TURING 图灵程序设计丛书

Apress®

Flash ActionScript 3.0 动画教程

[美] Keith Peters 著
王汝义 等 译



经典著作
Foundation ActionScript 3 Animation: Making Things Move! 中文版

跟随Flash大师深入ActionScript动画秘境

丰富的经典示例和专家技巧

融技术与艺术于一炉

人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目（CIP）数据

Flash ActionScript 3.0 动画教程 /（美）皮特斯（Peters, K.）著；王汝义等译. —北京：人民邮电出版社，2008.4（2010.3 重印）
（图灵程序设计丛书）
ISBN 978-7-115-17450-5

Ⅰ. F… Ⅱ. ①皮… ②王… Ⅲ. 动画-设计-图形软件，Flash ActionScript 3.0－教材 Ⅳ. TP391.41

中国版本图书馆 CIP 数据核字（2008）第 000407 号

内 容 提 要

ActionScript 3.0 是一种功能强大的面向对象编程语言。本书分五大部分全面讲解 ActionScript 3.0 动画。第一部分介绍了 ActionScript 动画基础，包括 ActionScript 3.0 基本知识和渲染技术等。第二部分和第三部分集中讲述了制作动画涉及的基本运动和高级运动，如速度和加速度、边界和摩擦力、用户交互、缓动和弹性、碰撞检测以及正向和反向运动等。第四部分介绍了 3D 动画基础。最后一部分介绍了矩阵数学等其他技术。通过阅读本书，读者可以制作出各种酷炫的动画效果。

不论是初学者还是有经验的 ActionScript 开发人员，本书都是一本优秀的参考书。

图灵程序设计丛书

Flash ActionScript 3.0 动画教程

◆ 著 [美] Keith Peters
译 王汝义 等
责任编辑 傅志红 刘艳娟

◆ 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 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京顺义振华印刷厂印刷

◆ 开本：800×1000 1/16
印张：26.75
字数：632 千字 2008 年 4 月第 1 版
印数：7 001－8 000 册 2010 年 3 月北京第 4 次印刷
著作权合同登记号 图字：01-2007-4257 号

ISBN 978-7-115-17450-5/TP

定价：59.00 元

读者服务热线：(010)88593802 印装质量热线：(010)67129223
反盗版热线：(010) 67171154

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

版 权 声 明

Original English language edition, entitled *Foundation ActionScript 3.0 Animation: Making Things Move!* by Keith Peters, published by Apress L.P., 2560 Ninth Street, Suite 219, Berkeley, CA 94710 USA.

Copyright © 2007 by Keith Peters. Simplified Chinese-language edition copyright © 2008 Posts & Telecom Press. All rights reserved.

本书中文简体字版由Apress L.P.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

www.TopSage.com

大家网
TopSage.com

对本书的赞誉

任何一种艺术形式的最佳学习方法，是通过学习和分享大师无价的经验。在这本书中，Keith Peters大师将带领你完成一次艺术编程之旅。Keith 就在你的身旁，他还将引导你完成一次探索之旅。

——Aral Balkan，RIA与Flash技术专家

本书显然是作者多年经验和思考的结果。每个技巧看似简单，其实都有深意存焉。很多情况下，当你在实际开发中遇到困难，回头再看书中内容时，才发现作者早已说明了将要面对的陷阱和解决方案。

——amazon.com

厌倦了使用关键帧和补间实现动画，想学习ActionScript？本书正适合你。虽然已经使用ActionScript多年，我仍然从本书中学到了大量东西。强烈推荐！

——flashmagazine.com

这是我看过的最好的ActionScript动画图书。

——amazon.com

这本书除了能够直接学到大量技术之外，还提供了极有价值的创意和启发性。书中的代码已经运用到我自己的项目中，我还从书中的一些概念获得灵感，开发出漂亮的游戏和用户界面元素……总而言之，如果你要使用ActionScript实现动画效果，请阅读本书。

——Winnipeg Adobe用户组织

如果今年你只打算买一种ActionScript书，请买本书，而且要买上两本，一本在家里读，一本放在办公室里。只需要一个周末，你将学到其他Flash开发人员要花上几年才能精通的知识和技巧。市面上Flash方面真正的好书其实很少，本书就是其中一本。这是多年来我读到的最好的Flash书，绝对是任何Flash开发人员的必读之作。

——Satori Canton，actionscript.com

www.TopSage.com

译 者 序

2007年Webstudio出版了《Flash网站建设技术精粹》简体版和繁体版以及《别具光芒——Flash 8 ActionScript滤镜效果》，除此之外，2007年对于我来说最高兴、最有收获的一件事情，就是Webstudio很荣幸地与人民邮电出版社图灵公司签约，翻译Keith Peters大师的新作*Foundation ActionScript 3.0 Animation: Making Things Move!*。之所以说荣幸，是因为从几年前开始接触Flash时，就对Keith Peters曾有耳闻，随着后来对Flash特别是ActionScript的深入学习，了解到Keith Peters是Flash界很有影响力的大师级人物，在其网站（www.bit-101.com）上提供了成百上千的实验作品。我从他许多关于ActionScript的文章中受益匪浅，了解到一些关于物理、数学以及3D等知识，以及它们如何在Flash中得以应用。在几年的Flash学习过程中，我一直将他作为学习的榜样，他也是我学习Flash的偶像之一。我想会有更多人将Keith Peters作为偶像的，在此我也向Keith Peters表达衷心的谢意。

本书是*Foundation ActionScript Animation: Making Things Move!*的第2版。第1版在编写时主要使用ActionScript 2.0语法，面向的是使用Flash MX 2004和Flash 8的用户。而在第2版中，Keith Peters完全抛弃了ActionScript 2.0语法，使用ActionScript 3.0语法来编写，面向使用Flash CS 3或Flex的用户。全书共有五大部分，19章，第一部分和第二部分为基础内容，第三部分和第四部分为高级动画内容，第五部分为作者提供的补充内容，包括矩阵数学和一些技巧。从动画效果上看，包括随机运动、速度、缓动、弹性、碰撞测试、正向运动、反向运动、粒子、坐标旋转、多物体碰撞、结点域，等等。最让人心动的是3D的应用，包括常见的3D透视、3D实体等。除了这些内容，最有价值的应该是作者将所有这些应用的大多数内容总结成易用的数学或物理公式，在你明白了基本原理后，只需要记住相应的数学或物理公式，就可以灵活地运用了。

本书讲述了各层次Flash用户都关心的问题。Keith Peters自己对本书的评价是——“这是一本非常棒的书”。的确，它非常棒，我可以大胆地说，本书不需要你是一名开发人员，也不需要对ActionScript 3.0有多么深入的研究，你只需要了解ActionScript 3.0基本的类结构、简单的继承关系、事件类型和事件侦听，以及一些编写外部类文件的基本常识，基本上就可以读懂本书了。需要说明的是，如果你有坚实的物理、数学知识，那么在学习过程中会更轻松。另外，本书第2章中详实地讲解了所需要的ActionScript 3.0的基础知识、简单的OOP概念以及继承的相关知识。当然，如果你有一点OOP的知识，那就再好不过了。本书适合于ActionScript 3.0初中级以上的用户，适合于Flash互动设计师、游戏开发人员、RIA应用程序开发人员、多媒体教学人员、各大艺术院校的大学生。它可以成为你的一本Flash案头书。

此书的翻译工作历时4个多月。在翻译过程中，我听取了一些网友的建议，尽量以中英文对照的形式表述相关术语，以方便读者阅读。当然因学识有限、时间仓促，译文中难免会存在疏漏、

www.TopSage.com

错译的现象，请读者多多指正。本书在翻译过程中得到了许多朋友和网友的大力支持和帮助，在此特别感谢黄存海、刘爽、唐贵鸿、马文静、唐俊茹、郑新泉、郭元庆、李得增、汤立红、闻晓丹、吴守江、秦诗富、袁健、赵军义、张文彬、李德英、李清芳、刘静文、李晓栋、李一博、张婷婷、吴丽娜、陈秀萍等朋友的热心帮助。

我们将在Webstudio网站的论坛开设专门的版块为本书提供友情支持，供读者在遇到问题时互相讨论。

Webstudio网站于2004年底创办。2007年底以此为基础创立了北京万博思图信息技术有限公司（www.cn-wbst.cn）。网站主要从事Flash网站建设、Flash应用程序开发、多媒体应用、Flash游戏、Flash lite手机应用程序开发和Flash图书出版等工作。欢迎读者朋友与我们交流，互相学习。

Webstudio网址：http://www.webstudio.com.cn
友情论坛网址：http://www.webstudio.com.cn/forum
Webstudio blog：http://blog.webstudio.com.cn
E-mail：ry_wang@163.com

王汝义
2007年11月15日

www.TopSage.com

序

Flash领域中有一道很深的鸿沟分隔了两种人：一种是在工作中非常了解和经常使用ActionScript的人；另一种是在工作中基本不用或者不懂ActionScript的人。我们有时会错误地将这两种人称为“设计师”（designer）和“开发员”（developer），并且假设“设计师”不懂代码，“开发员”不懂设计。但这充其量只是为了方便而过于简单的划分。

在Web早期，我们还在努力地搞清楚这种新的、令人兴奋的媒介时，使用“设计师”和“开发员”的称号可能还没什么问题。十多年后，我们发现这种媒介并不是全新的东西，它不是与世隔绝的，不能与它之前出现的东西和与它并存的东西相脱离。

今天，“设计师”和“开发员”的称号已经不足以描述他是什么人或者他是做什么的了，但是人们还是频繁使用这两个称号来描述那两种人。用于描述工作职责，这些称号并不是很到位。你做哪类设计工作？你在团队中的职责是什么？你是做图形设计，还是动画设计，还是信息设计（架构），还是对象设计（建模）？如果图形设计师、信息架构师、动画制作者和平面设计人员都包含在“设计师”这个大类中，那么“设计师”这个称号也就不能说明什么了。

界定“开发员”的角色要好一点（至少在Flash环境中）。可以认为“开发员”的工作包括构建用户界面，而且更可能是程序设计。

如果这些称号不足以描述工作职责，那么有一点是可以肯定的：用来描述一个人时，它们起到的是限制作用。例如，“我是一名设计师”往往用来证明时间不是花在学习程序设计上。类似地，“我不能绘制一条笔直的线”也是“开发员”经常用来说明没有时间学习图形基础、动画或用户界面设计的合理借口。（听上去有些不合常理，很多最优秀的图形和动画设计师也无法用画架和画笔画出一条非常直的线来；同样，有些才华横溢的艺术家怎么也设计不出一个像样的用户界面布局。）

不，就每个人而言，都应该接受“艺术家”称谓，而且要配得上这个称谓。获得这样的称谓，不是因为所取得的成就，而是因为一种态度——不懈地追求完美。

无论喜欢与否，我们都是这个时代的艺术家。这种集合了疯狂而美妙的数字媒介就是我们的画架、调色板和画笔。我们可以用它来创造与众不同的、美好的、令人深思的东西。相比过去，我们可以用更多的方式去接触更多的人。

我们拥有的艺术表现形式是非常多样的。比如图形设计——强调形式、对比、重复、色彩和印刷等静态表现；动画设计——以构筑、编辑、合成为主，结合动画设计者的想象力、表现力和来自编剧、导演、和演员的艺术表现能力；用户界面设计和信息架构；以及最新出现的、还未被大家认识到的艺术形式：编程。

但是，编程是一种艺术形式吗？目前只有摄影或电影以及它们之前的绘画、木偶和戏剧才是艺术形式。

www.TopSage.com

当摄影刚出现时，它被宣布为一项了不起的技术成就，但也只是如此。它被视为捕捉现实的技术行为。150年之后，我们可以把摄影语言理解为主观思想的表达方式和艺术表现的手段。

仅仅一个世纪前，我们看到的影片多为纪实影片，缺乏艺术性。今天，我们可以欣赏它们的思想和许多错综复杂的、可以将大师级作品与家庭录像区分开来的情节。但并非总是如此。当路易斯和奥古斯特·卢米首次在他们的电影中呈现特快列车到达里昂火车站的场面时，观众们一看到迎面而来的火车就惊恐地从剧院逃了出去，因为他们害怕会真地丧命。今天，我们知道得更多了。我们知道如何“读懂”电影，对电影有了更深一步的认识，但是对于程序设计，我们却不能这样说。

一条神奇的界线将艺术家和工匠区分开来：这条线在创造性的火花闪耀间孕育出来，但是经过了无限的激情、动力和奉献的辛勤磨砺。艺术家浑身洋溢着令人羡慕的能量，令人心生敬畏、感同身受，并产生情感共鸣。尽管这个称号曾专门用来称呼技艺高超画家、雕塑家、建筑师，但我们现在已经能够更广泛地理解什么构成了艺术，进而理解什么造就了艺术家。

今天，我们把程序设计视为一种纯粹的技术行业。我们总是在谈论创作人员与技术人员之间的区别，并把程序员归入后一类。然而，今天的程序员就像上个世纪的电影制片人：他长期默默无闻地工作着，期盼着整个社会都有相当的代码修养，懂得欣赏其代码的精妙之处。这要等100年吗？我认为不用。

非常感谢Keith Peters这样的程序员兼艺术家，我们看到了真实的程序与视觉艺术之间的一座桥梁，引导更多的人去了解程序背后的艺术。Keith的代码是可视的、有活力的、有规律的、流畅的、不断发展的，且总体上以不可预见的、独特的和奇妙的方式不断变化着。变量、语句、循环、方法和类组合地创建了有感染力的、动人的体验，使我们超越了日常生活体验，由此可以发现一个完全由声和光组成的新世界。

本书第1版讲解的是ActionScript 2.0。之后，随着新的脚本语言ActionScript 3.0和针对Flash Player 9的全新的虚拟机的引入，Flash平台发生了革命性的变化。这种改变开创了一个新的世界，我非常欣喜地看到Keith已经使用这令人兴奋的技术更新了*Foundation ActionScript Animation: Making Things Move!*。但是，我们先抛开令人兴奋的技术和版本号，不要忘记最重要的事情：这本书是关于艺术的。

编程是一种新的、令人兴奋的艺术形式，它的受众面、价值及人们对它的认识在日益提高。而任何一种艺术形式的最佳学习方法是经历一个学徒期——分享一位大师的宝贵经验。在本书中，你将发现有这样的一位大师带领你进入自己的艺术性编程旅行。本书的另一大好处是Keith就在你身旁，引导你进行探索之旅。

一些正在成长的艺术家都在准备着加入到Flash能手的行列中。本书中的知识可以为你提供发掘创造潜力的基础。其他的就如我们熟知的：10%的天赋，90%的努力。梦想家计划，艺术家的创造。马上行动起来吧！

画架和画笔已经就绪，你的旅途拉开了帷幕……

Aral Balkan
Flash与RIA技术国际知名专家
2007年1月8日
旧金山

www.TopSage.com

大家网
TopSage.com

引　言

这是一本非常棒的书!

在这里，我要再重复一遍！在本书第1版的引言中，我曾阐明，一本书的优秀并不反映我个人的优秀，但这本书确实覆盖了许多非常好的知识点。它回答了我这些年遇到的所有关于ActionScript动画的问题，许多人还在不断遭遇这些问题，许多人仍在发电子邮件问我这些问题。尽管那些答案并不是我自己想出的，但都是我通过搜索、查阅资料、请教能力更强的人而得来的。因此，我将那些我认为正确的答案按一定的逻辑顺序组织起来，并且用比较容易理解的方式解释给大家。不管怎么样，我相当自负的开篇声明并非虚言，因为书的销售数据和大家给我的个人反馈可以作证。所以，感谢大家的购买，感谢你们的反馈意见。

本书与上一版基本是同一本书，只是脚本转换为了ActionScript 3.0。本书同样包括第1版中的基础公式和技术，顺序上也是基本相同的。我希望能加入些新的内容：其他技术，但并没有删除之前的内容。实际上，尽管还是那些相同的基础内容，但在ActionScript 3.0中要占去更多的篇幅，因为几乎每个示例都给出了完整的代码。尽管包含的仍是一些基础内容，但还是有大量工作要做，这可以说是ActionScript带给我们的巨大变化。

好，如果你打算读这本书，就要学习一点面向对象编程知识。在第1版中，我是尽量地避开面向对象编程，但在这本书中我决定放弃时间线，并深入到ActionScript 3.0中，这意味着要编写类。如果你之前没有接触过类或是面向对象编程，可能会有一点点害怕，但我可以向你保证当完成第4章的时候，你将会掌握使用类编写程序，而且当你读到书的一半时，类的编写对你来说已经不成问题了。如果你已经很熟悉ActionScript 2.0的类，本书将帮助你过渡到ActionScript 3.0。

除了这些之外，我将要简要地重复我在第1版的引言中说过的几件事情。

第一，书中给出的示例和技术并不一定是唯一或最好的方法。我会说它们都做了应该做的，其中的大部分实现得非常好。但是如果你认为可以改进它们，那么请加以改进。[①]

第二，书中并没有包含完整的游戏教程。它们更像是各种独立技术的荟萃。我相信如果你能够理解和应用这些技术，就可以使用它们来制作许多很酷的东西。

最后，还有一点，我会在书中多次说到（甚至由于说得太频繁而被批评），就是尽管相应的章中包含了大量的数学公式和物理知识，但形式都比较自由。而经过特别设计都以使效果所得到的公式在Flash播放器中看上去更真实且运行得更快。因此如果你在大学物理书中看到有一些差别，请不要吃惊。

说得够多的了。如我在本书第1版所说的那样，这是一本*很有趣*的书！因此潜心研究它并且

① 另外也请将改进方案发送到contact@turingbook.com。——编者注

www.TopSage.com

从中享受乐趣吧！

排版约定

为了保证本书尽可能地清晰易读，下面是全书中使用的一些文本约定格式。

重要的词或概念在第一次出现时通常是以楷体表示。

代码均是以等宽字体表示，如int。新的或改变的代码通常使用**粗等宽字体**，如**int**。

伪代码和可变输入通常以*斜等宽字体*显示，如*int*。

菜单命令的书写都是按下面的形式：菜单→子菜单→子菜单。

在需要引起注意的地方，我会像下面这样强调：

> **嗨，不要说我没有提醒过你啊。**

有时候代码在一行中可能无法完全显示，这时我会使用这样的箭头：➥。

```
This is a very very long section of code that should be written all on ➥
the same line without a break.
```

致谢

我想感谢3类人。由于要感谢的人太多，为了避免漏掉任何人，这里就不一一列出名字了。

首先要感谢的是那些Flash社区的人，他们通过论坛、邮件列表、文章、教程、博客、邮件和书与大家分享知识。没有社区中强烈的分享知识的精神，这本书我甚至连想都不敢想。

其次要感谢Apress和friends of ED出版社中与本书相关的每个人。他们是一个真正的专业团队。是他们使得本书两个版本的写作都很顺利，并使此书成为一个质量很高的产品。

最后，我想感谢的是购买此书第1版的每一个读者，那些写了非常好的评论的读者，那些通过电子邮件发信或提出了许多好建议的读者。是他们使得这本书如我当初预想的那样成功。

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

目　录

第一部分　ActionScript 动画基础

第 1 章　基本动画概念……2
1.1　什么是动画……2
1.2　帧和运动……3
1.2.1　帧就是记录……4
1.2.2　程序帧……5
1.3　动态与静态……5
1.4　小结……7
第 2 章　ActionScript 3.0 动画基础……8
2.1　动画基础……8
2.2　关于ActionScript版本……8
2.3　类和OOP……9
2.3.1　基类……9
2.3.2　包……10
2.3.3　导入……11
2.3.4　构造函数……11
2.3.5　继承……11
2.3.6　MovieClip/Sprite子类……13
2.3.7　创建文档类……13
2.4　设置ActionScript 3.0应用程序……14
2.4.1　使用Flash CS3 IDE……15
2.4.2　使用Flex Builder 2……15
2.4.3　使用免费的命令行编译器……16
2.4.4　关于跟踪……18
2.4.5　缩放影片……19
2.5　使用代码动画……19
2.5.1　循环……19
2.5.2　帧循环……22
2.5.3　剪辑事件……23
2.5.4　事件和事件处理器……24
2.5.5　侦听器和处理器……25
2.5.6　动画事件……28
2.6　显示列表……29
2.7　用户交互……34
2.7.1　鼠标事件……34
2.7.2　鼠标位置……36
2.7.3　键盘事件……36
2.7.4　键盘代码……37
2.8　小结……38
第 3 章　动画中的三角学……39
3.1　什么是三角学……40
3.2　角……40
3.2.1　弧度和度……40
3.2.2　Flash坐标系统……42
3.2.3　三角形的边……43
3.3　三角函数……43
3.3.1　正弦……44
3.3.2　余弦……44
3.3.3　正切……45
3.3.4　反正弦和反余弦……46
3.3.5　反正切……46
3.4　旋转……48
3.5　波……50
3.5.1　光滑的上下运动……51
3.5.2　线性垂直运动……53
3.5.3　脉冲动画……53
3.5.4　两个角的波……54
3.5.5　用drawing API绘制波……55
3.6　圆和椭圆……56
3.6.1　圆形运动……56
3.6.2　椭圆形运动……58
3.7　毕达哥拉斯定理……59
3.8　两点之间的距离……59
3.9　本章重点公式……62
3.10　小结……63

www.TopSage.com

第4章 渲染技术……64
4.1 Flash中的颜色……64
4.1.1 使用16进制颜色值……65
4.1.2 透明度和32位色……65
4.1.3 新的数值类型：int和uint……66
4.1.4 组合颜色……66
4.1.5 提取组成色……68
4.2 drawing API……69
4.2.1 图形对象……70
4.2.2 使用clear移除绘画……70
4.2.3 使用lineStyle设定线条样式……70
4.2.4 使用lineTo和moveTo绘制线条……71
4.2.5 使用curveTo绘制曲线……73
4.2.6 使用beginFill和endFill创建图形……78
4.2.7 使用beginGradientFill创建渐变填充……79
4.3 颜色变换……82
4.4 滤镜……85
4.4.1 创建滤镜……85
4.4.2 动画滤镜……87
4.5 位图……88
4.6 载入或嵌入内容……91
4.6.1 载入内容……91
4.6.2 嵌入内容……92
4.7 本章重点公式……93
4.8 小结……94

第二部分 基本运动

第5章 速度和加速度……96
5.1 速度……96
5.1.1 向量和速度……97
5.1.2 一个轴上的速度……98
5.1.3 两个轴上的速度……99
5.1.4 角速度……100
5.1.5 速度扩展……103
5.2 加速度……105
5.2.1 一个轴上的加速度……105
5.2.2 两个轴上的加速度……108
5.2.3 重力加速度……109
5.2.4 角加速度……111
5.2.5 太空船……112
5.3 本章重点公式……116
5.4 小结……116

第6章 边界和摩擦力……117
6.1 环境边界……117
6.1.1 设置边界……118
6.1.2 移除物体……119
6.1.3 重新生成物体……123
6.1.4 屏幕折回……125
6.1.5 回弹……127
6.2 摩擦力……132
6.2.1 摩擦力，正确方法……132
6.2.2 摩擦力，容易方法……134
6.2.3 摩擦力的应用……134
6.3 本章重点公式……135
6.4 小结……136

第7章 用户交互：移动物体……137
7.1 按下和放开精灵……137
7.2 拖动精灵……140
7.2.1 使用mouseMove拖动……140
7.2.2 使用startDrag/stopDrag拖动……141
7.2.3 拖动与运动代码结合……142
7.3 抛……145
7.4 小结……148

第三部分 高级运动

第8章 缓动和弹性……150
8.1 比例运动……150
8.2 缓动……151
8.2.1 简单的缓动……151
8.2.2 何时停止缓动……155
8.2.3 移动的目标……157
8.2.4 缓动不只是应用于运动……158
8.2.5 高级缓动……159
8.3 弹性……160
8.3.1 一维弹性……160
8.3.2 二维弹性……162
8.3.3 移动目标点的弹性……163

www.TopSage.com

8.3.4 弹性在哪儿……164
8.3.5 弹性链……165
8.3.6 多目标点弹性……168
8.3.7 目标偏移……170
8.3.8 使用弹性贴加多个物体……172
8.4 本章重点公式……175
8.5 小结……176
第9章 碰撞检测……177
9.1 碰撞检测方法……177
9.2 hitTestObject和hitTestPoint……178
9.2.1 碰撞测试两个精灵……178
9.2.2 碰撞测试一个精灵和一个点……182
9.2.3 使用shapeFlag的碰撞测试……183
9.2.4 hitTest总结……184
9.3 基于距离的碰撞检测……184
9.3.1 简单的基于距离的碰撞检测……185
9.3.2 基于碰撞的弹性……186
9.4 多物体碰撞检测策略……189
9.4.1 基本的多物体碰撞检测……189
9.4.2 多物体弹性……191
9.5 其他的碰撞检测方法……195
9.6 本章重点公式……195
9.7 小结……196
第10章 坐标旋转和角度回弹……197
10.1 简单的坐标旋转……197
10.2 高级坐标旋转……199
10.2.1 旋转一个物体……200
10.2.2 旋转多个物体……201
10.3 沿角度回弹……203
10.3.1 执行旋转……204
10.3.2 优化代码……207
10.3.3 动态化……208
10.3.4 修复“脱离边界”问题……208
10.3.5 多角度回弹……212
10.4 本章重点公式……216
10.5 小结……216
第11章 台球物理……217
11.1 质量……217
11.2 动量……218
11.3 动量守恒……218
11.3.1 一个轴上的动量守恒……220
11.3.2 两个轴上的动量守恒……224
11.4 本章重点公式……239
11.5 小结……239
第12章 粒子引力和重力……240
12.1 粒子……240
12.2 重力……241
12.2.1 引力……242
12.2.2 碰撞检测和反应……244
12.2.3 沿轨道运行……245
12.3 弹性……246
12.3.1 引力与弹性……247
12.3.2 弹性结点域……247
12.3.3 结点的连接……249
12.3.4 有质量的结点……251
12.4 本章重点公式……252
12.5 小结……253
第13章 正向运动：行走……254
13.1 介绍正向和反向运动……254
13.2 开始正向运动编程……255
13.2.1 移动一个关节……255
13.2.2 移动两个关节……258
13.3 自动运行……260
13.3.1 创建自然的行走循环……261
13.3.2 动态化……263
13.4 使它真正地行走……266
13.4.1 给它一些空间……266
13.4.2 加入重力……267
13.4.3 控制碰撞……267
13.4.4 控制反应……269
13.4.5 屏幕折回……270
13.5 小结……274
第14章 反向运动：拖动和伸展……275
14.1 伸展和拖动一个关节……275
14.1.1 伸展一个关节……276
14.1.2 拖动一个关节……277
14.2 拖动多个关节……277
14.2.1 拖动两个关节……277

www.TopSage.com

14.2.2 拖动更多的关节 ……279
14.3 伸展多个关节 ……280
14.3.1 向鼠标伸展 ……281
14.3.2 向一个物体伸展 ……285
14.3.3 加入一些交互 ……287
14.4 使用标准的反向运动方法 ……288
14.4.1 余弦定律介绍 ……289
14.4.2 ActionScript余弦定律 ……290
14.5 本章重点公式 ……293
14.6 小结 ……293

第四部分 3D 动画

第 15 章 3D 基础 ……296
15.1 第3维和透视 ……296
15.1.1 z轴 ……297
15.1.2 透视 ……297
15.2 速度和加速度 ……301
15.3 回弹 ……303
15.3.1 单个物体回弹 ……303
15.3.2 多物体回弹 ……306
15.3.3 z轴排序 ……309
15.4 重力 ……310
15.5 折回 ……312
15.6 缓动和弹性 ……320
15.6.1 缓动 ……320
15.6.2 弹性 ……321
15.7 坐标旋转 ……323
15.8 碰撞检测 ……329
15.9 本章重点公式 ……332
15.10 小结 ……333
第 16 章 3D 线条和填充 ……334
16.1 创建点和线 ……334
16.2 生成面 ……340
16.3 创建3D填充 ……343
16.4 3D实体建模 ……349
16.4.1 旋转立方体建模 ……349
16.4.2 建模其他形体 ……351
16.5 移动3D实体模型 ……356
16.6 小结 ……358
第 17 章 背面剔除和 3D 灯光 ……359
17.1 背面剔除 ……359
17.2 深度排序 ……362
17.3 3D灯光 ……364
17.4 小结 ……374

第五部分 其 他 技 术

第 18 章 矩阵数学 ……376
18.1 矩阵基础 ……376
18.2 矩阵运算 ……377
18.2.1 矩阵加法 ……377
18.2.2 矩阵乘法 ……378
18.3 Matrix类 ……382
18.4 小结 ……385
第 19 章 实用技巧汇集 ……386
19.1 布朗（随机）运动 ……386
19.2 随机分布 ……389
19.2.1 方形分布 ……389
19.2.2 圆形分布 ……391
19.2.3 偏移分布 ……394
19.3 基于定时器和时间的动画 ……397
19.3.1 基于定时器的动画 ……397
19.3.2 基于时间的动画 ……400
19.4 相同质量物体之间的碰撞 ……402
19.5 声音集成 ……404
19.6 有用的公式 ……406
索引（图灵网站下载）

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

Part 1 第一部分

ActionScript 动画基础

本部分内容

- 第 1 章　基本动画概念
- 第 2 章　ActionScript 3.0 动画基础
- 第 3 章　动画中的三角学
- 第 4 章　渲染技术

www.TopSage.com

第1章 基本动画概念

本章内容

- 什么是动画
- 帧和运动
- 动态与静态

Flash本质上就是一个动画机器。从它最早的版本来看，Flash已经通过补间（tween）支持了动画。补间是指只需要创建两个不同的关键帧，而关键帧之间的工作由Flash来完成。然而，本书并不研究补间，而是研究内建在Flash中的一种强大的语言——ActionScript。书中涵盖了程序设计、数学和物理等技术，这些技术与ActionScript一起使物体运动起来。你将会获得各种层次的功能、控制和交互，使你不会再去使用补间。

在深入讨论专项技术和ActionScript让物体运动所使用的公式之前，我们先来快速地了解一下什么是动画，隐藏在它背后的一些基本技术以及在制作更动态和有趣的Flash动画时可能用到的一些概念。

当然，如果你只是想写一些代码，那么可以跳过本章，但是我仍然强烈建议你在某个时候再回头看看本章，也许能得到一些有趣的体会。

1.1 什么是动画

首要的一个问题是：什么是动画（animation）？根据《美国传统英语字典（第4版）》（Houghton Mifflin公司，2000）中所述，其动词形式animate的意思是：

(1) 使有生命，使充满生命力；
(2) 给予兴趣，给予热情，使有活力；
(3) 鼓励，激励：使充满精神，勇气或决心；
(4) 怂恿，驱使；
(5) 推动，驱动；
(6) 使栩栩如生地运动：生产、设计或制作（如卡通片）使之产生运动错觉。

虽然我能够从哲学角度理解前4种定义，但实际上我们这里探讨的是第5种和第6种定义。动

www.TopSage.com

画就意味着运动，但我想拓宽这一点，即动画是随着时间而改变的，具体来说是某种类型的视觉改变。运动本质上是物体的位置随时间改变。这一分钟它会在这里，而下一分钟它可能就会在另一个位置。理论上说，它处于这两点间的空间内，但我不想这样片面地理解（因为它不仅仅是这样）。 它移动了，而且从某一时刻的第一点移动到另一时刻的下一点消耗了一段时间。

对象不一定需要改变其位置才被认为是动画，它可能是改变其形状。还记得20世纪90年代末大行其道的图像变形程序吗？你可以让一幅图的画面从美女变成老虎，它们之间的动画由程序来创建；或改变物体的大小和方向，如一棵植物的生长过程或者某一个物体的高速旋转；甚至简单地改变其颜色，如果你接触计算机的时间比较长，可能会记起家用计算机上早期的一些动画只是颜色的循环而已。你可以用让一系列具有不同蓝色色度值的形状来制作一幅瀑布的图像，然后循环这些形状的颜色。如果你做得正确，图像看上去就像是瀑布在流动，而从技术角度上讲，其实根本没有东西在移动。

时间与动画的关系是非常重要的。没有运动或改变，就没有动画，当然也没有任何时间的感觉。有时你可能会看到摄像机镜头中的空房子场景或者城市空中轮廓线的景象，其中似乎没发生什么。你无法知道是在看一幅静止图像还是视频流。最后你可能会注意到一些微妙的变化——闪烁的灯光或者移动的影子。虽然只是轻微的闪烁，也能说明时间确实是存在的。并且如果继续观看，你可能会发现其他一些变化。在过了一段时间后，如果看不到任何改变，你就可以确信它是静止图像。但如果没有时间，你就不会感觉到画面中任何情况的发生。

这也引出了另一点：动画会让我们对一些事情保持着兴趣。虽然《蒙娜丽莎》是一幅了不起的作品，而且一直是最著名的油画之一，但我敢打赌：一般人看这幅画像15分钟以后就会变得厌烦了，他就会去看看其他可以向别人吹嘘的东西。但是如果让他看好莱坞最新的大制作动作片，那么在两个半小时中他不会有丝毫移动。即使他非常想去洗手间或者要吃点东西，也会等待一个“慢”镜头——没有很多动作的时候去。这就是动画的力量。

1.2 帧和运动

现在，让我们回过头来看一下动画的最后一个定义：

使栩栩如生地运动：生产、设计或制作（如卡通片）使之产生运动错觉。

有趣的是下定义的作者也许应该选择去掉错觉这个词，然而它却是非常准确的。碰巧的是，对于运动媒介的每一种形式，只存在一种运动错觉。这里就要接触到帧的概念了。

实际上所有的可视动画媒介都使用帧（frame）——一系列快速显示的静止图像来模拟运动或变化。在计算机、电视或电影银幕上所看到的一切都是基于帧的。这可以追溯到最早期的卡通动画时代（每一幅图像被绘制在一张玻璃纸上，被称为cel）和最早的电影（将类似的技术用于多张照片）。

概念很简单：显示大量的、相邻图像间仅有很小变化的图像，视觉上看上去就像是一幅正在移动的图像。但是为什么我们坚持把这种现象称为运动错觉？如果你在屏幕上看到一个人穿过房间，这不是运动吗？当然它只是一幅人物的图像，并不是真正的人，但是那并不是我们不将它称为真正运动的原因。

www.TopSage.com

当谈到一个物体从一个位置到另一个位置时，还记得我说它只是在空间中移动吗？好，那才是真正的运动。物体在空间中平滑地移动，而不是间隔地跳跃。（如果你是量子物理学家，请保持安静。）但是任何以及所有基于帧的运动都是在做间隔的跳跃。它并没有从一点平滑地移到另一点，它总是在下一帧的另一个位置上显示或者消失。移动得越快，跳越就越大。

如果我给你看一幅一个人在房间左边的图像，几秒之后再给你看另一幅这个人在房间右边的图像，你会说我显示的是两幅图像，并不是一个动画。如果我给你看这个人经过房间的过程的6幅图像，你也许仍然会说它只是一系列单独的图像（参见图1-1）。如果我足够快地给你看足够多的图像，不可改变的事实是它们仍只是一大堆静止的图像，但此时你也许不会再保持原来的观点了，你会真地认为这个人穿过了房间。这与最初展示两幅图像一样，没有真正的运动。但是在某
5 一点，原有的印象会被放弃而进入错觉状态。事实上，电影业已经充分地研究了这一点。

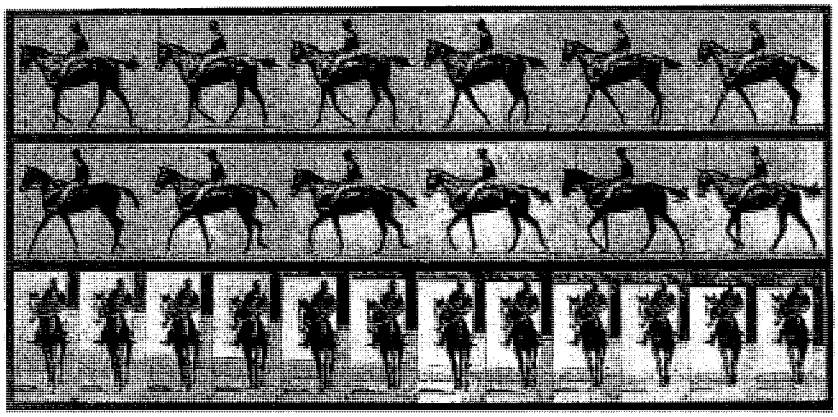


图1-1　Eadweard Muybridge制作的一系列静止图像

研究人员发现，以每秒24帧的速率，人们比较容易将这些帧识别为一幅移动的图像。太慢的速度会使跳跃的画面产生间断且令人烦躁。另外，似乎人类的眼睛也无法识别更高的帧频，所以从理论上讲，100帧每秒的速率不会让影片看上去更真实（然而在程序动画中，更高的帧频可以使交互过程响应更快，看上去更流畅，特别是对于那些快速移动的物体）。

1.2.1　帧就是记录

整个帧的概念使3件事成为可能：存储、传送和显示。你并不能真正地存储、传送和显示一个穿过房间的人。但是你可以存储一幅图像或者更多图像，并且你可以传送和显示它们。于是你可以显示动画，几乎在任何时间和任何地点，只要你拥有或能够获得存储的图像，并有办法显示它们。

现在，让我们对什么是帧进行更一般化的定义。到目前为止，我已经提到一个帧可以是一幅静止图像或者绘图。我们也可以称帧为某一时间点上某个系统的一个记录。这个“系统”可以是抓拍到的你正在笑的两岁女儿，而这个记录就是那幅图像。另一方面，系统可以是一个虚拟物体

www.TopSage.com

TopSage.com

的集合，系统记录是在某个特定时刻它们的形状、大小、颜色、位置等。于是，影片不再只是一系列静止图像，而是一系列图像信息的描述。计算机不但可以显示图像，而且可以根据图像的信息描述创建这个图像，并且显示出来。你甚至可以使用程序帧来执行进一步的操作。 6

1.2.2 程序帧

既然计算机可以快速地进行计算，实际上不需要对帧进行很长的描述。可以把它缩减为对第1帧的描述和创建后续帧的一些规则。那么现在计算机就不会单纯地依据描述来创建图像。它将先创建描述，然后创建基于描述的图像，最后显示图像。

想象一下使用此种方式可以节省多少文件空间。图像会占用许多磁盘空间和带宽，以每秒24幅图像的速度快速增加。如果能够将它浓缩为一个描述和一组规则，那么就可以成百倍地减少文件的大小。即便是针对物体移动和反应的比较复杂的一组规则，它所占用的空间也要少于一幅中等大小图像所占用的空间。实际上，对于脚本动画来说，人们关注的一个首要事情就是它最终有多小。

当然，也要折中考虑。当系统变得更大且规则变得更复杂时，计算机必须高速工作来计算下一场景的描述，且需要更长的时间来渲染它。如果试着保持特定的帧频，那么需要给CPU一定的时限（毫秒）来分析计算。如果它在时限内不能计算出这个场景，帧频将会受到影响。另一方面，基于图像的动画不会在意场景中有什么或者它有多复杂。它只是显示下一幅图像，通常会很准时。

> 我曾经从预先渲染的动画中获得过好处。我将一些复杂的Flash ActionScript动画放在一起演示。文件的大小不是问题，因为这些动画将在本地计算机上播放。但是时间是有要求的，而且我不知道在这台未知的、没有经过测试的计算机上ActionScript渲染出来的图像的平滑程度如何。因此我将Flash影片放入到Director中并且将它们导出为一个很大的QuickTime影片。因为现在的影片只是一系列预先渲染的图像，不管它有多复杂，都不会有太多的问题。只要计算机可以播放QuickTime影片，它就会保持图像的平滑。演示进行得很顺利。

1.3 动态与静态

使用代码编写动画的另一好处远远超过了简单的文件大小。事实上，代码编写的动画可以变成动态的。看过电影《泰坦尼克号》吗？电影里船最终只能沉没。它在影院里沉没，在录像机里沉没，同样在DVD中沉没。如果不按停止或暂停按钮，就不能阻止它继续下沉。那是因为电影是一系列静止图像。这个特定序列的临近结尾的图像显示船沉没，仅此而已。

我们再来看Flash网站。记得在20世纪90年代末，Flash刚开始流行的时候，每个Flash网站上都会有一个介绍网站的Flash片头。一些图形会滑入、放大或者淡出，一首音乐循环播放，一些时髦的词语淡入淡出或滑入滑出，有时还会有一束光或一些阴影出现。 7

好了，我不想过多地批评。在我印象中至少有两三个是真正值得叫好的——真正的艺术作品。Gabocorp网站的片头和Ray of Light网站的片头（如图1-2和图1-3所示）都是非常著名的。但是当

www.TopSage.com

我回过头来去想它时，我记得其实只是看过几次而已。它们有一两分钟的长度，我看过两次或三次。难道是因为它们不好吗？不，那是因为当看过几次之后，再没有更多可看的内容了。就像《泰坦尼克号》，网站片头每次播放的都是相同的内容。我称它们为静态动画（static animation）（我个人的命名），因为动画从来没有改变。从开头到结尾，每一帧都是预定义的。

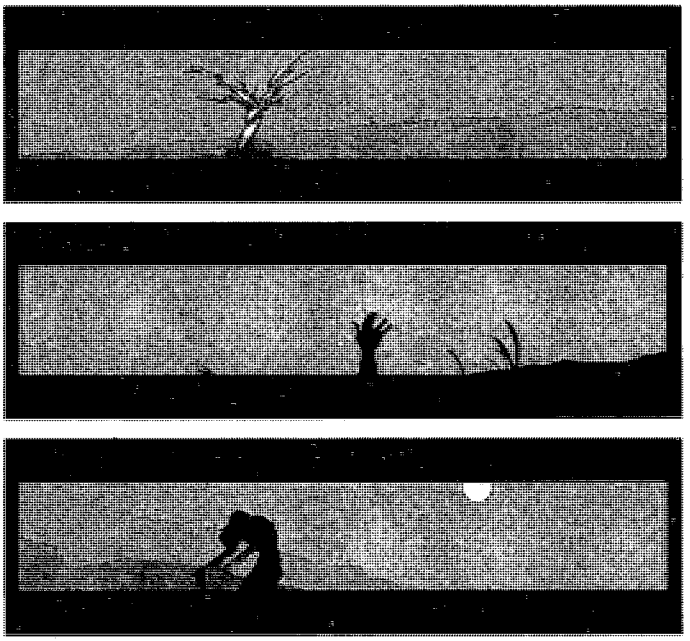


图1-2　著名的gabocorp.com片头



图1-3　rayoflight.com片头

代码编写的动画也不一定是动态的。可以选择一个物体，然后使用代码，将它放在某一个位置上并让它穿过屏幕。每次播放这个影片时，都会运行相同的代码，产生同样的移动。这不是动

www.TopSage.com

态的。

但是如果选择一个物体，依然使用代码，确定一个随机的位置放置它并以一个随机的方向和速度移动它，会怎么样呢？现在每次播放影片，你都会看到不同的画面。

或者，当影片开始时查出年月日时间，并使用这些数据创建一个场景——例如冬天的早上、夏天的下午或者4月的晚上，这会怎么样呢？

或者，在影片中加入一些在影片运行时可以通过键盘或鼠标来改变它们的因素，这又会怎样呢？这样就可以允许用户与屏幕上的物体进行交互。这些将远远超过在静态动画中所能得到的。甚至可以拯救《泰坦尼克号》！

动态动画最有趣的方面或许就是真实世界的数学和物理原理对影片中物体的应用（这也是本书的重点内容）。不仅可以让物体在随机方向上移动，还可以给它施加一些重力，使它在移动时就开始下落。当它碰到“地面”时，它会弹起，但不会弹到初始的高度。最后它会停下来并一直停留在那儿。现在对它增加一些用户交互操作，允许用户使用鼠标“拿起它”或使用键盘移动它。当将它抛出去时，用户开始感觉到他正在操纵一个实际物体。

对于这种类型的动画，用户不再只是坐在那里观看一些帧的播放，而是进入到你所创建的环境中。用户会在那里停留多长时间呢？只要这个环境一直让他感兴趣，他就会一直留在那里。他与这个环境交互的内容越多，感兴趣的时间就会越长。如果有足够的兴趣，他停留在那里的时间将远比观看片头的时间长（不幸的是，观看片头时很可能比观看《蒙娜丽莎》的时间长）。我曾收到很多人的电子邮件，他们告诉我他们花了整个早晨或者下午来玩www.bit-101.com上的实验作品。人们不但会留在那里很长时间，而且还会经常来。

8~9

1.4 小结

那么，所有这些内容的要点在哪里呢？这个开篇，已经介绍了一些动画的基础知识。但是你实际能用它做什么呢？这完全取决于你。

在下面的章节中，我会提供一些工具并快速教你如何使用它们。使用这些工具做什么完全由你决定。本书中最明显的应用就是创建游戏。游戏本质上是设定某些目标由玩家来完成的交互动画。但是我不想让它变成一本游戏书。我已经在某种专业工作（除了游戏）中使用了这里几乎所有的技术——从很炫的3D菜单和一些看上去不错的导航系统，到广告和教育的应用。

有一个警告：拿起任何一本关于网页设计的书，你将会发现有一章在告诉你动画是如何不好。我并不同意这一点，但是对此我并不想多说什么。如果你想了解动画，接下来的几百页内容将为你提供所需要的全部知识。

10

www.TopSage.com

第2章

ActionScript 3.0动画基础

本章内容

- 动画基础
- 精灵
- 类和面向对象编程（OOP）
- 用户交互

如果说第1章是有点抽象的动画概述，那么这一章则是关于用ActionScript制作动画的各种技术概述，尤其是ActionScript 3.0（AS 3）。这一章将涉及动画循环、影片剪辑和精灵（Sprite）、类以及用户交互的一些要点。它提供了理解本书后面用到的ActionScript技术的大部分信息。

2.1 动画基础

在开始之前，我们先快速回顾一下第1章所讲述的内容。

- 动画是由帧生成的，每一帧都略有不同以造成运动错觉。
- 逐帧或补间动画包含每一帧的一幅图像或一幅图像的描述。
- 动态动画包含一幅图像的初始描述和用于改变每一帧描述的规则。

本书后面的大多数内容集中关注动态动画的规则，提供了一个技术列表来改变图像的描述，并得到逼真的动画结果。在本章中，你将会明白如何组织初始描述、如何在每一帧上应用规则以及如何将它们整合在一起。在这个过程中，你将创建大量的应用例子。

2.2 关于 ActionScript 版本

本书的第1版是在Flash 8发布后不久出版的。虽然在我印象中它是使用ActionScript 2.0语法的，但尽量避开了许多难懂的OOP理论。实际上，许多代码可以直接放置在时间线上使用。结果看上去有些ActionScript 1.0和ActionScript 2.0混杂的现象。

本书使用的是Flash 9 和AS 3。基于这一点，至少有以下3种方式来创建一个Flash 9、AS 3影片：

- Flash CS3 IDE。

www.TopSage.com

- Flex Builder 2。
- 免费的Flex/AS 3命令行编译器和Flex 2软件开发包。

在Flash CS3 IDE的时间线上书写AS 3仍是可能的。而其他两种方法至少需要用到一个AS 3类文件。因此，为了保持一致性，我决定抛开时间线。本书中的所有例子都将使用AS 3类。 14

这一章，将说明如何在刚刚提到的每一种开发环境（Flash CS3、Flex Builder 2和免费的SDK）中建立一个AS 3项目。在本书的其余部分，只有可应用的类代码会列出来，并且它可以工作在3种开发环境中。

有时候，当描述一个简单概念时，我可能会给出一小段代码来表明其思想。虽然它没有完全明确地表明，但可以认为这段代码要插入到本章将要列出的类架构中。

鉴于类是书中示例非常重要的部分，我将很快深入讨论它们。尽管如此，本书并不是一本深入的OOP参考书。关于类实际上只是给出需要组织、运行和创建本书中示例的最基本内容。

2.3 类和 OOP

一些读者可能不知道什么是类或者“面向对象（object oriented）”这个术语的含义，而其他人可能已在ActionScript（或其他语言）中使用类多年了。为了适应所有人，我将尽量讲述所有的基础内容。有经验的AS 2 OOP开发人员可能也需要略读一下本节，因为在AS 3中有相当一部分的工作方式改变了。

如果你认为对类一点都不熟悉，那你可能错了。如果你曾在Flash中写过任何代码，就有可能使用过一些类。类（class）就是指一个对象的类型。MovieClip是一个类，它用来表示——你可能已经猜到——影片剪辑。文本域、影片剪辑、按钮、字符串和数字都有对应的类。

通常一个类有两项内容与之相关：属性（数据或信息）和行为（动作或它可以做的事情）。属性本质上是存放与类相关的信息的变量，行为相当于函数，而当函数是类的一部分时，我们通常称它为*方法*（method）。

2.3.1 基类

如果你使用Flash已经有一段时间了，就会知道可以在库中创建一个元件符号，然后在舞台上创建元件符号的几个实例。类似于元件符号与实例的关系，类是模板，而对象（也可以理解为实例）是指定类的个体表现。可以编写一个如下的简单类：

```
package {
   public class MyClass {
      public var myProperty:Number = 100;

      public function myMethod() {
         trace("I am here");
      }
   }
}
```

15

让我们快速浏览一下这段代码。首先，有些东西甚至对于很熟悉AS 2类的人员来说都是新的：

www.TopSage.com

大家网 TopSage.com

包声明。简单地说，包（package）就是把相关的类组合起来的方式。稍后将进一步讨论包，但是本书中的大部分示例都没有使用包。然而仍需要像示例中那样包括package关键词和大括号。这也被称为默认包。

下一步是类自身的定义。相对于旧版本的另一处改变是：类现在有了访问修饰符。访问修饰符是一个词，它指示哪些其他代码可以访问正在讨论的代码。这里的public（公开）意味着这个类可以被类外部的任意代码访问。在本书的所有示例中，类都将是公开的。如果想学习关于AS 3的更多内容，你可能会发现一些非公开类和一个类文件中的多个类，但是这些内容已经超出了本书的范畴。

可以看到类有一个名称MyClass，紧跟着是另一对打开和在类结尾处闭合的大括号。

在类的内部只有两部分内容：一个名为myProperty的变量和一个名为myMethod的函数。这些将变成你所创建的任意类实例的属性和方法。同样，对于属性和方法，public修饰符意味着这个对象之外的任何代码都可以访问这个属性和调用这个方法。如果你创建的属性或方法只用于类的自身使用，你可以将它们标记为private（私有），它会阻止类外部任何代码的访问。AS 3中还加入了internal（内部）和protected（受保护）修饰符。一个类的internal属性只能被同一个包中的其他类访问，protected属性意味着它只对扩展这个类的哪些类可见（这一点与AS 2中的私有属性非常相似）。

还没搞清楚？别着急，我过滤掉了复杂内容的一大部分，现在描述的许多内容并没有全部用在本书中。书中的大部分示例使用的是很小的类，一旦你看到一些类的执行过程，就会明白其中的意思。

类在编写完成后，将保存在一个外部的文本文件中，文件名与类名相同，使用的后缀为.as，如MyClass.as。可以使用Flash CS3 IDE和Flex Builder 2中的ActionScript编辑器、你喜欢的代码编辑器，或者其他任何文本编辑程序来创建类文件。

如果使用的是Flash CS3 IDE，则还需要创建一个FLA文件。如果使用的是默认包，则所创建的类文件必须与FLA文件位于同一目录下。如果你使用的是包结构，那么包结构的根目录应当与FLA位于同一目录。另外，可以将这些类保存在其他地方，然后将目录加入到类路径中。类路径（class path）只是一个简单的目录列表。当在代码中指定了一个类的名称，Flash将会在那些目录中搜索所指定名称的类。另外，在本书的示例中，将使用默认包，并且可以保证类与FLA文件始终在一起。

2.3.2　包

包的主要作用是用来组织类。包是根据它所在目录以及所嵌套的层级来构造的。包中的每一个名称对应一个真实的文件夹，这些名称通过点来隔开。例如，有一个名为Utils的类，它在下面的一系列文件夹中：com/friendsofed/makingthingsmove/。（通常是将域名作为包。这样可以确保包是独一无二的。）这个类将记作com.friendsofed.makingthingmove.Utils。

在AS 2中，需要使用完整的包名称来声明类，像下面这样：

www.TopSage.com

```
class com.friendsofed.makingthingsmove.Utils {
}
```

在AS3中，包部分代码用来声明包，类部分代码用来声明类，如下：

```
package com.friendsofed.makingthingsmove {
    public class Utils {
    }
}
```

2.3.3 导入

你可能已经想象到了，每次要使用某些实用函数时都要键入com.friendsofed.makingthingsmove.Utils，这种方式很快就会过时。用导入（import）方式可以缓解这种问题。在这个例子中，可以把下面的代码放置在包的顶部，类自身的前面：import com.friendsofed.makingthingsmove.Utils;。

这句只需要书写一次。在这之后，这个文件中只需要写Utils来引用那个类。现在，在AS 2中导入已经是很方便了，但是在AS 3中，必须从不同的包中导入所需要的任何类，即使每次都写完整的名称，否则代码将无法编译。Flex Builder 2实际上有一个非常好的特点，就是当键入类的完整的包名称时，它会自动地添加导入语句。在任何情况下，都要确保导入了类。当你编译影片时确保编译可以执行。

2.3.4 构造函数

可以为类设置一个构造函数（constructor），它是一个与类名称相同的方法，当一个新的实例被创建时，这个方法将自动地被调用。可以按下面的方式向构造函数传递参数。

首先，创建类：

```
package {
   public class MyClass {
      public function MyClass(arg:String) {
         trace("constructed");
         trace("you passed " + arg);
      }
   }
}
```

然后，假设你工作在Flash CS3 IDE下，在时间线上创建一个类实例如下：

```
var myInstance:MyClass = new MyClass("hello");
```

17

它将显示出两条信息"constructed"和"you passed hello"。如果你使用的是Flex Builder 2或是免费的SDK，那么先记住我这里所说的话。稍后会有大量的操作实践。

2.3.5 继承

一个类可以从另一个类中继承（inherit）或者说扩展另一个类。这意味着它可以获取另外一个类的所具有的所有属性和方法（除非属性或方法被标记为私有）。子类（subclass）（继承其他

www.TopSage.com

类的类）可以增加额外的属性和方法，或者改变父类（superclass，也称基类，base class，即被扩展的类）中的一些内容。可以通过创建如下两个类（分别放在两个.as文件中）来实现继承关系：

```
package {
    public class MyBaseClass {
        public function sayHello():void
        {
            trace("Hello from MyBaseClass");
        }
    }
}

package {
    public class MySubClass extends MyBaseClass {
        public function sayGoodbye():void {
            trace("Goodbye from MySubClass");
        }
    }
}
```

记住每个类都需要一个文件，文件名为类名称加上.as后缀，因此会有一个MyBase Class.as文件和一个MySubClass.as文件。再强调一次，如果使用的是Flash CS3 IDE，则要创建一个新的FLA文件并和那两个类保存在同一个目录中。现在，在时间线上创建两个类实例看看会发生什么。

```
var base:MyBaseClass = new MyBaseClass();
base.sayHello();

var sub:MySubClass = new MySubClass();
sub.sayHello();
sub.sayGoodbye();
```

第1个实例没有什么惊奇之处。但是注意第2个实例有一个sayHello方法，尽管MySubClass没有定义sayHello。这个类继承自MyBaseClass。还需要注意到它增加了一个新方法sayGoodbye，而基类（MyBaseClass）并没有这个方法。

接下来，假设想在子类中更改基类中已经存在的方法。在AS 2中，只需要简单地重新定义它。在AS 3中，重新定义时必须通过使用override（覆写）关键词来覆写它。

18

```
package {
    public class MySubClass extends MyBaseClass {
        override public function sayHello():void {
            trace("Hola from MySubClass");
        }
        public function sayGoodbye():void {
            trace("Goodbye from MySubClass");
        }
    }
}
```

注意那个sayHello，当从MySubClass调用时，现在已经有一个全新的信息，因为原有的方法已经被覆写。当然，要知道私有方法是不能被覆写的，因为它只允许在其类内访问。

www.TopSage.com

2.3.6 MovieClip/Sprite 子类

你可能写过类，并写过扩展它的另一个类，也可能从没有写过类。但是如果你使用AS 3而且抛开了时间线来做任何东西，最终都要扩展MovieClip或Sprite类。MovieClip类是所有ActionScript属性和方法的模板，ActionScript属性和方法都是影片剪辑对象的一部分。它所包含的属性你可能已经很熟悉了，如剪辑的*x*和*y*位置、缩放等，尽管许多属性已在AS 3中有了轻微的改变。AS 3也增加了Sprite类。通常可以把一个精灵理解为一个没有时间线的影片剪辑。在多数情况下，当你只用代码操作物体时，就不会处理时间线和帧。那么此时适合使用比较轻便的Sprite类。

如果编写一个扩展MovieClip或Sprite的类，它将自动继承这种类型的对象固有的所有属性和方法。然后可以为所创建的对象增加具体的行为或属性。例如，假设你想为一个游戏创建一个太空船对象。你可能想让它包含一些图像、在屏幕上有一个位置、来回移动、旋转、侦听enterFrame事件来动画，且侦听键盘和鼠标事件来交互。所有这些事情影片剪辑和精灵都可以做到，因此理应继承它们中的一个。然后你可以加入自定义的属性如speed、fuel和damage，以及自定义的行为如takeOff、crash、shoot和SelfDestruct。这个类在编写开始时大致如下：

```
package {
   import flash.display.Sprite;

   public class SpaceShip extends Sprite {
      private var speed:Number = 0;
      private var damage:Number = 0;
      private var fuel:Number = 1000;

      public function takeOff():void {
         // . . .
      }
      public function crash():void {
         // . . .
      }
      public function shoot():void {
         // . . .
      }
      public function selfDestruct():void {
         // . . .
      }
   }
}
```

19

注意在最开始时必须要先导入Sprite类，它存在于flash.display包中。如果决定扩展MovieClip类，那么同样需要从相同的包中导入MovieClip类：flash.display.MovieClip。

2.3.7 创建文档类

现在已经掌握了足够多的关于类的信息，可以开始编写一个实际使用的类。我已经几次提到过在创建基于AS 3的SWF时类是多么重要。这是因为在AS 3中引入了一个新的概念：*文档类*（document class）。总的来说，文档类就是扩展Sprite或MovieClip类，并且用作你的SWF的主类

www.TopSage.com

的类。当SWF被载入时，这个类的构造函数将会被调用。这将是你想在影片中做任何事情的入口点，如创建附加的影片剪辑、绘制图表、预载素材等。

如果你正在使用Flash CS3 IDE编写代码，那么是否使用文档类是可自由选择的。可以总是只在时间线上编写代码。但是如果使用的是Flex Builder 2或免费的Flex SDK，它们并没有可以编写代码的时间线。使代码处于SWF中的唯一方法就是把代码放置在类中。在那些工具中，每件事情都是围绕着强大的文档类，没有它也就不会有SWF。

下面是前面提到的文档类的架构：

```
package {
   import flash.display.Sprite;
   public class Example extends Sprite {
      public function Example() {
         init();
      }
      private function init():void {
         // 范例代码。
      }
   }
}
```

如果你是一步一步地在学习本章，那么这段代码里面没什么新东西，但是这里将所有东西放在了一个地方。我使用的是默认包，而且导入和扩展了Sprite类。构造函数只有一行代码，它调用init方法。将所有代码都放入构造函数中当然好，但是这通常被认为是用来限制构造函数中的代码数量的很好的习惯。因此我把它变为另一种方法。在本书中，如果我提供了用来试验的小段代码，你要使用所讲述的类将代码段正确地插入到init方法中。当影片被编译和运行时，构造函数就会调用init方法，它会运行你的代码。

现在要知道的事情就是如何将这个文档类与将要创建的SWF链接起来。

2.4　设置 ActionScript 3.0 应用程序

回顾之前所讲到的内容，主要有3种创建AS3 SWF的方法：

- Flash CS3 IDE；
- Flex Builder 2；
- 免费的Flex /AS 3命令行编译器和Flex 2 SDK。

以上每一种方法都是以一个文档类开始，以一个SWF结束，但是如何设置和使用每一种工具是非常不同的。我们每次介绍一种，从最简单的（Flash CS3 IDE）到最复杂的（免费的编译器和SDK）。我将使用下面的文档类，它只是简单地在舞台上绘制一个红色的圆，以作为它存在的证明。

```
package {
    import flash.display.Sprite;
    public class Test extends Sprite {
        public function Test() {
            init();
        }
        private function init():void {
```

www.TopSage.com

TopSage.com

```
            graphics.beginFill(0xff0000);
            graphics.drawEllipse(100, 100, 100, 100);
            graphics.endFill();
        }
    }
}
```

注意这与我刚刚给出类架构是一样的，只是在init方法中带有几条绘制命令。

2.4.1 使用 Flash CS3 IDE

Flash CS3 IDE是实现文档类的最容易的工具。只需要将之前的类保存到你所选择的目录下，文件名为Test.as。然后使用Flash CS3创建一个新的FLA文件并将它保存在与类相同的目录下。确保FLA的默认发布设置为Flash Player 9和AS 3。在Properties面板上，应注意到一个称为Document Class的新的域（见图2-1）。只需要简单地输入类名：Test。

21

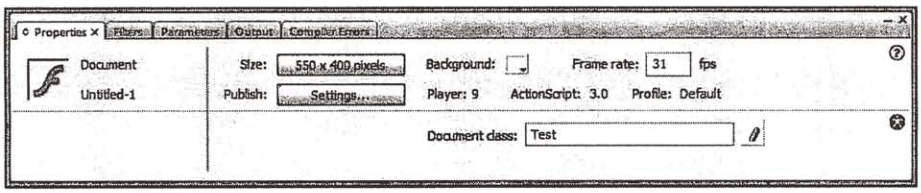


图2-1 设置文档类

注意输入的是类名，而不是文件名，因此不需要加.as后缀。然而，如果类在一个包中，则需要输入类的完整路径——例如com.friendsofed.chapter2.Test。

现在就可以按你多年使用的方式简单地测试或者发布你的影片。Flash将会查找那个类并将它编译为SWF。当SWF运行时，构造函数和init方法将会执行，那么你将得到可爱的红色的圆。

当使用一个文档类时，尽量避免在FLA的主时间线上加入任何代码——即便只是注解——因为这样可能会导致与编译器加入的代码产生冲突，会阻止影片的正常编译或运行。

2.4.2 使用 Flex Builder 2

我刚才说过Flash CS3 IDE是创建AS 3应用程序的最简单的方法。然而，事实上这主要是因为它对许多Flash设计师或开发员来说很可能是最熟悉的工具，他们可能已经使用这个工具多年了。

然而真实的情况是，使用Flex Builder 2创建AS 3应用程序可能一样简单，只要你已经安装了并习惯了工具本身。你甚至不需要担心创建多个文件（.as和.fla）和确认它们是否保存在了正确的位置。你要做的只是创建类并编译它。

假设你已经安装了Flex Builder 2并且已打开正在运行，首先创建一个新的ActionScript项目。在New ActionScript Project对话框中输入一个项目名称然后点击Finish。以后当你比较熟练时，你可以再研究其他的对话框，它提供了许多种自定义项目的其他方法，但是现在需要的只是你的项目名称，它将要变成你的文档类的名称。（注意，文档类这个术语并不会真正在Flex Builder 2工

具中出现。我在3种开发环境中使用它只是为了保持一致性。）

Flex Builder 2将会为你的项目创建一个目录（除了其他一些目录和子目录），并且会创建大部分类架构。你所需要做的就是加入init方法并从构造函数中调用它，然后在init方法中加入你想加入的任何代码。

然后点击工具栏上的Run按钮，就是里面带箭头的那个绿色圆环。它将会把你的类编译成一个SWF，并创建一个HTML文件来承载它，然后在你的默认浏览器下运行。非常简单，不是吗?

如果你希望为你的SWF指定不同的参数，如帧频、大小和背景色，在Navigator区右击项目名
22 称，然后选择Properties。在Properties对话框中选择ActionScript Compiler。在对话框的当前页中，你会看到一个域为Additional Comiler Arguments（额外的编译器参数）。在这个域中，可以输入下面的参数来改变你发布的SWF的属性：

- default-size *width height*;
- default-frame-rate *fps*;
- default-background-color *0xRRGGBB*。

只需要将它们都写成一行，替换为你想要的实际数值，如图2-2所示。

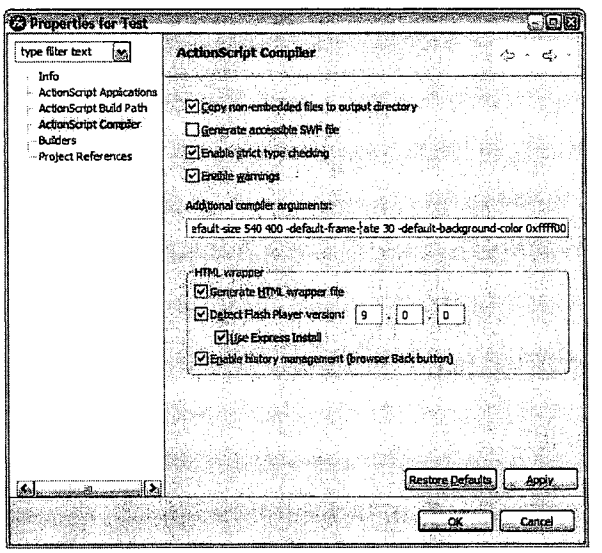


图2-2　添加编译器参数

有许多其他的命令行参数也可以在这里添加，它们都可以在Flex Builder 2和AS 3的文档中找到。但是上面列出的那些是现在你最常用到的。

也可以在文档类内部通过元数据来设置这些值。这种方法将在下一节的结尾处讲解。

2.4.3　使用免费的命令行编译器

非常抱歉，这次我不能再说这种方法比其他两种方法更简单了。它基本上要自己动手来编译

AS 3。但是好在它是完全免费的！像大多数命令行工具一样，命令行编译器也具有可以被其他应用程序调用的好处，因此你可以在惯用的编辑器中进行设置，让它以快捷方式运行，作为Apache Ant构建过程的一部分，等等。在这一节中我只是简要地描述编译器的编译过程。希望掌握了它的要领之后，可以知道怎样将编译器与其他开发工具集成。写这本书的时候，免费的ActionScript编辑器FlashDevelop（可以在www.flashdevelop.org得到）已经开始实现对AS 3的支持了。当这本书上架时，毫无疑问还有其他的。 23

你需要做的第1件事情是从www.adobe.com上下载免费的Flex 2 SDK。它是一个zip压缩包，你可以随时对它解压缩。最好是将它放在一个容易访问的地方，如PC机上C:\flex_2_sdk或者在Mac机器上/Applications/flex_2_sdk。

编译器自身其实有3个版本。有一个是mxmlc.exe，用于windows系统可执行的编译器。有一个是mxmlc，为Mac应用程序。这两者都在你所安装的主目录下的bin目录下。最后，还有一个是mxmlc.jar，为基于Java的编译器，它将工作在Mac或Pc上。Java编译器在与bin同一级的lib目录下。如果你准备好研究Java，你可以试一下Java版本，但是对于它如何被调用和它如何处理文件路径都会有些不同，因此为了更容易解释，我决定讲解bin目录下的应用程序，我发现它工作起来会更容易一些。

> 值得注意的是Adobe在Mac和PC版本制作上确实做得非常好，两种版本几乎是以同样的方式运行的。除PC版本有.exe文件扩展之外，其他大部分是难以区分的，这使得我在编写工作上更加容易！

既然已经安装好了你的编译器，就可以像之前描述的那样创建并保存你的类。因为免费的SDK没有任何类型的编辑器，所以你需要使用一种你认为比较舒服的编辑器，并将文件保存为普通的文本。然后打开一个命令窗口或者终端，导航到保存类文件的目录下，然后运行编译器，在命令行将类的路径作为参数传递给编译器，如下：

```
C:/flex_2_sdk/bin/mxmlc.exe Test.as
```

或者在Mac系统中如下：

```
/Applications/flex_2_sdk/mxmlc Test.as
```

当编译器工作时它会抛出一系列的信息，直到结束。如果在编译过程中有错误，它会告诉你相关的内容可能的位置，它会精确指出发生错误的行和字符。

现在你在与类相同的目录的下就会得到一个SWF文件，名为Test.swf。运行这个SWF，你就会看到一个红色的圆。祝贺你，你现在已经是一个很强的AS 3用户了！

当然，每次测试影片时需要键入比较长的mxmlc路径，因为这个原因，一些人不愿意去使用命令行工具。意识到有许多种方法可以让这个过程自动化是很重要的。例如，几乎每一个好点的代码编辑器都会允许你将工具设置为快捷键或工具按钮，通过参数设置来完成。你只需要将编译器的路径设置为编辑器中的工具，当前文档的路径设置为参数即可，这样你就可以做到一键编译了。 24

另一种自动化编译的方式是使用批处理文件或复制你手动输入内容的AppleScript文件。你只需要运行那个批处理文件或脚本文件，它会为你做所有的工作。如果你确实很勤奋，你可以深入

www.TopSage.com

Apache Ant，它是专为此目的而设计的一个程序。创建一个描述你的编译器路径和传递给编译器的各种参数的XML文件，Ant会解析它并运行程序。实际上它可以做的事情远不止这些，因此如果你在使用AS 3做大型的开发，Ant是非常值得学习的。

现在已经有很多不同的编辑器可以选择使用了，每个人都有他或她自己的开发环境设置偏好，因此我不能逐一详细地介绍如何设置这些编辑器。但是，如果你查看我的个人博客，地址是www.bit-101.com/blog，你将会发现很多关于使用Ant和其他的一些工具将FlashDevelop编辑器与AS 3集成使用的信息。

你可能已经注意到了，在刚才描述的设置中，没有关于设置基本的文档属性如SWF的大小、帧频、或背景色的方法。有两种方法可以完成这种设置。第1种方法是通过额外的命令行参数，重要的几个参数如下：

```
-default-background-color=color
-default-frame-rate=fps
-default-size=width,height
```

你只需要将它们加在类名称的后面，命令行的末尾。

对于PC机如下：

```
(path)mxmlc.exe Test.as -default-background-color=0xff0000
```

对于Mac如下：

```
(path)mxmlc Test.as -default-size=800,600
```

注意，你可以加入所需要的更多的参数，只是在命令行中每个参数使用空格隔开。这就是自动化操作得心应手之处。

另一种设置基本文档属性的方法是通过在类中正确的位置上加入元数据。*元数据*是由不是可编译的ActionScript语句，但是确实在编译过程中被编译器使用的任何语句组成。SWF元数据标签允许你在类文件中设置前面4种属性，像下面这样：

```
[SWF(backgroundColor="0xffffff", width="800",
    height="600", frameRate="31")]
```

这一行放置在包内，紧贴在类声明的前面。

2.4.4　关于跟踪

在许多示例中，特别是本书较前面的章节，我会指导你输出一些用于跟踪的值或信息，如前面给出的MyBaseClass和MySubClass示例中。如果你使用的是Flash CS3 IDE，这些示例都会正常工作，除了需要编写跟踪（trace）语句之外不需要做任何事情，而且跟踪信息都会显示在Flash的Output面板上。

如果你使用的是Flex Builder 2，也可以按如前面所述使用trace。有一个不同的地方是需要调试（debug）你的影片，而不是运行（run）它。这非常简单，只需要点击工具栏上的Debug按钮而不是Run按钮。影片将会在外部的浏览器上运行，但是如果切换回Flex Builder，跟踪信息将显示在控制台（console）上。

www.TopSage.com

最后，如果你使用的是免费的Flex 2 SDK，你需要多做点工作以使它可以输出信息。可能最简单的方法是使用第3方的跟踪面板。它们通常包括两部分：一个是可执行程序或包含一个用于显示跟踪信息的文本区域的SWF，另一个是一个自定义的trace类。必须将trace类放在你的类路径中。编译时使用-source-path命令行参数来增加类路径。例如，假设你把trace类放在C:/AS3Classes/中。编译时可以增加如下的命令行参数：

```
-source-path="C:/AS3Classes/"
```

它将会允许编译器在你的项目中编译trace类。

然后，当你想要跟踪时，一般需要调用为你使用的跟踪面板所指定的方法。例如，不能按下面的方式书写：

```
trace("Hello world");
```

你可能需要这样写：

```
Logger.debug("Hello world");
```

当trace类在一个嵌套的包中时可能需要先导入它。当然，需要先查阅关于如何使用你选择的跟踪面板的相关文档。我发现一个很漂亮而且非常容易使用的面板叫Xpanel，它可以在www.ajaxmaker.com/xpanel/xpanel.htm上获取。

如果你使用的是免费的SDK和一个跟踪面板，记得要用它使用的方法来取代所有使用trace的示例中的trace。

2.4.5 缩放影片

最后一个是我们在编码前要注意的设置问题：如果你是在Flash IDE或独立的Flash Player中测试影片，它将会进行100%的缩放或以自身实际大小来显示。但是如果你在一个浏览器中或被设置为用来显示SWF的其他程序中测试，它可能会发生一些缩放来充满可用空间，会扭曲你的影片中的内容。如果有这样的事情发生，可以在类的init()方法中加入如下两行：

```
stage.scaleMode = StageScaleMode.NO_SCALE;
stage.align = StageAlign.TOP_LEFT;
```

这样将会阻止影片的缩放并且始终把它放在窗口的左上角播放。如果使用上面的代码，要确保导入flash.display.StageScaleMode和flash.display.StageAlign类。在本书的示例中没有使用这些代码，因为在所有的例子中它们并不适用，但是要知道当需要时你可以使用它们。而且它们不会有任何负面影响。

26

2.5 使用代码动画

好，现在你应当已经理解了关于如何在AS 3编写代码的足够多的基础知识。假设你已经选择了一种开发环境，就准备好开始吧。下面让我们接触一些ActionScript动画内容。

2.5.1 循环

几乎所有用代码编写的动画都包含某种循环。如果你想到一个逐帧动画，你可能会提出有一

www.TopSage.com

大家网 TopSage.com

个流程图，它是像一系列位图一样的内容，每一帧都已经是一幅图像并且只是需要显示而已，如图2-3所示。

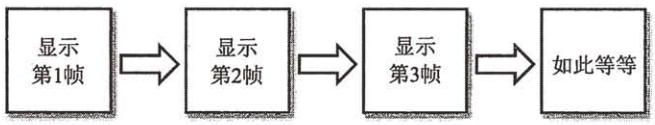


图2-3　逐帧动画

当使用Flash中的图形或元件符号时，情况会有些不同。Flash不能对每一帧创建和保存一张新的位图图像，即使是在逐帧影片中。对于每一帧，Flash保存舞台上每个对象的位置、大小、颜色等等。因此，如果你有一个穿过屏幕的小球，那么每一帧将会保存当前帧上小球的位置。可能第1帧的小球位于距左边10个像素的位置，第2帧中可能就是15个像素，如此这样。Flash播放器读取到这些数据，根据给定描述来设置舞台，并且显示帧。根据这些，可以得到另外一个扩展后的流程图，如图2-4所示。



图2-4　渲染和显示帧

但是当你思考我是如何描述一个动态的、用代码编写的动画时，它的流程图看起来更像图2-5。

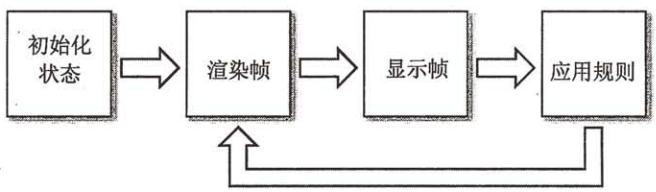


图2-5　脚本动画

如图2-5所示，没有第1帧、第2帧等的概念。ActionScript动画一般可以做到而且通常都是只在一帧上发生所有的事情。这里你可能开始明白我说的循环了。

首先设置一个初始状态。你可拖动一个影片剪辑到舞台上，如同要创建一个补间动画。或者可以只用代码描述整个场景。不管是哪种方法，接下来都要渲染和显示帧。

然后就是应用你的规则。规则可以像“这个小球将向右移动5个像素”一样简单或者它们可以由具有复杂的三角学的许多线组成。本书中的示例将会涉及这些范围。

应用这些规则将会得到一个新的状态——一个新的、之后会被渲染和显示的描述。然后再应用相同的规则。

注意那是相同的一组规则被反复使用。你不需要为第1帧设置一组规则，然后再为第2帧设置

www.TopSage.com

另一组规则，如此等等。因此，你的挑战是提出一组规则，它可以应付你场景中出现的所有可能的情况。当小球移动到了右边并移出屏幕怎么办？你的规则需要考虑到这种情况。是否允许用户使用鼠标与这些小球进行交互？你的规则同样需要将它考虑进去。

这听上去有点可怕，但实际上它并没有那么复杂。基本上可以先创建一两个规则来制作一些非常简单的行为，当它们可以正常工作时，再加入其他的规则。

我一直说的这些“规则”实际上是ActionScript语句。每一个规则可以是一个声明或由几个甚至许多语句组成。在小球向右移动5个像素这个例子中，它的规则看上去如下：

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

这只是说选取小球的任意x位置（水平轴），增加5就产生了一个新的x位置。甚至可以更简单地写如下代码：

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

+=操作符只是将右边的值增加到左边的变量中，并将结果赋给变量。

下面是本书中稍微高级一点的一组规则的应用，稍后会在书的后面看到。

```
var dx:Number = mouseX - ball.x;
var dy:Number = mouseY - ball.y;
var ax:Number = dx * spring;
var ay:Number = dy * spring;
vx += ax;
vy += ay;
vy += gravity;
vx *= friction;
vy *= friction;
ball.x += vx;
ball.y += vy;
graphics.clear();
graphics.lineStyle(1);
graphics.moveTo(ball.x, ball.y);
graphics.lineTo(mouseX, mouseY);
```

28

现在不用担心它们是什么意思，只要知道Flash将需要反复运行这段代码来生成每一个新的帧就可以了。

那么，如何让这些循环运行呢？我将展示一下我的第一次尝试，它也会是许多ActionScript的初学者容易出现的错误。它是基于几乎所有的程序语言中都有的循环结构，如for和while。可以使用它们中的一种结构来设置循环，这会使它里面的代码重复执行。下面是我所做的：

```
for(i = 0; i < 500; i++){
    ball.x = i;
}
```

它看上去非常简单。变量i的初始值为0，所以小球影片剪辑放在x轴上0的位置——舞台的最左边。i++会使变量i在循环过程中每次都增加1，从0到1，到2、3、4，如此等等，而且每执行一次，它的值就会赋给ball.x，让它从舞台的左边移到右边。当i达到500时，i<500的语句就会为假，循环就会停止。

如果你也犯过同样的错误，你就会知道小球是不会穿过舞台的——它只是简单地出现在右

www.TopSage.com

边。为什么没有发生中间的移动过程呢？实际上它做了！只是你没有看见而已，因为你从来没有让Flash更新屏幕。图2-6是另一个流程图来显示所发生的本质。

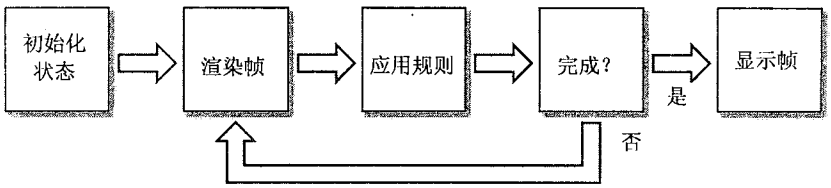


图2-6　为什么不能使用for循环来制作动画

你确实应用了规则并且将小球移动到了相应的位置，创建了500次的新场景。但是直到循环结束时它才得以显示。这是因为Flash只是在最后一帧的时候才更新了屏幕。这是非常重要的一点。

下面是Flash对每一帧执行的动作序列：

(1) 在舞台上放置好所有物体的位置，包括任何级别、任何层、载入的影片剪辑等。

(2) 执行那一帧上所有需要被调用的ActionScript，包括任何层或级别上的任何影片剪辑或按
29 钮，或是嵌套的影片剪辑，无论有多深。

(3) 检查是否到时间显示帧了。如果你设置了帧频为20帧每秒，Flash将检测从上一次显示一帧之后是否至少过去了50毫秒的时间。如果是，它将显示刚刚渲染的这一帧并且移动到下一帧。如果不是，它将等待直到正确的时间量已经过去。

你需要知道一些关于时间因素的问题。首先，帧频从来都是不准确的（即便是在Flash 9影片中）。不要指望它们会有严格的计时。其次，很可能出现渲染或ActionScript执行的时间超过分配给每一帧的时间。但是，不必担心你的脚本会被裁掉。Flash会在移到上面的第(3)步之前完成脚本的执行（第(2)步），尽管它会使帧频减小。Flash其实会在任意特定的帧上为完成你的脚本等待多达15秒。然后它会显示出一行提示信息：A script is causing the player to run slowly...（一个脚本导致了播放器运行缓慢……）。

在前面的示例中，Flash会等待for循环结束后才进入下一帧，并且它只在进入下一帧之前才更新屏幕。这就是为什么你只看到了一个跳跃，而没有看到移动过程。

因此，你所需要做的就是通过帧来结束这种循环，这样你就可以回到图2-5所示一系列事件。

2.5.2　帧循环

因为这本书并不是讲关于处理时间线和单独某一帧上的内容，所以我不打算给出帧循环的例子，而只是描述这个概念。但是，如果你使用的是Flash CS3 IDE，这对于你自己创建这个示例来说可能是一个很有意义的一步。

说到帧循环的思路，我们要回到Flash的最早版本，当时ActionScript并不像今天这样强大。它的思路是将一些代码放在一个指定的帧上，然后在下一帧上放一个语句，通常是gotoAndPlay，它可以让播放头回到前一帧上。这样就在两帧之间形成了一个无穷的循环，而且每当播放头遇到有代码的帧时，就会执行帧上的代码。

www.TopSage.com

需要注意的一件重要的事情是因为它有两帧，代码被执行，屏幕被更新和显示，然后播放头移动。这样屏幕会在每次重复执行代码时刷新，你就得到了动画。

例如，在舞台上有一个名为ball的影片剪辑。那么第1帧上的代码应是像下面这样：

```
ball.x ++;
```

第2帧的代码像下面这样：

```
gotoAndPlay(1);
```

实际上，在这个例子中第2帧可以什么也没有，因为在默认状态下时间线会自动循环回放到第1帧。 30

作为对这个例子的变换，你可以设置为3个帧，第1帧是一个初始化帧，你可以在它上面放置只运行一次的任何代码，而不是循环。第2帧将包括主要的动作，第3帧为：

```
gotoAndPlay(2);
```

第1帧可能设置了小球的初始位置，改变小球的颜色或大小，或做其他任何只发生一次的事情。时间线会在第2帧和第3帧之间无穷地循环，运行动画和更新屏幕。

在早期的Flash 5或更早的版本中，这是一种被广泛使用的设置，虽然现在看它有些不流行了，但它仍工作得很有效。即使你很快就会发现更灵活和更强大的设置方法，但是可以看到影片的不同阶段——初始化、动作和循环——这些概念在此后学习每一个动画设置过程中都会继续使用，这是非常有教育意义的。

2.5.3 剪辑事件

令人感到欣慰的是剪辑事件已完全从AS 3中移除了，但是仍执得作为一个脚注提及。回到Flash 5的那段时期，剪辑事件是帧循环最主要的可供替代的选择。

剪辑事件（clip event）是指放置的代码，并不是放在时间线上，而是直接放在影片剪辑自身上。它的操作方法是选择舞台上的影片剪辑然后打开Actions面板。当打开面板后在Actions面板上输入你的代码，这些代码就被直接分配给了那个影片剪辑。

我说这些代码被加在了影片剪辑“上”（on）。剪辑上的任何和所有的代码必须在剪辑事件代码块中显示，它看上去像这样：

```
onClipEvent(eventName){
   // 代码写在这里。
}
```

31

除了onClipEvent(eventName)之外，还有on(eventName)，on事件必须作用于鼠标和键盘动作，如按下（press）和松开（release）。

事件名称是指许多可能在Flash影片中发生的事件中的一种。一个事件只不过是影片中发生的某件事情。事件可以分为两大类别：系统事情和用户事件。系统事件（system event）是指当某件事情发生在计算机内、Flash或你的影片中，如数据载入、内容载入或一个新的帧开始播放。用户事件（user event）是指用户已经做的一些事情，通常是指键盘或鼠标活动。

对于剪辑事件来讲最常用的是load和enterFrame。load事件发生在一个影片剪辑实例第1次出现在舞台上时，并且对任何实例都只发生一次。你只需要将你的代码放在两个大括号中即可：

www.TopSage.com

```
onClipEvent(load){
   //初始化代码。
}
```

enterFrame事件发生在Flash每次准备开始渲染下一帧时，即便是只有一帧的影片。例如，如果帧频设为每秒钟20帧，enterFrame将会大约每隔50毫秒运行一次。这个enterFrame剪辑事件就是你经常应用“规则”或动作代码的位置，因为你有两个需求：重复代码和每次执行后的重新显示。它看上去如下：

```
onClipEvent(enterFrame){
    //动作代码。
}
```

这样，你可以将一个影片剪辑放置在时间线上，然后将下面的代码加在“上面”（注意这里是早期的AS 1语法）：

```
onClipEvent(load){
   this._x = 100;
   this._y = 100;
}
onClipEvent(enterFrame){
   this._x += 5;
}
```

另外，本书中的示例并没有使用这种设置方式（因为它不再是AS 3语言中的一部分），但是有3个要点需要注意：初始化、重复动作和屏幕刷新，这将是你在任何系统中制作任何动画的一部分。

2.5.4　事件和事件处理器

Flash MX在ActionScript上引入了许多重要的改变——这些改变和改进主要是将Flash带入丰富的互联网应用开发的前沿。其中一个最大的改变是新的事件架构，它使得对非常复杂的行为的编程比早期版本容易得多。

我已经介绍了一些关于事件的内容，它发生在系统有某件事情发生或用户通过键盘和鼠标做某件事情时。在Flash MX之前，只能通过影片剪辑或按钮上的onClipEvent（*eventName*）或on（*eventName*）代码块来捕获事件。这意味着你不得不在创作时在舞台上放置一个影片剪辑并在它上面编写代码。尽管各种工作区已被发掘，但没有一种是最理想的。MX的事件架构不能说是最好的，但是相比之前的版本它有一个很大的提高并且允许从程序中的任何位置访问事件、在任何时候停止任何指定的事件或动态地改变一个事件的行为。你可以想象它提供的这种功能是多么的强大和灵活。

随着Flash MX 2004的到来，在事件架构中增加了组件框架来提高事件机能。然而，这种提高只是在原有的事件架构基础上通过增加AS 2类来实现的。这些类提供了许多附加的事件操作，但是对于某些情况下AS 2类尝试提供的机能在事件架构的底层仍显得功能不足。

在AS 3中，事件架构已经从最底层重新写过。它很快、很强大，并且完全与Flash播放器集成。

www.TopSage.com

要理解事件，你需理解另外两个概念：侦听器和处理器。侦听器和处理器是真正的一对好名字，因为那就是它们所做的工作。一个侦听器（listener）是一个对象，它负责侦听事件。一个处理器（handler）是一个函数，当事件发生时它负责处理事件。在整个ActionScript历史中侦听器和处理器的执行有很大的不同，它们甚至在AS 2内的执行也会有不同。为了避免混淆，我将讲解AS 3，在AS 3整个系统中它会更纯粹、更漂亮和协调。

2.5.5 侦听器和处理器

像刚才所说的，一个侦听器是一个负责侦听事件的对象。可以通过调用名为addEventListener的函数来指定你的类作为一个侦听指定事件的侦听器。你需要传递想要侦听的事件名称和类中将作为处理器的函数名称。下面是一个例子：

```
addEventListener("enterFrame", onEnterFrame);
```

当加入事件侦听器时还可以使用其他的附加参数，但我们在本书中并没有这样做，在大多数的事件应用中上面的语法就足够了。

注意事件的名称"enterFrame"是一个字符串，这是我们都知道的，还会带点轻视，它是一个*神奇的字符串*。为什么神奇？因为如果你意外地把它错写成（例如）"entorFrame"，编译器会没有任何提示神奇地接受它，即便是没有这样的事件名称，然后你将花上几个小时的时间来搞清楚为什么你的事件处理器没有被调用。

但是AS 3已经有了对此的处理方法。可以使用Event类的属性，而不是去使用像"enterFrame"这样的神奇字符串。例如，在上面的例子中，可以按下面的方式书写：

```
addEventListener(Event.ENTER_FRAME, onEnterFrame);
```

如果你去查看Event.ENTER_FRAME的属性值，会发现它其实就是一个简单的字符串"enterFrame"。现在，你可能会想这个属性同样会非常容易错写成Event.ENTOR_FRAME。但是这里的改进是：如果你写错了，程序将拒绝编译并会马上告诉你在Event类中没有这种属性。它不但会精确地指出你错在第几行，而且也会精确到第一次遇到问题的那一行的具体字符上。它可以说好得不能再好了，除非有一天编译器可以纠正你的错误并且能为你写代码。

除了Event类的事件类型外，在其他类中还有相同方式的其他的事件类型设置，如MouseEvent.MOUSE_DOWN,KeyboardEvent.KEY_DOWN,TimerEvent.TIMER，如此等等。而他们都代表简单的字符串，如"mouseDown"、"keyDown"和"timer"，最好的方法是忘记这些字符串而只使用属性。

下一个需要知道的重点是：有些时候，如前面的例子，可以在一个类中直接调用addEventListener函数。在这种情况下，你实际上是告诉这个类侦听它自己的enterFrame事件。但是在有些案例中，你想侦听由其他对象产生的事件。例如，如果你有一个作为按钮使用的名为mySpriteButton的精灵。当用户点击这个精灵时，它将产生一个mouseDown事件。要侦听来自这个精灵的mouseDown事件，需要调用那个精灵的addEventListener方法，如下所示：

```
mySpriteButton.addEventListener(MouseEvent.MOUSE_DOWN, onSpritePress);
```

33

最后需要注意的事情是：不像之前版本的ActionScript，必须要用指定名字的事件处理器函数，

www.TopSage.com

如onEnterFrame，在AS 3中当你加入侦听器时，你可以为你的事件处理器命名任意的名称。在enterFrame的示例中，我使用onEnterFrame作为一个处理器，那只是因为它是一个合适的名字并且我已经习惯使用它。在AS 3中，onEnterFrame已不再有它本来的含义。我可以很随意地将enterFrame处理器命名为move、run或doSomethingColl。然而按照惯例，事件处理器通常是以on开头，后面跟随着事件的描述和可能来自于哪儿，如onStartButtonClick、onConfigXMLLoad或onRocketCrash。还有一些人喜欢在事件名称后面加上Handler后缀来作为事件处理器名称——例如，enterFrameHandler。这只是一个偏好的问题，最重要的事情是保持一致性，你只要能做到一看到它就知道它是事件处理器即可。

现在，我一直在说侦听器侦听事件，但是可能有一个更准确的描述，即事件通知侦听器。在内部，产生事件的对象将已把自己作为一个侦听器加入的每个对象保存为一个列表。如果一个对象能够产生不同类型的事件，如mouseDown、mouseUp和mouseMove，它将会为它能产生的每一种事件类型保存一个侦听器列表。无论什么时候事件发生时，对象就会浏览相应的列表并且让列表中的每一个对象知道什么事件发生了。

另一种描述事件的方法是变成侦听器的对象正在提交一个指定的事件。而且产生事件的对象正在向所有的提交者广播事件。

另外，如果不再想让一个对象侦听事件，可以告诉它停止侦听或停止提交，通过调用removeEventListener方法来完成：

```
removeEventListener(Event.ENTER_FRAME, onEnterFrame);
```

这就是在告诉对象从自身的侦听指定事件的侦听器列表中移除侦听器，因此它将不会再接收到其他任何的提醒。

让我们继续来看一些具体的应用。下面的代码创建一个新的精灵，将它放在舞台上，在它内部绘制一些图形，然后加入一个事件侦听器。以之前给出的基本的程序框架开始，加入代码如下：

```
package {
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class EventDemo extends Sprite {
        private var eventSprite:Sprite;
        public function EventDemo() {
            init();
        }
        private function init():void {
            eventSprite = new Sprite();
            addChild(eventSprite);
            eventSprite.graphics.beginFill(0xff0000);
            eventSprite.graphics.drawCircle(0, 0, 100);
            eventSprite.graphics.endFill();
            eventSprite.x = stage.stageWidth / 2;
            eventSprite.y = stage.stageHeight / 2;
            eventSprite.addEventListener(MouseEvent.MOUSE_DOWN,
                                         onMouseDown);
            eventSprite.addEventListener(MouseEvent.MOUSE_UP,
```


www.TopSage.com

```
                                        onMouseUp);
        }
        private function onMouseDown(event:MouseEvent):void {
            trace("mouse down");
        }
        private function onMouseUp(event:MouseEvent):void {
            trace("mouse up");
        }
    }
}
```

怎么理解呢？init函数最主要的作用是创建一个精灵，然后在里面绘制一个圆形，并将它对齐在舞台的中心。最后两行很重要。这个类将它自己加入作为侦听两个事件MOUSE_DOWN和MOUSE_UP侦听器。注意它们是MouseEvent类的属性，MouseEvent类必须要先导入。作为处理器，你传入onMouseDown和onMouseUp函数，他们刚好在其后面定义。处理器通常接收传过来的一个包含事件信息的对象。它至少包括了关于哪个对象触发了事件的信息。在本例中的鼠标事件，它包括了事件发生时鼠标的位置、哪个鼠标按键被按下（如果有的话）等信息。对于键盘事件，它将包括事件发生时哪些键被按下和其他按键（如Ctrl、Alt和Shift键）的状态。可以参考帮助文件来查看各种事件类型所包括的信息。

将上面的例子保存为EventDemo.as文件并使用你选择的编译环境来编译它，如前面的描述。当运行SWF时，将会看到每当在新的精灵上按下或放开鼠标时，它就会在输出面板上输出相应的信息。如果你使用的是Flex Builder 2，不要忘记是调试——而不是运行——你的程序；如果你使用的是免费的SDK，使用正确的方法针对你的解决方案进行跟踪。确保你可以让它正常工作。这是一个非常简单的例子，但是它可以很好地检测你的开发环境是否配置正确。

如果你刚刚接触ActionScript，而且已让它正常工作并且真正理解了它，那么恭喜你，你已经从“初学者”上升到“中级”状态了。

好，现在你对处理器已经有比较多的了解了，你可以更好地理解侦听器。前面我说过它是一个对象，产生事件广播事件或通知它的事件侦听器。它是怎样做的呢？它做的所有事情其实就是调用具有正确的处理器名称对象的函数。在前面的例子中，EventDemo类将它自己加入作为一个侦听器，来侦听精灵的两个事件。在内部，精灵为每一个事件保存了一个列表。那么它有一个mouseDown事件列表，还有一个mouseUp事件列表。这些可能只是简单的数组。每一个列表中包含一个对主影片的引用，其中主影片是EventDemo类的一个实例。

当用户在精灵上按下鼠标按钮时，精灵就会作出反应，“Whoa! Mouse down! Must notify listeners!（哇，鼠标按下了！必须通知侦听器!）”（或者类似的语句）。然后它就会浏览mouseDown列表并查看有什么在那里。精灵发现有一个主影片的引用和一指定作为处理器的函数的引用在那里。在这个例子中那个函数名为onMouseDown。然后它就简单地调用侦听器的函数。如果有其他对象注册成为mouseDown事件的侦听器，他们也会在列表中，它们已定义的处理器也将被调用。

同样的事情也会发生在放开鼠标的时候，只是它们在mouseUp列表中。

在AS 3中需要注意的一个重要技术就是很多作用域问题得到了全部的解决。这个问题有些复杂，但是理解它的一个简单的方法是当它被调用时处理器函数认为它在哪儿。尽管AS 2事件架构

www.TopSage.com

表面上看与AS 3非常相似，但是在AS 2中，作为事件处理器被调用的类方法执行时好像是调用对象的方法，而不属于定义它的类。这听上去真够复杂的，但事实确实如此，并且专门创建了一个类——Delegate类——来解决这个问题。如果这一段你一点也不明白，那也别沮丧，谢天谢地，在AS 3中你不需要再去想这些问题了。

以上这些就是事件和事件处理器的基础知识。我会随着我们的进度继续介绍很多其他的事件，但是现在，让我们回到动画上。

2.5.6　动画事件

那么，我们谈论的所有关于事件的问题把我们带到哪儿了呢？我们正在寻找一种能够用代码将物体制成动画和允许屏幕在每次迭代后刷新的方法。在前面，你已看过一个使用enterFrame剪辑事件来制作动画的示例。让我们以这项技术来作为开始。

在AS 3中，你可以像下面这样给enterFrame事件加入一个侦听器：

```
addEventListener(Event.ENTER_FRAME, onEnterFrame);
```

记得你需要先导入Event类并且创建一个真实的名为onEnterFrame的方法。

有一点有时会让人们糊涂，那就是即使影片只有一帧，你也能得到一个enterFrame事件。（并且如果你正在使用Flex Builder 2或SDK，你是看不到任何帧的，但是肯定的是，确实有一帧存在在那里！）播放头并不是真正地进入了一个新的帧，它只是停在第1帧上。意识到enterFrame事件并不是移动播放头到一个新的帧。而更多的是围绕着——事件告诉Flash什么时间移动播放头。可以把enterFrame想象为一个定时器，虽然不是很精确。Flash会根据帧频的设定来确定要每隔多长时间进入一个新的帧。然后它就会在那段时间间隔内重复地执行事件。如果有其他的帧可以去，它就会播放到那里。但是不管怎样，你都可以计算出事件执行的次数。

> 如果你对Flash的某些内部工作过程非常感兴趣，你可以查看Tinic Uro的博客，地址是www.kaourantin.net。Tinic是一位在Adobe工作的Flash播放器开发工程师，经常对Flash场景背后是怎样工作的发表一些深层次的解释。

在前面的示例中已经有了初始化代码，或者在一个初始化帧中或者在一个onClipEvent(load)代码块中。你可能已经猜到了，这些代码都将在init方法中发生。该方法只能能过构造函数调用一次。你需要习惯init方法在影片中绝不会再次被调用。如果你需要重新初始化任何其他变量或调用任何其他函数，那么就把它们放在其他的函数中并且在需要的时候调用它们。

下面是我们第1个基于AS 3的动画示例！

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class FirstAnimation extends Sprite {
        private var ball:Sprite;

        public function FirstAnimation() {
```

www.TopSage.com

```
            init();
        }
        private function init():void {
            ball = new Sprite();
            addChild(ball);
            ball.graphics.beginFill(0xff0000);
            ball.graphics.drawCircle(0, 0, 40);
            ball.graphics.endFill();
            ball.x = 20;
            ball.y = stage.stageHeight / 2;
            ball.addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void {
            ball.x++;
        }
    }
}
```

init函数创建一个名为ball的精灵并且设置了事件侦听器。onEnterFrame函数负责动画制作，移动小球，之后屏幕会被刷新。它并不是很复杂，但是这是本书中所有其他内容的基础，所以要很好地理解它。

好，关于如何用ActionScript组织动画的基础已经讲了不少。我已经解释了如何让规则反复运行和如何在每次应用规则后更新屏幕以产生运动错觉。但是你能用它来移动什么呢？目前为止，你已经可以移动一个精灵，但是目前的所有代码只是简单应用在一个影片剪辑上。下一步，我们将研究如何创建精灵和影片剪辑并使用显示列表来显示它们。

2.6 显示列表

在AS 3之前，你可以在一个Flash影片中创建大量不同类型的可视对象，包括影片剪辑、图形元件、按钮、文本域、位图、组件和基本的图形。这些对象实际上并没有被组织成一定的层次结构，并且它们都有各自不同的创建、销毁和操作的方法。举例来说，影片剪辑可以被贴加(attach)、复制、放在IDE的舞台上或从头开始创建。文本域可以从头开始创建或在IDE中创建。 37
当你接触位图、视频、组件和其他的内容时，好像每一种类型是在不同的平台上创建，然后被堆放在Flash中，并强行让它们在一起工作。

在AS 3中，这些对象之间的区别已经讨论过了。你所看到的舞台上的任何类型的对象都是由扩展DisplayObject类的类所创建的。换句话说，所有这些对象都是一个大家庭的一部分，而且它们都有着相同的基本行为，根据这点你可以知道怎样创建它们，把它们放在舞台上，销毁它们，并在舞台上操作它们。

现在，创建一个精灵、一个影片剪辑或一个文本域的代码非常相似。实际上，所有的显示对象现在的创建方法与其他任何类型对象的创建方法相同，都使用new操作符。为了验证这一点，下面给出如何创建这3个对象：

```
var myTextfield:TextField = new TextField();
var myMovieClip:MovieClip = new MovieClip();
var mySprite:Sprite = new Sprite();
```

www.TopSage.com

大家网 TopSage.com

这种方法是不是更简单呢？

正如在前面的示例中所看到的，如果已经创建了一个影片剪辑或精灵，现在可在那些对象里绘制内容了，如下：

```
mySprite.graphics.beginFill(0xff0000);
mySprite.graphics.drawCircle(0, 0, 40);
mySprite.graphics.endFill();
```

但是如果只是像上面那样做了，可能看不到很多内容。这就是我们要讨论显示列表的地方。“显示列表”术语是新增的，但是如果你使用Flash已经有一段时间了，那么它不应当是一个完全陌生的概念。把影片中的所有可视对象想象为一棵树。树的根部是舞台，它默认状态就是可视的。在舞台上，可能还有几个影片剪辑或其他类型的可视对象（文本域、图形等）。当把它们加到舞台上时，它们也会变成可视的。在那些影片剪辑内部可能还会有其他的影片剪辑或其他的可视对象，并且在其他的内部也可能会有其他的。这基本上就是你的显示列表。

在AS 2与AS 3中显示列表的工作方式的最大的不同就是：在AS 2中，当通过贴加它或使用createEmptyMovieClip创建一个影片剪辑时，必须在创建时指明它将放在哪里。在那个地方，影片剪辑会被创建，并被放置在一个指定位置的列表中，那就是它所要留的地方。如果不移除影片剪辑，就没有办法在位置列表中改变它的位置或者在不完全销毁的情况下将它从列表中去掉。

在AS 3中，当创建一个新的影片剪辑、精灵或其他的显示对象时，它不会自动加入到显示列表中。注意在刚刚给出的精灵示例中，创建精灵时没有提到父级（parent）或深度（depth）。它们不在显示列表中也可以存在和被操作。

如果从字面意义上理解“舞台”，你可以想象这些对象就是幕后的演员，收起翅膀，悬在舞台的上空，或者甚至趴在地底下。你不能看见他们，但是他们都在那里，忙着准备他们的入场，
38 到那时他们将跳上舞台并且吸引人们的注意。

到现在为止，我们已经使用了树的比喻和一个剧院的比喻——再使用一个比喻怎么样？就家庭而言，我们也可以将显示列表想象为一系列的父亲/孩子/兄弟关系。当加入某个东西到显示列表中时，它附加到的那个对象就是父级，新的对象就是父级的孩子。这样，当你增加一个对象到列表中时使用addChild方法来完成就易于理解了。

文档类代表一个树的根（或所有其他可视对象的伟大的祖先，如果你愿意）。它是自动可视的，然后你开始为它加入孩子。因此，如你看到的之前的示例，要使精灵或任何其他显示对象可视，调用addChild方法，把新创建的精灵传递给它：

```
var mySprite:Sprite = new Sprite();
mySprite.graphics.beginFill(0xff0000);
mySprite.graphics.drawCircle(0, 0, 40);
mySprite.graphics.endFill();
addChild(mySprite);
```

如果你试一试，则将这些代码放在之前给出的类架构中的init函数中。注意，最初要把它放置在原点0,0上，可以通过调整x和y属性来改变它的位置。

还要注意的是这里没有提到深度，当我们在AS 2中创建影片剪辑或贴加它们时也是如此。虽

www.TopSage.com

然深度管理大部分情况下是自动的，但仍有方法在指定的深度增加孩子或改变它的深度，这些在我们需要的时候会讲到。

要从显示列表中移除一个对象，调用removeChild方法，传递一个真实的子对象的引用。需要理解的第1个关键的地方是移除一个子对象并没有销毁它！它会保持与移除之前一样的状态而继续存在。并且在将它加回显示列表时仍保持原有的状态。换句话说，如果已经在子对象中绘制了一些图形或载入了一些外部内容，当要将它放回显示列表中时，就不需要再重新绘制或重新载入那些内容，而在AS 2中必须这样做。

第2个关键的地方是当将子对象加回到列表中时，可以自由地将它放在任何地方。这被称为重设父级（reparenting）并且是令人兴奋的（至少对我来说是这样）。可以从一个影片剪辑移除一个对象并将它贴加到另一个剪辑中，当移除它时它会保持原有的状态，这在之前的版本中是不可能的。实际上，你甚至不需要移除它。因为一个子对象只能有一个父级，将它加入另一个父级会自动从原来的父级移除。

下面的类演示了重设父级：

```
package {
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class Reparenting extends Sprite {
        private var parent1:Sprite;
        private var parent2:Sprite;
        private var ball:Sprite;
        public function Reparenting() {
            init();
        }
        private function init():void {
            parent1 = new Sprite();
            addChild(parent1);
            parent1.graphics.lineStyle(1, 0);
            parent1.graphics.drawRect(-50, -50, 100, 100);
            parent1.x = 60;
            parent1.y = 60;

            parent2 = new Sprite();
            addChild(parent2);
            parent2.graphics.lineStyle(1, 0);
            parent2.graphics.drawRect(-50, -50, 100, 100);
            parent2.x = 170;
            parent2.y = 60;

            ball = new Sprite();
            parent1.addChild(ball);
            ball.graphics.beginFill(0xff0000);
            ball.graphics.drawCircle(0, 0, 40);
            ball.graphics.endFill();
            ball.addEventListener(MouseEvent.CLICK, onBallClick);
        }
```


www.TopSage.com

