do nothing
  }
  else {
    result = resultBSP.toMesh();
    result.geometry.computeFaceNormals();
    result.geometry.computeVertexNormals();
    scene.add(result);
  }
}
```

在代码中我们首先将网格（也就是那些线框）包装成一个 ThreeBSP 对象。只有这样才能在这些对象上调用 subtract、intersect 和 union 函数。现在我们只在中间那个球的 ThreeBSP 对象（sphere1BSP）上调用指定的函数，该函数的结果包含了所有创建网格所需的信息。要创建该网格，我们只需调用 toMesh() 函数，并通过调用 computeFaceNormals() 和 computeVertexNormals() 函数确保所有的法向量可以正确计算出来。之所以要调用这两个函数，是因为在执行二元操作之后，几何体中顶点和面的法向量可能会改变。Three.js 在着色时会用到面法向量和顶点法向量。明确地重新计算面和顶点的法向量，可以保证新生成的对象着色光滑（在材质的着色方法设置成 THREE.SmoothShading 时）、渲染正确，并且可以将结果添加到场景中。

对于 intersect 和 union 操作，我们使用的方法基本一致。

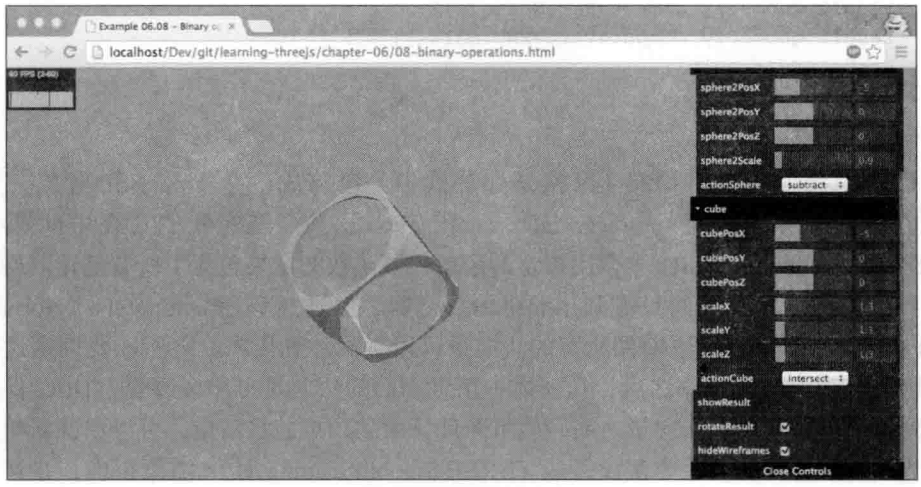
6.5.2 intersect 函数

我们已经在前一节解释了所有知识，有关 intersect 函数没有多少需要解释的了。调用该函数，只有网格交叠的部分可以保留下来。



如果你看过这个例子，并试验过这些设置，你会发现创建这些对象非常简单。要记住的是，这些操作可以应用到你创建的每个网格上，包括那些我们在本章看过的复杂网格，例如：ParametricGeometry 和 TextGeometry。

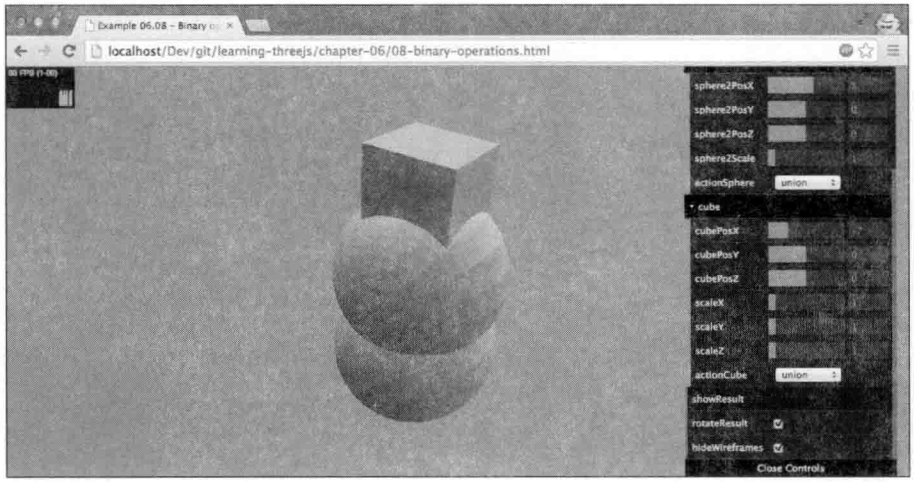
可以同时使用 subtract 和 intersect。我们在本章开头所展示的例子就是先减去一个小球，得到一个空心球，然后用一个方块跟这个空心球相交，得到想要的结果：



ThreeBSP 提供的最后一个函数是 union 函数。

6.5.3　union 函数

最后这个函数也是最无趣的一个。通过这个函数我们可以将两个网格连成一体，从而创建出新的几何体。所以当我们将这个函数应用于那两个球和方块时，我们将会得到一个单一的物体：



这并不是很有用，因为 Three.js 自己也提供了这个功能（参考第 8 章，我们会在那一章解释如何使用 THREE.GeometryUtils.merge），而且性能更好。如果旋转物体，你会发现这个联合操作（union）应用在中间那个球上，因为旋转的中心就是那个球的球心。其他两种操作也是一样。

6.6　总结

我们在本章学了很多东西。我们介绍了几种高级几何体，并且介绍了如何使用简单的二元操作创造出看上去很有趣的几何体。本章需要记住的最重要的东西是：

- 使用诸如 ConvexGeometry、TubeGeometry 和 LatheGeometry 你可以创造出相当有趣的几何体。再次提醒，你可以试验一下这些几何体。
- 将已有的 SVG 路径转换成 Three.js 路径是可能的。你可能需要使用诸如 Inkscape 的工具微调一下这个路径。
- 通过 ExtrudeGeometry，你可以轻松地将二维几何体转换成三维几何体。一般你可以沿着 z 轴进行拉伸，但也可以沿着自定义路径进行拉伸。
- 使用文本时需要指定字体。Three.js 提供了几种字体。你也可以创建自定义的字体，不过复杂字体往往不能正确地转换。
- 通过 ThreeBSP，你可以在网格上应用三种二元操作：union、subtract 和 intersect。通

过 union 你可以将两个网格联合在一起；通过 subtract，你可以从一个网格中移除它跟其他网格交叠的部分；通过 intersect，你可以只保留网格交叠的部分。

到目前为止我们看到的几何体都是固体（或线框），它们的顶点彼此相连构成物体表面。下一章我们将会看到另外一种展示几何体的方法，即所谓的粒子。使用粒子时我们不必渲染整个几何体，只要将顶点渲染成空间中的点即可。这样你就可以创建出漂亮的、高性能的三维效果。

第 7 章 Chapter 7

粒子和粒子系统

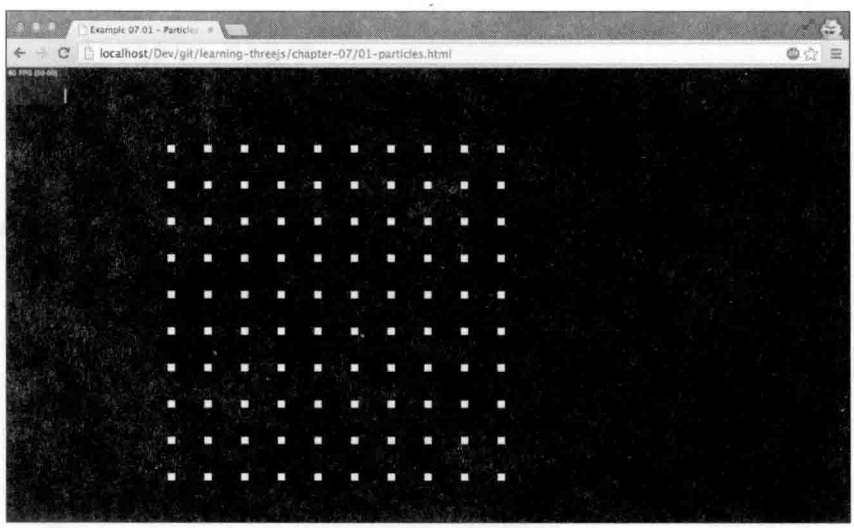
在前面的章节里我们讨论了最重要的一些概念、对象和 Three.js 提供的 API。本章，我们来看一下唯一一个我们一直忽略的概念：粒子（particle）。使用粒子可以很容易地创建出很多细小的物体，可以用来模拟雨滴和雪花。你也可以用粒子来创建有趣的三维效果。例如，你可以将某个单一几何体渲染成一组粒子，并分别对它们进行控制。本章我们将来探索一下 Three.js 提供的各种粒子功能。具体来说，我们将会在本章探讨如下的主题：

- 使用 ParticleBasicMaterial（基础粒子材质）创建和设计粒子。
- 使用 ParticleSystem（粒子系统）创建一个粒子集合。
- 使用已有的几何体创建一个粒子系统。
- 让粒子和粒子系统动起来。
- 用纹理为粒子造型。
- 使用 ParticleCanvasMaterial 在画布上为粒子造型。

让我们先从探索什么是粒子，以及如何创建粒子开始。

7.1 理解粒子

跟对待大多数新概念一样，我们将从一个示例开始。在本章的源码中你可以找到一个名为 01-particles.html 的示例。打开这个示例，你将会看到一些毫无趣味的白色小方块，如下图所示：



在这个截图里你所看到的是 100 个粒子。如果在创建粒子的时候没有指定任何属性，那么粒子就会被渲染成二维的白色小方块。创建这些粒子的代码如下所示：

```
function createParticles() {
    var material = new THREE.ParticleBasicMaterial();
    for (var x = -5; x < 5; x++) {
        for (var y = -5; y < 5; y++) {
            var particle = new THREE.Particle(material);
            particle.position.set(x * 10, y * 10, 0);
            scene.add(particle);
        }
    }
}
```

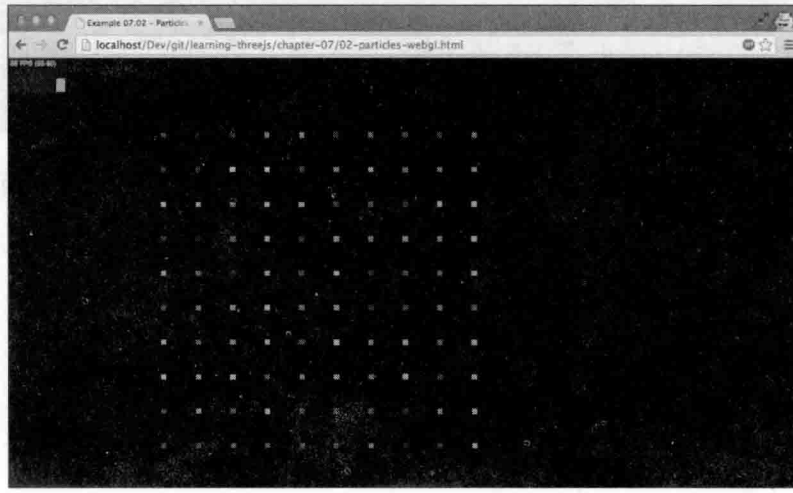
在这个例子里我们使用 THREE.Particle（material）构造函数手工创建粒子。唯一需要传入的参数是一个材质。该材质可以是 PatricleBasicMaterial，也可以是 ParticleProgramMaterial。我们将会在本章后面的章节里深入讲解这两种材质的细节。

在讲解有趣的粒子之前，我们先来仔细看一下 THREE.Particle。跟 THREE.Mesh 一样，THREE.Particle 也是 THREE.Object3D 对象的扩展。这也就是说 THREE.Mesh 大部分的属性和函数都可以用于 THREE.Particle。你可以用 Position 属性来定位，用 scale 属性来缩放，用 translate 属性来做相对位移。

对于这种创建粒子的方法，最后需要说明的是：如果你看过本示例的完整代码，你可能会发现我们使用的渲染器是 CanvasRenderer，而不是我们在大多数示例中使用的 WebGLRenderer。原因是创建粒子并直接添加到场景中，只对 CanvasRenderer 有效。如果是 WebGLRenderer，我们首先要创建一个 THREE.ParticleSystem 对象，然后通过这个对象来创建粒子。使用 WebGLRenderer 要获得前面截图所示的效果，我们需要使用如下的代码：

```
function createParticles() {

  var geom = new THREE.Geometry();
  var material = new THREE.ParticleBasicMaterial({size: 4,
    vertexColors: true, color: 0xffffff});

  for (var x = -5; x < 5; x++) {
    for (var y = -5; y < 5; y++) {
      var particle = new THREE.Vector3(x * 10, y * 10, 0);
      geom.vertices.push(particle);
      geom.colors.push(
        new THREE.Color(Math.random() * 0x00ffff));
    }
  }

  var system = new THREE.ParticleSystem(geom, material);
  scene.add(system);
}
```

正如你所看到的，我们需要为每个粒子创建一个顶点（用 Vector3 表示），并添加到一个几何体中，然后创建一个 PaticleSystem 对象，再把这个 ParticleSystem 对象添加到场景中。有关使用 WebGLRenderer 的实例可以看一下 02-particles-webgl.html，如下图所示：



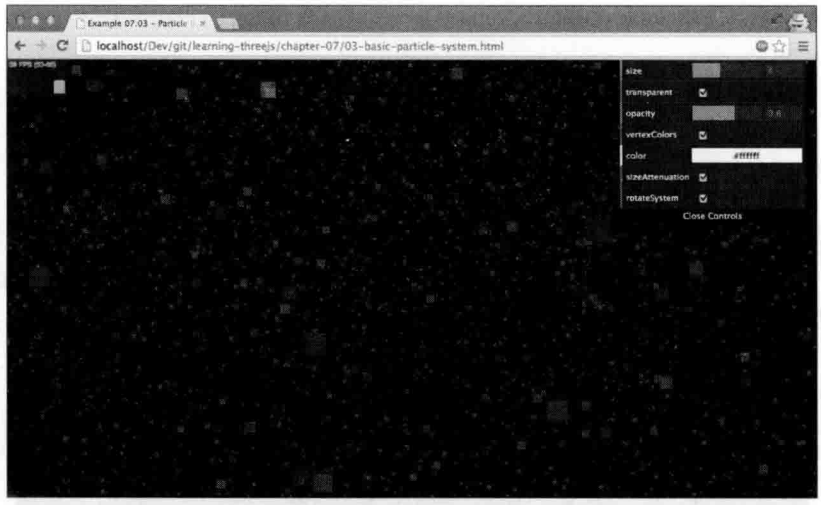
我们将会在下面的章节中进一步讨论 ParticleSystem。

7.2　粒子、粒子系统和 BasicParticleMaterial

在上节末尾我们简单介绍了一下粒子系统（ParticleSystem）。除非使用 CanvasRenderer 类，否则你就需要用 ParticleSystem 类来显示粒子。ParticleSystem 类的构造函数接受两个参

数：一个几何体和一个材质。材质用来给粒子上色和添加纹理（稍后即可看到），而几何体则是用来指定将粒子放在哪里。每个顶点，即定义几何体的各个点，将会以粒子的形态展示出来。如果我们基于一个 CubeGeometry 对象创建 ParticleSystem 对象，我们将会得到 8 个粒子，方块上的每个角一个。

但是一般来讲，我们不会使用标准的 Three.js 几何体来创建 ParticleSystem，而是从零开始手工将顶点添加到几何体上，就像我们在上一章末尾所做的那样。本节我们将进一步来看看这种方法，还要看看如何使用 BasicParticleMaterial 来格式化粒子。我们将会通过示例 03-basic-paticle-system.html 来讲解这些方法，如下图所示：



在这个例子里我们创建了一个 ParticleSystem，并添加了 15000 个粒子。所有这些粒子都用 BasicParticleMaterial 格式化。创建这个 ParticleSystem 的代码如下所示：

```
function createParticles(size, transparent, opacity,
  vertexColors, sizeAttenuation, color) {


    var geom = new THREE.Geometry();
    var material = new THREE.ParticleBasicMaterial({
      size: size,
      transparent: transparent,
      opacity: opacity,
      vertexColors: vertexColors,
      sizeAttenuation: sizeAttenuation,
      color: color
    });

    var range = 500;
    for (var i = 0; i < 15000; i++) {
      var particle = new THREE.Vector3(
```

```
            Math.random() * range - range / 2,
            Math.random() * range - range / 2,
            Math.random() * range - range / 2
        );
        geom.vertices.push(particle);
        var color = new THREE.Color(0x00ff00);
        color.setHSL(color.getHSL().h,
        color.getHSL().s,
        Math.random() * color.getHSL().l);
        geom.colors.push(color);
    }

    system = new THREE.ParticleSystem(geom, material);
    scene.add(system);
}
```

在上面列出的代码里，我们先创建了一个 THREE.Geometry 对象。然后我们会把用 THREE.Vector3 对象表示的粒子添加到这个几何体中。为此我们使用了一个简单的循环，在这个循环中我们在随机的位置上创建 THREE.Vector3，并把它添加到几何体中。在同一个循环中我们还定义了一个颜色数组，geom.colors。该数组只有在 ParticleBasicMaterial 的 vertexColors 属性设为 true 时才会用到。最后要做的是使用指定的属性创建 ParticleBasicMaterial 对象，创建 ParticleSystem，并添加到场景中。下表是 ParticleBasicMaterial 对象中所有可设置属性的说明：

名　称	描　述
color	ParticleSystem 对象中所有粒子的颜色。如果 vertexColors 属性值为 true，而且也指定了几何体的 colors 属性，那么该属性就会被忽略。默认值是 0xFFFFFF
map	通过这个属性可以在粒子上应用某种材质。例如可以让粒子看起来像雪花。本例中没有使用该属性，不过在本章下面的内容中会有进一步的解释
size	该属性指定粒子的大小。默认是 1
sizeAnnutation	如果设为 false，那么所有粒子都将拥有相同的尺寸，无论它们距离相机有多远。如果设为 true，粒子的大小决定于其距离相机的远近。默认是 true
vetexColors	通常情况下 ParticleSystem 里的所有粒子都具有相同的颜色。如果该属性设为 true，而且几何体的 colors 数组也有值，那就使用颜色数组中的值。默认是 false
opacity	跟 transparent 属性一起使用，用来设置粒子的透明度。默认是 1（不透明）
transparent	如果设为 true，那么粒子在渲染时会根据 opacity 属性的值来确定其透明度。默认是 false
blending	渲染粒子时的融合模式。有关融合模式的更多内容请参考第 9 章
fog	粒子是否受场景的雾化效果影响。默认是 true

前面的例子里有一个简单控制菜单，你可以用它来试验 ParticalBasicMaterial 的各种属性。

到目前为止我们还只是把粒子渲染成一些小方块，默认即是如此。但是，Three.js 还

提供了两种可以用来格式化粒子的方法。我们可以应用 ParticleCanvasMaterial 对象，将 HTML5 画布上绘制的内容作为粒子的纹理；或者使用 ParticleBasicMaterial 的 map（贴图）属性，加载外部的图片。下一节我们将来看看如何使用这两种方法。

7.3 使用 HTML5 画布格式化粒子

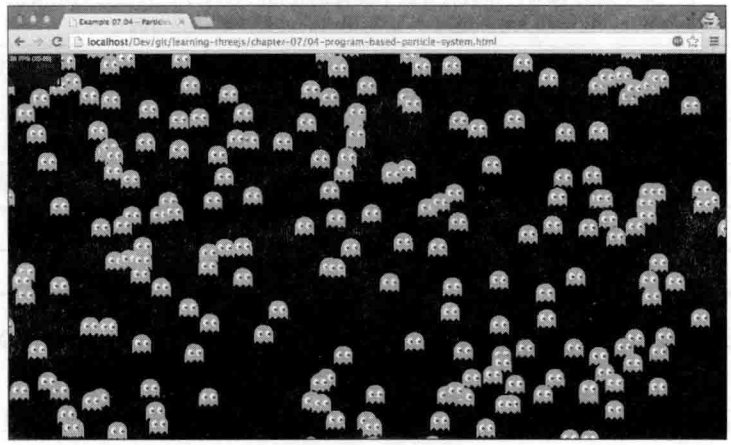
Three.js 提供了两种使用 HTML5 画布格式化粒子的方法。如果你使用的是 CanvasRenderer 类，那么你可以在 ParticleCanvasMaterial 对象里直接引用 HTML5 画布。如果你用的是 WebGLRenderer 类，那么需要采取一些额外的步骤，才能在格式化粒子时使用 HTML5 画布。我们将在下面的两节里分别讲述这两种方法。

7.3.1 在 CanvasRenderer 类里使用 HTML5 画布

通过 ParticleCanvasMaterial，你可以将 HTML5 画布的输出结果作为粒子的纹理。该材质是特别为 CanvasRenderer 创建的，而且只能用于这种渲染器。在解释如何使用该材质之前，我们先来看看该材质可以设置的属性：

名　称	描　述
color	粒子的颜色。根据特定的融合模式，可以影响画布的颜色
program	这是一个以画布上下文为参数的函数。该函数在渲染粒子时调用。调用该函数将在画布上下文中产生一个输出，该输出将会以粒子的形态显示出来
opacity	粒子的透明度。默认是 1，不透明
transparent	粒子是否透明。同 opacity 属性一起使用
blending	融合模式。更多细节请参考第 9 章

要想看一个 ParticleCanvasMaterial 实例，可以打开文件 04-program-based-particle-system.html：



在这个例子里，粒子是通过 createParticles 函数创建的：

```
function createParticles() {

  var material = new THREE.ParticleCanvasMaterial({
    program: draw,
    color: 0xffffff});
    var range = 500;
    for (var i = 0; i < 1000; i++) {
      var particle = new THREE.Particle(material);
      particle.position = new THREE.Vector3(
        Math.random() * range - range / 2,
        Math.random() * range - range / 2,
        Math.random() * range - range / 2);
      particle.scale = 0.1;
      particle.rotation.z = Math.PI;
      scene.add(particle);
  }
}
```

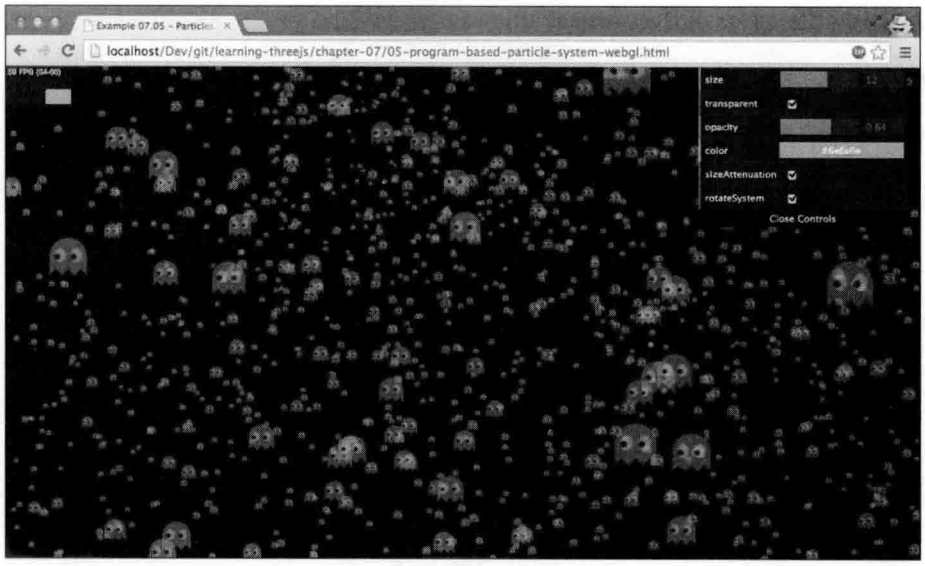
这段代码跟前一节的那段代码很像。主要的不同是：因为我们用的是 CanvasRenderer，所以我们可以直接创建 THREE.Particle 对象，而不必使用 ParticleSystem。在这段代码里，我们还定义了一个 ParticleCanvasMaterial 对象，该对象的 program 属性指向一个 draw 函数。draw 函数定义粒子的外观（Pac-Man 中的精灵）：

```
var draw = function(ctx) {
  ctx.fillStyle = "orange";
  ...
  // lots of other ctx drawing calls
  ...
  ctx.beginPath();
  ctx.fill();
}
```

我们不会详细讲解绘制粒子外形的画布代码。重要的是我们定义了一个函数，接受二维画布上下文作为参数。在这个上下文中绘制的结果将会作为粒子（THREE.Particle）的外形。

7.3.2　在 WebGLRenderer 中使用 HTML5 画布

如果要用 WebGLRenderer 类做同样的事情，我们不得不采用另外的方式。ParticleCanvasMaterial 在这里不能用，只能使用 ParticleBasicMaterial 来达到同样的目的。我们曾经提到 ParticleBasicMaterial 有一个 map（贴图）属性。通过 map 属性我们可以为粒子加载纹理。该纹理在 Three.js 中也可以是 HTML5 画布的输出。你可以在 05-program-based-particle-system-webgl.html 中找到这个概念的示例。其输出如下图所示：



让我们来看看实现该效果的代码。大部分代码跟之前那个 WebGL 的例子一样，所以我们不会解释太多细节。这个例子中最重要的改变是下面这段代码：

```
var getTexture = function() {
  var canvas = document.createElement('canvas');
  canvas.width = 32;
  canvas.height = 32;

  var ctx = canvas.getContext('2d');
  ...
  // draw the ghost
  ...
  ctx.fill();
  var texture = new THREE.Texture(canvas);
  texture.needsUpdate = true;
  return texture;
}

function createParticles(size, transparent, opacity,
  sizeAttenuation, color) {

    var geom = new THREE.Geometry();

    var material = new THREE.ParticleBasicMaterial({
      size: size,
      transparent: transparent,
      opacity: opacity,
      map: getTexture(),
```

```
    sizeAttenuation: sizeAttenuation,
    color: color});

  var range = 500;
  for (var i = 0; i < 5000; i++) {
    var particle = new THREE.Vector3(
      Math.random() * range - range / 2,
      Math.random() * range - range / 2,
      Math.random() * range - range / 2);
      geom.vertices.push(particle);
  }

  system = new THREE.ParticleSystem(geom, material);
  system.sortParticles = true;
  system.name = "particles";
  scene.add(system);
}
```

在这两个 JavaScript 函数中的第一个函数（getTexture）里面，我们基于 HTML5 画布创建了一个 THREE.Texture 对象。在第二个函数（createParticles）里面，我们将这个纹理赋予 map 属性。在这个函数里面你还可以看到我们将 ParticleSystem 对象的 sortParticles 属性设成了 true。这样可以保证粒子在渲染之前沿着屏幕上的 z 轴排好序。如果你发现有部分交叠的粒子或者透明度不正确，那么将这个属性设为 true 通常可以纠正类似的错误。在我们讨论 ParticleSystem 的属性时，有一个我们可以设置的属性叫作 ParticleSystem:FrustrumCulled。如果该属性设为 true，意味着落在相机可见范围外的粒子不会被渲染。必要时，使用该设置可以提高效率和帧频。

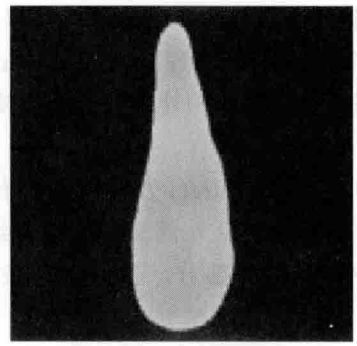
上述函数的调用结果是：在 getTexture() 方法中，所有画在画布上的东西都会用在粒子中。下一节我们将深入看一下如何从外部文件中加载纹理。在这个例子中需要记住的是，我们只看了纹理用途中很小的一部分。我们将会在第 10 章深入看一下如何使用纹理。

7.4　使用纹理格式化粒子

在前面的例子里我们看了如何使用 HTML5 的画布来格式化一个粒子系统。因为你可以画任何你想画的，甚至是加载外部的图片，所以你可以用该方法为粒子系统添加所有格式。但是还有一种更加直接的、使用图片格式化粒子的方法。在 Three.js 中可以使用 THREE.ImageUtils.loadTexture() 方法加载外部的图片。

我们将会在本节展示两个例子，并解释它们是如何创建的。这两个例子都是用图片作为粒子的纹理。在第一个例子（06-rainy-scene.html）里我们模拟了雨滴，如下图所示：

我们首先要做的是获取表示雨滴的纹理。你可以在 assets/textures/particles 文件夹下找到几个例子。我们将在第 9 章讲解使用纹理的所有细节和要求。现在你只要知道纹理应该是正方形的，尺寸是 2 的倍数（例如 64 × 64、128 × 128、256 × 256）。在本例中我们使用如下的纹理：



该图片使用了黑色的背景（为了能够正确地融合），展示了一个雨滴的形状和颜色。在 ParticleBasicMaterial 里使用该纹理之前，我们首先要加载它。可以用下面的代码来完成：

```
var texture = THREE.ImageUtils.loadTexture(
  "../assets/textures/particles/raindrop-2.png");
```

这行 Three.js 代码可以加载纹理，这样我们就可以在材质中使用它。在本例中我们使用如下的代码片段来定义材质：

```
var material = new THREE.ParticleBasicMaterial({
  size: 3,
  transparent: true,
```

```
    opacity: true,
    map: texture,
    blending: THREE.AdditiveBlending,
    sizeAttenuation: true,
    color: 0xffffff}
);
```

这些属性我们已经在本章中讨论过了。这里主要需要理解的是：map 属性指向我们用 THREE.ImageUtils.loadTexture() 函数加载的纹理，而且我们还将 blending（融合）模式设成了 THREE.AdditiveBlending。这个融合模式的含义是在画新像素时，背景像素的颜色会被添加到新像素上。对于我们的雨滴纹理来说，这意味着黑色背景不会显示出来。另外一种方式是将纹理中的黑色定义成透明的，但是这种组合在粒子和 WebGL 中不起作用。

这样粒子系统就格式化好了。当你打开这个示例时，你还会发现这些粒子都在移动。在前面的例子里我们移动的是整个粒子系统，这次我们设置的是粒子系统中每个单独粒子的位置。这其实非常简单。每个粒子都是构成 ParticleSystem 对象几何体上的顶点。我们来看看如何在这个 ParticleSystem 中添加粒子：

```
var range = 40;
for (var i = 0; i < 1500; i++) {
  var particle = new THREE.Vector3(
    Math.random() * range - range / 2,
    Math.random() * range * 1.5,
    Math.random() * range - range / 2);
  particle.velocityY = 0.1 + Math.random() / 5;
  particle.velocityX = (Math.random() - 0.5) / 3;
  geom2.vertices.push(particle);
}
```

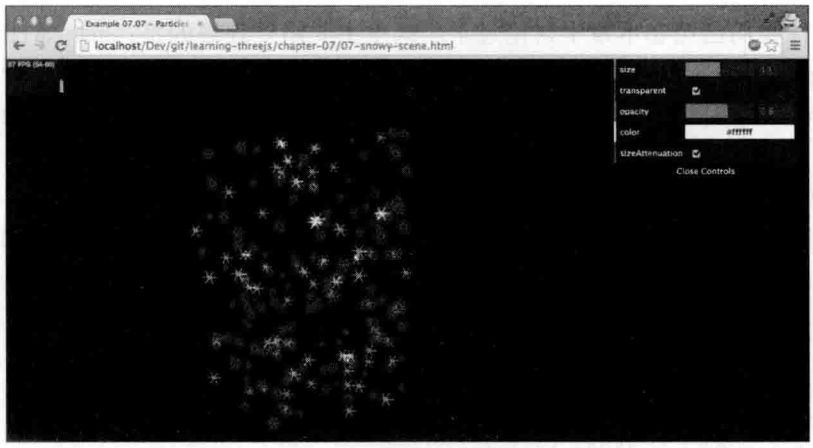
这跟我们前面看的例子没有太大差别。唯一改变的是每个粒子（THREE.Vector3 对象）上都增加了两个属性：velocityX 和 velocityY。第一个属性定义的是粒子（水滴）以多快的速度横向移动，第二个属性定义的是粒子以多快的速度下降。横向运动速度的范围是 -0.16 到＋ 0.16，纵向运动速度的范围是 0.1 到 0.3。现在每个雨滴都有自己的速度，我们可以在渲染循环体中移动每一个粒子，代码如下所示：

```
var vertices = system2.geometry.vertices;
vertices.forEach(function (v) {
  v.y = v.y - (v.velocityY);
  v.x = v.x - (v.velocityX);

  if (v.y <= 0) v.y = 60;
  if (v.x <= -20 || v.x >= 20) v.velocityX = v.velocityX * -1;
});
```

在这段代码中我们从几何体中获取用来创建 ParticleSystem 对象的所有顶点（vertices，即粒子）。对于每个粒子我们用 velocityX 和 velocityY 来改变它们的当前位置。最后两行用来保证粒子处在我们定义的范围内。如果 y 方向的位置低于 0，我们就把雨滴放回顶部；如

果 x 方向的位置超过了任何一条边界，我们就把横向运动速度取反，让雨滴反弹。

我们来看看另一个例子。这次我们模拟的不是雨，而是雪。另外我们也不会使用单一纹理，而是使用五个不同的图片（取自 Three.js 的例子）。同样，我们先来看看结果（参考 07-snowy-scene.html）：



在这张图片里你可以看到我们用多张图片作为纹理，而不是一张。你可能会奇怪我们是怎么做到的。你应该还记得一个 ParticleSystem 只能有一种材质。如果要用多个材质，那么只能使用多个粒子系统：

```
function createParticles(size, transparent, opacity,
  sizeAttenuation, color) {

  var texture1 = THREE.ImageUtils.loadTexture(
    "../assets/textures/particles/snowflake1.png");
  var texture2 = THREE.ImageUtils.loadTexture(
    "../assets/textures/particles/snowflake2.png");
  var texture3 = THREE.ImageUtils.loadTexture(
    "../assets/textures/particles/snowflake3.png");
  var texture4 = THREE.ImageUtils.loadTexture(
    "../assets/textures/particles/snowflake5.png");

  scene.add(createSystem("system1", texture1, size, transparent,
    opacity, sizeAttenuation, color));
  scene.add(createSystem("system2", texture2, size, transparent,
    opacity, sizeAttenuation, color));
  scene.add(createSystem("system3", texture3, size, transparent,
    opacity, sizeAttenuation, color));
  scene.add(createSystem("system4", texture4, size, transparent,
    opacity, sizeAttenuation, color));
}
```

在这里你可以看到我们对纹理分别进行加载，然后将所有信息传递给创建 ParticleSystem

的 createSystem 函数。该函数代码如下所示：

```
function createSystem(name, texture, size, transparent, opacity,
  sizeAttenuation, color) {
    var geom = new THREE.Geometry();

    var color = new THREE.Color(color);
    color.setHSL(color.getHSL().h,
      color.getHSL().s,
      (Math.random()) * color.getHSL().l);

    var material = new THREE.ParticleBasicMaterial({
      size: size,
      transparent: transparent,
      opacity: opacity,
      map: texture,
      blending: THREE.AdditiveBlending,
      depthWrite: false,
      sizeAttenuation: sizeAttenuation,
      color: color});

    var range = 40;
    for (var i = 0; i < 50; i++) {
      var particle = new THREE.Vector3(
        Math.random() * range - range / 2,
        Math.random() * range * 1.5,
        Math.random() * range - range / 2);
      particle.velocityY = 0.1 + Math.random() / 5;
      particle.velocityX = (Math.random() - 0.5) / 3;
      particle.velocityZ = (Math.random() - 0.5) / 3;
      geom.vertices.push(particle);
    }

    var system = new THREE.ParticleSystem(geom, material);
    system.name = name;
    system.sortParticles = true;
    return system;
}
```

在这个函数里我们首先做的是为要渲染的粒子材质指定颜色。做法是随机改变传入的颜色的“亮度”。接下来是跟以前一样创建材质。这里唯一不同的是将 depthWrite 属性设置为 false。该属性决定这个对象是否影响 WebGL 的深度缓存。将它设成 false，可以保证各个粒子系统之间不会互相影响。如果不是如此设置，那么当一个粒子处在另外一个粒子的前面，而后者来自于别的粒子系统，有时候你会看到纹理的黑色背景。这段代码的最后一步是随机放置粒子，并随机设定每个粒子的速度。现在我们可以在渲染循环里刷新每个粒子系统中的粒子，代码如下所示：

```
scene.children.forEach(function (child) {
  if (child instanceof THREE.ParticleSystem) {
    var vertices = child.geometry.vertices;
```

```
    vertices.forEach(function (v) {
      v.y = v.y - (v.velocityY);
      v.x = v.x - (v.velocityX);
      v.z = v.z - (v.velocityZ);

    if (v.y <= 0) v.y = 60;
    if (v.x <= -20 || v.x >= 20)
      v.velocityX = v.velocityX * -1;
    if (v.z <= -20 || v.z >= 20)
      v.velocityZ = v.velocityZ * -1;
    });
  }
});
```

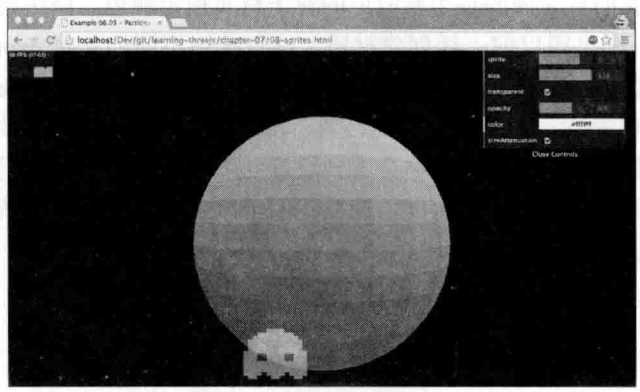
这样我们就可以为每个粒子赋予不同的纹理。但是该方法有一些限制。我们想要的纹理种类越多，那么需要创建和管理的粒子系统也就越多。如果能够使用单个粒子并格式化（就像我们在本章开头所展示的 CanvasRenderer），就会简单很多。但是如果使用 CanvasRenderer，我们很快就会遇到性能问题。而使用单一的 THREE.Pariticle 类，又不能跟 WebGLRenderer 一块儿用。但还有一种方式，就是我们曾经提到过的——THREE.Sprite（精灵）。

使用精灵

THREE.Sprite 类可以用于如下两种目的：

- 创建一个可以基于屏幕坐标移动、定位和缩放的对象。你可以用它来创建一个平视显示器（Head-Up display，简称 HUD），就像在三维场景上蒙了一层。
- 创建一个类似粒子的、可以在三维空间移动的对象，类似使用 CanvasRenderer 的 THREE.Particle。三维场景中的精灵有时也称作广告牌。所谓广告牌指的是精灵总是面向镜头，就像高速路上的广告牌总是面向司机。

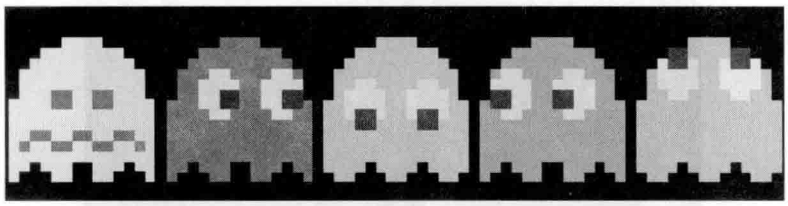
这两种情形我们都会看一下，先从第一个开始。在这个例子里我们会创建一个简单的 THREE.Sprite 对象，从左到右划过屏幕。作为背景，我们会渲染一个带有移动相机的三维场景，用来说明精灵的移动是独立的。下图所示就是我们在第一个例子（08-sprites.html）里创建的结果：



如果你在浏览器里打开这个例子，你就会看到一个类似 Pac-Man（吃豆人）的精灵绕着屏幕移动，并在碰到右侧边框时改变颜色和外形。所以我们最好从解释如何构建 THREE.Sprite，以及如何加载精灵的各种外形开始：

```
function getTexture() {
  var texture = new THREE.ImageUtils.loadTexture(
    "../assets/textures/particles/sprite-sheet.png");
  return texture;
}

function createSprite(size, transparent, opacity, color,
  spriteNumber) {
    var spriteMaterial = new THREE.SpriteMaterial({
      opacity: opacity,
      color: color,
      transparent: transparent,
      useScreenCoordinates: true,
      map: getTexture()}
    );

