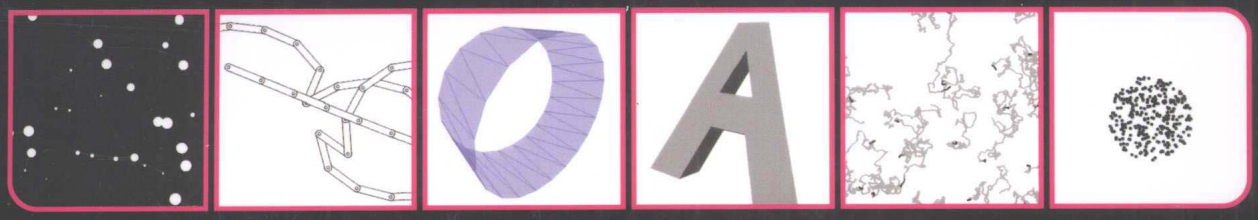
HTML5+JavaScript 动画基础

[美] Billy Lamberta 著
Keith Peters
徐宁 李强 译

friendsof
an Apress company

人民邮电出版社
POSTS & TELECOM PRESS



在本书中，你将学到：

- 通过代码实现动画所需的 JavaScript 和 HTML5 代码（包括数学和三角函数）；
- 基本运动原则，如速度、加速度、摩擦力、缓动、反弹等；
- 如何通过键盘和鼠标处理用户交互；
- 高级动画技术，如弹动、坐标旋转、动量守恒定律、正向运动学和反向运动学；
- 使用 HTML5 制作 3D 效果必须掌握的所有 3D 基本概念——从简单的透视图到三维实体，包括背面剔除和动态灯光。

HTML5+JavaScript 动画基础

通过本书，你将进入 Web 编程创意的新时代。本书详细介绍了如何使用 HTML5 的 canvas 元素来创建高级 Web 图形和动态脚本动画。它涵盖了要完成各种动画项目必需掌握的所有技能——从游戏到导航系统，从广告到教育应用程序。

在本书中，作者 Billy Lamberta 首先清晰地介绍了所有相关的数学知识，然后引入了加速度、速度向量、缓动、弹动、碰撞检测、动量守恒、3D、正向运动学和反向运动学等物理概念。他还帮你建立了一个工具集，你可以把这些工具融入自己编写的任意动画脚本中来创建动态效果。

在任何时候，你都能够理解隐藏在脚本动画背后的概念，而且还可以创建各种各样令人激动的动画和游戏。对于有使用 HTML5 或从 Flash 转到 HTML5 的 Web 开发人员来说，本书正是你们梦寐以求的，它可以引领你创建各种能够在当前所有浏览器和大多数移动设备（包括 iPhone、iPad 和 Android 设备）上运行的完全符合标准的游戏、应用程序、动画。

美术编辑：王建国

分类建议：计算机 / 程序设计 / HTML
人民邮电出版社网址：www.ptpress.com.cn

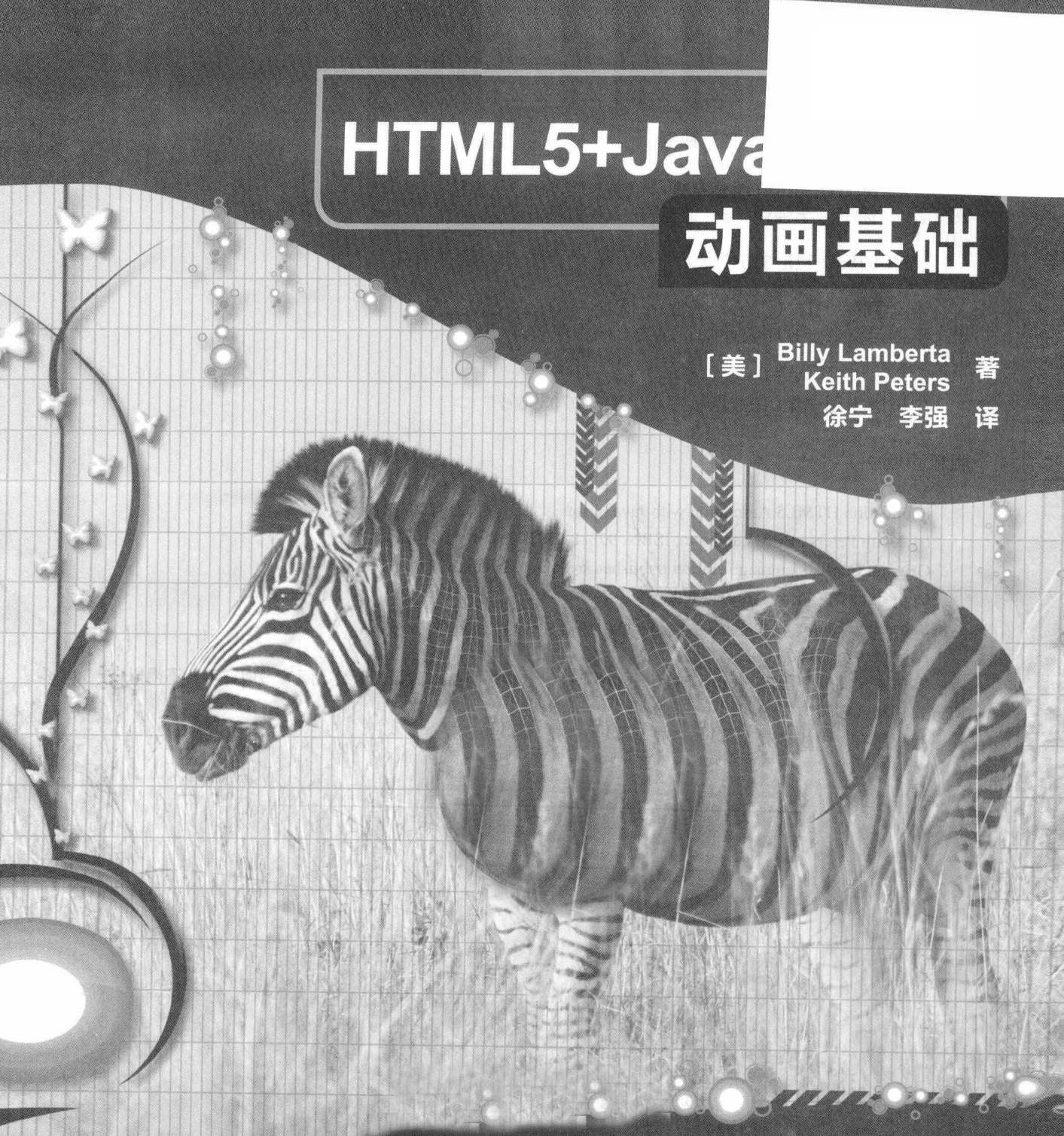
ISBN 978-7-115-31547-2

ISBN 978-7-115-31547-2
定价：69.00 元

HTML5+Java

动画基础

[美] Billy Lamberta
Keith Peters 著
徐宁 李强 译



人民邮电出版社
北京

图书在版编目（CIP）数据

HTML 5+JavaScript动画基础 / (美) 兰贝塔, (美) 彼得著 ; 徐宁, 李强译. -- 北京 : 人民邮电出版社, 2013.6
ISBN 978-7-115-31547-2

Ⅰ. ①H… Ⅱ. ①兰… ②彼… ③徐… ④李… Ⅲ. ①超文本标记语言－程序设计②JAVA语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第070390号

版权声明

◆ 著 [美] Billy Lamberta
Keith Peters
译 徐 宁 李 强
责任编辑 杜 洁
责任印制 程彦红 杨林杰

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京中新伟业印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：26
字数：553 千字 2013 年 6 月第 1 版
印数：1-3 000 册 2013 年 6 月北京第 1 次印刷

著作权合同登记号 图字：01-2012-0699 号

定价：69.00 元

读者服务热线：(010)67132692 印装质量热线：(010)67129223
反盗版热线：(010)67171154

内容提要

本书包括了基础知识、基础动画、高级动画、3D 动画和其他技术 5 大部分，分别介绍了动画的基本概念、动画的 JavaScript 基础、动画中的三角学、渲染技术、速度向量和加速度、边界与摩擦力、用户交互：移动物体、缓动与弹动、碰撞检测、坐标旋转与斜面反弹、撞球物理、粒子与万有引力、正向运动学：让事物行走、反向运动学：拖曳与伸出、三维基础、三维线条与填充、背面剔除与三维灯光、矩阵数学、秘诀与技巧等内容。

这些内容都是 Web 开发人员在深入如加速度、速度、缓冲、弹簧、碰撞检测、动量守恒、3D 以及正向和反向运动物理概念之前，需要知道的所有关于三角函数的知识。在阅读本书的过程中，读者不但可以掌握脚本动画背后的概念，还可以创造出各种形式的精彩动画和游戏。

本书面向所有使用 HTML5 或从 Flash 转过来的 Web 开发人员。

译者序

HTML5 指的是万维网（WWW）核心语言 HTML 的第 5 个版本。目前，万维网中主要采用的 HTML 正式版本为 1999 年发布的 4.01 版，这个版本在万维网发展过程中发挥了巨大作用。然而，随着万维网从面向文档（Document-Based）的方法向应用（Application）和移动（Mobile）等方向发展，HTML 4.01 已经不能满足现代万维网在应用开发和移动性等方面的新需求了，HTML 需要在语法、语言和 API 等方面进行革新，因此，HTML5 应运而生。目前，HTML5 仍处于草案阶段，但是已经引起了浏览器厂商及 Web 开发人员的广泛关注，基于这一点，我们看到了它巨大的发展潜力和创新意义。

在 HTML5 的诸多新特性中，HTML5 Canvas 是其中最吸引人的特性之一。它由 JavaScript 脚本进行控制，可以动态地绘制出各种 2D 图形，甚至可以对图像像素进行任意处理。目前，HTML5 Canvas 已经得到了很多浏览器的支持，包括 Mozilla Firefox、Google Chrome、IE、Safari 和 Opera 等在内。而且，有许多的网页游戏厂商和开发者也已经开始使用它来开发网页游戏。我们可以大胆地想象，在不久的将来，采用 HTML5 开发的无插件网页游戏将广泛流行。

本书是面向游戏和娱乐应用开发的 HTML5 图书，在书中我们将通过最流行的 Web 前端开发语言 JavaScript 操纵 Canvas 来实现各种各样的动画效果，通过各种各样的示例可以触发你联想到如何将它们组合在一起以实现你的创意。并且，在每一章的最后，还会抽象出一些跟当前话题相关的实用的 JavaScript 函数，使得你可以在将来的应用中重用它们。本书共有 19 章内容，根据讨论的主题可以分成如下 5 部分。

第一部分，基础知识，包括第 1～4 章。分别讲述动画的背景知识与 JavaScript 基础知识，帮助读者了解 HTML5 动画的一些基本概念，为读者掌握本书后续内容提供所需要的 JavaScript 基础知识，同时也介绍了动画中三角学知识以及如何用 JavaScript 实现动画的渲染。

第二部分，基础动画，包括第 5～7 章。介绍了速度、速度向量以及动画中对边界和摩擦力的处理，并在最后引出了如何实现用户交互的技术和手段。

第三部分，高级动画，包括第 8～14 章。循序渐进地讲解在如何实现各种高级动画的效果，其中包含了大量的示例。

第四部分，3D 动画；包括第 15～17 章。追赶最新的 3D 潮流，这部分介绍了如何实现 3D 动画的基础知识。

第五部分，其他技术，包括第 18～19 章。在本书的最后，引入了矩阵的概念，你会发现之前一些复杂的动画效果在它的帮助下变得简单了许多。

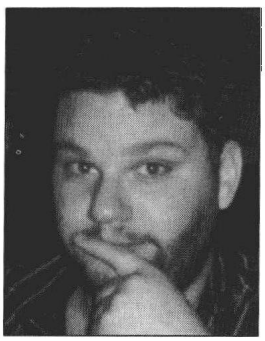
正如作者所介绍的，你可以把这本书想象成一个包含了各种动画元素的目录。

在这里，特别感谢人民邮电出版社的编辑们，是他们的见识使这本好书有可能尽早地与国内读者见面。感谢我的家人和朋友们在我翻译过程中对我的支持，特别是我的女朋友，因为在翻译这本书的期间少了很多陪伴她的时间。由于本书覆盖面广，翻译难度也较大，我们虽然在翻译中下了不少工夫，但译文仍难免会出现一些疏漏，恳请认真阅读的同行和朋友们不吝赐教。

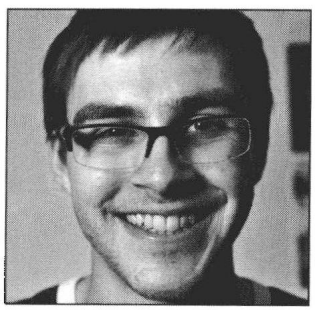
徐　宁

2013 年 5 月

技术审核者介绍

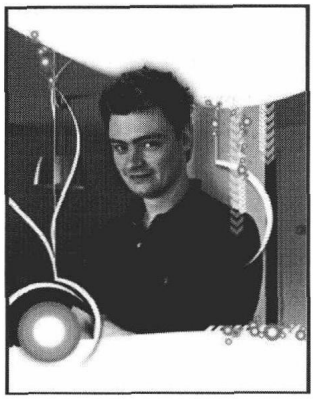


Brian Danchilla 是一名 PHP 与 Java 开发人员，他曾经开发过 Web、数值分析、图形与网络电话应用。Danchilla 拥有计算机科学与数学专业双学士学位并在近期与他人合作了《Pro PHP Programming》一书，该书由 Apress 出版社在 2011 年出版。在业余时间，他喜爱弹奏吉他以及户外运动。他居住在加拿大萨斯卡通。



Robert Hawkes 擅长通过代码解决问题。他沉迷于可视化编程技术，并对 HTML5 canvas 上瘾。当他不在睡觉的时候，他的大部分时间都花在了涉及各种令人兴奋的新技术的疯狂的项目中，其中既包含在线项目也包括离线项目。除了他的本职工作，Rob 还为 Apress 出版社撰写了《Foundation HTML5 Canvas》一书，该书围绕如何使用新的 Web 技术编写游戏展开深入的介绍。他同时还是 Mozilla 基金会的技术布道者之一。

封面设计者简介



Corné van Dooren 为本书设计了封面。在短暂离开他在 friends of ED 出版社的工作后，他又开始为 Foundation 系列图书创作新的设计，这次他尝试将技术与大自然界中的动植物组合在一起，从而使其作品出现在了包括本书在内的多本书的封面上。

Corné 的童年沉浸在手上作画中，并由此开启了他探索广阔的多媒体世界的征程，至今也从未停下。他的座右铭就是“想象力是多媒体技术的唯一束缚”，这句话推动他不断前进。

Corné 服务于众多国际客户，他为多媒体杂志撰写专题文章，评审并测试软件，发表多媒体研究报告并参与设计 friends of ED 出版社的众多书籍。你可以通过他的网站 www.cornevandooren.com 了解他的更多工作并与他联系。

致谢

本书对于那些曾经在 JavaScript 和 Flash 社区中贡献了无数程序与教程并使他人受益的人们深表谢意。我个人的编程知识很大程度受益于免费软件的蓬勃发展，以及人们乐于分享代码的行为。谢谢你们对我的教育。

谢谢你 Keith Peters，没有你就没有这本书的诞生。我从你的工作中学到了很多知识并非常荣幸可以使其增色，虽然只有一点点。

感谢 Apress 出版社的所有工作人员为本书的辛勤劳动，以及他们为本书挑出的瑕疵，正是由于你们的帮助，这本书才得以出版。特别感谢 Ben Renow-Clarke 与 Adam Heath 对本书的指导，以及 Brian Danchilla 与 Rob Hawkes 为完善本书所提供的宝贵意见。

最后，感谢我的家人：感谢我那充满远见的父母为其孩子提供了一台疯狂并且不切实际的家用计算机，他们甚至需要我教他们怎么开机。感谢我妹妹长期以来对我的支持。当然，感谢你 Jessica，谢谢你有耐心陪我度过整个写作过程。我爱你们每个人。

前言

本书介绍了如何借助计算机代码和数学知识在 Web 上创建交互式的动画。你无须为记不清高中代数课的内容而感到担心，你只需有对它们稍微有一些了解就可以开始本书的学习。本书并不是为了让你记住各种数学公式和理论，而是教给你一些可以用于实现和表达创意的工具。虽然在此过程我们也会介绍一些根本性的原理，但是更加重要的是，告诉你如何将技术实际运用到你的工作中。你将看到各种概念和公式如何在你面前即时地发挥作用。尝试将本书想象成各种运动元素，或包含各种创意组合、竞赛与引用的目录。

本书中会出现大量可供学习的示例，并且当你看到自己创建的各种物体在屏幕上运动时，你会很有成就感，因为它们看上去栩栩如生。而当你把这些动画以一个可以在 Web 浏览器中访问的链接的形式分享给你的朋友们时，你将获得更大的成就感，而这恰恰得益于经由互联网发布的巨大优势。

本书是基于 Keith Peters 的优秀图书《Foundation ActionScript Animation》重写的。然而，不同之处在于，那本书针对的是 Flash 技术，本书采用了诸如 HTML5 与 JavaScript 等最近流行的 Web 技术。本书秉承着上一本书的数学原理发展而来，它们在这一方面是相通的。当你理解了一些基础模块后，你就不会再依赖于一些具体的开发工具，而能够将这些概念运用于你所涉猎的各种编程环境。

由于书中的示例都是通过 HTML5 与 JavaScript 实现的，因此接下来我们会指导你学习那些用于理解示例所需的特定编程技术。JavaScript 是一门有趣、强大而又相对精简的语言，不过因为它的灵活性，它的用法可以变得千奇百怪。不同于那些只能以某种特定方式编程的、更加结构化的语言，JavaScript 允许你编写风格迥异的代码。这种自由度虽然可以带来非常强大的功能，但是它也会为初学者掌握这门语言的主要思路增加困惑。如果这是你首次接触 JavaScript，最好能够在学习本书的示例前简要略读一段关于它的介绍。在学习 JavaScript 的过程中，最大的困惑往往来自于你从其他语言中带来的种种假设。请将相关的参考文档放在你的手边，如果有任何疑惑，直接到浏览器的开

发控制台中测试代码。如果你是 Flash 开发人员，请抵制住将 JavaScript 想象成 ActionScript 的某个变种的诱惑。JavaScript 有它自己独特的程序结构与风格，如果能够在一开始的时候摒弃那些先入为主的观点，则可以为以后的学习避免很多麻烦。

本书的编写过程充满了乐趣，我也希望你能够在阅读它的时候享受其中的快乐。编写你的程序，体验各种各样的动画效果，分享它们并向他人学习。创新是一个主动的过程，请不要守株待兔，让我们开始编码吧！

作者简介

Billy Lamberta 是一名程序员和多媒体实践者。他在家乡弗吉尼亚州里士满市曾经从事过电视新闻摄影记者的职业，之后他将他的注意力转向了 Web 开发以及使用 Flash 和 JavaScript 的交互式程序设计。出于讲故事的目的，Billy 对于影像资料的会聚很有兴趣，并致力于将它们发布在开放的网络上。他生活在纽约州的水牛城，并热爱观看冰球比赛。

译者简介

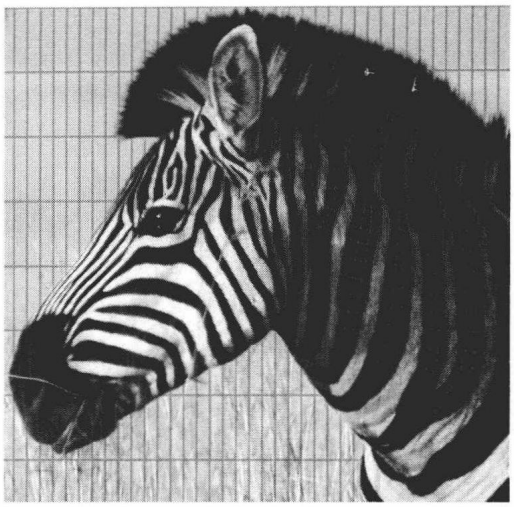
徐宁，软件架构师，从2001年开始接触.NET开发，于2007年7月获得C#方向的微软MVP称号。现任职于道富银行技术中心，从事金融软件架构设计的工作。在博客园（idior.cnblogs.com）曾发表多篇技术文章并参与过多本技术图书的翻译，现关注于.NET企业应用与敏捷开发。可以通过xuning.net@gmail.com与他联系。

李强，软件工程师，2008年毕业于西北大学软件学院，现就职于道富银行技术中心。开发了多个大型金融项目。爱好技术，喜欢折腾，在WCF、WPF、ASP.NET MVC等多个领域都有较深入的研究。可以通过Email/MSN: Sparkli@hotmail.com与他联系。

目录

第一部分

JavaScript 动画基础



第 1 章　动画的基本概念

本章涵盖以下内容：

- 动画
- 帧与运动
- 动态与静态动画

看看 Web 浏览器已经发展到何种程度！最初 Web 浏览器仅仅是一个用于在网络上访问文本文件的程序，很快它就彻底改变了我们沟通与分享信息的方式，现在它已经演变成一个完全图形化的交互式编程环境。HTML5 作为网页标记语言的最新标准加入了大量原本只存在于本地应用中的图形功能。经过短暂的停滞，得益于 HTML5 和 JavaScript 技术引发的新一轮的竞争与创新，现在的 Web 浏览器发展迅速。新的 canvas 元素提供了一种创建标准化的游戏、应用与动画的方式，使用这种方式创造出来的产物可以正常地运行在最新的 Web 浏览器以及移动设备上，包括各种流行的移动电话与平板电脑，例如，iPhone、iPad 与安卓设备。

本书将介绍借助 JavaScript 在 HTML5 canvas 元素中实现动画所需的各种编程、数学与物理技术。你将会看到开发人员在这些技术的支持下，首次在遵循标准的 Web 浏览器上获得了更强的开发能力、控制能力以及优良的交互性。

在我们深入使用 JavaScript 实现动画的具体技术和公式之前，让我们首先了解什么是动画，其背后的一些基本原理以及一些可以使动画变得更加动态、更加有趣的概念。

无论你是否第一次接触电脑动画还是你之前有使用 Adode Flash 实现动画的经历，本书都会对实现编程动画起到很大的指导作用。本书在从 Flash 版本移植到 JavaScript 版本的过程中经历了大

量的修改，不过它的成功也验证了底层的技术和数学概念是与语言无关的。本书的动画虽然是针对Web浏览器实现的，不过在适度的图像功能支持下，书中的公式与示例可以应用于任何开发环境。

如果你等不及开始编码，你完全可以跳过本章的内容。不过，强烈建议你在某一时刻翻回本章。通过下面的介绍，你应该会对动画的工作原理产生一些基本的了解。

1.1 动画

动画是运动。运动是一个物体随着时间在空间中改变它的位置，前一分钟它在这里，下一分钟它到了那里。将数学公式应用于改变一个物体的位置后，你就可以决定它的下一个位置并影响运动的行为，也就赋予了物体生命。

但是动画并不仅仅就是运动，它还包括对任意可视属性的变化，如形状、大小、方向与颜色等。一些早期的电脑动画通过循环变换颜色模拟运动。例如，以某种频率改变大量形状不一的蓝色像素的色调就可以模拟水的流动，从而创造出一个瀑布，而在此过程中没有一个物体的位置发生了变化。

时间是动画的基本组成部分。它是用于表现对象位置变化的途径。没有时间就没有运动，动画也就变成了静止的图像。同理，没有运动，我们也无法感知时间的存在，即使是现在正在发生的事情，例如，一段来自空停车场的监控视频。没有物体的运动，你无法确认你是在观看一个实时的视频，或是5秒钟前的一个画面，亦或是一幅没有变化的静止图像。只有当一个塑料袋被风吹过屏幕，你才能确定时间的存在以及可能发生进一步的变化。没有时间，图像中就不会发生任何事情。

这引入了另一个话题，动画可以让人保持注意力。如果有变化，人们的大脑自然会感到好奇。什么变化了？为什么它会变化？我让它变得吗？这种变化符合我对这个物体的心理预期吗或者我应该调整我的假设？现在的媒体（例如，音乐和电影）之所以能够引人注目是因为，在现实生活中，我们无法确认下一步会发生什么。我们可能有一个大概的想法，而在此基础上的所作的模式推理则会让人感到有趣，而我们之所以能够从中体会到快乐恰恰是因为我们游走在不确定的边缘。传统的媒体则是一成不变的，如相片、绘画和文本，我们会探索它们的细节，我们对它们的理解和解读可能随着时间而变化，但是它们本身却永远不变。这就是动画引人注目的原因。媒体总是包含了变化，这些变化把握住了我们天生就习惯于的一些常识。因此，人们很容易就在一部电影中沉浸了几个小时或者被一款视频游戏迷住几天。如果有什么事情要发生，人们总是想要知道那会是什么。

1.2 帧与运动

动画是一个创造运动假象的过程。几乎所有的投影运动媒体都采用帧实现运动。

帧是将一系列离散的图像以极快的速度连续播放从而模拟物体运动或变化。帧是你能在电脑、电视或电影屏幕上看到任何东西的基础。这个想法要回溯最早期的卡通，当时动画家把很多独立的图片绘制到一张张赛璐珞片（又称明片）上，而早期的运动影片同样是针对多张照片

采用了类似的技术。

利用帧实现动画的概念很简单：当我们连续播放一系列有略微差别的图像时，大脑会将它们想象成一幅不断运动的图像。但是为什么我们坚持将其称为一种运动的假象？如果你在电影屏幕上看到一个人走过一间房间，这不算是运动吗？当然不算，因为这只不过一个人的图像而不是真人，但是这并不是我们不将其视为运动的原因。

注意，运动的物体通过穿越两点间的空间实现从一个点移动到另一个点。穿越空间的过程是流畅的，而不是跳跃的，这才是真实的运动。而基于帧的运动恰恰是跳跃式的。它的移动不是从一点到另一个点，而是在每一帧中从一个位置消失并出现在另一个位置。移动得越快，跳跃的感觉就越明显。

如果你看到一张图片里有一个人在房间的左边，而几秒钟后同一个人又出现在房间的右边，那么你会认为这是两张图片而绝不会把它们当做动画。即便你看到 6 张图片连在一起表现出一个人穿过房间的过程，你也只会把它们当做一系列独立的照片（见图 1-1）。如果有足够多的照片以足够快的速度播放，虽然还是改变不了它们只是一系列静态照片的事实，但是你的看法却会发生变化。你的脑子会把它们当做一个人穿过了房间。与最开始的两张照片相比，它并没有什么本质不同，它始终不是真实的运动，不过，在到达某个频率之后，我们的大脑被欺骗了，认可了这一运动的假象。

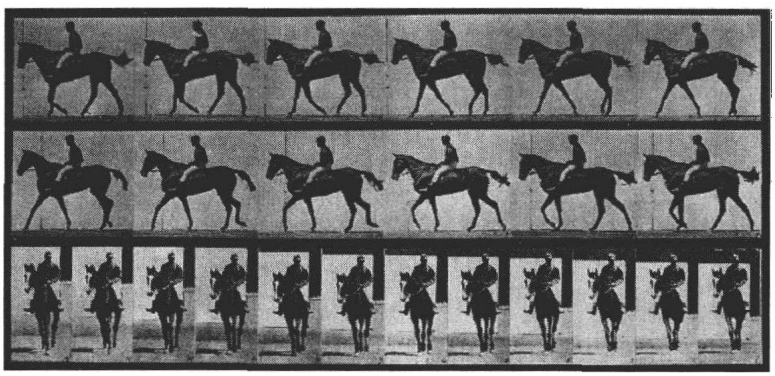


图 1-1 摄影师埃德沃德·迈布里奇拍摄的一系列静态照片

电影产业对这个频率做了大量的实验。研究表明将频率保持在每秒 24 帧，人们就会把这些帧视为一张运动的图像。如果低于此频率，人们就会察觉到跳帧，也就打破了这种运动的假象。并且人类的眼睛似乎无法区分比这更高的频率，每秒播放 100 帧并不会让你的动画显得更真实，尽管更高的帧率在电脑动画中会带来更好的互动感和流畅度。

1.2.1 记录帧

帧的完整概念使得以下 3 件事得以实现：储存、传播以及播放。你无法储存、传播以及播

放一个真实的人走过一个房间，但是你可以保存那个人走过房间的很多照片，你可以传播这些图像并播放它们。这样，只要你能够解读这些保存的图像并有办法播放它们，你就可以在任何地方以及任意时间展现这一动画。

现在，我们需要给帧一个更加通用的定义。目前为止，我们一直将帧视为一张静止的照片或图画。让我们将之称为对一个系统在某一特定时间点的记录。这个系统可以是一个人穿越房间，时间可以是当他走到房间中间的时候，而此时的记录就是他走到房间中间的那张照片。另一方面，系统也可以是一个虚拟对象集合，此时的记录就是它们在某一特定时刻的形状、颜色以及位置。这样，你的动画就不再是一系列静止的图像，而是一系列关于图像的描述信息。电脑会根据这些描述信息创建并显示图像，而不是直接播放这些图像。将这一思想进一步演化就衍生出了程序帧。

1.2.2 程序帧

由于你有一台可以任由你使用并能够根据需要完成计算的电脑，因此你并不需要通过一个很长的帧描述列表实现动画。你可以将之简化为起始帧的一个描述，而后遵循某些特定的规则构建后续帧。此时，电脑所要做的就不仅仅是根据图像描述创建出一幅图像，而是要先创建图像描述，再根据描述创建图像并最终显示图像。

考虑一下采用这一方式你可以节省多少硬盘空间。图像需要占用硬盘的磁盘空间以及带宽，每秒 24 幅图像会迅速占用大量的空间。如果你将其简化为一份描述与一系列规则，那么文件大小则会急剧缩减。即便一系列用于表现物体运动与交互的复杂规则所占的空间也不会超过一幅中等大小的图像。事实上，人们对程序动画的第一印象就是文件是多么小。

自然，这是有代价的。随着系统变得越来越大，规则变得越来越复杂，电脑必须能够迅速地计算出下一份图像描述，并将其显示出来。如果你想保持一定的帧率，那么给予电脑的处理时间（毫秒级）就极其有限。如果电脑无法及时完成计算，帧率就会下降。而另一方面，基于图像播放的动画就无须关心场景中的内容以及它有多复杂，它只需要不断显示下一幅图像，而这通常是准时的。

1.3 动态动画与静态动画

使用程序帧实现动画的一个大的优势在于它可以变成动态的，因为图像是在运行时才创建出来的。动态地创建新的图像则可以带给每一次观看独一无二的视觉体验，而不是观看一组事先排好顺序的帧，举个例子，无论你将一部电影翻来覆去看多少遍，它的结局都不会改变。如果能够根据用户提供的值计算物体的位置，例如，鼠标光标的坐标，那么动画就可以根据用户的操作做出变化从而实现与用户的交互，营造出别的媒体类型所不能达到的一种深入其境的氛围。

不过编码的动画并不一定就是交互式的。可以使用代码让一个物体从某个点穿过屏幕。每

次播放动画的时候，会运行同样的代码，导致同样的运动重复发生。这就是静态动画的一个例子。每一帧，从开始到结束都是预先定义好的。跟电影相似，你观看的是一组事先排好顺序的帧，而且每一次观看的内容是不变的。

但是如果同样使用代码创建一个物体，却将其摆放在一个随机的位置，并让它以一个随机的速度和方向运动呢？此时，每次播放动画都会有不同的事情发生。再想象一下，如果在动画开始的时候，你根据当前的日期和时间创建不同的场景，比如一个冬日的早晨，一个夏日的午后，即根据动画播放的日期展现截然不同的图像又会如何？

再比如，你在动画运行的时候可以通过键盘和鼠标设定动画中的一些参数。这使得用户可以与屏幕上的物体产生交互，远远脱离静态动画。

或许动态动画中最有意思的方面，以及本书的焦点在于如何将现实世界中的数学与物理原理运用到动画中的物体上。可以让一个物体往任意方向移动，还可以给它施加一些重力，这样当它移动时，它还会不断下落。当它碰到地面的时候，它会弹起，并且弹起的高度不会超过它的起始高度。最终它落到地面并停在那里。你还可以加入一些用户交互，让用户能够用鼠标捡起该物体并用键盘移动它。当用户不断抛投该物体时，他会感到他在操作一个真实的物体。

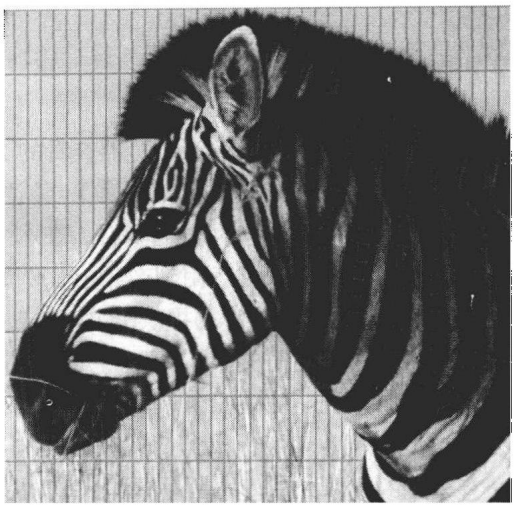
在这种动画下，用户不再是被动接受一串顺序帧的观众，而是进入了你创造的场景。你可以建模一个遵循你自己的物理规则的世界，提供更加真实的体验，你也可以完全摆脱现实世界的约束。作为一名程序员，你可以自由发挥你的想象，只要你觉得合适。这正是创造性的编码的乐趣所在。通过把工作交给电脑使其不断更新显示的内容，你能够创建一个非常丰富的场景使观众沉浸其中，而这是人类历史上之前的种种媒体都做不到的。观众会为此停留多久？他们会一直沉浸其中，只要你始终让他们感到有兴趣。他们与之交互的越多就越容易再次回来。

1.4 小结

这个起始章节概述了动画的基本原理。在帧和动态的概念的基础上，可以在动画中创造出运动和互动的感觉。

下面的章节将研究运动所涉及的数学原理并为之构建一系列工具函数，可以将它们用于动画程序以构建各种运动，当然还会介绍如何使用这些函数。可以使用这些工具函数创建任何东西，这完全是你自己的决定。本书介绍的技术最常用的一个场景则是游戏开发。游戏本质上是具备交互性的动画，其中包含玩家需要达到的一些目标。不过，这不仅仅是一本关于游戏编程的书。书中的技术适用于各种不同的动画项目，包括导航系统、广告、教学应用乃至互动艺术。

归功于创新驱动的现代浏览器，Web 编程的一个新纪元创造性地拉开序幕。HTML5 中的 canvas 元素为我们带来了一个遵循标准。跨平台的组件，使用它可以创建更加高级的 Web 图像。本书将探究那些以编程方式创建动画的原理，而这些动画则是构成下一代图形交互的重要元素。



第 2 章　动画的 JavaScript 基础

本章涵盖以下内容：

- 动画基础
- HTML5 简介
- 用代码实现动画
- JavaScript 对象
- 用户交互

如果说第 1 章是在原理上对动画的概述，那么本章将从技术角度概述如何创建 HTML5 文件以及如何用 canvas 元素和 JavaScript 实现动画。本章将介绍 HTML5 文件结构的要点、动画循环、JavaScript 对象以及用户交互的相关内容。本章所学的技巧将贯穿本书的始终。

2.1 动画基础

在开始本章的介绍之前，先回顾下第 1 章的内容：

- 动画由帧组成，每一帧在表现运动的假象上有细微差别；
- 逐帧动画包含每一帧的图像或图像描述；
- 动态动画包含一幅图片的起始描述以及后续每一帧图像的变化规则。

本书着重于动态动画的规则，其中会介绍改变图像描述的各种不同技术，正是依赖于这些技术才得以实现逼真的动画效果。本章将介绍如何定义图像起始描述的结构，如何为每一帧应用变化规则以及如何将两者结合在一起完成一个程序。在此过程中我们会创建大量可行的实例。

2.2 HTML5 简介

本书将创建可通过 Web 浏览器查看的 HTML5 文件。HTML5 是最新一代 HTML（Hyper Text Markup Language 超文本标记语言），该语言用于将内容结构化为有组织的层次并展现在万维网上。HTML 元素是创建网页的基石。在本书编撰期间，HTML5 规范仍在制定阶段，不过许多功能已趋于稳定并可随时运用到产品上，例如，canvas 元素。HTML5 加入了一些新元素并为多媒体内容（例如，音频与视频）提供了更好的支持，从而进一步完善了在 1997 被制定为标准的 HTML4。由于这些新的并在语义上有意义的元素暴露了一系列可在 HTML 文件中访问的属性以及控件，因此可以通过 JavaScript 用编程方式操作它们，这赋予了我们创建并控制媒体的有效手段。

HTML5 是一系列独立功能的集合。当谈及一个特定浏览器是否支持 HTML5 时，这不是一个全有全无的问题，合理的做法是测试该浏览器是否支持规范中定义的某个特定功能。不同的浏览器可能对不同的功能有不同程度的支持。不过这使得开发者很难决定是否采用某个 HTML5 元素，因为他猜不到用户会使用何种浏览器查看文件。随着各个浏览器的不断改进，跨越各个平台的 HTML5 功能检测变得越来越容易，目前开发者应该总是检测用户的 Web 浏览器是否支持某个 HTML5 功能并为之提供备用方案，即便备用方案只是一句委婉地提示用户升级浏览器的消息。

2.2.1 对 canvas 的支持

一个至少对本书而言有利的好消息是所有主流的浏览器厂商都提供了对 canvas 元素的支持。这意味着，只要用户将他的浏览器升级到了最新的版本，他就可以看到我们创建的动画。游戏和动画对推动用户升级他们的浏览器是一种有利的刺激，因为大多数的人在经历过近十年的视频游戏的发展后很容易理解顶尖的图像效果需要最新的软硬件支持。至少说服用户升级浏览器比让他买一个全新的游戏主机要容易多了。

万一某个 Web 浏览器不支持 canvas 元素，在 HTML 文件中，开发者仍可以在 canvas 标签中提供如下备用内容：

```
<canvas width="400" height="400">
 <p>This browser does not support the<code>canvas</code> element.</p>
</canvas>
```

该警告消息仅在浏览器无法识别 canvas 标签时才会显示。如果浏览器支持该标签，它会呈现 canvas 元素并忽略其中内嵌的<p>元素。

还可以在 HTML 文件中加入以下 JavaScript 代码，以编程方式测试浏览器是否支持 canvas 元素：

```
if (document.createElement('canvas').getContext) {
  console.log("Success! The canvas element is supported.");
}
```

以上代码创建了一个新的 canvas 元素并测试 getContext 属性是否存在，而我们知道一个有效的 HTML5 canvas 元素是包含该属性的。因此，如果浏览器确实支持 canvas 元素，你将会在

调试控制台上看到成功的消息。

表 2-1 列出了最流行的 Web 浏览器以及它们开始支持 canvas 元素的最小版本号。

表 2-1 主流浏览器对 canvas 元素的支持

IE	Firefox	Safari	Chrome	Opera	iOS Safari	Android Browser
9	3.5	3.2	9	10.6	3.2	2.1

2.2.2 性能

编程图形一直并且在可预见的将来也是一项计算量很大的操作。原因很简单，你能做的越多，你想要做的也越多，同时对系统的性能要求也越高。视频游戏过去 25 年的历史是一段令人惊叹的技术发展之旅，从游乐场游戏机中的块状角色发展到了今天运行在游戏主机上的让人身临其境的 3D 世界。不过，我们还是想做得更好。我们时常拿计算机动画与现实世界中的一些特效做对比：例如，人物的真实感、光影特效与物理效果。令人惊叹的是，这些模拟竟经得起人们的审视，而且不止一两个例子如此。我们目前还处在计算机动画的起步阶段，随着计算机不断地变快（得益于摩尔定律），开发人员不断改善技术，人类在视觉创造上的能力将永无止境。

不过使用 canvas 元素在 Web 上绘制动画还处在它的孵化期，现在这只是被视为使用诸如 Adobe Flash 这样的浏览器插件实现动画的一种替代方式。最近几年，开发者在 Web 浏览器以及 JavaScript 引擎中的速度和性能上取得了突破性的发展，鉴于此领域竞争异常激烈，我们可以预见到将来会有更多的优化和提升。

本书中编写的示例可以在近期的电脑和 Web 浏览器上获得流畅以及合理的性能表现。不过读者的电脑配置可能与作者有差异，所以当你在实验本书中的源代码时，尽可以调整某些参数的设定以获得更加流畅的体验。而且尝试不同的参数值并观察结果的变化也是一种学习公式工作原理的更好途径。

不过，在你对外分享任何动画前，请在尽可能多的设备上加以测试。因为越来越多的人开始使用移动电话与平板电脑而不是传统的台式机来访问 Web，开发者需要兼顾到各种不同设备上的性能差异。针对各种不同平台的测试和度量是确保代码能够表现良好的唯一手段。

2.2.3 HTML5 基本文档

Web 开发的一个非常大的优势在于创建和查看文件非常容易，你所需的仅仅是一个文本编辑器和 Web 浏览器。下面这个简单的代码片段提供本书中所有示例的初始设置。以下是开发者将使用的一份基本的 HTML5 文件，在介绍完这些元素的结构后，为清晰起见，我们会做一些小的补充：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
```

```
 <title></title>
 <link rel="stylesheet" href="style.css">
</head>
<body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <script>
 window.onload = function () {
  //Our code here...
 };
 </script>
</body>
</html>
```

将该文件另存为01-skeleton.html并在Web浏览器中打开它。因为这是一份空白的文件，所以在浏览器中你不会看到任何内容，不过该页面确实已载入并且它还是一份完全有效的HTML5文件（可以通过在浏览器中查看源代码确认网页中包含特定内容）。

现在逐步介绍文件中所涉及的每个元素。第一行只是简单地声明了本文件类型为HTML5。如果你接触过HTML4的各种文件类型的话，你会发现HTML5的声明是多么简单：

```
<!doctype html>
```

接下来，声明作为根节点的HTML元素以及header元素：

```
<html>
 <head>
  <meta charset="utf-8">
  <title></title>
  <link rel="stylesheet" href="style.css">
 </head>
```

在head元素中，首先将文件的字符编码方式设置为utf-8。UTF-8是Unicode的一种编码方式，它是一种通用字符集，其中定义了世界上大多数语言所用到的字符。浏览器使用该编码方式读取文件中的字符并将它们以格式化的文本显示出来。这些文件以一系列的字节存储在某个服务器的文件中，它们通过网络传输，然后在用户的电脑上重新组装并显示在Web浏览器中。字符编码方式告诉浏览器如何将一系列字节转换为一串字符，这些字符经过处理并显示为一张网页。如果不在文件中声明编码格式，浏览器将试图（错误地）猜测文件的编码格式，或（错误地）使用一个默认的设置，而这可能导致页面显示出错。开发者最好在文件中显式指定字符编码格式以避免潜在的困惑。

所有有效的HTML5文件都包含一个也内嵌在header中的title元素，由于用到了CSS样式表，因此在实例中创建了一个指向外部文件的link元素。外部文件会包含文件中所用的样式定义，稍后再对style.css文件做进一步介绍。

在设置好header元素后，让我们来看看文件的剩余部分：

```
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
   //Our code here...
  };
  </script>
 </body>
</html>
```

在 body 元素中放置了一个 canvas 元素。这是我们画动画以及脚本中将引用的地方。为了让我们可以在浏览器中看到它，这里为 canvas 设定了 height 与 width，同时通过指定 id，我们可以通过 DOM 接口访问该元素。

在 canvas 元素后，加入了一个 script 标签，其中将包含每个示例的 JavaScript 代码。为了确保不要在加载文件中其他元素前执行脚本，我们将 script 标签放在其他元素之后，末尾的 body 标签之前。这样做还有一个好处在于当脚本来自另一个文件，甚至有可能是另一个服务器上的文件时，可以避免等待下载该文件时无法并行加载文件的其他内容。这有助于加快加载速度和文件响应速度。

框架脚本非常简单并且实际上没做什么事情。window 对象是 Document Object Model（文件对象模型）中的顶层对象，通过它可以访问 DOM。当文件加载完毕后，window 对象会执行与之 onload 属性关联的函数。

```
<script>
window.onload = function () {
  //Our code here...
};
</script>
```

本书中的示例代码将放置在 window.onload 的回调函数中。由于该方法在文件中的所有元素加载完成后才会执行，因此可以确保 canvas 元素在调用代码的时候可以正确访问。不过随之而来了的问题是如果文件中包含大量的内嵌数据，比如大图片，window.onload 方法就不得不等待相当长得时间才能被执行到。此时，最好使用 JavaScript 加载那些耗时的资源，在第四章将会介绍如何做到这一点。

最终，随着我们关闭 script、body 以及 html 标签，一个基本但是完全有效的 HTML5 文件创建完毕。

2.2.4 CSS 样式表

在文件的 header 元素中，创建了一个 link 指向 CSS 样式表。现在，让我们看看那个样式表文件。本书用到的样式定义刻意简化了，只声明了文件中 body 与 canvas 元素的背景色。canvas 默认的背景色是透明的，这个颜色可能是你想要的，不过为了让你看清楚 canvas 元素在文件中的确切位置，把背景色改成了白色。下面是 style.css 文件的样式表定义：

```
body {
  background-color: #bbb;
}
#canvas {
  background-color: #fff;
}
```

这里假设文件中包含一个 id 为 canvas 的元素。样式表的复杂程度将随着文件的复杂程度而变化。HTML 文件定义了文件的结构，CSS 样式表则定义了元素的样式或外观。一般来说，最好将文件的内容、样式和脚本分布在不同的文件中。

2.2.5 额外的脚本

当示例变得更加复杂时，可能需要重用部分代码，此时为了更加清晰，将代码分离到不同的文件中则成为一种方便的做法。当新声明的类将用于多个练习时，当函数的繁杂实现会分散我们对当前话题的注意力时，我们会把它们放入另一个文件中。

在本书中，我们会将那些工具函数放在一个名为 utils.js 的文件加以维护。该脚本包含了为示例创建样板代码的函数，从而让我们更关注于本质的动画原理。因为我们会在引入每个函数的时候加以解释，所以这个脚本文件对你来说不会是一个黑盒子。

在该文件中，许多工具函数会作为属性添加到一个名为 utils 的 JavaScript 全局对象中。这样做可以避免在全局名称空间中堆满许多函数。为此，确保在 utils.js 文件中最开始的地方像下面这样声明一个空对象：

```
var utils = {};
```

为了把该文件或其他脚本导入文件中，需要创建一个 script 元素并设置它的 src 属性为脚本文件的地址。为了确保每个函数在试图使用前加载，请在示例代码前引入该元素：

```
<script src="utils.js"></script>
<script>
window.onload = function () {
 //Our code here...
};
</script>
```

2.2.6 调试

编写代码的一个最重要的方面就是调试代码。在 Web 开发的早期，调试代码意味不断弹出警告窗口。值得庆幸的是，Web 浏览器现在提供了越来越高级的调试工具用于代码审查和性能分析。因为这些工具使得你可以单步调试一个正在运行的程序并与之交互，所以你可以确切地了解代码在某一特定时刻在做什么。

可以肯定地说，每一个支持 HTML5 的浏览器都有内置的开发工具以及一个用于输入 JavaScript 语句并输出结果的控制台。你可能需要仔细查看应用程序的菜单，不过这些功能肯定存在于某个地方。

例如，在 Chromium Web 浏览器中，单击右上角的扳手图标，下拉至工具菜单项再单击 JavaScript 控制台。日志消息将输出在此窗口中。图 2-1 显示了一个在 Chromium 浏览器中运行的调试会话。Firefox Web 浏览器也有类似的功能：在“文件”菜单中，单击“工具”菜单，再选择“Web 控制台”。IE9 和 Opera 都有各自的开发环境。你最好能够轻松地使用这些主流浏览器中的工具，因为为了兼容各个浏览器，你需要在所有这些浏览器中调试代码。如果你无法找到这个浏览器的 Web 开发工具，请务必查看它们的帮助文件。

当打开一个 Web 开发者控制台后，可以直接在浏览器中输入 JavaScript 表达式并获得计算

结果，试着输入以下代码：

```
console.log("Hello, World!");
2 + 2
```

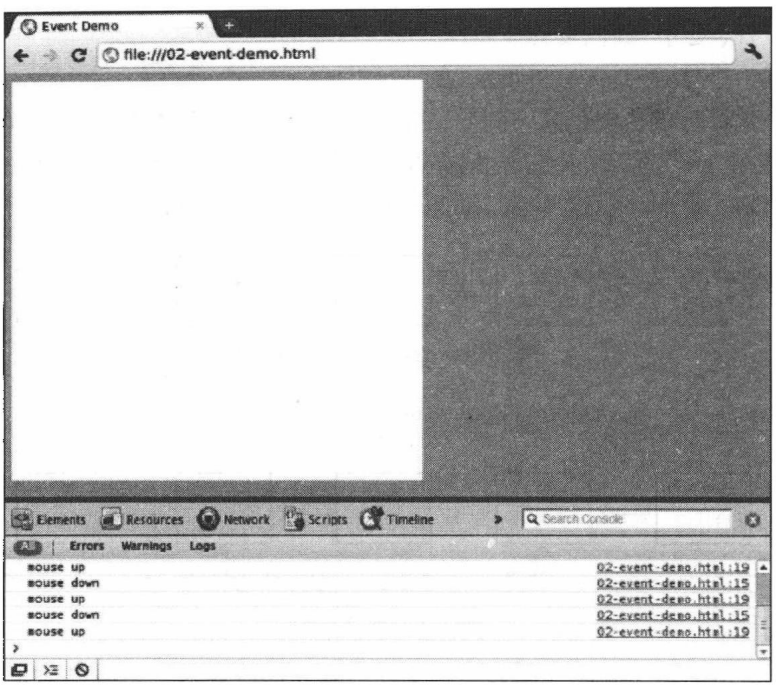


图 2-1 Chromium 浏览器的调试控制台

在控制台中，还可以访问 DOM 元素以及脚本变量来查看它们的值（确保它们处在恰当的作用域中），这样可以方便地推算出程序运行的情况。通过这种方式，可以在将小段代码合并到一个大程序前更好地测试与调试它们，从而尽早地发现 bug!

2.3 用代码实现动画

在准备好 HTML5 文件的基本结构之后，我们已经了解了足够多的基础知识，可以开始编码了。我们需要一个文本编辑器用来输入示例代码以及一个支持 HTML5 的 Web 浏览器运行这些示例。同时我们需要熟悉该浏览器内置的开发者控制台。在准备好这些工具后（它们很可能已经在你的电脑上）我们就可以出发了，让我们深入学习一些动画吧。

2.3.1 动画循环

几乎所有的程序动画都会表现为某种形式的循环。我们会创建一个展现一系列图像的流程图以实现逐帧动画，其中每一帧只需要绘制出来即可，如图 2-2 所示。

当你开始绘制图像时，尽管每幅图有细小的差别，JavaScript 代码也不会为每一帧创建并保存一幅新的图像，即便是逐帧动画。我们会为每一帧图像保存其绘制到 canvas 上的每一个对象的位置、尺寸、颜色及其他属性。因此如果要实现一个球滚过屏幕的动画，我们会在每一帧中保存球的位置信息。例如，第 1 帧标明球处在距离左边缘第 10 个像素的位置，第 2 帧则标明在第 15 个像素的位置，诸如此类。代码会读取这些相关的数据，并根据这些数据将对象绘制出来，从而显示这些帧。据此过程可以衍生出如下流程图，如图 2-3 所示。

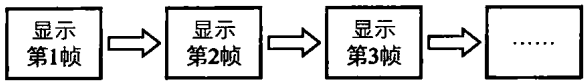


图 2-2 逐帧动画

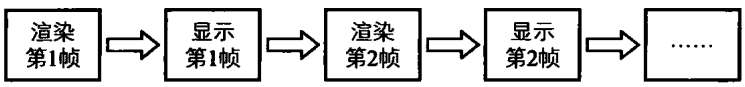


图 2-3 渲染并显示每一帧

而描述一个动态的，编码的动画的流程图则如图 2-4 所示。

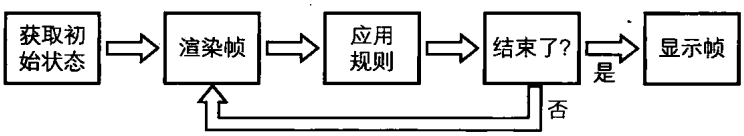


图 2-4 脚本动画

从图 2-4 中可以看出，其中并不存在第 1 帧，第 2 帧等概念。程序动画通常并总是可以通过一帧就完成动画。由此可以看出我们所指的循环的含义。

首先，设置初始状态，比如，用 canvas 内置的绘图 API 向屏幕上绘制一个圆形，渲染并显示这一帧。然后应用规则，规则可以简单如“球向右移动 5 个像素”，当然也可能由几十条遵循复杂三角函数的线构成。本书中的示例包含从简单到复杂的各种情况。应用规则后会变迁到新的状态，此时会根据一个新的图像描述渲染并显示。随后同样的规则会重复应用。

同一套规则会不断重复地应用，而不会出现为第一帧关联一套规则而又为第二帧关联另一套规则的情况。所以这里的挑战在于如何拿出一套规则，使之能够处理场景中可能出现的各种情况。当球滚出 canvas 的右边界怎么办？这套规则需要考虑到这种情况。你是否希望用户可以使用鼠标与球产生互动？这套规则也需要将其处理好。

这听上去有些令人气馁，不过实际上并没想象中那么复杂。可以通过创建一两个规则实现一些简单的行为，以此为基础再不断添加新的规则。这里所指的规则实际上就是程序语句。每条规则可以由一条或多条语句构成。以球向右移动 5 个像素为例，该规则在 JavaScript 中可以通过以下语句表达：

```
ball.x = ball.x + 5;
```

以上语句表示在当前球体所处的 x 轴（水平轴）上右移 5 个像素，以此作为球体在 x 轴上的新位置。甚至可以将语句简化成以下这样：

```
ball.x += 5;
```

+=操作符会将右侧的值与左侧的变量相加，并把结果赋给左边的变量。

下面是一套更加高级的规则，它会出现在本书的后续章节中：

```
var dx = mouse.x - ball.x,
    dy = mouse.y - ball.y,
     ax = dx * spring,
     ay = dy * spring;
vx += ax;
vy += ay;
vy += gravity;
vx *= friction;
vy *= friction;

ball.x += vx;
ball.y += vy;
ball.draw(context);
```

现在我们不用担心以上语句的含义，只需知道这些代码会重复执行，从而不断产生新的帧。

那么，这些循环如何运行呢？下面的解释是许多入门的程序员都会有的一种误解：循环的运行依赖于几乎存在于所有编程语言中的 while 循环结构。通过以下代码设置一个无限循环用于更新球体的位置：

```
while (true) {
 ball.x += 1;
}
```

这段代码看上去很简单：因为 ture 值永远为真，while 子句的条件判断永远成立，所以循环不断执行。在循环中球体的 x 轴坐标每次增加 1 个像素，从 0 到 1、2、3、4 等。球体在 canvas 上从左向右不断地移动。

如果你也犯过类似的错误，你会知道通过以上代码你并不会看到球体穿越 canvas 的动画，实际上你压根看不到球体，因为它已经越过了屏幕的右边界。为什么球体没有移动到循环中的各个位置？实际上，它已经移动到那些位置。你之所以看不到它是因为我们没有更新 canvas 元素的显示。图 2-5 的流程图展现了本质上发生的事件：

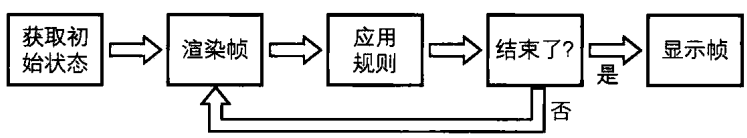


图 2-5 为什么无法通过 while 循环实现动画

应用规则并把球体移动到新的位置，新的图像得以创建，不过我们始终没有机会将其显示出来，因为在帧结束的时候并没有将对象绘制到 canvas 上。而这才是重点。

为了实现动画，需要为每一帧执行以下操作：

（1）执行该帧所要调用到的代码；

（2）将所有对象绘制到 canvas 上；

（3）重复这一过程渲染下一帧。

将这些步骤记在脑子里，为此创建一个函数用于不断更新对象的位置并将它绘制到 canvas 元素上。然后创建一个 JavaScript 定时器启动循环：

```
function drawFrame () {
 ball.x += 1;
 ball.draw(context);
}

window.setInterval(drawFrame, 1000/60);
```

以上代码定义 drawFrame 函数用于更新球体的位置并使用 draw 方法（尚未创建）将其绘制到 canvas 上。然后将 drawFrame 作为参数传递给 window.setInterval 方法，该方法会根据第二个参数指定的间隔时间以毫秒为单位重复执行 drawFrame 函数。在本例中，间隔时间为 1000/60，即一秒 60 帧，差不多 17ms 一帧。

在相当长时间内，开发者通过以上方式使用 JavaScript 建立一个动画循环。如果你坚持，你仍旧可以在本书的所有示例代码中运用此方式。不过问题在于，JavaScript 定时器并不是为实现动画设计的。因为它无法达到毫秒级的精确度（每个浏览器的定时器分辨率不同），所以无法依赖它实现高质量的动画。除此之外，通过第二个参数指定的间隔时间仅仅是请求在那一时刻执行而已。如果同一时刻在浏览器的任务队列中有其他任务的话，动画代码的执行任务将不得不等待在那里。

由于动画并不是之前 HTML 规范所包含的功能，因此浏览器厂商并没有将此种优化放在一个优先级很高的位置上。不过，随着 HTML5 中 canvas 元素的引入以及对多媒体内容的关注，不同的浏览器之间再次在性能和速度上展开了激烈竞争。由于认识到动画已成为 Web 应用中一个日益重要的组成部分，浏览器厂商开始不断提出新的解决方案以应对这一需求。

2.3.2 使用 requestAnimationFrame 的动画循环

由于开发者对基于 HTML5 的动画兴趣日增，因此 Web 浏览器为此专门为 JavaScript 的开发者实现了一个 API，通过它提供了基于浏览器的优化实现。window.requestAnimationFrame 函数接收一个回调函数作为参数，并确保在重绘屏幕前执行该回调函数。在某些浏览器中，还支持一个可选的第二参数，可以通过它指定的一个 HTML 元素提供动画的可视区域。在回调函数中对程序的修改必定发生在下一个浏览器重绘事件之前。可以通过对 requestAnimationFrame 函数的链式调用实现动画循环：

```
(function drawFrame () {
 window.requestAnimationFrame(drawFrame, canvas);

 //animation code...
}());
```

这看上去只是一小段代码，但是了解其工作方式非常重要，因为这是实现动画循环的核心思想，它贯穿了本书的所有示例。这里定义了 drawFrame 函数，其中包含了每一帧将要执行的动画代码。该函数的第一行代码调用了 window.requestAnimationFrame 函数并将 draw Frame 函数自身的引用作为参数值传入。第二个可选参数是要绘制的 canvas。你可能会觉得惊讶，我们

居然可以在完成一个函数的定义之前就把它作为一个参数值传入另一个函数。切记，在函数运行到需要将它作为参数值传入时，它早已定义好了。

当执行 drawFrame 函数时，window.requestAnimationFrame 将 drawFrame 函数放入队列等待在下一个动画间隔中再次执行，而当它再次执行时又会重复这一过程。由于不断地请求执行该函数，因此就串联成了一个循环。所以，该函数中定义的代码会不断地调用，使得我们可以在 canvas 上以细微的间隔时间绘制动画。

为了启动循环，在定义好 drawFrame 函数后，就用一个圆括号将其包起来并立即调用它。这是一种更节省空间，也更加清晰（这里有争议）的做法，另一种传统的方式是首先定义函数，然后立即在下一行代码中调用它。

由于 requestAnimationFrame 是一个相对新的功能，因此目前的浏览器还致力于各自的实现。如果你希望代码具备更好的跨平台性，下面这一小段代码就可以用来规范该函数在不同浏览器中的实现。

```
if (!window.requestAnimationFrame) {
 window.requestAnimationFrame = (window.webkitRequestAnimationFrame ||
                                 window.mozRequestAnimationFrame ||
                                 window.oRequestAnimationFrame ||
                                 window.msRequestAnimationFrame ||
                                 function (callback) {
                                   return window.setTimeout(callback, 1000/60);
                                 });
}
```

这段代码先检查了 window.requestAnimationFrame 函数的定义是否存在，如果不存在，就遍历已知的各种浏览器实现并替代该函数。如果它还是找不到一个与浏览器相关的实现，它最终会采用基于 JavaScript 定时器的动画以每秒 60 帧的间隔调用 setTimeout 函数。

由于以上针对浏览器的环境检查会被所有示例用到，因此把这个函数放入了 utils.js 文件中以导入 HTML5 文件中。如此，就可以确保动画循环可以工作在多个浏览器上，而通过这种方式也使得脚本更加简洁，从而可以把注意力放在理解每个示例的核心思想上。

2.4 JavaScript 对象

本书的重心在于使用 JavaScript 创建动画所需的各种原理和公式而不是具体的编程技巧。因此，本书中不会创建庞大的框架库或者复杂的数据结构，而是尽可能保持代码的简洁。

你可以将在本书中学到的动画概念纳入更加高级的 JavaScript 项目中，不过，本书的目的不是交给你一份预先构建好的代码供你复制和粘贴，而是试图让你理解每一种动画的工作原理。

由于本书采用 JavaScript 编写示例，因此你需要了解该语言的一些重要概念以便于理解示例代码。JavaScript 中最重要的东西就是对象和函数（一种特殊的对象），让我们从它们入手吧。

2.4.1 基础对象

JavaScript 被设计成一个简单的面向对象系统。对象是一个包含若干属性的数据结构。这些属性可以是变量、函数或其他对象。当给一个函数赋给一个属性时，它称为一个对象的方法。浏览器中预定义了很多对象，不过开发者也可以创建自定义的对象。例如，可以像下面这样创建一个空对象并将它赋给一个变量供将来引用：

```
var objA = {};
```

这里创建了一个不具备任何属性的对象并将其保存到变量 objA 中。由于 JavaScript 对象默认情况下可以在任何时间修改，因此可以像下面这样为它添加一个新的属性：

```
objA.name = "My Object A";
```

以上代码在 objA 对象中创建了一个名为 name 的新属性并把一个字符串值"My Object A"赋予它。总是可以通过 objA.name 的方式访问对象的属性值。也可以如下在声明一个新对象的时候就为之创建属性：

```
var objB = {
 name: "My Object B",
 hello: function (person) {
  console.log("Hello, " + person);
 }
};
```

这里创建一个新对象 objB，它包含两个属性，一个包含字符串的 name 属性以及一个保存函数的 hello 属性。由于在 JavaScript 中函数也是对象，因此可以像对待其他值一样将其传来传去，也可以将其赋给某个变量。上面这个例子中的方法接收一个字符串参数，并向浏览器的调试控制台输出一条消息：

```
objB.hello("Gentle Reader"); //prints: "Hello, Gentle Reader"
```

2.4.2 创建一类新对象

可以声明对象使其包含所需的属性，不过如果你想要创建多个包含相同属性定义的对象，该怎么办呢？当然可以逐一创建它们，不过更加高效的方式是使用构造函数。构造函数是一种特别的函数，它会根据分配给它的属性创建新的对象。在定义好该函数后，可以使用 new 命令调用构造函数以创建新的对象。为了区分构造函数与普通函数，约定构造函数的名称首字母大写：

```
function MyObject (person) {
 this.name = person;
 this.say = function () {
  console.log("It's " + this.name);
 };
}

var objA = new MyObject("Gentle Reader");

objA.say(); //prints: "It's Gentle Reader"
```

留意，在构造函数中用于添加属性的特殊对象 this。该对象将作为构造函数的返回值。在构造函数中声明的任何变量如果没有关联到 this 对象，则无法在构造函数外直接访问。

2.4.3 原型

使用构造函数创建对象实例是在基于类的语言中的习惯做法。事实上，当你查看 JavaScript 文件中对类的介绍时，你会发现它通常所指的就是构造函数。不过该术语有时候会让人忽略了 JavaScript 实际上是基于原型的语言。

当在 JavaScript 中创建新对象实例时，实际上创建新对象，该对象从其构造函数对象（即其原型对象）中继承它的所有属性。可以通过构造函数的 prototype 属性直接访问这些对象。赋予原型的任何属性都将被派生自该类型的所有对象所共享。以下代码基于之前的示例构造函数：

```
MyObject.prototype.hello = function () {
  console.log("Hello, " + this.name);
};

objA.hello(); //prints: "Hello, Gentle Reader"

var objB = new MyObject("Inspired Coder");

objB.hello(); //prints: "Hello, Inspired Coder"
```

这里为构造函数的原型对象添加了 hello 函数，这样做之后，把该函数同时添加到之前声明的 objA 对象以及新创建的 objB 对象中。

本书中创建了一些被很多示例共享的类（一个构造函数与 prototype 属性）。通常情况下，把这些类放入一个独立的文件然后导入 HTML5 文件中。

2.4.4 函数风格

JavaScript 的一大优势在于函数是作为一等公民而存在的对象。这意味着，可以将函数赋予变量，将它们传来传去并将它们作为其他函数的参数，而这是很多编程语言所不具备的强大功能与概念。尽管它蕴含的内容可以演变得相当复杂，但是这个概念却非常简单明了，并且在本书大量运用。如果你能够很好地掌握这一概念，你将成为一个成功的 JavaScript 程序员。

你已经在之前创建一个动画循环的时候使用过函数参数，setInterval 与 requestAnimationFrame 都有使用回调函数作为它们的参数。除此之外，还可以用函数简化代码。例如，下面用一个 for 循环遍历一个数组中的值。创建计数器变量 i，它在每次循环中递增并用于访问数组的值。声明一个包含三个元素的数组并将它们轮流输出到控制台。

```
var arr = [1, 2, 3];
for (var i = 0, len = arr.length; i < len; i++) {
  console.log(arr[i]);
}
```

也可以用函数实现上述功能。每个 JavaScript 数组对象都有一个名为 forEach 的方法，该

方法接收一个函数作为参数。forEach 方法会遍历所有的数组元素，将每个元素作为第一个参数传递给用户定义的函数，同时该元素在数组中的索引位置会作为第二个可选参数传入。以下是采用该方法实现之前那个例子的代码：

```
var arr = [1, 2, 3];
arr.forEach(function (element, i) {
 console.log(element);
});
```

以上代码段跟之前的示例实现了相同的功能（将数组中的每个元素输出到控制台），不过结构却大不相同。这么做有以下几点好处。首先，不用为了遍历声明任何临时变量，尤其是这些变量在这段代码执行完后就没用了。不过更重要的在于，这种风格的代码使得你能够写出函数层面的代码。这意味着，可以通过查看作为参数传入的函数与函数接收的参数了解代码如何工作，而无须担心循环的状态或依赖的变量是否设置正确。这有利于调试，因为当发生错误时，可以通过一层层深入栈跟踪上的函数调用来定位发生错误的代码。

由于执行函数需要一些额外的计算，采用函数实现遍历的一个缺点就是其速度会慢于 for 循环。当开发者在编写程序时，经常需要同时考虑代码的可读性与执行速度，而这只能交由测试决定。通常情况，你应该尽可能编写简单易懂的程序，而让浏览器厂商去关心如何让代码执行得更快。

在本书中根据速度和简洁性的不同需求，两种遍历方式都有用到。使用 JavaScript 与 DOM 更常见也更核心的用法也需要将函数作为事件处理程序传入。我们会在下一节介绍用户交互时了解这是如何工作的。

2.5 用户交互

用户交互可能是你选择本书的一个重要原因。毕竟，如果没有交互或者无法对动画产生一些动态的影响，你可能就去看电影了。用户交互是基于用户事件的，这些事件通常是鼠标事件、触摸事件以及键盘事件。让我们快速地浏览各种用户事件类型以及如何处理这些事件。

2.5.1 事件与事件处理程序

为了理解事件，你必须理解一些额外的概念：监听器与事件处理程序。监听器决定一个元素是否应该响应某个事件，而事件处理程序则是当事件发生时将要调用的函数。

我们绘制到 canvas 元素上的形状自身并不具备监测事件的功能。不过，HTML 元素具备这一功能，这意味着，可以通过 DOM 接口捕获用户输入，相对于绘制的对象计算出事件发生的位置，然后根据这一信息作出进一步的决定。本节将介绍如何捕获 DOM 事件，而随后几章则会介绍如何利用这一点实现动画交互。

2.5.2 监听器与事件处理程序

正如之前介绍的，监听器是监听事件的对象。可以通过调用 DOM 元素的 addEventListener

方法指定它作为某个特定事件的监听器。可以传入一个字符串作为该方法的第一个参数，该字符串用于指定所要监听的事件的类型，该方法的第二个参数则是元素接收到事件后将要调用的事件处理程序。函数的语法示意如下：

```
element.addEventListener(type, handler [, useCapture]);
```

第三个参数通常是可选的，不过在有些浏览器中例外。为此，在本书中，总会将 useCaptured 的默认值 false 传给作为第三个参数的值传入。该参数会影响到事件如何沿着 DOM 树向上传递，不过该特性与本书的示例无关。有关 DOM 事件流详见 http://www.w3.org/TR/DOM-Level-3-Events/#event-flow 中的规范。

一个典型的例子就是监听 canvas 元素的鼠标单击事件（鼠标事件很快就会讨论到）：

```
canvas.addEventListener('mousedown', function (event) {
  console.log("Mouse pressed on element!");
}, false);
```

事件类型 mousedown 作为第一个参数是一个字符串，因此，请确保你仔细检查了所要监听的事件类型，如果拼写错误就意味着你将监听一个不存在的事件。因为 addEventListener 不会对你指定的事件类型有任何异议，所以这里的 bug 会很难追踪到，那个时候你就郁闷了。

之前说到监听器会监听事件，其实更确切的说法应该是会通知监听器事件的发生。在内部实现中，触发事件的对象会维护一个列表，其中包含每个监听器对象。如果一个对象能够触发多种类型的事件，如 mousedown、mouseup 与 mousemove，则它会为它可以触发的每一个事件类型维护一个监听器列表。每当其中一个事件发生时，该对象就会遍历对应的列表并通知其中的每个对象什么事件发生了。

描述事件的另一种说法是监听器对象订阅了某个特定的事件，而触发事件的对象则负责把那个事件广播给所有的订阅者。

此外，如果你不想让一个对象再监听某个特定事件，就还可以通过调用其 removeEventListener 停止监听，或者取消订阅。请注意，该方法的参数与 addEventListener 方法完全一致。

```
element.removeEventListener(type, handler [, useCapture]);
```

以上代码将监听器从某个特定事件的监听器列表中移除，这样它就无法再接收到之后的事件通知。

让我们看看上述内容是如何应用到示例中的。在之前的框架文件中加入以下代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Event Demo</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas');
```

```
    canvas.addEventListener('mousedown', function (event) {
     console.log("mouse down");
    }, false);

    canvas.addEventListener('mouseup', function (event) {
     console.log("mouse up");
    }, false);
   };
   </script>
  </body>
 </html>
```

在本例中，首先通过DOM接口使用document.getElementById方法访问canvas元素，然后将其保存在变量canvas中。然后为mouseup和mousedown事件添加了监听器。其中传入内联的回调函数（记住，函数可以作为对象传递），作为事件处理程序，该函数会把消息输出到调试控制台中。作为事件处理程序的回调函数接收一个事件对象作为参数，其中包含了事件相关的信息，如事件的名称以及触发事件的对象。以鼠标事件为例，该参数包含了事件触发时鼠标所在的位置信息，哪个鼠标按键被按下等。而键盘事件则包含了当事件触发时哪个键被按下的信息。

将以上示例代码另存为02-event-demo.html文件。当你在Web浏览器中载入该文件时，你将看到在canvas元素上每次按下或释放鼠标的时候都会在调试控制台中转出一条消息。请确保该示例可以正常运行，以及CSS文件的路径设置正确。这虽然是一个简单的示例，不过它却是检验你的开发环境是否设置正确的一个好测试。

如果你刚开始接触JavaScript，现在能够成功运行该示例并且理解它，那么恭喜你，你已经从初学者升级成为中级学员。

既然你已经对事件处理程序有了大致了解，就可以尝试更好地理解监听器了。一个对象触发一个事件，并将其广播出去，或者通知事件的监听器。它到底做了什么？其实很简单，它只是调用了那个对象上那个函数名与事件处理程序名称一致的函数而已。在之前的示例中，02-event-demo.html文件将canvas添加为一系列鼠标事件的监听器。canvas元素在其内部实现中会为每一个事件维持一个列表，也就是说，针对mousedown事件它会有一个列表，而针对mouseup事件会有另一个列表。

当用户在canvas元素上按鼠标按钮时，canvas回应："嘿，鼠标按下了！赶紧通知监听器！"。然后它遍历mousedown列表并查看其中内容，当它找到一个被指定为事件处理程序的函数引用后，就会调用监听器上被引用的那个函数。如果还有其他对象也注册为mousedown事件的监听器，那么它们也会出现在那个列表中，并且之前定义的所有事件处理程序都将随之调用到。

同样事情会发生在鼠标释放时，唯一不同的地方在于它此时查看的是mouseup列表。

以上是关于事件和事件处理程序的基础知识。接下来，我们将介绍用于交互的各种事件类型。

2.5.3 鼠标事件

为了捕获一个鼠标事件，必须为一个DOM元素添加一个监听器以处理事件。鼠标事件类

型由字符串指定，不同的浏览器所支持的事件类型各不相同。以下是最常见的一些事件：

- mousedown
- mouseup
- click
- dblclick
- mousewheel
- mousemove
- mouseover
- mouseout

这些事件类型不言自明。为了了解它们，试着创建并运行以下文件，它会将发生在 canvas 元素上的每个鼠标事件输出到控制台中。可以在 03-mouse-events.html 文件中找到完整的示例代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Mouse Events</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas');

   function onMouseEvent (event) {
    console.log(event.type);
   }

   canvas.addEventListener('mousedown', onMouseEvent, false);
   canvas.addEventListener('mouseup', onMouseEvent, false);
   canvas.addEventListener('click', onMouseEvent, false);
   canvas.addEventListener('dblclick', onMouseEvent, false);
   canvas.addEventListener('mousewheel', onMouseEvent, false);
   canvas.addEventListener('mousemove', onMouseEvent, false);
   canvas.addEventListener('mouseover', onMouseEvent, false);
   canvas.addEventListener('mouseout', onMouseEvent, false);
  };
  </script>
 </body>
</html>
```

注意，我们为每个鼠标事件类型绑定了同一个处理程序，该函数会输出被分派的事件类型。

因为我们只能引用到 canvas 元素，而无法引用到那些绘制在 canvas 上的线和形状，所以我们也无法为特定的线或形状添加事件监听器。如果你画了一个矩形并将其作为按钮，canvas 是无法获知按钮的边界的。为了监测到按钮单击，需要捕获 canvas 元素的鼠标事件并经由计算得出鼠标相对于按钮的位置，而这一切都必须由你自己完成。在阅读本书的过程中，你将会看

到用于达到这一目的的一些方法。

2.5.4 鼠标位置

每个鼠标事件有两个属性用于确定鼠标的当前位置：pageX 与 pageY。结合这两个属性以及 canvas 元素相对文件的偏移量，可以确定鼠标在 canvas 元素上的相对坐标。遗憾的是，并不是所有的浏览器都支持这两个属性，所以在这些情况下，可能要用到 clientX 与 clientY 属性。

考虑到每次需要鼠标位置的时候都要计算偏移量会比较麻烦，同时考虑到示例代码的简洁性，我们决定将这段跨平台的、计算鼠标位置的代码封装到工具函数 utils.captureMouse 中，并放入 untils.js 文件中，然后再将其导入文件中：

```
utils.captureMouse = function (element) {
 var mouse = {x: 0, y: 0};

 element.addEventListener('mousemove', function (event) {
  var x, y;
  if (event.pageX || event.pageY) {
    x = event.pageX;
    y = event.pageY;
  } else {
    x = event.clientX + document.body.scrollLeft +
      document.documentElement.scrollLeft;
    y = event.clientY + document.body.scrollTop +
      document.documentElement.scrollTop;
  }
  x -= element.offsetLeft;
  y -= element.offsetTop;

  mouse.x = x;
  mouse.y = y;
 }, false);

 return mouse;
};
```

该函数接受一个 DOM 元素作为参数并为之添加一个 mousemove 事件的处理程序，最终返回一个包含 x 与 y 属性的对象。当鼠标在该元素上移动时，处理程序会根据事件发生的位置（每个浏览器提供不同支持）以及该元素相对于文件的偏移量计算出鼠标在元素上的相对位置，然后将表示位置的两个值赋值给作为函数返回值的对象，最终可以在调用该函数的主脚本中访问它们。鼠标的 x、y 坐标值是相对于元素的左上角坐标（0,0）而言的。

例如，以下代码，你在脚本开始的地方调用该函数并将 canvas 元素作为参数值传入：

```
var canvas = document.getElementById('canvas'),
  mouse = utils.captureMouse(canvas);
```

在初始化 mouse 对象之后，每当你需要确认鼠标的当前位置时，可以随时查询它的 x 与 y 属性。以下是一个完整的示例，它演示了在本书中如何使用该函数：

```
<!doctype html>
<html>
```

```
<head>
 <meta charset="utf-8">
 <title>Mouse Position</title>
 <link rel="stylesheet" href="style.css">
</head>
<body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <script src="utils.js"></script>
 <script>
 window.onload = function () {
  var canvas = document.getElementById('canvas'),
      mouse = utils.captureMouse(canvas);

  canvas.addEventListener('mousedown', function () {
   console.log("x: " + mouse.x + ", y: " + mouse.y);
  }, false);
 };
 </script>
</body>
</html>
```

在打开调试控制台的情况下运行该文件（04-mouse-position.html）。确保导入了 utils.js 文件中的 utils.captureMouse 函数并把 utils.js 文件导入该文件。

当用鼠标单击 canvas 元素时，你会看到一条消息使用 mouse.x 与 mouse.y 属性显示了鼠标的当前位置。

2.5.5 触摸事件

由于触摸屏设备越来越流行，因此看看如何借助 JavaScript 使用它们就成了一个很有价值的想法。触摸事件与鼠标事件相似，不过也有一些显著的不同。一个触摸点可以被想象成一个鼠标光标，不过，鼠标光标会一直停留在屏幕上，而手指却会从设备上按下、移动以及释放，所以某些时刻光标会从屏幕上消失。当查询触摸点的位置时，请务必考虑到这一点。其次，不存在与 mouseover 等效的触摸事件——要么发生了一次触摸要么没有，不存在手指悬停在触摸屏上的概念。最后，同一时间可能发生多点触摸。某个触摸点的信息会保存在触摸事件的一个数组中，不过在以下示例中仅会用到第一个触摸点。

以下触摸事件将用于与动画的交互：

- touchstart
- touchend
- touchmove

示例 05-touch-events.html 示范了以上事件的用法，如果你想看到它们发挥作用，请确保你使用了一个支持触摸事件的设备或模拟器。

```
<!doctype html>
<html>
 <head>
```

```
<meta charset="utf-8">
<title>Touch Events</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
<script>
window.onload = function () {
 var canvas = document.getElementById('canvas');

 function onTouchEvent (event) {
  console.log(event.type);
 }

 canvas.addEventListener('touchstart', onTouchEvent, false);
 canvas.addEventListener('touchend', onTouchEvent, false);
 canvas.addEventListener('touchmove', onTouchEvent, false);
};
</script>
</body>
</html>
```

以上代码做了与鼠标事件的示例（03-mouse-events.html）中完全一样的事情：当在canvas元素上监测到一个触摸事件时，它会把事件类型输出到调试控制台中。

2.5.6 触摸位置

类似utils.captureMouse函数，可以使用utils.captureTouch函数确定第一个手指在元素上触摸的位置。两个函数的实现类似，不过由于可能出现没有手指触摸到屏幕的情况，因此为返回的对象添加了一个isPressed属性。同时，如果没有手指触摸到屏幕，x与y属性就会设置为null。将以下函数添加到utils.js文件中：

```
utils.captureTouch = function (element) {
 var touch = {x: null, y: null, isPressed: false};

 element.addEventListener('touchstart', function (event) {
  touch.isPressed = true;
 }, false);

 element.addEventListener('touchend', function (event) {
  touch.isPressed = false;
  touch.x = null;
  touch.y = null;
 }, false);

 element.addEventListener('touchmove', function (event) {
  var x, y,
   touch_event = event.touches[0]; //first touch

  if (touch_event.pageX || touch_event.pageY) {
   x = touch_event.pageX;
   y = touch_event.pageY;
```

```
    } else {
     x = touch_event.clientX + document.body.scrollLeft +
       document.documentElement.scrollLeft;
     y = touch_event.clientY + document.body.scrollTop +
       document.documentElement.scrollTop;
    }
    x -= offsetLeft;
    y -= offsetTop;

    touch.x = x;
    touch.y = y;
   }, false);

   return touch;
};
```

以上函数定义与获取鼠标位置的那个版本类似，不过加入了一些额外的事件监听器。它接收一个 HTML 元素作为参数同时返回一个包含 x、y 与 isPressed 属性的对象。Touchmove 事件处理程序负责追踪第一个触摸点与元素的相对坐标，其中一些跨浏览器的代码用来计算偏移量。还分别为 touchstart 与 touchend 事件添加了事件处理程序，其中会分别将 isPressed 属性设置为 ture 与 false，如果不存在有效的触摸点，touchend 事件处理程序还会将 x、y 属性设置为 null。

可以像之前一样初始化一个 touch 对象：

```
var canvas = document.getElementById('canvas'),
  touch = utils.captureTouch(canvas);
```

在访问 x、y 属性之前务必确保此时有触摸点按下，否则，它们的值将为 null，并很可能把动画计算弄错。因此，在查询触摸位置前，务必检查是否有触摸点按下：

```
if (touch.isPressed) {
  console.log("x: " + touch.x + ", y: " + touch.y);
}
```

由于我们更关注使得示例代码能够工作的数学原理，因此触摸事件不会在本书中大量使用。不过，如果你想在一个触摸设备上体验动画效果，以上内容应该足以让你知道如何使用它。

2.5.7 键盘事件

键盘事件类型与所有事件类型一样都是由字符串定义的，不过仅包含以下两个：

- keydown
- keyup

可以在支持字符输入的 HTML 元素上监听键盘事件，例如，一个文本区域元素，监听的方法与你监听 canvas 元素上的鼠标事件一致。不过，在捕获键盘事件之前，首先需要将屏幕的焦点设置到该元素上。假设变量 textarea 中保存了一个 HTML textarea 元素，可以通过如下方式设置它的焦点并捕获 keydown 事件：

```
textarea.focus();
textarea.addEventListener('keydown', function (event) {
 console.log(event.type);
}, false);
```

不过，在大多数情况下，更加简单的做法是直接在网页上监听键盘事件而不管谁获得了焦点。为了做到这一点，可以直接将一个键盘事件监听器绑定到全局的 window 对象上。以下示例会监测何时用户按下与释放某个键，而不管哪个元素持有焦点（示例 06-keyboard-events.html）。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Keyboard Events</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <script>
  window.onload = function () {

    function onKeyboardEvent (event) {
     console.log(event.type);
    }
    window.addEventListener('keydown', onKeyboardEvent, false);
    window.addEventListener('keyup', onKeyboardEvent, false);
  };
  </script>
 </body>
</html>
```

2.5.8 键盘码

通常在事件处理程序中，你想知道哪个键按下。之前在介绍事件的时候，我们看到事件处理程序中传入了一个事件对象，其中包含事件发生的相关信息。在一个键盘事件中，可以通过事件对象的 keyCode 属性获知哪个键按下。

KeyCode 属性包含一个代表按下的物理键的数字值。如果用户按下 a 键，那么无论是否有其他键按下，keyCode 都会包含数字 65。如果用户先按下 Shift 键再同时按下 a，我们将获得两个键盘事件，一个是 Shift 键的事件（键盘码 16），另一个为 a 键的事件（键盘码 65）。

在下面这个示例中，测试按下的键盘码是否是方向键的键盘码，方向键的键盘码包含 37、38、39 与 40 这些值。如果按下的键与其中任何一个值匹配，会向控制台输出一条消息，显示按下的方向；否则，输出那个未知的键盘码。以下是 HTML 文件 07-key-codes.html 的代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Key Codes</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <script>
  window.onload = function () {
```

```
 function onKeyboardEvent (event) {
  switch (event.keyCode) {
  case 38:
   console.log("up!");
   break;
  case 40:
   console.log("down!");
  break;
 case 37:
  console.log("left!");
  break;
 case 39:
  console.log("right!");
  break;
 default:
  console.log(event.keyCode);
 }
}

 window.addEventListener('keydown', onKeyboardEvent, false);
};
</script>
</body>
</html>
```

通常你不会记得你想捕获的键的数字值。为了方便查询这一信息，创建了一个 keycode.js 文件作为参考速记表，甚至可以把它导入文件中，这样就可以利用键的名称查询键盘码而不用直接使用数字，并且使代码更加易懂。该脚本将创建一个全局对象，其中包含一系列以键的名称作为属性名的属性，把属性值映射到键所对应的键盘码，这些属性涵盖了键盘上几乎所有的键。你可以在 www.apress.com 或 http://github.com/lamberta/html5-animation 这两个网站上找到该文件以及本书中的所有源代码。

让我们用使用键名称而不是键盘码改写之前的示例。将 keycode.js 文件放在该示例所在的目录下，并导入 08-key-names.html 文件中：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Key Names</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <script src="keycode.js"></script>
  <script>
  window.onload = function () {

  function onKeyboardEvent (event) {
    switch (event.keyCode) {
    case keycode.UP:
     console.log("up!");
     break;
    case keycode.DOWN:
```

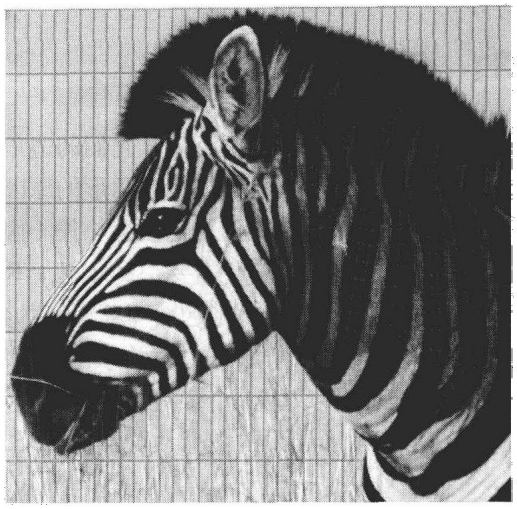
```
        console.log("down!");
        break;
      case keycode.LEFT:
        console.log("left!");
        break;
      case keycode.RIGHT:
        console.log("right!");
        break;
      default:
        console.log(event.keyCode);
      }
    }
    window.addEventListener('keydown', onKeyboardEvent, false);
  };
  </script>
 </body>
</html>
```

以上示例会像之前的示例 07-key-codes.html 一样输出同样的消息，不过由于我们现在使用键名称而不是键盘码，因此 switch 语句变得更加容易理解了。而且除了让代码更加清晰之外，这种做法还可以避免不小心输错数字。

不过本书中的示例并不打算使用 keycode.js，因为书中的示例相对简单，而且我希望读者能更加关注于示例的中心思想而不是这些细枝末节。但是，当我们编写一个更加复杂的程序时，我们应该考虑使用该文件，因为它确实有助于提升代码的可读性。

2.6 小结

本章介绍了理解书中示例所需要的 JavaScript 基础知识。现在你应该了解了如何创建 HTML5 文件、调试、循环、事件以及事件处理程序。本章简单介绍了 JavaScript 对象，基本的用户交互，并且创建了一系列用于简化代码的工具函数。内容很多，不过即便你对有些内容仍旧感到模糊也不用担心，因为以上大多数内容将在我们介绍具体技术时给予更具体的描述，你也可以随时翻回到本章复习这些基础知识。至少，你现在熟悉了很多术语和概念，已经准备好了学习后续内容。



第 3 章　动画中的三角学

本章涵盖以下内容：

- 三角学
- 角
- 三角函数
- 旋转
- 波形
- 圆与椭圆
- 勾股定律
- 两点间距离

本章可以说是本书的开篇，因为三角学在动画技术中广泛使用，从第 5 章开始的示例中你就可以看出这点。事实上，甚至在介绍渲染技术的第 4 章也会涉及它。尽管如此，如果你已经掌握了基本的三角学知识或者盼望学习动画，你还是可以直接跳过本章。当你遇到某些难以理解的概念时你可以随时翻回本章。

许多人会下意识地回避数学与三角学，他们会用诸如“我不太擅长数字”之类的借口，而使用三角学编程的一个有趣的地方在于你几乎不需要和数字打交道，你只需关注于可视化的图形以及它们之间的关系。大多数情况下，你只需要处理与位置、距离以及角相关的变量，而不太会跟实际的数字打交道。所以学习三角学主要就是掌握各种边角关系，事实上，用于实现基本动画的 90%的三角学知识都可以归结于两个函数：Math.sin 与 Math.cos。

3.1 三角学

三角学主要研究三角形以及它们的边角关系。观察任意一个三角形，你都会发现它有三条边和三个角（这也是它为什么称为三角形的原因），而它们之间拥有特殊的关系。如果某个三角形的一个角扩大，那么它所对应的边就会拉长（假设另外两条边保持原有长度），并且另外两个角也会随之变小。它们的具体变化需要一些计算，不过其中的比率已经过充分研究并可以通过编码实现。

有一类特殊的三角形包含一个 90° 的角，它称为直角三角形并由一个位于直角处的小方块加以标识。直角三角形的边角关系比较简单，可以很容易地利用一些基本公式推算出结果。这使得直角三角形成为一个很有用的概念，本章中的所有三角学知识以及本书接下来的内容都是围绕直角三角形展开的。

3.2 角

由于三角学关注于角，因此让我们先看看这个主题。一个角是由两条相交线形成的形状或两条线之间的空间。空间越大，角的度数也就越高。事实上，两条交叉线可以构成四个角，如图 3-1 所示。

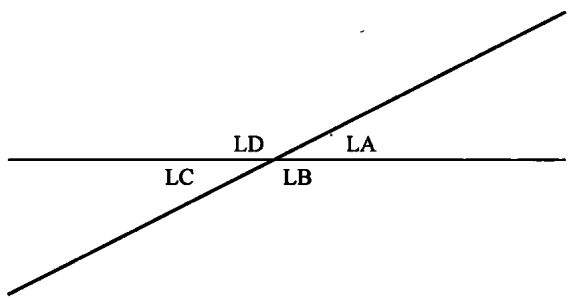


图 3-1 两条线构成四个角

3.2.1 弧度和角度

角的度量单位分别有角度与弧度两大系统。你可能对角度比较熟悉，并且可以按要求画出 45° 或 90° 的角。一个圆包含 360° 的系统已经成为我们文化的一部分。当人们说他们转了 180° 弯时，意味着他们转向了相反的方向，甚至于有些时候我们不是在谈论物理上的方向，而是指相反的观点。不过，在衡量角的大小的时候，计算机更倾向于使用弧度的概念。所以，无论喜欢与否，你都需要了解弧度。

一弧度约等于 57.295 8°。乍一看，这个数字毫无规律，实际上背后隐含着相关的数学原理。一个完整的圆，或者说 360° 等于 6.283 2 弧度。还记得 pi，那个符号 π 吗？它的值约等于

3.141 6，也就是说一个圆（6.2832 弧度）恰好等于 2pi。虽然这么听上去还不是特别有道理，但是也足够我们继续工作了，很快你就会习惯于将 360° 想象成 2π，180° 是π，90° 是π/2……。图 3-2 展示了一些常见的弧度值。

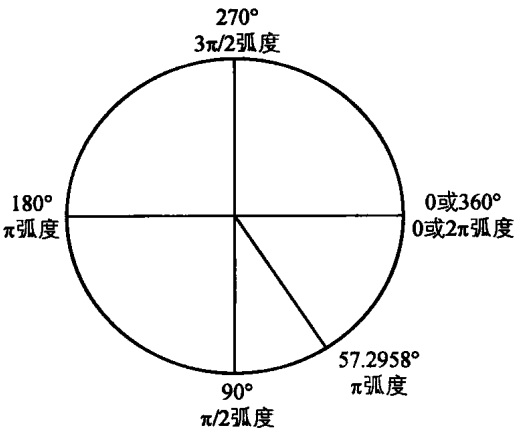


图 3-2 弧度与角度

现在，我们不再就刚才的问题做进一步的讨论，你只需要知道现在你可以放心地使用弧度。你可能会有点失望，不过没关系，适应它并放心去使用弧度就好了。而且，你还会遇到很多需要同时用到角度与弧度的情况。例如，当你在 canvas 元素上旋转一幅图像时，你会用到 context.rotate（angle）方法，其中参数 angle 的大小由弧度表示。不过，在许多场景中我们还是会使用更容易让人理解的角度来设置场景中的物体。因此，我们需要一个能够方便转化它们的方法。

那么为什么要有两套完全不同的系统呢？实际上，这并不是必需的。你可以总是用弧度作为度量单位，而且这也是 canvas 乐于看到的。不过，对于大多数人来说，用角度去衡量角的大小是一种更加自然的思维方式。原因很简单，因为大多数人都是这么想的。我可以肯定如果你告诉设计师输入一个弧度值用于旋转他们为商标创建的文字的话，他们一定会对你的说法表现得目瞪口呆。当程序员围绕着 HTML 工作时，必须跨越设计与开发的界线，而 JavaScript 与大多数的开发语言类似，都使用弧度作为角的度量单位。由于你很可能要同时用到角度与弧度，因此你需要知道如何在两者间来回转换。以下是转换公式：

```
radians = degrees * Math.PI / 180
degrees = radians * 180 / Math.PI
```

在本书中会出现大量的公式。我会特别指出你应该牢记在脑海里的那些公式，而这个转换公式就是其中的第一个。你应该做到当你需要用到它们的时候就能立刻在键盘上把它敲出来。如果你还是记不住，就把它们写在一张纸上并贴在你的电脑屏幕边。这里甚至用到了 JavaScript 中的 Math.PI 常量，因为这就是你在以后的开发中重复输入的内容。

根据该公式，你可以很容易地推算出 180° 等于 3.14…弧度。换句话说，半圆可以换算成π弧度，这很容易理解因为一个整圆就是 2π。反之，一弧度约等于 57.29。

3.2.2 canvas 坐标系

尽管我们现在讨论的主题是角度，不过从数字的角度来看现在也是一个介绍 canvas 上的空间分布的好时机。如果你之前接触过任何坐标系，在这里你可能稍微有点犯晕，因为这里的一切都有点颠倒的感觉。

最常见的二维坐标系以 x 轴作为水平坐标，以 y 轴作为垂直坐标，canvas 元素也遵循同样的方式。不过，通常情况下（0,0）坐标会显示在空间的正中心，随后 x 轴的坐标值向右以正数

形式不断增大，向左以负数形式不断变小，而 y 轴的坐标值向上以正数形式不断增大，向下以负数形式不断变小，如图 3-3 所示。

而 canvas 元素却是基于视频画面的坐标系，其中（0,0）处在空间的左上角，如图 3-4 所示。x 轴的坐标值还是从左往右不断增大，而 y 轴的坐标值的变化却与标准坐标系相反了，向下以正数形式不断增大，向上以负数形式不断变小。这个坐标系有它的历史背景，因为电子枪是从左往右，从上往下扫描屏幕的。不过这一背景并不是很重要，你只需要知道它就是这么定义的，而且短时间内不会发生变化。

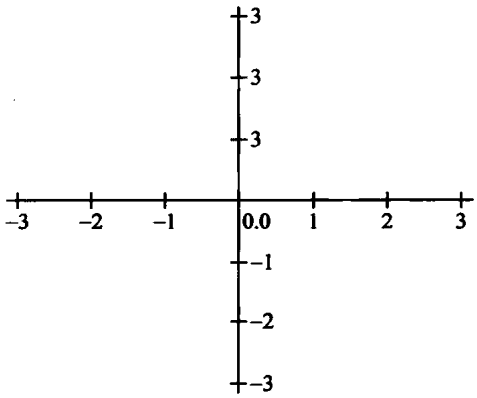


图 3-3 标准坐标系

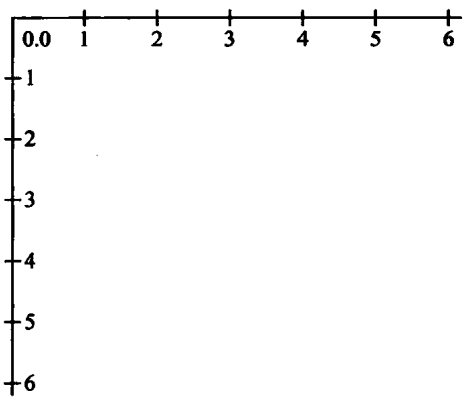


图 3-4 canvas 元素坐标系

不过，请等等，除了坐标系之外，我们还要介绍与角度测量相关的内容。在大多数系统中，角度的测量是以逆时针方向为正值的，0° 表示一条线沿着 x 轴正方向延伸，如图 3-5 所示。

canvas 元素在这方面又显示了它的与众不同，如图 3-6 所示。在 canvas 元素中，顺时针的角度才是正值，逆时针的角度反而成了负值。

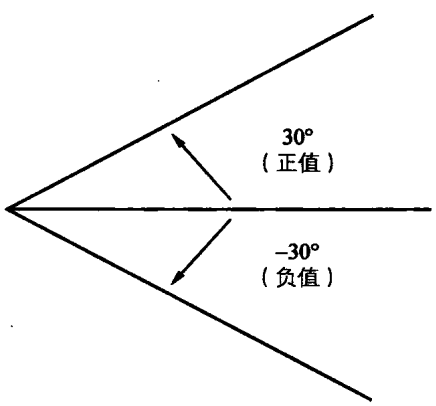


图 3-5 常见的角度测量

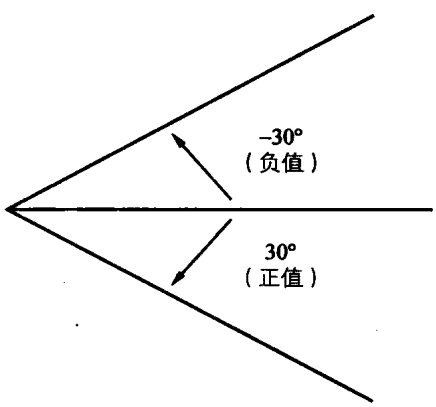


图 3-6 Canvas 元素上的角度度量

在这种情况下，有两个选择，要么创建一个可以在渲染图像之前自动反转角度的系统，要

么适应这个系统在每次的计算中反转角度。建议采用后者，这也是本书中代码所采取的策略。

3.2.3 三角形的边

三角形的边本身并没有什么值得一提的，不过它们会涉及一些特别的术语。如果没有特别说明，以下内容都是针对直角三角形的，即其中包含一个 90° 角的三角形。在这种三角形中，与 90° 角相邻的两条边称为**直角边**，而与它相对的边则称为**斜边**。斜边总是最长的一条边。

当提到角的**对边**时，就是指那条与角不相邻的边。而邻边则是与角相邻的两条边。大多数情况，会跟除直角外的另外两个角中的一个角打交道。此时，邻边仍旧是指该角相邻的直角边，而不是其相邻的斜边。

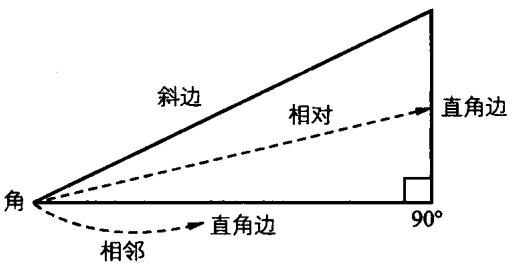


图 3-7 直角三角形的各个部分

三角形最有趣的地方就在于其边角的大小关系，因为它们是相互影响的。这些关系对于动画起到了重要作用，因此它们就成为下面所要介绍的内容。

3.2.4 三角函数

JavaScript 包含用于计算多种三角关系的三角函数，如正弦、余弦、正切、反正弦、反余弦以及反正切。本节会给出这些三角关系的定义以及用于访问它们的 JavaScript 函数，然后我们会看到有关这些函数的一些实际应用。

1. 正弦

以下是你将要接触到的第一个实际的三角关系：一个角的**正弦**表示与该角相对的直角边与斜边的比例（当谈及正弦时，我们总是相对一个角而言的）。在 JavaScript 中，可以使用 Math.sin(angle) 函数。

图 3-8 显示了一个 30° 角的正弦值。该角的对边长度为 1，斜边长度为 2。它们的比率是 1:2，用数字表示即 1/2 或 0.5。因此，我们可以认为 30° 角的正弦值为 0.5。可以在浏览器调试控制台中输入以下代码验证这一点：

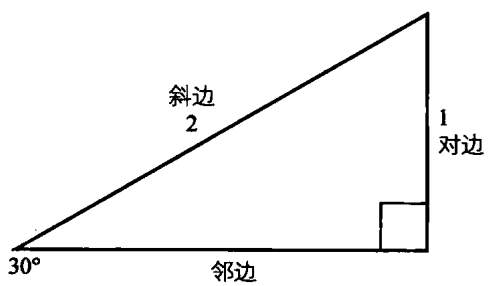


图 3-8 角的正弦值为对边长度/斜边长度

```
console.log(Math.sin(30));
```

不过你会看到在控制台的输出结果却是–0.988 031 624 092 862，这与 0.5 相差太多了。你能猜到哪里出错了吗？对了，我们忘记把角度转换为弧度了。你可能会时不时犯这样的错误，出错的时候能习惯性地想到它就可以了。以下是加上了角度转换的正确代码：

```
console.log(Math.sin(30 * Math.PI / 180));
```

成功输出 0.5!