```
        public function onBallClick(event:MouseEvent):void
        {
            parent2.addChild(ball);
        }
    }
}
```

这个类创建了3个精灵：parent1、parent2和ball。父级精灵直接加到显示列表中，并且它们里面都绘制了正方形。ball精灵被加入到parent1中，因此它现在在显示列表中并是可视的。

当小球被点击时，它会加入到parent2中。注意这里没有改变它的x,y位置的代码。它会发生移动是因为它在位置不同的精灵中。还要注意的是动态绘制的圆形会始终保持，即使精灵曾被移除和加回。

子类中的显示对象

在前面，你已经学习了在创建文档类时Sprite或MovieClip的子类。这些类的子类还有许多其他用途。

首先，你可能在想，既然attachMovieClip已经不在AS 3中了，那如何在影片运行时从Flash CS3 IDE库中获取一个影片剪辑元件并放入影片中呢？答案是使用一个扩展精灵或影片剪辑的类。虽然我努力不把太多重点放在创建SWF影片时使用的任何一种工具上，但我还是要绕行到
40 IDE中来解释这个概念。

这个概念通过下面的例子可以得到最好的说明：

(1) 创建一个新的FLA，然后在舞台上绘制一些内容。

(2) 选中内容并按下F8以将其转换为一个元件。

(3) 在Convert to Symbol对话框中，输入元件的名称，并选择元件类型为Movie Clip。

(4) 选择Export for ActionScript。

在这一点上，在之前版本的Flash中，你将获得一个自动生成的链接标识符，你可以选择性地输入一个类。

在Flash CS3中，Identifier文本域被禁止了，Class文本域强制要求输入并且填充了一个默认值。还有一个新的文本域Base class，它预置的是flash.display.MovieClip。可以根据需要将它改为扩展影片剪辑或精灵的其他类。

输入你想要的任何一个类的名称，不用担心你还没有创建这个类。然后点击OK。这里会比较有趣。因为Flash不能找到你指定名称的类，它将在编译时自动生成一个类。这并不意味着Flash真地会创建一个你可以找到的ActionScirpt类文件，但是它会在你的SWF中创建一个代表基于精灵或影片剪辑的新类的字节代码（bytecode）。除了扩展那个基类，这个类不能做任何事情，但是它会链接到你库中的元件。

现在，假设你输入的类名为Ball。在你的文档类中或时间线上，你可以书写如下代码：

```
var ball:Ball = new Ball();
addChild(ball);
```

www.TopSage.com

这会使库中的元件在舞台上被创建，就像以前AS 2中的attachMovie方法。

如果你只是想将库中的影片剪辑内容“贴加”到影片中，这是一个非常好的方法。但是不是简单地接受这个自动生成的类，你可以给它一个自己创建的真实的类的名字和路径，给元件赋予一些额外的功能。现在让我们跳出Flash IDE，回到下一个示例的类中来。

实际上，我们将重新来看前面刚刚给出的重设父亲的示例，那里的一些复制可以被整理为另一个类。在那个示例中，我们创建了一个名为parent1的精灵并用下面的代码在它的内部绘制了一个盒子：

```
parent1.graphics.lineStyle(1, 0);
parent1.graphics.drawRect(-50, -50, 100, 100);
```

我们接着创建了另外一个精灵parent2，在它的内部也用同样的代码绘制了同样的盒子。尽管这是一个很小的例子，但是它可以清晰地显示出精灵的子类是多么有用。首先，我们将创建一个扩展精灵的名为ParentBox的类。在这个类中，我们会编写绘制盒子的代码。 41

```
package {
    import flash.display.Sprite;

    public class ParentBox extends Sprite {
        public function ParentBox()
        {
            init();
        }
        private function init():void
        {
            graphics.lineStyle(1, 0);
            graphics.drawRect(-50, -50, 100, 100);
        }
    }
}
```

然后我们将改变类来创建两个ParentBox，而不是两个Sprite。

```
package {
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class Reparenting2 extends Sprite {
        private var parent1:ParentBox;
        private var parent2:ParentBox;
        private var ball:Sprite;

        public function Reparenting2() {
            init();
        }
        private function init():void {
            parent1 = new ParentBox();
            addChild(parent1);
            parent1.x = 60;
            parent1.y = 60;

            parent2 = new ParentBox();
```

www.TopSage.com

```
            addChild(parent2);
            parent2.x = 170;
            parent2.y = 60;

            ball = new Sprite();
            parent1.addChild(ball);
            ball.graphics.beginFill(0xff0000);
            ball.graphics.drawCircle(0, 0, 40);
            ball.graphics.endFill();
            ball.addEventListener(MouseEvent.CLICK, onBallClick);
        }
        public function onBallClick(event:MouseEvent):void
        {
            parent2.addChild(ball);
        }
    }
}
```

42

作为ParentBox的实例，他们仍是Sprite，因此可以在它们中加入子显示对象。它们也可以使用新的init方法，它自动绘制它们的图形。这是一个很小的例子，但是希望你能够理解它，因为当你继续往后看这本书时，你会看到许多更为复杂的示例。

你也许想要针对此例加入一个Ball类来绘制小球的图形。这样做不能减少任何复制，但是当你的类变得更加复杂时，将某些可分离的功能定义成不同的类是一个好主意，而不应把所有东西都放到一个类中。这种方式也是最值得推荐的。实际上，在接下来本书的内容中，你很快就会创建一个你需要用到的像Ball这样的类了。

2.7　用户交互

最后，我们来看*用户交互*（user interaction），可能这就是你要阅读本书的原因之一。毕竟，如果没有使用交互或影片中的一些动态输入，你也许要使用的就是补间了。

实际上，我们在前面的章节上并不是一点也没有讨论这部分的内容。用户交互是基于用户事件的，而且这些一般可归纳为鼠标事件和键盘事件。让我们快速地过一遍各种用户事件和它们的处理器。

2.7.1　鼠标事件

鼠标事件在AS 3中已经发生了巨大的变化。在AS 2中，影片剪辑自动侦听所有鼠标事件。现在任何对象都明确地将自身加入为一个侦听器。另外，对于要在AS 3中响应的鼠标事件，需要鼠标的光标停放在显示对象的可视内容上。这一点不同于AS 2，AS 2中的mouseUp、mouseDown和mouseMove对于任何影片剪辑都会响应，而不管鼠标的光标在哪儿。AS 3中的mouseUp和mouseDown事件现在等同于AS 2中的onPress和onRelease。

鼠标事件的名称用字符串来定义，但是就像前面所说的，最好是使用MouseEvent类的属性以避免出错。下面是MouseEvent类中可用的鼠标事件属性：

- CLICK

www.TopSage.com

TopSage.com

- DOUBLE_CLICK
- MOUSE_DOWN
- MOUSE_MOVE
- MOUSE_OUT
- MOUSE_OVER
- MOUSE_UP
- MOUSE_WHEEL
- ROLL_OUT
- ROLL_OVER

这些属性名称自身就是很好的解释。如果要感觉一下它们，可以创建和编译下面的类，当每一个鼠标事件在它创建的精灵上发生时它都会跟踪这个鼠标事件。

```
package {
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class MouseEvents extends Sprite {
        public function MouseEvents() {
            init();
        }
        private function init():void
        {
            var sprite:Sprite = new Sprite();
            addChild(sprite);
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawCircle(0, 0, 50);
            sprite.graphics.endFill();
            sprite.x = stage.stageWidth / 2;
            sprite.y = stage.stageHeight / 2;

            sprite.addEventListener(MouseEvent.CLICK, onMouseEvent);
            sprite.addEventListener(MouseEvent.DOUBLE_CLICK,
                                    onMouseEvent);
            sprite.addEventListener(MouseEvent.MOUSE_DOWN,
                                    onMouseEvent);
            sprite.addEventListener(MouseEvent.MOUSE_MOVE,
                                    onMouseEvent);
            sprite.addEventListener(MouseEvent.MOUSE_OUT,
                                    onMouseEvent);
            sprite.addEventListener(MouseEvent.MOUSE_OVER,
                                    onMouseEvent);
            sprite.addEventListener(MouseEvent.MOUSE_UP,
                                    onMouseEvent);
            sprite.addEventListener(MouseEvent.MOUSE_WHEEL,
                                    onMouseEvent);
            sprite.addEventListener(MouseEvent.ROLL_OUT,
                                    onMouseEvent);
            sprite.addEventListener(MouseEvent.ROLL_OVER,
                                    onMouseEvent);
```

www.TopSage.com

```
        }
44      public function onMouseEvent(event:MouseEvent):void
        {
            trace(event.type);
        }
    }
}
```

注意，这个类对每一种事类型都使用了同一个处理器（handler），而且只是简单地输出被发送出来的事件类型。

2.7.2　鼠标位置

除了鼠标事件，还有两个非常重要的属性可以从文档类或任何显示对象中获得，它们用来确定鼠标指针mouseX和mouseY的当前位置。注意它们是影片剪辑的属性，而且返回值是相对于剪辑注册点的鼠标位置。例如，如果你有一个名为sprite的精灵在舞台上的位置为100,100，鼠标指针在150,250，你会得到下面的结果：

- mouseX is 150.
- mouseY is 250.
- sprite.mouseX is 50.
- sprite.mouseY is 150.

注意影片剪辑是如何得出相对于它自身位置的鼠标位置的。

2.7.3　键盘事件

键盘事件是另一个在AS 3中被整理的部分。例如，在AS 2中，影片剪辑是键盘事件的自动侦听器，但是它们只在某种环境下才会得到那些事件。因此最好的办法是显式地加入一个影片剪辑作为侦听器，但是当影片接收多个事件而不只是单一的键盘事件时，这种方法会失效。接着，AS 2组件架构的一大部分致力于处理Flash播放器的底层架构中与键盘交互相关的问题，包括：tab管理、焦点管理、文本域中的回车键和Tab键等。

我们只能说现在的情况好多了。

键盘事件的名称就像鼠标事件一样被定义成字符串，但它们也是KeyboardEvent类的属性。这个类的属性只有两个：

- KEY_DOWN
- KEY_UP

你可以在一个指定的对象上侦听键盘事件，其方式就像我们前面讲到的侦听鼠标事件的一样。然而，为了做到这一点，你需要设置对象的焦点以致它可以捕获那些事件。你可以像下面这样书写来完成此操作：

```
45  stage.focus = sprite;
```

而在许多情况下，侦听键盘事件而忽视的它的焦点更合乎情理。要做到这一点，你可以直接

www.TopSage.com

在舞台上侦听键盘事件。下面是一个示例：

```
package {
    import flash.display.Sprite;
    import flash.events.KeyboardEvent;

    public class KeyboardEvents extends Sprite {
        public function KeyboardEvents() {
            init();
        }
        private function init():void {
            stage.addEventListener(KeyboardEvent.KEY_DOWN,
                                   onKeyboardEvent);
            stage.addEventListener(KeyboardEvent.KEY_UP,
                                   onKeyboardEvent);
        }
        public function onKeyboardEvent(event:KeyboardEvent):void {
            trace(event.type);
        }
    }
}
```

2.7.4 键盘代码

通常，你不但想知道一个键被按下或放开，还想知道它是哪个键。有两种方法可以读取键盘事件处理器中的这个信息。

在之前讨论事件的过程中，我说过一个事件处理器接收传过来的一个事件对象，这个事件对象中包含着刚刚发生的事件的数据信息。在一个键盘事件中，有两个属性与事件中的按键有关：charCode和keyCode。

charCode属性给出的是表示刚被按下的那个键的字符。例如，如果用户在键盘上按下了“a”键，则charCode将包含字符串“a”。如果用户同时也按下了Shift键，则charCode将包含的是“A”。

keyCode包含的是描述被按下的物理键的数字。如果一个用户按下“a”键，keyCode将包含数字65，而不管其他同时按下的键。如果用户先按下Shift键然后按“a”，则你将得到两个键盘事件：一个是Shift的（keyCode 16），另一个是“a”的（65）。

flash.ui.Keyboard类也有一些属性来代表一些非字母数字键，因此你不需要记住它们。例如，Keyboard.SHIFT等于数字16，因此你可检测keyCode是否等于Keyboard.SHIFT来发现Shift键是否被按下。这一章中的最后一个例子做了一些非常类似的工作，示范如下：

46

```
package {
    import flash.display.Sprite;
    import flash.events.KeyboardEvent;
    import flash.ui.Keyboard;

    public class KeyCodes extends Sprite {
        private var ball:Sprite;

        public function KeyCodes() {
```

www.TopSage.com

```
            init();
        }
        private function init():void {
            ball = new Sprite();
            addChild(ball);
            ball.graphics.beginFill(0xff0000);
            ball.graphics.drawCircle(0, 0, 40);
            ball.graphics.endFill();
            ball.x = stage.stageWidth / 2;
            ball.y = stage.stageHeight / 2;
            stage.addEventListener(KeyboardEvent.KEY_DOWN,
                                   onKeyboardEvent);
        }
        public function onKeyboardEvent(event:KeyboardEvent):void {
            switch(event.keyCode) {
                case Keyboard.UP :
                ball.y -= 10;
                break;

                case Keyboard.DOWN :
                ball.y += 10;
                break;

                case Keyboard.LEFT :
                ball.x -= 10;
                break;

                case Keyboard.RIGHT :
                ball.x += 10;
                break;

                default:
                break;
            }
        }
    }
}
```

关于键处理方面需要知道的一点是：当你在Flash创作环境中测试一个影片时，IDE会拦截一些键来控制IDE自身。在测试过程中，Tab键、所有的功能键和所有作为菜单项的快捷方式的键都不会被影片接收。当影片运行时，你可以通过在菜单中选择Control再选择其中的Disable Keyboard Shortcuts来解决此问题。这允许你测试影片时就像它真正工作在浏览器中一样。

2.8 小结

这一章覆盖了用ActionScript制作动画的所有基础。你现已经知道了帧循环、事件、侦听器、处理器和显示列表。我也介绍了类、对象和基本的用户交互。这是许多有用的资料！如果有些地方还有点模糊不用担心。当讲到具体的技术内容时，我会更加详细地讲解它们，并且你可以随时回到这里重温任何概念。至少，现在你已经熟悉了这些术语和概念，可以准备继续阅读了。

www.TopSage.com

第3章

动画中的三角学

本章内容

- 什么是三角学
- 角
- 三角函数
- 旋转
- 波
- 圆和椭圆
- 毕达哥拉斯定理
- 两点之间的距离
- 本章重点公式

我想把这一章放在书的前面，这样你就可以在动画技术中更多地使用三角学，从第5章开始有相应的示例。实际上，在下一章的渲染技术中就会涉及它。尽管如此，如果你已经熟悉了三角学的基础或非常渴望进入到动画的学习中，可以选择跳过它继续阅读。当遇到一些不理解的地方时再随时回来查阅。

许多人都回避数学和三角学，如经常会说，“我对数字不是很精通，只会计算2+2。”有趣的是使用三角学编程时，几乎不需要处理数字。它更多的是关于可视化形状和关系。大部分情况下处理的是包含位置、距离和角度的变量，根本不会看到真正的数字。它的主要问题在于记住各种关系。实际上，基础动画中所需要的90%的三角学可归结为两个函数：Math.sin和Math.cos。

当我写本书第1版时，曾经说过，除了高中代数和几何（大部分已经忘光了）之外，我没有受过任何正规的数学培训，本章中所原创出来的内容都是我从各种书、网站和其他在线资源中学习到的。这说明它并不难学，并且如果我能理解这些内容，你也能理解。从那之后，我已真正完成了大学里的代数和微积分预备课程，并且学习了比我原来所知道的更多的三角学知识。我非常高兴地说，本章中的资料已经非常好了。没什么需要修正的，如果有的话，就是对这个题目的更

www.TopSage.com

大家网 TopSage.com

深的理解，我认为我能把几个要点讲得更清晰。

3.1　什么是三角学

三角学基本上研究的是三角形及其边与角之间的关系。对于任何三角形，都有3条边和3个角（因此命名为三角形（tri-angle））。这些边与角恰好有着非常特殊的关系。例如，选取任何一个三角形并让其中一个角变大，那么与此角相对的边将会变长（假设其他两边保持不变）。同时，另外两个角将会变小。这些边和角到底改变了多少需要一些计算，但是这些比例已经被计算出并且形成了定理。

三角形的一种特殊类型是其中一个角等于90度，它被称为直角三角形，通过在该角的角落里标一个小方块来表示。在直角三角形中，各种关系变得更简单且很容易用一些基本公式来计算。这使得直角三角形成为一个非常有用的概念。本章后面内容所使用的三角形都是直角三角形。

3.2　角

因为三角学主要是关于角的，所以我们先来看这个主题。角（angle）是由两条相交线形成的形状，或者说是两条相交线之间的空间。空间越大，角就越大。实际上，两条相交线将形成4个角，如图3-1所示。

52

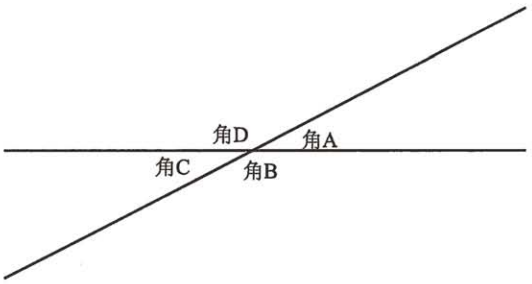


图3-1　两条线形成4个角

3.2.1　弧度和度

测量角的两个主要的系统是度（degree）和弧度（radian）。你可能非常熟悉度，毫无疑问你可以闭着眼睛画出一个45度或90度的角。圆的360度系统已经变成了我们文化习惯的一部分。“转180度”意思是说“去相反的方向”，即使谈论的不是物理方向，指的也是相反的视点。但是当谈到角时，计算机更喜欢使用弧度。因此，不管你喜不喜欢，都需要了解弧度。

1弧度近似等于57.295 8度。现在，你可能转动着眼珠在说，“噢，不符合逻辑吧？”但是它确实有一些逻辑在里面。一个完整的圆，即360度，计算出的结果是6.283 2弧度。仍然没有意义吗？好，记得pi—符号π吗？它约等于3.141 6，这意味着一个圆（6.283 2弧度）正好为2 pi。现在可能仍看不出太多的逻辑，但如果你使用的次数足够多，就会习惯将360度想为2 pi，180度为pi，90度为pi/2，等等。图3-2显示了一些常用的弧度的测量。

这个先讨论到这儿，但你要清楚，以后我们使用弧度。开始你可能会感叹一下，然后会习惯

它，甚至以后会用得很高兴。你将会遇到许多需要用弧度来表示度的情形。例如，精灵或影片剪辑的rotation属性恰好是使用度测量的，而且这个属性你会经常用到。

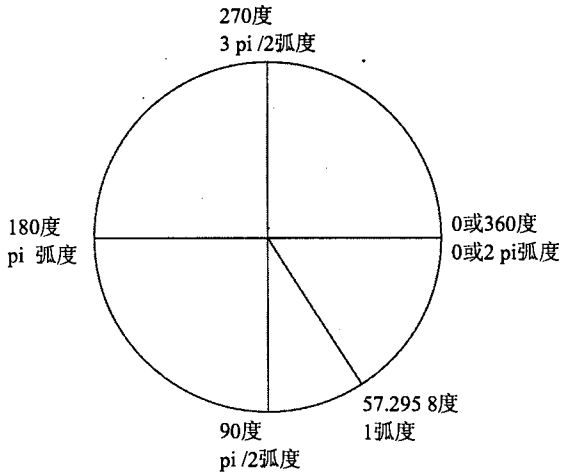


图3-2 弧度和度

例如，假设你有一辆车需要改变其运动方向，如果你使用三角学来计算运动，得到的角度应是用弧度表示的，但是要旋转这辆车，需要用度表示。或者倒过来，假设你想在某个方向上将物体移动到某个点。你可以检测它的rotation来得到那个角，但这是以度测量的，但是如果要用三角公式，需要使用弧度。

另一个应用度的地方就是在滤镜中。不管是在Flash IDE中创建滤镜还是使用ActionScrip编写一个DropShadowFilter类，如果想使用投影滤镜将阴影从对象位置延伸45度，需要用度来指定这个角，而不是弧度。

为什么在同一程序中使用两种不同的度量系统？这是Flash双重特性的一种表现。一方面，它是一个设计者工具。在Flash IDE中，你可以使用所有的绘图和转换工具绘制出漂亮的图形。我确信大多数设计师在你告诉他们输入一个弧度值来旋转其所创建的标志时，他们都会奇怪地看着你。另一方面，Flash也是一个开发者工具，就像大多数程序语言一样，ActionScript使用的是弧度。因此，不管你喜不喜欢，你将使用它们两个来工作，并且需要知道如何将度转换为弧度，反之亦然。下面是它们的转换公式：

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

在阅读本书时，将会碰到许多公式，到处都有，当需要牢牢记住它们时我会指出来。以下是第1处需要记住的。如果每次用到它时都回来翻看这些公式，那么你将永远无法编写代码。它们应当是在你想用的时候就能从键盘上打出来。我已经写好它的ActionScript公式，使用的是Math.PI而不是pi或其有趣的符号，因为那样你会反复键入它们。

从现在开始，你可以很容易地知道180度约为3.14…弧度。换句话说，半圆是pi弧度，这更有

www.TopSage.com

意义，因为一个整圆是2 pi。反过来，你可以知道1弧度约为57.29…度。

3.2.2 Flash 坐标系统

当讨论角时，更利于说明Flash中的空间是如何布局的。如果你在Flash坐标系统之前处理过任何坐标系统，在这里可能会有点头晕眼花，因为几乎每件事都颠倒了。

最常用的二维坐标系统使用*x*表示水平坐标，使用*y*表示垂直坐标。Flash也是这样做的。*x*是0且*y*是0的位置（0,0），通常位于中心，自中心向右是正*x*，向左是负*x*，向上是正*y*，向下是负*y*，如图3-3所示。

53
54

然而，Flash坐标系统是基于视频屏幕的坐标系统，在这个系统中0,0位于左上角，如图3-4所示。*x*值始终是从左向右增加的，而*y*坐标是相反的，向下是正值而向上是负值。我认为这个系统有它的历史根源，就是电子枪扫描屏幕以构建图像时采用的方式为：从左到右，从顶端到底端，但是其实这没什么。那是它的工作方式，而且短期内不会改变。

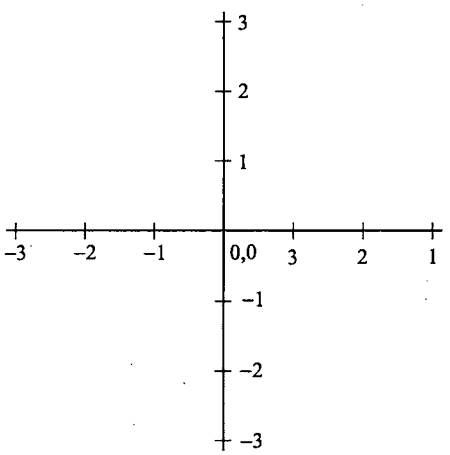


图3-3　标准坐标系统

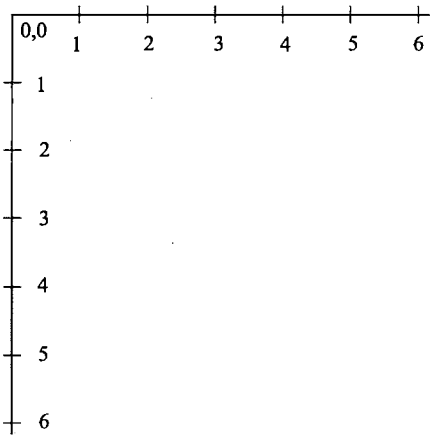


图3-4　Flash坐标系统

可以创建一个系统，在这个系统中使用标准坐标系统中的数字工作，并在渲染之前将它们转换为Flash坐标。总地说来，需要将*y*值取反，并将所有的内容移动半个屏幕使中心到左上角。然而，问题是所有那些计算都需要时间。虽然这些年已经快了很多，但Flash在这部分性能上与其他程序语言相比仍然差一些。因此就个人而言，我用脑力来减少计算量。学习使用反转的坐标并保留酷炫效果的计算方法。

55

稍等，还有呢！我们来讨论一下角的测量。在大多数系统中，角是按逆时针测量的，0度显示为一条沿*x*轴正方向伸展的线，如图3-5所示。

而Flash是相反的，如图3-6所示。它是顺时针旋转得到正的角，逆时针意味着负的角。

你可以制作一个系统在渲染之前反转角度，或者保持当前的这种计算并学会适应它。我建议使用后者，而且这是我对本书中所有代码采用的策略。

www.TopSage.com

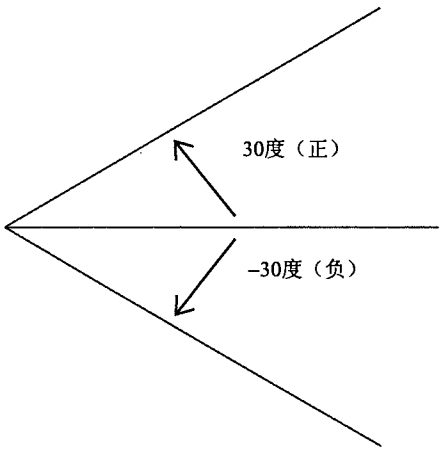


图3-5 一般的角测量

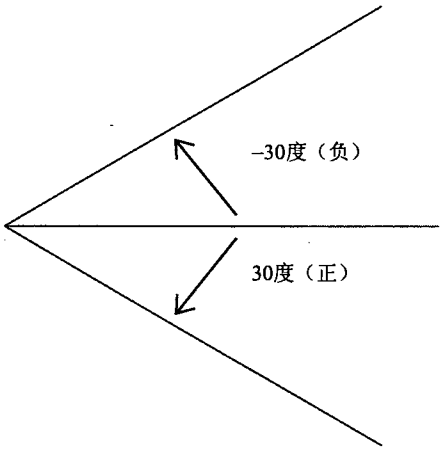


图3-6 Flash的角测量

3.2.3 三角形的边

单独讨论三角形的边没有什么可说的，但是有一些特殊的术语需要掌握。目前，我将谈论直角三角形，其中一个角为90度三角形。在这种情况下，那些边都有特殊的名称，如图3-7所示。形成90度夹角的两条边称为直角边，90度角相对的边称为斜边。斜边总是最长的边。

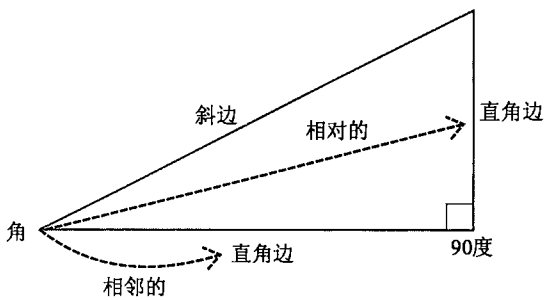


图3-7 直角三角形的各部分

当谈到与一个角相对的（opposite）边（对边）时，指的是没有接触到这个角的边。当谈到一条相邻的（adjacent）边（邻边）时，指的是接触到角的其中一条边。多数情况下，示例将处理两个非90度角中的一个。这种情况下，当谈到相邻时，指的是相邻的直角边，而不是斜边。

三角形中边的尺寸与角的大小之间的关系很有趣。这些关系对于动画制作是非常有用的。因此我们将在下一节中探讨。

3.3 三角函数

ActionScript有用来计算各种三角关系的三角函数：正弦（sine）、余弦（cosine）、正切（tangent）、

www.TopSage.com

反正弦（arcsine）、反余弦（arccosine）和反正切（arctangent）。这里将定义这些函数和访问它们的ActionScript函数。然后介绍这些函数的一些实际应用。

3.3.1 正弦

56
~
57

这是第1个现实点的三角学。一个角的正弦是这个角的对边与斜边之比。（当谈到正弦时，总是指一个角的正弦。）在ActionScript中，可以使用Math.sin（angle）函数。

图3-8显示了一个30度角的正弦。对边的测量值为1，斜边的测量值为2，因此比例为1比2，或者使用数学方式表示为1/2或0.5。因此，可以说30度角的正弦是0.5。在Flash中可以像下面这样测试它：

```
trace(Math.sin(30));
```

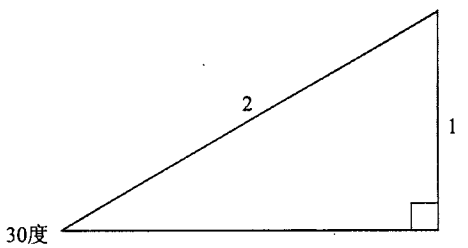
等一下，得出的结果是–0.988 031 624 092 862，它根本不是一个接近的值。错在哪里？我们忘记将度转换为弧度了。我保证你偶尔会犯这种错误（我也是），因此要习惯于留心它。下面是使用了转换的正确代码：

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

成功了！得出的结果是0.5。

实际上，你可能得到的是类似0.499 9…这样的数值，它在程序中是没有错误的，但是二进制计算机有时会终止不断重复的浮点值。然而它已经很接近了，可认定是0.5。

现在，对于一个角为30度边长为1和2的抽象三角形，这非常好。但是让我们把它移到现实世界中来，或者至少是Flash坐标系统中。在Flash中，垂直方向上向下是正值，且正的角测量是顺时针的。因此在这种情况下，那个角和它的对边都应是负值，如图3-9所示。



58

图3-8　一个角的正弦是它的对边 / 斜边

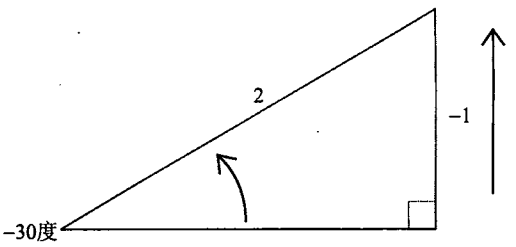


图3-9　Flash坐标系统中的相同三角形

因此，比例变为了–1/2，这是–30度角的正弦。那么–30度角的正弦是–0.5。仍然使用Flash的trace方法来验证它。

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

这并不是很痛苦，不是吗？现在我们来看另一个三角函数：余弦。

3.3.2 余弦

你可以使用Math.cos（angle）函数访问Flash中的余弦。余弦被定义为与角的邻边与斜边的

www.TopSage.com

TopSage.com

比例。图3-10显示了它们的关系。

图3-10显示的是与图3-9相同的角度，但是现在加入了邻边的近似测量值：1.73。注意它的方向是向右的，因此对于*x*测量，它是正值。这个角的余弦为1.73/2或0.865。我们可以认为–30度角的余弦值是0.865。使用下面的代码测试：

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

它与上一次得出的结果一样，只是这次调用的是Math.cos，而不是Math.sin。这次结果为0.866 025 403 784 439，它与0.865非常接近。这个不同是因为对邻边的长度进行了舍入。对于图中的三角形，实际的长度应更接近于1.732 050 807 568 88。如果用它除以2，你会发现其结果与–30度的实际余弦值几乎相等。

到目前为止，我们都是从左下角开始的。要是从右上角开始会怎么样呢？好，首先需要重新定向三角形使正在讨论的那个角符合这个坐标系统，如图3-11所示。这被称为*把角放置在标准位置上*（尽管Flash中的“标准”与通常“标准”相反）。这个角度等于60度，因为它是顺时针的，所以为正的。重直方向的测量是从这个角开始向下的，因此它是正的，水平方向的测量是向右的，因此它也是正的。（我增加了一个+号是为指出其不同，但在通常情况下，这个+号是不需要的。如果没有明确标明值为负则值将是正值。）

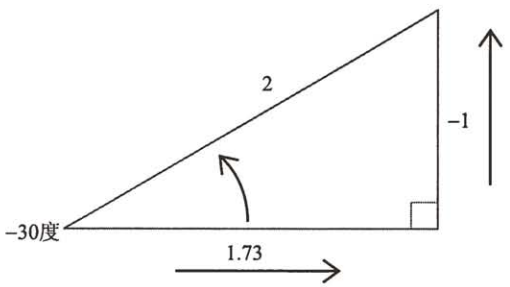


图3-10 角的余弦是邻边 / 斜边

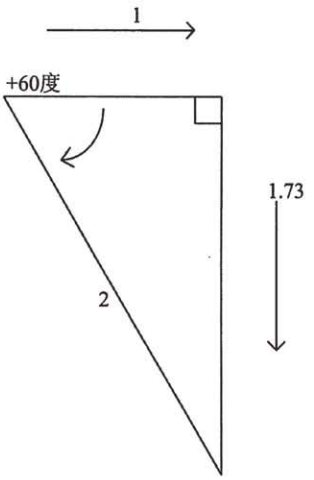


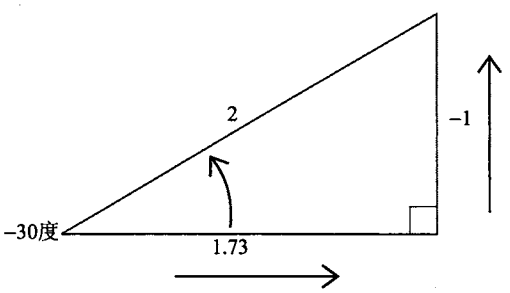
图3-11 看相对的角

现在，这个角的正弦值是对边与斜边的比例，即1.73/2（0.865）。它的余弦值是邻边与斜边的比例，即1/2或0.5。因此，基本上，一个角的余弦等于另一个角的正弦，且一个角的正弦是另一个角的余弦。我不确定这在Flash中会有什么用途，但重要的是要注意它们的关系和比例，它们之间都是相互关联的。

3.3.3 正切

另一个主要的三角函数是正切函数，在Flash中使用Math.tan（angle）。它描述的是对边与邻

边之间的关系，如图3-12所示。



59
~
60

图3-12　一个角的正切是对边/邻边

这里，比例的计算结果是-1/1.73或-0.578。要得到更准确的值，可以直接在Flash中查看：

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

你将得到-0.577 350 269 189 626。

实践告诉我们，在你每天的ActionScript动画编码过程中，你并不会经常使用正切函数。用得更多的是正弦和余弦函数，尽管有时正切也可以用来创建出一些有趣的效果。

另一方面，反正切函数却是非常有用，稍后你会看到，所以先记住正切比例。

3.3.4　反正弦和反余弦

类似于正切，反正弦和反余弦在一般的Flash动画中并不是很有用。然而，了解它们和知道怎样使用它们是很重要的。基本上，它们要做的就是正弦和余弦的反操作。换句话说，你提供一个比例，就会得回一个角（弧度）。ActionScript函数为Math.asin（ratio）和Math.acos（ratio）。让我们进行一下快速的测试以确认它们可以工作。

好，你已经知道30度角的正弦值是0.5。因此，0.5的反正弦应当是30度。测试如下：

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

记得将弧度转换为度，以使我们能够看到30，而不是弧度值0.523。

你已经知道了30度角的余弦值约为0.865。记住如果测试这个经过舍入的值，你可能不会得到精确的30，但是它会非常接近以证明这一点。下面是测试：

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

你将得到30.117 294 747 322 1的结果。如果想回过头来使用准确的30度角的余弦值，你将得到更加精确的结果。

看，这些内容并不是很难，不是吗？并且你几乎已经学完了基本的函数。你只要再多知道一点，就可以开始看使用三角学能真正实现的东西了。

3.3.5　反正切

毫无疑问，你会猜到，反正切只是正切的逆。只要提供对边和邻边的比例，就会得到那个角

www.TopSage.com

度。在Flash中，实际上有两个函数来检查反正切。第1个的命名和工作方式与前面的例子一样，它是Math.atan (ratio)，你需要提供给它一个对边和邻边相除得到的小数。 61

例如，从前面的讨论中已经知道30度角的正切值为0.577（舍入的结果）。你可以试一下这个：

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

你将得到一个非常接近30的值。现在，它看上去很简单而且很直观，那为什么还需要用另外一个函数来处理同样的事情呢？好，为了回答这个问题，参见图3-13。

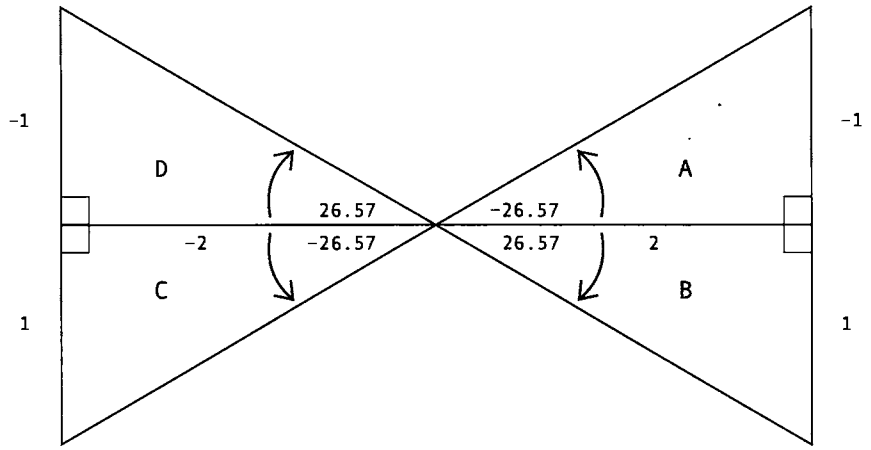


图3-13 4个象限中的角

图3-13显示了4个不同的三角形：A、B、C和D。三角形A和B有一个正的x值，三角形C和D延伸到负的x值。同样，三角形A和D位于负的y空间，而三角形B和C有负的y值。因此，针对这4个内角的比例，你得到如下：

- A: –1/2 或 –0.5
- B: 1/2 或 0.5
- C: 1/–2 或 –0.5
- D: –1/–2 或 0.5

因此，假设用对边除以邻边得到的比例为0.5，提供这个比例并计算Math.atan (0.5)，将它转换为度，得的值约为26.57。但是你指的是哪一个三角形：B还是D呢？确实没有办法知道，因为两者的比例都是0.5。这看起来好像不重要，但就像你将在稍后出现的一些实际的例子中所看到的，它变得非常重要。

欢迎Math.atan2 (y,x)。 它是Flash中的另一个反正切函数，并且比Math.atan (ratio) 有用得多。实际上，我甚至可以说你最后可能会只用这个函数。这个函数有两个参数：对边的大小 62
与邻边的大小。对于Flash中的大多数用法，它意味着y测量和x测量。一个常见的错误是作为x,y输入它们，而不是指定的y, x。对于给出的例子，你应该输入Math.atan2 (1,2)。继续测试它，别忘了将它转换为度。

www.TopSage.com

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

它给出的角应为26.565 051 177 078，对于前面提到的三角形B它是正确的。现在，还有那个-1/-2（三角形D）会让我们有些迷惑，让我们测试一下。

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

它给出了一个意想不到的结果-153.434 948 822 922。这是什么呢？可能下面的图3-14会给你解释。

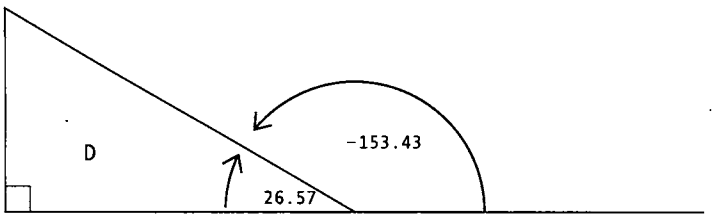


图3-14　一个角的两种测量方式

在三角形D中，从它自身的底部的直角边算起，内角确实是+26.57，但是，记住在Flash中角是从*x*轴的正方向顺时针测量的。因此，从Flash的屏幕测量的角度看，你正在看的这个角的度数是-153.43。

这有什么用途呢？好，让我们开始第1个有关三角学在Flash中的实践应用。

3.4　旋转

任务目标：你想旋转一个精灵或一个影片剪辑以使它总是可以指向鼠标。旋转是一个需要加入到你工具箱中的非常有用的工具。它可以用于游戏、鼠标跟随、跟随屏幕上的鼠标的那些眼睛、界面元素等。实际上，旋转不只是限制在鼠标上。因为鼠标的坐标只是*x*和*y*值，你能够扩展这项技术以强制一个影片剪辑定位在任意指定的点上，如定位在另一个影片剪辑上或屏幕的中心或某一个角落。

让我们来看一个示例。你可以跟着每一步学习或打开文档类RotateToMouse.as和Arrow.as
63 （你可以从www.friendsofed.com下载，包括本书所有其他的代码）来完成你的所有工作。

首先，你需要一个东西来旋转。它将是一个绘制有箭头的精灵。实际上，因为你可能会再次使用这个箭头精灵，所以你可以通过扩展Sprite类来创建它。

```
package {
   import flash.display.Sprite;

   public class Arrow extends Sprite {
      public function Arrow() {
         init();
      }
      public function init():void {
         graphics.lineStyle(1, 0, 1);
         graphics.beginFill(0xffff00);
         graphics.moveTo(-50, -25);
```

www.TopSage.com

```
            graphics.lineTo(0, -25);
            graphics.lineTo(0, -50);
            graphics.lineTo(50, 0);
            graphics.lineTo(0, 50);
            graphics.lineTo(0, 25);
            graphics.lineTo(-50, 25);
            graphics.lineTo(-50, -25);
            graphics.endFill();
        }
    }
}
```

这只是使用drawing API（我将在下一章中讲解）在精灵被创建时绘制一个箭头。现在，无论什么时候需要一个箭头精灵时，你只要说new Arrow()!可以在图3-15上看到结果。要记住的最重要的事情是当我们绘制任何用来旋转的内容时，要确保它是“指向”右边的，或者说*x*轴正方向，因为这是它旋转到0度时的样子。

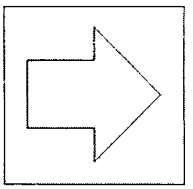


图3-15 使用drawing API绘制的箭头

你将需要创建一个Arrow类的实例，将它放置在舞台的中心并让它指向鼠标。图3-16显示了你要做的事情。

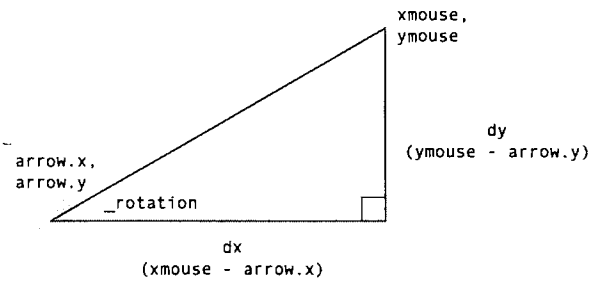


图3-16 显示接下来你要计算的内容

64

看上去熟悉吗？它与刚刚处理的三角形是一样的，只是换成了鼠标的坐标和箭头的坐标。鼠标位置总是可以通过mosueX和mouseY属性读取。你可以从箭头的x和y属性获取它的位置。把它们相减，就可以得到三角形的两个直角边的长度。现在，你只需使用Math.atan2（dy,dx）来获取角度。然后再将它转换成度并将它设置为箭头的rotation属性。代码如下：

```
var dx:Number = mouseX - arrow.x;
var dy:Number = mouseY - arrow.y;
var radians:Number = Math.atan2(dy, dx);
arrow.rotation = radians * 180 / Math.PI;
```

当然，要得到动画，你需要设置一个循环。像前面的章节讨论的那样，事件处理器是最好的办法。让我们使用enterFrame事件。下面是完整的文档类（document class）：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
```

www.TopSage.com

大家网 TopSage.com

```
public class RotateToMouse extends Sprite {
    private var arrow:Arrow;

    public function RotateToMouse()   {
        init();
    }
    private function init():void {
        arrow = new Arrow();
        addChild(arrow);
        arrow.x = stage.stageWidth / 2;
        arrow.y = stage.stageHeight / 2;
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    public function onEnterFrame(event:Event):void {
        var dx:Number = mouseX - arrow.x;
        var dy:Number = mouseY - arrow.y;
        var radians:Number = Math.atan2(dy, dx);
        arrow.rotation = radians * 180 / Math.PI;
    }
  }
}
```

确保Arrow.as文件与RotateToMouse.as文件在同一个目录下并且使用RotateToMouse作为文档类创建SWF。你现在知道了什么？它立见功效！

现在，假设你没有使用Math.atan2。你可以通过dy除以dx得到对边和邻边的比例，并且将它传给Math.atan函数。你所需要做的就是将前面代码中的Math.atan2改成Math.atan，如下：

65

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

试一下这个，你很快就会发现问题。如果鼠标在箭头的左边，箭头将不会指向它，而是直接离开它。你能明白这是为什么吗？回到之前显示三角形A、B、C和D的那个图（图3-13），记得三角形A和C有相同的比例，三角形B和D的也一样。这对Flash来说没有办法知道你指的是哪个角，因此它只是给出了A或B。如果你的鼠标在D象限，Flash将返回B象限的角并让鼠标旋转到那里。

无疑，你现在已经看到Math.atan2的好处了。在本书中你将会多次使用它。

3.5　波

让我们接触一些三角学在Flash中的更具体的应用。你在之前一定听说过术语正弦波（sine wave），而且也见过图3-17所示的形状，它是正弦波的一个图形表示。

但是这个形状与正弦函数到底有什么关系呢？它是表示正弦函数的那些结果的图形，当提供了从0到360（或0到2 pi弧度）的所有角时。从左到右是使用的角的值，*y*方向的值是指定角的正弦值。在图3-18中，我已经标出了一些特殊的值。

现在你可以看到0度的正弦值为0。90度或pi/2弧度的正弦值是1。180度或pi弧度的正弦值又是0。270度或3/2 pi的正弦值是–1。360度或2 pi的正弦值再一次返回到0。让我们在Flash中用一下这个正弦波。将下面的代码放入到一个文档类架构中并且测试它：

www.TopSage.com

```
for(var angle:Number = 0;angle < Math.PI * 2; angle += .1){
     trace(Math.sin(angle));
}
```

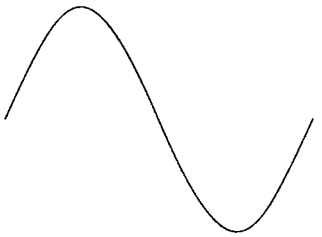


图3-17 正弦波

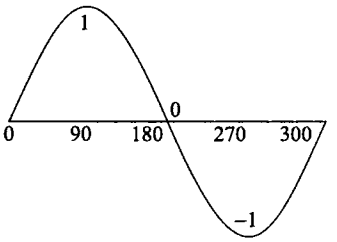


图3-18 正弦值

从这里开始，你需要开始习惯只使用弧度。把度抛在脑后，除非你为旋转或其他的目的而真正需要使用它们时再使用它们。

在这个例子中，角变量从0开始每次增加0.1直到它超过Math.PI*2为止。它会输出那个角的正弦值。如果你看一下这些结果，就会发现它们从0开始，增长到几乎为1的时候，然后又降到几乎为-1，最后返回到约为0的值。你永远无法得到精确的1或0，因为我们使用的增量是0.1，你永远得不到pi或pi/2的精确倍数。

3.5.1 光滑的上下运动

那么我们可以使用Math.sin（angle）来做些什么呢？你是否需要上下或者前后光滑地移动某个东西呢？这就是你的函数。考虑一下这个：并不是只从0到1然后到-1然后返回到0，而且停在那里，而是不断地增加它的角度。你将会反复得到那个波。还有就是并不是只取1和-1，让这些值乘以某个更高的值，如100，你就会得到一连串的从100到-100的值，来来回回，持续不断。

下面的文档类Bobbing.as使用了一个新的自定义的精灵，它是在Ball类中定义的。你可以在下面的代码中看到：

```
package {
    import flash.display.Sprite;

    public class Ball extends Sprite {
        private var radius:Number;
        private var color:uint;

        public function Ball(radius:Number=40, color:uint=0xff0000) {
            this.radius = radius;
            this.color = color;
            init();
        }
        public function init():void {
            graphics.beginFill(color);
            graphics.drawCircle(0, 0, radius);
            graphics.endFill();
        }
```

www.TopSage.com

66
~
67

```
    }
}
```

当它被创建时，它只是绘制了一个圆。如果需要，你可传入一个半径和一个颜色。如果没有传入，它将使用默认的半径40，颜色为红色（AS 3的另一个新增变化）。这个类很简单，但是很有用，你会在本书的后面内容中经常用到它。因此要保持它是方便的。

文档类创建一个Ball类的实例，将它加到舞台上，并设置了一个使小球上下移动的enterFrame侦听器：

```
package {
   import flash.display.Sprite;
   import flash.events.Event;

   public class Bobbing extends Sprite {
      private var ball:Ball;
      private var angle:Number = 0;
      public function Bobbing()  {
         init();
      }
      private function init():void {
         ball = new Ball();
         addChild(ball);
         ball.x = stage.stageWidth / 2;
         addEventListener(Event.ENTER_FRAME, onEnterFrame);
      }
      public function onEnterFrame(event:Event):void {
         ball.y = stage.stageHeight / 2 + Math.sin(angle) * 50;
         angle += .1;
      }
   }
}
```

首先，你需要创建一个angle属性并将其初始化为0。

在onEnterFrame方法中，求得那个角的正弦值并将其乘以50，这样将获得一个从50到–50的值的范围。如果再将它与舞台的高度的一半相加，你将获得从250到150（基于400像素高的舞台）的值的范围。这就构成了小球的y位置。然后在后面每次循环时那个角增加0.1。你将得到一个漂亮的光滑的上下运动。

通过使用各种值，你将会发现将0.1变成另一个值就会改变运动的速度。这很容易理解，因为角增加得越快或越慢，Math.sin的返回值从1到–1变化得就越快或越慢。显而易见，改变50那个值就会改变小球移动的距离，改变stage.stageHeight/2为其他某个值就会改变它的上下运动的开始位置。从这一点上看，你可以把这些值抽象成一些像这样的变量（只给出了附加的或类中改变的部分）：

```
// 在类的顶端:
private var angle:Number = 0;
private var centerY:Number = 200;
private var range:Number = 50;
private var speed:Number = 0.1;
```

68

www.TopSage.com

```
// 在处理器函数中:
public function onEnterFrame(event:Event):void {
      ball.y = centerY + Math.sin(angle) * range;
      angle += speed;
}
```

将实际数值保持在动画代码之外是一个非常好的习惯，并且你应该尽可能多地这样做。在这个例子中，不管怎么说，它们都非常好地放在了同一个代码位置上。但是当你有好几页代码并且那些值会在多个位置使用时会有什么情况呢？那么每次你需要修改速度时，你需要找到具有0.1值的每个实例然后修改它。你可以通过查找和替换，但是如果文件中其他某个东西也有0.1时，它也会被替换。如果那些数都保持在代码之外，更适宜放在程序的最顶端，你会很准确地知道所有的变量在哪儿。

3.5.2 线性垂直运动

在Wave1.as文件中，我加入了线性垂直运动，只是想给你些动画制作的灵感。这里是文件的代码：

```
package
{
   import flash.display.Sprite;
   import flash.events.Event;

   public class Wave1 extends Sprite {
      private var ball:Ball;
      private var angle:Number = 0;
      private var centerY:Number = 200;
      private var range:Number = 50;
      private var xspeed:Number = 1;
      private var yspeed:Number = .05;

      public function Wave1()   {
         init();
      }
      private function init():void {
         ball = new Ball();
         addChild(ball);
         ball.x = 0;
         addEventListener(Event.ENTER_FRAME, onEnterFrame);
      }
      public function onEnterFrame(event:Event):void {
         ball.x += xspeed;
         ball.y = centerY + Math.sin(angle) * range;
         angle += yspeed;
      }
   }
}
```

69

3.5.3 脉冲动画

需要记住的一件比较重要的事情就是可以将正弦值应用到除了物理位置以外的其他内容。在

www.TopSage.com

Pulse.as文件中，我使用正弦值来影响小球的缩放。它呈现出一种脉冲的现象，代码为：

```
package {
   import flash.display.Sprite;
   import flash.events.Event;

   public class Pulse extends Sprite {
      private var ball:Ball;
      private var angle:Number = 0;
      private var centerScale:Number = 1;
      private var range:Number = .5;
      private var speed:Number = .1;

      public function Pulse()   {
         init();
      }
      private function init():void {
         ball = new Ball();
         addChild(ball);
         ball.x = stage.stageWidth / 2;
         ball.y = stage.stageHeight / 2;
         addEventListener(Event.ENTER_FRAME, onEnterFrame);
      }
      public function onEnterFrame(event:Event):void {
         ball.scaleX = ball.scaleY = centerScale +
                                     Math.sin(angle) * range;
         angle += speed;
      }
   }
}
```

原理是一样的。你有一个中心点（在本例中是100%缩放），一个范围和一个速度。不要停在那里。将某些正弦波应用到alpha（透明度）、rotation或其他任何有趣的属性。

3.5.4　两个角的波

下面是让你开始的另一个想法：不是只指定一个角，而是设定两个值angle1和angle2，两者有不同的中心和速度。一个正弦波用于一个属性，另一个正弦波用于另一个属性，如位置和缩放。我不能保证你会想出任何有用的事情，但是像这样的使用方式，你可以真正感受到这些函数是如何工作的。

我已经在文档类Random.as中包括了一个例子。它设定了两个角、速度和中心，并且将其中一个角应用到小球的x位置而另一个角应用到y位置。最终结果就像一个虫子在一个屋子里面飞。尽管它们是预定好的数学公式，但是看上去很随机。下面是代码：


```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Random extends Sprite {
        private var ball:Ball;
```

www.TopSage.com

TopSage.com

```
        private var angleX:Number = 0;
        private var angleY:Number = 0;
        private var centerX:Number = 200;
        private var centerY:Number = 200;
        private var range:Number = 50;
        private var xspeed:Number = .07;
        private var yspeed:Number = .11;

        public function Random()    {
            init();
        }
        private function init():void {
            ball = new Ball();
            addChild(ball);
            ball.x = 0;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        public function onEnterFrame(event:Event):void {
            ball.x = centerX + Math.sin(angleX) * range;
            ball.y = centerY + Math.sin(angleY) * range;
            angleX += xspeed;
            angleY += yspeed;
        }
    }
}
```

3.5.5 用 drawing API 绘制波

最后，在Wave2.as中，我去掉了小球而使用drawing API来绘制正弦波。下面是这个文件的代码：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Wave2 extends Sprite {
        private var angle:Number = 0;
        private var centerY:Number = 200;
        private var range:Number = 50;
        private var xspeed:Number = 1;
        private var yspeed:Number = .05;
        private var xpos:Number;
        private var ypos:Number;

        public function Wave2()    {
            init();
        }
        private function init():void {
            xpos = 0;
            graphics.lineStyle(1, 0, 1);
            graphics.moveTo(0, centerY);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
```


www.TopSage.com

```
        }
        public function onEnterFrame(event:Event):void {
            xpos += xspeed;
            angle += yspeed;
            ypos = centerY + Math.sin(angle) * range;
            graphics.lineTo(xpos, ypos);
        }
    }
}
```

我将在下一章详细地讲解drawing API，但是你应该有兴趣修改这个文件并看你能用它绘制的各种波形。注意正弦波不是上下的，因为在Flash中的*y*轴是反向的。

3.6　圆和椭圆

现在你已经掌握了正弦波，下面让我们转到与正弦波类似但又很少有人知道的余弦波。它们的形成方式与正弦波相同，只是使用的是余弦函数而不是正弦函数。如果你能从之前的讨论中想起正弦与余弦两者基本上是相反的，了解到两个波形成了同样的形状但位置不同时，你并不会吃惊。如图3-19显示的是一个余弦波。

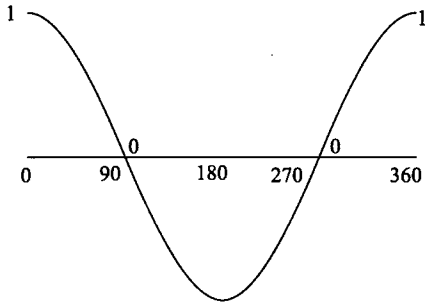


图3-19　余弦波

它显示了0度的余弦值为1，并且当它移到2 pi弧度或360度时，它变为了0、–1、0，然后又回到了1。因此，它本质是与正弦波产生的一样的曲线，只是被移动了一点。

3.6.1　圆形运动

不言而喻，在任何情况下当你只需要一个往复运动时都可以使用余弦代替正弦。但是余弦在与正弦相配合的情况下有一个更常见的功能：在圆中移动一个物体。图3-20显示了一个物体绕着圆移动时的几个不同。

如果将图3-20中的圆转动到可以从右边直接看到它的边，你将只看到物体在作上下运动。它的中心将是圆的中心，并且它的运动范围就是圆的半径。你可以像在第1个正弦练习中所做的一样计算出它的位置：通过让角的正弦值乘以运动范围。在这个例子中，正弦当然是最合适的函数，因为如果你看一下已形成的三角形，你正在计算的是y的长度——角的对边。

现在，假设你正在从圆的底边上看圆。从这个视角上看，你会看到物体是左右运动的。这一

www.TopSage.com

次，你正在计算的是x的长度——和角的邻边——因此你应当使用余弦。

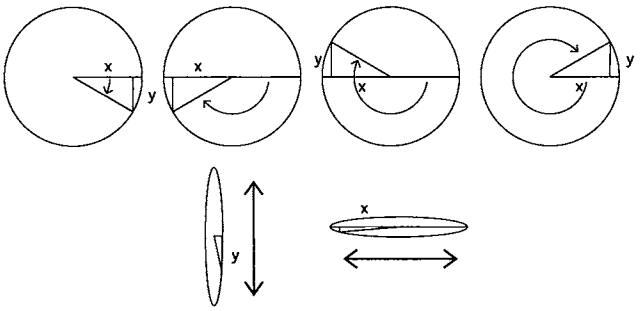


图3-20 一个物体绕圆移动时的一些位置

需要意识到的比较重要的事情是这些计算使用的是相同的角，不像之前的Random.as示例，它使用的是两个不同的角来计算x和y位置。你只需要使用正弦来计算y和使用余弦来计算x。在ActionScript中代码如下：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    public class Circle extends Sprite {
        private var ball:Ball;
        private var angle:Number = 0;
        private var centerX:Number = 200;
        private var centerY:Number = 200;
        private var radius:Number = 50;
        private var speed:Number = .1;

        public function Circle()    {
            init();
        }
        private function init():void {
            ball = new Ball();
            addChild(ball);
            ball.x = 0;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        public function onEnterFrame(event:Event):void {
            ball.x = centerX + Math.sin(angle) * radius;
            ball.y = centerY + Math.cos(angle) * radius;
            angle += speed;
        }
    }
}
```

你可以自己创建这个示例，或者打开并使用作为你的Circle.as文档类，它可以为你做所有的工作。测试它并验证你确实有一个完美的圆。

注意这两种情况中的范围实际上是三角形的斜边，它等于圆的半径。因此，我将range改为

了radius来更好地反映事实。

这段代码做的所有事情就是使用余弦来得到x位置和使用正弦来得到y位置。你应当熟悉这些关系。在Flash中，几乎每次谈论到x，你应当立即想到余弦，并且几乎总是应当将y与正弦联系起来。事实上，你需要花多点时间来完全理解最后这一点代码。它将是你的ActionScript动画工具箱中的最有用的工具之一。

3.6.2　椭圆形运动

虽然圆很美，但有些时候一个非常完美的圆并不是你真正需要的。你可能需要的是一个扁圆或是椭圆。问题在于半径。它使得*x*运动和*y*运动的范围相同，所以你会得到一个圆。

要得到一个椭圆形，你所要做的就是计算*x*和*y*位置时使用不同的半径值。我把它们称为radiusX和radiusY。从严格的几何观点来看这个术语可能是一个不合适的选择，但是它很简单、直观，并且容易记住和想象。因此我把它们用作我的变量名称。下面是如何使用它们，正如我在
74 Oval.as文件中所包含的：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Oval extends Sprite {
        private var ball:Ball;
        private var angle:Number = 0;
        private var centerX:Number = 200;
        private var centerY:Number = 200;
        private var radiusX:Number = 200;
        private var radiusY:Number = 100;
        private var speed:Number = .1;

        public function Oval()    {
            init();
        }
        private function init():void {
            ball = new Ball();
            addChild(ball);
            ball.x = 0;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        public function onEnterFrame(event:Event):void {
            ball.x = centerX + Math.sin(angle) * radiusX;
            ball.y = centerY + Math.cos(angle) * radiusY;
            angle += speed;
        }
    }
}
```

这里，radiusX是200，它意味着小球环行时将从centerX向后和向前移动200像素。radiusY是100，它意味着它的上下移动每次都是100像素。因此，现在你有一个不规则的圆，它根本不是一个圆，而是一个椭圆。

www.TopSage.com

3.7 毕达哥拉斯定理

最后，我们来看毕达哥拉斯定理。我并不确定它是怎样成为三角学的一部分的，但它跟这个主题交织在一起而且是你将使用很多的另一个公式。因此，把它放在这里讲最好不过了。

毕达哥拉斯是很久以前希腊的一个伟大的人物。这在你的历史课程中有介绍。简单来说，它的定理内容是A的平方+B的平方=C的平方。现在，如果你之前就已经知道这个定理，那是再好不过的。否则，它听起来有点像是奇怪的押韵的短诗。因此让我们更深一步地解释它。 75

对这个定理的更加描述性的表达为：直角三角形的两条直角边的平方和等于斜边的平方。这说起来有点拗口。假设你有一个如图3-21所示的三角形。

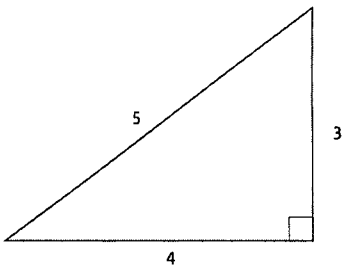


图3-21 一个直角三角形

两条直角边，A和B的长度分别是3和4。斜边C的长度是5。毕达哥拉斯先生告诉我们$A^2+B^2=C^2$。让我们测试一下它。加入实际数值后就有$3^2+4^2=5^2$，它的结果是9+16=25。嗯，结果是完全正确的。

现在，如果你恰好知道那3条边的长度，那么毕达哥拉斯定理只是一个有趣的关系而已。但是如果你只知道其中的两个长度，它就变成了一个可以快速得到第3条边的长度的强大的工具。在Flash中，最常见的情况是知道两条直角边的长度并且你想知道斜边的长度。更具体地说，其实你想得到的是两点之间的距离。

3.8 两点之间的距离

假设有两个精灵在舞台上，你想知道它们之间有多远。这是毕达哥拉斯定理在Flash中最常见的应用。

你需要哪些东西来工作呢？你需要有每个精灵的x和y位置。让我们把其中一个位置称为x1,y1，另一个称为x2,y2。因此，情况就像是图3-22中描述的那样。

如果你在阅读本章的过程中已经被灌输了一些思想，你将在图3-22中看到一个直角三角形在图中形成，距离线作为斜边。在图3-23中，我完成了它并加入了具体的数值。

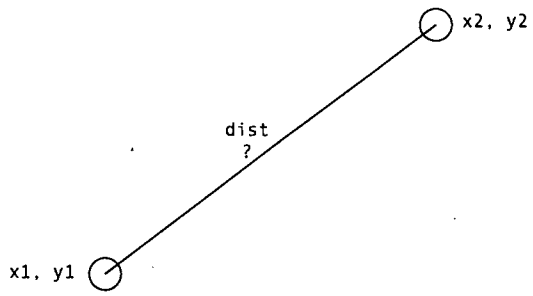


图3-22 两个物体之间的距离是多少？

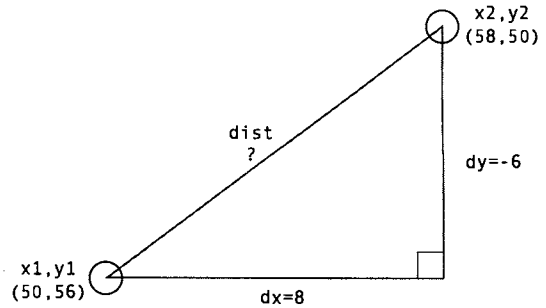


图3-23 转换为一个直角三角形

这里，dx是两个精灵在*x*轴上的距离，dy是两者在*y*轴上的距离。你可以用x2减去x1很容易地

www.TopSage.com

大家网

得到dx：58–50=8。类似地，用y2减去y1就得到了dy的值，它是6。现在，根据毕达哥拉斯定理，如果将两者分别平方再加起来，你将得到距离的平方。换句话说，$(-6)^2+8^2=dist^2$。计算一下，将得到$36+64=dist^2$或$100=dist^2$。从基本的代数学上说你可将它简化为$\sqrt{100}=dist$。而且根据它，你可以很容易地得到两个剪辑之间的距离为10。

现在，让我们抽象一下以使你可以得到一个可以在任何情况下使用的公式。给定两个点x1,y1和x2,y2，你计算出x距离和y距离，对它们分别求平方，取它们的和，然后再开平方。下面是ActionScript代码：

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

要特别注意这几行。它们在你的工具箱中构成了另一个大工具。前两行得到的只是每个轴上的距离。注意如果你只是计算距离，可以不必使用dx和dy计算任何角，使用x1减x2或反过来x2减x1都不会有任何问题。因为最终的结果总是正数。最后一行是将3步集中到了一起：它对每个值取平方，然后将它们加起来，最后开平方。为了清晰，你可以将它们细分成3个单独的语句。但是一旦你熟悉了那一个单行，它对你来说是很清楚的。你看到它就会想到“毕达哥拉斯定理”。然后你就会知道你已经学会了。

76～77

现在，让我们在现实世界中试验一下。下面这个文档类Distance.as创建了两个精灵，随机对它们进行定位，然后计算并跟踪它们之间的距离。

```
package
{
    import flash.display.Sprite;

    public class Distance extends Sprite {

        public function Distance()    {
            init();
        }
        private function init():void {
            var sprite1:Sprite = new Sprite();
            addChild(sprite1);
            sprite1.graphics.beginFill(0x000000);
            sprite1.graphics.drawRect(-2, -2, 4, 4);
            sprite1.graphics.endFill();
            sprite1.x = Math.random() * stage.stageWidth;
            sprite1.y = Math.random() * stage.stageHeight;

            var sprite2:Sprite = new Sprite();
            addChild(sprite2);
            sprite2.graphics.beginFill(0xff0000);
            sprite2.graphics.drawRect(-2, -2, 4, 4);
            sprite2.graphics.endFill();
            sprite2.x = Math.random() * stage.stageWidth;
            sprite2.y = Math.random() * stage.stageHeight;

            var dx:Number = sprite1.x - sprite2.x;
```

www.TopSage.com

```
            var dy:Number = sprite1.y - sprite2.y;
            var dist:Number = Math.sqrt(dx * dx + dy * dy);
            trace(dist);
        }
    }
}
```

编译这个影片，然后你将得到这两个精灵之间的距离。每次运行它，这两个精灵都会在不同的位置。其中一个在另一个的左边、右边、上边或下边都是没有关系的。你将始终得到一个正的距离值。

好，它是很有趣的，但并不是很动态。只是示范你可以在现实生活中使用它，而且它并不限制于精灵或影片剪辑，试一下下面的文档类MouseDistance.as：

```
package
{
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import flash.text.TextField;

    public class MouseDistance extends Sprite {
        private var sprite1:Sprite;
        private var textField:TextField;
        public function MouseDistance()    {
            init();
        }
        private function init():void {
            sprite1 = new Sprite();
            addChild(sprite1);
            sprite1.graphics.beginFill(0x000000);
            sprite1.graphics.drawRect(-2, -2, 4, 4);
            sprite1.graphics.endFill();
            sprite1.x = stage.stageWidth / 2;
            sprite1.y = stage.stageHeight / 2;

            textField = new TextField();
            addChild(textField);
            stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
        }
        public function onMouseMove(event:MouseEvent):void {
            graphics.clear();
            graphics.lineStyle(1, 0, 1);
            graphics.moveTo(sprite1.x, sprite1.y);
            graphics.lineTo(mouseX, mouseY);
            var dx:Number = sprite1.x - mouseX;
            var dy:Number = sprite1.y - mouseY;
            var dist:Number = Math.sqrt(dx * dx + dy * dy);
            textField.text = dist.toString();
        }
    }
}
```

这里，dx和dy是用sprite1的位置减去当前鼠标的位置得到的。dist值放到了一个文本域中，一条线从影片剪辑绘制到鼠标的位置（在下一章你还会发现在drawing API上绘制的线）。最后，

www.TopSage.com

把所有的事情交给了onMouseMove处理器，以在每次鼠标移动时更新它。

测试该文件并移动你的鼠标。一条线将把它连到影片剪辑上，你将获得一条固定长度的连接线条。

在后面的章节中，当我谈到碰撞检测时，你将发现内置碰撞测试方法的一些弱点，并且看到如何使用毕达哥拉斯定理公式来创建一个基于距离的碰撞检测方法。在计算像重力或弹力这样的力时它也是非常有用的，因为两个物体之间的力与它们之间的距离是成比例的。

3.9　本章重点公式

看这儿，你已经有一个崭新的工具箱，而且已经有6个以上的工具放在了里面。所有的工具也都将在第19章中出现，但是现在让我们看一下到目前为止你都加了哪些工具。注意我已经使这些公式尽可能简单和抽象，没有包括数据类型或变量声明。你需要根据情况使用所有需要的正确
79 语法来将公式加到你自己的类中。

计算基本的三角函数：

```
角的正弦 = 对边 / 斜边
角的余弦 = 邻边 / 斜边
角的正切 = 对边 / 邻边
```

转换弧度为度和转换度为弧度：

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

旋转到鼠标位置（或任意点）：

```
// 用x,y点替换mouseX,mouseY来旋转。
dx = mouseX - sprite.x;
dy = mouseY - sprite.y;
sprite.rotation = Math.atan2(dy, dx) * 180 / Math.PI;
```

创建波：

```
// 指定值给精灵或影片剪辑的x,y或其他属性，
// 用于绘制坐标等。
public function onEnterFrame(event:Event){
   value = center + Math.sin(angle) * range;
   angle += speed;
}
```

创建圆：

```
// 指定位置给精灵或影片剪辑的x和y，
// 用于绘制坐标等。
public function onEnterFrame(event:Event){
   xposition = centerX + Math.cos(angle) * radius;
   yposition = centerY + Math.sin(angle) * radius;
   angle += speed;
}
```

www.TopSage.com

创建椭圆：

```
// 指定位置给精灵或影片剪辑的x和y，
// 用于绘制坐标等。
public function onEnterFrame(event:Event){
   xposition = centerX + Math.cos(angle) * radiusX;
   yposition = centerY + Math.sin(angle) * radiusY;
   angle += speed;
}
```

80

获得两点之间的距离：

```
// 点为 x1,y1和x2,y2,
// 可以是精灵或影片剪辑的位置、鼠标的坐标等。
dx = x2 - x1;
dy = y2 - y1;
dist = Math.sqrt(dx*dx + dy*dy);
```

当然，如果我只是想发布一系列公式，它会省去我很多时间。因此仔细查看这些公式并确保你已经完全理解它们是如何工作的。如果你有任何问题，就回到本章介绍它的位置，用它做个试验，而且如果需要就进一步研究它，直到你确实可以用这个概念来思考。

3.10 小结

这一章几乎覆盖了用ActionScript制作动画所需要的所有三角学。有一个原则称为余弦定理，我把它留到现在，是因为它太复杂而且用来处理非直角三角形（它们没有90度角）。如果你沉溺于三角学并且认为这些还不够，你可以提前跳到第14章，那里讲解了反向运动，是三角学的真正用武之地。

但是，现在，你不但知道了正弦、余弦和正切以及它们的反函数：反正弦、反余弦和反正切，而且知道了在ActionScript中计算它们的方法。

最好的是，你得到了一些在Actionscript中使用它们的实际操作经验，有一些是最常见的现实的应用。当翻阅本书的时候，你将发现这些技术还在很多方面很有用。但是现在你已经牢牢掌握了这些概念，当你碰到那些例子，就应当可以理解它们或知道它们是怎样工作的。

下一章我们涉及一些更常见的获取屏幕图像的渲染技术，包括极重要的drawing API。当学习那章时，看你是否可以使用渲染方式来可视化所学过的一些三角函数。我确信你完全可以使用三角学创建出一些漂亮的图形或动画。

81

www.TopSage.com

第4章

渲染技术

本章内容

- Flash中的颜色
- drawing API
- 颜色变换
- 滤镜
- 位图控制
- 本章重点公式

目前为止，本书的示例程序中的图形已经使用了一些简单的命令来绘制，并且我已经几次提过“drawing API”这个神秘的词语。但对于大部分我只是给出了代码而没有过多的解释。

在这一章中，我们将深入到在ActionScirpt中创建可视内容。特别是，我将讲解ActionScirpt和drawing API中的颜色、ColorTransform类、滤镜和BitmapData类。

在本章中你将学习许多关于颜色的知识，让我们先从这个题目开始。

4.1 Flash 中的颜色

在Flash中，颜色被指定为数字。一个颜色的数值可以是从0到16 777 215中的任何数字，称为24位色。在这里你可能在疑惑这些数字的意义，它来自事实，就是存在16 777 216个可能的颜色值，即256×256×256。Flash使用RGB颜色，意味着每一种颜色是由红、绿和蓝构成的。每一种组成色的值都可以是从0到255之间的数值。因此对于红、绿和蓝都有256种可能的色调。这导致了将近1 680万种颜色。

这个系统称为24位色，因为它使用的8位（若干个1或0）来表示数字256。8位乘以3（红、绿、蓝）意味着它使用24位来表示1 680万种可能的颜色。另外，还有一个32位的颜色系统，我们稍后讲到。其额外的8位数字用来表示alpha，即透明度。

现在，显示出11 273 634颜色可能的样子是非常困难的，所以开发人员通常趋向于采用另一个表示这种数字的系统：16进制。如果你在HTML中使用过颜色值，那么它对你来说并不新鲜，但是不管怎样让我们从基础开始。

www.TopSage.com

TopSage.com

4.1.1 使用 16 进制颜色值

Hexadecimal（16进制），或简称为hex，是一个16进制系统。换句话讲，每一个数字都可以是0到15之间的数，而不是像通常的10进制系统那样从0到9。我们没有可用的单个数字用来表示10到15，因此我们借用字母表的前6个字母：A到F。所以在16进制中的每个数字可以从0到F。（在Flash中，16进制数字没有大小写区分，因此你可以使用A到F或a到f。）要在HTML中标记16进制数，我们使用前缀#。和很多其他语言一样，在ActionScript中，我们使用前缀0x。例如，数字0xA等同于10进制的10，0xF等同于10进制中的15，而0x10等同于10进制中的16。

在10进制中，每位数字是它右边数字的10倍。换句话说，数值243意味着100的2倍，10的4倍和1的3倍。在16进制中，每个数字是右边相邻数字的16倍。例如，0x2B3意味着256的2倍，16的B（即11）倍和1的3倍。

对于24位色，其值可达到0xFFFFFF，如果你用数学计算一下，它非常神奇地等于16 777 215。而且，那6个16进制数字可以分成3对。第1对描述红色，第2对描述绿色，剩下两个数字用来描述蓝色。这经常表示为符号0xRRGGBB。（你绝不能将其中的R、G或B当做一个真实的16进制数来用，这里只是一个符号，告诉我们每个数字控制着哪个颜色通道而已。）

84

记住每一种组成色的值可以是0到255，或16进制中的0x00到0xFF的任意数值。因此，红色可以描述成0xFF0000，它意味着纯红色，没有绿色和蓝色。同样，0x0000FF是一个纯蓝色。

如果你取之前提到的数值11 273 634并将它转换为16进制数（一会我将示范一种简单的转换方法），你得到的是0xAC05A2。你可以很容易地将它分成红色＝AC，绿色=05，蓝色=A2。看到红色和蓝色相当高，而绿色几乎没有，你可以猜到它是带有紫色的，而用10进制数却不能说明这一点。

注意在ActionScirpt中使用什么样的数字格式是没有关系的。对于任意使用颜色值的函数，可以使用10进制或16进制。数值11 273 634 与0xAC05A2在Flash中是一样的。只是其中的一种更易读一些。

你可能不知道如何进行两种格式间的转换。好，从16进制转换为10进制非常简单。只是跟踪一下数值。Trace函数会将它转换为10进制的数值。

```
trace(0xAC05A2);
```

转换为16进制可以使用toString（16）方法，如下：

```
trace((11273634).toString(16));
```

它将输出ac05a2。如果你想在某个地方使用这个数值，不要忘记在它的前面加上前缀0x。

4.1.2 透明度和 32 位色

前面我们提到过了，除了24位色之外，还有32位色，它使用了额外的8位来指定透明度。像度和弧度（第3章内容），AS 3使用了24位色和32位色两者。AS 3中的drawing API很大一部分是基于Flash MX（Flash 6）的drawing API来构建的。通常情况下drawing API函数会提供额外的参数来指定透明度，因此它们可以继续使用24位色。另一方面，BitmapData类是在Flash 8的时候引入的

www.TopSage.com

并同时引入了32位颜色。如果对某个特定的函数使用哪个颜色系统有任何疑问，只需要查看一下ActionScript语言参考手册。

我已经讲过的一个情况是你可以按照0xRRGGBB格式创建一个24位色为16进制数。同样，32位色也表示为16进制数，使用的格式为0xAARRGGBB，AA代表着alpha或透明度。因此，0xFFFFFFFF创建了一个不透明的白色，0x00FFFFFF创建了一个完全透明的白色，0x80FFFFFF创建了一个近似50%透明的白色。

4.1.3　新的数值类型：int 和 uint

在之前版本的ActionScript中，只有一种数值类型。它使用Number类型来表示，可以是正或负的整数或浮点值（或0）。Number类型现在仍和我们在一起，而且就像你始终拥有它一样，你可以自由使用它。但是现在增加了两种类型，可以使你的代码更清晰。

第1种新类型是int。这种类型可以是正或负的整数或者是0。如果你试图指定一个浮点数给一个声明为int的变量，那么小数部分就会被截掉。例如，如果你指定值1.9给一个int变量，那么这个变量值只会是1。因此，int类型只能被用在你确定这个变量只用作整数的情况下。一个经常使用int类型的地方是在for循环中用作计数变量。在下面的一小段代码中，变量i永远不会是一个浮点数，因此很显然它的类型应当是int。

85

```
for(var i:int = 0; i < 100; i++) {
    // 在这里做一些事情。
}
```

第2种新类型是uint。它代表着无符号（unsigned）整数（integer）。“无符号”意味着它没有加或减符号，因此总是正数。在AS 3中颜色值几乎总是以uint的类型保存的，而这是由于32位色的引入。因为一个无符号的整数类型可以存放比一个标准（有符号）整数类型大很多的整数。int和uint类型的整数在内部都是保存为32位数，它们可以存放超过40亿的值。但是int要使用其中的1位来指定数的符号（加号或减号）。所以你实际得到的是31位数（只是超过20亿的数值）。

所以一个正的32位颜色值太大而无法使用int类型！要是试一下会发生什么呢？测试下面的代码来看结果：

```
var color1:int = 0xffffffff;
trace(color1);
var color2:uint = 0xffffffff;
trace(color2);
```

值0xFFFFFFFF等同于10进制中的4 294 967 295。因为它太大了而无法适应int类型，所以它“卷起来”并变成了-1！这当然不是你想要的结果。但是如果你使用的是uint，这个结果正好。

因此，因为颜色值将总是正数，并且可能会超过int类型所能容纳的数值，所以我们使用uint来存储它们。

4.1.4　组合颜色

一个比较常见的问题是如何将红、绿和蓝3个值组合成为一个有效的综合颜色值。假设你有3

www.TopSage.com

个变量：红、绿和蓝。它们中的每一个都保存着从0到255的一个数值，并且你想使用它们创建一个可以实际应用的颜色值。下面是实现它的公式：

```
color24 = red << 16 | green << 8 | blue;
```

如果你还有一个alpha值，你可以使用下面这个公式创建一个32位数值：

```
color32 = alpha << 24 | red << 16 | green << 8 | blue;
```

这里使用了你之前可能没有见过的两个位运算符。

位运算符对2进制数进行运算——若干个1或0。

仍使用24位色为例，如果你列出一个颜色值的各个位，你将得到一个由24个1和0组成的字符串。将16进制的0xRRGGBB变成2进制，你会有下面的结果：

86

```
RRRRRRRRGGGGGGGGBBBBBBBB
```

你看到8位是红色，8位是绿色和8位是蓝色。这很容易理解，因为8位等于256。

在颜色组合公式中，第1个位运算符是<<，它是位左移运算符。这个左移表示将数值的2进制表示向左移动几位。假设你有一个红色值0xFF或255。它用2进制表示为：

```
11111111
```

向左移动16位之后如下：

```
111111110000000000000000
```

作为一个24位色的数，它是红色，它的16进制表示为0xFF0000——再一次，它是纯红色。现在，假设你有一个绿色值0x55（10进制为85）。在2进制中，它看上去如下样子：

```
01010101
```

向左移动8位，你得到如下：

```
000000000101010100000000
```

那么，原始的8位现在完全处于绿色的范围中。

最后，你有一个蓝色值0xF3（十进制中是243），它在二进制中如下：

```
11110011
```

因为它已经处于蓝色的范围中，所以你不需要动它。现在你已经有了红、绿、蓝的3个值：

```
111111110000000000000000
000000000101010100000000
000000000000000011110011
```

你只需要将它们加在一起形成一个24位色的数值，但这里有一个快速并且很酷的方法：使用位或（OR）运算，符号为 | 。它在二进制级别上比较两个数的数字，如果其中的一个数为1，那么结果就是1。如果两个数都是0，那么结果就是0。你可以对红、绿和蓝数值一起进行位或（OR）运算如可以说成，“如果这个OR这个OR这个是1，结果就是1。”那么你得到如下结果：

```
111111110101010111110011
```

如果将它转换为16进制数，则得到的是0xFF55F3。当然，你永远不会看到位中的0和1的处理过程。你只需要键入下面的代码：

```
var color24:Number = 0xFF << 16 | 0x55 << 8 | 0xF3;
```

www.TopSage.com

或者，如果使用的是10进制数值，你可以使用下面的公式：

87
```
var color24:Number = 255 << 16 | 85 << 8 | 243;
```

另外，Flash不会在在意你使用的是10进制还是16进制。

你可以做一些类似的工作：把红、绿和蓝的值分别转换为16进制字符串的工作，然后把它们连接成一个长的字符串，然后把它们转换回一个数。但是如果你曾经试过，你将会知道你进入了一个复杂而又缓慢的字符串逻辑。相反，位运算符可能是ActionScript中最快的运算符了，因为它们工作在一种低级别的状态。

回到32位数值上，它是完全相同的概念。你只需要让你的8位alpha通道向左移动24位。例如，如果我们有一个32位颜色值为0xFFFF55F3，左移alpha值24位，我们得到如下：

```
11111111111111111010101011110011
```

第1个8位描述的是透明度级别。剩下的是红、绿和蓝的数值，和以前的一样。

4.1.5　提取组成色

最后，你也可能需要取一个颜色值并从中提取出各种组成色。让我们用相反的方法来从0xFFFF55F3提取出红、绿和蓝色值。下面是你工具箱中的另一个公式，同样，先看24位色：

```
red = color24 >> 16;
green = color24 >> 8 & 0xFF;
blue = color24 & 0xFF;
```

让我们一步一步来。首先，你可能已经猜到了>>这个符号是右移运算符，它会将若干位向右移动许多位。移动过程中不可以有断开的位。因此向右移动的所有超出范围的位都将被放弃。那么，以红色开始：

```
111111110101010111110011
```

当将颜色值向右移动16位后，结果如下：

```
11111111
```

或者简单地写为0xFF，或255。

对于绿色，向右移动8位，得到如下：

```
1111111101010101
```

这里已经把蓝色值挤了出去，但是仍保留有红色值。这里就需要使用另一种位运算符：&（与），也称为AND。就像OR（或）一样，它比较两个数值，但是要说成，“如果这个数AND这个数都是1，那么结果就是1。如是其中的一个是0，那么结果就是0。”将它与0xFF相比较，因此要比较下面的两组数：

88
```
1111111101010101
0000000011111111
```

因为所有的红色位都是与0作比较，所以它们的结果都将是0。只有两个对应数字都是1才会有1出现，因此你得到如下结果：

```
0000000001010101
```

www.TopSage.com

对于蓝色，你不需要移动任何内容，只需将它与0xFF做AND运算，它将保留蓝色而去掉红色和绿色。

对于32位色，可以使用相同的技术，但是需要一点改动：

```
alpha = color32 >> 24;
red = color32 >> 16 & 0xFF;
green = color32 >> 8 & 0xFF;
blue = color32 & 0xFF;
```

这里，当得到alpha值时，将它向右移动24位。仍需要将它向右移动16位以得到红色，但是需要将其与0xFF做AND运算，目的是为了去掉alpha值。绿色和蓝色两行与上面的情况相同。

现在已经知道了比预想的还要多的Flash颜色知识了。让我们开始使用它们。

4.2 drawing API

现在我要讲一下什么是API。它代表的是应用程序接口。通常，API是指一系列的方法和属性，你可以在程序中使用它来访问一系列相关的行为和属性。drawing API是指允许你通过ActionScript绘制线条、曲线、填充、渐变填充等等的属性和方法。API没有什么惊人的方法，但是有许多关于它们内容需要知道，而且有许多你可以使用的技巧。

drawing API是在Flash MX中被引入的。在那之前，所有的图形内容都需要在创建环境中绘制。使用drawing API，更多的事情都变得可能。

> 因为早期Flash版本的绘画限制，人们想出了一种灵活的动态绘画的方法。最好的一个例子是45度线的影片剪辑。它是由库中的一个单独的影片剪辑组成，设置它为导出以便可以在运行时间贴加到或放到舞台上，这样就可以复制它。它包含一个从0,0到100,100绘制而成的简单线条。在运行时，要在任意两点之间画一条线，你需要创建这个线条剪辑的一个新的实例，把它放在第1个点，并且进行缩放以使它可以适合第1个点和第2个点。虽然这种技术有一些缺陷，例如不允许填充或设置不同宽度的线条，但是它是非常有效的，而且在出现Flash MX之前，几乎所有的动态绘制使用的都是这种方法。

89

从Flash MX开始，你得到下面的方法：

- clear()
- lineStyle(width, color, alpha)
- moveTo(x, y)
- lineTo(x, y)
- curveTo(x1, y1, x2, y2)
- beginFill(color, alpha)
- endFill()
- beginGradientFill(fillType, colors, alphas, ratios, matrix)

在Flash 8 中，lineStyle和beginGradinetFill方法有了一些改进。这个版本也引入了beginBitmapFill和lineGradientStyle方法。AS 3增加了几个非常有用的方法：

www.TopSage.com

大家网 TopSage.com

- drawCircle(x,y,radius)
- drawEllipse(x,y,width,height)
- drawRect(x,y,width,height)
- drawRoundRect(x,y,width,height,ellipseWidth,ellipseHeight)

这只是一个快速浏览，我不打算详细讲解每一个方法，但我将涉及基础内容，你可以在需要额外内容的时候查看相关的文档。

4.2.1 图形对象

在早期的Flash版本中，drawing API方法是MovieClip类的方法，可以直接从影片剪辑的实例上调用，像这样：

```
myMovieClip.lineTo(100, 100);
```

现在，精灵和影片剪辑都可以访问drawing API。但是它们有一点点不同。现在，精灵和影片剪辑都有一个属性称为graphics，它用来访问所有的drawing API方法。因此，不能使用刚才的例子中直接访问drawing API的方式，而是使用下面的方法：

```
mySprite.graphics.lineTo(100, 100);
```

现在让我们看一些方法和一些它们基本应用的示例。

4.2.2 使用 clear 移除绘画

clear方法是所有当中最简单的方法。它只是从精灵或影片剪辑中移除之前绘制的线条、曲线或填充。注意它将只影响使用其他drawing API方法绘制的图形。换个角度讲，如果你在创作环境中在舞台上绘制了一个图形，在运行时调用clear()方法是不会对图形有所影响的。

90

clear方法的使用在绘制能力方面有一个重要而且有点意想不到的效果。在drawing API中，在一个指定的影片剪辑中绘制得越多，速度就会越慢。对于绘制几十个图形，变慢是不会被察觉的，但是慢慢地，每绘制一个新的图形会用越来越长的时间。即使是新的图形覆盖掉之前的其他所有已绘制的东西，但那些旧的图形的向量信息仍会保留在那里，并且每次都将会重新绘制。这点对于动画来说很重要，它会使你在那里不断地重新绘制几乎一样的内容。而使用clear方法可以彻底地从影片剪辑中移除之前绘制的所有图形的向量信息。

4.2.3 使用 lineStyle 设定线条样式

使用lineStyle (width,color,alpha) 方法，你可以设置随后绘制的所有线条的呈现样式。这个命令不会影响到之前绘制的线条。实际上，不同于清除它们或在它们的上面绘制些别的什么，没有办法影响或改变已经绘制好的线条或填充。一旦它们已经在那里了，就没有别的办法。

我列出来的参数是你最常用的。还有一些其他的可选参数，包括像素提示（pixel hinting）、缩放模式（scale mode）、末端端点类型（cap）、拐角连接类型（joint）和切断尖角限制位置（mitres）。如果你需要调整这些内容，它们对你来说都是可用的，但是大多数时候，你需要的可能只是我列出的那些参数。这些参数名本身就是非常好的解释，但还是让我们快速地浏览一下：

www.TopSage.com

- width：线条宽度的像素值。有效值只能是0或正值。虽然你可以传入小数数值，它们将舍入到最接近的正整数。如果你传入0或一个负值，Flash将绘制一个没有缩放的1像素宽的线条。这相当于你在Flash IDE的Properties面板中的线条选项中选择了“hairline”。
- color：你想要的线条颜色。它是我们在本章前面讲过的24位的10进制或16进制数值。
- alpha：线条的透明度。它的值可以是从0.0到1.0，描述的是不透明性的百分比。设置为1.0表示完全不透明，设置为0.0表示完全透明或不可见。注意它不同于AS 2版本，在AS 2版本中它的值的范围是从0到100。

同样，这些参数都是可选的。你可以用lineStyle(1)得到一个1像素宽的、黑色的线条，其他的所有参数使用的都是默认值。实际上，第1个参数也是可选的。如果你不设置width参数，只是调用lineStyle()，那么线条的样式将被清除掉，你会得到一条不可见的线条。这与你没有设置线条样式就发出绘制命令出现的问题是一样的：你不会看到任何线条。

另外一个经常困扰人们的事实是clear方法不仅清除当前的绘制，也会清除掉当前的线条样式设置。如果你设置了一个自定义的样式，在一个影片剪辑中绘制了一些线条，然后清除它们，你将需要在绘制其他内容之前重新设置线条的样式。否则，你又会发生绘制不可见线条的问题。调用clear方法还需要移动绘制的起始点到0,0位置。读完下面的内容你将会更加明白。

4.2.4 使用 lineTo 和 moveTo 绘制线条

在图形语言中有两种不同方式实现线条绘制。一个是使用带有一个起始点和一个结束点的line命令，并在它们之间绘制一条线。另一种方法是使用只带有一个点——线条的结束点的lineTo命令。这就是ActionScript的工作方式。那么，如果你正在绘制一个到某一点的线条，你是从哪开始绘制的呢？

91

好，如果你现在还没有绘制任何东西，你将从要在其中绘制的图形对象的0,0点开始绘制。假设在你的影片中你输入这行代码：

```
lineTo(100, 100);
```

你将会看到从左上角(0,0)到100,100的一条线（假设你已经设置了线条样式）。当你绘制完至少一条线以后，这条线的结束点将成为下一条线的起始点。最后，你可以使用moveTo方法来为下一条线指定一个新的起始点。

想象drawing API为一个拿着钢笔的机器人。当你开始时，钢笔在0,0位置。当你告诉它绘制一条线到一个点时，它只是在纸上移动钢笔到一个新的点上。每画一条新线，它只是从刚才的位置移到下一个新的点上。moveTo方法像是在说：“好，现在举起钢笔并把它移动到下一个点上。”虽然发出一个moveTo命令不会产生任何新的图形内容，但它将会影响下一个图形命令的执行方式。除非你想从0,0点绘制第1条线，否则，通常你都是要先调用moveTo命令来放置drawing API“钢笔”到你想要开始的位置上。

你现在已经学会了足够的命令来真正做一些事情了。让我们创建一个简单的绘制程序。这个程序完全依赖它自身的drawing API。下面是文档类：

www.TopSage.com

```
package
{
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class DrawingApp extends Sprite
    {
        public function DrawingApp()
        {
            init();
        }

        private function init():void
        {
            graphics.lineStyle(1);
            stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
            stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
        }

        private function onMouseDown(event:MouseEvent):void
        {
            graphics.moveTo(mouseX, mouseY);
            stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
        }

        private function onMouseUp(event:MouseEvent):void
        {
92          stage.removeEventListener(MouseEvent.MOUSE_MOVE,
                                      onMouseMove);
        }

        private function onMouseMove(event:MouseEvent):void
        {
            graphics.lineTo(mouseX, mouseY);
        }
    }
}
```

你可以在本书的可下载文件（可以在www.friendsofed.com下载）中找到DrawingApp.as，即该文件。我将一步步地介绍。除了绘制函数之外，这个程序还很好地利用了事件处理器（第2章讨论的内容）应用的一个很好的例子。

这个类是在第2章的一个类架构的基础上编写的，现在你应当可以很熟练地使用类架构了。它首先是导入MouseEvent类，因为它将用于任何需要使用鼠标的地方。在init方法中，线条样式被设置为一个像素的黑色线条并且加入了mouseDown和mouseUp事件侦听器。

那么对于onMouseDown方法。当用户按下鼠标时它将被调用。通常，这意味着用户想要从当前的鼠标位置开始绘制线条。好，这就是该函数所要做的。它通过调用moveTo方法和传入鼠标坐标来将虚拟钢笔放置在当前鼠标的位置。随即加一个侦听mouseMove事件的侦听器。

在这一点上，每当用户移动鼠标，onMouseMove方法将会被调用。那么这一步是做什么呢？做得不多，尽管它是此程序的重点。它只是在当前的鼠标位置上绘制线条。

www.TopSage.com

最后，有一个onMouseUp方法。它移除mouseMove处理器以致不再有绘制过程发生。

好，在这里你已经试验了一个小的绘制程序。你也许不需要费太多力气就可以设置一些简单的控制来使它成为一个特性完整的绘制程序。只需要为color和width创建一些变量，然后创建一些按钮或其他的控制来改变它们，再使用新的值来调用lineStyle方法。对了，再放入一个用来调用clear方法的按钮。如果你有兴趣，我将把它作为一个练习留给你自己做。同时，我将深入到其他的drawing API中。

4.2.5 使用 curveTo 绘制曲线

下一个绘制函数curveTo（x1,y1,x2,y2）的开始过程与lineTo相同，它的绘制开始于上一次绘制的结束点、一个moveTo命令将钢笔所移到的位置或0,0点（如果还没有任何绘制）。它也按照与lineTo一样的方法使用lineStyle。唯一不同的是所绘制的线条的形状。

如你所看到的，curveTo使用两个点。第1个点是影响曲线形状的控制点。第2个点是曲线的结束点。使用的公式是称为二次贝赛尔曲线的标准公式。这个公式计算从一点到另一点的曲线，它会朝向控制点弯曲。注意曲线将不会碰到（touch）控制点。它更像曲线被吸在控制点上。 93

下面让我们看它是如何工作的，下面是文档类DrawingCurves.as：

```
package
{
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class DrawingCurves extends Sprite
    {
        private var x0:Number = 100;
        private var y0:Number = 200;
        private var x1:Number;
        private var y1:Number;
        private var x2:Number = 300;
        private var y2:Number = 200;

        public function DrawingCurves()
        {
            init();
        }

        private function init():void
        {
            stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
        }

        private function onMouseMove(event:MouseEvent):void
        {
            x1 = mouseX;
            y1 = mouseY;
            graphics.clear();
            graphics.lineStyle(1);
            graphics.moveTo(x0, y0);
```

www.TopSage.com

```
            graphics.curveTo(x1, y1, x2, y2);
        }
    }
}
```

测试文件并到处移动你的鼠标。这个文件使用了两个预调点作为开始和结束点，使用鼠标的位置作为控制点。注意曲线永远不会到达控制点的位置，但是已经很接近了。

1. 曲线穿过控制点

现在，假设你真地想让曲线碰到控制点。好，这里有另外一个工具箱提示。你可以使用下面的公式计算真正的控制点以使曲线穿过你指定的点。另外，你将以x0,y0作为起始点，以x2,y2作为结束点，而x1,y1作为控制点。我将你想让曲线穿过的点称为xt,yt（t代表目标）。从另一个角度讲，如果你想要曲线穿过xt,yt，那么你需要使用吗x1,y1干什么呢？下面是这个公式：

94

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

基本上，你只是把目标点乘以2，然后减去起始点和结束点的平均值。可以把它绘制出来并且看它到底是如何工作的。或者只是测试它，看它的工作过程，以确信它总能工作。

要看它的工作过程，将鼠标的坐标作为xt,yt使用，只需要改变上面的文档类中的两行，将下面的代码

```
x1 = mouseX;
y1 = mouseY;
```

改为如下样子：

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

或者可以直接查看CurveThroughPoint.as文件，我已经做好它了。

2. 创建多曲线

你可能已经看到了下一个方向是创建多曲线，而不是单一曲线，你有一条很长的而且向多个方向平滑弯曲的线。首先，我将示范错误的创建方法，这种方法我最初时尝试过了。基本上，你开始有许多点，从第1点穿过第2点到第3点绘制曲线，然后再穿过第4点到第5点，然后再穿过第6点到第7点，等等。下面是代码（在文档类MultiCurve1.as中）：

```
package
{
    import flash.display.Sprite;

    public class MultiCurves1 extends Sprite
    {
        private var numPoints:uint = 9;

        public function MultiCurves1()
        {
            init();
        }

        private function init():void
```

www.TopSage.com

TopSage.com

```
        {
            // 首先建立由随机点构成的一个数组。
            var points:Array = new Array();
            for (var i:int = 0; i < numPoints; i++)
            {
                points[i] = new Object();
                points[i].x = Math.random() * stage.stageHeight;
                points[i].y = Math.random() * stage.stageHeight;
            }
            graphics.lineStyle(1);

            // 现在移到第1个点。
            graphics.moveTo(points[0].x, points[0].y);

            // 并循环每一个紧邻的后续对。
            for (i = 1; i < numPoints; i += 2)
            {
                graphics.curveTo(points[i].x, points[i].y,
                                 points[i + 1].x, points[i + 1].y);
            }
        }
    }
}
```

95

首先在init方法中的for循环只设置了9个点的数组，每个点是一个包含x和y属性的对象，并且它们随机摆放在舞台上。当然，在一个真实的程序中，你的点可能不是随机点，我使用这种方式只是为了更快一些。

然后设置线条的样式，虚拟钢笔被移动到了第1个点上。下一个for循环从1开始且每次增加2，因此它可以从第1点到第2点绘制曲线，然后穿过第3点到第4点，穿过第 5 点到第 6 点，然后再穿过第7点绘制到第8点。循环会停止在那里，因为第8点是最后一个点。你可能也会注意到至少要有3个点，并且点的个数一定是奇数。

当你测试时它总体看上去还不错。如图4-1所示，它看上去没有足够的曲度。实际上，它经常看上去有些尖锐。问题出在一条曲线与下一条曲线之间不协调，只是共享一个点。

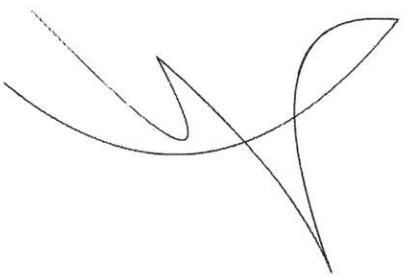


图4-1 多曲线，错误的方法。你可以很明显地看出一条曲线的结束位置和另一条曲线的开始位置

你将再插入一些点以使它正确工作。下面讲一下方法：在每两个点之间，你需要一个新的点准确地位于它们中间。然后使用它们作为每条曲线的起始和结束点，并使用原来的点来作为控制点。

96

www.TopSage.com

图4-2表明了其解决方法。在图中，白色的点是原来的点，黑色的点是插入的点。这里有3个curveTo方法，在这些方法中我已经给出了不同的颜色以使你可以看出它们的起始和结束位置。（图4-2实际上是文件multicurvedemo.fla的屏幕截图，你可以从www.friendsofed.com上下载它。）

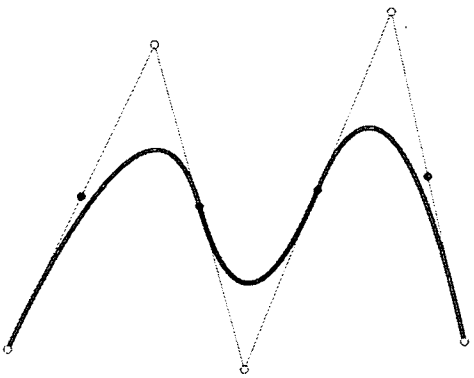


图4-2　带有中间点的多曲线

注意图4-2中的第1个和最后1个原来的点没有被使用，这两点仍是曲线的端点。你实际上只需要在第2个点和倒数第2个点之间插入点。下面是前面的随机曲线的例子的一个更新版本，即文档类MultiCurve2.as：

```
package
{
    import flash.display.Sprite;

    public class MultiCurves2 extends Sprite
    {
        private var numPoints:uint = 9;

        public function MultiCurves2()
        {
            init();
        }

        private function init():void
        {
            // 首先建立由随机点构成的一个数组。
            var points:Array = new Array();
            for (var i:int = 0; i < numPoints; i++)
            {
                points[i] = new Object();
                points[i].x = Math.random() * stage.stageHeight;
                points[i].y = Math.random() * stage.stageHeight;
            }
            graphics.lineStyle(1);

            // 现在移到第1个点。
            graphics.moveTo(points[0].x, points[0].y);
```


www.TopSage.com

```
            // 穿过其余点绘制曲线，在每个中间点停止。
            for (i = 1; i < numPoints - 2; i ++)
            {
                var xc:Number = (points[i].x + points[i + 1].x) / 2;
                var yc:Number = (points[i].y + points[i + 1].y) / 2;
                graphics.curveTo(points[i].x, points[i].y, xc, yc);
            }
            // 穿过最后两个点绘制曲线。
            graphics.curveTo(points[i].x, points[i].y, points[i+1].x,
                             points[i+1].y);
        }
    }
}
```

注意在新代码中，for循环是从1开始的，结束于points.length-2。这样可以阻止绘制过程中处理第1个点和最后1个点。它要做的是创建新的*x*,*y*点，该点是指数组中接下来两个点的平均值。然后它从数组中的下一个点到那个新的平均位置点绘制曲线。当循环结束时，索引i仍指向倒数第2个点。因此，你可以绘制一条穿过它到最后1个点的曲线。

这次将得到一个非常平滑的曲线形状，而不是有尖角的曲线，如图4-3所示。还需要注意你不再受到原来奇数个点的限制，只要以至少3个点开始就可以了。

对上面的话题做一点变动，下面的代码，MultiCurve3.as文档类，使用同样的技术创建了一个闭合曲线。基本上，它计算了一个初始的中间点，它是第1个点和最后1个点的平均值，然后移到那上面。然后再循环其他的所有点，计算每一个中间点，最后将最后那条曲线绘制到初始的中间点。

```
package
{
    import flash.display.Sprite;

    public class MultiCurves3 extends Sprite
    {
      private var numPoints:uint = 9;

      public function MultiCurves3()
      {
          init();
      }

      private function init():void
      {
          var points:Array = new Array();
          for (var i:int = 0; i < numPoints; i++)
          {
              points[i] = new Object();
              points[i].x = Math.random() * stage.stageHeight;
              points[i].y = Math.random() * stage.stageHeight;
          }

          // 找到第1个中间点并移到那儿。
          var xc1:Number = (points[0].x + points[numPoints - 1].x) / 2;
```

www.TopSage.com

```
            var yc1:Number = (points[0].y + points[numPoints - 1].y) / 2;
            graphics.lineStyle(1);
            graphics.moveTo(xc1, yc1);

            // 穿过其余点绘制曲线，在中间点停止。
            for (i = 0; i < numPoints - 1; i ++)
            {
                var xc:Number = (points[i].x + points[i + 1].x) / 2;
                var yc:Number = (points[i].y + points[i + 1].y) / 2;
                graphics.curveTo(points[i].x, points[i].y, xc, yc);
            }

