TURING 图灵程序设计丛书

Apress®

Flash ActionScript 3.0 动画高级教程

[美] Keith Peters 著
苏金国 荆涛 等译

跟随Flash大师进入ActionScript开发辉煌殿堂
丰富的经典示例和专家技巧
涵盖Flash 10最新特性

人民邮电出版社
POSTS & TELECOM PRESS

大家网
TopSage.com

图书在版编目（CIP）数据

Flash ActionScript 3.0 动画高级教程 /（美）彼得斯（Peters, K.）著；苏金国等译．—北京：人民邮电出版社，2010.1（2010.4 重印）
（图灵程序设计丛书）
书名原文：AdvancED ActionScript 3.0 Animation
ISBN 978-7-115-21625-0

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

中国版本图书馆CIP数据核字（2009）第193958号

内 容 提 要

本书是介绍 Flash 10 ActionScript 动画高级技术的经典之作，是作者在这一领域中多年实践经验的结晶。书中不仅涵盖了 3D、最新绘图 API 以及 Pixel Bender 等 Flash 10 ActionScript 特性，深入介绍了碰撞检测、转向、寻路等 Flash 游戏开发技术，还通过实例具体讲解了等角投影和数值积分的基本理论和应用。

本书内容紧扣实际应用，适合各层次 Flash 开发人员阅读。

图灵程序设计丛书

Flash ActionScript 3.0 动画高级教程

◆ 著 [美] Keith Peters
译 苏金国 荆 涛 等
责任编辑 傅志红
执行编辑 罗 婧

◆ 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
北京艺辉印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：24
字数：573 千字 2010 年 1 月第 1 版
印数：3 001 –4 000 册 2010 年 4 月北京第 2 次印刷

著作权合同登记号 图字：01-2009-5723号
ISBN 978-7-115-21625-0

定价：65.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223
反盗版热线：(010)67171154

www.TopSage.com

版权声明

Original English language edition, entitled *AdvancED ActionScript 3.0 Animation* by Keith Peters, published by Apress, 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705 USA.

Copyright © 2009 by Keith Peters. Simplified Chinese-language edition copyright © 2010 by 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

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

献给亲爱的Miranda和Kristine，再次感谢她们的耐心与支持。

www.TopSage.com

大家网
TopSage.com

译 者 序

这本书是鼎鼎大名的Keith Peters继《Flash ActionScript 3.0动画教程》之后的又一力作，相比之下，它将带领我们从“基础”进阶到“高级”领域。

那么，我们从中能得到些什么？也许并没有高深的理论，但这里的技术绝对贴近实战，如果充分领会并巧妙应用这些技术，定能让你的作品标新立异，也会让别人对你刮目相看。书中各个主题相对独立，又彼此关联，你可以有目的地对感兴趣的章节重点研究，也可以跟随作者的思路通读全文。作者不仅解决了开发游戏中最关心的碰撞检测、转向和寻路问题，还通过生动的实例让我们了解了等角投影和数值积分的基本理论和应用途径，特别谈到了将摄像头和麦克风纳入输入手段的简捷方法，另外补充了Flash 10中的3D特性和绘图API，最后帮助我们了解了Pixel Bender和各种补间引擎的作用和用法。

阅读这本书，肯定会让你跃跃欲试，迫不及待地想要亲手体验这些高级技术带来的绝妙效果，这也正是本书要达到的目的。技术如果只是纸上谈兵、束之高阁，那它就是死的，只有真正用于实践并不断扩展演进，才能算是名副其实的实用技术。特别值得一提的是，英文书名中AdvancED一词中的“ED”很值得玩味，也许是为了强调“高级”，也许是要告诉你“做过才知道”，各种理解见仁见智，不过可以肯定的是，只有真正读过、做过，才会有最大的收获。

在此深深地感谢我们的家人和朋友。在翻译过程中，他们给予了我们莫大的关心、支持和帮助。

全书主要由苏金国、荆涛翻译，并得到高强、王小振、刘鑫、范松峰、王恒、牛亚峰、刘亮、刘跃邦等人的帮助，最后由苏金国、荆涛统稿。

由于时间仓促，且译者的水平有限，译文中难免会出现一些错误，请读者批评指正。

www.TopSage.com

致　谢

本书里只有很少一部分（如果有的话）是我自己构想出来的。要感谢成百上千的程序员、开发人员、科学家、数学家和物理学家们，是他们的努力研究和探索，精心编程和转换，才使得千百万人能够受益于他们的工作成果。

排版约定

为了让本书尽可能简明易读，本书将采用以下排版约定。

- 重要的词或概念在第一次出现时一般会用黑体突出显示。
- 代码用Letter Gothic字体表示。
- 新的代码或有修改的代码通常用粗体Letter Gothic字体表示。
- 伪代码和变量输入采用斜体Letter Gothic字体。

www.TopSage.com

目　录

第 1 章　高级碰撞检测 ····· 1

1.1　不规则形状对象的碰撞测试 ····· 1

1.1.1　位图用于碰撞检测 ····· 4

1.1.2　半透明形状的碰撞测试 ····· 7

1.1.3　使用BitmapData.hitTest测试非位图对象 ····· 10

1.2　大量对象的碰撞测试 ····· 12

1.2.1　实现基于网格的碰撞检测 ····· 14

1.2.2　编写网格代码 ····· 17

1.2.3　测试和调整网格 ····· 24

1.2.4　建成一个可重用的类 ····· 27

1.2.5　碰撞检测：并不只是碰撞 ····· 37

1.3　小结 ····· 42

第 2 章　转向行为 ····· 43

2.1　行为 ····· 44

2.2　Vector2D类 ····· 45

2.3　Vehicle类 ····· 52

2.4　SteeredVehicle类 ····· 59

2.4.1　搜寻行为 ····· 60

2.4.2　逃避行为 ····· 62

2.4.3　到达行为 ····· 66

2.4.4　追捕行为 ····· 68

2.4.5　躲避行为 ····· 70

2.4.6　漫游行为 ····· 72

2.4.7　对象规避 ····· 74

2.4.8　路径追随 ····· 78

2.5　群落 ····· 82

2.6　小结 ····· 85

第 3 章　等角投影 ····· 87

3.1　等角投影与二等角投影 ····· 89

3.2　创建等角图形 ····· 90

3.3　等角变换 ····· 91

3.3.1　世界坐标变换为屏幕坐标 ····· 92

3.3.2　屏幕坐标转换为世界坐标 ····· 96

3.3.3　IsoUtils类 ····· 96

3.4　等角对象 ····· 98

3.5　深度排序 ····· 108

3.6　等角世界类 ····· 113

3.7　3D中的移动 ····· 115

3.8　碰撞检测 ····· 120

3.9　使用外部图形 ····· 123

3.10　等角区块图 ····· 127

3.11　小结 ····· 133

第 4 章　寻路 ····· 134

4.1　寻路基础 ····· 134

4.2　A* ····· 134

4.2.1　A*基础 ····· 136

4.2.2　A*算法 ····· 136

4.2.3　计算代价 ····· 137

4.2.4　算法图示 ····· 138

4.2.5　编写代码 ····· 142

4.2.6　常用A*启发函数 ····· 153

4.3　实现AStar类 ····· 157

4.4　在游戏中使用AStar ····· 164

4.5　高级地形 ····· 167

4.6　小结 ····· 169

第 5 章　其他输入方式：摄像头和麦克风 ····· 170

5.1　摄像头和麦克风 ····· 171

5.2　声音作为输入 ····· 171

www.TopSage.com

5.2.1 一个由声音控制的游戏 …… 175
5.2.2 活动事件 …… 178
5.3 视频作为输入 …… 180
5.3.1 视频大小和质量 …… 182
5.3.2 视频与位图 …… 183
5.3.3 分析像素 …… 184
5.4 小结 …… 203
第 6 章 高级物理：数值积分 …… 204
6.1 数值积分以及为什么欧拉积分“不好” …… 204
6.2 龙格–库塔积分 …… 206
6.2.1 基于时间的运动 …… 208
6.2.2 编写龙格–库塔二阶积分（RK2）代码 …… 212
6.2.3 编写龙格–库塔四阶积分（RK4）代码 …… 214
6.2.4 薄弱环节 …… 217
6.2.5 龙格–库塔小结 …… 218
6.3 Verlet积分 …… 218
6.3.1 Verlet点 …… 219
6.3.2 Verlet线段 …… 224
6.3.3 Verlet结构 …… 228
6.3.4 关节 …… 233
6.3.5 进一步深入 …… 234
6.4 小结 …… 235
第 7 章 Flash 10 中的 3D …… 236
7.1 Flash 10 3D基础 …… 236
7.2 3D定位 …… 242
7.2.1 深度排序 …… 243
7.2.2 3D容器 …… 245
7.3 3D旋转 …… 247
7.4 视场与焦距 …… 256
7.5 屏幕和3D坐标 …… 259
7.6 指向 …… 263
7.7 小结 …… 264
第 8 章 Flash 10 绘图 API …… 265
8.1 路径 …… 265
8.1.1 一个简单的绘图程序 …… 268
8.1.2 绘制曲线 …… 270
8.1.3 Wide绘制命令和NO_OP …… 271
8.1.4 环绕 …… 274
8.2 三角形 …… 278
8.2.1 位图填充与三角形 …… 281
8.2.2 三角形与3D …… 289
8.3 图形数据 …… 299
8.4 小结 …… 304
第 9 章 Pixel Bender …… 305
9.1 什么是Pixel Bender …… 305
9.2 编写一个Pixel Shader …… 307
9.3 数据类型 …… 310
9.4 获得当前像素坐标 …… 311
9.5 参数 …… 315
9.6 输入图像采样 …… 319
9.7 Flash的旋转Shader …… 323
9.8 在Flash中使用Pixel Bender Shader …… 325
9.8.1 加载shader与嵌入shader …… 326
9.8.2 shader用于填充 …… 327
9.8.3 在Flash中访问shader元数据 …… 329
9.8.4 在Flash中设置shader参数 …… 330
9.8.5 变换shader填充 …… 331
9.8.6 实现shader填充动画 …… 332
9.8.7 指定shader输入图像 …… 333
9.9 使用Shader作为滤镜 …… 335
9.10 使用Shader作为混合模式 …… 336
9.11 小结 …… 338
第 10 章 补间引擎 …… 339
10.1 Flash Tween类 …… 340
10.1.1 缓动函数 …… 341
10.1.2 结合补间 …… 343
10.2 Flex Tween类 …… 344
10.2.1 Flex Tween类的缓动函数 …… 349
10.2.2 多重补间 …… 350
10.2.3 补间序列 …… 352
10.3 补间引擎 …… 353
10.4 Tweener …… 354
10.4.1 Tweener中的缓动函数 …… 355
10.4.2 Tweener中的多重补间 …… 355

www.TopSage.com

10.4.3 Tweener中的序列 ····· 355
10.5 TweenLite/TweenGroup ····· 358
10.5.1 TweenLite中的缓动函数 ····· 359
10.5.2 TweenLite的多重补间 ····· 360
10.5.3 TweenLite/TweenGroup中的序列 ····· 361
10.6 KitchenSync ····· 365
10.6.1 KitchenSync中的缓动函数 ····· 366
10.6.2 利用KitchenSync对多个对象/属性补间 ····· 367
10.6.3 KitchenSync中的补间序列 ····· 369
10.7 gTween ····· 370
10.7.1 gTween中的缓动函数 ····· 371
10.7.2 利用gTween完成多个对象补间 ····· 371
10.7.3 gTween中的补间序列 ····· 372
10.8 小结 ····· 373

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

大家网
TopSage.com

第1章

高级碰撞检测

碰撞检测可谓一门数学、艺术或科学，或者就是一般意义上的猜测推断，用来确定某个对象是否与另一个对象发生碰撞。这听起来很简单，不过当你面对的是只存在于计算机内存中并由一组不同属性表示的对象时，复杂性便会接踵而至。

碰撞检测的基本方法在《Flash ActionScript 3.0 动画教程》[①]一书中做了详细说明。本章将讨论该书中未涉及的一种碰撞检测方法，并介绍一种处理大量对象互相碰撞的策略。

需要指出，主题是碰撞检测，因此不会深入研究在检测到碰撞之后要做的事情。如果你在开发一个游戏，可能希望发生碰撞的对象爆炸、改变颜色，或者只是简单地消失。处理碰撞结果还有一种相当复杂的方法，这个方法在《Flash ActionScript 3.0 动画教程》的11.3节中做了介绍。不过，最终还是要由你（根据所构建应用或游戏的规范）来确定检测到碰撞时该如何做出响应。

1.1 不规则形状对象的碰撞测试

《Flash ActionScript 3.0 动画教程》中介绍了几个碰撞检测的基本方法，包括内置的hitTestObject和hitTestPoint方法，以及基于距离的碰撞检测。以上各种方法分别适用于不同形状的参与碰撞检测的对象。hitTestObject方法非常适合检测两个矩形对象之间的碰撞，但是对于其他形状往往会生成假警报。hitTestPoint方法适用于以下两种情况：查看鼠标是否位于一个特定对象之上，或者一个很小的点状对象是否与任何其他形状的对象发生碰撞，但是这个方法对于检测两个较大的对象则束手无策。基于距离的碰撞检测非常适于处理圆形对象，不过对于其他形状的对象常常会产生碰撞漏检。

Flash中实现碰撞检测的理想方法（相当于这个领域的“圣杯”）是对两个不规则形状的对象进行相互测试，从而准确地知道它们是否接触。从Flash 8开始，已经有一种方法可以通过BitmapData类做到这一点（不过《Flash ActionScript 3.0 动画教程》中没有讨论这种方法）。实际上，这种方法有一个很贴切的名字：hitTest。

首先，需要对一些术语做一点说明。ActionScript包含一个BitmapData类，其中保存要显示的具体位图图像，另外还有一个Bitmap类，这是包含一个BitmapData的显示对象，并允许将这个

① 此书已由人民邮电出版社于2008年4月出版。——编者注

www.TopSage.com

BitmapData添加到显示列表中。在后面的介绍中，如果我特别指出其中某一个类或者某个类的实例，会使用首字母大写的类名（BitmapData或Bitmap）。不过，通常情况下，我可能会用**位图**（bitmap）一词不太正式地指一个位图图像。不要把它与Bitmap类相混淆。

BitmapData.hitTest会比较两个BitmapData对象，指出它们的像素是否发生重叠。重申一次，这听起来很容易，但是如果再多加考虑就会发现其中存在一些复杂性。位图是像素的矩形网格，所以如果只是采用最简单的形式，这个方法不会比作用于显示对象的hitTestObject方法更复杂（当然同样也不会有太大用处）。只有当使用了绘制有一个形状的透明位图时这个方法才真正有用。

创建一个BitmapData对象时，要在构造函数中明确指定它是否支持透明性：

```
new BitmapData(width, height, transparent, color);
```

第3个参数是一个布尔值（true/false），这个参数将设置透明性选项。如果将其设置为false，位图就是完全不透明的。最初这呈现为一个用指定背景颜色填充的矩形。可以使用不同的BitmapData方法来改变位图中的任何像素，不过它们总是完全不透明的，并且会覆盖该BitmapData之下的所有内容。各像素的颜色值是一个形式为0xRRGGBB的24位数字。这是一个6位的十六进制数，其中第一对数指定了红色通道的值，取值范围是00（0）～FF（255），第二对设置了绿色通道，第三对设置蓝色通道。例如，0xFFFFFF是白色，0xFF0000是红色，0xFF9900是橙色。要设置和获取单个像素的值，可以使用setPixel和getPixel方法，它们都使用24位颜色值。

不过，在BitmapData类中将透明性选项指定为true时，每个像素还会支持一个*α*通道，相应地会使用一个格式为0xAARGGBB的32位数。在这里，前两位十六进制数表示一个像素的透明度，00为完全透明，FF则是完全不透明。在一个透明的BitmapData中，需要使用setPixel32与getPixel32来设置和读取单个像素的颜色。这些方法都取32位数作为参数。需要指出，如果向其中某个方法传入一个24位数，*α*通道将计算为0，也就是完全透明。

为了了解二者的具体差别，下面分别创建一个例子。可以在一个Flex Builder 3或4版本的ActionScript工程中使用下面的类作为主类，或者在Flash CS3或CS4中作为文档类。这个BitmapCompare类可以从本书网站www.friendsofed.com下载[①]。

```
package
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.geom.Rectangle;

    public class BitmapCompare extends Sprite
    {
        public function BitmapCompare()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
```

① 本书代码也可以从图灵网站www.turingbook.com本书网页免费注册下载。——编者注

www.TopSage.com

```
            // draw a bunch of random lines
            graphics.lineStyle(0);
            for(var i:int = 0; i < 100; i++)
            {
                graphics.lineTo(Math.random() * 300, Math.random() * 400);
            }

            // create an opaque bitmap
            var bmpd1:BitmapData = new BitmapData(300, 200, false, 0xffffff);
            bmpd1.fillRect(new Rectangle(100, 50, 100, 100), 0xff0000);
            var bmp1:Bitmap = new Bitmap(bmpd1);
            addChild(bmp1);

            // create a transparent bitmap
            var bmpd2:BitmapData = new BitmapData(300, 200, true, 0x00ffffff);
            bmpd2.fillRect(new Rectangle(100, 50, 100, 100), 0xffff0000);
            var bmp2:Bitmap = new Bitmap(bmpd2);
            bmp2.y = 200;
            addChild(bmp2);
        }
    }
}
```

这段代码首先在场景上画一组随机的线条，以便区分场景和位图。然后创建两个位图，并在各位图的中央画上红色的方块。上面的位图是不透明的，会完全覆盖所有线条。下面的位图是透明的，所以只有红色方块覆盖了场景中的线条。其结果如图1-1所示。

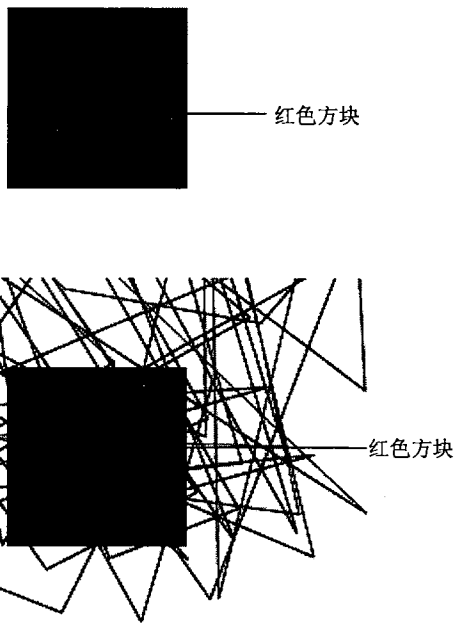


图1-1 上面是不透明位图，下面是透明位图

www.TopSage.com

大家网
TopSage.com

另外，对于透明位图还可以使用部分透明度。对以上代码示例中的第二个fillRect语句做如下修改：

```
bmpd2.fillRect(new Rectangle(100, 50, 100, 100), 0x80FF0000);
```

注意，这里使用了一个32位AARRGGBB颜色值来填充，α值减半为0x80，即十进制数128。这样红色方块就会变得半透明，如图1-2所示。

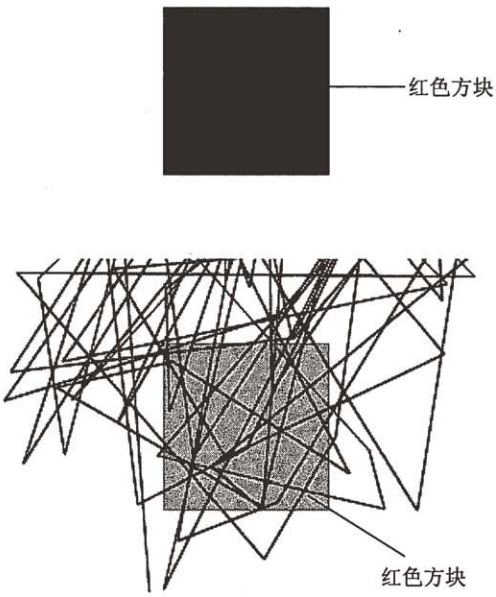


图1-2　一个半透明的方块

1.1.1　位图用于碰撞检测

现在来看如何用位图实现碰撞检测。首先需要一个适当的不规则形状来完成测试，五角星就很合适，可以建立一个五角星类以便以后重用。以下就是Star类，这个类也可以从本书网站下载得到。

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

    public class Star extends Sprite
    {
        public function Star(radius:Number, color:uint = 0xFFFF00):void
        {
            graphics.lineStyle(0);
            graphics.moveTo(radius, 0);
            graphics.beginFill(color);
            // draw 10 lines
            for(var i:int = 1; i < 11; i++)
```

www.TopSage.com

```
            {
                var radius2:Number = radius;
                if(i % 2 > 0)
                {
                 // alternate radius to make spikes every other line
                    radius2 = radius / 2;
                }
                var angle:Number = Math.PI * 2 / 10 * i;
                graphics.lineTo(Math.cos(angle) * radius2, Math.sin(angle) * radius2);
            }
        }
    }
}
```

这里只是画出角度不断递增而且半径交替变化的一系列线条，从而巧妙地构成了一个五角星。下面这个类可以完成碰撞测试。类似于本书中的大多数代码，这个类可以作为一个文档类在Flash CS3或CS4中使用，也可以作为主应用类在Flex Builder 3或4中使用，同样也可以从本书网站下载得到。

```
package
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.filters.GlowFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;

    public class BitmapCollision1 extends Sprite
    {
        private var bmpd1:BitmapData;
        private var bmp1:Bitmap;
        private var bmpd2:BitmapData;
        private var bmp2:Bitmap;

        public function BitmapCollision1()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            // make a star
            var star:Star = new Star(50);

            // make a fixed bitmap, draw the star into it
            bmpd1 = new BitmapData(100, 100, true, 0);
            bmpd1.draw(star, new Matrix(1, 0, 0, 1, 50, 50));
            bmp1 = new Bitmap(bmpd1);
            bmp1.x = 200;
            bmp1.y = 200;
            addChild(bmp1);
```

www.TopSage.com

```
            // make a moveable bitmap, draw the star into it, too
            bmpd2 = new BitmapData(100, 100, true, 0);
            bmpd2.draw(star, new Matrix(1, 0, 0, 1, 50, 50));
            bmp2 = new Bitmap(bmpd2);
            addChild(bmp2);

            stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMoving);
        }

        private function onMouseMoving(event:MouseEvent):void
        {
            // move bmp2 to the mouse position (centered).
            bmp2.x = mouseX - 50;
            bmp2.y = mouseY - 50;

            // the hit test itself.
            if(bmpd1.hitTest(new Point(bmp1.x, bmp1.y), 255, bmpd2,
                             new Point(bmp2.x, bmp2.y), 255))
            {
                bmp1.filters = [new GlowFilter()];
                bmp2.filters = [new GlowFilter()];
            }
            else
            {
                bmp1.filters = [];
                bmp2.filters = [];
            }
        }
    }
}
```

这里使用Star类创建了一个五角星，并分别绘制在两个位图中。我们使用了一个矩阵从而在绘制时将五角星在各个轴上分别偏移50个像素，因为五角星的对准点位于其中心，而位图的对准点位于左上角。通过偏移就可以看到整个五角星。

其中一个位图（bmp1）位于场景中的固定位置，另一个位图（bmp2）设置为跟随鼠标移动。这里的关键代码如下所示：

```
if(bmpd1.hitTest(new Point(bmp1.x, bmp1.y), 255, bmpd2,
                 new Point(bmp2.x, bmp2.y), 255))
```

正是这段代码确定了两个位图是否接触。BitmapData.hitTest方法的签名如下：

```
hitTest(firstPoint:Point,
        firstAlphaThreshold:uint,
        secondObject:Object,
        secondPoint:Point,
        secondAlphaThreshold:uint);
```

注意到，参数可分为两组：分别以first和second开头[①]。要分别为其提供一个点值（第一个

① 分别对应第一个对象和第二个对象。——译者注

www.TopSage.com

点和第二个点），这对应于BitmapData的左上角。之所以这样做，其原因在于各个位图有可能嵌套在另一个实体中，或者可能深深地嵌套在多重实体中。在这种情况下，它们可能在完全不同的坐标系统中。如果有必要，可以使用DisplayObject.localToGlobal方法指定一个任意点将两个坐标系统对齐。不过，在这个例子中，两个位图正好放在场景上，所以可以直接使用其局部位置来构造各自的点。

下一组first/last参数对应α阈值。前面已经了解到，在一个透明的BitmapData中，各像素的透明度取值范围可以从0（完全透明）到255（完全不透明）。α阈值参数指定了像素要有怎样的不透明度才能算一次碰撞。在这个例子中，我们将两个α阈值都设置为255，这表示对于任一位图中的像素，要认为它通过了碰撞测试，那么这个像素必须完全不透明。稍后还会介绍另一个例子，展示使用较低阈值的情况。

最后，这里还有一个secondObject参数。需要注意，这个参数的类型指定为对象（Object）。这里可以使用Point、Rectangle或者另一个BitmapData作为测试对象。如果使用Point或Rectangle，那么就不再需要使用最后两个参数。如果希望测试鼠标是否接触到一个位图，就可以用一个Point进行测试，这很有用。下面是一个简单的例子：

```
if(myBitmapData.hitTest(new Point(myBitmapData.x, myBitmapData.y), 255,
                        new Point(mouseX, mouseY)))
{
    // mouse is touching bitmap
}
```

我想不出特别有用的例子来展示如何用一个矩形测试位图，不过要知道确实可以这样做。如果需要，你完全可以利用矩形完成位图测试。

不过在我们的例子中使用了另一个BitmapData对象，所以还要传入第二个Point和α阈值。

最后，如果出现碰撞，就使用一个默认的发光滤镜为各五角星指定一个红色发光光环。如果没有碰撞，则删除所有滤镜。可以看到的结果如图1-3和图1-4所示。

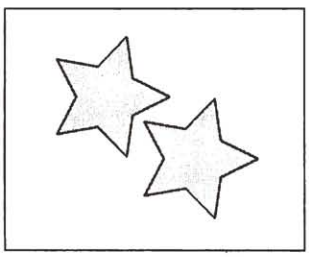


图1-3 五角星未接触

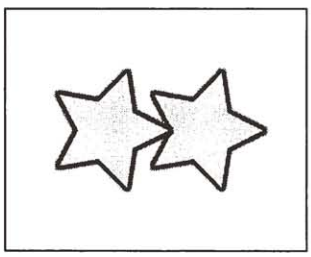


图1-4 现在五角星发生碰撞

可以做一些尝试，你会看到这确实是像素与像素间的碰撞检测。

1.1.2 半透明形状的碰撞测试

在前面的例子中，我们在每个位图中绘制了一个完全不透明的五角星。由于要对每位图中完全不透明的像素进行测试，因此将每个位图中的α阈值设置为255。（实际上可以把α阈值设置为

www.TopSage.com

任何大于0的值，其效果是相同的。）

下面来看如何对一个并非完全不透明的形状完成碰撞测试。这里对BitmapCollsion1类稍做修改，将其命名为BitmapCollision2（可以从本书网站下载得到）：

```
package
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.GradientType;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.filters.GlowFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;

    public class BitmapCollision2 extends Sprite
    {
        private var bmpd1:BitmapData;
        private var bmp1:Bitmap;
        private var bmpd2:BitmapData;
        private var bmp2:Bitmap;

    public function BitmapCollision2()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        // make a star
        var star:Star = new Star(50);

        // make a gradient circle
        var matrix:Matrix = new Matrix();
        matrix.createGradientBox(100, 100, 0, -50, -50);
        var circle:Sprite = new Sprite();
        circle.graphics.beginGradientFill(GradientType.RADIAL, [0, 0], [1, 0], [0, 255], matrix);
        circle.graphics.drawCircle(0, 0, 50);
        circle.graphics.endFill();

        // make a fixed bitmap, draw the star into it
        bmpd1 = new BitmapData(100, 100, true, 0);
        bmpd1.draw(star, new Matrix(1, 0, 0, 1, 50, 50));
        bmp1 = new Bitmap(bmpd1);
        bmp1.x = 200;
        bmp1.y = 200;
        addChild(bmp1);

        // make a moveable bitmap, draw the star into it, too
        bmpd2 = new BitmapData(100, 100, true, 0);
        bmpd2.draw(circle, new Matrix(1, 0, 0, 1, 50, 50));
        bmp2 = new Bitmap(bmpd2);
        addChild(bmp2);
```

www.TopSage.com

```
            stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMoving);
        }

        private function onMouseMoving(event:MouseEvent):void
        {
            // move bmp2 to the mouse position (centered).
            bmp2.x = mouseX - 50;
            bmp2.y = mouseY - 50;

            // the hit test itself.
            if(bmpd1.hitTest(new Point(bmp1.x, bmp1.y), 255, bmpd2,
                             new Point(bmp2.x, bmp2.y), 255))
            {
                bmp1.filters = [new GlowFilter()];
                bmp2.filters = [new GlowFilter()];
            }
                else
                {
                    bmp1.filters = [];
                    bmp2.filters = [];
                }
            }
        }
    }
```

这里建立了一个新的名为circle的Sprite对象，并在其中绘制了一个放射状渐变填充的圆形。将这个图形（而不是五角星）绘制到bmpd2中。进行测试时，在圆形的圆心接触到五角星之前，不会看到出现任何碰撞，因为只有圆心处的像素才是完全不透明的。其结果如图1-5和图1-6所示。

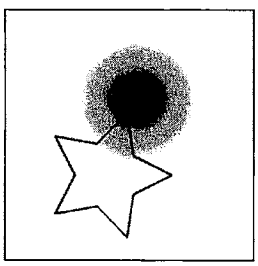


图1-5 五角星接触到圆形，但是相应像素没有达到期望的α阈值

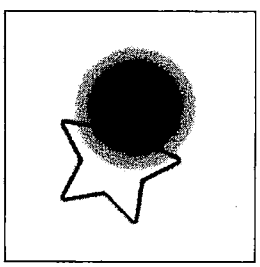


图1-6 只有圆心处的α值为255，所以此时出现碰撞

修改完成碰撞测试的代码，将第二个α阈值改为一个较小的值，如下所示：