实际上，你有可能会得到类似 0.499 9…的结果，这并不是计算出现了错误，而是二进制电脑在表示浮点数时经常会出现的结果。不过这已经足够接近了，可以把它当做 0.5。

对于像上图一样的三角形来说，你可以理直气壮地说它的角度是 30°，对边长度为 1，斜边长度为 2。但是到了真实世界中，至少说在 canvas 坐标系中，这种说法就不太对了。还记得在 canvas 坐标系中，垂直往下以及逆时针的角度才是正值吗？所以，在上面这个例子中，角的对边和角自身都是负值，如图 3-9 所示。

此时，对边和斜边的比率变成了-1/2，而正弦值所对应的角度也变成了-30°。因此-30°角的正弦值为-0.5。改变 JavaScript 中的 console.log 语句以验证这一点：

```
console.log(Math.sin(-30 * Math.PI / 180));
```

不算很难懂是吧？接下来让我们看看另一个三角函数：余弦。

2. 余弦

可以通过 Math.cos（angle）函数在 JavaScript 中实现余弦。余弦表示角的邻边与斜边的比率。图 3-10 表现了这一关系。

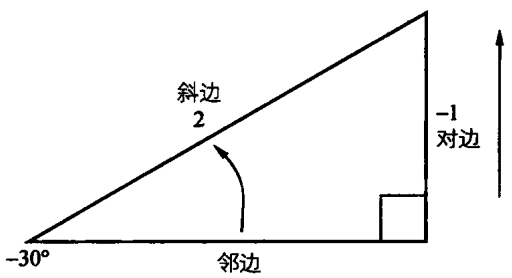


图 3-9 在 canvas 坐标系中的同一个三角形

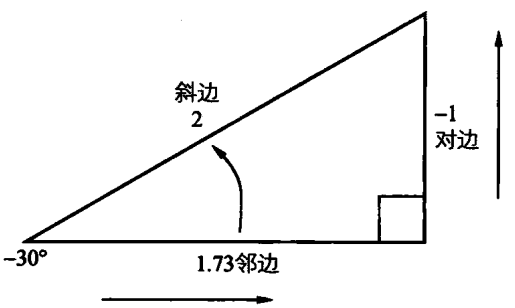


图 3-10 角的余弦值为邻边长度/斜边长度

图 3-10 展示了与图 3-9 同样的角，不过加注了邻边长度的近似值 1.73。注意，因为这条边相对于角向右方延伸，所以在 x 轴上它的长度是正值。该角度的余弦值为 1.73/2 即 0.865。因此可以说-30° 的余弦值为 0.865，用以下代码验证一下：

```
console.log(Math.cos(-30 * Math.PI / 180));
```

以上代码跟之前类似，唯一的区别在于调用了 Math.cos 函数而不是 Math.sin。它的输出结果是 0.866 025 403 784 439，接近于 0.865。它们的差别源于我们之前四舍五入了邻边的长度。对于显示的三角形，实际的邻边长度接近于 1.732 050 807 568 88。如果你将它除以 2，你就会得到接近于-30° 的余弦值。

目前为止，我们研究的一直都是左下方的那个角。如果从右上方的那个角的视点出发又会得出什么不同的结论呢？首先，需要调整三角形的朝向，使右上方的角与坐标系相适应，如图 3-11 所示。我们把这称为将角放入标准位置（尽管 canvas 元素中的标准位置与通常的标

准位置是上下颠倒的）。该角的大小为 60°，因为它是逆时针方向的，所以它是正值。因为垂直方向的边沿着角向下，所以它也是正值。水平方向的边往右延伸，同样它是正值。（在图 3-11 中，对于那些有区别的地方加注了正号，不过通常情况下，正值是无须特别注明的，除非有负号出现，所有的数字默认为正值。）

该角的正弦值是它的对边长度除以斜边长度，即 1.73/2（或 0.865），余弦值为邻边长度除以斜边长度，即 1/2 或 0.5。因此，基本上一个角的正弦值等于另一个角的余弦值，而这个角的余弦值又恰好等于另一个角的正弦值。可能这一发现对于代码并没有用，不过通过这个可以注意，正弦、余弦只不过是一些边角关系和比率，而且它们还是有联系的，这点很重要。

3. 正切

正切是另一个重要的三角函数，在 JavaScript 中通过 Math.tan（angle）函数表示。它表现了对边与邻边的关系，如图 3-12 所示。

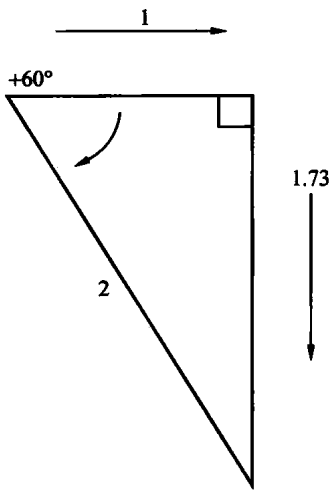


图 3-11 观察另一个角

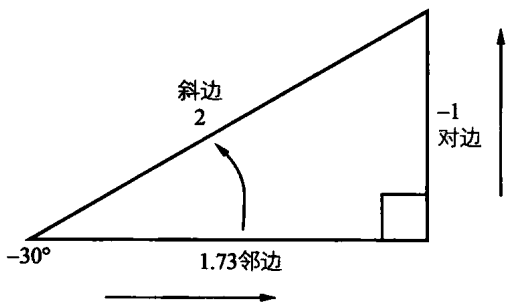


图 3-12 角的正切为对边长度/邻边长度

此时，它们的比率为-1/1.73 或-0.578。为了获得更精确的数字，直接在浏览器的调试控制台中输入以下代码验证结果：

```
console.log(Math.tan(-30 * Math.PI / 180));
```

控制台输出-0.577 350 269 189 627。

说实话，在日常的动画代码中很少用到正切。尽管正切可偶尔用于创建一些有趣的特效，但是你会发现正弦和余弦用得更频繁一些。

另一方面，反正切却非常有用，你很快就会看到示例，现在先把正切记在脑海里吧。

4. 反正弦与反余弦

与正切相似，反正弦和反余弦在一般的动画场景中也不太有用。不过，了解它们的存在以

及如何使用它们仍然是有意义的。简单来说，它们就是正弦与余弦的逆运算，换句话说，输入一个比率，获得对应的角的弧度。它们对应的 JavaScript 函数分别为 Math.asin（ratio）与 Math.acos（ratio）。让我们快速地做一个测试以确保它们以我们预想的方式工作。

因为我们知道 30° 的正弦值为 0.5，所以 0.5 的反正弦值应该为 30°。让我们来试试看：

```
console.log(Math.asin(0.5) * 180 / Math.PI);
```

记住，将结果转换为角度才能看到近似 30° 的值，否则你看到的是 0.523，也就是 30° 对应的弧度制。

30° 的余弦值近似于 0.865。记住，如果你直接拿这个经过四舍五入的值测试的话，由于计算机表示小数的机制，你无法获得 30 这个精确的数字。不过结果会非常接近，足以证明这个概念的正确。以下是测试代码：

```
console.log(Math.acos(0.865) * 180 / Math.PI);
```

你应该会得到近似 30.117 294 747 322 1 的结果。如果用之前 30° 余弦值的测试结果替换掉 0.865，就可以得到更加精确的一个结果。

看，这并没有那么复杂，对吧？你现在几乎已经学习了所有的基础函数。还剩下最后一个，然后你就可以看到三角学的实际作用。

5. 反正切

正如你已经猜到的，反正切也就是正切的逆运算而已。输入角的对边与邻边比率，通过计算反正切可以得到角的弧度。JavaScript 提供了两个函数用于计算反正切。第一个函数为 Math.atan，它的名字和使用方式都与前例类似，它接受角的对边长度除以邻边长度所得的小数作为参数。

例如，从之前的讨论中得知 30° 的正切值为 0.577（经过四舍五入），可以以此为例尝试以下代码：

```
console.log(Math.atan(0.577) * 180 / Math.PI);
```

控制台的输出结果为一个接近 30 的值。现在，这看上去非常简单明了，那么为什么还需要另一个同样功能的函数呢？为了解答这一问题，让我们看看下面这张图：

图 3-13 显示了 4 个不同的三角形：*A*、*B*、*C* 与 *D*。三角形 *A*、*B* 的 *x* 轴坐标为正值。三角形 *C*、*D* 的 *x* 轴坐标则延伸至 *x* 轴的负象限。同理，三角形 *A*、*D* 处于 *y* 轴的负象限，而三角形 *B*、*C* 处于 *y* 轴的正象限。因此，对于 4 个内部角而言，将得到以下的正切值：

- *A*:-1/2 即−0.5;
- *B*: 1/2 即 0.5;
- *C*: 1/(−2)即−0.5;
- *D*: (−1)/(−2)即 0.5。

假如说我们用对边长度除以临边长度得到结果 0.5，你将这个结果作为参数输入 Math.atan（0.5），再将其结果转化为弧度，你得到一个接近 26.57° 的值。然而，此时你并无法分辨这个三角形是 *B* 还是 *D*，因为这两个角所对应的正切值都是 0.5。这似乎看上去有点无足轻重，不过当你接触到本章后面将要介绍的现实示例中的内容后，你就会意识到它的重要性了。

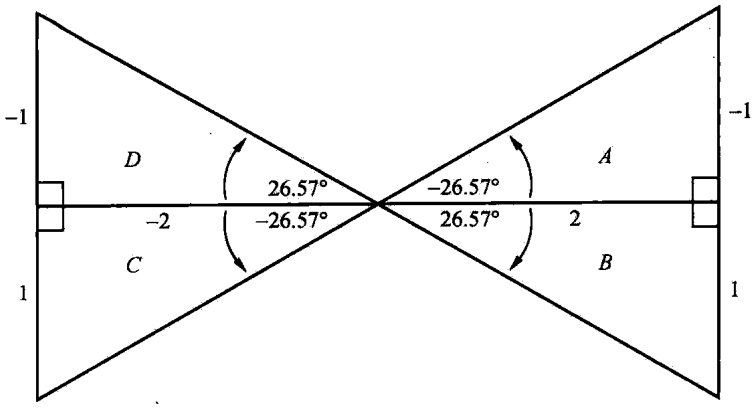


图 3-13 四个象限中的角

让我们欢迎 Math.atan2（y, x）的登场，它是 JavaScript 中的另一个反正切函数，相对而言，它比 Math.atan（ratio）显得更加重要。而且，实际上，你可能专门使用它。该函数接受两个参数：对边的长度与邻边的长度。大多数情况下，该长度为 *x* 轴或 *y* 轴的坐标。一个常见的错误是将对边和邻边的顺序弄反。以之前的例子为例，应该输入 Math.atan2（1, 2）。尝试在控制台中输入以下代码，并记得将结果转换为角度：

```
console.log(Math.atan2(1, 2) * 180 / Math.PI);
```

结果输出 26.565 051 177 078，这正是三角形 *B* 的角度大小。之前提到过可能会无法通过 (−1)/(−2) 这个正切值确定三角形 *D*，那么让我们试试看：

```
console.log(Math.atan2(-1, -2) * 180 / Math.PI);
```

结果为−153.434 948 822 922，你可能预想不到。那么它是怎么一回事呢？或许下面这张图可以作出解释。

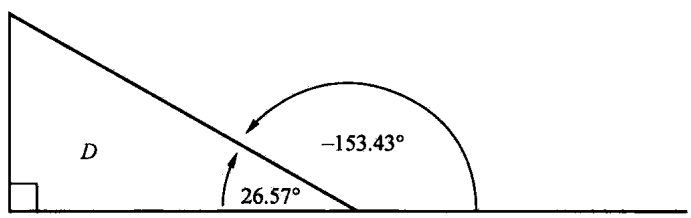


图 3-14 测量角的两种方式

尽管从三角形 *D* 的底边算起，它的内角确实是+26.57°，不过如果你还记得在 canvas 中，角是从 *x* 正轴开始以逆时针方向计算的。因此，在 canvas 元素的视角中我们所关注的这个角的角度为−153.43°。

这有什么用处呢？让我们开始介绍第一个在 canvas 元素中运用三角学的实际应用吧。

3.3 旋转

让我们看看下面这个挑战：绘制一个物体并让它随着鼠标旋转，使其总能指向鼠标。旋转功能很有用可以置于你的工具箱，它可以用于游戏和界面元素等其他方面。并且实际上，旋转功能并不限于鼠标。因为鼠标坐标只不过就是 *x*、*y* 轴上的值而已，所以你可以将该功能演变

为强制一个物体围绕一个特定的点旋转，比如，另一个物体或屏幕的中央或边缘。

让我们来看一个示例。你可以按照书中的指示一步步做下去，也可以直接打开 01-rotate-to-mouse.html 文件以及 arrow.js 文件，从而直接获得相关的代码。这些文件可以在 www.apress.com 网站中与本书相关的页面找到，同时你也可以找到书中的其余示例代码。

首先，需要一个可供旋转的对象。该对象包含使用 canvas 上下文在 canvas 上绘制一个箭头的功能。由于该箭头对象会重复使用，我们将其封装为一个类，这样每个实例就可以拥有各自不同的属性值。将以下 JavaScript 代码添加到 arrow.js 文件中，把该文件导入 HTML 主文件中：

```
function Arrow () {
  this.x = 0;
  this.y = 0;
  this.color = "#ffff00";
  this.rotation = 0;
}

Arrow.prototype.draw = function (context) {
  context.save();
  context.translate(this.x, this.y);
  context.rotate(this.rotation);
  context.lineWidth = 2;
  context.fillStyle = this.color;
  context.beginPath();
  context.moveTo(-50, -25);
  context.lineTo(0, -25);
  context.lineTo(0, -50);
  context.lineTo(50, 0);
  context.lineTo(0, 50);
  context.lineTo(0, 25);
  context.lineTo(-50, 25);
  context.lineTo(-50, -25);
  context.closePath();
  context.fill();
  context.stroke();
  context.restore();
};
```

draw（context）方法根据传入的 canvas 上下文参数使用 canvas 绘图 API（第 4 章会有详细介绍）绘制一个箭头。这样，每当需要一个箭头对象时，只须调用 new Arrow()方法即可。你可以在图 3-15 中看到这段代码的执行结果。当你绘制任何图像并使其如此旋转时，请确保它指向了正确的或者正向的 x 轴坐标，因为此时的图像就是它旋转 0° 时的表象。

创建了 Arrow 类的一个实例，将其放置在 canvas 元素的正中央并使其指向鼠标所在的位置（如图 3-16 所示）。

看上去很眼熟？没错，之前已经围绕这个三角形讨论了很多内容了，这里只是把箭头的坐标和鼠标的坐标引入了进来。当导入 utils.js 文件并通过 mouse = utils.captureMouse（canvas）捕获到鼠标位置后，就可以访问 mouse.x 与 mouse.y 属性从而获得它的坐标值。而另一方面，可以通过箭头对象的 x 与 y 属性获得它的坐标位置。通过这两个坐标的差值，就可以计算出三角形两边的长度。此时，只须通过 Math.atan2（dy, dx）方法算出角度的大小并将其赋值给箭头

对象的 rotation 属性。该过程如下所示：

```
var dx = mouse.x - arrow.x,
    dy = mouse.y - arrow.y;
arrow.rotation = Math.atan2(dy, dx); //in radians
```

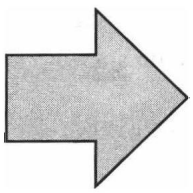


图 3-15 使用 canvas 绘图 API 绘制的箭头

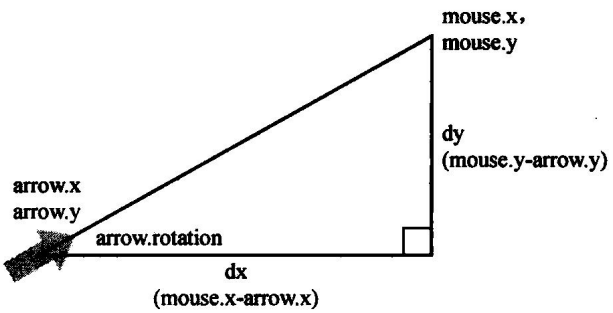


图 3-16 将要计算的内容

当然，需要设置一个循环才能实现动画。正如第 2 章所介绍的，使用 window.requestAnimationFrame 函数在设定好的间隔时间内不断清空 canvas 并绘制新的帧。以下是 01-rotate-to-mouse.html 中的完整的 HTML 文件：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Rotate to Mouse</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="arrow.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        arrow = new Arrow();

    arrow.x = canvas.width / 2;
    arrow.y = canvas.height / 2;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var dx = mouse.x - arrow.x,
          dy = mouse.y - arrow.y;

      arrow.rotation = Math.atan2(dy, dx); //radians
      arrow.draw(context);
```

```
    }());
  };
  </script>
 </body>
</html>
```

请确保 arrow.js 文件与 01-rotate-to-mouse.html 文件处在同一目录下，并使用你最喜欢的、支持 HTML5 的 Web 浏览器运行该示例。如果一切正常，当在 canvas 元素上移动鼠标时，你会发现箭头始终指向它！

现在，假设不存在 Math.atan2 函数。可以将 dy 除以 dx 获得对边与临边的比率并将其传入 Math.atan 函数。仅将上述代码中的 Math.atan2 像下面这样替换成 Math.atan：

```
var radians = Math.atan(dy / dx);
```

试着再次执行示例，你很快就会发现问题。当鼠标移动到箭头的左边时，箭头将不再指向它，而是指向相反的方向。你能搞明白怎么回事吗？让我们回到那张展示了三角形 *A*、*B*、*C* 与 *D* 的图（图 3-13），还记得三角形 *A* 与 *C* 拥有同样的对边与临边比率，三角形 *B* 和 *D* 同样也是。由于 JavaScript 代码无法区分它们，因此它就简单地返回 *A* 和 *B*。这样，如果鼠标处于 *D* 象限，你却会得到 *B* 象限的角度值并将鼠标旋转至那个区域。

现在你应该可以理解 Math.atan2 的意义了，并会将其大量地运用到本书剩余的示例中。

3.4 波

让我们再来看三角学在 canvas 元素中一些更具体的应用。你可能之前听说过正弦波，也可能看过如图 3-17 所示的形状，该形状正是正弦波的图形表示方式：

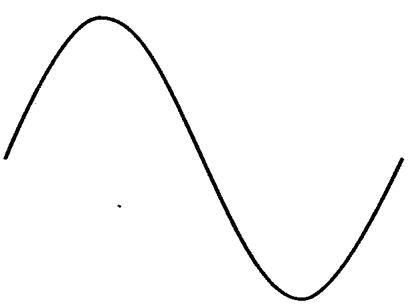


图 3-17 一个正弦波

不过究竟该形状与正弦波有怎样的关联呢？它实际上是正弦函数的结果所对应的图像，此时的输入参数涵盖了从 0° 到 360°（或 0～2π弧度）的所有值。从左到右是函数所接受的角度值（以弧度为单位），*y* 轴则对应了这些角度的正弦值。在图 3-18 中，标注了一些特殊值。

从中你可以看出 0° 的正弦值也是 0，90° 或π/2 弧度的正弦值是 1，180° 或π弧度的正弦值又变回 0，270° 或 3π/2 弧度的正弦值为−1，360° 或 2π弧度的正弦值再次变回 0。在调试控制台中输入以下代码，从而让我们进一步了解正弦波在 JavaScript 中的应用：

```
for (var angle = 0; angle < Math.PI * 2; angle += 0.1) {
  console.log(Math.sin(angle));
}
```

从此以后，我们会渐渐遗忘角度的概念，而更多地使用弧度，除非由于旋转或其他原因确实需要用到它。

在本例中，角度值从 0 以 0.1 为步长渐增，直至它超过 Math.PI * 2，在此过程中同时输出

该角度的正弦值。在结果集中，可以看出正弦值从 0 开始逐渐增长至接近 1，随后下降到接近-1，最后又回到 0 点附近。不过在此过程中并没有出现整数的 1 或 0，因为以 0.1 为步长是不会出现π或者π/2 的整数倍的。

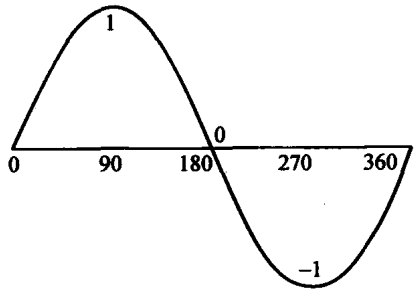


图 3-18 正弦值

3.4.1 平滑的上下运动

那么可以用 Math.sin（angle）做些什么呢？是否曾需要将某些物体平滑地上下或前后移动吗？这时候它就可以发挥作用了。考虑以下场景：通过不断地增加角的度数可以模拟实现从 0 变到 1 再变到-1 最后回到 0 的效果。通过不断增加角的度数可以不断得到上下波动的波形。此时如果将正弦值乘以更大的数字，比如 100，就可以得到一系列从-100 到 100 不断变化的数值。

下一个示例用到了一个根据 Ball 类定义的新对象。ball.js 文件如下所示：

```
function Ball (radius, color) {
  if (radius === undefined) { radius = 40; }
  if (color === undefined) { color = "#ff0000"; }
  this.x = 0;
  this.y = 0;
  this.radius = radius;
  this.rotation = 0;
  this.scaleX = 1;
  this.scaleY = 1;
  this.color = utils.parseColor(color);
  this.lineWidth = 1;
}

Ball.prototype.draw = function (context) {
   context.save();
   context.translate(this.x, this.y);
   context.rotate(this.rotation);
   context.scale(this.scaleX, this.scaleY);
   context.lineWidth = this.lineWidth;
   context.fillStyle = this.color;
   context.beginPath();
   //x, y, radius, start_angle, end_angle, anti-clockwise
   context.arc(0, 0, this.radius, 0, (Math.PI * 2), true);
   context.closePath();
   context.fill();
   if (this.lineWidth > 0) {
       context.stroke();
   }
   context.restore();
};
```

该对象在 draw 方法中利用传入的 canvas 上下文参数绘制了一个圆形。当创建 ball 对象时，可以根据需要指定半径和颜色。如果没有提供初始值，该方法会使用 40 作为圆的默认半径并以红色作为默认颜色。这个类很简单，却很有用，你会在本书剩下的章节中多次看到它，所以

请把它保存在可以随时访问到的地方。

该示例创建 Ball 类的一个实例，并将其添加到 canvas 上，随后通过 drawFrame 函数与 window.requestAnimationFrame 函数设置一个动画循环使得球体上下运动，代码如 02-bobbing-1.html 文件所示：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Bobbing 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        angle = 0;

    ball.x = canvas.width / 2;
    ball.y = canvas.height / 2;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      ball.y = canvas.height / 2 + Math.sin(angle) * 50;
      angle += 0.1;
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

首先，需要创建一个 angle 属性并将其初始化为 0。在 drawFrame 函数中，将 angle 属性的正弦值乘以 50，这些乘积处在-50 到 50 的范围内。如果将该值加上 canvas 高度的一半，就会得到介于 150～250 的数值（以一个高度为 400 像素的 canvas 元素作为参考标准）。以此作为球形的 y 坐标，并在循环中以 0.1 弧度为步长不断增加 angle 属性的大小。这样，就可以获得一个平滑的上下运动。

接下来让我们看看替换不同的参数值又会出现怎样不同的效果。你会发现将 0.1 改成其他值会影响运动的速度。这很容易理解，因为角度增加的快慢，会影响到它的正弦值从 1 变到-1 的速度。而改变 50 这个值则会影响到球形运动范围的远近。canvas.height/2 决定了球形运动的中心点。终上所述，可以根据这些值抽象出以下变量（仅显示对脚本的增加和修改部分）：

```
// At the top of the script:
var angle = 0,
    centerY = 200,
    range = 50,
    speed = 0.05;

// And the animation function:
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  ball.y = centerY / 2 + Math.sin(angle) * range;
  angle += speed;
  ball.draw(context);
}());
```

避免在动画代码中出现具体数字是一个很好的习惯，应该尽可能朝着这个方向努力。在这例子中，变量与具体数字出现在相近的位置。不过试想一下，如果代码散布在多个页面，并且这些值出现在多处，又该怎么办？当每次想修改速度的时候，都需要找到每个 0.1 所在的位置并修改它。可以使用搜索替换功能，不过如果文件中恰好有其他值也定义为 0.1，那么它们也会被无意地覆盖掉。将数字从代码中剥离出来，首选放在文件的开头，你就很容易找到所有的变量。

3.4.2 线性垂直运动

在 04-wave-1.html 示例中，加入了线性垂直运动，希望这可以给你的动画带来一些启发。下面是该文件包含的代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Wave 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        angle = 0,
        centerY = 200,
        range = 50,
        xspeed = 1,
        yspeed = 0.05;
    ball.x = 0;

    (function drawFrame () {
```

```
        window.requestAnimationFrame(drawFrame, canvas);
        context.clearRect(0, 0, canvas.width, canvas.height);

        ball.x += xspeed;
        ball.y = centerY / 2 + Math.sin(angle) * range;
        ball.draw(context);
      }());
    };
    </script>
  </body>
</html>
```

3.4.3 脉冲运动

请记住一件重要的事情，除了用于改变对象的坐标，正弦值还可以应用于对象的其他属性。在 05-pulse.html 示例中，使用正弦值改变球形的比例，从而制造出脉冲的效果。代码如下：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Pulse</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        angle = 0,
        centerScale = 1,
        range = 0.5,
        speed = 0.05;

    ball.x = canvas.width / 2;
    ball.y = canvas.height / 2;

    (function drawFrame () {
       window.requestAnimationFrame(drawFrame, canvas);
       context.clearRect(0, 0, canvas.width, canvas.height);

       ball.scaleX = ball.scaleY = centerScale + Math.sin(angle) * range;
       angle += speed;
       ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

原理跟之前的示例是一样的。这里出现了 centerScale（其比例为 100%）变量、range 变量

和 speed 变量。不过千万不要因为这些属性限制了你的思维，你还可以进一步将正弦波应用于各种不同的属性上从而创建出各种有趣的视觉效果。

3.4.4 使用两个角的产生波

让我们从另一个想法开始新的旅程：我们从单个角发展为两个角，并且为它们设置不同的中心点和速度。然后将其中一个角的正弦波应用于某个属性，再将另一个角的正弦波应用于另一个属性，比如，球形的位置和比例属性。

在以下示例中用到了两个角，其中一个角的正弦波用于影响球形的 x 轴坐标，另一个用于改变球形的 y 轴坐标。它造成的结果就像一只虫子在房间里飞来飞去，尽管根据数学原理可以预测出它的飞行轨迹，但是它看上去还是很随机的。代码如文件 06-random.html 所示：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Random</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        angleX = 0,
        angleY = 0,
        range = 50,
        centerX = canvas.width / 2,
        centerY = canvas.height / 2,
        xspeed = 0.07,
        yspeed = 0.11;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      ball.x = centerX + Math.sin(angleX) * range;
      ball.y = centerY + Math.sin(angleY) * range;
      angleX += xspeed;
      angleY += yspeed;
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

3.4.5 使用绘图 API 产生的波

在接下来的 07-wave-2.html 示例中，直接使用 canvas 的绘图 API 绘制正弦波而不再通过 ball 对象。在 drawFrame 函数中取消了对 context.clearRect 的调用，这样 canvas 元素就不会在每一帧开始的时候擦除之前绘制的图像，使得每一帧绘制的图像都会保留在 canvas 上。文件中的代码如下：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Wave 2</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        angle = 0,
        range = 50,
        centerY = canvas.height / 2,
        xspeed = 1,
        yspeed = 0.05,
        xpos = 0,
        ypos = centerY;

    context.lineWidth = 2;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);

      context.beginPath();
      context.moveTo(xpos, ypos);
      //Calculate the new position.
      xpos += xspeed;
      angle += yspeed;
      ypos = centerY + Math.sin(angle) * range;
      context.lineTo(xpos, ypos);
      context.stroke();
    }());
  };
  </script>
 </body>
</html>
```

canvas 绘图 API 在第 4 章中会给予详细介绍，此时你可以尽情体验使用它所能绘制的各种波形。你可以注意到这里的正弦波是上下颠倒的，因为 canvas 元素中的 y 坐标轴与正常坐标系

中的是相反的。

3.5 圆与椭圆

既然你已经熟练掌握了正弦波，那么让我们来看看他的兄弟余弦波吧。它的产生方式与正弦波是一样的，不过它是由余弦函数产生的而不是正弦函数。如果你能回忆起之前关于正弦余弦的讨论结果，就不会惊讶于它们在不同的 x 坐标上产生了同样形状的波形，因为它们的结果在某个方面是相互颠倒的。图 3-19 展示了一个余弦波。

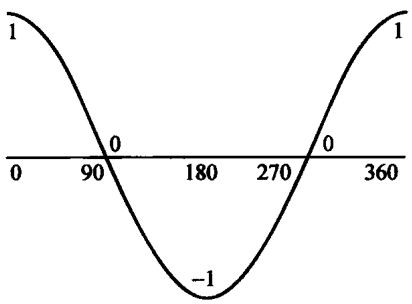


图 3-19 余弦波

如图 3-19 所示，0 弧度的余弦值是 1。在角的弧度变为 2π（即 360°）的过程中，对应的余弦值依次变为 0、−1、0，最终又变回 1。这里的波形本质上与正弦波的波形是一模一样的，只是在 x 轴上产生了一些偏移而已。

3.5.1 圆周运动

如果你只需要获得一个振荡运动，那么在所有情况下余弦和正弦是可以互换使用的。不过当余弦配合正弦使用时会获得一个更常用并更有价值的功能：让物体做圆周运动。图 3-20 展示了物体在做圆周运动时所处的多个位置。

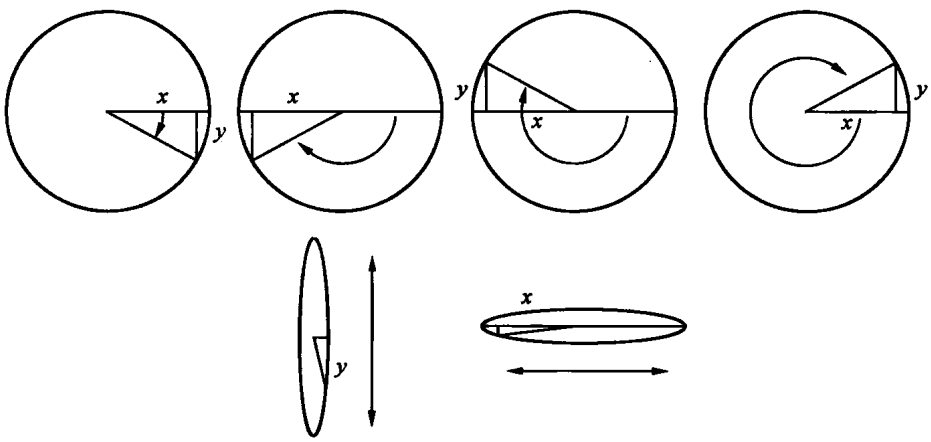


图 3-20 物体在做圆周运动时的不同位置

如果你从右侧面观察图 3-20 的圆形，你会发现当物体在做圆周运动时实际是在做上下运动。上下运动的中心点是圆心所在的位置，而上下运动的范围则恰好是圆的半径。物体在运动中所处的位置为物体所处角度的正弦值与圆的半径的乘积。此时之所以使用正弦函数是因为我

们要计算的是三角形的对边 y，请回想之前示例中的直角三角形。

现在，再想象你从圆的底部进行观察。在这个视角中，你则会发现物体在做前后运动。此时，在计算物体位置时应该计算三角形邻边的长度 x，因此使用余弦函数。

不同于之前看到的 06-random.html 示例中使用两个不同角度计算 x、y 的坐标位置，这里的计算都是基于同一个角的。这里，正弦函数用于计算 y 坐标，而余弦函数则用于计算 x 坐标。以下是示例代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Circle</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        angle = 0,
        centerX = canvas.width / 2,
        centerY = canvas.height / 2,
        radius = 50,
        speed = 0.05;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      ball.x = centerX + Math.sin(angle) * radius;
      ball.y = centerY + Math.cos(angle) * radius;
      angle += speed;
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

可以用以上代码自己创建文件或者直接使用 08-circle.html 示例文件，该文件包含以上所有代码。在浏览器中打开该文件以验证你可以获得一个完美的圆形。

注意，在两个视角中圆周运动的范围都在三角形斜边的长度内，它也等同于圆的半径，因此将 range 变量改名为 radius 以体现这一概念。

以上代码使用余弦函数获得 x 坐标的大小，又通过正弦函数取得 y 坐标的大小。请牢记这一关系。当我们从事动画编程时，每当我们谈到 x 时，你应该立即想到余弦，而当我们谈到 y

时，就应该马上想到正弦。请花费尽可能多的时间以便你完全理解最后一段代码，因为它将成为你的动画工具箱中最有用的一个工具。

3.5.2 椭圆运动

尽管圆形很好看，不过有些时候你并不想要一个完美的圆形，某些时候你可能需要一个椭圆。上一示例的问题在于当使用半径时，上下沿（y 轴）运动和前后沿（x 轴）运动的范围是一致的，正是在这种情况下我们得到了一个完美的圆形。

为了获得一个椭圆，可以使用不同的半径用于计算 x 与 y 的坐标位置，让我们将其分别命名为 radiusX 与 radiusY。下面是将它们应用到文件 09-oval.html 中的代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Oval</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        angle = 0,
        centerX = canvas.width / 2,
        centerY = canvas.height / 2,
        radiusX = 150,
        radiusY = 100,
        speed = 0.05;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      ball.x = centerX + Math.sin(angle) * radiusX;
      ball.y = centerY + Math.cos(angle) * radiusY;
      angle += speed;
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

此时，radiusX 为 150，这意味着，球形在做圆周运动时为以 centerX 为中心在前后 150 个像素的范围内运动。radiusY 则设置为 100，这意味着，球形会在上下 100 个像素内做上下运动。

这样你就获得了一个不完美的圆形，其实它已经不再是圆形而成为椭圆。

3.6 勾股定律

终于，我们开始介绍勾股定律，另一个会频繁使用的公式。勾股定律在西方是由一位名为毕达哥拉斯的希腊数学家、哲学家提出的，他因为提出了一种计算三角形边长的方便方法而被载入史册。这一方法简而言之就是：A 的平方+B 的平方=C 的平方。下面我们会更加深入地介绍这一定律。

以下是对上一定律一个更具描述性的解释：直角三角形的两条直角边的平方和等于斜边的平方。假设有图 3-21 所示的三角形，两条直角边 A 与 B 的长度分别为 3 和 4，斜边的长度为 5。勾股定律告诉我们 $A^2+B^2=C^2$。让我们验证一下，代入具体数字，我们得到 $3^2+4^2=5^2$，即 9+16=25，公式成立。

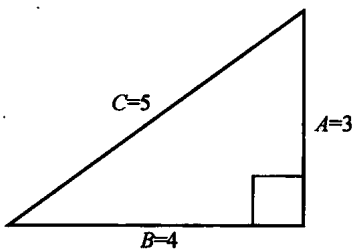


图 3-21 一个直角三角形

如果你恰好知道三条边的长度，那么勾股定律没什么太大作用，它只是反映了一个有趣的关系。但是如果你只知道两条边的长度，那么它就成为一个能够快速计算出第三条边的长度的强大工具。在动画编程中，在已知两条直角边的长度的情况下，计算斜边长度是一个非常常见的应用场景。具体来说，就是你想知道两点间的距离。

3.6.1 两点间距离

假设在 canvas 元素上有两个物体，你想知道它们相距多远。这是在动画编程中最常用到勾股定律的场景。

这里的已知条件是什么？此时我们能够获得每个物体的（x，y）坐标。假设其中一个物体的坐标为（$x1$，$y1$），另一个为（$x2$，$y2$），就像图 3-22 所示一样。

如果你沿着本章的内容一路阅读，你会在图 3-22 中看到一个直角三角形。而两个物体间的连线正是三角形的斜边。在图 3-23 中，我们将三角形补充完整并加入一个具体数字。

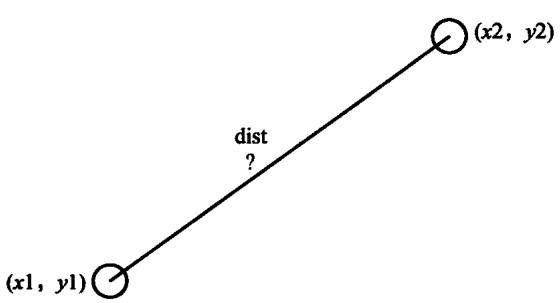


图 3-22 两个物体间的距离为多少

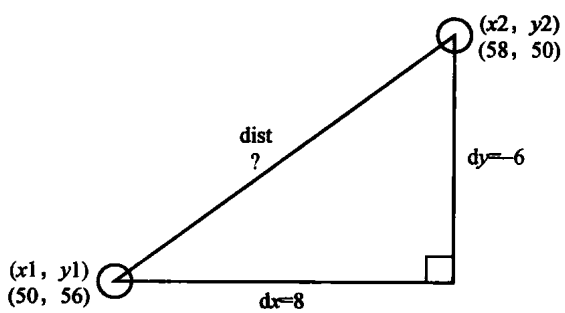


图 3-23 将两个物体间的连线转化为直角三角形

这里，dx 表示两个物体在 x 轴上的距离，而 dy 则为它们在 y 轴上的距离。dx 可以由 $x1$ 减去 $x2$ 计算得出，即 58−50=8，类似地，可以由 $y1$ 减去 $y2$ 获得 dy 的值−6。此时，根据勾股定律，将两个值取平方后相加即可得到两个物体间的距离的平方，即$-6^2+8^2=dist^2$。进一步计算得到 36+64 即$100=dist^2$。根据代数学基础知识，可以推导出$\sqrt{100}=dist$。由此你可以很容易得出两物体间距离为 10。

让我们尝试对上述计算过程进行抽象，以获得一个适用于任意场景的公式。给定两点（$x1$，$y1$）与（$x2$，$y2$），分别计算出 x 轴与 y 轴上的距离，取其平方，再求和，最后取平方根。以下是对应的 JavaScript 代码：

```
var dx = x2 - x1,
    dy = y2 - y1,
    dist = Math.sqrt(dx * dx + dy * dy);
```

请务必留意以上代码，因为它们为你的工具箱贡献了另一个重要工具。前两行代码获得了两点在 x、y 轴上的距离。如果你只关注于两点间的距离而不会用 dx 与 dy 计算任何角度的话，那么你用 $x1$ 减去 $x2$ 抑或 $x2$ 减去 $x1$ 都不会对结果造成什么影响，因为最终结果总是一个正值。最后一行代码集中了三个计算步骤：求两个距离的平方，再求和，最后计算平方根。你也可以将这些步骤分解为独立的语句以便于理解。当你熟悉这行代码后，就很容易理解它了。下次当你看到它就会联想到勾股定律。

让我们在一个现实例子中实验一下。下一个示例 10-distance.html 创建了若干矩形，再将它们随机排列，然后计算并输出它们之间的距离。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Distance</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <textarea id="log"></textarea>
  <script src="utils.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        log = document.getElementById('log');

    //Create a black square, assign random position.
    var rect1 = {
      x: Math.random() * canvas.width,
      y: Math.random() * canvas.height
    };
    context.fillStyle = "#000000";
    context.fillRect(rect1.x - 2, rect1.y - 2, 4, 4);

    //Create a red square, assign random position.
    var rect2 = {
      x: Math.random() * canvas.width,
```

```
      y: Math.random() * canvas.height
    };
    context.fillStyle = "#ff0000";
    context.fillRect(rect2.x - 2, rect2.y - 2, 4, 4);

    //Calculate the distance between the two squares.
    var dx = rect1.x - rect2.x,
        dy = rect1.y - rect2.y,
        dist = Math.sqrt(dx * dx + dy * dy);

    //log output of distance value to screen
    log.value = "distance: " + dist;
  };
  </script>
 </body>
</html>
```

在Web浏览器中运行该文件，两个矩形间的距离会显示在该文件的textarea元素中。每次运行该文件，两个矩形所处的位置的总会发生变化。而无论两个矩形的相对位置如何，上、下、左或右，它们的距离永远都是一个正值。

上面这个示例虽然很有趣，不过还不够动态。下面这个示例11-mouse-distanc.html则证明了我们可以实时地计算出两个矩形的距离：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Mouse Distance</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <textarea id="log"></textarea>
  <script src="utils.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        log = document.getElementById('log'),
        rect = {x: canvas.width / 2, y: canvas.height / 2};

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var dx = rect.x - mouse.x,
          dy = rect.y - mouse.y,
          dist = Math.sqrt(dx * dx + dy * dy);

      //draw square
      context.fillStyle = "#000000";
      context.fillRect(rect.x - 2, rect.y - 2, 4, 4);
```

```
        //draw line
        context.beginPath();
        context.moveTo(rect.x, rect.y);
        context.lineTo(mouse.x, mouse.y);
        context.closePath();
        context.stroke();

        //log output of distance value to screen
        log.value = "distance: " + dist;
      }());
    };
    </script>
  </body>
</html>
```

这里 dx 与 dy 由当前鼠标位置与矩形所在位置相减获得。我们将 dist 变量的值显示到 textarea 元素上，并在矩形和鼠标所在位置之间绘制了一条连线（第 4 章会详细介绍 canvas 绘图 API）。

测试以上文件并移动鼠标，你会发现总是存在一条线连接了矩形与鼠标，并且连接线的长度总是实时地显示在 textarea 元素中。

在后续章节介绍冲突检测时，我们会提到通用测试函数的一些缺陷，并会介绍根据勾股定律发展出来的一种基于距离的冲突检测方法。该定律也会用于计算力（如重力或弹力）的大小，因为两个物体间的引力跟它们之间的距离也存在一定的正比关系。

3.7 本章中的重要公式

让我们来看看手头上这个崭新耀眼的开发工具箱，其中已经包含了超过 6 个工具。所有的工具函数可以在附录 A 中找到，不过让我们先看看目前添加的这几个。我会尽可能保持这些公式的简洁性和并提高它们的抽象程度，这样它们就无须包含任何变量声明，可以完全由开发者根据具体的场景决定如何使用合适的语法将这些公式应用到自己的脚本中。

3.7.1 三角学基础函数

```
sine of angle = opposite / hypotenuse
cosine of angle = adjacent / hypotenuse
tangent of angle = opposite / adjacent
```

3.7.2 角度与弧度互转

```
radians = degrees * Math.PI / 180
degrees = radians * 180 / Math.PI
```

3.7.3 朝鼠标（或任意一点）旋转

```
//substitute mouse.x, mouse.y with the x, y point to rotate to
```

```
dx = mouse.x - object.x;
dy = mouse.y - object.y;
object.rotation = Math.atan2(dy, dx) * 180 / Math.PI;
```

3.7.4 创建波

```
// assign value to x, y or other property of an object,
// use as drawing coordinates, etc.
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);

  value = center + Math.sin(angle) * range;
  angle += speed;
}());
```

3.7.5 创建圆形

```
//assign position to x and y of object or drawing coordinate
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);

  xposition = centerX + Math.cos(angle) * radius;
  yposition = centerY + Math.sin(angle) * radius;
  angle += speed;
}());
```

3.7.6 创建椭圆形

```
//assign position to x and y of object or drawing coordinate
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);

  xposition = centerX + Math.cos(angle) * radiusX;
  yposition = centerY + Math.sin(angle) * radiusY;
  angle += speed;
}());
```

3.7.7 获取两点间的距离

```
//points are x1, y1 and x2, y2
//can be object positions, mouse coordinates, etc.
dx = x2 - x1;
dy = y2 - y1;
dist = Math.sqrt(dx * dx + dy * dy);
```

当然，本书的内容不仅仅是一堆公式。将这些公式浏览一遍并确保你完全理解了每个公式的工作原理。如果你有任何疑问，请翻回到介绍这些公式的对应章节，尝试运行示例并根据需要进一步研究它们，直到你可以看到名字就能想起它们的具体内容。

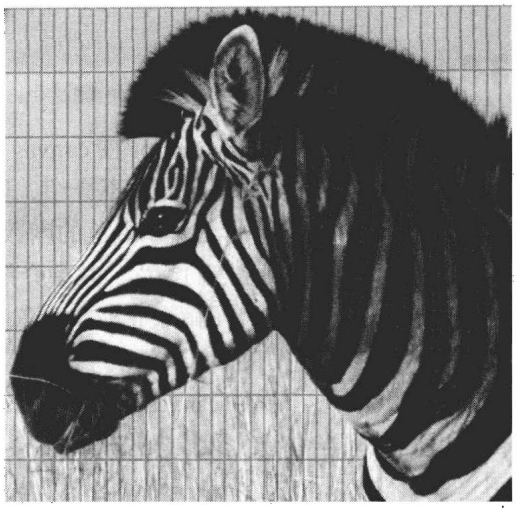
3.8 小结

本章涵盖了使用 JavaScript 在 canvas 元素上实现动画所需的几乎所有的三角学知识。还有一个定理（余弦定理），目前暂不介绍，因为它相对而言更加复杂一些，并且它还适用于非直角三角形（即三个角中没有一个角的度数为 90°）。如果你现在已经对三角学着迷并且还意犹未尽，你可以直接跳到第 14 章，其中介绍的反向运动学在三角学中发挥了重要作用。

不过，现在你已经了解了正弦、余弦与正切以及它们的逆函数：反正弦、反余弦与反正切，还包括用于计算它们的 JavaScript 函数。

最重要的是，你在 canvas 元素上亲手实践了以上大部分函数，并且是在一些最常见的实际例子做测试。当你继续阅读本书时，你会发现更多能够发挥它们作用的用法。而现在我们为学习这些概念打下了扎实的基础，这对于我们以后看懂用到它们的示例是非常有帮助的，你将轻而易举地看懂那些示例并理解其原理。

第 4 章将介绍一些常见的渲染技术，该技术用于将图像呈现到屏幕上，其中还会介绍所有重要的 Canvas 绘图 API。在你学习第 4 章时，请尝试寻找能够利用那些渲染技术将本章中三角学函数曲线呈现到屏幕上的方法。对此，我们应该不存在任何问题。



第 4 章　渲染技术

本章涵盖以下内容：

- canvas 上的颜色
- 绘图 API
- 图片加载
- 像素处理

目前为止，在本书示例程序中绘制的图像仅用到了一些简单的绘图命令，而 canvas 绘图 API 也多次被提及。不过在大多数的示例代码中并没有对绘图命令做过多解释。

本章将介绍如何使用 canvas 元素创建可视内容。具体而言，你会在本章中看到如何使用 JavaScript 与颜色打交道以及关于 canvas 绘图 API 的一段简介。

4.1 canvas 上的颜色

我们使用 CSS 风格的字符串指定应用到 canvas 元素上的颜色，该字符串包含以下格式。

- "#RRGGBB"——十六进制格式，红绿蓝三原色的范围为 0～9、a～f。
- "#RGB"——简写的十六进制格式，转化成 6 位数字时重复三原色的值，而不是追加 0。例如，"#fb0"会转化为"#ffbb00"。
- "rgb(R, G, B)" ——函数表达式，三原色分别由 0～255 的整数值表示。
- "rgba(R, G, B, A)" ——包含透明度的函数表达式，与上一表达式类似，其中 alpha 参数由 0～1 的小数构成。需要指定透明度的颜色必须使用该格式。

尽管颜色值最终需要由一个字符串表示，我们还是可能希望通过数字操作红绿蓝每个合成色的值。因此我们必须使用一些字符串解析与拼接技术完成 JavaScript 类型的转换。在展示完该技术后，本章稍后介绍一对工具函数以简化该过程，而这对函数也会应用到本书后续的示例中。

一个 24 位的颜色值可以经由 0～16 777 215 之间的任意一个整数表示。之所以会有 16 777 215 这个数字是因为存在 256 × 256 × 256 个可能的组合。canvas 元素使用 RGB 颜色，这表明每种颜色都是由红绿蓝三原色构成的，而三原色中的每个合成色都可以经由 0～255 的一个整数表示，又因为红绿蓝每个合成色都存在 256 个可能的值，它们的组合就构成了接近 1.68 千万种颜色。

由于 256 由 8 位二进制数字（0 或 1）表示，并且存在红绿蓝 3 种合成色，8 乘以 3 等于 24，也就是说，需要用 24 位数字表示 1.68 千万种可能的颜色，所以将这一系统称为 24 位颜色系统。此外，还有一个 32 位的颜色系统，其中用一个额外的 8 位数字表示颜色的透明度。

由于我们很难联想到数字 16 733 683 所代表的颜色，开发者会经常使用另一种颜色格式——十六进制格式表示颜色。如果你曾经在 HTML 中使用过颜色值，你应该对这一格式很熟悉，不过还是让我们从基础开始讲起。

4.1.1 使用十六进制表示颜色值

十六进制是一个以 16 作为基数的系统。不同于常见的十进制系统中使用 0～9 的整数表示每一位数字，十六进制中的每一位数字可以来自于 0～15 的任意整数。因为不存在可以表示 10～15 的一位数字，所以我们借用了字母表中的前 6 个字母 A～F。因此，十六进制数字中每一位数字可以经由 0～F 的任意值表示（在 JavaScript 中，十六进制数字是不区分大小写的，因此可以使用 A～F 或 a～f 中的任意字母）。为了强调在 HTML 的 canvas 元素中使用十六进制数，我们会在十六进制字符串的开头加上'#'号作为前缀。跟其他许多语言类似，在 JavaScript 中会为十六进制数加上 0x 作为前缀。例如，0xA 等于十进制数字 10，0xF 等于 15，0x10 则等于 16。对应以上数字，HTML 中的十六进制数字字符串表示分别为：'#A'、'#F'与'#10'。

在十进制中，每个数字都是其右边数字的 10 倍。所以，数字 243 代表了 2 乘以 100、4 乘以 10 以及 3 乘以 1。而在十六进制中，每个数字则是右边相邻数字的 16 倍。例如，0x2b3 表示 2 乘以 256、B（即 11）乘以 16 以及 3 乘以 1。

在 24 位颜色值中，这一数字可以一直增长到 0xF～FFFF，经过计算你会发现它等于 16 777 215。除此之外，这 6 个十六进制数还可以分为三组，分别代表三原色。第一组数字代表红色，第二组代表绿色，最后两个数字则代表蓝色。这一点通常引用为 0xRR GGBB。（R、G 或 B 绝不会作为真实的十六进制数被填入上一表达式，该表达式只不过是一种符号化的表达方式，用以告知每个数字控制哪些颜色通道。）

请记住，每个合成色的值可以是 0～255 的任意数字，即十六进制中 x00～0xFF 的任意数字。因此红色可以由 0xFF 0000 表示，它代表纯红色，不夹杂一丝绿色及蓝色。同样，0x00 00FF

表示纯蓝色。

假如将之前提到的示例数字 16 733 683 转化为十六进制数，可以得到 0xFF 55F3。此时可以方便地获得三原色各个颜色的值。红色为 FF，绿色为 55，蓝色为 F3。观察这些数字，可以发现红色和蓝色的值偏大而绿色的值则偏小。这样你就可以大概猜到该颜色偏紫色，而从十进制值中你就很难推断出这一信息了。

对于所有需要用到数字的 JavaScript 操作而言，是不会在意数字是十六进制还是十进制的。对这些操作来说，数字 16 733 683 与 0xFF 55F3 的值是相同的，只不过后者更利于我们人类的理解。不过，在使用 canvas 元素时，你必须牢记将任意数字格式转化为一个正确的 CSS 风格的字符串。

你可能想知道如何在这些数字格式间做转换。从十六进制转换到十进制非常简单，只要输出该值即可。console.log 函数可以将十六进制数转换为十进制数并输出到调试控制台：

```
console.log(0xFF55F3);
```

输出结果为 16 733 683。使用 Number.toString 方法，并指定 16 作为基数，即可把十进制数转换为十六进制数。

```
var hex = (16733683).toString(16);
console.log(hex);
```

正如方法名所暗示的，该方法将一个数字转换为一个字符串并输出十六进制的值'ff55f3'。再加上'#'号作为前缀，就可使其成为一个符合 CSS 风格的字符串。

```
var hexColor = '#' + (16733683).toString(16);
```

这里的输出结果为'#ff55f3'。为了将其还原为十进制数，首先利用 String.slice 方法去除'#'号前缀，再将剩余字符串传入内置 window.parseInt 函数并以 16 作为基数：

```
var color = window.parseInt(hexColor.slice(1), 16);
console.log(color);
```

输出结果为 16 733 683，此时我们又回到了最开始的那个数字。

4.1.2 色彩合成

假设你想通过红绿蓝三原色的 3 个数值合成一个完整有效的颜色值，你可以借助以下公式，其中 red、green 和 blue 分别为代表 3 种颜色的变量，每个变量持有 0～255 之间的一个整数值。

```
color = red << 16 | green << 8 | blue;
```

例如，可以通过将红色值设置为最大，即 0xFF（十进制数 255），绿色值设置为 0x55（十进制数 85）并将蓝色值设置为 0xF3（十进制 243）创建出紫色。

```
var color = 0xFF << 16 | 0x55 << 8 | 0xF3;
```

此时 color 变量的值为 16 733 683，它等价于十六进制数 0xFF 55F3。该公式用到了之前你可能没有遇到过的两个位操作符。位操作符在二进制层面（即 0 和 1）上对数字加以操作。我们会在下一节中解释它的工作原理。

按位组合

如果要列出一个 24 位的颜色值，就会得到一个包含 24 个 0 或 1 的字符串。将十六进制数

0xRR GGBB 分解为二进制数，其中有 8 位是为了表示红色值，8 位是为了表示绿色值，8 位是为了表示蓝色值：

```
RRRRRRRRGGGGGGGGBBBBBBBB
```

在上一个颜色合成公式中，第一个位操作符是<<，这是按位左移操作符，它根据指定的位数将一个数值的二进制格式向左移动若干位置。一个 0xFF 的红色值，即十进制数 255 的二进制格式如下：

```
11111111
```

将其左移 16 位得到以下结果：

```
111111110000000000000000
```

作为一个 24 位的颜色值，它代表红色，并且在十六进制中它（0xFF 0000）仍旧代表红色。然后绿色值 0x55（十进制数 85）的二进制格式如下所示：

```
01010101
```

左移 8 位得到以下结果：

```
000000000101010100000000
```

最初的 8 位数字完全落到了代表绿色值的区间中。

最后，蓝色值 0xF3（十进制数 243）的二进制格式如下：

```
11110011
```

因为这些数字已经落入了代表蓝色值的区间，所以不需要再移动它们了。

此时，我们拥有以下红绿蓝三原色值：

```
111111110000000000000000
000000000101010100000000
000000000000000011110011
```

你可以将这些数字相加得到单个 24 位数字，不过还有一个更简单的方法：使用位“或”操作符，即|符号。该操作会在二进制层面比较两个数字，如果其中一个为 1，那么结果为 1。如果两个数字都为 0，那么结果为 0。可以对红色、绿色与蓝色值一起使用或操作，即“如果 R 或 G 或 B 为 1，则结果为 1”。这会将三原色合并为一个单独的颜色值。如果你明白了或操作符的原理就会很容易理解合并后的结果。合并示例中的三原色值得到以下结果：

```
111111110101010111110011
```

该数值的十六进制格式为 0xFF55F3。当然，在 JavaScript 中，你没有机会接触到位的概念或直接与 0 或 1 打交道。你只会用到下面这个表达式：

```
var color = 0xFF << 16 | 0x55 << 8 | 0xF3;
```

或者它会以如下形式出现，如果你工作在十进制环境下：

```
var color = 255 << 16 | 85 << 8 | 243;
```

再次申明，采用十进制或十六进制对结果是没有影响的，它们的结果是一样的。

4.1.3 提取三原色

你可能想从一种颜色中提取出每个独立的三原色的值。以下分别是从一个组合颜色值中提取出红色、绿色以及蓝色的公式：

```
red = color >> 16 & 0xFF;
green = color >> 8 & 0xFF;
blue = color & 0xFF;
```

如果 color 的值为 0xFF 55F3，那么 red、green 以及 blue 变量的值将分别为 255、85 与 243。这里的公式再次用到了两个位操作符，我们同样会在下一节中给出详细介绍。

逐位提取

在上一个公式中，你可能会猜到>>为按位右移操作符，该操作符会将二进制数右移若干位数。由于不会出现小数位，因此所有右移过多的数字会直接丢弃。再次查看示例颜色的二进制值，从红色开始提取：

```
111111110101010111110011
```

当将颜色值右移 16 位后，该值变为：

```
11111111
```

这就是 0xFF 或 255。针对绿色，只需要将最开始的数字右移 8 位，得到如下结果：

```
1111111101010101
```

此时蓝色已剔除，不过红色也同时保留下来。这时候我们会用到另一个位名为“与”的操作符“&”。与“或”类似，它也用于比较两个二进制值，不过它所表达的意思是：如果两个二进制数都为 1，那么结果为 1；如果其中一个为 0，那么结果为 0。我们将该值与 0xFF 比较，即比较下面两个数字：

```
1111111101010101
0000000011111111
```

由于所有代表红色的二进制位都与 0 做比较，因此它们的结果自然为 0。最后只有那些原来为 1 的二进制位保留下来，所以最终结果为：

```
0000000001010101
```

它的十六进制表示为 0x55。针对蓝色，我们不需要做任何移位操作。只需要将原始值与 0xFF 做“与”运算，其结果会保留蓝色并剔除红色和绿色的二进制值。这返回结果 0xF3。

4.1.4 透明度

如果你想为 canvas 上的颜色设置透明度，就必须使用以下 CSS 风格的函数"rgba（R, G, B, A）"。该函数中红绿蓝三原色均为 0～255 范围内的整数，而 alpha 值却是 0～1 范围内的一个小数。alpha 为 0 表示完全透明，而 1 则表示完全可见。例如，如果一种颜色中的 alpha 值为 0.5 则表示半透明。

在获得每个单独的三原色后，必须将它们的值拼接成特定格式的字符串才能将颜色应用到 canvas 上，例如：

```
var r = 0xFF,
    g = 0x55,
    b = 0xF3,
    a = 0.2,
    color = "rgba(" + r + "," + g + "," + b + "," + a + ")";

console.log(color);
```

以上代码组装并输出字符串："rgba(255,85,243,0.2)"，该值代表一种淡紫色，颜色的透明度

为 80%，该字符串可用于 canvas 元素。

由于这种类型的转换实在有些麻烦，因此在下一节中创建了一些工具函数用于简化这一转换过程。

4.1.5 与颜色相关的工具函数

既然我们已经介绍了合成与抽取三原色的过程，我们就创建一对工具函数来简化这一过程。通过它们就可以非常方便地实现颜色值在数字与字符串之间的相互转换，并保持示例代码的简洁性。通常当在 canvas 元素上绘制动画时，需要使用 JavaScript 操作数值型的颜色值，然后将其格式化为一个 CSS 风格的字符串。

第一个工具函数 utils.colorToRGB 接受一个表示颜色值的数字或十六进制字符串作为输入参数，然后抽取出各个独立的三原色，并将它们拼接成一个 CSS 风格的函数作为返回值。此外，该函数还接受一个可选的第二参数用于指定 alpha 值，该值必须是 0～1 范围内的一个小数。

例如，把十六进制数 0xFFFF00 或字符串"#FFFF00"传入 utils.colorToRGB 将返回以下字符串值"rgb(255,255,0)"。如果在调用该函数时传入一个 alpha 值，比如 utils.colorToRGB(0xFFFF00, 0.5)，它将返回"rgba(255,255,0,0.5)"，而这正是 canvas 元素所要求的格式。

将该函数添加到我们的全局对象 utils 中并保存到 utils.js 文件，会把该文件导入创建的 HTML5 文件中。请记住你可以到 www.apress.com 网站上下载包含该文件在内的所有示例代码。以下是该函数的定义：

```
utils.colorToRGB = function (color, alpha) {
  //if string format, convert to number
  if (typeof color === 'string' && color[0] === '#') {
      color = window.parseInt(color.slice(1), 16);
  }
  alpha = (alpha === undefined) ? 1 : alpha;

  //extract component values
  var r = color >> 16 & 0xff,
       g = color >> 8 & 0xff,
       b = color & 0xff,
       a = (alpha < 0) ? 0 : ((alpha > 1) ? 1 : alpha); //check range

  //use 'rgba' if needed
  if (a === 1) {
     return "rgb("+ r +","+ g +","+ b +")";
  } else {
     return "rgba("+ r +","+ g +","+ b +","+ a +")";
  }
};
```

第二个工具函数比较简单，不过却为实现颜色值在数字和字符串之间的相互转换减少了很大的麻烦。utils.parseColor 函数可以将一个颜色的数字值转换为一个十六进制格式的字符串，也可以将一个十六进制格式的字符串还原为一个数字值。

调用该函数并传入一个十六进制数（如 utils.parseColor(0xFFFF00)），将返回字符串值"#ffff00"。传入一个 CSS 风格的十六进制字符串将原值返回。该函数还接受一个可选的第二参数 toNumber，当该值设置为 true 时，该函数将返回一个数值型的颜色值。例如，当调用 utils.parseColor("#FFFF00", true)或 utils.parseColor（0xFFFF00, true）时，都将返回数字 16 776 960。

有了该函数后，我们就可以随时使用颜色的数字值，并可以方便地将数字值转换为 canvas 元素所支持的正确格式的字符串。该函数也被添加到 utils.js 文件的全局对象 utils 中。

```
utils.parseColor = function (color, toNumber) {
  if (toNumber === true) {
     if (typeof color === 'number') {
        return (color | 0); //chop off decimal
     }
     if (typeof color === 'string' && color[0] === '#') {
       color = color.slice(1);
     }
     return window.parseInt(color, 16);
  } else {
     if (typeof color === 'number') {
       //make sure our hexadecimal number is padded out
       color = '#' + ('00000' + (color | 0).toString(16)).substr(-6);
     }
     return color;
  }
};
```

由于在前几节中你已经掌握了这些工具函数的工作原理，因此这里仅仅是为了便于以后的程序开发而引入它们。或许你会创建一个不同的函数，抑或是在你自己的代码中显式地完成这些转换，不过我们还是在这里提供这些函数以方便你的使用。

此时，你应该了解了颜色是如何在 canvas 元素中以及 JavaScript 中工作的，让我们开始在动画中使用它吧。

4.2 绘图 API

先介绍 API 是什么，它是应用程序编程接口（Application Programming Interface）的缩写。总的来说，API 是指一系列方法和属性，可以在程序中使用它们访问已实现的相关功能。而 canvas 绘图 API 特指那些用于实现画直线、曲线以及填充某些区域的 JavaScript 方法与属性。这套 API 包含不太多的几个方法，不过它们却是创建本书中各种动画的基础。以下是将要用到的一些方法：

- strokeStyle
- fillStyle
- lineWidth
- save()
- restore()
- beginPath()

- closePath()
- stroke()
- lineTo(x, y)
- moveTo(x, y)
- quadraticCurveTo(cpx, cpy, x, y)
- bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)
- arcTo(x1, y1, x2, y2, radius)
- arc(x, y, radius, startAngle, endAngle [, anticlockwise])
- createLinearGradient(x0, y0, x1, y1)
- createRadialGradient(x0, y0, r0, x1, y1, r1)
- clearRect(x, y, width, height)
- fillRect(x, y, width, height)

本章作为一个概述，主要介绍一些基础知识便于看懂后续章节，因此这里并不会详细介绍每个方法。要深入了解这些 canvas 绘图 API 的话，可以参考位于 http://developers. whatwg.org/the-canvas-element.html 的规范，或阅读本书开篇所列的更具体的参考资料。

4.3 canvas 上下文

每个 canvas 元素都包含一个绘图上下文，通过它可以访问绘图 API。如以下代码所示：

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d');
```

第一行代码使用 DOM 接口通过 id'canvas'查找 canvas 元素。随后调用该对象上的 getContext 方法并传入字符串参数'2d'，该参数用于指定将要使用的绘图 API 的类型。

值得一提的是，在本书编写的时候只有'2d'这么一个标准的绘图上下文。尽管一些厂商已经开始尝试 3D API，但是该 API 并没有被其他浏览器所支持也尚未成为 HTML5 标准的一部分。

现在我们拥有了自己的 canvas 元素并能访问它的上下文，让我们开始使用它绘图吧！

4.4 使用 clearRect 消除图案

在大多数的动画中，必须在绘制下一帧图案前清除 canvas，我们正是通过这一手段模拟出物体正在运动的效果。通过绘制图案，擦除它，再将其绘制到另一个位置，这样就可以把一系列图像模拟成一个物体发生了移动而不仅仅是顺序地播放了两帧图像。

context.clearRect(x, y, width, height)方法会通过将像素的颜色设置为全透明的黑色擦除指定区域内的每一个像素，从而清除指定的矩形区域。这样，之前绘制的任意图像（比如，直线或实心形状）都会被擦除掉。由于需要使用该方法清除整个动画帧，因此它在示例中将会接如下方式调用：

```
context.clearRect(0, 0, canvas.width, canvas.height);
```

这里创建了一个与整个 canvas 相同大小的矩形，并将其放置在 canvas 的原点，从而能够清除之前所绘制的所有图像。

这对于 canvas 而言是一次非常彻底的清除，不过这其中存在一些值得优化的地方。如果你能确定某个动画只影响到帧的一个局部区域，那么可以在 clearRect 中指定一个稍小的矩形进而避免一些处理。另一个技巧是将多个 canvas 元素相互叠加。由于 canvas 是透明的，因此最终看到的会是它们组合而成的图像。如果在动画中有一张不变的背景图片，而你又不想在每一帧中重复绘制它的话，就可以使用这一技巧。可以将这张背景图片作为一个背景 canvas 元素或 image 元素并在其之上用另一个 canvas 元素绘制动画。

4.4.1 设置线条的外观

绘图上下文包含一系列可用于更改随后绘制的线条的外观的属性。更改这些属性并不影响之前绘制的线条。而一旦将线条或实心形状绘制好之后，除了擦除或覆盖之外是没有办法再去修改它们的。

用于控制线条样式的最常用属性是颜色和宽度。

- strokeStyle: 该属性用于指定线条的颜色。该值可以是一个颜色值（CSS 风格的字符串）、一个渐变对象或一个模式对象。线条的颜色默认为黑色（#000000）。
- lineWidth: 该属性用于指定线条在其路径上的宽度。因此当默认值为 1 时，该线条会在其路径的两侧分别延伸出半个像素。无法将线条绘制在其路径的内部或外部。该属性必须为正值。

除此之外，还有一些控制线条样式的额外属性，不过在本书中只会用到它们的默认值。

- lineCap: 线条的终点将如何绘制，它可以是平的、圆形的或者一段延长线。该属性值包含 butt、round 与 square 选项，默认值为 butt。
- lineJoin: 决定两条相连的线段如何接合，或者连接线的弯头部分如何绘制。该属性值包含 round、bevel 或 miter 选项，默认值为 miter。
- miterLimit: 当 lineJoin 属性设置为 miter 时，该属性可用于控制两条相交线外侧交点与内侧交点的距离。它必须是一个大于零的有限数，默认值为 10。

使用 context.save()与 context.restore()方法可以实现在不同的样式间切换。context.save()方法将 canvas 的当前状态存入栈中，其中包含各种样式，例如，strokeStyle、fillStyle 以及应用于 canvas 的变换效果。额外的样式变化会影响到后续的绘制指令。调用 context.restore()可以使得当前的 canvas 状态出栈转而使用下一个状态，也就是之前的状态。

4.5 使用 lineTo 与 moveTo 绘制路径

在图形语言中存在多种绘制线条的不同方式。一种使用 line 命令在通过参数传入的起点和

终点间绘制一条线。另一种使用 lineTo 命令并接受单个点作为参数：线段的终点。canvas 绘图 API 采用的是 lineTo 命令，它会把起点和终点的信息保存在一个路径对象中。

canvas 上下文总是持有一条唯一的当前路径。一条路径可以拥有零条或多条子路径，每条子路径由一系列通过直线或曲线相连的点构成。如果一条路径的终点和起点间首尾相连，这条路径称为闭合路径。调用 context.beginPath()即可表示你想开始绘制一条新的路径。一条路径只不过是构成一条线的一系列坐标位置，为了将它渲染到 canvas 上，需要调用 context.strokes()方法。

你会在 canvas 上沿着路径不断地移动，每次到达一个新的坐标位置就从上一个位置开始画一条线。开始画线前，必须有一个起点，这里以（0,0）坐标点作为起点：

```
context.beginPath();
context.moveTo(0, 0);
context.lineTo(100, 100);
context.stroke();
```

在 canvas 上出现了一条从左上角（0,0）到（100,100）的直线。在画完一条线后，这条线的终点将自动成为下一条线的起点。或者，可以使用 context.moveTo 方法为下一条线指定一个新的起点。

可以将绘图 API 想象成一个握着一支笔在一张纸上作画的机器人。首先，将笔移动到起点（0,0）坐标处。当命令机器人从起点到另一个点画线时，它会拖着笔在纸上移动到新的位置。每一条新画的线都会以上次停下的点作为起点，而将新的坐标位置作为终点。context.moveTo 就像在说："现在请将笔举起并移动到下一个坐标位置"。尽管这一举动并不会绘制出任何图像，但是它却会影响到下一个命令的执行结果。通常，会调用 context.moveTo 作为首个绘图指令，从而将绘图 API（即笔）移动到你作画的起始位置。

现在你已经的学会绘图指令是以开始做一些有趣的事情了。让我们使用 canvas 绘图 API 创建一个简单的绘图程序。示例如下：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Drawing App</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),
       mouse = utils.captureMouse(canvas);

   function onMouseMove () {
     context.lineTo(mouse.x, mouse.y);
     context.stroke();
   }
   canvas.addEventListener('mousedown', function () {
```

```
        context.beginPath();
        context.moveTo(mouse.x, mouse.y);
        canvas.addEventListener('mousemove', onMouseMove, false);
      }, false);

      canvas.addEventListener('mouseup', function () {
        canvas.removeEventListener('mousemove', onMouseMove, false);
      }, false);
    };
    </script>
  </body>
</html>
```

你可以在本书配套的下载示例（可以在 www.apress.com 上得到）中找到名为 01-drawing-app.html 的文件。该程序除了用到绘图函数外，它也很好地示范了事件处理程序的使用（在第2章中介绍过）。下面让我们深入研究这段代码。

该示例是基于第2章中的框架文件的，你现在应该对它很熟悉了。在脚本的开头，它使用工具函数 utils.captureMouse 记录鼠标的位置，该函数从 utils.js 文件中导入，接下来还为 mousedown 与 mouseup 事件添加了各自的处理程序。

用户每次在 canvas 元素上按下鼠标键的时候都会触发 Mousedown 事件处理程序，而这也正是用户想要在鼠标的当前位置开始画线的时候。事件处理程序会创建一条新的路径并通过调用 context.moveTo 方法将虚拟的绘图笔移动到鼠标所在的位置。随后它为 mousemove 事件添加了监听器。

这样，用户每次移动鼠标，onMouseMove 函数都会被调用。该函数是本程序的核心所在，不过它的实现很简单。它会绘制一条到当前的鼠标位置的直线，并将路径轮廓渲染到 canvas 上。

最后，还有一个 mouseup 事件处理程序，其会移除 mousemove 事件处理程序，这样当鼠标释放后，就不再会有线条绘制到 canvas 上。

这是一个有趣的基础程序，只要添加一些简单的控件，就可以由它构建出一个功能齐备的绘图程序。可以创建一些代表线条颜色和宽度的变量，并通过一些 HTML 窗体实现对它们的修改，再将这些变量赋给 strokeStyle 与 lineWidth 属性。如果你有兴趣的话，就把它当做课后练习吧。

4.5.1 使用 quadraticCurveTo 绘制曲线

canvas 绘图 API 提供了用于在路径上绘制曲线的大量方法。在接下去的几个示例中，将介绍 context.quadraticCurveTo 方法，该方法通过一个控制点实现两点间的曲线连接。它跟 context.lineTo 方法类似，都是以上一次方法调用的终点或上一个 context.moveTo 指令的位置作为起点，就连对线条的风格的设置它也是和 context.lineTo 方法一模一样。两者间唯一的区别在于绘制出来的线条的形状。

context.quadraticCurveTo（cpx, cpy, x, y）接受两个点作为参数：第一个点是控制点，用于影响曲线的形状，第二个点是曲线的终点。该形状由一个名为二次贝塞尔曲线的标准算法决定。该函数会计算出从一个点到另一个点的一条曲线，该曲线会弯向但永不触及控制点，就好像它

被控制点的引力所吸引了。

让我们在下面这个 02-drawing-curves.html 文件中看一下它的实际用法：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Drawing Curves</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),
       mouse = utils.captureMouse(canvas),
       x0 = 100,
       y0 = 200,
       x2 = 300,
       y2 = 200;

   canvas.addEventListener('mousemove', function () {
     context.clearRect(0, 0, canvas.width, canvas.height);

     var x1 = mouse.x,
         y1 = mouse.y;
     //curve toward mouse
     context.beginPath();
     context.moveTo(x0, y0);
     context.quadraticCurveTo(x1, y1, x2, y2);
     context.stroke();
   }, false);
  };
  </script>
 </body>
</html>
```

将该文件载入浏览器中并移动鼠标。该脚本使用预先设定的两个点作为起点和终点，而采用鼠标位置作为控制点。注意，曲线永远不会触碰到控制点，而会在中途偏离控制点。

穿过控制点的曲线

如果你希望曲线能够穿过控制点，可以使用下面这个公式计算控制点。跟之前一样，以（*x0*，*y0*）作为起点，（*x2*，*y2*）作为终点，（*x1*，*y1*）作为控制点。那个我们希望曲线穿过的点称为（*xt*，*yt*）代表目标）。换句话说，如果你希望曲线穿过（*xt*，*yt*），那么（*x1*，*y1*）应该设为什么值呢？下面是计算公式：

```
x1 = xt * 2 - (x0 + x2) / 2;
y1 = yt * 2 - (y0 + y2) / 2;
```

简单来说，将目标点的坐标乘以 2 再减去起点和终点的坐标的平均值。可以画张图探寻该

公式的工作原理，或者拿一些数据测试，根据测试结果去推断该公式是否成立。

为了实践一下该公式，使用鼠标坐标作为（*xt*、*yt*），只需要修改之前的示例代码中的两行代码。将以下代码：

```
x1 = mouse.x;
y1 = mouse.y;
```

改为：

```
x1 = mouse.x * 2 - (x0 + x2) / 2;
y1 = mouse.y * 2 - (y0 + y2) / 2;
```

也可以直接使用 03-curve-through-point.html 文件，其中包含了上述改动。

4.5.2 创建多条曲线

接下来，你可能想要创建多条曲线，不过你很可能想要一条很长的平滑曲线在图中弯向多个方向，而不是一条单独的曲线。为了解决该问题，我们会首先尝试一种错误的方法，不过这也是通常人们会想到的第一个方法。在示例中，预先画好一些点，然后从第 1 个点画一条曲线经过第 2 个点到第 3 个点，再经过第 4 个点到达第 5 个点，再经过第 6 个点到达第 7 个点，以此类推。代码如下所示（在文件 04-multi-curve-1.html 中）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Multi Curve 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        points = [],
        numPoints = 9;

   //array of random point objects
   for (var i = 0; i < numPoints; i++) {
      points.push({
        x: Math.random() * canvas.width,
        y: Math.random() * canvas.height
      });
   }

   //move to the first point
   context.beginPath();
   context.moveTo(points[0].x, points[0].y);

   //and loop through each successive pair
   for (i = 1; i < numPoints; i += 2) {
```

```
        context.quadraticCurveTo(points[i].x, points[i].y, points[i+1].x, points[i+1].y);
    }
    context.stroke();
  };
  </script>
 </body>
</html>
```

window.onload 函数中的第一个 for 循环创建一个包含 9 个点的数组。每个点都是一个拥有 x、y 属性的对象，它们随机地放置在 canvas 上。

开始一条新的路径，把画笔移到第一个点的位置。接下来的 for 循环从 1 开始以 2 为步长递增，绘制一条曲线经过点 1 到达点 2，然后经过点 3 到达点 4，在经过点 5 到达点 6，最后经过点 7 到达点 8。循环在点 8 结束，而它恰好是最后一个点。在上述示例中，至少要包含三个点，而且点的个数必须为奇数。

该程序看上去很合理，直到你测试它。如图 4-1 所示，它根本就不像一条平滑的曲线，而且在某些地方看上去还很尖锐，它的问题在于在连续的两条曲线间并没有协调好它们的走向，而只是简单地穿过了同一个点。

我们必须插入一些更多的点让它看上去更像曲线。方法如下：在每两个点之间，加入一个恰好位于它们中间的新点，并使用它们作为每条曲线的起点和终点，而将原始点作为曲线的控制点。

图 4-2 演示了该解决方案。在图 4-2 中，白色的点是原始点，黑色的点为它们的中点。可以看出 context.quadraticCurveTo 方法在图 4-2 中用到了三次。

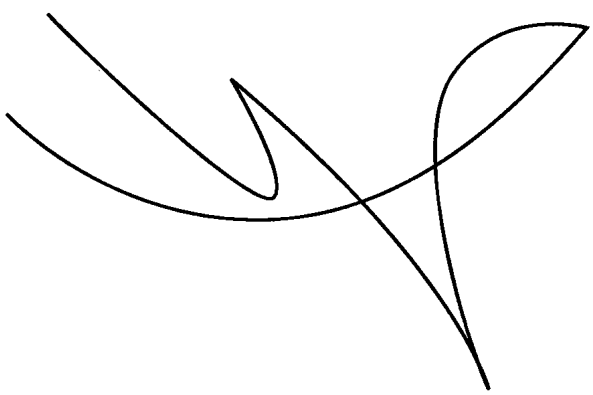


图 4-1 有多条曲线的走向出错，可以明显看出曲线的中断和新的开始

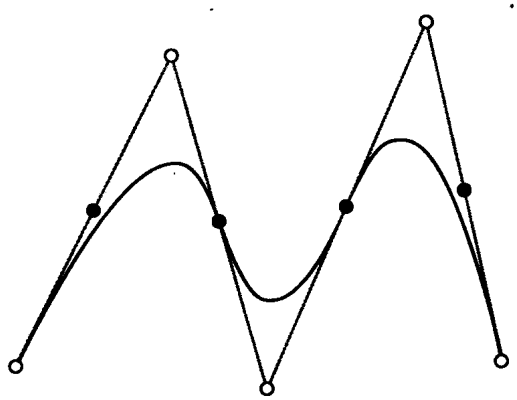


图 4-2 包含中点的多条曲线

注意，在图 4-2 中第一个和最后一个中点并没有用到，而第一个和最后一个原始点仍旧是曲线的起点和终点。只需要为那些从第二个点开始到倒数第二个点之间的点创建中点。以下是之前随机曲线示例的更新版（document 05-multi-curve-2.html）：

```
<!doctype html>
<html>
 <head>
```

```
<meta charset="utf-8">
<title>Multi Curve 2</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
<script>
window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      points = [],
      numPoints = 9,
      ctrlPoint = {};

  for (var i = 0; i < numPoints; i++) {
    points.push({
      x: Math.random() * canvas.width,
      y: Math.random() * canvas.height
    });
  }

  //move to the first point
  context.beginPath();
  context.moveTo(points[0].x, points[1].y);

  //curve through the rest, stopping at each midpoint
  for (i = 1; i < numPoints - 2; i++) {
    ctrlPoint.x = (points[i].x + points[i+1].x) / 2;
    ctrlPoint.y = (points[i].y + points[i+1].y) / 2;
    context.quadraticCurveTo(points[i].x, points[i].y, ctrlPoint.x, ctrlPoint.y);
  }

  //curve through the last two points
  context.quadraticCurveTo(points[i].x, points[i].y, points[i+1].x, points[i+1].y);
  context.stroke();
};
</script>
</body>
</html>
```

在新的代码中，第二个 for 循环从 1 开始到 numPoints–2 结束，这也就略过第一个和最后一个点。在循环中创建一个新的控制点，其 x、y 坐标分别设置为循环中当前点和后续点的 x、y 坐标的平均值。然后，绘制一条穿过当前点并以控制点结尾的曲线，重复此过程直到循环结束。当循环结束时，索引 i 指向倒数第二个点，此时绘制一条曲线穿过它到达最后一个点。

这时，就得到了一条非常平滑的曲线，如图 4-3 所示，而不再是之前那些尖锐的线条。该方法并不限制点的个数为奇数，只要点的个数等于或大于三个即可。

作为该主题的一个变种，document 06-multi-curve-3.html 文件中的以下代码使用相同的技术创建一条闭合曲线。它计算初始中点，即第一个点和最后一个点的平均值，并移动到该位置。然后它遍历剩下的点，获得接下来每两个相邻点中点，最终将最后一条曲线画回到初始的中点，如图 4-4 所示。

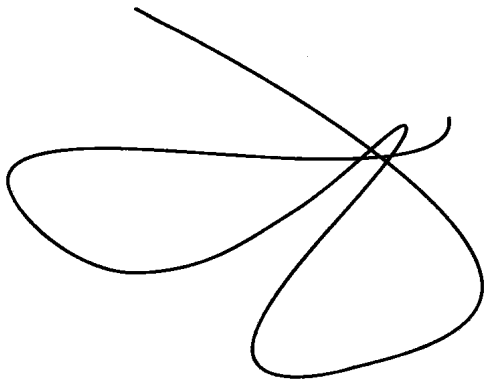


图 4-3 平滑的多条曲线

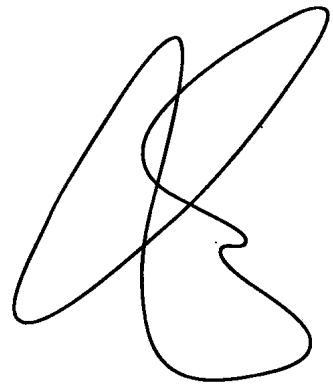


图 4-4 多条闭合曲线

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Multi Curve 3</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        points = [],
        numPoints = 9,
        ctrlPoint = {},
        ctrlPoint1 = {};

    for (var i = 0; i < numPoints; i++) {
      points.push({
        x: Math.random() * canvas.width,
        y: Math.random() * canvas.height
      });
    }

    //find the first midpoint and move to it
    ctrlPoint1.x = (points[0].x + points[numPoints-1].x) / 2;
    ctrlPoint1.y = (points[0].y + points[numPoints-1].y) / 2;
    context.beginPath();
    context.moveTo(ctrlPoint1.x, ctrlPoint1.y);

    //curve through the rest, stopping at each midpoint
    for (i = 0; i < numPoints - 1; i++) {
      ctrlPoint.x = (points[i].x + points[i+1].x) / 2;
      ctrlPoint.y = (points[i].y + points[i+1].y) / 2;
      context.quadraticCurveTo(points[i].x, points[i].y, ctrlPoint.x, ctrlPoint.y);
    }
```

```
        //curve through the last point, back to the first midpoint
        context.quadraticCurveTo(points[i].x, points[i].y, ctrlPoint1.x, ctrlPoint1.y);
        context.stroke();
      };
      </script>
    </body>
  </html>
```

4.5.3 其他形式的曲线

除了之前使用的 context.quadraticCurveTo 方法之外，还有其他一些用于绘制曲线的方法，每个方法都有各自的形状特征。

- bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y): 增加一个到当前路径的点，并由三次贝塞尔曲线连接两个控制点。
- arcTo(cp1x, cp1y, cp2x, cp2y, radius): 使用两个控制点和指定半径为连接到前一个点的直线路径添加一个弧度。
- arc(x, y, radius, startAngle, endAngle [, antiClockwise]): 为连接到前一个点的直线路径添加一个弧度，该弧度将是以 x、y 作为圆心，以 radius 为半径的一个圆的一部分，该部分的起始角度和终止角度分别由 startAngle 和 endAngle 指定。

以上这两个方法在本书并未大量使用，不过 context.arc 方法在示例代码中出现的次数也不少，因为它是绘制圆形一个简单可行的方法。以下代码就示范了如何以点（100,100）为圆心绘制一个半径为 50 的圆。

```
context.beginPath();
context.arc(100, 100, 50, 0, (Math.PI * 2), true);
context.closePath();
context.stroke();
```

4.6 使用填充色创建图形

既然我们已经使用直线或者曲线绘制了多条路径，就可以通过这些线条创建一些图形。context.fill()方法会使用当前的填充样式填充子路径。

跟 context.strokeStyle 类似，context.fillStyle 属性通过一个十六进制格式或 RGB 格式的 CSS 风格的颜色字符串进行设置，其默认值为黑色，即"#000000"。如果你想要使用一种透明色，应该回忆起你需要将其格式化为一个正确的 RGBA 字符串。

可以在绘制图像时随时调用 context.fill()方法，它会填充从起始点到当前位置中所有子路径所构成的形状。一般来说，绘图的顺序如下所示：

- beginPath;
- moveTo;
- lineStyle（若有的话）;

- fillStyle（若有的话）；
- 一系列 lineTo 与 quadraticCurveTo 方法（或类似的曲线）；
- closePath（闭合图形）；
- fill（图形的颜色）；
- stroke（图形的轮廓）。

可以以任意顺序调用前 4 个方法而不用担心影响到将要绘制的图像。线条和填充的样式不是必需的，如果你不指定的话，将会采用默认的颜色（即黑色）。

如果在填充图形前没有将线条画回到起始点，当调用 context.fill 方法时，canvas 还是会填充图形，只不过它不会关闭当前路径并绘制完整的轮廓。为了避免出现这种情况，会调用 context.closePath 方法，这样当前路径就会自动闭合了。

可以进一步体验填充图形，甚至可以使用之前章节中的闭合曲线的示例（06-multi-curve-3.html），因为该示例已经帮你绘制了一个闭合的图形。可以在第一次调用 context.quadraticCurveTo 方法前插入一条 context.fillStyle 语句，比如 context.fillStyle = "#ff0000"，并在代码的末尾调用 context.fill()方法。

在完成了以上操作后，你会发现在曲线相交的区域是没有填充效果的。对此，我们并没有太多办法，不过在大多数情况下，你并不会绘制一个相交的并且如此复杂的图形，而且你还可以利用这一行为创建一些有趣的图形。

4.6.1 创建渐变填充色

渐变填充色包含至少两种颜色。图形的一部分以一种颜色开始，然后渐渐地变成另一种颜色，在其变化过程中会经历一种或多种预定义的颜色。

指定渐变类型

canvas 支持两种渐变填充：**线性渐变**或**放射性渐变**。在**线性渐变填充**中，渐变色在一条直线上从一个点延伸到另一个点。使用 context.createLinearGradient（x0, y0, x1, y1）方法将沿着点（x0,y0）与点（x1,y1）之间的直线填充渐变色。图 4-5 展示了一些线性渐变填充的示例。

放射性渐变填充从你指定的空间的中心开始向各个方向扩散，从而创建出一个圆形。渐变从圆心开始到圆的边缘结束。context.createRadialGradient（x0, y0, r0, x1, y1, r1）方法以两个圆形作为参数，每个圆形由其圆心坐标和半径指定。图 4-6 展示了一个使用放射性填充的例子。

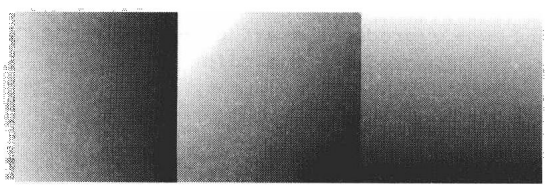


图 4-5 线性渐变填充

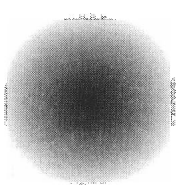


图 4-6 放射性渐变填充

4.6.2 设置渐变色的颜色

为了实现渐变填充，在渐变过程中需要至少两种不同的颜色。在仅有两种颜色的情况下，填充的图形会从第一种颜色慢慢地变成第二种颜色。如果加入更多的中间颜色，填充的图形就会自然地从第一种颜色变到第二种颜色再到第三种颜色，以此类推，直到变成最终的颜色。

在设置颜色的同时，还需为每种颜色指定它们的位置，该值是介于0～1的小数。每种颜色的这个位置值称为填充的比例。比如，在两种颜色的填充中，分别指定0与1作为比例。而在一个等距的三色渐变中则会使用0、0.5与1作为填充比例，0.5表示第二种颜色恰好位于第一种颜色和最后一种颜色的中间位置。如果填充比例设置为0、0.1与1的话，第一种颜色就会很快变为第二种颜色，再慢慢变成最后一种颜色。请时刻牢记，这里的数值不是表示像素值而是与1的比率。

以颜色及其比例值作为参数，可以使用Gradient对象的Gradient.addColorStop（ratio, color）方法为渐变色添加颜色：

```
var gradient = context.createLinearGradient(0, 0, 100, 100);
gradient.addColorStop(0, "#ffffff");
gradient.addColorStop(1, "#ff0000");
```

以上代码在点（0,0）至点（100,100）之间创建一种线性渐变色，并定义两种颜色以及各自的比例，该渐变色将从白色渐变为红色。

然后将该渐变色赋值给context.fillStyle属性，并使用它绘制一个图形：

```
context.fillStyle = gradient;
context.fillRect(0, 0, 100, 100);
```

这里使用了一个简便方法 context.fillRect，它会像画线指令一样绘制并填充一个矩形而不用频繁地移动绘图笔。让我们看一下把以上方法结合在一起的完整示例（document 07-gradient-fill-1.html）会在canvas上绘制一个什么图形：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Gradient Fill 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        pt1 = {x: 0, y: 0},              //gradient start point
        pt2 = {x: 100, y: 100},          //gradient end point
        gradient = context.createLinearGradient(pt1.x, pt1.y, pt2.x, pt2.y);

    //gradient from white to red
    gradient.addColorStop(0, "#ffffff");
    gradient.addColorStop(1, "#ff0000");
```

```
    context.fillStyle = gradient;
    context.fillRect(0, 0, 100, 100);
  };
  </script>
 </body>
</html>
```

在浏览器中运行该示例，你将会看到一个从白色渐变到红色的方块。现在，让我们来修改绘图的代码把方块画在另一个位置。在 08-gradient-fill-2.html 文件中，改变了方块的起始位置：

```
context.fillRect(100, 100, 100, 100);
```

此时方块变为了纯红色。怎么回事？渐变色从点（0,0）开始到点（100,100）结束，而点（100,100）恰恰又是方块的起始位置。在这个位置上，渐变色已经变为纯红色，并且在此之后的颜色都将是纯红色。所以，需要相应地调整渐变色的 x、y 坐标，使其以图形的左上角坐标作为起始点，如下所示：

```
pt1 = {x: 100, y: 100},          //gradient start point
pt2 = {x: 200, y: 200},          //gradient end point
gradient = context.createLinearGradient(pt1.x, pt1.y, pt2.x, pt2.y);
```

此时，把渐变色的起始点设置为方块的起始点。使用同一个方块，可以尝试加入更多渐变色，看看会达成什么样的效果。首先，尝试以下 3 种颜色：

```
//gradient from white to blue to red
gradient.addColorStop(0, "#ffffff");
gradient.addColorStop(0.5, "#0000ff");
gradient.addColorStop(1, "#ff0000");
```

改变中间颜色的比例看看渐变色会如何变化，试着将其改为 0.25 或 0.75。

以下是透明度渐变的示例。两种颜色的色彩值是一样的，只是透明度发生了变化：

```
//all black alpha blend
gradient.addColorStop(0, "#000000");          //alpha 100%
gradient.addColorStop(1, "rgba(0, 0, 0, 0)"); //alpha   0%
```

最后，让我们尝试通过两个圆形定义的放射性渐变：

```
c1 = {x: 150, y: 150, radius: 0},
c2 = {x: 150, y: 150, radius: 50},
gradient = context.createRadialGradient(c1.x, c1.y, c1.radius,
                                        c2.x, c2.y, c2.radius);
```

可以将在线性渐变填充中所作的尝试都运用到放射性渐变填充的示例中，或者尝试运行 09-gradient-fill-radial.html 示例。

4.7 加载并绘制图片

你可能希望在动画中绘制一张外部图片，而有两种方式用于在动画中访问外部图片：在脚本运行过程中加载一个 URL 或使用 DOM 接口访问一个内嵌在 HTML 中的图片元素。当图片加载完成后，再使用绘图 API 将其渲染到 canvas 元素上。

4.7.1 加载图片

为了实现在运行时加载一张图片，可以创建一个 Image 对象并将其 src 属性设置为某个图

片文件的 URL 路径。当图片加载完成后，它就会执行其 onload 方法所关联的回调函数。在以下这个简单的示例中（10-load-image.html 文件），在图片加载完成后将其绘制到 canvas 元素上：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Load Image</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
     context = canvas.getContext('2d'),
     image = new Image();

   image.src = "picture.jpg";
   image.onload = function () {
    context.drawImage(image, 0, 0);
   };
  };
  </script>
 </body>
</html>
```