            // 穿过最后一个点绘制曲线，回到第1个中间点。
            graphics.curveTo(points[i].x, points[i].y, xc1, yc1);
        }
    }
}
```

图4-4显示了其结果。

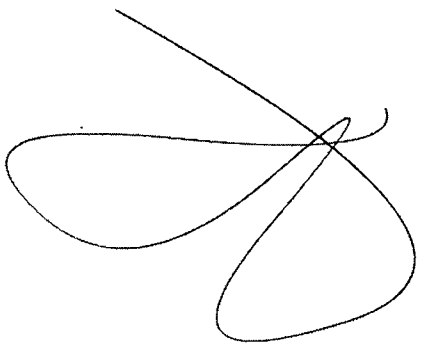


图4-3　光滑的多曲线

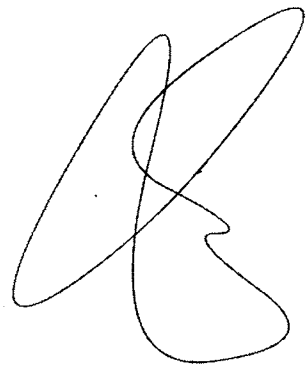


图4-4　多闭合曲线

4.2.6　使用 beginFill 和 endFill 创建图形

beginFill（color,alpha）方法非常简单，所以没什么可讲的。但是有一件事情值得注意，就是像lineStyle一样，alpha值现在被指定为从0.0到1.0之间的值，而不是0到100。它也是可选的，默认值为1.0。你在绘制过程中几乎可以随时发出beginFill命令。Flash将计算从现在的绘制位置开始，到发出endFill命令时的位置结束的一个图形，或者，如果从来没有发出过endFill命令，那么帧上代码结束之前的所有帧将被渲染。但是，通常情况下，顺序是这样的：

- moveTo；
- lineStyle（如果有）；
- beginFill；
- 一系列的lineTo和curveTo方法，移动到起始点；
- endFill。

www.TopSage.com

实际上，你可以以任意顺序使用前3个方法，而不会对绘制结果产生任何影响。你不需要指定线条的样式。记住如果你没有指定线条的样式，你将得到一个不可见的线条，它对于绘制填充是很有效的。还有不错的一点是绘制与填充是同时进行的。

如果你绘制的线条没有返回到原来的起点，为了在你调用endFill命令后马上关闭图形Flash将为你绘制它。最终绘制的线条将使用你调用endFill时存在的线条样式。然而，一个非常好的习惯是要注意结束你的图形。即使Adobe有一天决定改变指定的绘制行为，你也要确保绘制了最后的线条，而且要确保当其他人看你的代码时，他会知道你确实是打算绘制那个图形。

下面我们来试一下绘制填充。你可以继续使用上一个闭合曲线的例子（MultiCurves3.as），因为它已经生成了一个闭合图形。只要在第1个curveTo之前放置一个beginFill语句——如beginFill(0xff00ff);，它将创建一个亮紫色的填充——然后使用endFill();结束填充。

98
100

如果你这样做了，我确保你注意到了这些曲线自身相交的部分，它们的填充不见了。对于这种情况你也无能为力。但大多数情况，你所绘制的图形不会出现相交的情况。从另一个角度看，一旦你习惯了它的工作方式，你可以使用这种行为创建一些有趣的图形。

4.2.7 使用 beginGradientFill 创建渐变填充

现在我们来见识一下drawing API中的不同寻常的函数：beginGradientFill(fillType,colors,alphas,rations,matrix)。像lineStyle一样，这个方法有一些附加的可选的参数——spreadMethod、interpolationMethod和focalPointRatio。这些参数都可以用来调节渐变的属性，但是在大多数简单的情况下，它们并不是必需的并且我们不会在这里讨论。从多种意义上来讲，它的工作方式与beginFill相同，它有着相同的怪癖和行为，你总是可以使用相同的老命令endFill来结束它。比较大的不同是填充完成后的样子。我感觉我没必要说beginGradientFill创建的是一个渐变填充，但如果我不说我会觉得遗漏了什么东西。现在我们继续。

渐变填充基本是一种至少需要两种颜色的填充。图形的一部分以一种颜色开始，然后慢慢过渡到另一种颜色，中间可能经过了一种或几种其他预定义的颜色。

1. 指定填充类型

你可以创建两种渐变填充：线性和放射状。在线性填充中，渐变沿着一条线从一点到另一点。默认情况下，线性渐变是从左到右，但是你可以将它改为从上到下或是任何的角度。图4-5显示了几个线性填充的例子。

要看到一个线性填充，你至少需要两种不同的颜色。如果它们都是你指定好的，填充将从第1种颜色融合过渡到第2种颜色。如果你指定了更多的颜色，填充将光滑地从第1种颜色到第2种颜色，再到第3种颜色，如此等等直到最后一种颜色。

放射状填充使用线性填充使用的所有参数工作，但是对它们的解释有一点点不同，它是创建一个从你定义的空间中心开始然后向所有的外围方向辐射的渐变，创建一个圆或椭圆。你指定的第1个颜色将用做圆的中心，最后的颜色用在圆的外围。唯一不需要的是角度。图4-6显示了一个放射状填充的例子。

www.TopSage.com

大家网 TopSage.com



图4-5 线性填充



图4-6 放射状填充

那么，对于beginGradientFill(fillType,colors,alphas,ratios,matrix)方法，第1个参数是fillType。它非常简单。它是一个字符串，可以包含值“liner”或“radial”中的一个。你就得到了所求的。类似于我们第3章讲过的事件类型，填充类型现在是flash.display.GradientType类的静态属性，要避免任何拼写错误。因此，不要键入字符串，你可以（而且应该）
101 导入GradientType类然后键入GradientType.LINER或GradientType.RADIAL。

2. 设置颜色、透明度和比率

依据颜色参数——要指定颜色，当然——你必须也要通过指定0～255的一个值来确定每种颜色填充的位置，0是填充的开始位置，255为结束位置。这些数，每一种颜色，被称为填充中的ratios。因此，如果你想制作一个两种颜色的填充，你可能要指定0和255作为ratios。要制作3种颜色的融合，你可以指定0、128、255。这样会将第2种颜色精确地放在另外两种颜色的中间。如果你指定0、20、255，第1种颜色会立即过渡到第2种颜色，然后非常缓慢地过渡到第3种颜色。记住这些值不是像素值，而是对255的分割。

你也必须要指定每种颜色在融合过程中的透明度。它被称为alpha并且它的值是0.0～1.0。还要注意的是它不同于AS 2中的0～100的范围。如果你不想要任何透明，你可以指定它们的alpha都为1.0。如果你设置第1个颜色的alpha为1.0而最后一个颜色的alpha为0.0，渐变将不仅融合颜色，还会光滑地淡出。这一点对于制作柔和的阴影（尽管你可能有更好的运气使用投影滤镜）非常有用。

这里的每一个参数都是一个数组。这是很明显的，因为你可能至少要传入两种颜色、透明度和比率，还有可能更多。你可以创建一个新的数组并使用每个参数的值来对它进行赋值，像这样：

```
var colors:Array = new Array();
colors[0] = 0xffffff;
colors[1] = 0x000000;
```

但是有更容易的方法来完成它。你可以通过将你想要放在其中的元素列在中括号中来一次性地创建并赋值一个数组，元素之间使用逗号隔开：

```
var colors:Array = [0xffffff, 0x000000];
```

事实上，你甚至不需要创建colors变量。你可以直接将数组作为一个参数插入到beginGradientFill语句中。因此，你会得到下面的代码：

```
graphics.beginGradientFill(GradientType.LINEAR,
                           [0xffffff, 0x000000],
                           [1, 1],
                           [0, 255],
                           matrix);
```

www.TopSage.com

它定义了两种颜色、两个alpha值（两个都是100%）和开始与结束的比率值，所以渐变将从白色开始并最后过渡到黑色。当然，如果你想你可以为每个参数创建变量，并且如果你要定义许多颜色，这样会使代码看起来更清晰。有一个需要注意的地方是这3个数组必须都包含相同数量的元素。如果你定义了3种颜色，那么你必须要3个alpha和3个比率。如果任何一个比其他的多或者少，它将会失败——没有渐变，没有填充，没有错误提示。

现在你所需要的是开始和结束位置，还有就是填充的角度。你可能已经猜到了，那就是神秘的matrix参数所代表的含义。 102

3. 创建矩阵

矩阵是一个包含行和列数值的二维网格，它用来计算各种值矩阵可以扩展应用在图形的旋转、缩放和移动物体上。这正是你在这里使用渐变所要做的。你需要设定它的位置，定义它的大小，还有尽可能旋转它。要使用矩阵，你需要创建一个矩阵对象。它是flash.geom.Matrix类的一个实例，它本质上是保存了一个Flash可以用来设定位置、设置大小和旋转你的渐变的数据网格。（实际上，Matrix类不只是用于控制渐变填充，但这里我们只用它来处理渐变。更多关于matrix矩阵的应用请参看第18章。）

虽然Matrix类的使用会比较复杂，但是有一种专门用来为渐变填充创建所需要的矩阵类型的方法。它称为createGradientBox。它的参数至少包括你想创建渐变的宽和高，旋转和开始的*x*、*y*位置是可选的。首先你创建一个Matrix类实例，然后调用它的createGradientBox方法来自动设定它的内部数值。下面显示其代码。

```
var matrix:Matrix = new Matrix();
matrix.createGradientBox(width, height, rotation, x, y);
```

如果你要使用flash.geom.Matrix类，不要忘记在你的类的上方先导入它。

如果只是指定了一个宽和高，后面的3个参数值将默认为0。

让我们看它的工作过程。下面是文档类GradientFill.as：

```
package
{
    import flash.display.GradientType;
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import flash.geom.Matrix;

    public class GradientFill extends Sprite
    {
        public function GradientFill()
        {
            init();
        }

        private function init():void
        {
            graphics.lineStyle(1);
            var colors:Array = [0xffffff, 0xff0000];
            var alphas:Array = [1, 1];
```

www.TopSage.com

```
            var ratios:Array = [0, 255];
            var matrix:Matrix = new Matrix();
            matrix.createGradientBox(100, 100, 0, 0, 0);
            graphics.beginGradientFill(GradientType.LINEAR, colors,
                                       alphas, ratios, matrix);
            graphics.drawRect(0, 0, 100, 100);
            graphics.endFill();
        }
    }
}
```

103

当测试时，你将会看到一个漂亮的白到红的渐变方块。现在让我们绘制一个不同位置的方块，将绘制代码改为如下代码（在GradientFill2.as中）：

```
graphics.drawRect(100, 100, 100, 100);
```

现在你的方块已经全部是红色了。发生了什么呢？好，渐变开始于*x*轴上的0，并且它只有100像素宽，因此，在你的方块的开始位置，渐变就已经是纯红色了，并且这之外的颜色都是红色。那么，你需要让矩阵的*x*,*y*在你的方块的左上角，如下所示：

```
matrix.createGradientBox(100, 100, 0, 100, 100);
```

这里，*x*,*y*与方块的起始位置相同。使用同样的方块，你可以试验矩阵和渐变填充设置来观察可能创建的各种效果。首先，试一下3种颜色：

```
var colors:Array = [0xffffff, 0x0000ff, 0xff0000];
var alphas:Array = [1, 1, 1];
var ratios:Array = [0, 128, 255];
```

不要忘记了alpha和比率。移动中间的比率值来看一下它对渐变的影响。使用64或220来替换128。

下一个例子只是alpha的融合。颜色是一样的，只是alpha改变了。

```
var colors:Array = [0x000000, 0x000000];
var alphas:Array = [1, 0];
var ratios:Array = [0, 255];
```

试着改变一下角度。现在我们使用45度角：

```
matrix.createGradientBox(100, 100, Math.PI / 4, 100, 100);
```

使用Math.PI/2将通过把它旋转90度制作一个垂直渐变。-Math.PI/2会使它向上，而Math.PI将使它从右到左而不是从左到右。

最后，切换成放射状类型：

```
beginGradientFill(GradientType.RADIAL, colors, alphas, ratios, matrix);
```

104

现在在放射状版本中试验线性渐变填充中使用的所有相同的技巧。

4.3　颜色变换

接下来是你的渲染工具库中的flash.geom.ColorTransform类和它的方法。不像drawing API，它们不允许你创建图形，只允许你改变存在于精灵或其他显示对象实例中的图形的颜色。让我深

www.TopSage.com

入地学习一下它们是如何工作的。

使用 ColorTansform 类改变颜色

在ActionScript 2.0中，最常用的的控制影片剪辑的颜色的方法是使用Color类，该类有setRGB和setTransform方法。ColorTransform类是在Flash 8时被引入的并且不提倡再使用Color类，但是我想大多数人会继续使用Color类，因为他们已经习惯使用它了。好，Color类并不是AS 3的一部分，因此现在我们集中精力学习新的方法。ColorTransform的工作方法本质上与Color类是一样的，并且它们是做同样的事情，只是语法有些不同。

首先，你需要知道精灵、影片剪辑或任何其他的显示对象都有一个属性称为transform。它是flash.geom.Transform类的一个实例，包含了你可以用来进行缩放、旋转、定位和对影片剪辑应用颜色的各种属性。这里用到的属性是colorTransform。意识到它是一个显示对象属性的属性，因此要像这样访问它：

```
mySprite.transform.colorTransform
```

或者，在一个扩展Sprite的类中，你可以只引用类自身的transform属性：

```
transform.colorTransform
```

要改变一个对象的颜色，你需要分配一个ColorTransform对象给这个属性。这时，你可能要问，是一个ColorTransform对象吗？好，如果你用过AS 2的color类中的方法setTransform，你就知道要传给它一个具有可以告诉它如何改变颜色的各种属性的对象。好，ColorTransform与它是非常相似的，但它不是一个普通的对象，它现在有自己正式的类。在AS 2中，你可能像这样操作：

```
myTransform = new Object();
myTransform.ra = 50;
myTransform.ga = 40;
myTransform.ba = 12;
myTransform.aa = 40;
myTransform.rb = 244;
myTransform.gb = 112;
myTransform.bb = 90;
myTransform.ab = 70;
```

在AS 3中，你需要像这样创建同样的ColorTransform对象：

```
myTransform = new ColorTransform(0.5, 0.4, 0.12, 0.4, 244, 112, 90, 70);
```

105

前4个值是增量，后面4个是偏移值。我一会儿会列出它使用的公式，并且你会知道为什么它使用那样的名称。而且，你可能注意到了增量的百分比值是从–1.0到1.0范围内的小数，而不是–100到100。实际上，在Flash文档中说的是0.0到1.0，但你仍可以使用负的增量来做一些有趣的效果（一会你会看到）。偏移值仍是从–255到255。ColorTransform对象的真正的构造函数像这样：

```
ColorTransform(redMultiplier,
               greenMultiplier,
               blueMultiplier,
```

www.TopSage.com

```
                alphaMultiplier,
                redOffset,
                greenOffset,
                blueOffset,
                alphaOffset)
```

变换一个特定的颜色通道的公式是这样的，以使用红色通道为例：

```
newRed = oldRed * redMultiplier + redOffset;
```

当使用ColorTransform时，记住它存在于flash.geom包中，因此你需要先导入此类。

现在给出一个示例来看它们是如何工作的，下面是文档类TansformColor.as，该例在SWF中嵌入了一幅图像来作为位图（一个Bitmap类的实例）。因为Bitmap类是一个显示对象，它有一个transform属性。代码中设定了位图的transform属性的colorTransform属性，使用这些设置来产生一幅反相效果的图像。你可以任意改变增量和偏移值来感觉一下它们是如何工作的。

```
package
{
    import flash.display.Bitmap;
    import flash.display.Sprite;
    import flash.geom.ColorTransform;

    public class TransformColor extends Sprite
    {
        [Embed(source="picture.jpg")]
        public var Picture:Class;

        public function TransformColor()
        {
            init();
        }

        private function init():void
        {
            var pic:Bitmap = new Picture();
            addChild(pic);
            pic.transform.colorTransform = new ColorTransform(-1, -1,
                                                              -1, 1,
                                                              255, 255,
                                                              255, 0);
        }
    }
}
```

106

如果你在亲自试验它，只需要更改下面这行

```
[Embed(source="picture.jpg")]
```

来匹配你正使用的图的路径。如果你使用的是Flash IDE，只需要导入图像到你的库中，并导出为ActionScript，然后输入类名Picture。不管哪种方式，不用担心关于嵌入内容的部分。我将在4.5节讲到它。最重要的是colorTransform的设置。

www.TopSage.com

TopSage.com

4.4 滤镜

滤镜（filter）是可以应用于任何显示对象的位图效果。它可以在Flash IDE中通过Filters面板使用或是在运行时通过ActionScritp来使用。因为本书是关于ActionScript的，所以我将只讨论应用滤镜的脚本方法。下面是AS 3中所包含的滤镜：

- 投影（Drop shadow）；
- 模糊（Blur）；
- 发光（Glow）；
- 浮雕（Bevel）；
- 渐变浮雕（Gradient bevel）；
- 渐变发光（Gradient glow）；
- 颜色矩阵（Color matrix）；
- 卷积滤镜（Convolution）；
- 置换贴图（Displacement map）。

我不会详细地介绍每一个滤镜。它们在Flash的帮助文档都有详细的说明，如果要讲解它们可能要用一整章（如果不是整本书）。但是因为这本书中已经涉及了足够多的内容，所以我将只给出一些使用介绍和两个示例。

4.4.1 创建滤镜

滤镜是通过使用new操作符和滤镜名称并传入需要的参数来创建的。例如，要创建一个模糊滤镜（最简单的滤镜之一），你可以这样：

```
var blur:BlurFilter = new BlurFilter(5, 5, 3);
```

107

在代码中，参数是blurX、blurY和quality。这个例子将创建*x*和*y*轴上5个像素、质量中等的模糊。

另一个需要意识到的事情是滤镜都在它们自己的包中，称为flash.filters。因此，你可以把它放在文件的开头：

```
import flash.filters.BlurFilter;
```

如果你想导入包中的所有滤镜，可以使用下面的快捷方式：

```
import flash.filters.*;
```

现在，你可以直接创建任何类型的滤镜。但是通常，除非你要使用包中的所有或大部分类，否则最好避免使用通配符*而是显式地列出你想要导入的类。这会使得你的代码更清晰，所以你需要准确地知道你要导入哪些类和不导入哪些类。

那么现在你已经有了一个模糊滤镜，但是如何将它应用在一个特定的对象上呢？

任何显示对象都有一个属性称为filters。它是一个包含着已经应用于对象的所有滤镜的数组，因为任何对象都可以应用多个滤镜。你只需要将你的模糊滤镜加入到那个数组中。理想情况

www.TopSage.com

下，这只是应用一个基本的数组操作push，如mySprite.filters.push(blur);。但它并不是那么简单。Flash不能侦听到filters数组的改变并且不能通知滤镜做相应的改变直到有一个完整的新数组指定给filters。

如果你知道对象还没有应用任何滤镜，或你想覆盖它们，你只需要创建一个崭新的数组，直接将模糊滤镜放在其中，并把这个新的数组指定给filters属性。让我们试一下。下面是一个文档类Filters.as，它创建了一个新的精灵并在其中绘制了一个黄色的方块。然后创建了一个滤镜，并将其加入一个数组，然后把这个数组指定给精灵的filters属性：

```
package
{
    import flash.display.Sprite;
    import flash.filters.BlurFilter;

    public class Filters extends Sprite
    {
        public function Filters()
        {
            init();
        }

        private function init():void
        {
            var sprite:Sprite = new Sprite();
            sprite.graphics.lineStyle(2);
            sprite.graphics.beginFill(0xffff00);
            sprite.graphics.drawRect(100, 100, 100, 100);
            sprite.graphics.endFill()
            addChild(sprite);
            var blur:BlurFilter = new BlurFilter(5, 5, 3);
            var filters:Array = new Array();
            filters.push(blur);
            sprite.filters = filters;
        }
    }
}
```

108

瞧，你已经有了一个黄色的模糊的盒子。最重要的部分是粗体部分。你可以把这一段简化为：

```
var blur:BlurFilter = new BlurFilter(5, 5, 3);
var filters:Array = [blur];
sprite.filters = filters;
```

或者再简化一些：

```
sprite.filters = [new BlurFilter(5, 5, 3)];
```

只要你创建了一个新的数组，将滤镜放在其中，然后把它指定给filters属性。Flash将会很顺利地工作。

但是如果你已经应用了一个滤镜并且仍想保留它，或更坏一点的情况，你不确定是否还要有其他的滤镜？在Flash 8中这会比较复杂，因为如果一个显示对象没有应用滤镜，它的filters属

www.TopSage.com

性将是未定义的。但是在AS 3中，filters数组总是可以创建为一个空的数组。所有你要做的只是获取这个数组的引用，将滤镜放在其中，并将它指定给对象的filters属性。下面是它的代码：

```
var filters:Array = sprite.filters;
filters.push(new BlurFilter(5, 5, 3));
sprite.filters = filters;
```

使用这种方法，是否已经应用了滤镜都没有关系，你的滤镜将只是加入到列表当中。因为filters属性是一个功能齐全的数组，使用各种不同的数组方法完成同一件事可能有多种方法。例如，下面是一个使用concat方法的例子：

```
sprite.filters = sprite.filters.concat(new BlurFilter(5, 5, 3));
```

我认为并没有真正“正确”的方法来做它。只要你创建了一个指定给你的filters属性的包含滤镜的数组，你就是好样的。

4.4.2 动画滤镜

现在你已经知道如何使用ActionScript创建并使用滤镜，让我们根据你所学的来制作一个投影滤镜动画的应用。要制作这个效果，使用下面的文档类AnimatedFilters.as：

109

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.filters.DropShadowFilter;

    public class AnimatedFilters extends Sprite
    {
        private var filter:DropShadowFilter;
        private var sprite:Sprite;

        public function AnimatedFilters()
        {
            init();
        }

        private function init():void
        {
            sprite = new Sprite();
            sprite.graphics.lineStyle(2);
            sprite.graphics.beginFill(0xffff00);
            sprite.graphics.drawRect(-50, -50, 100, 100);
            sprite.graphics.endFill();
            sprite.x = 200;
            sprite.y = 200;
            addChild(sprite);

            filter = new DropShadowFilter(0, 0, 0, 1, 20, 20, .3);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
```

www.TopSage.com

```
        private function onEnterFrame(event:Event):void
        {
            var dx:Number = mouseX - sprite.x;
            var dy:Number = mouseY - sprite.y;

            filter.distance = -Math.sqrt(dx * dx + dy * dy) / 10;
            filter.angle = Math.atan2(dy, dx) * 180 / Math.PI;
            sprite.filters = [filter];
        }
    }
}
```

这个类首先在一个精灵中绘制了一个方块使这个方块位于精灵的中心，然后将精灵移到舞台
110 上。使用一些默认属性值来创建投影滤镜。

一个侦听enterFrame事件和处理器的侦听器被加入。onEnterFrame方法使用三角学计算鼠标与精灵之间的距离和角度，关于三角学我们在第3章讨论过。它使用这些来设定投影滤镜的angle和distance，并将该滤镜应用到精灵。注意你不需要每次都创建一个新的滤镜。你可以继续使用同一个，只是改变它的属性。然而，只改变这些属性不会使精灵的可视显示更新。要使改变生效，你需要重新指定滤镜给filters属性。

4.5 位图

与滤镜相似，我可以使用一整本书书告诉你关于Ditmap和DitmapData类的所有内容。这将很有意思，但它并不是本书的目的。我将通过几个例子来介绍它的基础内容并指出从AS 2到AS 3处理位图的一些变化。

和在AS 2中一样，你首先需要使用下面的参数调用new BitmapData()来创建一个新的BitmapData对象：

```
new BitmapData (width:Number,
                height:Number,
                transparent:Boolean,
                fillColor:Number)
```

现在你可能已经猜到了，BitmapData也包含在一个嵌套的包中。它的完整修饰名为flash.display.BitmapData。因此，你需要先导入它的包。至于它的参数，width和height都是显而易见的。Transparent意味着你所创建的图像可以包含一个alpha通道来表示透明度，你可以选择true或false。fillColor是图像创建的初始颜色。如果透明度为真，位图将使用32位色创建，因此你需要使用0xAARRGGBB的颜色格式指定颜色。如果透明度为假，你可以使用安全的24位色。

当你创建一个BitmapData对象时，你可能想使用某种方式让其可见。在AS 2中，你将使用attachBitmap方法将位图加入到一个影片剪辑当中。你可能会想到使用aadChild方法将位图加入到显示对象中。好，也是也不是

问题是addChild方法只能工作在扩展了DisplayObject类的对象上，如精灵、影片剪辑和文本域。然而，如果深究这个类的结构，你将会发现BitmapData并没有扩展DisplayObject，因此

www.TopSage.com

它不能直接加入。这时就需要引入Bitmap类了。Bitmap类几乎只有一个功能——做为BitmapData实例的一个容器。你可以按如下方式创建：

```
var myBitmapData:BitmapData = new BitmapData(100, 100, false, 0xff0000);
var myBitmap:Bitmap = new Bitmap(myBitmapData);
```

非常重要的是Bitmap类继承自DisplayObject，因此现在你可以将这个对象加入显示列表中了：

```
addChild(myBitmap);
```

111

并且使它可见。Bitmap实例也可以用来对位图定位、缩放和加入滤镜等操作。

要测试它，只需要在第2章给出的类架构中的init方法中加入上面的3行代码。不要忘记导入flash.display.Bitmap和flash.display.BitmapData。当你运行它时，你将看到一个红色的方块。乍一看，这好像并不会比你使用drawing API绘制的相同的东西更有意思。但是意识到它并不是使用红色填充绘制的向量图形。它是一个位图图像，它的每一个像素都是分开定义的且是可以改变的。

事实上，每一个像素值都可以通过getPixel、getPixel32、setPixel和setPixel32来读取或改变。两个版本的不同之处在于getPixel和setPixel使用了24位数值而忽略了alpha通道，而“32”版本使用了32位数值来包括透明度信息。让我们进一步来试验一下并且制作一个像你在任何位图软件中看到的基本的喷笔工具。

这里是文档类SprayPaint.as：

```
package
{
    import flash.display.Sprite;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.MouseEvent;
    import flash.events.Event;
    import flash.filters.BlurFilter;

    public class SprayPaint extends Sprite
    {
        private var canvas:BitmapData;
        private var color:uint;
        private var size:Number = 50;
        private var density:Number = 50;

        public function SprayPaint()
        {
            init();
        }

        private function init():void
        {
            canvas = new BitmapData(stage.stageWidth,
                                    stage.stageHeight,
                                    true, 0x00000000);
            var bmp:Bitmap = new Bitmap(canvas);
            addChild(bmp);
```

www.TopSage.com

大家网 TopSage.com

```
            stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
            stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
112     }
        private function onMouseDown(event:MouseEvent):void
        {
            color = Math.random() * 0xffffff + 0xff000000;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onMouseUp(event:MouseEvent):void
        {
            removeEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            for(var i:int = 0; i < density; i++)
            {
                var angle:Number = Math.random() * Math.PI * 2;
                var radius:Number = Math.random() * size;
                var xpos:Number = mouseX + Math.cos(angle) * radius;
                var ypos:Number = mouseY + Math.sin(angle) * radius;
                canvas.setPixel32(xpos, ypos, color);
            }
        }
    }
}
```

这可能是我到目前为止给出的最复杂的代码，但除了BitmapData之外，其他的内容在之前都已经涉及过。我只是在这一次用得多一些。尽管如此，让我们来一步一步地分析。

首先，它创建了一些类的变量，包括用来保存BitmapData实例的canvas变量。接着创建实例自身，设定了在屏幕上的大小，并且使用透明的背景。然后，使用canvas创建一个位图，然后将其加入显示列表当中。

鼠标事件处理器只是选择一个随机颜色并且加入和移除enterFrame事件的处理器，所有的动作都是在enterFrame事件中执行的。这里，我们又用上了一点三角函数。首先，你要计算一个从0到Math.PI*2的随机角度，需要记住的是Math.PI*2现在是弧度值，等同于360度。接下来，你要计算一个随机的半径，并使用三角函数将radius和angle转化为x、y值。然后使用setPixel32设置鼠标位置处的像素值加上喷色的随机的x、y值，每次你开始绘制时它都是随机确定的。

事实上，你将看到这个例子中有一个for循环，这在每一帧上都要发生许多次。要发生多少次取决于density的值。

还有一处是颜色的计算是使用随机的24位色，然后0xFF000000会与它相加以使alpha通道完全不透明。如果你没有加入额外的值，颜色将完全透明。如果你用Math.random()与0xFFFFFFFF相乘，你也会得到随机的透明颜色，那可能是你想要的，但不是我想要的。

继续测试它，可以通过改变density和size值来观看所创建的不同的效果。你可能会想如何
113 加入一些控制以使用户可以改变这些参数。

www.TopSage.com

乍一看，你可能会想，“好主意，我可以使用drawing API或是通过贴加一些影片剪辑和变换它们的颜色来做相同的事。”没错，你可以那样做，但是如果你已经使用drawing API绘制了成百上千的单独的图形，你将会发现你绘制得越多，它运行得就越慢。当几百个图形绘制完后，迟延会变得非常明显并且程序变得不可用了。对于贴加影片剪辑也是同样的。但是对于位图有很大的不同，当然，你可以用一整天时间使用这个程序绘制，它的速度和效果是不会改变的。

如果你想看一些更酷一点的效果，可以将下面的代码加入到文件当中，刚好在创建Bitmap对象bmp之后加入：

```
bmp.filters = [new BlurFilter(2, 2, 3)];
```

它加入了一个模糊到内容中并且它使得你正在处理的位图更加明显，而不是向量图。

当然，设置像素是你对BitmapData对象进行的一个最简单的操作之一。除了读取和设置像素之外，你可以对BitmapData对象应用另外的大约24种方法。使用它们，你可以复制像素、设置阈值（thresholds）、像素化（dissolve）、合并（merge）、滚动（scroll），等等。我个人最喜欢的一个是柏林噪声（PerlinNoise）函数，它允许你创建随机的、有机的图案。对于从烟和云到地形和水波的任何东西的制作都非常有用。使用这些方法进行实验会大大增加你的兴趣。

4.6 载入或嵌入内容

最后一个渲染图形的重要内容是获取外部内容至你的影片，如位图或外部的SWF。关于此有两种方法。一种是当影片在运行时获取外部内容，这被称为载入（loading）。另一种方式是当SWF在编译时将外部内容嵌入SWF。

4.6.1 载入内容

要载入一个外部内容，你需要创建一个Loader对象。它是flash.display.Loader类的一个实例。Loader是一个显示对象，意味着你可以使用addChild方法将它加入到显示列表中，就像一个sprite或bitmap。然后你就可以告诉Loader要载入外部SWF还是位图，如JPEG、PNG等等。

这里有一个小技巧。在AS 2中，当你要处理外部文件或URL的路径时，你只需要使用一个简单的字符串作为路径。在AS 3中，你需要创建一个flash.net.URLLoader的实例，将路径以字符串的形式传入。它是一个额外的步骤，而且有点儿讨厌，但是你需要习惯于使用它。

下面是一个在运行时载入外部内容的例子（文档类为LoadAsset.as）：

```
package
{
    import flash.display.Sprite;
    import flash.display.Loader;
    import flash.net.URLRequest;
    public class LoadAsset extends Sprite
    {
        public function LoadAsset()
        {
            init();
```


www.TopSage.com

```
        }

        private function init():void
        {
            var loader:Loader = new Loader();
            addChild(loader);
            loader.load(new URLRequest("picture.jpg"));
        }
    }
}
```

4.6.2 嵌入内容

尽管在某些情况下运行时载入内容非常好，但很多时候你想将某个外部的图形内容直接包含在SWF中。如果你用的是Flash IDE，你可以简单地将它导入到库中并且导出为ActionScript。但是如果你使用的是Flex Builder 2或是Flex 2 SDK和命令行编译器，它们没有库。那么你怎样将外部内容包含到SWF中呢？

答案是使用[Embed]元数据标签嵌入它们。*元数据*（metadata）是指包含到ActionScript文件中但并不是ActionScript的语句。而且，它们会在编译过程中通知编译器做某些事情。对于[Embed]标签，它会告诉编译器包含一个特定的外部内容到最终的SWF文件中。这个外部内容可以是一个位图或一个外部的SWF。你可以通过source属性来告诉编译器哪些内容是需要嵌入的，如下：

```
[Embed(source="picture.jpg")]
```

在元数据这一行的后面，你需要声明一个类型为Class的变量，如下：

```
[Embed(source="picture.jpg")]
private var Image:Class;
```

现在你就可以使用这个变量来创建外部内容的新实例，如下：

```
var img:Bitmap = new Image();
```

注意你所创建的对象的类型为Bitmap。如果你已经嵌入了一个外部的SWF，那么所创建的对象类型为Sprite，如下：

```
[Embed(source="animation.swf")]
private var Anim:Class;
var anim:Sprite = new Anim();
```

115

这里有一个嵌入外部的JPEG图像到一个SWF中的例子：

```
package
{
    import flash.display.Sprite;
    import flash.display.Bitmap;

    public class EmbedAsset extends Sprite
    {
        [Embed(source="picture.jpg")]
        private var Image:Class;

        public function EmbedAsset()
```

www.TopSage.com

```
        {
            init();
        }

        private function init():void
        {
            var img:Bitmap = new Image();
            addChild(img);
        }
    }
}
```

另外，如果你正在使用Flash IDE，你只需要将它导入到库中，导出为ActionScirpt，并给它一个类名。你不需要使用[Embed]标签和Class变量。实际上，Flash IDE所使用的编译器不支持[Embed]标签。

我只是简单介绍了一下这部分内容，因为在本书中我们将不会进一步使用该技术。然而，在这里提到它很显然是很有用的。

4.7 本章重点公式

在本章中你已经收集了一些更有用的工具，多数是用于处理颜色。

转换16进制为10进制：

```
trace(hexValue);
```

转换10进制为16进制：

```
trace(decimalValue.toString(16));
```

116

组合组成色：

```
color24 = red << 16 | green << 8 | blue;
color32 = alpha << 24 | red << 16 | green << 8 | blue;
```

提取组成色：

```
red = color24 >> 16;
green = color24 >> 8 & 0xFF;
blue = color24 & 0xFF;

alpha = color32 >> 24;
red = color32 >> 16 & 0xFF;
green = color32 >> 8 & 0xFF;
blue = color232 & 0xFF;
```

通过一个点绘制曲线：

```
// xt, yt 是你想要穿过的点，
// x0, y0 和 x2, y2 是曲线的端点。
x1 = xt * 2 - (x0 + x2) / 2;
y1 = yt * 2 - (y0 + y2) / 2;
moveTo(x0, y0);
curveTo(x1, y1, x2, y2);
```

www.TopSage.com

4.8　小结

这一章没有涉及太多让物体移动的内容，但它确实向你展示了大量创建可视内容的方法，你将在后面的章节中学会如何动画。具体来说本章包含了下面的内容：

- 颜色，24位和32位。
- drawing API。
- 滤镜。
- Bitmap和BitmapData类。

这些主题给出了当你需要为你的动画生成动态的可表达的内容时而需要的工具，因为这里涉及的内容都是基于ActionScript的，你可以使用这里的所有方法直接制作动画。只需要使用一些代码创建内容，改变代码中的变量，然后再渲染它们。

你在本书中将用到本章中所介绍的很多技术，因此很好地理解它们是很有用的。实际上，在
117 下一章中，你将使用本章的一些技术来做第1个有用的实验，它将涉及速度和加速度。

www.TopSage.com

大家网
TopSage.com

Part 2 第二部分

基本运动

本部分内容

- 第 5 章　速度和加速度
- 第 6 章　边界和摩擦力
- 第 7 章　用户交互：移动物体

www.TopSage.com

第5章

速度和加速度

本章内容

- 速度
- 加速度
- 本章重点公式

好，恭喜！你已到了本书真正的要点部分。这意味着：(a)你已经耐着性子读完了前面的所有章节；(b)你已经浏览了前面的章节并感觉到你已经知道了足够多的内容；(c)你已经厌倦了并且是跳跃式前进的。然而，碰巧你在这里。记住，如果你遇到问题，可能要回到前面的章节寻求帮助。

这一章首先从基本的运动开始：速度、向量和加速度。这些概念几乎要在你从这以后所做的所有的ActionScript动画中用到。

5.1 速度

正在移动的物体都有一个非常基本的属性就是速度（velocity）。很多人将术语速度等同于速率（speed）。速率是速度的一部分，但是只是一部分。速度的概念还包含了一个非常重要的因素：方向。下面是一个非常外行的速度定义：特定方向上的速率。让我们来看一下这个定义与简单的速率究竟有什么不同。

如果我告诉你我在位置*X*上了车并且以每小时30公里的速率行驶了一个小时，你找我会很困难。但是，如果我说我以相同的速率向北行驶了一小时，你就能够准确地知道我在哪里。这在动画中是相当重要的，因为你需要知道将你的精灵放在哪儿。说精灵以某个速率移动是可以的，但你需要在每一帧将某个具体的*x*,*y*坐标。这样就引出了速度的概念。如果你知道某一帧开始时某个物体的位置，它移动得有多快，向哪个方向，你就可以知道在下一帧开始时将它放在哪儿。

它的速率部分通常定义为多少像素每帧。换一种说法，如果一个影片剪辑进入帧时位于某个位置点，它从离开那个点到结束帧会走过这么多像素。

在大部分情况下，使用像素每帧会工作得很好，而且它确实是最易于编程的。然而，在Flash中由于帧频的不稳定性，当有太多的事情发生时，使用像素每帧可能会使你的动画在某个点上变慢，这可能是因为要计算的东西太多，或CPU正在忙于其他某些事情。如果你正在编写某种需要

www.TopSage.com

动画是以严格的恒定速率前进的游戏或仿真，你可能要使用基于间隔的动画。我在第19章包含了关于这种方法的一个示例。

在开始速度编码之前，我想先来说一下向量，因为它是速度的一般描述方式。

5.1.1 向量和速度

向量（vector）是有大小和方向的某个东西。在速度中，大小是指速率。向量是以箭头的形式绘制的。箭头的长度就是它的大小，箭头的方向很自然是向量的方向。图5-1显示了一些向量。 122

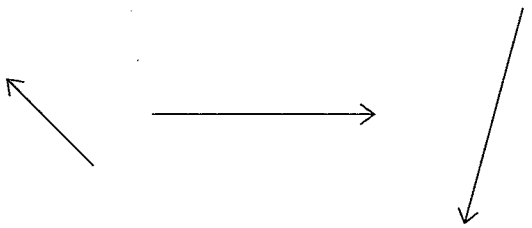


图5-1 一些向量

需要注意的一件事是向量的大小总是正值。一个大小为负值的向量只是表示向量是相反方向的，如图5-2所示。

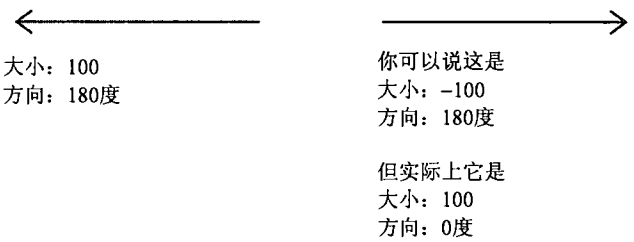


图5-2 负的速度实际上是反方向的速度

另外一件需要注意的事情是向量并没有真正的位置。甚至在速度这种情况中，向量没有陈述某个物体从哪里开始或在哪里结束。它只指出了物体移动得有多快和向哪个方向移动。因此，如果两个向量有同样的大小和方向，那么它们是相等的，即使它们描述的是在两个不同的位置的两个不同的物体，如图5-3所示。

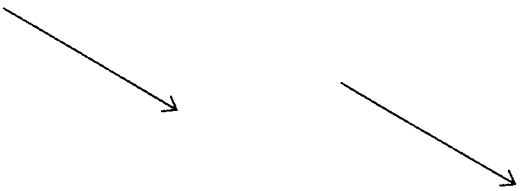


图5-3 如果向量有相同的大小和方向，则它们是相同的，与位置没有关系 123

www.TopSage.com

5.1.2　一个轴上的速度

我打算先把事情简化，把速度限制在一个轴上：*x*轴或水平运动。这基本上是在说方向是0度、向东或从屏幕的左边到屏幕的右边——不管你多么想看到它。速率只是它在那个方向上每帧移动的像素数。因此，如果我说*x*轴上的速度是5，意思是说物体将从左向右每帧移动5个像素。这也意味着如果我说*x*轴上的速度是–5，它将从右向从左移动，每帧移动5个像素。

现在，你可以理解我的意思了，不是吗？我们已经讨论过关于负的大小的问题，当时我只是告诉你没有这种事情。从技术角度讲，速度实际上是5并且方向是180度。类似地，一个正的*y*速度是指90度（垂直向下）的速率，而一个负的*y*速度将是270度或负的90度（垂直向上）。然而，在实际应用中，当你计算一组*x*、*y*速度时，它们总是表示成正值和负值，而且你经常会看到我这样写，“*x*速度是–5。”所以，如果它能帮助你，把负号想成是指示*x*轴速度“向左”或*y*轴速度“向上”，而并不是意味着负值。

在整个书中，我将使用*vx*代表*x*轴上的速度，*vy*代表*y*轴上的速度。正的*vx*意味着向右，负的*vx*意味着向左。正的*vy*意味着向下，负的*vy*意味着向上。

对于一个轴上的速度，你只需要将速度加到对象在那个轴上的位置上。不管*vx*是什么，在每一帧都需要将它加到对象的*x*属性上。

下面让我们看一下它是如何工作的。在这一章（包括后面的）的许多例子都将需要某种对象用来移动。为了避免用光宝贵的空间在每一个类和例子中都绘制这样一个对象，我们将创建一个我们可以并且将频繁重用的Ball类。下面为代码：

```
package {
    import flash.display.Sprite;

    public class Ball extends Sprite {
        public var radius:Number;
        private var color:uint;

        public function Ball(radius:Number=40, color:uint=0xff0000) {
            this.radius = radius;
            this.color = color;
            init();
        }
        public function init():void {
            graphics.beginFill(color);
            graphics.drawCircle(0, 0, radius);
            graphics.endFill();
        }
    }
}
```


现在，任何时候只要我引用这个类，你就应当将这个类放在你的项目中，而且只需要使用new Ball(size,color)语句就可以创建一个新的球。或者，可以将一个类放在一个固定的位置，并把其目录加到你的类路径中，就像我们在第2章中提到的那样。

现在你已经有东西可以用来移动了，下面是第1个速度示例——文档类Velocity1.as：

www.TopSage.com

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Velocity1 extends Sprite
    {
        private var ball:Ball;
        private var vx:Number = 5;

        public function Velocity1()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.x = 50;
            ball.y = 100;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            ball.x += vx;
        }
    }
}
```

在这个示例中，首先设置*x*速度(*vx*)为5。记住这是指每帧移动5个像素，所以在每一帧上，*vx*都要加到球的*x*属性上。init方法将球置于舞台上并且设置enterFrame事件的侦听器。在每一帧上，小球将放在离它上一帧所在位置向右5个像素的地方。试验一下。相当好的运动错觉，嗯？。

试验过程中，给*vx*一些更高或更低的值。试一些负的值并观察对象在反方向上移动。看看你是否可以让对象在*y*轴上移动。

125

5.1.3 两个轴上的速度

在两个轴向上移动非常简单。只需要定义一个*vx*和一个*vy*，然后在每一帧将*vx*加到*x*属性上并将*vy*加到*y*属性上。那么，所说的就是：对于每一帧，对象将在*x*轴上移动*vx*像素且会在*y*轴上移动*vy*像素。

下一个示例(Velocity2.as)演示的就是这个内容，下面是代码：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Velocity2 extends Sprite
    {
        private var ball:Ball;
```

www.TopSage.com

大家网 TopSage.com

```
        private var vx:Number = 5;
        private var vy:Number = 5;

        public function Velocity2()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.x = 50;
            ball.y = 100;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            ball.x += vx;
            ball.y += vy;
        }
    }
}
```

再一次，对速度变量进行试验直到你认为得到一个合适的值。不要忘记试验一些负值。

5.1.4 角速度

好，到目前为止，一切都好。你已经有一些真正的动画，使用的是两个轴上的速度。但是在
126 许多情况中——也许是大多数情况中——*x*和*y*速度并不是你的初始数据。

事实上这里我对速度的定义有点欺骗性。我把它说成是在一个方向上的速率，可是现在我已经给了你两个不同方向上的两个不同的速率。我之所以这么做是因为：在Flash中，你的对象是放在*x*,*y*坐标上的。所以你需要两个轴上的位置和速度。但是你不一定使用它们来开始。

要是每次定义时你只有一个速率值和一个方向角会怎么样呢？假设你想让某个物体以45度角方向和每帧3个像素的速率移动。在这个描述中我看不到任何的*vx*或*vy*，或任何类似的东西。

幸运的是，已经向你介绍了从那个描述中获取*vx*和*vy*的工具。回过头来想一想第3章讨论的三角学。现在，看图5-4，它显示了你想让小球在每帧做的动作：以45度角移动3个像素。

还熟悉这个图吗？加上一条线怎么样，如图5-5所示？现在你知道了什么？你有了一个直角三角形，具有一个定义好的角和斜边!

注意到三角形的两个直角边在*x*和*y*轴上。实际上，在*x*轴上的直角边就等于小球将在*x*轴上移动的距离。同理*y*轴上的直角边也是这样。记得在直角三角形中，只要有一条边和一个角，就可计算出其他所有的角和边。那么，给出了45度角和3个像素的斜边，可以使用Math.cos和Math.sin计算出*vx*和*vy*的长度。

与角相邻的边是*vx*，一个角的余弦是邻边/斜边。或另一种表示方法为，邻边是角的余弦值乘以斜边。同样，对边是*vy*。正弦是对边/斜边，或对边是正弦乘以斜边。

www.TopSage.com

下面是你将用到的准确的代码：

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

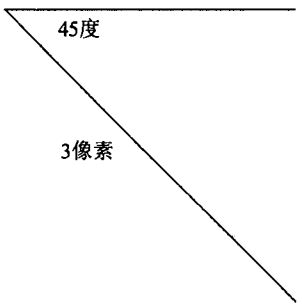


图5-4 大小和方向

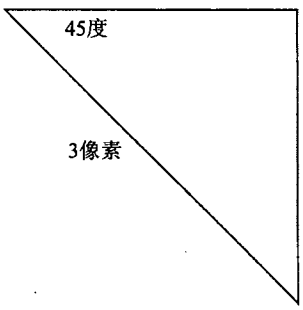


图5-5 大小和方向映射变为一个直角三角形

现在不要忘记在将角传到Math函数中之前先把45度转换为弧度！一旦有了*vx*和*vy*，你可以很容易地将这些加到动画中物体的*x*、*y*位置上。

下一个示例（VelocityAngle.as）具有如下代码：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    public class VelocityAngle extends Sprite
    {
        private var ball:Ball;
        private var angle:Number = 45;
        private var speed:Number = 3;

        public function VelocityAngle()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.x = 50;
            ball.y = 100;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            var radians:Number = angle * Math.PI / 2;
            var vx:Number = Math.cos(angle) * speed;
            var vy:Number = Math.sin(angle) * speed;
            ball.x += vx;
```


www.TopSage.com

```
            ball.y += vy;
        }
    }
}
```

这里主要的不同之处在于你是以angle和speed开始的而不是*vx*和*vy*。速度是作为局部变量来计算、使用和抛弃的。当然，在一个像这样的简单情况中，角和速率是永远不变的，很明显它计算了一次速度并将其保存为类变量。但在非常高级的运动中，运动的方向和速率都将不断变化，因此*vx*和*vy*将不会保持相同的值。

试验这个示例时，任意改变那个角。确保你可以使小球以任意的角和任意的速率运动，只通过改变那两个数值。

现在，让我们从向量角度看一下这个示例。

1. 向量加法

向量加法（vector addition）是在一个系统中当你有两个向量并且想得到它们的合成向量时才需要。这里，你有一个*x*轴上的向量、*y*轴上的另一个向量和一个总的速度向量。将向量相加只是简单地将它们的头尾相连。最终的合成向量是从向量链路中第一个向量的起点到最后一个向量的
128 终点所绘制出的连线。在图5-6中，你可以看到3个要相加的向量和它们最终的合成向量。

它与你所放置向量的顺序是无关的，结果总是相同的。时间并不是它的一部分。你可以说物体沿着这个方向移动，然后移到那个方向上，最后移到另一个方向上，以任何顺序均可。或者你可以说它同时在3个方向上移动。最终的结果看上去它是以某个速度在某个方向上移动的。

让我们回到我们的示例上。如果你放置了*x*轴速度向量，然后在它的结尾处放置*y*轴速度向量，最终的那条线就是合成速度。图5-7显示了作为向量的速度。

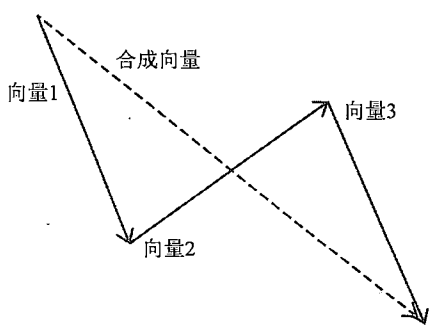


图5-6　向量加法

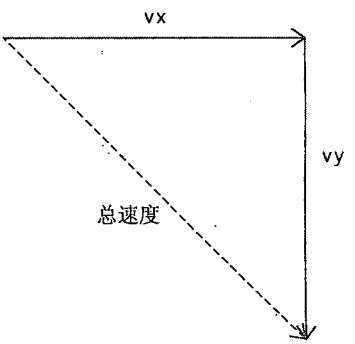


图5-7　速度作为向量

你了解到了什么？它们是相同的图片！它标志着你所做的是正确的。

2. 鼠标跟随

让我们使用速度概念来扩展一下前面的内容。在第3章，你构建过一个始终指向鼠标的箭头的示例。这个示例使用的是Math.atan2来计算鼠标与箭头之间的角，然后旋转箭头以使它与那个
129 角对齐。

www.TopSage.com

根据你刚学到的，你可以加入速率并且得到基于当前角的速度。这个示例使用相同的Arrow类，而不是小球，所以查找、下载或重写那个类并将它放在与下面的文档类FollowMouse.as相同的路径下：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class FollowMouse extends Sprite
    {
        private var arrow:Arrow;
        private var speed:Number = 5;

        public function FollowMouse()
        {
            init();
        }

        private function init():void
        {
            arrow = new Arrow();
            addChild(arrow);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            var dx:Number = mouseX - arrow.x;
            var dy:Number = mouseY - arrow.y;
            var angle:Number = Math.atan2(dy, dx);
            arrow.rotation = angle * 180 / Math.PI;
            var vx:Number = Math.cos(angle) * speed;
            var vy:Number = Math.sin(angle) * speed;
            arrow.x += vx;
            arrow.y += vy;
        }
    }
}
```

虽然这是一个颇为复杂的效果，但现在你应没有什么理解不了的地方了。获取到鼠标的x距离和y距离，然后通过这两个值使用Math.atan2来得到它们之间形成的角。再使用这个角来旋转箭头，并使用Math.cos和Math.sin结合速率来计算出x速度和y速度。最后，将速度加到位置上。

5.1.5 速度扩展

这里我正在进入一个危险的境地，但是我要开始把速度定义成它从未有过的意义。虽然从严格意义上讲速度是指位置的改变和空间中的物理运动，但没有必要把刚刚学到的概念只限制在显示对象的x和y属性上。 130

一个精灵、影片剪辑或其他的显示对象有许多你可以修改的属性，几乎它们都可以接受你可以随时间更改的一系列的值来生成动画。也许速度这个词对于它来说不够准确，但概念是相似的，

www.TopSage.com

因此当描述某个随时间改变某一属性的东西时，我经常继续使用速度这个词来作为变量名。

我们将制作一个精灵旋转的例子。在这种情况下，你在每一帧上通过加上一个数值来改变对象的rotation属性。你可以在每帧通过给ratation属性加上一个比较大的数值来使对象快速旋转，或加上一个比较小的值使它旋转得慢一些。不管恰当与否，我通常将保存旋转速率的变量记为*vr*，意思是旋转的速度。

使用我们熟悉的箭头影片剪辑，你可以像如下（RotationalVelocity.as）这样做：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class RotationalVelocity extends Sprite
    {
        private var arrow:Arrow;
        private var vr:Number = 5;

        public function RotationalVelocity()
        {
            init();
        }

        private function init():void
        {
            arrow = new Arrow();
            addChild(arrow);
            arrow.x = stage.stageWidth / 2;
            arrow.y = stage.stageHeight / 2;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            arrow.rotation += vr;
        }
    }
}
```

就速度而言，速率是5且方向是顺时针。我可能需要脱逃了，因为还有一些运动在继续，但是当我使用像“alpha速度”（是指某个对象淡入和淡出的速度）这样的术语时，我确信我会听到
131 一些指责的声音。然而，我发现使用类似这样的术语来考虑属性是很有用的。我想以某一个速率来增加或减小alpha。因为我发现自己经常要随着时间改变几个属性，且每个都是以某个速率来改变，所以通过将它们称之为速度是来把它们联系起来是很好的。因此，我可能会这样做：

```
arrow.x += vx;
arrow.y += vy;
arrow.alpha += vAlpha;
arrow.rotation += vr;
arrow.scaleX = arrow.scaleY += vScale;
// 等等。
```

你将在后面看到许多像这样的例子，因此我希望你能原谅我偶尔使用v字符。

www.TopSage.com

大家网 TopSage.com

关于速度和速率就讲这么多，现在我们讲述加速度。

5.2 加速度

通常我们会认为加速度是指速度加快，减速度是指速度减慢。然而这是不正确的。在本书中，我对加速度使用了一个比较专业的定义。

速度和加速度之间在描述方式上存在很多相似之处。它们都是向量。和速度一样，加速度被描述为一个数值和一个方向。然而，速度改变的是对象的位置，而加速度改变的却是它的速度。

按这种方式想一下。你进入车里，发动它并使它开始工作。你的速度是多少？是0。踩下油门（也称为是加速器……），速度开始改变（至少是它的速率部分，方向由方向盘来改变）。一两秒之后，你正在以每小时4或5英里（1英里=1.609千米）的速率（mph）前进。然后速率上升到10mph，20mph，30mph，等等。发动机正在给汽车施加力量来改变它的速度。

因此，你对加速度有了一个不专业的定义：*改变对象速度的力*。

以纯正的ActionScript术语来讲，你可以说加速度是加入到速度属性上的一个数值。

这里是另外一个例子。假设你有一个太空船需要从行星A到行星B。它的方向将是行星B正好与行星A相对的方向。它指向那个方向并点燃火箭。只要火箭在燃烧，力就会施加给太空船，它的速度将发生改变。换句话说，太空船将以持续的越来越快的速度飞行。

在某个点上，船长决定太空船正以足够快的速度飞行，他可能要保留一些燃油，因此关闭了火箭。假设在太空中没有摩擦力，太空船将会继续以同一速度飞行。因为火箭不再燃烧，所以不会再有力施加到太空船上。没有加速度，速度就不会改变。

然后太空船接近了它的目标。它需要慢下来或要飞到行星B的右边（或如果驾驶员足够准确，太空船将成为行星B的一部分）。那么船长会怎么做呢？在太空中你没有制动器——也没有什么可以抓住的东西。然而，船长会调转太空船头面向相反的方向，然后重新点燃火箭。这样就施加了 132
一个负的加速度或反方向的加速度。其实，这与我们讨论的向量是一回事。力再一次改变了速度，但这次它降低了速度。速度将会越来越慢，最终会减到0。想象一下，它将与太空船位于行星表面上2英寸（1英寸=0.025 4米）的位置一致。在那个位置，船长将关闭火箭，剩下的工作交给重力来完成。（我将在5.2.3节中讨论重力，很快就到了。）

5.2.1 一个轴上的加速度

让我们把刚刚学到的在Flash中实践一下。就像第一个速度示例，第 1 个加速度示例针对一个轴。我们在下面的几个例子中将会用到前面的Ball类。

这里是第一个示例（Acceleration1.as）的代码：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
```

www.TopSage.com

```
public class Acceleration1 extends Sprite
{
    private var ball:Ball;
    private var vx:Number = 0;
    private var ax:Number = .2;

    public function Acceleration1()
    {
        init();
    }

    private function init():void
    {
        ball = new Ball();
        addChild(ball);
        ball.x = 50;
        ball.y = 100;
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onEnterFrame(event:Event):void
    {
        vx += ax;
        ball.x += vx;
    }
}
}
```

这里，以速度(*vx*)为0开始。加速度(*ax*)是0.2。它在每一帧时加入到速度中，然后速度再加入
133 到小球的位置中。

测试这个示例。你将会看到小球开始时移动得很慢，但是到它离开舞台时，它一直是快速移动的。

现在，让我们接近太空船的例子，允许小球关闭和打开加速度，甚至将它反向。可以使用光标键来完成它。你知道如何侦听键盘事件，并且你可通过检测传给事件处理器的事件对象的keyCode属性来得到导致那个事件的键。然后将它与flash.ui.Keyboard类中发现的常数进行比较，它包括一些易读的键编码属性，如Keyboard.LEFT、Keyboard.SPACE、Keyboard.SHIFT等等。现在，我只关心左右光标键，它们是Keyboard.LEFT和Keyboard.RIGHT。我们可以使用他们来改变加速度。下面是代码（Acceleration2.as）：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
    import flash.ui.Keyboard;

    public class Acceleration2 extends Sprite
    {
        private var ball:Ball;
        private var vx:Number = 0;
        private var ax:Number = 0
```

www.TopSage.com

```
        public function Acceleration2()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.x = stage.stageWidth / 2;
            ball.y = stage.stageHeight / 2;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
            stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
        }

        private function onKeyDown(event:KeyboardEvent):void
        {
            if(event.keyCode == Keyboard.LEFT)
            {
                ax = -0.2;
            }
            else if(event.keyCode == Keyboard.RIGHT)
            {
                ax = 0.2;
            }
        }

        private function onKeyUp(event:KeyboardEvent):void
        {
            ax = 0;
        }

        private function onEnterFrame(event:Event):void
        {
            vx += ax;
            ball.x += vx;
        }
    }
}
```

134

在这个示例中，你只是检测左右光标键是否被按下。如果左光标键被按下，则设置*ax*为负值。如果右光标键被按下，则设置*ax*为正值。如果哪个键也没有按下，则设置它为0。在onEnterFrame方法中，像以前一样将速度加到位置上。

测试这个示例。你会发现你并不能完全控制这个对象移动的速度。也可以说你不能让它立刻停下来。你只能让它减速移动到要停的位置。如果减速太多，它就会开始移向相反的方向。看一下你是否能在它刚要碰到舞台边缘之前让它停下来。

我确信有一些游戏的想法已经在你的脑海中形成。保持住它们，你即将对这个技术进行多次扩展。

www.TopSage.com

5.2.2 两个轴上的加速度

同速度一样，你可以同时在*x*轴和*y*轴上都有加速度。你只需要为每个轴（我通常使用变量名*ax*和*ay*）设置一个加速度值，将它们加到*vx*和*vy*上，最后将*vx*和*vy*加到*x*和*y*属性上。

调整上一个例子使它也使用*y*轴工作是非常简单的。只需要增加下面的内容：

- *ay*和*vy*变量；
- 检测上下光标键；
- 将对应的加速度加到对应的速度上；
- 将速度加到相对应的轴位置上。

135

下面是代码(Acceleration3.as)：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
    import flash.ui.Keyboard;

    public class Acceleration3 extends Sprite
    {
        private var ball:Ball;
        private var vx:Number = 0;
        private var vy:Number = 0;
        private var ax:Number = 0
        private var ay:Number = 0

        public function Acceleration3()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.x = stage.stageWidth / 2;
            ball.y = stage.stageHeight / 2;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
            stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
        }

        private function onKeyDown(event:KeyboardEvent):void
        {
            switch(event.keyCode)
            {
                case Keyboard.LEFT :
                ax = -0.2;
                break;
```

136

www.TopSage.com

```
                case Keyboard.RIGHT :
                ax = 0.2;
                break;

                case Keyboard.UP :
                ay = -0.2;
                break;
                case Keyboard.DOWN :
                ay = 0.2;
                break;

                default :
                break;
            }
        }

        private function onKeyUp(event:KeyboardEvent):void
        {
            ax = 0;
            ay = 0;
        }

        private function onEnterFrame(event:Event):void
        {
            vx += ax;
            vy += ay;
            ball.x += vx;
            ball.y += vy;
        }
    }
}
```

注意我已经把左/右/上/下/键的检测移到了switch语句中，这样就不至于我们写太多的if条件语句。

测试一下上面的例子。你将会看到你可以导航屏幕上的所有方向。试着让物体移动起来，如从左到右，然后按上光标键。注意*x*方向的速度根本不会受到影响。物体会保存原有的*x*轴上的速率。你刚刚已经加入了某个*y*速度到图中。这相当于太空船调转90度和点燃火箭。

5.2.3 重力加速度

到目前为止，我一直都是从对象自身施加的一个力方面来讨论加速度的，如一辆汽车的引擎或一个火箭。这毫无疑问是理有据的，但是还有另一方面的加速度。碰巧的是任何改变物体速度的力都使用了加速度原理，包括重力、磁力、弹力、摩擦力等等。

你可以用两种不同的方式来看重力。从太阳系这个宽角度看，重力是两个天体之间的引力。计算每个天体的真实加速度时必须考虑两个天体之间的距离和夹角。

另一种看重力的方法是从一个比较精确的视点上看，这里是地球，或者是离它非常近。在我们每天的日常生活中，重力并不会根据我们离地球的远近而发生明显的改变。尽管从技术角度讲，当你在飞机或是山上时重力会轻微地变小，但你是不会觉察到的。因此当在这个基础上模拟重力

www.TopSage.com

大家网 TopSage.com

137 时，你可以定义它为一个定值，如同你在前面示例中定义加速变量一样。

还有，因为地球是如此地大而我们是如此地小，很容易忽视任何真实的加速度方向，而只是说它的方向是“向下”的。换句话说，你的物体在哪里没有关系，你可以安全地定义重力只为*y*轴上的加速度。

将它放到ActionScript中，所有你要做的只是定义一个重力值，然后将它加在每帧的*vy*上。通常情况下，使用小数值比较好，如0.5或再小一点。比这个值大很多时，物体会“感觉”太重。更小的值，物体会像羽毛一样好像要浮起来。当然，你也可以把这些效果当作是一种好处。例如，模拟有不同重力的行星。

下一示例中加入了重力。完整代码在Gravity.as中，但是我不打算完全把它列出来。它与Acceleration3.as只有几处不同。除了类和构造函数的名称改变外，我还在类的开始部分增加了一个变量到变量列表中：

```
private var gravity:Number = 0.1;
```

然后在onEnterFrame方法中加入了一行：

```
private function onEnterFrame(event:Event):void
{
    vx += ax;
    vy += ay;
    vy += gravity;
    ball.x += vx;
    ball.y += vy;
}
```

我设置重力为很小的值以使它不至于太快地向下离开屏幕，而且你仍可以通过光标键来控制它。你刚刚所创建的就是老的月球登陆游戏的所有物理原理了。加入一些好的图形和一些碰撞检测，你就可以自己玩了！（我将帮你完成后者，图像就由你自己来完成。）

注意首先要通过按下光标键加入加速度到*vx*和*vy*，然后加入重力到*vy*。在处理多个力的复杂系统时并不会像想象的那样复杂。只需要计算出每个力的加速度然后将它们附加给速度。没有复杂的求合力或力的分解要做。每一个力只是加上去。当你已经研究并处理完所有的力时，需要将速度加到位置上。这里，只用到了两个力，但在本书的后面，将会针对一个对象计算许多不同的力。

这刚好回到了向量加法上。如果你以最初的速度作为一个向量来开始，每一个加速度、重力或力都是附加到速度上的一个额外的向量。当你把所有的都加上去了，你只需要从开始位置到结束位置绘制一条线，就得到了结果的速度。你会发现它与你将每个力的*x*和*y*上的分力加起来的结果是相同的。

现在假设你的精灵是一个热气球。你可能想加入一个称为*升力*的力。这是*y*轴上的另外一个加速度。然而，这次它是负值，或说是方向“向上”的。现在，你有3个力作用在对象上：键盘力、重力和升力。如果你考虑一下，或是试验一下，你会发现升力需要比重力大一些才能保证气球是上升的。这是非常合理的——如果它们完全相等，则它们两者将互相抵消，那么它又会回到
138 老样子，只有键盘的力会有一些效果。

另外还有一点可以试验的是创建一些风。显然，它是*x*轴上的力。根据风吹的方向，它可能

www.TopSage.com

是一个正或负的力，或你现在已经知道的，0度或180度方向。

5.2.4 角加速度

如我已经提到的，加速度有一个数值——力——和一个方向，就像速度一样。而且与速度一样，如果你以这两个因素开始，你需要分解出合力中的*x*方向和*y*方向的力。现在，如果你很细心，你应当知道可以使用Math.cos和Math.sin来完成此操作。下面是实现代码：

```
var force:Number = 10;
var angle:Number = 45; // 度，需要转换!
var ax:Number = Math.cos(angle * Math.PI / 180) * force;
var ay:Number = Math.sin(angle * Math.PI / 180) * force;
```

现在你已经有了每一个轴上的加速度，你可以在每一个轴上更新速度，与此同时也会更新物体的位置。

现在让我们重新使用前面章节中的鼠标跟随，让它使用加速度工作而不只使用单纯的速度。记得在先前的制作过程中，你获得了箭头到鼠标的角并且使用这个角来确定*vx*和*vy*。这次，你将使用相同的计算方法，但确定的是*ax*和ay值而不是*vx*和*vy*。然后你将加速度的值加到速度值上，再将速度值加到*x*和*y*属性上。下面是代码（FollowMouse2.as）：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class FollowMouse2 extends Sprite
    {
        private var arrow:Arrow;
        private var vx:Number = 0;
        private var vy:Number = 0;
        private var force:Number = 0.5;

        public function FollowMouse2()
        {
            init();
        }

        private function init():void
        {
            arrow = new Arrow();
            addChild(arrow);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            var dx:Number = mouseX - arrow.x;
            var dy:Number = mouseY - arrow.y;
            var angle:Number = Math.atan2(dy, dx);
            arrow.rotation = angle * 180 / Math.PI;
            var ax:Number = Math.cos(angle) * force;
            var ay:Number = Math.sin(angle) * force;
```


www.TopSage.com

```
            vx += ax;
            vy += ay;
            arrow.x += vx;
            arrow.y += vy;
        }
    }
}
```

注意我们将原来的speed转变成了force并且让它变得更小。因为加速度是增加的，所以以一个较小的值开始。它增长得很快还需要注意的是现在vx和vy被声明为类的变量，这样使类中的任何方法都可以访问它们。之前，它们被用来计算每帧上的新的速度，但是现在你需要跟踪它们并且每次从它们的值中加或减数值。当然，你可以把这里的ax和ay变量都去掉而只把正弦和余弦行的结果直接加到速度上，我之所以分开写是为了让程序看上去更清晰。

现在，代码看上去不是太复杂，是吗？但是回头看一看你在本章开始时做的第1个运动示例，就会明白你已经走了多远了。第1个是很平淡的，它可能只是一个补间动画。通过学习两个基本原理，你现在已经可以创建非常流畅的动态的内容了感觉像真的一样的东西。而现在本章内容还没有完全结束呢！

好，让我们把所有的内容集合在一起，来看一下你可以深入到哪一层。

5.2.5　太空船

关于从这里行驶到那里的太空船我已经讨论很多了。根据目前为止所涉及的内容，你应该可以将它们集中在一起形成一个合理的太空船模拟。

这儿是一个计划。太空船自身将是一个只用来绘制自己的类，很像我们之前用到的Arrow类或Ball类。你可以使用左右键来向左或向右旋转它。上键将用来点燃火箭。当然，火箭在太空船的后面并且直接在后面点燃。因此，火箭产生的力将会在太空船当时面对的方向形成加速度。其实，你要做的大部分像是老的行星游戏（Asteroids）中的船，不包括真正的行星。

首先，你需要一个太空船，我在类中是使用drawing API代码绘制了4条短的白线，以尊重原创。如果你比我更有艺术天份，你可以直接在PhotoShop或Swift 3D中绘制你的太空船，将它保存为位图，然后使用我们在第4章中讨论过的嵌入外部位图的技术将它嵌入。它与我们现在使用代
140 码的工作情况是一样的。不管怎样，下面的类是我创建的，名为Ship.as。

```
package
{
    import flash.display.Sprite;

    public class Ship extends Sprite
    {
        public function Ship()
        {
            draw(false);
        }

        public function draw(showFlame:Boolean):void
        {
```

www.TopSage.com

```
            graphics.clear();
            graphics.lineStyle(1, 0xffffff);
            graphics.moveTo(10, 0);
            graphics.lineTo(-10, 10);
            graphics.lineTo(-5, 0);
            graphics.lineTo(-10, -10);
            graphics.lineTo(10, 0);

            if(showFlame)
            {
                graphics.moveTo(-7.5, -5)
                graphics.lineTo(-15, 0);
                graphics.lineTo(-7.5, 5);
            }
        }
    }
}
```

注意这里有一个公开的绘制方法，它带有一个真/假的值。这种方式，你绘制它时可以有火焰显示，也可以没有火焰显示。这对于显示太空船的引擎正在燃烧将是非常有用的。你可以在图5-8和图5-9中看到有火焰和没有火焰的样子。

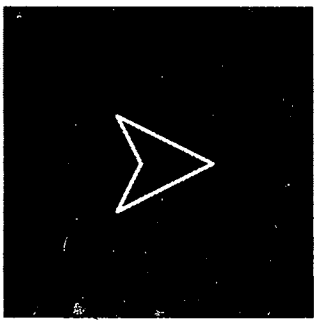


图5-8 看到太空旅行的未来

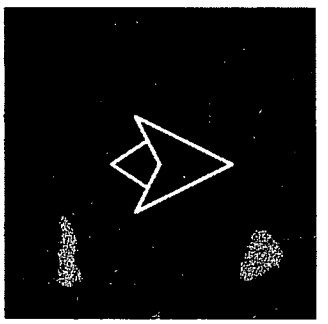


图5-9 注意那火焰

141

太空船控制

好，现在你已经创建了你的太空船，让我们开始控制它工作。正如我刚刚提到的，它有3个控制：向左转、向右转和点燃火箭，分别映射为左、右和上键。这个示例的基本结构非常类似于本章前面的Acceleration3.as，使用keyDown和keyUp事件和一个switch语句来处理被按下的键。我将把整件事都交给你然后完成它（你可以在ShipSim.as中看到这些内容）：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
    import flash.ui.Keyboard;

    public class ShipSim extends Sprite
    {
```

www.TopSage.com

```
private var ship:Ship;
private var vr:Number = 0;
private var thrust:Number = 0;
private var vx:Number = 0;
private var vy:Number = 0;
public function ShipSim()
{
    init();
}

private function init():void
{
    ship = new Ship();
    addChild(ship);
    ship.x = stage.stageWidth / 2;
    ship.y = stage.stageHeight / 2;
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
    stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
}

private function onKeyDown(event:KeyboardEvent):void
{
    switch(event.keyCode)
    {
        case Keyboard.LEFT :
        vr = -5;
        break;

        case Keyboard.RIGHT :
        vr = 5;
        break;

        case Keyboard.UP :
        thrust = 0.2;
        ship.draw(true);
        break;

        default :
        break;
    }
}

private function onKeyUp(event:KeyboardEvent):void
{
    vr = 0;
    thrust = 0;
    ship.draw(false);
}

private function onEnterFrame(event:Event):void
{
    ship.rotation += vr;
    var angle:Number = ship.rotation * Math.PI / 180;
```

142

www.TopSage.com

大家网 TopSage.com

```
            var ax:Number = Math.cos(angle) * thrust;
            var ay:Number = Math.sin(angle) * thrust;
            vx += ax;
            vy += ay;
            ship.x += vx;
            ship.y += vy;
        }
    }
}
```

首先你定义了vr：旋转速度，或当你让它旋转时太空船转得有多快。它开始时设置为0，意味着太空船不能转动。

```
private var vr:Number = 0;
```

但是在onMouseDown方法中，如果switch语句发现左或右光标键被按下，它会分别设置vr为–5或5：

```
case Keyboard.LEFT :
vr = -5;
break;

case Keyboard.RIGHT :
vr = 5;
break;
```

然后，在onEnterFrame处理器函数中，你加入vr到太空船的当前旋转中，一个或另一个方向转动太空船。当一个光标键被放开，则重新使vr设置为0。这些都是关于旋转的。现在让我们来看一下驱动力。

143

你的太空船已经在那里了，你已经确定好了它要去哪了——3，2，1，发射！那么，你怎么让它执行呢？好，它刚好与你在那个修改的鼠标跟随示例中所做的是相反的。在那个示例中，你计算一个角然后计算基于角的旋转和加速度。在这里，你是从旋转开始并反向工作来计算角和*x*和*y*方向的力。

声明一个thrust变量是用来跟踪在任意给定的时间有多少力施加在太空船上。显而易见，加速度只有在火箭被点燃时才会产生，因此它是以0开始的：

```
private var thrust:Number = 0;
```

然后switch语句开始发挥作用。如果上光标键被按下，则设置thrust变量为一个很小的值，如0.2。这也是你想从视觉上表明正在施加某个力的位置，我们是通过在船上绘制火焰来表现的：

```
case Keyboard.UP :
thrust = 0.2;
ship.draw(true);
break;
```

另外，当一个键被释放时，设置thrust变量为0，并且清除掉火焰。

```
private function onKeyUp(event:KeyboardEvent):void
{
    vr = 0;
    thrust = 0;
    ship.draw(false);
}
```

www.TopSage.com

现在，当你开始注意onEnterFrame函数时，你会知道旋转度数，并且你会知道力将被施加（如果有的话）。将旋转转换为弧度然后使用正弦和余弦，结合thrust变量，来计算出每个轴上的加速度。

```
private function onEnterFrame(event:Event):void
{
    ship.rotation += vr;
    var angle:Number = ship.rotation * Math.PI / 180;
    var ax:Number = Math.cos(angle) * thrust;
    var ay:Number = Math.sin(angle) * thrust;
    vx += ax;
    vy += ay;
    ship.x += vx;
    ship.y += vy;
}
```

测试它，让你的太空船飞行。相当令人惊奇的是你可以很容易地制作出这么复杂的动画。如果你使用了我的绘制太空船的代码，你需要确保背景设置为黑色或其他深一些的颜色，这样白色
144 的线条才会看得清。

5.3 本章重点公式

那么，现在你又有几个工具可以添加到你的工具箱中了，就是下面这些。

转换角速度为*x*、*y*速度：

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

转换角加速度（任何作用在物体上的力）为*x*、*y*加速度：

```
ax = force * Math.cos(angle);
ay = force * Math.sin(angle);
```

将加速度加到速度中：

```
vx += ax;
vy += ay;
```

将速度加到位置上：

```
movieclip._x += vx;
sprite.y += vy;
```

5.4 小结

这一章讲解了基本的速度和加速度，这两个因素将会在你的大多数脚本动画中应用。你已经学习了向量和向量加法。你已经明白了如何实现一个轴上的速度、两个轴上的速度和通过角度来转换为*x*和*y*速度的角速度。同时你也学习了加速度——它与速度有着怎样的联系和如何将它施加到一个轴上、两个轴上或一个角上。

本章最重要的是理解加速度和速度的应用，如下面描述的步骤：

转换已经存在的角速度为*x*和*y*方向上的分解速度。转换角加速度为*x*、*y*方向上的分解加速度。将加速度加到每一个轴的速度上。将速度加到每一个轴的位置上。

145 在下一章，你将在这些概念的基础上进行构建，使用弹力和摩擦力来增加一些环境的交互。

www.TopSage.com

第6章

边界和摩擦力

本章内容

- 环境边界
- 摩擦力
- 本章重点公式

在前面的章节中你已经学习了许多理论，现在你可在精灵中放上图形，并使用各种各样的力让它在屏幕上移动。然而，到目前为止，你在创建这些示例时可能会遇到一些小麻烦：当你将物体移出屏幕时——它突然不见了。有时候，如果它正在以某个角快速地移动，你不知道如何让它回来。唯一的选择就是重新运行影片。

在大多数现实世界的场景中，存在一些边界（boundary）——有时候是墙和一个隔板，但最低限度几乎总是有地面的。除非你正在制作太空模拟器，你需要一些环境障碍来始终保持物体是可见的。如果你正在做太空中的某些东西，至少需要保持主要物体是可见的。

你可能注意或没有注意的另一件事是，这个环境（或者没有这个环境）在物体移动时不会对物体产生任何影响。你开始让一个物体移动，它会一直沿着它出发的方向以相同的速度移动，直到你对它施加了其他的力。实际上，这根本不是一个缺陷，而是宇宙运行的方式。术语惯性（inertia）是用来描述一个物体在穿过太空后会继续以原来的方向和速率运行，直到有其他的力施加到它身上。或者说，它的速度会一直不变直到有某个东西来改变它。在一般情况下，能够改变速度的有某种摩擦力（friction）——哪怕只是空气的摩擦力。因此，当你成功地模拟了一个物体在真空中移动，你可能会渴望它更像你每天的生活环境中的一些东西。

本章将讲解这些问题。首先你将要学习如何使用边界，然后学习如何模拟摩擦力。让我们开始吧。

6.1 环境边界

我们将以设置一些边界开始。根据你参加过的任何活动——不管是在运动、在工作、建房子等等——它们通常都有一个活动的空间概念。如同你正在说，“我只对这个区域发生的事情感兴趣。如果某些事情发生在这个区域之外，此时我不会去关心它。但是如果某些事情发生在当前这

www.TopSage.com

个空间中，我需要注意它。”

如果在你感兴趣的区域中的某个物体移出了这个空间，你有一个选择。你可以通过一些方法将它移回这个区域中，或者你可以不再注意它。另一种选择就是跟随着它。从另一个角度讲，你移动空间以致它可以继续围住物体，即使这个物体正在移动。它与在Flash中并没有多大的不同之处。

在你的影片中，为你的物体设置一个空间。通常，这就是整个舞台，但是这个区域可以是舞台的某一部分，或比舞台还要大的空间。因为物体是移动的，所以它们极有可能离开这个空间。当它们离开时，你可以忽略它们、把它们移回到区域中或跟随着它们。我将讲解前面两种情况的策略。首先，让我们先解决在哪里设置边界和如何设置它们。

6.1.1 设置边界

在大多数情况下，使用一个简单的矩形来作为一个边界。让我们以一个最简单的例子开始，一个基于舞台大小的边界。在AS 3中，舞台由一个名为stage的属性来描述，它是每一个显示对
148 象的一部分。因此，在你的文档类（它扩展了MovieClip或Sprite）中，你可以直接使用这个属性来访问舞台和它的属性。如果想让你的舞台在播放器窗口发生变化时同时自动地缩放，需要设置下面两个属性：

```
stage.stageAlign = StageAlign.TOP_LEFT;
stage.scaleMode = StageScaleMode.NO_SCALE;
```

注意你需要先在类顶端导入flash.display.StageAlign和flash.display.StageScaleMode类。

现在影片的顶端和左边的边界都将是0，那么右边和底端的边界将是stage.stageWidth和stage.stageHeight。你可以把它们保存在一些变量中，如下：

```
private var left:Number = 0;
private var top:Number = 0;
private var right:Number = stage.stageWidth;
private var bottom:Number = stage.stageHeight;
```

如果你确实像这样将它们保存在了一些变量中，意味着当舞台大小发生变化时它们将不会发生改变。如果你想有一个固定的边界区域，使用这种形式是很好的。

然而，如果你想总是使用整个舞台区域的边界，即使舞台的大小会发生变化，只需要直接在你的代码中使用stage.stageWidth和stage.stageHeight。本章的大多数示例将使用这种形式。

还需要注意的是我将在一些示例中使用整个舞台，这并不代表着你也必须要这样做。例如，你可以制作一个物体需要呆在其中的“房间”。在这个例子中，边界可能是：top=100，bottom =300，left =50，right =40。

好，现在你已经有了边界，你用它们来做什么呢？好，查看所有你正在注意的移动物体，看它们是否还在这个空间里。你可以使用两个if条件语句来完成。下面是一个简单的版本，你在下面可以看到：

```
if(ball.x > stage.stageWidth) {
   // 做一些事情。
```

www.TopSage.com

```
} else if(ball.x < 0) {
   // 做一些事情。
}

if(ball.y > stage.stageHeight) {
   // 做一些事情。
} else if(ball.y < 0) {
   // 做一些事情。
}
```

这里为了高效我使用了if和else语句来实现。如果小球的x位置比右边界大，那么它向右边的路已经结束了。同样在向左时如果小于左边的边界，它向左方的路也会结束，因此你不需要在一个单独的if语句中检查它。你需要在第1个if条件失败后检查left变量。对于top和bottom也是同理。然而，物体有可能同时在*x*和*y*轴两个方向上出界，因此，我们要保持两组分开的if条件语句。 149

但假设物体出了边界你应该做什么呢？你将要了解到下面这4种方式：

- 移除它。
- 将它移回，就好像它是一个全新的物体（重新生成）。
- 让它在屏幕上折回，就好像相同的物体在不同的位置上重新出现。
- 让它弹回到原来的区域。

让我们先开始简单的移除。

6.1.2 移除物体

如果你总是不断地有新的物体生成，那么当某个物体移出边界时简单地移除它是非常有用的。你所移除的物体将被新进来的物体所代替，因此你不会以一个空的舞台结束。另外，你也不能让太多的物体同时在移动，它将会导致Flash播放器变慢。

要移除一个精灵或任何其他显示对象，只需要调用removeChild(*objectName*)。它将移除舞台上的实例。注意移除一个显示对象并不是销毁它。如果真地想从内存中移除一个对象，你应当调用delete *objectName*来清除它。

你还需要确保代码不会继续从外部的enterFrame处理器中移动精灵。如果实例是唯一被移动的内容，并且动画只是被enterFrame事件处理器控制的，你可以使用removeEventListener(Event.ENTER_FRAME,onEnterFrame)；来停止执行处理器。另一方面，如果有许多物体在移动，你想保持运行代码以继续移动大多数的物体，但是想让已经移除的精灵从移动的物体列表中清除。在这种情况下，你需要将移动对象的引用保存在一个数组中然后通过循环数组来移动每一个对象。那么，当你移除这些对象中的一个，使用Array.splice方法来从数组中移除它。稍后你会看到这样的例子。

那么，如果你想移除一个精灵，而且已经知道了边界，你可以使用向下面这样的if条件语句：

```
if(ball.x > stage.stageWidth ||
   ball.x < 0 ||
   ball.y > stage.stageHeight ||
   ball.y < 0)
{
   removeChild(ball);
}
```

www.TopSage.com

大家网 TopSage.com

符号||意味着“或”因此，你实际上是在说，“如果物体超过右边界，或物体超过左边界，或它超过上边界或超过下边界，则移除它。”这种技术可能对于多数情况下的移除动作是不错的，因为它不需要考虑物体在哪儿出的边界。但是在其他情况下，你可能想让物体碰上特定的边界时做出不同的反应，如左边的墙和它对面的右边的墙。在这些例子中，需要使用分开的if条件语句。稍后你将会在屏幕折回的示例中看到。

150

> 关于垃圾回收器的注意事项：垃圾回收（Garbage collection）是指清除程序中不需要或不使用或是被删除的内容的进程。在AS 3中，当你删除了一个对象，它不会像早期的ActionScript版本一样立即被销毁，而只是被标记为删除。垃圾回收会定期清理内存，清除它所发现的标记为删除的内容。因此，如果删除了一个对象然后跟踪它，你会发现它仍然会存在一段时间。在我们刚刚说到的这些例子中，你只是使用针对精灵的外部代码来移动一个精灵，所以精灵自身只是一个承载图像的容器。然而，新的删除动作变得很重要，如果你的对象自身内部包含enterFrame代码或某种定时器或间隔代码。这些代码可能会在你“删除”了正在运行它的对象之后仍继续运行。因此你需要停止掉enterFrame、间隔或定时器代码，而不能单单依靠删除来停止它。在本书的后面我们会碰到这样的例子。

所有的看上去都不错，但是有一个小问题。现在，你可能在许多情况下都没有注意到这样的问题，但有些时候它会让一些事情看上去是错的。

要了解我所说的意思，先看一下图6-1。

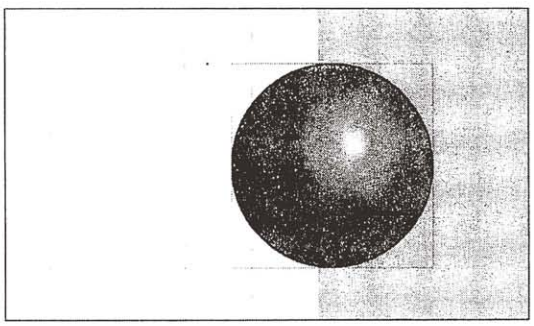


图6-1　这个小球并没有完全离开舞台，但它将会被移除

在图6-1中，你可以看到小球的位置，根据于它的中心注册点，它确实已经越过了屏幕的右边界。它将被移除。如果这小球移动得足够快，那么它看上去可能是没有问题的。但是如果它移动得非常慢，每帧只移动一个像素或更少呢？你会看到它移出屏幕的边界，当移动到半路上时，突然！它就消失了。这非常类似于一个演员在退到舞台中途就开始卸妆一样。这是一种错误的视觉。

151

因此，你需要让小球完全离开画面。那么，当它离开视线，你可以很从容地执行移除了。

要做到这些，你需要考虑物体的宽度。实际上，因为注册点在中心，你只需要关心它的一半宽度，它恰好可以存储在radius属性中。要做这些，你的if语句会变得稍微复杂一些。它看起像如下这样：

www.TopSage.com

```
if(ball.x - ball.radius > stage.stageWidth ||
   ball.x + ball.radius < 0 ||
   ball.y - ball.radius > stage.stageHeight ||
   ball.y + ball.radius < 0)
{
   removeChild(ball);
}
```

下面给出图6-2所示的图像。

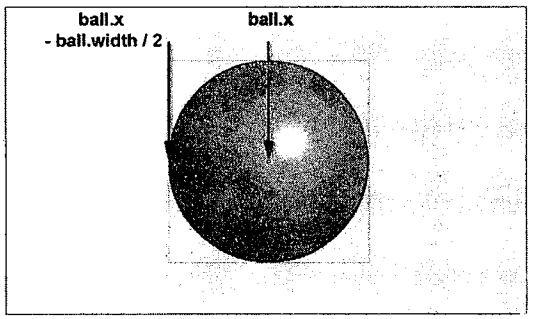


图6-2 这个小球已经完全移出屏幕，可以安全移除了

注意尽管当前我们的示例中使用的是一个小球，或圆形物体，这些代码同样适用于注册点在中心的任何形状的物体。

让我们进一步来试验一下。在下一个示例中，我们将使用前面章节中使用过的Ball类，并增加一些内容。我们加入公开的vx和vy属性，以使每一个小球都有自己的速度。下面是它的代码：

```
package {
    import flash.display.Sprite;

    public class Ball extends Sprite {
        public var radius:Number;
        private var color:uint;
        public var vx:Number = 0;
        public var vy:Number = 0;
        public function Ball(radius:Number=40, color:uint=0xff0000) {
            this.radius = radius;
            this.color = color;
            init();
        }
        public function init():void {
            graphics.beginFill(color);
            graphics.drawCircle(0, 0, radius);
            graphics.endFill();
        }
    }
}
```

152

在纯正的面向对象编程中，你可能不需要像这样创建公开属性，而是保持它们为私有，并且使用公开的getter/setter方法来改变它们。但是为了让一些事情简单一些，我们将打破这个规

www.TopSage.com

则并将它们设置为公开(public)。

下面是文档类Removal.as，它设置了很多小球并且当它们移出屏幕时移除它们。

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class Removal extends Sprite
    {
        private var count:int = 20;
        private var balls:Array;

        public function Removal()
        {
            init();
        }

        private function init():void
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align=StageAlign.TOP_LEFT;
            balls = new Array();
            for(var i:int = 0; i < count; i++)
            {
                var ball:Ball = new Ball(10);
                ball.x = Math.random() * stage.stageWidth;
                ball.y = Math.random() * stage.stageHeight;
                ball.vx = Math.random() * 2 - 1;
                ball.vy = Math.random() * 2 - 1;
                addChild(ball);
                balls.push(ball);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            for(var i:Number = balls.length - 1; i > 0; i--)
            {
                var ball:Ball = Ball(balls[i]);
                ball.x += ball.vx;
                ball.y += ball.vy;
                if(ball.x - ball.radius > stage.stageWidth ||
                   ball.x + ball.radius < 0 ||
                   ball.y - ball.radius > stage.stageHeight ||
                   ball.y + ball.radius < 0)
                {
                    removeChild(ball);
                    balls.splice(i, 1);
                    if(balls.length <= 0)
```

153

www.TopSage.com

```
                {
                    removeEventListener(Event.ENTER_FRAME,
                                        onEnterFrame);
                }
            }

        }
    }
}
```

这对你来说是很简单的。首先，你创建了20个Ball类的实例，将它们随机地放在舞台上，分配给它们随机的*x*和*y*速度，将它们加入到显示列表中，并将它们加入到数组中。

onEnterFrame方法只是使用均速移动小球，检测边界，并移除已经移出边界的小球。注意它不仅从显示列表中删除小球，同时也使用了Array.splice方法来从数组中移除它的引用。Array.splice方法提供了你想从哪开始移除元素的索引值和你想删除多少元素的参数。在此例中，你只需要移除一个元素：当前索引的一个元素。当你测试时，你甚至意识不到它们被移除，这是件好事情，不过你放心，它们不可能始终在舞台上徘徊，而消耗宝贵的内存和CPU资源。

你可能会注意到本例中的for语句与其他示例中的有些不同：

```
for(var i:Number = balls.length - 1; i > 0; i--)
```

这使得for循环是从后向前而不是从前向后。这是必要的，因为如果你在数组中使用了splice方法，整个数组的索引将发生改变，当你增加i时，你将会跳过其中的某一个元素。所以使用后退的方式来处理。 154

最后，注意当从数组中删除一个元素之后，它会检测数组的长度是否降到了0。在这个例子中，最后没有东西在移动，因此enterFrame侦听器被移除。

6.1.3 重新生成物体

下一个计划是重新生成已经移出舞台的物体。实际上，它并不是像重定位或重新设置速度那样真正地重新生成。思路是当一个对象已经离开舞台并且不再需要时，你可以把它作为一个崭新的对象重新放回舞台让它播放。这让你有了源源不断的对象出现，你不需要担心会有过多的物体出现在舞台上，因为总是要设置一定的数量的。

这项技术对于喷泉效果是非常有用的，如你有一个恒定喷射的粒子系统。粒子离开舞台然后又重新回到喷射源点。

重新生成的技巧非常类似于移除。你需要等对象移出边界后，不是移除它，而是要移动它。

让我们来制作一个喷泉。对于喷泉的粒子，我们使用同样的Ball类，但是设置它为2像素大小并给它一个随机的颜色值。喷泉的源头将被设置在舞台底端的中心位置。所有粒子将从那里喷射，然后当它们离开舞台后，它们将重新回到喷泉的源头位置。还有，所有粒子将以一个随机的负的*y*速度和一个较小的随机的*x*速度开始。这意味着它们在向上喷射的同时也会轻微地向左或右方向喷射。当粒子重新生成时，它们的速度被以同样的方式设置。粒子也将会对重力做出反应。

www.TopSage.com

下面是文档类Fountain.as:

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class Fountain extends Sprite
    {
        private var count:int = 100;
        private var gravity:Number = 0.5;
        private var balls:Array;

        public function Fountain()
        {
            init();
        }

        private function init():void
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align=StageAlign.TOP_LEFT;
            balls = new Array();
            for(var i:int = 0; i < count; i++)
155         {
                var ball:Ball = new Ball(2, Math.random() * 0xffffff);
                ball.x = stage.stageWidth / 2;
                ball.y = stage.stageHeight;
                ball.vx = Math.random() * 2 - 1;
                ball.vy = Math.random() * -10 - 10;
                addChild(ball);
                balls.push(ball);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            for(var i:Number = 0; i < balls.length; i++)
            {
                var ball:Ball = Ball(balls[i]);
                ball.vy += gravity;
                ball.x += ball.vx;
                ball.y += ball.vy;
                if(ball.x - ball.radius > stage.stageWidth ||
                   ball.x + ball.radius < 0 ||
                   ball.y - ball.radius > stage.stageHeight ||
                   ball.y + ball.radius < 0)
                {
                    ball.x = stage.stageWidth / 2;
                    ball.y = stage.stageHeight;
                    ball.vx = Math.random() * 2 - 1;
```

www.TopSage.com

TopSage.com

```
                ball.vy = Math.random() * -10 - 10;
            }

        }
    }
}
```

你开始在init方法中设置了所有粒子的开始位置点并赋予它们随机向上的速度。

onEnterFrame方法循环balls数组中所有的元素。对于每一个元素，它都加入了gravity到它的vy值，加入速度到它的位置，并且检测它是否已经越过了边界。如果它超过了边界，它将回到开始位置点并以一个新的速度“重生”。它的行为就像是新生成的粒子。因此，你的喷泉将永不停止。

你应当试验一下。让喷泉向不同的方向喷射。让它从墙上或天花板上喷出来。改变一些随机因素以使喷泉的喷射范围更宽或更窄。试着加入一些风在里面提示：创建一个wind变量然后将它加入到vx中。

156

6.1.4 屏幕折回

下一个常用的控制物体移出边界的方式是*屏幕折回*（screen wrapping）。这个概念很简单：如果一个物体移出了屏幕左边的边界，它会在右边重新出现。如果它移出了右边的边界，它会重新回到左边。如果它超过了上边界，它会在底端重新出现。这是它的思路。

屏幕折回非常类似于重新生成物体，它需要你将物体移回到屏幕的不同位置。而在重新生成物体中，你通常是把移出屏幕的物体重新移回到相同的位置，使它们看上去像是崭新的物体。在屏幕折回中，你通常需要让移回来的物体感觉上是同一个物体，可以说它只是刚刚从后门出去然后又从前门进来。因此，在屏幕折回中你通常不需要改变速度。

再次回到我最喜欢的老游戏上，行星游戏。你可能还记得第5章，这是飞船影片存在的一个问题：飞船将飞出屏幕，并且有些时候无法知道它在哪里以及如何让它回来。使用屏幕折回，你的精灵不会离开屏幕超过一个像素的距离。

让我们重建太空船示例并且加入这个行为。下面是你需要用到的文档类（ShipSim2.as），新的内容以粗体显示：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
    import flash.ui.Keyboard;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    public class ShipSim2 extends Sprite
    {
        private var ship:Ship;
        private var vr:Number = 0;
        private var thrust:Number = 0;
```

```
private var vx:Number = 0;
private var vy:Number = 0;
public function ShipSim2()
{
    init();
}

private function init():void
{
    stage.scaleMode = StageScaleMode.NO_SCALE;
    stage.align=StageAlign.TOP_LEFT;
    ship = new Ship();
    addChild(ship);
    ship.x = stage.stageWidth / 2;
    ship.y = stage.stageHeight / 2;
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
    stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
}

private function onKeyDown(event:KeyboardEvent):void
{
    switch(event.keyCode)
    {
        case Keyboard.LEFT :
        vr = -5;
        break;

        case Keyboard.RIGHT :
        vr = 5;
        break;

        case Keyboard.UP :
        thrust = 0.2;
        ship.draw(true);
        break;

        default :
        break;
    }
}

private function onKeyUp(event:KeyboardEvent):void
{
    vr = 0;
    thrust = 0;
    ship.draw(false);
}

private function onEnterFrame(event:Event):void
{
    ship.rotation += vr;
    var angle:Number = ship.rotation * Math.PI / 180;
    var ax:Number = Math.cos(angle) * thrust;
```

157

www.TopSage.com

```
            var ay:Number = Math.sin(angle) * thrust;
            vx += ax;
            vy += ay;
            ship.x += vx;
            ship.y += vy;
            var left:Number = 0;
            var right:Number = stage.stageWidth;
            var top:Number = 0;
            var bottom:Number = stage.stageHeight;
            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;
            }
        }
    }
}
```

158

这里面使用到了原来第5章中的Ship类，因此要确保Ship类在你的类路径中。因为飞船是使用白线绘制的，不要忘记将背景颜色改为黑色（查看第4章关于如何修改的信息）。如你所看到的，这个新的类增加了边界定义并且检测它们。注意，你需要使用分开的if和else语句，因为在每种情况下动作是不同的。

6.1.5 回弹

现在我们已经到了可能是最常用也是最复杂的边界控制方式阶段。但实际上它真地没有比屏幕折回复杂多少，因此不必害怕。

回弹（bouncing）的思路是检测物体什么时候离开屏幕，就在这时，你把它留在那里然后只改变它的方向。规则很简单：如果它移出了左边界或右边界，反转它的*x*方向速度。如果它移出了上边界或下边界，反转它的*y*方向速度。反转一个轴上的速度很简单。只要将它乘以–1。如果速度是5，反转后就成了–5。如果它是–13，反转后它就变成了13。代码也同样简单：vx*= –1或vy*= –1。

屏幕折回中的一些内容需要修改。首先，在屏幕折回中，你要让物体完全离开舞台之后才能重新定位它。你是通过它的位置与它自身的宽的一半相加或相减来实现的。而对于回弹，你需要做的几乎是相反的。不能等到物体完全离开舞台后才开始让它回弹。实际上，你也不希望物体的某个部分离开画面。如果将一个小球扔向一个真实的墙上，你不希望看见它在回弹之前小球进入到墙中。它将碰到墙上，停在那里然后回弹回来。因此，你需要知道的第1件事情就是物体的任

www.TopSage.com

何部分超过墙面的时刻，你所有需要做的是反转增加宽或高的一半的方式。例如，不是像这样：

```
if(ball.x - ball.radius > right) . . .
```

而是：

159

```
if(ball.x + ball.radius > right) . . .
```

两者的不同你可以从图6-3中看到。

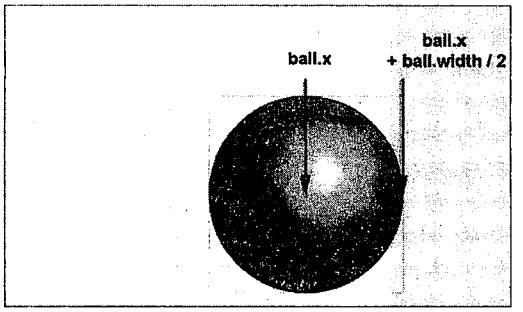


图6-3　这个小球刚刚接触到舞台右边界，但是它需要回弹

现在，你已经确定物体超过了一点边界。我在前面说过反转那个轴上的速度。但除此之外还有一点儿内容。你还需要重新定位物体以使它处在右侧的边界上。很明显这主要是为了有物体碰到边界和回弹的效果，而不是陷入到墙中。但它也是必要的一步。你会发现如果你并没有调整物体的位置，那么在下一帧时，它可能仍会超过边界，继续移动。如果发生了这样的情况，物体将会再一次反转它的方向并且陷入墙中！那么你看到的现象是小球在墙边上来回进出，始终在那里颤抖。这是不好看的。

你需要把物体放置在右侧边界的位置实际上就是使用if语句检测的位置。你只需要使用一点基本的代数学重新表述它。下面是*x*轴上的整个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;
}
```

160

图6-4显示了小球重新放置后的状态。

因此，回弹的步骤如下：

- 检测物体是否超过了边界。
- 如果是这样，把它放在边界的边上。
- 反转速度。

www.TopSage.com

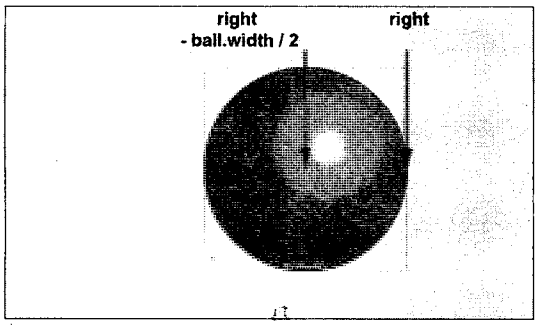


图6-4 小球已经被精确地重新定位在边界上

这就是关于回弹的所有内容。我想现在该停止讨论并且显示一些具体的内容了。下面的例子我们又一次使用了相同的Ball类，但将它缩小到了一个合适的大小。这里是一个文档类(Bouncing.as)：

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class Bouncing extends Sprite
    {
        private var ball:Ball;
        private var vx:Number;
        private var vy:Number;

        public function Bouncing()
        {
            init();
        }

        private function init():void
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align=StageAlign.TOP_LEFT;
            ball = new Ball();
            ball.x = stage.stageWidth / 2;
            ball.y = stage.stageHeight / 2;
            vx = Math.random() * 10 - 5;
            vy = Math.random() * 10 - 5;
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            ball.x += vx;
```

161

www.TopSage.com

大家网 TopSage.com

```
            ball.y += vy;
            var left:Number = 0;
            var right:Number = stage.stageWidth;
            var top:Number = 0;
            var bottom:Number = stage.stageHeight;

            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;
            }
        }
    }
}
```

测试这个影片几次来观看小球不同角度的移动。试着提高或降低小球的速度。

我不得不承认这是本书中关于数学和计算与现实中的物理差别比较大的地方。如果你看到图6-5，你将会发现小球真正碰到墙的位置应该在哪儿以及我们模拟的位置在哪儿。

162

要得到正确的位置需要更复杂的计算。虽然你可以去弄明白如何做到这一点（使用第3章中的三角学），但我保证你多半不会看出它们之间有多大的差别。当然，如果你正在做一些需要精确的计算和位置模拟，你可能需要去找另一本书，并且要重新考虑选择软件。但对于在Flash中创建的大多数游戏或是视觉效果，这种方法对你来说刚刚好。然而，你还可以再加入一点东西以使它更接近现实。

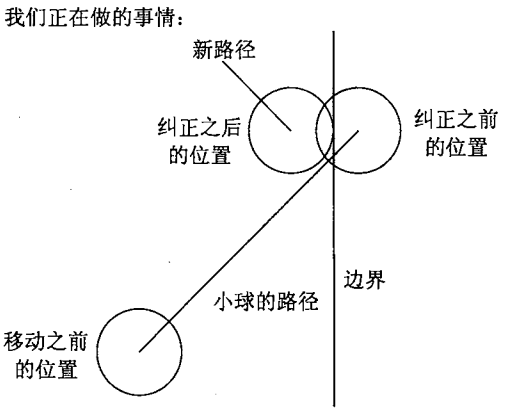
如果你拿着一个橡胶弹力球然后伸直胳膊并且放开手，它将落到地板上然后回弹到几乎开始下落的位置。事实上它永远不会再回弹到你的手上。这是因为它在回弹过程中有了一定的能量损失。它可能会失去一些类似声一样的能量，也有可能是一点热能。地板会吸收它的一些能量，也可能周围的空气也会吸收一些能量。然而，最重要的一点是小球在回弹之后慢了下来。从另一个角度讲，它在回弹的轴上失去了一些速度。

你可以在Flash中很容易地重建这个过程。目前为止，你一直使用–1作为回弹的系数。这意味着物体以它碰撞的100%的力量回弹。要想让它损失些能量，只需要使用比–1大一点的数。与其他数值一样，最好是将它作为一个变量定义在你的文件上方然后在你的代码中使用这个变量。

www.TopSage.com

只需要创建一个名为bounce的变量名，然后在文件的上方设置如-0.7大小的值：

```
private var bounce:Number = -0.7;
```



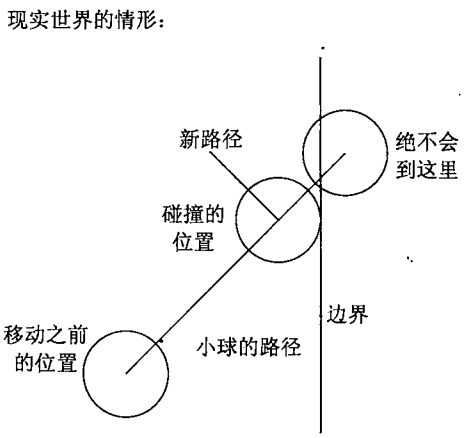


图6-5 这项技术不是很完美，但是它很快、容易，而且对于大多数情况已经足够接近真实了

然后替换所有的if语句中的-1为bounce变量。然后继续测试它，你会发现现在的回弹更加真实了。试着使用不同的bounce值直到你了解了它是如何工作的。

要学习这部分内容的最好方法是选择你学过的每种原理并看你能把它与多少种其他的原理结合起来应用。你可以在www.friendsofed.com网站上找到本书的文档类Bouncing2.as，它是一个回弹与重力相结合的例子。但我确信你已经有足够的能力自己加一些像这样的其他因素在里面。也许你想尝试使用键盘为小球增加一些加速度，或应用一些其他你已经学到的内容。

现在，关于边界的讨论就到此为止，我们来讨论一下边界之间发生的事情——物体穿过空间时发生的事情。

www.TopSage.com

6.2 摩擦力

到目前为止，你已经看到两种方案：

- 物体以它所拥有的速度简单地移动，直到碰到一个边界。
- 物体自身或是某个外力使物体产生加速度，改变了它的速度。

对于这两种设置，除非物体被推或是拉，或它碰到什么东西，否则它会始终保持相同的速度向相同的方向移动。但在现实生活中，它并不是这样的。

这里有一个例子。拿一张纸，把它揉成比较松的一团，然后尽力把它扔出去。它很可能穿不过房屋。确实，重力把它拉了下来（*y*轴向），但是当它离开你的手之后，它在*x*轴上移动的速度相当快。然而，很快，它的*x*速度几乎为0。

很明显，没有“负的加速度”将纸拉回到你的手上，而它的速度确实被改变了。这被称为摩擦力或阻力。尽管在技术上它不是一个力，但确实改变了物体的速度。规则是摩擦力只减少速度的大小。而对于物体前进的方向，是不会被摩擦力改变的。从另一个角度讲，摩擦力可以将速度减小到0，但它永远不能使一个物体转弯或是改变为其他方向。

163 ~ 164

那么你如何使用代码来执行摩擦力呢？有两种方式。像我们生活中的大多数事情一样，你有正确的方法和容易的方法。但不管怎样，我将推荐使用容易的方法。尽管是这样，我将向你展示这两种方法，先从“正确”的方法开始，以让你对它有个认识。

6.2.1 摩擦力，正确方法

摩擦力实际上是削减速度，也就是说你有某一个摩擦力值，然后从你的速度中减去它。实际上，你需要从速度的大小，或速率中减掉它。你不能单独地从*x*轴和*y*轴上减去摩擦力。否则如果当一个物体沿某一角度移动时，其中的一个轴向的速度会在另一个轴向速度之前为0，而物体会沿着水平或垂直方向继续移动一段时间，这看上去是很怪的。

那么，你需要做的就是找到角速度中的速率和方向（如果你还没有这些值）。要找到速率，你需对vx的平方加上vy的平方的和开平方（你应可以认出它是我们在第3章中学到的毕达哥拉斯定理）。要找到角度，你可以通过计算Math.atan2(vy,vx)来得到，它的代码看上去如下：