```
if(bmpd1.hitTest(new Point(bmp1.x, bmp1.y), 255, bmpd2,
                 new Point(bmp2.x, bmp2.y), 128))
```

现在只需朝着五角星部分移动圆形[1]，一旦达到α至少为128的像素时就会出现碰撞。试着将

① 而不必移动到使圆心与五角星接触。——译者注

www.TopSage.com

第二个α阈值设置为不同的值来看相应的效果。需要指出，如果将α阈值设置为0，甚至圆形尚未接触到五角星，也有可能认为发生了碰撞，因为尽管处于位图拐角的像素完全透明，也能成功地通过碰撞测试。要记住，位图本身仍是一个矩形（尽管你根本看不到它）。还要注意，修改第一个α阈值（设置为任何非0的值）不会带来任何改变，因为五角星没有半透明的像素——它们要么完全透明，要么完全不透明。

1.1.3　使用 BitmapData.hitTest 测试非位图对象

在到目前为止的所有例子中，我们一直都直接使用Bitmap对象作为要移动和测试的显示对象。但是在很多情况下（几乎是大多数情况），实际上会移动不同类型的显示对象，如MovieClip、Sprite或Shape对象。由于不能对这些类型的对象完成上述碰撞测试，因此需要对设置稍做修改。具体策略是维护一些隐式BitmapData对象，但是不放在显示列表中。每当需要检查两个实际显示对象之间是否存在碰撞时，将显示对象分别绘制到一个位图上，并对位图进行碰撞测试。

应该认识到，使用位图完成碰撞检测并不是唯一的办法，也不一定是最好的做法。还有几十种可能的方法，当然使用位图完成碰撞测试确实能得到想要的效果。可以原样使用这种方法，也可以在它的基础上加以改进。

下面是BitmapCollision3类（可以从本书网站下载）：

```
package
{
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.filters.GlowFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;

    public class BitmapCollision3 extends Sprite
    {
        private var bmpd1:BitmapData;
        private var bmpd2:BitmapData;
        private var star1:Star;
        private var star2:Star;

        public function BitmapCollision3()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            // make two stars, add to stage
            star1 = new Star(50);
            addChild(star1);

            star2 = new Star(50);
```

www.TopSage.com

```
            star2.x = 200;
            star2.y = 200;
            addChild(star2);

            // make two bitmaps, not on stage
            bmpd1 = new BitmapData(stage.stageWidth, stage.stageHeight, true, 0);
            bmpd2 = bmpd1.clone();

            stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMoving);
        }

        private function onMouseMoving(event:MouseEvent):void
        {
            // move star1 to the mouse position
            star1.x = mouseX;
            star1.y = mouseY;

            // clear the bitmaps
            bmpd1.fillRect(bmpd1.rect, 0);
            bmpd2.fillRect(bmpd2.rect, 0);

            // draw one star to each bitmap
            bmpd1.draw(star1, new Matrix(1, 0, 0, 1, star1.x, star1.y));
            bmpd2.draw(star2, new Matrix(1, 0, 0, 1, star2.x, star2.y));

            // the hit test itself.
            if(bmpd1.hitTest(new Point(), 255, bmpd2, new Point(), 255))
            {
                star1.filters = [new GlowFilter()];
                star2.filters = [new GlowFilter()];
            }
            else
            {
                star1.filters = [];
                star2.filters = [];
            }
        }
    }
}
```

在这个构造函数中，我们建立了两个BitmapData对象和两个五角星。没有必要把BitmapData对象放在Bitmap中，因为它们不会加入到显示列表。另一方面，五角星确实要增加到显示列表。第一个五角星star1会随鼠标移动。每当鼠标移动时，两个位图都使用fillRect并传入颜色值0来清空。要记住，如果α通道未指定，则取值为0，因此其结果等同于令所有像素完全透明。接下来，将每个五角星绘制到其相应的位图上：

```
bmpd1.draw(star1, new Matrix(1, 0, 0, 1, star1.x, star1.y));
bmpd2.draw(star2, new Matrix(1, 0, 0, 1, star2.x, star2.y));
```

矩阵使用五角星的x和y位置作为平移值，这样一来，各个五角星就会绘制与原场景中相同的位置上。现在可以完成碰撞测试：

www.TopSage.com

```
if(bmpd1.hitTest(new Point(), 255, bmpd2, new Point(), 255))
```

由于BitmapData不在显示列表上，甚至不在Bitmap包装器中，而且由于两个五角星都位于相同的坐标空间，并分别以其相对位置绘制到各BitmapData上，因此不需要做任何坐标空间修正。只需为每个Point参数分别传入一个新的默认Point（其x和y均为0）。这里仍保留α阈值为255，因为两个五角星都是完全不透明的。

尽管这个例子看上去与其他例子没有两样，但实际上已是截然不同，因为位图不可见，而五角星是可见的。不过，其工作方式完全相同。

这些只是使用BitmapData.hitTest对非圆形、矩形或点状对象完成碰撞检测的几个例子。我相信，一旦了解其工作原理，你肯定能想出更多更酷的改进用法。

接下来，讨论如何完成大规模的碰撞检测。

1.2　大量对象的碰撞测试

Flash Player 10中的ActionScript比之前的版本运行速度更快，这就允许我们可以一次做更多工作，而且可以同时移动更多对象。但是限制仍然存在。如果开始在屏幕上移动大量对象，速度迟早会变得越来越慢。大量对象之间的碰撞检测更会引发问题，因为每一个对象都需要与其他各个对象进行比较。这一点并不仅限于碰撞检测，任何包含大量对象而且对象相互之间需要交互（如通过重力或群落）的粒子系统或游戏（参见第2章）都会遇到同样的问题。

如果只有6个相互作用的对象，每个对象则需要与其他的每一个对象结对，并完成碰撞测试、重力吸引或所需的任何动作。乍一看，这意味着需要6×6共36次单独比较，但其实并非如此。正如《Flash ActionScript 3.0 动画教程》所述，实际上比较次数还不及其一半，准确地讲，应该是15次比较。给定对象A, B, C, D, E, F，需要完成以下结对比较：

AB, AC, AD, AE, AF

BC, BD, BE, BF

CD, CE, CF

DE, DF

EF

注意，B不必再检查A，因为A已经与B做过碰撞检查。达到E时，已经检查过除F以外的所有对象。而在此之后，F已经得到了所有其他对象的检查。下面是所需实际比较次数的计算公式，这里N是对象个数：

$(N^2 - N)/2$

对于6个对象，则计算为（36－6）/2，即15。

对于10个对象，则是（100－10）/2，即45次检查。

20个对象意味着190次检查，30个对象将是435次检查!

可以看到，检查次数会飞速增长，因此需要采取一些措施来加以限制。用ActionScript 3.0在

www.TopSage.com

TopSage.com

屏幕上移动100个对象并不太困难，但是如果开始进行碰撞检测或者对象间的另外一些比较，就需要完成4 950次单独的检查！如果使用基于距离的碰撞检测，这就意味着需要计算4 950次两个对象之间的距离。如本章前面所述，如果使用位图碰撞，这就需要完成4 950次下述过程：清除两个位图，绘制两个对象，并调用hitTest方法。每一帧都是如此！这势必导致SWF文件速度的下降。

幸好我们有一种技巧可以限制需要完成的检查次数。可以这样来考虑：如果两个相对较小的对象分别位于屏幕的两侧，它们就不太可能发生碰撞。但是为了证明这一点，需要先计算二者之间的距离，对不对？这样一来我们又回到了最初的问题上。不过，也许还有其他办法。

假设将屏幕分割成正方形单元格网格，其中每个单元格至少与最大的对象一样大，然后根据对象中心所在位置将各个对象分配到该网格的某个单元格中。如果一切正常，给定单元格中的对象只可能与其周围的8个单元格中的对象发生碰撞。例如，请看图1-7。

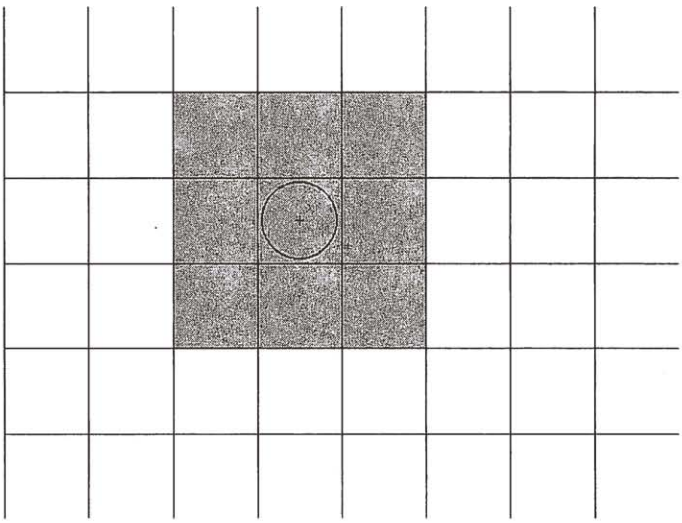


图1-7 这个球只可能与加阴影的单元格中的对象发生碰撞

图中所示的球根据其圆心位置分配到一个单元格。只有加阴影的单元格中的对象才有可能与它发生碰撞。这个球不可能与任何白色单元格中的对象碰撞。即使球位于其单元格的边缘，而另一个球位于某个白色单元格的边缘，它们也不可能相互接触（参见图1-8）。

重申一次，这种情况的前提条件是单元格的大小至少与所要比较的最大对象同样大。如果任何一个球大于单元格，那么在上述情况下它们则有可能相互碰撞。

以上就是基本思路。应该知道，可能还有很多其他方法。我不能确保这是最好的方法，但是可以明确我们的目标是将各个对象与它可能接触到的所有其他对象两两测试，并确保绝对不会将两个对象相互测试两次。正是这一点有些技巧。

这里将概要介绍我提出的方法，这种方法看上去可能相当抽象。目前只需大致了解要对网格的哪些区域进行碰撞检测。至于具体怎样做到将在后面讨论。

www.TopSage.com

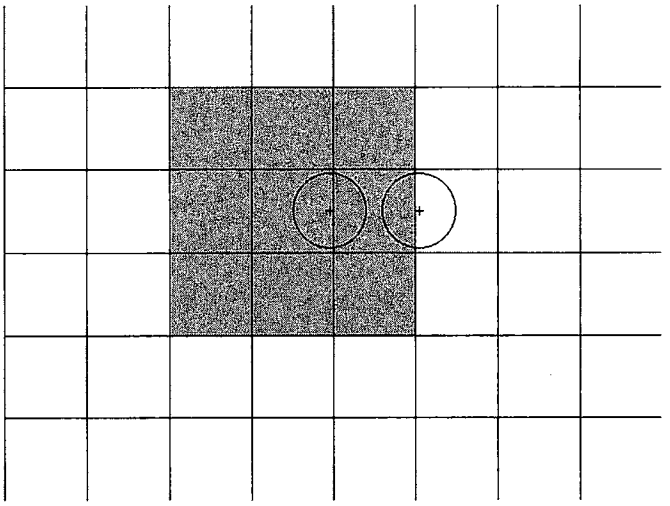


图1-8　两个球不可能碰撞

1.2.1　实现基于网格的碰撞检测

先从左上角开始，这里稍稍缩小了网格大小以使问题更为简单，参见图1-9。

你希望对第一个深灰色单元格中的所有对象与其周围单元格中的所有对象进行测试。当然，在它的左边或上面不会有其他单元格，所以只需检查3个浅灰色单元格。再次指出，这个深灰色单元格中的对象不可能与任何白色单元格中的对象发生碰撞。

完成测试后，移至下一个单元格，参见图1-10。

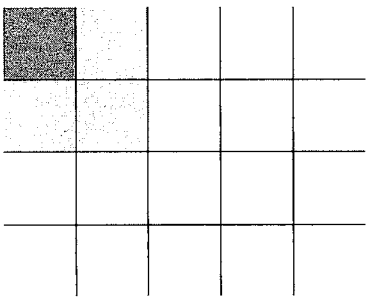


图1-9　对第一个单元格中的所有对象与周围单元格中的所有对象进行测试

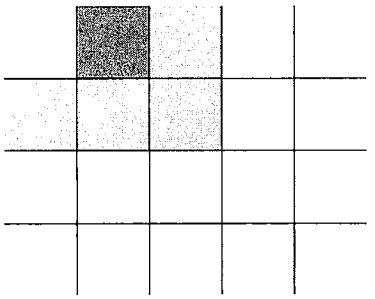


图1-10　继续处理下一个单元格

这个单元格周围有更多单元格，不过要记住，我们已经将第一个单元格中的所有对象与周围3个单元格中的所有对象完成了比较，其中也包括现在要测试的这个单元格。所以没有必要再对第一个单元格做任何测试。

继续以同样的方式处理第一行。只需测试当前单元格，其右边的单元格以及下面的3个单元格。参见图1-11、图1-12和图1-13。

www.TopSage.com

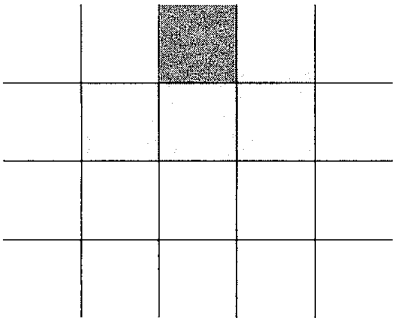


图1-11 继续处理第一行

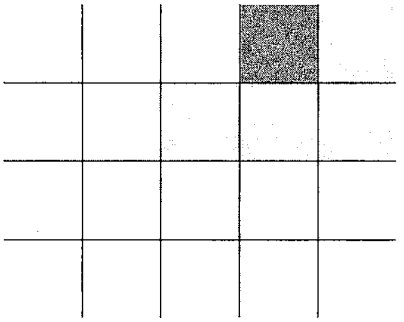


图1-12 第一行的下一列

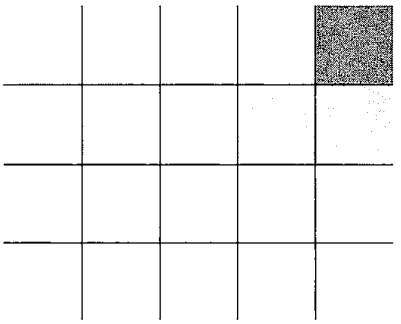


图1-13 第一行的最后一列

当然，达到这一行的最后一个单元格时，它右边再没有单元格，所以只需测试下面的两个单元格。

再从第二行开始，参见图1-14和图1-15。

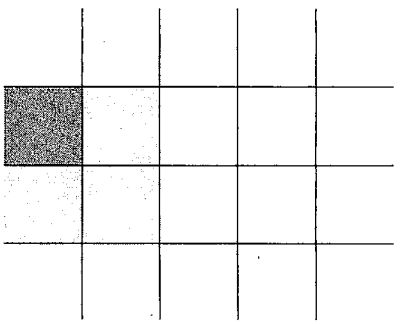


图1-14 开始处理第二行

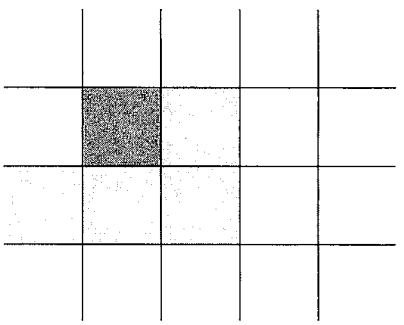


图1-15 第二行的下一列

开始时周围都有单元格，但是上面一行已经与第二行中的所有对象做过检查，所以可以忽略上面一行。看来这与第一行并无不同，能重用代码总是件好事。

最终到达最后一行，参见图1-16和图1-17。

www.TopSage.com

大家网 TopSage.com

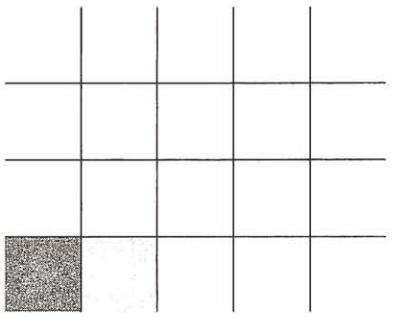


图1-16　最后一行

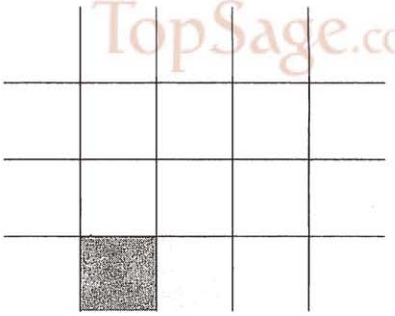


图1-17　最后一行第二列

在这里不用再考虑下面一行，并且上面一行已经做过测试。所以只需将各单元格与右边的单元格进行测试。到达最后一个单元格时，甚至无需做任何测试，因为所有其他单元格都已经与它进行过测试。参见图1-18。

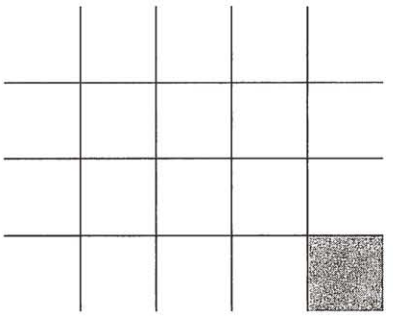


图1-18　这里无需做任何测试

以上就是我们所要做的。那么怎样才能做到呢？我们最多要处理5个单元格：所检查的主单元格、右边的单元格，以及下面的3个单元格。其中各个“单元格”实际上是一个对象数组。将这些数组称为cell0、cell1、cell2、cell3和cell4。为简单起见，假设每个单元格只包含Ball对象。

先来看cell0，也就是第一个球对象数组。这个单元格中的任何球都有可能与其中任何其他球碰撞，所以需要对所有这些球相互之间进行测试。通过一个双重循环来检测，如《Flash ActionScript 3.0 动画教程》所述。下面是相应代码的一个大致流程：

```
for(var i:int = 0; i < cell0.length - 1; i++)
{
    var ballA:Ball = cell0[i] as Ball;
    for(var j:int = i + 1; j < cell0.length; j++)
    {
        var ballB:Ball = cell0[j] as Ball;
        // hit test or other reaction between ballA and ballB
    }
}
```

www.TopSage.com

这就将每个球与其他各个球进行测试，同时保证每个球不会与自身进行测试，而且任意一对球不会重复测试两次。这就完成了cell0中球之间的所有碰撞检测。现在再来看cell0和cell1之间的测试。情况会稍有不同：我们要取cell0中的每个球，逐一地与cell1中的每个球完成测试。同样地，这也实现为一个双重循环：

```
for(var i:int = 0; i < cell0.length; i++)
{
    var ballA:Ball = cell0[i] as Ball;
    for(var j:int = 0; j < cell1.length; j++)
    {
        var ballB:Ball = cell1[j] as Ball;
        // hit test or other reaction between ballA and ballB
    }
}
```

注意，这段代码对两个数组中的所有元素进行了完全迭代处理，这与第一个比较有所不同，第一个比较中使用了一些有意思的技巧来避免重复检查。但这里不需要担心这一点，因为我们正在处理包含完全不同元素的两个数组。

可以通过重复这种检查来完成cell0与cell2、cell3和cell4的比较。此时，cell0的比较结束，再来比较下一个单元格，它将成为下一个cell0。当然，对于位于左边沿、右边沿或下边沿的所有单元格来说，其周围的单元格要少于4个，所以必须考虑到这个问题。

如果你像我一样，可能觉得前面的介绍晦涩难懂，不清楚引入这种复杂性的做法为什么比所有对象相互比较的做法效率可能更高。那我们可以通过数学运算来理解。应该记得，如果将100个对象相互比较，就要完成4 950次检查。在下面的例子中，我们将弄清楚具体发生了多少次比较，而具体的次数和屏幕大小、对象大小、对象个数、网格大小以及对象的随机分布位置密切相关。在我们的测试中，100个对象之间实际发生的碰撞检测平均次数在100次到200次之间。这就节省了大约4 800次检查！由于每次碰撞测试都包括多行代码，其中还包含一个开销很大的平方根计算，因此节省了测试次数也就能显著节省CPU时间。

当然，创建和更新网格、将所有对象分配到网格以及循环处理所有数组等工作会带来很大开销，而且通常每一帧上都需要完成这些工作。如果拥有大量对象，减少计算次数所节省的开销会远远超过完成上述工作带来的开销。但是在一个只有少量对象的系统中，直接将各个对象与所有其他对象进行检查则会更为高效。本章后面将讨论如何度量这两种方法的好处，从而确定这两种方法的适用情况。

1.2.2 编写网格代码

我们的第一个目标是力求清晰。为此，首先介绍各个函数，以便你了解在整个过程中发生了什么。之后会进行整理，建立一个可重用的类。

在具体实现碰撞检测之前，先建立要完成碰撞检测的对象：Ball类，这个类可以从本书网站下载（www.friendsofed.com）：

```
package
{
```

www.TopSage.com

```
import flash.display.Sprite;

public class Ball extends Sprite
{
    private var _color:uint;
    private var _radius:Number;
    private var _vx:Number = 0;
    private var _vy:Number = 0;


    public function Ball(radius:Number, color:uint = 0xffffff)
    {
        _radius = radius;
        _color = color;
        draw();
    }

    private function draw():void
    {
        // draw a circle with a dot in the center
        graphics.clear();
        graphics.lineStyle(0);
        graphics.beginFill(_color, .5);
        graphics.drawCircle(0, 0, _radius);
        graphics.endFill();
        graphics.drawCircle(0, 0, 1);
    }

    public function update():void
    {
        // add velocity to position
        x += _vx;
        y += _vy;
    }

    public function set color(value:uint):void
    {
        _color = value;
        draw();
    }

    public function get color():uint
    {
        return _color;
    }

    public function set radius(value:Number):void
    {
        _radius = value;
        draw();
    }

    public function get radius():Number
    {
```

www.TopSage.com

TopSage.com

```
            return _radius;
        }

        public function set vx(value:Number):void
        {
            _vx = value;
        }

        public function get vx():Number
        {
            return _vx;
        }

        public function set vy(value:Number):void
        {
            _vy = value;
        }

        public function get vy():Number
        {
            return _vy;
        }

    }
}
```

这里没有什么特别之处。它接收一个半径和一个颜色，然后画出一个圆。每个圆要记录其x和y速度，通知它更新时将把这个速度增加到它的当前位置（来得到新位置）。仅此而已。

下面开始讨论应用类，在Flex Builder中可以把它用作主类，或者在Flash CS3或CS4中可以用作文档类。要确保Ball类与这个类同在一个位置。目前我把所有类都放在默认包中，这同样是为了保证清晰。完全可以将这些类组织到适合你的一个包结构中。以下是GridCollision类，可以从本书下载页面下载。

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

    public class GridCollision extends Sprite
    {
        private const GRID_SIZE:Number = 50;
        private const RADIUS:Number = 25;

        private var _balls:Array;
        private var _grid:Array;
        private var _numBalls:int = 100;
        private var _numChecks:int = 0;

        public function GridCollision()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
```

www.TopSage.com

```
            makeBalls();
            makeGrid();
            drawGrid();
            assignBallsToGrid();
            checkGrid();
            trace(_numChecks);
        }

    // the rest of the methods described in this section will go here
    }
}
```

程序最初为网格大小和球半径定义了相应的常量。要记住，网格大小应当至少为最大对象的大小，对于圆来说，就应当是其半径的两倍。所以这里可以满足这个需求。

接下来为球建立一个数组，另外再建立一个数组作为网格。我们将对100个球完成测试，还要建立一个变量来保存所完成碰撞测试的累积次数。

构造函数调用了多个方法来创建球，建立网格，绘制网格，将球分配到网格，以及检查网格中是否出现碰撞。最后，它会输出碰撞测试的次数。

先来介绍这个类的这些方法（而不是构造函数）。首先是makeBalls：

```
private function makeBalls():void
{
    _balls = new Array();
    for(var i:int = 0; i < _numBalls; i++)
    {
        // create a Ball and add it to the display list
        // and the _balls array
        var ball:Ball = new Ball(RADIUS);
        ball.x = Math.random() * stage.stageWidth;
        ball.y = Math.random() * stage.stageHeight;
        addChild(ball);
        _balls.push(ball);
    }
}
```

同样地，这段函数也没有任何复杂的内容。在此会建立一个数组，运行一个循环来创建一组Ball实例，将这些Ball实例随机地分布在场景中，然后添加到显示列表，并放入_balls数组中。

接下来是makeGrid()方法：

```
private function makeGrid():void
{
    _grid = new Array();
    // stage width / grid size = number of columns
    for(var i:int = 0; i < stage.stageWidth / GRID_SIZE; i++)
    {
        _grid[i] = new Array();
        // stage height / grid size = number of rows
        for(var j:int = 0; j < stage.stageHeight / GRID_SIZE; j++)
        {
            _grid[i][j] = new Array();
        }
```

www.TopSage.com

```
    }
}
```

这里创建了一个二维数组，其中各元素分别代表屏幕上被划分成的网格中的一格。二维数组中的各个元素又包含另一个数组。（可以称之为一个三维数组，不过这个概念在这里不太适用。）最后的这些数组用来保存分配到网格各个区域的对象。

下一个方法drawGrid()纯粹是为了辅助之用。这个方法对于碰撞检测来说并无帮助，它只是帮助你显示出各个网格元素在哪里。在最终的游戏或应用中很可能并不绘制网格。

```
private function drawGrid():void
{
    // draw lines to indicate rows and columns
    graphics.lineStyle(0, .5);
    for(var i:int = 0; i <= stage.stageWidth; i += GRID_SIZE)
    {
        graphics.moveTo(i, 0);
        graphics.lineTo(i, stage.stageHeight);
    }
    for(i = 0; i <= stage.stageHeight; i += GRID_SIZE)
    {
        graphics.moveTo(0, i);
        graphics.lineTo(stage.stageWidth, i);
    }
}
```

再来看最重要的方法之一assignBallsToGrid：

```
private function assignBallsToGrid():void
{
    for(var i:int = 0; i < _numBalls; i++)
    {
        // dividing position by grid size
        // tells us which row and column each ball is in
        var ball:Ball = _balls[i] as Ball;
        var xpos:int = Math.floor(ball.x / GRID_SIZE);
        var ypos:int = Math.floor(ball.y / GRID_SIZE);
        _grid[xpos][ypos].push(ball);
    }
}
```

需要对这段代码多做一些解释。第一部分很明显：循环处理数组，得到其中各个Ball对象的一个引用。然后用该球的x位置除以网格大小，将结果去掉小数部分取整为最接近的整数值。这会告诉我们这个球应该位于网格的哪一列。对y位置做同样的处理，从而可以得到应该把球放在网格的哪一行。参见图1-19。

在这个例子中，假设网格元素为100×100个。这里看到的球的位置在（380, 280）。将380除以100得到3.8，去掉小数部分后取整得到3，由此可知这个球要放在网格的列3上。应该记得，数组元素是从0开始索引的，所以索引为3实际上是指第4列。对y坐标做同样的处理，可以得出这个球位于行2（第3行）。查看图1-19可以看到球的中心确实位于网格的第4列、第3行（从左上角开始计起），从而能轻松地验证以上数学计算的正确性。

www.TopSage.com

大家网 TopSage.com

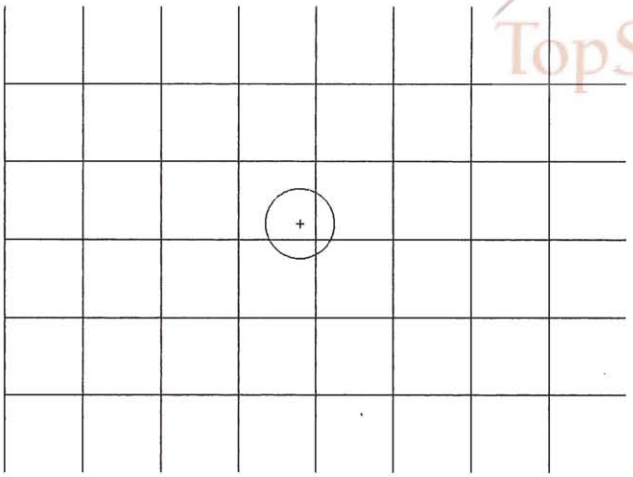


图1-19　明确一个球属于哪个网格元素

再来看代码，将这些计算的结果赋给xpos和ypos，并用它们来索引这个二维网格。由于二维数组的各元素本身也是一个数组，因此将对象压入该元素中的数组。

循环结束时，各对象将位于网格中的一个特定元素中。有些网格元素可能包含一个对象，有些则包含多个对象，还有一些可能为空。现在已经准备就绪，可以进行碰撞测试了。

所有实质性工作都由checkGrid()方法完成。实际上，它还要依赖于另外一些方法，这些方法稍后就会介绍。先来看checkGrid()方法：

```
private function checkGrid():void
{
    // loop through each row and column of grid
    for(var i:int = 0; i < _grid.length; i++)
    {
        for(var j:int = 0; j < _grid[i].length; j++)
        {
            // examine all the objects in the first cell
            // against each other
            checkOneCell(i, j);
            checkTwoCells(i, j, i + 1, j);     // cell to the right
            checkTwoCells(i, j, i - 1, j + 1); // below to the left
            checkTwoCells(i, j, i, j + 1);     // directly below
            checkTwoCells(i, j, i + 1, j + 1); // below to the right
        }
    }
}
```

我们使用一个双重循环来循环处理网格中的各行和各列。索引i和j表示当前正在检查的单元格。首先要通过checkOneCell()方法对该单元格中的所有对象进行相互比较：

```
private function checkOneCell(x:int, y:int):void
{
    // check all the balls in a single cell against each other
    var cell:Array = _grid[x][y] as Array;
```

www.TopSage.com

```
        for(var i:int = 0; i < cell.length - 1; i++)
        {
            var ballA:Ball = cell[i] as Ball;
            for(var j:int = i + 1; j < cell.length; j++)
            {
                var ballB:Ball = cell[j] as Ball;
                checkCollision(ballA, ballB);
            }
        }
    }
```

这段代码对数组完成了一个双重循环，如《Flash ActionScript 3.0 动画教程》所述，其结果是该单元格中的每个对象与其他每一个对象比较一次且仅一次。

然后调用4次checkTwoCells()方法：

```
checkTwoCells(i, j, i + 1, j);     // cell to the right
checkTwoCells(i, j, i - 1, j + 1); // cell below to the left
checkTwoCells(i, j, i, j + 1);     // cell directly below
checkTwoCells(i, j, i + 1, j + 1); // cell below to the right
```

索引i和j仍指示当前检查的主单元格。因此i + 1，j指示其右边的单元格；i -1，j + 1是其左下方的单元格；i，j + 1是正下方的单元格；i + 1，j + 1则是右下方的单元格，如图1-9到图1-18所示。