使用 context.drawImage 方法将图片绘制到 canvas 元素上，该方法接受一个图片元素与 canvas 上的一个 x、y 坐标。特意把这段代码放在 image.onload 方法的回调函数，以免该方法在图片加载完成前就执行而无法渲染任何内容。

context.drawImage 方法有以下 3 种调用方式，分别使用多个参数。

- drawImage(image, dx, dy)：在 canvas 的（dx,dy）坐标上绘制一张图片，（dx,dy）是图片左上角所在的位置。
- drawImage(image, dx, dy, dw, dh)：分别根据 dw、dh 的值设定图片的宽度与高度，并将其绘制在 canvas 的（dx,dy）坐标。
- drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)：将图片裁剪到矩形区域（sx, sy, sw, sh）中，并缩放至（dh, dw），再将其绘制到（dx,dy）坐标。

4.7.2 使用图片元素

也可以使用 DOM 接口访问 HTML 文件中的一个图片元素，该方法的优点在于在脚本执行时图片已经完全加载完成。当采用该方法时，请记得使用 CSS 隐藏图片元素，否则你可能会看到图片在文件中渲染了两次。以下是完整的示例代码（11-embed-image.html）：

```
<!doctype html>
<html>
 <head>
```

```
 <meta charset="utf-8">
 <title>Embed Image</title>
 <link rel="stylesheet" href="style.css">
 <style>
 #picture {
  display: none;
 }
 </style>
</head>
<body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <img id="picture" src="picture.jpg">
 <script>
 window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      image = document.getElementById('picture');

  context.drawImage(image, 0, 0);
 };
 </script>
</body>
</html>
```

4.7.3 使用视频元素

除了可以将一幅静止的图片绘制到 canvas 之外，canvas 还支持逐帧的视频渲染。video 元素是 HTML5 规范中引入的一个激动人心的功能，通过它可以直接在 Web 浏览器中播放视频，而无须使用诸如 Adobe Flash 之类的插件。尽管视频不同于本书所涉及的动画，但是当它在 canvas 上播放时，也可以通过它实现一些有趣的视觉效果。

为了在文件中引入一段视频，可以在 HTML 文件中加入以下元素：

```
<video controls autobuffer>
 <source src="movieclip.mp4" type="video/mp4"/>
 <source src="movieclip.webm" type="video/webm"/>
 <source src="movieclip.ogv" type="video/ogg"/>
 <p>This browser does not support the <code>video</code> element.</p>
</video>
```

这里，controls 属性会为视频播放加入播放和停止按钮，而 autobuffer 属性则会通知浏览器立即下载视频文件，这样当用户单击播放按钮后就可以立即播放视频。<p>元素则用于在用户浏览器不支持 HTML5 视频元素时给予相应的提示，而且当浏览器支持 HTML5 视频元素时，不会出现这段提示。

这里用到了多个<source>标签，它们为视频格式提供了多种选择。由于授权许可的限制，不同的浏览器支持不同的视频文件格式。为了确保视频可以在各个支持 HTML5 的浏览器上正常播放，必须提供多个版本的视频文件。MP4 作为一个封闭的格式（在本书编写时）被 IE 和 Safari 浏览器要求使用。而 WebM 与 Ogg 都是开放的并且无须授权的视频格式，它们被 Chrome、Firefox 以及 Opera 所支持。由于浏览器领域正处于急速发展，为了确保视频文件可以正确播放，

请在尽可能多的的浏览器上测试视频及其相应格式。

在下面这个例子中，载入了一段视频并将其逐帧渲染到 canvas 元素上。由于一段视频本质上就是按顺序播放的一系列静止的图像，因此 canvas 元素会在一个动画循环中不断绘制视频中的当前帧。在该文件中把 canvas 元素设置成与视频一样的大小以便能够展示完整的一帧图像。可能需要根据视频的大小对 canvas 元素的大小做出相应的调整。代码如下（12-video-frames.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Video Frames</title>
  <link rel="stylesheet" href="style.css">
  <style>
  #movieclip {
   display: none;
  }
  </style>
 </head>
 <body>
  <canvas id="canvas" width="640" height="360"></canvas>

  <video id="movieclip" width="640" height="360" autoplay>
   <source src="movieclip.mp4" type="video/mp4"/>
   <source src="movieclip.webm" type="video/webm"/>
   <source src="movieclip.ogv" type="video/ogg"/>
   <p>This browser does not support the <code>video</code> element.</p>
  </video>
  <script src="utils.js"></script>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),
       video = document.getElementById('movieclip');

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);

      context.drawImage(video, 0, 0);
    }());
  };
  </script>
 </body>
</html>
```

粗体部分是重要代码。向 video 元素添加了 autoplay 属性，这样它就会在页面加载时自动播放。之所以这么做是因为其 HTML 元素被 CSS 所隐藏了，而我们又无法访问到其内置的控件，这样我们就无法通过代码播放视频了。如果你手头上没有任何视频，你就可以使用 www.apress.com 网站上本书的源代码中的示例视频。

video 元素跟 audio 元素相关，而 audio 元素也是在 HTML5 中引入的新元素。不过，暂时不要太着急，第 19 章会介绍如何为动画添加声音。

4.8 操纵像素

通过绘图 API 可以直接操纵组成 canvas 上图像的各个像素。可以创建、读取并写入像素数据，从而得以在 Web 浏览器上创造出一些先进的视觉效果与动画。

尽管直接操纵像素是一项非常强大的功能，但是如何用好这个功能是一个高级的话题。以下代码旨在展示一些可以通过操纵像素实现的有趣的视觉效果。由于我们不会在本书中的其他章节使用该项技术，因此你可以把这部分内容当做选修教材。如果你对代码有些困惑（这里的实现不是那么简单明了）或者想跳到后面的精彩内容，你尽可以跳过这部分内容。而在你需要的时候随时翻回到这里。

4.8.1 获取像素数据

直接访问像素的功能是由 canvas 上下文中的 ImageData 对象提供的。调用以下代码则可以获得 canvas 元素上一个矩形区域的像素数据：

```
var imagedata = context.getImageData(x, y, width, height);
```

这里返回的对象包含 3 个属性：width、height 与 data。width 与 height 属性用于表示以像素为单位的图像大小，而 data 则是一个表示图像数据的数组。很快我们会深入介绍其中的内容。

指定一个矩形的大小则可以新建一个空白的 ImageData 对象，代码如下：

```
var imagedata = context.createImageData(width, height);
```

或者，也可以为该方法传入另一个 ImageData 对象，即 context.createImageData(anotherImageData)。此时会创建一个与传入参数相同大小的新对象，因为该方法并不会复制输入参数中的像素数据，所以新建对象中的像素为透明黑色。

> 当读取像素数据时，请确保任何渲染到 canvas 上的外部资料（图像或视频）来自于与运行脚本所在 HTML 文件相同的域。如果把一个跨域的图像绘制到 canvas 上，它会被标为脏数据，你也无法再从中读取像素数据。为了避免出现这一情况，会把一个安全异常抛到调试控制台。这一行为是在 HTML5 规范中定义的，并且这一行为与大多数 Web 浏览器所遵循的同域安全策略是一致。所以，你甚至有可能无法在本地机器上测试某些示例，除非浏览器提供选项让绕过这一安全策略，否则需要将所有资料上传到一台服务器上并在那里进行测试。

访问像素数据

ImageData 对象的 data 属性返回一个一维数组，该数组包含构成像素颜色每个合成色的值，它们以 RGBA 的顺序出现在数组中，并通过 0～255 之间的整数表示（注意 alpha 值也在此范围内，而不是 CSS 风格的颜色格式中使用的 0.0～1.0）。这些像素以从左到右，从上到下的顺利排列。

以下数组表现了一个 2×2 正方形范围内的图像数据，其中的 4 个像素都设置为红色：

```
var pixels = [255, 0, 0, 255, 255, 0, 0, 255,
              255, 0, 0, 255, 255, 0, 0, 255];
```

数组中的第一个像素的颜色值标记为粗体，之后每4个元素（RGBA）为一组，分别代表图像中的每个像素。以上每个像素中的红色和透明度通道都设置为最大值（255），而绿色和蓝色通道则设置为最小值（0）。

通过指定数组的下标可以访问任何一个特定的像素。因为每个像素由数组中的4个元素表示，所以可以通过以下方式逐个遍历图像数据中的每个像素：

```
for (var offset = 0, len = pixels.length; offset < len; offset += 4) {
  var r = pixels[offset],        //red
      g = pixels[offset + 1],    //green
      b = pixels[offset + 2],    //blue
      a = pixels[offset + 3];    //alpha
}
```

有些时候，你可能希望访问图像中某个特定坐标的像素。以下代码展示了如何获得该像素在数组中的偏移量。

```
var offset = (xpos + ypos * imagedata.width) * 4;
```

当你获得偏移下标量后，你就像之前那样访问像素的每个合成色了。

4.8.2 绘制像素数据

context.putImageData（imagedata, x, y）方法用于将一个ImageData对象绘制到canvas上，其中的参数x、y指定了所绘对象相对canvas左上角的坐标。

可以在图像数据数组中定位某个像素，并修改其三原色的值从而改变该像素的颜色。将更新后的像素值作为第一个参数传入putImageData方法，就可以将其重新绘制到canvas上。

例如，为了清除canvas上一幅图像中红色通道的值，可以遍历每一个像素，将它们的红色部分的值都设置为0：

```
var imagedata = context.getImageData(0, 0, canvas.width, canvas.height),
  pixels = imagedata.data

for (var offset = 0, len = pixels.length; offset < len; offset += 4) {
  pixels[offset]     = 0;                    //red
  pixels[offset + 1] = pixels[offset + 1]; //green
  pixels[offset + 2] = pixels[offset + 2]; //blue
  pixels[offset + 3] = pixels[offset + 3]; //alpha
}

context.putImageData(imagedata, 0, 0);
```

这里将图像数据赋予变量pixels。（因为JavaScript对象和这里的数组都是通过引用赋值的，所以对pixels数组的修改会作用到imagedata.data对象上，因为它们本质上是同一个对象。）在程序中遍历了每个像素，每次跳过4个元素并为其设置新的颜色值。当循环结束后，把更新后的imagedata对象重新绘制到canvas上。

在下面这个例子中经历了同样的像素遍历过程。首先，把一些红色、绿色与蓝色条纹绘制到canvas上，这样我们便有了一些初始的图像数据用于后续工作。随后，将每个像素的颜色颠倒，做到这点很容易，因为每个颜色值的范围都在0～255之间。下面是示例13-invert-color.html的代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Invert Color</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
     context = canvas.getContext('2d');

   //draw some red, green, and blue stripes
   for (var i = 0; i < canvas.width; i += 10) {
     for (var j = 0; j < canvas.height; j += 10) {
       context.fillStyle = (i % 20 === 0) ? "#f00" : ((i % 30 === 0) ? "#0f0" : "#00f");
       context.fillRect(i, j, 10, 10);
     }
   }

   var imagedata = context.getImageData(0, 0, canvas.width, canvas.height),
     pixels = imagedata.data;

   //pixel iteration
   for (var offset = 0, len = pixels.length; offset < len; offset += 4) {
    //invert each color component of the pixel: r,g,b,a (0-255)
    pixels[offset]     = 255 - pixels[offset];     //red to cyan
    pixels[offset + 1] = 255 - pixels[offset + 1]; //green to magenta
    pixels[offset + 2] = 255 - pixels[offset + 2]; //blue to yellow
    //pixels[offset + 4] is alpha (the fourth component)
   }

   context.putImageData(imagedata, 0, 0);
  };
  </script>
 </body>
</html>
```

虽然这只是一个相对简单的图像变换，但是你可以想象到，诸如此类的效果可能变得很复杂，并且占用处理器的大量资源，尤其是当把它应用到动画中的时候。如果类似的效果不是简单地用于一个好玩的演示程序，请一定考虑到这样做可能给用户的机器所带来的负担。

可以采用类似方法方便地将一张彩色图片转换为一张灰度图。根据人眼对不同颜色的光波长的不同敏感程度，可以计算出每个像素的亮度。为了避免读者认为我们随意编造了这些数字，这里可以明确告诉你，这里用到的公式是由国际照明委员会定义的。将上一个例子中遍历像素的 for 循环中的代码替换为如下代码（14-grayscale.html 文件）：

```
//pixel iteration
for (var offset = 0, len = pixels.length; offset < len; offset += 4) {
 var r = pixels[offset],
     g = pixels[offset + 1],
```

```
    b = pixels[offset + 2],
    y = (0.2126 * r) + (0.7152 * g) + (0.0722 * b); //luminance

  pixels[offset] = pixels[offset + 1] = pixels[offset + 2] = y;
 }
```

此时之前的条纹图像就变得好像是古老的黑白电视中的图像。

让我们为下一个例子引入一些与用户交互的元素。我们可以通过移动鼠标的位置影响图像数据，并把对像素的处理放入动画循环的每一帧中。

你可以看到这里用第 3 章中的距离公式计算出了鼠标光标与当前像素的距离。在这个例子中遍历像素的方法略有不同，使用了一个双重循环以指定像素在 canvas 元素上的坐标位置。以下代码可以在示例 15-pixel-move.html 中找到：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Pixel Move</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),
       mouse = utils.captureMouse(canvas);

   (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);

    //draw some stripes: red, green, and blue
    for (var i = 0; i < canvas.width; i += 10) {
     for (var j = 0; j < canvas.height; j += 10) {
      context.fillStyle = (i % 20 === 0) ? "#f00" : ((i % 30 === 0) ? "#0f0" : "#00f");
      context.fillRect(i, j, 10, 10);
     }
    }

   var imagedata = context.getImageData(0, 0, canvas.width, canvas.height),
       pixels = imagedata.data;

   //pixel iteration
   for (var y = 0; y < imagedata.height; y += 1) {
    for (var x = 0; x < imagedata.width; x += 1) {

      var dx = x - mouse.x,
        dy = y - mouse.y,
        dist = Math.sqrt(dx * dx + dy * dy),
        offset = (x + y * imagedata.width) * 4,
         r = pixels[offset],
         g = pixels[offset + 1],
```

```
        b = pixels[offset + 2];

      pixels[offset]     = Math.cos(r * dist * 0.001) * 256;
      pixels[offset + 1] = Math.sin(g * dist * 0.001) * 256;
      pixels[offset + 2] = Math.cos(b * dist * 0.0005) * 256;
     }
    }
       context.putImageData(imagedata, 0, 0);
    }());
   };
   </script>
  </body>
 </html>
```

当运行以上脚本时，你可以看到一种称为“迷幻闪光灯”的效果。把颜色值作为参数传入一些三角函数并且缩放，不过始终控制在有效的颜色值（0～255）范围内。因为这个例子纯粹是出于娱乐目的，所以不用太在意这些函数的具体功能，只要知道我们在尝试利用它们变换出一些有趣的颜色即可。可以随意更改这些值，尝试组合出特有的颜色。

在本章的最后一个示例中，又回归到了第一个示例：简单的绘图应用（01-drawing-app.html）。不过，这次将通过直接绘制像素创造出喷漆的效果。代码如下（示例 16-spray-paint.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Spray Paint</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),
       mouse = utils.captureMouse(canvas),
       imagedata = context.getImageData(0, 0, canvas.width, canvas.height),
       pixels = imagedata.data,
       brush_size = 25,
       brush_density = 50,
       brush_color;

   canvas.addEventListener('mousedown', function () {
     brush_color = utils.parseColor(Math.random() * 0xffffff, true); //to number
     canvas.addEventListener('mousemove', onMouseMove, false);
   }, false);

   canvas.addEventListener('mouseup', function () {
     canvas.removeEventListener('mousemove', onMouseMove, false);
   }, false);

   function onMouseMove () {
     //loop over each brush point
```

```
        for (var i = 0; i < brush_density; i++) {
         var angle = Math.random() * Math.PI * 2,
              radius = Math.random() * brush_size,
              xpos = (mouse.x + Math.cos(angle) * radius) | 0, //remove decimal
              ypos = (mouse.y + Math.sin(angle) * radius) | 0,
              offset = (xpos + ypos * imagedata.width) * 4;

           //pixel component colors: r,g,b,a (0-255)
           pixels[offset]     = brush_color >> 16 & 0xff; //red
           pixels[offset + 1] = brush_color >> 8 & 0xff;  //green
           pixels[offset + 2] = brush_color & 0xff;       //blue
           pixels[offset + 3] = 255;                      //alpha
        }

        context.putImageData(imagedata, 0, 0);
        }
    };
    </script>
   </body>
  </html>
```

该脚本使用在第一个示例中创建的事件处理程序，并以同样的方式工作。当按下鼠标按钮后，会指定一个随机的颜料色彩并启动绘制像素的代码。鼠标的周围区域被填充了随机像素从而模拟出喷漆的效果。可以随意更改画刷的大小以及密度以获得不同的效果，如果你有冒险精神，你甚至可以更改像素中的各个合成色。

操作像素是一项技术，它可以用于实现一些高级的动画和视觉效果。尽管它并不是本书的重点，不过作为渲染图像到 canvas 元素上的另一种方式，它很值得体验。

4.9 本章中的重要公式

在本章中你又为你的收藏加入了一些更有价值的工具函数，这些函数大多与颜色处理相关。

4.9.1 从十六进制转换到十进制

```
console.log(hexValue);
```

4.9.2 从十进制转换到十六进制

```
console.log(decimalValue.toString(16));
```

4.9.3 组合三原色

```
color = red << 16 | green << 8 | blue;
```

4.9.4 提取三原色

```
red = color24 >> 16 & 0xFF;
```

```
green = color24 >> 8 & 0xFF;
blue = color24 & 0xFF;
```

4.9.5 绘制一条穿越某个点的曲线

```
// xt, yt is the point you want to draw through
// x0, y0 and x2, y2 are the end points of the curve
x1 = xt * 2 - (x0 + x2) / 2;
y1 = yt * 2 - (y0 + y2) / 2;
context.moveTo(x0, y0);
context.quadraticCurveTo(x1, y1, x2, y2);
```

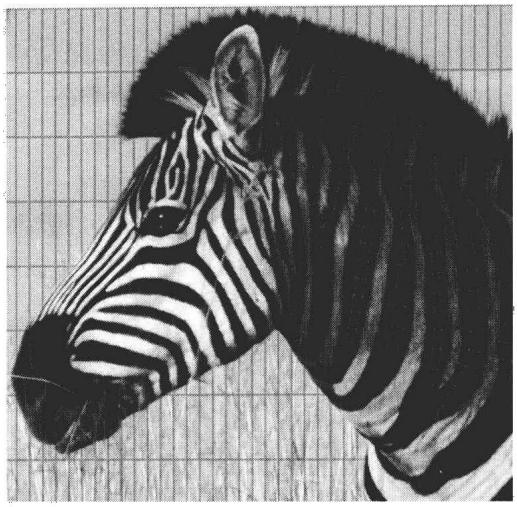
4.10 小结

本章虽然没有过多涉及动画，但它还是很认真地向你展示了一些用于创建可视内容的方法，这些方法将有助于你在后续章节中进一步学习如何实现动画。具体来说，本章介绍了颜色、canvas 绘图 API、加载图片与操纵像素。

这些知识为了实现动态的、富有表现力的动画提供了所必需的一些基础绘图工具。因为你会在本书中用到本章所介绍的大量内容，所以接触并很好地理解它们是非常有帮助的。实际上，你在第 5 章介绍速度与加速度的时候就会首次亲身体验其中的一些技术。

第二部分

基 本 动 画



第 5 章　速度向量和加速度

本章涵盖以下内容：

- 速度
- 加速度

恭喜各位，从本章开始我们开始进入真正的动画编程部分，这就意味着：（a）你已经坚持学习了前面的所有章节；（b）你感觉已经掌握了前几章的内容，所以跳过了前面的章节；（c）你感觉无聊，所以跳过了前面的章节。但是不管怎么样，你还是进入到了本章的学习。你只须记住，当你遇到问题时，很可能可以在之前的章节中寻找到答案。

本章从介绍一些基本的运动属性开始：速度、向量以及加速度。今后几乎所有的动画代码中都会用到这些概念。

5.1　速度向量

物体运动的最基本属性就是速度。很多人把速度向量（velocity）和速度（speed）等同，这其实并不准确，因为速度仅仅是速度向量的一部分，速度向量的概念还包含一个非常重要的因素：方向。速度向量通俗一点的说法是：某个方向上的速度。下面举个例子，让我们来看看它们的不同之处：

如果你开车从 X 位置出发，以每小时 30 英里的速度行驶一个小时，仅仅通过这样的描述很难判断出你的位置。但是，如果我说以同样的速度向北行驶一个小时，那么大家就可以知道我的准确位置了。这在动画中是非常重要的，因为你需要准确知道物体所处的位置。仅仅描述

一个物体以某个速度移动是不够的，为了实现物体在运动的假象，还需要为它在每一帧中指定一些具体的 x、y 轴坐标。而这恰恰是引入速度向量的原因，如果知道物体在某一帧时的位置，那么只要知道它的运动速度和方向，就可以知道物体在下一帧时所处的位置了。

速度向量的速度部分通常以每帧像素数来定义。换句话说，如果一个物体在一帧开始的时候处在某个点，那么它的速度就是指在这帧结束时它距离开始那个点有多少个像素。

使用每帧像素数衡量速度适用于大多数情况，并且这也是最方便编码的方式。不过，由于 Web 浏览器中的帧率并不稳定，因此当浏览器中有太多复杂计算或 CPU 忙于其他计算工作时，使用每帧像素数可能会使动画变慢。如果你需要开发一些对帧率要求恒定的游戏或仿真时，那么你可以考虑使用基于时间间隔的动画作为替代方案。第 19 章会介绍一个使用该方案的例子。

在开始针对速度向量编码之前，先要为大家介绍一些有关向量的知识，因为速度向量只不过是一种特殊的向量而已。

5.1.1 向量与速度向量

向量由大小和方向组成。在速度向量中，大小就是速度。向量用带有箭头的线段表示，箭头的长度就是向量的大小，箭头所指的方向就是向量的方向。图 5-1 展示了一些向量。

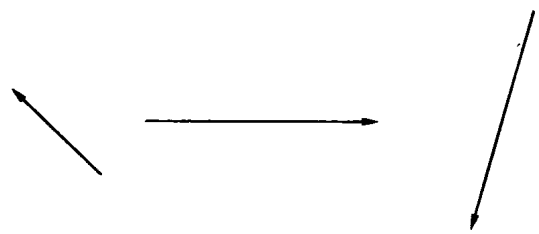


图 5-1 几个向量

需要注意的是，大小总是正数。即使一个向量的大小为负数也只是表示该向量指向一个相反的方向，如图 5-2 所示。

还要注意，向量并不包含任何位置信息。即使在速度向量中也没有说明物体运动的起点或终点，它仅仅指出运动的速度和方向。所以，即便两个向量描述位于不同位置的两个不同物体，它们仍旧可能是相等的，如图 5-3 所示。

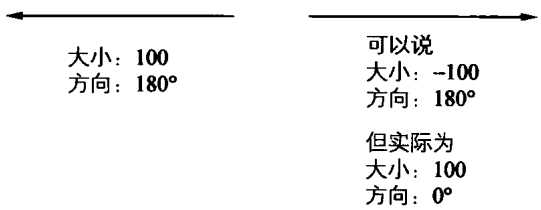


图 5-2 负向量只是指向相反方向的正向量

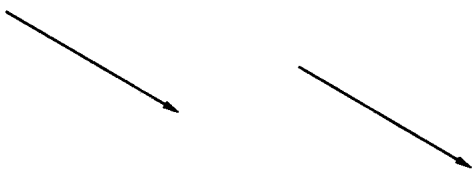


图 5-3 如果向量的大小与方向均相同，则它们相等，而不取决于它们所处的位置

5.1.2 单轴上的速度向量

首先，让我们尝试把问题简化，把速度（向量）只放在一条轴上：x 轴（即水平运动）。即

让物体从屏幕的左侧移动到右侧——这也是我们最习惯的观察方式。此时，物体的移动速度就是每一帧它从左到右所移动的像素值。因此，如果速度向量在 x 轴上为 1，就意味着，物体在每一帧都会从左向右移动 1 个像素。同样，如果速度向量在 x 轴上为-1，那么物体每一帧就会从右向左移动 1 个像素。

我们刚刚说过向量不存在负值，现在却又自相矛盾地提起负向量。科学地讲，所谓-1 的速度向量实际上是大小为 1，并且方向为 180° 的速度向量。同理，y 轴正半轴上的速度向量的方向应为 90°（垂直向下），而 y 轴负半轴上的速度向量的方向为 270° 或-90°（即垂直向上方向）。事实上，当计算速度向量在 x、y 轴上的分量时，通常可以将其记作正数或负数，比如你会经常在书上看到诸如“x 轴速度向量为-1”的描述。为了便于理解这个概念，你完全可以把负号看做是 x 轴上的“向左”指示符，y 轴上的“向上”指示符。

在本书中，我们用 **vx** 表示 x 轴上的速度向量，用 **vy** 表示 y 轴上的速度向量。**vx** 为正数表示向右移动，负数则表示向左移动，**vy** 为正数表示向下，负数则表示向上。

当速度向量在一条轴上的时候，只须简单地将其与物体在那条轴上的位置相加。无论 vx 的有多大，只须在每一帧中将其与物体在 x 轴上的位置属性相加就可以得到其在 x 轴上移动后的新位置。

下面是一个与之相关的真实示例。本章中的许多例子都会让物体产生各种移动效果。为了避免在每个例子中重复出现绘制物体的代码，下面创建一个可以重用的 **Ball** 类。这样就可以重复使用它而不必每次重写一段重复的代码。该类首次出现在第 3 章中，不过你很可能并没有收藏这段代码，所以下面又列了一遍。请将以下这段代码复制到 **ball.js** 文件中，并在主文件中导入该文件：

```
function Ball (radius, color) {
  if (radius === undefined) { radius = 40; }
  if (color === undefined) { color = "#ff0000"; }
  this.x = 0;
  this.y = 0;
  this.radius = radius;
  this.rotation = 0;
  this.scaleX = 1;
  this.scaleY = 1;
  this.color = utils.parseColor(color);
  this.lineWidth = 1;
}

Ball.prototype.draw = function (context) {
  context.save();
  context.translate(this.x, this.y);
  context.rotate(this.rotation);
  context.scale(this.scaleX, this.scaleY);
  context.lineWidth = this.lineWidth;
  context.fillStyle = this.color;
  context.beginPath();
  //x, y, radius, start_angle, end_angle, anti-clockwise
  context.arc(0, 0, this.radius, 0, (Math.PI * 2), true);
  context.closePath();
  context.fill();
  if (this.lineWidth > 0) {
```

```
    context.stroke();
  }
  context.restore();
};
```

今后，无论何时引用这个类，都只须将该段脚本导入主文件并调用 **new Ball（size, color）** 方法就可以创建一个新的球体。

既然我们已经有了运动主体，以下就是第一个速度向量的示例，如文件 **01-velocity-1.html** 所示：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Velocity 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        vx = 1;

    ball.x = 50;
    ball.y = 100;
    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      ball.x += vx;
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

在本示例中，首先将 x 轴上的速度（**vx**）设置为 1。注意，因为这里的 1 表示每帧 1 个像素，所以在每一帧中，球体的 x 轴坐标属性上都会被加上 vx 个像素。**window.onload** 函数在浏览器完全加载 HTML 文件后调用，其中实现了将球体绘制到 canvas 上，并在 **drawFrame** 函数中使用 **window.requestAnimationFrame** 函数设置了动画循环。在每一帧中，球体在 x 轴上的位置都相对前一帧的位置右移了 1 个像素。试着运行本示例，你会发现动画中制造出来的运动的假象还是挺真实的。

给 **vx** 一个更大的值，或者换个负数试试，注意观察物体运动的方向。你还可以试试看能否让物体沿着 y 轴运动。

5.1.3 双轴上的速度向量

让物体沿着两条轴运动也很简单。只须定义 vx 和 vy，并在每一帧中将 vx 的值加到 x 属性上，vy 的值加到 y 属性上。这样，物体在每一帧中就会分别在 x 轴和 y 轴上移动若干个像素。如下例（**02-velocity-2.html**）所示：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Velocity 2</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        vx = 1,
        vy = 1;

    ball.x = 50;
    ball.y = 100;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      ball.x += vx;
      ball.y += vy;
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

同样，大家可以试着改变一下速度变量，别忘了试一下负数。

5.1.4 角速度

目前为止，一切顺利。你已经有一些可以运行的真实动画，还用到了在两条轴上的速度向量。不过在很多场景中，甚至是大多数场景中，x 与 y 轴的速度向量无法在动画开始时获得，

其实在这里，对速度向量的定义并不是很严谨。我们说过速度向量是针对某个方向而言的，但是现在我们却在两个方向拥有两个不同的速度。在 canvas 上，通过设定物体的 x、y 轴的坐标

放置物体，所以得到了物体在两条轴上各自的速度以及坐标，但这却偏离了本章一开始的说法。

如果根据最开始的定义，就应该用一个数值描述速度并用一个角度描述方向，那么结果又将如何？假如换一种说法，一个物体以每帧 1 像素的速度向 45° 的方向移动，你可以看到在这个描述中我们根本看不到 **vx**、**vy** 或类似概念的影子。

幸运的是，我们已经介绍过如何根据角度推导出 **vx** 与 **vy** 的概念，回忆一下第 3 章所讲的三角学知识，再看一下图 5-4，其中展示了如何让球体以 1 个像素的速度向 45° 的方向移动。

这幅图是不是看上去有点眼熟？如果在这幅图中再加入一条边，如图 5-5 所示，它是不是就是一个直角三角形？一个由一个角度和斜边定义的直角三角形？

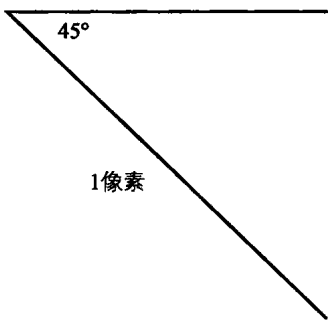


图 5-4 大小与方向

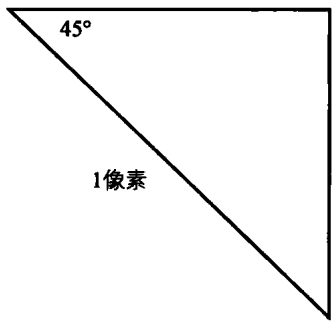


图 5-5 大小及方向映射形成一个直角三角形

注意，三角形的两条邻边恰好落在 *x*、*y* 轴上。事实上，*x* 轴上的直角边恰好等于球体在 *x* 轴上所要移动的距离，而 *y* 轴上的直角边则是球体在 *y* 轴上所要移动的距离。不要忘记，在直角三角形中，只要知道一条边和一个角，就可以计算出其他所有边和角的信息。所以，在已知角度为 45°，斜边长为 1 个像素的情况下，可以使用 **Math.cos** 与 **Math.sin** 函数计算出 **vx** 与 **vy** 的数值。

角的邻边长度为 **vx**，且角的余弦值等于邻边/斜边。换句话说，邻边等于角的余弦值乘以斜边的长度。同理，角的对边为 **vy**，且角的正弦值等于对边/斜边，或者说对边等于正弦值乘以斜边。以下是所要用到的公式：

```
vx = Math.cos(angle) * speed; vy = Math.sin(angle) * speed;
```

在使用 **Math** 函数前，你还会忘记将 45° 转换为弧度制吗？下面就是将示例中的数值运用到公式中的示例代码：

```
vx = Math.cos(45 * Math.PI / 180) * 1;
vy = Math.sin(45 * Math.PI / 180) * 1;
```

以上函数的返回值都约等于 0.707。一旦获得了 **vx** 与 **vy** 的值，就可以将它们添加到动画中球体的 *x*、*y* 轴坐标上。

代码（03-velocity-angle.html）如下所示：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Velocity Angle</title>
```

```
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        angle = 45,
        speed = 1;

    ball.x = 50;
    ball.y = 100;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var radians = angle * Math.PI / 180,
          vx = Math.cos(radians) * speed,
          vy = Math.sin(radians) * speed;

      ball.x += vx;
      ball.y += vy;
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

与前面 **vx**、**vy** 主要不同的地方在于本例中的计算是从角度和速度开始的，其中计算出的速度向量仅仅作为局部变量使用，并且在用过后就丢弃了。当然，在这样一个角度和速度都不会产生变化的简单示例中，更好的做法是在脚本起始处就计算出两条轴上的速度并将它们保存在临时变量中。而对于更复杂的运动来说，角度和速度会在运动中不断变化，这样 **vx** 和 **vy** 的值也就需要在循环中不断改变。

为了更好地体验上述代码的功能，请尝试改变角度值。实际上，只需要调整 **angle** 与 **speed** 两个变量的值，你就可以任意改变球体运动的速度及角度。

下面，让我们从向量的角度重新审视一下这个例子。

5.1.5 向量加法

当在一个平面坐标中出现两个向量时，可以使用向量加法计算出两个向量的合成向量。当向量相加时，只须将它们从头到尾依次相连，最终的合成向量就是那条从第一个向量的起点连接到最后一个向量的终点的向量。在图 5-6 中，可以看到三个向量相加及它们的合成向量。

向量相加的结果与向量本身所处的位置以及时间无关。可以说物体是先沿着这条路前进，再沿那条路前进，然后再到另一条路，路的顺序可以任意选择，甚至可以说物体一次性走完了三条路。其结果就是物体以某个特定的速度和方向完成了一次运动。

让我们回到之前的示例。如果将 x 轴的速度向量向右，再将 y 轴的速度向量竖直向下，那么起点和终点相连的那个向量就是合成速度，见图 5-7。

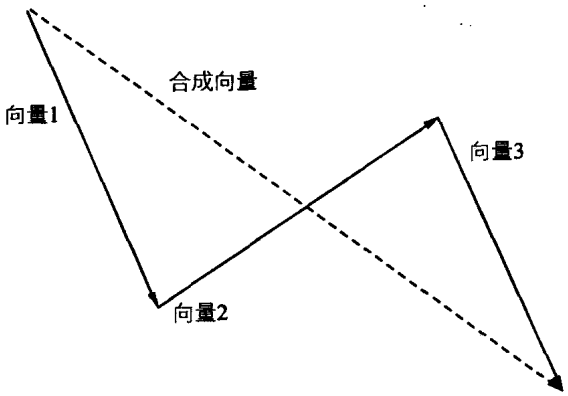


图 5-6 向量加法

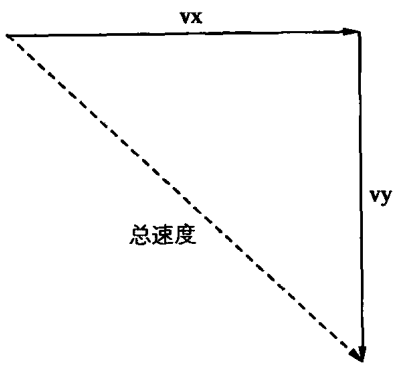


图 5-7 这里是速度的向量表现形式，它跟图 5-5 中的直角三角形一模一样！这预示着我们走在正确的方向上

5.1.6 鼠标追随者

让我们使用速度向量的概念解释早前的一个问题。在第 3 章中，我们曾编写过一个始终让箭头指向鼠标的例子。这个示例使用 **Math.atan2** 计算鼠标与箭头之间的夹角，并旋转箭头使其保持这个角度。

根据刚才所学的知识，可以再为这个角度加入速度的概念，从而得到一个基于当前角度的速度向量。随后，根据鼠标位置移动箭头，使其始终跟随光标移动。这个示例使用与第 3 章的示例中相同的 **Arrow** 类，而不再使用球体。所以找出 **Arrow** 类所在的文件或重写这个类，然后将其放在 **04-follow-mouse.html** 文件所处的目录下：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Follow Mouse</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="arrow.js"></script>
```

```
<script>
window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      mouse = utils.captureMouse(canvas),
      arrow = new Arrow(),
      speed = 3;

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);
    var dx = mouse.x - arrow.x,
        dy = mouse.y - arrow.y,
        angle = Math.atan2(dy, dx), //radians
        vx = Math.cos(angle) * speed,
        vy = Math.sin(angle) * speed;

    arrow.rotation = angle;
    arrow.x += vx;
    arrow.y += vy;
    arrow.draw(context);
  }());
};
</script>
</body>
</html>
```

这是一个相当复杂的动画效果，不过相信大家都能看懂以上这段代码。其中，首先计算出鼠标在 x、y 轴上与箭头的距离，并由此利用 **Math.atan2** 函数计算出它们之间的夹角。然后让箭头依此角度旋转，再使用 **Math.cos** 和 **Math.sin** 与速度相乘计算出 x、y 轴上的速度向量，最后将它们与箭头的 x、y 坐标相加。

5.1.7 速度向量扩展

我们正在步入一块危险的领域，不过本书在诠释速度向量的定义方面有一定独到的见解。尽管速度在狭义上是针对位置的迁移以及空间上的物理运动而言的，但是这并不代表它就一定要限制在物体的 x、y 轴坐标上。

一个绘制到 canvas 上的物体隐含了许多可供调整的属性，并且其中大多数属性都有比较大的取值范围，可以随着时间的变化而改变从而制作出动画效果。此时我们讨论的是这些属性值的改变频率或速度，也许速度向量一词用于此并不是最合适的，但是由于它代表了类似的概念，因此当我们描述一些随着时间而改变的属性时，也通常使用 v（velocity）作为其变量名的首字母。

在下面这个示例中，将旋转一个物体，其中会在动画的每一帧中增加物体的 rotation 属性值。只要增加更大的数值就可以让物体旋转得更快，反之它就会变慢。不论正确与否，通常将旋转速度变量命名为 **vr**，用于表示旋转速度。

同样使用 **Arrow** 类（查看它的 **draw** 方法，你会发现 rotation 属性在调用 **context.rotate** 方法时用到），可以得到以下代码（**05-rotational-velocity**.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Rotational Velocity</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="arrow.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        arrow = new Arrow(),
        vr = 2; //degrees

    arrow.x = canvas.width / 2;
    arrow.y = canvas.height / 2;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      arrow.rotation += vr  * Math.PI / 180; //to radians
      arrow.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

从速度向量的定义来看，此处的速度值为 2，方向为顺时针运动。不过我们不需要严格遵守这个定义，因为仍然在进行一些运动。当使用诸如 alpha 速度向量（物体淡入淡出的快慢）的概念时，所谓的方向就会让人很困惑了。不过把这些属性当做速度向量看待还是很有助于我们对它们的理解。你想在每一帧中通过增加或减少速度向量属性值的大小改变 alpha 属性变化的频率。我们经常发现自己随着时间的变化以特定的频率不断地改变物体的一些属性，这时如果通过首字母 v 对它们进行统一命名会便于我们识别这些属性。因此，我们得到了诸如以下的代码：

```
arrow.x += vx;
arrow.y += vy;
arrow.alpha += vAlpha;
arrow.rotation += vr;
arrow.scaleX = arrow.scaleY += vScale;
// etc.
```

在本书中可以看到很多这样的示例，所以希望大家原谅我经常使用 v 字母作开头，这么做只是为了说明它们是速度向量的“兄弟”。下面来学习加速度。

5.2 加速度

通常认为加速度就是使速度加快，减速度就是使速度减慢。尽管这种说法没什么不对的，但是为了更加准确，还是在本书中为加速度作了一个更为科学的定义。

速度（向量）和加速度有很多相似之处，它们都是向量，都通过大小和方向描述。然而，速度向量改变的是物体的位置，而加速度改变的是其速度向量。

想象一下。你坐进车里，然后启动车子，踩油门。此时，车子的速度为多少？零。踩下油门（也称为加速器）以后，汽车的速度向量开始发生变化（至少其速度部分开始变大，而方向的变化由方向盘决定）。过一两秒后，速度将提升至每小时 4～5 英里，随后时速会变为 10 英里每小时、20 英里每小时、30 英里每小时等。发动机对汽车施加作用力从而改变了其速度向量。

由此，得出加速度的通俗定义：改变物体速度的力量。

而从纯粹的代码角度来看，加速度就是增加到速度向量上的数值。

再举一个例子。假设有一艘宇宙飞船要从 A 星球飞到 B 星球，它的飞行方向取决于 A 星球与 B 星球的相对位置。调整好方向后，飞船开始点燃它的火箭。只要火箭还在喷发，力就持续作用于船体，飞船的速度就会不断变化。换句话说，飞船的前进速度变得越来越快。

到某个时刻，飞船指挥官认为飞船已经足够快了，同时为了节约燃料，他决定关闭火箭的引擎。假设宇宙空间中不存在阻力，飞船就会以恒定速度保持前进。只要火箭不再点火，就不会再有力驱动船体。因此，飞船就失去了加速度，其速度也就不再发生变化。

当飞船接近目的地时，它就需要减慢速度，否则它就会飞过 B 星球（或者在导航仪足够精确的情况下，飞船甚至会成为 B 星球的一部分）。此时，指挥官应该怎么做呢？在宇宙空间中是没法刹车的，因为在宇宙空间中飞船并不接触任何物体。这时，指挥官会让飞船调头，使其朝向与原来前进方向相反的方向，并再次点燃火箭。此时产生了负加速度，或者说反方向的加速度。说实话，这跟之前介绍的向量的概念一模一样。这时火箭产生的力再次改变了飞船的速度向量，不过这次是在减缓速度。飞船的速度将变得越来越慢，并最终减至零。理想状况下，此时飞船应该位于星球地表几英寸的位置，而指挥官也可以熄灭火箭让飞船通过重力着陆。（5.2.3 节将详细介绍重力的概念。）

5.2.1 单轴加速度

让我们将前面学到的知识结合 JavaScript 实践一下。与第一个速度向量的示例类似，在第一个加速度的示例中物体还是沿着一条轴运动。后面几个示例中会再次用到 **Ball** 类。

以下是第一个加速度示例的代码（**06-acceleration-1.html**）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
```

```
 <title>Acceleration 1</title>
 <link rel="stylesheet" href="style.css">
</head>
<body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <script src="utils.js"></script>
 <script src="ball.js"></script>
 <script>
 window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),
       ball = new Ball(),
       vx = 0,
       ax = 0.1;

   ball.x = 50;
   ball.y = 100;

   (function drawFrame () {
     window.requestAnimationFrame(drawFrame, canvas);
     context.clearRect(0, 0, canvas.width, canvas.height);

     vx += ax;
     ball.x += vx;
     ball.draw(context);
   }());
 };
 </script>
</body>
</html>
```

开始的速度向量（**vx**）为零，加速度（**ax**）为 0.1。在每一帧中都把加速度添加到速度向量上，而又把速度向量追加到球体的位置上。

尝试运行该示例。你会看到球体缓慢启动，随后速度越来越快地飞向右侧，并径直移出 canvas。

现在，让我们模拟之前的飞船示例，让球体停止加速乃至反向加速。可以使用方向键做到这点。我们在第 2 章中已经学会如何监听键盘事件，在这里，只须根据作为事件处理程序的参数的事件对象的 **keyCode** 属性，判断按下的键盘键是否是方向键，因为 **keyCode** 属性的值正是由按下的键的数值表示。现在，只需要关心左方向键（**keyCode** 值为 37）和右方向键（**keyCode** 值为 39），并使用它们改变球体的加速度。代码如下所示（**07-acceleration-2.html**）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Acceleration 2</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
```

```
window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      ball = new Ball(),
      vx = 0,
      ax = 0;

  ball.x = canvas.width / 2;
  ball.y = canvas.height / 2;

  window.addEventListener('keydown', function (event) {
    if (event.keyCode === 37) {       //left
      ax = -0.1;
    } else if (event.keyCode === 39) {  //right
      ax = 0.1;
    }
  }, false);

  window.addEventListener('keyup', function () {
    ax = 0;
  }, false);

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    vx += ax;
    ball.x += vx;
    ball.draw(context);
  }());
};
</script>
</body>
</html>
```

在这个例子中，只检查左右方向键是否按下。如果按下了左键，就把 **ax** 设置为负值。如果按下了右键，则将其设置为正值，如果左右键都没有按下则设置为零。在 **drawFrame** 函数中，跟之前一样把速度向量添加到物体的位置坐标上。

尝试运行该示例，你会发现，你无法直接控制物体的速度。也就是说，你无法将球体立即停止。你只能使其缓慢减速直至完全停止。而且当你过度减速时，它甚至会向反方向运动。试试看你能否在它飞出 canvas 元素的边缘前让它停下。

我相信一些跟游戏相关的想法已经蹦入你的脑海。不过在开始有那些想法前，请先缓一缓，因为我们在前面所学的技术会应用到后面的很多场景中。

5.2.2 双轴加速度

与速度向量一样，可以同时在 x、y 轴上设置加速度。为每条轴设定的加速度（通常使用 **ax** 与 **ay** 作为变量名）会分别添加到 **vx** 与 **vy** 上，而后它们又会添加到物体的 x、y 轴坐标属性上。

我们可以很容易地修改上一个例子，为其加入 y 轴加速度。只须加入以下代码：

- **ay 与 vy 变量；**
- **检查上下方向键；**
- **将加速度添加到相应的速度向量上；**
- **将速度向量添加到相应的轴坐标上。**

代码如下（08-acceleration-3.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Acceleration 3</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        vx = 0,
        vy = 0,
        ax = 0,
        ay = 0;

   ball.x = canvas.width / 2;
   ball.y = canvas.height / 2;

   window.addEventListener('keydown', function (event) {
     switch (event.keyCode) {
     case 37:      //left
       ax = -0.1;
       break;
     case 39:      //right
       ax = 0.1;
       break;
     case 38:      //up
       ay = -0.1;
       break;
     case 40:      //down
       ay = 0.1;
       break;
     }
   }, false);

   window.addEventListener('keyup', function () {
     ax = 0;
     ay = 0;
```

```
    }, false);

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      vx += ax;
      vy += ay;
      ball.x += vx;
      ball.y += vy;
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

请注意，在本例中将上/下/左/右键的检查放到了 **switch** 语句中，因为如果全部用 **if** 语句的话代码会变得很繁琐。

尝试运行上例，可以发现小球能够在整个 canvas 上随意移动。试着让物体从左到右移动，然后按“上”键。注意，此时 x 轴上的速度向量没有受到影响，物体在 x 轴上的移动速度保持不变，不过却为球体加入了 y 轴方向的速度。这类似于让飞船转弯 90° 并再次点火。

5.2.3 重力加速度

到目前为止，我们讨论的加速度都是物体对自身施力所造成的，比如，汽车和飞船火箭。这种情况很常见，不过现实中同样还存在其他形式的加速度。任何通过施加作用力导致物体速度变化的情况都是加速度的体现，这包括重力、磁力、弹力、摩擦力等。

观察重力有两种方式。从太阳系的广角镜头来看，重力就是两个天体之间的吸引力，需要综合考虑两个天体间的距离和角度才能计算出每个天体的实际加速度。

另一种观察重力的方法是在地球上或者说非常接近地面的地方使用特写镜头。在我们日常生活中，物体与地球的距离对重力的影响微乎其微。虽然科学地说，当我们乘坐高空的飞机或者攀登在高山上时重力会减小一些，不过这种变化几乎是感觉不到的。所以，在这个层面模拟重力时，你基本上可以将其定义为一个固定值，如同上例中的加速度变量。

同样，因为地球太大而地球上的物体都太小，这样重力加速度的实际方向也变得可以忽略不计了，你可以简单认为它就是向“下”的。换句话讲，无论物体在什么位置，都可以放心地将重力定义为 y 轴方向上的一个加速度。

从代码的角度来说，只须为重力给定一个数值并将其添加到球体在每一帧中的 **vy** 属性上。总的来说，这个数值使用一个诸如 **0.1** 或者更小的小数就可以了。如果数值过大，物体就显得太重了。如果数值太小，那么物体看起来就会像是漂浮的羽毛。当然，可以根据需要使用不同的数值以达到不同的效果，例如，通过不同的重力变化来模拟不同的星球。

下面这个例子添加了重力效果。完整的代码见 **09-gravity.html** 文件，这里就不把它全部列

出来了。这段代码与 **08-acceleration-3.html** 仅有少量不同。除了文件名有所变化之外，在脚本开头的变量列表中加入下面这个变量：

```
var gravity = 0.02;
```

然后在 **drawFrame** 函数中加入一行代码：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  vx += ax;
  vy += ay;
  vy += gravity;
  ball.x += vx;
  ball.y += vy;
  ball.draw(context);
}());
```

这里把重力值设置的比较小以免球体过快地离开屏幕，不过仍然可以使用方向键对其进行控制。其实这里所创建的就是古老的《登月》游戏的物理模型，只须为其再加入一些美观的图案和碰撞检测就可以完美地重现这个游戏了。（后面会介绍到有关碰撞检测的内容，而图形美化部分就需要你自己完成了。）

注意，首先将方向键产生的加速度添加到 **vx** 与 **vy** 上，然后再将重力加速度叠加到 **vy** 上。由此可见处理多个合力并不那么复杂，只须将每个力产生的加速度叠加到速度向量上即可，其中并不涉及什么复杂的加权平均或因式分解计算。当所有力都处理过后，将所得的速度向量添加到物体的坐标位置上就可以得到最终结果。在这个例子中仅仅用到了几个力而已。在本书的后续章节中，将会计算施加到一个物体上的多个合力。

这又回到了向量加法。如果以初始速度作为向量的起点，每个加速度、重力或者其他力都是添加到这个速度向量的附加向量。将它们逐一相加后，绘制一条从起点到终点的直线，就得到了它们的合成向量。你会发现其结果与每个力在 x、y 轴上的分力的总和是相等。

现在，假象一个包含热气球的动画。你可能想要加入一个**上升力**，这是在 y 轴上的另一个加速度，不过它将是一个负数，表示“向上”的方向。此时，这个物体上已经施加了三个力：方向键产生的力、重力以及上升力。为让气球能够自动上升，你可以猜想到或者实验出，上升力必须略高于重力。这很符合逻辑，如果这两个力相等，它们就会相互抵消，你就回到本章的起点，在那个例子中只有方向键会影响到球体的速度。

还可以试着加入风力。很明显，这会是在 x 轴方向上的一个力。取决于风向，这个力可以是正数也可以是负数，换句话说，力的方向可以是 0° 或者 180°。

5.2.4 角加速度

我们说过，加速度和速度向量类似，由大小（力的大小）和方向组成，由此出发，我们可以将其分解到 x 轴与 y 轴上。如果大家加以留心，就会记得可以通过使用 **Math.cos** 与 **Math.sin** 方法做到这点。代码如下：

```
var force = 10,
    angle = 45, //degrees. Need to convert!
    ax = Math.cos(angle * Math.PI / 180) * force,
    ay = Math.sin(angle * Math.PI / 180) * force;
```

既然我们就获得了每条轴上的加速度，就可以更新物体在每条轴上的速度向量乃至位置。

让我回到本章前面提到的鼠标跟随示例，并在速度向量的基础上为该例引入加速度的概念。由于该示例曾使用过 **Arrow** 类，找出 **arrow.js** 文件并将其再次导入。回忆早前的例子，我们曾根据鼠标与箭头之间的夹角计算出 **vx** 与 **vy**。这一次，使用同样的方法先计算出 **ax** 与 **ay**，然后将其累加到速度向量中，再将速度值和速度向量添加到物体的 **x**、**y** 属性上。代码如下所示（**10-follow-mouse-2.html**）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Follow Mouse 2</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="arrow.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        arrow = new Arrow(),
        vx = 0,
        vy = 0,
        force = 0.05;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var dx = mouse.x - arrow.x,
          dy = mouse.y - arrow.y,
          angle = Math.atan2(dy, dx), //radians
          ax = Math.cos(angle) * force,
          ay = Math.sin(angle) * force;
      arrow.rotation = angle;
      vx += ax;
      vy += ay;
      arrow.x += vx;
      arrow.y += vy;
      arrow.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

请注意，本例中将变量 **speed** 替换为 **force**，并将它的值变小。因为加速度对速度的影响是

有递增效果的，所以最好在开始的时候将值设置得小一点，否则速度变化会过快。同样，注意，把 **vx** 与 **vy** 的声明放在了脚本的头部，并在每一帧中增加或减少它们的值，而之前它们会在每一帧中重新计算。当然，在上述代码中完全可以不引入 **ax**、**ay** 变量，而直接将力的正弦与余弦结果添加到速度向量上，之所以这么做是为了增加代码的可读性。

运行示例，你会看到一个箭头始终朝着移动中的鼠标加速移动。回顾本章开篇给大家演示的第一个动画，可以发现我们已经迈出了很大的一步。在仅仅学习了一些有关运动的基本原理后，你就可以创建出一些非常流畅和动态的动画，其中一些甚至达到了非常逼真的程度。而此时，我们甚至还没有结束这一章。

现在让我们将所学的知识组合起来，看看我们能够做出怎样的动画吧！

5.2.5 宇宙飞船

我们之前谈论了很多关于宇宙飞船飞行的例子。现在，根据之前所学的知识，我们已经具备了模拟出一架宇宙飞船正常飞行的能力。

计划如下。首先创建一个宇宙飞船类，跟前面用过的 **Arrow** 与 **Ball** 类一样，它会负责将自己绘制到 canvas 上。你将使用左右方向键控制飞船旋转的方向，并通过上方向键点燃飞船的火箭。当然，火箭位于飞船的尾部，而点燃的火焰也将喷向飞船的后方。因此，火箭产生的推力会使飞船朝着它的头部加速前进。实际上，这里制作的动画跟古老的《行星》游戏非常相像，只是少了游戏背景中的行星而已。

首先，需要创建一个 **ship** 类。它的 **draw** 方法使用少量的 canvas 绘图 API 代码绘制了 4 条白色的线，以向古老的《行星》游戏致敬。以下是 **ship.js** 文件中的代码，会把该文件导入下一个示例中：

```
function Ship () {
  this.x = 0;
  this.y = 0;
  this.width = 25;
  this.height = 20;
  this.rotation = 0;
  this.showFlame = false;
}
Ship.prototype.draw = function (context) {
  context.save();
  context.translate(this.x, this.y);
  context.rotate(this.rotation);
  context.lineWidth = 1;
  context.strokeStyle = "#ffffff";
  context.beginPath();
  context.moveTo(10, 0);
  context.lineTo(-10, 10);
  context.lineTo(-5, 0);
  context.lineTo(-10, -10);
  context.lineTo(10, 0);
  context.stroke();
  if (this.showFlame) {
    context.beginPath();
```

```
    context.moveTo(-7.5, -5);
    context.lineTo(-15, 0);
    context.lineTo(-7.5, 5);
    context.stroke();
  }
  context.restore();
};
```

draw 方法引用了飞船的 **showFlame** 属性，该属性是一个 bool 值，由此可以控制飞船在点火时是否显示火焰，而通过火焰可以容易看出飞船的引擎是否启动。可以在图 5-8 与图 5-9 中看到它们的差别。

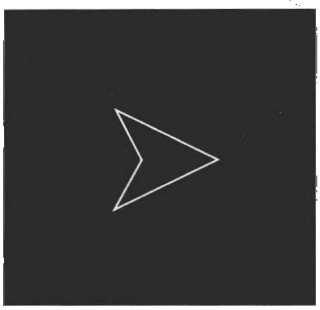


图 5-8 未来的空间旅行

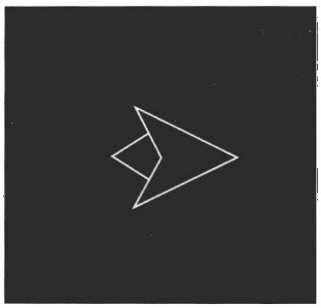


图 5-9 点燃火焰

5.2.6 飞船控制

飞船已经准备好了，接下来就要实现对它的控制了。之前提到，飞船有 3 种控制方式：左转、右转和点火，分别经由左、右、上 3 个方向键控制。此处代码的基本结构与之前的 **08-acceleration-3.html** 文件相似，它们都为 **keydown** 与 **keyup** 事件添加了事件处理程序，并在其中通过一条 **switch** 语句对按键进行分类处理。因为在外层空间中绘制的飞船是白色，所以为了能够让它在 canvas 上显示出来，需要将 canvas 的背景色设置为黑色。这点可以通过在 html 文件的开头引入一个 CSS 风格文件实现。先将代码（示例 **11-ship-sim.html**）列举如下，随后会对代码展开解释：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Ship Sim</title>
  <link rel="stylesheet" href="style.css">
  <style>
  #canvas {
    background-color: #000000;
  }
  </style>
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
```

```
<script src="ship.js"></script>
<script>
window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      ship = new Ship(),
      vr = 0,
      vx = 0,
      vy = 0,
      thrust = 0;

ship.x = canvas.width / 2;
ship.y = canvas.height / 2;

window.addEventListener('keydown', function (event) {
  switch (event.keyCode) {
  case 37:      //left
    vr = -3;
    break;
  case 39:      //right
    vr = 3;
    break;
  case 38:      //up
    thrust = 0.05;
    ship.showFlame = true;
    break;
  }
}, false);

window.addEventListener('keyup', function () {
  vr = 0;
  thrust = 0;
  ship.showFlame = false;
  }, false);

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    ship.rotation += vr * Math.PI / 180;
    var angle = ship.rotation, //in radians
        ax = Math.cos(angle) * thrust,
        ay = Math.sin(angle) * thrust;
    vx += ax;
    vy += ay;
    ship.x += vx;
    ship.y += vy;
    ship.draw(context);
  }());
};
</script>
</body>
</html>
```

首先定义旋转速度向量 vr，即当控制飞船转向时它的旋转速度。它的初值设置为零，这意味着，飞船根本就不会旋转。

```
|var vr = 0;
```

不过在 **keydown** 事件的监听器中，如果在 **swtich** 语句中发现左右方向键按下，则将 **vr** 分别设置为-3 或 3。

```
case 37:  //left
  vr = -3;
  break;
case 39:  //right
  vr = 3;
  break;
```

然后，在 **drawFrame** 函数中，把 **vr** 添加到飞船当前的旋转角度，从而改变飞船的方向。当释放方向键后，**vr** 又重置为零。以上代码负责飞船的转向，接下来让我们看看如何实现飞船的推力。

我们已经绘制好了我们的飞船，并指定了它旅行的目的地，3，2，1，发射。那么我们如何让飞船行驶到目的地呢？其实这个方案恰好与之前更新过的鼠标追随示例的过程相反。在那个例子中，首先计算好角度，然后基于它推算出旋转方向和加速度。这里，则会以旋转角度作为起始参数，反推出角度和施加在 x、y 轴上的力。

声明一个 **thrust** 变量用于追踪飞船在任意时刻的推力大小。很明显，只有在火箭点火后飞船才具有加速度，所以该推力的初始值为零：

```
var thrust = 0;
```

然后再次用到 **switch** 语句。如果上方向键按下，将 **thrust** 设置为一个较小的数值，如 0.05。此处也是控制飞船是否显示火焰的地方，我们会在此时绘制火焰，从而以图像化的方式将火箭推力的产生展现出来：

```
case 38:   //up
  thrust = 0.05;
  ship.showFlame = true;
  break;
```

同样，当释放按键后，将 **thrust** 重置为零，并消除火焰。

```
window.addEventListener('keyup', function () {
  vr = 0;
  thrust = 0;
  ship.showFlame = false;
}, false);
```

此时，在 **drawframe** 函数内，可以计算出旋转角度和向飞船施加的反推力。将旋转角度转换为弧度制并连同 **thrust** 变量使用正余弦函数，就可以得出每条轴上的加速度：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  ship.rotation += vr * Math.PI / 180;
  var angle = ship.rotation, //in radians
        ax = Math.cos(angle) * thrust,
        ay = Math.sin(angle) * thrust;

  vx += ax;
  vy += ay;
  ship.x += vx;
```

```
  ship.y += vy;
  ship.draw(context);
}());
```

尝试运行该示例，并让飞船飞起来。你会惊喜地发现制作如此复杂的运动原来这么简单。

5.3 本章中的重要公式

现在我们的工具箱中又多出不少工具函数，来看看吧。

5.3.1 将角速度分解为 x、y 轴上的速度向量

```
vx = speed * Math.cos(angle);
vy = speed * Math.sin(angle);
```

5.3.2 将角加速度（作用域物体上的力）分解为 x、y 轴上的加速度

```
ax = force * Math.cos(angle);
ay = force * Math.sin(angle);
```

5.3.3 将加速度加入速度向量

```
vx += ax;
vy += ay;
```

5.3.4 将速度向量加入位置坐标

```
object.x += vx;
object.y += vy;
```

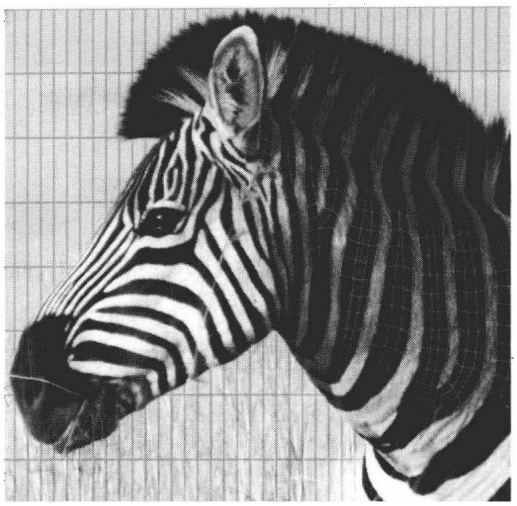
5.4 小结

这一章讲解了关于速度和加速度的基础知识，它们是构成大多数编程动画的两个基本要素。你学习了向量与向量加法，并掌握了如何实现单轴以及双轴上的速度向量以及如何将角速度分解到 x、y 轴上。你还学习了加速度、它跟速度的关系以及如何将其应用到单轴、双轴以及角度上。

本章最重要的部分在于理解加速度与速度的应用，如以下步骤所述：

- 将已知的角速度分解为 x、y 轴上的速度向量；
- 将角加速度同样分解到 x、y 轴上；
- 分别将每条轴上的加速度与速度相加；
- 再分别将每条轴上的速度与物体的位置坐标相加。

将在这些概念的基础上，第 6 章将引入弹力和摩擦力来增加一些物体与外界环境的交互。



第 6 章　边界与摩擦力

本章涵盖以下内容：

- 环境边界
- 摩擦力

我们在前面的章节中介绍了大量的基础知识。你已经学会了如何使用 canvas 绘图 API 绘制图形并使其在各种力的作用下围绕屏幕移动。不过，你可能还是在许多例子中感觉到了一丝烦恼，比如，当一个物体移出屏幕后，它就消失了。有些时候，我们完全不知道如何让它再回到屏幕中，此时你唯一能做的事就是重新加载网页。

然而，在真实的世界中却存在着**边界**的概念，比如，一直在你脚下的大地以及某些时候出现在你身边的墙壁和天花板。除非你想要在动画中模拟宇宙空间，否则我们会制造出一些环境边界使得动画中的物体能够始终保持在我们的视线中。即便动画真的发生在宇宙空间中，我们也需要一种能够将动画中的主体对象保持在视线中的方法。

到目前为止的大量示例中还存在另一个困扰我们的问题是：动画所处的环境对于动画中的物体没有产生任何影响。一旦动画对象开始移动，它就会以不变的速度保持最初的前进方向运动，直到把其他力施加到它身上。实际上，这并没有什么不对的地方，这恰恰是这个宇宙运作的方式。运动中的物体将一直保持其运动状态直到其他力作用于它，这正是**惯性**的体现。从编程的角度来看，除非有力作用于物体，否则物体的速度向量将维持不变。然而，在真实的物理世界中有一类广泛存在的**摩擦力**，它会引起物体的速度向量发生改变，比如，来自空气的阻力。所以，尽管我们已经成功地模拟出了一个物体在真空中的运动，但是创建出一个更符合现实环

境的动画才是我们真正想要实现的目标。

本章将同时解决上述两个问题：首先，你会学习到如何添加边界用于包含动画对象，然后，我们会介绍如何将摩擦力作用于这些对象。

6.1 环境边界

不管你是在运动还是在工作，抑或是在盖房子，你参与的所有活动，都存在一个活动的空间。这样你就可以将注意力集中在发生在这个空间中的事情，并忽略，至少是暂时忽略发生在这个空间以外的事情。

如果在你关注范围内的一个物体移出了那个空间，你就不得不为它做出一个选择：将该物体移回空间或者不再关注该物体，抑或是追随那个物体，即移动空间使其继续包含该物体，即便物体仍在移动。在动画编程中，以上几种选择其实并没有太大差别。

在 canvas 动画中，为物体设置一个空间。一般来说，它就是整个 canvas 元素，不过也可以将其设置为 canvas 的一部分，甚至是一个大于 canvas 的区域。由于动画对象在不断移动，它们最终很有可能会离开这个空间。当它们离开时，可以忽略它们，将它们移回空间或者跟随它们移动。本章将会介绍前两种方法。不过，让我们先确定边界的位置以及如何指定它。

6.1.1 设置边界

在大多数情况下，一个简单的矩形就可以构成一个边界。让我们从最简单的例子开始，基于 canvas 大小的边界。在之前的示例中，一直使用 DOM 接口访问 HTML 文件中的 canvas 元素，通过指定元素的 **id** 获得 canvas 元素的引用并将其赋给一个变量：

```
var canvas = document.getElementById('canvas');
```

动画边界的左上角坐标为(0,0)，右下角坐标则为（**canvas.width, canvas.height**）。可以将这些值保存在变量中，如下所示：

```
var left = 0,
    top = 0,
    right = canvas.width,
    bottom = canvas.height;
```

如果你像上面这样将边界的坐标位置都保存到变量中，那么请注意，即使 canvas 元素的大小将来发生了变化，边界的位置仍将保持不变。所以，只有当你确定要使用一个固定区域作为边界的时候，才采用以上方法。如果你希望始终将整个 canvas 区域作为边界，请直接在代码中引用 **canvas.width** 与 **canvas.height** 属性。

尽管上述示例采用整个 canvas 区域作为边界，但这并不代表必须这么做。比如，可以为物体创建一个活动的空间，并像下面这样为其设置边界：

```
top = 100, bottom = 300, left = 50, right = 400
```

在准备好边界之后，可以检查你关注的所有移动物体，看看它们是否处在这个空间内。可

以通过一系列 if-else 语句做到这点，以下是一段简化的代码：

```
if (ball.x > canvas.width) {
  // do something
} else if (ball.x < 0) {
  // do something
}
if (ball.y > canvas.height) {
  // do something
} else if (ball.y < 0) {
  // do something
}
```

逐句查看这些边界检查，如果球体的 x 轴坐标大于右边界（即 **canvas.width**），就意味着，它已经超出了边界的右边。因为球体 x 轴坐标大于右边界，必然大于 0，所以此时无须检查球体是否超出边界的左边（即 x 轴坐标小于 0），仅在第一条 **if** 语句失败的情况下，才有必要去判断球体的 x 轴坐标是否小于 0。同理，比较球体的 y 坐标与 **canvas.height** 的大小可以判断它是否超出边界的上下区域。不过，因为物体有可能在 x、y 轴上同时越界，所以需要将两个方向上的检查独立开来。

不过我们又该如何处理物体越界呢？我们已经知道存在以下 4 种选择。

- 将物体移除。
- 将其物体置回边界内，就像产生了一个新的物体（重置对象）。
- 让同一个物体出现在边界内的另一个位置。
- 将物体反弹回到边界内。

让我们从最简单的移除物体开始。

6.1.2 移除物体

如果物体会不断产生，那么移除物体的做法将非常有效。被移除的物体会被新近加入的对象所取代，这样 canvas 就永远不会为空。同时，边界内也不会有太多物体在移动，从而导致浏览器变慢。

当多个物体在移动时，应该将它们的引用保存到一个数组中，再通过遍历整个数组来移动它们。可以使用 **Array.splice** 方法移除数组中的某个物体。在已知某个物体的坐标位置时，可以在 if 语句中采用以下方法将其移除：

```
if (ball.x > canvas.width ||
    ball.x < 0 ||
    ball.y > canvas.height ||
    ball.y < 0) {
  balls.splice(balls.indexOf(ball), 1);
}
```

||符号表示“或”，这段代码的意思是：“如果物体超出了右边界或左边界或上边界或下边界，就将其从数组中移除。”以上方法适用于大多数的移除场景，因为它不关心物体在哪儿移出边界，只要它越界了，就将其移除。不过在其他一些场景中，你可能希望能以不同方式处理

在不同边界越界的情况。比如，物体在左边越界的反应不同于在右边越界。为此，你需要用到独立的 **if** 语句，很快你会实现屏幕环绕时看到相应的示例。

不过，这里其实存在一个可能会导致出错的小问题，见图 6-1。

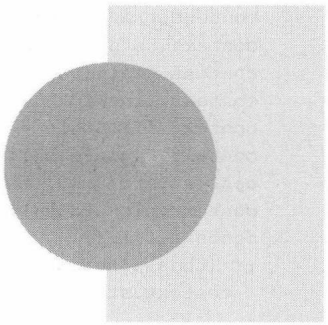


图 6-1 球体尚未完全移出 canvas 时就会被突然移除

在图 6-1 中，你会发现球体的位置（即球体中心的 x、y 轴坐标）正好处在 canvas 的右边界，所以会在此时移除球体。如果球体的运动速度足够快，你可能不会注意到这个小问题。不过如果它移动得比较慢（比如，每帧一个像素），它将会缓慢地移出 canvas 的边界。而当它刚移出一半时，它就会突然从屏幕中消失，以至于破坏了动画的真实性。

为了解决这个问题，球体需要在完全离开 canvas 后再移除。为了实现这点，必须将物体的宽度考虑进来。因为物体的位置坐标在其正中心，所以只须考虑其宽度的一半即可，可以通过访问 **Ball** 类的 **radius** 属性获得该值。**if** 语句会变得稍微复杂一点，修改如下：

```
if (ball.x - ball.radius > canvas.width ||
    ball.x + ball.radius < 0 ||
    ball.y - ball.radius > canvas.height ||
    ball.y + ball.radius < 0) {
  balls.splice(balls.indexOf(ball), 1);
}
```

变化如图 6-2 所示。

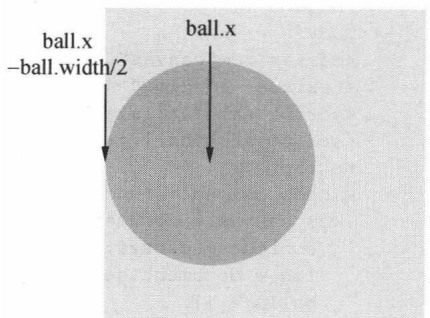


图 6-2 这个球体已经完全移出 canvas，可以被安全地移除

虽然该示例只用到了球体或圆形物体，但是同样的边界检查适用于任何一个形状，只要物体的坐标位置在其正中心即可。

让我们试验一下。在下一个示例中，使用第 5 章中出现的 **Ball** 类，但这次要加上一点新的内容。为该类加入 **vx** 与 **vy** 属性，使得每个球体可以拥有其各自的速度向量。该类的完整代码如下：

```
function Ball (radius, color) {
  if (radius === undefined) { radius = 40; }
  if (color === undefined) { color = "#ff0000"; }
  this.x = 0;
  this.y = 0;
  this.radius = radius;
  this.vx = 0;
  this.vy = 0;
  this.rotation = 0;
  this.scaleX = 1;
  this.scaleY = 1;
  this.color = utils.parseColor(color);
  this.lineWidth = 1;
}
```

```
Ball.prototype.draw = function (context) {
  context.save();
  context.translate(this.x, this.y);
  context.rotate(this.rotation);
  context.scale(this.scaleX, this.scaleY);
  context.lineWidth = this.lineWidth;
  context.fillStyle = this.color;
  context.beginPath();
  context.arc(0, 0, this.radius, 0, (Math.PI * 2), true);
  context.closePath();
  context.fill();
  if (this.lineWidth > 0) {
    context.stroke();
  }
  context.restore();
};
```

下面这个示例（**01-removal.html**）创建了多个球体并在它们移出 canvas 后将它们移除。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Removal</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <textarea id="log"></textarea>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
     context = canvas.getContext('2d'),
     log = document.getElementById('log'),
     balls = [],
     numBalls = 10;

   for (var ball, i = 0; i < numBalls; i++) {
    ball = new Ball(20);
    ball.id = "ball" + i;
    ball.x  = Math.random() * canvas.width;
    ball.y  = Math.random() * canvas.height;
    ball.vx = Math.random() * 2 - 1;
    ball.vy = Math.random() * 2 - 1;
    balls.push(ball);
   }

   function draw (ball, pos) {
    ball.x += ball.vx;
    ball.y += ball.vy;
    if (ball.x - ball.radius > canvas.width ||
        ball.x + ball.radius < 0 ||
        ball.y - ball.radius > canvas.height ||
        ball.y + ball.radius < 0) {
```

```
    balls.splice(pos, 1); //remove ball from array
    if (balls.length > 0) {
      log.value = "Removed " + ball.id;
    } else {
      log.value = "All gone!";
    }
   }
   ball.draw(context);
  }

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    var i = balls.length;
    while (i--) {
     draw(balls[i], i);
    }
   }());
  };
 </script>
</body>
</html>
```

以上代码应该很容易理解。首先，我们创建了 10 个球体的实例，将它们随机放置在 canvas 上，并为它们随机地指定了 x、y 轴上的速度向量，最后将它们都放入一个数组中。

drawFrame 函数则定义了一个标准的动画循环，它会在每一帧中清除我们的 Canvas，遍历数组中的每个球体，并将其作为参数传给 draw 函数。该函数会根据球体的速度向量更新球体的位置，使其移动，并检查球体是否出界，当发现球体出界时则将其从数组中移除。其中 Array.splice 函数的参数分别为被移除的元素在数组中的起始位置以及将被移除的元素的个数。在本示例中，我们只移除了一个元素，即当前索引所指向的元素。当你尝试运行该示例时，你几乎感觉不到有球体被移除，这是一种好的动画体验，不过你仍然可以确定没有球体在 canvas 周围游荡。

大家也许注意到本示例中的 while 循环与之前的示例中有所不同：

```
var i = balls.length;
while (i--) {
 draw(balls[i], i);
}
```

这会让 while 循环反向遍历数组而不是正向遍历。这么做是有必要的，因为当 splice 数组时，数组中剩下的元素的索引会受到影响，此时如果增加索引的值，则会在遍历中跳过下一个元素。而反向遍历则不存在此问题，只要在遍历过程中没有新的元素被添加到数组的头部。

最后，当我们将一个元素中数组中移除后，会再检查一下数组的长度是否为零，如果为零，则显示一条消息说明所有的球体已经从动画中移出。

6.1.3 重置物体

下一个处理物体离开预定义边界的方法是重置物体，更准确的说，是重新设定物体的位置

坐标。大致的思路是，当一个物体移出 canvas 且不再需要时，我们会重新设定其位置，看上去就像创建了一个全新的对象。该方法可以为 canvas 源源不断地提供运动的物体，而又不用担心 canvas 上的物体过多以至于影响浏览器的速度，因为物体的数量是固定不变的。

该技术可用于创建喷泉或其他各种粒子特效，水滴不断地飞溅出来，飞出 canvas 后并重新加入到水流的源头。

重置物体的方法跟移除物体的方法非常相似：只不过当物体移出边界后，不再是简单地移除它们，而是移动它们的位置。

现在就制作一个喷泉的动画。还是使用同样的 **Ball** 类作为喷泉的水滴，不过为其设置一个很小的半径，只有两个像素，然后再赋予它一个随机的颜色。水源在 canvas 元素底部的中心位置。每个水滴都从那里喷出，而当它们移出 canvas 后，又会再次回到水源处。每个水滴都会被赋予一个随机的负 y 轴速度向量与一个（较小的）随机的 x 轴速度向量。水滴会向上喷出并伴随轻微的左右移动，同时还受到重力的牵引。当重置水滴时，它的位置和速度向量都重新设置。文件 **02-fountain.html** 的代码如下所示：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Fountain</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
     context = canvas.getContext('2d'),
     balls = [],
     numBalls = 80,
     gravity = 0.5;

   for (var ball, i = 0; i < numBalls; i++) {
    ball = new Ball(2, Math.random() * 0xffffff);
    ball.x  = canvas.width / 2;
    ball.y  = canvas.height;
    ball.vx = Math.random() * 2 - 1;
    ball.vy = Math.random() * -10 - 10;
    balls.push(ball);
   }

   function draw (ball) {
    ball.vy += gravity;
    ball.x += ball.vx;
    ball.y += ball.vy;
    if (ball.x - ball.radius > canvas.width ||
        ball.x + ball.radius < 0 ||
```

```
        ball.y - ball.radius > canvas.height ||
        ball.y + ball.radius < 0) {
      ball.x = canvas.width / 2;
      ball.y = canvas.height;
      ball.vx = Math.random() * 2 - 1;
      ball.vy = Math.random() * -10 - 10;
    }
    ball.draw(context);
  }

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    balls.forEach(draw);
  }());
 };
 </script>
</body>
</html>
```

在代码开始的位置，为所有水滴设置运动的起点以及一个初始的、随机的、向上的速度向量。

drawFrame 函数遍历 **balls** 数组中的所有元素，将每个元素作为参数传入 **draw** 函数。该函数会将重力叠加到球体的 **vy** 属性上，再进一步将 **vy** 添加到球体的位置坐标上，最后检查球体是否越界。如果越界了，就把球体放回它的起点，再为其重新生成一个新的速度向量。这样就像新产生了一个水滴，喷泉就会不停流转了。

你应该试着体验一下这个特效，它非常的简单但是看上去的效果很不错。让喷泉向不同的方向喷射，射向一堵墙甚至是天花板。调整随机数因子让喷泉变宽或者变窄，射得更高或是更低。你甚至可以尝试加入风的效果（提示：创建一个 **wind** 变量将其添加到 **vx** 属性上）。

6.1.4 屏幕环绕

另一个处理物体越界的常用方法是屏幕环绕。思路很简单：如果一个物体从屏幕左边移出，它就会在屏幕右边再次出现；反之亦然。很容易猜到，当物体从屏幕上方移出时，它又会出现在屏幕下方。

屏幕环绕与重置比较类似，它们都会将越界的物体放回边界中再改变它们的位置。只不过在重置时，一般会统一将所有物体移回到同一个位置，让它们看上去像新产生的物体。而在屏幕环绕时，我们还是遵循同一个物体的原则，只不过是让从后门出去的物体转而出现在前门。所以，在屏幕环绕的方法中一般不会改变物体的速度。

此时，让我们再次回到那个经典游戏《行星》。第 5 章中的飞船动画中存在一个问题：飞船一旦飞出 canvas 的范围，就很难将其找回。而如果采用屏幕环绕技术，飞船就永远不会离开屏幕边缘超过一个像素了。

让我们更新飞船示例并为其加入这一行为。下面是要用到的文件（**03-ship-sim-2.html**），

其中新增的代码用粗体标出：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Ship Sim 2</title>
  <link rel="stylesheet" href="style.css">
  <style>
  #canvas {
    background-color: #000000;
  }
  </style>
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ship.js"></script>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),
       ship = new Ship(),
       vr = 0,
       vx = 0,
       vy = 0,
       thrust = 0;

   ship.x = canvas.width / 2;
   ship.y = canvas.height / 2;

   window.addEventListener('keydown', function (event) {
    switch (event.keyCode) {
    case 37:      //left
     vr = -3;
     break;
    ase 39:      //right
     vr = 3;
     break;
    case 38:      //up
     thrust = 0.05;
     ship.showFlame = true;
     break;
    }
   });

   window.addEventListener('keyup', function () {
    vr = 0;
    thrust = 0;
    ship.showFlame = false;
   });

   (function drawFrame () {
     window.requestAnimationFrame(drawFrame, canvas);
```

```
    context.clearRect(0, 0, canvas.width, canvas.height);

    ship.rotation += vr * Math.PI / 180;
    var angle = ship.rotation, //in radians
      ax = Math.cos(angle) * thrust,
      ay = Math.sin(angle) * thrust,
      left = 0,
      right = canvas.width,
      top = 0,
      bottom = canvas.height;

    vx += ax;
    vy += ay;
    ship.x += vx;
    ship.y += vy;

    //screen wrapping
    if (ship.x - ship.width / 2 > right) {
      ship.x = left - ship.width / 2;
    } else if (ship.x + ship.width / 2 < left) {
      ship.x = right + ship.width / 2;
    }
    if (ship.y - ship.height / 2 > bottom) {
      ship.y = top - ship.height / 2;
    } else if (ship.y < top - ship.height / 2) {
      ship.y = bottom + ship.height / 2;
    }
    ship.draw(context);
  }());
 };
 </script>
</body>
</html>
```

这里用到了第 5 章中的 **Ship** 类，请确保在 HTML 文件中引用了实现该类的脚本文件。可以看到，在这个更新的文件中加入了有关边界定义和检查的代码。这里还用到了多条 **if** 与 **else** 语句，因为在每个情况下动画所执行的动作不同。

6.1.5 反弹

下面将要介绍到的越界处理方法也许最常见但也最复杂。不过，你不用太担心，它并不比屏幕环绕复杂多少。

反弹方法需要检测物体何时离开屏幕，在它刚要离开时，要保持它的位置不变而仅改变它的速度向量。方法很简单：如果物体从屏幕的左边或右边越界，就对它的 x 轴速度向量取反。同理，如果物体从上方或下方越界，则对它的 y 轴速度向量取反。对单轴速度向量取反非常简单，只须将对应的值乘以–1 即可。如果速度向量为 5，则变成–5；如果是–13，则变成 13。代码就更简单了：

```
vx *= -1;
vy *= -1;
```

反弹方法与屏幕环绕相比还有些许不同。首先，在屏幕环绕中，只有当物体完全移出 canvas

后，才重置它的位置。为了实现这一点，在计算中会将物体的位置减去或加上它的自身长度的一半。在反弹方法的实现中，恰好与之相反，千万不能等物体完全移出 canvas 才开始反弹。如果你在现实世界中将一个球扔向一面墙，你不希望看到球在部分进入墙体后才发生反弹。你只会看到球在撞墙后，停在那里并很快反弹回来。因此，首先要判断出球体的任何一部分超出边界的那个瞬间。而要做到这点，只须反过来减去或加上物体自身宽度或高度的一半（见图 6-3）。比如，在判断中，不再是减去球体的半径：

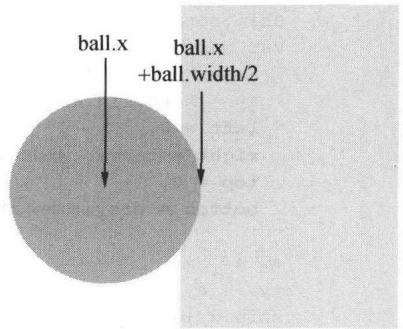


图 6-3 这里的球体刚刚越界，但它需要立即反弹

```
if (ball.x - ball.radius > right) ...
```

而是反过来加上它：

```
if (ball.x + ball.radius > right) ...
```

当检测出物体在某个坐标轴方向上略微有点越界时，就需要立即对它在这条坐标轴的速度向量取反。除此之外，还要将物体的位置置回边界，这样才能造成撞击反弹的一个非常明显的视觉效果，而不是物体陷入墙壁的效果。不仅如此，如果不能将物体的位置立刻置回边界，你会发现在下一帧，尽管物体已经开始反向移动，不过由于它可能还尚未回到边界内，因此根据之前的说法，物体的速度向量再次取反，这样物体反而会掉头钻进墙里！然后，就会出现物体陷在墙里，或进或出徘徊震荡的效果，而这并不是我们想要的。

需要将物体置回边界的那个位置正是你在 **if** 语句中检查的那个点。你只须做一些简单的代数运算就可以将其置回那个位置。以下是针对 *x* 轴的完整的 **if** 语句：

```
if (ball.x + ball.radius > right) {
  ball.x = right - ball.radius;
  vx *= -1;
} else if (ball.x - ball.radius < left) {
  ball.x = left + ball.radius;
  vx *= -1;
}
```

图 6-4 展示了球体重置后的画面。

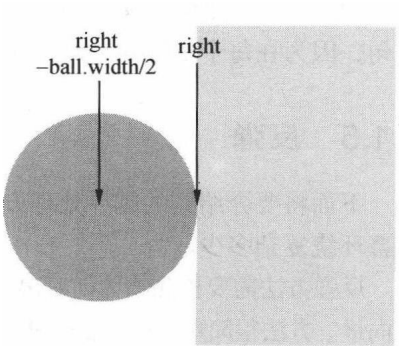


图 6-4 球体被重置回刚刚越界的位置

反弹的步骤如下：

- 检查物体是否越过任意边界；
- 如果发生越界，立即将物体置回边界；
- 反转物体的速度向量的方向。

理论部分到此为止，让我们看看真实的示例吧。下一个示例用到了同一个 **Ball** 类，不过将其大小做了适当的缩放。以下是 **04-bouncing-1.html** 文件中的代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Bouncing 1</title>
```

```
 <link rel="stylesheet" href="style.css">
</head>
<body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <script src="utils.js"></script>
 <script src="ball.js"></script>
 <script>
 window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      ball = new Ball(),
      vx = Math.random() * 10 - 5,
      vy = Math.random() * 10 - 5;

  ball.x = canvas.width / 2;
  ball.y = canvas.height / 2;

  (function drawFrame () {
     window.requestAnimationFrame(drawFrame, canvas);
     context.clearRect(0, 0, canvas.width, canvas.height);

     var left = 0,
         right = canvas.width,
         top = 0,
         bottom = canvas.height;

     ball.x += vx;
     ball.y += vy;

     if (ball.x + ball.radius > right) {
         ball.x = right - ball.radius;
         vx *= -1;
     }   else if (ball.x - ball.radius < left) {
         ball.x = left + ball.radius;
         vx *= -1;
     }
     if (ball.y + ball.radius > bottom) {
         ball.y = bottom - ball.radius;
         vy *= -1;
     }   else if (ball.y - ball.radius < top) {
         ball.y = top + ball.radius;
         vy *= -1;
     }
     ball.draw(context);
  }());
 };
 </script>
</body>
</html>
```

多运行几次这个例子，观察球体朝着不同角度的运动情况。尝试将速度变大或变小。

不得不承认，这里是本书中少数几处数学计算与现实世界的物理模型有出入的地方。在图 6-5 中，你会同时观察到在现实世界中球体与墙壁相撞的位置以及在模拟中相撞的位置。

要得到更为准确的位置就需要更复杂的计算。尽管我们完全可以做到这点（运用第 3 章中的三角学知识），但是在大多数情况人们并不会留意这么小的差别。该方法已经足以很好地服务于基于 canvas 元素实现的大多数游戏和视觉效果。不过，可以在加入一点额外的处理使得该方法与现实更加贴近。

如果我们手握一个橡胶球，然后松手使它落到地上，当小球落到地面时，会发生向上的反弹，它会反弹到接近最初的位置，但它永远不会回到我们的手中。这是因为在反弹的过程中小球损失了一部分能量。损失的能量也许转换成了声音，也可能转换成了热量，地面或周围的空气也会吸收一部分能量。重点在于球体在发生反弹后的运动速度比之前要慢一些。换句话讲，它在反弹的坐标轴上的速度向量变小了。

你可以非常容易地在动画中重现这一效果。之前，使用-1 作为弹性系数。这意味着，物体反弹的力量与撞击的力量 100%相等。可以简单地为-1 乘上一个百分比以实现能量损耗的效果。跟其他的数值参数类似，可以将这个百分比作为变量定义在代码的开头，并在以后的代码中引用它。创建一个名为 **bounce** 的变量并将其设置为-0.7：

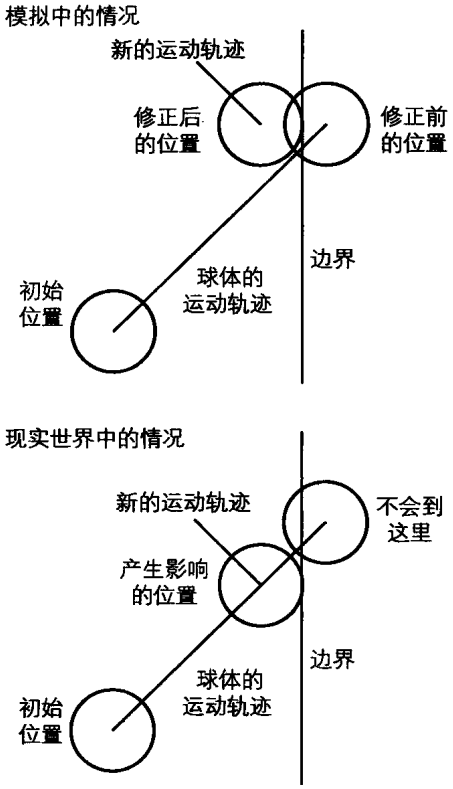


图 6-5 该方法并不完美，不过却是最快、最简单并足够接近真实的方法

```
var bounce = -0.7;
```

然后将 **if** 语句中的-1 都替换成 **bounce** 变量。试着尝试一下，你会发现反弹的效果变得更加真实了。请改变 **bounce** 变量的值以尝试不同的弹性系数，直到你能够理解它的原理。

理解这些概念的最佳方式就是将每次所学的原理与之前的那些原理结合起来。你可以在 www.apress.com 网站中找到 **05-bouncing-2.html** 示例文件，该示例同时演示了反弹与重力的效果，不过相信大家已经可以根据之前掌握的知识自行将重力的效果加到之前的示例中。或许你还想试着通过键盘为球体加入加速度或将之前所学的其他效果应用到这个示例中。

让我们先暂停对边界的讨论，去看看当物体穿过空间时会发生什么。

6.2 摩擦力

目前为止，示例都可以被以下两种情况所涵盖：

- 物体以某个速度持续前进直至撞上边界；
- 物体自身或外力作用于物体从而产生加速度，并不断变化物体的速度。

在前一种情况中，除非物体被外力推拉或撞上别的物体，否则它将保持原速度和原有的运动方向持续前进。不过这其实与现实世界中的运动是有出入的。

举个例子，假设有一张纸，将它搓成一个球后用力丢向房间外面，但是这张纸很有可能无法飞出这个房间。当然我们知道重力会将纸拉向地面，不过在纸刚离开你的手的时候，它在 x 轴的移动速度还是足够快的，之所以来不及飞出房间是因为这个速度很快就降到零了。

很明显，这里并没有负的加速度将纸拉回你的手中，但是纸的速度向量发生了改变。这个力称为**摩擦力**、**阻力**或**阻尼**。尽管它不是一种严格意义上的力，但其作用效果是相同的，它也会改变物体的速度。不过，摩擦力只会改变速度向量的大小而不会改变它的方向。换句话说，摩擦力只能将物体的速度降至零，但它无法让物体掉头向相反的方向移动。

那么如何在代码中实现摩擦力呢？有两个方法，就像生活总的很多事情一样，有一种正确方法同时还有一套简便方法。本书会更多地使用简便的方法，不过下面会同时介绍两个方法的示例，你可以根据情况作出你自己的选择。

6.2.1 摩擦力，正确方法

摩擦力是与速度向量相反的力，这意味着，可以用速度向量减去摩擦力。更准确地说，只能沿着速度向量的方向减去与摩擦力相等的大小，而不能分别在 x、y 轴上减小速度向量。这样做的话，如果物体正沿着某个角度运动，就会出现物体在某条轴上的速度降为零，而继续在另一条轴上运动的奇怪现象。

所以需要将角速度分解为速度和方向两部分（如果还不知道的话）。将 **vx** 与 **vy** 平方后求和，再开平方求出速度（是的，这里用到了勾股定律，曾经出现在第 3 章中）。再通过计算 **Math.atan2（vy,vx）**获得角度，代码如下所示：

```
var speed = Math.sqrt(vx * vx + vy * vy),
    angle = Math.atan2(vy, vx);
```

然后从速度部分中减去摩擦力，但是不要将速度变为负值，这样会逆转速度向量。如果摩擦力大于速度，则速度变为零。计算代码如下：

```
if (speed > friction) {
  speed -= friction;
} else {
  speed = 0;
}
```

此时，再通过正余弦函数将角速度转回 **vx**、**vy** 的格式：

```
vx = Math.cos(angle) * speed;
vy = Math.sin(angle) * speed;
```

下面是 **06-friction-1.html** 文件中的完整代码示例：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Friction 1</title>
```

```
 <link rel="stylesheet" href="style.css">
</head>
<body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <script src="utils.js"></script>
 <script src="ball.js"></script>
 <script>
 window.onload = function () {
  var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ball = new Ball(),
    vx = Math.random() * 10 - 5,
    vy = Math.random() * 10 - 5,
    friction = 0.1;

  ball.x = canvas.width / 2;
  ball.y = canvas.height / 2;

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    var speed = Math.sqrt(vx * vx + vy * vy),
     angle = Math.atan2(vy, vx);

    if (speed > friction) {
     speed -= friction;
    } else {
     speed = 0;
    }
    vx = Math.cos(angle) * speed;
    vy = Math.sin(angle) * speed;
    ball.x += vx;
    ball.y += vy;
    ball.draw(context);
  }());
 };
 </script>
</body>
</html>
```

这里的摩擦力设置为 0.1，并且球体在 x、y 轴上的速度是随机分配的。在 **drawFrame** 函数中，**speed** 与 **angle** 变量的计算方法同前面介绍的一样。如果 **speed** 变量大于 **friction**，二者就相减；否则，让 **speed** 变为 0。最后，重新计算 **vx**、**vy** 并将它们加到球体的位置上。

尝试用不同的速度和角度值运行该示例，你会了解摩擦力的效果是怎样的。这个例子中用到了十几行代码和 4 个三角函数才实现了摩擦力的应用。此时你一定会对那套简便方法产生兴趣了吧？

6.2.2 摩擦力，简便方法

大家也许都猜到了，模拟摩擦力的简便方法不会像之前面介绍的方法那样精确，不过其中

的细微差别普通人是很难注意到的。该方法只要两行简单的乘法组成。将 x、y 轴上的速度向量乘以一个百分数即可，一个接近 0.9 的系数就能够很好地模拟出摩擦力的效果，当然可以根据你的观察做出相应的调整。这样，在每一帧中，**vx** 和 **vy** 的值都会降为上一帧中对应值的 80% 或 90%。从理论上来看，采用这一方法，速度向量会无限接近零，而永远不会等于零。不过，实际上计算机在表达小数时只会精确到某个特定的精度，所以最终结果也会变为零。不过早在到达那个精度之前，我们就已经无法感觉到物体的移动了。

因为这一方法的好处在于速度向量永远不会变为负数，所以你无须做出任何条件判断。而且，x、y 轴的速度以同样的速率接近零，这也就避免了在轴速度与角速度之间的转换。

只须将 **friction** 变量的值设置为 0.95 并将以下改动应用到 **drawFrame** 函数中（这段代码可以在文件 **07-friction-2.html** 中找到）：

```
(function drawFrame () {
 window.requestAnimationFrame(drawFrame, canvas);
 context.clearRect(0, 0, canvas.width, canvas.height);

 vx *= friction;
 vy *= friction;
 ball.x += vx;
 ball.y += vy;
 ball.draw(context);
}());
```

现在，的确简便了不少吧！尝试运行该示例几次，体验摩擦力的效果。这里的运动跟之前“正确”的方法从视觉上根本感觉不到任何不同，而计算量却少了很多。绝大多数观众根本不会注意到它们之间的区别。

由于该方法会不断将摩擦力作用于速度向量从而改变球体的位置，因此可以通过比较 **vx** 与一个指定的较小的值从而节省一些不必要的计算，代码如下：

```
if (Math.abs(vx) > 0.001) {
 vx *= friction;
 ball.x += vx;
}
```

Math.abs 返回 vx 的绝对值，**friction** 是 0～1 之间一个任意的值，它的大小取决于你对动画的感觉。请尝试不同的值，看看它会对物体的运动产生怎样的影响。

6.2.3 摩擦力应用

让我们回到熟悉的宇宙飞船的模拟中，现在让我们将摩擦力应用到宇宙空间中。在示例 **08-ship-sim-friction.html** 文件中使用了 **03-ship-sim-2.html** 文件中的代码并加入了一个摩擦力变量：

```
var friction = 0.97;
```

接着，将 **drawFrame** 函数修改为如下代码：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
```

```
    context.clearRect(0, 0, canvas.width, canvas.height);

    ship.rotation += vr * Math.PI / 180;
    var angle = ship.rotation,
        ax = Math.cos(angle) * thrust,
        ay = Math.sin(angle) * thrust,
        left = 0,
        right = canvas.width,
        top = 0,
        bottom = canvas.height;
    vx += ax;
    vy += ay;
    vx *= friction;
    vy *= friction;
    ship.x += vx;
    ship.y += vy;

    //screen wrapping
    if (ship.x - ship.width / 2 > right) {
     ship.x = left - ship.width / 2;
    } else if (ship.x + ship.width / 2 < left) {
     ship.x = right + ship.width / 2;
    }
    if (ship.y - ship.height / 2 > bottom) {
     ship.y = top - ship.height / 2;
    } else if (ship.y < top - ship.height / 2) {
     ship.y = bottom + ship.height / 2;
    }
    ship.draw(context);
}());
```

