



全面讲解Three.js开发的实用指南，涵盖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

[PACKT]  
PUBLISHING

Web开发技术丛书

# Learning Three.js

Programming 3D animations and visualizations  
for the web with HTML5 and WebGL, Third Edition

[美] 乔斯·德克森 (Jos Dirksen) 著  
周翀 张薇 译

# Three.js开发指南

基于WebGL和HTML5在网页上渲染3D图形和动画  
(原书第3版)



机械工业出版社  
China Machine Press

# Three.js开发指南

## 基于 WebGL 和 HTML5 在网页上渲染 3D 图形和动画

### (原书第3版)

[美] 乔斯·德克森 (Jos Dirksen) 著  
周翀 张薇 译

## Learning Three.js

Programming 3D animations and visualizations for the  
web with HTML5 and WebGL, Third Edition



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

Three.js 开发指南：基于 WebGL 和 HTML5 在网页上渲染 3D 图形和动画（原书第 3 版）/（美）乔斯·德克森 (Jos Dirksen) 著；周翀，张薇译。—北京：机械工业出版社，2019.6

(Web 开发技术丛书)

书名原文：Learn Three.js: Programming 3D animations and visualizations for the web with HTML5 and WebGL, Third Edition

ISBN 978-7-111-62884-2

I. T… II. ① 乔… ② 周… ③ 张… III. JAVA 语言—程序设计—指南 IV. TP312.8-62

中国版本图书馆 CIP 数据核字 (2019) 第 103782 号

---

本书版权登记号：图字 01-2018-8338

Jos Dirksen: Learn Three.js: Programming 3D animations and visualizations for the web with HTML5 and WebGL, Third Edition (ISBN: 978-1-78883-328-8).

Copyright © 2018 Packt Publishing. First published in the English language under the title “Learn Three.js: Programming 3D animations and visualizations for the web with HTML5 and WebGL, Third Edition”.

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2019 by China Machine Press.

本书中文简体字版由 Packt Publishing 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

## Three.js 开发指南

### 基于 WebGL 和 HTML5 在网页上渲染 3D 图形和动画（原书第 3 版）

---

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：张梦玲

责任校对：殷 虹

印 刷：北京市荣盛彩色印刷有限公司

版 次：2019 年 6 月第 1 版第 1 次印刷

开 本：186mm×240mm 1/16

印 张：20.5

书 号：ISBN 978-7-111-62884-2

定 价：99.00 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) 88379426 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294

读者信箱：hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

## *The Translator's Words* 译者序

早年间，不论在何种硬件平台上，由于软硬件技术局限，交互式 3D 图形程序一直与低级的汇编语言和 C/C++ 语言深度绑定，这是因为更高级的编程语言要么难以访问底层图形 API（例如 OpenGL 或 Direct3D），要么没有足够的运行效率。

随着 Web 技术的不断发展，在网页上显示更丰富的内容逐渐成为时代趋势。其中，网页 3D 图形技术一直拥有较高呼声。然而由于上述编程语言绑定，这一技术的发展一直举步维艰。

幸运的是，个人计算机技术在最近 20 年间已经在很多关键方面取得了重大发展：CPU 的核心频率提升了约 40 倍；GPU 在早期只能做到全屏反锯齿和辅助线性插值，如今几乎可以直接渲染由 CAD 软件生成的 3D 模型；而更关键的是，随着 WebGL API 在 2011 年由 Khronos Group 推出后，3D 图形技术正式向高级编程语言 JavaScript 敞开大门。众所周知，JavaScript 语言是网页开发的当红主角，这也就意味着网页 3D 图形的时代正式到来，而随之而来的是无限的商机和全新的就业岗位。

本书将介绍 2018 年推出的 r95 版本 Three.js 库，以及与之相关的物理模拟库 Physijs。

Three.js 库是建立在 WebGL API 基础之上的高级 API，其功能介于底层图形 API 和通用 3D 引擎（例如 Unity 3D 等）之间。它具有通常在 3D 引擎中才能见到的“材质”概念，不但能够直接实现 Phong-Blinn 实时光照、实时阴影、法向贴图、环境贴图等传统技术，还能支持时下流行的“基于物理渲染”（PBR）技术。同时，它还提供了许多现成的着色器程序，可以实现 3D 引擎中常见的高级效果，包括但不限于全屏环境光遮挡（SSAO）、景深（DOF）等。

本书对基于 Three.js 库的网页 3D 图形编程进行了细致入微的介绍。每一个技术点的学习都建立在对示例程序的分析和把玩基础之上，并且没有涉及任何 3D 图形渲染所需的数学和物理原理及公式推导。这不但让 3D 图形零基础的读者能充分理解和快速掌握这些技术，而且使其学习过程像游戏一样生动有趣。

近些年，对于高级技术的研究和学习，在网络社区里有一句流行的调侃语：“从入门到放弃”。然而这一现象在学习本书的过程中并不存在。

# 前 言 *Preface*

在最近的几年中，浏览器的功能变得愈发强大，并且成为展现复杂的应用和图形的平台。然而其中大部分都是标准的二维图形。大多数现代浏览器已经支持 WebGL，不仅可以在浏览器端创建二维应用和图形，而且可以通过 GPU 的功能创建好看并且运行良好的三维应用。

然而，直接使用 WebGL 编程还是很复杂的。编程者需要知道 WebGL 的底层细节，并且学习复杂的着色语言来获得 WebGL 的大部分功能。Three.js 提供了一个很简单的关于 WebGL 特性的 JavaScript API，所以用户不需要详细地学习 WebGL，就能创作出好看的三维图形。

Three.js 为直接在浏览器中创建三维场景提供了大量的特性和 API。阅读这本书，读者将通过许多交互示例和示例代码学习 Three.js 提供的各种 API。

## 本书内容

第 1 章介绍使用 Three.js 的基本步骤。阅读完本章就能立刻创建你的第一个 Three.js 场景，也能直接在浏览器中创建你的第一个三维场景并让它动起来。

第 2 章解释使用 Three.js 时需要了解的基本组件。将介绍灯光、网格、几何形状、材质和摄像机。阅读本章，读者也将对 Three.js 提供的不同灯光和在场景中使用的摄像机有一个大概的印象。

第 3 章深入介绍可以在场景中使用的不同灯光。本章会通过示例讲解如何使用聚光灯、平行光、环境光、点光源、半球光和区域光。此外，本章还会展示如何使用镜头光晕来影响光源。

第 4 章讲述 Three.js 中可在网格中使用的材质。本章展示可以设置的所有属性，并且提供不同的交互示例来阐述 Three.js 中可用的材质。

第 5 章讲述 Three.js 中的几何体。本章将介绍如何在 Three.js 中创建和配置几何体，并且提供了几何体的交互示例，包括平面、圆形、任意图形、长方体、球体、圆柱体、圆环、环状扭结和多面体。

第 6 章接着第 5 章讲解如何使用几何体。本章介绍如何配置和使用 Three.js 提供的更

高级的几何体，如凸面体和旋转体。在本章，你还能学到如何从二维形状拉伸出三维几何体，并且学会通过二元操作来组合几何体，从而创建出一个新的几何体。

第 7 章解释如何使用 Three.js 中的精灵和粒子，介绍从头开始或者通过现有的几何体来创建一个粒子系统。这一章还介绍如何使用精灵或点云材质来改变单个点的外观。

第 8 章介绍如何通过外部资源导入网格和几何体，以及如何使用 Three.js 内置的 JSON 格式来保存几何体和场景。本章还会解释如何加载不同格式的模型，如 OBJ、DAE、STL、CTM、PLY 等。

第 9 章展示可以让场景动起来的各种动画，包括如何组合使用 Tween.js 库和 Three.js，以及如何使用基于变形和骨骼的动画模型。

第 10 章延续第 4 章的内容——使用 Three.js 材质。在第 4 章已经初步介绍了材质，本章着重介绍现有的各种材质以及如何把材质应用到网格中。另外，本章还介绍如何直接使用 HTML5 中的视频元素和画布元素作为材质来源。

第 11 章展示如何使用 Three.js 对已经渲染过的场景进行后期处理。通过后期处理，你可以将模糊、倾斜移位和着色等效果添加到已经渲染过的场景中。除此之外，本章还介绍如何创建自己的后期处理效果、自定义的顶点和片段着色器。

第 12 章解释如何将物理效果添加到 Three.js 场景中。通过物理效果，你能检测物体之间的碰撞，使物体对重力有响应，以及应用摩擦力。本章会展示如何用 Physijs JavaScript 库实现这些效果。此外，本章还会展示如何在 Three.js 场景中添加声音。

## 本书的使用要求

使用本书只需稍微了解 JavaScript，并按照第 1 章的指导搭建一个本地的网络服务器和获取书中使用的示例。

## 面向的读者

这本书对于每一个知道 JavaScript 并且想要在浏览器中创建运行的三维图形的人来说都是很棒的。你不需要知道任何高级的数学知识或 WebGL，所需的只是对 JavaScript 和 HTML 有大致的了解。书中所需要的材料和示例都可以免费下载，所用的工具都是开源的。所以，如果你想创建可以在任何现代浏览器中运行的交互的三维图形，这本书就适合你。

## 下载示例代码及彩图

本书的示例代码及彩图，可从 <http://www.packtpub.com> 通过个人账号下载，也可以访问华章公司官网 <http://www.hzbook.com>，通过注册并登录个人账号下载。

# 目 录 *Contents*

译者序

前言

<b>第 1 章 使用 Three.js 创建你的第一个三维场景</b>	1
1.1 准备工作	3
1.2 获取源码	5
1.2.1 通过 Git 获取代码仓库	5
1.2.2 下载并解压缩档案文件	5
1.2.3 测试示例	6
1.3 搭建 HTML 框架	9
1.4 渲染并查看三维对象	10
1.5 添加材质、光源和阴影效果	14
1.6 让你的场景动起来	16
1.6.1 引入 requestAnimationFrame() 方法	16
1.6.2 旋转立方体	18
1.6.3 弹跳球	19
1.7 使用 dat.GUI 简化试验流程	20
1.8 场景对浏览器的自适应	22
1.9 总结	23

<b>第 2 章 构建 Three.js 应用的基本组件</b>	24
2.1 创建场景	24
2.1.1 场景的基本功能	25
2.1.2 给场景添加雾化效果	29
2.1.3 使用 overrideMaterial 属性	30
2.2 几何体和网格	32
2.2.1 几何体的属性和方法	32
2.2.2 网格对象的属性和方法	36
2.3 选择合适的摄像机	40
2.3.1 正交投影摄像机和透视投影摄像机	41
2.3.2 将摄像机聚焦在指定点上	45
2.4 总结	46
<b>第 3 章 学习使用 Three.js 中的光源</b>	47
3.1 Three.js 中不同种类的光源	47
3.2 基础光源	48
3.2.1 THREE.AmbientLight	48
3.2.2 THREE.SpotLight	53
3.2.3 THREE.PointLight	58
3.2.4 THREE.DirectionalLight	62

3.3 特殊光源 .....	63	5.1.2 三维几何体 .....	109
3.3.1 THREE.HemisphereLight .....	64	5.2 总结 .....	120
3.3.2 THREE.AreaLight .....	65	<b>第 6 章 高级几何体和二元操作 .....</b>	122
3.3.3 镜头光晕 .....	67	6.1 THREE.ConvexGeometry .....	122
3.4 总结 .....	69	6.2 THREE.LatheGeometry .....	124
<b>第 4 章 使用 Three.js 的材质 .....</b>	70	6.3 通过拉伸创建几何体 .....	125
4.1 理解材质的共有属性 .....	71	6.3.1 THREE.ExtrudeGeometry .....	126
4.1.1 基础属性 .....	71	6.3.2 THREE.TubeGeometry .....	127
4.1.2 融合属性 .....	73	6.3.3 从 SVG 拉伸 .....	129
4.1.3 高级属性 .....	73	6.4 THREE.ParametricGeometry .....	131
4.2 从简单的网格材质开始 .....	74	6.5 创建三维文本 .....	133
4.2.1 THREE.MeshBasicMaterial .....	75	6.5.1 渲染文本 .....	133
4.2.2 THREE.MeshDepthMaterial .....	77	6.5.2 添加自定义字体 .....	136
4.2.3 联合材质 .....	79	6.6 使用二元操作组合网格 .....	137
4.2.4 THREE.MeshNormalMaterial .....	80	6.6.1 subtract 函数 .....	139
4.2.5 在单几何体上使用多种材质 .....	82	6.6.2 intersect 函数 .....	142
4.3 高级材质 .....	84	6.6.3 union 函数 .....	143
4.3.1 THREE.MeshLambertMaterial .....	85	6.7 总结 .....	143
4.3.2 THREE.MeshPhongMaterial .....	86	<b>第 7 章 粒子和精灵 .....</b>	145
4.3.3 THREE.MeshStandardMaterial .....	88	7.1 理解粒子 .....	145
4.3.4 THREE.MeshPhysicalMaterial .....	89	7.2 THREE.Points 和 THREE.	
4.3.5 用 THREE.ShaderMaterial		PointsMaterial .....	148
创建自己的着色器 .....	89	7.3 使用 HTML5 画布样式化粒子 .....	151
4.4 线性几何体的材质 .....	95	7.3.1 在 THREE.CanvasRenderer	
4.4.1 THREE.LineBasicMaterial .....	95	中使用 HTML5 画布 .....	151
4.4.2 THREE.LineDashedMaterial .....	97	7.3.2 在 WebGLRenderer 中使用	
4.5 总结 .....	97	HTML5 画布 .....	152
<b>第 5 章 学习使用几何体 .....</b>	99	7.4 使用纹理样式化粒子 .....	155
5.1 Three.js 提供的基础几何体 .....	101	7.5 使用精灵贴图 .....	160
5.1.1 二维几何体 .....	101		

7.6 从高级几何体创建	207
THREE.Points .....	163
7.7 总结.....	165
<b>第 8 章 创建、加载高级网格和几何体.....</b>	<b>166</b>
8.1 几何体组合与合并.....	166
8.1.1 对象组合 .....	166
8.1.2 将多个网格合并成一个网格 .....	168
8.2 从外部资源加载几何体 .....	170
8.2.1 以 Three.js 的 JSON 格式保存 和加载 .....	171
8.2.2 使用 Blender.....	177
8.3 导入三维格式文件.....	181
8.3.1 OBJ 和 MTL 格式 .....	181
8.3.2 加载 Collada 模型 .....	185
8.3.3 从其他格式的文件中加载 模型 .....	186
8.3.4 展示蛋白质数据银行中的 蛋白质 .....	190
8.3.5 从 PLY 模型中创建粒子系统 .....	192
8.4 总结.....	194
<b>第 9 章 创建动画和移动摄像机.....</b>	<b>195</b>
9.1 基础动画 .....	195
9.1.1 简单动画 .....	196
9.1.2 选择对象 .....	197
9.1.3 使用 Tween.js 实现动画 .....	198
9.2 使用摄像机.....	201
9.2.1 轨迹球控制器 .....	202
9.2.2 飞行控制器 .....	204
9.2.3 第一视角控制器 .....	205
9.2.4 轨道控制器 .....	207
9.3 变形动画和骨骼动画 .....	208
9.3.1 用变形目标创建动画 .....	209
9.3.2 用骨骼和蒙皮创建动画 .....	217
9.4 使用外部模型创建动画 .....	220
9.4.1 使用 Blender 创建骨骼动画 .....	220
9.4.2 从 Collada 模型加载动画 .....	223
9.4.3 从雷神之锤模型中加载动画 .....	225
9.4.4 使用 gltfLoader.....	225
9.4.5 利用 fbxLoader 显示动作 捕捉模型动画 .....	227
9.4.6 通过 xLoader 加载古老的 DirectX 模型.....	228
9.4.7 利用 BVHLoader 显示骨骼 动画 .....	230
9.4.8 如何重用 SEA3D 模型 .....	231
9.5 总结.....	232
<b>第 10 章 加载和使用纹理 .....</b>	<b>233</b>
10.1 将纹理应用于材质 .....	233
10.1.1 加载纹理并应用到网格 .....	233
10.1.2 使用凹凸贴图创建褶皱 .....	238
10.1.3 使用法向贴图创建更加 细致的凹凸和褶皱 .....	239
10.1.4 使用移位贴图来改变顶点 位置 .....	240
10.1.5 用环境光遮挡贴图实现 细节阴影 .....	241
10.1.6 用光照贴图产生假阴影 .....	243
10.1.7 金属光泽度贴图和粗糙度 贴图 .....	244
10.1.8 Alpha 贴图 .....	246

10.1.9	自发光贴图	247	11.4	创建自定义后期处理着色器	287
10.1.10	高光贴图	248	11.4.1	自定义灰度图着色器	287
10.1.11	使用环境贴图创建伪镜面 反射效果	250	11.4.2	自定义位着色器	290
10.2	纹理的高级用途	255	11.5	总结	292
10.2.1	自定义 UV 映射	255			
10.2.2	重复纹理	258			
10.2.3	在画布上绘制图案并作为 纹理	260			
10.2.4	将视频输出作为纹理	262			
10.3	总结	263			
<b>第 11 章 自定义着色器和后期处理</b> 265					
11.1	配置 Three.js 以进行后期处理	265	12.1	创建基本的 Three.js 场景	294
11.2	后期处理通道	268	12.2	Physi.js 材质属性	298
11.2.1	简单后期处理通道	270	12.3	Physi.js 基础形体	300
11.2.2	使用掩码的高级效果组合器	274	12.4	使用约束限制对象的移动	304
11.2.3	高级渲染通道：景深效果	278	12.4.1	使用 PointConstraint 限制 对象在两点间移动	305
11.2.4	高级渲染通道：环境光遮挡	280	12.4.2	使用 HingeConstraint 创建 类似门的约束	306
11.3	使用 THREE.ShaderPass 自定义效果	281	12.4.3	使用 SliderConstraint 将移动 限制在一个轴上	309
11.3.1	简单着色器	283	12.4.4	使用 ConeTwistConstraint 创建 类似球销的约束	311
11.3.2	模糊着色器	285	12.4.5	使用 DOFConstraint 实现 细节的控制	312
			12.5	在场景中添加声源	315
			12.6	总结	317

# 使用Three.js创建你的第一个三维场景

现代浏览器直接通过 JavaScript 就可以实现非常强大的功能。使用 HTML5 的标签可以很容易地添加语音和视频，而且在 HTML5 提供的画布上可以添加具有交互功能的组件。现在，现代浏览器也开始支持 WebGL。通过 WebGL 可以直接使用显卡资源来创建高性能的二维和三维图形，但是直接使用 WebGL 编程来从 JavaScript 创建三维动画场景十分复杂而且还容易出问题。使用 Three.js 库可以简化这个过程。Three.js 带来的好处有以下几点：

- 创建简单和复杂的三维几何图形。
- 创建虚拟现实（VR）和增强现实（AR）场景。
- 在三维场景下创建动画和移动物体。
- 为物体添加纹理和材质。
- 使用各种光源来装饰场景。
- 加载三维模型软件所创建的物体。
- 为三维场景添加高级的后期处理效果。
- 使用自定义的着色器。
- 创建点云（即粒子系统）。

通过几行简单的 JavaScript 代码，你可以创建从简单三维模型到类似图 1.1（在浏览器中访问 <http://www.vill.ee/eye>）所示的具有真实感的场景。

本章将会通过示例来阐述 Three.js 的工作原理，但是不会对其中的技术细节进行深入探究，这些细节我们将会在后面的章节中介绍。本章主要涵盖以下几方面：

- 使用 Three.js 所需的工具。
- 下载本书所需的源码和示例。

- 创建第一个 Three.js 场景。
- 使用材质、光源和动画来完善第一个场景。
- 引入辅助库以统计和控制场景。



图 1.1

首先将会对 Three.js 进行简单的介绍，然后再讲解第一个示例及其代码。在开始之前我们先来看下当前主流浏览器对 WebGL 的支持情况，几乎所有浏览器的桌面版和移动版均支持 WebGL，唯一的例外是移动版迷你 Opera 浏览器（Opera Mini），因为这个浏览器有一种特殊的工作模式，可以在 Opera 服务器端进行页面渲染，而 Opera 服务器往往禁止运行 JavaScript。不过从 8.0 版开始，Opera Mini 的默认工作模式已改为使用 iOS Safari 引擎渲染页面，从而也可以很好地支持 JavaScript 和 WebGL。但是新版 Opera Mini 仍然可以被设置为不支持 JavaScript 的“迷你模式”（mini mode）。

除了 IE 的一些低版本浏览器，基本所有的现代浏览器都支持 Three.js。如果想在低版本的 IE 浏览器上运行 Three.js，你还需要做额外的操作。对于 IE10 和更低的版本，你可以安装 iewebgl 插件，下载地址为 <https://github.com/iewebgl/iewebgl>。

使用 WebGL 能够创建出具有交互性的 3D 模型，而且这些模型在电脑和手机设备上都能够很好地运行。



本书主要使用 Three.js 提供的基于 WebGL 的渲染器。但是 Three.js 也提供了基于 CSS 3D 的渲染器，使用其 API 能够很容易地创建出三维场景，而且这个渲染器的优点在于几乎所有手机和电脑上的浏览器都支持 CSS 3D，并且可以在三维空间内渲染 HTML 元素。第 7 章会进一步介绍如何使用 CSS 3D。

通过本章的学习，你可以直接创建第一个三维场景，而且这个场景可以在上述的所有浏览器中运行。目前我们不会介绍太多 Three.js 的复杂特性，但是在本章结束的时候你能够创建出如图 1.2 所示的场景。

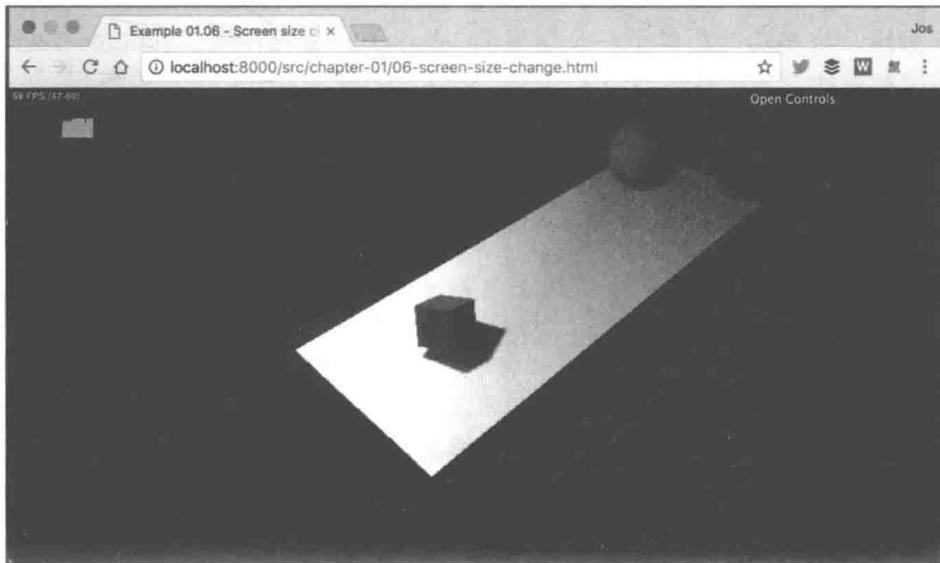


图 1.2

通过创建这个场景，你可以学到 Three.js 的基础知识，并可以创建第一个动画。在开始实现这个示例之前，我们会先介绍几个工具，这些工具可以帮助你更方便地使用 Three.js。我们还会介绍如何下载本书所用到的示例。

## 1.1 准备工作

Three.js 是一个 JavaScript 库，所以你只需要一个文本编辑器和支持 Three.js 的浏览器就可以使用 Three.js 来创建 WebGL 应用。这里推荐如下几款 JavaScript 文本编辑器：

- Visual Studio Code：Visual Studio Code 是微软公司面向所有常见操作系统推出的一款免费编辑器。该编辑器能够基于源程序里的类型和函数声明以及导入库，提供效果良好的语法高亮和代码补全功能。同时它还具有简洁明了的用户界面，以及对 JavaScript 的完美支持。下载地址为 <https://code.visualstudio.com/>。
- WebStorm：WebStorm 是 JetBrains 公司旗下的一款 JavaScript 编辑工具。它支持代码补全、自动部署和代码调试功能。除此之外，WebStorm 还支持 GitHub 和其他各种版本控制器。读者可从 <http://www.jetbrains.com/webstorm/> 下载一个试用版本。
- Notepad++：Notepad++ 是 Windows 操作系统下的通用文本编辑器，它支持各种

编程语言语法高亮度显示，而且可以很容易地对 JavaScript 进行布局和格式化。Notepad++ 的下载地址为 <http://notepad-plus-plus.org/>。

- ❑ Sublime Text：Sublime Text 是一款对 JavaScript 支持非常好的文本编辑器。除此之外，Sublime Text 的一大亮点是支持多重选择——同时选择多个区域，然后同时进行编辑。这些功能提供了一个很好的 JavaScript 编程环境。Sublime Text 是一个收费闭源软件，下载地址为 <http://www.sublimetext.com/>。

除此之外，还有很多可以编写 JavaScript 进而创建 Three.js 应用的开源和商用编辑器。还有一款基于云的代码编辑平台 Cloud9，网址是 <http://c9.io>，该平台可以连接 GitHub 账号，由此可以直接获取本书相关的代码和示例。



除了使用这些文本编辑器来运行本书相关的代码和示例，Three.js 自身也提供了在线场景编辑器，访问地址为 <http://threejs.org/editor>。使用该编辑器，可以用图形化方法创建 Three.js 场景。

虽然现代浏览器基本都支持 WebGL 并能运行 Three.js 应用，但是本书所采用的是 Chrome 浏览器。因为 Chrome 是对 WebGL 支持最好的浏览器，并且拥有强大的 JavaScript 代码调试功能。如图 1.3 所示，使用调试器的断点和控制台功能可以快速地定位问题。在本书中，还会进一步介绍各种调试的方法和技巧。

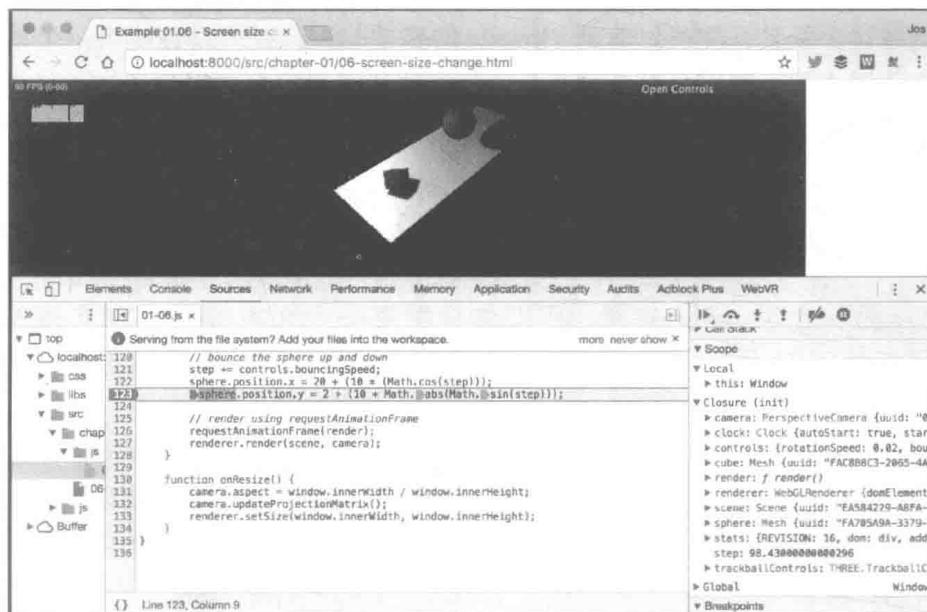


图 1.3

Three.js 就介绍到此，下面就开始获取源码并创建我们的第一个场景。

## 1.2 获取源码

本书所有的源码都可以从 GitHub (<https://github.com/>) 获取。GitHub 是基于 Git 的在线代码仓库，你可以使用它来存储、访问源码和进行版本控制。获取 GitHub 上源码的方式有以下两种：

- 通过 Git 获取代码仓库
- 下载并解压缩档案文件

接下来会详细讲解这两种方式。

### 1.2.1 通过 Git 获取代码仓库

Git 是开源、分布式的版本控制系统，本书的示例都是基于 Git 进行创建和版本管理的，GitHub 仓库的访问地址为 <https://github.com/josdirksen/learning-threejs>。

如果你已经在操作系统中安装了 Git 的客户端，那么就可以使用 git 命令来克隆示例的代码仓库。如果你还没有安装，可以访问 <http://git-scm.com> 下载，或者使用 GitHub 公司自己为 Mac 或 Windows 系统提供的客户端。在安装完 Git 客户端后，打开控制台并在你想要存储代码的目录中运行如下的命令：

```
git clone https://github.com/josdirksen/learning-threejs-third
```

如图 1.4 所示，代码就开始下载了。

```
1. jos@Joss-MacBook-Pro-3: /Users/ljts (zsh)
→ ljts git clone https://github.com/josdirksen/learning-threejs-third.git
Cloning into 'learning-threejs-third'...
remote: Counting objects: 493, done.
remote: Total 493 (delta 0), reused 0 (delta 0), pack-reused 493
Receiving objects: 100% (493/493), 23.02 MiB | 6.37 MiB/s, done.
Resolving deltas: 100% (273/273), done.
→ ljts
```

图 1.4

下载完毕后，在 learning-threejs-third 文件夹中会看到本书所用的所有的示例。

### 1.2.2 下载并解压缩档案文件

如果你不想使用 Git 的方式从 GitHub 上获取源码，那么还可以在 GitHub 上下载档案文件。在浏览器上访问 <https://github.com/josdirksen/learning-threejs-third>，点击右侧的 Clone or download 按钮。如图 1.5 所示。

解压文件到指定的目录，这样就可以获取所有的示例了。

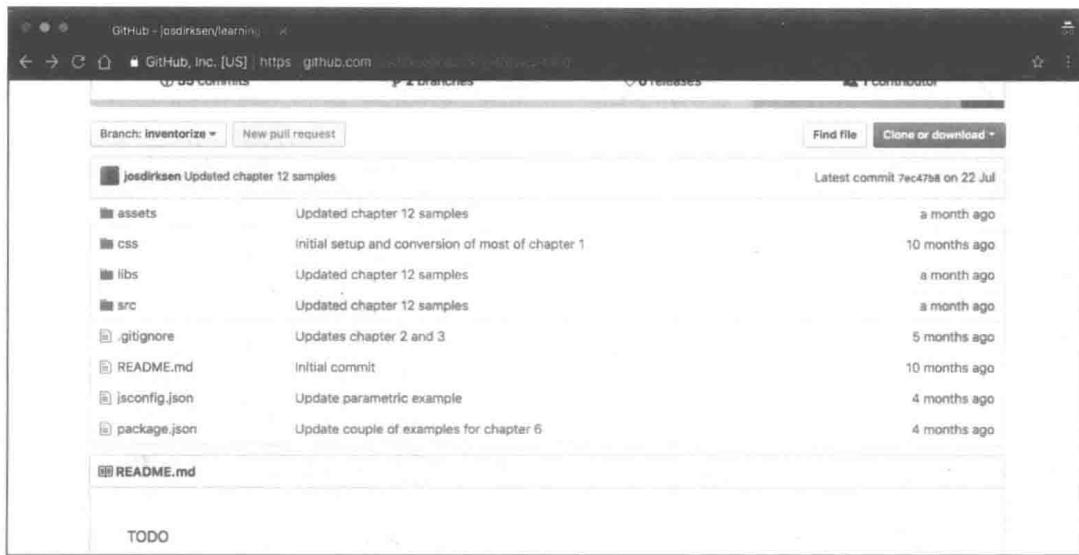


图 1.5

### 1.2.3 测试示例

现在你已经下载或者克隆了示例源码，下面我们就来测试示例是否能够正常工作，进而了解工程的目录结构。本书的示例是按照章节进行组织的。如果要运行本书的示例，你可以使用浏览器打开 HTML 文件，或者安装本地 Web 服务器。第一种方法对于简单的示例是可行的，但是如果示例中需要下载外部资源时，比如模型或者纹理图像，那么仅仅使用浏览器打开 HTML 文件是行不通的。这个时候，我们就需要本地 Web 服务器来确保外部资源正确加载。接下来我们会介绍几种安装本地服务器的方式，如果你无法安装本地服务器但使用 Chrome 或 Firefox 浏览器，那么我们也会介绍如何禁用安全性检测来运行示例。

接下来，我们将会介绍几种安装本地服务器的方式，依据系统的配置，你可以选择最合适的方式。

#### 1. 适用于 Unix/Mac 系统的基于 Python 的 Web 服务器

大部分的 Unix/Linux/Mac 系统默认安装了 Python，在示例源码目录中运行如下的命令就可以将本地 Web 服务器启动起来了。

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

在你下载了源码的目录中执行上述命令。

#### 2. 基于 NPM 的 Web 服务器

如果你已经在使用 Node.js 了，那么你肯定已经安装了 NPM。使用 NPM，有两个方式可以快速地搭建本地 Web 服务器：第一种方式是使用 http-server 模块，如下所示：

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

第二种方式是使用 simple-http-server，如下所示：

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

第二种方式的缺点在于无法自动地显示目录列表，而第一种方式是可以的。

### 3. Mac/Windows 上的轻量级服务器——Mongoose

如果你还没有安装 Python 或者 NPM，那么还有一个简单、轻量级的 Web 服务器——Mongoose。首先，从 <https://code.google.com/p/mongoose/downloads/list> 下载你的系统所支持的二进制安装文件。如果你使用的是 Windows 系统，那么将下载好的二进制文件复制到示例所在的目录，双击即可启动一个运行于该目录下的 Web 服务器。

对于其他的操作系统，则须将下载的二进制文件复制到指定的目录中，但是启动的方式不是双击，而是通过命令行的方式，如图 1.6 所示。

图 1.6

在这两种情况下都会在 8080 端口启动一个本地 Web 服务器。本书的示例目录如图 1.7 所示。

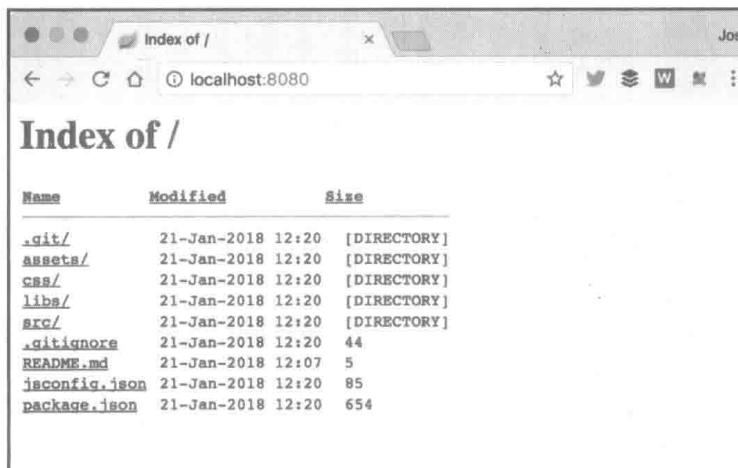


图 1.7

点击各章节的目录就可以获取相应章节的示例。在本书讲解提到某个示例时会指出示例的名称和所在的文件夹，这样你就可以找到该示例的源代码以便测试运行或自行研究。

#### 4. 禁用 Firefox 和 Chrome 的安全性检测

如果你使用的是 Chrome 浏览器，有种方式可以禁用安全性设置，这样就可以在没有 Web 服务器的情况下查看示例。需要注意的是，应尽量避免用下面的方法访问真正的网站，因为这样做会使浏览器向各种恶意代码敞开大门。用下面的命令可以启动 Chrome 浏览器同时禁用所有安全特性。

- 对于 Windows 操作系统：

```
chrome.exe --disable-web-security
```

- 对于 Linux 操作系统：

```
google-chrome --disable-web-security
```

- 对于 Mac OS 操作系统：

```
open -a "Google Chrome" --args --disable-web-security
```

通过这种方式启动 Chrome 浏览器就可以直接运行本地文件系统中的示例。

对于 Firefox 浏览器来说，还需要其他的配置。打开 Firefox 浏览器并在地址栏内输入 about:config 会看到图 1.8 所示的页面。

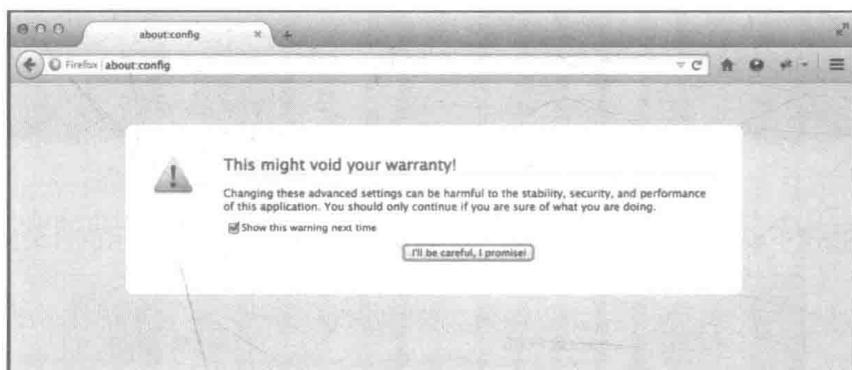


图 1.8

在该页面中点击“ I'll be careful, I promise! ”按钮，然后就会列出所有用于调整 Firefox 的属性。在搜索框中输入 security.fileuri.strict\_origin\_policy，并将其值修改为 false。如图 1.9 所示。

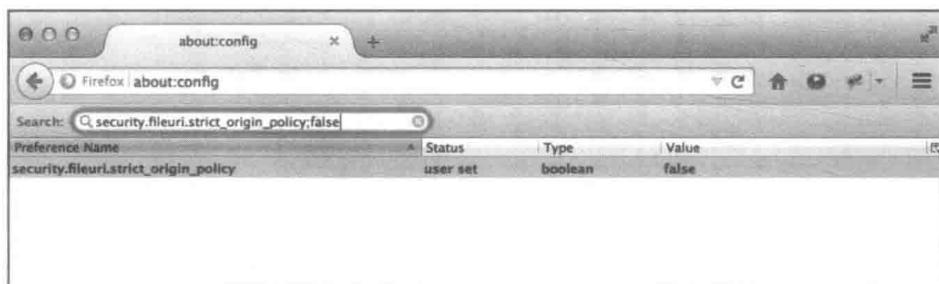


图 1.9

这时候就可以使用 Firefox 浏览器直接运行本书的示例了。

到现在为止，你应该已经安装了 Web 服务器或者禁用了浏览器的安全性设置，那么就开始创建第一个三维场景吧。

## 1.3 搭建 HTML 框架

首先我们将会创建空的 HTML 框架，后面的示例将在这个框架的基础上完成。具体如下所示：

```
<!DOCTYPE html>
<html>

<head>
    <title>Example 01.01 - Basic skeleton</title>
    <meta charset="UTF-8" />
    <script type="text/javascript" charset="UTF-8"
src="../../libs/three/three.js"></script>
    <script type="text/javascript" charset="UTF-8"
src="../../libs/three/controls/TrackballControls.js"></script>
    <script type="text/javascript" src=".js/01-01.js"></script>
    <link rel="stylesheet" href="../../css/default.css">
</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">
        (function () {
            // contains the code for the example
            init();
        })();
    </script>
</body>

</html>
```

你应该已经发现了，这个框架是个仅包含一些基本元素的 HTML 网页。在 `<head>` 标签中列出了示例所使用的外部 JavaScript 库，在这里至少要包含 Three.js 库。此外，这里还包含一个控制器库 TrackballControls.js。有了它便可以利用鼠标任意移动摄像机，以便从不同角度观察场景。在 `<head>` 标签中最后一个被包含的 JavaScript 文件是本章的示例程序，文件名为 01-01.js。最后，在 `<head>` 标签中我们还添加了几行 CSS 代码，这些 CSS 代码用于移除 Three.js 场景网页中的滚动条。在 `<body>` 标签中我们只添加了一个 `<div>` 元素，当我们写 Three.js 代码时，会把 Three.js 渲染器的输出指向这个元素。在框架网页的最后还有少量 JavaScript 代码。这些代码将在网页加载完成后被自动调用，我们利用这个机会调用

init() 函数。init() 函数也在 01-01.js 文件中定义，它将为 Three.js 场景做必要的初始化设置。不过在本章中，init() 函数暂时仅仅向控制台窗口打印当前 Three.js 的版本信息。

```
function init() {
    console.log("Using Three.js version: " + THREE.REVISION);
}
```

在浏览器中打开本章示例代码文件并观察控制台窗口，可以看到如图 1.10 所示的内容。



图 1.10

在 <head> 元素中包含 Three.js 的源代码。Three.js 有两个不同的版本：

- ❑ three.min.js：这个版本的 JS 库一般应用于网上部署 Three.js 时。该版本是使用 UglifyJS 压缩过的，它的大小是普通 Three.js 版本的四分之一。本书示例所使用的是 2018 年 7 月发布的 Three.js r95 版本。
- ❑ three.js：这个是普通的 Three.js 库。为了便于进行代码调试和理解 Three.js 的源码，本书的示例使用的都是这个库。

接下来，我们将会创建第一个三维对象并将其渲染到已经定义好的 <div> 元素中。

## 1.4 渲染并查看三维对象

在这一步，我们将会创建第一个场景并添加几个物体和摄像机。我们的第一个示例将会包含表 1.1 所列对象。

表 1.1

对 象	描 述
平面	该对象是二维矩形，可以作为场景中的地面。渲染的结果是屏幕中央的灰色矩形
方块	该对象是三维立方体，渲染为红色
球体	该对象是三维球体，渲染为蓝色
摄像机	摄像机决定着你所能够看到的输出结果
轴	分为x、y和z轴。通过它可以确定对象在三维空间的位置。其中x轴着色为红色，y轴着色为绿色，z轴着色为蓝色

下面将会通过代码示例（带注释的代码在文件 chapter-01/is/01-02.js 中）来解释如何创建场景并渲染三维对象：

```

function init() {
    var scene = new THREE.Scene();
    var camera = new THREE.PerspectiveCamera(45, window.innerWidth / window.innerHeight, 0.1, 1000);
    var renderer = new THREE.WebGLRenderer();
    renderer.setClearColor(new THREE.Color(0x000000));
    renderer.setSize(window.innerWidth, window.innerHeight);
    var axes = new THREE.AxesHelper(20);
    scene.add(axes);

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

    var plane = new THREE.Mesh(planeGeometry, planeMaterial);
    plane.rotation.x = -0.5 * Math.PI;
    plane.position.set(15, 0, 0);
    scene.add(plane);

    // create a cube
    var cubeGeometry = new THREE.BoxGeometry(4, 4, 4);
    var cubeMaterial = new THREE.MeshBasicMaterial({
        color: 0xFF0000,
        wireframe: true
    });
    var cube = new THREE.Mesh(cubeGeometry, cubeMaterial);
    cube.position.set(-4, 3, 0);
    scene.add(cube);

    // create a sphere
    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.set(20, 4, 2);
    scene.add(sphere);
}

```

```

// position and point the camera to the center of the scene
camera.position.set(-30, 40, 30);
camera.lookAt(scene.position);

// add the output of the renderer to the html element
document.getElementById("webgl-
output").appendChild(renderer.domElement);

// render the scene
renderer.render(scene, camera);
}

```

在浏览器中将示例打开，看到的结果和我们的目标，即本章开始时所展示的那张渲染图接近，但效果还有些差距。目前的效果如图 1.1 所示。

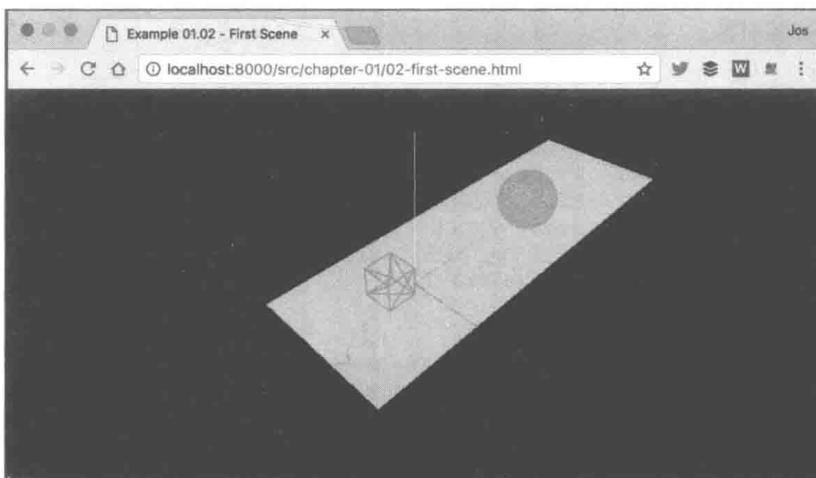


图 1.11

接下来将会对代码进行分析，这样我们就可以了解代码是如何工作的：

```

var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(45, window.innerWidth /
window.innerHeight, 0.1, 1000);
var renderer = new THREE.WebGLRenderer();
renderer.setClearColor(new THREE.Color(0x000000));
renderer.setSize(window.innerWidth, window.innerHeight);

```

在代码中，首先定义了场景（`scene`）、摄像机（`camera`）和渲染器（`renderer`）对象。场景是一个容器，主要用于保存、跟踪所要渲染的物体和使用的光源。如果没有 `THREE.Scene` 对象，那么 Three.js 就无法渲染任何物体，在第 2 章还会对 `THREE.Scene` 进行详细介绍。示例中所要渲染的方块和球体稍后将会添加到场景对象中。

在示例中我们还定义了摄像机对象，摄像机决定了能够在场景看到什么。在第 2 章，你还会进一步了解摄像机对象能够接受的参数。接下来，我们定义了渲染器对象，该对象会基于摄像机的角度来计算场景对象在浏览器中会渲染成什么样子。最后 `WebGLRenderer` 将会使用电脑显卡来渲染场景。



如果你已经看了 Three.js 的源码和文档（网址为 <http://threejs.org>），你会发现除了基于 WebGL 的渲染器外，还有其他的渲染器，比如基于 HTML canvas 的渲染器、基于 CSS 的渲染器，甚至还有基于 SVG 的渲染器。尽管它们也能够渲染简单的场景，但是不推荐使用，因为它们已经停止更新、十分耗 CPU 的资源，而且也缺乏对一些功能的支持，比如材质和阴影。

在示例中，我们调用 `setClearColor` 方法将场景的背景颜色设置为接近黑色 (`new THREE.Color(0X00000000)`)，并通过 `setSize` 方法设置场景的大小。使用 `window.innerWidth` 和 `window.innerHeight` 可将整个页面窗口指定为渲染区域。

到目前为止，我们已经创建了空白的场景、渲染器和摄像机，但是还没有渲染任何东西。接下来将会在代码中添加轴和平面：

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

var planeGeometry = new THREE.PlaneGeometry(60, 20);
var planeMaterial = new THREE.MeshBasicMaterial({
  color: 0xAAAAAA
});
var plane = new THREE.Mesh(planeGeometry, planeMaterial);
plane.rotation.x = -0.5 * Math.PI;
plane.position.set(15, 0, 0);
scene.add(plane);
```

如代码所示，我们创建了坐标轴 (`axes`) 对象并设置轴线的粗细值为 20，最后调用 `scene.add` 方法将轴添加到场景中。接下来要创建平面 (`plane`)，平面的创建分为两步来完成。首先，使用 `THREE.Plane Geometry (60,20)` 来定义平面的大小，在示例中将宽度设置为 60，高度设置为 20。除了设置高度和宽度，我们还需要设置平面的外观（比如颜色和透明度），在 Three.js 中通过创建材质对象来设置平面的外观。在本例中，我们将会创建颜色为 `0xAAAAAA` 的基本材质 (`THREE.MeshBasicMaterial`)。然后，将大小和外观组合进 `Mesh` 对象并赋值给平面变量。在将平面添加到场景之前，还需要设置平面的位置：先将平面围绕 `x` 轴旋转 90 度，然后使用 `position` 属性来定义其在场景中的位置。如果你对 `Mesh` 对象感兴趣，那么可以查看第 2 章中的示例 `06-mesh-properties.html`，该示例详细介绍了旋转和定位。最后，我们将平面添加到场景中。

使用同样的方式将方块和球体添加到平面中，但是需要将线框 (`wireframe`) 属性设置为 `true`，这样物体就不会被渲染为实体物体。接下来就是示例的最后一部分：

```
camera.position.set(-30, 40, 30);
camera.lookAt(scene.position);

document.getElementById("webgl-output").appendChild(renderer.domElement);
renderer.render(scene, camera);
```

现在，所有物体都已经添加到场景中的合适位置。在之前我们提到过，摄像机将决定

哪些东西会被渲染到场景中。在这段代码中，我们使用 x、y、z 的位置属性来设置摄像机的位置。为了确保所要渲染的物体能够被摄像机拍摄到，我们使用 lookAt 方法指向场景的中心，默认状态下摄像机是指向 (0,0,0) 位置的。最后需要做的就是将渲染的结果添加到 HTML 框架的 <div> 元素中。我们使用 JavaScript 来选择需要正确输出的元素并使用 appendChild 方法将结果添加到 div 元素中。最后告诉渲染器使用指定的摄像机来渲染场景。

接下来，我们还会使用光照、阴影、材质和动画来美化这个场景。

## 1.5 添加材质、光源和阴影效果

在 Three.js 中添加材质和光源是非常简单的，做法和前一节讲的基本一样。首先我们在场景中添加一个光源（完整代码请参见示例 js/03-03.js）。代码如下所示：

```
var spotLight = new THREE.SpotLight(0xFFFFFF);
spotLight.position.set(-40, 40, -15);
spotLight.castShadow = true;
spotLight.shadow.mapSize = new THREE.Vector2(1024, 1024);
spotLight.shadow.camera.far = 130;
spotLight.shadow.camera.near = 40;
```

通过 THREE.SpotLight 定义光源并从其位置 (spotLight.position.set (-40,60,-10)) 照射场景。通过将 castShadow 属性设置为 true，THREE.js 的阴影功能被启用。此外，上面的代码还通过设置 shadow.mapSize、shadow.camera.far 和 shadow.camera.near 三个参数来控制阴影的精细程度。有关光源属性的更多细节将在第 3 章详细介绍。如果这时候渲染场景，那么你看到的结果和没有添加光源时是没有区别的。这是因为不同的材质对光源的反应是不一样的。我们使用的基本材质 (THREE.MeshBasicMaterial) 不会对光源有任何反应，基本材质只会使用指定的颜色来渲染物体。所以，我们需要改变平面、球体和立方体的材质：

```
var planeGeometry = new THREE.PlaneGeometry(60, 20);
var planeMaterial = new THREE.MeshLambertMaterial({color:
    0xffffffff});
var plane = new THREE.Mesh(planeGeometry, planeMaterial);
...
var cubeGeometry = new THREE.BoxGeometry(4, 4, 4);
var cubeMaterial = new THREE.MeshLambertMaterial({color:
    0xffff0000});
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);
```

如代码所示，我们将场景中物体的材质改为 MeshLambertMaterial。Three.js 中的材质 MeshPhysicalMaterial 和 MeshStandardMaterial（以及被弃用的 MeshPhongMaterial）在渲染时会对光源产生反应。

渲染的结果如图 1.12 所示，但是这还不是我们想要的结果。

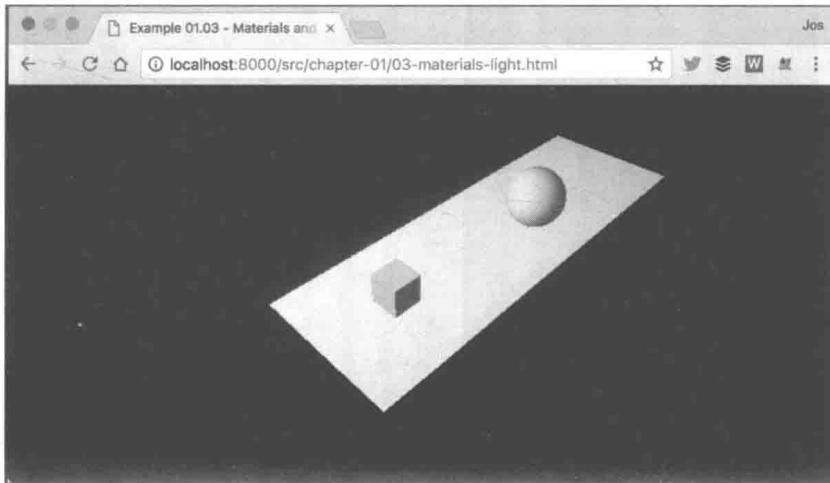


图 1.12

虽然立方体和球体已经好看了很多，但是还缺少阴影的效果。

由于渲染阴影需要耗费大量的计算资源，所以默认情况下 Three.js 中是不会渲染阴影的。为了渲染阴影效果，我们需要对代码做如下修改：

```
renderer.setClearColor(new THREE.Color(0x000000));
renderer.setSize(window.innerWidth, window.innerHeight);
renderer.shadowMap.Enabled = true;
```

首先通过设置 shadowMapEnabled 属性为“true”来告诉渲染器需要阴影效果。这时候如果查看修改的效果，那么你将会发现还是没有任何区别，因为你还需要明确地指定哪个物体投射阴影、哪个物体接受阴影。在示例中，我们通过将相应的属性设置为“true”来指定球体和立方体在地面上投射阴影。代码如下所示：

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

接下来我们还需要定义能够产生阴影的光源。因为并不是所有的光源都能够产生阴影，但是通过 THREE.SpotLight 定义的光源是能够产生阴影的。我们只要将属性 castShadow 设置为 true 就可以将阴影渲染出来了，代码如下所示：

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

这样，场景中就有了光源产生的阴影，效果如图 1.13 所示。



仔细观察 01-03.js 的源代码会发现，这段程序还创建了一个拥有不同物体的场景。若将那些名为 createXXX 的函数的注释去掉，并且删除前面已经创建的立方体和球，场景中便会出现一些更复杂的物体。这些物体将展示出如图 1.13 所示的更复杂的阴影。

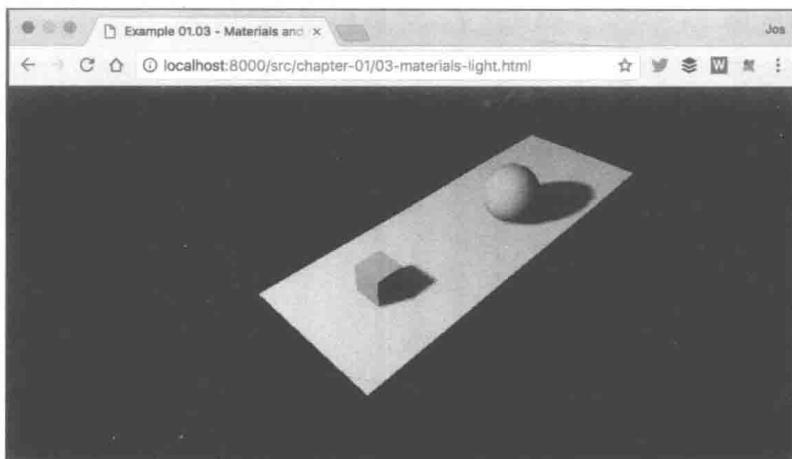


图 1.13

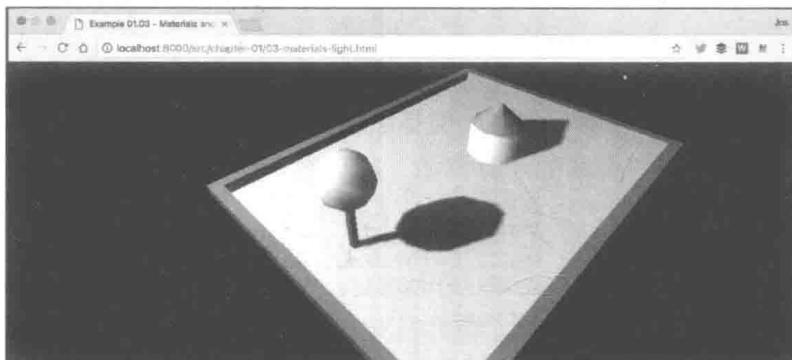


图 1.14

最后一个将要添加到场景中的效果是简单动画。在第 9 章中，你将会学习使用更高级的动画。

## 1.6 让你的场景动起来

如果希望我们的场景动起来，那么首先需要解决的问题是如何在特定的时间间隔重新渲染场景。在 HTML5 和相关的 JavaScript API 出现之前，是通过使用 `setInterval(function,interval)` 方法来实现的。比如，通过 `setInterval()` 方法指定某个函数每 100 毫秒调用一次。但是这个方法的缺点在于它不管浏览器当前正在发生什么（比如正浏览其他网页），它都会每隔几毫秒执行一次。除此之外，`setInterval()` 方法并没与屏幕的刷新同步。这将会导致较高的 CPU 使用率和性能不良。

### 1.6.1 引入 `requestAnimationFrame()` 方法

幸运的是，现代浏览器通过 `requestAnimationFrame` 函数为稳定而连续的渲染场景提供

了良好的解决方案。通过这个函数，你可以向浏览器提供一个回调函数。你无须定义回调间隔，浏览器将自行决定最佳回调时机。你需要做的是在这个回调函数里完成一帧绘制操作，然后将剩下的工作交给浏览器，它负责使场景绘制尽量高效和平顺地进行。这个功能使用起来也非常简单（完整源码在04-04.js文件中），你只需要创建负责绘制场景的回调函数：

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

在renderScene()方法中，requestAnimationFrame()方法又一次被调用了，这样做的目的是保证动画能够持续运行。接下来我们还需要对代码做的修改是：在场景创建完毕后，不再调用renderer.render()方法，而是调用renderScene()来启动动画，代码如下：

```
...
document.getElementById("webgl-output")
.appendChild(renderer.domElement);
renderScene();
```

这时候如果运行代码，效果和之前的示例相比没有任何区别，这是因为我们还没有为物体添加任何动画效果。在添加动画之前，我们先来介绍一个辅助库，这个库也是Three.js作者开发的，主要用于检测动画运行时的帧数。在动画运行时，该库可以在一个图片中显示画面每秒传输帧数。

为了能够显示帧数，首先我们需要在HTML的<head>标签中引入这个辅助库：

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

然后初始化帧数统计模块并将它添加到页面上。

```
function initStats(type) {
    var panelType = (typeof type !== 'undefined' && type) && (!isNaN(type))
        ? parseInt(type) : 0;
    var stats = new Stats();

    stats.showPanel(panelType); // 0: fps, 1: ms, 2: mb, 3+: custom
    document.body.appendChild(stats.dom);
    return stats;
}
```

上面的函数初始化帧数统计模块，并用输入的type参数来设置将要显示的统计内容。它可以是：每秒渲染的帧数、每渲染一帧所花费的时间或者内存占用量。最后，在前面介绍过的init函数末尾调用上述函数来初始化统计模块。

```
function init() {
    var stats = initStats();
    ...
}
```



由于initStats函数并不是本章示例所特有的操作，因此它的实现代码并不在本章的示例代码中，而是像其他有用的辅助函数一样保存在辅助函数库util.js文件中。

辅助函数库在 HTML 页面代码的开始部分被引用。

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

示例代码为统计帧数显示所做的最后一件事，是在 renderScene 函数中每渲染完一帧后，调用 stats.update 函数更新统计。

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

添加完上述代码之后再次运行示例，统计图形将会显示在浏览器左上方，如图 1.15 所示。

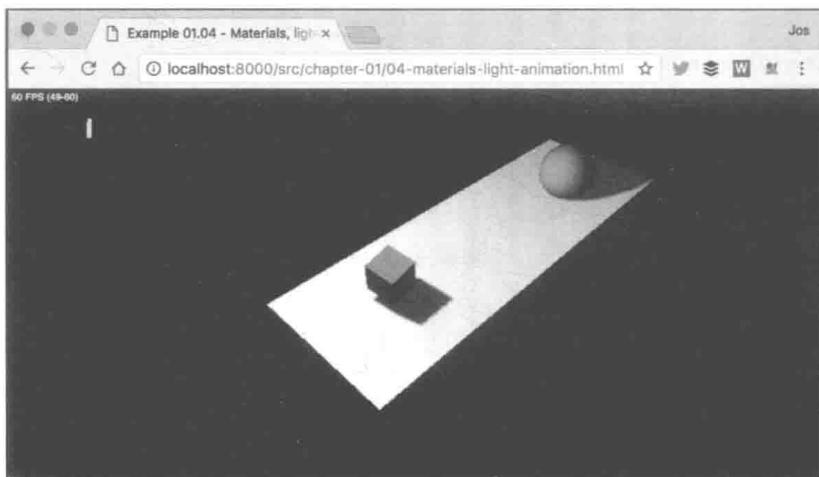


图 1.15

## 1.6.2 旋转立方体

引入 requestAnimationFrame() 方法并配置完统计对象之后，接下来就该添加动画代码了。在本节中，我们将扩展 renderScene() 方法来实现红色立方体围绕轴进行旋转。代码如下所示：

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

看起来是不是很简单？我们所做的只是在每次调用 renderScene() 时使得每个坐标轴的 rotation 属性增加 0.02，其效果就是立方体将围绕它的每个轴进行缓慢的旋转。在下一节中我们将让蓝色球体弹跳起来，这个实现起来也不是特别难。

### 1.6.3 弹跳球

为了让小球弹跳起来，只需要在 renderScene() 方法中添加如下代码即可：

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

旋转立方体时我们修改的是 rotation 属性；而让小球弹跳起来，我们所要修改的是球体在场景中的位置（position）属性。我们的目的是让小球依照一条好看的、光滑的曲线从一个地方跳到另一个地方，如图 1.16 所示。

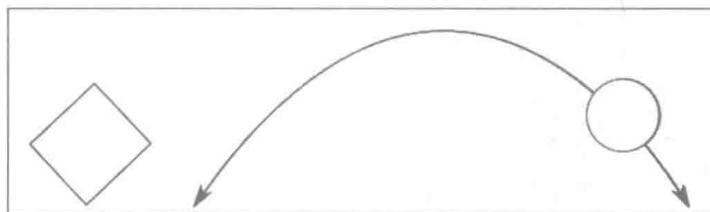


图 1.16

为了实现这个效果，我们需要同时改变球体在 x 轴和 y 轴的位置。Math.cos() 和 Math.sin() 方法使用 step 变量就可以创建出球体的运行轨迹。在这里不具体展开解释它们是怎么工作的，现在你只需要知道 step+=0.04 定义了球体弹跳的速度就可以，在第 8 章中还会详细介绍这些方法是如何用于实现动画的。图 1.17 展示的就是球体在弹跳中的效果。

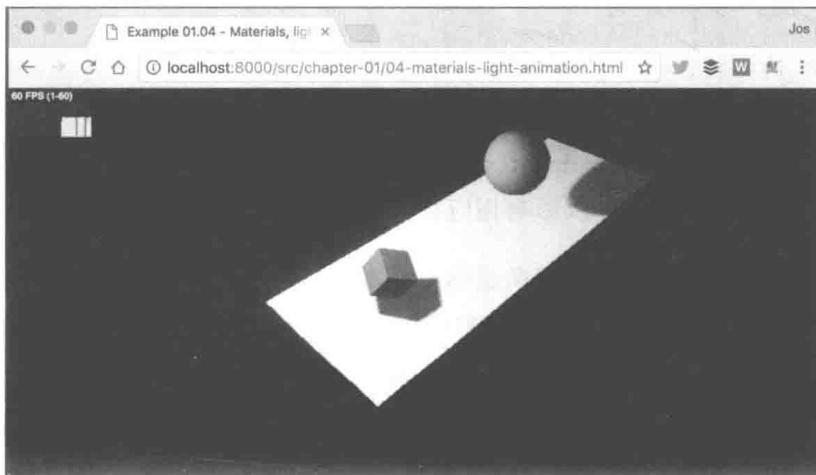


图 1.17

在结束本章之前，我还想在场景中引入几个辅助库。可能你已经发现，在创建类似三维场景和动画时，我们需要尝试很多次才能够确定最合适的速度和颜色。如果有一个 GUI（可视化图形界面）允许我们在运行期间修改这些属性，那么事情就会变得很简单。幸运的是，这样的 GUI 是存在的。

## 1.7 使用 dat.GUI 简化试验流程

Google 员工创建了名为 dat.GUI 的库（相关文档见 <http://code.google.com/p/dat-gui/>），使用这个库可以很容易地创建出能够改变代码变量的界面组件。在本章最后，将会使用 data.GUI 库为我们的示例添加用户操作界面，使得我们可以：

- 控制小球弹跳的速度；
- 控制立方体的旋转。

就像引入统计对象一样，首先我们需要在 HTML 的 <head> 标签中添加这个库，代码如下所示：

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

接下来我们需要定义一个 JavaScript 对象，该对象将保存希望通过 dat.GUI 改变的属性。在 JavaScript 代码中添加如下的 JavaScript 对象：

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

在这个 JavaScript 对象中，我们定义了两个属性——this.rotationSpeed 和 this.bouncingSpeed，以及它们的默认值。接下来需要将这个 JavaScript 对象传递给 data.GUI 对象，并设置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);
```

立方体旋转速度（rotationSpeed）和球体弹跳速度（bouncingSpeed）的取值范围为 0~0.5。现在需要做的就是在 renderScene() 中直接引用这两个属性，这样当我们在 dat.GUI 中修改这两个属性的值时，就可以影响相应物体的旋转速度和弹跳速度。代码如下所示：

```
function renderScene() {
  ...
  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，你就会看到一个可以控制弹跳速度和旋转速度的用户界面。如图 1.18 所示。

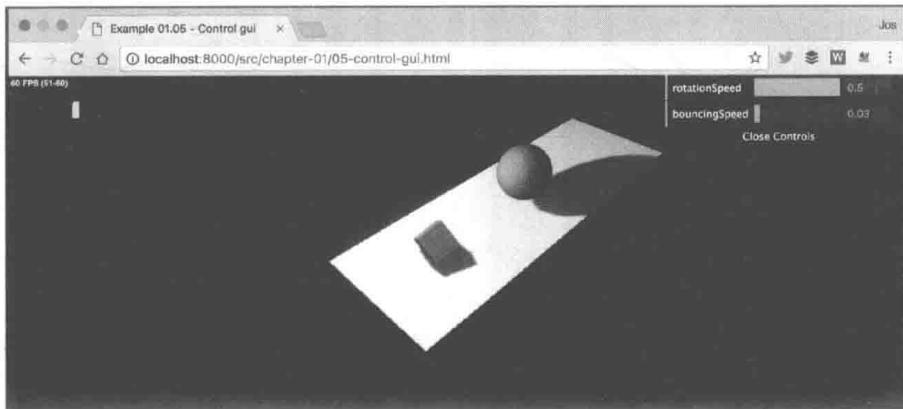


图 1.18

读者应该还记得在本章开头时创建的框架页面代码中，我们引用了 `TrackballControl.js` 文件。该文件用于实现利用鼠标移动摄像机，以便以不同角度观察场景。这一点将在第 9 章详细介绍。该文件同样需要初始化。由于它需要响应文档对象模型（DOM）元素的事件，它的初始化代码必须出现在下面代码中的 `appendChild` 函数调用之后。

```
document.getElementById("webgl-output").appendChild(renderer.domElement);
// add the two lines below
var trackballControls = initTrackballControls(camera, renderer);
var clock = new THREE.Clock();
```

`initTrackballControls` 函数也定义于 `util.js` 文件中，它的具体实现将在本书后面章节中详细介绍。最后，与帧数统计模块相似，`TrackballControl.js` 库也需要在 `render` 函数渲染完一帧后更新。

```
function render() {
    trackballControls.update(clock.getDelta());
    ...
}
```

至此本章的代码便已完成。再次打开 `05-control-gui.html` 文件时，可以通过按下鼠标左键并移动鼠标来转动摄像机，以便从不同角度观察场景。此外，按 S 键可以拉近或拉远摄像机，按 D 键可以平移摄像机。如图 1.19 所示。

后续章节中的所有实践都将使用这个库来移动摄像机。

在运行这个示例并改变浏览器大小的时候，你可能已经发现场景是不会随着浏览器大小的改变而自动做出调整的。那么在接下来的小节中，我们会添加本章的最后一个特性来解决这一问题。

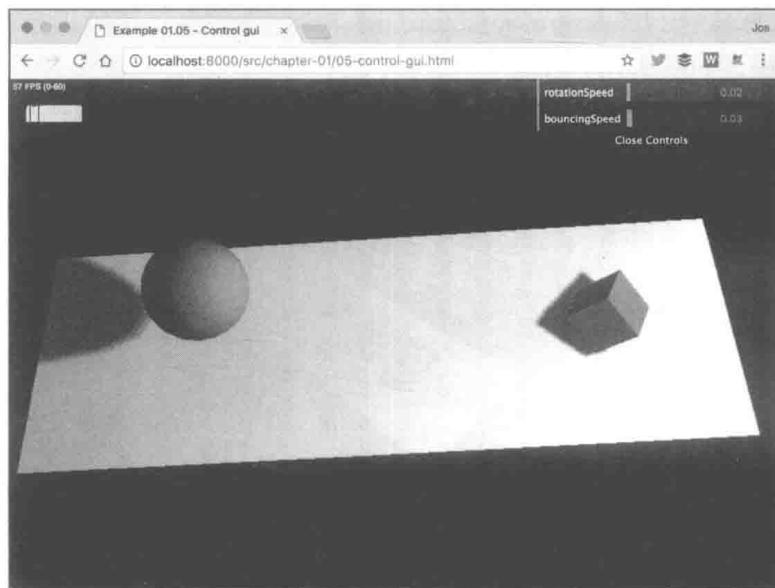


图 1.19

## 1.8 场景对浏览器的自适应

当浏览器大小改变时改变摄像机是很容易实现的。首先我们需要做的就是为浏览器注册一个事件监听器，如下所示：

```
window.addEventListener('resize', onResize, false);
```

这样，每当浏览器尺寸改变时 `onResize()` 方法就会被执行。在 `onRaise()` 方法中需要更新摄像机和渲染器，代码如下：

```
function onResize() {
    camera.aspect = window.innerWidth / window.innerHeight;
    camera.updateProjectionMatrix();
    renderer.setSize(window.innerWidth, window.innerHeight);
}
```

对于摄像机，需要更新它的 `aspect` 属性，这个属性表示屏幕的长宽比；对于渲染器，需要改变它的尺寸。最后我们需要将变量 `camera`、`renderer` 和 `scene` 的定义移到 `init()` 方法的外面，这样其他的方法（如 `onResize()` 方法）也可以访问它们。代码如下所示：

```
var camera;
var scene;
var renderer;

function init() {
    ...
    scene = new THREE.Scene();
    camera = new THREE.PerspectiveCamera(45, window.innerWidth /
        window.innerHeight, 0.1, 1000);
```

```
renderer = new THREE.WebGLRenderer();  
***  
}
```

运行示例 06-screen-size-change.html 并改变浏览器的大小，就可以看到实际的效果。

## 1.9 总结

本章到此结束。在这一章里你学到了如下内容：如何搭建开发环境，如何获取本书相关的源码和示例；使用 Three.js 渲染场景时，首先需要做的是创建 THREE.Scene 对象，添加摄像机、光源和需要渲染的物体；如何给场景添加阴影和动画效果；添加辅助库 dat.GUI 和 stats.js 创建用户控制界面和快速获取场景渲染时的帧数。

在下一章中我们将会进一步扩展这个示例，你将会学到更多 Three.js 中非常重要的场景构建模块。

## 构建 Three.js 应用的基本组件

在第 1 章中我们介绍了 Three.js 库的基础知识。通过示例展示了 Three.js 是如何工作的，然后创建了第一个完整的 Three.js 应用。在本章中我们将会深入了解 Three.js 库，介绍构建 Three.js 应用的基本组件。通过本章你将了解以下内容：

- 在 Three.js 应用中使用的主要组件。
- THREE.Scene 对象的作用。
- 几何图形和网格是如何关联的。
- 正交投影摄像机和透视投影摄像机的区别。

我们首先来介绍如何创建场景并添加对象。

### 2.1 创建场景

在第 1 章中我们已经创建了一个 THREE.Scene，想必你已经了解了 Three.js 库的基础知识。我们可以看到，一个场景想要显示任何东西，需要表 2.1 所示三种类型的组件。

表 2.1

组 件	说 明
摄像机	决定屏幕上哪些东西需要渲染
光源	决定材质如何显示以及用于产生阴影（细节部分将在第 3 章中讨论）
对象	它们是摄像机透视图里主要的渲染对象，如方块、球体等
渲染器	基于摄像机和场景提供的信息，调用底层图形 API 执行真正的场景绘制工作

THREE.Scene 对象是所有不同对象的容器，但这个对象本身没有那么多的选项和函数。



THREE.Scene 对象有时被称为场景图，可以用来保存所有图形场景的必要信息。在 Three.js 中，这意味着 THREE.Scene 保存所有对象、光源和渲染所需的其他对象。有趣的是，场景图，顾名思义，不仅仅是一个对象数组，还包含了场景图树形结构中的所有节点。每个你添加到 Three.js 场景的对象，甚至包括 THREE.Scene 本身，都是继承自一个名为 THREE.Object3D 的对象。一个 THREE.Object3D 对象也可以有自己的子对象，你也可以使用它的子对象来创建一个 Three.js 能解释和渲染的对象树。

### 2.1.1 场景的基本功能

了解一个场景功能的最好方法就是看示例。在本章的源代码中，你可以找到一个名为 01-basic-scene.html 的例子。我将使用这个例子来解释一个场景所拥有的各种方法和选项。当我们在浏览器中打开这个示例的时候，其效果大致如图 2.1 所示。请记住除了鼠标之外，键盘上的 A、S 和 D 键也可用于在渲染场景中转向、缩放和平移。



图 2.1

这跟我们在第 1 章中看到的例子非常像。尽管这个场景看上去有点儿空荡荡，但其实它已经包含了好几个对象。通过下面的源代码可以看到，我们使用 THREE.Scene 对象的 `scene.add(object)` 方法添加了一个 THREE.Mesh 对象（你看到的平面）、一个 THREE.SpotLight 对象（聚光灯光源）和一个 THREE.AmbientLight 对象（环境光）。渲染场景的时候，THREE.Camera 对象会自动地被 Three.js 添加进来。但是我们手动添加它会是一个更好的实践，尤其是当你需要处理多个摄像机的时候。部分源码如下：

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(45, window.innerWidth
    / window.innerHeight, 0.1, 100);
scene.add(camera);
```

```

...
var planeGeometry = new THREE.PlaneGeometry(60, 40, 1, 1);
var planeMaterial = new THREE.MeshLambertMaterial({color: 0xffffffff});
var plane = new THREE.Mesh(planeGeometry, planeMaterial);
...
scene.add(plane);
var ambientLight = new THREE.AmbientLight(0x3c3c3c);
scene.add(ambientLight);
...
var spotLight = new THREE.SpotLight(0xffffffff, 1.2, 150, 120);
spotLight.position.set(-40, 60, -10);
spotLight.castShadow = true;
scene.add(spotLight);

```

在深入了解 THREE.Scene 对象之前，先说明一下你可以在这个示例中做些什么，然后我们再来看代码。在浏览器中打开 01-basic-scene.html 示例，看看右上角的那些控件。如图 2.2 所示。

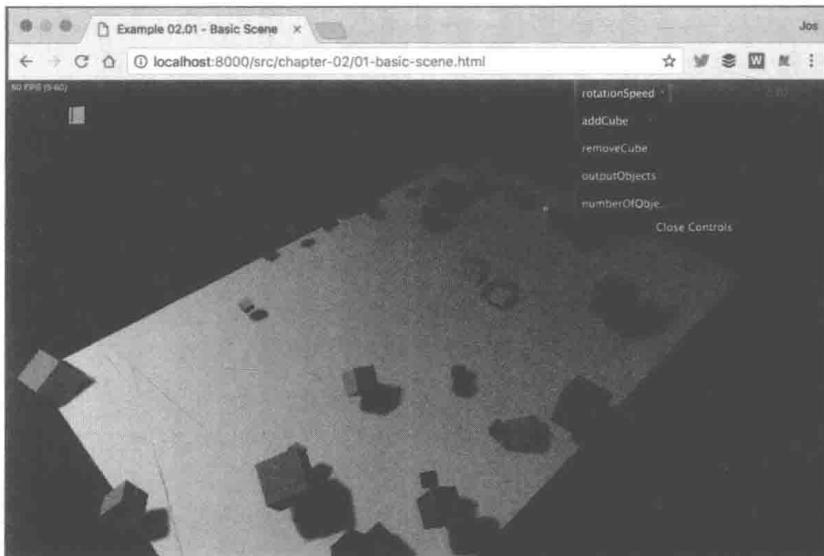


图 2.2

通过这些控件，你可以往场景中添加方块、移除最后一个添加到场景中的方块以及在浏览器控制台中显示当前场景中的所有对象。控件区的最后一项显示了当前场景中所有对象的数量。你可能会发现场景在启动的时候就已经包含了 4 个对象，它们是地面、环境光、点光源和我们之前提到的摄像机。让我们从最简单的 addCube 方法开始逐一了解控件区的所有方法。

```

this.addCube = function() {

  var cubeSize = Math.ceil((Math.random() * 3));
  var cubeGeometry = new THREE.BoxGeometry(cubeSize, cubeSize, cubeSize);
  var cubeMaterial = new THREE.MeshLambertMaterial({color: Math.random() *
  0xffffffff });
}

```

```

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，因为这里没有引入很多新的概念。当你点击 addCube 按钮的时候，一个新的 THREE.BoxGeometry 对象就会被创建，它的长、宽和高都是一个从 1 到 3 之间的随机数。除了尺寸是随机的，这个方块的颜色和位置也是随机的。



这段代码里的新东西是我们使用 name 属性为整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 scene.children.length）。给对象命名在调试的时候是很有用的，而且还可以直接通过名字来获取场景中的对象。如果使用 Three.Scene.getObjectByName(name) 方法，可以直接获取场景中名为 name 的对象，然后可以执行一些比如改变位置的操作。你或许想知道最后一行代码的作用，我们的控制界面就是使用 numberOfObjects 变量来显示场景中对象数量的。所以，无论什么时候添加或者删除对象，我们都要将该变量设置为更新后的数量。

在控制界面中能够调用的另一个方法是 removeCube()。顾名思义，点击 removeCube 按钮，将会移除最后一个添加到场景中的方块。代码实现如下所示：

```

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;
  }
}

```

在场景中添加对象使用的是 add() 方法，而从场景中移除对象就要使用 remove() 方法。由于 Three.js 将子对象保存在数组中（最新的对象保存在数组的最后），所以我们可以使用 THREE.Scene 对象的 children 属性来获取最后一个添加到场景中的对象，children 属性将场景中的所有对象存储为数组。在移除对象时，我们还需要检查该对象是不是 THREE.Mesh 对象，这样做的原因是避免移除摄像机和光源。当我们移除对象之后，需要再次更新控制界面中表示场景中对象数量的属性 numberOfObjects。

控制界面上的最后一个按钮的标签是 outputObjects。你或许已经发现在点击该按钮后什么都没有发生。因为这个按钮的功能是在浏览器的控制台中打印出场景中的所有对象信息，如图 2.3 所示。

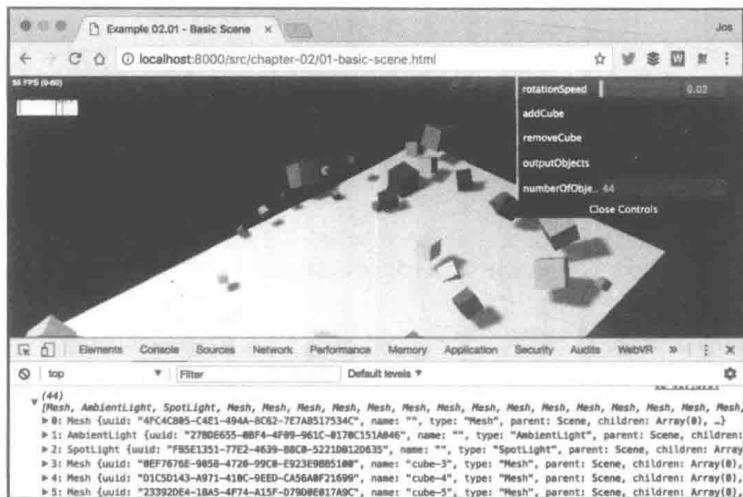


图 2.3

我们使用的是内置的 `console` 对象在浏览器控制台日志中输出对象信息，代码如下所示：

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

这样做对于代码调试是非常有用的，尤其是当你为对象命名时。它对查找某个特定对象相关的问题是非常有用的。例如，对象 cube-17（如果你已经知道对象的名字，就可以使用 `console.log(scene.getObjectByName("cube-17"))` 来输出对应的信息。输出结果如图 2.4 所示。

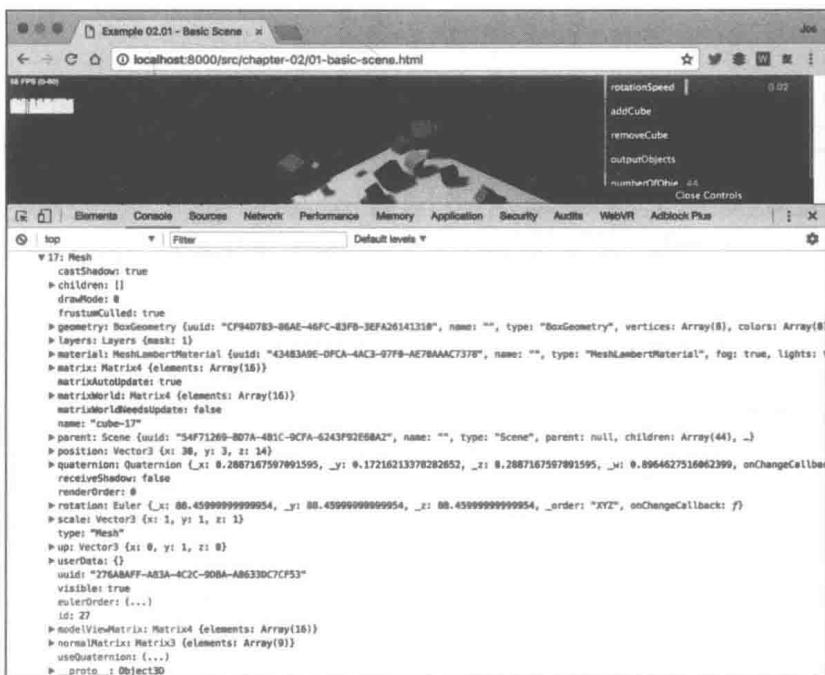


图 2.4

目前为止，我们已经学习了如下一些和场景相关的方法：

- THREE.Scene.Add：用于向场景中添加对象
- THREE.Scene.Remove：用于移除场景中的对象
- THREE.Scene.children：用于获取场景中所有的子对象列表
- THREE.Scene.getObjectByName：利用 name 属性，用于获取场景中特定的对象

这些方法是和场景相关的重要方法，通常情况下这些方法就可以满足大部分需求了。但是，还有几个辅助方法可能会被用到，请看下面的示例代码片段。

如你在第1章中所看到的，我们使用了 render 循环来渲染场景。该循环代码如下所示：

```
function render() {
  stats.update();
  scene.traverse(function( + + 6 + obj) {
    if (obj instanceof THREE.Mesh && obj != plane ) {
      obj.rotation.x+=controls.rotationSpeed;
      obj.rotation.y+=controls.rotationSpeed;
      obj.rotation.z+=controls.rotationSpeed;
    }
  });
  requestAnimationFrame(render);
  renderer.render(scene, camera);
}
```

在这里，我们使用了 THREE.Scene.traverse() 方法。我们可以将一个方法作为参数传递给 traverse() 方法，这个传递来的方法将会在每一个子对象上执行。由于 THREE.Scene 对象存储的是对象树，所以如果子对象本身还有子对象，traverse() 方法会在所有的子对象上执行，直到遍历完场景树中的所有对象为止。

我们使用 render() 方法来更新每个方块的旋转状态（我们特意忽略了表示地面的 plane 对象）。我们还可以使用 for 循环或者 forEach 来遍历 children 属性数组来达到同样的目的，因为只向 THREE.Scene 增加对象且没有创建嵌套结构。

在我们深入讨论 THREE.Mesh 和 THREE.Geometry 对象之前，先来介绍 THREE.Scene 对象的两个属性：fog（雾化）和 overrideMaterial（材质覆盖）。

## 2.1.2 给场景添加雾化效果

使用 fog 属性就可以为整个场景添加雾化效果。雾化效果是：场景中的物体离摄像机越远就会变得越模糊。如图 2.5 所示。

为了更好地观察雾化效果，可以利用鼠标推进或拉远摄像机，这样就可以观察物体是如何受雾化效果的影响。在 Three.js 中为场景添加雾化效果是很简单的，在定义完场景后只要添加如下代码即可：

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

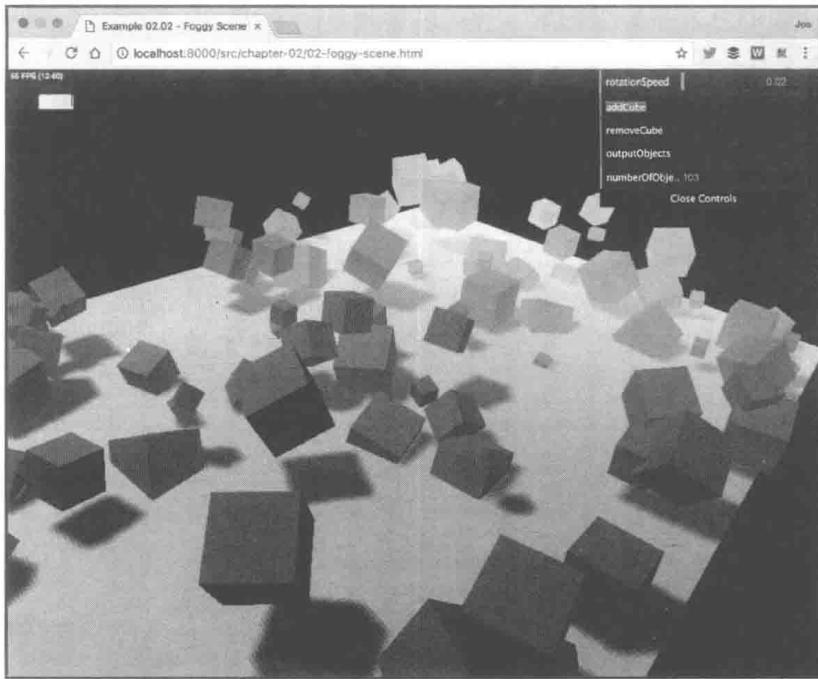


图 2.5

我们在这里定义一个白色雾化效果（0xffffffff）。后面的两个参数是用来调节雾的显示，0.015 是 near（近处）属性的值，100 是 far（远处）属性的值。通过这两个属性可以决定雾化开始和结束的地方，以及加深的程度。使用 THREE.Fog 创建的对象，雾的浓度是线性增长的，除此之外还有另外一种添加雾化效果的方法，定义如下：

```
scene.fog = new THREE.FogExp2( 0xffffffff, 0.01 );
```

在这个方法中不再指定 near 和 far 属性，只需要设置雾的颜色（0xffffffff）和浓度（0.01）即可。需要注意的是，该方法中雾的浓度不再是线性增长的，而是随着距离呈指数增长。

### 2.1.3 使用 overrideMaterial 属性

我们要介绍的最后一个场景属性是 overrideMaterial。当设置了 overrideMaterial 属性后，场景中所有的物体都会使用该属性指向的材质，即使物体本身也设置了材质。当某一个场景中所有物体都共享同一个材质时，使用该属性可以通过减少 Three.js 管理的材质数量来提高运行效率，但是实际应用中，该属性通常并不非常实用。该属性的使用方法如下所示：

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

使用了上述代码中显示的 overrideMaterial 属性之后，场景的渲染结果如图 2.6 所示。



图 2.6

从图中可以看出，所有的立方体都使用相同的材质和颜色进行渲染。在这个示例中，我们用的材质是 THREE.MeshLambertMaterial，而且使用该材质类型，还能够创建出不发光但是可以对场景中的光源产生反应的物体。在第 4 章中你将会学到更多关于这种材质的内容。

在本节中我们介绍了 Three.js 中最核心的概念：场景。关于场景我们需要记住的是：它是在渲染时你想使用的所有物体、光源的容器。表 2.2 列出了 THREE.Scene 中最常用的方法和属性。

表 2.2

方法(属性)	描述
add(object)	用于向场景中添加对象。使用该方法还可以创建对象组
children	用于返回一个场景中所有对象的列表，包括摄像机和光源
getObjectByName(name, recursive)	在创建对象时可以指定唯一的标识 name，使用该方法可以查找特定名字的对象。当参数 recursive 设置为 false 时，在调用者子元素上查找；当参数 recursive 设置为 true 时，在调用者的所有后代对象上查找
remove(object)	object 为场景中对象的引用，使用该方法可以将对象从场景中移除
traverse(function)	children 属性可以返回场景中的所有物体。该方法也可以遍历调用者和调用者的所有后代，function 参数是一个函数，被调用者和每一个后代对象调用 function 方法
fog	使用该属性可以为场景添加雾化效果，可以产生隐藏远处物体的浓雾效果
overrideMaterial	使用该属性可以强制场景中的所有物体使用相同的材质

在下一节中，我们将会对可以添加到场景中的物体作详细介绍。

## 2.2 几何体和网格

在前面的例子中我们已经使用了几何体和网格。比如在向场景中添加球体时，代码如下所示：

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

我们使用 THREE.SphereGeometry 定义了物体的形状、使用 THREE.MeshBasicMaterial 定义了物体的外观和材质，并将它们合并成能够添加到场景中的网格（THREE.Mesh）。在这一节中将进一步介绍什么是几何体和网格。

### 2.2.1 几何体的属性和方法

Three.js 提供了很多可以在三维场景中使用的几何体。图 2.7 是 04-geometries 示例的截图，该图展示了 Three.js 库中可用的标准几何体。

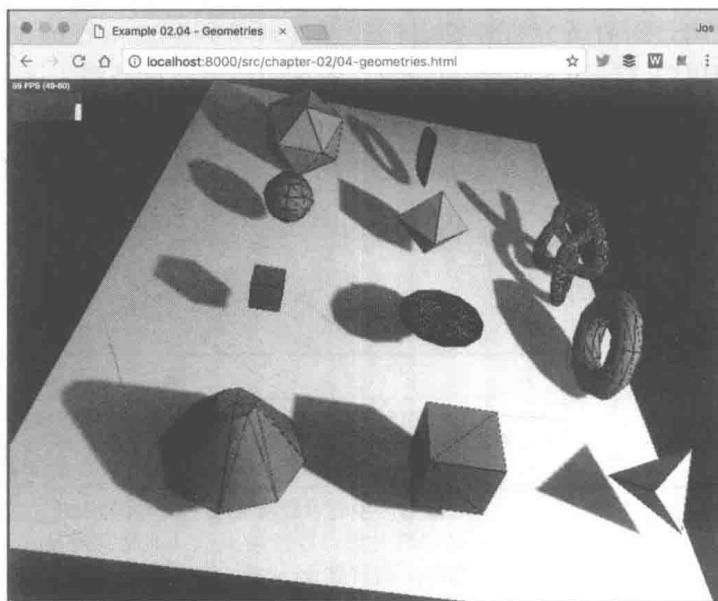


图 2.7

在第 5 章和第 6 章中我们将会深入讨论 Three.js 提供的所有基本几何体和高级几何体。在这一节中，我们主要介绍什么是几何体。

像其他大多数三维库一样，在 Three.js 中几何体基本上是三维空间中的点集（也被称作顶点）和将这些点连接起来的面。以立方体为例：

- 一个立方体有 8 个角。每个角都可以用 x、y 和 z 坐标点来定义，所以每个立方体在三维空间中都有 8 个点。在 Three.js 中，这些点称为顶点。

- 一个立方体有6个面，每个角有一个顶点。在Three.js中，每个面都是包含3个顶点的三角形。所以，立方体的每个面都是由两个三角形面组成的。

当你使用Three.js库提供的几何体时，不需要自己定义几何体的所有顶点和面。对于立方体来说，你只要定义长、宽、高即可。Three.js会基于这些信息在正确的位置创建一个拥有8个顶点和12个三角形面的立方体。尽管可以使用Three.js提供的几何体，但是你仍然可以通过定义顶点和面来自定义创建几何体。创建几何体的方法如下代码所示：

```
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.computeFaceNormals();
```

上述代码展示了如何创建简单的立方体。在vertices数组中保存了构成几何体的顶点，在faces数组中保存了由这些顶点连接起来创建的三角形面。如new THREE.Face3(0,2,1)就是使用vertices数组中的点0、2和1创建而成的三角形面。需要注意的是创建面的顶点时的创建顺序，因为顶点顺序决定了某个面是面向摄像机还是背向摄像机的。如果你想创建面向摄像机的面，那么顶点的顺序是顺时针的，反之顶点的顺序是逆时针的。



在这个示例中，我们使用THREE.Face3元素定义立方体的6个面，也就是说每个面都是由两个三角形面组成的。在Three.js以前的版本中，可以使用四边形来定义面。到底是使用四边形还是三角形来创建面，在三维建模领域里一直存在比较大的争议。基本上，大家都习惯于用四边形来创建面，因为它比三角形更容易增强和平滑。但是对于渲染器和游戏引擎来说，使用三角形更加容易，因为三角形渲染起来效率更高。

有了这些顶点和面，我们就可以创建一个新的 THREE.Geometry 的实例对象，然后将 vertices 数组赋值给 vertices 属性，将 faces 数组赋值给 faces 属性。最后我们需要做的就是在创建的几何体上执行 computeFaceNormals() 方法，当该方法执行时，Three.js 会决定每个面的法向量，法向量用于决定不同光源下的颜色。

基于几何体，我们就可以创建网格了。我已经创建了一个例子，你可以尝试修改顶点的位置来体验一下。在示例 05-custom-geometry.html 中，你可以修改立方体的每个顶点并能够看到相应的面是如何变化的。如图 2.8 所示（按 H 键来隐藏 GUI）。

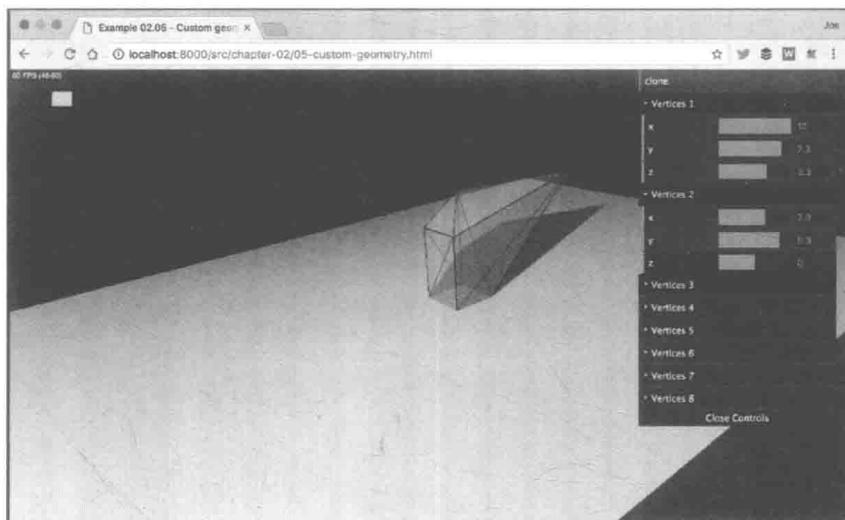


图 2.8

这个示例和其他示例一样，都有一个 render 循环。无论何时修改了顶点的属性，立方体都会基于修改后的值重新进行渲染。出于性能方面的考虑，Three.js 认为组成网格的几何体在整个生命周期内是不会改变的，而且对于大部分的几何体而言，这个假设是成立的。为了使我们的示例工作，我们还需要在 render 循环中添加如下的代码：