```
var speed:Number = Math.sqrt(vx * vx + vy * vy);
var angle:Number = Math.atan2(vy, vx);
```

然后你可以从速度中减去摩擦力。但你需要确保你的速率不是负值，否则它会反转速度。因此，如果你的摩擦力比你的速率大，速率则应置为0。下面是计算的代码：

```
if(speed > friction)
{
   speed -= friction;
}
else
{
   speed = 0;
}
```

www.TopSage.com

现在，你需要将角速度转换回*vx*和*vy*，使用的是正弦和余弦，如下：

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

165

很多工作，嗯？下面是把它们结合在一起的文档类Friction1.as：

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class Friction1 extends Sprite
    {
        private var ball:Ball;
        private var vx:Number = 0;
        private var vy:Number = 0;
        private var friction:Number = 0.1;

        public function Friction1()
        {
            init();
        }

        private function init():void
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align=StageAlign.TOP_LEFT;

            ball = new Ball();
            ball.x = stage.stageWidth / 2;
            ball.y = stage.stageHeight / 2;
            vx = Math.random() * 10 - 5;
            vy = Math.random() * 10 - 5;
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
          var speed:Number = Math.sqrt(vx * vx + vy * vy);
          var angle:Number = 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;
```

166

www.TopSage.com

```
            ball.y += vy;
        }
    }
}
```

这里，我已经给出恢复速度代码。friction被设置为0.1，小球被指定了一个随机的*x*和*y*轴上的速度。在onEnterFrame方法中，speed和angle如我刚才描述的那样被计算出来。如果speed小于friction，则相减；否则，speed为0。最后，*vx*和*vy*被重新计算然后加入到位置当中。

测试这个影片几次来观看不同的速率和角度时的效果。这就是有摩擦力的工作过程。它使用了许多行代码和4个三角函数来完成。你可能会想出一个更优化的版本，但所有的计算都会在那儿。下面来说说容易的方法。

6.2.2　摩擦力，容易方法

你可能已经预料到了，完成摩擦力的容易的方法并没有像我前面所述的技术那样精确，但是我打赌没有人会注意到。它只是两行简单的相乘代码。所有你要做的事情就是把*x*和*y*的速度乘以一个小于1的数。像是0.9或0.8通常比较合适。因此，在每一帧上，vx和vy变成了上一次的80%或90%。在理论上，速度将趋近于0，但永远不会等于0。在实际中，计算机只能计算到像是“分”这样大小的数值，最终它的速度将会变成0。在变成0之前的一段时间里，动画会很小很小将不会被察觉出来。

一个比较好的消息是使用这种方式速度永远不会是负值，因此你不需要因为担心而去检测它。还有，*x*和*y*轴上的速度将会以同样的速率到达0，因此不需要将轴向速度转换为角速度然后再转换回去。

唯一需要在代码中（你可以在文档类Friction2.as中找到）注意的是改变了friction变量值为0.9和下面onEnterFrame方法中的一些变化：

```
private function onEnterFrame(event:Event):void
{
    vx *= friction;
    vy *= friction;
    ball.x += vx;
    ball.y += vy;
}
```

现在，它真地很容易！多测试几回这个影片并且感觉一下它。你能看出它与前面正确方法之间的差别吗？可能会看出来，因为你已经看过两者的内部代码了。但是对于一般的观看者来说，
167 这个看上去是相当真实的。

6.2.3　摩擦力的应用

现在，让我们回到熟悉的太空船上并且在太空中应用一些摩擦力。文档类名为ShipSim Friction.as，我们只是在ShipSim2.as的基础上加入了一个摩擦力变量：

```
private var friction:Number = 0.97;
```

除了其他的类变量之外，还改变了onEnterFrame方法如下：

```
private function onEnterFrame(event:Event):void
{
```

www.TopSage.com

大家网 TopSage.com

```
        ship.rotation += vr;
        var angle:Number = ship.rotation * Math.PI / 180;
        var ax:Number = Math.cos(angle) * thrust;
        var ay:Number = Math.sin(angle) * thrust;
        vx += ax;
        vy += ay;
        vx *= friction;
        vy *= friction;
        ship.x += vx;
        ship.y += vy;
        var left:Number = 0;
        var right:Number = stage.stageWidth;
        var top:Number = 0;
        var bottom:Number = stage.stageHeight;
        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;
        }
    }
```

尽管只有3行新的代码但它的感觉是相当不同的。

不要忘记思考一下除了x、y之外的内容。摩擦力可以应用在任何有速度的地方。假设你有一个旋转的物体（使用vr）。对它应用摩擦力同样会使它慢下来并停止旋转。你可以使用第5章中旋转的箭头来试验。 168

你可以试着使用它来处理各种物体，如一个轮盘、电扇或一个飞船。

6.3 本章重点公式

让我们回顾一下本章所介绍的重要的公式。

移除超过边界的物体：

```
if(sprite.x - sprite.width / 2 > right ||
   sprite.x + sprite.width / 2 < left ||
   sprite.y - sprite.height / 2 > bottom ||
   sprite.y + sprite.height / 2 < top)
{
      // 移除精灵的代码。
}
```

重新生成超过边界的物体：

www.TopSage.com

```
if(sprite.x - sprite.width / 2 > right ||
   sprite.x + sprite.width / 2 < left ||
   sprite.y - sprite.height / 2 > bottom ||
   sprite.y + sprite.height / 2 < top)
{
    // 重置精灵的位置和速度。
}
```

对超过边界的物体进行屏幕折回：

```
if(sprite.x - sprite.width / 2 > right)
{
   sprite.x = left - sprite.width / 2;
}
else if(sprite.x + sprite.width / 2 < left)
{
   sprite.x = right + sprite.width / 2;
}
if(sprite.y - sprite.height / 2 > bottom)
{
   sprite.y = top - sprite.height / 2;
}
else if(sprite.y + sprite.height / 2 < top)
{
   sprite.y = bottom + sprite.height / 2;
}
```

169

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;
```

应用摩擦力（容易方法）：

```
vx *= friction;
vy *= friction;
```

6.4　小结

本章涉及了一个物体与它自身的环境的交互——确切地说，一个物体与它的领域的边界和它的领域自身的交互。你已经学习了处理一个物体离开边界后的各种方法，包括移除、重新生成、折回和回弹。并且，你现在知道了可能比你之前想知道的更多的关于摩擦力的知识。而使用这些简单的技术，你可以使你影片中的物体移动起来更具有真实感。在下一章中，你将研究允许用户与物体对象进行交互操作。

170

www.TopSage.com

第7章

用户交互：移动物体

本章内容

- 按下和放开精灵
- 拖动精灵
- 抛

很可能你的动画的首要目标就是允许与用户之间光滑的交互，而许多交互都是通过鼠标来完成的。我们在前面的第2章中讨论过鼠标事件，但是你并没有真正使用它来做些什么。现在，你将动手来实践它们。

在这一章中，你将迈出用户交互的第一步。你将学到如何控制拖动、落下和抛。但是首先，我们从最基本的按下与放开开始。

7.1 按下和放开精灵

鼠标是一个非常酷的发明，但是从本质上讲，它是一个简单的设备。它实际上只能做两件事情：检测运动和按钮的点击。计算机则使用那些信息来做更多的事情：始终跟踪鼠标光标的位置，当点击事件发生时判断光标所在的位置，判断鼠标移动得有多快，并要弄清双击什么时候发生。但实际上，当你查看它的事件时，所有这些可归结为点击和移动。（当然，现在的一些鼠标有了滚轮和轨迹球，和比一个便宜的手机上还多的按钮，但我们这里所要说的是最基本的那种鼠标。）

你可以把点击事件分成两部分：鼠标按下时的事件和放开时的事件。有时候，这些事件几乎是同时发生的。另外一些时候，这些事件被时间和运动分离开来，这通常被解释为*拖动*（drag）——点击，移动，然后放开。在这一章中，你将集中精力在3件事情上：鼠标按下、鼠标放开和两者之间的发生的任何运动。

鼠标处理在AS 3中有了很大的改变，虽然这需要一段时间去重新学习基础，但总体来看，它是值得的。AS 3中事件的整体架构是经过深思熟虑和有逻辑性的。不像之前版本中给人一种黑匣子的感觉。

基本上，如果鼠标光标位于精灵、影片剪辑或其他的交互对象的可视图形部分，则鼠标事件可以被该对象接收。在AS 2中，这对于一些鼠标事件来说是成立的，但对其他的不成立。这经常

www.TopSage.com

会引起很多迷惑。它也使得多重事件如onRelease和onReleaseOutside的处理成为必要，对于这些事件你往往只想让它们做同样的事情，但不得不分开处理。

另一件彻底改变的事情与鼠标事件和嵌套对象相关。在AS 2中，如果你侦听一个影片剪辑的鼠标事件，那么就没有办法侦听这个影片剪辑内部的任何子影片剪辑的鼠标事件。外面的影片剪辑会截获所有的鼠标事件而不会让它们传到内部。在AS 3中就没有这种限制。你可以侦听一个影片剪辑或一个精灵上的鼠标事件，它们的嵌套对象也同样可以侦听。

你需要知道的主要鼠标事件有mouseDown、mouseUp和mouseMove。它们被定义成MouseEvent类的静态属性：

- MouseEvent.MOUSE_DOWN
- MouseEvent.MOUSE_UP
- MouseEvent.MOUSE_MOVE

174

当你的鼠标指针位于一个精灵、影片剪辑或其他交互对象的可视的图形部分时，当按下鼠标按钮时，mouseDown事件发生。它本质上与AS 2中的onPress事件是一样的。

当你在精灵等的图形上放开放鼠标按钮时，mouseUp事件发生，它等同于onRelease。

mouseMove事件发生在你移动鼠标时——但是只有当你在精灵或其他对象上移动它时才会发生。这与AS 2有很大的不同，在AS 2中只要你的鼠标在移动，onMouseMove事件对于任何影片剪辑都会发生，而与指针的位置是无关的。

然而，有时候，你只是想让这些鼠标事件起作用而不管指针的位置在哪里。这种情况可以在AS 2中使用onMouseMove、onMouseUp和onMouseDown来完成，它不会在意鼠标的位置。而在AS 3中，没有与上面等同的方法，但是你可以在舞台上侦听mouseDown、mouseUp和mouseMove事件，在这种情况下，这些鼠标事件会被激活而不管图形内容是什么。

好，已经有足够多的讨论了，我们来实践一下。本章中的第1个例子——文档类MouseEvent.as仍使用前面章节中的Ball类。下面是代码：

```
package
{
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class MouseEvents extends Sprite
    {
        public function MouseEvents()
        {
            init();
        }

        private function init():void
        {
            var ball:Ball = new Ball();
            ball.x = 100;
            ball.y = 100;
            addChild(ball);
```

www.TopSage.com

```
            ball.addEventListener(MouseEvent.MOUSE_DOWN,
                                  onMouseDownBall);
            ball.addEventListener(MouseEvent.MOUSE_UP,
                                  onMouseUpBall);
            ball.addEventListener(MouseEvent.MOUSE_MOVE,
                                  onMouseMoveBall);

            stage.addEventListener(MouseEvent.MOUSE_DOWN,
                                   onMouseDownStage);
            stage.addEventListener(MouseEvent.MOUSE_UP,
                                   onMouseUpStage);
            stage.addEventListener(MouseEvent.MOUSE_MOVE,
                                   onMouseMoveStage);
        }

        private function onMouseDownBall(event:MouseEvent):void
        {
            trace("mouse down - ball");
        }

        private function onMouseUpBall(event:MouseEvent):void
        {
            trace("mouse up - ball");
        }

        private function onMouseMoveBall(event:MouseEvent):void
        {
            trace("mouse move - ball");
        }

        private function onMouseDownStage(event:MouseEvent):void
        {
            trace("mouse down - stage");
        }

        private function onMouseUpStage(event:MouseEvent):void
        {
            trace("mouse up - stage");
        }

        private function onMouseMoveStage(event:MouseEvent):void
        {
            trace("mouse move - stage");
        }

    }
}
```

这里只是设置了我们一直讨论的3种鼠标事件的处理器，先是在小球上设置，然后在舞台上设置。它可以让你知道它们是什么时发生的。你可以创建这个文件并且试验它来观看特定的鼠标事件究竟在哪个位置和什么时间发生的，直到完全清楚了。这里有一些内容是要注意的：

- 小球的事件只当鼠标的指针位于小球上面时才会发生。

www.TopSage.com

大家网 TopSage.com

- 特别要注意小球的mouseMove事件，只有当鼠标在小球上移动时，事件才会发生。
- 不管鼠标在哪儿都会获取舞台事件——即便是它在小球上。在这样的情况中，你将获取两个事件——一个是小球上的事件，另一个是舞台的事件。
- 没有先获取mosueDown事件就不可能获取mouseUp事件。

176

既然你已经知道了几个比较重要事件的基础，我们接下来将探讨拖动。

7.2 拖动精灵

你可以使用两种不同的技术来处理拖动精灵：使用mouseMove事件或使用startDrag/stopDrag方法。尽管在本书的示例中将使用后者来完成示例，但是先让我们看一下如何使用mouseMove事件进行拖动。它将会给你一些处理mouseMove事件的经验并让你深入理解事件是怎样工作的。

7.2.1 使用 mouseMove 拖动

通过手工处理mouseMove事件，你可以在每次鼠标移动时，更新精灵的位置来匹配鼠标光标的位置。但值得注意的是这是一次拖动多个物体的唯一途径。你有时候会在精灵之外自制一个鼠标光标时遇到这种情况。这个光标需要跟随鼠标的位置，但是如果你还想拖动某个东西怎么办呢？一个解决办法就是对自制的鼠标光标使用mouseMove，然后留着常用的拖动方法(startDrage/stopDrag)用于正常的拖动。所以它是一个非常漂亮的小技术，即使你暂时用不上它。

思路是这样的：在onmouseDown事件中，你想设置一个mouseMove处理器。这个处理器将只是设置小球的x和y位置以等于当前鼠标的位置。在mouseUp事件中，你移除掉这个处理器。

现在这儿是技巧部分。你需要在你要拖动的精灵上设置mouseDown侦听器，但是对于其他两个事件，你要在舞台上侦听它们。这么做的原因是你可能会出现为了将精灵保持在鼠标下而使拖动的速度比Flash更新精灵位置的速度快的情况。在那时，如果你正在侦听精灵上的mouseUp和mouseMove事件，事件将永远不会被触发，因为鼠标指针不再在精灵上了。然而，记住，不管鼠标在哪里舞台都会响应这些事件。

下一个文档类MouseMoveDrag.as可以使上面说的问题更清晰：

```
package
{
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class MouseMoveDrag extends Sprite
    {
        private var ball:Ball;

        public function MouseMoveDrag()
        {
            init();
        }

        private function init():void
```

www.TopSage.com

177

```
        {
            ball = new Ball();
            ball.x = 100;
            ball.y = 100;
            addChild(ball);
            ball.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
        }

        private function onMouseDown(event:MouseEvent):void
        {
            stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
            stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
        }

        private function onMouseUp(event:MouseEvent):void
        {
            stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
            stage.removeEventListener(MouseEvent.MOUSE_MOVE,
                                      onMouseMove);
        }

        private function onMouseMove(event:MouseEvent):void
        {
            ball.x = mouseX;
            ball.y = mouseY;
        }
    }
}
```

开始，你只是想侦听mouseDown事件，并且只是在小球上。onMouseDown方法在舞台上加入了mouseUp和mouseMove事件的侦听器。onMouseMove方法更新小球的位置来匹配鼠标的位置。onMouseUp方法从舞台上移除mouseUp和mouseMove侦听器，因为你只在拖动过程中需要它们。

使用这种设置时你可能已经发现了一个问题。如果你点击的是小球的边缘然后开始拖动它，你将会看到它突然跳跃使自身的中心与鼠标光标对齐。这是因为你设置了小球的*x*和*y*位置精确地等于鼠标光标的位置。你可以在其中加入一些表现，当鼠标按下时找到鼠标与小球之间的偏移距离，然后在拖动时将它加到小球的位置中。我将把它作为一个项目留给你，如果你有兴趣可以继续完成它。而我将继续讲解拖动精灵的常用方法。

7.2.2 使用 startDrag/stopDrag 拖动

所有的精灵和影片剪辑都有内置的方法startDrag和stopDrag，它们处理我刚才描述的方法的所有功能，大概唯一的缺点就是它每次只允许拖动一个物体。

178

这个概念很简单。在mouseDown处理器中，调用startDrag。在mouseUp处理器中，调用stopDrag。

startDrag可以不使用任何参数来调用。可以以上次文档类为基础，将onMouseDown和onMouseUp方法修改成如下（如文档类Drag.as中所示）：

www.TopSage.com

```
private function onMouseDown(event:MouseEvent):void
{
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    ball.startDrag();
}

private function onMouseUp(event:MouseEvent):void
{
    stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    ball.stopDrag();
}
```

你也可删去onMouseMove方法，因为它已经不需要了。

相当简单，是吗？测试一下，你将会发现原来的“突然跳跃”问题不见了。不管你单击的是小球的哪个位置它都会被拖动。但是，如果想让光标对齐中心，那么，可以通过把true传递给startDrag方法的第1个参数来完成此操作：

```
ball.startDrag(true);
```

你也可以通过传递区域的左、上、右和下的坐标来限制拖动的矩形区域。在AS 3中这一点有了一些改变，现在传入的是一个单一的Rectangle对象（一个flash.geom.Rectangle类的实例——不要忘记导入它）。你恰好可以把这4个值传递到Rectangle构造函数中，像下面这样：

```
var rect:Rectangle = new Rectangle(10, 10, 200, 200);
```

这里是一个完整的startDrag语法：

```
startDrag(lockCenter, boundsRectangle)
```

试着使用下面的拖动代码修改上一个示例：

```
ball.startDrag(false, new Rectangle(100, 100, 300, 300));
```

另外我再次提到：不要忘记先导入Rectangle类！

7.2.3　拖动与运动代码结合

目前你知道了关于在Flash中简单的拖动和放开的具体内容。但是，在这个过程中，你已经回到一个静态的物体上，它停在那里直到你拖动它。让我们加入一些速度，也许还有一些加速度和回弹。

179

你已经在前面章节的Bouncing2.as文档类中设置好了速度、重力和回弹。这是一个很好的起点。在逻辑上我们只需要加入拖放代码即可。让我们试一下。你需要做如下的内容（DragAndMove1.as）：

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class DragAndMove1 extends Sprite
```

www.TopSage.com

```
{
    private var ball:Ball;
    private var vx:Number;
    private var vy:Number;
    private var bounce:Number = -0.7;
    private var gravity:Number = .5;

    public function DragAndMove1()
    {
        init();
    }

    private function init():void
    {
        stage.scaleMode = StageScaleMode.NO_SCALE;
        stage.align=StageAlign.TOP_LEFT;

        ball = new Ball();
        ball.x = stage.stageWidth / 2;
        ball.y = stage.stageHeight / 2;
        vx = Math.random() * 10 - 5;
        vy = -10;
        addChild(ball);
        ball.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        vy += gravity;
        ball.x += vx;
        ball.y += vy;
        var left:Number = 0;
        var right:Number = stage.stageWidth;
        var top:Number = 0;
        var bottom:Number = stage.stageHeight;
        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;
```

180

www.TopSage.com

```
                vy *= bounce;
            }
        }

        private function onMouseDown(event:MouseEvent):void
        {
            stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
            ball.startDrag();
        }

        private function onMouseUp(event:MouseEvent):void
        {
            stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
            ball.stopDrag();
        }
    }
}
```

可以看到，加到原始代码中的是onMouseDown和onMouseUp处理器。

如果现在就测试它，当开始拖动时，你将会立即发现问题。的确，拖动是可以工作的，但是同时运动代码也在工作。结果感觉就像小球要从我们的手中滑出。你需要某种打开或关闭运动代码的方法，以使这种现象在拖动时不会发生。最简单的方法是当你开始拖动时移除掉EnterFrame
181 事件，当你停止拖动时重置它。

这只需要在onMouseDown和onMouseUp方法中加入额外的几行。

```
private function onMouseDown(event:MouseEvent):void
{
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    ball.startDrag();
    removeEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onMouseUp(event:MouseEvent):void
{
    stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    ball.stopDrag();
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

测试时，你将会发现已经很接近了，但是并不是十分完美。

当开始拖动时，所要做的就是拖动。当你放开小球时，运动代码会在它停止的位置继续。现在的主要问题是速度也会继续保持停止时的速度。当你放开小球时这有时会导致小球飞向其他方向，这是不正常的。你可以很容易地通过设置vx和vy为0来解决这个问题，在开始拖动或停止拖动时设置。只要在运动代码继续之前执行，在哪儿设置都可以工作。让我们将它加到开始拖动代码段中：

```
private function onMouseDown(event:MouseEvent):void
{
    vx = 0;
    vy = 0;
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
```

www.TopSage.com

TopSage.com

```
        ball.startDrag();
        removeEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
```

解决了上面的问题，那么你就得到一个完整功能的拖放特性，集成了速度、加速度和回弹。你可在文档类DragAndMove2.as中看到全部的代码。

只是还存在一个问题：当你放开小球时，它垂直下落——不再是*x*轴上的运动。尽管这是正确的行为，还是有点儿令人厌烦。但如果你能真正地把球抛出去并且让它沿着你抛出去的方向飞出，那这将是一个很完美的交互。好，这就是我们下面将要做的。

7.3 抛

在Flash影片中抛（throwing）是什么意思呢？它意味着你点击一个物体开始拖动它，并沿着一个特定的方向移动它，然后当放开它时，它保持原有拖动的方向继续移动。 182

对于速度，需要确定物体被拖动时的速度，然后当它下落时要将原来的速度赋给物体。换个方式说，如果你以每帧10像素向左拖动一个精灵，那么当使它下落时，它的速度为vx=–10。

设置新的速度对你来说应没有什么问题，只是指定新的值给*vx*和*vy*，如图7-1所示。但是测定它们的值可能有一点技巧性。实际上，计算拖动速度几乎与你在运动代码中应用速度相反。在运动代码中应用速度时，你加入速度到物体的旧的位置以生成物体的新的位置。那么公式将为old + velocity = new。当一个物体正在被拖动时，要测定它的速度，你只是重新安排这个等式得到velocity = new–old。

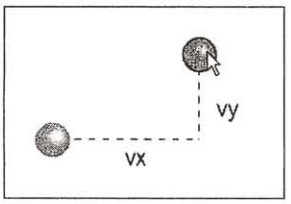


图7-1 小球已被拖到一个新的位置。速度是指它的上一个位置到这个新位置

当你拖动一个精灵时，它在每一帧上都将会有一个新的位置。如果将它与前一帧的位置相减，你将知道在一帧时间里它移动了多远。这就是你的像素每帧（pixels-per-frame）速度！

让我们来看一个例子，将它简化到一个轴上。一个精灵正在被拖动。在一个特定的帧上，你注意到它的*x*位置为150。在下一帧上，你看到它的*x*轴位置是170。因此，在一帧内，它在*x*轴位置上被拖动了20个像素，它在这一点上的*x*速度为+20。如果这时放开它，你将可以预料到它将仍以*x*速度为20继续移动。那么，当然你需要设置vx =20。

这需要在你已经有的类中修改两处。首先，你需要跟踪每一帧的速度，所以将需要一个enterFrame事件的处理器。但是因为你在每一帧上所做的事件与当前的onEnterFrame方法所做的事情有很大的不同。所以我们创建一个全新的方法来跟踪拖动速度。事实上，我们称它为trackVelocity。你还需要两个变量来保存旧的*x*和*y*位置值。你可以称它们为oldX和oldY，并且声明它们为类的变量。现在，只要你开始拖动，你就想注意小球的位置并且将它保存在oldX和oldY变量中。然后，当你移除了enterFrame事件的处理器onEnterFrame之后，需要加入trackVelocity作为侦听器。所有这些都是在onMouseDown方法中完成的，如下：

```
private function onMouseDown(event:MouseEvent):void
{
```

www.TopSage.com

```
        oldX = ball.x;
        oldY = ball.y;
        stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
        ball.startDrag();
        removeEventListener(Event.ENTER_FRAME, onEnterFrame);
        addEventListener(Event.ENTER_FRAME, trackVelocity);
    }
```

然后，在trackVelocity方法中，你只需要从当前的*x*位置减去oldX，从当前的*y*位置减去oldY。这样就得到了当前的速度，因此可以将这些值直接存储在vx和vy中。然后你可以重置oldX和oldY为当前小球的位置。

183

```
    private function trackVelocity(event:Event):void
    {
        vx = ball.x - oldX;
        vy = ball.y - oldY;
        oldX = ball.x;
        oldY = ball.y;
    }
```

最后，当放开小球时，需要再次切换enterFrame事件，这次移除trackVelocity并且重新应用onEnterFrame：

```
    private function onMouseUp(event:MouseEvent):void
    {
        stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
        ball.stopDrag();
        removeEventListener(Event.ENTER_FRAME, trackVelocity);
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
```

此时对于速度，你不需要做任何事情。在拖动过程中它会一直被跟踪，并且最新的速度已经保存在vx和vy变量中。只要重新应用运动代码，小球会以你刚刚拖动时的速度继续移动。结果是：你抛出了这个小球！

下面是这个例子的最终文档类（Throwing.as）：

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class Throwing extends Sprite
    {
        private var ball:Ball;
        private var vx:Number;
        private var vy:Number;
        private var bounce:Number = -0.7;
        private var gravity:Number = .5;
        private var oldX:Number;
        private var oldY:Number;
```

www.TopSage.com

```
public function Throwing()
{
    init();
}
private function init():void
{
    stage.scaleMode = StageScaleMode.NO_SCALE;
    stage.align=StageAlign.TOP_LEFT;

    ball = new Ball();
    ball.x = stage.stageWidth / 2;
    ball.y = stage.stageHeight / 2;
    vx = Math.random() * 10 - 5;
    vy = -10;
    addChild(ball);
    ball.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    vy += gravity;
    ball.x += vx;
    ball.y += vy;
    var left:Number = 0;
    var right:Number = stage.stageWidth;
    var top:Number = 0;
    var bottom:Number = stage.stageHeight;

    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;
    }
}
private function onMouseDown(event:MouseEvent):void
{
    oldX = ball.x;
    oldY = ball.y;
```

184

185

www.TopSage.com

```
            stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
            ball.startDrag();
            removeEventListener(Event.ENTER_FRAME, onEnterFrame);
            addEventListener(Event.ENTER_FRAME, trackVelocity);
        }

        private function onMouseUp(event:MouseEvent):void
        {
            stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
            ball.stopDrag();
            removeEventListener(Event.ENTER_FRAME, trackVelocity);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function trackVelocity(event:Event):void
        {
            vx = ball.x - oldX;
            vy = ball.y - oldY;
            oldX = ball.x;
            oldY = ball.y;
        }
    }
}
```

现在它已经是一个相当好的交互程序了，一个对真实世界中的物理现象的重建。它确实像是你向某个方向抛出某个物体。你甚至可以和你自己玩传球的游戏。你可以试验所有的变量，如重力、回弹等等。你可以加入一点摩擦力来模拟一些环境。这是一个可以让你感受到所有这些事物是如何工作的非常好的示例文件，因为你可以实时做试验来看所有这些参数是如何影响运动的。

7.4　小结

这一章并不长，但它涉及了一些非常有价值的内容，并进一步深入了交互。到现在为止，你应该可以拖动任何物体、使它下落和将它抛出去了。

最重要的是，你已经使用了用来完成一个真正专业的交互作品的很多小细节进行操作。在后面的章节中，你将会看到允许用户与影片中的物体交互的很多其他的方法。难点是我们的进度会
186 更快一些，但是如果你已经掌握了这些基础，你将会做得很好。

www.TopSage.com

Part 3 第三部分

高级运动

本部分内容

- 第 8 章　缓动和弹性
- 第 9 章　碰撞检测
- 第 10 章　坐标旋转和角度回弹
- 第 11 章　台球物理
- 第 12 章　粒子引力和重力
- 第 13 章　正向运动：行走
- 第 14 章　反向运动：拖动和伸展

www.TopSage.com

大家网
TopSage.com

第8章 缓动和弹性

本章内容

- 比例运动
- 缓动
- 弹性
- 本章重点公式

很难相信我们用了7章来学习“基础内容”，这里是第8章，高级内容的开始。或者，只是我这样想的，关键是从这里开始的内容会更有趣。在这之前，每一章中都覆盖了基本的技术和概念。从这一章开始，我将在每一章中集中讲解一种或两种特殊形式的运动。

本章将涉及缓动（比例速度）和弹性（比例加速度）。但不要认为只有两项内容，就可以快速地略读它。通过这两项技术我得到了比其他任何内容更多的收益。并且我将带你完成大量的示例，以使你体会到这两项技术有多么强大。

8.1 比例运动

缓动（easing）和弹性（springing）有着紧密的联系。尽管一开始我计划在同一章中将它们设定为两个主题内容，但直到我坐下来详细的规划它们的内容时，我才意识到它们两者是多么相似。

这两种技术都涉及使一个物体（通常是指一个精灵或影片剪辑）从现有的位置移动到目标位置。在缓动中，精灵以滑动的方式到达目标点并停止。在弹性中，它有一些来回弹跳的动作，最终停在目标位置。这两种技术有下面的公共特点：

- 需要设定一个目标位置；
- 需要检测出距目标点的距离；
- 运动是与距离成正比的——距离越大，运动的强度就越大。

缓动与弹性的区别在于运动的哪个方面是成比例的。在缓动中，速度与距离成正比，距离目标点越远，物体移动得就越快。当它非常非常接近物体时，它几乎不移动。

在弹性中，加速度与距离成正比。如果物体与目标点离得很远，就会被指定一个很大的加速度，速度会很快增加。当物体接近它的目标点时，加速度就会很小，但它仍在加速！它会穿过目

www.TopSage.com

标，然后加速度再将它拉回来。最后，摩擦力会使它停下来。

让我们深入地研究每一种技术，先从缓动开始。

8.2 缓动

有一件事需要马上弄清楚，即缓动并不只有一种类型。甚至在Flash IDE中，当使用补间做动画时，你有能力进行“缓入”和“缓出”，这里我们将要讨论的缓动类型如同补间动画中的“缓出”。在8.2.5节中，将提供一个链接，在那里可以找到如何实现各种缓动。 190

8.2.1 简单的缓动

简单的缓动是一个非常基础的概念。你有一个物体在这里，并且你想将它移动到那里。因为你要创建“运动错觉”，所以你想渐进地移动它，通过几帧来移动。你可以简单地找到两点之间的角度，设定一个速率，使用某个三角函数来计算出*vx*和*vy*，然后在运动中设定它们。那么你就可以检测出每帧距目标点的距离（使用毕达哥拉斯定理，在第3章我们学习过），当它到了那里，停止它。这种方法对于某些情况来说可能已经足够了，但如果想让某物体在移动时看上去很自然，上面的方法是做不到的。

问题在于物体将以一个恒定的速度移动，到达目的地，然后停在那里。如果正在讨论的是物体移动后碰到砖墙上，那么它可能是相似的。但是正在将一个物体移动到一个目标点（target），这通常意味着某个人或某物体知道目标点在哪里，并正在移动某物体到设定好的位置。在这样的例子中。运动将以非常快的速度开始，然后在接近目标点时慢下来。换句话说，它的速度将与到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。当距离你家还有几英里远时，你将会以所允许的尽可能高的速度驾驶（好，甚至可能会更快——甚至可能足够快以至于会收到一张罚款单）。当你下了高速公路进入城区时，将会慢下来一些。一旦进入去往你家里的街道，只有一两个街区远时，你将会更慢一些。当接近你的车道时，你的速度会降到每小时几英里。当还有几英尺远的时候，你的速度要比刚才进入车道的速度还要慢。在距你停下来只有几英寸距离时，你将以很小很小的速度移动。

如果你愿意花时间去看一下，将发现这种行为本身与关闭一个抽屉或门这些小事情是非常相似的。开始的速度快然后渐渐慢下来。如果下次关门，尽量使用与开始一样的速度去关门。那么你要做好准备向周围的人解释为什么你这样砰地关门。

那么，当你使用缓动移动一个物体到一个位置时，它会自动呈现出一个非常自然的外观。其中最酷的事情是简单的缓动是非常容易做到的。实际上，它可能比计算角、*vx*和*vy*和以恒定的速度移动更容易。

这里是缓动的思路：

(1) 设定一个比例系数。它将是一个比1小的数值。

(2) 指定你的目标点。

(3) 计算物体到目标点之间的距离。

www.TopSage.com

(4) 将距离与比例系数相乘。这是你的速度。

191 (5) 加入速度值给当前物体的位置。

(6) 重复第3步到第5步直到物体到达目标点。

图8-1阐述了这个概念。

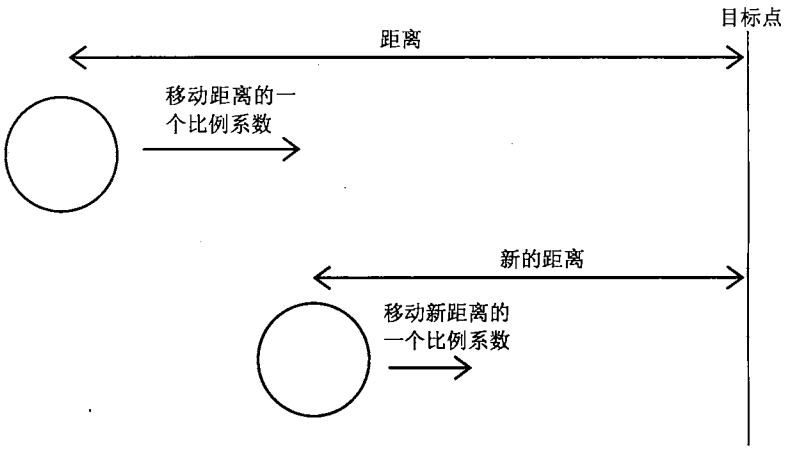


图8-1　基本的缓动

让我们一个一个地来看这些步骤，看它们在ActionScript中是如何应用的。不要担心代码放在哪里。我只是告诉你这些代码看上去是什么样以及它们是什么意思。

首先，设定一个系数来表示比例。如我所说过的，速度将与运动成正比。特别地，这意味着速度将是距离的一个比例系数，0到1之间的数值。越接近于1，物体移动得越快。越接近于0，移动得越慢，但需要注意，因为太小的值将会阻止物体到达目标点。对于初学者来说，选择值0.2。我将把这个变量称为easing。因此你的代码如下：

```
var easing:Number = 0.2;
```

接下来，检测你的目标点。这只是简单的*x*，*y*位置。如果没有什么更好的地方你可以指定它为舞台中心。

```
var targetX:Number = stage.stageWidth / 2;
var targetY:Number = stage.stageHeight / 2;
```

然后计算到目标点之间的距离。假设有一个精灵名为ball，你只需要用目标点的*x*和*y*值减去小球的*x*和*y*值即可。

```
var dx:Number = targetX - ball.x;
var dy:Number = targetY - ball.y;
```

那么速度就是距离乘以比例系数：

```
vx = dx * easing;
```

192

```
vy = dy * easing;
```

从这儿开始你就知道如何做了：

www.TopSage.com

```
ball.x += vx;
ball.y += vy;
```

第3步到第5步需要重复，因此它们将在enterFrame处理器中进行。让我们更近一步来看这3个步骤，因为它们可以在很大程度上简化：

```
var dx:Number = targetX - ball.x;
var dy:Number = targetY - ball.y;
vx = dx * easing;
vy = dy * easing;
ball.x += vx;
ball.y += vy;
```

你可以非常容易地将前两行压缩成另外两行：

```
vx = (targetX - ball.x) * easing;
vy = (targetY - ball.y) * easing;
ball.x += vx;
ball.y += vy;
```

或者，如果想更简洁，可以将它们压缩得更短：

```
ball.x += (targetX - ball.x) * easing;
ball.y += (targetY - ball.y) * easing;
```

前几次制作缓动，你可能想让你的代码稍微啰嗦一点以使它看上去更清晰。但是一旦你做过几百次之后，第3种方式是非常简洁适用的。这里我将使用第2种方式，目的只是在你处理速度时加深印象。

现在，让我们看它的动作脚本。仍需要你一直使用的Ball类。这里是文档类Easing1.as：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Easing1 extends Sprite
    {
        private var ball:Ball;
        private var easing:Number = 0.2;
        private var targetX:Number = stage.stageWidth / 2;
        private var targetY:Number = stage.stageHeight / 2;

        public function Easing1()
        {
            init();
        }
        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
```

193

www.TopSage.com

```
            var vx:Number = (targetX - ball.x) * easing;
            var vy:Number = (targetY - ball.y) * easing;
            ball.x += vx;
            ball.y += vy;
        }
    }
}
```

试着使用不同的easing值来观看它是如何影响小球的运动的。

接下来你可能想做的事情是使小球变成可拖动的，这样你可以四处移动它并且看它是如何总是回到目标点的。这非常类似于第7章中使用的拖放（drag-and-drop）技术。这里，当按下鼠标时开始拖动小球。同时移除enterFrame事件处理器并且侦听舞台上的mouseUp。在mouseUp处理器中，你停止拖动，移除mouseUp处理器，然后重新启用enterFrame。这里是文档类Easing2.as：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class Easing2 extends Sprite
    {
        private var ball:Ball;
        private var easing:Number = 0.2;
        private var targetX:Number = stage.stageWidth / 2;
        private var targetY:Number = stage.stageHeight / 2;

        public function Easing2()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onMouseDown(event:MouseEvent):void
        {
            ball.startDrag();
            removeEventListener(Event.ENTER_FRAME, onEnterFrame);
            stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
        }

        private function onMouseUp(event:MouseEvent):void
        {
            ball.stopDrag();
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
        }
```


www.TopSage.com

TopSage.com

```
        private function onEnterFrame(event:Event):void
        {
            var vx:Number = (targetX - ball.x) * easing;
            var vy:Number = (targetY - ball.y) * easing;
            ball.x += vx;
            ball.y += vy;
        }
    }
}
```

8.2.2 何时停止缓动

如果你正在向一个目标点做简单的缓动，那么最终你将到达目标点并且缓动的目的已经达到。但是，到目前为止，缓动代码仍在继续执行，尽管物体移动得并不是很明显。如果你只是缓动物体到目标点并把物体留在那里，继续运行缓动代码实际上是浪费CPU资源。如果你已经达到了目的，不妨想办法让缓动停下来。乍一看，这好像只是简单地检测物体是否到了目标点然后关闭enterFrame代码，如下：

```
if(ball.x == targetX && ball.y == targetY)
{
    // 停止缓动的代码。
}
```

但是，它并不是这么简单的。

我们正在讨论的缓动的类型涉及芝诺悖论。芝诺是一位希腊人，他喜欢测量。芝诺试图这样质疑运动理论：要使某物体从A点移动到B点，它首先必须要移动到A和B之间的中点上。然后再从这一点移动到与B点之间的中点上。如此这样一直移动下去。因为你总是要先移动到目标点一半的距离，所以你永远不可能到达目标点。

这个悖论听上去很合理，但是很明显我们每天都在做从一点到另一点的移动，因此一定有些错误在里面。让我们看一下它在Flash中是怎样的。在*x*轴上，一个精灵在位置0处。假设你想将它 195
在*x*轴上移动到100的位置。应用我们刚才说到的悖论，使easing变量为0.5，因此它总是移动距目标点一半的距离。它会像下面这样前进：

- 开始在0的位置，一帧之后，它将到达50。
- 第2帧它将到达75。
- 现在距离是25。它的一半是12.5，因此新的位置将是87.5。
- 按着这样的方式，位置将是93.75，96.875，98.437 5，等等。20帧之后，它将是99.999 809 265。

如你所看到的，它越来越接近目标但永远不能真正地到达目标。然而，当你检查代码做了什么时事情会有一些不同。它带来的问题是，“你能把一个像素分成多少份?”答案是20。实际上，一个像素的二十分之一有一个名字：*twip*（缇）。在Flash内部，Flash计算任何内容都使用twip。这包括舞台上的精灵、影片剪辑和其他任何物体的位置。因此，如果你跟踪一个精灵的位置，你可能会注意到它总是0.05的倍数。

这里有一个这样的例子，一个精灵目标点是100但实际上它到的位置只能是99.95。当你试图

www.TopSage.com

在这一点上再次进行分割时，你试图加上（100–99.95）/2。它得到的是0.025，或者说是四十分之一像素。但是一个twip是你可以移动的最小值。你不能再加入一个twip的一半了，因此你最后加到精灵位置上的是0。如果你不相信我，可以试验下面的代码（提示：将下面的代码放在第2章中提到的类架构中的init方法中）：

```
var sprite:Sprite;
sprite = new Sprite();
addChild(sprite);
sprite.x = 0;
var targ:Number = 100;
for(var i:Number=0;i<20;i++)
{
    trace(i + ": " + sprite.x);
    sprite.x += (targ - sprite.x) *0.5;
}
```

这只是一个20次的循环，每次将精灵移动到距离目标点一半的位置。它是基本的缓动代码。我之所以把它放在一个for循环中因为我只是对跟踪它的位置有兴趣，而不是看它的动画效果。你将发现的是到第11次循环时，小球就已经到达99.95的位置了，这是它所能到达的最远的位置。

> 在跟踪位置的过程中，你可能也会注意到偶尔会得到像96.850 000 000 000 01这样的值。这与以二进制格式存储小数的方式有关，而与像素、twip或Flash自身无关。要得到更多的信息，你可以在网上搜索“二进制舍入错误（binary round-off error）”。

长话短说，精灵不再是越来越接近了，但是正确的是，精灵可能永远不能到达目标地点。所以，如果做一个简单的比较，以之前的例子为例，你的缓动代码将永远不会关闭。你需要回答这
196 样一个问题，“什么时候可以关闭缓动？”这实际上就是检测距目标点的距离是否小于某一个数值。对于多数应用程序来说，我已经发现如果一个物体距离它的目标点小于1个像素，它可以被认为是已经到达了，那么我可以关闭缓动了。

如果你正在处理两个方向上的缓动，你可通过第3章中介绍的公式来计算距离：

```
distance = Math.sqrt(dx * dx + dy * dy)
```

如果你正在处理一个的距离值，如当你只在一个轴上移动物体时，需要使用距离的绝对值，因为它可能是负值。可以使用Math.abs方法来使距离成为正值。

好，不管怎样，我已经讨论或写了很多了。让我们来看一些代码。这是一个用来演示关闭缓动的简单的文档类（EasingOff.as）：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class EasingOff extends Sprite
    {
        private var ball:Ball;
        private var easing:Number = 0.2;
        private var targetX:Number = stage.stageWidth / 2;
```

www.TopSage.com

```
        public function EasingOff()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.y = stage.stageHeight / 2;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }


        private function onEnterFrame(event:Event):void
        {
            var dx:Number = targetX - ball.x;
            if(Math.abs(dx) < 1)
            {
                ball.x = targetX;
                removeEventListener(Event.ENTER_FRAME, onEnterFrame);
                trace("done");
            }
            else
            {
                var vx:Number = dx * easing;
                ball.x += vx;
            }
        }
    }
}
```

197

如你所看到的，这个示例对缓动公式进行了一点扩展，先计算了距离，因为你需要用距离来判断是否需要停止缓动。可能你现在已经明白了为什么要使用dx的绝对值。如果小球在目标点的右侧，dx将是一个负值，那么if(dx<1)语句将为真，那么就无法正确地应用缓动。通过使用Math.abs方法，可以确保小球距目标点的真正距离小于1。然后你将小球放到它所要去的位置并停止执行运动代码。

记住，如果你在做些像是拖放操作的缓动，当小球被释放时需要重新应用运动代码。为什么不试一下看自己能否解决缓动这个问题。

8.2.3 移动的目标

到目前为止的示例中，目标点都是单一的、固定的位置，但这并不是必需的。先计算每一帧上的距离，然后在距离的基础上计算速度。Flash并不关心它是否到达了目标点，或目标点是否在移动。Flash只是在每一帧上说，“我的目标点在哪儿？距离是多少？我的速度是多少？”

可以很容易地将鼠标作为一个缓动目标。只需要把鼠标的坐标（mouseX和mouseY）加到targetX和targetY之前。这儿有一个简单的例子（EaseToMouse.as）：

www.TopSage.com

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class EaseToMouse extends Sprite
    {
        private var ball:Ball;
        private var easing:Number = 0.2;

        public function EaseToMouse()
        {
            init();
        }

        private function init():void
        {
198         ball = new Ball();
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }


        private function onEnterFrame(event:Event):void
        {
            var vx:Number = (mouseX - ball.x) * easing;
            var vy:Number = (mouseY - ball.y) * easing;
            ball.x += vx;
            ball.y += vy;
        }
    }
}
```

到处移动鼠标来看小球是如何跟随的，而且当你将鼠标移得更远时小球是如何变得更快的。

想一下你可能会有什么其他的移动目标。一个精灵可能可以缓动到另一个精灵的位置上。回到早期的Flash上，鼠标跟随——一系列跟随在鼠标的周围的影片剪辑——十分流行。缓动是制作它的方法之一。第1个影片剪辑缓动到鼠标的位置，第2个影片剪辑缓动到第1影片剪辑的位置，第3个缓动到第2个位置上，依此类推。实现它并对它进行试验（但是如果你把它放到了直播网站上，我将拒绝承担一切责任）。

8.2.4　缓动不只是应用于运动

整本书中始终贯穿的内容就是我给出的示例。对于每一个示例，主要是操作用作精灵或其他物体的属性的数。通常，使用*x*和*y*属性来控制精灵的位置。记住，精灵、影片剪辑和其他的显示对象还有很多其他的属性你可以操作，并且多数属性都是以数值来描述的。因此当你在看某一个示例时，一定要试验一下，不要把它们留在那里。对同一个例子试一下其他属性，这里将给出几个思路。

1．透明度

将缓动应用到影片剪辑的alpha属性中。开始设置它为0，然后设置它的目标为1（注意在AS 3

www.TopSage.com

中，alpha是一个从0.0到1.0的值）：

```
ball.alpha = 0;
var targetAlpha:Number = 1;
```

然后可以在enterFrame事件处理器中使用缓动淡入：

```
ball.alpha += (targetAlpha - ball.alpha) * easing;
```

199

或者将0和1反过来使它淡出。

2. 旋转

设置rotation属性和旋转的目标。当然，需要有可以可视旋转的东西，例如箭头：

```
arrow.rotation = 90;
var targetRotation:Number = 270;
```

然后缓动它：

```
arrow.rotation += (targetRotation - arrow.rotation) * easing;
```

3. 颜色

如果你想进行真正的挑战，试一下缓动24位色。需要从红、绿和蓝的初始值和目标值开始，在每一种单独的组成色上执行缓动，然后再把它们组合成一个24位色。例如，你可以从红色缓动到蓝色。从初始和目标颜色开始：

```
red = 255;
green = 0;
blue = 0;
redTarget = 0;
greenTarget = 0;
blueTarget = 255;
```

然后在enterFrame处理器中，在每一种组成色上执行缓动。这里只是红色值：

```
red += (redTarget - red) * easing;
```

然后将3种组成色组合为一个颜色值（在第4章中讲过）：

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

并在ColorTransform（见第4章）中使用它或将它应用在线条色、填充色或其他任何你想应用颜色的地方。

8.2.5 高级缓动

你已经明白简单的缓动是如何工作了，可能会考虑使用更复杂的缓动公式来制作一些特殊的效果。例如，你可能想让某个物体缓慢地开始，加速，然后当接近目标点时再慢下来。或者想让某物体在某个时间段内或某些帧内缓动到指定的位置。

Robert Penner因为对缓动公式进行收集、分类，并在ActionScript中实现它们而变得有名。你可以在www.robertpenner.com网站上找到他的缓动公式。在写这些公式时，还没有AS 3版本，但是它们是很容易转换到AS 3中的。

好，让我们转移到在Flash中可能我最喜欢的主题：弹性。

200

www.TopSage.com

大家网 TopSage.com

8.3 弹性

也许只有我，但是我总是认为弹性是ActionScript动画中最强大、最有用的物理概念。好像你几乎可以使用弹性做任何事情。当然，它只是具有特殊应用的另一项技术。我已经从这项技术中得到了很多的好处，以至于我有些热衷于它。那么，让我们来看一下什么是弹性和如何在Flash中对它编程。

如我在本章的开始提到的，弹性的加速度与它到目标点的距离是成正比的。想一下真实的物理弹性，或更好的事物，一个在橡皮筋末端的小球。当小球挂在那里没有外力时，此时小球的位置就是它的目标点。那也是它想在的位置。现在拉动一下小球让它运动。此时小球的速度是0，但是橡皮筋施加力给小球，将它拉回到目标点。现在把小球拉到尽可能远的位置让它运动。橡皮筋施加一个更大的力给小球。小球迅速地穿过目标点向另一个方向运动。它的速度非常快。但是当小球超过目标点时，橡皮筋开始将它拉回一些——改变了它的速度。小球继续运动，但是它，橡皮筋对它的拉力越大。最后小球的速度达到0，方向为反方向。整个过程不断地重复，最后经过几次来回摆动之后，它慢下来并停在——你可能已经猜到在哪儿——它的目标点。

现在，让我们开始将它转换成ActionScript以便你可以使用它。为了更简单一些，让我们先从一维的开始。

8.3.1 一维弹性

拖动我们的好朋友——红色的小球，让它重新为我们服务。你需要将它放在默让的*x*轴的0位置并让它弹到中央。和使用缓动一样，需要一个变量来保存弹性的比例数值。你可以把它想象成将要加到速度中的距离的比例系数。一个比较高的弹性值将形成非常剧烈的弹性。比较低的值看上去更像是松动的橡皮筋。你需要从0.1开始。下面是到目前为止的代码：

```
private var spring:Number = 0.1;
private var targetX:Number = stage.stageWidth / 2;
private var vx:Number = 0;
```

另外，先不要担心这些代码放在哪里，只需要确保你知道这些变量和语句代表什么意思。

然后移到运动代码并找到与目标点之间的距离：

```
var dx:Number = targetX - ball.x;
```

现在，计算加速度。加速度将与距离成正比。实际上，它是距离与spring数值的乘积：

```
var ax:Number = dx * spring;
```

一旦你得到了加速度的值，你就又回到了你熟悉的内容上了。将加速度加到速度上，将速度加到位置上，对吗？

```
vx += ax;
ball.x += vx;
```

201

在你写任何代码之前，我们先用一些样本值来模拟它。假设x位置是0，vx是0，目标点x位置是100，spring变量值为0.1。下面是它可能的过程：

www.TopSage.com

(1) 用spring乘以距离(100)，你得到的是10。将它加到vx，那么vx就变成了10。将速度加到位置，使x位置变为了10。

(2) 下一个阶段，距离(100–10)是90。加速度是0.1的90倍，或说是9。将它加到vx，得到的是19。x位置变为29。

(3) 再下一个阶段，距离是71，加速度是7.1，将其加到vx，vx变为26.1，x位置变为55.1。

(4) 接下来，距离是44.9，加速度是4.49，vx变为30.59。x位置为85.69。

需要注意的是当物体接近它的目标点时每一帧的加速度会变得越来越小，但是速度会继续增大。它没有前面的几帧中增大得那么快，但是它仍然是移动得越来越快。

再经过几个阶段之后，物体穿过目标点到达x大约为117的位置。距离现在是100–117，也就是–17。将它加到速度中，使物体慢下来一些。

现在你理解了弹性是如何工作了，让我们来写一个真正的文件。与往常一样，确保Ball类是可用的。使用下面的文档类（Spingl.as）：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Spring1 extends Sprite
    {
        private var ball:Ball;
        private var spring:Number = 0.1;
        private var targetX:Number = stage.stageWidth / 2;
        private var vx:Number = 0;

        public function Spring1()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.y = stage.stageHeight / 2;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            var dx:Number = targetX - ball.x;
            var ax:Number = dx * spring;
            vx += ax;
            ball.x += vx;
        }
    }
}
```

202

对它进行测试，你将发现确实有某个类似弹性的东西已经开始工作了。唯一的问题是它好像永远不会停下来。在这之前，当我介绍弹性时，我说过：“它会变慢然后停下来”。原因是，小球

www.TopSage.com

在每一侧摆动过程中保持了相同的速度，因此它保持以相同的速率来回进行弹性运动。你需要用一些东西来降低它的速度以使它慢下来。现在听起来熟悉吗？没错，你需要施加一个摩擦力。很简单——只是设置一个friction变量，对于初学者来说赋给它一个类似0.95的值。与其他类变量一样放置在类的顶端：

```
private var friction:Number = 0.95;
```

然后在enterFrame处理器的某个地方将friction与*vx*相乘，下面是文档类Spring2.as中更正后的onEnterFrame方法：

```
private function onEnterFrame(event:Event):void
{
    var dx:Number = targetX - ball.x;
    var ax:Number = dx * spring;
    vx += ax;
    vx *= friction;
    ball.x += vx;
}
```

此时，尽管是一维的，你已经有了一个完全合格的弹性。确保你多测试一下它。观察使用不同的spring和friction值所产生的不同效果，并了解它们之间是如何相互作用的。试验不同的起点或目标点如何影响整个系统的行为、小球的速率和它变慢并停下来的比例。如果你已经完全掌握了，你可以准备转到二维的弹性上了。

8.3.2　二维弹性

如果你认为二维弹性只是简单地加入第2个目标点速度和加速度，那么我告诉你：你是对的。因此，我们直接给出一个二维的弹性（Spring3.as）：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    public class Spring3 extends Sprite
    {
        private var ball:Ball;
        private var spring:Number = 0.1;
        private var targetX:Number = stage.stageWidth / 2;
        private var targetY:Number = stage.stageHeight / 2;
        private var vx:Number = 0;
        private var vy:Number = 0;
        private var friction:Number = 0.95;

        public function Spring3()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
```


www.TopSage.com

```
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var dx:Number = targetX - ball.x;
            var dy:Number = targetY - ball.y;
            var ax:Number = dx * spring;
            var ay:Number = dy * spring;
            vx += ax;
            vy += ay;
            vx *= friction;
            vy *= friction;
            ball.x += vx;
            ball.y += vy;
        }
    }
}
```

如你所看到的，唯一的不同就是加入了与y轴相关的内容。问题是，它仍像是一维的。是的，小球现在在x和y两个轴上移动，但是它只是沿着直线移动。那是因为它的开始速度为0，并且只有朝向目标点的拉力作用在它上面，因此它是沿直线移动到目标点。

要使它更有趣一些，则初始化vx为比0大一些的值。试一下很大的值例如50。现在，它看上去更自然和流畅了。但是你才刚刚开始，还有许多更酷的内容。 204

8.3.3 移动目标点的弹性

当你听到弹性不需要在每一个帧上有相同的目标点时，你可能不会吃惊。当我在介绍缓动时，我演示了一个非常简单的小球跟随鼠标的例子。将该例变成使小球弹到鼠标是很容易的。不使用targetX和targetY，而使用鼠标的坐标。在弹性中，同缓动一样，与目标点之间的距离总是要在每一帧上重新计算。加速度是基于距离的，并且加速度要加到速度上。

效果是很酷的，我觉得我应当持续地写它，但事实上关于这个主题并没有太多可说的，代码并不是完全不同。在前面的例子中只需要修改下面的几行：

```
var dx:Number = targetX - ball.x;
var dy:Number = targetY - ball.y;
```

改后如下：

```
var dx:Number = mouseX - ball.x;
var dy:Number = mouseY - ball.y;
```

也可以删去声明targetX和targetY变量的代码行，但是如果把它们留在那里它们也不会产生任何影响。更新后的文档类Spring4.as是可用的。

这里还有一个好的地方就是停止和播放。仔细感受一下这些变量是如何工作的，然后对很多变量进行试验。把它弄坏，并查出是哪里出问题了。用它尽情地玩吧！

www.TopSage.com

8.3.4 弹性在哪儿

此时，已经有一个看起来非常真实的在橡皮筋末端的小球。但是它看起来像是一个不可见的橡皮筋。好，使用drawing API的几行代码可以很容易地解决这个问题。

因为有一个不包含其他复杂内容的相当简单的文件，所以你可以放心地直接在主类中应用你的绘图代码，它扩展的是Sprite类。在一个更加复杂的应用中，你可能需要创建一个空的精灵来用作绘制图形的层。

思路很简单。在每一帧上，当小球就位之后，调用clear()方法来擦除之前的线条。然后重置lineStyle并且绘制一条从小球到鼠标的线条。你只需要在enterFrame处理器中使用下面的代码，代码紧跟在小球的位置（稍后你将在完整的代码中看到它）之后。

```
graphics.clear();
graphics.lineStyle(1);
graphics.moveTo(ball.x, ball.y);
graphics.lineTo(mouseX, mouseY);
```

好，这很有意思！你还可以做什么呢？加入一个重力到小球上使它看起来像是真地挂在鼠标
205 末端怎么样？这很容易。对每一帧只需增加一个gravity变量然后将它加到*vy*上。现在你应当知道怎样操作了。下面是将线条绘制与重力相结合的代码（Spring5.as）：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Spring5 extends Sprite
    {
        private var ball:Ball;
        private var spring:Number = 0.1;
        private var vx:Number = 0;
        private var vy:Number = 0;
        private var friction:Number = 0.95;
        private var gravity:Number = 5;

        public function Spring5()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }


        private function onEnterFrame(event:Event):void
        {
            var dx:Number = mouseX - ball.x;
```

www.TopSage.com

TopSage.com

```
            var dy:Number = mouseY - ball.y;
            var ax:Number = dx * spring;
            var ay:Number = dy * spring;
            vx += ax;
            vy += ay;
            vy += gravity;
            vx *= friction;
            vy *= friction;
            ball.x += vx;
            ball.y += vy;
            graphics.clear();
            graphics.lineStyle(1);
            graphics.moveTo(ball.x, ball.y);
            graphics.lineTo(mouseX, mouseY);
        }
    }
}
```

206

当测试这个版本时，将会看到类似于图8-2的显示。

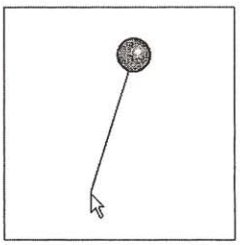


图8-2 来自鼠标的弹性，带有一个可视的弹性

注意，为了使小球真正地向下垂，需要将gravity变量增加到5。如果比5小很多的话，弹力就会克服重力，你将看不到下垂的效果。

现在，还有另外一点，我已经糟蹋了真实世界的物理。当然，不能人为地“增加重力”到物体上！重力是一个常量，它取决于你所在行星的大小和质量。你所能做的是增加物体的质量，以使重力对它的作用更明显。因此，从技术上讲我将保持重力为0.5，然后创建一个mass（质量）属性并且让它为10。然后可以将mass与gravity相乘得到的还是5。或者我可以将gravity变量的名字改为forceThatGravityIsExertingOnThisObjectBasedOnItsMass。但既然你知道了我的意思，我就节省空间把它缩短为gravity。

再次，试验这个例子。试着减小gravity和spring的值。试着改变friction的值。你将会发现可以有几乎无穷多种组合，允许你创建各种系统。

8.3.5 弹性链

随着向右移动，让我们把几个弹性链在一起。在缓动一节中，我简单讨论了鼠标跟随，它是指一个物体缓动到鼠标，另一个物体缓动到那个物体的位置，等等。我没有给出例子是因为它太老了，也不再是什么时髦的效果了。但是，当你将这个概念应用到弹性中时，就不同了。

这里是一个计划：开始时创建3个小球，命名为ball0、ball1和ball2。第1个小球ball0的行为非常像之前示例中的那个小球。那么ball1将弹到ball0，ball2将弹到ball1。所有的小球都有重力，因此它们看上去是挂着的一串小球。代码并不是你之前从来没有见过，只是稍微复杂一点而已。这里是一个文档类名为Chain.as：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
```

www.TopSage.com

```
    public class Chain extends Sprite
    {
        private var ball0:Ball;
        private var ball1:Ball;
        private var ball2:Ball;
        private var spring:Number = 0.1;
        private var friction:Number = 0.8;
207     private var gravity:Number = 5;
        public function Chain()
        {
            init();
        }

        private function init():void
        {
            ball0 = new Ball(20);
            addChild(ball0);
            ball1 = new Ball(20);
            addChild(ball1);
            ball2 = new Ball(20);
            addChild(ball2);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }


        private function onEnterFrame(event:Event):void
        {
            moveBall(ball0, mouseX, mouseY);
            moveBall(ball1, ball0.x, ball0.y);
            moveBall(ball2, ball1.x, ball1.y);

            graphics.clear();
            graphics.lineStyle(1);
            graphics.moveTo(mouseX, mouseY);
            graphics.lineTo(ball0.x, ball0.y);
            graphics.lineTo(ball1.x, ball1.y);
            graphics.lineTo(ball2.x, ball2.y);
        }

        private function moveBall(ball:Ball,
                                  targetX:Number,
                                  targetY:Number):void
        {
            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;
        }
    }
}
```

www.TopSage.com

回头再看看Ball类，你会看到类的每一个实例都有它自己的vx和vy属性，并且它们初始化为0。因此，在init方法中，只需要创建每一个小球并将它们加到显示列表中。 208

然后在onEnterFrame函数中，执行所有的弹性。注意并不是为每一个小球复制相同的代码3次，我们使用了moveBall方法，它包含了所有的运动代码。这需要引用一个小球和一个x和一个y目标点。针对每一个小球调用这个函数，传递鼠标的x和y位置给第1个小球，然后传递第1个小球的位置给第2个小球，第2个小球的位置给第3个小球。

最后，当所有的小球都就位之后，设置一个线条的样式，将绘制光标移到鼠标的位置，然后再依次绘制到每个小球的线条，创建一条把所有小球链接在一起的橡皮筋。注意这里的摩擦力系数使用的是0.8以使它们可以更快地停下来。

可以让这个类更灵活一点，通过创建一个数组来保存对链中每一个小球的引用，然后循环整个数组来移动每一个小球，然后再绘制线条。这只需要少量的改动。首先，需要两个新的变量，一个是数组另一个是你所创建的物体的数量：

```
private var balls:Array;
private var numBalls:Number = 5;
```

init函数通过一个for循环创建每一个物体，并将它们加到数组中：

```
private function init():void
{
    balls = new Array();
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball = new Ball(20);
        addChild(ball);
        balls.push(ball);
    }
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

最后，onEnterFrame方法的变化最大。开始的时候设置线条的样式，移动绘制的位置到鼠标的位置，移动第1个小球，然后绘制一个到它的线条。接下来循环数组中剩下的所有小球，轮流移动并绘制到每一个小球的线条。你可以加入任意多的小球，只需要改变变量numBalls的值即可。

```
private function onEnterFrame(event:Event):void
{
    graphics.clear();
    graphics.lineStyle(1);
    graphics.moveTo(mouseX, mouseY);
    moveBall(balls[0], mouseX, mouseY);
    graphics.lineTo(balls[0].x, balls[0].y);

    for(var i:uint = 1; i < numBalls; i++)
    {
        var ballA:Ball = balls[i-1];
        var ballB:Ball = balls[i];
        moveBall(ballB, ballA.x, ballA.y);
```

209

www.TopSage.com

```
            graphics.lineTo(ballB.x, ballB.y);
        }
    }
```

你可以在图8-3上看到结果并且在ChainArray.as中找到这个类。

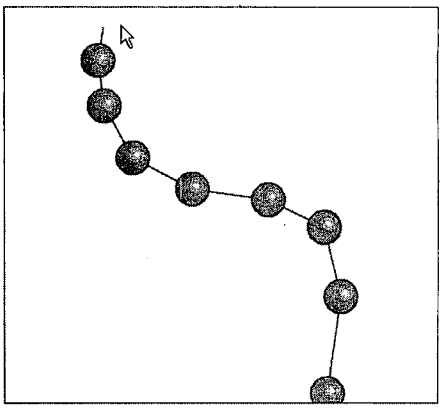


图8-3　弹性链

8.3.6　多目标点弹性

回到第5章介绍的关于速度和加速度的内容上，我讨论过怎样可以使多个力作用在同一个物体上。每一个力就是一个加速度，你只需要逐个地将它们加到速度上。好，那么弹性也无非是加在物体上的一个加速度，所以创建作用在单一物体上的多个弹性动作是很简单的。

这里是针对多目标弹性的设置情况：需要创建3个“控制点”，它们只是Ball类的实例，赋予它们简单的拖放功能。它们也是小球所要弹到的目标点。小球将要尽可能一次弹到3个目标点并在它们之间找到一个均衡点。或者，用另一个种方式说，每一个目标会对小球施加一定数量的加速度，它的运动将是所有那些力的总和。

这个示例非常复杂，使用了几个方法来处理各种动作。我将给出代码（文档类MultiSpring.as）然后一段一段地来讨论它。

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class MultiSpring extends Sprite
    {
        private var ball:Ball;
        private var handles:Array;
        private var spring:Number = 0.1;
        private var friction:Number = 0.8;
        private var numHandles:Number = 3;

        public function MultiSpring()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball(20);
            addChild(ball);

            handles = new Array();
            for(var i:uint = 0; i < numHandles; i++)
```


www.TopSage.com

```
            {
                var handle:Ball = new Ball(10, 0x0000ff);
                handle.x = Math.random() * stage.stageWidth;
                handle.y = Math.random() * stage.stageHeight;
                handle.addEventListener(MouseEvent.MOUSE_DOWN,
                                        onPress);
                addChild(handle);
                handles.push(handle);
            }

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            addEventListener(MouseEvent.MOUSE_UP, onRelease);
        }

        private function onEnterFrame(event:Event):void
        {
            for(var i:uint = 0; i < numHandles; i++)
            {
                var handle:Ball = handles[i] as Ball;
                var dx:Number = handle.x - ball.x;
                var dy:Number = handle.y - ball.y;
                ball.vx += dx * spring;
                ball.vy += dy * spring;
            }

            ball.vx *= friction;
            ball.vy *= friction;
            ball.x += ball.vx;
            ball.y += ball.vy;

            graphics.clear();
            graphics.lineStyle(1);
            for(i = 0; i < numHandles; i++)                    211
            {
                graphics.moveTo(ball.x, ball.y);
                graphics.lineTo(handles[i].x, handles[i].y);
            }
        }

        private function onPress(event:MouseEvent):void
        {
            event.target.startDrag();
        }

        private function onRelease(event:MouseEvent):void
        {
            stopDrag();
        }
    }
}
```

init方法通过for循环创建了一个小球和3个控制点，并且它们的位置是随机的，然后设置它们使它们具有拖放行为。

www.TopSage.com

大家网 TopSage.com

onEnterFrame方法循环每一个控制点，让小球向它们做弹性运动。然后它指定新的速度给小球的位置，然后再循环一次，绘制从小球到各个控制点的线条。

onPress方法比较直观，但是注意在onRelease函数中，你没有办法知道哪一个控制点正在被拖动。幸运的是，在任何显示对象上调用stopDrag可以停止所有的拖动，因此你只需要在主类中调用它就可以了。

注意，可以很容易地改变控制点的数量，只需要改变numHandles变量值即可。

图8-4显示了这段代码的一个示例结果。

到现在为止，我确信你已经开始绕道并且创建一些我不曾涉及的效果（或可能已有了一些想法）。如
212 果是这样，那太好了！这就是我写这本书的目标。

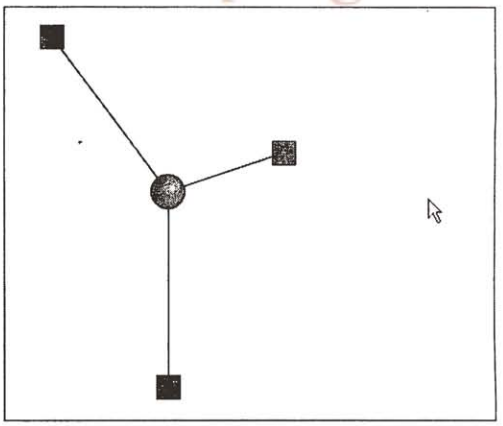


图8-4　多目标点弹性

8.3.7　目标偏移

如果取一个真正的弹簧——一圈真正有弹性的金属，将一端固定在某个物体上，而另一端是小球或其他某个物体，那么哪儿是目标点呢？目标点是弹簧被固定的点吗？不，不一定。小球永远不会到达那一点，因为弹簧本身也在运动中。而且，一旦弹簧收缩到它的正常长度，它将不再施加给小球任何力。因此，目标点实际上是弹簧处于没有被拉长的状态时的末端端点的位置。但是当弹簧绕固定点旋转时这一点会发生变化。

要找到真正的目标点，首先需要找到物体和固定点之间的角，然后沿着那个角从固定点向外移动一段距离——弹簧的长度。换句话说，如果弹簧的长度是50，小球和固定点之间的角是45度，将需要以45度角的方向从固定点移出50个像素，那么那一点将是小球弹到的目标点。图8-5演示了它是如何工作的。

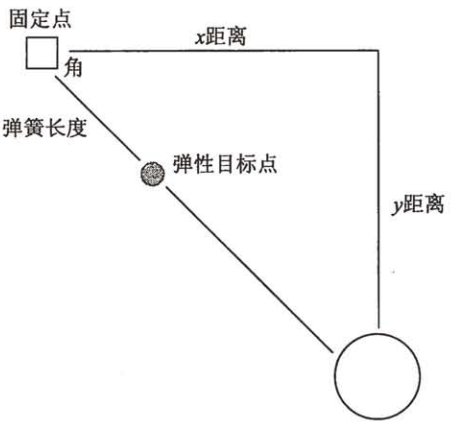


图8-5　弹性偏移

www.TopSage.com

在这种情况下查找目标点的代码大致如下：

```
var dx:Number = ball.x - fixedX;
var dy:Number = ball.y - fixedY;
var angle:Number = Math.atan2(dy, dx);
var targetX:Number = fixedX + Math.cos(angle) * springLength;
var targetY:Number = fixedY + Math.sin(angle) * springLength;
```

213

因此，结果是物体将向固定点弹动，但是会偏离固定点一些距离。另外，注意虽然我这里称的是“固定点”，它只是意味着弹簧被固定的点。它并不代表着那一点不能移动。可能最好是能看到它在运动。你将又要使用鼠标位置了，但这一次，它将是弹簧的固定点。弹簧的长度是100像素，下面是文档类（OffsetSpring.as）：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class OffsetSpring extends Sprite
    {
        private var ball:Ball;
        private var spring:Number = 0.1;
        private var vx:Number = 0;
        private var vy:Number = 0;
        private var friction:Number = 0.95;
        private var springLength:Number = 100;

        public function OffsetSpring()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball(20);
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }


        private function onEnterFrame(event:Event):void
        {
            var dx:Number = ball.x - mouseX;
            var dy:Number = ball.y - mouseY;
            var angle:Number = Math.atan2(dy, dx);
            var targetX:Number = mouseX + Math.cos(angle) *
                                 springLength;
            var targetY:Number = mouseY + Math.sin(angle) *
                                 springLength;
            vx += (targetX - ball.x) * spring;
            vy += (targetY - ball.y) * spring;
            vx *= friction;
            vy *= friction;
            ball.x += vx;
```

www.TopSage.com

```
            ball.y += vy;
214         graphics.clear();
            graphics.lineStyle(1);
            graphics.moveTo(ball.x, ball.y);
            graphics.lineTo(mouseX, mouseY);
        }
    }
}
```

尽管你可以看到效果了，但是你可能没有那么容易发现它可以在哪些地方应用。好，下一节将给出一个特殊的示例。

8.3.8 使用弹性贴加多个物体

记得我曾意识到可以实现它的准确时间。大致是这样，“好的，我知道了如何使一个物体弹到一个点上，并且知道那一点并不一定是固定的。我甚至可以让一个物体弹到另外一个物体。好，如果让另外一个物体弹回到第1个物体怎么样？那么这两个物体应当是通过一个弹簧连接在一起。移动其中的任何一个，另一个开始弹到前一个物体。”

我的初始印象是它可能会造成Flash的死循环，或至少会出现各种警告信息。但是我毅然地继续去尝试它。让我惊喜的是，它工作得非常完美!

我已经详细地描述了思路，再进行一下细化：物体A将物体B作为它的目标。它向物体B弹动。反过来物体B将物体A作为它的目标。实际上，这一点就是偏移（offset）发挥重要作用的位置。如果每个物体将另外的物体作为直接的目标，它们将互相挤压对方并争相占领相同的某一点。通过应用偏移，可以让它们保持一段距离，如图8-6所示。

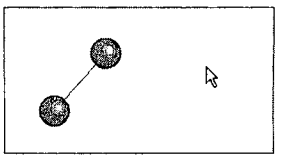


图8-6　使用弹簧连接的两个物体

下一个例子，将需要两个Ball类的实例，我称它们为ball0和ball1。ball0结合偏移向ball1弹动。同时ball1结合偏移向ball0弹动。不需要书写全部的偏移、弹力和运动代码两次，我将它放在一个称为springTo的函数里。因此，你可以调用springTo(ball0,ball1)来让ball0弹到ball1，调用springTo(ball1,ball0)来让ball1弹到ball0。我又添加了两个变量ball0Dragging和ball1Dragging，以使当拖动每个小球时停止弹动。下面是文档类(DoubleSpring.as)：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class DoubleSpring extends Sprite
    {
        private var ball0:Ball;
        private var ball1:Ball;
        private var ball0Dragging:Boolean = false;
215     private var ball1Dragging:Boolean = false;
```

www.TopSage.com

```
private var spring:Number = 0.1;
private var friction:Number = 0.95;
private var springLength:Number = 100;

public function DoubleSpring()
{
    init();
}

private function init():void
{
    ball0 = new Ball(20);
    ball0.x = Math.random() * stage.stageWidth;
    ball0.y = Math.random() * stage.stageHeight;
    ball0.addEventListener(MouseEvent.MOUSE_DOWN, onPress);
    addChild(ball0);

    ball1 = new Ball(20);
    ball1.x = Math.random() * stage.stageWidth;
    ball1.y = Math.random() * stage.stageHeight;
    ball1.addEventListener(MouseEvent.MOUSE_DOWN, onPress);
    addChild(ball1);

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.addEventListener(MouseEvent.MOUSE_UP, onRelease);
}


private function onEnterFrame(event:Event):void
{
    if(!ball0Dragging)
    {
        springTo(ball0, ball1);
    }
    if(!ball1Dragging)
    {
        springTo(ball1, ball0);
    }
    graphics.clear();
    graphics.lineStyle(1);
    graphics.moveTo(ball0.x, ball0.y);
    graphics.lineTo(ball1.x, ball1.y);
}

private function springTo(ballA:Ball, ballB:Ball):void
{
    var dx:Number = ballB.x - ballA.x;
    var dy:Number = ballB.y - ballA.y;
    var angle:Number = Math.atan2(dy, dx);
    var targetX:Number = ballB.x - Math.cos(angle) *
                         springLength;
    var targetY:Number = ballB.y - Math.sin(angle) *
                         springLength;
    ballA.vx += (targetX - ballA.x) * spring;
```


www.TopSage.com

```
            ballA.vy += (targetY - ballA.y) * spring;
            ballA.vx *= friction;
            ballA.vy *= friction;
            ballA.x += ballA.vx;
            ballA.y += ballA.vy;
        }

        private function onPress(event:MouseEvent):void
        {
            event.target.startDrag();
            if(event.target == ball0)
            {
                ball0Dragging = true;
            }
            if(event.target == ball1)
            {
                ball1Dragging = true;
            }
        }

        private function onRelease(event:MouseEvent):void
        {
            ball0.stopDrag();
            ball1.stopDrag();
            ball0Dragging = false;
            ball1Dragging = false;
        }
    }
}
```

对于这个文件，小球被放置在舞台上并且设置为可以进行拖放操作。enterFrame处理器函数简单地对每一个球调用springTo函数。在程序中需要注意的是，每次执行下面这两行之前都要先检测以确保小球没有被拖动：

```
springTo(ball0, ball1);
springTo(ball1, ball0);
```

所有动作发生在springTo函数中。函数的中每一句你应当已经比较熟悉了。首先，它求出到另外一个小球的距离和角，然后在此基础上计算一个目标点。然后在那个目标点上执行基本的弹性机制。当使用调换后的参数再次调用函数时，小球互换了角色，原来作为目标的小球向另一个
217 小球弹动。这可能不是效率最高的代码，但是它很清楚地演示了可能发生的一切。

你将会看到没有哪个小球被贴加在固定点上或是鼠标上，它们两者都是自由浮动的。它们之间的唯一的约束是彼此保持一定的距离。这种方式的最大好处是你可以很容易地加入其他的物体。例如，如果你创建了第3个小球（ball2），然后像其他小球一样设置它（需要有ball2Dragging变量），你可以像这样将其加入代码中：

```
if(!ball0Dragging)
{
    springTo(ball0, ball1);
    springTo(ball0, ball2);
}
```

www.TopSage.com

TopSage.com

```
if(!ball1Dragging)
{
    springTo(ball1, ball0);
    springTo(ball1, ball2);
}
if(!ball2Dragging)
{
    springTo(ball2, ball0);
    springTo(ball2, ball1);
}
```

这将会创建一个三角形的结构，如图8-7所示，这对我来说是相当地酷。我确信一旦你掌握这种方法，将会快速转移到方形图形上，甚至各种复杂的弹性结构。

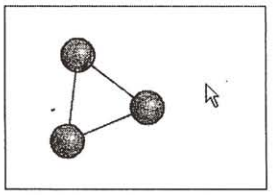


图8-7 用弹簧连接的3个物体

8.4 本章重点公式

现在来回顾一下本章所用到的重点公式。

简单的缓动，较长的形式：

```
var dx:Number = targetX - sprite.x;
var dy:Number = targetY - sprite.y;
vx = dx * easing;
vy = dy * easing;
sprite.x += vx;
sprite.y += vy;
```

218

简单的缓动，简化的形式：

```
vx = (targetX - sprite.x) * easing;
vy = (targetY - sprite.y) * easing;
sprite.x += vx;
sprite.y += vy;
```

简单的缓动，极简的形式：

```
sprite.x += (targetX - sprite.x) * easing;
sprite.y += (targetY - sprite.y) * easing;
```

简单的弹性，较长的形式：

```
var ax:Number = (targetX - sprite.x) * spring;
var ay:Number = (targetY - sprite.y) * spring;
vx += ax;
vy += ay;
vx *= friction;
vy *= friction;
sprite.x += vx;
sprite.y += vy;
```

简单的弹性，简化的形式：

```
vx += (targetX - sprite.x) * spring;
vy += (targetY - sprite.y) * spring;
vx *= friction;
vy *= friction;
```

www.TopSage.com

```
sprite.x += vx;
sprite.y += vy;
```

简单的弹性，极简的形式：

```
vx += (targetX - sprite.x) * spring;
vy += (targetY - sprite.y) * spring;
sprite.x += (vx *= friction);
sprite.y += (vy *= friction);
```

偏移弹性：

```
var dx:Number = sprite.x - fixedX;
var dy:Number = sprite.y - fixedY;
var angle:Number = Math.atan2(dy, dx);
var targetX:Number = fixedX + Math.cos(angle) * springLength;
var targetY:Number = fixedX + Math.sin(angle) * springLength;
// 像上面一样弹到目标点targetX,targetY。
```

219

8.5　小结

这一章涉及了两个基本的比例运动技术：缓动和弹性。你已经知道了缓动是比例运动，而弹性是比例速度，你应当已经很好地理解了如何应用这两项技术。

我希望你能理解为什么我偏爱弹性，可以亲自使用它们创建一些真正有趣的效果。

现在你已经学习了各种移动物体的方法，让我继续进入下一章，那里你将会看到当它们互相碰撞时如何处理！

220

www.TopSage.com

第9章

碰撞检测

本章内容

- 碰撞检测方法
- 内置的hitTest方法
- 基于距离的碰撞检测
- 多物体碰撞检测策略
- 本章重点公式

当阅读到本章时，你已经知道了如何移动物体以及如何使物体与它们所占有的空间产生交互。现在将深入研究这些物体之间的交互。通常，这将涉及判断两个物体何时产生接触。该问题被称为*碰撞检测*（collision detection）或*碰撞测试*（hit testing）。

在这一章中，我将尽量讲解了解碰撞检测所需要的每一项内容。这包括两个精灵的碰撞测试、一个精灵与一个点的碰撞测试、基于距离的碰撞检测和多个物体的碰撞测试策略。首先，让我们来看一下对于碰撞检测有哪些方法可用。

9.1 碰撞检测方法

总地来说，碰撞检测或碰撞测试是一个非常简单的概念。你需要知道两个物体是否同时占据着同一个空间的任何部分。当然，你可能有很多物体，而且可能想知道它们之间任何两者是否发生碰撞。可是当把问题分解后，你需要测试每一个物体与其他每一个物体的碰撞。因此，它总是归结为总是要测试一个物体与另一个物体的碰撞。

如本节的标题所示，你有两种方法来进行碰撞检测：

- 你可以基于每一个对象（精灵或影片剪辑）的实际像素来测试，也就是说，一个精灵的图形是否覆盖了另一个精灵的图形？对于这个测试，你是否考虑了精灵图形的实际可见像素？或者你基于的是否是精灵的方形边界？因此，有两个具有额外选项的内置的碰撞测试方法可以自定义你所要进行的测试方法。
- 你可以基于距离进行碰撞测试。首先得到两个物体之间的距离，然后你问："它们是否已经足够接近以至于要发生碰撞？"你需要计算距离并判定何时物体已经足够接近了。

www.TopSage.com

每一种方法都有它的用处。在本章中我将详细讲解这两种方法，你将有一个很好的机会看到它们是如何使用的。

在本章中有一部分内容没有详细讲解，就是当你确实得到一个碰撞时你要做什么。当两个物体相互碰撞时它们是如何反应的？我将在第11章中讨论动量守恒时毫无遗漏地讲解这些内容。

9.2 hitTestObject 和 hitTestPoint

对于Flash的很多版本，影片剪辑有一个内置的方法称为hitTest，你可以以几种不同的方式来使用它。现在，它已经被更合理地分成了两种独立的方法。hitTestObject方法用于测试一个显示对象（精灵、影片剪辑等等）是否碰撞到另外一个，hitTestPoint方法用来检测一个指定的
224 点是否碰撞到一个显示对象。

9.2.1 碰撞测试两个精灵

使用hitTestObject来检测两个精灵是否彼此碰撞可能是最简单的碰撞检测方法。它也是最容易编程和执行最快的方法。你调用作为一个精灵方法的函数并传入对另一个精灵的引用。注意尽管我现在谈论的是精灵，但这两种碰撞测试方法都是DisplayObject类的成员，而且可以应用于扩展了DisplayObject的任何类，比如MovieClip、Bitmap、Video、TextField等。它看上去像这样：

```
sprite1.hitTestObject(sprite2)
```

它通常是if条件语句中的一部分，像这样：

```
if(sprite1.hitTestObject(sprite2))
{
   // 碰撞反应。
}
```

如果发生了碰撞此方法将返回true，那么if块内的代码将执行。

然而，对于所有的事情而言，都有一个折中。碰撞检测方法越简单，它们就越不准确。要变得更加准确，它们就会变得更复杂并且要花更多的时间。因此，虽然这是一个最简单的方法，但它也是一个准确性最差的方法。

现在，我们所说的碰撞测试到底是指什么呢？是不是指某个东西是否有碰撞，对吗？我希望它会是那么简单。它归结为这样的一个问题：基于两个精灵的位置，你怎样断定它们是否发生了碰撞？

这里是断定是否有碰撞发生的最简单的方法：你准备第1个物体，然后在它的周围绘制一个矩形。矩形的上边缘与物体图形的最上端的可视像素对齐，底边与最下端的可视像素对齐，左和右边与它们的最远的可视像素对齐。然后你再使用同样的方法处理另一个准备用作碰撞测试的物体。然后你检测这两个矩形是否有相交。如果有，就说明有碰撞发生。

这个物体周围的矩形被称为是边界盒（bounding box）。每一个显示对象都有一个边界盒。如

www.TopSage.com

果你在Flash IDE的舞台上单击一个元件，你将会看到一个蓝色的线框，如图9-1所示。

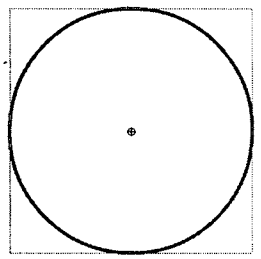


图9-1 边界盒

当然，在Flash 播放器中，没有任何东西真正地绘制矩形并检测它们。它完全是通过数学运算来完成的，基于精灵的位置和大小。

现在，为什么这种方法不准确呢？一种想法是如果边界盒相交，那么物体必定会发生接触。好，看一下图9-2。你能说出哪几组图形彼此接触了吗？

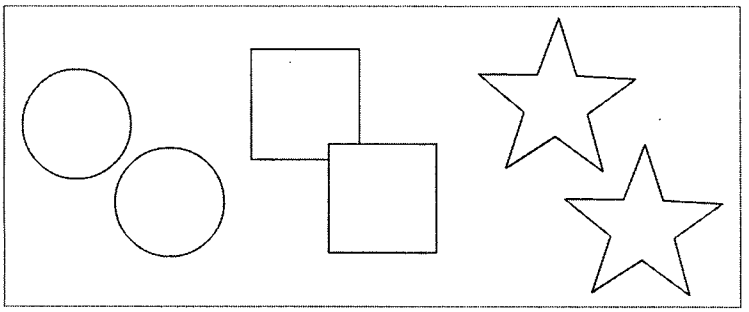


图9-2 哪几组图形相互接触了？

很明显，只有方形是真正的碰撞，对吗？好，让我们绘制上边界盒然后在Flash中看它的样子。图9-3显示了结果。

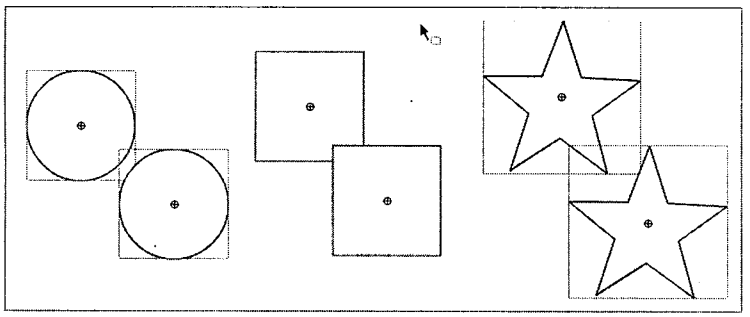


图9-3 不是你所预料的？

现在你了解了什么？就Flash而言每一组图形都发生了碰撞。如果你不相信我，可以测试下面的文档类ObjectHitTest.as。它使用的是我们之前创建的Ball类，所以要确保它在你的类路径下。

225
~
226

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class ObjectHitTest extends Sprite
```

www.TopSage.com

大家网 TopSage.com

```
    {
        private var ball1:Ball;
        private var ball2:Ball;

        public function ObjectHitTest()
        {
            init();
        }

        private function init():void
        {
            ball1 = new Ball();
            addChild(ball1);
            ball1.x = stage.stageWidth / 2;
            ball1.y = stage.stageHeight / 2;

            ball2 = new Ball();
            addChild(ball2);
            ball2.startDrag(true);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            if(ball1.hitTestObject(ball2))
            {
                trace("hit");
            }
        }
    }
}
```

这个类只创建了Ball类的两个实例并设置其中一个可以拖动。然后在每一帧上使用hitTestObject方法来检测两个物体是否碰撞。注意如果你从上、下、左或右边接近静止的小球，它是准确的，但是如果你从斜角方向接近总是会得到一个错误的位置。如果你愿意，可以试着在精灵中绘制一些其他的图形来替换Ball。你会发现矩形会工作得很好，但是使用的形状越不规则，
227 你会发现越多的不准确性。因此，除了矩形之外，你必须要小心地使用此种方法。

让我们试一个使用矩形的简单hitTestObject的例子。这个例子中使用了一个新的Box类，它非常类似于Ball类，我确信你完全可以理解它。代码如下：

```
package {
    import flash.display.Sprite;

    public class Box extends Sprite {
        private var w:Number;
        private var h:Number;
        private var color:uint;
        public var vx:Number = 0;
        public var vy:Number = 0;

        public function Box(width:Number=50,
```

www.TopSage.com

```
                        height:Number=50,
                        color:uint=0xff0000) {
            w = width;
            h = height;
            this.color = color;
            init();
        }
        public function init():void {
            graphics.beginFill(color);
            graphics.drawRect(-w / 2, -h / 2, w, h);
            graphics.endFill();
        }
    }
}
```

这个示例的思路是在屏幕的顶端创建一个盒子然后让它下落到底端。当盒子放好后，它向下掉落直到碰到舞台底部或与其他盒子碰撞。如果它碰到其他盒子，它将停在所碰撞的盒子之上。这里是代码（文档类Boxes.as）：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Boxes extends Sprite
    {
        private var box:Box;
        private var boxes:Array;
        private var gravity:Number = 0.2;

        public function Boxes()
        {
            init();
        }
        private function init():void
        {
            boxes = new Array();
            createBox();
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            box.vy += gravity;
            box.y += box.vy
            if(box.y + box.height / 2 > stage.stageHeight)
            {
                box.y = stage.stageHeight - box.height / 2;
                createBox();
            }
            for(var i:uint = 0; i < boxes.length; i++)
            {
                if(box != boxes[i] && box.hitTestObject(boxes[i]))
                {
```

228

www.TopSage.com

```
                box.y = boxes[i].y - boxes[i].height / 2 -
                        box.height / 2;
                createBox();
            }
        }
    }

    private function createBox():void
    {
        box = new Box(Math.random() * 40 + 10,
                      Math.random() * 40 + 10);
        box.x = Math.random() * stage.stageWidth;
        addChild(box);
        boxes.push(box);
    }
  }
}
```

在onEnterFrame方法中，代码首先检测盒子是否到达了舞台的底部。如果到了，则让它停下并且创建一个新的盒子。然后运行一个for循环，以检测当前的盒子是否与其他所有的盒子相碰撞。首先要确保不是检测当前盒子与自身相碰撞，然后，整个程序的中心就是hitTestObject代码行了，它用来确定当前盒子是否与任何其他盒子相碰撞。如果有碰撞，代码重新定位移动的盒子到它刚碰撞的盒子上方并且创建一个新的盒子。如果你真地想看到这种方法的不准确性，不使用Box类而使用Ball类，或其他某个不规则图形，你会看到物体停在半空，因为它已经"碰撞"到其他物体了。

229

9.2.2　碰撞测试一个精灵和一个点

hitTestPoint方法的工作方式有很大的不同，甚至它自身有两种选择。说实话，这种方法本身对于两个精灵之间的碰撞测试通常没有太大用处。这种方法有两个数值参数，然后使用它们来定义一个点。然后它根据那一点是否碰到正在讨论中的精灵而返回true或false。在大多数情形下，它可能像这样使用（100和100是指一个点的*x*和*y*位置）：

```
sprite.hitTestPoint(100, 100)
```

还有，你需要在if条件语句中使用它来指明只有当碰撞测试为true时，才执行此代码。

但是再一次，我们回到这样的问题上：一个碰撞的本质是什么？并且，再一次，我们看到了老朋友边界盒。Flash只是检测所提供的点是否在精灵的边界盒之中。

让我们使用下面的文档类PointHitTest.as做一个快速测试：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class PointHitTest extends Sprite
    {
        private var ball:Ball;
```

www.TopSage.com

```
        public function PointHitTest()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.x = stage.stageWidth / 2;
            ball.y = stage.stageHeight / 2;

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            if(ball.hitTestPoint(mouseX, mouseY))
            {
                trace("hit");
            }
        }
    }
}
```

230

这里使用的是鼠标的*x*,*y*坐标作为与小球碰撞测试的点。当你移动鼠标接近小球时，你会发现在你真正接碰到图形之前它很可能已经输出了碰撞到的信息，尤其是你从一个角上接近时。再试一下方块形状，它将是很准确的。

那么又一次，这个方法看上去只对矩形物体有用。但这里还有另外一个选项称为shapeFlag。

9.2.3 使用 shapeFlag 的碰撞测试

shapeFlag是hitTestPoint的第3个并且是可选的参数。它是一个布尔值，因此它是true或false。设置shapeFlag为true意味着你的碰撞测试将检查是否碰到精灵的可视图形，而不是边界盒。注意shapeFlag只能用于测试一个精灵与一个点的碰撞。如果你测试的是两个精灵，你不能使用shapeFlag作为一个选项。

让我们进一步测试它。它只是简单地加入一个值。只需要在前面的例子中加入它。改变你的hitTestPoint代码行如下：

```
if(ball.hitTestPoint(mouseX, mouseY, true))
```

（如果你想显式表明你不想使用shapeFlag，你可以设置它为false，它与你不写这个参数是相同的效果。）用一个小球测试这个例子，或绘制其他的某个不规则图形。你将会看到现在你必须使用鼠标真正碰到图形自身才会触发碰撞。

因此，这里你有了一个准确无误的碰撞测试。虽然准确，但可能不能应用于所有的碰撞检测。问题是它只能测试一个点，这使得要检测一个精灵的任意图形部分是否碰到另一个精灵的任意图形部分变得很难。你可能第一时间会想到做如下事情：

www.TopSage.com

```
sprite1.hitTestPoint(sprite2.x, sprite2.y, true)
```

但是在这种情况下，你所检测的是sprite2的注册点是否位于sprite1里面。这是个非常大的限制。Sprite2的某一部分接触到了sprite1。实际上，一定与sprite1重叠了。但是只要sprite2的注册点还在sprite1的外面，Flash会认为它并没有发生碰撞。在实际应用中，针对点的碰撞测试的最好的应用是用于鼠标交互或非常小的精灵上，注册点与外围边界之间可以只有一两个像素远。

人们已经尝试使用这个碰撞检测方法的方式是沿着物体的周边来测试几个点。例如，假设你有一个星形的精灵。你可以计算出星形的5个点在哪，然后用另外一个精灵与星形的5个点做碰撞测试。但是如果你有两个星形，你需要测试每一个星形与另一个星形的5个点之间的碰撞。对于一个星形，这种方法可能会工作得很好。一个更复杂的图形很显然需要更多的点。你会看到这将是非常复杂的并且会使用大量的CPU资源。只是两个星形，你的测试数量就高达10倍。那么这种
231 准确代价也太大了。

9.2.4 hitTest 总结

那么，对于两个不规则形状的精灵你要如何进行碰撞测试，以致如果其中的任何一部分碰到另外一个，你得到碰撞了吗？很不幸，答案是hitTest方法在精灵中不能得到支持。

概括出来，你的基本选择如下：

- 对于粗糙的矩形影片剪辑，使用hitTestObject (displayObject)。
- 对于非常小的精灵，你可以使用hitTestPoint(x,y,true)（注意是shapeFlag设置为true)。
- 对于形状非常不规则的精灵，你要么忍受不准确，要么使用自定义的程序来解决，有可能使用的是hitTestPoint(x,y,true)。

当然，很多内容我们还没有说到，还有内置方法之外的解决方案。如果你具有的是圆形或粗糙的圆形物体，那么基于距离的碰撞检测很可能是最好的办法。实际上，你会惊奇地发现有许多图形属于“粗糙矩形”或“粗糙圆形”类。

9.3 基于距离的碰撞检测

在这一节中，你将抛弃内置的hitTest系列方法，而使用碰撞检测。它包括使用两个物体之间的距离来确定两者是否发生碰撞。

把它放到现实的世界中，如果你的汽车中心距离我的汽车中心是100英尺(1英尺=0.304 8米)，你知道这两辆车足够远是不可能会相互接触的。然而，如果我们的车都是6英尺宽12英尺长，并且我的车的中心距离你的车的中心是5英尺，你可以相当肯定，车里有一些扭曲的金属，而且有一些保险文件要填写。

换句话说，没有办法让两者的中心挨到一起而不让车的某些部分相碰。这就是基于距离的碰撞测试背后的整个概念。你判断出两个物体分离所需要的最小距离，计算出当前的距离，然后对比两者。如果当前的距离小于最小距离，那么你就知道它们正在碰撞。

www.TopSage.com

大学网 TopSage.com

很自然，这里有一个疑问。简单的hitTestObject方法对于矩形可以工作得很好，但对于不是矩形的其他形状就会出现问题，而当前这种方法偏爱完美的圆形。事实上，它对完美的圆形会工作得非常好。因此，如果这就是你要做的，那么就使用这种方法。

当使用的并不是一个圆形时，你将会看到一些不准确的情况。但并不是你之前使用hitTest和边界盒时出现的将没有碰撞的物体报告为碰撞的问题。你可能会遇到相反的问题：看上去已经接触的物体并没有触发碰撞，因为它们的中心还没有足够接近。

9.3.1 简单的基于距离的碰撞检测

让我们从理想的情形开始：两个正圆。我们可以再次使用Ball类。（可能现在你会明白为什么在面向对象编程中经常会使用“可复用的”这个词。）当做这种类型的碰撞检测时，注册点应当位于圆的正中心。Ball类当前可以适应这个需求。像以前一样，你需要创建两个小球，设置其 232
中一个为可拖动。同样需要在一个enterFrame处理器中执行碰撞检测。因此，目前看上去，它与本章中的第1个例子几乎是一样的。只是并没有使用if(ball1.hitTestObject(ball2)来检测碰撞，你将要在if语句中使用距离。你应该已经知道如何计算两个物体之间的距离了。回忆一下我们在第3章讲到的非常好用的毕达哥拉斯定理。因此，你开始会这样写：

```
var dx:Number = sprite2.x - sprite1.x;
var dy:Number = sprite2.y - sprite1.y;
var dist:Number = Math.sqrt(dx * dx + dy * dy);
```

好，现在你有了距离，但是如何判断这个距离已经足够小可以认为碰撞发生了呢？好，看一下图9-4。

这里，你看到两个圆形精灵处于刚刚接触的位置。假设每一个精灵是60像素宽。则它的半径为30。因此，在它们刚刚接触时，它们相距正好是60像素。哈！有答案了。对于两个大小一样的圆形精灵，如果两者的距离小于它们的直径，那么它们正在发生碰撞。此例（Distance.as）代码和ObjectHitTest.as一样，保留了onEnterFrame方法，其代码如下：

```
private function onEnterFrame(event:Event):void
{
    var dx:Number = ball2.x - ball1.x;
    var dy:Number = ball2.y - ball1.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    // 默认的球的直径是80（半径是40）。
    if(dist < 80)
    {
        trace("hit");
    }
}
```

当你运行这个文件时，你会注意到它与你从哪个角度接近目标小球无关。直到图形真正重叠时才会触发碰撞。

现在，我已经涉及了一个事实：在代码中使用硬编码的数字通常不是一个好主意。每次需要使用不同大小的物体时，都需要修改代码。而且当两个精灵的大小不同时你要怎样处理呢？你需

www.TopSage.com

要将这个概念抽象成某种公式以使它可以适应各种情况。

看一下图9-5。它显示的是两个不同大小的精灵，而且又是刚好接触。左边那个直径是60像素，右边那个直径是40像素。你可以通过编程检测它们的width属性来获得这些值。因此，一个半径为30，另一个半径为20。那么，在它们刚刚接触时两者的距离刚好为50。当然，在Ball类中我们已经准备好了半径这个属性，因此我们可以直接使用它。

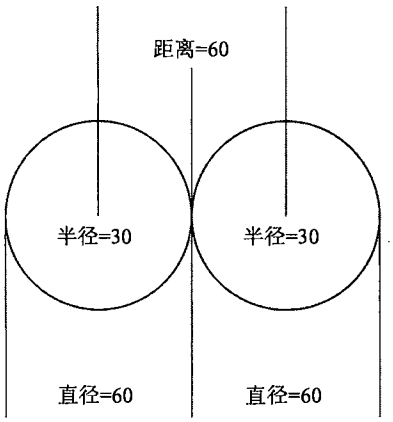


图9-4　碰撞的距离

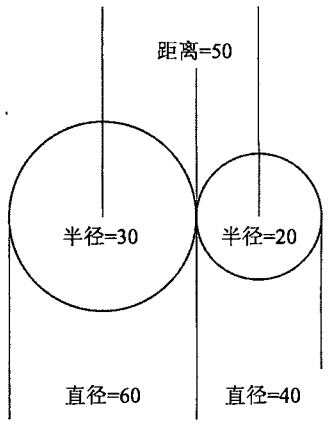


图9-5　两个不同大小的物体之间的碰撞距离

一种模式渐渐地显露出来了。这个神奇的距离是一个小球的半径加上另一个小球的半径。现在你可以从代码中移除所有硬编码的数值然后像这样编写（你可以在文档类Distance2.as中找到）：

233 ~ 234

```
private function onEnterFrame(event:Event):void
{
    var dx:Number = ball2.x - ball1.x;
    var dy:Number = ball2.y - ball1.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    if(dist < ball1.radius + ball2.radius)
    {
        trace("hit");
    }
}
```

继续改变其中一个或两个小球的大小（记住，你可以传递半径作为第1个参数）来观察代码工作，即使是一个圆很大，而另一个圆很小。实际上，在之前的整个类中，我像这样声明了两个物体：

```
ball1 = new Ball(Math.random() * 100);
```

这样，每次文件运行时它们的大小总是不同的，而碰撞检测总是会工作得很好。

9.3.2　基于碰撞的弹性

现在的问题是对于我给出的基于距离的碰撞测试的好的工作示例，它的完整程序涉及了大

www.TopSage.com

量与我还没有讲到的内容相关的问题，如当两个物体碰撞时如何反应和如何有效地处理多个物体之间的交互。但是我确实成功创建了某个表明了碰撞测试而又不涉及太多你未学过的内容的示例。

这里是思路：将一个称为centerBall（中心球）的大球，放在舞台的中央。然后加入一系列的小球，赋给它们随机的大小和速度。它们使用基本的运动代码移动并从墙上弹起。在每一帧上，在每一个移动的小球和中心球之间进行基于距离的碰撞检测。如果你得到了一个碰撞，则根据两个球之间的角度和避免相碰的最小距离计算出偏移弹性目标点。这听上去好像不是很清晰。它实际上是在说如果一个移动的小球碰到了中心球，你要让它再弹回来。你需要中心球外面设置一个目标点的来完成此操作。移动的小球弹到目标点。然后，一旦它到达目标点，它不再进行碰撞，所以弹性动作结束，并且它只使用正常的运动代码来移动。

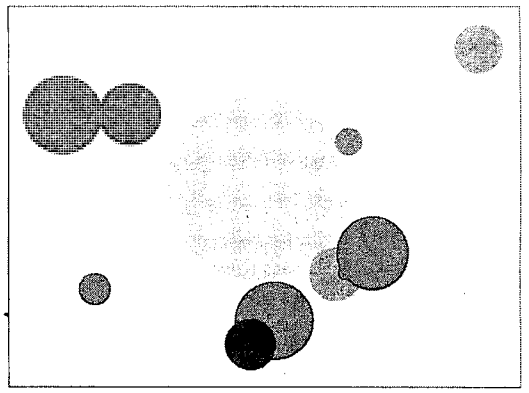


图9-6 基于碰撞的弹性

结果非常类似于许多气泡跳跃地离开一个很大的气泡，如图9-6所示。小的气泡进入大的气泡一小部分，这取决于它们移动的快慢，但会立即弹回来。（哈，你看我对弹性的痴迷又回来了。）

这里是代码（也可以在Bubbles.as中找到它）：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Bubbles extends Sprite
    {
        private var balls:Array;
        private var numBalls:Number = 10;
        private var centerBall:Ball;
        private var bounce:Number = -1;
        private var spring:Number = 0.2;

        public function Bubbles()
        {
            init();
        }

        private function init():void
        {
            balls = new Array();
            centerBall = new Ball(100, 0xcccccc);
            addChild(centerBall);
            centerBall.x = stage.stageWidth / 2;
            centerBall.y = stage.stageHeight / 2;
```

www.TopSage.com

```
        for(var i:uint = 0; i < numBalls; i++)
        {
            var ball:Ball = new Ball(Math.random() *
                            40 + 5,
                            Math.random() * 0xffffff);
            ball.x = Math.random() * stage.stageWidth;
            ball.y = Math.random() * stage.stageHeight;
            ball.vx = Math.random() * 6 - 3;
            ball.vy = Math.random() * 6 - 3;
            addChild(ball);
            balls.push(ball);
        }

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        for(var i:uint = 0; i < numBalls; i++)
        {
            var ball:Ball = balls[i];
            move(ball);
            var dx:Number = ball.x - centerBall.x;
            var dy:Number = ball.y - centerBall.y;
            var dist:Number = Math.sqrt(dx * dx + dy * dy);
            var minDist:Number = ball.radius + centerBall.radius;
            if(dist < minDist)
            {
                var angle:Number = Math.atan2(dy, dx);
                var tx:Number = centerBall.x +
                                Math.cos(angle) * minDist;
                var ty:Number = centerBall.y +
                                Math.sin(angle) * minDist;
                ball.vx += (tx - ball.x) * spring;
                ball.vy += (ty - ball.y) * spring;
            }
        }
    }

    private function move(ball:Ball):void
    {
        ball.x += ball.vx;
        ball.y += ball.vy;
        if(ball.x + ball.radius > stage.stageWidth)
        {
            ball.x = stage.stageWidth - ball.radius;
            ball.vx *= bounce;
        }
237     else if(ball.x - ball.radius < 0)
        {
            ball.x = ball.radius;
            ball.vx *= bounce;
        }
        if(ball.y + ball.radius > stage.stageHeight)
```

www.TopSage.com

```
            {
                ball.y = stage.stageHeight - ball.radius;
                ball.vy *= bounce;
            }
            else if(ball.y - ball.radius < 0)
            {
                ball.y = ball.radius;
                ball.vy *= bounce;
            }
        }
    }
}
```