仅仅加了3行代码，就为我们带来了不同的感觉。

不要局限在 x、y 轴的框框中，其实摩擦力可以应用到任何一个存在速度向量的地方。比如，在一个旋转的物体上应用摩擦力（通过 vr 属性），会使其转速变慢并最终停止旋转。你可以在第5章的旋转箭头示例中尝试一下。

该方法适用于任何物体，比如，轮盘、电风扇或螺旋桨。

6.3 本章中的重要公式

让我们来回顾本章中出现的重要公式。

6.3.1 移除越界物体

```
if (object.x - object.width / 2 > right ||
    object.x + object.width / 2 < left ||
    object.y - object.height / 2 > bottom ||
    object.y + object.height / 2 < top) {
  //code to remove object
}
```

6.3.2 重置越界物体

```
if (object.x - object.width / 2 > right ||
    object.x + object.width / 2 < left ||
    object.y - object.height / 2 > bottom ||
    object.y + object.height / 2 < top) {
  //reset object position and velocity
}
```

6.3.3 越界物体的屏幕环绕

```
if (object.x - object.width / 2 > right) {
  object.x = left - object.width / 2;
} else if (object.x + object.width / 2 < left) {
  object.x = right + object.width / 2;
} if (object.y - object.height / 2 > bottom) {
  object.y = top - object.height / 2;
} else if (object.y + object.height / 2 < top) {
  object.y = bottom + object.height / 2;
}
```

6.3.4 应用摩擦力（正确方法）

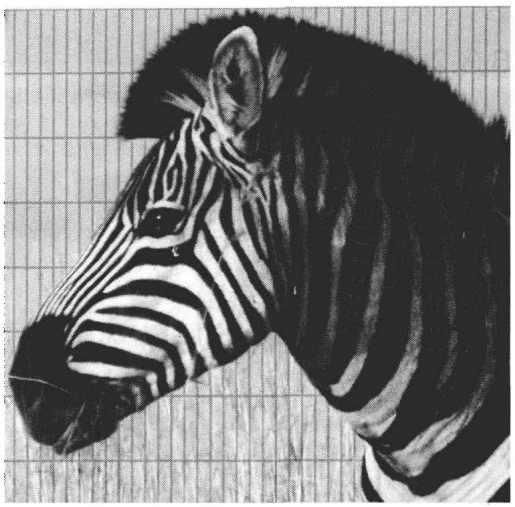
```
speed = Math.sqrt(vx * vx + vy * vy);
angle = Math.atan2(vy, vx);
if (speed > friction) {
  speed -= friction;
} else {
  speed = 0;
}
vx = Math.cos(angle) * speed;
vy = Math.sin(angle) * speed;
```

6.3.5 应用摩擦力（简便方法）

```
vx *= friction;
vy *= friction;
```

6.4 小结

本章介绍了物体如何与它的环境发生交互，确切地说，物体如何与它的边界以及边界内的物质交互。你学习了包括移除、重置、屏幕环绕与反弹在内的各种处理物体越界的方法。并且，你还了解了如何将摩擦力作用于物体。借助这个简单的技术，你可以让动画中的物体运动得更加逼真。在第 7 章中，你会学到如何实现用户与物体间的交互。



第 7 章　用户交互：移动物体

本章涵盖以下内容：

- 按下及释放物体
- 拖曳物体
- 投掷物体

交互动画的一个主要目标是创建出流畅的用户体验，其中大多数的用户交互都是通过鼠标和触摸屏实现的。第 2 章曾经介绍过鼠标和触摸事件，只不过在之前的示例中都没有怎么用过，下面我们会亲身体验一下。

本章将踏出交互动画的第一步。我们将学习如何处理拖曳、抛落及投掷。不过，让我们先从基本的按下与释放说起。

7.1　按下及释放物体

鼠标是一个非常有用却又非常简单的设备。它只负责两件事件，检测自身的移动以及按钮是否单击，而计算机又可以基于这些信息做很多事情：追踪鼠标指针的位置，确定鼠标按钮被按下时指针的位置，计算出鼠标的移动速度以及确定何时发生双击事件。不过从事件的角度来看，以上所做的事情都可以简单归结为鼠标的单击和移动。

你也可以将一次单击事件分解成两个事件：鼠标键按下的事件及按键弹起的事件。通常情况下这两个事件是同时发生的。不过，有时候鼠标键按下后鼠标还会移动一段时间，这种操作称为拖曳，即按下、移动、再释放。本章将围绕下面 3 个操作展开：鼠标按下、鼠标弹起，以

及发生在两者之间的鼠标移动。

在动画中，鼠标事件只能被 HTML DOM 树上的 canvas 元素所捕获。因此，我们需要手动计算鼠标事件在 canvas 上的发生位置并判断出它发生在哪个绘制到 canvas 上的物体上。第 2 章曾经展示了如何使用工具函数 **utils.captureMouse** 追踪鼠标的位置。下面，我们会留意发生在 canvas 上与鼠标相关的各种事件，并在这些事件发生时检查鼠标所处的位置是否位于某个物体的边界内。需要关注的鼠标事件如下。

- **mousedown**：当鼠标指针位于某个 HTML 元素上方时按下鼠标按钮触发该事件。在该示例中，这是 **Canvas** 元素。
- **mouseup**：当鼠标指针位于某个 HTML 元素上方时释放鼠标按钮触发该事件。
- **mousemove**：当鼠标指针位于某个 HTML 元素上方时移动鼠标触发该事件。

有时候你可能只关注鼠标事件而不关心与之相关的 HTML 元素，在这种情况下，可以直接将事件监听器添加到全局的 **window** 对象中而不是 **canvas** 对象中。

让我们看看是否可以在动画中使用鼠标事件。在下一个示例中，会用到之前章节中出现的 **Ball** 类，不过我们会为其添加一个新的方法 **Ball.getBounds**。该方法返回一个恰好能够容纳小球的矩形，该矩形区域也称为小球的**边界**。返回的对象拥有 **x**、**y**、**width** 与 **height** 属性，这些属性的值都是经由小球所处的位置以及它的半径计算出来的。下面是更新后的 ball.js 文件：

```
function Ball (radius, color) {
 if (radius === undefined) { radius = 40; }
 if (color === undefined) { color = "#ff0000"; }
 this.x = 0;
 this.y = 0;
 this.radius = radius;
 this.vx = 0;
 this.vy = 0;
 this.rotation = 0;
 this.scaleX = 1;
 this.scaleY = 1;
 this.color = utils.parseColor(color);
 this.lineWidth = 1;
}

Ball.prototype.draw = function (context) {
 context.save();
 context.translate(this.x, this.y);
 context.rotate(this.rotation);
 context.scale(this.scaleX, this.scaleY);
 context.lineWidth = this.lineWidth;
 context.fillStyle = this.color;
 context.beginPath();
 context.arc(0, 0, this.radius, 0, (Math.PI * 2), true);
 context.closePath();
 context.fill();
 if (this.lineWidth > 0) {
  context.stroke();
 }
 context.restore();
```

```
};

Ball.prototype.getBounds = function () {
 return {
  x: this.x - this.radius,
  y: this.y - this.radius,
  width: this.radius * 2,
  height: this.radius * 2
 };
};
```

同时我们也会在即将导入 HTML 文件中的 **utils.js** 文件中新增一个工具函数 **utils.containsPoint**。该函数定义了三个参数，第一个参数接受一个与 **Ball.getBounds()**方法返回值一样拥有 **x**、**y**、**width** 与 **height** 属性的矩形对象，第二个与第三个参数则分别代表一个坐标位置在 x、y 轴上的值。**utils.containsPoint** 函数会通过返回值 **true** 与 **false** 决定一个指定的坐标位置是否位于矩形的边界内。下面就是这个即将加入 **utils.js** 文件中的简单却实用的函数的实现：

```
utils.containsPoint = function (rect, x, y) {
  return !(x < rect.x || x > rect.x + rect.width ||
           y < rect.y || y > rect.y + rect.height);
};
```

下面是用到了更新后的 **Ball** 类与 **utils.containsPoint** 函数的完整示例，该实例位于 **01-mouse-events.html** 文件中：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Mouse Events</title>
 <link rel="stylesheet" href="style.css">
 </head>
 <body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <textarea id="log"></textarea>
 <script src="utils.js"></script>
 <script src="ball.js"></script>
 <script>
 window.onload = function () {
  var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    mouse = utils.captureMouse(canvas),
    log = document.getElementById('log'),
    ball = new Ball();

  ball.x = canvas.width / 2;
  ball.y = canvas.height / 2;
  ball.draw(context);

  canvas.addEventListener('mousedown', function () {
   if (utils.containsPoint(ball.getBounds(), mouse.x, mouse.y)) {
     log.value = "in ball: mousedown";
   } else {
     log.value = "canvas: mousedown";
   }
```

```
}, false);

canvas.addEventListener('mouseup', function () {
 if (utils.containsPoint(ball.getBounds(), mouse.x, mouse.y)) {
   log.value = "in ball: mouseup";
 } else {
   log.value = "canvas: mouseup";
 }
}, false);

canvas.addEventListener('mousemove', function () {
 if (utils.containsPoint(ball.getBounds(), mouse.x, mouse.y)) {
   log.value = "in ball: mousemove";
 } else {
   log.value = "canvas: mousemove";
 }
}, false);
};
</script>
</body>
</html>
```

以上代码为 canvas 元素添加了之前介绍到的三个鼠标事件：**mousedown**、**mouseup** 与 **mousemove**。在检测到一个鼠标事件后，事件处理函数会通过 **Ball.getBounds** 与 **utils.containsPoint** 函数检查鼠标坐标是否位于小球的边界内并输出结果。试着运行该示例，观察何时会触发某个特定的鼠标事件以及发生的位置。请同时留意以下几点。

- 无论鼠标在哪里，即便它不在小球上，canvas 上的事件始终会触发。
- **mouseup** 事件始终发生在 **mousedown** 事件之后。

7.1.1 使用触摸事件

在本书的示例中，大量使用了鼠标作为用户输入设备，这基于一个假设，那就是我们使用一台配备了键盘和鼠标的电脑作为开发机器。不过，伴随着触摸屏设备的流行，我们很有可能需要在动画中捕捉用户的触摸事件。尽管触摸屏与鼠标是不同的设备，但幸运的是，在 DOM 树上捕捉触摸事件与捕捉鼠标事件并没有太大差别。

能够与本书中示例兼容的触摸事件及其鼠标事件对应的触摸事件分别为 **touchstart**、**touchend** 与 **touchmove**。**touchstart** 事件在第一次触摸时触发，**touchend** 在一个手指离开触摸屏后触发，而当手指在屏幕上拖动时则会触发 **touchmove** 事件。关于触摸事件的更多详细内容请浏览位于 http://www.w3.org/TR/touch-events/的 W3C 规范。在本书编写时，该文件还尚未完成，不过其中大多数事件已被很多移动 Web 浏览器所支持。如果对该规范有任何疑问，请直接到浏览器上测试一下。

使用手指与鼠标的一个重大区别在于，鼠标指针始终出现在屏幕上，而手指却并不是一直处在触摸状态。在第 2 章中，曾经创建了一个工具函数 **utils.captureTouch**，该函数会追踪触摸发生的位置。该函数的返回值还包含一个 **isPressed** 属性，它会告诉我们屏幕上是否有手指在触摸。当修改示例以支持触摸输入时，应该首先检查是否有触摸发生；否则，访问触摸位置属性时会得到空值：

```
var touch = utils.captureTouch(canvas);

if (touch.isPressed) {
  console.log(touch.x, touch.y);
}
```

下一个示例 **02-touch-events.html** 会基于本章的第一个示例，将其修改成一个处理触摸事件的示例（新增的代码会以粗体显示）。为了让示例能够正常工作，请确保你在一个能够处理这些类事件的 Web 浏览器中运行该示例（在一个触摸屏上或一个模拟器上）。

在 HTML 文件的头部，加入了一个 meta 标签用于设置 viewport。这会告知浏览器如何将内容呈现在设备的屏幕上，并优化该页面使其便于在移动设备上进行查看。

```
<meta name="viewport" content="width=device-width,initial-scale=1.0,user-scalable=no">
```

此处，viewport 被设置为与设备等宽并且初始的缩放比例被设为 1。用户还被禁止对本文件进行缩放。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <meta name = "viewport" content="width=device-width,initial-scale=1.0,user-scalable=no">
  <title>Touch Events</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <textarea id="log"></textarea>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        touch = utils.captureTouch(canvas),
        log = document.getElementById('log'),
        ball = new Ball();

    ball.x = canvas.width / 2;
    ball.y = canvas.height / 2;
    ball.draw(context);
    canvas.addEventListener('touchstart', function (event) {
     event.preventDefault();
     if (utils.containsPoint(ball.getBounds(), touch.x, touch.y)) {
       log.value = "in ball: touchstart";
     } else {
       log.value = "canvas: touchstart";
     }
    }, false);

    canvas.addEventListener('touchend', function (event) {
     event.preventDefault();
     log.value = "canvas: touchend";
    }, false);
```

```
    canvas.addEventListener('touchmove', function (event) {
     event.preventDefault();
     if (utils.containsPoint(ball.getBounds(), touch.x, touch.y)) {
       log.value = "in ball: touchmove";
     } else {
       log.value = "canvas: touchmove";
     }
    }, false);
  };
  </script>
 </body>
</html>
```

在触摸事件的监听器中，将 event 对象传给事件处理函数并调用 event.preventDefault()方法。这会让浏览器在事件处理函数执行完后继续处理该事件。同时它还会阻止与触摸事件对应的鼠标事件的触发，这是该方法的默认行为，不过最好不要太依赖这一特性。

现在你应该掌握了重要交互事件的基础，下面让我们开始学习拖曳。

7.2 拖曳对象

通过不断更新物体的坐标位置使其追随鼠标指针的位置，就可以实现在 canvas 元素上拖曳物体。实现代码包含以下过程：捕捉 **mousedown** 事件，当鼠标单击小球时为 canvas 添加 **mousemove** 事件的处理程序，并在该处理程序内将小球的 x、y 坐标更新为鼠标指针当前的坐标位置。最后，在 **mouseup** 事件触发时，将 **mousemove** 事件的处理程序移除。

阅读以下的示例文件 **03-mouse-move-drag.html**，可以更加清楚这一过程：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Mouse Move Drag</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),
       mouse = utils.captureMouse(canvas),
       ball = new Ball();

  ball.x = canvas.width / 2;
  ball.y = canvas.height / 2;

  canvas.addEventListener('mousedown', function () {
    if (utils.containsPoint(ball.getBounds(), mouse.x, mouse.y)) {
     canvas.addEventListener('mouseup', onMouseUp, false);
```

```
    canvas.addEventListener('mousemove', onMouseMove, false);
  }
}, false);

function onMouseUp () {
 canvas.removeEventListener('mouseup', onMouseUp, false);
 canvas.removeEventListener('mousemove', onMouseMove, false);
}

function onMouseMove (event) {
 ball.x = mouse.x;
 ball.y = mouse.y;
}

(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  ball.draw(context);
}());
};
 </script>
</body>
</html>
```

在脚本刚开始执行时，只监听 **mousedown** 事件，并查看发生该事件时鼠标是否刚好击中小球。当调用时，该事件处理程序向 **mouseup** 和 **mousemove** 事件的 canvas 元素添加额外的监听器。**OnMouseMove** 函数负责将小球的位置更新为鼠标所在的位置。**onMouseUp** 函数则会将 **mouseup** 与 **mousemove** 的事件监听器从 canvas 上移除掉，因为只有在拖曳小球时，我们才有必要处理这些事件。

从这些设置中，也许大家已经发现了一个问题。当单击小球的边缘并拖曳小球时，你会看到小球的中心点突然跳到了鼠标光标的位置上。这是因为我们将小球的位置设置为与鼠标位置完全相等，实际上为了解决这个问题，你可以计算出当 **mousedown** 事件发生时鼠标单击位置与小球中心位置的偏移量，并在拖曳小球时加上这个偏移量。不过我们不会在本书中实现这一细节，而是把它留作课后练习。

7.2.1 结合运动代码的拖曳

目前为止，我们已经非常清楚如何在 canvas 上实现简单的拖放。不过在此过程中，我们的动画反而倒退了，变得更简单了。此时，除非拖曳物体，否则它将一直保持静止。我们可以尝试为该动画引入一些速度、加速度以及反弹的内容，使其变得更加有趣。

因为我们在示例文件 **05-bouncing-2.html** 中已经实现了速度、重力加速度与反弹，所以我们可以非常合理地在此基础上加入拖放的代码。让我们尝试一下，最终的结果应该与以下代码（**04-drag-and-move-1.html**）非常接近：

```
<!doctype html>
<html>
 <head>
```

```
<meta charset="utf-8">
<title>Drag and Move 1</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
<script src="utils.js"></script>
<script src="ball.js"></script>
<script>
window.onload = function () {
 var canvas = document.getElementById('canvas'),
   context = canvas.getContext('2d'),
   mouse = utils.captureMouse(canvas),
   ball = new Ball(),
   vx = Math.random() * 10 - 5,
   vy = -10,
   bounce = -0.7,
   gravity = 0.2,
   isMouseDown = false;

ball.x = canvas.width / 2;
ball.y = canvas.height / 2;

canvas.addEventListener('mousedown', function () {
 if (utils.containsPoint(ball.getBounds(), mouse.x, mouse.y)) {
  isMouseDown = true;
  canvas.addEventListener('mouseup', onMouseUp, false);
  canvas.addEventListener('mousemove', onMouseMove, false);
 }
}, false);

function onMouseUp () {
 isMouseDown = false;
 canvas.removeEventListener('mouseup', onMouseUp, false);
 canvas.removeEventListener('mousemove', onMouseMove, false);
}

function onMouseMove (event) {
 ball.x = mouse.x;
 ball.y = mouse.y;
}

function checkBoundaries () {
 var left = 0,
     right = canvas.width,
     top = 0,
     bottom = canvas.height;

 vy += gravity;
 ball.x += vx;
 ball.y += vy;

 //boundary detect and bounce
 if (ball.x + ball.radius > right) {
  ball.x = right - ball.radius;
```

```
    vx *= bounce;
   } else if (ball.x - ball.radius < left) {
    ball.x = left + ball.radius;
    vx *= bounce;
   }
   if (ball.y + ball.radius > bottom) {
    ball.y = bottom ball.radius;
    vy *= bounce;
   } else if (ball.y - ball.radius < top) {
    ball.y = top + ball.radius;
    vy *= bounce;
   }
  }

  (function drawFrame () {
   window.requestAnimationFrame(drawFrame, canvas);
   context.clearRect(0, 0, canvas.width, canvas.height);

   if (!isMouseDown) {
    checkBoundaries();
   }
   ball.draw(context);
  }());
 };
 </script>
</body>
</html>
```

正如你所见，以上代码中为各种鼠标事件添加了对应的处理程序，并且所有有关边界检查的代码都移入了 **checkBoundaries** 函数。这里还声明了一个名为 **isMouseDown** 的变量，我们将通过由处器程序更新它来追踪鼠标按钮的状态。只有当鼠标没有按下的时候，才会执行边界检查。

试着运行该示例，你会发现在停止拖曳时存在一个问题。是的，拖曳确实能工作，不过在拖曳的过程还伴随着物体的运动。为了避免在拖曳时产生不必要的运动，需要一些方法来开启或关闭让物体产生运动的代码。

当你开始拖曳物体时，我们希望物体不要伴随有其他运动。而当你放下小球时，我们又希望小球能够继续它在被拖曳前的运动。不过，这里有一个很大的问题在于，当小球被放下后，它仍会保持之前的运动速度，这就会在某些时候造成小球被放下后突然向某个方向飞出的现象，这会显得很不自然。为了解决这一问题，只须简单地将小球的 **vx** 与 **vy** 设置为零，可以任意选择在拖曳开始时或结束时作出这一设置，只要在恢复物体的运动之前即可。下面选择将这一设置放入 **mousedown** 事件的处理程序中：

```
canvas.addEventListener('mousedown', function () {
 if (utils.containsPoint(ball.getBounds(), mouse.x, mouse.y)) {
  isMouseDown = true;
  vx = vy = 0;
  canvas.addEventListener('mouseup', onMouseUp, false);
  canvas.addEventListener('mousemove', onMouseMove, false);
 }
}, false);
```

此时问题基本得以解决，我们获得了一个具备完整拖放功能并且集成了速度、加速度与反弹内容的示例。你可以在 **05-drag-and-move-2.html** 文件中找到完整的示例代码。

还有一个小问题，当你放下小球时，它会垂直下落而没有 *x* 轴方向的速度。尽管这么做并没有什么问题，但是导致动画稍显生硬。如果你可以在放下小球时将其扔向某个方向，使其向着那个方向运动，就能获得一种更好的交互体验。

7.3 投掷

在动画中如何表现投掷呢？用鼠标选中一个物体，拖曳着它向某个方向移动，松开鼠标后物体沿着拖曳的方向继续移动。

在投掷物体时，必须在拖曳物体的过程中计算物体的速度向量，并在释放物体时将这个速度向量赋给物体。举个例子，如果你以每帧 10 个像素的速度向左拖曳小球，那么在你释放小球时，它的速度向量应该是 **vx=-10**。

设置速度向量对大家来说没有什么问题。只须将新的值赋给物体的 **vx** 与 **vy** 属性，如图 7-1 所示。不过，如何计算出这两个值却需要一点小技巧，实际上，计算拖曳时物体的速度向量的过程恰好与对物体应用速度向量的过程相反。在对物体应用速度向量时，将速度向量追加到物体原来所在的位置上从而计算出物体的新位置。这个公式可以写成：**旧的位置+速度向量=新的位置**。而想要计算出物体被拖曳时的速度向量，只须将这个等式做一个简单的变形：**速度向量=新的位置-旧的位置**。

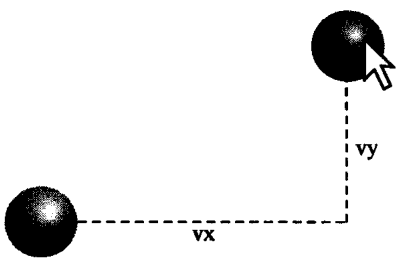


图 7-1 小球被拖曳到新的位置。速度向量等于新位置减去旧位置的距离

在拖曳物体时，它会在每一帧拥有一个新的位置。用当前帧的位置加以上一帧的位置，就可以计算出在这一帧所移动的距离。这就是每帧移动像素的速度向量值！

让我们尝试一个例子，将问题简化为在一条轴上拖曳小球。在某一帧上，我们观察到小球的 *x* 轴坐标为 150，在下一帧上，它的 *x* 轴坐标变为 170。因此，在这一帧内，它在 *x* 轴上移动了 20 个像素，所以它的 *x* 轴速度向量为+20。如果在此时释放该小球，它就会在 *x* 轴上以每帧 20 个像素的速度继续前进，也就是说小球的 **vx=20**。

为了实现以上行为，需要对前面的代码做一些改动。首先，在 **drawFrame** 函数中检查鼠标是否按下，如果按下，调用 **trackVelocity** 函数更新小球的拖曳速度。此时，需要用 **oldX** 与 **oldY** 变量保存小球旧的 *x*、*y* 坐标位置，它们会声明在代码最开始的地方，并且一旦开始对小球进行拖曳，就会把小球的位置保存在这两个变量中。

```
canvas.addEventListener('mousedown', function () {
  if (utils.containsPoint(ball.getBounds(), mouse.x, mouse.y)) {
    isMouseDown = true;
    oldX = ball.x;
```

```
        oldY = ball.y;
        canvas.addEventListener('mouseup', onMouseUp, false);
        canvas.addEventListener('mousemove', onMouseMove, false);
        }
    }, false);

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      if (isMouseDown) {
      trackVelocity();
    } else {
      checkBoundaries();
      }
      ball.draw(context);
}());
```

然后，在 **trackVelocity** 函数中，用小球当前的 x、y 轴坐标分别减去 **oldX** 与 **oldY**，从而获得小球当前的速度向量，并将它们保存在 **vx** 与 **vy** 变量中。最后将 **oldX** 与 **oldY** 变量更新为小球当前的位置：

```
function trackVelocity () {
  vx = ball.x - oldX;
  vy = ball.y - oldY;
  oldX = ball.x;
  oldY = ball.y;
}
```

此时，无须用到这个速度向量。代码会确保在拖曳小球时这个速度向量得到更新，这样 **vx** 与 **vy** 就可以始终保存着最新的速度向量。一旦在 **checkBoundaries** 函数中恢复了小球了移动，它就会以最新的那个速度向量继续移动，从而实现了投掷小球的效果！

下面列出了完整的示例代码（文件 **06-throwing.html**），以免你有所遗漏：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Throwing</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        ball = new Ball(),
        vx = Math.random() * 10 - 5,
        vy = -10,
        bounce = -0.7,
```

```
    gravity = 0.2,
    isMouseDown = false,
    oldX, oldY;

ball.x = canvas.width / 2;
ball.y = canvas.height / 2;

canvas.addEventListener('mousedown', function () {
  if (utils.containsPoint(ball.getBounds(), mouse.x, mouse.y)) {
      isMouseDown = true;
      oldX = ball.x;
      oldY = ball.y;
      canvas.addEventListener('mouseup', onMouseUp, false);
      canvas.addEventListener('mousemove', onMouseMove, false);
  }
}, false);

function onMouseUp () {
  isMouseDown = false;
  canvas.removeEventListener('mouseup', onMouseUp, false);
  canvas.removeEventListener('mousemove', onMouseMove, false);
}

function onMouseMove (event) {
  ball.x = mouse.x;
  ball.y = mouse.y;
}
function trackVelocity () {
  vx = ball.x - oldX;
  vy = ball.y - oldY;
  oldX = ball.x;
  oldY = ball.y;
}

function checkBoundaries () {
  var left = 0,
      right = canvas.width,
      top = 0,
      bottom = canvas.height;

  vy += gravity;
  ball.x += vx;
  ball.y += vy;

  //boundary detect and bounce
  if (ball.x + ball.radius > right) {
      ball.x = right - ball.radius;
      vx *= bounce;
  } else if (ball.x - ball.radius < left) {
      ball.x = left + ball.radius;
      vx *= bounce;
  }
  if (ball.y + ball.radius > bottom) {
      ball.y = bottom - ball.radius;
```

```
        vy *= bounce;
      } else if (ball.y - ball.radius < top) {
        ball.y = top + ball.radius;
        vy *= bounce;
      }
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      if (isMouseDown) {
          trackVelocity();
      } else {
         checkBoundaries();
      }
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

此时我们获得了一个使用 JavaScript 以及事件处理函数实现的、一个非常好的模拟现实中物理现象的交互式动画。它让你体验到在动画中投掷物体的感觉。试着改变 **gravity** 与 **bounce** 变量的值，你甚至可以加入一些摩擦力以模拟空气阻力。

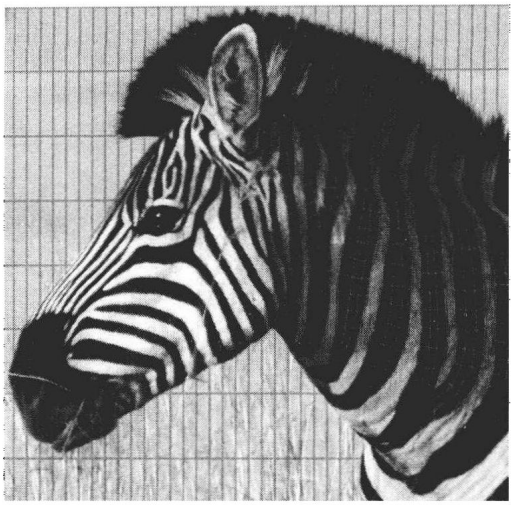
7.4 小结

这一节的内容并不多，但是它涉及了一些非常有价值的基础知识，并进一步介绍了动画中的交互行为。此时，你应该可以在动画中实现对物体的拖曳、释放以及投掷。

最重要的是，你在此过程中处理了许多小的细节从而完成了一个非常专业的交互式动画。在后续章节中，你会学到更多让用户与物体在动画中产生交互的方法。它们会变得越来越复杂，不过当你掌握了这些基础知识后，你就能够非常轻松地掌握它们。

第三部分

高 级 动 画



第 8 章　缓动与弹动

本章涵盖以下内容：

- 比例运动（proportional motion）
- 缓动（easing）
- 弹动（springing）

很难相信我们用了前面 7 章来打通基础，现在终于到第 8 章了。前面的章节主要介绍了一些通用的技术和概念。但是从本章开始，我们开始介绍高级动画，每个章节将专注于介绍一两个案例。你会发现阅读下去会变得越来越有趣。

本章将介绍缓动（比例速度）和弹动（比例加速度）。不必深究这两个词的意思，这不过是两个专业术语。你可以快速地浏览本章一遍，本章所介绍的技术可以用来构建非常复杂的动画，将来肯定会经常用到。这里会有不少示例，通过这些例子，你就能知道这些技术是多么强大。

8.1　比例运动

缓动和弹动关系紧密，这两种技术都是把对象从已有位置移动到目标位置的方法。缓动是指物体滑动到目标点就停下来了。弹动是指物体来回地反弹一会儿，最终停在目标点的运动。这两种技术有以下几个共同点。

- 需要设定一个目标点。
- 需要确定物体到目标点的距离。
- 运动和距离是成正比的——距离越远，运动的程度越大。

缓动和弹动的不同点是运动和距离成正比的方面（或者方式）不一样。缓动是指速度与距离成正比；物体离目标点越远，物体运动的速度越快。当物体运动到很接近目标点的时候，它几乎就停下来了。

在弹动的时候，随着距离成比例变化的是加速度。如果物体离目标点很远，加速度就很大，速度就会快速增大。当物体很接近目标点的时候，加速度变得很小，但是它还是在加速的！当它越过目标点之后，随着距离的变大，反向加速度也随之变大，就会把它拉回来。最终在摩擦力的作用下停住。

让我们分别学习一下这两种技术，先从缓动开始。

8.2 缓动

缓动的类型并不止一种，你可以“缓入”（ease in）到一个位置，也可以从一个位置“缓出”（ease out）。另外，缓动可以有一些不同的运动特征，比如，正弦波、跳跃、弹性等。这里主要讨论“缓出”，8.2.2 小节将会告诉你去哪里学习一些其他类型的缓动。

8.2.1 简单缓动

为了更好地理解简单缓动，想象一下你要把物体从一个地方移动到另一个地方。因为运动发生在你的幻想之中，所以你可以渐渐地，一帧一帧地移动这个物体。首先找出物体与目标点的角度，设定一个速度，用三角函数计算出 vx 和 vy，并把它应用在物体的移动上。每移动一帧，重新计算一次物体到目标的距离（用第 3 章里面学到的勾股定理），当它到达目标点时就停下来。这种方法只在某些情况下有用，但是如果你想要一个很自然的运动效果，这种方法显然不行。

这个方法的问题是，物体以一个固定的速度在移动，当到达目标点时，突然停止。如果你想要表现物体笔直撞墙的效果，这种方法可能还有点意思。但是，当把一个物体向目标点移动的时候，这通常意味着，你知道目标点在哪里，并且很谨慎地移动物体。在这种情况下，你应该会一开始移动得很快，越接近目标就越慢。换句话说，它的速度和它与目标点的距离成正比。

让我们来看一个例子：你正在开车回家，当你离家还有几英里的时候，你会开得尽可能快（限速之内）。当你从高速路上下来，进入社区的时候，就会慢一些。当你开到自己的那条街道，还有一两个街区就要到家的时候，会更慢一些。当你快到家门口的时候，你会减速到每小时几英里。当你快到车库门口的时候，你会再慢一些。当车子还有几英寸就要停下来的时候，速度已经非常非常慢了。

如果你花点时间想想，就会发现在关门或关抽屉这样的小事中也能体现出这种行为。一开始比较快，然后逐渐慢下来。下次关门的时候，你可以试试保持速度不变，恐怕你不得不向周围的人解释为什么摔门吧。当使用“缓动”的方式来移动一个物体到某个点的时候，它就会显得很自然。最酷的是，简单缓动非常易于实现。实际上，它比需要计算角度、vx 和 vy，匀速运动还要简单。

以下是实现缓动的策略。

- 为运动确定一个比例系数，这是一个小于 1 且大于 0 的小数。

- 确定目标点。
- 计算出物体与目标点的距离。
- 计算速度，速度=距离 × 比例系数。
- 用当前位置加上速度来计算新的位置。
- 重复第 3 步到第 5 步，直到物体到达目标。

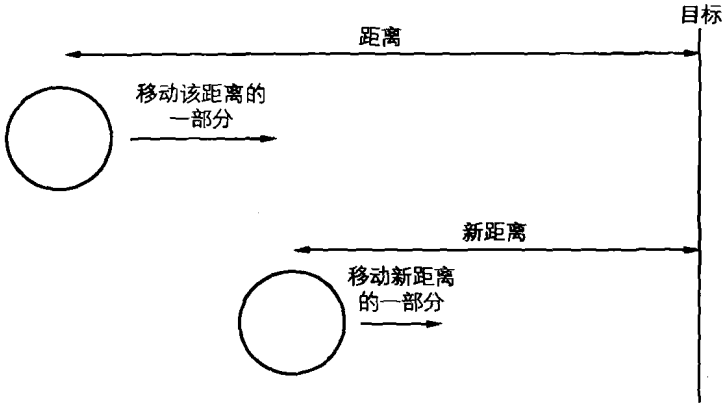


图 8-1 简单缓动

我们来逐步地看一下这个过程，然后考虑如何用程序实现。先别担心怎么一下子把程序写出来，先来看看一些代码片段和它们的含义。

首先，确定一个小数作为比例系数，正如前面提到的，速度与运动成正比。更具体一点，用速度除以距离就能得到这个系数，这个系数是一个大于 0 小于 1 的数。当系数越接近于 1，物体移动得越快；当系数越接近于 0，物体移动得越慢。但要注意的是，如果这个系数太小，你就要等很久物体才能到达目标点。首先，把这个系数设置为 0.05，用变量 **easing** 表示。代码如下：

```
var easing = 0.05;
```

下一步要确定目标点，这是一个简单的坐标点。使用 canvas 元素的中心点最合适不过了。

```
var targetX = canvas.width / 2,
    targetY = canvas.height / 2;
```

然后计算物体到目标点的距离。假设这个物体是一个小球，变量名为 ball，用 ball 的 x、y 减去目标点的 x、y 就能得到距离。

```
var dx = targetX - ball.x,
    dy = targetY - ball.y;
```

速度就是距离乘以系数。

```
var vx = dx * easing,
    vy = dy * easing;

ball.x += vx;
ball.y += vy;
```

因为最后几步需要重复执行，所以会把这些代码放到 **drawFrame** 函数里面。让我们再斟酌一下这些步骤，因为它们还可以大幅地简化。

```
var dx = targetX - ball.x,
    dy = targetY - ball.y,
    vx = dx * easing,
    vy = dy * easing;

ball.x += vx;
ball.y += vy;
```

可以很容易地把前 4 行合并为两行：

```
var vx = (targetX - ball.x) * easing,
    vy = (targetY - ball.y) * easing;

ball.x += vx;
ball.y += vy;
```

或者还可以进一步简化：

```
ball.x += (targetX - ball.x) * easing;
ball.y += (targetY - ball.y) * easing;
```

在一开始接触缓动时，你可能喜欢多写几个步骤使得代码看起来更清楚。但是一旦你理解了缓动如何工作之后，上面第三个版本就是最好的。这里仍然使用第二个版本，以巩固对速度的理解。

现在我们来看看实际的例子，还是用前面的 Ball 类来举例，如 01-easing-1.html 所示：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Easing 1</title>
  <link rel="stylesheet" href="style.css">
</head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        easing = 0.05,
        targetX = canvas.width / 2,
        targetY = canvas.height / 2;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var vx = (targetX - ball.x) * easing,
          vy = (targetY - ball.y) * easing;

      ball.x += vx;
      ball.y += vy;
```

```
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

试着调整变量 **easing** 的值，看看会对结果产生什么影响。

接下来你可能想随意拖动这个小球，当你松开它的时候，它就会缓动回目标位置。这和第 7 章介绍的“拖放”技术很相似。这里，当小球被鼠标按住时开始拖动。在 drawFrame 函数里检查小球是否拖动了，如果拖动，则执行缓动计算。下面给出示例 02-easing-2.html：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Easing 2</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        ball = new Ball(),
        easing = 0.05,
        targetX = canvas.width / 2,
        targetY = canvas.height / 2,
        isMouseDown = false;

    canvas.addEventListener('mousedown', function () {
      if (utils.containsPoint(ball.getBounds(), mouse.x, mouse.y)) {
        isMouseDown = true;
        canvas.addEventListener('mouseup', onMouseUp, false);
        canvas.addEventListener('mousemove', onMouseMove, false);
      }
    }, false);

    function onMouseUp () {
      isMouseDown = false;
      canvas.removeEventListener('mouseup', onMouseUp, false);
      canvas.removeEventListener('mousemove', onMouseMove, false);
    }

    function onMouseMove () {
      ball.x = mouse.x;
      ball.y = mouse.y;
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);
```

```
        if (!isMouseDown) {
            var vx = (targetX - ball.x) * easing,
                vy = (targetY - ball.y) * easing;

            ball.x += vx;
            ball.y += vy;
        }
        ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

何时停止缓动

当计算一个单目标点的简单缓动时，物体最终会到达这个目标点，缓动也就完成了。但是，在前面的几个例子里，即使该物体看起来已经停住了，计算缓动的代码还是一直在运行，这样比较浪费系统资源。一旦物体到达了目标点，代码就应该不再执行。初看会觉得实现这个功能很简单，在动画循环里面判断一下物体是否到达目标点就行了，就像这样：

```
if (ball.x === targetX && ball.y === targetY) {
    //code to stop the easing
}
```

但是实际上这里还是有点棘手的。第 6 章提到过这个问题，现在会更详细地介绍一下。

我们所探讨的这种类型的缓动涉及了**芝诺悖论**（*Zeno's Paradox*）。埃利亚的芝诺（Zeno of Elea）是一个希腊哲学家，他设计了一系列问题来表明：与我们的感觉不同，万物都是不变的，运动只是一种幻觉。为了解释他的观点，他这样描述运动：为了把一个物体从 *A* 点移动到 *B* 点，就必须把它先移动到 *A* 和 *B* 的中间点 *C*，然后再移动到 *C* 和 *B* 的中间点，然后再折半，以此论推，每次移动物体到距离目标点的一半。这样就会无穷循环下去，物体永远也到不了目标点。

之所以称此为悖论，是因为它听起来还挺有道理的。但是经验告诉我们，确实可以把一个物体从 *A* 点移动到 *B* 点。让我们来看看在 JavaScript 代码中何时停止计算。在 *x* 轴上，有一个物体位于 0 的位置，要把它移动到 100。把 easing 变量设置为 0.5，因此它每次移动距离的一半。具体过程如下。

- 从原点开始，在第一帧后，它移动到 50。
- 在第二帧后，移动到 75。
- 现在距离终点的距离是 25，再次移动一半，现在的位置是 87.5。
- 按照这样循环下去，位置将变化到 93.75、96.875、98.437 5 等。经过 20 帧后，位置为 99.999 809 265。

正如你所看到的，它会离目标点越来越近，但是理论上永远不会到达目标点。但是当从代码的角度考虑这个问题的时候，情况又有一点不一样。在视觉上有这么一个问题：“还能把一个像素再做切分吗？”在 HTML5 Canvas 规范里并没有规定，不同浏览器可能有不同的实现。

浏览器开发商可以实现高分辨率的 canvas（精确到小数后多位小数），比如精度为 0.1，就是在一个像素的距离上播放 10 帧，但是一般人肉眼都分辨不出来。

举个例子，在 20 帧后，物体到了最接近 100 的位置：99.999 904 632 568 36。

```
var position = 0,
    target = 100;

for (var i = 0; i < 20; i++) {
  console.log(i + ": " + position);
  position += (target - position) * 0.5;
}
```

循环了 20 次，每次折半移动。这只是一段缓动的基础代码，并不会产生动画效果，我们只是用来输出位置的变化。但是你会发现，当循环了 11 次之后，位置就到了 99.9。在 canvas 上看的话，它已经到终点了。

即使在 canvas 上看来小球已经到目标点了，但从数学的角度来看，它永远也不会到达目标点。所以，如果你只是和前面提到的例子一样，在循环中简单地判断是否已到达目标点，缓动代码就永远都不会停止运行。所以，你要确定的问题是："多近才是足够近？"这需要判断物体到目标点的距离是否小于特定值。在本书的很多例子里，如果物体到终点的距离小于一个像素，我们就认为它到终点了，这时计算缓动的代码就可以停止了。

如果你在用二维坐标，就可以用第 3 章介绍的公式来计算距离：

```
var distance = Math.sqrt(dx * dx + dy * dy);
```

如果是在一维坐标上，因为距离算出来可能是负数，所以需要计算距离的绝对值。可以用 Math.abs 方法。

下面是一个停止缓动计算的例子（03-easing-off.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Easing Off</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <textarea id="log"></textarea>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        log = document.getElementById('log'),
        ball = new Ball(),
        easing = 0.05,
        targetX = canvas.width / 2,
        animRequest;

    ball.y = canvas.height / 2;
```

```
    (function drawFrame () {
      animRequest = window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var dx = targetX - ball.x;

      if (Math.abs(dx) < 1) {
          ball.x = targetX;
          window.cancelRequestAnimationFrame(animRequest);
          log.value = "Animation done!";
       } else {
          var vx = dx * easing;
          ball.x += vx;
       }
       ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

本例将缓动公式展开了，首先计算出了距离，因为后面要用它来判断是否停止代码的执行。也许现在你可以看出为什么需要计算 dx 的绝对值了，如果小球在目标点的右边，dx 就是一个负数，If (dx<1)这条语句就会一直是 true，动画也就不会继续了。通过使用 Math.abs，你就能确定真实的距离是否小于 1，从而控制是否停止缓动。

animRequest 这个变量用来存储每一帧的播放请求。当需要停止动画循环的时候，调用 window.cancelRequestAnimationFrame，参数就是 animRequest。如果浏览器没有原生支持这个函数，就可以把下面这段支持跨浏览器的代码加入 utils.js 文件中：

```
if (!window.cancelRequestAnimationFrame) {
    window.cancelRequestAnimationFrame = (window.cancelAnimationFrame ||
                                          window.webkitCancelRequestAnimationFrame ||
                                          window.mozCancelRequestAnimationFrame ||
                                          window.oCancelRequestAnimationFrame ||
                                          window.msCancelRequestAnimationFrame ||
                                          window.clearTimeout);
}
```

如果你想把拖放与缓动结合，别忘了在放开小球时重新开启动画代码。

移动的目标点

在前面的例子中，目标点都只有一个，并且是固定的，但是这并不是必要条件。我们在每一帧都会重新计算距离，然后根据距离计算速度。代码并不关心物体是否到达了目标点或者目标点是否在移动。它只须在播放每一帧的时候知道目标点的位置，然后计算距离和速度。

可以很容易地把鼠标位置作为目标点，只要把前面例子中的 targetX 和 targetY 替换为鼠标位置（mouse.x 和 mouse.y）就可以了。下面给出这个实现（04-ease-tomouse.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Ease to Mouse</title>
```

```
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        ball = new Ball(),
        easing = 0.05;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var vx = (mouse.x - ball.x) * easing,
          vy = (mouse.y - ball.y) * easing;

      ball.x += vx;
      ball.y += vy;
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

移动鼠标，观察小球如何跟随指针运动。鼠标离得越远，小球就运动得越快。

思考一下，还有没有其他东西可以作为移动的目标？或许一个物体可以把另一个物体作为目标点，这样你就可以用这种简单的技术做出很复杂的效果。

缓动不仅仅适用于运动

有一点很重要，在本书的例子中，我们都是在操作物体的一些属性值。大多数情况下是通过操作 x 和 y 属性来改变物体的位置。但是，还有很多其他属性。只要这个属性是可以用数字表示的，就可以操作它。当你阅读示例的时候，多尝试，别只是一看而过。可以尝试改变其他属性值，下面给你一些建议。

旋转

设置旋转角度的当前值和目标值，当然，需要一个可以旋转并且可以绘制在 canvas 上的对象，比如前几章提到的 Arrow 类。

```
var rotation = 90,
    targetRotation = 270;
```

然后使用缓动：

```
rotation += (targetRotation - rotation) * easing;
arrow.rotation = rotation * Math.PI / 180; //degrees to radians
```

颜色

尝试在24位颜色上使用缓动。要设置红、绿、蓝的初始值和目标值，用缓动改变每一种单独的颜色，然后再把它们合并为单个颜色值。例如，可以把红色缓动到蓝色。首先初始化变量：

```
var red = 255,
    green = 0,
    blue = 0,
    redTarget = 0,
    greenTarget = 0,
    blueTarget = 255;
```

然后在drawFrame函数里，使用缓动改变它们：

```
red += (redTarget - red) * easing;
green += (greenTarget - green) * easing;
blue += (blueTarget - blue) * easing;
```

最后把这三个单色值合并为一种颜色（参照第4章中的介绍），并应用：

```
var color = red << 16 | green << 8 | blue;
```

透明度

将缓动应用在alpha属性（CSS风格的颜色字符串）上，初始值是0，目标值是1（alpha的取值范围是0.0～1.0）：

```
var alpha = 0,
    targetAlpha = 1;
```

然后在drawFrame函数里，使用缓动实现淡入效果，然后把它拼接成一个RGBA字符串：

```
alpha += (targetAlpha - alpha) * easing;
ball.color = "rgba(255, 0, 0," + alpha + ")";
```

或者逆转0和1，就能实现淡出效果。

8.2.2 高级缓动

既然你已经知道简单缓动如何工作了，就可能还想试试更复杂的缓动公式来实现一些特效。例如，你想要一个物体先加速，接近目标点时又减速。也许你还希望在特定时间或者特定帧数内完成缓动。

罗伯特·皮诺（Robert Penner）收集了许多缓动公式并加以分类，同时在Flash中实现。可以在http://robertpenner.com中找到他的这些缓动公式。为了便于使用和查看，我已经把他的这些代码迁移到JavaScript。可以从http://github.com/lamberta/html5-animation下载这些代码以及本书中的其他示例。

8.3 弹动

弹动是动画编程中最重要和最强大的物理概念之一。你几乎可以用弹动来做任何事情，但

是当然，它只是一项技术。既然它这么有用，我们就来看看什么是弹动，以及如何编程。

正如本章一开始提到的，在弹动中，物体的加速度与它到目标点的距离成比例。想象一下现实中弹动的例子：在橡皮筋的一头系上一个小球，另一头固定起来。小球的目标点就是它静止悬空的那个点。将小球拉开一小段距离然后松开，刚松手那一瞬间，它的速度为零，但是橡皮筋给它施加了外力，把它拉向目标点。如果把小球尽可能地拉远，橡皮筋对它施加的外力就会变得很大。松手后，小球会急速飞过目标点，这时它的速度很高。但是，当它飞过目标点后，橡皮筋又把它向回拉，使其速度减小。它飞得越远，橡皮筋施加的力就越大。最终，它的速度降为零，又掉头往回飞。反复几次后，小球逐渐慢下来，停在目标点上。

接下来，我们用代码来实现这个过程。为了便于理解，我们先从一维坐标上的例子开始。

8.3.1 一维坐标上的弹动

我们继续使用红色小球举例。弹动的起始位置为 *x* 轴上的 0 点，目标点是 canvas 的中心点。与缓动类似，需要一个变量来存储弹性比例系数，可以把它看作距离的一部分，并会累加在速度上。较大的弹性系数会表现出较硬的弹簧效果。反之，如果弹性系数较小，动画就会看起来和橡皮筋一样松松垮垮。先把弹性系数设为 0.1，下面是初始化代码。

```
var spring = 0.1,
    targetX = canvas.width / 2,
    vx = 0;
```

再次重申，先别关心这些代码要写在什么地方，确保首先理解这些变量和表达式在干什么。

接下来是动画代码，计算小球到目标点的距离：

```
var dx = targetX - ball.x;
```

现在，计算加速度。加速度与距离是成比例的，也就是，加速度等于距离乘以 spring 的值：

```
var ax = dx * spring;
```

当你算出加速度后，接下来应该就很熟悉了。把加速度累加在速度上，然后把速度加在小球的位置上：

```
vx += ax;
ball.x += vx;
```

在开始写代码之前，先模拟一下整个过程。假设小球的 x 是 0，vx 也是 0，目标点的 x 是 100，spring 变量的值是 0.1。下面是执行过程。

① 用距离（100）乘以 spring，得到 10。将它加在 vx 上，vx 变为 10。把 vx 加在小球的位置上，小球的 x 为 10。

② 下一轮，距离（100–10）为 90。加速度为 90 乘以 0.1，等于 9，加在 vx 上，vx 就变为 19。小球的 x 变为 29。

③ 再下一轮，距离是 71，加速度是 7.1，vx 是 26.1，小球的 x 为 55.1。

④ 再下一轮，距离是 44.9，加速度是 4.49，vx 变成 30.59，小球的 x 变为 85.69。

随着小球一帧一帧地靠近目标，加速度变得越来越小，但是速度一直在增加。虽然增加的幅度在减小，但是速度越来越快。

几轮过后，小球越过了目标点，到达了 x 轴上的 117 点。与目标点的距离现在是 100–117 等于–17，加速度就是–1.7。这个加速度累加在速度上，使得小球减速。

既然你已经知道弹动的原理了，下面我们就来看一个真实的例子。和前面一样，确保正确引用 Ball 类，代码在 05-spring-1.html 里：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Spring 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        spring = 0.03,
        targetX = canvas.width / 2,
        vx = 0;

    ball.y = canvas.height / 2;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var dx = targetX - ball.x,
        ax = dx * spring;

      vx += ax;
      ball.x += vx;
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

在浏览器里试试看，你会看到类似弹簧的效果。但问题是它永远不会停下来。在前面讲述弹动的时候，我们说过，弹动会越来越慢，直到停下来。但是在本例中，由于小球的摆动幅度不变，因此它在某个点的速度不会减小，而会一直弹跳下去。需要使用摩擦力让它停下来。很简单，创建一个 friction 变量，初始值为 0.95。把这些变量的初始化代码写在脚本的顶部：

```
var friction = 0.95;
```

然后在 drawFrame 函数里，用 vx 乘以 friction。代码如 06-spring-2.html 所示：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
```

```
  context.clearRect(0, 0, canvas.width, canvas.height);

  var dx = targetX - ball.x,
      ax = dx * spring;

  vx += ax;
  vx *= friction;
  ball.x += vx;
  ball.draw(context);
}());
```

现在，实现了一个完整的弹动，虽然只是在一维坐标上。玩玩看，尝试改变 spring 和 friction 的值，观察效果的变化。再试试不同的起始点和目标点对系统的行为、小球的速度、它减慢的速率以及将要停止的速率有什么影响。

与缓动类似，虽然小球看起来已经停下了，但是代码还在一直运行。同样，为了节省系统资源，用以下代码来判断是否停止计算：

```
if (Math.abs(vx) > 0.001) {
  vx += ax;
  vx *= friction;
  ball.x += vx;
}
```

理解这个例子对接下来的学习非常有帮助，下面就要准备好开始学习二维坐标上的弹动了。

8.3.2 二维坐标上的弹动

前一个例子是让小球在 x 轴上运动，或者说从左到右。如果我们想让小球同时在 x 轴和 y 轴上运动——从左上角到右下角，就需要引入二维坐标上的弹动。很简单，只须要把目标点、速度和加速度扩展到二维坐标系上就行了。参见例子（07-spring-3.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Spring 3</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        spring = 0.03,
        friction = 0.95,
        targetX = canvas.width / 2,
        targetY = canvas.height / 2,
        vx = 0,
```

```
        vy = 0;

      (function drawFrame () {
        window.requestAnimationFrame(drawFrame, canvas);
        context.clearRect(0, 0, canvas.width, canvas.height);

        var dx = targetX - ball.x,
            dy = targetY - ball.y,
            ax = dx * spring,
            ay = dy * spring;

        vx += ax;
        vy += ay;
        vx *= friction;
        vy *= friction;
        ball.x += vx;
        ball.y += vy;
        ball.draw(context);
      }());
    };
    </script>
  </body>
</html>
```

和前一个例子唯一的不同是增加了一条 y 轴。现在的问题是它看起来仍然像是一维运动。虽然小球同时在 x 轴和 y 轴上运动，但它仍然是一条直线。原因是它的初始速度为 0，也仅受一个把它拉向目标点的外力，所以它沿直线向目标点运动。

为了让动画更有趣一点，把 **vx** 的初始值增大一点，设为 50。现在是不是看起来更加自然和流畅一些？这仅仅只是一个开始，接下来的内容更酷。

8.3.3 向移动的目标点弹动

你可能不会惊讶，弹动的目标点并不需要是固定点。在介绍缓动时，有个简单的例子，实现了小球跟随鼠标缓动。让小球跟随鼠标弹动同样很简单。只要把 targetX 和 targetY 替换为鼠标坐标即可。和缓动一样，在每一帧中重新计算物体到目标点的距离，再计算加速度，然后累加在速度上。

效果非常酷，而且代码基本没变。只需要在前面的例子里改动如下几行：

```
var dx = targetX - ball.x,
    dy = targetY - ball.y;
```

改为：

```
var dx = mouse.x - ball.x,
    dy = mouse.y - ball.y;
```

别忘了把下面一行代码加入脚本的开始。

```
var mouse = utils.captureMouse(canvas)
```

可以把声明 targetX 和 targetY 的代码删除了，因为已经不需要它们了。完整的示例在 08-spring-4.html 中。

花点时间玩玩这个例子，感受这些变量的作用，尝试做一些改动以加深理解。

8.3.4 弹簧在哪儿

在上面的例子中，小球看起来很像是拴在橡皮筋上，但是这个橡皮筋看不见。你需要用画图 API 画一条线来解决这个问题。

由于上面的例子比较简单，因此可以把画图语句直接放在 drawFrame 函数里。在一个更复杂的程序里，可能需要重新创建一个对象来封装这些画图语句，相当于新建一个图层。

在每一帧里，调整小球的位置之后，创建一个画图路径，然后在小球的位置和鼠标的位置之间画一条线，最后调用 stroke 函数来把线条显示在 canvas 上：

```
context.beginPath();
context.moveTo(ball.x, ball.y);
context.lineTo(mouse.x, mouse.y);
context.stroke();
```

现在引入重力怎么样？这样使得小球看起来像是真的挂在鼠标上一样。只需要加一个 gravity 变量，在每一帧中，把 gravity 加在 vy 上。下面给出代码（09-spring-5.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Spring 5</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        ball = new Ball(),
        spring = 0.03,
        friction = 0.9,
        gravity = 2,
        vx = 0,
        vy = 0;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var dx = mouse.x - ball.x,
          dy = mouse.y - ball.y,
          ax = dx * spring,
          ay = dy * spring;

     vx += ax;
     vy += ay;
```

```
        vy += gravity;
        vx *= friction;
        vy *= friction;
        ball.x += vx;
        ball.y += vy;

        context.beginPath();
        context.moveTo(ball.x, ball.y);
        context.lineTo(mouse.x, mouse.y);
        context.stroke();
        ball.draw(context);
      }());
    };
    </script>
  </body>
</html>
```

图 8-2 跟随鼠标的弹动，可见的弹簧

执行结果如图 8-2 所示。

注意，把 gravity 的值增大到了 2，这样做的目的是为了让小球逼真地下垂。如果 gravity 的值太小，弹力就会克服重力，你就看不出效果了。

这里，我们又偏离了现实世界中的物理学。当然，你不能对物体施加“递增重力”！重力是一个常数，只由你所在星球的质量来决定。我们只能增加物体的质量，这样重力的效果会更明显。理论上，应该保持 gravity 的值不变，比如 0.5，增加一个 mass 属性，值为 10。然后用 mass 乘以 gravity 得到 5。不过，还是把变量名叫做 gravity，但是你要明白我所说的重力与物体的质量相关。

再次试验一下这个例子。尝试减小 gravity 和 spring 的值，改变 friction 的值。你会看到几乎有无数种组合，可以用来建立各种各样的系统。

8.3.5 链式弹动

现在要把几个弹簧连起来。8.2 节简略地讨论了链式运动，一个物体向鼠标缓动，另一个向这个物体缓动，诸如此类。这里我们来看一些例子，展示如何在弹动中运用同样的概念。

计划是：创建三个小球，名字分别为 ball0、ball1 和 ball2。ball0 的行为与上例相同，然后 ball1 向 ball0 弹动，ball2 向 ball1 弹动。因为每个小球都受重力作用，所以它们是挂成一串的。所有的代码之前都见过，不过在这里更复杂一点。代码在 10-chain.html 中：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Chain</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
```

```
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        ball0 = new Ball(),
        ball1 = new Ball(),
        ball2 = new Ball(),
        spring = 0.03,
        friction = 0.9,
        gravity = 2;

    function move (ball, targetX, targetY) {
      ball.vx += (targetX - ball.x) * spring;
      ball.vy += (targetY - ball.y) * spring;
      ball.vy += gravity;
      ball.vx *= friction;
      ball.vy *= friction;
      ball.x += ball.vx;
      ball.y += ball.vy;
    }

    (function drawFrame () {
       window.requestAnimationFrame(drawFrame, canvas);
       context.clearRect(0, 0, canvas.width, canvas.height);

       move(ball0, mouse.x, mouse.y);
       move(ball1, ball0.x, ball0.y);
       move(ball2, ball1.x, ball1.y);

       //draw spring
       context.beginPath();
       context.moveTo(mouse.x, mouse.y);
       context.lineTo(ball0.x, ball0.y);
       context.lineTo(ball1.x, ball1.y);
       context.lineTo(ball2.x, ball2.y);
       context.stroke();
       //draw balls
       ball0.draw(context);
       ball1.draw(context);
       ball2.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

如果你再观察一下 Ball 类，就会发现每一个对象实例都有自己的 vx 和 vy 属性，并且初始化为 0。所以，在脚本的开始部分，只需要创建 3 个小球的实例，而不用重复初始化它们。

然后在 drawFrame 函数中，执行所有弹动和绘图语句。把重复代码提取到了 move 函数里，它处理所有动画代码，它接受一个小球的引用和目标点的坐标作为参数。只需要为每一个小球调用 move 函数。对于第一个小球，传入鼠标的 x、y 坐标作为目标点，然后把第一个小球的位置作为第二个小球的目标点，把第二个小球的位置作为第三个小球的目标点。

最后，当所有小球都移动到位后，用画图语句绘制橡皮筋，把几个小球连接起来。在这里

例子中，friction 调整到 0.9，使得动画更敏捷一点。

可以让这个例子更具灵活性。用一个数组来保存链上的每个小球，循环访问每一个小球，然后移动、连线。这个变化只需要改动很少代码就能实现。首先，创建几个与数组相关的变量，然后加入一些对象：

```
var balls = [],
    numBalls = 5;
```

在脚本顶部，用 while 循环来初始化每个小球并把它加入数组：

```
while (numBalls--) {
    balls.push(new Ball(20));
}
```

最后，在 drawFrame 函数中，把每一个小球传到一个新的 draw 函数里，这个函数有两个参数，一个是 Ball 类的实例，另一个是这个小球在数组里的索引位置（由 Array.forEach 提供）。遍历数组中所有的小球，把第一个小球移动到鼠标位置，其余的小球依次移动到前一个小球的位置，画线把它们连接起来。只须改变 numBalls 的值，就可以加入任意多的小球。

```
function draw (ballB, i) {
  //if first ball, move to mouse
  if (i === 0) {
    move(ballB, mouse.x, mouse.y);
    context.moveTo(mouse.x, mouse.y);
  } else {
    var ballA = balls[i-1];
    move(ballB, ballA.x, ballA.y);
    context.moveTo(ballA.x, ballA.y);
  }
  context.lineTo(ballB.x, ballB.y);
  context.stroke();
  ballB.draw(context);
}

(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  context.beginPath();
  balls.forEach(draw);
}());
```

运行结果如图 8-3 所示，可以在 11-chain-array.html 中找到完整的代码。

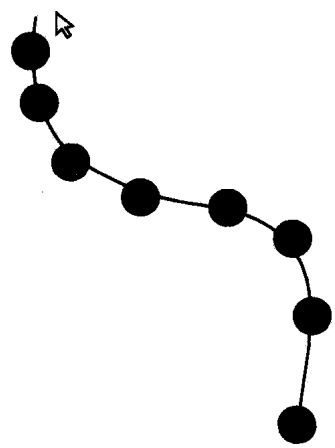


图 8-3 链式弹动

8.3.6 多个目标点的弹动

在第 5 章中介绍速度和加速度时，我们学习了如何把多个外力施加在物体上。每个外力都用一个值来表示，累加到速度上，最后运用到物体的运动上。因为弹力不过就是在一个物体上施加一个加速度，所以在一个物体上施加多个弹力也很简单。

为了演示多目标点的弹动，创建 3 个“控制点”——其实就是 Ball 类的 3 个实例，并让它们具

有拖放功能。它们就是小球弹动的目标点，小球同时向 3 个点运动，最终停在一个平衡点上。或者换句话说，每个目标点都会在小球上施加一定的加速度。小球的运动就是这些加速度叠加的结果。

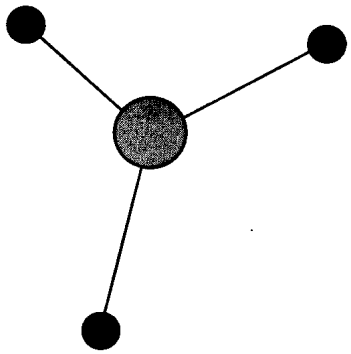


图 8-4 多目标点的弹动

这个例子变得相当复杂，有很多函数来处理不同的行为。首先来看看这个完整的例子（12-multi-spring.html），然后逐步分析它。图 8-4 是运行结果。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Multi Spring</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        ball = new Ball(20),
        handles = [],
        numHandles = 3,
        spring = 0.03,
        friction = 0.9,
        movingHandle = null;

    for (var handle, i = 0; i < numHandles; i++) {
       handle = new Ball(10, "#0000ff");
       handle.x = Math.random() * canvas.width;
       handle.y = Math.random() * canvas.height;
       handles.push(handle);
    }

    canvas.addEventListener('mousedown', function () {
      handles.forEach(function (handle) {
        if (utils.containsPoint(handle.getBounds(), mouse.x, mouse.y)) {
          movingHandle = handle;
        }
      });
    }, false);

    canvas.addEventListener('mouseup', function () {
      if (movingHandle) {
        movingHandle = null;
      }
    }, false);

    canvas.addEventListener('mousemove', function () {
```

```
    if (movingHandle) {
      movingHandle.x = mouse.x;
      movingHandle.y = mouse.y;
    }
  }, false);

  function applyHandle (handle) {
    var dx = handle.x - ball.x,
        dy = handle.y - ball.y;

    ball.vx += dx * spring;
    ball.vy += dy * spring;
  }

  function drawHandle (handle) {
    context.moveTo(ball.x, ball.y);
    context.lineTo(handle.x, handle.y);
    context.stroke();
    handle.draw(context);
  }

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    handles.forEach(applyHandle);
    ball.vx *= friction;
    ball.vy *= friction;
    ball.x += ball.vx;
    ball.y += ball.vy;

    context.beginPath();
    handles.forEach(drawHandle);
    ball.draw(context);
  }());
};
</script>
</body>
</html>
```

在脚本刚开始的地方，创建了一个小球和三个控制点，控制点具有拖放功能，位置随机。

在 drwaFrame 函数里，循环遍历所有的控制点，调用 applyHandle 函数使小球向控制点运动，改变小球的速度和位置，然后画线连接小球和每个控制手柄。最后，绘制小球。

在 mousedown 事件监听器中，遍历所有的控制点，如果发现鼠标光标在当前控制点内，则把这个控制点设置为 movingHandle。在 mouseup 事件处理程序中，检查如果有控制手柄正在被拖动（movingHandle！=null），则停止拖动。

改变 numHandles 变量的值，就可以很简单地设置控制程序的数量。

8.3.7 目标偏移量

如果使用一个用弹性金属材料制作的、真正的弹簧。把一头固定起来，另一头系上一个小球

或其他东西，小球弹动的目标点是什么？是固定弹簧的那个地方吗？不是，小球永远都到不了那个点，因为真正的弹簧也是有体积的，会把它挡住。此外，一旦弹簧收缩到它正常的长度，它就不会对小球施加拉力。所以，真正的目标点就是弹簧处于松弛状态时，系着小球一端的那个点。不过随着弹簧绕着它的轴心点（固定点）旋转，目标点也会发生变化。

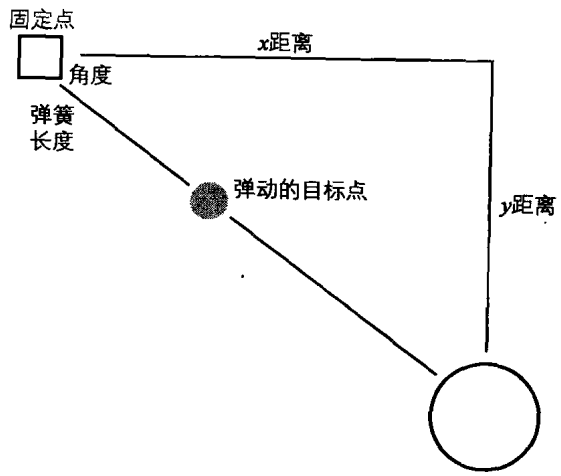


图 8-5 弹簧偏移量

要确定物体真正的目标点，首先要确定物体相对固定点的角度，然后沿着这个角度向外延伸弹簧的长度那么多。换句话说，如果弹簧的长度是 50，小球和固定点的角度是 45°，沿着 45° 向外延伸 50，这个点就是小球的目标点，如图 8-5 所示。

计算目标点的代码如下：

```
var dx = ball.x - fixedX,
    dy = ball.y - fixedY,
    angle = Math.atan2(dy, dx),
    targetX = fixedX + Math.cos(angle) * springLength,
    targetY = fixedY + Math.sin(angle) * springLength;
```

运行结果是，小球向固定点弹动，但是会停在固定点前面一段距离。我们叫它“固定点”，只是说它是固定弹簧的点，并不是说这个点不能移动。可能直接看代码会更容易理解一些。我们继续用鼠标位置，不过这次它作为弹簧的固定点。弹簧长度为 100 个像素。下面是例子（13-offset-spring.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Offset Spring</title>
  <link rel="stylesheet" href="style.css">
 </head>
<body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        ball = new Ball(),
        spring = 0.03,
        friction = 0.9,
        springLength = 100,
        vx = 0,
```

```
        vy = 0;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var dx = ball.x - mouse.x,
          dy = ball.y - mouse.y,
          angle = Math.atan2(dy, dx),
          targetX = mouse.x + Math.cos(angle) * springLength,
          targetY = mouse.y + Math.sin(angle) * springLength;

      vx += (targetX - ball.x) * spring;
      vy += (targetY - ball.y) * spring;
      vx *= friction;
      vy *= friction;
      ball.x += vx;
      ball.y += vy;

      context.beginPath();
      context.moveTo(ball.x, ball.y);
      context.lineTo(mouse.x, mouse.y);
      context.stroke();
      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

虽然我们看到了本例的效果，但是这并没有很明显地体现出这个技术的用处。在下一节你将会看到一个更具体的例子。

8.3.8 用弹簧连接多个物体

我们已经知道怎么把一个物体用弹簧连在某个点上，并且这个点可以移动。接下来，可以把两个物体用一根弹簧连接起来，让它们互相向对方弹动。移动其中一个，另一个就要跟随弹动过去。

这是个一般策略，但是用本章的术语来解释一下：物体 *A* 的目标点是物体 *B*，并向 *B* 弹动，物体 *B* 反过来以 *A* 作为目标点弹动。这就是引入偏移量的一个重要原因，如果一个物体以另一个物体作为直接目标点，它们就会因为冲向同一目标点而撞毁（重叠）。引入偏移量的概念，就可以让它们保持一点距离，如图 8-6 所示。

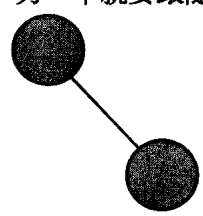


图 8-6 两个物体相连

为此举一个例子，创建 Ball 的两个实例，用弹簧相连。变量名叫 ball0 和 ball1。ball0 向 ball1 弹动，ball1 向 ball0 弹动，它们之间有一定的偏移量。为了消除重复代码，写一个叫 springTo 的函数。调用 springTo(ball0, ball1)就是让 ball0 弹动到 ball1，反之调用 springTo(ball1, ball0)。还要引入两个变量：ball0_dragging 和 ball1_dragging，作为是否拖动小球的标志，下面是代码

(14-double-spring.html):

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Double Spring</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        ball0 = new Ball(20),
        ball1 = new Ball(20),
        ball0_dragging = false,
        ball1_dragging = false,
        spring = 0.03,
        friction = 0.9,
        springLength = 100,
        vx = 0,
        vy = 0;

    ball0.x = Math.random() * canvas.width;
    ball0.y = Math.random() * canvas.height;
    ball1.x = Math.random() * canvas.width;
    ball1.y = Math.random() * canvas.height;
    canvas.addEventListener('mousedown', function () {
      if (utils.containsPoint(ball0.getBounds(), mouse.x, mouse.y)) {
        ball0_dragging = true;
      }
      if (utils.containsPoint(ball1.getBounds(), mouse.x, mouse.y)) {
        ball1_dragging = true;
      }
    }, false);

    canvas.addEventListener('mouseup', function () {
      if (ball0_dragging || ball1_dragging) {
        ball0_dragging = false;
        ball1_dragging = false;
      }
    }, false);

    canvas.addEventListener('mousemove', function () {
      if (ball0_dragging) {
        ball0.x = mouse.x;
        ball0.y = mouse.y;
      }
      if (ball1_dragging) {
```

```
        ball1.x = mouse.x;
        ball1.y = mouse.y;
      }
    }, false);

    function springTo (ballA, ballB) {
      var dx = ballB.x - ballA.x,
          dy = ballB.y - ballA.y,
          angle = Math.atan2(dy, dx),
          targetX = ballB.x - Math.cos(angle) * springLength,
          targetY = ballB.y - Math.sin(angle) * springLength;

      ballA.vx += (targetX - ballA.x) * spring;
      ballA.vy += (targetY - ballA.y) * spring;
      ballA.vx *= friction;
      ballA.vy *= friction;
      ballA.x += ballA.vx;
      ballA.y += ballA.vy;
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      if (!ball0_dragging) {
        springTo(ball0, ball1);
      }
      if (!ball1_dragging) {
        springTo(ball1, ball0);
      }
      context.beginPath();
      context.moveTo(ball0.x, ball0.y);
      context.lineTo(ball1.x, ball1.y);
      context.stroke();
      ball0.draw(context);
      ball1.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

本例中的小球在 canvas 元素上都是可以拖放的。在 drawFrame 中判断如果小球没有正在拖动，就调用 springTo 函数。

```
if (!ball0_dragging) {
   springTo(ball0, ball1);
}
if (!ball1_dragging) {
   springTo(ball1, ball0);
}
```

弹动的所有代码都写在 springTo 函数中，这里的代码你应该已经比较熟悉了。首先，它计算出两球之间的距离和角度，然后根据距离计算出目标点。然后向目标点弹动。当第二次调用 springTo 时，参数对调，两个球的角色互换。原来那个作为目标点的小球向另一个小球弹动。

你会看到，没有一个小球悬挂在固定点或鼠标位置上。它们都可以自由活动。唯一的约束是彼此之间保持一段距离。这种写法最棒的地方是可以很容易地加入更多的小球。例如，让我们来加入第三个小球（在示例 15-triple-spring.html 中）。添加两个新变量：ball2 和 ball2_dragging。再加入一些鼠标事件处理程序以及动画循环（参照下面的代码）。依据这个模式，可以加入任意多个小球。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Triple Spring</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        ball0 = new Ball(20),
        ball1 = new Ball(20),
        ball2 = new Ball(20),
        ball0_dragging = false,
        ball1_dragging = false,
        ball2_dragging = false,
        spring = 0.03,
        friction = 0.9,
        springLength = 100,
        vx = 0,
        vy = 0;

    ball0.x = Math.random() * canvas.width;
    ball0.y = Math.random() * canvas.height;
    ball1.x = Math.random() * canvas.width;
    ball1.y = Math.random() * canvas.height;
    ball2.x = Math.random() * canvas.width;
    ball2.y = Math.random() * canvas.height;

    canvas.addEventListener('mousedown', function () {
      if (utils.containsPoint(ball0.getBounds(), mouse.x, mouse.y)) {
          ball0_dragging = true;
      }
      if (utils.containsPoint(ball1.getBounds(), mouse.x, mouse.y)) {
          ball1_dragging = true;
      }
      if (utils.containsPoint(ball2.getBounds(), mouse.x, mouse.y)) {
          ball2_dragging = true;
      }
    }, false);

    canvas.addEventListener('mouseup', function () {
```

```
  if (ball0_dragging || ball1_dragging || ball2_dragging) {
      ball0_dragging = false;
      ball1_dragging = false;
      ball2_dragging = false;
  }
}, false);

canvas.addEventListener('mousemove', function () {
  if (ball0_dragging) {
      ball0.x = mouse.x;
      ball0.y = mouse.y;
  }
  if (ball1_dragging) {
      ball1.x = mouse.x;
      ball1.y = mouse.y;
  }
  if (ball2_dragging) {
      ball2.x = mouse.x;
      ball2.y = mouse.y;
  }
}, false);

function springTo (ballA, ballB) {
  var dx = ballB.x - ballA.x,
      dy = ballB.y - ballA.y,
      angle = Math.atan2(dy, dx),
      targetX = ballB.x - Math.cos(angle) * springLength,
      targetY = ballB.y - Math.sin(angle) * springLength;

  ballA.vx += (targetX - ballA.x) * spring;
  ballA.vy += (targetY - ballA.y) * spring;
  ballA.vx *= friction;
  ballA.vy *= friction;
  ballA.x += ballA.vx;
  ballA.y += ballA.vy;
}

(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  if (!ball0_dragging) {
      springTo(ball0, ball1);
      springTo(ball0, ball2);
  }
  if (!ball1_dragging) {
      springTo(ball1, ball0);
      springTo(ball1, ball2);
  }
  if (!ball2_dragging) {
      springTo(ball2, ball0);
      springTo(ball2, ball1);
  }

  context.beginPath();
```

```
        context.moveTo(ball0.x, ball0.y);
        context.lineTo(ball1.x, ball1.y);
        context.lineTo(ball2.x, ball2.y);
        context.lineTo(ball0.x, ball0.y);
        context.stroke();
        ball0.draw(context);
        ball1.draw(context);
        ball2.draw(context);
      }());
    };
    </script>
  </body>
</html>
```

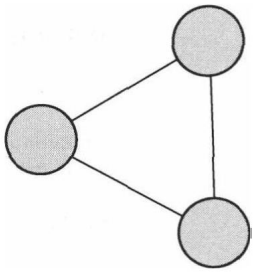


图 8-7 弹簧连接的三个物体

这样就创建了一个三角形结构，如图 8-7 所示。当你掌握窍门后，你也可以创建正方形结构，进而还创建更复杂的弹簧连接结构。

8.4 本章中的重要公式

又到了回顾本章重要公式的时候了。

8.4.1 简单缓动，详细版

```
var dx = targetX - object.x,
    dy = targetY - object.y;
vx = dx * easing;
vy = dy * easing;
object.x += vx;
object.y += vy;
```

8.4.2 简单缓动，缩略版

```
vx = (targetX - object.x) * easing;
vy = (targetY - object.y) * easing;
object.x += vx;
object.y += vy;
```

8.4.3 简单缓动，简易版

```
object.x += (targetX - object.x) * easing;
object.y += (targetY - object.y) * easing;
```

8.4.4 简单弹动，详细版

```
var ax = (targetX - object.x) * spring,
    ay = (targetY - object.y) * spring;
vx += ax;
vy += ay;
```

```
vx *= friction;
vy *= friction;
object.x += vx;
object.y += vy;
```

8.4.5 简单弹动，缩略版

```
vx += (targetX - object.x) * spring;
vy += (targetY - object.y) * spring;
vx *= friction;
vy *= friction;
object.x += vx;
object.y += vy;
```

8.4.6 简单弹动，简易版

```
vx += (targetX - object.x) * spring;
vy += (targetY - object.y) * spring;
object.x += (vx *= friction);
object.y += (vy *= friction);
```

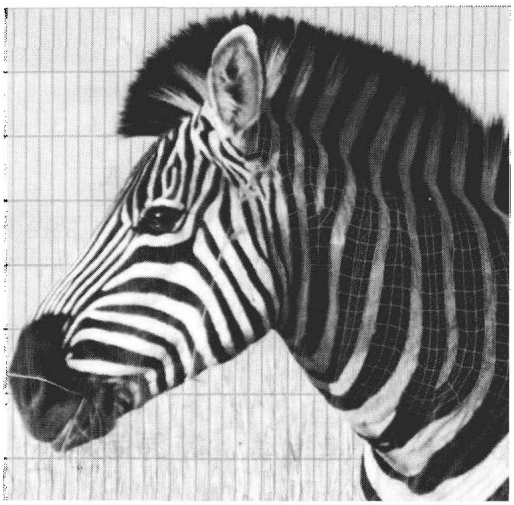
8.4.7 有偏移量的弹动

```
var dx = object.x - fixedX,
    dy = object.y - fixedY,
    angle = Math.atan2(dy, dx),
    targetX = fixedX + Math.cos(angle) * springLength,
    targetY = fixedX + Math.sin(angle) * springLength;
//spring to targetX, targetY as above
```

8.5 小结

本章介绍了两个比例运动的基础技术：缓动和弹动，缓动是比例速度，弹动是比例加速度。你已经很好地理解了如何运用这两个技术，可以开始用它们创建一些很有趣的效果了。

现在，你已经学习了各种各样移动物体的方法。让我们进入第 9 章，你将会学到如何处理物体间的碰撞。



第 9 章　碰撞检测

本章涵盖以下内容：

- 碰撞检测的方法
- 基于几何图形的碰撞检测
- 基于距离的碰撞检测
- 多物体碰撞的检测策略

读到这里，你已经学会了如何让物体在空间内运动和交互。现在，你将要学习如何让物体和物体交互。大多数情况是检测两个物体是否接触。这个话题叫做**碰撞检测**（collision detection 或 hit testing）。

本章将尽量覆盖所有碰撞检测的知识。这包括：两个物体间的碰撞检测，一个物体和一个点的碰撞检测，基于距离的碰撞检测，以及多物体的碰撞检测策略。首先，我们来看看都有哪些碰撞检测方法。

9.1　碰撞检测的方法

碰撞检测的概念很简单：就是判定两个物体是否在同一时间内占用同一块空间。当然，你也可能想要判断多个物体之间是否碰撞，但是分解开来看，需要每次选取一个物体，依次检测它有没有和其他物体碰撞。

有几种检测碰撞的方法。

- 可以从几何图形的角度来检测，就是判断一个物体是否和另一个物体有重叠。我们会

用物体的矩形边界来判断。

- 也可以检测距离，就是判断两个物体是否足够近到发生碰撞。你需要计算距离和判断两个物体是否足够近。

每种方法都有自己的应用场景，我们将在本章中详细地介绍。

但是，这里不会详细讲解碰撞后的行为，或者两个物体碰撞后会如何反应，这些内容将在第 11 章中讨论动量守恒时覆盖到。

9.2 基于几何图形的碰撞检测

在第 7 章中，在 Ball 类里新添加了一个 getBounds 方法，这个方法将在本章中多次用到。回忆一下，这个方法的返回值是一个矩形对象，它包含有 x、y、width 和 height 几个属性。这个矩形就是小球相对于 canvas 元素的边界框。

有了这个矩形，我们就能检查它是否和其他矩形相交，或者某一个坐标点是否落在矩形内。我们用这些方法来检测两个物体是否碰撞，或者物体是否和一个特定的点发生碰撞。

9.2.1 两个物体间的碰撞检测

现在有两个定义了 getBounds 方法的对象，可以通过检测两个对象的边界框是否相交，来判断两个物体是否碰撞。首先，在 utils.js 文件里添加一个工具函数 utils.intersects。

```
utils.intersects = function (rectA, rectB) {
  return !(rectA.x + rectA.width < rectB.x ||
             rectB.x + rectB.width < rectA.x ||
             rectA.y + rectA.height < rectB.y ||
             rectB.y + rectB.height < rectA.y);
};
```

这里没有什么特别炫的东西，只是些简单的几何运算。这个函数的参数是两个矩形对象。如果两个矩形相交，返回 true；否则，返回 false。调用方法如下：

```
utils.intersects(rectA, rectB)
```

当检测两个边界框是否相交时，通常把这个函数放在 if 语句里调用，像下面这样：

```
if (utils.intersects(objectA.getBounds(), objectB.getBounds()) {
    //react to collision
}
```

上面的代码段中，如果 objectA 和 objectB 发生碰撞，则 utils.intersects 返回 true，if 块中的语句就会执行。这大概是最简单也最易于编程的碰撞检测方法了。

但是与所有事情一样，这里需要一个权衡。碰撞检测算法越简单，精度就越低。如果精度要求越高，算法就越复杂而且需要大量计算。因为这个方法最简单，所以精度也是最低的。

我们怎么定义检测精度呢——不就是要么碰撞，要么没有碰撞吗？又回到了这个问题：基于两个物体的位置，如何确定它们是否碰撞？

在上面讨论的边界框算法里，先选择一个物体，在它周围画一个矩形。矩形的上边穿过物体顶

端的那个像素，下边穿过物体底端的那个像素，左边和右边也一样。用同样的方法在另一个用于检测的物体周边画一个矩形。最后，检测这两个矩形是否相交，如果相交，就认为两个物体碰撞了。

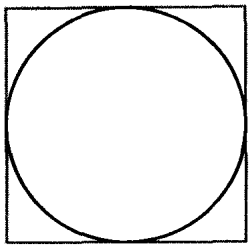


图 9-1 边界框

这个物体周围的矩形叫做边界框，它由小球的位置和大小计算得来。它通常是不可见的，但在图 9-1 中它画出来了，是为了让你能看到。

为什么这种方法不精确呢？你可能认为，如果两个边界框相交，那么两个物体肯定碰撞了。看看图 9-2，哪一对碰撞了呢？

很明显，只有那一对正方形真正地碰撞了，对吗？让我们为每一对图形都画上边界框，然后从计算的角度来看看，如图 9-3 所示。

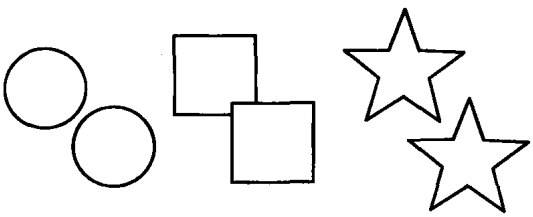


图 9-2 哪一对碰撞了

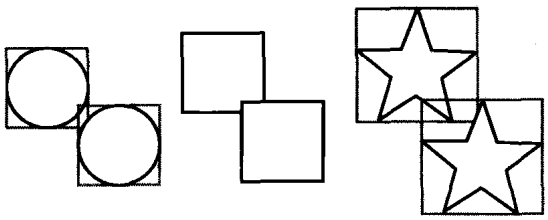


图 9-3 不是你期望的结果吗

站在代码的角度而言，每一对图形都相撞了。如果你不信，测试这个例子（01-object-hit-test.html）。因为它用到了前面创建的 Ball 类，所以确保导入 ball.js 和最新的 utils.js 文件。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Object Hit Test</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <textarea id="log"></textarea>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        log = document.getElementById('log'),
        mouse = utils.captureMouse(canvas),
        ballA = new Ball(),
        ballB = new Ball();

    ballA.x = canvas.width / 2;
    ballA.y = canvas.height / 2;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
```

```
        context.clearRect(0, 0, canvas.width, canvas.height);

        ballB.x = mouse.x;
        ballB.y = mouse.y;

        if (utils.intersects(ballA.getBounds(), ballB.getBounds())) {
          log.value = "Hit!";
        } else {
         log.value = '';
        }

        ballA.draw(context);
        ballB.draw(context);
      }());
    };
    </script>
  </body>
</html>
```

在本例中，创建了 Ball 类的两个实例，其中一个可以用鼠标拖放。在每一帧中都调用 utils.intersects 函数来检测两球是否相撞。当拖动小球从上、下、左、右方向靠近目标时，结果都很精确，但是当斜着靠近目标时，结果总是错的。如果用其他图形来代替小球，你会发现：矩形总是能得到很精确的结果，但是图形越不规则，结果就越不精确。所以，如果检测对象不是矩形，就要谨慎地使用这种方法了。

为了让小球的边界框可见，只需要把这些矩形绘制在 canvas 上。在 drawFrame 函数里加入以下代码：

```
var boundsA = ballA.getBounds(),
    boundsB = ballB.getBounds();

context.strokeRect(boundsA.x, boundsA.y, boundsA.width, boundsA.height);
context.strokeRect(boundsB.x, boundsB.y, boundsB.width, boundsB.height);
```

现在看一个使用 utils.intersects 来检测矩形的例子。这个例子用了一个新的 Box 类，与 Ball 很相似，我相信你肯定能理解，代码如下：

```
function Box (width, height, color) {
  if (width === undefined) { width = 50; }
  if (height === undefined) { height = 50; }
  if (color === undefined) { color = "#ff0000"; }
  this.x = 0;
  this.y = 0;
  this.width = width;
  this.height = height;
  this.vx = 0;
  this.vy = 0;
  this.rotation = 0;
  this.scaleX = 1;
  this.scaleY = 1;
  this.color = utils.parseColor(color);
  this.lineWidth = 1;
}
```

```
Box.prototype.draw = function (context) {
  context.save();
  context.translate(this.x, this.y);
  context.rotate(this.rotation);
  context.scale(this.scaleX, this.scaleY);
  context.lineWidth = this.lineWidth;
  context.fillStyle = this.color;
  context.beginPath();
  context.rect(0, 0, this.width, this.height);
  context.closePath();
  context.fill();
  if (this.lineWidth > 0) {
    context.stroke();
  }
  context.restore();
};
```

注意，把 getBounds 方法从 Box 类中移除了，因为矩形对象已经包含了所需的属性，它们可以直接传递给 utils.intersects 函数来检测碰撞。

在下面的例子中，方块从 canvas 的顶端落下。如果它到落在了 canvas 底部或者与其他方块发生碰撞，它就停止下落。如果在下落的过程中它碰到另一个方块，它就停在这个方块上面。下面是代码（02-boxes.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Boxes</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="box.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        boxes = [],
        activeBox = createBox(),
        gravity = 0.2;

    function createBox () {
      var box = new Box(Math.random() * 40 + 10, Math.random() * 40 + 10);
      box.x = Math.random() * canvas.width;
      boxes.push(box);
      return box;
    }

    function drawBox (box) {
      if (activeBox !== box && utils.intersects(activeBox, box)) {
        activeBox.y = box.y - activeBox.height;
        activeBox = createBox();
      }
      box.draw(context);
    }
```

```
    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      activeBox.vy += gravity;
      activeBox.y += activeBox.vy;

      if (activeBox.y + activeBox.height > canvas.height) {
          activeBox.y = canvas.height - activeBox.height;
          activeBox = createBox();
      }
      boxes.forEach(drawBox);
    }());
  };
  </script>
 </body>
</html>
```

在 drawFrame 函数中，检测方块是否落到了 canvas 底部。如果落到了，则它停止下落，并创建一个新的方块。然后它遍历 boxes 数组，把每个方块传入 drawBox 函数，它会检测方块是否和当前活动方块碰撞。首先，它要确保检测的对象不是自己，然后用 utils.intersects 检测两个方块是否相撞，这里是程序的核心。如果相撞，则把活动方块放在与它碰撞方块的上面，然后创建一个新的方块。让这个程序运行一会儿，你就能看到类似于图 9-4 的效果。

如果你想要看看这种方法不精确的例子，用圆形或其他不规则图形来替换矩形。当一些对象发生“碰撞”时，它们就会漂浮在空中，如图 9-5 所示。

图 9-4 矩形边界框的碰撞检测

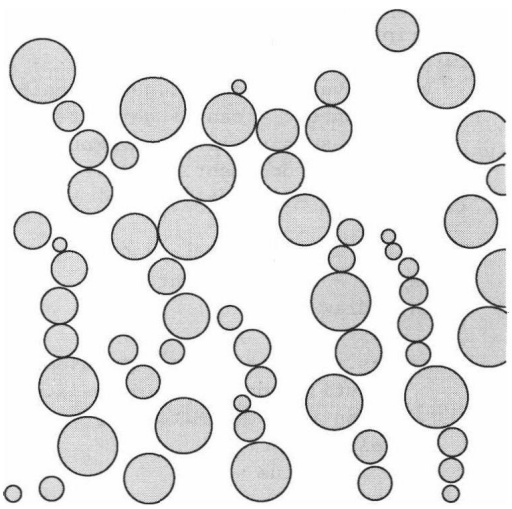


图 9-5 用圆形的边界框做碰撞检测

9.2.2 物体和点的碰撞检测

第 7 章介绍了一个工具函数 utils.containsPoint，这里我们将要看到它在碰撞检测中的作用，尽管它在两个物体间的碰撞检测中没什么用。前面介绍过，这个函数有 3 个参数：第一个是包含

了 x、y、width 和 height 属性的矩形对象，其余两个是一个点的 x、y 坐标。它的返回值是 true 或 false，这取决于参数中的点与矩形是否相撞。确保这个函数在 utils.js 文件中：

```
utils.containsPoint = function (rect, x, y) {
  return !(x < rect.x || x > rect.x + rect.width ||
           y < rect.y || y > rect.y + rect.height);
};
```

比如，要检测点(100，100)是否在一个矩形中，代码写法如下：

```
if (utils.containsPoint(rect, 100, 100)) {
    //react to collision
}
```

但是再次回到我们的问题：什么才算是碰撞？这里我们又看到边界框了——代码只是检查这个点是不是在物体的边界框内部。

让我们来快速地看一下例子 03-point-hit-test.html:

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Point Hit Test</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <textarea id="log"></textarea>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        log = document.getElementById('log'),
        ball = new Ball();

    ball.x = canvas.width / 2;
    ball.y = canvas.height / 2;
    ball.draw(context);

    canvas.addEventListener('mousemove', function () {
      if (utils.containsPoint(ball.getBounds(), mouse.x, mouse.y)) {
        log.value = "Hit!";
      } else {
        log.value = '';
      }
    }, false);
  };
  </script>
 </body>
</html>
```

这个例子检测鼠标光标是否与小球碰撞。当你将鼠标光标靠近小球时，你就会发现在鼠标光标真正地碰上小球之前就提示碰撞了，特别是从 45° 角靠近时更加明显，如图 9-6 所示。把

小球换成正方形试试，结果就会非常精确。所以，再一次证明了这种方法只是对矩形有用。

为了使用这种方法得到更精确的检测结果，可以在图形的轮廓上检测多个点。这样，你就要更精确地定义物体的边界，但是这也意味着，程序需要运行更多测试。如果图形很复杂，大量的测试会拖慢Web浏览器的速度。例如，如果有一个五角星的图案，就要分别对五个点做检测。如果有两个五角星，你就要做5×5次检测。对于五角星来说，这还算好，但如果有更复杂的图形，就需要检测更多的点。你就会发现复杂度急剧上升，只是两个五角星就带来了数十倍的测试量，追求精度是有代价的。

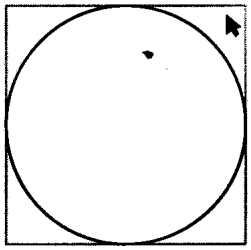


图9-6 用圆形的边界框做碰撞检测导致错误

如果必须对不规则的图形做碰撞检测，一个较好的办法是把复杂图形分解为小的矩形，然后检测这些矩形。这样提高了精度，但是也增加了计算量，所以需要在两者之间找一个平衡点。更高效一些的办法是，首先只检测图形的边界框，仅在检测到碰撞后，再把图形分解成小的矩形，进行更高精度的检测。

9.2.3 几何图形碰撞检测法的总结

对于两个不规则图形间的碰撞检测，这种基于几何图形的简单碰撞检测并不能很容易地支持。但我们还是有一些办法来应对。

- 对于近似于矩形对象，用utils.intersects(rectA, rectB)。
- 对于不规则的图形，如果要求高精度，可以基于utils.containsPoint(rect，x，y)来定制一个方案。

当然，本章还远没有结束，下面还会提出其他方案。如果检测的对象是圆形或者近似圆形，使用基于距离的碰撞检测法可能就是最合适的选择。你会惊奇地发现，非常多的图形都可以归类为“近似圆形”或“近似矩形”。

9.3 基于距离的碰撞检测

在本节中，你将要抛弃前面的几何图形检测函数，自己来处理碰撞检测。这包括使用两个物体间的距离来判断它们是否相撞。

举现实中存在的一个例子，如果你的车的中心点距离我的车的中心点100英尺，通过这个距离，你就知道我们两个人的车离得足够远，不可能撞上。但是，如果我们的车有6英尺宽，12英尺长，我车的中心点和你车的中心点相距5英尺，那么就可以肯定我们有麻烦了。换句话说，如果车没有撞得变形的话，两辆车不可能相距这么近。这就是基于距离的碰撞检测的全部概念。先确定两个物体的最小距离，再计算当前距离，然后进行比较，如果当前距离比最小距离小，那么肯定发生了碰撞。

当然，事情总是需要权衡的，正如边界框检测法很适合于矩形，但并不适合于其他图形一样。这种方法对于圆形来说很完美，但是对于除了圆形以外的其他图形，这种方法也不精确。

9.3.1 基于距离的简单碰撞检测

让我们从一个理想情况开始：有两个正圆形要进行碰撞检测，从圆的中心点开始计算。我们可以继续使用 Ball 类，开始创建本章的第一个例子。创建两个小球，把其中一个设置为可拖曳的。碰撞检测的代码仍旧写在 drawFrame 函数中，但是我们不再使用 utils.intersects 来检测碰撞，而是使用两球间的距离。你应该已经知道如何使用第 5 章中学到的勾股定理来计算两球之间的距离。程序是这样的：

```
var dx = ballB.x - ballA.x,
    dy = ballB.y - ballA.y,
    dist = Math.sqrt(dx * dx + dy * dy);
```

现在，距离已经有了，如何判断这个距离是否足够近到发生了碰撞呢？让我们来看图 9-7。

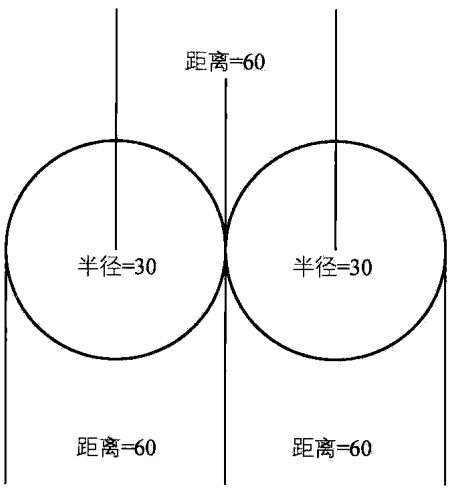


图 9-7 碰撞的距离

图 9-7 中的两个球正好处于一个刚刚接触的位置。考虑到每个球的直径有 60 个像素，半径就是 30 个像素。所以，在它们碰上的那一瞬间，两球中心点的距离就是 60 个像素。对于两个大小相同的球，如果距离小于球的直径，它们就发生了碰撞。代码如下（04-distance-1.html）所示。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Distance 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <textarea id="log"></textarea>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        log = document.getElementById('log'),
        ballA = new Ball(30),
        ballB = new Ball(30);

    ballA.x = canvas.width / 2;
```

```
    ballA.y = canvas.height / 2;
    canvas.addEventListener('mousemove', drawFrame, false);
    drawFrame();

    function drawFrame () {
      context.clearRect(0, 0, canvas.width, canvas.height);

      ballB.x = mouse.x;
      ballB.y = mouse.y;

      var dx = ballB.x - ballA.x,
          dy = ballB.y - ballA.y,
          dist = Math.sqrt(dx * dx + dy * dy);

      if (dist < 60) {
          log.value = "Hit!";
      } else {
          log.value = '';
      }
      ballA.draw(context);
      ballB.draw(context);
    }
  };
  </script>
 </body>
</html>
```

运行后，你就会发现，无论以任何角度接近目标球，程序都能精确地判断。

我在前面已经提到过硬编码数字通常是不好的风格，因为每次需要创建不同大小的图形时都要更改代码。此外，如果两个对象的大小不一样呢？所以，我们需要把这个概念抽象成一个可以适应任何情况的公式。