```
mesh.children.forEach(function(e) {
  e.geometry.vertices = vertices;
  e.geometry.verticesNeedUpdate = true;
  e.geometry.computeFaceNormals();
});
```

在循环中的第一行，我们将组成网格的几何体的 vertices 属性值指向一个更新后的顶点数组。如果顶点数组 vertices 没有更新，不需要重新配置这些面，因为它们仍然连接到原来的顶点。如果顶点被更新了，我们还需要告诉几何体顶点需要更新，在代码中是将 verticesNeedUpdate 属性设置为 true 来实现这一点的，最后需要调用 computeFaceNormals() 方法来重新计算每个面，从而完成整个模型的更新。

我们将要介绍的最后一个关于几何体的函数是 clone()。我们说过几何体可以定义物体

的形状，添加相应的材质后就可以创建出能够添加到场景中并由Three.js渲染的物体。通过clone()方法我们可以创建出几何体对象的拷贝。为这些拷贝对象添加不同的材质，我们就可以创建出不同的网格对象。在示例05-custom-geometry.html里，你可以在控制界面的顶端看到一个clone按钮。如图2.9所示。

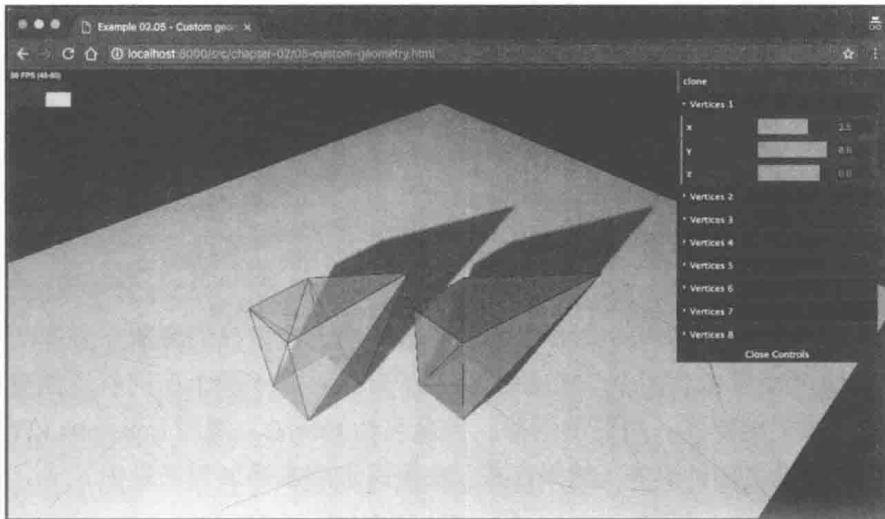


图 2.9

如果点击clone按钮就可以按照几何体当前的状态创建出一个拷贝，而且这个新对象被赋予了新的材质并被添加到场景中。实现这个功能的代码是非常简单的，但是由于我使用的材质导致代码看起来有点复杂。首先我们先看一下绿色材质的实现代码：

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

如你所看到的，我们使用的不是一个材质，而是由两个材质构成的数组。这样做的原因是，除了显示绿色透明的立方体外，我还想显示一个线框。因为使用线框可以很容易地找出顶点和面的位置。当然，Three.js支持使用多种材质来创建网格。你可以使用SceneUtils.createMulti-MaterialObject()方法来达到这个目的。代码如下所示：

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

这个方法创建的并不是一个THREE.Mesh对象实例，而是为materials数组中每个指定的材质创建一个实例，并把这些实例存放在一个组里（THREE.Object3D对象）。你可以像使用场景中的对象那样使用这个组，如添加网格、按名称获取对象等。如果要为这个组中所有的子对象添加阴影，我们可以这样做：

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

现在，我们继续讨论 `clone()` 函数：

```
this.clone = function() {
    var clonedGeom = 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(clonedGeom,
materials);
    mesh2.children.forEach(function(e) {e.castShadow=true});
    mesh2.translateX(5);
    mesh2.translateZ(5);
    mesh2.name="clone";
    scene.remove(scene.getObjectByName("clone"));
    scene.add(mesh2);
}
```

点击 `clone` 按钮，这段 JavaScript 代码就会被调用。这里我们复制立方体的第一个子对象。请记住，`mesh` 变量包含两个 `THREE.Mesh` 子对象：基于两个不同材质创建的。通过这个复制的几何体我们创建了一个新的网格，并命名为 `mesh2`。使用 `translate()` 方法移动这个新创建的网格，删除之前的副本（如果存在）并把这个副本添加到场景中。

 在前面的章节中，我们使用 `THREE.SceneUtils` 对象的 `createMultiMaterialObject()` 方法为几何体添加了线框。在 Three.js 中还可以使用 `THREE.WireframeGeometry` 来添加线框。假设有一个几何体对象名为 `geom`，可以通过下面代码基于 `geom` 创建一个线框对象：

```
var wireframe = new THREE.WireframeGeometry(geom);
```

然后，基于新建的线框对象创建一个 `Three.LineSegments` 对象并将它添加到场景中：

```
var line = new THREE.LineSegments(wireframe);
scene.add(line);
```

最后便可以利用它来绘制线框了，并且还可以像下面代码那样设置线框的宽度：

```
line.material.linewidth = 2;
```

关于 `Three.js` 中几何体的知识，我们暂时就学习到这里。

## 2.2.2 网格对象的属性和方法

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。当网格创建好之后，我们就可以将它添加到场景中并进行渲染。网格对象提供了几个属性用于改变它在场景中的位置和显示效果。下面我们来看下网格对象提供的属性和方法，具体见表 2.3。

表 2.3

方法(属性)	描述
position	该属性决定该对象相对于父对象的位置。通常父对象是 THREE.Scene 对象或者 THREE.Object3D 对象
rotation	通过该属性可以设置绕每个轴的旋转弧度。Three.js 还提供了设置相对特定轴的旋转弧度的方法: rotateX()、rotateY() 和 rotateZ()
scale	通过该属性可以沿着 x、y 和 z 轴缩放对象
translateX(amount)	沿 x 轴将对象平移 amount 距离
translateY(amount)	沿 y 轴将对象平移 amount 距离
translateZ(amount)	沿 z 轴将对象平移 amount 距离。对于平移, 也可使用 translateOnAxis (axis, distance) 函数, 该函数允许沿指定轴平移网格
visible	该属性值为 false 时, THREE.Mesh 将不会被渲染到场景中

我们同样准备了一个示例, 你可以修改这些属性的值来感受下效果。当你在浏览器中打开示例 06-mesh-properties.html 时, 可以看到一个下拉菜单。通过该下拉菜单你就可以修改属性的值, 并立即看到修改后的效果, 如图 2.10 所示。

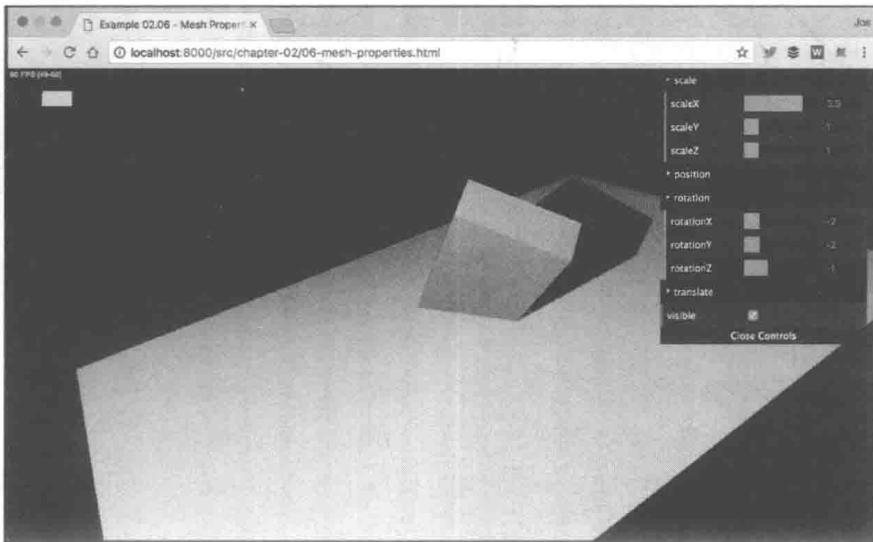


图 2.10

下面来逐一讲解这些属性和方法, 先从 position 属性开始。通过这个属性你可以设置对象在 x、y 和 z 轴的坐标。对象的位置是相对于它的父对象来说的, 通常父对象就是添加该对象的场景, 但有的时候可能是 THREE.Object3D 对象或其他 THREE.Mesh 对象。在第 5 章讨论对象组合时我们再来讨论这个问题。有三种方式用于设置对象的位置。第一种是直接设置坐标, 代码如下所示:

```
cube.position.x = 10;
cube.position.y = 3;
cube.position.z = 1;
```

也可以一次性地设置  $x$ 、 $y$  和  $z$  坐标的值：

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

还有第三种方式。`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该对象：

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

效果图如图 2.11 所示。

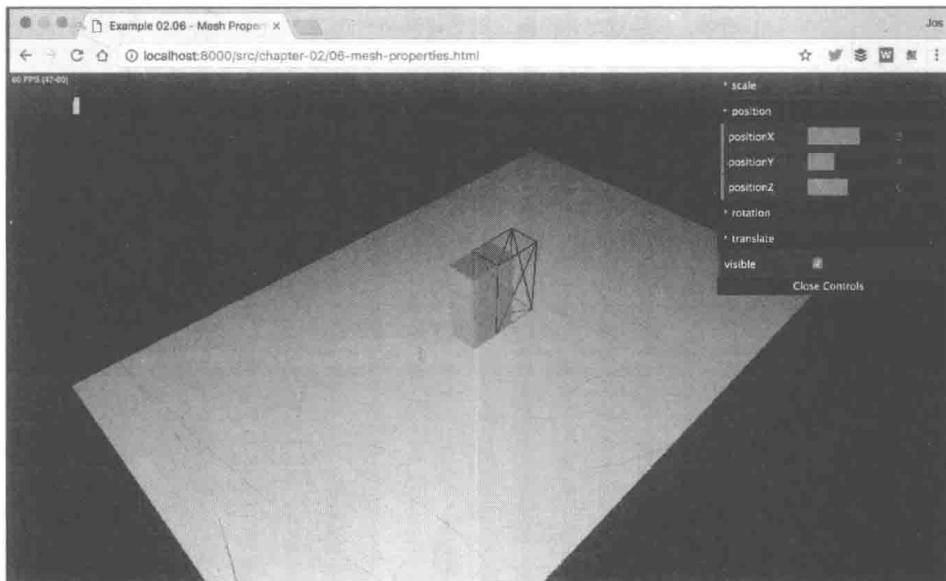


图 2.11

但是如果现在移动该对象组，会发现其位移值保持不变。在第 5 章中我们还会深入讨论这种父子关系，以及对象组是如何影响变换（如缩放、旋转和平移）的。

下一个我们将要介绍的是 `rotation`（旋转）属性。在本章和前一章中我们已经多次使用这个属性了，通过这个属性可以设置对象绕轴的旋转弧度。我们可以像设置 `position` 属性那样来设置 `rotation` 属性。在数学上物体旋转一周的弧度值为  $2\pi$ ，所以可以用如下三种方式设置旋转：

```
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);
```

如果想使用度数（0 到 360）来设置旋转，那么需要对度数做如下转换：

```
Var degrees = 45;
Var inRadians = degrees * (Math.PI / 180);
```

你可以通过示例 06-mesh-properties.html 来体验该属性。

属性列表中还没有讨论的属性是 scale (缩放)。从名字我们就已经知道该属性是用来做什么了。该属性让我们可以沿指定轴缩放对象。如果设置的缩放值小于 1，那么物体就会缩小，如图 2.12 所示。

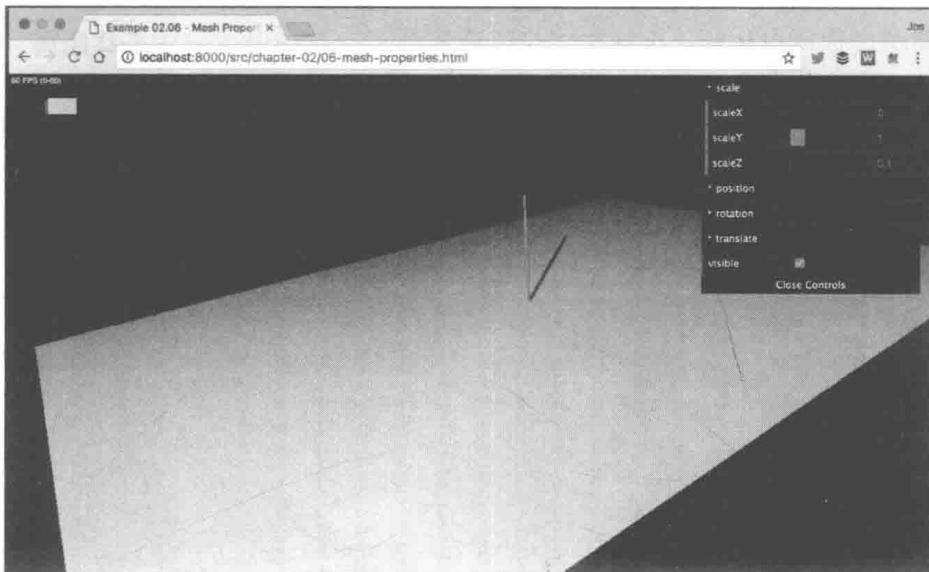


图 2.12

如果设置的缩放值大于 1，那么物体就会变大，如图 2.13 所示。

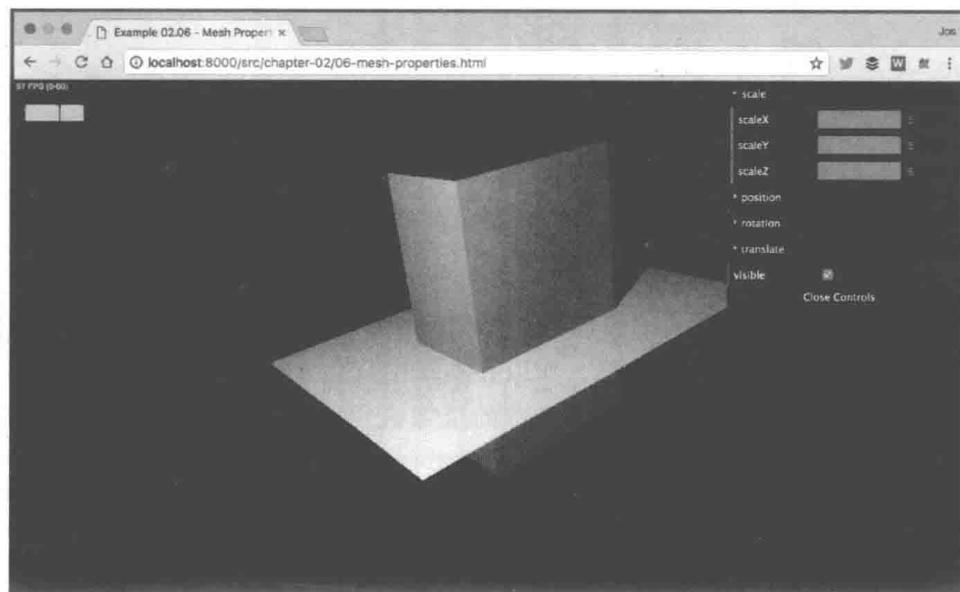


图 2.13

在本章中将要继续讨论的是网格的 `translate()` 方法。使用 `translate()` 方法你可以改变对象的位置，但是该方法设置的不是物体的绝对位置，而是物体相对于当前位置的平移距离。假设在场景中有个球体，其位置为  $(1,2,3)$ 。如果我们想让这个对象相对于  $x$  轴平移 4 个单位：`translateX(4)`，那么物体的位置就会变为  $(5,2,3)$ 。如果我们想重置物体的位置为原来的位置，可以调用 `translateX(-4)`。在示例 `06-mesh-properties.html` 中有个 `translate` 菜单项，通过这个菜单项你可以体验这个功能。只要设置沿  $x$ 、 $y$  和  $z$  轴方向的平移距离，然后点击 `translate` 按钮，你就可以看到物体依照这三个值平移到一个新的位置。

最后一个可以使用的是菜单项右上角的 `visible` 属性。当你点击 `visible` 菜单项时，会发现立方体消失了，如图 2.14 所示。

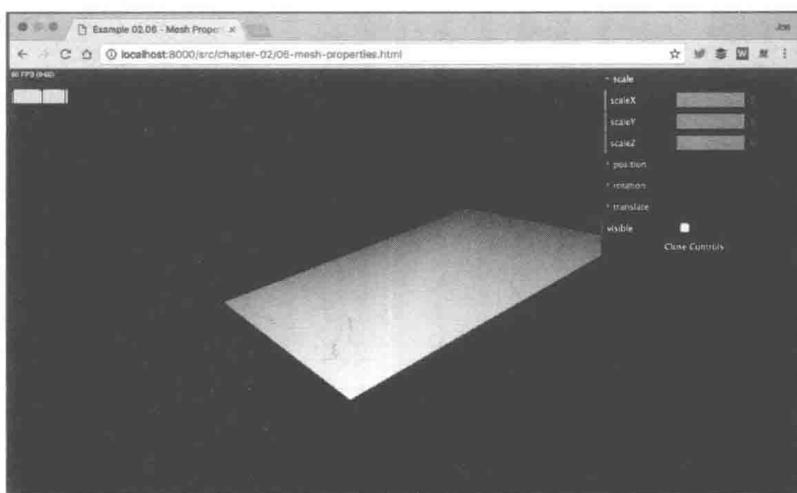


图 2.14

当你再次点击 `visible` 按钮时，立方体又再次出现了。在第 5 章和第 7 章中，我们还会进一步介绍网格和几何体，以及如何使用这些对象。

## 2.3 选择合适的摄像机

Three.js 库提供了两种不同的摄像机：正交投影摄像机和透视投影摄像机。在第 3 章中将会详细介绍如何使用这些摄像机，所以在本章中只会介绍一些比较基础的内容。值得注意的是，Three.js 还提供了一些非常特殊的摄像机用于支持 3D 眼镜和 VR 设备。由于它们与本章将要讲述的基础摄像机的工作方式类似，因此本书不会深入介绍这些特殊摄像机。

 如果你只需要简单 VR 摄像机（即标准的立体视觉效果），可以使用 `THREE.StereoCamera` 将左右眼画面并排渲染，或者也可以使用其他特殊摄像机渲染视差屏

障式的（例如3DS提供的设备）3D图像，或者是传统的红蓝重叠式的3D图像。此外，Three.js还实验性地支持WebVR这一被浏览器广泛支持的标准（更多信息请参考<https://webvr.info/developers/>）。为了启用这一支持，只需要设置renderer.vr.enabled = true;，后续工作Three.js会为你做好。在Three.js的官方网站可以找到有关这一属性值以及其他WebVR相关特性的示例：<https://threejs.org/examples/>。

下面将会使用几个示例来解释正交投影摄像机和透视投影摄像机的不同之处。

### 2.3.1 正交投影摄像机和透视投影摄像机

在本章的示例中有个名为07-both-cameras.html的例子。当你打开该示例时，会看到如图2.15所示的结果。

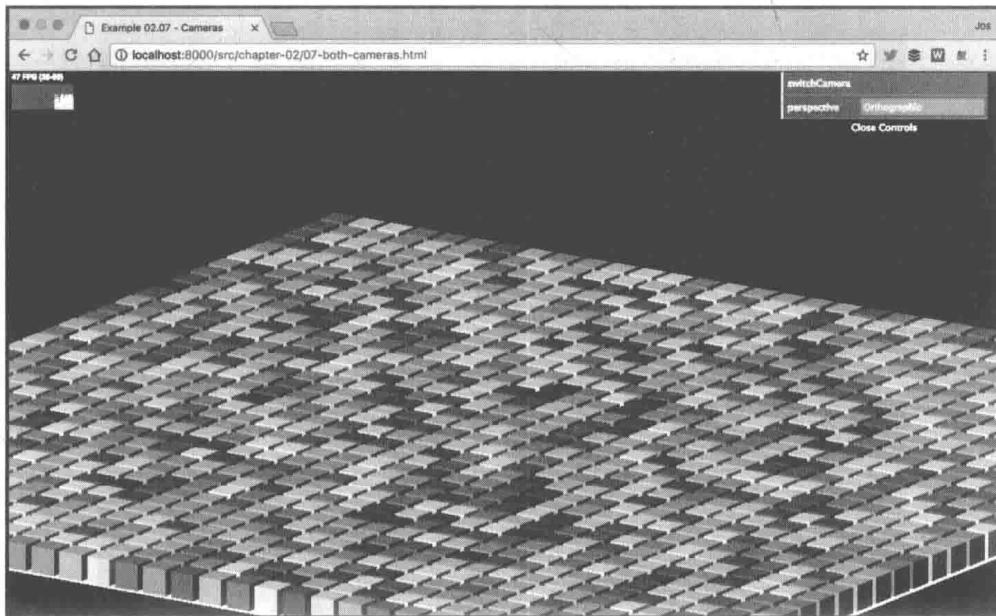


图 2.15

这就是透视视图，也是最自然的视图。正如你所看到的，这些立方体距离摄像机越远，它们就会被渲染得越小。如果我们使用另外一种摄像机——正交投影摄像机，对于同一个场景你将会看到如图2.16所示的效果。

使用正交投影摄像机的话，所有的立方体被渲染出来的尺寸都是一样的，因为对象相对于摄像机的距离对渲染的结果是没有影响的。这种摄像机通常被用于二维游戏中，比如《模拟城市4》和早期版本的《文明》。如图2.17所示。

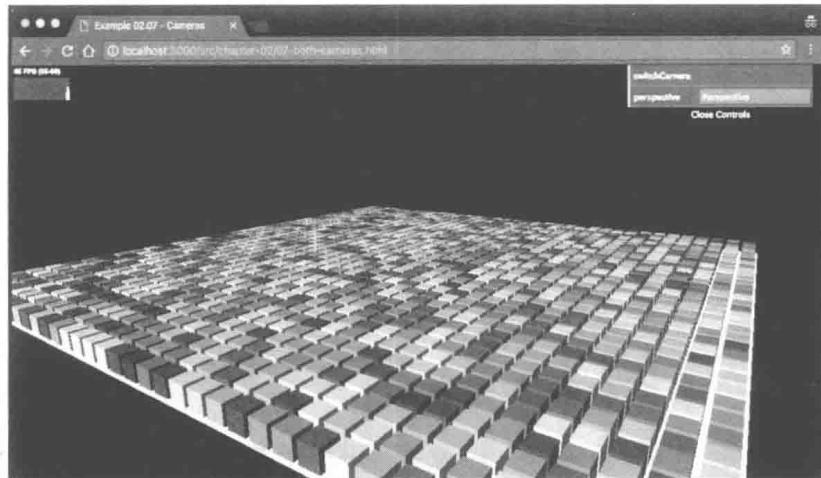


图 2.16



图 2.17

在我们的示例中，大部分使用的是透视投影摄像机，因为这种摄像机的效果更贴近真实世界。在 Three.js 中改变摄像机是很简单的，当你点击 07-both-cameras.html 示例中的 switchCamera 按钮时，下面的代码将会被执行：

```
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 = 120;
camera.position.y = 60;
camera.position.z = 180;
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";
}
};


```

THREE.PerspectiveCamera 和 THREE.OrthographicCamera 的创建方法是不一样的。首先我们先来看下 THREE.PerspectiveCamera，该方法接受的参数如表 2.4 所示。

表 2.4

参 数	描 述
fov	fov 表示视场。这是在摄像机中能够看到的那部分场景。比如，人类有接近 180 度的视场，而有些鸟类有接近 360 度的视场。但是由于计算机不能完全显示我们能够看到的景象，所以一般会选择一块较小的区域。对于游戏而言，视场大小通常为 60 度到 90 度 推荐默认值：50
aspect (长宽比)	这是渲染结果的横向尺寸和纵向尺寸的比值，在我们的示例中，由于使用窗口作为输出界面，所以使用的是窗口的长宽比。这个长宽比决定了横向视场和纵向视场的比例关系 推荐默认值：window.innerWidth/window.innerHeight
near (近面距离)	near 属性定义了从距离摄像机多近的距离开始渲染。通常情况下这个值会设置得尽量小，从而能够渲染从摄像机位置可以看到的所有物体 推荐默认值：0.1
far (远面距离)	far 属性定义了摄像机从它所处的位置能够看多远。如果这个值设置得较小，那么场景中有一部分不会被渲染；如果设置得较大，那么就会影响渲染的性能 推荐默认值：1000
zoom (变焦)	使用 zoom 属性你可以放大和缩小场景。如果这个值设置得小于 1，那么场景就会被缩小；如果这个值设置得大于 1，那么场景就会被放大。需要注意的是，如果设置的值为负数，那么场景就会上下颠倒 推荐默认值：1

这些属性结合到一起影响你所看到的景象，如图 2.18 所示。

摄像机的 `fov` 属性决定了横向视场。基于 `aspect` 属性，纵向视场也就相应地确定了。`near` 属性决定了近面距离，`far` 属性决定了远面距离。近面距离和远面距离之间的区域将会被渲染。

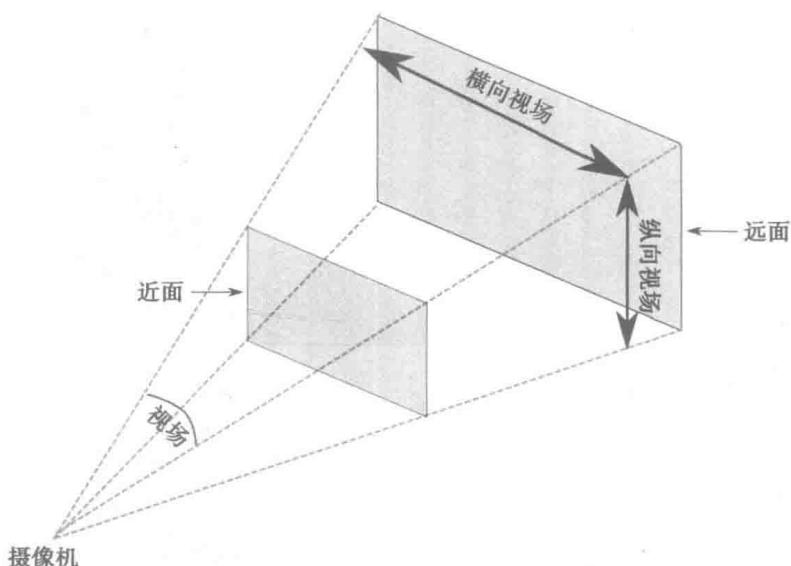


图 2.18

如果要配置正交投影摄像机，我们需要使用其他的一些属性。由于正交投影摄像机渲染出的物体大小都是一样的，所以它并不关心使用什么长宽比，或者以什么样的视角来观察场景。当使用正交投影摄像机时，你要定义的是一个需要被渲染的方块区域。表 2.5 列出了正交投影摄像机相应的属性。

表 2.5

参 数	描 述
<code>left</code> (左边界)	Three.js 文档里该属性被描述为可视范围的左平面。你可以将它看作是渲染部分的左边界。如果这个值设置为 -100，那么你将不会看到比这个左边界更远的物体
<code>right</code> (右边界)	和 <code>left</code> 属性一样，只不过它定义的是可被渲染区域的另一个侧面。任何比这个右边界远的物体都不会被渲染
<code>top</code> (上边界)	可被渲染区域的最上面
<code>bottom</code> (下边界)	可被渲染区域的最下面
<code>near</code> (近面距离)	基于摄像机所处的位置，从这一点开始渲染场景
<code>far</code> (远面距离)	基于摄像机所处的位置，渲染场景到这一点为止
<code>zoom</code> (变焦)	使用该属性可以放大和缩小场景。如果这个值设置得小于 1，那么场景就会被缩小；如果这个值设置得大于 1，那么场景就会被放大。需要注意的是，如果设置的值为负数，那么场景就会上下颠倒。默认值为 1

所有这些属性可以总结为图 2.19。

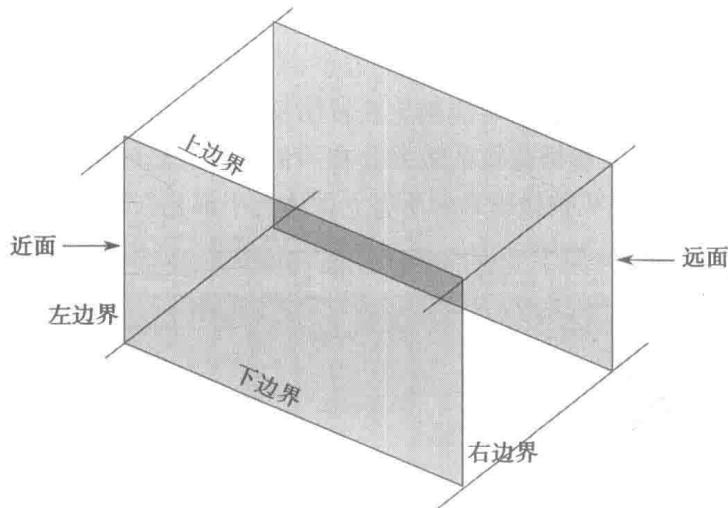


图 2.19

### 2.3.2 将摄像机聚焦在指定点上

到目前为止，我们已经介绍了如何创建摄像机，以及各个参数的含义。在前面一章中，我们也讲过摄像机需要放置在场景中的某个位置，以及摄像机能够看到的区域将会被渲染。通常来说，摄像机会指向场景的中心，用坐标来表示就是  $\text{position}(0,0,0)$ 。但是我们可以很容易地改变摄像机所指向的位置，代码如下所示：

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

在我们的示例中，摄像机是可以移动的，而且它所指向的位置标记一个红点，如图 2.20 所示。

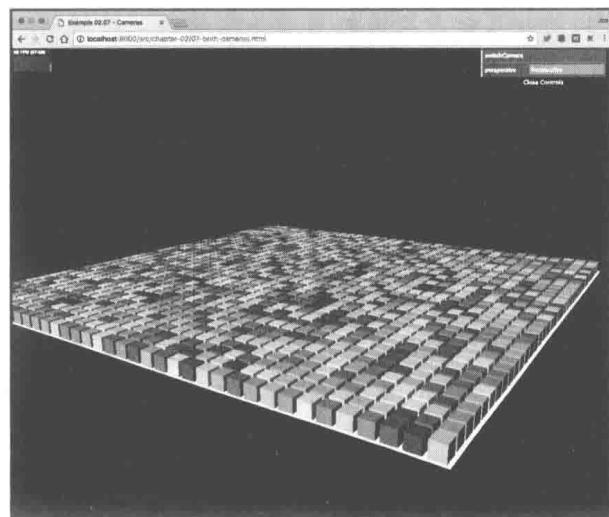


图 2.20

如果你打开示例 08-cameras-lookat.html，就会发现场景正从左向右移动。其实场景并没有移动，而是摄像机从不同点（屏幕中央的红点）拍摄场景，其带来的效果就是场景从左向右移动。在该示例中，你还可以切换到正交投影摄像机；你会看到改变摄像机拍摄位置所带来的效果和使用透视投影摄像机的效果是相同的。有意思的是，不管摄像机拍摄的位置如何改变，正交投影摄像机拍摄出来的所有立方体大小都是一样的。如图 2.21 所示。

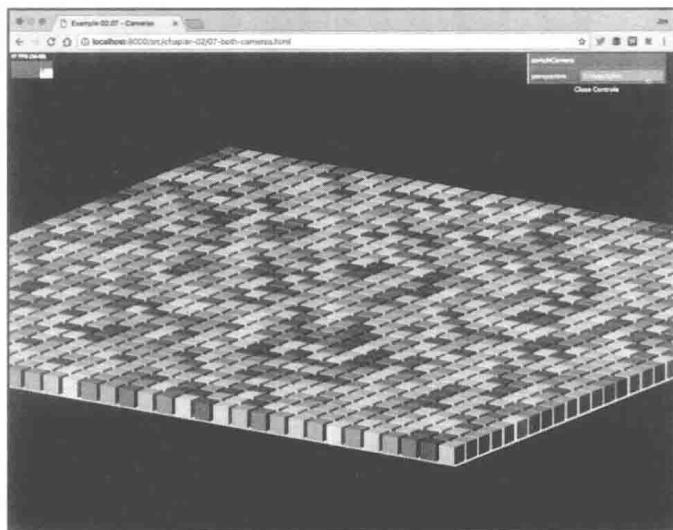


图 2.21

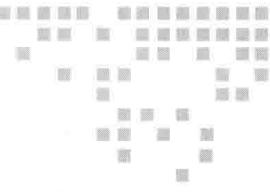


当使用 `lookAt()` 方法时，可以在某个特定的位置设置摄像机。使用该方法还可以让摄像机追随场景中的某个物体。由于 `THREE.Mesh` 对象的位置都是 `THREE.Vector3` 对象，所以可以使用 `lookAt()` 方法使摄像机指向场景中特定的某个网格。你所需要做的就是输入如下代码：`camera.lookAt(mesh.position)`。如果在 `render` 循环中执行该代码，你所看到的效果就是摄像机随着物体的移动而移动。

## 2.4 总结

在本章中我们介绍了 `THREE.Scene` 的所有属性和方法，并解释了如何使用这些属性来配置主场景。我们还展示了如何使用 `THREE.Geometry` 对象或者使用 Three.js 内置的几何体来创建几何体。最后我们介绍了如何配置 Three.js 提供的两种摄像机：透视投影摄像机使用接近真实世界的视角来渲染场景，而正交投影摄像机提供了一种在游戏中被广泛采用的伪三维效果。我们还介绍了 Three.js 中的几何体是如何工作的。现在你就可以很简单地创建你自己的几何体了。

在下一章中，我们将会介绍 Three.js 库提供的各种不同光源。你将会学到各种不同光源的行为，如何创建和配置这些光源，以及它们对特定材质的影响。



第3章

Chapter 3

## 学习使用 Three.js 中的光源

在第1章中，我们学习了Three.js的基础知识，而在上一章，我们对场景中最重要的部分进行了一些深入的了解，包括几何体、网格和摄像机。你可能已经注意到，尽管灯光也是场景中十分重要的一部分，但是在之前的章节中却略过了。没有光源，渲染的场景将不可见（除非你使用基础材质或线框材质）。Three.js中包含大量的光源，每一个光源都有特别的用法，所以我们会用一整章来阐述不同光源的详情，并为下一章材质的使用做准备。



WebGL本身并不支持光源。如果不使用Three.js，则需要自己写WebGL着色程序来模拟光源。查阅使用WebGL从头开始模拟光源的推荐资料请访问如下网址：  
[https://developer.mozilla.org/en-US/docs/Web/WebGL/Lighting\\_in\\_WebGL](https://developer.mozilla.org/en-US/docs/Web/WebGL/Lighting_in_WebGL)。

在本章中你将学到以下几个主题：

- Three.js中可用的光源。
- 特定光源使用的时机。
- 如何调整和配置所有光源的行为。
- 简单地介绍如何创建镜头光晕。

在本书所有的章节中，有大量的示例可以用来试验光源的行为。本章的示例代码可以在提供的源码的chapter-03文件夹中找到。

### 3.1 Three.js中不同种类的光源

Three.js中有许多不同种类的光源，每种光源都有特别的行为和用法。在本章中，将讨论表3.1中所列光源。

表 3.1

名 字	描 述
THREE.AmbientLight	这是一个基本光源，该光源的颜色将会叠加到场景现有物体的颜色上
THREE.PointLight	这是一个点光源，从空间的一点向所有方向发射光线。点光源不能用来创建阴影
THREE.SpotLight	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。这种光源可以投射阴影
THREE.DirectionalLight	这种光源也称作无限光。从这种光源发出的光线可以看作是平行的，打个比方，就像太阳光。这种光源也可以用来创建阴影
THREE.HemisphereLight	这是一种特殊的光源，可以通过模拟反光面和光线微弱的天空来创建更加自然的室外光线。这个光源也不提供任何与阴影相关的功能
THREE.AreaLight	使用这种光源可以指定散发光线的平面，而不是一个点。THREE.AreaLight 不投射任何阴影
THREE.LensFlare	这不是一种光源，但是通过使用 THREE.LensFlare，可以为场景中的光源添加镜头光晕效果

本章主要分为两部分内容。首先，我们先看一下基础光源：THREE.AmbientLight、THREE.PointLight、THREE.SpotLight 和 THREE.DirectionalLight。所有这些光源都是基于 THREE.Light 对象扩展的，这个对象提供公用的功能。以上提到的光源都是简单光源，只需一些简单的配置即可，而且可用于创建大多数场景的光源。在第二部分，我们将会看到一些特殊用途的光源和效果：THREE.HemisphereLight、THREE.AreaLight 和 THREE.LensFlare。只有在十分特殊的情况下才会用到这些光源。

## 3.2 基础光源

我们将从最基本的 THREE.AmbientLight 光源开始。

### 3.2.1 THREE.AmbientLight

在创建 THREE.AmbientLight 时，颜色将会应用到全局。该光源并没有特别的来源方向，并且 THREE.AmbientLight 不会生成阴影。通常，不能将 THREE.AmbientLight 作为场景中唯一的光源，因为它会将场景中的所有物体渲染为相同的颜色，而不管是什么形状。在使用其他光源（如 THREE.SpotLight 或 THREE.DirectionalLight）的同时使用它，目的是弱化阴影或给场景添加一些额外的颜色。理解这点的最简单方式就是查看 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 THREE.AmbientLight 光源。请注意，在这个场景中，也使用了 THREE.SpotLight 光源来照亮物体并生成阴影。

从如图 3.1 所示的截图里，你可以看到我们使用了第 1 章中的场景，并且 THREE.

AmbientLight 的颜色和强度是可调的。在这个例子中，可以关闭聚光灯来查看只有 THREE.AmbientLight 光源的效果。

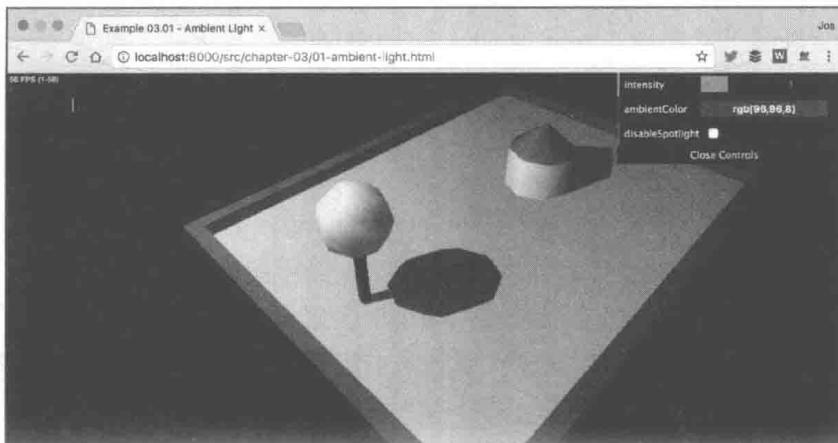


图 3.1

我们在这个场景中使用的标准颜色是 #606008。这是该颜色的十六进制表示形式。前两个值表示颜色的红色部分，紧接着的两个值表示绿色部分，而最后两个值表示蓝色部分。在该例的用户界面中显示的是颜色的十进制值。

在这个例子里，我们使用一个非常暗淡的灰色，用来弱化网格对象在地面上生硬的投影。通过右上角的菜单，你可以将这个颜色改成比较明显的黄橙色：rgb(190,190,41)，这样所有对象就会笼罩在类似阳光的光辉下，如图 3.2 所示。

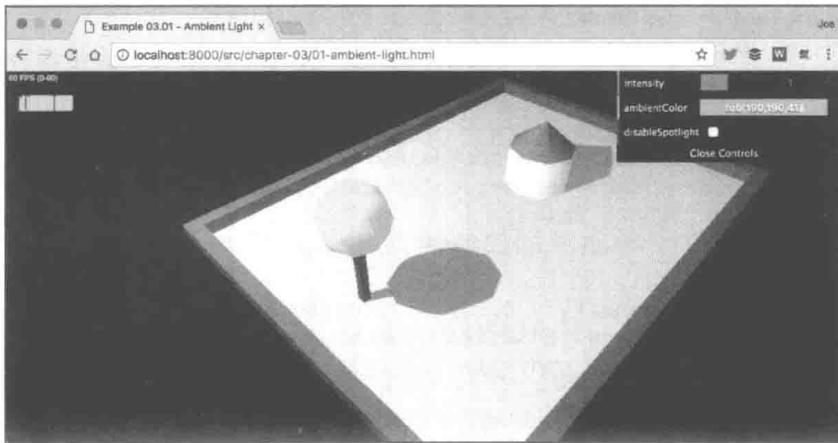


图 3.2

正如图 3.2 所示，这个黄橙色应用到了所有的对象，并在整个场景中投下了一片橙色的光辉。使用这种光源时要记住：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的颜色过于饱和了。除了颜色之外，还可以为环境光设置强度值。这

一个参数决定了光源 THREE.AmbientLight 对场景中物体颜色的影响程度。如果将该参数调小，则光源对颜色的影响会很微弱。如果将该值调大，则整个场景会变得过于明亮。实际效果如图 3.3 所示。

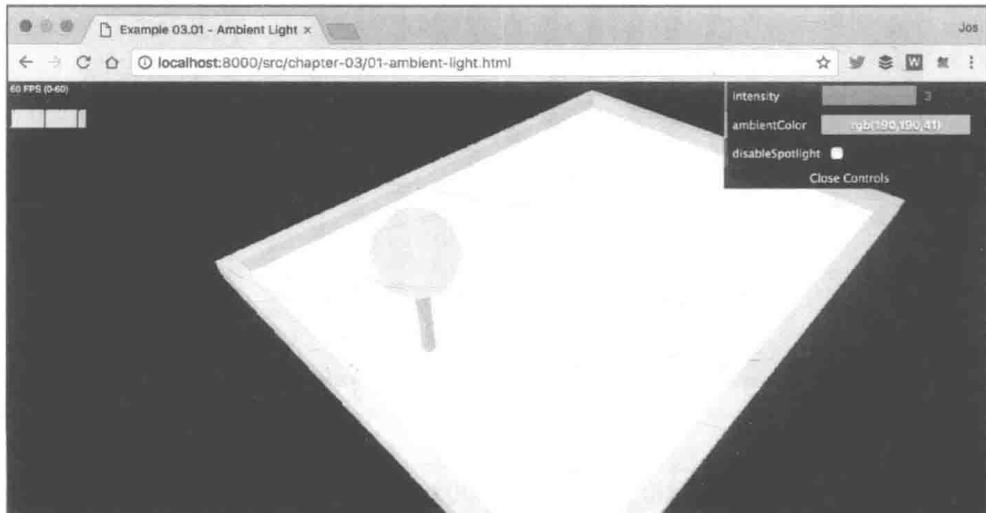


图 3.3

既然我们已经知道了 THREE.AmbientLight 光源能做什么，接下来学习如何创建和使用 THREE.AmbientLight 光源。下面几行代码展示了如何创建 THREE.AmbientLight 光源，以及如何将该光源与我们前几章已见过的 GUI 控制菜单关联起来。

```
var ambientLight = new THREE.AmbientLight("#606008");
scene.add(ambientLight);
...

var controls = new function () {
    this.intensity = ambientLight.intensity;
    this.ambientColor = ambientLight.color.getStyle();
    this.disableSpotlight = false;
};

var gui = new dat.GUI();
gui.add(controls, 'intensity', 0, 3, 0.1).onChange(function (e) {
    ambientLight.color = new THREE.Color(controls.ambientColor);
    ambientLight.intensity = controls.intensity;
});
gui.addColor(controls, 'ambientColor').onChange(function (e) {
    ambientLight.color = new THREE.Color(controls.ambientColor);
    ambientLight.intensity = controls.intensity;
});
gui.add(controls, 'disableSpotlight').onChange(function (e) {
    spotLight.visible = !e;
});
```

创建 THREE.AmbientLight 光源非常简单。由于 THREE.AmbientLight 光源不需要指

定位置并且会应用到全局，所以只需使用 new THREE.AmbientLight("#606008") 来指定颜色（十六进制），并用 scene.add(ambientLight) 将此光源应用到整个场景。环境光类的构造函数还有一个可选参数 intensity，用于指定光的强度。上面的代码并没有指定强度值，则该值使用默认值 1。在这个例子里，将 THREE.AmbientLight 光源的颜色和强度绑定到控制菜单。为此，可以使用与前面章节相同的配置方法。唯一需要改变的是调用 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){...})。通过这个函数，我们告诉 dat.GUI 控制菜单在每次颜色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 THREE.AmbientLight 光源的颜色设置为一个新值。最后我们通过类似方法确保控制菜单中 intensity 选项的任何修改都会作用于场景中的环境光对象。

### 使用 THREE.Color 对象

在讲述下一个光源之前，我们先简单介绍一下 THREE.Color 对象。在 Three.js 中需要（例如为材质、灯光等）指定颜色时，可以使用 THREE.Color 对象，也可以像我们在设置环境光时所做的那样，以一个字符串指定颜色。此时 Three.js 将基于该字符串自动创建一个 THREE.Color 对象。实际上 Three.js 在构造 THREE.Color 对象时非常灵活，它可以基于下面所列的任何一种方式来完成：

- 无参数构造：new THREE.Color() 这种构造形式会创建一个代表白颜色的对象。
- 十六进制数值：new THREE.Color(0xababab) 这种构造形式会将十六进制值转换为颜色分量值并基于此构造颜色对象。这是最佳的颜色对象构造形式。
- 十六进制字符串：new THREE.Color("#ababab")，此时 Three.js 会将字符串当作 CSS 颜色字符串去解释，并构造颜色对象。
- RGB 字符串：顾名思义，这种构造形式需要为每个 RGB 分量指定亮度值，其具体形式可以是 new THREE.Color("rgb(255, 0, 0)") 或者 new THREE.Color("rgb(100%, 0%, 0%)")。
- 颜色名称：可以使用 Three.js 能够识别的颜色名称字符串，例如 new THREE.Color("skyblue")。
- HSL 字符串：如果相比 RGB，你更熟悉 HSL 色域，也可以使用 HSL 值来构造颜色对象，例如 new THREE.Color("hsl(0%, 100%, 50%)")。
- 分离的 RGB 值：最后也可以直接使用 RGB 颜色分量来构造颜色对象。这三个值的范围都是 0 到 1。形式例如 new THREE.Color(1, 0, 0)。

如果需要修改一个现有颜色对象的颜色，可以用新颜色值构造一个临时颜色对象，并将其复制给现有对象。另一种修改颜色的方式是使用 THREE.Color 类携带的方法来读取和修改其内部颜色值，如表 3.2 所列。

表 3.2

函数名	描述
set(value)	将当前颜色设置为指定的十六进制值。这个值可以是字符串、数值或是已有的 THREE.Color 实例
setHex(value)	将当前颜色设置为指定的十六进制值
setRGB(r,g,b)	根据提供的 RGB 值设置颜色。参数范围从 0 到 1
setHSL(h,s,l)	根据提供的 HSL 值设置颜色。参数范围从 0 到 1。可以查看 <a href="http://en.wikibooks.org/wiki/Color_Models:_RGB,_HSV,_HSL">http://en.wikibooks.org/wiki/Color_Models:_RGB,_HSV,_HSL</a> 了解 HSL 如何用于设置颜色
setStyle(style)	根据 CSS 设置颜色的方式来设置颜色。例如：可以使用 "rgb(255,0,0)"、"#ff0000"、"#f00" 或 "red"
copy(color)	从提供的颜色对象复制颜色值到当前对象
copyGammaToLinear(color)	用 THREE.Color 提供的实例设置对象的颜色。颜色是由伽马色彩空间转换到线性色彩空间得来的。伽马色彩空间也使用 RGB 颜色，但是会使用指数系数而不是线性系数
copyLinearToGamma(color)	用 THREE.Color 提供的实例设置对象的颜色。颜色是由线性色彩空间转换到伽马色彩空间得来的
convertGammaToLinear()	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：435241
getHexString()	以十六进制字符串形式从颜色对象中获取颜色值："0c0c0c"
getStyle()	以 CSS 值的形式从颜色对象中获取颜色值："rgb(112,0,0)"
getHSL(optionalTarget)	以 HSL 值的形式从颜色对象中获取颜色值。如果提供了 optionalTarget 对象，Three.js 将把 h、s 和 l 属性设置到该对象
offsetHSL(h,s,l)	将提供的 h、s 和 l 值添加到当前颜色的 h、s 和 l 值上
add(color)	将 r、g 和 b 值添加到当前颜色
addColors(color1,color2)	将 color1 和 color2 相加，再将得到的值设置到当前颜色
addScalar(s)	在当前颜色的 RGB 分量上添加值。谨记内部值范围从 0 到 1
multiply(color)	将当前颜色的 RGB 值与 THREE.color 对象的 RGB 值相乘
multiplyScalar(s)	将当前颜色的 RGB 值与提供的 RGB 值相乘。谨记内部值范围从 0 到 1
lerp(color,alpha)	找出介于对象的颜色和提供的颜色之间的颜色，alpha 属性定义了当前颜色与提供的颜色的差距
equals(color)	如果 THREE.Color 对象实例提供的颜色的 RGB 值与当前颜色相等，则返回 true
fromArray(array)	与 setRGB 方法具有相同的功能，只是 RGB 值可以通过数字数组的方式作为参数传入
toArray	返回三个元素的数组：[r, g, b]
clone()	复制当前颜色

从这张表中可以看出，要改变当前颜色有很多方法。其中很多是Three.js库内部使用的，但同样也提供了一些方法轻松修改光源和材质的颜色。

在进入对THREE.PointLight、THREE.SpotLight和THREE.DirectionalLight的讨论之前，我们先看一下它们之间最主要的区别，那就是它们发射光线的方式。图3.4展示了这三种光源如何发射光线。

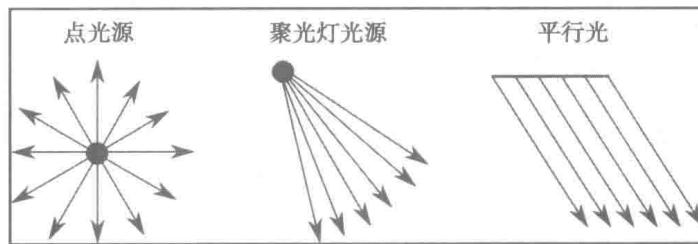


图 3.4

从这个图中可以看到如下信息：

- THREE.PointLight 从特定的一点向所有方向发射光线。
- THREE.SpotLight 从特定的一点以锥形发射光线。
- THREE.DirectionalLight 不是从单个点发射光线，而是从二维平面发射光线，光线彼此平行。

在接下来的几节里，我们将更详细地介绍这些光源。

### 3.2.2 THREE.SpotLight

THREE.SpotLight（聚光灯光源）是最常使用的光源之一（特别是如果你想要使用阴影的话）。THREE.SpotLight是一种具有锥形效果的光源。你可以把它与手电筒或灯塔产生的光进行对比。该光源产生的光具有方向和角度。图3.5展示了聚光灯光源的效果（02-spot-light.html）。

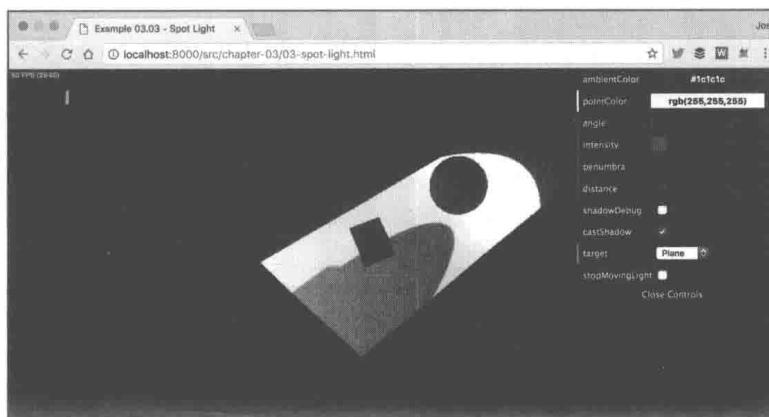


图 3.5

表 3.3 列出了适用于 THREE.SpotLight 的所有属性。我们会先研究那些与光照直接相关的属性，之后再看一看与渲染阴影有关的属性。

表 3.3

属性	描述
angle (角度)	光源发射出的光束的宽度。单位是弧度，默认值为 <code>Math.PI/3</code>
castShadow (投影)	如果设置为 <code>true</code> ，这个光源就会生成阴影
color (颜色)	光源颜色
decay (衰减)	光源强度随着离开光源的距离而衰减的速度。该值为 2 时更接近现实世界中的效果，默认值为 1。只有当 <code>WebGLRenderer</code> 的属性 <code>physicallyCorrectLights</code> （物理正确光源）被设置为启用时， <code>decay</code> 属性才有效
distance (距离)	光源照射的距离。默认值为 0，这意味着光线强度不会随着距离增加而减弱
intensity (强度)	光源照射的强度。默认值为 1
penumbra( 半影区)	该属性设置聚光灯的锥形照明区域在其区域边缘附近的平滑衰减速度。取值范围在 0 到 1 之间，默认值为 0
position (位置)	光源在场景中的位置
power (功率)	当物理正确模式启用时（即 <code>WebGLRenderer</code> 的属性 <code>physicallyCorrectLights</code> 被设置为启用），该属性指定光源的功率，以流明为单位，默认值为 <code>4*Math.PI</code>
target (目标)	使用 <code>THREE.SpotLight</code> 光源时，它的指向很重要。使用 <code>target</code> 属性，你可以将 <code>THREE.SpotLight</code> 光源指向场景中的特定对象或位置。注意，此属性需要一个 <code>THREE.Object3D</code> 对象（如 <code>THREE.Mesh</code> ）。这与我们上一章在 <code>lookAt</code> 方法中使用 <code>THREE.Vectors3</code> 对象时看到的摄像机不同
visible (是否可见)	如果该属性设置为 <code>true</code> （默认值），该光源就会打开；如果设置为 <code>false</code> ，光源就会关闭

当 `THREE.SpotLight` 的 `shadow` 属性为 `enable` 时，可以通过表 3.4 中的属性来调节阴影特性。

表 3.4

属性	描述
shadow.bias (阴影偏移)	用来偏置阴影的位置。当你使用非常薄的对象时，可以使用它来解决一些奇怪的效果（可访问网址 <a href="http://www.3dbuzz.com/training/view/unity-fundamentals/lights/8-shadows-bias">http://www.3dbuzz.com/training/view/unity-fundamentals/lights/8-shadows-bias</a> 来查看例子）。如果你看到奇怪的阴影效果，将该属性设置为很小的值（例如 0.01）通常可以解决问题。此属性的默认值为 0
shadow.camera.far (投影远点)	到距离光源的哪一个位置可以生成阴影。默认值为 5000。你可以设置所有提供给 <code>THREE.PerspectiveCamera</code> 的其他属性
shadow.camera.fov (投影视场)	用于生成阴影的视场大小（参见 2.3 节）。默认值为 50
shadow.camera.near (投影近点)	从距离光源的哪一个位置开始生成阴影。默认值为 50

(续)

属性	描述
shadow.mapSize.width 和 shadow.mapSize.height (阴影映射宽度和阴影映射高度)	决定了有多少像素用来生成阴影。当阴影具有锯齿状边缘或看起来不光滑时，可以增加这个值。在场景渲染之后无法更改。两者的默认值均为 512
shadow.radius (半径)	当该值大于 1 时，阴影的边缘将有平滑效果。该属性在 THREE.WebGLRenderer 的 shadowMap.type 属性为 THREE.BasicShadowMap 时无效

创建聚光灯光源非常简单。只要指定颜色、设置想要的属性并将其添加到场景中即可，如下代码所示：

```
var spotLight = new THREE.SpotLight("#ffffff");
spotLight.position.set(-40, 60, -10);
spotLight.castShadow = true;
spotLight.shadow.camera.near = 1;
spotLight.shadow.camera.far = 100;
spotLight.target = plane;
spotLight.distance = 0;
spotLight.angle = 0.4;
spotLight.shadow.camera.fov = 120;
```

在上面代码中，我们创建了 THREE.SpotLight 对象实例，并且将 castShadow 属性设置为 true，因为我们想要阴影。此外，由于需要让这个光源照向指定的方向，因此我们通过设置 target 属性来实现。在本例中，我们将其指向名为 plane 的对象。当运行示例（02-spot-light.html）时，你将看到如图 3.6 所示的场景。

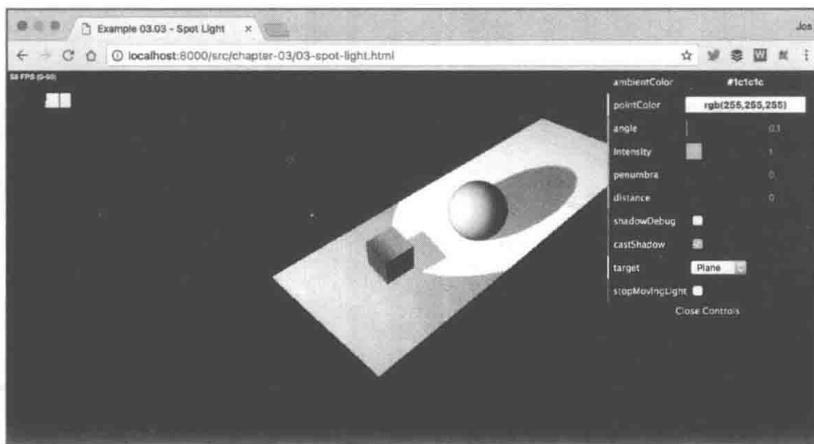


图 3.6

在这个例子里，可以设置一些 THREE.SpotLight 对象独有的属性。其中之一就是 target 属性。如果我们对蓝色球体（sphere 对象）设置此属性，那么这个光源会一直瞄准球体的中心，即使它在绕场景移动。我们创建这个光源时，瞄准的是地面（plane）对象，而在我们的例子中，也可以将光源瞄准另外两个物体。但是如果你不想把光源瞄准一个特定的对象，

而是空间中的任意一点呢？可以通过创建一个 THREE.Object3D() 对象来实现，如下代码所示：

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

然后，设置 THREE.SpotLight 对象的 target 属性：

```
spotlight.target = target
```

在表 3.3 中列出了几个适用于 THREE.SpotLight 的属性，这些属性可以控制光线如何从 THREE.SpotLight 对象发出。distance 属性和 angle 属性定义了光锥的形状。angle 属性定义了光锥的角度，而 distance 属性则可以用来设置光锥的长度。图 3.7 解释了这两个值如何一起定义了从 THREE.SpotLight 对象发出光线的区域。

通常情况是不需要设置这些值的，因为它们的默认值比较合适，但是也可以使用这些属性，例如创建一个光柱很窄或光强递减很快的 THREE.SpotLight 对象。此外，还有最后一个可以更改 THREE.SpotLight 光源渲染方式的属性——penumbra 属性。通过这个属性，可以设置光强从光锥中心向锥形边缘递减的速度。在图 3.8 中，可以看到 penumbra 属性的运行结果——束非常明亮的光（intensity 值很高），离中心越远光强衰减得越快（penumbra 值很高）。

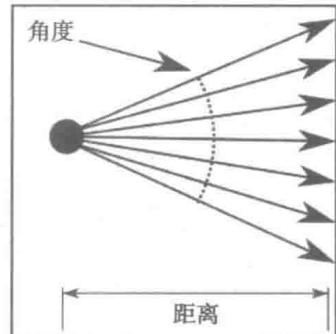


图 3.7

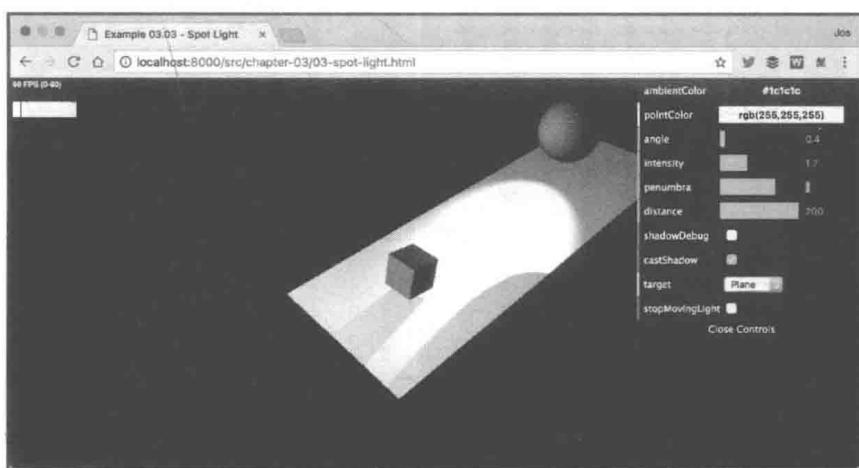


图 3.8

在开始学习下一个光源之前，我们先简要介绍一下 THREE.SpotLight 光源提供的几个与阴影相关的属性。我们已经学过，将 THREE.SpotLight 对象的 castShadow 属性设置为 true 可以生成阴影。（当然，在场景中渲染 THREE.Mesh 对象时，要确保为要投

射阴影的对象设置 `castShadow` 属性，为要显示阴影的对象设置 `receiveShadow` 属性。) Three.js 库也允许对阴影渲染的方式进行微调。这些已经在表 3.3 中进行了介绍。通过 `shadow.camera.near`、`shadow.camera.far` 和 `shadow.camera.fov`，可以控制光线如何投射阴影和在哪里投射阴影。其工作原理与我们前面章节中讲的透视摄像机的工作原理是一致的。想看看这些是如何起作用的，最简单的方法添加 `THREE.CameraHelper`。在示例程序中可以通过勾选菜单上的 `shadowDebug`（阴影调试）复选框来设置。这样可以把用来决定阴影的光照区域显示出来，如图 3.9 所示。

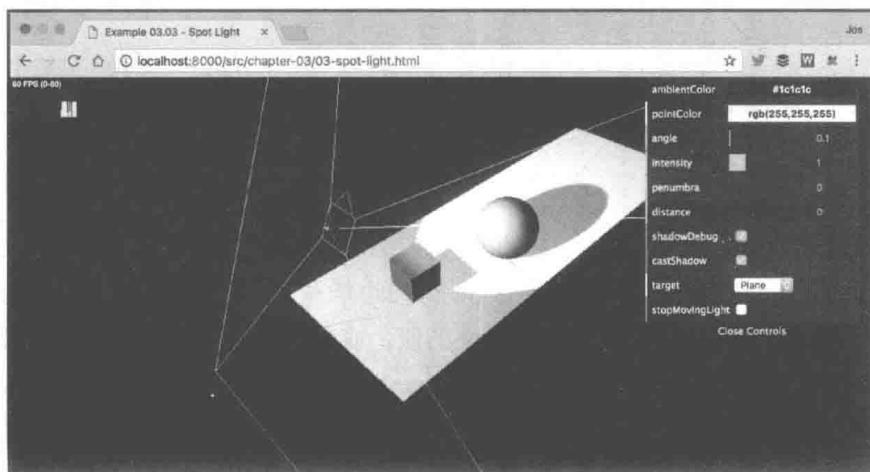


图 3.9

当实际阴影效果与所希望的不一致时，使用 `THREE.CameraHelper` 可以非常方便地帮助我们发现问题所在。下面代码展示了如何使用 `THREE.CameraHelper`。

```
var debugCamera = new THREE.CameraHelper(spotLight.shadow.camera);
scene.add(debugCamera)
```

与调试阴影类似，如果你需要调试聚光灯光源本身存在的问题，可以使用 Three.js 提供的 `THREE.SpotLightHelper`，并通过下面代码所示的方法使用该类。

```
var helper = new THREE.SpotLightHelper(spotLight);
scene.add(helper);

function render() {
  ...
  helper.update();
  ...
}
```

在 `THREE.SpotLightHelper` 的帮助下，我们可以直观地看到聚光灯的形状和朝向，如图 3.10 所示。

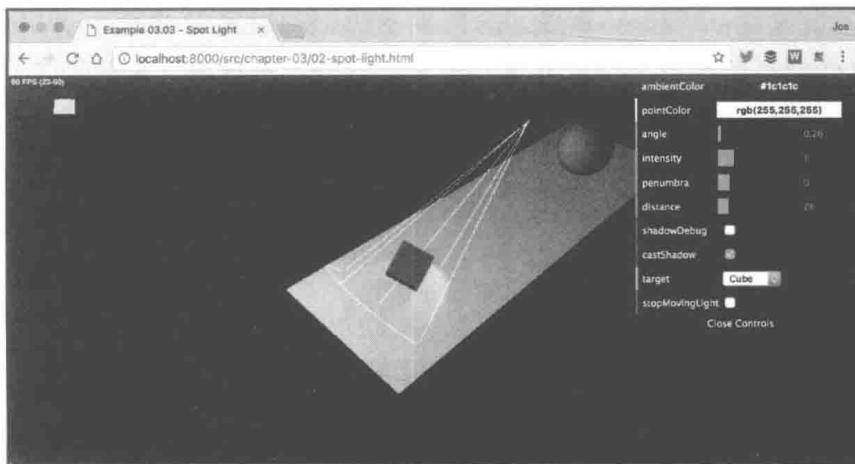


图 3.10

下面针对使用阴影的过程中可能遇到的问题给出几个提示：

- 如果阴影看上去有点粗糙（如阴影形状的边缘呈块状），可以增加 shadow.mapSize.width 和 shadow.mapSize.height 属性的值，或者保证用于计算阴影的区域紧密包围在对象周围。可以通过 shadow.camera.near、shadow.camera.far 和 shadow.camera.fov 属性来配置这个区域。
- 记住，不仅要告诉光源生成阴影，而且还必须通过配置每个几何体的 castShadow 和 receiveShadow 属性来告诉几何体对象是否接收或投射阴影。
- 如果你在场景中使用薄对象，在渲染阴影时，可能会出现奇怪的渲染失真现象。通常可以使用 shadow.bias 属性轻微偏移阴影来修复这些问题。
- 如果想要阴影更柔和，可以在 THREE.WebGLRenderer 对象上设置不同的 shadowMapType 属性值。默认情况下，此属性的值为 THREE.PCFShadowMap；如果将此属性设置为 PCFSoftShadowMap，则会得到更柔和的阴影。

### 3.2.3 THREE.PointLight

Three.js 库中的 THREE.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的例子。与所有光源一样，我们有一个专门的例子，你可以通过这个例子来试验 THREE.PointLight。打开 chapter-03 文件夹下的 03-point-light.html，你会看到一个点光源绕场景移动的例子。如图 3.11 所示。

本例中的场景还是第 1 章的那个场景，只是这次有一个点光源绕场景移动。为了更清楚地看到这个点光源在哪里，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这个光源从不同的侧面照亮。

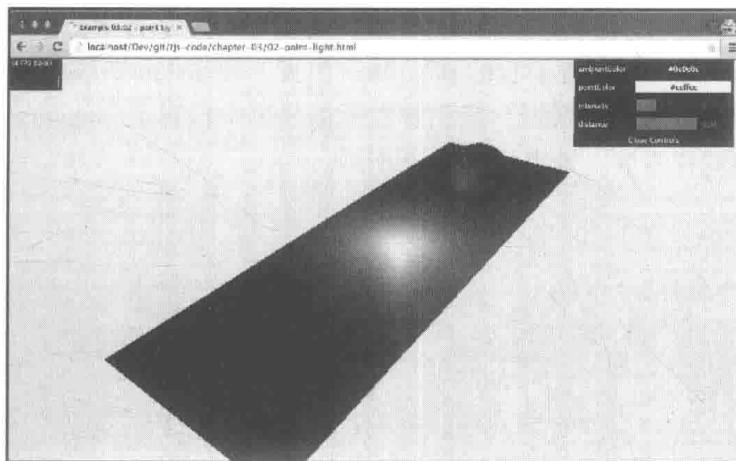


图 3.11



如果你使用过旧版 Three.js 便会知道点光源 (THREE.PointLight) 不会产生阴影，然后在新版 Three.js 中，点光源也可以像聚光灯 (THREE.SpotLight) 和平行光 (THREE.DirectionalLight) 一样产生阴影了。

用 THREE.PointLight 可以对光源设置很多额外的属性，具体见表 3.5。

表 3.5

属性	描述
color (颜色)	光源颜色
distance (距离)	光源照射的距离。默认值为 0，这意味着光的强度不会随着距离增加而减少
intensity (强度)	光源照射的强度。默认值为 1
position (位置)	光源在场景中的位置
visible (是否可见)	如果该属性设置为 true (默认值)，该光源就会打开；如果设置为 false，光源就会关闭
decay (衰减)	光源强度随着离开光源的距离而衰减的速度。该值为 2 时更接近现实世界中的效果，默认值为 1。只有当 WebGLRenderer 的属性 physicallyCorrectLights (物理正确光源) 被设置为启用时，decay 属性才有效
power (功率)	当物理正确模式启用时 (即 WebGLRenderer 的属性 physicallyCorrectLights 属性被设置为启用)，该属性指定光源的功率，以流明为单位，默认值为 $4 * \text{Math.PI}$ 。该属性与 intensity 属性为简单的线性关系 ( $\text{power} = \text{intensity} * 4\pi$ )

点光源可以像聚光灯光源一样启用阴影并设置其属性。在接下来的几个例子和截图中，我们将解释这些属性。首先，我们先看看如何创建 THREE.PointLight：

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

我们使用指定的颜色创建了一个光源（这里使用了一个字符串值，也可以使用一个数字或 THREE.Color 对象），设置了它的 position（位置）和 distance（距离）属性，并将它添加到场景中。在介绍聚光灯光源时，我们曾在示例代码中设置了 intensity 和 distance 属性，这些属性对点光源同样有效。效果如图 3.12 所示。

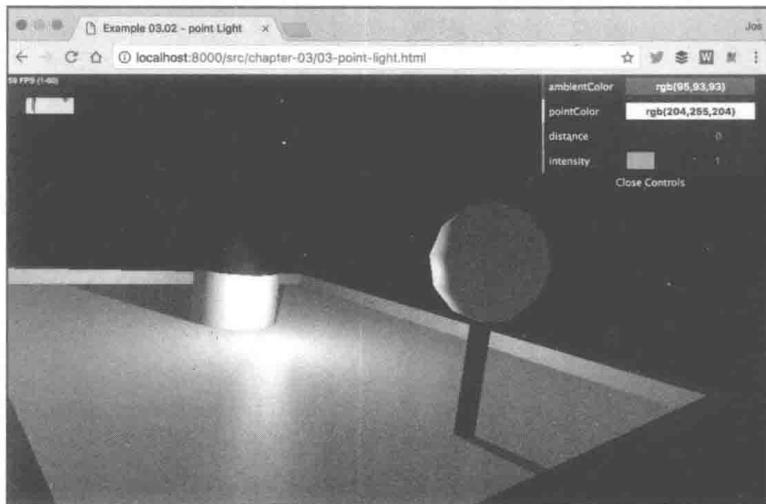


图 3.12

示例代码中没有设置 power（功率）和 decay（衰减）这两个属性，但它们对于模拟真实世界很有意义。下面网站为这两个属性的使用提供了很好的示例程序：[https://threejs.org/examples/#webgl\\_lights\\_physical](https://threejs.org/examples/#webgl_lights_physical)。在前面介绍聚光灯光源时，已经展示了使用 intensity 属性的效果，该属性对点光源同样适用，并能产生类似效果，如图 3.13 所示。

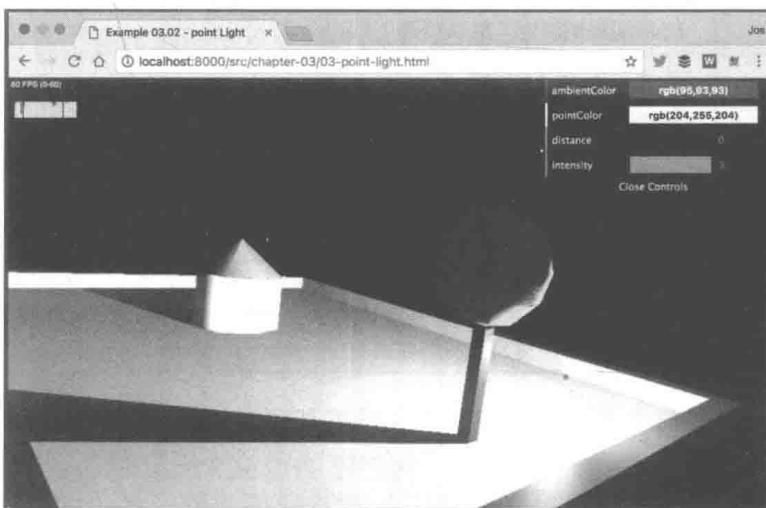


图 3.13

此外，聚光灯光源的 `distance` 属性在点光源上也能产生相似效果。下面的图 3.14 演示了当光的 `intensity` 值很高而 `distance` 值却很小时所产生的光源效果。

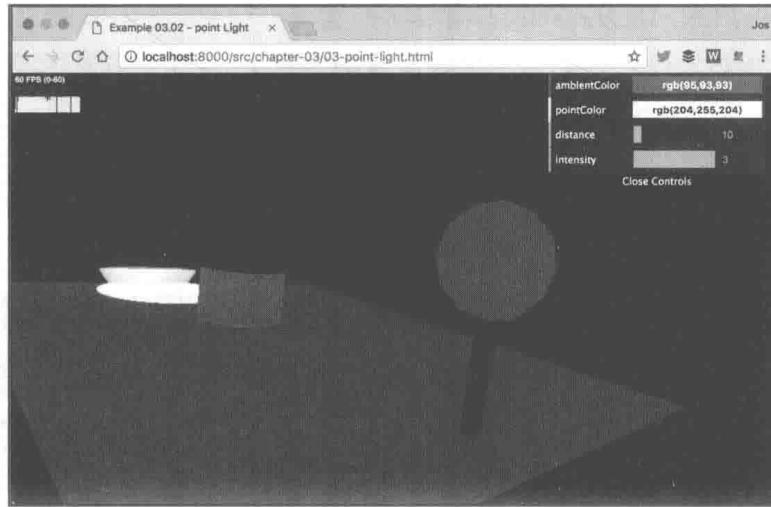


图 3.14

聚光灯光源中的 `distance` 属性决定了在光线强度变为 0 之前光线的传播距离。在图 3.14 中，光线强度在距离为 14 的地方慢慢地减少为 0。这就是为什么在这个例子中你仍然可以看到一个被照亮的明亮区域，但光却不会把更远的地方照亮。`distance` 属性的默认值为 0，这意味着光线强度不会随着距离的增加而减弱。

THREE.PointLight 同样使用摄像机来决定如何绘制阴影，所以也可以使用辅助类 THREE.CameraHelper 来展示场景中哪些部分被光源的摄像机所覆盖，以及使用辅助类 THREE.PointLightHelper 来展示点光源的光线所照射的位置。两个辅助类一起使用时，我们便可以获得非常直观的调试信息，如图 3.15 所示。

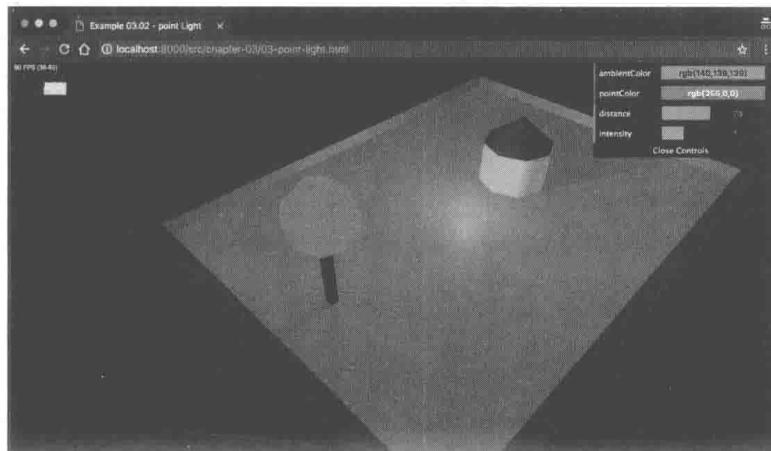


图 3.15

### 3.2.4 THREE.DirectionalLight

我们要看的最后一个基本光源是 THREE.DirectionalLight（平行光）。这种类型的光可以看作是距离很远的光。它发出的所有光线都是相互平行的。平行光的一个范例就是太阳光。太阳是如此遥远，以至于到达地球时所有的光线（几乎）都是相互平行的。THREE.DirectionalLight 和我们之前看过的 THREE.SpotLight 之间的主要区别是：平行光不像聚光灯（可以通过 distance 和 exponent 属性来微调）那样离目标越远越暗淡。被平行光照亮的整个区域接收到的光强是一样的。

可以在示例 04-directional-light.html 中看到实际效果，如图 3.16 所示。

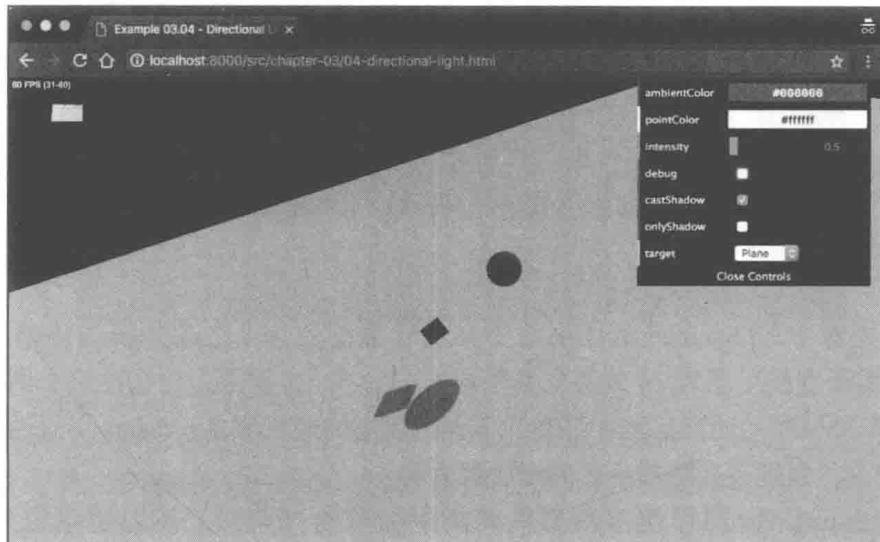


图 3.16

从上图可以看到，场景里没有那种锥形效果的光线。所有对象接收的都是相同光强的光。只有光源的方向（direction）、颜色（color）和强度（intensity）属性用来计算颜色和阴影。

与 THREE.SpotLight 一样，可以设置一些属性来控制光照的强度和投射阴影的方式。THREE.DirectionalLight 对象和 THREE.SpotLight 对象有许多属性相同，例如 position、target、intensity、castShadow、shadow.camera.near、shadow.camera.far、shadow.mapSize、width、shadow.mapSize.height 和 shadow.bias。关于这些属性更多的信息，可以查看 3.2.3 节。下面只讨论平行光光源特有的几个属性。

如果你研究一下 THREE.SpotLight 的例子，会发现我们必须定义生成阴影的光锥。然而，对于 THREE.DirectionalLight，由于所有的光线都是平行的，所以不会有光锥，而是一个立方体区域，如图 3.17 所示（如果想看到它，可以移动摄像机远离场景并勾选 debug 复选框）。

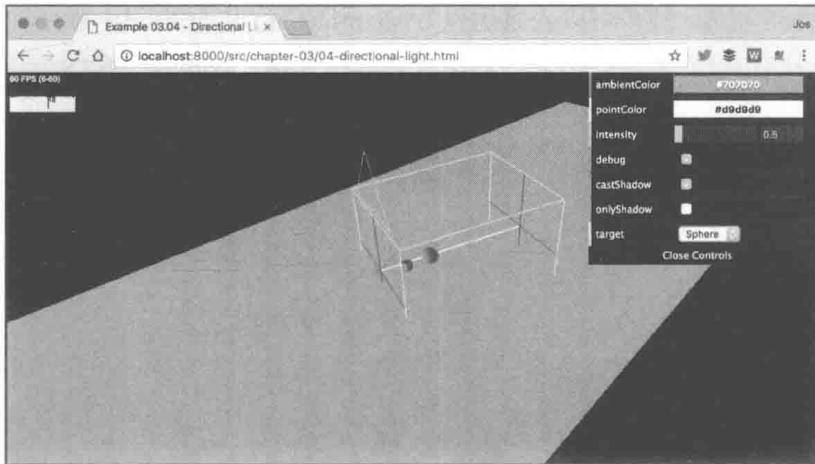


图 3.17

在这个立方体范围内的所有对象都可以投影和接收阴影。与 THREE.SpotLight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来定义这个立方体范围：

```
directionalLight.castShadow = true;
directionalLight.shadow.camera.near = 2;
directionalLight.shadow.camera.far = 80;
directionalLight.shadow.camera.left = -30;
directionalLight.shadow.camera.right = 30;
directionalLight.shadow.camera.top = 30;
directionalLight.shadow.camera.bottom = -30;
```

可以把这个与第2章中配置正交投影摄像机的方法比较一下。

### 3.3 特殊光源

这一节讲述的是特殊光源，我们将讨论Three.js提供的两个特殊光源。首先要讨论的是THREE.HemisphereLight（半球光源），这种光源可以为户外场景创建更加自然的光照效果。然后我们会看一看THREE.AreaLight（区域光源），它可以从一个很大的区域发射光线，而不是从单个点。最后，会展示一下如何在场景中添加镜头光晕的效果。

#### 3.3.1 THREE.HemisphereLight

第一个特殊光源是THREE.HemisphereLight。使用THREE.HemisphereLight，可以创建出更加贴近自然的户外光照效果。如果不使用这个灯光，要模拟户外光照，可以创建一个THREE.DirectionalLight来模拟太阳光，并且可能再添加一个THREE.AmbientLight来为场景提供基础色。但是，这样的光照效果看起来并不怎么自然。在户外，并不是所有的光照都来自上方：很多是来自于大气的散射和地面以及其他物体的反射。Three.js中的THREE.HemisphereLight光源就是为这种情形创建的。它为获得更自然的户外光照效果提供了一种

简单的方式。关于它的示例，可以查看示例 05-hemisphere-light.html，截图如图 3.18 所示。



图 3.18

 注意这是第一个需要加载额外的资源，并且不能从本地的文件系统直接运行的例子。因此，如果你还没有这样做，可以参考第 1 章，了解如何创建一个本地的 Web 服务器，或者禁用浏览器中的安全设置，使之可以加载外部资源。

仔细观察蓝色球体可以发现，该球体表面的底部有接近草地的绿色，而顶部有接近天空的蓝色（通过设置 color 属性获得）。在这个示例中，可以打开或关闭 THREE.HemisphereLight，也可以设置颜色和光强。创建一个半球光光源就像创建其他光源一样简单：

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

你只需要给它指定接收来自天空的颜色，接收来自地面的颜色，以及这些光线的光照强度。之后如果想修改这些属性值，可以使用表 3.6 所列出的属性。

表 3.6

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

### 3.3.2 THREE.AreaLight

我们最后要看的光源是 THREE.AreaLight。使用 THREE.AreaLight，可以定义一个长

方形的发光区域。在旧版 Three.js 中，THREE.AreaLight 并不在标准的 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。而在新版 Three.js 中则可以直接使用 THREE.AreaLight。在深入细节之前先来看一下我们追求的结果（打开 06-area-light.html 示例），如图 3.19 所示。

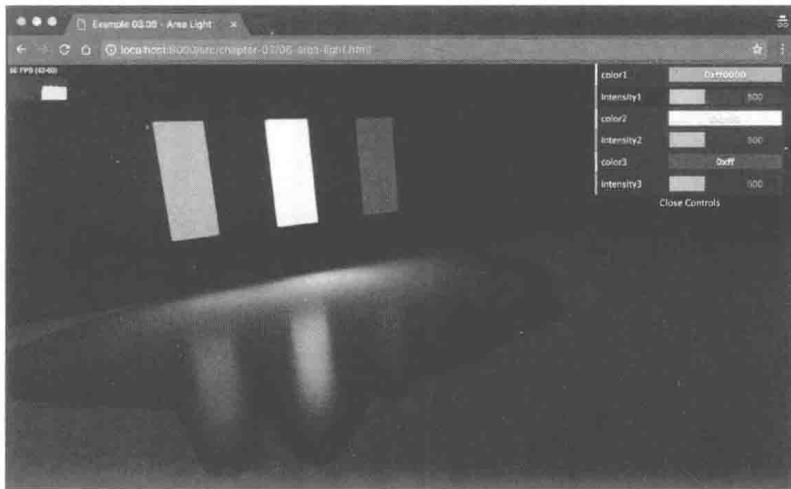


图 3.19

从截图中可以看到，我们定义了三个 THREE.AreaLight 对象，每个都有自己的颜色，你也可以看到这些光源是如何影响整个区域的。要使用 THREE.AreaLight 光源，需要先在 HTML 文件的 head 标签中添加如下导入库：

```
<head>
  <script type="text/javascript" src="../libs/three.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/three/lights/RectAreaLightUniformsLib.js"></script>
</head>
```

添加了上述导入库后，便可以像添加其他光源一样来添加 THREE.AreaLight 光源：

```
var areaLight1 = new THREE.RectAreaLight(0xff0000, 500, 4, 10);
areaLight1.position.set(-10, 10, -35);
scene.add(areaLight1);
```

在这个示例里，创建了一个新的 THREE.AreaLight 对象。这个光源的颜色为 0xff0000，光强的值为 500，width 就 4，height 是 10。与其他光源一样，可以使用 position 属性设置该光源在场景中的位置。在创建 THREE.AreaLight 时，会创建出一个垂直平面。在这个示例中，创建了三个 THREE.AreaLight 对象，有不同的颜色。当你第一次尝试该光源的时候，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？这是因为你不

能看到光源本身，而只能看到它发射出的光，而且只有当这些光照射到某个物体上时才能看到。如果你想创建出例子中所展示的场景，可以在相同的位置（`areaLight1.position`）增加 `THREE.PlanGeometry` 或 `THREE.BoxGeometry` 对象来模拟光线照射的区域，代码如下所示：

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

通过 `THREE.AreaLight` 可以创建出非常漂亮的效果，但是可能要多试验才能获得想要的效果。拉下右上角的控制面板，会找到一些控件来设置场景中三个光源的颜色和光强，并立即看到设置后的效果，如图 3.20 所示。

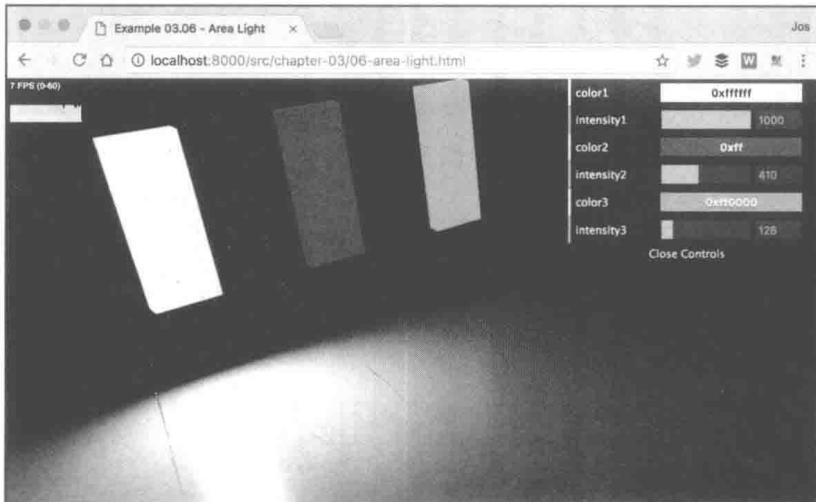


图 3.20

### 3.3.3 镜头光晕

本章将要讨论的最后一个主题是镜头光晕（lens flare）。你可能已经对镜头光晕很熟悉了。例如，当你直接朝着太阳或另一个非常明亮的光源拍照时就会出现镜头光晕效果。在大多数情况下，需要避免出现这种情形，但是对于游戏和三维图像来说，它提供了一种很好的效果，让场景看上去更加真实。

Three.js 库也支持镜头光晕，而且在场景中添加它非常简单。在最后这一节中，我们将在场景中添加一个镜头光晕，并创建出如图 3.21 所示的效果。可以打开示例文件 `07-lensflares.html` 来查看这个效果。



图 3.21

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

```
flare = new THREE.LensFlare(texture, size, distance, blending, color,
opacity);
```

这些参数的含义在表 3.7 中列出。

表 3.7

参 数	描 述
texture (纹理)	纹理就是一个图片，用来决定光晕的形状
size (尺寸)	可以指定光晕多大。这个尺寸的单位是像素。如果将它指定为 -1，那么将使用纹理本身的尺寸
distance (距离)	从光源 (0) 到摄像机 (1) 的距离。使用这个参数将镜头光晕放置在正确的位置
blending (混合)	我们可以为光晕提供多种材质。混合模式决定了如何将它们混合在一起。镜头光晕默认的混合方式是 THREE.AdditiveBlending。更多关于混合的内容将在下一章讲解
color (颜色)	光晕的颜色
opacity (不透明度)	定义光晕的透明度。0 代表完全透明，1 代表完全不透明

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

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

var flareColor = new THREE.Color(0xffaa00);
var lensFlare = new THREE.LensFlare(textureFlare0, 350, 0.0,
  THREE.AdditiveBlending, flareColor);

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

首先需要加载一个纹理。在这个示例里，我使用的是 Three.js 示例库中的镜头光晕纹理，如图 3.22 所示。

如果将这张图与图 3.21 展示的截图相比较，会发现它定义了镜头光晕的样子。接下来，使用 “`new THREE.Color(0xffacc);`” 定义镜头光晕的颜色，这将使镜头光晕泛着红光。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，把光晕的尺寸设置为 350，距离设置为 0.0（就在光源处）。

创建完 `LensFlare` 对象之后，将它放在光源处，添加到场景中，如图 3.23 所示。



图 3.22



图 3.23

这看起来已经很好了，但是如果把这张截图与图 3.21 相比较，你会发现在页面中央少了一些小的圆形失真图形。

我们会使用与创建主光晕相同的方法来创建它们，如下代码所示：