    // we have 1 row, with five sprites
    spriteMaterial.uvOffset.set(1 / 5 * spriteNumber, 0);
    spriteMaterial.uvScale.set(1 / 5, 1);
    spriteMaterial.alignment = THREE.SpriteAlignment.bottomCenter;
    spriteMaterial.scaleByViewport = true;
    spriteMaterial.blending = THREE.AdditiveBlending;

    var sprite = new THREE.Sprite(spriteMaterial);
    sprite.scale.set(size, size, size);
    sprite.position.set(200, window.innerHeight - 2, 0);
    sprite.velocityX = 5;

    scene.add(sprite);
}
```

我们在 getTexture() 函数里加载纹理。我们加载的是一张包含所有精灵图形的图片，而不是为每个精灵各自加载一张（总共 5 张），如下图所示：



通过 uvOffset（偏移）和 uvScale（缩放）属性，我们可以正确地选择要显示的精灵图形。通过 uvOffset 属性可以决定纹理在 x 轴（u）和 y 轴（v）上的偏移量。缩放比例的取值范围是 0 到 1。在我们的例子里，如果要选择第 3 个精灵，需要将 u 偏移（x 轴）设为 0.4。由于

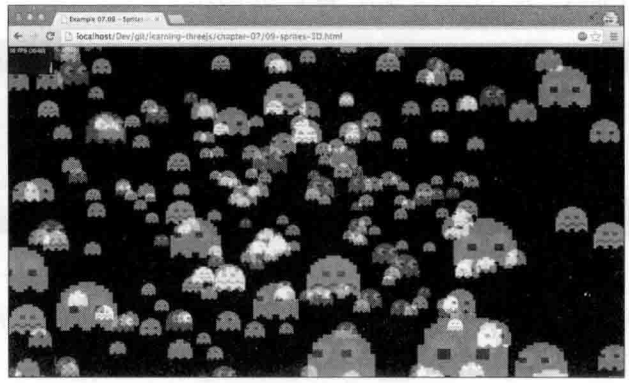
只有一行图片，所以不必改变 v 偏移（y 轴）。如果我们只设置该属性，那么显示出来的纹理是第 3、4、5 个图形压缩在一起。要想只显示其中一个，我们还需要放大。为此我们可以把 uvScale 属性中的 u 值设为 1/5。这意味着我们会放大（只针对 x 轴）纹理，只显示其中的 20%，也就是一个精灵。

另外一个需要解释的属性是 useScreenCoordinates。如果该属性设为 true，那么你可以只使用相对于窗口左上角的 x 和 y 坐标来定位精灵。当该属性为 true 时，场景中的相机就会被完全忽略。其他属性如下表所示：

名　称	描　述
Color	粒子的颜色
Map	精灵所用纹理。例如本节例子中所用的 sprite-sheet 图片
sizeAnnutation	如果设为 false，那么距离镜头的远近不会影响精灵的大小。默认是 true
opacity	设置精灵的透明度。默认是 1（不透明）
blending	渲染精灵时所用的融合模式。有关融合模式的更多信息，可以参考第 9 章
fog	精灵是否受场景中的雾化效果影响。默认是 true
useScreenCoordinates	如果设为 true，精灵的位置就是绝对位置。原点是屏幕的左上角
scaleByViewport	精灵的大小取决于视图窗口的尺寸。如果设为 true，那么精灵的尺寸 = 图片宽度 / 视图窗口高度。如果设为 false，那么精灵的尺寸 = 图片宽度 / 1.0
alignment	当精灵被缩放的时候（使用 scale 属性），该属性指定精灵从哪里开始缩放。如果将该属性设为 THREE.SpriteAlignment.topLeft，那么当增加或减少精灵的缩放比例时，精灵的左上角保持不动
uvOffset	结合 uvOffset 属性，选择精灵所用的纹理。参考本节代码示例中的解释
uvScale	结合 uvScale 属性，选择精灵所用的纹理。参考本节代码示例中的解释

你还可以在这个材质上设置 depthTest 属性和 depthWrite 属性。有关这些属性的更多信息，可以参考第 4 章。

在进入粒子的最后一节之前，我们来看一看 THREE.Sprite 的第二个用途：可以在三维空间中定位的粒子。我们也为这个用途创建了一个示例：09-sprites-3D.html，其输出结果如下图所示：



如果你仔细看看上表中列出的属性，那么你很容易想出实现该效果的代码：

```
function createSprite(size, transparent, opacity, color,
  spriteNumber, range) {

    var spriteMaterial = new THREE.SpriteMaterial({
      opacity: opacity,
      color: color,
      transparent: transparent,
      useScreenCoordinates: false,
      sizeAttenuation: true,
      map: getTexture()}
    );

    // we have one row, with five sprites
    spriteMaterial.uvOffset.set(1 / 5 * spriteNumber, 0);
    spriteMaterial.uvScale.set(1 / 5, 1);
    spriteMaterial.alignment = THREE.SpriteAlignment.bottomCenter;
    spriteMaterial.blending = THREE.AdditiveBlending;

    var sprite = new THREE.Sprite(spriteMaterial);
    sprite.scale.set(size, size, size);
    sprite.position = new THREE.Vector3(
      Math.random() * range - range / 2,
      Math.random() * range - range / 2,
      Math.random() * range - range / 2);
    sprite.velocityX = 5;
  return sprite;
}
```

在这段代码中我们使用之前所示的图片创建了 400 个精灵。你应该已经知道并理解这里用到的大部分属性和概念。如果要在三维空间中使用精灵，那么主要应记住的是将 useScreenCoordinates 属性设为 false。如果将这个属性设为 false，那么精灵的行为就会跟我们在本章其他部分所讨论的粒子一样。由于所有粒子已经添加到了一个组中，所以旋转这些粒子非常简单，代码如下所示：

```
group.rotation.x+=0.1;
```

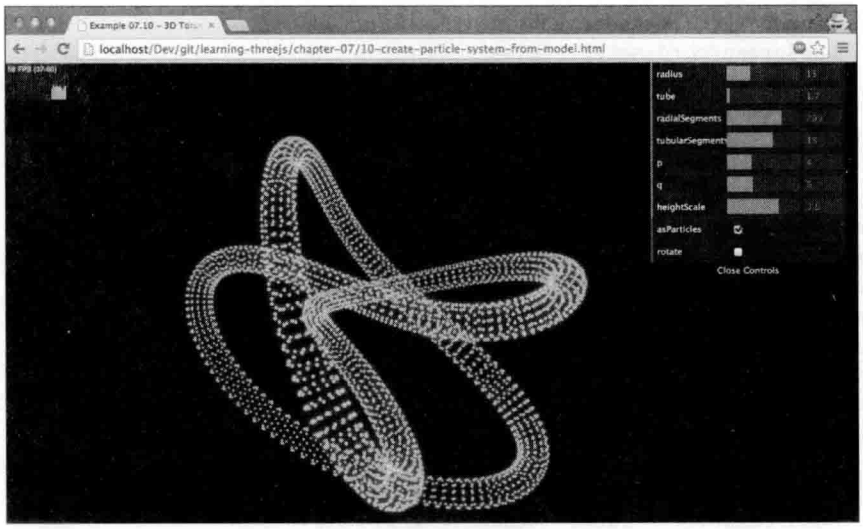
到目前为止本章主要探讨的是如何从零开始创建粒子、精灵和粒子系统。但还有一个有趣的事情是从已有的几何体中创建粒子系统。

7.5　从高级几何体中创建粒子系统

正如你所记得的，粒子系统所渲染的粒子来自于几何体的顶点。这也就是说如果我们提供一个复杂几何体，诸如环面纽结或者管道，我们就可以基于这个几何体的顶点创建出一个粒子系统。在本章最后一节里，我们将会创建一个环面纽结（类似我们在前一章所看到的那

个），然后将它渲染成一个粒子系统。

我们在前一章已经描述过环面纽结了，所以在这里不再涉及过多的细节。我们使用的代码跟前一章的完全相同。我们还添加了一个菜单选项，你可以用它将这个网格转换成粒子系统。你可以在 10-create-particle-system-from-model.html 里找到该示例的代码，其输出如下图所示：



正如你所看到的，每一个用来生成环面纽结的顶点都是一个粒子。我们还在这个例子里用 HTML 画布添加了一种漂亮的材质，用来创建这种发光效果。我们只要看一下创建材质和粒子系统的代码，其他属性我们在本章中已经讨论过了：

```
function generateSprite() {

  var canvas = document.createElement('canvas');
  canvas.width = 16;
  canvas.height = 16;

  var context = canvas.getContext('2d');
  var gradient = context.createRadialGradient(
    canvas.width / 2, canvas.height / 2,
    0,
    canvas.width / 2, canvas.height / 2,
    canvas.width / 2);

  gradient.addColorStop(0, 'rgba(255,255,255,1)');
  gradient.addColorStop(0.2, 'rgba(0,255,255,1)');
  gradient.addColorStop(0.4, 'rgba(0,0,64,1)');
  gradient.addColorStop(1, 'rgba(0,0,0,1)');

  context.fillStyle = gradient;
```

```
  context.fillRect(0, 0, canvas.width, canvas.height);

  var texture = new THREE.Texture(canvas);
  texture.needsUpdate = true;
  return texture;
}

function createParticleSystem(geom) {
  var material = new THREE.ParticleBasicMaterial({
    color: 0xffffff,
    size: 3,
    transparent: true,
    blending: THREE.AdditiveBlending,
    map: generateSprite()
  });

  var system = new THREE.ParticleSystem(geom, material);
  system.sortParticles = true;
  return system;
}

// use it like this
var geom = new THREE.TorusKnotGeometry(...);
var knot = createParticleSystem(geom);
```

在这段代码里你会看到两个函数：createParticleSystem() 和 generateSprite()。在第一个函数里，我们直接从指定的几何体（在本例中是一个环面纽结）中创建了一个简单的粒子系统，然后通过 generateSprite() 函数将粒子的纹理（map 属性）设置成发光点（在 HTML5 的画布元素上生成）。

7.6　总结

这就是本章所有的内容了。我们解释了什么是粒子、精灵和粒子系统，以及如何使用相应的材质格式化这些对象。需要记住的重要内容如下所示：

- 如果使用 CanvasRenderer 类，那么你可以直接使用 THREE.Particle 对象。
- 如果使用 WebGLRenderer 类，则不能使用 THREE.Particle 对象，但你可以用 THREE.Sprite 对象来创建一个粒子。
- 如果你想创建大量粒子，并共享同一个材质，那么你应该使用 THREE.ParticleSystem 对象。这样几何体中的每个顶点都会被渲染成粒子，并使用指定的材质。
- 你可以很容易地让粒子动起来，只要改变它们的位置即可。THREE.Particle、THREE.Sprite，以及用来创建粒子系统的几何体中的顶点都可以如此使用。
- 通过 map 属性，你可以使用图片或者 HTML5 的画布来格式化粒子。
- 你也可以使用 THREE.Sprite 类来为三维场景创建一种贴图对象。将该对象的 useScreen-

Coordinates 属性设为 true，那么该对象就会按照屏幕上绝对位置来定位。

一直到本章，我们都是用 Three.js 提供的几何体来创建网格。这对简单的几何体（例如球和方块）来说足够用了，但当你想要创建复杂的三维模型时就不太好用了。通常情况下你可以用三维建模工具（如 Blender 和 3D Studio Max）来创建复杂几何体。下一章，你将学习如何加载和展示这些三维建模工具所创建的模型。

第 8 章 Chapter 8

创建、加载高级网格和几何体

在这一章里我们会看一下创建高级、复杂几何体和网格的几种方法。我们曾经在第 5 章向你展示了如何使用 Three.js 自带的对象来创建高级几何体。在本章我们将使用下面两种方法来创建高级几何体和网格：

- 组合和合并：首先要讲解的是使用 Three.js 自带功能来组合和合并已有的几何体。从已有的几何体中创建出新的几何体。
- 从外部加载：在本章我们还会讲解如何从外部资源中加载网格和几何体。我们将会向你展示如何使用 Blender 导出 Three.js 支持的网格格式。

我们将从“组合和合并”方法开始。通过这种方法，我们使用标准的 Three.js 组合函数和 GeometryUtils.merge() 函数来创建新对象。

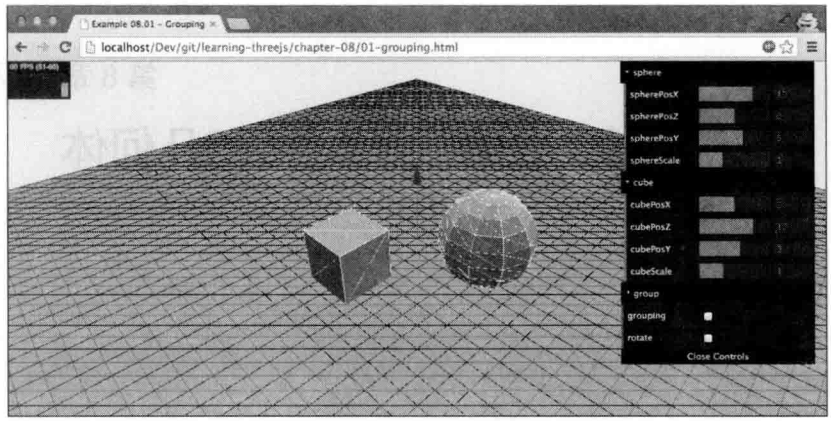
8.1 几何体组合和合并

我们将会在本节介绍 Three.js 的两个基本功能：将对象组合在一起，以及将多个网格合并成一个网格。我们先从对象组合开始讲起。

8.1.1 对象组合

在前面某些章节里使用多种材质时，你已经见过对象组合。当从一个几何体创建网格，并且使用多种材质时，Three.js 就会创建一个组。该几何体的多份副本会添加到这个组里，每份副本都有自己特定的材质。而这个组就是我们得到的结果，看上去就像是一个网格拥有多份材质。但是实际上它是一个包含多个网格的组。

创建一个组非常简单。每个你创建的网格都可以包含子元素，子元素可以使用 add 函数来添加。在组中添加子元素的效果是：你可以移动、缩放、旋转和变形父对象，而所有的子对象都会受到影响。让我们来看一个例子：01-grouping.html，如下图所示：



在这个例子里你可以使用控制菜单来移动球和方块。如果选中了 rotate 选项，你会看到这两个网格会绕着它们的中心旋转。这不是什么新东西，也并不令人激动。但是这两个网格并不是被直接添加到场景中的，而是添加到一个组中：

```
sphere = createMesh(new THREE.SphereGeometry(5, 10, 10));
cube = createMesh(new THREE.CubeGeometry(6, 6, 6));

group = new THREE.Object3D();
group.add(sphere);
group.add(cube);

scene.add(group);
```

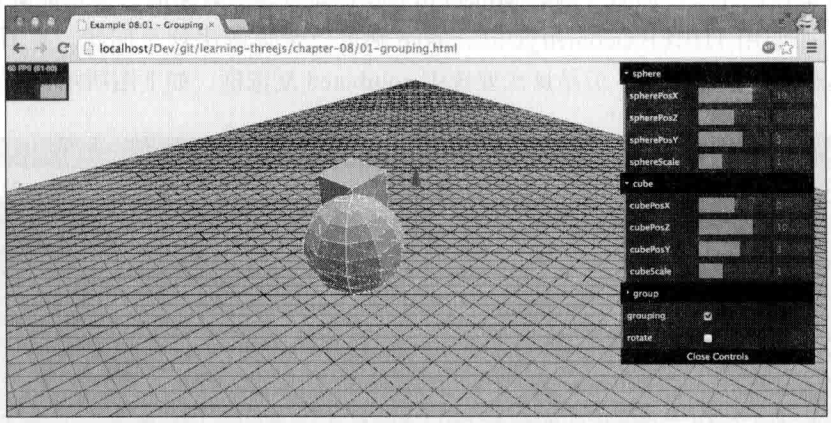
在这段代码里你可以看到我们创建了一个 THREE.Object3D 对象。这是 THREE.Mesh 对象和 THREE.Scene 对象的基类，但是它本身并不包含什么东西，也不会渲染任何东西。在这个例子里，我们使用 add 函数将 sphere 和 cube 添加到该对象中，并把它添加到 scene 实例。如果再看看这个例子，你会发现你依然可以移动球和方块，也可以对它们进行缩放和旋转。你也可以对这些对象所在的组做这些操作。如果看一下组菜单，你还会看到 position 和 scale 菜单项。你可以用这些菜单项来缩放、移动整个组。组中对象则会随着组的缩放和移动而缩放和移动。

缩放和移动很直观。但是要记住的是：当你旋转一个组时，并不是分别旋转组中的每一个对象，而是绕中心旋转整个组。在这个例子里，我在组的中心放置了一个箭头：

```
var arrow = new THREE.ArrowHelper(
  new THREE.Vector3(0, 1, 0), group.position, 10, 0x0000ff);
scene.add(arrow);
```

如果你选中了 grouping 和 rotate 选项，那么这个组就会开始旋转。而且你会看到球和方

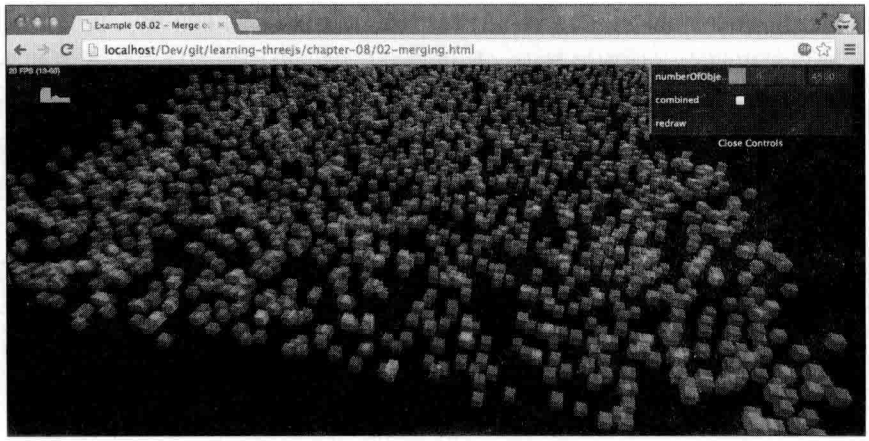
块一起绕着组的中心（箭头所指）旋转，如下图所示：



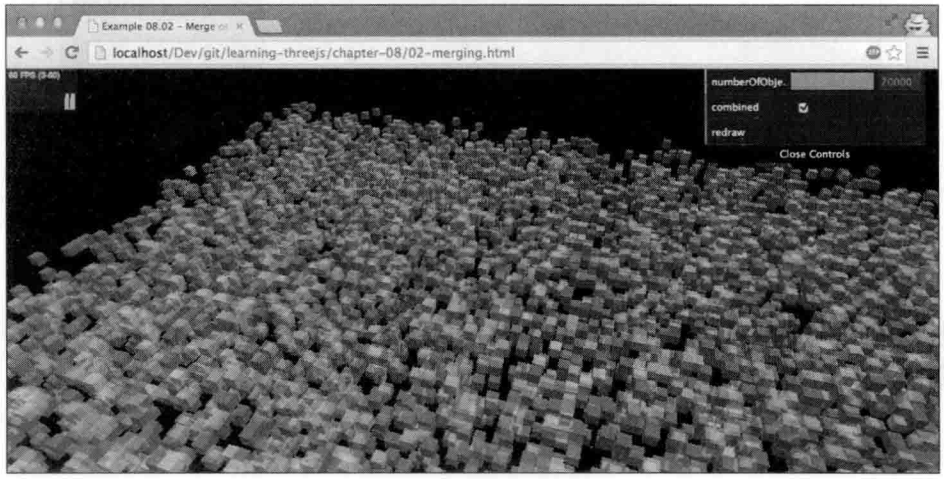
使用组的时候你依然可以引用、修改每一个单独的几何体。下一节我们将看一下合并，合并的结果是一个单独的新几何体。

8.1.2　将多个网格合并成一个网格

多数情况下使用组可以很容易操纵和管理大量网格。但是当对象的数量非常多时，性能就会成为一个瓶颈。在组里每个对象还是独立的，需要对它们分别进行处理和渲染。通过 THREE.GeometryUtils.merge 函数你可以将多个几何体合并起来，创建一个联合体。在下面的例子你将会看到如何使用该函数，以及它对性能的影响。打开示例 02-merging.html，你会看到一个随机分布着很多半透明方块的场景。通过菜单中的滑块，你可以设置场景中的方块数目，点击 redraw 按钮可以重新绘制场景。根据你所使用的硬件，当方块的数目达到一定量时，性能就开始下降。你可以从下图中看到，在我的环境中，性能在 4000 个对象左右时开始下降。刷新速度从通常的 60 fps 下降到了 40 fps。



正如你所看到的，场景中能够添加的网格数目有一个上限。一般来讲你并不需要那么多的网格，但是在创建某些游戏（例如 Minecraft）或者高级显示效果时，你可能会遇到这样的性能问题。而使用 THREE.GeometryUtils.merge 则可以帮你解决这个问题。在看代码之前，我们先来运行一下这个示例，但是这次要选中 combined 复选框，如下图所示：



正如你所看到的，我们可以轻松渲染 20 000 个方块，而且没有任何性能损失。为达到该效果我们要使用如下代码：

```
var geometry = new THREE.Geometry();
for (var i = 0; i < controls.numberOfObjects; i++) {
  THREE.GeometryUtils.merge(geometry, addCube());
}
scene.add(new THREE.Mesh(geometry, cubeMaterial));
```

在这段代码里，函数 addCube() 返回的是一个 THREE.CubeGeometry() 对象。通过执行“THREE.GeometryUtils.merge（geometry，addCube()）;”语句我们可以将方块几何体合并到 THREE.Geometry 对象，该对象一开始是空的。如此做 20 000 次，我们得到的将是一个几何体，然后可以将它添加到场景中。如果看过代码，你会发现这个方法有几个缺陷。由于我们得到的只是一个几何体，所以你不能为每个几何体添加材质。但是，这个问题可以用 THREE.MeshFaceMaterial 对象来解决。最大的缺陷是失去了对每个对象的单独控制。想要移动、旋转或者缩放某个方块是不可能的（除非你能搜索到相应的面和顶点，并分别对它们进行操作）。

通过组合和合并，你可以使用 Three.js 提供的基本几何体来创建大型的、复杂几何体。如果你想创建更加高级的几何体，那么使用 Three.js 所提供的编程方式就不是最好、最简单的方法。幸运的是 Three.js 提供了其他创建几何体的方法。下一节，我们将会学习如何从外部资源中创建、加载几何体和网格。

8.2　从外部资源中加载几何体

Three.js 可以读取几种三维文件格式，并从中导入结合体和网格。下表所列就是 Three.js 支持的以及我们将要在本节讲解的文件格式：

格　式	描　述
JSON	Three.js 有它自己的 JSON 文件格式，你可以用它以声明的方式定义几何体和场景。但它并不是一种正式的格式。它很容易使用，当你想要复用复杂的几何体或场景时非常有用
OBJ 和 MTL	OBJ 是一种简单的三维文件格式，由 Wavefront 科技公司创立。它是使用最广泛的三维文件格式，用来定义对象的几何体。MTL 文件常同 OBJ 文件一起使用，在一个 MTL 文件中，对象的材质定义在 OBJ 文件中
Collada	Collada 是一种用来定义 XML 类文件中数字内容的格式。这也是一种被广泛使用的格式，差不多所有的三维软件和渲染引擎都支持这种格式
STL	STL 是 STereoLithography（立体成型术）的缩写，广泛用于快速成型。例如三维打印机的模型文件通常都是 STL 文件
	Three.js 有一个可定制的 STL 导出工具，STLExporter.js。你可以用它将 Three.js 中的模型导出到一个 STL 文件
CTM	CTM 是由 openCTM 创建的文件格式。可以用来压缩存储表示三维网格的三角形面片
VTK	VTK 是由 Visualization Toolkit 定义的文件格式，用来指定顶点和面。VTK 有两种格式，Three.js 支持旧的，即 ASCII 格式。
PDB	这是一种非常特别的格式，由 Protein Databank（蛋白质数据银行）创建，用来定义蛋白质的形状。Three.js 可以加载并显示用这种格式描述的蛋白质
PLY	该格式全称是多边形（polygon）文件格式。通常用来保存三维扫描仪的数据

我们将会在下一章讲解动画时再来看看这个列表中的某些格式（以及另外一种格式 MD2）。现在，我们将从这个列表中的第一种格式，Three.js 独有的格式，开始学习。

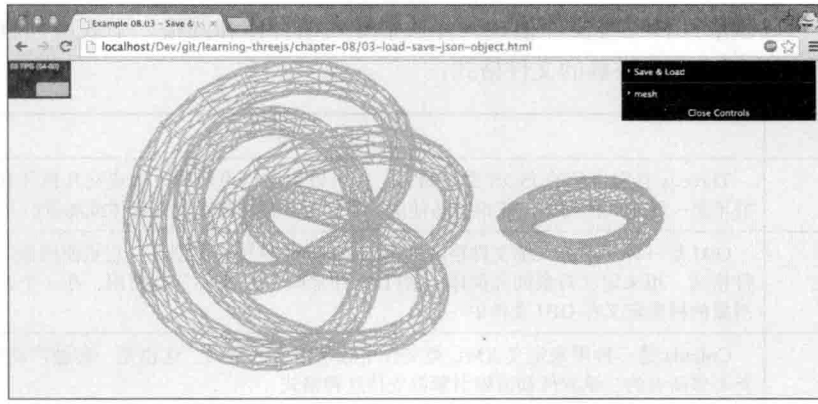
8.3　以 Three.js 的 JSON 格式保存和加载

你可以在两种情形下使用 Three.js 的 JSON 文件格式。你可以用它来保存和加载某个几何体，或者用它来保存和加载整个场景。

8.3.1　保存和加载几何体

为了展示保存和加载功能，我们基于 THREE.TorusKnotGeometry 类创建了一个简单的示例。在这个例子里你可以像我们在第 5 章所做的那样，创建一个环面纽结。然后使用 Save & Load 菜单中的 save 按钮保存当前的几何体。在这个例子里，我们使用了 HTML5 的本地存储 API。通过这个 API 可以很容易地将持久化信息保存在客户端浏览器里，以后还可以读取（即使在浏览器关闭，并重启之后）。

你可以在 03-load-save-json-object.html 中找到这个例子，如下图所示：



要让该示例能够工作，你首先要将 Three.js 发布包里的 GeometryExporter.js 包含在你的网页里。你可以在 examples/js/exporters 目录中找到该文件。

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

现在你可以用下面的 JavaScript 代码将几何体保存在浏览器的本地存储中：

```
var exporter = new THREE.GeometryExporter();
var result = exporter.parse(knot.geometry);
localStorage.setItem("json", JSON.stringify(result));
```

在保存之前，我们先要用 JSON.stringify 函数将 GeometryExporter 对象解析的结果，从一个 JavaScript 对象转换成一个字符串。JSON 字符串格式的结果如下所示（大部分顶点和面没有列出来）：

```
{
  "metadata": {
    "version": 4,
    "type": "geometry",
    "generator": "GeometryExporter"
  },

  "vertices": [14.000030624610355, -0.006999878543058498, ...],
  "uvs": [
    []
  ],
  "faces": [49, 0, 8, ...]
}
```

正如你所看到的，Three.js 保存的是原始的几何体。它将所有的顶点和面都保存了下来，但是你并不知道这是一个环、方块，还是别的什么。要把这些信息通过 HTML5 的本地存储保存下来，我们要做的只是调用 localStorage.setItem 函数。第一个参数是键值（json），以后

我们可以通过它将第二个参数传递的信息取出来。

将这个几何体加载到 Three.js 中只需要以下几行代码：

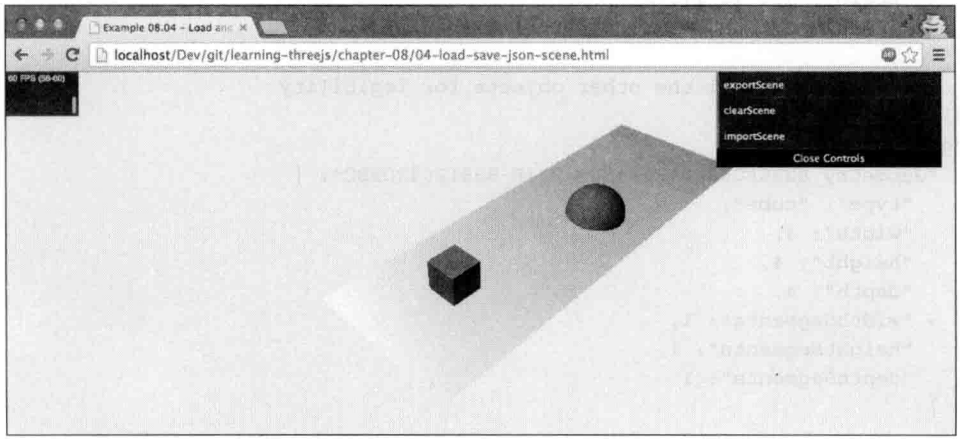
```
var json = localStorage.getItem("json");

if (json) {
  var loadedGeometry = JSON.parse(json);

  var loader = new THREE.JSONLoader();
  var geom = loader.parse(loadedGeometry);
  loadedMesh = createMesh(geom.geometry);
  loadedMesh.position.x = -35;
  loadedMesh.position.z = -5;
  scene.add(loadedMesh);
}
```

在这里我们先根据保存时指定的名字（本例中是 json）从本地存储中获取几何体。为此我们调用的是由 HTML5 本地存储 API 提供的 localStorage.getItem 函数。接下来我们将这个字符串（JSON.parse）转换成原来的 JavaScript 对象，然后用 THREE.JSONLoader 类将这个 JSON 对象转换成一个几何体。我们可以在这个几何体基础上创建网格，并将它添加到场景中。在这个例子里我用的是 JSONLoader（加载器）的 parse 方法来直接解析 JSON 字符串，而 JSONLoader 还提供了一个 load 函数，你可以传入一个 URL 地址，该地址指向一个含有 JSON 定义的文件。

正如你在这里所看到的，我们只保存了几何体。其他信息都丢失了。如果你想保存整个场景，包括材质、光源、位置等，可以使用 SceneExporter 对象。

8.3.2　保存和加载场景

如果你想保存整个场景，你同样可以使用在前一节里我们所看到的保存几何体的方法。你可以在 04-load-save-json-scene.html 里找到一个可以工作的示例，如下图所示：



在这个例子里，你有三个选项可以选择：exportScene（导出场景）、clearScene（清空场景）和 importScene（导入场景）。通过 exportScene，场景的当前状态就会被保存在浏览器的本地存储中。要测试场景的导入功能，你可以点击 clearScene 按钮将场景移除，然后点击 importScene 按钮从本地存储中加载场景。实现这些功能的代码很简单，但是在使用之前，你先要从 Three.js 发布包里引入所需的导出器（可以在 examples/js/exporters 目录中找到）。

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

通过页面中的导出器，你可以用下面的 JavaScript 代码将场景导出：

```
var exporter = new THREE.SceneExporter();
var sceneJson = JSON.stringify(exporter.parse(scene));
localStorage.setItem('scene', sceneJson);
```

这个方法跟我们在前一节所用的 GeometryExporter 类几乎一模一样，只是这次我们用的是 THREE.SceneExporter（ ）方法。但生成的 JSON 字符串稍有不同：

```
{
  "metadata": {
  "formatVersion": 3.2,
  "type": "scene",
  "generatedBy": "SceneExporter",
  "objects": 5,
  "geometries": 3,
  "materials": 3,
  "textures": 0
},
"urlBaseType": "relativeToScene",
"objects": {
  "Object_78B22F27-C5D8-46BF-A539-A42207DDDCA8": {
    "geometry": "Geometry_5",
    "material": "Material_1",
    "position": [15, 0, 0],
    "rotation": [-1.5707963267948966, 0, 0],
    "scale": [1, 1, 1],
    "visible": true
     }
  ... // removed all the other objects for legibility
},
"geometries": {
  "Geometry_8235FC68-64F0-45E9-917F-5981B082D5BC": {
    "type": "cube",
    "width": 4,
    "height": 4,
    "depth": 4,
    "widthSegments": 1,
    "heightSegments": 1,
    "depthSegments": 1
  }
```

```
    ... // removed all the other objects for legibility
}
  ... other scene information like textures
```

主要的区别是：这个导出器创建的 JSON 文件明确描述了物体、光源、材质，以及场景中的其他数据，而不是对象的原始信息。在加载的时候，Three.js 只是按照其导出时的定义重新创建而已。加载场景的代码如下所示：

```
var json = (localStorage.getItem('scene'));
var sceneLoader = new THREE.SceneLoader();
sceneLoader.parse(JSON.parse(json), function(e) {
  scene = e.scene;
}, '.');
```

传递给 loader（加载器）的最后一个参数（"."）是一个 URL 相对地址。例如，在材质中使用的纹理（如外部图片）就可以从这个相对地址中获取。在这个例子里我们没有使用纹理，所以只要传入当前目录即可。跟 GeometryLoader 一样，你也可以使用 load 函数直接加载一个 JSON 文件。

有很多的三维软件工具可以用来创建复杂的网格。其中有一个流行的开源软件叫做 Blender（www.blender.org）。它有（Maya 和 3D Studio Max 也都有）一个导出器可以直接将文件导出成 Three.js 的 JSON 格式。下一节，我们将会带领你配置 Blender，以便能够使用这个导出器，并向你展示如何在 Blender 里导出一个复杂模型，以便在 Three.js 里展示出来。

8.4　使用 Blender

在开始配置之前，我们先来看一下我们想要达到的效果。在下面的截图里你会看到一个简单的 Blender 模型，我们将用一个 Three.js 插件将它导出，并在 Three.js 中用 JSONLoader 类将它导入：

8.4.1 在 Blender 中安装 Three.js 导出器

要想从 Blender 中导出 Three.js 模型，我们首先要将 Three.js 导出器添加到 Blender 中。下面所列的步骤是 Mac OS X 上的，但对于 Windows 和 Linux 来讲也基本一样。你可以从 www.blender.org 上下载 Blender，然后按照相应平台上的安装提示进行安装。安装完之后就可以添加 Three.js 插件。首先使用终端窗口，在 Blender 的安装目录下找到 addons 文件夹，如下图所示：

```
Default
jos@Joss-MacBook-Pro.local:~/Downloads/Blender/blender.app/Contents/MacOS/2.68/scripts/addons$ ls -l | head
total 3104
drwxr-xr-x@  6 jos  staff     204 Jul 24 16:13 add_curve_extra_objects
-rw-r--r--@  1 jos  staff   26032 Jul 24 16:13 add_curve_ivygen.py
drwxr-xr-x@  5 jos  staff     170 Jul 24 16:13 add_curve_sapling
drwxr-xr-x@  7 jos  staff     238 Jul 24 16:13 add_mesh_BoltFactory
-rw-r--r--@  1 jos  staff   27994 Jul 24 16:13 add_mesh_ant_landscape.py
drwxr-xr-x@ 14 jos  staff     476 Jul 24 16:13 add_mesh_extra_objects
-rw-r--r--@  1 jos  staff   35389 Jul 24 16:13 add_mesh_pipe_joint.py
-rw-r--r--@  1 jos  staff   27191 Jul 24 16:13 add_mesh_solid.py
-rw-r--r--@  1 jos  staff   13200 Jul 24 16:13 animation_add_corrective_shape_key.py
jos@Joss-MacBook-Pro.local:~/Downloads/Blender/blender.app/Contents/MacOS/2.68/scripts/addons$
```

在我的 Mac 上，其路径是：./blender.app/Contents/MacOS/2.68/scripts/addons/。在 Windows 上这个路径可能是：

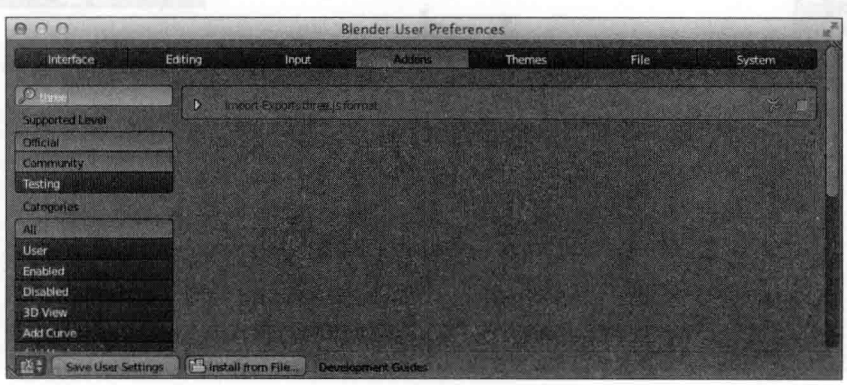
```
C:\Users\USERNAME\AppData\Roaming\Blender
Foundation\Blender\2.6X\scripts\addons
```

而在 Linux 上这个路径可能是：

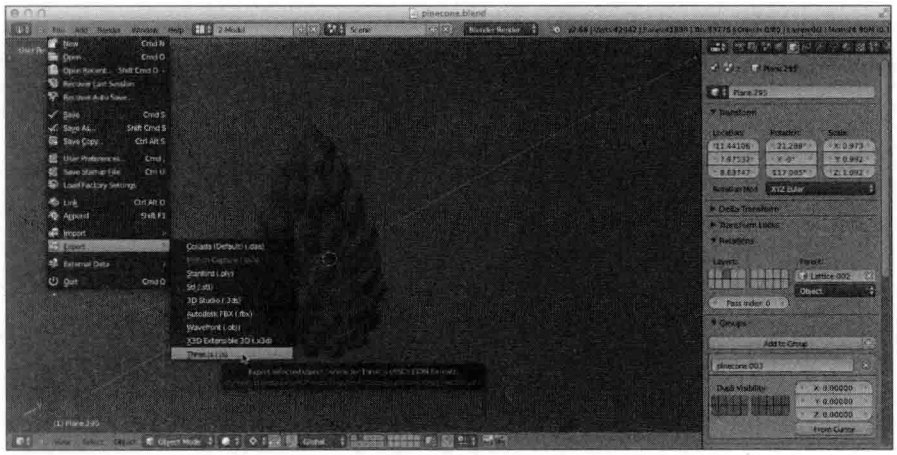
```
/home/USERNAME/.config/blender/2.6X/scripts/addons
```

接下来你要做的是获取 Three.js 的发布包，并解压到本地。在这个发布包里，你会看到目录：utils/exporters/blender/2.66/scripts/addons。在这个目录里有个子目录叫做 io_mesh_threejs。将这个目录复制到 Blender 安装目录里的 addons 文件夹下。

现在我们要做的就只是启动 Blender，激活导出器。在 Blender 里，打开 Blener User Preferences（File | User Preferences）。在弹出的窗口中找到 Addons 标签页，然后在搜索框里输入 three。结果如下图所示：



现在虽然已经找到了 Three.js 插件，但是它还没有激活。选择右面的复选框，Three.js 导出器就可以激活了。最后来检查下是不是所有设置都可以正确地工作。打开 File | Export 菜单项，你会发现 Three.js 出现在 Export 选项中。

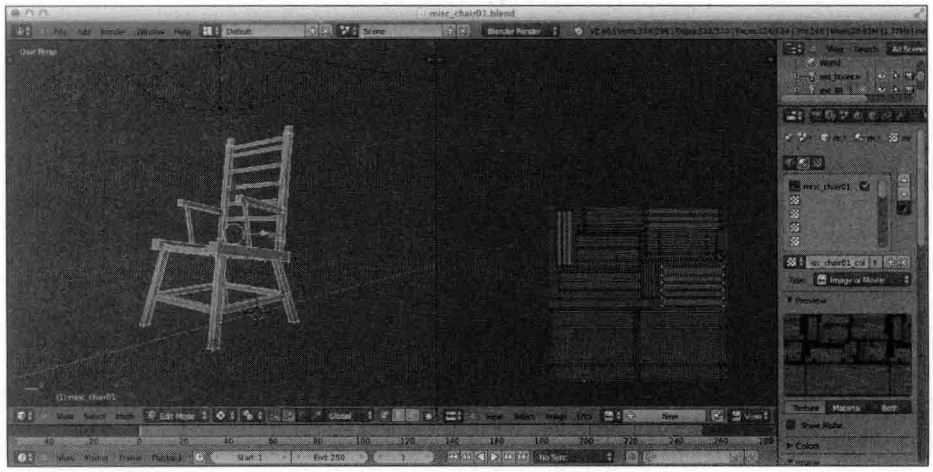


安装好插件后，我们就可以加载第一个模型了。

8.4.2　在 Blender 里加载和导出模型

作为示例，我们添加了一个简单的 Blender 模型，即 assets/models 文件夹下的 misc_chair01.blend，你可以在本书附带的源码中找到。我们将会在本节加载这个模型，并用最少的步骤将这个模型导出到 Three.js。

首先我们要将这个模型加载到 Blender。使用 File | Open 菜单，找到包含 misc_chair01.blend 文件的文件夹。选中这个文件，然后点击 Open。其结果如下图所示：



将这个模型导出成 Three.js.JSON 格式非常简单明了。在 File 菜单下找到 Export | Three.js，输入文件名，然后选择 Export Three.js。这样就可以创建出一个 Three.js 能够理解的 JSON 文件。该文件的部分内容如下代码所示：