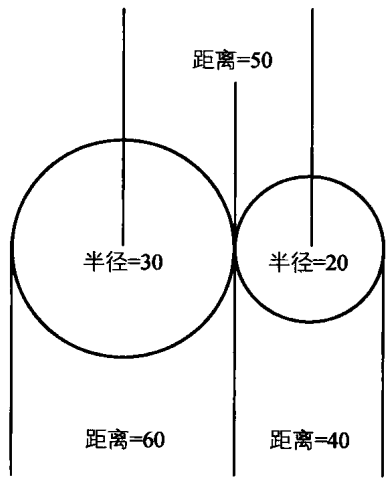


图 9-8 两个不同大小物体的碰撞距离

如图 9-8 所示，两个大小不同的小球再次刚刚接触。左边那个占了 60 个像素，右边那个占了 40 个像素。也就是说，一个球的半径是 30，另一个球的半径是 20。这样，它们的最小距离就是 50。当然，因为在 Ball 类里面已经添加了一个 radius（半径）属性，所以可以直接使用它。

一个模式浮现了出来。碰撞距离就是一个球的半径加上另一个球的半径。现在可以删除代码中硬编码的数字了，把 drawFrame 函数改为这样（可以在文件 05-distance-2.html 中找到）：

```
function drawFrame () {
  context.clearRect(0, 0, canvas.width, canvas.height);

  ballB.x = mouse.x;
  ballB.y = mouse.y;

  var dx = ballB.x - ballA.x,
      dy = ballB.y - ballA.y,
      dist = Math.sqrt(dx * dx + dy * dy);
```

```
    if (dist < ballA.radius + ballB.radius) {
        log.value = "Hit!";
    } else {
        log.value = '';
    }

    ballA.draw(context);
    ballB.draw(context);
}
```

改变一个或两个球的大小（可以在实例化该物体时从第一个参数传入半径的值）再观察结果。即使一个球很大，另一个很小，运行结果依然正常。声明小球的代码如下：

```
var ballA = new Ball(Math.random() * 100),
    ballB = new Ball(Math.random() * 100);
```

这样，每次运行都会产生不同大小的小球，但是碰撞检测依然正常工作。

9.3.2 弹性碰撞

现在没法给出一个很好的基于距离碰撞检测的例子，因为一个完整的程序肯定在很多地方要用到一些还没学过的内容，比如，两个球碰撞后如何反应，怎么高效地处理多个对象碰撞的交互。我们编写一个例子，尽量做到既能说明问题，又不会包含太多陌生的内容。

基本的想法是创建一个大球，叫做 centerBall，放在 canvas 的中央。然后加入多个随机大小和随机速度的小球，让它们做匀速运动，遇到墙就反弹。在每一帧中使用基于距离的方法来检测小球和中央大球是否碰撞。如果发生了碰撞，则计算弹动目标点和两球间的最小距离来避免小球完全撞入大球。上面的意思是，如果一个运动的小球碰到了中央大球，就会弹回去。这通过在中央大球外设置目标点，然后让小球向目标点弹动来实现。一旦小球到达目标点，就不再继续碰撞，弹性运动就结束了，继续做匀速运动。

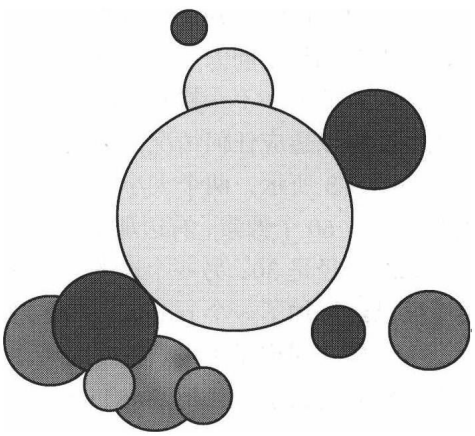


图 9-9 弹性碰撞

运行效果就像是一群小气泡在大气泡上反弹。如图 9-9 所示。小气泡撞入大气泡一点距离，这个距离取决于小气泡的速度，然后被弹出来。

下面是本例的代码（文件 06-bubbles-1.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Bubbles 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
```

```
<script>
window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      centerBall = new Ball(100, "#cccccc"),
      balls = [],
      numBalls = 10,
      spring = 0.03,
      bounce = -1;

  centerBall.x = canvas.width / 2;
  centerBall.y = canvas.height / 2;

  for (var ball, i = 0; i < numBalls; i++) {
     ball = new Ball(Math.random() * 40 + 5, Math.random() * 0xffffff);
     ball.x = Math.random() * canvas.width / 2;
     ball.y = Math.random() * canvas.height / 2;
     ball.vx = Math.random() * 6 - 3;
     ball.vy = Math.random() * 6 - 3;
     balls.push(ball);
  }

  function move (ball) {
    ball.x += ball.vx;
    ball.y += ball.vy;

    if (ball.x + ball.radius > canvas.width) {
       ball.x = canvas.width - ball.radius;
       ball.vx *= bounce;
    } else if (ball.x - ball.radius < 0) {
       ball.x = ball.radius;
       ball.vx *= bounce;
    }
    if (ball.y + ball.radius > canvas.height) {
        ball.y = canvas.height - ball.radius;
        ball.vy *= bounce;
    } else if (ball.y - ball.radius < 0) {
        ball.y = ball.radius;
        ball.vy *= bounce;
    }
  }

  function draw (ball) {
    var dx = ball.x - centerBall.x,
         dy = ball.y - centerBall.y,
         dist = Math.sqrt(dx * dx + dy * dy),
         min_dist = ball.radius + centerBall.radius;

    if (dist < min_dist) {
      var angle = Math.atan2(dy, dx),
           tx = centerBall.x + Math.cos(angle) * min_dist,
           ty = centerBall.y + Math.sin(angle) * min_dist;
      ball.vx += (tx - ball.x) * spring;
      ball.vy += (ty - ball.y) * spring;
```

```
        }
        ball.draw(context);
      }

      (function drawFrame () {
        window.requestAnimationFrame(drawFrame, canvas);
        context.clearRect(0, 0, canvas.width, canvas.height);

        balls.forEach(move);
        balls.forEach(draw);
        centerBall.draw(context);
      }());
    };
    </script>
  </body>
</html>
```

代码很多，但是大多数技术点在前面的章节都介绍过。我们快速地解释一下。

从头开始看，首先创建一个 centerBall，然后循环创建多个小球并让小球运动起来。小球的颜色、大小、速度和位置都随机。

在 drawFrame 函数里遍历了两次 ball。为了功能分离，动画代码放在 move 函数里。每次它从 ball 里接受一个 ball 的引用，传入 move（它是具有弹动的基本速度代码）来执行动画。然后，在 draw 函数里，计算小球和 CenterBall 的距离、最小距离，判断是否碰撞。如果碰撞了，则计算两者之间的角度，再用角度和最小距离算出目标点的 x 和 y，目标点恰好在 CenterBall 的边缘上。

然后，用基础弹动代码使小球向目标点弹动（如第 8 章所述）。当然，一旦它到达了目标点，它就不再继续碰撞，而是反弹回去。最后把小球绘制在 canvas 上。

瞧，我们如何用简单的技术组合起来得到非常复杂的动画效果！

9.4 多物体的碰撞检测策略

如果屏幕上有只有两三个物体在运动，检测它们是否碰撞很简单。但是如果有许多个物体，你就需要一些策略来确保不漏掉任何可能的碰撞。而且，检测的物体越多，你就越要多考虑算法的效率。

9.4.1 基础的多物体碰撞检测

只有两个物体的话，只可能是它们两个碰撞——*A* 撞 *B*。有 3 个物体的话，就有 3 种碰撞的可能：*A-B*，*B-C* 和 *C-A*。4 个物体就有 6 种碰撞的可能，5 个物体就有 10 种可能。如果有 20 个物体，你就要考虑 190 种碰撞可能！这意味着，在 drawFrame 函数中，要进行 190 次碰撞检测。

190 次足够了，肯定不需要更多的碰撞检测次数了。但是许多初学者往往会写出一些低效的算法，有人写出的算法的检测次数甚至会比最优算法多两倍以上！这样的话，20 个物体就要进行 380 次检测（20 个物体分别与其他 19 个物体检测碰撞，20 × 19=380）。现在明白为什么这个话题如此重要了吧。

为了理解这个问题，我们先来看一下要做什么，以及通常是怎么做的。如果你有 6 个物体，分别为 object0、object1、object2、object3、object4 和 object5，它们都是运动的。你想要检测它们之间是否碰撞，首先想到的肯定是双重循环：外层循环每次拿出一个物体，内层循环再把它依次与其他物体相比较。下面是伪代码：

```
objects.forEach(function (objectA, i) {
  for (var j = 0; j < objects.length; j++) {
     var objectB = objects[j];
     if (hitTestObject(objectA, objectB)) {
         //do something
    }
  }
});
```

对于 6 个物体，测试循环了 36 次。看起来还挺对的，不过这段代码有两个大问题。

首先，看第一次循环做了些什么：因为变量 i 和 j 都是 0，所以现在 objectA 和 objectB 都指向 object0。这意味着，你在用同一个物体和它自己做碰撞检测，这是在浪费资源。你可以在碰撞检测之前，用 objectA!=objectB 判断一下，或者简单地用 i!=j 也是一样的。代码如下：

```
objects.forEach(function (objectA, i) {
  for (var j = 0; j < objects.length; j++) {
     var objectB = objects[j];
     if (i !== j && hitTestObject(objectA, objectB)) {
       //do something
     }
   }
});
```

这样节省了 6 次碰撞检测，检测次数现在降到了 30 次，但还是太多。下面列出了这段代码做的每一次检测：

```
object0  与  object1, object2, object3, object4, object5
object1  与  object0, object2, object3, object4, object5
object2  与  object0, object1, object3, object4, object5
object3  与  object0, object1, object2, object4, object5
object4  与  object0, object1, object2, object3, object5
object5  与  object0, object1, object2, object3, object4
```

首先看第一行的第一次检测：object0 和 object1，然后看第二行的第一次检测：object1 和 object0，干了同样的事情！如果 object0 没碰上 object1，那么 object1 肯定也没碰上 object0。或者，如果 object0 碰上了 objcet1，那么 object1 肯定碰上了 object0。上面有很多类似的重复。如果去掉了重复的检测，应该是这样的：

```
object0  与  object1, object2, object3, object4, object5
object1  与  object2, object3, object4, object5
object2  与  object3, object4, object5
object3  与  object4, object5
object4  与  object5
object5  与  没有了
```

你可以看到在第一轮检测中，object0 会与其他所有物体做碰撞检测。因为其他几个物体没必要再与 object0 做检测了，所以在第二轮中，把 object0 去掉了，object1 和剩下的 4 个物体做碰撞检测，然后把 object1 从列表中去掉。到最后一轮时，只剩下 object5 了，因为其他几个物

体都已经和它做过碰撞检测了，没必要再把它与其他物体做碰撞检测了。这次，碰撞检测的次数下降到了 15 次。现在你可以看到为什么我们一开始的解决方案做了两倍的计算量。

为了把上面的方案写成代码，仍然需要双重循环，一个 for 循环嵌套在 forEach 里，如下所示：

```
objects.forEach(function (objectA, i) {
  for (var j = i + 1; j < objects.length; j++) {
     var objectB = objects[j];
     if (hitTestObject(objectA, objectB) {
        //do whatever
     }
   }
});
```

内层循环的索引总是从外层索引加 1 开始，这是因为前面物体的已经检测过了，没必要再做一次检测，这样就减少了检测次数。代码只是在前一个版本上改了几个字符，但是性能提高了一倍！

即使不考虑性能问题，很多时候重复的碰撞检测也会导致异常结果。比如，如果当你检测到碰撞的时候需要改变物体的速度，这样重复的检测就会导致重复修改速度，从而导致不可预料的结果，而且很难调试。

如果你对这个话题感兴趣，还想深入学习。可以了解组合数学，这是一个研究选择和排列的数学领域，通俗来讲，就是研究“如何计算不计算”。

9.4.2 多物体弹动

我们继续结合例子来学习，继续使用气泡效果，不过这次所有的气泡之间都可以相互反弹，效果如图 9-10 所示。

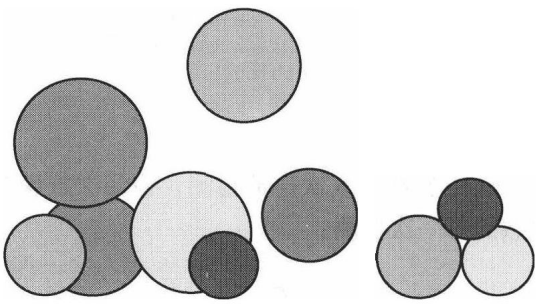


图 9-10 多物体碰撞

下面是文件 07-bubbles-2.html 中的代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Bubbles 2</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
         balls = [],
         numBalls = 10,
         bounce = -0.5,
         spring = 0.03,
```

```
    gravity = 0.1;

for (var ball, i = 0; i < numBalls; i++) {
  ball = new Ball(Math.random() * 30 + 20, Math.random() * 0xffffff);
  ball.x = Math.random() * canvas.width / 2;
  ball.y = Math.random() * canvas.height / 2;
  ball.vx = Math.random() * 6 - 3;
  ball.vy = Math.random() * 6 - 3;
  balls.push(ball);
}

function checkCollision (ballA, i) {
  for (var ballB, dx, dy, dist, min_dist, j = i + 1; j < numBalls; j++) {
    ballB = balls[j];
    dx = ballB.x - ballA.x;
    dy = ballB.y - ballA.y;
    dist = Math.sqrt(dx * dx + dy * dy);
    min_dist = ballA.radius + ballB.radius;

    if (dist < min_dist) {
      var angle = Math.atan2(dy, dx),
          tx = ballA.x + Math.cos(angle) * min_dist,
          ty = ballA.y + Math.sin(angle) * min_dist,
          ax = (tx - ballB.x) * spring * 0.5,
          ay = (ty - ballB.y) * spring * 0.5;

      ballA.vx -= ax;
      ballA.vy -= ay;
      ballB.vx += ax;
      ballB.vy += ay;
    }
  }
}

function move (ball) {
  ball.vy += gravity;
  ball.x += ball.vx;
  ball.y += ball.vy;

  if (ball.x + ball.radius > canvas.width) {
    ball.x = canvas.width - ball.radius;
    ball.vx *= bounce;
  } else if (ball.x - ball.radius < 0) {
    ball.x = ball.radius;
    ball.vx *= bounce;
  }
  if (ball.y + ball.radius > canvas.height) {
    ball.y = canvas.height - ball.radius;
    ball.vy *= bounce;
  } else if (ball.y - ball.radius < 0) {
    ball.y = ball.radius;
    ball.vy *= bounce;
  }
}
```

```
    function draw (ball) {
      ball.draw(context);
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      balls.forEach(checkCollision);
      balls.forEach(move);
      balls.forEach(draw);
    }());
  };
  </script>
 </body>
</html>
```

这里简单地使用双重循环来做碰撞检测。不过，这次还需要做一些补充说明，下面是碰撞后的反应代码，可以在 checkCollision 函数中看到：

```
if (dist < min_dist) {
  var angle = Math.atan2(dy, dx),
      tx = ballA.x + Math.cos(angle) * min_dist,
      ty = ballA.y + Math.sin(angle) * min_dist,
      ax = (tx - ballB.x) * spring * 0.5,
      ay = (ty - ballB.y) * spring * 0.5;

  ballA.vx -= ax;
  ballA.vy -= ay;
  ballB.vx += ax;
  ballB.vy += ay;
}
```

上面的代码会在检测到 ballA 和 ballB 碰撞后执行。它和前面小球碰撞中央大球的例子本质上一样。现在，我们只不过是把 ballA 当做中央大球。你计算出两球间的角度，计算 ballB 目标点的 x、y，注意加上偏移量，以防两个球互相重叠。计算出 ballB 在 x 轴和 y 轴上的加速度 ax 和 ay，然后把加速度减半（这个下面会解释）。

这里有个小技巧：在本例中，不仅 ballB 要从 ballA 弹开，而且 ballA 要从 ballB 弹开，它们加速度的绝对值相同，方向相反。所以加速度不必计算两次，只需要把 ax 和 ay 加在 ballB 的速度上，然后把它们从 ballA 的速度中减去就行。结果是一样的，不过避免了一次加速度的计算。也许你会认为这样会使最终加速度扩大了一倍，确实是这样的。为了抵消，在计算 ax 和 ay 的时候，给它乘以 0.5，或者在声明 spring 变量时，把它减半，设为 0.015，这样就可以避免两次乘法运算。

说到优化技巧，这里还有一个：用 Math.atan2 预先计算角度，然后用 Math.cos 和 Math.sin 计算目标点坐标：

```
var angle = Math.atan2(dy, dx),
    tx = ballA.x + Math.cos(angle) * min_dist,
    ty = ballA.y + Math.sin(angle) * min_dist;
```

但是别忘了，正弦是对边与斜边之比，余弦是临边与斜边之比。因为这个角的对边是 dy，

临边是 dx，斜边是 dist。这样上面的三行代码就能缩减到两行：

```
var tx = ball1A.x + dx / dist * min_dist,
    ty = ball1A.y + dy / dist * min_dist;
```

这是把三行需要用到三角函数的复杂代码缩减到了两行简单的除法代码。

你可以在文件 08-bubbles-3.html 里看到优化后的例子，可以在 www.apress.com 上下载到。在继续下面的内容之前，花些时间动手调试一下这个例子，可以试着调整小球的 spring、gravity、number 和 size 值。你可能还想试着加入摩擦力或者一些鼠标交互效果。

9.5 本章中的重要公式

是时候回顾一下本章中的两个重要公式了。

9.5.1 基于距离的碰撞检测

```
//starting with objectA and objectB
//if using an object without a radius property,
//you can use width or height divided by 2
var dx = objectB.x - objectA.x,
     dy = objectB.y - objectA.y,
     dist = Math.sqrt(dx * dx + dy * dy);

if (dist < objectA.radius + objectB.radius) {
    //handle collision
}
```

9.5.2 多物体碰撞检测

```
objects.forEach(function (objectA, i) {
  for (var j = i + 1; j < objects.length; j++) {
     //evaluate reference using j. For example:
     var objectB = objects[j];
     //perform collision detection between objectA and objectB
  }
});
```

9.6 小结

本章已经讨论了你需要了解的关于碰撞检测的所有知识，包括基于几何图形的碰撞检测和

```
    ball.y += ball.vy;

    if (ball.x + ball.radius > bounds.x && ball.x - ball.radius < bounds.x + bounds.width) {
        //all the rest of the code that was in this function
    }

    ball.draw(context);
    line.draw(context);
}());
```

可以在文件 08-angle-bounce-bounds.html 中找到改动后的代码。

10.3.5 修复“线下”问题

在检测碰撞时，首先要判断小球是否在直线附近，然后进行坐标旋转，得到旋转后的位置和速度。接着，判断小球旋转后的纵坐标 y2 是否越过了直线，如果超过了，则执行反弹。

但是如果小球位于直线下方该怎么办呢？比如，我们想要这样的效果：直线在 canvas 的中间，小球在“地板”上弹跳。如果碰撞检测和边界框检测都返回 true，程序会认为小球在直线上反弹了，它就会把小球从直线下移到直线上。

有一个解决方案是比较 vy1 和 y2，仅当 vy1 大于 y2 的时候才执行反弹。如图 10-8 所示。

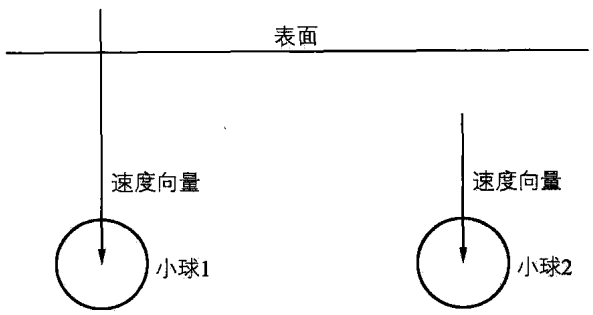


图 10-8 正穿过直线还是已经到了线下

左边小球在 y 轴上的速度大于它与直线的相对距离，这意味着，它刚刚从直线上穿越下来。右边小球的速度向量小于它和直线的相对距离，也就是说，它在这一帧和上一帧中都位于线下，因此它只是在线下运动。需要做的只是在小球穿过直线的那一瞬间才执行反弹，下面来看一段修改过的 drawFrame 函数：

```
//rotate coordinates
var y2 = y1 * cos - x1 * sin;

//perform bounce with rotated values
if (y2 > -ball.radius) {
    //rotate coordinates
    var x2 = x1 * cos + y1 * sin,

        //rotate velocity
        vx1 = ball.vx * cos + ball.vy * sin,
        vy1 = ball.vy * cos - ball.vx * sin;
```

需要把 y2 < vy1 加入到 if 语句中：

```
if (y2 > -ball.radius && y2 < vy1) { ...
```

但是为此，就要预先计算 vy1。所以把计算 vy1 的代码从 if 语句块中提出来，代码修改如下：

```
//rotate coordinates
var y2 = y1 * cos - x1 * sin,
```

```
        //rotate velocity
        vy1 = ball.vy * cos - ball.vx * sin;

    //perform bounce with rotated values
    if (y2 > -ball.radius && y2 < vy1) {
        //rotate coordinates
        var x2 = x1 * cos + y1 * sin,

            //rotate velocity
            vx1 = ball.vx * cos + ball.vy * sin;
```

使用这段代码，每一帧都会多出一些计算量，但是会提高精度和真实感，这就需要权衡，找到一个平衡点。如果小球不可能运动到线下，这些判断就不需要了，就可以把计算 vy1 的代码放进 if 语句块里面，从而减少计算量。

文件 09-angle-bounce-final.html 加入了墙面和地板的反弹效果，你可以看到小球最终到了直线下方。

10.3.6 从多个斜面反弹

到目前为止，都是在一条单独的直线或斜面上反弹。从多个斜面上反弹也不复杂，只需要创建多个斜面并循环就可以做到。可以把斜面反弹的代码抽象到一个函数里，然后在循环中调用。

下一个练习是个完整的程序，它用到了前面章节中学到的所有技术。本例与前面几个例子类似，继续使用 ball 和 line 对象。不同的是这里有多条直线，每条都短一些。在 canvas 上放置了一个小球和五条直线，直线保存在一个名字叫 lines 的数组中，如图 10-9 所示。

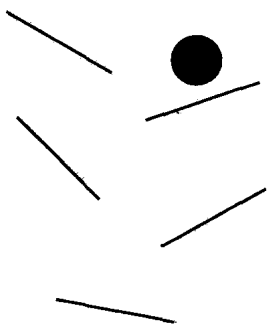


图 10-9 多条直线

以下是本例的代码（可以在 10-multi-angle-bounce.html 中找到）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Multi Angle Bounce</title>
  <link rel="stylesheet" href="style.css">
 </head>
<body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script src="line.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(20),
        lines = [],
        gravity = 0.2,
        bounce = -0.6;
```

```
ball.x = 100;
ball.y = 50;

//create 5 lines, position and rotate
lines[0] = new Line(-50, 0, 50, 0);
lines[0].x = 100;
lines[0].y = 100;
lines[0].rotation = 30 * Math.PI / 180;

lines[1] = new Line(-50, 0, 50, 0);
lines[1].x = 100;
lines[1].y = 200;
lines[1].rotation = 45 * Math.PI / 180;

lines[2] = new Line(-50, 0, 50, 0);
lines[2].x = 220;
lines[2].y = 150;
lines[2].rotation = -20 * Math.PI / 180;

lines[3] = new Line(-50, 0, 50, 0);
lines[3].x = 150;
lines[3].y = 330;
lines[3].rotation = 10 * Math.PI / 180;

lines[4] = new Line(-50, 0, 50, 0);
lines[4].x = 230;
lines[4].y = 250;
lines[4].rotation = -30 * Math.PI / 180;

function checkLine (line) {
  var bounds = line.getBounds();

  if (ball.x + ball.radius > bounds.x && ball.x - ball.radius < bounds.x + bounds.width) {
     //get sine and cosine of angle
     var cos = Math.cos(line.rotation),
          sin = Math.sin(line.rotation),

          //get position of ball, relative to line
          x1 = ball.x - line.x,
          y1 = ball.y - line.y,

          //rotate coordinates
          y2 = y1 * cos - x1 * sin,

          //rotate velocity
          vy1 = ball.vy * cos - ball.vx * sin;

     //perform bounce with rotated values
     if (y2 > -ball.radius && y2 < vy1) {
        //rotate coordinates
        var x2 = x1 * cos + y1 * sin,

             //rotate velocity
             vx1 = ball.vx * cos + ball.vy * sin;

        y2 = -ball.radius;
```

```
            vy1 *= bounce;

            //rotate everything back
            x1 = x2 * cos - y2 * sin;
            y1 = y2 * cos + x2 * sin;
            ball.vx = vx1 * cos - vy1 * sin;
            ball.vy = vy1 * cos + vx1 * sin;
            ball.x = line.x + x1;
            ball.y = line.y + y1;
          }
        }
      }

      function drawLine (line) {
        checkLine(line);
        line.draw(context);
      }

      (function drawFrame () {
        window.requestAnimationFrame(drawFrame, canvas);
        context.clearRect(0, 0, canvas.width, canvas.height);

        //normal motion code
        ball.vy += gravity;
        ball.x += ball.vx;
        ball.y += ball.vy;

        //bounce off ceiling, floor, and walls
        if (ball.x + ball.radius > canvas.width) {
            ball.x = canvas.width - ball.radius;
            ball.vx *= bounce;
        } else if (ball.x - ball.radius < 0) {
            ball.x = ball.radius;
            ball.vx *= bounce;
        }
        if (ball.y + ball.radius > canvas.height) {
            ball.y = canvas.height - ball.radius;

            ball.vy *= bounce;
        } else if (ball.y - ball.radius < 0) {
            ball.y = ball.radius;
            ball.vy *= bounce;
        }

        lines.forEach(drawLine);
        ball.draw(context);
      }());
    };
    </script>
  </body>
</html>
```

本例的代码很多，但是你应该都能看得懂。复杂的程序并不是必须由复杂的片段组成，它们通常由一些简单的片段以合适的方式组合而成。在本例中，checkLine 函数与上一个例子的 drawFrame 函数的功能是一样的，只不过在一帧中调用了它 5 次而已。

10.4 本章中的重要公式

这里回顾一下本章介绍的两个主要公式。

10.4.1 坐标旋转

```
x1 = x * Math.cos(rotation) - y * Math.sin(rotation);
y1 = y * Math.cos(rotation) + x * Math.sin(rotation);
```

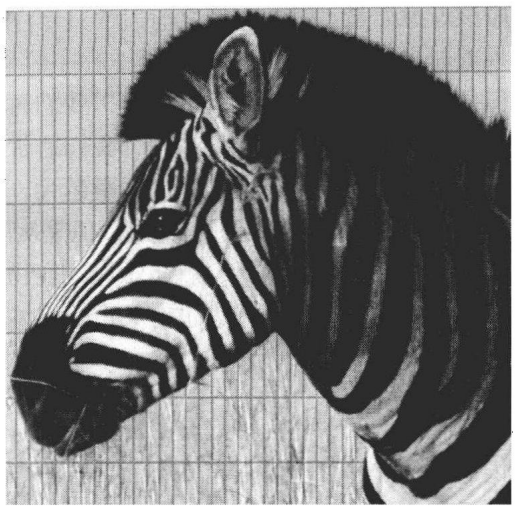
10.4.2 反向坐标旋转

```
x1 = x * Math.cos(rotation) + y * Math.sin(rotation);
y1 = y * Math.cos(rotation) - x * Math.sin(rotation);
```

10.5 小结

正如你在本章中所看到的，坐标旋转可以用来实现非常复杂的行为，但是它总是可以归结为两个不变的公式。一旦你熟练掌握了这些公式，你就可以把它们用在任何地方。我希望你已经慢慢开始体会到如何通过加入一些简单的技术来实现非常复杂的、效果逼真的动画。

在第 12 章中，你将要学习如何处理不同速度和质量的物体间的碰撞，坐标旋转公式还将大量用到。



第 11 章　撞球物理

本章涵盖以下内容：

- 质量
- 动量
- 动量守恒

本书正如你所期望的技术书籍一样，从简单的内容讲起，逐步深入。本章的内容已经到了复杂的顶峰。这并不是说本章之后的内容会变得简单，而是要求大家不要跳过前面的任何内容。我们需要一步一步来学习这些概念，如果前面的内容你都能理解，那就很好。

具体地说，本章着重介绍动量：两个物体碰撞后动量如何变化，动量守恒原理，以及如何将动量守恒应用在动画中。

因为例子中的物体都是圆形，所以为了简单起见，这个主题就叫做“撞球物理”。你很快就能看到这些不同大小的撞球相互碰撞的例子。

与前面的章节一样，为了简单起见，先从一维坐标上的例子开始，然后再过渡到二维坐标上，这时就需要用到坐标旋转（第 10 章的主题）。本质上，你将要把二维坐标场景旋转为平面，这样就可以和一维坐标的场景一样，只关心一条轴。让我们先从质量和动量的概念开始吧。

11.1　质量

本书前面的章节从多个方面介绍了运动：速度、加速度、向量、摩擦力、反弹、缓动、弹动以及重力。不过到目前为止，我们都没有考虑物体质量对运动的影响。科学地讲，在方程中

应该考虑质量。但是我们之前通常只专注于把大部分重要的事情做正确，并且只着重于保证效果看起来是正确的，最重要的是，程序效率必须非常高，以保证 Web 浏览器能流畅地播放动画。但是，质量与动量密不可分，所以我们再也不能忽略它了。

什么是质量呢？在地球上，我们通常认为质量就代表物体有多重。它们确实关系紧密，因为重量与质量成正比。物体的质量越大，重量也就越大。实际上，我们用同样的单位来测量质量和重量，比如，公斤、英镑等。但是从严格意义上来说，质量是指物体保持运动速度的能力。因此，物体的质量越大，就越难以改变物体的运动状态（减速、加速或改变方向）。

质量与加速度、外力也有关系。物体的质量越大，就需要对它施加越大的外力来产生一定的加速度。用方程表示如下：

```
F = m × a
```

举个例子，一辆小汽车的发动机是基于小汽车的质量设计的，它有足够的动力为小汽车提供适当的加速度。但是它并不能为一辆大卡车提供相同的加速度，因为卡车的质量更大，发动机需要更大的动力。

11.2 动量

现在我们开始学习动量，它是物体的质量与速度的乘积。换句话说，就是质量乘以速度。动量用字母 p 来表示，质量用 m 表示，公式如下：

```
p = m × v
```

这意味着，一个质量小、速度大的物体可以和另一个质量大、速度小的物体拥有差不多的动量。比如，前面提到的卡车以每小时 20 英里的速度运动，或者一个质量很小的子弹以非常高的速度运动，它们都可以瞬间杀死人。这里你可以看到两个质量和速度都不同的物体为什么可以有相等的动量（m/s 表示米/每秒）：

```
4 kg × 15 m/s = 20 kg × 3 m/s = 60 kg m/s
```

用这个公式，一个 4 千克的球从山上以 15 米/秒的速度滚下来，它的动量就是 60kg m/s。因为速度是一个向量（方向和大小），所以动量也是一个向量。动量向量的方向与速度向量的方向相同。因此动量的完整描述如下：

```
4 kg × 15 m/s at 23 degrees
```

有了这些背景知识，下面开始学习如何把它们应用到碰撞中。

11.3 动量守恒

到目前为止，你应该已经比较熟悉碰撞了。我们用了一整章来介绍碰撞检测，并且模拟了两个物体间的一些碰撞效果。动量守恒是制作真实的碰撞效果的基本原理。

使用动量守恒原理，可以确定两个物体碰撞后如何反应。因此你可以说：“碰撞前，一个物体以速度 A 运动，另一个物体以速度 B 运动；碰撞后，一个物体速度变为了 C，另一个物体

的速度变为了 *D*。”进一步分解来看，因为速度只由大小和方向组成，如果已知两个物体碰撞前的速度大小和运动方向，就能计算出碰撞后的速度大小和运动方向，这是很有用的。

但是有一个前提：需要已知每个物体的质量。这就是说，如果已知碰撞前每个物体的质量以及速度大小和运动方向，就能计算出它们碰撞后的速度大小和运动方向。

这就是动量守恒的作用，但到底什么是动量守恒呢？动量守恒定理是一个基本的物理概念：系统在碰撞前的总动量等于系统在碰撞后的总动量。但是定理中所指的系统是什么呢？它是指一些拥有动量的物体的集合。大多数时候这指的是一个**封闭系统**，也就是说，它不受外力影响。换句话说，可以忽略除了碰撞以外的任何因素。对我们来说，我们只考虑两个物体碰撞后的反应，所以系统就是物体 *A* 和物体 *B*。

系统的总动量是系统内所有物体的组合动量。对我们来说，就是物体 *A* 和物体 *B* 的组合动量。如果在碰撞前计算一次总动量，在碰撞后再计算一次总动量，两次计算的结果应该一样。

在跳转到数学知识之前，这里有一个建议。先不要担心如何把这些转化为代码，后面很快会讲到。首先从概念的角度上来学习下面几个公式，“*A* 加 *B* 等于 *B* 加 *A*”。在本章的最后会把它们转化为代码。

如果总动量在碰撞前后相同，并且动量等于速度乘以质量，那么对于两个物体 object0 和 object1，可以得出如下结论：

```
momentum0 + momentum1 = momentum0Final + momentum1Final
```

或者

$$(m0 \times v0) + (m1 \times v1) = (m0 \times v0Final) + (m1 \times v1Final)$$

现在要计算 object0 和 object1 的最终速度，也就是 *v*0Final 和 *v*1Final。解一个有两个未知数方程的方法，就是找出另一个含有相同两个未知数的方程，物理学的殿堂中恰好有这么一个方程。它与动能有关，但是你不必了解，甚至不用关心动能是什么。你只需要借用这个公式来解你的方程。下面是动能公式：

$$KE = 0.5\ m \times v^2$$

严格来讲，动能不是一个向量，所以尽管使用 *v* 表示速度，但是它是表示速度的大小，并不关心方向。但是这并不影响计算。

动能在碰撞前后也是相同的，所以可以这样做：

$$KE0 + KE1 = KE0Final + KE1Final$$

或者

$$(0.5 \times m0 \times v0^2) + (0.5 \times m1 \times v1^2) = (0.5 \times m0 \times v0Final^2) + (0.5 \times m1 \times v1Final^2)$$

消去 0.5 得到：

$$(m0 \times y0^2) + (m1 \times v1^2) = (m0 \times v0Final^2) + (m1 \times v1Final^2)$$

现在得到了一个不同的方程，它含有相同的两个未知数 *v*0Final 与 *v*1Final。使用“代入消元法”可以得到计算两个未知数的公式：

$$v0Final = \frac{(m0 - m1) \times v0 + 2 \times m1 \times v1}{m0 + m1}$$

$$v1Final = \frac{(m1 - m0) \times v1 + 2 \times m0 \times v0}{m0 + m1}$$

现在你了解我为什么说到了最复杂的地方了吧。事实上，你还没到。下面你会把这些应用到单轴上，继而更加深入地应用到双轴上，这时还要加入坐标旋转。坚持！

11.3.1 单轴上的动量守恒

既然你已经得到了公式，就可以开始制作动画了。第一个例子依然使用 **Ball** 类，但是要加入一个 **mass** 属性，下面是新的代码（**ball.js**）：

```
function Ball (radius, color) {
  if (radius === undefined) { radius = 40; }
  if (color === undefined) { color = "#ff0000"; }
  this.x = 0;
  this.y = 0;
  this.radius = radius;
  this.vx = 0;
  this.vy = 0;
  this.mass = 1;
  this.rotation = 0;
  this.scaleX = 1;
  this.scaleY = 1;
  this.color = utils.parseColor(color);
  this.lineWidth = 1;
}
Ball.prototype.draw = function (context) {
  context.save();
  context.translate(this.x, this.y);
  context.rotate(this.rotation);
  context.scale(this.scaleX, this.scaleY);
  context.lineWidth = this.lineWidth;
  context.fillStyle = this.color;
  context.beginPath();
  context.arc(0, 0, this.radius, 0, (Math.PI * 2), true);
  context.closePath();
  context.fill();
  if (this.lineWidth > 0) {
    context.stroke();
  }
  context.restore();
};

Ball.prototype.getBounds = function () {
  return {
    x: this.x - this.radius,
    y: this.y - this.radius,
    width: this.radius * 2,
    height: this.radius * 2
  };
};
```

在下面的例子（文件 **01-billiard-1.html**）中，创建两个大小、位置和质量都不相同的小球。

先忽略 y 轴，因此小球处于同一竖直位置上。在浏览器中查看的话，大概如图 11-1 所示。

在脚本的开头先创建和初始化两个小球。在 **drawFrame** 动画循环中，放置单轴上的基本运动代码和基于距离的简单碰撞检测代码，随后我们很快会加入碰撞后的反应代码。

图 11-1 单轴上的动量守恒：小球设置的预期效果

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Billiard 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball0 = new Ball(),
        ball1 = new Ball();

    ball0.mass = 2;
    ball0.x = 50;
    ball0.y = canvas.height / 2;
    ball0.vx = 1;

    ball1.mass = 1;
    ball1.x = 300;
    ball1.y = canvas.height / 2;
    ball1.vx = -1;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      ball0.x += ball0.vx;
      ball1.x += ball1.vx;
      var dist = ball1.x - ball0.x;

      if (Math.abs(dist) < ball0.radius + ball1.radius) {
          //reaction will go here
      }

      ball0.draw(context);
      ball1.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

基础的代码都到位了，现在我们来看如何处理碰撞后的反应。先来看 **ball0**，把 **ball0** 作为物体 0，**ball1** 作为物体 1，需要应用如下公式：

$$v0Final = \frac{(m0 - m1) \times v0 + 2 \times m1 \times v1}{m0 + m1}$$

在 JavaScript 中用如下代码表示：

```
var vx0Final = ((ball0.mass - ball1.mass) * ball0.vx + 2 * ball1.mass * ball1.vx) / (ball0.mass+
ball1.mass);
```

应该不难理解这段代码是如何得出的。可以用同样的方法计算 **ball1**，像这样：

$$v1Final = \frac{(m1 - m0) \times v1 + 2 \times m0 \times v0}{m0 + m1}$$

得出代码：

```
var vx1Final = ((ball1.mass - ball0.mass) * ball1.vx + 2 * ball0.mass * ball0.vx) / (ball0.mass+
ball1.mass);
```

加入了反应代码之后，完整的 **drawFrame** 函数是这样的：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  ball0.x += ball0.vx;
  ball1.x += ball1.vx;
  var dist = ball1.x - ball0.x;

  if (Math.abs(dist) < ball0.radius + ball1.radius) {
    var vx0Final = ((ball0.mass - ball1.mass) * ball0.vx + 2 * ball1.mass * ball1.vx) /
                   (ball0.mass + ball1.mass),
        vx1Final = ((ball1.mass - ball0.mass) * ball1.vx + 2 * ball0.mass * ball0.vx) /
                   (ball0.mass + ball1.mass);
    ball0.vx = vx0Final;
    ball1.vx = vx1Final;

    ball0.x += ball0.vx;
    ball1.y += ball1.vx;
  }

  ball0.draw(context);
  ball1.draw(context);
}());
```

因为计算动量要用到每个小球的速度，所以需要把速度的计算结果保存到临时变量 **vx0Final** 和 **vx1Final** 中，而不是直接把结果赋值给 **ball0.vx** 和 **ball1.vx**。

调整物体位置

碰撞反应代码的最后两行需要解释一下。在求出了每个小球的新速度后，把它们累加在小球的位置上。如果你回想一下第 6 章中关于反弹的例子就会发现，为了不出现小球陷入墙壁的效果，当它碰到边界时就要调整位置。否则，小球可能会忽略边界并且在随后的帧上，在边界上振荡运动。本例中存在同样的问题，只不过这里的情况是你不希望两个移动的物体嵌在一起。

可以把一个小球调整到另一个小球的边缘上，但是移动哪个呢？无论移动哪个，都会看起来像是跳跃一样不自然，在速度比较慢的时候尤其明显。

有不少方法都可以用来计算小球的调整位置，这些方法有的简单有的复杂，有的精确有的粗糙。在第一个例子中选用了一个简单的方案，把新速度加在物体的位置上，再次让它们弹开。这个方法效果逼真，也十分简单，只要两行代码就能实现。碰撞后 **vx0Final** 的值为-1/3，**vx1Final** 为 5/3，你可以看到系统的总动量在碰撞前后没有变化：

```
(ball0.mass * ball0.vx) + (ball1.mass * ball1.v) = (2 * 1) + (1 * -1) = (2 * -1/3) + (1 * 5/3) = 1
```

在 11.3.2 小节中，你会看到这种方法有一个问题，到时候我们会介绍一个更加完善的方案。

用浏览器打开 **01-billiard-1.html** 文件，改变每个小球的质量和速度再观察效果。然后再尝试改变小球的尺寸，你可以发现尺寸对碰撞后的反应没有影响。在大多数情况下，尺寸越大的物体质量也越大，你可以体现出这种关系，依据小球的尺寸大小来设置它们的质量。但是通常情况下，我们会不断尝试调整质量的值，直到动画看起来感觉比较好。因为我们在创建一个可视的动画程序，所以有时候眼睛是最好的判断。

优化代码

这个解决方案最大的缺点是代码中有大段的方程式，几乎同样的方程出现了两次，所以我们需要消除一个。

但是这个需要用到更多的数学和代数知识。你首先需要得到两个物体的相对速度，这是它们的叠加总速度。然后，当你计算出一个物体的最终速度后，再根据前面得到的相对速度，就能计算出另一个物体的最终速度。

用两个物体的速度相减就能得到相对速度。这看起来比较奇怪，但是让我们从整个系统的视角来考虑。来想象一下系统内有两辆汽车在高速路上同向行驶，一辆汽车的速度是每小时 50 英里，另一辆汽车的速度是每小时 60 英里。当你坐在某一辆汽车里面时，就会看到另一辆汽车的速度是每小时 10 或-10 英里。换句话说，另一辆汽车在慢慢地超越或落后你。

在碰撞之前，要用 **ball1.vx** 减去 **ball0.vx** 来计算出总速度（从 **ball1** 的视角）：

```
var vxTotal = ball0.vx - ball1.vx;
```

然后在计算出 **vx0Final** 后，把它与 **vxTotal** 相加得到 **vx1Final**。这可能不是最直观的公式，但是你可以看到它可以工作：

```
vx1Final = vxTotal + vx0Final;
```

现在这样比那两个可怕的公式更容易理解一些了吧。而且，因为计算 **ball1.vx** 的公式不再引用 **ball0.vx**，所以可以消除一些临时变量。下面是改进后的 **drawFrame** 函数（在文件 **02-billiard-2.html** 里）：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  ball0.x += ball0.vx;
```

```
    ball1.x += ball1.vx;
    var dist = ball1.x - ball0.x;

    if (Math.abs(dist) < ball0.radius + ball1.radius) {
      var vxTotal = ball0.vx - ball1.vx;
      ball0.vx = ((ball0.mass - ball1.mass) * ball0.vx + 2 * ball1.mass * ball1.vx) /
                 (ball0.mass + ball1.mass);
      ball1.vx = vxTotal + ball0.vx;

      ball0.x += ball0.vx;
      ball1.y += ball1.vx;
    }

    ball0.draw(context);
    ball1.draw(context);
  }());
```

我们已经消除了不少数学计算，并且结果没变——还不错吧。

你并不需要把这个公式理解得很透彻，也不一定需要记住。但是在程序中需要用到的时候，至少你要知道在这里能找到这个公式。每当程序需要用到这个公式时，可以把书本拿出来查一下。

11.3.2 双轴上的动量守恒

深呼吸，因为你要进入下一个层次。到目前为止，你已经应用了一个冗长的公式，但是这几乎可以即插即用。你只要把两个物体的质量和速度代入公式，就能得到结果。

现在，我们再加入一层复杂性——另一个维度。可以通过坐标旋转来完成，不过我们先来看看原理。

理解原理和策略

图 11-2 表示了你刚才看到的那个例子：一维碰撞。

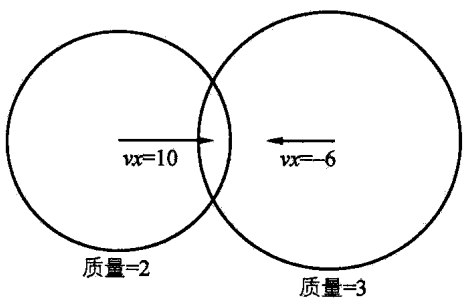


图 11-2 一维碰撞

可以看到，两个物体的尺寸、质量和速度都不相同。从每个圆心指向外的箭头表示速度向量，指向物体运动的方向，箭头的长度表示速度的大小。

一维的例子比较简单，因为两个速度向量都在 x 轴上，所以你可以直接加减它们的值。但是让我们看图 11-3，它描述两个球在二维空间中的碰撞。

因为速度的方向完全不同，所以你不能直接把速度代入动量守恒公式。那么如何解决这个问题呢？

可以旋转图 11-3 来得到与图 11-2 相似的结果。首先，计算两个小球位置形成的角度，并且逆时针旋转整个场景——位置和速度。比如，如果角度是 30°，就把所有东西旋转−30°。这和你在第 10 章的斜面反弹中所做的完全一样。结果如图 11-4 所示。

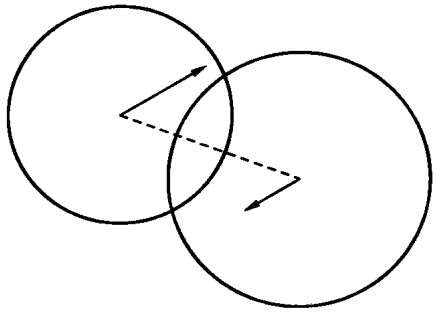


图 11-3 二维碰撞

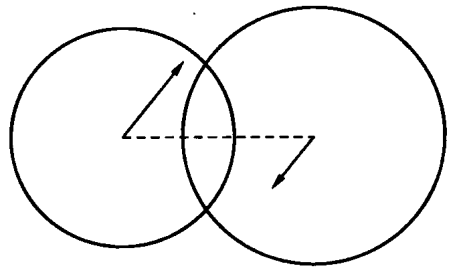


图 11-4 旋转后的二维碰撞

两球间的角度很重要，这就是**碰撞角度**。你只需要关心小球的速度位于碰撞角度上的那一部分。

看图 11-5，这里用向量箭头标出了两球的 **vx** 和 **vy**。两球的 **vx** 正好沿着碰撞的角度。

因为我们只需要关心碰撞角度上的那一部分速度——现在就是 **vx**，所以你可以忘了 **vy**。图 11-6 是去除了 **vy** 后的效果。

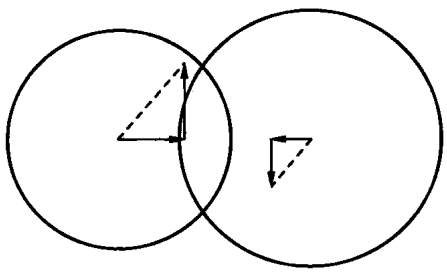


图 11-5 标出 *x*、*y* 轴上的速度

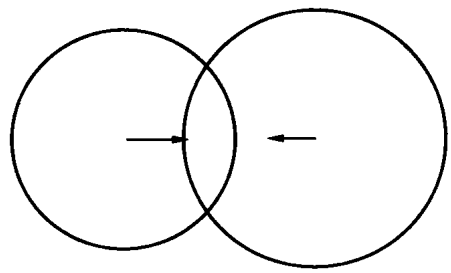


图 11-6 只需关心 *x* 轴上的速度

这应该看起来很眼熟，因为这就是图 11-2！你可以很简单地使用即插即用的动量公式来得到结果。使用这个公式得到两个新的 **vx** 值，记着 **vy** 的值永远不变，但是 **vx** 的变化单独影响了总体速度，如图 11-7 所示。

现在把所有东西都旋转回原位，如图 11-8 所示，就得到了每个球最终真实的 **vx** 和 **vy**。

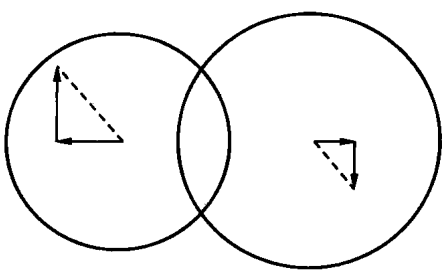


图 11-7 新的 x 速度，不变的 y 速度，新的总体速度

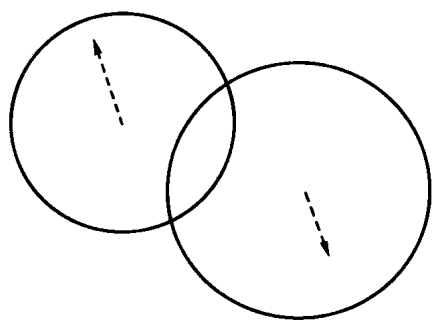


图 11-8 所有东西旋转回原位

上面就是整个过程的图示，下面把它们转化为代码。

编写代码

先来写一个可以让两个小球以任意角度运动并最终互相碰撞的程序。和前面的设置相同，有 **Ball** 的两个实例：**ball0** 和 **ball1**。这次让它们的尺寸大一些，以增加碰撞的几率，如图 11-9 所示。

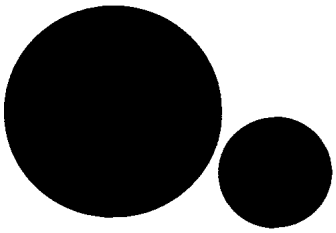


图 11-9 设置双轴上的动量守恒的对象

这里是一个示例（**03-billiard-3.html**），还没有测试它，因为还没有定义 **checkCollision** 函数，不过接下来马上就定义它。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Billiard 3</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball0 = new Ball(80),
        ball1 = new Ball(40),
        bounce = -1.0;

    ball0.mass = 2;
    ball0.x = canvas.width - 200;
    ball0.y = canvas.height - 200;
    ball0.vx = Math.random() * 10 - 5;
    ball0.vy = Math.random() * 10 - 5;

    ball1.mass = 1;
    ball1.x = 100;
    ball1.y = 100;
    ball1.vx = Math.random() * 10 - 5;
    ball1.vy = Math.random() * 10 - 5;

    function checkCollision (ball0, ball1) {
      //not defined yet...
    }

    function checkWalls (ball) {
      if (ball.x + ball.radius > canvas.width) {
         ball.x = canvas.width - ball.radius;
         ball.vx *= bounce;
      } else if (ball.x - ball.radius < 0) {
         ball.x = ball.radius;
         ball.vx *= bounce;
```

```
    }
    if (ball.y + ball.radius > canvas.height) {
       ball.y = canvas.height - ball.radius;
       ball.vy *= bounce;
    } else if (ball.y - ball.radius < 0) {
       ball.y = ball.radius;
       ball.vy *= bounce;
    }
  }

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    ball0.x += ball0.vx;
    ball0.y += ball0.vy;
    ball1.x += ball1.vx;
    ball1.y += ball1.vy;

    checkCollision(ball0, ball1);
    checkWalls(ball0);
    checkWalls(ball1);

    ball0.draw(context);
    ball1.draw(context);
  }());
 };
 </script>
</body>
</html>
```

在本例中，设置了边界和随机的速度，并加入了质量，根据每个小球的速度移动它们，检查边界。注意，把边界检查的代码放到了一个单独的 **checkWalls** 函数里，我们还需要实现碰撞检测的函数。

checkCollision 一开始比较简单，它只是一个基于距离的碰撞检测函数。

```
function checkCollision (ball0, ball1) {
  var dx = ball1.x - ball0.x,
      dy = ball1.y - ball0.y,
      dist = Math.sqrt(dx * dx + dy * dy);

  if (dist < ball0.radius + ball1.radius) {
     //collision handling code here
  }
}
```

碰撞处理代码首先需要计算两球之间的角度。回忆一下第 3 章中学到的三角学知识，可以用 **Math.atan3(dy,dx)**来计算角度。计算出正弦和余弦并保存下来，因为它们将会反复用到。

```
//calculate angle, sine, and cosine
var angle = Math.atan2(dy, dx),
    sin = Math.sin(angle),
    cos = Math.cos(angle);
```

然后，要对两个小球的速度和位置进行坐标旋转。旋转后的位置是 **x0**、**x0**、**x1** 和 **y1**，旋

转后的速度为 **vx0**、**vy0**、**vx1** 和 **vy1**。

因为使用 **ball0** 作为“中心点”，所以它的坐标为(0，0)。因为坐标旋转后它也不变，所以可以这么写：

```
//rotate ball0's position
var x0 = 0,
    y0 = 0;
```

接下来，**ball1** 的位置与 **ball0** 的位置有关，这与刚才计算出的距离 **dx** 和 **dy** 相对应，所以可以旋转它们来得到 **ball1** 旋转后的位置：

```
//rotate ball1's position
var x1 = dx * cos + dy * sin,
    y1 = dy * cos - dx * sin;
```

现在旋转所有小球的速度。你可以看到一个模式正在形成：

```
//rotate ball0's velocity
var vx0 = ball0.vx * cos + ball0.vy * sin,
    vy0 = ball0.vy * cos - ball0.vx * sin;

//rotate ball1's velocity
var vx1 = ball1.vx * cos + ball1.vy * sin,
    vy1 = ball1.vy * cos - ball1.vx * sin;
```

所有的旋转代码都到位了：

```
function checkCollision (ball0, ball1) {
  var  dx = ball1.x - ball0.x,
       dy = ball1.y - ball0.y,
       dist = Math.sqrt(dx * dx + dy * dy);

  if (dist < ball0.radius + ball1.radius) {
     //calculate angle, sine, and cosine
     var angle = Math.atan2(dy, dx),
          sin = Math.sin(angle),
          cos = Math.cos(angle),

          //rotate ball0's position
          x0 = 0,
          y0 = 0,
          //rotate ball1's position
          x1 = dx * cos + dy * sin,
          y1 = dy * cos - dx * sin,

          //rotate ball0's velocity
          vx0 = ball0.vx * cos + ball0.vy * sin,
          vy0 = ball0.vy * cos - ball0.vx * sin,

          //rotate ball1's velocity
          vx1 = ball1.vx * cos + ball1.vy * sin,
          vy1 = ball1.vy * cos - ball1.vx * sin;
  }
}
```

现在使用 **vx0**、**ball0.mass** 和 **vx1**、**ball1.mass** 执行一维的碰撞反应。在前面的一维例子中，有这样的代码：

```
var vxTotal = ball0.vx - ball1.vx;
ball0.vx = ((ball0.mass - ball1.mass) * ball0.vx + 2 * ball1.mass * ball1.vx) /
           (ball0.mass + ball1.mass);
ball1.vx = vxTotal + ball0.vx;
```

可以像下面这样重写这段代码：

```
var vxTotal = vx0 - vx1;
vx0 = ((ball0.mass - ball1.mass) * vx0 + 2 * ball1.mass * vx1) /
      (ball0.mass + ball1.mass);
vx1 = vxTotal + vx0;
```

所有要做的就是用旋转后的 **vx0** 和 **vx1** 来替换 **ball0.vx** 和 **ball1.vx**。把代码插入函数定义中：

```
function checkCollision (ball0, ball1) {
  var dx = ball1.x - ball0.x,
      dy = ball1.y - ball0.y,
      dist = Math.sqrt(dx * dx + dy * dy);

  if (dist < ball0.radius + ball1.radius) {
     //calculate angle, sine, and cosine
     var angle = Math.atan2(dy, dx),
         sin = Math.sin(angle),
         cos = Math.cos(angle),

         //rotate ball0's position
         x0 = 0,
         y0 = 0,

         //rotate ball1's position
         x1 = dx * cos + dy * sin,
         y1 = dy * cos - dx * sin,

         //rotate ball0's velocity
         vx0 = ball0.vx * cos + ball0.vy * sin,
         vy0 = ball0.vy * cos - ball0.vx * sin,
         //rotate ball1's velocity
         vx1 = ball1.vx * cos + ball1.vy * sin,
         vy1 = ball1.vy * cos - ball1.vx * sin,

         //collision reaction
         vxTotal = vx0 - vx1;
     vx0 = ((ball0.mass - ball1.mass) * vx0 + 2 * ball1.mass * vx1) /
           (ball0.mass + ball1.mass);
     vx1 = vxTotal + vx0;
     x0 += vx0;
     x1 += vx1;
  }
}
```

和一维例子一样，这段代码也把新的 x 速度加在 x 位置上来反弹小球。

既然得到了碰撞后的位置与速度，就把一切旋转归位。首先计算旋转归位后的位置：

```
//rotate positions back
var x0Final = x0 * cos - y0 * sin,
    y0Final = y0 * cos + x0 * sin,
    x1Final = x1 * cos - y1 * sin,
    y1Final = y1 * cos + x1 * sin;
```

别忘了调换旋转方程式中“+”和“-”的位置，因为现在是往反方向旋转。这些“最终”位置并不是真正的最终，它们相对于系统中心点，也就是 **ball0** 的原始位置。你需要把它们与 **ball0** 的位置相加来得到真正的坐标位置。首先计算 **ball1**，以便它使用 **ball0** 的原始位置，而不是更新后的位置：

```
//adjust positions to actual screen positions
ball1.x = ball0.x + x1Final;
ball1.y = ball0.y + y1Final;
ball0.x = ball0.x + x0Final;
ball0.y = ball0.y + y0Final;
```

最后，把速度旋转归位，直接改变小球的 **vx** 和 **vy** 属性：

```
//rotate velocities back
ball0.vx = vx0 * cos - vy0 * sin;
ball0.vy = vy0 * cos + vx0 * sin;
ball1.vx = vx1 * cos - vy1 * sin;
ball1.vy = vy1 * cos + vx1 * sin;
```

最终来看一下整个完整的函数：

```
function checkCollision (ball0, ball1) {
  var dx = ball1.x - ball0.x,
      dy = ball1.y - ball0.y,
      dist = Math.sqrt(dx * dx + dy * dy);
  if (dist < ball0.radius + ball1.radius) {
      //calculate angle, sine, and cosine
      var angle = Math.atan2(dy, dx),
          sin = Math.sin(angle),
          cos = Math.cos(angle),

          //rotate ball0's position
          x0 = 0,
          y0 = 0,

          //rotate ball1's position
          x1 = dx * cos + dy * sin,
          y1 = dy * cos - dx * sin,

          //rotate ball0's velocity
          vx0 = ball0.vx * cos + ball0.vy * sin,
          vy0 = ball0.vy * cos - ball0.vx * sin,

          //rotate ball1's velocity
          vx1 = ball1.vx * cos + ball1.vy * sin,
          vy1 = ball1.vy * cos - ball1.vx * sin,

          //collision reaction
          vxTotal = vx0 - vx1;
      vx0 = ((ball0.mass - ball1.mass) * vx0 + 2 * ball1.mass * vx1) /
            (ball0.mass + ball1.mass);
      vx1 = vxTotal + vx0;
      x0 += vx0;
      x1 += vx1;

      //rotate positions back
```

```
    var x0Final = x0 * cos - y0 * sin,
        y0Final = y0 * cos + x0 * sin,
        x1Final = x1 * cos - y1 * sin,
        y1Final = y1 * cos + x1 * sin;

    //adjust positions to actual screen positions
    ball1.x = ball0.x + x1Final;
    ball1.y = ball0.y + y1Final;
    ball0.x = ball0.x + x0Final;
    ball0.y = ball0.y + y0Final;

    //rotate velocities back
    ball0.vx = vx0 * cos - vy0 * sin;
    ball0.vy = vy0 * cos + vx0 * sin;
    ball1.vx = vx1 * cos - vy1 * sin;
    ball1.vy = vy1 * cos + vx1 * sin;
  }
}
```

试验一下这个例子，改变小球的大小、初始速度和质量等，确保它正常工作。

checkCollision 函数很复杂。但是如果你阅读注释，就会发现它被切分（相对地）成了简单的小块。我们做了一定的优化，通过代码重用消除了一些重复。这样可以更容易地维护它。可以在 **04-billiard-4.html** 中看到最终结果：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Billiard 4</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball0 = new Ball(80),
        ball1 = new Ball(40),
        bounce = -1.0;

    ball0.mass = 2;
    ball0.x = canvas.width - 200;
    ball0.y = canvas.height - 200;
    ball0.vx = Math.random() * 10 - 5;
    ball0.vy = Math.random() * 10 - 5;

    ball1.mass = 1;
    ball1.x = 100;
    ball1.y = 100;
    ball1.vx = Math.random() * 10 - 5;
    ball1.vy = Math.random() * 10 - 5;
```

```
function rotate (x, y, sin, cos, reverse) {
  return {
    x: (reverse) ? (x * cos + y * sin) : (x * cos - y * sin),
    y: (reverse) ? (y * cos - x * sin) : (y * cos + x * sin)
  };
}
function checkCollision (ball0, ball1) {
  var dx = ball1.x - ball0.x,
      dy = ball1.y - ball0.y,
      dist = Math.sqrt(dx * dx + dy * dy);

  //collision handling code here
  if (dist < ball0.radius + ball1.radius) {
      //calculate angle, sine, and cosine
      var angle = Math.atan2(dy, dx),
          sin = Math.sin(angle),
          cos = Math.cos(angle),
          //rotate ball0's position
          pos0 = {x: 0, y: 0}, //point

          //rotate ball1's position
          pos1 = rotate(dx, dy, sin, cos, true),

          //rotate ball0's velocity
          vel0 = rotate(ball0.vx, ball0.vy, sin, cos, true),

          //rotate ball1's velocity
          vel1 = rotate(ball1.vx, ball1.vy, sin, cos, true),

          //collision reaction
          vxTotal = vel0.x - vel1.x;
    vel0.x = ((ball0.mass - ball1.mass) * vel0.x + 2 * ball1.mass * vel1.x) /
             (ball0.mass + ball1.mass);
    vel1.x = vxTotal + vel0.x;

    //update position
    pos0.x += vel0.x;
    pos1.x += vel1.x;

    //rotate positions back
    var pos0F = rotate(pos0.x, pos0.y, sin, cos, false),
        pos1F = rotate(pos1.x, pos1.y, sin, cos, false);

    //adjust positions to actual screen positions
    ball1.x = ball0.x + pos1F.x;
    ball1.y = ball0.y + pos1F.y;
    ball0.x = ball0.x + pos0F.x;
    ball0.y = ball0.y + pos0F.y;

    //rotate velocities back
    var vel0F = rotate(vel0.x, vel0.y, sin, cos, false),
        vel1F = rotate(vel1.x, vel1.y, sin, cos, false);
    ball0.vx = vel0F.x;
    ball0.vy = vel0F.y;
```

```
        ball1.vx = vel1F.x;
        ball1.vy = vel1F.y;
      }
    }

    function checkWalls (ball) {
      if (ball.x + ball.radius > canvas.width) {
         ball.x = canvas.width - ball.radius;
         ball.vx *= bounce;
      } else if (ball.x - ball.radius < 0) {
         ball.x = ball.radius;
         ball.vx *= bounce;
      }
      if (ball.y + ball.radius > canvas.height) {
         ball.y = canvas.height - ball.radius;
         ball.vy *= bounce;
      } else if (ball.y - ball.radius < 0) {
         ball.y = ball.radius;
         ball.vy *= bounce;
      }
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      ball0.x += ball0.vx;
      ball0.y += ball0.vy;
      ball1.x += ball1.vx;
      ball1.y += ball1.vy;

      checkCollision(ball0, ball1);
      checkWalls(ball0);
      checkWalls(ball1);

      ball0.draw(context);
      ball1.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

在本例中，创建了一个 **rotate** 函数，它接收一些参数，返回一个对象，表示旋转后的点，包括了 **x** 和 **y** 属性。这个版本在学习概念阶段并不易于阅读，但是它尽量消除了重复代码。

加入更多物体

让两个小球碰撞并做出反应并不是容易的事，不过你已经做到了。现在让我们加入更多的碰撞对象——比如，8 个。这看起来好像要复杂四倍，其实并没有。现在的函数每次检测两个小球，不过这正是你需要的。加入更多物体，让它们运动起来，然后检测每一个小球是否与其

他小球碰撞。你已经在第 9 章的碰撞检测例子中见过了，需要做的就是把 **checkCollision** 函数插入通常完成碰撞检测的地方。

在示例 **05-multi-billiard-1.html** 中，一开始把 8 个小球放入数组中。**checkCollision** 和 **checkWalls** 函数没有完整列出，但是你可以在上一个例子中找到完全相同的定义。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Multi Billiard 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
<body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        balls = [],
        numBalls = 8,
        bounce = -1.0;

    for (var radius, ball, i = 0; i < numBalls; i++) {
      radius = Math.random() * 20 + 15;
      ball = new Ball(radius);
      ball.mass = radius;
      ball.x = i * 100;
      ball.y = i * 50;
      ball.vx = Math.random() * 10 - 5;
      ball.vy = Math.random() * 10 - 5;
      balls.push(ball);
    }

    function rotate (x, y, sin, cos, reverse) {
      return {
        x: (reverse) ? (x * cos + y * sin) : (x * cos - y * sin),
        y: (reverse) ? (y * cos - x * sin) : (y * cos + x * sin)
      };
    }

    function checkCollision (ball0, ball1) {
      //not listed, same as previous example...
    }

    function checkWalls (ball) {
      //not listed, same as previous example...
    }

    function move (ball) {
      ball.x += ball.vx;
```

```
      ball.y += ball.vy;
      checkWalls(ball);
    }

    function draw (ball) {
      ball.draw(context);
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      balls.forEach(move);
      for (var ballA, i = 0, len = numBalls - 1; i < len; i++) {
         ballA = balls[i];
         for (var ballB, j = i + 1; j < numBalls; j++) {
           ballB = balls[j];
           checkCollision(ballA, ballB);
         }
      }
      balls.forEach(draw);
    }());
  };
  </script>
 </body>
</html>
```

这些小球的初始位置有一定间隔，这是为了让它们不会一开始就发生碰撞，以避免它们有可能卡在一起。

drawFrame 函数简单得惊人。它遍历了三次 balls：一次是基本的运动，一次是碰撞检测，当然，另外一次是绘图。第一个 **forEach** 遍历 balls，每次把一个 ball 传入 **move** 函数，**move** 函数更新小球的位置，移动小球并处理墙面反弹。然后，那个双重 **for** 循环执行碰撞检测。这里，你每次得到两个小球的引用，把它们传入 **checkCollision** 函数。因为 **checkWalls** 和 **checkCollision** 函数与前面相同，所以只要把它们从前一个例子添加到该文件中即可。

如果想要加入更多小球，只需要改变 **numBalls** 变量，并且确保它们在动画开始的时候不要接触。

解决一个潜在问题

本章提过一个关于设置的警告：两个物体有可能卡在一起。这通常发生在有多个物体碰撞的拥挤的环境中，当它们高速运动时情况会更糟。你可以偶然看到两三个小球在角落里发生这种现象。

如果屏幕上有三个小球——**ball0**、**ball1** 和 **ball2**，它们恰好离得很近。下面是要发生的事情。

- 程序依照三个小球的速度移动它们。
- 程序检测 **ball0** 和 **ball1**，**ball0** 和 **ball2**，发现它们并没有碰撞。
- 程序检测 **ball1** 和 **ball2**。因为它们发生了碰撞，所以它们的速度和位置都要重新计算，然后弹开。这恰好不小心让 **ball1** 和 **ball0** 接触上了。然而，这一组已经进行过检测了，所以就忽略它。

- 在下一轮循环中，程序依然按照它们的速度移动小球。这样有可能使得 **ball0** 和 **ball1** 更为靠近。
- 现在程序发现 **ball0** 和 **ball1** 碰撞了。它会重新计算两个小球的速度和位置，将它们分开。但是，因为它们已经发生了接触，所以这可能并不能真正地分开它们，它们就卡在了一起。

再次重复，这种情况最容易发生在空间小物体多并且移动速度高的情况下。它也会发生在物体一开始就接触的情况下。你可能在测试本例的时候已经看到过了，因为这个问题随时可能出现，所以很有必要知道问题出在哪里。确切的位置是在 **checkCollision** 函数里，更确切地说，是下面两行代码：

```
//update position
pos0.x += vel0.x;
pos1.x += vel1.x;
```

这里假设碰撞只是由两个小球自己的速度引起的，然后把它们新的速度加回去以分开它们。大多数情况下，这是对的。但是在我们刚才说的那个场景是例外。如果你遇到了这个问题，就需要在移动之前更加严谨地确保两个物体是分离的。试试下面的方法：

```
//update position
var absV = Math.abs(vel0.x) + Math.abs(vel1.x),
    overlap = (ball0.radius + ball1.radius) - Math.abs(pos0.x - pos1.x);
pos0.x += vel0.x / absV * overlap;
pos1.x += vel1.x / absV * overlap;
```

这可能不是数学中最精确的方法，但是它看起来工作得很好。它首先确定绝对速度（两个物体速度的绝对值之和）。例如，如果一个物体的速度是-5，另一个是 10，速度的绝对值是 5 和 10，之和是 5+10，也就是 15。

接下来，它确定两个小球的重叠量，这通过总半径减去距离来获得。

然后，根据每个小球速度与绝对速度的比例，把它们移开重叠量的一部分距离。最后的结果是两个小球刚刚接触，但没有重叠。这比前一个版本要复杂一点，但是它同时也消除了 bug。

实际上，在下一个版本（**06-multi-billiard-2.html**）中，创建了 15 个更大一些的小球，把它们随机放置在 canvas 上。在最初的几帧内还是有问题，不过，因为这些新的代码，最终结果会正常。下面是完整的代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Multi Billiard 2</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        balls = [],
        numBalls = 15,
```

```
    bounce = -1.0;

for (var radius, ball, i = 0; i < numBalls; i++) {
  radius = Math.random() * 20 + 15;
  ball = new Ball(radius, Math.random() * 0xffffff);
  ball.mass = radius;
  ball.x = Math.random() * canvas.width;
  ball.y = Math.random() * canvas.height;
  ball.vx = Math.random() * 10 - 5;
  ball.vy = Math.random() * 10 - 5;
  balls.push(ball);
}

function rotate (x, y, sin, cos, reverse) {
  return {
    x: (reverse) ? (x * cos + y * sin) : (x * cos - y * sin),
    y: (reverse) ? (y * cos - x * sin) : (y * cos + x * sin)
  };
}

function checkCollision (ball0, ball1) {
  var dx = ball1.x - ball0.x,
      dy = ball1.y - ball0.y,
      dist = Math.sqrt(dx * dx + dy * dy);

  //collision handling code here
  if (dist < ball0.radius + ball1.radius) {
     //calculate angle, sine, and cosine
     var angle = Math.atan2(dy, dx),
          sin = Math.sin(angle),
          cos = Math.cos(angle),

          //rotate ball0's position
          pos0 = {x: 0, y: 0}, //point

          //rotate ball1's position
          pos1 = rotate(dx, dy, sin, cos, true),

          //rotate ball0's velocity
          vel0 = rotate(ball0.vx, ball0.vy, sin, cos, true),

          //rotate ball1's velocity
          vel1 = rotate(ball1.vx, ball1.vy, sin, cos, true),

          //collision reaction
          vxTotal = vel0.x - vel1.x;
     vel0.x = ((ball0.mass - ball1.mass) * vel0.x + 2 * ball1.mass * vel1.x) /
               (ball0.mass + ball1.mass);
     vel1.x = vxTotal + vel0.x;

     //update position - to avoid objects becoming stuck together
     var absV = Math.abs(vel0.x) + Math.abs(vel1.x),
          overlap = (ball0.radius + ball1.radius) - Math.abs(pos0.x - pos1.x);
     pos0.x += vel0.x / absV * overlap;
```

```
      pos1.x += vel1.x / absV * overlap;

      //rotate positions back
      var pos0F = rotate(pos0.x, pos0.y, sin, cos, false),
          pos1F = rotate(pos1.x, pos1.y, sin, cos, false);

      //adjust positions to actual screen positions
      ball1.x = ball0.x + pos1F.x;
      ball1.y = ball0.y + pos1F.y;
      ball0.x = ball0.x + pos0F.x;
      ball0.y = ball0.y + pos0F.y;

      //rotate velocities back
      var vel0F = rotate(vel0.x, vel0.y, sin, cos, false),
          vel1F = rotate(vel1.x, vel1.y, sin, cos, false);
      ball0.vx = vel0F.x;
      ball0.vy = vel0F.y;
      ball1.vx = vel1F.x;
      ball1.vy = vel1F.y;
    }
  }

  function checkWalls (ball) {
    if (ball.x + ball.radius > canvas.width) {
        ball.x = canvas.width - ball.radius;
        ball.vx *= bounce;
    } else if (ball.x - ball.radius < 0) {
        ball.x = ball.radius;
        ball.vx *= bounce;
    }
    if (ball.y + ball.radius > canvas.height) {
       ball.y = canvas.height - ball.radius;
       ball.vy *= bounce;
    } else if (ball.y - ball.radius < 0) {
       ball.y = ball.radius;
       ball.vy *= bounce;
    }
  }

  function move (ball) {
    ball.x += ball.vx;
    ball.y += ball.vy;
    checkWalls(ball);
  }

  function draw (ball) {
    ball.draw(context);
  }

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    balls.forEach(move);
    for (var ballA, i = 0, len = numBalls - 1; i < len; i++) {
```

```
        ballA = balls[i];
        for (var ballB, j = i + 1; j < numBalls; j++) {
          ballB = balls[j];
          checkCollision(ballA, ballB);
        }
      }
      balls.forEach(draw);
    }());
  };
  </script>
 </body>
</html>
```

当然，你也可以研究你自己的解决方案来解决这个问题。如果你找到了更简单、更高效和更精确的方案，请分享给大家！

11.4 本章中的重要公式

本章中的一个重要公式是动量守恒。

11.4.1 动量守恒的数学表示

$$v0Final = \frac{(m0 - m1) \times v0 + 2 \times m1 \times v1}{m0 + m1}$$

$$v1Final = \frac{(m1 - m0) \times v1 + 2 \times m0 \times v0}{m0 + m1}$$

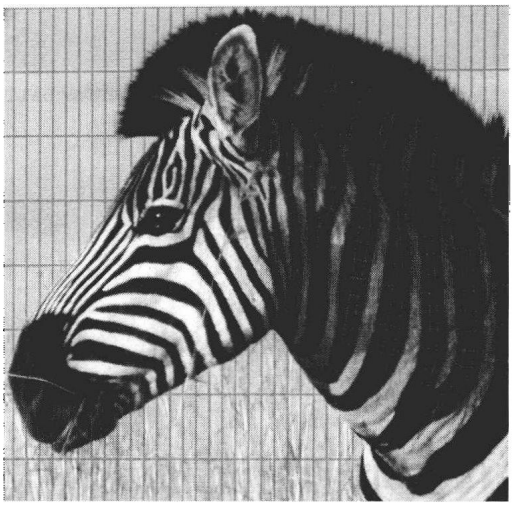
11.4.2 动量守恒的 JavaScript 代码

```
var vxTotal = vx0 - vx1;
vx0 = ((ball0.mass - ball1.mass) * vx0 + 2 * ball1.mass * vx1) / (ball0.mass + ball1.mass);
vx1 = vxTotal + vx0;
```

11.5 小结

恭喜！你掌握了本书中最需要用到数学知识的部分，现在你已经掌握了如何精确地处理碰撞反应。为了保持示例简单易懂，我们忽略了一个概念：摩擦力。你可以试着把它加入到系统中，因为你现在肯定有足够的知识来做这件事。一定要读第 19 章，在那里你将会看到处理两个相同质量物体的一些技巧。

在第 12 章中，我们将缓和一下，来学习粒子引力，不过在示例中加入一些撞球物理会非常合适。



第 12 章　粒子与万有引力

本章涵盖以下内容：

- 粒子
- 万有引力
- 弹力

随着本书的进展，每章都加入了一个新的概念，以提升动画的质量。一开始，你只是有一些可以运动的物体。接下来让它们与环境交互，然后与用户交互，最后是物体间通过碰撞交互。本章将详述物体间交互的途径，特别是相距一定距离的物体。具体来说，我们将介绍粒子、万有引力（与前面介绍过的重力稍有不同）、弹力（再一次！），以及让它们相互作用。

12.1　粒子

在本章中，粒子仅仅代表一个单元，通常会伴随着几个或更多相同的单元。比如，一个粒子可以是一个弹球、一个气球，或者一个星球。

粒子通常都有一些共有的行为，而且可以有它们自己的特定行为。在本书的示例中，你已经见过当我们使用 **Ball** 类的一些实例时，每一个对象拥有自己的属性：速度、质量、大小、颜色等，但是所有的小球都以相同的规则在运动。

本章继续使用 **Ball** 类，因为它已经有了你需要用到的一些功能。粒子对象只保存属性，其余的脚本处理每个粒子的移动。另一个策略是在粒子类中定义一个 **move** 方法，这样每个对象要自己负责移动自己，以及处理交互。两种方法都可行，但是在本书的例子中，为了简单起见，

把动画代码放在粒子对象之外。

每个示例的基础设置代码都相同，这是因为大多数变化点都在粒子间的相互作用和引力上。下面是我们创建的一份基础文件，它创建几个粒子，并把它们随机放置在 canvas 元素上。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Particles</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        particles = [],
        numParticles = 30;

    for (var particle, i = 0; i < numParticles; i++) {
       particle = new Ball(5);
       particle.x = Math.random() * canvas.width;
       particle.y = Math.random() * canvas.height;
       particle.mass = 1;
       particles.push(particle);
    }

    function draw (particle) {
      particle.draw(context);
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      particles.forEach(draw);
    }());
  };
  </script>
 </body>
</html>
```

这里，你创建了 30 个半径（radius）为 5，质量（mass）为 1 的粒子。稍后，我们将改变这些值来得到不同的效果。你也可以把这些粒子初始化为随机速度和随机大小。

这个文件很重要，因为本章其余的例子都以此作为基础设置，在它的基础上添加代码。但是在我们开始之前，你需要先学习一些背景理论。

12.2 万有引力

我们要讨论的第一种粒子间的吸引就是万有引力。第 5 章已经介绍过万有引力（重力），

但是那是从微观角度来看的重力。

站在地球上，重力的描述很简单：它把物体向下拉。实际上，它以特定的速率把物体向下拉。它施加在物体上的加速度大约是每秒 32 英尺。表达加速度的一个方法就是在特定时间内速度增加了多少。引力使物体的速度每秒增加 32 英尺。在最高的山峰上或者最深的峡谷里来看，32 这个数字的变化微小到不足以引起你的注意（除非用专门的测量工具）。

12.2.1 万有引力

当你往后退时，离一个星球或者很大的物体越远，受到的引力就越小。这对于地球和其他行星来说是个很好的现象，避免了被吸进太阳里捣得粉碎。从遥远的、宏观的视角看太阳系，你可以把行星看作粒子，它们间的距离会影响万有引力。

距离与万有引力的关系很容易描述：万有引力与距离的平方成反比。也许需要解释一下，引力与质量关系紧密，一个物体的质量越大，它对其他物体的引力就越大，它受到其他物体的引力同样也越大。这里还有一个叫做万有引力常数（缩写为 G）的概念。完整的引力方程式如下：

$$force = G \times m_1 \times m_2 / distance^2$$

这意味着，两个物体间的引力等于万有引力常量乘以两个物体的质量，然后除以它们之间距离的平方。听起来很简单，你只要知道万有引力常量是多少就行了。下面是它的官方定义：

$$G = 6.674 \times 10^{-11} \times m^3 \times kg^{-1} \times S^{-2}$$

它并不是那么简单。但幸运的是，我们可以把 G 从公式里移除掉，动画代码将会看起来像这样：

$$force = m_1 \times m_2 / distance^2$$

这看起来有偷懒的嫌疑，确实是，但是移除的是我们不想注意到的东西，而且避免了一堆的计算。想想万有引力常数 G，它用来标准化方程式以得到引力在特定环境中的合适计量。如果你想要对宇宙建模，用牛顿作为计量单位，就必须把 G 设置到前面的方程式中。但是在动画中，我们可以在我们自己的“小宇宙”中工作，就把 G 设置为 1，并去除它。实际上，粒子物理学家在他们自己的方程式里也是这么做的。如果你在为美国宇航局（NASA）做一个基于 canvas 的卫星导航系统，你就需要把 G 留下来。但是如果你在编写下一个伟大的星球大战游戏，就可以把它去掉了。

既然你有了一个新的公式，让我们把它写入代码吧。在 **drawFrame** 动画循环中，遍历所有的粒子，把每一个都传入 **move** 函数。在 **move** 中，调用另一个函数 **gravitate**，这样你就能把处理引力的代码分离出来：

```
function move (partA, i) {
  partA.x += partA.vx;
  partA.y += partA.vy;

  for (var partB, j = i + 1; j < numParticles; j++) {
    partB = particles[j];
    gravitate(partA, partB);
  }
}

(function drawFrame () {
```

```
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  particles.forEach(move);
  particles.forEach(draw);
}());
```

在 **move** 函数中，使用 **for** 循环遍历其余的粒子。当获得了 **partA** 和 **partB** 后，把它们传入 **gravitate** 函数，如下：

```
function gravitate (partA, partB) {
  var dx = partB.x - partA.x,
      dy = partB.y - partA.y,
      distSQ = dx * dx + dy * dy,
      dist = Math.sqrt(distSQ),
      force = partA.mass * partB.mass / distSQ,
      ax = force * dx / dist,
      ay = force * dy / dist;

  partA.vx += ax / partA.mass;
  partA.vy += ay / partA.mass;
  partB.vx -= ax / partB.mass;
  partB.vy -= ay / partB.mass;
}
```

在这个函数中，首先计算两个粒子间的距离（**dx** 和 **dy**），以及总距离。记住，这个引力公式 **force** = $m_1 \times m_2$ / ***distance***^{2} 包含距离的平方。通常，我们用 **dist = Math.sqrt(dx * dx + dy * dy)** 来计算距离，但是接下来我们又要计算距离的平方，也就是计算一个平方根的平方，这显然是重复工作。如果我们用变量 **distSQ** 来保存 **dx * dx + dy * dy**，就避免了不必要的计算。

接下来，我们用两个粒子质量的乘积除以距离的平方来得到总引力。然后求出 x 轴和 y 轴上的总加速度。然后，我们再次使用第 9 章结尾时介绍的捷径：用 **dx/dist** 代替 **Math.cos(angle)**，用 **dy/dist** 代替 **Math.sin(angle)**。这样一来，我们就不需要首先用 **Math.atan2(dy，dx)**来求出角度。

现在，请注意，我们是在谈总引力和总加速度。这是两个粒子间的组合引力。需要根据两个粒子的质量把它分摊到两个粒子上。想一下地球和太阳，它们之间有特定的引力，这是由它们的质量乘积除以它们距离的平方得来的。所以，它们在总引力的作用下相互吸引，地球被拉向太阳，太阳也被拉向地球。显然，地球获得的加速度更大，因为它的质量比太阳小得多。所以，用总加速度除以对象的质量，就能得到系统中每个对象的单独加速度。这样，就有了最后 4 行公式。注意，**partA** 加上了加速度，**partB** 减去了加速度，这仅是由于计算 **dx** 和 **dy** 时相减的顺序。

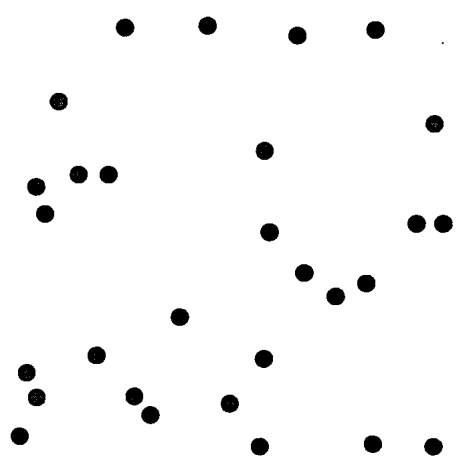


图 12-1 一堆粒子

该示例最终的代码可以在 **01-gravity.html** 中找到。测试一下，可以得到类似图 12-1 的结果。这些粒子一开始静止，然后相互吸引。偶尔两个粒子会相互绕圈，但

是大多数情况下，这些粒子相互接近，然后向相反的方向飞出。

这样碰撞后高速飞出是代码中的 bug 吗？并不是，这正是期望的效果。这个行为叫做弹弓效应（slingshot effect），NASA 就是使用这种效应把探测器发射到外太空去。随着一个物体与一个星球越来越近，它的加速度越来越大，速度会越来越高。如果你瞄准得恰到好处，物体就会掠过星球，以足够大的速度摆脱星球的引力，进入太空。在程序中，当两个物体距离很小的时候——几乎是零距离，它们之间的引力变得非常大，几乎是无限大。这样在数学上是对的。不过，从模拟的角度看，这并不真实。应该发生的结果是：在两个物体足够接近时，我们来控制碰撞。如果你把空间探测器瞄准一个星球，它就不能以无限大的速度接近，这样只能撞出一个火山口。

12.2.2 碰撞检测及反应

这些粒子需要很好的碰撞检测和碰撞后的反应，怎么做要取决于你。你可以让它们爆炸或消失，或者可以让一个粒子消失，然后把它的质量加在另一个上，就像是两个融合了。在下一个例子里，我们只是让它们弹开。

我们可以重用以前章节中漂亮的碰撞检测和碰撞反应代码，函数名叫做 checkCollision。把它插入 move 函数中，像这样：

```
function move (partA, i) {
  partA.x += partA.vx;
  partA.y += partA.vy;

  for (var partB, j = i + 1; j < numParticles; j++) {
    partB = particles[j];
    checkCollision(partA, partB);
    gravitate(partA, partB);
  }
}
```

只有粗体的那行代码发生了变化。当然，需要把 checkCollision 和 rotate 函数拷贝到本文件内，代码可以在文件 02-gravity-bounce.html 中找到。

现在粒子互相吸引，但是碰撞后会弹开。改变粒子的质量，观察引力如何变化。你甚至可以做些奇异的事情，比如，给粒子一个负质量，观察它们互斥。

在文件 03-gravity-random.html 中，除了下面几行初始化粒子的代码，其他代码都保持不变：

```
for (var size, particle, i = 0; i < numParticles; i++) {
  size = Math.random() * 20 + 5;
  particle = new Ball(size);
  particle.x = Math.random() * canvas.width;
  particle.y = Math.random() * canvas.height;
  particle.mass = size;
  particles.push(particle);
}
```

每个粒子初始化为不同大小以及和大小相匹配的质量，如图 12-2 所示，这变得越来越有趣了。

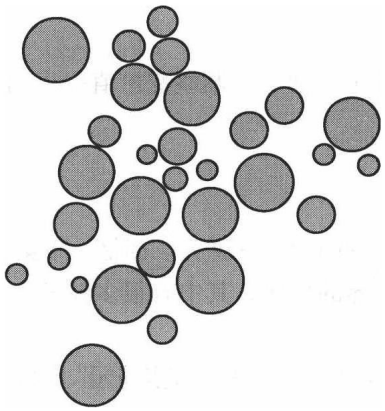


图 12-2 大碰撞

12.2.3 轨道运动

为了演示轨道运动，我们建立一个简单的行星系，其中有一个太阳和一个行星。设置太阳的质量为 10 000，行星的质量为 1。让行星距离太阳一段距离，然后给它一个沿太阳切线方向的初速度，如图 12-3 所示。

如果设置了合适的质量、距离和速度，你就能让行星进入轨道（经过一些试验和错误）。你可以在文件 04-orbit.html 中找到一个可行的例子。下面是初始化的代码，我们创建了两个粒子，大的是太阳，小的是围绕太阳旋转的行星，其余代码都不变：

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    particles = [],
    numParticles = 2,
    sun = new Ball(100, "#ffff00"),
    planet = new Ball(10, "#00ff00");

sun.x = canvas.width / 2;
sun.y = canvas.height / 2;
sun.mass = 10000;
particles.push(sun);

planet.x = canvas.width / 2 + 200;
planet.y = canvas.height / 2;
planet.vy = 7;
planet.mass = 1;
particles.push(planet);
```

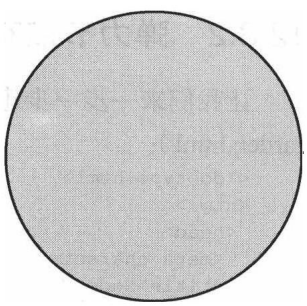


图 12-3 建立太阳系

把 planet.vy 调低一点看看它如何从轨道掉落，坠入太阳，然后反弹回去。

像是一个额外的奖励，在 drawFrame 函数中不清理 canvas 元素，就能得到轨道路径和一条直线：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);

  particles.forEach(move);
```

```
  particles.forEach(draw);
}());
```

该例子在 04-orbit-draw.html 中，你可以调整变量值来得到一些有趣的效果。

12.3 弹力

你可能想尝试另一种粒子间的引力——弹力。回忆一下第 8 章中弹簧链以及物体间弹动的例子。这里，你将看到一个有边界的程序，其中有很多粒子，它们之间互相弹动，就像前面引力的例子一样。

这个例子的灵感来源于 Jared Tarbell 创作的“节点花园”（Node Garden）（www.levitated.net），它是用 Flash 写的。这个想法是你有很多节点（粒子），每个节点都与相邻的节点有不同类型的交互。

12.3.1 万有引力 VS 弹力

如果你观察万有引力和弹力，就会发现它们相似但几乎完全相反。它们都是在两个物体上施加加速度使它们接近。但是对于万有引力，两个物体距离越大，加速度越小；对于弹力，两个物体距离越大，加速度越大。

可以用弹力代码来替换前一个例子中的万有引力代码，但是结果并不有趣。粒子最终会黏成一团——弹力不能容忍距离。

这是一个两难的处境，你既想让粒子间通过弹力相互吸引，又想让它们保持一定距离，不粘在一起。我们设置一个最小距离来解决这个问题，如果两个粒子间的距离大于这个最小距离，则忽略对方。

12.3.2 弹力节点花园

让我们来一步一步建立自己的弹力节点花园吧，这里列出的是全部代码（文件 06-node-garden.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Node Garden</title>
  <link rel="stylesheet" href="style.css">
  <style>
  #canvas {
    background-color: #000000;
  }
  </style>
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
```

```
<script src="ball.js"></script>
<script>
window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      particles = [],
      numParticles = 30,
      minDist = 100,
      springAmount = 0.001;

  for (var particle, i = 0; i < numParticles; i++) {
    particle = new Ball(5, "#ffffff");
    particle.x = Math.random() * canvas.width;
    particle.y = Math.random() * canvas.height;
    particle.vx = Math.random() * 6 - 3;
    particle.vy = Math.random() * 6 - 3;
    particles.push(particle);
  }

  function spring (partA, partB) {
    var dx = partB.x - partA.x,
        dy = partB.y - partA.y,
        dist = Math.sqrt(dx * dx + dy * dy);
    if (dist < minDist) {
        var ax = dx * springAmount,
            ay = dy * springAmount;
        partA.vx += ax;
        partA.vy += ay;
        partB.vx -= ax;
        partB.vy -= ay;
    }
  }

  function move (partA, i) {
    partA.x += partA.vx;
    partA.y += partA.vy;

    if (partA.x > canvas.width) {
        partA.x = 0;
    } else if (partA.x < 0) {
        partA.x = canvas.width;
    }
    if (partA.y > canvas.height) {
        partA.y = 0;
    } else if (partA.y < 0) {
        partA.y = canvas.height;
    }
    for (var partB, j = i + 1; j < numParticles; j++) {
        partB = particles[j];
        spring(partA, partB);
    }
  }

  function draw (particle) {
    particle.draw(context);
  }
```

```
    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      particles.forEach(move);
      particles.forEach(draw);
    }());
  };
  </script>
 </body>
</html>
```

确保 canvas 元素的背景色设置为黑色（在文件头内使用 CSS），这样就能看到白色的节点。在脚本开始的地方，声明粒子的相关变量，以及刚才提到的最小距离和弹性系数。

```
var particles = [],
    numParticles = 30,
    minDist = 100,
    springAmount = 0.001;
```

在第 8 章的弹动例子中，弹性系数设置为 0.03。这里要设置得低一些，因为有很多粒子交互，如果设置得太高，速度会累加到很高。但是，如果设置得太低，粒子就会在屏幕上缓慢运动，看不出来它们间的互动。声明变量后，接下来进行初始化：

```
for (var particle, i = 0; i < numParticles; i++) {
  particle = new Ball(5, "#ffffff");
  particle.x = Math.random() * canvas.width;
  particle.y = Math.random() * canvas.height;
  particle.vx = Math.random() * 6 - 3;
  particle.vy = Math.random() * 6 - 3;
  particles.push(particle);
}
```

这里创建了多个粒子，给予它们随机的速度和位置。这些粒子没有质量，但是在后面的小节里，我们会演示加回质量的例子。

接下来是 drawFrame 动画循环，每一帧遍历两次粒子，分别把每个粒子传入 move 和 draw 函数：

```
function move (partA, i) {
  partA.x += partA.vx;
  partA.y += partA.vy;

  if (partA.x > canvas.width) {
    partA.x = 0;
  } else if (partA.x < 0) {
    partA.x = canvas.width;
  }
  if (partA.y > canvas.height) {
      partA.y = 0;
  } else if (partA.y < 0) {
      partA.y = canvas.height;
  }
  for (var partB, j = i + 1; j < numParticles; j++) {
      partB = particles[j];
      spring(partA, partB);
  }
}
```

这与前面的屏幕环绕的例子相似，只不过这里不使用 gravitate，而使用 spring 函数：

```
function spring (partA, partB) {
  var dx = partB.x - partA.x,
      dy = partB.y - partA.y,
      dist = Math.sqrt(dx * dx + dy * dy);

  if (dist < minDist) {
    var ax = dx * springAmount,
        ay = dy * springAmount;

    partA.vx += ax;
    partA.vy += ay;
    partB.vx -= ax;
    partB.vy -= ay;
  }
}
```

这里，计算两个粒子间的距离。如果它不小于 minDist，则继续。但是如果它小于 minDist，则根据 springAmount 来计算每条轴上的加速度。把加速度累加在 partA 的速度上，从 partB 的速度中减去，这样就使得两个粒子接近。

图 12-4 活动的节点

运行结果如图 12-4 所示。粒子都聚成一团，就像一群苍蝇围着垃圾堆嗡嗡叫。但是这些团也会移动，散开，与其他团结合。这是个有趣的自然行为。改变 minDist 和 springAmount 的值，看看会发生什么。

12.3.3 相连的节点

尽管很明显节点与其他节点有交互，但是哪两个节点正在交互并不明显。然而，我们可以在两个节点间连线，来实现可视化。这很容易做到——只需要在 spring 函数中加入画线代码。如果两个节点正在交互，则画线把它们相连：

```
function spring (partA, partB) {
  var dx = partB.x - partA.x,
      dy = partB.y - partA.y,
      dist = Math.sqrt(dx * dx + dy * dy);

  if (dist < minDist) {
    context.strokeStyle = "#ffffff";
    context.beginPath();
    context.moveTo(partA.x, partA.y);
    context.lineTo(partB.x, partB.y);
    context.stroke();

    var ax = dx * springAmount,
        ay = dy * springAmount;
    partA.vx += ax;
    partA.vy += ay;
    partB.vx -= ax;
    partB.vy -= ay;
```

```
    }
}
```

现在节点间已经有线相连了，但是当节点进入或离开其他节点的交互范围时，这些线会突然断开或连接——我们可以让这个变化有些渐变，看起来更舒服些。如果两个节点在 minDist 距离外，连线完全透明，随着它们越来越接近，让连线变得越来越亮。所以，如果计算 dist/minDist，就得到 0～1 之间的一个透明度。但是刚好相反，因为当 dist 等于 minDist 时，透明度是 1，当 dist 接近 0 时，透明度为 0。所以用 1 减去这个值，逆转该效果。

回忆一下第 4 章介绍的颜色。为了应用透明度，要把颜色值格式化为 CSS 风格的 RGBA 字符串。把颜色和透明度传入 utils.colorToRGB 函数，这个函数已经包含在 utils.js 文件中。下面是最终的画线代码：

```
var alpha = 1 - dist / minDist;
context.strokeStyle = utils.colorToRGB("#ffffff", alpha);
context.beginPath();
context.moveTo(partA.x, partA.y);
context.lineTo(partB.x, partB.y);
context.stroke();
```

如图 12-5 所示，这是个很漂亮的效果。代码见 07-node-garden-lines.html。

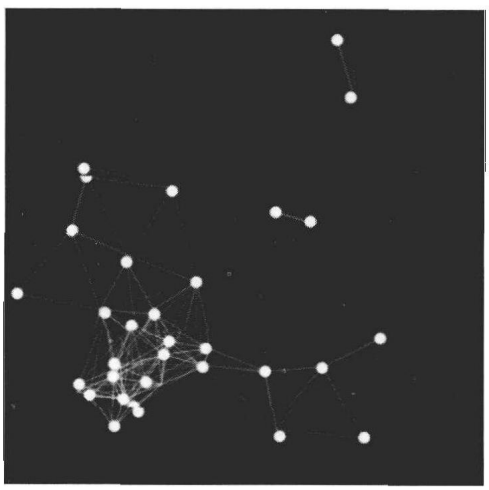


图 12-5 微小的变化，截然不同的效果

12.3.4 有质量的节点

我们可以将这些例子更进一步，为节点加入质量（文件 08-nodesmass.html）。在脚本开始时，给每个节点设置随机的大小，以及和大小相匹配的质量：

```
for (var size, particle, i = 0; i < numParticles; i++) {
  size = Math.random() * 10 + 2;
  particle = new Ball(size, "#ffffff");
  particle.x = Math.random() * canvas.width;
```

```
    particle.y = Math.random() * canvas.height;
    particle.vx = Math.random() * 6 - 3;
    particle.vy = Math.random() * 6 - 3;
    particle.mass = size;
    particles.push(particle);
  }
```