```
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);
```

但是这次并没有创建一个新的 `THREE.LensFlare`，而是使用了刚创建的 `THREE.LensFlare` 对象的 `add` 方法。在这个方法中，只需指定纹理 (`texture`)、尺寸 (`size`)、距离 (`distance`) 和混合 (`blending`) 模式。



注意 add 方法可以接受两个额外的参数。在 add 方法中，你还可以给新光晕设置颜色（color）和不透明度（opacity）属性。这些新光晕使用的纹理是一个颜色很淡的圆，在目录 assets/textures/flares/ 下可以找到本示例所使用的光晕图片。

如果你再来看这个场景，就会发现失真图形出现在用 distance 参数指定的位置。

## 3.4 总结

本章涵盖了 Three.js 库中提供的各种不同灯源，信息量非常大。在本章中，我们学习了配置光源、颜色和阴影，并且知道了它们不是严谨的科学。要获得正确的结果，需要不断试验，使用 dat.GUI 控件可以微调配置。不同的光源以不同的方式表现，正如第 4 章将介绍的，材质也会对光源有不同的反应。THREE.AmbientLight 光源的颜色可以附加到场景中的每一种颜色上，通常用来柔化生硬的颜色和阴影。THREE.PointLight 光源会朝所有方向发射光线，不能被用来创建阴影。THREE.SpotLight 光源类似于手电筒。它有一个锥形的光束，可以配置它随着距离的增大而逐渐变弱，并且可以生成阴影。我们还学习了 THREE.DirectionalLight 光源。这个光源相当于远光的效果，比如太阳光。它的光线彼此平行，其光强并不会随着与目标对象距离的增大而减弱。除了这些标准光源之外，我们还学习了几个更加特殊的光源。如果想要一个更加自然的户外效果，可以使用 THREE.HemisphereLight 光源，它考虑了天空和地面的反射。THREE.AreaLight 不从单个点发射光线，而是从一个很大的区域发射光线。我们还展示了如何通过 THREE.LensFlare 对象添加图像化的镜头光晕。

到本章为止，我们已经介绍了几种不同的材质。在本章，你也看到了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。

## 使用 Three.js 的材质

前面的章节中简单地提了一下材质。你已经了解到一个材质结合 THREE.Geometry 对象，可以构成 THREE.Mesh 对象。材质就像物体的皮肤，决定了几何体的外表。例如，皮肤定义了一个几何体看起来是否像金属、透明与否，或者显示为线框。然后得到的 THREE.Mesh 对象才可以添加到 Three.js 渲染的场景中。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 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创建漂亮的三维对象。本章将要探讨的材质如表 4.1 所示。

表 4.1

名 称	描 述
MeshBasicMaterial (网格基础材质)	基础材质，用于给几何体赋予一种简单的颜色，或者显示几何体的线框
MeshDepthMaterial (网格深度材质)	这个材质使用从摄像机到网格的距离来决定如何给网格上色
MeshNormalMaterial (网格法向材质)	这是一种简单的材质，根据法向向量计算物体表面的颜色
MeshLambertMaterial (网格 Lambert 材质)	这是一种考虑光照影响的材质，用于创建暗淡的、不光亮的物体
MeshPhongMaterial (网格 Phong 式材质)	这是一种考虑光照影响的材质，用于创建光亮的物体
MeshStandardMaterial (网格标准材质)	这种标准材质使用“基于物理的渲染 (PBR)”算法来绘制物体表面。它能够计算出表面与光线的正确互动关系，从而使渲染出的物体看起来更加真实
MeshPhysicalMaterial (网格物理材质)	这是 MeshStandardMaterial 的扩展材质，它为光线反射计算模型提供了更多的控制
MeshToonMaterial (网格卡通材质)	这是 MeshPhongMaterial 的扩展材质，它使得物体渲染更加卡通化



(续)

名 称	描 述
ShadowMaterial (阴影材质)	这是一个专门用于接收阴影图的特殊材质。在该材质中只有阴影图像，非阴影部分为完全透明的区域
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
LineBasicMaterial (直线基础材质)	这种材质可以用于 THREE.Line (直线) 几何体，用来创建着色的直线
LineDashMaterial (虚线材质)	这种材质与 LineBasicMaterial (直线基础材质) 一样，但允许创建出一种虚线的效果

如果浏览 Three.js 库的源代码，你可能看到过 THREE.RawShaderMaterial。这是一种特殊的材质，只能和 THREE.BufferedGeometry 一起使用。此几何体是用来优化静态几何体（如顶点和面不会改变的几何体）的一种特殊形式。本章并不会探讨这种材质，但会在第 11 章使用它。在源代码中，你也可以找到 THREE.SpriteMaterial 和 THREE.PointMaterial，这些材质用来给单个点设置样式。本章不会讨论这些材质，但我们在第 7 章探讨它们。

材质对象有一些共同的属性，所以在讨论第一个材质 MeshBasicMaterial 之前，我们先看一下所有材质共有的属性。

## 4.1 理解材质的共有属性

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，它列出了所有的共有属性。我们将这些共有属性分成了三类，如下所示：

- **基础属性**：这些属性是最常用的。通过这些属性，可以控制物体的不透明度、是否可见以及如何被引用（通过 ID 或是自定义名称）。
- **融合属性**：每个物体都有一系列的融合属性。这些属性决定了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 上下文对象渲染物体的方式。大多数情况下是不需要使用这些属性的。

请注意，在本章，我们会跳过任何与纹理和贴图相关的属性。大多数材质允许使用图片作为纹理（如木质或类岩石材质）。在第 10 章，会对各种纹理和可用的贴图选项进行深入的探讨。有些材质也有与动画相关的特殊属性（skinning、morphNormals 和 morphTargets），这些属性也可以略过，在第 9 章会讨论它们。而 clipIntersection、clippingPlanes 和 clipShadow 3 个属性则会在第 6 章讨论。

我们将从基础属性开始。

### 4.1.1 基础属性

THREE.Material 对象的基础属性列在表 4.2 中。（在后续章节中介绍 THREE.

BasicMeshMaterial 材质时，有对这些属性的实际操作。)

表 4.2

属性	描述
<code>id</code> (标识符)	用来识别材质，并在材质创建时赋值。第一个材质的值从 0 开始，每新加一个材质，这个值增加 1
<code>uuid</code> (通用唯一识别码)	这是生成的唯一 ID，在内部使用
<code>name</code> (名称)	可以通过这个属性赋予材质名称，用于调试的目的
<code>opacity</code> (不透明度)	定义物体的透明度。与 <code>transparent</code> 属性一起使用。该属性的赋值范围从 0 到 1
<code>transparent</code> (是否透明)	如果该值设置为 <code>true</code> ，Three.js 会使用指定的不透明度渲染物体。如果设置为 <code>false</code> ，这个物体就不透明——只是着色更明亮些。如果使用 <code>alpha</code> (透明度) 通道的纹理，该属性应该设置为 <code>true</code>
<code>overdraw</code> (过度描绘)	当你使用 <code>THREE.CanvasRender</code> 时，多边形会被渲染得稍微大一点。当使用这个渲染器渲染的物体有间隙时，可以将这个属性设置为 <code>true</code>
<code>visible</code> (是否可见)	定义该材质是否可见。如果设置为 <code>false</code> ，那么在场景中就看不到该物体
<code>side</code> (侧面)	通过这个属性，可以定义几何体的哪一面应用材质。默认值为 <code>THREE.FrontSide</code> (前面)，这样可以将材质应用到物体的前面 (外侧)。也可以将其设置为 <code>THREE.BackSide</code> (后面)，这样可以将材质应用到物体的后面 (内侧)。或者也可以将它设置为 <code>THREE.DoubleSide</code> (双侧)，可将材质应用到物体的内外两侧
<code>needsUpdate</code> (是否更新)	对于材质的某些修改，你需要告诉 Three.js 材质已经改变了。如果该属性设置为 <code>true</code> ，Three.js 会使用新的材质属性更新它的缓存
<code>colorWrite</code> (是否输出颜色)	如果该属性值为 <code>false</code> ，则具有该材质的物体不会被真正绘制到场景中。实际效果是该物体本身是不可见的，但其他物体被它挡住的部分也不可见
<code>flatShading</code> (平面着色)	该属性控制物体表面法线的生成方式，从而影响光照效果。属性值为 <code>true</code> 时，在两个相邻但不共面的三角形之间，光照会因为生硬过渡而产生棱角；为 <code>false</code> 时则会产生非常平滑的过渡效果
<code>lights</code> (光照)	该属性值为布尔值，控制物体表面是否接受光照。默认值为 <code>true</code>
<code>premultipliedAlpha</code> (预计算 Alpha 混合)	该属性控制半透明表面的颜色混合方式。默认值为 <code>false</code>
<code>dithering</code> (抖动)	该属性控制是否启用颜色抖动模式。该模式可以在一定程度上减轻颜色不均匀的问题。默认值为 <code>false</code>
<code>shadowSide</code> (投影面)	这个属性与上面的 <code>side</code> 属性有些类似，但它控制的是物体的哪个面会投射阴影。默认值为 <code>null</code> 。当该属性值为 <code>null</code> 时，投射阴影的面按照如下原则推定： 当 <code>side</code> 为 <code>THREE.FrontSide</code> 时， <code>shadowSide</code> 为后面； 当 <code>side</code> 为 <code>THREE.BackSide</code> 时， <code>shadowSide</code> 为前面； 当 <code>side</code> 为 <code>THREE.DoubleSide</code> 时， <code>shadowSide</code> 也为前面双侧

(续)

属性	描述
vertexColors (顶点颜色)	你可以为物体的每一个顶点指定特有的颜色。该属性的默认值为 THREE.NoColors。如果设定属性值为 THREE.VertexColors，则渲染时将使用 THREE.Face3 vertexColors 数组指定的颜色，为每一个顶点设定颜色。如果该属性值为 THREE.FaceColors，则会使用每一个面自己的颜色属性来设定面的颜色。CanvasRenderer 不支持该属性，但 WebGLRenderer 能够支持。LineBasicMaterial 示例将展示如何会使用该属性为一条线的不同部分设定颜色
fog (雾)	该属性控制材质是否受到雾的影响。后续示例中并没有使用这个属性。如果这个属性设置为 false，则在第 2 章的全局雾化效果示例中，场景中的物体将不会有雾化效果

对于每个材质还可以设置一些融合属性。

### 4.1.2 融合属性

材质有几个与融合相关的一般属性。融合决定了我们渲染的颜色如何与它们后面的颜色交互。稍后讨论合并材质的时候，会探讨这个主题。融合属性如表 4.3 所示。

表 4.3

名称	描述
blending (融合)	该属性决定物体上的材质如何与背景融合。一般的融合模式是 THREE.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，还可以通过设置 blendSrc、blendDst 和 blendEquation 来创建自定义的融合模式。这个属性定义了该物体（源）如何与背景（目标）相融合。默认值为 THREE.SrcAlphaFactor，即使用 alpha（透明度）通道进行融合
blendSrcAlpha (融合源透明度)	该属性为 blendSrc 指定透明度。默认值为 null
blendDst (融合目标)	这个属性定义了融合时如何使用背景（目标），默认值为 THREE.OneMinusSrcAlphaFactor，其含义是目标也使用源的 alpha 通道进行融合，只是使用的值是 1（源的 alpha 通道值）
blendDstAlpha (融合目标透明度)	该属性为 blendDst 指定透明度。默认值为 null
blendEquation (融合公式)	定义了如何使用 blendSrc 和 blendDst 的值。默认值为使它们相加 (AddEquation)。通过使用这三个属性，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。它们与 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 规范是一个很好的起点。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](http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf)。这些高级属性的简要说明如表 4.4 所示。

表 4.4

名 称	描 述
depthTest	这是一个高级 WebGL 属性。使用这个属性可以打开或关闭 GL_DEPTH_TEST 参数。此参数控制是否使用像素深度来计算新像素的值。通常情况下不必修改这个属性。更多信息可以在我们前面提到的 OpenGL 规范中找到
depthWrite	这是另外一个内部属性。这个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（例如一个套子），应该将这个属性设置为 false。但是，通常不应该修改这个属性
depthFunc (深度测试函数)	该属性指定所使用的深度测试算法，它对应着 WebGL 规格中的 glDepthFunc 属性。
polygonOffset、polygonOffsetFactor 和 polygonOffsetUnits	通过这些属性，可以控制 WebGL 的 POLYGON_OFFSET_FILL 特性。一般不需要使用它们。有关这些属性具体做什么的解释，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 值小于该值，那么该像素不会显示出来。可以使用这个属性移除一些与透明度相关的毛边
precision (精度)	设置当前材质的计算精度，可使用 WebGL 参数值：highp、mediump 或 lowp

现在我们来看看所有可用的材质，你会看出这些属性对渲染结果的影响。

## 4.2 从简单的网格材质开始

本节将介绍几种简单的材质：MeshBasicMaterial、MeshDepthMaterial 和 MeshNormalMaterial。我们将从 MeshBasicMaterial 开始。

在开始研究这些材质的属性之前，先来浏览一下关于如何通过传入属性来配置材质的说明。有两个选项：

- 可以在构造函数中通过参数对象的方式传入参数：

```
var material = new THREE.MeshBasicMaterial(
{
    color: 0xff0000, name: 'material-1', opacity: 0.5,
    transparency: true, ...
});
```

- 另外，还可以创建一个实例，并分别设置属性，如下所示：

```
var material = new THREE.MeshBasicMaterial();
material.color = new THREE.Color(0xff0000);
material.name = 'material-1';
material.opacity = 0.5;
material.transparency = true;
```

一般来说，如果知道所有属性的值，最好的方式是在创建材质对象时使用构造方法传入参数。这两种方式中参数使用相同的格式。唯一例外的是 color 属性。在第一种方式中，可以只传入十六进制值，Three.js 会自己创建一个 THREE.Color 对象。而在第二种方式中，必须创建一个 THREE.Color 对象。在这本书中，我们对这两种方式都会进行介绍。

#### 4.2.1 THREE.MeshBasicMaterial

MeshBasicMaterial 是一种非常简单的材质，这种材质不考虑场景中光照的影响。使用这种材质的网格会被渲染成简单的平面多边形，而且也可以显示几何体的线框。除了在上一节提及的那些共有属性之外，还可以设置表 4.5 所列的这些属性。（我们在此处仍然会忽略与材质有关的属性，因为这些属性将在材质专属的章节讨论。）

表 4.5

名 称	描 述
color (颜色)	设置材质的颜色
wireframe (线框)	设置这个属性可以将材质渲染成线框，非常适用于调试
wireframeLineWidth (线框线宽)	如果已经打开了 wireframe，这个属性定义线框中线的宽度
wireframeLinecap (线框线段端点)	这个属性定义了线框模式下顶点间线段的端点如何显示。可选的值包括 butt (平)、round (圆) 和 square (方)。默认值为 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)	这个属性定义了线段的连接点如何显示。可选的值有 round (圆)、bevel (斜角) 和 miter (尖角)。默认值为 round。如果你在一个使用低透明度和 wireframeLineWidth 值很大的例子里靠近观察，就可以看到这个属性的效果。WebGLRenderer 对象不支持该属性

在前面我们已经了解了如何创建材质，并把它们赋给对象。对于 THREE.MeshBasicMaterial，设置如下：

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

这会创建一个 THREE.MeshBasicMaterial 对象，并将其颜色属性初始化为 0x7777ff（紫色）。

我添加了一个例子，你可以用它试验一下 THREE.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，如图 4.1 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，并且选择各种类型的网格（你也可以修改渲染器）。图 4.2 展示了一个地鼠模型在线框模式下的渲染效果。

也可以像图 4.3 所示的那样，选择使用 THREE.CanvasRenderer，看一看 THREE.MeshBasicMaterial 的各种属性在这个渲染器上会产生什么效果。

在这个例子中有一个可以设置的属性为 side。通过这个属性，可以指定在 THREE.Geometry 对象的哪一面应用材质。通过选择 plane（平面）网格，可以检验该属性是如何起作用的。由于材质通常应用在物体前面的面上，所以在平面旋转的时候会有一半时间看不见

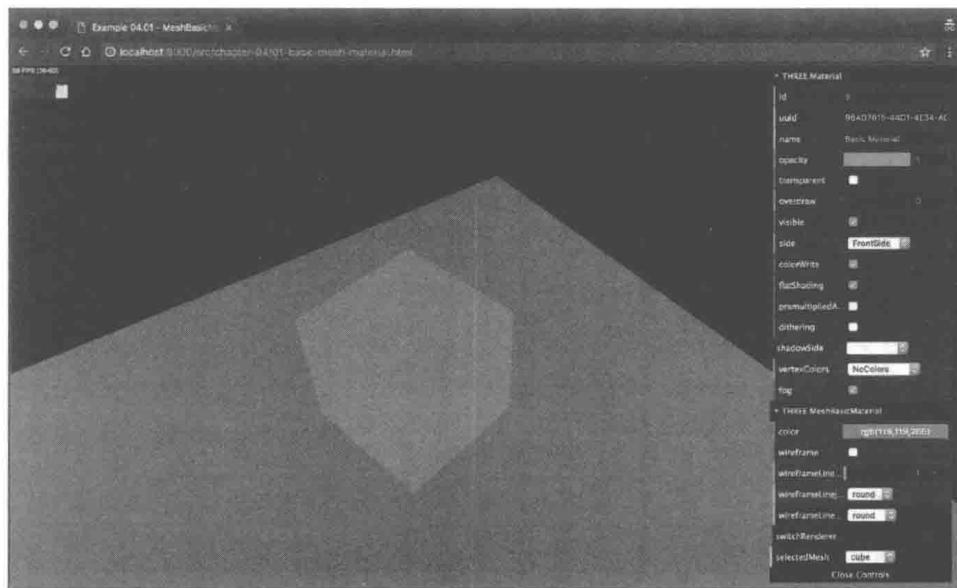


图 4.1

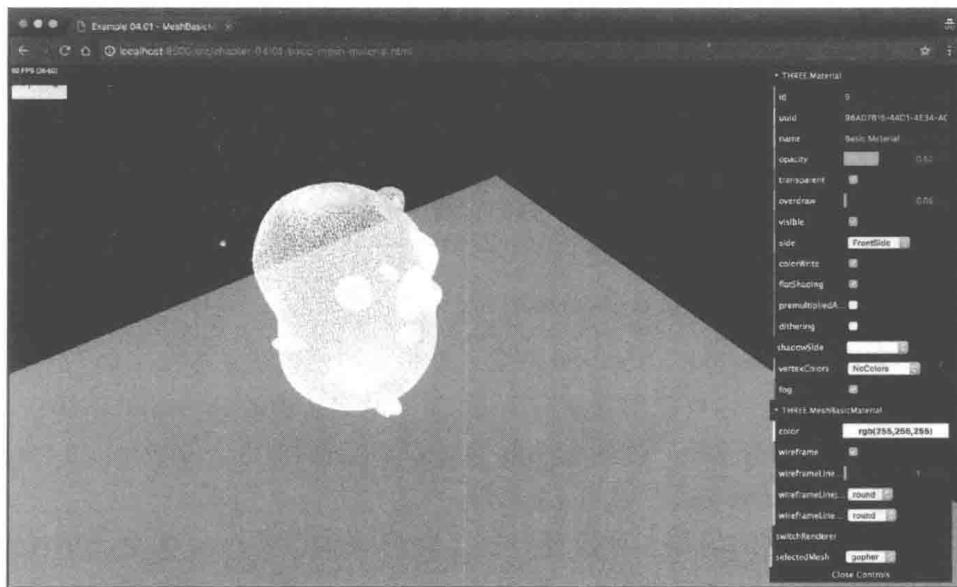


图 4.2

到它（当它背面朝向你时）。如果将 side 属性设置为 double，那么由于几何体两面都有材质，这个平面始终都可以看见。尽管如此，仍要注意，当 side 属性设置为 double 时，渲染

器需要做更多的工作，所以对场景的性能会有影响。

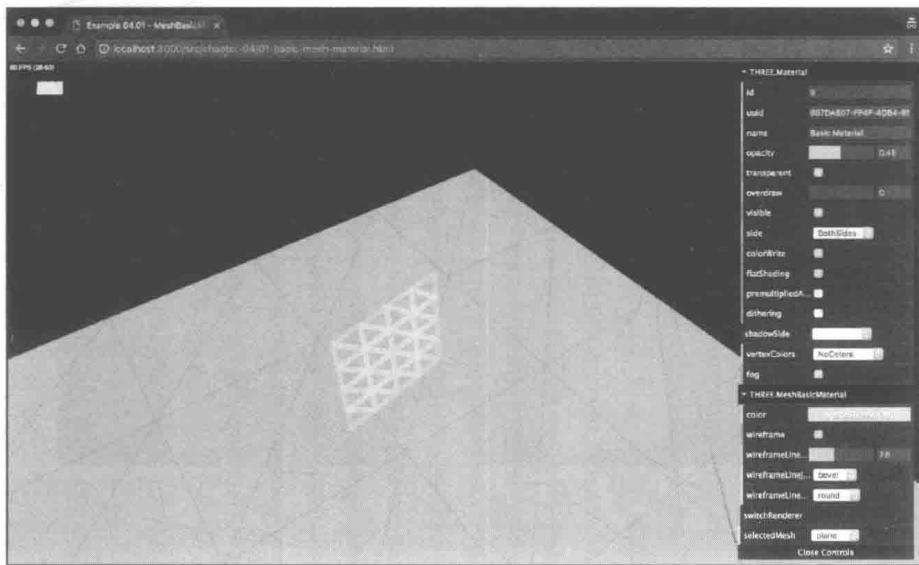


图 4.3

#### 4.2.2 THREE.MeshDepthMaterial

下一个材质是 THREE.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，而是由物体到摄像机的距离决定的。可以将这种材质与其他材质结合使用，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显示的属性，如表 4.6 所示。

表 4.6

名 称	描 述
wireframe	该属性指定是否显示线框
wireframeLineWidth	该属性指定线框线的宽度（这个属性只对 THREE.CanvasRenderer 有效）

为了展示该材质，我们修改了第 2 章中方块的例子（chapter-04 文件夹下 02-depth-material.html）。记住，必须点击 addCube 按钮才能在场景中添加方块。示例修改之后的屏幕截图如图 4.4 所示。

它展示了渲染过的方块，基于距摄像机的距离上色。尽管这种材质没有多少属性可以控制物体的渲染效果，但我们仍然可以控制物体颜色消失的速度。在这个例子里，将摄像机的 near 和 far 属性展现出来。你可能还记得，在第 2 章中，通过使用这两个属性，可以设置摄像机的可视区域。所有与摄像机距离小于 near 属性的物体不会显示出来，而所有与摄像机距离大于 far 属性的物体也都落在摄像机的可视区域之外。

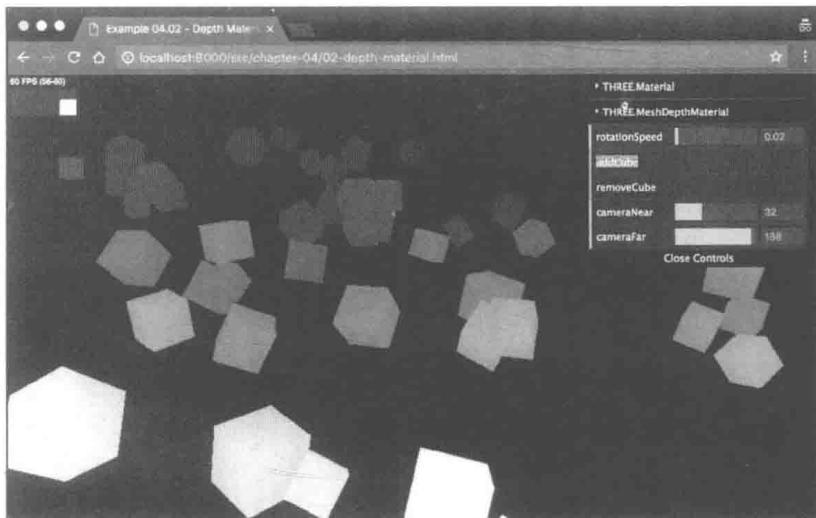


图 4.4

摄像机 near 属性和 far 属性之间的距离决定了场景的亮度和物体消失的速度。如果这个距离非常大，那么当物体远离摄像机时，只会稍微消失一点。如果这个距离非常小，那么物体消失的效果会非常明显，如图 4.5 所示。

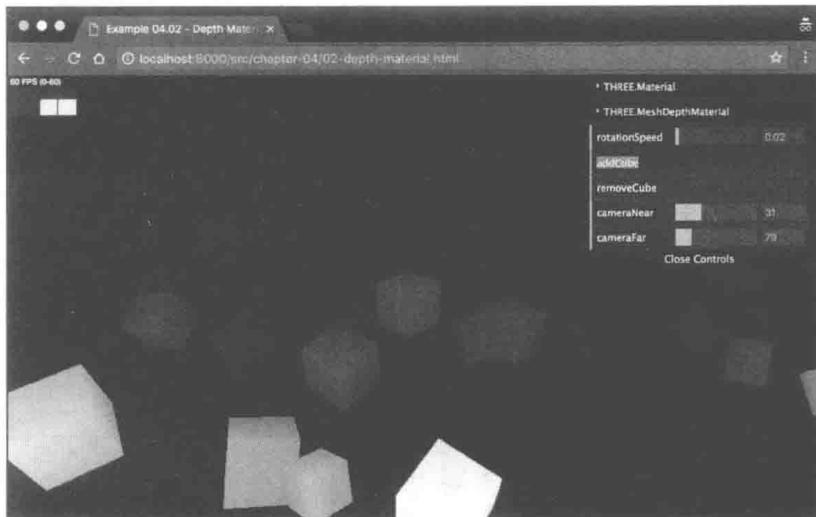


图 4.5

方块由摄像机的 near 和 far 属性之间的小距离渲染。创建 THREE.MeshDepthMaterial 对象非常简单，并且这个对象不需要什么参数。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，而不需要在每个 THREE.Mesh 对象上显式地声明它：

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

本章接下来要谈论的主题并不是某种特定的材质，而是讲述将多个材质联合在一起使用的方法。

### 4.2.3 联合材质

如果回头看看 THREE.MeshDepthMaterial 的属性，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）。下面的代码展示了如何联合材质：

```
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);
```

这样就可以获得如图 4.6 所示的绿色方块，这些方块从 THREE.MeshDepthMaterial 对象获得亮度，并从 THREE.MeshBasicMaterial 获得颜色（打开 03-combined-material.html 查看例子）。效果如图 4.6 所示。



图 4.6

让我们来看看要达到这种效果需要采取的步骤。

首先，需要创建两种材质。对于 THREE.MeshDepthMaterial，没什么特别要做的；但是对于 THREE.MeshBasicMaterial，要把 transparent 属性设置为 true，并且指定一个融合模式。如果不将 transparent 属性设置为 true，就只会得到纯绿色的物体，因为 Three.js 不会执行任何融合操作。如果将 transparent 属性设置为 true，Three.js 就会检查 blending 属性，以查看这个绿色的 THREE.MeshBasicMaterial 材质如何与背景相互作用。这里所说的背景是用 THREE.MeshDepthMaterial 材质渲染的方块。

在这个例子里用的是 THREE.MultiplyBlending 对象。这种模式会把前景色和背景色相乘，得到想要的结果。上述代码片段的最后一行也很重要。当调用 THREE.SceneUtils.createMultiMaterialObject() 方法创建一个网格的时候，几何体会被复制，返回一个网格组，里面的两个网格完全相同。如果没有最后一行，那么在渲染的时候画面会有闪烁。当渲染的物体有一个在别的物体上，并且有一个物体是透明的，这种情况有时会发生。通过缩小带有 THREE.MeshDepthMaterial 材质的网格，就可以避免这种现象。可通过如下代码实现：

```
cube.children[1].scale.set(0.99, 0.99, 0.99);
```

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 THREE.MeshNormalMaterial

要理解这种材质是如何渲染的，最简单的方式是查看一个例子。打开 chapter-04 文件夹下的例子 04-mesh-normal-material.html。如果选择 sphere（球体）作为网格，你就会看到如图 4.7 所示的图像。（需要将 flatshading 属性设置为 true，以便产生下图中的效果。）

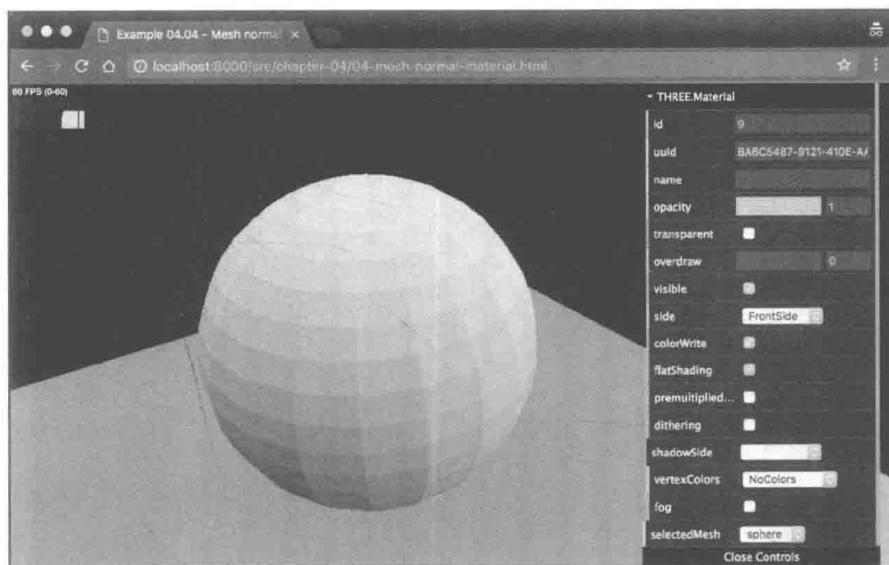


图 4.7

正如你所看到的，网格的每一面渲染的颜色都稍有不同，而且即使在球体旋转时，这些颜色也基本保持在原来的位置。之所以会这样，是因为每一面的颜色是由从该面向外指的法向量计算得到的。所谓法向量是指与面垂直的向量。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，有助于将纹理映射到三维模型，并提供有关如何计算光照、阴影和为表面像素着色的信息。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，所以不需要自己计算。图 4.8 显示了所有 THREE.SphereGeometry 对象上的法向量。

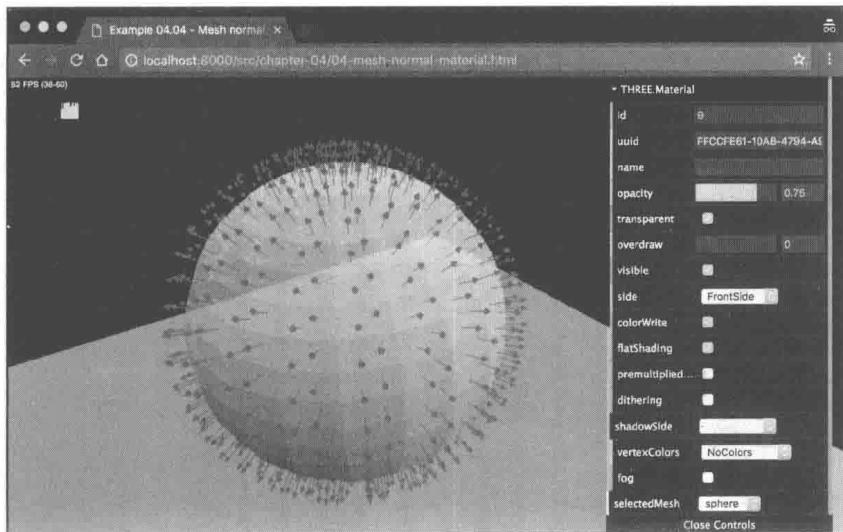


图 4.8

法向量所指的方向决定了在使用 THREE.MeshNormalMaterial 材质时，每个面获取的颜色。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体。要添加这些法向量，可以使用 THREE.ArrowHelper 对象，如下代码所示：

```
for (var f = 0, fl = sphere.geometry.faces.length; f < fl; f++) {
    var face = sphere.geometry.faces[f];
    var centroid = new THREE.Vector3(0, 0, 0);
    centroid.add(sphere.geometry.vertices[face.a]);
    centroid.add(sphere.geometry.vertices[face.b]);
    centroid.add(sphere.geometry.vertices[face.c]);
    centroid.divideScalar(3);

    var arrow = new THREE.ArrowHelper(face.normal, centroid, 2,
        0x3333FF, 0.5, 0.5);
    sphere.add(arrow);
}
```

在这段代码中，我们遍历 THREE.SphereGeometry 的所有面。对于每个 THREE.Face3 对象来说，我们通过把构成该面的顶点相加再除以 3 来计算中心（质心）。使用这个质心连同该面的法向量来绘制箭头。THREE.ArrowHelper 对象接受以下参数：direction、origin、length、color、headLength 和 headWidth。

MeshNormalMaterial 还有几个属性可以设置，具体见表 4.7。

表 4.7

名称	描述
wireframe	该属性指定是否显示线框
wireframeLineWidth	该属性指定线框线的宽度

在 4.2.1 节的那个例子里，我们已经见识过 `wireframe` 和 `wireframeLineWidth` 属性了，但是略过了 `shading` 属性。通过 `shading` 属性，我们可以告诉 Three.js 如何渲染物体。如果使用 `THREE.FlatShading`，那么每个面是什么颜色就渲染成什么颜色（正如你在前面的那些截图中看到的那样），或者也可以使用 `THREE.SmoothShading`，这样就可以使物体的表面变得更光滑。举例来说，如果我们使用 `THREE.SmoothShading` 对象来渲染球体，得到的结果如图 4.9 所示。

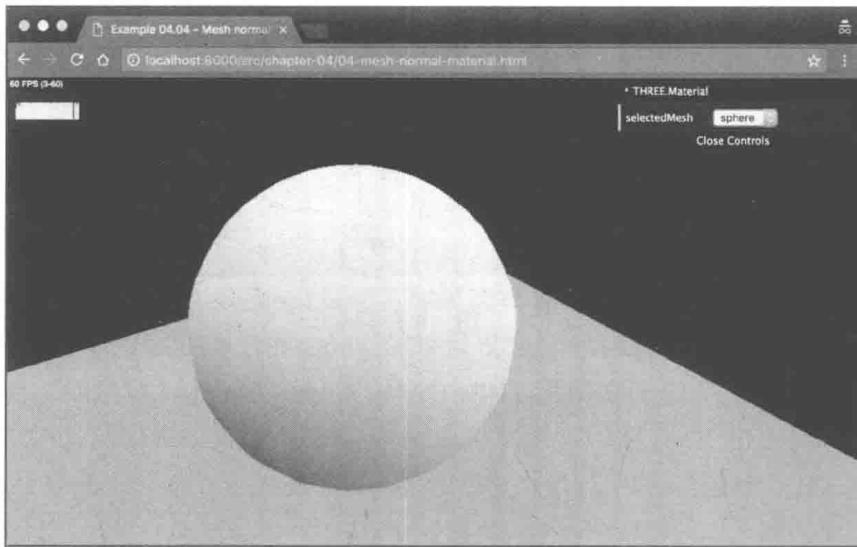


图 4.9

简单材质基本讲完了。在开始下一章之前，这里只剩下最后一个主题：我们将看一看如何在一个 3D 形体上为特定的面指定不同的材质。

#### 4.2.5 在单几何体上使用多种材质

在创建 `THREE.Mesh` 时，到目前为止，我们已经使用过一种材质。为几何体的每个面指定不同的材质是可能的。例如，如果你有一个方块，它有 12 个面（请记住，Three.js 只作用于三角形），你可以用这种材质给方块的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，如下代码所示：

```
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: 0xffffffff }));

var cubeGeom = new THREE.BoxGeometry(3,3,3);
var cube = new THREE.Mesh(cubeGeom, matArray);
```

我们先创建了一个名为 matArray 的数组，用来保存所有的材质，并用该数组来创建 THREE.Mesh 对象。你可能已经注意到虽然方块几何体有 12 个面，但这里却只向数组中添加了 6 个材质对象。要想理解为什么可以这样做，我们需要观察一下 Three.js 如何将材质指定给特定的面。在 Three.js 中，几何体的每一个面都具有一个 materialIndex 属性。该属性指定了该面将使用哪一个具体的材质。可以像下面那样在浏览器的控制台窗口中观察 THREE.CubeGeometry 对象的每一个面的 materialIndex 属性值：

```
> cubeGeom.faces.forEach((p, i) => console.log("face " + i + " : "
+p.materialIndex))
face 0 : 0
face 1 : 0
face 2 : 1
face 3 : 1
face 4 : 2
face 5 : 2
face 6 : 3
face 7 : 3
face 8 : 4
face 9 : 4
face 10 : 5
face 11 : 5
```

可以看出，Three.js 已经自动将方块几何体的面按照其所在侧面组合在一起，所以我们不必提供 12 个材质对象，而是只提供 6 个即可，每一个材质将应用于一整个侧面。让我们深入研究一下这段代码，来看看要再现下面的例子（一个三维魔方）需要做些什么。本例的文件名为 05-mesh-facematerial，屏幕截图如图 4.10 所示。

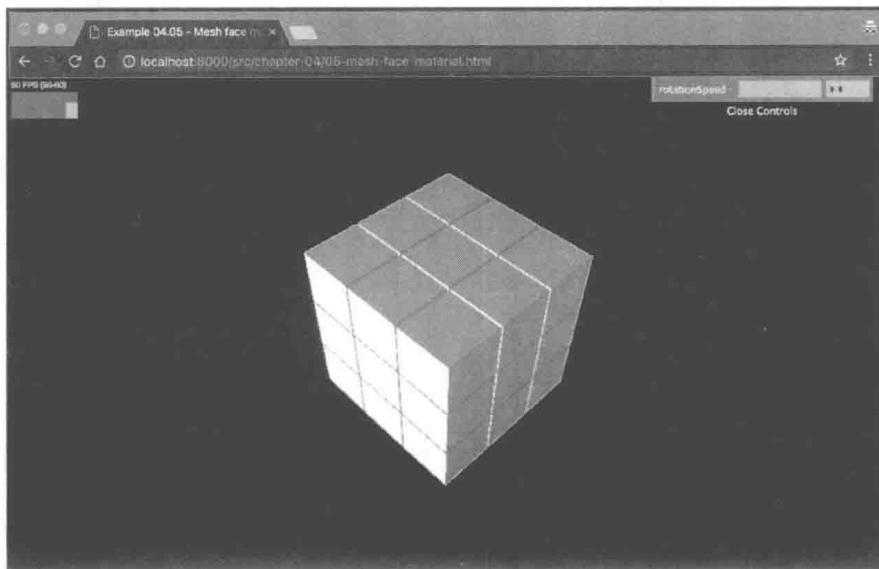


图 4.10

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: 0xffffffff }));
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.BoxGeometry(2.9, 2.9, 2.9);
            var cube = new THREE.Mesh(cubeGeom, mats);
            cube.position.set(x * 3 - 3, y * 3 - 3, z * 3 - 3);
            group.add(cube);
        }
    }
}
}

```

在这段代码里，先创建了一个 THREE.Mesh 对象，它（group 对象）将用来保存所有的方块；接下来，为每个面创建一个材质，并将它们保存在 mats 数组中。接着会创建三个循环，以保证创建出正确数量的方块。在循环里会创建每一个方块、赋予材质、定位，并把它们添加到组中。你应该记住的是，方块的位置是相对于组的位置的。如果移动或旋转组，所有的立方体将随之移动和旋转。更多有关如何使用组的信息，请参考第 8 章。

如果你已经在浏览器中打开了这个例子，将可以看到整个魔方在旋转，而不是单个方块在旋转。这是因为我们在渲染循环里添加了如下代码：

```

group.rotation.y = step += controls.rotationSpeed;
group.rotation.z = step -= controls.rotationSpeed;
group.rotation.x = step += controls.rotationSpeed;

```

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，确保它们放在中心点的周围。这就是为什么会在 cube.position.set(x\*3-3, y\*3, z\*3-3) 这行代码看到 -3 这个偏移值。

## 4.3 高级材质

本节介绍 Three.js 库提供的高级材质。首先我们看一下 THREE.MeshPhongMaterial 和 THREE.MeshLambertMaterial。这两种材质会对光源做出反应，并且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：THREE.ShaderMaterial。通过 THREE.ShaderMaterial，可以创建自己的着色程序，定义材质和物体如何显示。

### 4.3.1 THREE.MeshLambertMaterial

这种材质可以用来创建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众多基本属性。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，如表4.8所示。

表 4.8

名 称	描 述
color (颜色)	这是材质的环境色
emissive (自发光)	材质自发光的颜色。该属性虽然不会让使用它的物体变成光源，但会使物体的颜色不受其他光源的影响。(译注：作者在这里的意思是即使在场景中没有光源的暗处，物体表面的emissive颜色也可见，从而可以实现物体自发光。) 该属性的默认值为黑色

创建这个材质与其他材质都一样，如下所示：

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

关于这个材质的例子，可以查看06-mesh-lambert-material.html。此例效果如图4.11所示。

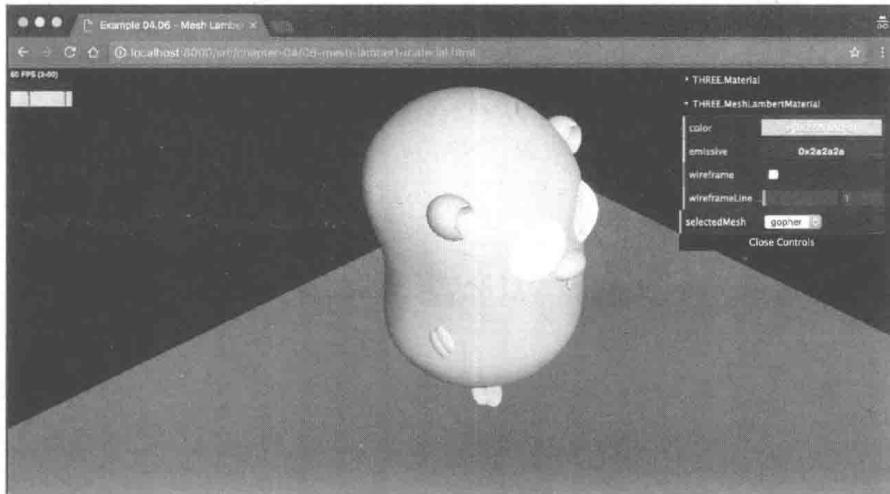


图 4.11

上面截图的场景中有一个橘黄色、带有轻微自发光效果。THREE.LambertMaterial材质的另一个有趣的功能是可以支持线框绘制属性，因此利用它可以绘制具有光照效果的线框物体，如图4.12所示。

从上面的场景截图可见THREE.LambertMaterial材质的效果看起来比较暗淡。下面将要介绍的THREE.MeshPhongMaterial材质则可以使物体显得非常光亮。

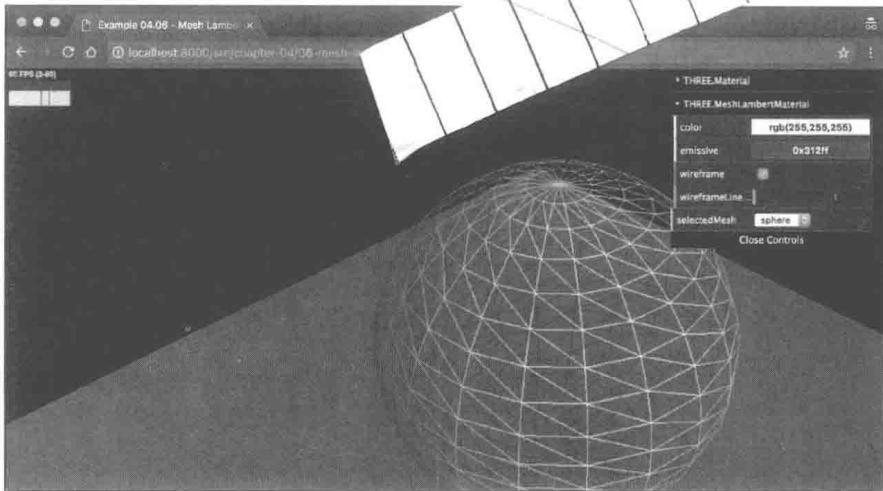


图 4.12

### 4.3.2 THREE.MeshPhongMaterial

通过 THREE.MeshPhongMaterial 可以创建一种光亮的材质。这种材质可使用的属性与暗淡材质 THREE.MeshLambertMaterial 可使用的属性基本一样。在旧版 Three.js 中，这是唯一一种既可以在物体表面实现高光效果的材质。它既可以模拟塑料质感，也可以在一定程度上模拟金属质感。新版 Three.js 提供了 THREE.MeshStandardMaterial 和 THREE.MeshPhysicalMaterial 两个新型材质。不过在讨论它们之前，还是先要学习一下传统的 THREE.MeshPhongMaterial 材质。

我们同样会略过那些基础的、已经讨论过的属性。这种材质中比较有意义的属性列在表 4.9 中。

表 4.9

名 称	描 述
color (颜色)	这是材质的环境色。它与上一章讲过的环境光源一起使用。这个颜色会与环境光提供的颜色相乘。默认值为白色
emissive (自发光)	材质自发光的颜色。它与 THREE.MeshLambertMaterial 对应的属性作用相同
specular (高光颜色)	该属性指定该材质的光亮程度及高光部分的颜色。如果将它设置成与 color 属性相同的颜色，将会得到一个更加类似金属的材质。如果将它设置成灰色 (grey)，材质将变得更像塑料
shiness (高光度)	该属性指定物体表面镜面高光部分的轮廓的清晰程度。越光滑的表面，高光部分越清晰，反之则越模糊。该属性的默认值为 30

THREE.MeshPhongMaterial 对象的初始化与我们看过的其他材质都一样，如下代码所示：

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

为了更好地进行比较，我们为这个材质创建了一个与 THREE.MeshLambertMaterial 的例子一样的示例。可以通过 GUI 控制界面来试验一下这个材质。例如，图 4.13 的设置可以创建一个看起来像塑料的材质。可以打开 07-mesh-phong-material.html 来看看这个例子。

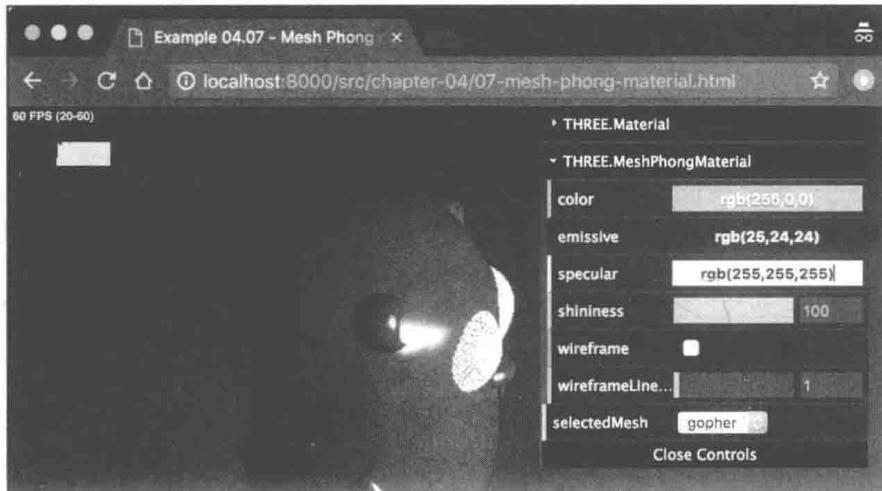


图 4.13

上面截图展示了如何使用 THREE.MeshPhongMaterial 材质来创建塑料质感的物体。Three.js 还提供了一个 THREE.MeshPhongMaterial 材质的扩展材质：THREE.MeshToonMaterial。它的属性与 THREE.MeshPhongMaterial 完全相同，但是会以不同的方式渲染物体，如图 4.14 所示。



图 4.14

前面提到过新版 Three.js 提供了两个新型材质：THREE.MeshStandardMaterial（标准

材质) 和 THREE.MeshPhysicalMaterial (物理材质)。使用它们可以实现更接近真实世界的渲染效果。接下来我们就学习一下这两个材质。

### 4.3.3 THREE.MeshStandardMaterial

THREE.MeshStandardMaterial 这种材质使用更加正确的物理计算来决定物体表面如何与场景中的光源互动。这种材质不但能够更好地表现塑料质感和金属质感的表面，而且像开发者提供了表 4.10 中所列的两个新属性。

表 4.10

名 称	描 述
metalness (金属感程度)	该属性控制物体表面的金属感程度。0 代表完全塑料质感，1 代表完全金属质感。默认值为 0.5
roughness (粗糙程度)	该属性控制物体表面的粗糙程度。在视觉上，它决定表面对入射光的漫反射程度。默认值为 0.5。当值为 0 时会产生类似镜面的反射，为 1 时会产生完全的漫反射效果

除了上面两个属性之外，其他基础材质属性，如 color (颜色) 和 emissive (自发光) 等也可以用来改变这种材质的效果。图 4.15 中的物体通过调节 metalness 和 roughness 两个属性实现了一种哑光金属表面。

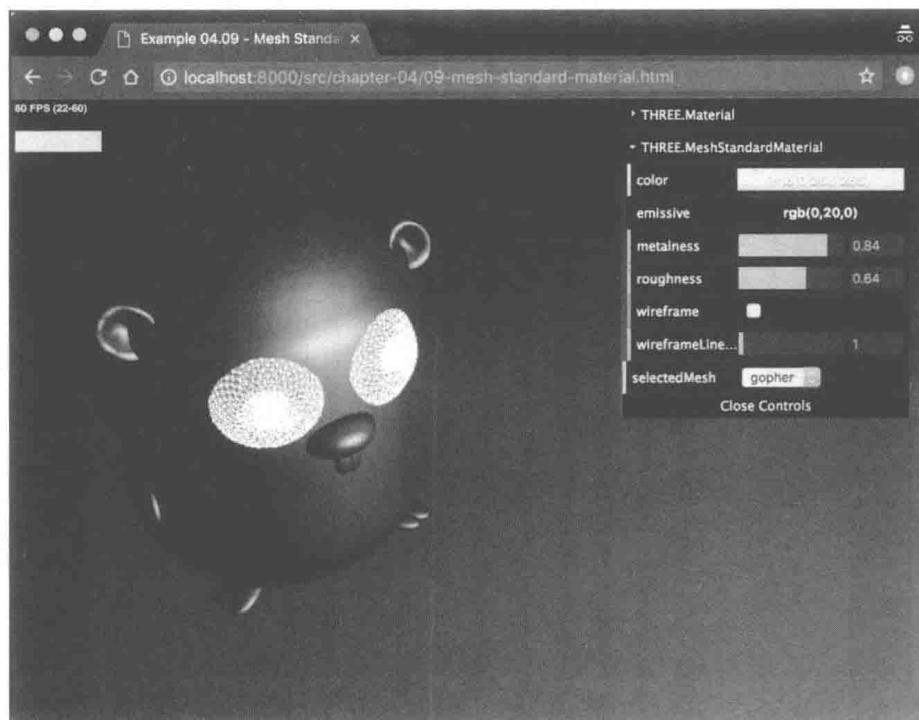


图 4.15



### 4.3.4 THREE.MeshPhysicalMaterial

该材质与 THREE.MeshStandardMaterial 非常相似，但提供了对反光度的更多控制，因此它除了具有标准材质的全部属性之外，还增加了表 4.11 列出新属性。

表 4.11

名 称	描 述
clearCoat (清漆)	该属性控制物体表面清漆涂层效果的明显程度。该属性值越高，则清漆涂层越厚，其结果是 clearCoatRoughness 属性带来的影响越明显。取值范围在 0 到 1 之间。默认值为 0
clearCoatRoughness (清漆粗糙度)	该属性控制物体表面清漆涂层的粗糙程度。粗糙程度越高，漫反射越明显。该属性需要与上面列出的 clearCoat 属性配合使用。取值范围在 0 到 1 之间。默认值为 0
reflectivity (反光度)	该属性用于控制非金属表面的反光度，因此当 metalness (金属感程度) 为 1 或接近 1 时该属性的作用很小。取值范围在 0 到 1 之间。默认值为 0.5

与其他材质类似，这两种新材质在不动手实验的情况下，也难以确定什么样的参数值才能最符合某种特定需求。因此最佳的实践方法就是给程序增加一个简单的 UI（就像我们在前面示例中所做的那样），并通过一边调节参数一边观察的方法来寻找最满意的参数组合。下面的图 4.16 展示了示例程序 10-mesh-physical-material.html 的渲染效果：

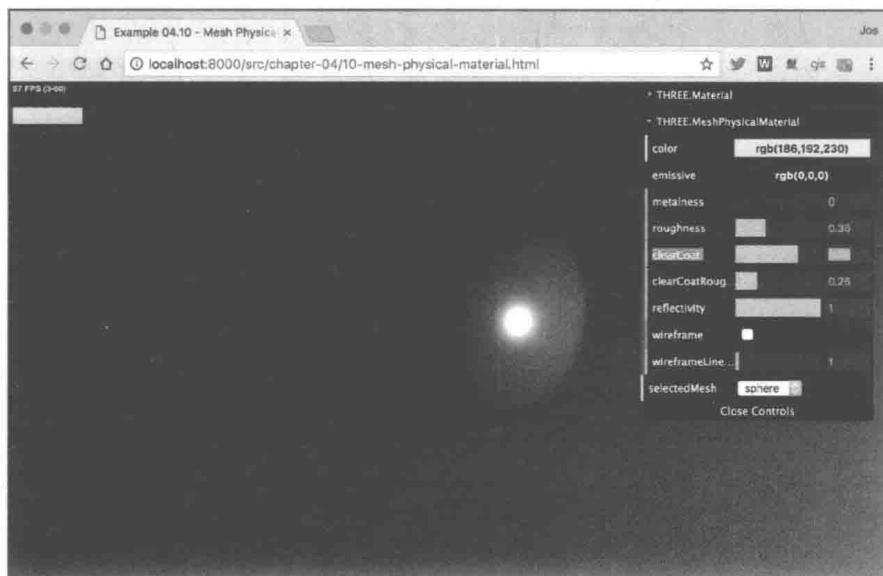


图 4.16

最后一个讲述的高级材质是 THREE.ShaderMaterial。

### 4.3.5 用 THREE.ShaderMaterial 创建自己的着色器

THREE.ShaderMaterial 是 Three.js 库中最通用、最复杂的材质之一。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 网格转换为屏幕上的像素。通过这些自定义的着色器，可以明确地指定对象如何渲染，以及如何覆盖或修改 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例子，展示一下如何设置这种材质。

THREE.ShaderMaterial 有一些我们已经见过的可以设置的属性。使用 THREE.ShaderMaterial，Three.js 传入这些属性的所有信息，但是仍然必须在自己的着色器程序中处理这些信息。表 4.12 列出了我们已经看到过的 THREE.ShaderMaterial 的属性。

表 4.12

名 称	描 述
wireframe	设置这个属性可以将材质渲染成线框。非常适合调试目的
wireframeLineWidth	如果已经打开了 wireframe，这个属性定义线框中线的宽度
lineWidth	该属性定义了要绘制的线的宽度
shading	该属性定义如何着色。可选的值有 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 节找到相应的例子
vertexColors	可以通过这个属性给每个顶点定义不同的颜色。该属性对 CanvasRenderer 不起作用，但是对 WebGLRenderer 起作用。关于该属性的例子，可以参考 4.4.1 节的例子，在这个例子里使用该属性为线段的不同部分设置不同的颜色
fog	该属性指定当前材质是否受全局雾化效果设置的影响。我们在例子中没有用到这个属性。不过要是将该属性设置为 false，那么我们在第 2 章中看到的全局雾化效果设置就不会影响当前对象的渲染

除了这些传入着色器的属性之外，THREE.ShaderMaterial 还有几个特别的属性，可以用来给自己定制的着色器传入更多的信息（它们现在看起来可能不太好理解，更多细节可以参考第 11 章），如表 4.13 所示。

表 4.13

名 称	描 述
fragmentShader	这个着色器定义的是每个传入的像素的颜色。你需要传入像素着色器程序的字符串值
vertexShader	这个着色器允许你修改每一个传入的顶点的位置。你需要传入顶点着色器程序的字符串值
uniforms	通过这个属性可以向你的着色器发信息。同样的信息会发给每一个顶点和片段
defines	转换成 #define 代码片段。这些片段可以用来设置着色器程序里的一些额外的全局变量
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，那么你需要为几何体中的每个顶点提供信息
lights	该属性定义光照数据是否传递给着色器。默认值为 false

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

- `vertexShader`：它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`：它会在几何体的每一个片段上执行。在 `VertexShader` 里，我们会返回这个特定片段应该显示的颜色。

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

本节会给出一个简单的例子，这个例子使用了一个非常简单的 `vertexShader` 程序，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` 程序，使用来自 <http://glslsandbox.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。请注意，着色器不是用 JavaScript 编写的。需要使用类似 C 的 GLSL 语言（`WebGL` 支持 `OpenGL ES` 着色语言 1.0——更多关于 GLSL 的信息，参考 <https://www.khronos.org/webgl/>）来写着色器，代码如下所示：

```
<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，而是关注代码中最重要的部分。要与 JavaScript 中的着色器通信，我们会使用一种称为“统一值”（`uniform`）的方法。在这个例子中，我们使用“`uniform float time;`”语句传入一个外部值。基于此值，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。我们最后一步要做的就是将这个新坐标传回 `Three.js`，如下所示：

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

`gl_Position` 变量是一个特殊变量，用来返回最终的位置。接下来创建一个 `shader Material`，

并传入 vertexShader。出于这个目的，我们创建了一个简单的辅助函数，可以这样使用它：

```
var meshMaterial1 = createMaterial("vertex-shader", "fragment-shader-1");
```

辅助函数代码如下：

```
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。这个变量用来从渲染器向着色器传递信息。这个例子完整的渲染循环如下所示：

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

    cube.rotation.y += 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，time 变量就会增加 0.01。这个信息会传递给 vertexShader，用来计算方块顶点的新位置。现在打开例子 08-shader-material.html，你会看到这个方块沿着它的轴收缩、膨胀，如图 4.17 所示。

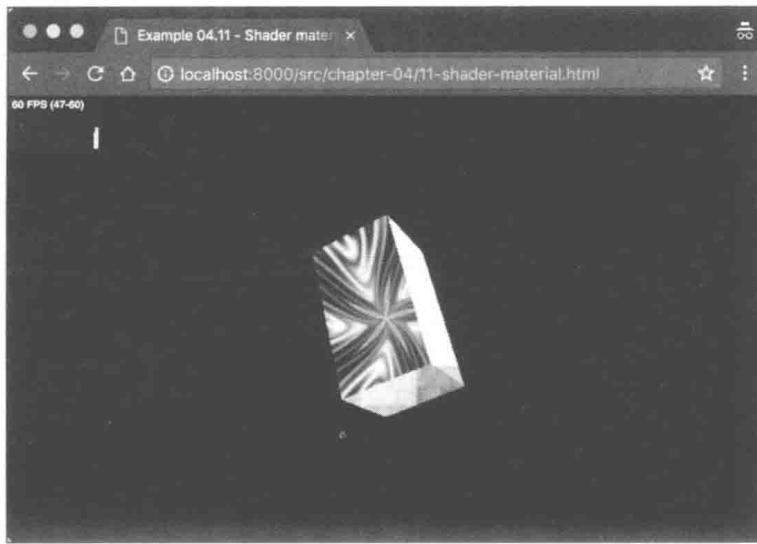


图 4.17

在这个例子里，方块的每个面都在不断变化。正是每个面上的 fragment Shader 造就了这种变化。正如你所猜想的一样，这里使用了 THREE.MeshFaceMaterial（和之前解释过的 createMaterial 函数）：

```
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。在本例中，所有的 fragmentShader 对象都是从 <http://glslsandbox.com/> 复制过来的。这个网站提供了一个试验环境，可以在这里创建和分享 fragmentShader 对象。我不会在这里解释太多，不过这个例子里使用的 fragment-shader-6 类似下面的代码：

```
<script id="fragment-shader-6" type="x-shader/x-fragment">
#ifndef 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 的颜色是通过语句 `gl_FragColor = color_final` 设置的。要获得对 `fragmentShader` 的更多感性认识，可以研究一下 <http://glslsandbox.com/> 上提供的着色器，并在你自己的对象中使用这些代码。在讨论下一种材质之前，这里还有一个使用自定义 `vertexShader` 程序 ([https://www.shadertoy.com /view/Mdx3zr](https://www.shadertoy.com/view/Mdx3zr)) 的例子，如图 4.18 所示。



图 4.18

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 线性几何体的材质

我们将要看的最后几种材质只能用于一个特别的几何体：THREE.Line（线段）。顾名思义，这只是一条线，线段由顶点组成，不包含任何面。Three.js库提供了两种可用于线段的不同材质，如下所示：

- THREE.LineBasicMaterial：用于线段的基础材质，可以设置colors、lineWidth、lineCap和lineJoin属性。
- THREE.LineDashedMaterial：它的属性与THREE.LineBasicMaterial的属性一样，但是可以通过指定虚线和空白间隙的长度来创建出虚线效果。

我们将从基础的变体开始，然后再来看看虚线的变体。

### 4.4.1 THREE.LineBasicMaterial

这种用于THREE.Line几何体的材质非常简单。表4.14列出了这种材质可用的属性。

表 4.14

名称	描述
color	该属性定义线的颜色。如果指定了vertexColors，这个属性就会被忽略
lineWidth	该属性定义线的宽度
lineCap	这个属性定义了线框模式下顶点间线段的端点如何显示。可选的值包括butt（平）、round（圆）和square（方）。默认值为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
lineJoin	这个属性定义了线段的连接点如何显示。可选的值有round（圆）、bevel（斜角）和miter（尖角）。默认值为round。如果你在一个使用低透明度和很大wireframeLineWidth值的例子中靠近观察，就可以看到这个属性的效果。WebGLRenderer对象不支持该属性
vertexColors	将这个属性设置成THREE.VertexColors值，就可以给每个顶点指定一种颜色

在看LineBasicMaterial示例之前，我们先来快速浏览一下如何通过顶点集合创建一个THREE.Line网格，并与LineMaterial材质组合以创建网格，如下面代码所示：

```

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(0xffffffff);
    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);`”作为获取一些`x`、`y`坐标的示例。这个函数返回一个`gosper`曲线（更多信息可以参考[http://en.wikipedia.org/wiki/Gosper\\_curve](http://en.wikipedia.org/wiki/Gosper_curve)），这是一种填充二维空间的简单算法。接下来，我们要创建一个`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`lines`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。要使用 HSL，需提供色调（hue）、饱和度（saturation）和亮度（lightness），而不是提供红、绿、蓝的值。用 HSL 比 RGB 更直观，而且也更容易创建出匹配的颜色集合。你可以在 CSS 规范里找到关于 HSL 的解释，网址是：<http://www.w3.org/TR/2003/CR-css3-color-20030514/#hsl-color>。

既然几何体已经准备好了，我们可以创建`THREE.LineBasicMaterial`，结合这个几何体就可以创建一个`THREE.Line`网格。你可以在示例`12-line-material.html`中看到结果。如图 4.19 所示。

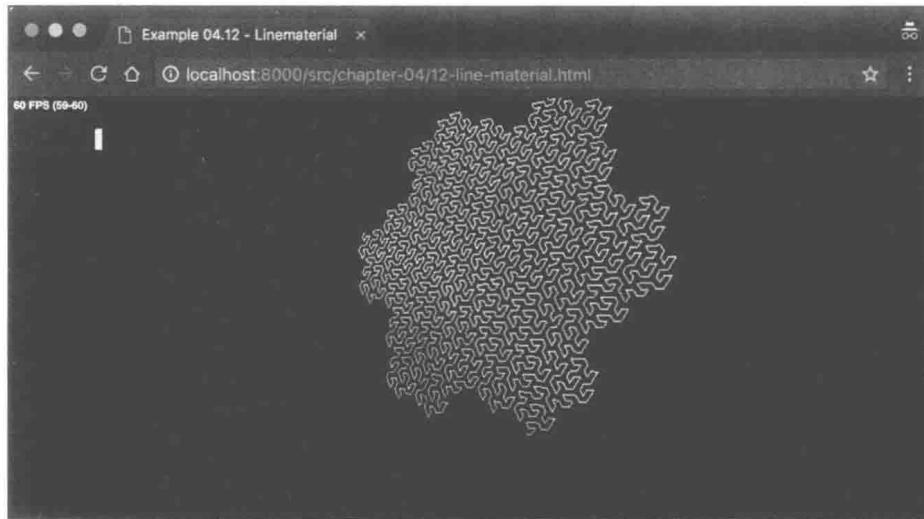


图 4.19

这是一个使用`LineBasicMaterial`及其`vertexColors`属性实现的线段几何体。本章要讨论的下一个也是最后一个材质与`THREE.LineBasicMaterial`稍有不同。通过`THREE.LineDashedMaterial`，不但可以给线段上色，还可以添加一种虚线效果。

#### 4.4.2 THREE.LineDashedMaterial

这种材质有与 THREE.LineBasicMaterial一样的属性，还有几个额外的属性，可以用来定义虚线的宽度和虚线之间的间隙的宽度，如表 4.15 所示。

表 4.15

名 称	描 述
scale	缩放 dashSize 和 gapSize。如果 scale 的值小于 1，dashSize 和 gapSize 就会增大；如果 scale 的值大于 1，dashSize 和 gapSize 就会减小
dashSize	虚线段的长度
gapSize	虚线间隔的宽度

这个材质的用法与 THREE.LineBasicMaterial 基本一样。如下所示：

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

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

唯一的区别是必须调用 computeLineDistance()（用来计算线段顶点之间的距离）。如果不这么做，间隔就不会正确地显示。可以在 13-line-material-dashed.html 中找到关于这种材质的例子，运行效果如图 4.20 所示。

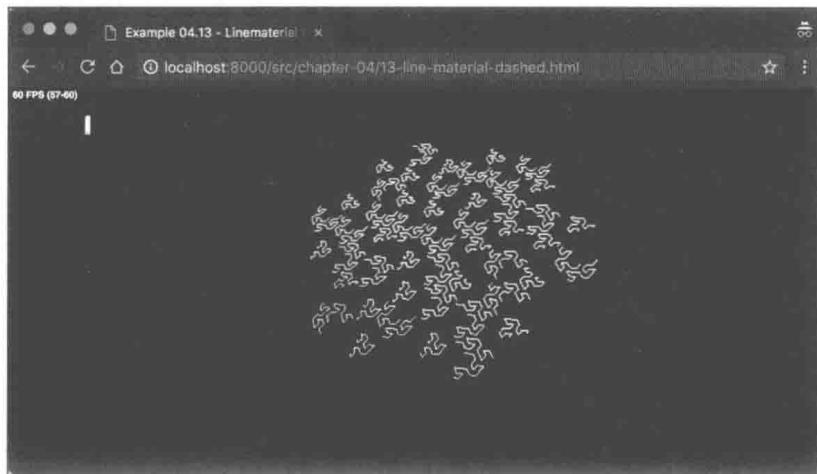


图 4.20

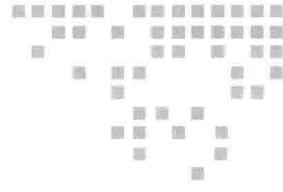
### 4.5 总结

Three.js 提供了很多材质用于给几何体指定皮肤。从简单的 THREE.MeshBasicMaterial 到复杂的 THREE.ShaderMaterial，通过 THREE.ShaderMaterial 可以提供自己的顶点着

色器和像素着色器程序。材质共享很多基础属性。如果你知道如何使用一种材质，可能也知道如何使用其他材质。注意，不是所有的材质都对场景中的光源做出反应。如果希望一个材质计算光照的影响，如果希望一个材质计算光照的影响，应该尽量使用标准材质 THREE.MeshStandardMaterial。而当你需要更多控制时，可以考虑使用 THREE.MeshPhysicalMaterial、THREE.MeshPhongMaterial 或 THREE.MeshLambertMaterial。仅仅从代码确定某种材质属性的效果是非常困难的。通常，使用 dat.GUI 控制面板来试验这些属性是一个不错的方法。

同样该记住的是，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）不能在运行时修改。如果你要修改这些属性的值，需要将 needsUpdate 属性设置为 true。要了解运行时哪些属性可以修改，哪些不属性不能修改，可以参考网址 <https://github.com/mrdoob/three.js/wiki/Updates>。

在本章和前几章中，我们提到过一些几何体，并用在了一些例子中，而且还探究过其中的几个。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

在前面的几章中，我们已经学了 Three.js 库的很多用法，知道了如何创建基础场景、添加光源以及为网格配置材质。在第 2 章，我们接触了 Three.js 库中可用的几何体，可以用这些几何体创建三维物体，但我们并没有真正深入讨论这些几何体的细节。本章和下一章将展示 Three.js 提供的所有几何体（不包括在前一章探讨过的 THREE.Line）。本章涉及的几何体包括：

- THREE.CircleGeometry (圆形)
- THREE.RingGeometry (环形)
- THREE.PlaneGeometry (平面)
- THREE.ShapeGeometry (任意图形)
- THREE.BoxGeometry (长方体)
- THREE.SphereGeometry (球体)
- THREE.CylinderGeometry (圆柱体)
- THREE.ConeGeometry (圆锥体)
- THREE.TorusGeometry (圆环)
- THREE.TorusKnotGeometry (环状扭结)
- THREE.PolyhedronGeometry (多面体)
- THREE.IcosahedronGeometry (二十面体)
- THREE.OctahedronGeometry (八面体)
- THREE.TetraHedronGeometry (四面体)
- THREE.DodecahedronGeometry (十二面体)

在开始学习 Three.js 提供的几何体之前，需要首先深入了解一下 Three.js 中用于代表几何体的两个可用基类：THREE.Geometry 和 THREE.BufferGeometry。前者是旧版本中所有

几何体的基类，而后者则由新版本提供给开发者。新几何体基类 THREE.BufferGeometry 的内部数据组织形式与 GPU 所期待的数据结构保持一致，从而进一步提高了运行效率。但是作为代价，新几何体基类的易用性稍差一些。

旧几何体基类 THREE.Geometry 使用表 5.1 所列的属性来定义物体形状。

表 5.1

属性	描述
vertices (顶点)	该属性是一个顶点数组，存储了用于定义一个几何体的所有顶点的空间位置
faces (面)	该属性是一个 Face3 数组，其中每一个元素定义了一个面。每个面都通过索引指定了 vertices 属性中的三个顶点

可见，THREE.Geometry 非常易于理解和使用。你可以直接向几何体对象添加新顶点并定义新的面，或者通过修改现有顶点来修改现有的面。新的 THREE.BufferGeometry 并没有 vertices 和 faces 属性，而只有 attribute 属性以及可选的 index 属性。（译注：attribute 虽然也常常被译作“属性”，但这里应该当作“分量”来理解，因为一个顶点的数据往往由多个 attribute 分量组成，包括但不限于空间坐标、颜色、法向量、纹理坐标等。）上述两个属性如表 5.2 所示。

表 5.2

属性	描述
attributes (分量)	分量属性所存储的信息将被直接送往 GPU 进行处理。比如，若要定义一个形体，你需要至少创建一个 Float32Array 数组，其中每三个数组元素指定了一个顶点的三维空间坐标，而每三个顶点（即该数组中每九个元素）确定一个面。该数组可以向下面这样创建并添加到分量属性中： <code>geometry.addAttribute('position', new THREE.BufferAttribute(arrayOfVertices, 3));</code>
index (索引)	一般不需要特意指定面，因为默认情况下 position（位置）分量中每三个空间坐标确定一个面。但我们也可以通过 index 属性像 THREE.Geometry 类一样去指定用于组成每一个面的顶点

在本章的示例程序中，你暂时无须关心这两个几何体类的区别。如果想使用基于 THREE.Geometry 类的几何体，可以使用本章开始时罗列的几何体类；而如果想使用这些几何体基于 THREE.BufferGeometry 类的等价实现，则直接在类名中加上“Buffer”即可。例如将 THREE.PlaneGeometry 改为 THREE.PlanerBufferGeometry 即可，其他几何体类与之类似。接下来让我们看看 Three.js 提供的所有基础几何体。

 有时我们需要在基于 THREE.Geometry 的几何体和基于 THREE.BufferGeometry 的几何体之间转换。例如有的模型加载器只能创建基于 THREE.BufferGeometry 的模型对象。但是如果需要在加载模型之后修改模型，则适宜将它转换成为基于 THREE.Geometry 的对象，从而使我们能够直接修改顶点和面，避免去操作庞大的 attribute 数组。幸运的是，THREE.Geometry 提供了 fromBufferGeometry 方法，可

以接收基于 THREE.BufferGeometry 的对象，并将其数据导入到 THREE.Geometry 对象中。对应地，THREE.BufferGeometry 也提供了 fromGeometry 方法，用于实现反向转换。例如，下面代码将基于 THREE.BufferGeometry 的对象转换为基于 THREE.Geometry 的对象：

```
var normalGeometry = new THREE.Geometry();
normalGeometry.fromBufferGeometry(bufferGeometry);
```

下面代码则进行反向转换：

```
var bufferGeometry = new THREE.BufferGeometry();
bufferGeometry.fromGeometry(normalGeometry);
```

## 5.1 Three.js 提供的基础几何体

Three.js 库中，有几种几何体可以用来创建二维网格，还有大量的几何体可以用来创建三维网格。本节先介绍二维几何体：THREE.CircleGeometry、THREE.RingGeometry、THREE.PlanarGeometry 和 THREE.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。下面先介绍二维几何体 THREE.PlanarGeometry。

#### 1. THREE.PlanarGeometry

PlaneGeometry 对象可以用来创建一个非常简单的二维矩形。关于这种几何体的例子，可以查看本章示例 01-basic-2d-geometries-plane.html。通过 PlaneGeometry 创建的矩形如图 5.1 所示。

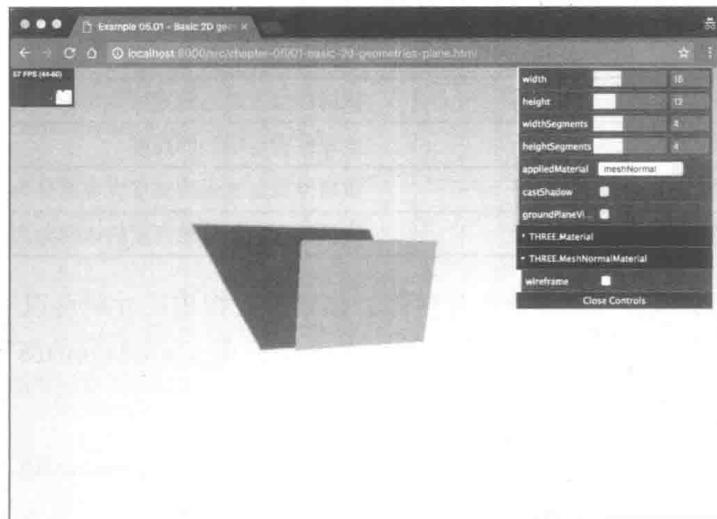


图 5.1

该示例程序有控制界面，既可用于修改几何体的属性（例如高、宽、横向网格宽度和纵向网格高度），也可用于修改材质（及其属性），禁用阴影和隐藏地面等。例如，如果希望观察该几何体单独的面，可以通过隐藏地面并将所选材质设置为线框图模式来实现，如图 5.2 所示。

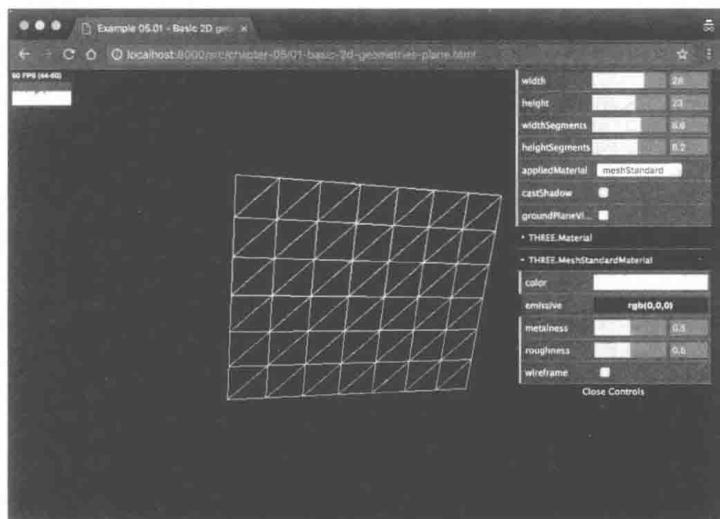


图 5.2

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 THREE.PlaneGeometry 的例子中，可以修改它的属性，并直接查看修改的结果。关于这些属性的解释见表 5.3。

表 5.3

属性	是否必需	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为几段。默认值为 1
heightSegments (高度段数)	否	该属性指定矩形的高度应该划分为几段。默认值为 1

如你所见，这并不是一个复杂的几何体，你只需要指定尺寸就可以了。如果你想创建更多面（比如，一种拥有多个方格的样式），你可以使用 widthSegments 和 heightSegments 参数，将几何体分成多个小面。



如果要在几何体创建后访问其属性，则不能仅仅使用 `plane.width`。要访问几何体的属性，必须使用对象的 `parameters` 属性。因此，要获取我们在本节中创建的平面对象的 `width` 属性，必须使用 `plane.parameters.width`。

## 2. THREE.CircleGeometry

你可能已经猜到 THREE.CircleGeometry 可以创建什么了。通过这个几何体，你可以创建一个非常简单的二维圆（或部分圆）。我们先看看这个几何体的例子 02-basic-2d-geometries-circle.html。如图 5.3 所示，我们创建了一个 THREE.CircleGeometry 几何体，其 thetaLength 属性值小于  $2\pi$ 。

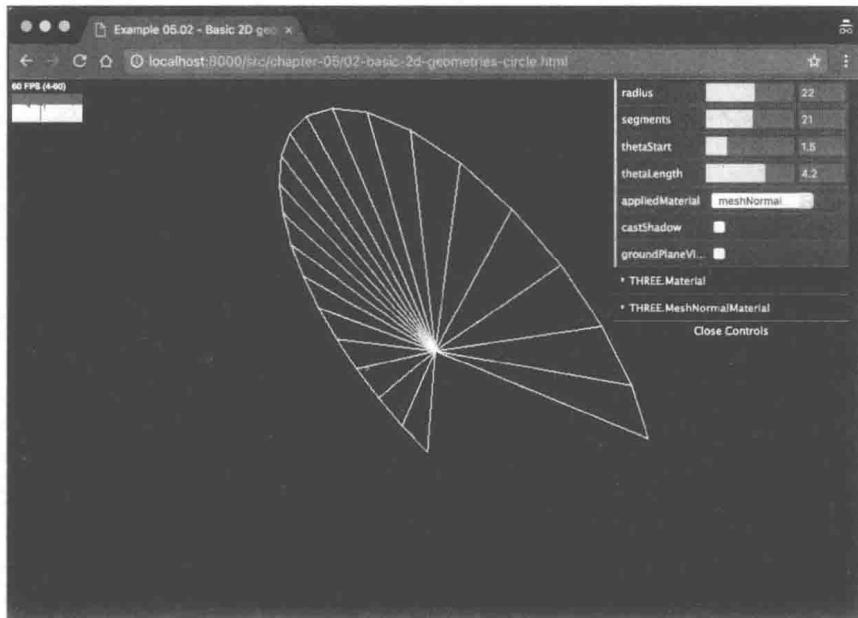


图 5.3



注意， $2\pi$  表示一个以弧度表示的完整圆形。如果要使用的是度数而不是弧度，它们之间的转换非常简单。下面两个函数可以完成弧度和度数之间的转换，如下所示：

```
function deg2rad(degrees) {
    return degrees * Math.PI / 180;
}

function rad2deg(radians) {
    return radians * 180 / Math.PI;
}
```

在这个示例里，你可以观察和控制用 THREE.CircleGeometry 创建的网格。当你创建 THREE.CircleGeometry 对象时，可以指定几个属性来定义圆的样子，如表 5.4 所示。

表 5.4

属性	是否必需	描述
radius	否	圆的半径决定了圆的大小。半径是指从圆心到圆弧的距离。默认值为 50

(续)

属性	是否必需	描述
segments	否	该属性定义了创建圆所用面的数量。最小值为 3 个，如果没有特别说明，默认值为 8。值越大，创建出的圆越光滑
thetaStart	否	该属性定义从哪里开始画圆。这个值的范围是 0 到 $2\pi$ ，默认值为 0
thetaLength	否	该属性定义圆要画多大。如果没有指定，默认值为 $2\pi$ （整圆）。例如，如果这个值指定的是 $0.5\pi$ ，那么得到的是四分之一圆。该属性和 thetaStart 属性结合使用即可定义圆的形状

你可以用如下代码片段创建一个完整的圆：

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

如果想使用这个几何体创建一个半圆，你可以这样做：

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

上面代码创建了一个半圆，其半径为 3，由 12 个面组成，起始角度为 0，终止角度为 Math.PI（译注：即  $\pi$ ）。

在进入下一个几何体之前，我们先对 Three.js 创建二维图形（THREE.PlaneGeometry、THREE.CircleGeometry 和 THREE.ShapeGeometry）时所用的方向做简单说明：Three.js 创建的这些对象都是“直立”的，所以它们只位于  $x$ - $y$  平面。这很符合逻辑，因为它们是二维图形。但是一般来讲，特别是 THREE.PlaneGeometry，你会希望这个网格“躺”下来，以便构建一种地面 ( $x$ - $z$  平面)，好把其他对象放在上面。创造一个水平放置而不是竖直的二维对象，最简单的方法是将网格沿  $x$  轴向后旋转四分之一圈 ( $-\pi/2$ )。代码如下所示：

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

这就是 THREE.CircleGeometry 的全部内容。接下来要讨论的几何体是 THREE.RingGeometry，它看起来和 THREE.CircleGeometry 很像。

### 3. THREE.RingGeometry

使用 THREE.RingGeometry 可以创建一个如图 5.4 所示的二维对象，这个对象不仅非常类似于 THREE.CircleGeometry，而且可以在中心定义一个孔（参考 03-basic-3d-geometries-ring.html）。

THREE.RingGeometry 没有任何必需的属性（参阅表 5.5 中的默认值），因此，要创建此几何体，只需要指定以下内容：

```
Var ring = new THREE.RingGeometry();
```

你可以通过将表 5.5 所示参数传递到构造函数中来进一步自定义环形几何体的外观。

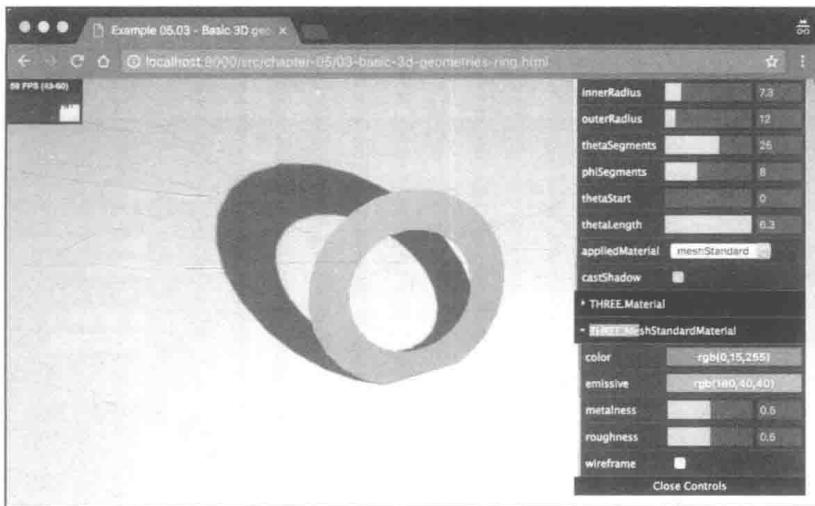


图 5.4

表 5.5

属性	是否必需	描述
innerRadius	否	圆环的内半径，它定义中心圆孔的尺寸。如果将该属性设为 0，则将不显示圆孔。默认值为 0
outerRadius	否	圆环的外半径，它定义圆环的尺寸。这个半径是指从圆心到圆弧的距离。默认值为 50
thetaSegments	否	该属性用来定义创建圆环的对角线段的数量。较高的值意味着更平滑的圆环。默认值为 8
phiSegments	否	该属性定义沿着圆环的长度所需要使用的线段的数量。默认值为 8。这并不会真的影响圆的平滑度，但是会增加面的数量
thetaStart	否	该属性定义从哪里开始画圆。这个值的范围是 0~2*PI，默认值为 0
thetaLength	否	该属性定义圆要画多大。如果没有指定，默认值为 2*PI（整圆）。例如，如果这个值指定的是 0.5*PI，那么得到的是四分之一圆。结合该属性和 thetaStart 属性即可定义圆的形状

图 5.5 所示的场景中有一个用线框图模式绘制的圆环几何体。在图中可以明确的观察到 thetaSegments 和 phiSegments 两个属性对渲染产生的影响。

下面我们会介绍最后一个二维图形：THREE.ShapeGeometry。

#### 4. THREE.ShapeGeometry

THREE.PlanGeometry 和 THREE.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，可以使用 THREE.ShapeGeometry。通过 THREE.ShapeGeometry，你可以调用几个函数来创建自己的图形。你可以将该功能与 HTML 画布元素和 SVG 里的 <path> 元素的功能相比较。我们先从一个示例开始，然后，我们会向你

展示如何使用这些函数绘制你自己的图形。可以在本章的源代码中找到这个示例 04-basic-2d-geometries-shape.html。本例效果如图 5.6 所示。

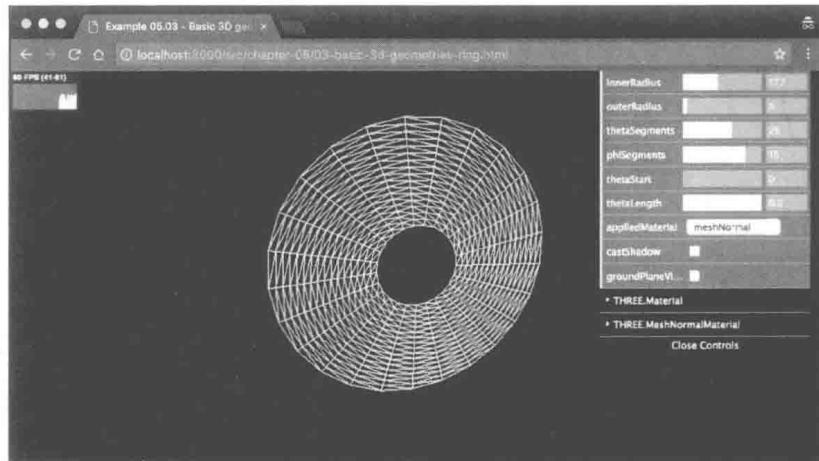


图 5.5

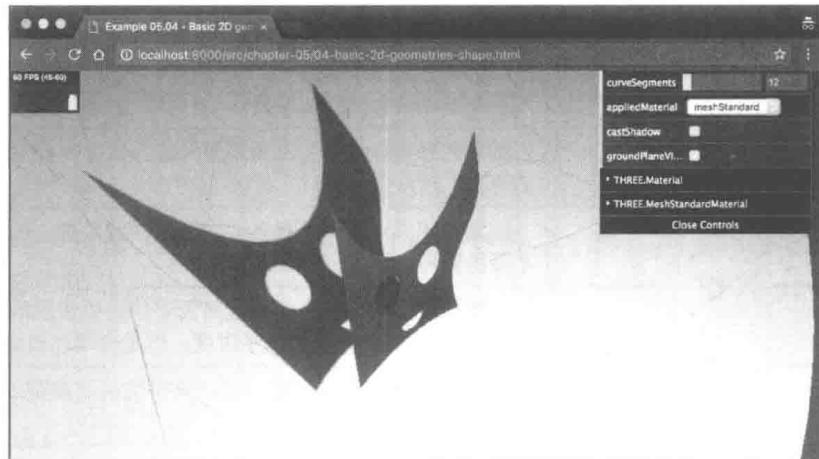


图 5.6

在这个例子里，你可以看到一个自定义的二维图形。在开始描述该几何体的属性前，让我们首先来看看创建该图形的代码。在创建 THREE.ShapeGeometry 对象之前，必须先创建 THREE.Shape 对象。你可以通过查看前一个示例来了解这些步骤。创建 THREE.Shape 对象如下所示：

```
function drawShape() {
    // create a basic shape
    var shape = new THREE.Shape();