以下是checkTwoCells()的代码：

```
private function checkTwoCells(x1:int, y1:int, x2:int, y2:int):void
{
    // make sure the second cell really exists
    if(x2 < 0) return;
    if(x2 >= _grid.length) return;
    if(y2 >= _grid[x2].length) return;
    var cell0:Array = _grid[x1][y1] as Array;
    var cell1:Array = _grid[x2][y2] as Array;

    // check all the balls in one cell against all in the other
    for(var i:int = 0; i < cell0.length; i++)
    {
        var ballA:Ball = cell0[i] as Ball;
        for(var j:int = 0; j < cell1.length; j++)
        {
            var ballB:Ball = cell1[j] as Ball;
            checkCollision(ballA, ballB);
        }
    }
}
```

在这里，i和j变成x1和y1。这些变量用来得到第一个单元格的一个引用。后面两个参数变成x2和y2。需要确保它们在合法的范围内。如果x2小于0，或者大于等于_grid.length，_grid[x2]将为null（当前格子位于场景的两侧）。如果主单元格位于第一列或最后一列就会出现这种情况（参见图1-9和图1-13）。类似地，如果y2大于该列的单元格数目，则越界（当前格子处于场景底部），_grid[x2][y2]将是null（参见图1-16到图1-18）。如果出现以上任何条件，将直接从方法

www.TopSage.com

退出，因为不会再有下一个需要检查的合法单元格。

不过，如果通过了这个条件检查，就可以成功地得到要检查的两个单元格的引用。正如前面所述，我们使用一个双重循环来循环处理第一个单元格中的所有元素，并与第二个单元格中的所有元素进行比较。

最后一个问题就是碰撞检测本身（放在最后介绍并不表示它不重要）。我们已经经过一系列处理走到目前这一步，现在已经有两个要相互测试的对象。为此调用checkCollision()，并传入各对象的一个引用。这个方法如下所示：

```
private function checkCollision(ballA:Ball, ballB:Ball):void
{
    // if distance is less than sum of radii, collision
    _numChecks++;
    var dx:Number = ballB.x - ballA.x;
    var dy:Number = ballB.y - ballA.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    if(dist < ballA.radius + ballB.radius)
    {
        ballA.color = 0xff0000;
        ballB.color = 0xff0000;
    }
}
```

函数执行到这里就说明确实进行了两个对象间的实际碰撞检测，所以更新_numChecks属性来统计检测的次数。然后完成一个标准的基于距离的碰撞检测，为此要使用两球之间在*x*轴和*y*轴上的距离的平方。如果两球距离[①]小于其半径之和，则将两个球都变为红色。实际上，还可以进一步优化，去除平方根计算，对半径之和求平方，再对二者进行比较[②]。只要能避开Math函数，就能节省一些CPU时间。

工作真是不少，不过终于避免了最后这个方法可能多达几千次的执行。如果一切正常，最后会得到如图1-20所示的结果。

验证所有接触的球都变为红色，而未接触的球都为白色。如果并非如此，说明某个地方可能出错。找出问题并加以修正，然后才能继续进行下一步。碰撞检测例程中最重要的当然是准确地检测碰撞。

1.2.3　测试和调整网格

每次运行这个应用时，应当记录完成了多少次碰撞检查。在我的计算机上，屏幕分辨率为1440×900，在浏览器最大化的情况下运行这个SWF，对于以上代码，会出现大约80到130次碰撞测试：这里设置有100个半径为25的球，以及一个大小为50的网格。场景越小，产生碰撞测试的次数就越多。可以多运行几次来了解测试次数的大致范围。由于连续运行中有变化的只是球的随机分布，这样就能对平均情况有所了解。

① 两球间的距离是对*x*轴和*y*轴上距离的平方和求平方根。不过作者在下面指出不必具体求出这个距离，即通过去除平方根计算可以优化性能。——译者注

② 即*x*轴和*y*轴上距离的平方和与两球半径和的平方进行比较。——译者注

www.TopSage.com

大家网 TopSage.com

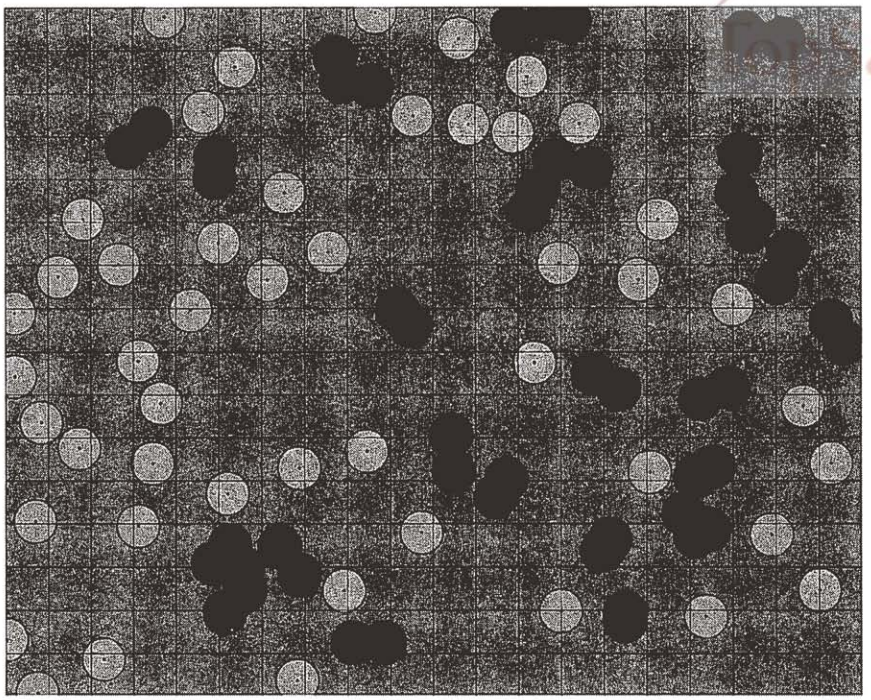


图1-20 100个对象，成功地完成碰撞测试

现在来增加网格大小，如将其设置为100。你会注意到，碰撞测试次数将显著增加，因为每个单元格中会有更多的球。这听起来很糟糕，但是另一方面，网格大小的增加也就意味着单元格个数以及需要循环处理的数组个数减少了，这是有好处的，稍后你就会了解到这些。

现在再把网格大小减少到40，甚至30。碰撞测试会更少，但是再仔细查看结果。你可能会发现碰撞偶然漏检的情况——两个对象确实已经接触，但是没有标志为碰撞（也就是说，它们仍为白色）。这很不好，不过由此可以再次提醒你重视网格大小规则的重要性，即网格大小至少要与最大对象同样大。可以更进一步，根据仿真中的最大对象动态地设置网格大小。

再将网格大小设置回50，修改makeBalls中创建各个球的代码行，如下：

```
var ball:Ball = new Ball(Math.random() * RADIUS);
```

这个修改对于检查次数应该没有任何影响，一切正常。这个代码只是要表明：对象有不同大小时这个系统仍能正常工作——当然要牢记有关网格大小的规则。

现在再做一些重要的调整，看看执行效率到底如何，特别是与“各个对象与其他每一个对象两两测试”的方法作个比较。

为此，需要另一个测试方法。将这个方法称为basicCheck()，方法如下：

```
private function basicCheck():void
{
    for(var i:int = 0; i < _balls.length - 1; i++)
    {
```

www.TopSage.com

```
            var ballA:Ball = _balls[i] as Ball;
            for(var j:int = i + 1; j < _balls.length; j++)
            {
                var ballB:Ball = _balls[j] as Ball;
                checkCollision(ballA, ballB);
            }
        }
    }
```

这个代码对于100个对象会生成4 950次碰撞测试。我们知道，从检查次数来看网格方法要远远胜出，但是与网格方法相比，这个方法的开销却少得多。那么这两个方法的较量结果到底怎样？我们确实能从网格方法得到好处吗？

显然肯定能得到一些好处，否则我也不会专门写这一章，不过你可以先不听我讲，而是自己试试看。我们要做的就是将以上基于网格的代码运行10次，再将这个方法运行10次，看看哪一个方法运行的时间更短。这些都在构造函数中完成，构造函数修改如下：

```
public function GridCollision()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    makeBalls();
    drawGrid();
    var startTime:int;
    var elapsed:int;
    var i:int;

    // get initial time
    startTime = getTimer();
    // do this 10 times
    for(i = 0; i < 10; i++)
    {
        makeGrid();
        assignBallsToGrid();
        checkGrid();
    }
    // subtract current time from start time
    elapsed = getTimer() - startTime;
    trace("Grid-based:", elapsed);

    // do again for basic check
    startTime = getTimer();
    for(i = 0; i < 10; i++)
    {
        basicCheck();
    }
    elapsed = getTimer() - startTime;
    trace("Basic check", elapsed);
}
```

这里为每次测试的开始时间和逝去时间创建了相应变量。将开始时间设置为getTimer()的结果，并将以下3个方法运行10次：

www.TopSage.com

```
makeGrid();
assignBallsToGrid();
checkGrid();
```

从当前时间减去开始时间，看看共花费了多长时间。再运行10次basicCheck()，做同样的处理。

你会得到什么结果？对我的设置而言，可以看到基于网格的方法与基本检查方法相比运行速度大约快2.5倍。没有我想象中那么突出，要知道我们已经减少了4 800次碰撞测试，由此可以看出网格方法确实存在很大开销。不过不管怎样，速度提升2.5倍也不错，不是吗？

另外，对象越多，会看到开销节省得越多。如果将_numBalls属性提高到1 000，基于网格的方法会在不到1秒的时间内处理完成，而基本检查耗时则会超过13秒。（当然，每秒1帧并不是一个能够让人接受的帧速率，但比起每一帧都需要13秒来说则要好得多！）另一方面，如果将球数减少到50，实际上基本检查反而更快。所以这里有一个分界点，从这一点开始，网格方法带来的开销将超出因减少碰撞测试所节省的开销，此时最好转而采用简单测试。具体的分界点在哪里取决于具体的应用以及测试需求。

将_numBalls重新设置为100，再运行几次，从而对基于网格的方法花费的时间有所认识。在我的机器上，10次运行的平均时间大约为55毫秒。现在将网格大小从50改为75。平均时间则降低到大约37毫秒。我发现网格大小在85到100之间时，平均时间约为32毫秒，这就比基本检查快了4倍！需要指出，所有这些数字都只是一些例子，仅作为基本指导。你自己可能会得到截然不同的结果。这里的重点在于，通过调整网格大小可以找到一个设置，能为你的应用提供最优结果。这里有很多存在变数的方面：场景大小、对象个数、对象大小、网格大小以及所使用的碰撞测试算法（还不包括各个计算机的性能）。在此我无法给出一个魔法公式来提供可用的最佳参数。

1.2.4 建成一个可重用的类

通过分析GridCollision类，希望能帮助你很好地掌握基于网格的碰撞检测的基本概念，并明确这种方法的意义。不过这里还存在几个问题。首先，所有这些都包装在一个主文档类中，所以如果要在另一个应用中使用就需要复制和粘贴。另外它与Ball类还存在紧密耦合。如果希望针对另一类对象使用这个类，就必须修改该类的所有引用。除此以外，这里将碰撞测试算法编码实现为一个基于距离的碰撞检测，而你可能希望使用一个简单的hitTestObject或者甚至希望使用基于位图的碰撞测试。所以下面来创建一个新类，这个新类将解决上述所有问题，并尽可能优化。这个类名为CollisionGrid，可以从本书网站www.friendsofed.com下载：

```
package
{
    import flash.display.DisplayObject;
    import flash.display.Graphics;
    import flash.events.EventDispatcher;

    public class CollisionGrid extends EventDispatcher
    {
        private var _checks:Vector.<DisplayObject>;
        private var _grid:Vector.<Vector.<DisplayObject>>;
```

www.TopSage.com

大家网 TopSage.com

```
private var _gridSize:Number;
private var _height:Number;
private var _numCells:int;
private var _numCols:int;
private var _numRows:int;
private var _width:Number;


public function CollisionGrid(width:Number, height:Number, gridSize:Number)
{
    _width = width;
    _height = height;
    _gridSize = gridSize;
    _numCols = Math.ceil(_width / _gridSize);
    _numRows = Math.ceil(_height / _gridSize);
    _numCells = _numCols * _numRows;
}

public function drawGrid(graphics:Graphics):void
{
    // make lines to represent grid
    graphics.lineStyle(0, .5);
    for(var i:int = 0; i <= _width; i += _gridSize)
    {
        graphics.moveTo(i, 0);
        graphics.lineTo(i, _height);
    }
    for(i = 0; i <= _height; i += _gridSize)
    {
        graphics.moveTo(0, i);
        graphics.lineTo(_width, i);
    }
}

public function check(objects:Vector.<DisplayObject>):void
{
    var numObjects:int = objects.length;
    _grid = new Vector.<Vector.<DisplayObject>>(_numCells);
    _checks = new Vector.<Vector.<DisplayObject>>();
    // loop through all objects
    for(var i:int = 0; i < numObjects; i++)
    {
        var obj:DisplayObject = objects[i];
        // use a single index to represent positon in
        // one dimensional grid
        var index:int = Math.floor(obj.y / _gridSize) *
                    _numCols + Math.floor(obj.x / _gridSize);
        // only create cell here if an object is assigned to i
        if(_grid[index] == null)
        {
            _grid[index] = new Vector.<DisplayObject>;
        }
        // put the object in the cell
```

www.TopSage.com

```
            _grid[index].push(obj);
        }

        checkGrid();
    }

    private function checkGrid():void
    {
        // loop through each cell of grid
        for(var i:int = 0; i < _numCols; i++)
        {
            for(var j:int = 0; j < _numRows; j++)
            {
                // all the objects in the first cell
                // against each other
                checkOneCell(i, j);
                checkTwoCells(i, j, i + 1, j);     // right
                checkTwoCells(i, j, i - 1, j + 1); // lower left
                checkTwoCells(i, j, i,     j + 1); // lower
                checkTwoCells(i, j, i + 1, j + 1); // lower right
            }
        }
    }

    private function checkOneCell(x:int, y:int):void
    {
        // get cell represented by x, y
        var cell:Vector.<DisplayObject> = _grid[y * _numCols + x];
        if(cell == null) return;

        // how many objects in cell
        var cellLength:int = cell.length;

        // compare all objects to each other
        for(var i:int = 0; i < cellLength - 1; i++)
        {
            var objA:DisplayObject = cell[i];
            for(var j:int = i + 1; j < cellLength; j++)
            {
                var objB:DisplayObject = cell[j];
                _checks.push(objA, objB);
            }
        }
    }

    private function checkTwoCells(x1:int, y1:int, x2:int, y2:int):void

    {
        // make sure cell exists in grid
        if(x2 >= _numCols || x2 < 0 || y2 >= _numRows) return;
        // get each cell, make sure there are objects in each
        var cellA:Vector.<DisplayObject> = _grid[y1 * _numCols + x1];
        var cellB:Vector.<DisplayObject> = _grid[y2 * _numCols + x2];
```

www.TopSage.com

```
            if(cellA == null || cellB == null) return;
            var cellALength:int = cellA.length;
            var cellBLength:int = cellB.length;

            // compare all objs in one cell to all in the other
            for(var i:int = 0; i < cellALength; i++)
            {
                var objA:DisplayObject = cellA[i];
                for(var j:int = 0; j < cellBLength; j++)
                {
                    var objB:DisplayObject = cellB[j];
                    _checks.push(objA, objB);
                }
            }
        }

        public function get checks():Array
        {
            return _checks;
        }
    }
}
```

通过前一个例子，你应该已经对以上的大部分代码很熟悉了。这里做了很多工作使之尽可能优化，特别是向量的使用。向量是Flash 10新增的特性，实际上就是相当于一些具有类型的数组。由于编译器知道向量中的各个元素都是同一个类型，因此能创建更为高效的字节码，在运行时可以更快地执行。从数组改为向量几乎能使这个应用的性能加倍！

drawGrid()方法与前面看到的版本没有任何不同，仍然只是绘制一个网格！

check()方法是这个类中需要交互的主要公共方法。为之传入一个DisplayObject向量。选择使用DisplayObject是因为用于碰撞检测的对象往往是一些sprite[①]、视频片段、形状或位图，这些都由继承自DisplayObject的类来定义。DisplayObject都有x和y属性，需要由此确定其位置以及它们在网格中的位置。如果使用一个并非由DisplayObject继承的定制类，一定要修改此类中的类型。

check()方法创建了一个名为_grid的向量以及另一个名为_checks的向量。对于_grid我们已经很熟悉了，不过其实现则稍有不同——这里实现为一个一维向量（索引的实现有些技巧），而不是一个二维数组。这样做是因为循环处理一个数组比循环处理两个数组的速度更快，而且使用的资源也更少，稍后还会更详细地分析这一点。_checks向量用于存储一组需要进行碰撞测试的对象。需要指出，CollisionGrid类本身并不处理具体的碰撞测试。它只是建立网格，将对象分配到网格，并生成一组可能发生碰撞的对象。要由作为类用户的你来进行处理，完成具体的碰撞检测。

接下来，循环处理传入的DisplayObject向量，并将各对象分配到网格。可能需要对这部分代码多做一些解释：

① sprite意指精灵，专用来代表游戏中的动画角色。——译者注

www.TopSage.com

TopSage.com

```
for(var i:int = 0; i < numObjects; i++)
{
    var obj:DisplayObject = objects[i];
    // use a single index to represent positon in
    // one dimensional grid
    var index:int = Math.floor(obj.y / _gridSize) * _numCols +
                    Math.floor(obj.x / _gridSize);
    // only create cell here if an object is assigned to i
    if(_grid[index] == null)
    {
        _grid[index] = new Vector.<DisplayObject>;
    }
    // put the object in the cell
    _grid[index].push(obj);
}
```

再次申明，我们使用了一个一维向量而不是二维数组。index变量明确该向量的哪个元素对应一个特定的行列号。基本公式如下所示，这里x和y分别是整数类型的列和行索引：

```
index = y * numColumns + x
```

图1-21可以帮助我们解释这个索引公式。

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24

图1-21 使用一个简单的平面数组作为网格

图上的网格有5行5列。对象位于第2行第3列中，也就是x＝3，y＝2。因此，索引为2 * 5＋3即13，这正是在网格中所看到的位置。

在这个代码中，仍采用之前使用的方法得到列和行索引：

```
Math.floor(obj.x / _gridSize)
```

和

```
Math.floor(obj.y / _gridSize)
```

唯一的区别在于所有工作都在一行代码中完成：

```
index = Math.floor(obj.y / _gridSize) * _numCols + Math.floor(obj.x / _gridSize);
```

下一个语句是另一个优化技巧。避免循环处理并为每个网格单元格创建一个向量，因为很多向量从来都不会用到，这里只是检查是否存在一个向量。如果没有则创建。这种方式是一种懒实例化。在这种情况下，“懒惰”并不是一个贬义词，它只是指延迟到真正需要时才创建内容的行为。有些情况下这是合适的，但有些情况下这种做法并不适用。不过在这里使用这种懒方法不失为一个妙方：

```
if(_grid[index] == null)
{
    _grid[index] = new Vector.<DisplayObject>;
}
```

当执行到这个循环的最后一行代码时，就可以知道向量已经存在，并且向其中压入对象：

```
_grid[index].push(obj);
```

check()方法的最后一行代码调用了checkGrid()，下面再次列出这个方法：

```
private function checkGrid():void
{
    // loop through each cell of grid
    for(var i:int = 0; i < _numCols; i++)
    {
        for(var j:int = 0; j < _numRows; j++)
        {
            // all the objects in the first cell against each other
            checkOneCell(i, j);
            checkTwoCells(i, j, i + 1, j);     // right
            checkTwoCells(i, j, i - 1, j + 1); // lower left
            checkTwoCells(i, j, i,     j + 1); // lower
            checkTwoCells(i, j, i + 1, j + 1); // lower right
        }
    }
}
```

这与前一个例子没有什么不同。

checkOneCell()和checkTwoCells90()两个方法本质上基本相同。当然，它们使用了一维向量而不是二维数组，而且由于只是在确实需要时才为单元格创建向量，因此首先查看单元格是否为null。还要注意，这里没有进行碰撞检测，而只是将要检查的两个对象压入到_checks向量。

执行到最后时，_checks将包含一组需要相互检查的对象。这是一个简单的列表，其中每两个连续元素都需要进行检查（也就是说，需要检查元素0与元素1，元素2与元素3，元素4与元素5，依此类推）。最后我们提供了一个名为checks的公共获取方法来访问这个列表。

使用类

现在我们有了这个“可爱”的类，下面来看它的实际使用。在这里，我修改了主类，将其命名为GridCollision2。同样地，这个类也可以从www.friendsofed.com下载。

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

www.TopSage.com

```
import flash.display.StageAlign;
import flash.display.StageScaleMode;
import flash.utils.getTimer;
import flash.display.DisplayObject;

public class GridCollision2 extends Sprite
{
    private const GRID_SIZE:Number = 80;
    private const RADIUS:Number = 25;

    private var _balls:Vector.<DisplayObject>;
    private var _grid:CollisionGrid;
    private var _numBalls:int = 100;

    public function GridCollision2()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        _grid = new CollisionGrid(stage.stageWidth, stage.stageHeight, GRID_SIZE);
        _grid.drawGrid(graphics);

        makeBalls();

        var startTime:int;
        var elapsed:int;

        // loop 10 times and measure elapsed time
        startTime = getTimer();
        for(var i:int = 0; i < 10; i++)
        {
            _grid.check(_balls);
            var numChecks:int = _grid.checks.length;
            for(var j:int = 0; j < numChecks; j += 2)
            {
                checkCollision(_grid.checks[j] as Ball, _grid.checks[j + 1] as Ball);
            }
        }
        elapsed = getTimer() - startTime;
        trace("Elapsed:", elapsed);
    }


    private function makeBalls():void
    {
        // create balls, add to display list and _balls list
        _balls = new Vector.<DisplayObject>(_numBalls);
        for(var i:int = 0; i < _numBalls; i++)
        {
            var ball:Ball = new Ball(RADIUS);
            ball.x = Math.random() * stage.stageWidth;
            ball.y = Math.random() * stage.stageHeight;
            ball.vx = Math.random() * 4 - 2;
```

www.TopSage.com

大家网 TopSage.com

```
                ball.vy = Math.random() * 4 - 2;
                addChild(ball);
                _balls[i] = ball;
            }
        }

        private function checkCollision(ballA:Ball, ballB:Ball):void
        {
            // perform collision ctest between two individual balls
            var dx:Number = ballB.x - ballA.x;
            var dy:Number = ballB.y - ballA.y;
            var dist:Number = Math.sqrt(dx * dx + dy * dy);
            if(dist < ballA.radius + ballB.radius)
            {
                ballA.color = 0xff0000;
                ballB.color = 0xff0000;
            }
        }
    }
}
```

主要改变的是构造函数。由于建立和绘制网格封装在CollisionGrid类中，而且每个操作分别只需一行代码，因此这里删除了单独的makeGrid()和drawGrid()方法，而直接在构造函数中完成这些工作。

接下来调用了makeBalls()方法。这与前例基本相同，只不过使用了一个向量而不是数组。

这里有一个定时器循环，因为我只对测试这个新类感兴趣。当然也可以为基本检查再增加一个定时器循环来进行比较，不过你应该已经对基本检查所需的时间有所了解。这个循环中调用了_grid.check(_balls)。我们已经深入分析了这个方法。执行结束时，可以知道_grids.checks包含要比较的对象的一个向量。下一个for循环显示了如何使用这个向量：

```
for(var j:int = 0; j < numChecks; j += 2)
{
    checkCollision(_grid.checks[j] as Ball, _grid.checks[j + 1] as Ball);
}
```

这里循环处理该向量，每次递增2。然后用索引j和j + 1得到后面两项的引用。将这两项的类型强制转换为Ball，并把它们发送到checkCollision()方法，这个方法未做任何修改。

在我的测试中，这个应用运行的速度是前例的两倍，这主要是因为使用了向量，以及只是在确实需要时才为单元格创建向量（懒实例化），这一点可能也对性能提高有一定帮助。我们已经做好准备迎接这一关键时刻，也就是说可以开始实现动画了。

实现动画实际上与单个碰撞测试并无太大不同，其实你经常做这个工作。另外，由于我们预先针对场景大小和对象数调整了引擎，而且已经确定了理想的网格大小，因此可以去除定时循环。但是要记住，在一个新工程中再次使用这种碰撞网格时或者修改这个应用的参数时，你可能会希望再次完成定时测试来进行优化调整。

我们的主类修改如下，可以在本书下载页面找到相应代码：

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

www.TopSage.com

```
import flash.display.StageAlign;
import flash.display.StageScaleMode;
import flash.display.DisplayObject;
import flash.events.Event;

public class GridCollision3 extends Sprite
{
    private const GRID_SIZE:Number = 80;
    private const RADIUS:Number = 25;

    private var _balls:Vector.<DisplayObject>;
    private var _grid:CollisionGrid;
    private var _numBalls:int = 100;

    public function GridCollision3()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        _grid = new CollisionGrid(stage.stageWidth, stage.stageHeight, GRID_SIZE);
        _grid.drawGrid(graphics);

        makeBalls();
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    function onEnterFrame(event:Event):void
    {
        // add velocity to position of each ball
        updateBalls();
        // determine which balls need to be checked
        _grid.check(_balls);
        var numChecks:int = _grid.checks.length;
        for(var j:int = 0; j < numChecks; j += 2)
        {
            // check each pair
            checkCollision(_grid.checks[j] as Ball, _grid.checks[j + 1] as Ball);
        }
    }

    private function makeBalls():void
    {
        // create all the balls
        _balls = new Vector.<DisplayObject>(_numBalls);
        for(var i:int = 0; i < _numBalls; i++)
        {
            var ball:Ball = new Ball(RADIUS);
            ball.x = Math.random() * stage.stageWidth;
            ball.y = Math.random() * stage.stageHeight;
            ball.vx = Math.random() * 4 - 2;
            ball.vy = Math.random() * 4 - 2;
            addChild(ball);
            _balls[i] = ball;
        }
```

www.TopSage.com

```
        }

        private function updateBalls():void
        {
            for(var i:int = 0; i < _numBalls; i++)
            {
                // move each ball, bounce off walls
                var ball:Ball = _balls[i] as Ball;
                ball.update();
                if(ball.x < RADIUS)
                {
                    ball.x = RADIUS;
                    ball.vx *= -1;
                }
                else if(ball.x > stage.stageWidth - RADIUS)
                {
                    ball.x = stage.stageWidth - RADIUS;
                    ball.vx *= -1;
                }
                if(ball.y < RADIUS)
                {
                    ball.y = RADIUS;
                    ball.vy *= -1;
                }
                else if(ball.y > stage.stageHeight - RADIUS)
                {
                    ball.y = stage.stageHeight - RADIUS;
                    ball.vy *= -1;
                }
                ball.color = 0xffffff;
            }
        }

        private function checkCollision(ballA:Ball, ballB:Ball):void
        {
            // perform check between two individual balls
            var dx:Number = ballB.x - ballA.x;
            var dy:Number = ballB.y - ballA.y;
            var dist:Number = Math.sqrt(dx * dx + dy * dy);
            if(dist < ballA.radius + ballB.radius)
            {
                ballA.color = 0xff0000;
                ballB.color = 0xff0000;
            }
        }
    }
}
```

这里的主要修改是增加了onEnterFrame()方法和updateBalls()方法。现在构造函数只是创建网格和球，并建立ENTER_FRAME监听器。onEnterFrame()方法通过updateBalls()方法更新各个球的位置，并且采用与之前完全一致的方法完成基于网格的碰撞检测。updateBalls()方法只是移动对象，将对象分布在屏幕上，并且在每一帧开始时将各个球的颜色改回为白色。仅当

www.TopSage.com

checkCollision()方法中检查到球发生碰撞时才会将其重置为红色。

当然，这里可以使用不同类型的碰撞反应来达到一种效果，看上去就好像球确实发生了相互反弹。这个内容在《Flash ActionScript 3.0 动画教程》的“Conservation of Momentum”（动量守恒）一节中做了详尽的介绍。

通过这种碰撞检测方法，可以增加场景中能同时移动而且相互之间作用多次的对象的个数。要记住这些优化提示，而且要测试、测试、再测试，以确保获得最佳性能以及准确的检测。

1.2.5 碰撞检测：并不只是碰撞

考虑**碰撞检测**（collision detection）这个词时，很自然地只会想到两个对象相互碰撞。不过，使用基于距离的方法时，有时从两个对象之间的空间关系来考虑会更好。也许你不仅对碰撞本身感兴趣，还想知道两个对象相互之间是否在一定距离内。例如，有一种游戏，敌人必须在“好人”的一定像素范围内，“好人”才能发现这个敌人。

为此，我们仍然可以使用一种基于网格的碰撞检测设置，但是并不是由对象大小来确定网格大小，此时两个对象之间的关键距离更为重要。在《Flash ActionScript 3.0 动画教程》中，我给出了一个粒子交互的例子，这个例子受到了Jared Tarbell的Node Garden（www.levitated.net）的启发。我意识到这将是一个转换为基于网格设置的绝好例子，所以照此完成了开发。首先来看原来的代码：

```
package {
    import flash.display.Sprite;
    import flash.display.StageScaleMode;
    import flash.display.StageAlign;
    import flash.events.Event;
    import flash.geom.Point;

    [SWF(backgroundColor=0x000000)]
    public class NodeGardenLines extends Sprite
    {
        private var particles:Array;
        private var numParticles:uint = 30;
        private var minDist:Number = 100;
        private var springAmount:Number = .001;

        public function NodeGardenLines()
        {
            init();
        }

        private function init():void
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;
            particles = new Array();
            for(var i:uint = 0; i < numParticles; i++)
            {
                // create all the particles
```

www.TopSage.com

```
            var particle:Ball = new Ball(3, 0x******);
            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);
    }

    private function onEnterFrame(event:Event):void
    {
        graphics.clear();
        for(var i:uint = 0; i < numParticles; i++)
        {
            // update each particle wrap around screen
            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++)
            {
                // spring each particle to each other particle
                var partB:Ball = particles[j];
                spring(partA, partB);
            }
        }
    }
    private function spring(partA:Ball, partB:Ball):void
    {
        var dx:Number = partB.x - partA.x;
        var dy:Number = partB.y - partA.y;
```

www.TopSage.com

```
                var dist:Number = Math.sqrt(dx * dx + dy * dy);
                if(dist < minDist)
                {
                    // if particles are close enough, draw a line
                    // and spring them toward each other
                    graphics.lineStyle(1, 0xffffff, 1 - dist / minDist);
                    graphics.moveTo(partA.x, partA.y);
                    graphics.lineTo(partB.x, partB.y);
                    // acceleration is proportional to distance
                    var ax:Number = dx * springAmount;
                    var ay:Number = dy * springAmount;
                    // add accel to each particle, in opposite directions
                    partA.vx += ax;
                    partA.vy += ay;
                    partB.vx -= ax;
                    partB.vy -= ay;
                }
            }
        }
    }
```