好，这是全部的代码，但是这里的大多数技术你已经在之前的章节见过。让我们快速地过一遍代码。

从init函数开始，创建了centerBall并且使用循环创建较小的移动小球。赋给它们随机的大小、位置、颜色和速度。

enterFrame处理器只是循环并获取对每一个小球的引用。为了将功能分开，我已经将运动代码移到称为move的函数中。它将通过对每一个小球的引用应用对它的所有基本的运动代码。这个函数对你来说应当已经很熟悉了。它只是基本的回弹速度代码。然后你需要找到小球与centerBall之间的距离，计算出用于确定碰撞的最小距离。如果有碰撞，你找到两者之间的角度，然后用它加上最小的距离来计算出目标x和y。这个目标点将正好在centerBall的外边缘。

从这里开始，你只是应用基本的弹性代码以使小球弹到目标点（如第8章中所描述的）。当然，一旦小球到达那个点，它不再发生碰撞并且将沿着行进的方向飞离中心球。

看看你如何使用简单的技术来完成复杂的运动。

9.4 多物体碰撞检测策略

当只有两个物体在屏幕上移动时，测试它们是非常容易的。但是当有几个物体甚至几十个物体，你需要某种策略来检测它们以致不漏掉任何一个可能发生的碰撞。而且，当你有越来越多的物体要检测时，以较高的效率来执行你的检测是非常重要的。 238

9.4.1 基本的多物体碰撞检测

当只有两个物体时，只有一种碰撞可能——A对B。3个物体时，有3种可能：A-B、B-C和C-A。4个物体会有6种碰撞可能，5个物体会有10种碰撞可能。当你有20个物体时，你需要考虑190个独立的碰撞!这意味着在你的enterFrame处理器中，需要运行含有hitTest调用或距离计算的if条件语句190次。

这已经足够多了，不需要在你的游戏计划中再添加其他没有必要的碰撞测试。但是许多第1次研究这种计算方式的人所做的额外的碰撞测试不只是一小部分的，而是必要的两倍！对于20个物体，他们做了380次if条件语句（20个影片剪辑每次要测试19个其他物体，或20×19=380）。因此，你能够明白为什么我们需要真正地理解这个内容。

www.TopSage.com

大家网 TopSage.com

要明白这个问题，让我们看一下需要做什么，并且它通常是怎样处理的。假设你有6个精灵，命名为spript0、sprite1、sprite2、sprite3、sprite4和sprite5。你可以让它们移动、回弹或其他的什么，你需要知道的是何时有其中的一个影片剪辑碰到其他的剪辑。当你充分考虑之后，一般需要做的是创建两个嵌套的for循环。外层循环控制6个精灵，依次得到每一个精灵的引用，然后再循环一次，与其他每个精灵相对比。下面是一些伪代码：

```
numSprites = 6;
for(i = 0; i < numSprites; i++)
{
      spriteA = sprites[i];
      for(j = 0; j < numSprites; j++)
      {
            spriteB = sprites[j];
            if(spriteA.hitTestObject(spriteB))
            {
                  // 做任何事情。
            }
      }
}
```

对于这6个精灵有36个碰撞测试，看上去好像是合理的，是这样的吗？好，这段代码有两个很严重的问题。

首先，看看第1次循环发生了什么。变量i和j的值都为0。因此spriteA将保存一个对sprite0的引用，spriteB也一样。你正在测试的是精灵与其自身的碰撞啊！无语。好，你可以确保在执行测试前spriteA != spriteB，或者你可以直接让i !=j。那么代码如下所示：

```
numSprites = 6;
for(i = 0; i < numSprites; i++)
{
      spriteA = sprites[i];
      for(j = 0; j < numSprites; j++)
      {
            spriteB = sprites[j];
239         if(i != j && spriteA.hitTestObject(spriteB))
            {
                  // 做任何事情。
            }
      }
}
```

好的，你已经用它去掉了6个碰撞测试，现在你只有30个碰撞测试。但这些还是很多。

让我们跟踪一下你正在做的测试。你正在对比下面的精灵：

```
sprite0   with   sprite1, sprite2, sprite3, sprite4, sprite5
sprite1   with   sprite0, sprite2, sprite3, sprite4, sprite5
sprite2   with   sprite0, sprite1, sprite3, sprite4, sprite5
sprite3   with   sprite0, sprite1, sprite2, sprite4, sprite5
sprite4   with   sprite0, sprite1, sprite2, sprite3, sprite5
sprite5   with   sprite0, sprite1, sprite2, sprite3, sprite4
```

看一下第1个测试：sprite0与sprite1。现在看一下第2行的第1个测试：sprite1与sprite0。

www.TopSage.com

好，它们是相同的，不是吗？如果sprite0没有碰到sprite1，sprite1必定不会碰到sprite0。或如果一个精灵碰到了另一个精灵，你可以确定另一个精灵也碰到了前面的精灵。这表明你有太多重复的检测。如果你移除了所有的重复检测，你得到的是这样的一张表：

```
sprite0   with   sprite1, sprite2, sprite3, sprite4, sprite5
sprite1   with   sprite2, sprite3, sprite4, sprite5
sprite2   with   sprite3, sprite4, sprite5
sprite3   with   sprite4, sprite5
sprite4   with   sprite5
sprite5   with   nothing!
```

你可以看到，在第1轮测试中，测试的是sprite0与其他每一个精灵的碰撞情况。因此其他精灵没有必要再与sprite0做碰撞测试。你可以将sprite0移出列表了。那么sprite1再与剩下的精灵进行测试，然后将sprite1移出列表。到最后一个sprite5时，其他每一个精灵都已经与它做过了碰撞测试。因此，没有必要将它与任何一个精灵做测试了。最终的结果是你做了15次的碰撞测试。所以，你现在可以明白为什么我刚才说最初的解是所需要的两倍了。

现在，如何用代码实现呢？好，你仍需要两个嵌套的for循环。但现在的代码如下：

```
numSprites = 6;
for(i = 0; i < numSprites - 1; i++)
{
      spriteA = sprites[i];
      for(j = i + 1; j < numSprites; j++)
      {
            spriteB = sprites[j];
            if(spriteA.hitTestObject(spriteB))
            {
                  // 做任何事情。
            }
      }
}
```

240

注意，第1个外层的for循环现在是执行到比剪辑的总数小1的数。就像你刚刚在最后的对比列表中所看到的一样，你不需要将最后一个剪辑与任何一个剪辑测试。因为它已经被测试过了。

在内层循环中，你总是以高于外部循环的索引开始。这是因为你已经测试过了低索引的精灵，并且你不想测试相同的索引，就像你之前所看到的，一个剪辑与其自身的测试。那么使用这种方式可以让你去掉这种测试。最终的结果是只有少数几个字符与原代码不同，但给你的是100%性能的提高。

还有，除性能影响之外，在许多情况下，重复的检测可能会产生意想不到的结果。当你在检测碰撞时，如果你改变了速度或其他的值，你最终可能是改变它两次，这导致没有人知道它会是一种什么样的效果。而通常我们希望的只是一个碰撞产生一个动作结果。

9.4.2 多物体弹性

让我们来快速做一个另外的应用程序。还是气泡类型的反应，但这一次，所有的气泡可以彼此回弹。图9-7显示了效果。

www.TopSage.com

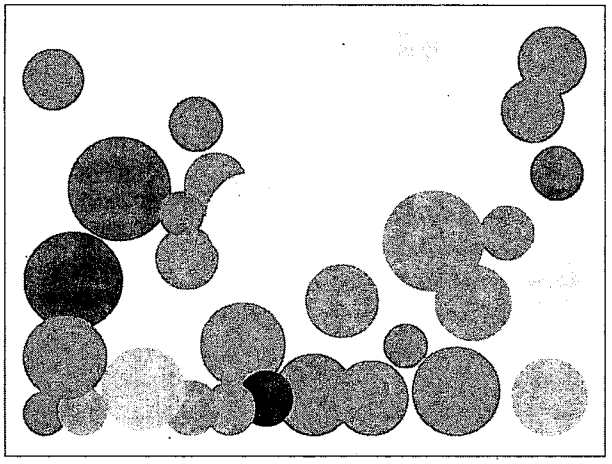


图9-7　多物体碰撞

这里是文档类Bubbles2.as中的代码：

```
package
{
    import flash.display.Sprite;
241 import flash.events.Event;
    public class Bubbles2 extends Sprite
    {
        private var balls:Array;
        private var numBalls:Number = 30;
        private var bounce:Number = -0.5;
        private var spring:Number = 0.05;
        private var gravity:Number = 0.1;


        public function Bubbles2()
        {
            init();
        }

        private function init():void
        {
            balls = new Array();

            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball = new Ball(Math.random() * 30 + 20,
                                         Math.random() * 0xffffff);
                ball.x = Math.random() * stage.stageWidth;
                ball.y = Math.random() * stage.stageHeight;
                ball.vx = Math.random() * 6 - 3;
                ball.vy = Math.random() * 6 - 3;
                addChild(ball);
                balls.push(ball);
            }
```

www.TopSage.com

```
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    for(var i:uint = 0; i < numBalls - 1; i++)
    {
        var ball0:Ball = balls[i];
        for(var j:uint = i + 1; j < numBalls; j++)
        {
            var ball1:Ball = balls[j];
            var dx:Number = ball1.x - ball0.x;
            var dy:Number = ball1.y - ball0.y;
            var dist:Number = Math.sqrt(dx * dx + dy * dy);
            var minDist:Number = ball0.radius + ball1.radius;
            if(dist < minDist)
            {
                var angle:Number = Math.atan2(dy, dx);
                var tx:Number = ball0.x + Math.cos(angle) *
                                minDist;
                var ty:Number = ball0.y +
                                Math.sin(angle) *
                                minDist;
                var ax:Number = (tx - ball1.x) *
                                spring;
                var ay:Number = (ty - ball1.y) *
                                spring;
                ball0.vx -= ax;
                ball0.vy -= ay;
                ball1.vx += ax;
                ball1.vy += ay;
            }
        }
    }

    for(i = 0; i < numBalls; i++)
    {
        var ball:Ball = balls[i];
        move(ball);
    }
}

private function move(ball:Ball):void
{
    ball.vy += gravity;
    ball.x += ball.vx;
    ball.y += ball.vy;
    if(ball.x + ball.radius > stage.stageWidth)
    {
        ball.x = stage.stageWidth - ball.radius;
        ball.vx *= bounce;
    }
    else if(ball.x - ball.radius < 0)
    {
```

242

www.TopSage.com

```
                ball.x = ball.radius;
                ball.vx *= bounce;
            }
            if(ball.y + ball.radius > stage.stageHeight)
            {
                ball.y = stage.stageHeight - ball.radius;
                ball.vy *= bounce;
            }
            else if(ball.y - ball.radius < 0)
            {
                ball.y = ball.radius;
                ball.vy *= bounce;
            }
        }
    }
}
```

243

这里，你只是使用双层嵌套的for循环来执行碰撞检测。在这个例子中，反应需要一些额外的解释。下面是碰撞反应代码：

```
if(dist < minDist)
{
    var angle:Number = Math.atan2(dy, dx);
    var tx:Number = ball0.x + Math.cos(angle) * minDist;
    var ty:Number = ball0.y + Math.sin(angle) * minDist;
    var ax:Number = (tx - ball1.x) * spring;
    var ay:Number = (ty - ball1.y) * spring;
    ball0.vx -= ax;
    ball0.vy -= ay;
    ball1.vx += ax;
    ball1.vy += ay;
}
```

记住一旦在ball0和ball1之间发生碰撞，上面的代码将运行。实际上，它和之前的有不能移动的中心球的那个示例一样。现在，让ball0使用中心球的位置。你找到两者之间的角度，然后计算出目标点x和y。这个点就是需要放置ball1的位置以使两个球不发生接触。基于这些，你可以获取ball1弹到那一点的x和y加速度。这就是ax和ay。

但是可以用一些小技巧。在这个例子中，不光ball1需要弹性离开ball0，还有ball0必须弹性离开ball1。它们的加速度大小将是相同的只是方向是相反的。因此不需要计算两次加速度，只需加入ax和ay到ball1的速度，并从ball0的速度中减去ax和ay加速度。你会得到相同的结果，并且节省了许多计算。你可能会认为这使最终加速度加倍了，因为它被应用了两次。的确如此。为了补偿，需要让spring变量比通常的值小很多。

虽然我们正在讨论优化技巧，但我想提一下另一点。先前的代码在计算角度时使用的是Math.atan2，然后使用Math.cos和Math.sin来计算目标点：

```
var angle:Number = Math.atan2(dy, dx);
var tx:Number = ball0.x + Math.cos(angle) * minDist;
var ty:Number = ball0.y + Math.sin(angle) * minDist;
```

但要注意的是正弦是对边比斜边，而余弦是邻边比斜边。并且要意识到角的对边是dy，邻边

www.TopSage.com

大学网 TopSage.com

是dx，斜边是dist。因此你可以将上面3行简化为两行：

```
var tx:Number = ball0.x + dx / dist * minDist;
var ty:Number = ball0.y + dy / dist * minDist;
```

瞧！你已经将三角函数替换为简单的除法。

在继续向下之前，花点时间继续试验一下弹性气泡示例。你可以调节许多变量。试着改变小球的弹性、重力、个数和大小。你可能还希望加入一些摩擦力或一些鼠标的交互。 244

9.5 其他的碰撞检测方法

内置的hitTest方法和基于距离的碰撞检测并不是查出一个物体是否碰到另一个物体的仅有的方法，但是它们两者，会帮你解决绝大多数的碰撞检测问题。

如果你做了足够多的搜索，你可能会发现一些聪明的开发人员已经提出了其他几种检测两个物体何时碰撞的巧妙方法。例如，Grant Skinner已经提出了使用多个BitmapData对象来检测各种物体的像素何时并在哪产生覆盖的方法。你可以在www.gskinner.com找到的。

9.6 本章重点公式

现在来回顾一下本章提到的两个重点公式。

基于距离的碰撞检测：

```
// 以spriteA和spriteB开始，如果使用的是平板精灵或没有radius属性的物体，你可以使用width或height除以2。
var dx:Number = spriteB.x - spriteA.x;
var dy:Number = spriteB.y - spriteA.y;
var dist:Number = Math.sqrt(dx * dx + dy * dy);
if(dist < spriteA.radius + spriteB.radius)
{
    //处理碰撞。
}
```

多物体碰撞检测：

```
var numObjects:uint = 10; // for example
for(var i:uint = 0; i < numObjects - 1; i++)
{
    //使用变量i获取引用，例如：
    var objectA = objects[i];

    for(var j:uint = i+1; j<numObjects;j++)
    {
        //使用变量j获取引用，例如：
        var objectB = objects[j];
        // 执行碰撞检测，在objectA和objectB之间。
    }
}
```

245

9.7　小结

本章主要讲解了你需要知道的关于碰撞检测的所有内容，包括各种内置的hitTest函数、基于距离的碰撞检测和如何有效地跟踪多个物体的碰撞。你需要知道每种方法中的加与减和每种情况工作的好坏。在本书后面的内容中你需要用到这里的所有内容。毫无疑问，你将要在你的项目中继续扩展它。

在下一章中，你将看到物体与有角度的平面之间的碰撞，使用的技术称为坐标旋转。这项技术也将在第11章中用到，在那里，你将学会如何对那些你知道如何检测的碰撞创建一个真实的碰撞反应。

246

www.TopSage.com

第10章

坐标旋转和角度回弹

本章内容

- 简单的坐标旋转
- 高级坐标旋转
- 沿角度回弹
- 本章重点公式

本章讲解了一项特殊的技术，称为*坐标旋转*（coordinate rotation）。顾名思义，这项技术是指围绕一个点的坐标旋转。对于它的用处，坐标旋转是几种有趣的效果不可或缺的技术。其中之一是多年来一直在Flash论坛上问的问题“我怎样才能从一个有角度的表面回弹物体？”的答案。在本章中我将讲解如何实现它。

坐标旋转的另一个用处是用来计算彼此回弹的两个物体的反应。我将在下一章中展示如何计算，那里将讨论动量守恒。在下一章中，我将假设你已经了解了坐标旋转。因此，如果你想直接跳到下一章，建议先看这一章。

10.1 简单的坐标旋转

尽管我在第3章中讨论三角学时已经涉及了简单的坐标旋转技术，但还是让我们快速回顾一下。有一个中心点、一个物体、一个半径和一个角。你只是不断地增加或减小角度，然后使用基本的三角学来围着中心点放置物体。你可以设置一个这样的像vr（旋转的速度）变量来控制角度增加或减小多少。并且不要忘记角使用的是弧度。

代码结构像这样：

```
vr = 0.1;
angle = 0;
radius = 100;
centerX = 250;
centerY = 200;

// 在enterFrame处理器中的代码。
   sprite.x = centerX + cos(angle) * radius;
   sprite.y = centerY + sin(angle) * radius;
   angle += vr;
```

www.TopSage.com

你只是使用了基本的三角学基于角和半径来设置物体的x和y属性，并在每一帧上改变它的角。让我们创建两个示例影片来演示它的动作。

这里是第1个示例，文档类Rotate1.as：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Rotate1 extends Sprite
    {
        private var ball:Ball;
        private var angle:Number = 0;
        private var radius:Number = 150;
        private var vr:Number =0.05;
        public function Rotate1()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            ball.x = stage.stageWidth / 2 + Math.cos(angle) * radius;
            ball.y = stage.stageHeight / 2 + Math.sin(angle) * radius;
            angle += vr;
        }
    }
}
```

250

可以看到，这些代码对你来说并不是什么新代码。继续试验它。当你知道或可以从一个中心点指定一个角度和半径时这种方法会工作得很好。

但是如果你知道的只是物体的位置和中心点的位置会怎样呢？好，很自然，要计算出基于*x*和*y*位置的当前角度和半径并不难。代码像这样：

```
var dx:Number = ball.x - centerX;
var dy:Number = ball.y - centerY;
var angle:Number = Math.atan2(dy, dx);
var radius:Number = Math.sqrt(dx * dx + dy * dy);
```

对于单个物体，这种基于坐标的旋转方法可能是很好的，尤其是在只需要计算出角度和半径一次的情况下。但是在更加动态的例子中，你可能有许多物体要旋转，并且它们与中心旋转点的相对位置可能是变化的。因此，对于每一个物体，需要计算出它的距离、角度和半径，然后加入vr到角度，最后计算出新的x和y位置。这看上去并不是很好，也可能不够高效。但幸运的是，还有更好的办法。

www.TopSage.com

10.2 高级坐标旋转

如果你打算围绕一个点旋转物体，你开始只是需要它们的位置，我已经为你准备了公式。公式只需要与中心点和角度相关的物体的*x*、*y*位置。它返回物体相对于中心的新的*x*、*y*位置。这里是基本的公式：

```
x1 = cos(angle) * x - sin(angle) * y;
y1 = cos(angle) * y + sin(angle) * x;
```

251

如果看上去只是一些字母和符号而恰巧排列方式又有某种对称性，感觉上还是不错的。这是我第1次使用这个公式的感觉。实际上，尽管我经常使用这个公式，但我仍认为它是很漂亮的。我已经多次坐下来用图来研究正弦和余弦、*x*轴的和*y*轴的、加和减等三角学和坐标方面的工作到底是怎样的。每次我这样做时，我就完全明白了。30分钟之后，它看上去就是有很好对称性的一串符号。

尽管我通常会努力地给你对技术的完整概念的理解，但是如果我在这里这样做可能要算是一个虚伪的人。因为，就个人而言，我只是记住了这个公式以致我睡觉时都可以把它敲出来。好的消息是如果你知道的关于三角学方面的知识比我多，你可能会更深入地理解这项技术。但是即使你并不是很精通，你可以记住这个公式并且总是可以得到很好的结果。

那么，接下来让我们看一下这个公式在说什么，我们在图10-1中做了演示。当然，*x*和*y*代表着你要旋转的物体的坐标。而且，它们是相对于要围绕其旋转的中心点的物体坐标。因此，如果你的中心点在200,100，而你的物体是在300,150的位置，那么*x*将是300–200或100，而*y*将是150–100或50。

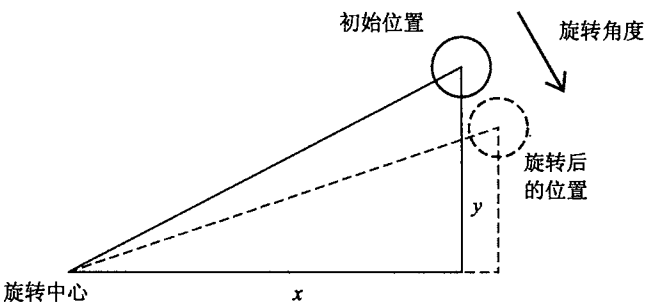


图10-1 旋转坐标

在这个时候所旋转的物体角度是多少。它不是当前的角度或最终的角度，而是两者的差。换句话说，我们谈论的只是某一时刻的角度，如果物体从当前的中心点算起在45度角位置，而这里需要旋转的角度是5度，你将再旋转5度将它放在50度的位置。但是要记住，在这项技术中，你可能不知道并且不会关心初始和最终的角度。你感兴趣的只是当前旋转了多少角度。另外，我肯定我没必要提到它，但是通常，这个角度将是一个弧度，而不是度。

好的，那么让我们来看一下动作脚本。

www.TopSage.com

大家网 TopSage.com

10.2.1　旋转一个物体

这个示例将一个小球放在一个随机的位置，然后使用我们刚才讨论的技术来旋转它（文档类 Rotate2.as）：

252

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Rotate2 extends Sprite
    {
        private var ball:Ball;
        private var vr:Number = .05;
        private var cos:Number = Math.cos(vr);
        private var sin:Number = Math.sin(vr);

        public function Rotate2()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.x = Math.random() * stage.stageWidth;
            ball.y = Math.random() * stage.stageHeight;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var x1:Number = ball.x - stage.stageWidth / 2;
            var y1:Number = ball.y - stage.stageHeight / 2;
            var x2:Number = cos * x1 - sin * y1;
            var y2:Number = cos * y1 + sin * x1;
            ball.x = stage.stageWidth / 2 + x2;
            ball.y = stage.stageHeight / 2 + y2;
        }
    }
}
```

这里，你设置了vr为和你之前使用的相同的值0.5。然后你计算出这个角的正弦和余弦值。因为在本例中它是不变的，你只需要在类的顶端计算一次，而不用在每一帧重复地计算它。x1和y1是相对于围绕的中心点——舞台的中心来计算的。然后你应用刚才谈论到的坐标旋转公式。这样就得出了x2和y2，即小球的新的坐标位置。另外，它是相对于中心点的，因此你需要将x2和y2与中心点相加得出最终的小球位置。

尽量地去试验它，你会发现它与早期版本工作起来是一样的。现在，我确信你在想如果它与以前的版本工作是一样的，为什么还要这么麻烦使用新的公式，这实际上看起来更复杂？好，在

这样一个非常简单的例子中，可能你是对的。但是让我们看一下使事情简化了的一些情况。首先，假设旋转的是多个物体。 253

10.2.2 旋转多个物体

假设有多个物体要旋转，如有很多个精灵在一个称为sprites的数组中。for循环像这样：

```
for(var i:uint = 0; i < numSprites; i++)
{
    var sprite:Sprite = sprites[i];
    var dx:Number = sprite.x - centerX;
    var dy:Number = sprite.y - centerY;
    var angle:Number = Math.atan2(dy, dx);
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    angle += vr;
    sprite.x = centerX + Math.cos(angle) * dist;
    sprite.y = centerY + Math.sin(angle) * dist;
}
```

而高级坐标旋转方法是像这样的：

```
var cos:Number = Math.cos(vr);
var sin:Number = Math.sin(vr);
for(var i:uint = 0; i < numSprites; i++)
{
    var sprite:Sprite = sprites[i];
    var x1:Number = sprite.x - centerX;
    var y1:Number = sprite.y - centerY;
    var x2:Number = cos * x1 - sin * y1;
    var y2:Number = cos * y1 + sin * x1;
    sprite.x = centerX + x2;
    sprite.y = centerY + y2;
}
```

注意第1个版本在循环中调用了4个Math函数，意味着每一个物体在旋转时都要调用这4个Math函数。第2个版本只调用了两个Math函数，而且都是在循环外面，这意味着它们只需要执行一次，而不管有多少个物体。因此，如果你有30个精灵，使用第1个版本你需要在每一帧上调用120个Math函数，而在第2个版本中被压缩成2次。你可以确定哪一个更有效了。

在前面的示例中，你可以在enterFrame处理器之外移除sin和cos的计算。这是因为你使用的是固定的角度，所以你可以设置完后忘记它。然而在许多情况下，这些角度可能是变化的，那么你需要在每次它们改变时重新计算正弦和余弦值。

要演示这些最新的概念，让我们快速创建一个使用鼠标位置控制多个物体的旋转速度的例子。如果鼠标在屏幕的中心，则没有旋转发生。当它向左移动时，物体将快速地沿着逆时针方向移动。当鼠标向右移动时，它们向顺时针方向旋转。这个示例与前面的示例非常相似，不同的只是你要创建Ball的多个实例，并将它们保存在一个名为balls的数组中。这里是文档类(Rotation3.as)： 254

www.TopSage.com

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Rotate3 extends Sprite
    {
        private var balls:Array;
        private var numBalls:uint = 10;
        private var vr:Number = .05;

        public function Rotate3()
        {
            init();
        }

        private function init():void
        {
            balls = new Array();
            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball = new Ball();
                balls.push(ball);
                addChild(ball);
                ball.x = Math.random() * stage.stageWidth;
                ball.y = Math.random() * stage.stageHeight;
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var angle:Number = (mouseX - stage.stageWidth / 2) * .001;
            var cos:Number = Math.cos(angle);
            var sin:Number = Math.sin(angle);
            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball = balls[i];
                var x1:Number = ball.x - stage.stageWidth / 2;
                var y1:Number = ball.y - stage.stageHeight / 2;
                var x2:Number = cos * x1 - sin * y1;
                var y2:Number = cos * y1 + sin * x1;
                ball.x = stage.stageWidth / 2 + x2;
                ball.y = stage.stageHeight / 2 + y2;
            }
        }
    }
}
```

255

你可以看到这些代码并不都是那么复杂的。如果你能做到的话，可以试着使用角度和半径的方法重新写代码，并观看它是变好了还是变坏了和它是如何执行的。

在第15章讨论3D时将要重新使用这个公式。实际上，你将在同一个方法中使用它两次，以在2个轴向和3个轴向上旋转物体。但是现在还不想吓到你。你在学到3D之前还有许多要学

www.TopSage.com

的内容。

10.3 沿角度回弹

记得当我第一次沉迷于Flash、数学和物理时，就已经解决了物体在墙上、地板和天花板上的回弹。如果一个障碍物是固定水平的或垂直的，我知道怎样去做。但是没有多长时间我就感到厌烦了。在现实的情况中，一些东西并不总是水平或垂直的。有时候，物体是有角度的。我不知道如何在Flash中像在平的表面上回弹一样轻松地模拟这个。我曾在各种在线的Flash论坛询问。我并不是唯一问这个问题的人。我记得在一个板块上有3个与本节标题相同的帖子。

一些数学专家尝试着回答了这个问题。物体的反射角与入射角是相同的。我记得一个非常直观的公式可以告诉你一个移动物体碰撞到有角度的表面后行进的角度。就其本身而言，这是好的。但它只能解决一部分的问题。如果你回过头讨论障碍物的回弹，需要包括下面几个步骤：

(1) 确定你何时通过一个边界；

(2) 重置物体以使它直接靠在边界位置上；

(3) 在碰撞的轴向上反转速度。

知道最终的角度就解决了第3步的一半问题。但它没有给出一个方法来解决何时与有角度的表面发生碰撞，或将物体放在哪以致它在回弹之前看起来像是已经停在了表面上。看上去没有人能回答所有的这些问题。我试验了所有我能想到的。我绘制了许多的图表，写了许多的测试代码。我开始沮丧了。如果表面是平的它是非常简单的。但是当它们被旋转后，是很复杂的。

当时，www.illogicz.com的Stuart Schoneveld有一个惊人的在线物理引擎可以平滑地控制这种碰撞。我问他是怎样做到的。尽管他没有给我展示任何代码，但他用一两句话说出了实现的大体思路。当时，我突然感觉自己像是个白痴。

他所告诉我的内容是这样的："沿角度回弹？只需要旋转系统让表面是平的，做你的回弹，然后再将它旋转回去。"

256

哇！这就是我所需要的全部。你需要做的是旋转整个系统使你有角度的表面变成平的。这意味着旋转表面、旋转物体坐标、旋转物体的速度向量。

现在，旋转速度看上去有些复杂，但是仔细考虑一下。你已经将速度保存在*vx*和*vy*变量中。*vx*和*vy*只是定义了一个向量，它有角度和大小，或是长度。如果你知道了角度，可以直接旋转它。但如果你只知道*vx*和*vy*，可以对它应用高级坐标旋转公式而且也会得到同样的结果，就像你处理小球的位置一样。

一些图表可能帮你更直观地理解它。在图10-2中，你可以看到有角度的表面，小球很明显碰到了表面，向量箭头指出了小球的方向和速度。

在图10-3中，你看到所有的东西都被旋转了并且表面现在是水平的，就像是最初的回弹示例中的最下端的屏障。注意速度向量也随着其他所有东西的旋转而发生了旋转。

按照现在的图来看，执行回弹是非常简单的，不是吗？调整位置，然后改变*y*速度，如图10-4所示。

现在小球已经有了新的位置和速度。下一步，将所有的东西旋转回原来的角度，如图10-5所示。

www.TopSage.com

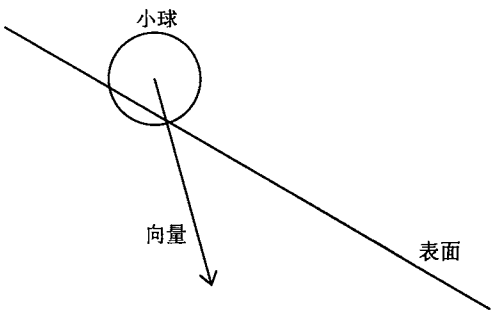


图10-2　小球碰到有角度的表面

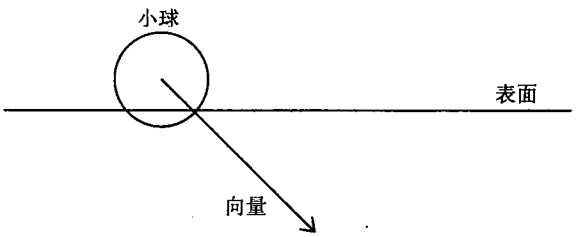


图10-3　同一个场景，旋转后

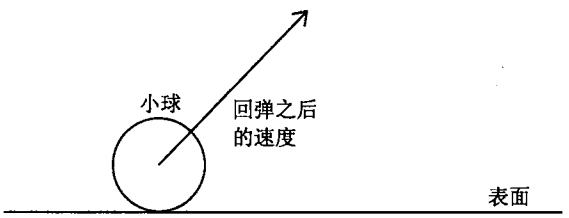


图10-4　回弹之后

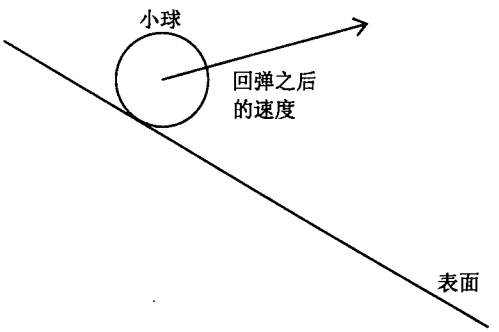


图10-5　旋转回去之后

瞧，你已经检测了碰撞，调整了位置，并改变了速度，所有这些都是在有角度的表面上完成的。如果你明白了这些图示，接下来让我们看一些代码和真正的图形。

10.3.1　执行旋转

首先，你需要某个东西来充当一个有角度的表面。这更需要你的眼力而不只是数学。对于平坦表面的回弹，你可以使用舞台的边界，它本身就具有可视的位置。而对于有角度的表面，使用一个有角度的真实的线条是很有帮助的，这样你可以看到小球在上面回弹。

257
~
258

对于此，只需要创建一个精灵，将它加入到显示列表中，然后在其中使用**drawing API**绘制一条水平线。然后可以旋转一点使它成为一个有角度的表面。

同样还需要Ball类，你应该知道它可以很方便地拿来使用。当定位所有的东西时，确保小球在这条线上方，这样就可以让它落到线上了。

这里是文档类（AngleBounce.as）：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class AngleBounce extends Sprite
```

www.TopSage.com

TopSage.com

```
{
    private var ball:Ball;
    private var line:Sprite;
    private var gravity:Number = 0.3;
    private var bounce:Number = -0.6;

    public function AngleBounce()
    {
        init();
    }

    private function init():void
    {
        ball = new Ball();
        addChild(ball);
        ball.x = 100;
        ball.y = 100;

        line = new Sprite();
        line.graphics.lineStyle(1);
        line.graphics.lineTo(300, 0);
        addChild(line);
        line.x = 50;
        line.y = 200;
        line.rotation = 30;

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        // 标准的运动代码。
        ball.vy += gravity;
        ball.x += ball.vx;
        ball.y += ball.vy;
        // 获得角、正弦和余弦。
        var angle:Number = line.rotation * Math.PI / 180;
        var cos:Number = Math.cos(angle);
        var sin:Number = Math.sin(angle);

        // 获取小球相对于线条的位置。
        var x1:Number = ball.x - line.x;
        var y1:Number = ball.y - line.y;

        // 旋转坐标。
        var x2:Number = cos * x1 + sin * y1;
        var y2:Number = cos * y1 - sin * x1;

        // 旋转速度。
        var vx1:Number = cos * ball.vx + sin * ball.vy;
        var vy1:Number = cos * ball.vy - sin * ball.vx;

        // 使用旋转后的值执行回弹。
        if(y2 > -ball.height / 2)
```

259

www.TopSage.com

```
            {
                y2 = -ball.height / 2;
                vy1 *= bounce;
            }

            // 将所有东西旋转回去。
            x1 = cos * x2 - sin * y2;
            y1 = cos * y2 + sin * x2;
            ball.vx = cos * vx1 - sin * vy1;
            ball.vy = cos * vy1 + sin * vx1;
            ball.x = line.x + x1;
            ball.y = line.y + y1;
        }
    }
}
```

开始先声明小球的变量、线条、重力和回弹。enterFrame处理器开始是基本的运动代码。

然后获得线条的角度并且将它转换为弧度。一旦你有了角度，你就得到了这个角度的正弦和余弦值。

然后通过用小球的位置减去线条的位置，获得与线条相对的小球初始的*x*、*y*位置。

最后，你准备旋转了。现在，当你看到下面的两行代码时，你可能会注意到有的地方看起来像是错的。

```
// 旋转坐标。
var x2:Number = cos * x1 + sin * y1;
var y2:Number = cos * y1 - sin * x1;
```

260

加和减与我所给的原始的坐标旋转公式是相反的，原始的是这样：

```
x1 = cos(angle) * x - sin(angle) * y;
y1 = cos(angle) * y + sin(angle) * x;
```

这并不是一个错误。想象一下你所做的。假设线条被旋转了17度。使用原始的公式，你还要将它旋转17度，最后成了34度。哎呀，你实际上是想将它从–17度旋转到0度。现在，你可能已经分别计算出正弦和余弦为Math.sin(-angle)和Math.cos(-angle)。但是最后，你需要原始角度的正弦和余弦，目的是要将所有东西旋转回原来的位置。

那么，并不是有两组余弦和正弦变量，你可以使用坐标旋转的一种可替换形式向反方向旋转所有东西。正如你所看到的，它只不过是反转了加和减。如果线条在17度位置，它会将任何物体都旋转–17度，放置在0度的位置或平面上。然后对速度做同样的操作。

注意你并不是真地需要旋转line精灵。另外，从你的眼中看去它几乎总是在那里的——可以让你看到小球应该向哪儿回弹。它也是存储表面的角度和位置的方便的地方，因为你可以使用代码去移动和旋转。

然后你就可以执行回弹了。使用x2、y2位置值和vx1、vy1速度值。意识到因为y2是相对于line精灵的，“最下面”的边界就是线条自身，它将是0。考虑一下小球的大小，检测y2的值是否大于0–ball.height/2。这个检测像这样：

```
if(y2 > -ball.height / 2)
```

剩下的回弹就显而易见了。

www.TopSage.com

然后你使用原始的公式将每一样物体旋转回原始位置。你已经知道了更新后的x1、y1、ball.vx和ball.vy的值。所有你要做的就是通过将x1和y1与line.x和line.y相加重置小球实例的真正位置。

花些时间去测试这个示例。尝试让线条处于不同的角度，让线条和小球处于不同的位置。确保它们工作得都很好。

10.3.2 优化代码

你已经看过一些改变一些代码可以进一步使程序优化的示例。这通常只需要做一次而不是多次，或根本不去做，除非你确信它们需要优化。

我在前面的示例中所写的代码只是为了更清晰易懂。因此，在每帧上发生的许多事情不需要代码。大多数代码只在小球真正碰到线条时才执行。多数时候，你只需基本的运动代码和检测小球是否碰到线条的少量计算。换句话说，所有你需要的数据都在if条件语句中：

```
if(y2 > -ball.height / 2)
```

261

因此，你需要y2变量。那么为了得到它，你需要x1和y1，还有sin和cos。但是如果小球没有碰到线条，你不需要x2或vx1和vy1。因此，这些可以放在if条件语句中。

另外，如果没有发生碰撞，就没有必要把任何一样物体旋转回去或重置小球的位置。因此，所有这些在if语句之后的内容也可以放在if条件语句中。这样你就得到了onEnterFrame方法的优化版本（你将会在AngleBounceOpt.as中找到代码）：

```
private function onEnterFrame(event:Event):void
{
    // 标准的运动代码。
    ball.vy += gravity;
    ball.x += ball.vx;
    ball.y += ball.vy;

    // 获得角、正弦和余弦。
    var angle:Number = line.rotation * Math.PI / 180;
    var cos:Number = Math.cos(angle);
    var sin:Number = Math.sin(angle);

    // 获得小球相对于线条的位置。
    var x1:Number = ball.x - line.x;
    var y1:Number = ball.y - line.y;

    // 旋转坐标。
    var y2:Number = cos * y1 - sin * x1;

    // 使用旋转后的值执行回弹。
    if(y2 > -ball.height / 2)
    {
        // 旋转坐标。
        var x2:Number = cos * x1 + sin * y1;

        // 旋转速度。
```

www.TopSage.com

```
            var vx1:Number = cos * ball.vx + sin * ball.vy;
            var vy1:Number = cos * ball.vy - sin * ball.vx;

            y2 = -ball.height / 2;
            vy1 *= bounce;

            // 将所有东西旋转回去。
            x1 = cos * x2 - sin * y2;
            y1 = cos * y2 + sin * x2;
            ball.vx = cos * vx1 - sin * vy1;
            ball.vy = cos * vy1 + sin * vx1;
            ball.x = line.x + x1;
            ball.y = line.y + y1;
        }
262 }
```

所有的粗体部分代码都已从if条件语句之外移到了if条件语句之内，因此只有当碰撞发生时才会执行，而不是在每一个单独的帧上都执行代码。你能想出你节省了多少CPU资源吗？这样思考问题是很重要的，特别是当你的程序越来越复杂的时候。

10.3.3　动态化

现在可以开始使脚本变得更动态一些。让我们实时调整线条的角度。你可以使用一行代码完成，在onEnterFrame方法中的第1行：

```
line.rotation = (stage.stageWidth/ 2 - mouseX) * .1;
```

现在你可以来回移动你的鼠标，那么线条将会做不同的倾斜。小球会相应地不断调整它的位置。你可以在文档类AngleBounceRotate.as中找到代码。

10.3.4　修复“脱离边界”问题

现在你可能会注意到，即使小球已经离开线条的边界，但它还是会沿着线条的角度继续滚动。这看上去让人感觉有些奇怪。但是记住小球实际上根本没有与line精灵交互。所有这些都是由数学来完成的。但结果是很精确的。这很容易让人忘记实际上并没有发生“碰撞”。并且因为小球不“知道”关于line精灵的任何信息，它不知道线条从哪开始或结束。但是你可以告诉小球线条在哪儿——使用简单的碰撞测试或更准确的回弹检测。让我们看一下这两种方法，以方便你决定使用哪一种方法。

1. 碰撞测试

找到线条位置的最简单的方法是除了基本的运动代码之外将其他的所有代码放在带有碰撞测试的if条件语句中，如下：

```
private function onEnterFrame(event:Event):void
{
    // 标准的运动代码。
    ball.vy += gravity;
    ball.x += ball.vx;
    ball.y += ball.vy;
```

www.TopSage.com

```
        if(ball.hitTestObject(line)
        {
            // 这个函数中的所有其他的内容。
        }
    }
```

这种方法非常简单并且对于很多实现来说可能已经足够了，但还有另外一种方法可以做得更精确一些，也是我比较喜欢的。当然，那也意味着它稍微复杂一些而且对CPU时间要求较多。 263

2. 边界检测

我个人认为在ActionScript中getBounds方法是没有被充分认识的函数之一。我原本考虑在第9章提到它，因为它非常适合于碰撞检测，但在那里已经有了足够多的内容，并且我发现的它的一个最好的应用就是我下面所要描述的内容，所以我决定把它放在这一章来讲。

如果你回忆一下有关碰撞测试的讨论，你应该还记得边界盒是什么。我将重述一下，以防你的记性像我一样。边界盒是由舞台上的显示对象的可视图元素所描述的矩形。如果你选中任何一个物体然后在它的周围绘制一个矩形框，使上边界接触物体的最顶端，下边界接触物体的最底端，左右边界也做同样的处理，这样你就有了一个边界盒。在Flash IDE中，当在舞台上选中一个影片剪辑时，你所看到的蓝色矩形框就是它的边界盒。

这个边界盒可以用在hitTestObject和hitTestPoint两种函数中。在那些例子中，边界盒的位置在场景背后被控制着，你从函数中得到的是true或false返回值。

getBounds函数直接给你的是那个盒子的位置和大小的数值。这里是基本的语法：

```
bounds = displayObject.getBounds(targetCoordinateSpace)
```

如你所看到的，它作为任何显示对象的方法被调用，并且它返回一个flash.geom.Retangle类的实例，很明显用它来描述矩形的大小和位置。首先，让我们看一下这个参数targetCoordinateSpace。它是指什么呢？

使用targetCoordinateSpace参数来指定从哪个角度描述这个边界盒。在多数情况下，它是指正在讨论的物体的上一级显示对象。例如，如果在你的主文档类中有一个精灵，你可以说sprite.getBounds(this)，意思是说，“在主影片的坐标系统中，给我这个精灵的边界盒。”从另一个方面讲，如果你创建或贴加一个精灵到另一个精灵中，你可能想得到的是相对于外部精灵的位置的边界盒，它可能不同于舞台上的0,0点。在这样的例子中，你可以这样做：

```
childSpite.getBounds(parentSprite);
```

这意味着你想得到的是parentSprite精灵中名为childSprite的精灵的边界盒，而且你想根据parentSprites的坐标空间来描述它。

当然，targetCoordinateSpace需要是一个显示对象，或是扩展了DisplayObject类的某个类的实例。并且主文档类、精灵和影片剪辑都是显示对象，它们都算在内。

现在，看一下getBounds函数返回的是什么。我之前说过它返回的是包括边界盒信息的Rectangle实例。之前我们处理过Rectangle类并看到它包括4个属性：x、y、width和height。虽然在这里我们可以使用这些属性，但它还包含了一些更有用的属性：left、right、top和bottom。 264

让我们试验一下它。在显示列表中加入一个Ball类的实例然后插入如下代码（不要忘记导入

www.TopSage.com

大家网 TopSage.com

flash.geom.Rectangle类）：

```
var bounds:Rectangle = ball.getBounds(this);
trace(bounds.left);
trace(bounds.right);
trace(bounds.top);
trace(bounds.bottom);
```

做一个有趣的实验，试着把第1行代码改成如下：

```
var bounds:Rectangle = ball.getBounds(ball);
```

现在，你在它自己的视点上得到了小球的边界。换句话说，与它自身的注册点相关。因为小球在绘制时是以0,0点作为中心的，left和top值将是负值并且等于-right和-bottom。还有值得注意的是，如果你不使用任何参数调用getBounds方法，你将会得到相同的结果，因为你正在检测其边界的物体将作为它的目标坐标空间使用。

现在，你可能忘记了为什么我们在开始时讨论边界了，那么现在让我回到正轨上。我们是要试着解决小球何时掉落离开线条的边界，还记得吗？那么，你可以在线条上调用getBounds方法，找到它的left和right值。如果小球的x位置小于bounds.left或大于bounds.right，那它已经离开了线条的边界。它描述起来要比刚才的复杂很多，代码在这里：

```
private function onEnterFrame(event:Event):void
{
    line.rotation = (stage.stageWidth/ 2 - mouseX) * .1;

    // 标准的运动代码。
    ball.vy += gravity;
    ball.x += ball.vx;
    ball.y += ball.vy;

    var bounds:Rectangle = line.getBounds(this);
    if(ball.x > bounds.left && ball.x < bounds.right)
    {
        // 这个函数中的所有其他内容。
    }
}
```

你可以在文件AngleBounceBounds.as文件中看到这些改变。

3. 修复“线条下方”问题

在碰撞测试或边界检测的任何一种方法中，你首先要知道小球是否在线条的附近，然后进行坐标旋转来得到调整后的位置和速度。在这一点上，你检查旋转后的y位置即y2是否有越过线条，
265 如果是，则执行回弹。

但是如果小球穿过线条下方呢？ 假设线条在舞台的中间，小球在“地板”上弹跳。如果碰撞测试或边界检测返回的是true，Flash将会认为小球刚才是在线条上回弹，并且会将小球从线条的下方移到线条的上方。

我得出的解决它的办法是对比vy1和y2，只有当vy1比较大时做回弹。我是怎样得出这个方法的呢？看一下面的图10-6。

www.TopSage.com

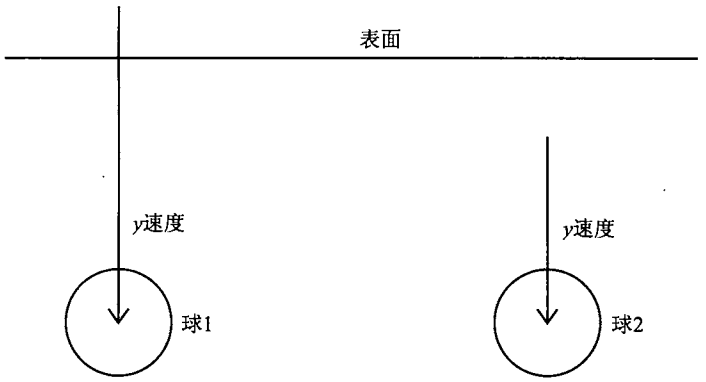


图10-6 让它穿过还是只是在线条下方?

对于左边的小球，y速度大于相对于线条的y位置。这意味着它刚才正在移动，而且它一定在线条的上方。对于右边的小球，速度小于相对的y位置。从另一个角度说，在这一帧上它在线条的下方。并且在最后一帧上它也在线条的下方。因此它只在帧的下面移动。只有当小球从线条的上方移到线条的下方时你才可以做回弹。现在，让我们看一下如何修复代码。下面是enterFrame处理器中的一部分代码：

```
// 旋转坐标。
var y2:Number = cos * y1 - sin * x1;

// 使用旋转角度执行回弹。
if(y2 > -ball.height / 2)
{
    // 旋转坐标。
    var x2:Number = cos * x1 + sin * y1;

    // 旋转速度。
    var vx1:Number = cos * ball.vx + sin * ball.vy;
    var vy1:Number = cos * ball.vy - sin * ball.vx;
...
```

你只需要将y2 < vy1加到if条件语句中：

```
if(y2 > -ball.height / 2 && y2 < vy1)
```

266

但是要做到这些，你需要预先计算vy1的值，以便在if条件语句中使用。正确的代码片段如下：

```
// 旋转坐标。
var y2:Number = cos * y1 - sin * x1;

// 旋转速度。
var vy1:Number = cos * ball.vy - sin * ball.vx;

// 使用旋转角度执行回弹。
if(y2 > -ball.height / 2 && y2 < vy1)
{
    // 旋转坐标。
    var x2:Number = cos * x1 + sin * y1;
```

www.TopSage.com

```
        // 旋转速度。
        var vx1:Number = cos * ball.vx + sin * ball.vy;
...
```

那么，你需要在每一帧上做一点额外的计算，以获得大的准确性和逼真效果——这就是折中。你可以决定它是否是必需的。假设你有一个使小球不可能到线条下方的设置。你并不需要担心这个问题，那么你可将vy1的计算移回到if条件语句中并且移除多余的检测。

在文档类AngleBounceFinal.as中，我加入了地板和墙壁的回弹，以致你最终可以看到小球到线条的下方。试着加入和去掉我们刚刚讨论的代码来观察它们的不同。

好的，我们已经回弹了，也做了动态化，我们已经有了边界！让我们进行本章最后一个且大规模的示例。

10.3.5　多角度回弹

到目前为止，你已经学会了处理单一的线条或有角度的表面。处理多个表面并没有那么复杂。你只需要制作一些表面并且循环它们。你可以把角度回弹代码抽象出来放在它自己的函数中，然后在循环中调用它。

另外，目前为止在本章所有的示例中，我已经尽力地保持简单，用尽量少的代码来展示原理。然而，下一个示例是一个完整的程序，使用了你在前面章节所见到的所有技术。

这个示例的设置类似于上几个示例，使用了同样的ball和基本的line精灵，只是线条小一些这样可以放置更多线条。我已经放置了5个线条和一个小球在舞台上。所有线条放在一个名为
267 lines的数组中。然后在舞台上排列它们。你可以在图10-7中看到结果。

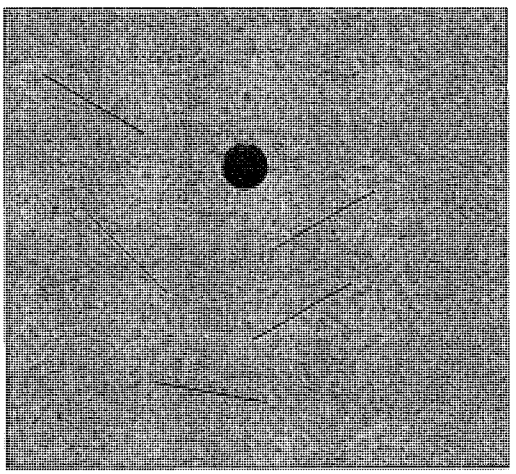


图10-7　多个线条

这里是代码（你可以在MultiAngleBounce.as中找到它）：

```
package
{
    import flash.display.Sprite;
```

www.TopSage.com

```
import flash.events.Event;
import flash.geom.Rectangle;

public class MultiAngleBounce extends Sprite
{
    private var ball:Ball;
    private var lines:Array;
    private var numLines:uint = 5;
    private var gravity:Number = 0.3;
    private var bounce:Number = -0.6;

    public function MultiAngleBounce()
    {
        init();
    }

    private function init():void
    {
        ball = new Ball(20);
        addChild(ball);
        ball.x = 100;
        ball.y = 100;
        // 创建5个线条。
        lines = new Array();
        for(var i:uint = 0; i < numLines; i++)
        {
            var line:Sprite = new Sprite();
            line.graphics.lineStyle(1);
            line.graphics.moveTo(-50, 0)
            line.graphics.lineTo(50, 0);
            addChild(line);
            lines.push(line);
        }

        // 放置并旋转它们。
        lines[0].x = 100;
        lines[0].y = 100;
        lines[0].rotation = 30;

        lines[1].x = 100;
        lines[1].y = 230;
        lines[1].rotation = 45;

        lines[2].x = 250;
        lines[2].y = 180;
        lines[2].rotation = -30;

        lines[3].x = 150;
        lines[3].y = 330;
        lines[3].rotation = 10;

        lines[4].x = 230;
        lines[4].y = 250;
        lines[4].rotation = -30;
```

268

www.TopSage.com

```
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        // 标准的运动代码。
        ball.vy += gravity;
        ball.x += ball.vx;
        ball.y += ball.vy;

        // 回弹天花板、地板和墙。
        if(ball.x + ball.radius > stage.stageWidth)
        {
269         ball.x = stage.stageWidth - ball.radius;
            ball.vx *= bounce;
        }
        else if(ball.x - ball.radius < 0)
        {
            ball.x = ball.radius;
            ball.vx *= bounce;
        }
        if(ball.y + ball.radius > stage.stageHeight)
        {
            ball.y = stage.stageHeight - radius;
            ball.vy *= bounce;
        }
        else if(ball.y - ball.radius < 0)
        {
            ball.y = ball.radius;
            ball.vy *= bounce;
        }

        // 检查每个线条。
        for(var i:uint = 0; i < numLines; i++)
        {
            checkLine(lines[i]);
        }
    }

    private function checkLine(line:Sprite):void
    {
        // 得到线条的边界盒。
        var bounds:Rectangle = line.getBounds(this);
        if(ball.x > bounds.left && ball.x < bounds.right)
        {

            // 得到角、正弦和余弦。
            var angle:Number = line.rotation * Math.PI / 180;
            var cos:Number = Math.cos(angle);
            var sin:Number = Math.sin(angle);

            // 得到小球相对于线条的位置。
            var x1:Number = ball.x - line.x;
            var y1:Number = ball.y - line.y;
```

www.TopSage.com

大家网 TopSage.com

```
                // 旋转坐标。
                var y2:Number = cos * y1 - sin * x1;

                // 旋转速度。
                var vy1:Number = cos * ball.vy - sin * ball.vx;
                // 使用旋转后的值执行回弹。
                if(y2 > -ball.height / 2 && y2 < vy1)
                {
                    // 旋转坐标。
                    var x2:Number = cos * x1 + sin * y1;

                    // 旋转速度。
                    var vx1:Number = cos * ball.vx + sin * ball.vy;

                    y2 = -ball.height / 2;
                    vy1 *= bounce;

                    // 将所有东西旋转回去。
                    x1 = cos * x2 - sin * y2;
                    y1 = cos * y2 + sin * x2;
                    ball.vx = cos * vx1 - sin * vy1;
                    ball.vy = cos * vy1 + sin * vx1;
                    ball.x = line.x + x1;
                    ball.y = line.y + y1;
                }
            }
        }
    }
}
```

270

代码的确很多，但是像我在开始所说的，所有的内容你都应当是熟悉的。复杂的程序不一定是由复杂的部分组成的，但是它们通常是由大量熟悉的部分组建成的，最后只是将它们合在一起。在这个例子中，checkLine方法的内容与之前版本的onEnterFrame中的内容是相同的。它只是在一个for循环中被调用了5次。

如果你有兴趣，你可以试着优化一些代码。假设你有许多的表面需要检测，并且你在循环它们并且检测每一个。在许多系统中，一旦你发现了一个小球正在碰撞的表面并且处理了其反应，就不需要再继续检测其他的表面了。因此你可以在那一点上跳出for循环。要做这样的事情，你可能需要让checkLine函数基于是否碰到线条而返回true或false。然后在onEnterFrame函数中的循环像这样：

```
for(var i:uint = 0; i < numLines; i++)
{
    if(checkLine(lines[i]))
    {
        break;
    }
}
```

但是在某些情况下，特别是在有密集线条的区域中，你可能需要在每一帧上检测所有的线条。因此，我把它留给你，由你来决定这样的优化是否恰当。

271

www.TopSage.com

10.4　本章重点公式

这里是本章介绍的两个主要的公式。

坐标旋转：

```
x1 = Math.cos(angle) * x - Math.sin(angle) * y;
y1 = Math.cos(angle) * y + Math.sin(angle) * x;
```

反转坐标旋转：

```
x1 = Math.cos(angle) * x + Math.sin(angle) * y;
y1 = Math.cos(angle) * y - Math.sin(angle) * x;
```

10.5　小结

如你在本章中所看到的，坐标旋转可以给你一些非常复杂的行为，但所有的这些都可以压缩为两个不会变化的公式。一旦你适应了这些公式，你可以应用在任何地方。我希望你已经开始明白如何通过加入越来越简单的技术创建出非常复杂且非常真实的动画。

在下一章你还将大量使用坐标旋转公式，在那里你将学会如何控制不同速度和质量的物体碰撞所产生的结果。

272

www.TopSage.com

第11章

台球物理

本章内容

- 质量
- 动量
- 动量守恒
- 本章重点公式

可能如你所预料的，在像这样的技术图书中，事情往往是开始简单然后渐渐变得复杂。在这一章中，你到了一个最复杂的极点。并不是说本章之后的所有内容就都不复杂了，但是我认为这一章的内容不能像以前章节那样吝惜材料。也就是说，我将一步一步地讲解概念，如果你到现在为止一直跟得很好，你就是好样的。

具体地说，这一章重点是动量：发生碰撞的两个物体的动量会发生什么、动量守恒以及如何在ActionScript中对精灵应用动量守恒。

因为这些例子中使用的物体都是圆的，为了简单，这个课题通常称为“台球物理（billiard ball physics）”，并且你很快会看到这些例子看起来确实像许多不同大小的台球互相碰撞。

像在前面的章节那样，当我说到代码时，我将先从一维空间开始以尽量保持简单易懂。然后将进入到二维空间，在这里你需要跳转到坐标旋转（前一章的内容）。实际上，你将旋转二维场景以使它变为水平。然后你可以忽略一个轴并且把它当做是一维场景。现在说这些只是要激发你对下面内容的兴趣。让我们先以简单的质量和动量概念开始。

11.1 质量

本书的前面几章已经介绍了几种形式的运动：速度、加速度、向量、摩擦力、回弹、缓动、弹性和重力。一件相当成功的事情是忽略了移动物体的质量。现在，我要在这里补充我所放弃的内容，可以说目前为止在本书中可能有几点被放弃，从学术角度说，质量应当存在于公式中。但我通常会尽量地保证正确性，并且强调确保让它看起来是正确的。最重要的是结果必须是有效的，以使它不会大量地耗用CPU。

不幸，或也可能是幸运的，我已经到了不能再忽略质量的阶段。质量是与动量捆绑的一个课

www.TopSage.com

题，我将不得不直接面对它。

那么什么是质量呢？在地球上，我们通常会将质量想成某物体有多重。这已经相当接近了。因为重量是与质量成正比的。物体的质量越大，重量就越大。实际上，我们使用相同的术语来测量质量和重量：千克、磅等等。但是从技术角度上说，质量是对物体阻止速度改变多少的衡量。因此，一个物体的质量越大，移动它就越困难或改变物体的移动方式就越困难（减速、加速或改变它的方向）。

这也与加速度和力有关。物体的质量越大，你就越需要施加给它更大的力以产生特定的加速度。我的雪佛兰汽车中的引擎被设计成能针对它自身质量产生足够的力来提供合适的加速度。它将不会为一个大型的货车提供足够的力来产生加速度。引擎将需要更大的力，因为货车的质量非
276 常大。

11.2　动量

现在我们来看动量。从根本上说，它是物体的质量和速度的乘积。动量通常用字母*p*表示，质量用*m*表示。因此，下面是相当好的一个解释：

```
p = m * v
```

这意味着一个质量较小速度很高的物体会产生与质量较大速度较慢的物体相似的动量。前面说的大货车以每小时20英里的速度移动可以很轻易地要你的命，而一颗子弹，它有相对很小的质量但有很高的速度，它同样会让人致命。

因为速度是向量（方向和大小），所以动量也一定是向量。动量向量方向与速度向量方向是相同的。因此，要完整地描述动量，你可以像这样说（m/s是指米每秒）：

```
5 kg * 20 m/s at 23 degrees
```

相当复杂，嗯？现在你应当知道为什么我等到现在才讲这项内容了。

11.3　动量守恒

最后，我们到了本章的核心内容：动量守恒。它是什么意思呢？动量是守恒的吗？什么时候？在哪里？是怎样的？好，让我们慢慢来。动量守恒往往要有一些物体的碰撞，碰撞问题现在你已经很熟悉了。你已经学习了一整章的关于碰撞检测的内容，甚至模拟了两个物体之间的一些碰撞反应。动量守恒是你对碰撞做出真实的反应所需要的原理。

动量守恒你可以说成，“在碰撞之前，这个物体以速度A移动，另外一个物体以速度B移动。现在，在碰撞之后，这个物体将以速度C移动而另一个物体将以速度D移动。”进一步地，我们知道速度包括速率和方向，你可以说，如果你知道两个物体碰撞前的速率和方向，那么就可以得出它们碰撞后的速率和方向。这是非常有用的，我确信你会同意这一点的。

但还有一点：你将需要知道每一个物体的质量。另外，你已经明白了为什么直到现在我才把它放在这里讨论。那么，实际上，你是说如果在碰撞之前知道了质量、速度和每一个物体的方向，你可以解决碰撞之后物体会去哪儿和移动的速度。

www.TopSage.com

好的，这些就是动量守恒能为你做到的。但它具体是什么呢？基本上，动量守恒定律是在说碰撞前系统的总动量等于碰撞后系统的总动量。但定律中的*系统*（system）指的是什么呢？它只是一些具有动量的物体的集合。多数的讨论也会指定这将是一个*封闭的系统*（closed system），没有任何外力影响的系统。换句话说，除了真正的碰撞你可以忽略任何东西。对于我们的目的，我们将总是只考虑两个物体之间的反应，因此我们的系统总是类似于物体A和物体B的东西。 277

系统的总动量只是系统中的所有物体的合并动量。对于我们来说，这将意味着物体A和物体B的合并动量。因此，如果你在碰撞之前合并动量，并在碰撞之后合并动量，结果是相同的。“好，那真是太好了，”你说，“但它并没有告诉我如何找到新的动量。”耐心一点，我们正要说到这儿。我告诉过你我们将一步一步地学习它。对于现在开始的一些段落，你可能想让我慢一点，因为这里就要有公式了！

在我们研究数学之前，这儿有一个建议。开始不要担心如何将它转换成真正的代码。你很快就能做到的。先从概念的视角看下面几个公式。“这个加上那个等于那个加上这个。正确，就是这样。”在本章的结尾处它会很清晰地被转成代码。

如果在碰撞前后合并的动量相同，并且动量是速度与质量的乘积，那么对于两个物体——物体0和物体1——你可以得到如下：

```
momentum0 + momentum1 = momentum0Final + momentum1Final
```

或者

```
(m0 * v0) + (m1 * v1) = (m0 * v0Final) + (m1 * v1Final)
```

现在，你想知道的是物体0和物体1的最终速度。它们是v0Final和v1Final。解一个带有两个未知数的方程的方法是找到*另一个*含有同样的两个未知数的方程。恰好在物理世界中有这样的一个方程。它与动能有关。有趣的事情是，你甚至不需要知道或关心动能是什么。你只需要借用这个公式来帮助你解决问题，然后得到你想要的。下面是动能的公式：

$$KE = 0.5 * m * v^2$$

技术上，动能不是一个向量，因此，尽管这里你是使用*v*来代表速度，但它只是用来处理速度的大小的。它与方向是无关的。但是这不会妨碍你的计算。

现在，恰好在碰撞前后的动能同样也是相同的。因此，你可以这样做：

```
KE0 + KE1 = KE0Final + KE1Final
```

或者

$$(0.5 * m0 * v0^2) + (0.5 * m1 * v1^2) = (0.5 * m0 * v0Final^2) + (0.5 * m1 * v1Final^2)$$

你可以提出0.5得到如下等式：

$$(m0 * v0^2) + (m1 * v1^2) = (m0 * v0Final^2) + (m1 * v1Final^2)$$

你知道了什么？你有了一个不同的方程，也包含同样的两个未知变量：v0Final和v1Final。你可以把这些变量提出来并为每一个未知变量得到一个方程。我将省去大家头痛的代数计算，直接给出最后的公式。如果你真地喜欢代数方程，我建议你坐下来准备好铅笔和纸张，自己动手计 278

www.TopSage.com

大家网 TopSage.com

算它。最后你会得到下面的等式（如果不是，要么是你错了，要么是我的出版社的问题）。

$$v0Final = \frac{(m0 - m1) * v0 + 2 * m1 * v1}{m0 + m1}$$

$$v1Final = \frac{(m1 - m0) * v1 + 2 * m0 * v0}{m0 + m1}$$

现在你可能会明白为什么我在本章的开头部分说你到达了一个复杂的极点。实际上，你现在还没有真正到达。你马上要将它应用在一个轴上，之后，需要深入研究并且当你移到两个轴上时要加入坐标旋转。坚持住!

11.3.1　一个轴上的动量守恒

现在已经得到了一些公式，可以使用它们让物体移动起来。对于第1个示例，你将再次使用Ball类，但是我已经加入了一个mass属性。这里是Ball类的新代码（Ball.as）：

```
package {
    import flash.display.Sprite;

    public class Ball extends Sprite {
        private var radius:Number;
        private var color:uint;
        public var vx:Number = 0;
        public var vy:Number = 0;
        public var mass:Number = 1;

        public function Ball(radius:Number=40, color:uint=0xff0000) {
            this.radius = radius;
            this.color = color;
            init();
        }
        public function init():void {
            graphics.beginFill(color);
            graphics.drawCircle(0, 0, radius);
            graphics.endFill();
        }
    }
}
```

279

你将要创建该类的两个具有不同的大小、位置和质量的小球实例。这次将忽略掉y轴，所以它们在垂直方向的位置是相同的。所以当移动开始时这些基本的设置看起来像图11-1所示。

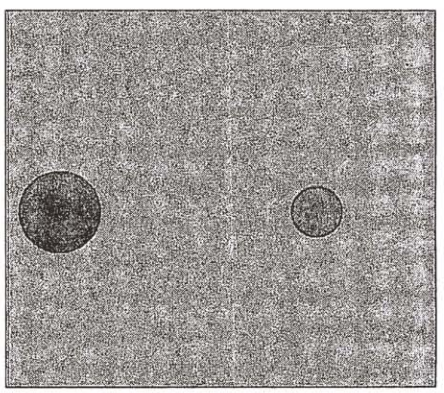


图11-1　设置一个轴上的动量守恒的舞台

类的开始只需要创建小球并且布置好它们的位置，然后对一个轴上的速度做一些基本的运动代码和简单的基于距离的碰撞检测：

www.TopSage.com

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Billiard1 extends Sprite
    {
        private var ball0:Ball;
        private var ball1:Ball;

        public function Billiard1()
        {
            init();
        }

        private function init():void
        {
            ball0 = new Ball(40);
            ball0.mass = 2;
            ball0.x = 50;
            ball0.y = stage.stageHeight / 2;
            ball0.vx = 1;
            addChild(ball0);
            ball1 = new Ball(25);
            ball1.mass = 1
            ball1.x = 300;
            ball1.y = stage.stageHeight / 2;
            ball1.vx = -1;
            addChild(ball1);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            ball0.x += ball0.vx;
            ball1.x += ball1.vx;
            var dist:Number = ball1.x - ball0.x;
            if(Math.abs(dist) < ball0.radius + ball1.radius)
            {
                // 碰撞反应代码放在这里。
            }
        }
    }
}
```

280

现在，唯一的问题是如何用代码编写它们的反应。好，先我们先看ball0。假设ball0是物体0而ball1是物体1，你需要应用下面的公式：

$$v0Final = \frac{(m0 - m1) * v0 + 2 * m1 * v1}{m0 + m1}$$

在ActionScript中，该公式变成下面的代码：

www.TopSage.com

```
var vx0Final:Number = ((ball0.mass - ball1.mass) * ball0.vx +
                       2 * ball1.mass * ball1.vx) /
                       (ball0.mass + ball1.mass);
```

并不难看出它是来源于哪儿。你可以对ball1做相同的操作。所以公式

$$v1Final = \frac{(m1 - m0) * v1 + 2 * m0 * v0}{m0 + m1}$$

变成这样：

```
var vx1Final:Number = ((ball1.mass - ball0.mass) * ball1.vx +
                       2 * ball0.mass * ball0.vx) /
                       (ball0.mass + ball1.mass);
```

281

onEnterFrame中的最终代码如下：

```
private function onEnterFrame(event:Event):void
{
    ball0.x += ball0.vx;
    ball1.x += ball1.vx;
    var dist:Number = ball1.x - ball0.x;
    if(Math.abs(dist) < ball0.radius + ball1.radius)
    {
        var vx0Final:Number = ((ball0.mass - ball1.mass) * ball0.vx +
                               2 * ball1.mass * ball1.vx) /
                               (ball0.mass + ball1.mass);
        var vx1Final:Number = ((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.x += ball1.vx;
    }
}
```

注意vx0Final的计算使用了ball1.vx，反之亦然。因此，我不得不将它们存储在临时变量中，而不是直接把它们指定给ball0.vx和ball1.vx属性。

1. 放置物体

前面的ActionScript中的最后两行代码需要解释一下。当得到了每个球的新的速度之后，将它们加到小球的位置上。这里面有一些新内容。为什么这么说呢？好，记得在之前的回弹示例中，你需要重新定位精灵的位置以使它不会嵌入到墙壁中。你只需要将它移出来以使它刚好接触到墙壁的边缘。这里需要做同样的事情，但是现在你有两个移动的物体。你不希望它们互相嵌入。因为这样不仅看起来是错的，而且通常会导致两个物体永久地粘连在一起。

你可以将其中一个小球放置在另一个小球的边缘位置。但是你要移动哪一个小球呢？不管你移动哪个小球，它都会反常地跳到它的新位置，特别是在速度较慢的情况下比较明显。

可能有很多方法可以检测小球的位置是否正确有简单的和复杂的，有准确的和不准确的。针对这里的第1个示例我使用的简单的方法是将新的速度加回物体，再次让它们彼此分开。我发现这是非常真实而且很简单的方法，只有两行代码。在后面的11.4.4节中，我将演示使用这种方法

www.TopSage.com

可能产生的问题并且给你一个更健壮的解决方法。

编译并运行Billiard1.as文档类。然后试着改变每个小球的速度和质量直到你看到有变化发生。也试着改变每个小球的大小。注意大小并不会对小球的反应产生任何影响。在多数情况下，较大的物体应当有一个较大的质量，并且你可能计算出两个小球的相对体积，然后根据它们的大小得出真实的质量。然而，通常我只是估计它们的质量，直到事情看上去感觉是正确的。我的比较科学的说法像这样，“好，这个球大约是两倍大小，那么我将给它两倍的质量”。 282

2. 优化代码

这个解决方案最糟糕的部分在于中间那个很大的等式。实际上，更糟糕的是代码中用了两次这样的几乎相同的等式。如果你能想办法去掉一个，我相信你会感觉好很多。有一个好的消息是你可以做到。

它还要用到一些数学和代数学，但这里我将不会非常细致地介绍它。基本上，你需要在条件语句之前找到两个物体的总速度。然后，当你得到一个物体的最终速度后，你可以得出它和总速度之间的差来计算出另一个物体的最终速度。

实际上你可以通过两个物体的速度相减得到总速度。这看起来可能有些奇怪，但是要从系统的视角出发仔细考虑它。假设系统是两辆在高速公路上的汽车。一个速度是50mph而另一个是60mph。根据你所在的车的不同，你可以看到另一辆车的速度是10mph或-10mph。从另一个角度说，它或者在你前面缓慢移动或者在你后面缓慢移动。

因此，在碰撞之前，你需要从ball0.vx减去ball1.vx计算出总的速度（从ball1的角度看）。

```
var vxTotal:Number = ball0.vx - ball1.vx;
```

最后，计算出vx0Final之后，将它加到vxTotal上，那么你就得到了vx1Final。另外，这看上去好像不合理，但是试一下它，你会发现它工作得很好。

```
vx1Final = vxTotal + vx0Final;
```

太神奇了！这要比可怕的两个公式好多了。还有，现在ball1.vx的公式不再引用ball0.vx了，你甚至可以去掉临时变量。下面是新的onEnterFrame方法（从文档类Billiard2.as中找到代码）：

```
private function onEnterFrame(event:Event):void
{
    ball0.x += ball0.vx;
    ball1.x += ball1.vx;
    var dist:Number = ball1.x - ball0.x;
    if(Math.abs(dist) < ball0.radius + ball1.radius)
    {
        var vxTotal:Number = 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.x += ball1.vx;
    }
}
```

283

www.TopSage.com

你已经去掉了大量的数学操作并且得到了相同的结果——不错。

现在，这并不是你必须要理解的一个公式，除非你有很好的物理学背景。你甚至可能记不住它，除非你经常使用它。就个人而言，当我想用这项技术时，我就会找到本书然后把这个公式复制下来。

11.3.2　两个轴上的动量守恒

好，做一个深呼吸。你将进入下一个阶段。到目前为止，你已经应用了一个很长的公式，但它是相当地即插即用。你得到两个物体的质量和速度，将它们插入到公式中，然后你就可以得到结果了。

现在我将再增加一层复杂性——也就是另一维。我已经在本章的开始给出了思路，所以你知道你将需要使用坐标旋转。让我们看一下为什么。

1．理解原理和算法

图11-2演示你刚才看到的示例：一维碰撞。

如你所看到的，物体有不同的大小，不同的质量和不同的速度。速度是由从每个小球的中心向外的箭头描述的。它们是向量。回想一下，一个速度向量指向运动方向，它的长度表示速率。

一维空间的示例是非常简单的，因为两个速度向量都是沿着*x*轴的。因此你只需要直接加或减它们的大小。现在，看一下图11-3，它显示了二维空间上的两个球的碰撞。

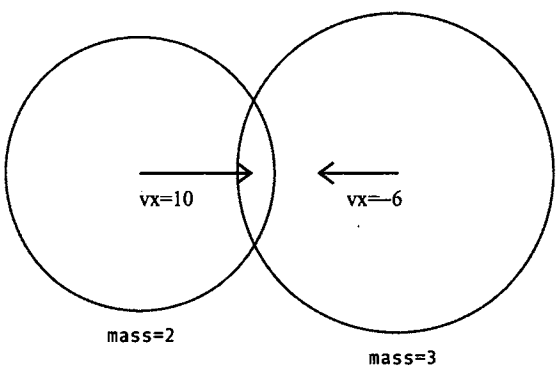


图11-2　一维碰撞

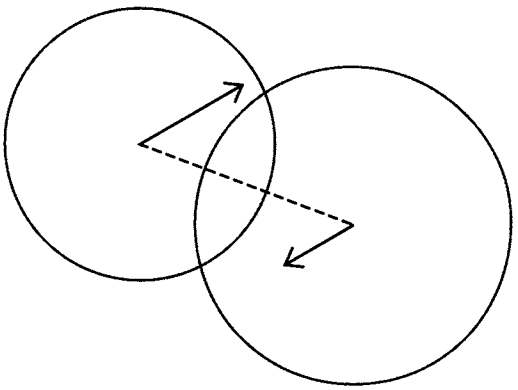


图11-3　二维碰撞

现在速度有完全不同的方向。不能只是把速度放在刚才的动量守恒公式中。那样会得出完全
284 错误的结果。那么如何解决呢？

好，你现在看到的第2张图如旋转后有点像第1张图。首先，计算出两个小球的位置形成的角度并且旋转整个场景——位置和速度——逆时钟旋转一定角度。例如，如果角是30度，旋转场景中的所有物体–30度。这与你在第10章处理的沿有角度的表面回弹是一样的。结果图像如图11-4所示。

www.TopSage.com

topsage.com

两个小球之间的角是很重要的，它不仅让事情看起来好看，而且这个角度可以被称为碰撞角（angle of collision）。它重要是因为它是你需要关注的小球速度的唯一一部分——沿角度方向的速度的一部分。

现在，看一下图11-5。这里，我已经为*vx*和*vy*两个速度加入了向量线。注意两个小球的*vx*恰好是沿着碰撞角方向的。

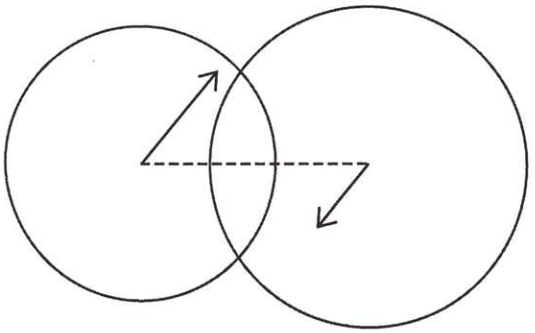


图11-4 旋转后的二维碰撞

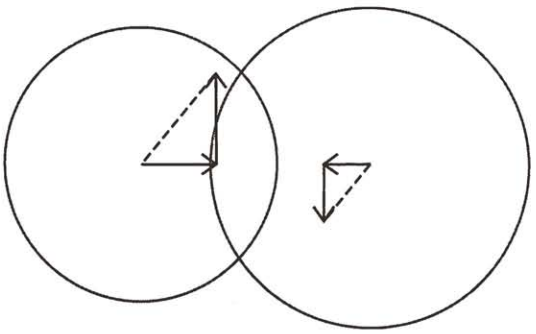


图11-5 在*x*和*y*速度上绘制向量线条

如我刚才所说的，你唯一需要关注速度的部分是沿碰撞角的部分。就是现在的*vx*。实际上，现在你可以忘记*vy*了。我把它移出图形，如图11-6所示。

这个看上去熟悉吗？它又变成第1张图了。你可以很容易地使用即插即用的动量公式来解决它。（注意我是怎样向你灌输思想的!你现在已经开始同意我说的动量守恒公式是很简单的说法了，不是吗？）

当你应用这个公式时，你会得到两个新的*vx*值。记住*vy*值是永远不会改变的。而*vx*的改变将改变整个速度，如图11-7所示。

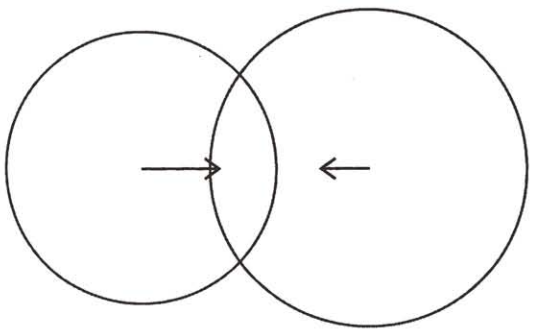


图11-6 你所要关注的是*x*速度

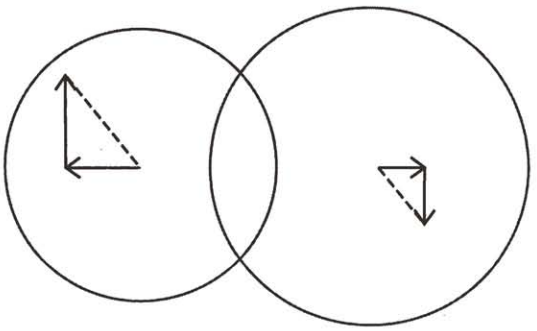


图11-7 新的*x*速度，相同的*y*速度，产生一个全新的总速度

你是否已经猜到下一步要做什么？你只需要再将所有的东西旋转回去。如图11-8所示，那么你就得到了每个小球最后真实的*vx*和*vy*。

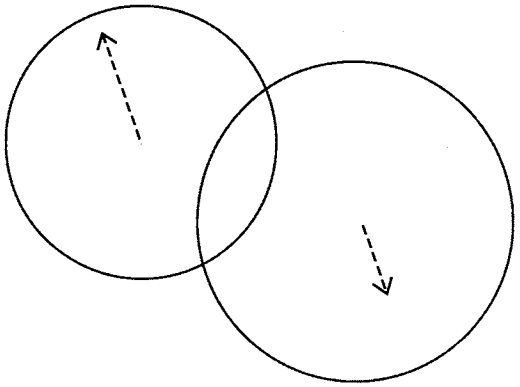


图11-8　所有的东西旋转回去

好，这看起来非常漂亮。但是现在是时候研究如何将所有这些转换成代码了。对我来说最难的部分是如何继续说服你它是简单的。

2. 写代码

在开始时，你需要创建一个允许两个小球从任意角度移动并最终互相碰撞的基文件。与之前的示例的设置相同，你有两个ball实例：ball0和ball1。让我们把它制作得大一些，像图11-9中的一样，以使它们有更多的机会碰撞。

285 ~ 287

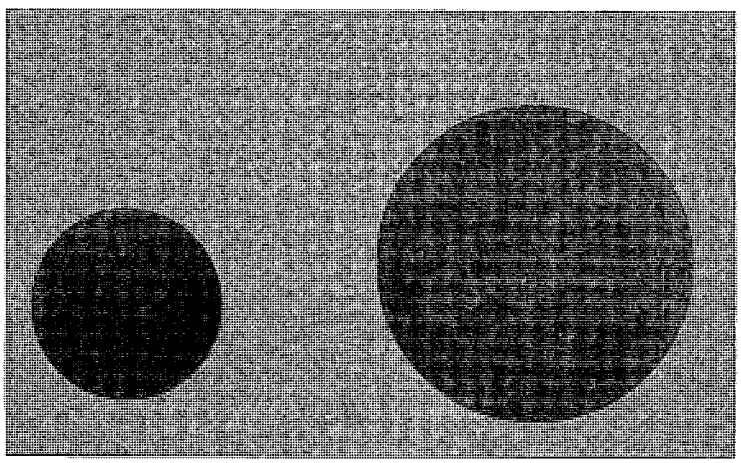


图11-9　为两个轴上的动量守恒设置舞台

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Billiard3 extends Sprite
    {
        private var ball0:Ball;
```

www.TopSage.com

```
private var ball1:Ball;
private var bounce:Number = -1.0;

public function Billiard3()
{
    init();
}

private function init():void
{
    ball0 = new Ball(150);
    ball0.mass = 2;
    ball0.x = stage.stageWidth - 200;
    ball0.y = stage.stageHeight - 200;
    ball0.vx = Math.random() * 10 - 5;
    ball0.vy = Math.random() * 10 - 5;
    addChild(ball0);
    ball1 = new Ball(90);
    ball1.mass = 1
    ball1.x = 100;
    ball1.y = 100;
    ball1.vx = Math.random() * 10 - 5;
    ball1.vy = Math.random() * 10 - 5;
    addChild(ball1);

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    ball0.x += ball0.vx;
    ball0.y += ball0.vy;
    ball1.x += ball1.vx;
    ball1.y += ball1.vy;

    checkWalls(ball0);
    checkWalls(ball1);
}

private function checkWalls(ball:Ball):void
{
    if(ball.x + ball.radius > stage.stageWidth)
    {
        ball.x = stage.stageWidth - ball.radius;
        ball.vx *= bounce;
    }
    else if(ball.x - ball.radius < 0)
    {
        ball.x = ball.radius;
        ball.vx *= bounce;
    }
    if(ball.y + ball.radius > stage.stageHeight)
    {
        ball.y = stage.stageHeight - ball.radius;
```

288

www.TopSage.com

```
                ball.vy *= bounce;
            }
            else if(ball.y - ball.radius < 0)
            {
                ball.y = ball.radius;
                ball.vy *= bounce;
            }
        }
    }
289 }
```

现在，这里的所有内容你闭着眼就可以完成。你设置了边界，设置了随机速度，设置了一些质量，根据每个小球的速度移动它们，还有检测边界。注意我把边界检测内容放在一个单独的函数中，名为checkWalls，这样我可以使用它两次而不需要再重新键入一次。

同样，我也把碰撞检测内容放在一个名为checkCollision的函数中。那么，onEnterFrame变成这样：

```
private function onEnterFrame(event:Event):void
{
    ball0.x += ball0.vx;
    ball0.y += ball0.vy;
    ball1.x += ball1.vx;
    ball1.y += ball1.vy;
    checkCollision(ball0, ball1);
    checkWalls(ball0);
    checkWalls(ball1);
}
```

现在看上去，我只需要向你展示checkCollision函数和用到的任何其他函数。剩下的代码不需要改变。如果你想看最终的完整代码，它在文件Billiard3.as中。

checkCollision函数的开始部分是很简单的。它只是一个基于距离的碰撞检测设置。

```
private function checkCollision(ball0:Ball, ball1:Ball):void
{
    var dx:Number = ball1.x - ball0.x;
    var dy:Number = ball1.y - ball0.y;
    var dist:Number = Math.sqrt(dx*dx + dy*dy);
    if(dist < ball0.radius + ball1.radius)
    {
        // 碰撞处理代码。
    }
}
```

看到了吗？现在已经有大约三分之二的代码写好了，目前为止它已经是小菜一碟了！

碰撞处理代码需要做的第1件事情是计算出两个小球之间的角度。可以使用Math.atan2(dy,dx)来计算。（如果你还是想不起来如何去做，可以回到第3章重温一下三角学）然后保存下此角度的正弦和余弦值，因为你需要不断使用它们。

```
// 计算角、正弦和余弦。
var angle:Number = Math.atan2(dy, dx);
var sin:Number = Math.sin(angle);
var cos:Number = Math.cos(angle);
```

www.TopSage.com

接下来，你需要为两个小球的速度和位置做坐标旋转。让我们称旋转后的位置为x0、y0、x1和y1，旋转后的速度为vx0、vy0、vx1和vy1。

因为你使用ball0作为“支点”它的坐标将是0,0。旋转后它的坐标不会发生改变，因此你可以说：

```
// 旋转ball0的位置。
var x0:Number = 0;
var y0:Number = 0;
```

接下来，ball1的位置是相对于ball0的位置。它相当于你已经计算出来的距离值dx和dy。所以，你可以只旋转它们得到旋转后的ball1的位置：

```
// 旋转ball1的位置。
var x1:Number = dx * cos + dy * sin;
var y1:Number = dy * cos - dx * sin;
```

最后，旋转所有的速度。你将看到下面的一个模式：

```
// 旋转ball0的速度。
var vx0:Number = ball0.vx * cos + ball0.vy * sin;
var vy0:Number = ball0.vy * cos - ball0.vx * sin;

// 旋转ball1的速度。
var vx1:Number = ball1.vx * cos + ball1.vy * sin;
var vy1:Number = ball1.vy * cos - ball1.vx * sin;
```

这里是所有的旋转代码：

```
private function checkCollision(ball0:Ball, ball1:Ball):void
{
    var dx:Number = ball1.x - ball0.x;
    var dy:Number = ball1.y - ball0.y;
    var dist:Number = Math.sqrt(dx*dx + dy*dy);
    if(dist < ball0.radius + ball1.radius)
    {
        // 计算角、正弦和余弦。
        var angle:Number = Math.atan2(dy, dx);
        var sin:Number = Math.sin(angle);
        var cos:Number = Math.cos(angle);

        // 旋转ball0的位置。
        var x0:Number = 0;
        var y0:Number = 0;

        // 旋转ball1的位置。
        var x1:Number = dx * cos + dy * sin;
        var y1:Number = dy * cos - dx * sin;

        // 旋转ball0的速度。
        var vx0:Number = ball0.vx * cos + ball0.vy * sin;
        var vy0:Number = ball0.vy * cos - ball0.vx * sin;
        // 旋转ball1的速度。
        var vx1:Number = ball1.vx * cos + ball1.vy * sin;
        var vy1:Number = ball1.vy * cos - ball1.vx * sin;
```

www.TopSage.com

大家网 TopSage.com

```
    }
}
```

它并不是那么可怕，对吗？好，先停在这儿。对于那些很难的内容你已经学了三分之一了。

现在你可以通过vx0和ball0.mass及vx1和ball1.mass来做简单的一维碰撞反应。对于之前的一维碰撞示例，你有下面的代码：

```
var vxTotal:Number = 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;
```

可以重写代码如下：

```
var vxTotal:Number = vx0 - vx1;
vx0 = ((ball0.mass - ball1.mass) * vx0 +
       2 * ball1.mass * vx1) /
       (ball0.mass + ball1.mass);
vx1 = vxTotal + vx0;
```

所有你要做的就是把ball0.vx和ball1.vx换成旋转后的版本vx0和vx1。让我们把它插入到函数中：

```
private function checkCollision(ball0:Ball, ball1:Ball):void
{
    var dx:Number = ball1.x - ball0.x;
    var dy:Number = ball1.y - ball0.y;
    var dist:Number = Math.sqrt(dx*dx + dy*dy);
    if(dist < ball0.radius + ball1.radius)
    {
        // 计算角、正弦和余弦。
        var angle:Number = Math.atan2(dy, dx);
        var sin:Number = Math.sin(angle);
        var cos:Number = Math.cos(angle);

        // 旋转ball0的位置。
        var x0:Number = 0;
        var y0:Number = 0;

        // 旋转ball1的位置。
        var x1:Number = dx * cos + dy * sin;
        var y1:Number = dy * cos - dx * sin;
        // 旋转ball0的速度。
        var vx0:Number = ball0.vx * cos + ball0.vy * sin;
        var vy0:Number = ball0.vy * cos - ball0.vx * sin;

        // 旋转ball1的速度。
        var vx1:Number = ball1.vx * cos + ball1.vy * sin;
        var vy1:Number = ball1.vy * cos - ball1.vx * sin;

        // 碰撞反应。
        var vxTotal:Number = vx0 - vx1;
        vx0 = ((ball0.mass - ball1.mass) * vx0 +
               2 * ball1.mass * vx1) /
```

292

www.TopSage.com

```
                (ball0.mass + ball1.mass);
            vx1 = vxTotal + vx0;
            x0 += vx0;
            x1 += vx1;
        }
    }
```

这些代码也加入了新的*x*速度到*x*位置，将它们分离开，如同一维示例中的处理方式。

现在你已经更新了碰撞后的位置和速度，再把所有的东西旋转回去。开始先获得未旋转的、最终的位置。

```
// 将位置旋转回去。
var x0Final:Number = x0 * cos - y0 * sin;
var y0Final:Number = y0 * cos + x0 * sin;
var x1Final:Number = x1 * cos - y1 * sin;
var y1Final:Number = y1 * cos + x1 * sin;
```

记得要交换旋转等式中的+号和–号，因为现在你要去的是另外的方向。这些“最终的”位置并不真正是最终的。它们是相对于系统支点的位置。那么，你需要将所有的这些加到ball0的位置来获得真正的屏幕位置。让我们先处理ball1，因为它使用了不需要更新的ball0的原始位置。

```
// 调节位置到屏幕位置。
ball1.x = ball0.x + x1Final;
ball1.y = ball0.y + y1Final;
ball0.x = ball0.x + x0Final;
ball0.y = ball0.y + y0Final;
```

最后，将速度旋转回去。它们可以直接应用到小球的vx和vy属性：

```
// 将速度旋转回去。
ball0.vx = vx0 * cos - vy0 * sin;
ball0.vy = vy0 * cos + vx0 * sin;
ball1.vx = vx1 * cos - vy1 * sin;
ball1.vy = vy1 * cos + vx1 * sin;
```

293

让我们看一下整个完成的函数：

```
private function checkCollision(ball0:Ball, ball1:Ball):void
{
    var dx:Number = ball1.x - ball0.x;
    var dy:Number = ball1.y - ball0.y;
    var dist:Number = Math.sqrt(dx*dx + dy*dy);
    if(dist < ball0.radius + ball1.radius)
    {
        // 计算角、正弦和余弦。
        var angle:Number = Math.atan2(dy, dx);
        var sin:Number = Math.sin(angle);
        var cos:Number = Math.cos(angle);

        // 旋转ball0的位置。
        var x0:Number = 0;
        var y0:Number = 0;

        // 旋转ball1的位置。
        var x1:Number = dx * cos + dy * sin;
```

www.TopSage.com

```
            var y1:Number = dy * cos - dx * sin;

            // 旋转ball0的速度。
            var vx0:Number = ball0.vx * cos + ball0.vy * sin;
            var vy0:Number = ball0.vy * cos - ball0.vx * sin;

            // 旋转ball1的速度。
            var vx1:Number = ball1.vx * cos + ball1.vy * sin;
            var vy1:Number = ball1.vy * cos - ball1.vx * sin;

            // 碰撞反应。
            var vxTotal:Number = vx0 - vx1;
            vx0 = ((ball0.mass - ball1.mass) * vx0 +
                   2 * ball1.mass * vx1) /
                   (ball0.mass + ball1.mass);
            vx1 = vxTotal + vx0;
            x0 += vx0;
            x1 += vx1;

            // 将位置旋转回去。
            var x0Final:Number = x0 * cos - y0 * sin;
            var y0Final:Number = y0 * cos + x0 * sin;
            var x1Final:Number = x1 * cos - y1 * sin;
            var y1Final:Number = y1 * cos + x1 * sin;

            // 将位置调整到实际的屏幕位置。
            ball1.x = ball0.x + x1Final;
            ball1.y = ball0.y + y1Final;
            ball0.x = ball0.x + x0Final;
294         ball0.y = ball0.y + y0Final;
            // 将速度旋转回去。
            ball0.vx = vx0 * cos - vy0 * sin;
            ball0.vy = vy0 * cos + vx0 * sin;
            ball1.vx = vx1 * cos - vy1 * sin;
            ball1.vy = vy1 * cos + vx1 * sin;
        }
    }
```

测试这个示例。改变Ball实例的大小、初始的速度、质量等等。确信它可以工作得很好。

至于checkCollision，它已经很不错了。但是如果你读了注释，会看到它分成了（相对）简单的几块。你也许可以再优化它，或你可以移除一些重复的部分。实际上，为了好的实践，我已经走在前面并且在文件Billiard4.as中完成了它，下面你可以看到：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Point;

    public class Billiard4 extends Sprite
    {
        private var ball0:Ball;
        private var ball1:Ball;
```

www.TopSage.com

```
private var bounce:Number = -1.0;

public function Billiard4()
{
    init();
}

private function init():void
{
    ball0 = new Ball(150);
    ball0.mass = 2;
    ball0.x = stage.stageWidth - 200;
    ball0.y = stage.stageHeight - 200;
    ball0.vx = Math.random() * 10 - 5;
    ball0.vy = Math.random() * 10 - 5;
    addChild(ball0);

    ball1 = new Ball(90);
    ball1.mass = 1
    ball1.x = 100;
    ball1.y = 100;
    ball1.vx = Math.random() * 10 - 5;
    ball1.vy = Math.random() * 10 - 5;
    addChild(ball1);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    ball0.x += ball0.vx;
    ball0.y += ball0.vy;
    ball1.x += ball1.vx;
    ball1.y += ball1.vy;
    checkCollision(ball0, ball1);
    checkWalls(ball0);
    checkWalls(ball1);
}

private function checkWalls(ball:Ball):void
{
    if(ball.x + ball.radius > stage.stageWidth)
    {
        ball.x = stage.stageWidth - ball.radius;
        ball.vx *= bounce;
    }
    else if(ball.x - ball.radius < 0)
    {
        ball.x = ball.radius;
        ball.vx *= bounce;
    }
    if(ball.y + ball.radius > stage.stageHeight)
    {
        ball.y = stage.stageHeight - ball.radius;
        ball.vy *= bounce;
```

295

www.TopSage.com

```
        }
        else if(ball.y - ball.radius < 0)
        {
            ball.y = ball.radius;
            ball.vy *= bounce;
        }
    }

    private function checkCollision(ball0:Ball, ball1:Ball):void
    {
        var dx:Number = ball1.x - ball0.x;
        var dy:Number = ball1.y - ball0.y;
        var dist:Number = Math.sqrt(dx*dx + dy*dy);
        if(dist < ball0.radius + ball1.radius)
        {
            // 计算角、正弦和余弦。
            var angle:Number = Math.atan2(dy, dx);
            var sin:Number = Math.sin(angle);
296         var cos:Number = Math.cos(angle);

            // 旋转ball0的位置。
            var pos0:Point = new Point(0, 0);

            // 旋转ball1的位置。
            var pos1:Point = rotate(dx, dy, sin, cos, true);

            // 旋转ball0的速度。
            var vel0:Point = rotate(ball0.vx,
                                    ball0.vy,
                                    sin,
                                    cos,
                                    true);

            // 旋转ball1的速度。
            var vel1:Point = rotate(ball1.vx,
                                    ball1.vy,
                                    sin,
                                    cos,
                                    true);

            // 碰撞反应。
            var vxTotal:Number = 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;

            // 更新位置。
            pos0.x += vel0.x;
            pos1.x += vel1.x;

            // 将位置旋转回去。
            var pos0F:Object = rotate(pos0.x,
                                      pos0.y,
                                      sin,
```

www.TopSage.com

TopSage.com

```
                                    cos,
                                    false);

            var pos1F:Object = rotate(pos1.x,
                                    pos1.y,
                                    sin,
                                    cos,
                                    false);

            // 将位置调整到实际的屏幕位置。
            ball1.x = ball0.x + pos1F.x;
            ball1.y = ball0.y + pos1F.y;
            ball0.x = ball0.x + pos0F.x;
            ball0.y = ball0.y + pos0F.y;
            // 将速度旋转回去。
            var vel0F:Object = rotate(vel0.x,
                                    vel0.y,
                                    sin,
                                    cos,
                                    false);
            var vel1F:Object = rotate(vel1.x,
                                    vel1.y,
                                    sin,
                                    cos,
                                    false);
            ball0.vx = vel0F.x;
            ball0.vy = vel0F.y;
            ball1.vx = vel1F.x;
            ball1.vy = vel1F.y;
        }
    }

    private function rotate(x:Number,
                            y:Number,
                            sin:Number,
                            cos:Number,
                            reverse:Boolean):Point
    {
        var result:Point = new Point();
        if(reverse)
        {
            result.x = x * cos + y * sin;
            result.y = y * cos - x * sin;
        }
        else
        {
            result.x = x * cos - y * sin;
            result.y = y * cos + x * sin;
        }
        return result;
    }
  }
}
```

297

www.TopSage.com

这里，我创建了一个旋转函数并且返回了一个flash.geom.Point类的实例。这是带有预定义的x和y属性（还有很多其他的内容我们这里不使用）的一个对象。返回的点的x和y属性被旋转。当你正在学习这里面所包含的原理时，这个版本并不是很容易读的，而它的代码是很少有重复的。

3. 加入多个物体

现在，让两个精灵相互碰撞和相互作用并不是简单的任务，但你已经做到了。恭喜。现在让
298 我们使用更多一些的碰撞物体——假设有8个精灵。看起来好像复杂性是之前的4倍，但实际上并不是那样。现在你的函数只是一次检测两个小球，但不管怎样，这就是你真正想做的事情。放置更多的物体在舞台上，移动它们，然后检测每一个物体相对于其他所有物体的情况。实际上你已经在第9章的碰撞检测示例中做过这样的事情。所有你要做的只是在需要碰撞检测的地方插入checkCollision函数。

对于这个示例（MultiBilliard.as），从一个数组中的8个小球开始，循环它们，并为它们指定各种属性：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Point;

    public class MultiBilliard extends Sprite
    {
        private var balls:Array;
        private var numBalls:uint = 8;
        private var bounce:Number = -1.0;

        public function MultiBilliard()
        {
            init();
        }

        private function init():void
        {
            balls = new Array();
            for(var i:uint = 0; i < numBalls; i++)
            {
                var radius:Number = Math.random() * 20 + 20;
                var ball: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;
                addChild(ball);
                balls.push(ball);
            }

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
```

www.TopSage.com

```
        }

        private function onEnterFrame(event:Event):void
        {
            // 即将发生的……
        }
        // checkWalls、checkCollision和rotate方法没有列出来，但是它们和前边例子中的是一样的。
    }
}
```

299

你可能注意到我让每个小球的初始的x和y位置都拉开了距离，以确保它们在开始时不会相互接触。否则，它们就会粘连在一起。

onEnterFrame方法相当简单。它只有两个循环：一个是用于基本的移动，另一个是用于碰撞检测。

```
private function onEnterFrame(event:Event):void
{
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball = balls[i];
        ball.x += ball.vx;
        ball.y += ball.vy;
        checkWalls(ball);
    }
    for(i = 0; i < numBalls - 1; i++)
    {
        var ballA:Ball = balls[i];
        for(var j:Number = i + 1; j < numBalls; j++)
        {
            var ballB:Ball = balls[j];
            checkCollision(ballA, ballB);
        }
    }
}
```

第1个循环只是遍历舞台上的每个小球，移动它并在碰到墙时回弹。然后你有一个嵌套循环让每一个球与其他每个小球进行比较，也就是我在讨论碰撞检测（第9章）时所涉及的内容。在这里，你每次获取两个小球的引用，名为ballA和ballB，然后将它们传递给checkCollision函数。并且，非常令人惊奇的是，这就是要传给它的全部了。CheckWalls、checkCollision和rotate都是一样的，因此这里就不再重复了。

要加入更多的小球，只需要更新numBalls的变量值，并且保证它们开始的位置不相接触。

4．解决潜在的问题

我在描述本章的设置时说过一句警告的话：对于一对物体仍有可能出现粘连的问题。这种情况多数会出现在有许多精灵相互回弹的拥挤环境中，当它们以很高的速率移动时，情况好像会更糟。如果两个或3个精灵在舞台的一个角上碰撞时，你偶尔也会看到这种现象。

300

假设你有3个小球在舞台上——ball0、ball1和ball2——并且它们恰好靠在一起。这里是基本要发生的情况：

www.TopSage.com

- 代码根据3个小球的速度来移动它们。
- 代码检测ball0和ball1，然后是ball0和ball2。它发现没有碰撞发生。
- 代码检测ball1和ball2。这两者发生了碰撞，因此要计算它们新的速度并更新它们的位置以使它们不再接触。这里不经意间将ball1与ball0联系了起来。然而，这个特定的组合已经被检测过了，因此现在它会被忽略掉。
- 在下一次循环中，代码再次根据它的速度移动每一个小球。ball0和ball1相互粘连在一起的可能性被进一步加深了。
- 现在代码注意到了ball0和ball1有碰撞。它计算出它们的新的速度并加入新的速度到它们的位置上，把它们移开。但是，因为它们已经接触了，已经没有办法真正地将它们分开了。它们粘连在一起了。

另外，当你有许多物体在一个比较小的空间中并且以较高的速率移动时，也是非常容易发生粘连的。如果物体在影片开始时已经互相接触了也是很容易发生粘连的。正如你所看到的，这种问题会不时地出现，最好是知道问题出在哪里。问题的要点在checkCollision函数中，特别是下面的代码行：

```
// 更新位置。
pos0.x += vel0.x;
pos1.x += vel1.x;
```

这些代码只假设碰撞只是由于两个小球自身的速度而发生的，然后加入回新的速度将把它们分离开。多数时候，这是正确的。但我刚才所描述的情况是例外的。如果你处在例外的情形之下，你还需要一些更加严格的东西来确保精灵在移动之前是肯定分离开的。我提出了下面的方法：

```
// 更新位置。
var absV:Number = Math.abs(vel0.x) + Math.abs(vel1.x);
var overlap:Number = (ball0.radius + ball1.radius)
                     - Math.abs(pos0.x - pos1.x);
pos0.x += vel0.x / absV * overlap;
pos1.x += vel1.x / absV * overlap;
```

这完全是我自己创造出来的，因此我不能确定它有多准确，但它看起来工作得很好。首先检测绝对速度（我自己造的术语）。它是两个速度绝对值的和。例如，如果一个速度是-5而另一个是10，绝对值则为5和10，那么总数为5+10，或15。

接下来，它检测这些小球真正重叠了多少。它是通过得到它们的总的半径然后减去它们之间的距离来完成的。

然后根据它们绝对速度的百分比移动每一个小球的重叠部分。结果是这些小球将准确地相互接触，没有重叠。它要比之前的版本复杂一些，但它确实可以清除那些bug。

301

实际上，在下一个版本MultiBilliard2.as中，我创建了20个小球，让他们变得大一些，随机摆放在屏幕上。发生重叠的小球在某些帧上仍然会有强烈反应，但由于使用了新的代码，它们最终都会平静下来。

www.TopSage.com

当然，你可以继续研究自己的解决方案，如果得出了更简单、更有效和更准确的方案，请不要忘记分享!