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

    // straight line upwards
    shape.lineTo(10, 100);
```

```

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，你可以看到我们使用线条（line）、曲线（curve）和样条曲线（spline）创建了此图形的轮廓。然后我们使用 THREE.Shape 对象的 holes 属性给这个图形打了几个孔。但是，本节讨论的是 THREE.ShapeGeometry 而不是 THREE.Shape。要从 THREE.Shape 创建出一个几何体，需要把 THREE.Shape（在我们的例子中从 drawShape() 函数返回）作为参数传递给 THREE.ShapeGeometry，如下所示：

```
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。当你已经有一个图形时，还有一种方式创建 THREE.ShapeGeometry 对象。可以调用 shape.makeGeometry(options)，这个函数会返回一个 THREE.ShapeGeometry 的实例（有关选项的说明请参见表 5.6）。

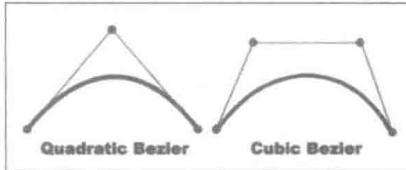
我们首先来看看可以传给 THREE.ShapeGeometry 的参数，如表 5.6 所示。

THREE.ShapeGeometry 中最重要的部分是 THREE.Shape，它可以用来创建图形。所以下面介绍用来创建 THREE.Shape 的绘图函数。请注意，这些函数实际上是 THREE.Shape 从 THREE.Path 对象继承而来的，具体见表 5.7。

表 5.6

属性	是否必需	描述
shapes	是	用来创建 THREE.Geometry 的一个或多个 THREE.Shape 对象。可以传入单个 THREE.Shape 对象或是一个 THREE.Shape 对象数组
options	否	<p>还可以传入一些选项，这些选项可以应用于使用 shapes 参数传入的所有图形。关于这些选项的解释如下：</p> <ul style="list-style-type: none"> <li>• curveSegments——此属性确定从图形创建的曲线的平滑程度。默认值为 12</li> <li>• material——这是用于为指定图形创建的面的 materialIndex 属性。当把 THREE.MeshFaceMaterial 和此几何体一起使用时，materialIndex 属性决定传入的材质中的哪些材质用于传入的图形的面</li> <li>• UVGenerator——当对材质使用纹理时，UV 映射决定纹理的哪个部分用于特定的面。使用 UVGenerator 属性，可以传入自己的对象，这将为传入的图形创建的面创建 UV 设置。有关 UV 设置的更多信息请参见第 10 章。如果没有指定，THREE.ExtrudeGeometry.WorldUVGenerator 会被使用</li> </ul>

表 5.7

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 函数设定的位置）绘制一条线到指定的 x 和 y 坐标处
quadraticCurveTo(aCPx, aCPy, x, y)	可以使用两种不同的方式来指定曲线：使用 quadraticCurveTo 函数，或使用 bezierCurveTo 函数。两个函数的区别在于指定曲线曲率的方法。下图展示了这两个选项之间的区别：
	
bezierCurveTo(aCPx1, aCPy1, aCPx2, aCPy2, x, y)	对于二次曲线，我们要额外指定一个点（使用 aCPx 和 aCPy 参数），曲线仅基于该点绘制，当然还需要指定端点（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位置
splineThru(pts)	根据提供的参数绘制一条曲线。相关说明可以参考前一行的内容。该曲线的绘制基于两个定义曲线的坐标（aCPx1 和 aCPy1, aCPx2 和 aCPy2）以及终点坐标（x 和 y）。起始点是路径的当前位置
arc(aX, aY, aRadius, aStartAngle, aEndAngle, aClockwise)	该函数用来画圆（或一段圆弧）。圆弧起始于路径的当前位置。aX 和 aY 用来指定与当前位置的偏移量。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

(续)

名 称	描 述
absArc(aX, aY, aRadius, aStartAngle, aEndAngle, AClockwise)	参考 arc 函数的描述。其位置是绝对位置，而不是相对于当前位置
ellipse(aX, aY, xRadius, yRadius, aStartAngle, aEndAngle, aClockwise)	参考 arc 函数的描述。作为补充，通过 ellipse 函数，可以分别指定 x 轴半径和 y 轴半径
absEllipse(aX, aY, xRadius, yRadius, aStartAngle, aEndAngle, aClockwise)	参考 ellipse 函数的描述。其位置是绝对位置，而不是相对于当前位置
fromPoints(vectors)	如果给该函数传入一个 THREE.Vector2 (或 THREE.Vector3) 对象数组，THREE.js 会创建一条通过提供的顶点使用直线绘制的路径
holes	holes 属性包含一个 THREE.Shape 对象数组。这个数组中的每一个对象会渲染为一个孔。关于这个属性的一个很好的例子就是我们在本节开头看到的例子。在那段代码片段中，我们添加了三个 THREE.Shape 对象到这个数组，一个用来渲染左边的孔，一个用来渲染右边的孔，还有一个渲染主要的 THREE.Shape 对象——嘴

要想理解不同的属性对几何体的形状所产生的影响，最好的方法就是启用线框图模式，然后一边修改属性值一边观察画面变化。图 5.7 展示了当 curveSegments 属性值很低时，几何体会发生什么变化。

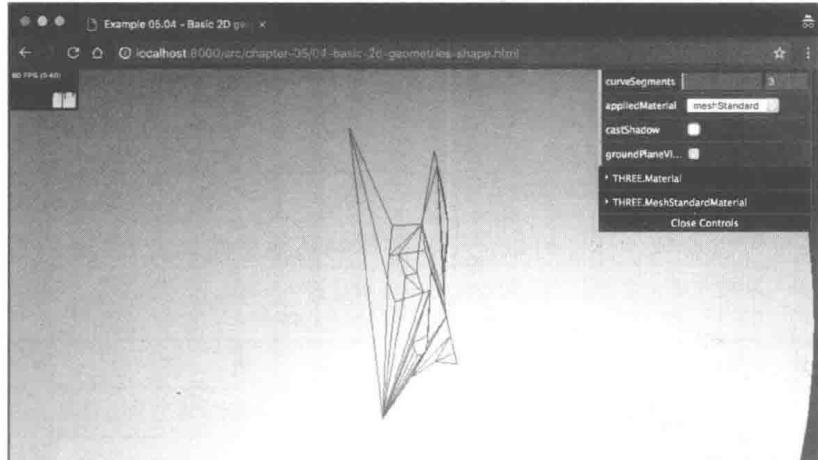


图 5.7

从上图中的示例可以看出，此时几何体的形状失去了原本平滑、优美的边缘，但同时也减少所需的三角形数量。（译注：这样可以在较低配置的硬件系统上提高程序的运行效率。）以上就是关于所有 2D 形状的介绍，接下来将继续学习 Three.js 提供的 3D 形状。

### 5.1.2 三维几何体

本节讨论基础三维几何体，我们从一个已经看过多次的几何体 THREE.BoxGeometry 开始。

## 1. THREE.BoxGeometry

THREE.BoxGeometry 是一种非常简单的三维几何体，只需要指定宽度、高度和深度就可以创建一个长方体。我们已经添加了一个示例 05-basic-3d-geometries-cube.html，你可以在这个示例上试验这些属性。图 5.8 展示了这个几何体。

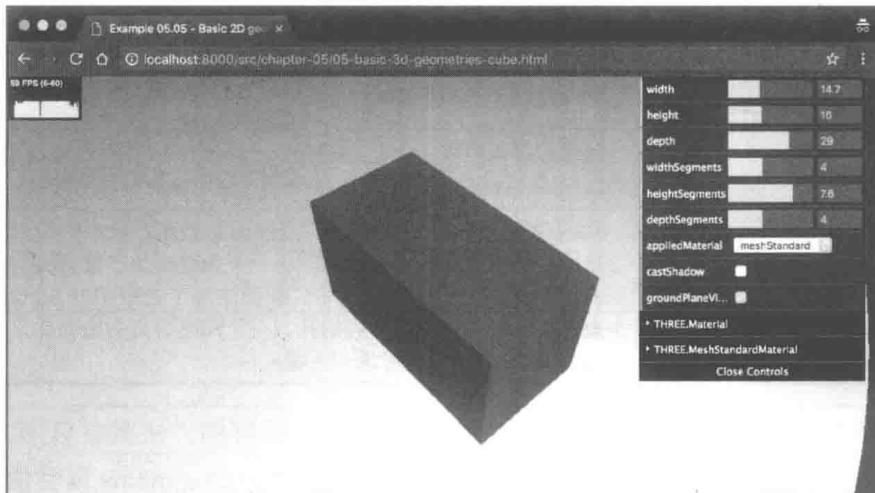


图 5.8

正如你在图中看到的，通过改变 THREE.BoxGeometry 对象的 width、height 和 depth 属性，可以控制网格的尺寸。这三个属性也是创建一个长方体时必须要提供的，如下所示：

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

在这个例子里可以看到，你还可以定义长方体的其他属性。表 5.8 是所有属性的说明。

表 5.8

属性	是否必需	描述
width	是	该属性定义长方体的宽度。所谓宽度是长方体沿 x 轴方向的长度
height	是	该属性定义长方体高度。所谓高度是长方体沿 y 轴方向的长度
depth	是	该属性定义长方体的深度。所谓深度是长方体沿 z 轴方向的长度
widthSegments	否	该属性定义的是沿 x 轴方向将面分成多少份。默认值为 1
heightSegments	否	该属性定义的是沿 y 轴方向将面分成多少份。默认值为 1
depthSegments	否	该属性定义的是沿 z 轴方向将面分成多少份。默认值为 1

通过增加多个分段 (segment) 属性，你可以将长方体的 6 个大面分成很多小面。这在你用 THREE.MeshFaceMaterial 为长方体的不同部分设置特定属性时比较有用。THREE.BoxGeometry 是一个非常简单的几何体。另一个简单几何体是 THREE.SphereGeometry。

## 2. THREE.SphereGeometry

通过 SphereGeometry，你可以创建一个三维球体。我们直接看例子 06-basic-3d-geometries-sphere.html，如图 5.9 所示。

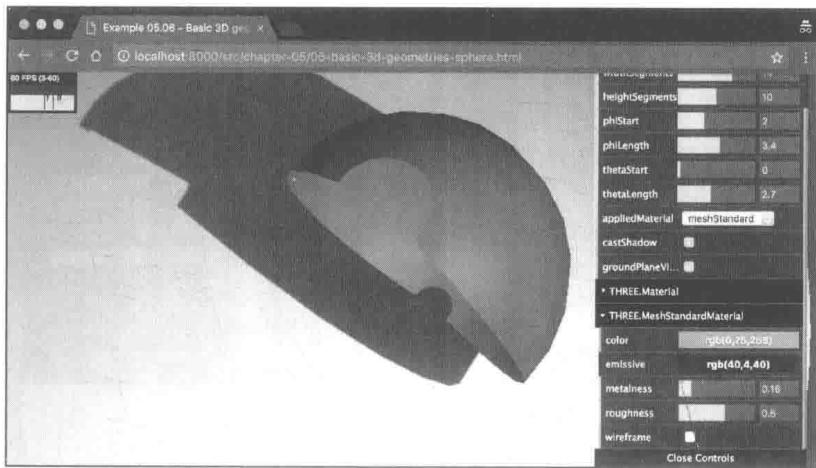


图 5.9

图 5.9 中展示的是半个打开的球，它是基于 THREE.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 THREE.SphereGeometry 可以简单地通过 new THREE.SphereGeometry() 来创建。表 5.9 中的属性可以用来调节结果网格的外观。

表 5.9

属性	是否必需	描述
radius	否	该属性设置球体的半径。它决定最终网格有多大。默认值为 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。默认值为 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。默认值为 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球体。取值范围是 0 到 $2\pi$ 。默认值为 0
phiLength	否	该属性用来指定球体从 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。默认值为 $2\pi$
thetaStart	否	该属性用来指定从 y 轴的什么地方开始绘制球体。取值范围是 0 到 $\pi$ 。默认值为 0
thetaLength	否	该属性用来指定球体从 thetaStart 开始画多少。 $\pi$ 是整球， $0.5\pi$ 只会绘制上半球。默认值为 $\pi$

radius、widthSegments 和 heightSegments 属性应该很清楚了，我们已经在别的例子里

看过这些属性了。如果没有例子，phiStart、phiLength、thetaStart 和 thetaLength 属性会有点难懂。幸运的是，你可以通过示例 06-basic-3d-geometries-sphere.html 的菜单来试验这些属性，并且创建一些比较有趣的几何体，如图 5.10 所示。

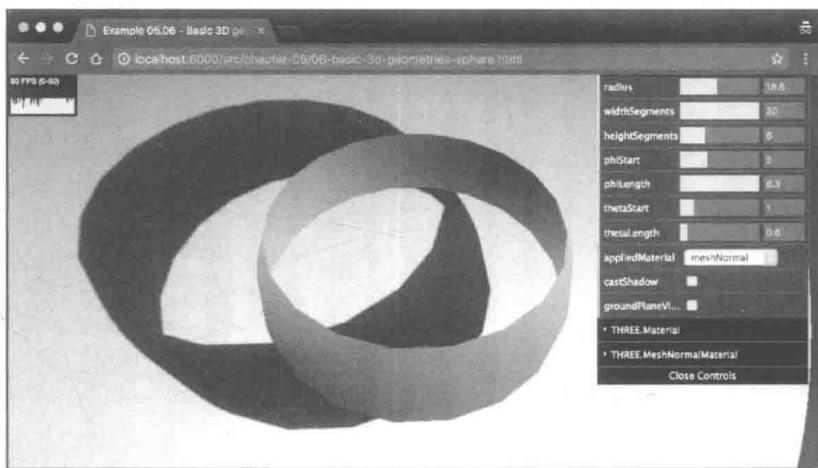


图 5.10

下一个几何体是 THREE.CylinderGeometry。

### 3. THREE.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（07-basic-3d-geometries-cylinder.html），你可以在上面试验这个几何体的属性。如图 5.11 所示。

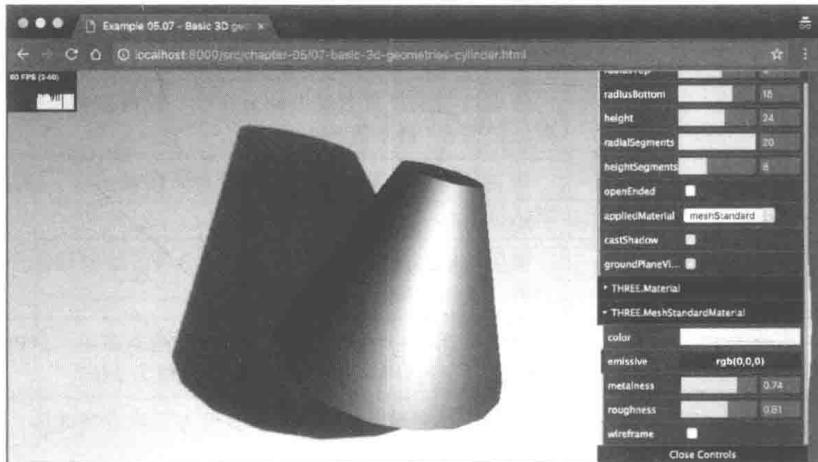


图 5.11

创建 THREE.CylinderGeometry 时，没有必要提供参数。所以你只要调用 new THREE.CylinderGeometry() 即可创建一个圆柱。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。这些属性列在表 5.10 中。

表 5.10

属性	是否必需	描述
radiusTop	否	该属性设置圆柱顶部的尺寸。默认值为 20
radiusBottom	否	该属性设置圆柱底部的尺寸。默认值为 20
height	否	该属性设置圆柱的高度。默认值为 100
radialSegments	否	该属性设置沿圆柱的半径分成多少段。默认值为 8。分段数越多，意味着圆柱越光滑
heightSegments	否	该属性设置沿圆柱的高度分成多少段。默认值为 1。分段数越多，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。默认为 false
thetaStart	否	该属性决定了在 x 轴上开始绘制圆柱的位置。取值范围为 0 到 $2\pi$ ，默认值为 0
thetaLength	否	该属性决定了有多少圆柱面被绘制。取值为 $2\pi$ 时将绘制完整圆柱面，取值为 $\pi$ 时可绘制半个圆柱面。默认值为 $2\pi$

这些都是配置圆柱的非常基础的属性。但是有趣的是，你可以在顶部（或底部）使用负数的半径。如果这么设置，你就可以使用这个几何体创建出一个类似沙漏的图形，如图 5.12 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。

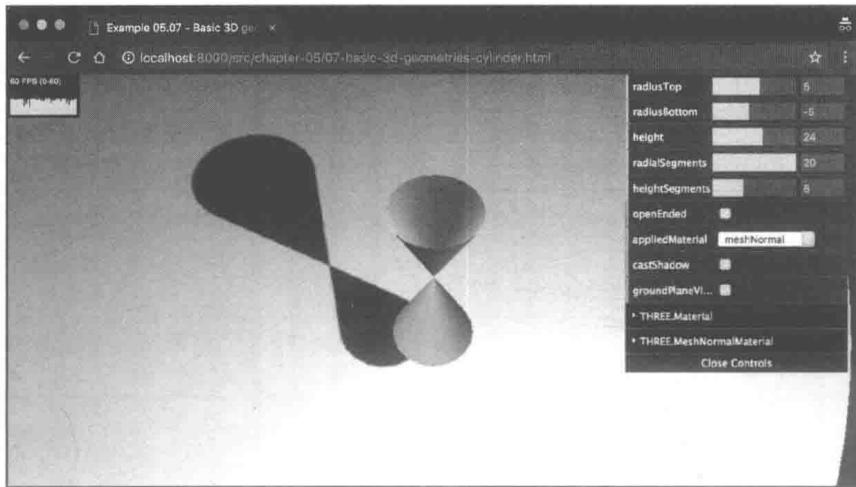


图 5.12

下一个几何体是 THREE.ConeGeometry（圆锥体）。它与 THREE.CylinderGeometry 的非常相似，唯一区别就是其顶部半径默认为 0。

#### 4. THREE.ConeGeometry

THREE.ConeGeometry 为圆锥几何体，它几乎拥有 THREE.CylinderGeometry 的所有属性，唯一的区别是它不支持独立的 radiusTop (顶半径) 和 radiusBottom (底半径)，而是只有一个作为底半径的 radius 属性。其绘制效果如图 5.13 所示。

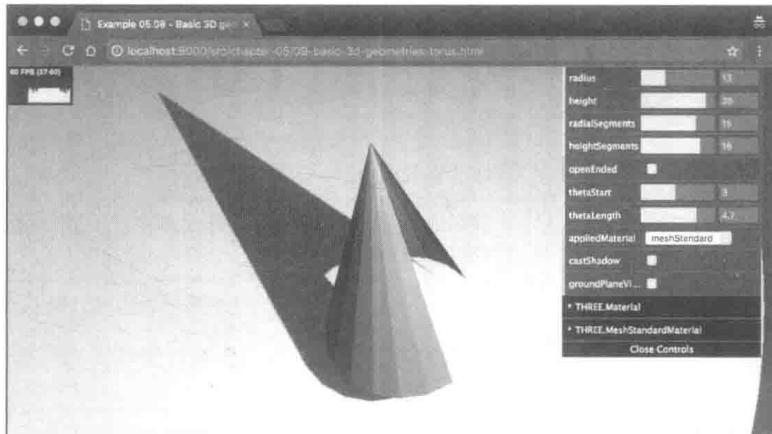


图 5.13

圆锥几何体支持表 5.11 中列的属性。

表 5.11

属性	是否必需	描述
radius	否	该属性设置圆锥底部的尺寸。默认值为 20
height	否	该属性设置圆锥的高度。默认值为 100
radialSegments	否	该属性设置沿圆锥底部半径分成多少段。默认值为 8。分段数越多，意味着圆锥越光滑
heightSegments	否	该属性设置沿圆锥的高度分成多少段。默认值为 1。分段数越多，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。默认为 false
thetaStart	否	该属性决定了在 x 轴上开始绘制圆柱的位置。取值范围为 0 到 2*PI，默认值为 0
thetaLength	否	该属性决定了有多少圆柱面被绘制。取值为 2*PI 时将绘制完整圆柱面，取值为 PI 时可绘制半个圆柱面。默认值为 2*PI

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

#### 5. THREE.TorusGeometry

Torus (圆环) 是一种看起来像甜甜圈的简单图形。图 5.14 展示的是一个真实的 THREE.TorusGeometry，打开示例 08-basic-3d-geometries-torus.html 就可以看到它。

跟大多数简单几何体一样，创建 THREE.TorusGeometry 时没有必须提供的参数。表 5.12 列出的是创建这个几何体时可以指定的参数。

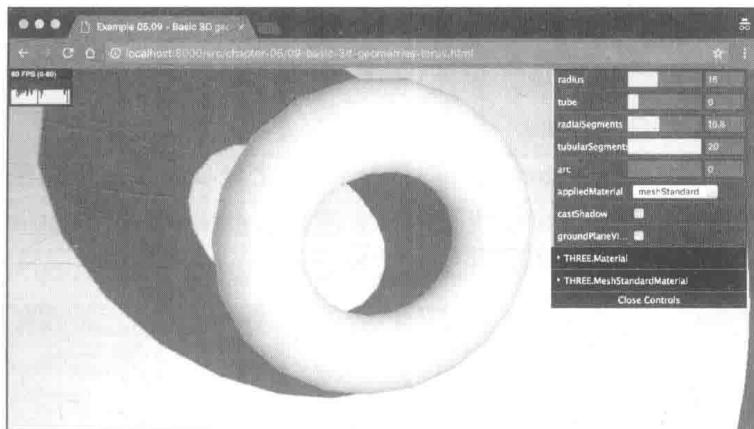


图 5.14

表 5.12

属性	是否必需	描述
radius	否	这个参数设置的是完整圆环的尺寸。默认值为 100
tube	否	这个参数设置的是管（实际圆环）的半径。默认值为 40
radiusSegments	否	这个参数设置的是沿圆环长度方向分成的段数。默认值为 8。可以在示例中看看修改此值的效果
tubularSegments	否	这个参数设置的是沿圆环宽度方向分成的段数。默认值为 6。可以在示例中看看修改此值的效果
arc	否	通过这个属性，可以控制是否绘制一个完整的圆环。默认值为 $2*\text{PI}$ （完整圆环）

大多数都是你已经见过的基础属性。但是其中的 arc 属性是一个非常有趣的属性。通过这个属性，你可以指定是绘制一个完整的圆环还是部分圆环。通过试验这个属性，你可以创建一些非常有趣的网格，如图 5.15 所示，它的 arc 属性值是  $0.5*\text{PI}$ 。

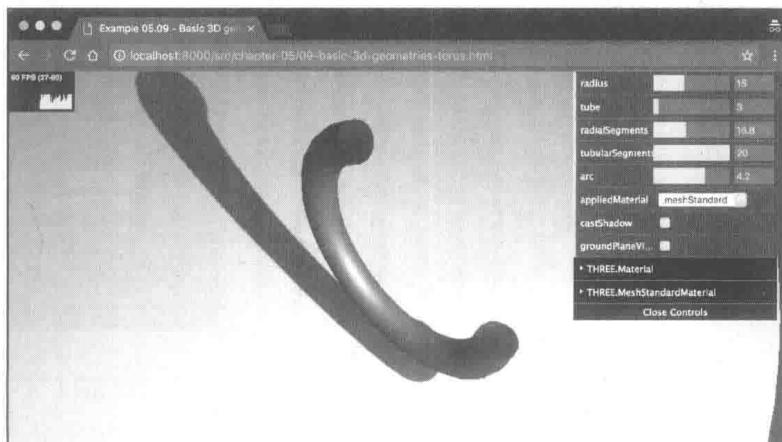


图 5.15

THREE.TorusGeometry 是一个非常简洁明了的几何体。下面我们将要看的几何体是 THREE.TorusKnotGeometry，虽然名字跟 TorusGeometry 差不多，但是没有那么简单。

## 6. THREE.TorusKnotGeometry

通过 THREE.TorusKnotGeometry，你可以创建一个环状扭结。环状扭结是一种比较特别的结，看起来就像一根管子绕自己转了几圈。例子 09-basic-3d-geometries-torus-knot.html 很好地解释了这一点。这个几何体如图 5.16 所示。

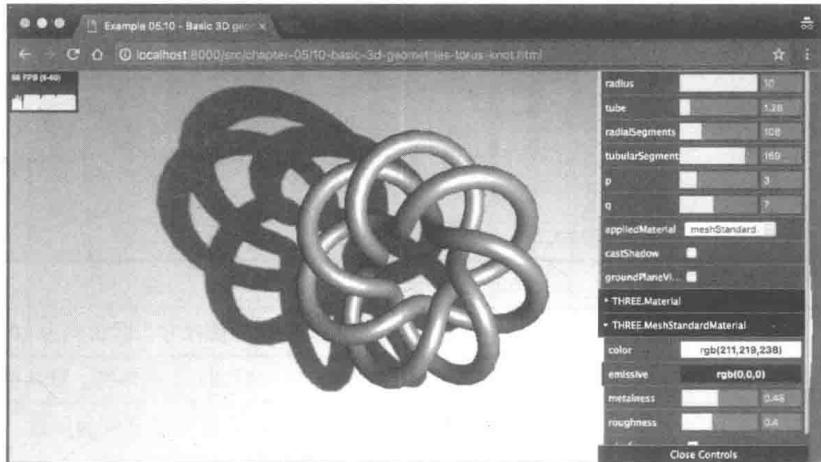


图 5.16

如果你打开这个例子，修改属性 p 和 q，就可以创建各种各样漂亮的几何体。p 属性定义扭结绕其轴线旋转的频率，q 属性定义扭结绕其内部缠绕多少次。

如果这听起来有点模糊，不要担心。你不必理解这些属性也能创建出如图 5.17 所示的漂亮扭结。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](http://en.wikipedia.org/wiki/Torus_knot)。

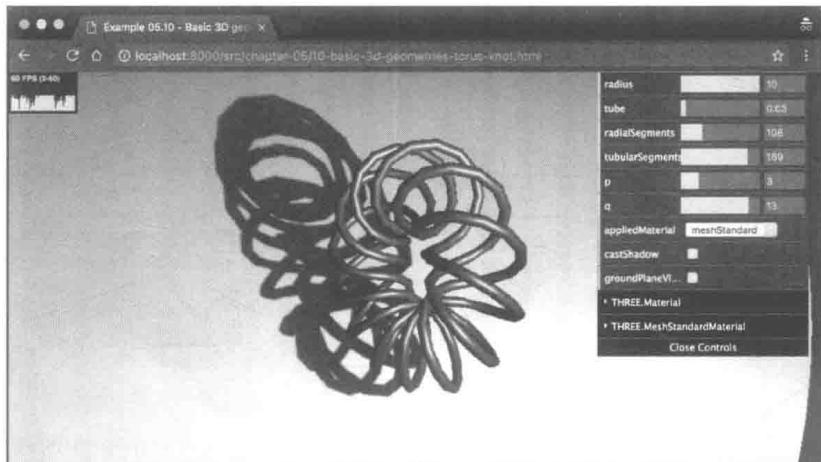


图 5.17

在这个几何体的例子里，你可以试验表 5.13 所列的属性，看看不同的 p 和 q 组合对几何体的影响效果。

表 5.13

属性	是否必需	描述
radius	否	这个参数设置的是完整圆环的尺寸。默认值为 100
tube	否	这个参数设置的是管（实际圆环）的半径。此属性的默认值为 40
radiusSegments	否	这个参数设置的是沿环状扭结长度方向分成的段数。默认值为 64。可以在示例中看看修改此值的效果
tubularSegments	否	这个参数设置的是沿环状扭结宽度方向分成的段数。默认值为 8。可以在示例中看看修改此值的效果
p	否	该属性定义扭结的形状。默认值为 2
q	否	该属性定义扭结的形状。默认值为 3
heightScale	否	通过这个属性，可以拉伸这个环状扭结。默认值为 1

下一个几何体也是最后一个基础几何体：THREE.PolyhedronGeometry。

## 7. THREE.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，而不必指定 THREE.PolyhedronGeometry 的顶点和面。我们将在本节稍后讨论这些多面体。如果你想直接使用 THREE.PolyhedronGeometry，必须指定各个顶点和面（就像我们在 3 章使用方块时那样）。例如，我们要创建一个简单的四面体（参见 THREE.TetrahedronGeometry），如下所示：

```

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

var indices = [
  2, 1, 0,
  0, 3, 2,
  1, 3, 0,
  2, 3, 1
];

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

```

为了构建一个 THREE.PolyhedronGeometry 对象，我们需要传入 vertices、indices、radius 和 detail 属性。这个 THREE.PolyhedronGeometry 对象的结果可以在示例 10-basic-3d-geometries-polyhedron.html 中看到（在右上角的菜单 type 中选择 Custom），如图 5.18 所示。

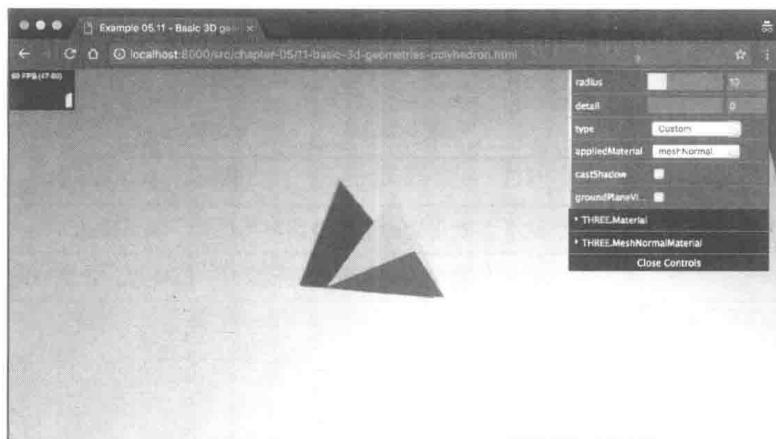


图 5.18

当你创建多面体时，可以传入表 5.14 中所列的四个属性。

表 5.14

属性	是否必需	描述
vertices	是	该属性设置构成多面体的顶点
indices	是	该属性设置由 vertices 创建出的面
radius	否	该属性指定多面体的大小。默认值为 1
detail	否	通过这个属性，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 4 个小三角形中的每一个将会继续分成 4 个小三角形，以此类推

图 5.19 展示了具有更高 detail 属性值的多面体。

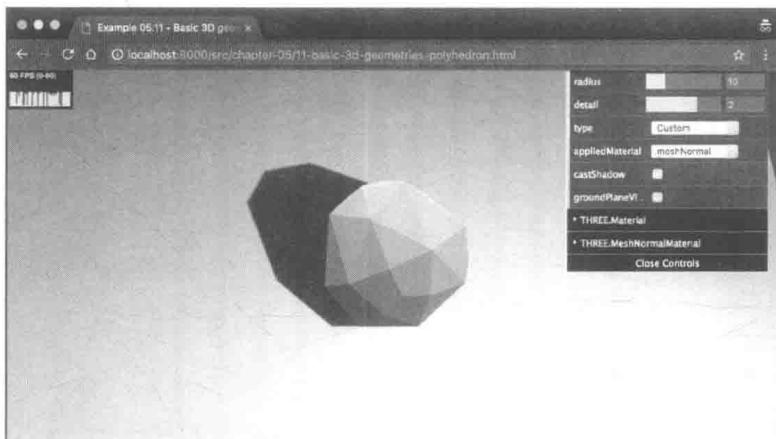


图 5.19

在本节的开始，我们提到过 Three.js 库提供了几个开箱即用的多面体。接下来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。

### 8. THREE.IcosahedronGeometry

THREE.IcosahedronGeometry 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 的值。图 5.20 展示了使用 THREE.IcosahedronGeometry 创建的一个多面体。

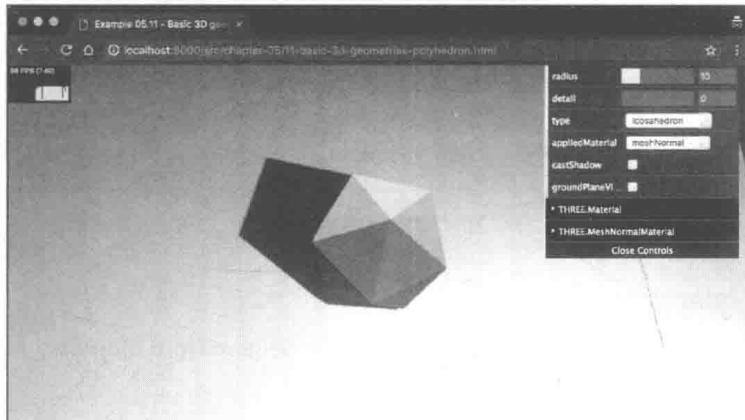


图 5.20

### 9. THREE.TetrahedronGeometry

Tetrahedron（正四面体）是最简单的多面体。这个多面体只包含由 4 个顶点创建的 4 个三角形面。创建 THREE.TetrahedronGeometry 跟创建 Three.js 提供的其他多面体一样，只要指定 radius 和 detail 的值。图 5.21 展示了使用 THREE.TetrahedronGeometry 创建的一个多面体。

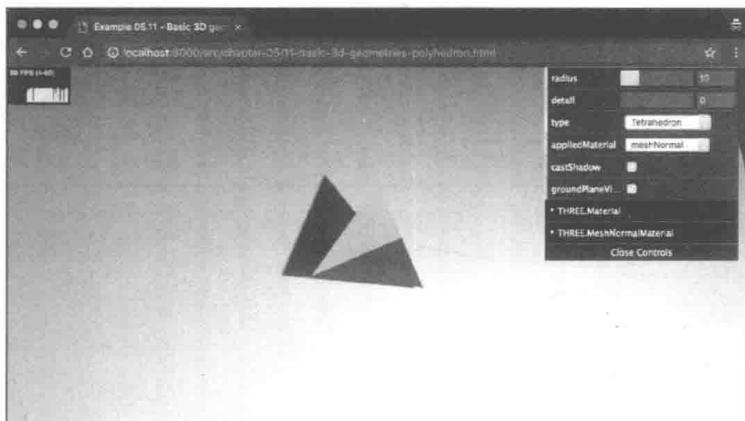


图 5.21

### 10. THREE.OctahedronGeometry

Three.js 库还提供了实现八面体的方法。顾名思义，这个多面体有 8 个面。这些面是从 6 个顶点创建出来的。图 5.22 展示的就是这样一个多面体。

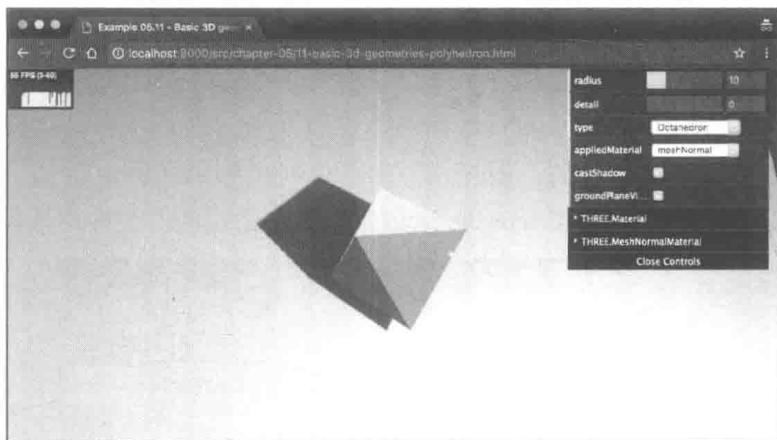


图 5.22

### 11. THREE.DodecahedronGeometry

Three.js 库提供的最后一个多面体是 THREE.DodecahedronGeometry。这个多面体有 12 个面，如图 5.23 所示。

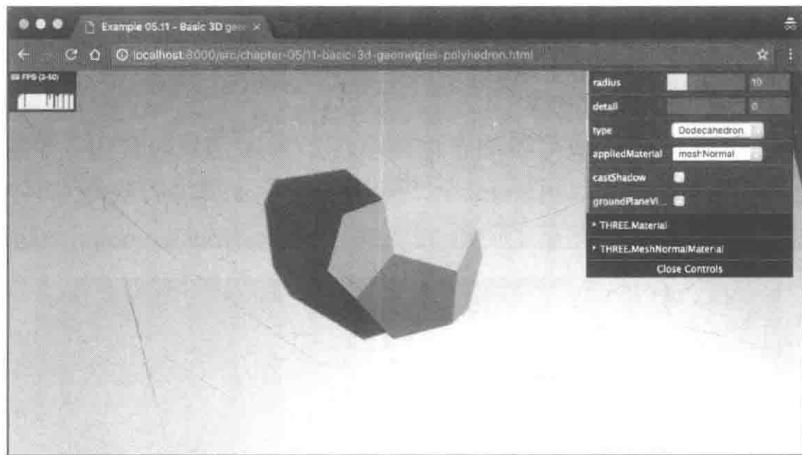


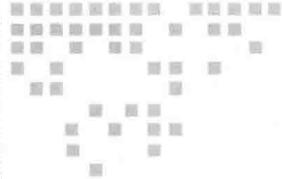
图 5.23

## 5.2 总结

本章我们讨论了 Three.js 库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。学习如何使用这些几何体的最好方式就是去试验它们。使用本章的示例来了解那些 Three.js 库提供的标准几何体的属性。一种好的方法是：开始创建几何体时，可以选择一种简单的材质，不要直接使用那些复杂的材质，可以从简单的 THREE.MeshBasicMaterial 材质开始，并将 wireframe 属性设为 true，或者使用 THREE.MeshNormal-Material 材质也

可以。这样，你就可以对几何体的真实形状有一个更好的了解。对于二维图形，重要的是要记住它们是放置在  $x$ - $y$  平面上的。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  $-0.5 * PI$ 。最后，如果你要旋转一个二维图形，或者一个开放的三维图形（例如圆柱或管），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。

本章内容聚焦在那些简单、易懂的网格。Three.js 也提供了一些方法来创建复杂的几何体。下一章你将学习如何创建它们。



# 高级几何体和二元操作

在上一章我们向你展示了 Three.js 库提供的所有基础几何体。除了这些基础几何体，Three.js 库还提供了一些比较高级而且特别的对象。本章我们会展示这些高级几何体，并涵盖以下这些主题：

- 如何使用高级几何体，例如 THREE.ConvexGeometry、THREE.LatheGeometry 和 THREE.TubeGeometry。
- 如何使用 THREE.ExtrudeGeometry 从二维图形创建三维图形。我们会使用 Three.js 提供的函数来画这个二维图形，并且我们还会展示如何从一个外部加载的 SVG 图片创建出一个三维图形。
- 你可以通过轻松地修改前面章节中讨论过的图形来创建自定义图形。不过 Three.js 库也提供了一个 THREE.ParametricGeometry 对象。使用这个对象，可以基于一组方程来创建几何体。
- 最后，介绍如何使用 THREE.TextGeometry 来创建三维文字效果。
- 另外，我们还会展示如何使用二元操作从已有的几何体创建出新几何体。二元操作是由 Three.js 的扩展 ThreeBSP 提供的功能。

我们从 THREE.ConvexGeometry 开始讨论。

## 6.1 THREE.ConvexGeometry

通过 THREE.ConvexGeometry，我们可以围绕一组点创建一个凸包。所谓凸包就是包围这组点的最小图形。理解这个概念最简单的方式是看一个例子：打开示例 01-advanced-3d-geometries-convex.html，你会看到一个包围一组随机点的凸包。此几何体如图 6.1 所示。

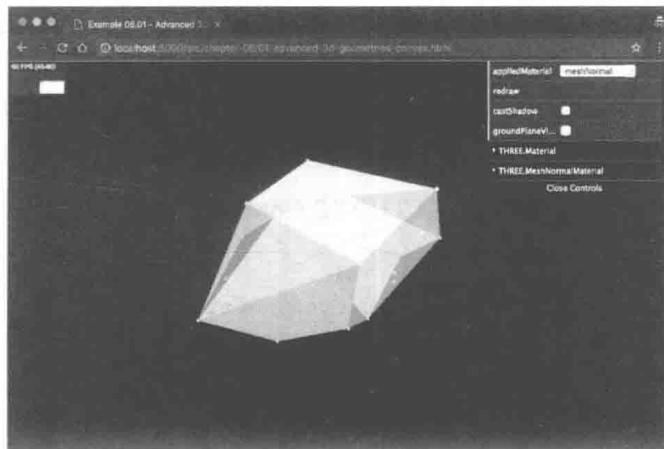


图 6.1

在这个例子里，我们随机生成了一组点，然后在这组随机点的基础上创建了一个 THREE.ConvexGeometry 对象。你可以在这个例子中点击 redraw（重新绘制）按钮，这将生成 20 个新的随机点并且绘制凸包。当在浏览器中运行示例程序时，请确保启用材质的半透明属性，并且将不透明度设置为一个小于 1 的值，这样就可以看到用于生成这个凸包的所有点。为了能够观察到这些点，我们用 THREE.SphereGeometry 在每个点的位置上生成了一个小球体。Three.js 的标准发布版中不包含 THREE.ConvexGeometry，因此你必须包含一个额外的 JavaScript 文件才能使用该几何体。在 HTML 页面的顶部添加以下代码：

```
<script src="../../libs/three/geometries/ConvexGeometry.js"></script>
```

下面的代码片段展示了这些点是如何创建并添加到场景中的：

```
var spGroup;
function generatePoints() {
    if (spGroup) scene.remove(spGroup)
    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: 0xffff00,
        transparent: false
    });
    points.forEach(function (point) {
        var spGeom = new THREE.SphereGeometry(0.2);
        var spMesh = new THREE.Mesh(spGeom, material);
        spMesh.position.copy(point);
        spGroup.add(spMesh);
    });
    // add the points as a group to the scene
    scene.add(spGroup);
}
```

正如你在这段代码中所看到的，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 THREE.SphereGeometry 对象，并把它们定位在这些点上（position.clone(point)）。所有这些点都被添加到一个组中，因此我们可以轻松地通过旋转这个组旋转它们。

一旦你有了这样一组点，创建一个 THREE.ConvexGeometry 对象就非常简单，如以下代码片段所示：

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

// if we want a smooth rendered object, we have to compute the vertex and
// face normals
convexGeometry.computeVertexNormals();
convexGeometry.computeFaceNormals();
convexGeometry.normalsNeedUpdate = true;
```

THREE.ConvexGeometry 的构造函数接收一个顶点数组（THREE.Vector3 类型对象的数组）作为唯一参数。在上面的代码中可以看到，我们明确地调用了 computeVertexNormals 和 computeFaceNormals 两个函数来计算法线，这是因为顶点和面的法线是 Three.js 为物体渲染平滑的表面所必需的数据。虽然大部分几何体在创建对象时会自动计算这些法线，但是只有这个类的对象是个例外，它需要我们主动调用函数去执行该计算。

接下来介绍的复杂几何体是 THREE.LatheGeometry，它用于创建类似花瓶的图形。

## 6.2 THREE.LatheGeometry

THREE.LatheGeometry 允许你从一条光滑曲线创建图形。此曲线是由多个点（也称为节点）定义的，通常称作样条曲线。这条样条曲线绕物体的中心 z 轴旋转，得到类似花瓶或铃铛的图形。同样，要理解 THREE.LatheGeometry 最简单的方式是看一个例子。示例 02-advanced-3d-geometries-lathe.html 展示的就是这样一个几何体，如图 6.2 所示。

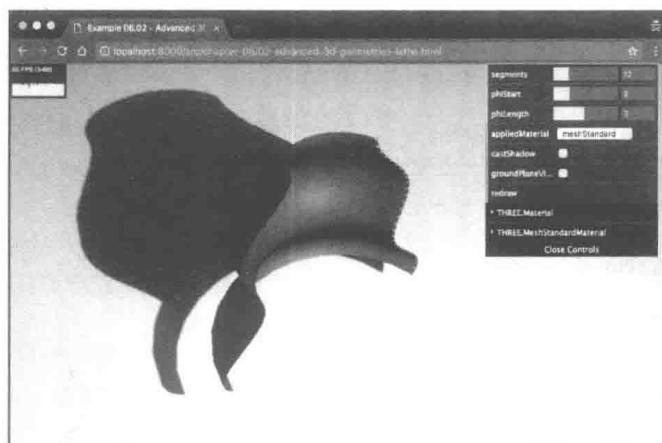


图 6.2

在上面的截图中，你可以看到这条样条曲线是由一组红色小球组成的。这些小球的位置连同一些其他参数一起传递给 THREE.LatheGeometry。在看所有参数之前，我们先来看看创建样条控制点的代码，以及 THREE.LatheGeometry 如何使用这些点：

```
function generatePoints(segments, phiStart, phiLength) {
    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 LatheGeometry
    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  $x$  坐标由正弦函数和余弦函数决定， $z$  坐标基于变量  $i$  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屏幕上。

基于这些点我们就可以创建 THREE.LatheGeometry。除了这个顶点数组，THREE.LatheGeometry 还接受其他几个参数。表 6.1 所列的就是所有这些参数。

表 6.1

属性	是否必需	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。这个数值越高，最终的图形越圆。默认值为 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 $2\pi$ 。默认值为 0
phiLength	否	该属性指定创建的图形有多完整。例如四分之一图形就是 $0.5\pi$ 。默认值为完整的 360 度或 $2\pi$

在下一节，我们将学习另一种创建几何体的方法，即如何把二维图形拉伸成三维图形。

### 6.3 通过拉伸创建几何体

Three.js 提供了几种方法，让我们可以从一个二维图形拉伸出三维图形。拉伸指的是沿

着 z 轴拉伸二维图形，将它转换成三维图形。例如，如果我们拉伸 THREE.CircleGeometry，就会得到一个类似圆柱体的图形；如果我们拉伸 THREE.PlaneGeometry，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 THREE.ExtrudeGeometry

通过 THREE.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。示例效果如图 6.3 所示。

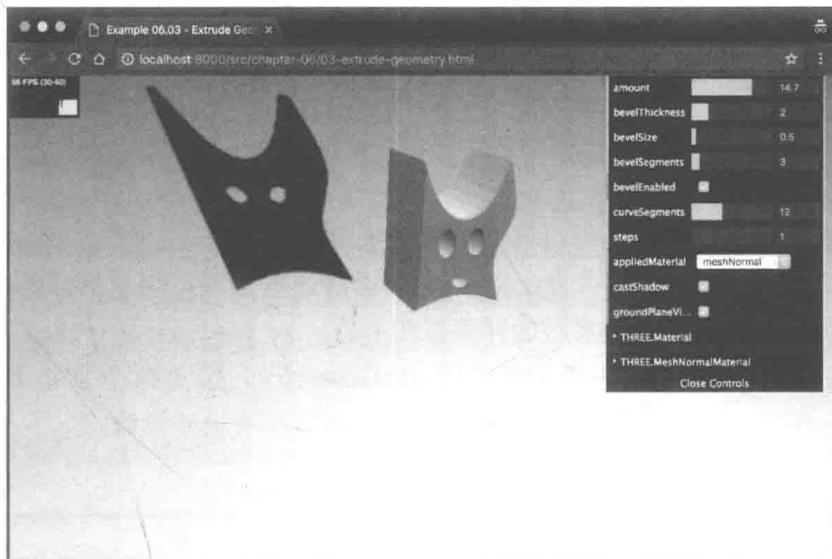


图 6.3

在这个例子里，我们使用了上一章创建的二维图片，并使用 THREE.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 z 轴拉伸，最终形成了一个三维图形。创建 THREE.ExtrudeGeometry 的代码非常简单：

```
var options = {
  amount: controls.amount,
  bevelThickness: controls.bevelThickness,
  bevelSize: controls.bevelSize,
  bevelSegments: controls.bevelSegments,
  bevelEnabled: controls.bevelEnabled,
  curveSegments: controls.curveSegments,
  steps: controls.steps
};

var geom = new THREE.ExtrudeGeometry(drawShape(), options);
```

跟上一章的一样，在这段代码中，我们使用 drawShape() 函数创建了一个图形。然后将这个图形作为参数连同 options 对象一起传递给 THREE.ExtrudeGeometry 构造函数。通过 options 对象，你可以明确定义图形应该怎样拉伸。表 6.2 是对可以传递给 THREE.ExtrudeGeometry 的各个选项的解释。

表 6.2

属性	是否必需	描述
shapes	是	拉伸几何体需要提供一个或多个图形（THREE.Shape 对象）。有关如何创建这样的图形请参阅前面的章节
amount	否	该属性指定图形可以拉多高（深度）。默认值为 100
bevelThickness	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。该值定义斜角进入图形的深度。默认值为 6
bevelSize	否	该属性指定斜角的高度。这个高度将被加到图形的正常高度上。默认值为 bevelThickness - 2
bevelSegments	否	该属性定义斜角的分段数。分段数越多，斜角越平滑。默认值为 3
bevelEnabled	否	如果这个属性设为 true，就会有斜角。默认值为 true
curveSegments	否	该属性指定拉伸图形时曲线分成多少段。分段数越多，曲线越平滑。默认值为 12
steps	否	该属性指定拉伸体沿深度方向分成多少段。默认值为 1。值越大，单个面越多
extrudePath	否	该属性指定图形沿着什么路径（THREE.CurvePath）拉伸。如果没有指定，则图形沿着 z 轴拉伸
uvGenerator	否	当你给材质使用纹理时，UV 映射确定纹理的哪一部分用于特定的面。使用 uvGenerator 属性，你可以传入自己的对象，该对象将为传入的图形创建的面创建 UV 设置。有关 UV 设置的更多信息请参考第 10 章。如果没有指定，则使用 THREE.ExtrudeGeometry.WorldUVGenerator

如果希望在正面和侧面使用不同的材质，可以向 THREE.Mesh 传入一个材质数组。数组中的第一个材质将被应用于正面，第二个则被应用于侧面。可以通过示例程序 03-extrude-geometry.html 中的菜单试验这些方法。

在这个例子里，我们沿 z 轴拉伸图形。正如你在选项列表中看到的，你也可以使用 extrudePath 选项沿着一条路径拉伸图形。在对下一个几何体 THREE.TubeGeometry 的介绍里，我们就会那么做。

### 6.3.2 THREE.TubeGeometry

THREE.TubeGeometry 沿着一条三维的样条曲线拉伸出一根管。你可以通过指定一些顶点来定义路径，然后用 THREE.TubeGeometry 创建这根管。你可以在本章的源代码路径

下找到一个可以试验的例子（04-extrude-tube.html）。示例效果如图 6.4 所示。

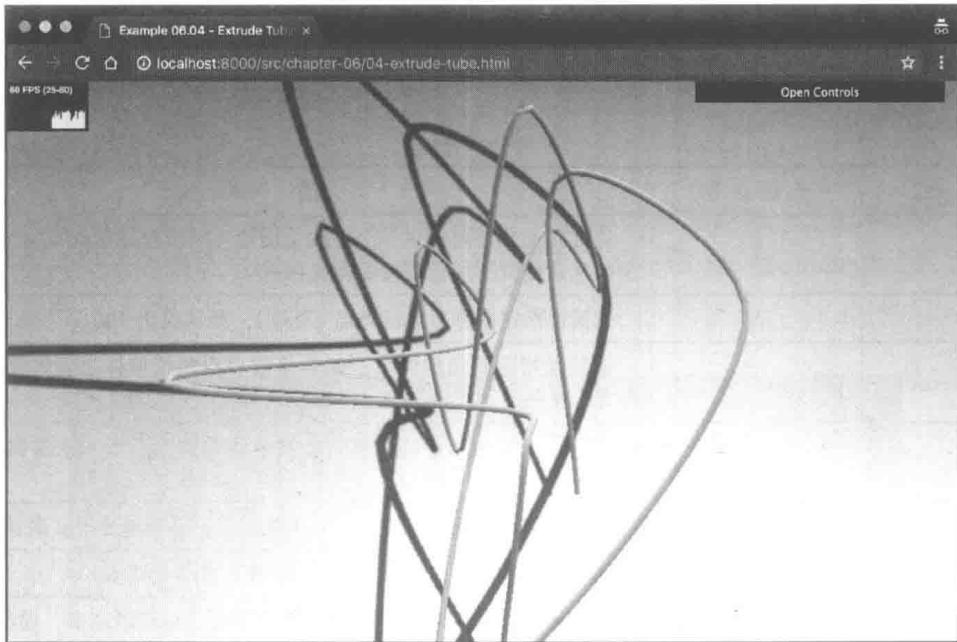


图 6.4

正如你在例子中看到的，我们随机生成了一些点，然后用这些点来绘制管。通过右上角的控件，我们可以定义管的外观，或者点击 newPoints 按钮生成新的管。创建管的代码很简单，如下所示：

```
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.CatmullRomCurve3(points), segments, radius, radiusSegments, closed);

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

我们首先要做的是获取一组 THREE.Vector3 类型的顶点，跟我们之前在 THREE.ConvexGeometry 和 THREE.LatheGeometry 示例里做的一样。但是，在使用这些点创建管之前，我们先要将这些点转换成 THREE.CatmullRomCurve3 对象。换句话说，我们需要用这些点来定义一条平滑的曲线。我们可以简单地通过将顶点数组传递到 THREE.CatmullRomCurve3 的构造函数来完成。使用这个样条曲线和其他参数（我们将在稍后解

释), 我们就可以创建管并将其添加到场景中。除了 THREE.SplineCurve3 之外, THREE.TubeGeometry 还接受一些其他参数。表 6.3 中列出了 THREE.TubeGeometry 的所有参数。

表 6.3

属性	是否必需	描述
path	是	该属性用一个 THREE.SplineCurve3 对象来指定管应当遵循的路径
segments	否	该属性指定构建这个管所用的分段数。默认值为 64。路径越长, 指定的分段数应该越多
radius	否	该属性指定管的半径。默认值为 1
radiusSegments	否	该属性指定管道圆周的分段数。默认值为 8。分段数越多, 管看上去越圆
closed	否	如果该属性设为 true, 管的头和尾会连起来。默认值为 false

本章要展示的最后一个拉伸示例不是什么新几何体。在下一节中, 我们将向你展示如何使用 THREE.ExtrudeGeometry 从已有的 SVG 路径中创建拉伸图形。

### 6.3.3 从 SVG 拉伸

我们在讨论 THREE.ShapeGeometry 时, 曾经提到过 SVG 与创建图形的方式基本相同。SVG 跟 Three.js 处理图形的方式非常一致。本节我们将看看如何使用来自 <https://github.com/asunderland/d3-threeD> 的小型库 SVG 路径转换成 Three.js 图形。(第 8 章还将介绍一个 Three.js 自带的用于读取 SVG 图形的加载器 THREE.SVGLoader。)

在示例 05-extrude-svg.html 中, 使用 ExtrudeGeometry 将一个蝙蝠侠标识符的 SVG 图形转换为三维图形。如图 6.5 所示。

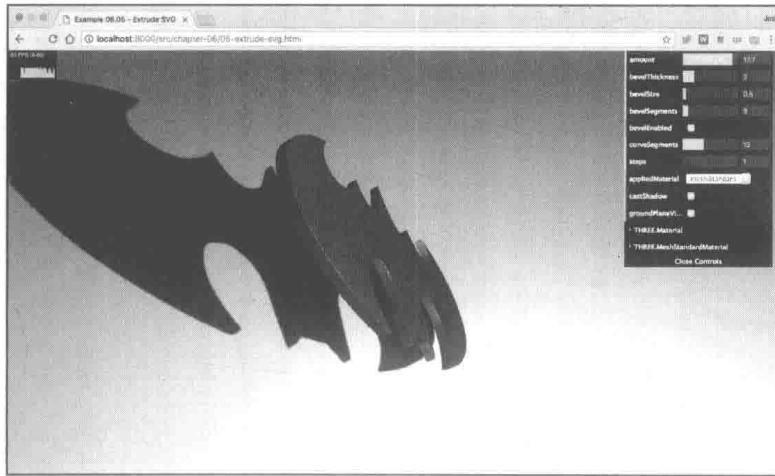


图 6.5

首先我们来看看原始的 SVG 代码的样子 (可以在本例的源代码中找到):

```

<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，否则这些代码对你来说毫无意义。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贝塞尔曲线，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。最初该库被创建为与优秀的 D3.js 库一起使用，但是我们对其做了一些小的修改，以便能够单独使用这个特定的功能。



SVG 的含义是可缩放矢量图（Scalable Vector Graphics）。它基于 XML 的标准，用来在网页上创建二维矢量图。该标准是一个所有现代浏览器都支持的开放标准。但是，直接使用 SVG，通过 JavaScript 来操作它并不直观。幸运的是，有几个开源 JavaScript 库使得使用 SVG 更简单。Paper.js、Snap.js、D3.js 和 Raphael.js 是其中最好的几个。

下面的代码展示了如何加载你之前看到的 SVG，将它转换为 THREE.ExtrudeGeometry，并显示在屏幕上：

```

function drawShape() {
  var svgString = document.querySelector("#batman-path").getAttribute("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 函数被调用。这个函数是由 d3-threeD 库提供的，接受一个 SVG 字符串作为参数。我们使用 document.querySelector("#batman-path").getAttribute("d") 表达式直接从 SVG 元素中获取 SVG 字符串。SVG 元素

的 `d` 属性包含的就是用来绘制图形的路径表达式。添加上好看的、闪亮的材质和一个聚光灯光源，就可以重建该示例。

下面后要讨论的几何体是 `THREE.ParametricGeometry`，通过该几何体，你可以指定几个函数来自动生成几何体。

## 6.4 THREE.ParametricGeometry

通过 `THREE.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。当你下载 `Three.js` 发布包后，你会得到 `examples/js/ParametricGeometries.js` 文件。在这个文件中，你可以找到几个公式的例子，它们可以和 `THREE.ParametricGeometry` 一起使用。最基础的例子是创建平面的函数，代码如下所示：

```
plane: function ( width, height ) {
    return function ( u, v, optionalTarget ) {
        var result = optionalTarget || new THREE.Vector3();
        var x = u * width;
        var y = 0;
        var z = v * height;
        return result.set( x, y, z );
    };
}
```

`THREE.ParametricGeometry` 会调用这个函数。 $u$  和  $v$  的取值范围从 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。当这个函数被调用时，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。但是创建的不是一个平面，而是一个类似波浪的图形，可以参见示例 `06-parametric-geometries.html`。本例效果如图 6.6 所示。

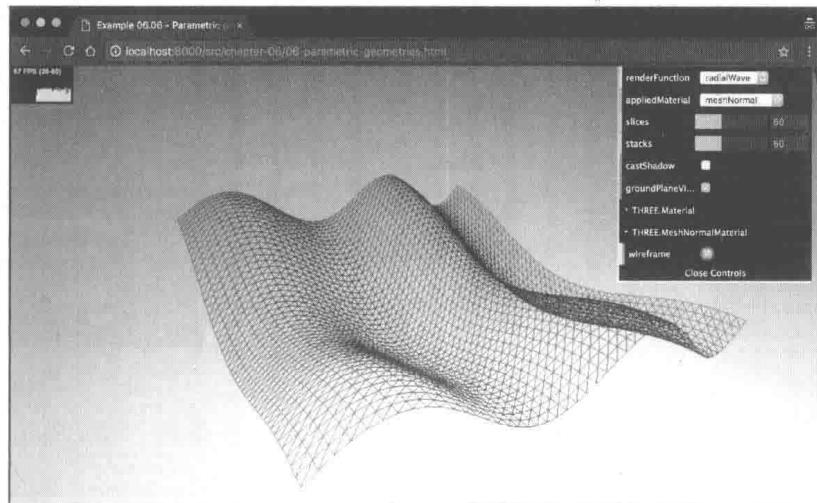


图 6.6

要创建这样的图形，我们需要将如下函数传给 THREE.ParametricGeometry：

```
radialWave = function (u, v, optionalTarget) {
    var result = optionalTarget || new THREE.Vector3();
    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 result.set( x, y, z );
};

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

正如你在示例中看到的，只要几行代码，我们就可以创建出非常有趣的几何体。在这个例子中，你也可以看到传递给 THREE.ParametricGeometry 的参数。表 6.4 是对这些参数的解释。

表 6.4

属性	是否必需	描述
function	是	该属性是一个函数，该函数以 $u$ 、 $v$ 值作为参数来定义每个顶点的位置
slices	是	该属性定义 $u$ 值应该分成多少份
stacks	是	该属性定义 $v$ 值应该分成多少份

使用一个不同的公式就可以绘制出完全不同的图形，如图 6.7 所示。

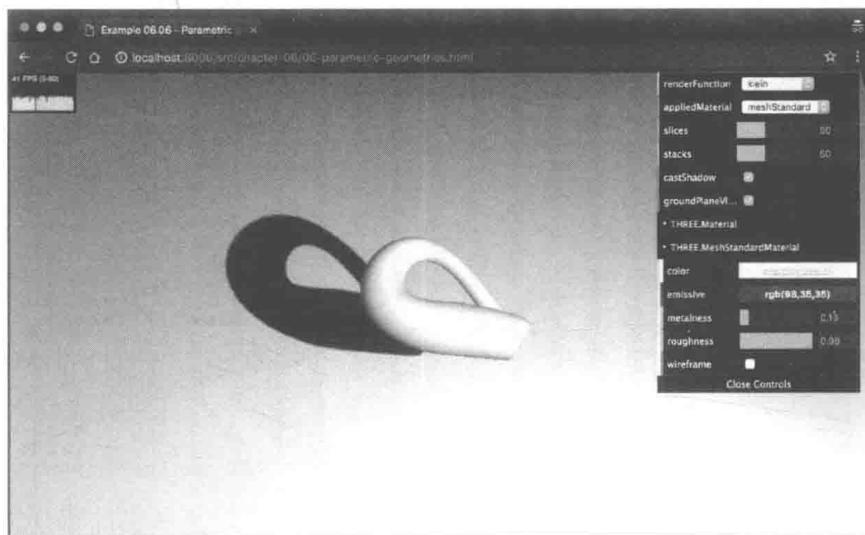


图 6.7

在进入下一节之前，先对 slices 和 stacks 属性的使用做一下最后的说明。我们曾经提到过  $u$ 、 $v$  属性会传递给由 function 属性指定的函数，并且这两个属性的取值范围是从 0 到 1。通过 slices 和 stacks 属性，我们可以指定传入的 function 函数的调用频率。例如，如果将 slice 属性设置为 5 并且将 stacks 属性设为 4，那么在调用函数时将使用以下参数值：

```
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
```

由此可知，值越大，生成的顶点越多，创建出来的几何体越平滑。在示例 06-parametric-geometries.html 中，你可以使用菜单来查看这个效果。

可以参考 Three.js 发布包里的 examples/js/ParametricGeometries.js 文件查看更多例子。这个文件包含的函数可以创建以下几何体：

- 克莱因瓶
- 平面
- 二维莫比乌斯带
- 三维莫比乌斯带
- 管
- 环状扭结
- 球体

下一节讲述的是如何创建三维文本对象。

## 6.5 创建三维文本

本节我们将会简要介绍如何创建三维文本效果。首先，我们会学习如何使用 Three.js 提供的字体来渲染文本，然后学习如何使用自定义的字体。

### 6.5.1 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要用的字体，以及我们在讨论 THREE.ExtrudeGeometry 时见过的基本拉伸属性。图 6.8 展示的就是如何在 Three.js 中渲染文本的例子（07-text-geometry.html）。

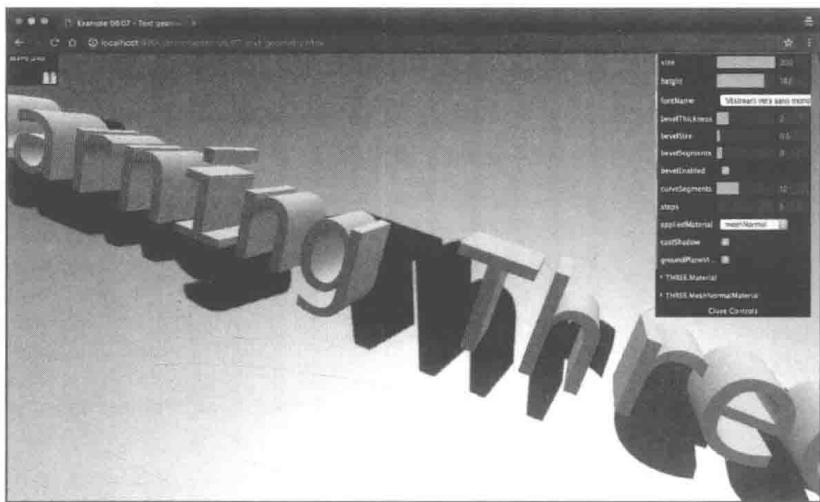


图 6.8

用来创建这个三维文本的代码如下所示：

```

var loadedFont;
var fontload = new THREE.FontLoader();
fontload.load(
    '../assets/fonts/bitstream_vera_sans_mono_roman.typeface.json',
    function ( response ) {
        loadedFont = response;
        render();
    }
);

var options = {
    size: 90,
    height: 90,
    font: loadedFont,
    bevelThickness: 2,
    bevelSize: 4,
    bevelSegments: 3,
    bevelEnabled: true,
    curveSegments: 12,
    steps: 1
};

// the createMesh is the same function we saw earlier
text = createMesh(new THREE.TextGeometry("Learning Three.js", options));
scene.add(text);

```

上面的代码首先利用 Three.js 提供的 THREE.FondLoader 类加载字体。在调用 THREE.FondLoader.load() 方法时，我们除了提供字体名称之外，还提供了一个临时回调函数，用于接收创建好的字体对象（即上面代码中的 response 对象），然后将字体对象保存到变量 loadedFont 中，并调用 render() 函数来重新绘制场景。接着我们将接收到的字体对象包含在一个名为 options 的映射表变量中，并用它来创建 THREE.TextGeometry 几何体对象。大部

分通过 options 向 THREE.TextGeometry 传入的参数与 THREE.ExtrudeGeometry 相同，至少少量参数专用于 THREE.TextGeometry。

让我们来看一看可以在 THREE.TextGeometry 里指定的所有选项，具体见表 6.5。

表 6.5

属性	是否必需	描述
size	否	该属性指定文本的大小。默认值为 100
height	否	该属性指定拉伸的长度（深度）。默认值为 50
font	是	该属性指定要用的字体名
bevelThickness	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。该值定义斜角进入图形的深度。默认值为 10
bevelSize	否	该属性指定斜角的高度。默认值为 8
bevelSegments	否	该属性定义斜角的分段数。分段数越多，斜角越平滑。默认值为 3
bevelEnabled	否	如果这个属性设为 true，就会有斜角。默认值为 false
curveSegments	否	该属性指定拉伸图形时曲线分成多少段。分段数越多，曲线越平滑。默认值为 4
steps	否	该属性指定拉伸体被分成多少段。默认值为 1
extrudePath	否	该属性指定图形沿着什么路径拉伸。如果没有指定，则图形沿着 z 轴拉伸
uvGenerator	否	当你给材质使用纹理时，UV 映射确定纹理的哪一部分用于特定的面。使用 uvGenerator 属性，你可以传入自己的对象，该对象将为传入的图形创建的面创建 UV 设置。有关 UV 设置的更多信息请参考第 10 章。如果没有指定，则使用 THREE.ExtrudeGeometry.WorldUVGenerator

Three.js 所包含的字体也被添加到本书的源代码中了。你可以在文件夹 assets/fonts 中找到它们。由于 THREE.TextGeometry 类是基于 THREE.ExtrudeGeometry 类的扩展，因此同样可以为字体形体的正面和侧面指定不同材质。如果向字体对象提供包含两个材质的数组，则数组中的第一个材质会被用于字体的正面，而第二个材质则被用于侧面。



如果你想渲染二维文字并用作材质的纹理，那么你就不应该使用 THREE.TextGeometry。THREE.TextGeometry 在内部使用 THREE.ExtrudeGeometry 构建三维文本，并且 JavaScript 字体引入了大量开销。渲染一个简单的二维字体，使用 HTML5 画布会更好。通过 context.font，可以设置要使用的字体，通过 context.fillText，可以将文本输出到画布。然后，你可以使用此画布作为纹理的输入。我们将在第 10 章告诉你如何做到这一点。

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

## 6.5.2 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。这些字体基于由 TypeFace.js 库提供的字体。TypeFace.js 库可以将 TrueType 和 OpenType 字体转换为 JavaScript 文件或者 JSON 文件，以便在网页中的 JavaScript 程序中直接使用。在旧版本的 Three.js 使用字体时，需要用转换得到的 JavaScript 文件，而新版 Three.js 改为使用 JSON 文件了。

可以访问网站 [http://gero3.github.io/facetype.js](https://gero3.github.io/facetype.js) 在线转换所需的 TrueType 和 OpenType 字体。如图 6.9 所示。

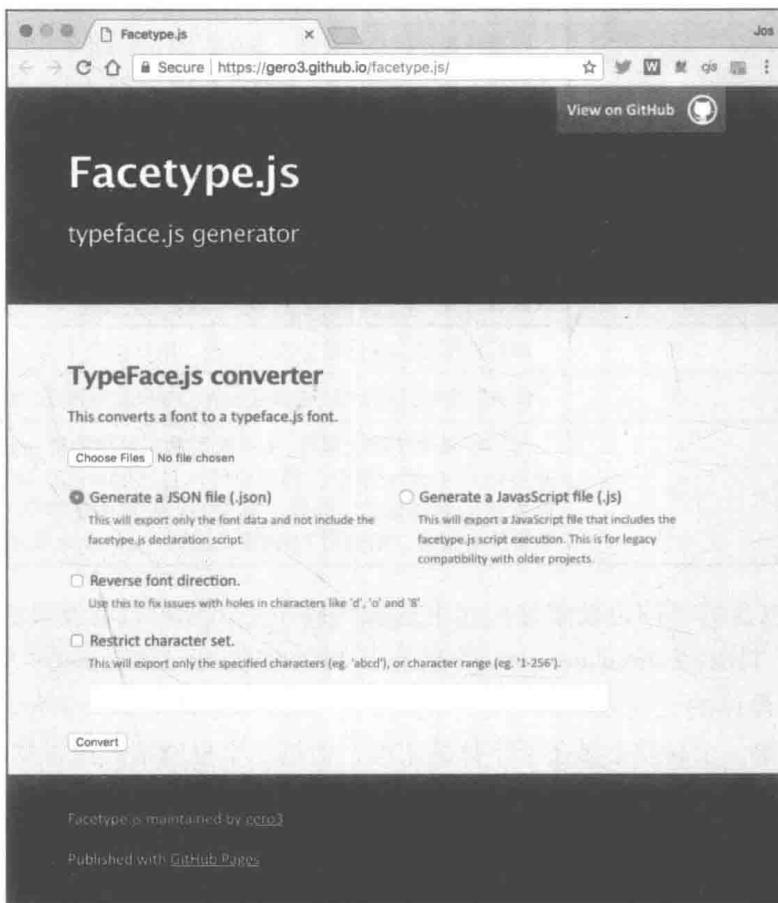


图 6.9

在上面的网页中，你可以上传一个字体并将它转换为 JSON 文件。请注意，并不是所有类型的字体都适用。在 Three.js 中使用时，字体越简单（直线更多），越容易被正确地渲染。在转换结果文件中，每一个字符都被描述为一段代码，看起来像下面这样：

```
{"glyphs": {"l": {"x_min": 359, "x_max": 474, "ha": 836, "o": "m 474 971 1 474 457 1
359 457 1 359 971 1 474 971 m 474 277 1 474 -237 1 359 -237 1 359 277 1 474
277 "}, "Ž": {"x_min": 106, "x_max": 793, "ha": 836, "o": "m 121 1013 1 778 1013 1
277 "}},
```

```

778 908 1 249 115 1 793 115 1 793 0 1 106 0 1 106 104 1 620 898 1 121 898 1
121 1013 m 353 1109 1 211 1289 1 305 1289 1 417 1168 1 530 1289 1 625 1289
1 482 1109 1 353 1109 "}, "A": {"x_min": 25, "x_max": 811, "ha": 836, "o": "m 417
892 1 27 ....

```

转换得到 JSON 文件后，你可以使用 THREE.FontLoader（就像我们在前面展示过的那样）加载字体，并将字体对象赋给 THREE.TextGeometry 的 font 属性。本章的下一节将会介绍 ThreeBSP 库，该库使用二元操作创建出相交、相减和联合等非常有趣的几何体。

## 6.6 使用二元操作组合网格

本节我们将看到另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。通过默认的属性组合，你可以创建出漂亮的模型，但是你所能做的也会受限于 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——一种被称为构造实体几何体（Constructive Solid Geometry, CSG）的技术，为此，我们引入了 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>。这个扩展库提供如表 6.6 所示的三个函数。

表 6.6

名 称	描 述
intersect (相交)	使用该函数可以基于两个现有几何体的交集创建出新的几何体。两个几何体重叠的部分定义此新几何体的形状
union (联合)	使用该函数可以将两个几何体联合起来创建出一个新的几何体。你可以将这个函数与我们在第 8 章讨论过的 mergeGeometry 功能相比较
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移除两个几何体重叠的部分来创建新的几何体

在下面的几节里我们将会详细地介绍每个函数。图 6.10 所示的例子是依次使用 union 和 subtract 功能所创建出来的几何体。

要使用这个库，需要把它包含在我们的页面中。这个库是用 CoffeeScript 写的，这是一种对用户更加友好的 JavaScript 脚本的变体。要使这个库可以正常工作，我们有两个选择：添加 CoffeeScript 文件，并在运行时编译；或者将它预先编译成 JavaScript 文件，然后直接包含它。对于第一种方法，我们需要这样做：

```

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

```

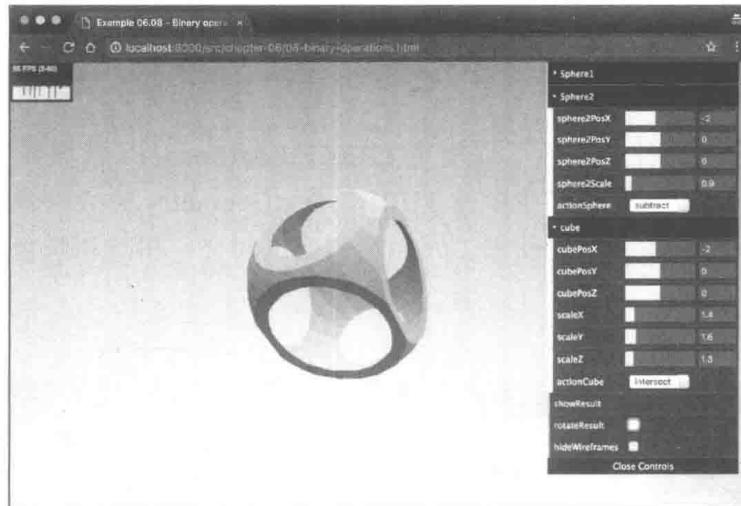


图 6.10

ThreeBSP.coffee 文件中包含我们在这个示例中所需要的功能，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，我们在文件的底部添加如下代码：

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

然后我们将原来的 `onload` 函数重命名为 `onReady`：

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

如果我们使用 CoffeeScript 命令行工具将 CoffeeScript 预编译为 JavaScript，就可以在页面中直接包含编译好的 JavaScript 文件。但是，在这样做之前，我们需要先安装 CoffeeScript。你可以按照 CoffeeScript 网站上的安装说明进行操作，网址为 <http://coffeescript.org/>。安装 CoffeeScript 后，就可以使用以下命令行将 ThreeBSP 文件从 CoffeeScript 转换成 JavaScript：

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

该命令行可以创建出一个 `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。在示例里我们使用的是第二种方法，因为相比每次加载网页时都要重新编译 CoffeeScript，这种方法加载起来更快。为此，我们仅需在 HTML 页面的顶部添加如下代码：

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

现在 `ThreeBSP` 库已经加载了，我们可以使用它提供的函数。

### 6.6.1 subtract 函数

在我们开始讲解 subtract 函数之前，有一个重要的内容需要记住：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是使用多种材质，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，应当确保使用的是未经组合的网格。

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试验这三种操作。第一次打开这个示例时，你看到的可能是如图 6.11 所示的开始屏幕。

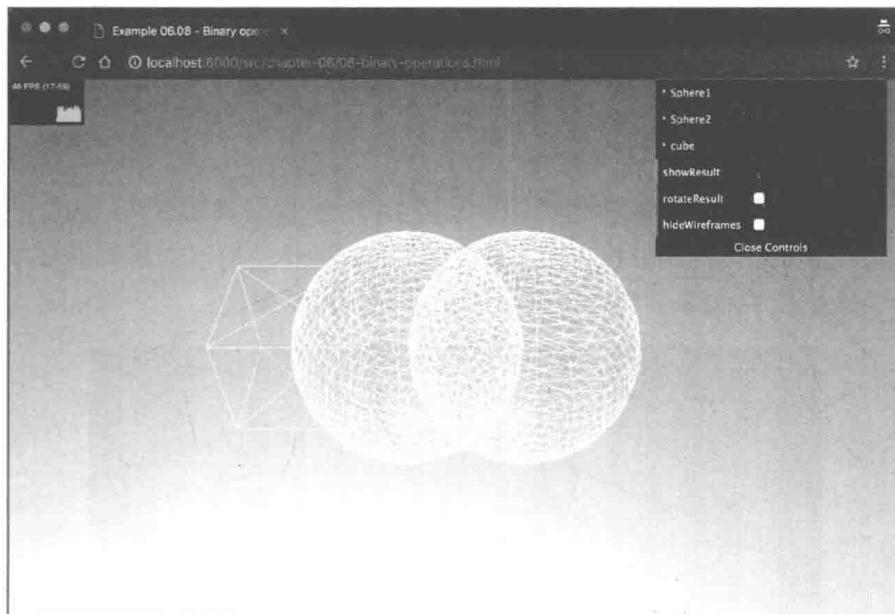


图 6.11

该场景中共有三个线框：一个方块和两个球体。Sphere1 是中间的那个球体，所有操作都会在这个对象上执行；Sphere2 是右边的那个球体，而 Cube 是左边的方块。在 Sphere2 和 Cube 上你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。当我们把 Sphere2 的操作设为 subtract 并选择 showResult（和隐藏线框），显示结果将是 Sphere1 减去 Sphere1 和 Sphere2 重叠的区域。

 需要说明的是，当点击 showResult 按钮后，其中的一些操作可能需要几秒钟来完成，所以在 busy 指示灯亮的时候请耐心等待。

图 6.12 展示的是一个球体在减去另一个球体后的操作结果。



图 6.12

可以看到右侧球体与中间球体相交的部分被切掉（或者说被减去）。在这个示例里，先定义在 Sphere2 上的操作，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），将会得到如图 6.13 所示的结果。

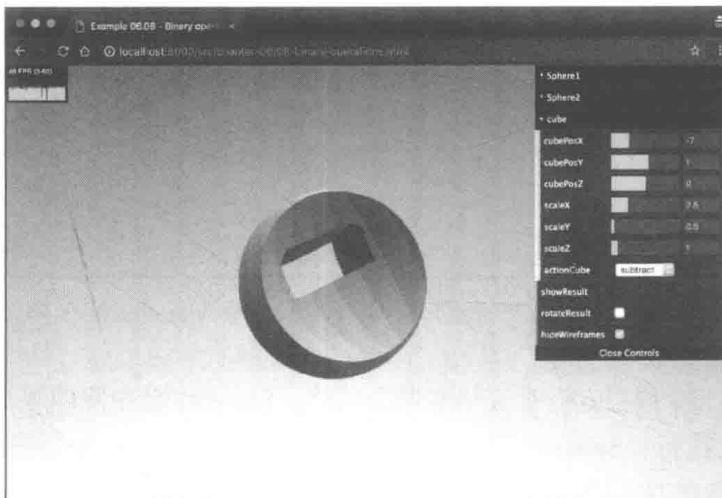


图 6.13

理解相减功能最好的方法是试验一下这个例子。完成该操作的 ThreeBSP 代码非常简单，本例是在 `redrawResult` 函数中实现的，当点击示例中 `showResult` 按钮时，该函数即被调用，代码如下所示：

```
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);
}

```

在这段代码中，我们首先将网格（那些我们能看到的线框）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，我们可以在中间那个球体的 ThreeBSP 对象（sphere1BSP）上调用指定的函数，并且该函数的结果将包含创建一个新的网格所需的所有信息。要创建这个网格，我们只需在 sphere1BSP 对象上调用 toMesh() 函数。在生成的对象上，我们必须确保所有的法线都是通过先调用 computeFaceNormals，然后调用 computeVertexNormals() 正确计算出来的。

之所以要调用这两个函数，是因为在执行任一二元操作之后，几何体的顶点和面会改变，并且面的法向量也会改变。显式地重新计算它们，可以保证新生成的对象着色光滑（当材质上的着色方式设置成 THREE.SmoothShading 时），并且正确呈现。最后，我们将结果添加到场景中。

对于 intersect 和 union，我们使用完全相同的方法。

## 6.6.2 intersect 函数

我们在前一节已经解释了所有的知识，有关 intersect 函数没有多少需要解释的了。使用该函数，只有网格重叠的部分会保留下。图 6.14 展示了球体和方块都设置为 intersect 的截图。

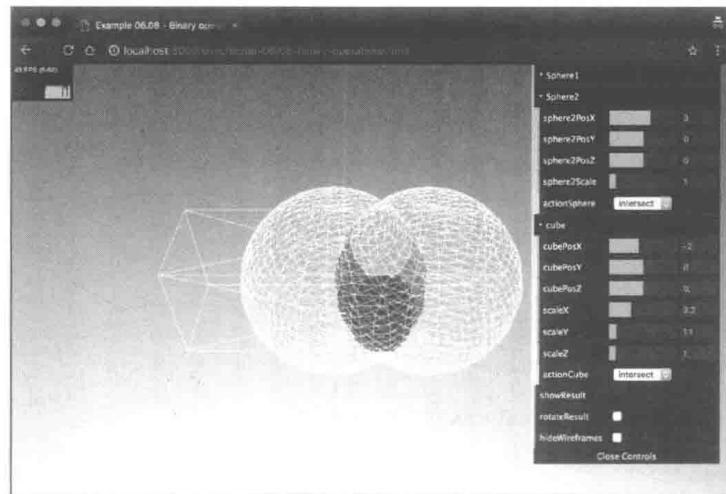


图 6.14

如果你看看这个例子并试验这些设置，你会发现创建这类对象非常简单。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 THREE.ParametricGeometry 和 THREE.TextGeometry。

subtract 和 intersect 函数可以同时使用。我们在本节开头所示的例子就是先减去一个小球，得到一个空心球体，然后再使用一个方块跟这个空心球相交，得到如图 6.15 所示的结果（具有圆角的空心立方体）。

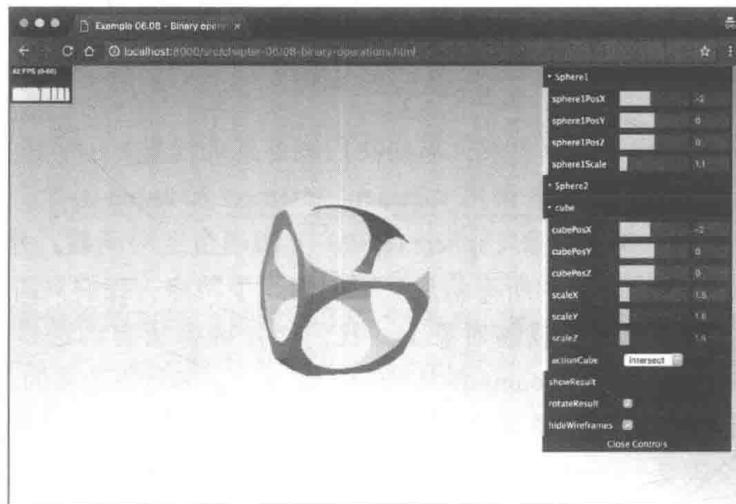


图 6.15

ThreeBSP 提供的最后一个函数是 union 函数。

### 6.6.3 union 函数

最后这个函数是 ThreeBSP 提供的函数里最无趣的一个。通过这个函数，我们可以将两个网格连成一体，从而创建出一个新的几何体。当我们将这个函数应用于两个球体和一个方块时，我们将得到一个单一的物体——union 函数的结果。如图 6.16 所示。

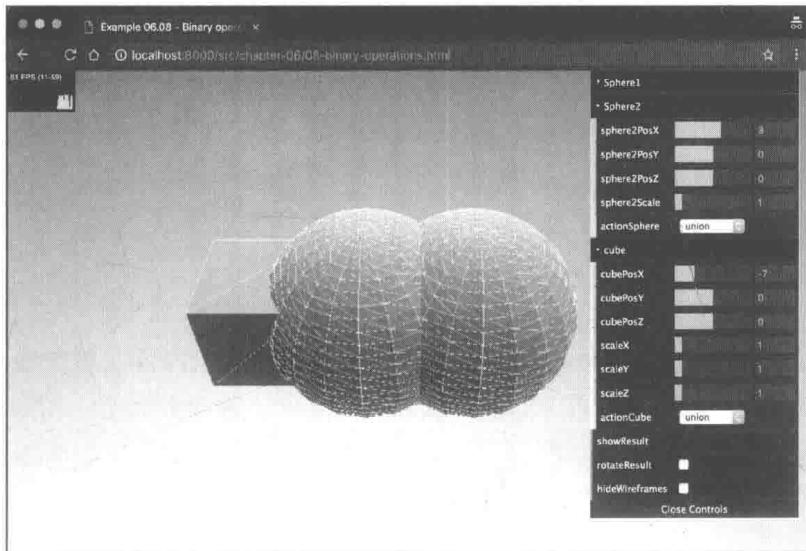


图 6.16

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.Geometry.merge），而且性能更好。如果旋转物体，你会发现这个联合操作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的。

## 6.7 总结

我们在本章学了很多东西。本章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建出看起来很有趣的几何体。我们展示了如何使用 THREE.ConvexGeometry、THREE.TubeGeometry 和 THREE.LatheGeometry 等高级几何体创建非常漂亮的图形，并且试验了这些几何体以得到你想要的结果。一个非常好的特性是可以将已有的 SVG 路径转换为 Three.js。但是，需要记住的是，你可能仍然需要使用诸如 GIMP、Adobe Illustrator 或 Inkscape 之类的工具来微调一下路径。

创建三维文本时，需要指定字体。Three.js 提供了几种字体，但你也可以创建自定义字

体。但是，复杂的字体往往不能被正确地转换。最后，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，你可以从源网格中移除它跟其他网格重叠的部分；通过 intersect，只保留网格重叠的部分。

到目前为止，我们看到的几何体都是立体（或线框）几何体，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。

在前面的章节里，我们讨论了最重要的概念、对象以及 Three.js 提供的 API。本章看一下我们一直忽略的一个概念：粒子和精灵。使用 THREE.Points（有时也叫作精灵（sprite）），可以非常容易地创建很多细小的物体，用来模拟雨滴、雪花、烟和其他有趣的效果。例如，你可以将整个几何体渲染成一组粒子，并分别控制它们。本章我们将探讨 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主题：

- 使用 THREE.SpriteMaterial 创建和样式化粒子。
- 使用 THREE.Points 创建一个粒子集合。
- 使用现有的几何体创建一个 THREE.Points 对象。
- 让 THREE.Points 对象动起来。
- 使用纹理对每一个粒子进行样式化。
- 使用画布通过 THREE.SpriteMaterial 对粒子进行样式化。

让我们先来探索什么是粒子，以及如何创建粒子。但是，在我们开始之前，对本章要用到的一些名称做一个快速的说明。Three.js 的最新版本更新了与粒子相关的对象的名称。THREE.Points 以前称为 THREE.PointCloud，在更早以前曾被称为 THREE.ParticleSystem。THREE.Sprite 以前称为 THREE.Particle，而且还更改了一些材质的名称。所以，如果你看到使用这些旧名称的在线示例，请记住，它们谈论的是相同的概念。在本章中，我们使用 Three.js 的最新版本中的新命名约定。

## 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-sprites.html 的示例。打开这个示例，你会看到一个由无趣的彩色方块组成的网格，如图 7.1 所示。

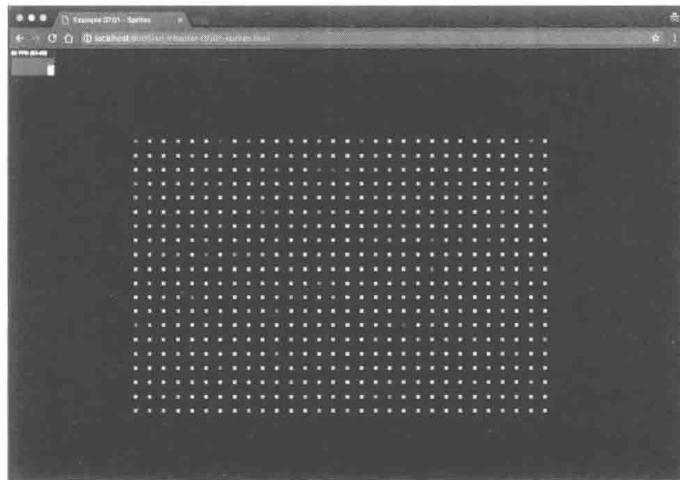


图 7.1

上面的示例程序也支持鼠标控制，所以你可以使用鼠标和触控板在场景中移动。在移动过程中，你会注意到一个事实，那么就是无论从哪个角度观察，阵列中的那些彩色方块本身看起来都没有变化。例如，在下面的截图中渲染了与前一张截图相同的场景，只不过是从不同角度观察所得，如图 7.2 所示。

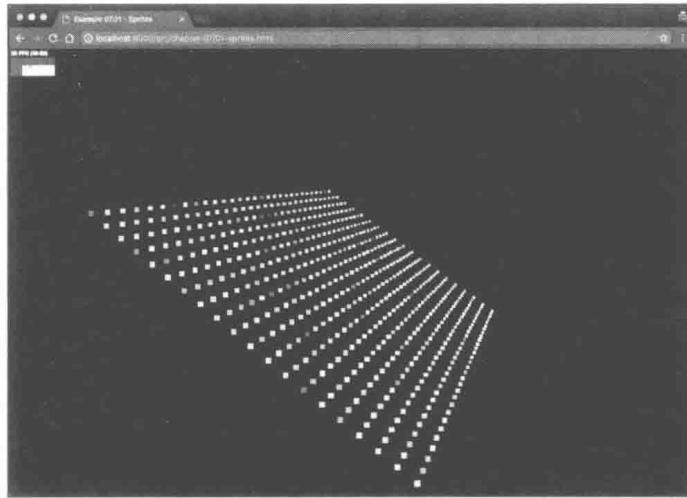


图 7.2

在上图中可以看到很多粒子。每一个粒子都是一个永远面向摄像机的二维平面。如果在创建粒子的时候没有指定任何属性，那么它们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 createSprites() {
    for (var x = -15; x < 15; x++) {
        for (var y = -10; y < 10; y++) {
            var material = new THREE.SpriteMaterial({
                color: Math.random() * 0xffffff
            });

            var sprite = new THREE.Sprite(material);
            sprite.position.set(x * 4, y * 4, 0);
            scene.add(sprite);
        }
    }
}

```

在这个例子里，我们使用 `Three.Sprite(material)` 构造函数手工创建粒子。我们传入的唯一参数是材质，它只能是 `THREE.SpriteMaterial` 或 `THREE.SpriteCanvasMaterial`。在本章后面的内容里，我们将会深入讲解这两种材质。

在讲解更有趣的 `THREE.Points` 对象之前，我们先来仔细看一下 `THREE.Sprite` 对象。跟 `THREE.Mesh` 一样，`THREE.Sprite` 对象也是 `THREE.Object3D` 对象的扩展。也就是说 `THREE.Mesh` 的大部分属性和函数都可以用于 `THREE.Sprite`。你可以用 `position` 属性来定位，用 `scale` 属性来进行缩放，用 `translate` 属性指定相对位移。



注意，在 `Three.js` 之前的版本中，是不能使用 `THREE.WebGLRender` 渲染 `THREE.Sprite` 对象的，只能使用 `THREE.CanvasRenderer` 渲染它。在当前版本中，`THREE.Sprite` 可以与这两个渲染器一起使用。

通过 `THREE.Sprite`，你可以非常容易地创建一组对象并在场景中移动它们。当你使用少量的对象时，这很有效，但是当你想使用大量的 `THREE.Sprite` 对象时，你会很快遇到性能问题，因为每个对象需要分别由 `Three.js` 进行管理。`Three.js` 提供了另一种方式来处理大量的粒子，这需要使用 `THREE.Points`。通过 `THREE.Points`，`Three.js` 不再需要管理大量单个的 `THREE.Sprite` 对象，而只需管理 `THREE.Points` 实例。

要使用 `THREE.Points` 获得与我们之前看到的截图相同的结果，可使用如下代码：

```

function createPoints() {

    var geom = new THREE.Geometry();
    var material = new THREE.PointsMaterial({
        size: 2,
        vertexColors: true,
        color: 0xffffffff
    });

    for (var x = -15; x < 15; x++) {
        for (var y = -10; y < 10; y++) {
            var particle = new THREE.Vector3(x * 4, y * 4, 0);
            geom.vertices.push(particle);
            geom.colors.push(new THREE.Color(Math.random() * 0xffffffff));
        }
    }
}

```

```

var cloud = new THREE.Points(geom, material);
scene.add(cloud);
}

```

我们需要为每个粒子创建一个顶点（用 THREE.Vector3 表示），并将它添加到 THREE.Geometry 中，然后使用 THREE.Geometry 和 THREE.PointsMaterial 一起创建 THREE.Points，再把云添加到场景中。在示例 02-points-webgl.html 中可以找到 THREE.Points 的例子，例子效果如图 7.3 所示。

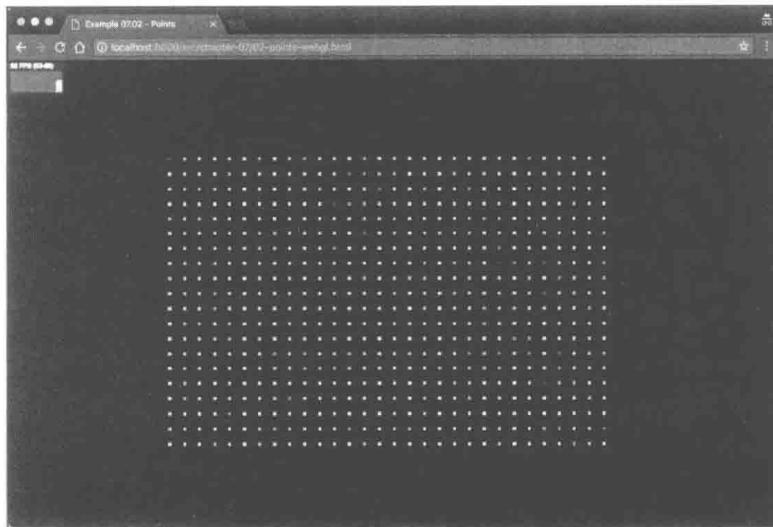


图 7.3

我们将在下面进一步讨论 THREE.Points。

## 7.2 THREE.Points 和 THREE.PointsMaterial

在上节末尾我们快速介绍了一下 THREE.Points。THREE.Points 的构造函数接收两个属性：几何体和材质。材质用来给粒子着色和添加纹理（稍后即可看到），而几何体则用来指定单个粒子的位置。每个顶点和每个用来定义几何体的点，将会以粒子的形态展示出来。如果我们基于 THREE.BoxGeometry 对象创建 THREE.Points 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 THREE.Points，而是从零开始手工将顶点添加到几何体上（或使用一个外部加载的模型），就像我们在上一节末尾所做的一样。本节我们将进一步来看看这种方法，并且看看如何使用 THREE.PointsMaterial 来样式化粒子。我们将会通过示例 03-basic-point-cloud.html 来讲解这种方法。本例效果如图 7.4 所示。

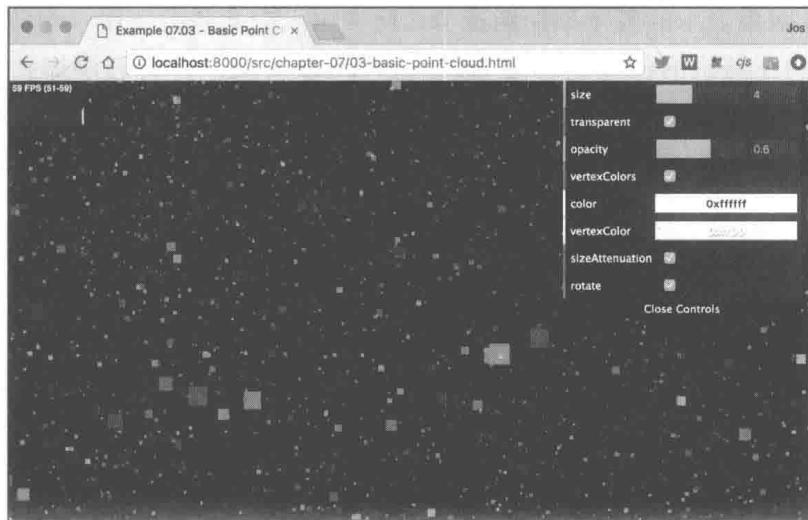


图 7.4

在这个例子中，我们创建了一个 THREE.Points 对象，并添加了 15 000 个粒子。所有这些粒子都用 THREE.PointsMaterial 样式化。创建 THREE.Points 的代码如下所示：

```
function createParticles(size, transparent, opacity, vertexColors,
sizeAttenuation, colorValue, vertexColorValue) {
    var geom = new THREE.Geometry();
    var material = new THREE.PointsMaterial({
        size: size,
        transparent: transparent,
        opacity: opacity,
        vertexColors: vertexColors,
        sizeAttenuation: sizeAttenuation,
        color: new THREE.Color(colorValue)
    });

    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(vertexColorValue);
        var ashsl = {};
        color.getHSL(ashsl);
        color.setHSL(ashsl.h, ashsl.s, ashsl.l * Math.random());
        geom.colors.push(color);
    }

    cloud = new THREE.Points(geom, material);
    cloud.name = "particles";
    scene.add(cloud);
}
```

在上面列出的代码里，我们首先创建了一个 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 对象，并将它添加到几何体中。在这个循环中，我们还定义了一个颜色数组：geom.colors。它在 THREE.PointsMaterial 的 vertexColors 属性设置为 true 时会用到。最后要做的是创建 THREE.PointsMaterial 对象，并添加到场景中。

表 7.1 是 THREE.PointsMaterial 对象中所有可设置属性的说明。

表 7.1

名 称	描 述
color	粒子系统中所有粒子的颜色。将 vertexColors 属性设置为 true，并且通过颜色属性指定了几何体的颜色来覆盖该属性（更准确地说，顶点的颜色将乘以此值以确定最终颜色）。默认值为 0xFFFFFFFF
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，不过在本章后面的内容中会有进一步的解释
size	该属性指定粒子的大小。默认值为 1
sizeAnnotation	如果该属性设置为 false，那么所有粒子都将拥有相同的尺寸，无论它们距离摄像机有多远。如果设置为 true，粒子的大小取决于其距离摄像机的远近。默认值为 true
vertexColors	通常，THREE.Points 中的所有粒子都拥有相同颜色。如果该属性设置为 THREE.VertexColors，并且几何体的颜色数组也有值，那就会使用颜色数组中的值。默认值为 THREE.NoColors
opacity	该属性与 transparent 属性一起使用，用来设置粒子的不透明度。默认值为 1（完全无透明）
transparent	如果该属性设置为 true，那么粒子在渲染时会根据 opacity 属性的值来确定其透明度。默认值为 false
blending	该属性指定渲染粒子时的融合模式
fog	该属性决定粒子是否受场景中的雾化效果影响。默认值为 true

前面例子里有一个简单的控制菜单，你可以使用它试验 THREE.ParticlesMaterial 的各种属性。

到目前为止，我们只是把粒子渲染为小方块，这是默认行为。但是，还有其他的一些方式可以用来样式化粒子：

- 我们可以应用 THREE.SpriteCanvasMaterial（只适应于 THREE.CanvasRenderer），将 HTML 画布元素绘制的内容作为粒子的纹理。
- 在使用 THREE.WebGLRenderer 时，可以使用 THREE.SpriteMaterial 渲染 HTML 画布的元素。
- 使用 THREE.PointsMaterial 的 map 属性，加载外部图片文件（或使用 HTML5 画布）来样式化 THREE.Points 对象中的所有粒子。

下面我们将来看看如何使用这些方法。

## 7.3 使用HTML5画布样式化粒子

Three.js 提供了三种使用 HTML5 画布来样式化粒子的方法。如果你使用的是 THREE.CanvasRenderer，那么可以通过 THREE.SpriteCanvasMaterial 直接引用 HTML5 画布。如果你用的是 THREE.WebGLRenderer，那么需要采取一些额外的步骤才能使用 HTML5 画布样式化粒子。我们将在下面的两节里分别讲述这些不同的方法。

### 7.3.1 在 THREE.CanvasRenderer 中使用 HTML5 画布

通过 THREE.SpriteCanvasMaterial，你可以将 HTML5 画布的输出结果作为粒子的纹理。这种材质是专门为 THREE.CanvasRenderer 创建的，而且只能在使用这种渲染器时才有效。在我们开始介绍如何使用这种材质之前，让我们先来看看可以设置在这种材质上的属性，详见表 7.2。

表 7.2

名 称	描 述
color	该属性指定粒子的颜色。根据指定的混合模式，影响画布图片的颜色
program	该属性指定一个函数，这个函数以 canvas 上下文作为参数。当渲染粒子时，此函数会被调用。调用此二维绘图上下文的输出将显示为粒子
opacity	该属性决定粒子的不透明度。默认值为 1，表示完全不透明
transparent	该属性决定粒子是否透明显示。它与 opacity 属性一起使用
blending	该属性指定粒子的融合模式
rotation	此属性允许你旋转画布的内容。通常需要将该属性设置为 PI，以正确对齐画布的内容。请注意，此属性不能传入到材质的构造函数，而是需要明确地设置

要查看 THREE.SpriteCanvasMaterial 的效果，可以打开示例文件 04-program-based-sprites.html。示例效果如图 7.5 所示。

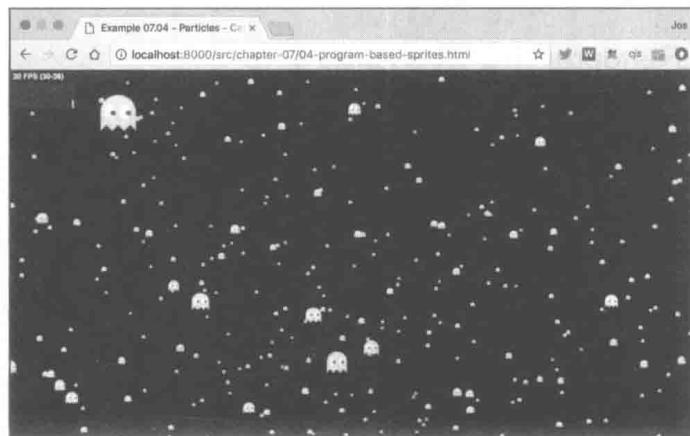


图 7.5

在本示例中，粒子是通过 `createSprites()` 函数创建的：

```
function createSprites() {
    var material = new THREE.SpriteCanvasMaterial({
        program: getTexture
    });
    material.rotation = Math.PI;

    var range = 500;
    for (var i = 0; i < 1000; i++) {
        var sprite = new THREE.Sprite(material);
        sprite.position = new THREE.Vector3(Math.random() * range -
            range / 2, Math.random() * range - range / 2, Math.random() *
            range - range / 2);
        sprite.scale.set(0.1, 0.1, 0.1);
        scene.add(sprite);
    }
}
```

这段代码与上一节看到的代码非常相似。因为我们使用的是 `THREE.CanvasRenderer`，所以主要的改变是不使用 `THREE.Points`，而是直接创建 `THREE.Sprite` 对象。在这段代码中，我们还使用 `program` 属性来定义 `THREE.SpriteCanvasMaterial`，`program` 属性指向 `getTexture` 函数。`getTexture` 函数定义了粒子的外观（在我们的例子中，是 Pac-Man 中的精灵）：

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

我们不会深入讲解绘制形状所需的画布（canvas）代码。重要的是，我们定义了一个接受二维画布上下文（ctx）作为其参数的函数。所有绘制到该上下文的内容都会被用作粒子（`THREE.Sprite` 对象）的外形。

### 7.3.2 在 `WebGLRenderer` 中使用 HTML5 画布

如果我们想要在 `THREE.WebGLRenderer` 中使用 HTML 画布，可以采取两种不同的方法。可以使用 `THREE.PointsMaterial` 并创建 `THREE.Points` 对象，或者可以使用 `THREE.Sprite` 和 `THREE.SpriteMaterial` 的 `map` 属性。

我们先从第一种方法开始，并创建 `THREE.Points` 对象。在讲解 `THREE.PointsMaterial` 的属性时，我们提到过 `map` 属性。通过 `map` 属性，我们可以为粒子加载纹理。在 `Three.js` 中，该纹理也可以是 HTML5 画布的输出。你可以在 `05-program-based-points-webgl.html` 中找到这个概念的示例。其输出如图 7.6 所示。



图 7.6

让我们来看看实现该效果的代码。大部分代码跟之前 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：

```
// code can be found in the util.js file
var createGhostTexture = 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 createPoints(size, transparent, opacity,
    sizeAttenuation, color) {

    var geom = new THREE.Geometry();
    var material = new THREE.PointsMaterial ({
        size: size,
        transparent: transparent,
        opacity: opacity,
        map: createGhostTexture(),
        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);
}

cloud = new THREE.Points(geom, material);
scene.add(cloud);
}

```

在这两个 JavaScript 函数中，在第一个函数（`createGhostTexture`）里，我们基于 HTML5 画布元素创建了一个 `THREE.Texture` 对象。在第二个函数（`createPoints`）里，我们将这个纹理赋给 `THREE.PointsMaterial` 的 `map` 属性。



在旧版 Three.js 里 `THREE.Points` 对象（旧称 `THREE.PointsCloud`）有 `sortParticles` 属性。在 JavaScript 中设置该属性可以令 Three.js 先将粒子按照相对摄像机的 z 坐标进行排序，然后再送往 WebGL 渲染。由于该操作会消耗过多 CPU 资源，因此在新版 Three.js（比如 r70 版）中被去除了。带来的结果是，当你仔细观察本示例渲染的画面时，会发现当一些粒子重叠时，有时会有错误的透明效果。想要保证粒子不重叠，你将不得不自行实现排序功能，或者修改材质的 `alphaTest` 或 `depthWrite` 属性。示例程序 `07-rainy-scene.html` 演示了排序操作，而示例程序 `08-snowy-scene.html` 演示了操作 `depthWrite` 属性。

上述函数的调用结果是：在 `createGhostTexture()` 方法中绘制到画布上的所有内容都会用于 `THREE.Points` 中的粒子。在下一节，我们将深入了解如何从外部文件加载纹理。



在本例中，我们只了解了关于纹理的很小一部分内容。第 10 章将深入了解纹理能够实现的更多功能。

在本节开头提到过，也可以使用 `THREE.Sprite` 和 `map` 属性来创建基于画布的粒子。为此，我们使用相同的方法来创建 `THREE.Texture`，如前面的示例所示。但是，这一次我们将它赋给 `THREE.Sprite`，如下所示：

```

function createSprites() {
    var material = new THREE.SpriteMaterial({
        map: getTexture(),
        color: 0xffffffff
    });

    var range = 500;
    for (var i = 0; i < 1500; i++) {
        var sprite = new THREE.Sprite(material);
        sprite.position.set(Math.random() * range - range / 2,
            Math.random() * range - range / 2, Math.random() * range -
            range / 2);
        sprite.scale.set(4, 4, 4);
        scene.add(sprite);
    }
}

```

## 7.4 使用纹理样式化粒子

前面我们介绍了如何使用HTML5画布对THREE.Points和单个THREE.Sprite对象进行样式化。既然可以绘制任何想要的东西，甚至加载外部图像，那么也可以使用这种方式将各种样式添加到粒子系统中。然而，使用图像样式化粒子还有一种更直接的方法。你可以使用THREE.TextureLoader().load()函数将图像加载为THREE.Texture对象，然后就可以将THREE.Texture分配给材质的map属性。