我并不打算深入地解释这个代码。这里主要创建了一些粒子，并四处移动。如果两个粒子相互之间的距离小于一个指定距离，它们就会彼此发生反弹，而且二者之间会画一条线。粒子越接近，它们之间的吸引力就越强，所画的线条也就越明亮。还要注意，利用类最上面的SWF元数据已经将场景颜色设置为黑色：

```
[SWF(backgroundColor=0x000000)]
```

这个元数据适用于Flex Builder或Flash CS4，但Flash CS3会将其忽略，这就要求你在Properties Inspector中手动地设置背景颜色。不过，由于我们很快就要使用Flash 10特有的特性（向量），因此这里假设你使用的是Flash CS4。

这个文件还使用了一个名为Ball的类，它与本章目前为止使用的Ball类稍有差别，但影响不大。

这个例子使用了30个粒子，如果粒子之间的距离等于或小于100像素就会发生相互作用。采用目前的方法能很好地工作，而不需要任何改进。不过下面改变几个参数，看看参数超出其适宜范围时会发生什么情况：

```
private var numParticles:uint = 500;
private var minDist:Number = 50;
```

我们把粒子数增加到500，距离则减小到50。如果使用的是Flash创作环境，可能希望场景更大一些以便处理所有粒子，而不会让粒子堆积在一起，比如说1 000×800。现在运行文件时，会看到性能显著下降，可能达到每秒几帧。

现在来看实现基于网格的碰撞检测有什么帮助。完成所有粒子之间检查的循环代码用粗体显示。这一部分需要替换为网格方法。可能还可以将数组改为向量来得到一些改进。如果你很好奇，可以首先尝试将Array的各个实例改为Vector.<DisplayObject>，看看能得到多少改进。还需要导入flash.display.DisplayObject类，并在访问对象时将对象强制置换为Ball类，如下：

www.TopSage.com

大家网 TopSage.com

```
var particle:Ball = particles[i] as Ball;
```

在我的测试中，尽管看起来有一些改进，但收效甚微。对于有500个元素的双重循环，每一帧会有24 750次检查。减少这个检查次数至关重要，因为其中只有很小一部分有可能发生碰撞。实现基于网格的碰撞检测将使检查次数减少到小于10 000。以下是名为NodeGardenGrid的新类中的代码（只有粗体显示的代码行相对于原代码来说有所修改）：

```
package {
    import flash.display.DisplayObject;
    import flash.display.Sprite;
    import flash.display.StageScaleMode;
    import flash.display.StageAlign;
    import flash.events.Event;
    import flash.geom.Point;

    [SWF(backgroundColor=0x000000)]
    public class NodeGardenGrid extends Sprite
    {
        private var particles:Vector.<DisplayObject>;
        private var numParticles:uint = 500;
        private var minDist:Number = 50;
        private var springAmount:Number = .001;
        private var grid:CollisionGrid;

        public function NodeGardenGrid()
        {
            init();
        }

        private function init():void
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            grid = new CollisionGrid(stage.stageWidth, stage.stageHeight, 50);
            particles = new Vector.<DisplayObject>();
            for(var i:uint = 0; i < numParticles; i++)
            {
                // create all particles
                var particle:Ball = new Ball(3, 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);
        }

        private function onEnterFrame(event:Event):void
        {
```

www.TopSage.com

```
        graphics.clear();
        for(var i:uint = 0; i < numParticles; i++)
        {
            // update particles, wrap around stage
            var particle:Ball = particles[i] as Ball;
            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;
            }
        }
        // determine which particles need to be checked
        grid.check(particles);
        var checks:Vector.<DisplayObject> = grid.checks;
        trace(checks.length);
        var numChecks:int = checks.length;
        for(i=0; i < numChecks; i += 2)
        {
            // spring each pair toward each other
            var partA:Ball = checks[i] as Ball;
            var partB:Ball = checks[i + 1] as Ball;
            spring(partA, partB);
        }
    }
    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, 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.vy += ay;
            partB.vx -= ax;
            partB.vy -= ay;
        }
```

www.TopSage.com

```
            }
        }
    }
```

要确保设置视频发布为Flash 10格式，而且CollisionGrid.as文件必须在类路径中。在我的测试中，这会得到每帧大约6 500次检查。注意，创建网格时网格大小为50，这与minDist值相同。这个代码并不依赖于粒子的半径，而只依赖于粒子之间的距离。不过要记住，你可能希望完成一些具体的定时测试，并调整网格大小来得到最佳性能。网格较大会导致更多的检查，但是却可以减少对数组的循环处理次数。肯定存在一个最佳性能黄金点。不过即使是最小值50，性能也有显著的提高。

1.3　小结

以上讨论的是高级碰撞检测。这两种技术（基于位图和基于网格的碰撞检查）有很多不同的用法，这里给出的例子只是让你有所认识，可以由此起步。重申一次，最重要的一点（特别是在基于网格的系统中）是测试、测试、再测试，度量性能，然后优化调整碰撞引擎来得到最佳结果。而且要记住，对象数目很大时这种方法会很凑效，但是如果对象数目较小，则有些得不偿失，其开销反而会超过它带来的好处。

第2章将讨论一个完全不同的主题：转向行为。我们还将涉及一些人工智能的内容。

www.TopSage.com

大家网
TopSage.com

第2章 转向行为

转向行为（steering behavior）指的是一组算法，它允许对象采用一种看似智能的方式四处移动。这些行为都可以归入**人工智能**（artificial intelligence）和**人工生命**（artificial life）的范畴，使角色看起来好像被赋予了生命，能够决定要移动到哪里、何时移动以及以何种方式移动，从而能到达目标，追捕或逃避其他角色，避开障碍物，追随路径，等等。

这是Craig Reynolds创造的一个术语，首次出现于他在1999年游戏开发者大会上发表的一篇名为*Steering Behaviors for Autonomous Characters*（自主角色的转向行为）的文章。文章中描述了大量算法，用来创建一个动力系统，可以应用于游戏和模拟中的角色。这些动力用于影响角色的行动，从而完成角色需要实现或希望实现的各种任务。其中还包括多种群组动力学，如群落（flocking）[①]。实际上，Reynolds已经因其对鸟群的“boids”模拟而闻名（参见图2-1）。

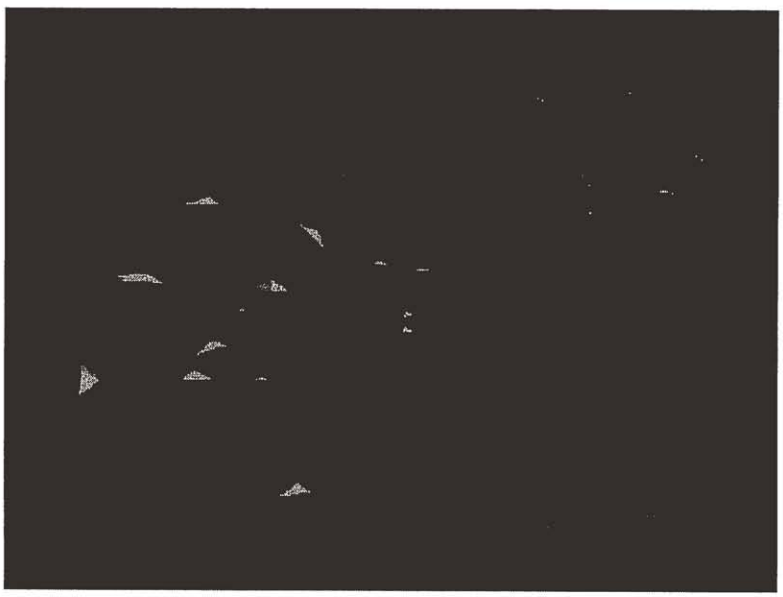


图2-1 Craig Reynolds的“boids”模拟

① 也称为群集。——译者注

www.TopSage.com

写这本书时，可以在www.red3d.com/cwr/steer/上查到Reynolds的文章，不过只需快速搜索这个标题，就能得到很多下载地址。在这篇文章中，Reynolds并没有深入地涉及算法的实现，但上述网页链接了一个关于行为的C++版本开源网站：OpenSteer（http://opensteer.sourceforge.net/）。不过，这篇文章中描述的系统已经成为此后创建的许多（甚至大多数）人工智能运动引擎的基础。本章也将跟随其步伐，为文章中描述的大部分行为创建一个基于ActionScript 3.0的实现版本。

在深入讨论之前，首先明确本章的目标是介绍行为，了解行为，并展示一个实现这些行为的框架。后面一些更繁杂的行为可以采用多种不同的方式实现，复杂度可能也大不相同。并不存在一种适用于所有行为的“标准”或“正确”的方法，这里给出的实现只是最简单的形式。换句话说，可以认为这里给出的方法只是一个入门或起点。要是用于产品应用开发中的话，可能需要对这里提供的代码做大幅改进，还要针对你的特定需要做相应调整。

2.1　行为

首先对基本行为做一个概要介绍，指出每种行为具体做什么，以及使用它是为了什么。

- 搜寻（seek）：角色试图移动到一个指定点，这可能是一个固定点，也可能是一个移动的目标，如另一个角色。
- 逃避（flee）：与搜寻正好相反，角色试图尽可能远离给定点。同样，这个点可以是固定点，也可以是移动点。
- 到达（arrive）：与搜寻基本相同，只不过一旦与目标点足够接近，角色的速度会减慢，完成一个缓动[①]从而最终恰好在目标点停止。
- 追捕（pursue）：这是对搜寻的一种改进，因为要考虑到目标的速度。这里并非搜寻目标目前所在的点，角色会预测目标将去往哪里，然后移动到该点。显然，这只适用于移动目标，因为固定点没有速度。
- 躲避（evade）：与追捕正相反，角色根据目标的速度预测目标点的位置，并试图尽可能远离这个点。
- 漫游（wander）：这是一种随机但却平滑而又真实的动作。
- 对象规避（object avoidance）：角色可以感知到有对象在其路径上，并转向避开。
- 路径追随（path following）：角色尽其所能沿着一个给定的路径移动，但是会考虑物理方面以及所应用的其他行为的真实性。

除了这些行为，还有一种复杂的复合行为，称为**群落**（flocking），可以模拟类似角色的群组行为，通常应用另外3种简单行为来创建。

- 分离（separation）：群组中的各个角色都尽量与其相邻角色保持一个可接受的距离。
- 内聚（cohesion）：每个角色尽量保证与群落中的其他成员不会相距太远。
- 排队（alignment）：每个角色尽量与其相邻角色沿相同方向行进。

① 缓动（easing motion）是指动画中渐近加速或渐近减速的运动效果。——译者注

www.TopSage.com

尽管这3个行为相当简单，但结合在一起时会产生让人难以想象的群组运动效果，使你感觉确实在处理鸟群或其他生物群体。调整这3种子行为的各个参数可以改变群组的角色，从而得到不同类型的群组，可以是一个经常分散的松散群组，或者是一个紧密的群体，也可能是由一个领导者带队的队伍，还可以有很多其他变化。

不过，在具体讨论移动对象的行为之前，首先需要有一种移动对象的方法。

2.2 Vector2D 类

前面介绍的转向行为已经采用多种不同语言实现过多次（可能用ActionScript就曾实现过一两次）。原先使用向量来描述（你会发现大多数实现都采用了这种做法）。如果想更多地了解向量，可以参考《Flash ActionScript 3.0 动画教程》的第5章，其中有更详细的介绍。

简单地说，**向量**（vector）描述了一个包含两部分的量：方向和大小。例如，移动对象的速度就包含移动的路径（方向）及其速率[①]（大小）。因此，速度非常适合表述为向量。加速度（用来改变对象速度的作用力）同样包含这个作用力的大小和方向（也是一个向量）。向量还很适合描述两个对象之间的相对位置，在这里大小就是二者之间的距离，方向则是一个对象到另一个对象的角度。

另外，可以使用向量来描述一个角色的行进方向或朝向。在这种情况下，不考虑大小而只涉及方向，所以可以将大小设置为1.0。这种向量称为**单位向量**（unit vector），由于它的长度只有一个单位，因此能够完成高度优化的向量数学运算。

向量的所有这些特性对转向行为来说都非常有用，因为转向行为大量使用了速度、方向力、对象之间的距离以及对象朝向等向量。所以如果ActionScript 3.0中有一个向量类对于创建转向行为会很有帮助。另外，作为一个好心人，我提前为你创建了这样一个类。当然，后来我发现Flash CS4内置有它自己的Vector3D类，所以也许你希望最后移植为使用这个类。不过，我也发现增加一维会大大提升复杂性，所以决定在本章中仍然使用我自己的Vector2D类（一个更轻量级的版本）。在此我不会深入解释这个类，不过其中包含的大多数方法都可以在其他语言的相应类中见到，特别是创建转向行为使用的那些方法。这里将全面介绍这个类，因为它可以作为一个很好的参考。使用这个类的各个方法时我还会更详细地对其进行阐释。可以从本书下载页面（www.friendsofed.com）下载这个类（Vector2D.as文件）。

```
package com.foed
{
    import flash.display.Graphics;

    /**
     * A basic 2-dimensional vector class.
     */
    public class Vector2D
    {
```

① 本书中velocity和speed交替出现，通常均指速度，不做区分。但特殊情况下会强调speed为标量，即“速率”，而velocity泛指作为向量的“速度”。——译者注

www.TopSage.com

大家网 TopSage.com

```
private var _x:Number;
private var _y:Number;

/**
 * Constructor.
 */
public function Vector2D(x:Number = 0, y:Number = 0)
{
    _x = x;
    _y = y;
}

/**
 * Can be used to visualize the vector.
 * Generally used for debug purposes only.
 * @param graphics The Graphics instance to draw the vector on.
 * @param color The color of the line used to for the vector.
 */
public function draw(graphics:Graphics, color:uint = 0):void
{
    graphics.lineStyle(0, color);
    graphics.moveTo(0, 0);
    graphics.lineTo(_x, _y);
}

/**
 * Generates a copy of this vector.
 * @return Vector2D A copy of this vector.
 */
public function clone():Vector2D
{
    return new Vector2D(x, y);
}

/**
 * Sets this vector's x and y values, and thus length, to zero.
 * @return Vector2D A reference to this vector.
 */
public function zero():Vector2D
{
    _x = 0;
    _y = 0;
    return this;
}

/**
 * Whether or not this vector is equal to zero,
 * i.e. its x, y, and length are zero.
 * @return Boolean True if vector is zero, otherwise false.
 */
public function isZero():Boolean
{
    return _x == 0 && _y == 0;
}
```

www.TopSage.com

```
/**
 * Sets / gets the length or magnitude of this vector.
 * Changing the length will change the x and y but not
 * the angle of this vector.
 */
public function set length(value:Number):void
{
    var a:Number = angle;
    _x = Math.cos(a) * value;
    _y = Math.sin(a) * value;
}
public function get length():Number
{
    return Math.sqrt(lengthSQ);
}

/**
 * Gets the length of this vector, squared.
 */
public function get lengthSQ():Number
{
    return _x * _x + _y * _y;
}

/**
 * Gets / sets the angle of this vector.
 * Changing the angle changes the x and y but retains the
 * same length.
 */
public function set angle(value:Number):void
{
    var len:Number = length;
    _x = Math.cos(value) * len;
    _y = Math.sin(value) * len;
}
public function get angle():Number
{
    return Math.atan2(_y, _x);
}

/**
 * Normalizes this vector. Equivalent to setting
 * the length to one, but more efficient.
 * @return Vector2D A reference to this vector.
 */
public function normalize():Vector2D
{
    if(length == 0)
    {
        _x = 1;
        return this;
    }
    var len:Number = length;
    _x /= len;
```

www.TopSage.com

```
        _y /= len;
        return this;
    }

    /**
     * Ensures the length of the vector is no longer
     * than the given value.
     * @param max The maximum value this vector should be.
     * If length is larger than max, it will be truncated
     * to this value.
     * @return Vector2D A reference to this vector.
     */
    public function truncate(max:Number):Vector2D
    {
        length = Math.min(max, length);
        return this;
    }

    /**
     * Reverses the direction of this vector.
     * @return Vector2D A reference to this vector.
     */
    public function reverse():Vector2D
    {
        _x = -_x;
        _y = -_y;
        return this;
    }

    /**
     * Whether or not this vector is normalized,
     * i.e. its length is equal to one.
     * @return Boolean True if length is one, otherwise false.
     */
    public function isNormalized():Boolean
    {
        return length == 1.0;
    }

    /**
     * Calculates the dot product of this vector and
     * another given vector.
     * @param v2 Another Vector2D instance.
     * @return Number The dot product of this vector and
     * the one passed in as a parameter.
     */
    public function dotProd(v2:Vector2D):Number
    {
        return _x * v2.x + _y * v2.y;
    }

    /**
     * Calculates the angle between two vectors.
     * @param v1 The first Vector2D instance.
     * @param v2 The second Vector2D instance.
```

www.TopSage.com

TopSage.com


```
 * @return Number the angle between the two given vectors.
 */
public static function angleBetween(v1:Vector2D, v2:Vector2D):Number
{
    if(!v1.isNormalized()) v1 = v1.clone().normalize();
    if(!v2.isNormalized()) v2 = v2.clone().normalize();
    return Math.acos(v1.dotProd(v2));
}

/**
 * Determines if a given vector is to the right or left
 * of this vector.
 * @return int If to the left, returns -1. If to the right, +1.
 */
public function sign(v2:Vector2D):int
{
    return perp.dotProd(v2) < 0 ? -1 : 1;
}

/**
 * Finds a vector that is perpendicular to this vector.
 * @return Vector2D A vector perpendicular to this vector.
 */
public function get perp():Vector2D
{
    return new Vector2D(-y, x);
}

/**
 * Calculates the distance from this vector to another
 * given vector.
 * @param v2 A Vector2D instance.
 * @return Number The distance from this vector to the
 * vector passed as a parameter.
 */
public function dist(v2:Vector2D):Number
{
    return Math.sqrt(distSQ(v2));
}

/**
 * Calculates the distance squared from this vector to
 * another given vector.
 * @param v2 A Vector2D instance.
 * @return Number The distance squared from this vector
 * to the vector passed as a parameter.
 */
public function distSQ(v2:Vector2D):Number
{
    var dx:Number = v2.x - x;
    var dy:Number = v2.y - y;
    return dx * dx + dy * dy;
}
```

www.TopSage.com

```
/**
 * Adds a vector to this vector, creating a new
 * Vector2D instance to hold the result.
 * @param v2 A Vector2D instance.
 * @return Vector2D A new vector containing the
 * results of the addition.
 */
public function add(v2:Vector2D):Vector2D
{
    return new Vector2D(_x + v2.x, _y + v2.y);
}

/**
 * Subtracts a vector from this vector, creating a
 * new Vector2D instance to hold the result.
 * @param v2 A Vector2D instance.
 * @return Vector2D A new vector containing the
 * results of the subtraction.
 */
public function subtract(v2:Vector2D):Vector2D
{
    return new Vector2D(_x - v2.x, _y - v2.y);
}

/**
 * Multiplies this vector by a value, creating a
 * new Vector2D instance to hold the result.
 * @param v2 A Vector2D instance.
 * @return Vector2D A new vector containing the
 * results of the multiplication.
 */
public function multiply(value:Number):Vector2D
{
    return new Vector2D(_x * value, _y * value);
}

/**
 * Divides this vector by a value, creating a new
 * Vector2D instance to hold the result.
 * @param v2 A Vector2D instance.
 * @return Vector2D A new vector containing the
 * results of the division.
 */
public function divide(value:Number):Vector2D
{
    return new Vector2D(_x / value, _y / value);
}

/**
 * Indicates whether this vector and another
 * Vector2D instance are equal in value.
 * @param v2 A Vector2D instance.
 * @return Boolean True if the other vector is
 * equal to this one, false if not.
```

www.TopSage.com

```
         */
        public function equals(v2:Vector2D):Boolean
        {
            return _x == v2.x && _y == v2.y;
        }

        /**
         * Sets / gets the x value of this vector.
         */
        public function set x(value:Number):void
        {
            _x = value;
        }
        public function get x():Number
        {
            return _x;
        }

        /**
         * Sets / gets the y value of this vector.
         */
        public function set y(value:Number):void
        {
            _y = value;
        }
        public function get y():Number
        {
            return _y;
        }

        /**
         * Generates a string representation of this vector.
         * @return String A description of this vector.
         */
        public function toString():String
        {
            return "[Vector2D (x:" + _x + ", y:" + _y + ")]";
        }
    }
}
```

C或C++这类语言有一个巧妙的特性，称为**操作符重载**（operator overloading），利用操作符重载特性，可以将语言的一个内置操作符（如+或-）映射到类的一个方法。所以不必写为vectorC= vectorA.add(vectorB)，而是可以写做vectorC = vectorA + vectorB。

尽管操作符重载使类看起来更像是基本类型，而且更易于使用，但ActionScript还不支持这个特性，所以我们必须使用稍有些繁琐的实现。

与实现问题紧密相关，构建这样一个类遇到的挑战之一是：需要确定在实例上调用某些类方法时它们会如何工作——在这里，这些类方法包括truncate、normalize、reverse、add、subtract、multiply和divide。对此有两种可能性：如add等方法可能会直接改变调用该方法的对象的内部状态，或者可能返回一个新对象作为向量add操作的结果。

www.TopSage.com

大家网 TopSage.com

例如，假设有一个vectorA（3,2），这说明x值为3，而y值为2，另外有一个vectorB（4,5）。运行以下代码：

```
vectorA.add(vectorB);
```

按照第一种想法，vectorB保持不变，而vectorA将修改为等于（7,7）。

另一种可能则如下所示：

```
vectorC = vectorA.add(vectorB);
```

在这里，vectorA和vectorB都将保持其原始值（3,2）和（4,5），新创建的vectorC将等于（7,7）。

那么哪一种才是add方法的正确做法呢？创建类时我对此反反复复做了多番考虑，最后发现在很多情况下需要把一个对象（如其位置或速度）用向量来表示，并对这个值完成某些数学运算，而对于对象本身不会做任何改变。所以add、subtract、multiply和divide仍保持原对象不变。

不过，truncate、reverse和normalize等操作则不同，它们在很大程度上意味着要对对象本身做某种处理。所以这些方法确实会修改内部值，并返回对象本身的一个实例，这个对象实例会很有用。

如果要以相反的方式实现以上行为也并不困难。例如，如果希望将vectorB增加到vectorA，并由vectorA得到加法操作的结果，只需执行以下代码：

```
vectorA = vectorA.add(vectorB);
```

如果需要得到一个向量的截断、逆置或归一值，而不改变原向量，可以使用clone方法，如下所示：

```
normalizedA = vectorA.clone().normalize();
```

这个方法会创建vectorA的一个副本，并对其完成归一，保持原vectorA不变。

Seb Lee-Delisle在为本书做技术审校时指出，采用上述做法创建对象的副本可能会导致这样一种情况：有可能会创建很多对象，但只是使用这些对象稍稍做一点工作就要将其销毁。其副作用之一是Flash Player中的垃圾回收器必须跟踪所有这些“死”对象并将其删除，这就会出现性能问题。为清晰起见，这里仍保留了这种做法，不过我们已经认识到这个方面有待优化调整。

以上已经建立了一个向量类来表示角色的位置、速度和各种作用力，下面需要一个表示角色的基类。

2.3　Vehicle类

Vehicle类是转向角色的基类，不过它没有任何转向行为，只是处理一些与运动相关的基本内容，如位置、速度、质量以及角色到达屏幕边缘时发生的情况（可能弹回，也可能再次出现在屏幕的另一侧）。SteeredVehicle类扩展了Vehicle，增加了转向行为。使用这种结构，其目的是为了让Vehicle类用于需要移动但是不需要转向行为的其他类型对象。同时也可以让

www.TopSage.com

SteeredVehicle类只关注转向功能的实现，而不必考虑基本运动的细节。

到目前为止，我们一直使用**角色**（character）表示移动并且拥有行为的对象。从现在起，我会交替使用角色和**机车**（vehicle）这两种说法。也许这样考虑会有帮助：可以把角色认为是驾车的对象，或者甚至把“机车”当作角色本身。如果这样考虑仍不能理解，可以读一读本章开始时提到的文章*Steering Behaviors for Autonomous Characters*中的“Locomotion”一节，其中的解释非常清晰。

所以无需多言，下面给出这个类，可以从本书下载网站下载（Vehicle.as文件）：

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

    /**
     * Base class for moving characters.
     */
    public class Vehicle extends Sprite
    {
        protected var _edgeBehavior:String = WRAP;
        protected var _mass:Number = 1.0;
        protected var _maxSpeed:Number = 10;
        protected var _position:Vector2D;
        protected var _velocity:Vector2D;

        // potential edge behaviors
        public static const WRAP:String = "wrap";
        public static const BOUNCE:String = "bounce";
    /**
     * Constructor.
     */
    public function Vehicle()
    {
        _position = new Vector2D();
        _velocity = new Vector2D();
        draw();
    }

    /**
     * Default graphics for vehicle. Can be overridden in subclass.
     */
    protected function draw():void
    {
        graphics.clear();
        graphics.lineStyle(0);
        graphics.moveTo(10, 0);
        graphics.lineTo(-10, 5);
        graphics.lineTo(-10, -5);
        graphics.lineTo(10, 0);
    }

    /**
     * Handles all basic motion.
```

www.TopSage.com

```
 * Should be called on each frame / timer interval.
 */
public function update():void
{
    // make sure velocity stays within max speed.
    _velocity.truncate(_maxSpeed);

    // add velocity to position
    _position = _position.add(_velocity);

    // handle any edge behavior
    if(_edgeBehavior == WRAP)
    {
        wrap();
    }
    else if(_edgeBehavior == BOUNCE)
    {
        bounce();
    }

    // update position of sprite
    x = position.x;
    y = position.y;
    // rotate heading to match velocity
    // convert radians to degrees: degrees = radians * 180 / PI
    rotation = _velocity.angle * 180 / Math.PI;
}

/**
 * Causes character to bounce off edge if edge is hit.
 */
private function bounce():void
{
    if(stage != null)
    {
        if(position.x > stage.stageWidth)
        {
            position.x = stage.stageWidth;
            velocity.x *= -1;
        }
        else if(position.x < 0)
        {
            position.x = 0;
            velocity.x *= -1;
        }

        if(position.y > stage.stageHeight)
        {
            position.y = stage.stageHeight;
            velocity.y *= -1;
        }
        else if(position.y < 0)
        {
            position.y = 0;
```

www.TopSage.com

TopSage.com

```
            velocity.y *= -1;
        }
    }
}

/**
 * Causes character to wrap to opposite edge if edge is hit.
 */
private function wrap():void
{
    if(stage != null)
    {
        if(position.x > stage.stageWidth) position.x = 0;
        if(position.x < 0) position.x = stage.stageWidth;
        if(position.y > stage.stageHeight) position.y = 0;
        if(position.y < 0) position.y = stage.stageHeight;
    }
}

/**
 * Sets / gets what will happen if character hits edge.
 */
public function set edgeBehavior(value:String):void
{
    _edgeBehavior = value;
}
public function get edgeBehavior():String
{
    return _edgeBehavior;
}

/**
 * Sets / gets mass of character.
 */
public function set mass(value:Number):void
{
    _mass = value;
}
public function get mass():Number
{
    return _mass;
}

/**
 * Sets / gets maximum speed of character.
 */
public function set maxSpeed(value:Number):void
{
    _maxSpeed = value;
}
public function get maxSpeed():Number
{
    return _maxSpeed;
}
```

```
        /**
         * Sets / gets position of character as a Vector2D.
         */
        public function set position(value:Vector2D):void
        {
            _position = value;
            x = _position.x;
            y = _position.y;
        }
        public function get position():Vector2D
        {
            return _position;
        }

            /**
             * Sets / gets velocity of character as a Vector2D.
             */
            public function set velocity(value:Vector2D):void
            {
                _velocity = value;
            }
            public function get velocity():Vector2D
            {
                return _velocity;
            }

            /**
             * Sets x position of character.
             * Overrides Sprite.x to set internal Vector2D position.
             */
            override public function set x(value:Number):void
            {
                super.x = value;
                _position.x = x;
            }

            /**
             * Sets y position of character.
             * Overrides Sprite.y to set internal Vector2D position.
             */
            override public function set y(value:Number):void
            {
                super.y = value;
                _position.y = y;
            }

    }
}
```

如果熟悉《Flash ActionScript 3.0 动画教程》中的基本概念，可以看到这里并没有任何新的概念，不过处理方式有所不同。首先，这里位置和速度没有表示为x和y值（例如，x, y, vx, vy），而是分别使用一个Vector2D来表示：_position和_velocity。

大多数工作都在update方法中完成。首先它对速度完成截断，确保不会超过_maxSpeed；然后

www.TopSage.com

将速度向量加至位置向量。《Flash ActionScript 3.0 动画教程》中采用了以下做法：

```
x += _vx;
y += _vx;
```

而利用向量只需一行代码就可以做到：

```
_position = _position.add(_velocity);
```

这个方法再检查是否到达边缘，并且（如果到达边缘）调用wrap或bounce方法。最后，它会更新Sprite的屏幕位置从而与_position的x和y值匹配，然后调整rotation，这样角色就指向它移动的方向：

```
x = position.x;
y = position.y;
rotation = _velocity.angle * 180 / Math.PI;
```

余下的大部分代码主要是各个保护属性的获取方法和设置方法。这里包括一个默认的draw方法，它在类实例化时调用。可以在子类中覆盖这个方法，为你的角色创建适当的图片，不过利用这个方法可以在图片设置之前使对象在屏幕上可见。

来看Vehicle类的一个快速测试，下面创建一个新的文档类，名为VehicleTest，可以从本书下载网页下载（VehicleTest.as文件）：

```
package
{
    import com.foed.Vector2D;
    import com.foed.Vehicle;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class VehicleTest extends Sprite
    {
        private var _vehicle:Vehicle;

        public function VehicleTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _vehicle = new Vehicle();
            addChild(_vehicle);