11.4 本章重点公式

本章的重点公式是关于动量守恒的公式。

动量守恒，直接的数学方式：

$$v0Final = \frac{(m0 - m1) * v0 + 2 * m1 * v1}{m0 + m1}$$

$$v1Final = \frac{(m1 - m0) * v1 + 2 * m0 * v0}{m0 + m1}$$

Actionscript中的动量守恒，简化方式：

```
var vxTotal:Number = 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!你已经通过了本书中可能最难的数学问题。你已经掌握了控制准确的碰撞反应的方法。这也是人们经常写信问我的问题之一。趁着你现在有兴趣，可以制作一个相当不错的台球游戏，并且是完全符合台球物理的。为了保持简单，在这些示例中我忽略的一件事是摩擦力概念。你可能想试着将它们加到系统中。你现在已经有足够的能力来做这件事。你也许想先看一下第19章，那里我给出了处理两个相同质量的物体时使用的一些技巧。

在下一章中，我将把事情缓和一下开始探讨粒子引力，虽然例子中加上了一些台球物理现象，但它看上去是很合适的。

www.TopSage.com

大家网
TopSage.com

第12章 粒子引力和重力

本章内容

- 粒子
- 重力
- 弹性
- 本章重点公式

我对这些章的进展很满意，每一章增加了一个通常都是扩展交互方面的内容的新的概念。开始，你只是让物体移动。然后物体开始与周围的环境进行交互，接下来是用户，再接下来是碰撞。在这一章中，我将继续扩展讲解物体之间的交互，特别是从距离上。具体来说，我将讲解粒子、重力（这次有点不同）、弹性（再次讲），还有世界著名的结点域（Node Garden）。现在就让我们开始吧!

12.1 粒子

首先应该定义一下我所指的粒子是什么。对于我来说，就本章而言，一个粒子（particle）是指一个单元，通常是几个或多个构成的群体中的一个个体。因此，一个粒子可以说是一粒灰尘、一个球或一个行星。

粒子通常会共享一种行为，但也会有它们自己的特性。面向对象编程的爱好者将会看到这里对对象的比喻。一个粒子类的所有对象通常会共享相同的行为，该行为由类的方法来定义，单独的实例可以通过指定不同的属性值来自定义。你已经看到了你所用过的Ball类实例是如何工作的。每一个实例都有它自己的属性：速度、质量、大小、颜色等等，但是所有的小球都是按相同的规则在移动。

你已经有了Ball类，它几乎具有你在这里需要的所有功能。在这种情况下，像以前的示例一样，粒子实例只是保存着属性。文档类将负责移动每一个粒子和控制所有的反应。另一种策略是将控制行为的代码移到粒子类中。那么每一个粒子将有它自己的enterFrame处理器或动画时间，并且负责自身的移动和控制所有的反应。两种方法中的任何一种都可工作，而且两者都有它们的优缺点。在本书的示例中，为了简单，你仍将保持在文档类中进行动画。

www.TopSage.com

大致的设置与之前的示例是一样的。比较大的变化是粒子之间的相互作用和引力，它们会在onEnterFrame方法中出现。基本的设置是要创建一些粒子然后随机将它们散布在屏幕上。这里是代码：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class ClassName extends Sprite
    {
        private var particles:Array;
        private var numParticles:uint = 30;

        public function ClassName()
        {
            init();
        }
        private function init():void
        {
            particles = new Array();
            for(var i:uint = 0; i < numParticles; i++)
            {
                var particle:Ball = new Ball(5);
                particle.x = Math.random() * stage.stageWidth;
                particle.y = Math.random() * stage.stageHeight;
                particle.mass = 1;
                addChild(particle);
                particles.push(particle);
            }

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {

        }
    }
}
```

306

这里，你初始化了每个粒子的radius是5并且设置质量为1。你可能想试着改变这些值来得到不同的效果。你也可能想让粒子在开始时有随机的速度或随机的大小。

在本章中的剩下的示例中都假定你做好了上面代码的设置并且偶尔会向其中添加一些内容。但我将主要讨论onEnterFrame方法和其他需要写的函数。首先，我将描述一些理论背景，你刚刚创建的这个文件暂时放在这里，因为很快就要回到这个文件上。

12.2 重力

我将要谈到的第1种粒子引力是重力。现在，我可以听到你在说，“你不是在书中的第2部分讲过了吗？”是的，我讲过了，但那是从非常近的角度看的重力。

www.TopSage.com

站在地球上，重力有一个相当简单的描述：它把物体拉下来。实际上，它以一个非常特殊的速率将物体拉下来。它施加的加速度相当于32英尺/秒^2。表述加速度的一种方法是通过计算在一个指定的时间段内增加了多少速度。重力使物体的移动每秒增加32英尺，每一秒都在拉动物体。你可以到高山或峡谷中，而32这个数值以你的辨别力（在没有精密设备的情况下）来说
307 它是不变的。

12.2.1 引力

当你开始后退时，你会发现，实际上，你离一个行星或一个很大的物体越远，引力就会变得越小。这是非常好的事情。否则，我们都将被吸到太阳里，太阳又将被吸到太阳系和行星中，而这些将很快撞到一起。因此，当你到达可以将行星称之为粒子的某个位置点时，它们之间的距离很大程度上影响着引力。

距离对力的影响有多大是很容易描述的。它与距离的平方成反比。好，也许还需要一些解释。首先，引力也是与质量紧密相连的。某物体质量越大，它作用在其他物体上的拉力越大，同时它所受到的拉力也越大。也有一个被称为*万有引力常量*（gravitational constant）（简写为G）的东西比较适合用在这里。下面是关于引力的完整等式：

$$force = G * m1 * m2 / distance^2$$

通俗地讲，一个物体和另一个物体之间的引力等于万有引力常量乘以两者的质量，再除以两个物体之间距离的平方。嗯，这看起来很简单。你只需要知道万有引力常量是多少，其他的就可以全都得到。好，这里是它的正式定义：

$$G = (6.6742 \pm 0.0010) * 10^{-11} * m^3 * kg^{-1} * s^{-2}$$

现在，你要解决如何在ActionScript中编码。对于我来说，当看到这样的公式，立刻就知道我会很快就可以做到。你想知道我是怎么做到的吗？我这样写了引力公式：

$$force = m1 * m2 / distance^2$$

没错，我忽略了G。现在我又使用了可以放弃的权利。如果你现在正在为NASA制作一个基于Flash的太空导航系统，应当把G保留在那里。但如果你要制作一个在线的太空战斗的游戏，那么可以不需要它。

现在已经有了一个公式，让我们将它加入到代码中。先设置onEnterFrame函数，并且让它调用一个gravitate函数以使你可以把处理重力的函数区分开：

```
private function onEnterFrame(event:Event):void
{
    for(var i:uint = 0; i < numParticles; i++)
    {
        var particle:Ball = particles[i];
        particle.x += particle.vx;
        particle.y += particle.vy;
    }
    for(i=0; i < numParticles - 1; i++)
    {
```

www.TopSage.com

```
        {
            var partA:Ball = particles[i];
            for(var j:uint = i + 1; j < numParticles; j++)
            {
                var partB:Ball = particles[j];
                gravitate(partA, partB);
            }
        }
    }
```

308

这里，你只是在一个for循环中使用基本的运动代码移动所有的粒子，然后再使用嵌套循环获得它们之间的相互作用。一旦你得到了partA和partB，将它们传递给grativate函数。下面是这个函数：

```
private function gravitate(partA:Ball, partB:Ball):void
{
    var dx:Number = partB.x - partA.x;
    var dy:Number = partB.y - partA.y;
    var distSQ:Number = dx * dx + dy * dy;
    var dist:Number = Math.sqrt(distSQ);
    var force:Number = partA.mass * partB.mass / distSQ;
    var ax:Number = force * dx / dist;
    var ay:Number = force * dy / dist;
    partA.vx += ax / partA.mass;
    partA.vy += ay / partA.mass;
    partB.vx -= ax / partB.mass;
    partB.vy -= ay / partB.mass;
}
```

首先，你找到两个粒子之间的dx和dy以及总的距离。记住引力公式——force=G×m1×m2/distance^2——使用的是距离的平方。正常情况下，我使用dist= Math.sqrt(dx*dx +dy*dy)一次计算出距离。但是接下来要得到距离的平方，我需要再重新平方已得到的平方根，这将是重复的工作。如果我在开平方之前只是使用变量distSQ来保存dx*dx+dy*dy，这样就可以节省一些计算。

接下来我通过将质量相乘再除以距离的平方得到总的力。那么我就可以计算出在*x*和*y*轴上的总的加速度。另外，我使用的是第9章最后讨论的简化公式，使用dx/dist取代Math.cos(angle)以及使用dy/dist取代Math.sin(angel)。这样就省掉了使用Math.atan2(dy/dx)来计算角度。

现在，注意我一直在说的总的力和总的加速度。这是作用在两个物体之间的合力。你需要基于它们的质量来分配这个合力。如果你把物体想成是地球和太阳，那么在它们之间有一个特殊的力。这个力是它们质量的乘积除以它们之间距离的平方。因此，它们都在通过总的合力相互拉对方。地球被拉向太阳，同时太阳也被拉向地球。很明显，地球获得了更大的加速度，因为它的质量要远小于太阳。因此，要得到系统中每个物体单独的加速度，需要使用总的加速度除以此物体的质量。因此，得到了最后4行公式。注意partA是加上加速度，而partB是减去加速度。这仅仅取决于我过去得到dx和dy的减法的顺序。

309

这个示例的最终代码可以在Gravity.as中找到。继续测试它。可以看到粒子开始是静止的，但是彼此慢慢地相互吸引，如你在图12-1所看到的。偶尔，有些粒子会相互围绕着运动。但多数

www.TopSage.com

情况下发生的是它们碰撞后飞向相反的方向。

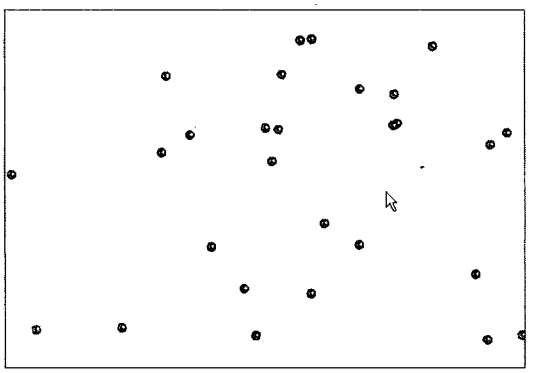


图12-1　我们有粒子了

这种快速飞出是一个bug吗？可以说是也可以说不是。在代码中它不是一个bug。它确实是预料之中的行为，即所谓的*弹弓效应*（slingshot effect）。NASA用它来将控测器发射到太空深处。当一个物体与一个行星离得越来越近的时候，它就会有越来越大的加速度，并且开始以一个非常高的速度前进。如果你恰好可以对准它，然后它慢慢地接近行星，但刚好又错过了它，那么这个物体将快速地消失在太空中，而行星的引力是无法捕获到它的。

但是回到Flash中，这里发生的是物体彼此离得很近很近——几乎是零。因此，两者之间的力变得很大，几乎是无穷大。因此，从数学角度来看，它是正确的，但是从一个模拟的角度看，它是不现实的。如果两个物体离得足够近，那么应当发生的是它们产生了碰撞。如果你直接将太空探测器正对行星，那么它不会以无穷大的速度离行星而去，它会留下一下很大的陨石坑。

12.2.2　碰撞检测和反应

那么，对于你的粒子，你需要有某种碰撞检测和反应。怎样做由你来决定，你可以让它们爆炸然后消失。你可让一个粒子消失然后将它的质量加到另一个粒子上。

对于这个例子，我意识到我已经有了一些在之前的章节中写过的非常漂亮的碰撞和反应代码，这些代码在函数checkCollision中。让我们把它插入在这里，像这样：

310

```
private function onEnterFrame(event:Event):void
{
    for(var i:uint = 0; i < numParticles; i++)
    {
        var particle:Ball = particles[i];
        particle.x += particle.vx;
        particle.y += particle.vy;
    }
    for(i=0; i < numParticles - 1; i++)
    {
        var partA:Ball = particles[i];
        for(var j:uint = i + 1; j < numParticles; j++)
        {
```

www.TopSage.com

TopSage.com

```
                var partB:Ball = particles[j];
                checkCollision(partA, partB);
                gravitate(partA, partB);
            }
        }
    }
```

只有那行粗体代码是改过的。当然，我还复制并粘贴了checkCollision和rotate函数到这个文件中。（不要忘记导入flash.geom.Point类，因为这些函数需要用到它。）这些代码可以在GravityBounce.as中找到。

现在粒子之间在相互吸引，但是当它们碰撞后就互相弹开。试着改变粒子的质量来看它们的相互吸引有何不同。你甚至可以做些奇怪的事情，例如让粒子的质量为负值来观察它们之间的互相排斥效果。

在文件GravityRandom.as中，我保持了原来的文件内容，只是在init方法中增加了一行并修改了两个地方：

```
private function init():void
{
    particles = new Array();
    for(var i:uint = 0; i < numParticles; i++)
    {
        var size:Number = Math.random() * 25 + 5;
        var particle:Ball = new Ball(size);
        particle.x = Math.random() * stage.stageWidth;
        particle.y = Math.random() * stage.stageHeight;
        particle.mass = size;
        addChild(particle);
        particles.push(particle);
    }

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

这里只是给了每个粒子随机的大小和基于大小的质量，如图12-2所示。看起来事情变得有趣了。

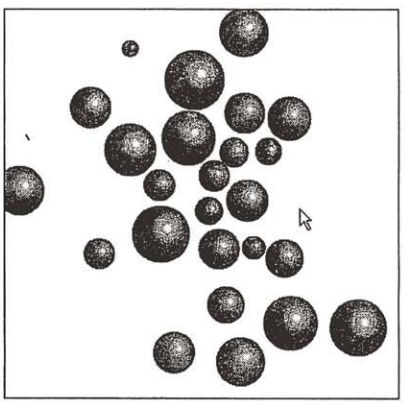


图12-2 碰撞中的行星

12.2.3 沿轨道运行

最后，只是向你展示如何变得更真实一些。让我们来设置一个简单的天体系统，一个太阳和一个行星。创建一个质量为10 000的大太阳和一个质量为1的行星。然后移动行星离开太阳一段距离并给它一些与两者连线垂直的速度。设置效果如图12-3所示。

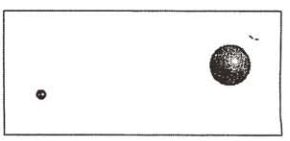


图12-3 设置舞台

如果得到了质量、距离和速度，就可以使行星处于轨道中。它需要反复试验，但是我在Orbit.as得出了它。这里我给出的只是init方法的代码。在文件中唯一的其他修改是将numParticles变量设置为2。

```
private function init():void
{
    particles = new Array();
```

www.TopSage.com

```
        var sun:Ball = new Ball(100, 0xffff00);
        sun.x = stage.stageWidth / 2;
        sun.y = stage.stageHeight / 2;
        sun.mass = 10000;
        addChild(sun);
        particles.push(sun);

        var planet:Ball = new Ball(10, 0x00ff00);
        planet.x = stage.stageWidth / 2 + 200;
        planet.y = stage.stageHeight / 2;
        planet.vy = 7;
        planet.mass = 1;
        addChild(planet);
        particles.push(planet);

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
```

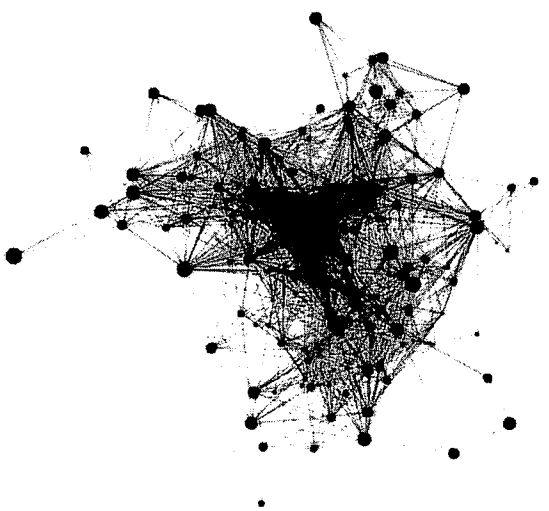
311
~
312

作为一个额外的例子，查看OrbitDraw.as文件，它绘制了跟踪轨道的线条。你可以得到一些有趣的线条图案。

12.3　弹性

可能你想尝试的另一类粒子引力是弹性。是的，弹性，我最喜欢的话题！回顾第8章，你试验了弹性链和物体之间的弹性。这里，将看到一个更大的应用，在该应用中你有许多粒子，彼此之间存在着弹性，如同之前刚看到的引力的示例。

这个例子的灵感来自Jared Tarbell在www.levitated.net做的称为结点域的实验，如图12-4所示。思路是你需要有这些结点（粒子）的一个域，而且每个结点与其他任何邻近的结点之间有着不同类型的交互。这就是我让其中的一个反应为弹性的思想。



313

图12-4　Jared Tarbell的结点域

www.TopSage.com

12.3.1 引力与弹性

如果你看过引力和弹性，你可以发现它们很相似，而且几乎是相反的。两者都是应用加速度到两个物体上来互相拉动。但是在引力中，物体越远，加速度越小。在弹性中，当距离增大时加速度会变得更大。

因此，在之前的示例中你只需要换掉引力的代码，然后将弹性的代码插入其中，效果可能并不是很有意思，因为粒子最后只是集成了一堆。弹性不允许它们之间有距离。引力是“眼不见心不想”，弹性是“距离产生美”。

因此你进入了一个进退两难的境地。你希望粒子使用弹力相互吸引，但是又想让它们保持一定的距离而不至于都聚在一起。我的解决方案是提出一个最小距离变量。从数学角度讲，它可能更像是一个最大距离，因为它是使一个反应仍有效的最大距离。但是我总是认为它是最小的距离，因为它意味着粒子至少这么近时才会产生反应。如果它们离得太远，它们就会忽略彼此。

12.3.2 弹性结点域

让我们开始制作自己的弹性结点域。为了使它看起来与本书原来版本的示例尽可能相近，我将影片的背景改成黑色并且让粒子变为白色。它可能并不是很好，但由于没有使用复杂的渐变填空代码或嵌入外部图形，它只能这样。我将漂亮的图像效果留给你。

下面讲解代码。你可以在NodeGarden.as文件中找到它。但是让我们一步一步来了解。首先，设置粒子的一些变量，包括粒子数量、我刚才说到的最小距离和弹性值。

```
var numParticles:uint = 30;
var minDist:Number = 100;
var springAmount:Number = .001;
```

在之前第8章的弹性示例中，你使用了0.2作为弹性值。这里你需要使用更小的值，因为有大量的粒子产生交互。如果你设置得太高，速度将会变得很快。然而，如果你设置得太低，粒子只是慢慢地在屏幕上移动，好像并不会互相觉察到。

然后进行初始化：

```
private function init():void
{
    particles = new Array();
    for(var i:uint = 0; i < numParticles; i++)
    {
        var particle:Ball = new Ball(5, 0xffffff);
        particle.x = Math.random() * stage.stageWidth;
        particle.y = Math.random() * stage.stageHeight;
        particle.vx = Math.random() * 6 - 3;
        particle.vy = Math.random() * 6 - 3;
        addChild(particle);
        particles.push(particle);
    }

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

314

www.TopSage.com

这只是创建很多粒子，将它们扔在屏幕上，然后给它们随机的速度。注意在这个例子中我去掉了质量。后面，在12.3.4节中，我将向你展示将质量加进来的实验。

接下来是onEnterFrame方法：

```
private function onEnterFrame(event:Event):void
{
    for(var i:uint = 0; i < numParticles; i++)
    {
        var particle:Ball = particles[i];
        particle.x += particle.vx;
        particle.y += particle.vy;
        if(particle.x > stage.stageWidth)
        {
            particle.x = 0;
        }
        else if(particle.x < 0)
        {
            particle.x = stage.stageWidth;
        }
        if(particle.y > stage.stageHeight)
        {
            particle.y = 0;
        }
        else if(particle.y < 0)
        {
            particle.y = stage.stageHeight;
        }
    }

    for(i=0; i < numParticles - 1; i++)
    {
        var partA:Ball = particles[i];
        for(var j:uint = i + 1; j < numParticles; j++)
        {
            var partB:Ball = particles[j];
            spring(partA, partB);
        }
    }
}
```

315

这些代码看上去非常熟悉。它与之前的示例非常相似。我只是加入了屏幕折回并且调用了spring函数，而没有调用gravitate函数。

现在看一下它的核心部分，spring函数本身：

```
private function spring(partA:Ball, partB:Ball):void
{
    var dx:Number = partB.x - partA.x;
    var dy:Number = partB.y - partA.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    if(dist < minDist)
    {
        var ax:Number = dx * springAmount;
        var ay:Number = dy * springAmount;
```

www.TopSage.com

```
            partA.vx += ax;
            partA.vy += ay;
            partB.vx -= ax;
            partB.vy -= ay;
        }
    }
```

首先，计算出两个粒子之间的距离。如果它不小于minDist，继续运行其他代码。然而如果小于minDist，根据距离和弹性值计算出每一个轴上的加速度。然后将加速度加到partA的速度上并从partB的速度上减去加速度。这可以使粒子相互靠近。

继续测试它。你将看到如图12-5所示的结果。注意这些粒子是如何像苍蝇一样聚在一起。但这个群体在移动，分散开，然后又加入其他群体等等。很有意思的现象。试着改变minDist和springAmount值来观察发生的变化。

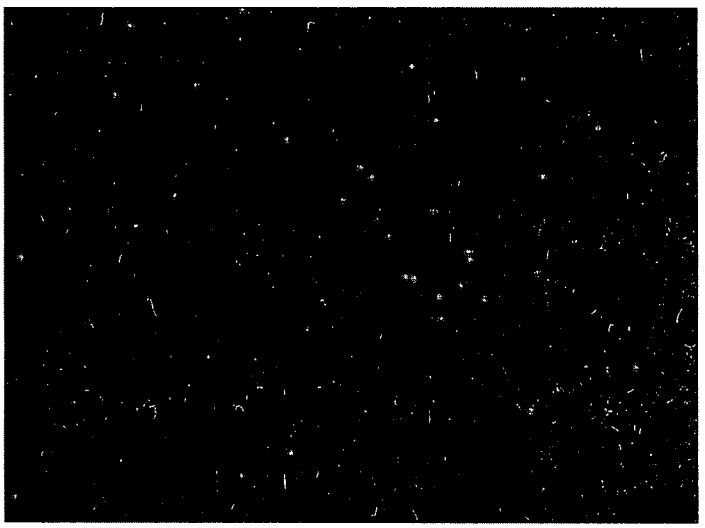


图12-5 运动的结点

316

12.3.3 结点的连接

这里很明显在结点之间发生了什么，但是我想真正地标示出每一对结点之间的具体交互。最好的方法是在两个结点之间绘制线条。这是很简单的。只需稍微修改一下spring函数：

```
private function spring(partA:Ball, partB:Ball):void
{
    var dx:Number = partB.x - partA.x;
    var dy:Number = partB.y - partA.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    if(dist < minDist)
    {
        graphics.lineStyle(1, 0xffffff);
        graphics.moveTo(partA.x, partA.y);
        graphics.lineTo(partB.x, partB.y);
```

www.TopSage.com

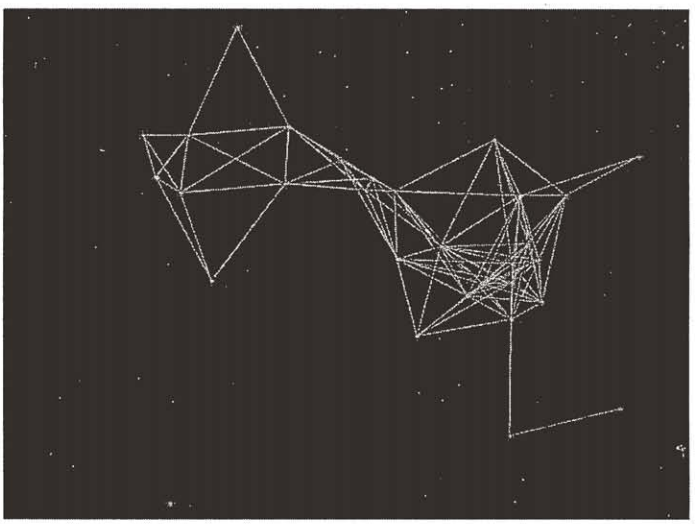
大家网 TopSage.com

```
            var ax:Number = dx * springAmount;
            var ay:Number = dy * springAmount;
            partA.vx += ax;
            partA.vy += ay;
            partB.vx -= ax;
            partB.vy -= ay;
        }
    }
```

如果两个结点正在相互作用，这个函数将设置一个线条样式，然后在两者之间绘制一个线条。你也需要在onEnterFrame方法的第1行加入下面的代码，以使在每一帧的时候刷新屏幕：

```
graphics.clear();
```

因此，结点被连接起来了，如图12-6所示。这还可以，但是我不喜欢现在的结点连接效果。



317

图12-6　连接结点

我需要更多的渐变处理。我想象的是如果两个结点刚好在minDist范围内时，线条将完全透明。当它们越来越接近时，线条会变得越来越亮。因此，如dist/minDist，它给了我们一个可用于alpha的从0到1的比值。但是它是反向的，因为如果dist等于minDist，alpha将是1，当dist接近为0时，alpha将是0。因此，使用1来减去这个比值，这样就有效地颠倒了效果。这里是最终的线条绘制代码：

```
graphics.lineStyle(1, 0xffffff, 1 - dist / minDist);
graphics.moveTo(partA.x, partA.y);
graphics.lineTo(partB.x, partB.y);
```

这对我来说是非常漂亮的效果，如图12-7所示。最终结果代码在NodeGardenLines.as中。

www.TopSage.com

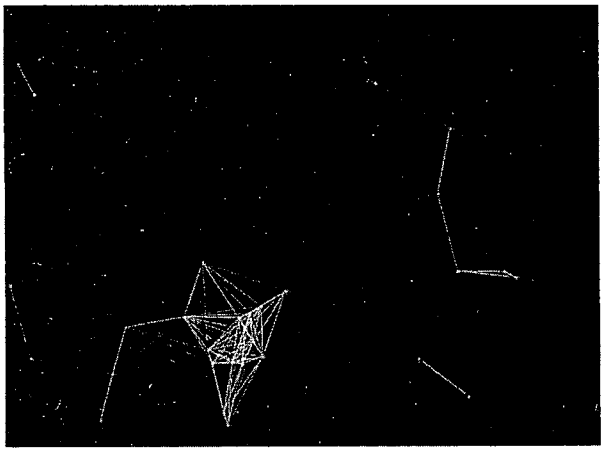


图12-7 很小的改变，但是有很大的不同

12.3.4 有质量的结点

在写这章时，我自己提出了有质量的结点的想法，在之前从来没有想过这个问题。因此，我试着去做它并完成了NodeMass.as文件。在里面，我为每一个结点分配了随机的缩放和基于缩放的质量。

```
private function init():void
{
    particles = new Array();
    for(var i:uint = 0; i < numParticles; i++)
    {
        var size:Number = Math.random() * 10 + 2;
        var particle:Ball = new Ball(size, 0xffffff);
        particle.x = Math.random() * stage.stageWidth;
        particle.y = Math.random() * stage.stageHeight;
        particle.vx = Math.random() * 6 - 3;
        particle.vy = Math.random() * 6 - 3;
        particle.mass = size;
        addChild(particle);
        particles.push(particle);
    }
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

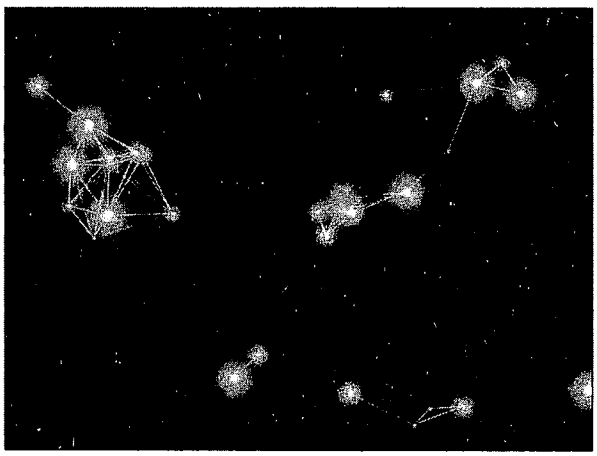
这些代码只在我将速度加入到spring函数中时使用。我通过每个粒子的质量来分配速度。这样就使较大的结点有了较大的惯性。

```
private function spring(partA:Ball, partB:Ball):void
{
    var dx:Number = partB.x - partA.x;
    var dy:Number = partB.y - partA.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    if(dist < minDist)
    {
```

www.TopSage.com

```
            graphics.lineStyle(1, 0xffffff, 1 - dist / minDist);
            graphics.moveTo(partA.x, partA.y);
            graphics.lineTo(partB.x, partB.y);
            var ax:Number = dx * springAmount;
            var ay:Number = dy * springAmount;
            partA.vx += ax / partA.mass;
            partA.vy += ay / partA.mass;
            partB.vx -= ax / partB.mass;
            partB.vy -= ay / partB.mass;
        }
    }
```

因为我正在削减弹性的整体效果，所以我将springAmount值增加到了0.002 5。我喜欢这种整体的效果，如图12-8所示。



319

图12-8　更多的效果

那么，你可以将结点用在哪儿呢？个人来说，它们看上去很酷，并且我曾经使用它们制作了漂亮的屏幕保护程序。但是对于初学者来说，你可以用它们来创建各种游戏场景。将一艘小行星型的飞船放在那儿，试着让它躲避那些结点。我打赌这将是一个很好的挑战！

12.4　本章重点公式

很明显，这里最重要的公式是引力公式。

基本的引力：

$$force = G * m1 * m2 / distance^2$$

Actionscript中的引力实现：

```
function gravitate(partA:Ball, partB:Ball):void
{
      var dx:Number = partB.x - partA.x;
      var dy:Number = partB.y - partA.y;
      var distSQ:Number = dx * dx + dy * dy;
```

www.TopSage.com

```
        var dist:Number = Math.sqrt(distSQ);
        var force:Number = partA.mass * partB.mass / distSQ;
        var ax:Number = force * dx / dist;
        var ay:Number = force * dy / dist;
        partA.vx += ax / partA.mass;
        partA.vy += ay / partA.mass;
        partB.vx -= ax / partB.mass;
        partB.vy -= ay / partB.mass;
    }
```

注意在函数中，我把Ball类保留在那儿了，但实际上你可以使用任何自定义的类，只要它能存储速度值、质量和位置即可。

12.5 小结

这章涉及了处于某个距离的粒子之间的交互和如何使用引力和弹性来制作有趣的效果。因此，你有了两种新的方法来制作多个物体的动态运动效果。

在下面的两章中，我将讲解非常新的课题：正向运动和反向运动。这些技术可以让你制作类似机器人的手臂和行走等很酷的效果。 320

www.TopSage.com

第13章

正向运动：行走

本章内容

- 介绍正向和反向运动
- 开始正向运动编程
- 自动运行
- 使它真正地行走

到目前为止，你一直沿着一条特别的路前进，并且进行得很顺利。实际上，前面的章节只涉及了交互的ActionScript动画的所有基础，也可以说它们是非常高级的“基础”。现在，我将岔开话题来研究使用运动学的一些有趣的技术。

那么，到底什么是运动学（kinematics）？当我尽力去查找关于它的内容时，它看起来总让人有些提心吊胆。我知道它可能用于一些高级的3D动画程序，并且如果你是在线观看它们，你会发现一些页面上呈现给你的是许多有着奇怪符号的等式，它使得你之前在本书中所做的任何东西看起来就像是很初级的算术。那么，首先，我想说它并不是真地这么糟糕。之前的章节中所涉及的只是你所需要的所有概念。你只需要将它们用正确的方法组合起来。

运动学基本上是数学的一个分支，用来处理与力和质量无关的物体运动。因此，它只是速率、方向和速度。看起来很简单，是吗？好，这是个非常简单的定义，并且我确定它包含一些复杂的东西，但就我们而言，这基本上就够了。

当人们在计算机科学、图形学、游戏等领域中开始谈论运动学时，主要谈论运动学的两个特殊的分支：正向运动和反向运动。让我们从这里开始。

13.1 介绍正向和反向运动

正向和反向运动通常与几个部分连接在一起构成的系统有关，例如一个链子或一个连接的手臂。它们与系统的移动方式和每一部分相对于其他部分和整个系统的移动方式有关。

通常，一个运动系统有两个端点：基础端和自由端。一个连接的手臂通常一端连到某个固定物体上，而另一端可以四处移动以够到或抓取东西。一个链子的一端或两端可能会与某物体固定，或者是两端都不固定。

www.TopSage.com

TopSage.com

正向运动（forward kinematics，FK）处理的是在系统基础端发起的运动，然后传递给自由端。*反向运动*（inverse kinematics，IK）处理的是相反的：运动源于或取决于自由端然后再传递回到基础端（如果有基础端的话）。

有一些例子很合适。在多数情况下，在行走循环中身体的肢体使用的是正向运动。大腿移动，它传递给小腿。小腿移动，又传递给脚。然后脚开始移动。在这种情况下，脚不能决定任何事情。它最终在哪儿，完全基于它前面的所有肢体的位置。

一个反向运动的例子是用手拉住某个人。这里，力作用于自由端——手——控制它的位置和移动，然后传递到前臂，上臂，最后到整个身体。

另外一个更为微妙的反相运动的例子是手臂去拿某个东西。手是用来驱动整个系统的。当然，你是可以这样说：在这个例子中，上臂和前臂在移动，并且它们控制着手的位置。这是正确的，但一个直接的意图就是把手放在一个指定的地方。这就是驱动力。在这种情况下，它不是一个物理上的力，只是一个意图。前臂和上臂只是根据手的位置需要来安排它们自己的位置。 324

当你看完了这章和下一章的例子后，两者的区别就会越来越清晰。但是现在，记住拖动和伸手拿通常是反向运动，而一个反复的运动，如行走，通常为正向运动，该运动是本章的主题。

13.2 开始正向运动编程

编写正向运动和反向运动的程序包括下面几个基本要素：

- 系统的各个部分。我称之为关节。
- 每一个关节的位置。
- 每一个关节的旋转。

在这些例子中的每一个关节都是像前臂或上臂或腿的任何一部分那样的椭圆形的形状。当然，最后一个关节可以是其他形状，如一只手、一只脚、一个夹子、一个钳子或一个向入侵者发射绿色激光束的设备。

在每一个关节的一端都有一个轴心点，它可以围绕轴心点旋转。如果这个关节有许多子关节，那么子关节的轴心点位于此关节的相对的一端。就像你的上臂的轴心点在你的肩上，你的前臂的轴心点在你的肘部，而你的手的轴心点在你的手腕上。

当然，在许多现实的系统中，那些轴心点会有许多方向。想象一下你可以有多少种方式来活动你的手腕。到本书结束时，你可能想亲自在Flash中做些像那样的例子，但现在，系统将是严格的二维系统。

13.2.1 移动一个关节

让我们从一个关节开始并且让它以某种方式移动。首先，你需某个东西来作为关节使用。如果你猜到它可能是扩展了Sprite类的一个自定义的类，那么你是对的。一个精灵可以有图形，它可以有位置，可以被旋转，并且可以被加到显示列表中。我们将使用它。下面是我完成的Segment.as类：

www.TopSage.com

```
package
{
    import flash.display.Sprite;
    import flash.geom.Point;

    public class Segment extends Sprite
    {
       private var color:uint;
       private var segmentWidth:Number;
       private var segmentHeight:Number;

       public var vx:Number = 0;
       public var vy:Number = 0;

       public function Segment(segmentWidth:Number,
                               segmentHeight:Number,
                               color:uint = 0xffffff)
        {
            this.segmentWidth = segmentWidth;
            this.segmentHeight = segmentHeight;
            this.color = color;
            init();
        }

        public function init():void
        {
            // 绘制关节本身。
            graphics.lineStyle(0);
            graphics.beginFill(color);
            graphics.drawRoundRect(-segmentHeight / 2,
                                   -segmentHeight / 2,
                                   segmentWidth + segmentHeight,
                                   segmentHeight,
                                   segmentHeight,
                                   segmentHeight);
            graphics.endFill();

            // 绘制两个“轴栓”。
            graphics.drawCircle(0, 0, 2);
            graphics.drawCircle(segmentWidth, 0, 2);
        }

        public function getPin():Point
        {
            var angle:Number = rotation * Math.PI / 180;
            var xPos:Number = x + Math.cos(angle) * segmentWidth;
            var yPos:Number = y + Math.sin(angle) * segmentWidth;
            return new Point(xPos, yPos);
        }
    }
}
```

325

这基本上就是设置一个宽、高和颜色然后绘制一个圆角矩形来用做关节。它也在关节（0,0）的注册点和端点上分别绘制了一个小圆。这是两个“轴栓”，是关节与其他关节之间相连的位置。

www.TopSage.com

（你可能也注意到了两个公开变量vx和vy。这些在13.4.4节还会用到。）

下面的代码段创建了几个不同宽和高的关节，来演示如何使用Segment类：

```
var segment:Segment = new Segment(100, 20);
addChild(segment);
segment.x = 100;
segment.y = 50;
var segment1:Segment = new Segment(200, 10);
addChild(segment1);
segment1.x = 100;
segment1.y = 80;
var segment2:Segment = new Segment(80, 40);
addChild(segment2);
segment2.x = 100;
segment2.y =120;
```

326

图13-1显示了这段代码所创建的不同的关节。

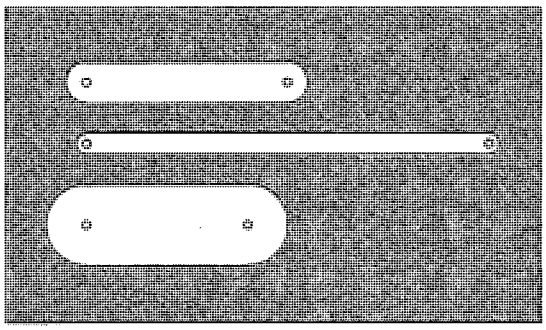


图13-1 一些关节示例

在图13-1中需要注意的一个重点地方是关节的宽度是指两个轴栓之间的距离。关节的实际宽度要超出两个轴栓之间的距离。你可以观察每个关节被放置在*x*位置为100时的状态。尽管它们的左边没有对齐，但它们左边的所有轴栓的位置是对齐的。当你旋转关节时，它们将会围绕左手边的轴栓旋转。

你可能也会注意到在Segment类中有一个公共的方法getPin()，它返回的是一个flash.geom.Point实例。这里它将返回右手边轴栓的*x*、*y*位置。很明显当关节旋转时它会改变。因此我们使用基本的三角学来计算出它的位置点。这也是用于连接下一个关节的位置点，在本章的下一节中你将会看到这方面的内容。

对于第1个示例SingleSegment.as，我已经创建了一个关节并把它放置在舞台上。我还使用了滑钮类（slider class），SimpleSlider.as，我已经创建好了SimpleSlider.as并加到了项目中。你可以从www.friendsofed.com的本书下载页面上下载它，并且你可以免费使用你想用的东西。它对于动态调节数值是非常有用的。对于这个滑钮，你可以在构造函数中设置minimum、maximum和value参数。你将看到我设置了minimum为-90，maximum为90，value为0。这里是结合滑钮类一起使用的类文件SingleSegment.as。

www.TopSage.com

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class SingleSegment extends Sprite
    {
        private var slider:SimpleSlider;
327     private var segment:Segment;
        public function SingleSegment()
        {
            init();
        }

        private function init():void
        {
            segment = new Segment(100, 20);
            addChild(segment);
            segment.x = 100;
            segment.y = 100;

            slider = new SimpleSlider(-90, 90, 0);
            addChild(slider);
            slider.x = 300;
            slider.y = 20;
            slider.addEventListener(Event.CHANGE, onChange);
        }

        private function onChange(event:Event):void
        {
            segment.rotation = slider.value;
        }
    }
}
```

上面的代码只是在说当滑钮的value值发生了变化，它将调用onChange方法，该方法将设置segment的旋转大小为滑钮的value值。测试它，你将会看到如图13-2所示的效果。如果代码都能正确工作，那么你已经完成了正向运动的第一步。

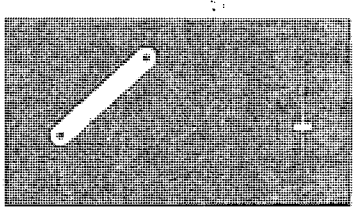


图13-2　它移动了

13.2.2　移动两个关节

使用一个关节工作是非常简单的而且也是乏味的，但它是必需的。现在让我们继续。原始的滑钮和关节将被命名为slider0和segment0。创建另一个Segment的实例，并将它命名为segment1。同样也创建另一个滑钮实例并命名为slider1。新的滑钮将控制新的关节旋转，新的关节将被放置在由segment0的getPin()方法返回的位置点上。

这里是代码，你可以在TwoSegments.as中找到：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
```

www.TopSage.com

```
public class TwoSegments extends Sprite
{
    private var slider0:SimpleSlider;
    private var slider1:SimpleSlider;
    private var segment0:Segment;
    private var segment1:Segment;

    public function TwoSegments()
    {
        init();
    }

    private function init():void
    {
        segment0 = new Segment(100, 20);
        addChild(segment0);
        segment0.x = 100;
        segment0.y = 100;

        segment1 = new Segment(100, 20);
        addChild(segment1);
        segment1.x = segment0.getPin().x;
        segment1.y = segment0.getPin().y;

        slider0 = new SimpleSlider(-90, 90, 0);
        addChild(slider0);
        slider0.x = 320;
        slider0.y = 20;
        slider0.addEventListener(Event.CHANGE, onChange);

        slider1 = new SimpleSlider(-90, 90, 0);
        addChild(slider1);
        slider1.x = 340;
        slider1.y = 20;
        slider1.addEventListener(Event.CHANGE, onChange);
    }

    private function onChange(event:Event):void
    {
        segment0.rotation = slider0.value;
        segment1.rotation = slider1.value;
        segment1.x = segment0.getPin().x;
        segment1.y = segment0.getPin().y;
    }
  }
}
```

328

快速地看一下onChange方法的最下面，你将会看到它现在包含了基于segment0.getPin()的返回值定位setment1的代码。它与第1次创建关节的代码是相同的，以确保它开始的位置是正确的。

329

你设置了slider1来调用onChange方法，其方式与slider0的相同。并且，很显然，你已经得到了基于slider1的segment1的旋转。

www.TopSage.com

大家网

如果测试这个文件，你将会看到当旋转segment0时，segment1保持在segment0的端点位置上，如图13-3所示。意识到在两个关节之间并没有真正的物理连接。它完全是由数学来完成的。你可以使用滑钮自由地旋转segment1。为了更有趣一点，改变其中一个关节的宽或高，或两个关节都改变。你会看到它们仍然工作得非常好。

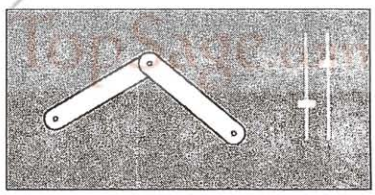


图13-3　两个关节的正向运动

有一件事情看上去有点奇怪，那就是当segment1随着segment0移动时，它并不会随着它旋转。就像有陀螺稳定器在里面一样，保持着它原有稳定的状态。我不知道你怎么样，但是我的前臂里没有陀螺稳定器（尽管它看上去挺酷），因此对我来说，这看起来是不正常的。真正应当发生的事情是segment1的旋转等于segment0的旋转加上slider1的value值。TwoSegment2.as文档类使用下面的代码来控制它。

```
private function onChange(event:Event):void
{
    segment0.rotation = slider0.value;
    segment1.rotation = segment0.rotation + slider1.value;
    segment1.x = segment0.getPin().x;
    segment1.y = segment0.getPin().y;
}
```

现在，看起来像一个真正的手臂了。当然，如果正在谈论的是人的手臂，你可能不喜欢现在的肘部可以在两个方向上弯曲的方式。只需要更改slider1的范围，让它的minimum为–160，maximum为0，如下面的代码，这样看起来就像是正常的手臂了。

```
slider1 = new SimpleSlider(-160, 0, 0);
```

这时可能是一个反映出正向运动术语的好机会。这个系统的基础端是segment0的轴心点。自由端是segment0的另一个端点。你可以想象有一只手在那儿。系统基础端的旋转和位置决定着segment1的位置。segment1的旋转和位置决定着自由端的位置。对于自由端，没办法说它在哪儿、应该在哪儿或想在哪儿。因此，控制是从基础端移向自由端的。

13.3　自动运行

所有这些滑钮可以让你在很大程度上控制旋转，但是你所创建的滑钮像一部建筑机器，液压操纵杆围绕零件移动。如果你想让它真正地动起来，就需要给它一些自我控制的能力。

对于每一个关节你需要一种方式来让它们光滑地前后移动，然后就是如何让它们同步。这听起来有点像是正弦波的工作。

330

在Walking1.as中，我已经使用三角函数替换了滑钮。它使用的是cycle变量（初始值为0）的正弦值并且将它乘以90，结果为90到–90之间的一个值。cycle变量不断地增加，因此你就得到了一个振荡。现在，我使用得到的angle变量来控制两个关节。我加了一个enterFrame处理器来控制这些动作，因此运动是连续的。

```
package {
    import flash.display.Sprite;
```

www.TopSage.com

```
import flash.events.Event;

public class Walking1 extends Sprite
{
    private var segment0:Segment;
    private var segment1:Segment;
    private var cycle:Number = 0;

    public function Walking1()
    {
        init();
    }

    private function init():void
    {
        segment0 = new Segment(100, 20);
        addChild(segment0);
        segment0.x = 200;
        segment0.y = 200;

        segment1 = new Segment(100, 20);
        addChild(segment1);
        segment1.x = segment0.getPin().x;
        segment1.y = segment0.getPin().y;

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        cycle += .05;
        var angle:Number = Math.sin(cycle) * 90;
        segment0.rotation = angle;
        segment1.rotation = segment0.rotation + angle;
        segment1.x = segment0.getPin().x;
        segment1.y = segment0.getPin().y;
    }
}
}
```

331

13.3.1 创建自然的行走循环

现在你已经有了移动起来非常逼真的手臂了。让我们把它变成一条腿。做如下修改：

(1) 将segment0的旋转加上90来降低它升级的高度并减小它的运动范围，在两个方向上从90度降低到45度。

(2) 为每一个关节使用一个单独的角度，因此你将有angle0和angle1。

(3) 将angle1的范围减到45，然后再加上45。这样它的最终旋转范围为0到90，以使它只可以在一个方向上弯曲，就像一个真实的膝盖。如果你还不是很清楚，试着使用增加45和不增加45两种方式来看它们的变化，并且可以试着使用其他的数值，直到你认为它们是合适的为止。

按照上面的步骤完成Walking2.as文件。这里我只显示了onEnterFrame方法，因为其他地方

www.TopSage.com

并没有改动：

```
private function onEnterFrame(event:Event):void
{
    cycle += .05;
    var angle0:Number = Math.sin(cycle) * 45 + 90;
    var angle1:Number = Math.sin(cycle) * 45 + 45;
    segment0.rotation = angle0;
    segment1.rotation = segment0.rotation + angle1;
    segment1.x = segment0.getPin().x;
    segment1.y = segment0.getPin().y;
}
```

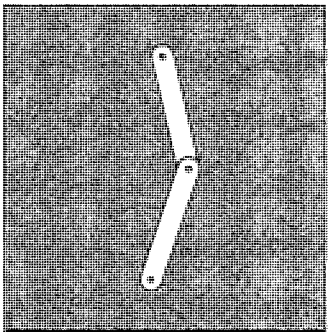


图13-4　初级的行走循环

好，你已经做到了，如图13-4所示。它开始看起来像一条腿了，或至少它开始像腿一样移动了。

这里的问题是它看起来并不像是在行走。可能它正在漫不经心地踢着足球，或是在练习芭蕾舞的某个动作，但并不是在行走。现在发生的是两个关节同时向同一方向移动。它们完全是同步的，如果你分析过真实的行走循环，它并不是这样的。

332

关节同步是因为它们两者都使用cycle变量来计算它们的角度。要想让它们不同步，你可以使用cycle0和cycle1变量，但实际上不需要这样去走弯路。当使用cycle变量计算angle1时只需要将它偏移一点，像这样：

```
var angle1:Number = Math.sin(cycle + offset) * 45 + 45;
```

当然，你需要在你的代码中提前定义offset。但是offset应是多大呢？我并不知道要将它设置为多少。实验一下直到你找到一个看上去合适的值。我将给你一个提示：它应是Math.PI和-Math.PI（或3.14和-3.14）之间的数值。任何大于或小于这之间的数值都将产生不真实的感觉。例如，我使用-Math.PI/2，它是cycle变量的四分之一。当然，-Math.PI/2大约是-1.57，那么也许你想尝试一个这个值周围的其他数值，像-1.7或-1.3，可以看一下它们的效果是变好了还是变差了。一会，我将使用滑钮来生成完全动态的行走。使用了偏移的文件是Walking3.as。

现在，完全是“一条腿走路”的情况，那么让我们来加入另外一条腿。你需要再加入两个关节，名为segment2和segment3。segment2的位置与segment0的位置相同。因为它也是顶级，或称是整个腿的基础端。segment3使用segment1的getPin()方法来决定它的位置。

现在，并不是复制使segment0和segment1行走的所有代码，我将它抽象为它自己的方法，称为walk：

```
private function walk(segA:Segment, segB:Segment, cyc:Number):void
{
    var angleA:Number = Math.sin(cyc) * 45 + 90;
    var angleB:Number = Math.sin(cyc + offset) * 45 + 45;
    segA.rotation = angleA;
    segB.rotation = segA.rotation + angleB;
    segB.x = segA.getPin().x;
    segB.y = segA.getPin().y;
}
```

注意这个函数中有3个参数：两个关节segA和segB，还有cyc，它代表cycle变量。剩下的代

www.TopSage.com

码基本上是你已经使用过的代码。现在，要使segment0和segment1行走，只需要像这样调用它：

```
walk(segment0, segment1, cycle);
```

现在为止，你应当已经明白我要用它做什么，并且准备也让segment2和segment3行走。如果你做得正确，你会这样写onEnterFrame方法：

```
private function onEnterFrame(event:Event):void
{
    walk(segment0, segment1, cycle);
    walk(segment2, segment3, cycle);
    cycle += .05;
}
```

但是如果测试它，你可能想知道第2条腿在哪儿。问题在于两条腿完全是同步状态，因此它们看上去就像是一条。再一次，你需要让它们不同步。上次，你偏移下面的关节以使它与上面的关节有不同的位置。这一次，你将偏移的是第2条腿，让它与第1条腿位置不同。再一次，需要改变cycle的值。而且再一次，并不是跟踪两个不同的变量，只是在将cycle传递给walk方法之前，将它加上或减去某个值。因此onEnterFrame方法变为如下这样：

333

```
private function onEnterFrame(event:Event):void
{
      walk(segment0, segment1, cycle);
      walk(segment2, segment3, cycle + Math.PI);
      cycle += .05;
}
```

为什是Math.PI？比较长的答案是将第2条腿加上180度以使它与第1条腿不同步，那么当第1条腿向前时，第2条腿就会向后移动。短的答案就是因为它工作得很好！你可尝试使用不同的数值，假设是Math.PI/2，那么它看上去更像是在飞奔而不是在行走或跑。但是你可以记住它——你可能哪天会用到飞奔！

此内容的文件为Walking4.as并且结果如图13-5所示。注意我使基础关节（“大腿”）比下面的关节（“小腿”）更大一些。另外，记住因为所有内容都是动态设置的，所以不管大小是多少它都会工作。在下一个版本中，你将结合使用滑钮制作更多动态的内容，但现在我强烈建议你手工使用一些变量来测试当前这个文件，通过改变代码中的变量值来观察这些数值对效果的影响。

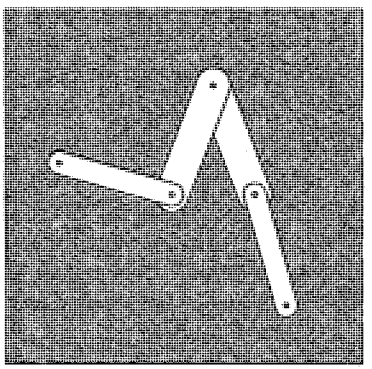


图13-5 看！它已经行走了

13.3.2 动态化

接下来，让我们真正试验这个行走循环，并且通过改变不同的变量值看一下它可以改变多少。文件Walking4.as文件再次使用了Slider类以使你可以动态改变它的变量。

334

在这个示例中，我在影片的顶端创建并放置了5个滑钮，如图13-6所示。

www.TopSage.com

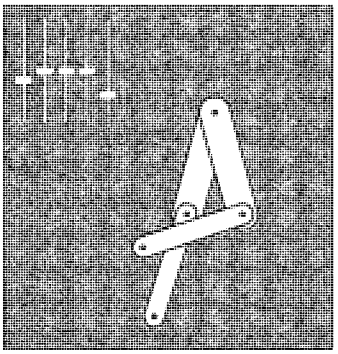


图13-6　加入滑钮

表13-1显示了滑钮的名称（从左到右）、它们做什么和对它们的设置。它们只是一些我认为合适的范围和值。所有这些，你都可以用其他数值来进行实验。

表13-1　使用滑钮控制行走循环

名　称	描　述	设　置
speedSlider	控制系统移动的速度	minimum:0,maximum:0.3,value:0.12
thighRangeSlider	控制顶级关节（大腿）可以前后移动的范围	minimum:0,maximum:90,value:45
thighBaseSlider	控制顶级关节的初始角度，目前设置为90，意味着当前的腿是垂直向下的，并从这个状态开始前后移动，你可以通过改变这个值来得到一些有趣的效果	minimum:0,maximum:1B0, value: 90
calfRangeSlider	控制下面关节（小腿）移动的范围	minimum:0,maximum:90,value:45
calfOffsetSlider	控制偏移值（现在你使用的是-Math.PI/2）	minimum:-3.14,maximum:3.14, value: -1.57

现在你需要进入到程序中修改代码，以使它可以使用滑钮提供的数值，而不是使用硬编码的数值。

335

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Walking5 extends Sprite
    {
        private var segment0:Segment;
        private var segment1:Segment;
        private var segment2:Segment;
        private var segment3:Segment;
        private var speedSlider:SimpleSlider;
        private var thighRangeSlider:SimpleSlider;
        private var thighBaseSlider:SimpleSlider;
        private var calfRangeSlider:SimpleSlider;
        private var calfOffsetSlider:SimpleSlider;
        private var cycle:Number = 0;

        public function Walking5()
```

www.TopSage.com

TopSage.com

```
{
    init();
}

private function init():void
{
    segment0 = new Segment(100, 30);
    addChild(segment0);
    segment0.x = 200;
    segment0.y = 100;

    segment1 = new Segment(100, 20);
    addChild(segment1);
    segment1.x = segment0.getPin().x;
    segment1.y = segment0.getPin().y;

    segment2 = new Segment(100, 30);
    addChild(segment2);
    segment2.x = 200;
    segment2.y = 100;

    segment3 = new Segment(100, 20);
    addChild(segment3);
    segment3.x = segment2.getPin().x;
    segment3.y = segment2.getPin().y;

    speedSlider = new SimpleSlider(0, 0.3, 0.12);
    addChild(speedSlider);
    speedSlider.x = 10;
    speedSlider.y = 10;
    thighRangeSlider = new SimpleSlider(0, 90, 45);
    addChild(thighRangeSlider);
    thighRangeSlider.x = 30;
    thighRangeSlider.y = 10;

    thighBaseSlider = new SimpleSlider(0, 180, 90);
    addChild(thighBaseSlider);
    thighBaseSlider.x = 50;
    thighBaseSlider.y = 10;

    calfRangeSlider = new SimpleSlider(0, 90, 45);
    addChild(calfRangeSlider);
    calfRangeSlider.x = 70;
    calfRangeSlider.y = 10;

    calfOffsetSlider = new SimpleSlider(-3.14, 3.14, -1.57);
    addChild(calfOffsetSlider);
    calfOffsetSlider.x = 90;
    calfOffsetSlider.y = 10;

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
```

336

www.TopSage.com

```
        {
            walk(segment0, segment1, cycle);
            walk(segment2, segment3, cycle + Math.PI);
            cycle += speedSlider.value;
        }

        private function walk(segA:Segment, segB:Segment,
                              cyc:Number):void
        {
            var angleA:Number = Math.sin(cyc) *
                                thighRangeSlider.value +
                                thighBaseSlider.value;
            var angleB:Number = Math.sin(cyc +
                                calfOffsetSlider.value) *
                                calfRangeSlider.value +
                                calfRangeSlider.value;
            segA.rotation = angleA;
            segB.rotation = segA.rotation + angleB;
            segB.x = segA.getPin().x;
            segB.y = segA.getPin().y;
        }

    }
}
```

337

这些代码实际上就是之前的代码，只是现在在文件中使用的是滑钮的数值而不是硬编码的数值。我确信你会对这个文件有很大的兴趣，因为你可以探索不同的行走循环效果。

13.4　使它真正地行走

现目前为止，你已经有两条以非常逼真的方式移动的腿了。但它们只是悬浮在空间中。在本书的前面章节，你已经学习了使用速度和加速度移动某物体，并让它与周围环境交互的相关内容。现在我们要做的是同样的事情。

本章的这一部分有些复杂，因此我采用每次一个概念的方式讲解代码。集合了所有概念的最终文件在RealWalk.as中。

13.4.1　给它一些空间

因为这个东西要真正地来回走动，所以让我们把它的各个部分做得小一些，以使有空间让它来回走动。我使用了原来的4个关节并且让它们的大小是原来的一半，例如：

```
segment0 = new Segment(50, 15);
```

接下来，因为它将要到处移动并且与边界相互作用，所以需要定义变量vx和vy：

```
private var vx:Number = 0;
private var vy:Number = 0;
```

此时，如果你运行这个示例，你得到的是之前版本的小型工作版。

www.TopSage.com

13.4.2 加入重力

接下来，你需要创建一些重力。否则，即使你对边界反应编了程序，但是腿仍是悬浮在空间中。你甚至要使用另外一个滑钮来创建运行时的重力变量。因此，创建一个新的滑钮实例并且命名为gravitySlider。设置minimum为0，maximum为1，value值为0.2。这里是创建最后一个滑钮的部分代码：

```
gravitySlider = new SimpleSlider(0, 1, 0.2);
addChild(gravitySlider);
gravitySlider.x = 110;
gravitySlider.y = 10;
```

现在，你需要结合重力加速度来计算速度，并不是将所有计算速度的代码塞到onEnterFrame方法中，只是加入了一个doVelocity方法的调用：

338

```
private function onEnterFrame(event:Event):void
{
    doVelocity();
    walk(segment0, segment1, cycle);
    walk(segment2, segment3, cycle + Math.PI);
    cycle += speedSlider.value;
}
```

在doVelocity方法中，只需要将重力加到vy，然后将vx和vy加到segment0和segment2的位置上。记住你不需要担心segment1和segment3的位置，因为它们的位置是根据更高级别的关节来计算的。

```
private function doVelocity():void
{
    vy += gravitySlider.value;
    segment0.x += vx;
    segment0.y += vy;
    segment2.x += vx;
    segment2.y += vy;
}
```

如果你需要可以测试这个版本，但它并不能让你很激动。这里并没有*x*方向的速度发生，并且重力只是将腿拉到地板上。因此，你需要检测腿是否有碰到地板。这意味着是我们进行碰撞检测的时候了。

13.4.3 控制碰撞

开始，让onEnterFrame方法调用另外一个方法checkFloor。它将发生在调用walk之后，这样它可以对最新位置进行操作。通常来讲，你只需要检测segment1和segment3——更低级别的关节——来观察它们是否碰到地板。因此对两者都调用checkFloor方法。

```
private function onEnterFrame(event:Event):void
{
    doVelocity();
    walk(segment0, segment1, cycle);
    walk(segment2, segment3, cycle + Math.PI);
```

www.TopSage.com

```
        cycle += speedSlider.value;
        checkFloor(segment1);
        checkFloor(segment3);
    }
```

现在进入了第1个有趣的部分：碰撞检测。注意我说的是“有趣”而不是“不难”。它实际上很简单。你需要知道的是正在讨论中的关节的任何部分是否到了底端边界（通过bottom变量来指定）。

339

可能最简单的实现方法是通过对关节调用getBounds方法并观察边界的bottom属性是否大于舞台的高度。下面是开始checkFloor的方法：

```
private function checkFloor(seg:Segment):void
{
    var yMax:Number = seg.getBounds(this).bottom;
    if(yMax > stage.stageHeight)
    {

    }
}
```

在函数的第1行我使用了简化的形式，它对你来说可能看起来有点儿奇怪。

```
var yMax:Number = seg.getBounds(this).bottom;
```

正常情况下，你可能需要两步，如下：

```
var bounds:Rectangle = seg.getBounds(this);
var yMax:Number = bounds.bottom;
```

但有一些人没有意识到，seg.getBounds(this)语句返回的是一个Rectangle，你可以像这样直接从这个语句访问Rectangle的属性：seg.getBounds(this).bottom。Flash首先作为一个Rectangle对setBounds的调用求值，然后它会看到那个点和后面的属性，并查找对象的这个属性。如果这样的语法让你有些迷惑不解，那么你可以使用两行语法的方式，记得要先导入flash.geom.Rectangle类。如果你想用bounds对象来做其他的一些事情，如读取一些附加的属性，需要先存储它们。但在这个例子中，你只是对bottom有兴趣，因此，你应该立刻获取它而没有必要去获取其他的属性。

现在假设你已经检测到yMax值大于stage.stageHeight，或使用现实中的语言，腿已经碰到了地板。你要做什么呢？好，就像其他的边界碰撞一样，首先需要移动物体以使它刚好接触到边界。意识到如果yMax是关节的最底端，而stage.stageHeight是地板，你需要将关节移回，移动的大小是它们之间的距离的大小。换句话说，假设stage.stageHeight是600，而yMax是620，你需要将关节的位置移动-20。但是你并不只是想移动这个关节，而是需要将所有的关节移动这么多，因为它们都是整体的一部分并且必须作为整体移动。因此代码如下：

```
private function checkFloor(seg:Segment):void
{
    var yMax:Number = seg.getBounds(this).bottom;
    if(yMax > stage.stageHeight)
    {
        var dy:Number = yMax - stage.stageHeight;
```

www.TopSage.com

```
            segment0.y -= dy;
            segment1.y -= dy;
            segment2.y -= dy;
            segment3.y -= dy;
        }
    }
```

340

这样的重复是值得的。调整滑钮的数值来观察不同的行走循环动作。你会对腿与环境的真正交互有更多的感觉。当然，它仍然没有直正地行走，继续下一节。

13.4.4 控制反应

现在，你已经成功地让腿与地板碰撞了，但除了重定位它们自己，并没有真正的反应。在这个例子中，要实现行走就需要有水平方向的运动——*x*速度。而且，你的行走循环也应该给你一点*y*方向的速度——至少要足以克服重力。你会在多数快速跑的腿部循环中看到短暂的悬停。

看它的一种方式是你的脚正在向下移动。当它碰到地面时，它不能够再继续向下移动，因此竖直方向的动量会返回到你的身体上，使身体向上移动。你的脚向下移动的力越大，你得到的提升力就越大。同样的道理，如果你的脚碰到地面时向后移动，水平方向的动量就会返回到你的身体上，身体就会向前运动。你的脚移动得越快，得到的水平的推力就越大。

好的，我们这里已经有了原理。如果你可以距踪“脚”的*x*和*y*的速度，那么你得到碰撞时，你可以从vx和vy数值中减去*x*和*y*速度。

你遇到的第一个问题是你并没有任何的脚存在。实际上，尽管你自己可以加上一些真实的脚，并按照计算第2个关节的位置的方式计算出它们的位置，但在本例中我并不打算使用真实的脚。相反，我只是计算虚拟脚的位置，你可能已经猜出来了，该位置是较低的关节的getPin()返回的数值。

如果你可以跟踪出轴栓在关节移动前和移动后的位置，就可以将两者相减得到“脚”在*x*和*y*方向的速度。你可以在walk方法中计算它并将值保存在关节的公开的vx和vy属性中（现在你知道它们从哪儿来的）。注意，因为你使用了Point类，所以你需要先导入flash.geom.Point类。

```
private function walk(segA:Segment, segB:Segment, cyc:Number):void
{
    var foot:Point = segB.getPin();
    var angleA:Number = Math.sin(cyc) *
                        thighRangeSlider.value +
                        thighBaseSlider.value;
    var angleB:Number = Math.sin(cyc + calfOffsetSlider.value) *
                        calfRangeSlider.value +
                        calfRangeSlider.value;
    segA.rotation = angleA;
    segB.rotation = segA.rotation + angleB;
    segB.x = segA.getPin().x;
    segB.y = segA.getPin().y;
    segB.vx = segB.getPin().x - foot.x;
    segB.vy = segB.getPin().y - foot.y;
}
```

341

www.TopSage.com

大家网 TopSage.com

现在每一个底部的关节都有一个vx和vy属性，它描述的不是关节自身的速度，而是底下的轴心点或虚拟脚的速度。

那么，这个速度用来做什么呢？好，等到脚与地板有碰撞发生时，从总的速度中减去这个速度。换句话说，如果发生碰撞时脚正以每帧3像素的速率向下移动（vy为3），则从总的vy中减去3，vx也是同样。在代码中，它确实很简单：

```
private function checkFloor(seg:Segment):void
{
    var yMax:Number = seg.getBounds(this).bottom;
    if(yMax > stage.stageHeight)
    {
        var dy:Number = yMax - stage.stageHeight;
        segment0.y -= dy;
        segment1.y -= dy;
        segment2.y -= dy;
        segment3.y -= dy;
        vx -= seg.vx;
        vy -= seg.vy;
    }
}
```

这里，我将不再解释为什么这样描述力在行走中的作用是极其简单而且完全不准确的。但是我会笑着说“测试影片”。看看你是否喜欢这个效果。对我来说它真地很酷。我已经得到了两条腿穿过屏幕的效果。

13.4.5　屏幕折回

你可能已经注意到两条腿走出屏幕，再也没有回来。使用屏幕折回可以修复它。当腿离开右边界时，将它们移回到左边。它比之前的屏幕折回要复杂一些，因为现在你在同时移动4部分，而不是一个物体。另外注意，你只需要检测两个顶级关节中的一个，因为它们的位置总是相同的，而下面的关节的位置完全取决于上一级的关节。在onEnterFrame方法的结尾处加入对名为checkWalls的调用：

```
private function onEnterFrame(event:Event):void
{
    doVelocity();
    walk(segment0, segment1, cycle);
    walk(segment2, segment3, cycle + Math.PI);
    cycle += speedSlider.value;
    checkFloor(segment1);
    checkFloor(segment3);
    checkWalls();
}
```

让我们保留100像素的边缘，以使任何一条腿在折回之前可以离开屏幕右侧100像素。如果它超出了，则将所有的东西重新定位到左边。到左边有多远？舞台的宽度加上200，每一边的边缘是100。因此，在checkWalls方法中的if语句像这样：

342

```
private function checkWalls():void
```

www.TopSage.com

```
    {
        var w:Number = stage.stageWidth + 200;
        if(segment0.x > stage.stageWidth + 100)
        {
            segment0.x -= w;
            segment1.x -= w;
            segment2.x -= w;
            segment3.x -= w;
        }
    }
```

然后左边也是一样的，因为有些行走循环可以真正让腿向后走。这里是最终的checkWalls方法：

```
    private function checkWalls():void
    {
        var w:Number = stage.stageWidth + 200;
        if(segment0.x > stage.stageWidth + 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;
        }
    }
```

现在你已经完成了。为了不至于让你感觉混乱，所有的代码在下面（可以在RealWalk.as中找到）。

```
    package {
        import flash.display.Sprite;
        import flash.events.Event;
        import flash.geom.Point;

        public class RealWalk extends Sprite
        {
            private var segment0:Segment;
            private var segment1:Segment;
            private var segment2:Segment;
            private var segment3:Segment;
            private var speedSlider:SimpleSlider;
            private var thighRangeSlider:SimpleSlider;
            private var thighBaseSlider:SimpleSlider;
            private var calfRangeSlider:SimpleSlider;
            private var calfOffsetSlider:SimpleSlider;
            private var gravitySlider:SimpleSlider;
            private var cycle:Number = 0;
```

343

www.TopSage.com

```
private var vx:Number = 0;
private var vy:Number = 0;

public function RealWalk()
{
    init();
}

private function init():void
{
    segment0 = new Segment(50, 15);
    addChild(segment0);
    segment0.x = 200;
    segment0.y = 100;

    segment1 = new Segment(50, 10);
    addChild(segment1);
    segment1.x = segment0.getPin().x;
    segment1.y = segment0.getPin().y;

    segment2 = new Segment(50, 15);
    addChild(segment2);
    segment2.x = 200;
    segment2.y = 100;

    segment3 = new Segment(50, 10);
    addChild(segment3);
    segment3.x = segment2.getPin().x;
    segment3.y = segment2.getPin().y;

    speedSlider = new SimpleSlider(0, 0.3, 0.12);
    addChild(speedSlider);
    speedSlider.x = 10;
    speedSlider.y = 10;

    thighRangeSlider = new SimpleSlider(0, 90, 45);
    addChild(thighRangeSlider);
    thighRangeSlider.x = 30;
    thighRangeSlider.y = 10;
    thighBaseSlider = new SimpleSlider(0, 180, 90);
    addChild(thighBaseSlider);
    thighBaseSlider.x = 50;
    thighBaseSlider.y = 10;

    calfRangeSlider = new SimpleSlider(0, 90, 45);
    addChild(calfRangeSlider);
    calfRangeSlider.x = 70;
    calfRangeSlider.y = 10;

    calfOffsetSlider = new SimpleSlider(-3.14, 3.14, -1.57);
    addChild(calfOffsetSlider);
    calfOffsetSlider.x = 90;
    calfOffsetSlider.y = 10;
```

344

www.TopSage.com

```
        gravitySlider = new SimpleSlider(0, 1, 0.2);
        addChild(gravitySlider);
        gravitySlider.x = 110;
        gravitySlider.y = 10;

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        doVelocity();
        walk(segment0, segment1, cycle);
        walk(segment2, segment3, cycle + Math.PI);
        cycle += speedSlider.value;
        checkFloor(segment1);
        checkFloor(segment3);
        checkWalls();
    }

    private function walk(segA:Segment, segB:Segment,
                          cyc:Number):void
    {
        var foot:Point = segB.getPin();
        var angleA:Number = Math.sin(cyc) *
                            thighRangeSlider.value +
                            thighBaseSlider.value;
        var angleB:Number = Math.sin(cyc +
                            calfOffsetSlider.value) *
                            calfRangeSlider.value +
                            calfRangeSlider.value;
        segA.rotation = angleA;
        segB.rotation = segA.rotation + angleB;
        segB.x = segA.getPin().x;
        segB.y = segA.getPin().y;
        segB.vx = segB.getPin().x - foot.x;
        segB.vy = segB.getPin().y - foot.y;
    }

    private function doVelocity():void
    {
        vy += gravitySlider.value;
        segment0.x += vx;
        segment0.y += vy;
        segment2.x += vx;
        segment2.y += vy;
    }

    private function checkFloor(seg:Segment):void
    {
        var yMax:Number = seg.getBounds(this).bottom;
        if(yMax > stage.stageHeight)
        {
            var dy:Number = yMax - stage.stageHeight;
            segment0.y -= dy;
```

345

www.TopSage.com

```
                segment1.y -= dy;
                segment2.y -= dy;
                segment3.y -= dy;
                vx -= seg.vx;
                vy -= seg.vy;
            }
        }

        private function checkWalls():void
        {
            var w:Number = stage.stageWidth + 200;
            if(segment0.x > stage.stageWidth + 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;
            }
        }
    }
}
```

346

13.5　小结

在本章中你已经制作了非常漂亮的作品，掌握了正向运动的基础内容。注意这里我给出的方法可能并不是唯一的解决方案。它们显然是针对某一特殊的技术应用而设计的：让某物体行走。你可以不用这些，或改变些什么，或是加入你想加入的。不断实验它看你可以制作出什么。

347

下一步，你将看到另一面：反向运动。

www.TopSage.com

大家网
TopSage.com

第14章

反向运动：拖动和伸展

本章内容

- 伸展和拖动一个关节
- 拖动多个关节
- 伸展多个关节
- 使用标准的反向运动方法
- 本章重点公式

在第13章，我已经讲解了一些运动学的基础和正向和反向运动之间的区别。第13章研究的是正向运动。现在你准备研究的是与正向运动紧密联系的反向运动。它包括拖动和伸展两种移动。

与在正向运动中的示例一样，本章中的示例使用关节来创建系统。将从一个关节开始，然后转移到多个关节。首先，我将示范给你计算各种角度和位置的最简单的方法。它类似于使用你已经学过的基本三角学的方法。最后，我将简单地介绍另一种使用余弦定律的方法，它可以更精确，但是是以复杂为代价的——熟悉的折中。

14.1 伸展和拖动一个关节

如我在前一章中所提到的，反向运动系统可以分成两种不同的类型：伸展和拖动。

当系统的自由端正在向一个目标伸展时，系统的另一端——基础端——可能是固定的，因此如果目标点太远，自由端可能永远无法到达目标点。这种情况的一个例子是你努力去抓住某个东西。你的手指移向那个物体，手腕的轴心点让手指尽可能地靠近，同时你的肘、肩和身体的其他部分都在移动，以使你尽可能地靠近物体。有时，所有这些位置合起来会让手指接触到物体；有时，你并不能碰到那个物体。如果物体要从左到右移动，所有的肢体会不断地自行调整位置来保证手指可以尽可能地靠近物体。反向运动将向你展示如何布置每一个部分的位置以达到最佳的伸展状态。

反向运动的另一种类型是当某个物体被拖动时。在这种情况下，自由端被某些外力移动。不管它在哪，整个系统的其他部分都会跟随在它后面，有规律的定位它们自身的位置。对于此，想象一个无知觉的身体或一具尸体（报歉，我只能想到这些）。你抓住它的手并拖动它。你施加给

www.TopSage.com

它的手的力使得手腕、肘、肩膀和身体的其他部分在它们被拖动时围绕轴心点转动并以可能的方式移动。在这个例子中，反向运动向你显示当这些肢体被拖动时，它们是如何定位到正确的位置的。

为了快速向你展示这两种方式的不同之处，让我们快速看一下使用一个关节为每一种方式创建的示例。你将需要我们在上一章中用到的Segment类，所以确保它已经在你的项目或类路径当中。

14.1.1　伸展一个关节

对于伸展，所有关节都将可以转向目标。对于目标，这里指的鼠标。要将关节转向目标，你需要两者之间在*x*和*y*轴上的距离。然后使用Math.atan2得到两者之间的弧度角。将其转换为度后，
350 你就知道如何旋转关节了。这里是代码（你也可以OneSegment.as中找到代码）：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class OneSegment extends Sprite
    {
        private var segment0:Segment;

        public function OneSegment()
        {
            init();
        }

        private function init():void
        {
            segment0 = new Segment(100, 20);
            addChild(segment0);
            segment0.x = stage.stageWidth / 2;
            segment0.y = stage.stageHeight / 2;

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var dx:Number = mouseX - segment0.x;
            var dy:Number = mouseY - segment0.y;
            var angle:Number = Math.atan2(dy, dx);
            segment0.rotation = angle * 180 / Math.PI;
        }
    }
}
```

图14-1显示了结果。测试这个文件并观察关节是如何随着鼠标移动的。即使关节离得很远，你可能看到它是如何向鼠标伸
351 展的。

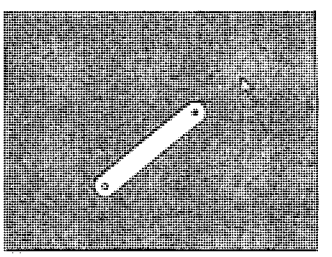


图14-1　一个关节向鼠标伸展

www.TopSage.com

14.1.2 拖动一个关节

现在，让我们试一下拖动。这里，你并不是使用startDrag和stopDrag方法来拖动（尽管你认为可以那样做）。而是你将假设关节的第2个轴心点刚好贴在鼠标的位置上。

拖动方法的第1部分与伸展方法是一模一样的：旋转精灵指向鼠标。然后进一步移动关节以使关节的第2个轴心点放置在鼠标的位置上。要做到这点，你需要知道两个轴栓之间在每个轴上的距离。你可以通过计算关节的getPin()方法获得的位置点与关节实际的*x*,*y*位置之间的差来得到此值。我们称它们为w和h。然后只需要从当前鼠标的位置减去w和h，那么你就可以知道将关节放置在哪里了。这里是OneSegmentDrag.as中的onEnterFrame方法，唯一修改过的一部分：

```
private function onEnterFrame(event:Event):void
{
    var dx:Number = mouseX - segment0.x;
    var dy:Number = mouseY - segment0.y;
    var angle:Number = Math.atan2(dy, dx);
    segment0.rotation = angle * 180 / Math.PI;

    var w:Number = segment0.getPin().x - segment0.x;
    var h:Number = segment0.getPin().y - segment0.y;
    segment0.x = mouseX - w;
    segment0.y = mouseY - h;
}
```

你可以看到关节是如何始终贴在鼠标上并旋转以跟随在鼠标的后面拖动。你也可以向相反的方向推动关节。

14.2 拖动多个关节

使用反向运动拖动一个系统实际上要比伸展简单一点儿，因此我先讲解拖动。让我们以两个关节开始。

14.2.1 拖动两个关节

从上一个示例开始，创建另一个关节，命名为segment1，然后将它加入显示列表中。思路很简单。你已经有了随着鼠标拖动的segment0。你只需要再有一个随着segment0拖动的segment1。开始时，你可以简单地复制和粘贴代码，然后改变一些引用。新的代码使用的是粗体。

```
private function onEnterFrame(event:Event):void
{
    var dx:Number = mouseX - segment0.x;
    var dy:Number = mouseY - segment0.y;
    var angle:Number = Math.atan2(dy, dx);
    segment0.rotation = angle * 180 / Math.PI;
    var w:Number = segment0.getPin().x - segment0.x;
    var h:Number = segment0.getPin().y - segment0.y;
    segment0.x = mouseX - w;
    segment0.y = mouseY - h;
```

352

www.TopSage.com

```
        dx = segment0.x - segment1.x;
        dy = segment0.y - segment1.y;
        angle = Math.atan2(dy, dx);
        segment1.rotation = angle * 180 / Math.PI;

        w = segment1.getPin().x - segment1.x;
        h = segment1.getPin().y - segment1.y;
        segment1.x = segment0.x - w;
        segment1.y = segment0.y - h;
    }
```

在新的代码块中，你已经看到了如何计算出segment1到segment0的距离，并使用它来计算segment1的角度、旋转和位置。你可以测试这个示例并观察它是一个多么真实的两个关节系统。

现在，你有许多复制的代码在里面，这种方式并不是很好。如果你想加入更多的关节，这个代码将会变得越来越长，而所有的代码都是重复的。这里的解决方法是将复制代码移到它自己的函数中，称之为drag。这个函数需要知道哪个关节要拖动和拖动到的*x*,*y*点。那么你就可以拖动segment0到mouseX,mouseY和拖动setgment1到segment0.x,segment0.y。这里是全部的代码（它也出现在文件TwoSegmentDrag.as中）：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class TwoSegmentDrag extends Sprite
    {
        private var segment0:Segment;
        private var segment1:Segment;

        public function TwoSegmentDrag()
        {
            init();
        }

        private function init():void
        {
            segment0 = new Segment(100, 20);
            addChild(segment0);
            segment1 = new Segment(100, 20);
            addChild(segment1);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            drag(segment0, mouseX, mouseY);
            drag(segment1, segment0.x, segment0.y);
        }

        private function drag(segment:Segment,
                              xpos:Number,
```

353

www.TopSage.com

```
                                ypos:Number):void
        {
            var dx:Number = xpos - segment.x;
            var dy:Number = ypos - segment.y;
            var angle:Number = Math.atan2(dy, dx);
            segment.rotation = angle * 180 / Math.PI;

            var w:Number = segment.getPin().x - segment.x;
            var h:Number = segment.getPin().y - segment.y;
            segment.x = xpos - w;
            segment.y = ypos - h;
        }
    }
}
```

14.2.2 拖动更多的关节

现在你可以加入你想要的更多的关节。假设你总共放了6个关节，名为segment0到segment5，将它们放在一个数组中，使用一个for循环针对每一个关节调用drag函数。你可以在MultiSegmentDrag.as中看到这个示例。这里是文件的代码：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class MultiSegmentDrag extends Sprite
    {
        private var segments:Array;
        private var numSegments:uint = 6;

        public function MultiSegmentDrag()
        {
            init();
        }
        private function init():void
        {
            segments = new Array();
            for(var i:uint = 0; i < numSegments; i++)
            {
                var segment:Segment = new Segment(50, 10);
                addChild(segment);
                segments.push(segment);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            drag(segments[0], mouseX, mouseY);
            for(var i:uint = 1; i < numSegments; i++)
            {
                var segmentA:Segment = segments[i];
                var segmentB:Segment = segments[i - 1];
```

354

www.TopSage.com

大家网 TopSage.com

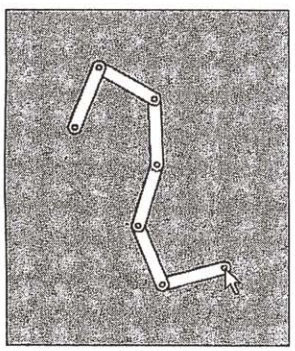
```
                drag(segmentA, segmentB.x, segmentB.y);
            }
        }

        private function drag(segment:Segment,
                              xpos:Number,
                              ypos:Number):void
        {
            var dx:Number = xpos - segment.x;
            var dy:Number = ypos - segment.y;
            var angle:Number = Math.atan2(dy, dx);
            segment.rotation = angle * 180 / Math.PI;

            var w:Number = segment.getPin().x - segment.x;
            var h:Number = segment.getPin().y - segment.y;
            segment.x = xpos - w;
            segment.y = ypos - h;
        }
    }
}
```

这里，segmentB是正在被拖动的关节所要到达的目标点，而segmentA是正在被拖动的关节。
355 你只需要将它们传递给drag函数。如图14-2显示了结果。

好，这里你已经有了基本的反向运动。这并不是很复杂，不是吗？你可以加入更多的关节，只需要更改numSegments变量的值。如图14-3所示，你可以看到50个关节，这表明了这个系统是多么地健壮。



356 图14-2　拖动多个关节

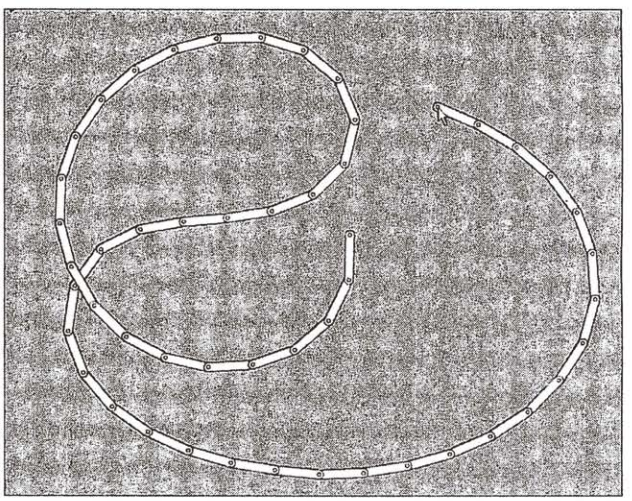


图14-3　拖动50个关节

14.3　伸展多个关节

要开始反向运动的伸展，你需要使用本章的第1个示例OneSegment.as，并向其中添加代码。

www.TopSage.com

这个文件只是让关节旋转指向一个目标点，目标点为鼠标的位置。

14.3.1 向鼠标伸展

首先，你需要确定关节在哪个位置可以准确地接触到目标点。这与拖动关节时计算位置的方法一样。然而，在这个例子中，你其实并没有移动关节。你只需要找到位置。那么，位置用来做什么呢？使用它来作为下一个关节的目标点，并且让这个关节旋转到这个位置。当你到达系统的基础端时，你的工作又回到起始状态，定位每一部分在它的父级末端。图14-4显示了它是如何工作的。

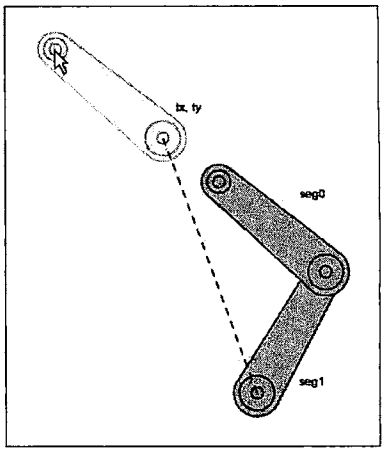


图14-4 segment0旋转向鼠标。tx,ty是segment0要达到的位置。segment1将向tx,ty位置旋转。

这一章的第1个文件OneSegment.as只有一个关节segment0，它向鼠标伸展。我们将创建另一个关节名为segment1，然后将它加入显示列表中。下一步是找到segment0会碰到目标的位置。再一次，它与你在拖动示例中移动关节到达的目标点是一样的。但是不要移动它，只存储这个位置。因此，你得到如下代码：

```
private function onEnterFrame(event:Event):void
{
    var dx:Number = mouseX - segment0.x;
    var dy:Number = mouseY - segment0.y;
    var angle:Number = Math.atan2(dy, dx);
    segment0.rotation = angle * 180 / Math.PI;

    var w:Number = segment0.getPin().x - segment0.x;
    var h:Number = segment0.getPin().y - segment0.y;
    var tx:Number = mouseX - w;
    var ty:Number = mouseY - h;
}
```

357

我计算出了tx,ty的位置，因为它是segment1旋转所要达到的目标。

接下来，你可以复制和粘贴并调整旋转代码来让segment1向它的目标点旋转：

```
private function onEnterFrame(event:Event):void
{
    var dx:Number = mouseX - segment0.x;
    var dy:Number = mouseY - segment0.y;
    var angle:Number = Math.atan2(dy, dx);
    segment0.rotation = angle * 180 / Math.PI;

    var w:Number = segment0.getPin().x - segment0.x;
    var h:Number = segment0.getPin().y - segment0.y;
    var tx:Number = mouseX - w;
    var ty:Number = mouseY - h;

    dx = tx - segment1.x;
    dy = ty - segment1.y;
    angle = Math.atan2(dy, dx);
```

www.TopSage.com

```
    segment1.rotation = angle * 180 / Math.PI;
}
```

这些粗体代码与函数中的前4行一样，只是使用了一个不同的关节和不同的目标点。

最后，重新定位segment0以使它位于segment1的末端，因为现在segment1已经旋转到了一个不同的位置。

```
private function onEnterFrame(event:Event):void
{
    var dx:Number = mouseX - segment0.x;
    var dy:Number = mouseY - segment0.y;
    var angle:Number = Math.atan2(dy, dx);
    segment0.rotation = angle * 180 / Math.PI;

    var w:Number = segment0.getPin().x - segment0.x;
    var h:Number = segment0.getPin().y - segment0.y;
    var tx:Number = mouseX - w;
    var ty:Number = mouseY - h;

    dx = tx - segment1.x;
    dy = ty - segment1.y;
    angle = Math.atan2(dy, dx);
    segment1.rotation = angle * 180 / Math.PI;
    segment0.x = segment1.getPin().x;
    segment0.y = segment1.getPin().y;
}
```

358

当测试这个示例时，你将会看到这些关节作为一个整体在向鼠标伸展。现在，整理一下代码以使你可以容易地加入更多的关节。首先，将所有旋转相关内容移到它自己的函数中，称之为reach。

```
private function reach(segment:Segment, xpos:Number, ypos:Number):Point
{
    var dx:Number = xpos - segment.x;
    var dy:Number = ypos - segment.y;
    var angle:Number = Math.atan2(dy, dx);
    segment.rotation = angle * 180 / Math.PI;

    var w:Number = segment.getPin().x - segment.x;
    var h:Number = segment.getPin().y - segment.y;
    var tx:Number = xpos - w;
    var ty:Number = ypos - h;
    return new Point(tx, ty);
}
```

注意函数的返回类型是Point，而且最后一行基于tx和ty创建并返回了一个Point。这允许你调用reach函数来旋转关节，并且它将返回目标点，你可以将这个目标点传递给下一次调用的函数。不要忘记导入Point类。那么，onEnterFrame函数变成了这样：

```
private function onEnterFrame(event:Event):void
{
    var target:Point = reach(segment0, mouseX, mouseY);
    reach(segment1, target.x, target.y);
```

www.TopSage.com

```
        segment0.x = segment1.getPin().x;
        segment0.y = segment1.getPin().y;
    }
```

这里，segment0总是向鼠标伸展，而segment1向segment0伸展。那么我们也可以将定位代码移到它自己的函数中，称之为position。

```
    private function position(segmentA:Segment, segmentB:Segment):void
    {
        segmentA.x = segmentB.getPin().x;
        segmentA.y = segmentB.getPin().y;
    }
```

然后你可以将segment0重新定位到segment1的结尾，代码如下：

359

```
    position(segment0, segment1);
```

这里是TwoSegmentReach.as的最终代码：

```
    package {
        import flash.display.Sprite;
        import flash.events.Event;
        import flash.geom.Point;

        public class TwoSegmentReach extends Sprite
        {
            private var segment0:Segment;
            private var segment1:Segment;

            public function TwoSegmentReach()
            {
                init();
            }

            private function init():void
            {
                segment0 = new Segment(100, 20);
                addChild(segment0);

                segment1 = new Segment(100, 20);
                addChild(segment1);
                segment1.x = stage.stageWidth / 2;
                segment1.y = stage.stageHeight / 2;

                addEventListener(Event.ENTER_FRAME, onEnterFrame);
            }

            private function onEnterFrame(event:Event):void
            {
                var target:Point = reach(segment0, mouseX, mouseY);
                reach(segment1, target.x, target.y);
                position(segment0, segment1);
            }
```

www.TopSage.com

```
        private function reach(segment:Segment,
                               xpos:Number,
                               ypos:Number):Point
        {
            var dx:Number = xpos - segment.x;
            var dy:Number = ypos - segment.y;
            var angle:Number = Math.atan2(dy, dx);
            segment.rotation = angle * 180 / Math.PI;
        var w:Number = segment.getPin().x - segment.x;
        var h:Number = segment.getPin().y - segment.y;
        var tx:Number = xpos - w;
        var ty:Number = ypos - h;
        return new Point(tx, ty);
    }

    private function position(segmentA:Segment,
                              segmentB:Segment):void
    {
        segmentA.x = segmentB.getPin().x;
        segmentA.y = segmentB.getPin().y;
    }
}
}
```

360

所有的代码都已经准备好了，创建一个数组来保存你想创建的许多关节是很容易的。MultiSegmentReach.as代码如下：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Point;

    public class MultiSegmentReach extends Sprite
    {
        private var segments:Array;
        private var numSegments:uint = 6;

        public function MultiSegmentReach()
        {
            init();
        }

        private function init():void
        {
            segments = new Array();
            for(var i:uint = 0; i < numSegments; i++)
            {
                var segment:Segment = new Segment(50, 10);
                addChild(segment);
                segments.push(segment);
            }
            // 将最后一个放在中心。
            segment.x = stage.stageWidth / 2;
            segment.y = stage.stageHeight / 2;
```

www.TopSage.com

TopSage.com

```
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            var target:Point = reach(segments[0], mouseX, mouseY);
            for(var i:uint = 1; i < numSegments; i++)
            {
                var segment:Segment = segments[i];
                target = reach(segment, target.x, target.y);
            }
            for(i = numSegments - 1; i > 0; i--)
            {
                var segmentA:Segment = segments[i];
                var segmentB:Segment = segments[i - 1];
                position(segmentB, segmentA);
            }
        }

        private function reach(segment:Segment,
                               xpos:Number,
                               ypos:Number):Point
        {
            var dx:Number = xpos - segment.x;
            var dy:Number = ypos - segment.y;
            var angle:Number = Math.atan2(dy, dx);
            segment.rotation = angle * 180 / Math.PI;

            var w:Number = segment.getPin().x - segment.x;
            var h:Number = segment.getPin().y - segment.y;
            var tx:Number = xpos - w;
            var ty:Number = ypos - h;
            return new Point(tx, ty);
        }

        private function position(segmentA:Segment,
                                  segmentB:Segment):void
        {
            segmentA.x = segmentB.getPin().x;
            segmentA.y = segmentB.getPin().y;
        }
    }
}
```

你可以从图14-5中看到它的结果。现在，它比你开始所做的要好得多。但是为什么这个关节链要整天追着鼠标？似乎是它自己的意愿。让我们来看一下如果你给它一个玩具会发生什么！

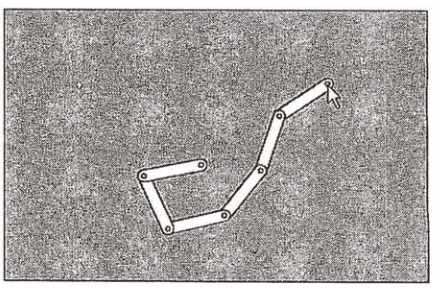


图14-5 伸展多个关节

14.3.2 向一个物体伸展

对于下一个示例，我重新使用了Ball类，因此要把它放在你的项目或类路径中。

www.TopSage.com

然后为小球的移动创建一些变量。注意这部分的代码是以上一个示例为基础的，所以你只需要加入下面的代码：

```
private var ball:Ball;
private var gravity:Number = 0.5;
private var bounce:Number = -0.9;
```

然后，在init方法中，创建Ball类的一个实例并将它加入显示列表中。

```
private function init():void
{
    ball = new Ball();
    ball.vx = 10;
    addChild(ball);
    segments = new Array();
    for(var i:uint = 0; i < numSegments; i++)
    {
        var segment:Segment = new Segment(50, 10);
        addChild(segment);
        segments.push(segment);
    }
    segment.x = stage.stageWidth / 2;
    segment.y = stage.stageHeight;
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

363

在onEnterFrame中，你调用一个名为moveBall的函数。这只是将小球移动的代码与其他代码分开，不至于使整个代码看上去混乱。

```
private function onEnterFrame(event:Event):void
{
    moveBall();
    var target:Point = reach(segments[0], mouseX, mouseY);
    for(var i:uint = 1; i < numSegments; i++)
    {
        var segment:Segment = segments[i];
        target = reach(segment, target.x, target.y);
    }
    for(i = numSegments - 1; i > 0; i--)
    {
        var segmentA:Segment = segments[i];
        var segmentB:Segment = segments[i - 1];
        position(segmentB, segmentA);
    }
}
```

下面是moveBall函数的代码：

```
private function moveBall():void
{
    ball.vy += gravity;
    ball.x += ball.vx;
    ball.y += ball.vy;
    if(ball.x + ball.radius > stage.stageWidth)
    {
        ball.x = stage.stageWidth - ball.radius;
```

www.TopSage.com

```
            ball.vx *= bounce;
        }
        else if(ball.x - ball.radius < 0)
        {
            ball.x = ball.radius;
            ball.vx *= bounce;
        }
        if(ball.y + ball.radius > stage.stageHeight)
        {
            ball.y = stage.stageHeight - ball.radius;
            ball.vy *= bounce;
        }
        else if(ball.y - ball.radius < 0)
        {
            ball.y = ball.radius;
            ball.vy *= bounce;
        }
    }
```

然后修改onEnterFrame函数中的第2行代码以让它向小球伸展而不是鼠标：

```
var target:Point = reach(segments[0], ball.x, ball.y);
```

所有的都在这儿了。你可以看到如图14-6的结果。小球现在四处回弹，而且手臂跟随着它。非常有趣，对吗？

但是，你可以做得更好。现在，手臂在接触小球时做得很好，但是小球完全没有注意到手臂。让我们为它们加入交互。

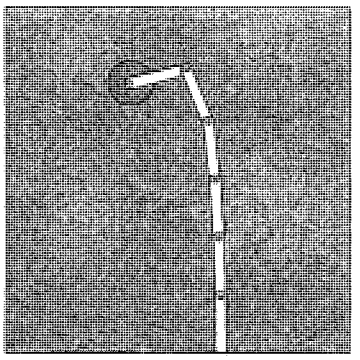


图14-6 它喜欢玩球

14.3.3 加入一些交互

小球与手臂如何交互取决于你想要它们做什么。但是，不管你想要做什么，首先要做的是碰撞检测。然后如果有碰撞发生你可以让它们有反应。另外，你要将所有的内容放在它自己的函数中并且在onEnterFrame中调用它。

```
private function onEnterFrame(event:Event):void
{
    moveBall();
    var target:Point = reach(segments[0], ball.x, ball.y);
    for(var i:uint = 1; i < numSegments; i++)
    {
        var segment:Segment = segments[i];
        target = reach(segment, target.x, target.y);
    }
    for(i = numSegments - 1; i > 0; i--)
    {
        var segmentA:Segment = segments[i];
        var segmentB:Segment = segments[i - 1];
        position(segmentB, segmentA);
    }
    checkHit();
}
```


www.TopSage.com

我已经将这个函数命名为checkHit，并把它放置在函数的最后，这样任何东西都在最终的位置上。下面是checkHit函数：

```
public function checkHit():void
{
    var segment:Segment = segments[0];
    var dx:Number = segment.getPin().x - ball.x;
    var dy:Number = segment.getPin().y - ball.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    if(dist < ball.radius)
    {
        // 碰撞反应。
    }
}
```

首先要做的是获得第1节手臂的末端轴栓与小球之间的距离，然后使用基于距离的碰撞检测来检查是否有碰撞发生。

现在我们回到当你得到一碰撞要怎样做的问题上。这是我的计划：手臂将把小球扔上空中（负的y速度）并在x轴（随机的x速度）上随机地移动它，像这样：

```
public function checkHit():void
{
    var segment:Segment = segments[0];
    var dx:Number = segment.getPin().x - ball.x;
    var dy:Number = segment.getPin().y - ball.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    if(dist < ball.radius)
    {
        ball.vx += Math.random() * 2 - 1;
        ball.vy -= 1;
    }
}
```

它工作得相当好，最终代码可以在PlayBall.as中找到，我可以把它放在那里运行一个晚上，并且到第2天早上，手臂仍很高兴地在玩着它的玩具！但是不要把它作为你应该做的“标准的”任何东西。你可能会想让它抓住球然后向某个目标点扔过去。或许是一个篮球游戏？或有两个手臂在玩捕捉？尝试不同的反应。你一定已经有了足够的工具并可以准备做一些有趣的东西了。

14.4　使用标准的反向运动方法

对你我是绝对诚实的。到目前为止我描述的计算反向运动的方法完全是我自己想出来的。我想我第一次做它时，我可能根本不知道我所做的被称为反向运动。我只是想让某个东西向另外一个东西伸展，并且我得出了每一个部分要怎样做才能完成它，试验它，让它工作，并且将它形成了一个系统使我可以很容易地复制并向其他人描述。它工作得相当好，看起来非常不错，不占用
366 CPU，因此，我对它很满意。我希望你也是如此。

但是，让人吃惊的是，我并不是第一个考虑这个问题的人。许多有着很高智商和很好的数学基础的人已经解决了这个问题，并且提出了与真实的物体移动方式非常相似的另一种解决方案。

www.TopSage.com

那么让我们来看“标准”的反向运动方法。你将有两种不同的方法供你使用而且你可以选择你喜欢的一种方法。

14.4.1 余弦定律介绍

如标题上所示，做反向运动通常使用的方法为余弦定律（law of cosines）。更多的是三角学吗？是的。回顾第3章，所有的示例使用的都是直角三角形——有一个直角（90度）的三角形。对于这样的三角形它的规则是很简单的：正弦等于对边除以斜边，余弦等于邻边除以斜边，等等。在这本书中这些规则使用得很广泛。

但是如果你的三角形不是直角三角形呢？它被冷落了吗？不，古希腊人也想到过这个问题，并且提出了余弦定律来帮助我们计算这种三角形的各种角和边长。当然，它有些复杂，但是如果你对这个三角形有足够的信息，你可以使用这个定律来解决剩下的所有问题。

现在你应该问的问题是“这与反向运动到底有什么关系呢？”好，看一下图14-7的图。

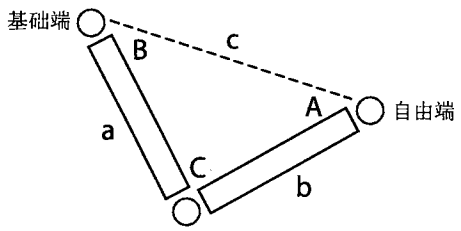


图14-7 来自三角形的两个关节，这个三角形的3条边为a、b、c，3个角为A、B、C。

这里，你有了两个关节。左边的那个是基础端。它是固定的，因此可以知道它的位置。你需要把它的自由端放在显示的位置。你也已经有了一个任意的三角形。

关于这个三角形你都知道什么呢？你可以很容易地计算出两端之间的距离——边c。而你又知道每一个关节的长度——边a和b。那么你就知道了所有3条边的长度。

那么你需要知道这个三角形的什么呢？只需要知道两个关节的两个角——角B和C。余弦定律可以帮你把它们计算出来。接下来我把它介绍给你：

$$c^2 = a^2 + b^2 - 2 * a * b * \cos C$$

367

现在，你需要知道角C，所以你可把它隔离到一边。我不能细致到每一步，因为它只是简单的代数运算。你将得到下面的结果：

$$C = \text{acos}((a2 + b2 - c2) / (2 * a * b))$$

这里的acos是指arccosine，或者说是反余弦。一个角度的余弦给你一个比值，或小数。这个比值的反余弦会给你这个角度。在Flash函数中它是Math.acos()。既然你知道了边a、b和c，所以你可以计算出角C。同样地，你需要知道角B。余弦定律处理如下：

$$b^2 = a^2 + c^2 - 2 * a * c * \cos B$$

计算结果如下：

www.TopSage.com

大家网 TopSage.com

```
B = acos((a² + c² - b²)/ (2 * a * c))
```

将它转换为ActionScript后得到如下：

```
B = Math.acos((a * a + c * c - b * b) / (2 * a * c));
C = Math.acos((a * a + b * b - c * c) / (2 * a * b));
```

现在你几乎有了开始定位物体所需要的所有内容。说成几乎，是因为角B和C其实并不是用于关节的旋转角度。查看图14-8。

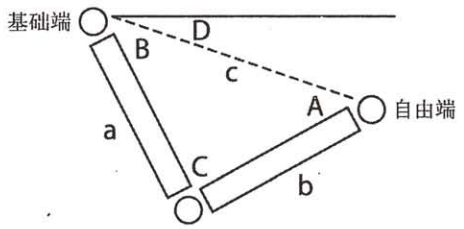


图14-8　计算seg1的旋转

虽然知道了角B，但需要确定是seg1真正旋转了多少度。它距离0（或水平位置）有多远，它可以使用角D加上B来描述。幸运的是，你可以通过计算基础端与自由端之间的角得到角D，如图14-9所示。

368

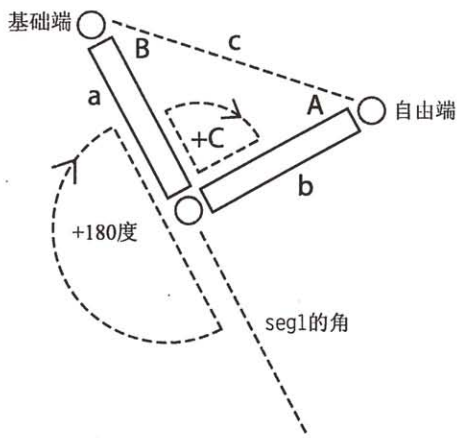


图14-9　计算seg0的旋转

然后你知道了角C，但它只与seg1相关。你所需要的旋转是seg1的旋转加上180度再加上C。我称这个角度为E。

好了，讨论得差不多了，让我们看一下它的代码，它会变得更清晰。

14.4.2　ActionScript 余弦定律

我打算先给出一些反向运动代码，然后再解释它。下面是代码（你也可以在文件Cosines.as

www.TopSage.com

中找到）：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Point;

    public class Cosines extends Sprite
    {
        private var segment0:Segment;
        private var segment1:Segment;

        public function Cosines()
        {
            init();
        }

        private function init():void
        {
            segment0 = new Segment(100, 20);
            addChild(segment0);
            segment1 = new Segment(100, 20);
            addChild(segment1);
            segment1.x = stage.stageWidth / 2;
            segment1.y = stage.stageHeight / 2;

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var dx:Number = mouseX - segment1.x;
            var dy:Number = mouseY - segment1.y;
            var dist:Number = Math.sqrt(dx * dx + dy * dy);

            var a:Number = 100;
            var b:Number = 100;
            var c:Number = Math.min(dist, a + b);

            var B:Number = Math.acos((b * b - a * a - c * c) /
                                     (-2 * a * c));
            var C:Number = Math.acos((c * c - a * a - b * b) /
                                     (-2 * a * b));
            var D:Number = Math.atan2(dy, dx);
            var E:Number = D + B + Math.PI + C;

            segment1.rotation = (D + B) * 180 / Math.PI;

            segment0.x = segment1.getPin().x;
            segment0.y = segment1.getPin().y;
            segment0.rotation = E * 180 / Math.PI;
        }
    }
}
```

369

www.TopSage.com

这里是它的过程：

(1) 计算segment1与鼠标之间的距离。

(2) 获取3条边的长度。边a和b很容易。它们都等于100，因为它是我们创建的关节的长度。（你可以移除掉数值100，使代码变得更加动态一些。我只是想让事情更简单一些。）边c等于dist和a+b中更小的一个。这是因为三角形的一条边不能大于其他两条边的和。如果你不相信我，可以尝试画一个这样的形状。这样又回到了伸展示例。如果从基础端到鼠标的距离是200，但是两
370 个关节的长度和只有120，无法制作这样的例子。

(3) 计算出使用余弦定律所需要的角B和C，那么角D使用Math.atan2来计算。刚才提到的角E等于D+B+180+C。当然，在代码中，你使用Math.PI弧度代替180度。

(4) 像图14-9所示那样，转换弧度角D+B为度，这就是seg1的旋转。使用相同的角度来计算seg1的末端并将seg0定位在上面。

(5) 最后，seg0的旋转是转换为度后的E。

这样你就得到了它：使用余弦定律的反向运动。你可能注意到关节只能以相同的方式弯曲。如果你在创建像只能以一种方式弯曲的肘或膝盖一样的东西，那么这可能是非常好的。

当你像这样分析来计算角度时，有两种方案来解决这个问题：它可以是这样的弯曲方式，或者它可以使用另外一种弯曲方式。你已经通过D加B再加上C对它硬编码来使它向一个方向弯曲，如果减去它们，你得到的将是同样的效果，但是肢体部分可以向另一个方向弯曲。

```
private function onEnterFrame(event:Event):void
{
    var dx:Number = mouseX - segment1.x;
    var dy:Number = mouseY - segment1.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);

    var a:Number = 100;
    var b:Number = 100;
    var c:Number = Math.min(dist, a + b);

    var B:Number = Math.acos((b * b - a * a - c * c) / (-2 * a * c));
    var C:Number = Math.acos((c * c - a * a - b * b) / (-2 * a * b));
    var D:Number = Math.atan2(dy, dx);
    var E:Number = D - B + Math.PI - C;

    segment1.rotation = (D - B) * 180 / Math.PI;

    segment0.x = segment1.getPin().x;
    segment0.y = segment1.getPin().y;
    segment0.rotation = E * 180 / Math.PI;
}
```

如果想让它以任一方式弯曲，你需要解决某种条件逻辑，如，“如果它在这个位置，用这种方式弯曲；否则，按那种方式弯曲。”不幸的是，我只能给你介绍这种余弦定律的方法。但是如果你对这个问题很感兴趣，我确信你可以找到大量关于这个课题的信息。搜索“inverse kinematics”
371 （反向运动）你会得到90 000多个结果。那么你将可以发现某些东西了！

www.TopSage.com

14.5 本章重点公式

对于反向运动的标准形式，使用余弦定律公式。

余弦定律：

$$a^2 = b^2 + c^2 - 2 * b * c * \cos A$$
$$b^2 = a^2 + c^2 - 2 * a * c * \cos B$$
$$c^2 = a^2 + b^2 - 2 * a * b * \cos C$$