我们将会在本节展示两个例子，并解释如何创建它们。这两个例子都是使用图像作为粒子的纹理。在第一个例子（06-rainy-scene.html）里，我们模拟了雨滴。如图7.7所示。

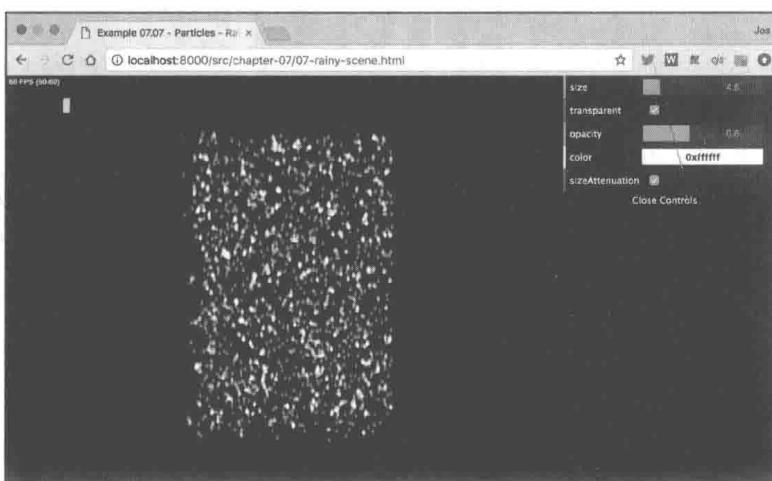


图 7.7

我们首先要做的是获得表示雨滴的纹理。你可以在assets/textures/particles文件夹下找到几个例子。我们将在后面章节讲解使用纹理的所有细节和要求。现在你只要知道这个纹理应该是正方形的，并且尺寸最好是2的幂（例如64×64、128×128、256×256）。在本例中，我们使用如图7.8所示纹理。

该图片使用了黑色的背景（为了能够正确地融合），并且展示了一个雨滴的形状和颜色。在THREE.PointsMaterial中使用这个纹理之前，我们首先需要加载它。可以使用下面的代码来完成：

```
var texture = new
THREE.TextureLoader().load("../assets/textures/particles/raindrop-3.png");
;
```

通过这行代码，Three.js会加载纹理，这样我们就可以在材质中使用它。在本例中，我们使用如下代码来定义材质：

```
var material = new THREE.PointsMaterial({
size: size,
```

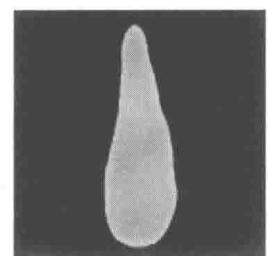


图 7.8

```

        transparent: transparent,
        opacity: opacity,
        map: texture,
        blending: THREE.AdditiveBlending,
        sizeAttenuation: sizeAttenuation,
        color: color
    });
}

```

我们已经在本章讨论了所有这些属性。需要重点理解的是，map 属性指向我们用 THREE.TextureLoader().load() 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。另外一种合理的方式是将纹理里的黑色背景定义成透明背景，这样便可以获得类似的效果。



在旧版 Three.js 中，无法基于 WebGL 和 THREE.Points（或相关对象）正确地渲染使用透明材质的粒子，但是这一问题在新版 Three.js 中已经解决。

这样就完成了 THREE.Points 的样式化。当你打开这个示例时，你还会发现这些粒子在移动。在前面的例子中，我们移动了整个对象；这次我们设置的是 THREE.Points 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 THREE.Points 对象几何体上的顶点。我们来看看如何为 THREE.Points 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.velocityX = (Math.random() - 0.5) / 3;
    particle.velocityY = 0.1 + (Math.random() / 5);
    geom.vertices.push(particle);
}

```

这跟我们前面看到的例子没有太大区别。在这里，我们给每个粒子（THREE.Vector3 对象）添加了两个额外的属性：velocityX 和 velocityY。第一个属性定义粒子（雨滴）如何水平移动，而第二个属性定义雨滴以多快的速度落下。横向运动速度的范围是  $-0.16 \sim +0.16$ ，纵向运动速度的范围是  $0.1 \sim 0.3$ 。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：

```

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

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

```

在这段代码中，我们从几何体中获取用来创建 THREE.Points 对象的所有顶点（粒子）。

对于每个粒子，我们用 velocityX 和 velocityY 来改变它们的当前位置。最后两行代码用来保证粒子处于我们定义的范围内。如果 v.y 的位置低于 0，我们就把雨滴放回顶部；如果 v.x 的位置到达任何一条边界，我们就反转横向速度，让雨滴反弹。

如果运行包含上述代码的示例程序，你很可能会注意到画面中很多重叠的雨滴之间有不太理想的叠加效果，如图 7.9 所示。

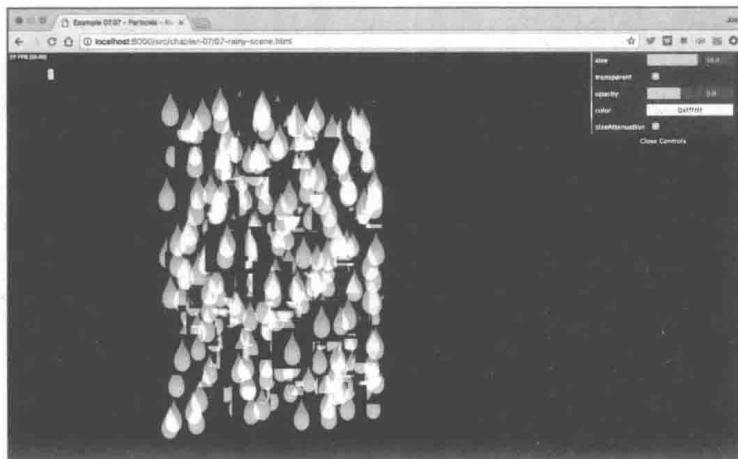


图 7.9

这是由于上面代码使用了 `Math.random() * range-range / 2` 的方法来计算随机 z 坐标，因此这些雨滴在 z 轴上是乱序的。前面提到过，在旧版 Three.js 中可以通过将 `sortParticles` 属性设置为 `true` 来启用自动排序功能，但是这一功能在新版 Three.js 中已经被去除，因此我们必须自行解决乱序问题。为了修正这一问题，我们只需将 z 坐标赋值为 `1 + (i/100)`，即可保证在 z 轴上后加入的粒子总是位于先加入的粒子上面，从而使粒子在 z 轴上有序。修改之后便可获得正确的效果，如图 7.10 所示。

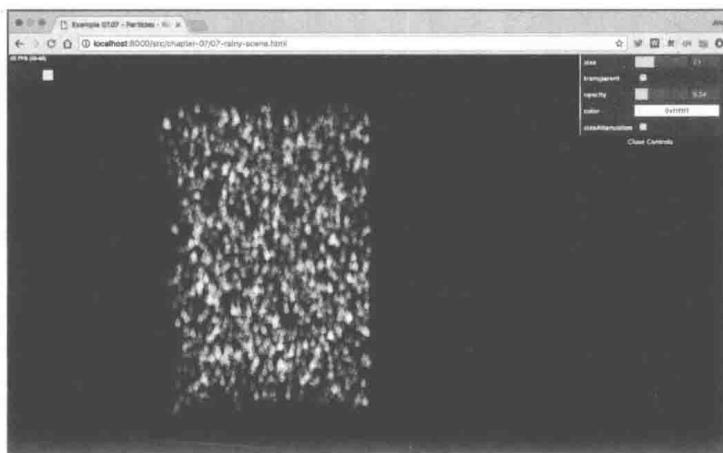


图 7.10

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，我们也不会仅仅只使用单一纹理，而是使用 5 个不同的图片（取自 Three.js 的例子）。同样，我们先来看看如图 7.11 所示的结果（参考 07-snowy-scene.html）。

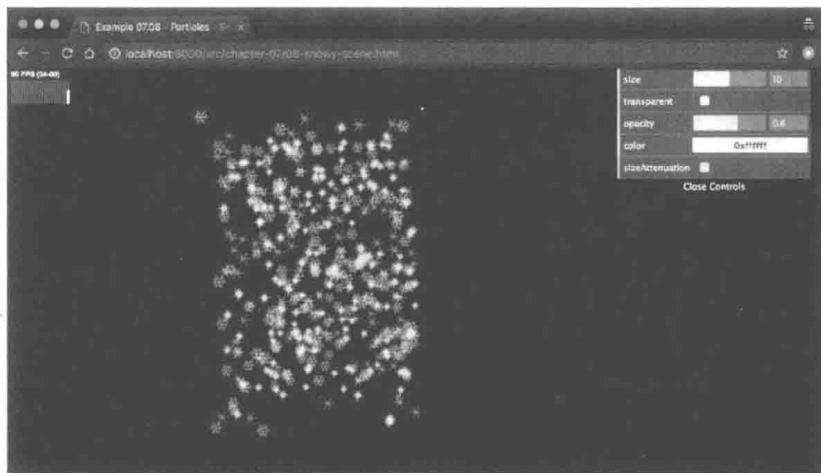


图 7.11

你可以看到我们用多张图片（其中使用透明背景而非黑色背景）作为纹理，而不是一张。你可能想知道我们是如何做到的。你应该还记得一个 THREE.Points 只能有一种材质。如果要使用多个材质，那么只能使用多个 THREE.Points 实例，如以下代码所示：

```
function createPointInstances(size, transparent, opacity, sizeAttenuation,
color) {
    var loader = new THREE.TextureLoader();
    var texture1 =
        loader.load("../assets/textures/particles/snowflake1_t.png");
    var texture2 =
        loader.load("../assets/textures/particles/snowflake2_t.png");
    var texture3 =
        loader.load("../assets/textures/particles/snowflake3_t.png");
    var texture4 =
        loader.load("../assets/textures/particles/snowflake4_t.png");
    var texture5 =
        loader.load("../assets/textures/particles/snowflake5_t.png");

    scene.add(createPoints("system1", texture1, size, transparent, opacity,
        sizeAttenuation, color));
    scene.add(createPoints("system2", texture2, size, transparent, opacity,
        sizeAttenuation, color));
    scene.add(createPoints("system3", texture3, size, transparent, opacity,
        sizeAttenuation, color));
    scene.add(createPoints("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 THREE.Points 的 createPointInstances 函数。该函数代码如下所示：

```

function createPointCloud(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.PointsMaterial({
        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 < 150; 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.Points(geom, material);
    system.name = name;
    return system;
}

```

在这个函数里我们首先要做的是给被渲染的粒子的特定纹理指定颜色。做法是随机改变传入的颜色的 lightness (亮度) 值。接下来是像以前一样创建材质。这里唯一的改变是将 depthWrite 属性设置为 false。这个属性决定这个对象是否影响 WebGL 的深度缓存。将它设置为 false，可以保证在各个的位置上渲染的雪花之间不会互相影响。如果这个属性不设置为 false，那么当一个粒子在另一个 THREE.Points 对象的粒子前面时，你可能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。现在我们可以渲染循环里更新每个 THREE.Points 对象中的粒子，代码如下所示：

```

scene.children.forEach(function (child) {
    if (child instanceof THREE.Points) {
        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;
  });
}
);

```

通过这种方式，我们就可以为不同的粒子赋予不同的材质。但是这种方法有一些局限：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。如果你有一组数量不多的不同样式的粒子，那么最好使用我们在本章开头展示的 THREE.Sprite 对象。

## 7.5 使用精灵贴图

在本章的开头，我们在 THREE.CanvasRenderer 和 THREE.WebGLRenderer 中使用一个 THREE.Sprite 对象渲染单个粒子。这些精灵在三维世界的某个地方，方向永远面对着摄像机并且它们的尺寸取决于离摄像机的距离（这有时也被称为广告牌）。我们将在本节展示另一种使用 THREE.Sprite 对象的方法。我们将展示如何使用 THREE.Sprite 和额外的 THREE.OrthographicCamera 为你的三维内容创建一个类似于平视显示器（head-up display，简称 HUD）的层。我们还会展示如何使用精灵贴图选择 THREE.Sprite 对象的图像。

例如，我们将会创建一个简单的 THREE.Sprite 对象，从左到右滑过屏幕。作为背景，我们将会渲染一个带有移动摄像机的三维场景，用来说明 THREE.Sprite 的移动是独立于摄像机的。图 7.12 所示的就是我们在第一个例子（09-sprites.html）里创建的结果。

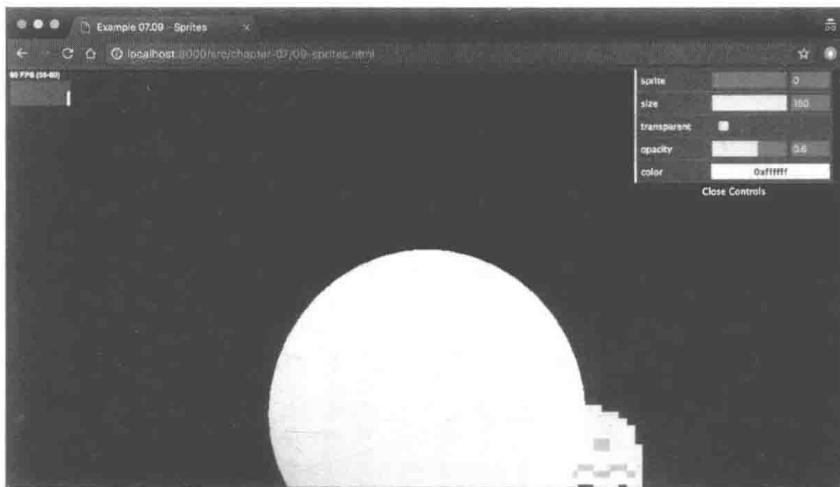


图 7.12

如果你在浏览器里打开这个例子，就会看到一个类似吃豆人（Pac-Man）的精灵绕着屏幕移动，并在碰到右边缘时改变颜色和外形。我们先来看如何创建 THREE.OrthographicCamera 对象和一个单独的场景来渲染 THREE.Sprite：

```
var sceneOrtho = new THREE.Scene();
var cameraOrtho = new THREE.OrthographicCamera( 0, window.innerWidth,
window.innerHeight, 0, -10, 10 );
```

接下来我们看一下 THREE.Sprite 对象的构造方法以及精灵可接受的各种形状是如何加载的：

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

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

    // we have 1 row, with five sprites
    spriteMaterial.map.offset = new THREE.Vector2(0.2 * spriteNumber, 0);
    spriteMaterial.map.repeat = new THREE.Vector2(1 / 5, 1);
    spriteMaterial.blending = THREE.AdditiveBlending;
    // make sure the object is always rendered at the front
    spriteMaterial.depthTest = false;
    var sprite = new THREE.Sprite(spriteMaterial);
    sprite.scale.set(size, size, size);
    sprite.position.set(100, 50, -10);
    sprite.velocityX = 5;

    sceneOrtho.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的纹理（也称为精灵贴图），而不是为每个精灵各自加载一张图片。纹理如图 7.13 所示。

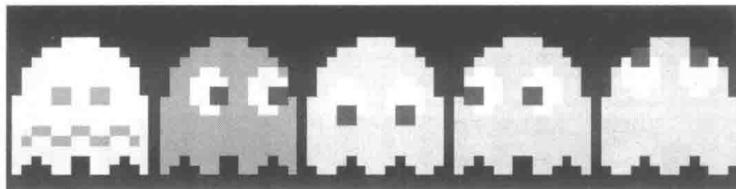


图 7.13

通过 map.offset 和 map.repeat 属性，我们可以正确地选择要显示在屏幕上的精灵。通过 map.offset 属性可以决定纹理在 x 轴（参数 u）和 y 轴（参数 v）上的偏移量。这些属性的缩放比例是 0~1。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）设置为 0.4。由于这里只有一行图片，所以不必改变 v 偏移（y 轴）。如果我们只设置该属性，那么显示在屏幕上的纹理是第 3、4、5 个精灵压缩在一起。要想只显示其中一个，我们还需要

放大。为此，我们可以将 `u-value` 的 `map.repeat` 属性设置为 `1/5`。这意味着我们会放大纹理（只针对 `x` 轴），只显示其中的 20%，也就是一个精灵。

最后一步是更新 `render` 函数，代码如下所示：

```
webGLRenderer.render(scene, camera);
webGLRenderer.autoClear = false;
webGLRenderer.render(sceneOrtho, cameraOrtho);
```

首先，我们通过普通摄像机和移动的球体来渲染场景，然后渲染包含精灵的场景。



要注意的是，我们需要将 `WebGLRenderer` 的 `autoClear` 属性设置为 `false`。如果不这样做，`Three.js` 会在渲染粒子前清空场景，那么球体将不会显示。

表 7.3 展示了我们之前用过的 `THREE.SpriteMaterial` 对象的所有属性的概述。

表 7.3

名 称	描 述
<code>color</code>	精灵的颜色
<code>map</code>	精灵所用的纹理。它可以是一组精灵（sprite sheet），就像本节例子中那样
<code>sizeAnnutation</code>	如果该属性设置为 <code>false</code> ，那么距离摄像机的远近不影响精灵的大小。默认值为 <code>true</code>
<code>opacity</code>	该属性设置精灵的不透明度。默认值为 <code>1</code> （不透明）
<code>blending</code>	该属性指定渲染精灵时所用的融合模式
<code>fog</code>	该属性决定精灵是否受场景中的雾化效果影响。默认值为 <code>true</code>

你还可以在这个材质上设置 `depthTest` 和 `depthWrite` 属性。更多关于这些属性的信息可以参考第 4 章。当然，我们还可以在三维空间中定位粒子时使用精灵贴图（就像本章开头所做的一样）。我们也为这个用途创建了一个示例（`10-sprites-3D.html`），其输出结果如图 7.14 所示。

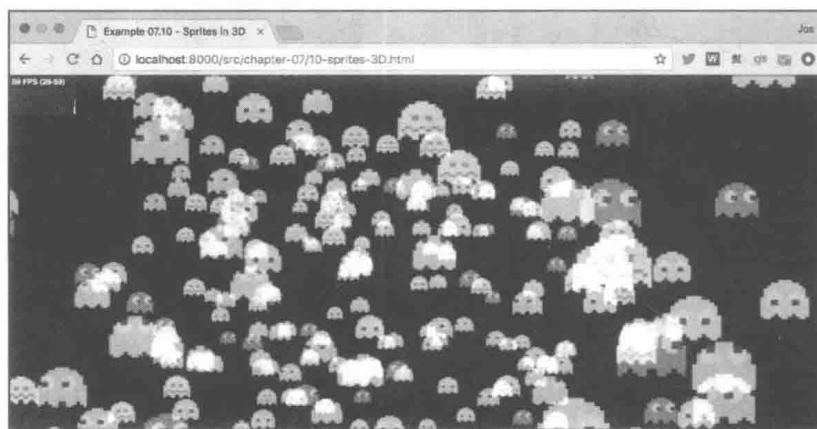


图 7.14

通过我们在表 7.3 中看到的属性，可以很容易地创建上图所示效果，代码如下所示：

```
function createSprites() {
    group = new THREE.Object3D();
    var range = 200;
    for (var i = 0; i < 400; i++) {
        group.add(createSprite(10, false, 0.6, 0xffffffff, i % 5, range));
    }
    scene.add(group);
}

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

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

    // we have 1 row, with five sprites
    spriteMaterial.map.offset = new THREE.Vector2(0.2 * spriteNumber, 0);
    spriteMaterial.map.repeat = new THREE.Vector2(1 / 5, 1);
    spriteMaterial.depthTest = false;

    spriteMaterial.blending = THREE.AdditiveBlending;

    var sprite = new THREE.Sprite(spriteMaterial);
    sprite.scale.set(size, size, size);
    sprite.position.set(
        Math.random() * range - range / 2,
        Math.random() * range - range / 2,
        Math.random() * range - range / 2);

    return sprite;
}
```

在这个例子中，我们使用之前所示的精灵贴图图片创建了 400 个精灵。你可能已经知道并理解这里用到的大部分属性和概念。由于我们已将独立的精灵添加到了一个组中，所以旋转它们是非常容易的，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和点云。但还有一个有趣的选项是从已有的几何体中创建 THREE.Points。

## 7.6 从高级几何体创建 THREE.Points

正如你所记得的，THREE.Points 基于几何体的顶点来渲染每个粒子。这也就是说如果我们提供一个复杂的几何体（比如环状扭结或者管），我们就可以基于这个几何体的顶点创建出一个 THREE.Points 对象。在这一节里，我们将会创建一个环状扭结（类似我们在前一

章所看到的），然后将它渲染成 THREE.Points。

我们在前一章已经描述过环状扭结，所以在这里不再涉及过多的细节。我们使用跟前一章完全相同的代码，还添加了一个菜单选项，你可以用它将这个网格转换为 THREE.Points。你可以在本章的示例代码中找到这个例子（11-create-particle-system-from-model.html）。其输出效果如图 7.15 所示。

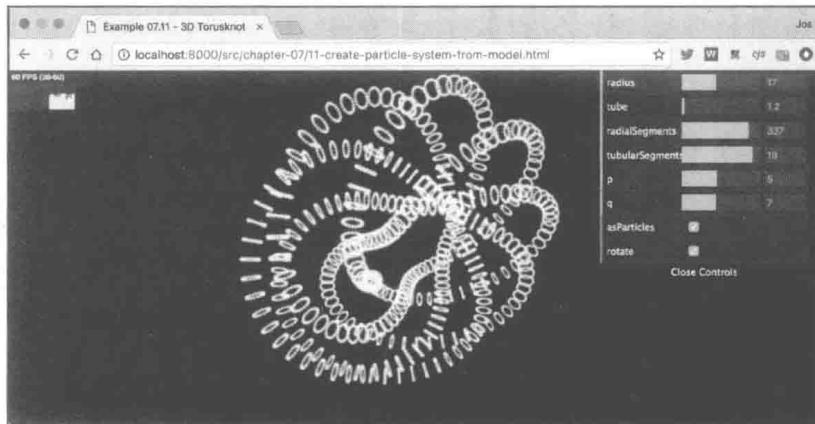


图 7.15

如上图所示，每一个用来生成环状扭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。我们只需看一下创建材质和粒子系统的代码，其他属性我们已经在本章讨论过了：

```
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 createPoints(geom) {
```

```
var material = new THREE.PointsMaterial({  
    color: 0xffffffff,  
    size: 3,  
    transparent: true,  
    blending: THREE.AdditiveBlending,  
    map: generateSprite(),  
    depthWrite: false // instead of sortParticles  
});  
  
var cloud = new THREE.Points(geom, material);  
return cloud;  
}  
  
// use it like this  
var geom = new THREE.TorusKnotGeometry(...);  
var knot = createPoints(geom);
```

在这段代码里，你可以看到两个函数：createPoints() 和 generateSprite()。在第一个函数里，我们直接从指定的几何体（在本例中是一个环状扭结）创建了一个简单的 THREE.Points 对象，然后通过 generateSprite() 函数将粒子的纹理（map 属性）设置成发光点（由 HTML5 画布元素生成）。发光点如图 7.16 所示。



图 7.16

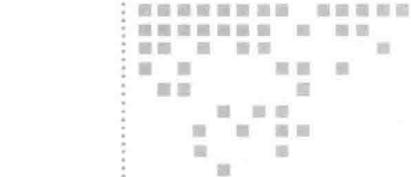
## 7.7 总结

这就是本章所有的内容了。我们解释了什么是粒子、精灵，以及如何使用可用的材质来样式化这些对象。

本章讨论了如何通过 THREE.CanvasRenderer 和 THREE.WebGLRenderer 直接使用 THREE.Sprite 对象。但是，如果你想创建大量的粒子，应该使用 THREE.Points，这样所有的粒子将共享同一个材质，并且单个粒子唯一可以改变的属性是颜色，这通过将材质的 vertexColors 属性设置为 THREE.VertexColors 并为创建 THREE.Points 的 THREE.Geometry 的 colors 数组提供一个颜色值来实现。我们还展示了通过改变粒子的位置让粒子很容易地动起来。单个的 THREE.Sprite 实例和用于创建 THREE.Points 的几何体的顶点都可以如此使用。

到目前为止，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（如球体和方块）来说非常有效，但当你想要创建复杂的三维模型时，这不是最好的方法。通常使用三维建模工具（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由这些三维建模工具所创建的模型。

# 创建、加载高级网格和几何体



在这一章，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 章和第 6 章向你展示了如何使用 Three.js 自带的对象来创建一些高级几何体。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。
- 从外部加载。

我们将从组合和合并方法开法。通过这种方法，我们使用标准的 Three.js 组合函数和 THREE.Geometry.merge() 函数来创建新对象。

## 8.1 几何体组合与合并

我们将在本节介绍 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为一个网格。我们先从对象组合开始。

### 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种材质。但是实际上它是一个包含多个网格的组。

创建组非常简单。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，而所有的子对象都将会受到影响。让我们来看一个例子（01-grouping.html），如图 8.1 所示。

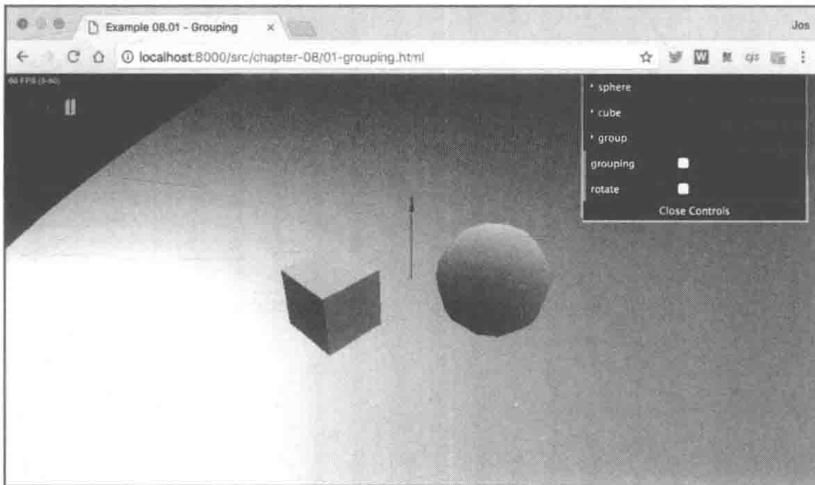


图 8.1

在这个例子里，你可以使用控制菜单来移动球体和方块。如果选中了 `rotate` 选项，你会看到这两个网格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，这两个对象并不是被直接添加到场景中的，而是添加到一个组中。代码如下所示：

```
sphere = createMesh(new THREE.SphereGeometry(5, 10, 10));
cube = createMesh(new THREE.BoxGeometry(6, 6, 6));

group = new THREE.Group();
group.add(sphere);
group.add(cube);

scene.add(group);
```

在上面的代码中，我们创建了一个 `THREE.Group`（组）类的对象。该类与 `THREE.Mesh` 和 `THREE.Scene` 类共同的基类 `THREE.Object3D` 非常接近，唯一的不同是它本身并不含有任何可渲染的数据。

在这个例子中我们通过组对象的 `add` 函数将 `sphere` 和 `cube` 对象添加到组中，然后再将组对象整体添加到场景对象 `scene` 中。被添加到组对象中的三维物体，其本身仍然可以被单独移动、缩放和旋转。此外还可以对整个组对象进行相同的操作。看一下组菜单，你会发现 `position`（位置）和 `scale`（缩放）选项，可以使用这些选项来缩放和移动整个组。

这里要注意的一点是，当三维物体被添加到组对象中后，它们自身的位置、缩放和旋转参数便都是相对于组对象的对应参数的<sup>⊖</sup>。

缩放和移动都很直观。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，而是绕其中心旋转整个组（在我们的例子里，是绕着 `group` 对象的中心旋转整个组）。在这个例子里，我们在组的中心使用 `THREE.ArrowHelper` 对象放置了一个箭头，用来指示旋转点：

<sup>⊖</sup> 例如某组对象的位置参数为  $(10, 0, 0)$ ，组内某物体的位置参数为  $(5, 0, 0)$ ，当该组对象被直接添加到场景中时，该组内物体在场景中的位置为  $(15, 0, 0)$ 。缩放和旋转参数与位置参数类似。——译者注

```
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。你将会看到球体和方块一起绕着组的中心（箭头所指）旋转，如图 8.2 所示。

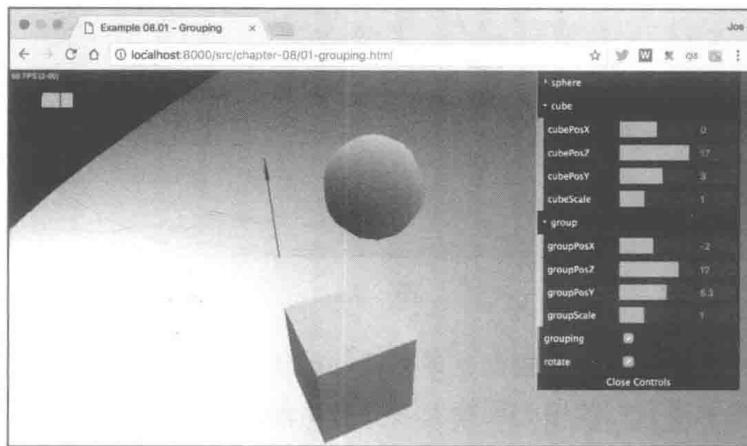


图 8.2

使用组的时候，你依然可以引用、修改和定位每一个单独的几何体。唯一需要记住的是：所有的定位、旋转和变形都是相对父对象的。下一节我们将看一下合并，可以将多个几何体组合并最终生成一个 THREE.Geometry 对象。

### 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地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。使用组，每个对象还是独立的，仍然需要对它们分别进行处理和渲染。通过 THREE.Geometry.merge() 函数，你可以将多个几何体合并起来创建一个联合体。在下面的例子里，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量，点击 redraw 按钮可以重新绘制场景。根据你所使用的硬件，当方块的数量达到一定量时，性能就会开始下降。你可以从图 8.3 中看到，在我的环境中，性能在 6000 个对象左右时开始下降，刷新速度从通常的 60fps 下降到 40fps。

正如你所看到的，场景中能够添加的网格数量有一个上限。一般来讲你并不需要那么多网格，但是在创建某些游戏（例如 Minecraft）或者高级显示效果时，你可能需要管理大量的独立网格。而使用 THREE.Geometry.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 combine 复选框。当选中该选项时，我们将所有的方块合并成一个 THREE.Geometry，然后将它添加到场景中，如图 8.4 所示。

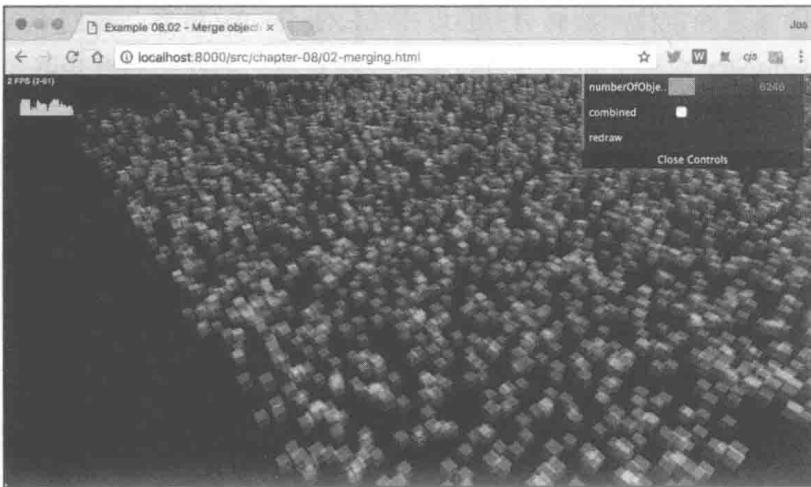


图 8.3



图 8.4

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++) {
    var cubeMesh = addcube();
    cubeMesh.updateMatrix();
    geometry.merge(cubeMesh.geometry, cubeMesh.matrix);
}
scene.add(new THREE.Mesh(geometry, cubeMaterial));
```

在这段代码里，`addCube()` 函数返回的是一个 `THREE.Mesh` 对象。在旧版本的 `Three.js` 中，我们可以使用 `THREE.GeometryUtils.merge` 函数将 `THREE.Mesh` 对象合并到 `THREE.Geometry` 对象中。在最新版本中，此功能已被弃用，而改为支持 `THREE.Geometry.merge`

函数。为了确保能正确定位和旋转合并的 THREE.Geometry 对象，我们不仅向 merge 函数提供 THREE.Geometry 对象，还提供该对象的变换矩阵。当我们此矩阵添加到 merge 函数后，那么合并的方块将被正确定位。

如此做 20 000 次，我们得到的将是一个几何体，然后将它添加到场景中。如果看过代码，你会发现这个方法有几个缺陷。由于我们得到的是一个几何体，所以你不能为每个几何体单独添加材质。但是，这个问题在某种程度上可以用 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、旋转或缩放某个方块是不可能的（除非你能探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、复杂的几何体。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 从外部资源加载几何体

Three.js 可以读取几种三维文件格式，并从中导入几何体和网格。表 8.1 所列就是 Three.js 支持的文件格式。

表 8.1

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。尽管它不是一种正式的格式，但是它很容易使用，而且当你想要复用复杂的几何体或场景时非常方便
OBJ 或 MTL	OBJ 是一种简单的三维文件格式，由 Wavefront Technologies 创建。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，用 OBJ 文件定义对象的材质 Three.js 还有一个可定制的 OBJ 导出器，叫作 OBJExporter.js，可以用来将 Three.js 中的模型导出到一个 OBJ 文件
Collada	Collada 是一种用基于 XML 的格式定义数字内容的格式。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是 STL 文件 Three.js 还有一个可定制的 STL 导出器，叫作 STLExporter.js，可以用来将 Three.js 中的模型导出到一个 STL 文件
CTM	CTM 是由 openCTM 创建的一种文件格式，可以用来以压缩格式存储三维网格的三角形面片
VTK	VTK 是由 Visualization Toolkit 创建的一种文件格式，用来指定顶点和面。VTK 有两种格式：二进制和基于文本的 ASCII。Three.js 只支持基于 ASCII 的格式
AWD	AWD 是一种用于三维场景的二进制格式，并且通常与 <a href="http://away3d.com/">http://away3d.com/</a> 引擎一起使用。 请注意，此加载程序不支持压缩的 AWD 文件

(续)

格 式	描 述
Assimp	Open asset import library (也称为 Assimp) 是一种导入各种三维模型格式的标准方法，使用此加载程序，由 assimp2json 转换的大量三维格式都可以导入模型，更多详细信息请访问 <a href="https://github.com/acgessler/assimp2json">https://github.com/acgessler/assimp2json</a>
VRML	VRML 是 Virtual Reality Modeling Language 的缩写。这是一种基于文本的格式，允许定义三维对象和世界。它已被 X3D 文件格式取代。Three.js 不支持加载 X3D 模型，但这些模型可以很容易地转换成其他格式。更多信息请访问 <a href="http://www.x3dom.org/?page_id=532#">http://www.x3dom.org/?page_id=532#</a>
Babylon	Babylon 是一个三维 JavaScript 游戏库，它以自己的内部格式存储模型。有关这方面的信息请访问 <a href="http://www.babylonjs.com/">http://www.babylonjs.com/</a>
PDB	这是一种非常特殊的格式，由 Protein Data Bank (蛋白质数据银行) 创建，用于定义蛋白质的形状。Three.js 可以加载并显示用这种格式描述的蛋白质
PLY	这种格式的全称是多边形 (Polygon) 文件格式，通常用来保存三维扫描仪的信息
TDS	Autodesk 3DS 格式。更多详细信息请访问 <a href="https://www.autodesk.com/">https://www.autodesk.com/</a>
3MF	3MF 是 3D 打印的标准格式之一。更多详细信息请访问 3MF Consortium 的官方主页 <a href="https://3mf.io/">https://3mf.io/</a>
AMF	AMF 是另一个 3D 打印的标准格式，但是目前已经停止开发。更多详细信息请访问下面的维基百科页面 <a href="https://en.wikipedia.org/wiki/">https://en.wikipedia.org/wiki/</a>
PlayCanvas	PlayCanvas 是一个非常出色的基于 WebGL 的开源游戏引擎。Three.js 提供的加载器可以帮助加载基于 PlayCanvas 的模型并在场景中使用。PlayCanvas 的官方主页在 <a href="https://playcanvas.com/">https://playcanvas.com/</a>
Draco	Draco 格式可以高效地保存几何体和点云。更多详细信息请访问 Github 主页 <a href="https://github.com/google/draco">https://github.com/google/draco</a>
PRWM	PRWM 的全称为 Packed Raw WebGL Model (WebGL 模型原始数据包)。这是另一个专注于高效存储和解析 3D 几何体的格式。更多详细信息和使用方法请访问 <a href="https://github.com/kchapelier/PRWM">https://github.com/kchapelier/PRWM</a>
GCODE	GCode 是计算机程序与 3D 打印机和 CNC 机器通信的标准模式之一。在打印模型时，计算机程序可以向 3D 打印机发送 GCode 命令来控制打印机。更多详细信息请访问 <a href="https://www.nist.gov/publications/nistrs274ngc-interpreter-version-3?pub_id=823374">https://www.nist.gov/publications/nistrs274ngc-interpreter-version-3?pub_id=823374</a>
NRRD	NRRD 是一种存储体素数据的文件格式。它可以用于保存 CT 扫描数据。更多详细信息请访问 <a href="http://teem.sourceforge.net/nrrd/">http://teem.sourceforge.net/nrrd/</a>
SVG	SVG 是一种矢量图形标准格式。Three.js 的加载器会把 SVG 加载成 THREE.Path 对象。该对象可以被拉伸成为 3D 几何体，或者也可以直接当作 2D 几何体来渲染

在下一章讲解动画时，我们再来看看这个列表中的某些格式（以及另外两种格式，MD2 和 glTF）。现在，我们将从这个列表中的第一种格式——Three.js 的内部格式开始学习。

### 8.2.1 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：用它来保存和加载单个 THREE.Mesh，或者用它来保存和加载整个场景。

### 1. 保存和加载 THREE.Mesh

为了展示保存和加载功能，我们基于 THREE.TorusKnotGeometry 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，创建一个环状扭结，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 来保存。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）。

我们来看一看示例 03-load-save-json-object.html，如图 8.5 所示。

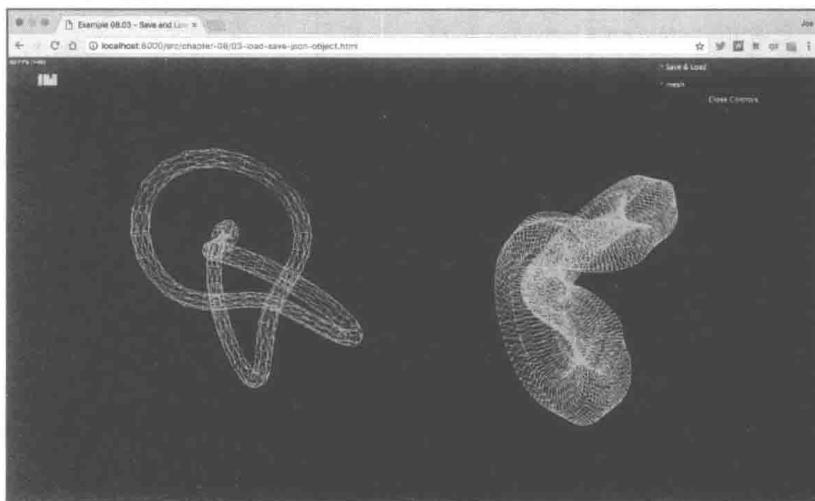


图 8.5

在 Three.js 中导出 JSON 文件非常容易，并且不需要引入额外的库。你需要做的只是将 THREE.Mesh 导出为 JSON，代码如下：

```
var result = knot.toJSON();
localStorage.setItem("json", JSON.stringify(result));
```

在保存前，我们先要用 JSON.stringify 函数将 toJSON 函数解析的结果从一个 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": 4.5,
    "type": "Object",
    "generator": "Object3D.toJSON"
  },
  "geometries": [
    {
      "uuid": "8EEBC72A-C436-4431-89F0-175D3EA0A04E",
      "type": "TorusKnotGeometry",
      "radius": 10,
      "tTube": 1,
      "tTubularSegments": 64,
      "radialSegments": 8,
```

```

    "p": 2,
    "q": 3
  },
],
"materials": [
  {
    "uuid": "9609BC46-EB00-47B3-AD62-3424D12821D9",
    "type": "MeshBasicMaterial",
    "color": 11184810,
    "side": 2,
    "vertexColors": 2,
    "depthFunc": 3,
    "depthTest": true,
    "depthWrite": true,
    "wireframe": true,
    "wireframeLineWidth": 2
  }
],
"object": {
  "uuid": "64B07BE0-8EC0-4CC4-BCA5-0A1DD29E07B2",
  "type": "Mesh",
  "matrix": [
    0.8560666518372149,
    0,
    0.5168654443974957,
    0,
    0,
    0,
    1,
    0,
    0,
    -0.5168654443974957,
    0,
    0.8560666518372149,
    0,
    20,
    0,
    0,
    1
  ],
  "geometry": "8EEBC72A-C436-4431-89F0-175D3EA0A04E",
  "material": "9609BC46-EB00-47B3-AD62-3424D12821D9"
}
}
}

```

正如你所看到的，Three.js 保存了 THREE.Mesh 对象的所有信息。要把这些信息通过 HTML5 的本地存储 API 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 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.ObjectLoader();

  loadedMesh = loader.parse(loadedGeometry);
}

```

```

    loadedMesh.position.x -= 40;
    scene.add(loadedMesh);
}

```

在这里，我们先根据保存里指定的名字（本例中是 json）从本地存储中获取 JSON 数据。为此，我们调用的是由 HTML5 本地存储 API 提供的 `localStorage.getItem` 函数。接下来我们需要将这个字符串转换成 JavaScript 对象（`JSON.parse`），然后将 JSON 对象转换成 `THREE.Mesh` 对象。Three.js 提供了一个叫作 `THREE.ObjectLoader` 的辅助对象，使用它可以将 JSON 转换成 `THREE.Mesh` 对象。在本例中，我们使用的是加载器的 `parse` 方法来直接解析 JSON 字符串。该加载器还提供了一个 `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 `THREE.Mesh`，其他信息都丢失了。如果你想要保存整个场景，包括光源和摄像机，可以使用 `THREE.SceneExporter`。

## 2. 保存和加载场景

如果你想保存整个场景，同样可以使用前一节介绍的保存几何体的方法。你可以在 `04-load-save-json-scene.html` 里找到示例，如图 8.6 所示。

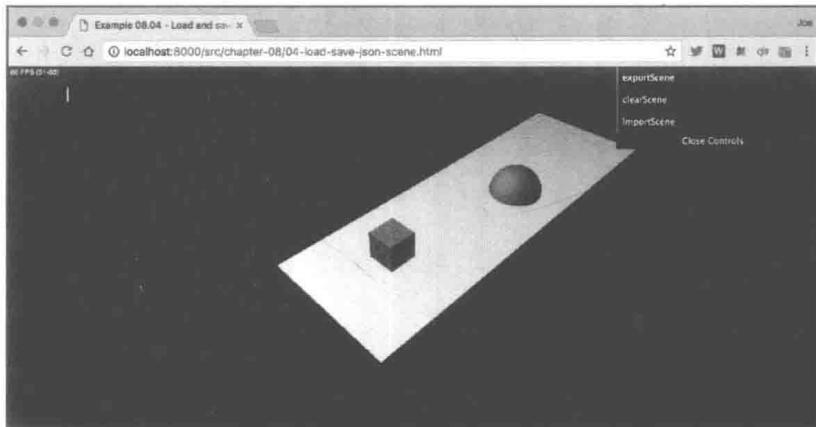


图 8.6

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非常简单：

```
localStorage.setItem('scene', JSON.stringify(scene.toJSON()));
```

这个方法跟我们在前一节所用的方法几乎一模一样，只是这次我们导出的是整个 `THREE.Scene` 而不是某个模型：

```
{
  "metadata": {
```

```
"version": 4.5,
"type": "Object",
"generator": "Object3D.toJSON"
},
"geometries": [
{
  "uuid": "C13443D6-3BE2-4E78-BD8E-C07D5279AA59",
  "type": "PlaneGeometry",
  "width": 60,
  "height": 20,
  "widthSegments": 1,
  "heightSegments": 1
},
***,
],
"materials": [
{
  "uuid": "B57ED003-B65A-480A-81AC-FAA63A912F73",
  "type": "MeshLambertMaterial",
  "color": 16777215,
  "emissive": 0,
  "depthFunc": 3,
  "depthTest": true,
  "depthWrite": true
},
***,
],
"object": {
  "uuid": "245E4001-CB51-47D8-BC35-6D767B68B15F",
  "type": "Scene",
  "matrix": [
    1,
    0,
    0,
    0,
    0,
    1,
    0,
    0,
    0,
    0,
    1,
    0,
    0,
    0,
    0,
    0,
    1
  ],
  "children": [
    {
      "uuid": "12129DB6-48E2-4409-A015-5421F445D9DE",
      "type": "Mesh",
      "matrix": [
        1,
        0,
        0,
        0,
        0,
```

```
    0,
    2.220446049250313e-16,
    -1,
    0,
    0,
    1,
    2.220446049250313e-16,
    0,
    15,
    0,
    0,
    1
  ],
  "geometry": "C13443D6-3BE2-4E78-BD8E-C07D5279AA59",
  "material": "B57ED003-B65A-480A-81AC-FAA63A912F73"
},
...
{
  "uuid": "6CC1F6E5-86CF-41FB-9F33-D38870F009C7",
  "type": "PointLight",
  "matrix": [
    1,
    0,
    0,
    0,
    0,
    1,
    0,
    0,
    0,
    0,
    1,
    0,
    0,
    0,
    0,
    1,
    0,
    -40,
    60,
    -10,
    1
  ],
  "color": 16777215,
  "intensity": 1,
  "distance": 0,
  "decay": 1,
  "shadow": {
    "camera": {
      "uuid": "65C7746B-8896-4469-A2C5-97C9CDD5D483",
      "type": "PerspectiveCamera",
      "fov": 90,
      "zoom": 1,
      "near": 0.5,
      "far": 500,
      "focus": 10,
      "aspect": 1,
      "filmGauge": 35,
      "filmOffset": 0
    }
  }
}
```

]  
}  
}

当再次加载这个 JSON 时，Three.js 只是按照其导出时的定义重新创建对象而已。加载场景的代码如下所示：

```
var loadedSceneAsJson = JSON.parse(json);
var loader = new THREE.ObjectLoader();
var scene = loader.parse(loadedSceneAsJson);
```



你可能会看到一些基于旧版 Three.js 的开发程序在使用 THREE.SceneLoader 对象。

在新版 Three.js 中，这个老方法已经被替换为上面的新方法。

有很多三维软件可以用来创建复杂的网格。其中有一个流行的开源软件叫作 Blender ([www.blender.org](http://www.blender.org))。Three.js 库有一个 Blender (以及 Maya 和 3D Studio Max) 导出器，可以直接将文件导出为 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，并在 Three.js 里展示出来。

### 8.2.2 使用 Blender

在开始配置之前，先来看一下我们想要达到的效果。在图 8.7 里你会看到一个简单的 Blender 模型，我们用一个 Three.js 插件将它导出，并在 Three.js 中用 THREE.JSONLoader 将它导入。

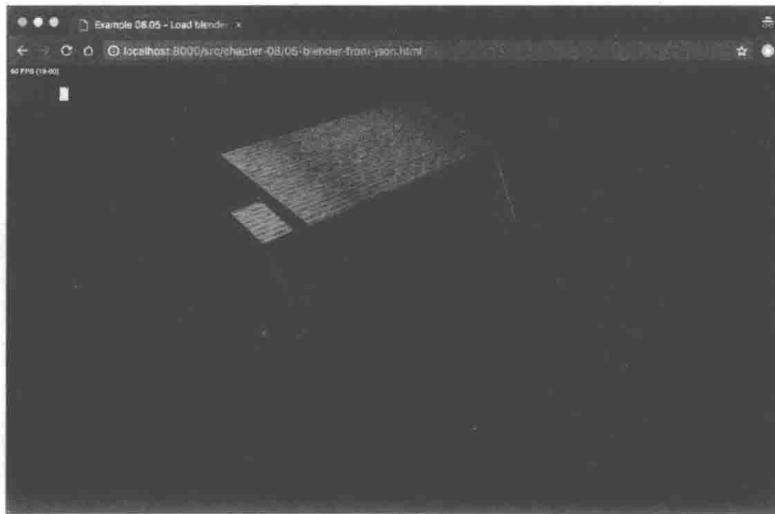


图 8.7

## 1. 在 Blender 中安装 Three.js 导出器

要想从 Blender 中导出 Three.js 模型，首先要将 Three.js 导出器添加到 Blender 中。下面所列的步骤是针对 Mac OS X 系统的，但对于 Windows 和 Linux 系统来讲也基本一样。你可以从 [www.blender.org](http://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，使用终端在 Blender 的安装目录下找到 addons 文件夹，如图 8.8 所示。

```
6.jos@Joss-MBP-3: ~/Downloads/blender-2.79-macOS-10.6/blender.app/Contents/Resources
→ addons ls -l | head
total 4176
drwxr-xr-x@ 20 jos  staff      680 Sep 11  2017 add_advanced_objects_menu
drwxr-xr-x@ 10 jos  staff      340 Sep 11  2017 add_advanced_objects_panels
drwxr-xr-x@ 13 jos  staff     442 Sep 11  2017 add_curve_extra_objects
-rw-r--r--@  1 jos  staff   26581 Aug 18  2017 add_curve_ivygen.py
drwxr-xr-x@  5 jos  staff      170 Sep 11  2017 add_curve_sapling
drwxr-xr-x@  5 jos  staff      170 Sep 11  2017 add_mesh_BoltFactory
drwxr-xr-x@ 26 jos  staff     884 Sep 11  2017 add_mesh_extra_objects
-rw-r--r--@  1 jos  staff  13097 Aug 18  2017 animation_add_corrective_shape_key.py
-rw-r--r--@  1 jos  staff  17700 Aug 18  2017 animation_animall.py
→ addons
```

图 8.8

- 在 Mac 系统上，其路径是 ./blender.app/Contents/MacOS/2.79/scripts/addons。
- 在 Windows 系统中，这个目录可以在路径 C:/Users/USERNAME/AppData/Roaming/BlenderFoundation/Blender/2.79/scripts/addons 下找到。
- 在 Linux 系统中，这个目录的路径是 /home/USERNAME/.config/blender/2.79/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，你会看到如下文件夹：utils/exporters/blender/addons/。在这个目录里有一个子目录叫作 io\_three。将这个目录复制到 Blender 安装目录里的 addons 文件夹下。

现在我们要做的只是启动 Blender，激活导出器。在 Blender 里，打开 Blender User Preferences (File | User Preferences)。在弹出的窗口中找到 Add-ons 标签页，然后在搜索框里输入 three。结果如图 8.9 所示。

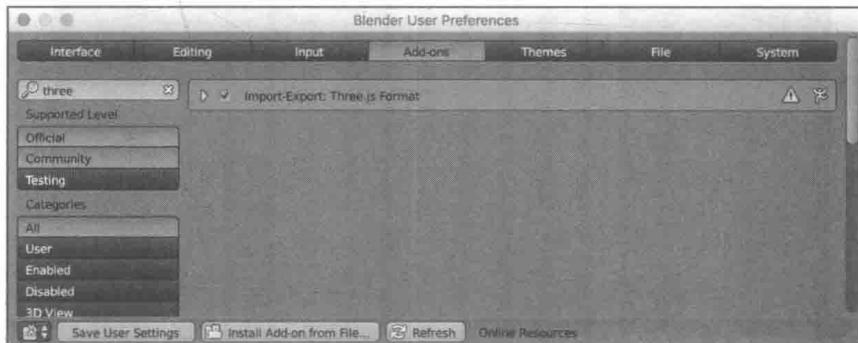


图 8.9

现在虽然已经找到了 Three.js 插件，但是它还没有被激活。选择右边的小复选框，Three.js 导出器就可以激活了。最后来检查一下是不是所有设置都可以正确地工作。打开 File | Export 菜单项，你会看到 Three.js 出现在 Export 选项中，如图 8.10 所示。



图 8.10

安装好插件后，我们就可以加载第一个模型了。

## 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 `hjmediastudios_house_dist.blend`，你可以在本书附带的源码中找到它。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 中。点击 `File | Open` 菜单，找到包含 `hjmediastudios_house_dist.blend` 文件的文件夹。选中这个文件，然后点击 `Open`，其结果如图 8.11 所示。

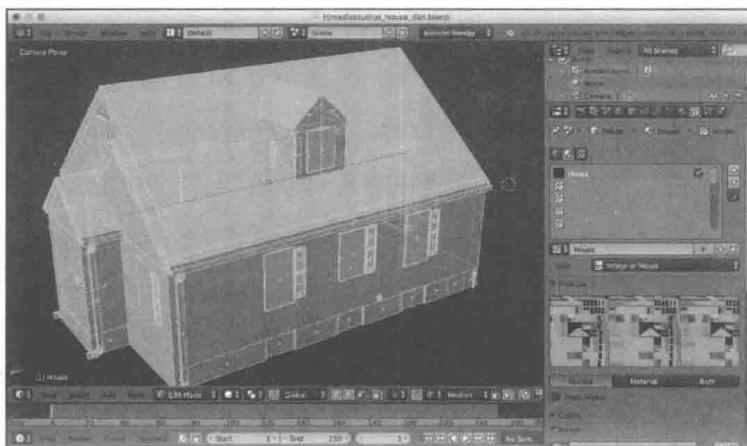


图 8.11

将这个模型导出为 `Three.js` 的 JSON 格式非常简单。在 `File` 菜单下找到 `Export | Three.js`，输入导出文件的文件名，然后选择 `Export Three.js`。在弹出的对话框中会显示一系列选项，用于精细控制导出效果。这里我们保持所有选项的默认值，并且确保已经勾选了 `Textures`

和 Export textures 选项。最后在指定的文件夹里会生成下面两个文件：

- `hjmediastudios_house_dist.json`: 模型的 Three.js 格式 JSON 文件。
- `Lightmap.png`: 导出的纹理图片，将用于为模型创建材质。

该文件的部分内容如下所示：

```
...
    "metadata": {
        "version": 3,
        "type": "Geometry",
        "uvs": 1,
        "faces": 782,
        "vertices": 1036,
        "materials": 1,
        "normals": 280,
        "generator": "io_three"
    }
...
}
```

仔细观察导出的 JSON 文件，可以找到一个材质定义。如果前面的操作均成功完成，则会看到这个材质引用了 `Lightmap.png` 纹理图片文件：

```
"materials": [
    {
        "DbgColor": 15658734,
        "transparent": false,
        "mapDiffuseAnisotropy": 1,
        "mapDiffuseWrap": ["repeat", "repeat"],
        "DbgIndex": 0,
        "wireframe": false,
        "mapDiffuse": "Lightmap.png",
        "depthTest": true,
        "DbgName": "House",
        "colorDiffuse": [0.64, 0.64, 0.64],
        "depthWrite": true,
        "blending": 1,
        "mapDiffuseRepeat": [1, 1],
        "visible": true,
        "colorEmissive": [0, 0, 0],
        "shading": "lambert",
        "opacity": 1,
        "doubleSided": true
    }
]
```

从上面 JSON 文件片段可见该材质引用了 `Lightmap.png` 纹理图片文件。到此我们已经可以在 Three.js 中加载该导出模型了。加载模型的代码如下所示：

```
var loader = new THREE.JSONLoader();
loader.load('../assets/models/house/house.json', function (geometry, mat) {
    var mesh = new THREE.Mesh(geometry, mat[0]);
    mesh.castShadow = true;
    mesh.receiveShadow = true;

    // call the default render loop.
    loaderScene.render(mesh, camera);
});
```

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。在这个函数里我们指定要加载的 URL（指向导出的 JSON 文件），并且指定加载结束后的回调函数。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 参数包含一组材质对象。我们知道只有一种材质，所以在创建 THREE.Mesh 实例时，我们直接引用该材质。请注意在本例以及其他实验加载器的示例程序中，我们将使用 loaderScene.render 函数去绘制模型。这是一个自定义的函数，它可以将加载获得的模型绘制在一个具有少量光源并支持鼠标控制的简单场景中。如果打开示例 05-blender-from-json.html，你就会看到我们刚刚从 Blender 导出的房子。

使用 Three.js 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 也可以导出其中的几种格式。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，并展示一个用于 OBJ 和 MTL 格式的基于 Blender 的示例。

## 8.3 导入三维格式文件

在本章开头，我们列出了几种 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 examples/js/loaders 目录下的 Three.js 发布包中找到这些文件。

### 8.3.1 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，而 MTL 文件定义所用的材质。OBJ 和 MTL 都是基于文本的格式。某个 OBJ 文件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 的 JSON 格式文件。在 Three.js 里提供两种不同的加载器。如果你只想加载几何体，可以使用 OBJLoader。示例 06-load-obj.html 里使用的就是这个加载器。其结果如图 8.12 所示。

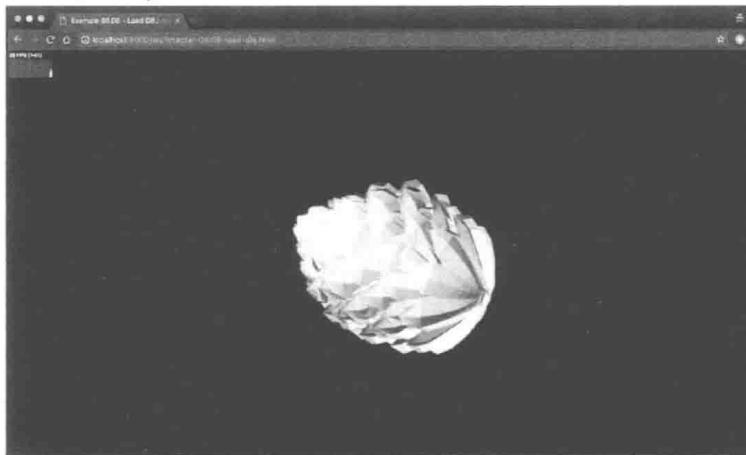


图 8.12

要在 Three.js 里导入它，你必须包含 OBJLoader JavaScript 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/pinecone.obj', function (mesh)
{
    var material = new THREE.MeshLambertMaterial({
        color: 0x5C3A21
    });

    // loadedMesh is a group of meshes. For
    // each mesh set the material, and compute the information
    // three.js needs for rendering.
    mesh.children.forEach(function (child) {
        child.material = material;
        child.geometry.computeVertexNormals();
        child.geometry.computeFaceNormals();
    });

    mesh.scale.set(120, 120, 120)

    // call the default render loop.
    loaderScene.render(mesh, camera);
});
```

在这段代码里，我们使用 OBJLoader 从一个 URL 加载模型。一旦模型加载成功，我们提供的回调函数就会被调用，然后我们把这个模型添加到场景中。



一般来讲，最好先在回调函数里将响应信息打印到控制台（console），以了解加载的对象是如何构建的。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，放置和应用正确的材质就会简单很多。同时还要检查一下几个顶点的位置，以便决定是否需要缩放模型，以及在何处放置摄像机。在本例中，我们还调用了 computeFaceNormals() 和 computeVertexNormals，这是确保正确渲染使用的材质（THREE.MeshLambertMaterial）所必需的。

下一个例子（07-load-obj-mtl.html）使用 OBJLoader 和 MTLLoader 加载模型并直接赋予材质。结果如图 8.13 所示。

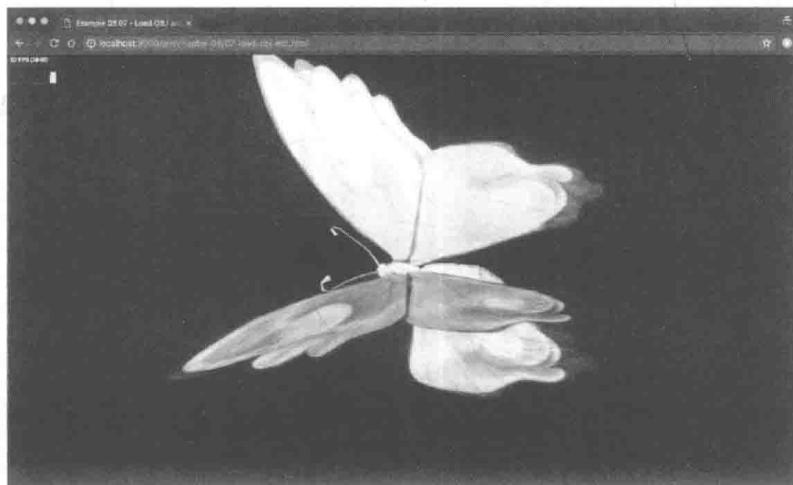


图 8.13

首先我们要在页面中添加正确的加载器：

```
<script type="text/javascript" charset="UTF-8">
<script type="text/javascript" charset="UTF-8">
<script type="text/javascript" charset="UTF-8">
```

然后我们就可以像下面这样从 OBJ 文件和 MTL 文件加载模型：

```
var mtlLoader = new THREE.MTLLoader();
mtlLoader.setPath("../assets/models/butterfly/");
mtlLoader.load('butterfly.mtl', function (materials) {
    materials.preload();

    var objLoader = new THREE.OBJLoader();
    objLoader.setMaterials(materials);
    objLoader.load('../assets/models/butterfly/butterfly.obj', function
(object) {
```

```

// move wings to more horizontal position
[0, 2, 4, 6].forEach(function (i) {
    object.children[i].rotation.z = 0.3 * Math.PI
});

[1, 3, 5, 7].forEach(function (i) {
    object.children[i].rotation.z = -0.3 * Math.PI
});

// configure the wings,
var wing2 = object.children[5];
var wing1 = object.children[4];

wing1.material.opacity = 0.9;
wing1.material.transparent = true;
wing1.material.depthTest = false;
wing1.material.side = THREE.DoubleSide;

wing2.material.opacity = 0.9;
wing2.material.depthTest = false;
wing2.material.transparent = true;
wing2.material.side = THREE.DoubleSide;

object.scale.set(140, 140, 140);
mesh = object;

object.rotation.x = 0.2;
object.rotation.y = -1.3;

loaderScene.render(mesh, camera);
});
});
}
);

```

在看代码之前首先要说明的是：当收到 OBJ、MTL 文件以及所需的纹理文件时，必须看一下 MTL 文件是如何引用纹理的。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。代码与我们在 THREE.ObjLoader 看到的代码没什么不同。我们首先需要用 MTLLoader 加载 MTL 材质文件，并通过 setMaterials 函数将加载的材质对象指定给 OBJLoader 对象，然后再使用 OBJLoader 加载模型文件。本例中使用的是一个复杂模型，所以我们在回调函数中设置了一些特定属性来修复一些渲染问题：

- 由于源文件中的透明度设置得不正确，所以看不到翅膀。为了修复这个问题，我们需要自己设置 opacity 和 transparent 属性。
- 默认情况下，Three.js 只渲染模型表面的正面。因为我们要从两侧看到翅膀，所以需要将 side 属性设置为 THREE.DoubleSide 值。
- 当画面中蝴蝶的两只翅膀在画面中处于重叠位置时，会导致渲染效果不太正确。我们通过将 depthTest 属性设置为 false 来修复它。这对性能有轻微的影响，但通常可以解决一些奇怪的渲染伪像问题。

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。但你可能还需要微调一些材质属性。

### 8.3.2 加载 Collada 模型

Collada 模型（文件扩展名为 .dae）是另外一种非常通用的、用来定义场景和模型（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 模型的步骤和加载 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，我们加载如图 8.14 所示的模型。

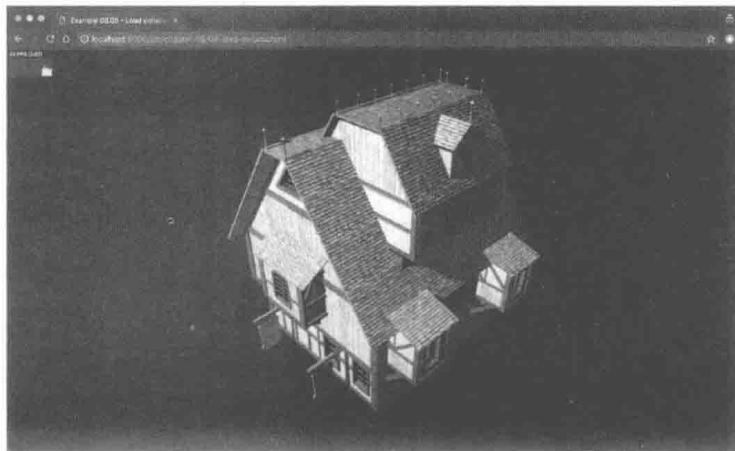


图 8.14

加载卡车模型的代码同样也很简单：

```
// load the model
var loader = new THREE.ColladaLoader();
loader.load("../assets/models/medieval/Medieval_building.DAE",
function (result) {
    var sceneGroup = result.scene;

    sceneGroup.children.forEach(function (child) {
        if (child instanceof THREE.Mesh) {
            child.receiveShadow = true;
            child.castShadow = true;
        } else {
            // remove any lighting sources from the model
            sceneGroup.remove(child);
        }
    });

    // correctly scale and position the model
    sceneGroup.rotation.z = 0.5 * Math.PI;
    sceneGroup.scale.set(8, 8, 8);
    // call the default render loop.
    loaderScene.render(sceneGroup, camera);
});
```

主要区别是返回到回调函数的结果 (result) 对象。这个 result 对象具有以下结构：

```
var result = {
  scene: scene,
  animations: [...],
  kinematic: {...},
  library: {...},
};
```

在本例中，我们感兴趣的是在 scene 参数里面包含了哪些对象。为了快捷地看一下 scene 所包含的对象，将 scene 对象的内容打印到控制台一般会是个好主意。在这里，我们将去除所有非 THREE.Mesh 对象，同时启用所有 THREE.Mesh 对象的阴影功能。

正如你所看到的，对于大多数包含材质的复杂模型来说，为了得到想要的结果，通常需要采取一些额外的步骤。仔细看看材质如何配置（使用 `console.log()`）或将材质替换为一些测试材质，通常很容易找到问题所在。

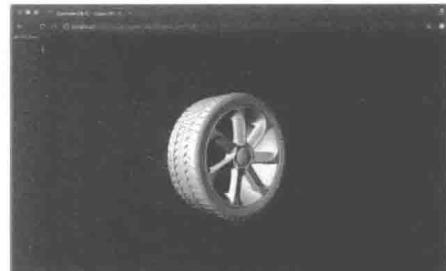
### 8.3.3 从其他格式的文件中加载模型

下面几种文件格式我们只会快速浏览一下，因为它们都遵从相同的原则：

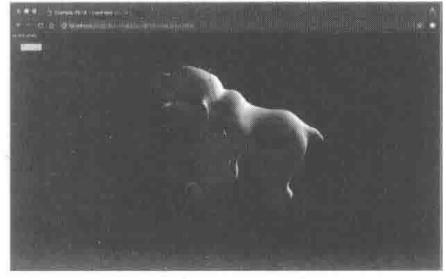
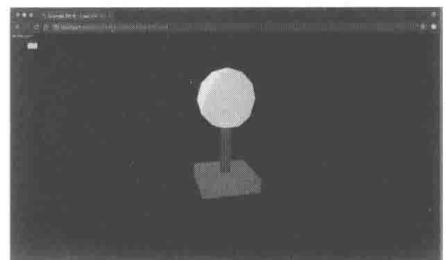
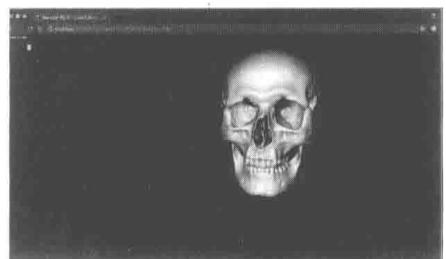
- (1) 在网页中包含 [ 格式名称 ]Loader.js 文件。
- (2) 使用 [ 格式名称 ]Loader.load() 函数从 URL 中加载。
- (3) 检查一下传递给回调函数的返回结果，并对它进行渲染。

我们给每种格式分别提供了一个示例，具体见表 8.2。

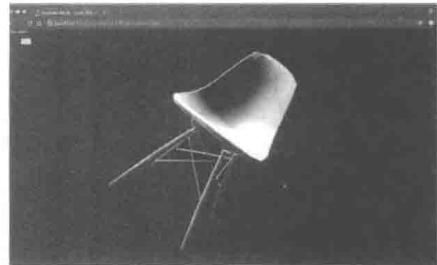
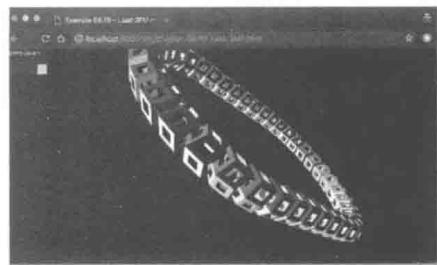
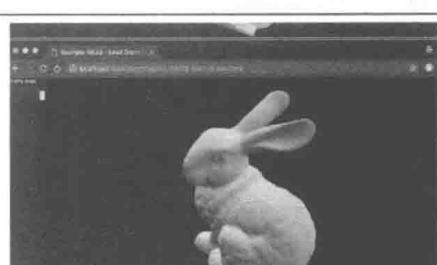
表 8.2

格 式	示 例	屏 幕 截 图
STL	09-load-STL.html	
CTM	10-load-CTM.html	

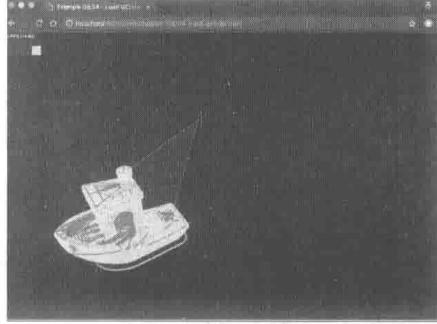
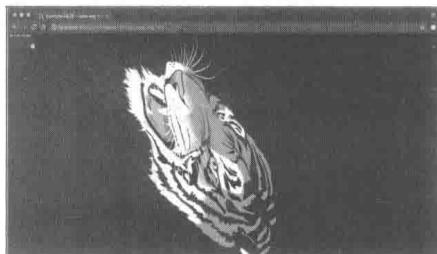
(续)

格 式	示 例	屏 幕 截 图
VTK	11-load-vtk.html	
AWD	14-load-awd.html	
Assimp	15-load-assimp.html	
VRML	16-load-vrml.html	
Babylon	<p>14-load-babylon.html          Babylon 加载器和表中的其他加载器稍有不同。使用此加载器，你不会加载单个 THREE.Mesh 或 THREE.Geometry 实例，而是加载一个完整的场景，包括光源</p>	

(续)

格 式	示 例	屏 幕 截 图
3DS	使用 18-load-tds.html 中 的 TDS-Loader 示例	
3MF	19-load-tmf.html	
AMF	20-load-amf.html	
PlayCanvas	21-load-play-canvas.html	
Draco	22-load-draco.html	

(续)

格 式	示 例	屏 幕 截 图
PRWM	23-load-prwm.html	
GCode	24-load-gcode.html	
NRRD	25-load-nrrd.html	
SVG	26-load-svq.html	

如果你看看这些例子的源码，会看到在有些例子中，我们需要在模型被正确渲染之前

改变一些材质的属性或是对其进行缩放。我们需要这样做是因为：在外部应用程序中创建模型的方式，会给出不同于我们通常在 Three.js 中使用的尺寸和组。

我们已经展示了几乎所有的格式。接下来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 PDB 格式中加载并渲染蛋白质，最后我们会使用 PLY 格式文件里定义的模型来创建一个粒子系统。

### 8.3.4 展示蛋白质数据银行中的蛋白质

PDB，即蛋白质数据银行 ([www.rcsb.org](http://www.rcsb.org)) 包含了很多分子和蛋白质的详细信息。除了提供蛋白质的信息，还可以用 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。在本节中，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首先要做的是包含正确的加载器，代码如下所示：

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

包含了这个加载器之后，我们就可以创建如图 8.15 所示分子结构的三维模型（见示例 12-load-ptb.html）。

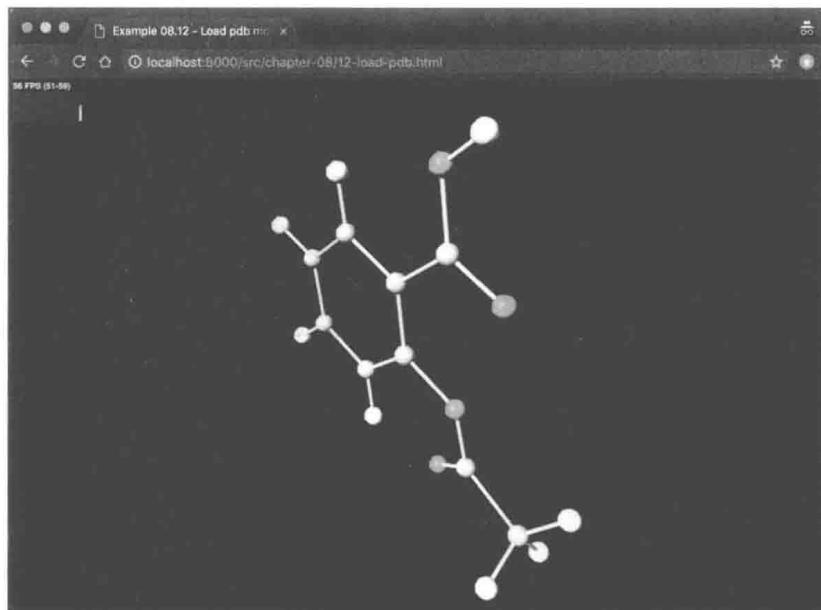


图 8.15

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，代码如下：

```
loader.load("../assets/models/molecules/aspirin.pdb", function
(geometries) {

    var group = new THREE.Object3D();

    // create the atoms
```

```

var geometryAtoms = geometries.geometryAtoms;

for (i = 0; i < geometryAtoms.attributes.position.count; i++) {
    var startPosition = new THREE.Vector3();
    startPosition.x = geometryAtoms.attributes.position.getX(i);
    startPosition.y = geometryAtoms.attributes.position.getY(i);
    startPosition.z = geometryAtoms.attributes.position.getZ(i);

    var color = new THREE.Color();
    color.r = geometryAtoms.attributes.color.getX(i);
    color.g = geometryAtoms.attributes.color.getY(i);
    color.b = geometryAtoms.attributes.color.getZ(i);

    var material = new THREE.MeshPhongMaterial({
        color: color
    });

    var sphere = new THREE.SphereGeometry(0.2);
    var mesh = new THREE.Mesh(sphere, material);
    mesh.position.copy(startPosition);
    group.add(mesh);
}

// create the bindings
var geometryBonds = geometries.geometryBonds;

for (var j = 0; j < geometryBonds.attributes.position.count; j += 2) {
    var startPosition = new THREE.Vector3();
    startPosition.x = geometryBonds.attributes.position.getX(j);
    startPosition.y = geometryBonds.attributes.position.getY(j);
    startPosition.z = geometryBonds.attributes.position.getZ(j);

    var endPosition = new THREE.Vector3();
    endPosition.x = geometryBonds.attributes.position.getX(j + 1);
    endPosition.y = geometryBonds.attributes.position.getY(j + 1);
    endPosition.z = geometryBonds.attributes.position.getZ(j + 1);

    // use the start and end to create a curve, and use the curve to draw
    // a tube, which connects the atoms
    var path = new THREE.CatmullRomCurve3([startPosition, endPosition]);
    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);
}

loaderScene.render(group, camera);

```

正如你在这个例子中所看到的，我们实例化了一个 THREE.PDBLoader 对象，将想要加载的模型文件传递给它，还提供了一个回调函数，该函数会在模型加载完成之后调用。在这个加载器中，回调函数有两个参数：geometryAtom 和 geometryBonds。参数 geometryAtom 里的各个顶点表示每一个原子的位置和颜色，而 geometryBonds 用来定义原

子间的键。

基于位置和颜色，我们创建 THREE.Mesh 对象并将它添加到组里：

```
var material = new THREE.MeshPhongMaterial({
  color: color
});

var sphere = new THREE.SphereGeometry(0.2);
var mesh = new THREE.Mesh(sphere, material);
mesh.position.copy(startPosition);
group.add(mesh);
```

原子间的键用相同的方法。我们得到了连接的开始和结束位置并以此画出连接：

```
var path = new THREE.CatmullRomCurve3([startPosition, endPosition]);
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);
```

对于原子间的键，我们先用 THREE.CatmullRomCurve3 对象创建了一个三维路径。该路径用作 THREE.Tube 对象的输入，并用来创建原子间的键。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。

图 8.16 所展示的就是钻石的分子结构。

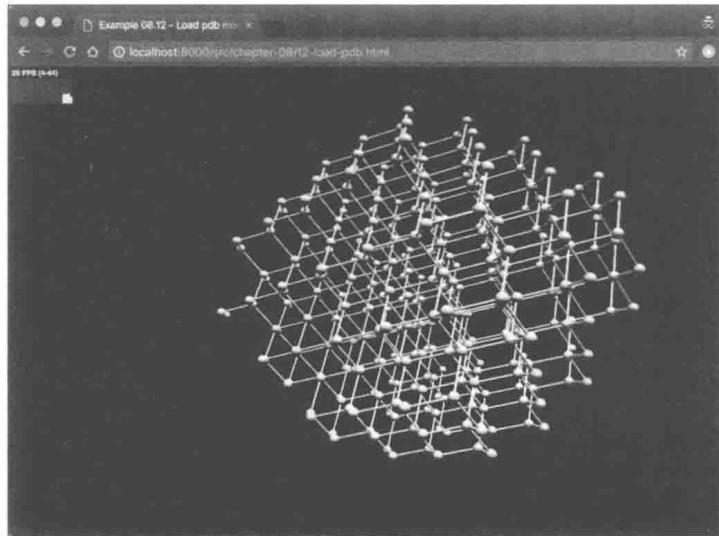


图 8.16

### 8.3.5 从 PLY 模型中创建粒子系统

PLY 格式的使用跟其他格式并无很大的区别，只要包含加载器，提供一个回调函数，并渲染模型即可。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。详见示例 15-load-ply.html。本例效果如图 8.17 所示。

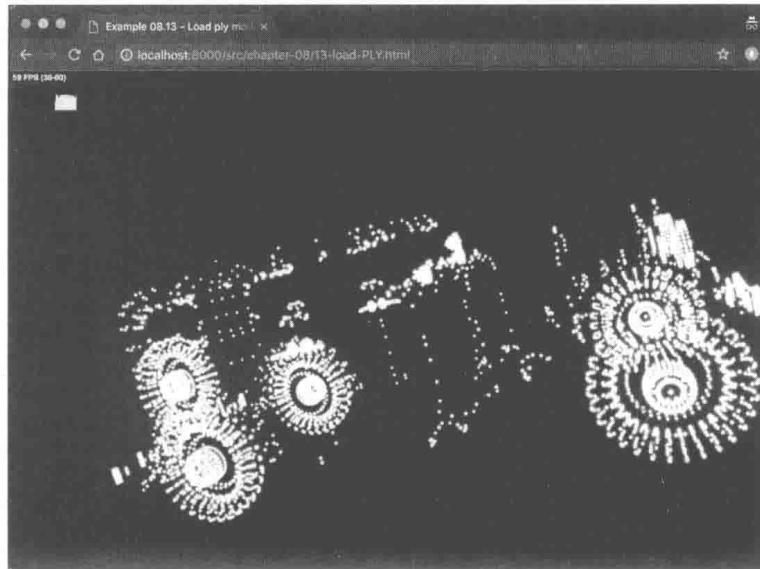


图 8.17

渲染出上述截图的 JavaScript 代码实际上非常简单，如下所示：

```
loader.load("../assets/models/carcloud/carcloud.ply", function(geometry) {
    var material = new THREE.PointsMaterial({
        color: 0xffffffff,
        size: 1,
        opacity: 0.6,
        transparent: true,
        blending: THREE.AdditiveBlending,
        depthWrite: false,
        map: generateSprite()
    });

    var group = new THREE.Points(geometry, material);
    group.scale.set(2.5, 2.5, 2.5);

    loaderScene.render(group, camera);
});
```

正如你所看到的，我们使用 THREE.PLYLoader 来加载模型。回调函数接受 geometry 作为参数，然后我们用这个 geometry 作为 THREE.Points 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 可以很容易地结合各种源文件中的模型，并且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。

## 8.4 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，只要区区几步就可以完成。在使用外部模型或者对模型进行组合和合并时，最好记住以下几点。首先，在组合对象的时候，每个对象依然可以单独进行操作。对父对象进行变换也会影响子对象，但是你仍然可以单独对每个子对象进行变换。除了组合之外，还可以将几何体合并在一起。这样的话，你就失去了对单个几何体的控制，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。

Three.js 支持大量的外部格式。使用这些格式加载器时，最好看看源码，并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，一般是材质设置导致的。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。通常可以通过测试材质来检测这种错误，也可以在 JavaScript 控制台中输出材质信息，看看有没有比较意外的值。要想导出网格和场景，只需要调用 scene 对象的 `asJson` 函数即可；反过来，使用 `JSONLoader` 可以将导出的内容重新加载到场景中。

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 中各种控制摄像机的方法。你可以在场景中移动、平移和旋转摄像机。

## 创建动画和移动摄像机



在上一章中我们已经使用了一些简单的动画。在第1章中我们介绍了基础的渲染循环，并在第2章使用渲染循环旋转场景中的物体，还展示了其他几个基础的动画概念。本章我们将深入了解Three.js提供的动画是如何工作的。我们将详细讨论下面四个主题：

- 基础动画。
- 移动摄像机。
- 变形和骨骼动画。
- 使用外部模型创建动画。

我们将从动画背后的基本概念开始。

### 9.1 基础动画

在开始本章示例之前，我们先来回顾第1章中的渲染循环。为了实现动画，我们需要告诉Three.js多久渲染一次场景。因此我们使用HTML5提供的requestAnimationFrame()方法来实现，具体代码如下所示：

```
render();  
  
function render() {  
    // render the scene  
    renderer.render(scene, camera);  
    // schedule the next rendering using requestAnimationFrame  
    requestAnimationFrame(render);  
}
```

在这段代码里，我们只需在场景初始化完成后调用一次render()方法即可。在render()方

法内部我们使用 `requestAnimationFrame()` 方法来触发下一次的场景渲染。这样，浏览器就可以保证以合适的时间间隔（通常是 60 次每秒）执行 `render()` 方法。在浏览器支持 `requestAnimationFrame()` 方法之前，一般用 `setInterval(function, interval)` 或 `setTimeout(function, interval)` 方法。这两个方法会以指定的时间间隔触发指定的方法，但是这两个方法的缺点是在触发指定方法时不会考虑当前正在发生的事情。即使动画没有显示或者隐藏在某个标签下，这两个函数依然会触发指定的方法，这样就会耗费某些资源。除此之外，这两个函数还存在一个缺点，那就是它们一旦被调用就会刷新屏幕，不管这个时刻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()` 方法，我们不需要通知浏览器什么时候需要刷新屏幕，而是请求浏览器在合适的时候执行我们提供的方法，通常情况下是 60fps（帧频）。使用 `requestAnimationFrame()` 方法实现的动画运行得更加平滑，对于 CPU 和 GPU 更加友好，而且也不必担心渲染时机方面的问题。

### 9.1.1 简单动画

我们可以通过改变物体的旋转、缩放、位置、材质、顶点、面以及其他你所能想到的属性来实现动画。Three.js 会在下一次的 `render` 循环中渲染修改后的属性。在第 1 章有个简单的动画（01-basic-animation.html），效果如图 9.1 所示。

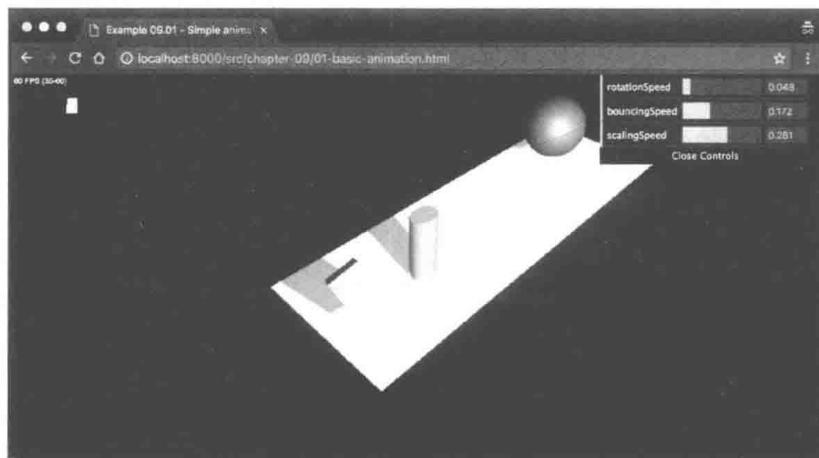


图 9.1

该示例的 `render` 循环非常简单，我们只是修改了网格的属性，然后 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);
}

```

虽然这段代码没有特别之处，但是它却很好地展示了本书所要讨论的各种动画背后的概念。当使用 Three.js 实现复杂场景时，除了动画之外，还有个重要的方面需要考虑，那就是如何通过鼠标选中场景中的对象。

### 9.1.2 选择对象

本章主要讨论摄像机和动画，尽管如何选择对象跟动画没有直接的联系，但是在讨论一下有助于更好地了解本章的内容。我们在这里展示的是如何使用鼠标选择场景中的对象，具体代码如下所示：

```

var projector = new THREE.Projector();
function onDocumentMouseDown(event) {

    var vector = new THREE.Vector3((event.clientX / window.innerWidth) * 2
- 1, -(event.clientY / window.innerHeight) * 2 + 1, 0.5);
    vector = vector.unproject(camera);

    var raycaster = new THREE.Raycaster(camera.position,
vector.sub(camera.position).normalize());
    var intersects = raycaster.intersectObjects([sphere, cylinder, cube]);

    if (intersects.length > 0) {
        console.log(intersects[0]);
        intersects[0].object.material.transparent = true;
        intersects[0].object.material.opacity = 0.1;
    }
}

```

在这段代码中，我们使用 THREE.Projector 和 THREE.Raycaster 来检测是否使用鼠标点击了某个对象。当我们在屏幕上点击鼠标时，会发生如下的事情：

- (1) 首先，基于屏幕上的点击位置会创建一个 THREE.Vector3 向量。
- (2) 接着，使用 vector.unproject 方法将屏幕上的点击位置转换成 Three.js 场景中的坐标。换句话说，就是将屏幕坐标转换成三维场景中的坐标。
- (3) 然后，创建 THREE.Raycaster。使用 THREE.Raycaster 可以向场景中发射光线。在该示例中，从摄像机的位置 (camera.position) 向场景中鼠标的点击位置发射光线。
- (4) 最后，我们使用 raycaster.intersectObjects 方法来判断指定的对象中哪些被该光线照射到了。

上述最后一步的结果中包含了所有被光线照射到的对象的信息。这些信息包括：

```
▼ {distance: 62.06043252608528, point: Vector3, object: Mesh, uv: Vector2, face: Face3, ...}
  distance: 62.06043252608528
  ▶ face: Face3 {a: 14, b: 15, c: 7, normal: Vector3, vertexNormals: Array(3), ...}
    faceIndex: 13
  ▶ object: Mesh {uuid: "19E6C6A6-5DA3-436C-B42E-63A87568E680", name: "", type: "Mesh", parent: Scene, children: Array(0), ...}
  ▶ point: Vector3 {x: -0.2918410329675761, y: -6.669043305314233, z: 1.8774922991090282}
  ▶ uv: Vector2 {x: 0.8281414821798634, y: 0.03480514310623093}
  ▶ __proto__: Object
```

我们所点击的网格是对象，face 和 faceIndex 指的是该网格中被选中的面。distance 属性是从摄像机到被点击对象的距离，point 属性则表明网格上哪个点被点击了。最后，我们还可以获得点击位置所对应的 2D 纹理的 uv 值（uv 值的值域在 0 到 1 之间。更多关于纹理的知识将在第 10 章学习）。你可以通过示例 02-selecting-objects.html 来测试这个功能。在该示例中任何被选中的对象会变得透明，而且相关的信息会输出在浏览器的控制台上。

如果你想看到光线的投射路径，你可以打开菜单中的 showRay 属性，如图 9.2 所示。

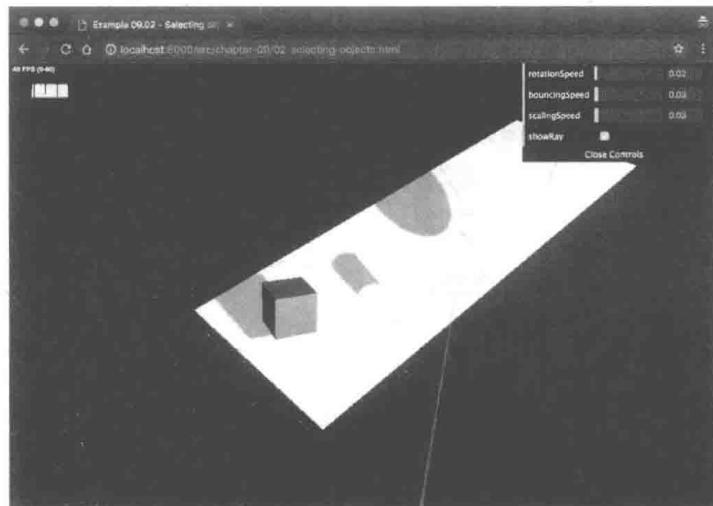


图 9.2

我们已经介绍了如何选择场景中的对象，接下来重新回到动画上来。现在通过在渲染循环中修改对象的属性来让对象动起来，那么在下一节中，我们将会介绍一个轻量的 JavaScript 库，使用该库可以使得动画定义变得更加简单。

### 9.1.3 使用 Tween.js 实现动画

Tween.js 是一个轻量级的 JavaScript 库，可以从 <https://github.com/sole/tween.js/> 下载。通过这个库可以很容易地实现某个属性在两个值之间进行过渡，而且起始值和结束值之间的所有中间值都会自动计算出来，这个过程叫作 tweening（补间）。例如，你可以使用这个库将网格的 x 轴坐标值在 10 秒内从 10 递减到 3，代码如下所示：

```
var tween = new TWEEN.Tween({x: 10}).to({x: 3}, 10000)
  .easing(TWEEN.Easing.Elastic.InOut)
```

```
.onUpdate( function () {
    // update the mesh
})
```

在这个例子中我们创建了 TWEEN.Tween 对象，这个对象会确保 `x` 属性值在 10 000 毫秒内从 10 变化到 3。通过 Tween.js，你还可以指定属性值是如何变化的，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](http://sole.github.io/tween.js/examples/03_graphs.html)）。属性值在指定时间内的变化被称为 easing（缓动），在 Tween.js 中你可以使用 `easing()` 方法来配置缓动效果。

在 Three.js 中使用这个库是非常简单的。打开示例 03-animation-tween.html，你可以看到 Tween.js 库的效果。如图 9.3 所示。

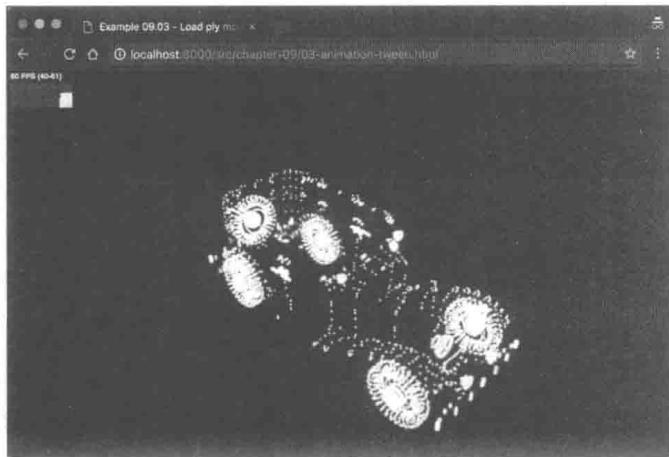


图 9.3

该示例程序将使用 Tween.js 通过编写一个特殊的 `easing()` 函数将上图中的模型以动画的形式收缩为一个点。图 9.4 为收缩后的画面：



图 9.4

在该示例中我们使用了第 7 章中创建的粒子系统，并给粒子添加落向地面的效果，而这些粒子的位置是由 Tween.js 库创建出来的，如下所示：

```
var posSrc = { pos: 1 };
var tween = new TWEEN.Tween(posSrc).to({pos: 0}, 2000);
tween.easing(TWEEN.Easing.Bounce.InOut);

var tweenBack = new TWEEN.Tween(posSrc).to({pos: 1}, 2000);
tweenBack.easing(TWEEN.Easing.Bounce.InOut);
tweenBack.chain(tween);
tween.chain(tweenBack);
tween.start();

var onUpdate = function () {
    var count = 0;
    var pos = this.pos;

    loadedGeometry.vertices.forEach(function (e) {
        var newY = ((e.y + 3.22544) * pos) - 3.22544;
        particleCloud.geometry.vertices[count++].set(e.x, newY, e.z);
    });

    particleCloud.sortParticles = true;
};

onUpdate();
```

在这段代码中，我们创建了两个补间：tween 和 tweenBack。第一个补间定义了 position 属性如何从 1 过渡到 0，第二个刚好相反。通过 chain() 方法可以将这两个补间衔接起来，这样当动画启动之后，程序就会在这两个补间循环。Tween 计算获得的数据可以通过两个方法获取并加以利用：可以在 Tween 库的 onUpdate 函数里获取数据并调用我们自己的代码进行进一步处理（每一次当 TWEEN.update 函数被调用时，onUpdate 函数都会被调用），也可以直接访问 Tween 的内部数据。在本示例程序中，我们将使用后一个方法。在开始修改渲染函数之前，首先需要看一下加载模型之后，我们需要增加的额外一步工作。由于我们希望在动画过程中，让模型的形态在原始值和 0 之间循环，因此需要先将模型的原始顶点数据做备份。可以在模型加载器的加载完成回调函数里保存原始数据。

```
// copy the original position, so we can referene that when tweening
var origPosition = geometry.attributes['position'].clone()
geometry.origPosition = origPosition
```

无论何时需要获得模型的原始数据，直接使用 origPosition 变量中的数据即可。



本例使用了 BufferGeometry 型加载器，这是因为我们只需要备份模型的顶点位置信息，而不需要备份顶点的全部数据。

现在就可以基于备份的原始数据，以及从 Tween 中获取的变化因数数据来为动画的每一帧计算当前数据。我们需要在渲染函数里添加下面代码：

```
TWEEN.update();

var positionArray = mesh.geometry.attributes['position']
```

```

var origPosition = mesh.geometry.origPosition

for (i = 0; i < positionArray.count ; i++) {
    var oldPosX = origPosition.getX(i);
    var oldPosY = origPosition.getY(i);
    var oldPosZ = origPosition.getZ(i);
    positionArray.setX(i, oldPosX * posSrc.pos);
    positionArray.setY(i, oldPosY * posSrc.pos);
    positionArray.setZ(i, oldPosZ * posSrc.pos);
}
positionArray.needsUpdate = true;

```

在上面的渲染函数代码中，我们先调用 TWEEN.Update 函数令 Tween 重新计算新的数据变化因数（变化因数会在 1 和 0 之间循环）。这里请回忆一下在前面的示例中曾使用过的 posSrc 变量。然后我们遍历模型所有顶点，并为它们计算和更新顶点位置数据。

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，使用这个库比自己来管理这些属性值之间的过渡要简单得多。除了让物体动起来，或者改变其外观，我们还可以通过移动摄像机的位置来让整个场景动起来。在前面的章节中，我们是通过手动更新摄像机的位置来实现这一点。Three.js 还提供了其他几种方法来更新摄像机。