            _vehicle.position = new Vector2D(100, 100);

            _vehicle.velocity.length = 5;
            _vehicle.velocity.angle = Math.PI / 4;

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
```

www.TopSage.com

大家网 TopSage.com

```
            _vehicle.update();
        }
    }
}
```

这会创建一个新的Vehicle，并把它添加到显示列表中。这里将位置（position）设置为一个新的Vector2D实例：

```
_vehicle.position = new Vector2D(100, 100);
```

为此另一种方法是直接设置position的x和y值：

```
_vehicle.position.x = 100;
_vehicle.position.y = 100;
```

或者甚至可以利用x和y的覆盖设置方法直接设置机车本身的x值和y值，position向量将会得到这些值：

```
_vehicle.x = 100;
_vehicle.y = 100;
```

这个类还可以使用另一种方式设置速度：设置velocity属性的长度（length）和角度（angle），由此也可以看出使用向量的灵活性：

```
_vehicle.velocity.length = 5;
_vehicle.velocity.angle = Math.PI / 4;
```

在这里length将是速率（速度大小），angle为方向。不要忘记角度以弧度度量，所以Math.PI /4就等于45°。再次指出，如果能确定机车移动的x和y速度，就可以像处理位置一样，用这些值指定一个新的Vector2D，或者直接设置速度向量的x和y属性。

最后，它为ENTER_FRAME事件设置一个监听器，并且每一帧都对机车调用update。这会导致机车沿指定的方向移动，到达屏幕边缘时会折返到屏幕的另一侧，如图2-2所示。

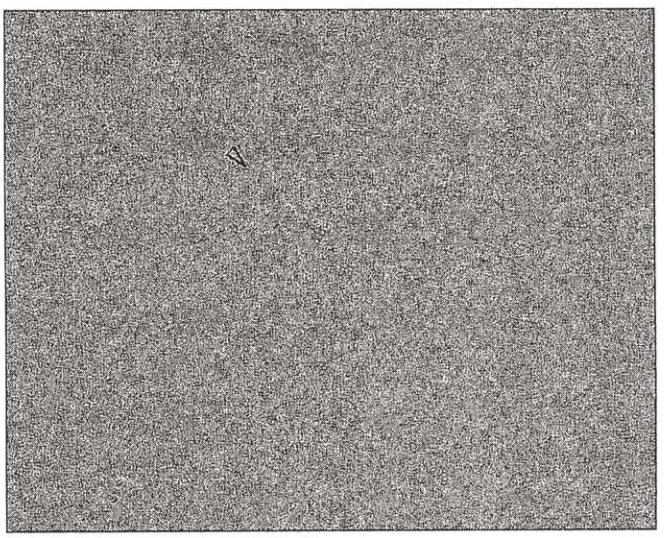


图2-2　一个移动的机车

www.TopSage.com

Vehicle类的测试可以告一段落。下面来讨论更出色更强大的转向行为，这正是本章的重点。

2.4 SteeredVehicle 类

SteeredVehicle类扩展了Vehicle类，并为之增加转向行为。每个行为分别定义为一个公共方法，可以在每一帧调用或者以某个定时器间隔调用，从而应用相应的转向力。通常会应用所有转向力然后再调用机车的update方法。

例如，如果希望创建一个只是四处漫游的机车，可以在每一帧调用其wander方法，然后调用update方法：

```
private function onEnterFrame(event:Event):void
{
    _vehicle.wander();
    _vehicle.update();
}
```

转向方法都采用以下做法：调用一个转向方法时，它会计算**转向力**（steering force），这个作用力会导致机车顺时针或逆时针旋转。例如，seek方法将计算一个作用力，足以让机车从当前朝向转向直接面向搜寻的目标点。可能会有不止一个转向行为作用于给定机车——它可能要搜寻一个点，与此同时还要逃避或躲避另一个机车。所以这些作用力会叠加，最终调用update方法时，机车将取所应用的全部转向行为的总和来改变其速度（方向和速率）。

以下是SteeredVehicle的核心内容，不过这里没有提供任何行为的具体代码（相应文件为SteeredVehicle.as，可以从本书网站下载）：

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

    public class SteeredVehicle extends Vehicle
    {
        private var _maxForce:Number = 1;
        private var _steeringForce:Vector2D;

        public function SteeredVehicle()
        {
            _steeringForce = new Vector2D();
            super();
        }

        public function set maxForce(value:Number):void
        {
            _maxForce = value;
        }
        public function get maxForce():Number
        {
            return _maxForce;
        }

        override public function update():void
```

www.TopSage.com

```
            {
                _steeringForce.truncate(_maxForce);
                _steeringForce = _steeringForce.divide(_mass);
                _velocity = _velocity.add(_steeringForce);
                _steeringForce = new Vector2D();
                super.update();
            }
        }
    }
```

首先看到的是一个_steeringForce属性，这是一个Vector2D。该属性会跟踪由各个行为累加得到的总转向力（注意，这里还有一个_maxForce属性）。在现实生活中不会看到机车或角色“瞬间”转向，所以要对一帧中实际应用的转向力有所限制。利用maxForce获取方法/设置方法可以公开存取这个值。通过调整maxForce，可以让机车急转弯，并迅速准确地移向其目的地，或者也可以让机车绕大圈缓慢调整方向。

现在来分析update方法。假设有多个转向行为方法，而且已经调用了其中一个或多个方法，因此_steeringForce属性包含一个有意义的向量。这个方法首先对_steeringForce完成截断，使之不超过可能的最大转向力。然后将_steeringForce除以机车的质量。在现实生活中，较重的机车会有更大的动量，则需要更大的转弯弧度，而车重较小的机车可以更快地转弯。接下来将转向力加至机车的当前速度，再将_steeringForce变量重置为一个0向量，以便完成下一轮作用力的叠加。最后，调用超类的update方法，完成Vehicle类中已有的所有基本动作。

现在来看各个行为的实现，首先介绍搜寻行为。每个行为都将成为SteeredVehicle类的一个新的公共方法。有些行为还需要增加新的类属性或额外的方法，这些会在必要时给出。

2.4.1　搜寻行为

如前所述，搜寻只是让机车移动到一个特定的点。方法如下所示：

```
public function seek(target:Vector2D):void
{
    var desiredVelocity:Vector2D = target.subtract(_position);
    desiredVelocity.normalize();
    desiredVelocity = desiredVelocity.multiply(_maxSpeed);
    var force:Vector2D = desiredVelocity.subtract(_velocity);
    _steeringForce = _steeringForce.add(force);
}
```

首先计算期望速度，也就是能使机车准确到达目标的期望速度，由目标位置减去机车位置就可以得到这个速度。这个向量的含义是：如果沿这个方向移动这么远，就能到达想要的目标。参见图2-3。

当然，现实不是想到就能做到。不过，多尝试几次后就会找出一个看上去不错的算法。在这里，我们将期望速度归一，并乘以最大速率。这样可以得到一个仍指向目标的向量，但是大小等于机车所能行驶的最大速度，其含义等价于“尽管不能即刻到达目标，但是沿着这个方向尽可能快地行驶，就能尽早地到达”。

不过，这仍是一个期望速度，机车已经有一个现有速度。将期望速度减去现有速度，就能得

www.TopSage.com

TopSage.com

到这样一个向量，其含义是：将这个向量增加到当前速度，会有最佳表现，机车将沿着正确的方向并以最大速度到达目标。

然后将这个向量增加到总转向力。不过要记住，在update方法中，_steeringForce会完成截断从而限制在_maxForce以内。所以尽管没有达到我们真正希望的目标，但是已经在_maxForce和_maxSpeed的约束范围内尽了最大努力，如图2-4所示。

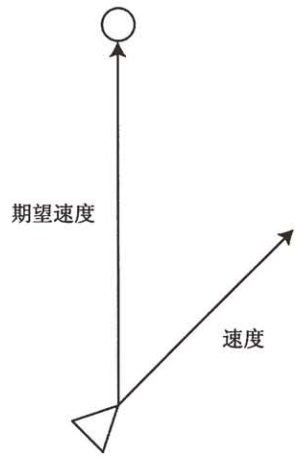


图2-3 到达当前目标的期望速度

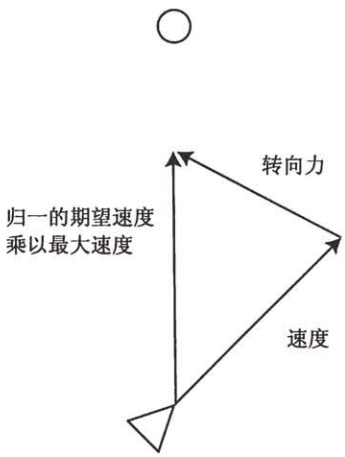


图2-4 可能的最佳期望速度，以及将当前速度改变为期望速度所需的作用力

现在来看搜寻行为实际作用的一个例子：

```
package
{
    import com.foed.SteeredVehicle;
    import com.foed.Vector2D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class SeekTest extends Sprite
    {
        private var _vehicle:SteeredVehicle;

        public function SeekTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _vehicle = new SteeredVehicle();
            addChild(_vehicle);

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

www.TopSage.com

```
        }

        private function onEnterFrame(event:Event):void
        {
            _vehicle.seek(new Vector2D(mouseX, mouseY));
            _vehicle.update();
        }
    }
}
```

这是一个相当简单的例子。它只是创建了一个新的转向机车，把它放在场景中，然后这个机车在每一帧都会搜寻鼠标并更新。试着改变机车的最大速度(maxSpeed)和最大作用力(maxForce)的值，或者甚至可以试着改变其mass，从而了解所有这些因素会如何影响转向行为。还可以让它搜寻一个固定点而不是鼠标。或者如果你想更刺激一点，可以创建另一个Vehicle，并让转向机车搜寻这个新创建的机车。这样一来onEnterFrame方法中会有如下的代码：

```
_vehicle.seek(_targetVehicle.position);
```

甚至可以试着让一个转向机车搜寻鼠标并让另一个转向机车搜寻第一个机车。如果已经充分了解所有这些工作，我们来讨论下一个行为逃避。

2.4.2　逃避行为

逃避行为与搜寻行为正相反。实际上，除了最后一行代码，二者的实现完全相同，这里的最后一行代码将从_steeringForce减去计算得到的作用力，而不是增加这个力：

```
public function flee(target:Vector2D):void
{
    var desiredVelocity:Vector2D = target.subtract(_position);
    desiredVelocity.normalize();
    desiredVelocity = desiredVelocity.multiply(_maxSpeed);
    var force:Vector2D = desiredVelocity.subtract(_velocity);
    _steeringForce = _steeringForce.subtract(force);
}
```

没有必要详细地解释这段代码，因为所有工作都与搜寻行为完全相同。最后一行代码的含义是：很好，你已经清楚了到达这个目标要怎样做。现在调转方向反其道行之。以下是逃避行为的一个简单测试（可以从本书下载网站下载FleeTest.as文件）：

```
package
{
    import com.foed.SteeredVehicle;
    import com.foed.Vector2D;
    import com.foed.Vehicle;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class FleeTest extends Sprite
    {
```

www.TopSage.com

```
        private var _vehicle:SteeredVehicle;

        public function FleeTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _vehicle = new SteeredVehicle();
            _vehicle.position = new Vector2D(200, 200);
            _vehicle.edgeBehavior = Vehicle.BOUNCE;
            addChild(_vehicle);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            _vehicle.flee(new Vector2D(mouseX, mouseY));
            _vehicle.update();
        }
    }
}
```

当然这里调用了flee而不是seek，除了这个明显的不同外，还有一个主要的区别是，用两行代码将机车定位在离场景边缘有一段距离的地方，并要求机车到达场景边缘时反弹。至于为什么要增加这两行代码，只要去除这两行就能看出原因了。如果没有这两行代码，机车会保持在角落里以避开鼠标，这样你可能甚至看不到机车。

现在已经有了两个完全相反的行为，接下来显然要针对这两种行为创建一对机车，看看它们相互之间如何作用。下面的例子就要完成这个测试，可以从下载网站下载这个SeekFleeTest1.as文件：

```
package
{
    import com.foed.SteeredVehicle;
    import com.foed.Vector2D;
    import com.foed.Vehicle;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class SeekFleeTest1 extends Sprite
    {
        private var _seeker:SteeredVehicle;
        private var _fleer:SteeredVehicle;

        public function SeekFleeTest1()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _seeker = new SteeredVehicle();
```

www.TopSage.com

大家网 TopSage.com

```
            _seeker.position = new Vector2D(200, 200);
            _seeker.edgeBehavior = Vehicle.BOUNCE;
            addChild(_seeker);

            _fleer = new SteeredVehicle();
            _fleer.position = new Vector2D(400, 300);
            _fleer.edgeBehavior = Vehicle.BOUNCE;
            addChild(_fleer);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _seeker.seek(_fleer.position);
            _fleer.flee(_seeker.position);
            _seeker.update();
            _fleer.update();
        }
    }
}
```

这里有两个机车：_seeker和_fleer。我相信不必对“搜寻者（seeker）要搜寻逃避者（fleer），逃避者要避开搜寻者”（试着把这句话快速念10遍！）再做详细解释，这一点显而易见。下面运行这个示例，看看两个机车之间如何相互作用。然后修改各个参数，看看会发生什么。

现在已经有不止一个行为，我们还可以向一个机车应用多个行为。在下面的例子中，机车A会搜寻机车B，B会搜寻C，C会搜寻A。每个机车还要逃避搜寻它的机车，这样一来这3个机车就会循环地相互追逐而形成一个圆环。这个例子见SeekFleeTest2.as文件，如图2-5所示。

```
package
{
    import com.foed.SteeredVehicle;
    import com.foed.Vector2D;
    import com.foed.Vehicle;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class SeekFleeTest2 extends Sprite
    {
        private var _vehicleA:SteeredVehicle;
        private var _vehicleB:SteeredVehicle;
        private var _vehicleC:SteeredVehicle;

        public function SeekFleeTest2()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _vehicleA = new SteeredVehicle();
            _vehicleA.position = new Vector2D(200, 200);
```

www.TopSage.com

```
            _vehicleA.edgeBehavior = Vehicle.BOUNCE;
            addChild(_vehicleA);

            _vehicleB = new SteeredVehicle();
            _vehicleB.position = new Vector2D(400, 200);
            _vehicleB.edgeBehavior = Vehicle.BOUNCE;
            addChild(_vehicleB);
            _vehicleC = new SteeredVehicle();
            _vehicleC.position = new Vector2D(300, 260);
            _vehicleC.edgeBehavior = Vehicle.BOUNCE;
            addChild(_vehicleC);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _vehicleA.seek(_vehicleB.position);
            _vehicleA.flee(_vehicleC.position);

            _vehicleB.seek(_vehicleC.position);
            _vehicleB.flee(_vehicleA.position);

            _vehicleC.seek(_vehicleA.position);
            _vehicleC.flee(_vehicleB.position);

            _vehicleA.update();
            _vehicleB.update();
            _vehicleC.update();
        }
    }
}
```

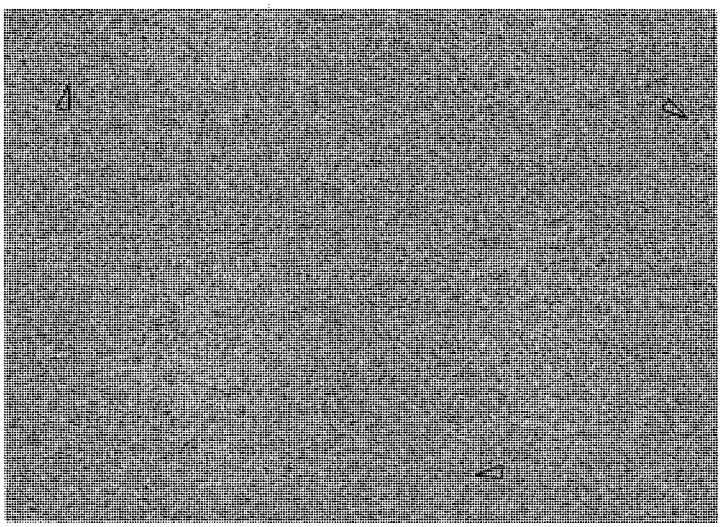


图2-5 A追B，B追C，C又追A

www.TopSage.com

同样地，可以试着修改参数，将它们综合起来看看能创建哪些情况。完成后，再来看下一个行为到达。

2.4.3　到达行为

从很多方面来看，到达行为与搜寻行为是一样的。实际上，它们通常使用相同的算法，而且往往采用完全相同的处理方式。其区别在于，采用到达模式时，如果机车到达其目标的某个特定距离，它会切换到一种精确模式，此时将缓动移向目标点，而且随着距离越来越接近，速度也会越来越慢。

为了说明为什么有些情况下到达行为很有必要，下面再来运行前面的SeekTest类。将鼠标移到机车以外的某个位置，然后让机车“抓住”它。你会看到，机车会快速地越过目标，然后意识到错误（冲过头了），再掉头反方向行驶。这个过程会无限循环延续下去，因为机车总是会保持最大速率朝着目标行进，哪怕与目标只相隔几个像素。

到达行为可以解决这个问题，当机车接近目标时，它会放慢速度：

```
public function arrive(target:Vector2D):void
{
    var desiredVelocity:Vector2D = target.subtract(_position);
    desiredVelocity.normalize();

    var dist:Number = _position.dist(target);
    if(dist > _arrivalThreshold)
    {
        desiredVelocity = desiredVelocity.multiply(_maxSpeed);
    }
    else
    {
        desiredVelocity = desiredVelocity.multiply(_maxSpeed * dist /_arrivalThreshold);
    }

    var force:Vector2D = desiredVelocity.subtract(_velocity);
    _steeringForce = _steeringForce.add(force);
}
```

前两行代码与搜寻行为方法中的代码完全相同。不过，在将期望速度乘以最大速率之前，我们会检测机车与目标的距离。如果这个距离大于某个阈值，则照旧正常相乘。退出if语句后，则会像搜寻方法中一样继续执行。

不过，如果距离小于这个阈值，则必须采取其他措施。并不是乘以_maxSpeed，而是要乘以_maxSpeed * dist/_arriveThreshold。如果距离恰好比这个阈值少一点，那么dist/_arriveThreshold就是一个非常接近于1.0的数，如0.99。因此desiredVelocity将非常接近（但略小于）_maxSpeed。不过，随着距离越来越接近0，这个比值会变得越来越小，相应地所得到的desiredVelocity也会越来越小。所以最终速度也会趋于0（假设这是作用于这个机车的唯一行为或作用力），如图2-6所示。

当然，现在SteeredVehicle类需要增加这个新的阈值属性，所以将这个属性加到类的最上面：

www.TopSage.com

TopSage.com

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

    public class SteeredVehicle extends Vehicle
    {
        private var _maxForce:Number = 1;
        private var _steeringForce:Vector2D;
        private var _arrivalThreshold:Number = 100;
        ...
    }
}
```

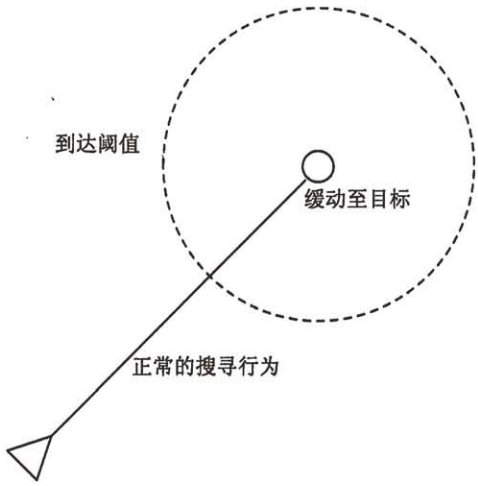


图2-6 如果在到达阈值范围以内，则缓动移向目标。否则，仍使用搜寻方法

然后为arrivalThreshold增加一个设置方法和一个获取方法，以便调整这个值：

```
public function set arriveThreshold(value:Number):void
{
    _arrivalThreshold = value;
}
public function get arriveThreshold():Number
{
    return _arrivalThreshold;
}
```

在以下测试类中可以看到这个方法的实际使用：

```
package
{
    import com.foed.SteeredVehicle;
    import com.foed.Vector2D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
```

```
    public class ArriveTest extends Sprite
    {
        private var _vehicle:SteeredVehicle;

        public function ArriveTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _vehicle = new SteeredVehicle();
            addChild(_vehicle);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _vehicle.arrive(new Vector2D(mouseX, mouseY));
            _vehicle.update();
        }
    }
}
```

这个类与SeekTest类之间唯一的不同是onEnterFrame中的一行代码，这里调用了arrive而不是seek。可以看到，如果将鼠标移开，机车的表现与搜寻时完全一样，但是如果保持鼠标不动，让机车接近鼠标，它就会放缓从而在鼠标光标处恰好停住。再次移动鼠标，它又回恢复为搜寻模式。通过调整机车的arriveThreshold属性，可以修改机车切换到到达模式之前与目标的接近程度。

如果愿意，完全可以再增加另外一些机车，与这个机车一同运行。或者现在就进入下一个行为追捕。

2.4.4　追捕行为

我要再一次重复这句话："这与搜寻行为非常相似。"确实如此，实际上，追捕行为的最后一个动作就是调用搜寻行为。追捕的关键是要预测目标将来处于什么位置。这说明目标是一个移动对象，所以它有位置和速度。因此，可以认为目标是一个Vehicle。事实上，这可以是另一个SteeredVehicle，不过由于继承关系[①]这也是一个Vehicle。

那么，如何预测目标将来的位置呢？可以取其当前速度，基于此预测将来，然后使用预测点作为搜寻的目标。但是要计算将来多久以后才合适呢？这个问题问得好！我们将称之为**前瞻时间**（look ahead time）。如果计算到将来很久以后（高前瞻时间），就会越过目标。如果时间不够久（低前瞻时间），可能仍落后于目标。实际上，搜寻行为可以定义为前瞻时间为0的追捕行为。（将来的0秒时它在哪里？就在目前的位置。）

一种策略是根据两个机车之间的距离确定前瞻时间。如果离目标很远，需要花费一段时间才能到达，所以要预测得更长一些。如果与目标已经非常接近，说明很快就能到达，所以前瞻时间

① SteeredVehicle继承了Vehicle。——译者注

www.TopSage.com

就会小得多。这正是SteeredVehicle使用的技术。下面来看这个方法：

```
public function pursue(target:Vehicle):void
{
    var lookAheadTime:Number = position.dist(target.position) / _maxSpeed;
    var predictedTarget:Vector2D =
        target.position.add(target.velocity.multiply(lookAheadTime));
    seek(predictedTarget);
}
```

首先根据两个机车之间的距离除以最大速率来计算前瞻时间。这样可以得到到达目标所需的帧数或定时器间隔数（如果目标不移动）。然后预测经过相应帧数以后目标将来的位置，可以取其当前位置再加上其速度乘以前瞻时间得到。最后，只需搜寻这个新的预测目标，参见图2-7。

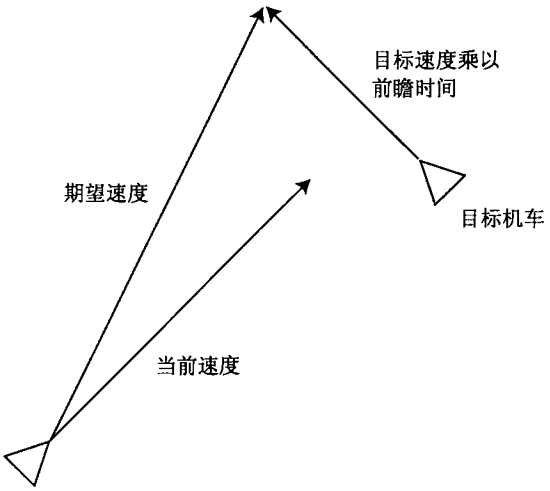


图2-7 追捕行为计算机车将来所在位置，然后搜寻该目标点

再次指出，这个方法只是得到一个大致的估计，因为一切都在运动中。不过，这起码比简单的搜寻更准确，不仅相当简单，而且速度也很快。对于追捕行为还有一些更准确（也更复杂）的算法，如果有兴趣你可以深入研究这些算法并具体实现。不过，至少可以从这个算法起步。

现在来测试一下。这一次我们将创建3个机车。一个是简单的移动Vehicle，它将作为目标。另外两个分别是SteeredVehicle，其中一个搜寻目标的位置，而另一个追捕该目标。如果一切正常，追捕者会胜出，这要归功于它采用了更先进的算法。可以从本书网站下载这个PursueTest.as文件。

```
package
{
    import com.foed.SteeredVehicle;
    import com.foed.Vector2D;
    import com.foed.Vehicle;

    import flash.display.Sprite;
```

www.TopSage.com

大家网 TopSage.com

```
import flash.display.StageAlign;
import flash.display.StageScaleMode;
import flash.events.Event;

public class PursueTest extends Sprite
{
    private var _seeker:SteeredVehicle;
    private var _pursuer:SteeredVehicle;
    private var _target:Vehicle;

    public function PursueTest()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        _seeker = new SteeredVehicle();
        _seeker.x = 400;
        addChild(_seeker);

        _pursuer = new SteeredVehicle();
        _pursuer.x = 400;
        addChild(_pursuer);

        _target = new Vehicle();
        _target.position = new Vector2D(200, 300);
        _target.velocity.length = 15;
        addChild(_target);

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onEnterFrame(event:Event):void
    {
        _seeker.seek(_target.position);
        _seeker.update();

        _pursuer.pursue(_target);
        _pursuer.update();

        _target.update();
    }
}
}
```

需要指出，两个转向机车都从相同的位置出发，不过搜寻者会向给定时刻目标的所在位置移动，而追捕者会赶到目标前方将其截获。

全面测试这个方法，可以尝试修改多个可能的参数和值来查看对结果的影响。接下来讨论躲避行为。

2.4.5　躲避行为

可能你已经猜到，躲避行为与追捕行为正相反。另外由于追捕类似于（而且使用了）搜寻行为，相应地，躲避类似于逃避，并且利用了逃避行为。

www.TopSage.com

本质上讲，躲避要预测出机车将来要到的位置，并立即远离该点。在介绍追捕行为的小节中讨论的所有原则在这里仍然适用。实际上，这与追捕其实是同一个方法，不过最后一行代码调用了flee而不是seek：

```
public function evade(target:Vehicle):void
{
    var lookAheadTime:Number = position.dist(target.position) / _maxSpeed;
    var predictedTarget:Vector2D =
     target.position.subtract(target.velocity.multiply(lookAheadTime));
    flee(predictedTarget);
}
```

关于这个行为无需多做解释，下面对pursue和evade方法做一个简单的测试，这个例子可以从本书网站下载（PursueEvadeTest.as文件）：

```
package
{
    import com.foed.SteeredVehicle;
    import com.foed.Vector2D;
    import com.foed.Vehicle;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class PursueEvadeTest extends Sprite
    {
        private var _pursuer:SteeredVehicle;
        private var _evader:SteeredVehicle;

        public function PursueEvadeTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _pursuer = new SteeredVehicle();
            _pursuer.position = new Vector2D(200, 200);
            _pursuer.edgeBehavior = Vehicle.BOUNCE;
            addChild(_pursuer);

            _evader = new SteeredVehicle();
            _evader.position = new Vector2D(400, 300);
            _evader.edgeBehavior = Vehicle.BOUNCE;
            addChild(_evader);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _pursuer.pursue(_evader);
            _evader.evade(_pursuer);
```

www.TopSage.com

```
            _pursuer.update();
            _evader.update();
        }
    }
}
```

可以看到，这个例子与SeekFleeTest1基本上是相同的测试，只不过这里使用了追捕方法（pursue）和躲避方法（evade）。由于两个机车都使用了更高级的方法，二者都没有表现出额外的优点，因此很难看出这个测试与搜寻（seek）/逃避（flee）测试之间的区别。不过，可以多做一些尝试，穿插使用某些方法[①]，看看会有什么发现。接下来，我们将从搜寻/逃避类行为转向随机性更强的漫游行为。

2.4.6 漫游行为

漫游行为可谓顾名思义。角色只是在屏幕上无目的地移动。这种行为通常用来模拟游戏或仿真中的巡查或觅食——或者只是纯粹的漫游。

听上去好像很简单，但实际上漫游行在实现上却更为复杂。可以对角色简单地应用某种随机布朗运动，不过这会让角色看起来有些抖动或颠簸。我们寻求一种更平滑更自然的运动。通常采用的方法是在角色前面放置一个假想的圆，选择该圆上的一个随机点，并使用这个点作为目标。每一帧或每一次更新时，这个随机点会以某种方式随机移动。但是由于这个目标点总是在一个假想的圆上，因此从一帧到下一帧的转向力不会有太大反差。该行为如图2-8所示。

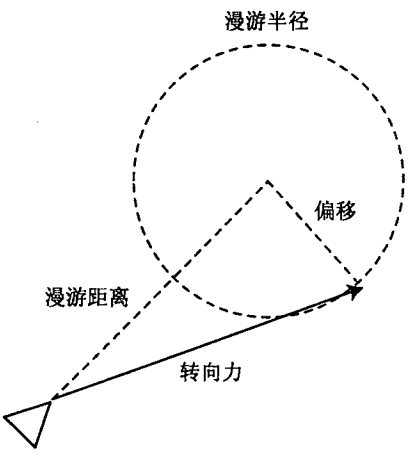


图2-8　漫游者使用距离、半径和偏移来生成一个转向力

可以改变一些因素来提供不同风格的漫游，这包括圆的大小、圆与角色的距离，以及每一帧目标点可以随机移动的幅度。

以下是漫游方法：

① 例如，一个机车使用搜寻方法，而另一个机车使用躲避方法。或者一个机车追捕，而另一个机车逃避。——译者注

www.TopSage.com

TopSage.com

```
public function wander():void
{
    var center:Vector2D =
                velocity.clone().normalize().multiply(_wanderDistance);
    var offset:Vector2D = new Vector2D(0);
    offset.length = _wanderRadius;
    offset.angle = _wanderAngle;
    _wanderAngle += Math.random() * _wanderRange - _wanderRange * .5;
    var force:Vector2D = center.add(offset);
    _steeringForce = _steeringForce.add(force);
}
```