ActionScript余弦定律：

```
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

反向运动是一个非常广的课题——远远不是使用一章就可以说完的。尽管如此，我认为本章描述了一些相当酷和有用的内容。你知道了如何设置一个反向运动系统和看它的两种方式：拖动和伸展。我希望至少和你在这个课题能碰撞出一些火花。我所尽力传授的最主要的思想是可以让你用它做出一些有趣的作品来，并且它并不是那么复杂。在Flash中可以制作许多反向运动的内容，我确信你已经准备开始探索它并应用它了。

在下一章，你将进入一个全新的维度，它允许你在影片中加入深度。是的，我们将学习3D。 372

www.TopSage.com

www.TopSage.com

大家网
TopSage.com

Part 4 第四部分

3D 动画

本部分内容

第 15 章　3D 基础
第 16 章　3D 线条和填充
第 17 章　背面剔除和 3D 灯光

www.TopSage.com

第15章

3D基础

本章内容

- 第3维和透视
- 速度和加速度
- 回弹
- 重力
- 折回
- 缓动和弹性
- 坐标旋转
- 碰撞检测
- 本章重点公式

到目前为止你所做的任何内容都只是二维的（有的只是一维的），并且你已经做出了很酷的作品。现在让我们深入到下一个级别。

创建3D图形总是让人激动的。额外的维度使事情看起来更接近我们的生活。怎样在Flash中创建3D已经在无数的图书和教程中介绍过。但我并不打算省掉任何内容，我将快速地讲解基础内容。然后，我将解释在前面的章节所讨论的运动效果如何在三维中实现。具体来说，本章涉及了速度、加速度、摩擦力、重力、回弹、折回、缓动、弹性、坐标旋转和碰撞检测。

现在，你可能最关心的是让一个精灵在3D空间中移动，使用透视来计算它的大小和在屏幕上的位置。当然，精灵本身将是平板形的。它不会有你可以看到的背面、边、顶或底。在下面的两章中，你将会在3D中做一些点、线、面和实体的模型。

15.1 第3维和透视

不言而喻，3D背后的主要概念是在*x*和*y*之外存在着另一维度。这一维度就是深度，它通常被标示为*z*。

Flash并没有内置的*z*维度，但是使用ActionScript创建它并不是很难。它实际上要比你在前面的章节中完成的很多内容要简单得多。

www.TopSage.com

15.1.1 z轴

开始时，你需要决定z轴是哪个方向：进或出。我来解释一下。如果你回忆一下第3章中讨论的Flash坐标系统，可能会记起来它与我们常见的很多其他坐标系统在某些方面是相反的。y轴是向下而不是向上，角度的测量是顺时针的而不是逆时针的。

那么，你需要创建一个z轴以使一个物体的z位置增加，它是远离你还是走向你？哪种方式都不一定比另一种方式更准确（correct）。实际上，这个课题已经被讨论过多次，甚至已经有了描述两种方法的名字：左手系统和右手系统。

伸出你的右手并让你的拇指和食指构成一个L型，然后弯曲你的中指与食指成90度，每一个手指都指向不同的维度。现在将你的食指指向正的x轴方向，中指指向正的y轴方向。在右手系统中，你的拇指指向的方向就是正的z轴方向。对于Flash来说，这意味着当远离当前的视口时z轴将不断增加，而如果走向视口方向则它将不断减小，如图15-1所示。 376

如果试着使用左手，你将得到相反的结果。图15-2显示了左手坐标系统。

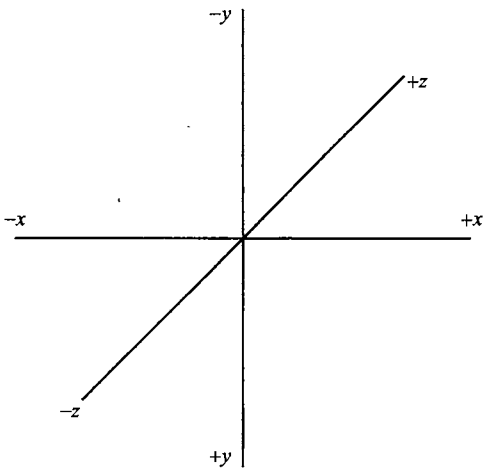


图15-1 右手坐标系统

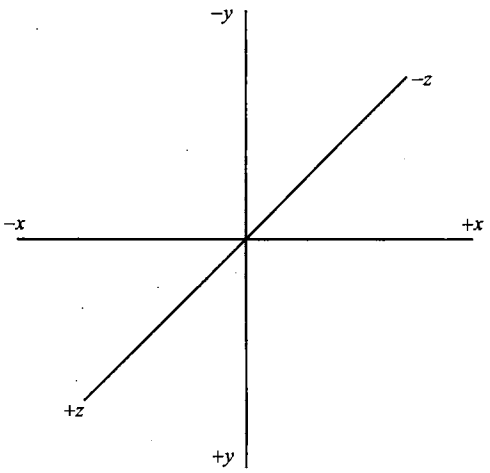


图15-2 左手坐标系统

对于这里的示例我使用的是右手坐标系统（图15-1）。这不是说你不能使用左手系统，但是好像使z轴向里要好一些。

在Flash中创建第3维（z）的下一步就是要解决如何模拟透视。 377

15.1.2 透视

透视就是我们怎样表示某个物体靠近我们或远离我们。或者，换种方式说，我们怎样使某物体看起来它是在近处或远处。而美术领域已经有大量的技术用来描述透视了，这里我将它浓缩为两点：

www.TopSage.com

- 当物体离我们越远时它们变得越小。
- 物体如果离我们足够远它们会聚集在一个灭点上。

我确信你已经看过足够多的火车轨道到达地平线上的一点的例子。因此，基本上当你在z轴上移动某物体时，你需要做两件事情：

- 把它放大或缩小；
- 移动物体靠近或远离灭点。

当只工作在二维环境时，你通常可以使用屏幕的x和y坐标用于你的物体的x和y位置。只需要做一个一对一的映射。在3D中这种方式是不能工作的，因为两个物体可以有相同的x和y位置，然而由于它们的深度，它们在屏幕上有不同的位置。因此，你在3D中移动的每一个物体都需要有它自己的与屏幕位置无关的x、y和z坐标。这些坐标描述的是虚拟空间中的一个位置。透视计算可以告诉你将物体放在屏幕的哪个位置上。

1．透视公式

基本的思路是当某物体离我们越来越远（z轴增加），它的缩放比例趋近于0并且它的x,y位置会聚在灭点的0,0上。好消息是缩放比例和到灭点的距离的比例相同。因此，你只需要计算出对于一个给定的距离它的比例是多少并把它用在两个位置。图15-3可以帮助你解释这个概念。

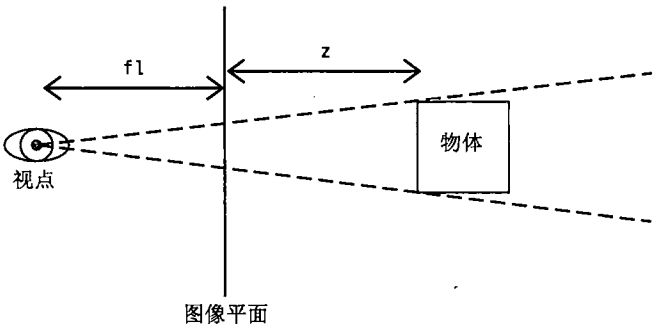


图15-3　从一侧看的透视

你有一个离你有一段距离的物体，并且有一个视点：你的眼睛。你有一个图像平面，你可以认为它是你的电脑屏幕。你有了物体到图像平面的距离，它就是z值，并且，最后你有了从视点到图像点的距离。后者是关键。它并不能完全等于照相机镜头的焦距，但它们是类似的，因此我
378 通常使用fl变量来描述它。这里是公式：

```
scale = fl / (fl + z)
```

它产生的值通常在0.0到1.0之间，它就是缩放和聚为灭点的比例。然而，当z变为负值时，fl+z会趋近于0同时scale会趋近于无穷大。这在ActionScript中等同于你被戳中了眼睛。

那么，你要怎样来处理scale值呢？好，假设你正在处理一个精灵（或Sprite类的子类），你指定这个缩放值给精灵的scaleX和scaleY。然后用这个数值乘以物体的x和y位置来计算它在屏幕上的x和y位置。

www.TopSage.com

让我们来看一个例子。通常fl的值约为200到300。让我们使用250。如果z是0——换句话说，物体正好在图像平面上——那么缩放将是250/(250+0)。这得出的正好是1.0。这就是你的scaleX和scaleY（注意对于scaleX和scaleY来说，1.0意味着100%）。将物体的x和y位置乘以1.0得到的是相同的值作为结果，因此物体的屏幕位置正好等于它原来的x和y。

现在将物体移出去使z是250。这使得scale为250/(250+250)，或scaleX和scaleY为0.5。它也移动了物体在屏幕上的位置。如果物体原来在的x,y位置为200,300，那么它的屏幕位置现在是100,150。它已经被移到了距离灭点一半的位置。（实际上，屏幕位置是与灭点位置相关的，稍后你将会看到。）

现在，沿z方向移动到9 750。这使得scale为250/10 000，或scaleX和scaleY为0.025。物体将变成一个小点，已经非常接近灭点了。

好，已经说了足够多的理论了，是时候说一些代码了。

2. ActionScript透视

可能你已经猜到了，我将再次使用Ball类。当然，你可以将任何东西作为物体使用，但是我把注意力集中在代码上并且把一些酷的图形留给你来做。对于交互，让我们使用有趣的鼠标和键盘。你将使用鼠标来控制小球的x和y位置，使用上和下键来沿着z轴向前和向后移动它。注意因为x和y在Actionscript中已被用作属性，所以我们使用xpos、ypos和zpos来描述3D位置。

这里是文档类Perspective1.as的代码：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
    import flash.ui.Keyboard;

    public class Perspective1 extends Sprite
    {
        private var ball:Ball;
        private var xpos:Number = 0;
        private var ypos:Number = 0;
        private var zpos:Number = 0;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;

        public function Perspective1()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
        }
```

379

www.TopSage.com

大家网 TopSage.com

```
        private function onEnterFrame(event:Event):void
        {
            xpos = mouseX - vpX;
            ypos = mouseY - vpY;
            var scale:Number = fl / (fl + zpos);
            ball.scaleX = ball.scaleY = scale;
            ball.x = vpX + xpos * scale;
            ball.y = vpY + ypos * scale;
        }

        private function onKeyDown(event:KeyboardEvent):void
        {
            if(event.keyCode == Keyboard.UP)
            {
                zpos += 5;
            }
            else if(event.keyCode == Keyboard.DOWN)
            {
                zpos -= 5;
            }
        }
    }
}
```

首先，为xpos、ypos和zpos创建了变量，同样也创建了fl变量。然后创建了一个灭点vpX，
380 vpY。注意当物体越来越远时，它们最终聚合到0,0点。如果你没有按某种方式偏移它，任何东西将聚合在屏幕的左上角，这并不是你想要的。你将使用vpX，vpY来保证灭点在屏幕的中心位置。

接下来，在onEnterFrame方法中，设置xpos和ypos等于鼠标的位置，作为灭点偏移。换句话说，如果鼠标在中心点右边200像素，那么*x*就是200。如果鼠标在中心点左边200像素，那么*x*将是–200。

然后加入一个keyDown事件的侦听器并相应地改变zpos。如果上键被按下，zpos增加，如果下键被按下，zpos就会减小。这将使得小球远离或接近视点移动。

最后，使用我所讲过的公式计算出缩放比例，并依此来设置小球的大小和位置。注意小球在屏幕上的*x*和*y*位置是由灭点加上xpos和ypos乘以scale计算出来的。因此，当scale变得很小时，小球将聚在灭点上。

当测试它时，开始看上去就像是在用鼠标拖动小球。这是因为zpos的值是0，scale为1.0，没有明显的透视被应用。当你按下上键后，小球显示出向远方滑动，如图15-4所示。现在当你四处移动鼠标时，小球也会随着移动，但是移动得会比较少，给了你一个视差的效果。

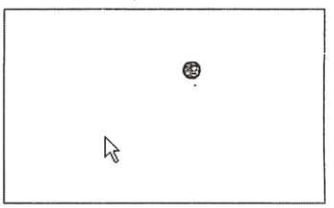


图15-4　透视动作

你可能还注意到一件事情，就是如果你按住下键很长时间，小球开始变得很大。这是正确的，如果你捡起一粒小石子靠近你的眼睛，它看起来就会像一个大的鹅卵石。但是，如果你一直按着下键，你会发现它变到无穷大然后又缩小了，但是现在上下键动作是相反的。那么当小球消失在视点之后发生了什么。我猜如

www.TopSage.com

果你的眼睛能向后看你一定能看到。

从数学上讲，当zpos等于-fl时会发生什么，公式scale = fl/(fl+zpos)变成了scale=fl/0。在许多语言中，除以0会提示你出错。在Flash中，它给出的是Infinity。当你继续减小zpos，你是在用fl除以一个负数。缩放比例就变成了负值，这就是小球产生相反变化的原因。

你怎样来处理它呢？一个简单的方法是如果小球穿过某个点就让它变为不可见的。如果zpos小于或等于-fl，就会存在这样的问题，你可以在Perspective2.as（类的其他部分与Perspective1.as相同）中的enterFrame处理器中测试这个条件并处理小球。

```
private function onEnterFrame(event:Event):void
{
    if(zpos > -fl)
    {
        xpos = mouseX - vpX;
        ypos = mouseY - vpY;
        var scale:Number = 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;
    }
}
```

381

注意如果小球不可视，你不需要担心它的缩放和位置。另外注意如果小球在可视范围内，你需要确保它是可视的。它会增加一点额外的开销，但这是完全必要的。

好，你已经得到了它，最基本的3D。并不是很难，不是吗？确保你已经测试了这个示例以对它有一些好感。特别是试着改变fl的值来看看你会得什么样的不同效果。这等同于改变了照相机的焦距。较大的fl值就像是远距离照相镜头，会给你一个比较小的视野，比较少的可视透视。比较小的fl值会创建一个广角镜头，具有比较夸张的透视。

本章其他的内容将致力于用ActionScript制作各种运动效果，是前面章节的内容，但现在使用的是3D方式。

15.2 速度和加速度

在3D中实现速度和加速度非常容易。对于2D，你使用*vx*和*vy*变量来描述两个轴上的速度。你只需要加入一个*vz*来描述第3维。同样，如果你有像*ax*和*ay*这样的变量来表示加速度，你只需要加入一个*az*变量。

你可以把上一个例子改成小行星型飞船，但是它在3D中。让我们完全使用键盘控制。光标键提供*x*和*y*轴上的驱动力，使用Shift和Ctrl来控制*z*轴上的驱动力。

这里是代码（你也可以在文件Velocity3D.as中找到）：

www.TopSage.com

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
    import flash.ui.Keyboard;

    public class Velocity3D extends Sprite
    {
        private var ball:Ball;
        private var xpos:Number = 0;
        private var ypos:Number = 0;
        private var zpos:Number = 0;
        private var vx:Number = 0;
        private var vy:Number = 0;
        private var vz:Number = 0;
        private var friction:Number =0.98;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;

        public function Velocity3D()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
        }

        private function onEnterFrame(event:Event):void
        {
            xpos += vx;
            ypos += vy;
            zpos += vz;

            vx *= friction;
            vy *= friction;
            vz *= friction;

            if(zpos > -fl)
            {
                var scale:Number = 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;
```

382

www.TopSage.com

```
            }
        }

        private function onKeyDown(event:KeyboardEvent):void
        {
            switch(event.keyCode)
            {
                case Keyboard.UP :
                vy -= 1;
                break;

                case Keyboard.DOWN :
                vy += 1;
                break;

                case Keyboard.LEFT :
                vx -= 1;
                break;

                case Keyboard.RIGHT :
                vx += 1;
                break;

                case Keyboard.SHIFT :
                vz += 1;
                break;

                case Keyboard.CONTROL :
                vz -= 1;
                break;

                default :
                break;
            }
        }
    }
}
```

383

现在你已经为每个轴上的速度加入了变量并加入了一些摩擦力。当这6个键中的一个被按下去时，它会增加或减小速度（注意是加速度在改变速度）。然后将速度加到每个轴的位置上并计算摩擦力。

现在你已经有了一个通过加速度、速度和摩擦力移动的3D物体。哇，一石三鸟。我说过它是很容易的。

15.3 回弹

就本节而言，我将讨论回弹一个平面——换句话说，一个完全对齐x、y或z轴的表面。它类似于你在2D中所做的在屏幕边缘的回弹。

15.3.1 单个物体回弹

对于3D中的回弹，你仍需要检测物体何时穿过边界，调整它的位置以使它靠在边界上，然

www.TopSage.com

后反转它所在的轴上的速度。使用3D的其中一个不同之处在于你怎么样判断边界的位置。在2D中，你通常使用的是舞台的坐标或其他一些可视的矩形区域。在3D中，事情并不是那么简单。并没有真正的可视边界的概念，除非在3D中绘制一个边界。你将在下一章中学习在3D中绘制，所以现在，你将会在一个任意放置的无形的墙上碰撞。

384 那么，你像以前那样设置边界，但是现在要在一个3D空间中设置它们，这意味着它们可能是正的位置或负的位置。你也可以选择性地在z轴上设置边界。边界的设置像这样：

```
private var top:Number = -250;
private var bottom:Number = 250;
private var left:Number = -250;
private var right:Number = 250;
private var front:Number = 250;
private var back:Number = -250;
```

然后，当确定物体的新位置后，你需要针对6个边界进行检测。记住你使用的是物体的宽度的一半来进行碰撞检测，并且已经把该值保存在Ball类的称为radius的属性中。这里是全部的3D回弹代码（可以在Bounce3D.as中找到）：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Bounce3D extends Sprite
    {
        private var ball:Ball;
        private var xpos:Number = 0;
        private var ypos:Number = 0;
        private var zpos:Number = 0;
        private var vx:Number = Math.random() * 10 - 5;
        private var vy:Number = Math.random() * 10 - 5;
        private var vz:Number = Math.random() * 10 - 5;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;
        private var top:Number = -100;
        private var bottom:Number = 100;
        private var left:Number = -100;
        private var right:Number = 100;
        private var front:Number = 100;
        private var back:Number = -100;

        public function Bounce3D()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball(15);
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
385     }
```

www.TopSage.com

TopSage.com

```
private function onEnterFrame(event:Event):void
{
    xpos += vx;
    ypos += vy;
    zpos += vz;

    var radius:Number = ball.radius;
    if(xpos + radius > right)
    {
        xpos = right - radius;
        vx *= -1;
    }
    else if(xpos - radius < left)
    {
        xpos = left + radius;
        vx *= -1;
    }
    if(ypos + radius > bottom)
    {
        ypos = bottom - radius;
        vy *= -1;
    }
    else if(ypos - radius < top)
    {
        ypos = top + radius;
        vy *= -1;
    }
    if(zpos + radius > front)
    {
        zpos = front - radius;
        vz *= -1;
    }
    else if(zpos - radius < back)
    {
        zpos = back + radius;
        vz *= -1;
    }

    if(zpos > -fl)
    {
        var scale:Number = 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;
        }
    }
}
```


www.TopSage.com

注意我已经移除了所有的键盘控制内容并且在每个轴上给了小球一个随机的速度。现在你已经看到它在回弹了，但是不能说出它碰的是哪里——正如我所说的，它们是任意放置的无形的边界。

15.3.2 多物体回弹

要帮助墙可视得更明显一些，有一件事你可以做，那就是使用更多的物体填满空间。要做到这点，我们将需要Ball类的更多实例。但是每一个实例都需要有它自己的xpos、ypos、zpos和每个轴上的速度。在主文档类中跟踪所有的这些内容会比较困难，因此我创建了一个新的类Ball3D，你可以在下面看到：

```
package {
    import flash.display.Sprite;

    public class Ball3D extends Sprite {
        public var radius:Number;
        private var color:uint;
        public var xpos:Number = 0;
        public var ypos:Number = 0;
        public var zpos:Number = 0;
        public var vx:Number = 0;
        public var vy:Number = 0;
        public var vz:Number = 0;
        public var mass:Number = 1;

        public function Ball3D(radius:Number=40, color:uint=0xff0000) {
            this.radius = radius;
            this.color = color;
            init();
        }
        public function init():void {
            graphics.beginFill(color);
            graphics.drawCircle(0, 0, radius);
            graphics.endFill();
        }
    }
}
```

如你所看到的，这段代码所做的是加入了每一个轴上的位置和速度属性。另外，对于面向对象编程来说，把属性以公开的形式放在类中是一个非常坏的习惯，但是现在我们只是为了保持简单，是为了演示公式。在MultiBounce3D.as中，我已经创建了50个该类的新的实例。每一个实例都有它自己的xpos、ypos和zpos以及vx、vy、vz。onEnterFrame方法通过循环，取得每一个Ball3D实例的引用，然后传递给move方法。move方法对每个小球所做的事情与原来的onEnterFrame代码对每个小球所做的是一样的。这里是代码（你可以在MultiBounce3D.as中找到）：

387

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class MultiBounce3D extends Sprite
    {
```

www.TopSage.com

```
private var balls:Array;
private var numBalls:uint = 50;
private var fl:Number = 250;
private var vpX:Number = stage.stageWidth / 2;
private var vpY:Number = stage.stageHeight / 2;
private var top:Number = -100;
private var bottom:Number = 100;
private var left:Number = -100;
private var right:Number = 100;
private var front:Number = 100;
private var back:Number = -100;

public function MultiBounce3D()
{
    init();
}

private function init():void
{
    balls = new Array();
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball3D = new Ball3D(15);
        balls.push(ball);
        ball.vx = Math.random() * 10 - 5;
        ball.vy = Math.random() * 10 - 5;
        ball.vz = Math.random() * 10 - 5;
        addChild(ball);
    }
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball3D = balls[i];
        move(ball);
    }
}
private function move(ball:Ball3D):void
{
    var radius:Number =  ball.radius;

    ball.xpos += ball.vx;
    ball.ypos += ball.vy;
    ball.zpos += ball.vz;

    if(ball.xpos + radius > right)
    {
        ball.xpos = right - radius;
        ball.vx *= -1;
    }
    else if(ball.xpos - radius < left)
```

388

www.TopSage.com

```
            {
                ball.xpos = left + radius;
                ball.vx *= -1;
            }
            if(ball.ypos + radius > bottom)
            {
                ball.ypos = bottom - radius;
                ball.vy *= -1;
            }
            else if(ball.ypos - radius < top)
            {
                ball.ypos = top + radius;
                ball.vy *= -1;
            }
            if(ball.zpos + radius > front)
            {
                ball.zpos = front - radius;
                ball.vz *= -1;
            }
            else if(ball.zpos - radius < back)
            {
                ball.zpos = back + radius;
                ball.vz *= -1;
            }

            if(ball.zpos > -fl)
            {
                var scale:Number = 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;
            }
        }
    }
}
```

389

运行这个文件时，小球在很大程度上填满了6个边界组成的空间，如图15-5所示，你可以感觉到这个空间的形状。

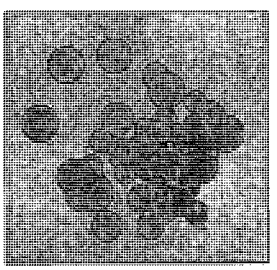


图15-5　3D小球回弹

www.TopSage.com

15.3.3 z轴排序

现在，加入多个物体后你的代码还缺少一项内容——称为z轴排序（z-sorting）。z轴排序的意思跟它听起来的差不多：物体在z轴上是如何排列的，或哪一个物体应当在另一个物体的前面。实际上现在看起来还不是很明显，原因是物体使用了实色。要看得明显一些，改变一下Ball3D的init方法，然后再运行上一个示例。其改动如下：

```
public function init():void {
    graphics.lineStyle(0);
    graphics.beginFill(color);
    graphics.drawCircle(0, 0, radius);
    graphics.endFill();
}
```

通过给小球加入描边，你可以看清楚哪一个小球在另一个小球的上方。当小的物体显示在大的物体前面时，它严重破坏了3D效果。z轴排序可以处理这些，但它并不是自动的。Flash并不知道你在模拟3D。它只知道你在移动和缩放精灵。当然它也不知道你正在使用的是左手系统还是右手系统。因此，它并不知道较远的小球应该出现在较近的另一个小球的后面。它只是根据它们在显示列表中的索引来排列它们。 390

在AS 2中，z轴排序很容易通过使用movieclip.swapDepths(depth)改变影片剪辑的深度来控制。有较高的深度的影片剪辑将排列在较低深度的影片剪辑的前面。

然而，在AS 3中，事情有一点复杂。显示对象并没有专门的深度值可以任意设置。显示列表的工作更像是一个数组。列表中的每个显示对象都有一个索引。这个索引从0开始，并增加到列表中的对象的个数数值。例如，假设你加入了3个精灵A、B和C到你的类中。它们的索引为0、1和2。对于实例来说，没有办法将其中一个放在索引是100或是–100的位置。更进一步，如果你移除了精灵B，精灵A和C的索引将变成了0和1。你现在应该明白了。在显示列表中没有任何的“空间”。

就深度而言，索引0是最底端，并且任何有较高索引值的显示对象将出现在较低索引值的显示对象前面。你可以用几种其他的方法来改变显示对象的深度。

- setChildIndex(child:DisplayObject,index:int)移动指定的对象到指定的索引。简单。
- swapChildren(child1:DisplayObject,child2:DisplayObject)交换两个指定的对象。
- swapChildrenAt(index1:int,index2:int)交换具有指定的索引的两个对象。

最简单的方法是setChildIndex。你已经将所有小球放在了数组中。如果你能从最高的深度（最远）到最低的深度（最近）对数组进行排序，那么你可以快速浏览一下并且为每一个小球指定索引为从0（最远）到49（最近）。下面的方法完成该操作：

```
private function sortZ():void
{
    balls.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball3D = balls[i];
        setChildIndex(ball, i);
    }
}
```

www.TopSage.com

数组的排序是基于数组中每一个对象的zpos属性。因为它指定了Array.DESCENDING和Array.NUMERIC，它将按逆序排序——换句话说，从高到低排序。结果是最远的小球（最高的zpos值）将成为数组中的第1个元素，那么最近的小球将是最后1个。

391 然后循环这个数组，将每个小球的索引设置为它们现在在数组中的顺序。

一旦这个方法存在于类中，你只需要在移动所有的小球之后调用它，它刚好在onEnterFrame方法的最后：

```
private function onEnterFrame(event:Event):void
{
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball3D = balls[i];
        move(ball);
    }
    sortZ();
}
```

其他的代码与上一个示例的一样。所有的代码可以在Zsort.as中找到。

15.4 重力

这里我所说的只是来自地球表面的简单的重力，如第5章中描述的那样。在这种情况下，重力在3D中的工作过程与2D中非常相似。所有你要做的就是选择一个作用在物体身上的重力数值，然后在每一帧或每次迭代中将这个数值加到物体的*y*轴速度上。

既然重力在3D中是如此简单，我本可以把它掩盖过去并且说，“它跟2D中的一样，好，进入下一个主题。”但我决定使用一个非常漂亮的例子（像3D烟花）来演示创建一个非常棒的效果是多么地简单。

开始时，你需要用一个对象来表示一个“烟花”——你知道，可以和其他的小点组合起来形成一个很大的爆炸的一个发光的小点。我们可以信任的Ball3D类，如果使用很小的半径，刚好可以达到这个目的。我们将为每个小点创建随机的颜色以产生漂亮的效果。如果将你的影片背景设置为黑色也是有帮助的。我在类中使用了SWF元数据语句，但是如果你使用的是Flash CS3 IDE，你只需要在文档属性中设置它的背景色。

目前为止我相信你可以操作它，所以我给出了全部的代码（Fireworks.as），然后解释它。

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    [SWF(backgroundColor=0x000000)]
    public class Fireworks extends Sprite
    {
        private var balls:Array;
        private var numBalls:uint = 100;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
392     private var vpY:Number = stage.stageHeight / 2;
```

www.TopSage.com

```
private var gravity:Number = 0.2;
private var floor:Number = 200;
private var bounce:Number = -0.6;

public function Fireworks()
{
    init();
}

private function init():void
{
    balls = new Array();
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball3D = new Ball3D(3, Math.random() *
                                        0xffffff);
        balls.push(ball);
        ball.ypos = -100;
        ball.vx = Math.random() * 6 - 3;
        ball.vy = Math.random() * 6 - 6;
        ball.vz = Math.random() * 6 - 3;
        addChild(ball);
    }
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball3D = balls[i];
        move(ball);
    }
    sortZ();
}

private function move(ball:Ball3D):void
{
    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:Number = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
```

393

www.TopSage.com

```
                ball.visible = true;
            }
            else
            {
                ball.visible = false;
            }
        }

        private function sortZ():void
        {
            balls.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball3D = balls[i];
                setChildIndex(ball, i);
            }
        }
    }
}
```

首先，加入一些属性：gravity bounce和floor。前两个你已经知道了。floor属性是物体可以在回弹之前碰到的最底端的*y*值。

除了加入重力到每个小球的vy属性之外，还有每个小球碰到地板时的回弹，所有这些我们都涉及过，但是它看上去很酷，不是吗？

394 结果如图15-6所示。



图15-6　烟花（相信我，它的动画效果非常好）

15.5　折回

在第6章中，我讨论过当物体碰到边界时会有3种可能的反应。我刚刚讲了回弹。还有折回和重新生成。对于3D，我发现折回是非常有用的，但只是在*z*轴上。

在2D折回中，你检测物体是否在*x*或*y*轴上离开了屏幕。它会工作得很好，因为当物体超出某

www.TopSage.com

个边界时，你就不能再看到它了，因此，你可以很容易地在观众不注意的时候重新定位它。在3D中你却不能这样做。

使用3D，实际上只有两个位置可以安全地移除和重定位一个物体。一个是当物体移动到视点之后。在前面的例子中你已经测试过它并且在这种情况下你将物体设置成了不可视。另一个是当物体距目前位置很远并且缩为很小的一点以致它是不可见的或几乎不可见。这意味着你可以安全地在*z*轴上折回。当某个东西移到你的视点后面时，你要把它放回它原来的位置，让它重新向你的方向移动。或者，如果某个东西已经离你很远很远几乎看不见了，移除它并将它放在你的视点后面。如果你想，你也可以试着在*x*或*y*轴上折回，但在大多数情况下，你将会得到一些不自然的进或出的效果。

好的消息是*z*轴折回是很有用的。我已经使用它创建了真实的3D赛车类游戏，在这里我将重新创建它的一部分。

思路是在视点的前方放置各种3D物体。然后让这些物体向视点移动。换句话说，你给它们一些负的*z*轴速度。根据你对它的设置，这或者看起来像是许多物体在向你移动或者它可以欺骗眼睛使它看起来你正在向物体移动。一旦一个物体移到了视点的后方，你将它重新放置在远方。这样，就会有源源不断的物体在行进。 395

在这个示例中我使用的物体是简单风格的树，使用随机的树枝位置来创建一个基本的树结构。我相信你可以做得更好！

绘制树的代码在名为Tree的类中，如你在下面看到的，它只有3个位置属性和绘制一些树枝的代码。

```
package {
    import flash.display.Sprite;

    public class Tree extends Sprite {
        public var xpos:Number = 0;
        public var ypos:Number = 0;
        public var zpos:Number = 0;

        public function Tree() {
            init();
        }
        public function init():void {
            graphics.lineStyle(0, 0xffffff);
            graphics.lineTo(0, -140 - Math.random() * 20);
            graphics.moveTo(0, -30 - Math.random() * 30);
            graphics.lineTo(Math.random() * 80 - 40,
                            -100 - Math.random() * 40);
            graphics.moveTo(0, -60 - Math.random() * 40);
            graphics.lineTo(Math.random() * 60 - 30,
                            -110 - Math.random() * 20);
        }
    }
}
```

同样，我在黑色背景上使用了白色，对于背景色使用了SWF元数据。但是你可以制作任意类

www.TopSage.com

型的物体，并且可以是你想要的复杂程度。文档类只是创建了所有这些树（100棵树）。它们在x轴上随机展开着，向任一方向伸展了10 000像素。它们也可以在z轴上伸展，从0到10 000。但是，它们都有着相同的y位置，基于的是floor属性，它给人的是一个接地层的感觉。

这里是代码（你可以在Trees.as中找到）：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
    import flash.ui.Keyboard;

    [SWF(backgroundColor=0x000000)]
    public class Trees extends Sprite
    {
        private var trees:Array;
        private var numTrees:uint = 100;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;
        private var floor:Number = 50;
        private var vz:Number = 0;
        private var friction:Number = 0.98;

        public function Trees()
        {
            init();
        }

        private function init():void
        {
            trees = new Array();
            for(var i:uint = 0; i < numTrees; i++)
            {
                var tree:Tree = new Tree();
                trees.push(tree);
                tree.xpos = Math.random() * 2000 - 1000;
                tree.ypos = floor;
                tree.zpos = Math.random() * 10000;
                addChild(tree);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
        }

        private function onEnterFrame(event:Event):void
        {
            for(var i:uint = 0; i < numTrees; i++)
            {
                var tree:Tree = trees[i];
                move(tree);
            }
```

396

www.TopSage.com

TopSage.com

```
            vz *= friction;
            sortZ();
        }

        private function onKeyDown(event:KeyboardEvent):void
        {
            if(event.keyCode == Keyboard.UP)
            {
                vz -= 1;
            }
            else if(event.keyCode == Keyboard.DOWN)
            {
                vz += 1;
            }
        }
        private function move(tree:Tree):void
        {
            tree.zpos += vz;

            if(tree.zpos < -fl)
            {
                tree.zpos += 10000;
            }
            if(tree.zpos > 10000 - fl)
            {
                tree.zpos -= 10000;
            }

            var scale:Number = fl / (fl + tree.zpos);
            tree.scaleX = tree.scaleY = scale;
            tree.x = vpX + tree.xpos * scale;
            tree.y = vpY + tree.ypos * scale;
            tree.alpha = scale;
        }

        private function sortZ():void
        {
            trees.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
            for(var i:uint = 0; i < numTrees; i++)
            {
                var tree:Tree = trees[i];
                setChildIndex(tree, i);
            }
        }
    }
}
```

397

注意到现在只有一个表示*z*速度的变量，所以树不能在*x*或*y*轴上移动，所有的树将统一沿着*z*轴移动。在onEnterFrame方法中，检测被按下的上下光标键并相应地增加或减小vz的值。一点点摩擦力可以防止速度增加到无穷大，并且在没有键按下时也会慢下来。

代码循环每一棵树，使用当前的*z*速度更新它的*z*轴位置。然后检测树是否已经移到了你的后方。如果是这样，不要将树设置为不可视，而是要将它移动到*z*轴深处10 000像素的位置上。同样，

如果它超过了10 000–f1，将它移回10 000像素的位置。

然后你进行标准的透视动作。这里，我还加入了一点可以加强深度效果的内容：

```
tree.alpha = scale;
```

它设置了与z轴深度相关的树的透明度。树越远，它的透明度就越小。这就是大气环境透视，
398 模拟了观察者与物体之间的大气效果。当你像在本例中一样将物体移向远方时它是特别有效的。

这种特殊的计算可以得到黑暗诡异的效果。你可能想试一下下面的代码：

```
tree.alpha = scale * .7 + .3;
```

这将给树至少30%的透明度。而不是雾茫茫的。这里面基本没有正确或错误的数值——只是会创建出不同的效果的不同数值。

你可能还注意到我把z轴排序方法保留在那里了。在这个特殊的例子中，它在视觉上并没有很大的不同，因为这些树只是具有相同颜色的一些简单的线条，但是如果你要绘制更复杂、有重叠的物体，那么它的效果会很明显的。你可以在图15-7中看到这个文件的结果。



图15-7　注意这些树

我非常喜欢这个示例，所以我添加了折回课题之外的一些改善，这样做只是提示你一些还能对它做些什么。这里是结果（你可以在Trees2.as中找到）：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
399  import flash.ui.Keyboard;

    [SWF(backgroundColor=0x000000)]
    public class Trees2 extends Sprite
    {
        private var trees:Array;
        private var numTrees:uint = 100;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;
        private var floor:Number = 50;
```

www.TopSage.com

```
private var ax:Number = 0;
private var ay:Number = 0;
private var az:Number = 0;
private var vx:Number = 0;
private var vy:Number = 0;
private var vz:Number = 0;
private var gravity:Number = 0.3;
private var friction:Number = 0.98;

public function Trees2()
{
    init();
}

private function init():void
{
    trees = new Array();
    for(var i:uint = 0; i < numTrees; i++)
    {
        var tree:Tree = new Tree();
        trees.push(tree);
        tree.xpos = Math.random() * 2000 - 1000;
        tree.ypos = floor;
        tree.zpos = Math.random() * 10000;
        addChild(tree);
    }
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
    stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
}

private function onEnterFrame(event:Event):void
{
    vx += ax;
    vy += ay;
    vz += az;
    vy -= gravity;
    for(var i:uint = 0; i < numTrees; i++)
    {
        var tree:Tree = trees[i];
        move(tree);
    }
    vx *= friction;
    vy *= friction;
    vz *= friction;
    sortZ();
}

private function onKeyDown(event:KeyboardEvent):void
{
    switch(event.keyCode)
    {
        case Keyboard.UP :
        az = -1;
```

400

www.TopSage.com

```
            break;

            case Keyboard.DOWN :
            az = 1;
            break;

            case Keyboard.LEFT :
            ax = 1;
            break;

            case Keyboard.RIGHT :
            ax = -1;
            break;

            case Keyboard.SPACE :
            ay = 1;
            break;

            default :
            break;
        }
    }

    private function onKeyUp(event:KeyboardEvent):void
    {
        switch(event.keyCode)
        {
            case Keyboard.UP :
            case Keyboard.DOWN :
            az = 0;
            break;

            case Keyboard.LEFT :
            case Keyboard.RIGHT :
            ax = 0;
            break;

            case Keyboard.SPACE :
            ay = 0;
            break;

            default :
            break;
        }
    }

    private function move(tree:Tree):void
    {
        tree.xpos += vx;
        tree.ypos += vy;
        tree.zpos += vz;

        if(tree.ypos < floor)
        {
            tree.ypos = floor;
```

401

www.TopSage.com

```
            }

            if(tree.zpos < -fl)
            {
                tree.zpos += 10000;
            }
            if(tree.zpos > 10000 - fl)
            {
                tree.zpos -= 10000;
            }

            var scale:Number = fl / (fl + tree.zpos);
            tree.scaleX = tree.scaleY = scale;
            tree.x = vpX + tree.xpos * scale;
            tree.y = vpY + tree.ypos * scale;
            tree.alpha = scale;
        }

        private function sortZ():void
        {
            trees.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
            for(var i:uint = 0; i < numTrees; i++)
            {
                var tree:Tree = trees[i];
                setChildIndex(tree, i);
            }
        }
    }
}
```

402

这里，我已经加入了x和y轴上的速度，还有一些重力。我也不得不做一些工作以能够获取到多个键。我很怀念AS 2中的key.isDown()方法，你可以在任何时候调用它来检测是否有指定的键被按下。因为在AS 3中我们只能知道最后被按下或松开的键，所以我必须检测哪个键已经被按下并且设置对应轴上的加速度为 1 或–1。那么当那个键被松开时，我将加速度设置回0。每个轴上的加速度在onEnterFrame开始时加入到那个轴的速度上。左右光标键很显然是控制x速度，我使用了空格键用于y轴。有趣的一点是我实际上从vy中减去了重力。这是因为我想让它看起来就像观众下落在树林中，如图15-8所示。实际上，是树在“落向”视点的位置，但它看起来是一样的。还要注意我限制了树的y轴位置为50，这使它看起来像是你已经着地了。



图15-8 看，我正在飞

www.TopSage.com

大家网 TopSage.com

我并没对x轴上的移动做任何的限制，这意味着你可以看到树林的侧面。设置一些限制对你来说并不难，但我认为这里我已经做了足够多的东西，你可以开始了。

15.6 缓动和弹性

缓动和弹性在3D中也不会比它们在2D（第8章中的内容）中更复杂。你主要需要加入一两个针对z轴的变量，这就是你要做的全部设置。

15.6.1 缓动

这里并不是要说关于缓动的全部内容。在2D中，你已经有了tx和ty作为一个目标点。你只需要为z轴加入tz。在每一帧上，你检测物体到每一个轴的目标点的距离，然后将它移动这个距离的一部分。

403

让我们看一个简单的、缓动物体到一个随机目标点的例子，当小球到那儿时，它拾取另外一个点作为目标点并移动物体到那里。注意在下面的两个例子中我重新使用了Ball3D类。这里是代码（也可以在Easing3D.as中找到）：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Easing3D extends Sprite
    {
        private var ball:Ball3D;
        private var tx:Number;
        private var ty:Number;
        private var tz:Number;
        private var easing:Number = .1;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;

        public function Easing3D()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball3D();
            addChild(ball);

            tx = Math.random() * 500 - 250;
            ty = Math.random() * 500 - 250;
            tz = Math.random() * 500

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
```

www.TopSage.com

```
        {
            var dx:Number = tx - ball.xpos;
            var dy:Number = ty - ball.ypos;
            var dz:Number = tz - ball.zpos;
            ball.xpos += dx * easing;
            ball.ypos += dy * easing;
            ball.zpos += dz * easing;

            var dist:Number = Math.sqrt(dx*dx + dy*dy + dz*dz);

            if(dist < 1)
            {
                tx = Math.random() * 500 - 250;
                ty = Math.random() * 500 - 250;
                tz = Math.random() * 500
            }

            if(ball.zpos > -fl)
            {
                var scale:Number = 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;
            }
        }
    }
}
```

404

最有趣的是下面的这行代码：

```
var dist:Number = Math.sqrt(dx*dx + dy*dy + dz*dz);
```

如果你有印象，在2D中，你计算两个点之间的距离是使用下面的等式：

```
var dist:Number = Math.sqrt(dx*dx + dy*dy);
```

要移动3D空间中的距离，只需要加入第3个轴上的距离的平方。这对我来说是很简单的。它就像是我需要使用3次方根而不是平方根，现在我已经加入了这额外的一项。但它并没有像3次方根那样工作。

15.6.2 弹性

弹性，作为缓动的近亲，在3D中需要相似的调整。你只需使用到目标点的距离来改变速度，而不是位置。我将给你另外一个例子。在这个例子中(Spring3D.as)，点击鼠标将创建一个新的随机目标点以使小球可以弹到那儿。

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
```

www.TopSage.com

```
import flash.events.MouseEvent;

public class Spring3D extends Sprite
{
    private var ball:Ball3D;
    private var tx:Number;
    private var ty:Number;
    private var tz:Number;
    private var spring:Number =0.1;
    private var friction:Number =0.94;
    private var fl:Number = 250;
    private var vpX:Number = stage.stageWidth / 2;
    private var vpY:Number = stage.stageHeight / 2;

    public function Spring3D()
    {
        init();
    }

    private function init():void
    {
        ball = new Ball3D();
        addChild(ball);

        tx = Math.random() * 500 - 250;
        ty = Math.random() * 500 - 250;
        tz = Math.random() * 500

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
        stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    }

    private function onEnterFrame(event:Event):void
    {
        var dx:Number = tx - ball.xpos;
        var dy:Number = ty - ball.ypos;
        var dz:Number = 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:Number = fl / (fl + ball.zpos);
            ball.scaleX = ball.scaleY = scale;
            ball.x = vpX + ball.xpos * scale;
            ball.y = vpY + ball.ypos * scale;
            ball.visible = true;
```

405

www.TopSage.com

406

```
            }
            else
            {
                ball.visible = false;
            }
        }

        private function onMouseDown(event:MouseEvent):void
        {
            tx = Math.random() * 500 - 250;
            ty = Math.random() * 500 - 250;
            tz = Math.random() * 500
        }
    }
}
```

如你所看到的，它使用的是基本的弹性公式（来自第8章），只是带有第3个轴。

15.7 坐标旋转

接下来是在3D中的坐标旋转。它确实要比第10章和第11章讲的2D内容复杂一些，你不仅可以在3个轴之间选择一个轴来旋转，你甚至可以同时在2个或3个轴上旋转。

在2D坐标旋转中，那些点是围绕*z*轴旋转的，如图15-9所示。想象一下围绕着中心轴旋转的幸运轮盘（Wheel of Fortune）。它的中心轴就是*z*轴。只有*x*和*y*的坐标在改变。

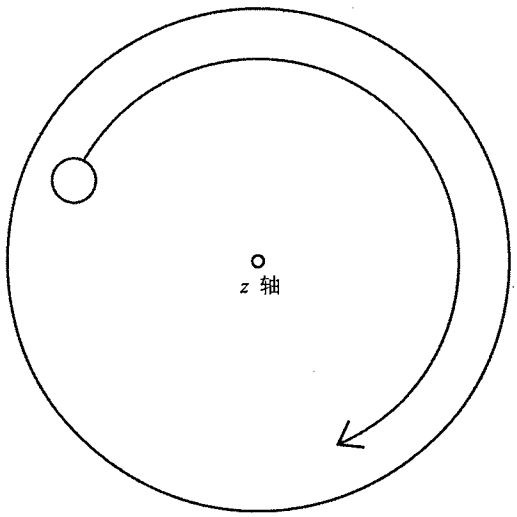


图15-9 围绕*z*轴旋转

407

在3D中，你也可以围绕着*x*或*y*轴旋转。一个*x*轴旋转就像是车轮在向你滚动过来，如图15-10所示。旋转轴是*x*轴。所有点都围绕它旋转并改变它们的*y*和*z*的位置。

www.TopSage.com

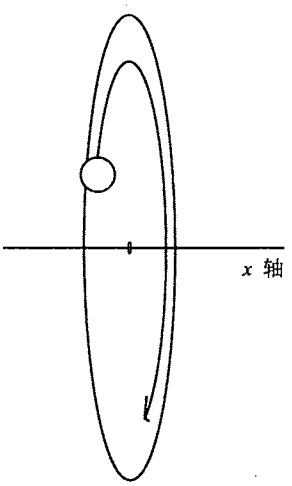


图15-10　围绕x轴旋转

对于y轴旋转，想象为一个老式的留声机，如图15-11。转轴是y轴。所有点在x和z轴上改变。

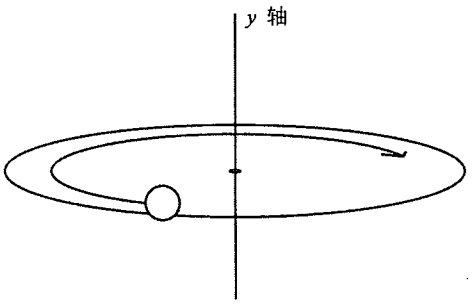


图15-11　围绕y轴旋转

因此，对于3D，当你在一个轴上旋转物体时，它的位置将在其他两个轴上发生改变。

如果你回到第10章，会找到下面的2D旋转公式：

```
x1 = cos(angle) * x - sin(angle) * y;
y1 = cos(angle) * y + sin(angle) * x;
```

408

在3D中，基本上是一样的，但是你需要指定角度是关于哪个轴的：x、y或z。因此，得到下面的3个公式：

```
x1 = cos(angleZ) * x - sin(angleZ) * y;
y1 = cos(angleZ) * y + sin(angleZ) * x;

x1 = cos(angleY) * x - sin(angleY) * z;
z1 = cos(angleY) * z + sin(angleY) * x;

y1 = cos(angleX) * y - sin(angleX) * z;
z1 = cos(angleX) * z + sin(angleX) * y;
```

www.TopSage.com

让我们试一下y轴旋转。下面的代码可以在RotateY.as中找到。它创建了50个Ball3D实例并随机放置它们。然后它获取了一个基于鼠标x位置的y角度。鼠标向右移动得距离越大，角度值就越高。这使得物体看起来像是跟随着鼠标旋转。

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class RotateY extends Sprite
    {
        private var balls:Array;
        private var numBalls:uint = 50;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;

        public function RotateY()
        {
            init();
        }

        private function init():void
        {
            balls = new Array();
            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball3D = new Ball3D(15);
                balls.push(ball);
                ball.xpos = Math.random() * 200 - 100;
                ball.ypos = Math.random() * 200 - 100;
                ball.zpos = Math.random() * 200 - 100;
                addChild(ball);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            var angleY:Number = (mouseX - vpX) * .001;
            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball3D = balls[i];
                rotateY(ball, angleY);
            }
            sortZ();
        }

        private function rotateY(ball:Ball3D, angleY:Number):void
        {
            var cosY:Number = Math.cos(angleY);
            var sinY:Number = Math.sin(angleY);

            var x1:Number = ball.xpos * cosY - ball.zpos * sinY;
            var z1:Number = ball.zpos * cosY + ball.xpos * sinY;
```

409

www.TopSage.com

```
                ball.xpos = x1;
                ball.zpos = z1;

                if(ball.zpos > -fl)
                {
                    var scale:Number = 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;
                }
            }

            private function sortZ():void
            {
                balls.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
                for(var i:uint = 0; i < numBalls; i++)
                {
                    var ball:Ball3D = balls[i];
                    setChildIndex(ball, i);
                }
            }
        }
410     }
```

重要的部分是粗体部分。你获取了一个角度然后针对第1个小球使用这个角度调用rotateY方法。在这个方法中，计算出这个角的正弦和余弦，接着做旋转，然后将x1和z1赋回给ball.xpos和ball.zpos。之后，只是标准的透视和z轴排序。图15-12显示了结果。

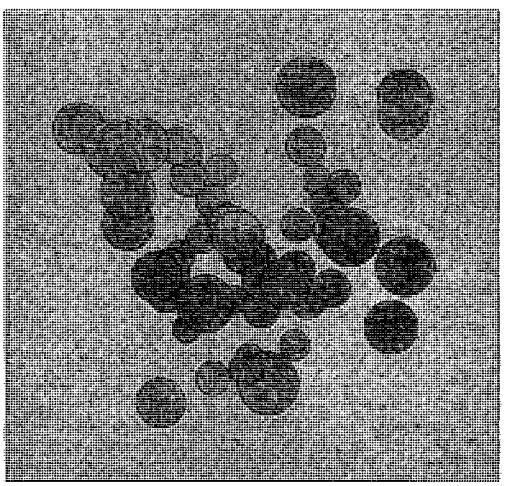


图15-12　围绕y轴旋转

www.TopSage.com

一旦你试验了这个例子，你可以试着把它切换成围绕*x*轴旋转。只需要改变onEnterFrame方法并加入一个rotateX方法：

```
private function onEnterFrame(event:Event):void
{
    var angleX:Number = (mouseY - vpY) * .001;
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball3D = balls[i];
        rotateX(ball, angleX);
    }
    sortZ();
}

private function rotateX(ball:Ball3D, angleX:Number):void
{
    var cosX:Number = Math.cos(angleX);
    var sinX:Number = Math.sin(angleX);

    var y1:Number = ball.ypos * cosX - ball.zpos * sinX;
    var z1:Number = ball.zpos * cosX + ball.ypos * sinX;

    ball.ypos = y1;
    ball.zpos = z1;
    if(ball.zpos > -fl)
    {
        var scale:Number = 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;
    }
}
```

411

现在，angleX是基于鼠标的*y*位置。计算出这个角的正弦和余弦，使用它们获得y1和z1并传递给小球的ypos和zpos属性。

接下来，让我们合并这两个旋转。这里是代码（你也可以在RotateXY.as中找到）：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class RotateXY extends Sprite
    {
        private var balls:Array;
        private var numBalls:uint = 50;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;
```

www.TopSage.com

```
public function RotateXY()
{
    init();
}

private function init():void
{
    balls = new Array();
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball3D = new Ball3D(15);
        balls.push(ball);
        ball.xpos = Math.random() * 200 - 100;
        ball.ypos = Math.random() * 200 - 100;
        ball.zpos = Math.random() * 200 - 100;
        addChild(ball);
    }
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
private function onEnterFrame(event:Event):void
{
    var angleX:Number = (mouseY - vpY) * .001;
    var angleY:Number = (mouseX - vpX) * .001;
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball3D = balls[i];
        rotateX(ball, angleX);
        rotateY(ball, angleY);
        doPerspective(ball);
    }
    sortZ();
}

private function rotateX(ball:Ball3D, angleX:Number):void
{
    var cosX:Number = Math.cos(angleX);
    var sinX:Number = Math.sin(angleX);

    var y1:Number = ball.ypos * cosX - ball.zpos * sinX;
    var z1:Number = ball.zpos * cosX + ball.ypos * sinX;

    ball.ypos = y1;
    ball.zpos = z1;
}

private function rotateY(ball:Ball3D, angleY:Number):void
{
    var cosY:Number = Math.cos(angleY);
    var sinY:Number = Math.sin(angleY);

    var x1:Number = ball.xpos * cosY - ball.zpos * sinY;
    var z1:Number = ball.zpos * cosY + ball.xpos * sinY;
```

412

www.TopSage.com

```
            ball.xpos = x1;
            ball.zpos = z1;
        }

        private function doPerspective(ball:Ball3D):void
        {
            if(ball.zpos > -fl)
            {
                var scale:Number = 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;
            }
        }

        private function sortZ():void
        {
            balls.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball3D = balls[i];
                setChildIndex(ball, i);
            }
        }
    }
}
```

413

与之前示例相比主要的变化是用粗体表示的。现在，你计算出了angleY和angleX，然后对每一个小球调用rotateX和rotateY。注意我已经将透视代码从rotate方法中移出，移到它自己的一个方法中，否则它要被调用两次。我确信你可以根据你刚学过的内容和前面的公式很容易地加入rotateZ方法。

测试这个示例。通过将3D坐标旋转与15.5节的赛车游戏示例中的一些概念相结合，你可以创建出某些丰富的交互式3D环境。

15.8 碰撞检测

最后在3D中要介绍的是碰撞检测。在3D中进行碰撞检测的唯一可行的方法是基于距离的碰撞检测。它与在2D中的碰撞检测没有太大区别。你找到两个物体之间的距离（使用3D距离公式），如果该距离小于它们两者半径的和，则发生了碰撞。

要制作一个3D碰撞检测的示例，我修改了之前的一个3D回弹的例子，给它更少的物体和更多的空间。在代码中，首先做的是标准的3D运动和透视，然后使用一个嵌套的for循环对比所有小球的位置。如果有两者距离小于两者半径和的情况，使用颜色变换代码将它们两个标示为蓝色。非常简单。这里是代码(Collision3D.as)：

www.TopSage.com

大家网 TopSage.com

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.ColorTransform;

    public class Collision3D extends Sprite
    {
        private var balls:Array;
414     private var numBalls:uint = 20;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;
        private var top:Number = -200;
        private var bottom:Number = 200;
        private var left:Number = -200;
        private var right:Number = 200;
        private var front:Number = 200;
        private var back:Number = -200;

        public function Collision3D()
        {
            init();
        }

        private function init():void
        {
            balls = new Array();
            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball3D = new Ball3D(15);
                balls.push(ball);
                ball.xpos = Math.random() * 400 - 200;
                ball.ypos = Math.random() * 400 - 200;
                ball.zpos = Math.random() * 400 - 200;
                ball.vx = Math.random() * 10 - 5;
                ball.vy = Math.random() * 10 - 5;
                ball.vz = Math.random() * 10 - 5;
                addChild(ball);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball3D = balls[i];
                move(ball);
            }
            for(i = 0; i < numBalls - 1; i++)
            {
                var ballA:Ball3D = balls[i];
                for(var j:uint = i + 1; j < numBalls; j++)
                {
```

www.TopSage.com

```
        {
            var ballB:Ball3D = balls[j];
            var dx:Number = ballA.xpos - ballB.xpos;
            var dy:Number = ballA.ypos - ballB.ypos;
            var dz:Number = ballA.zpos - ballB.zpos;
            var dist:Number = Math.sqrt(dx*dx + dy*dy + dz*dz);
            if(dist < ballA.radius + ballB.radius)
            {
                var blueTransform:ColorTransform =
                    new ColorTransform(0, 1, 1, 1,
                                       0, 0, 255, 0);
                ballA.transform.colorTransform = blueTransform;
                ballB.transform.colorTransform = blueTransform;
            }
        }
    }
    sortZ();
}

private function move(ball:Ball3D):void
{
    var radius:Number =  ball.radius;

    ball.xpos += ball.vx;
    ball.ypos += ball.vy;
    ball.zpos += ball.vz;

    if(ball.xpos + radius > right)
    {
        ball.xpos = right - radius;
        ball.vx *= -1;
    }
    else if(ball.xpos - radius < left)
    {
        ball.xpos = left + radius;
        ball.vx *= -1;
    }
    if(ball.ypos + radius > bottom)
    {
        ball.ypos = bottom - radius;
        ball.vy *= -1;
    }
    else if(ball.ypos - radius < top)
    {
        ball.ypos = top + radius;
        ball.vy *= -1;
    }
    if(ball.zpos + radius > front)
    {
        ball.zpos = front - radius;
        ball.vz *= -1;
    }
    else if(ball.zpos - radius < back)
    {
```

415

416

www.TopSage.com

```
                ball.zpos = back + radius;
                ball.vz *= -1;
            }

            if(ball.zpos > -fl)
            {
                var scale:Number = 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;
            }
        }

        private function sortZ():void
        {
            balls.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball3D = balls[i];
                setChildIndex(ball, i);
            }
        }
    }
}
```

关键部分是以粗体表示的。小球开始都是红色的，当它们碰撞时它们改变了颜色。不久，它们都变成了蓝色。

15.9　本章重点公式

本章的重点公式是关于3D透视、坐标旋转和距离的公式。

基本的透视：

```
scale = fl / (fl + zpos);
sprite.scaleX = sprite.scaleY = scale;
sprite.alpha = scale;      // 可选的。
sprite.x = vanishingPointX + xpos * scale;
sprite.y = vanishingPointY + ypos * scale;
```


z轴排序：

```
// 假定3D物体数组具有zpos属性。
objectArray.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
for(var i:uint = 0; i < numObjects; i++)
{
    setChildIndex(objectArray[i], i);
}
```

www.TopSage.com

坐标旋转：

```
x1 = cos(angleZ) * xpos - sin(angleZ) * ypos;
y1 = cos(angleZ) * ypos + sin(angleZ) * xpos;

x1 = cos(angleY) * xpos - sin(angleY) * zpos;
z1 = cos(angleY) * zpos + sin(angleY) * xpos;

y1 = cos(angleX) * ypos - sin(angleX) * zpos;
z1 = cos(angleX) * zpos + sin(angleX) * ypos;
```

3D 距离：

```
dist = Math.sqrt(dx * dx + dy * dy + dz * dz);
```

15.10 小结

你现在已有了3D的基础，并且已经看到多数的基本运动代码是适用于3D的。我不得不说出连我自己都感到惊讶的话，“这与2D是一样的，你只需要加入一个*z*变量……”或者类似的话。我原以为这里要解释更加复杂的内容，但是结果大部分内容都相当简单。

在下一章你将用到在这里学到的许多内容，在那里你将真正开始使用点和线条雕刻你的3D模型。 418

www.TopSage.com

第16章

3D线条和填充

本章内容

- 创建点和线
- 生成面
- 创建3D填充
- 3D实体建模
- 移动3D实体模型

第15章已经对3D进行了介绍，但只是讲到了在3D空间中放置物体和计算它们的大小和在屏幕上的位置。物体本身还是2D。它有点儿像那些老的3D游戏，你可以绕着一个物体或一个人走而且它好像会转过来朝向你。物体或人并没有真正地转动——但之所以这样是因为它是一个2D物体并且那是唯一的显示方式。

在本章中，你将真正地开始在Flash中制作3D模型。尤其是，你将学习如何创建和使用3D的点、线、填充和实体。当你学完这一章后，就可以创建各种形体，移动它们，旋转它们，所有都是在3D中进行的。

16.1 创建点和线

如果不是为了绘制一些3D线条在3D中创建点并没有太多的意义，因为从定义上看，点没有维度，它是不可见的。但是，开始时，你需要继续使用具有非常小的半径的Ball3D类实例，这样你就能看到它在哪里。从这儿开始，你只需要绘制一些线条将小球连接起来。非常容易。你已经做过类似的事情了，但是现在这些点将有透视应用在它们身上，以将它们放在一个3D空间中。

你可以创建一个黑色的点的实例并且假设10 像素的直径。你需要创建一些这样的点，基于鼠标的位置旋转它们（就像前面的章节中后面的一些例子），然后在它们之间绘制线条。代码与上一章的文件RotateXY.as中的代码几乎是一样的。主要的不同之处是onEnterFrame的最后部分，在这里你通过循环来绘制线条，从第1个点开始，经过其他所有的点，到最后1个点。我也移除了sortZ方法，因为它在这里并不会产生出什么不同的效果。这里是Lines3D1.as的全部代码，图16-1显示了结果。

www.TopSage.com

TopSage.com

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Lines3D1 extends Sprite
    {
        private var balls:Array;
        private var numBalls:uint = 50;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;

        public function Lines3D1()
        {
            init();
        }

        private function init():void
        {
            balls = new Array();
            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball3D = new Ball3D(5, 0);
                balls.push(ball);
                ball.xpos = Math.random() * 200 - 100;
                ball.ypos = Math.random() * 200 - 100;
                ball.zpos = Math.random() * 200 - 100;
                addChild(ball);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var angleX:Number = (mouseY - vpY) * .001;
            var angleY:Number = (mouseX - vpX) * .001;
            for(var i:uint = 0; i < numBalls; i++)
            {
                var ball:Ball3D = balls[i];
                rotateX(ball, angleX);
                rotateY(ball, angleY);
                doPerspective(ball);
            }
            graphics.clear();
            graphics.lineStyle(0);
            graphics.moveTo(balls[0].x, balls[0].y);
            for(i = 1; i < numBalls; i++)
            {
                graphics.lineTo(balls[i].x, balls[i].y);
            }
        }

        private function rotateX(ball:Ball3D, angleX:Number):void
        {
```

422

www.TopSage.com

```
            var cosX:Number = Math.cos(angleX);
            var sinX:Number = Math.sin(angleX);

            var y1:Number = ball.ypos * cosX - ball.zpos * sinX;
            var z1:Number = ball.zpos * cosX + ball.ypos * sinX;

            ball.ypos = y1;
            ball.zpos = z1;
        }
        private function rotateY(ball:Ball3D, angleY:Number):void
        {
            var cosY:Number = Math.cos(angleY);
            var sinY:Number = Math.sin(angleY);

            var x1:Number = ball.xpos * cosY - ball.zpos * sinY;
            var z1:Number = ball.zpos * cosY + ball.xpos * sinY;

            ball.xpos = x1;
            ball.zpos = z1;
        }

        private function doPerspective(ball:Ball3D):void
        {
            if(ball.zpos > -fl)
            {
                var scale:Number = 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;
            }
        }
    }
}
```

423

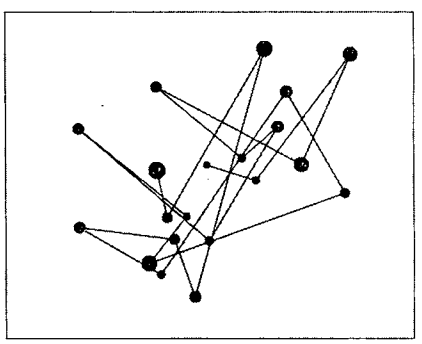


图16-1　3D点和线

现在的效果很酷。实际上这还不是它的全部。

www.TopSage.com

如果需要的是3D实体，你最终要去掉那些黑色的小点。第一次尝试是很基础的。只要在每个Ball3D实例创建时设置它的半径为0，如下：

```
var ball:Ball3D = new Ball3D(0);
```

图16-2中可以看到结果。

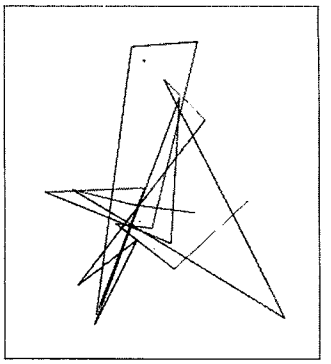


图16-2 隐藏了点的3D线条

如果查看之前的代码，你会发现有一些部分是多余的。关于scaleX、scaleY和Visible的部分与不可视的点是完全不相关的。这里没有必要看到或缩放物体。你可以沿着这条线再想得远一点，意识到我们只让当前扩展了Sprite类的这个自定义类用于承载5个变量：xpos、ypos、zpos、x和y。这样确实有点大才小用。一个精灵有许多能力——事件广播、图形、承载其他显示对象的能力等等——并且用光可用的资源。用这种方式使用精灵就像是开着坦克去街角的商店里拿一块面包。

如果我们所做的只是存储我刚才提到的那些变量，那么让我们创建一个类将那些变量作为它自己的属性。我已经做好了。实际上，不光是保存一些属性，我还把透视和坐标旋转方法移入其中了！下面我向你展示Point3D类：

```
package
{
    public class Point3D
    {
        public var fl:Number = 250;
        private var vpX:Number = 0;
        private var vpY:Number = 0;
        private var cX:Number = 0;
        private var cY:Number = 0;
        private var cZ:Number = 0;
        public var x:Number = 0;
        public var y:Number = 0;
        public var z:Number = 0;

        public function Point3D(x:Number=0, y:Number=0, z:Number=0)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }

        public function setVanishingPoint(vpX:Number, vpY:Number):void
        {
            this.vpX = vpX;
            this.vpY = vpY;
        }

        public function setCenter(cX:Number,
                                  cY:Number,
                                  cZ:Number=0):void
```


www.TopSage.com

```
        {
            this.cX = cX;
            this.cY     = cY;
            this.cZ = cZ;
        }

        public function get screenX():Number
        {
            var scale:Number = fl / (fl + z + cZ);
            return vpX + (cX + x) * scale;
        }

        public function get screenY():Number
        {
            var scale:Number = fl / (fl + z + cZ);
            return vpY + (cY + y) * scale;
        }

        public function rotateX(angleX:Number):void
        {
            var cosX:Number = Math.cos(angleX);
            var sinX:Number = Math.sin(angleX);

            var y1:Number = y * cosX - z * sinX;
            var z1:Number = z * cosX + y * sinX;

            y = y1;
            z = z1;
        }

        public function rotateY(angleY:Number):void
        {
            var cosY:Number = Math.cos(angleY);
            var sinY:Number = Math.sin(angleY);

            var x1:Number = x * cosY - z * sinY;
            var z1:Number = z * cosY + x * sinY;

            x = x1;
            z = z1;
        }

        public function rotateZ(angleZ:Number):void
        {
            var cosZ:Number = Math.cos(angleZ);
            var sinZ:Number = Math.sin(angleZ);

            var x1:Number = x * cosZ - y * sinZ;
            var y1:Number = y * cosZ + x * sinZ;

            x = x1;
            y = y1;
        }
    }
}
```

426

www.TopSage.com

使用这个类，现在当创建一个3D点时，你可以传递*x*、*y*和*z*位置，或之后使用它们的公开*x*、*y*和*z*属性设置它们。它也包括了一个默认的fl属性，和一些用于保存setVanishingPoint()方法可设置的灭点的属性。你可以使用这3种旋转方法在任意轴上围绕着点的中心来旋转点。另外，有一个setCenter方法，你可以使用它来改变中心。一会你可以看到它的应用。最后，它还控制着所有的透视。当你已经定位和旋转了点，只需要使用所有的透视计算来获取screenX和screenY以得到点的屏幕位置。虽然我已经尽力避免让你创建这样的类，但是它将使后面几章的例子变得更容易。

当你重新使用新的类来构建旋转线条的程序时，它变得非常简单，如你在Lines3D2.as所看到的：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Lines3D2 extends Sprite
    {
        private var points:Array;
        private var numPoints:uint = 50;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;

        public function Lines3D2()
        {
            init();
        }

        private function init():void
        {
            points = new Array();
            for(var i:uint = 0; i < numPoints; i++)
            {
                var point:Point3D =
                    new Point3D(Math.random() * 200 - 100,
                                Math.random() * 200 - 100,
                                Math.random() * 200 - 100);
                point.setVanishingPoint(vpX, vpY);
                points.push(point);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            var angleX:Number = (mouseY - vpY) * .001;
            var angleY:Number = (mouseX - vpX) * .001;
            for(var i:uint = 0; i < numPoints; i++)
            {
                var point:Point3D = points[i];
                point.rotateX(angleX);
                point.rotateY(angleY);
```

427

www.TopSage.com

大家网 TopSage.com

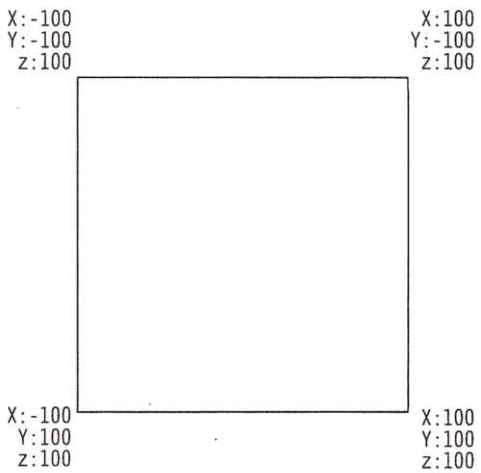
```
            }

            graphics.clear();
            graphics.lineStyle(0);
            graphics.moveTo(points[0].screenX, points[0].screenY);
            for(i = 1; i < numPoints; i++)
            {
                graphics.lineTo(points[i].screenX, points[i].screenY);
            }
        }
    }
}
```

主要的改变使用粗体标示。你应该可以很容易地理解它们。

16.2　生成面

随机的线条对于展示来说是很酷的，但是没有理由说不能在现有的混乱基础上加入一些秩序。所有要做的就是去掉创建那些点的随机的*x*、*y*和*z*值的初始循环并且替换为一些具体的、预先确定的数值。例如，让我们制作一个方形。图16-3显示了你要绘制的方形和它的4个角的3D位置。



428

图16-3　3D空间中一个方形的坐标

注意所有的*z*值都是一样的。这是因为方形位于一个平面上。保持方形在一个平面上的最简单的方法是将某一个轴上的所有点设为相同的值（这里我选择的是*z*轴）并定义方形在其他两个轴（*x*和*y*轴）的位置。

这里的代码将替换掉创建随机点的循环：

```
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);
```

接下来你必须手动设置每个点的灭点。在循环中操作是很容易的：

www.TopSage.com

```
for(var i:uint = 0; i < numPoints; i++)
{
    points[i].setVanishingPoint(vpX, vpY);
}
```

还有，因为你现在只有4个点，所以你需要改变numPoints值为4。其他的代码都没问题，但你还需要增加一点：最后1个点需要连接到第1个点上，以关闭这个方形。这里是全部的Square3D.as代码（图16-4显示了结果）：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Square3D extends Sprite
    {
        private var points:Array;
        private var numPoints:uint = 4;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;

        public function Square3D()
        {
            init();
        }

        private function init():void
        {
            points = new Array();
            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);
            for(var i:uint = 0; i < numPoints; i++)
            {
                points[i].setVanishingPoint(vpX, vpY);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var angleX:Number = (mouseY - vpY) * .001;
            var angleY:Number = (mouseX - vpX) * .001;
            for(var i:uint = 0; i < numPoints; i++)
            {
                var point:Point3D = points[i];
                point.rotateX(angleX);
                point.rotateY(angleY);
            }

            graphics.clear();
            graphics.lineStyle(0);
            graphics.moveTo(points[0].screenX, points[0].screenY);
```


www.TopSage.com

```
            for(i = 1; i < numPoints; i++)
            {
                graphics.lineTo(points[i].screenX, points[i].screenY);
            }
            graphics.lineTo(points[0].screenX, points[0].screenY);
        }
    }
}
```

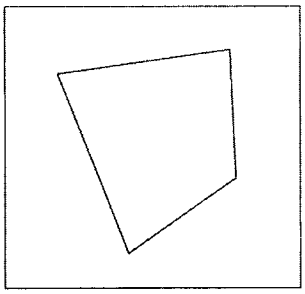


图16-4　3D旋转方形

太好了——一个旋转的方形！现在你可以创建任意的平面图形。我通常先在一张绘图纸上绘制出那些点（如图16-5所示）来帮助我创建这个图形。

430

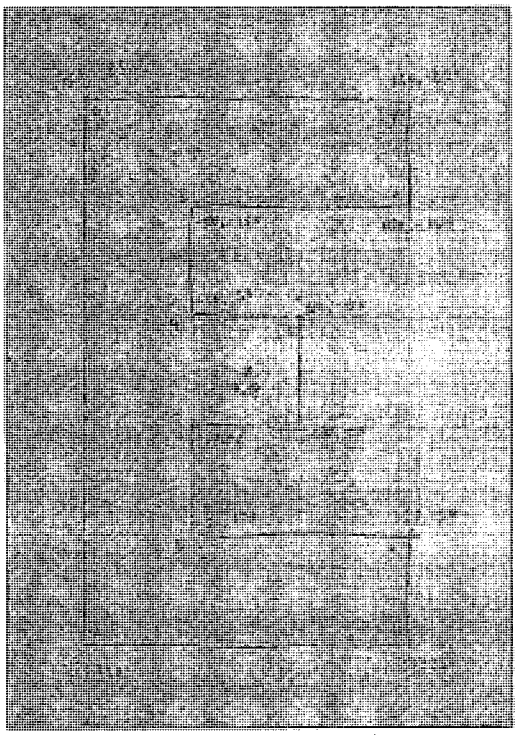


图16-5　使用绘图纸绘制点

www.TopSage.com

根据这个框架，我可以像这样创建这些点：

```
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);
```

431

那么就得到了一个旋转的E，如你在SpinningE.as中所看到的并且结果如图16-6。不要忘记改变numPoints变量为12。稍后你将通过使用points.length使这个值动态化。

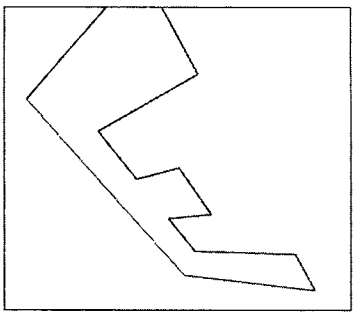


图16-6 3D旋转字母E

运行这个示例时，你会注意到当E字形在移向你时，一些点由于太近而像第1个3D例子那样倒转。你最先想到的可能是增加所有点的*z*坐标值，但是这样会使效果更差。例如，假设我设置*z*为500。当它旋转时，*z*将从500变到-500，这样会使一些点离得更远而移动到视点的后面从而产生更糟的效果。反而，使用setCenter方法将所有点沿*z*轴向后推可能解决这个问题，像这样：

```
for(var i:uint = 0; i < numPoints; i++)
{
    points[i].setVanishingPoint(vpX, vpY);
    points[i].setCenter(0, 0, 200);
}
```

如果回过头去看Point3D类，你会发现计算scale时用到了这一点：

```
var scale:Number = fl / (fl + z + cZ);
```

它将整个系统，包括系统的旋转，向后推了200个像素。*z*值是保持不变的，但是因为透视是由较高的数值计算出来的，所以所有的内容都将很好地呈现在观察者前面。试着使用其他的值来观察它是如何工作的。一会儿，我们将试验在其他的轴上移动图形。

16.3 创建3D填充

你可能已经想到，绝大部分的填充工作已经做完了。你已经为形体创建了点并且从第1个点到最后1个点用线把它们连接起来了。你现在准备要做的就是将beginFill和endFill加入绘制代码中。代码在文件FilledE.as中，图16-7为文件结果。这里是相关的部分代码，变化部分用粗体表示。

www.TopSage.com

```
graphics.clear();
graphics.lineStyle(0);
graphics.beginFill(0xffcccc);
graphics.moveTo(points[0].screenX, points[0].screenY);
for(i = 1; i < numPoints; i++)
{
    graphics.lineTo(points[i].screenX, points[i].screenY);
}
graphics.lineTo(points[0].screenX, points[0].screenY);
graphics.endFill();
```

432

它们在onEnterFrame函数的结尾处。

此时，观察传统的3D模型的形体和实体是怎样的倒是一个好主意。在之前的示例中，方形和字母E都是多边形。一个多边形只是由至少3条线段组成的封闭形体。因此，三角形是最简单的多边形。你会在许多3D建模和渲染程序中看到它——尽管它们会使用贴片（patch）、网格（mesh）、曲面建模（nurb）和复杂的多边形——所有的3D模型最终会在渲染之前被简化成一系列的三角形。

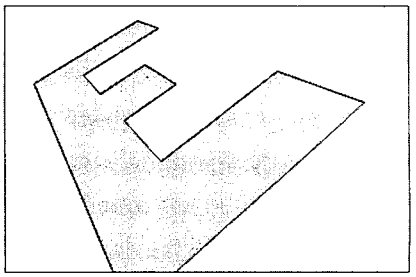


图16-7　第1个3D填充

使用三角形

使用三角形有许多好处——可能比我知道的还要多，但是我只在这里介绍几个。首先，使用一个三角形，你可确保多边形上的所有点都始终在同一个平面上，因为一个三角形确定一个平面。如果你还不确信它为什么这么重要，那么使用字母E的例子并将它的点的z值随机改变一下。那么你会得到一些有趣的结果，它们很快就会变得出乎意料而且无法预知。

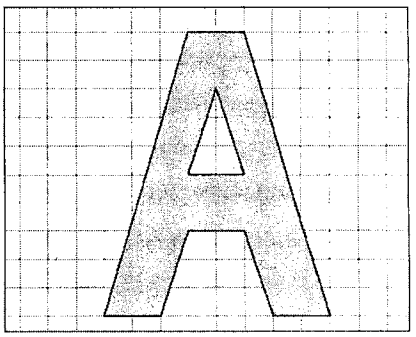


图16-8　复杂的3D形体

其次，使用三角形，有些时候绘制复杂的形体时会比较容易。以图16-8为例。

这种类型的图形使用单一的多边形来创建会很棘手。你将至少要有一次的重复工作。也不得不去解决你所创建的有多个不同点的多边形要放在哪里和需要怎样的特殊控制。而另一方面，使用三角形，你可以创建A的模型为图16-9所示。

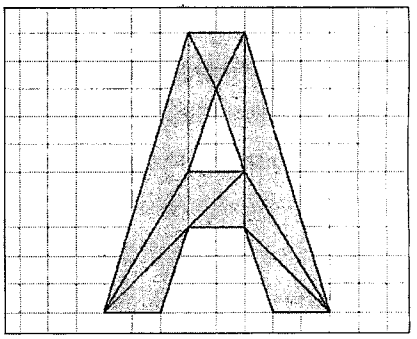


图16-9　与图16-8中的图形相同，使用三角形渲染

那么你可以建立一个函数来用3个点渲染一个三角形。只需要一个点的列表和一个三角形的列表。一个循环遍历点的列表，定位它们并应用透视。另一个循环用来遍历三角形列表并渲染每一个三角形。

这并不是说你只能使用三角形的方法。你可以创建一

www.TopSage.com

个动态渲染任意条边组成的多边形的函数。但为了保持简单和灵活性，让我们使用三角形。接下来让我们以字母A为例来试一下。首先，你需要定义所有的点和三角形。如图16-10所示，我已经制作好了图形，并且标示了所有的点和三角形。 433

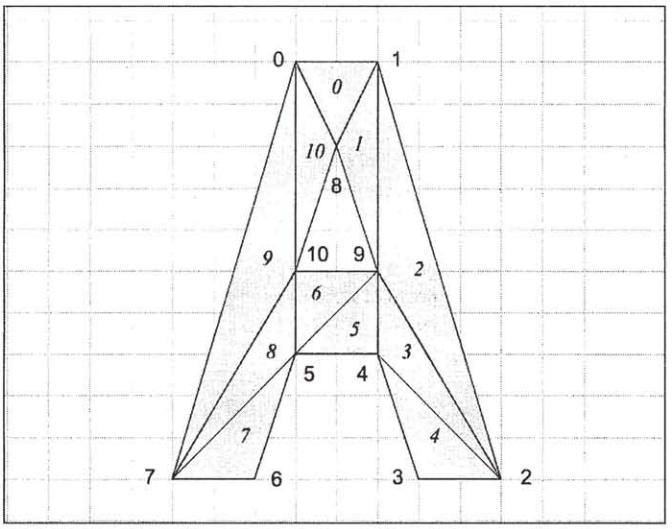


图16-10 点和多边形构成的形体

当你绘制所有的点时，得到下面的数值：

```
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，你需要定义三角形。每一个三角形是一个简单的3个点的列表，让我们称它们为a、b和c。我们创建一个Triangle类来跟踪每一个三角形的点。甚至可以为每个三角形准备一个draw方法这样它就可以知道如何绘制自己。我把它列在这里，一会你将会看到如何使用它。

```
package
{
    import flash.display.Graphics;

    public class Triangle
    {
        private var pointA:Point3D;
        private var pointB:Point3D;
        private var pointC:Point3D;
```

434

www.TopSage.com

```
        private var color:uint;

        public function Triangle(a:Point3D,
                                 b:Point3D,
                                 c:Point3D, color:uint)
        {
            pointA = a;
            pointB = b;
            pointC = c;
            this.color = color;
        }

        public function draw(g:Graphics):void
        {
            g.beginFill(color);
            g.moveTo(pointA.screenX, pointA.screenY);
            g.lineTo(pointB.screenX, pointB.screenY);
            g.lineTo(pointC.screenX, pointC.screenY);
            g.lineTo(pointA.screenX, pointA.screenY);
            g.endFill();
        }
    }
}
```

现在需要另一个数组来保存三角形列表。因此，在类的顶端定义一个数组：

```
private var triangles:Array;
```

然后，在init函数中，定义完所有的点之后，定义三角形。注意每一个三角形使用指定的颜色来创建。

```
triangles = new Array();
triangles[0] =  new Triangle(points[0], points[1],
                             points[8], 0xffcccc);
triangles[1] =  new Triangle(points[1], points[9],
                             points[8], 0xffcccc);
triangles[2] =  new Triangle(points[1], points[2],
                             points[9], 0xffcccc);
triangles[3] =  new Triangle(points[2], points[4],
                             points[9], 0xffcccc);
triangles[4] =  new Triangle(points[2], points[3],
                             points[4], 0xffcccc);
triangles[5] =  new Triangle(points[4], points[5],
                             points[9], 0xffcccc);
triangles[6] =  new Triangle(points[9], points[5],
                             points[10],0xffcccc);
triangles[7] =  new Triangle(points[5], points[6],
                             points[7], 0xffcccc);
triangles[8] =  new Triangle(points[5], points[7],
                             points[10],0xffcccc);
triangles[9] =  new Triangle(points[0], points[10],
                             points[7], 0xffcccc);
triangles[10] = new Triangle(points[0], points[8],
                             points[10],0xffcccc);
```


www.TopSage.com

你可能注意到一点：我在放置每个三角形的点时使用的是顺时针方向。对于当前来说这并不是重要的，但在下一章中它将是很重要的，所以要养成这样的好习惯。

> 顺便提一点，如果你想用手工用测量所有的点和三角形是很郁闷的，因为它太多了。而且当你在创建实体模型时情况会变得更糟。这也是大多数的3D程序都有可视的前端后端建模工具的原因，这可以提供创建模型和提取点和多边形的各种工具。创建一个有前端和后端的模型在Flash中是可能的，但它已经超出了本书的范围。

现在，你的渲染循环看起来像是这样（不要担心，一会我会给出全部代码）：

```
graphics.clear();
for(i = 0; i < triangles.length; i++)
{
    triangles[i].draw(graphics);
}
```

你看到它使用了Triangle类中定义的draw方法，传递一个graphics的引用。就这些三角形开始以它定义的颜色填充，移动到第1个点的位置，绘制它的形体，然后结束填充。非常漂亮，你可能会说。

实际上，对于这些，我奇怪为什么有的人说使用面向对象编程会把程序变得更复杂。Triangle类是非常容易理解的，并且在主类中使用到它的地方也没有那么不易理解。将它与本书较早的版本中使用的线性解决方案相比，很显然使用线性解决方案要复杂得多，并且需要大量的解释。

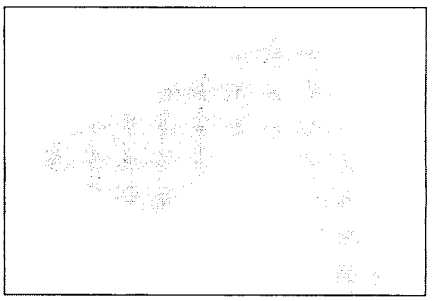


图16-11 A图形

你可以在图16-11中看到运行的结果。

为了保持内容是最新的，这里给出了Triangles.as代码：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Triangles extends Sprite
    {
        private var points:Array;
        private var triangles:Array;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;

        public function Triangles()
        {
            init();
        }

        private function init():void
```

www.TopSage.com

436
~
437

```
{
    points = new Array();
    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);
    for(var i:uint = 0; i < points.length; i++)
    {
        points[i].setVanishingPoint(vpX, vpY);
        points[i].setCenter(0, 0, 200);
    }
    triangles = new Array();
    triangles[0] = new Triangle(points[0], points[1],
                                points[8], 0xffcccc);
    triangles[1] = new Triangle(points[1], points[9],
                                points[8], 0xffcccc);
    triangles[2] = new Triangle(points[1], points[2],
                                points[9], 0xffcccc);
    triangles[3] = new Triangle(points[2], points[4],
                                points[9], 0xffcccc);
    triangles[4] = new Triangle(points[2], points[3],
                                points[4], 0xffcccc);
    triangles[5] = new Triangle(points[4], points[5],
                                points[9], 0xffcccc);
    triangles[6] = new Triangle(points[9], points[5],
                                points[10],0xffcccc);
    triangles[7] = new Triangle(points[5], points[6],
                                points[7], 0xffcccc);
    triangles[8] = new Triangle(points[5], points[7],
                                points[10],0xffcccc);
    triangles[9] = new Triangle(points[0], points[10],
                                points[7], 0xffcccc);
    triangles[10] =new Triangle(points[0], points[8],
                                points[10],0xffcccc);

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    var angleX:Number = (mouseY - vpY) * .001;
    var angleY:Number = (mouseX - vpX) * .001;
    for(var i:uint = 0; i < points.length; i++)
    {
        var point:Point3D = points[i];
        point.rotateX(angleX);
        point.rotateY(angleY);
```

www.TopSage.com

```
            }

            graphics.clear();
            for(i = 0; i < triangles.length; i++)
            {
                triangles[i].draw(graphics);
            }
        }
    }
}
```

16.4 3D 实体建模

好，最后我们来看本章的最终目的：在Flash中创建3D实体模型!

在计算机世界中，每本书或教程的第1个示例几乎都是“Hello，World”——一个会使用这样或那样的方法将这行字打印在屏幕上的程序。在3D实体模型程序中，它相当于一个旋转的立方体。那么我们不要打破传统，从立方体开始。

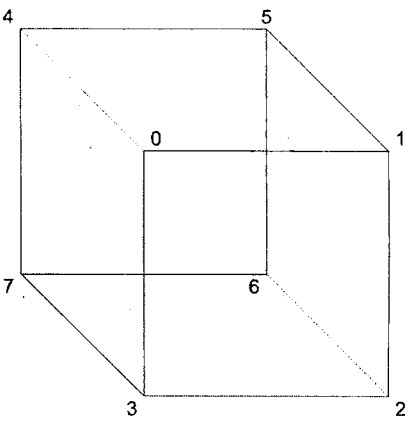


图16-12 3D立方体的点

16.4.1 旋转立方体建模

首先，你需要8个点来定义立方体的8个角。如图16-12所示。

在代码中像这样定义这些点：

```
// 前面的4个点。
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);
// 后面的4个点。
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);
```

然后你需要定义三角形。立方体的每一面由两个三角形组成。总共有12个三角形——6个面中每个面有两个。另外，我将使用顺时针方向列出每个三角形的点，正如从三角形的外边的面所看到的。它有一点小的技巧，但是试着在你的脑海中旋转立方体以使你正在定义的三角形正对着你，然后从视点按顺时针顺序列出点。例如，那个正面很容易。图16-13显示了两个三角形。

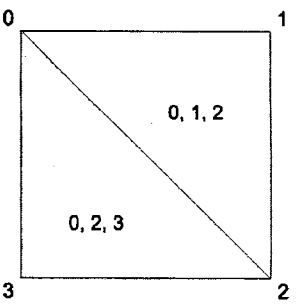


图16-13 立方体的正面

图16-14显示的是上面。而图16-15显示的是背面。

www.TopSage.com

大家网 TopSage.com

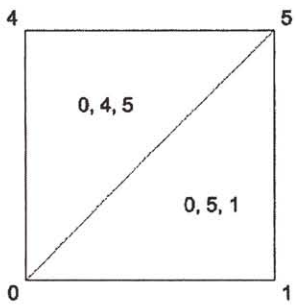


图16-14　立方体的上面

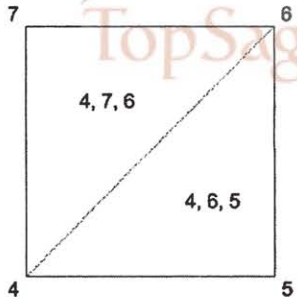


图16-15　立方体的背面

继续转动每一个面，你得到下面的三角形定义：

```
// 前面。
triangles[0] = new Triangle(points[0], points[1],
                            points[2], 0x6666cc);
triangles[1] = new Triangle(points[0], points[2],
                            points[3], 0x6666cc);
// 上面。
triangles[2] = new Triangle(points[0], points[5],
                            points[1], 0x66cc66);
triangles[3] = new Triangle(points[0], points[4],
                            points[5], 0x66cc66);
// 背面。
triangles[4] = new Triangle(points[4], points[6],
                            points[5], 0xcc6666);
triangles[5] = new Triangle(points[4], points[7],
                            points[6], 0xcc6666);
// 底面。
triangles[6] = new Triangle(points[3], points[2],
                            points[6], 0xcc66cc);
triangles[7] = new Triangle(points[3], points[6],
                            points[7], 0xcc66cc);
// 右侧。
triangles[8] = new Triangle(points[1], points[5],
                            points[6], 0x66cccc);
triangles[9] = new Triangle(points[1], points[6],
                            points[2], 0x66cccc);
// 左侧。
triangles[10] =new Triangle(points[4], points[0],
                            points[3], 0xcccc66);
triangles[11] =new Triangle(points[4], points[3],
                            points[7], 0xcccc66);
```

还要注意每一个面都有一个不同的颜色，而构成一个面的两个三角形有相同的颜色。再次，顺时针方向与整个立方体无关，但在下一章中，你将会在背面剔除中用到它。这个术语是指确定哪个面正对着你和哪个面背对着你的一种方法。你将很快发现为什么顺时针方向是很重要的。

Cube.as文件与Triangles.as文件几乎是一样的，在init方法中使用这些新的点和三角形。

运行文件。噢——完全乱了!你可以看到一些黑色的面，其他的面好像总是不可见的。发生

了什么？立方体的背面始终在绘制。并且根据三角形在triangles数组中的位置它们是以相同的颜色绘制的。因此在列表底端的面总是显示在列表顶端的面之上，而且你得到的是像这样古怪、不可预知的结果。你需要剔除或是删除那些背面，因为你不需要渲染它们了。

438 ≀ 440

另外，背面剔除将在下一章中详细讨论，并且你会学到如何在每个面上基于它的角度应用一些基本的灯光。

作为本章的一个临时修复，你可以进入到Triangle类的draw方法并设置填充透明度为0.5：

```
public function draw(g:Graphics):void
{
    g.beginFill(color, .5);
    g.moveTo(pointA.screenX, pointA.screenY);
    g.lineTo(pointB.screenX, pointB.screenY);
    g.lineTo(pointC.screenX, pointC.screenY);
    g.lineTo(pointA.screenX, pointA.screenY);
    g.endFill();
}
```

它允许你看到任何一个面，而且会使整个实体看上去是由有色玻璃制成的。另外，在我们学到背面剔除之前这只是一个临时演示。

图16-16显示了最终完成的3D立方体。

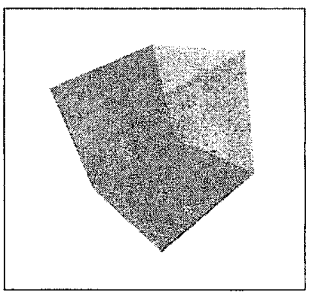


图16-16 最终完成的3D立方体

16.4.2 建模其他形体

恭喜!你已经掌握了旋转立方体的建模。现在你可以移到各种形体的建模。只需要在一张绘图纸上把它们绘制出来，标上点和三角形，然后把它们放入数组中。通常绘制几个旋转后的视图是有帮助的，这样你可以看清楚每个面和哪些点构成了三角形。这一节提供了几个示例。

1. 棱锥

这里是3D棱锥的代码（你也可以在Pyramid.as中找到）。首先是点：

```
points[0] = new Point3D(   0, -200,    0);
points[1] = new Point3D( 200,  200, -200);
points[2] = new Point3D(-200,  200, -200);
points[3] = new Point3D(-200,  200,  200);
points[4] = new Point3D( 200,  200,  200);
```

然后是三角形：

```
triangles[0] = new Triangle(points[0], points[1], points[2], 0x6666cc);
triangles[1] = new Triangle(points[0], points[2], points[3], 0x66cc66);
triangles[2] = new Triangle(points[0], points[3], points[4], 0xcc6666);
triangles[3] = new Triangle(points[0], points[4], points[1], 0x66cccc);
triangles[4] = new Triangle(points[1], points[3], points[2], 0xcc66cc);
triangles[5] = new Triangle(points[1], points[4], points[3], 0xcc66cc);
```

441

图16-17显示了结果。

www.TopSage.com

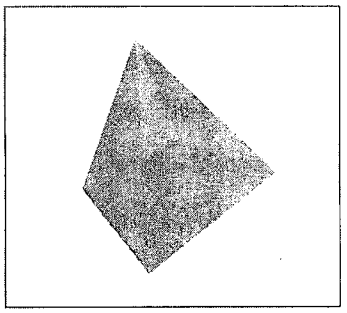


图16-17 一个3D棱锥

2. 挤压后的字母A

在ExtrudedA.as中，我尽全力对之前的字母A的例子进行了挤压。这意味着首先要复制11个点，将一组的z位置移动-50，另一组的z位置移动+50，然后为第2组点（确保它们从后面看仍是顺时钟方向）创建三角形，最后将三角形加入两个面。麻烦吗？但一个非常漂亮的效果就要做出来了：

```
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);

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],  0x6666cc);
triangles[1] =new Triangle(points[1],   points[9],
                           points[8],  0x6666cc);
triangles[2] =new Triangle(points[1],   points[2],
                           points[9],  0x6666cc);
triangles[3] =new Triangle(points[2],   points[4],
                           points[9],  0x6666cc);
triangles[4] =new Triangle(points[2],   points[3],
                           points[4],  0x6666cc);
```


www.TopSage.com

```
triangles[5] =new Triangle(points[4],   points[5],
                           points[9],  0x6666cc);
triangles[6] =new Triangle(points[9],   points[5],
                           points[10], 0x6666cc);
triangles[7] =new Triangle(points[5],   points[6],
                           points[7],  0x6666cc);
triangles[8] =new Triangle(points[5],   points[7],
                           points[10], 0x6666cc);
triangles[9] =new Triangle(points[0],   points[10],
                           points[7],  0x6666cc);
triangles[10] = new Triangle(points[0], points[8],
                             points[10], 0x6666cc);
triangles[11] = new Triangle(points[11], points[19],
                             points[12], 0xcc6666);
triangles[12] = new Triangle(points[12], points[19],
                             points[20], 0xcc6666);
triangles[13] = new Triangle(points[12], points[20],
                             points[13], 0xcc6666);
triangles[14] = new Triangle(points[13], points[20],
                             points[15], 0xcc6666);
triangles[15] = new Triangle(points[13], points[15],
                             points[14], 0xcc6666);
triangles[16] = new Triangle(points[15], points[20],
                             points[16], 0xcc6666);
triangles[17] = new Triangle(points[20], points[21],
                             points[16], 0xcc6666);
triangles[18] = new Triangle(points[16], points[18],
                             points[17], 0xcc6666);
triangles[19] = new Triangle(points[16], points[21],
                             points[18], 0xcc6666);
triangles[20] = new Triangle(points[11], points[18],
                             points[21], 0xcc6666);
triangles[21] = new Triangle(points[11], points[21],
                             points[19], 0xcc6666);
triangles[22] = new Triangle(points[0],  points[11],
                             points[1],  0xcccc66);
triangles[23] = new Triangle(points[11], points[12],
                             points[1],  0xcccc66);
triangles[24] = new Triangle(points[1],  points[12],
                             points[2],  0xcccc66);
triangles[25] = new Triangle(points[12], points[13],
                             points[2],  0xcccc66);
triangles[26] = new Triangle(points[3],  points[2],
                             points[14], 0xcccc66);
triangles[27] = new Triangle(points[2],  points[13],
                             points[14], 0xcccc66);
triangles[28] = new Triangle(points[4],  points[3],
                             points[15], 0xcccc66);
triangles[29] = new Triangle(points[3],  points[14],
                             points[15], 0xcccc66);
triangles[30] = new Triangle(points[5],  points[4],
                             points[16], 0xcccc66);
triangles[31] = new Triangle(points[4],  points[15],
                             points[16], 0xcccc66);
```


www.TopSage.com

```
triangles[32] = new Triangle(points[6],  points[5],
                             points[17], 0xcccc66);
triangles[33] = new Triangle(points[5],  points[16],
                             points[17], 0xcccc66);
triangles[34] = new Triangle(points[7],  points[6],
                             points[18], 0xcccc66);
triangles[35] = new Triangle(points[6],  points[17],
                             points[18], 0xcccc66);
triangles[36] = new Triangle(points[0],  points[7],
                             points[11], 0xcccc66);
triangles[37] = new Triangle(points[7],  points[18],
                             points[11], 0xcccc66);
triangles[38] = new Triangle(points[8],  points[9],
                             points[19], 0xcccc66);
triangles[39] = new Triangle(points[9],  points[20],
                             points[19], 0xcccc66);
triangles[40] = new Triangle(points[9],  points[10],
                             points[20], 0xcccc66);
triangles[41] = new Triangle(points[10], points[21],
                             points[20], 0xcccc66);
triangles[42] = new Triangle(points[10], points[8],
                             points[21], 0xcccc66);
triangles[43] = new Triangle(points[8],  points[19],
                             points[21], 0xcccc66);
```

444

图16-18显示了结果。

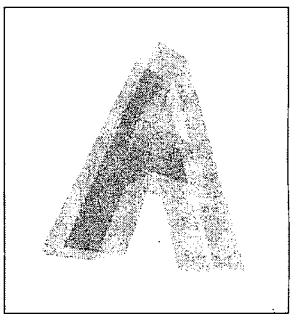


图16-18　挤压后的字母A

如你所看到的，这些东西创建得很快。原始的平板的A有11个三角形。现在已是原来的4倍，代码运行得仍然很光滑，但你不会在Flash中得到使用数以千计的多边形生成的大型3D模型。然而你仍可以做出一些很酷的东西。这个程序在AS 2中同样运行得很好，使用AS 3会更快一些，这里我并不认为已经接近了它的极限。而且随着Flash播放器性能的提升，谁也不知道将来会发生什么。

3. 圆柱体

再来一个形体的示例，这次我将向你展示如何使用一些数学来创建点和三角形。在Cylinder.as中我唯一修改的地方是init函数（而且我在顶端加入了一个numFaces变量）。不是手动地定义点和三角形，而是我创建了一个算法来帮助完成这些并生成一个圆柱。这儿是init函数：

```
private var numFaces:uint = 20;

private function init():void
{

    points = new Array();
    triangles = new Array();

    var index:uint = 0;
    for(var i:uint = 0; i < numFaces; i++)
    {
```

www.TopSage.com

TopSage.com

```
            var angle:Number = Math.PI * 2 / numFaces * i;
            var xpos:Number = Math.cos(angle) * 200;
            var ypos:Number = Math.sin(angle) * 200;
            points[index] =     new Point3D(xpos, ypos, -100);
            points[index + 1] = new Point3D(xpos, ypos,  100);
            index += 2;
        }
        for(i = 0; i < points.length; i++)
        {
            points[i].setVanishingPoint(vpX, vpY);
            points[i].setCenter(0, 0, 200);
        }
        index = 0;
        for(i = 0; i < numFaces - 1; i++)
        {
            triangles[index] =     new Triangle(points[index],
                                                points[index + 3],
                                                points[index + 1],
                                                0x6666cc);
            triangles[index + 1] = new Triangle(points[index],
                                                points[index + 2],
                                                points[index + 3],
                                                0x6666cc);
            index += 2;
        }

        triangles[index] =   new Triangle(points[index],
                                          points[1],
                                          points[index + 1],
                                          0x6666cc);
        triangles[index+1] = new Triangle(points[index],
                                          points[0],
                                          points[1],
                                          0x6666cc);

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
```

445

现在，我知道这并不是最清晰的代码，所以我来解释一下而且可能要用到一两个图表。

你将在一个整圆内循环并以某个间隔插入一些点。对于每一个循环，首先获得一个角，它是一个整圆除以面数，乘以你正在使用的段数。

你使用那个角，通过三角学——到目前为止你应可以很熟练地使用三角学——来确定圆上那个点的x,y位置。然后创建两个点，一个使z为–100，另一个使z为+100。当这个循环结束后，你会得到两个点圆，一个是靠近你的，另一个是远离你的。现在你要做的只是要使用三角形连接它们。

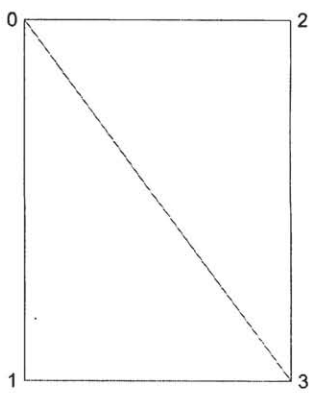


图16-19 圆柱的第1个面

再次，你循环每一个面。这一次，你创建两个三角形。从侧面

www.TopSage.com

看，第1个面如图16-19所示。

它使得两个三角形为：

446
```
0, 3, 1
0, 2, 3
```

因为index值为0，也可以这样定义：

```
index, index + 3, index + 1
index, index + 2, index + 3
```

这就是你定义那两个三角形的方法。然后将index增加2以控制下一个点为2、3、4和5的面。

按上面的方法一直到倒数第2个面，然后将最后1个面与第1个面的两个点0和1相连，如图16-20所示。

最后的两个三角形如下：

```
index, 1, index + 1
index, 0, 1
```

结果如图16-21所示。如果想让它的形状更清晰一些，你可以进入Triangle类中并在绘制每个三角形的代码中加入一行lineStyle语句。

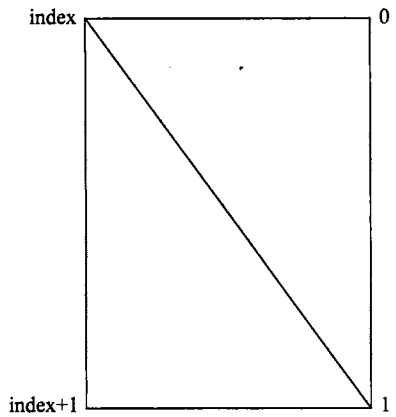


图16-20　圆柱的最后一个面

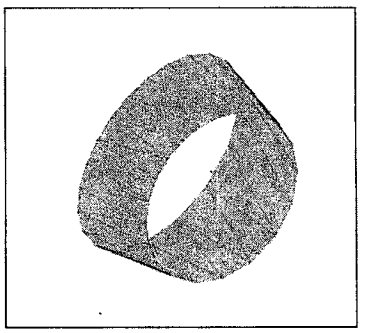


图16-21　最后生成的3D圆柱

16.5　移动3D实体模型

移动一个3D实体非常简单，因为你有了Point3D类。只需要使用它的setCenter方法改变它的中心。你已经做过在z轴上移动图形。现在只是在其他轴上做相同的事情。但是让我们快速看一下当我们改变了那些值后代码做了些什么。所有的都在Point3D类的screenX和screenY方法中：

```
public function get screenX():Number
{
    var scale:Number = fl / (fl + z + cZ);
    return vpX + (cX + x) * scale;
}
```
447

www.TopSage.com

```
public function get screenY():Number
{
    var scale:Number = fl / (fl + z + cZ);
    return vpY + (cY + y) * scale;
}
```

中心的z属性cZ被加到z轴位置数值上，但只是当缩放被计算时，会起到向视点前方推动点而不会真正改变z轴自身位置的效果。

cX和cY也是同样。它们被加入到x和y的位置值上，只发生在缩放被应用之前。它会在某个轴上推动点，点自身的位置并不会真正改变。因为我们没有改变它们的任何值，这些点——因此整个大的形体的一部分——将继续围着它自身的中心旋转，而不是围绕着整个系统的中心旋转。

让我们看一下它的动作脚本。回到Cube.as类中，在类的顶端加入两个类属性：

```
private var offsetX:Number = 0;
private var offsetY:Number = 0;
```

然后在init方法中，加入下面的行：

```
stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
```

然后创建处理器函数：

```
private function onKeyDown(event:KeyboardEvent):void
{
    if(event.keyCode == Keyboard.LEFT)
    {
        offsetX = -5;
    }
    else if(event.keyCode == Keyboard.RIGHT)
    {
        offsetX = 5;
    }
    else if(event.keyCode == Keyboard.UP)
    {
        offsetY = -5;
    }
    else if(event.keyCode == Keyboard.DOWN)
    {
        offsetY = 5;
    }
    for(var i:Number = 0; i < points.length; i++)
    {
        points[i].x += offsetX;
        points[i].y += offsetY;
    }
}
```