```
{

  "metadata" :
  {
    "formatVersion" : 3.1,
    "generatedBy"   : "Blender 2.66 Exporter",
    "vertices"      : 208,
    "faces"         : 124,
    "normals"       : 115,
    "colors"        : 0,
    "uvs"           : [270,151],
    "materials"     : 1,
    "morphTargets"  : 0,
    "bones"         : 0
  },
...
```

正如你所看到的，这跟我们之前所看到的格式很相似。如果仔细看的话，你可能会注意到 formatVersion 不一样。在前面的例子里这个属性的值是 3.2，而在这个例子里则是 3.1。原因是 3.2 版是最近才引入的，而 Blender 的导出器还没有更新到这个新版本。但是我们还没有完全完成。在前面的屏幕截图中，你会发现这个椅子带有木质纹理。如果检查一下导出的 JSON 文件，你会发现导出的椅子中也指定了材质：

```
"materials": [{
  "DbgColor": 15658734,
  "DbgIndex": 0,
  "DbgName": "misc_chair01",
  "blending": "NormalBlending",
  "colorAmbient": [0.53132, 0.25074, 0.147919],
  "colorDiffuse": [0.53132, 0.25074, 0.147919],
  "colorSpecular": [0.0, 0.0, 0.0],
  "depthTest": true,
  "depthWrite": true,
  "mapDiffuse": "misc_chair01_col.jpg",
  "mapDiffuseWrap": ["repeat", "repeat"],
  "shading": "Lambert",
  "specularCoef": 50,
  "transparency": 1.0,
  "transparent": false,
  "vertexColors": false
}],
```

这个材质在 mapDiffuse 属性中标明了一个纹理：misc_chair01_col.jpg。所以除了导出模型，我们还需要确保 Three.js 能够找到这个纹理文件。幸运的是，我们可以在 Blender 里直

接保存这个纹理。

在 Blender 里打开 UV/Image Editor 视图。你可以从下拉菜单（File 菜单的左侧）中选择该视图。这样顶部的菜单就会被如下的菜单所取代：

确保选中（你可以用那个小的图片图标来选择）想要导出的纹理（本例是 misc_chair_01_col.jpg）。然后点击 Image 菜单，用 Save as Image 选项来保存图片。将它保存在跟模型相同的文件夹下，名字就是 JSON 文件中所用的名字。现在我们已经准备好在 Three.js 中加载模型了。

在 Three.js 中加载模型的代码如下所示：

```
var loader = new THREE.JSONLoader();
loader.load('../assets/models/misc_chair01.js', function
  (geometry, mat) {
    mesh = new THREE.Mesh(geometry, mat[0]);

    mesh.scale.x = 15;
    mesh.scale.y = 15;
    mesh.scale.z = 15;

    scene.add(mesh);

  }, '../assets/models');
```

前面我们已经见过了 JSONLoader 类，但是这次我们将使用 load 函数，而不是 parse 函数。在这个函数里我们要指定想要加载的 URL（指向导出的 JSON 文件），指定加载结束后的回调函数，指定纹理所在的路径（相对于该网页）../assets/models。这个回调函数接受两个参数 geometry 和 mat。geometry 参数包含模型对象，而 mat 参数包含的是一组材质对象。我们知道只有一个材质，所以在创建 THREE.Mesh 实例时，我们直接引用该材质。如果打开示例 05-blender-from-json.html，你就会看到我们刚刚从 Blender 导出的椅子。

使用 Three.js 的导出器并不是将 Blender 里的模型加载到 Three.js 的唯一方法。Three.js 能够理解好几种三维文件格式，而且 Blender 也可以导出这几种格式。但是使用 Three.js 格式非常简单，而且一旦哪里出错了可以很快找到。

下一节我们将会介绍 Three.js 支持的几种格式，并展示几个基于 Blender 的 OBJ 和 MTL 格式的文件。

8.5　导入三维格式文件

在本章开头我们列出了几个 Three.js 支持的格式。本节我们将快速浏览一下这些格式的几个例子。要注意的是所有这些格式都要引入一个额外的 JavaScript 文件。你可以在 Three.js 发布包的 examples/js/loaders 目录下找到这些文件。

8.5.1 OBJ 和 MTL 格式

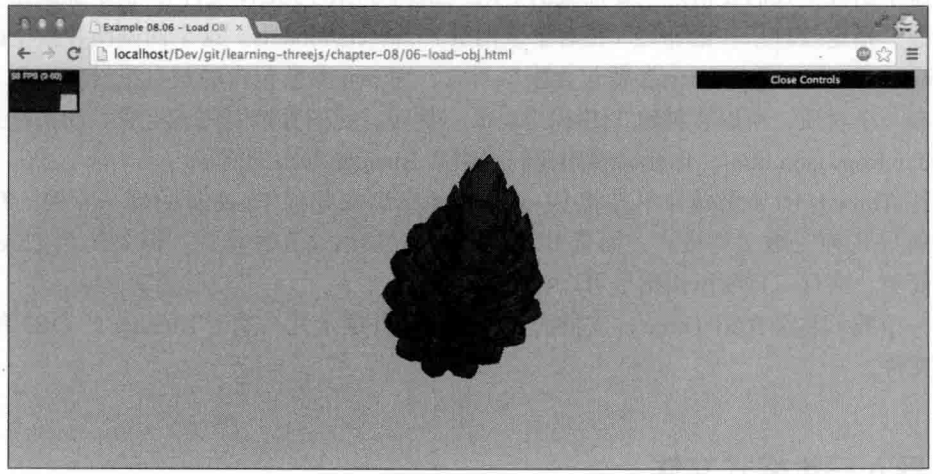
OBJ 和 MTL 是相互配合的两种格式，经常一起使用。OBJ 文件定义几何体的格式，而 MTL 文件定义所用的材质。OBJ 和 MTL 都是文本格式的。某个 OBJ 文件中的部分内容如下所示：

```
v -0.032442 0.010796 0.025935
v -0.028519 0.013697 0.026201
v -0.029086 0.014533 0.021409
usemtl Material
s 1
f 2731 2735 2736 2732
f 2732 2736 3043 3044
```

MTL 文件则用类似下面的代码定义材质：

```
newmtl Material
Ns 56.862745
Ka 0.000000 0.000000 0.000000
Kd 0.360725 0.227524 0.127497
Ks 0.010000 0.010000 0.010000
Ni 1.000000
d 1.000000
illum 2
```

Three.js 可以很好地理解 OBJ 文件和 MTL 文件，而且 Blender 也支持这两种文件格式。所以作为替代方案，你也可以在 Blender 里选择导出 OBJ/MTL 格式的文件，而不是 Three.js JSON 格式文件。在 Three.js 里提供了两种不同的加载器。如果你只想加载几何体，你可以用 OBJLoader。我们在示例 06-load-obj.html 里使用的就是这个加载器，其结果如下图所示：



要在 Three.js 中导入上述几何体，你必须包含 ObjLoader.js 文件：

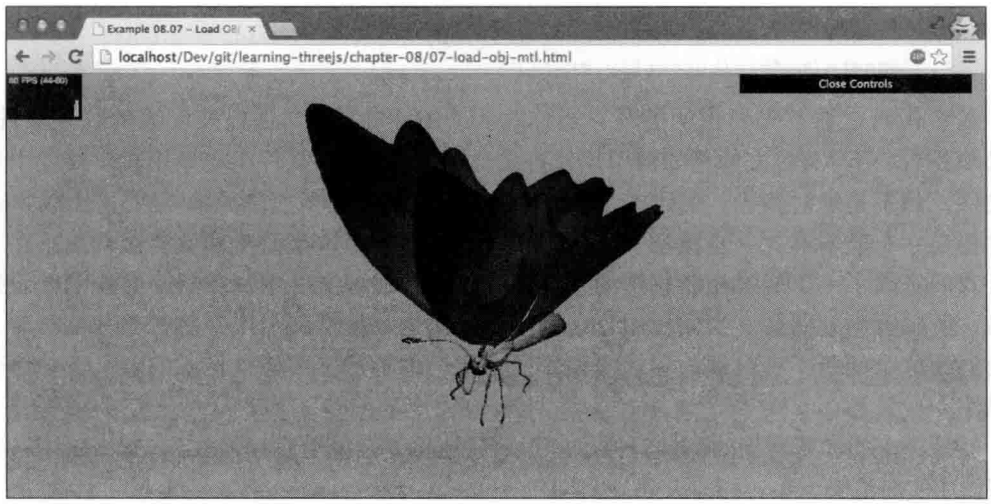
```
<script type="text/javascript"
  src="../libs/OBJLoader.js"></script>
```

然后用下面的代码导入模型：

```
var loader = new THREE.OBJLoader();
loader.load('../assets/models/pinecone.obj', function(geometry) {
  var material = new THREE.MeshLambertMaterial({
    color: 0x5C3A21
  });

  // geometry is a group of children.
  // If a child has one additional child it's probably a mesh
  geometry.children.forEach(function(child) {
    if (child.children.length == 1) {
      if (child.children[0] instanceof THREE.Mesh) {
        child.children[0].material = material;
      }
    }
  });

  geometry.scale.set(100, 100, 100);
  geometry.rotation.x = -0.3;
  scene.add(geometry);
});
```

在这段代码里我们利用 OBJLoader 类从一个 URL 加载模型。模型加载成功后，我们提供的回调函数就会被调用。然后我们就把这个模型添加到场景中。一般来讲在回调函数里最好先输出一段应答信息，以显示导入了多少几何体。大多数情况下这些加载器会返回一个几何体组合的层次化结构。理解了这个之后，正确地放置和应用材质就会简单很多。同时还要检查一下几个顶点的位置，以决定是否要缩放几何体，以及在何处放置相机。

下一个例子（07-load-obj-mtl.html）使用的是 OBJMTLLoader，加载模型并直接赋予材质。



首先我们要在网页中添加正确的加载器：

```
<script type="text/javascript"
  src="../libs/OBJLoader.js"></script>
<script type="text/javascript"
  src="../libs/MTLLoader.js"></script>
<script type="text/javascript"
  src="../libs/OBJMTLLoader.js"></script>
```

然后我们就可以像下面这样从 OBJ 文件和 MTL 文件中加载模型：

```
var loader = new THREE.OBJMTLLoader();
loader.addEventListener( 'load', function ( event ) {

  var object = event.content;
  var wing2 = object.children[5].children[0];
  var wing1 = object.children[4].children[0];

  wing1.material.alphaTest = 0.5;
  wing1.material.opacity = 0.6;
  wing1.material.transparent = true;

  wing2.material.alphaTest = 0.5;
  wing2.material.opacity = 0.6;
  wing2.material.transparent = true;

  object.scale.set(140, 140, 140);
  object.rotation.x = 0.2;
  object.rotation.y = -1.3;

  scene.add(object);
});


loader.load('../assets/models/butterfly.obj',
  '../assets/models/butterfly.mtl');
```

在看代码之前首先要说明的是：当你收到 OBJ 和 MTL 文件以及所需的纹理文件时，你必须要看一下 MTL 文件是如何引用纹理文件的。在 MTL 文件中应该用相对路径引用纹理文件，而不是绝对路径。代码本身没什么特殊的。使用这个加载器时需要为加载（load）事件指定一个监听器。一旦模型、材质和纹理加载完毕，这个监听器就会被调用。在本例中，我们修改了一下蝴蝶翅膀材质的某些属性。由于在源文件中蝴蝶翅膀的透明度设置得不对，所以看不到翅膀。当你使用复杂的模型，而这个模型又引用某些材质和纹理时，你就会遇到这个问题。有时候为了让模型恢复正常，你不得不去检查材质的定义，并修改一些属性。

但是，正如你所看到的那样，你可以很容易地将复杂模型直接加载到 Three.js 中，并在浏览器中实时渲染。

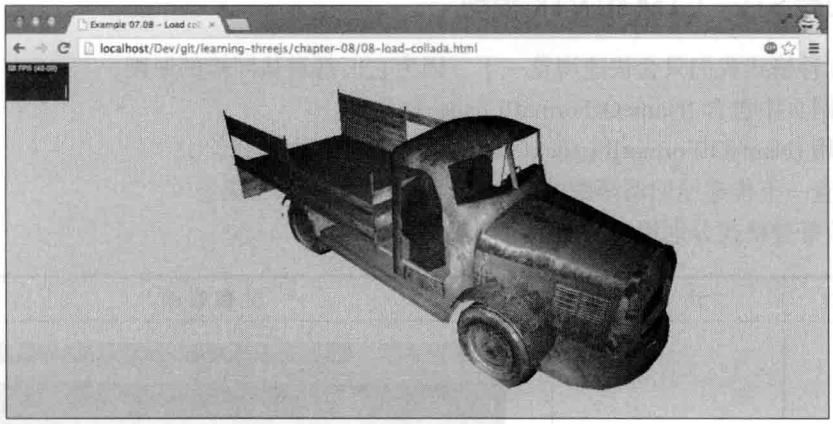
8.5.2　加载 Collada 模型

Collada 模型（文件扩展名是 .dae）是另外一种非常通用的、定义场景和模型（以及动画，我们将在下一章讲述）的文件格式。Collada 模型中不仅定义了几何体，也定义了材质。甚至还可以定义光源。

加载 Collada 模型所需的步骤跟加载 OBJ 和 MTL 模型的步骤基本一样。首先要包含正确的加载器：

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

在本例中我们加载如下的模型：



加载 Collada 模型的代码同样也很简单：

```
var mesh;
loader.load("../assets/models/dae/Truck_dae.dae",
  function (result) {
    mesh = result.scene.children[0].children[0].clone();
    mesh.scale.set(4, 4, 4);
    scene.add(mesh);
});
```

主要区别是 result（结果）对象会传递给回调函数。这个 result 对象的结构如下所示：

```
var result = {

  scene: scene,
  morphs: morphs,
  skins: skins,
  animations: animData,
  dae: {
     ...
  }
};
```

在本章我们感兴趣的是这个对象的 scene 参数。首先把场景（即 result.scene.children[0].children[0]）打印到控制台，以锁定我们感兴趣的网格位置。剩下要做的就只是将它缩放到合适的尺寸并添加到场景中。关于这个例子最后要注意的是：第一次加载这个模型的时候，材质不能正确渲染。因为材质上的纹理是 TGA 格式的，而 WebGL 并不支持该格式。为了纠正这个问题，我不得不把 TGA 文件转换成 PNG 文件，并修改 .dae 模型文件中的 XML 元素，指向转换后的 PNG 文件。

正如你所看到的，对于大多数含有材质的复杂模型来说，为了得到想要的结果，你不得不采取额外的一些步骤。如果仔细看一下（使用 console.log()）材质是如何配置的，或者用一些测试性的材质来替换，通常很容易找到问题所在。

8.5.3 加载 STL、CTM 和 VTK 模型

下面几种格式我们只会快速浏览一下，因为它们都遵从同样的原则：

1）在网页中包含 [NameOfFormat]Loader.js 文件。

2）使用 [NameOfFormat]Loader.load() 函数从 URL 中加载。

3）检查一下传递给回调函数的返回结果，并对它进行渲染。

我们给每种格式分别提供了一个示例：

格　式	示　例	屏幕截图
STL	09-load-STL.html	
CTM	10-load-CTM.html	

（续）

格　式	示　例	屏幕截图
VTK	11-load-VTK.html	

我们已经展示了几乎所有的格式。在接下来的两节里，我们要看一些不一样的加载模型的方法。首先我们会看一看如何从蛋白质数据银行（PDB 格式）中加载并渲染蛋白质，最后我们会使用在 PLY 格式文件里定义的模型来创建一个粒子系统。

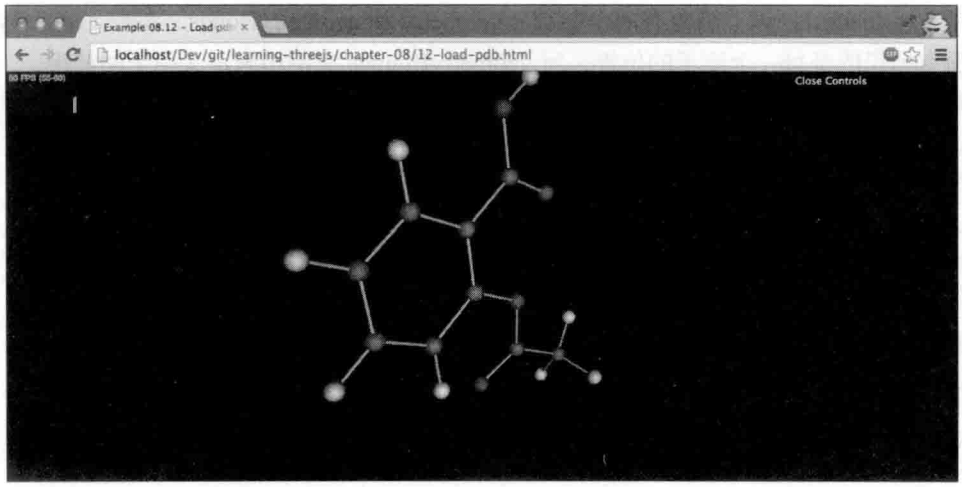
8.5.4　展示蛋白质数据银行中的蛋白质

蛋白质数据银行（www.rcsb.org）收集了很多分子、蛋白质的详细信息。除了提供蛋白质的信息，还可以以 PDB 格式下载这些分子的结构数据。在 Three.js 中有一种加载器可以加载 PDB 格式的文件。本节，我们将向你展示如何解析 PDB 文件，并在 Three.js 中显示出来。

照例，加载一种新的文件格式我们首先要做的是包含正确的加载器：

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

包含了这个加载器之后，我们就可以创建如下描述分子结构的三维模型：



加载 PDB 文件的方法，跟加载我们之前所看到的其他格式文件的方法一样：

```
var loader = new THREE.PDBLoader();
var group = new THREE.Object3D();
loader.load("../assets/models/diamond.pdb", function (geometry,
  geometryBonds) {
    var i = 0;

    geometry.vertices.forEach(function (position) {
      var sphere = new THREE.SphereGeometry(0.2);
      var material = new THREE.MeshPhongMaterial(
        {  color: geometry.colors[i++]  });
      var mesh = new THREE.Mesh(sphere, material);
      mesh.position = position;
      group.add(mesh);
    });

    for (var j = 0; j < geometryBonds.vertices.length; j += 2) {
      var path = new THREE.SplineCurve3(
        [geometryBonds.vertices[j],
        geometryBonds.vertices[j + 1]]);
      var tube = new THREE.TubeGeometry(path, 1, 0.04)
      var material = new THREE.MeshPhongMaterial(
        {  color: 0xcccccc  });
      var mesh = new THREE.Mesh(tube, material);
      group.add(mesh);
    }
    console.log(geometry);
    console.log(geometryBonds);

    scene.add(group);
});
```

正如你在这个例子里所看到的，我们实例化了一个 PDBLoader 对象，并将想要加载的模型文件传递给它，我们还提供了一个回调函数，该函数会在模型加载完之后调用。在这个加载器中，callback（回调）函数有两个参数：geometry 和 geometryBonds。参数 geometry 里的各个顶点标识的是每一个原子的位置，而 geometryBounds 实例用来定义原子间的键。

对于每个顶点，我们会创建一个带有颜色的球体。而这些颜色也是由模型文件提供的：

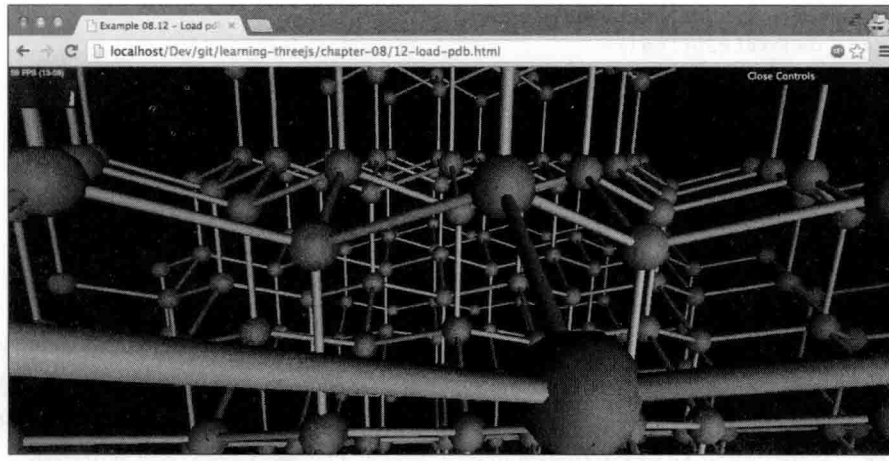
```
var sphere = new THREE.SphereGeometry(0.2);
var material = new THREE.MeshPhongMaterial({color:
  geometry.colors[i++]});
var mesh = new THREE.Mesh(sphere, material);
mesh.position = position;
group.add(mesh)
```

原子间的键用如下代码定义：

```
var path = new THREE.SplineCurve3(
  [geometryBonds.vertices[j], geometryBonds.vertices[j + 1]]);
var tube = new THREE.TubeGeometry(path, 1, 0.04)
var material = new THREE.MeshPhongMaterial({color: 0xcccccc});
```

```
var mesh = new THREE.Mesh(tube, material);
group.add(mesh);
```

对于连接，我们先用 THREE.SplineCurve3 对象创建一个三维路径。该路径用作 THREE.Tube 对象的输入，创建原子间的键。所有键和原子都会被添加到一个组中，然后将这个组添加到场景中。在蛋白质数据银行有很多模型可以下载。下图所展示的就是钻石的分子结构：



8.5.5　从 PLY 模型中创建粒子系统

PLY 格式的使用跟其他格式并没有很大的不同。你只要包含加载器、提供一个回调函数、渲染模型即可。但是，在最后这个示例中，我们要做一些不一样的事情。我们将会使用模型中的信息来创建一个粒子系统，而不是将模型渲染成一个网格。

渲染出上述截图的 JavaScript 代码实际上很简单：

```
var loader = new THREE.PLYLoader();
var group = new THREE.Object3D();
loader.load("../assets/models/test.ply", function (geometry) {
  var material = new THREE.ParticleBasicMaterial({
    color: 0xffffff,
    size: 0.4,
    opacity: 0.6,
    transparent: true,
    blending: THREE.AdditiveBlending,
    map: generateSprite()
  });

  group = new THREE.ParticleSystem(geometry, material);
  group.sortParticles = true;

  console.log(group);
  scene.add(group);
});
```

正如你所看到的，我们使用 PLYLoader 对象加载模型。回调函数接受 geometry 作为参数，然后我们用这个 geometry 作为 ParticleSystem 对象的输入。我们所用的材质，跟上一章最后那个例子所用的材质一样。正如你所看到的，使用 Three.js 可以很容易地加载各种源文件中的模型，并用不同的方法将它们渲染出来，而这些只需要几行代码就可以完成。

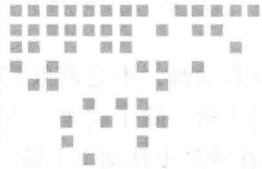
8.6 总结

在 Three.js 中使用外部源文件中的模型并不难。特别是那些简单的模型，你只要区区几步就可以完成。在使用外部的模型，或者对模型进行组合和合并时，最好记住下面的事情：

- 组合对象的时候，每个对象依然可以单独进行操作。对父对象进行操作可以影响子对象。
- 如果将对象合并在一起，你就失去了对每个对象的控制，你所得到的是一个新几何体。当你要渲染成千上万的几何体，而性能成为瓶颈的时候，这么做特别有用。
- 请记住，Three.js 的 GeometryExporter 还在开发中，SceneExporter 类和 SceneLoader 类也一样。
- 使用 Three.js 提供的各种格式的加载器时，在回调函数中输出其接收到的信息。这将帮你理解正确获取模型网格所需的步骤，并设置正确的位置和缩放比例。
- 如果模型不能正确显示，一般是材质导致的。可能是用了不兼容的纹理格式，透明度不正确，或者是该格式文件中指向纹理的连接有误。可以用测试性的材质来检测这种错误，也可以在 JavaScript 控制台中输出材质信息，看看有没有比较奇怪的值。

- 使用 Blender 时，你可以用多种方法导出模型。你可以使用 Three.js 的插件。Blender 可以很好地支持和理解 OBJ 和 MTL 格式。

本章以及前面章节所用的模型大都是静态模型。没有动画，不会四处移动，也不会变形。下一章你将学习如何让你的模型动起来，赋予它们生命。除了动画，下一章还会讲解 Three.js 中各种控制相机的方法。通过控制相机，你可以在场景中移动、平移、旋转相机。



Chapter 9 第 9 章

创建动画和移动相机

到本章为止，我们已经看到过一些简单的动画，它们都不太复杂。我们在第 1 章就引入了一个基础的渲染循环，在随后的各章里我们使用这个循环简单地旋转对象，还展示了其他几个基础的动画概念。本章我们将进一步介绍 Three.js 是如何支持动画操作的。我们将会详细讨论下面这四个主题：

- **基础动画**：Three.js 里所有动画的基础，修改对象的三个属性：位置、旋转和缩放。在这一部分，我们将快速回顾一下在第 1 章引入的那个循环里如何处理这些操作。
- **移动相机**：动画中一个很重要的部分是在场景中移动相机的能力。本节我们将带领你领略 Three.js 中各种控制相机的方法。
- **变形和蒙皮**：制作复杂模型的动画主要有两种方式。使用变形技术，定义几何体之间的过渡；以及使用骨骼和蒙皮技术处理这种过渡。我们将会在本章探讨这两种方式。
- **加载外部动画**：在前面的章节里，我们已经见过 Three.js 可以支持几种外部的文件格式。本章我们将会对此做进一步的探讨，看看如何从外部文件中加载动画。

我们将从动画背后的基础概念开始。

9.1 基础动画

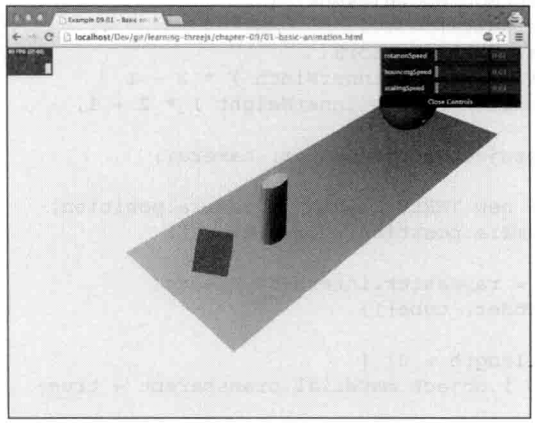
在看例子之前，我们先快速回顾一下第 1 章中的 render（渲染）循环。为了支持动画，我们需要告诉 Three.js 多久渲染一次场景。为此，我们要使用由 HTML5 提供的标准的 requestAnimationFrame 函数：

```
render();

function render() {

  // render the scene
  renderer.render(scene, camera);

  // schedule the next rendering using requestAnimationFrame
  requestAnimationFrame(render);
}
```

在这段代码里，我们只要在初始化场景时调用一次 render() 函数。在 render() 函数中，我们用 requestAnimationFrame 来安排下一次渲染。这样，浏览器就可以保证以正确的时间间隔（通常是 60 次每秒）调用 render() 函数。在 requestAnimationFrame 函数添加到浏览器之前，一般会使用 setInterval（function，interval）或者 setTimeout（function，interval）。它们会按照设置的时间间隔调用指定的函数。问题是它们并没有考虑其他正在发生的事情。即使动画没有显示或者隐藏在某个标签页下，这两个函数依然会被调用，而且会耗费某些资源。这些函数还有另外一个问题，那就是它们一旦被调用就会刷新屏幕，不管这时对浏览器来说是不是恰当的时机。这也就意味着较高的 CPU 使用率。通过 requestAnimationFrame，我们不必告诉浏览器什么时候需要刷新屏幕，而是请求浏览器在最适合的时候执行我们提供的函数。通常情况下其结果是 60fps（帧频）。使用 requestAnimationFrame 会让你的动画运行得更平缓，而且对 CPU 和 GPU 更友好，你也不必再担心渲染时机方面的问题。

9.1.1　简单动画

使用这种方法，我们通过改变对象的旋转、缩放、位置、材质、顶点、面，以及其他你想得到的东西，可以很容易地制作出动画。在下一次的 render 循环中，Three.js 就可以渲染这些修改后的属性。你可以在 01-basic-animation.html 中找到一个非常简单的例子，该示例基于第 1 章中的那个例子，如下图所示：



该示例的 render 循环非常简单。你只要修改网格的属性即可，Three.js 会处理剩下的事情：

```
function render() {
  cube.rotation.x += controls.rotationSpeed;
  cube.rotation.y += controls.rotationSpeed;
  cube.rotation.z += controls.rotationSpeed;

  step += controls.bouncingSpeed;
  sphere.position.x = 20 + ( 10 * (Math.cos(step)));
  sphere.position.y = 2 + ( 10 * Math.abs(Math.sin(step)));

  scalingStep += controls.scalingSpeed;
  var scaleX = Math.abs(Math.sin(scalingStep / 4));
  var scaleY = Math.abs(Math.cos(scalingStep / 5));
  var scaleZ = Math.abs(Math.sin(scalingStep / 7));
  cylinder.scale.set(scaleX, scaleY, scaleZ);

  renderer.render(scene, camera);
  requestAnimationFrame(render);
}
```

这段代码没有什么特别的，只是很好地展示了本书中所讨论的各种动画背后的基础概念。下一节，我们将会快速浏览一下跟动画有关的内容。当你使用 Three.js 处理复杂场景时，除了动画，一个很重要的方面是使用鼠标在屏幕上选择对象的能力。

9.1.2 选择对象

我们在本章探讨的是相机和动画，尽管跟动画没有直接的联系，但是在这里讨论一下对象的选择也是对本章主题的有益补充。我们在这里所展示的是如何使用鼠标在场景中选择一个对象。在看示例之前，我们先来看一下代码：

```
var projector = new THREE.Projector();

function onDocumentMouseDown(event) {
  event.preventDefault();
  var vector = new THREE.Vector3(
    (event.clientX / window.innerWidth ) * 2 - 1,
    -( event.clientY / window.innerHeight ) * 2 + 1,
      0.5);
    projector.unprojectVector(vector, camera);

  var raycaster = new THREE.Raycaster(camera.position,
    vector.sub(camera.position).normalize());

  var intersects = raycaster.intersectObjects(
    [sphere, cylinder, cube]);

  if (intersects.length > 0) {
    intersects[ 0 ].object.material.transparent = true;
```

```
        intersects[ 0 ].object.material.opacity = 0.1;
    }
}
```

在这段代码里，我们使用 THREE.Projector 类和 THREE.Raycaster 类，来检测我们是否点在了一个对象上。当我们在屏幕上点击时，会发生如下的事情：

1）首先，基于我们在屏幕上点击的位置创建出一个向量。

2）接着，用 unprojectVector 函数，将屏幕上的点击位置转换成 Three.js 场景中的坐标。

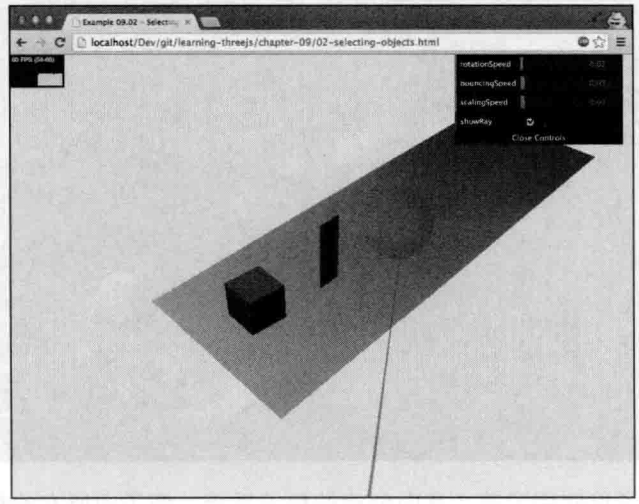
3）然后，我们用 THREE.Raycaster 对象（projector.pickingRay 函数的返回值）从屏幕上的点击位置向场景中发射一束光线。

4）最后，我们用 raycaster.intersectObjects 函数来判断指定的对象中有没有被这束光线击中的。

上述步骤中最后一步的结果包含了所有被光线击中的对象的信息。这些信息包括：

```
distance: 49.9047088522448
face: THREE.Face4
faceIndex: 4
object: THREE.Mesh
point: THREE.Vector3
```

其中 object 属性就是我们点击的网格，face 和 faceIndex 指的是该网格中被选中的面。distance 属性是从相机到被点物体间的距离，而 point 则是被选中物体网格上的点。你可以在示例 02-selecting-objects.html 中测试该功能。任何你所选中的对象都会变得透明，相关的信息会输出到控制台。如果你想看看光线的投射路径，你可以在菜单上打开 showRay 属性，如下图所示：



现在我们已经在渲染循环中修改了各种属性。下一节，我们将会看一个小型库，使用它

可以让动画的定义简单很多。

9.1.3 用 Tween.js 做动画

Tween.js 是一个小型的 JavaScript 库，可以从 https://github.com/sole/tween.js/ 下载。这个库可以用来定义某个属性在两个值之间的过渡，自动计算出起始值和结束值之间的所有中间值。这个过程叫做 tweening（补间）。例如，你可以用这个库将一个网格 x 轴上的位置，在 10 秒钟内从 10 递减到 3：

```
var tween = new TWEEN.Tween({x: 10})
  .to({x: 3}, 10000)
  .easing(TWEEN.Easing.Elastic.InOut)
  .onUpdate( function () {
    // update the mesh
  })
```

在这个例子里我们创建了一个 TWEEN.Tween 对象。这个对象可以保证 x 属性在 10000 毫秒内，从 10 变化到 3。通过 Tween.js 你还可以指定在指定时间内属性如何变化，是线性的、指数性的，还是其他任何可能的方式（完整列表可以参照 http://sole.github.io/tween.js/examples/03_graphs.html）。属性值在指定时间内的变化称为 easing（缓动）。在 Tween.js 中，你可以用 easing() 函数来配置缓动效果。

在 Three.js 中使用这个库非常简单。打开示例 03-animation-tween.html，你就可以看到 Tween.js 库的效果，如下图所示：

我们在这个例子里使用了第 7 章中创建的粒子系统，例子中所有的粒子正在落向地面。而这些粒子的位置是由 Tween.js 库创建出的中间值：

```
// first create the tweens
var posSrc = {pos: 1}
var tween = new TWEEN.Tween(posSrc).to({pos: 0}, 5000);
tween.easing(TWEEN.Easing.Sinusoidal.InOut);

var tweenBack = new TWEEN.Tween(posSrc).to({pos: 1}, 5000);
tweenBack.easing(TWEEN.Easing.Sinusoidal.InOut);

tween.chain(tweenBack);
tweenBack.chain(tween);

var onUpdate = function () {
  var count = 0;
  var pos = this.pos;

  loadedGeometry.vertices.forEach(function (e) {
    var newY = ((e.y + 3.22544) * pos) - 3.22544;
    particleSystem.geometry.vertices[count++].set(e.x,
      newY, e.z);
  });

  particleSystem.sortParticles = true;
};

tween.onUpdate(onUpdate);
tweenBack.onUpdate(onUpdate);
```

在这段代码里我们创建了两个补间：tween 和 tweenBack。第一个补间定义的是属性 position 如何从 1 过渡到 0，第二个正好相反。通过 chain() 函数，我们可以让这两个补间首尾相连。这样在启动动画之后，程序就会在这两个补间间循环。最后我们定义了一个 onUpdate 方法。在这个方法里我们会遍历粒子系统的所有顶点，并用补间（this.pos）提供的位置更新顶点的位置。

补间动画在模型加载完毕时启动，所以我们在下面函数的末尾调用 tween.start() 函数：

```
var loader = new THREE.PLYLoader();
loader.load( "../assets/models/test.ply", function (geometry) {
  ...
  tween.start()
  ...
});
```

开启补间之后，我们需要告知 Three.js 库什么时候应该刷新已知的所有补间。为此可以调用 TWEEN.update() 函数：

```
function render() {
  TWEEN.update();
  webGLRenderer.render(scene, camera);
  requestAnimationFrame(render);
}
```

这些都做好之后，tween 库就会负责计算粒子系统中每个粒子的位置。正如你所看到的，使用这个库比自己管理属性值之间的过渡要简单得多。

除了让对象动起来，或者改变其外观，我们还可以通过移动相机，让整个场景动起来。在前面几章里，我们已经这么做了，当时我们是手工更新相机的位置。Three.js 还提供了其他几种方法来更新相机。

9.2 使用相机

Three.js 提供了几个相机控件，可以用来控制场景中的相机。这些控件在 Three.js 发布包中，你可以在 examples/js/controls 目录下找到它们。在本节中我们将会看到如下的控件：

控件名称	描　述
FirstPersonControls（第一人称控件）	该控件的行为类似第一人称射击游戏中的相机，用键盘移动，用鼠标转动
FlyControls（飞行控件）	飞行模拟器控件，用键盘和鼠标来控制相机的移动和转动
RollControls（翻滚控件）	该控件是 FlyControl 的简化版，让你可以绕着 z 轴旋转
TrackballControls（轨迹球控件）	最常用的控件，你可以用鼠标（或轨迹球）来轻松地移动、平移和缩放场景
OrbitControls（轨道控件）	用于特定的场景，模拟轨道中的卫星，你可以用鼠标和键盘在场景中游走
PathControls（路径控件）	使用这个控件，相机可以沿着预定义的路径移动。你可以将它跟过山车相比较，在过山车上你可以朝四周看，但不能改变自身的位置

当然，除了使用这些相机控件，你还可以通过设置相机的 position 属性，调用 lookAt() 函数改变相机指向的位置，来移动相机。

我们首先要看的控件是 TrackballControls（轨迹球控件）。

9.2.1 轨迹球控件

使用 TrackballControls 之前，你首先要在网页中包含正确的 JavaScript 文件：

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

包含了这个文件之后，我们就可以创建控件，并将它绑定到相机上：

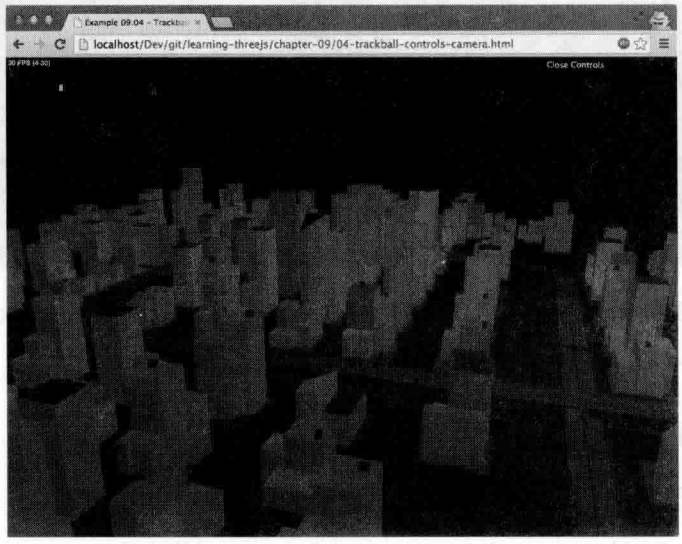
```
var trackballControls = new THREE.TrackballControls(camera);
trackballControls.rotateSpeed = 1.0;
trackballControls.zoomSpeed = 1.0;
trackballControls.panSpeed = 1.0;
```

更新相机的位置可以在 render 循环中完成：