首先要找到圆的中心。由于速度向量直接指向机车正前方，通过归一化这个向量并乘以_wanderDistance，就可以得到圆的位置（center）。接下来，再增加另一个名为offset的向量，这就是圆上的随机点。所以offset的长度（length）就是_wanderRadius，角度（angle）就是_wanderAngle。然后在_wanderRange范围内随机调整_wanderAngle。center加上offset可以得到一个向量，这就是这个行为所需的转向力。将这个转向力增加到总转向力就大功告成了。

需要指出，为此SteeredVehicle类需要增加一些新的变量。我们将在类开始处定义这些变量，并初始化为一些默认值：

```
private var _wanderAngle:Number = 0;
private var _wanderDistance:Number = 10;
private var _wanderRadius:Number = 5;
private var _wanderRange:Number = 1;
```

然后为其中大部分变量提供存取方法，从而可以改变各个机车的相应属性（_wanderAngle是完全随机的，不需要显式地设置）：

```
public function set wanderDistance(value:Number):void
{
    _wanderDistance = value;
}
public function get wanderDistance():Number
{
    return _wanderDistance;
}

public function set wanderRadius(value:Number):void
{
    _wanderRadius = value;
}
public function get wanderRadius():Number
{
    return _wanderRadius;
}

public function set wanderRange(value:Number):void
{
    _wanderRange = value;
}
public function get wanderRange():Number
```

www.TopSage.com

```
{
    return _wanderRange;
}
```

当然，最后还要看一个例子来了解这个行为的具体工作（WanderTest.as文件）：

```
package
{
    import com.foed.SteeredVehicle;
    import com.foed.Vector2D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class WanderTest extends Sprite
    {
        private var _vehicle:SteeredVehicle;

        public function WanderTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _vehicle = new SteeredVehicle();
            _vehicle.position = new Vector2D(200, 200);
            addChild(_vehicle);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _vehicle.wander();
            _vehicle.update();
        }
    }
}
```

可以尝试修改各个漫游者变量来了解它们的作用。另外不要忘记查看其他属性（例如，mass、maxSpeed和maxForce）对漫游有什么影响。可能你还想重新完成前面的搜寻或追捕测试，让搜寻者/追捕者追赶一个漫游机车。

现在越来越有趣了，随着内容的展开后面还会更有意思。我们已经介绍了比较简单的一些行为。接下来会研究更复杂的行为，先来讨论对象规避。

2.4.7　对象规避

不同于目前为止所看到的一些行为，对象规避不仅更复杂，而且每一个实现过这个行为的人的处理都不尽相同，从相对简单的方法到极其复杂的解决方案都有可能采用。由于对于这个行为并没有一个普遍公认的标准，因此我首先介绍一个非常基本的实现方式，可以把它作为起点，鼓励大家根据自己的实际需要在此基础上加以改进。

www.TopSage.com

对象规避（object avoidance）主题的完整意义是指，在机车行驶的路径中存在某个障碍物，机车必须绕开以避免发生碰撞。这看起来与碰撞检测有些关联，不过这里引入了一个预测因素更增加了复杂性，如果采用目前的速度，将来就会与这个对象发生碰撞。

当然，一旦预测到碰撞，就需要采取一些明智的行动确保这个碰撞不会发生。你可能天真地认为停下或者调头就可以了，不过要记住，可能会有多个行为在同时起作用。如果试图逃避或躲避一个食肉动物，看到路上有一棵树就停下或调头可不是一个明智的选择。理想情况下，可以采取任何必要的行动来避开障碍物而同时继续逃避或躲避。食肉动物本身很可能也需要避开这个障碍物并继续追捕你。

另外可以想象，与所躲避的对象越接近，就越有必要改变路线来避开它。如果你在沙漠上行进，看到远处有一座金字塔，只需稍加调整（方向改变一两度）就能顺利前进而不会碰上它。但是，如果一直等到金字塔就在面前时才采取行动，就需要转几乎90° 才能避开。

通过以上介绍，你应该了解了这种行为的复杂性，也能明白为什么有如此多不同的解决方案。大多数解决方案所做的第一个简化就是将障碍物看作是圆（在3D中是球体）。实际上障碍物可能并不是一个圆，但是如果把它想成是有半径和中心点的一个圆，计算会容易得多，而且一般来讲这都是可行的。要记住，通常情况下碰撞检测并不需要严格到像素级，只需知道障碍物的位置和大致大小即可，从而设法避开它。以下就是表示要避开的障碍物的Circle类：

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

    public class Circle extends Sprite
    {
        private var _radius:Number;
        private var _color:uint;

        public function Circle(radius:Number, color:uint = 0x000000)
        {
            _radius = radius;
            _color = color;
            graphics.lineStyle(0, _color);
            graphics.drawCircle(0, 0, _radius);
        }

        public function get radius():Number
        {
            return _radius;
        }

        public function get position():Vector2D
        {
            return new Vector2D(x, y);
        }
    }
}
```

这个类只是取一个半径和一个颜色，并以此半径画一个圆。对于半径和位置分别有一个获取

www.TopSage.com

方法，可以得到一个Vector2D，从而与所有基于向量的代码相一致。现在来看规避行为的实现。

由于要避开的对象往往可能有多个，avoid方法会取一个圆数组，对这个数组循环处理，查看是否需要规避。如果需要，则计算一个转向力来完成规避。这段代码相当复杂，而且对于如何完成对象规避有太多不同做法，所以这里只是提供一个示例而不做详细解释。

预测碰撞的代码主要基于David M. Bourg和Glenn Seemann所著*AI for Game Developers*中的一个示例（这本书由O'Reilly出版），不过这里做了大幅简化。转向力的计算综合汇集了多方资料和思想，另外还有我自己的一些想法。代码的解释请参见注释（由图2-9可以了解具体会发生什么）：

```
public function avoid(circles:Array):void
{
    for(var i:int = 0; i < circles.length; i++)
    {
        var circle:Circle = circles[i] as Circle;
        var heading:Vector2D = _velocity.clone().normalize();

        // vector between circle and vehicle:
        var difference:Vector2D = circle.position.subtract(_position);
        var dotProd:Number = difference.dotProd(heading);

        // if circle is in front of vehicle...
        if(dotProd > 0)
        {
            // vector to represent "feeler" arm
            var feeler:Vector2D = heading.multiply(_avoidDistance);
            // project difference vector onto feeler
            var projection:Vector2D = heading.multiply(dotProd);
            // distance from circle to feeler
            var dist:Number = projection.subtract(difference).length;

            // if feeler intersects circle (plus buffer),
            //and projection is less than feeler length,
            // we will collide, so need to steer
            if(dist < circle.radius + _avoidBuffer &&
               projection.length < feeler.length)
            {
                // calculate a force +/- 90 degrees
                // from vector to circle
                var force:Vector2D = heading.multiply(_maxSpeed);
                force.angle += difference.sign(_velocity) * Math.PI / 2;

                // scale this force by distance to circle.
                // the farther away, the smaller the force
                force = force.multiply(1.0 - projection.length / feeler.length);

                // add to steering force
                _steeringForce = _steeringForce.add(force);

                // braking force - slows vehicle down so it has
                // time to turn. The closer it is, the harder it brakes
```

www.TopSage.com

```
                _velocity = _velocity.multiply(projection.length /
                                               feeler.length);

            }
        }
    }
}
```

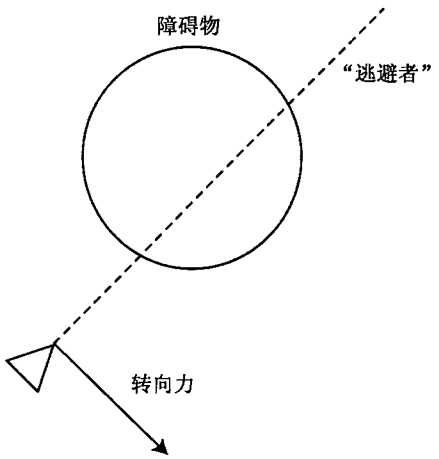


图2-9 检测与一个对象的碰撞时，会应用一个转向力

需要指出，必须为类增加两个额外的属性：

```
private var _avoidDistance:Number = 300;
private var _avoidBuffer:Number = 20;
```

规避距离（avoid distance）是指机车查找障碍物的距离范围。规避缓冲区（avoid buffer）是机车在其自身与所避开障碍物之间所要预留的空间大小。

这段代码并不完善，不过在执行上还不错，不会导致CPU性能的显著下降。可以原样使用以上代码，也可以在它的基础上加以改进，或者从头开始重新创建。以下是使用这段代码的一个例子（参见AvoidTest.as文件）：

```
package
{
    import com.foed.Circle;
    import com.foed.SteeredVehicle;
    import com.foed.Vector2D;
    import com.foed.Vehicle;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class AvoidTest extends Sprite
    {
        private var _vehicle:SteeredVehicle;
```

www.TopSage.com

```
        private var _circles:Array;
        private var _numCircles:int = 10;

        public function AvoidTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _vehicle = new SteeredVehicle();
            _vehicle.edgeBehavior = Vehicle.BOUNCE;
            addChild(_vehicle);

            _circles = new Array();
            for(var i:int = 0; i < _numCircles; i++)
            {
                var circle:Circle = new Circle(Math.random() * 50 + 50);
                circle.x = Math.random() * stage.stageWidth;
                circle.y = Math.random() * stage.stageHeight;
                addChild(circle);
                _circles.push(circle);
            }

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _vehicle.wander();
            _vehicle.avoid(_circles);
            _vehicle.update();
        }
    }
}
```

这里创建了一个圆数组，机车在周围漫游并尽量避开这些圆。尽管实现很简单，但表现很好。这个实现可以处理漫游或直行，但是如果与搜寻结合就会比较麻烦，因为机车一方面试图直接搜寻一个对象，另一方面同时还要尽力避开这个对象。这种行为可能最好利用寻路（path finding）来完成，参见第4章的介绍。接下来讨论的行为路径追随，也可以利用寻路很好地加以处理。

2.4.8　路径追随

顾名思义，**路径追随**（path following）是指机车沿着一个预定的路线行驶。尽管在地图或游戏世界中路径可以采用图形方式表示，但对于转向行为，路径则通常表示为一系列航点（waypoint）。在一个ActionScript 3.0实现中，这可以简单到只是一个Vector2D对象数组。

策略极其简单。可以取第一个航点，并以它作为目标进行搜寻。机车在该航点的一定距离范围内时，可以认为已经到达该航点，并继续移向下一个航点。如此继续，直至到达最后一个航点。还可以有一个回路，在这种情况下，到达最后一个航点时会再次从第一个航点重新开始，参见图2-10。

www.TopSage.com

TopSage.com

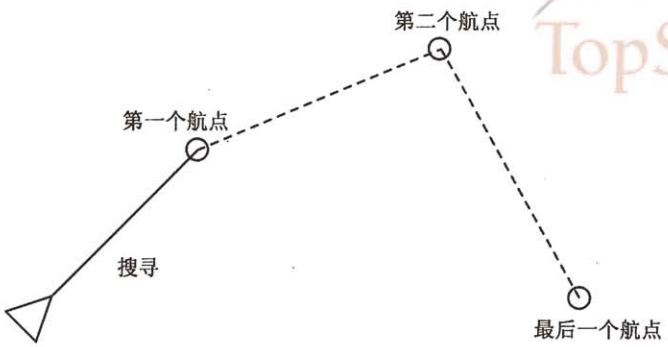


图2-10 搜寻第一个航点。足够接近时再搜寻下一个航点。依此类推

必须向这个类增加两个新属性：

```
private var _pathIndex:int = 0;
private var _pathThreshold:Number = 20;
```

下面再为这些属性提供一些存取方法：

```
public function set pathIndex(value:int):void
{
    _pathIndex = value;
}
public function get pathIndex():int
{
    return _pathIndex;
}

public function set pathThreshold(value:Number):void
{
    _pathThreshold = value;
}

public function get pathThreshold():Number
{
    return _pathThreshold;
}
```

第一个存取方法_pathIndex维护了数组中指向下一个航点的索引。_pathThreshold存取方法定义了在认为“到达”某个航点之前与该航点的接近程度。

以下是具体实现：

```
public function followPath(path:Array, loop:Boolean = false):void
{
    var wayPoint:Vector2D = path[_pathIndex];
    if(wayPoint == null) return;
    if(_position.dist(wayPoint) < _pathThreshold)
    {
        if(_pathIndex >= path.length - 1)
        {
            if(loop)
            {
```

www.TopSage.com

```
                _pathIndex = 0;
            }
        }
        else
        {
            _pathIndex++;
        }
    }
    if(_pathIndex >= path.length - 1 && !loop)
    {
        arrive(wayPoint);
    }
    else
    {
        seek(wayPoint);
    }
}
```

这里有相当多的条件。首先，从传入的路径数组取得第一个航点。如果出于某种原因这不是一个合法的Vector2D对象，则直接返回。由此允许传入一个空数组而不会导致任何错误。

如果这个点是合法的，则查看它是否在阈值距离以内。如果确实在这个距离范围内，再查看它是否是路径上的最后一个航点。如果确实是最后一个点，而且如果有回路，则循环返回到第一个航点。倘若这不是最后一个航点，则移向下一个航点。这段话写起来有些复杂，不过如果在一张纸上画出一系列航点形成一条路径，然后尝试以上逻辑，你会发现其实相当简单。

如果执行到这里，我们已经有了作为下一步目标的航点。这里可以只执行最后一行代码——seek(wayPoint)——它也能工作得很好。不过，还可以更精巧一些，允许出现一种特殊情况：如果移动到路径中的最后一个航点，而且不打算循环，则完成一个到达行为而不是搜寻。这样就会使机车完美地缓动到路径的最后一个点而不是来回反复。

这就是路径追随行为的全部内容。当然，我们还需要一个实际例子来加以说明（PathTest.as文件）：

```
package
{
    import com.foed.SteeredVehicle;
    import com.foed.Vector2D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class PathTest extends Sprite
    {
        private var _vehicle:SteeredVehicle;
        private var _path:Array;

        public function PathTest()
        {
```

www.TopSage.com

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

            _vehicle = new SteeredVehicle();
            addChild(_vehicle);

            _path = new Array();

            stage.addEventListener(MouseEvent.CLICK, onClick);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _vehicle.followPath(_path, true);
            _vehicle.update();
        }

        private function onClick(event:MouseEvent):void
        {
            graphics.lineStyle(0, 0, .25);
            if(_path.length == 0)
            {
                graphics.moveTo(mouseX, mouseY);
            }
            graphics.lineTo(mouseX, mouseY);
            graphics.drawCircle(mouseX, mouseY, 10);
            graphics.moveTo(mouseX, mouseY);
            _path.push(new Vector2D(mouseX, mouseY));
        }
    }
}
```

非常简单，相信你对此肯定没有异议。这个类的大部分内容都与绘制路径有关，以便你能具体看到路径。这里有一个_path数组，onEnterFrame方法中将机车设置为要追随这个路径。开始时，路径将是一个空数组（没有航点），所以机车不会移动。但是每次单击鼠标时，就会根据点击的位置向路径增加一个新的航点。同样地，onClick方法中的大多数代码也只是要让路径可见。

一旦开始单击鼠标，机车就会开始沿着你创建的路径行驶。由于指定loop参数为true，因此机车会沿路径循环行驶。需要指出，机车并不是机械地固守这个路径，它会自然地移动并转过可能随处出现的急转弯和小拐角。通过修改阈值，可以影响机车对拐角的处理。SteeredVehicle的其他属性（如mass、maxSpeed和maxForce）也会影响转弯行为。因此可以调整很多属性值来得到你希望的外观。

路径追随通常与寻路结合使用（参见第4章的介绍）。寻路所关心的是找到两点之间的最佳路径，它要考虑不允许角色通过的区域，可能还要考虑粗糙地形与平滑地形带来的不同影响。这通常基于网格来表示，所以结果是角色到达目标所经过的网格单元格的一个数组。这个数组可以用作路径追随转向行为中的路径，这样感觉会更自然。

下面来讨论最后一种行为——群落，这实际上是由3种子行为组成的一种复杂行为。

www.TopSage.com

大家网 TopSage.com

2.5　群落

提到群落行为，不能不提到Craig Reynolds和他的“boids”模拟。Reynolds巧妙地将一个看似不可能的复杂过程拆分为一些简单的行为。

考虑一个鸟群，它有3个主要特征。

- 首先，鸟都停留在同一个区域。如果一只鸟离鸟群太远，它会很快飞回。这称为**内聚**（cohesion），参见图2-11。
- 其次，尽管都在一起飞，但鸟会采取某种方式避免相互碰撞。它们分别有各自的空间，避免与其他鸟太过接近，也不允许其他鸟与它太接近。这称为**分离**（separation），参见图2-12。

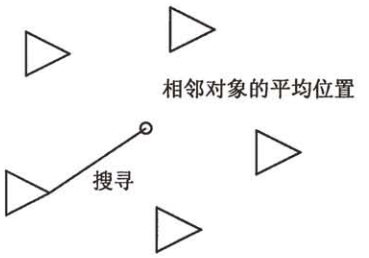


图2-11　内聚：搜寻相邻对象的平均位置

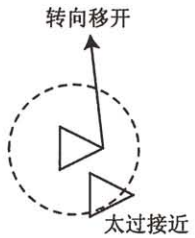


图2-12　分离：如果相邻对象过于接近，则移开

- 最后，所有鸟会朝同一个大方向移动。没错，可能各自的方向会稍有变化，但总体来讲，鸟群会向同一个方向飞行，最终每只鸟都会归入一个队列。这称为**排队**（alignment），参见图2-13。

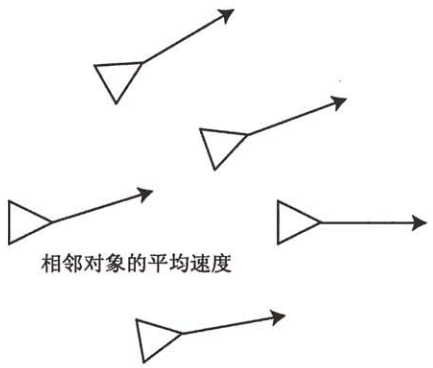


图2-13　排队：调整为相邻对象的平均速度

这3个行为——内聚、分离和排队——就构成了复杂的群落行为。

考虑一个鸟群的复杂行为时，有人可能会想象有某种智慧在从整体上控制鸟群，或者认为每只鸟都完全了解其他的每一只鸟以及整个鸟群。对于上述看法我不打算争论什么，不过必须指出，如果理解了构成群落的3个行为，你会发现每只鸟根本没有必要无所不知，而且也不需要一种控

www.TopSage.com

制鸟群的总的“群体智慧”。实际上，每只鸟只需要能够看到其邻近范围内的其他几只鸟就可以了。如果它需要转向来接近这些鸟，但是又不太过靠近，并且要按它们所朝的方向飞行，这就自然而然地出现了看似神奇的群落行为。

尽管从理论上讲群落包括3个子行为，但是这3个行为几乎总是一同使用。通常你不会希望角色只体现其中的一种或两种行为，所以这3个行为都编写在一个方法中。这种实现也更为高效，因为往往要执行大量循环，所以能够在一个循环中（而不是3个循环）完成所有工作就很有好处。下面就来看这个方法：

```
public function flock(vehicles:Array):void
{
    var averageVelocity:Vector2D = _velocity.clone();
    var averagePosition:Vector2D = new Vector2D();
    var inSightCount:int = 0;
    for(var i:int = 0; i < vehicles.length; i++)
    {
        var vehicle:Vehicle = vehicles[i] as Vehicle;
        if(vehicle != this && inSight(vehicle))
        {
            averageVelocity = averageVelocity.add(vehicle.velocity);
            averagePosition = averagePosition.add(vehicle.position);
            if(tooClose(vehicle)) flee(vehicle.position);
            inSightCount++;
        }
    }
    if(inSightCount > 0)
    {
        averageVelocity = averageVelocity.divide(inSightCount);
        averagePosition = averagePosition.divide(inSightCount);
        seek(averagePosition);
        _steeringForce.add(averageVelocity.subtract(_velocity));
    }
}
```

首先，传入一个机车数组。这里的策略是循环处理这个数组，并查看哪些机车在当前机车的视线范围内。如果一个机车在视线范围内，则将其速度和位置分别增加到各自的总量中。另外还要跟踪视线范围内的机车总数，从而能够计算平均值。还要注意，如果我们处理的机车就是当前机车，则将其忽略。

如果一个机车过于接近，则执行一个逃避行为移开，从而支持群落的分离行为。

一旦处理完整个数组，如果确实在视线范围内找到了机车，则计算平均位置和速度。然后搜寻这个平均位置，并将转向力加至平均速度。

不算太难，不过你可能会注意到这里有几个方法调用以前没有见过，即inSight和tooClose：

```
public function inSight(vehicle:Vehicle):Boolean
{
    if(_position.dist(vehicle.position) > _inSightDist) return false;
    var heading:Vector2D = _velocity.clone().normalize();
    var difference:Vector2D = vehicle.position.subtract(_position);
    var dotProd:Number = difference.dotProd(heading);
    if(dotProd < 0) return false;
```

www.TopSage.com

```
        return true;
    }
    public function tooClose(vehicle:Vehicle):Boolean
    {
        return _position.dist(vehicle.position) < _tooCloseDist;
    }
```

inSight方法确定一个机车是否能看到另一个机车。为此，它首先检查与该机车的距离是否在某个给定范围内。如果不是，则返回false。然后完成一个有趣的向量运算以确定这个机车在它前面还是后面。如果另一个机车在前面，第一个机车就可以看到该机车，如果它在后面则无法看到。这确实是一个相当严格的实现，因为它只支持180° 的视场。作为展示这已经足够了，但是如果你想对这个行为有所改进，首先要做的就是一个可变的视场。窄视场意味着角色只能看到它前方的对象，越往两边越看不到。宽视场意味着角色甚至可以看到它后面的东西。视场的这些变化会得到不同类型的群落行为。

接下来是tooClose方法，如果机车在一个指定范围内则返回true，否则返回false。

可以看到这两个方法都依赖于另外一对类变量。所以下面增加这两个类变量以及相应的存取方法：

```
    private var _inSightDist:Number = 200;
    private var _tooCloseDist:Number = 60;

    public function set inSightDist(value:Number):void
    {
        _inSightDist = value;
    }
    public function get inSightDist():Number
    {
        return _inSightDist;
    }

    public function set tooCloseDist(value:Number):void
    {
        _tooCloseDist = value;
    }
    public function get tooCloseDist():Number
    {
        return _tooCloseDist;
    }
```

最后，需要一个类来进行测试（可以在FlockTest.as文件中找到）：

```
    package
    {
        import com.foed.SteeredVehicle;
        import com.foed.Vector2D;
        import com.foed.Vehicle;
        import flash.display.Sprite;
        import flash.display.StageAlign;
        import flash.display.StageScaleMode;
        import flash.events.Event;

        public class FlockTest extends Sprite
```

www.TopSage.com

TopSage.com


```
    {
        private var _vehicles:Array;
        private var _numVehicles:int = 30;

        public function FlockTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _vehicles = new Array();
            for(var i:int = 0; i < _numVehicles; i++)
            {
                var vehicle:SteeredVehicle = new SteeredVehicle();
                vehicle.position = new Vector2D(Math.random() * stage.stageWidth,
                                                Math.random() * stage.stageHeight);
                vehicle.velocity =
                            new Vector2D(Math.random() * 20 - 10, Math.random() * 20 - 10);
                vehicle.edgeBehavior = Vehicle.BOUNCE;
                _vehicles.push(vehicle);
                addChild(vehicle);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            for(var i:int = 0; i < _numVehicles; i++)
            {
                _vehicles[i].flock(_vehicles);
                _vehicles[i].update();
            }
        }
    }
}
```

这里只是建立了一组转向机车，将其分布在场景上，并添加到一个数组中。每一帧都循环处理这个数组，对每个机车运行群落行为，然后完成更新。你会看到，尽管每个机车都在独自移动，但它很了解周围的机车，这就逐渐形成了一种群组运动效果。可以尝试修改这些变量以及机车数目等，从而对其工作有更充分的认识。

另一个有待改进的方面是数组处理的优化。每个机车目前都要检查包含所有其他机车的整个数组，这样效率可能相当低下。你可能希望重构为使用一个双重循环或者使用一种基于网格的方法（如第1章所述）。不过，这可能需要将某些群落代码移到SteeredVehicle类以外，可能移到主文档类中，或者需要移到另外某个群落管理类中。这些工作留给你来完成。

2.6 小结

本章介绍了转向行为的基本内容，包括简单的搜寻和逃避行为，以及更为复杂的障碍物规避和群落。结合这些行为来产生更复杂的运动效果时会非常有趣。再次说明，本章只是这个主题的

www.TopSage.com

一个简要介绍。还可以在网上以及其他有关转向的书中找到大量相关材料，这些内容通常都放在人工智能标题之下。希望你至少能有所启发，开始研究有关内容并用于你自己的ActionScript 3.0应用和游戏中。

第3章将讨论等角世界和寻路。将其与转向行为结合（可能还可以结合某种高级碰撞检测），你就能开发出一流的游戏。

www.TopSage.com

第3章 等角投影

等角投影（isometric projection）是一种广泛运用于计算机游戏的技术，出现在80年代早期。这是模拟三维空间的一种快速而高效的方法，能让人产生一种有深度的立体错觉，同时无需“真实3D”中必须完成的大量代价高昂的透视计算。在早期，大多数视频游戏都是自顶向下或者单向滚动的。Zaxxon（参见图3-1）和Qbert（参见图3-2）是第一批采用等角技术的商业游戏。

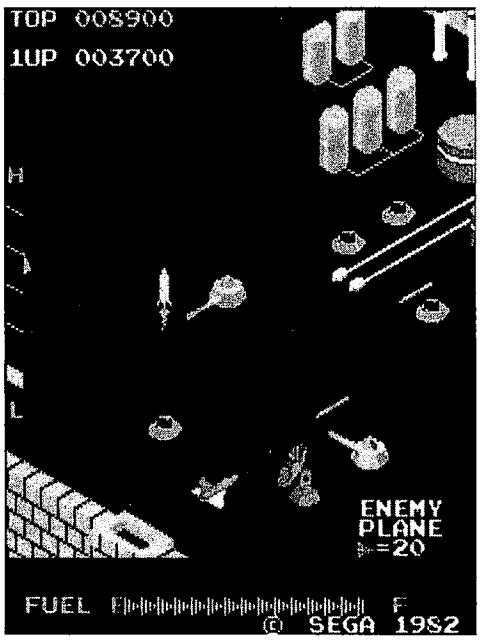


图3-1 Zaxxon

如今，尽管Halo等第一人称射击游戏中采用了更前沿的3D技术，但是等角游戏仍很流行，特别是RPG（角色扮演类游戏）或SLG（策略游戏）。

要理解等角投影，首先来看什么是投影，以及等角投影与其他可视化3D方法的区别。

投影（projection）是指一个三维物体或场景在二维平面上的表示，这个二维平面可以是一张

www.TopSage.com

大家网 TopSage.com

纸也可以是计算机屏幕。照相机使用镜头创建前方物体的投影，并投影到一个胶片或电子传感器上。甚至人的眼睛也在做同样的工作，它会将图像投影到视网膜上。

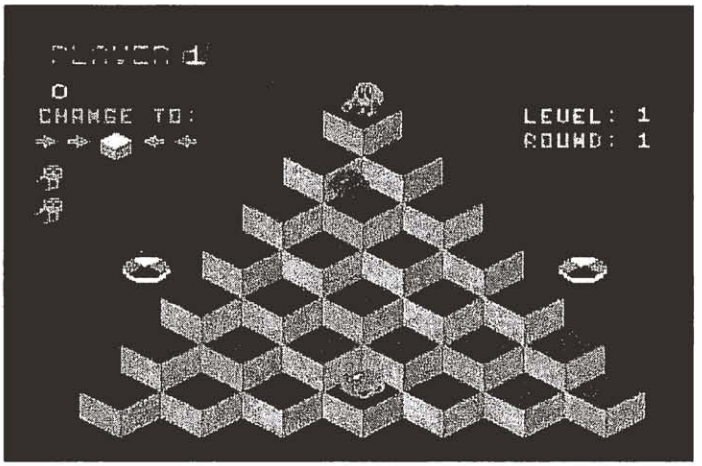


图3-2　Qbert

当在纸张上和计算机屏幕上发生移动时，会存在多种不同类型的投影。我们最熟悉的一类投影可能是**透视投影**（perspective projection）。在这种投影中，离视点越远的物体显示得越小，直至接近于一个行将消失的点。这正是《Flash ActionScript 3.0 动画教程》3D相关章节中描述的投影类型，Papervision3D等3D框架和其他3D软件中都默认使用了这种类型的投影，相机也采用这种投影来呈现图像，而且这也能最好地模拟如何用我们自己的眼睛感知事物。

另一方面，**等角投影**（isometric projection）则是一种**轴测投影**（axonometric projection）。在这种投影中，物体的大小并不随物体与相机距离的改变而发生变化。

等角（isometric）一词的含义是“等量”。这是指这种投影中x、y和z轴之间的角度相同，都为120°，如图3-3所示。

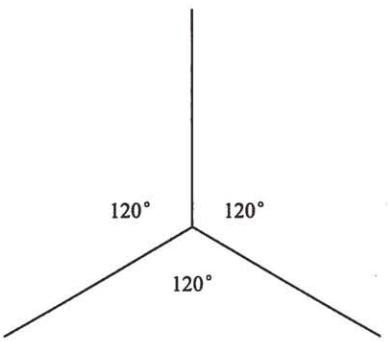


图3-3　可以看作是房间一角，在此两面墙与地面相互接触。在等角投影中，各轴之间的夹角为120°

游戏中的等角世界几乎总是基于区块（tile）的世界，所以会有一个由单个区块组成的地面，

www.TopSage.com

而不是一个连续无缝的环境。这个世界中的对象就是区块本身以及置于区块之上的物体。在大多数情况下，区块可以重用多次，所以可以多次使用一个区块来创建一个完整的地面（参见图3-4），或者可以用少量区块创建变化很大的环境（如在Zaxxon中，参见图3-1）。