## 9.2 使用摄像机

Three.js 提供了一些摄像机控件，使用这些控件，你可以控制场景中的摄像机。这些控件在 Three.js 发布包中，你可以在 examples/js/controls 目录中找到它们。在本节中，我们将介绍表 9.1 所列的控件。

表 9.1

名 称	描 述
第一视角控制器 (FirstPersonControls)	该控制器的行为类似第一视角射击游戏中的摄像机。使用键盘移动，使用鼠标转动
飞行控制器 (FlyControls)	飞行模拟控制器，用键盘和鼠标控制摄像机的移动
翻滚控制器 (RollControls)	该控制器是飞行控制器的简化版，允许绕着 z 轴旋转
轨迹球控制器 (TrackBallControls)	最常用的控制器，你可以使用鼠标（或控制球）来轻松移动、平移和缩放场景。注意，如果你使用的是 OrthographicCamera，你可以使用 OrthographicTrackBallControls，它是这个摄像机类型专用的
轨道控制器 (OrbitControls)	该控件可以在特定的场景中模拟轨道中的卫星，你可以使用鼠标和键盘在场景中游走

这几个是最常用的控件，除此之外，Three.js 还提供了许多其他可用的控件（在本书中不会进行介绍）。其他控件的使用方式和上一个表格列出的控件是一样的，见表 9.2。

表 9.2

名 称	描 述
设备朝向控制器 (DeviceOrientationControls)	该控制器可以使得摄像机依据设备的朝向来进行调整。它的实现是基于 HTML 的设备朝向 API ( <a href="http://www.w3.org/TR/orientation-event/">http://www.w3.org/TR/orientation-event/</a> )
编辑控制器 (EditorControls)	该控制器是为在线三维编辑器而创建的，并被用于 Three.js 的在线编辑器中 <a href="http://threejs.org/editor/">http://threejs.org/editor/</a> ，具体请访问
Oculus 控制器 (OculusControls)	该控制器可以允许使用 Oculus Rift 设备来环顾场景
正交轨迹球控制器 (OrthographicTrackball Controls)	该控制器和轨迹球控制器类似，只不过是用于 THREE.Orthographic Camera
鼠标锁定控制器 (PointerLockControls)	该控制器使用场景中渲染的 DOM 元素来锁定鼠标。可以为 3D 游戏提供基本的功能
变换控制器 (TransformControls)	这个是 Three.js 编辑器内部使用的控制器
VR 控制器 (VRControls)	该控制器使用 PositionSensorVRDevice API 来控制场景。更多信息参照 <a href="https://developer.mozilla.org/en-US/docs/Web/API/Navigator.getVRDevices">https://developer.mozilla.org/en-US/docs/Web/API/Navigator.getVRDevices</a>



仔细观察示例代码可以发现我们使用了 DragControls 控件类。它虽然也叫作控件但却与其他摄像机控件不同。当用鼠标在场景中拖动时，这个控件会移动场景中的物体而不是摄像机。

除了使用控制器，你还可以通过修改 position 属性来移动摄像机，通过 lookAt() 方法来改变摄像机的朝向。



如果你对旧版 Three.js 比较熟悉，可能会怀念名为 THREE.PathControls 的摄像机控件，该控制器可以用于定义一个路径（如使用 THREE.Spline）并让摄像机沿着这个路径进行移动。但是由于代码较复杂，所以在新版本中移除了这个控制器。Three.js 的开发团队正着手提供该控制器的替代功能，可惜暂时还没有实现。

另一个在新版中被移除了的控件是 THREE.RollControls。该控件的功能被合并到了 THREE.FlyControls 控件中，因此在新版中被移除，但是在必要时，对 THREE.FlyControls 进行合理设置便可完全重现 THREE.RollControls 的功能。

下面将介绍第一个控制器——轨迹球控制器 (TrackballControls)。

### 9.2.1 轨迹球控制器

如果要使用 TrackballControls，首先需要在网页中包含对应的 JavaScript 文件：

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

包含了这个文件后，我们可以创建控制器，并将它绑定到摄像机上，代码如下所示：

```
var trackballControls = new THREE.TrackballControls(camera);
trackballControls.rotateSpeed = 1.0;
trackballControls.zoomSpeed = 1.0;
trackballControls.panSpeed = 1.0;
```

摄像机的位置更新可以在 render 循环中完成，代码如下所示：

```
var clock = new THREE.Clock();
function render() {
    var delta = clock.getDelta();
    trackballControls.update(delta);
    requestAnimationFrame(render);
    webGLRenderer.render(scene, camera);
}
```

在这个代码片段里，我们创建了一个新的 Three.js 对象：THREE.Clock。调用 clock.getDelta() 方法可以精确地计算出此次调用距离上次调用的时间间隔，或者一个渲染循环花费的时间。若要更新摄像机的位置，可以调用 trackballControls.update() 方法，这个方法的参数是距离上次调用的时间间隔，所以我们需要使用 THREE.Clock 对象的 getDelta() 方法。或许你觉得我们可以直接将帧频（1/60 秒）传递给 update 方法，没有这样做是由于受外部因素的影响，无法保证帧频始终为 60fps。为了让摄像机能够平缓地移动和旋转，我们需要传入精确的时间间隔。

实际的例子可以参看示例 04-trackball-controls-camera.html，如图 9.5 所示。

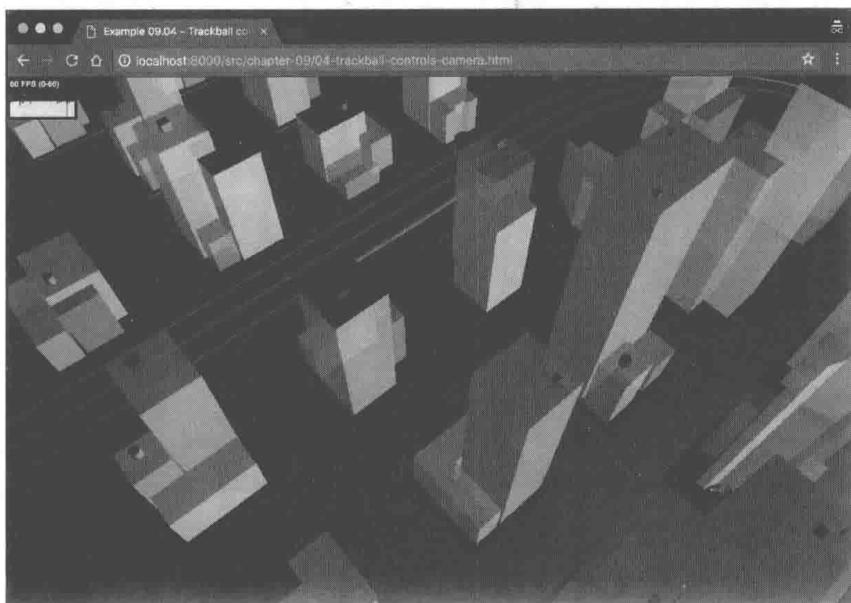


图 9.5

你可以使用如表 9.3 中所列的方式来控制摄像机。

表 9.3

操 控	效 果
按住鼠标左键，拖动	摄像机在场景中旋转和翻滚
转动鼠标滚轮	放大和缩小
按住鼠标中键，拖动	放大和缩小
按住鼠标右键，拖动	摄像机在场景中平移

还有几个属性可以用来对摄像机的行为进行微调。例如，你可以使用 `rotateSpeed` 属性来设置摄像机的旋转速度，将 `noZoom` 属性设置为“`true`”来禁用缩放。在本章我们不会详细地介绍每个属性，因为属性的名字就可以很好地解释其用途。要想全面了解这些属性，可以参照 `rackballControls.js` 文件。

### 9.2.2 飞行控制器

接下来我们将要介绍的是飞行控制器（`FlyControls`）。使用 `FlyControls` 你可以实现像飞行模拟器一样在场景中飞行。具体示例参看 `05-fly-controls-camera.html`，效果如图 9.6 所示。

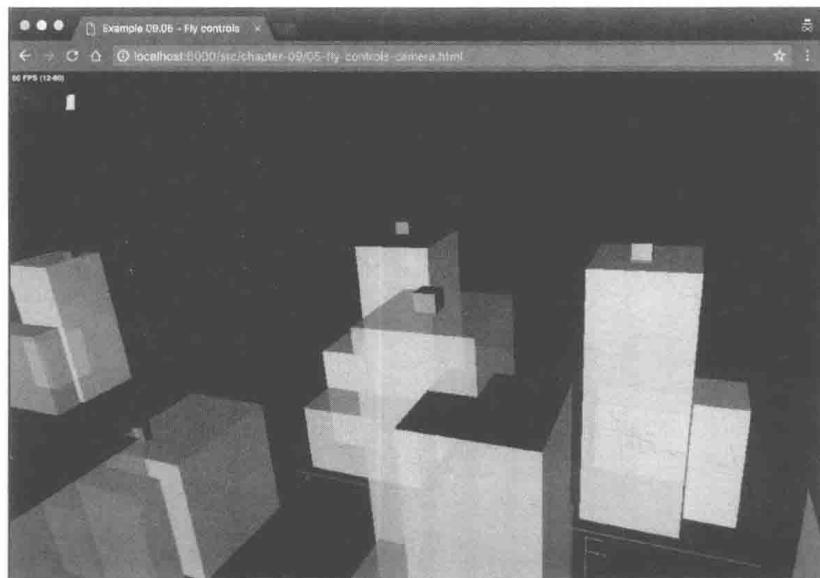


图 9.6

`FlyControls` 的使用和 `TrackballControls` 一样，首先需要加载对应的 JavaScript 文件：

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

然后配置控制器并将其绑定到摄像机上，代码如下：

```
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，我们不会对所有的细节进行解释，若想了解细节，可以参看 FlyControls.js 文件。这里我们只对保证控制器正常工作的 `domElement` 属性进行介绍。该属性值应该指向场景所渲染到的元素。在本书示例中，场景所渲染到的元素为：

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

所以 `domElement` 属性的配置如下：

```
flyControls.domElement = document.querySelector('#webgl-output');
```

如果没有正确地设置 `domElement` 属性，那么移动鼠标时会导致奇怪的行为出现。可以使用表 9.4 中所列的方式来控制摄像机。

表 9.4

操    控	效    果
按住鼠标左键和中键	向前移动
按住鼠标右键	向后移动
移动鼠标	向四周看
W	向前移动
S	向后移动
A	向左移动
D	向右移动
R	向上移动
F	向下移动
上、下、左、右方向键	向上、下、左、右看
G	向左翻滚
E	向右翻滚

下一个我们需要介绍的控件是翻滚控制器（RollControls）。

### 9.2.3 第一视角控制器

顾名思义，通过第一视角控制器（FirstPersonControls）你可以像第一视角射击游戏那样控制摄像机。鼠标用于控制视角，键盘用于控制移动角色。具体示例参看 07-first-person-camera.html，效果图如图 9.7 所示。



图 9.7

该控制器的创建跟我们之前看到的控件一样。我们展示的示例配置如下：

```
var fpControls = new THREE.FirstPersonControls(camera);
fpControls.lookSpeed = 0.4;
fpControls.movementSpeed = 20;
fpControls.lookVertical = true;
fpControls.constrainVertical = true;
fpControls.verticalMin = 1.0;
fpControls.verticalMax = 2.0;
fpControls.lon = -150;
fpControls.lat = 120;
```

在使用该控件时，需要特别注意的两个属性是：lon 和 lat。这两个属性定义的是场景初次渲染时摄像机所指向的位置。

该控件的控制方法见表 9.5。

表 9.5

操作	效果
移动鼠标	向四周看
上、下、左、右方向键	向上、下、左、右移动
W	向前移动
A	向左移动
S	向后移动
D	向右移动

(续)

操 控	效 果
R	向上移动
F	向下移动
Q	停止移动

下面介绍最后一个控制器，该控制器用于从空间视角观察场景。

#### 9.2.4 轨道控制器

轨道控制器（OrbitControl）可以用于控制场景中的对象围绕场景中心旋转和平移。示例 08-controls-orbit.html 展示了该控件是如何工作的，效果如图 9.8 所示。

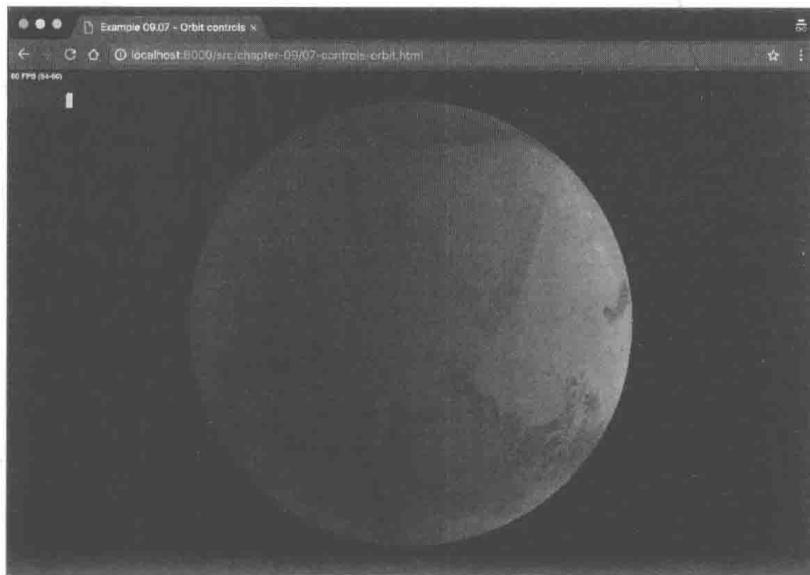


图 9.8

像其他控件的使用方式一样，我们需要引入对应的 JavaScript 文件，将控件绑定到摄像机上，然后使用 THREE.Clock 对象来更新控制器：

```
<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);
```

轨道控制器主要使用鼠标，见表 9-6。

表 9.6

操    控	效    果
按住鼠标左键并移动	摄像机围绕场景中心旋转
转动鼠标滑轮或按住中键并移动	放大和缩小
按住鼠标右键并移动	在场景中平移
上、下、左、右方向键	在场景中平移

这就是有关摄像机移动的所有内容，而且到现在我们介绍了许多控制摄像机行为的控制器。在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

当你使用外部软件（如 Blender）创建动画时，通常会有两种主要的动画定义方式：

- 变形动画：使用变形动画，你需要定义网格变形之后的状态，或者说是关键位置。对于变形目标，所有的顶点位置都会被存储下来。你所需要做的是将所有的顶点从一个位置移动到另外一个定义好的关键位置，并重复该过程。图 9.9 展示的是一组表示面部表情变形的动画（图片由 Blender 基金组织提供的）。

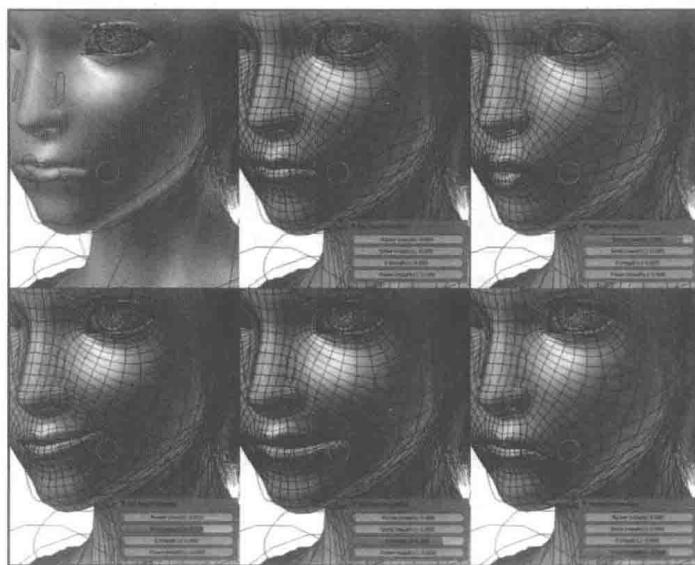


图 9.9

- 骨骼动画：另一种替代方法是骨骼动画。使用骨骼动画时你需要定义骨骼，也就是网格的骨头，并将顶点绑定到特定的骨头上。然后，当你移动一块骨头时，任何与其相连的骨头都会做相应的移动，同时骨头上绑定的顶点也会随之移动。网格的变

形也是基于骨头的位置、移动和缩放实现的。图 9.10（也是由 Blender 基金会提供）展示的就是如何使用骨头来移动和变形物体。

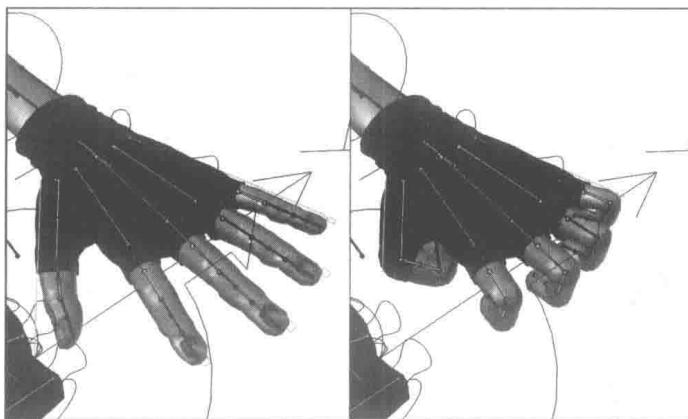


图 9.10

Three.js 对这两种模式都支持，但是相比较而言，使用变形动画能够得到更好的效果。骨骼动画的主要问题是如何从 Blender 等三维程序中较好地导出数据，从而在 Three.js 中制作动画，但是变形动画就可以很容易获取良好的工作模型。

本节会详细介绍这两种模型，同时还会介绍几个 Three.js 支持的可以定义动画的外部格式。

### 9.3.1 用变形目标创建动画

变形目标是实现动画的最直接方式。你可以为所有顶点定义一系列的关键位置（也称关键帧），然后让 Three.js 将这些顶点从一个位置移动另外一个位置。但这种方法有一个缺点，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，所有顶点的位置都需要重复存储一遍。

我们将用两个例子来阐述如何使用变形目标。在第一个例子里我们会让 Three.js 来处理关键帧（或者变形目标，此后我们将会这样称呼它）之间的过渡。在第二个例子中，我们会手动处理这些变化。请记住，本书只能浅尝辄止地讲述 Three.js 的动画功能。在本章接下来的内容里你会看到 Three.js 对动画控制、动画同步以及动画帧间的平滑过渡都有着丰富的功能支持，而这些功能本身就可以另著一本书来讲述。所以在下面的小节中我们只讲述 Three.js 中关于动画的基本功能，使你能够以此为起点自行探索关于动画的更多更复杂的主题。

#### 1. 使用混合器和变形目标创建动画

在研究示例程序之前，先来看一看 Three.js 的三个核心动画类，因为接下来所有动画相关的功能和属性均由这三个类提供：

- ❑ THREE.AnimationClip：当一个具有动画数据的模型被加载后，获得的模型对象往

往具有一个名为 `animations` 的成员对象。该对象包含了一个 `THREE.AnimationClip` 对象集合。

一个模型所包含的 `THREE.AnimationClip` 对象通常保存有某种特定类型的动画数据，或者是该模型能够执行的某种动作。比如当加载了一个鸟模型时，它可能包含了两个 `THREE.AnimationClip` 对象，一个保存了拍打翅膀的动作，而另一个保存了张嘴闭嘴的动作。

- `THREE.AnimationMixer`: `THREE.AnimationMixer` 对象用于控制多个 `THREE.AnimationClip` 对象，确保这些动画在适当的时间发生，使动画同步或者控制从一个动画过渡到另一个。
- `THREE.AnimationAction`: 当向 `THREE.AnimationMixer` 对象添加一个 `THREE.AnimationClip` 对象时，调用者会获得一个 `THREE.AnimationAction`。很多动画控制功能是通过 `THREE.AnimationAction` 来调用的，而 `THREE.AnimationMixer` 本身并没有提供很全面的控制接口。除了初次添加动画之外，`THREE.AnimationAction` 对象也可以随时从 `THREE.AnimationMixer` 获取。

在下面的示例程序中，你可以控制 `THREE.AnimationMixer` 或 `THREE.AnimationAction` 对象，后者所对应的 `THREE.AnimationClip` 对象包含了马奔跑的动作。

作为第一个变形动画示例，我们使用了 Three.js 自带的动画模型：马。为了更好地理解变形目标类型动画，请打开示例程序 `10-morph-targets.html` 并动手操作一下。图 9.11 展示了该示例的一个静态截图。

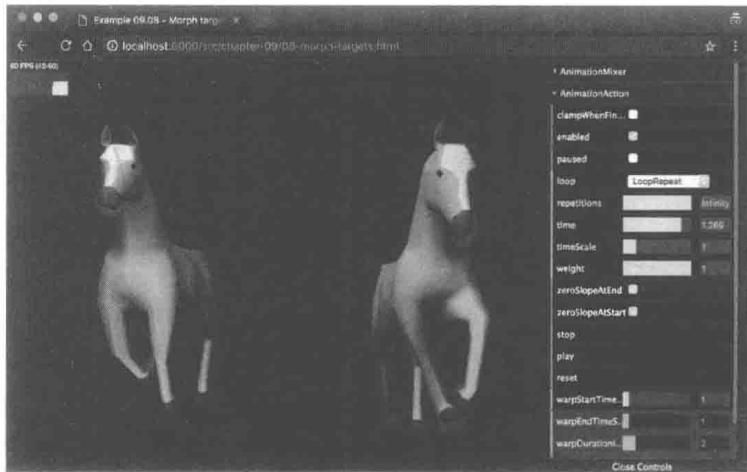


图 9.11

在该示例中，右侧的那匹马是奔跑着的，而左侧的那匹马是静止的。左侧那匹马是从基础模型（即原始顶点集）渲染而来的。通过右上角的菜单，你可以看到所有的变形目标，以及左侧那匹马所有可能的位置。在菜单里还有对 `THREE.AnimationMixer` 和 `THREE.`

AnimationAction 的控制（关于这些控制的更多详细描述，请参考本章后面的表格）。

在旧版 Three.js 中，必须使用特殊网格类（比如 THREE.MorphAnimMesh 或 THREE.MorphBlendMesh）才能使用动画功能，而在新版中则可以使用普通的 THREE.Mesh 类。为了播放一个动画，下面要使用 THREE.AnimationMixer 类：

```
var loader = new THREE.JSONLoader();
loader.load('.../assets/models/horse/horse.js', function (geometry, mat)
{
    geometry.computeVertexNormals();
    geometry.computeMorphNormals();

    var mat = new THREE.MeshLambertMaterial({morphTargets: true,
    vertexColors: THREE.FaceColors});
    mesh = new THREE.Mesh(geometry, mat);
    mesh.scale.set(0.15, 0.15, 0.15);
    mesh.translateY(-10);
    mesh.translateX(10);

    mixer = new THREE.AnimationMixer( mesh );
    // or create a custom clip from the set of morphtargets
    // var clip = THREE.AnimationClip.CreateFromMorphTargetSequence(
    'gallop', geometry.morphTargets, 30 );
    mixer.clipAction( geometry.animations[0] ).setDuration( 1 ).play();
    scene.add(mesh)
})
```

一个 THREE.AnimationMixer 对象控制一个或多个动画模型。在本示例程序中，我们为加载后的几何体创建了一个 THREE.AnimationMixer 对象，并用 mixer.clipAction 方法来播放模型包含的第一个动画。该方法接收一个 THREE.AnimationClip 对象并返回一个 THREE.AnimationAction 控制对象。我们利用返回的控制对象设置动画时长为 1 秒，并调用其 play 方法开始播放。



了解一个模型所包含的哪些动画可以被支持，最好的方法是将动画信息打印到控制台。不同的加载器会将动画加载为不同的动画类型。此外，在让一个模型动起来之前，最好先让程序渲染一下模型的原始形态（译注：这样有助于判断动画效果是正常合理的）。

到目前为止，我们的程序还不能播放动画，需要在渲染函数里做一点小修改：

```
function render() {
    stats.update();
    var delta = clock.getDelta();
    trackballControls.update(delta);
    requestAnimationFrame(render);
    renderer.render(scene, camera)

    if (mixer) {
        mixer.update( delta );
    }
}
```

从上面的代码中可以看到，我们必须以 `mixer.update(delta)` 的形式去调用 `update` 函数，以便告诉混合器本次渲染和上次渲染之间的时间差。混合器将根据时间差来判断应该将模型的顶点从上一个变形目标（即关键帧）向下一个变形目标移动多远。

`THREE.AnimationMixer` 和 `THREE.AnimationClip` 还提供了更多控制功能，用于控制动画或者创建新的 `THREE.AnimationClip` 对象。表 9.7 列举了 `THREE.AnimationClip` 类的属性和方法。

表 9.7

名 称	描 述
<code>duration</code>	当前动画轨道的时长（单位：秒）
<code>name</code>	本动画的名称，在马的例子中，名称为 <code>gallop</code>
<code>tracks</code>	内部属性，用于标记模型的某一特定属性的动画状态
<code>uuid</code>	本动画的 ID，由系统自动分配
<code>optimize()</code>	优化 <code>AnimationClip</code> 的动画数据
<code>resetDuration()</code>	恢复本动画的默认时长
<code>trim()</code>	根据新指定的时长对内部各个动画轨道进行剪裁
<code>CreateClipsFromMorphTargetSequences(name, morphTargetSequences, fps, noLoop)</code>	被 <code>Three.js</code> 内部调用。当加载外部模型资源中的变形动画数据时， <code>THREE.JsonLoader</code> 调用此函数来创建 <code>AnimationClip</code> 对象
<code>CreateFromMorpTargetSequences(name, morphTargetSequence, fps, noLoop)</code>	基于给定的变形动画序列创建单一 <code>AnimationClip</code> 对象
<code>findByName(objectOrClipArray, name)</code>	根据名称查找特定的 <code>AnimationClip</code> 对象
<code>Parse()</code> 和 <code>toJson()</code>	从 JSON 文件加载 <code>AnimationClip</code> 对象或者将内存中的对象保存到 JSON 文件
<code>parseAnimation()</code>	被 <code>Three.js</code> 内部调用，用于创建 <code>AnimationClip</code> 对象

当获得一个 `AnimationClip` 对象后，可以将它传给 `THREE.AnimationMixer` 去使用。表 9.8 列举了 `THREE.AnimationMixer` 类的属性和方法。

表 9.8

名 称	描 述
<code>AnimationMixer(rootObject)</code>	构造函数。它接受一个 3D 物体对象作为参数，例如一个 <code>THREE.Mesh</code> 对象或者一个 <code>THREE.Group</code> 对象。
<code>.time</code>	当前混合器对象创建的时间，为从 0 开始的全局时间
<code>.timeScale</code>	指定由当前混合器对象所控制的所有动画的时间缩放值，它可以用来加快或者放慢动画的速度。如果该属性为 0，则所有被它控制的动画会暂停播放

(续)

名 称	描 述
.clipAction(animationClip,optionalRoot)	根据指定的动画对象 THREE.AnimationClip 来创建用于控制动画的 THREE.AnimationAction 对象。动画对象所对应的 3D 物体对象可以与从构造函数传入的对象不同
.existingAction(animationClip,optionalRoot)	根据指定的动画对象 THREE.AnimationClip 返回一个之前创建过的 THREE.AnimationAction 对象。动画对象所对应的 3D 物体对象可以与从构造函数传入的对象不同

前面提到 THREE.AnimationAction 的作用是控制动画，表 9.9 列举了该类的属性和方法。

表 9.9

名 称	描 述
clampWhenFinished	设置为 true 时，动画会在播放到末帧时进入暂停状态。默认值是 false
enabled	设置为 false 时，动画会停止改变其宿主模型。(译注：此时动画帧仍然会继续前进，只是不会改变其宿主模型的形状。这是与 pause 函数的主要区别。) 设置为 true 时，动画会从当前所在的帧继续改变其宿主模型
loop	动画的循环模式(可以通过 setLoop 函数设置模式值)。有如下三种循环模式： <ul style="list-style-type: none"><li>• THREE.LoopOnce：动画只播放一遍。</li><li>• THREE.LoopRepeat：动画循环播放 repetitions 遍。</li><li>• THREE.LoopPingPong：动画循环播放 repetitions 遍。与 LoopRepeat 的区别是动画播放会在正向和反向之间交替循环</li></ul>
paused	设为 true 时，动画会暂停播放
repetitions	动画在循环播放模式下的循环次数。默认值是 Infinity
time	动画已经播放的时间。其取值在 0 到 THREE.AnimationClip.duration 之间循环
timeScale	指定当前动画的时间缩放值，它可以用来加快或者放慢动画播放的速度。如果该属性为 0，则动画会暂停播放
weight	动画对其宿主模型的形状的影响程度。若设置为 0，则动画在模型上完全失去效果；若为 1，则动画在模型上会完全发挥其设计时的效果
zeroSlopeAtEnd	设置为 true 时，当前动画的末尾会尽量与下一段动画的开头平滑过渡。默认值为 true
zeroSlopeAtStart	设置为 true 时，当前动画的开头会尽量与上一段动画的末尾平滑过渡。默认值为 true
crossFadeFrom(fadeOutAction, durationInSeconds, warpBoolean)	详细指定当前动画与上一段动画的平滑过渡参数。fadeOutAction 指定要平滑过渡的上一段动画，durationInSeconds 设定过渡时长，而当 warpBoolean 为 true 时，则会对过渡动画在时间尺度上进行进一步平滑
crossFadeTo(fadeInAction, durationInSeconds, warpBoolean)	与 crossFadeFrom 类似，该函数详细指定平滑过渡到下一段动画的参数

(续)

名 称	描 述
fadeIn(durationInSeconds)	指定在当前动画开始时， weight 值由 0 到 1 过渡的快慢
fadeOut(durationInSeconds)	指定在当前动画结束时， weight 值由 1 到 0 过渡的快慢
getEffectiveTimeScale()	在平滑过渡过程中， timeScale 和 weight 值会随时间而变化。该函数获得当前的瞬时 timeScale 值
getEffectiveWeight()	与 getEffectiveTimeScale 函数类似，该函数获得当前的瞬时 weight 值
getClip()	获得当前 THREE.AnimationAction 对象所绑定的 THREE.AnimationClip 对象
getMixer()	获得负责播放当前动画的 THREE.AnimationMixer 对象
getRoot()	获得动画的宿主模型
halt(durationInSeconds)	在 durationInSeconds 秒内将 timeScale 值逐渐降为 0
isRunning()	检查动画是否正在播放
isScheduled()	检查动画在 THREE.AnimationMixer 中是否处于激活状态。(译注：处于激活状态的动画或者正在播放，或者处于等待播放的状态。)
play()	开始播放动画
reset()	复位动画。这会将 paused 设置为 false， enabled 设置为 true， time 值设置为 0
setDuration(durationInSeconds)	设定动画完整播放一遍的时长。该函数会自动将 timeScale 设定为恰当的值，以确保动画在 durationInSeconds 秒内完整播放一遍
setEffectiveTimeScale(timeScale)	强制修改瞬时 timeScale 值
setEffectiveWeight()	强制修改瞬时 weight 值
setLoop(loopMode, repetitions)	设定循环模式和循环次数
startAt(startTimeInSeconds)	修改当前动画在 THREE.AnimationMixer 中开始播放的时间
stop()	停止播放动画
stopFading()	停止平滑过渡
stopWarping()	停止在时间尺度上的平滑过渡
syncWith(otherAction)	设定当前动画与另一个指定动画同步。这将使当前动画的 time 和 timeScale 与指定的动画保持一致
warp(startTimeScale, endTimeScale, durationInSeconds)	使 timeScale 值在 durationInSeconds 秒之内，从 startTimeScale 逐渐过渡到 endTimeScale

除了上述用于控制动画的属性和函数之外，THREE.AnimationMixer 还提供了两个可侦听的事件。可以调用混合器的 addEventListener 函数指定事件侦听函数。当一个动画已经完整播放完一遍时，侦听函数会收到“loop”事件通告；当一个动画已经彻底播放完成时，侦听函数会收到“finished”事件通告。

回顾前面的代码片段，你会注意到在模型加载完成之后，有如下两行代码：

```
geometry.computeVertexNormals();
geometry.computeMorphNormals();
```

如果不调用 `computeMorphNormals`（该函数必须在 `computeVertexNormals` 之后调用），则在动画过程中模型看起来会很粗糙。`computeMorphNormals` 可以确保在动画过程中 Three.js 有正确的光照数据来渲染模型。图 9.12 展示了在不调用 `computeMorphNormals` 时的效果。

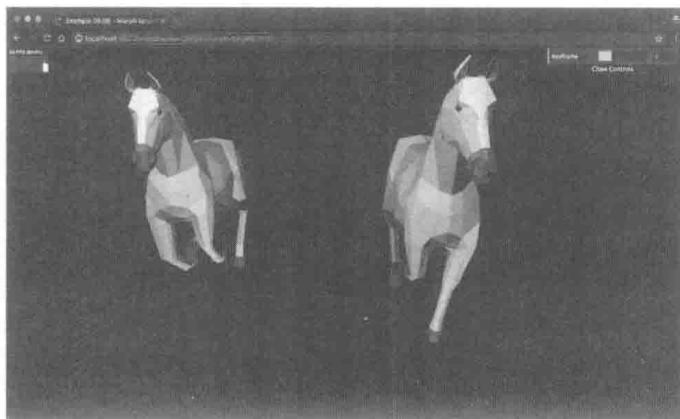


图 9.12

本节介绍的方法可以使你快速地为本身带有变形动画数据的模型设置和播放动画。下一节我们将学习如何手动为不带动画数据的模型创建动画。

## 2. 使用多个 THREE.AnimationClip 对象

我们在上一节中使用了自带动画数据的模型，因此可以直接加载模型并播放动画。此外，那个模型只包含一个 `THREE.AnimationClip` 对象（译注：意味着只包含一段动画）。在接下来的例子中，我们将为一个立方体模型手动创建包含两个 `THREE.AnimationClip` 对象的动画。第一段动画将立方体的尺寸从  $(2,2,2)$  变为  $(2,20,2)$ ；第二段动画再接着将其尺寸变为  $(40,2,2)$ 。实例程序在 `09-morph-targets-manually.html` 中，其截图如图 9.13 所示。

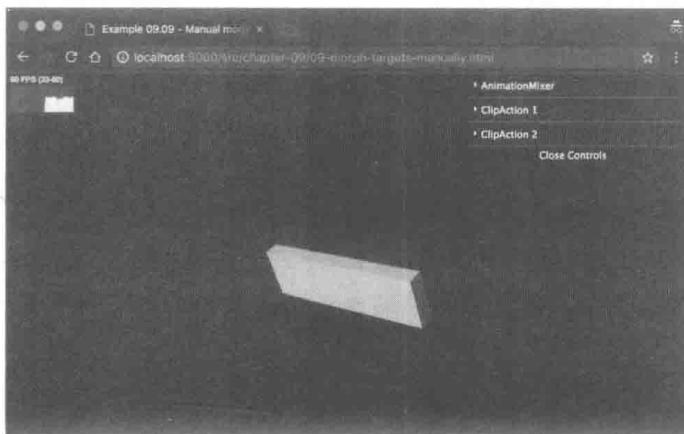


图 9.13

这个示例程序中，右边有两个菜单，分别控制 THREE.AnimationClip 和 THREE.AnimationClip。在保持默认参数设置的情况下，两段动画会同时生效。你会看到立方体的宽逐渐被拉长到 40，同时它的高被拉长到 20。在研究菜单上面的选项如何影响动画之前，我们先来看一看如何在程序里从无到有手动创建动画：

```
// initial cube
var cubeGeometry = new THREE.BoxGeometry(2, 2, 2);
var cubeMaterial = new THREE.MeshLambertMaterial({morphTargets: true,
color: 0xff0000});

// define morphtargets, we'll use the vertices from these geometries
var cubeTarget1 = new THREE.BoxGeometry(2, 20, 2);
var cubeTarget2 = new THREE.BoxGeometry(40, 2, 2);

// define morphtargets and compute the morphnormal
cubeGeometry.morphTargets[0] = {name: 't1', vertices:
cubeGeometry.vertices};
cubeGeometry.morphTargets[1] = {name: 't2', vertices:
cubeTarget2.vertices};
cubeGeometry.morphTargets[2] = {name: 't3', vertices:
cubeTarget1.vertices};
cubeGeometry.computeMorphNormals();

// create a mesh
var mesh = new THREE.Mesh(cubeGeometry, cubeMaterial);

// position the cube
mesh.position.x = 0;
mesh.position.y = 3;
mesh.position.z = 0;

// add the cube to the scene
scene.add(mesh);
mixer = new THREE.AnimationMixer( mesh );

animationClip = THREE.AnimationClip.CreateFromMorphTargetSequence('first',
[cubeGeometry.morphTargets[0],
cubeGeometry.morphTargets[1]], 1);
animationClip2 =
THREE.AnimationClip.CreateFromMorphTargetSequence('second',
[cubeGeometry.morphTargets[0],
cubeGeometry.morphTargets[2]], 1);
clipAction = mixer.clipAction( animationClip ).play();
clipAction2 = mixer.clipAction( animationClip2 ).play();
```

在上面代码中，我们定义了三个变形目标 (morphTargets)。第一个是尺寸为 (2,2,2) 的原始立方体的顶点集合，另外两个变形目标是从两个更大尺寸的立方体中获得的顶点集合。这三个变形目标最后都被添加到原始立方体模型对象的 morphTargets 属性中。在通过一些例行公事的操作将模型添加到场景中后，我们又调用 CreateFromMorphTargetSequence 函数创建了 animationClip 动画类的两个对象：animationClip1 和 animationClip2。定义动画时，指定第一个动画从模型的原始状态，即第一个变形目标 morphTargets[0] 变化到第二个变形目标 morphTargets[1]；并且指定第二个动画仍然从模型的原始状态 morphTargets[0] 开始，但是变化到第三个变形目标 morphTargets[2]。

至此，两个动画对象（THREE.AnimationClip）便可以同时播放和生效。你也可以随意修改菜单选项，看一看会对动画带来什么影响，实际上这也是理解这个动画系统如何运作的最佳实践方式。不过接下来我们需要重点了解两个最重要的属性：weight 和 timeScale。前面讲过，weight 属性决定了动画对模型的形状的影响程度。如果你将两段动画中的 weight 属性的其中之一修改为 0.5，便会立刻发现相应的变化明显减轻了；而如果设为 0，则无法再看到变化。如图 9.14 所示。

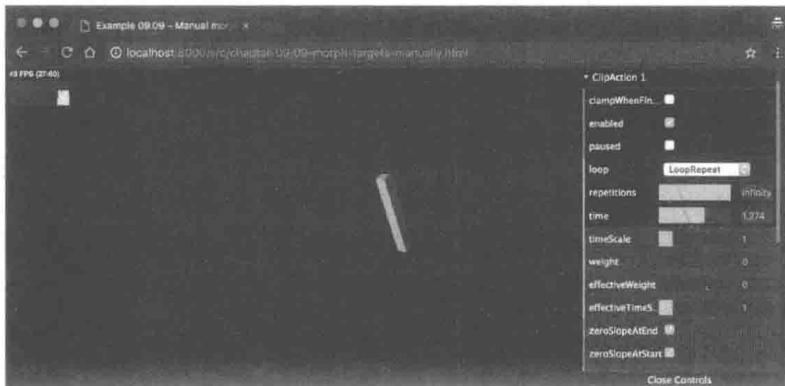


图 9.14

如果想在首尾相接的两段动画之间平滑过渡，则这一特性会变得非常重要。（为此 Three.js 还提供了几个辅助函数：fadeIn、fadeOut、crossFadeFrom 和 crossFadeTo。）另一个重要的特性便是 timeScale 属性。利用它可以调节动画的速度：如果将它设定为 2，则动画会比正常速度快一倍；相反，如果将它设定为 0，则动画会立刻暂停进行。

在后续各小节中，所有示例程序都会具有相同的控制菜单，你可以随时通过菜单来试验 THREE.ClipAction 所提供的各种属性的作用。

### 9.3.2 用骨骼和蒙皮创建动画

变形动画十分简洁。Three.js 知道所有目标顶点的位置，实现变形动画所要做的就是将每个顶点从一个位置变换到另一个位置。而骨骼和蒙皮则要复杂一些。当你使用骨骼创建动画时，你移动一下骨骼，Three.js 需要决定如何相应地移动相应的皮肤（一系列顶点）。我们使用了一个从 Blender 导出的 Three.js 格式模型来介绍骨骼（models/hand 文件夹中的 hand-1.js 文件）。这是一个手模型，上面有几块骨头，通过移动这几块骨头，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：

```
loader.load('../assets/models/hand/hand-1.js', function (geometry,
mat) {
  var mat = new THREE.MeshLambertMaterial({color: 0xF0C8C9, skinning:
true});
  mesh = new THREE.SkinnedMesh(geometry, mat);
  mesh.scale.set(15,15,15);
```

```

mesh.position.x = -5;
mesh.rotateX(0.5*Math.PI);
mesh.rotateZ(0.3*Math.PI);
scene.add(mesh);
startAnimation();
});

```

加载用于创建骨骼动画的模型与加载其他模型是一样的。我们只需要指定模型文件即可，模型文件包含了顶点、面和骨骼的定义，以及创建网格所采用的几何体。Three.js 也提供了带有蒙皮的网格对象 THREE.SkinnedMesh。需要注意的是将模型所使用材质的 skinning 属性设置为“true”，否则骨头是不会运动的。最后我们要做的是将所有骨头的 useQuaternion 属性设置为“false”。在示例中我们使用 tween 对象来处理动画，tween 对象在 StartAnimation() 函数里创建，具体代码如下所示：

```

function startAnimation() {
  tween = new TWEEN.Tween({pos: -1.5})
    .to({pos: 0}, 3000)
    .easing(TWEEN.Easing.Cubic.InOut)
    .yoyo(true)
    .repeat(Infinity)
    .onUpdate(onUpdate);

  tween.start();
}

```

使用 tween 对象，我们将 pos 变量的值从 -1.5 过渡到 0，yoyo 属性设置为“true”使得动画在运行完后会反着运行。为了保证动画能够不停地运行，将 repeat 属性设置为“Infinity”。在代码中我们还定义了一个 onUpdate 方法，这个方法用于更新每块骨头的位置。

在移动这些骨头之前，我们先来看下示例 10-bones-manually.html。效果如图 9.15 所示。



图 9.15

打开这个示例后，你会看到一只手正在做抓东西的动作。我们在 onUpdate() 方法中通过设置指部骨头绕 z 轴旋转来实现这个效果：

```

var onUpdate = function () {
  var pos = this.pos;

```

```

// rotate the fingers
mesh.skeleton.bones[5].rotation.set(0, 0, pos);
mesh.skeleton.bones[6].rotation.set(0, 0, pos);
mesh.skeleton.bones[10].rotation.set(0, 0, pos);
mesh.skeleton.bones[11].rotation.set(0, 0, pos);
mesh.skeleton.bones[15].rotation.set(0, 0, pos);
mesh.skeleton.bones[16].rotation.set(0, 0, pos);
mesh.skeleton.bones[20].rotation.set(0, 0, pos);
mesh.skeleton.bones[21].rotation.set(0, 0, pos);

// rotate the wrist
mesh.skeleton.bones[1].rotation.set(pos, 0, 0);
};

}

```

每当 `onUpdate()` 方法被调用时，相关的骨头都会被设置到 `pos` 指定的位置。为了能够知道哪些骨头被移动了，可以将 `mesh.skeleton` 属性输出在控制台上，该属性列出了所有的骨头及其名字。



Three.js 提供了一个能够展示模型骨头的辅助方法，具体代码如下：

```

helper = new THREE.SkeletonHelper( mesh );
helper.material.linewidth = 2;
helper.visible = false;
scene.add( helper );

```

更多信息可以参考示例 `10-bones-manually.html`。

启用 `THREE.SkeletonHelper` 后可以看到如图 9.16 所示的画面。在画面中，用于控制皮肤的骨骼会以直线的形式绘制出来。



图 9.16

正如你所看到的，使用骨骼动画比变形动画要复杂很多。在这个示例中我们只是改变骨头的旋转角度，除此之外我们还可以改变骨头的位置或缩放比例。在下一节中，我们将会介绍如何从外部模型中加载动画，还会使用模型来改进本节的示例，不再手动地移动这些骨头。

## 9.4 使用外部模型创建动画

在第 8 章中我们已经见过了几个 Three.js 支持的三维格式，其中有些格式也支持动画。在本节我们会介绍如下几个示例：

- 带有 JSON 导出器的 Blender：我们会在 Blender 中创建一个动画，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，我们会从 Collada 文件中加载动画，然后用 Three.js 来进行渲染。
- MD2 模型：MD2 模型是老式雷神（Quake）引擎所使用的简单格式。尽管这种格式有点过时了，但是它依然是一种很好的存储角色动画的文件格式。
- glTF 模型：该模型的全称为“GL Transmission (glTF) format”（译注：可译作“GL 传输格式”。由于其开发者为负责维护 OpenGL 的 Khronos Group，因此名称中的“GL”应该指 OpenGL 或者 WebGL 图形 API）。该格式专用于存储 3D 场景和模型，其优势在于可以最小化文件尺寸，并且可以高效地加载模型。
- FBX 模型：FBX 模型是 Mixamo 工具程序的文件格式，可以在 <https://www.mixamo.com> 找到该工具程序。Mixamo 工具程序易于使用，没有丰富的建模经验的使用者也能够用它制作模型动画。
- DirectX 模型：DirectX API 以前拥有自己专用的模型、材质和动画文件格式。虽然如今这种格式的模型已经不多见，但是 Three.js 还是可以通过 XLoader 库来加载该种模型。但是使用该种模型的方式与其他格式稍有不同，因为它将模型和动画分成多个文件存储。
- BVH 模型：Biovision (BVH) 模型为 Autodesk MotionBuilder 所使用，与其他种类的模型有些不同。该种文件里一般只有骨骼信息，而没有几何形体的信息。加载该文件之后，你甚至可以将骨骼信息应用到自己的模型上。
- SEA 模型：SEA3D 模型用于存储场景、动画、声音等数据，可以使用 SEA3D Editor (SEA3D 编辑器) 来创建和编辑该种文件。Three.js 版 SEA3D 加载器除了可以加载动画之外，还可以加载声音和物理信息。但在本节中我们将只了解如何从 SEA3D 加载动画。



Three.js 的 MMDLoader 还可以加载 MikuMikuDance 模型，但它并未包含在本章之中。其实加载该种模型的方式与其他模型并无太大区别，但是由于该模型不允许用于其他商业目的，所以本书不介绍它。

下面先从 Blender 模型开始介绍。

### 9.4.1 使用 Blender 创建骨骼动画

为了使用 Blender 创建动画，你需要加载 models 文件夹下的示例，然后找到一个名为

hand.blend 的文件并加载到 Blender 中。图 9.17 展示了示例的静态效果。

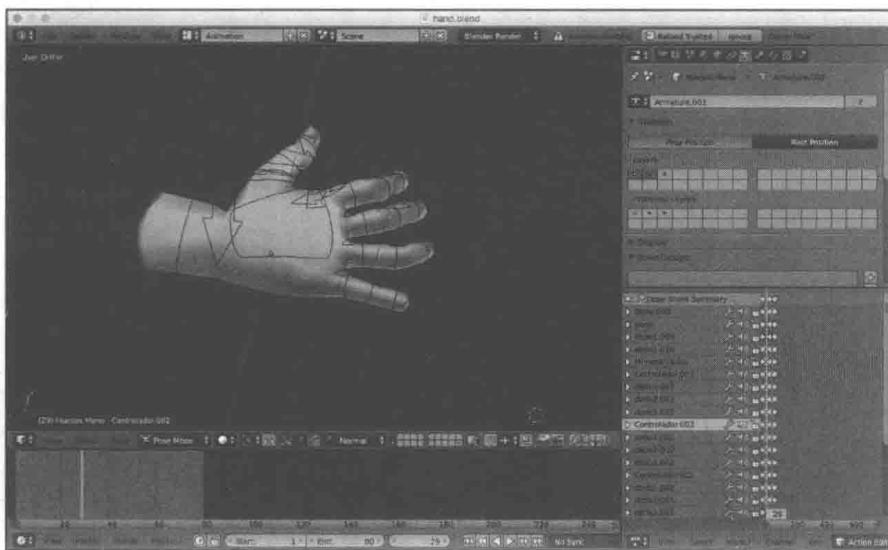


图 9.17

在本书中未涉及如何在 Blender 中创建动画，但是你需要注意以下几点：

- 模型的顶点至少要在一个顶点组中。
- Blender 中顶点组的名字必须跟所要控制的骨头的名字相对应。这样，Three.js 才知道在移动骨头时哪些顶点需要做相应的改变。
- 只有第一个 action (动作) 可以被导出。所以需要保证你想要导出的动画是第一个 action。
- 在创建关键帧时，最好选择所有的骨头，即使它们没有变化。
- 导出模型时，要保证模型处于放松状态。如果不这样，在播放动画的过程中有可能会出现不良变形<sup>⊖</sup>。
- 一次最好只导出单个模型及其动画数据，而不是整个场景。

在 Blender 中创建好动画后，可以使用我们在上一章中用过的导出器将动画导出。当使用 Three.js 的导出器导出文件时，你需要确认如图 9.18 和图 9.19 所示的属性被选中。

这样就可以将 Blender 中创建的动画以骨骼动画的方式导出，而不是变形动画。在导出骨骼动画时，骨骼的移动也会被导出，这样就可以在 Three.js 中重现这个移动：

```
var loader = new THREE.JSONLoader();
loader.load('../assets/models/hand/hand-8.json', function (result) {
    var mesh = new THREE.SkinnedMesh(result, new
        THREE.MeshNormalMaterial({skinning: true}));
    mesh.scale.set(18, 18, 18)
    scene.add(mesh);
```

<sup>⊖</sup> Blender 可以将骨骼动画模型恢复到 rest pose，具体方法请参考 Blender 的说明文档。——译者注

```
// setup the mixer
mixer = new THREE.AnimationMixer( mesh );
animationClip = mesh.geometry.animations[0];
clipAction = mixer.clipAction( animationClip ).play();
animationClip = clipAction.getClip();
});

function render() {
  ...
  var delta = clock.getDelta();
  mixer.update( delta );
  ...
}
```

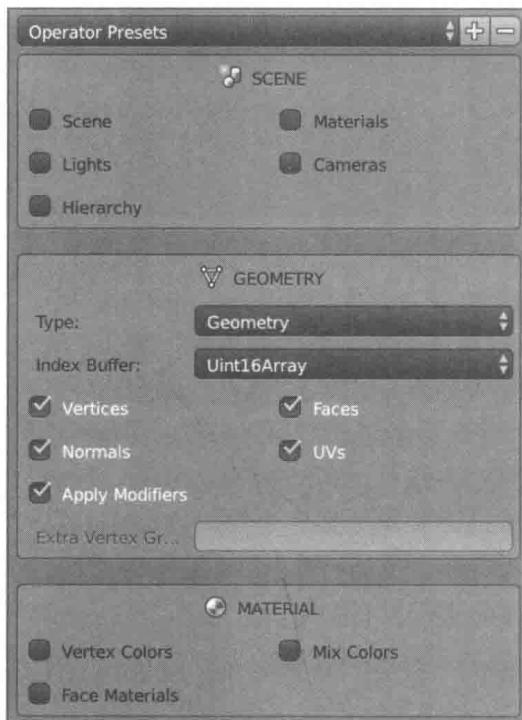


图 9.18

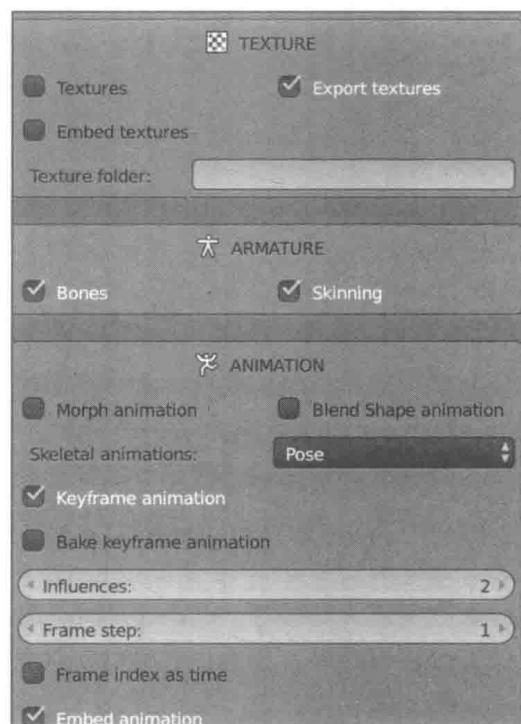


图 9.19

为了能够让动画运行起来，我们还需要创建 THREE.SkinnedMesh 对象，就前一个例子所做的，共同使用 THREE.AnimationMixer、THREE.AnimationClip 和 THREE.ClipAction 告诉 Three.js 如何运行导人的动画。要记住在渲染循环中必须调用 mixer.update() 函数。这个示例（13-animation-from-blender.html）展示的是正在动作的手，效果如图 9.20 所示。

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 Collada 模型。

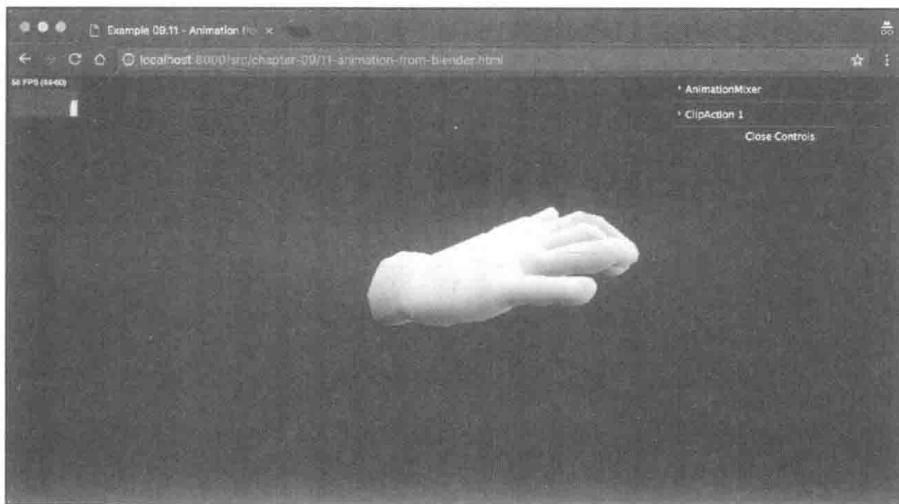


图 9.20

#### 9.4.2 从 Collada 模型加载动画

从 Collada 文件中加载模型与其他格式的文件加载模型是一样的。首先需要引入对应的 JavaScript 文件：

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

由于普通的 Collada 模型是不压缩的，因此它们的文件往往非常大。Three.js 还有一个 KMZLoader 加载器，用于加载 KMZ (Keyhole Markup language Zipped) 模型。该模型基本上就是压缩过的 Collada 模型。如需加载此类模型，只需要将 ColladaLoader 替换为 KMZLoader 即可。

然后我们创建一个加载器，并用它来加载模型文件：

```
var loader = new THREE.ColladaLoader();
loader.load('../assets/models/monster/monster.dae', function (result)
{
    scene.add(result.scene);
    result.scene.rotateZ(-0.2*Math.PI)
    result.scene.translateX(-20)
    result.scene.translateY(-20)

    // setup the mixer
    mixer = new THREE.AnimationMixer(result.scene);
    animationClip = result.animations[0];
    clipAction = mixer.clipAction( animationClip ).play();
    animationClip = clipAction.getClip();

    // add the animation controls
    enableControls();
});
```

Collada 文件不仅可以包含模型，还可以保存包含摄像机、光源和动画等的场景。使用 Collada 模型最好的方式是将 loader.load 方法的调用结果输出在控制台，然后决定使用哪些组件。在本例中，可以在控制台打印中看到类似图 9.21 所示内容。

```

▼ {animations: Array(1), kinematics: {}, library: {}, scene: Group} □
  ► animations: [AnimationClip]
  ► kinematics: {}
  ► library: {animations: {}, clips: {}, controllers: {}, images: {}, effects: {}}
  ▼ scene: Group
    castShadow: false
  ▼ children: Array(2)
    ► 0: Bone {uuid: "C265DD0E-9956-46A1-8A7E-06794B2658FF", name: "Bone1", type: "Bone", parent:
    ▼ 1: SkinnedMesh
      bindMatrix: Matrix4 {elements: Array(16)}
      bindMatrixInverse: Matrix4 {elements: Array(16)}
      bindMode: "attached"
      castShadow: false
    ▶ children: []
    drawMode: 0
    frustumCulled: true
  
```

图 9.21

Three.js 允许将 THREE.Group 对象添加到 THREE.Scene 中，因此在本例中我们暂时一次性将加载获得的整个场景（译注：即整个 THREE.Group 对象）添加到 Three.js 的场景中。如果你希望将一部分加载获得的内容丢弃，可以根据在控制台中打印出来的内容列表进行挑选然后再添加到场景中。与 Blender 方式比较，你会发现本例所使用的方法跟它基本相同，甚至渲染循环也是一样的。

加载 COLLADA 文件的效果如图 9.22 所示。

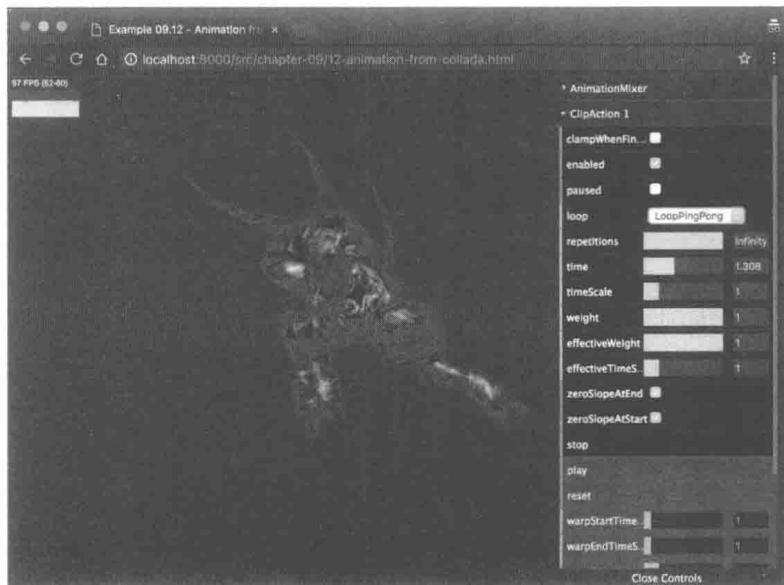


图 9.22

最后一个外部模型的例子是 MD2 文件格式。

### 9.4.3 从雷神之锤模型中加载动画

MD2 是雷神之锤中所使用的角色模型文件格式。雷神之锤是一部诞生于 1996 年的非常出色的计算机游戏。虽然该游戏的新版引擎已经不再使用该文件格式，但是仍然能够找到很多非常有趣的模型储存于 MD2 格式文件中。虽然加载 MD2 文件的方式与前面介绍的其他文件格式非常相似，但是由于 MD2 只存储几何体，因此在加载 MD2 文件时，你需要自己创建材质对象，并自行为其加载纹理资源。具体方法如下面代码片段所示：

```

var textureLoader = new THREE.TextureLoader();
var loader = new THREE.MD2Loader();
loader.load('.../assets/models/ogre/ogre.md2', function (result) {

    var mat = new THREE.MeshStandardMaterial(
        { morphTargets: true,
          color: 0xffffffff,
          metalness: 0,
          map: textureLoader.load('.../assets/models/ogre/skins/skin.jpg')
    });

    var mat2 = new THREE.MeshNormalMaterial();
    var mesh = new THREE.Mesh(result, mat);
    scene.add(mesh);

    // // setup the mixer
    mixer = new THREE.AnimationMixer(mesh);
    animationClip1 = result.animations[7];
    clipAction1 = mixer.clipAction( animationClip1 ).play();
    animationClip2 = result.animations[9];
    clipAction2 = mixer.clipAction( animationClip2 );
    animationClip3 = result.animations[10];
    clipAction3 = mixer.clipAction( animationClip3 );

    // add the animation controls
    enableControls(result);
});

```

上面代码使用了标准材质 THREE.MeshStandardMaterial 来为模型添加皮肤。并且基于加载获得的动画数据，创建了三个动画对象。由于该模型中没有骨骼信息，因此 morphTargets 属性被设置为 true，以便启用变形动画模式。示例程序 15-animation-from-md2.html 有一个选择动画的下拉菜单，可以在菜单中选择播放模型提供的所有动画，如图 9.23 所示。

### 9.4.4 使用 gltfLoader

glTF 格式近年来受到越来越多的关注。若希望更深入地了解这种文件格式，可以访问网站 <https://github.com/KhronosGroup/glTF>。glTF 格式本身侧重于优化文件尺寸以及提高资源使用效率，但在 Three.js 中通过 glTFLoader 加载器来使用这种格式，与使用其他格式



图 9.23

并没有太大区别。这种格式的新版加载器名为 THREE.GLTFLoader，它支持 2.0 版（这是目前的 glTF 标准格式）以及更高版本的 glTF 文件；而它还有一个旧版加载器，名为 THREE.LegacyGLTFLoader，专门用于加载旧版的 glTF 文件。为了使用该加载器，首先需要引入对应的 JavaScript 文件：

```
<script type="text/javascript" charset="UTF-8"
src="../../libs/three/loaders/GLTFLoader.js"></script>
```

然后按照下面方式使用 glTFLoader 加载器：

```
var loader = new THREE.GLTFLoader();
loader.load('../../assets/models/CesiumMan/CesiumMan.gltf', function
(result) {
    // correctly position the scene
    result.scene.scale.set(6, 6, 6);
    result.scene.translateY(-3);
    result.scene.rotateY(-0.3*Math.PI)
    scene.add(result.scene)

    // setup the mixer
    mixer = new THREE.AnimationMixer( result.scene );
    animationClip = result.animations[0];
    clipAction = mixer.clipAction( animationClip ).play();
    animationClip = clipAction.getClip();

    // add the animation controls
    enableControls();
});
```

glTFLoader 加载器同样会加载整个场景。如果只需要加载其中一部分内容，则必须从加载结果中选择子元素来使用。示例程序 14-animation-from-gltf.html 演示了加载和渲染该

类型的模型。渲染结果如图 9.24 所示。

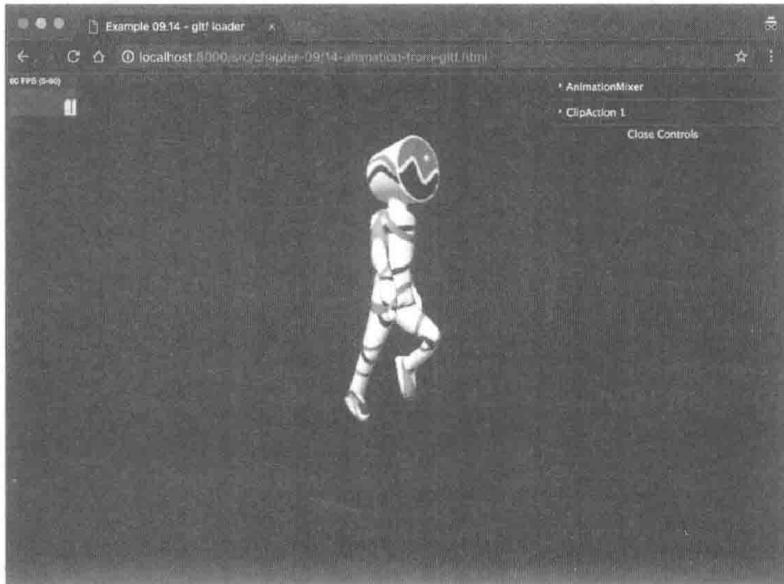


图 9.24

下一个模型文件格式是 Autodesk 的 Filmbox (FBX) 格式。

#### 9.4.5 利用 fbxLoader 显示动作捕捉模型动画

Autodesk 的 FBX 格式已经推出很长时间了，这是一种非常易于使用的格式。网站 <https://www.mixamo.com/> 上面提供了约 2500 个动画模型可供下载和使用。该网站如图 9.25 所示。

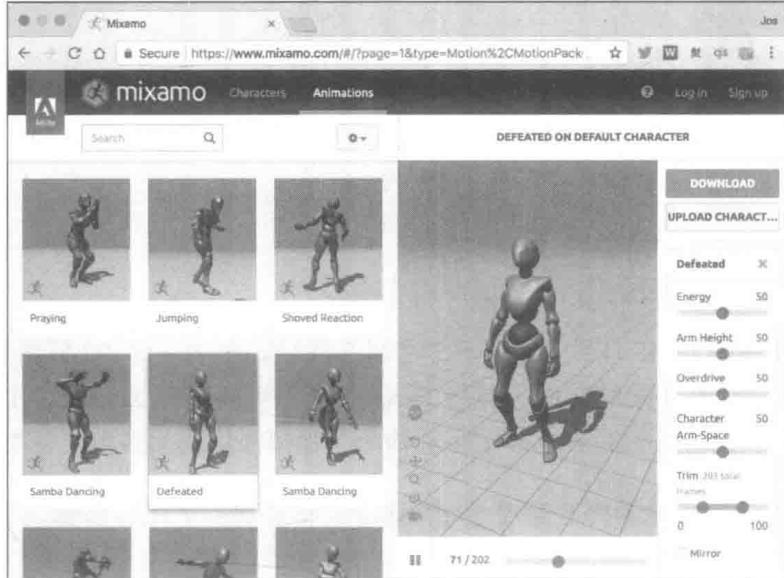


图 9.25

下载动画模型后，可以像下面这样简单地在 Three.js 中使用模型：

```
var loader = new THREE.FBXLoader();
loader.load('../assets/models/salsa/salsa.fbx', function (result) {
    result.scale.set(0.2, 0.2, 0.2);
    result.translateY(-13);
    scene.add(result)
    // setup the mixer
    mixer = new THREE.AnimationMixer( result );
    animationClip = result.animations[0];
    clipAction = mixer.clipAction( animationClip ).play();
    animationClip = clipAction.getClip();
    enableControls();
});
```

图 9.26 展示了示例程序 15-animation-from-fbx.html 的渲染画面。

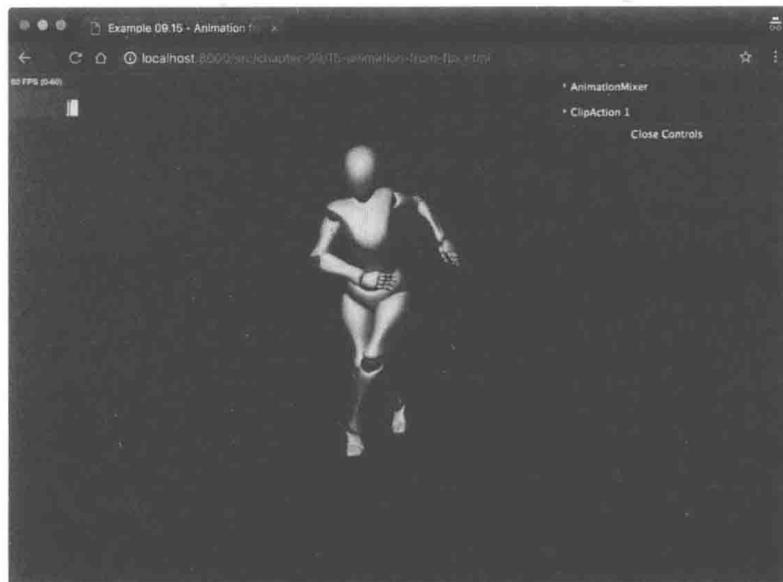


图 9.26

#### 9.4.6 通过 xLoader 加载古老的 DirectX 模型

微软公司的 DirectX 在旧版中曾有一个专属文件格式用于存储 3D 模型和动画。虽然现在已经不容易找到这种格式的文件，但是 Three.js 还是可以通过 THREE.XLoader 加载器来支持它。示例程序 16-animation-from-x.html 演示了如何使用该加载器，如图 9.27 所示。



为了能够正确播放动画，需要将 timeScale 设置为一个较高的值，否则动画会以非常缓慢的速度播放。

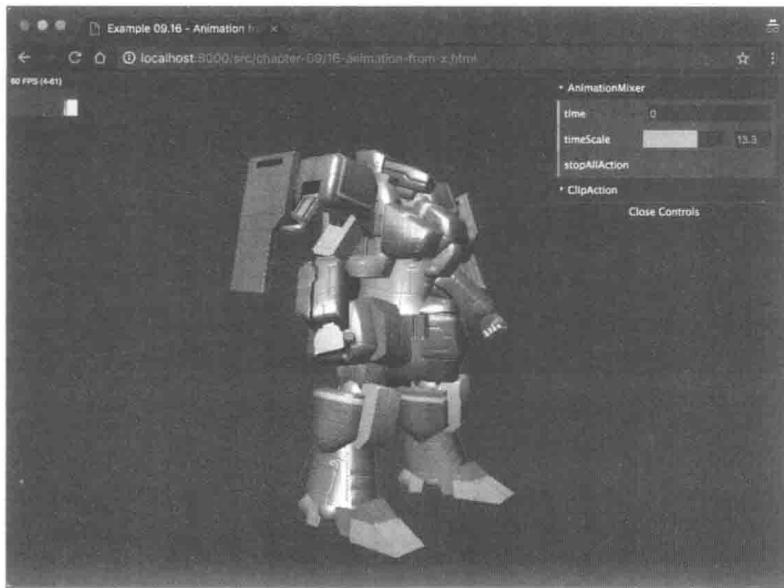


图 9.27

由于这种文件格式将模型和动画分开存储，因此加载这种文件的方法与加载其他类型的模型略有不同。下面代码演示了如何加载这种格式的模型，并播放动画：

```
var manager = new THREE.LoadingManager();
var textureLoader = new THREE.TextureLoader();
var loader = new THREE.XLoader( manager, textureLoader );
var animLoader = new THREE.XLoader( manager, textureLoader );

// we could also queue this or use promises
loader.load(["../../../assets/models/x/SSR06_model.x"], function (result) {
    var mesh = result.models[0];
    animLoader.load(["../../../assets/models/x/stand.x", { putPos: false,
putScl: false }], function (anim) {
        animLoader.assignAnimation(mesh);
        // at this point we've got a normal mesh, and can get the mixer and
clipaction
        mixer = mesh.animationMixer;
        clipAction = mixer.clipAction( "stand" ).play();
        var clip = clipAction.getClip();

        mesh.translateY(-6)
        mesh.rotateY(-0.7*Math.PI);
        scene.add(mesh)
    });
});
```

为了加载这种格式的模型，我们在上面代码中创建了两个 THREE.XLoader 加载器对象，分别用于加载模型文件和动画文件。加载完成后，需要调用 assignAnimation 函数将动画对象指定给模型对象，此后才能按照一般方法去播放动画。

#### 9.4.7 利用 BVHLoader 显示骨骼动画

BVHLoader 的特殊之处在于该加载器不返回具有动画的网格或者几何体，它只返回骨骼和动画。示例程序 17-animation-from-bvh.html 展示了这种加载器返回的内容，如图 9.28 所示。

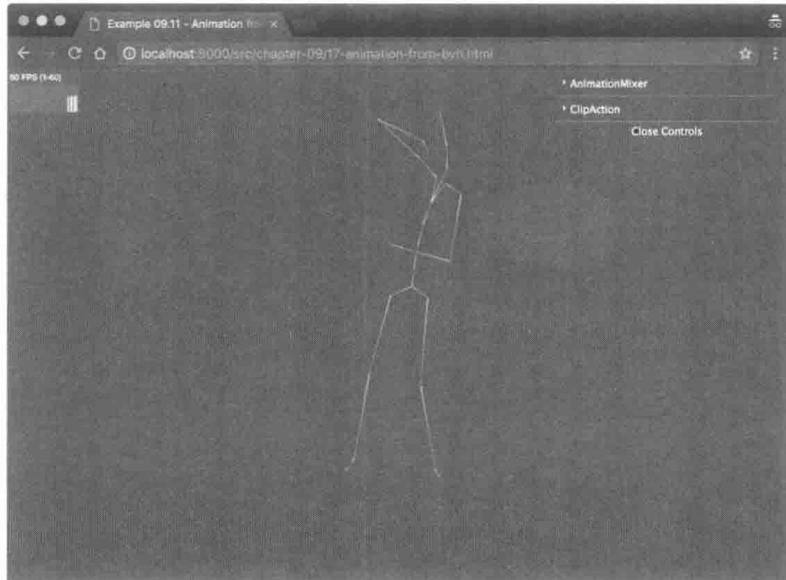


图 9.28

为了可视化，在程序中再次使用 THREE.SkeletonHelper：

```
var loader = new THREE.BVHLoader();
loader.load('../assets/models/amelia-dance/DanceNightClub7_t1.bvh',
function (result, mat) {

    skeletonHelper = new THREE.SkeletonHelper( result.skeleton.bones[ 0 ] );
    // allow animation mixer to bind to SkeletonHelper directly
    skeletonHelper.skeleton = result.skeleton;
    var boneContainer = new THREE.Object3D();
    boneContainer.translateY(-70);
    boneContainer.translateX(-100);
    boneContainer.add( result.skeleton.bones[ 0 ] );
    scene.add( skeletonHelper );
    scene.add( boneContainer );
    mixer = new THREE.AnimationMixer( skeletonHelper );
    clipAction = mixer.clipAction( result.clip ).setEffectiveWeight( 1.0
).play();
})
```

作为最后一个动画格式，我们将介绍如何加载和播放来自开源项目 SEA3D 的动画模型。

#### 9.4.8 如何重用 SEA3D 模型

SEA3D 是一个开源软件项目，它的功能很丰富，通常可以用于制作游戏、创建模型、添加动画等。在本节我们只介绍如何在 Three.js 中加载和显示用它生成的模型和动画。使用的方法和其他模型的大同小异：

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

var loader = new THREE.SEA3D({
    container: sceneContainer
});

loader.load('../assets/models/mascot/mascot.sea');
loader.onComplete = function( e ) {
    var skinnedMesh = sceneContainer.children[0];
    skinnedMesh.scale.set(0.1, 0.1, 0.1);
    skinnedMesh.translateX(-40);
    skinnedMesh.translateY(-20);
    skinnedMesh.rotateY(-0.2*Math.PI);
    scene.add(skinnedMesh);

    // and set up the animation
    mixer = new THREE.AnimationMixer( skinnedMesh );
    animationClip = skinnedMesh.animations[0].clip;
    clipAction = mixer.clipAction( animationClip ).play();
    animationClip = clipAction.getClip();
    enableControls();
};

};

}

// and set up the animation
mixer = new THREE.AnimationMixer( skinnedMesh );
animationClip = skinnedMesh.animations[0].clip;
clipAction = mixer.clipAction( animationClip ).play();
animationClip = clipAction.getClip();
enableControls();
```

上述代码的主要特殊之处在于，当我们创建 THREE.SEA3D 加载器对象的时候，就要同时向它提供场景容器 THREE.Scene 的对象。此外，加载状态回调函数也不是在调用 load 函数时提供，而是直接向 onComplete 属性提供我们的回调函数。一旦场景加载完成，后续的操作便和一般方法相一致了。

示例程序 chapter-09/18-animation-fromsea.html 的截图如图 9.29 所示。

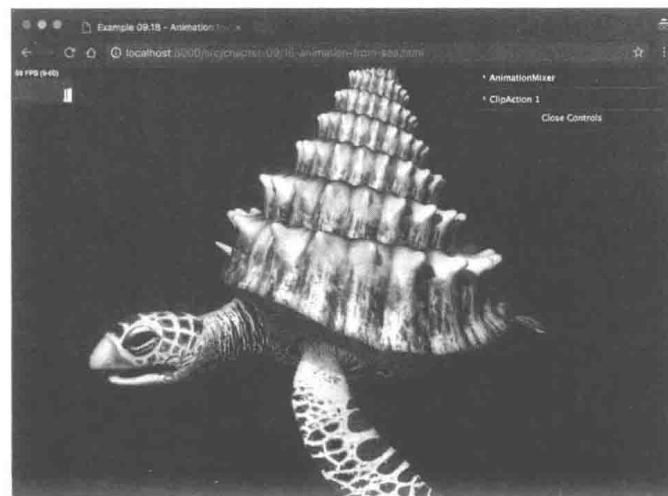


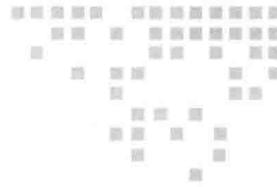
图 9.29

从外部资源加载动画的方法到这里就介绍完成了。

## 9.5 总结

在本章中，我们看到了几种不同的让场景动起来的方法。我们从基本的动画技巧开始，然后是摄像机的移动和控制，最后是用变形目标和骨骼来制作动画。通过本章的学习，我们发现只要有了渲染循环，添加动画是很简单的，只需要修改网格的属性即可，Three.js 会在下次渲染时更新网格。

在前面的章节里，我们介绍了各种可以用来覆盖对象的材质，也介绍了如何修改材质的颜色、光泽和不透明度，但是我们还没有详细介绍如何在材质中使用外部图片（也叫纹理）。使用纹理，可以使得对象看上去像是用木料、金属和石头等做出来的。下一章，我们将会介绍纹理的各个方面，以及如何在 Three.js 中使用它们。



## 第 10 章

# 加载和使用纹理

*Chapter 10*

在第 4 章中我们介绍了 Three.js 中的各种材质，但是没有介绍如何将纹理应用到网格上。在本章中我们将会从以下几个方面来讨论这个主题：

- 在 Three.js 中加载纹理并应用到网格上。
- 使用凹凸贴图、法线贴图和移位贴图为网格添加深度和细节。
- 使用光照贴图创建假阴影。
- 使用高光贴图、金属感贴图和粗糙度贴图为网格的特定部分设定闪亮度。
- 使用 Alpha 贴图使物体部分透明。
- 使用环境贴图为材质增加细节反射。
- 微调和自定义网格的 UV 映射。
- 将 HTML5 的画布和视频作为纹理的输入。

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 将纹理应用于材质

在 Three.js 中可以使用纹理实现不同的效果。你可以使用它们来定义网格的颜色，也可以使用它们来定义高光、凹凸和反光。但是我们首先要介绍的是最基本的用法：使用纹理为网格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作为贴图被添加在材质上，当你使用这样的材质创建网格时，网格的颜色则来源于纹理。

可以用如下的方式来加载纹理并应用于网格：

```
var textureLoader = new THREE.TextureLoader();
textureLoader.load("../assets/textures/general/metal-rust.jpg")
```

在这段代码中，我们使用 THREE.TextureLoader 从指定的位置加载图片文件。图片格式可以是 PNG、GIF 或 JPEG，在本章的后面还将介绍如何加载其他纹理格式。请注意，纹理的加载是异步的：如果要加载的纹理较大，而程序在纹理加载完成之前开始渲染场景，则会在最开始的瞬间看到场景中的一些物体表面没有贴图。如果希望等待纹理加载完成，可以像下面这样为 THREE.TextureLoader.load() 函数提供回调函数：

```
var textureLoader = new THREE.TextureLoader();
textureLoader.load("../assets/textures/general/metal-rust.jpg",
    onLoadFunction, onProgressFunction,
    onErrorFunction)
```

在上面代码中，load 函数接收三个回调函数作为参数：onLoadFunction 在纹理加载完成时被调用；onProgressFunction 可以随时汇报加载进度；onErrorFunction 在纹理加载或解析出故障时被调用。

可以使用任何你喜欢的图片来作为纹理使用，但是为了达到最好的效果，最好使用长宽大小为 2 的次方的正方形图片，例如大小为  $256 \times 256$ 、 $512 \times 512$ 、 $1024 \times 1024$  的图片最合适。图 10.1 展示的就是大小为 2 的次方的纹理图片。



图 10.1

由于纹理通常需要放大或缩小，所以纹理上的像素不会一一对地映射到面的像素上。为此，WebGL 和 Three.js 提供了各种不同的选项，你可以设置 magFilter 属性来指定纹理如何放大，设置 minFilter 属性来指定纹理如何缩小。这些属性可以设置成表 10.1 中的值。

表 10.1

名 称	描 述
THREE.NearestFilter (最邻近过滤)	这个过滤器会将纹理上最近的像素颜色应用于面上。在放大时，会导致方块化；在缩小时会丢失很多细节
THREE.LinearFilter (线性过滤)	这个过滤器比较高级，最终的颜色是由周围四个像素值来决定的。这样虽然在缩小时仍会丢失一些细节，但是在放大时会平滑很多，方块化也比较少出现

除了上述值，我们还可以使用 mipmap。mipmap 是把纹理按照 2 的倍数进行缩小。这些图片是在加载纹理时创建的，可以用于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，只需要几步就可以达到更好的过滤效果。mipmap 的纹理过滤模式如表 10.2 所示。

表 10.2

名 称	描 述
THREE.NearestMipMapNearestFilter	选择最邻近的 mip 层，并执行前表中的最邻近过滤。虽然放大时仍然会有方块化，但是缩小时效果会好很多
THREE.NearestMipMapLinearFilter	选择最邻近的两个 mip 层，并分别在这两个 mip 层上运行最邻近过滤获取两个中间值，最后将这两个中间值传递到线性过滤器中获取最终值
THREE.LinearMipMapNearestFilter	选择最邻近的 mip 层，并执行前表中的线性过滤
THREE.LinearMipMapLinearFilter	选择最邻近的两个 mip 层，并分别在这两个 mip 层上运行线性过滤获取两个中间值，最后将这两个中间值传递到线性过滤器中获取最终值

如果没有明确指定 magFilter 和 minFilter 属性的值，Three.js 会将 THREE.LinearFilter 作为 magFilter 属性的默认值，将 THREE.LinearMipMapLinearFilter 作为 minFilter 属性的默认值。在我们的示例中使用的就是上述默认值。关于基于纹理的示例可以在文件 01-basic-texture.html 中找到，效果如图 10.2 所示。

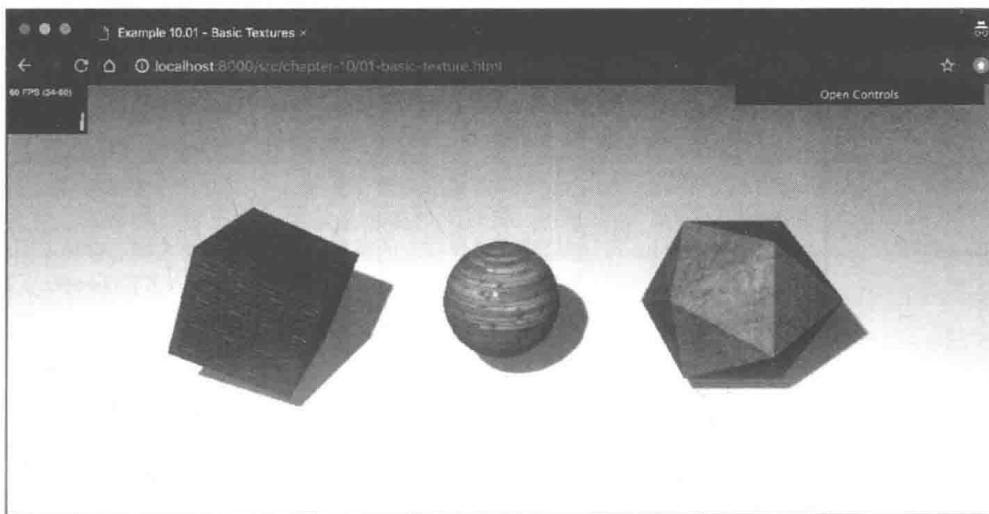


图 10.2

在这个示例里，我们加载了多个纹理并把它们应用到不同形状的图形上。你可以使用右边的菜单改变任何材料特性。

如你所看到的，纹理可以很好地贴合在图形上，这个效果是 Three.js 在创建几何体时通过 UV 贴图来实现的（更多内容将在本章讨论）。基于 UV 贴图，我们可以告诉渲染器将纹理的哪部分应用到指定的面上。本章稍后会介绍 UV 贴图的详细信息。

除了使用 THREE.TextureLoader 方法加载标准格式的图片，Three.js 还提供了一些自定义的加载器，以此来加载其他格式的纹理文件。表 10.3 列出了可供选择的加载器。

表 10.3

加载器	描述
THREE.DDSLoader	<p>该加载器可以加载 DirectDraw Surface 格式的纹理文件。这是由微软公司控制版权的一种压缩纹理格式。首先在你的 HTML 页面中引入 DDSLoader.js 文件，然后像下面这样加载纹理：</p> <pre>var textureLoader = new THREE.DDSLoader(); var texture = textureLoader.load('../..../assets/textures/dds/test-dxt1.dds');</pre> <p>示例代码可以在 02-basic-texture-dds.html 中找到。该加载器会使用 THREE.CompressedTextureLoader</p>
THREE.PVRLoader	<p>Power VP 也是一种私有版权的压缩纹理文件格式。Three.js 支持 Power VR 3.0 版本的文件。在 HTML 页面中引入 PVRLoader.js 文件后，像下面这样加载纹理：</p> <pre>var textureLoader = new THREE.PVRLoader(); var texture = textureLoader.load('../..../assets/textures/pvr/tex_base.pvr');</pre> <p>请注意，不是所有设备上的浏览器都支持 PVR 格式。如果在桌面版 Chrome 浏览器中加载这种格式会在控制台看到错误信息：WEBGL_compressed_texture_pvrtc extension not supported，但是这种格式却在 iOS 系统上被广泛使用。示例代码可以在 03-basic-texture-pvr.html 中找到</p>
THREE.TGALoader	<p>Targa 是一种在 3D 软件中仍被广泛使用的栅格图像文件格式。（译注：古老的 Targa 格式的历史可以追溯到 20 世纪 80 年代初。）在 HTML 页面引入 TGALoader.js 文件后，像下面这样加载纹理：</p> <pre>var loader = new THREE.TGALoader(); var texture = loader.load('..../..../assets/textures/tga/dried_grass.tga');</pre> <p>示例代码可以在 04-basic-texture-tga.html 中找到</p>
THREE.KTXLoader	<p>Khronos Texture (KTX) 文件格式来自于 Khronos 工作组。目前该工作组负责开发和维护 OpenGL 和 WebGL。该格式的初衷是提供一个压缩纹理格式，使其可以被 WebGL 直接使用，以便尽量降低处理它的额外开销。KTX 格式拥有多种不同的编码，不同的硬件对编码的支持有可能不同。在 HTML 页面中引入 KTXLoader.js 文件后，像下面这样加载纹理：</p> <pre>(if ( renderer.extensions.get( 'WEBGL_compressed_texture_astc' ) !== null) return "astc";   if ( renderer.extensions.get( 'WEBGL_compressed_texture_etc1' ) !== null) return "etc1";   if ( renderer.extensions.get( 'WEBGL_compressed_texture_s3tc' ) !== null) return "s3tc";   if ( renderer.extensions.get( 'WEBGL_compressed_texture_pvrtc' ) !== null) return "pvrtc"; }  var ktxTextureLoader = new THREE.KTXLoader(); var texture  switch (determineFormat()) {   case "astc":   texture = ktxTextureLoader.load('..../..../assets/textures/ktx/disturb_ASTC4x4.ktx');   break;   case "etc1":   texture = ktxTextureLoader.load('..../..../assets/textures/ktx/disturb_ETC1.ktx');   break;   case "s3tc":   texture = ktxTextureLoader.load('..../..../assets/textures/ktx/disturb_BC1.ktx');   break;   case "pvrtc":   texture = ktxTextureLoader.load('..../..../assets/textures/ktx/disturb_PVR2bpp.ktx');   break; }</pre> <p>示例代码可以在 06-basic-texture-tga.html 中找到。（译注：上述代码先查询 WebGL 当前所支持的 KTX 编码，然后选择加载特定编码的 KTX 文件。）</p>

上面的纹理都是直接存储或者压缩存储的普通图片。除了这些普通图片之外，Three.js 还支持 HDR 图像（高动态范围图像）。相比普通图片，HDR 图像包含了更高的亮度范围。它的亮度范围更接近人眼的光学特性。Three.js 支持 EXR 和 RGBE 格式。由于 HDR 图像所包含的亮度范围大于屏幕能够支持的范围，因此在后续的示例程序中，你可以尝试微调 Three.js 对 HDR 图像的渲染参数，并观察渲染效果的变化。这种参数调节可以通过设置 THREE.WebGLRenderer 类的属性来实现。表 10.4 列出了相关属性。

表 10.4

加载器	描述
toneMapping (色调映射)	<p>该属性控制 Three.js 如何将 HDR 色彩域映射到屏幕所支持的色彩域上，有如下选项可用：</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> THREE.NoToneMapping</li> <li><input type="checkbox"/> THREE.LinearToneMapping</li> <li><input type="checkbox"/> THREE.ReinhardToneMapping</li> <li><input type="checkbox"/> THREE.Uncharted2ToneMapping</li> <li><input type="checkbox"/> THREE.CineonToneMapping</li> </ul> <p>在 HDR 的示例程序里，你可以一一尝试上面列出的映射方法，并观察它们对 HDR 纹理渲染效果的影响。默认映射为 THREE.LinearToneMapping</p>
toneMappingExposure	该属性控制色调映射的曝光级别，可用于微调渲染场景中纹理贴图的色彩
toneMappingWhitePoint	该属性设置色调映射中的白点值

表 10.5 列举了如何在程序中使用 THREE.EXRLoader 和 THREE.RGBELoader 加载器，以及如何设置 toneMapping、toneMappingExposure 和 toneMappingWhitePoint 属性。

表 10.5

加载器	描述
THREE.EXRLoader	<p>EXR 文件是 Industrial Light &amp; Magic 公司为存储 HDR 图像而开发的图像文件格式。在 HTML 页面中引入 EXRLoader.js 文件后，像下面这样加载纹理：</p> <pre>var loader = new THREE.EXRLoader(); exrTextureLoader.load('../assets/textures/exr/Rec709.exr');</pre> <p>示例代码可以在 06-basic-texture-exr.html 中找到</p>
THREE.RGBELoader	<p>RGBE 文件是 Radiance 渲染系统的图像文件格式。在 HTML 页面中引入 RGBELoader.js 文件后，像下面这样加载纹理：</p> <pre>var hdrTextureLoader = new THREE.RGBELoader(); hdrTextureLoader.load('../assets/textures/hdr/dani_cathedral_oBBC.hdr',   function(texture, metadata) {     texture.encoding = THREE.RGBEEncoding;     texture.flipY = true;     ... and use the texture });</pre> <p>加载 RGBE 纹理时，必须将加载器返回的纹理对象的 encoding 属性设定为 THREE.RGBEEncoding，否则将无法正确渲染。此外还需要将 flipY 属性设置为 true，否则图像会上向颠倒。示例代码可以在 07-basic-texture-rgbe.html 中找到。</p>

在上述示例中，我们已经用纹理来定义网格中像素的颜色，除此之外纹理还有很多用途。在下面的两个示例中，我们将使用纹理来为材质添加阴影效果和为网格添加凹凸的褶皱效果。

### 10.1.2 使用凹凸贴图创建褶皱

凹凸贴图用于为材质添加厚度。凹凸贴图的使用效果可以参看示例 02-bump-map.html，效果如图 10.3 所示。

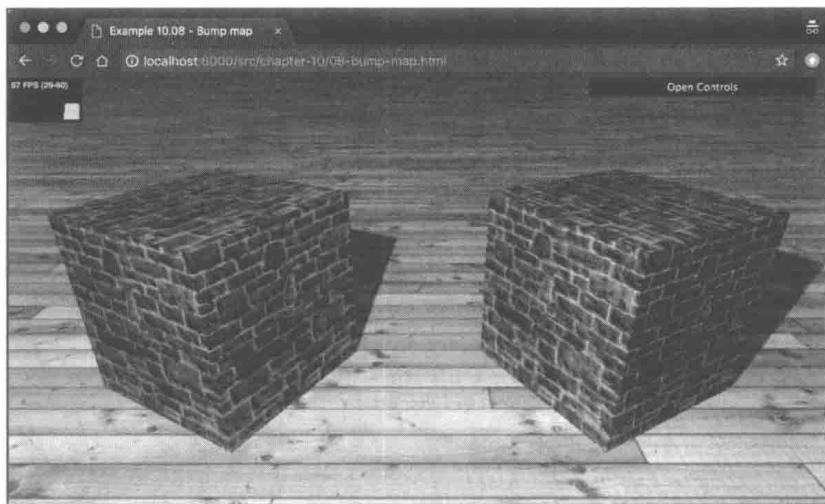


图 10.3

在该示例中可以发现：与左侧那面墙相比，右侧的墙有更多的细节，而且看上去也更厚。这是通过为材质设置额外的纹理（凹凸贴图）来实现的：

```
var cubeMaterial = new THREE.MeshStandardMaterial({
    map: textureLoader.load("../assets/textures/stone/stone.jpg"),
    bumpMap: textureLoader.load("../assets/textures/stone/stone-
bump.jpg"),
    metalness: 0.2,
    roughness: 0.07
});
```

在这段代码中可以看到，除了设置 map 属性，我们还设置了 bumpMap 属性。另外，通过 bumpScale 属性，我们可以设置凹凸的高度（如果值为负数，则表示的是深度）。本例中使用的纹理如图 10.4 所示。

这里的凹凸贴图是一张灰度图，当然你也可以使用彩色图。像素的密集程度定义的是凹凸的高度，但是凹凸图只包含像素的相对高度，没有任何倾斜的方向信息。所以使用凹凸贴图所能表达的深度信息有限，要想实现更多细节可以使用法向贴图。

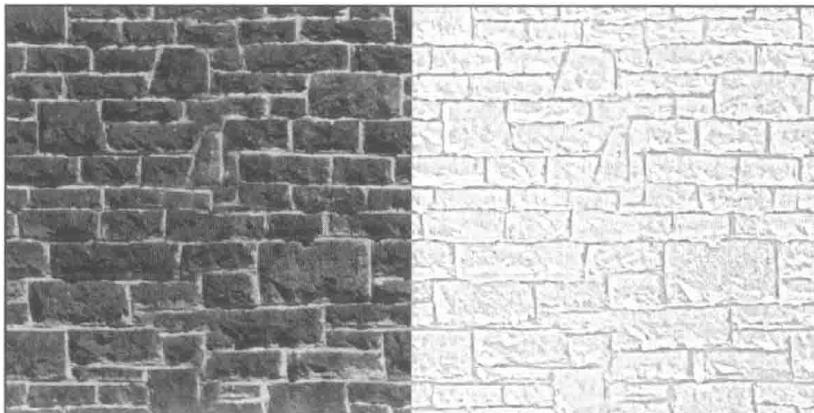


图 10.4

### 10.1.3 使用法向贴图创建更加细致的凹凸和褶皱

法线贴图保存的不是高度信息，而是法向量的方向。简单来讲，使用法向贴图只需要使用很少的顶点和面就可以创建出细节很丰富的模型。例如示例 09-normal-map.html，效果如图 10.5 所示。



图 10.5

如图 10.5 所示，右侧的图像细节更加丰富。当光源围绕方块移动时，你会看到纹理会对光源做出很自然的反应。这样会使得模型看上去很真实，而且只需要一个简单的模型和几个纹理就可以实现。下面的代码展示的就是如何在 Three.js 中使用法向贴图：

```
var cubeMaterial = new THREE.MeshStandardMaterial({
  map: textureLoader.load("../assets/textures/general/plaster.jpg"),
  normalMap: textureLoader.load("../assets/textures/general/plaster-
```

```
normal.jpg"),
    metalness: 0.2,
    roughness: 0.07
});
```