质量在 spring 函数增加粒子速度的时候会用到。用速度除以每个粒子的质量，大一些的物体得到更多的惯性：

```
function spring (partA, partB) {
  var dx = partB.x - partA.x,
      dy = partB.y - partA.y,
      dist = Math.sqrt(dx * dx + dy * dy);

  if (dist < minDist) {
    var alpha = 1 - dist / minDist;
    context.strokeStyle = utils.colorToRGB("#ffffff", alpha);
    context.beginPath();
    context.moveTo(partA.x, partA.y);
    context.lineTo(partB.x, partB.y);
    context.stroke();

    var ax = dx * springAmount,
        ay = dy * springAmount;
    partA.vx += ax / partA.mass;
    partA.vy += ay / partA.mass;
    partB.vx -= ax / partB.mass;
    partB.vy -= ay / partB.mass;
  }
}
```

因为这削减了弹力的整体效果，所以把 springAmount 减小为 0.000 5。你可以看到如图 12-6 所示的效果。

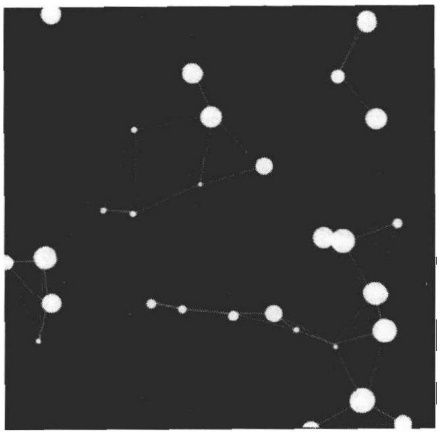


图 12-6 更进一步

除了这些看起来很炫的节点，尽可能想象一下用这种方法能建立什么游戏场景。你可能想把前面章节中的宇宙飞船加进来，让它来躲避这些节点。这是个很好的挑战。

12.4 本章中的重要公式

很显然，这里最重要的是引力公式。

12.4.1 基本引力

$$force = G \times m_1 \times m_2 / distance^2$$

12.4.2 引力公式的 JavaScript 实现

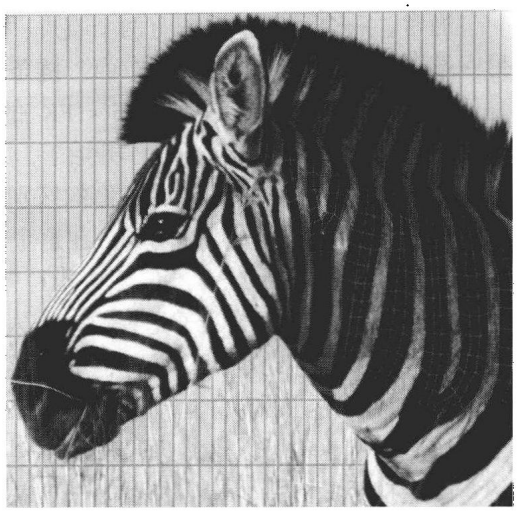
```
function gravitate (partA, partB) {
  var dx = partB.x - partA.x,
      dy = partB.y - partA.y,
      distSQ = dx * dx + dy * dy,
      dist = Math.sqrt(distSQ),
      force = partA.mass * partB.mass / distSQ,
      ax = force * dx / dist,
      ay = force * dy / dist;

  partA.vx += ax / partA.mass;
  partA.vy += ay / partA.mass;
  partB.vx -= ax / partB.mass;
  partB.vy -= ay / partB.mass;
}
```

这个函数用了 **Ball** 类的一个实例，但是你可以用任何对象，只要它能存储速度、质量和位置。

12.5 小结

本章涉及了有距离的粒子间的交互，以及如何使用万有引力和弹力实现有趣的效果。结果是，你学到了两种新的方法来创建拥有很多物体并且非常动态的动画。

第 13 章和第 14 章将讨论一些新的话题：正向动力学和反向动力学。这些技术可以用来制作灵巧的动画效果，比如，机器人的胳膊和行走效果。



第 13 章　正向运动学：让物体行走

本章涵盖以下内容：

- 介绍正向和反向运动学
- 开始编写正向运动学程序
- 自动化处理
- 让它真正地行走

前面的章节介绍了在 canvas 元素上使用 JavaScript 编写交互动画的基础知识，甚至有一些高级的“基础”。现在我们来进入另一个分支，探索一些用到运动学的有趣技术。

准确地说，**运动学**是什么呢？这个名字听起来挺吓人，但是它实际上并不是那么困难。事实上，前面的章节已经覆盖到了所有需要的概念。你只需要用正确的方法把它们组合起来就可以了。

简单地定义一下运动学：它是一个数学分支，用来处理物体的运动，但不关心质量和外力，因此它关心速度、方向。

当计算机科学、图形学、游戏领域的人说起运动学时，他们通常涉及运动学的两个特殊分支：正向运动学和反向运动学，现在让我们开始吧。

13.1　介绍正向和反向运动学

正向和反向运动学通常与多个部件组合而成的系统相关，比如，一个链条或者一个由关节组成的手臂。它们来解决整个系统如何运动，以及每个部件相对于其他部件和整个系统如何运动。

通常，一个运动学系统有两个端点：基础端和自由端。由关节组成的手臂通常一端固定，另一端可以随意伸出去拿一个东西。链条可能有一端或者两端固定，或者都不固定。

正向运动学（Forward Kinematics，FK）的动作起源于固定端，移动向自由端。**反向运动学**（Inverse Kinematics，IK）刚好相反，动作开始于，或者决定于自由端，移动向固定端（如果有的话）。

一些例子更能清楚地说明它们的区别。大多数情况下，在行走时，四肢的运动是正向运动学。大腿带动小腿，小腿带动脚。在这个例子里，脚不决定其他任何东西。它的运动取决于大腿和小腿的运动。

一个反向运动学的例子是去拉一个人的手，这时力量施加在自由端（那个人的手上），移动手的位置就能移动小臂、上臂，最终影响到整个身体。

另一个更精细的反向运动学的例子是：用胳膊去拿一个东西。和前面一样，是手驱动了这个系统。当然，你可以这么说，在这个例子里，上臂和小臂控制了手的位置。没错，但是直接的意图是把手放到一个特定位置——这就是驱动力。在这个例子里，它并不是一个物理的驱动力，只是个意图。小臂和上臂只需要简单地根据手的意图来动作。

当你看完本章和第 14 章的例子后，会更清楚它们间的区别。但是现在，要记住拖曳（Dragging）和伸手去拿（reaching）一般是反向运动学，但是一个重复周期的运动，比如，行走，往往是正向运动学，也就是本章的主题。

13.2 正向运动学编程入门

编写两种运动学的程序都包含如下一些基本元素：

- 系统的部件——节段（segment）；
- 每个节段的位置；
- 每个节段的旋转。

例子中的每个节段都是一个长椭圆形，比如，小臂和上臂，或者腿的一部分。当然末端的节段可以是其他形状，比如，手或者脚。

每个节段都有一端是轴心点，它可以围绕轴心点旋转。如果一个节段的一端有子节段，那么它的轴心点在另一端。比如，上臂的轴心点在肩膀，前臂的轴心点在肘部，手的轴心点在腕部。

当然，在很多真实的系统中，节段可能在多个方向上绕轴心点旋转。想一下你有多少种方法转动手腕。不过我们用作示例的这些系统严格来说都是二维的。

13.2.1 移动一个节段

我们先从单节段的运动入手。下面是 Segment 类，你将在后面两章中用到它，segment.js：

```
function Segment (width, height, color) {
  this.x = 0;
  this.y = 0;
  this.width = width;
```

```
  this.height = height;
  this.vx = 0;
  this.vy = 0;
  this.rotation = 0;
  this.scaleX = 1;
  this.scaleY = 1;
  this.color = (color === undefined) ? "#ffffff" : utils.parseColor(color);
  this.lineWidth = 1;
}

Segment.prototype.draw = function (context) {
  var h = this.height,
      d = this.width + h, //top-right diagonal
      cr = h / 2;        //corner radius

  context.save();
  context.translate(this.x, this.y);
  context.rotate(this.rotation);
  context.scale(this.scaleX, this.scaleY);
  context.lineWidth = this.lineWidth;
  context.fillStyle = this.color;
  context.beginPath();
  context.moveTo(0, -cr);
  context.lineTo(d-2*cr, -cr);
  context.quadraticCurveTo(-cr+d, -cr, -cr+d, 0);
  context.lineTo(-cr+d, h-2*cr);
  context.quadraticCurveTo(-cr+d, -cr+h, d-2*cr, -cr+h);
  context.lineTo(0, -cr+h);
  context.quadraticCurveTo(-cr, -cr+h, -cr, h-2*cr);
  context.lineTo(-cr, 0);
  context.quadraticCurveTo(-cr, -cr, 0, -cr);
  context.closePath();
  context.fill();
  if (this.lineWidth > 0) {
      context.stroke();
  }
  //draw the 2 "pins"
  context.beginPath();
  context.arc(0, 0, 2, 0, (Math.PI * 2), true);
  context.closePath();
  context.stroke();
  context.beginPath();
  context.arc(this.width, 0, 2, 0, (Math.PI * 2), true);
  context.closePath();
  context.stroke();
  context.restore();
};

Segment.prototype.getPin = function () {
  return {
    x: this.x + Math.cos(this.rotation) * this.width,
    y: this.y + Math.sin(this.rotation) * this.width
  };
};
```

可以指定宽度、高度和颜色来绘制一个圆角矩形的节段。节段的原点（0，0）和末端各有一个小圆形，这是两个插销，这两个地方用做节段间的连接。（也许你已经注意到有一些属性，比如，vx 和 vy，我们将在 13.4.5 小节中讨论）。

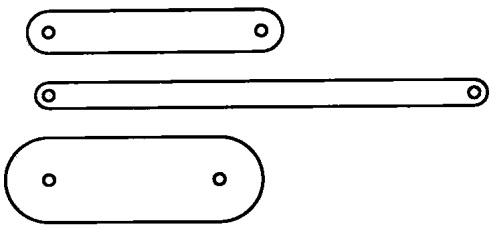


图 13-1 一些简单的节段

下面的例子创建了几个大小不同的节段，用来说明如何使用 Segment 类（文件 01-segment.html），结果如图 13-1 所示。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Segment</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="segment.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        segment0 = new Segment(100, 20),
        segment1 = new Segment(200, 10),
        segment2 = new Segment(80, 40);

    segment0.x = 100;
    segment0.y = 50;
    segment0.draw(context);

    segment1.x = 100;
    segment1.y = 80;
    segment1.draw(context);
    segment2.x = 100;
    segment2.y = 120;
    segment2.draw(context);
  };
  </script>
 </body>
</html>
```

注意，在图 13-1 中，节段的宽度是指两个插销的间距，节段的实际宽度要大于这个距离。你可以看到，每个节段的 x 位置都是 100。尽管它们的左边缘没有对齐，但是左边的插销都是对齐的。当旋转节段时，它绕左边的插销旋转。

Segment 类也包含 getPin 方法，它返回一个 Point 对象，里面包含了右插销点的 x、y 坐标。因为随着节段的旋转，右边插销点的位置会变化，我们使用基本的三角学来计算它。这个点是连接下一个节段的位置——本章的下一小节会讲到。

在下一个例子里，创建一个节段并把它放在 canvas 上。同时添加一个 Slider 类，用来调整节

段的位置。尽管这不是理解运动学所必需的，但是这是一个可以很方便地调整数值的控件。别忘了，你可以从 www.apress.com 下载本书中所有代码，并且都可以免费使用。因为它很简单，我们就不花过多的时间来解释了。你可以拖动滑块来调整数值。别把这个类与 HTML5 中的 slider 元素混淆了，这个控件完全是用 JavaScript 在 canvas 上绘制的。这里是代码，它将在后面的章节中用到（slider.js）：

```
function Slider (min, max, value) {
  this.min = (min === undefined) ? 0 : min;
  this.max = (max === undefined) ? 100 : max;
  this.value = (value === undefined) ? 100 : value;
  this.onchange = null;
  this.x = 0;
  this.y = 0;
  this.width = 16;
  this.height = 100;
  this.backColor = "#cccccc";
  this.backBorderColor = "#999999";
  this.backWidth = 4;
  this.backX = this.width / 2 - this.backWidth / 2;
  this.handleColor = "#eeeeee";
  this.handleBorderColor = "#cccccc";
  this.handleHeight = 6;
  this.handleY = 0;
  this.updatePosition();
}

Slider.prototype.draw = function (context) {
   context.save();
   context.translate(this.x, this.y);
   context.fillStyle = this.backColor;
   context.beginPath();
   context.fillRect(this.backX, 0, this.backWidth, this.height);
   context.closePath();
   context.strokeStyle = this.handleBorderColor;
   context.fillStyle = this.handleColor;
   context.beginPath();
   context.rect(0, this.handleY, this.width, this.handleHeight);
   context.closePath();
   context.fill();
   context.stroke();
   context.restore();
};

Slider.prototype.updateValue = function () {
   var old_value = this.value,
         handleRange = this.height - this.handleHeight,
         valueRange = this.max - this.min;

   this.value = (handleRange - this.handleY) / handleRange * valueRange + this.min;

  if (typeof this.onchange === 'function' && this.value !== old_value) {
     this.onchange();
  }
};
```

```
Slider.prototype.updatePosition = function () {
   var handleRange = this.height - this.handleHeight,
       valueRange = this.max - this.min;

   this.handleY = handleRange - ((this.value - this.min) / valueRange) * handleRange;
};

Slider.prototype.captureMouse = function (element) {
   var self = this,
       mouse = utils.captureMouse(element),
       bounds = {};

   setHandleBounds();

  element.addEventListener('mousedown', function () {
    if (utils.containsPoint(bounds, mouse.x, mouse.y)) {
        element.addEventListener('mouseup', onMouseUp, false);
        element.addEventListener('mousemove', onMouseMove, false);
    }
  }, false);

  function onMouseUp () {
    element.removeEventListener('mousemove', onMouseMove, false);
    element.removeEventListener('mouseup', onMouseUp, false);
    setHandleBounds();
  }

  function onMouseMove () {
    var pos_y = mouse.y - self.y;
    self.handleY = Math.min(self.height - self.handleHeight, Math.max(pos_y, 0));
    self.updateValue();
  }

  function setHandleBounds () {
    bounds.x = self.x;
    bounds.y = self.y + self.handleY;
    bounds.width = self.width;
    bounds.height = self.handleHeight;
  }
};
```

使用 slider（可以在图 13-2 中看到），可以在初始化时传入 **minimun**、**maximum** 和 **value** 作为参数。在下面的例子中，设置 minimum 为–90，maximum 为 90，value 为 0。为了让滑块“看到”鼠标，需要用 slider.captureMouse(canvas)为它注册 canvas 元素，这与前面章节中的 utils.captureMouse（canvas）函数类似。它在 canvas 元素中加入了一些鼠标事件的监听函数，然后检查鼠标光标的位置是否在滑块的边界内。在本例中，你可以看到使用滑块来旋转节段（文件 02-singlesegment.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Single Segment</title>
```

```
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="segment.js"></script>
  <script src="slider.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        segment = new Segment(100, 20),
        slider = new Slider(-90, 90, 0);

    segment.x = 100;
    segment.y = 100;

    slider.x = 300;
    slider.y = 20;
    slider.captureMouse(canvas);
    slider.onchange = drawFrame;

    function drawFrame () {
     context.clearRect(0, 0, canvas.width, canvas.height);

     segment.rotation = slider.value * Math.PI / 180;
     segment.draw(context);
     slider.draw(context);
    }

    drawFrame(); //call once for initial display
  };
   </script>
 </body>
</html>
```

在本例中，只要滑块移动，就会调用绑定在**slider.onchange**事件上的方法，在这里就是**drawFrame**，它把**segment**的角度设置为滑块的值。运行一下，应该能得到如图 13-2 所示的结果，如果工作正常，恭喜你已经完成了正向运动学第一阶段的学习。

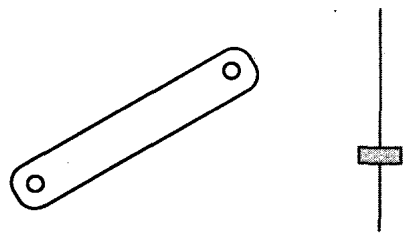


图 13-2 它动了！

13.2.2 移动两个节段

既然我们已经实现了用滑块旋转单个节段，接下来我们就可以实现更加令人兴奋的效果——两个节段！把前面的滑块和节段分别重命名为**slider0**和**segment0**，同时创建另一个滑块和节段，分别为**slider1**和**segment1**。新的滑块用来旋转新的节段，新节段的位置由**segment0**的**getPin()**方法得到。下面是示例**03-two-segments.html**的代码。

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
```

```
 <title>Two Segments</title>
 <link rel="stylesheet" href="style.css">
</head>
<body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <script src="utils.js"></script>
 <script src="segment.js"></script>
 <script src="slider.js"></script>
 <script>
 window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),
       segment0 = new Segment(100, 20),
       segment1 = new Segment(100, 20),
       slider0 = new Slider(-90, 90, 0),
       slider1 = new Slider(-90, 90, 0);

   segment0.x = 100;
   segment0.y = 100;

   slider0.x = 320;
   slider0.y = 20;
   slider0.captureMouse(canvas);
   slider0.onchange = drawFrame;

   slider1.x = 340;
   slider1.y = 20;
   slider1.captureMouse(canvas);
   slider1.onchange = drawFrame;

   function drawFrame () {
    context.clearRect(0, 0, canvas.width, canvas.height);

   segment0.rotation = slider0.value * Math.PI / 180;
   segment1.rotation = slider1.value * Math.PI / 180;
   segment1.x = segment0.getPin().x;
   segment1.y = segment0.getPin().y;

   segment0.draw(context);
   segment1.draw(context);
   slider0.draw(context);
   slider1.draw(context);
  }

  drawFrame(); //call once for initial display
 };
 </script>
</body>
</html>
```

看一下 **drawFrame** 函数中的新代码，你会发现 **segment1** 的位置由 **segment0.getPin()**得到。**drawFrame** 函数在所有对象初始化后第一次调用，以确保它们的位置正确地初始化。

与 **slider0** 一样，通过把 slider1 的 **onchange** 属性指定为 **drawFrame** 函数，可以创建 **slider1** 以调用 **drawFrame** 函数。很明显，**segment1** 的旋转现在由 **slider1** 决定。

如果在浏览器中测试该文件，你会看到当旋转 **segment0** 时，**segment1** 依然与它的末端相连。如图 13-3 所示。但是两者之间并没有什么实际的连接——这只是数学计算。也可以使用滑块独立地旋转 **segment1**。调整两个节段的高度和宽度，它们依然能完美地工作。有一点看起来比较奇怪，尽管 **segment1** 随着 **segment0** 运动，但是并不随着它旋转，就好像里面有个陀螺仪将它的方向稳定住了一样。由于这并不是我们胳膊自然运动的样子，因此它看起来不太自然。正确的效果应该是：**segment1** 的旋转量等于 **segment0** 的旋转量加上 **slider1** 的值。文件 **04-two-segments-2.html** 中的代码做到了这一点：

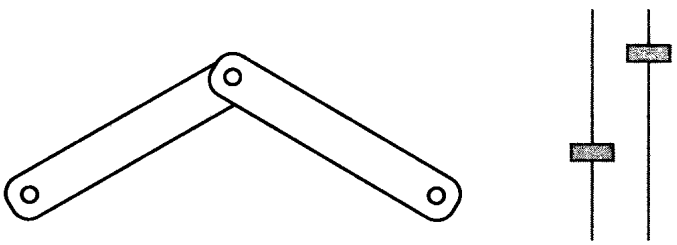


图 13-3 两个节段的正向运动学

```
function drawFrame () {
  context.clearRect(0, 0, canvas.width, canvas.height);
  segment0.rotation = slider0.value * Math.PI / 180;
  segment1.rotation = segment0.rotation + (slider1.value * Math.PI / 180);
  segment1.x = segment0.getPin().x;
  segment1.y = segment0.getPin().y;

  segment0.draw(context);
  segment1.draw(context);
  slider0.draw(context);
  slider1.draw(context);
}
```

现在，它看起来更像是一条真正的手臂了。当然，如果你想要一个真人手臂的效果，就不能像现在这样，肘部可以双向弯曲。为了看起来更自然一些，把 **slider1** 的最小值设置为−160，最大值设置为 0，代码如下：

```
var slider1 = new Slider(-160, 0, 0);
```

现在是反思**正向运动学**这个术语的时候了。整个系统的基础端是 **segment0** 的轴心点，自由端是 **segment1** 的端点，你可以想象一下手臂。基础端的位置和旋转决定 **segment1** 的位置，而 **segment1** 的位置和旋转决定自由端的位置。自由端不能决定它自己的位置，它只是来凑凑热闹。这样，控制权从基础端转移到自由端。

13.3 过程自动化

用于旋转的这些滑块能让你控制图形，但是我们实现的效果像是用液压杆来控制工程机械一样。如果你想要真实的行走效果，就要实现自动控制。

你需要有一个方法来让每个节段平滑地前后摆动，并且要保持同步。这听起来与正弦波有点关系。

在示例 05-walking-1.html 中，用三角函数替换了滑块。用变量 cycle（初始化为 0）的正弦值乘以 90，这样得到-90～90 之间的一个值。变量 cycle 不断增长，就能得到一个振荡波。接下来，用计算出的变量 angle 来控制两个节段。因为把 drawFrame 函数转变为动画循环，用来控制动作，所以动画是不间断的。效果和前一个例子差不多，唯一不同的是现在动画是自动的。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Walking 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="segment.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        segment0 = new Segment(100, 20),
        segment1 = new Segment(100, 20),
        cycle = 0;

    segment0.x = 200;
    segment0.y = 200;

    segment1.x = segment0.getPin().x;
    segment1.y = segment0.getPin().y;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      cycle += 0.02;
      var angle = (Math.sin(cycle) * 90) * Math.PI / 180;

      segment0.rotation = angle;
      segment1.rotation = segment0.rotation + angle;
      segment1.x = segment0.getPin().x;
      segment1.y = segment0.getPin().y;

      segment0.draw(context);
      segment1.draw(context);
   }());
  };
  </script>
 </body>
</html>
```

13.3.1 建立一个自然行走周期

既然你已经实现了一个茫然运动的胳膊，就让我们把它变成腿，进行如下调整。

- 把 segment0 的旋转角度增加 90°，让整个系统向下一些，并且把它的运动范围从 90°

缩小到 45°。

- 为每个节段使用独立的角度变量，angle0 和 angle1。
- 把 angle1 减小为 45，然后再加上 45，这样就使得它的角度范围为 0～90，所以它只能单向弯曲，像是一个真实的膝盖。如果解释得不是很清楚，试一下不加上 45° 会是什么效果，也可以试试其他数值，直到你感觉比较合适为止。

最终的代码在 06-walking-2.html 中可以找到。这里只列出了 drawFrame 方法，因为其他部分没有变化。

```
(function drawFrame () {
   window.requestAnimationFrame(drawFrame, canvas);
   context.clearRect(0, 0, canvas.width, canvas.height);

  cycle += 0.02;
  var angle0 = (Math.sin(cycle) * 45 + 90) * Math.PI / 180,
      angle1 = (Math.sin(cycle) * 45 + 45) * Math.PI / 180;

  segment0.rotation = angle0;
  segment1.rotation = segment0.rotation + angle1;
  segment1.x = segment0.getPin().x;
  segment1.y = segment0.getPin().y;

  segment0.draw(context);
  segment1.draw(context);
}());
```

好了，成功！如图 13-4 所示，这开始看起来像条腿了，至少运动方式像是条腿。

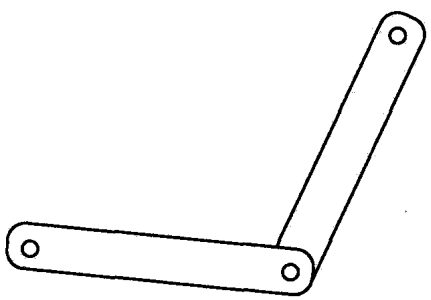


图 13-4 行走周期的开始

但是这并不像是真实的行走，也许它像漫不经心地尝试踢球，而不是行走。现在两个节段同步运动，也就是，同一时间的运动方向相同。如果你分析一下实际的行走周期，就会发现并不是这样。

它们同步运动的原因是使用相同的 cycle 变量来计算其角度。为了避免同步，可以使用两个变量，cycle0 和 cycle1。但是并不需要做这么大的改动，只需要在计算 angle1 的时候在 cycle 上加一点偏移量就可以了，像这样：

```
var angle1 = Math.sin(cycle + offset) * 45 + 45;
```

需要在代码中预先定义 offset，但是 offset 应该是多少呢？只能不断尝试，直到你看起来比较满意为止，不过有一些提示：这个偏移量应该是介于 Math.PI～-Math.PI 之间的一个值（或者 3.14 和-3.14）。任何大于或者小于这个区间的数，都会折回到这个区间。举个例子，使用 -Math.PI/2，会把 angle0 往回旋转四分之一圈。当然，因为-Math.PI/2 大约是-1.57，所以你可以试试附近的其他值，比如，-1.7 或者-1.3，然后观察效果的好坏。随后，我们会放一个滑块来动态调整这个值。加入了这个偏移量的例子在文件 07-walking-3.html 中。

是时候加入另外一条腿了。再加入两个节段，命名为 segment2 和 segment3。segment2 的

位置要与 segment0 相同，因为它也是大腿，segment3 的位置应该由 segment2 的 getPin()决定。

为了避免重复代码，把 segment0 和 segment1 的行走代码抽象为一个方法，叫做 walk：

```
function walk (segA, segB, cyc) {
  var angle0 = (Math.sin(cyc) * 45 + 90) * Math.PI / 180,
      angle1 = (Math.sin(cyc + offset) * 45 + 45) * Math.PI / 180;

  segA.rotation = angle0;
  segB.rotation = segA.rotation + angle1;
  segB.x = segA.getPin().x;
  segB.y = segA.getPin().y;
}
```

这个函数有三个参数：两个节段（segA 和 segB）与 cyc（表示 cycle）。其他代码都是前面用过的。现在，要让 segment0 和 segment1 走起来，这样调用就行了：

```
walk(segment0, segment1, cycle);
```

现在也可以让 segment2 和 segment3 走起来，最终的 drawFrame 函数如下：

```
(function drawFrame () {
   window.requestAnimationFrame(drawFrame, canvas);
   context.clearRect(0, 0, canvas.width, canvas.height);

   cycle += 0.02;
   walk(segment0, segment1, cycle);
   walk(segment2, segment3, cycle);

   segment0.draw(context);
   segment1.draw(context);
   segment2.draw(context);
   segment3.draw(context);
}());
```

但是如果试一下的话，你会发现看不到第二条腿。问题在于，现在两条腿完全同步移动，它们重叠了。你再一次需要消除同步。上次是用一个偏移量在 cycle 上相对于上面节段的位置来调整下面节段的位置，这次，需要用偏移量来调整第二条腿的位置。同样，这实际上是调整 cycle 的值，同样为了避免使用两个变量，在调用 walk 方法之前，为 cycle 加上或减去偏移量就可以了。因此，drawFrame 函数如下：

```
(function drawFrame () {
   window.requestAnimationFrame(drawFrame, canvas);
   context.clearRect(0, 0, canvas.width, canvas.height);

   cycle += 0.02;
   walk(segment0, segment1, cycle);
   walk(segment2, segment3, cycle + Math.PI);
   segment0.draw(context);
   segment1.draw(context);
   segment2.draw(context);
   segment3.draw(context);
}());
```

用 Math.PI 使第二条腿与第一条相差了 180° ，这样当第一条腿向前运动时，第二条向后，反过来也一样。你可以试试其他值，比如 Math.PI/2，这样它看起来更像是急速飞奔，而不是行走或跑步——说不定哪天能用上呢！

代码在文件 08-walking-4.html 中，效果如图 13-5 所示。基础节段（大腿）比下节段（小腿）

稍微大一些。因为使用动态方法来设置所有变量，所以无论尺寸大小它都能正常工作。在下一个版本中，你将会用滑块动态控制更多参数，目前你只能通过手动调整这些变量的值来观察效果的变化。

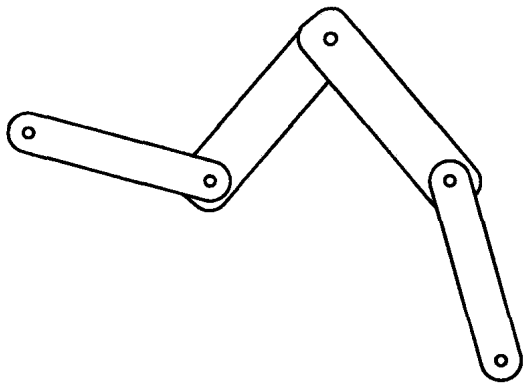
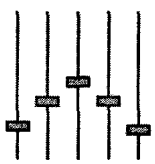


图 13-5 瞧！它走起来了

13.3.2 动态调整

接下来，让我们看看调整一些变量会有什么不同的效果。因为示例 09-walking-5.html 再次使用了 Slider 类，所以你可以随时调整这些变量。

在本例中，我们在屏幕上方放置了 5 个滑块，如图 13-6 所示。



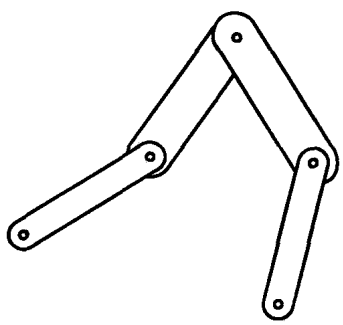


图 13-6 用滑块控制行走

表 13-1 从左到右分别是：滑块名称、滑块作用和设置。这些取值范围和默认值只是让动

画看起来比较自然，你想试试其他值也是完全可以的。

表 13-1 控制行走的滑块

实 例	描 述	设 置
speedSlider （速度滑块）	控制系统整体的速度	最小值：0 最大值：0.2 默认值：0.08
thighRangeSlider （大腿范围滑块）	控制大腿的前后运动范围	最小值：0 最大值：90 默认值：45
thighBaseSlider （大腿基础角度滑块）	控制大腿的基础角度，默认是 90°，也就是说大腿从垂直向下的方向开始前后摆动。通过改变这个值可以得到很多有趣的效果	最小值：0 最大值：180 默认值：90
calfRangeSlider （小腿范围滑块）	控制小腿的前后运动范围	最小值：0 最大值：90 默认值：45
CalfOffsetSlider （小腿偏移量滑块）	控制小腿偏移量（前面用过–Math.PI / 2）	最小值：-3.14 最大值：3.14 默认值：-1.57

现在修改代码，用这些滑块的值来替换以前的硬编码。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Walking 5</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="segment.js"></script>
  <script src="slider.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        segment0 = new Segment(100, 30),
        segment1 = new Segment(100, 20),
        segment2 = new Segment(100, 30),
        segment3 = new Segment(100, 20),
        speedSlider = new Slider(0, 0.2, 0.08),
        thighRangeSlider = new Slider(0, 90, 45),
        thighBaseSlider = new Slider(0, 180, 90),
        calfRangeSlider = new Slider(0, 90, 45),
        calfOffsetSlider = new Slider(-3.14, 3.14, -1.57),
        cycle = 0;
```

```
segment0.x = 200;
segment0.y = 200;

segment1.x = segment0.getPin().x;
segment1.y = segment0.getPin().y;

segment2.x = 200;
segment2.y = 200;

segment3.x = segment2.getPin().x;
segment3.y = segment2.getPin().y;

speedSlider.x = 10;
speedSlider.y = 10;
speedSlider.captureMouse(canvas);

thighRangeSlider.x = 30;
thighRangeSlider.y = 10;
thighRangeSlider.captureMouse(canvas);

thighBaseSlider.x = 50;
thighBaseSlider.y = 10;
thighBaseSlider.captureMouse(canvas);

calfRangeSlider.x = 70;
calfRangeSlider.y = 10;
calfRangeSlider.captureMouse(canvas);

calfOffsetSlider.x = 90;
calfOffsetSlider.y = 10;
calfOffsetSlider.captureMouse(canvas);

function walk (segA, segB, cyc) {
 var angle0 = (Math.sin(cyc) * thighRangeSlider.value + thighBaseSlider.value) *
Math.PI / 180,
     angle1 = (Math.sin(cyc + calfOffsetSlider.value) * calfRangeSlider.value +
calfRangeSlider.value) * Math.PI / 180;

  segA.rotation = angle0;
  segB.rotation = segA.rotation + angle1;
  segB.x = segA.getPin().x;
  segB.y = segA.getPin().y;
}

(function drawFrame () {
   window.requestAnimationFrame(drawFrame, canvas);
   context.clearRect(0, 0, canvas.width, canvas.height);

   cycle += speedSlider.value;
   walk(segment0, segment1, cycle);
   walk(segment2, segment3, cycle + Math.PI);

   segment0.draw(context);
   segment1.draw(context);
```

```
        segment2.draw(context);
        segment3.draw(context);
        speedSlider.draw(context);
        thighRangeSlider.draw(context);
        thighBaseSlider.draw(context);
        calfRangeSlider.draw(context);
        calfOffsetSlider.draw(context);
      }());
    };
    </script>
  </body>
</html>
```

代码基本和前面的一样，只是现在使用滑块的值而不是硬编码的值。试试不同的行走效果，你可以从这个练习中得到不少乐趣。

13.4 让它真实地行走

你现在已经有了两条逼真的腿，但是它们还是悬浮在空中。在本书前面，你学过如何让物体具有速度和加速度，以及如何与环境交互，这里的情况也差不多。

因为本章的这一部分会变得相当复杂，所以我们一步一步来，每次介绍一个概念。最终我们会把所有概念组合起来得到一个例子 10-real-walk.html。

13.4.1 给它一些空间

因为我们想要的效果是让它在屏幕上走来走去，所以把所有部件都缩小一点，给它腾出空间。把原来的 4 个节段都缩小一半：

```
var segment0 = new Segment(50, 15),
    segment1 = new Segment(50, 10),
    segment2 = new Segment(50, 15),
    segment3 = new Segment(50, 10);
```

接下来，因为它要行走并与边界交互，所以定义 vx 和 vy 变量。

```
var vx = 0,
    vy = 0;
```

现在，如果运行动画，你就会看到上一个例子的缩小版。

13.4.2 加入重力

现在需要加入重力，否则，即使你加入了边界检测，两条腿还是会悬浮在空中。新加入一个滑块来控制重力，命名为 gravitySlider，设置最小值为 0，最大值为 1，默认值为 0.2。下面是初始化这个滑块的代码：

```
var gravitySlider = new Slider(0, 1, 0.2);

gravitySlider.x = 110;
gravitySlider.y = 10;
gravitySlider.captureMouse(canvas);
```

还需要根据重力加速度计算速度。为了避免所有代码都混在 drawFrame 函数中，新建一个函数叫做 setVelocity：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  cycle += speedSlider.value;
  setVelocity();
  walk(segment0, segment1, cycle);
  walk(segment2, segment3, cycle + Math.PI);

  segment0.draw(context);
  segment1.draw(context);
  segment2.draw(context);
  segment3.draw(context);
  speedSlider.draw(context);
  thighRangeSlider.draw(context);
  thighBaseSlider.draw(context);
  calfRangeSlider.draw(context);
  calfOffsetSlider.draw(context);
  gravitySlider.draw(context);
}());
```

在这个函数中，把重力加在 vy 上，然后把 vx 和 vy 加在 segment0 与 segment2 的位置上。记住，不用担心 segment1 和 segment3，因为它们的位置是根据上级节段计算而来的。

```
function setVelocity () {
  vy += gravitySlider.value;
  segment0.x += vx;
  segment0.y += vy;
  segment2.x += vx;
  segment2.y += vy;
}
```

根据意愿可以测试目前这个版本，不过它并不令人兴奋。还没有 x 轴上的速度，而且重力把腿拉到了地面以下。所以，需要检测腿与地面是否接触，这意味着，要进行碰撞检测。

13.4.3 处理碰撞

首先，让 drawFrame 调用另一个函数：checkFloor。因为这个调用在 walk 调用之后，所以它使用的是最新的位置。只检测 segment1 和 segment3——也就是小腿是否接触了地面。所以，把它们传入 checkFloor。

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  cycle += speedSlider.value;
  setVelocity();
  walk(segment0, segment1, cycle);
  walk(segment2, segment3, cycle + Math.PI);
  checkFloor(segment1);
```

```
    checkFloor(segment3);

    segment0.draw(context);
    segment1.draw(context);
    segment2.draw(context);
    segment3.draw(context);
    speedSlider.draw(context);
    thighRangeSlider.draw(context);
    thighBaseSlider.draw(context);
    calfRangeSlider.draw(context);
    calfOffsetSlider.draw(context);
    gravitySlider.draw(context);
}());
```

现在进入了第一个有趣的部分：碰撞检测。你想知道节段的底部是否超出了下边界。在本例中，也就是 canvas.height。

可能最简单的做法就是做个粗略的计算：取得节段下端插销点的 y 坐标，然后加上节段末端多余的部分，最后判断这个值是否大于 canvas 的高度。这不是最精确的计算方法，但是它已经能满足本例的目的了，下面开始写 checkFloor 函数：

```
function checkFloor (seg) {
  var yMax = seg.getPin().y + (seg.height / 2);

  if (yMax > canvas.height) {
     //do reaction...
  }
}
```

如果你检测到 yMax 大于 canvas.height（腿碰到了地面），你要做什么？就像其他的边界碰撞一样，首先要调整物体的位置。如果 yMax 是节段的下边界，canvas.height 是地面，就要把节段往回调整一段距离，这个距离等于它们的间距。换句话说，如果 canvas.height 是 400，yMax 是 420，就需要把节段的 y 坐标减去 20，但是并不能只调整这个节段，要移动所有的节段，因为它们都是身体的一部分，必须作为一个整体移动。所以，得到如下代码：

```
function checkFloor (seg) {
  var yMax = seg.getPin().y + (seg.height / 2);

  if (yMax > canvas.height) {
      var dy = yMax - canvas.height;
      segment0.y -= dy;
      segment1.y -= dy;
      segment2.y -= dy;
      segment3.y -= dy;
  }
}
```

这个版本的例子值得多运行几遍。调整滑块，观察不同的行走效果。双腿与地板的交互是不是看起来更有感觉。当然，这仍不是真实的行走——胜利就在前方。

13.4.4 处理反作用力

现在双腿已经成功地与地面接触，但是仅仅重置它们还不够，这还不是真实的交互。行走

的目的是为了水平移动——在本例中就是 x 轴上的速度。此外，一个行走周期也会产生一点点 y 速度，至少是在抵消重力的那一瞬间。在跑步时会更加明显，每个周期都会有点腾空。

观察这个问题的一个方法是：你的脚向下运动，当它碰到地面时，它就不能再向下了，所以垂直方向的动量就会反弹到你的身体，使身体向上运动。脚踩下去的力量越大，反弹回来的力量也就越大。同理，如果脚落地时向后蹬，则水平方向的动量也会推动身体向前。脚向后的力量越大，水平方向的推动力越大。

如果你能知道脚的 x 和 y 速度，然后当你检测到碰撞时，就能从 vx 和 vy 值中减去 x、y 速度。但是，现在并没有脚，只能添加一双脚，然后计算它们的位置，或者作为替代，使用虚拟的脚，使用小腿节段的 getPin()的返回值作为脚的位置。

如果你能记录节段的插销点在运动前后的位置，你就可以把这两个位置相减，从而得到脚在 x、y 轴上的速度。这可以在 walk 函数中实现，把计算出的值保存在节段的 vx 和 vy 属性中（现在你看到了哪些属性的来源）。

```
function walk (segA, segB, cyc) {
  var angle0 = (Math.sin(cyc) * thighRangeSlider.value + thighBaseSlider.value) * Math.PI / 180,
     angle1=(Math.sin(cyc + calfOffsetSlider.value) * calfRangeSlider.value +
  calfRangeSlider.value)* Math.PI / 180,
     foot = segB.getPin();

segA.rotation = angle0;
segB.rotation = segA.rotation + angle1;
segB.x = segA.getPin().x;
segB.y = segA.getPin().y;
segB.vx = segB.getPin().x - foot.x;
segB.vy = segB.getPin().y - foot.y;
}
```

每个小腿节段都有 vx 和 vy 属性，它们并不表示节段自己的速度，而是节段底部插销点或者虚拟脚的速度。

那么用这速度来做什么呢？当检测到与地面接触时，然后把它从总速度中减去。换句话说，如果脚以每帧 3 个像素移动（vy 为 3），当它接触到地面时，就从总 vy 中减去 3。对于 vx 也是如此，在代码里很简单：

```
function checkFloor (seg) {
  var yMax = seg.getPin().y + (seg.height / 2);

  if (yMax > canvas.height) {
      var dy = yMax - canvas.height;
      segment0.y -= dy;
      segment1.y -= dy;
      segment2.y -= dy;
      segment3.y -= dy;
      vx -= seg.vx;
      vy -= seg.vy;
  }
}
```

这是个极度简单，而且可能很不精确的方法，用来体现行走的力量。如果测试一下，就能

看到两条腿在屏幕上走过！

13.4.5 屏幕环绕，重复

现在这双腿走过屏幕，再也回不来了——但是屏幕环绕可以解决这个问题。当两条腿从右边离开时，就把它们移动到左边。这比之前复杂一点，因为现在有 4 个部件要一起移动，而不是一个。但是别忘了，只需要移动一条大腿，因为它们的位置始终相同，并且小腿的位置由大腿决定。添一个叫做 checkWalls 的函数，在 drawFrame 检查完地面碰撞后调用该函数。

```
(function drawFrame () {
   window.requestAnimationFrame(drawFrame, canvas);
   context.clearRect(0, 0, canvas.width, canvas.height);

   cycle += speedSlider.value;
   setVelocity();
   walk(segment0, segment1, cycle);
   walk(segment2, segment3, cycle + Math.PI);
   checkFloor(segment1);
   checkFloor(segment3);
   checkWalls();

   segment0.draw(context);
   segment1.draw(context);
   segment2.draw(context);
   segment3.draw(context);
   speedSlider.draw(context);
   thighRangeSlider.draw(context);
   thighBaseSlider.draw(context);
   calfRangeSlider.draw(context);
   calfOffsetSlider.draw(context);
   gravitySlider.draw(context);
}());
```

让我们留出 100 个像素的边缘空白，这样，当两条腿中的任意一条走到屏幕右边 100 个像素后才会回到左边。这就相当于给 canvas 增加了 200 个像素的宽度，每一边各 100 个像素的边缘。checkWalls 函数里面的 if 语句如下：

```
function checkWalls () {
  var w = canvas.width + 200;

  if (segment0.x > canvas.width + 100) {
      segment0.x -= w;
      segment1.x -= w;
      segment2.x -= w;
      segment3.x -= w;
  }
}
```

然后左边也一样，因为有些时候腿会倒走，下面是完整的 checkWalls 函数：

```
function checkWalls () {
  var w = canvas.width + 200;

  if (segment0.x > canvas.width + 100) {
      segment0.x -= w;
```

```
        segment1.x -= w;
        segment2.x -= w;
        segment3.x -= w;
    } else if (segment0.x < -100) {
        segment0.x += w;
        segment1.x += w;
        segment2.x += w;
        segment3.x += w;
    }
}
```

大功告成，如果你觉得还有些不清楚，这里给出了完整的代码，也可以在文件 10-real- walk. html 中找到：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Real Walk</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="segment.js"></script>
  <script src="slider.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        segment0 = new Segment(50, 15),
        segment1 = new Segment(50, 10),
        segment2 = new Segment(50, 15),
        segment3 = new Segment(50, 10),
        speedSlider = new Slider(0, 0.2, 0.08),
        thighRangeSlider = new Slider(0, 90, 45),
        thighBaseSlider = new Slider(0, 180, 90),
        calfRangeSlider = new Slider(0, 90, 45),
        calfOffsetSlider = new Slider(-3.14, 3.14, -1.57),
        gravitySlider = new Slider(0, 1, 0.2),
        cycle = 0,
        vx = 0,
        vy = 0;

    segment0.x = 200;
    segment0.y = 200;

    segment1.x = segment0.getPin().x;
    segment1.y = segment0.getPin().y;

    segment2.x = 200;
    segment2.y = 200;

    segment3.x = segment2.getPin().x;
    segment3.y = segment2.getPin().y;

    speedSlider.x = 10;
    speedSlider.y = 10;
```

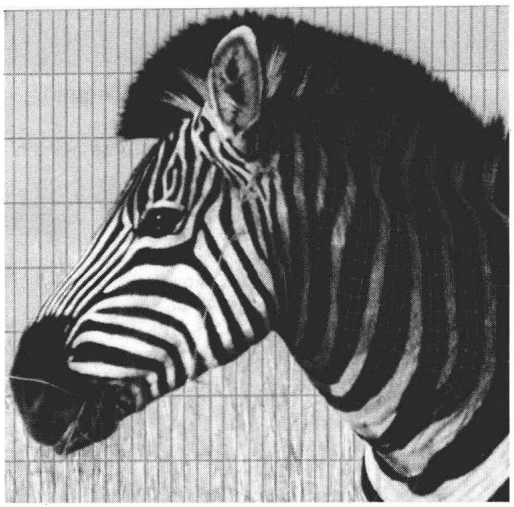
```
        speedSlider.captureMouse(canvas);

        thighRangeSlider.x = 30;
        thighRangeSlider.y = 10;
        thighRangeSlider.captureMouse(canvas);

        thighBaseSlider.x = 50;
        thighBaseSlider.y = 10;
        thighBaseSlider.captureMouse(canvas);

        calfRangeSlider.x = 70;
        calfRangeSlider.y = 10;
        calfRangeSlider.captureMouse(canvas);

        calfOffsetSlider.x = 90;
        calfOffsetSlider.y = 10;
        calfOffsetSlider.captureMouse(canvas);

        gravitySlider.x = 110;
        gravitySlider.y = 10;
        gravitySlider.captureMouse(canvas);

        function setVelocity () {
          vy += gravitySlider.value;
          segment0.x += vx;
          segment0.y += vy;
          segment2.x += vx;
          segment2.y += vy;
        }

        function walk (segA, segB, cyc) {
          var angle0 = (Math.sin(cyc) * thighRangeSlider.value + thighBaseSlider.value) * Math.PI / 180,
              angle1 = (Math.sin(cyc + calfOffsetSlider.value) * calfRangeSlider.value +
calfRangeSlider.value) * Math.PI / 180,
              foot = segB.getPin();

          segA.rotation = angle0;
          segB.rotation = segA.rotation + angle1;
          segB.x = segA.getPin().x;
          segB.y = segA.getPin().y;
          segB.vx = segB.getPin().x - foot.x;
          segB.vy = segB.getPin().y - foot.y;
        }
        function checkFloor (seg) {
           var yMax = seg.getPin().y + (seg.height / 2);

           if (yMax > canvas.height) {
               var dy = yMax - canvas.height;
               segment0.y -= dy;
               segment1.y -= dy;
               segment2.y -= dy;
               segment3.y -= dy;
               vx -= seg.vx;
               vy -= seg.vy;
        }
```

```
    }

    function checkWalls () {
      var w = canvas.width + 200;

      if (segment0.x > canvas.width + 100) {
          segment0.x -= w;
          segment1.x -= w;
          segment2.x -= w;
          segment3.x -= w;
        } else if (segment0.x < -100) {
          segment0.x += w;
          segment1.x += w;
          segment2.x += w;
          segment3.x += w;
        }
    }

    (function drawFrame () {
       window.requestAnimationFrame(drawFrame, canvas);
       context.clearRect(0, 0, canvas.width, canvas.height);

       cycle += speedSlider.value;
       setVelocity();
       walk(segment0, segment1, cycle);
       walk(segment2, segment3, cycle + Math.PI);
       checkFloor(segment1);
       checkFloor(segment3);
       checkWalls();

       segment0.draw(context);
       segment1.draw(context);
       segment2.draw(context);
       segment3.draw(context);
       speedSlider.draw(context);
       thighRangeSlider.draw(context);
       thighBaseSlider.draw(context);
       calfRangeSlider.draw(context);
       calfOffsetSlider.draw(context);
       gravitySlider.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

13.5 小结

你在本章已经完成了非常强大的作品，征服了基础的正向运动学。这里给出的方法可能并不是这个学科唯一的解决方案，它们是为特定的应用和技术定制的：让物体行走。不必局限于此，尝试去改变，或者加入新的东西，多做实验，看看能得到什么结果。

接下来，你将看到事物的另一面：反向运动学。



第 14 章　反向运动学：拖曳与伸出

本章涵盖以下内容：

- 伸出（reaching）或拖曳（dragging）单个节段
- 拖曳多个节段
- 伸出多个节段
- 使用标准的反向运动学方法

第 13 章介绍了运动学基础以及正向运动学和反向运动的区别。第 13 章是关于正向运动学的，本章将介绍与它密切相关的反向运动学：拖曳和伸出的运动。

与正向运动学的例子类似，本章也是从单个节段开始建立系统。一开始只有一个节段，然后是多个。首先，你将学到计算角度和位置最简单的方法，这只是用基本的三角学计算的近似值。然后，我们会介绍另一种使用余弦定理的方法，这个方法更精确，但代价是更复杂——这就需要权衡。

14.1　伸出和拖曳单个节段

当系统的自由端伸向一个目标时，系统的另一端（基础端）可能是固定的。所以，如果目标超出范围，自由端有可能永远都够不到它。举个例子，当你伸手去抓一个东西时，你的手指向目标移动，手腕转动使得手指越来越接近，肘部、肩膀以及身体的其他部分都尽量地调整到最佳位置，以尽可能地够到那个目标。有时候，这些肢体的位置组合可以让手指够到那个物体，但是有时候并不能。如果物体来回摆动，你所有的肢体都要即时调整位置以保证手指离目标越

来越近。反向运动学会告诉你如何调整位置以实现最佳的伸出效果。

反向运动学的另一种类型是拖曳物体。这种情况下，自由端被外力拖动。无论它被拖到哪里，系统的其他部分都跟随其后，位置由物理原理决定。举个例子，想象一个人躺在地板上，你抓住他的手把他拖来拖去，你施加在他手上的力量会传递到手腕、肘部、肩膀以及身体其他部位，驱动它们沿着拖动的方向移动。在这个例子里，反向运动学告诉我们当各个部件被拖动时位置如何变化。

为了说明这两种方法的区别——拖曳和伸出，让我们开始用单个节段为它们分别举例，因为你需要用到第 13 章中的 Segment 类，所以你要先确保在文件中引用了那段脚本。

14.1.1 伸出单个节段

对于伸出而言，所有节段都要向目标旋转。在这个例子中，目标就是鼠标指针。为了让节段向目标旋转，需要知道在 x 轴和 y 轴上两点间的距离。然后可以用 Math.atan2 得到它们间的弧度值，这个可以用来旋转节段。下面给出代码（也可以在 01-one-segment.html 中找到）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>One Segment</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="segment.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        segment0 = new Segment(100, 20);

    segment0.x = canvas.width / 2;
    segment0.y = canvas.height / 2;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var dx = mouse.x - segment0.x,
          dy = mouse.y - segment0.y;
      segment0.rotation = Math.atan2(dy, dx);
      segment0.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

运行结果如图 14-1 所示，测试本例，观察节段如何随着鼠标指针旋转。即使节段距离鼠标很远，它看起来也像是在尽量地够到鼠标指针。

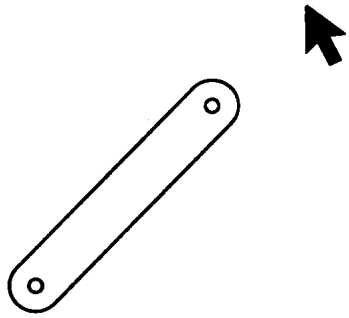


图 14-1 单个节段伸向鼠标指针

14.1.2 拖曳单个节段

现在，我们来试试拖曳。拖曳方法的开始部分与伸出方法相同：向鼠标指针方向旋转节段。但是还要多出一步，把节段的第二个轴心点移动到鼠标指针位置。这需要知道两个轴心点在 x、y 轴上的距离——可以叫做 w 和 h，它们可以通过节段的位置以及 getPin()方法的返回值计算得到。然后从当前鼠标指针位置中间再减去 w 和 h，这就是节段要移动到的位置。下面是 02-one-segment-drag.html 中的 drawFrame 函数，只有这段代码相对上例发生了变化。

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  var dx = mouse.x - segment0.x,
      dy = mouse.y - segment0.y;

  segment0.rotation = Math.atan2(dy, dx);

  var w = segment0.getPin().x - segment0.x,
      h = segment0.getPin().y - segment0.y;

  segment0.x = mouse.x - w;
  segment0.y = mouse.y - h;

  segment0.draw(context);
}());
```

现在这个节段与鼠标指针永远相连，被鼠标指针拖着走。你甚至可以推着这个节段反方向走。

14.2 拖曳多个节段

因为用反向运动学拖曳一个系统比伸出要简单一点，所以先介绍拖曳。我们先从两个节段开始。

14.2.1 拖曳两个节段

使用上一个例子，在代码开始部分再创建一个节段 segment1。因为已经把 segment0 拖曳在鼠标指针位置上了，现在把 segment1 拖曳在 segment0 上。首先，可以复制前面的一些代码——只须改变几个引用。在 drawFrame 函数中用粗体标出的是新加入的部分：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  var dx = mouse.x - segment0.x,
      dy = mouse.y - segment0.y;

  segment0.rotation = Math.atan2(dy, dx);

  var w = segment0.getPin().x - segment0.x,
      h = segment0.getPin().y - segment0.y:

  segment0.x = mouse.x - h;
  segment0.y = mouse.y - w;

  dx = segment0.x - segment1.x;
  dy = segment0.y - segment1.y;

  segment1.rotation = Math.atan2(dy, dx);

  w = segment1.getPin().x - segment1.x;
  h = segment1.getPin().y - segment1.y;
  segment1.x = segment0.x - w;
  segment1.y = segment0.y - h;

  segment0.draw(context);
  segment1.draw(context);
}());
```

在新的代码中，计算出 segment1 和 segment0 的距离，然后基于距离计算角度、旋转和移动 segment1——重用前一次计算中的变量。你可以测试该示例，看看这个漂亮又真实的双节段系统。

在接下来的例子里，新建一个 drag 函数，用于移除一些重复代码。函数的参数是一个节段以及拖动的目标点（x，y）。然后把 segment0 移动到(mouse.x，mouse.y)，把 segment1 移动到(segment0.x，segment0.y)。下面是完整的代码（03-two-segment-drag.htm）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Two Segment Drag</title>
  <link rel="stylesheet" href="style.css">
 </head>
```

```
<body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <script src="utils.js"></script>
 <script src="segment.js"></script>
 <script>
 window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),
       mouse = utils.captureMouse(canvas),
       segment0 = new Segment(100, 20),
       segment1 = new Segment(100, 20);

   function drag (segment, xpos, ypos) {
     var dx = xpos - segment.x,
         dy = ypos - segment.y;

     segment.rotation = Math.atan2(dy, dx);

     var w = segment.getPin().x - segment.x,
         h = segment.getPin().y - segment.y;

     segment.x = xpos - w;
     segment.y = ypos - h;
   }

   (function drawFrame () {
     window.requestAnimationFrame(drawFrame, canvas);
     context.clearRect(0, 0, canvas.width, canvas.height);

     drag(segment0, mouse.x, mouse.y);
     drag(segment1, segment0.x, segment0.y);

     segment0.draw(context);
     segment1.draw(context);
   }());
 };
 </script>
</body>
</html>
```

14.2.2 拖曳更多节段

现在你想加入多少节段都可以。在接下来的例子中，加入 5 个节段，命名为 segment0～segment4，把它们存储在一个数组中。循环把每个节段传入 drag 函数。下面是代码（文件 04-multi-segment-drag.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Multi Segment Drag</title>
  <link rel="stylesheet" href="style.css">
 </head>
```

```
<body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <script src="utils.js"></script>
 <script src="segment.js"></script>
 <script>
 window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        segments = [],
        numSegments = 5;

    while (numSegments--) {
      segments.push(new Segment(50, 10));
    }

    function drag (segment, xpos, ypos) {
      var dx = xpos - segment.x,
          dy = ypos - segment.y;

      segment.rotation = Math.atan2(dy, dx);

      var w = segment.getPin().x - segment.x,
          h = segment.getPin().y - segment.y;

      segment.x = xpos - w;
      segment.y = ypos - h;
    }

    function move (segment, i) {
      if (i !== 0) {
          drag(segment, segments[i-1].x, segments[i-1].y);
      }
    }

    function draw (segment, i) {
      segment.draw(context);
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      drag(segments[0], mouse.x, mouse.y);
      segments.forEach(move);

      segments.forEach(draw);
    }());
 };
 </script>
</body>
</html>
```

在 drawFrame 动画循环中，把第一个节段放置在鼠标指针位置。然后，把每个节段作为参

数传入 move 函数，后一个节段跟随前一个节段（用 drag 函数）。在 move 函数中不处理第一个节段，因为它已经处理过了。图 14-2 展示了运行效果。

现在，你已经有了反向运动学的基础。你可以修改 numSegments 变量来加入任意多个节段。如图 14-3 所示，你可以看到加入了 50 个节段，这展示了系统是多么健壮。

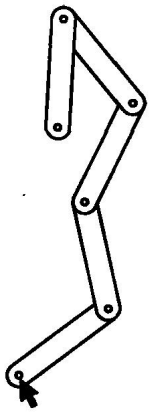


图 14-2 拖动多个节段

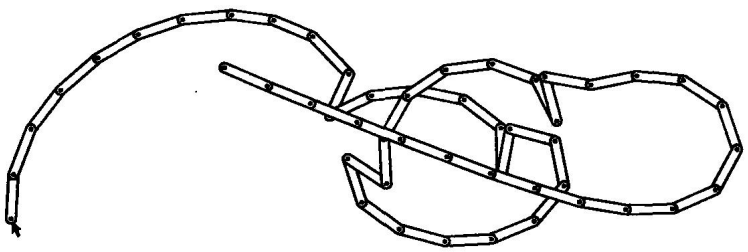


图 14-3 拖曳 50 个节段

14.3 伸出多个节段

下面的例子基于本章的第一个例子（01-one-segment.html）创建。在一个节段伸向鼠标指针位置的基础上，增加了伸向一个目标物体的内容。

14.3.1 伸向鼠标位置

为了完成多个节段的伸出，首先要确定触及目标的那个节段的位置，这与拖曳节段时用于放置节段的计算方法相同。在当前情况下，只需要找出那个位置，而并不需要移动节段。这个位置作为下一个节段旋转的目标点，依次循环。直到系统的基础端时，然后再反向回去，把每个节段放在其父级节段的末端。图 14-4 阐明了这一过程：

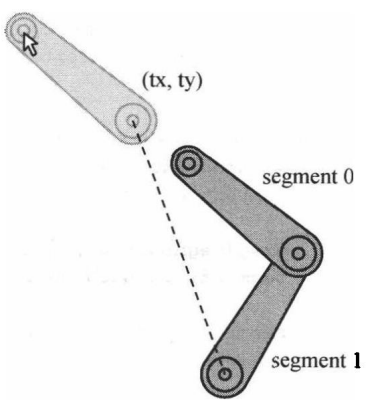


图 14-4 segment0 向鼠标指针移动。segment1 跟随 segment0，向（tx，ty）移动

首先为例子加入另一个节段。在脚本的开始，创建一个节段，把它放置在 canvas 元素的中央。因为这个节段将作为基础端，所以同时移除掉 segment0 的位置初始化代码：

```
var segment1 = new Segment(100, 20);
```

```
segment1.x = canvas.width / 2;
segment1.y = canvas.height / 2;
```

下一步是找出 segment0 接触到目标点的位置。再次重申，这个点与拖曳例子中的一样，但是不要移动它，只要把它把存起来就行了。所以，得到如下代码：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  var dx = mouse.x - segment0.x,
      dy = mouse.y - segment0.y;

  segment0.rotation = Math.atan2(dy, dx);

  var w = segment0.getPin().x - segment0.x,
      h = segment0.getPin().y - segment0.y,
      tx = mouse.x - w,
      ty = mouse.y - h;

  segment0.draw(context);
  segment1.draw(context);
}());
```

我们把这个点叫做(tx,ty)，因为这是 segment1 旋转的目标点。

接下来，加入代码让 segment1 向它的目标点旋转。代码与计算 segment0 的一样，只是节段与目标点不同。

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  var dx = mouse.x - segment0.x,
      dy = mouse.y - segment0.y;

  segment0.rotation = Math.atan2(dy, dx);

  var w = segment0.getPin().x - segment0.x,
      h = segment0.getPin().y - segment0.y,
      tx = mouse.x - w,
      ty = mouse.y - h;

  dx = tx - segment1.x;
  dy = ty - segment1.y;

  segment1.rotation = Math.atan2(dy, dx);

  segment0.draw(context);
  segment1.draw(context);
}());
```

最终，重置 segment0 到 segment1 的末端，因为 segment1 现在已经旋转到不同的位置：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);
```

```
    var dx = mouse.x - segment0.x,
        dy = mouse.y - segment0.y;

    segment0.rotation = Math.atan2(dy, dx);

    var w = segment0.getPin().x - segment0.x,
        h = segment0.getPin().y - segment0.y,
        tx = mouse.x - w,
        ty = mouse.y - h;

    dx = tx - segment1.x;
    dy = ty - segment1.y;

    segment1.rotation = Math.atan2(dy, dx);

    segment0.x = segment1.getPin().x;
    segment0.y = segment1.getPin().y;

    segment0.draw(context);
    segment1.draw(context);
}());
```

运行代码，你会看到两个节段作为一个整体伸向鼠标指针。现在，整理一下代码，这样你就能简单地加入更多节段。把所有旋转代码放到 reach 函数中：

```
function reach (segment, xpos, ypos) {
  var dx = xpos - segment.x,
      dy = ypos - segment.y;

  segment.rotation = Math.atan2(dy, dx);

  var w = segment.getPin().x - segment.x,
      h = segment.getPin().y - segment.y;

  return {
    x: xpos - w,
    y: ypos - h
  };
}
```

这个函数旋转作为参数传入的 segment，然后返回一个包含 x、y 属性的对象（基于前一次的 tx 和 ty 计算得来）。这样就可以用 reach 函数来旋转一个节段，并返回传递给下一次调用的目标点。因此，drawFrame 函数变成下面这样：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  var target = reach(segment0, mouse.x, mouse.y);
  reach(segment1, target.x, target.y);

  segment0.draw(context);
  segment1.draw(context);
}());
```

这里，segment0 伸向鼠标指针，segment1 伸向 segment0。然后可以把重置位置的代码封装为其自己的函数：

```
function position (segmentA, segmentB) {
  segmentA.x = segmentB.getPin().x;
  segmentA.y = segmentB.getPin().y;
}
```

你可以像下面这样调用，把 segment0 放置在 segment1 的末端：

```
position(segment0, segment1);
```

这里是最终的代码（05-two-segment-reach.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Two Segment Reach</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="segment.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        segment0 = new Segment(100, 20),
        segment1 = new Segment(100, 20);

    segment1.x = canvas.width / 2;
    segment1.y = canvas.height / 2;

    function reach (segment, xpos, ypos) {
      var dx = xpos - segment.x,
          dy = ypos - segment.y;

      segment.rotation = Math.atan2(dy, dx);

      var w = segment.getPin().x - segment.x,
          h = segment.getPin().y - segment.y;

      return {
        x: xpos - w,
        y: ypos - h
      };
    }

    function position (segmentA, segmentB) {
      segmentA.x = segmentB.getPin().x;
      segmentA.y = segmentB.getPin().y;
    }

    (function drawFrame () {
```

```
        window.requestAnimationFrame(drawFrame, canvas);
        context.clearRect(0, 0, canvas.width, canvas.height);

        var target = reach(segment0, mouse.x, mouse.y);
        reach(segment1, target.x, target.y);
        position(segment0, segment1);

        segment0.draw(context);
        segment1.draw(context);
      }());
    };
    </script>
  </body>
</html>
```

在现在代码的基础上，我们很容易就能使用一个数组存储任意多个节段，代码如下（06-multi-segment-reach.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Multi Segment Reach</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="segment.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        segments = [],
        numSegments = 5,
        target; //referenced by drawFrame and move

    while (numSegments--) {
      segments.push(new Segment(50, 10));
    }

    //center the last one
    segments[segments.length - 1].x = canvas.width / 2;
    segments[segments.length - 1].y = canvas.height / 2;

    function reach (segment, xpos, ypos) {
      var dx = xpos - segment.x,
          dy = ypos - segment.y;

      segment.rotation = Math.atan2(dy, dx);

      var w = segment.getPin().x - segment.x,
          h = segment.getPin().y - segment.y;
```

```
    return {
      x: xpos - w,
      y: ypos - h
    };
  }

  function position (segmentA, segmentB) {
    segmentA.x = segmentB.getPin().x;
    segmentA.y = segmentB.getPin().y;
  }

  function move (segment, i) {
    if (i !== 0) {
        target = reach(segment, target.x, target.y);
        position(segments[i - 1], segment);
    }
  }

  function draw (segment) {
    segment.draw(context);
  }

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    target = reach(segments[0], mouse.x, mouse.y);
    segments.forEach(move);

    segments.forEach(draw);
  }());
 };
 </script>
</body>
</html>
```

本例的运行结果如图 14-5 所示。当然，这个节段链条并不需要一直追着鼠标指针跑。让我们来看看给它一个玩具会发生什么。

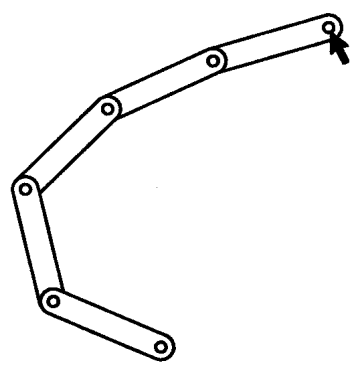


图 14-5 伸出多个节段

14.3.2 伸向一个物体

再次以上一个例子为基础，这次需要用到 Ball 类，在文件中引用它，然后在代码开始部分加入几个变量，用于小球的移动：

```
var ball = new Ball(20),
    gravity = 0.5,
    bounce = -0.9;
ball.vx = 10; //start the ball with velocity on the x axis
```

还要把这些节段缩小一些，腾出更多的运动空间。你可以调整这些值，尝试找出最佳动画效果：

```
while (numSegments--) {
  segments.push(new Segment(20, 10));
}
```

为了避免混乱，创建另一个函数 moveBall，把移动小球的代码从程序中分离出来。

```
function moveBall () {
  ball.vy += gravity;
  ball.x += ball.vx;
  ball.y += ball.vy;
  if (ball.x + ball.radius > canvas.width) {
   ball.x = canvas.width - ball.radius;
   ball.vx *= bounce;
  } else if (ball.x - ball.radius < 0) {
   ball.x = ball.radius;
   ball.vx *= bounce;
  }
  if (ball.y + ball.radius > canvas.height) {
   ball.y = canvas.height - ball.radius;
   ball.vy *= bounce;
  } else if (ball.y - ball.radius < 0) {
   ball.y = ball.radius;
   ball.vy *= bounce;
  }
}
```

最后，在 drawFrame 动画循环中，调用 moveBall 函数，让第一个节段伸向球而不是鼠标指针：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);
  moveBall();
  target = reach(segments[0], ball.x, ball.y);
  segments.forEach(move);
  segments.forEach(draw);

  ball.draw(context);
}());
```

运行结果如图 14-6 所示。小球跳来跳去，胳膊紧紧相随。很神奇，是吧？

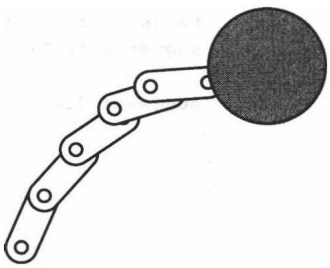


图 14-6 胳膊喜欢玩球

现在的现象是：胳膊去触摸小球，但小球完全不理胳膊。接下来让它们发生交互。

14.3.3 加入一些交互

小球与胳膊如何交互取决于你想让它们怎么做。但是，无论怎么做，首先都需要检测碰撞。然后，如果发生碰撞则做出相应的反应。同样要把这些代码封装在自己的函数里，并在 drawFrame 动画循环中调用它。

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  moveBall();
  target = reach(segments[0], ball.x, ball.y);
  segments.forEach(move);
  checkHit();
```

```
  segments.forEach(draw);
  ball.draw(context);
}());
```

因为碰撞检测需要在运动代码之后执行，所以它在最后调用。下面开始写 checkHit 函数：

```
function checkHit () {
  var segment = segments[0],
       dx = segment.getPin().x - ball.x,
       dy = segment.getPin().y - ball.y,
       dist = Math.sqrt(dx * dx + dy * dy);

  if (dist < ball.radius) {
      //reaction goes here
  }
}
```

首先要做的是计算第一节手臂的末端插销点到小球的距离，然后使用基于距离的方法来检测碰撞。

当检测到碰撞时，就要做些事情。比如，手臂把小球抛向空中（负 y 速度）并随机地在 x 轴上移动（随机 x 速度），像这样：

```
function checkHit () {
  var segment = segments[0],
       dx = segment.getPin().x - ball.x,
       dy = segment.getPin().y - ball.y,
       dist = Math.sqrt(dx * dx + dy * dy);

  if (dist < ball.radius) {
     ball.vx += Math.random() * 2 - 1;
     ball.vy -= 1;
  }
}
```

效果非常好，最终的代码可以在 07-play-ball.html 中找到。如果胳膊把球丢掉了，别难过，可以始终再次刷新浏览器就能继续玩。但是这个例子终究只是个例子，你可能想用这个技术来使胳膊抓住球，扔向一个目标，或者让两个手臂互相传球。试试不同的交互，你已经掌握了足够的知识来制作一些有趣的东西了。

14.4 使用标准反向运动学方法

我们前面描述的反向运动学方法比较容易理解和编程，而且不需要占用过多的系统资源，最重要的是能正常工作。但是还有一个更标准的数学方法来解决这个问题，这就是余弦定理。我们来看看处理反向运动学的标准方法，这样你就有了两种不同的方法可以选择。

14.4.1 介绍余弦定理

计算反向运动学的方法叫做余弦定理。回忆一下第 3 章，所有例子都使用直角三角形，也

就是有一个是直角（90°）的三角形。这样的三角形有一些很简单的规则：正弦等于对边长度比斜边长度，余弦等于临边长度比斜边长度，等等。这些规则在本书中广泛用到。

但是如果三角形没有 90° 的角会怎样呢？这时候余弦定理就派上用场了，它可以帮我们计算出三角形的各个角和边的值。这个定理有点复杂，但是如果你有了足够的三角形信息，就可以用公式计算出其他信息。但是这和反向运动学有什么关系呢？请看图 14-7。

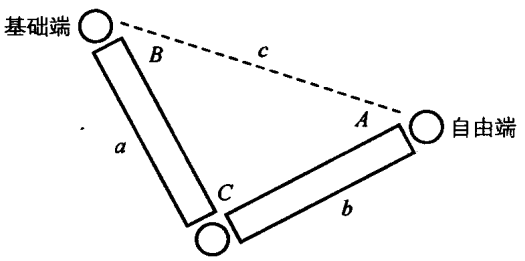


图 14-7 两个节段形成一个三角形，有 *a*、*b*、*c* 三条边，*A*、*B*、*C* 三个角

这里有两个节段。左边的那条（*a*）作为基础端，是固定的，因此它的位置已知。把另一条（*b*）放在自由端的位置。这样就形成一个三角形。

这个三角形的已知条件有哪些呢？你可以很容易地计算出两端的距离——*c* 边的长度。两个节段的长度已知——*a* 边和 *b* 边。所以，三边长度已知。

我们需要知道这个三角形的哪些信息呢？你只需要两个节段的角——∠*B* 和∠*C*。这就是余弦定理可以帮到你的地方：

$$c^2 = a^2 + b^2 - 2 \times a \times b \times \cos C$$

现在，还需要知道∠C，因此可以把它从公式中分离出来。这里就不列出每一步了，经过简单的代数运算，得到：

$$\angle C = \text{acos}\ ((a^2 + b^2 - c^2) / (2 \times a \times b))$$

acos 是反余弦，计算一个角的余弦能得到一个比值，计算这个比值的反余弦则能得到这个角。JavaScript 中对应的函数是 Math.acos。因为边 *a*、*b* 和 *c* 已知，所以可以计算得到∠*C*。类似，可以用下面的余弦定理得到∠*B*：

$$b^2 = a^2 + c^2 - 2 \times a \times c \times \cos B$$

化简后就能得到∠*B*：

$$\angle B = \text{acos}\ ((a^2 + c^2 - b^2) / (2 \times a \times c))$$

将这些公式转换为以下 Java Scripr 代码：

```
var B = Math.acos((a * a + c * c - b * b) / (2 * a * c)),
    C = Math.acos((a * a + b * b - c * c) / (2 * a * b));
```

现在你差不多得到了设置节段位置的所有信息。之所以说是差不多，是因为∠*B* 和∠*C* 并不是节段需要旋转的角度。请看图 14-8。

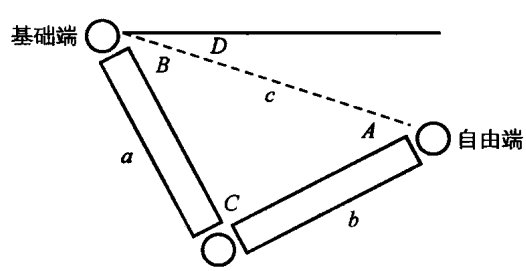


图 14-8 计算 *a* 边要旋转的角度

尽管知道了∠*B*，仍然需要确定 *a* 边（segment1）需要旋转的实际角度。这是指从 0 开始或者说从水平位置开始的角度。它是∠*D* 与∠*B* 之和。可以用 Math.atan2 和自由端与基础端的位置差计算得到∠*D*。接下来是 *b* 边（segment0）的旋转角度，∠*C* 已知，但这只是相对于 *a* 边的角度，所以用 *a* 边的旋转角度加上 180°，再加上∠*C*（见

图 14-9）得到一个∠*E*。

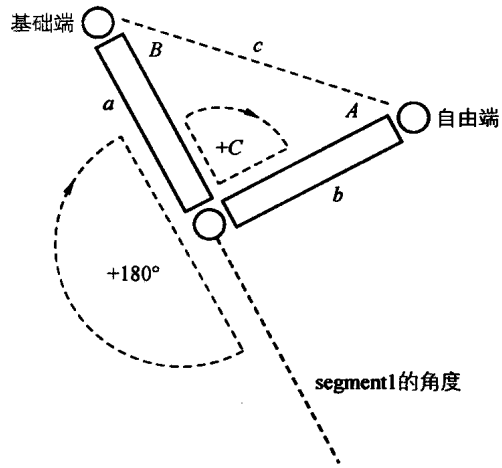


图 14-9 计算∠*E* 和 *b* 边的旋转角度

这个方法需要很多数学知识，但是让我们来看看如何用代码实现。这有助于更好地理解。

14.4.2 编程实现余弦定理

这里先给出下一个示例的完整代码（08-cosines-1.html），然后再一步一步地解释：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Cosines 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="segment.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        segment0 = new Segment(100, 20),
        segment1 = new Segment(100, 20);

    segment1.x = canvas.width / 2;
    segment1.y = canvas.height / 2;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
```

```
        context.clearRect(0, 0, canvas.width, canvas.height);

        var dx = mouse.x - segment1.x,
            dy = mouse.y - segment1.y,
            dist = Math.sqrt(dx * dx + dy * dy),
            a = 100,
            b = 100,
            c = Math.min(dist, a + b),
            B = Math.acos((b * b - a * a - c * c) / (-2 * a * c)),
            C = Math.acos((c * c - a * a - b * b) / (-2 * a * b)),
            D = Math.atan2(dy, dx),
            E = D + B + Math.PI + C;

        segment1.rotation = (D + B);

        var target = segment1.getPin();
        segment0.x = target.x;
        segment0.y = target.y;
        segment0.rotation = E;

        segment0.draw(context);
        segment1.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

下面给出操作步骤。

- 获得 segment1 到鼠标指针的距离。
- 获得三条边的长度。*a* 和 *b* 很简单，它们都是 100，因为这就是节段的长度。*c* 边等于 dist 或 *a*+*b* 中较小的一个。这是因为三角形的一条边不可能大于其余两边之和。
- 用余弦定理公式计算∠*B* 和∠*C*，用 Math.atan2 计算∠*D*。前面提到过∠*E* 等于∠*D*+∠*B*+180°+∠*C*。但是在代码中，要用弧度 Math.PI 代替 180°。
- 如图 14-8 所示，segment1 的旋转角度为∠*D*+∠*B*。再使用这个角度计算 segment1 的端点，然后把 segment0 放置过去。
- 最终，segment0 的旋转角度等于∠*E*。

这就是余弦定理在反向运动学中的使用。你可能已经注意到了，关节永远都向一个方向弯曲。如果你想建立一个肘部或膝盖的话，这可能很好。但是如果你想让它向另一个方向弯曲呢？

当需要计算这样的角度时，有两个方案：向一个方向弯曲，或者向另一个方向弯曲。前面的代码使用前者，使用∠*D* 加∠*B* 加∠*C* 来完成。如果把它们全部减去，也可以得到同样的效果，但是节段会向反向弯曲。下面用修改过的 **drawFrame** 函数来说明问题（文件 09-cosines-2.html）：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  var dx = mouse.x - segment1.x,
      dy = mouse.y - segment1.y,
```

```
        dist = Math.sqrt(dx * dx + dy * dy),
        a = 100,
        b = 100,
        c = Math.min(dist, a + b),
        B = Math.acos((b * b - a * a - c * c) / (-2 * a * c)),
        C = Math.acos((c * c - a * a - b * b) / (-2 * a * b)),
        D = Math.atan2(dy, dx),
        E = D - B + Math.PI - C;

    segment1.rotation = (D - B);

    var target = segment1.getPin();
    segment0.x = target.x;
    segment0.y = target.y;
    segment0.rotation = E;

    segment0.draw(context);
    segment1.draw(context);
}());
```

如果你想让它双向弯曲，只需要加入一些判断逻辑："如果它在某个位置，就向某个方向弯曲；否则，反向弯曲。"不过这些例子应该已经给你足够的知识来实现它了。

14.5 本章中的重要公式

对于标准的反向运动学，使用余弦定理公式。

14.5.1 余弦定理

$$a^2 = b^2 + c^2 - 2 \times b \times c \cos A$$
$$b^2 = a^2 + c^2 - 2 \times a \times c \cos B$$
$$c^2 = a^2 + b^2 - 2 \times a \times b \cos C$$

14.5.2 JavaScript 中的余弦定理

```
var A = Math.acos((b * b + c * c - a * a) / (2 * b * c)),
    B = Math.acos((a * a + c * c - b * b) / (2 * a * c)),
    C = Math.acos((a * a + b * b - c * c) / (2 * a * b));
```

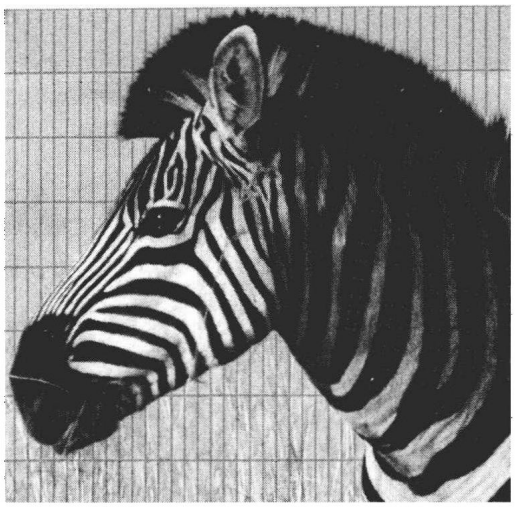
14.6 小结

反向运动学是一个广阔的主题——远不是一章就可以覆盖完全的。虽然如此，本章描述了一些非常有趣也非常有用的内容。你学到了如何建立一个反向运动学系统，包括两种方法：拖曳和伸出。我希望你可以用这些知识做出一些很有趣但不复杂的东西。反向运动学可以制作出很多效果，你应该已经准备好去探索和使用它了。

在第 15 章中，你将进入一个全新的维度，可以让你在动画中加入深度。

第四部分

3D 动画



第 15 章　三维基础

本章涵盖以下内容：

- 第三维度与透视图
- 速度与加速度
- 反弹
- 重力
- 屏幕环绕
- 缓动和弹动
- 坐标旋转
- 碰撞检测

到目前为止，本书中所有的内容都是二维的（有些只是一维），并且你已经创建了一些非常有趣的动画了。现在可以进入下一个级别了。

制作三维图形令人兴奋，因为新加入一个维度使物体看起来非常真实。我们很快地讲解一下三维编程基础，然后我们学习如何在三维世界中实现前面的章节中介绍的运动效果。具体地说，就是指速度、加速度、摩擦力、重力、反弹、屏幕环绕、缓动、弹动、坐标旋转以及碰撞检测。

首先，关注如何让一个物体在三维空间中运动，如何使用透视图来计算物体大小和在屏幕上的位置。物体绘制在屏幕上都是平面的，当然，你不会看到它的背面、侧面、顶部或底部。在接下来的几章里，你将会学习点、线、面以及三维实体。

值得一提的是，在本书写作的同时，canvas 元素的一个三维规范正在制定中，叫做 WebGL。但是，它并没有纳入 HTML5 规范，而且目前没有跨所有主流 Web 浏览器的支持（未来是否支

持也不知道）。WebGL 是一个底层规范，也就是说，直接在计算机的显卡上执行代码。WebGL 很强大，它支持在浏览器上运行硬件加速的图形。它是相当先进的，它使用 JavaScript 与叫做 shader 的程序配合，这种程序用另外一种 shader 语言编写，在网页加载时编译。WebGL 可以提供下一代更易使用的 Web 图形库以及三维引擎的基础。尽管它还面临着推广方面的很多问题，但是从它隐约可以看到基于 Web 的图形学以及游戏的未来。

15.1 第三维度与透视图

三维背后的主要概念就是存在一个除了 x、y 轴之外的维度。这个维度表示深度，它通常叫做 z。

canvas 元素没有内置的 z 维度，但是用 JavaScript 创建一个并不难。实际上，这比前面章节中的很多例子都简单！

15.1.1 z 轴

首先，必须确定 z 轴的方向是向内还是向外。如果你回想一下第 3 章中关于 canvas 坐标系的描述，就会发现它与通常的坐标系相反，y 轴方向不是向上，而是向下，角度以顺时针方向测量，而不是逆时针。

如果一个物体的 z 轴位置增加，它是接近你还是远离你呢？不能说哪种方法更正确。实际上，这个话题已经讨论了很久，甚至为这两种方法取了两个名字：左手系统和右手系统。

任意伸出一只手，把手指向 x 轴的正向，然后把手指向 y 轴的正向弯曲，大拇指的方向指向或离开你自己，这就是 z 轴在坐标系中的正方向。所以，如果你伸出的是右手，伸向 x 轴正方向，然后把手指弯向地面（y 轴正向），拇指会指向前方——右手坐标系的 z 轴正方向。在代码中，这意味着，z 轴的增加是远离观察者，减少是接近观察者，如图 15-1 所示。

如果用左手试试，会得到相反的结果——拇指指向你自己，如图 15-2 所示。

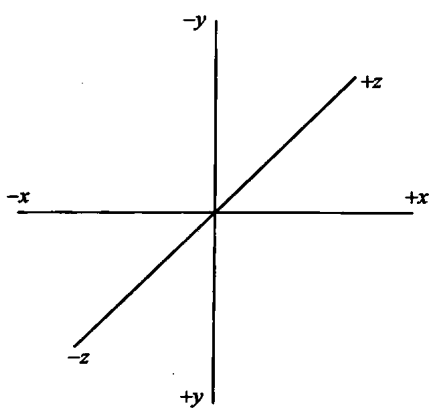


图 15-1 右手坐标系统

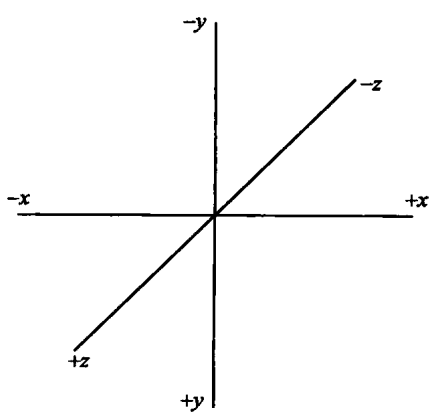


图 15-2 左手坐标系统

因为我们在接下来的例子中使用右手坐标系统（参照图 15-1），所以物体的 z 轴位置增加，它就向前远离我们。当然，这只是本书的偏好，使用左手坐标系也完全可以。

接下来的步骤是创建第三个维度（z）来说明如何模拟透视图。

15.1.2 透视图

透视图用来确定物体是接近还是远离我们。有很多技术可以用来表现透视图，但是这里我们只关心两种。

- 物体变小代表它远离。
- 远离的物体会汇聚在一个消失点上。

你可能见过铁轨消失在地平线上，那个点就是消失点。所以，当在 z 轴上移动物体时，要做两件事。

- 放大或缩小物体。
- 让它靠近或远离消失点。

在二维系统中，可以使用屏幕的 x、y 坐标作为物体的 x、y 坐标，因为它们是一一对应的。但是这在三维系统中行不通，因为两个物体可能有相同的 x、y 坐标，但是由于它们的深度不同，它们在屏幕上的位置就不一样。在三维系统中的任何一个物体都有自己的 x、y 和 z 坐标，这与屏幕上的坐标点没什么关系，这个坐标描述物体在虚拟空间内的位置。透视图的计算告诉我们应该把物体放在屏幕上的哪个位置。

透视图公式

基本思想是：随着物体的远离（z 坐标增加），它的大小缩小到 0，同时 x、y 坐标向消失点移动。因为缩放的比例和接近消失点的比例相同，所以只须根据距离计算出缩放比例，然后两个地方都能使用这个比例进行计算了。图 15-3 解释了这个概念。

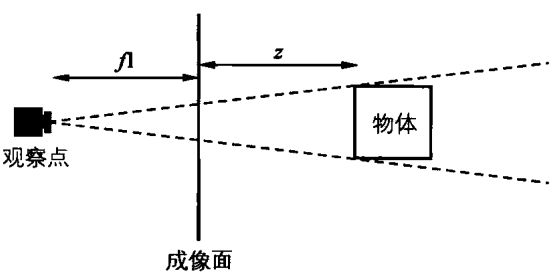


图 15-3 从侧观察透视图

在这里，有一个正在远离你的物体，一个观察点（相机）和一个成像面，也就是我们观察场景的地方（屏幕）。物体和成像面之间有一段距离，也就是 z 的值。观察点到成像面也有一段距离，这与照相机镜头的焦距相似，所以用变量 fl 表示。长焦距可以比作长焦镜头，它可以拉近远处的物体，但视野比较小。短焦距就像是广角镜头，视野很大，但是有些变形。中等的焦距类似人类的眼睛，fl 为 200～300 之间的值。下面是透视图公式：

```
scale = fl / (fl + z)
```

公式通常会产生 0.0～1.0 之间的一个值，这就是你用来做缩放和靠近消失点的比例。但是，

随着 z 坐标越来越接近 $-fl$，（$fl+z$）会趋近 0，*scale* 趋近无限大。这就像是戳进了眼睛里。

用这个比例来干嘛呢？你可以在绘制物体前调整 canvas 上下文中的缩放比例，以贯穿本书的 Ball 类为例，可以在它的 draw 方法中使用 scaleX 和 scaleY 属性，像这样：

```
context.scale(this.scaleX, this.scaleY);
```

当缩放比例确定后，用物体的 x 和 y 坐标乘以这个比例，得到新的 x 和 y 坐标。

让我们来看一个例子，我们使用 250 作为焦距。如果 z 是零，也就是说，物体正好在成像面上，缩放比例应该是 250/（250+0），也就是 1.0。这就是 scaleX 和 scaleY（别忘了，在缩放的时候，1.0 就是 100%）。用 1.0 乘以物体的 x 和 y 坐标，得到相同的结果。所以物体在屏幕上的位置没有发生变化。

现在向外移动它让 z 等于 250。缩放比例为 250/（250+250），也就是 scaleX 和 scaleY 为 0.5。同时要移动物体的屏幕位置，如果物体之前的位置为（200，300），它现在的位置应该是（100，150），也就是向消失点移动了一半（实际上，屏幕位置应该是相对于消失点的，你很快会看到）。

把 z 向外移动到 9750。缩放比例等于 250/10 000，也就是 scaleX 和 scaleY 等于 0.025。物体已经变成了一个很接近消失点的小点。

现在我们将上面讲的这些理论用代码来实现。

编写透视图程序

再次使用 Ball 类，我们使用鼠标和键盘来得到更炫的交互效果。鼠标控制小球的 x 和 y 位置，键盘的上下键控制小球的 z 坐标。变量 xpos、ypos 和 zpos 用来表示小球的三维位置。下面给出该示例的代码（01-perspective-1.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Perspective 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        ball = new Ball(),
        xpos = 0,
        ypos = 0,
        zpos = 0,
        fl = 250,
        vpX = canvas.width / 2,
        vpY = canvas.height / 2;
```

```
    window.addEventListener('keydown', function (event) {
      if (event.keyCode === 38) {        //up
        zpos += 5;
      } else if (event.keyCode === 40) {  //down
        zpos -= 5;
      }
    }, false);
    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var scale = fl / (fl + zpos);
      xpos = mouse.x - vpX;
      ypos = mouse.y - vpY;
      ball.scaleX = ball.scaleY = scale;
      ball.x = vpX + xpos * scale;
      ball.y = vpY + ypos * scale;

      ball.draw(context);
    }());
  };
  </script>
 </body>
</html>
```

首先，创建变量 xpos、ypos 和 zpos，还有 fl。然后，创建一个消失点（vpX，vpY）。别忘了，随着物体的远离，它们都会汇聚到（0，0）。如果不进行偏移，它们都会向屏幕左上角汇聚，这并不是你想要的效果。使用（vpX，vpY）指向屏幕中央，作为消失点。

接下来，添加 keydown 事件的监听器来改变 zpos 变量，如果按下向上的方向键则增加 zpos，如果按下向下的方向键则减小它。这用来控制物体远离或接近观察者。

在 drawFrame 动画循环中，把 xpos 和 ypos 设置为鼠标指针位置与消失点的相对位置。换句话说，如果鼠标指针在屏幕中央右边 200 个像素，xpos 就是 200，如果鼠标指针在屏幕中央左边 200 个像素，xpos 就是−200。

使用刚才学过的透视图公式计算 scale，相应地缩放和移动小球。小球在屏幕上的位置由消失点加上 xpos 和 ypos 再乘以 scale 计算得来。这样，随着 scale 的减小，小球越来越接近消失点。

在浏览器中运行本例，一开始看起来像是用鼠标指针在拖动小球。这是因为 zpos 是零，导致 scale 为 1.0，所以，注意不到透视图的存在。当按向上的方向键时，小球好像在远去，如图 15-4 所示。现在当你移动鼠标指针时，小球随之移动，但是范围会小一些，产生视差效果。如果你想设置一个视差效果的场景，可以硬编码 z 的值来实现。

图 15-4 透视图在其作用

你可能注意到，如果你按住向下的方向键不放，小球会变得非常大。这是对的，如果你把一个小石子靠近眼睛，它会看起来像是一块巨石。但是如果继续按住向下的方式键，你会看到

小球增长到无限大，然后又变小了（颠倒了，但是很难解释）。小球跑到了观察点的后面去了。

理论上，其实就是当 zpos 等于-fl 时，公式 scale = fl / (fl +zpos)变成了 scale=fl/0。在很多语言里，除以零会导致错误。但在 JavaScript 里，会得到 Infinity。随着 zpos 的减小，用 fl 除以一个负数，scale 也就变成了负数，把它应用到 canvas 的上下文中，会导致图像坐标系翻转，所以你看到小球变大然后变小。

解决这个问题有个很简单的方法，当小球越过某个点时就让它消失。如果 zpos 小于或等于-fl，就会有问题，所以可以检测这个点。下面是更新过的 drawFrame 函数（其余代码与上例相同，文件 02-perspective-2.html）：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);
  if (zpos > -fl) {
    var scale = fl / (fl + zpos);
    xpos = mouse.x - vpX;
    ypos = mouse.y - vpY;
    ball.scaleX = ball.scaleY = scale;
    ball.x = vpX + xpos * scale;
    ball.y = vpY + ypos * scale;
    ball.visible = true;
  } else {
    ball.visible = false;
  }
  if (ball.visible) {
    ball.draw(context);
  }
}());
```

如果小球不在我们前面了，就不需要再计算比例和它的位置了。同样，如果不必显示小球了，也就不需要在 canvas 上绘制它。

现在你已经学会了最简单的三维基础。并不是很难吧？一定要多调试几次这个例子，并理解它。尤其是改变 fl 观察效果的变化，这就相当于相机的变焦镜头。

本章其余的部分都是编写前面章节中的各种动画效果，不过这次是三维的。

15.2 速度与加速度

在三维系统中实现速度与加速度出奇得简单。在二维系统中，用 vx 和 vy 变量来表示两条轴上的速度，现在只须再添加一个变量 vz 来表示第三个维度上的速度。如果使用 ax 和 ay 来表示加速度，则再添加一个 az。

你可以把最后一个练习修改为三维版本的宇宙飞船示例。它是由键盘控制的，用方向键来提供 *x* 轴和 *y* 轴上的推进力，Shift 和 Control 键用于 *z* 轴上的推进。下面是代码（文件 03-velocity-3d.html）：

```
<!doctype html>
<html>
```

```
<head>
 <meta charset="utf-8">
 <title>Velocity 3D</title>
 <link rel="stylesheet" href="style.css">
</head>
<body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <script src="utils.js"></script>
 <script src="ball.js"></script>
 <script>
 window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),
       ball = new Ball(),
       xpos = 0,
       ypos = 0,
       zpos = 0,
       vx = 0,
       vy = 0,
       vz = 0,
       friction = 0.98,
       fl = 250,
       vpX = canvas.width / 2,
       vpY = canvas.height / 2;

  window.addEventListener('keydown', function (event) {
    switch (event.keyCode) {
    case 38:        //up
       vy -= 1;
       break;
    case 40:        //down
       vy += 1;
       break;
    case 37:        //left
       vx -= 1;
       break;
    case 39:        //right
       vx += 1;
       break;
    case 16:        //shift
       vz += 1;
       break;
    case 17:        //control
       vz -= 1;
       break;
    }
  }, false);

  (function drawFrame () {
     window.requestAnimationFrame(drawFrame, canvas);
     context.clearRect(0, 0, canvas.width, canvas.height);

     xpos += vx;
     ypos += vy;
```

```
        zpos += vz;
        vx *= friction;
        vy *= friction;
        vz *= friction;

        if (zpos > -fl) {
          var scale = fl / (fl + zpos);
          ball.scaleX = ball.scaleY = scale;
          ball.x = vpX + xpos * scale;
          ball.y = vpY + ypos * scale;
          ball.visible = true;
        } else {
          ball.visible = false;
        }

        if (ball.visible) {
          ball.draw(context);
        }
      }());

  };
  </script>
 </body>
</html>
```

我们加入了每条轴上的速度变量以及一些摩擦力。当按下 6 个键中的一个时，加上或减去适当的速度（别忘了加速度改变速度）。然后把速度累加在每条轴的位置上，以及计算摩擦力。

现在你有了一个具有加速度、速度和摩擦力的三维物体，很简单吧。

15.3 反弹

在本节中，我们让一个小球从三维空间中的平面（换句说话，与 x、y 或 z 轴完全平行的平面）上反弹。这与我们在前面做过的类似：在二维空间中，我们让小球从 canvas 元素的边界上反弹。

15.3.1 单物体反弹

在三维系统中反弹，同样需要检测物体超出边界的时机，把它的位置调整到边界上，然后在相应的轴上反转它的速度。有一点不同的是，在三维系统中，要如何判断边界的位置。在二维系统中，通常使用 canvas 或其他可见的矩形区域作为边界。在三维系统中，就没那么简单了，因为并没有什么可见边界的概念，除非你画一个。在第 16 章中我们将绘制边界，但是现在先让物体从一个随意放置的隐形墙上反弹。