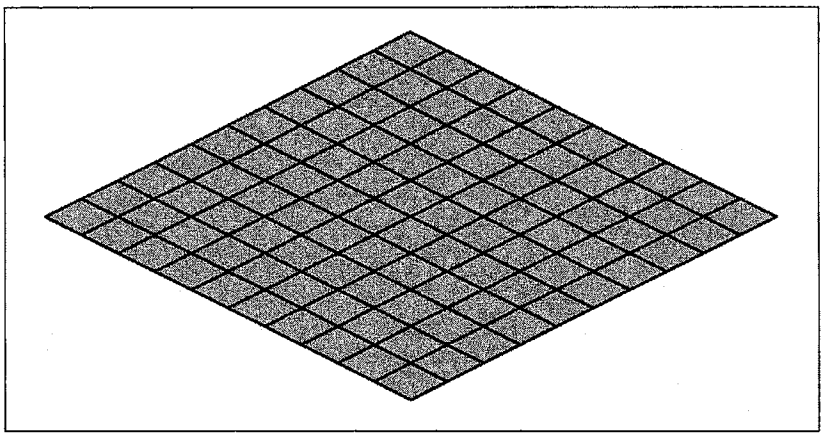


图3-4 一个区块用于创建整个地面

3.1 等角投影与二等角投影

下面告诉你一个小秘密，如果你在鸡尾酒会上与别人交谈时能提到这个小秘密，也许会让你备受瞩目（不过也可能只是让人生厌）：几乎所有声称为等角的游戏、引擎、艺术作品等实际上都不是等角的，它们只是**二等角**（dimetric），这意味着3个轴之间存在两种不同的角度（参见图3-5）。注意，二等角只表示“两种度量”。还有第三种轴测投影，称为**三度投影**（trimetric），其中每个轴都有一个不同的角度。二等角投影并不一定必须使用此例中显示的角度，不过这是计算机游戏、等角作品等中最常用的角度。

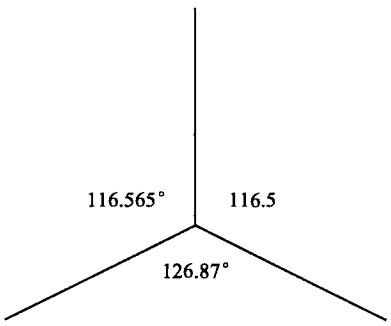


图3-5 二等角投影使用两个角度

现在要使用这些看起来有些奇怪的角度而不是漂亮的120°。不过，处理像素时确实需要使用这些角度。下面分别采用真正的等角投影以及二等角投影来呈现一个区块（见图3-6）。

www.TopSage.com

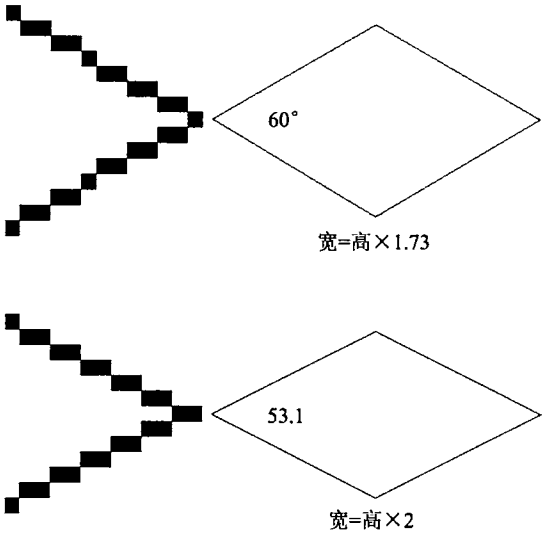


图3-6　真正的等角投影以及二等角投影中的区块

上面的区块采用等角投影呈现，角度标为60°。区块宽与高的比例最后为1.73:1。下面的区块使用了二等角投影。尽管角度不是一个整数，但是宽与高的比例正好是2:1。这样一来，为这种系统创建图片就要容易得多。例如，不必为区块创建173×100像素的图片，而是创建200×100的区块。区块的定位同样会更为简单。

如果认为这还不算重要，还可以看到二等角区块看起来会好得多。右边可以看到标准大小的区块——采用二等角投影的区块线条很平滑，而等角区块看上去有些锯齿。在左边的放大版中，可以看到为什么会出现这种情况。在等角投影中，每垂直移动1个像素，就需要水平移动1.73个像素。由于无法移动一个像素的某一部分，因此有时要移动两个像素，而有时只移动一个像素。这也正是看起来有锯齿的原因。不过，在采用二等角投影的版本中，每次都会向上或向下移动一个像素，同时水平移动两个像素，准确而平滑。

由于这样更易于编程和设计，另外也由于所有人都采用了这种做法，因此我们也将使用二等角系统。既然已经对相关概念有了全面了解，下面将不再严格区分等角和二等角，尽管我们知道这并不是真正的等角，但仍会继续这样称呼（因为所有人都是如此）。

3.2　创建等角图形

尽管不是编写具体的代码，不过了解如何为等角系统创建图形会很有用。（实际上我所说的是比例为2:1并使用26.565°的二等角系统，不过这是最后一次澄清，后面将不再做特别说明。）下面将具体建立一个区块。

首先，打开Flash、Fireworks、PhotoShop或想用来创建图形的任何程序。创建一个正方形，这必须是四边完全相等的真正的正方形，而不是近似正方形的矩形。正方形大小为100×100。

现在旋转45°。再次申明，要使用一个允许正方形精确旋转45°的工具，而不能是大致

www.TopSage.com

TopSage.com

45°。很多工具都可以让用户按下shift键来绘制正方形并旋转45°和90°。现在就得到了一个菱形。

最后，垂直方向缩减为50%，而保持水平方向仍不改变。

这样就能得到图3-6下半部分所示的形状。这个形状现在大致为141.4×70.7，即2:1的比例。Fireworks会四舍五入为最接近的像素，所以将得到141×71，这对我们来说已经足够用了。

现在，如果你去告诉设计人员必须让所有图形都采用某种怪异的尺寸，如141.4×70.7，他们可能会对你很有意见。所以接下来选择对象，并将其高度修改为100，再将宽度改为200（如果它没有自动修改）。现在形状就是200×100，这会让你的设计人员很满意。当然，也可以使用更小或更大的区块大小，只要保持2:1的比例不变即可。在实际运用中，你可能会建立比这更小的区块，从而支持更细致、更有效的碰撞检测（本章后面将介绍有关内容）。

现在可以在这个菱形中填入你想要的任何图形：绿草、岩石、水流、泥土、树木、石块等。理想情况下这些图形会平铺，从而可以无缝地放在一起。（正是这个原因称之为区块！）如果想让区块有立体感，只需将所有区块上移一点，并从4个角向下画几条线，再把它们连接起来，如图3-7所示。

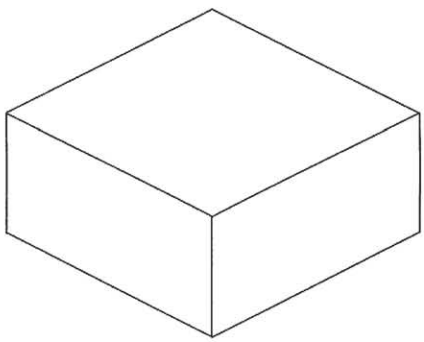


图3-7 一个等角的盒子

可以在网上找到各种有关如何创建等角图形的优秀教程——它们会介绍如何着色、加阴影、实现平滑平铺，等等。只需搜索“isometric art tutorials”（等角艺术教程）或“isometric pixel art”（等角像素艺术）就可以查到。本章对此将不再深入讲解，接下来转向编程工作。

3.3 等角变换

在等角世界的创建中，最重要（可能也是最难理解、最容易混淆，而且问到最多）的一个主题就是如何将等角世界中的x, y, z坐标转换为屏幕上的x, y, z坐标。或者反之，如何将屏幕上的x, y, z坐标转换为等角世界中的x, y, z坐标。

我见过至少五六种不同的方法，每种方法都大相径庭。有些方法可能很精确，但是效率却不高。还有一些方法速度很快，能很好地使用，但是却未能真正反映等角视图涉及的具体3D变换。还有几种方法……我实在不知道这些方法有何原理，也不清楚作者因何提出这些思想，也许这些

www.TopSage.com

方法确实能奏效，不过我无法理解它们所采用的逻辑。

3.3.1　世界坐标变换为屏幕坐标

首先来看等角视图中查看对象时完成的具体3D变换，这能帮助你理解准确计算坐标的方法以及简便方法的化简形式。这里创建了一个SWF文件来展示坐标变换，文件名为IsoProjection.swf（见本书下载页面提供的文件）。

打开这个SWF时，会看到通过俯视图、正视图和侧视图呈现的一个立方体，如图3-8所示。

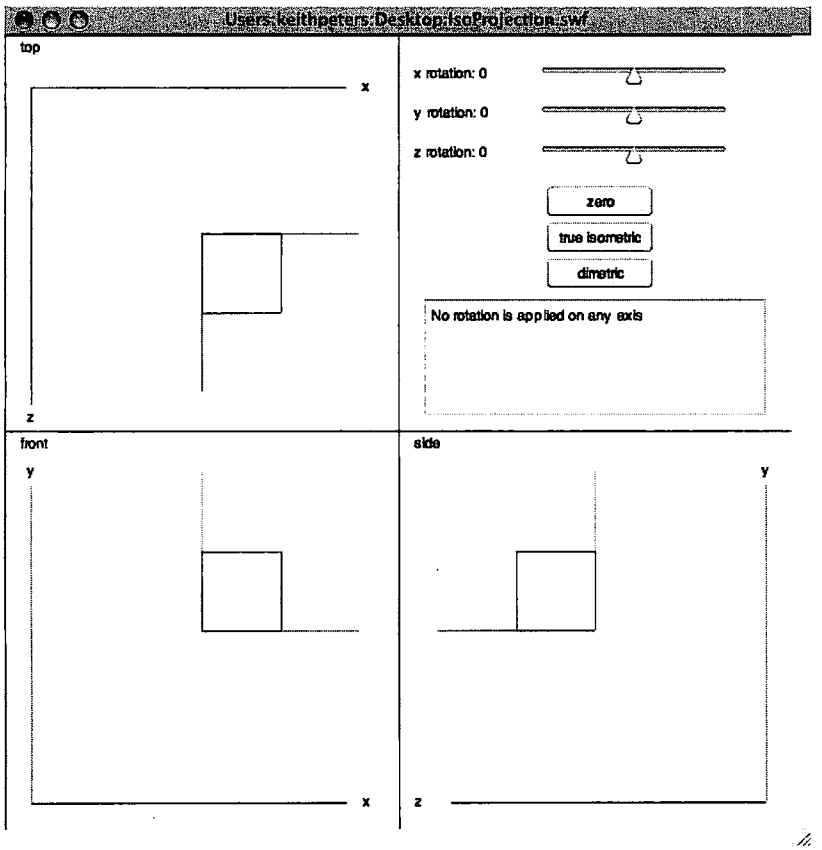


图3-8　未旋转立方体的三视图

这个SWF还显示了我们使用的坐标系统，它与大多数基于Flash的3D系统中使用的3D坐标系统相同：*x*轴从左到右，*y*轴从上到下，*z*轴从里到外“进入或伸出”屏幕（用来表示深度）。

第一次变换是在*y*轴上旋转45°，可以利用*y*轴旋转滑块来做到。然后再按键盘的左右箭头键使它准确地旋转45°，这会得到图3-9所示的图形。

第二次变换将在*x*轴上再旋转–30°。同样地，可以使用*x*轴旋转滑块和键盘来做到，结果如图3-10所示。

www.TopSage.com

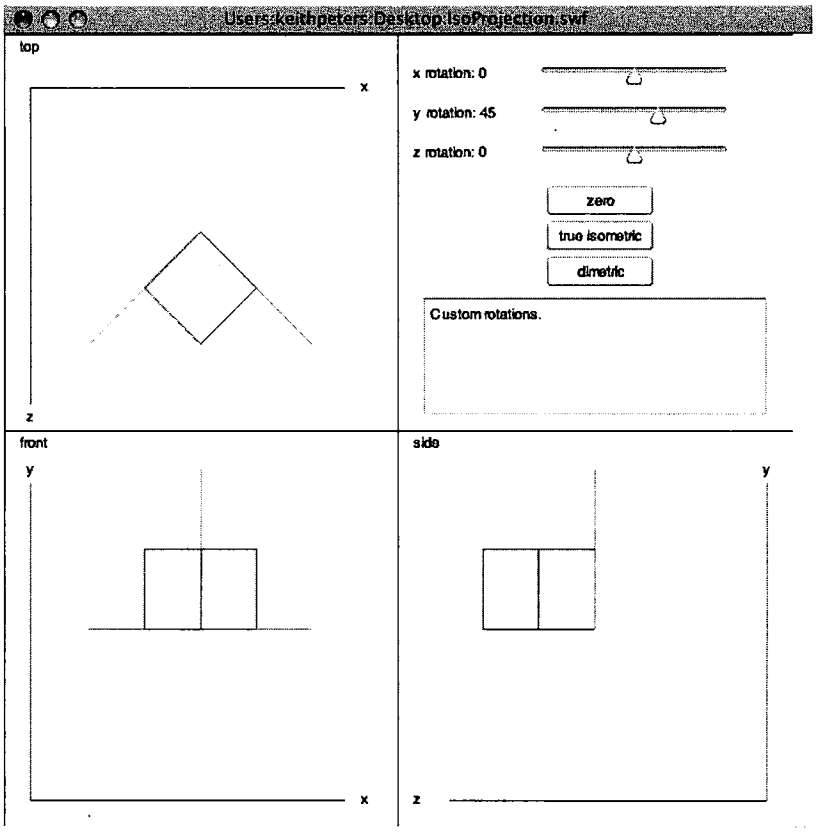


图3-9 第一次旋转：*y*轴上旋转45°

现在可以看到，立方体的顶面和底面正是如图3-6下半部分所显示的形状。这些形状的宽度是其高度的两倍，并且有前面所述的平滑斜边（比例为2:1）。

单击dimetric按钮也可以产生同样的变换，另外可以通过单击“true isometric”按钮查看“真正的等角投影”。

创建这些变换的伪代码如下所示：

```
sX = x * cos(45) - z * sin(45);
z1 = z * cos(45) + x * sin(45);
sY = y *  cos(-30) - z1 * sin(-30);
z2 = z1 * cos(-30) + y *  sin(-30);
```

在此取3D空间中一个点的*x*, *y*, *z*世界坐标，并计算sX和sY，即可以在屏幕上绘制的一个点。这表示将点沿*y*轴旋转45°，再沿*x*轴旋转–30°（参见《Flash ActionScript 3.0 动画教程》中15.7节）。实际上，这里并不需要倒数第二行的代码（即计算z2），但我们还是包含了这行代码来展示完整的坐标旋转。不过，在后面完成深度排序时确实需要这行代码。

可以编写这样一个方法并安心使用，因为你很清楚这里遵循了数学、几何和三角学的所有规

大家网 TopSage.com

则，尽管如此，从实用的角度来看这里还存在着几个问题。首先是所有调用的函数都是三角函数。实际上，对此可以做一些优化，以便从长远来看三角学方面不再成问题。

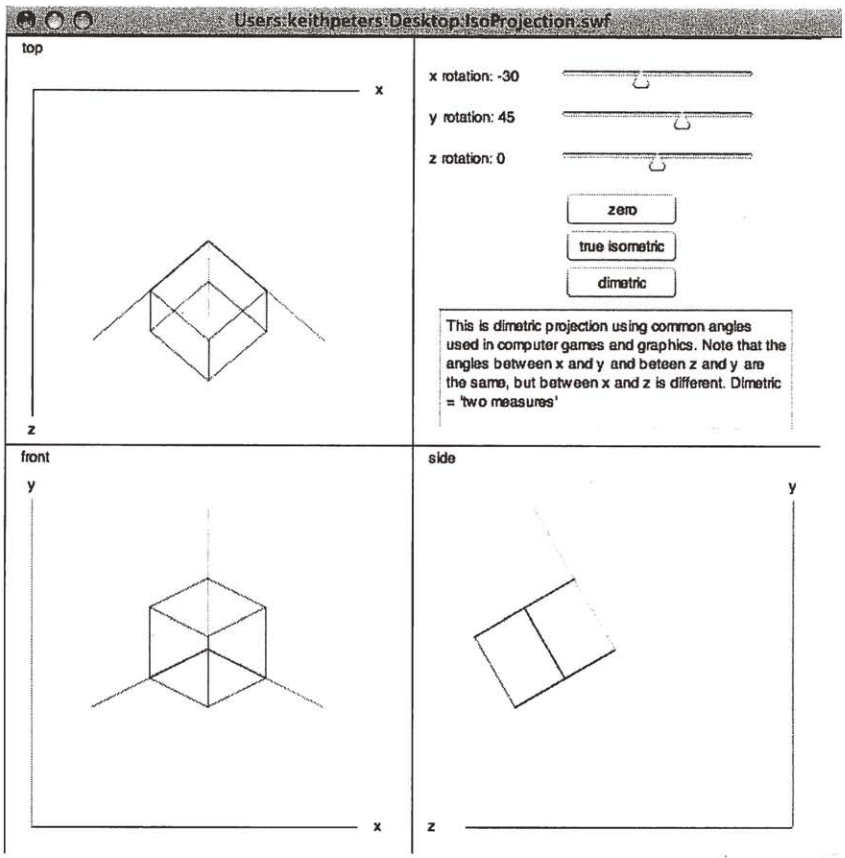


图3-10　第二次旋转：*x*轴上旋转-30°

不过，还有一个更严重的问题：这个方法使用了一些怪异的区块尺寸。还记得3.2节介绍图形创建时建立的100×100的正方形吗？在我们旋转并缩放时，那个正方形是怎样变成141.4×70.7的？我们在代码中就是要采用同样的做法。所以，使用前面所示的公式旋转一个100×100的正方形的4个顶点，最后它们会构成一个141.4×70.7的菱形。到目前为止还一切正常。

不过应该记得，接下来我们要将这个菱形区块大小调整为200×100。这里宽度和高度的缩放系数约为1.414，正好是2的平方根。这并不是一个奇特的巧合，而是因为45° 角的各种三角关系。如果你对此感兴趣，可以自行推导。如果要像这样缩放图形，也需要使用代码以相同的步骤执行缩放，否则代码得出的对象将与图形不一致。技巧就是这里。

将等角变换中的坐标乘以2的平方根后，继续化简，可以得到以下结果：

```
sx = x - z;
sy = y * 1.2247 + (x + z) * .5;
```

www.TopSage.com

如果你有兴趣了解这个化简是如何做到的，下面将详细说明这一推导过程。不过相信我，你完全可以跳过这个内容直接看下一节。

先来介绍坐标旋转公式：

```
x1 = x * cos(45) - z * sin(45)
z1 = z * cos(45) + x * sin(45)
y1 = y  * cos(-30) - z1 * sin(30)
z2 = z1 * cos(-30) + y  * sin(30)
```

将三角函数转化为它的近似值：

```
x1 = x * .707 - z * .707
z1 = z * .707 + x * .707
y1 = y  * .866 - z1 * -.5
z2 = z1 * .866 + y  * -.5
```

化简：

```
x1 = (x - z) * .707
z1 = (x + z) * .707
y1 = y  * .866 - z1 * -.5
z2 = z1 * .866 + y  * -.5
```

将z1代入到y1和z2的公式中以消去z1：

```
x1 = (x - z) * .707
y1 = y  * .866 - ((x + z) * .707) * -.5
z2 = ((x + z) * .707) * .866 + y  * -.5
```

将x1, y1，z2分别乘以2的平方根（1.414）。由于0.707 * 1.414 = 1，这就能大大化简方程：

```
x1 = x - z
y1 = y * 1.2247 + (x + z) * .5
z2 = (x + z) * .866 - y  * .707
```

讨论深度排序时还会再次使用这个z2计算。

不论能否理解上述推导过程，起码现在这个函数比先前的函数要简单得多。你要记住的是，等角世界中的100×100正方形最终在2D平面渲染出来就是一个200×100的菱形区块。当然，并不要求必须使用200×100这个数字。如果开始时采用的是50×50的正方形，则会得到一个100×50的菱形。简单地讲，菱形的高度就等于区块所表示的正方形的边长。非常简单！

很多算法会去掉1.2247，直接采用以下计算：

```
x1 = x - z
y1 = y + (x + z) * .5
```

如果场景中所有对象的高度都相同，这样做完全可以。在简单的系统中，所有区块的y都等于0。在这种情况下，公式还可以进一步化简，如下：

```
x1 = x - z
y1 = (x + z) * .5
```

然而，对于对象不在同一高度的情况，为了在屏幕上正确地放置对象，1.2247则是不可去掉的。如果把它去掉，这样一来，所有置为非0高度的对象会有不正确的高度，这将给整个屏幕造成一种被挤压的感觉。

www.TopSage.com

3.3.2　屏幕坐标转换为世界坐标

现在我们还需要一种与之正好相反的方法：将屏幕坐标转换为等角3D坐标。

这里有一个小问题，因为屏幕坐标只表示两维，而我们需要三维。大多数情况下，这种转换用于将一个鼠标点击位置转换为地面或场地上的某个位置。因此，我们真正想要的是将x、y屏幕坐标转换为x、z世界坐标，而保持y（高度）为0。因此，如果使用等角坐标到屏幕坐标的转换算法，可以得到：

```
sx = x - z;
sy = y * 1.2247 + (x + z) * .5;
```

将y设置为0，求解x和z，并进行化简——就可以得到以下伪代码：

```
x = sy + sx / 2
y = 0
z = sy - sx / 2
```

这是基本的代数问题，可不是我自己能想出来的。可以从屏幕坐标sx和sy还原出一个x, y, z坐标，表示世界空间中的一个3D点。

3.3.3　IsoUtils类

之前已经介绍了相当多的理论，也提供了足够多的伪代码。下面来看一些实际的代码，了解具体如何使用。首先，我们需要一个结构来保存一个3D坐标点。与本书中的所有代码一样，可以从本书下载网页（www.friendsofed.com）得到这个结构（Point3D.as文件）。

```
package com.friendsofed.isometric
{
    public class Point3D
    {
        public var x:Number;
        public var y:Number;
        public var z:Number;

        public function Point3D(x:Number = 0, y:Number = 0, z:Number = 0)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }
    }
}
```

下面是IsoUtils类，可以下载IsoUtls.as文件得到这个类。前面已经指出，这个类会处理世界坐标和屏幕坐标之间的转换：

```
package com.friendsofed.isometric
{
    import flash.geom.Point;

    public class IsoUtils
```

www.TopSage.com

TopSage.com

```
    {
        // a more accurate version of 1.2247...
        public static const Y_CORRECT:Number = Math.cos(-Math.PI / 6) * Math.SQRT2;

        /**
         * Converts a 3D point in iso space to a 2D screen position.
         * @arg pos the 3D point.
         */
        public static function isoToScreen(pos:Point3D):Point
        {
            var screenX:Number = pos.x - pos.z;
            var screenY:Number = pos.y * Y_CORRECT + (pos.x + pos.z) * .5;
            return new Point(screenX, screenY);
        }

        /**
         * Converts a 2D screen position to a 3D point in iso space,
         * assuming y = 0.
         * @arg point the 2D point.
         */
        public static function screenToIso(point:Point):Point3D
        {
            var xpos:Number = point.y + point.x * .5;
            var ypos:Number = 0;
            var zpos:Number = point.y - point.x * .5;
            return new Point3D(xpos, ypos, zpos);
        }
    }
}
```

注意，这里将魔法数字1.2247创建为一个静态常量，并利用三角函数来计算。这样一来，它将作为一个精确值而不是四舍五入的结果，而且只会计算一次。准确又高效！

以下是使用这些方法的大致思路：

```
package
{
    import com.friendsofed.isometric.IsoUtils;
    import com.friendsofed.isometric.Point3D;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.geom.Point;

    public class IsoTransformTest extends Sprite
    {
        public function IsoTransformTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var p0:Point3D = new Point3D(0, 0, 0);
            var p1:Point3D = new Point3D(100, 0, 0);
            var p2:Point3D = new Point3D(100, 0, 100);
            var p3:Point3D = new Point3D(0, 0, 100);
```

www.TopSage.com

```
            var sp0:Point = IsoUtils.isoToScreen(p0);
            var sp1:Point = IsoUtils.isoToScreen(p1);
            var sp2:Point = IsoUtils.isoToScreen(p2);
            var sp3:Point = IsoUtils.isoToScreen(p3);

            var tile:Sprite = new Sprite();
            tile.x = 200;
            tile.y = 200;
            addChild(tile);

            tile.graphics.lineStyle(0);
            tile.graphics.moveTo(sp0.x, sp0.y);
            tile.graphics.lineTo(sp1.x, sp1.y);
            tile.graphics.lineTo(sp2.x, sp2.y);
            tile.graphics.lineTo(sp3.x, sp3.y);
            tile.graphics.lineTo(sp0.x, sp0.y);
        }
    }
}
```

可以在IsoTransformTest.as文件中找到这个类。它首先创建4个Point3D实例，在*x*-*z*平面上构成一个正方形。然后使用IsoUtils.isoToScreen将它们转换为4个二维的点，这4个点将被绘制在屏幕上（需要在*x*轴和*y*轴上偏移200，以避免“堵”在角落里）。

运行这个类会看到我们熟悉的一个漂亮的菱形。如果在类的最后增加一行代码：trace(width, height)，你会看到这个菱形的大小恰好是所期望的200×100。

当然，只是绘制3D点和单个线条是无法创建一个丰富的等角世界的。这些工具方法主要用于完成等角对象在世界中的定位。下一步将创建一个类来表示等角对象。

3.4　等角对象

一般来讲，构成这个世界的对象要由图形设计师利用一个图形程序（如Fireworks、Photoshop，甚至是Flash本身）来创建（或者由你自己亲自承担起图形设计师的角色，利用这样一个图形程序创建）。这些对象由多个用草木、树、泥土、水、建筑等装饰的区块组成，或者还包括多个角色，甚至可能有动画角色。要由你将这些图形运用于3D世界中（如果完全由你一人完成开发，现在你就要回归程序员的身份），准确地定位，并移动其中一些对象。这就得出了一个概念，即需要在等角空间中放置和移动图形对象。听起来很适合实现为一个类。以下给出IsoObject类，可以下载IsoObject.as文件得到这个类：

```
package com.friendsofed.isometric
{
    import flash.display.Sprite;
    import flash.geom.Point;
    import flash.geom.Rectangle;

    public class IsoObject extends Sprite
    {
        protected var _position:Point3D;
        protected var _size:Number;
```

www.TopSage.com

```
protected var _walkable:Boolean = false;

// a more accurate version of 1.2247...
public static const Y_CORRECT:Number = Math.cos(-Math.PI / 6) * Math.SQRT2;

public function IsoObject(size:Number)
{
    _size = size;
    _position = new Point3D();
    updateScreenPosition();
}

/**
 * Converts current 3d position to a screen position
 * and places this display object at that position.
 */
protected function updateScreenPosition():void
{
    var screenPos:Point = IsoUtils.isoToScreen(_position);
    super.x = screenPos.x;
    super.y = screenPos.y;
}

/**
 * String representation of this object.
 */
override public function toString():String
{
    return "[IsoObject (x:" + _position.x + ",
            y:" + _position.y + ", z:" + _position.z + ")]";
}

/**
 * Sets / gets the x position in 3D space.
 */
override public function set x(value:Number):void
{
    _position.x = value;
    updateScreenPosition();
}
override public function get x():Number
{
    return _position.x;
}

/**
 * Sets / gets the y position in 3D space.
 */
override public function set y(value:Number):void
{
    _position.y = value;
    updateScreenPosition();
}
override public function get y():Number
```

www.TopSage.com

大家网 TopSage.com

```
{
    return _position.y;
}

/**
 * Sets / gets the z position in 3D space.
 */
public function set z(value:Number):void
{
    _position.z = value;
    updateScreenPosition();
}
public function get z():Number
{
    return _position.z;
}

/**
 * Sets / gets the position in 3D space as a Point3D.
 */
public function set position(value:Point3D):void
{
    _position = value;
    updateScreenPosition();
}
public function get position():Point3D
{
    return _position;
}

/**
 * Returns the transformed 3D depth of this object.
 */
public function get depth():Number
{
    return (_position.x + _position.z) * .866 - _position.y * .707;
}

/**
 * Indicates whether the space occupied by this object
 * can be occupied by another object.
 */
public function set walkable(value:Boolean):void
{
    _walkable = value;
}
public function get walkable():Boolean
{
    return _walkable;
}

/**
 * Returns the size of this object.
```

www.TopSage.com

```
        */
        public function get size():Number
        {
            return _size;
        }

        /**
         * Returns the square area on the x-z plane
         * that this object takes up.
         */
        public function get rect():Rectangle
        {
            return new Rectangle(x - size / 2, z - size / 2, size, size);
        }

    }
}
```

这个类主要用于获取和设置对象的3D位置，并且根据这个3D位置计算其屏幕位置。由于这个类扩展了Sprite，因此可以通过对super.x和super.y赋值设置屏幕位置。（必须有super，因为x和y的获取方法和设置方法已经被覆盖为获取和设置3D位置，而不是屏幕位置。）注意，update-ScreenPosition方法使用了前面的IsoUtils.isoToScreen方法。

如果要广泛使用这个类，以下建议可能对你很有帮助。按照以上代码，要设置x、y和z，就要调用3次updateScreenPosition，这样做效率很低。通常可以按照如下方式进行处理：建立一个失效方法，将对象标志为需要更新，并设置一个enterFrame监听器，一旦进入下一帧就完成更新。这样一来，可以将x、y和z连续更新100次，而只是在完成时才计算1次屏幕位置。

其他方法（如depth、walkable和rect的获取方法和设置方法）主要用来处理深度排序和碰撞检测，这些内容将在本章后面再做介绍。

需要指出，构造函数接受一个参数size。所有IsoObject都有一个size属性来确定其覆盖区域（footprint），即它们在*x*-*z*平面上所占据的空间。这个覆盖区域通常认为是一个正方形，在一个基于区块的世界中这样考虑是合理的。在3.5节讨论区块绘制以及在后面谈到碰撞检测阶段时，你会对此有更多了解。

现在这个IsoObject类中还不包含图形。虽然可以手动地绘制一些图形，不过下面还是为此创建一个新类，或者一个已绘制图形的等角区块，即DrawnIsoTile类（见DrawnIsoTile.as文件）：

```
package com.friendsofed.isometric
{
    public class DrawnIsoTile extends IsoObject
    {
        protected var _height:Number;

        protected var _color:uint;

        public function DrawnIsoTile(size:Number, color:uint, height:Number = 0)
        {
```

www.TopSage.com

```
            super(size);
            _color = color;
            _height = height;
            draw();
        }

        /**
         * Draws the tile.
         */
        protected function draw():void
        {
            graphics.clear();
            graphics.beginFill(_color);
            graphics.lineStyle(0, 0, .5);
            graphics.moveTo(-size, 0);
            graphics.lineTo(0, -size * .5);
            graphics.lineTo(size, 0);
            graphics.lineTo(0, size * .5);
            graphics.lineTo(-size, 0);
        }

        /**
         * Sets / gets the height of this object.
         * Not used in this class, but can be used in subclasses.
         */
        override public function set height(value:Number):void
        {
            _height = value;
            draw();
        }
        override public function get height():Number
        {
            return _height;
        }

        /**
         * Sets / gets the color of this tile.
         */
        public function set color(value:uint):void
        {
            _color = value;
            draw();
        }
        public function get color():uint
        {
            return _color;
        }
    }
}
```