法向贴图的使用方法和凹凸贴图是一样的。只是这次我们将 `normalMap` 属性设置为法向纹理。我们还可以设置 `normalScale` 属性为 `mat.normalScale.set(1,1)` 来指定凹凸的程度，通过这两个参数，你可以沿着 `x` 轴和 `y` 轴进行缩放，但是最好的方式是将它们的值设置成一样。需要注意的是，如果设置的值为负数，那么高度就会反转。图 10.6 展示的就是纹理（左图）和法向贴图（右图）。

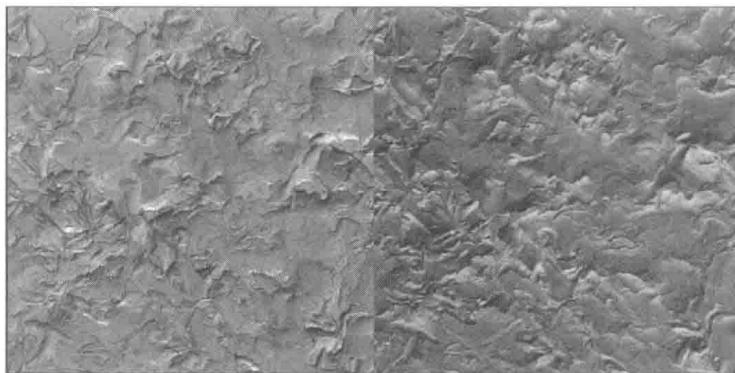


图 10.6

使用法向贴图的最大问题是它们很难创建，需要使用比如 Blender 和 Photoshop 这样的特殊工具。这些工具可以将高分辨率的效果图或者纹理作为输入来创建法向贴图。

使用法向贴图和凹凸贴图来增加物体表面细节时，不需要改变模型的实际形状；所有顶点都保持在原始位置不变。这些贴图利用场景中的光照来制造伪深度和细节。除了上面两种方法，Three.js 还提供了第三个方法来基于贴图为模型表面增加细节，这就是使用移位贴图（displacement map）。

#### 10.1.4 使用移位贴图来改变顶点位置

在 Three.js 里有一种纹理贴图可以用于修改模型的顶点。法向贴图和凹凸贴图都只能在物体表面生成一种凹凸不平的假象，而移位贴图则能够根据贴图的内容，真正改变模型的形状。移位贴图的使用方法与其他贴图类似：

```
var textureLoader = new THREE.TextureLoader();
var sphere = new THREE.SphereGeometry(8, 180, 180)
var sphereMaterial = new THREE.MeshStandardMaterial({
  map: textureLoader.load("../assets/textures/w_c.jpg"),
  displacementMap: textureLoader.load("../assets/textures/w_d.png"),
  metalness: 0.02,
  roughness: 0.07,
  color: 0xffffffff
});
```

上述代码加载一个如图 10.7 所示的纹理贴图。

图中越亮的颜色会使顶点移位越远。运行示例程序 10-displacement-map.html 可以看到模型的形状已经根据贴图而改变，效果如图 10.8 所示。（译注：如果想与法向贴图和凹凸贴图的效果对比，则应该注意模型的轮廓部分。当使用法向贴图和凹凸贴图时，无论物体表面看起来如何凹凸不平，但其轮廓依然是平整的。这其实是这两种方法的一个缺憾，而移位贴图在这方面的表现则是完美的。）

除了给 displacementMap 属性指定纹理对象之外，displacementScale 和 displacementOffset 两个属性也可以用来控制顶点的移位程度。使用移位贴图时还有最后一点需要特别注意：模型必须具有大量顶点，否则其顶点移位效果看起来会与移位贴图并不相像。这是因为顶点过少的模型没有足够的顶点可以移动。

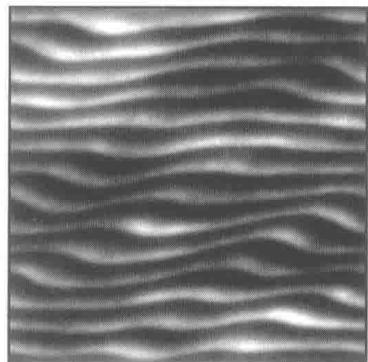


图 10.7

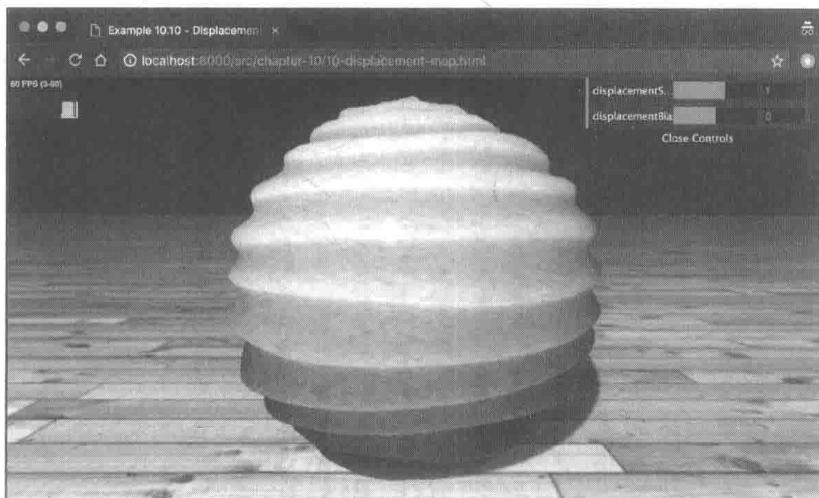


图 10.8

### 10.1.5 用环境光遮挡贴图实现细节阴影

在前面章节中我们介绍了在 Three.js 中使用阴影的方法。只要正确设定了模型的 castShadow 和 receiveShadow 属性，并且也正确地设定了光源的阴影摄像机，则 Three.js 就能够渲染出阴影效果。然而，在渲染循环里重复的渲染阴影是一个负担沉重的操作。如果场景中的光源或物体持续运动，则我们别无选择，只能使用这种方法。但是在更多的情形里，场景中总是有静止不动的光源和物体，这意味着投射在物体上的阴影也不会变化，因为如果能够计算一次阴影数据并在渲染循环里重复利用，那将是一个好主意。为了做到这一点，Three.js 提供了两种不同的专用贴图：环境光遮挡贴图和光照贴图。我们将在本节和

下一节分别介绍这两种贴图。

环境光遮挡技术用于决定模型的哪一部分暴露于环境光之中。在 Blender 或类似的软件里，环境光通常被当作半球光源或者平行光源来考虑。在这种情况下，虽然模型的大部分表面都能接收到环境光，但它们接收光线的多少仍然有差别。我们以一个站立的人物模型为例，头顶往往接收的环境光更多，而胳膊下侧则接收的更少。这种光照的差异可以被渲染（又称烘焙）到一张纹理贴图上，然后与颜色贴图混合在一起应用到模型上。这样便可以避免在渲染循环中重复计算光照差异。图 10.9 展示在 Blender 中烘焙环境光遮挡贴图的屏幕截图。



图 10.9

获得环境光遮挡贴图后，可以将它设置到模型的 `aoMap` 属性上。这样 Three.js 就可以根据贴图中的信息来决定模型的特定部分受 THREE.AmbientLight 光源影响的程度。下面的代码片段展示使用环境光遮挡贴图的方法。

```
var textureLoader = new THREE.TextureLoader();
var material = new THREE.MeshStandardMaterial({
  aoMap: textureLoader.load("../assets/models/baymax/ambient.png"),
  aoMapIntensity: 2,
  color: 0xffffffff,
  metalness: 0,
  roughness: 1
});
```

可以像加载其他纹理一样使用 `THREE.TextureLoader` 来加载环境光遮挡贴图，并将它设置到材质的正确属性上。类似地，可以通过调节 `aoMapIntensity` 属性来指定环境光遮挡贴图的影响程度。为了使这一机制正常运转，我们还需要多做一步工作。前面已经介绍过，UV 贴图实质上就是指定模型上的哪一部分需要被映射到纹理的相应位置。对于环境光遮挡贴图以及下一个例子中将要介绍的光照贴图来说，Three.js 使用与其他纹理贴图不同的 UV 贴图

来进行纹理采样。但在这个例子中，我们暂时直接复制其他纹理贴图的 UV 数据，请记住，在使用 aoMap 属性或者 lightMap 属性时，对于光照贴图来说，需要使用 faceVertexUvs[1]。

```
geometry.faceVertexUvs.push(geometry.faceVertexUvs[0]);
```

在 11-aomap.html 中可以找到环境光遮挡贴图的示例程序。如图 10.10 所示。

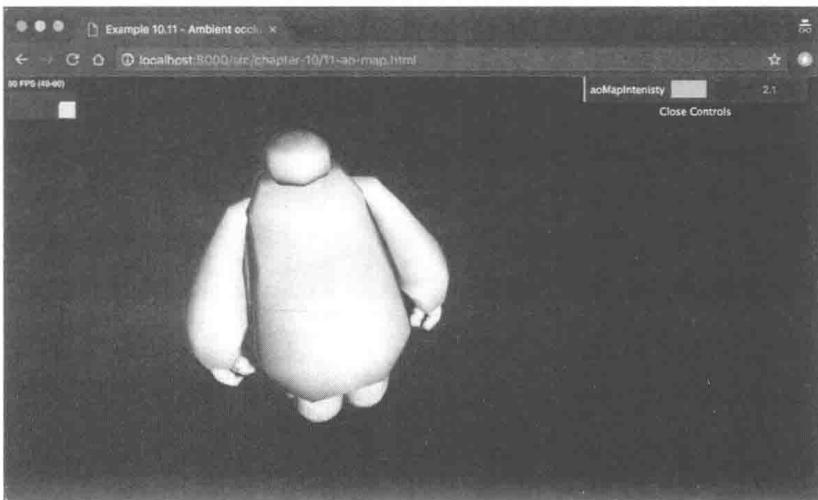


图 10.10

可以看出上图中的模型有阴影效果，但这一效果完全使用环境光遮挡贴图来实现。场景中的光源并不投射任何阴影。

环境光遮挡贴图的作用是指出模型上哪些部分处于阴影中，应该从环境光中接受较少的光照。与环境光遮挡贴图作用相反的是光照贴图。该贴图用来决定模型的那些部分需要补充更多光照。

### 10.1.6 用光照贴图产生假阴影

本节介绍光照贴图。光照贴图里面的信息用于指出一个模型的特定部分应该从场景中接收多少光照。换句话说，模型的光照信息被预先烘焙到了纹理贴图中。有很多 3D 图形软件可以用于烘焙光照贴图。图 10.11 展示了用 Blender 烘焙光照贴图。

在 Blender 界面截图中有本节示例程序将要使用的光照贴图。在编辑窗口的右侧显示的光照贴图将用于渲染地面。从图中可以看出，大部分地面都被白色光照亮，但上面有一处较暗的阴影部分。这是因为在地面上将要摆放一个模型，它会在其附近的地面上遮挡一些光照。光照贴图的代码与上一节中的环境光遮挡贴图的代码相似：

```
plane.geometry.faceVertexUvs.push(plane.geometry.faceVertexUvs[0]);
plane.material = new THREE.MeshBasicMaterial({
  map: textureLoader.load("../assets/textures/general/floor-wood.jpg"),
  lightMap:
  textureLoader.load("../assets/textures/lightmap/lightmap.png"),
});
```

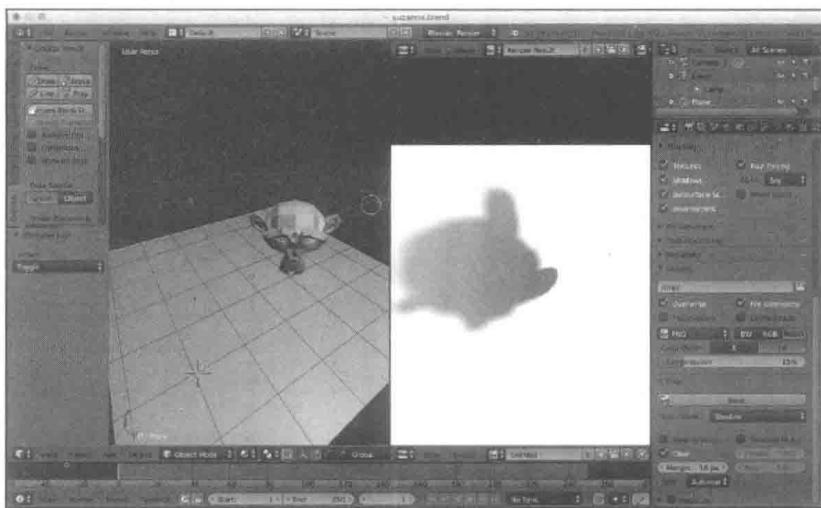


图 10.11

我们再次向 Three.js 指定一套独立的 `faceVertexUvs`，并且使用 `THREE.TextureLoader` 来加载纹理图片。在本例中，地面的颜色纹理和光照贴图均使用了简单图片。示例程序的代码可以在 `12-light-map.html` 中找到，如图 10.12 所示。



图 10.12

从上图可以看出，预先烘焙的光照贴图可以为场景中的模型产生相当好的阴影投射效果，但是请记住，烘焙到纹理贴图中的阴影、光照以及环境光遮挡信息只能用于静态场景，或者场景中静态的物体。一旦光源或物体发生移动或者改变，就不得不实时计算阴影了。

### 10.1.7 金属光泽度贴图和粗糙度贴图

前面章节介绍过 Three.js 提供的默认材质：`THREE.MeshStandardMaterial`。这个优秀

的材质既可用于生成闪亮的金属质感表面，也可以通过调节粗糙度属性来生成木质或者塑料质感的表面。实际上通过仔细调节该材质的 metalness 和 roughness 两个属性，我们可以生成大部分所需的表面质感。两个属性除了可以直接用数值来设定之外，也可以通过纹理贴图来设置。如果希望在一个表面粗糙的物体上指定一些闪亮的局部，则可以为 metalnessMap 属性设置一张金属质感贴图。或者相反的，若希望在一个光滑的物体上指定一些粗糙的局部，则可以在 roughnessMap 属性上使用纹理贴图来实现。当使用纹理贴图来设置这两个属性时，在模型的具体位置上，metalness 和 roughness 两个属性的实际值等于属性值本身与相应的贴图中的值的乘积。在示例程序 13-metal-roughness-map.html 中可以看到上面提及的几个属性的典型应用情形。示例程序的截图如图 10.13 所示。

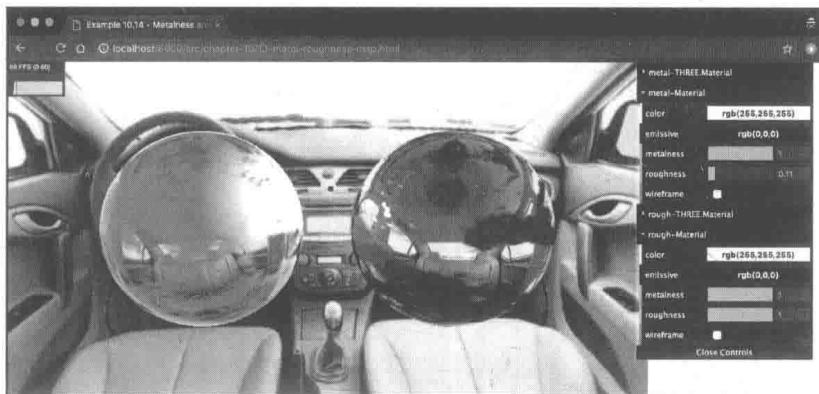


图 10.13

上面示例程序所涉及的内容稍微有些超前：除了使用金属光泽度贴图和粗糙度贴图之外，还使用环境贴图在物体表面产生了镜面反射效果。在环境贴图的基础上，金属光泽度属性值越高的物体，表面的反射越清晰，粗糙度值越高的物体，反射越浑浊。画面中右边的球在 roughnessMap 属性上使用了粗糙度贴图，从而使球面显得有些锈迹斑斑。而左边的球在 metalnessMap 属性上使用了金属光泽度贴图，其产生的效果也正相反：整个球面是粗糙的，只有一些区域被磨得很亮，显示出金属表面特有的镜面反射效果。上面示例程序所使用的贴图如图 10.14 所示。

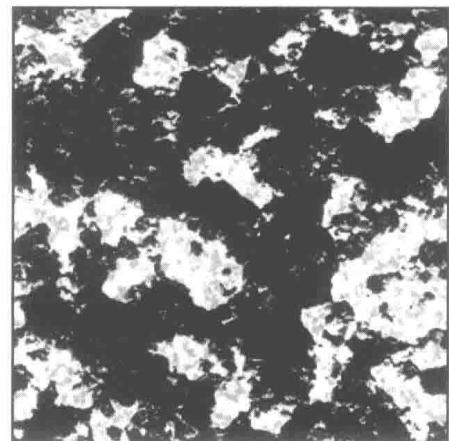


图 10.14

这里要再次强调：金属光泽度属性 metalness 的值会先与来自 metalnessMap 贴图中的值相乘、粗糙度属性 roughness 的值则会先与来自 roughnessMap 贴图中的值相乘，然后再应用于物体渲染中。加载这两种贴图的方法很简单，如下面代码片段所示。

```
mat1.metalnessMap =
textureLoader.load("../assets/textures/engraved/roughness-map.jpg")
mat2.roughnessMap =
textureLoader.load("../assets/textures/engraved/roughness-map.jpg")
```

接下来将要介绍 Alpha 贴图。该贴图的作用是指定物体的部分表面为透明。

### 10.1.8 Alpha 贴图

Alpha 贴图用于控制物体表面的透明度。贴图中的纯黑色部分代表该部分表面完全透明，纯白色部分则代表完全不透明。在介绍这种贴图及其使用方法之前，先来看一下相应的示例程序的截图（14-alpha-map.html）。截图如图 10.15 所示。

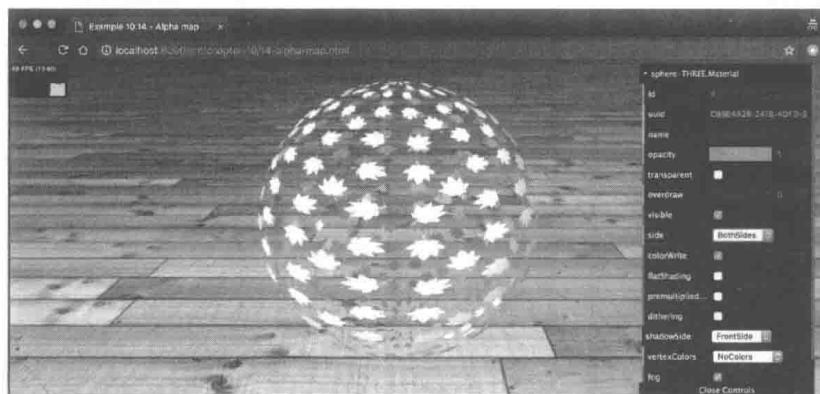


图 10.15

该示例程序在场景中绘制了一个球。其材质的 `alphaMap` 属性被设置了 Alpha 贴图。当你运行该示例程序时，会注意到只有球的正面（译注：朝向摄像机的一面）可见，而上面截图中的球既可以看到正面，也可以透过球面透明的部分看到其另一面。这是由于示例程序中，`side` 属性被设置成了 `THREE.FrontSide`。若想同时看到正反面，可以将该属性的值改为 `THREE.BothSides`。

示例程序使用的 Alpha 贴图如图 10.16 所示。

加载该贴图的代码如下：

```
var sphereMaterial = new THREE.MeshStandardMaterial({
  alphaMap: textureLoader.load("../assets/textures/alpha/partial-
transparency.png"),
  envMap: alternativeMap,
  metalness: 0.02,
  roughness: 0.07,
  color: 0xffffffff,
  alphaTest: 0.5
});
```

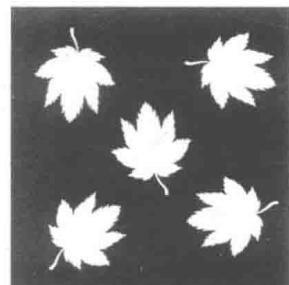


图 10.16

```
sphereMaterial.alphaMap.wrapS = THREE.RepeatWrapping;
sphereMaterial.alphaMap.wrapT = THREE.RepeatWrapping;
sphereMaterial.alphaMap.repeat.set(8, 8);
```

观察上面的代码，你会注意到 `alphaTest` 属性值为设置为 0.5。这样做的目的是避免使用半透明特性时可能出现的一些小斑点问题。如果你在使用多个半透明物体时，或者使用 `alphaMap` 属性时，可以解决这个问题。此外，在上面代码中你还会注意到我们为纹理设置 `wrapS`、`wrapT` 和 `repeat` 属性。这些属性将会在本章后面章节详细讨论，这里你只需要知道这些属性控制纹理在模型表面的重复模式和频率。当设置为 (1,1) 时，纹理不会重复。当设置为更高的值时，纹理会被缩小，并在模型表面重复贴图。在本例中我们设置纹理在两个坐标轴方向都重复 8 次。

### 10.1.9 自发光贴图

自发光贴图是一个控制模型表面实现自发光效果的纹理贴图，它的功能类似于前面介绍过的 `emissive` 属性，但后者只能将模型作为一个整体来控制。在浏览器中打开示例程序 `15-emissive-map.html` 后，可以看到画面中有两个表面上流淌着岩浆的物体。通过右上角的菜单可以控制自发光的亮度。你会发现即使当场景中的光源很暗淡时，物体的自发光部分仍然会保持固定的亮度。截图如图 10.17 所示。

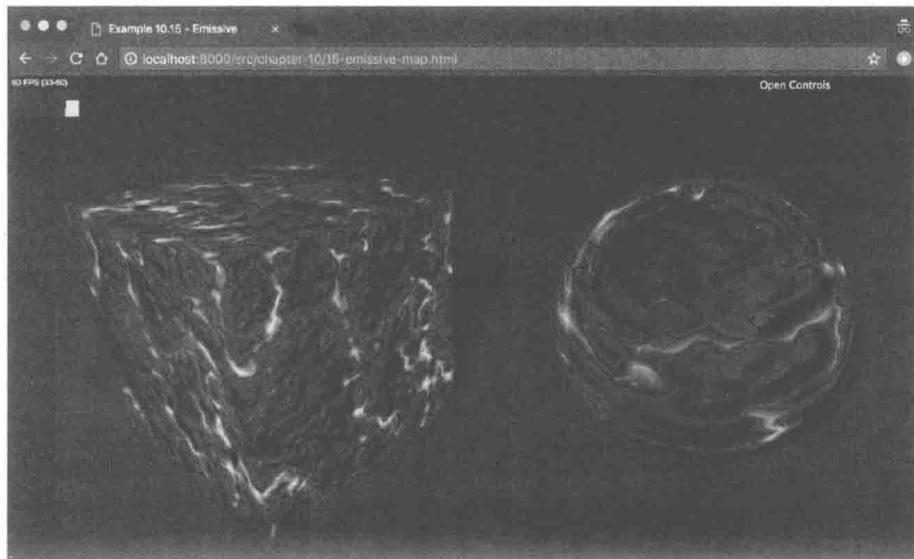


图 10.17

仔细观察渲染画面可以发现即使自发光物体自身看起来闪耀着光芒，但它对周围环境中的物体并没有影响。也就是说，自发光特性只能单独影响物体本身，却不能使该物体变成光源。

示例程序使用的自发光贴图如图 10.18 所示。

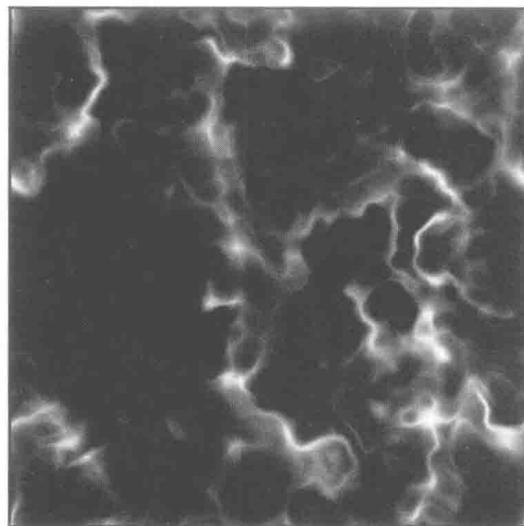


图 10.18

加载该贴图的代码如下：

```
var cubeMaterial = new THREE.MeshStandardMaterial({
    emissive: 0xfffffff,
    emissiveMap:
        textureLoader.load("../assets/textures/emissive/lava.png"),
        normalMap: textureLoader.load("../assets/textures/emissive/lava-
normals.png"),
        metalnessMap:
        textureLoader.load("../assets/textures/emissive/lava-smoothness.png"),
        metalness: 1,
        roughness: 0.4,
        normalScale: new THREE.Vector2(4, 4)
});
```

由于来自 `emissiveMap` 的颜色由 `emissive` 属性值调制，因此需要将该属性值设为非 0 值才能看到自发光贴图产生的效果。

### 10.1.10 高光贴图

在前面的例子中我们反复使用了 `THREE.MeshStandardMaterial` 材质，并且尝试了该材质所支持的各种贴图。一般情况下，在新版 Three.js 中 `THREE.MeshStandardMaterial` 材质是你的最佳选择，可以通过调节其各种属性来生成现实世界中的各种真实材质。而在旧版 Three.js 中，若想要生成闪亮的表面，需要使用 `THREE.MeshPhongMaterial` 材质，反之，则需要使用 `THREE.MeshLambertMaterial` 材质。

但是若要使用高光贴图，则不得不使用 `THREE.MeshPhongMaterial` 材质。高光贴图用于指定物体表面中哪部分比较闪亮，或者哪部分相对暗淡，这与 `THREE.MeshStandardMaterial` 材质的金属光泽度贴图和粗糙度贴图的作用有些接近。

示例程序 `16-specularmap.html` 在场景中渲染了一个地球，其中使用了高光贴图来生成

相对闪亮的海面和相对暗淡的陆地表面。截图如图 10.19 所示。

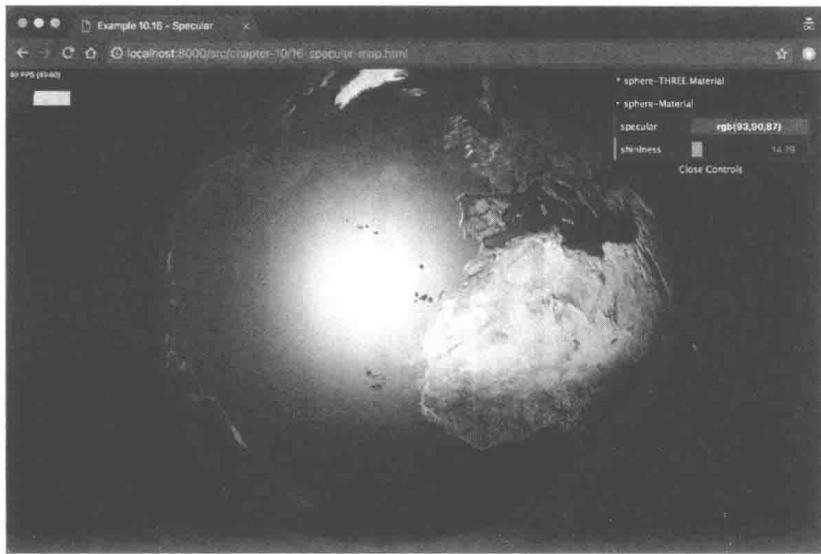


图 10.19

在右上角的菜单里可以设置 specular color (高光颜色) 和 shininess (闪亮程度) 属性。修改这些属性并观察画面中的变化，可以发现与前面章节的示例程序不同，在本节的示例程序中对这些属性的改变只影响海面的渲染效果，而陆地则不受影响。这是因为我们使用了图 10.20 中的高光贴图来指定模型的哪些部分具有高光效果，而哪些部分没有。

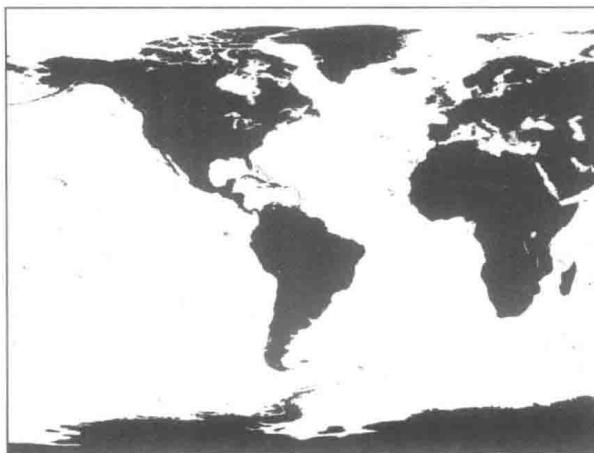


图 10.20

在贴图中，黑色部分表示完全没有高光效果，而白色部分则表示有完全的高光效果。

加载该贴图的代码如下。这段代码使用 THREE.TextureLoader 加载了纹理后，将它设置到 THREE.MeshPhongMaterial 材质的 specularMap 属性上。

```

var earthMaterial = new THREE.MeshPhongMaterial({
  map: textureLoader.load("../assets/textures/earth/Earth.png"),
  normalMap:
  textureLoader.load("../assets/textures/earth/EarthNormal.png"),
  specularMap:
  textureLoader.load("../assets/textures/earth/EarthSpec.png"),
  normalScale: new THREE.Vector2(6, 6)
});

```

到高光贴图为止，我们已经介绍了所有的基础纹理贴图。使用这些贴图可以在物体表面实现凹凸感、颜色、透明以及丰富的光照效果。在接下来的一小节里，我们将介绍如何使物体表面像镜面一样反射其周围环境的效果。

### 10.1.11 使用环境贴图创建伪镜面反射效果

计算镜面反射效果对 CPU 的耗费是非常大的，而且通常会使用光线追踪算法。在 Three.js 中你依然可以实现镜面反射效果，只不过是做一个假的。你可以通过创建一个对象所处环境的纹理来伪装镜面反射，并将它应用到指定的对象上。首先看一下我们所要实现的效果（17-env-map-static.html），贴图如图 10.21 所示。



图 10.21

如图 10.21 所示，你可以看到球和方块反射着周围的环境。如果移动鼠标，你还可以看到镜面反射跟随摄像机的移动而变化，并且与对应的城市环境相关联。这样的效果可以通过如下步骤创建：

- (1) 创建 CubeMap 对象。CubeMap 对象是包含 6 个纹理的集合，这些纹理可以应用到方块的每个面上。
- (2) 创建“天空盒”。当我们有了 CubeMap 对象后，可以将它设置为场景的背景。如

果这样做，实际相当于创建了一个很大的方块，并且将摄像机和场景物体都置于其中。这样当我们移动摄像机时，就能够看到背景环境跟随摄像机变换。因此我们也可以手动创建一个大方块模型，将 CubeMap 对象设置到模型上，最后将这个模型添加到场景中。

(3) 将 CubeMap 作为纹理使用。我们用来模拟环境的 CubeMap 对象同样可以作为纹理应用在模型上。Three.js 会让它看上去就像是镜面反射一样。

如果你已经有了制作材质的原材料，那么创建 CubeMap 对象就会非常简单。你所需要的就是 6 张用来构建整个场景的图片。这 6 张图片是：超前的 (posz)、朝后的 (negz)、朝上的 (posy)、朝下的 (negy)、朝右的 (posx) 和朝左的 (negx)。Three.js 会将这些图片整合到一起来创建一个无缝的环境贴图。网上有很多站点可以下载全景图片，但是它们经常以球形等距圆柱 (spherical equirectangular) 投影图的形式提供。贴图如图 10.22 所示。



图 10.22

要使用这种图片，最简单的方法是使用在线转换工具将它们转换成分离的图片文件。下面两个网站可以实现这种转换：

- <https://jaxry.github.io/panorama-to-cubemap/>
- <https://www.360toolkit.co/convert-spherical-equirectangular-tocubemap.html>

除了通过上述方法将球形等距圆柱投影图转换为可用的纹理图片之外，在 Three.js 中也可以直接将这种投影图当作 CubeMap 使用，只不过需要一些额外的操作。

球形等距圆柱投影图仍然需要通过标准纹理加载器来加载，但我们必须向 Three.js 明确指出该纹理需要使用的映射算法：

```
var textureLoader = new THREE.TextureLoader();
var cubeMap = textureLoader.load(
  "textures/2294472375_24a3b8ef46_o.jpg" );
cubeMap.mapping = THREE.EquirectangularReflectionMapping;
cubeMap.magFilter = THREE.LinearFilter;
cubeMap.minFilter = THREE.LinearMipMapLinearFilter;
```



现在就可以将 CubeMap 设定给模型的 envMap 以产生前面介绍的镜面反射效果了。但是这种 CubeMap 不能直接用于场景背景，我们需要自行创建天空盒以及对应的材质来实现背景：

```
var equirectShader = THREE.ShaderLib[ "equirect" ];
var equirectMaterial = new THREE.ShaderMaterial( {
    fragmentShader: equirectShader.fragmentShader,
    vertexShader: equirectShader.vertexShader,
    uniforms: equirectShader.uniforms,
    depthWrite: false,
    side: THREE.BackSide
});
equirectMaterial.uniforms[ "tEquirect" ].value = cubeMap;

// create the skybox
var skybox = new THREE.Mesh(new THREE.BoxGeometry(10000,
10000, 10000), material);
scene.add(skybox);
```



可见，如果不转换图片，通过多做几步操作也可以直接在程序里使用球形等距圆柱投影图。

在接下来的章节中，我们默认使用分离的全景图片，而不是球形等距圆柱投影图。对于分离的全景图片，可以通过下面的方法将它们加载成 CubeMap：

```
var urls = [
    '../../../../../assets/textures/cubemap/flowers/right.png',
    '../../../../../assets/textures/cubemap/flowers/left.png',
    '../../../../../assets/textures/cubemap/flowers/top.png',
    '../../../../../assets/textures/cubemap/flowers/bottom.png',
    '../../../../../assets/textures/cubemap/flowers/front.png',
    '../../../../../assets/textures/cubemap/flowers/back.png'
];

var cubeLoader = new THREE.CubeTextureLoader();
var cubeMap = cubeLoader.load(urls);
```

不使用 THREE.TextureLoader，而是使用 THREE.CubeTextureLoader。在其 load 函数里，我们需要按照正确的顺序指定图片的 URL。加载获得的 CubeMap 既可用于模型的环境贴图，也可以用于场景背景。下面的代码将 CubeMap 直接设定为场景背景：

```
scene.background = cubeMap
```

在设定完场景背景后，最后一步便是将 CuebMap 设定为场景中模型的环境贴图：

```
var sphereMaterial = new THREE.MeshStandardMaterial({
    envMap: cubeMap,
    color: 0xffffff,
    metalness: 1,
    roughness: 0,
});
```

在本节示例程序中，我们最终获得的效果是一个开阔的室外场景，其中有两个反射着场

景环境的物体。在右侧菜单上可以控制材质的属性，观察属性改变所带来的效果可以看到，增加 metalness 值可以使物体的镜面反射效果更明显，而增加 roughness 值可以使镜面反射变得更加模糊。示例程序的默认将这两个属性值都设定为较高值，效果如图 10.23 所示。



图 10.23

除了镜面反射以外，Three.js 还可以用环境贴图来实现环境折射效果（类似于玻璃工艺品）。图 10.24 展示了这一效果（可以通过在右侧菜单上修改属性来达到这一效果）。

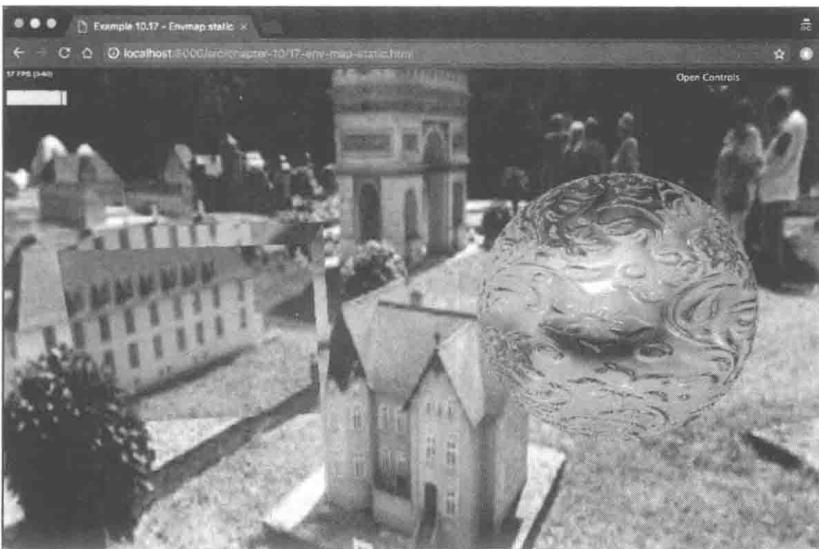


图 10.24

在程序中为了实现环境折射效果，需要将 CubeMap 的 mapping 属性设置为 THREE.

`CubeRefractionMapping`, 其默认值为环境反射:

```
cubeMap.mapping = THREE.CubeRefractionMapping
```

环境折射的效果同样由 `metalness` 和 `roughness` 两个属性控制。

上面的示例程序中只实现了静态环境的镜面反射效果。换句话说，模型表面的镜面反射只有场景背景，而没有同一场景中的其他模型。打开浏览器中的示例程序 `18-env-mapdynamic.html`, 你会看到如图 10.25 所示的效果。接下来将要介绍如何创建包含其他模型的镜面反射效果。



图 10.25

在程序中为了实现包含其他物体的镜面反射，我们需要使用 Three.js 提供的 `THREE.CubeCamera` 类：

```
var cubeCamera = new THREE.CubeCamera(0.1, 10, 512);
cubeCamera.position.copy(cubeMesh.position);
scene.add(cubeCamera);
```

我们将使用 `THREE.CubeCamera` (译注: CubeCamera 可以理解为一个立方体相机，拍照时它会分别对着六个方向拍摄六张照片) 对包含所有物体的场景拍摄快照并用获得的快照图片来创建 `CubeMap`。CubeCamera 构造函数中的前两个参数分别为摄像机的近平面和远平面。我们分别指定它们为 0.1 到 1.0，因此这个摄像机只渲染距离该摄像机这个区间之内的物体。第三个参数为渲染图片的尺寸。尺寸越大，将来用于镜面反射时，产生的反射画面越精细。在使用 CubeCamera 拍摄场景快照时，要将该摄像机准确摆放在需要产生镜面反射的物体的位置上。具体到本示例程序，CubeCamera 被放置在立方体的几何中心位置。

设置好 CubeCamera 的位置后，还需要将该摄像机的输出，即 `cubeCamera.renderTarget`，与立方体模型的环境贴图属性 `envMap` 相连，这样才能使摄像机获得的快照

变为物体表面的镜面反射图像。具体做法为：

```
cubeMaterial.envMap = cubeCamera.renderTarget;
```

最后，我们需要给 cubeCamera 一个机会去对场景进行拍照。要做到这一点，需要对渲染循环做如下所示的修改：

```
function render() {
    ...
    cube1.visible = false;
    cubeCamera.updateCubeMap(renderer, scene);
    cube1.visible = true;

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

正如你所看到的，首先将 cube1 的网格体设为不可见。这样做的原因在于我们只是想在反光中看到那个球体。然后通过调用 cubeCamera 的 updateCubeMap 函数来渲染场景。最后再将 cube1 的网格体恢复为可见，并正常地渲染场景。这样做的结果就是，在 cube1 的表面上能够看到球体的反光影像。你可以试着修改球体的材质设定，会看到球体在 cube1 的表面上的反光影像也会实时更新。

## 10.2 纹理的高级用途

在前面的章节中，我们看到了纹理的基本用法，Three.js 还提供了一些更高级的用途。在本节中我们将会介绍一些高级用途。

### 10.2.1 自定义 UV 映射

我们先来深入了解 UV 映射。前面我们已经解释过，通过 UV 映射可以指定纹理的哪部分会显示在物体的表面。当你在 Three.js 中创建几何体时，UV 映射会基于几何体的类型自动创建。多数情况下，你不需要修改默认的 UV 映射。如果想了解 UV 映射的工作原理，可以参看 Blender 里的示例，示例效果如图 10.26 所示。

在这个示例里可以看到两个窗口，左侧的窗口中有一个方块，而右侧的窗口表示的是 UV 映射，这里我们加载了纹理来展示映射效果。在示例的左侧窗口中我们选中方块的一个面，这个面相应的 UV 映射就会在右侧的窗口显示。正如你所看到的，这个面上的每一个顶点都在右侧 UV 映射的一个角上（用小圆圈表示），这意味着整个纹理都会应用到这个面上。这个方块的其他各个面也是如此处理的，这样我们就会得到一个展示所有纹理的方块。参看示例 19-uv-mapping.html，效果如图 10.27 所示。

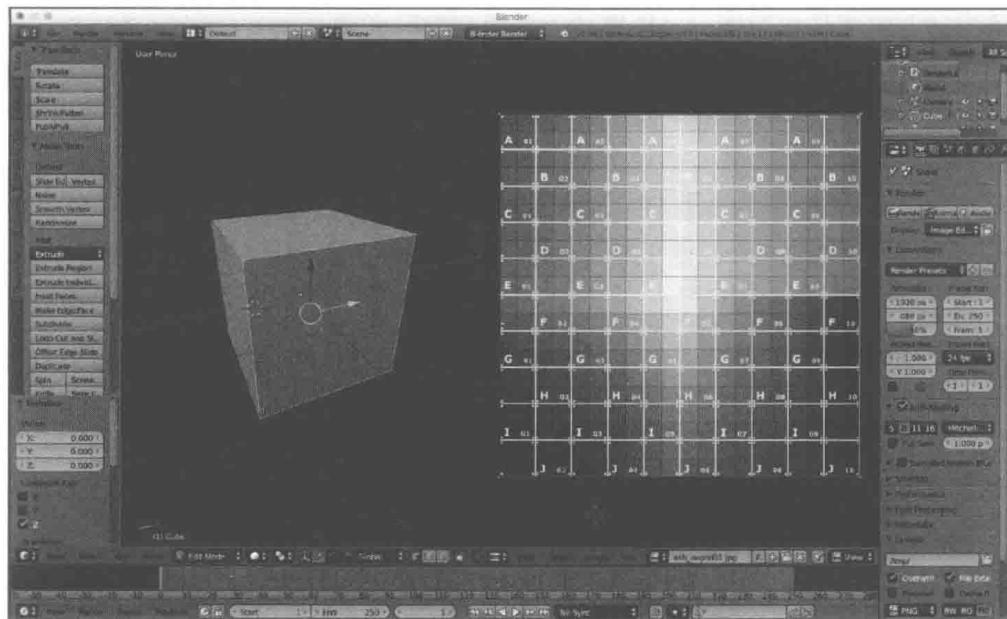


图 10.26

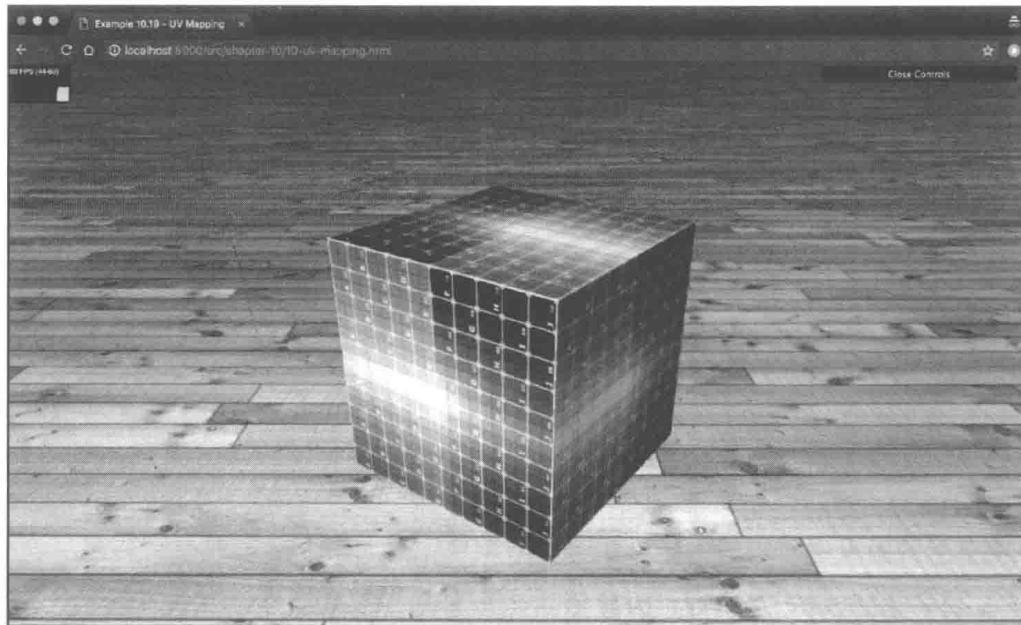


图 10.27

这是 Blender (也是 Three.js) 中在方块上贴图的默认行为。我们来修改 UV 映射，看看修改后的纹理是如何应用到物体表面的，如图 10.28 所示。

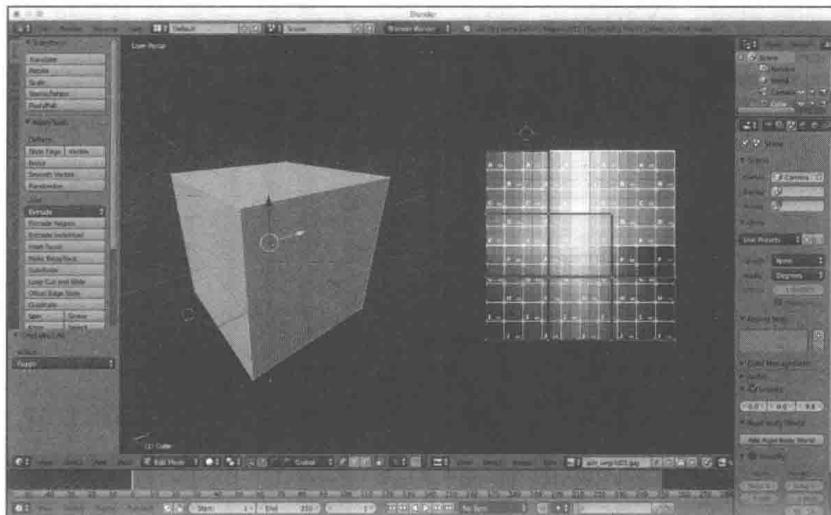


图 10.28

如果现在使用 Three.js 来显示，你会看到使用的纹理已经不一样了，如图 10.29 所示。

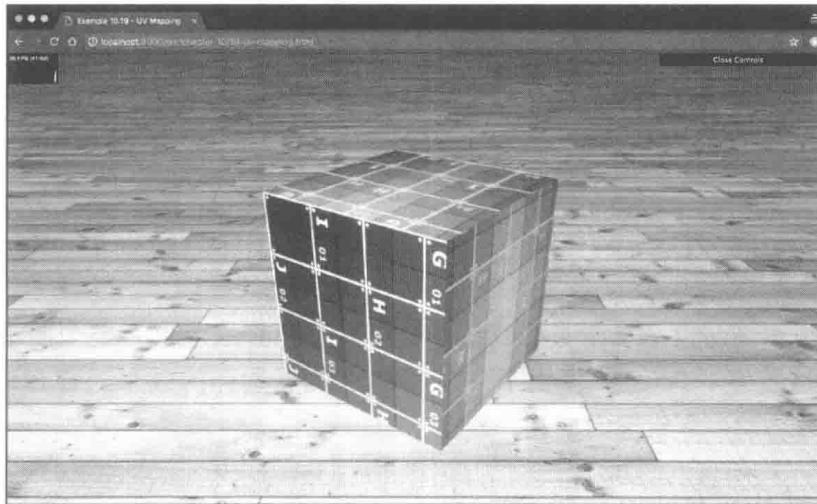


图 10.29

UV 映射的自定义通常是在诸如 Blender 的软件中完成的，特别是当模型变得复杂时。这里需要注意的是 UV 映射有两个维度—— $u$  和  $v$ ，其取值范围是从 0 到 1。自定义 UV 映射时，你需要为物体的每个面指定要显示纹理的哪一部分。为此，你需要为面的每个顶点设置  $u$  坐标和  $v$  坐标，具体设置方式如下所示：

```
geom.faceVertexUvs[0][0][0].x = 0.5;
geom.faceVertexUvs[0][0][0].y = 0.7;
geom.faceVertexUvs[0][0][1].x = 0.4;
geom.faceVertexUvs[0][0][1].y = 0.1;
geom.faceVertexUvs[0][0][2].x = 0.4;
geom.faceVertexUvs[0][0][2].y = 0.5;
```

这个代码片段将第一个面的 uv 属性设置为特定值。需要注意的是每个面都是用三维坐标表示的，所以在为每个面的 uv 属性设置值时，你需要设置 6 个值。打开示例 20-uv-mapping-manual.html 并手动修改 UV 映射的值，你会看到如图 10.30 所示的效果。

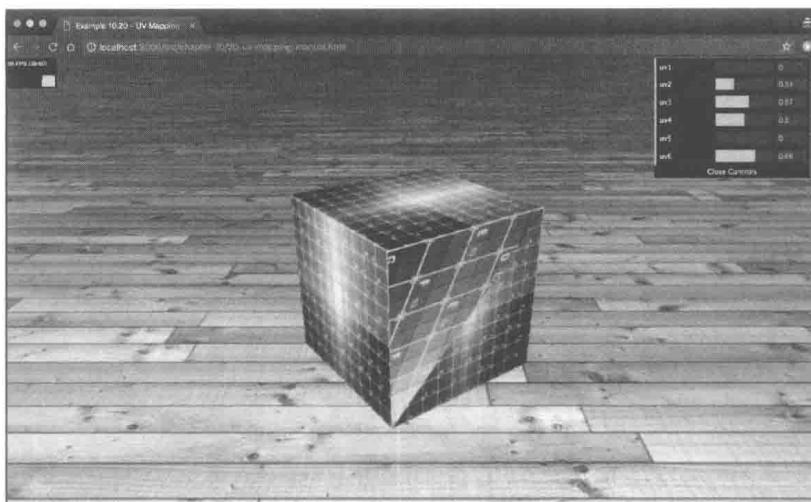


图 10.30

下面我们将要介绍的是如何重复（或者说复制）纹理，这是由一些内部 UV 映射来实现的。

## 10.2.2 重复纹理

当在 Three.js 中创建的几何体上应用纹理时，Three.js 会尽量实现最好的效果。例如，在方块上使用纹理时，Three.js 会在每个面上都显示完整的纹理。对于球体，Three.js 会用完整的纹理包围球体。但是对于有些情形，你可能不想将纹理遍布整个面或整个几何体，而是让纹理进行重复。Three.js 提供了一些功能来实现这样的效果。打开示例 21-repeat-wrapping.html 来体验一下纹理的重复，如图 10.31 所示。

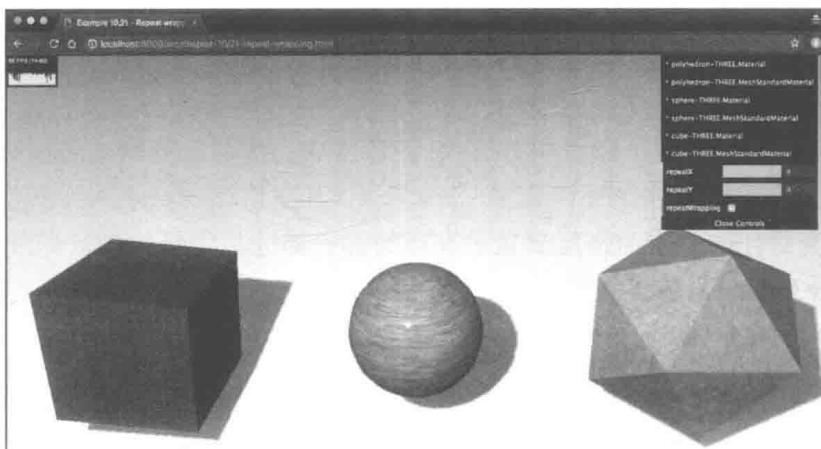


图 10.31

在通过这个属性达到所需要的效果之前，需要保证将纹理的包裹属性设置为 THREE.RepeatWrapping，如以下代码片段所示：

```
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 为这些属性提供了如下两个选项：

- THREE.RepeatWrapping 允许纹理重复自己。
- THREE.ClampToEdgeWrapping 是属性的默认值。属性值为 THREE.ClampToEdgeWrapping 时，那么纹理的整体不会重复，只会重复纹理边缘的像素来填满剩下的空间。

如果你取消了菜单项 repeatWrapping，那么 THREE.ClampToEdgeWrapping 就会被使用。如图 10.32 所示。

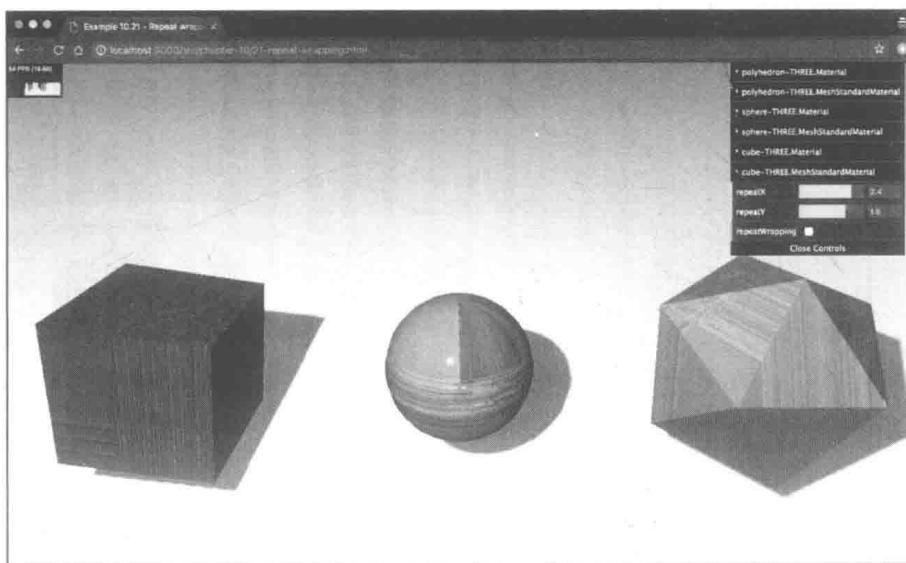


图 10.32

如果使用 THREE.RepeatWrapping，我们可以通过下面的代码来设置 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 则用来指定纹理在 y 轴方向多久重复一次。如果变量的值为 1，那么纹理则不会重复；如果设置成大于 1 的值，那么你就会看到纹理开始重复。当然你也可以将值设置为小于 1 的值，那么你会看到纹理被放大了。如果值设置为负数，那么会产生一个纹理的镜像。

当你修改 repeat 属性时，Three.js 会自动更新纹理，并使用新的值来进行渲染。如果你将值从 THREE.RepeatWrapping 修改为 THREE.ClampToEdgeWrapping 时，你需要明确地更新纹理：

```
cube.material.map.needsUpdate = true;
```

到目前为止，我们都是将静态的图像作为纹理。但是 Three.js 也支持将 HTML5 的画布作为纹理使用。

### 10.2.3 在画布上绘制图案并作为纹理

在本节中，我们将会介绍两个不同的示例。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。然后我们会创建一个画布，并将随机生成的图形作为凹凸贴图。

#### 1. 将画布作为纹理

在第一个示例中，我们会使用 Literally 库 (<http://literallycanvas.com/>) 来创建一个交互式的画布，在这个画布上你可以进行绘图。参见图 10.33 的左下角，该示例为 09-canvas-texture。

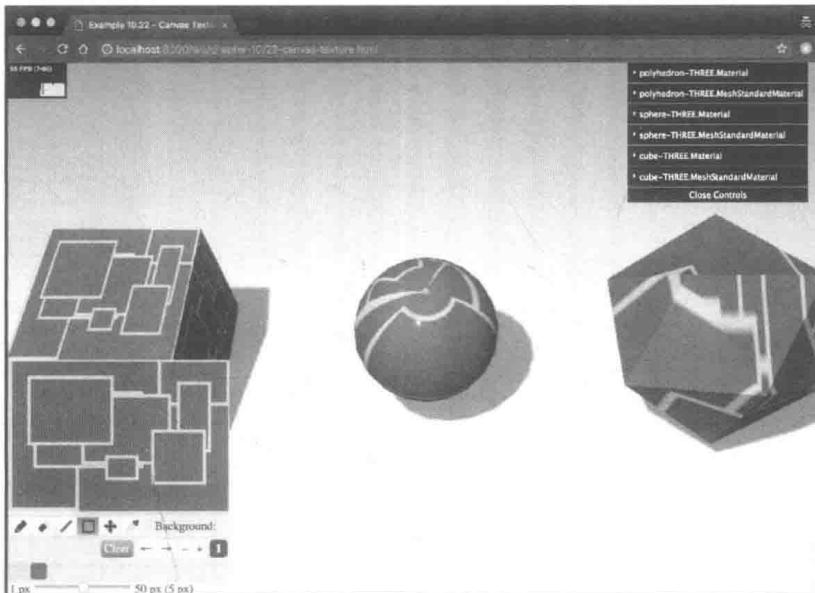


图 10.33

在画布上绘制的任何东西都会作为纹理直接渲染到方块上。在 Three.js 中只需要几步就可以实现这样的效果。首先我们需要做的是创建一个画布元素，然后配置该画布使用 Literally 库。代码如下：

```
<div class="fs-container">
  <div id="canvas-output" style="float:left">
  </div>
</div>

...
var canvas = document.createElement("canvas");
document.getElementById('canvas-output').appendChild(canvas);
$('#canvas-output').literallycanvas({imageURLPrefix:
  '.../.../libs/other/literally/img'});
```



在这里，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 元素上。通过调用 literallycanvas，我们可以创建一个绘图工具，使用该工具你可以直接在画布上绘画。接下来我们将画布上绘制的结果作为输入创建一个纹理：

```
var texture = new THREE.Texture(canvas);
```

正如代码所示，你唯一需要做的是在创建纹理时将画布元素的引用作为参数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。最后需要做的就是在渲染时更新材质，这样画布上的内容就会显示在方块上：

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

    polyhedronMesh.rotation.x += 0.01;
    sphereMesh.rotation.x += 0.01;
    cubeMesh.rotation.x += 0.01;

    polyhedronMesh.material.map.needsUpdate = true;
    sphereMesh.material.map.needsUpdate = true;
    cubeMesh.material.map.needsUpdate = true;
}
```

为了告知 Three.js 我们需要更新材质，需要将纹理的 needsUpdate 属性设置为 true。在这个示例中我们将画布作为最简单纹理的输入。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。在下面的示例中，我们将用它来生成凹凸贴图。

## 2. 将画布用作凹凸贴图

正如我们在本章前面所看到的，我们可以使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，所以也可以在画布上创建这个图片，并将画布作为凹凸贴图的输入。

在下面的示例中，我们在画布上随机生成了一幅灰度图，并将该图作为方块上凹凸贴图的输入。参见示例 23-canvas-texture-bumpmap.html，效果如图 10.34 所示。

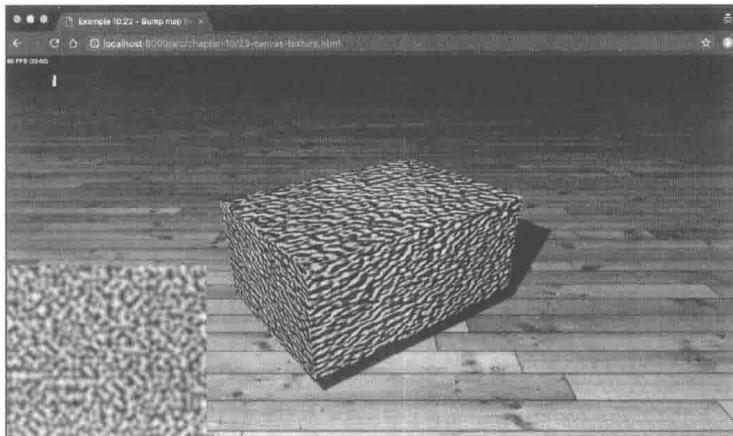


图 10.34

实现该功能的 JavaScript 代码和前面的示例基本一样。我们需要创建一个画布元素，然后用一些随机噪声来填充该画布。至于噪声，我们使用的是 Perlin 噪声。Perlin 噪声 ([http://en.wikipedia.org/wiki/Perlin\\_noise](http://en.wikipedia.org/wiki/Perlin_noise)) 可以产生看上去非常自然的随机纹理，如图 10.22 所示。我们可以使用 <https://github.com/wwwtyro/perlin.js> 中的噪声函数，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(0xffffffff);
            var value = perlin.noise(x / 10, y / 10, 0);
            base.multiplyScalar(value);
            ctx.fillStyle = "#" + base.getHexString();
            ctx.fillRect(x, y, 1, 1);
        }
    }
}
```

我们使用 `perlin.noise` 方法依据画布 `x` 坐标和 `y` 坐标的值生成一个从 0 到 1 的值。该值可以用于在画布上画一个像素点。可以使用这个方法生成所有的像素点。其结果就是图 10.22 中左下角的那个随机贴图。这个随机贴图可以作为凹凸贴图使用。下面的代码展示了如何创建随机贴图：

```
var cube = new THREE.CubeGeometry(23, 10, 16)
var cubeMaterial = new THREE.MeshStandardMaterial({
    bumpMap: new THREE.Texture(canvas),
    metalness: 0,
    roughness: 1,
    color: 0xffffffff,
    bumpScale: 3,
    map: textureLoader.load('../assets/textures/general/wood-2.jpg')
});
```



在这个示例中，我们使用 HTML 的画布元素来渲染 Perlin 噪声。Three.js 还提供了其他方法来动态地创建纹理。你可创建一个 `THREE.DataTexture` 纹理。这个纹理的 `image.data` 属性包含的 `Uint8Array`，可以直接用于设置纹理的 RGB 值。

最后要讨论的是可以作为纹理输入的另一个 HTML 元素：HTML5 视频元素。

#### 10.2.4 将视频输出作为纹理

如果你看过前面有关将画布作为纹理的内容，那么你可能会想到将视频输出到画布上，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 的视频元素。参见示例 24-video-texture.html，效果如图 10.35 所示。

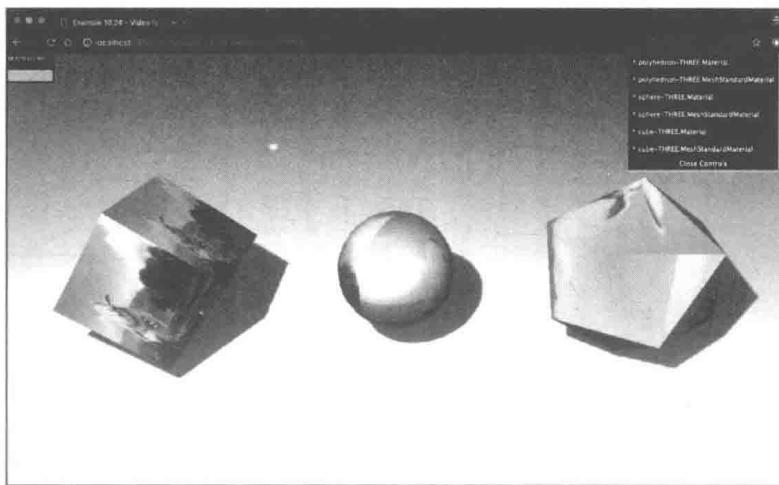


图 10.35

将视频作为纹理的输入和使用画布一样简单。首先，我们要有一个视频元素用来播放视频：

```
<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>
```

这只是一个基本的 HTML5 的视频元素，可以用来自动播放视频。接下来我们可以设置 Three.js，使用该视频作为纹理的输入：

```
var video = document.getElementById( 'video' );
var texture = new THREE.VideoTexture(video);
texture.minFilter = THREE.LinearFilter;
texture.magFilter = THREE.LinearFilter;
texture.format = THREE.RGBFormat;
```

由于我们的视频不是正方形的，所以需要确保材质不会生成 mipmap。由于材质变化很频繁，所以我们还需要设置简单高效的过滤器。最后我们就可以基于上面的纹理对象来创建材质。具体示例程序以及渲染结果请参考 24-video-texture.html。

### 10.3 总结

关于纹理的内容到此就结束了。正如你所看到的，在 Three.js 中有各种各样的纹理，每种都有其独特的用途。你可以使用任何一张 PNG、JPG、GIF、TGA、DDS、PVR、KTX、EXR 或者 RGBE 格式的图片作为纹理。需要注意的是图片的加载是异步的，所以在加载图片时要么使用渲染，要么在加载纹理时提供一个回调方法。

使用纹理，你可以在低阶模型上创建出效果非常好的物体，甚至使用凹凸贴图和法向贴图为物体添加丰富的虚假细节。使用 Three.js 可以很容易地使用 HTML5 的画布或者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，并在纹理更新时将 `needsUpdate` 属性设置为 `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的内容是后期处理。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，你可以让那个场景变得模糊，或者变得色彩艳丽，或者使用扫描线添加一些类似电视的效果。下一章就会介绍后期处理，以及如何将它应用于场景中。

## 自定义着色器和后期处理

本书接近结束了，但是还有个主要功能没有介绍，那就是渲染后期处理。在本章将会对这个功能进行介绍，除此之外还会向你介绍如何自定义着色器。在本章讨论的内容包括如下几点：

- 配置 Three.js，用于后期处理。
- 介绍 Three.js 提供的基本后期处理通道，如 THREE.BloomPass 和 THREE.FilmPass。
- 使用掩码将效果应用到部分场景。
- 使用 THREE.ShaderPass 添加更基础的后期渲染处理，如褐色滤镜、镜像效果和颜色调整。
- 使用 THREE.ShaderPass 产生各种模糊效果和高级滤镜。
- 通过开发简单的着色器来创建自定义后期处理。

我们在第 1 章中创建了一个 render 循环，并在本书中一直使用该 render 循环来渲染场景和制作动画。对于后期处理，我们需要对这个渲染循环进行修改，使得 Three.js 库能够对最终的渲染结果进行后期处理。我们将会在 11.1 节中介绍如何对渲染循环进行修改。

### 11.1 配置 Three.js 以进行后期处理

为了能够使用 Three.js 进行后期处理，我们需要对当前的配置进行如下修改：

- 创建 THREE.EffectComposer（效果组合器）对象，在该对象上我们可以添加后期处理通道。
- 配置 THREE.EffectComposer 对象，使它可以渲染场景，并应用后期处理。
- 在渲染循环中，使用 THREE.EffectComposer 来渲染场景、应用通道和输出结果。

同样，我们也提供了相应的示例供读者实验或做其他用途。本章的第一个示例可以在文件 01-basic-effect-composer.html 中找到。在示例中，你可以使用右上角的菜单来调整各个后期处理步骤的属性。该示例渲染了一个地球，并添加电视栅格效果，栅格效果是在场景渲染结束后使用 THREE.EffectComposer 来添加的。效果如图 11.1 所示。

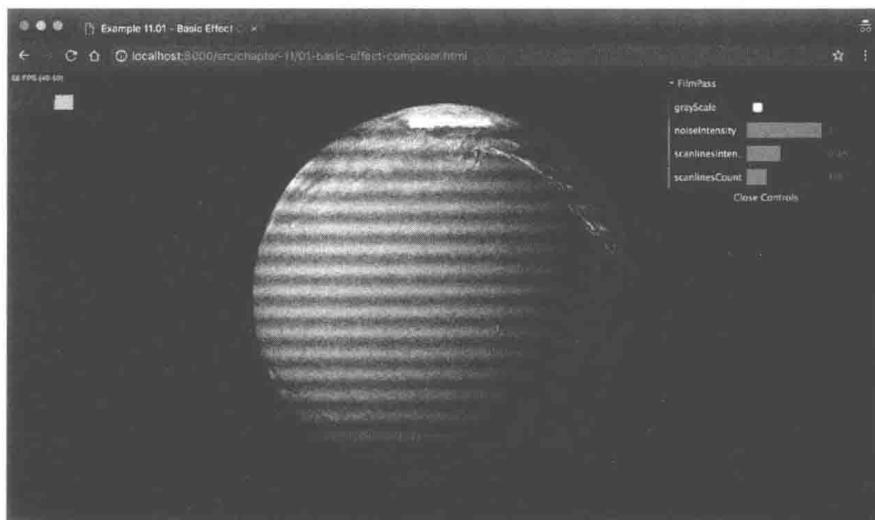


图 11.1

### 1. 创建 THREE.EffectComposer 对象

首先需要包含相应的 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。

为了使得 THREE.EffectComposer 正常工作，你至少需要包含如下的文件：

```
<script type="text/javascript"
src="../../libs/three/postprocessing/EffectComposer.js"></script>
<script type="text/javascript"
src="../../libs/three/postprocessing/ShaderPass.js"></script>
<script type="text/javascript"
src="../../libs/three/postprocessing/MaskPass.js"></script>
<script type="text/javascript"
src="../../libs/three/postprocessing/RenderPass.js"></script>
<script type="text/javascript"
src="../../libs/three/shaders/CopyShader.js"></script>
```

EffectComposer.js 文件提供 THREE.EffectComposer 对象，以便添加后期处理步骤。MaskPass.js, ShaderPass.js 和 CopyShader.js 文件是 THREE.EffectComposer 内部使用到的文件。RenderPass.js 文件则用于在 THREE.EffectComposer 对象上添加渲染通道，如果没有通道，我们的场景就不会被渲染。

在这个示例中，我们添加了另外两个 JavaScript 文件，用来在场景中添加一种类似胶片的效果，如下所示：

```

<script type="text/javascript"
src="../../libs/three/shaders/FilmShader.js"></script>
<script type="text/javascript"
src="../../libs/three/postprocessing/FilmPass.js"></script>

```

首先我们需要做的是创建一个 THREE.EffectComposer 对象，你可以在对象的构造函数里传入 THREE.WebGLRenderer，如下所示：

```

var renderer = new THREE.WebGLRenderer();
var composer = new THREE.EffectComposer(renderer);

```

接下来我们要在这个组合器中添加各种通道。

## 2. 为后期处理配置 THREE.EffectComposer

每个通道都会按照其加入到 THREE.EffectComposer 的顺序执行。第一个加入的通道是 THREE.RenderPass。下面这个通道会渲染场景，但是渲染结果不会输出到屏幕上：

```

var renderPass = new THREE.RenderPass(scene, camera);
composer.addPass(renderPass);

```

调用 addPass() 方法就可以将 THREE.RenderPass 添加到 THREE.EffectComposer 对象上。接下来我们要添加一个可以将结果输出到屏幕上的通道，当然了，并不是所有的通道都能够实现这个功能，在本例中我们使用 THREE.FilmPass 通道来将结果输出到屏幕上。在添加 THREE.FilmPass 通道时，我们首先要创建该对象，然后将其添加到组合器中。实现代码如下所示：

```

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(renderer);
composer.addPass(renderPass);
composer.addPass(effectFilm);

```

如你所看到的，我们创建了 THREE.FilmPass 对象，并将它的 renderToScreen 属性设置为“true”。这个通道是在 renderPass 之后添加到 THREE.EffectComposer 组合器中的，所以当使用这个组合器时，场景会被渲染并通过 THREE.FilmPass 将结果输出到屏幕上。

## 3. 更新渲染循环

现在我们需要稍微对渲染循环进行修改，也就是在循环中使用组合器来替换 THREE.WebGLRenderer：

```

function render() {
  stats.update();
  var delta = clock.getDelta();
  trackballControls.update(delta);
  earth.rotation.y += 0.001;
  pivot.rotation.y += -0.0003;

  // request next and render using composer
  requestAnimationFrame(render);
  composer.render(delta);
}

```

在代码中我们移除了 `renderer.render( scene, camera )`，然后使用 `composer.render( delta )`，这样就会调用 `EffectComposer` 对象的 `render` 方法。由于我们已经将 `FilmPass` 的 `renderToScreen` 属性设置为 `true`，所以 `FilmPass` 的结果会输出到屏幕上。

现在如果运行示例程序，会看到如图 11.2 所示的画面。



图 11.2



本示例程序仍然支持用于在场景中移动的常规操作。本章介绍的所有后期效果都是在已经渲染完成一帧场景画面之后添加的。在完成了这些基本配置之后，我们将会在接下来的章节中介绍其他的后期处理通道。

## 11.2 后期处理通道

Three.js 库提供了许多后期处理通道，这些通道可以直接添加到 `THREE.EffectComposer` 组合器中使用。



本章中的大部分着色器和着色工序都可被配置。当你想要使用自己的着色器时，最好也为它添加一个 UI，以便对该着色器的参数进行配置。因为这样可以方便你更快地找到最合适特定场景的参数值。

表 11.1 包含了所有可用的通道以及功能概述。

表 11.1

名 称	描 述
THREE.AdaptiveToneMappingPass	该通道根据场景的光亮度自动调节场景的亮度
THREE.BloomPass	该通道通过增强场景中明亮的区域来模拟真实世界中的摄像机
THREE.BokehPass	该通道可以实现类似大光圈镜头的景深效果
THREE.ClearPass	该通道清空当前纹理缓存
THREE.CubeTexturePass	该通道用于渲染天空盒
THREE.DotScreenPass	该通道会将黑点图层应用到屏幕的原始图片上
THREE.FilmPass	该通道通过扫描线和失真来模拟电视屏幕效果
THREE.GlitchPass	该通道会随机地在屏幕上显示电脉冲
THREE.HalfTonePass	该通道可以模拟传统印刷技术中的半色调效果。半色调技术使用网格点的大小和疏密来表现亮度
THREE.MaskPass	使用该通道可以在当前图片上添加掩码，后续的通道只会影响掩码区域
THREE.OutlinePass	该通道可以勾勒场景中物体的轮廓
THREE.RenderPass	该通道会在当前场景和摄像机的基础上渲染出一个新场景
THREE.SAOPass	该通道可以实现实时环境光遮挡效果
THREE.SMAAPass	该通道添加全屏反锯齿效果
THREE.SSAARenderPass	该通道通过另一种算法实现全屏反锯齿效果
THREE.SSAOPass	该通道通过另一种算法实现实时环境光遮挡效果
THREE.SavePass	当该通道执行时会复制当前的渲染结果，在后续的步骤中可以使用。但是该通道在实际应用中作用不大，所以在我们的示例中并没有使用
THREE.ShaderPass	该通道接受自定义的着色器作为参数，以生成一个高级、自定义的后期处理通道
THREE.TAARenderPass	该通道也是一种全屏反锯齿效果
THREE.TexturePass	该通道将组合器的当前状态保存为纹理，然后将其作为参数传入到其他的 EffectComposer 组合器中
THREE.UnrealBloomPass	该通道与 THREE.Bloom 类似，但它实现的效果更接近 Unreal 3D 引擎的 Bloom 效果



Three.js 还提供了另一种更好的景深效果，该效果可以利用 THREE.ShaderPass 调用着色器 THREE.BokehShader2 和 THREE.DOFMipMapShader 来实现。使用这些着色器需要一些较为复杂的设置。示例代码可以在下面网站里找到：

[http://threejs.org/examples/webgl\\_postprocessing\\_dof2.html](http://threejs.org/examples/webgl_postprocessing_dof2.html)

下面我们先从简单的通道开始进行介绍。

### 11.2.1 简单后期处理通道

对于简单后期处理通道，我们来看看使用 THREE.FilmPass、THREE.BloomPass 和 THREE.DotScreenPass 可以做什么。关于这些通道的示例参看 02-post-processing-simple，在示例中你可以体验一下通道如何对原始结果进行不同的影响的。示例输出结果如图 11.3 所示。

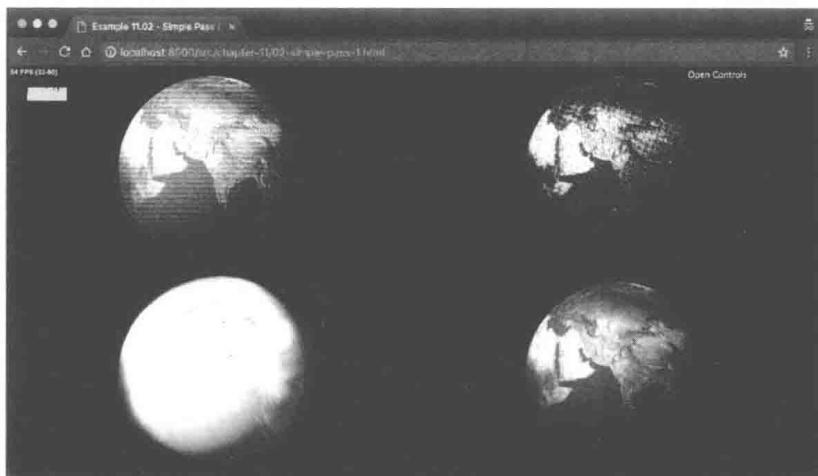


图 11.3

在该示例中我们同时展示了四个场景，这四个场景使用的是不同的后期处理通道。左上角场景使用的是 THREE.FilmPass 通道，右上角使用的是 THREE.DotScreenPass 通道，左下角使用的是 THREE.BloomPass 通道，右下角的是没有使用通道的原始渲染结果。

在这个示例中，我们还使用了 THREE.ShaderPass 和 THREE.TexturePass，这两个通道可以将原始渲染结果的输出作为其他三个场景的输入，以此来重用原始渲染结果。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(renderer);
composer.addPass(renderPass);
composer.addPass(effectCopy);

// reuse the rendered scene from the composer
var renderedScene = new THREE.TexturePass(composer.renderTarget2);
```

在这段代码里我们创建了 THREE.EffectComposer 对象，该对象输出默认场景（右下角的原始场景）。这个组合器有两个通道：THREE.RenderPass（用来渲染场景）和 THREE.ShaderPass（使用 THREE.CopyShader 进行配置后，如果 renderToScreen 属性设置为“true”，渲染结果会直接输出到屏幕上而不会有进一步的后期处理）。如果你看过示例代码，就会发现相同的场景被输出了四次，但是每次都应用了不同的效果。我们可以使用 THREE.

RenderPass 每次都将场景从零开始渲染，但是这样有点浪费，因为我们可以重用第一个组合器的输出结果。要做到这点，我们可以创建一个 THREE.TexturePass 对象，并将其传入到 composer.renderTarget2 中。这样就可以将 renderScene 变量作为其他组合器的输入，而无需从零开始渲染场景。下面让我们先来看看 THREE.FilmPass，以及如何将 THREE.TexturePass 作为输入的。

### 1. 使用 THREE.FilmPass 创建类似电视的效果

我们已经在本章第一节中介绍了如何创建 THREE.FilmPass，现在我们就来介绍如何结合上一节的 THREE.TexturePass 来使用该效果。参考代码如下所示：

```
var effectFilm = new THREE.FilmPass(0.8, 0.325, 256, false);
effectFilm.renderToScreen = true;

var effectFilmComposer = new THREE.EffectComposer(renderer);
effectFilmComposer.addPass(renderedScene);
effectFilmComposer.addPass(effectFilm);
effectFilmComposer.addPass(effectCopy);
```

如果要使用 THREE.TexturePass，你唯一需要做的就是将它作为组合器的第一个通道。接下来我们就可以添加 THREE.FilmPass，而且该通道接受如下四个参数：

表 11.2

属性	描述
noiseIntensity	通过该属性你可控制场景的粗糙程度
scanlinesIntensity	THREE.FilmPass 会在场景中添加扫描线。通过该属性可以指定扫描线的显著程度
scanLinesCount	通过该属性可以控制扫描线的数量
grayscale	如果该属性设置为“true”，输出结果将会被转换为灰度图

有两种方法可以来传递这些参数。在这个示例里，我们将它们作为构造函数的参数进行传递。当然你可以直接设置它们，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;
```

在这个方法中，我们使用了 uniforms 属性，该属性可以直接和 WebGL 进行通信。在自定义着色器的章节中将会进一步讲解 uniforms 属性。现在你只要知道可以通过它直接更改后期处理通道和着色器的配置，而且修改后的结果可以立即看到。

### 2. 使用 THREE.BloomPass 在场景中添加泛光效果

在上一节图示的左下角你所看的效果被称为泛光效果。当应用泛光效果时，场景中的明亮区域将会变得更加显著，而且还会渗入到较暗的区域。创建 THREE.BloomPass 的代码

如下所示：

```
var bloomPass = new THREE.BloomPass();
var effectCopy = new THREE.ShaderPass(THREE.CopyShader);
effectCopy.renderToScreen = true;

var bloomComposer = new THREE.EffectComposer(renderer);
bloomComposer.addPass(renderedScene);
bloomComposer.addPass(bloomPass);
bloomComposer.addPass(effectCopy);
```

如果将这段代码与使用 THREE.FilmPass 的 THREE.EffectComposer 进行比较，你会发现我们在这里多添加了一个通道——effectCopy。这个通道在普通输出中也曾用到过，它不会添加任何特殊的效果，只是将最后一个通道的输出结果复制到屏幕上。之所以要添加这个通道，是因为 THREE.BloomPass 通道不能直接将渲染结果输出到屏幕上。表 11.3 列出的是 THREE.BloomPass 所有可以设置的属性。

表 11.3

属性	描述
Strength	该属性定义的是泛光效果的强度。其值越大，则明亮的区域越明亮，而且渗入到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	通过该属性可以控制泛光效果的锐利程度。其值越大，泛光效果看起来越模糊
Resolution	该属性定义的是泛光效果的精确度。其值越低，泛光效果的方块化越严重

同样，理解这些属性最好的方式就是在示例 02-post-processing-simples 上来试验。图 11.4 表示的是泛光效果，该示例使用了较高的 sigma 和较低的 strength 值。

最后一个要介绍的简单通道是 THREE.DotScreenPass。

### 3. 将场景作为点集输出

THREE.DotScreenPass 的使用方式和 THREE.BloomPass 是非常相似的。我们刚才看了 THREE.BloomPass 的示例，下面来看 THREE.DotScreenPass 的代码：

```
var dotScreenPass = new THREE.DotScreenPass();

var dotScreenComposer = new THREE.EffectComposer(renderer);
dotScreenComposer.addPass(renderedScene);
dotScreenComposer.addPass(dotScreenPass);
dotScreenComposer.addPass(effectCopy);
```

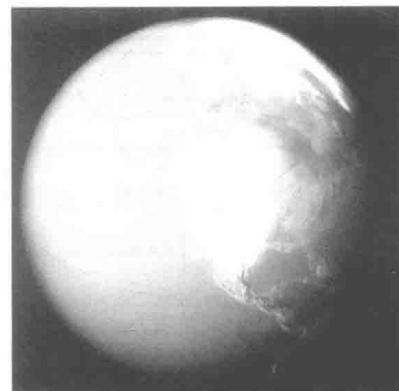


图 11.4

为了达到该效果，我们仍然需要添加 effectCopy 来将结果输出到屏幕上。THREE.DotScreenPass 可以配置的属性如表 11.4 所示。

表 11.4

属性	描述
center	通过该属性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 属性可以改变对齐的方式
Scale	该属性设置所用点的大小。其值越小，则点越大

跟其他着色器一样，利用试验可以获取合适的配置，如图 11.5 所示。在进行简单着色器的下一步之前，先看一下如何在同一个屏幕上渲染多个场景。

#### 4. 在同一个屏幕上显示多个渲染器的输出结果

本节不会介绍如何使用后期处理效果，而是解释如何将四个 THREE.EffectComposer 实例的输出结果显示在同一个屏幕上。首先让我们看看本例所使用的 render 循环：

```
renderer.autoClear = false;
function render() {
    stats.update();
    var delta = clock.getDelta();
    trackballControls.update(delta);

    renderer.clear();
    renderer.setViewport(0, 0, halfWidth, halfHeight);
    effectFilmComposer.render(delta);

    renderer.setViewport(0, halfHeight, halfWidth, halfHeight);
    bloomComposer.render(delta);
    renderer.setViewport(halfWidth, 0, halfWidth, halfHeight);
    dotScreenComposer.render(delta);
    renderer.setViewport(halfWidth, halfHeight, halfWidth, halfHeight);
    composer.render(delta);

    requestAnimationFrame(render);
}
```

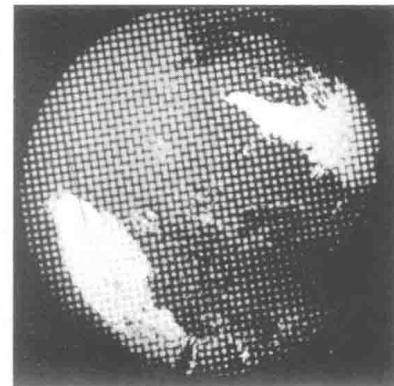


图 11.5

首先需要注意的是我们将 Renderer.autoClear 属性设置为“false”，并显式调用 clear() 方法。如果我们不这么做，那么每次调用组合器的 render() 方法时，之前的渲染场景将会被清除。通过这种方法，我们只是在 render 循环开始的时候将所有的东西进行清除。

为了避免所有效果组合器在同一个地方进行渲染，我们将效果组合器所用的 Renderer 的视图区域设置成屏幕的不同部分。设置视图区域的方法接受四个参数：x、y、width 和 height。正如你在示例代码中看到的，我们使用这个方法将屏幕分成四个部分，每个效果组合器在各自的区域中进行渲染。如果你愿意，你也可以将该方法用于多场景、多摄像机和

多 WebGLRenderer 情况下。

这样一来，渲染循环会将四个 THREE.EffectComposer 对象分别绘制到它们自己的屏幕区域中。除了示例程序 02-simple-pass-1.html 所展示的渲染通道之外，在示例程序 03-simple-pass-2.html 中还可以看到更多简单后期处理通道的效果。

## 5. 更多简单后期处理通道

示例程序 03-simple-pass-2.html 展示了更多简单后期处理通道的效果，如图 11.6 所示。

图 11.6

我们不在这里展开介绍这些额外的简单后期处理通道，因为它们的使用方法与前面介绍的其他通道相同。本示例展示了下面效果：

- 右上角是 THREE.OutlinePass 通道的效果。该通道可以为场景中的模型勾勒轮廓。
- 左上角是 THREE.GlitchPass 通道的效果。该通道可以模仿随机出现的电脉冲效果。
- 右下角是 THREE.UnrealBloom 通道的效果。该通道同样实现泛光效果，但其效果看起来更接近 Unreal 3D 引擎中的泛光效果。
- 左下角是 THREE.HalftonePass 通道的效果。该通道可以模仿传统印刷技术中的半色调效果。

在本章的所有示例程序中，右侧的菜单都可以用来设置每一个渲染通道的属性。

### 11.2.2 使用掩码的高级效果组合器

在前面的示例中，我们是在整个屏幕上应用后期处理通道。当然，Three.js 也可以在特定的区域使用通道。在本节中我们将会采取如下的步骤：

- (1) 创建一个作为背景图片的场景。
- (2) 创建一个场景，里面包含一个看起来像地球的球体。
- (3) 创建一个场景，里面包含一个看起来像火星的球体。

(4) 创建 EffectComposer 对象，用于将这三个场景渲染到一个图片中。

(5) 在渲染为火星的球体上应用彩色效果。

(6) 在渲染为地球的球体上应用褐色效果。

这看起来很复杂，其实完成起来相当简单。先看一下我们要实现的目标，可以参看示例 03-post-processing-masks.html，效果如图 11.7 所示。

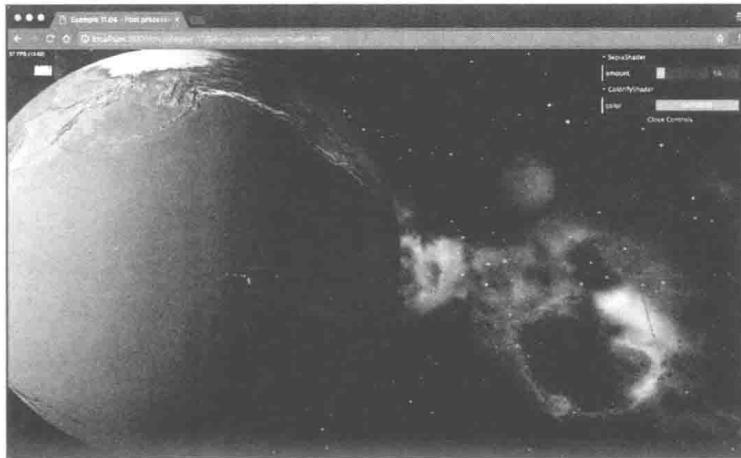


图 11.7

首先我们需要做的是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();
```

场景中的地球和火星是两个拥有特定材质的简单球体。下面代码中的 addEarth 和 addMars 为两个辅助函数，它们负责基于 THREE.SphereGeometry 创建 THREE.Mesh 对象，设置光照，并将它们添加到场景中。我们将这些琐碎工作封装在两个函数中，以便程序的逻辑更加清晰。

```
sceneBG.background = textureLoader.load("../assets/textures/bg/starry-
deep-outer-space-galaxy.jpg");
var earthAndLight = addEarth(sceneEarth);
sceneEarth.translateX(-16);
sceneEarth.scale.set(1.2, 1.2, 1.2);
var marsAndLight = addMars(sceneMars);
sceneMars.translateX(12);
sceneMars.translateY(6);
sceneMars.scale.set(0.2, 0.2, 0.2);
```



在本示例程序中，我们直接使用场景的背景属性来添加星光背景图片。还有一种可选方式也可以同于添加背景图片，那就是利用 THREE.OrthoGraphicCamera。因为 THREE.OrthoGraphicCamera 不会根据物体距离摄像机的距离来缩放物体，所以可以添加一个正对摄像机的 THREE.PlanGeometry 几何体，然后在上面添加图片。

现在我们有三个场景了，那么我们就可以开始配置后期处理通道和 THREE.EffectComposer。我们可从整个后期处理通道链开始，然后再具体看看每个通道：

```
var composer = new THREE.EffectComposer(renderer);
composer.renderTarget1.stencilBuffer = true;
composer.renderTarget2.stencilBuffer = true;
composer.addPass(bgRenderPass);
composer.addPass(earthRenderPass);
composer.addPass(marsRenderPass);
composer.addPass(marsMask);
composer.addPass(effectColorify);
composer.addPass(clearMask);
composer.addPass(earthMask);
composer.addPass(effectSepia);
composer.addPass(clearMask);
composer.addPass(effectCopy);
```

要使用掩码，我们还需要用不同的方式来创建 THREE.EffectComposer。在本例中，我们需要将渲染器内部使用的 stencilBuffer 属性设置为“true”。模板缓存（stencil buffer）是一种特殊类型的缓存，用于限制渲染区域，启用模板缓存后，就可以使用掩码了。首先让我们来看一下所添加的三个通道，这三个通道会渲染背景、地球场景和火星场景。如下所示：

```
var bgRenderPass = new THREE.RenderPass(sceneBG, camera);
var earthRenderPass = new THREE.RenderPass(sceneEarth, camera);
earthRenderPass.clear = false;
var marsRenderPass = new THREE.RenderPass(sceneMars, camera);
marsRenderPass.clear = false;
```

这里并没有新的概念，只不过是将通道的 clear 属性设置为“false”。如果我们不这样做，我们就只会看到 renderPass2 的输出结果，因为它会在开始渲染时清除所有东西。如果去看 THREE.EffectComposer 的代码，会看到接下来的三个通道是 marsMask、effectColorify 和 clearMask。首先我们来看这三个通道的定义：

```
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);
```

这三个通道的第一个是 THREE.MaskPass。创建 THREE.MaskPass 就像创建 THREE.RenderPass 一样，也需要传递场景和摄像机。THREE.MaskPass 会在内部渲染场景，但是并不会将结果显示在屏幕上，而是用它来创建掩码。如果将 THREE.MaskPass 添加到 THREE.EffectComposer 对象上，那么后续所有的通道只会被应用到 THREE.MaskPass 定义的掩码上，直到方法 THREE.ClearMaskPass 被执行。这意味着本例中使用的 effectColorify 通道只会应用到被渲染到 sceneMars 的对象上。

我们通过相同的方法将褐色滤镜应用于 Earth 物体。首先为 Earth 场景创建一个遮罩（mask），并将此遮罩应用于 THREE.EffectComposer。然后通过使用 THREE.MaskPass 将我们想要的效果应用于场景渲染（在本例中为 effectSepia，即褐色滤镜效果）。当不再需要使用该

效果时，通过 THREE.ClearMaskPass 将遮罩移除即可。使用 THREE.EffectComposer 的最后一步已经在前面见过：需要再次使用 effectCopy 将处理结果拷贝到屏幕上。有一点值得注意的是，如果 Mars 场景和 Earth 场景有重叠，则只能将这些效果应用于渲染图像的部分区域。

在不同部位同时应用了两个效果的场景如图 11.8 所示。

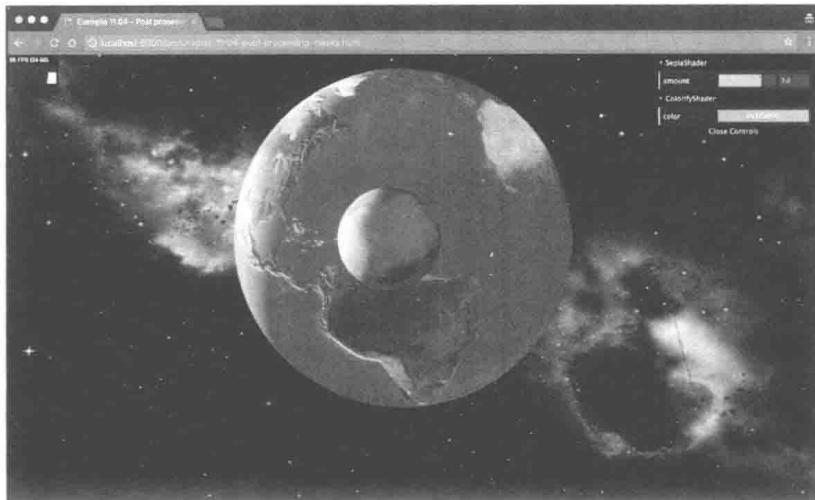


图 11.8

在 THREE.MaskPass 中有一个有趣的属性，即 inverse 属性。如果该属性值为“true”，则掩码会被反转。也就是说，该效果会被应用到所有对象上，除了传递给 THREE.MaskPass 的场景。效果如图 11.9 所示，其中设置了 earthMask 的 inverse 属性为 true。

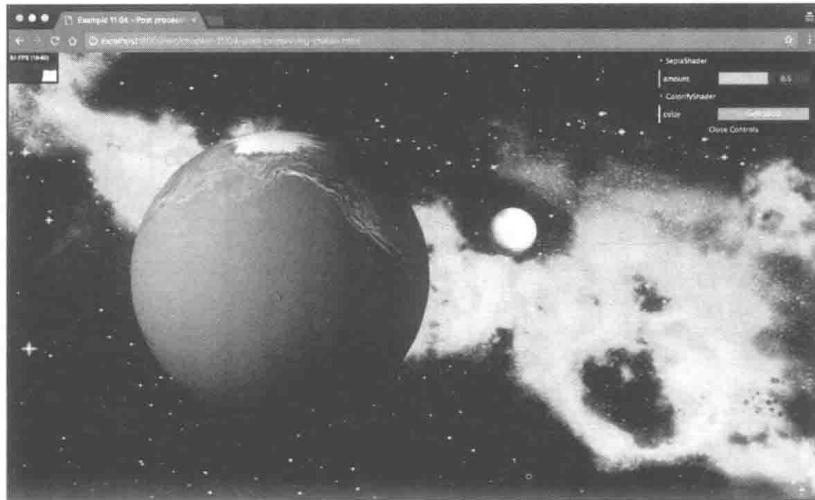


图 11.9

开始讨论 THREE.ShaderPass 之前，继续看两个可以提供先进效果（THREE.BokerPass

和 THREE.SSAOPass) 的通道。THREE.ShaderPass，使用该通道可以自定义效果，而且还有大量的着色器可以使用。

### 11.2.3 高级渲染通道：景深效果

使用 THREE.BokehPass 可以产生景深效果 (Bokeh)。景深效果使得仅有一部分场景处于聚焦范围内，从而可以获得清晰的图像，而其他部分场景则变得模糊。打开示例程序 05-bokeh.html 可以观察景深效果，如图 11.10 所示。



图 11.10

示例程序在初始状态下整个场景都是模糊的。调节右侧菜单上的焦距控制，可以使得一部分场景变得清晰。调节光圈属性可以控制处于聚焦范围的大小。图 11.11 展示了只有球体处于聚焦范围的场景。



图 11.11

如果将焦距拉长，可以使场景中的蓝色立方体处于聚焦范围内，如图 11.12 所示。

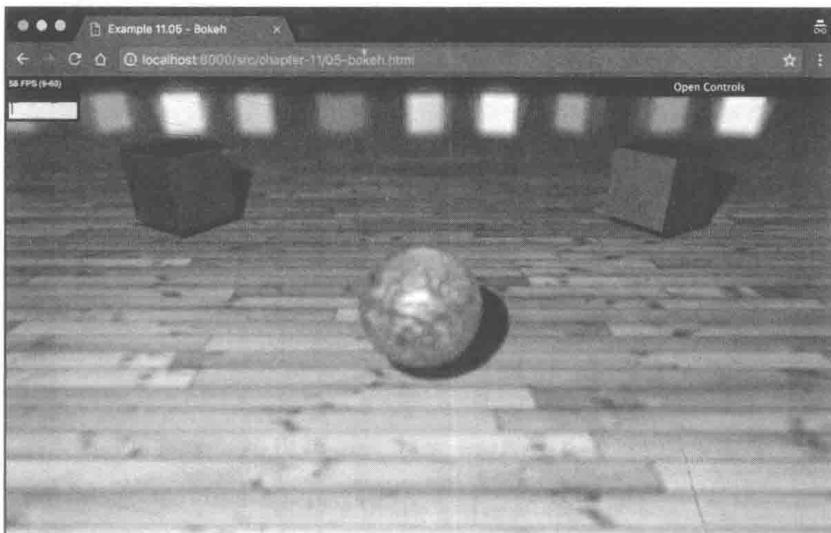


图 11.12

如果继续拉长焦距，则场景中只有最远处的一串绿色立方体处于聚焦范围内，如图 11.13 所示。

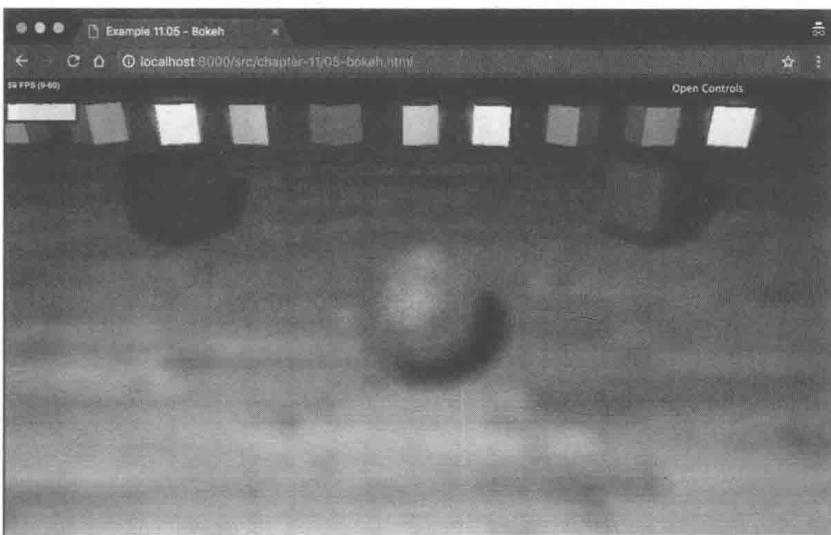


图 11.13

THREE.BokehPass 的使用方法与其他渲染通道的使用方法相同，如下面代码片段所示。