边界的设置和以前一样，不过现在要把它们放置在三维空间中，这意味着，它们可以是正的也可以是负的。边界也可以设置在 z 轴上。下一个例子中的边界是这样的：

```
var top = -100,
    bottom = 100,
    left = -100,
```

```
    right = 100,
    front = -100,
    back = 100;
```

在确定了物体的新位置后，判断它是否和6个边界相交。别忘了，你是用物体宽度的一半来计算碰撞的，这个值存在Ball类的radius属性里。下面是三维反弹的完整代码（文件04-bounce-3d.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Bounce 3d</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        xpos = 0,
        ypos = 0,
        zpos = 0,
        vx = Math.random() * 10 - 5,
        vy = Math.random() * 10 - 5,
        vz = Math.random() * 10 - 5,
        fl = 250,
        vpX = canvas.width / 2,
        vpY = canvas.height / 2,
        top = -100,
        bottom = 100,
        left = -100,
        right = 100,
        front = -100,
        back = 100;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      xpos += vx;
      ypos += vy;
      zpos += vz;

      //check boundaries
      if (xpos + ball.radius > right) {
          xpos = right - ball.radius;
          vx *= -1;
      } else if (xpos - ball.radius < left) {
          xpos = left + ball.radius;
```

```
            vx *= -1;
        }
        if (ypos + ball.radius > bottom) {
            ypos = bottom - ball.radius;
            vy *= -1;
        } else if (ypos - ball.radius < top) {
            ypos = top + ball.radius;
            vy *= -1;
        }
        if (zpos + ball.radius > back) {
           zpos = back - ball.radius;
           vz *= -1;
        } else if (zpos - ball.radius < front) {
           zpos = front + ball.radius;
           vz *= -1;
        }

        if (zpos > -fl) {
            var scale = fl / (fl + zpos);
            ball.scaleX = ball.scaleY = scale;
            ball.x = vpX + xpos * scale;
            ball.y = vpY + ypos * scale;
            ball.visible = true;
        } else {
            ball.visible = false;
        }
        if (ball.visible) {
            ball.draw(context);
        }
    }());
  };
  </script>
 </body>
</html>
```

上例中所有处理按键的代码都已经删除了，小球在每条轴上的速度随机。现在你可以看到小球如我们所愿地弹来弹去，但是无法辨认它在什么东西上反弹——别忘了，我们只是随意放了不可见的边界。

15.3.2 多物体反弹

有一个方法可以让这些墙的可见性稍微好一点，就是在空间内填充更多的物体。这样就需要 Ball 的多个实例，但是每个实例要有自己的 xpos、ypos、zpos 以及每条轴上的速度。如果在主干代码里跟踪管理这些值，就会变得很混乱，所以创建一个新类 Ball3D，用来跟踪这些值。把下面的脚本保存到 ball3d.js 文件中，我们将在下一个例子中导入它：

```
function Ball3d (radius, color) {
  if (radius === undefined) { radius = 40; }
  if (color === undefined) { color = "#ff0000"; }
  this.x = 0;
  this.y = 0;
```

```
    this.xpos = 0;
    this.ypos = 0;
    this.zpos = 0;
    this.vx = 0;
    this.vy = 0;
    this.vz = 0;
    this.radius = radius;
    this.mass = 1;
    this.rotation = 0;
    this.scaleX = 1;
    this.scaleY = 1;
    this.color = utils.parseColor(color);
    this.lineWidth = 1;
    this.visible = true;
}

Ball3d.prototype.draw = function (context) {
    context.save();
    context.translate(this.x, this.y);
    context.rotate(this.rotation);
    context.scale(this.scaleX, this.scaleY);
    context.lineWidth = this.lineWidth;
    context.fillStyle = this.color;
    context.beginPath();
    context.arc(0, 0, this.radius, 0, (Math.PI * 2), true);
    context.closePath();
    context.fill();
    if (this.lineWidth > 0) {
        context.stroke();
    }
    context.restore();
};
```

你可以看到，这里所做的就是为每条轴上的位置和速度添加了对应的属性。在05-multi-bounce-3d.html中，创建了50个实例，每个实例都有不同的xpos、ypos和zpos以及vx、vy和vz值。drawFrame函数遍历这些实例两次，分别把每一个对象传入move和draw函数。这两个函数与上一个例子中的功能相同。下面是完整的代码（05-multi-bounce-3d.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Multi Bounce 3d</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball3d.js"></script>
  <script>
  window.onload = function () {
     var canvas = document.getElementById('canvas'),
         context = canvas.getContext('2d'),
         balls = [],
```

```
    numBalls = 50,
    fl = 250,
    vpX = canvas.width / 2,
    vpY = canvas.height / 2,
    top = -100,
    bottom = 100,
    left = -100,
    right = 100,
    front = -100,
    back = 100;

for (var ball, i = 0; i < numBalls; i++) {
  ball = new Ball3d(15);
  ball.vx = Math.random() * 10 - 5;
  ball.vy = Math.random() * 10 - 5;
  ball.vz = Math.random() * 10 - 5;
  balls.push(ball);
}

function move (ball) {
  ball.xpos += ball.vx;
  ball.ypos += ball.vy;
  ball.zpos += ball.vz;

  //check boundaries
  if (ball.xpos + ball.radius > right) {
     ball.xpos = right - ball.radius;
     ball.vx *= -1;
  } else if (ball.xpos - ball.radius < left) {
     ball.xpos = left + ball.radius;
     ball.vx *= -1;
  }
  if (ball.ypos + ball.radius > bottom) {
     ball.ypos = bottom - ball.radius;
     ball.vy *= -1;
  } else if (ball.ypos - ball.radius < top) {
     ball.ypos = top + ball.radius;
     ball.vy *= -1;
  }
  if (ball.zpos + ball.radius > back) {
     ball.zpos = back - ball.radius;
     ball.vz *= -1;
  } else if (ball.zpos - ball.radius < front) {
     ball.zpos = front + ball.radius;
     ball.vz *= -1;
  }

  if (ball.zpos > -fl) {
     var scale = fl / (fl + ball.zpos);
     ball.scaleX = ball.scaleY = scale;
     ball.x = vpX + ball.xpos * scale;
     ball.y = vpY + ball.ypos * scale;
     ball.visible = true;
  } else {
```

```
            ball.visible = false;
        }
      }
      function draw (ball) {
        if (ball.visible) {
            ball.draw(context);
        }
      }

      (function drawFrame () {
        window.requestAnimationFrame(drawFrame, canvas);
         context.clearRect(0, 0, canvas.width, canvas.height);

         balls.forEach(move);
         balls.forEach(draw);
      }());
    };
    </script>
  </body>
</html>
```

在运行脚本后，你会发现这些小球填满了这 6 个边界内的空间，如图 15-5 所示。这样你就能感知到这个空间的形状了。

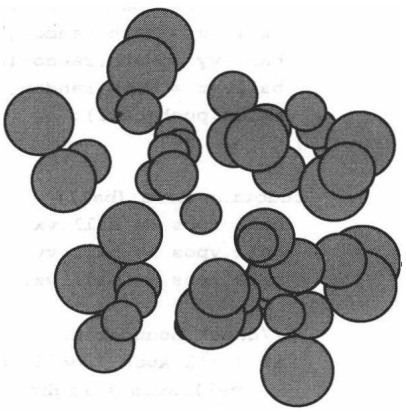


图 15-5 小球在三维空间内反弹

15.3.3 Z 排序

在添加了多个物体后带来了代码中没有考虑到的一个问题，叫做 Z 排序。Z 排序是指物体在 z 轴上如何排序，或者表示物体哪个在前，哪个在后。如果你仔细观察小球的轮廓，你就能发现哪个小球在另一个上面。现在是小的物体出现在大的物体前面，这几乎毁了三维的效果。但是我们可以使用 Z 排序来解决这个问题。

代码以小球在 balls 数组里的顺序来绘制它们。在前一个例子中，这个顺序是初始化小球并加入数组的顺序。所以，如果我们想依照小球在 z 轴上的位置来绘制它们，我们就需要重新排序。需要定义一个函数来在 JavaScript 中对数组重新排序。

JavaScript 数组有一个排序方法 Array.sort([compareFunction])。这个方法接收一个可选参数，这个参数是个函数，它定义元素之间的比较规则。如果省略这个参数，数组则按照元素名字的字母顺序排序。

参数 compareFunction 是个函数，它接受两个参数，把它们比作对整个集合进行循环遍历时的数组元素。数组如何排序取决于传入后续元素后 compareFunction 的返回值，应用如下规则：

令 n = compareFunction(a, b),

- 如果 n 小于 0，把元素 a 排在元素 b 的前面（a 的索引小）。
- 如果 n 等于 0，则元素 a 和元素 b 的顺序不变。
- 如果 n 大于 0，则把元素 b 排在元素 a 的前面。

举个例子，要按升序排列一组数字。

```
var arr = [3, 5, 1, 4, 2];
arr.sort(function (a, b) { return (a - b); });

console.log(arr); //prints [1, 2, 3, 4, 5]
```

根据三维的深度顺序，索引为 0 就表示在底部，索引更高的任何物体都会出现在比它的索引低的物体上面。我们想把数组中的物体从深到浅（从远到近）排序。下面的代码定义了排序函数，然后对 balls 数组进行排序。

```
function zSort (a, b) {
  return (b.zpos - a.zpos);
}
balls.sort(zSort);
```

这个排序基于每个元素的 zpos 属性，依照数字的反序排列，换句话说，就是从高到低。得到的结果是，最远的物体（zpos 值最大）处于数组中首位，因此第一个被绘制在 canvas 上。最近的小球是数组中的最后一个元素，它被绘制在所有其他小球之上。

当把这个函数加入到上一个例子中时，需要在绘制小球之前调用它。

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  balls.forEach(move);
  balls.sort(zSort);
  balls.forEach(draw);
}());
```

其余的代码和上一个例子中的相同。完整的代码可以在文件 06-z-sort.html 中找到。

15.4 重力

在本节中，我们来看看简单的重力，就和第 5 章中介绍的地球表面上的重力一样。这种情况下，三维系统中的重力与二维系统中的一样。选择一个数字作为施加在物体上的重力值，然后在每一帧中把这个值加在物体的 y 速度上。

三维系统中的重力也许很简单，但是简单的东西也可以创造出极好的视觉效果。在本例中，把重力和反弹组合，创建出一桶橡胶弹球倒在地板上的效果。

需要一个对象来表示单个橡胶弹球，因此继续使用 Ball3d 类，但是把半径调小一些。为每个小球设置随机的颜色，同时把背景颜色设置为黑色。下面是该练习的代码（07-bouncy-balls.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Fireworks</title>
  <link rel="stylesheet" href="style.css">
  <style>
  #canvas {
    background-color: #000000;
```

```
}
</style>
</head>
<body>
<canvas id="canvas" width="400" height="400"></canvas>
<script src="utils.js"></script>
<script src="ball3d.js"></script>
<script>
window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      balls = [],
      numBalls = 100,
      fl = 250,
      vpX = canvas.width / 2,
      vpY = canvas.height / 2,
      gravity = 0.2,
      floor = 200,
      bounce = -0.6;

  for (var ball, i = 0; i < numBalls; i++) {
    ball = new Ball3d(3, Math.random() * 0xffffff);
    ball.ypos = -100;
    ball.vx = Math.random() * 6 - 3;
    ball.vy = Math.random() * 6 - 3;
    ball.vz = Math.random() * 6 - 3;
    balls.push(ball);
  }

  function move (ball) {
    ball.vy += gravity;
    ball.xpos += ball.vx;
    ball.ypos += ball.vy;
    ball.zpos += ball.vz;

    if (ball.ypos > floor) {
      ball.ypos = floor;
      ball.vy *= bounce;
    }

    if (ball.zpos > -fl) {
      var scale = fl / (fl + ball.zpos);
      ball.scaleX = ball.scaleY = scale;
      ball.x = vpX + ball.xpos * scale;
      ball.y = vpY + ball.ypos * scale;
      ball.visible = true;
    } else {
      ball.visible = false;
    }
  }

  function zSort (a, b) {
    return (b.zpos - a.zpos);
  }
```

```
    function draw (ball) {
      if (ball.visible) {
        ball.draw(context);
      }
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      balls.forEach(move);
      balls.sort(zSort);
      balls.forEach(draw);
    }());
  };
  </script>
 </body>
</html>
```

这里添加了几个属性：gravity、bounce 和 floor。前两个属性你应该已经见过了，floor 就是小球反弹之前 y 坐标能到达的最小值。

除了把重力加到每个小球的 vy 属性上之外，当每个小球接触到地面时加入反弹效果，这里没什么陌生的内容，但是效果变得相当漂亮了。

运行效果如图 15-6 所示。

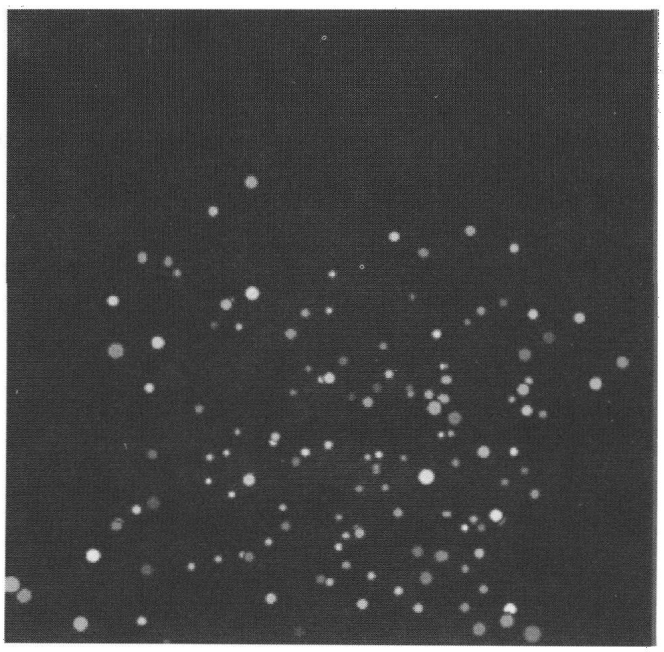


图 15-6 一堆橡皮弹球！（相信我，运行效果更好）

15.5 屏幕环绕

回忆一下，第 6 章讨论过物体碰到边界时有三种可能的反应。除了介绍过的反弹，还有屏幕环绕和再生。对于三维系统而言，屏幕环绕很有用，但主要是在 z 轴上。

在二维屏幕环绕中，在 x 轴或 y 轴上检测物体是否超出屏幕边界。这种做法很正确，因为当物体超出任意一个边界时，我们就看不到它了，所以你可以很容易地重置物体的位置，并不会引起观察者的注意。但是在三维系统中就没那么简单了。

在三维的屏幕环绕中，只有两个点可以放心地移除和重置物体。一个是当物体运动到观察点后面时，前面的例子中就是在这个时机把物体隐藏。另一个点是当物体离得足够远并且已经变成一个很小的点时，这意味着，可以放心地在 z 轴上环绕。当物体运动到你后面时，就把它抛到前面，然后让它再次向你运动。或者，如果一个物体远离你，当它远到看不清楚时，移除它并且把它重置到你身后。如果你想，也可以尝试在 x 轴或 y 轴上环绕，但是在大多数情况下，这会出现一些不自然的忽隐忽现效果。

值得一提的是，如果物体在 x 轴或 y 轴上超出了视野，就可以把它沿着 z 轴重置。这并不是真的屏幕环绕，但是可以重用对象，使得程序比较高效。

z 轴环绕很有用，特别是三维赛车类的游戏中，这里将制作一个很简单的示例。主体思想是在观察点前方放置多个三维物体，然后把这些物体向观察点移动。换句话说，给它们负的 z 速度。根据设置的不同，效果可以是很多物体向你移动，也可以是欺骗你的眼睛，让你感觉自己在向这些物体移动。一旦物体移动到观察点后方，就把它重置到远方。这样就能源源不断地提供向后移动的对象了。

下一个例子中用到的物体很简单，是一些有随机树枝的树。使用 Tree 类，它有三维位置的属性，同时绘制一些短的线条来表示一棵树。代码如下（文件 tree.js）：

```
function Tree () {
  this.x = 0;
  this.y = 0;
  this.xpos = 0;
  this.ypos = 0;
  this.zpos = 0;
  this.scaleX = 1;
  this.scaleY = 1;
  this.color = "#ffffff";
  this.alpha = 1;
  this.lineWidth = 1;
  this.branch = [];

  //generate some random branch positions
  this.branch[0] = -140 - Math.random() * 20;
  this.branch[1] = -30 - Math.random() * 30;
  this.branch[2] = Math.random() * 80 - 40;
  this.branch[3] = -100 - Math.random() * 40;
  this.branch[4] = -60 - Math.random() * 40;
```

```
  this.branch[5] = Math.random() * 60 - 30;
  this.branch[6] = -110 - Math.random() * 20;
}

Tree.prototype.draw = function (context) {
  context.save();
  context.translate(this.x, this.y);
  context.scale(this.scaleX, this.scaleY);
  context.lineWidth = this.lineWidth;
  context.strokeStyle = utils.colorToRGB(this.color, this.alpha);
  context.beginPath();
  context.moveTo(0, 0);
  context.lineTo(0, this.branch[0]);
  context.moveTo(0, this.branch[1]);
  context.lineTo(this.branch[2], this.branch[3]);
  context.moveTo(0, this.branch[4]);
  context.lineTo(this.branch[5], this.branch[6]);
  context.stroke();
  context.restore();
};
```

在本例中，同样使用黑色背景。在 x 轴上−1 000～1 000，z 轴上 0～10 000 的范围内，随机创建一些树。它们的 y 坐标相同，都等于 floor 属性，给人一种树长在地面上的感觉。

下面是该练习的代码（文件 08-trees-1.html），运行效果如图 15-7 所示。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Trees 1</title>
  <link rel="stylesheet" href="style.css">
  <style>
  #canvas {
    background-color: #000000;
  }
  </style>
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="tree.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        trees = [],
        numTrees = 100,
        fl = 250,
        vpX = canvas.width / 2,
        vpY = canvas.height / 2,
        floor = 200,
        vz = 0,
        friction = 0.98;

    for (var tree, i = 0; i < numTrees; i++) {
```

```
    tree = new Tree();
    tree.xpos = Math.random() * 2000 - 1000;
    tree.ypos = floor;
    tree.zpos = Math.random() * 10000;
    trees.push(tree);
  }

  window.addEventListener('keydown', function (event) {
    if (event.keyCode === 38) {        //up
      vz -= 1;
    } else if (event.keyCode === 40) {  //down
      vz += 1;
    }
  }, false);

  function move (tree) {
    tree.zpos += vz;
    if (tree.zpos < -fl) {
      tree.zpos += 10000;
    }
    if (tree.zpos > 10000 - fl) {
      tree.zpos -= 10000;
    }

  var scale = fl / (fl + tree.zpos);
    tree.scaleX = tree.scaleY = scale;
    tree.x = vpX + tree.xpos * scale;
    tree.y = vpY + tree.ypos * scale;
    tree.alpha = scale;
  }

  function zSort (a, b) {
    return (b.zpos - a.zpos);
  }

  function draw (tree) {
    tree.draw(context);
  }

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    trees.forEach(move);
    vz *= friction;
    trees.sort(zSort);
    trees.forEach(draw);
  }());
 };
 </script>
</body>
</html>
```

只有一个 z 速度变量，因为树不需要在 x 轴或 y 轴上移动，所有的运动都在 z 轴上。我们添加了 keydown 事件监听器，用来相应地增加或减小 vz。在 drawFrame 中应用一点摩擦力，

防止速度无限增大，并且在放开按键时把速度降下来。

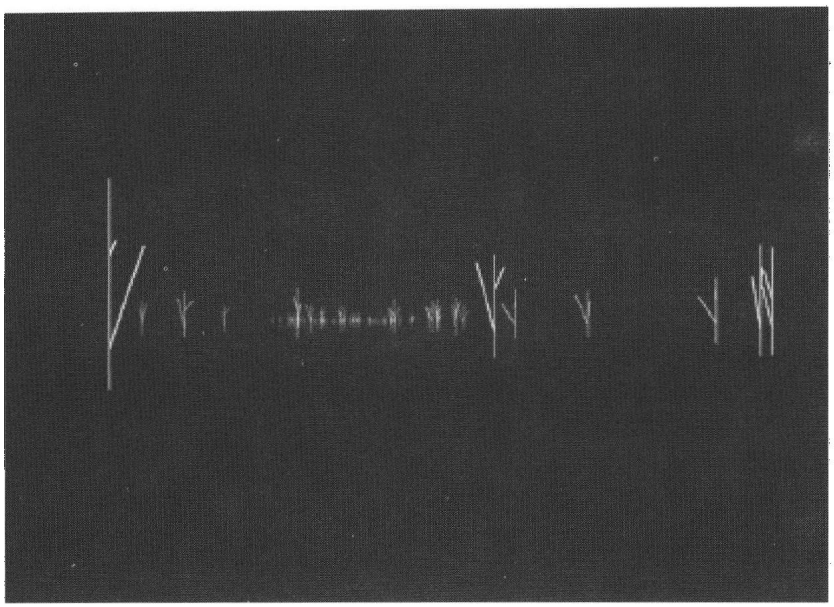


图 15-7 小心树

代码循环每棵树，用当前的 z 速度更新它的 z 坐标。然后它检测树的位置，如果它移动到了你身后，不要让它消失，而是把它在 z 轴上向内移动 10 000 个像素。同样，如果它越过了 10 000-fl 的位置，就把它向外移动 10 000 个像素。

然后执行标准的透视图计算。这里还加入了一些额外的东西来加强立体感：

```
tree.alpha = scale;
```

根据树在 z 轴上的深度来设置它的透明度。离得越远，颜色越淡。这就是大气透视图，用来模拟物体和观察者之间的空气效果。这用来表现物体远离的效果非常合适，比如在本例中。这个效果体现在代码中就是通过在树的 draw 方法中设置 canvas 的上下文来实现。

```
context.strokeStyle = utils.colorToRGB(this.color, this.alpha);
```

这个特别的透明度计算得到了一个漆黑、幽灵般的黑夜效果。你可能想试试下面的代码：

```
tree.alpha = scale * 0.7 + 0.3;
```

这让树的可见度最小为 30%——看起来不那么朦胧。这里的大部分值没有对与错，只是不同的值会得到不同的效果。

尽管在这个特殊的例子中，因为这些树只是一些颜色单一的简单线条，所以 Z 排序没有起到太大作用，我们还是把它留下。如果你绘制一些更复杂的、重叠的物体，它就会变得很重要。

让我们加入一些超出屏幕环绕之外的增强功能，只是为了给你一些启发——我们还可以做什么。下面是代码（可以在文件 09-trees-2.html 中找到）：

```
<!doctype html>
<html>
```

```
<head>
 <meta charset="utf-8">
 <title>Trees 2</title>
 <link rel="stylesheet" href="style.css">
 <style>
 #canvas {
   background-color: #000000;
 }
 </style>
</head>
<body>
 <canvas id="canvas" width="400" height="400"></canvas>
 <script src="utils.js"></script>
 <script src="tree.js"></script>
 <script>
 window.onload = function () {
   var canvas = document.getElementById('canvas'),
       context = canvas.getContext('2d'),

       trees = [],
       numTrees = 100,
       fl = 250,
       vpX = canvas.width / 2,
       vpY = canvas.height / 2,
       floor = 50,
       ax = 0,
       ay = 0,
       az = 0,
       vx = 0,
       vy = 0,
       vz = 0,
       gravity = 0.3,
       friction = 0.95;

   for (var tree, i = 0; i < numTrees; i++) {
       tree = new Tree();
       tree.xpos = Math.random() * 2000 - 1000;
       tree.ypos = floor;
       tree.zpos = Math.random() * 10000;
       trees.push(tree);
   }

   window.addEventListener('keydown', function (event) {
     switch (event.keyCode) {
     case 38:       //up
       az = -1;
       break;
     case 40:       //down
       az = 1;
       break;
     case 37:       //left
       ax = 1;
       break;
     case 39:       //right
```

```
      ax = -1;
      break;
  case 32:        //space
      ay = 1;
      break;
  }
}, false);

window.addEventListener('keyup', function (event) {
  switch (event.keyCode) {
  case 38:        //up
  case 40:        //down
     az = 0;
     break;
  case 37:        //left
  case 39:        //right
      ax = 0;
      break;
 case 32:        //space
      ay = 0;
      break;
  }
}, false);

function move (tree) {
  tree.xpos += vx;
  tree.ypos += vy;
  tree.zpos += vz;
  if (tree.ypos < floor) {
  tree.ypos = floor;
  }
  if (tree.zpos < -fl) {
    tree.zpos += 10000;
  }
  if (tree.zpos > 10000 - fl) {
    tree.zpos -= 10000;
  }
  var scale = fl / (fl + tree.zpos);
  tree.scaleX = tree.scaleY = scale;
  tree.x = vpX + tree.xpos * scale;
  tree.y = vpY + tree.ypos * scale;
  tree.alpha = scale;
}

function zSort (a, b) {
  return (b.zpos - a.zpos);
}

function draw (tree) {
  tree.draw(context);
}

(function drawFrame () {
   window.requestAnimationFrame(drawFrame, canvas);
   context.clearRect(0, 0, canvas.width, canvas.height);
```

```
        vx += ax;
        vy += ay;
        vz += az;
        vy -= gravity;
        trees.forEach(move);
        vx *= friction;
        vy *= friction;
        vz *= friction;
        trees.sort(zSort);
        trees.forEach(draw);
      }());
    };
    </script>
  </body>
</html>
```

这里加入了 x 轴和 y 轴上的速度，还有一些重力。同时也添加了一个 keydown 和 keyup 事件监听器，来实现检测多按键按下的躲闪功能。当按下按键时，把相应轴上的加速度设置为 1 或–1，然后当放开按键时，把加速度设置为 0。在 drawFrame 函数中，把每条轴上的加速度累加到速度上。左右按键很明显是用来控制 x 速度的，用空格键来控制 y 速度。有趣的一点是从 vy 中减去重力。这是因为我们想让它看起来像是观察者落到树林中的效果，如图 15-8 所示。实际上，这些树是向上移动，朝向观察点的位置，但是把树的 y 坐标最大值限定为 50，这看起来就像你在向下落到地上。

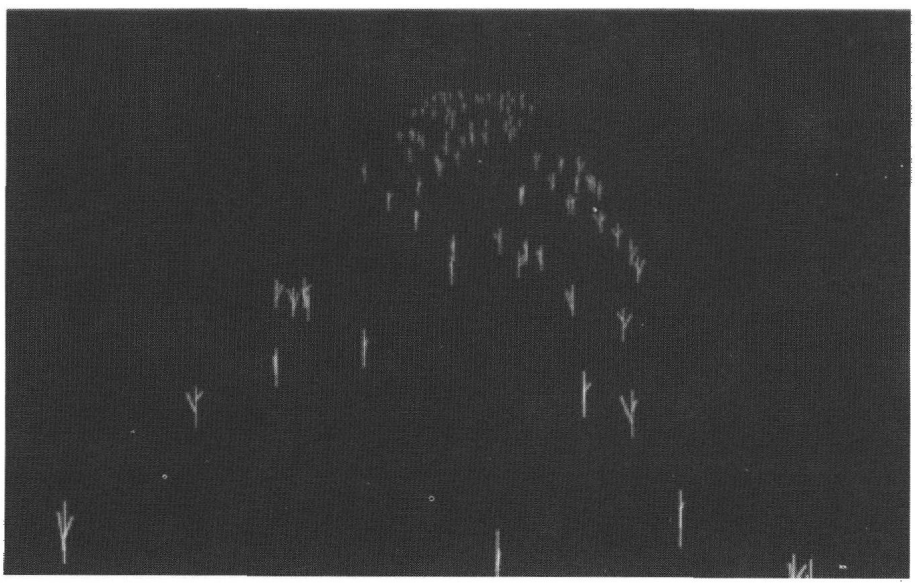


图 15-8 看，我在飞

这里没有对 x 轴上的移动做任何限制，这意味着，你可以前往森林的边缘。你可以尝试加入边界作为附加练习，这并不是太难。

15.6 缓动与弹动

三维系统中的缓动与弹动也并不比二维中（第 8 章的主题）的复杂多少。只需再为 z 轴添加一两个变量就行了。

15.6.1 缓动

这里介绍缓动的内容并不多。在二维系统中，使用 tx 和 ty 作为目标点，现在只需为 z 轴添加 tz。在每一帧动画中，计算每条轴上物体到目标点的距离，并把物体向目标点移动一小段距离。

让我们来看一个简单的例子，它把一个三维小球缓动向三维空间中的一个随机目标点。当小球到达目标点时，它重新选择一个目标并移动过去。下面是代码（文件 10-easing-3d.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Easing 3d</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball3d.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball3d(),
        tx = Math.random() * 500 - 250,
        ty = Math.random() * 500 - 250,
        tz = Math.random() * 500,
        easing = 0.1,
        fl = 250,
        vpX = canvas.width / 2,
        vpY = canvas.height / 2;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      var dx = tx - ball.xpos,
          dy = ty - ball.ypos,
          dz = tz - ball.zpos,
          dist = Math.sqrt(dx * dx + dy * dy + dz * dz);

      ball.xpos += dx * easing;
```

```
        ball.ypos += dy * easing;
        ball.zpos += dz * easing;

        if (dist < 1) {
            tx = Math.random() * 500 - 250;
            ty = Math.random() * 500 - 250;
            tz = Math.random() * 500;
        }

        if (ball.zpos > -fl) {
            var scale = fl / (fl + ball.zpos);
            ball.scaleX = ball.scaleY = scale;
            ball.x = vpX + ball.xpos * scale;
            ball.y = vpY + ball.ypos * scale;
            ball.visible = true;
         } else {
            ball.visible = false;
        }

        if (ball.visible) {
            ball.draw(context);
        }
      }());
    };
    </script>
  </body>
</html>
```

在三维空间中计算距离的这段代码最有趣：

```
var dist = Math.sqrt(dx * dx + dy * dy + dz * dz);
```

你应该还记得在二维系统中，我们用下面的方法计算两点间的距离：

```
var dist = Math.sqrt(dx * dx + dy * dy);
```

在三维系统中，只需要加入 z 轴上距离的平方，非常简单。

15.6.2 弹动

弹动是缓动的表兄弟，可以用类似的调整来让前者支持三维系统。需根据物体与目标的距离来改变速度，而不是改变位置。在本例（文件 11-spring-3d.html）中，每次单击鼠标都为小球创建一个新的随机目标点：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Spring 3d</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball3d.js"></script>
  <script>
```

```
window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      ball = new Ball3d(),
      tx = Math.random() * 500 - 250,
      ty = Math.random() * 500 - 250,
      tz = Math.random() * 500,
      spring = 0.1,
      friction = 0.94,
      fl = 250,
      vpX = canvas.width / 2,
      vpY = canvas.height / 2;

  window.addEventListener('mousedown', function () {
    tx = Math.random() * 500 - 250;
    ty = Math.random() * 500 - 250;
    tz = Math.random() * 500;
  }, false);

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    var dx = tx - ball.xpos,
        dy = ty - ball.ypos,
        dz = tz - ball.zpos;

    ball.vx += dx * spring;
    ball.vy += dy * spring;
    ball.vz += dz * spring;
    ball.xpos += ball.vx;
    ball.ypos += ball.vy;
    ball.zpos += ball.vz;
    ball.vx *= friction;
    ball.vy *= friction;
    ball.vz *= friction;

    if (ball.zpos > -fl) {
      var scale = fl / (fl + ball.zpos);
      ball.scaleX = ball.scaleY = scale;
      ball.x = vpX + ball.xpos * scale;
      ball.y = vpY + ball.ypos * scale;
      ball.visible = true;
    } else {
      ball.visible = false;
    }

    if (ball.visible) {
      ball.draw(context);
    }
  }());
};
</script>
</body>
</html>
```

代码使用了第 8 章中介绍的基本弹动公式，只不过加入了 z 轴。

15.7 坐标旋转

接下来是三维坐标旋转。这比第 10 章和第 11 章中介绍的二维坐标旋转要复杂一些。不仅仅是可以在三条不同的轴上旋转，甚至可以同时在两条以上的上轴旋转。

在二维坐标旋转中，坐标点是绕着 z 轴旋转的，如图 15-9 所示。想象一下风车绕车轴旋转的样子，车轴就是 z 轴。只有 x 坐标和 y 坐标在改变。

在三维系统中，也可以绕 x 轴或 y 轴旋转。x 轴旋转就像一个轮胎向你滚来，如图 15-10 所示，车轴就是 x 轴。点围绕 x 轴旋转并且只改变其 y 坐标和 z 坐标。

绕 y 轴旋转：想象一下唱片机，如图 15-11 所示，转轴就是 y 轴，x 坐标和 z 坐标变化。

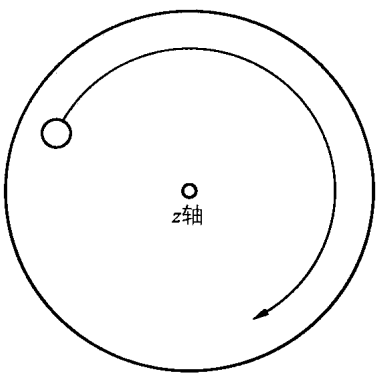


图 15-9 绕 z 轴旋转

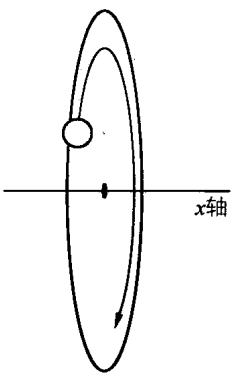


图 15-10 绕 x 轴旋转

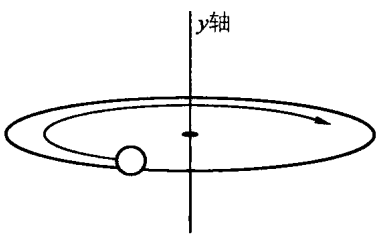


图 15-11 绕 y 轴旋转

因此，在三维系统中，当物体绕着某一条轴旋转时，它在其他两条轴上的坐标发生变化。

在第 10 章中，我们用如下公式来计算二维坐标旋转：

```
x1 = x * cos(angle) - y * sin(angle)
y1 = y * cos(angle) + x * sin(angle)
```

在三维系统中基本相同，但是要指定绕哪条轴旋转：x、y 或 z。这样就得到以下三组公式：

```
x1 = x * cos(angleZ) - y * sin(angleZ)
y1 = y * cos(angleZ) + x * sin(angleZ)

x1 = x * cos(angleY) - z * sin(angleY)
z1 = z * cos(angleY) + x * sin(angleY)

y1 = y * cos(angleX) - z * sin(angleX)
z1 = z * cos(angleX) + y * sin(angleX)
```

在下面的例子（12-rotate-y.html）中，我们将实现绕 y 轴旋转。它创建 Ball3d 的 50 个实例，位置随机。然后根据鼠标指针的 x 坐标计算出它与 y 轴的角度。鼠标越靠右，角度越大。物体就像是随着鼠标指针旋转一样。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Rotate Y</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball3d.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
      mouse = utils.captureMouse(canvas),
        balls = [],
        numBalls = 50,
        fl = 250,
        vpX = canvas.width / 2,
        vpY = canvas.height / 2,
        angleY; //referenced in drawFrame and move

    for (var ball, i = 0; i < numBalls; i++) {
      ball = new Ball3d(15);
      ball.xpos = Math.random() * 200 - 100;
      ball.ypos = Math.random() * 200 - 100;
      ball.zpos = Math.random() * 200 - 100;
      balls.push(ball);
    }

    function rotateY (ball, angle) {
      var cos = Math.cos(angle),
          sin = Math.sin(angle),
          x1 = ball.xpos * cos - ball.zpos * sin,
          z1 = ball.zpos * cos + ball.xpos * sin;
      ball.xpos = x1;
      ball.zpos = z1;

      if (ball.zpos > -fl) {
        var scale = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
      } else {
        ball.visible = false;
      }
    }

    function move (ball) {
      rotateY(ball, angleY);
    }
```

```
    function zSort (a, b) {
      return (b.zpos - a.zpos);
    }

    function draw (ball) {
      if (ball.visible) {
        ball.draw(context);
      }
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);
      angleY = (mouse.x - vpX) * 0.001;

      balls.forEach(move);
      balls.sort(zSort);
      balls.forEach(draw);
    }());
  };
  </script>
 </body>
</html>
```

粗体部分是重点。获得一个角度，把它和每个小球传入函数 rotateY。在这个函数中计算这个角度的正弦值和余弦值，进行旋转相关的计算，然后把旋转后的 x1 和 z1 赋值给小球的 xpos 和 zpos 属性。最后是标准的透视图和 z 排序。运行结果如图 15-12 所示。

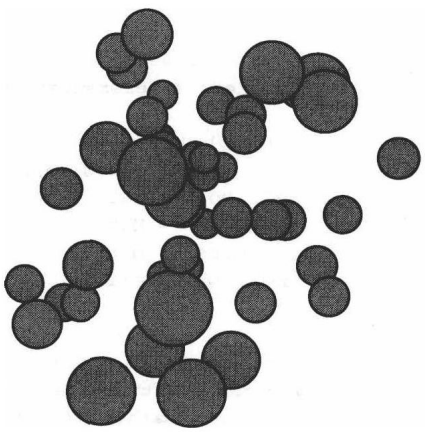


图 15-12 绕 y 轴旋转

运行一下试试看，然后可以试着把这个例子转换为绕 x 轴旋转。在脚本的开始添加一个新变量 anglex，然后更新 drawFrame 和 move 函数：

```
function move (ball) {
  rotateX(ball, angleX);
}

(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  angleX = (mouse.y - vpY) * 0.001;

balls.forEach(move);
  balls.sort(zSort);
  balls.forEach(draw);
}());
```

然后，只需要新建一个 rotateX 函数来执行旋转：

```
function rotateX (ball, angle) {
  var cos = Math.cos(angle),
      sin = Math.sin(angle),
```

```
        y1 = ball.ypos * cos - ball.zpos * sin,
        z1 = ball.zpos * cos + ball.ypos * sin;

    ball.ypos = y1;
    ball.zpos = z1;

    if (ball.zpos > -fl) {
        var scale = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
    } else {
        ball.visible = false;
    }
}
```

这个角度基于鼠标指针的 y 坐标。我们计算这个角度的正弦值和余弦值，并根据它们得到 y1 和 z1，再赋值给小球的 ypos 和 zpos 属性。

在下一个例子中，我们将把两种旋转组合起来，下面给出代码（13-rotate-xy.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Rotate XY</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball3d.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        balls = [],
        numBalls = 50,
        fl = 250,
        vpX = canvas.width / 2,
        vpY = canvas.height / 2,
        angleX, angleY; //referenced in drawFrame and move

    for (var ball, i = 0; i < numBalls; i++) {
      ball = new Ball3d(15);
      ball.xpos = Math.random() * 200 - 100;
      ball.ypos = Math.random() * 200 - 100;
      ball.zpos = Math.random() * 200 - 100;
      balls.push(ball);
    }

    function rotateX (ball, angle) {
      var cos = Math.cos(angle),
```

```
        sin = Math.sin(angle),
        y1 = ball.ypos * cos - ball.zpos * sin,
        z1 = ball.zpos * cos + ball.ypos * sin;
    ball.ypos = y1;
    ball.zpos = z1;
}

function rotateY (ball, angle) {
    var cos = Math.cos(angle),
        sin = Math.sin(angle),
        x1 = ball.xpos * cos - ball.zpos * sin,
        z1 = ball.zpos * cos + ball.xpos * sin;
    ball.xpos = x1;
    ball.zpos = z1;
}

function setPerspective (ball) {
    if (ball.zpos > -fl) {
        var scale = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
    } else {
        ball.visible = false;
    }
}

function move (ball) {
    rotateX(ball, angleX);
    rotateY(ball, angleY);
    setPerspective(ball);
}

function zSort (a, b) {
    return (b.zpos - a.zpos);
}

function draw (ball) {
    if (ball.visible) {
        ball.draw(context);
    }
}

(function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);
    angleX = (mouse.y - vpY) * 0.001;
    angleY = (mouse.x - vpX) * 0.001;

    balls.forEach(move);
    balls.sort(zSort);
    balls.forEach(draw);
}());
```

```
  };
  </script>
 </body>
</html>
```

这里，你同时计算出 angleY 和 angleX，并为每个小球同时调用 rotateX 和 rotateY 函数。透视图代码已经从 rotate 函数中移到自己的函数 setPerspective 中，因为没必要调用它两次。根据学到的内容和前面的公式，你应该能很容易地加入 rotateZ 函数。

15.8 碰撞检测

最后一个关于三维的知识点是碰撞检测。在基于 canvas 和 JavaScript 的三维动画中，进行碰撞检测唯一可行的办法就是距离检测。这与二维中的碰撞检测相差不大：首先找出两个物体间的距离（使用三维距离公式），如果距离小于两个物体的半径之和，则发生了碰撞。

我们把前面的三维反弹的例子修改为三维碰撞检测的例子，减少物体的数量并扩大空间。首先执行正常的三维运动和透视图代码，然后用一个双重循环来比较所有物体间的距离并检测碰撞。一旦发生了碰撞，我们就把两个小球的颜色改变为蓝色。下面是该练习的代码（14-collision-3d.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Collision 3d</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball3d.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        balls = [],
        numBalls = 20,
        fl = 250,
        vpX = canvas.width / 2,
        vpY = canvas.height / 2,
        top = -200,
        bottom = 200,
        left = -200,
        right = 200,
        front = -200
        back = 200;

    for (var ball, i = 0; i < numBalls; i++) {
      ball = new Ball3d(15);
      ball.xpos = Math.random() * 400 - 200;
      ball.ypos = Math.random() * 400 - 200;
```

```
    ball.zpos = Math.random() * 400 - 200;
    ball.vx = Math.random() * 5 - 1;
    ball.vy = Math.random() * 5 - 1;
    ball.vz = Math.random() * 5 - 1;
    balls.push(ball);
}

function move (ball) {
    ball.xpos += ball.vx;
    ball.ypos += ball.vy;
    ball.zpos += ball.vz;

    //check boundaries
    if (ball.xpos + ball.radius > right) {
        ball.xpos = right - ball.radius;
        ball.vx *= -1;
    } else if (ball.xpos - ball.radius < left) {
        ball.xpos = left + ball.radius;
        ball.vx *= -1;
    }
    if (ball.ypos + ball.radius > bottom) {
        ball.ypos = bottom - ball.radius;
        ball.vy *= -1;
    } else if (ball.ypos - ball.radius < top) {
        ball.ypos = top + ball.radius;
        ball.vy *= -1;
    }
    if (ball.zpos + ball.radius > back) {
        ball.zpos = back - ball.radius;
        ball.vz *= -1;
    } else if (ball.zpos - ball.radius < front) {
        ball.zpos = front + ball.radius;
        ball.vz *= -1;
    }
    if (ball.zpos > -fl) {
        var scale = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
    } else {
        ball.visible = false;
    }
}

function checkCollision (ballA, i) {
    for (var ballB, dx, dy, dz, dist, j = i + 1; j < numBalls; j++) {
        ballB = balls[j];
        dx = ballA.xpos - ballB.xpos;
        dy = ballA.ypos - ballB.ypos;
        dz = ballA.zpos - ballB.zpos;
        dist = Math.sqrt(dx * dx + dy * dy + dz * dz);

        if (dist < ballA.radius + ballB.radius) {
            ballA.color = "#0000ff";
```

```
            ballB.color = "#0000ff";
          }
        }
      }

      function zSort (a, b) {
        return (b.zpos - a.zpos);
      }

      function draw (ball) {
        if (ball.visible) {
          ball.draw(context);
        }
      }

      (function drawFrame () {
        window.requestAnimationFrame(drawFrame, canvas);
        context.clearRect(0, 0, canvas.width, canvas.height);
        balls.forEach(move);
        balls.forEach(checkCollision);
        balls.sort(zSort);
        balls.forEach(draw);
      }());
    };
    </script>
  </body>
</html>
```

重点部分用粗体标出。开始小球都是红色的，一旦发生碰撞，它们的颜色就改变。没多久就全变成蓝色的。

15.9 本章中的重要公式

本章中的重要公式是三维透视图、坐标旋转和距离计算。

15.9.1 基本透视图

```
scale = fl / (fl + zpos);
object.scaleX = object.scaleY = scale;
object.alpha = scale; // optional
object.x = vanishingPointX + xpos * scale;
object.y = vanishingPointY + ypos * scale;
```

15.9.2 Z 排序

```
//assumes an array of 3D objects with a zpos property
function zSort (a, b) {
return (b.zpos - a.zpos);
}
objects.sort(zSort);
```

15.9.3 坐标旋转

```
x1 = xpos * cos(angleZ) - ypos * sin(angleZ);
y1 = ypos * cos(angleZ) + xpos * sin(angleZ);

x1 = xpos * cos(angleY) - zpos * sin(angleY);
z1 = zpos * cos(angleY) + xpos * sin(angleY);

y1 = ypos * cos(angleX) - zpos * sin(angleX);
z1 = zpos * cos(angleX) + ypos * sin(angleX);
```

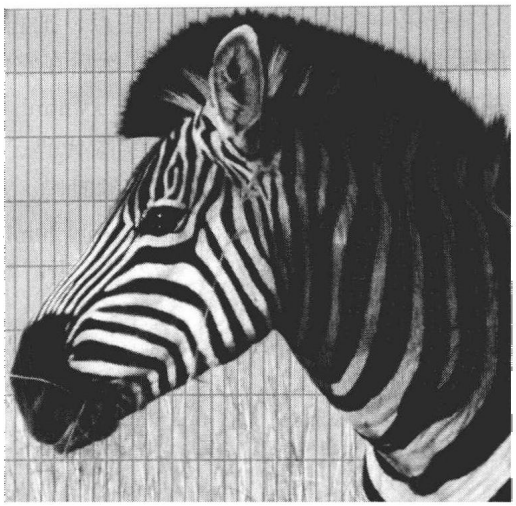
15.9.4 三维距离计算

```
dist = Math.sqrt(dx * dx + dy * dy + dz * dz);
```

15.10 小结

现在你已经有了三维的基础，并且已经看到了大多数基本运动在三维空间中的实现代码。令人惊讶的是，这些代码与二维系统中的非常相似，只不过多加了 *z* 轴上的一个变量。事实证明，大多数例子是非常简单的。

本章学到的内容将在第 16 章中大量用到。在第 16 章中你将开始真正地使用点和线雕塑三维造型。



第 16 章　三维线条与填充

本章涵盖以下内容：

- 创建点和线
- 创建图形
- 创建三维填充
- 三维实体建模
- 移动三维实体

第 15 章介绍了三维的基本内容，但只是通过计算物体的大小和屏幕位置把它们置于三维空间中，物体本身仍然是二维的。如果在空间中移动观察点的位置，物体看起来好像在随之旋转，永远都面向你。但是物体并没有旋转，只是看起来像是这样，因为它是二维物体，所以关于它只有这么一个视角。

本章用 canvas 绘图 API 来创建三维模型。具体地说，我们将学习如何创建和使用三维的点、线、填充以及实体。读完本章，你将学会创建多种三维图形，以及如何移动和旋转它们。

16.1　创建点和线

在理论上，点是不可见的，因为它没有维度。在三维空间中，如果创建一些点，但不用线把它们连起来，这是没有意义的。首先，你继续使用 Ball3D 类的实例，把它的半径设置得很小，它看起来就像是一个点。然后，绘制线条把小球连接起来。前面已经做过类似的事情了，不过这次要使用透视图，把它们放在三维空间中。

在第一个例子中，创建一些点，把它们的颜色设置为黑色，直径设置为 10 个像素。根据鼠标指针位置对它们进行旋转，然后在它们之间连线。代码基本与第 15 章中的例子 13-rotate-xy.html 相同。不同的地方是在 move 函数中加入了一些绘图代码，同时移除了 sortZ 函数。这是因为在绘制线框模型的时候，点的深度没什么意义。下面是代码（文件 01-lines- 3d-1.html），结果如图 16-1 所示。

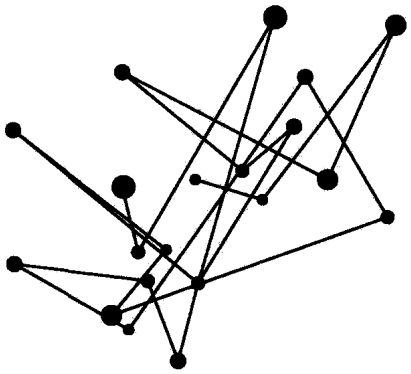


图 16-1 三维点与线

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Lines 3d 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball3d.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        balls = [],
        numBalls = 15,
        fl = 250,
        vpX = canvas.width / 2,
        vpY = canvas.height / 2,
        angleX, angleY; //referenced in drawFrame and move

    for (var ball, i = 0; i < numBalls; i++) {
      ball = new Ball3d(5, "#000000");
      ball.xpos = Math.random() * 200 - 100;
      ball.ypos = Math.random() * 200 - 100;
      ball.zpos = Math.random() * 200 - 100;
      balls.push(ball);
    }

    function rotateX (ball, angle) {
      var cos = Math.cos(angle),
          sin = Math.sin(angle),
          y1 = ball.ypos * cos - ball.zpos * sin,
          z1 = ball.zpos * cos + ball.ypos * sin;

      ball.ypos = y1;
      ball.zpos = z1;
    }

    function rotateY (ball, angle) {
      var cos = Math.cos(angle),
```

```
        sin = Math.sin(angle),
        x1 = ball.xpos * cos - ball.zpos * sin,
        z1 = ball.zpos * cos + ball.xpos * sin;

    ball.xpos = x1;
    ball.zpos = z1;
  }

  function setPerspective (ball) {
    if (ball.zpos > -fl) {
        var scale = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
    } else {
        ball.visible = false;
    }
  }

  function move (ball, i) {
    rotateX(ball, angleX);
    rotateY(ball, angleY);
    setPerspective(ball);

    //don't draw line to first ball
    if (i !== 0) {
      context.lineTo(balls[i].x, balls[i].y);
    }
  }

  function draw (ball) {
    if (ball.visible) {
      ball.draw(context);
    }
  }

  (function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);
    context.clearRect(0, 0, canvas.width, canvas.height);

    angleX = (mouse.y - vpY) * 0.001;
    angleY = (mouse.x - vpX) * 0.001;

    context.beginPath();
    //line starts at first ball
    context.moveTo(balls[0].x, balls[0].y);
    balls.forEach(move);
    context.stroke();
    balls.forEach(draw);
  }());
 };
 </script>
</body>
</html>
```

当你真正地绘制三维实体时，这些黑点就没有必要了。简单的方案是把小球的半径设置为 0，像这样：

```
var ball = new Ball3d(0);
```

可以看到的结果如图 16-2 所示。

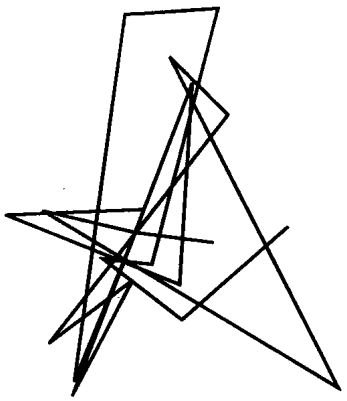


图 16-2 三维的线和不可见的点

但是如果观察 Ball3D 的定义，就会发现有一些代码现在变成多余的了。ScaleX、scaleY 和 visible 属性对于不可见的点没有什么用处，因为它不可见，也不会缩放。实际上，整个类主要是一个存储绘图命令的方便地方。因为三维点只需要有很少几个属性，所以可以重新创建一个更简单的类，只包含需要的属性。另外，透视图、坐标转换函数都将转移到这个新类中，这样可以让主文件结构保持清晰。下面是 Point3d 类（文件 point3d.js），它将贯穿与本章和第 17 章：

```
function Point3d (x, y, z) {
  this.x = (x === undefined) ? 0 : x;
  this.y = (y === undefined) ? 0 : y;
  this.z = (z === undefined) ? 0 : z;
  this.fl = 250; //focal length
  this.vpX = 0;  //vanishing point
  this.vpY = 0;
  this.cX = 0;   //center
  this.cY = 0;
  this.cZ = 0;
}

Point3d.prototype.setVanishingPoint = function (vpX, vpY) {
  this.vpX = vpX;
  this.vpY = vpY;
};

Point3d.prototype.setCenter = function (cX, cY, cZ) {
  this.cX = cX;
  this.cY = cY;
  this.cZ = cZ;
};

Point3d.prototype.rotateX = function (angleX) {
  var cosX = Math.cos(angleX),
      sinX = Math.sin(angleX),
      y1 = this.y * cosX - this.z * sinX,
      z1 = this.z * cosX + this.y * sinX;

  this.y = y1;
  this.z = z1;
};

Point3d.prototype.rotateY = function (angleY) {
  var cosY = Math.cos(angleY),
      sinY = Math.sin(angleY),
```

```
      x1 = this.x * cosY - this.z * sinY,
      z1 = this.z * cosY + this.x * sinY;

  this.x = x1;
  this.z = z1;
};

Point3d.prototype.rotateZ = function (angleZ) {
  var cosZ = Math.cos(angleZ),
      sinZ = Math.sin(angleZ),
      x1 = this.x * cosZ - this.y * sinZ,
      y1 = this.y * cosZ + this.x * sinZ;

  this.x = x1;
  this.y = y1;
};
Point3d.prototype.getScreenX = function () {
  var scale = this.fl / (this.fl + this.z + this.cZ);
  return this.vpX + (this.cX + this.x) * scale;
};
Point3d.prototype.getScreenY = function () {
  var scale = this.fl / (this.fl + this.z + this.cZ);
  return this.vpY + (this.cY + this.y) * scale;
};
```

为 Point3d 指定三维空间中的 x、y 和 z 坐标，初始化 Point 3D 类，就能创建一个三维点。这个类也包含焦距、消失点和中心点的属性。使用三种旋转方法，就可以让点围绕它的中心点在任意轴上旋转。这个类还包含所有的透视图计算方法。当移动或旋转点之后，可以调用 getScreenX 和 getScreenY 来获得这个点在 canvas 元素上经过透视图计算得到的坐标。为了帮助你更好地理解接下来几章中的例子，本书中这种预置的类都保持了最简化。使用新的 Point3d 类来重构前面旋转线的例子，将它变得更简单，代码如下（文件 02-lines-3d-2.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Lines 3d 2</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="point3d.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        points = [],
        numPoints = 50,
        fl = 250,
        vpX = canvas.width / 2,
        vpY = canvas.height / 2,
```

```
        angleX, angleY; //referenced in drawFrame and move
    for (var point, i = 0; i < numPoints; i++) {
      point = new Point3d(Math.random() * 200 - 100,
                          Math.random() * 200 - 100,
                          Math.random() * 200 - 100);
      point.setVanishingPoint(vpX, vpY);
      points.push(point);
    }

    function move (point) {
      point.rotateX(angleX);
      point.rotateY(angleY);
    }

    function draw (point, i) {
      //ignore first point
      if (i !== 0) {
        context.lineTo(point.getScreenX(), point.getScreenY());
      }
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      angleX = (mouse.y - vpY) * 0.001;
      angleY = (mouse.x - vpX) * 0.001;

      points.forEach(move);

      context.beginPath();
      //line starts at first point
      context.moveTo(points[0].getScreenX(), points[0].getScreenY());
      points.forEach(draw);
      context.stroke();
    }());
  };
  </script>
 </body>
</html>
```

主要的变化用粗体代码标出，应该很好理解。别忘了在文件中引用正确的 point3d.js 文件。

16.2 创建图形

用随机线条作为示例很不错，但是我们也可以把混乱的线条变得有序。你要做的就是用一些专门预定义的 x、y 和 z 值来初始化点，而不是使用一个初始循环来随机初始化。举个例子，我们来创建一个正方形。这个正方形和它 4 个角的三维坐标如图 16-3 所示。

z 坐标的值都一样，因为这是一个平面上的正方形。要保持正方形在同一平面上，最简单的方法是让它所有的点在某条轴上的坐标保持不变（这里选择 z 轴），然后在其他两条轴（x 和

y）上定义这个正方形。

用下面的代码来替换创建随机点的循环：

```
points[0] = new Point3d(-100, -100, 100);
points[1] = new Point3d( 100, -100, 100);
points[2] = new Point3d( 100, 100, 100);
points[3] = new Point3d(-100, 100, 100);
```

然后手动设置每个点的消失点，这可以通过遍历 points 数组来设置：

```
points.forEach(function (point) {
  point.setVanishingPoint(vpX, vpY);
});
```

其他代码应该可以正常运行，不过还要多做一点工作：用线连接最后一个点和第一个点，以闭合图形。这通过在所有线条都绘制完成之后调用 context.closePath 来实现。下面是完整的代码，可以在 03-square-3d.html 中找到，结果如图 16-4 所示。

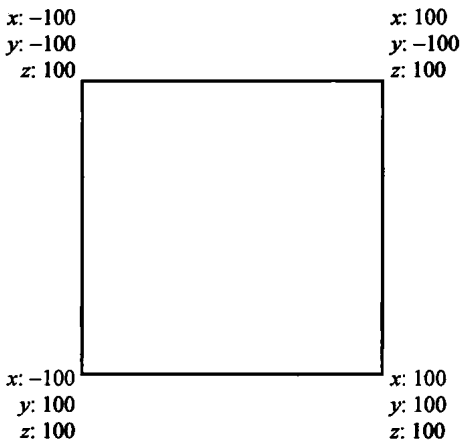


图 16-3 正方形在三维空间中的坐标

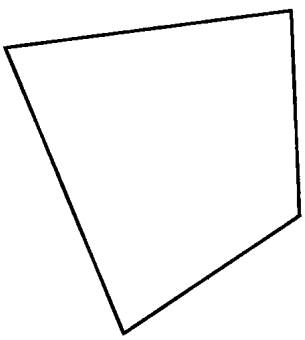


图 16-4 旋转的正方形

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Square 3d</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="point3d.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        points = [],
        fl = 250,
```

```
        vpX = canvas.width / 2,
        vpY = canvas.height / 2,
        angleX, angleY; //referenced in drawFrame and move

    //create 4 points
    points[0] = new Point3d(-100, -100, 100);
    points[1] = new Point3d( 100, -100, 100);
    points[2] = new Point3d( 100,  100, 100);
    points[3] = new Point3d(-100,  100, 100);

    points.forEach(function (point) {
      point.setVanishingPoint(vpX, vpY);
    });

    function move (point) {
      point.rotateX(angleX);
      point.rotateY(angleY);
    }

    function draw (point, i) {
      if (i !== 0) {
        context.lineTo(point.getScreenX(), point.getScreenY());
      }
    }

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      angleX = (mouse.y - vpY) * 0.0005;
      angleY = (mouse.x - vpX) * 0.0005;

      points.forEach(move);

      context.beginPath();
      context.moveTo(points[0].getScreenX(), points[0].getScreenY());
      points.forEach(draw);
      context.closePath();
      context.stroke();
    }());
  };
  </script>
 </body>
</html>
```

看，一个旋转的正方形！你现在可以创建任何平面图形了。你可能发现先把这些点标注在网格上，然后再把这些位置转为代码，创建图形会更简单一些，如图 16-5 所示。

参考这个素描，这样创建图形的点：

```
points[0] = new Point3d(-150, -250, 100);
points[1] = new Point3d( 150, -250, 100);
points[2] = new Point3d( 150, -150, 100);
points[3] = new Point3d( -50, -150, 100);
points[4] = new Point3d( -50, -50, 100);
```

```
points[5] = new Point3d( 50, -50, 100);
points[6] = new Point3d( 50, 50, 100);
points[7] = new Point3d( -50, 50, 100);
points[8] = new Point3d( -50, 150, 100);
points[9] = new Point3d( 150, 150, 100);
points[10] = new Point3d( 150, 250, 100);
points[11] = new Point3d(-150, 250, 100);
```

现在就有了一个旋转的字母 E，完整的代码在文件 04-spinning-e.html 中，运行结果如图 16-6 所示。

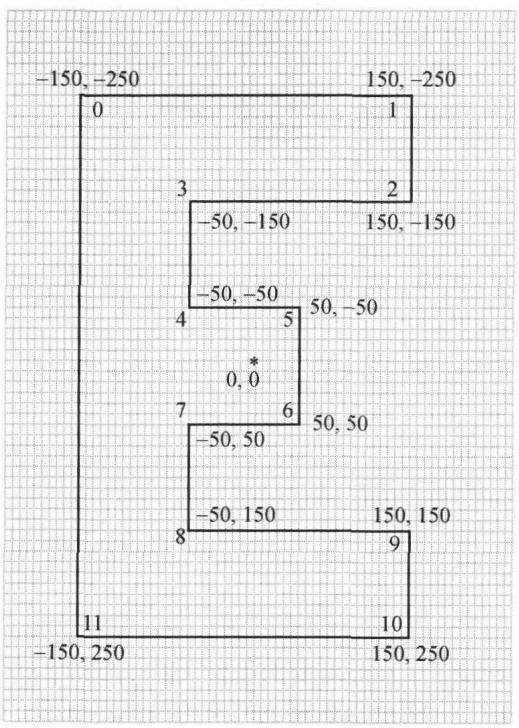


图 16-5 用网格标注点，描绘字母 E

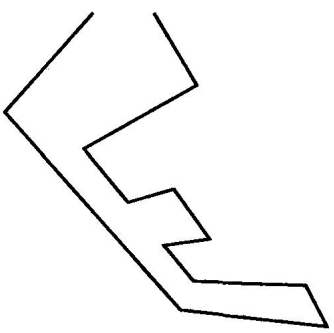


图 16-6 旋转的三维字母 E

当尝试其他不同的图形时，你可能会发现如果有些点的间距过近，就会发生第 15 章中的第一个三维例子里的倒置 bug。你也许首先会想到增加所有点的 z 坐标值，但是这样会变得更糟。比如，把 z 设置为 500，当它旋转时，z 的范围是−500～500，有些点就会跑到观察点后面去，这样就变得更加混乱。而做法是调用 setCenter 方法来把图形整体向 z 轴方向推移，像这样：

```
points.forEach(function (point) {
  point.setVanishingPoint(vpX, vpY);
  point.setCenter(0, 0, 200);
});
```

回顾一下 Point3d 类，你会看到计算 scale 的公式：

```
var scale = this.fl / (this.fl + this.z + this.cZ);
```

所以，当设置中心点时，是把整个系统，包括系统的旋转，都向后推移了 200 个像素，z

坐标并没有变，只是计算透视图的值更高了，所以整个图形都会在观察者的前面。尝试使用其他值并观察效果，稍后我们将在另外两条轴上移动图形。

16.3 创建三维填充

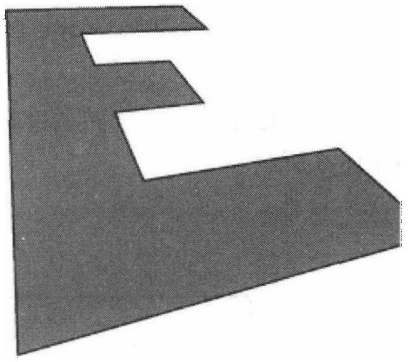


图 16-7 第一个三维填充

你可能已经想到，填充的大部分工作已经做完了。你已经创建了构成图形的点，而且用线条把它们勾勒出来了。剩下要做的就是在绘图代码中调用 context.fillStyle 和 context.fill。05-filled-e.html 文件给出了完整的代码，运行结果如图 16-7 所示。下面列出了 drawFrame 函数的相关部分，变化的部分用粗体标出：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  angleX = (mouse.y - vpY) * 0.0005;
  angleY = (mouse.x - vpX) * 0.0005;

  points.forEach(move);

  context.fillStyle = "#ff0000";
  context.beginPath();
  context.moveTo(points[0].getScreenX(), points[0].getScreenY());
  points.forEach(draw);
  context.closePath();
  context.stroke();
  context.fill();
}());
```

这是个理解传统的三维图形和实体建模的好例子。在前面的例子里，正方形和字母 E 都是多边形。多边形是指至少有三个顶点的封闭图形，三角形是最简单的多边形。在很多三维建模和绘制程序里（甚至那些使用了补丁（Patch）、网格（mesh）、非均匀 B 样曲线（NURBS）以及复杂多边形的三维建模和绘图程序），所有的三维图形最终都在绘制前化简为一个三角形集合。

16.3.1 使用三角形

使用三角形有不少好处。无论一个三角形的三个顶点怎么放置，都可以肯定这个三角形所有的点处于同一平面上，三角形的表面指向同一个方向。这个方向叫做**表面法线**（surface normal），它垂直于三角形的表面，如图 16-8 所示。如果不是三角形，而是其他的多边形，它就有可能会随着顶点位置的变化而变得扭曲。用前面例子中的字母 E 举例，随机地改变顶点的 *z* 坐标。虽然效果可能很有趣，但是结果也很快变得不可预料。

三角形的另外一个好处是，任何复杂的多边形都可以由三角形组成，如图 16-9 所示。

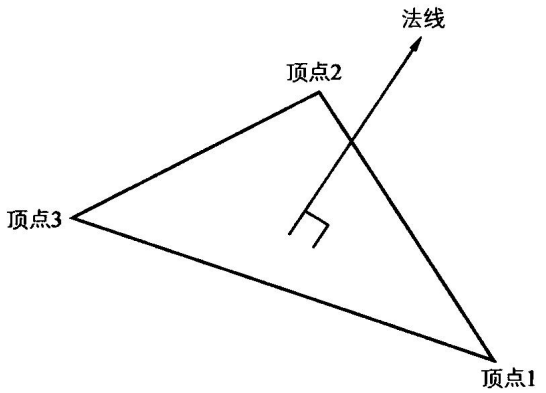


图 16-8 三维三角形的表面法线

这类封闭图形很难用一个多边形表示。你会发现自己陷入了这么一种情况：每一个多边形的顶点数目都不一定相同，并且在绘制的时候需要特殊处理。如果使用三角形组合，你就可以统一、高效并且规范（不需要特殊处理）地为所有图形建模。如图 16-10 所示，你可以看到，使用三角形把同样的封闭图形分解成了多个小多边形。

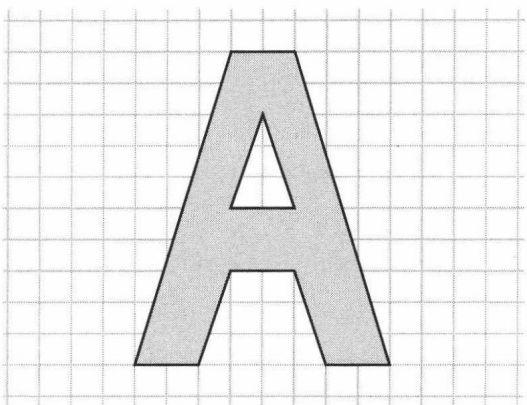


图 16-9 更复杂的三维图形

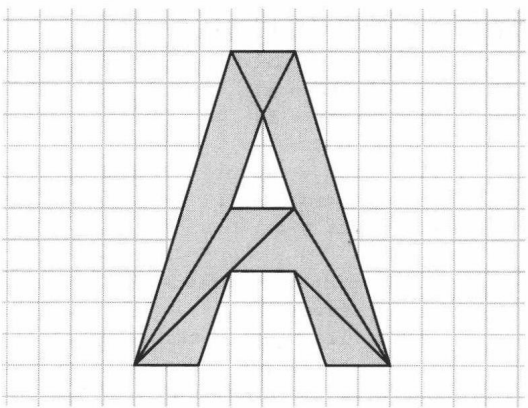


图 16-10 与图 16-9 相同的图形，用三角形渲染

你可以新建一个函数，它接受三个点作为参数，然后绘制一个三角形。这样，你只需要一个顶点列表和一个三角形列表。一个循环遍历顶点列表，并应用透视图计算它们的位置。另一个循环遍历三角形列表并渲染它们。

使用三角形也可以在几何学上做更进一步的优化。三角扇（triangle fans）和三角条带（triangle strips）是三维几何学中常用的方法，因为它们可以高效地遍历所有顶点，并消除重复顶点。使用三角形的另一个好处是，可以用常见的三角学计算来操作它们，就和贯穿本书的计算一样。

这并不是说你必须仅用三角形的方式。你可以创建一个函数来动态地渲染任意形状的多边形。但是这里为了保持简单和灵活性，我们使用三角形。

我可以从字母 A 的例子开始。首先，你需要定义所有的顶点和三角形。如图 16-11 所示，我们对形状进行布局并且给每个顶点和每个三角形都进行了编号。

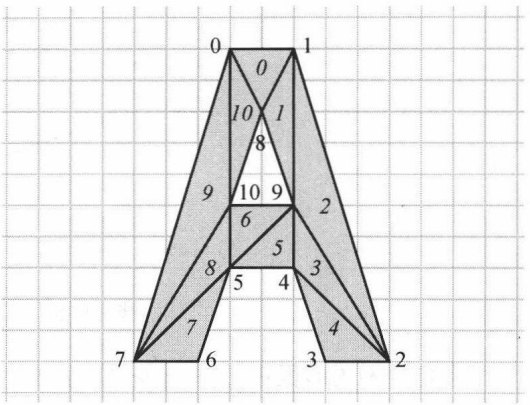


图 16-11 组成图形的顶点和多边形

当你标出所有顶点后，得到如下值：

```
points[0]  = new Point3d( -50, -250, 100);
points[1]  = new Point3d(  50, -250, 100);
points[2]  = new Point3d( 200,  250, 100);
points[3]  = new Point3d( 100,  250, 100);
points[4]  = new Point3d(  50,  100, 100);
points[5]  = new Point3d( -50,  100, 100);
points[6]  = new Point3d(-100,  250, 100);
points[7]  = new Point3d(-200,  250, 100);
points[8]  = new Point3d(   0, -150, 100);
points[9]  = new Point3d(  50,    0, 100);
points[10] = new Point3d( -50,    0, 100);
```

接下来，定义三角形。每个三角形只是一个有 a、b 和 c 三个点的列表。创建一个 Triangle 类来保存每个三角形的顶点，并且为它添加一个 draw 方法，这样它就能把自己绘制在 canvas 元素上。下面给出代码（文件 triangle.js），你很快就能看到它是如何使用的：

```
function Triangle (a, b, c, color) {
  this.pointA = a;
  this.pointB = b;
  this.pointC = c;
  this.color = (color === undefined) ? "#ff0000" : utils.parseColor(color);
  this.lineWidth = 1;
  this.alpha = 1;
}
Triangle.prototype.draw = function (context) {
  context.save();
  context.lineWidth = this.lineWidth;
  context.fillStyle = context.strokeStyle = utils.colorToRGB(this.color, this.alpha);
  context.beginPath();
  context.moveTo(this.pointA.getScreenX(), this.pointA.getScreenY());
  context.lineTo(this.pointB.getScreenX(), this.pointB.getScreenY());
  context.lineTo(this.pointC.getScreenX(), this.pointC.getScreenY());
  context.closePath();
  context.fill();
  if (this.lineWidth > 0) {
     context.stroke();
  }
  context.restore();
};
```

我们需要另一个数组来保存三角形列表。所以在脚本一开始的地方，定义这个数组：

```
var triangles = [];
```

然后，在定义所有顶点之后，用三个顶点和一种颜色来初始化这些三角形。

```
triangles[0]  = new Triangle(points[0], points[1],  points[8],  "#ffcccc");
triangles[1]  = new Triangle(points[1], points[9],  points[8],  "#ffcccc");
triangles[2]  = new Triangle(points[1], points[2],  points[9],  "#ffcccc");
triangles[3]  = new Triangle(points[2], points[4],  points[9],  "#ffcccc");
triangles[4]  = new Triangle(points[2], points[3],  points[4],  "#ffcccc");
triangles[5]  = new Triangle(points[4], points[5],  points[9],  "#ffcccc");
triangles[6]  = new Triangle(points[9], points[5],  points[10], "#ffcccc");
triangles[7]  = new Triangle(points[5], points[6],  points[7],  "#ffcccc");
triangles[8]  = new Triangle(points[5], points[7],  points[10], "#ffcccc");
triangles[9]  = new Triangle(points[0], points[10], points[7],  "#ffcccc");
triangles[10] = new Triangle(points[0], points[8],  points[10], "#ffcccc");
```

每个三角形的顶点都按照顺时针方向排序。这个在当前阶段并不重要，但是在第 17 章中很重要，所以要养成良好的习惯。

也许你觉得手动测绘这些顶点和三角形很乏味，这是因为它确实很乏味。当你为实体建模时这会变得更乏味。这就是为什么大多数三维程序都提供可视化建模工具，可以创建图形并提取出所有顶点和多边形。尽管创建三维建模程序超出了本书的范围，但是一个好的办法可能是解析一个开放的三维模型格式，比如 COLLADA，并把顶点位置保存为类似 JSON 的格式，然后读取这个文件并绘制这个模型。

现在，你的渲染循环看起来是下面的样子（别担心，一会儿就能看到完整版的文件）：

```
function draw (triangle) {
  triangle.draw(context);
}

(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  angleX = (mouse.y - vpY) * 0.0005;
  angleY = (mouse.x - vpX) * 0.0005;

  points.forEach(move);
  triangles.forEach(draw);
}());
```

这里遍历所有的三角形并传入 draw 函数，在 draw 中依次调用每个三角形自己的 draw 方法，绘制三角形并使用预定义的颜色来填充它。你可以看到该示例如图 16-12 所示的结果。

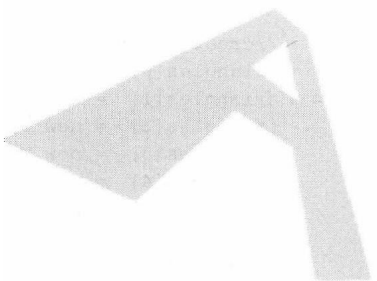


图 16-12 图形 A

这里给出完整的代码清单（文件 06-triangles.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Triangles</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
```

```
<script src="point3d.js"></script>
<script src="triangle.js"></script>
<script>
window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      mouse = utils.captureMouse(canvas),
      points = [],
      triangles = [],
      fl = 250,
      vpX = canvas.width / 2,
      vpY = canvas.height / 2,
      angleX, angleY; //referenced in drawFrame and move

  //the letter 'A', using 11 points
  points[0]  = new Point3d( -50, -250, 100);
  points[1]  = new Point3d(  50, -250, 100);
  points[2]  = new Point3d( 200,  250, 100);
  points[3]  = new Point3d( 100,  250, 100);
  points[4]  = new Point3d(  50,  100, 100);
  points[5]  = new Point3d( -50,  100, 100);
  points[6]  = new Point3d(-100,  250, 100);
  points[7]  = new Point3d(-200,  250, 100);
  points[8]  = new Point3d(   0, -150, 100);
  points[9]  = new Point3d(  50,    0, 100);
  points[10] = new Point3d( -50,    0, 100);

  points.forEach(function (point) {
    point.setVanishingPoint(vpX, vpY);
    point.setCenter(0, 0, 200);
  });

  //create 11 triangle objects from our points
  triangles[0]  = new Triangle(points[0], points[1],  points[8],  "#ffcccc");
  triangles[1]  = new Triangle(points[1], points[9],  points[8],  "#ffcccc");
  triangles[2]  = new Triangle(points[1], points[2],  points[9],  "#ffcccc");
  triangles[3]  = new Triangle(points[2], points[4],  points[9],  "#ffcccc");
  triangles[4]  = new Triangle(points[2], points[3],  points[4],  "#ffcccc");
  triangles[5]  = new Triangle(points[4], points[5],  points[9],  "#ffcccc");
  triangles[6]  = new Triangle(points[9], points[5],  points[10], "#ffcccc");
  triangles[7]  = new Triangle(points[5], points[6],  points[7],  "#ffcccc");
  triangles[8]  = new Triangle(points[5], points[7],  points[10], "#ffcccc");
  triangles[9]  = new Triangle(points[0], points[10], points[7],  "#ffcccc");
  triangles[10] = new Triangle(points[0], points[8],  points[10], "#ffcccc");

  function move (point) {
    point.rotateX(angleX);
    point.rotateY(angleY);
  }

  function draw (triangle) {
    triangle.draw(context);
  }
```

```
    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      angleX = (mouse.y - vpY) * 0.0005;
      angleY = (mouse.x - vpX) * 0.0005;

      points.forEach(move);
      triangles.forEach(draw);
    }());
  };
  </script>
 </body>
</html>
```

16.4 三维实体建模

在计算机世界里，每一本书的第一个例子几乎都是“Hello World”，也就是在屏幕上输出文字的一个程序。在我们开始绘制第一个三维实体时，我们也从简单的例子开始，绘制一个旋转的立方体。

16.4.1 建模旋转的立方体

要建模一个立方体，需要 8 个顶点来定义它的 8 个角，如图 16-13 所示。

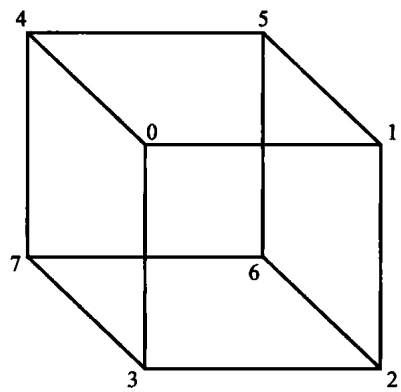


图 16-13 三维立方体的顶点

这些点的位置在代码中定义如下：

```
//front four corners
points[0] = new Point3d(-100, -100, -100);
points[1] = new Point3d( 100, -100, -100);
points[2] = new Point3d( 100,  100, -100);
points[3] = new Point3d(-100,  100, -100);

//back four corners
points[4] = new Point3d(-100, -100,  100);
points[5] = new Point3d( 100, -100,  100);
points[6] = new Point3d( 100,  100,  100);
points[7] = new Point3d(-100,  100,  100);
```

然后，需要定义三角形。因为立方体的每个面由两个三角形组成，所以一共有 12 个三角形（每面两个，共 6 个面）。同样，把每个三角形的顶点按顺时针方向排列，就和立方体正面的两个三角形一样。这有点棘手，但是你可以在脑海中旋转这个立方体，让正在定义的那个面朝向你。然后把 4 个顶点按顺时针方向编号。举个例子，立方体正面的两个三角形很容易定义，如图 16-14 所示。当你在脑海里继续旋转立方体时，你可以看到它的顶面，如图 16-15 所示，图 16-16 是它的背面。

继续旋转每个面，你会得到如下三角形定义：

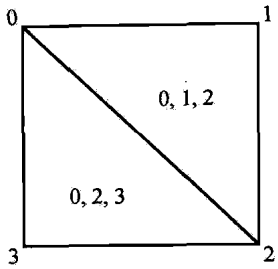


图 16-14 立方体的正面

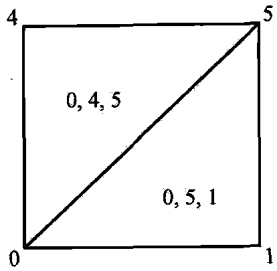


图 16-15 立方体的顶面

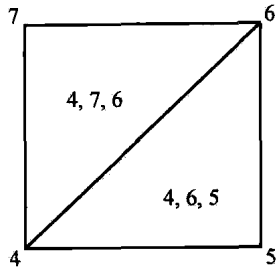


图 16-16 立方体的背面

```
//front
triangles[0]  = new Triangle(points[0], points[1], points[2], "#6666cc");
triangles[1]  = new Triangle(points[0], points[2], points[3], "#6666cc");

//top
triangles[2]  = new Triangle(points[0], points[5], points[1], "#66cc66");
triangles[3]  = new Triangle(points[0], points[4], points[5], "#66cc66");

//back
triangles[4]  = new Triangle(points[4], points[6], points[5], "#cc6666");
triangles[5]  = new Triangle(points[4], points[7], points[6], "#cc6666");

//bottom
triangles[6]  = new Triangle(points[3], points[2], points[6], "#cc66cc");
triangles[7]  = new Triangle(points[3], points[6], points[7], "#cc66cc");

//right
triangles[8]  = new Triangle(points[1], points[5], points[6], "#66cccc");
triangles[9]  = new Triangle(points[1], points[6], points[2], "#66cccc");

//left
triangles[10] = new Triangle(points[4], points[0], points[3], "#cccc66");
triangles[11] = new Triangle(points[4], points[3], points[7], "#cccc66");
```

每个面都有一种颜色，这是因为组成一个面的两个三角形颜色相同。这时，顺时针排列的三角形顶点还没有派上用场，但是这个将在第 17 章的**背面剔除**（backface culling）中用到。这是用来决定哪个面朝向你和哪个面背向你的一种方法。你很快就能看到它如此重要的原因。

例 07-cube.html 与 06-triangles.html 一样，只是脚本顶部使用了这些新的顶点和三角形的定义。

在浏览器中运行代码，全乱了！你有时可以看到一些背面。其他的面你永远都看不到。这是怎么了？立方体背后的那个面（背面）总是会绘制出来。三角形总是以它们在 triangles 数组中的顺序绘制出来，所以数组底部的面总是最后绘制出来，并且有了这种诡异的、不可预期的效果。你需要剔除（丢弃）这些背面，因为并不需要绘制它们。

第 17 章将详细介绍背面剔除，你还将学到如何在每个面上根据它的角度应用一些基本的灯光效果。但是作为本章的一个临时修复方案，在 Triangle 类的定义中，把 alpha 属性的值改为 0.5：

```
function Triangle (a, b, c, color) {
  this.pointA = a;
  this.pointB = b;
  this.pointC = c;
  this.color = (color === undefined) ? "#ff0000" : utils.parseColor(color);
```

```
    this.lineWidth = 1;
    this.alpha = 0.5;
}
```

这样你就能看到立方体所有的表面了，整个立方体看起来像是用彩色玻璃做的一样。同样，这只是一个临时的变通方案，我们后面会用背面剔除来修复它。

图 16-17 展示了三维立方体的效果。

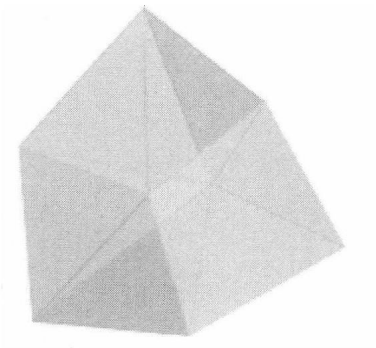


图 16-17 三维立方体的效果

16.4.2 建模其他形状

既然你已经掌握了旋转立方体，就可以建模所有的形状了。只需要把它们绘制在格子上，标注顶点和三角形，然后把它们放入数组中。从多个视角绘制物体是很有帮助的，这样你就能看到物体的每个面由哪些三角形组成。本节提供一些其他图形用来学习。

金字塔

下面是三维金字塔的代码（也可以在文件 08-pyramid.html 中找到），首先定义顶点：

```
points[0] = new Point3d(   0, -100,    0);
points[1] = new Point3d( 100,  100, -100);
points[2] = new Point3d(-100,  100, -100);
points[3] = new Point3d(-100,  100,  100);
points[4] = new Point3d( 100,  100,  100);
```

然后是三角形：

```
triangles[0] = new Triangle(points[0], points[1], points[2], "#6666cc");
triangles[1] = new Triangle(points[0], points[2], points[3], "#66cc66");
triangles[2] = new Triangle(points[0], points[3], points[4], "#cc6666");
triangles[3] = new Triangle(points[0], points[4], points[1], "#66cccc");
triangles[4] = new Triangle(points[1], points[3], points[2], "#cc66cc");
triangles[5] = new Triangle(points[1], points[4], points[3], "#cc66cc");
```

结果如图 16-18 所示。

拉伸的字母 A

在 09-extruded-a.html 中，我们拉伸（让形状具有深度，有立体感）前面例子中的字母 A。这意味着复制 11 个顶点，然后把一组顶点的 z 坐标设置为-50，另一组设置为+50，然后为第二组顶点也创建三角形（确保从背面看它们仍然是顺时针），最终用三角形把两个面连接起来。听起来很乏味，但是最终的效果会非常好（结果如图 16-19 所示）。

```
//first set
points[0]  = new Point3d( -50, -250, -50);
points[1]  = new Point3d(  50, -250, -50);
points[2]  = new Point3d( 200,  250, -50);
points[3]  = new Point3d( 100,  250, -50);
points[4]  = new Point3d(  50,  100, -50);
points[5]  = new Point3d( -50,  100, -50);
points[6]  = new Point3d(-100,  250, -50);
points[7]  = new Point3d(-200,  250, -50);
```

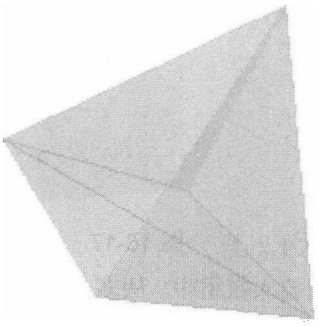


图 16-18 一个三维金字塔

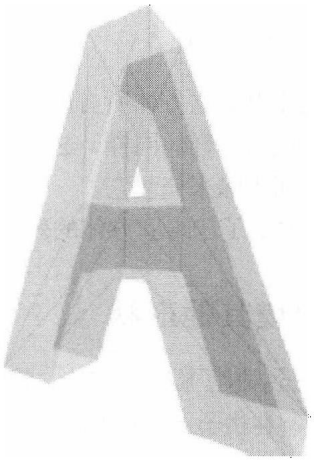


图 16-19 一个拉伸的字母 A

```
points[8]  = new Point3d(   0, -150, -50);
points[9]  = new Point3d(  50,    0, -50);
points[10] = new Point3d( -50,    0, -50);

//second set
points[11] = new Point3d( -50, -250,  50);
points[12] = new Point3d(  50, -250,  50);
points[13] = new Point3d( 200,  250,  50);
points[14] = new Point3d( 100,  250,  50);
points[15] = new Point3d(  50,  100,  50);
points[16] = new Point3d( -50,  100,  50);
points[17] = new Point3d(-100,  250,  50);
points[18] = new Point3d(-200,  250,  50);
points[19] = new Point3d(   0, -150,  50);
points[20] = new Point3d(  50,    0,  50);
points[21] = new Point3d( -50,    0,  50);

triangles[0]  = new Triangle(points[0],  points[1],  points[8],  "#6666cc");
triangles[1]  = new Triangle(points[1],  points[9],  points[8],  "#6666cc");
triangles[2]  = new Triangle(points[1],  points[2],  points[9],  "#6666cc");
triangles[3]  = new Triangle(points[2],  points[4],  points[9],  "#6666cc");
triangles[4]  = new Triangle(points[2],  points[3],  points[4],  "#6666cc");
triangles[5]  = new Triangle(points[4],  points[5],  points[9],  "#6666cc");
triangles[6]  = new Triangle(points[9],  points[5],  points[10], "#6666cc");
triangles[7]  = new Triangle(points[5],  points[6],  points[7],  "#6666cc");
triangles[8]  = new Triangle(points[5],  points[7],  points[10], "#6666cc");
triangles[9]  = new Triangle(points[0],  points[10], points[7],  "#6666cc");
triangles[10] = new Triangle(points[0],  points[8],  points[10], "#6666cc");
triangles[11] = new Triangle(points[11], points[19], points[12], "#cc6666");
triangles[12] = new Triangle(points[12], points[19], points[20], "#cc6666");
triangles[13] = new Triangle(points[12], points[20], points[13], "#cc6666");
triangles[14] = new Triangle(points[13], points[20], points[15], "#cc6666");
triangles[15] = new Triangle(points[13], points[15], points[14], "#cc6666");
triangles[16] = new Triangle(points[15], points[20], points[16], "#cc6666");
```

```
triangles[17] = new Triangle(points[20], points[21], points[16], "#cc6666");
triangles[18] = new Triangle(points[16], points[18], points[17], "#cc6666");
triangles[19] = new Triangle(points[16], points[21], points[18], "#cc6666");
triangles[20] = new Triangle(points[11], points[18], points[21], "#cc6666");
triangles[21] = new Triangle(points[11], points[21], points[19], "#cc6666");
triangles[22] = new Triangle(points[0],  points[11], points[1],  "#cccc66");
triangles[23] = new Triangle(points[11], points[12], points[1],  "#cccc66");
triangles[24] = new Triangle(points[1],  points[12], points[2],  "#cccc66");
triangles[25] = new Triangle(points[12], points[13], points[2],  "#cccc66");
triangles[26] = new Triangle(points[3],  points[2],  points[14], "#cccc66");
triangles[27] = new Triangle(points[2],  points[13], points[14], "#cccc66");
triangles[28] = new Triangle(points[4],  points[3],  points[15], "#cccc66");
triangles[29] = new Triangle(points[3],  points[14], points[15], "#cccc66");
triangles[30] = new Triangle(points[5],  points[4],  points[16], "#cccc66");
triangles[31] = new Triangle(points[4],  points[15], points[16], "#cccc66");
triangles[32] = new Triangle(points[6],  points[5],  points[17], "#cccc66");
triangles[33] = new Triangle(points[5],  points[16], points[17], "#cccc66");
triangles[34] = new Triangle(points[7],  points[6],  points[18], "#cccc66");
triangles[35] = new Triangle(points[6],  points[17], points[18], "#cccc66");
triangles[36] = new Triangle(points[0],  points[7],  points[11], "#cccc66");
triangles[37] = new Triangle(points[7],  points[18], points[11], "#cccc66");
triangles[38] = new Triangle(points[8],  points[9],  points[19], "#cccc66");
triangles[39] = new Triangle(points[9],  points[20], points[19], "#cccc66");
triangles[40] = new Triangle(points[9],  points[10], points[20], "#cccc66");
triangles[41] = new Triangle(points[10], points[21], points[20], "#cccc66");
triangles[42] = new Triangle(points[10], points[8],  points[21], "#cccc66");
triangles[43] = new Triangle(points[8],  points[19], points[21], "#cccc66");
```

正如你看到的，这些三角形的绘制速度相当快。在最初的例子里，屏幕上的平面字母 A 由 11 个三角形组成，拉伸 A 使得三角形的数目扩大了 4 倍！虽然代码仍然运行得很流畅，但是我们不会用这种方法在 canvas 元素上绘制有数千个多边形的绚丽三维效果。尽管如此，你仍然可以做一些很酷的事情。而且 Web 浏览器的性能也一直在不断地提高，新的技术（比如 WebGL）即将来临，谁知道未来会怎样？

圆柱体

再来举一个例子，这次将用数学方法来创建顶点和三角形。这个例子（文件 10-cylinder.html）只发生了一点点变化。在脚本顶部，不再手动定义顶点和三角形，而是用一个算法来创建它们，最终得到一个圆柱体。下面是这部分代码：

```
var points = [],
    numFaces = 20;

for (var angle, xpos, ypos, i = 0, idx = 0; i < numFaces; i++) {
  angle = Math.PI * 2 / numFaces * i;
  xpos = Math.cos(angle) * 200;
  ypos = Math.sin(angle) * 200;
  points[idx]   = new Point3d(xpos, ypos, -100);
  points[idx+1] = new Point3d(xpos, ypos,  100);
  idx += 2;
}
```

```
points.forEach(function (point) {
  point.setVanishingPoint(vpX, vpY);
  point.setCenter(0, 0, 200);
});

for (i = 0, idx = 0; i < numFaces - 1; i++) {
  triangles[idx] = new Triangle(points[idx], points[idx+3], points[idx+1], "#6666cc");
  triangles[idx+1] = new Triangle(points[idx], points[idx+2], points[idx+3], "#6666cc");
  idx += 2;
}

triangles[idx] = new Triangle(points[idx], points[1], points[idx+1], "#6666cc");
triangles[idx+1] = new Triangle(points[idx], points[0], points[1], "#6666cc");
```

因为这段代码可能并不直白，所以我们逐步解释它，或许还需要一两张图表。

首先用一个循环在圆上间隔性地创建一些点。每次循环时，用圆周除以表面的个数，再乘以当前表面的编号，计算出一个角度。

使用这个角度（一些三角公式要在这里用到），计算出圆上的点的 x、y 坐标。然后以此创建两个点，一个 z 坐标为-100，另一个 z 坐标为+100。当循环结束时，我们就有了两组可以组成圆的点。一组离你近一些，一组远一些。然后你需要用三角形连接它们。

再次，循环每一个面，每个面需要创建两个三角形。从侧面观察，第一个面如图 16-20 所示。

这里形成了两个三角形：

```
0, 3, 1
0, 2, 3
```

因为索引变量 idx 初始化为 0，所以也可以这么定义三角形：

```
idx, idx + 3, idx + 1
idx, idx + 2, idx + 3
```

这就是定义两个三角形的方法。然后可以把 idx 递增 2 来处理下一个面，来得到点 2、3、4 和 5。

一直循环到倒数第二个面，然后用最后一个面来连接最开始的两个点 0 和 1，如图 16-21 所示。

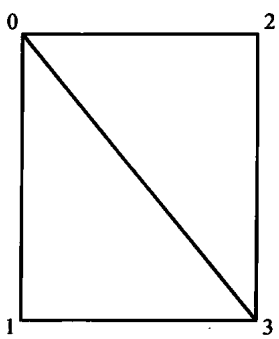


图 16-20 圆柱体的第一个面

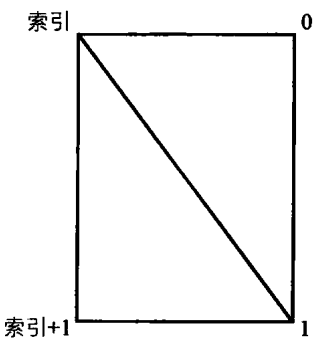


图 16-21 圆柱体的最后一个面

最后一个面的三角形如下：

```
idx, 1, idx + 1
idx, 0, 1
```

运行结果如图 16-22 所示，如果你想让圆柱体的骨架结构更为清晰，就把每个 Triangle 对象的 lineWidth 属性调大一些。

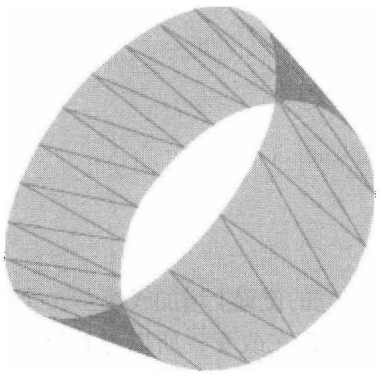


图 16-22 三维圆柱体的运行结果

16.5 移动三维实体

既然我们已经有了 Point3d 类，要移动一个三维实体就很容易，只需要用 setCenter 方法来改变它的中心位置。你已经在 *z* 轴上移动过图形了，在 *x* 轴和 *y* 轴上移动是一样的。但是首先来快速地看一下当这些值被改变时会发生什么。所有的逻辑都在 Point3d 类的 getScreenX 和 getScreenY 方法中：

```
Point3d.prototype.getScreenX = function () {
  var scale = this.fl / (this.fl + this.z + this.cZ);
  return this.vpX + (this.cX + this.x) * scale;
};

Point3d.prototype.getScreenY = function () {
  var scale = this.fl / (this.fl + this.z + this.cZ);
  return this.vpY + (this.cY + this.y) * scale;
};
```

在计算 scale 时，把中心 *z* 值（this.cZ）加在了 *z* 坐标上（this.z），这样就能在不改变 *z* 坐标的前提下把点向外推。

this.cX 与 this.cY 也类似。把它们加在了 *x* 和 *y* 坐标上，然后应用缩放。这样不必永久性地改变点的坐标就能移动点。因为我们没有改变这些坐标值，所以这些点以及它们组成的物体仍然围绕着自己的中心旋转，而不是绕着三维坐标空间的中心旋转。

快来行动吧，回到示例 07-cube.html，在脚本顶部加入两个变量：