不要忘记导入KeyboardEvent和Keyboard类。

448

这只是循环每个点然后从它的值中加或减5。因为所有的点的实际位置在改变。所以这个模型现在会围着整个3D空间的中心旋转。这可能是你想要的，但也可能不是。如果你只是想移动整个模型，并且仍让它围绕着自身的中心旋转，这时就需要应用setCenter方法。改变onKeyDown代码如下：

www.TopSage.com

```
private function onKeyDown(event:KeyboardEvent):void
{
    if(event.keyCode == Keyboard.LEFT)
    {
        offsetX -= 5;
    }
    else if(event.keyCode == Keyboard.RIGHT)
    {
        offsetX += 5;
    }
    else if(event.keyCode == Keyboard.UP)
    {
        offsetY -= 5;
    }
    else if(event.keyCode == Keyboard.DOWN)
    {
        offsetY += 5;
    }
    for(var i:Number = 0; i < points.length; i++)
    {
        points[i].setCenter(offsetX, offsetY, 200);
    }
}
```

现在整个立方体作为一个整体在移动，并已继续围绕着它的中心旋转，而不是围绕着整个系统的中心旋转。

16.6　小结

通过本章的学习，你应当可以在ActionScript中创建你自己的3D模型并在3D空间中控制它们。

这一章集中讲的是模型的外形以及如何绘制线条和填充。在下一章中，你将研究如何创建更丰富的实体模型。所有的内容都是在你已学过的点、线和填充的基础上创建上。因此，如果你准备好了，我们就继续吧。

449

www.TopSage.com

第17章

背面剔除和3D灯光

本章内容

- 背面剔除
- 深度排序
- 3D灯光

在第16章，我已经介绍了所有的3D 实体建模基础：如何创建点、线条和生成模型的多边形，甚至是如何给每一个多边形指定颜色。但如果你回顾起来，你只是使用了50%的透明颜色，因此你可以看到它们的全部。虽然对于复杂的3D实体建模来说它是比较酷的，但仍然缺少许多现实的感觉。

在这一章中，我将补充介绍背面剔除（并不是在背着你的方向绘制多边形），深度排序（在第15章时介绍过一点，但我们要在多边形的概念中重新看待它）和3D灯光。

我想一旦你应用了这3种技术你将会对3D模型的结果感到很惊奇。前两种技术介绍后，你可以创建出真正的实体模型。使用3D灯光，那它们是真正地活灵活现了。

在我们开始之前，我要给出一些赞誉。在这一章中几乎所有的代码都是来自Todd Yard的*Macromedia Flash MX Studio*（friends of ED，2002）的第10章。（Todd也是本书的技术评论人，因此我并不算是从他那窃取东西！）*Macromedia Flash MX Studio*的那一章可能是我所见到的关于这些课题的最好的资源，每当我需要一些高级的3D技术时我都会参考它。这本书已经有4年多了（对技术书来说是一本老书了）并且Flash已经经历了3个版本，我非常高兴我可以把这些知识向前推进以保持它在当前仍有可用性。

对于本章的示例，我将以上一章中末尾处所创建的旋转的、挤压的、3D字母A为基础。它是一个足够复杂的模型，当你什么地方做错时它会非常明显地显示出来，当你做得都正确时它看起来是相当酷的。

17.1 背面剔除

背面剔除（backface culling）在上一章中已经提到了两次，现在你将搞清楚它到底是什么和如何使用它。

www.TopSage.com

大家网 TopSage.com

记得在之前的模型中，所有的填充都是半透明的。这样做的原因是你总是在绘制每一个多边形，而并没有控制它们绘制的顺序。因此一个模型的背面的多边形在绘制后将会显示在模型的前面的多边形的上面，得出些奇怪的效果。给它们所有的alpha值为50%是为了使它们的关系均衡并且是为了让我们放弃对它的讨论而把精力集中在模型的创建上。但现在你将学习如何处理它。

从原理上看，背面剔除相当简单。你绘制面对你的多边形，并且你不需要绘制背对着你的那些多边形。复杂的部分是如何检测它们。

你可能还记得我经常提醒你定义每个多边形的点时使用顺时针方向。尽管到目前为止你所做过的东西是完全没有必要的，现在你将看到为什么它是很重要的和为什么从这里开始它是一个好的习惯。

当一个三角形面对你时，如果多边形是以顺时钟方向安排的则它是很容易观察的，当多边形背对着你时它们将是逆时针方向的。你可以看到如图17-1的演示，它有一个三角形面对着你。（因为你将使用的所有的多边形都是三角形，所以我偶尔会混用这两个术语。大多数情况，我使用“多
452 边形”作为一个通用术语，而“三角形”用来讨论由三角形构成的特殊的多边形。）

而在图17-2中，我已经旋转了三角形以使它面对相反的方向。

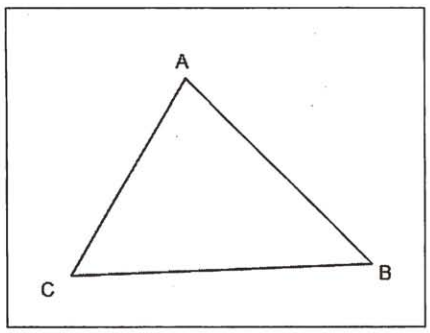


图17-1　一个面对着你的三角形，它有3个顺时针方针的点

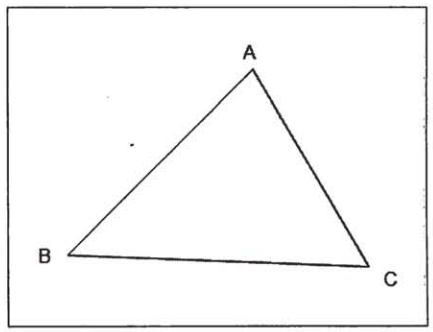


图17-2　一个背对着你的三角形，它有3个逆时针方向的点

现在你看到的点是逆时针方向的。

在这里有几点内容要澄清一下。首先，当我说一个多边形正“面对着你”是什么意思呢？我的意思是说多边形的外（exterior）表面正在面对着你。尽管展示一个三角形不是很明显，但记住我正在讨论的是3D实体。在这样的情况下，每一个多边形都有一外表面和一个内表面。

另一点是当确定顺时针和逆时针时，我们讨论的是点的屏幕位置（screen position）。它不是3D中的*x*、*y*和*z*位置，而是通过应用透视来确定的screenX、screenY位置。

最后，还有一点需要注意的是你可以反转设置，创建一个逆时针的多边形面对着你且顺时针
453 的多边形背对着你的系统，不管哪一种方式，只要你保持一致性它就可以工作。

那么，我们又回到这个问题上了，你如何确定3个点是顺时针方向还是逆时针方向安排的？仔细考虑一下。对你的眼睛来说分辨它们是很简单的事情，但是当把它放在代码中，它突然好像变成了一个很抽象的概念。

www.TopSage.com

如我提到的，这里我将给出的解决方案是基于*Macromedia Flash MX Studio*这本书中的。但为了确保我提供的方法是最好的，我决定看一下自己编写的函数是否可以区分出顺时针和逆时针。当我试着把一些东西放在一起时它们工作得很好，它的代码长度是现有解决方案的两倍而且要复杂得多，涉及了坐标旋转和许多的三角学。我决定给出那个简单的方法。不用担心，它和我的复杂方法一样准确！

你所要做的是在Triangle类中加入一个方法，命名为isBackFace。它会求出构成三角形的3个点并且如果是逆时针返回true，如果是顺时针则返回false。下面是有那个新增函数的类。也需要注意在调用beginFill方法时不再需要使用0.5的alpha值了。我们将绘制不透明的形体。

```
package
{
    import flash.display.Graphics;

    public class Triangle
    {
        private var pointA:Point3D;
        private var pointB:Point3D;
        private var pointC:Point3D;
        private var color:uint;

        public function Triangle(a:Point3D, b:Point3D,
                                 c:Point3D, color:uint)
        {
            pointA = a;
            pointB = b;
            pointC = c;
            this.color = color;
        }

        public function draw(g:Graphics):void
        {
            if(isBackFace())
            {
                return;
            }
            g.beginFill(color);
            g.moveTo(pointA.screenX, pointA.screenY);
            g.lineTo(pointB.screenX, pointB.screenY);
            g.lineTo(pointC.screenX, pointC.screenY);
            g.lineTo(pointA.screenX, pointA.screenY);
            g.endFill();
        }
        private function isBackFace():Boolean
        {
            // 参见 http://www.jurjans.lv/flash/shape.html。
            var cax:Number = pointC.screenX - pointA.screenX;
            var cay:Number = pointC.screenY - pointA.screenY;

            var bcx:Number = pointB.screenX - pointC.screenX;
            var bcy:Number = pointB.screenY - pointC.screenY;
```

454

www.TopSage.com

```
                return cax * bcy > cay * bcx;
            }
        }
    }
```

你看到一个网站的链接http://www.jurjans.lv/flash/shape.html。Todd Yard在*Macromedia Flash MX Studio*感谢过这个网站。另外网站还有很多类似课题的非常好的参考资料和教程。

下面做一个快速的解释，函数计算三角形两个边的长度，前且相乘然后比较，可以告诉你它们的走向。如果你想知道它为什么这样运算，参看刚才提到的网站，或在网上搜索“backface culling”（背面剔除）。我确信你会找到大量的阅读资料。但是现在，我只是说它很简单、快速、有效、并且100%可工作。

那么，怎样使用这个方法呢？实际上不需要考虑它。它是一个私有方法，意味着它只能在Triangle类的自身被调用。它在draw方法中的第1行被调用。如果返回的是true，则三角形是一个背面并且不应该绘制出来，那么draw方法只是停在那里并返回。如果isBackFace返回false，则三角形面对着你并且将会像平常一样绘制出来。

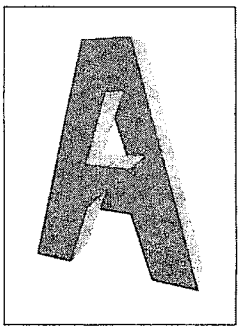


图17-3　背面剔除动作

现在，可以运行ExtrudedA.as或其他已经创建的任何3D模型，并且你会发现情况会有一些不同。当旋转形体时，你会发现背面不再被绘制了。然而情况并不是非常地好，因为仍有些离得较远的部分绘制在离得较近的部分的上面，但是我们就到这。如果你想到了“*z*轴排序”或“深度排序”，这就对了。这是我们接下来要说的内容。

现在你应该正在看类似于下面的图17-3所示的图形。

17.2　深度排序

深度排序（Depth sorting），或*z*轴排序，是我们在第15章已经讨论过的内容，当时只是将透视应用于精灵。在那种情况下，你是通过精灵的zpos属性对精灵（实际上是有3D属性的Sprite
455 的子类）的数组进行排序的。

然而在这一点上，你实际上并没有处理多个精灵。所有的多边形也是绘制在主文档类的同一个graphics对象中。因此，不管什么时候，只要有一个多边形被绘制出来，它就会显示在之前绘制的多边形之上。因此，不是交换它们的深度，你只需要确定哪个多边形是在什么时候绘制的。具体来说，你需要先绘制离你最远的那个多边形，然后再绘制其他的，按照这个顺序向前绘制，因此离你最近的那个多边形是最后绘制的，覆盖掉所有可能在它前面的多边形。

那么你要怎样做呢？好，让所有多边形放在一个名为triangles的数组中。当绘制形体时，循环这个triangles，从0开始一直到最后绘制每一个三角形。你所需要做的就是对这个数组排序以使最远的那个三角形在数组中的索引是0，而离你视点最近的那个多边形是数组中最后一个元素。

这与我们对精灵数组排序时是非常相似的。但在这个例子中，三角形只是3个Point3D对象的集合。它们没有单独地描述三角形的总体深度的属性。然而，创建一个这样的属性是很容易的。它转变成了算出三角形3个点中最小*z*值的问题。换句话说，如果一个三角形有3个点，深度分别

www.TopSage.com

为200、250和300，那么我们应该说这个三角形在z为200的位置上。

我们可以使用Math.min方法来计算出所有3个点中最小的z值。但我们不得不使用它两次，因为一次只能传递两个数值。我们将在Triangle类中的新的get depth方法中完成此计算。这儿是更新后的类：

```
package
{
    import flash.display.Graphics;

    public class Triangle
    {
        private var pointA:Point3D;
        private var pointB:Point3D;
        private var pointC:Point3D;
        private var color:uint;

        public function Triangle(a:Point3D, b:Point3D,
                                 c:Point3D, color:uint)
        {
            pointA = a;
            pointB = b;
            pointC = c;
            this.color = color;
        }

        public function draw(g:Graphics):void
        {
            if(isBackFace())
            {
                return;
            }
            g.beginFill(color);
            g.moveTo(pointA.screenX, pointA.screenY);
            g.lineTo(pointB.screenX, pointB.screenY);
            g.lineTo(pointC.screenX, pointC.screenY);
            g.lineTo(pointA.screenX, pointA.screenY);
            g.endFill();
        }

        private function isBackFace():Boolean
        {
            // 参见 http://www.jurjans.lv/flash/shape.html。
            var cax:Number = pointC.screenX - pointA.screenX;
            var cay:Number = pointC.screenY - pointA.screenY;

            var bcx:Number = pointB.screenX - pointC.screenX;
            var bcy:Number = pointB.screenY - pointC.screenY;

            return cax * bcy > cay * bcx;
        }

        public function get depth():Number
        {
```

456

www.TopSage.com

```
            var zpos:Number = Math.min(pointA.z, pointB.z);
            zpos = Math.min(zpos, pointC.z);
            return zpos;
        }
    }
}
```

现在你可以对这个triangles数组排序并且知道哪个最先绘制和哪个最后绘制。另外，还需要以降序排列以使有最高深度值（最远的）的三角形为数组的第1个元素。你必须在你的主文档类中完成它，并且要刚好在绘制任何三角形之前完成。我使用的是ExtrudedA类，因此这个类的onEnterFrame方法看起来像这样：

```
private function onEnterFrame(event:Event):void
{
    var angleX:Number = (mouseY - vpY) * .001;
    var angleY:Number = (mouseX - vpX) * .001;
    for(var i:uint = 0; i < points.length; i++)
    {
        var point:Point3D = points[i];
        point.rotateX(angleX);
        point.rotateY(angleY);
    }

    triangles.sortOn("depth", Array.DESCENDING | Array.NUMERIC);
    graphics.clear();
    for(i = 0; i < triangles.length; i++)
    {
        triangles[i].draw(graphics);
    }
}
```

457

你看到代码中只改变了一行。面向对象编程真地使我们的生活更容易了！

当运行它时，你会看到一个很完美的渲染后的实体，如图17-4所示。现在你的确到达了一定水平。下一步将带你看酷到极点的效果！

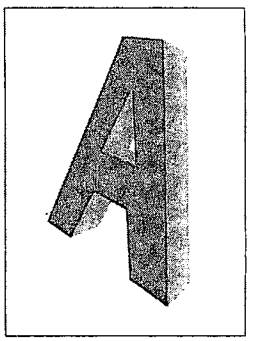


图17-4　深度排序

17.3　3D灯光

最后一个例子从渲染的角度说已经非常接近完美了，但它仍少点什么。它还是显得有些平。好的，你已经知道我要说什么了，因为在标题中也标出了，因此，让我们加入一些3D灯光。

就像背面剔除，在特殊的3D灯光后面可能是相当复杂和精密的数学。我没有那么多空间来对每一个细节进行探讨，但如果在网上搜索将会得到更多关于此话题的信息。在这里我要给你的是一些基本知识和一些你可以使用的、适应需要的函数。

首先，你需要一个光源。一个最简单的光源有两个属性：位置和亮度。在更加复杂的3D系统中，它也可以定位在某个方向上，有自身的颜色、衰减、光照的区域等等。但是所有这些都已超出了你要做的范围。

www.TopSage.com

TopSage.com

让我从制作一个Light类开始。它将包含我刚才提到的两个属性——位置和亮度。

```
package
{
    public class Light
    {
        public var x:Number;
        public var y:Number;
        public var z:Number;
        private var _brightness:Number;

        public function Light(x:Number = -100,
                              y:Number = -100,
                              z:Number = -100,
                              brightness:Number = 1)
        {
            this.x = x;
            this.y = y;
            this.z = z;
            this.brightness = brightness;
        }
        public function set brightness(b:Number):void
        {
            _brightness = Math.max(b, 0);
            _brightness = Math.min(_brightness, 1);
        }

        public function get brightness():Number
        {
            return _brightness;
        }
    }
}
```

458

现在，在你的主文档类的init方法中，可以创建一个新的默认灯光如下：

```
var light:Light = new Light();
```

或者，可以创建一个使用指定位置和亮度的灯光如下：

```
var light:Light = new Light(100, 200, 300,0.5);
```

这里有两件重要的事需要注意。一个是位置，只用来计算灯光的角度。你所创建的灯光的强度并没有随着距离衰减。因此改变x、y和z到–1 000 000或到–1就照射到物体上的亮度而言是没有区别的。只有brightness属性可以改变灯光的性能。你可以增加这个功能，基于距离的变化改变灯光的亮度值。它并不是那么难，但我还有很多内容要讲，因此把它留给你来做。

还有，亮度必须是一个从0.0到1.0的数值。如果超出了这个范围，你可能会得到一些奇怪的结果。因为这个原因，我们把_brightness作为一个私有属性，而且允许通过公开的getter和setter访问brightness。这允许我们验证那个传入的数字，以确保它在指定的范围之内。

理论上，一个类根本没有公开的属性，只有可以通过getter和setter访问的私有属性，即使是那些属性不需要被验证。我已经去除了许多没有用的代码以使它更简单，并能体现动画的原理。

www.TopSage.com

但在这个例子中，有一个比较好的理由可以使用特殊的方法。

现在，光源所要做的是改变三角形颜色的亮度，基于的是照在多边形上的灯光的角度。因此如果多边直接面对着灯光，它将显示它的全部颜色值。当离开灯光时，它将变得越来越暗。最后，当多边形完全离开光源时，它完全处在阴影中并且颜色是黑色。

因为Triangle类的成员跟踪着它们自己的颜色，并且知道如何绘制它们自己，好像每一个三角形都需要访问这个灯光来执行它的绘制函数。那么，让我们给这些三角形一个light属性。我们将再去掉一些没用的代码并将它变成一个公开属性：

459

```
public var light:Light;
```

然后在主文档类中，当你创建完三角形，循环它们并给每个三角形分配一个灯光的引用：

```
var light:Light = new Light();
for(i = 0; i < triangles.length; i++)
{
    triangles[i].light = light;
}
```

或者，你也可以将light作为一个附加的参数放在Triangle的构造函数中，它将会确保在创建每一个三角形时都会有一个光源。我把它留给你去选择。

那么，现在Triangle需要一个函数可以看到它的基础色和灯光的角度和亮度，然后返回一个调节后的颜色。这儿就是这个函数：

```
function getAdjustedColor():uint
{
    var red:Number =   color >> 16;
    var green:Number = color >> 8 & 0xff;
    var blue:Number =  color & 0xff;

    var lightFactor:Number = getLightFactor();

    red *= lightFactor;
    green *= lightFactor;
    blue *= lightFactor;

    return red << 16 | green << 8 | blue;
}
```

这个函数首先分解三角形的基础颜色为红色、绿色和蓝色（见第4章）。然后调用另一个方法getLightFactor。一会你将会看到这个函数。现在，你只需要知道它返回一个在0.0到1.0的数值。这就是改变指定三角形颜色所需要的数值，1.0是完全亮度，而0.0是完全黑色。

然后使用lightFactor乘以每一个组成色的值，再将它们组合成一个24位颜色值，并返回调整后的颜色值。这将是基于灯光的三角形的颜色。

现在，你怎样得到lightFactor呢？让我们看一下：

```
private function getLightFactor():Number
{
    var ab:Object = new Object();
    ab.x = pointA.x - pointB.x;
```

www.TopSage.com

```
        ab.y = pointA.y - pointB.y;
        ab.z = pointA.z - pointB.z;

        var bc:Object = new Object();
        bc.x = pointB.x - pointC.x;
        bc.y = pointB.y - pointC.y;
        bc.z = pointB.z - pointC.z;
        var norm:Object = new Object();
        norm.x = (ab.y * bc.z) - (ab.z * bc.y);
        norm.y = -((ab.x * bc.z) - (ab.z * bc.x));
        norm.z = (ab.x * bc.y) - (ab.y * bc.x);

        var dotProd:Number = norm.x * light.x +
                             norm.y * light.y +
                             norm.z * light.z;

        var normMag:Number = Math.sqrt(norm.x * norm.x +
                                       norm.y * norm.y +
                                       norm.z * norm.z);

        var lightMag:Number = Math.sqrt(light.x * light.x +
                                        light.y * light.y +
                                        light.z * light.z);

        return (Math.acos(dotProd / (normMag * lightMag)) / Math.PI)
               * light.brightness;
    }
```

460

哇，这就是那个函数吗？要完全（fully）理解这里所有的内容，你需要有很好的高级向量数学知识，我将尽力去讲解它的基础。

首先你需要做的是找到三角形的法线（normal）。它是一个垂直于三角形表面的向量，如图17-5所示。想像你有一个三角形的木板，你将一个钉子从它的背面直接穿过木板表面。这个钉子代表的就是这个表面的法线。如果你学习过任何关于3D渲染和灯光的内容，你会看到各种关于法线的参考。

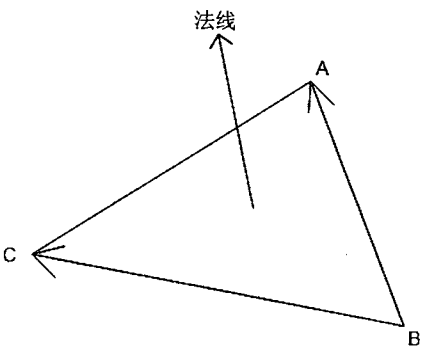


图17-5 垂直于三角形表面的法线

你可以通过选择形成表面的两个向量并计算它们的向量积（cross product）来得到一个表面的法线。两个向量的向量积是一个新的向量，它垂直于另外两个向量。你将要使用的两个向量是点A和B之间的线及点B和C之间的线。

461

```
var ab:Object = new Object();
ab.x = pointA.x - pointB.x;
ab.y = pointA.y - pointB.y;
ab.z = pointA.z - pointB.z;

var bc:Object = new Object();
bc.x = pointB.x - pointC.x;
```

www.TopSage.com

```
bc.y = pointB.y - pointC.y;
bc.z = pointB.z - pointC.z;
```

然后计算这个法线，它是另外一个向量。可称这个对象为norm。下面的代码计算向量ab和bc的向量积：

```
var norm:Object = new Object();
norm.x = (ab.y * bc.z) - (ab.z * bc.y);
norm.y = -((ab.x * bc.z) - (ab.z * bc.x));
norm.z = (ab.x * bc.y) - (ab.y * bc.x);
```

另外，我没有空间过多地去涉及为什么要这样计算的一些细节，但这是计算向量积的标准公式。如果你对它的来源感兴趣，可以查看一些有关线性代数方面的参考资料。

现在你需要知道法线对齐灯光的角度后它们有多近。另一个向量数学称为点积(dot product)，它是指两个向量的差。你已经有了法线向量、灯光向量。下面计算点积：

```
var dotProd:Number = norm.x * light.x +
                     norm.y * light.y +
                     norm.z * light.z;
```

如你所看到的，点积要比向量积简单一点！

好，你几乎要完成了！接下来，计算法线的大小和灯光的大小，你可以把它看成是3D版的毕达哥拉斯定理：

```
var normMag:Number = Math.sqrt(norm.x * norm.x +
                               norm.y * norm.y +
                               norm.z * norm.z);

var lightMag:Number = Math.sqrt(light.x * light.x +
                                light.y * light.y +
                                light.z * light.z);
```

注意这里的lightMag变量会在每次渲染三角形时被计算，它可以允许移动的光源。如果你知道光源将是固定的，你可以在代码开始处创建这个变量，并且一旦灯光被创建或指定给三角形，
462 就开始计算这个值。作为选择，你可以加入一个lightMag属性到Light类中。当它的*x*、*y*或*z*属性真正改变时可以计算它。看，我正在给你指示各种思路以使你可以改进它们！

最后，你得到所有的值，然后将它们放在这个神奇的公式中：

```
return (Math.acos(dotProd / (normMag * lightMag)) / Math.PI)
       * light.brightness;
```

基本上，dotProd是一个量度而normMag * lightMag是另一个。两者相除可以得到一个比例。回顾一下第3章，一个角度的余弦得到的是一个比例，而这个比例的反余弦得到的是一个角度。这实际上就是灯光照射到多边形平面的角度。它的范围是0到Math.PI弧度（0到180度），意味着它要么照射到正上面要么完全在后面。

将这个角度除以Math.PI得到的是一个百分比，然后乘以灯光的亮度的百分比，你就得到了最后的灯光因子可以用此来改变多边形的基础颜色。

好的，现在所有这些只是为了得到表面的一个新的颜色！在你原有代码中实现它是很简单的。最后是在draw方法中使用它。不是使用基础色，而是使用调整后的颜色，如下：

www.TopSage.com

```
g.beginFill(getAdjustedColor());
```

为了把所有的总结好，这里是Triangle.as和ExtrudedA.as的全部和最终的代码，展示了本章应用中的所有的改变。

首先是Triangle：

```
package
{
    import flash.display.Graphics;

    public class Triangle
    {
        private var pointA:Point3D;
        private var pointB:Point3D;
        private var pointC:Point3D;
        private var color:uint;
        public var light:Light;

        public function Triangle(a:Point3D, b:Point3D,
                                 c:Point3D, color:uint)
        {
            pointA = a;
            pointB = b;
            pointC = c;
            this.color = color;
        }

        public function draw(g:Graphics):void
        {
            if(isBackFace())
            {
                return;
            }
            g.beginFill(getAdjustedColor());
            g.moveTo(pointA.screenX, pointA.screenY);
            g.lineTo(pointB.screenX, pointB.screenY);
            g.lineTo(pointC.screenX, pointC.screenY);
            g.lineTo(pointA.screenX, pointA.screenY);
            g.endFill();
        }

        private function getAdjustedColor():uint
        {
            var red:Number = color >> 16;
            var green:Number = color >> 8 & 0xff;
            var blue:Number =color & 0xff;

            var lightFactor:Number = getLightFactor();

            red *= lightFactor;
            green *= lightFactor;
            blue *= lightFactor;

            return red << 16 | green << 8 | blue;
        }
```

463

www.TopSage.com

大家网 TopSage.com

```
        private function getLightFactor():Number
        {
            var ab:Object = new Object();
            ab.x = pointA.x - pointB.x;
            ab.y = pointA.y - pointB.y;
            ab.z = pointA.z - pointB.z;

            var bc:Object = new Object();
            bc.x = pointB.x - pointC.x;
            bc.y = pointB.y - pointC.y;
            bc.z = pointB.z - pointC.z;

            var norm:Object = new Object();
            norm.x = (ab.y * bc.z) - (ab.z * bc.y);
            norm.y = -((ab.x * bc.z) - (ab.z * bc.x));
            norm.z = (ab.x * bc.y) - (ab.y * bc.x);

            var dotProd:Number = norm.x * light.x +
                                 norm.y * light.y +
                                 norm.z * light.z;
464         var normMag:Number = Math.sqrt(norm.x * norm.x +
                                           norm.y * norm.y +
                                           norm.z * norm.z);

            var lightMag:Number = Math.sqrt(light.x * light.x +
                                            light.y * light.y +
                                            light.z * light.z);

            return (Math.acos(dotProd / (normMag * lightMag)) /
                    Math.PI) * light.brightness;
        }

        private function isBackFace():Boolean
        {
            // 参见 http://www.jurjans.lv/flash/shape.html。
            var cax:Number = pointC.screenX - pointA.screenX;
            var cay:Number = pointC.screenY - pointA.screenY;

            var bcx:Number = pointB.screenX - pointC.screenX;
            var bcy:Number = pointB.screenY - pointC.screenY;

            return cax * bcy > cay * bcx;
        }

        public function get depth():Number
        {
            var zpos:Number = Math.min(pointA.z, pointB.z);
            zpos = Math.min(zpos, pointC.z);
            return zpos;
        }
    }
}
```

www.TopSage.com

然后是ExtrudedA：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class ExtrudedA extends Sprite
    {
        private var points:Array;
        private var triangles:Array;
        private var fl:Number = 250;
        private var vpX:Number = stage.stageWidth / 2;
        private var vpY:Number = stage.stageHeight / 2;

        public function ExtrudedA()
        {
            init();
        }

        private function init():void
        {
            points = new Array();
            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);

            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);
            for(var i:uint = 0; i < points.length; i++)
            {
                points[i].setVanishingPoint(vpX, vpY);
                points[i].setCenter(0, 0, 200);
            }

            triangles = new Array();
            triangles[0] =new Triangle(points[0],   points[1],
                                       points[8],   0xcccccc);
```

www.TopSage.com

```
triangles[1] =new Triangle(points[1],   points[9],
                           points[8],   0xcccccc);
triangles[2] =new Triangle(points[1],   points[2],
                           points[9],   0xcccccc);
triangles[3] =new Triangle(points[2],   points[4],
                           points[9],   0xcccccc);
triangles[4] =new Triangle(points[2],   points[3],
                           points[4],   0xcccccc);
triangles[5] =new Triangle(points[4],   points[5],
                           points[9],   0xcccccc);
triangles[6] =new Triangle(points[9],   points[5],
                           points[10],  0xcccccc);
triangles[7] =new Triangle(points[5],   points[6],
                           points[7],   0xcccccc);
triangles[8] =new Triangle(points[5],   points[7],
                           points[10],  0xcccccc);
triangles[9] =new Triangle(points[0],   points[10],
                           points[7],   0xcccccc);
triangles[10] =new Triangle(points[0],  points[8],
                            points[10], 0xcccccc);
triangles[11] =new Triangle(points[11], points[19],
                            points[12], 0xcccccc);
triangles[12] =new Triangle(points[12], points[19],
                            points[20], 0xcccccc);
triangles[13] =new Triangle(points[12], points[20],
                            points[13], 0xcccccc);
triangles[14] =new Triangle(points[13], points[20],
                            points[15], 0xcccccc);
triangles[15] =new Triangle(points[13], points[15],
                            points[14], 0xcccccc);
triangles[16] =new Triangle(points[15], points[20],
                            points[16], 0xcccccc);
triangles[17] =new Triangle(points[20], points[21],
                            points[16], 0xcccccc);
triangles[18] =new Triangle(points[16], points[18],
                            points[17], 0xcccccc);
triangles[19] =new Triangle(points[16], points[21],
                            points[18], 0xcccccc);
triangles[20] =new Triangle(points[11], points[18],
                            points[21], 0xcccccc);
triangles[21] =new Triangle(points[11], points[21],
                            points[19], 0xcccccc);
triangles[22] =new Triangle(points[0],  points[11],
                            points[1],  0xcccccc);
triangles[23] =new Triangle(points[11], points[12],
                            points[1],  0xcccccc);
triangles[24] =new Triangle(points[1],  points[12],
                            points[2],  0xcccccc);
triangles[25] =new Triangle(points[12], points[13],
                            points[2],  0xcccccc);
triangles[26] =new Triangle(points[3],  points[2],
                            points[14], 0xcccccc);
triangles[27] =new Triangle(points[2],  points[13],
                            points[14], 0xcccccc);
```

www.TopSage.com

```
        triangles[28] =new Triangle(points[4],  points[3],
                                    points[15], 0xcccccc);
        triangles[29] =new Triangle(points[3],  points[14],
                                    points[15], 0xcccccc);
        triangles[30] =new Triangle(points[5],  points[4],
                                    points[16], 0xcccccc);
        triangles[31] =new Triangle(points[4],  points[15],
                                    points[16], 0xcccccc);
        triangles[32] =new Triangle(points[6],  points[5],
                                    points[17], 0xcccccc);
        triangles[33] =new Triangle(points[5],  points[16],
                                    points[17], 0xcccccc);
        triangles[34] =new Triangle(points[7],  points[6],
                                    points[18], 0xcccccc);
        triangles[35] =new Triangle(points[6],  points[17],
                                    points[18], 0xcccccc);
        triangles[36] =new Triangle(points[0],  points[7],
                                    points[11], 0xcccccc);
        triangles[37] =new Triangle(points[7],  points[18],
                                    points[11], 0xcccccc);
        triangles[38] =new Triangle(points[8],  points[9],
                                    points[19], 0xcccccc);
        triangles[39] =new Triangle(points[9],  points[20],
                                    points[19], 0xcccccc);
        triangles[40] =new Triangle(points[9],  points[10],
                                    points[20], 0xcccccc);
        triangles[41] =new Triangle(points[10], points[21],
                                    points[20], 0xcccccc);
        triangles[42] =new Triangle(points[10], points[8],
                                    points[21], 0xcccccc);
        triangles[43] =new Triangle(points[8],  points[19],
                                    points[21], 0xcccccc);

        var light:Light = new Light();
        for(i = 0; i < triangles.length; i++)
        {
            triangles[i].light = light;
        }

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        var angleX:Number = (mouseY - vpY) * .001;
        var angleY:Number = (mouseX - vpX) * .001;
        for(var i:uint = 0; i < points.length; i++)
        {
            var point:Point3D = points[i];
            point.rotateX(angleX);
            point.rotateY(angleY);
        }
        triangles.sortOn("depth", Array.DESCENDING |
                                  Array.NUMERIC);
```

467

468

www.TopSage.com

```
            graphics.clear();
            for(i = 0; i < triangles.length; i++)
            {
                triangles[i].draw(graphics);
            }
        }
    }
}
```

如你所看到的，在文档类中只有两处较小的改变。大多数的工作在Triangle类中。然而，我修改它以使所有三角形的颜色一样，这样我认为灯光的效果会更好一些（如图17-6）。

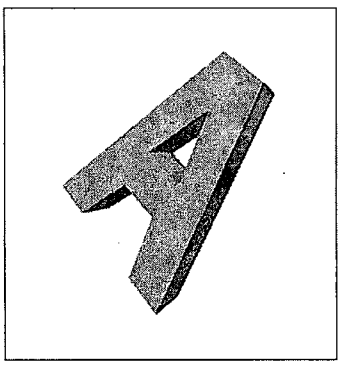


图17-6　使用了背面剔除、深度排序和3D灯光的实体模型

17.4　小结

哇！几页纸却说了这么多的内容！但我认为结果是不可思议的。现在你已经有工具可以制作极为漂亮的3D影片了。毫无疑问，你可以加入你想要的各种变化。例如，在最后一个例子中，灯光是静止的而物体是移动的。可以试着移动灯光。（这只是要改变它的*x*、*y*和/或*z*的位置。）

这就是关于3D讨论的总结。在下一章中，你会看到矩阵数学，它是你经常用来改变物体的
469 缩放和旋转的另一种方法，因此你在3D程序中也会经常看到。

www.TopSage.com

大家网
TopSage.com

Part 5 第五部分

其他技术

本部分内容

- 第 18 章　矩阵数学
- 第 19 章　实用技巧汇集

www.TopSage.com

第18章 矩阵数学

本章内容

- 矩阵基础
- 矩阵运算
- Matrix类

在这一章中，我将不再介绍任何新类型的运动或物理学或渲染图形的方法。我将向你介绍矩阵（matrix），提供一种你之前已经做过的许多事情的另一种方法。

矩阵经常被用在3D系统中，用于旋转、缩放和变换（移动）3D点。它们也经常用于各种2D图形的变换。你可能还记得beginGradientFill方法使用了矩阵来定义位置、大小和旋转渐变。

在本章中，你将学习如何创建一个3D矩阵系统来控制3D中的精灵以及Flash中的一些内置的矩阵用法。

18.1 矩阵基础

简单地说，一个矩阵是一组数字网格。它可以有一个或更多水平行和一个或更多的垂直列。图18-1显示的是一些矩阵。

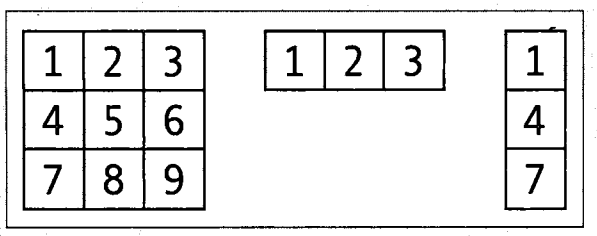


图18-1 3×3矩阵、1×3矩阵和3×1矩阵

任何指定的矩阵通常都会使用某个变量来表示，如使用M。要引用矩阵中的某一个具体的单元格，你在变量名的右下角加上它行和列的数字来表示。例如，如果图18-1中的3×3矩阵被称为是M，那么$M_{2,3}$等于6，因为它指的是第2行，第3列。

www.TopSage.com

矩阵的单元格不仅可以是数字，也可以是公式和变量。如果你用过电子表格，那基本上就是一个大的矩阵。你可以使用一个单元格来保存一列的和，而另外的单元格使用一些比例乘以和然后把结果放在另一个单元格中，如此这样。因此，你会发现矩阵是非常有用的。

18.2 矩阵运算

现在，电子表格是一种自由型的矩阵，我们将要处理的矩阵是更加结构化的，而且有各种与之相关的规则以决定我们能做什么和怎样去做它。

我所见过的大多数关于矩阵的教学资料在介绍矩阵时大约有两种方法。第1种方法详细描述了如何运算，你可以学习到规则，但你并不知道你为什么要那么做或结果代表什么含义。它就像是正在一种漂亮的模式下玩一个安排数字的游戏。 474

第2种方法是详细描述矩阵的内容而忽略了它的运算，而一些类似“然后你只需要将这两个矩阵相乘然后就得到这个……”的粗略介绍使读者不知道这个相乘是如何完成的。

为了确保你可以理解矩阵的工作，我选择同时使用这两种方法，开始介绍一些包含有意义的值的矩阵，然后描述如何去操作它们。

18.2.1 矩阵加法

矩阵中较常见的用法是控制3D中的点。一个3D点包括*x*、*y*和*z*位置的数值，我们可以很容易地把它视为一个1×3的矩阵，如下：

```
x  y  z
```

现在假设你想在空间中移动这个点，也称为变换（translate）这个点。你需要知道的是在每一个轴上移动多少。你可以把这放在一个变换矩阵（translation matrix）中。这是另一个1×3矩阵，如下：

```
dx  dy  dz
```

这里，dx、dy和dz是在每个轴上移动的距离。现在你需要以某种方式将变换矩阵应用到点的矩阵中。这里使用的是矩阵加法，它非常简单。只需要将每个单元格对应相加并将结果放在一个新的矩阵的单元格中。显然，要将两个矩阵相加，它们需要有相同的大小。对于变换，你这样做：

```
x  y  z  +  dx  dy  dz  = (x + dx)  (y + dy)  (z + dz)
```

这里，结果矩阵可能被称为x1,y1,z1，并包含了变换后的点的新位置。让我们试一下真实的数值。假设你的点在x,y,z轴上的100,50,70。你想将它移动–10,20,–35。如下：

```
100  50  75  +  -10  20  -35  =  (100 - 10)  (50 + 20)  (75 - 35)
```

因此，当你执行相加时，得到的是90,70,40作为点的新位置。相当简单，不是吗？你可能已经注意到相关的速度，每一个轴上的速度被加到每个轴的位置上。理论上讲它的处理方法是相同的，只是看起来有一点不同。

如果你有一个很大的矩阵，可能也会使用这样的方法。我不会在这里计算任何大于3×1的矩阵，但是我给你一个抽象的例子：

www.TopSage.com

```
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)
```

这就是所有你需要知道的矩阵加法的内容。当我介绍了矩阵乘法之后，我将向你展示如何与
475 一些实际的函数结合来用在一个基于矩阵的3D引擎中。

18.2.2 矩阵乘法

对于3D变换更常见的是*矩阵乘法*（matrix multiplication），它通常用于缩放和旋转。在这本书中我们并没有真正使用3D缩放，因为示例中使用的或者是点，它不能被缩放，或者是精灵，它没有任何3D“厚度”那么只能在2个轴上缩放。当然，你可以创建一个可以缩放整个3D实体模型的更复杂的引擎。然后你还需要写一个附加功能来改变3D的点形成一个基于新大小的3D模型。这已经超出了我们这里所说的范围，但因为缩放很简单并可以清晰地说明矩阵的乘法，我将通过一个示例来说明。

1. 使用矩阵缩放

首先你需要知道一个物体已存在的宽、高和深度——换句话说，就是在3个轴向上大小的度量。这当然是一个3×5的矩阵：

```
w  h  d
```

可能你已经意识到了，w、h和d代表着宽、高和深度。接下来你需要一个像这样的缩放矩阵：

```
sx   0   0
 0  sy   0
 0   0  sz
```

这里，sx、sy和sz是特定轴上缩放的百分比。这些是分数或小数，那么1.0就是100%，2.0就是200%，0.5是50%，等等。你一会将看到矩阵为什么要这样分布。

有一件事情你需要知道，在矩阵乘法中，为了让两个矩阵相乘，第1个矩阵的列数必须与第2个矩阵的行数相同。只要这些条件可以满足，第1个可以有任意数量的行数，而第2个可以有任意数量的列数，在这个例子中，你是对的，因为第1个矩阵有3列(w,h,d)，而缩放矩阵有3行。

那么，如何相乘呢？让我们继续做它看你是否会看到下面的模式：

```
                sx   0   0
w  h  d   *      0  sy   0
                 0   0  sz
```

这产生了下面的矩阵作为结果：

```
(w*sx + h*0  + d*0)  (w*0  + h*sy + d*0)  (w*0  + h*0  + d*sz)
```

当你去掉所有的0后，你得到下面的结果：

476

```
(w*sx)  (h*sy)  (d*sz)
```

这看上去有很好的逻辑，因为你在使用x的比例因子乘以宽，使用y的比例因子乘以高，还使用z的比例因子乘以深度。但是，我们到底做了什么呢？所有的0是一种偶然现象，因此让我们把它抽象为更清晰一点的模式。

www.TopSage.com

```
            a  b  c
u  v  w  *  d  e  f
            g  h  i
```

现在你可以看到模式以这种结果出现：

```
(u*a + v*d  + w*g)  (u*b  + v*e + w*h)  (u*c  + v*f  + w*i)
```

你可以看到你将第1个矩阵的第1行(u,v,w)分别乘以第2个矩阵每一行的第1个元素(a,d,g)。将它们加起来就得到了结果的第1行的第1个元素。同样第2个矩阵的第2列(b,e,h)也做相同的运算，则得到第2列的结果。

如果第1个矩阵要多于一行，你需要重复去做第2行，它得到的是第2行的结果：

```
u  v  w       a  b  c
x  y  z  *    d  e  f
              g  h  i
```

最后得到的是下面的3×2的矩阵作为结果：

```
(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)
```

现在让我们看一些矩阵乘法的真实应用——坐标旋转。希望这个缩放例子可以让你更清晰地知道我们在做什么。

2. 使用矩阵的坐标旋转

首先，你需要知道你的3D点矩阵：

```
x  y  z
```

这将保存你想要旋转的点的坐标。当然，现在你需要一个旋转矩阵。如你所知道的，你可以在3个轴中的任何一个轴上旋转。你可以为任何一个轴上的旋转创建一个单独的矩阵。让我们以*x*轴的矩阵旋转开始：

```
1    0    0
0   cos  sin
0  -sin  cos
```

现在这里有一些正弦和余弦，并且很明显的问题可能是“哪些值的正弦或余弦？”好，它是你要旋转角度的正弦或余弦值。如果你将那个点旋转45度，它将是45度的正弦和余弦值。(当然，在代码中，你需要使用弧度。)

477

现在，让我们使用这个矩阵与3D点的矩阵相乘并观察你得到的结果。

```
            1    0    0
x  y  z  *  0   cos  sin
            0  -sin  cos
```

对于此，你得到如下：

```
(x*1 + y*0  + z*0)  (x*0  + y*cos - z*sin)  (x*0  + y*sin  + z*cos)
```

整理之后得到的是：

```
(x)  (y*cos - z*sin)  (z*cos + y*sin)
```

这个可以粗略地变换成下面的Actionscript表示：

www.TopSage.com

大家网 TopSage.com

```
x = x;
y = Math.cos(angle) * y - Math.sin(angle) * z;
z = Math.cos(angle) * z + Math.sin(angle) * y;
```

现在，如果你想一下第10章中我讨论坐标旋转的地方，你将会发现它正好是我执行*x*轴旋转的公式。这并不是很令人惊奇，因为矩阵数学只不过是看待和组织各种公式和等式的不同方法。

从这里开始，你可以很容易创建一个用于*y*轴旋转的矩阵：

```
 cos  0  sin
  0   1   0
-sin  0  cos
```

最后，*z*轴的旋转矩阵是：

```
 cos  sin  0
-sin  cos  0
  0    0   1
```

继续将它与x,y,z矩阵相乘是一个很好的实践并且可以验证你得到的公式是否与我们在第10章所学到的两个轴上的坐标旋转的公式一致。

3. 使用矩阵的编码

好，你已经知道了足够多的基础来将这些内容放在代码中。你将重新使用和修改第15章的RotateXY.as类，那么打开它，这个类已经有了一个rotateX和rotateY方法来执行3D坐标旋转。我们将改变它们的工作方式以使它们使用矩阵。

以rotateX函数开始。先获取到小球的x,y,z坐标，将它们放在一个1×3矩阵中，然后在给定的角的基础上创建一个*x*旋转矩阵。这些矩阵都是数组形式。最后，这两个矩阵需要使用matrixMultiplay函数来进行相乘，这个函数也需要创建！相乘后的结果将是另一个数组，因此你必须将这些值指定回小球的*x*、*y*和*z*坐标。下面是新版本的方法：

478

```
private function rotateX(ball:Ball3D, angleX:Number):void
{
    var position:Array = [ball.xpos, ball.ypos, ball.zpos];

    var sin:Number = Math.sin(angleX);
    var cos:Number = Math.cos(angleX);
    var xRotMatrix:Array = new Array();
    xRotMatrix[0] = [1,    0,   0];
    xRotMatrix[1] = [0,  cos, sin];
    xRotMatrix[2] = [0, -sin, cos]

    var result:Array = matrixMultiply(position, xRotMatrix);
    ball.xpos = result[0];
    ball.ypos = result[1];
    ball.zpos = result[2];
}
```

这里是矩阵相乘的方法：

```
private function matrixMultiply(matrixA:Array, matrixB:Array):Array
{
      var result:Array = new Array();
      result[0] = matrixA[0] * matrixB[0][0] +
```

www.TopSage.com

```
                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;
    }
```

现在这个矩阵相乘函数是使用硬编码的方式将1×3矩阵与3×3矩阵相乘，因为这就是你在每个例子中要做的。你可以使用for循环来制作一个动态的函数以使它可以处理任意大小的矩阵，但是这里还是让我们保持尽量简单。

最后，创建rotateY函数。如果你理解了rotateX函数，这个就很容易了。只需要创建一个y轴的旋转矩阵而不是x轴旋转矩阵。

```
private function rotateY(ball:Ball3D, angleY:Number):void
{
    var position:Array = [ball.xpos, ball.ypos, ball.zpos];

    var sin:Number = Math.sin(angleY);
    var cos:Number = Math.cos(angleY);
    var yRotMatrix:Array = new Array();

    yRotMatrix[0] = [ cos, 0, sin];
    yRotMatrix[1] = [   0, 1,   0];
    yRotMatrix[2] = [-sin, 0, cos]

    var result:Array = matrixMultiply(position, yRotMatrix);
    ball.xpos = result[0];
    ball.ypos = result[1];
    ball.zpos = result[2];
}
```

479

这就是它的代码。如果你想，可以创建一个rotateZ函数。我们其实不需要它，所以我把它留给你来做。

现在如果你运行RotateXY.as并且与第15章版本相比较，它们应是一模一样的。在AS 2中，我发现没有使用矩阵的版本运行得更光滑一些。原因是你已经对3D的旋转、缩放等进行了大量的数学计算。当使用矩阵数学时，你实际上是做了许多额外的计算。当你开始对矩阵相乘时，实际做了4 次乘以0的运算，并将这4个结果加到其他数上。这就是说这8次的数学运算根本没有什么用处。再将它乘以50个物体和每一帧的2次旋转，那么你将在每帧上得到800个多余的运算！在Flash CS 3和AS 3中两个版本的运行并没有明显区别。但我确信当加入越来越多的物体时，你将会到达一个点，此时过量的计算会夺去它的速度。另外，我给出的矩阵代码都是很基础的，你可以对它进行优化，以获得更高的性能。

即使不在3D中使用矩阵，你也会发现矩阵在其他方面的用处，我将在下一节讲解它。

www.TopSage.com

18.3 Matrix类

如刚才提到过的，掌握矩阵的一个很好的理由是作为ActionScript的核心类它可以用在几个地方。实际上，在Flash中甚至有一个内置的Matrix类。如果你查看flash.geom.Matrix类的Flash帮助文件，会发现它们相当清晰和有用。如果现在你已经理解了目前为止本章所讲的内容，那么掌握这些内容应该是没有问题的。因为帮助文件已经写得很好了，我不想再花时间去重讲它，但是我将给出几个总结性的例子。

一个矩阵的主要用处是用来改变（旋转、缩放和变换）一个显示对象。任何显示对象（精灵、影片剪辑、文本域等等）都有一个属性称为transform。它是flash.geom.Transform类的一个实例，并且它包含另一个属性称为matrix。如果你创建一个Matrix类的实例并且把它指定给transform.matrix属性，它将改变这个显示对象的样子、大小或位置。很快你会看到一些具体的
480 例子。

基本上，在Matrix类中的矩阵是一个3×3的设置，如下：

```
a  b  tx
c  d  ty
u  v  w
```

u、v和w自动被设置为0、0和1并且用于内部的使用。它们是不可改变的，因此你不需要担心它们。（更多关于它们的解释可参看帮助文件。）可以使用下面的语法创建一个新的Matrix：

```
import flash.geom.Matrix;
var myMatrix:Matrix = new Matrix(a, b, c, d, tx, ty);
```

那么所有这些字母是什么意思呢？好，tx和ty是非常容易的。它们控制着显示对象*x*和*y*轴上的位置。a、b、c和d有一点复杂是因为它们之间互相依赖。如果设置b和c为0，那么可以使用a和d在*x*和*y*轴上缩放物体。如果设置a和d为1，那么可以分别使用b和c在*y*和*x*轴上斜切物体。最后，我确信你可能会以一种非常熟悉的方式一起使用a、b、c和d。在这个例子中，它们像这样分布：

```
 cos  sin  tx
-sin  cos  ty
  u    v   w
```

当然，你可以看到它包含了一个旋转矩阵，并且它确实用来旋转物体。很自然，在这个例子中的cos和sin是一个指定角（弧度）的余弦和正弦，这个角度是你想要旋转精灵的角度。让我们试一下它。

这儿你可以看到MatrixRotate.as文件，它创建了一个简单的红色盒子精灵。然后设置enterFrame事件处理器，所有代码如下：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Matrix;

    public class MatrixRotate extends Sprite
    {
```

www.TopSage.com

```
        private var angle:Number = 0;
        private var box:Sprite;

        public function MatrixRotate()
        {
            init();
        }

        private function init():void
        {
            box = new Sprite();
            box.graphics.beginFill(0xff0000);
            box.graphics.drawRect(-50, -50, 100, 100);
            box.graphics.endFill();
            addChild(box);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            angle += .05;
            var cos:Number = Math.cos(angle);
            var sin:Number = Math.sin(angle);
            box.transform.matrix = new Matrix(cos, sin,
                                              -sin, cos,
                                               stage.stageWidth / 2,
                                               stage.stageHeight / 2);
        }
    }
}
```

481

这里你有一个每帧都增加的角变量。代码计算出这个角的正弦和余弦并且将它们放到一个新的Matrix对象，这种方式主要用于旋转。我已经基于舞台的宽和高应用了变换，精灵的位置与舞台中心对齐。新的矩阵被指定给精灵的transform.matrix属性。测试它你会得到一个旋转的盒子。

现在，你可能会在想为什么不是改变精灵的rotation属性。对于这样简单的例子这样做是对的，而且可能会是一个更简单的解决方案。但是可能有很多情况你需要处理许多角度、弧度、正弦和余弦，此时使用矩阵真地会容易许多，只需要指定一个矩阵而不需要把所有的东西都转换回度数然后再改变rotation。

但对于其他更现实的演示，让我们试一下斜切。**斜切**（**skewing**）意味着让某个东西在某个轴上倾斜以使一部分移向这个方向，而另一部分移向另一个方向。斜体字母是一个斜切的例子。字母的顶端向右而底端向左。这在Flash中是一个很复杂的东西，但是使用Matrix类它是很简单的。如我之前说到的，你设置矩阵的a和d为1。b属性就是*y*轴上斜切的大小，而c控制着*x*轴上的斜切。让我们先来试一个*x*斜切。在SkewX.as中，我几乎使用了与上一个例子中相同的设置，但是在onEnterFrame中改变了矩阵的创建方式。

```
private function onEnterFrame(event:Event):void
{
```

www.TopSage.com

```
    var skewX:Number = (mouseX - stage.stageWidth / 2) * .01;
    box.transform.matrix = new Matrix(1,     0,
                                      skewX, 1,
                                      stage.stageWidth / 2,
                                      stage.stageHeight / 2);
}
```

这里我设置了与鼠标*x*位置相关的skewX变量，偏离舞台中心的偏移值。然后将它乘以0.01来
482 保持斜切在一个可操纵的范围内，并在矩阵中使用这个变量。

现在，当测试这个影片时，你将看到如何斜切整个精灵，如图18-2所示。虽然这在学习Matrix之前也是可能的，但是如果有人试验过它，向他们展示一下过程代码，并且看他们开始羡慕的样子！或者，如果你自己试过了，你就已经知道我的意思了。

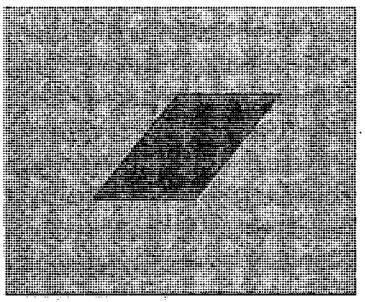
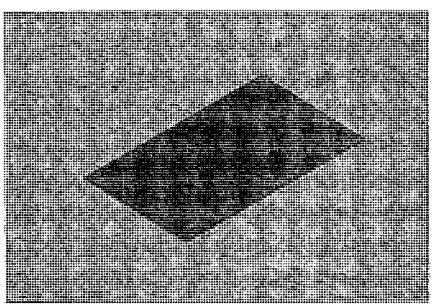


图18-2　精灵在*x*轴上斜切

在SkewXY中，我针对*y*轴做了同样的事情：

```
private function onEnterFrame(event:Event):void
{
    var skewX:Number = (mouseX - stage.stageWidth / 2) * .01;
    var skewY:Number = (mouseY - stage.stageHeight / 2) * .01;
    box.transform.matrix = new Matrix(1,     skewY,
                                      skewX, 1,
                                      stage.stageWidth / 2,
                                      stage.stageHeight / 2);
}
```

图18-3出示了如何在两个轴上斜切精灵。



483

图18-3　精灵在两个轴上的斜切

www.TopSage.com

这么容易就做出这样的效果是非常令人吃惊的。如果你不确定在哪里可以用上这样的效果，我将要告诉你斜切对于假3D是非常有用的。当你在上一个例子中移动鼠标时，你可能已经看到它是如何表现一些透视的，好像形体正在倾斜而且旋转。它并不是准确的3D，但它可能用于一些相当有说服力的效果。在网上也有一些教程向你展示了如何制作这种使用斜切的假3D。使用Matrix类很可能将示例中的代码减少一半。

另外，确保你已经读过了Matrix类的帮助文件，因为那里有许多好的思路。还有，它不光是在AS 3的矩阵工具类中，你可能也需要看一下ColorMatrixFilter、ConvolutionFilter、各种drawing API填充和渐变方法，还有flash.geom.Transform类。在那儿有更丰富的矩阵。

18.4 小结

我已经讲解了矩阵是什么、如何应用它们和合并它们等基础内容，并使用它们创建了一些非常酷的效果。现在在你的头脑中已经有了这些概念，你要准备应用矩阵提供的强大功能了，希望当你在AS 3的内置方法中遇到它们时，不要害怕使用它们，因为你确实能够使用它们了。 484

www.TopSage.com

第19章

实用技巧汇集

本章内容

- 布朗（随机）运动
- 随机分布
- 基于间隔和定时器的动画
- 相同质量物体之间的碰撞
- 声音集成
- 有用的公式

好，你已经学到了本书的最后一章。这一章我主要是想告诉你一些小的内容，但这些内容并不是放在哪儿都合适，否则会使你不能专心学习其他章的内容。

我也打算将我们得出的且列在每章的结尾的所有公式重新组织起来，以备日后参考。

因为这些不同的课题只是一些随机的内容（尽管是一些有用的内容），所以我无法进行太多的组织。因此每一节将是一个独立的单元。我们马上开始吧。

19.1 布朗（随机）运动

首先说一点历史。一天，一个名叫布朗·罗伯特的植物学家正在观察悬浮于水中的一些花粉，他发现它们是随机移动的。尽管在水中并没有什么水流或运动出现，但这些小的颗粒永远不会停下来。他在灰尘颗粒中发现了同样的现象。尽管他并不清楚为什么会这样，而且他或其他人在几十年间都没法给出恰当的解释，但是无论怎样这种现象是以他的名字命名的——只是因为他注意到了它。

布朗运动在今天可以解释为是水中的大量水分子不停地运动，即使水面表现出来的是平静的。水分子会与花粉和灰尘颗粒发生碰撞，并传递动量。即使一个灰尘比一个水分子重100万倍，每次碰撞并不能产生多大效果。但是当你每秒钟有几百万次的碰撞时，累积的效果就可观了。

现在一些水分子可能碰到一边，而一些碰到另一边。总的来说它们是比较平均的。但在这期间，你想看到波动，假设说，有更多的碰撞在左边，它足可以让粒子向右边移动。接着更多的粒子可能向下碰撞，而且粒子就会开始向上移动。另外，它们最终也是平均的，因此你通常不会在哪个方向上得到更多的动量。你得到的只是随机的游离运动。

www.TopSage.com

在Flash中你可以很容易地模拟它。在每一帧上，只需计算要加入到移动物体的x和y速度上的随机数。随机数应该计算为正值或负值，并且通常是很小的，假设在–0.1到+0.1范围之中，你可以这样做：

```
vx += Math.random() * 0.2 - 0.1;
vy += Math.random() * 0.2 - 0.1;
```

将随机数乘以0.2就会得到一个从0.0到0.2的数，再减去0.1就得到–0.1到0.1之间的数。加入一些摩擦力也很重要，否则速度会不断增加，许多东西会看上去不自然。在Brownian1.as中，我创建了50个粒子并且让它们做布朗运动。粒子是我们经常用的Ball类的实例，只是让它更小并且是黑色。这里是代码：

488

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Brownian1 extends Sprite
    {
        private var numDots:uint = 50;
        private var friction:Number = 0.95;
        private var dots:Array;

        public function Brownian1()
        {
            init();
        }

        private function init():void
        {
            dots = new Array();
            for(var i:uint = 0; i < numDots; i++)
            {
                var dot:Ball = new Ball(1, 0);
                dot.x = Math.random() * stage.stageWidth;
                dot.y = Math.random() * stage.stageHeight;
                dot.vx = 0;
                dot.vy = 0;
                addChild(dot);
                dots.push(dot);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            for(var i:uint = 0; i <numDots; i++)
            {
                var dot:Ball = dots[i];
                dot.vx += Math.random() * 0.2 - 0.1;
                dot.vy += Math.random() * 0.2 - 0.1;
                dot.x += dot.vx;
                dot.y += dot.vy;
```

www.TopSage.com

```
                dot.vx *= friction;
                dot.vy *= friction;

                if(dot.x > stage.stageWidth)
                {
                    dot.x = 0;
                }
                else if(dot.x < 0)
                {
                    dot.x = stage.stageWidth;
                }
                if(dot.y > stage.stageHeight)
                {
                    dot.y = 0;
                }
                else if(dot.y < 0)
                {
                    dot.y = stage.stageHeight;
                }
            }
        }
    }
}
```

489

大多数代码对你来说再熟不过了，相关的代码使用粗体表示。

图19-1显示了运行代码后显示在屏幕上的结果。

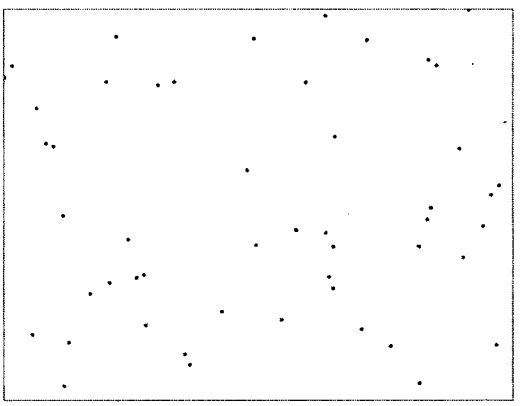


图19-1　布朗运动

在Brownian2.as中，我将点的数量减少到了20。然后我加入了下面的代码行

```
graphics.lineStyle(0, 0, 0.5);
```

490 到init方法中，并且改变了onEnterFrame方法的第1部分以包括一些绘制代码。

```
private function onEnterFrame(event:Event):void
{
    for(var i:uint = 0; i < numDots; i++)
    {
        var dot:Ball = dots[i];
```

www.TopSage.com

```
graphics.moveTo(dot.x, dot.y);
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;
graphics.lineTo(dot.x, dot.y);
```

它从每一个点移动之前的位置到移动后的位置绘制了一条线。因此它绘制的是它的路径，如图19-2所示。如果你查找“布朗运动”这个术语，你会经常看到类似这样的图。

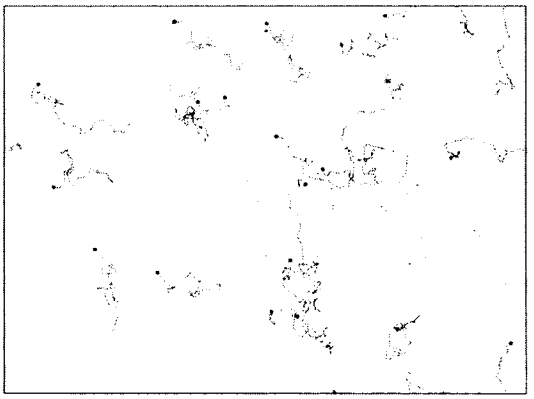


图19-2 拖尾的布朗运动

布朗运动对于那些没有任何目标的浮动，没有任何外力的移动是有用的。你也可以将它加入到一个施加了某个其他运动的精灵上，给它一种随机的感觉。一个例子就是快速移动的苍蝇或蜜蜂。你可以让它沿着某条路径移动，但是加入一些随机运动会让它看上去更真实。 491

19.2 随机分布

有时，你可能想创建许多物体并且将它们放置在随机的位置上。你已经通过本书的学习看到过多次怎么做了，但是这里我的方法有些不同，不同的结果一会儿会展示给你。

19.2.1 方形分布

如果你想让物体随机分布在整个舞台上，这是非常简单的。只需要选取一个最大值可达舞台的宽的随机数x，选取一个最大值可达舞台的高的随机数y，实际上，你在之前的小节中做过：

```
for(var i:uint = 0; i < numDots; i++)
{
    var dot:Ball = new Ball(1, 0);
    dot.x = Math.random() * stage.stageWidth;
    dot.y = Math.random() * stage.stageHeight;
    ...
```

但是假设说你想将这些点集中在舞台中间某一区域内，假设距离中心的左右两边、顶端或底端都是100像素。你可能会像下面这样做，文件为Random1.as：

www.TopSage.com

大家网 TopSage.com

```
package {
    import flash.display.Sprite;

    public class Random1 extends Sprite
    {
        private var numDots:uint = 50;

        public function Random1()
        {
            init();
        }

        private function init():void
        {

            for(var i:uint = 0; i < numDots; i++)
            {
                var dot:Ball = new Ball(1, 0);
                dot.x = stage.stageWidth / 2 +
                        Math.random() * 200 - 100;
                dot.y = stage.stageHeight / 2 +
                        Math.random() * 200 - 100;
                addChild(dot);
            }
        }
    }
}
```

492

这创建了一个从–100到100的随机数并将它们加到舞台的中心点，那么所有的点在每个轴上都不会超过距离屏幕中心100像素的范围。图19-3是你所得到的结果。

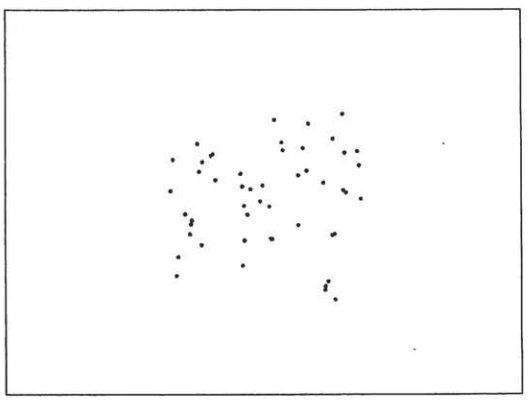


图19-3　随机放置点

这个结果并不坏，但是如果你放入了更多的点（300）并且区域减小到了100×100，你可能会注意到一些奇怪的事情开始发生，这里是Random2.as的代码：

```
package {
    import flash.display.Sprite;

    public class Random2 extends Sprite
```

www.TopSage.com

```
    {
        private var numDots:uint = 300;

        public function Random2()
        {
            init();
        }

        private function init():void
        {
            for(var i:uint = 0; i < numDots; i++)
            {
                var dot:Ball = new Ball(1, 0);
                dot.x = stage.stageWidth / 2 +
                        Math.random() * 100 - 50;
                dot.y = stage.stageHeight / 2 +
                        Math.random() * 100 - 50;
                addChild(dot);
            }
        }
    }
}
```

493

那么图19-4显示了你得到的结果。

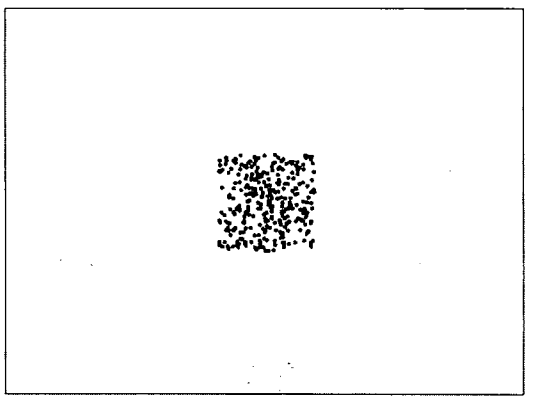


图19-4 这个方法开始形成一个方形。看上去并没有太多的随机性

如你所看到的，这些点开始形成一个方形。可能这是可以的，但如果你想要制作某种爆炸或星系或一些有序的内容，一个方形看上去并不自然。如果一个方形分布并不是你想要的，试着进入下一个技术。

19.2.2 圆形分布

虽然比方形分布复杂了一点点，圆形分布其实并不难做。首先你需要知道圆的半径。让我们设它为50，与上一个例子相符合。这将是一个点可以放置的距离中心最远的位置。你将使用一个从0 到你使用的半径之间的随机值来放置每一个点。那么你需要选择从0 到PI*2弧度（360度）之间的一个随机的角度，并且使用三角学来计算出点的*x*和*y*位置。这是来自Random3.as的代码：

www.TopSage.com

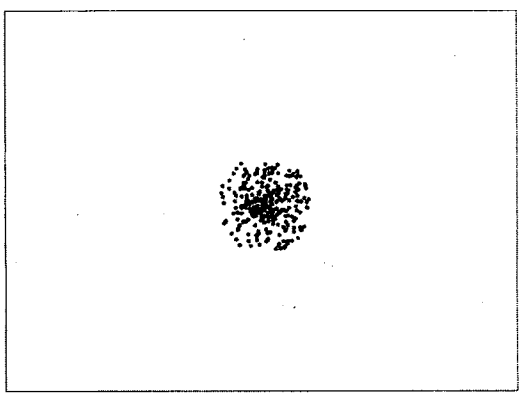
```
package {
    import flash.display.Sprite;

    public class Random3 extends Sprite
    {
        private var numDots:uint = 300;
494     private var maxRadius:Number = 50;

        public function Random3()
        {
            init();
        }

        private function init():void
        {
            for(var i:uint = 0; i < numDots; i++)
            {
                var dot:Ball = new Ball(1, 0);
                var radius:Number = Math.random() * maxRadius;
                var angle:Number = Math.random() *
                                   (Math.PI * 2);
                dot.x = stage.stageWidth / 2 +
                        Math.cos(angle) * radius;
                dot.y = stage.stageHeight / 2 +
                        Math.sin(angle) * radius;
                addChild(dot);
            }
        }
    }
}
```

那你会得到如图19-5所示的图形。



495　图19-5　圆形随机分布

对于之前提到的多数类型的应用来说，这个更加自然。然而，你可能会注意到点更愿意聚集到圆的中心点周围。这是因为分布是沿着半径的，意味着中心与边缘有相同数量的点，但因为中心位置的空间比较小，所以它们看上去很挤。

另外，这可能对一些应用来说是好的，但Sean O' Shell（www.pixelwit.com）曾挑战过我，提

www.TopSage.com

出了一种制作点在圆内实现均匀分布的方法。我不得不承认我是较笨的，而且我尝试的方法是比较复杂的。最后他给了我一个非常简单的解决方案。如你在Random4.as中所看到的：

```
package {
    import flash.display.Sprite;

    public class Random4 extends Sprite
    {
        private var numDots:uint = 300;
        private var maxRadius:Number = 50;

        public function Random4()
        {
            init();
        }

        private function init():void
        {
            for(var i:uint = 0; i < numDots; i++)
            {
                var dot:Ball = new Ball(1, 0);
                var radius:Number = Math.sqrt(Math.random()) *
                                    maxRadius;
                var angle:Number = Math.random() *
                                   (Math.PI * 2);
                dot.x = stage.stageWidth / 2 +
                        Math.cos(angle) * radius;
                dot.y = stage.stageHeight / 2 +
                        Math.sin(angle) * radius;
                addChild(dot);
            }
        }
    }
}
```

通过对随机数开平方，它就有了一个更接近1而远离0的偏移值，这样刚好能光滑分布。你可以在图19-6中看到结果。 496

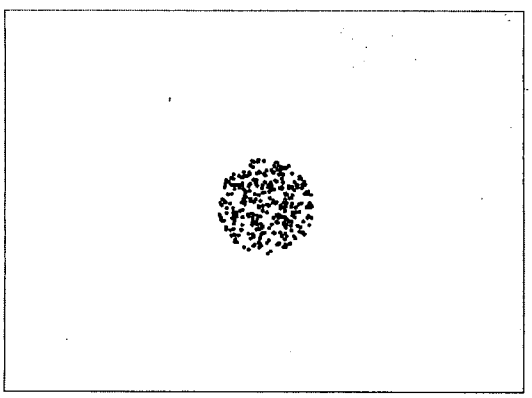


图19-6 一个更光滑的分布

www.TopSage.com

19.2.3 偏移分布

最后，你可能想给这些随机的物体一些自由的范围，而且想让它们显示在中心区域。你需要找到一些边界，但是越接近中心，点的数量越多。它有点类似于第1个圆形分布的例子，只是应用在矩形区域中。

要完成它需要为每个位置产生多个随机数，然后对它们求平均来得到最终结果。例如，假设舞台是500像素宽。如果你只用一个随机数为每一个物体生成一个*x*位置，每一个物体都有相同的机会分布在那个范围内的任何位置上。但是如果你生成两个从0到500的随机数，然后再计算平均，那么它们出现在中间位置的几率要高于出现在边上的几率。

让我们再深入一步来观察它。两个随机数都有可能处于一个“高”范围内，假设是从300到500。同样它们也都可能处于一个“低”范围内，从0到200。但是几率更高的情况是一个高和一个低，或一个是中等而另外一个是高或低，或两个都是中等。所有这些情况平均起来的结果将使大多数的点趋于中等范围。

好的，让我们来看一下代码。和往常一样，你开始时在一维空间上。这里是它的代码（你可以在Random5.as中找到）：

```
package {
    import flash.display.Sprite;

    public class Random5 extends Sprite
    {
        private var numDots:uint = 300;

        public function Random5()
        {
            init();
        }

        private function init():void
        {
            for(var i:uint = 0; i < numDots; i++)
            {
                var dot:Ball = new Ball(1, 0);
                var x1:Number = Math.random() * stage.stageWidth;
                var x2:Number = Math.random() * stage.stageWidth;
                dot.x = (x1 + x2) / 2;
                dot.y = stage.stageHeight / 2 +
                        Math.random() * 50 - 25;
                addChild(dot);
            }
        }
    }
}
```

497

这里你生成了两个随机数x1和x2，然后设置点的x位置为它们的平均值。而y位置只是随机地在中心附近。你得到的结果如图19-7所示。

www.TopSage.com

TopSage.com

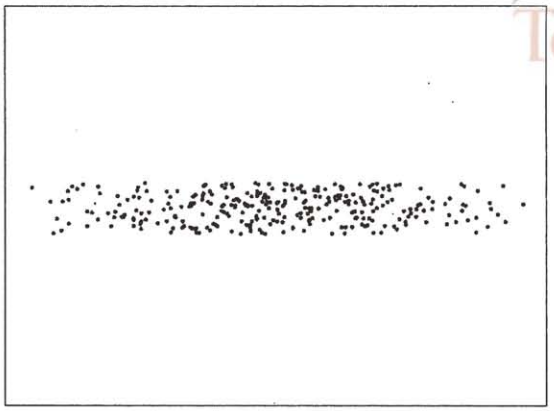


图19-7 使用1次迭代的偏移分布

498

这儿的效果并不是很明显，但是你可以看出有许多点聚集在中心位置，而边缘则留有更多的空间。创建更多的随机数然后取它们的平均值指定给点的x位置，这样会使效果更明显。你需要将它移到一个for循环中来生成动态（Random6.as）：

```
package {
    import flash.display.Sprite;

    public class Random6 extends Sprite
    {
        private var numDots:uint = 300;
        private var maxRadius:Number = 50;
        private var iterations:uint = 6;

        public function Random6()
        {
            init();
        }

        private function init():void
        {
            for(var i:uint = 0; i < numDots; i++)
            {
                var dot:Ball = new Ball(1, 0);
                var xpos:Number = 0;
                for(var j:uint = 0; j < iterations; j++)
                {
                    xpos += Math.random() * stage.stageWidth;
                }
                dot.x = xpos / iterations;
                dot.y = stage.stageHeight / 2 +
                        Math.random() * 50 - 25;
                addChild(dot);
            }
        }
    }
}
```

www.TopSage.com

这里的iterations变量控制着对多少个数取平均。你从xpos为0开始，然后加入每个随机数。
499 最后，除以迭代的次数得到最终的值。这样你得出如图19-8的结果。

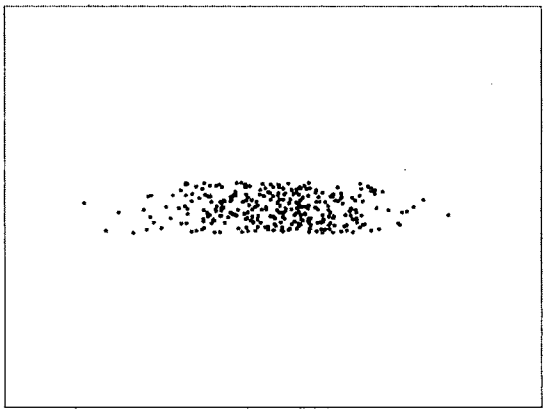


图19-8　使用6次迭代的偏移分布

现在对于*y*轴做同样的事情是很容易的，Random7.as如下：

```
package {
    import flash.display.Sprite;

    public class Random7 extends Sprite
    {
        private var numDots:uint = 300;
        private var maxRadius:Number = 50;
        private var iterations:uint = 6;

        public function Random7()
        {
            init();
        }

        private function init():void
        {
            for(var i:uint = 0; i < numDots; i++)
            {
                var dot:Ball = new Ball(1, 0);
                var xpos:Number = 0;
                for(var j:uint = 0; j < iterations; j++)
                {
                    xpos += Math.random() * stage.stageWidth;
                }
                dot.x = xpos / iterations;
                var ypos:Number = 0;
                for(j = 0; j < iterations; j++)
                {
                    ypos += Math.random() * stage.stageHeight;
                }
                dot.y = ypos / iterations;
```

500

www.TopSage.com

```
                addChild(dot);
            }
        }
    }
}
```

这些代码得出如图19-9所示的分布结果。

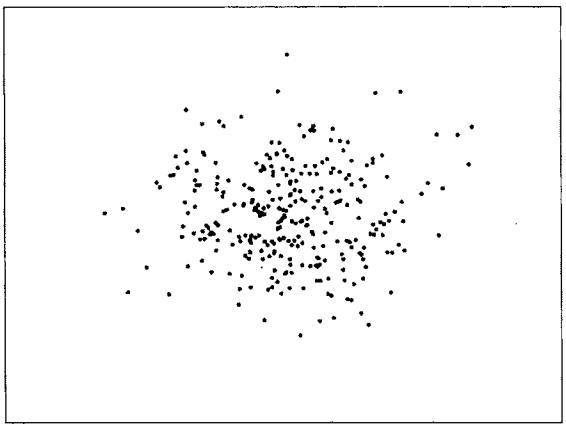


图19-9 二维偏移分布

19.3 基于定时器和时间的动画

到目前为止在本书中的所有示例中，动画都是采用将运动代码放置在onEnterFrame方法中并且将其指定为enterFrame事件的处理器来完成的。我一直认为这是最简单的方法，因为帧的概念在Flash中已经根深蒂固了，它始终在那里随时准备为你使用；并且我猜大多人都是这样，它只是一个老的习惯。

然而，许多人，特别是那些没有Flash编程环境的人，使用这种模式并不舒服。对于他们来说，一个基于定时器的动画模式（使用间隔或定时器）看起来可以提供更准确的动画速度的控制。因此我们来看一下它。

501

我们来看一下什么是我们称为的“基于时间的动画”——一种可能使用帧或定时器的技术。

19.3.1 基于定时器的动画

基于定时器的动画（timer-based animation）使用的关键类是flash.utils.Timer类。除此之外你还需要flash.events.TimerEvent类。

使用定时器来动画实际上与使用enterFrame并没有任何区别。它只是需要你创建一个定时器，告诉它时间间隔有多快，并侦听TimerEvent.TIMER事件，就像你要侦听Event.ENTER_FRAME事件一样。你还需要告诉定时器开始！从那儿开始，定时器将以一个标准的时间间隔广播一个定时器事件，它将调用你在事件指定的任何方法。定时器中的时间间隔指定是以毫秒来计算，通常在你创建定时器时设置。让我们看一个简单的例子（你也可以在Timer1.as中找到）：

www.TopSage.com

```
package {
    import flash.display.Sprite;
    import flash.utils.Timer;
    import flash.events.TimerEvent;

    public class Timer1 extends Sprite
    {
        private var timer:Timer;

        public function Timer1()
        {
            init();
        }

        private function init():void
        {
            timer = new Timer(30);
            timer.addEventListener(TimerEvent.TIMER, onTimer);
            timer.start();
        }

        private function onTimer(timer:TimerEvent):void
        {
            trace("timer!");
        }
    }
}
```

重点部分是粗体部分。你创建了定时器，告诉它每30毫秒触发一次，意味着每秒钟大概要触发33次。你加入了一个事件侦听器并告诉它开始。onTimer方法类似于你在本书中使用的
502 onEnterFrame。

这就是你需要知道的关于定时器的大部分内容。还有两个其他的特性。一个是当你创建一个定时器时，你可以使用第2个参数repeatCount告诉它只运行多少次。假设你想让一个定时器每秒钟运行一次，总共是5秒。你需要这样做：

```
timer = new Timer(1000, 5);
```

如果你没有指定重复次数，或传递的是0，则定时器的运行是不确定的。

另一件比较酷的事情是你可以在任何时候开始和停止它，只需要调用timer.stop和timer.start。在一些情况下，这可能要比移除和重新加入事件侦听器更容易一些。

现在，许多人喜欢定时器超过enterFrame的一个原因是：理论上，定时器给了你一个毫秒级别的动画速度控制——在帧上的很大提高，而帧的不精确是众所周知的。然而，我说“理论上”是因为这里有几件事是要注意的。

首先，定时器是依赖于帧频的。另一点是在你的定时器事件处理器中的代码运行的时间要加入到你的定时器的总的时间间隔中。我稍后会解释第2个问题。现在，让我们来看定时器是如何与帧频相关联的。下面是文档类Timer2.as，它使用的是我们之前使用的Ball类。

www.TopSage.com

```
package {
    import flash.display.Sprite;
    import flash.utils.Timer;
    import flash.events.TimerEvent;

    public class Timer2 extends Sprite
    {
        private var timer:Timer;
        private var ball:Ball;

        public function Timer2()
        {
            init();
        }

        private function init():void
        {
            stage.frameRate = 1;
            ball = new Ball();
            ball.y = stage.stageHeight / 2;
            ball.vx = 5;
            addChild(ball);
            timer = new Timer(20);
            timer.addEventListener(TimerEvent.TIMER, onTimer);
            timer.start();
        }

        private function onTimer(event:TimerEvent):void
        {
            ball.x += ball.vx;
        }
    }
}
```

503

这里我们创建了一个小球放在舞台的左边。它以vx为5的速度穿过屏幕。我们设置定时器为20毫秒，或每秒50次。我们也设置了影片的帧频为1，这只是要看帧频与定时器有什么关系。

当我们运行时，你看到的不是很光滑地移动穿过屏幕，而是小球每秒钟跳一次——帧频。并且每一次跳跃超过5个像素。这说明什么？

好，如果你回顾一下第1和第2章，在那里我们讨论过动画的基础，你可能还记得那个模型需要更新，而屏幕需要基于更新的模型来被刷新。这里，间隔函数确实更新了模型并且每次运行时移动小球5个像素，但屏幕只有在Flash进入到新的一帧的时候才被刷新。只是运行一个函数是不能强制Flash进行重新绘制的。

幸运的是，在Macromedia（现在的Adobe）的人很快发现了这点并给了你一个小工具：updateAfterEvent。它最初是在Flash MX时引入的，现在是传递给你的定时器事件处理器的TimerEvent对象的一个方法。顾名思义，它在一个事件后更新（update）屏幕。当然，因为它是TimerEvent类的一个方法，它只能被接收到它的事件调用。（实际上它也是KeyboardEvent和MouseEvent的方法，因此它也可以被其他处理器调用。）

因此，我们可以按下面的方式来修复onTimer事件：

www.TopSage.com

大家网 TopSage.com

```
private function onTimer(event:TimerEvent):void
{
    ball.x += ball.vx;
    event.updateAfterEvent();
}
```

现在事情看起来好多了。更光滑了一些。但是如果你意识到应该让小球每秒更新50次，你基本上要设置影片的帧频为50 fps。这意味着小球将比我们在第4章中创建的示例移动得还要光滑，而在第4章我们使用的是enterFrame事件，可能要小于50 fps的帧频，但它确实可以移动得更慢。

结果，问题是定时器确实在某些方面与帧频是相关的。通过我的测试，在1 fps的帧频时，定时器的最快运行时间间隔为100毫秒。这也与我听过的定时器每帧只能有10次间隔的说法是一致的。因此，试着将帧频设为5。这将允许有每秒钟50次的更新。对我来说，这看上去仍有点跳跃。它看上去并不是真正地光滑，除非你的帧频达到10帧。因此，你可能看到使用定时器并不能真正地让你摆脱帧频的束缚。

下一个问题是关于定时器的内部是怎样工作的，并且它是怎样影响时间的准确性的。实际上，
504 当timer.start()被调用时，Flash等待着指定的时间数值然后广播与定时器相关的事件。

只有当这些事件处理器已经执行完成，定时器才开始下一次的时间间隔的运行。例如，假设你的定时器设置为每20毫秒运行一次。但假设在它的事件处理器中有许多代码，这些代码全部运行完需要30毫秒。那么定时器的下一轮运行直到事件处理器中的所有代码运行完成后才能开始。在这种情况下，你的处理器实际上将约50毫秒运行一次。因为没有办法准确地测定到在用户的终端机器上代码运行得有多快，所以在多数情况下，基于间隔的动画并没有基于帧的动画准确。

如果你确实需要准确，基于时间的动画是一个选择。

19.3.2　基于时间的动画

基于时间的动画（time-based animation）是当你影片中的物体的速度需要保持一致时使用的动画方法。可能在多种游戏中都是这种情况。如你所看到的，基于帧和定时器的动画都不能让播放以指定的速率播放。一个复杂的影片在一台老的、很慢的计算机上也会运行得比最初设想的要慢。一会你会看到，当使用基于时间的动画时，你可以控制影片的速度，而与影片所运行的帧频无关。

首先你需要做的事情是改变你考虑速度的方式。到现在为止，当我说，例如，vx=5，你使用的单位是每帧多少像素。换句话说，物体每次进入到一个新的帧时将在x轴上移动5个像素。当然，如果是在基于定时器的动画中，它在每个时间间隔里移动的也是5个像素。

对于基于时间的动画，你需要使用准确的真实的时间度量，如秒。因为你处理的是完整的一秒，而不是秒的一部分，所以数值需要更高一些。如果你有什么东西以每帧10个像素在每秒30帧的影片中移动，它大致相当于每秒钟移动300个像素。下一个例子，我使用的是第6章的Bouncing2.as文档类并做了一些修改，你可以看到修改部分为粗体显示部分（你也可以在文件TimeBased.as中找到）：

www.TopSage.com

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.utils.getTimer;

    public class TimeBased extends Sprite
    {
        private var ball:Ball;
        private var vx:Number;
        private var vy:Number;
        private var bounce:Number = -0.7;
        private var time:Number;

        public function TimeBased()
        {
            init();
        }

        private function init():void
        {
            stage.frameRate = 10;
            ball = new Ball();
            ball.x = stage.stageWidth / 2;
            ball.y = stage.stageHeight / 2;
            vx = 300;
            vy = -300;
            addChild(ball);
            time = getTimer();
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var elapsed:Number = getTimer() - time;
            time = getTimer();
            ball.x += vx * elapsed / 1000;
            ball.y += vy * elapsed / 1000;
            var left:Number = 0;
            var right:Number = stage.stageWidth;
            var top:Number = 0;
            var bottom:Number = stage.stageHeight;

            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)
            {
```

505

www.TopSage.com

```
                ball.y = bottom - ball.radius;
                vy *= bounce;
            }
            else if(ball.y - ball.radius < top)
            {
                ball.y = top + ball.radius;
                vy *= bounce;
            }
        }
    }
506 }
```

我扩展了速度，如之前描述的那样，使用的是硬编码的数值，而不是随机确定的。然后创建了一个变量名为time，并设置它等于函数flash.utils.getTimer的结果。getTimer函数相当简单。它返回的是影片已经运行了多少毫秒——这就是它能做的全部。没有办法清除它、重新设置它、改变它或其他别的什么。它只是一个计算器。

现在，它看上去好像没有什么用，但是如果你调用getTimer一次并保存了它的值，然后一会再调用它并将两次的值相减，你就可以准确得到两次之间所消耗的时间（精确到毫秒）。

因此这里的思路是：你在每一帧的开始时调用getTimer并且查看从上一帧到现在消耗了多少毫秒。如果用它除以1 000，你就得到了消耗的1秒的一个分数。因为你的vx和vy现在是以每秒多少像素为单位，你可以使用它们乘以这个分数，从而得出物体移动了多少。还有，不要忘记重新设置time变量值以使你可以测量出下一帧。

测试它，你会看到小球移动的速率与原始例子中移动的速率是一样的。但是更让人惊奇的是你可以将影片发布成任意的帧频，小球仍会以相同的速率移动!试着通过修改stage.frameRate将帧频设为1 000 fps或更低的10 fps，你会看到小球仍是以相同的速率移动。当然，较高的帧频会使影片更光滑，低的帧频会使它有很大跳跃，但速度始终是恒定的。

你可以将这项技术应用在本书任何有速度的示例中。在应用时，你还需要应用类似的技术给加速度或重复的力，如重力，因为它们也是基于时间的。加速度的值在转换为这种动画后会大很多，因为加速度被定义为距离/时间间隔^2。例如，重力近似为32英尺/秒^2。

所以重力在基于帧的30 fps的影片中的大小可能是0.5，而在这里它的大小约为450。它是0.5*30*30。那么你要这样应用它：

```
vy += gravity * elapsed / 1000;
```

继续使用最后一个例子对重力进行测试，使用值450。你会发现它等同于在基于帧的影片中的值为0.5的重力效果。

使用这种技术设置影片的帧频可能会很高，像是100。尽管没有几个人的机器可能真正达到这个帧频，但它可以确保任何人在观看影片时看到的是最光滑的回放。

19.4　相同质量物体之间的碰撞

还记得在第11章中的动量守恒吗？那是一些非常严格的代码。当两个质量相同的物体发生碰撞时你可以使它变得更简单一些。基本上，这些物体只是沿着碰撞的那条线交换它们的速度。虽

www.TopSage.com

然你还有坐标旋转来确定碰撞的那条线和物体的速度，但是它清除了复杂的动量守恒。要看它如何工作的，让我们回到文件MultiBilliard2.as，我们使用它来作为下一个示例SameMass.as的基础。我不打算列出所有的代码，因为它实在是一个很大的文件。但是让我们来看一下它创建所有 507
小球的for循环：

```
for(var i:uint = 0; i < numBalls; i++)
{
    var radius:Number = Math.random() * 50 + 20;
    var ball:Ball = new Ball(radius);
    ball.mass = radius;
    ball.x = Math.random() * stage.stageWidth;
    ball.y = Math.random() * stage.stageHeight;
    ball.vx = Math.random() * 10 - 5;
    ball.vy = Math.random() * 10 - 5;
    addChild(ball);
    balls.push(ball);
}
```

对于新文件，你需要将粗体显示的那一行更改为如下：

```
var radius:Number = 30;
```

这将使所有的小球有相同的大小并移除掉质量，赋予它们相同的质量。

接下来你需要进入到checkCollision函数。找到下面这一段：

```
// 旋转ball0的速度。
var vel0:Point = rotate(ball0.vx,
                        ball0.vy,
                        sin,
                        cos,
                        true);

// 旋转ball1的速度。
var vel1:Point = rotate(ball1.vx,
                        ball1.vy,
                        sin,
                        cos,
                        true);

// 碰撞反应。
var vxTotal:Number = 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;
```

这部分计算碰撞速度，并使用它们的速度质量计算出碰撞的结果。标为“碰撞反应”的部分是动量守恒部分，这一部分你可以去掉。你可以使用交换vel0和vel1的代码替换这部分。这样整个代码段如下： 508

```
// 旋转ball0的速度。
var vel0:Point = rotate(ball0.vx,
                        ball0.vy,
                        sin,
```

www.TopSage.com

```
                        cos,
                        true);

// 旋转ball1的速度。
var vel1:Point = rotate(ball1.vx,
                        ball1.vy,
                        sin,
                        cos,
                        true);

// 交换两者速度。
var temp:Point = vel0;
vel0 = vel1;
vel1 = temp;
```

它可以被进一步优化，但是这样写代码是为了更清晰。这里你已经去掉了一些数学运算，如果你在修改之前和之后都测试过这个文件，会发现它们的结果是一样的。

19.5　声音集成

在本书中很明显被忽略的是声音的使用。虽然声音不是动画的直接部分，但好的声音效果可以使Flash影片更生动和现实。

你可以在Flash中用不同的方式使用声音。具体的在Flash IDE中使用声音的方法要追溯到早期版本——将声音导入到库中并且将它们指定到帧上。我并不打算介绍这种方式的声音使用。我将介绍一些在AS 3中声音使用的基础。

在AS 3中Sound类中已经有了许多改变，并且有几个附加的类来帮助你自定义你的声音。这里没有太多的空间来深入研究它，但是我认为有一个方面对本书目前为止所讲的内容比较有用。那就是当影片中某个事件发生时播放的声音。最明显的现象是一个碰撞。一个小球碰到墙或其他的球，那么你会听到“砰（bang）”或“啵嘤（boing）”或“啪（splat）”或其他的什么声音。对于这个，你需要有在ActionScript中使用声音的能力。

在这个例子中，我再一次回到了Bouncing2.as文件，使用小球在墙上回弹。每次小球碰到墙，我想让它播放一个声音。新的类是SoundEvents.as。

首先，你需要一个音效。在网页上有许多免费的音效。一个最受欢迎的音效网站是FlashKit。除了它们的循环部分——都是好听的音乐，他们还有一个音效库在www.flashkit.com/soundfx/。音效被分类整理为如卡通、界面、机械等等，整个网站上目前有6 000多个音效，因此你可以找到
509 你需要的音效。你可以先在网页上听一下效果，如果找到你喜欢的，下载它的MP3格式。将它保存在你的硬盘上，放在与影片相同的目录下。

有些时候将文件名改得简洁些是有帮助的。例如，我下载了一个“啵嘤”声音，那么我只需要将它重命名为boing.mp3。

在AS 3中声音的基本使用要比在AS 2中简单一点。

首先，你需要创建一个Sound对象。假设你已经声明了一个名为boing的变量：

```
private var boing:Sound;
```

www.TopSage.com

TopSage.com

创建一个声音对象如下：

```
boing = new Sound();
```

当然，像AS 3中的大多数类一样，Sound只是flash.media包的一部分，因此要确保先导入flash.media.Sound。

将声音对象链接到外部的声音文件的最简单的方法是在Sound的构造函数中为它传入一个请求。就像是载入外部图像（查看第4章），你不能只是传递URL链接到外部的声音文件。你需要将它包在URLRequest（它实际上是flash.net.URLRequest，因此需要导入）中。因此它看起来像这样：

```
boing = new Sound(new URLRequest("boing.mp3"));
```

这些就是需要的。现在你的声音已经准备好在你的命令下播放。你所要做的就是把语句

```
mySound.play();
```

放在任何位置音效都将播放。play有一些可选参数，如从多少毫秒开始播放，还有播放多少次，但是默认使用将从开始播放一次，它是通常所用的目的。这里是SoundEvents.as的全部代码，展示了Sound对象的创建过程，你可以看到无论什么时候小球碰到墙壁，它都会播放那个声音。

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.media.Sound;
    import flash.net.URLRequest;

    public class SoundEvents extends Sprite
    {
        private var ball:Ball;
        private var vx:Number;
        private var vy:Number;
        private var bounce:Number = -0.7;
        private var boing:Sound;
        public function SoundEvents()
        {
            init();
        }

        private function init():void
        {
            boing = new Sound(new URLRequest("boing.mp3"));

            ball = new Ball();
            ball.x = stage.stageWidth / 2;
            ball.y = stage.stageHeight / 2;
            vx = Math.random() * 10 - 5;
            vy = -10;
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
```


www.TopSage.com

```
        private function onEnterFrame(event:Event):void
        {
            ball.x += vx;
            ball.y += vy;
            var left:Number = 0;
            var right:Number = stage.stageWidth;
            var top:Number = 0;
            var bottom:Number = stage.stageHeight;

            if(ball.x + ball.radius > right)
            {
                boing.play();
                ball.x = right - ball.radius;
                vx *= bounce;
            }
            else if(ball.x - ball.radius < left)
            {
                boing.play();
                ball.x = left + ball.radius;
                vx *= bounce;
            }
            if(ball.y + ball.radius > bottom)
            {
                boing.play();
                ball.y = bottom - ball.radius;
                vy *= bounce;
            }
            else if(ball.y - ball.radius < top)
            {
                boing.play();
                ball.y = top + ball.radius;
                vy *= bounce;
            }
        }
    }
}
```

511

测试这个影片并且观察……听它所播放的声音有什么不同。当然，在正确的环境中找合适的音乐，而不要太夸张，这本身是一种艺术。

19.6　有用的公式

在本书中，我已经给出了与创建的各种不同的运动和效果有关的公式。我已经尽最大努力提取了最有用和最常用的公式、等式和代码片段，并且把它们列在了每一章的结尾处。我认为把它们全部放在一个地方会更有用，因此在此将它们集中起来，更便于快速参考，我想你会经常用到它的。

第 3 章

计算基本的三角函数：

www.TopSage.com

角的正弦=对边/斜边
角的余弦=邻边/斜边
角的正切=对边/邻边

转换弧度为度和转换度为弧度：

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

旋转到鼠标位置（或任意点）：

```
// 用x,y点替换mouseX,mouseY来旋转。
dx = mouseX - sprite.x;
dy = mouseY - sprite.y;
sprite.rotation = Math.atan2(dy, dx) * 180 / Math.PI;
```

创建波：

```
// 指定值给精灵或影片剪辑的x、y或其他属性，
// 用于绘制坐标等。
public function onEnterFrame(event:Event){
   value = center + Math.sin(angle) * range;
   angle += speed;
}
```

512

创建圆：

```
// 指定位置给精灵或影片剪辑的x和y，
// 用于绘制坐标等。
public function onEnterFrame(event:Event){
   xposition = centerX + Math.cos(angle) * radius;
   yposition = centerY + Math.sin(angle) * radius;
   angle += speed;
}
```

创建椭圆：

```
// 指定位置给精灵或影片剪辑的x和y，
// 用于绘制坐标等。
public function onEnterFrame(event:Event){
   xposition = centerX + Math.cos(angle) * radiusX;
   yposition = centerY + Math.sin(angle) * radiusY;
   angle += speed;
}
```

获得两点之间的距离：

```
// 两点为x1，y1和x2，y2，
// 可以是精灵/影片剪辑的位置、鼠标坐标等。
dx = x2 - x1;
dy = y2 - y1;
dist = Math.sqrt(dx*dx + dy*dy);
```

第4章

转换16进制为10进制：

```
trace(hexValue);
```

www.TopSage.com

转换10进制为16进制：

```
trace(decimalValue.toString(16));
```

组合组成色：

```
color24 = red << 16 | green << 8 | blue;
color32 = alpha << 24 | red << 16 | green << 8 | blue;
```

提取组成色：

```
red = color24 >> 16;
green = color24 >> 8 & 0xFF;
blue = color24 & 0xFF;
alpha = color32 >> 24;
red = color32 >> 16 & 0xFF;
green = color32 >> 8 & 0xFF;
blue = color232 & 0xFF;
```

513

通过一个点绘制曲线：

```
// xt,yt是想要穿过的点，x0,y0和x2,y2是曲线的端点。
x1 = xt * 2 - (x0 + x2) / 2;
y1 = yt * 2 - (y0 + y2) / 2;
moveTo(x0, y0);
curveTo(x1, y1, x2, y2);
```

第 5 章

转换角速度为*x*、*y*速度：

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

转换角度加速度（任何作用在物体上的力）为*x*、*y*加速度：

```
ax = force * Math.cos(angle);
ay = force * Math.sin(angle);
```

将加速度加到速度中：

```
vx += ax;
vy += ay;
```

将速度加到位置上：

```
movieclip._x += vx;
sprite.y += vy;
```

第 6 章

移除超过边界的物体：

```
if(sprite.x - sprite.width / 2 > right ||
   sprite.x + sprite.width / 2 < left ||
   sprite.y - sprite.height / 2 > bottom ||
   sprite.y + sprite.height / 2 < top)
{
    // 移除精灵的代码。
}
```

514

www.TopSage.com

重新生成超过边界的物体：

```
if(sprite.x - sprite.width / 2 > right ||
   sprite.x + sprite.width / 2 < left ||
   sprite.y - sprite.height / 2 > bottom ||
   sprite.y + sprite.height / 2 < top)
{
      // 重置精灵的位置和速度。
}
```

对超过边界的物体进行屏幕折回：

```
if(sprite.x - sprite.width / 2 > right)
{
   sprite.x = left - sprite.width / 2;
}
else if(sprite.x + sprite.width / 2 < left)
{
   sprite.x = right + sprite.width / 2;
}
if(sprite.y - sprite.height / 2 > bottom)
{
   sprite.y = top - sprite.height / 2;
}
else if(sprite.y + sprite.height / 2 < top)
{
   sprite.y = bottom + sprite.height / 2;
}
```

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;
```

应用摩擦力（容易方法）：

```
vx *= friction;
vy *= friction;
```

515

第 8 章

简单的缓动，较长的形式：

```
var dx:Number = targetX - sprite.x;
var dy:Number = targetY - sprite.y;
vx = dx * easing;
vy = dy * easing;
```

www.TopSage.com

大家网 TopSage.com

```
sprite.x += vx;
sprite.y += vy;
```

简单的缓动，简化的形式：

```
vx = (targetX - sprite.x) * easing;
vy = (targetY - sprite.y) * easing;
sprite.x += vx;
sprite.y += vy;
```

简单的缓动，极简的形式：

```
sprite.x += (targetX - sprite.x) * easing;
sprite.y += (targetY - sprite.y) * easing;
```

简单的弹性，较长的形式：

```
var ax:Number = (targetX - sprite.x) * spring;
var ay:Number = (targetY - sprite.y) * spring;
vx += ax;
vy += ay;
vx *= friction;
vy *= friction;
sprite.x += vx;
sprite.y += vy;
```

简单的弹性，简化的形式：

```
vx += (targetX - sprite.x) * spring;
vy += (targetY - sprite.y) * spring;
vx *= friction;
vy *= friction;
sprite.x += vx;
sprite.y += vy;
```

简单的弹性，极简的形式：

```
vx += (targetX - sprite.x) * spring;
vy += (targetY - sprite.y) * spring;
sprite.x += (vx *= friction);
sprite.y += (vy *= friction);
```

516

偏移弹性：

```
var dx:Number = sprite.x - fixedX;
var dy:Number = sprite.y - fixedY;
var angle:Number = Math.atan2(dy, dx);
var targetX:Number = fixedX + Math.cos(angle) * springLength;
var targetY:Number = fixedX + Math.sin(angle) * springLength;
// 像上面一样弹到targetX, targetY。
```

第9章

基于距离的碰撞检测：

```
// 以spritA和spriteB开始，如果使用的是平板精灵或一个没有radius属性的物
   体，你可以使用width或height除以2。
var dx:Number = spriteB.x - spriteA.x;
```

www.TopSage.com

```
var dy:Number = spriteB.y - spriteA.y;
var dist:Number = Math.sqrt(dx * dx + dy * dy);
if(dist < spriteA.radius + spriteB.radius)
{
    // 处理碰撞。
}
```

多物体碰撞检测：

```
var numObjects:uint = 10; // 例如。
for(var i:uint = 0; i < numObjects - 1; i++)
{
    // 使用变量i获取引用。例如：
    var objectA = objects[i];

    for(var j:uint = i+1; j<numObjects;j++)
    {
        // 使用变量j获取引用，例如：
        var objectB = objects[j];

        // 执行碰撞检测，在objectA和objectB之间。
    }
}
```

第 10 章

坐标旋转：

```
x1 = Math.cos(angle) * x - Math.sin(angle) * y;
y1 = Math.cos(angle) * y + Math.sin(angle) * x;
```

517

反转坐标旋转：

```
x1 = Math.cos(angle) * x + Math.sin(angle) * ;y
y1 = Math.cos(angle) * y - Math.sin(angle) * x;
```

第 11 章

动量守恒，直接的数学方式：

$$v0Final = \frac{(m0 - m1) * v0 + 2 * m1 * v1}{m0 + m1}$$

$$v1Final = \frac{(m1 - m0) * v1 + 2 * m0 * v0}{m0 + m1}$$

ActionScript中的动量守恒，简化方式：

```
var vxTotal:Number = vx0 - vx1;
vx0 = ((ball0.mass - ball1.mass) * vx0 +
       2 * ball1.mass * vx1) /
       (ball0.mass + ball1.mass);
vx1 = vxTotal + vx0;
```

www.TopSage.com

第 12 章

基本的引力：

```
force = G * m1 * m2 / distance2
```

ActionScript中的引力实现：

```
function gravitate(partA:Ball, partB:Ball):void
{
      var dx:Number = partB.x - partA.x;
      var dy:Number = partB.y - partA.y;
      var distSQ:Number = dx * dx + dy * dy;
      var dist:Number = Math.sqrt(distSQ);
      var force:Number = partA.mass * partB.mass / distSQ;
      var ax:Number = force * dx / dist;
      var ay:Number = force * dy / dist;
      partA.vx += ax / partA.mass;
      partA.vy += ay / partA.mass;
      partB.vx -= ax / partB.mass;
      partB.vy -= ay / partB.mass;
}
```

518

第 14 章

余弦定律：

$$a^2 = b^2 + c^2 - 2 * b * c * \cos A$$
$$b^2 = a^2 + c^2 - 2 * a * c * \cos B$$
$$c^2 = a^2 + b^2 - 2 * a * b * \cos C$$

ActionScript余弦定律：

```
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));
```

第 15 章

基本的透视：

```
scale = fl / (fl + zpos);
sprite.scaleX = sprite.scaleY = scale;
sprite.alpha = scale;      //可选的。
sprite.x = vanishingPointX + xpos * scale;
sprite.y = vanishingPointY + ypos * scale;
```

z轴排序：

```
// 假设一个3D物体数组有zpos属性。
objectArray.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
for(var i:uint = 0; i < numObjects; i++)
{
    setChildIndex(objectArray[i], i);
}
```

www.TopSage.com

坐标旋转：

```
x1 = cos(angleZ) * xpos - sin(angleZ) * ypos;
y1 = cos(angleZ) * ypos + sin(angleZ) * xpos;

x1 = cos(angleY) * xpos - sin(angleY) * zpos;
z1 = cos(angleY) * zpos + sin(angleY) * xpos;

y1 = cos(angleX) * ypos - sin(angleX) * zpos;
z1 = cos(angleX) * zpos + sin(angleX) * ypos;
```

3D距离：

```
dist = Math.sqrt(dx * dx + dy * dy + dz * dz);
```

519

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引