```
var clock = new THREE.Clock();
function render() {
  var delta = clock.getDelta();
  trackballControls.update(delta);
  requestAnimationFrame(render);
  webGLRenderer.render(scene, camera);
}
```

在前面这段代码里，我们会看到一个新的 Three.js 对象：THREE.Clock。THREE.Clock 可以用来精确地计算出上次调用后经过的时间，或者一个渲染循环耗费的时间。你只要调用 clock.getDelta() 函数即可。这个函数会返回此次调用和上次调用之间经过的时间。要更新相机的位置，我们可以调用 trackballControls.update() 函数。在这个函数里我们需要提供自上次 update() 函数调用以来经过的时间。为此我们要使用 THREE.Clock 对象的 getDelta() 函数。你或许会奇怪为什么我们不直接把帧频（1/60 秒）传递给 update 函数。原因是当我们请求动画帧时，我们期待的帧频是 60fps，但这并不能得到保证。受外部因素的影响，帧频可能会有不同。为了保证相机能够平缓地移动和旋转，我们需要传入精确的时间差。

实际的例子可以参看 04-trackball-controls-camera.html。如下图所示：



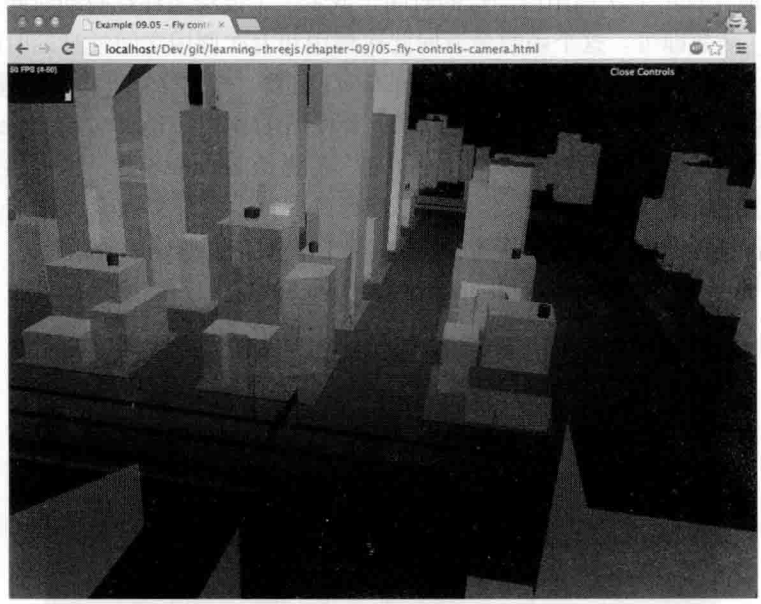
你可以用下面的方式来操控相机：

操　控	动　作
按住左键，拖动	在场景中旋转、翻滚相机
转动滚轮	放大和缩小
按住中键，拖动	放大和缩小
按住右键，拖动	在场景中平移

有几个属性可以用来对相机进行微调。例如，你可以通过 rotateSpeed 属性来控制相机的旋转速度，将 noZoom 设为 true 可以禁止缩放。我们不会讨论每个属性的细节，因为它们完全可以自我解释。要想全面了解这些属性，可以参照这些属性所在的 TrackballControls.js 文件。

9.2.2 飞行控件

接下来我们要看的控件是 FlyControls。通过 FlyControls 你可以使用类似飞行模拟器操纵杆的控件，在场景中飞行。你可以参看示例 05-fly-controls-camera.html，如下图所示：



FlyControls 的使用跟 TrackballControls 一样，首先要加载正确的 JavaScript 文件：

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

然后我们可以配置控件，并将它绑定到相机上：

```
var flyControls = new THREE.FlyControls(camera);
flyControls.movementSpeed = 25;
flyControls.domElement = document.querySelector('#WebGL-output');
flyControls.rollSpeed = Math.PI / 24;
flyControls.autoForward = true;
flyControls.dragToLook = false;
```

同样，我们不会解释所有细节。你可以参看 FlyControls.js 文件。我们只挑几个需要配置的、保证该控件能够正常工作的属性来讲。需要正确设置的属性是 domElement。该属性应该指向场景所在的元素。在本书的示例中，我们使用下面这个元素作为输出： ，所以该属性应当使用如下设置：flyControls.domElement = document.query-

Selector（'#WebGL-output'）;。如果我们没有正确地设置该属性，那么移动鼠标时会导致奇怪的行为。

使用该控件可以用如下的方式操控相机：

操　控	动　作	操　控	动　作
按住左键和中键	往前移动	D	右移
按住右键	往后移动	R	上移
移动鼠标	往四周看	F	下移
W	往前移动	上、下、左、右方向键	向上、下、左、右看
S	往后移动	G	向左翻滚
A	左移	E	向右翻滚

下一个我们要看的控件是 THREE.RollContorls（翻滚控件）。

9.2.3　翻滚控件

RollControls 的行为跟 FlyControls 基本一致，所以我们在这里不会解释太多细节。可以用如下代码创建 RollControls：

```
var rollControls = new THREE.RollControls(camera);
rollControls.movementSpeed = 25;
rollControls.lookSpeed = 3;
```

如果你想试用该控件，可以打开示例 06-roll-controls-camera.html。注意，如果你只能看到黑色的屏幕，请把鼠标移动到浏览器的底部。一幅城市景观即可出现在视图中。可以用如下方式来控制相机：

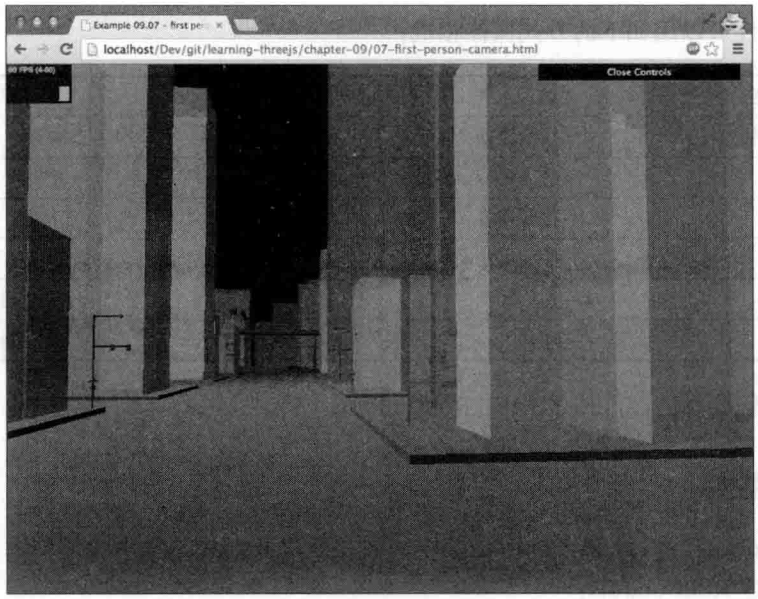
操　控	动　作	操　控	动　作
按住左键	往前移动	D	右移
按住右键	往后移动	Q	往左翻滚
上、下、左、右方向键	前、后、左、右移动	E	往右翻滚
W	前移	R	上移
A	左移	F	下移
S	后移		

最后一个我们要看的基础控件是 FirstPersonControls（第一人称控件）。

9.2.4　第一人称控件

顾名思义，通过 FristPersonControls 你可以像第一人称射击游戏中的枪手一样控制相机。用鼠标控制视角，用键盘来移动角色。你可以在 07-first-person-camera.html 中找到一个示例。

参照如下截图：



该控件的创建跟我们之前看到的控件一样。在前面展示的这个示例中，我们使用如下的配置：

```
var camControls = new THREE.FirstPersonControls(camera);
camControls.lookSpeed = 0.4;
camControls.movementSpeed = 20;
camControls.noFly = true;
camControls.lookVertical = true;
camControls.constrainVertical = true;
camControls.verticalMin = 1.0;
camControls.verticalMax = 2.0;
camControls.lon = -150;
camControls.lat = 120;
```

使用该控件时，只有最后两个属性（lon 和 lat）需要小心对待。这两个属性定义的是场景初次渲染时相机指向的位置。

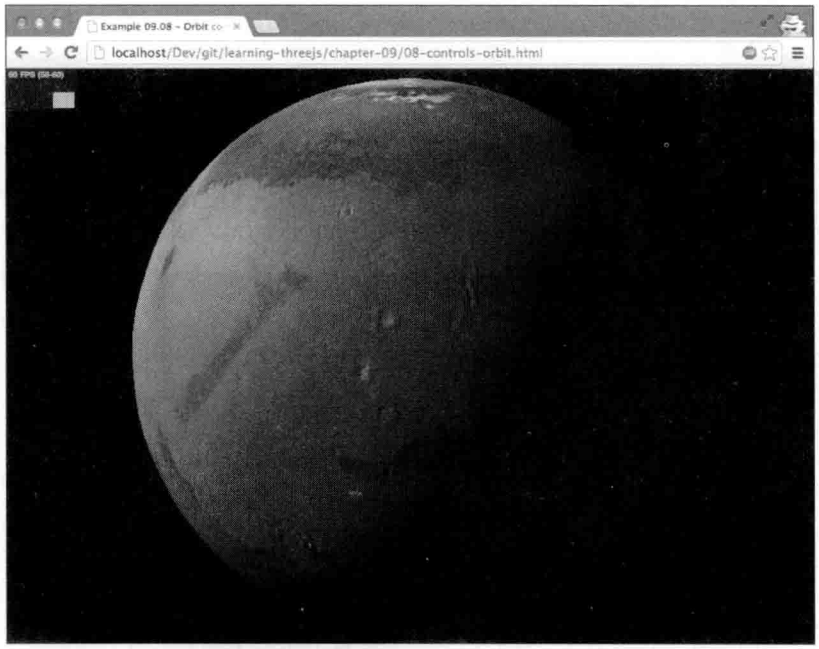
该控件的控制方法非常直白：

操　控	动　作	操　控	动　作
移动鼠标	往四周看	D	右移
上、下、左、右方向键	前、后、左、右移动	R	上移
W	前移	F	下移
A	左移	Q	停止
S	后移		

下一个控件，我们将从第一人称视角切换到透视视角。

9.2.5　轨道控件

OrbitControl（轨道控件）是在场景中绕某个对象旋转、平移的好方法。我们引入了示例 08-controls-orbit.html，可以展示该控件是如何工作的。参考下面的屏幕截图：



该控件使用起来跟其他控件一样简单。包含正确的 JavaScript 文件，将控件绑定到相机，然后使用 THREE.Clock 对象来更新控件：

```
<script type="text/javascript"
  src="../libs/OrbitControls.js"></script>
...
var orbitControls = new THREE.OrbitControls(camera);
orbitControls.autoRotate = true;
var clock = new THREE.Clock();
...
var delta = clock.getDelta();
orbitControls.update(delta);
```

操控轨道控件主要使用鼠标：

操　控	动　作
按住左键，并移动	绕场景中心旋转相机
转动滚轮或按住中键，并移动	放大缩小

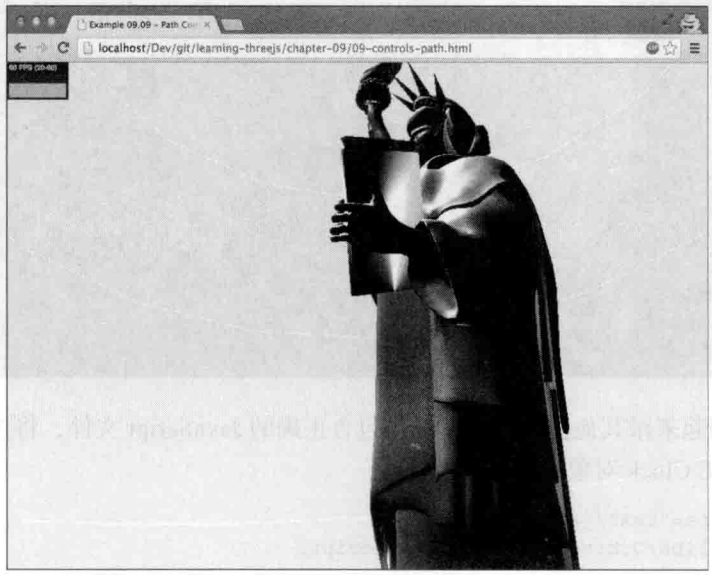
（续）

操　控	动　作
按住右键，并移动	在场景中移动
上、下、左、右方向键	在场景中移动

我们最后要看的控件是 PathControl（路径控件）。使用该控件可以让相机沿着固定路径移动。

9.2.6 路径控件

PathControl 是相当酷的控件。通过这个控件你可以设定一个路径，然后相机会沿着该路径移动，用户则可以控制相机向四周看。作为示例，我们加载了一个自由女神模型，并使用 PathControl 缓慢地螺旋式向上爬升。参看 09-path-controls.html，如下图所示：



使用该控件要多几个步骤才能运行起来。首先我们要创建一个路径，让相机沿着它移动：

```
function getPath() {
  var points = [];
  var r = 20;
  var cX = 0;
  var cY = 0;

  for (var i = 0; i < 1440; i += 5) {

    var x = r * Math.cos(i * (Math.PI / 180)) + cX;
    var z = r * Math.sin(i * (Math.PI / 180)) + cY;
    var y = i / 30;
```

```
    points.push(new THREE.Vector3(x, y, z));
  }
  return points;
}
```

这个函数的返回结果是一组构成螺旋线的点。该螺旋线的半径是 20，从底部开始慢慢上升。这些点可以用来构成路径，所以现在我们可以开始设置 PathControls。但是我们首先要加载正确的 JavaScript 文件。

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

加载控件之前你要保证没有手工设定相机位置，或者使用过相机的 lookAt() 函数，因为这可能会跟特定的控件相抵触。现在我们可以配置 pathControls 对象，并把它添加到场景中：

```
var pathControls = new THREE.PathControls(camera);

// configure the controller
pathControls.duration = 70
pathControls.useConstantSpeed = true;
pathControls.lookSpeed = 0.1;
pathControls.lookVertical = true;
pathControls.lookHorizontal = true;
pathControls.verticalAngleMap =
  {srcRange: [ 0, 2 * Math.PI ], dstRange: [ 1.1, 3.8 ]};
pathControls.horizontalAngleMap =
  {srcRange: [ 0, 2 * Math.PI ], dstRange: [ 0.3, Math.PI - 0.3 ]};
pathControls.lon = 300;
pathControls.lat = 40;

// add the path
controls.points.forEach(function(e) {
  pathControls.waypoints.push([e.x, e.y, e.z])});

// when done configuring init the control
pathControls.init();

// add the animationParent to the scene and start the animation
scene.add(pathControls.animationParent);
pathControls.animation.play(true, 0 );
```

相比其他控件，对于这个控件我们要多做几件事。开始部分跟我们之前已经做过的一样。我们可以设置某些特定的属性，对控件进行微调。接下来我们可以把之前定义的点添加到 waypoints 属性中。这样即可构成相机的移动路径。现在所有东西都配置好了，我们可以调用 init() 函数来结束该控件的初始化。

上述代码片段中的最后一步是必须的，以保证动画能够运转，相机可以自动移动。现在只剩下最后一步。在我们的 render 循环里，我们要添加如下代码：

```
var delta = clock.getDelta();
THREE.AnimationHandler.update(delta);
pathControls.update(delta);
```

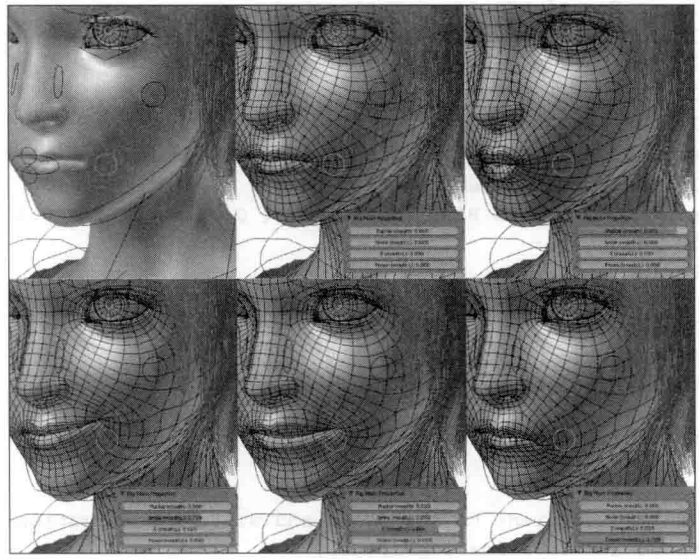
这将使得相机自动沿着轨道移动。

这就是有关相机移动的所有内容。在这一部分我们看到了很多控件，可以用来创建出有趣的相机动作。下一节，我们将会学习高级动画：变形动画和蒙皮动画[⊖]。

9.3 变形动画和骨骼动画

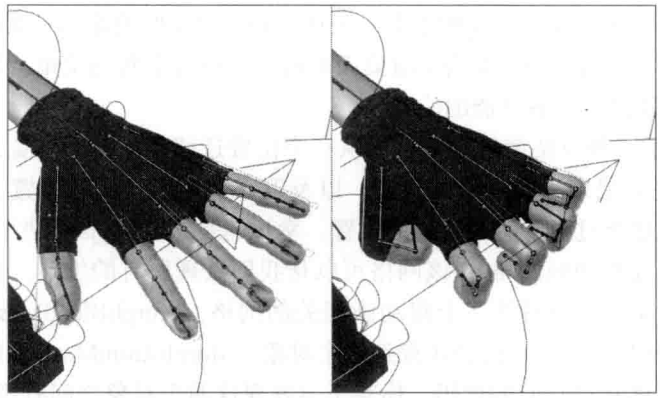
当你用外部的软件（例如 Blender）创建动画时，你通常会有两种主要的定义动画的方式：

- **变形动画**：通过变形目标，你可以定义网格经过变形之后的版本，或者说关键位置。对于这个变形目标，其所有顶点位置都会被存储下来。要让图形动起来，你所要做的只是将所有顶点从一个位置，移动到另外一个关键位置，并重复该过程。下面这个截图展示的是一组用来表示面部表情的变形目标（图片由 Blender 基金组织提供）。



- **骨骼动画**：另外一种方式就是骨骼动画。通过骨骼动画你可以定义骨骼，即网格的骨头，并把顶点绑定到特定的骨头上。现在，当你移动一块骨头时，任何相连的骨头都会做相应的移动，骨头上绑定的顶点也会随之移动。网格的变形基于骨头的位置、移动和缩放比例。下图（同样也是由 Blender 基金会提供）展示的就是一个在物体的移动和变形中使用骨头的例子。

⊖ 也叫骨骼动画。——译者注



这两种模式 Three.js 都支持，不过一般来讲使用变形目标可以得到更好的效果。骨骼动画的主要问题是如何从 Blender 等三维程序中比较好地导出数据，从而在 Three.js 中制作动画。用变形目标比用骨骼和皮肤更容易获取一个良好的工作模型。

本节会介绍这两种模型，同时还会讲解几个 Three.js 支持的可以定义动画的外部格式。

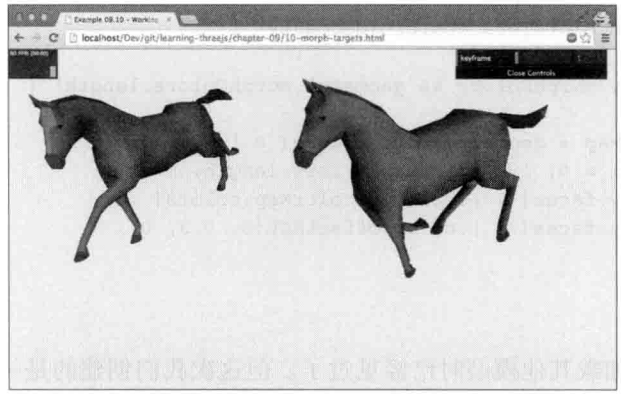
9.3.1　用变形目标制作动画

变形目标是制作动画最直接的方法。你可以为所有顶点指定每一个关键位置，然后让 Three.js 将这些顶点从一个关键位置移动到另一个。但这种方法有一个不足，那就是对于大型网格和大型动画，模型文件会变得非常大。原因是在每个关键位置上，所有顶点的位置都要重复一遍。

我们将用两个例子来向您展示如何使用变形目标。在第一个例子里我们会让 Three.js 处理各个关键帧（或者变形目标，此后我们将会如此称呼它）之间的变化；在第二个例子里，我们会手工处理这种变化。

9.3.1.1　用 MorphAnimMesh 制作动画

在我们的第一个变形动画示例中，我们将会使用 Three.js 发布包里的模型：一匹马。要理解基于变形的动画如何工作，最简单的方式就是打开示例：10-morph-targets.html。参考如下截图：



在该示例中，右侧的那匹马栩栩如生，正在奔跑，而左侧的那匹马则静静地站着。第二匹马是从基础模型（原有的顶点集合）渲染而来的。通过右上角的菜单，你可以看到所有变形目标，以及右侧那匹马所有可能的位置。

Three.js 提供了一种方法使得模型可以从一个位置迁移到另一个位置，但是这也意味着我们可能不得不手工记录当前所处的位置，以及下一个变形目标的位置。一旦到达目标位置，我们就得重复这个过程以达到下一个位置。幸运的是，Three.js 提供了一个特别的网格，MorphAnimMesh（变形动画网格），该网格可以帮我们处理所有的细节。在继续之前，我们简单提一下 Three.js 提供的另外一个跟动画相关的网格，MorphBlendMesh。如果你浏览过 Three.js 提供的所有对象，你可能会注意到这个对象。MorphAnimMesh 能做的，该网格差不多也可以做。如果你看过它们的代码，你甚至会发现这两个对象之间有很多东西是相同的。但是 MorphBlendMesh 好像已经不再推荐使用了，而且在所有 Three.js 官方提供的示例里也不再使用。所有你可以用 MorphBlendMesh 完成的事情都可以用 MorphAnimMesh 来做。因此对于这些功能应当使用 MorphAnimMesh。下面这段代码展示的就是如何加载模型，并根据这个模型创建一个 MorphAnimMesh 对象：

```
var loader = new THREE.JSONLoader();
loader.load('../assets/models/horse.js',
  function(geometry, mat) {

  var mat = new THREE.MeshLambertMaterial(
    {color: 0xffffff,
     morphNormals: false,
     morphTargets: true,
     vertexColors: THREE.FaceColors});

  morphColorsToFaceColors(geometry);
  geometry.computeMorphNormals();
  meshAnim = new THREE.MorphAnimMesh(geometry, mat );
  scene.add(meshAnim);

  },'../assets/models' );


function morphColorsToFaceColors(geometry) {

  if (geometry.morphColors && geometry.morphColors.length) {

    var colorMap = geometry.morphColors[ 0 ];
    for (var i = 0; i < colorMap.colors.length; i++) {
      geometry.faces[ i ].color = colorMap.colors[ i ];
      geometry.faces[ i ].color.offsetHSL(0, 0.3, 0);
    }
  }
}
```

该方法我们在加载其他模型时已经见过了。但这次我们创建的是一个 MorphAnimMesh

对象，而不是普通的网格。加载动画时，有几件事情需要注意：

- 保证你所使用的材质中 morphTargets 属性设为 true，否则网格不会动。
- 创建 MorphAnimMesh 对象之前，一定要对几何体调用 computeMorphNormals，以确保变形目标的所有法向量都会被计算。这对于正确地光照和阴影是必须的。
- 在某个特定的变形目标上为某些面指定颜色是可能的。通过辅助函数 morphColorsToFaceColors，我们可以保证动画过程中使用正确的颜色。
- 默认情况下，整个动画会一次放完。如果同一个几何体上定义了多个动画，那么你可以同时使用 parseAnimations() 函数和 playAnimation（name，fps）函数，播放这种动画中的一个。在本章最后一节从 MD2 模型中加载动画时，我们会使用该方法。

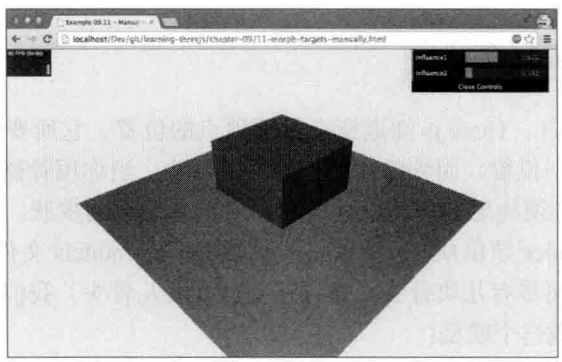
现在剩下要做的就只是在 render 循环中更新动画。为此，我们可以再次使用 THREE.Clock 对象来计算位置的变化，并用它来更新动画：

```
function render() {

  var delta = clock.getDelta();
  webGLRenderer.clear();
  if (meshAnim) {
    meshAnim.updateAnimation(delta *1000)
    meshAnim.rotation.y += 0.01;
  }

  // render using requestAnimationFrame
  requestAnimationFrame(render);
  webGLRenderer.render(scene, camera);
}
```

这种方法最简法，通过它你可以快速地从一个带有变形目标的模型中建立好动画。另外一种方法是手工建立动画。

9.3.1.2　通过设置 morphTargetInfluence 属性创建动画

我们要创建一个非常简单的示例，在这个示例里一个方块将从一种图形变换到另一种图形。但是这次我们要手工控制变形到哪一个目标。你可以打开示例 11-morph-targets-manually.html。参考下面的屏幕截图：



在这个例子里，我们基于一个简单的方块，手工创建了两个变形目标：

```
// create a cube
var cubeGeometry = new THREE.CubeGeometry(4, 4, 4);
var cubeMaterial = new THREE.MeshLambertMaterial({morphTargets:
  true, color: 0xff0000});

// define morphtargets, we'll use the vertices from these
  geometries
var cubeTarget1 = new THREE.CubeGeometry(2, 10, 2);
var cubeTarget2 = new THREE.CubeGeometry(8, 2, 8);

// define morphtargets and compute the morphnormal
cubeGeometry.morphTargets[0] = {name: 't1', vertices:
  cubeTarget2.vertices};
cubeGeometry.morphTargets[1] = {name: 't2', vertices:
  cubeTarget1.vertices};
cubeGeometry.computeMorphNormals();

var cube = new THREE.Mesh(cubeGeometry, cubeMaterial);
```

当你打开这个示例的时候，你会看到一个简单的方块。通过右上角的滑块，你可以设置 morphTargetInfluences 属性。也就是说你可以决定这个初始的方块如何变换到指定的方块 mt1，以及如何变换到方块 mt2。你可以通过网格的 morphTargetInfluences 属性来设置这个影响：

```
var controls = new function () {
  this.influence1 = 0.01;
  this.influence2 = 0.01;

  this.update = function () {
    cube.morphTargetInfluences[0] = controls.influence1;
    cube.morphTargetInfluences[1] = controls.influence2;
  };
}
```

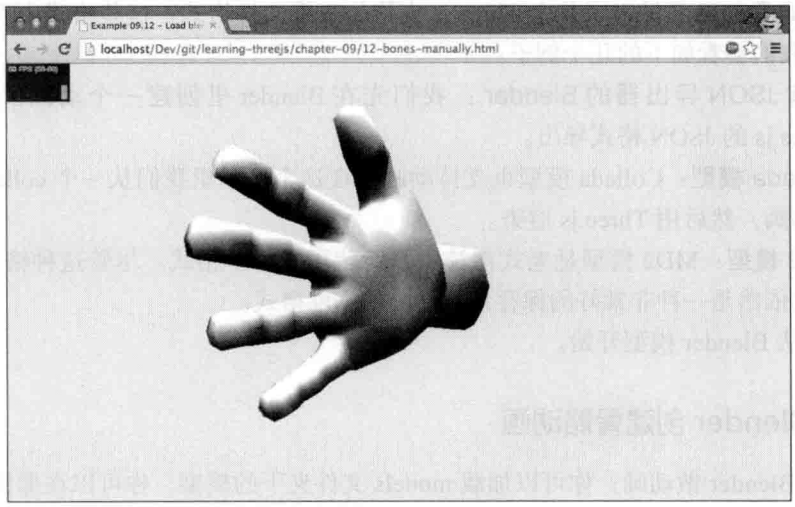
这两个例子展示了变形动画背后最重要的概念。下一节，我们会快速地看一下如何用骨骼和蒙皮来制作动画。

9.3.2 用骨骼和蒙皮制作动画

变形动画非常直白。Three.js 知道所有目标顶点的位置，它所要做的只是将每个顶点从一个位置迁移到下一个位置。而骨骼和蒙皮则要复杂些。当你用骨骼来做动画时，你移动一下骨骼，而 Three.js 必须决定如何相应地迁移附着在骨骼上的皮肤。为了展示这个概念，我们使用了一个从 Blender 导出成 Three.js 格式的模型（即 models 文件夹下的 hand-1.js）。这是一只手的模型，上面带有几块骨头。通过移动这几块儿骨头，我们就可以让整个模型动起来。先来看看如何加载这个模型：

```
var loader = new THREE.JSONLoader();
loader.load('../assets/models/hand-1.js', function (geometry, mat) {
  var mat = new THREE.MeshLambertMaterial(
    {color: 0xF0C8C9, skinning: true});

  mesh = new THREE.SkinnedMesh(geometry, mat);

  // rotate the complete hand
  mesh.rotation.x = 0.5 * Math.PI;
  mesh.rotation.z = 0.7 * Math.PI;

  // make sure to set quaternation to false for easy rotation
  mesh.bones.forEach(function (e) {
    e.useQuaternion = false;
  })

  // add the mesh
  scene.add(mesh);

  // and start the animation
  tween.start();

}, '../assets/models');
```

加载一个用于骨骼动画的模型，跟加载其他模型并没有什么不同。我们只要指定模型文件，该文件中带有骨骼的定义，然后基于几何体创建网格。Three.js 也提供了一个带有蒙皮的网格对象，THREE.SkinnedMesh。 你需要保证的是将模型所用材质的 skinning 属性设为 true。最后我们要做的是将所有骨头的 useQuaternion 属性设为 false。如果不这么做，我就不得不用四元数来定义骨头的旋转；如果设为 false，我们就可以用一般方式来设置这个旋转。在移动这些骨头之前，我们先来看一下 12-bones-manually.html。参考下面的截图：



打开这个示例你会看到一只手正在做抓握的动作。我们通过设置指部骨头绕 z 轴的旋转来达到这个效果：

```
var onUpdate = function () {
  var pos = this.pos;

  // rotate the fingers
  mesh.bones[5].rotation.set(0, 0, pos);
  mesh.bones[6].rotation.set(0, 0, pos);
  mesh.bones[10].rotation.set(0, 0, pos);
  mesh.bones[11].rotation.set(0, 0, pos);
  mesh.bones[15].rotation.set(0, 0, pos);
  mesh.bones[16].rotation.set(0, 0, pos);
  mesh.bones[20].rotation.set(0, 0, pos);
  mesh.bones[21].rotation.set(0, 0, pos);

  // rotate the wrist
  mesh.bones[1].rotation.set(pos, 0, 0);
};
```

每当 update 方法被调用的时候，相关的骨头都会被设置到 pos 所指定的位置。这里缺少的是如何以固定的时间间隔调用 update 方法。为此，我们会使用本章开始时见过的 Tween.js 库（更多细节可以参考源代码）。

正如你所看到的，使用骨骼动画需要做更多事情，但是比使用固定的变形目标更加灵活。在这个示例里我们改变的只是骨头的旋转角度，你还可以改变其位置或缩放比例。下一节，我们要看一看如何从外部模型中加载动画。我们还要用到该示例，不过这次运行的是模型中预定义好的动画，不需要手工移动这些骨头。

9.4 使用外部模型创建动画

在第 8 章我们已经见过了几个 Three.js 支持的三维文件格式。这些格式中有几个也支持动画。本节我们会看如下的几个例子：

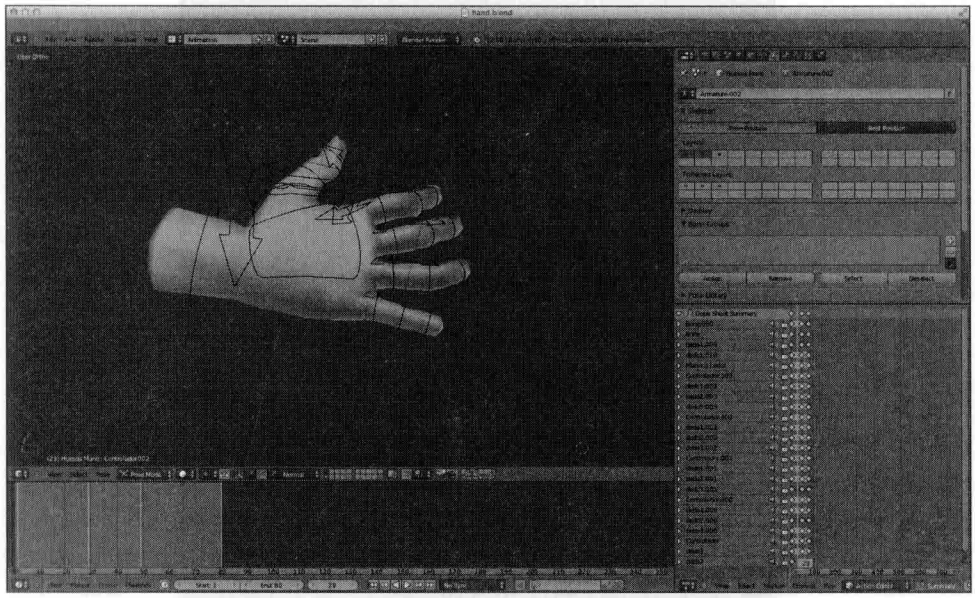
- **带有 JSON 导出器的 Blender**：我们先在 Blender 里创建一个动画，然后将它以 Three.js 的 JSON 格式导出。
- **Collada 模型**：Collada 模型也支持动画。在这个例子里我们从一个 collada 文件中加载动画，然后用 Three.js 渲染。
- **MD2 模型**：MD2 模型是老式雷神引擎所使用的简单格式。尽管这种格式有些老了，但它依然是一种非常好的保存角色动画的文件格式。

我们将从 Blender 模型开始。

9.4.1 用 Blender 创建骨骼动画

开始用 Blender 做动画，你可以加载 models 文件夹下的模型。你可以在那里找到一个名

为 hand.blend 的文件，并把它加载到 Blender 中。



限于篇幅，本书不能在这里涉及太多在 Blender 中如何创建动画的细节，但是你可以记住下面这几件事：

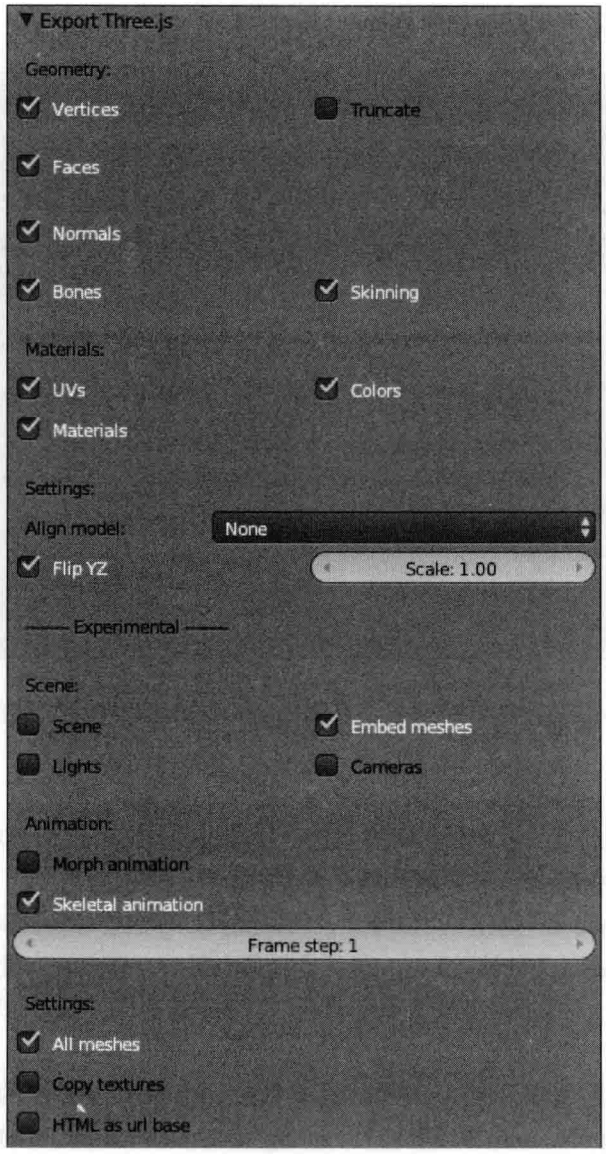
- 模型中的顶点至少要在一个顶点组中。
- Blender 中顶点组的名字必须跟控制这个顶点组的骨头的名字相对应。只有这样，当骨头被移动时 Three.js 才能找到需要修改的顶点。
- 只有第一个 action（动作）可以导出，所以要保证你想要导出的动画是第一个 action。
- 创建 keyframes（关键帧名）时，最好选择所有骨头，即便它们没有变化。
- 导出模型时，要保证模型处于静止状态。如果不是这样，那么你看到的动画将会非常混乱。

有关在 Blender 里创建和导出动画的更多信息，以及上述各点的原因，你可以参考 http://devmatrix.wordpress.com/2013/02/27/creating-skeletalanimation-in-blender-and-exporting-it-to-three-js/，这是个非常好的资源。

在 Blender 里创建好动画之后，你可以用我们在上一章用过的 Three.js 导出器将动画导出成文件。当使用 Three.js 导出器导出文件时，你需要保证选中下面所有的选项：

这样你就可以将 Blender 中定义的动画以骨骼动画的方式导出，而不是变形动画。通过骨骼动画可以导出骨骼的移动，然后可以在 Three.js 中重新播放这个移动。

在 Three.js 里加载这个模型跟前面的例子一样。但是，当模型加载好之后，动画也创建好了：

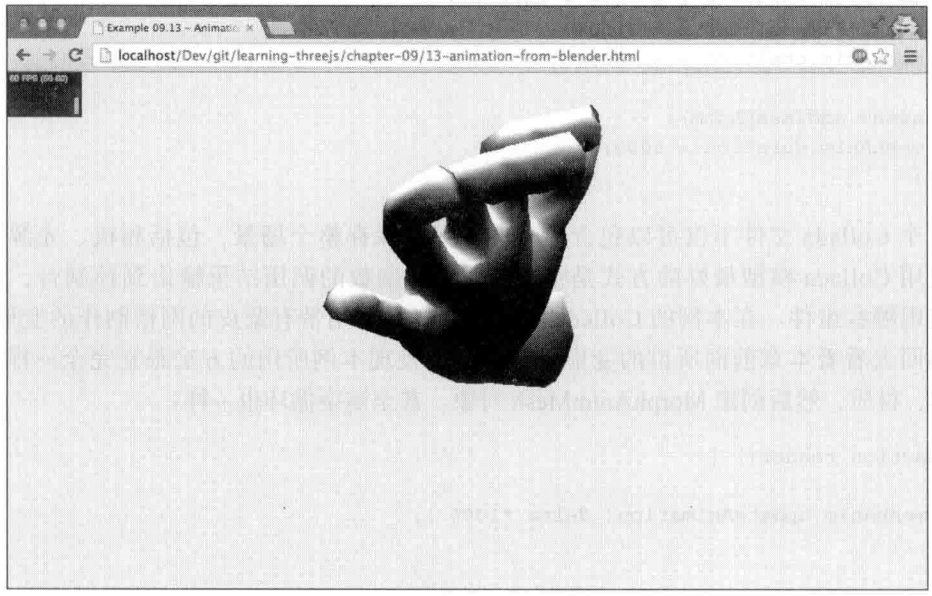



```
var loader = new THREE.JSONLoader();
loader.load('../assets/models/hand-2.js',
  function (geometry, mat) {

  // register the animation
  THREE.AnimationHandler.add(geometry.animation);

  // create a material
  var mat = new THREE.MeshLambertMaterial(
    {color: 0xF0C8C9, skinning: true});
```

```
    // create and position the mesh
    mesh = new THREE.SkinnedMesh(geometry, mat);
    mesh.rotation.x = 0.5 * Math.PI;
    mesh.rotation.z = 0.7 * Math.PI;
    scene.add(mesh);