```
var offsetX = 0,
    offsetY = 0;
```

然后使用方向键添加 keydown 事件处理函数用来调整偏移量：

```
window.addEventListener('keydown', function (event) {
  if (event.keyCode === 37) {        //left
    offsetX = -5;
  } else if (event.keyCode === 39) {  //right
    offsetX = 5;
  } else if (event.keyCode === 38) {  //up
    offsetY = -5;
  } else if (event.keyCode === 40) {  //down
    offsetY = 5;
  }

  points.forEach(function (point) {
    point.x += offsetX;
    point.y += offsetY;
  });
}, false);
```

这里首先循环每一个点，然后在它的位置上增加或减去 5 个像素（完整文件在 11-move-cube-1.html 中）。因为所有的点的实际位置都在改变，所以导致物体绕着三维空间的中心在旋转。这可能不是你想要的结果。如果你想移动整个模型，同时让它一直绕着自己的中心旋转，你就需要用到该点的 setCenter 方法了。把 keydown 事件处理函数做如下改动：

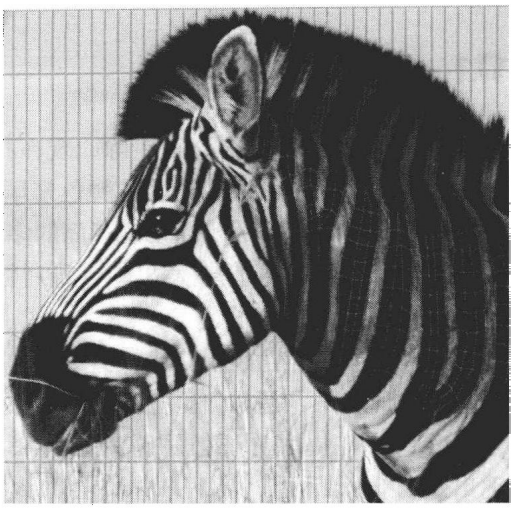
```
window.addEventListener('keydown', function (event) {
  if (event.keyCode === 37) {        //left
    offsetX -= 5;
  } else if (event.keyCode === 39) {  //right
    offsetX += 5;
  } else if (event.keyCode === 38) {  //up
    offsetY -= 5;
  } else if (event.keyCode === 40) {  //down
    offsetY += 5;
  }

  points.forEach(function (point) {
    point.setCenter(offsetX, offsetY, 200);
  });
}, false);
```

这个例子可以在文件 12-move-cube-2.html 中找到。现在整个立方体作为一个整体移动，并且围绕着自己的中心旋转，而不是围绕着三维坐标空间的场景中的中心旋转。

16.6 小结

使用本章学到的知识，你可以随意创建自己的三维模型，并在空间中操纵它们。

本章专注于介绍三维几何图形的创建以及如何绘制线条和用颜色填充。在第 17 章中，你将学习如何创建看起来更为结实的实体。第 17 章所需要的基础你都已经奠定了，比如点、线和填充。如果你准备好了，我们就继续吧！



第 17 章　背面剔除与三维灯光

本章涵盖以下内容：

- 背面剔除
- 增强的深度排序（z 排序）
- 三维灯光

在第 16 章中，我们学习了三维实体建模的基础：如何通过创建点、线和多边形来组成一个形状，以及如何为每个多边形着色。但是回想一下，因为把颜色的透明度设置为 50%，所以你可以看透它。虽然创建了一些复杂的三维模型，但是这些模型的真实度仍然有很大的欠缺。

在本章中，我们通过学习如下技术来修复前面的问题：背面剔除（不绘制你看不到的那些多边形）、增强的深度排序（第 15 章介绍过一点，但是我们将在多边形的角度上重新审视它）以及三维灯光。

你肯定会对应用了这些技术的三维效果感到惊讶。前两种技术会让你创建的三维实体看起来更结实。三维灯光使它们更生动。

本章的例子是建立在第 16 章中的旋转、拉伸的三维字母 A 这个例子的基础上。因为这是一个足够复杂的模型，所以一旦做错点什么就会很明显，但是当所有都做得正确时，效果将会非常棒！本章的基础来自于 Todd Yard 的《Macromedia Flash MX Studio》一书中第 10 章的技术。

17.1 背面剔除

第 16 章多次提到了背面剔除，现在我们来一探究竟，弄清楚它是如何工作的。

前面的模型都是用半透明的颜色来填充的。这是因为我们绘制了所有的多边形，并且没有控制它们的绘制顺序。所以模型背面的多边形就有可能会覆盖正面的多边形，这会造成奇怪的效果。当时我们专注于建模技术，所以暂且把每个表面都设置为半透明来绕过这个问题。现在到了该处理它的时候了。

大体上来说，背面剔除比较简单。你只需要绘制面向你的那些多边形，而不绘制背向你的那些多边形。难点是如何区分哪些多边形是面向你的，哪些是背向你的。

我一直在提醒你要按顺时针方向来定义多边形的点，但是到目前为止的例子中还没体现出这有什么用处。接下来你将会看到它为什么如此重要，以及为什么一开始就要养成这个习惯。

这是个很有趣的观察结果：当一个多边形面向你时，它的顶点是按顺时针排列的；当它背向你时，这些顶点将是按逆时针排列的。图 17-1 展示了一个面向你的三角形（本书中的术语提示：大多数情况下，“多边形”是一个通用的术语，而“三角形”是特殊的多边形）。

在图 17-2 中，这个三角形旋转到了相反的方向。

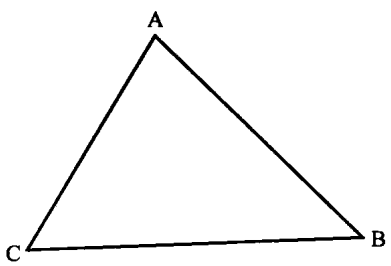


图 17-1 面向你的三角形，顶点按顺时针排列

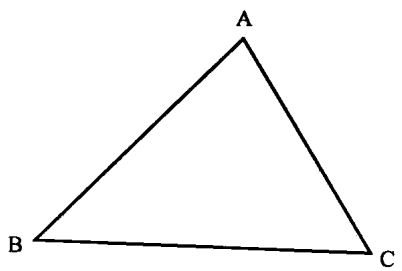


图 17-2 一个背向你的三角形，顶点以逆时针排列

现在我们看到这些顶点以逆时针方向排列。

但是，当一个多边形“面向你”时是什么意思呢？实际上，这表示这个多边形的**外表面朝向你**。尽管当只有一个三角形的时候并没有内外之分，但别忘了，我们是在讨论三维实体。这样一来，每个多边形都有一个外表面和一个内表面。

有时候，你需要绘制三角形的内表面。比如，当观察点位于一个立方体的内部并且向外看，你就要看到内表面——墙面、地面和天花板。但是对于这里的例子都观察一个模型，所以我们想要看到的都是几何体的外表面。

我们使用屏幕坐标来判断一个多边形的顶点是顺时针还是逆时针方向，不是三维 x、y、z 坐标，而是调用 getScreenX()、getScreenY()得到的经过透视计算的 canvas 坐标。

当然也可以把系统设计为正面逆时针，背面顺时针。这都是没有问题的，两种方式都能正常工作。

回到我们的问题：你如何判断三个点是顺时针还是逆时针？如果用眼睛来看，这是一件再简单不过的事了。但是如果用代码来处理的话，这个问题马上就变成了一个抽象的概念。

我们为 Triangle 类添加一个方法，叫做 isBackface。它会根据三角形的三个顶点来计算，如果它们逆时针排列则返回 true，如果它们顺时针排列则返回 false。下面是添加了新方法的 Triangle 类（triangle.js 文件）。注意，透明度设置为 1，因为我们要在本章开始绘制不透明的形状了。

```
function Triangle (a, b, c, color) {
  this.pointA = a;
  this.pointB = b;
  this.pointC = c;
  this.color = (color === undefined) ? "#ff0000" : utils.parseColor(color);
  this.lineWidth = 1;
  this.alpha = 1;
}
Triangle.prototype.draw = function (context) {
  if (this.isBackface()) {
    return;
  }
  context.save();
  context.lineWidth = this.lineWidth;
  context.fillStyle = context.strokeStyle = utils.colorToRGB(this.color, this.alpha);
  context.beginPath();
  context.moveTo(this.pointA.getScreenX(), this.pointA.getScreenY());
  context.lineTo(this.pointB.getScreenX(), this.pointB.getScreenY());
  context.lineTo(this.pointC.getScreenX(), this.pointC.getScreenY());
  context.closePath();
  context.fill();
  if (this.lineWidth > 0) {
    context.stroke();
  }
  context.restore();
};
Triangle.prototype.isBackface = function () {
  var cax = this.pointC.getScreenX() - this.pointA.getScreenX(),
      cay = this.pointC.getScreenY() - this.pointA.getScreenY(),
      bcx = this.pointB.getScreenX() - this.pointC.getScreenX(),
      bcy = this.pointB.getScreenY() - this.pointC.getScreenY();
  return (cax * bcy > cay * bcx);
};
```

快速地解释一下，isBackface 函数计算三角形两条边的长度，然后用乘法和比较，其中涉及三角形相对于 canvas 的法线向量，就能判断出方向。

那么如何使用这个方法呢？你不必去过多考虑。因为这个函数只是在 Triangle.draw 中用到。如果它返回 true，则说明是一个背面的三角形，不必绘制，draw 方法立即停止并返回。如果 isBackface 返回 false，则说明这个三角形是正面的，使用 canvas 绘图 API 正常绘制它。

现在可以运行一下 01-extruded-a.html，或者你自己创建的三维模型，你可以看到效果发生

了一点变化。随着物体的旋转，你会发现一旦某个面旋转到了背面，则不再绘制它。效果并不完美，因为还是有一些远的部分绘制在了近的部分之上，但是我们正在接近完美。如果你的脑海中出现了“z 排序”或“深度排序”这样的字眼，那就对了，这正是我们接下来要学习的。

现在你应该能看到如图 17-3 所示的效果。

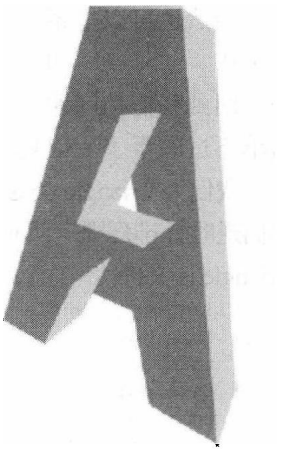


图 17-3 背面剔除的运行效果

17.2 增强的深度排序

深度排序，或者 z 排序，已经在第 15 章中介绍透视图时讨论过了。在那个例子中，依据一个数组中三维物体的 zpos 属性对它们进行排序。

但是现在，你并不是在处理多个物体。因为无论何时绘制一个多边形，它都会绘制在前面的多边形上面。所以这里不交换物体的深度，而是确定多边形的绘制顺序。具体地说，你想要让最远的那个多边形最先绘制，然后由远到近地绘制其他多边形，所以最近的多边形最后绘制，覆盖前面的。

如何实现呢？把所有多边形放入一个名为 triangles 的数组中。当绘制图形时，只须循环这个数组，从第一个绘制到最后一个。所以必须对这个数组进行排序，确保第一个元素是最远的三角形，最后一个元素是最近的三角形。

这与前面对一个数组中的物体进行排序很类似，但是三角形只是由 3 个 Point3d 对象组成的。它并没有一个属性来描述它的深度，然而创建这么一个属性也很简单。事实证明，最合适的值就是三个点的 z 坐标的最小值。换句话说，如果一个三角形的 3 个顶点的深度分别是 200、250 和 300，那么这个三角形的 z 坐标就是 200。

可以用 Math.min 函数来计算三个点的 z 坐标的最小值。为 Triangle 类加入一个新的方法 getDepth 来做这个计算。下面给出这个方法：

```
Triangle.prototype.getDepth = function () {
  return Math.min(this.pointA.z, this.pointB.z, this.pointC.z);
};
```

现在可以对这个三角形对象数组进行排序，确定三角形绘制的先后顺序。同样，要按升序排列，要让最远的那个排在第一个。在 drawFrame 动画循环中，绘制三角形之前要进行这些计算。下面是代码中更新的部分（02-extruded-a-depth.html）：

```
function depth (a, b) {
  return (b.getDepth() - a.getDepth());
}
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);
  angleX = (mouse.y - vpY) * 0.0005;
```

```
  angleY = (mouse.x - vpX) * 0.0005;
  points.forEach(move);
  triangles.sort(depth);
  triangles.forEach(draw);
}());
```

在浏览器中运行这个程序，我们得到了一个完美渲染的实体，如图 17-4 所示。现在你已经取得了真正的进展，下一步将把这个例子变得更加真实！

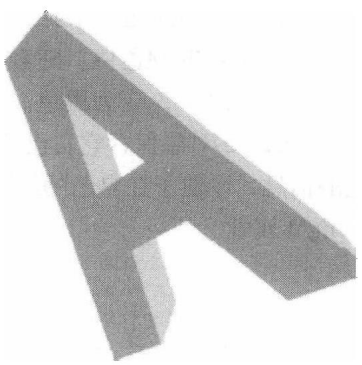


图 17-4 深度排序解决了问题

17.3 三维灯光

尽管刚才的例子用精确的渲染得到了很不错的效果，但是它还缺乏一些真实感，它有点平面的感觉。你应该已经知道我们要做些什么了，让我们加入一些三维灯光。

与背面剔除一样，三维灯光的细节是非常复杂而且需要数学知识的。我不想深入讨论每一个细节点，网上可以找到关于这些公式的大量信息。这里只给出一些基础的函数供你按需使用。

首先，你需要一个光源。最简单的光源有两个属性：位置和亮度。在更复杂的三维系统里，它可能还有照射方向、颜色、衰减率、圆锥区域等其他属性。但是这些都超出了本书的范围。

开始创建一个 Light 类来保存刚才提到的两个属性：位置和亮度。代码如下（文件 light.js）：

```
function Light (x, y, z, brightness) {
  this.x = (x === undefined) ? -100 : x;
  this.y = (y === undefined) ? -100 : y;
  this.z = (z === undefined) ? -100 : z;
  this.brightness = (brightness === undefined) ? 1 : brightness;
}
Light.prototype.setBrightness = function (b) {
  this.brightness = Math.min(Math.max(b, 0), 1);
};
```

现在，在代码里新建默认的灯光，像这样：

```
var light = new Light();
```

或者可以创建特定位置的灯光：

```
var light = new Light(100, 200, 300, 0.5);
```

这里有两个要点需要提到。首先，物体与灯的距离不影响物体的亮度。光从物体上反射到观察点被我们看到，灯的坐标只用来计算光线角度。因为创建的灯的亮度不会随着距离的增大而衰减，所以把 x、y 和 z 的值设置为−1 000 000 或−1 对物体来说亮度都是没有区别的。只有 brightness 属性能改变灯光的亮度值。当然也可以加入这个功能，让亮度随着距离衰减。不过把这个练习留给你自己。

其次，亮度的取值必须在 0.0～1.0 之间。如果超出了这个范围，就会带来一些奇怪的效果。可以尝试其他值，但是在真实的灯光效果里要确保这个取值范围。由于这个原因，用 setBrightness 方法来替换 brightness 属性。这样就能验证传入的参数以确保它处于这个区间内。这称作夹灯（clamping the light）。

现在，要根据光源照射在三角形上的角度来改变它的颜色亮度。如果多边形直接面向灯光，它就显示正常的颜色值。随着它相对光源旋转，它越来越暗。最终，当它背向光源时，它就完全处于阴影之下，变为黑色。

Triangle 对象保存它自己的颜色并且知道如何绘制自己，但同时，每个三角形还需要得到灯光的信息，以便于在自己的绘图函数中进行计算。所以，需要在 Triangle 的构造函数中，添加一个 light 属性。

```
function Triangle (a, b, c, color) {
  this.pointA = a;
  this.pointB = b;
  this.pointC = c;
  this.color = (color === undefined) ? "#ff0000" : utils.parseColor(color);
  this.lineWidth = 1;
  this.alpha = 1;
  this.light = null;
}
```

然后在代码中，为每个三角形设置创建的灯光对象的引用：

```
triangles.forEach(function (triangle) {
  triangle.light = light;
});
```

接下来，每个三角形需要根据它自己的基础色、灯光的角度和亮度，计算出一个调整后的颜色值，下面是这个方法的实现：

```
Triangle.prototype.getAdjustedColor = function () {
  var color = utils.parseColor(this.color, true),
      red = color >> 16,
      green = color >> 8 & 0xff,
      blue = color & 0xff,
      lightFactor = this.getLightFactor();

  red *= lightFactor;
  green *= lightFactor;
  blue *= lightFactor;
  return utils.parseColor(red << 16 | green << 8 | blue);
};
```

这个方法首先把三角形的基础色分离为红色、绿色、蓝色分量（见第 4 章）。使用命令 utils.parseColor（this.color, true）把颜色值从字符串转换为数字。然后调用一个方法 getLightFactor，稍后再解释这个方法，现在你只要知道它返回 0.0～1.0 之间的一个数字。它就是三角形颜色的改变系数，1.0 代表全亮度，0.0 代表黑色。

每个颜色分量都乘以这个系数，然后再组合为一个颜色值。使用 utils.parseColor 把它转换回颜色字符串，并作为调整过的颜色返回。这就是三角形基于灯光的颜色值。

现在，如何得到这个 lightFactor 方法呢？我们来看代码：

```
Triangle.prototype.getLightFactor = function () {
  var ab = {
    x: this.pointA.x - this.pointB.x,
    y: this.pointA.y - this.pointB.y,
    z: this.pointA.z - this.pointB.z
  };
```

```
  var bc = {
    x: this.pointB.x - this.pointC.x,
    y: this.pointB.y - this.pointC.y,
    z: this.pointB.z - this.pointC.z
  };
  var norm = {
    x:  (ab.y * bc.z) - (ab.z * bc.y),
    y:-((ab.x * bc.z) - (ab.z * bc.x)),
    z:  (ab.x * bc.y) - (ab.y * bc.x)
  };
  var dotProd = norm.x * this.light.x +
                norm.y * this.light.y +
                norm.z * this.light.z;
  var normMag = Math.sqrt(norm.x * norm.x +
                          norm.y * norm.y +
                          norm.z * norm.z);
  var lightMag = Math.sqrt(this.light.x * this.light.x +
                           this.light.y * this.light.y +
                           this.light.z * this.light.z);
  return (Math.acos(dotProd / (normMag * lightMag)) / Math.PI) * this.light.brightness;
};
```

既然这个函数比较复杂，为了完全理解它，你就需要具备高级向量数学的知识，不过我们会从基础内容讲起。

首先，你需要找到三角形的法线（normal）。这是一个与三角形表面垂直的向量，这在第 16 章中也提到过，如图 17-5 所示。试想一下，如果你有一块三角形木板，然后用钉子从背后钉穿它。这个钉子就代表这个平面的法线。无论你学习任何关于三维渲染和灯光的内容，都会涉及表面法线。

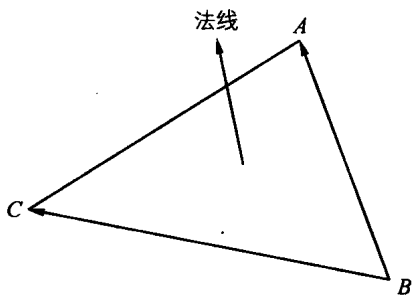


图 17-5 法线垂直于三角形的平面

可以通过计算组成平面的两个向量的叉积（cross product）来得到法线。两个向量的叉积是一个与它们垂直的新向量。这里使用的两个向量是 *A* 与 *B* 以及 *B* 与 *C* 的连线。每个向量都使用一个有 x、y、z 属性的对象来保存。

```
var ab = {
  x: this.pointA.x - this.pointB.x,
  y: this.pointA.y - this.pointB.y,
  z: this.pointA.z - this.pointB.z
};
var bc = {
  x: this.pointB.x - this.pointC.x,
  y: this.pointB.y - this.pointC.y,
  z: this.pointB.z - this.pointC.z
};
```

然后计算法线，这是另一个向量，叫做 norm。下面的代码使用线性代数中的标准公式来计算向量 ab 和 bc 的叉积。

```
var norm = {
  x:   (ab.y * bc.z) - (ab.z * bc.y),
  y: -((ab.x * bc.z) - (ab.z * bc.x)),
  z:   (ab.x * bc.y) - (ab.y * bc.x)
};
```

现在你需要知道法线与灯光的角度。这里又要用到一些向量数学知识：整体（dot product），它表示两个向量之间的差值。已知法线向量和灯光向量，下面计算点积：

```
var dotProd = norm.x * this.light.x +
              norm.y * this.light.y +
              norm.z * this.light.z;
```

我们看到，计算点积比计算叉积更容易一些！

接下来，计算法线的幅值与灯光的幅值，你可以把它看作三维版本的勾股定理：

```
var normMag = Math.sqrt(norm.x * norm.x +
                        norm.y * norm.y +
                        norm.z * norm.z);
var lightMag = Math.sqrt(this.light.x * this.light.x +
                         this.light.y * this.light.y +
                         this.light.z * this.light.z);
```

变量 lightMag 在每次三角形绘制时都会重新计算，这样就可以支持移动的光源。如果可以肯定光源是固定的，就可以在程序创建灯光或者给三角形赋值时只计算一次。或者，也可以为 Light 类添加一个 lightMag 属性。这样就可以在灯光的 x、y 或 z 属性发生变化时重新计算。

最后，把计算出来的这些值填入下面这个神奇的公式：

```
return (Math.acos(dotProd / (normMag * lightMag)) / Math.PI) * this.light.brightness;
```

简单解释一下，dotProd 是一个度量值，(normMag * lightMag)是另一个。两者相除得到一个比率。回忆一下第 3 章中的内容，一个角度的余弦值是一个比率，一个比率的反余弦值是一个角度。所以这里使用 Math.acos 来得到这个比率对应的角度。本质上，这就是灯光照射在多边形表面上的角度。它的范围介于 0～Math.PI 弧度（0°～180°）之间，也就是说，从灯光直射在表面上到完全照射不到。

用这个角度除以 Math.PI 得到一个百分比，然后乘以灯光的亮度值，得到最终的亮度系数，它可以用来改变多边形的颜色。

所有的这些只是为了计算出多边形表面的新颜色！在现有的代码中实现这一点很简单，只要把它加入 Triangle 类的 draw 方法即可。使用调整后的颜色来代替基础色，像这样：

```
context.fillStyle = context.strokeStyle = this.getAdjustedColor();
```

综合上述内容，得到以下完整版的最终代码（文件 triangle.js 和 03-extrudeda-light.html）。

首先是 Triangle 类：

```
function Triangle (a, b, c, color) {
  this.pointA = a;
  this.pointB = b;
  this.pointC = c;
  this.color = (color === undefined) ? "#ff0000" : utils.parseColor(color);
  this.lineWidth = 1;
  this.alpha = 1;
  this.light = null;
}
Triangle.prototype.draw = function (context) {
  if (this.isBackface()) {
    return;
  }
  context.save();
```

```
  context.lineWidth = this.lineWidth;
  context.fillStyle = context.strokeStyle = this.getAdjustedColor();
  context.beginPath();
  context.moveTo(this.pointA.getScreenX(), this.pointA.getScreenY());
  context.lineTo(this.pointB.getScreenX(), this.pointB.getScreenY());
  context.lineTo(this.pointC.getScreenX(), this.pointC.getScreenY());
  context.closePath();
  context.fill();
  if (this.lineWidth > 0) {
     context.stroke();
  }
  context.restore();
};
Triangle.prototype.isBackface = function () {
  var cax = this.pointC.getScreenX() - this.pointA.getScreenX(),
      cay = this.pointC.getScreenY() - this.pointA.getScreenY(),
      bcx = this.pointB.getScreenX() - this.pointC.getScreenX(),
      bcy = this.pointB.getScreenY() - this.pointC.getScreenY();
  return (cax * bcy > cay * bcx);
};
Triangle.prototype.getDepth = function () {
  return Math.min(this.pointA.z, this.pointB.z, this.pointC.z);
};
Triangle.prototype.getAdjustedColor = function () {
  var color = utils.parseColor(this.color, true),
      red = color >> 16,
      green = color >> 8 & 0xff,
      blue = color & 0xff,
      lightFactor = this.getLightFactor();
  red *= lightFactor;
  green *= lightFactor;
  blue *= lightFactor;
  return utils.parseColor(red << 16 | green << 8 | blue);
};
Triangle.prototype.getLightFactor = function () {
  var ab = {
    x: this.pointA.x - this.pointB.x,
    y: this.pointA.y - this.pointB.y,
    z: this.pointA.z - this.pointB.z
  };
var bc = {
     x: this.pointB.x - this.pointC.x,
     y: this.pointB.y - this.pointC.y,
     z: this.pointB.z - this.pointC.z
  };
  var norm = {
     x:  (ab.y * bc.z) - (ab.z * bc.y),
     y:-((ab.x * bc.z) - (ab.z * bc.x)),
     z:  (ab.x * bc.y) - (ab.y * bc.x)
  };
  var dotProd = norm.x * this.light.x +
                norm.y * this.light.y +
                norm.z * this.light.z;
  var normMag = Math.sqrt(norm.x * norm.x +
```

```
                        norm.y * norm.y +
                        norm.z * norm.z);
  var lightMag = Math.sqrt(this.light.x * this.light.x +
                           this.light.y * this.light.y +
                           this.light.z * this.light.z);
  return (Math.acos(dotProd / (normMag * lightMag)) / Math.PI) *
 this.light.brightness;
};
```

下面是示例 03-extruded-a-light.html 的代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Extruded A Light</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="point3d.js"></script>
  <script src="triangle.js"></script>
  <script src="light.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        mouse = utils.captureMouse(canvas),
        light = new Light(),
        points = [],
        triangles = [],
        fl = 250,
        vpX = canvas.width / 2,
        vpY = canvas.height / 2,
        angleX, angleY; //referenced in drawFrame and move

    //first set
    points[0]  = new Point3d( -50, -250, -50);
    points[1]  = new Point3d(  50, -250, -50);
    points[2]  = new Point3d( 200,  250, -50);
    points[3]  = new Point3d( 100,  250, -50);
    points[4]  = new Point3d(  50,  100, -50);
    points[5]  = new Point3d( -50,  100, -50);
    points[6]  = new Point3d(-100,  250, -50);
    points[7]  = new Point3d(-200,  250, -50);
    points[8]  = new Point3d(   0, -150, -50);
    points[9]  = new Point3d(  50,    0, -50);
    points[10] = new Point3d( -50,    0, -50);

    //second set
    points[11] = new Point3d( -50, -250,  50);
    points[12] = new Point3d(  50, -250,  50);
    points[13] = new Point3d( 200,  250,  50);
    points[14] = new Point3d( 100,  250,  50);
```

```
points[15] = new Point3d(  50,  100,  50);
points[16] = new Point3d( -50,  100,  50);
points[17] = new Point3d(-100,  250,  50);
points[18] = new Point3d(-200,  250,  50);
points[19] = new Point3d(   0, -150,  50);
points[20] = new Point3d(  50,    0,  50);
points[21] = new Point3d( -50,    0,  50);
points.forEach(function (point) {
  point.setVanishingPoint(vpX, vpY);
  point.setCenter(0, 0, 200);
});

triangles[0]  = new Triangle(points[0],  points[1],  points[8],  "#cccccc");
triangles[1]  = new Triangle(points[1],  points[9],  points[8],  "#cccccc");
triangles[2]  = new Triangle(points[1],  points[2],  points[9],  "#cccccc");
triangles[3]  = new Triangle(points[2],  points[4],  points[9],  "#cccccc");
triangles[4]  = new Triangle(points[2],  points[3],  points[4],  "#cccccc");
triangles[5]  = new Triangle(points[4],  points[5],  points[9],  "#cccccc");
triangles[6]  = new Triangle(points[9],  points[5],  points[10], "#cccccc");
triangles[7]  = new Triangle(points[5],  points[6],  points[7],  "#cccccc");
triangles[8]  = new Triangle(points[5],  points[7],  points[10], "#cccccc");
triangles[9]  = new Triangle(points[0],  points[10], points[7],  "#cccccc");
triangles[10] = new Triangle(points[0],  points[8],  points[10], "#cccccc");
triangles[11] = new Triangle(points[11], points[19], points[12], "#cccccc");
triangles[12] = new Triangle(points[12], points[19], points[20], "#cccccc");
triangles[13] = new Triangle(points[12], points[20], points[13], "#cccccc");
triangles[14] = new Triangle(points[13], points[20], points[15], "#cccccc");
triangles[15] = new Triangle(points[13], points[15], points[14], "#cccccc");
triangles[16] = new Triangle(points[15], points[20], points[16], "#cccccc");
triangles[17] = new Triangle(points[20], points[21], points[16], "#cccccc");
triangles[18] = new Triangle(points[16], points[18], points[17], "#cccccc");
triangles[19] = new Triangle(points[16], points[21], points[18], "#cccccc");
triangles[20] = new Triangle(points[11], points[18], points[21], "#cccccc");
triangles[21] = new Triangle(points[11], points[21], points[19], "#cccccc");
triangles[22] = new Triangle(points[0],  points[11], points[1],  "#cccccc");
triangles[23] = new Triangle(points[11], points[12], points[1],  "#cccccc");
triangles[24] = new Triangle(points[1],  points[12], points[2],  "#cccccc");
triangles[25] = new Triangle(points[12], points[13], points[2],  "#cccccc");
triangles[26] = new Triangle(points[3],  points[2],  points[14], "#cccccc");
triangles[27] = new Triangle(points[2],  points[13], points[14], "#cccccc");
triangles[28] = new Triangle(points[4],  points[3],  points[15], "#cccccc");
triangles[29] = new Triangle(points[3],  points[14], points[15], "#cccccc");
triangles[30] = new Triangle(points[5],  points[4],  points[16], "#cccccc");
triangles[31] = new Triangle(points[4],  points[15], points[16], "#cccccc");
triangles[32] = new Triangle(points[6],  points[5],  points[17], "#cccccc");
triangles[33] = new Triangle(points[5],  points[16], points[17], "#cccccc");
triangles[34] = new Triangle(points[7],  points[6],  points[18], "#cccccc");
triangles[35] = new Triangle(points[6],  points[17], points[18], "#cccccc");
triangles[36] = new Triangle(points[0],  points[7],  points[11], "#cccccc");
triangles[37] = new Triangle(points[7],  points[18], points[11], "#cccccc");
triangles[38] = new Triangle(points[8],  points[9],  points[19], "#cccccc");
triangles[39] = new Triangle(points[9],  points[20], points[19], "#cccccc");
triangles[40] = new Triangle(points[9],  points[10], points[20], "#cccccc");
triangles[41] = new Triangle(points[10], points[21], points[20], "#cccccc");
```

```
    triangles[42] = new Triangle(points[10], points[8],  points[21], "#cccccc");
    triangles[43] = new Triangle(points[8],  points[19], points[21], "#cccccc");

    triangles.forEach(function (triangle) {
      triangle.light = light;
    });
    function move (point) {
      point.rotateX(angleX);
      point.rotateY(angleY);
    }
    function depth (a, b) {
      return (b.getDepth() - a.getDepth());
    }
    function draw (triangle) {
      triangle.draw(context);
    }
    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      angleX = (mouse.y - vpY) * 0.0005;
      angleY = (mouse.x - vpX) * 0.0005;

      points.forEach(move);
      triangles.sort(depth);
      triangles.forEach(draw);
    }());
  };
  </script>
 </body>
</html>
```

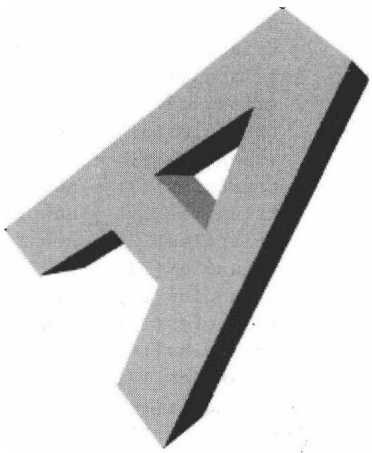


图 17-6 应用了背面剔除、深度排序和三维灯光的三维实体

可以看到，主干代码只有很少的变化，大部分工作集中在 Triangle 类中。然而，现在让所有的三角形颜色都相同，这样就能更好地表现出光照效果（如图 17-6 所示）。

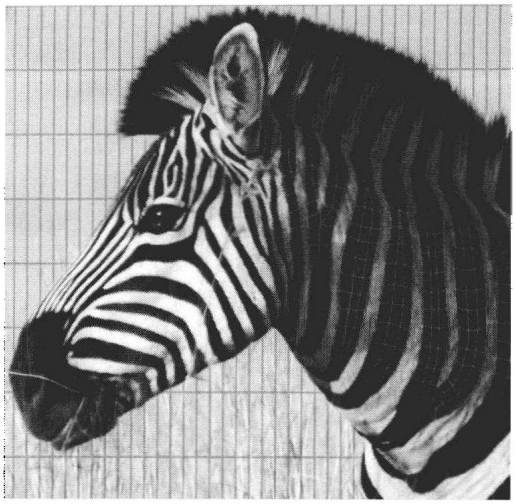
17.4 小结

几页篇幅却介绍了大量的内容！结果令人吃惊。现在你已经有了制作绝妙三维动画的工具。你可以创造出很多变种，比如，在上例中，光源是固定的，物体是移动的。试着去让灯光环绕物体运动（只不过就是改变它的 x、y、z 位置）。

关于三维的讨论就到此结束了。在第 18 章，你将会学习矩阵数学，这通常用来替代前面用过的缩放（scaling）和旋转（rotation）方法，因此你经常会在三维程序设计中看到它。

第五部分

其 他 技 巧



第 18 章　矩阵数学

本章涵盖以下内容：

- 矩阵基础
- 矩阵运算
- canvas 变换

本章不再介绍新的运动或图形渲染方法，而会介绍**矩阵**，它为视觉变换的应用提供另一套解决方案，并大量出现在本书的示例中。

矩阵被大量用于 3D 系统中，以实现旋转、缩放以及平移（移动）3D 坐标的功能。它也常用于各种 2D 图形的变换。在本章中大家将看到如何创建一个矩阵系统用于操纵动画中的物体，同时也会介绍 canvas 中那些内置的与矩阵相关的方法。

18.1　矩阵基础

矩阵最简单的定义就是一个数字表格。它可以由一个或多个水平行以及一个或多个垂直列构成。图 18-1 展示了一些矩阵。

矩阵通常由某个变量描述，如 M。为了获得矩阵中某个特定单元格的值，可以使用该单元格的行列值作为下标访问变量名。假设图 18-1 中的 3 × 3 矩阵以变量名 M 表示，那么通过行列下标表示的 $M_{2,3}$ 的值则为 6，因为它指向矩阵中第二行、第三列的单元格。矩阵的下标都是从

1 开始计数，它不同于 JavaScript 中的数组下标，后者是从 0 开始计数。

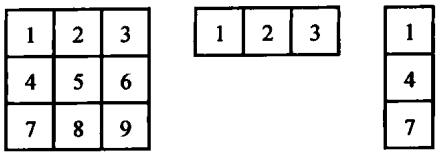


图 18-1 一个 3×3 的矩阵，一个 1×3 矩阵以及一个 3×1 的矩阵，都是经由行列标识的

矩阵的单元格不仅包含数字，还可以由公式和变量组成。如果你曾经使用过电子表格，你会发现它就是一个放大的矩阵。可以用一个单元格保存某一列的总和，再用另一个单元格保存这个总和与另一个单元格中数字的乘积，等等。由此，你可以想象到矩阵能够发挥的作用。

18.2 矩阵运算

电子表格就像一个自由组合的矩阵，而矩阵的结构性更强，并伴随着大量的使用规则。

矩阵数学的教学主要包含两种方法。第一种方法会以填满随机数的各种矩阵为基础，介绍各种矩阵操作的细节。在这个过程中，你学习各种运算规则，不过你并不清楚为什么要这么做以及这些运算的结果代表什么，就像在玩一个如何把数字排列成漂亮形状的游戏。

第二种方法则着重于矩阵的内容而简单略过操作步骤。使用诸如“将两个矩阵相乘而得到这个……”这样含糊的描述，以至于读者完全搞不清那些操作是如何完成的。

本章会采取一个折中的方案，会先介绍一些包含有意义的数值的矩阵，然后再描述如何操作它们。

18.2.1 矩阵加法

矩阵的一个常见用途是操作 3D 空间中的点，这样一个点分别包含 x、y 与 z 轴上的坐标。可以将其简单地视为一个 1×3 的矩阵。

```
x y z
```

为了实现点在空间中的移动，也称为点的平移，需要知道它在每条轴上的移动距离。可以将每条轴上的移动距离填充到一个平移矩阵中，就像下面这个 1×3 的矩阵：

```
dx dy dz
```

这里，dx、dy 与 dz 分别为 x、y 与 z 轴上的移动距离。现在要通过矩阵加法将平移矩阵作用于 3D 点上。只须将每个对应的单元格的数值加在一起就可以创造出一个包含每个单元格之和的新矩阵。只有两个同样大小的矩阵才能相加。点的平移如下所示：

```
x y z + dx dy dz = (x + dx) (y + dy) (z + dz)
```

新的矩阵可以叫做 x1、y1、z1，它包含经过平移后的点的新坐标。假设有一个点在 x、y、z 轴上的坐标为 100、50、75，要让它在 x、y、z 轴上分别移动−10、20、−35，则它将如下所示：

```
100 50 75 + -10 20 -35 = (100 - 10) (50 + 20) (75 - 35)
```

因此，在完成加法运算后，获得了点的新坐标 90、70、40。你可能联想到了它与速度很相似，x、y、z 轴上的速度也是分别追加到每条轴所对应的坐标位置上。它们的计算过程本质上是一样的，只不过视角略微不同而已。

如果你有一个较大的矩阵，你还是可以继续使用同样的方法，只要注意匹配每个单元格。尽

管在示例中是不会出现大于 1×3 的矩阵，但是为了便于大家理解，下面提供一个抽象的例子：

```
a b c     j k l     (a + j) (b + k) (c + l)
d e f  +  m n o  =  (d + m) (e + n) (f + o)
g h i     p q r     (g + p) (h + q) (i + r)
```

以上就是我们需要了解的有关矩阵加法的所有知识。在介绍完矩阵乘法之后，你会看到如何在一个基于矩阵的 3D 引擎中运用各种函数。

18.2.2 矩阵乘法

矩阵乘法是 3D 转化的计算中更为常用的一种方法，它通常用于缩放与旋转。本书不会涉及 3D 缩放，因为在所有的示例中，只出现了点和形状，前者完全没有缩放的概念，后者则缺少 3D 中的厚度元素，因此只有二维的缩放。当然，大家可以建立一个更为复杂的引擎使其支持 3D 实体的缩放，不过这就需要我们编写额外的函数实现 3D 实体上各个点的转换，从而改变实体的大小。而这些已经超出了本章所要讨论的范围，不过由于缩放操作是用于演示矩阵乘法的一个非常好的示例，下面还是通过一个例子演示一下。

使用矩阵进行缩放

首先，需要知道一个物体现有的宽度、高度与深度，换句话说，也就是它在三条轴上每个分量的大小。通过它们创建出一个 1×3 的矩阵：

```
w h d
```

这里的 w、h 与 d 分别代表宽度、高度与深度。然后，要用到像下面这样的一个缩放矩阵：

```
sx  0   0
0   sy  0
0   0   sz
```

在这个矩阵中，sx、sy 与 sz 分别为对应轴上的缩放比例。它们都是以分数或小数的形式出现，1.0 表示 100%，2.0 表示 200%，0.5 则表示 50%，等等。稍后大家会看到为什么这个矩阵中的变量以此种形式分布。

矩阵乘法有一个必要条件，第一个矩阵的列数必须等同于第二个矩阵的行数，只要符合这个标准，无论第一个矩阵有多少行，第二个矩阵有多少列，它们都可以相乘，否则它们无法进行乘法运算。在本例中，这个相乘的条件是满足的，因为第一个矩阵有三列（w, h, d），而缩放矩阵的行数又恰恰为 3。

那么两个矩阵如何相乘呢？让我们先将结果计算出来，你再看看能否从中找出规律：

```
                sx  0   0
w h d *         0   sy  0
                0   0   sz
```

计算结果如下：

```
(w * sx + h * 0 + d * 0) (w * 0 + h * sy + d * 0) (w * 0 + h * 0 + d * sz)
```

移除所有等于 0 的数字，结果如下：

```
(w * sx) (h * sy) (d * sz)
```

以上的计算结果很符合逻辑，因为我们将宽度乘以 x 轴上的缩放系数，将高度乘以 y 轴上的缩放系数，最终将深度乘以 z 轴上的缩放系数。那些等于 0 的数字让我们迷失了真相，让我

将这一运算进一步抽象，以便让计算的过程变得更加清晰：

```
          a b c
u v w *   d e f
          g h i
```

现在你可以从结果中发现其中的规律：

```
(u * a + v * d + w * g) (u * b + v * e + w * h) (u * c + v * f + w * i)
```

我们将第一个矩阵（u,v,w）的第一行中的每个元素与第二个矩阵中的每一行的第一个元素相乘，将它们相乘的结果相加就得到了结果矩阵中的第一行的第一个元素的值。对第二个矩阵的第二列（b,e,h）使用相同的方法，就可以计算出第二列的结果。

如果第一个矩阵的行数大于 1，对第二行重复以上计算，就可以得到新矩阵的第二行的结果：

```
u v w         a b c
x y z   *     d e f
              g h i
```

可以得到一个 2×3 的矩阵：

```
(u * a + v * d + w * g) (u * b + v * e + w * h) (u * c + v * f + w * i)
(x * a + y * d + z * g) (x * b + y * e + z * h) (x * c + y * f + z * i)
```

矩阵相乘后得到的新矩阵的大小的行列数分别由第一个矩阵的行数和第二个矩阵的列数决定。

现在让我们介绍可以让大家看到实际例子的矩阵乘法的应用——坐标旋转。希望可以通过这个例子让大家更加清楚乘法操作。

使用矩阵进行坐标旋转

首先，我们将再次用到表示 3D 空间中一个点的矩阵：

```
x y z
```

它持有待旋转点的三维坐标。现在，我们需要一个旋转矩阵，通过它我们可以在三条轴中任意一条轴上进行旋转。我们将分别为每种类型的旋转创建一个矩阵。先从 *x* 轴的旋转矩阵开始：

```
1    0     0
0    cos   sin
0    -sin  cos
```

该矩阵包含一些正余弦值。不过这里的正余弦又是针对哪个角度而言呢？很明显，它就是所要旋转角度的正余弦值。如果你想让这个点旋转 45° ，它们就是 45° 角的正余弦的值（当然，在代码中要使用弧度制）。

现在，让我们用这个旋转矩阵乘以代表 3D 点的坐标矩阵，看一下结果：

```
          1    0     0
x y z  *  0    cos   sin
          0    -sin  cos
```

计算得出：

```
(x * 1 + y * 0 + z * 0) (x * 0 + y * cos - z * sin) (x * 0 + y * sin + z * cos)
```

整理后结果如下：

```
(x) (y * cos - z * sin) (z * cos + y * sin)
```

与之对应的 JavaScript 代码如下：

```
x = x;
y = y * Math.cos(rotation) - z * Math.sin(rotation);
z = z * Math.cos(rotation) + y * Math.sin(rotation);
```

回忆一下 15.7 节，你会看到这里的方法跟当时介绍的围绕 x 轴的旋转方法完全一致。这并没有什么值得惊奇的，因为矩阵数学仅仅是组织各种公式与方程的另一种方法而已。

至此，要创建一个围绕 y 轴旋转的矩阵就非常容易了：

```
cos   0   sin
0     1   0
-sin  0   cos
```

最后，让我们看一下围绕 z 轴旋转的矩阵：

```
cos   sin   0
-sin  cos   0
0     0     1
```

请尝试将它们分别乘以 x、y、z 坐标矩阵，再验证一下所获得的公式与第 15 章中用于两条轴上的坐标旋转公式是否完全一致，这将是一次很有意义的实践。

矩阵编程

现在我们已经掌握了足够多的基础知识，可以开始使用矩阵编写代码了。这里会重用并改写第 15 章中的示例，先打开 13-rotate-xy.html 文件。该文件包含用于实现 3D 坐标旋转的 rotateX 与 rotateY 函数。我们会用矩阵改写这些函数实现同样的功能。

让我们先从 rotateX 函数开始，更新后的代码会将球体的 x、y、z 坐标放入一个 1×3 的矩阵，随后根据指定的角度创建一个围绕 x 轴旋转的旋转矩阵，该矩阵以数组形式表现。然后，借助 matrixMultiply 函数实现两个矩阵的相乘，最终得到的结果也会是一个数组，再将其中的值赋回给球体的 x、y、z 坐标。其中的 matrixMultiply 函数需要由我们自己实现。下面是新版 rotateX 函数的代码：

```
function rotateX (ball, angle) {
  var position = [ball.xpos, ball.ypos, ball.zpos],
      sin = Math.sin(angle),
      cos = Math.cos(angle),
      xRotMatrix = [];

  xRotMatrix[0] = [1,   0,   0];
  xRotMatrix[1] = [0,  cos,  sin];
  xRotMatrix[2] = [0, -sin,  cos];

  var result = matrixMultiply(position, xRotMatrix);
  ball.xpos = result[0];
  ball.ypos = result[1];
  ball.zpos = result[2];
}
```

下面是实现两个矩阵相乘的函数：

```
function matrixMultiply (matrixA, matrixB) {
  var result = [];
  result[0] = matrixA[0] * matrixB[0][0] +
              matrixA[1] * matrixB[1][0] +
              matrixA[2] * matrixB[2][0];
  result[1] = matrixA[0] * matrixB[0][1] +
              matrixA[1] * matrixB[1][1] +
              matrixA[2] * matrixB[2][1];
  result[2] = matrixA[0] * matrixB[0][2] +
              matrixA[1] * matrixB[1][2] +
```

```
            matrixA[2] * matrixB[2][2];
  return result;
}
```

该函数硬编码为只支持 1×3 的矩阵与 3×3 的矩阵相乘，因为我们只需要对这两个类型的矩阵做乘法。可以使用 for 循环创建出一个更加通用的函数，使其能够适用于任意大小的矩阵。不过出于实用角度考虑，这里先简单写成这样。

最后，改写 rotateY 函数，如果你已经完全理解了 rotateX 函数的实现，那么你会发现这两个函数的实现将非常类似。只须创建一个围绕 *y* 轴旋转的旋转矩阵即可。

```
function rotateY (ball, angle) {
  var position = [ball.xpos, ball.ypos, ball.zpos],
      sin = Math.sin(angle),
      cos = Math.cos(angle),
      yRotMatrix = [];

  yRotMatrix[0] = [ cos, 0, sin];
  yRotMatrix[1] = [   0, 1,   0];
  yRotMatrix[2] = [-sin, 0, cos];

  var result = matrixMultiply(position, yRotMatrix);
  ball.xpos = result[0];
  ball.ypos = result[1];
  ball.zpos = result[2];
}
```

将改写后的示例另存为 01-rotate-xy.html 文件并在你的浏览器中运行，与第 15 章中的版本相比，它们的运行结果看上去完全一致。你甚至还可以再创建一个 rotateZ 函数，不过由于本例并不需要用到，因此我们还是将其作为练习留给读者自己完成。

即便你无须为实现 3D 动画而使用矩阵，你也会在其他场合发现它的用途。下一节会对它展开介绍。不过矩阵在本节中的用途是一个很好的引子，因为你可以将它与你之前熟悉的公式进行对比。同时，由于矩阵还广泛用于其他图形环境，因此当你的程序生涯开始接触那些环境时，你就有了比较好的准备。

18.3 canvas 变换

矩阵的另一个重要功能是用于操纵 canvas 上显示的图形。通过应用一个变换矩阵，可以实现图形的旋转、缩放以及平移，进而改变它们的形状、大小与位置。canvas 上下文在内部使用像下面这样一个 3×3 的变换矩阵：

```
a c dx
b d dy
u v w
```

该变换也称为仿射变换，这意味着，为了能应用仿射变换，二维向量（x,y）需要改写为三维向量（x,y,1）。由于（u,v,w）并不会用到，它们会直接设置为（0,0,1），并保持不变，所以你不用管它们。

可以通过调用以下函数设置 canvas 上下文的变换矩阵：

```
context.setTransform(a, b, c, d, dx, dy);
```

要将当前的 canvas 上下文中的变换矩阵再乘上一个新的变换矩阵，可以调用以下函数：

```
context.transform(a, b, c, d, dx, dy);
```

如果矩阵中的元素都保存在一个数组中，可以方便地将该数组作为参数传入：

```
context.transform.apply(context, [a, b, c, d, dx, dy]);
```

如果没有为 canvas 设置任何变换矩阵，那么 canvas 会认为我们使用了一个单位矩阵（identity matrix）或一个空矩阵，就是类似下面这样一个矩阵：

```
1 0 0
0 1 0
0 0 1
```

为 canvas 上下文应用该矩阵不会产生任何变换。所以，每当你希望重置 canvas 上下文时，可以将它的变换矩阵设置为单位矩阵，如下所示：

```
context.setTransform(1, 0, 0, 1, 0, 0);
```

让我们回到变换矩阵中的那些字母元素，我们还没介绍它们的含义。dx 与 dy 控制 canvas 上下文将要在 x 与 y 轴上平移的距离——记住，坐标（0,0）位于 canvas 的左上角。而矩阵中的 a、b、c 与 d 则有点复杂，它们之间的联系非常紧密。如果将 b 与 c 设为 0，则可以借助 a 与 d 实现物体在 x 轴与 y 轴上的缩放。而如果将 a 与 d 设为 1，则可以通过 b 与 c 让物体在 y 轴与 x 轴上倾斜。甚至可以将 a、b、c 与 d 联合起来设置成下面这个我们熟悉的矩阵：

```
cos  -sin  dx
sin   cos  dy
 u     v   w
```

这里包含一个旋转矩阵。很容易想到，这里的 cos 与 sin 代表 canvas 上下文将要旋转的角度（以弧度为单位）的余弦和正弦值。让我们在下面这个示例（文件 02-matrix-rotate.html）中尝试一下这个矩阵，该示例在动画循环中绘制一个红色的盒子。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Matrix Rotate</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        angle = 0;

    (function drawFrame () {
      window.requestAnimationFrame(drawFrame, canvas);
      context.clearRect(0, 0, canvas.width, canvas.height);

      angle += 0.03;

      var cos = Math.cos(angle),
```

```
            sin = Math.sin(angle),
            dx = canvas.width / 2,
            dy = canvas.height / 2;

        context.save();
        context.fillStyle = "#ff0000";
        context.transform(cos, sin, -sin, cos, dx, dy);
        context.fillRect(-50, -50, 100, 100);
        context.restore();
      }());
    };
    </script>
  </body>
</html>
```

在示例中有一个逐帧递增的角度变量，它的正弦和余弦值会作为旋转矩阵的值传入 context.transform 方法。同时，根据 canvas 的宽度和高度对矩形做了平移，使其处在居中的位置。试着运行该示例，你会得到一个不断旋转的盒子。

在 drawFrame 函数中，我们将绘制矩形的代码包装在对 context.save 与 context.restore 两个方法的调用中。这两个方法分别用于保存及恢复 canvas 的绘图状态，也就是应用在 canvas 上的所有样式与变换矩阵。通过它们，可以借助于将变换矩阵出栈入栈实现遍历多个变换矩阵的功能。还可以嵌套多个 save 与 restore 调用，你只须牢记你正在使用的变换矩阵。而当需要重置 canvas 上下文时，将 canvas 的变换矩阵设为单位矩阵即可。

下面演示一个更加实用的变换——倾斜。倾斜是将物体沿着某条轴拉伸使得物体的两端沿着两个相反的方向运动。试着想象一下斜体字以及它们倾斜的方式，这就是一个很好的例子。斜体字的头部向右延伸，尾部则向左伸展。这种变换要想通过某个公式实现是非常复杂的，而借助于变换矩阵就即刻变容易了。将矩阵中的 a 与 d 设为 1，剩下的 b 可用于指定物体在 y 轴上倾斜的程度，而 c 则可用于指定物体在 x 轴上的倾斜程度。让我们先试着让物体沿着 x 轴倾斜。在文件 03-skew-x.html 中，我们继续沿用上一个示例中的设置，不过为了使用鼠标坐标，请确保在脚本的头部导入相应的工具函数：

```
var mouse = utils.captureMouse(canvas);
```

现在改变该矩阵应用于 drawFrame 函数的方式：

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);
  context.clearRect(0, 0, canvas.width, canvas.height);

  var skewX = (mouse.x - canvas.width / 2) * 0.01,
      dx = canvas.width / 2,
      dy = canvas.height / 2;
  context.save();
  context.fillStyle = "#ff0000";
  context.transform(1, 0, skewX, 1, dx, dy);
  context.fillRect(-50, -50, 100, 100);
  context.restore();
}());
```

这里 skewX 变量是由鼠标的 x 轴坐标决定的，将这个坐标减去 canvas 在 x 轴上的中心坐

标，再乘以 0.01，让倾斜的值处在一个可控的范围内，再将其作为参数传给 context.transform 方法。试着运行该示例，你会发现整个图形倾斜了，如图 18-2 所示。

在示例 04-skew-xy.html 中，针对 *y* 轴做了同样的事情：

```
(function drawFrame () {
   window.requestAnimationFrame(drawFrame, canvas);
   context.clearRect(0, 0, canvas.width, canvas.height);

   var skewX = (mouse.x - canvas.width / 2) * 0.01,
       skewY = (mouse.y - canvas.height / 2) * 0.01,
       dx = canvas.width / 2,
       dy = canvas.height / 2;

   context.save();
   context.fillStyle = "#ff0000";
   context.transform(1, skewY, skewX, 1, dx, dy);
   context.fillRect(-50, -50, 100, 100);
   context.restore();
}());
```

图 18-3 展示了在两条轴倾斜的矩形。

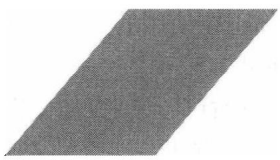


图 18-2 沿 *x* 轴倾斜的矩形

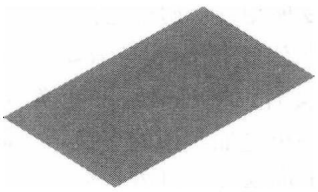


图 18-3 沿两条轴同时倾斜的矩形

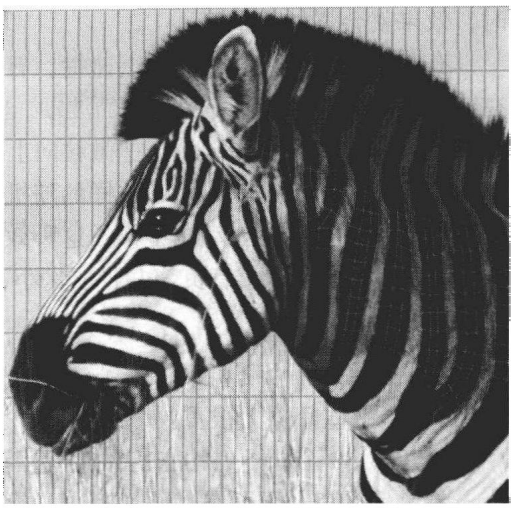
如此简单就可以实现这样的效果的确让人惊喜。倾斜效果经常用于实现伪 3D，尝试在上一个例子中移动鼠标指针，当图形开始倾斜并旋转时，你会发现图形开始展现出透视效果。这虽然不是真正意义上的 3D，不过它已经足以用在一些令人信服的特效中。

18.4 小结

矩阵是一件非常强大的工具，你可以在计算机图形学的各种应用中找到它的身影。它广泛应用于计算机视觉过滤器、图像处理（比如边缘检测）、锐化以及模糊变换。随着你不断地深入到更加高级的计算机图形编程中，你会发现更多有关矩阵的应用。

我们已经介绍了矩阵的基本概念以及如何使用与合并它们，我们还在本章中利用它们创建了一些非常酷的效果。借助于你脑袋里的这些概念，你已经具备了发挥矩阵强大功能的能力，希望当你遇到那些场景时勇敢地利用它。

第 19 章会为本书收尾并介绍一些其他的要诀和技巧。涉及的主题有：随机运动、随机分布以及如何为动画加入声音，它们可以使动画变得更加精彩。



第 19 章　秘诀与技巧

本章涵盖以下内容：

- 布朗（随机）运动
- 随机分布
- 基于定时器和基于时间的动画
- 等质量物体之间的碰撞
- 集成声音

你已经读到了最后一章。本章会介绍一些小知识点，因为它们过于随机和零碎（尽管非常有用），所以不太适合放在其他任何一章。本章中每一节都是一个独立的单元。

19.1　布朗（随机）运动

一天，一个名叫罗伯特·布朗（Robert Brown）的植物学家在观察水滴中的花粉时发现它们在随机运动。即使水没有流动或运动，这些小微粒也永远不会停下来。他发现粉尘也是如此，所以他推论花粉是自己在运动。尽管他并不知道为什么会产生这种现象，科学界在随后几十年内也没有给出解释。这个现象用他的名字来命名，只是因为他注意到了这个现象。

布朗运动的科学解释是：虽然水看起来是静止的，但是一滴水中有无数水分子，它们在不断地运动。一些分子与花粉或灰尘发生碰撞，这样就会把动量传递给它们。因为即使一个小小的灰尘都会比单个水分子重百万倍，所以每次碰撞的效果很小，但是当一秒钟有数百万次碰撞

时，效果就会累加起来。

有些分子可能在某个方向上碰撞，有些在其他方向。总体来说，它们最终达到平衡。但是在特定时间内，你可以看到它们的波动。当左边碰撞的分子多一些时，会把微粒向右推移一点。然后下面的碰撞多一些，又会把微粒向上推移。最终都会达到平衡，所以不会在某个方向上累积太多的动量，于是就产生了这样的随机漂浮运动。

我们可以很容易在动画中模拟这种效果。在每一帧中，计算随机数并累加在移动物体的 x、y 速度上。随机数有正有负，并且非常小，比如，在-0.1～0.1 之间。可以这样做：

```
vx += Math.random() * 0.2 -0.1;
vy += Math.random() * 0.2 - 0.1;
```

用 0.2 乘以一个随机数，得到 0.0～0.2 之间的一个数字，然后减去 0.1，让它落在-0.1～0.1 之间。加入一些摩擦力很重要，否则速度将会累积，产生不自然的效果。在例 01-brownian-1.html 中，创建了 50 个粒子，让它们做布朗运动。这些粒子是我们很熟悉的 Ball 类的实例，半径很小，颜色为黑色。下面是代码：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Brownian 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        dots = [],
        numDots = 50,
        friction = 0.95;
    for (var dot, i = 0; i < numDots; i++) {
        dot = new Ball(1, "#000000");
        dot.x = Math.random() * canvas.width;
        dot.y = Math.random() * canvas.height;
        dot.vx = 0;
        dot.vy = 0;
        dots.push(dot);
    }

    function draw (dot) {
      dot.vx += Math.random() * 0.2 - 0.1;
      dot.vy += Math.random() * 0.2 - 0.1;
      dot.x += dot.vx;
      dot.y += dot.vy;
      dot.vx *= friction;
      dot.vy *= friction;
```

```
      if (dot.x > canvas.width) {
        dot.x = 0;
      } else if (dot.x < 0) {
        dot.x = canvas.width;
      }
      if (dot.y > canvas.height) {
        dot.y = 0;
      } else if (dot.y < 0) {
        dot.y = canvas.height;
      }

      dot.draw(context);
    }

    (function drawFrame () {
       window.requestAnimationFrame(drawFrame, canvas);
       context.clearRect(0, 0, canvas.width, canvas.height);

       dots.forEach(draw);
    }());
  };
  </script>
 </body>
</html>
```

因为大多数内容都是老知识，所以我们把重要的部分用粗体标出。运行结果如图 19-1 所示。

在下一个例子（02-brownian-2.html）中，我们在上例的基础上稍作修改，让每个粒子的运动轨迹显示出来。把粒子数目减少为 20 个，添加两个变量 decay 和 decayColor，作为新的 canvas 清理色：

```
var numDots = 20,
    decay = 0.01,
    decayColor = utils.colorToRGB("#ffffff", decay);
```

因为清理色包含透明度，所以我们用工具函数 utils.colorToRGB 把它转换为 CSS 风格的 RGBA 字符串：'rgba（255,255,255,0.01）'。

然后以这个清理色绘制一个矩形来替换正常的 canvas 清理函数 context.clearRect。现在每一帧中都不会擦除粒子的运动轨迹了，它只会逐步地让图形越来越淡。

```
(function drawFrame () {
  window.requestAnimationFrame(drawFrame, canvas);

  context.fillStyle = decayColor;
  context.fillRect(0, 0, canvas.width, canvas.height);

  dots.forEach(draw);
}());
```

在浏览器中运行该示例，你就能看到每个粒子的运动轨迹。如图 19-2 所示，轨迹的活动部分是黑色的，其余部分渐变为灰色。

如果你想让物体做无意识、无外力、无方向的随机运动，布朗运动是很有用的。也可以把它应用在其他类型的运动上以增加随机感。比如，蜜蜂或苍蝇飞行的效果，你可以让它们沿着一定的轨迹飞行，但是加入一些随机运动会使它看起来更加栩栩如生。

图 19-1 布朗运动

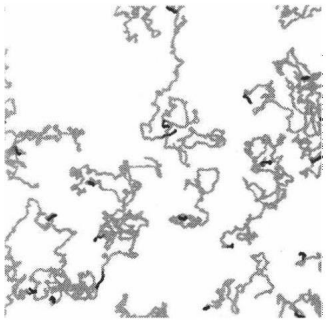


图 19-2 布朗运动的轨迹

19.2 随机分布

有时，你需要创建一些物体并把它们随机放置。在本书中已经多次出现这样的例子，但是这里要用一些不同的方法来解决这个问题，并且会有一些不同的效果。

19.2.1 正方形分布

如果你想让物体随机分布在整个 canvas 上，很简单。选择一个在 canvas 宽度之内的随机数作为 x，以及一个在 canvas 高度之内的随机数作为 y。实际上，在前面的例子中你已经这么做过了：

```
dot.x = Math.random() * canvas.width;
dot.y = Math.random() * canvas.height;
```

但是如果你想让这些小圆点聚集在 canvas 的中心附近，比如，距离中心点上下左右各 100 像素的范围内。你可以像下面这么做（示例 03-random-1.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Random 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        numDots = 50;

    while (numDots--) {
      var x = canvas.width / 2 + Math.random() * 200 - 100,
          y = canvas.height / 2 + Math.random() * 200 - 100;

      //draw circle...
```

```
        context.fillStyle = "#000000";
        context.beginPath();
        context.arc(x, y, 2, 0, (Math.PI * 2), true);
        context.closePath();
        context.fill();
      }
    };
    </script>
  </body>
</html>
```

这创建-100～+100 之间的一个随机数，并把它加在 canvas 的中心点上。这样所有的小圆点都会位于距离中心点 100 个像素的范围内。运行结果如图 19-3 所示。

效果还不错，但是如果想让它们更拥挤一点，把点的数量增加为 300 个，并把区域缩小为 100 × 100。你会发现有一些奇怪的事情发生，如图 19-4 所示。下面是代码（示例 04-random-2.html）：

图 19-3 随机分布的小圆点

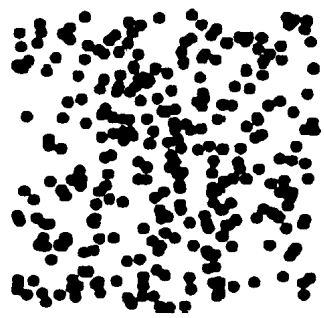


图 19-4 这种方法形成一个正方形，看起来不是很随机

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Random 2</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        numDots = 300;

    while (numDots--) {
      var x = canvas.width / 2 + Math.random() * 100 - 50,
          y = canvas.height / 2 + Math.random() * 100 - 50;

      context.fillStyle = "#000000";
```

```
        context.beginPath();
        context.arc(x, y, 2, 0, (Math.PI * 2), true);
        context.closePath();
        context.fill();
      }
    };
    </script>
  </body>
</html>
```

正如你看到的，这些小圆点形成一个正方形。也许这正是你想要的效果，但是如果你想要制作爆炸或星系的效果，这样的正方形看起来并不真实。所以让我们来继续看下一种技术。

19.2.2 圆形分布

圆形分布尽管比正方形分布稍微复杂一点，但实现图形分布并不算困难。首先你要知道圆的半径，为了与上例匹配，把它设置为50。这是小圆点距离中心的最大距离。用一个介于0～50之间的随机数作为半径，再取一个介于0～2π（360°）之间的随机数，最后使用一些三角运算来计算出小圆点的 x、y 坐标。下面是本例的代码（05-random-3.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Random 3</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        numDots = 300,
        maxRadius = 50;

    while (numDots--) {
      var radius = Math.random() * maxRadius,
          angle = Math.random() * (Math.PI * 2),
          x = canvas.width / 2 + Math.cos(angle) * radius,
          y = canvas.height / 2 + Math.sin(angle) * radius;

      context.fillStyle = "#000000";
      context.beginPath();
      context.arc(x, y, 2, 0, (Math.PI * 2), true);
      context.closePath();
      context.fill();
    }
  };
  </script>
 </body>
</html>
```

运行结果如图 19-5 所示。

这样看起来更像是爆炸的效果。然而，你可能注意，有很多小圆点都拥挤在中心位置附近。这是因为沿着半径的分布是均匀的，这就意味着，中心和边缘的小圆点数目差不多。因为中心区域的面积很小，所以小图点看起来就比较拥挤。

我们可以让小圆点看起来均匀分布在圆形区域内，如下例（06-random-4.html）所示：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Random 4</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        numDots = 300,
        maxRadius = 50;
    while (numDots--) {
      var radius = Math.sqrt(Math.random()) * maxRadius,
          angle = Math.random() * (Math.PI * 2),
          x = canvas.width / 2 + Math.cos(angle) * radius,
          y = canvas.height / 2 + Math.sin(angle) * radius;
      context.fillStyle = "#000000";
      context.beginPath();
      context.arc(x, y, 2, 0, (Math.PI * 2), true);
      context.closePath();
      context.fill();
    }
  };
  </script>
 </body>
</html>
```

通过计算随机数的平方根（偏向 1，远离 0），我们可以让分布更加平滑。你可以看到结果如图 19-6 所示。

图 19-5 圆形随机分布

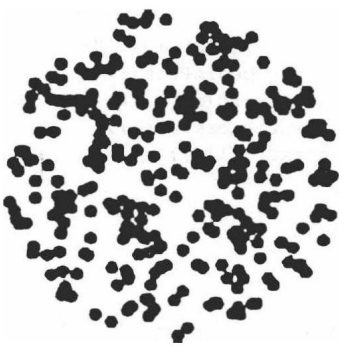


图 19-6 更加平滑的圆形分布

19.2.3 偏向分布

你可能想让小圆点随机分布在整个 canvas 上，但让它们趋向分布在中心区域。就是会有一些在边缘附近，但是越接近中心，分布得越多。这有些像第一个圆形分布的例子，不过这次是在一个矩形区域内。

这可以通过给为每一个位置产生多个随机数并计算它们的平均值来实现。比如，如果 canvas 的宽度是 400 个像素，你为每一个对象创建一个随机的 x 坐标，所有的对象都会均匀分布。但是如果你创建两个介于 0～400 之间的随机数然后计算平均值，对象分布在中心的几率就会更高一些。

让我们深入研究一下。两个随机数有一定的几率同时落在高区间内，比如 300～400。它们有相同的几率落在低区间内，比如 0～100。但是一个在低区间，另一个在高区间的几率更高，或者甚至两个都在中间区间。这些可能性的平均值就使小圆点落在中心区域的可能性更大。

在代码中看具体分布概率，像往常一样，先从单轴上的例子开始。下面是示例 07-random-5.html：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Random 5</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        numDots = 300;

    while (numDots--) {
      var x1 = Math.random() * canvas.width,
          x2 = Math.random() * canvas.width,
          x = (x1 + x2) / 2,
          y = canvas.height / 2 + Math.random() * 50 - 25;

      context.fillStyle = "#000000";
      context.beginPath();
      context.arc(x, y, 2, 0, (Math.PI * 2), true);
      context.closePath();
      context.fill();
    }
  };
  </script>
 </body>
</html>
```

这里产生了两个随机数 x1 和 x2，然后把小圆点的 x 坐标设置为它们的平均值，把 y 坐标简单地设置为中心附近区间内的一个随机数。本示例的运行效果如图 19-7 所示。

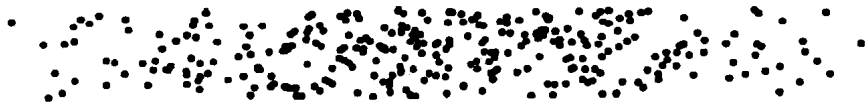


图 19-7 具有一次迭代的偏向分布

这里的效果并不是非常显著，但是你能看到中心区域的小圆点多一些，边缘少一些。如果创建更多的随机数并取平均值，效果会更明显。可以把它放在一个 for 循环中，让它变得更通用，如下例（文件 08-random-6.html）所示：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Random 6</title>
  <link rel="stylesheet" href="style.css">
</head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        numDots = 300,
        iterations = 6;

    while (numDots--) {
      for (var i = 0, xpos = 0; i < iterations; i++) {
        xpos += Math.random() * canvas.width;
      }

      var x = xpos / iterations,
          y = canvas.height / 2 + Math.random() * 50 - 25;

      context.fillStyle = "#000000";
      context.beginPath();
      context.arc(x, y, 2, 0, (Math.PI * 2), true);
      context.closePath();
      context.fill();
    }
  };
  </script>
 </body>
</html>
```

这里加入了一个 iterations 变量来控制有多少个随机数取平均值。变量 xpos 初始化为零，然后把随机数都累加在它上面，最后用累加值除以 iterations 得到最终值。运行结果如图 19-8 所示。

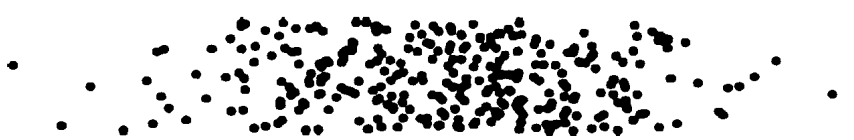


图 19-8 6 次迭代的偏向分布

在 y 轴上做同样的事也很简单，如示例 09-random-7.html 所示：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Random 7</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        numDots = 300,
        iterations = 6;

    while (numDots--) {
      for (var i = 0, xpos = 0; i < iterations; i++) {
         xpos += Math.random() * canvas.width;
      }

      for (var j = 0, ypos = 0; j < iterations; j++) {
         ypos += Math.random() * canvas.height;
      }

      var x = xpos / iterations,
          y = ypos / iterations;

      context.fillStyle = "#000000";
      context.beginPath();
      context.arc(x, y, 2, 0, (Math.PI * 2), true);
      context.closePath();
      context.fill();
    }
  };
  </script>
 </body>
</html>
```

运行本示例能得到类似图 19-9 所示的分布。

这是几个例子中最有随机感、最有爆炸感、最像星系分布的效果了。不过这也是计算最密集的算法。

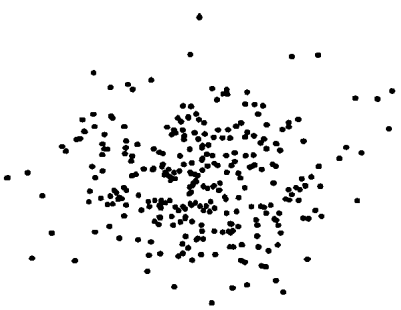


图 19-9 二维偏向分布

19.2.4 基于碰撞的分布

有时候，也许你想让物体随机分布在特定的形状内。最简单的方法可能是在一个屏幕上产生一个随机坐标，然后用碰撞检测的方法来检测这个坐标是否在特定区域内。如果这个点在特定区域外，则丢弃它，然后产生一个新的随机坐标，同时再进行一轮测试，直到找到特定区域

内的一个坐标为止。

在下一个例子中，我们让小圆点随机分布在两个圆形内，我们继续使用 Ball 类。我们使用第 9 章介绍的基于距离的碰撞检测公式来判断小圆点是否在这两个圆内，如果不是则继续尝试，直到我们找到圆内的一个点。本示例的代码如下（10-random-8.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Collision-Based Distribution</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        numDots = 300,
        ball0 = new Ball(),
        ball1 = new Ball(80);

    ball0.x = 100;
    ball0.y = canvas.height / 2;
    ball1.x = 300;
    ball1.y = canvas.height / 2;

    function detectCollision(x, y, ball) {
      var dx = x - ball.x,
          dy = y - ball.y,
          dist = Math.sqrt(dx * dx + dy * dy);
      return (dist < ball.radius);
    }

    while (numDots--) {
      //initialize variables
      var x = 0,
          y = 0;

      //if x, y not in ballA AND not in ballB, set new random position
      while (!detectCollision(x, y, ball0) && !detectCollision(x, y, ball1)) {
        //get random position on canvas
        x = Math.random() * canvas.width;
        y = Math.random() * canvas.height;
      }

      context.fillStyle = "#000000";
      context.beginPath();
      //x, y, radius, start_angle, end_angle, anti-clockwise
      context.arc(x, y, 2, 0, (Math.PI * 2), true);
      context.closePath();
```

```
        context.fill();
      }
    };
    </script>
  </body>
</html>
```

这个例子产生的分布效果如图 19-10 所示。

这里检测一个点是否落在圆中，不过这个概念可以用在任何形状中（需要考虑第 9 章介绍的碰撞检测的局限性）。你首先产生一个随机位置，然后用合适的碰撞检测技术来检测它是否落在预定义的区域内。这个方法的缺点是产生了很多实际上没有用的点，不过它并不用在每一帧中都运行。这在初始化程序时很有用（比如，在一个游戏中，你要随机初始化敌人的位置）。

图 19-10 两个圆内的随机分布，使用基于距离的碰撞检测

19.3 基于定时器和基于时间的动画

到目前为止，本书中的例子都是在 drawFrame 函数中使用 window.requestAnimationFrame 来循环创建动画。第 2 章讨论过，这个方法对基于 canvas 的动画来说是最合适的，因为浏览器可以优化这个函数，以保证其他方面的性能不受影响。详细内容参见第 2 章。

但是在以前那些不支持多媒体的浏览器中，JavaScript 定时器是制作动画的唯一方法。下面我们来看一下，这种方法如何使用时间间隔来控制动画播放速度。然后，我们将讨论基于时间的动画，这种技术既可以使用定时器，也可以使用帧。

19.3.1 基于定时器的动画

基于定时器的动画要用到 window.setTimeout 和 window.setInterval 两个函数。它们都接收两个参数：一个是待执行函数，另一个是等待时间，单位为毫秒。

```
function printMessage () {
  console.log("Hello, Timer!");
}
window.setTimeout(printMessage, 2000);
```

这里把函数 printMessage 传入 window.setTimeout，设置为在 2000 毫秒后执行它。当执行这段代码时，消息将在两秒后（1 秒等于 1000 毫秒）输出在调试控制台上。

window.setInterval 同样也会执行一个函数，不过它不是执行一次，它会每隔一定时间就调用一次（除非你自己告诉它停止）。

创建基于定时器的动画也不比本书中使用 window.requestAnimationFrame 的其他例子困

难。用时间间隔来控制帧率，定时器执行动画循环。下面是一个简单的例子（文件 11-timer.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Timer</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        ball = new Ball(),
        fps = 30;

    ball.y = canvas.height / 2;
    ball.vx = 5;

    function drawFrame () {
      context.clearRect(0, 0, canvas.width, canvas.height);

      ball.x += ball.vx;
      ball.draw(context);
    }
    window.setInterval(drawFrame, 1000/fps);
  };
  </script>
 </body>
</html>
```

在本例中，使用定时器把动画循环设置为每秒运行 30 帧。因为我们并不能精确地知道代码在用户机器上的执行速度，所以基于定时器的动画没有基于帧的动画精确。在这个简单的例子中并不能看出区别，不过在计算量较大的程序中，你就会看到在不同的电脑上运行效果的区别。

如果你真的需要较高的精确度，那就要使用基于时间的动画。

19.3.2 基于时间的动画

如果需要让动画对象的速度保持一致，像某类游戏中的场景，就需要用到基于时间的动画。无论是基于帧还是基于定时器的动画都不能以特定速率播放。复杂的动画在比较慢的电脑上播放，可能会比它的设计速度慢。下面你会看到，使用基于时间的动画，可以控制播放速度，从而与帧率无关。

首先要做的一件事就是改变你思考速度的方式。到目前为止，当你看到 vx=5 时，你使用的单位都是像素/每帧。换句话说，在每一帧中物体在 *x* 轴上移动 5 个像素。在基于定时器的动画中，它的意思是 5 个像素/每个时间间隔。

在基于时间的动画中，我们使用真实的时间来度量，比如秒。因为我们以秒为单位来处理，而不是更小的单位，所以速度要更高一些。如果，物体每帧移动 10 个像素，每秒播放 30 帧，

则它每秒移动 300 个像素。在下例（12-time-based-1.html）中，我们在第 6 章中的例子 05-bouncing-2.html 的基础上做一些小改动，改动部分用粗体标出。

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Time based 1</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <script src="utils.js"></script>
  <script src="ball.js"></script>
  <script>
  window.onload = function () {

    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),

        ball = new Ball(),

        start_time = new Date().getTime(),

        time = getTimer(),

        vx = 300,

        vy = -300,

        bounce = -0.7;

    ball.x = canvas.width * Math.random();

    ball.y = canvas.height / 2;

    function getTimer () {
      return (new Date().getTime() - start_time); //milliseconds
    }

    (function drawFrame () {
       window.requestAnimationFrame(drawFrame, canvas);
       context.clearRect(0, 0, canvas.width, canvas.height);

      var elapsed = getTimer() - time,
          left = 0,

          right = canvas.width,

          top = 0,

          bottom = canvas.height;

      time = getTimer();

      ball.x += vx * elapsed / 1000;

      ball.y += vy * elapsed / 1000;

      if (ball.x + ball.radius > right) {

          ball.x = right - ball.radius;

          vx *= bounce;
```

```
    } else if (ball.x - ball.radius < left) {
       ball.x = left + ball.radius;
       vx *= bounce;
    }
    if (ball.y + ball.radius > bottom) {
        ball.y = bottom - ball.radius;
        vy *= bounce;
    } else if (ball.y - ball.radius < top) {
       ball.y = top + ball.radius;
       vy *= bounce;
    }

    ball.draw(context);
  }());
};
</script>
</body>
</html>
```

正如前面所提到的，速度陡然提升，现在用硬编码的值，而不是随机值。变量 time 是 getTimer 函数的返回值。getTimer 只是一个计数器，表示动画已经播放了多久，单位是毫秒。

如果调用两次 getTimer，把两次的结果相减，你就能得到毫秒级别精度的时间差。

所以我们的策略是：在每一帧开始时调用 getTimer，然后计算相距上一帧过去了多少毫秒。再除以 1000，转换成以秒为单位。因为现在的 vx 和 vy 都是以像素/每秒为单位，所以用它们乘以这个时间差，就能知道要把物体移动多少。同时别忘了重置 time 变量，因为在下一帧中还需要用到它。

测试本例，你可以看到小球移动的速度和以前差不多。但是令人吃惊的是，无论帧率如何变化，它都能保持同样的速度！当然，帧率高一些动画会更平滑，帧率过低动画会跳跃，但是速度本身保持不变。

这个方法也可以用在基于定时器的动画中，这里就不进行详述了，但是可以看一下文件 13-time-based-2.html。正常的动画循环已经被 window.setInterval 代替，使用 getTimer 来计算速度。你可以调整 fps 变量，观察不同的帧率对动画的影响。

你可以在本书中任何与速度有关的例子中应用这个技术。这样一来，你还需要把这个技术应用在加速度或持续的外力上，比如重力，因为这些都基于时间。当使用这种类型的动画时，加速度的值要大一些。因为加速度的定义是：每个时间间隔的距离/每个时间间隔。比如，重力近似于 32 英尺/每秒。

如果 gravity 在基于帧的动画中为 0.1，这里就要把它增加为 300。然后像这样计算：

```
vy += gravity * elapsed / 1000;
```

在上一个例子中加入重力试试看，值为 300。你会看到它与在帧动画中 gravity 为 0.1 时的效果差不多。可以在练习 14-time-based-3.html 中找到相应的代码。

19.4 等质量物体之间的碰撞

还记得第 11 章的动量守恒吗？那些代码相当严谨。不过，当两个相同质量的物体碰撞时，

我们有更简单的做法：沿着碰撞的方向，物体简单地交换它们的速度。尽管仍然使用坐标旋转来确定碰撞角度与物体在这个角度上的速度，不过省去了复杂的动量守恒。为了看一下它如何工作，让我们回到第 11 章中的示例 06-multi-billiard-2.html，它将作为本示例（15-same-mass.html）的基础。这是个很大的文件，所以这里并没有列出所有代码，仅看看脚本顶端附近创建小球的 for 循环：

```
for (var radius, ball, i = 0; i < numBalls; i++) {
   radius = Math.random() * 20 + 15;
   ball = new Ball(radius, Math.random() * 0xffffff);
   ball.mass = radius;
   ball.x = Math.random() * canvas.width;
   ball.y = Math.random() * canvas.height;
   ball.vx = Math.random() * 10 - 5;
   ball.vy = Math.random() * 10 - 5;
   balls.push(ball);
}
```

在新例子中，我们将移除那行粗体代码，在脚本顶部创建一个新变量：

```
var radius = 20;
```

这样使所有小球的大小和质量都相同。

接下来，找到 checkCollision 函数中下面这段代码：

```
  //rotate ball0's velocity
  vel0 = rotate(ball0.vx, ball0.vy, sin, cos, true),

  //rotate ball1's velocity
  vel1 = rotate(ball1.vx, ball1.vy, sin, cos, true),

  //collision reaction
  vxTotal = vel0.x - vel1.x;

vel0.x = ((ball0.mass - ball1.mass) * vel0.x + 2 * ball1.mass * vel1.x) /
         (ball0.mass + ball1.mass);
vel1.x = vxTotal + vel0.x;
```

这段代码用来计算碰撞方向上的速度，然后根据它们的质量来计算碰撞结果。标记为“collision reaction”的那段是动量守恒的代码，不过现在不需要那段代码了，你可用简单的交换 vel0 和 vel1 来替换那段代码。整段代码现在应该是这样的：

```
  //rotate ball0's velocity
  vel0 = rotate(ball0.vx, ball0.vy, sin, cos, true),

  //rotate ball1's velocity
  vel1 = rotate(ball1.vx, ball1.vy, sin, cos, true);

//collision reaction, swap the two velocities
var temp = vel0;
vel0 = vel1;
vel1 = temp;
```

这里消除了不少数学概念，你可以测试改动之前和之后的代码，结果是一样的。

19.5 集成声音

本书中一直没有出现声音的运用。尽管声音并不直接是动画的一部分，但是如果把音效运

用好，对动画的真实感和沉浸感都大有帮助。

HTML5 中最受期待的一项功能是支持 audio 元素，就是可以在浏览器上播放本地声音。好消息是，大部分支持 HTML5 的浏览器都对声音有很好的支持。坏消息是，至少在撰写本文时，各种浏览器实现的程度各不相同。因为音频格式许可的问题，我们通常不清楚哪个浏览器支持哪种格式。尽管这是一个令人沮丧的开发环境，不过如果你提前做足功课的话，也不是不能克服它。很重要的一点是提供多种声音格式，这样浏览器就可以选择它支持的格式来播放。

在本示例中，我们再次回到第 6 章中小球从墙面反弹的例子 05-bouncing-2.html。每当小球击中墙面时，它就发出声音。

首先，需要一个声音。可以从网上下载一个，或者使用自己的，也可以使用其他例子中的音频剪辑。把它们与你的文件放在同一个目录下。

接下来，需要把下面的 audio 标签放在 HTML 文件的 body 中。

```
<audio id="sound">
  <source src="boing.ogg"/>
  <source src="boing.mp3"/>
  <p>This browser does not support the <code>audio</code> element.</p>
</audio>
```

这里创建了一个 audio 元素，提供了两种音频格式以供选择：Ogg Vorbis 和 MP3。这两种格式应该可以覆盖所有主流浏览器了。当浏览器不支持 HTML5 的 audio 元素时，我们还提供了一条反馈消息。如果浏览器支持，则忽略这条消息。

可以在 JavaScript 中用 DOM 接口来访问 audio 元素，并把它赋给一个变量：

```
var sound = document.getElementById('sound');
```

声音已经准备好了，剩下要做的就是调用 play 方法：

```
sound.play();
```

在下一个例子中，在开始动画之前用 JavaScript 来检查浏览器是否支持 audio 元素。如果不支持，就显示一个空白的 canvas 并在浏览器的调试命令行中抛出一条错误信息。可以通过检查 DOM 接口返回的 audio 对象是否包含 canPlayType 这个方法来判断浏览器是否支持 audio 元素：

```
if (typeof sound !== 'object' || !sound.canPlayType) {
    throw new Error("The audio element is not supported in this browser.");
}
```

下面是本示例的代码（16-sound-events.html）：

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title>Sound Events</title>
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <canvas id="canvas" width="400" height="400"></canvas>
  <audio id="sound">
    <source src="boing.ogg" />
    <source src="boing.mp3" />
    <p>This browser does not support the <code>audio</code> element.</p>
```

```
</audio>
<script src="utils.js"></script>
<script src="ball.js"></script>
<script>
window.onload = function () {
  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d'),
      sound = document.getElementById('sound'),
      ball = new Ball(),
      vx = Math.random() * 10 - 5,
      vy = Math.random() * 10 - 5,
      bounce = -0.7;

  //the animation only runs with audio
  if (typeof sound !== 'object' || !sound.canPlayType) {
    throw new Error("This browser does not support the audio element.");
  }

  (function drawFrame () {
     window.requestAnimationFrame(drawFrame, canvas);
     context.clearRect(0, 0, canvas.width, canvas.height);

     var left = 0,
         right = canvas.width,
         top = 0,
         bottom = canvas.height;

     ball.x += vx;
     ball.y += vy;

     if (ball.x + ball.radius > right) {
         sound.play();
         ball.x = right - ball.radius;
         vx *= bounce;
     } else if (ball.x - ball.radius < left) {
        sound.play();
        ball.x = left + ball.radius;
        vx *= bounce;
     }
     if (ball.y + ball.radius > bottom) {
         sound.play();
         ball.y = bottom - ball.radius;
         vy *= bounce;
     } else if (ball.y - ball.radius < top) {
        sound.play();
        ball.y = top + ball.radius;
        vy *= bounce;
     }

     ball.draw(context);
  }());
};
</script>
</body>
</html>
```

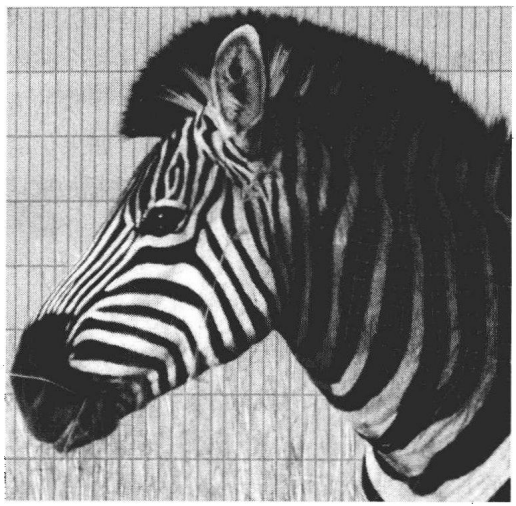
测试这个动画看看，不，是听听加入声音带来的不同感受。当然，要根据不同的场景使用正确的声音，不能过于夸张，这是个艺术活。

由于不同浏览器实现 audio 元素的方法不一致，因此出现了一些 JavaScript 库来帮助开发人员，让他们的工作变得简单。如果你遇到了跨浏览器的问题，别担心，可以找一些解决这个问题的项目。有些库甚至会在无法播放时降级到用 Flash 播放，不过至少你能知道用户会听到声音。

19.6 小结

祝贺你读完了本书！不简单，这里学到的技术也可以应用在其他各种图形编程环境中。当然，最令人兴奋的就是基于 Web 的动画和游戏的潜力。

本书提供了大量的示例以供参考、练习和破坏。学习和进步的最好方法就是把程序分解成碎片，直到弄清楚每一小块如何影响其他部分。当然，阅读原理很重要，但是你必须编写、感受以及应用这些思想。此外，总是有更多的东西要学，所以当你遇到不熟悉的数学知识或者新的编程风格时，不要放弃，要理解它，调试它，应用它，这也是最有趣的地方。从中你可以建立自己失败与成功的经验，不断创造更好的事物。



附录 A　常用公式

在本书中，关于各种动画效果的公式贯穿始终。在每一章的结尾，提炼出了最有用和最常用的公式、方程和代码片段。这里将它们统一收集起来，以供参考。

A.1　第 3 章

A.1.1　三角学基础函数

```
sine of angle = opposite / hypotenuse
cosine of angle = adjacent / hypotenuse
tangent of angle = opposite / adjacent
```

A.1.2　角度与弧度互转

```
radians = degrees * Math.PI / 180
degrees = radians * 180 / Math.PI
```

A.1.3　朝鼠标指针（或任意一点）旋转

```
//substitute mouse.x, mouse.y with the x, y point to rotate to
dx = mouse.x - object.x;
dy = mouse.y - object.y;
object.rotation = Math.atan2(dy, dx) * 180 / Math.PI;//radiams to degrees
```

A.1.4　创建波

```
(function drawFrame () {
```

```
    window.requestAnimationFrame(drawFrame, canvas);

    //assign value to x, y or other property of object
    value = center + Math.sin(angle) * range;
    angle += speed;
}());
```

A.1.5 创建圆形

```
(function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);

//assign position to x and y of object, or drawing coordinate
    xposition = centerX + Math.cos(angle) * radius;
    yposition = centerY + Math.sin(angle) * radius;
    angle += speed;
}());
```

A.1.6 创建椭圆形

```
(function drawFrame () {
    window.requestAnimationFrame(drawFrame, canvas);

//assign position to x and y of object or drawing coordinate
    xposition = centerX + Math.cos(angle) * radiusX;
    yposition = centerY + Math.sin(angle) * radiusY;
    angle += speed;
}());
```

A.1.7 获取两点间的距离

```
//points are x1, y1 and x2, y2
//can be object positions, mouse coordinates, etc.
dx = x2 - x1;
dy = y2 - y1;
dist = Math.sqrt(dx * dx + dy * dy);
```

A.2 第4章

A.2.1 从十六进制转换到十进制

```
console.log(hexValue);
```

A.2.2 从十进制转换到十六进制

```
console.log(decimalValue.toString(16));
```

A.2.3 组合三原色

```
color = red << 16 | green << 8 | blue;
```

A.2.4 提取三原色

```
red = color24 >> 16 & 0xFF;
green = color24 >> 8 & 0xFF;
blue = color24 & 0xFF;
```

A.2.5 绘制一条穿越某个点的曲线

```
// xt, yt is the point you want to draw through
// x0, y0 and x2, y2 are the end points of the curve
x1 = xt * 2 - (x0 + x2) / 2;
y1 = yt * 2 - (y0 + y2) / 2;
context.moveTo(x0, y0);
context.quadraticCurveTo(x1, y1, x2, y2);
```

A.3 第5章

A.3.1 将角速度分解为 *x*、*y*轴上的速度向量

```
vx = speed * Math.cos(angle);
vy = speed * Math.sin(angle);
```

A.3.2 将角加速度（作用于物体上的力）分解为 *x*、*y*轴上的加速度

```
ax = force * Math.cos(angle);
ay = force * Math.sin(angle);
```

A.3.3 将加速度加入速度向量

```
vx += ax;
vy += ay;
```

A.3.4 将速度向量加入位置坐标

```
object.x += vx;
object.y += vy;
```

A.4 第6章

A.4.1 移除越界物体

```
if (object.x - object.width / 2 > right ||
   object.x + object.width / 2 < left ||
```

```
    object.y - object.height / 2 > bottom ||
    object.y + object.height / 2 < top) {
    //code to remove object
}
```

A.4.2 重置越界物体

```
if (object.x - object.width / 2 > right ||
    object.x + object.width / 2 < left ||
    object.y - object.height / 2 > bottom ||
    object.y + object.height / 2 < top) {
    //reset object position and velocity
}
```

A.4.3 屏幕环绕越界物体

```
if (object.x - object.width / 2 > right) {
    object.x = left - object.width / 2;
} else if (object.x + object.width / 2 < left) {
    object.x = right + object.width / 2;
}
if (object.y - object.height / 2 > bottom) {
    object.y = top - object.height / 2;
} else if (object.y + object.height / 2 < top) {
    object.y = bottom + object.height / 2;
}
```

A.4.4 应用摩擦力（正确方法）

```
speed = Math.sqrt(vx * vx + vy * vy);
angle = Math.atan2(vy, vx);
if (speed > friction) {
    speed -= friction;
} else {
    speed = 0;
}
vx = Math.cos(angle) * speed;
vy = Math.sin(angle) * speed;
```

A.4.5 应用摩擦力（简便方法）

```
vx *= friction;
vy *= friction;
```

A.5 第8章

A.5.1 简单缓动，详细版

```
var dx = targetX - object.x,
    dy = targetY - object.y;
```

```
vx = dx * easing;
vy = dy * easing;
object.x += vx;
object.y += vy;
```

A.5.2 简单缓动，缩略版

```
vx = (targetX - object.x) * easing;
vy = (targetY - object.y) * easing;
object.x += vx;
object.y += vy;
```

A.5.3 简单缓动，简易版

```
object.x += (targetX - object.x) * easing;
object.y += (targetY - object.y) * easing;
```

A.5.4 简单弹动，详细版

```
var ax = (targetX - object.x) * spring,
    ay = (targetY - object.y) * spring;
vx += ax;
vy += ay;
vx *= friction;
vy *= friction;
object.x += vx;
object.y += vy;
```

A.5.5 简单弹动，缩略版

```
vx += (targetX - object.x) * spring;
vy += (targetY - object.y) * spring;
vx *= friction;
vy *= friction;
object.x += vx;
object.y += vy;
```

A.5.6 简单弹动，简易版

```
vx += (targetX - object.x) * spring;
vy += (targetY - object.y) * spring;
object.x += (vx *= friction);
object.y += (vy *= friction);
```

A.5.7 有偏移量的弹动

```
var dx = object.x - fixedX,
    dy = object.y - fixedY,
```

```
    angle = Math.atan2(dy, dx),
    targetX = fixedX + Math.cos(angle) * springLength,
    targetY = fixedX + Math.sin(angle) * springLength;

//spring to targetX, targetY as above
```

A.6 第 9 章

A.6.1 基于距离的碰撞检测

```
//starting with objectA and objectB
//if using an object without a radius property,
//you can use width or height divided by 2
var dx = objectB.x - objectA.x,
    dy = objectB.y - objectA.y,
    dist = Math.sqrt(dx * dx + dy * dy);

if (dist < objectA.radius + objectB.radius) {
    //handle collision
}
```

A.6.2 多物体碰撞检测

```
objects.forEach(function (objectA, i) {
  for (var j = i + 1; j < objects.length; j++) {
     //evaluate reference using j. For example:
     var objectB = objects[j];
     //perform collision detection between objectA and objectB
  }
});
```

A.7 第 10 章

A.7.1 坐标旋转

```
x1 = x * Math.cos(rotation) - y * Math.sin(rotation);
y1 = y * Math.cos(rotation) + x * Math.sin(rotation);
```

A.7.2 反向坐标旋转

```
x1 = x * Math.cos(rotation) + y * Math.sin(rotation);
y1 = y * Math.cos(rotation) - x * Math.sin(rotation);
```

A.8 第 11 章

A.8.1 动量守恒的数学表示

$$v0Final = \frac{(m0 - m1) \times v0 + 2 \times m1 \times v1}{m0 + m1}$$

$$v1Final = \frac{(m1 - m0) \times v1 + 2 \times m0 \times v0}{m0 + m1}$$

A.8.2 动量守恒的 JavaScript 代码

```
var vxTotal = vx0 - vx1;
vx0 = ((ball0.mass - ball1.mass) * vx0 + 2 * ball1.mass * vx1) / (ball0.mass + ball1.mass);
vx1 = vxTotal + vx0;
```

A.9 第12章

A.9.1 基本引力

$$force = G \times m_1 \times m_2 / distance^2$$

A.9.2 引力公式的 JavaScript 实现

```
function gravitate (partA, partB) {
  var dx = partB.x - partA.x,
      dy = partB.y - partA.y,
      distSQ = dx * dx + dy * dy,
      dist = Math.sqrt(distSQ),
      force = partA.mass * partB.mass / distSQ,
      ax = force * dx / dist,
      ay = force * dy / dist;

  partA.vx += ax / partA.mass;
  partA.vy += ay / partA.mass;
  partB.vx -= ax / partB.mass;
  partB.vy -= ay / partB.mass;
}
```

A.10 第14章

A.10.1 余弦定理

$$a^2 = b^2 + c^2 - 2 \times b \times c \times \cos A$$
$$b^2 = a^2 + c^2 - 2 \times a \times c \times \cos B$$
$$c^2 = a^2 + b^2 - 2 \times a \times b \times \cos C$$

A.10.2 JavaScript 中的余弦定理

```
var A = Math.acos((b * b + c * c - a * a) / (2 * b * c)),
    B = Math.acos((a * a + c * c - b * b) / (2 * a * c)),
    C = Math.acos((a * a + b * b - c * c) / (2 * a * b));
```

A.11 第15章

A.11.1 基本透视图

```
scale = fl / (fl + zpos);
object.scaleX = object.scaleY = scale;
object.alpha = scale; // optional
object.x = vanishingPointX + xpos * scale;
object.y = vanishingPointY + ypos * scale;
```

A.11.2 Z排序

```
//assumes an array of 3D objects with a zpos property
function zSort (a, b) {
  return (b.zpos - a.zpos);
}
objects.sort(zSort);
```

A.11.3 坐标旋转

```
x1 = xpos * cos(angleZ) - ypos * sin(angleZ);
y1 = ypos * cos(angleZ) + xpos * sin(angleZ);

x1 = xpos * cos(angleY) - zpos * sin(angleY);
z1 = zpos * cos(angleY) + xpos * sin(angleY);

y1 = ypos * cos(angleX) - zpos * sin(angleX);
z1 = zpos * cos(angleX) + ypos * sin(angleX);
```

A.11.4 三维距离计算

```
dist = Math.sqrt(dx * dx + dy * dy + dz * dz);
```