```
var params = {  
    focus: 10,  
    aspect: camera.aspect,  
    aperture: 0.0002,  
    maxblur: 1,
```

```

};

var renderPass = new THREE.RenderPass(scene, camera);
var bokehPass = new THREE.BokehPass(scene, camera, params)
bokehPass.renderToScreen = true;
var composer = new THREE.EffectComposer(renderer);
composer.addPass(renderPass);
composer.addPass(bokehPass);

```

要想获得理想的景深效果，需要仔细调节该渲染通道的属性。

#### 11.2.4 高级渲染通道：环境光遮挡

在第 10 章介绍纹理贴图的时候，我们曾使用预先烘焙的环境光遮挡贴图（aoMap）来直接生成阴影效果。环境光遮挡会影响物体表面的光照强度，使得部分表面接收到的光照较少，从而形成了阴影。除了使用 aoMap 之外，还可以通过在 THREE.EffectComposer 上增加一个渲染通道来实现类似的效果。打开示例程序 06-ambient-occlusion.html 可以看到渲染通道 THREE.SSAOPass 生成的效果，这是两个环境光遮挡效果通道之一，如图 11.14 所示。

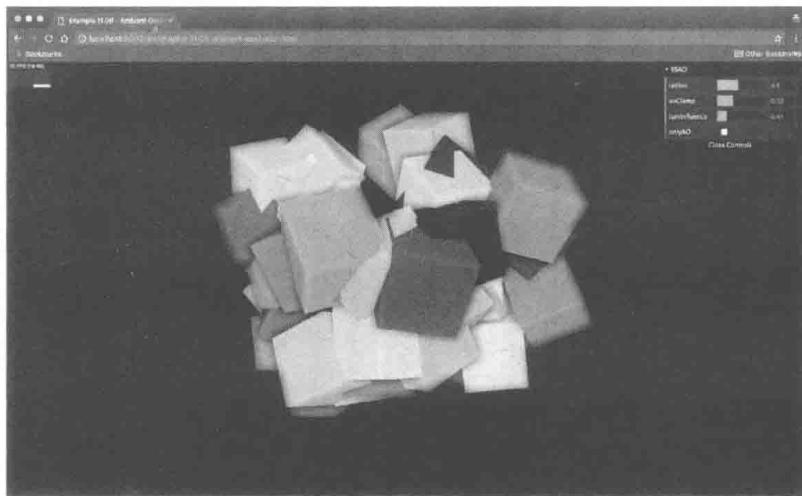


图 11.14

图 11.15 展示了同一个场景在不使用环境光遮挡效果通道时的情形。

在右侧菜单上可以控制效果的强度。从左上角显示的实时帧率可以看出，由 THREE.SSAOPass 生成的环境光遮挡效果对 GPU 资源消耗严重。将其用于静态场景或者场景中的静态部分，不成问题，但将其应用于场景中的动态部分通常会过度消耗 GPU。

到目前为止我们已经尝试了不少 Three.js 自带的标准渲染通道。除此之外，Three.js 还提供了一个支持自定义效果的 THREE.ShaderPass，基于它我们可以尝试使用大量着色器程序。

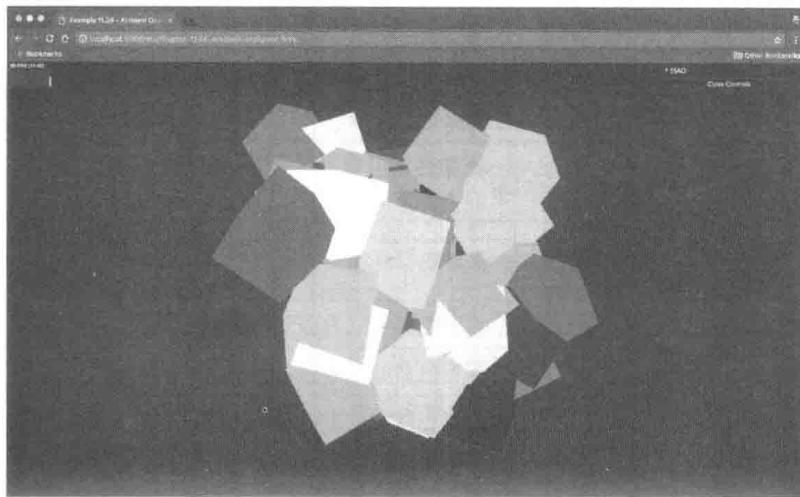


图 11.15

### 11.3 使用 THREE.ShaderPass 自定义效果

基于 THREE.ShaderPass，我们可以通过编写着色器程序来为场景添加更多自定义效果。Three.js 自带了一批现成的着色器程序，本节将它们分成三部分概要介绍。首先来看看表 11.5 所罗列的简单着色器。这些着色器可以在示例程序 07-shaderpass-simple.html 中通过右侧菜单选择使用。

表 11.5

名 称	描 述
THREE.BleachBypassShader	该着色器可以创建出漂白的效果。通过该效果可以使图片看上去像覆盖了一层银
THREE.BlendShader	THREE.BlendShader 并不能为单一场景的后期处理添加效果，但它可以将两个纹理混合在一起。你可以利用这个功能在两个场景之间进行平滑切换（示例程序 07-shaderpass-simple.html 没有展示这个着色器）
THREE.BrightnessContrastShader	该着色器可以改变图片的亮度和对比度
THREE.ColorifyShader	该着色器可以将颜色覆盖在整个屏幕上
THREE.ColorCorrectionShader	通过该着色器可以调整颜色的分布
THREE.FreiChenShader	该着色器提供图像边缘检测功能
THREE.GammaCorrectionShader	该着色器使用固定 Gamma 因数 2 对渲染画面进行 Gamma 校正。除了该着色器之外，也可以通过调节 THREE.WebGLRenderer 的 gammaFactor、gammaInput 和 gammaOutput 三个属性来实现 Gamma 校正

(续)

名 称	描 述
THREE.HueSaturationShader	该着色器可以改变颜色的色调和饱和度
THREE.KaleidoShader	该着色器可以对场景添加类似万花筒的效果，该效果会围绕场景的中央呈径向反射
THREE.LuminosityShader 和 THREE.LuminosityHighPassShader	该着色器提供了亮度效果，可以显示场景的亮度
THREE.MirrorShader	该着色器可以为部分屏幕创建镜面效果
THREE.PixelShader	该着色器可以为画面创建像素化效果
THREE.RGBShiftShader	该着色器可以将构成颜色的红、绿、蓝分开
THREE.SepiaShader	该着色器可以在屏幕上创建类似乌贼墨的效果
THREE.SobelOperatorShader	这是另一个提供图像边缘检测功能的着色器
THREE.VignetteShader	该着色器可以添加晕映效果。该效果可以在图片的中央周围显示暗边框

接下来介绍一些提供模糊效果的着色器。表 11.6 罗列的这些模糊效果可以在示例代码 08-shaderpass-blurs.html 中看到。

表 11.6

名 称	描 述
THREE.HorizontalBlurShader 和 THREE.VerticalBlurShader	这两个着色器可以向整个屏幕添加模糊效果
THREE.HorizontalTiltShiftShader 和 THREE.VerticalTiltShiftShader	这两个着色器可以创建出移轴摄影的效果。该效果使画面中一个横向或竖向的带清晰，其他部分模糊，从而看起来像是一个微缩场景
THREE.TriangleBlurShader	该着色器使用基于三角形的方法在场景中添加模糊效果
THREE.FocusShader	这是一个简单的着色器，其渲染结果是，中央比较尖锐而边界比较模糊

最后为了保证完整性，表 11.7 罗列了几个提供高级效果的着色器，但我们不会介绍这些高级着色器的细节，因为它们要么在其他渲染通道内部使用，要么已经在本章开头介绍渲染通道时介绍过。

表 11.7

名 称	描 述
THREE.FXAAShader	该着色器可以在后期处理阶段添加全屏抗锯齿效果。如果在渲染阶段添加抗锯齿效果的成本昂贵，那么可以使用该着色器来添加
THREE.ConvolutionShader	THREE.BloomPass 使用的内部着色器

(续)

名 称	描 述
THREE.DepthLimitedBlurShader	THREE.SAOPass 使用的内部着色器
THREE.HalftoneShader	THREE.HalftonePass 使用的内部着色器
THREE.SAOShader	着色器版本的实时环境光遮挡效果
THREE.SSAOShader	另一种着色器版本的实时环境光遮挡效果
THREE.SMAAShader	另一种全屏抗锯齿效果
THREE.ToneMapShader	THREE.AdaptiveToneMappingPass 使用的内部着色器
THREE.UnpackDepthRGBAShader	该着色器显示嵌入到 RGBA 纹理中的深度值通道

在 Three.js 的着色器路径下还可以找到一些这里没有提及的着色器，例如 FresnelShader、OceanShader、ParallaxShader 和 WaterRefractionShader。这些并不是后期处理着色器，而是应用于材质的着色器。前面介绍材质时曾介绍过使用自定义着色器的材质。

我们将从简单的着色器开始。

### 11.3.1 简单着色器

我们创建了示例程序 07-shaderpass-simple.html 来展示简单着色器的效果。你可以在示例程序的控制菜单中切换着色器并观察它产生的效果。下面的场景截图展示了这些效果。

图 11.16 展示了 THREE.BrightnessContrastShader 的效果。

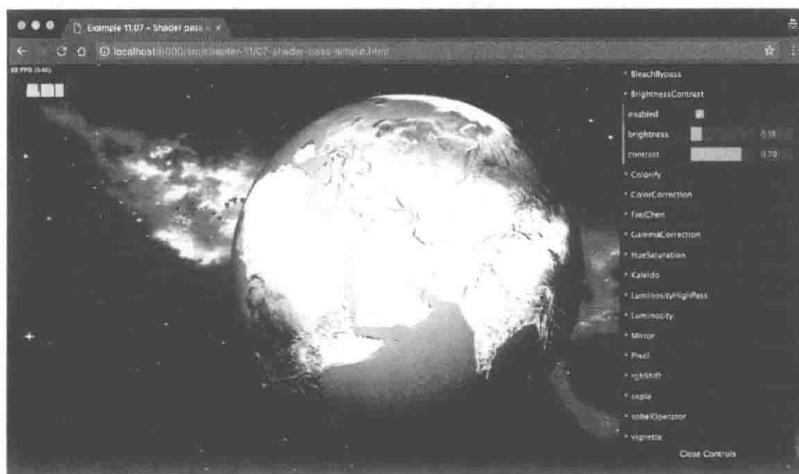


图 11.16

图 11.17 展示了 THREE.FreiChenShader 的效果。

图 11.18 展示了 THREE.KaleidoShader 的效果。

图 11.19 展示了 THREE.MirrorShader 的效果。

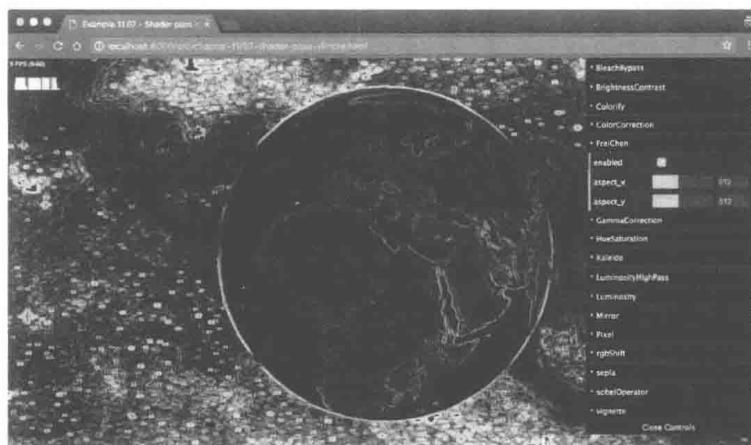


图 11.17

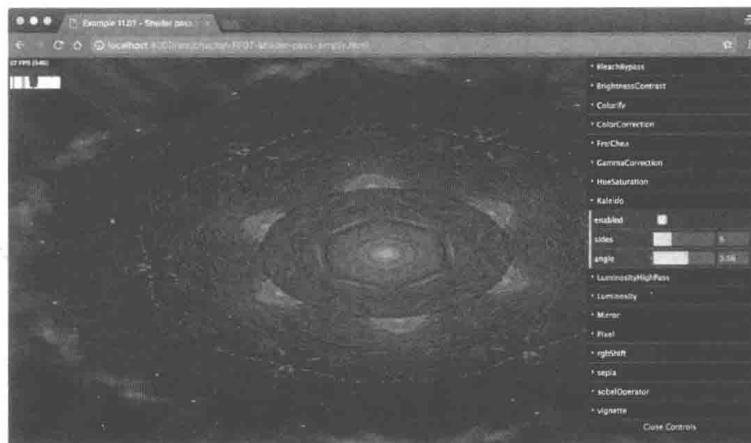


图 11.18

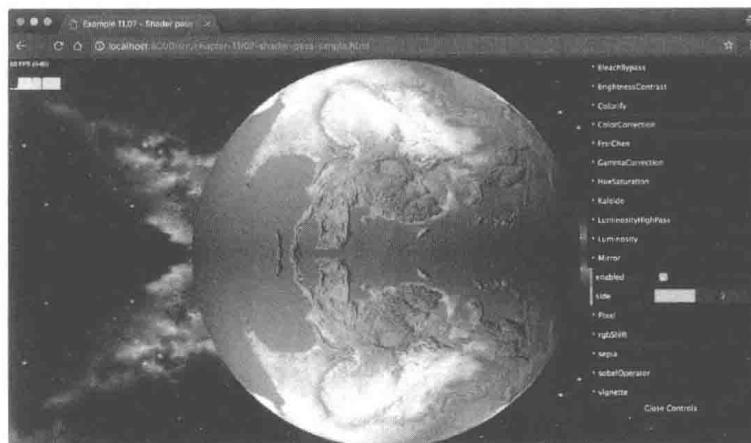


图 11.19

图 11.20 展示了 THREE.RGBShiftShader 的效果。

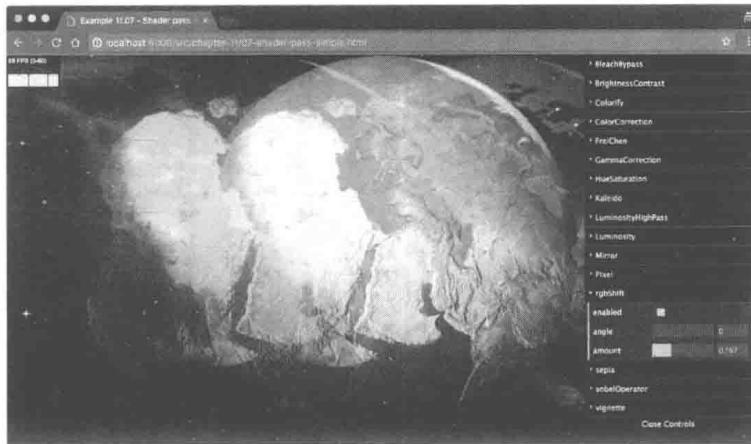


图 11.20

图 11.21 展示了 THREE.SobelOperatorShader 的效果。

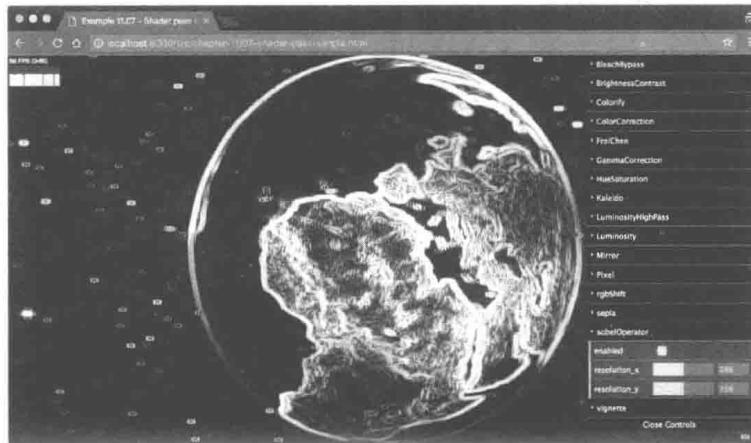


图 11.21

下一节将介绍 Three.js 提供的一些专用于产生模糊效果的着色器。

### 11.3.2 模糊着色器

在本节中我们不会设计代码，只是会展示模糊着色器的处理效果。你可以参见示例 08-shaderpass-blur.html 来体验该着色器。图 11.22 所示的效果是使用 HorizontalBlurShader 和 VerticalBlurShader 着色器处理的。

如图 11.22 所示，你看到的渲染结果是模糊的场景。除了这两个着色器，Three.js 还提供了 THREE.TriangleShader 着色器来使图片变得模糊，该着色器的处理结果如图 11.23 所示。

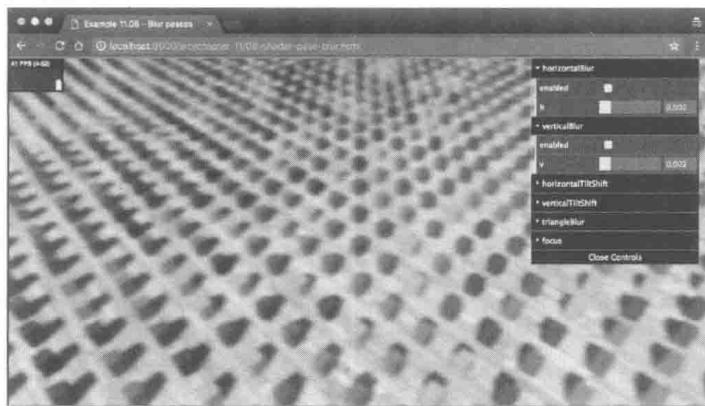


图 11.22

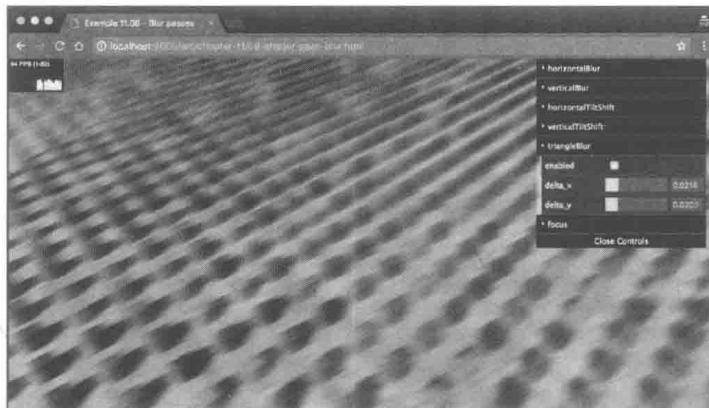


图 11.23

移轴摄影效果是用 THREE.HorizontalTiltShiftShader 和 THREE.VerticalTiltShiftShader 处理的。这两个着色器不会模糊整个场景，而只是模糊一个小区域。这种处理效果被称为倾斜移位，并可以使用该技术从普通照片中创建出微缩的场景。这种效果如图 11.24 所示。

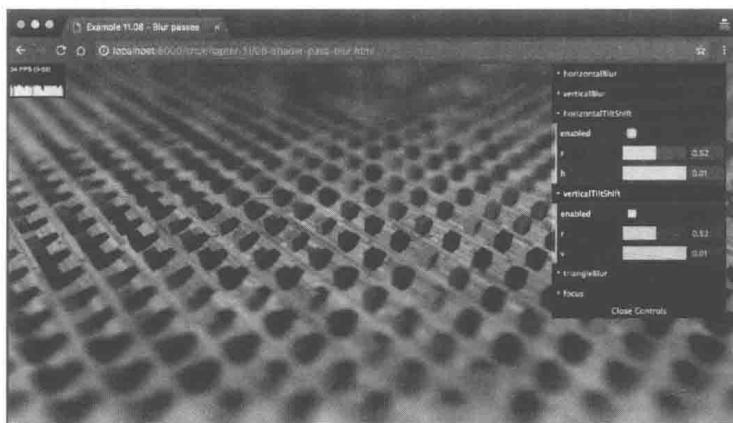


图 11.24

最后一种模糊效果由 THREE.FocusShader 产生，其效果如图 11.25 所示。

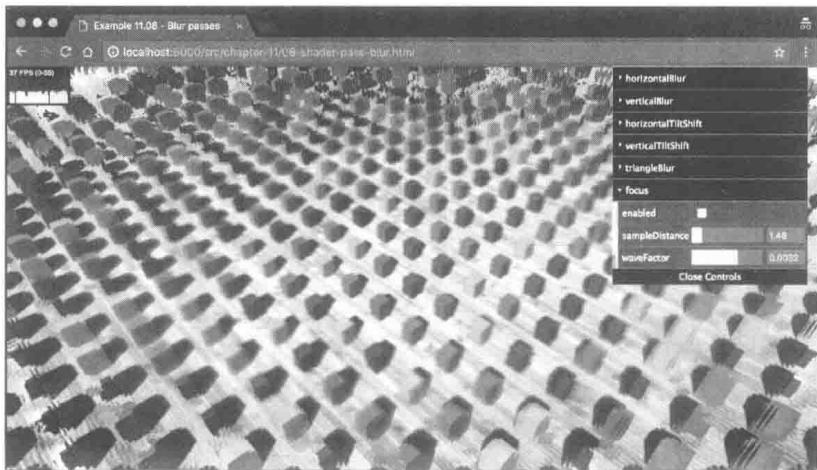


图 11.25

到目前为止，我们使用过的着色器均为 Three.js 自带的，但实际上也可以在 THREE.EffectComposer 中使用自行编写的着色器程序。

## 11.4 创建自定义后期处理着色器

在本节中，你将会学习如何创建出可以在后期处理中使用的自定义着色器。我们会创建两个不同的着色器。第一个着色器可以将当前的图片转换为灰度图，第二个着色器可以减少颜色的可用数目，从而将图片转换为 8 位图。



这里需要指出的是，创建顶点着色器和片段着色器是一个很大的话题，在本节中，我们只会简单介绍这些着色器的功能和工作原理。更多信息可以在 WebGL 规范 (<http://www.khronos.org/webgl>) 中找到。另一个介绍该着色器的资源是 Shadertoy，网址是 <https://www.shadertoy.com/>。

### 11.4.1 自定义灰度图着色器

如果要创建 Three.js(以及其他 WebGL 库)使用的自定义着色器，你需要实现两个组件：顶点着色器 (vertex shader) 和片段着色器 (fragment shader)。顶点着色器可以用于改变每个顶点的位置，片段着色器可以用于定义每个像素的颜色。对于后期处理着色器，我们只需要实现片段着色器就可以了，然后使用 Three.js 提供的默认顶点着色器。在看实现代码之前，我们知道 GPU 通常是可以支持多个着色器管道的，这也就意味着 GPU 在执行顶点着色器时会有多个着色器同时执行。对于片段着色器的执行也是一样的。

下面就来看看着色器的完整代码，该着色器可以在图片上创建出灰度效果（`customShader.js`）：

```
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")
};
```

如你所看到的，这些代码并不是 JavaScript 代码。当你创建着色器时需要使用 OpenGL 着色语言（OpenGL Shading Language, GLSL），它的语法看上去有点像 C 语言。关于 GLSL 的更多信息可以参考 <http://www.khronos.org/opengles/sdk/docs/manglsl/>。

首先我们来看看顶点着色器：

```
"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。从上面的代码来看这不过是 Three.js 库中顶点着色器的标准实现方式。代码中使用的 `projectionMatrix` 表示的是摄像机的投影矩阵，`modelViewMatrix` 表示的是场景中物体的位置到真实世界位置的映射，这两个

共同来决定将物体渲染到屏幕的那个位置。

对于后期处理，需要注意的是代码中的 uv 值，它表示的是 texel（纹理上的像素），该值会通过“varying vec2 vUv”变量传递到片段着色器中。然后我们会通过 vUv 来获取片段着色器中需要的像素值。下面让我们从变量的声明开始来看看片段着色器：

```
"uniform float rPower;,
"uniform float gPower;,
"uniform float bPower;,
"uniform sampler2D tDiffuse;,

"varying vec2 vUv;,"
```

在这里可以看到定义了 uniforms 属性的四个实例变量，这四个变量可以从 JavaScript 传递到着色器中。在本示例中，我们会传递三个浮点数，类型标识为 f（用来决定灰度图中所包含的颜色比例），还会传递一个纹理（tDiffuse），类型为 t。该纹理中包含的是 THREE.EffectComposer 组合器中前一个通道的处理结果。Three.js 会确保这个处理结果能够准确地传递给着色器。除此之外，我们也可以在 JavaScript 中设置其他 uniforms 变量的值。如果要在 JavaScript 中使用这些 uniforms 变量，我们必须定义哪些 uniforms 变量可以在着色器中使用，定义方式是在着色器文件的开头完成，如下所示：

```
uniforms: {

  "tDiffuse": { type: "t", value: null },
  "rPower": { type: "f", value: 0.2126 },
  "gPower": { type: "f", value: 0.7152 },
  "bPower": { type: "f", value: 0.0722 }

},
```

这样我们就可以接收从 Three.js 传递过来的配置参数，以及需要调整的图片。下面来看一下将每个像素转换为灰色的代码：

```
"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 );"
```

这段代码的作用是在传递过来的纹理上获取正确的像素。实现的方式是调用 texture2D 方法，在该方法中传递当前的图片（tDiffuse）和要处理的像素的位置（vUv）。处理的结果就是一个包含颜色和透明度（texel.w）的 texel。

接下来我们将会使用 texel 的属性 r、g 和 b 的值来计算灰度值。这个灰度值会保存在 gl\_FragColor 变量中，并最终显示在屏幕上。这样我们的着色器就定义完成了，而且该着色器和其他着色器的使用方式是一样的。首先需要设置 THREE.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(renderer);
composer.addPass(renderPass);
composer.addPass(shaderPass);
composer.addPass(effectCopy);
```

在渲染循环中调用 `composer.render(delta)` 方法。如果想要在运行期改变着色器的属性，我们只需要改变 `uniforms` 属性的值即可，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;
```

着色器的处理结果可以参见 `09-shaderpass-custom.html`，具体结果如图 11.26 所示。

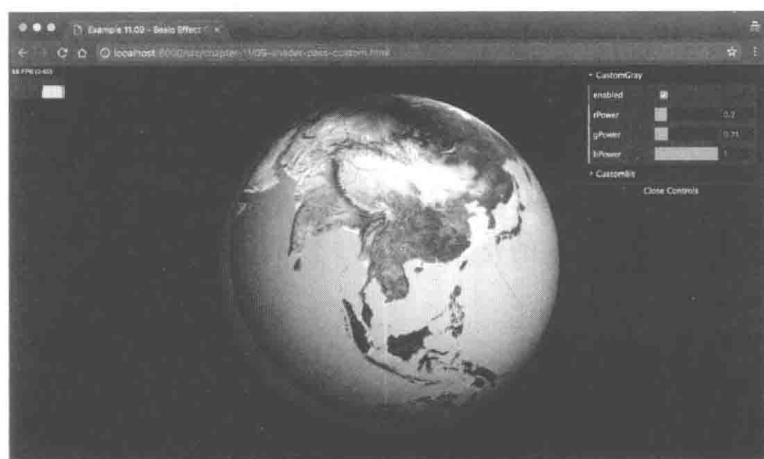


图 11.26

接下来我们自定义另外一个着色器。这次我们需要将输出结果由 24 位降低到 8 位。

### 11.4.2 自定义位着色器

通常来说颜色可以表示为 24 位数值，所以总共大约 1600 万种颜色。但是在早期的计算机中颜色一般用 8 位或者 16 位的数值来表示。使用着色器，我们可以自动将 24 位的颜色输出为 8 位（或者任何你想要的位数）。

由于我们要使用的示例和上一节的示例没太大区别，所以我们直接给出 `uniforms` 属性和片段着色器的定义。`uniforms` 定义如下：

```
uniforms: {
  "tDiffuse": { type: "t", value: null },
  "bitSize": { type: "i", value: 4 }
}
```

片段着色器定义如下：

```

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")

```

如代码所示，我们定义了两个 uniforms 属性变量，这两个变量可以用来对着色器进行配置。第一个 uniforms 属性变量用于传递当前屏幕的渲染结果；第二个 uniforms 属性变量是我们定义的整数类型 (type:"i") 的变量，用于表示我们要渲染的颜色深度。具体流程如下：

- (1) 首先依据传入的像素位置值 vUv，从纹理和 tDiffuse 中获取纹理 texel。
- (2) 然后根据 bitSize 属性计算出可以得到的颜色数量，计算的方式是取 2 的  $\text{bitSize}^{\text{pow}(\text{bitSize}, 2.0)}$  次方。
- (3) 接下来计算纹理 texel 的新颜色值，计算方式是将原颜色的值乘以 n，然后取整数  $\text{floor}(\text{texel.r} * \text{n})$ ，再除以 n。
- (4) 将上述的计算结果赋值给 gl\_FragColor (红、绿、蓝的值以及透明度)，然后显示在屏幕上。

你可以在上一节的示例 07-shaderpass-custom.html 中看到该着色器的处理结果。渲染结果如图 11.27 所示。



图 11.27

到此我们对于后期处理的介绍就结束了。

## 11.5 总结

在本章中我们介绍了各种不同的后期处理方法。如你所见，创建 THREE.EffectComposer 并将各种通道串接起来是很简单的。对于本章的内容需要记住以下几点：

- 不是所有通道的处理结果都会显示在屏幕上。如果想要将通道处理结果输出到屏幕上，你可以使用 THREE.ShaderPass 和 THREE.CopyShader。
- 效果组合器中通道的顺序是很重要的，因为后一个通道是在前一个通道处理结果的基础上进行处理的。
- 如果想要重用 THREE.EffectComposer 特定实例的处理结果，你可以使用 THREE.TexturePass。
- 如果在 THREE.EffectComposer 中有多个 THREE.RenderPass，需要确保 clear 属性被设置为 false。否则，你只会看到最后一个 THREE.RenderPass 的处理结果。
- 如果你只想在特定某个物体上应用某种效果，可以使用 THREE.MaskPass 来实现。当掩码使用完毕，执行 THREE.ClearMaskPass 来清楚该掩码。
- Three.js 除了提供很多标准通道之外，还有大量的标准着色器，你可以通过 THREE.ShaderPass 来使用它们。
- 使用 Three.js 库中的标准方法来创建自定义着色器是很简单的，只需要创建一个片段着色器即可。

到此，我们基本上已经介绍了 Three.js 中所有的内容。在最后一章中，我们将会介绍一个 Physijs 库。这个库可以扩展 Three.js 来实现比如碰撞、重力和约束等物理效果。

## 在场景中添加物理效果和声音



在这最后一章中，我们将会介绍另外一个用来扩展 Three.js 功能的库——Physijs。使用 Physijs 库可以在 3D 场景中添加物理效果。所谓物理效果是指对象会有重力效果，它们可以互相碰撞，施加力之后可以移动，还可以通过合页和滑块在移动过程中在对象上施加约束。这个库是基于另外一个著名的物理引擎 ammo.js 实现的。除了物理效果，我们还会介绍如何使用 Three.js 在场景中添加空间声音。

在本章中我们将会介绍如下几个方面：

- 创建一个 Physijs 场景，在该场景中物体具有重力效果，而且还可以互相碰撞。
- 展示如何改变场景中物体的摩擦系数和复原性（弹性）。
- 介绍 Physijs 提供的各种形体，以及如何使用这些形体。
- 如何利用简单图形创建组合形体。
- 如何通过 height field（高度场）模拟复杂形体。
- 通过添加点、合页、滑块、球销和自由度约束来限制对象的移动。
- 在场景中添加声源，而且声音的大小和方向会由离摄像机的距离来决定。



为 JavaScript 而设计的开源物理效果库有很多，但是它们大部分已经停止更新，Physijs 库便是其中之一。虽然该库已经有几年未见更新，但由于它稳定可靠，经过了完善的测试，并且足够用于展示将物理库与 3D 场景渲染相结合的一般做法，所以本章仍然选择它做重点介绍。实际上，了解 Physijs 库的使用方法之后，也有助于理解如何使用其他物理效果库，因为尽管不同的库提供的类名称和函数名称不同，但是它们的设计思路都大同小异。

首先我们要创建的是一个可以使用 Physijs 的 Three.js 场景。我们将在第一个示例中展示它。

## 12.1 创建基本的 Three.js 场景

创建一个使用 Physijs 的 Three.js 场景是非常简单的，仅需要几个步骤。首先我们需要做的是引入相应的 JavaScript 文件，该文件可以在 GitHub 代码仓库上获取，地址为 <http://chandlerprall.github.io/Physijs/>。在 HTML 页面上添加 Physijs 库的方式如下所示：

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



Physijs 库的点约束功能有一个错误，使得该功能无法正常使用。但在本书提供的 Physijs 库拷贝中，该错误已经被修正。

模拟这样的场景是非常耗费处理器资源的。如果在渲染线程中来做这样的模拟计算的话（JavaScript 实际上是单线程的），场景的帧频会受到严重的影响。为了弥补这一点，Physijs 在后台线程中做计算处理，这里的后台线程是由 web workers（网页线程）规范定义的，而且现在大多数的浏览器都实现了该功能。根据这个规范，你可以在单独的线程里执行 CPU 密集型的任务，这样就不会影响场景的渲染。关于网页线程更多的信息可以参考网址 <http://www.w3.org/TR/workers/>。

对于 Physijs 来说，意味着我们必须配置一个包含执行任务的 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。我们要包含 ammo.js 文件的原因是，Physijs 只是 ammo.js 的包装器。ammo.js（可以在 <https://github.com/kripken/ammo.js/> 找到）是一个实现了物理引擎的库，Physijs 只是对这个物理库做了封装，使其使用起来更加方便。由于 Physijs 只是一个包装器，所以 Physijs 也可以与其他的物理引擎一起工作。在 Physijs 的代码仓库中，你还可以发现一个使用不同的物理引擎 Cannon.js 的分支版本。

为了配置 Physijs，我们还需要设置下面这两个属性：

```
Physijs.scripts.worker = '../libs/other/physijs/physijs_worker.js';
Physijs.scripts.ammo = './ammo.js';
```

第一个属性指向我们所要执行的任务线程，第二个属性指向其内部使用的 ammo.js 库。接下来我们要做的就是创建场景。对于 Three.js 的普通场景，Physijs 还提供了一个包装器，所以你可以使用如下代码来创建场景：

```
var scene = new Physijs.Scene();
scene.setGravity(new THREE.Vector3(0, -10, 0));
```

这样就创建了一个应用了物理效果的新场景，而且我们还设置了重力。在该示例中，我们在 y 轴方向上设置了值为 -10 的重力效果。换句话说，物体会竖直下落。在运行时你也可以在各个坐标轴上将重力设置或者修改成任何你认为合适的值，而且场景会立刻做出相应的反应。

在模拟物理效果之前，我们需要在场景中添加一些对象。我们可以使用 Three.js 中的普通方法来定义对象，但是必须使用 Physijs 的特定对象将这些对象包装起来，这样对象才

能被 Physijs 库管理。代码如下所示：

```
var stoneGeom = new THREE.BoxGeometry(0.6, 6, 2);
var stone = new Physijs.BoxMesh(stoneGeom, Physijs.createMaterial(new
THREE.MeshStandardMaterial({
  color: colors[index % colors.length], transparent: true, opacity: 0.8
})));
scene.add(stone)
```

在该示例中，我们创建了简单对象 THREE.BoxGeometry。接下来我们并没有创建 THREE.Mesh 对象，而是创建了 Physijs.BoxMesh 对象。Physijs.BoxMesh 对象可以告诉 Physijs 在模拟物理效果和碰撞检测时将该网格视为一个盒子。Physijs 还提供了应用于各种图形的网格，本章后面的内容还会做出更详尽的介绍。

现在 THREE.BoxMesh 对象已经添加到场景中了，这样第一个 Physijs 场景的所有组成部分都已经有了。剩下要做的就是告诉 Physijs.js 来模拟物理效果，并更新场景中物体的位置和角度。为此，我们可以在刚创建的场景上调用 simulate 方法。这样我们要对 render 循环做如下的修改：

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

随着最后一步 scene.simulate() 的执行，Physijs 场景的基础配置也就完成了。如果运行这个示例，我们不会看到有多大区别，只会看到屏幕中央有个方块往下落。所以让我们来看一个复杂点的例子：模拟正在倒下的多米诺骨牌。

我们将会在示例中创建如图 12.1 所示的场景。

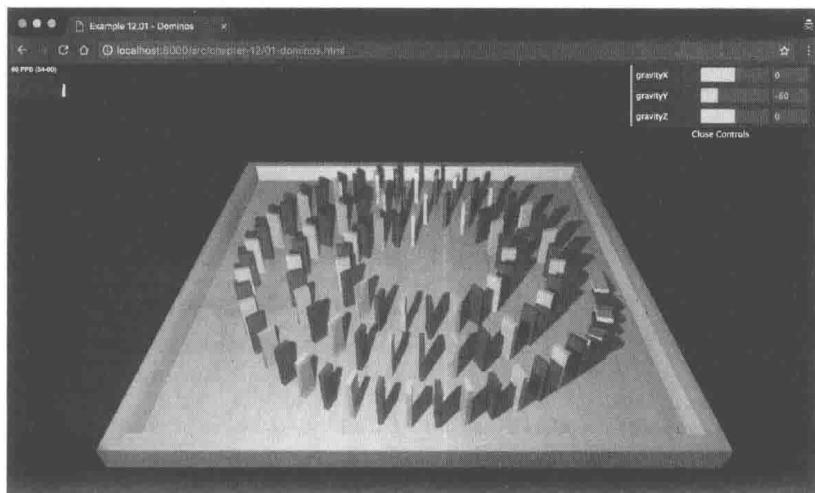


图 12.1

如果在浏览器中打开示例 01-dominos.html，在场景加载完成后你就会看到多米诺骨牌

正在倒下。第一块牌会碰倒第二块牌，依此类推。场景中所有的物理效果都是由 Physijs 操纵的，启动动画我们所要做的就是推倒第一块多米诺骨牌。创建这样的场景是非常简单的，只需要如下的几步：

- (1) 定义一个 Physijs 场景。
- (2) 定义放置多米诺骨牌的地面。
- (3) 放置多米诺骨牌。
- (4) 推倒第一块多米诺骨牌。

由于我们已经知道怎么创建 Physijs 场景了，所以我们跳过第一步直接看第二步如何实现。定义一个包含所有骨牌的地面，表示地面的图形是由几个长方体组合起来的，代码如下所示：

```
function createGroundAndWalls(scene) {
    var textureLoader = new THREE.TextureLoader();
    var ground_material = Physijs.createMaterial(
        new THREE.MeshStandardMaterial(
            {map: textureLoader.load('../assets/textures/general/wood-2.jpg')})
    ),
        .9, .3);

    var ground = new Physijs.BoxMesh(new THREE.BoxGeometry(60, 1, 60),
        ground_material, 0);
    ground.castShadow = true;
    ground.receiveShadow = true;

    var borderLeft = new Physijs.BoxMesh(new THREE.BoxGeometry(2, 3, 60),
        ground_material, 0);
    borderLeft.position.x = -31;
    borderLeft.position.y = 2;
    borderLeft.castShadow = true;
    borderLeft.receiveShadow = true;

    ground.add(borderLeft);

    var borderRight = new Physijs.BoxMesh(new THREE.BoxGeometry(2, 3, 60),
        ground_material, 0);
    borderRight.position.x = 31;
    borderRight.position.y = 2;
    borderRight.castShadow = true;
    borderRight.receiveShadow = true;

    ground.add(borderRight);

    var borderBottom = new Physijs.BoxMesh(new THREE.BoxGeometry(64, 3, 2),
        ground_material, 0);
    borderBottom.position.z = 30;
    borderBottom.position.y = 2;
    borderBottom.castShadow = true;
    borderBottom.receiveShadow = true;

    ground.add(borderBottom);

    var borderTop = new Physijs.BoxMesh(new THREE.BoxGeometry(64, 3, 2),
        ground_material, 0);
```

```

borderTop.position.z = -30;
borderTop.position.y = 2;
borderTop.castShadow = true;
borderTop.receiveShadow = true;

ground.add(borderTop);
scene.add(ground);
}

```

这段代码并不是特别复杂。首先我们创建了表示地面的长方体，然后添加几条边界来防止对象掉到地面的外面。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。Physijs 会将这个复合对象当作单一对象。代码中还有许多东西需要介绍，我们将会在本章后面几节中进行解释。首先是我们创建的 ground\_material，我们调用 Physijs.createMaterial 方法来创建该材质。这个方法对 Three.js 的标准材质进行了封装，但是我们可以在该材质上设置 friction（摩擦系数）和 restitution（弹性系数）。在下一节中还会对这个话题做更多介绍。另外一个比较新的方法是 Physijs.BoxMesh 构造函数的最后一个参数。在本节中我们创建的所有 BoxMesh 对象的最后一个参数都是 0，通过这个参数我们可以设置对象的重量。在这里我们将参数设置为 0，是为了避免地面受场景中重力的影响而下落。

现在我们已经有了地面，就可以放置多米诺骨牌了。为此，我们创建了一个 Three.BoxGeometry 实例对象并包装在 BoxMesh 中，然后放置在地面网格的指定位置，代码如下：

```

var stoneGeom = new THREE.BoxGeometry(0.6, 6, 2);
var stone = new Physijs.BoxMesh(stoneGeom, Physijs.createMaterial(new
THREE.MeshStandardMaterial({
  color: colors[index % colors.length], transparent: true, opacity: 0.8
})));
stone.position.copy(point); // point is the location where to position the
stone
stone.lookAt(scene.position);
stone._dirtyRotation = true;
stone.position.y=3.5;
scene.add(stone);

```

在这里我们并没有把计算每块多米诺骨牌位置的代码列出来（可以参考示例代码中的 getPoints() 方法）。在代码中只给出了如何放置多米诺骨牌。这里你看到的是我们又一次创建了一个 BoxMesh 对象来包装 THREE.BoxGeometry 对象。为了保证多米诺骨牌能够正确地对齐，我们调用 lookAt 方法来设置它们的角度。如果不这样做，它们都会朝向同一个方向，也就不会倒下了。在手动更新了 Physijs 所包装对象的角度（或位置）后，我们必须告诉 Physijs 发生了什么，然后 Physijs 才能对场景中的物体做出更新。对于角度，我们可以将 \_dirtyRotation 属性设置为 “true”；对于位置，我们可以将 \_dirtyPosition 设置为 “true”。

现在要做的就是推倒第一块多米诺骨牌。我们通过将其 x 轴的值设置为 0.2 来实现，也就是稍微推一下。场景中的重力会完成剩下的工作，并将第一块多米诺骨牌完全推倒。代码如下所示：

```

stones[0].rotation.x = 0.2;
stones[0]._dirtyRotation = true;

```

运行该示例程序时，会首先看到第一块多米诺骨牌轻微倾斜，并慢慢倒下撞向下一块

牌，然后便引发多米诺骨牌效果。这样我们的第一个示例就完成了，而且也展示了 Physijs 的一些特性。

如果你想看看重力效果，可以通过右上角的菜单来修改它。例如，如果增大 x 轴的重力值，则会看到如图 12.2 所示的效果。

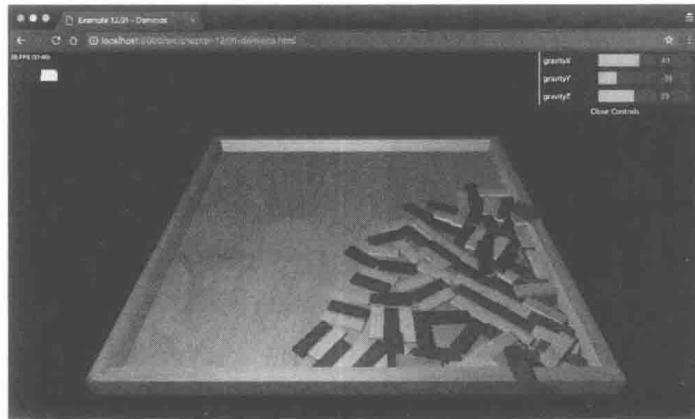


图 12.2

在下一节中我们将会进一步介绍 Physijs 材质的属性是如何影响对象的。

## 12.2 Physi.js 材质属性

我们先对示例进行介绍。当打开示例 02-material-properties.html 时，你会看到一个和上一节示例相似的空盒子。这个盒子围绕 x 轴上下转动，在右上角的菜单中有几个滑块，它们可以用来修改 Physijs 中的材质属性。这些属性可以通过 addCubes 和 addSpheres 按钮来添加到方块和球体上。当点击 addSpheres 按钮时，会有 5 个球体添加到场景中；点击 addCubes 按钮时，会有 5 个方块添加到场景中。如图 12.3 所示。

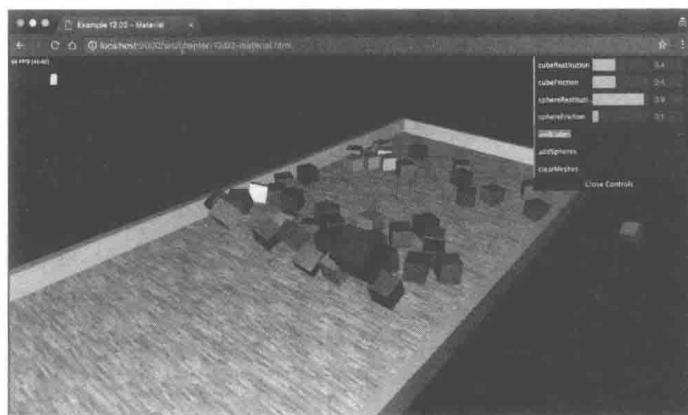


图 12.3

在该示例中可以体验在创建 Physijs 时所设置的 `restitution` 和 `friction` 属性。例如，如果将 `cubeFriction` 设置为 1 并添加一些方块，将会看到即使地面在转动，但这些方块也基本不动。如果将 `cubeFriction` 设置为 0，会发现一旦地面不是水平的，这些方块就开始滑动。图 12.4 给出的是高摩擦系数的效果。

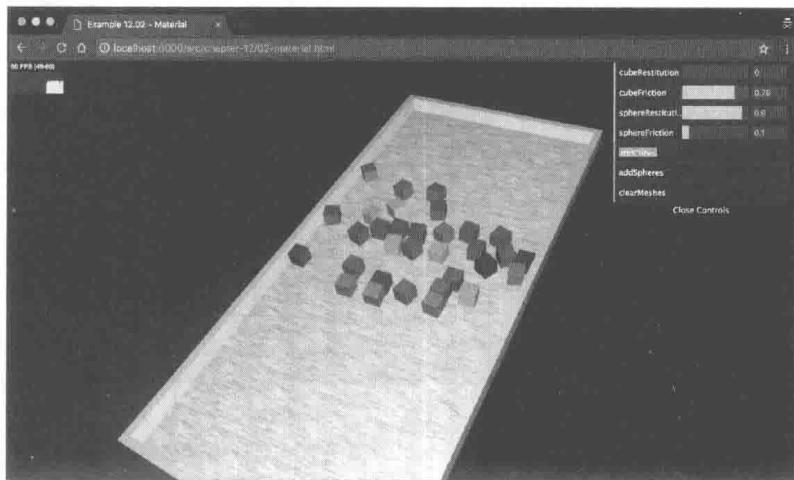


图 12.4

在该示例中可以设置的另外一个属性是 `restitution`。通过该属性可以定义物体在碰撞时所具有的能量。换句话说，`restitution` 值越高，物体越具有弹性，而 `restitution` 值较低则会使得物体在碰撞后立即停下来。



当使用物理引擎时，通常来说你不用担心碰撞检测，因为物理引擎会实现这一点。这对于获知两个物体何时发生碰撞是很有帮助的。比如你可能希望在物体碰撞时播放声音。使用 Physijs，你可以在 Physijs 网格上添加事件监听器，代码如下所示：

```
mesh.addEventListener( 'collision', function( other_object, relative_velocity, relative_rotation, contact_normal ) {
});
```

这样你就会被告知两个物体发生碰撞了。

解释这个概念更好的方式是使用球体并将它的 `restitution` 属性设置为 1，点击 `addSpheres` 按钮几次。这样就可以创建出很多有弹性的球体，如图 12-5 所示。

在进入下一节之前，我们来看看这个示例中的一些代码：

```
var sphere = new Physijs.SphereMesh(new THREE.SphereGeometry(2, 20),
  Physijs.createMaterial(
    new THREE.MeshStandardMaterial({ color: colorSphere }),
    controls.sphereFriction, controls.sphereRestitution));

sphere.position.set(Math.random() * 50 - 25, 20 + Math.random() * 5,
  Math.random() * 50 - 25);
```

```
meshes.push(sphere);
scene.add(sphere);
```

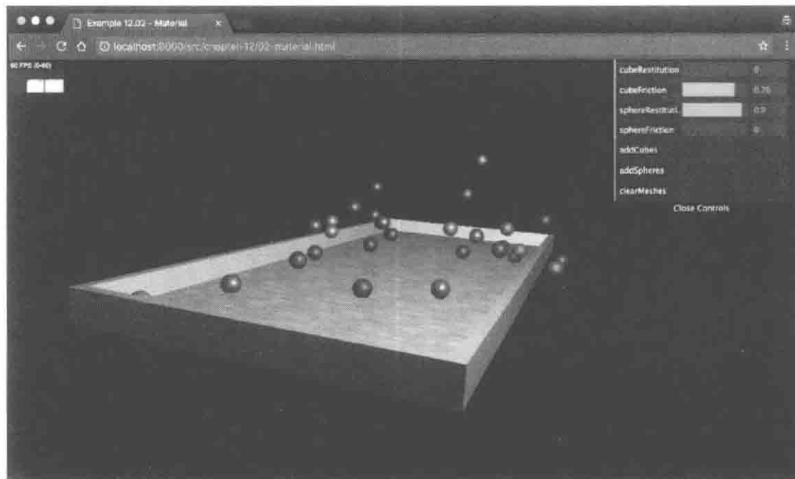


图 12-5

这段代码会在球体添加到场景中时被执行。这次我们使用了另外一种 Physijs 网格：Physijs.SphereMesh。我们创建了一个 THREE.SphereGeometry，而且从逻辑上来说 Physijs 提供的网格中与这个几何体最吻合的就是 Physijs.SphereMesh（在下一节做进一步介绍）。在创建 Physijs.SphereMesh 时，我们传入这个几何体，并用 Physijs.createMaterial 来创建 Physijs 材质。这样做的原因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，我们已经了解了 BoxMesh 和 SphereMesh。在下一节中，我们将会介绍 Physijs 提供的各种网格，你可以使用这些网格包装各种几何体。

### 12.3 Physi.js 基础形体

Physijs 提供了很多用于包装几何体的形体类。在本节中我们将会介绍 Physijs 中的网格并使用示例来阐述它们的用法。使用这些网格你唯一需要做的就是将 THREE.Mesh 的构造函数替换成这些网格对象的构造函数。

表 12.1 是 Physijs 中所有网格对象的概览。

表 12.1

名 称	描 述
Physijs.PlaneMesh	这个网格可以用于创建厚度为 0 的平面。你也可以用 BoxMesh 和 THREE.Box Geometry 一起来表示高度很低的平面
Physijs.BoxMesh	如果几何体看起来像方块，那么可以使用该网格。例如，它和 THREE.Box Geometry 就很匹配

(续)

名 称	描 述
Physijs.SphereMesh	球形可以使用该几何体。这个几何体和 THREE.SphereGeometry 非常匹配
Physijs.CylinderMesh	通过 THREE.Cylinder，你可以创建出各种类型的柱状图形。Physijs 为各种柱形提供了不同的网格。Physijs.CylinderMesh 可以用于顶面和底面半径一样的圆柱体
Physijs.ConeMesh	如果顶面的半径为 0，底面的半径大于 0，那么可以使用 THREE.Cylinder 来创建圆锥体。如果你想在这样的物体上添加物理效果，最好是使用 Physijs.ConeMesh
Physijs.CapsuleMesh	胶囊和 THREE.Cylinder 很像，但是其顶部和底部是圆的。在本节中将会展示如何在 Three.js 中创建胶囊
Physijs.ConvexMesh	Physijs.ConvexMesh 是一种可以用于创建复杂图形的粗略图形。它可以创建模拟复杂图形的凸包（类似 THREE.ConvexGeometry）
Physijs.ConcaveMesh	ConvexMesh 是一种比较粗略的图形，而 ConcaveMesh 则可以对复杂图形进行比较细致的表示。注意，使用 ConcaveMesh 对性能影响很大。通常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到一起（就像前面示例中对地面所做的那样）
Physijs.HeightfieldMesh	这是一个非常特殊的网格。使用该网格，可以从 THREE.PlanetGeometry 中创建一个高度场。参见示例 03-shapes.html

下面借助示例 03-shapes.html 来快速浏览这些图形。我们不会对 Physijs.ConcaveMesh 做过多的解释，因为它的使用范围十分有限。在看示例之前，我们先来看一下 Physijs.PlanetMesh。该网格可以基于 THREE.PlanetGeometry 创建出一个简单的平面，如下所示：

```
var plane = new Physijs.PlanetMesh(new THREE.PlanetGeometry(5, 5, 10, 10),
material);
scene.add(plane);
```

在这个方法中可以看到在创建网格时我们只传入了一个 THREE.PlanetGeometry 对象。如果将该网格添加到场景中，你会看到一些奇怪的事情：创建的网格对重力没有反应。原因是 Physijs.PlanetMesh 的重量固定为 0，所以它不会受重力影响，或者在和其他物体碰撞时发生移动。除了这个网格，其他的网格都会像你期待的那样受重力和碰撞的影响。图 12.6 展示的是一个高度场，其中的物体是可以下落的。

图 12.6 展示的是示例 03-shapes.html。在该示例中，我们随机创建了一个高度场（稍后详解），通过右上角的菜单你可以放置各种形状的对象。如果以该示例做试验，你会发现不同的图形是如何在高度场中运动的，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函数：

```
new Physijs.SphereMesh(new THREE.SphereGeometry(3, 20), mat);
new Physijs.BoxMesh(new THREE.BoxGeometry(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);
```

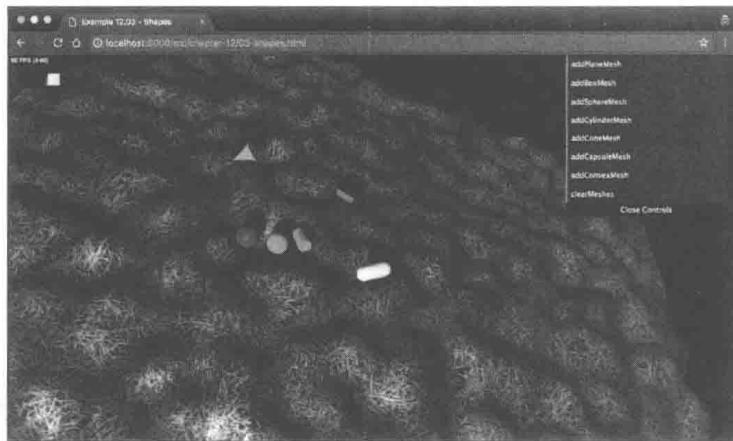


图 12.6

这段代码并没有什么特殊的，我们创建几何体，然后使用 Physijs 中最合适的网格创建添加到场景中的对象。但是如果我们要使用的是 Physijs.CapsuleMesh 该怎么办呢？Three.js 中并没有类似胶囊的几何体，所以我们需要自己创建，代码如下：

```
var merged = new THREE.Geometry(); var cyl = new THREE.CylinderGeometry(2, 2, 6); var top = new THREE.SphereGeometry(2); var bot = new THREE.SphereGeometry(2); var matrix = new THREE.Matrix4(); matrix.makeTranslation(0, 3, 0); top.applyMatrix(matrix); var matrix = new THREE.Matrix4(); matrix.makeTranslation(0, -3, 0); bot.applyMatrix(matrix); // merge to create a capsule merged.merge(top); merged.merge(bot); merged.merge(cyl); // create a physijs capsule mesh var capsule = new Physijs.CapsuleMesh(merged, getMaterial());
```

Physijs.CapsuleMesh 看上去像是圆柱体，但是它的顶部和底部都是圆的。在 Three.js 中我们可以很容易地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 merge() 方法将它们合并到一块。图 12.7 展示的是大量胶囊在高度场中滑落的效果。

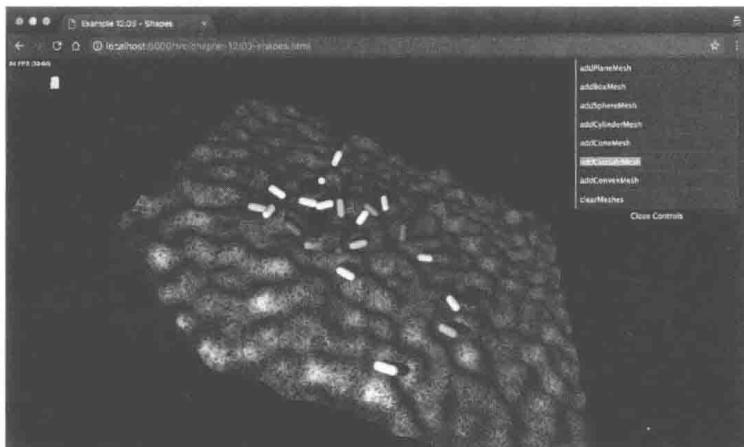


图 12.7

在看高度场之前，我们先来看看最后一个可以添加到示例中的图形对象：Physijs.ConvexMesh。凸包是可以包含几何体所有顶点的最小图形。其结果是所有角度都小于 180° 的图形。你可以在诸如环状扭结这样的复杂图形中使用这个网格，代码如下所示：

```
var convex = new Physijs.ConvexMesh(new
THREE.TorusKnotGeometry(0.5, 0.3, 64, 8, 2, 3, 10), material);
```

如果想要在该示例中模拟物理效果和碰撞，可以使用环状扭结的凸包。这是一种在复杂对象上应用物理效果和碰撞非常好的方法，而且对性能影响最小。Physijs 中最后一个要介绍的网格是 Physijs.HeightMap。图 12.8 展示的是使用 Physijs 创建的高度场。

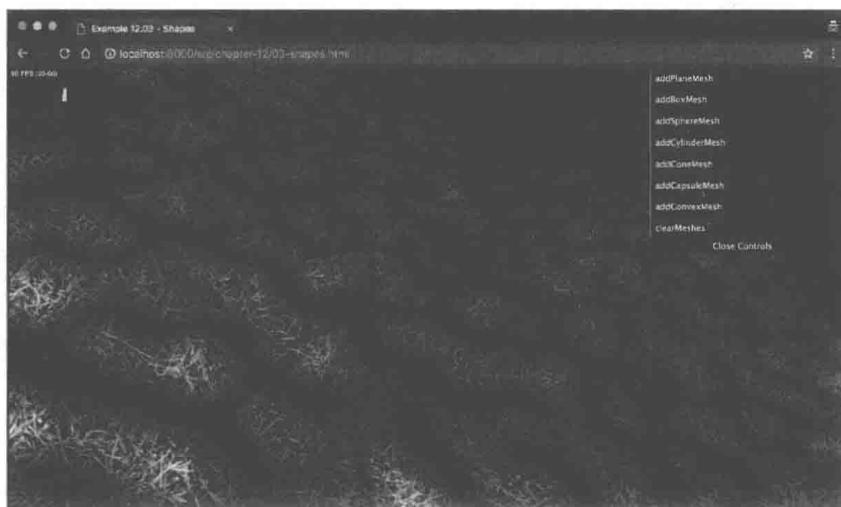


图 12.8

使用高度场可以很容易地创建出一个有凸起和洼地的地形。使用 Physijs.Heightmap 可以确保场景中所有的对象都能够对地形中的不同高度产生相应的反应。让我们看看实现该效果的代码：

```
var date = new Date();
var pn = new Perlin('rnd' + date.getTime());

function createHeightMap(pn) {
    var ground_material = Physijs.createMaterial(
        new THREE.MeshLambertMaterial({
            map: textureLoader.load('../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;
}

```

在这段代码中，我们需要经过好几步才能创建出在示例中所看到的高度场。首先，我们创建了 Physijs 材质和一个简单的 PlaneGeometry 对象。要从 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 噪声生成器，就像我们在第 10 章创建凹凸贴图时一样。我们需要调用 computeFaceNormals 和 computeVertexNormals 方法来保证正确地渲染纹理、光照和阴影。此时，我们用了包含高度信息的 PlaneGeometry 对象。基于 PlaneGeometry 对象，我们可以创建 Physijs.HeightFieldMesh。其构造函数的最后两个参数分别是 PlaneGeometry 对象在水平和垂直方向的分段数，这两个参数应该和 PlaneGeometry 对象构造函数的最后两个参数保持一致。最后，我们需要将 HeightFieldMesh 对象旋转到所需的位置，然后添加到场景中。现在所有的 Physijs 对象就可以和高度场做相应的互动了。

## 12.4 使用约束限制对象的移动

到目前为止我们已经介绍了一些物理效果的例子，包括各种图形是如何对重力、摩擦和弹性做出反应并影响碰撞的。Physijs 还提供了许多高级对象，使用这些高级对象可以限制场景中对象的移动。在 Physijs 中，这些对象称为约束。表 12.2 列出的是 Physijs 所提供的约束。

表 12.2

约 束	描 述
PointConstraint	通过这个约束可以将一个对象和另一个对象之间的位置固定下来。如果其中一个对象移动了，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 HingeConstraint 可以限制对象的移动，使其就像合页一样移动。例如门

(续)

约 束	描 述
SliderConstraint	通过这个约束可以将对象的移动限制在一个轴上。例如推拉门
ConeTwistConstraint	通过这个约束，你可以使用一个对象来限制另一个对象的移动和旋转。这个约束的功能类似于球销式关节。例如胳膊在肘关节中的活动
DOFConstraint	通过这个约束，你可以限制对象在任意轴上的移动，也可以设置对象运动的最小和最大角度。这是最灵活的约束方式

观察示例程序运行是理解这些约束的最好方式。为此我们提供了一批使用约束的示例。下面首先介绍 PointConstraint。

#### 12.4.1 使用 PointConstraint 限制对象在两点间移动

打开示例程序 04-point-constraint.html 之后可以看到如图 12.9 所示的画面。

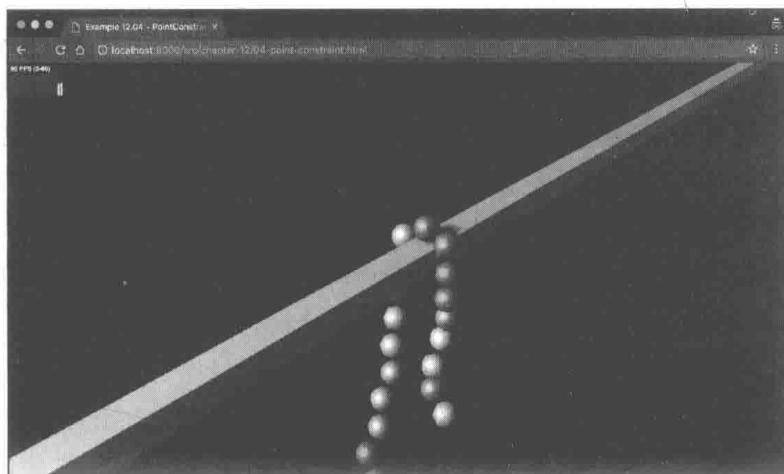


图 12.9

在该示例程序的场景中有一串珠子落在一个水平杆上，并在珠串接触杆时弯曲地挂在了杆上，这是因为每一个珠子都通过 PointConstraint 与相邻珠子构成位置约束，从而使得没有接触到杆子的珠子也不会继续下落。也正因此，当珠串的一侧比另一侧重时，在重力和约束的共同作用下，珠串会向较重的一侧滑动并最终从杆上落下。

本例中的 PointConstraint 创建方法如下：

```
function createPointToPoint(scene) {
    var beads = [];
    var rangeMin = -10;
    var rangeMax = 10;
    var count = 20;
    var scale = chroma.scale(['red', 'yellow']);
```

```

for (var i = 0 ; i < count ; i++) {
    var bead = new THREE.SphereGeometry(0.5);
    var physBead = new Physijs.SphereMesh(bead, Physijs.createMaterial(
        new THREE.MeshStandardMaterial({color:
            scale(Math.random().hex()), 0, 0}));
    physBead.position.set(i * (-rangeMin + rangeMax)/count + rangeMin, 10,
    Math.random()/2);
    scene.add(physBead);
    if (i != 0) {
        var beadConstraint = new Physijs.PointConstraint(beads[i-1],
        physBead, physBead.position);
        scene.addConstraint(beadConstraint);
    }
    physBead.castShadow = true;
    beads.push(physBead);
}
}

```

如代码所示，我们使用特定的 Physijs 网格（本例中用的是 SphereMesh）来创建对象，然后将它们添加到场景中。我们使用 Physijs.PointConstraint 构造函数创建约束。这个约束有三个参数：

- 前两个参数指定需要连接的两个对象。在本例中我们将两个球体进行连接。
- 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较大的对象上，你可以将这个位置设置在那个对象的右边。通常，如果你要将两个对象连接在一起，那么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上，而是绑定到场景中某个固定的点上，那么你可以忽略第二个参数。在这种情况下，第一个对象会与你指定的位置保持固定的距离，同时还遵从重力和其他物理规律。

当约束创建好后，我们可以通过 addConstraint 方法将它添加到场景中，从而使该约束生效。当你开始体验约束时，你可能会遇到一些奇怪的问题。为了方便调试，你可以在 addConstraint 方法中传递一个参数“true”。这样就能在场景中看到约束点和方向，帮助你获取约束的旋转角度和位置。

## 12.4.2 使用 HingeConstraint 创建类似门的约束

HingeConstraint 为合页约束，用于创建类似合页的物体。该约束使物体绕一个特定的轴、在一个特定的角度范围内运动。在示例程序 05-sliders-hinges.html 中，画面底部的两个绿色滑块被设定了 HingeConstraint 约束，如图 12.10 所示。

两个滑块通过 HingeConstraint 固定在两个绿色小方块上并绕它们转动。如果想观察滑块的运动效果，可以通过点击菜单上的 flipUp 和 flipDown 按钮来激活它们。在 motor 菜单上可以设置它们被激活时的初速度。

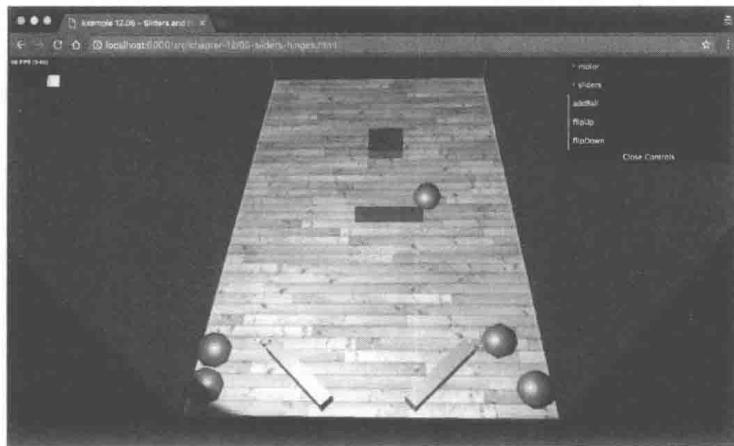


图 12.10

下面让我们来看看如何创建其中的一个挡板：

```
function createLeftFlipper(scene, mat) {
    var flipperLeft = new Physijs.BoxMesh(new THREE.BoxGeometry(12, 2, 2),
    mat, 10);
    flipperLeft.position.x = -8;
    flipperLeft.position.y = 2;
    flipperLeft.position.z = 30;
    flipperLeft.castShadow = true;
    scene.add(flipperLeft);

    var flipperLeftPivot = new Physijs.SphereMesh(new THREE.BoxGeometry(1, 1,
    1), mat, 0);
    flipperLeftPivot.position.y = 2;
    flipperLeftPivot.position.x = -15;
    flipperLeftPivot.position.z = 30;
    flipperLeftPivot.rotation.y = 1.4;
    flipperLeftPivot.castShadow = true;

    scene.add(flipperLeftPivot);

    var constraint = new Physijs.HingeConstraint(flipperLeft,
    flipperLeftPivot, flipperLeftPivot.position,
                           new THREE.Vector3(0, 1, 0));
    scene.addConstraint(constraint);

    constraint.setLimits(-2.2, -0.6, 0.3, 0.5);
    return constraint;
}
```

该约束接受 4 个参数。每个参数的含义如表 12.3 所示。

在场景中添加 HingeConstraint 方法和添加 PointConstraint 的方法是一样的。调用 add Constraint 方法并指定想要添加的约束，如果为了调试则可以添加 true 参数用来显示约束的确切位置和方向。但是对于 HingeConstraint 来说，我们还需要定义约束的活动范围。我们通过 setLimits 方法来实现。

表 12.3

参 数	描 述
mesh_a	该函数的第一个参数表示的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。该对象的移动将会被约束
mesh_b	该函数的第二个参数指定 mesh_a 对象受哪个对象的约束。在本示例中 mesh_a 对象受褐色小方块的约束。如果我们移动这个褐色小方块，那么 mesh_a 也会跟着移动，而且 HingeConstraint 始终起作用。例如，如果你已经创建了一个四处移动的汽车，想给为门的打开添加约束。如果忽略第二个参数，那么合页将会被约束到场景（不能四处移动）
position	该参数表示约束被应用到的点位置。在本例中就是 mesh_a 绕着旋转的点。如果你指定了 mesh_b，那么这个点就会随着 mesh_b 的位置和角度来移动
axis	该参数指定合页可以旋转的角度。在本例中，我们将合页设置在水平方向 (0, 1, 0)

setLimits 方法接受 4 个参数，如表 12.4 所示。

表 12.4

参 数	描 述
low	该参数指定旋转的最小弧度
high	该参数指定旋转的最大弧度
bias_factor	该参数指定处于错误位置时约束进行修正的速度。例如，当合页被某个对象推出约束范围之外时，该合页可以自动移到正确的位置。这个值越大，则自我修正的速度越快。该值最好保持在 0.5 以下
relaxation_factor	该参数指定约束以什么样的比率改变速度。该值越大，那么对象在达到最大或者最小角度时会被反弹回来

你也可以在运行时改变这些属性。如果在添加 HingeConstraint 时用到了这些属性，那么你将不会看到对象的移动。这些网格只会在被其他对象碰撞或者受到重力影响时才会移动。但是这个约束像其他约束一样，也可以由内部的电动机驱动。这就是我们在示例中点击 flipperUp 按钮时所看到的现象，如图 12.11 所示。

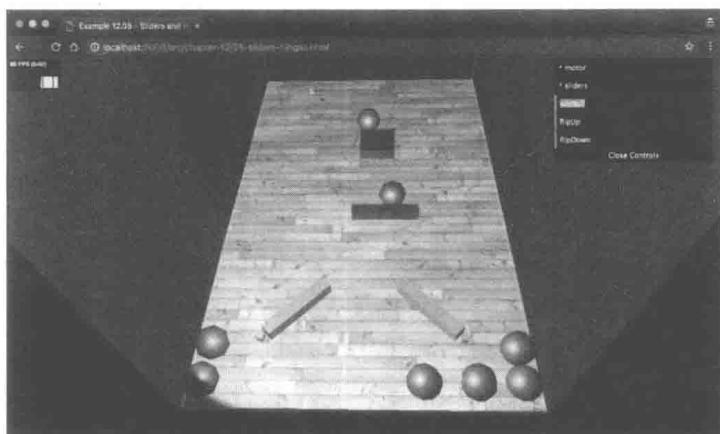


图 12.11

下面是电动机激活的代码：

```
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的限制，那么只要电动机不停，挡板就会一直旋转。如果要关闭电动机，可以执行如下的代码：

```
flipperLeftConstraint.disableMotor();
```

现在这个网格就会受摩擦、碰撞、重力和其他物理因素影响而逐渐停下来。

### 12.4.3 使用 SliderConstraint 将移动限制在一个轴上

接下来要介绍的约束是 SliderConstraint。通过这个约束，你可以将某个对象的移动限制在某个轴上。示例程序 05-slidershinges.html 中滑动条子菜单上的滑动条可以控制画面中的蓝色滑块。这个滑块只能沿特定的轴运动。顶部的滑块只能上下移动，而底部的只能左右移动，如图 12.12 所示。

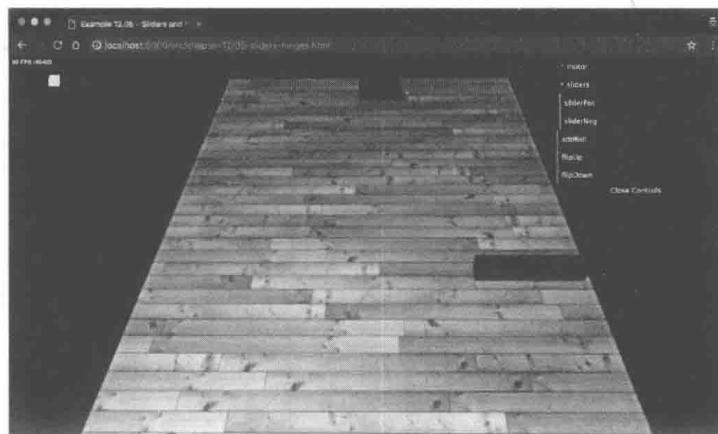


图 12.12

点击 sliderPos 按钮可以将滑块移动到左手边（下限）；点击 sliderNeg 按钮可以将滑块移动到右手边（上限）。创建这些约束的代码是很简单的：

```
function createSliderTop(scene, mat) {
    var sliderSphere = new THREE.BoxGeometry(7, 2, 7);
    var sliderMesh = new Physijs.BoxMesh(sliderSphere, mat, 100);
    sliderMesh.position.z = -15;
    sliderMesh.position.x = 2;
    sliderMesh.position.y = 1.5;
    scene.add(sliderMesh);

    //position is the position of the axis, relative to the ref, based on the
    //current position
    var constraint = new Physijs.SliderConstraint(sliderMesh,
        new THREE.Vector3(-15, 2, 1.5),
        new THREE.Vector3(Math.PI / 2, 0, 0));
    scene.addConstraint(constraint);
    constraint.setLimits(-18, 18, 0.5, -0, 5);
```

```

constraint.setRestitution(0.1, 0.1);

return constraint;
}

```

如代码所示，这个约束接受 3 个参数（也可以是 4 个，如果你想将一个对象约束到另一个对象）。该约束的参数含义如表 12.5 所示。

表 12.5

参 数	描 述
mesh_a	第一个参数表示的是所要约束的对象。在示例中，这个对象是表示滑块的绿色方块。这个对象的移动将会受到限制
mesh_b	第二个参数指定 mesh_a 受哪个对象的约束。这是一个可选的参数，在本例中忽略了该参数。如果忽略该参数，那么 mesh_a 将会受到场景的约束。如果指定了该参数，那么当指定的网格移动或转动时，滑块也会跟着移动
position	该参数指定约束所应用的位置。该参数在将 mesh_a 约束到 mesh_b 时比较重要
axis	这个参数指定 mesh_a 沿着哪个轴移动。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 当前版本中，如果使用线性电动机和线性上下限时，轴的方向会有点偏移。所以，在当前版本中沿着某个轴移动的代码如下所示： • 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)

当创建好约束并调用 `scene.addConstraint` 方法将约束添加到场景之后，可以设置应用这个约束所滑动的范围：`constraint.setLimits(-10, 10, 0, 0)`。对于 SliderConstraint 可以设置如表 12.6 所示的限制：

表 12.6

参 数	描 述
linear_lower	该参数指定对象的线性下限
linear_upper	该参数指定对象的线性上限
angular_lower	该参数指定对象的角度下限
angular_higher	该参数指定对象的角度上限

最后，可以设置在达到某个极限时的弹性。可以使用方法 `constraint.setRestitution(res_linear, res_angular)` 来设置，其中第一个参数指定达到线性限制时的弹性，第二个参数指定达到角度限制时的弹性。

现在约束就配置好了，我们只需要滑动滑块或者使用电动机然后等待发生碰撞即可。对于 SlideConstraint，我们还有两个选项：可以使用角度电动机来加速对象围绕指定的轴旋转并遵从角度约束；或使用线性电动机来加速对象沿指定轴移动并遵从线性约束。在这个示例中，我们使用的是线性电动机。角度电动机将会在 12.4.5 节讨论 DOFConstraint 时介绍。

#### 12.4.4 使用 ConeTwistConstraint 创建类似球销的约束

使用 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  $x$ 、 $y$  和  $z$  轴上的最小角度和最大角度。理解 ConeTwistConstraint 的最好方式就是看看创建该约束的代码：

```

var baseMesh = new THREE.SphereGeometry(1);
var armMesh = new THREE.BoxGeometry(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.MeshPhong
        Material({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 代码中，你会发现许多前面已经介绍过的概念。首先我们创建了几个用约束连接起来的对象：objectOne（一个球体）和 objectTwo（一个盒子）。调整这些对象的位置使得 objectTwo 在 objectOne 的下面。现在我们就可以创建 ConeTwistConstraint 了。如果你已经看过其他约束了，你会发现这个约束的参数没有什么不一样。第一个参数指定所要约束的对象，第二个参数指定第一个参数要约束到的对象，最后一个参数指定约束所应用到的位置（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，我们可以使用 setLimit 方法设置它的限制。该函数接受三个弧度值来表示对象绕每个轴旋转的最大角度。

与其他的约束一样，我们可以使用该约束提供的电动机驱动 objectOne。对于 ConeTwist Constraint，我们可以设置 MaxMotorImpulse 属性（即电动机能够施加的力），以及电动机可以将 objectOne 转动的角度。

### 12.4.5 使用 DOFConstraint 实现细节的控制

使用 DOFConstraint，即自由度约束，可以准确地控制对象的线性方向和角度方向的移动。下面我们将通过类似小车图形的示例来介绍如何使用该约束。这个小车图形用一个方块表示车身，四个圆柱表示车轮。首先我们来创建车轮：

```
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;
}
```

如代码所示，我们创建了一个简单的 CylinderGeometry 和 CylinderMesh 对象来表示小车的轮子。如图 12.13 所示。

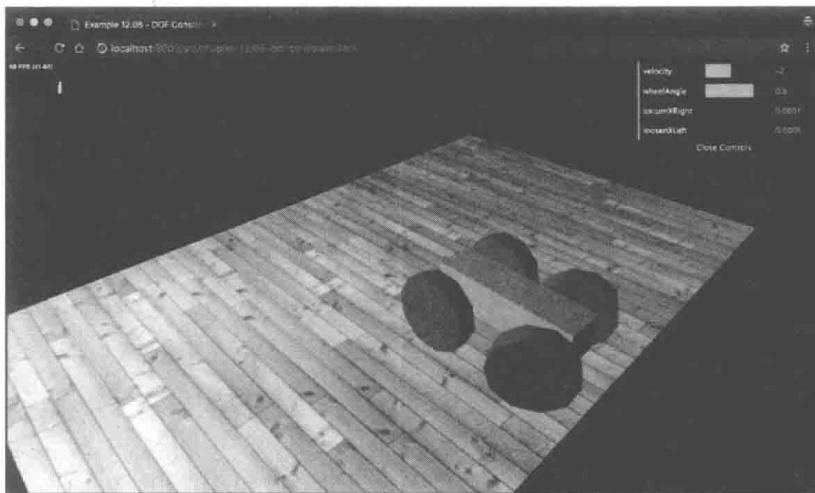


图 12.13

接下来我们创建车身，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.BoxGeometry(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);

```

到现在我们创建的只是小车的各个组成部分，如果要将这些组件组合到一起，就要创建约束了。每个轮子都被约束到车身上，创建约束的代码如下所示：

```

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）都有自己的约束，第二个参数指定车轮绑定到车身的那个位置。如果以这样的配置运行代码，我们会发现车轮将车身拖起来了。要让小车动起来，我们还需要做两件事情：为车轮设置约束（沿着哪个轴移动）并将配置正确的电动机。首先我们为两个前轮设置约束，让它们只能沿着 z 轴移动来拉动汽车。当然，这两个轮子是不能沿着其他轴移动的。约束的设置代码如下所示：

```

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 });

```

这段代码看起来比较奇怪。通过将上下限设置成一样的，可以保证对象在指定方向上不会旋转，这也意味着车轮不会沿着 z 轴旋转。我们这样做的原因是：当在指定轴上启动电

动机时，该轴上的限制就会被忽略。所以这里对 z 轴上设置的限制对于这两个前轮来说是没有任何影响的。

现在我们要驱动后面的两个车轮了。为了保证它们不会落下，我们需要将 x 轴固定下来。通过下面的代码，我们可以：固定 x 轴（将上下限设置为 0）；固定 y 轴，就像轮子已经转动一样；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 });
```

如你所看到的，如果要取消指定轴上的限制，我们需要将下限的值设置得比上限的值稍微大些。这样就可以绕这个轴自由旋转了。如果我们不这样设置 z 轴，那么这两个后轮只能被拖着走。在本示例中，由于和地面之间的摩擦作用，这两个后轮将会和其他的轮子一起移动。剩下要做的就是为前轮设置电动机了：

```
flConstraint.configureAngularMotor(2, 0.1, 0, -2, 1500);
frConstraint.configureAngularMotor(2, 0.1, 0, -2, 1500);
```

由于有三个轴的存在，所以创建电动机时需要指定其工作的轴：0 是 x 轴，1 是 y 轴，2 是 z 轴。第二个和第三个参数指定电动机的角度限制。这里我们再次将下限设置得比上限稍微高一点，从而使得电动机可以自由转动。第三个参数指定的是我们想要达到的速度，最后一个参数指定电动机可以施加的力。如果最后一个参数的值过小，那么小车是不会移动的；如果值过大，两个后轮将会脱离地面。

启动电动机的代码如下所示：

```
flConstraint.enableAngularMotor(2);
frConstraint.enableAngularMotor(2);
```

打开示例 06-dof-constraint.html，你可以体验使用各种约束和电动机来驱动小车。效果如图 12.14 所示。

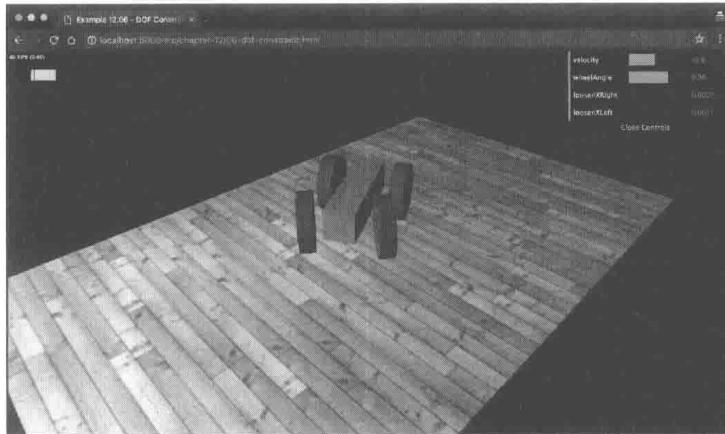


图 12.14

下一节将介绍本书的最后一个主题：如何在 Three.js 场景中添加声音。

## 12.5 在场景中添加声源

到目前为止，我们介绍了许多可以用来创建酷炫场景、游戏和任意三维效果的组件，但是我们还没有介绍如何在 Three.js 场景中添加声音。在这一节中我们将介绍往场景中添加声音的两个 Three.js 对象。声源是很有意思的，因为声源会受到摄像机距离的影响：

- 声源离摄像机的距离决定着声音的大小。
- 摄像机左右侧的位置分别决定着左右侧扬声器声音的大小。

解释它们最好的方式就是看看它们的效果。在浏览器中运行示例 07-audio.html，你将会看到三个带有动物图片的方块，如图 12.15 所示。

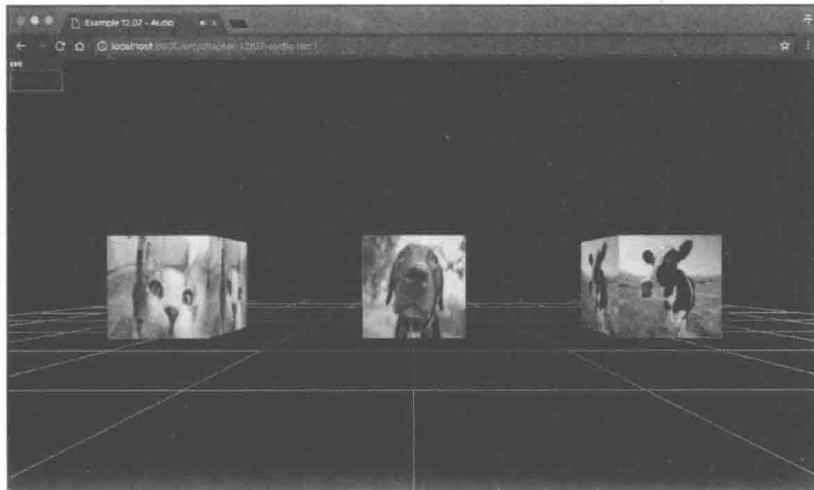


图 12.15

该示例使用的是第一视角控制器。使用鼠标来移动场景，你会发现离哪个方块近，哪个动物的声音就会大。如果将摄像机放置在狗和奶牛的中间，你会听到牛的声音来自右侧，狗的声音来自左侧。



在该示例中，我们使用 Three.js 中的辅助类 THREE.GridHelper 来创建方块下面的方格：

```
var helper = new THREE.GridHelper( 500, 10 );
scene.add( helper );
```

要创建方格，需要指定方格的大小（本例中大小为 500）和每个小方格的大小（本例为 10）。当然，你也可以通过设置 color1 和 color2 属性来指定水平线的颜色。

该示例的实现只需要很少的代码。首先我们需要定义 THREE.AudioListener 对象并添

加到 THREE.PerspectiveCamera，代码如下：

```
var listener1 = new THREE.AudioListener();
camera.add(listener1);
var listener2 = new THREE.AudioListener();
camera.add(listener2);
var listener3 = new THREE.AudioListener();
camera.add(listener3);
```

接下来我们需要创建 THREE.Mesh 并添加一个 THREE.PositionalAudio 对象到网格上，代码如下：

```
var cube = new THREE.BoxGeometry(40, 40, 40);
var material_1 = new THREE.MeshBasicMaterial({
  color: 0xffffffff,
  map: textureLoader.load("../assets/textures/animals/cow.png")
});

var mesh1 = new THREE.Mesh(cube, material_1);
mesh1.position.set(0, 20, 100);

var posSound1 = new THREE.PositionalAudio( listener1 );
var audioLoader = new THREE.AudioLoader();
audioLoader.load('../assets/audio/cow.ogg', function(buffer) {
  posSound1.setBuffer( buffer );
  posSound1.setRefDistance( 30 );
  posSound1.play();
  posSound1.setRolloffFactor(10);
  posSound1.setLoop(true);
});;
```

如代码所示，首先我们创建了一个标准的 THREE.Mesh 对象实例，然后创建一个 THREE.PositionalAudio 对象并关联到 THREE.AudioListener 对象上，最后加载音频文件并通过设置一些属性来定义声音如何播放和表现。

- ❑ **setRefDistance**: 该属性指定声音从距离声源多远的位置开始衰减其音量。
- ❑ **setLoop**: 该属性指定声音是否被循环播放。默认为只播放一遍。
- ❑ **setRolloffFactor**: 该属性指定声源音量随着距离衰减的速度。

我们为这个示例程序添加了主视角控制，因此你可以在场景中来回走动。在移动过程中你会注意到声音的音量和位置随着摄像机所在的位置变化。例如，假如你站在奶牛正前方，你基本上只能听到奶牛的声音。而猫和狗的声音只能在左声道里听到很微弱的一点，如图 12.16 所示。

Three.js 在内部使用 Web Audio API (<http://webaudio.github.io/web-audio-api/>) 来播放声音以及设置音量，但并不是所有的浏览器都支持这个 API。目前只有 Chrome 和 Firefox 能够较好的支持。

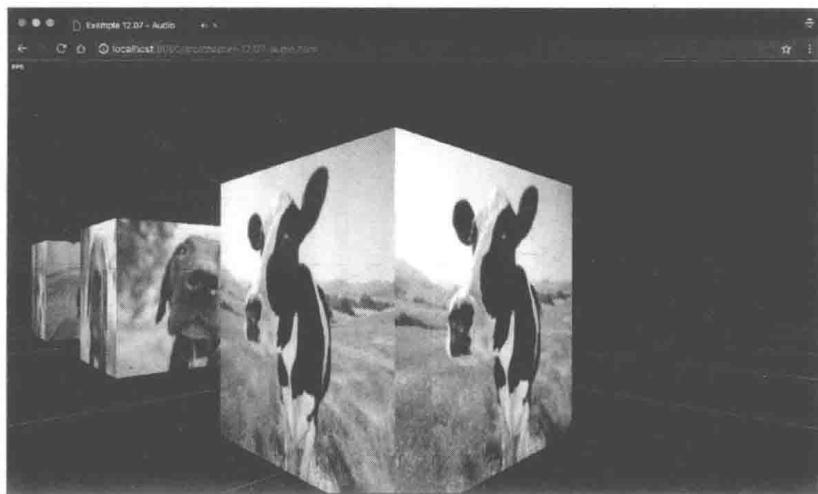


图 12.16

## 12.6 总结

在最后一章中，我们介绍了如何通过添加物理效果来扩展 Three.js 的基础三维功能。为此，我们使用了 Physijs 库，通过该库我们可以添加重力、碰撞和约束等。我们还介绍了如何使用 THREE.Audio 和 THREE.AudioListener 在场景中添加固定的声源。这就是本书所有的内容了，这些章节基本涵盖了 Three.js 的所有功能。在开始的几章，我们介绍了 Three.js 的核心概念和基本理论。然后我们介绍了可用的光源和材质是如何影响对象的渲染的。之后我们又介绍了 Three.js 提供的各种几何体，以及如何组合这些几何体来创建新的几何体。

本书的后半部分介绍了一些比较高级的主题。介绍了如何创建粒子系统，如何从外部资源加载模型，以及如何创建动画。最后，在后面几章我们介绍了高级纹理和场景渲染完成后使用的后期处理效果。我们以讲解物理效果作为本书的最后一章。在这一章中，除了介绍如何在 Three.js 场景中添加物理效果，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添加更多的功能。

我希望你能够愉快地阅读本书并体验书中提供的示例。

# Learning Three.js

Programming 3D animations and visualizations  
for the web with HTML5 and WebGL, Third Edition

## Three.js开发指南

基于WebGL和HTML5在网页上渲染3D图形

(3版)

现代浏览器都支持WebGL，这样不必使用Flash、Java等插件就能在浏览器中创建三维图形。然而，直接使用WebGL在浏览器中创建三维图形和动画也非常烦琐，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，帮助你充分利用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：

- 使用Three.js提供的各种材质并了解它们如何与3D模型和场景相互作用
- 通过Three.js提供的各种摄像机控制功能，在三维场景中轻松导航
- 通过直接操作顶点实现雨、雪以及宇宙星系效果
- 导入OBJ、STL、COLLADA等外部格式的模型和创建动画效果
- 创建和运行基于形态和框架的动画
- 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
- 创建自定义顶点和片段着色器，实现与WebGL的直接交互

[PACKT]  
PUBLISHING

投稿热线：(010) 88379604

购书热线：(010) 68326294

客服热线：(010) 88379426 88361066

华章网站：[www.hzbook.com](http://www.hzbook.com)

网上购书：[www.china-pub.com](http://www.china-pub.com)

数字阅读：[www.hzmedia.com.cn](http://www.hzmedia.com.cn)



上架指导：计算机/Web开发

ISBN 978-7-111-62884-2



9 787111 628842

定价：99.00元