    // create the animation
    var animation = new THREE.Animation(mesh, "wave");

    // start the animation
    animation.play();

}, '../assets/models');
```

跟之前示例的区别是：我们先用 Three.js 的 AnimationHandler 注册动画，即调用 AnimationHandler.add 函数。这样我们就可以使用新的 THREE.Animation（mesh，"wave"）语句来创建动画。这个动画的名词必须跟 Blender 中的名字一致。最后设置动画开始播放。

正如你所猜到的，我们仍然要做一些事情才能真正开始运行动画。在我们的 render 循环里调用 THREE.AnimationHandler.update（clock.getDelta()）函数来更新动画，然后 Three.js 就会用这些骨头将模型移动到正确位置。这个示例（13-animation-from-blender.html）的结果是一只简单的正在挥动的手。参考如下的截图：



除了 Three.js 自己的格式，我们还可以使用其他几种格式来定义动画。首先我们要看的是加载一个 Collada 模型。

9.4.2 从 Collada 模型中加载动画

从 Collada 文件中加载模型跟从其他格式文件中加载模型一样。首先要包含正确的 JavaScript 加载器文件：

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

然后我们要创建一个加载器，并用它来加载模型文件：

```
var loader = new THREE.ColladaLoader();
loader.load('../assets/models/monster.dae', function (collada) {

  var geom = collada.skins[0].geometry;
  var mat = collada.skins[0].material;

  // create a smooth skin
  geom.computeMorphNormals();
  mat.morphNormals = true;

  // create the animation
  meshAnim = new THREE.MorphAnimMesh(geom, mat);

  // position the mesh
  meshAnim.scale.set(0.15, 0.15, 0.15);
  meshAnim.rotation.x = -0.5 * Math.PI;
  meshAnim.position.x = -100;
  meshAnim.position.y = -60;

  scene.add(meshAnim);
  meshAnim.duration = 5000;
});
```

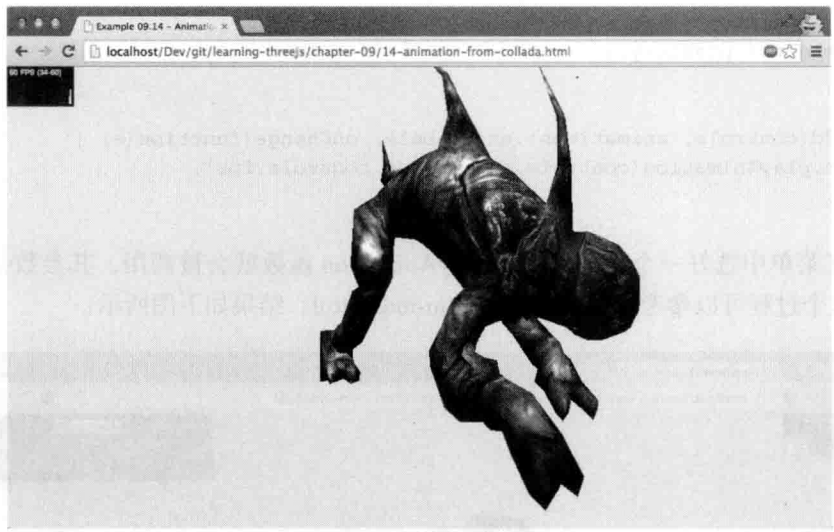
一个 Collada 文件不仅可以包含模型，还可以保存整个场景，包括相机、光源、动画等。使用 Collada 模型最好的方式是将 loader.load 函数的调用结果输出到控制台，然后再决定使用哪些组件。在本例的 Collada 文件中只有一个用带有蒙皮的网格制作的变形动画。如果你回去看看本章前面所讲的变形目标，你会发现本例所用的方法跟它完全一样。获取几何体、材质，然后创建 MorphAnimMesh 对象。甚至渲染循环也一样：

```
function render() {
  ...
  meshAnim.updateAnimation( delta *1000 );
  ...
}
```

加载这个 Collada 文件的结果如下图所示：

对于那些带有骨骼的 Collada 文件，你可以使用 9.3.2 节中的方法。

最后一个外部模型的例子是 MD2 文件格式，该例用的也是变形目标。



9.4.3　从雷神之锤模型中加载动画

MD2 格式是设计用来构建雷神之锤中的角色模型。尽管新一代引擎使用了不同的格式，但是你依然可以找到很多 MD2 格式的模型。为了使用这种格式的文件，我们首先要把它们转换成 Three.js 的 JavaScript 格式。你可以在网址 http://oos.moxiecode.com/js_webgl/md2_converter/ 完成这个转换：

转换好之后，你会得到一个 Three.js 格式的 JavaScript 文件，你可以用 MorphAnimMesh 类来加载和渲染。在前面的小节里我们已经看过如何加载，所以这里略过加载模型的代码。但是代码中有一个值得关注的是：我们会指定需要播放的动画的名字，而不是播放整个动画：

```
mesh.playAnimation('crattack', 10);
```

原因是 MD2 文件中通常会保存几个角色动画。但是幸运的是 Three.js 提供了一种功能可以用来选择动画，并调用 playAnimation 函数进行播放。我们首先要做的就是让 Three.js 来解析动画：

```
mesh.parseAnimations();
```

其返回结果是一组动画的名称，这组动画可以用 playAnimation 函数播放。在我们的示例中，你可以从右上角的菜单中选择要播放的动画的名称。可以用如下方法决定哪些动画可以播放：

```
mesh.parseAnimations();

var animLabels = [];
for (var key in mesh.geometry.animations) {
  if (key === 'length' ||
    !mesh.geometry.animations.hasOwnProperty(key))
```

```
        continue;
    animLabels.push(key);
}

gui.add(controls,'animations',animLabels).onChange(function(e) {
    mesh.playAnimation(controls.animations,controls.fps);
});
```

一旦在菜单中选好一个动画，mesh.playAnimation 函数就会被调用，其参数是所选动画的名称。这个过程可以参考 15-animation-from-md2.html。结果如下图所示：

9.5 总结

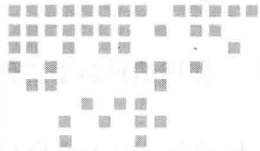
本章我们看了几种不同的让场景动起来的方法。我们从基础的动画技巧开始，然后是相机的移动和控制，最后是用变形目标和骨骼 / 骨头来制作动画。本章最重要的内容包括：

- 只要有了 render 循环，那么添加动画就很简单。只要修改网格的属性即可，Three.js 会在下次渲染时更新网格。
- Three.js 提供了很多相机控件。尽管它们的功能有些类似，不过也有它们各自适用的地方。如果你找不到一个能够切实满足你要求的控件，你可以研究一下代码，看看如何配置，或者基于该控件自己开发一个。
- 如果你想要一个相机控件，让你在场景中移动时只能朝四周看，那么你可以使用 PathControl。
- 让模型动起来主要有两种方法，使用变形目标或者骨骼动画。使用变形目标时，你可以让你的模型从一个关键帧变换到另一个关键帧，从而创建出动画。使用骨骼动

画时，你只要移动骨头就可以让模型动起来。根据这些骨头的移动，Three.js 会更新那些绑定到骨头上的顶点。

- 加载模型时，刚开始最好先将模型输出到控制台。使用编辑器手工创建新模型、更新材质或者修正一些小问题。
- 使用变形目标和骨骼时，Three.js 有两个很好的辅助类。对于变形目标，可以使用 MorphAnimMesh 类；对于骨骼动画，可以使用 SkinnedMesh 类。

在前面的章节里，我们已经见过了各种可以用来覆盖对象的材质。我们见过如何修改材质的颜色、光泽和透明度。但是我们还没有详细讨论的是如何在材质中使用外部图片（也叫纹理）。使用纹理你可以很容易地创建出一些对象，看上去像是用木料、金属、石头等做出来的。下一章，我们将会探索纹理的各个方面，以及如何在 Three.js 中使用它们。



Chapter 10 第 10 章

加载和使用纹理

在第 4 章我们向你介绍了 Three.js 中的各种材质，但是并没有讲如何将纹理应用到网格上。我们将会在本章讨论这个主题。具体来说，我们将会涉及以下内容：

- 在 Three.js 里加载纹理并应用到网格上。
- 使用凹凸贴图和法线贴图为网格添加深度和细节。
- 使用光照贴图创建假阴影。
- 使用环境贴图在材质上添加反光细节。
- 使用高光贴图，让网格的某些部分变得“闪亮”。
- 通过修改网格的 UV 贴图，对贴图进行微调。
- 将 HTML5 画布和视频元素作为纹理输入。

前面这些主题都需要加载、展示纹理。另外，我们还会快速看一下如何通过顶点着色器和片段着色器来定制你自己的着色器。但是我们将会从最基本的例子开始，展示如何加载和应用纹理。

10.1 在材质中使用纹理

在 Three.js 中纹理有几种不同的用法。你可以用它们来定义网格的颜色，也可以用它们来定义高光、凹凸和反光。但我们首先要看的是最基础的方法，用纹理为网格上的每个像素指定颜色。

10.1.1 加载纹理并应用到网格

纹理最基础的用法是在材质上设置贴图。当你使用这个材质时（和几何体一起构建网格），网格就会拥有颜色，而这个颜色则来源于纹理。

可以用如下的方式来加载纹理并应用于网格：

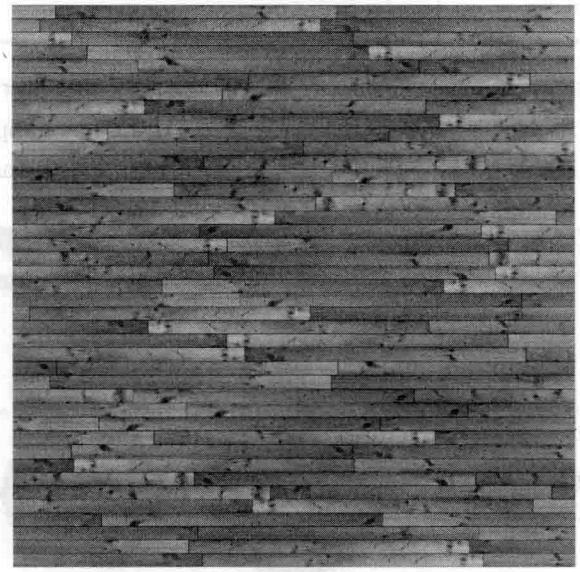
```
function createMesh(geom, imageFile) {
  var texture = THREE.ImageUtils.loadTexture
    ("../assets/textures/general/" + imageFile)

  var mat = new THREE.MeshPhongMaterial();
  mat.map = texture;

  var mesh = new THREE.Mesh(geom, mat);
  return mesh;
}
```

在这段代码中我们用 THREE.ImageUtils.loadTexture 函数从指定的位置加载图片文件。你可以用 PNG、GIF 或 JPEG 文件作为纹理的输入格式。注意，纹理的加载是异步的。这对我们来说这不是问题，因为我们有一个 render 循环，大约每秒钟渲染场景 60 次。如果你想在纹理加载完之前一直等待，可以使用如下的方法：

```
texture = THREE.ImageUtils.loadTexture('texture.png', {},
  0020function() { renderer.render(scene); });
```

在这个例子里我们为 loadTexture 提供了一个回调函数，纹理加载完时调用。我们的示例没有使用回调函数，而是依赖 render 循环在纹理加载完时显示纹理。

几乎所有图片都可以当作纹理来使用。但是为了达到最佳效果，最好使用正方形的图片，其长宽大小是 2 的次方。例如大小为 256 × 256、512 × 512、1024 × 1024 的图片最合适。



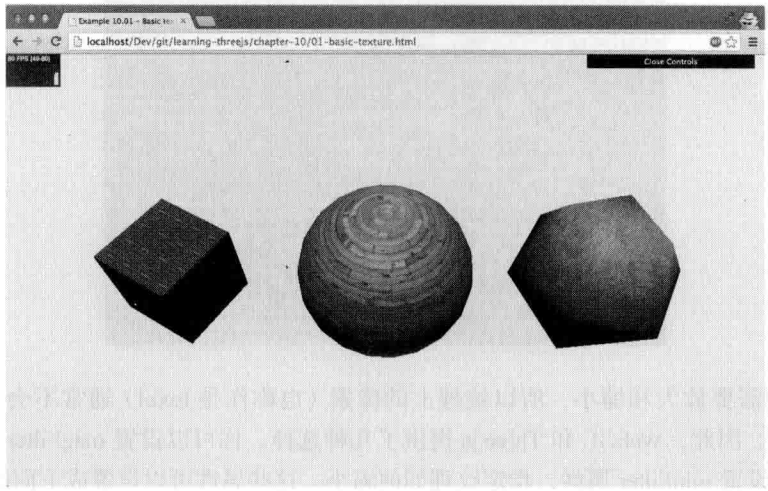
由于纹理需要放大和缩小，所以纹理上的像素（也称作是 texel）通常不会一对一地映射成面上的像素。因此，WebGL 和 Three.js 提供了几种选择。你可以设置 magFilter 属性，指定纹理如何放大；设置 minFilter 属性，指定纹理如何缩小。这些属性可以设置成下面的两个基础值：

名　称	描　述
THREE.NearestFilter（最近过滤器）	这个过滤器使用能够找到的最近 texel（纹理上的像素）的颜色。用于放大时，这会导致方块化；用于缩小时，这会丢失很多细节
THREE.LinearFilter（线性过滤器）	这个过滤器比较高级，它会使用周围四个 texel 的颜色值来确定颜色。这样虽然在缩小时仍然会丢失很多细节，但在放大时会平滑很多，方块化也比较少

除了这些基础值，我们还可以使用 mipmap。一个 mipmap 是一组纹理图片，每个图片的尺寸都是前一张图片的一半。这些图片是在加载纹理时创建的，可以生成比较光滑的过滤效果。所以如果你有一个正方形的纹理（尺寸为 2 的次方），只需要稍稍几步就可以达到更好的过滤效果。这些属性可以设置成下面这些值：

名　称	描　述
THREE.NearestMipMapNearestFilter	这个过滤器会选择最贴近目标解析度的 mipmap，然后应用前表中所讲的最近过滤原则。放大时仍然会有方块化，但是缩小时会好很多
THREE.NearestMipMapLinearFilter	这个过滤器选择的不是一个 mipmap，而是层次最近的两个 mipmap。然后在这两层上应用最近过滤原则获取两个中间值。这两个中间值会传递给一个线性过滤器，以获取最终结果
THREE.LinearMipMapNearestFilter	这个过滤器会选择最贴近目标解析度的 mipmap，然后应用前表中所讲的线性过滤原则
THREE.LinearMipMapLinearFilter	这个过滤器选择的不是一个 mipmap，而是层次最近的两个 mipmap。然后在这两层上应用线性过滤原则获取两个中间值。这两个中间值会传递给一个线性过滤器，以获取最终结果

如果你没有明确指定 magFilter 和 minFilter 属性的值，对于 magFilter 属性，Three.js 会使用 THREE.LinearFilter，对于 minFilter 属性，Three.js 会使用 THREE.LinearMipMapLinearFilter。在我们的示例中使用的就是这些默认值。关于基础纹理的示例可以在文件 01-basic-texture.html 中找到。

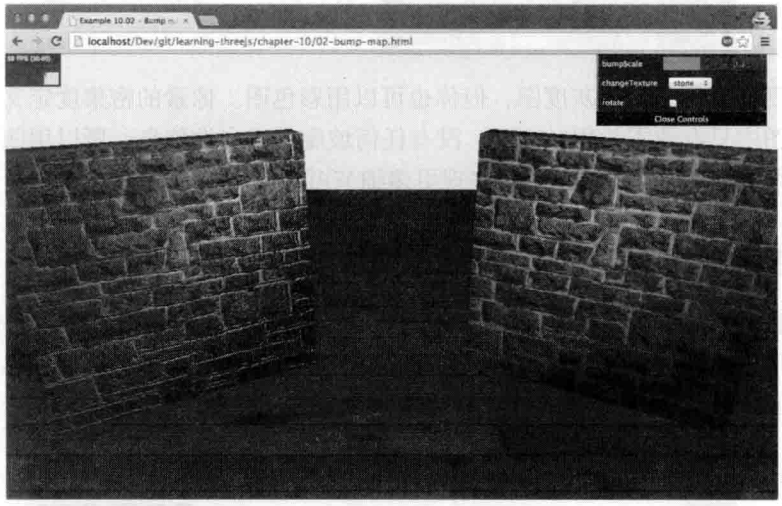


在这个示例里我们加载了几个纹理（使用你前面看到的代码），并把它们应用到不同的图形上。在这个示例中你可以看到，纹理很好地贴合在图形上。在 Three.js 里创建几何体时，任何纹理都能保证正确贴合。这是通过所谓的 UV 贴图（稍后进一步讨论）来完成的。通过 UV 贴图，我们可以告诉渲染器将纹理的某一部分应用到指定的面上。最简单的例子是给方块贴图。方块上某一面的 UV 贴图是这样：(0，1)，(0，0)，(1，0)，(1，1)。其含义是将整个纹理（UV 值的范围是 0 到 1）应用到这个面上。

我们在这个示例里用纹理指定网格像素的颜色。我们还可以将纹理用作其他目的。下面的两个例子展示材质是如何着色的。你可以用这种技术在网格表面上创建出凹凸不平、皱皱巴巴的效果。

10.1.2　使用凹凸贴图创建皱纹

凹凸贴图的目的是为材质增加厚度。理解这点最好的方式是打开如下的示例：02-bump-map.html。参考下面的屏幕截图：

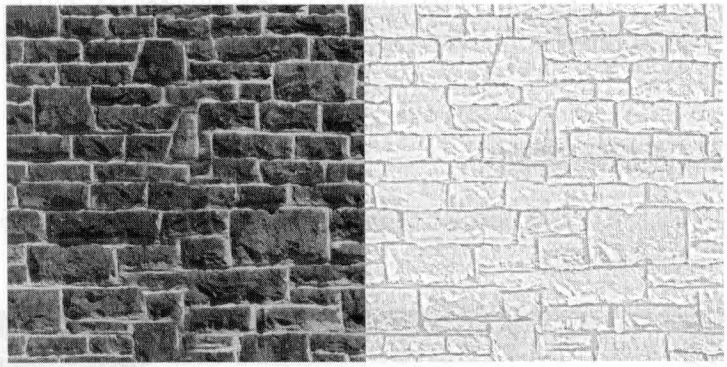


在这个示例里你可以看到：跟右面那面墙比较，左边的墙细节更多，而且看上去也更厚。这是通过为材质设置额外的纹理（所谓的凹凸贴图）来实现的：

```
function createMesh(geom, imageFile, bump) {
  var texture = THREE.ImageUtils.loadTexture(
    "../assets/textures/general/" + imageFile)
  var mat = new THREE.MeshPhongMaterial();
  mat.map = texture;

  var bump = THREE.ImageUtils.loadTexture(
    "../assets/textures/general/" + bump)
  mat.bumpMap = bump;
mat.bumpScale = 0.2;
```

```
    var mesh = new THREE.Mesh(geom, mat);
    return mesh;
}
```

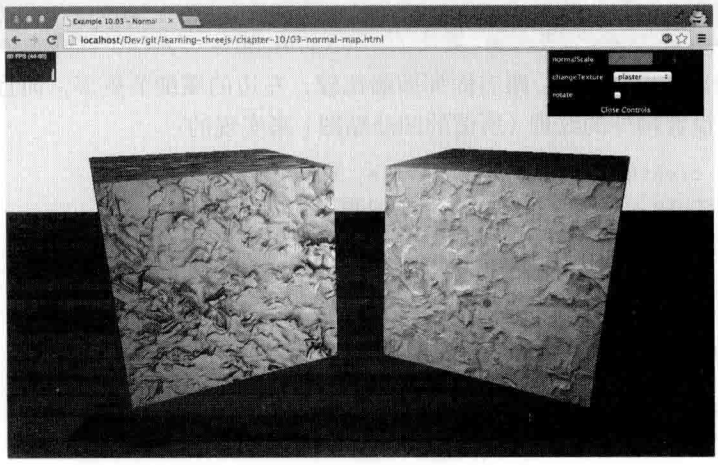
在这段代码中你可以看到，除了设置 map 属性，我们还设置了纹理的 bumpMap 属性。另外，通过 bumpScale 属性，我们还可以设置凹凸的高度（如果是负数，则指的是深度）。本例所用的纹理如下图所示：



这里的凹凸贴图是一张灰度图，但你也可以用彩色图。像素的密集度定义的是凹凸的高度。凹凸贴图中只有像素的相对高度，没有任何坡度的方向性信息。所以用凹凸贴图所能达到的厚度和细节程度是有限的。要想实现更多细节可以使用法向贴图。

10.1.3 使用法向贴图创建更加细致的凹凸和皱纹

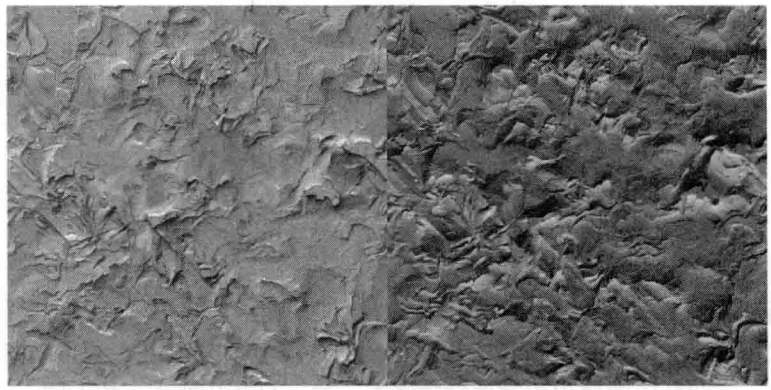
法向贴图中保存的不是每个像素的高度，而是像素的法向向量。简单来讲，使用法向贴图你只需用很少的顶点和面，就可以创建出细节非常丰富的模型。例如，你可以参照示例 03-normal-map.html，如下图所示：



在这张图片的左边你会看到一个细节丰富的、皱皱巴巴的方块。光源绕着方块移动，而且你也会看到纹理会对光源做出非常自然的反应。这样的结果就是一个看上去很真实的模型，而且只需要一个非常简单的模型，以及几个纹理。下面的代码展示的就是如何在 Three.js 里使用法向贴图：

```
function createMesh(geom, imageFile, normal) {
  var t = THREE.ImageUtils.loadTexture
    ("../assets/textures/general/" + imageFile);
  var m = THREE.ImageUtils.loadTexture
    ("../assets/textures/general/" + normal);
  var mat2 = new THREE.MeshPhongMaterial({
    map: t,
    normalMap: m
  });
  var mesh = new THREE.Mesh(geom, mat2);
  return mesh;
}
```

这里所用的方法跟凹凸贴图一样。只是这次我们将 normalMap 属性设置为一个法向纹理。我们还可以指定凹凸的程度，方法是设置 normalScale 属性：mat.normalScale.set（1,1）。通过这两个属性，你可以沿着 x 轴和 y 轴进行缩放。但是最好的方法是将它们的值设成一样的，以达到最佳效果。下图展示的就是纹理（左图）和法向贴图（右图）。



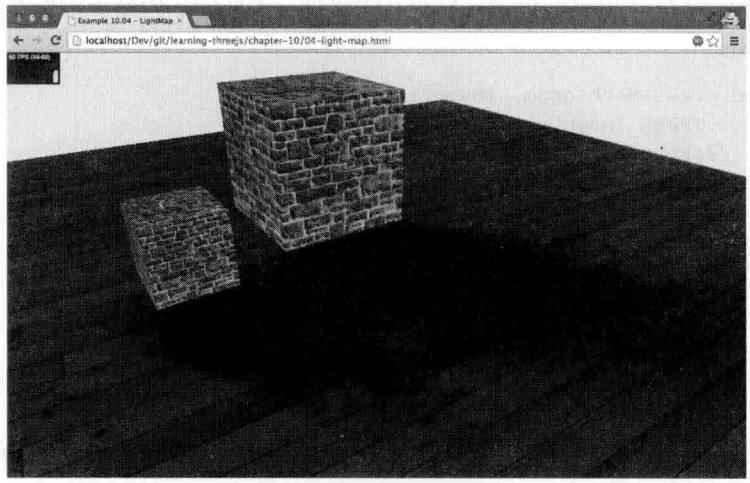
但是法向贴图的问题是不容易创建。你要用特殊的工具，例如 Blender 和 Photoshop。它们可以将高解析度的渲染结果或图片作为输入，从中创建出法向贴图。

在构建模型时，使用法向贴图是一个好方法，可以在低阶多边形上添加丰富的细节。下面的例子将会展示这是如何做到的。

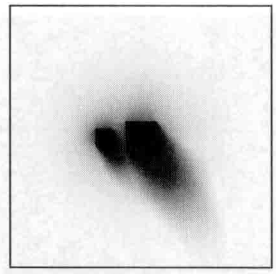
10.1.4　使用光照贴图创建假阴影

在前面的示例中，我们使用特定的贴图创建出了看上去比较真实的阴影，而且会对空间中的光照做出反应。还有一种方法可以用来创建虚假阴影。在本节我们用的是光照贴图。光

照贴图是预先渲染好的阴影，你可以用它来模拟真实的阴影。下图（来自示例 04-light-map.html）展示的就是这种效果：



在前面这个例子中你会看到两个非常真实的阴影，看上去就像是这两个方块的投影。但是，这两个阴影来源于一个光照贴图，如下图所示：



正如你所看到的，光照贴图中的阴影将会显示成地面上的阴影，从而模拟出真实阴影的效果。你可以用这种技术创建出解析度很高的阴影，而且不会损害渲染的性能。当然这只对静态场景有效。光照贴图的使用跟其他纹理基本一样，只有几处小小的不同：

```
var lm = THREE.ImageUtils.loadTexture(
  '../assets/textures/lightmap/lm-1.png');
var wood = THREE.ImageUtils.loadTexture(
  '../assets/textures/general/floor-wood.jpg');
var groundMaterial = new THREE.MeshBasicMaterial(
  {lightMap: lm, map: wood});
groundGeom.faceVertexUvs[1] = groundGeom.faceVertexUvs[0];
```

应用光照贴图时，我们只要将材质的 lightMap 属性设置成刚才所示的纹理即可。但是要将光照贴图显示出来我们还需要额外的几个步骤。我们需要为光照贴图明确指定 UV

映射（将纹理的哪一部分应用到表面）。只有这样你才能将光照贴图与其他纹理独立开来。在我们的例子中，只用基础的 UV 贴图，这是在创建地面时由 Three.js 自动创建。有关为什么要明确指定 UV 贴图的更多信息和背景知识可以参考网址：http://stackoverflow.com/questions/15137695/three-js-lightmap-causes-an-error-webglrenderingcontext-gl-error-gl-invalid-op。

放置好阴影的贴图之后，我们要把方块放在正确的位置，然后即可得到上述例子的效果。

Three.js 还提供了一种纹理，可以用来创建虚假的、高级的三维效果。下一节我们将会学习用环境贴图创建虚假的反光效果。

10.1.5　用环境贴图创建虚假的反光效果

计算环境反光非常耗费 CPU，而且通常会使用光线追踪算法。如果你想在 Three.js 里使用反光，你可以做，但是你不得不做一个假的。你可以通过创建一个对象所处环境的纹理来伪装反光，并将它应用到指定的对象上。我们先来看看所要达到的目标，参见 05-env-map-static.html，如下图所示：

在这个屏幕截图里你可以看到球和方块反射着周围环境。如果移动鼠标，你还可以看到这个反光是跟相机角度和你所看到的城市环境相关联的。要创建这样一个例子，我们要执行以下步骤：

1）**创建一个 CubeMap 对象**：我们首先要做的是创建一个 CubeMap 对象。一个 CubeMap 是有 6 个纹理的集合，而这些纹理可以应用到方块的每一个面上。

2）**创建一个带有这个 CubeMap 对象的方块**：带有 CubeMap 对象的方块就是移动相机时你所看到的环境。它可以在你向四周看时制造出一种幻象，好像你站在某个环境中一样。实

际上你是处在一个方块中，而这个方块内侧渲染出来的纹理让你感觉好像处在某个空间中。

3）**将 CubeMap 作为纹理**：我们用来模拟环境的 CubeMap 对象也可以用作网格的纹理。Three.js 会让它看上去像是环境的反光。

只要有制作材质的原材料，创建 CubeMap 对象就会非常简单。你所需要的是六张用来构建整个场景的图片。所以你需要如下的图片：朝前的（posz）、朝后的（negz）、朝上的（posy）、朝下的（negy）、朝右的（posx）、朝左的（negx）。Three.js 会将它们缝合在一起，创建出一个无缝的环境贴图。互联网上有几个网站可以下载这样的图片。本例所用的图片来自于 http://www.humus.name/index.php?page=Textures。

一旦你拿到了这些图片，你就可以用如下的代码来加载它们：

```
function createCubeMap() {

  var path = "../assets/textures/cubemap/parliament/";
  var format = '.jpg';
  var urls = [
    path + 'posx' + format, path + 'negx' + format,
    path + 'posy' + format, path + 'negy' + format,
    path + 'posz' + format, path + 'negz' + format
  ];

  var textureCube = THREE.ImageUtils.loadTextureCube( urls );
  return textureCube;
}
```

我们又一次用到了 THREE.ImageUtils 这个 JavaScript 对象，但是这次我们会传入一个纹理数组，用来创建 CubeMap 对象，并使用 loadTextureCube 函数。有了这个 CubeMap 对象之后，我首先要创建一个方块：

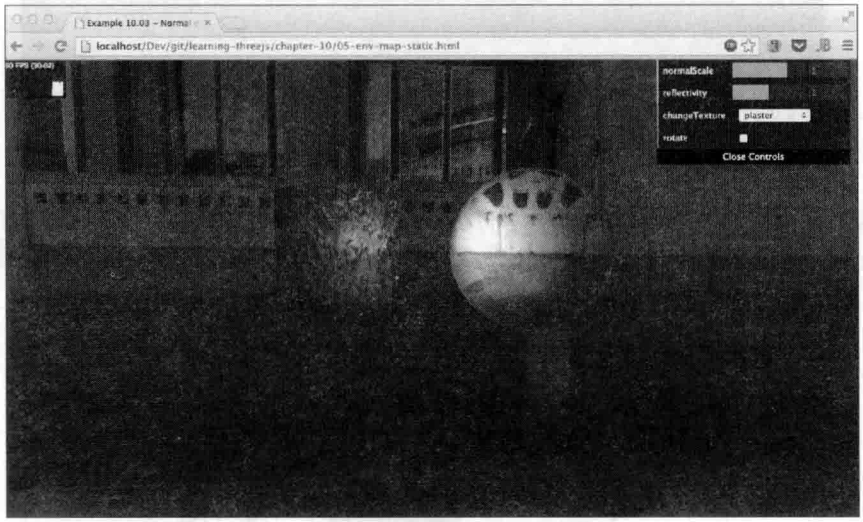
```
var textureCube = createCubeMap();
var shader = THREE.ShaderLib[ "cube" ];
shader.uniforms[ "tCube" ].value = textureCube;
var material = new THREE.ShaderMaterial( {
  fragmentShader: shader.fragmentShader,
  vertexShader: shader.vertexShader,
  uniforms: shader.uniforms,
  depthWrite: false,
  side: THREE.BackSide
});
cubeMesh = new THREE.Mesh(
  new THREE.CubeGeometry(100, 100, 100), material);
```

Three.js 提供了一个特别的着色器（var shader = THREE.ShaderLib["cube"];），结合 THREE.ShaderMaterial 类，我们可以基于 CubeMap 对象创建出一个环境。我们用 CubeMap 对象配置这个着色器，创建一个网格，并将它添加到场景中。如果从里面看，这个网格所代表的就是我们所处的那个虚假的环境。

同一个 CubeMap 对象可以应用到某个网格上，用来创建虚假的反光：

```
var sphere1 = createMesh(
  new THREE.SphereGeometry(10, 15, 15), "plaster.jpg");
sphere1.material.envMap = textureCube;
sphere1.rotation.y = -0.5;
sphere1.position.x = 12;
sphere1.position.y = 5;
scene.add(sphere1);

var sphere2 = createMesh(new THREE.CubeGeometry(10, 15, 15),
  "plaster.jpg","plaster-normal.jpg");
sphere2.material.envMap = textureCube;
sphere2.rotation.y = 0.5;
sphere2.position.x = -12;
sphere2.position.y = 5;
scene.add(sphere2);
```

正如你所看到的，我们将材质的 envMap 属性设置成我们创建的 cubeMap 对象，结果看上去好像我们站在一个宽阔的室外环境中，而这些网格上则会映射环境。如果使用滑块，你就可以设置材质的 reflectivity 属性。正如这个名字所暗示的那样，这个属性可以决定材质能够反射多少场景。除了反射，Three.js 还可以将 CubeMap 对象用于折射（类似玻璃的对象）：



要达到这个效果，我们只要修改纹理的加载方法即可：

```
var textureCube = THREE.ImageUtils.loadTextureCube(
  urls, new THREE.CubeRefractionMapping());
```

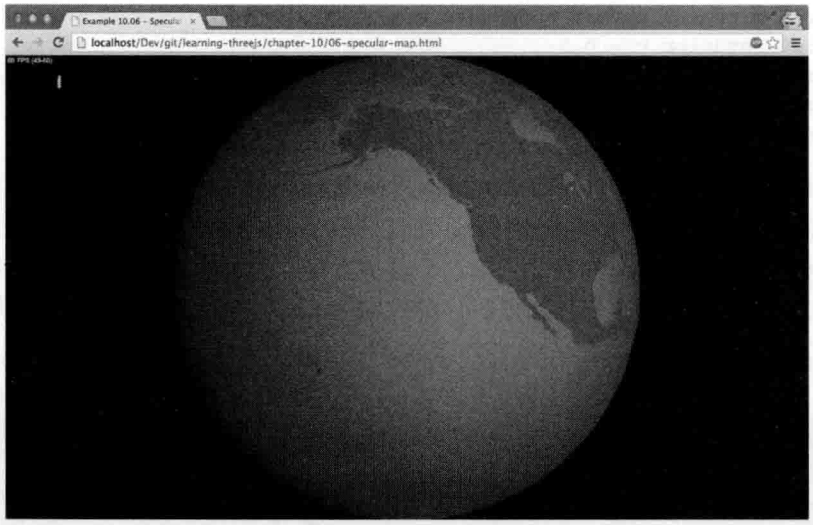
与 refelectivity 属性类似，通过材质的 refraction（折射）属性可以控制折射率。在这个示例里，我们在网格上应用的是静态环境贴图。也就是说我们只能看到环境的反光，看不到其他网格。在下面的例子里，我们将会向你展示如何创建一个能够映出场景中其他物体的

反光。

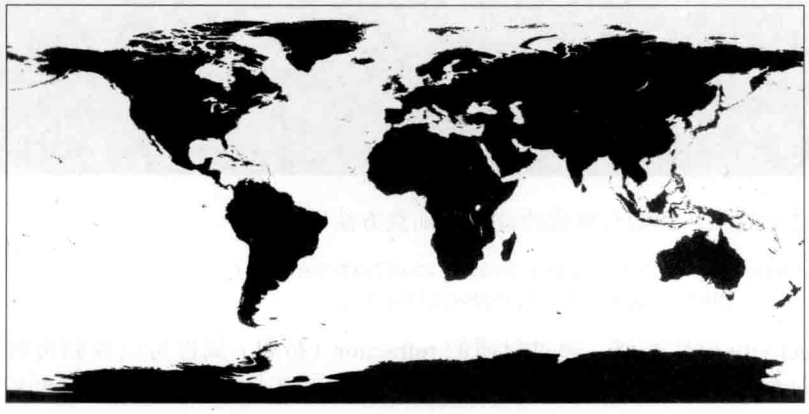
我们最后要看的基础纹理是高光贴图。

10.1.6 高光贴图

通过高光贴图你可以为材质指定一个闪亮的、色彩明快的贴图。例如，下图中的地球仪就是用高光贴图和法线贴图渲染出来的。你可以在浏览器中打开 06-specular-map.html，即可看到这个例子。其结果如下图所示：



在这个屏幕截图中你可以看到海洋的色彩比较明亮，而且还反光。但是陆地的色彩比较暗淡，也不反光（反射较少）。为达到该效果，我们并没有使用什么特殊的法向贴图，只使用了一个显示高度的法向贴图。海洋的高亮显示用的是下面这个高光贴图：



一般来讲，像素的值越高（从黑到白），物体表面就越闪亮。通常高光贴图会与 specular

属性一起使用，该属性可以用来决定反光的颜色。在本例中这个颜色是红色：

```
var specularTexture=THREE.ImageUtils.loadTexture(
  "../assets/textures/planets/EarthSpec.png");
var normalTexture=THREE.ImageUtils.loadTexture(
  "../assets/textures/planets/EarthNormal.png");

var planetMaterial = new THREE.MeshPhongMaterial();
planetMaterial.specularMap = specularTexture;
planetMaterial.specular = new THREE.Color( 0xff0000 );
planetMaterial.shininess = 1;

planetMaterial.normalMap = normalTexture;
```

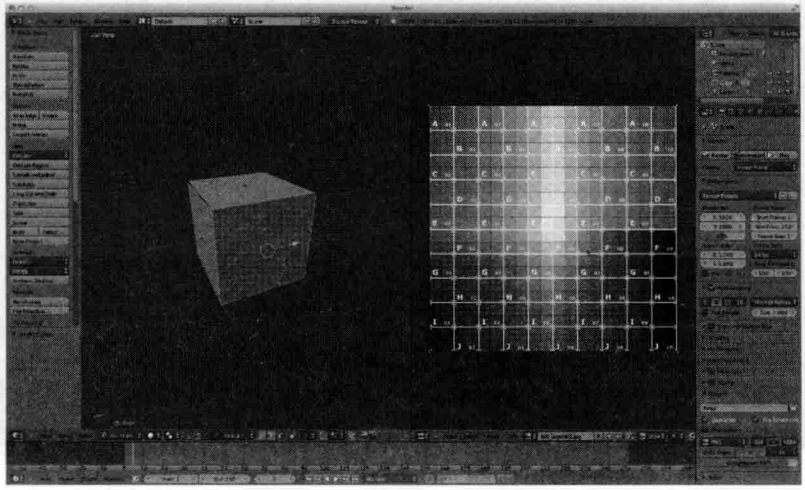
同时还要注意，最好的效果往往是用低光亮度实现的，但高光贴图还会受到光照的影响。所以你需要试验才能确定所需的效果。

10.2　纹理的高级用途

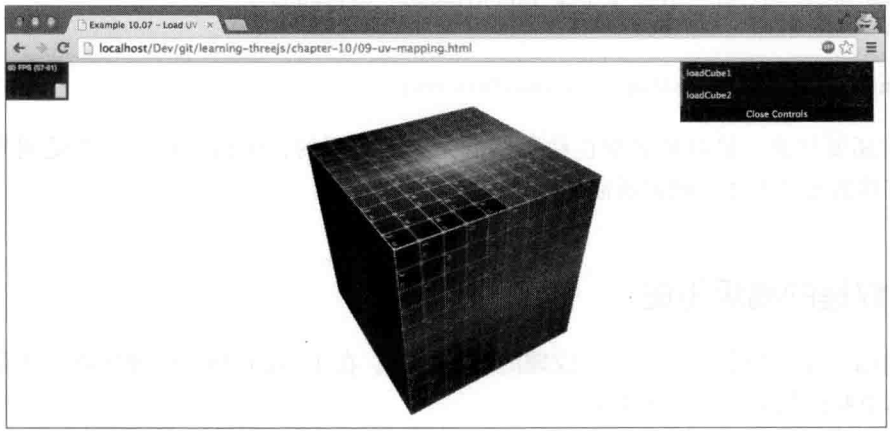
在前面的小节里我们看了一些纹理的基础用法。在 Three.js 里，纹理还有一些更高级的用途。本节我们就来看看这些用途。

10.2.1　定制 UV 映射

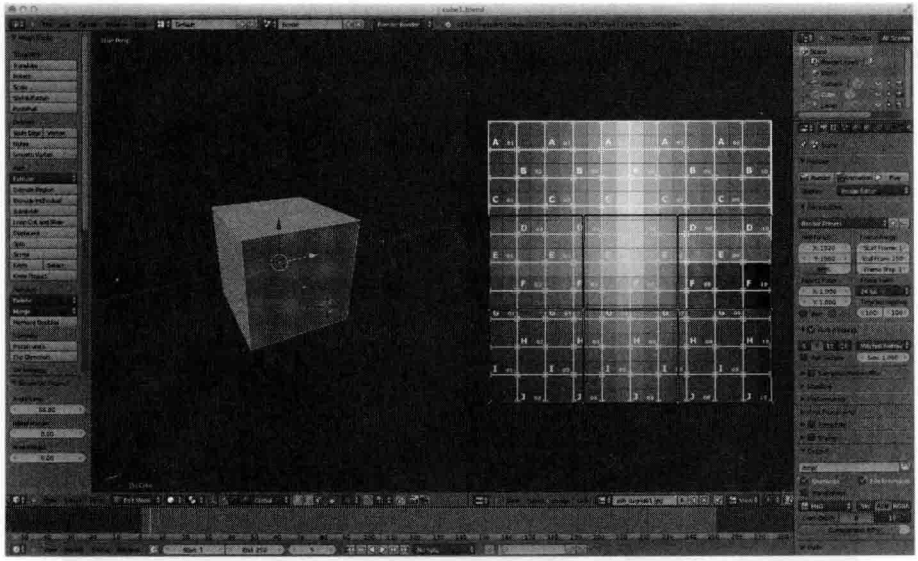
我们将从 UV 映射的深入观察开始。前面我们已经解释过，通过 UV 映射你可以指定纹理的哪一部分显示在物体表面上。当你在 Three.js 里创建一个几何体时，根据几何体的类型这些映射也一并自动创建。多数情况下，你不必修改默认的 UV 映射。要理解 UV 映射是如何工作的，最好的方法就是看一下 Blender 里的例子：



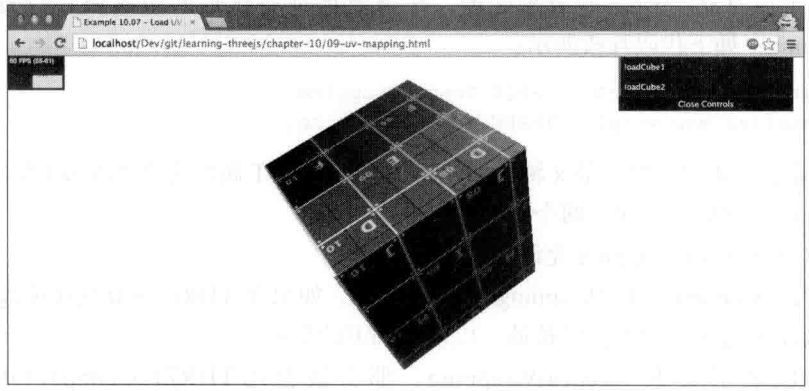
在这个示例里你可以看到两个窗口。左边窗口中有一个方块。右边窗口中是 UV 映射，这是我们加载的纹理示例，用来展示什么是映射。在这个示例里，我们在左边窗口中选了一个面，并在右边窗口中指定了这个面的 UV 映射。正如你所看到的，这个面上的每一个顶点都在右边 UV 映射的一个角上（用小圆圈标识）。这意味着整个纹理都会映射到那个面上。这个方块的其他各面也是如此处理的，这样我们就会得到一个展示所有纹理的方块。参考 07-uv-mapping.html，如下图所示：



这是 Blender（也是 Three.js）中在方块上贴图的默认行为。我们来改一下 UV 映射，看看修改后的纹理如何应用到物体表面上。这次我们在方块的每个面上都只显示部分纹理，而不是整个纹理。



如果用 Three.js 显示，你会看到纹理的应用方式已经不一样了。

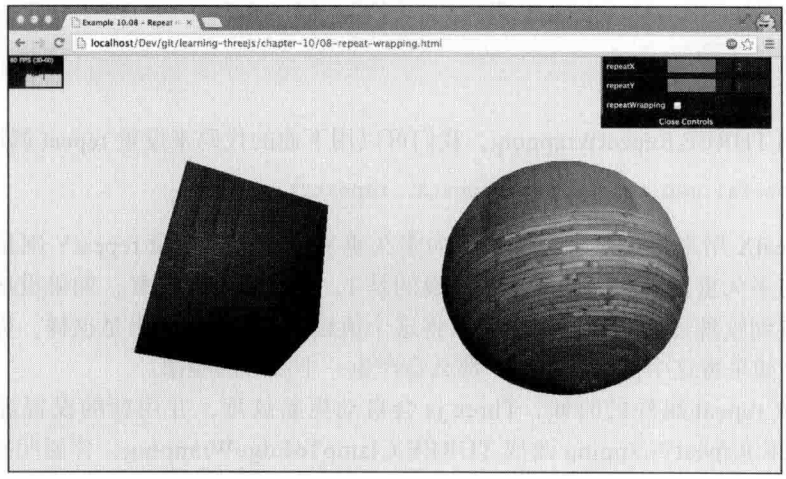


UV 映射的定制一般是在诸如 Blender 这样的软件中完成的，特别是当模型变得复杂时。这里需要记住的是 UV 映射有两个维度，U 和 V，取值范围是 0 到 1。定制 UV 映射时，你需要为物体的每个面指定其要显示纹理的哪一部分。为此你要为构成面的每一个顶点指定 u 坐标和 v 坐标。

接下来我们要看看如何复制纹理，可以使用 UV 映射内部的一些技巧。

10.2.2　重复映射

当你在 Three.js 创建的几何体上应用纹理的时候，Three.js 会尽量做到最优。例如，对于方块，Three.js 会在每一个面上显示完整的纹理；对于球体，Three.js 会用整个纹理包裹它。但是有一些情形，你可能不想将纹理遍布整个面或整个几何体，而是让纹理自己重复。Three.js 提供了一些功能可以实现这种控制。你可以打开示例 08-repeat-wrapping.html 来试验一下纹理的重复属性：



在这个例子里，你可以设置这个属性，控制属性如何复制它自己。

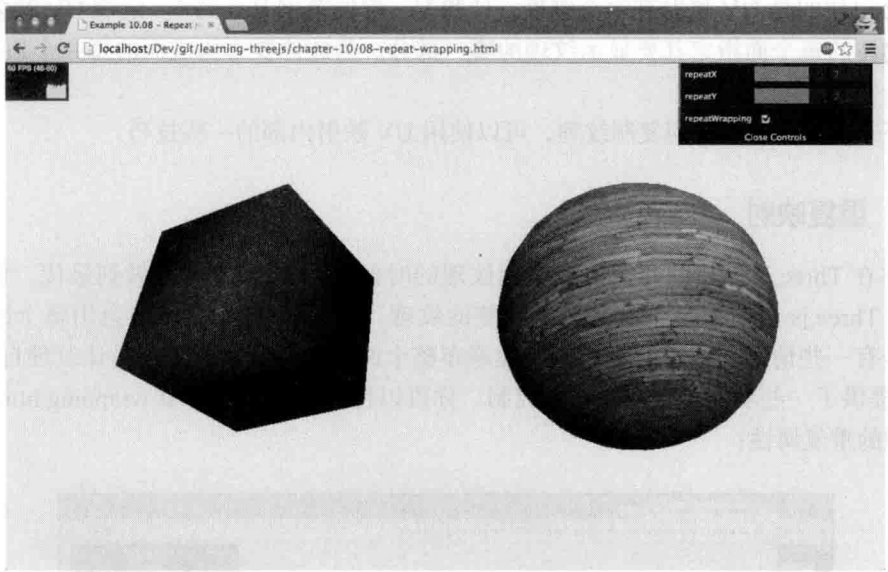
在用这个属性达到所需的效果之前，你需要保证将纹理的包裹属性设置为 THREE.RepeatWrapping，如下代码片段所示：

```
cube.material.map.wrapS = THREE.RepeatWrapping;
cube.material.map.wrapT = THREE.RepeatWrapping;
```

wrapS 属性定义的是纹理沿 x 轴方向的行为，而 wrapT 属性定义的是纹理沿 y 轴方向的行为。Three.js 为此提供了如下两个选项：

- THREE.RepeatWrapping 允许纹理重复自己。
- THREE.ClampToEdgeWrapping 是默认设置。如果是 THREE.ClampToEdgeWrapping，那么纹理边缘的像素会被拉伸，以填满剩下的空间。

如果你取消了菜单项 repeatWrapping，那么就会用 THREE.ClampToEdgeWrapping 选项。



如果使用 THREE.RepeatWrapping，我们可以用下面的代码来设置 repeat 属性：

```
cube.material.map.repeat.set(repeatX, repeatY);
```

变量 repeatX 用来指定纹理在 x 轴方向多久重复一次，而变量 repeatY 则是用来指定纹理在 y 轴方向多久重复一次。如果这些值设的是 1，则纹理不会重复；如果设成大一点儿的值，你就会看到纹理开始重复。你也可以将这个值设成小于 1。如果是这样，你就会看到纹理被放大了。如果将这个值设成负数，那么会产生一个纹理的镜像。

当你修改 repeat 属性的时候，Three.js 会自动更新纹理，并用新的设置进行渲染。如果你从 THREE.RepeatWrapping 改成 THREE.ClampToEdgeWrapping，你要明确更新纹理：

```
cube.material.map.needsUpdate = true;
```

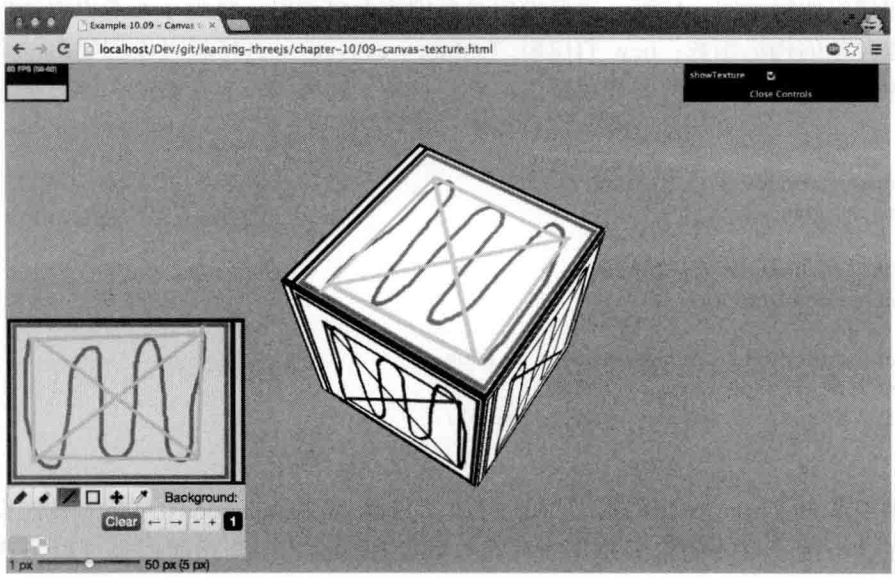
到目前为止我们一直是将静态图片作为纹理。但是 Three.js 也支持将 HTML5 的画布作为纹理。

10.2.3　在画布上绘制图案并作为纹理

本节我们将会来看一下两个不同的例子。首先我们会看一下如何在画布上创建简单的纹理，并应用到网格。然后我们会更进一步，创建一个画布，将随机生成的图形作为凹凸贴图。

10.2.3.1　用画布作纹理

在第一个例子里，我们会使用 literally 库（http://literallycanvas.com）创建一个交互式的画布，你可以在上面绘图。参见下图的左下角。你也可以打开示例：09-canvas-texture.html。



你画的任何东西都会作为纹理直接渲染到方块上。在 Three.js 中要达到这种效果非常简单，只需要几行代码。首先我们要创建一个画布元素，然后配置该画布使用 literally 库（只针对这个例子）：

```
<div class="fs-container">
<div id="canvas-output" style="float:left">
</div>
</div>
...
var canvas = document.createElement("canvas");
$('#canvas-output')[0].appendChild(canvas);
```

```
$('#canvas-output').literallycanvas(
  {imageURLPrefix: '../libs/literally/img'});
```

这并没有什么特殊。我们只是用 JavaScript 创建了一个 canvas（画布）元素，并将它添加到指定的 div 元素中。通过调用 literallycanvas 我们可以创建一个绘图工具，你可以用它直接画在画布上。接下来我们要将画布上的绘制结果作为输入创建一个纹理：

```
function createMesh(geom) {

  var canvasMap = new THREE.Texture(canvas);
  var mat = new THREE.MeshPhongMaterial();
  mat.map = canvasMap;
  var mesh = new THREE.Mesh(geom,mat);

  return mesh;
}
```

正如代码所揭示的，你唯一需要做的是在创建新纹理时，将画布元素以引用的方式传递给纹理对象的构造函数：new THREE.Texture（canvas）。这样就可以创建出一个以画布为来源的纹理。剩下要做的就是在渲染时更新材质，这样画布上最新的内容才会显示在方块上：

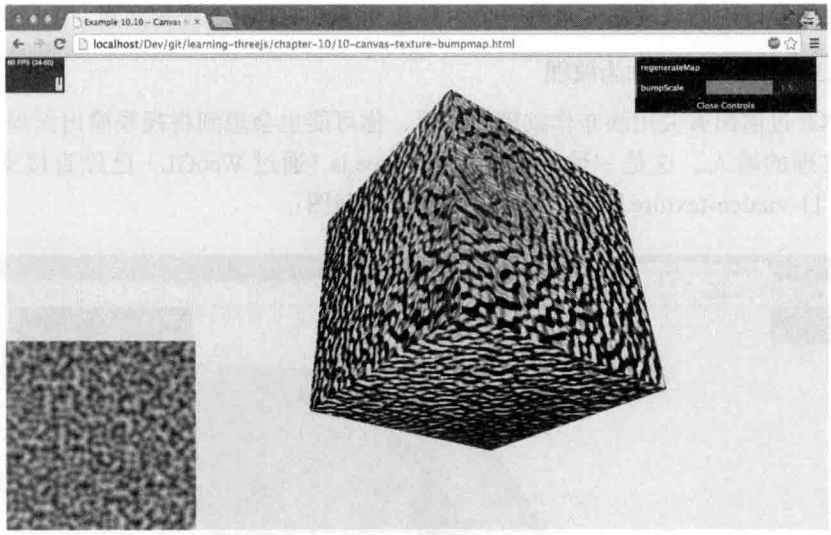
```
function render() {
  stats.update();

  cube.rotation.y += 0.01;
  cube.rotation.x += 0.01;

  cube.material.map.needsUpdate = true;
  requestAnimationFrame(render);
  webGLRenderer.render(scene, camera);
}
```

如果要告知 Three.js 我们需要更新材质，只要将纹理的 needsUpdate 属性设为 true 即可。在这个例子里我们将画布作为最简单的纹理的输入。当然我们也可以将这种方法应用于目前我们所看到的所有类型的贴图中。在下面这个例子中，我们将用它来生成凹凸贴图。

10.2.3.2 用画布作凹凸贴图

正如我们在本章前面所看到的，我们可以用凹凸贴图创建简单的有皱纹的纹理。贴图中像素的密集程度越高，贴图看上去越皱。由于凹凸贴图只是简单的黑白图片，所以没有道理不可以在画布上创建这个贴图，并将这个画布作为凹凸贴图的输入。

在下面的例子里，我们在画布上随机生成了一副灰度图，并将该图作为方块上凹凸贴图的输入。参见示例 09-canvas-texture-bumpmap.html：



完成该功能的 JavaScript 代码跟前面例子里的没有什么大的不同。我们要创建一个画布元素，然后用一些随机噪音填充该画布。至于噪音，我们可以使用 Perlin 噪音。Perlin 噪音（http://en.wikipedia.org/wiki/Perlin_noise）可以产生看上去非常自然的随机纹理，如上图所示。我们可以使用 https://github.com/wwwtyro/perlin.js 中的 Perlin 噪音函数，如下所示：

```
var ctx = canvas.getContext("2d");
function fillWithPerlin(perlin, ctx) {

  for (var x = 0; x < 512; x++) {
    for (var y = 0; y < 512; y++) {
      var base = new THREE.Color(0xffffff);
      var value = perlin.noise(x / 10, y / 10, 0);
      base.multiplyScalar(value);
      ctx.fillStyle = "#" + base.getHexString();
      ctx.fillRect(x, y, 1, 1);
    }
  }
}
```

我们用 perlin.noise 函数在画布 x 坐标和 y 坐标的基础上生成一个 0 到 1 之间的值。该值可以用来在画布上画一个像素点。可以用这个方法生成所有的像素点，其结果就是上图左下角的那个随机贴图。这个贴图即可作为凹凸贴图：

```
var bumpMap = new THREE.Texture(canvas);

var mat = new THREE.MeshPhongMaterial();
mat.color = new THREE.Color(0x77ff77);
mat.bumpMap = bumpMap;
bumpMap.needsUpdate = true;

var mesh = new THREE.Mesh(geom, mat);
return mesh;
```

最后我们要讨论的纹理输入是另一个 HTML 元素：HTML5 视频元素。

10.2.3.3 用视频输出作为纹理

如果你看过前面有关用画布作纹理的段落，你可能也会想到将视频输出到画布，然后将画布作为纹理的输入。这是一种方法，但是 Three.js（通过 WebGL）已经直接支持 HTML5 视频元素：11-viedeo-texture.html。参见下面的屏幕截图：

将视频作为纹理的输入跟用画布一样，都很简单。首先，我们要有一个视频元素可以播放视频：

```
<video  id="video"
  style="display: none;
  position: absolute; left: 15px; top: 75px;"
  src="../assets/movies/Big_Buck_Bunny_small.ogv"
  controls="true" autoplay="true">
</video>
```

这只是一个基础的 HTML5 视频元素，可以自动播放视频。接下来我们可以配置 Three.js 使用该视频作为纹理的输入：

```
var video  = document.getElementById('video');
texture = new THREE.Texture(video);
texture.minFilter = THREE.LinearFilter;
texture.magFilter = THREE.LinearFilter;
texture.generateMipmaps = false;
```

由于我们的视频不是正方形，所以要保证材质不会生成 mipmap。由于材质变化得很频繁，所以我们还需要设置简单高效的过滤器（参见 10.1.1 节）。在本例中，我们使用的是 MeshFaceMaterial 和 MeshBasicMaterial：

```
var materialArray = [];
materialArray.push(new THREE.MeshBasicMaterial(
  {color: 0x0051ba}));
materialArray.push(new THREE.MeshBasicMaterial(
  {color: 0x0051ba}));
materialArray.push(new THREE.MeshBasicMaterial(
  {color: 0x0051ba}));
materialArray.push(new THREE.MeshBasicMaterial(
  {color: 0x0051ba}));
materialArray.push(new THREE.MeshBasicMaterial(
  {map: texture }));
materialArray.push(new THREE.MeshBasicMaterial(
  {color: 0xff51ba}));

var faceMaterial = new THREE.MeshFaceMaterial(materialArray);
var mesh = new THREE.Mesh(geom,faceMaterial);
```

剩下要做的是要保证在 render 循环中更新材质：

```
if ( video.readyState === video.HAVE_ENOUGH_DATA ) {
  if (texture) texture.needsUpdate = true;
}
```

在这个例子里，我们只是将视频渲染到方块的一个面上。由于它只是一种普通的纹理，我们可以任意处理它。例如我们可以通过定制 UV 映射，沿着方块的面将它分成几部分。我们甚至还可以将视频作为凹凸贴图的输入。

10.3　总结

关于纹理的内容至此就结束了。正如你所看到的，Three.js 里有各种各样的纹理，每种都有独特的用途。使用纹理时最需要记住的是如下内容：

- 你可以使用任何一张 PNG、JPG 或 GIF 格式的图片作为纹理。图片的加载是异步的，所以请记住要么使用渲染循环，要么在加载纹理时提供一个回调函数。
- 使用正方形的纹理（其尺寸是 2 的次方，例如 256 × 256、512 × 512 和 1024 × 1024）可以达到最佳效果。原因是可以用 mipmap 缩放图片，从而达到较好的效果。
- 你可以在低阶模型上使用纹理创建出非常好看的图形。使用凹凸贴图和法向贴图可以创建出虚假的细节丰富，凹凸感很强的，带有阴影的简单模型。
- 标准情况下，Three.js 并不支持反光。但是你可以用环境贴图很容易地创建出虚假的反光。
- 如果你想直接控制物体表面的光亮程度，你可以使用高光贴图。
- 你可以设置纹理的 repeat 属性，从而让纹理可以自我复制。还要记住将材质的包裹属性从 ClampToEdgeWrapping 改为 RepeatWrapping。
- 在 Three.js 里，可以用 HTML5 的画布元素或视频元素创建动态纹理。你只需要将这

些元素作为纹理的输入，并在更新纹理时，将 needUpdate 属性设为 true。

到本章为止，我们差不多已经涵盖了 Three.js 中所有重要的概念。但是我们还没有涉及 Three.js 中一个有趣的功能，即所谓的后期处理。通过后期处理，你可以在场景渲染完毕后添加一些效果。例如，你可以让场景变得模糊，或者变得色彩艳丽，或者使用扫描线添加一种类似电视的效果。下一章，我们就来介绍后期处理，以及如何将它应用于你的场景中。

第 11 章 Chapter 11

定制着色器和渲染后期处理

本书接近结束了，但是有一个 Three.js 提供的主要功能——渲染后期处理——我们一直还没有涉及，我们将会在本章讨论这个主题。除了这个主题，我们还会在本章向你介绍如何创建自定义的着色器。我们在本章讨论的内容包括如下几点：

- 配置 Three.js 库，用于后期处理
- Three.js 提供的基本后期处理通道，例如 BloomPass（泛光通道）和 FilmPass（胶片通道）
- 使用掩码，将效果应用到部分场景
- 使用 TexturePass（纹理通道）保存渲染结果
- 使用 ShaderPass（着色器通道）添加更基础的后期处理效果，例如褐色滤镜、镜像效果，以及颜色调整
- 使用 ShaderPass 产生模糊效果，以及更高级的滤镜
- 通过开发一个简单的着色器，来创建自定义的后期处理效果

我们在第 1 章创建了一个 render 循环，并在本书中一直使用该循环来渲染场景、制作动画。对于后期处理，我们需要修改这个循环，好让 Three.js 库对最终的渲染结果进行后期处理。我们将会在 11.1 节介绍如何修改。

11.1 设置后期处理

设置 Three.js 库为后期处理做准备，我们需要通过以下步骤对当前的配置进行修改：

1）创建一个 EffectComposer（效果组合器）对象，然后在该对象上添加后期处理通道。

2）配置该对象，使它可以渲染我们的场景，并应用额外的后期处理步骤。

3）在 render 循环中，使用 EffectComposer 渲染场景、应用通道，并输出结果。

我们照例会提供一个示例，你可以在上面试验，或者根据需要进行调整。本章的第一个示例可以在文件 01-basic-effect-composer.html 中找到。你可以使用右上角的菜单调整本例中用到的各个后期处理步骤的属性，如下图所示：

11.1.1 创建 EffectComposer 对象

我们先来看看需要包含的 JavaScript 文件。这些文件可以在 Three.js 的发布包里找到，路径是 examples/js/postprocessing 和 examples/js/shaders。

至少包含下面的文件：

```
<script type="text/javascript"
        src="../libs/postprocessing/EffectComposer.js"></script>
<script type="text/javascript"
        src="../libs/postprocessing/MaskPass.js"></script>
<script type="text/javascript"
        src="../libs/postprocessing/RenderPass.js"></script>
<script type="text/javascript"
        src="../libs/shaders/CopyShader.js"></script>
<script type="text/javascript"
        src="../libs/postprocessing/ShaderPass.js"></script>
```

EffectComposer.js 文件提供 EffectComposer 对象，以便添加后期处理步骤。MaskPass.js、ShaderPass.js 和 CopyShader.js 文件是 EffectComposer 内部使用的文件，RenderPass.js 文件则可以用来在 EffectComposer 对象上添加渲染通道。如果没有这些通道，我们的场景就不会被渲染。

在这个示例里，我们添加了两个 JavaScript 文件，用来在场景中添加一种胶片效果，如下所示：

```
<script type="text/javascript"
        src="../libs/postprocessing/FilmPass.js"></script>
<script type="text/javascript"
        src="../libs/shaders/FilmShader.js"></script>
```

首先我们要创建一个 EffectComposer 对象，你可以在这个对象的构造函数里传入 WebGLRenderer，如下所示：

```
var composer = new THREE.EffectComposer(webGLRenderer);
```

接下来我们要在这个组合器中添加各种通道。

11.1.1.1　为后期处理配置 EffectComposer 对象

每个通道会按照其加入 EffectComposer 的顺序执行。第一个要加入的通道是 RenderPass。下面这个通道会渲染场景，但不会将渲染结果输出到屏幕上：

```
var renderPass = new THREE.RenderPass(scene, camera);
composer.addPass(renderPass);
```

创建 RenderPass 时要传入需要渲染的场景和所用的相机。调用 addPass（）函数，我们即可将这个 RenderPass 添加到 EffectComposer 对象中。接下来我们要添加一个可以将结果输出到屏幕的通道。不是所有的通道都可以做到这一点（稍后解释），但是本例中所用的 FilmPass 可以将其结果输出到屏幕。要添加 FilmPass，我们先要创建该对象，然后添加到效果组合器中。最终的代码大致如下所示：

```
var renderPass = new THREE.RenderPass(scene,camera);
var effectFilm = new THREE.FilmPass(0.8, 0.325, 256, false);
effectFilm.renderToScreen=true;

var composer = new THREE.EffectComposer(webGLRenderer);
composer.addPass(renderPass);
composer.addPass(effectFilm);
```

正如你所看到的，我们创建了一个 FilmPass 对象，并将它的 renderToScreen 属性设为 true。这个通道是在 RenderPass 后面添加的，所以如果使用了这个效果组合器，我们就可以看到输出结果。

11.1.1.2　修改渲染循环

现在我们需要稍稍修改一下渲染循环，用效果组合器来取代 WebGLRenderer：

```
var clock = new THREE.Clock();
function render() {
    stats.update();

    var delta = clock.getDelta();
    orbitControls.update(delta);

    sphere.rotation.y += 0.002;

    requestAnimationFrame(render);
    composer.render(delta);
}
```

在这段代码里我们移除了 "webGLRenderer.render（scene，camera）;"，用 "composer.render（delta）;" 替代。这将调用 EffectComposer 的 render() 函数。由于我们已经将 FilmPass

的 renderToScreen 属性设置成 true，所以 FilmPass 的结果将会输出到屏幕上。

完成这些基础配置后，我们将在接下来的几节看看另外几个后期处理通道。

11.2 后期处理通道

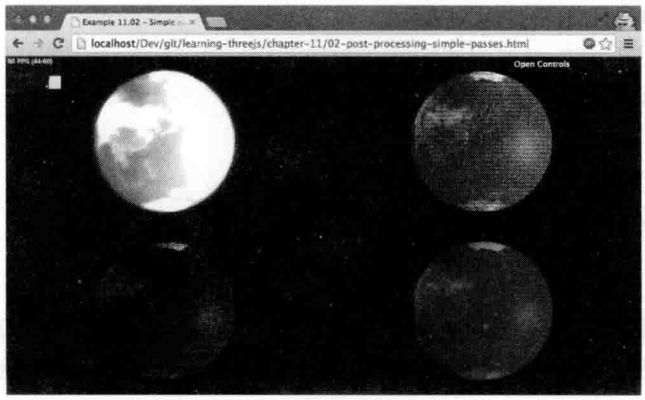
Three.js 库提供了几个后期处理通道，你可以直接将它们添加到 EffectComposer 对象中。下表是这些通道的概览。注意，实际使用之前最好试验一下本章提供的示例，观察一下这些通道的结果，理解它们的功能。

通　道	描　述
BloomPass	该通道会使得明亮区域渗入较暗的区域。模拟相机照到过多亮光的情形
DotScreenPass	将一层黑点贴到代表原始图片的屏幕上
FilmPass	通过扫描线和失真模拟电视屏幕
MaskPass	在当前图片上贴一层掩膜，后续通道只会影响被贴的区域
RenderPass	该通道在指定的场景和相机的基础上渲染出一个新的场景
SavePass	执行该通道时，它会将当前渲染步骤的结果复制一份，方便后面使用。这个通道在实际应用中作用不大，所以也没有用在我们的示例中
ShaderPass	使用该通道你可以传入一个自定义的着色器，用来生成高级的、自定义的后期处理通道
TexturePass	该通道可以将效果组合器的当前状态保存为一个纹理，然后可以在其他 EffectComposer 对象中将该纹理作为输入参数

我们先从简单的通道开始。

11.2.1 简单后期处理通道

对于简单的后期处理通道，我们可以看看 FilmPass、BloomPass 和 DotScreenPass。对于这些通道，你可以看看示例 02-post-processing-simple-passes.html。你可以在这个示例里试验这些通道，看看它们如何影响原始的输出。结果如下图所示：



我们在这个示例里同时展示了四个不同的场景。右上角的是 FilmPass，左上角的是 BloomPass，左下角的是 DotScreenPass，右下角的是原始渲染的结果。

在这个示例里，我们也使用了 ShaderPass 和 TexturePass 来重用原始渲染结果的输出，并将该输出作为其他三个场景的输入。所以在看各个通道之前，我们先来看看这两个通道：

```
var renderPass = new THREE.RenderPass(scene, camera);
var effectCopy = new THREE.ShaderPass(THREE.CopyShader);
effectCopy.renderToScreen = true;

var composer = new THREE.EffectComposer(webGLRenderer);
composer.addPass(renderPass);
composer.addPass(effectCopy);

var renderScene = new THREE.TexturePass(composer.renderTarget2);
```

在这段代码里我们设立了一个 EffectComposer 对象，该对象输出默认场景（右下角的那个）。这个组合器有两个通道：RenderPass（用来渲染场景）和 ShaderPass。如果用 CopyShader 来设置 ShaderPass，那么它渲染的输出结果就不会有进一步的后期处理；如果将 renderToScreen 属性设为 true，那么渲染结果将会输出到屏幕。如果你看过示例代码，那么你会发现相同的场景输出了四次，但是应用了不同的效果。我们可以使用 RenderPass 从零开始渲染场景，但是这样有点儿浪费，因为我们可以重用第一个效果组合器的输出。要做到这一点儿，我们要创建一个 TexturePass，并传入 composer.renderTarget2。现在我们就可以用 renderScene 变量作为其他效果组合器的输入，而不用从零开始渲染场景。让我们再来看看 FilmPass，以及如何将 TexturePass 作为输入。

11.2.1.1　用 FilmPass 创建类似电视的效果

我们已经在 11.1 节介绍了如何创建一个 FilmPass，现在让我们来看看如何结合上一节的 TexturePass 来使用该效果。参考下面的代码片段：

```
var effectFilm = new THREE.FilmPass(0.8, 0.325, 256, false);
effectFilm.renderToScreen=true;

var composer4 = new THREE.EffectComposer(webGLRenderer);
composer4.addPass(renderScene);
composer4.addPass(effectFilm);
```

要使用 TexturePass 唯一要采取的步骤是将它作为效果组合器的第一个通道。接下来我们可以添加 FilmPass，应用效果。FilmPass 本身接受如下四个参数：

属　性	描　述
noiseIntensity	通过该属性你可以控制屏幕的颗粒程度
scanlinesIntensity	FilmPass 会在屏幕上添加一些扫描线。通过该属性，你可以指定扫描线的显著程度
scanlinesCount	该属性可以控制显示出来的扫描线数量
grayscale	如果设为 true，输出结果将会转换成灰度图

实际上有两种方法传递这些参数。在这个示例里，它们是作为构造函数的参数传递的，但是你也可以直接设置它们，如下所示：

```
effectFilm.uniforms.grayscale.value = controls.grayscale;
effectFilm.uniforms.nIntensity.value = controls.noiseIntensity;
effectFilm.uniforms.sIntensity.value =
                    controls.scanlinesIntensity;
effectFilm.uniforms.sCount.value = controls.scanlinesCount;
```

在这段代码里我们使用了 uniforms 属性。该属性可以直接跟 WebGL 通信。在 11.3 节中，我们将会进一步讲解 uniforms。现在你只要知道你可以直接更改后期处理通道和着色器的配置，而且可以立即看到结果。

11.2.1.2 用 BloomPass 在场景中添加泛光效果

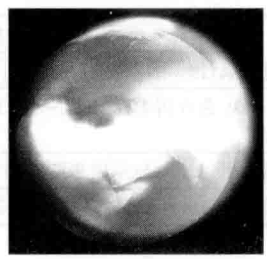
在左上角你所看到的效果称作泛光效果（bloom effect）。当你应用泛光效果时，场景中的明亮区域将会变得更加显著，而且会渗入到较暗的区域。创建 BloomPass 的代码如下所示：

```
var bloomPass = new THREE.BloomPass(3, 25, 5, 256);
var composer3 = new THREE.EffectComposer(webGLRenderer);
composer3.addPass(renderScene);
composer3.addPass(bloomPass);
composer3.addPass(effectCopy);
```

如果将这段代码与使用 FilmPass 的 EffectComposer 相比，你会发现我们在这里多添加了一个通道——effectCopy。这一步我们在普通输出中也曾用过，它不会添加任何特殊效果，只是将最后一个通道的结果复制到屏幕上。之所以要添加这一步，是因为 BloomPass 不能直接将渲染结果输出到屏幕。下表列出的是 BloomPass 的所有可以设置的属性：

属　性	描　述
Strength	该属性定义的是泛光效果的强度。其值越高，则明亮的区域越明亮，而且渗入较暗区域的也就越多
kernelSize	该属性控制的是泛光效果的偏移量
sigma	通过 sigma 属性，你可以控制泛光效果的锐利程度。其值越高，泛光越模糊
Resolution	该属性定义的是泛光效果的解析图。如果该值太低，那么结果的方块化会比较严重

理解这些属性最好的办法就是在前面提到的那个示例（02-post-processing-simple-passes.html）里试验一下。你可以在下面的截图里看到泛光效果的样子。该示例中使用了较高的 kernelSize、sigma 和较低的 Strength。



我们最后要看的简单通道是 DotScreenPass。

11.2.1.3　使用 DotScreenPass 将场景输出成点集

DotScreenPass 的使用跟我们刚刚看到的 BloomPass 非常相似。参考如下代码：

```
var dotScreenPass = new THREE.DotScreenPass();
var composer1 = new THREE.EffectComposer(webGLRenderer);
composer1.addPass(renderScene);
composer1.addPass(dotScreenPass);
composer1.addPass(effectCopy);
```

要到达该效果，我们仍然需要添加 effectCopy，以便将结果输出到屏幕。在 DotScreenPass 中可以配置如下属性：

属　性	描　述
center	通过 center 属性，你可以微调点的偏移量
angle	这些点是按照某种方式对齐的。通过 angle 属性，你可以更改对齐方式
scale	该属性设置所用点的大小。scale 越小，则点越大

跟其他着色器一样，要获取合适的配置最好先试验一下。在下面的截图中你所看到的是使用了高 scale 值的 DotScreenPass：

11.2.1.4 将多个渲染器的输出结果显示在同一屏幕上

本节不涉及如何使用后期处理效果的细节，而是解释如何将所有四个 EffectComposer 实例的输出结果显示在同一屏幕上。首先让我们来看看本例所用的 render 循环：

```
function render() {
    stats.update();

    var delta = clock.getDelta();
    orbitControls.update(delta);

    sphere.rotation.y += 0.002;

    requestAnimationFrame(render);

    webGLRenderer.autoClear = false;
    webGLRenderer.clear();

    webGLRenderer.setViewport(0, 0,
                2 * halfWidth, 2 * halfHeight);
    composer.render(delta);

    webGLRenderer.setViewport(0, 0,
                halfWidth, halfHeight);
    composer1.render(delta);

    webGLRenderer.setViewport(halfWidth, 0,
                halfWidth, halfHeight);
    composer2.render(delta);

    webGLRenderer.setViewport(0, halfHeight,
                halfWidth, halfHeight);
    composer3.render(delta);

    webGLRenderer.setViewport(halfWidth, halfHeight,
                halfWidth, halfHeight);
    composer4.render(delta);
}
```

这里首先要注意的是我们将 webGLRenderer.autoClear 属性设为了 false，而且我们将明确调用 clear() 函数。如果不这么做，那么每次调用效果组合器的 render() 函数时，之前渲染的场景会被清掉。通过这种方法，我们只会在 render 循环开始时将所有东西清一遍。

为了避免所有效果组合器在同一地方渲染，我们将效果组合器所用的 webGLRenderer 的视图区设置成屏幕上不同的部分。设置视图区的函数接受四个参数：x、y、width 和 height。正如你在代码中所看到的，我们用这个函数将屏幕分成四个区，每个效果组合器在各自的区域中渲染。

如果你愿意，你也可以将该方法用于多场景、多相机和多 WebGLRenderer 实例的情况下。

到目前为止，我们只是将多个简单通道串联起来。在下一个示例里，我们将会配置一个复杂的 EffectComposer，并用掩膜在部分窗口上应用效果。

11.2.2　使用掩膜的高级效果组合器

在前面的示例里，我们是在整个场景上应用后期处理通道。但是 Three.js 库也具有在特定区域上应用通道的能力。本节我们将会采取如下步骤：

1）创建一个作为背景图的场景。

2）创建一个场景，里面有一个看上去像地球的球体。

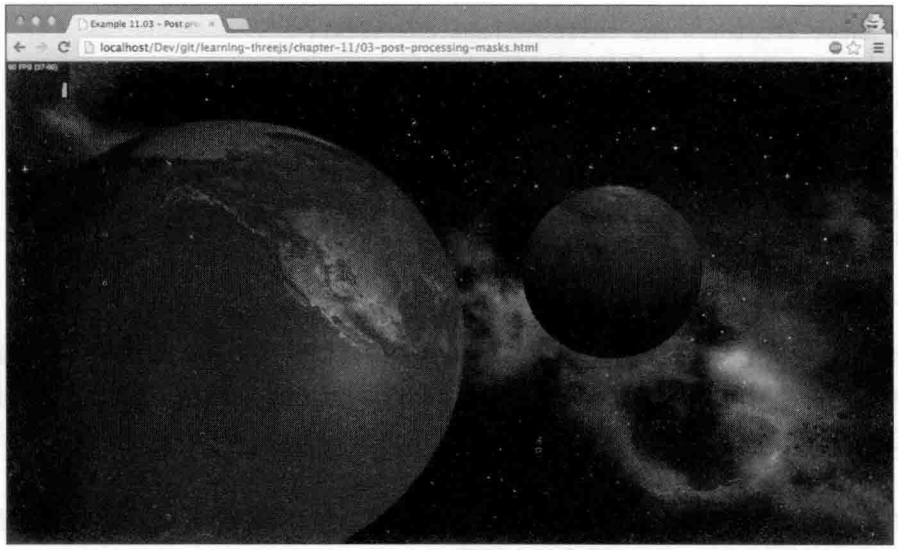
3）创建一个场景，里面有一个看上去像火星的球体。

4）创建一个 EffectComposer 对象，将这三个场景渲染到一个图片里。

5）在渲染成火星的球体上应用一个彩色效果。

6）在渲染成地球的球体上应用褐色效果。

这听上去好像有点儿复杂，但是完成起来却相当简单。先看一下我们要达到的目标，如下图所示。可以参考示例：03-post-processing-masks.html：



首先我们要配置各个需要渲染的场景，如下所示：

```
var sceneEarth = new THREE.Scene();
var sceneMars = new THREE.Scene();
var sceneBG = new THREE.Scene();
```

要创建代表地球和火星的球体，我们只要创建好带有相应材质和纹理的球体，然后将它

们添加到对应的场景中。代码如下所示：

```
var sphere = createEarthMesh(
                        new THREE.SphereGeometry(10, 40, 40));
sphere.position.x = -10;
var sphere2 = createMarshMesh(
                        new THREE.SphereGeometry(5, 40, 40));
sphere2.position.x = 10;
sceneEarth.add(sphere);
sceneMars.add(sphere2);
```

就像一个普通场景一样，我们还要给它添加一些光源，但我们不在这里讨论。唯一要记住的是同一光源不能添加到不同场景中，所以你要为每个场景创建各自的光源。这些就是这两个场景要做的所有配置。

至于背景图，我们要创建一个 OrthographicCamera 对象。根据第 2 章中的内容，对象的正交投影尺寸不受相机距离的影响。

```
var cameraBG = new THREE.OrthographicCamera(
          -window.innerWidth,
           window.innerWidth,
           window.innerHeight,
          -window.innerHeight, -10000, 10000);
cameraBG.position.z = 50;

var materialColor = new THREE.MeshBasicMaterial(
    { map: THREE.ImageUtils.loadTexture(
       "../assets/textures/starry-deep-outer-space-galaxy.jpg"),
       depthTest: false });
var bgPlane = new THREE.Mesh(new THREE.PlaneGeometry(1, 1),
                             materialColor);
bgPlane.position.z = -100;
bgPlane.scale.set(window.innerWidth * 2,
                  window.innerHeight * 2, 1);
sceneBG.add(bgPlane);
```

这部分我们不会解释太多细节，但是创建背景图我们要采取几个步骤。首先，我们要从背景图中创建一个材质，并应用到简单的 plane（平面）上。然后将这个 plane 添加到场景中，并对它进行缩放以适应整个屏幕。所以当我们用这个相机渲染场景时，背景图会布满整个窗口。

现在我们有了三个场景，可以开始配置后期处理通道和 EffectComposer 对象。我们会从整个后期处理通道链开始，然后看看每个通道：

```
var composer = new THREE.EffectComposer(webGLRenderer);
composer.renderTarget1.stencilBuffer = true;
composer.renderTarget2.stencilBuffer = true;

composer.addPass(bgPass);
composer.addPass(renderPass);
```

```
composer.addPass(renderPass2);

composer.addPass(marsMask);
composer.addPass(effectColorify1);
composer.addPass(clearMask);

composer.addPass(earthMask);
composer.addPass(effectSepia);
composer.addPass(clearMask);

composer.addPass(effectCopy);
```

要使用掩膜，我们需要用不同的方式创建 EffectComposer 对象。在本例中，我们要创建一个新的 WebGLRenderTarget 对象，然后将这个内部使用的渲染对象的 stencilBuffer 属性设成 true。让我们来看一下先添加的三个通道。这三个通道会渲染背景场景、地球场景和火星场景，如下所示：

```
var bgPass = new THREE.RenderPass(sceneBG, cameraBG);
var renderPass = new THREE.RenderPass(sceneEarth, camera);
renderPass.clear = false;
var renderPass2 = new THREE.RenderPass(sceneMars, camera);
renderPass2.clear = false;
```

这里并没有什么新东西，除了将两个通道的 clear 属性设为 false。如果不这么做，我们将只会看到 renderPass2 的输出，因为它会在开始渲染时清除所有东西。如果你回头看看 EffectComposer 的代码，你会发现接下来的三个通道是 marsMask、effectColorify 和 clearMask。首先我们要看看这三个通道是怎么定义的：

```
var marsMask = new THREE.MaskPass(sceneMars, camera );
var clearMask = new THREE.ClearMaskPass();
var effectColorify = new THREE.ShaderPass(THREE.ColorifyShader );
effectColorify.uniforms['color'].value.setRGB(0.5, 0.5, 1);
```

这三个通道中的第一个是 MaskPass。创建 MaskPass 时，跟创建 RenderPass 一样你要传入场景和相机。MaskPass 会在内部渲染场景，但它不是将结果显示在屏幕上，而是用它来创建掩膜。如果将 MaskPass 添加到一个 EffectComposer 对象上，那么后续所有通道将只会应用到 MaskPass 定义的掩膜上，直到遇到 ClearMaskPass。这也就意味着在本例中添加蓝色光芒的 effectColorify 通道只会应用到 sceneMars 中渲染的对象上。

我们也会在表示地球的那个对象上用同样的方法应用褐色滤镜。我们先基于地球场景创建一个掩膜，然后在 EffectComposer 中使用该掩膜。添加完 MaskPass 后我们再添加其他想要的效果（本例中是 effectSepia），然后添加 ClearMaskPass 移除掩膜。这个 EffectComposer 的最后一步我们曾经见过。我们需要将最终结果复制到屏幕，同样，我们用的是 effectCopy 通道。

在 MaskPass 中有一个有趣的属性，即 inverse 属性。如果将该属性设为 true，则掩膜会

反转。也就是说，该效果会应用到所有对象上，除了传递给 MaskPass 的场景。

到目前为止我们用的一直是 Three.js 提供的标准通道。Three.js 还提供了一个 ShaderPass（着色器通道），该通道可以用来定制效果，而且有大量的着色器可用或者试验。

11.2.3 用 ShaderPass 定制效果

通过 ShaderPass，我们可以传递一个自定义的着色器，将大量额外的效果应用到场景中。本节将分为三个部分。我们先来看一看下面所列的简单着色器：

名　称	描　述
MirrorShader	该着色器可以为部分屏幕创建镜面效果
HueStaturationShader	该着色器可以改变颜色的色调和饱和度
VignetteShader	该着色器可以添加晕映效果。该效果可以在图片中央的周围显示黑色的边框
ColorCorrectionShader	通过这个着色器，你可以调整颜色的分布
RGBShiftShader	该着色器可以将构成颜色的红、绿、蓝分开
BrightnessContrasShader	该着色器可以更改图片的亮度和对比度
ColorifyShader	该着色器可以在屏幕上蒙上一层颜色
SepiaShader	该着色器可以在屏幕上创建出类似乌贼墨的效果

接下来，我们将会看一些提供模糊效果的着色器：

名　称	描　述
HorizontalBlurShader 和 VerticalBlurShader	这两个着色器在整个场景中应用模糊效果
HorizontalTiltShiftShader 和 VerticalTiltShiftShader	这两个着色器可以创建出移轴效果。在移轴效果中只有部分图片显示得比较锐利，从而创建出一个看上去像是微缩景观的场景
TriangleBlurShader	该着色器使用基于三角形的方法，在场景中应用模糊效果

最后，我们会看几个提供高级效果的着色器：

名　称	描　述
BleachBypassShader	该着色器可以创建一种漂白效果。在该效果下，图片上像是镀了一层银
EdgeShader	该着色器可以探测图片中锐利的边界，并突出显示这些边界
FXAAShader	该着色器可以在后期处理阶段应用抗锯齿效果。如果在渲染时抗锯齿影响效率，那么就可以使用该着色器
FocusShader	这是一个简单的着色器，其结果是中央区域渲染得比较锐利，但周围比较模糊

我们不会解释所有这些着色器的细节，你只要知道其中一个是如何工作的，那么其他的你也就了解了。下一节，我们会着重讲几个比较有意思的。你可以用各小节提供的交互式例子试验一下其他的着色器。

我们将从简单的着色器开始。

11.2.3.1　简单的着色器

对于那些基础的着色器，我们创建了一个示例，你可以在这个示例中尝试一下这些着色器，直接在场景中看到它们的效果。你可以参考示例 04-shaderpass-simple.html，如下图所示：

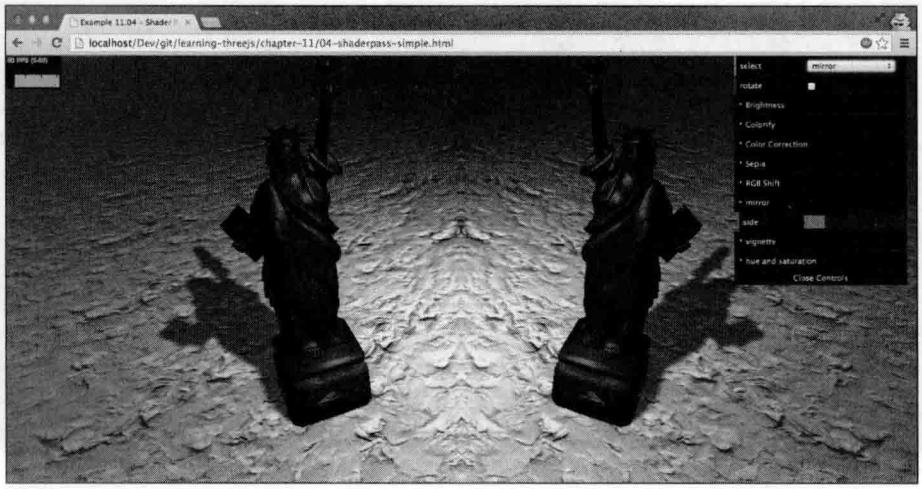
通过右上角的菜单，你可以选择想要应用的效果。在各个下拉菜单中，你还可以设置所选着色器的属性。例如，下图所示的就是 RGBShiftShader 效果：

当你更改着色器的某个属性时，场景会立即更新。在这个示例里，我们直接将修改后的值设置到着色器上。例如，当 RGBShiftShader 的值改变时，我们会更新相应的着色器，代码如下所示：

```
this.changeRGBShifter = function() {
  rgbShift.uniforms.amount.value = controls.rgbAmount;
  rgbShift.uniforms.angle.value = controls.angle;
}
```

我们来看看其他着色器。下图显示的是 VignetteShader 的效果：

MirrorShader 的效果如下图所示：



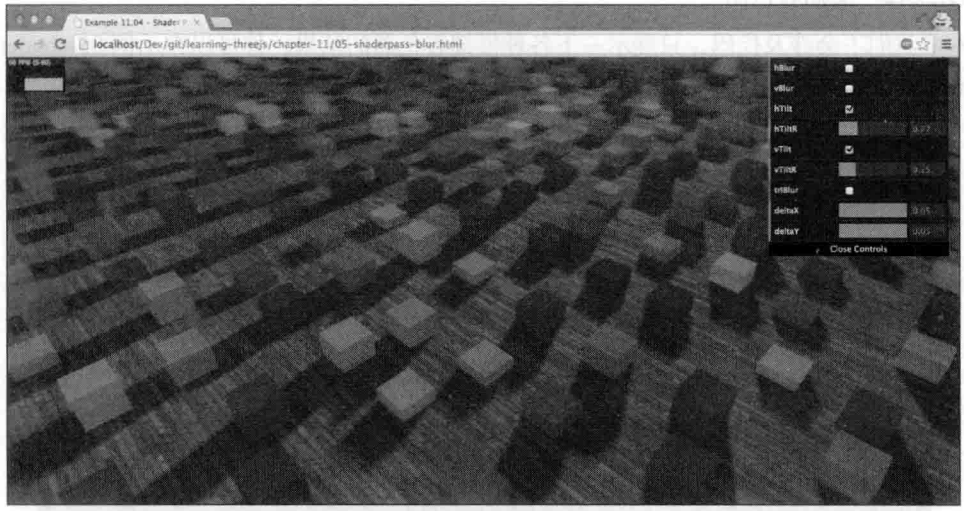
这对了解简单着色器来讲已经足够了。正如你所看到的，它们的功能很多，可以创建出非常有趣的效果。在这个示例里，我们每次应用一个着色器，但是你想在 EffectComposer 里添加多少 ShaderPass 步骤，就可以添加多少。

11.2.3.2　模糊着色器

本节我们不会涉及代码，只展示一下各种模糊着色器的结果。你可以在示例 05-shaderpass-blur.html 中试验这些着色器，如下图所示：

前面这个屏幕截图展示的是 HorizontalBlurShader 和 VerticalBlurShader 的效果。你可以看到它们的渲染结果是一个模糊的场景。除了这两个模糊着色器之外，Three.js 还提供一个额外的着色器（TriangleShader）可以让图片变得模糊，其结果如下图所示：

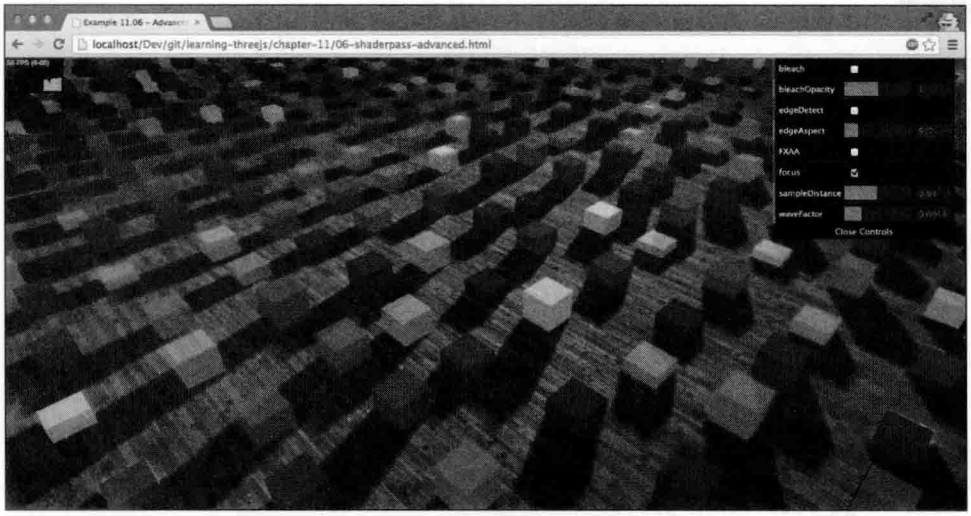
最后一个模糊效果是由 HorizontalTiltShiftShader 和 VerticalTiltShiftShader 提供的。这两个着色器不是模糊整个场景，而只是一个小区域。它们可以提供一种称作移轴的效果。通常可以用它们在一张普通照片上创建出一种类似微缩景观的场景。下图展示的就是这种效果：



11.2.3.3 高级着色器

对于高级着色器，我们也会像前面的模糊着色器那样处理。我们只会展示这些着色器的输出结果。至于这些着色器的细节，可以参考示例 06-shaderpass-advanced.html，如下图所示：

上图展示的是 EdgeShader 的效果。通过该着色器你可以探测到场景中对象的边缘。在下面的屏幕截图中你会看到 FocusShader 的效果，该着色器在渲染时只会聚焦于屏幕的中央：



到目前为止我们使用的都只是 Three.js 库提供的着色器。但是，你自己也可以很容易地创建着色器。

11.3　创建自定义的后期处理着色器

本节你将会学习如何创建一个可以在后期处理中使用的、自定义的着色器。我们会创建两个着色器。第一个着色器可以将当前图片转换成灰度图，第二个着色器可以通过减少可用

颜色的数目，将图片转换成 8 位图。

创建顶点着色器和片段着色器是一个非常宽泛的话题。我们在本节只会涉及这些着色器的功能及其工作原理的比较浅显的内容。更多信息你可以在 WebGL 规范（http://www.khronos.org/webgl/）中找到。另外一个比较好的、而且带有示例的资源是 shadertoy，网址为 https://www.shadertoy.com/。

11.3.1 定制灰度图着色器

为 Three.js 库（以及其他的 WebGL 库）创建自定义的着色器，需要实现两个组件：vertexShader 和 fragmentShader。组件 vertexShader 可以用来调整每个顶点的位置，组件 fragmentShader 可以用来决定每个像素的颜色。对于后期处理着色器来说，我们只要实现 fragmentShader 即可，然后使用 Three.js 库提供的、默认的 vertexShader。在看代码之前，很重要的一点是 GPU 通常会支持多个着色器管道。这也就是说在 GPU 中执行 vertexShader 时会有多个着色器在同时执行，这对 fragmentShader 步骤来说也是一样。

现在我们来看看这个着色器的完整代码，该着色器会在图片（custom-shader.js）上应用灰度图效果：

```
THREE.CustomGrayScaleShader = {

    uniforms: {

        "tDiffuse": { type: "t", value: null },
        "rPower":   { type: "f", value: 0.2126 },
        "gPower":   { type: "f", value: 0.7152 },
        "bPower":   { type: "f", value: 0.0722 }

    },

    vertexShader: [
        "varying vec2 vUv;",
        "void main() {",
        "vUv = uv;",
        "gl_Position = projectionMatrix * modelViewMatrix
                       * vec4( position, 1.0 );",
        "}"
    ].join("\n"),

    fragmentShader: [

        "uniform float rPower;",
        "uniform float gPower;",
        "uniform float bPower;",
        "uniform sampler2D tDiffuse;",
```

```
        "varying vec2 vUv;",

        "void main() {",
        "vec4 texel = texture2D( tDiffuse, vUv );",
        "float gray = texel.r*rPower +
                      texel.g*gPower + texel.b*bPower;",
        "gl_FragColor = vec4( vec3(gray), texel.w );",
        "}"
    ].join("\n")
};
```

正如你所看到的，这些代码并不是用 JavaScript 编写的。开发一个着色器需要使用 OpenGL Shading Language（GLSL，OpenGL 着色语言），它的语法看上去很像 C 语言。有关 GLSL 的更多信息可以参考：http://www.khronos.org/opengles/sdk/docs/manglsl/。

我们先来看看这个顶点着色器：

```
"varying vec2 vUv;",
        "void main() {",
        "vUv = uv;",
        "gl_Position = projectionMatrix * modelViewMatrix
                      * vec4( position, 1.0 );",
        "}"
```

对于后期处理来说，这个着色器其实并没有做什么。这里给出的代码是 Three.js 库实现 vertexShader 的标准方法。代码中用到的 projectionMatrix 是相机的投影矩阵，modelViewMatrix 将对象每个顶点映射到屏幕坐标，用来决定在屏幕的什么地方渲染该对象。

对于后期处理，这段代码中唯一有趣的是 uv 值，它表示的是纹理上的 Texel（纹理上的像素），通过 varying vec2 vUv 变量传递给 fragmentShader。我们会在 fragmentShader 中使用这个 vUv 值获取正确的像素。现在让我们来看看这个 fragmentShader 及其代码的功能。我们将从变量的声明开始：

```
"uniform float rPower;",
"uniform float gPower;",
"uniform float bPower;",
"uniform sampler2D tDiffuse;",

"varying vec2 vUv;",
```

在这里可以看到四个 uniforms 变量。uniforms 变量中的值可以从 JavaScript 传递给着色器。在本例中我们会传递三个浮点数，类型标识为 f（用来决定最终的灰度图中所包含的颜色分量大小），我们还会传递一个纹理（tDiffuse），类型标识为 t。该纹理中包含的是 EffectComposer 中前一个通道的渲染结果。Three.js 可以保证这个渲染结果能够正确地传递给这个着色器。其他的 uniforms 变量我们可以在 JavaScript 中设置。在 JavaScript 中使用这些

uniforms 变量之前，我们必须要指定着色器可以使用哪些 uniforms 变量。而这一步是在着色器文件的开头完成的，如下代码所示：

```
uniforms: {

    "tDiffuse": { type: "t", value: null },
    "rPower":   { type: "f", value: 0.2126 },
    "gPower":   { type: "f", value: 0.7152 },
    "bPower":   { type: "f", value: 0.0722 }

},
```

现在我们就可以接收 Three.js 库传来的配置参数，以及需要调整的图片。来看看将每个像素变成灰色的代码：

```
"void main() {",
"vec4 texel = texture2D( tDiffuse, vUv );",
"float gray = texel.r*rPower +
              texel.g*gPower + texel.b*bPower;",
"gl_FragColor = vec4( vec3(gray), texel.w );"
```

这段代码的作用是从传递进来的纹理上获取正确的像素。方法是调用 texture2D() 函数，传入当前图片（tDiffuse）和要处理的像素的位置（vUv）。其结果是一个包含颜色和透明度（texel.w）的 Texel。

接下来我们将会用这个 Texel 属性 r、g 和 b 的值来计算灰度值。这个灰度值保存在 gl_FragColor 变量中，最终显示在屏幕上。这样我们就完成了自定义的着色器。这个着色器的用法跟其他着色器一样。首先我们要设置 EffectComposer，如下所示：

```
var renderPass = new THREE.RenderPass(scene, camera);

var effectCopy = new THREE.ShaderPass(THREE.CopyShader);
effectCopy.renderToScreen = true;

var shaderPass = new THREE.ShaderPass(THREE.CustomGrayScaleShader);

var composer = new THREE.EffectComposer(webGLRenderer);
composer.addPass(renderPass);
composer.addPass(shaderPass);
composer.addPass(effectCopy);
```

然后在 render 循环中调用 composer.render（delta）函数。如果想在运行时更改着色器的属性，可以直接修改我们定义的 uniforms 变量，如下所示：

```
shaderPass.enabled = controls.grayScale;
shaderPass.uniforms.rPower.value = controls.rPower;
shaderPass.uniforms.gPower.value = controls.gPower;
shaderPass.uniforms.bPower.value = controls.bPower;
```

结果如下图所示（示例 07-shaderpass-custom.html）：

现在我们再来创建一个自定义的着色器。这次我们要把输出结果由 32 位降低到 8 位。

11.3.2　定制位着色器

一般来讲颜色可以表示为 24 位的数值，大约 1600 万种颜色。而在计算机发展的早期，这是不可能的，颜色一般是用 8 位的数值或者 16 位的数值来表示的。通过这个着色器，我们可以自动将 24 位的输出转换成颜色深度为 8（或者任何你想要的深度）的结果。

我们将略过 vertexShader 不谈，因为它跟之前的例子没有太大差别。我们直接列出各个 uniforms 变量和 fragmentShader，代码如下所示：

```
uniforms: {

        "tDiffuse": { type: "t", value: null },
        "bitSize":  { type: "i", value: 4 }

    }
```

fragmentShader 定义如下所示：

```
fragmentShader: [

    "uniform int bitSize;",

    "uniform sampler2D tDiffuse;",

    "varying vec2 vUv;",

    "void main() {",
```

```
    "vec4 texel = texture2D( tDiffuse, vUv );",
    "float n = pow(float(bitSize),2.0);",
    "float newR = floor(texel.r*n)/n;",
    "float newG = floor(texel.g*n)/n;",
    "float newB = floor(texel.b*n)/n;",

    "gl_FragColor = vec4(newR, newG, newB, texel.w );",

    "}"

].join("\n")
```

我们会定义两个用来配置这个着色器的 uniforms 变量。Three.js 库使用第一个 uniforms 变量，传递当前屏幕上的渲染结果；第二个 uniforms 变量是由我们定义的，类型是整数（type：“i”），用来表示渲染结果的颜色深度。这段代码本身非常简单明了。流程如下：

1）先根据传入的 vUv 值，从纹理（tDiffuse）中获取一个 texel。

2）然后根据 bitSize 属性计算可能得到的颜色的数量，方法是取 2 的 bitSize 次方（pow(float(bitSize)，2.0））。

3）接下来要计算 texel 颜色的新值，方法是将原颜色值乘以 n，然后以去尾法圆整（floor(texel.r*n)）再除以 n。

4）上述计算结果保存在 gl_FragColor（红、绿、蓝分量的值，以及透明度）中，然后显示在屏幕上。

你可以在上一个自定义着色器的例子（07-shaderpass-custom.html）里看到这个着色器的渲染结果：

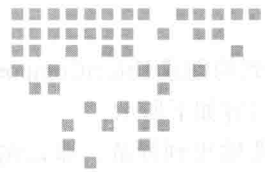
这就是后期处理的所有内容了。

11.4　总结

本章我们讨论了很多后期处理的方法。正如你所看到的创建 EffectComposer 对象，然后将各种通道串联在一起非常简单。本章主要应该记住的内容如下所示：

- 不是所有通道的结果都会输出到屏幕。如果想要输出到屏幕，那么你可以使用带有参数 CopyShader 的 ShaderPass。
- 在效果组合器中添加通道的顺序很重要。各种效果按照添加顺序起作用。
- 如果你想重用某个 EffectComposer 的结果，你可以使用 TexturePass。
- 如果在 EffectComposer 中有多个 RenderPass，那么你需要将 clear 属性设为 false。如果不这么做，那么你只会看到最后一个 RenderPass 的输出。
- 如果你只想在特定对象上应用某种效果，你可以使用 MaskPass。用完后，可以调用 ClearMaskPass 来清除掩膜。
- Three.js 中除了提供很多标准的通道之外，还提供了大量的标准着色器。你可以通过 ShaderPass 使用它们。
- 在 Three.js 库中使用标准方法，为后期处理创建自定义的着色器非常简单，你只需要创建一个 fragmentShader。

到目前为止，我们差不多已经涵盖了 Three.js 中所有应该知道的事情。下一章，也就是最后一章，我们将会介绍一个名为 Physijs 的库。你可以用物理效果来扩展 Three.js，例如碰撞、重力和约束。



Chapter 12 第 12 章

用 Physijs 在场景中添加物理效果

在最后这一章里，我们要看看另一个可以用来扩展 Three.js 功能的库。本章所要讨论的这个库叫做 Physijs。通过 Physijs 库可以在场景中引入物理效果。所谓物理效果指的是对象会有重力，它们可以相互碰撞，施加力之后可以移动，而且通过铰链和滑块还可以在移动过程中在对象上施加约束。这个库利用的是另外一个著名的物理引擎，ammo.js。

本章我们将会看一看 Physijs 的功能：

- 创建一个 Physijs 场景，在这个场景中对象具有重力，而且可以相互碰撞。
- 展示一下如何修改场景中对象的摩擦系数和复原性（弹性）。
- 讲解 Physijs 支持的各种图形，以及如何使用它们。
- 如何通过将简单图形联结在一起，创建组合图形。
- 展示一下如何通过 height field（高度场）模拟复杂图形。
- 通过添加点、铰链、螺钉和自由度约束来限制对象的移动。

我们首先要做的是创建一个可以使用 Physijs 的 Three.js 场景。这将在第一个示例中完成。

12.1 创建可用 Physijs 的基本 Three.js 场景

创建一个可用 Physijs 的 Three.js 场景非常简单，只要几个步骤即可。首先我们要包含正确的文件，你可以从 GitHub 代码仓库上获取，网址是：http://chandlerprall.github.io/Physijs/。

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

模拟这样的场景非常耗费 CPU。如果我们在 render 线程中做的话，场景的帧频会受到严重的影响。为了弥补这一点，Physijs 选择在后台线程中执行计算。这里的后台线程是由 Web workers（网页线程）规范定义的，现在大多数浏览器都实现了该功能。根据这个规范，你可以在一个单独的线程里执行 CPU 密集的任务，这样就不会影响渲染。有关 Web workers 的更多信息可以参考网址：http://www.whatwg.org/specs/web-apps/current-work/multipage/workders.html。

对 Physijs 来说这也就意味着我们必须要配置一个带有执行任务的 JavaScript 文件，并告诉 Physijs 在哪里可以找到用来模拟场景的 ammo.js 文件。之所以要包含 ammo.js 文件，原因是 Physijs 只是 ammo.js 的一个包装器，使得它更容易使用。ammo.js（你可以在网址 https://github.com/kripken/ammo.js 找到它）是一个实现物理引擎的库，Physijs 只是在这个物理库的基础上提供了便于使用的接口。由于 Physijs 只是一个包装器，所以我们在 Physijs 中也可以使用其他的物理引擎。在 Physijs 的代码仓库中你还可以找到一个使用 Cannon.js（另一个物理引擎）的分支版本。

我们的做法是设置下面这两个属性：

```
Physijs.scripts.worker = '../libs/physijs_worker.js';
Physijs.scripts.ammo = '../libs/ammo.js';
```

第一个属性指向我们想要执行的任务线程，第二个属性指向内部使用的 ammo.js 库。接下来我们要创建一个场景。Physijs 在 Three.js 的普通场景外又提供了一个包装器，所以在代码中你可以像这样创建场景：

```
var scene = new Physijs.Scene();
scene.setGravity(new THREE.Vector3(0, -10, 0));
```

这样就可以创建出一个应用了物理效果的场景，而且我们还设置了重力。在本例中，我们将重力设置在 y 轴方向上，值为 -10。也就是说，场景中的对象可以竖直下落。你可以在运行时，在各坐标轴方向上将重力设置或修改成你认为合适的值，然后场景就会做出相应的反应。

在开始模拟物理效果之前，我们需要在场景中添加一些对象。为此，我们可以用 Three.js 中的普通方法来定义对象，但必须用一个特定的 Physijs 对象将这些对象包裹起来。

```
var stoneGeom = new THREE.CubeGeometry(0.6,6,2);
var stone = new Physijs.BoxMesh(stoneGeom,
  new THREE.MeshPhongMaterial({color: 0xff0000}));
scene.add(stone);
```

在这个例子里我们创建了一个简单的 CubeGeometry，但接着我们创建了一个 Physijs.BoxMesh 对象，而不是 THREE.Mesh 对象。BoxMesh 可以告诉 Physijs 在模拟和检测碰撞时，将这个网格当做一个盒子。Physijs 提供了很多网格，你可以将它们用于各种图形上。本章稍后有更详细的讲解。

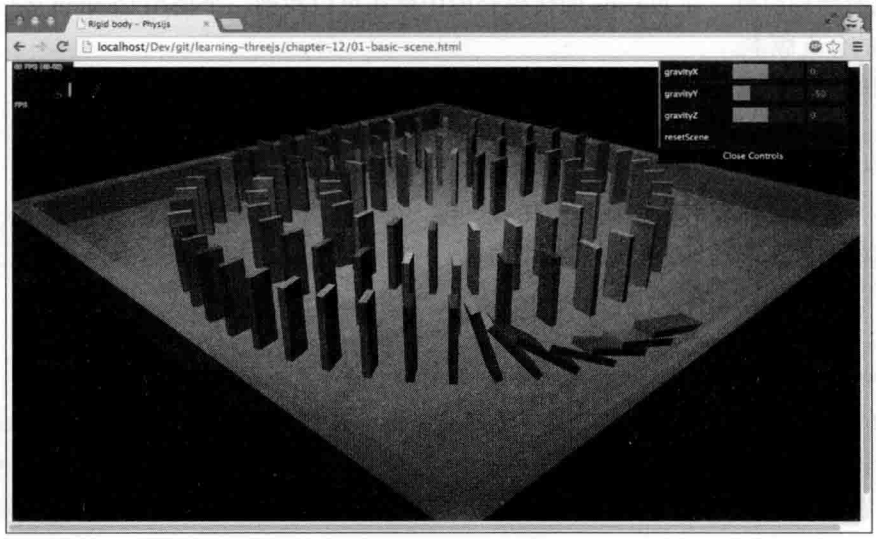
现在 BoxMesh 对象已经添加到了场景中，我们第一个 Physijs 场景中的各个部分都有

了。剩下要做的就是告诉 Physijs 模拟物理效果，并更新场景中各对象的位置和角度。为此，我们可以调用刚创建的场景的 simulate 方法。修改基础 render 循环的代码如下所示：

```
render = function() {
  requestAnimationFrame(render);
  renderer.render(scene, camera);
  scene.simulate();
}
```

随着最后一步的完成，用于 Physijs 场景的基础配置也就完成了。但是运行这个例子，我们看不到多少不同。我们只会看到在场景渲染时，屏幕中央有个方块正在下落。所以让我们来看一个复杂一点儿的例子，模拟正在倒下的多米诺骨牌。

我们将在这个例子中创建如下的场景：



如果在浏览器中打开示例 01-basic-scene.html，在场景加载完之后你就会看到有一组多米诺骨牌正在倒下。第一块儿骨牌会碰倒第二块儿骨牌，依次类推。这个场景中的物理效果都是由 Physijs 负责的。要让它们动起来，我们唯一要做的就是推倒第一块儿多米诺骨牌。创建这样一个场景非常简单，只要采取下面几步：

1）定义 Physijs 场景。

2）定义放置多米诺骨牌的地面。

3）放置骨牌。

4）推倒第一块儿骨牌。

因为我们已经知道怎么做了，所以略过第一步，直接看第二步，定义地面。表示地面（ground）的图形是由几个长方体组合起来的：

```
function createGround() {
```

```
var ground_material = Physijs.createMaterial(
new THREE.MeshPhongMaterial({ map: THREE.ImageUtils.loadTexture(
  '../assets/textures/general/wood-2.jpg' )}),0.9,0.3);

var ground = new Physijs.BoxMesh(
  new THREE.CubeGeometry(60, 1, 60), ground_material, 0);

var borderLeft = new Physijs.BoxMesh(
  new THREE.CubeGeometry(2, 3, 60), ground_material, 0);
borderLeft.position.x=-31;
borderLeft.position.y=2;
ground.add(borderLeft);

var borderRight = new Physijs.BoxMesh(
  new THREE.CubeGeometry(2, 3, 60), ground_material, 0);
borderRight.position.x=31;
borderRight.position.y=2;
ground.add(borderRight);

var borderBottom = new Physijs.BoxMesh(
  new THREE.CubeGeometry(64, 3, 2), ground_material, 0);
borderBottom.position.z=30;
borderBottom.position.y=2;
ground.add(borderBottom);

var borderTop = new Physijs.BoxMesh(
  new THREE.CubeGeometry(64, 3, 2), ground_material, 0);
borderTop.position.z=-30;
borderTop.position.y=2;
ground.add(borderTop);

scene.add(ground);
}
```

这段代码并不复杂。首先我们创建了一个作为地面的长方体，然后添加几条边界，防止对象掉到地面外。我们将这些边界添加到 ground（地面）对象上，构建出一个复合对象。Pyhsijs 将复合对象当做一个对象。更多关于复合对象的内容将在本章后面进一步讲述。在这段代码中还有些新东西，我们将在本章后面的小节中进一步解释。首先是我们创建的 ground_material。我们调用 Physijs.createMaterial 函数创建这个材质。这个函数是对标准 Three.js 材质的包装，但是我们可以在上面设置属性 friction（摩擦系数）和 restitution（弹性系数）。关于这个主题下一节会有进一步的介绍。另外一个新东西是 Physijs.BoxMesh 构造函数的最后一个参数。我们创建的所有 BoxMesh 对象，其最后一个参数都是 0。通过这个参数我们可以设置对象的重力。这么做可以避免地面受场景中的重力影响。

现在我们有了地面，可以放置多米诺骨牌了。为此我们创建了一些简单的 Three.CubeGeometry 对象，将它们包装在 BoxMesh 对象中，然后放置在地面网格的各个指定点上。

```
var stoneGeom = new THREE.CubeGeometry(0.6,6,2);
var stone = new Physijs.BoxMesh(stoneGeom, Physijs.createMaterial(new
THREE.MeshPhongMaterial(
```

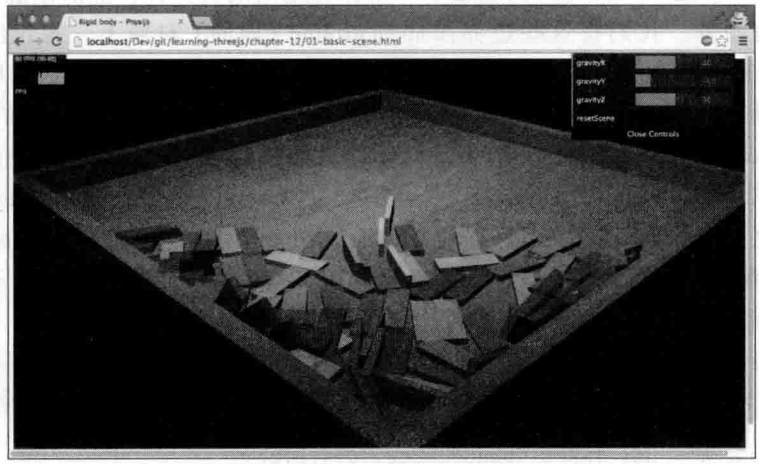
```
    {
      color: scale(Math.random()).hex(),
      transparent:true, opacity:0.8
    })));
  stone.position=point.clone();
  stone.lookAt(scene.position);
  stone.__dirtyRotation = true;
  stone.position.y=3.5;
  scene.add(stone);
```

我们并没有把计算每块多米诺骨牌位置的代码列出来（可以参考该实例的源代码），这段代码展示的只是如何放置多米诺骨牌。这里你所看到的是我们又创建了一个包装 CubeGeometry 对象的 BoxMesh 对象。为了保证这些多米诺骨牌能够正确地对齐，我们调用了 lookAt 函数来设置它们的角度。如果不这么做，那它们都将朝向同一方向，因此也就不会倒下。在手工更新了 Physijs 包装的对象的角度（或位置）之后，我们必须要告诉 Physijs 有什么东西发生了改变。对于角度，我们可以将 __dirtyRotation 属性设成 true；对于位置，我们可以将 __dirtyPosition 属性设成 true。

现在剩下要做的就是推到第一块儿多米诺骨牌。要推到它，我们只要将其绕 x 轴的角度设置成 0.2，也就是稍稍推一下。场景中的重力可以完成剩下的工作，并将第一块儿多米诺骨牌完全推翻。

```
stones[0].rotation.x=0.2;
stones[0].__dirtyRotation = true;
```

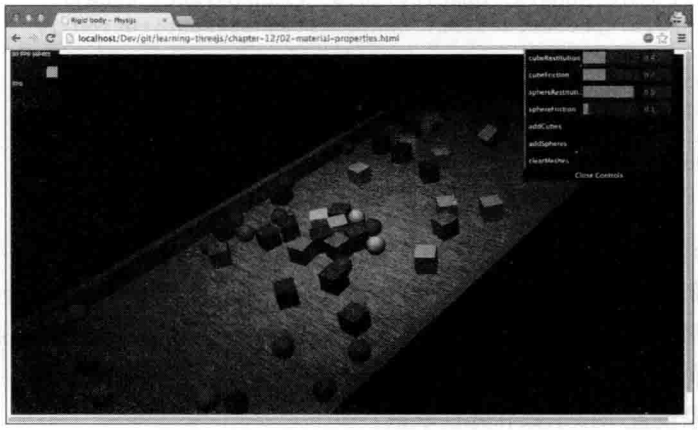
这样我们的第一个例子就完成了，而且也展示了 Physijs 中的很多功能。如果你想试验一下重力，你可以通过场景右上角的菜单来修改它。修改好的重力值在你点击 resetScene 按钮时才起作用。



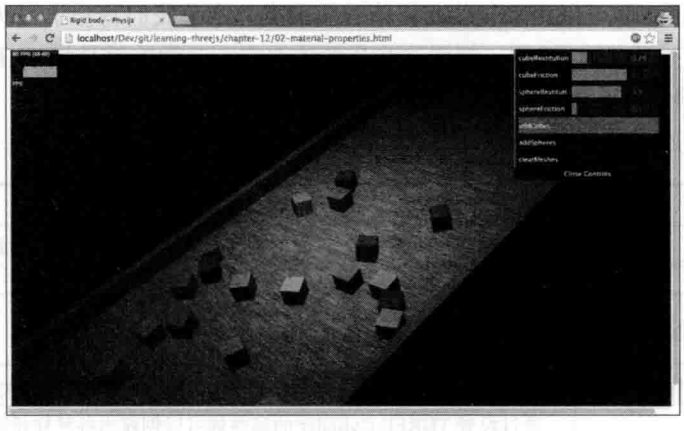
下一节我们将进一步学习 Physijs 材质的属性是如何影响对象的。

12.2　材质属性

让我们从讲解示例开始。当打开示例 02-material-properties.html 时，你会看到一个跟前面示例中有点儿相似的空盒子。这个盒子正在绕 x 轴上下转动。在右上角的菜单上有几个滑块可以用来修改 Physijs 的属性。这些属性可以应用到场景中的方块和球上，这些方块和球可以通过 addCubes 和 addSpheres 按钮添加到场景中。当你点击 addSpheres 按钮时，会有 5 个球添加到场景中；点击 addCubes 按钮时，会有 5 个方块添加到场景中。如下图所示：



在这个示例中你可以试验创建 Physijs 时设置的 restitution 和 friction 属性。例如，如果你将 cubeFriction 属性设成 1，然后添加几个方块，你会看到即使地面在转动，但这些方块基本不动。如果你将 cubeFriction 属性设成 0，你会发现一旦地面不再水平，这些方块就开始滑动。如下图所示：



在这个示例里另外一个你可以设置的属性是 restitution。通过属性 restitution 你可以定义一个对象在碰撞后复原时所具有的能量。也就是说，一个高 restitution 值可以创建出一个富

有弹性的对象，而低 restitution 值则会使得一个对象碰倒另一个对象时会立即停下来。展示这个概念时最好使用球体，将 restitution 设成 1，点击 addSpheres 按钮几次。这样就可以创建出很多有弹性的球，到处弹跳。

在进入下一节之前，让我们来看看这个示例中的代码：

```
box = new Physijs.SphereMesh(
  new THREE.SphereGeometry( 2, 20 ),
  Physijs.createMaterial(
    new THREE.MeshPhongMaterial(
      {color: colorSphere, opacity: 0.8, transparent: true}),
       controls.sphereFriction, controls.sphereRestitution));

box.position.set(
  Math.random() * 50 -25,
  20 + Math.random() * 5,
  Math.random() * 50 -25);
scene.add( box );
```

这段代码会在将球添加到场景中时调用。这次我们使用了另外一种 Physijs 网格：SphereMesh。我们创建了一个 SphereGeometry，在 Physijs 提供的网格集合中，从逻辑上来讲跟这个几何体最吻合的就是 SphereMesh（下一节将进一步讲解）。创建 SphereMesh 时，我们传入了这个几何体，并用 Physijs.createMaterial 创建了一个 Physijs 材质。我们这么做就是为了能够为该对象设置摩擦系数和弹性系数。

到目前为止，我们看到的 Physijs 网格只是 BoxMesh 和 SphereMesh。下一节我们将讲解并展示 Physijs 提供的所有类型的网格，你可以用它们来包装各种对象。

12.3 基础图形

Physijs 提供了一些可以用来包装几何体的图形类。本节我们将会学习所有这些网格，并在示例中展示它们的用法。记住，使用这些几何体你唯一要做的就是将 THREE.Mesh 的构造函数替换成这些网格对象的构造函数。

下表是 Physijs 中所有网格对象的概览：

名　称	描　述
Physijs.PlaneMesh	这个网格可以用来创建一个厚度为 0 的平面。这样的平面也可以用 BoxMesh 对象包装一个高度很低的 THREE.CubeGeometry 来表示
Physijs.BoxMesh	如果是类似方块的几何体，你可以使用这个网格。例如，它的属性跟 THREE.CubeGeometry 的属性很相配
Physijs.SphereMesh	对于球形可以使用这个网格。它跟 THREE.SphereGeometry 的属性很相配
Physijs.CylinderMesh	通过设置 THREE.Cylinder 的属性你可以创建出各种柱状图形。Physijs 为各种柱形提供了不同网格。Physijs.CylinderMesh 可以用于一般的、上下一致的圆柱形

（续）

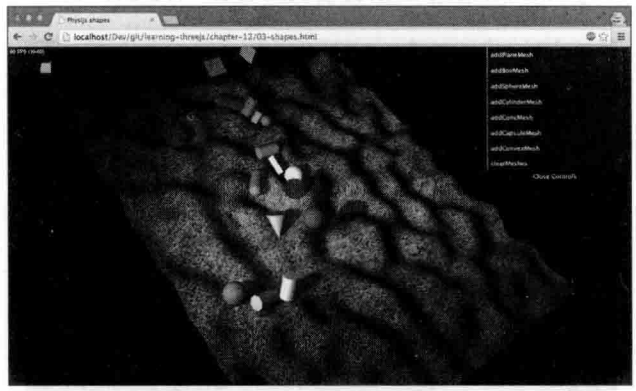
名　称	描　述
Physijs.ConeMesh	如果顶部的半径为 0，底部的半径值大于 0，那么你可以用 THREE.Cylinder 创建出一个圆锥体。如果你想在这样一个对象上应用物理效果，那么可以使用的、最相配的网格类就是 ConeMesh
Physijs.CapsuleMesh（胶囊网格）	胶囊跟 THREE.Cylinder 属性很像，但其顶部和底部是圆的。在本节下面部分我们会展示如何在 Three.js 中创建胶囊
Physijs.ConvexMesh（凸包网格）	Physijs.ConvexMesh 是一种比较粗略的图形，可用于多数复杂图形。它可以创建一个模拟复杂图形的凸包（类似于 THREE.ConvexGeometry 属性）
Physijs.ConcaveMesh	ConvexMesh 是一个比较粗略的图形，而 ConcaveMesh 则可以对复杂图形进行比较细致的表现。需要注意的是使用 ConcaveMesh 对效率的影响比较大。一般来讲比较好的方式是为每个几何体创建特定的 Physijs 网格，或者将它们组合在一起（就像我们在前一个示例里对地面所做的那样）
Physijs.HeightfieldMesh（高度场网格）	这是一种非常特殊的网格。通过该网格你可以从一个 THREE.PlaneGeometry 对象创建出一个高度场。参考 03-shapes.html

我们将使用 03-shapes.html 作为参考，快速浏览一遍这些图形。我们不会进一步讲解 Physijs.ConcaveMesh，因为它的使用范围十分有限。

在看示例之前，我们先快速看一下 Physijs.PlaneMesh。该网格可以基于 THREE.PlanceGeometry 创建出一个简单的平面：

```
var plane = new Physijs.PlaneMesh(
  new THREE.PlaneGeometry(5,5,10,10),
  material
);

scene.add( plane );
```

在这个函数里你可以看到我们只传入了一个 THREE.PlaneGeometry 即可创建网格。如果你把这个网格添加到场景中你会发现一些奇怪的事情：你刚创建的这个网格对重力没有反应。原因是 Physijs.PlaneMesh 的重量固定为 0，所以它不会受重力影响，或是在跟别的对象碰撞时移动。除了这个网格，其他所有网格都会像你所期待的那样受重力和碰撞的影响。



上图所展示的是示例 03-shapes.html。在这个示例里我们随机创建了一个高度场（稍后解释），通过右上角的菜单你可以放置各种图形的对象。如果你在这个示例里做试验，你会发现各种图形是如何在高度场动作的，以及它们之间是如何碰撞的。

让我们来看看这些图形的构造：

```
new Physijs.SphereMesh(new THREE.SphereGeometry(3,20),mat);
new Physijs.BoxMesh(new THREE.CubeGeometry(4,2,6),mat);
new Physijs.CylinderMesh(new THREE.CylinderGeometry(2,2,6),mat);
new Physijs.ConeMesh(new THREE.CylinderGeometry(0,3,7,20,10),mat);
```

这里并没有什么特殊的，我们创建出几何体，然后用 Physijs 中最贴合的网格创建出那些添加到场景中的对象。但是如果我们想用的网格是 Physijs.CaspsuleMesh 该怎么办呢？Three.js 中并没有类似胶囊的网格，所以我们必须自己创建一个：

```
var cyl = new THREE.CylinderGeometry(2,2,6 );
var top = new THREE.SphereGeometry(2);
var bot = new THREE.SphereGeometry(2);

// create normal meshes
var topMesh = new THREE.Mesh(top);
var botMesh = new THREE.Mesh(bot);
topMesh.position.y=3;
botMesh.position.y=-3;

// merge to create a capsule
THREE.GeometryUtils.merge(cyl,topMesh);
THREE.GeometryUtils.merge(cyl,botMesh);

// create a physijs capsule mesh
var capsule = new Physijs.CapsuleMesh(
  cyl,
  getMaterial()
);
```

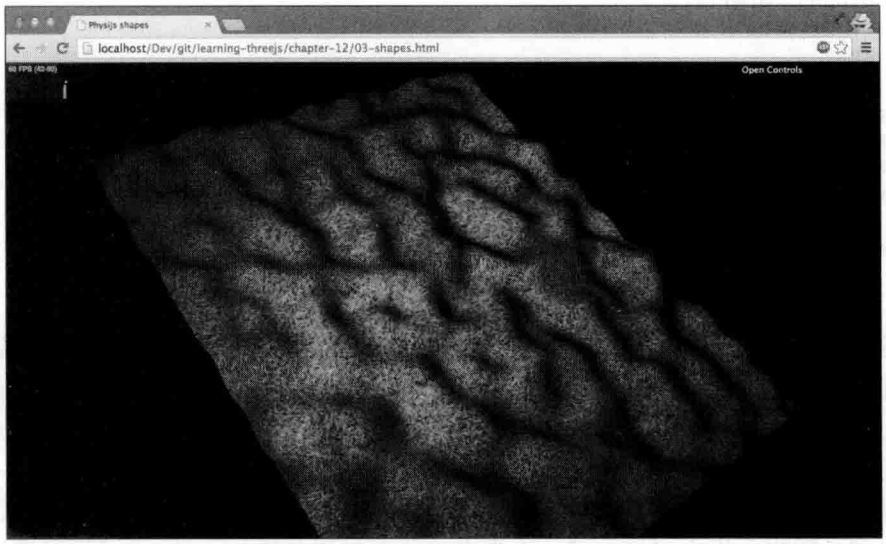
Physijs.CapsuleMesh 看上去像是圆柱体，但是它的顶部和底部是圆的。在 Three.js 里我们很容易就可以创建出这样的几何体，只要创建一个圆柱体（cyl）和两个球体（top 和 bot），然后用 THREE.GeometryUtils.merge() 函数将它们融合在一块儿即可。

在看高度场之前，我们先来看看最后一个可以添加到这个示例中的对象：Physijs.ConvexMesh（凸包网格）。凸包是包围某个几何体上所有顶点的最小图形。其结果是一个所有夹角都小于 180 度的图形。你可以在诸如环面纽结这样的复杂图形中使用这个网格：

```
var convex = new Physijs.ConvexMesh(
  new THREE.TorusKnotGeometry(0.5,0.3,64,8,2,3,10),
  material);
```

在本例中如果要模拟物理效果和碰撞，可以使用环面纽结的凸包。这是一种模拟复杂对象物理效果和碰撞的非常好的方法，而且可以保证对效率的影响最小。

Physijs 中最后一个要讨论的网格是 Physijs.HeightMap（高度场网格）。



通过高度场你很容易就可以创建出一个有凸起和洼地的地形。使用 Physijs.HeightMap 类我们可以保证所有对象能够对地形中的不同高度做出正确的反应。让我们来看看要达到该效果所需的代码：

```
var date = new Date();
var pn = new Perlin('rnd' + date.getTime());

function createHeightMap(pn) {

  var ground_material = Physijs.createMaterial(
    new THREE.MeshLambertMaterial({
      map: THREE.ImageUtils.loadTexture(
        '../assets/textures/ground/grasslight-big.jpg')
    }),
    0.3, // high friction
```

```
    0.8 // low restitution
  );

  var ground_geometry = new THREE.PlaneGeometry(120, 100, 100, 100);
  for (var i = 0; i < ground_geometry.vertices.length; i++) {
    var vertex = ground_geometry.vertices[i];
    var value = pn.noise(vertex.x / 10, vertex.y / 10, 0);
    vertex.z = value * 10;
  }
  ground_geometry.computeFaceNormals();
  ground_geometry.computeVertexNormals();

  var ground = new Physijs.HeightfieldMesh(
    ground_geometry,
    ground_material,
    0, // mass
    100,
    100
  );
  ground.rotation.x = Math.PI / -2;
  ground.rotation.y = 0.4;
  ground.receiveShadow = true;

  return ground;
}
```

在这段代码里我们要经过几个步骤才能创建出你在示例中所见的那个高度场。首先，我们创建了一个 Physijs 材质和一个简单的 PlaneGeometry 对象。要在 PlaneGeometry 对象上创建出凹凸不平的地形，我们需要遍历这个几何体的每个顶点，并随机设置顶点的 z 属性。为此我们使用了 Perlin 噪音生成器，就像我们在 10.2.3.2 节中创建凹凸贴图一样。我们需要调用 computeFaceNormals 和 computeVertexNormals 函数，以保证能够正确地渲染纹理、光照和阴影。现在我们有了一个带有正确高度信息的 PlaneGeometry 对象。基于这个 PlaneGeometry 对象我们可以创建出一个 Physijs.HeightFieldMesh 对象。其构造函数的最后两个参数分别是 PlaneGeometry 对象水平和垂直方向的分段数，这两个参数应当跟 PlaneGeometry 对象构造函数的最后两个参数一致。最后我们要把 HeightFieldMesh 对象旋转到所需的位置，然后就可以添加到场景中了。现在其他 Physijs 对象可以跟高度场正确地互动了。

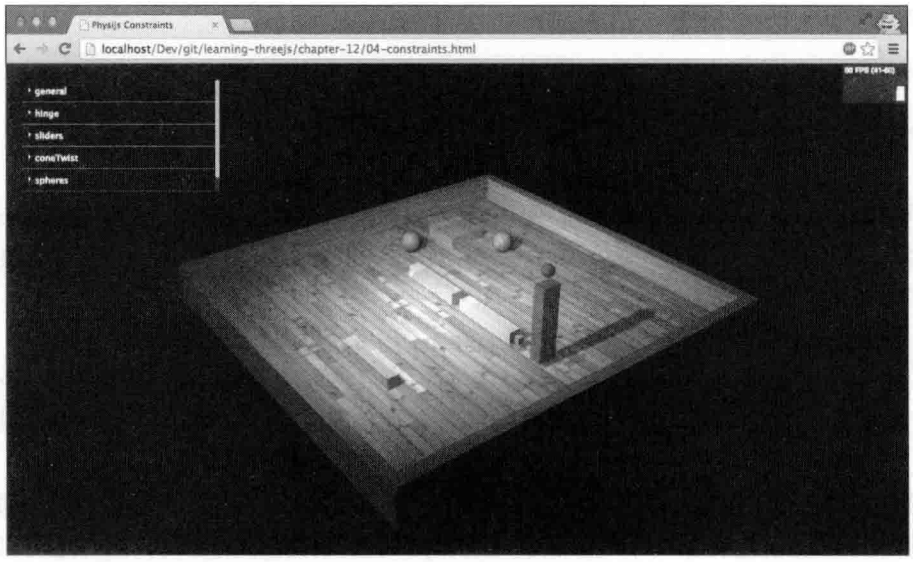
12.4 使用约束限制对象移动

现在我们已经看过一些基础的物理模拟的例子，包括各种图形如何对重力、摩擦和弹性做出反应，并影响碰撞。Physijs 还提供了一些高级对象，让你可以限制对象的移动。在 Physijs 里，这些对象称作约束。下表是 Physijs 中可用约束的概览：

约　束	描　述
PointConstraint	通过这个约束，你可以将一个对象与另一个对象之间的位置固定下来。如果一个对象动了，另一个对象也会随之移动，它们之间的距离和方向保持不变
HingeConstraint	通过活页约束，你可以限制一个对象只能像活页一样移动，例如门
SliderConstraint	顾名思义，这个约束，将对象的移动限制在一个轴上。例如移门
ConeTwistConstraint	通过这个约束，你可以用一个对象限制另一个对象的旋转和移动。这个约束的功能类似一个球销式关节。例如，胳膊在肩关节中的活动
DOFConstraint	通过自由度约束，你可以限制对象在任意轴上的活动，你可以设置对象活动的最小、最大角度。这是最灵活的约束方式

理解这些约束最简单的方法就是看一个实例，并亲手试验一下。为此我们提供了一个示例（04-physijs-constraints.js），里面用到了所有约束。

参考下面的屏幕截图：



我们会在这个示例的基础上讲解这 5 个约束中的 4 个。对于 DOFConstraint 我们创建了一个单独的示例。我们首先要看的是 PointConstraint。

12.4.1　用 PointConstraint 限制两点间的移动

打开这个示例你会看到两个红色的球。这两个球用 PointConstraint 对象连在一起。通过左上角的菜单你可以移动滑块。当滑块碰到了其中一个红球时，你会看到它们以相同的方式移动，并保持距离不变，而且仍然遵从重力、摩擦、弹性，以及其他物理规律。

本例中的 PointConstraint 创建方法如下：

```
function createPointToPoint() {
  var obj1 = new THREE.SphereGeometry(2);
  var obj2 = new THREE.SphereGeometry(2);

  var objectOne = new
    Physijs.SphereMesh(obj1,Physijs.createMaterial(
      new THREE.MeshPhongMaterial({color: 0xff4444, transparent:
        true, opacity:0.7}),0,0));
  objectOne.position.z=-18; objectOne.position.x=-10;
  objectOne.position.y=2;
  scene.add(objectOne);

  var objectTwo = new
    Physijs.SphereMesh(obj2,Physijs.createMaterial(
      new THREE.MeshPhongMaterial({color: 0xff4444, transparent:
        true, opacity:0.7}),0,0));
  objectTwo.position.z=-5; objectTwo.position.x=-20;
  objectTwo.position.y=2;
  scene.add(objectTwo);

  var constraint = new Physijs.PointConstraint(objectOne,
    objectTwo, objectTwo.position);
  scene.addConstraint(constraint);
}
```

你可以在这段代码里看到，我们使用特定的 Physijs 网格（本例中用的是 SphereMesh）创建对象，然后将它们添加到场景中。我们使用了 Physijs.PointConstraint 构造函数创建约束。这个约束有三个参数：

- 前两个参数指定要连接的两个对象。在本例中我们将两个球连接在一起。
- 第三个参数指定约束绑定的位置。例如，如果你要将第一个对象绑定到一个非常大的对象上，你可以将这个位置设置在那个对象的右边。一般来讲，如果你只想将两个对象连接在一起，那么你最好将这个位置设置在第二个对象的位置上。

如果你不想将一个对象绑定到另一个对象，而是绑定到场景中某个固定的点，那么你可以忽略第二个参数。这样第一个对象就会跟你指定的位置保持固定距离，同时还可以遵从重力和其他物理规律。

约束创建好之后，我们可以通过 addConstraint 函数将它添加到场景中，从而启用该约束。当你开始试验约束时，你有可能会遇到一些奇怪的问题。为了方便调试，你可以在 addConstraint 函数中传递一个参数 true。这样可以帮助你获取约束的旋转角度和位置。

12.4.2 用 HingeConstraint 创建类似门的约束

顾名思义，通过 HingeConstraint 你可以创建一个行为类似活页的对象。它可以绕固定的轴旋转，并可以限制在一定角度内。在我们的示例中，HingeConstraint 对象是通过场景中央的两块儿挡板显示出来的。这两块儿挡板被约束到两个褐色的小方块儿上，并绕它们旋转。

如果你想试验一下这两块儿挡板，你可以在 hinge 菜单中选中 enableMotor 复选框。这样可以将挡板的速度提升到 general 菜单中指定的速度。如果速度为负数，则挡板会向下转动；如果速度为正数，那它们就会向上转动。

让我们来看看如何创建一块儿挡板。

```
var constraint = new Physijs.HingeConstraint(
  flipperLeft, flipperLeftPivot, flipperLeftPivot.position,
  new THREE.Vector3(0,1,0));
scene.addConstraint(constraint);
constraint.setLimits(-2.2, -0.6, 0.1, 0);
```

该约束接受四个参数。每个参数的含义如下：

参　数	描　述
mesh_a	传递给该函数的第一个对象是将要被约束的对象。在本例中该对象是一个代表挡板的白色长方体。该对象的移动将会受到限制
mesh_b	第二个对象指定 mesh_a 受哪个对象约束。在本例中 mesh_a 受褐色的小方块约束。如果我们在场景中移动这个褐色的小方块，那么 mesh_a 也会跟着它移动，而且 HingeConstraint 始终起作用。你会发现所有约束都是这样。例如，你可以创建四处移动的汽车，而车门的打开必须受限制时使用这个约束。如果忽略第二个参数，那么活页将会被约束到场景（也不能移动）
position	约束应用的点。在本例中这个点就是 mesh_a 绕着旋转的点。如果你指定了 mesh_b，那么这个点会随着 mesh_b 的位置和旋转而移动
axis	活页绕着旋转的轴。在本例中我们将活页设置在水平方向（0，1，0）

将 HingeConstraint 对象添加到场景中的方法跟添加 PointConstraint 的方法一样。调用 addConstraint 方法，指定要添加的约束。为了调试，可以选择添加 true 参数，用来显示约束确切的位置和方向。但是对于 HingeConstraint 对象，我们需要设置这个约束的属性。为此我们可以使用 setLimits 函数。

该函数接受四个参数：

参　数	描　述
low	该属性可以指定旋转的最小弧度
high	该属性可以指定旋转的最大弧度
bias_factor	该属性指定处于错误位置时，约束进行纠正的速度。例如，当某个活页被别的对象推得超出了约束范围，该活页可以被移动到正确的位置。这个值越高，则位置纠正得越快。最好将该值保持在 0.5 以下
relaxation_factor	该属性指定约束以什么样的比率改变速度。如果该属性的值很高，那么对象在达到最小或最大角度时会被弹回来

如果需要，你可以在运行时修改这些属性。如果你在添加 HingeConstraint 时用到了这些属性，那么你并不会看到多少对象运动。这些网格只会在被别的对象碰到，或者受重力影响时才会移动。但是这个约束跟其他约束一样，可以由内部的马达驱动。这就是你在我们的示例中选中 enableMotor 复选框时所看到的现象。可以用下列代码激活马达：

```
constraint.enableAngularMotor( controls.velocity,
  controls.acceleration );
```

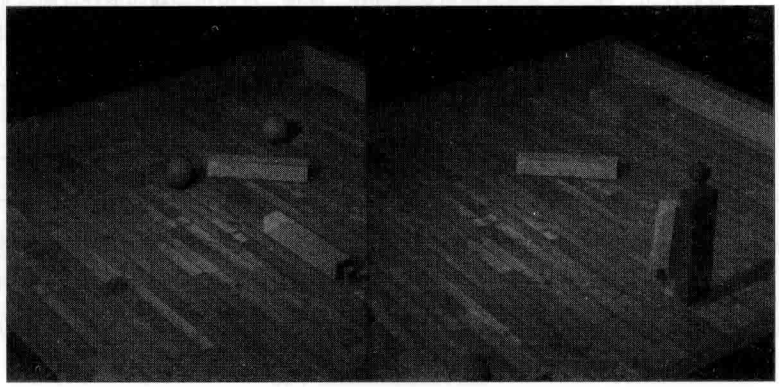
这段代码可以使用指定的加速度，将网格（在本例中是挡板）加速到指定的速度。如果想反向移动挡板，只要将速度设置为负数即可。如果我们不添加任何限制，那只要马达一直不停，这个挡板就会旋转起来。要关闭马达，我们可以调用：

```
flipperLeftConstraint.disableMotor();
```

现在这个网格受到摩擦、碰撞、重力和其他物理因素的影响，逐渐慢下来。

12.4.3 用 SliderConstraint 将移动限制到一个轴

接下来的约束是 SliderConstraint。通过这个约束，你可以将某个对象的移动限制到某个轴上。



示例 04-constraints.html 中的滑块儿可以通过子菜单 sliders 进行控制。点击 SlidersLeft 按钮，滑块儿向左移动（下限）；点击 SliderRight 按钮，它们会向右移动（上限）。用代码创建这些约束非常简单：

```
var constraint = new Physijs.SliderConstraint(
  sliderMesh,
  new THREE.Vector3(0, 2, 0),
  new THREE.Vector3(0, 1, 0));

scene.addConstraint(constraint);
constraint.setLimits(-10, 10, 0, 0);
constraint.setRestitution(0.1, 0.1);
```

正如你在代码中所看到的，这个约束接受三个参数（或是四个，如果想将一个对象约束到另一个对象）。该约束参数的含义如下表所示：

参　数	描　述
mesh_a	传递给该函数的第一个对象是需要约束的对象。在本例中，这个对象是一个表示滑块儿的绿色方块。这个对象的移动将受到限制
mesh_b	第二个参数指定的是 mesh_a 受哪个对象约束。这是一个可选的参数，在本例中没有提供。如果不指定该参数，那么 mesh_a 将会受场景的约束。如果指定了，那么当指定的网格移动或转动时，滑块儿也会跟着移动
position	约束应用的位置。该参数在将 mesh_a 约束到 mesh_b 时比较重要
axis	这个轴指的是 mesh_a 沿着移动的轴。注意，如果指定了 mesh_b，那么这个轴是相对于 mesh_b 方向的。在 Physijs 的当前版本中，如果使用线性马达和线性上下限，那么轴的方向会有一点儿奇怪的偏移。所以在当前版本中，如果你想沿某轴移动，所编写的具体代码如下所示： ❑ x 轴：new THREE.Vector3 (0, 1, 0) ❑ y 轴：new THREE.Vector3 (0, 0, Math.PI/2) ❑ z 轴：new THREE.Vector3 (Math.PI/2, 0, 0)

创建好约束，并调用 scene.addConstraint 添加到场景中之后，你可以设置这个约束的限制，以指定滑块能滑多远：constraint.setLimits（-10，10，0，0）。针对 SliderConstraint 你可以设置如下的限制：

参　数	描　述
linear_lower	该属性指定对象的线性下限
linear_upper	该属性指定对象的线性上限
angular_lower	该属性指定对象的角度下限
angular_higher	该属性指定对象的角度上限

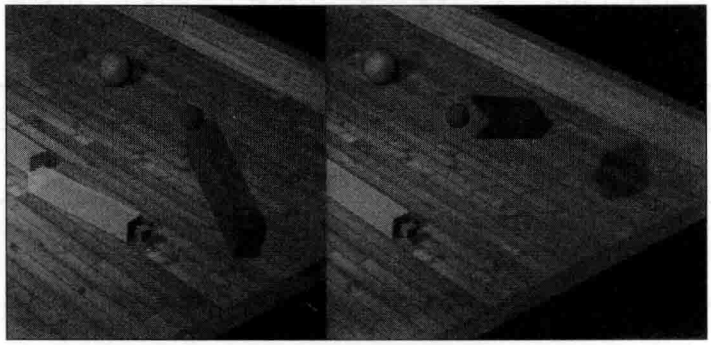
最后你可以设置达到限制时的弹性（反弹）。你可以使用函数 constraint.setRestitution(res_linear，res_angular)。其中第一个参数设置的是达到线性限制时的弹性；第二个参数设置的

是达到角度限制时的弹性。

现在约束配置好了，我们只要滑动滑块或使用马达，等待发生碰撞。对于 SlideConstraint，我们可以使用两种马达。我们可以使用角度马达，绕指定轴加速旋转对象，并遵从角度约束；或者使用线性马达，沿指定轴加速移动对象，并遵从线性约束。在这个例子里，我们用的是线性马达。如果要用角度马达，可以参考 12.4.5 节的内容。

12.4.4 用 ConeTwistConstraint 创建类似球销的约束

通过 ConeTwistConstraint 可以创建出一个移动受一系列角度限制的约束。我们可以指定一个对象绕另一个对象转动时在 x、y、z 轴上的最小角度和最大角度。



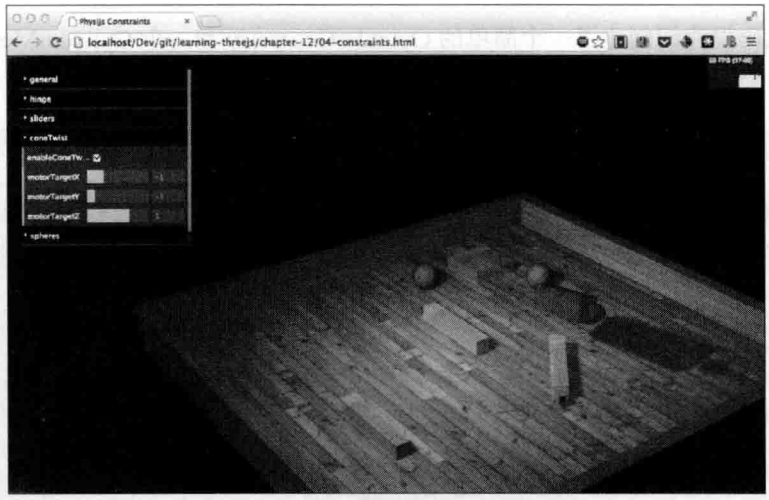
理解 ConeTwistConstraint 最好的方法就是看看创建约束的代码：

```
var baseMesh = new THREE.SphereGeometry(1);
var armMesh = new THREE.CubeGeometry(2, 12, 3);

var objectOne = new
  Physijs.BoxMesh(baseMesh,Physijs.createMaterial(
    new THREE.MeshPhongMaterial({color: 0x4444ff,
      transparent: true, opacity:0.7}), 0, 0), 0);
objectOne.position.z = 0;
objectOne.position.x = 20;
objectOne.position.y = 15.5;
objectOne.castShadow = true;
scene.add(objectOne);

var objectTwo = new
  Physijs.SphereMesh(armMesh,Physijs.createMaterial(
    new THREE.MeshPhongMaterial({color: 0x4444ff,
      transparent: true, opacity:0.7}), 0, 0), 10);
objectTwo.position.z = 0;
objectTwo.position.x = 20;
objectTwo.position.y = 7.5;
scene.add(objectTwo);
objectTwo.castShadow = true;
```

```
var constraint = new Physijs.ConeTwistConstraint(
  objectOne, objectTwo, objectOne.position);

scene.addConstraint(constraint);

constraint.setLimit(0.5*Math.PI, 0.5*Math.PI, 0.5*Math.PI);
constraint.setMaxMotorImpulse(1);
constraint.setMotorTarget(new THREE.Vector3(0, 0, 0));
```

在这段 JavaScript 代码里，你会发现几个我们在前面已经讨论过的概念。我们先是创建出几个用约束连接起来的对象：objectOne（球）和 objectTwo（盒子）。调整它们的位置，使得 objectTwo 处在 objectOne 的下面。现在我们可以创建 ConeTwistConstraint。如果你已经看过了其他的约束，那么这个约束的参数就没什么新鲜的了。第一个参数是要约束的对象，第二个参数是第一个参数要约束到的对象，最后一个参数是约束应用的位置（在本例中，这个位置就是 objectOne 绕着旋转的位置）。将约束添加到场景中之后，我们就可以通过 setLimts 函数设置它的限制。该函数接受三个弧度值，表示对象绕每个轴旋转的最大角度。

跟其他多数约束一样，我们可以使用该约束提供的马达驱动 objectOne。对于 ConeTwistConstraint，我们可以设置 MaxMotorImpulse 属性（即马达能够施加多大的力量），以及马达可以将 objectOne 转动到的角度。在本例中，我们将它移动到它的起始点，即球的正下方。在本例中，你可以设置目标值，试验该功能。下图展示的就是示例 04-constraints.html 的输出结果：



我们将要讨论的最后一个，也是最灵活的约束是：DOFConstraint。

12.4.5　用 DOFConstraint 实现细节的控制

通过 DOFConstraint，即自由度约束，你可以确切地控制对象线性和角度方向的移动。

我们将通过一个示例向你展示如何使用该约束。在这个示例里，你可以驱动一个类似小车的图形。这个图形中有一个方块用来表示车身，四个圆柱表示车轮。我们先创建车轮：

```
function createWheel(position) {
  var wheel_material = Physijs.createMaterial(
    new THREE.MeshLambertMaterial({
      color: 0x444444,
      opacity: 0.9,
      transparent: true
    }),
    1.0, // high friction
    0.5 // medium restitution
  );

  var wheel_geometry = new THREE.CylinderGeometry(4, 4, 2, 10);
  var wheel = new Physijs.CylinderMesh(
    wheel_geometry,
    wheel_material,
    100
  );

  wheel.rotation.x = Math.PI / 2;
  wheel.castShadow = true;
  wheel.position = position;
  return wheel;
}
```

在这段代码里我们创建了一个简单的 CylinderGeometry 和一个 CylinderMesh，可以用来表示小车的轮子。如下图所示：

接下来我们要创建车身，并将所有对象添加到场景中：

```
var car = {};
var car_material = Physijs.createMaterial(
  new THREE.MeshLambertMaterial({
    color: 0xff4444,
    opacity: 0.9,  transparent: true
  }),   0.5, 0.5
);

var geom = new THREE.CubeGeometry(15, 4, 4);
var body = new Physijs.BoxMesh(geom, car_material, 500);
body.position.set(5, 5, 5);
body.castShadow = true;
scene.add(body);

var fr = createWheel(new THREE.Vector3(0, 4, 10));
var fl = createWheel(new THREE.Vector3(0, 4, 0));
var rr = createWheel(new THREE.Vector3(10, 4, 10));
var rl = createWheel(new THREE.Vector3(10, 4, 0));

scene.add(fr);
scene.add(fl);
scene.add(rr);
scene.add(rl);
```

到目前为止我们创建的只是组成汽车的各个单独的组件。要把它们连在一块儿，我们需要创建约束。每个车轮可以约束到车身上：

```
var frConstraint = new Physijs.DOFConstraint(fr,body,
  new THREE.Vector3(0,4,8));
scene.addConstraint(frConstraint);
var flConstraint = new Physijs.DOFConstraint (fl,body,
  new THREE.Vector3(0,4,2));
scene.addConstraint(flConstraint);
var rrConstraint = new Physijs.DOFConstraint (rr,body,
  new THREE.Vector3(10,4,8));
scene.addConstraint(rrConstraint);
var rlConstraint = new Physijs.DOFConstraint (rl,body,
  new THREE.Vector3(10,4,2));
scene.addConstraint(rlConstraint);
```

每个车轮（第一个参数）都有它自己的约束，其中位置参数（第三个参数）指定车轮绑定到车身的什么地方。如果这样设置之后运行代码，我们就会看到车轮将车身托了起来。我们还要再做两件事才能让车动起来：设置车轮的约束（它们沿着移动的轴），并配置合适的马达。首先我们要为两个前轮设置约束，让它们只能绕 z 轴旋转，并拉动汽车。它们还不能沿其他轴运动。我们可以用如下代码实现这个约束：

```
frConstraint.setAngularLowerLimit({ x: 0, y: 0, z: 0 });
frConstraint.setAngularUpperLimit({ x: 0, y: 0, z: 0 });
flConstraint.setAngularLowerLimit({ x: 0, y: 0, z: 0 });
flConstraint.setAngularUpperLimit({ x: 0, y: 0, z: 0 });
```

这段代码第一眼看上去比较奇怪。将上限和下限设置成一样，可以保证对象在指定方向上不会旋转。这也就意味着车轮不会绕 z 轴旋转。我们之所以要这么指定，原因是当你在指定轴上启动马达时，这些限制将会被忽略。所以在这里设置 z 轴上的限制不会对这两个前轮有什么影响。

现在我们要驱动后面的车轮。要保证它们不会被落下，我们得将 x 轴固定住。通过下面的代码，我们可以：固定住 x 轴（将上限和下限都设为 0）；固定住 y 轴，就像轮子已经转过一样；取消 z 轴上的限制。代码如下所示：

```
rrConstraint.setAngularLowerLimit({ x: 0, y: 0.5, z: 0.1 });
rrConstraint.setAngularUpperLimit({ x: 0, y: 0.5, z: 0 });
rlConstraint.setAngularLowerLimit({ x: 0, y: 0.5, z: 0.1 });
rlConstraint.setAngularUpperLimit({ x: 0, y: 0.5, z: 0 });
```

正如你所看到的，要取消指定轴上的限制，我们必须把下限的值设得比上限的值稍微大一些。这样就可以绕这个轴自由地旋转。如果 z 轴不这么设置，那么这两个车轮就只能被拖着往前走。在本例中，由于和地面之间的摩擦作用，它们将会和其他车轮一起转动。

现在剩下要做的就只是为前轮设置马达了：

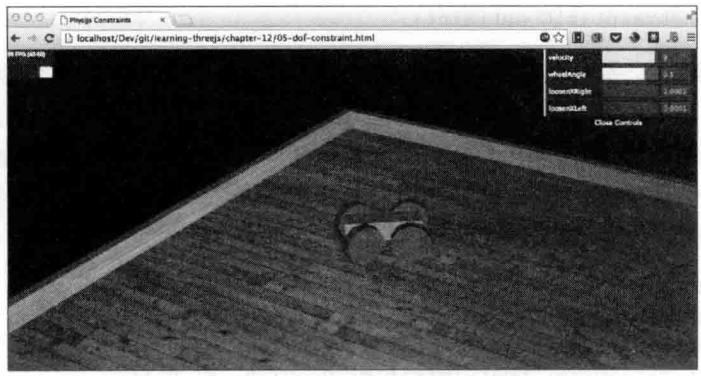
```
flConstraint.configureAngularMotor(2, 0.1, 0, -2, 1500);
frConstraint.configureAngularMotor(2, 0.1, 0, -2, 1500);
```

由于有三个轴，所以创建马达时我们要指定其工作的轴：0 是 x 轴，1 是 y 轴，2 是 z 轴。第二和第三个参数指定马达的角度限制。这里我们又将下限（0.1）设置得比上限（0）高一点儿，从而让它可以自由转动。第三个参数定义的是我们想要达到的速度。最后一个参数指的是这个马达可以施加的力。如果最后这个参数太小，那么小车就不会移动；如果太大，两个后轮将会脱离地面。

启动马达：

```
flConstraint.enableAngularMotor(2);
frConstraint.enableAngularMotor(2);
```

如果打开示例 05-dof-constraint.html，你就可以试验一下各种约束和马达，并驱动小车。如下图所示：



12.5　总结

在最后这章，我们探索了一下如何通过添加物理效果来扩展 Three.js 基础的三维功能。为此，我们使用了 Physijs 库，通过它你可以添加重力、碰撞、约束等。使用这个库最需要记住的是：

- 要使用 Physijs，你需要对已创建的场景进行修改，并指定重力加速度。你还要修改 render 循环，包含一个模拟步骤，以便告诉 Physijs 计算场景中所有对象的位置和角度。
- 只有那些被 Physijs 对应网格对象包装的几何体才能受物理效果的影响。多数情况下，你只要将 THREE.Mesh 换成 Physijs 中对应的网格对象即可。
- 通过 Physijs 材质你可以指定一个对象如何跟其他对象互动。在这个材质上你可以指定对象的摩擦系数和弹性系数。
- 在几何体上创建网格时一定要用正确的 Physijs 网格。对象间的碰撞是结合 Physijs 网格和底下的几何体一起计算完成的，不仅仅是几何体。
- 当你在场景中添加好对象之后，Physijs 就会负责计算这个对象的位置和角度。如果对象的位置和角度是在 Physijs 外部修改的，那么你要将 __dirtyRotation 属性或 __dirtyPostition 属性设为 true。
- 如果你的几何体比较复杂，尽量避免使用 ConcaveMesh。它对场景的性能影响很大。
- 约束是在场景中增加交互和物理效果非常有效的一种方式。但是使用约束并不是那么简单明了。你可以通过在 addConstraint 函数中增加一个额外的 true 参数，查看约束的实际行为。
- 通常来讲使用约束最好的方法是先用一个可以工作的模型，通过配置和调整来达到你期待的结果。

这就是本书所有的内容了。在这些章节里我们探讨了很多不同的主题，几乎涵盖了 Three.js 提供的所有功能。在开头的几章里，我们解释了一些核心的概念和一些 Three.js 背后的理论。然后我们介绍了可用的光源，以及材质是如何影响对象的渲染。在这些基础内容之后，我们又学习了 Three.js 提供的各种几何体，以及如何将这些几何体组合起来创建新的几何体。

在本书的第二部分，我们探讨了一些比较高级的主题。你学习了如何创建粒子系统，如何从外部文件中加载模型，以及创建动画。最后，在后面几章里我们看了可以用在骨骼动画中的高级纹理，以及可以在场景渲染完毕后应用的后期处理效果。我们以讲解物理效果的一章作为本书的结束。在这章除了解释如何在场景中添加物理效果，我们还给出了以 Three.js 为核心的一些活跃的社区项目，通过它们你可以在一个已经很棒的库中增加更多的功能。

我希望你在阅读本书，并试验书中示例时感到愉快，就像我在写本书时一样！

Learning Three.js

The JavaScript 3D Library for WebGL

Three.js开发指南

随着HTML 5标准的颁布，以及流行浏览器的日益强大，直接在浏览器中展示三维图形和动画已经变得越来越容易，也越来越受关注。但是三维图形和动画本身比较复杂，不仅需要有丰富的数学、图形学等方面的知识，还需要了解材质、贴图等各种创建三维场景所必备的要素。另外，直接使用WebGL在浏览器中创建三维图形和动画也非常繁琐，它所提供的各种接口尽管非常丰富且强大，但对于用户来说未免过于复杂了。

Three.js的出现则完美地帮助人们解决了这个矛盾。Three.js将WebGL的强大功能融汇其中，同时又非常易于使用，即便用户对其中的原理不甚了解，也能借助Three.js创建出绚丽多姿的三维场景和动画。

本书先从基本概念和Three.js的基本模块讲起，然后伴随着大量的示例和代码，逐步扩展到更多的主题，循序渐进地讲解Three.js的各种功能，帮助读者充分利用WebGL和现代浏览器的潜能，直接在浏览器中创建动态的华丽场景。

通过阅读本书，你将学会：

- 为Three.js工程创建HTML页面框架和渲染循环
- 使用Three.js提供的各种几何体
- 使用Three.js提供的各种相机控件，在三维场景中轻松导航
- 使用粒子系统创建出雨、雪，以及类似银河系的效果
- 从OBJ、STL、Collada等外部格式的文件中导入动画模型
- 在材质上应用高级纹理（凹凸贴图、法向贴图、高光贴图和光照贴图），创建逼真的三维图形
- 使用Physijs这个JavaScript库，实现三维物体的物理效果

[PACKT] PUBLISHING

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/程序设计
ISBN 978-7-111-48832-3
9 787111 488323 >
定价：59.00元