可以看到，这个类为构造函数增加了color和height值，这些值会传入类属性。还为这两个值提供了获取方法和设置方法。尽管height值在这个类中没有用到，但可以在子类中使用，以绘制立体方块而不只是平面区块。

www.TopSage.com

TopSage.com

只要高度或颜色发生改变，构造函数就会调用draw方法。它会使用指定的颜色绘制本章中反复出现的这个比例为2:1的菱形。尽管非常简单，不过接下来你可以在此基础上创建由多个区块组成的更大的等角区域，如下例所示，这个例子见TileTest.as文件：

```
package
{
    import com.friendsofed.isometric.DrawnIsoTile;
    import com.friendsofed.isometric.Point3D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    public class TileTest extends Sprite
    {
        public function TileTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var world:Sprite = new Sprite();
            world.x = stage.stageWidth / 2;
            world.y = 100;
            addChild(world);

            for(var i:int = 0; i < 20; i++)
            {
                for(var j:int = 0; j < 20; j++)
                {
                    var tile:DrawnIsoTile = new DrawnIsoTile(20, 0xcccccc);
                    tile.position = new Point3D(i * 20, 0, j * 20);
                    world.addChild(tile);
                }
            }
        }
    }
}
```

这段代码首先创建一个名为world的sprite，它将作为所有区块的一个容器，从而可以将所有区块作为一个组来指定位置。将这个sprite水平移动到屏幕上的中心位置，并距上边缘有一定距离。

然后运行一个双重循环。在内循环体中创建一个新的DrawnIsoTile，大小为20，并使用默认颜色。其*x*轴和*z*轴位置设置为循环变量的一个乘积，并将这个区块增加到world。运行这段代码时，会得到如图3-11所示的结果。

这是一个基于区块的等角世界的基本设置。可以尝试对它做一些试验。需要指出，改变这两个循环的长度可以控制最后得到的网格的大小。因为区块的大小都为20，而且在*x*轴和*z*轴上均相隔20个单位，所以可以完美地排列在一起。试着改变其中某个值，你会看到区块或者重叠，或者区块之间将留有间隙。通常你总是希望区块之间是无缝的，不过应该知道如果确实需要可以如何

www.TopSage.com

进行修改，了解这一点很有好处。同样还可以试着调整y位置。在这个例子中，所有区块的y值都设置为0，所以它们构成一个平面。尝试采用不同的方法修改y，看看会得到什么。另外还可以尝试采用某种方式改变颜色。

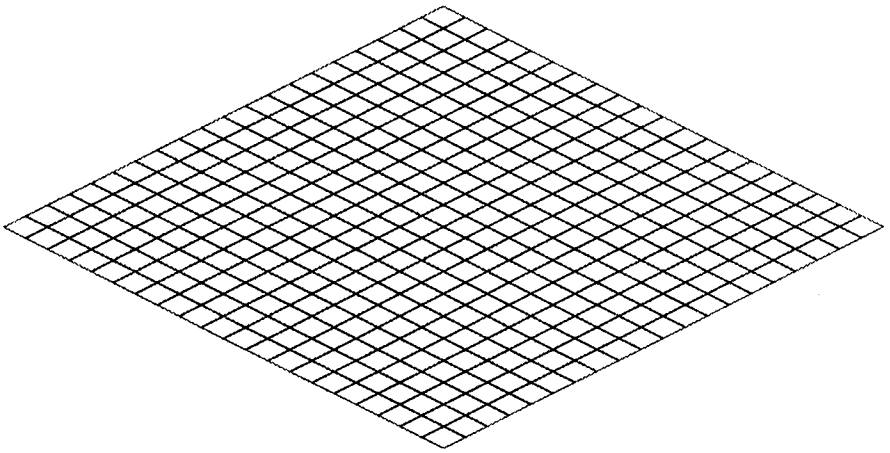


图3-11　网格中的DrawnIsoTile布局

这里甚至没有必要使用双重循环，也不必创建网格。可以通过尝试单独地放置各个区块，或者提出另外某种算法来摆放这些区块。

尽管区块很不错，不过还可以更进一步创建一个高度可变的3D等角体。一旦进入3D世界，就必须开始考虑光照情况。一个等角体有3个可视面，每个可视面分别面向一个不同的方向。为了增强空间感，每个面的阴影应当略有不同。通常左上角或右上角会有一个光源，这样一来上面最亮，左面或右面亮度稍低，这取决于光源在左侧还是右侧。在DrawnIsoBox类（见DrawnIsoBox.as文件）中，假设光源放在右边，所以右面亮度中等，而左面稍暗：

```
package com.friendsofed.isometric
{
    public class DrawnIsoBox extends DrawnIsoTile
    {

        public function DrawnIsoBox(size:Number, color:uint, height:Number)
        {
            super(size, color, height);
        }

        override protected function draw():void
        {
            graphics.clear();
            var red:int = _color >> 16;
            var green:int = _color >> 8 & 0xff;
            var blue:int = _color & 0xff;

            var leftShadow:uint = (red * .5) << 16 | (green * .5) << 8 | (blue * .5);
```

www.TopSage.com

```
            var rightShadow:uint = (red * .75) << 16 | (green * .75) << 8 | (blue * .75);

            var h:Number = _height * Y_CORRECT;

            // draw top
            graphics.beginFill(_color);
            graphics.lineStyle(0, 0, .5);
            graphics.moveTo(-_size, -h);
            graphics.lineTo(0, -_size * .5 - h);
            graphics.lineTo(_size, -h);
            graphics.lineTo(0, _size * .5  - h);
            graphics.lineTo(-_size, -h);
            graphics.endFill();

            // draw left
            graphics.beginFill(leftShadow);
            graphics.lineStyle(0, 0, .5);
            graphics.moveTo(-_size, -h);
            graphics.lineTo(0, _size * .5 - h);
            graphics.lineTo(0, _size * .5);
            graphics.lineTo(-_size, 0);
            graphics.lineTo(-_size, -h);
            graphics.endFill();

            // draw right
            graphics.beginFill(rightShadow);
            graphics.lineStyle(0, 0, .5);
            graphics.moveTo(_size, -h);
            graphics.lineTo(0, _size * .5 - h);
            graphics.lineTo(0, _size * .5);
            graphics.lineTo(_size, 0);
            graphics.lineTo(_size, -h);
            graphics.endFill();
        }
    }
}
```

由于只传入了一个颜色，这个类将该颜色分解为其分量值：红（red）、绿（green）和蓝（blue），将各个分量分别降低某个百分比，再重新组合这3个分量形成左面和右面较暗的颜色。另一种做法是用单独的sprite或形状绘制各个面，然后对用作为右面和左面的形状调整颜色转换值。如果允许动态光照还可以更为灵活，不过这将作为一个练习留给你来完成，你可以继续研究有关的等角内容。

另一个要注意的重要问题是下面这行代码：

```
var h:Number = _height * Y_CORRECT;
```

这行代码取传入的高度，并将其转换为一个精确的转换后等角高度。回忆一下我们在推导等角变换公式时有以下计算：

```
sx = x - z;
sy = y * 1.2247 + (x + z) * .5;
```

www.TopSage.com

我曾提到，有些实现会去掉这里的1.2247，这在某种程度上是可行的，不过这样做可能会让物体看起来像被压扁了似的。由于height是一个y值，因此这里也需要这种修正，否则这个立方体看起来也有被压扁的感觉。创建立方体时大多都会出现这个问题。将大小和高度设置为相同的值（如设置为20）。你期待看到一个立方体，但是它看起来可能更像一个扁扁的蛋糕盒。通过修正这个值，立方体看上去才能真正像个立方体。

当然，我们还需要一个使用DrawnIsoBox类的具体例子。可以在创建对象时通过改变类名很容易地修改TileTest类，不过这样没有多大意思，因为高度仍默认为0，所以它看上去只是一个区块。也可以改变高度，但是这样一来只是得到一个有厚度的平面（尽管不是先前的扁平面）。实际上，我们要做一些更有意思的事情：使用IsoUtils类的screenToIso方法捕获鼠标点击事件，并在点击的位置上动态增加方块。可以下载BoxTest.as得到下面的例子：

```
package
{
    import com.friendsofed.isometric.DrawnIsoBox;
    import com.friendsofed.isometric.DrawnIsoTile;
    import com.friendsofed.isometric.IsoUtils;
    import com.friendsofed.isometric.Point3D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.geom.Point;

    [SWF(backgroundColor=0xffffff)]
    public class BoxTest extends Sprite
    {
        private var world:Sprite;
        public function BoxTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            world = new Sprite();
            world.x = stage.stageWidth / 2;
            world.y = 100;
            addChild(world);
            for(var i:int = 0; i < 20; i++)
            {
                for(var j:int = 0; j < 20; j++)
                {
                    var tile:DrawnIsoTile = new DrawnIsoTile(20, 0xcccccc);
                    tile.position = new Point3D(i * 20, 0, j * 20);
                    world.addChild(tile);
                }
            }

            world.addEventListener(MouseEvent.CLICK, onWorldClick);
        }
```

www.TopSage.com

```
        private function onWorldClick(event:MouseEvent):void
        {
            var box:DrawnIsoBox = new DrawnIsoBox(20, Math.random() * 0xffffff, 20);
            var pos:Point3D =
                IsoUtils.screenToIso(new Point(world.mouseX, world.mouseY));

            box.position = pos;
            world.addChild(box);
        }

    }
}
```

这个类与TileTest类中的大部分代码都相同。创建了区块网格之后，我们为world sprite增加一个鼠标点击事件监听器。点击world时，会创建一个新的方块（box），并指定一个随机的颜色。然后从world sprite得到鼠标坐标，转换为一个等角点，并使用这个点来设置方块的位置。最后，将这个方块增加到world。

利用这个类可以随意地放置方块。最好将x、y和z值四舍五入为最接近的20的倍数，这样可以让新方块与现有的区块对齐：

```
private function onWorldClick(event:MouseEvent):void
{
    var box:DrawnIsoBox = new DrawnIsoBox(20, Math.random() * 0xffffff, 20);
    var pos:Point3D = IsoUtils.screenToIso(new Point(world.mouseX, world.mouseY));
    pos.x = Math.round(pos.x / 20) * 20;
    pos.y = Math.round(pos.y / 20) * 20;
    pos.z = Math.round(pos.z / 20) * 20;
    box.position = pos;
    world.addChild(box);
}
```

现在，点击world时，所创建的方块会与区块构成的网格对齐，如图3-12所示。

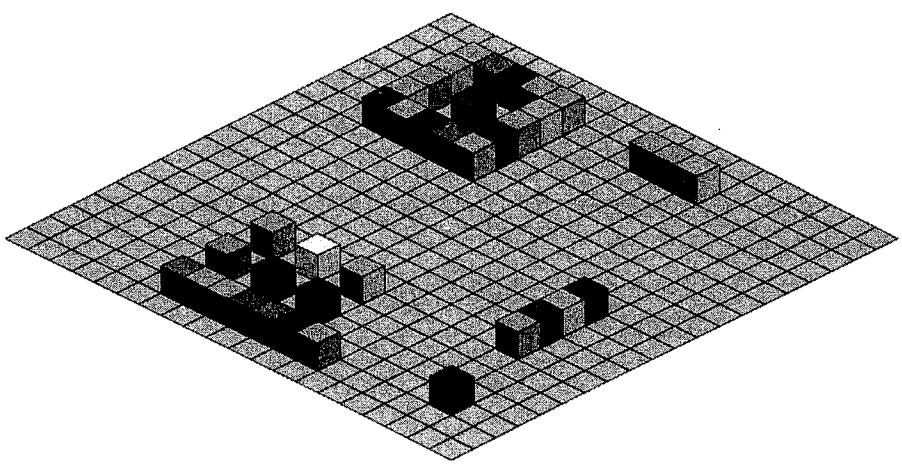


图3-12 区块网格上的DrawnIsoBox

www.TopSage.com

你可能已经注意到，尝试这个示例时还存在一些问题。其中大多数问题都与深度排序有关，这正是我们接下来要讨论的内容。

3.5　深度排序

在上一个演示示例中如果任意点击鼠标，你肯定会看到有时一些方块本来应当在另外一些方块的后面，却错误地放在了前面。图3-13给出了这样一个例子。

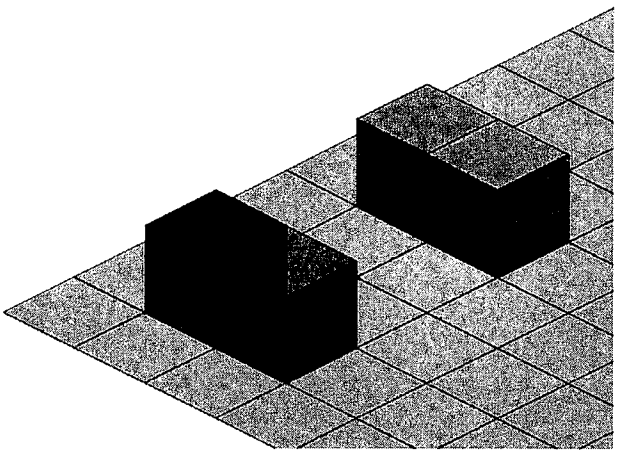


图3-13　糟糕！深度问题

问题在于world sprite的addChild方法只是将新的子对象放在显示列表的最上面，也就是所有其他对象的上面。如果从后向前绘制方块，可能一切正常。图3-13右侧的两个方块就是采用这种方式绘制的。但是左侧的方块则是从前向后绘制，所以尽管第二个方块应当在第一个方块后面，addChild却会把它放在最上面。要处理这种情况，需要确定world中各个对象的深度，并相应地进行排序。

在本章前面我曾提到，有多种不同的等角投影方法。等角深度排序的情况则更为糟糕，每个人对于如何完成对象排序都有自己的小“技巧”。我肯定能找出至少十来种这样的方法，而且各不相同。各个论坛上关于哪一种方法最为适当总是存在争论——甚至可以称得上是“大战”！如果所有对象都有相同的y值，那么很多方法都会很有效。但是一旦开始处理有不同高度的对象，有些方法则会束手无策。

我不打算介绍那些我都不能完全理解的部分可行的“技巧”。对我来说，这个问题完全可以采用一种相当直接的数学方式加以解决。

再回忆前面为创建一个等角视图（*y*轴上旋转45°，*x*轴上旋转–30°）讨论旋转变换时，得到以下公式：

```
x1 = x - z
y1 = y * 1.2247 + (x + z) * .5
z2 = (x + z) * .866 - y  * .707
```

www.TopSage.com

x1和y1是变换后的x和y坐标，用于确定点或对象的屏幕位置。z2是变换后的z轴坐标。尽管当时我们把它搁置一旁未用，不过现在可以用它来完成深度排序，因为正是这个z轴坐标能够提供给我们与视点的距离。这也是所有相关技巧所要实现的（并且分别取得了不同程度的成功）。不过这是精确的数学值，它能妥善地完成排序，而不论物体在各个轴上的位置。

所以这正是我要采用的方法。再来看IsoObject类，会看到以下方法：

```
/**
 * Returns the transformed 3D depth of this object.
 */
public function get depth():Number
{
    return (_position.x + _position.z) * .866 - _position.y * .707;
}
```

可以看到，这个方法计算了变换后的z值。在这里其实也可以使用更精确的值，不过由于这个方法实际上并不用于完成对象的可视化渲染，而只是用来比较两个对象，所以就使用了一些通过四舍五入得到的近似值。

排序的策略是将所有IsoObject置于一个数组中，然后根据depth属性对这个数组进行排序。接下来我们可以根据这个数组的顺序重排显示列表。每次增加一个对象时都必须对数组排序并重排显示列表。我们将在下面的演示中完成这个工作（可以下载DepthTest.as文件得到这个例子）：

```
package
{
    import com.friendsofed.isometric.DrawnIsoBox;
    import com.friendsofed.isometric.DrawnIsoTile;
    import com.friendsofed.isometric.IsoUtils;
    import com.friendsofed.isometric.Point3D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.geom.Point;

    [SWF(backgroundColor=0xffffff)]
    public class DepthTest extends Sprite
    {
        private var world:Sprite;
        private var objectList:Array;

        public function DepthTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            world = new Sprite();
            world.x = stage.stageWidth / 2;
            world.y = 100;
            addChild(world);
```

www.TopSage.com

```
            objectList = new Array();

            for(var i:int = 0; i < 20; i++)
            {
                for(var j:int = 0; j < 20; j++)
                {
                    var tile:DrawnIsoTile = new DrawnIsoTile(20, 0xcccccc);
                    tile.position = new Point3D(i * 20, 0, j * 20);
                    world.addChild(tile);
                    objectList.push(tile);
                }
            }
            sortList();
            world.addEventListener(MouseEvent.CLICK, onWorldClick);
        }

        private function onWorldClick(event:MouseEvent):void
        {
            var box:DrawnIsoBox = new DrawnIsoBox(20, Math.random() * 0xffffff, 20);
            var pos:Point3D = IsoUtils.screenToIso(new Point(world.mouseX, world.mouseY));
            pos.x = Math.round(pos.x / 20) * 20;
            pos.y = Math.round(pos.y / 20) * 20;
            pos.z = Math.round(pos.z / 20) * 20;
            box.position = pos;
            world.addChild(box);
            objectList.push(box);
            sortList();
        }

        private function sortList():void
        {
            objectList.sortOn("depth", Array.NUMERIC);
            for(var i:int = 0; i < objectList.length; i++)
            {
                world.setChildIndex(objectList[i], i);
            }
        }
    }
}
```

在这里，我们创建了一个名为objectList的数组。各个区块和方块都将增加到这个数组中，然后运行sortList方法。这个方法根据depth属性对数组排序。不要忘记指定排序规则为Array.NUMERIC，否则它会按字符串值对数组排序，这样一来会导致"70"大于"100"。（实在不好意思，我得承认自己曾经花了很长时间来调试这个问题！）

等角深度排序有一个限制，它的前提是世界中的对象都有相同的大小（这里的大小是指IsoObject类的size属性，它定义了对象在x–z平面上的正方形覆盖区域）。对象可以是任何高度，而这不会导致问题的出现，不过不能比同一世界中的其他对象更宽或更深。如果只是一、两个角色对象比其他区块小，通常这是可以的，但还是应当选择一个标准对象大小，并坚持这一标准。我见过很多人试图利用不同的对象大小建立一个等角世界。这往往会导致极

www.TopSage.com

其复杂的深度排序，以至于除了程序员本人，其他人都无法真正理解，而且通常也很容易出错。或者他们可能会对对象的形状和位置提出额外的复杂限制，以弥补标准区块大小限制。要建立比较大型的对象，更好的做法是将这些大对象分解为多个对象，每个对象都有标准的大小。

不论你把方块放在哪里，最后这个例子能很好地完成方块排序。不过，有时区块反而出现在方块的上面。这是因为将方块放在一个区块的“上面”时，实际上它们有完全相同的x, y, z位置，这说明它们转换后的深度完全一样。所以从排序的角度看，谁先谁后没有影响。这个问题可以用两种方法来加以处理：将区块稍向下放，或者将方块稍稍提高一点。幅度不必太大——可能眼睛看不出来，但是足以影响深度计算。可以在创建区块的代码行中做到这一点：

```
tile.position = new Point3D(i * 20, 0.1, j * 20);
```

将各个区块的y设置为0.1就足以把它放在方块下面，但是从视觉上看它并没有移动。

另一种稍微复杂一些（但更高效）的方法是建立两个不同的world sprite——一个对应区块，另一个对应方块。不需要每次都对所有这400个区块进行排序。它们不会移动，而且你总是希望所有区块都在所有方块的下面。所以把这些区块放在相应的sprite中，再把该sprite放在包含方块的sprite下面。只需将方块放在objectList中并进行排序。以下是修改后的类，见DepthTest2.as文件：

```
package
{
    import com.friendsofed.isometric.DrawnIsoBox;
    import com.friendsofed.isometric.DrawnIsoTile;
    import com.friendsofed.isometric.IsoUtils;
    import com.friendsofed.isometric.Point3D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.geom.Point;

[SWF(backgroundColor=0xffffff)]
public class DepthTest2 extends Sprite
{
    private var floor:Sprite;
    private var world:Sprite;
    private var objectList:Array;

    public function DepthTest2()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        floor = new Sprite();
        floor.x = stage.stageWidth / 2;
        floor.y = 100;
        addChild(floor);
```

www.TopSage.com

大家网 TopSage.com

```
            world = new Sprite();
            world.x = stage.stageWidth / 2;
            world.y = 100;
            addChild(world);

            objectList = new Array();

            for(var i:int = 0; i < 20; i++)
            {
                for(var j:int = 0; j < 20; j++)
                {
                    var tile:DrawnIsoTile =  new DrawnIsoTile(20, 0xcccccc);
                    tile.position = new Point3D(i * 20, 0, j * 20);
                    floor.addChild(tile);
                }
            }
            stage.addEventListener(MouseEvent.CLICK, onWorldClick);
        }

        private function onWorldClick(event:MouseEvent):void
        {
            var box:DrawnIsoBox = new DrawnIsoBox(20, Math.random() * 0xffffff, 20);
            var pos:Point3D =
                        IsoUtils.screenToIso(new Point(world.mouseX, world.mouseY));
            pos.x = Math.round(pos.x / 20) * 20;
            pos.y = Math.round(pos.y / 20) * 20;
            pos.z = Math.round(pos.z / 20) * 20;
                box.position = pos;
                world.addChild(box);
                objectList.push(box);
                sortList();
            }

            private function sortList():void
            {
                objectList.sortOn("depth", Array.NUMERIC);
                for(var i:int = 0; i < objectList.length; i++)
                {
                    world.setChildIndex(objectList[i], i);
                }
            }
        }
    }
```

在这里创建了一个floor sprite（地面sprite），放在world sprite相同的位置上。由于它先增加到显示列表，因此将位于world sprite下面。所有区块都增加到这个floor中。因为它们不需要排序，所以不会放入objectList。

这里修改了鼠标点击处理程序来监听对场景的点击事件，因为world sprite最初是空的，无法接收任何鼠标点击事件。除此以外，所有代码都完全相同。

下一步将结合之前完成的所有工作来建立一个可重用的世界类。

3.6 等角世界类

之前创建了一个floor sprite、一个world sprite和对象列表，并建立了排序算法来完成一些工作，由于其中的大部分工作可能在大多数项目中都需要做到，因此有必要创建一个类以一种通用的方式处理所有这些工作，从而能够在每一个项目中使用，这很有意义。以下是IsoWorld类，可以下载IsoWorld.as文件得到：

```
package com.friendsofed.isometric
{
    import flash.display.Sprite;
    import flash.geom.Rectangle;

    public class IsoWorld extends Sprite
    {
        private var _floor:Sprite;
        private var _objects:Array;
        private var _world:Sprite;

        public function IsoWorld()
        {
            _floor = new Sprite();
            addChild(_floor);

            _world = new Sprite();
            addChild(_world);

            _objects = new Array();
        }

        public function addChildToWorld(child:IsoObject):void
        {
            _world.addChild(child);
            _objects.push(child);
            sort();
        }

        public function addChildToFloor(child:IsoObject):void
        {
            _floor.addChild(child);
        }

        public function sort():void
        {
            _objects.sortOn("depth", Array.NUMERIC);
            for(var i:int = 0; i < _objects.length; i++)
            {
                _world.setChildIndex(_objects[i], i);
            }
        }
    }
}
```

其中大部分内容正是我们在上一个例子中所做的工作：它创建了一个floor sprite、一个world

www.TopSage.com

sprite以及一个对象列表，并处理排序。它还提供了两个方法来增加对象。addChildToFloor方法把一个对象放入floor sprite，而不是对象列表，而且不进行排序。这里认为floor（地面）是创建在非重叠网格中的平面区块，所以没有必要排序。addChildToWorld方法将对象增加到world sprite和对象列表，对列表排序并重排显示列表。后面还将为这个类增加其他功能以便于完成碰撞检测。

IsoWorld类的使用相当容易，如WorldTest.as文件中的例子所示：

```
package
{
    import com.friendsofed.isometric.DrawnIsoBox;
    import com.friendsofed.isometric.DrawnIsoTile;
    import com.friendsofed.isometric.IsoUtils;
    import com.friendsofed.isometric.IsoWorld;
    import com.friendsofed.isometric.Point3D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.geom.Point;

    [SWF(backgroundColor=0xffffff)]
    public class WorldTest extends Sprite
    {
        private var world:IsoWorld;

        public function WorldTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            world = new IsoWorld();
            world.x = stage.stageWidth / 2;
            world.y = 100;
            addChild(world);

            for(var i:int = 0; i < 20; i++)
            {
                for(var j:int = 0; j < 20; j++)
                {
                    var tile:DrawnIsoTile = new DrawnIsoTile(20, 0xcccccc);
                    tile.position = new Point3D(i * 20, 0, j * 20);
                    world.addChildToFloor(tile);
                }
            }
            stage.addEventListener(MouseEvent.CLICK, onWorldClick);
        }

        private function onWorldClick(event:MouseEvent):void
        {
            var box:DrawnIsoBox = new DrawnIsoBox(20, Math.random() * 0xffffff, 20);
            var pos:Point3D =
                        IsoUtils.screenToIso(new Point(world.mouseX, world.mouseY));
```

www.TopSage.com

TopSage.com

```
                pos.x = Math.round(pos.x / 20) * 20;
                pos.y = Math.round(pos.y / 20) * 20;
                pos.z = Math.round(pos.z / 20) * 20;
                box.position = pos;
                world.addChildToWorld(box);
            }
        }
    }
```

实际上，这个文件与原来的BoxTest类几乎相同，只是world是一个IsoWorld而不是一个Sprite，另外使用了addChildToFloor和addChildToWorld而不是addChild。不过IsoWorld要负责创建不同的sprite并完成所有排序。

3.7 3D中的移动

在等角3D世界中，移动本身并不算大问题，特别是如果有类似IsoObject这样一个类时，它可以自动负责将3D坐标转换为屏幕位置。只需修改属性（x, y或z），对象就会移动到正确的屏幕位置上。唯一要记住的是，每次移动一个对象时，都需要调用IsoWorld的sort方法（或者如果没有使用IsoWorld，则要更新深度排序例程），考虑到对象可能有不同的深度。

在此基础上，实现各种运动就非常容易了：基本速度、加速度、重力、摩擦力、反弹、缓冲、跳跃等（参见《Flash ActionScript 3.0 动画教程》中的介绍）。实际上，为简洁起见，下面将为IsoObject增加3个新属性来处理速度：

```
protected var _vx:Number = 0;
protected var _vy:Number = 0;
protected var _vz:Number = 0;
```

另外还要为这3个属性增加一些获取方法和设置方法：

```
/**
 * Sets / gets the velocity on the x axis.
 */
public function set vx(value:Number):void
{
    _vx = value;
}
public function get vx():Number
{
    return _vx;
}

/**
 * Sets / gets the velocity on the y axis.
 */
public function set vy(value:Number):void
{
    _vy = value;
}
public function get vy():Number
{
    return _vy;
```

www.TopSage.com

```
}

/**
 * Sets / gets the velocity on the z axis.
 */
public function set vz(value:Number):void
{
    _vz = value;
}
public function get vz():Number
{
    return _vz;
}
```

显然要建立一些键盘事件监听器，并根据所按下的按键移动对象。下面的演示示例（见可下载的MotionTest.as文件）就是要完成这个工作：

```
package
{
    import com.friendsofed.isometric.DrawnIsoBox;
    import com.friendsofed.isometric.DrawnIsoTile;
    import com.friendsofed.isometric.IsoWorld;
    import com.friendsofed.isometric.Point3D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
    import flash.ui.Keyboard;

    [SWF(backgroundColor=0xffffff)]

    public class MotionTest extends Sprite
    {
        private var world:IsoWorld;
        private var box:DrawnIsoBox;
        private var speed:Number = 20;

        public function MotionTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            world = new IsoWorld();
            world.x = stage.stageWidth / 2;
            world.y = 100;
            addChild(world);
            for(var i:int = 0; i < 20; i++)
            {
                for(var j:int = 0; j < 20; j++)
                {
                    var tile:DrawnIsoTile = new DrawnIsoTile(20, 0xcccccc);
                    tile.position = new Point3D(i * 20, 0, j * 20);
```

www.TopSage.com

```
                world.addChildToFloor(tile);
            }
        }

        box = new DrawnIsoBox(20, 0xff0000, 20);
        box.x = 200;
        box.z = 200;
        world.addChildToWorld(box);

        stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
        stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
    }

    private function onKeyDown(event:KeyboardEvent):void
    {
        switch(event.keyCode)
        {
            case Keyboard.UP :
            box.vx = -speed;
            break;

            case Keyboard.DOWN :
            box.vx = speed;
            break;

            case Keyboard.LEFT :
            box.vz = speed;
            break;

            case Keyboard.RIGHT :
            box.vz = -speed;
            break;
            default :
            break;

        }
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onKeyUp(event:KeyboardEvent):void
    {
        box.vx = 0;
        box.vz = 0;
        removeEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        box.x += box.vx;
        box.y += box.vy;
        box.z += box.vz;
        world.sort();
    }
  }
}
```

www.TopSage.com

大家网 TopSage.com

这相当直观。先建立一个区块地面，然后在上面放置一个方块。这里对按键弹起和按键压下事件创建了相应的监听器。如果压下某个箭头键，就会相应地设置方块的vx或vz。在onEnterFrame方法中，方块的速度值会增加到其位置。相当简单。不要忘记每次移动后对world排序，见onEnterFrame的最后一行代码。实际上，在这种情况下（world中只有一个对象），这一点并不重要，不过最好养成这个好习惯。

开始时speed设置为20，所以每次方块都会移动一个完整的区块空间。试着将这个值改为一个更小的值，让它以更慢的速度移动。

看看对于其他类型的物理运动还能做些什么，以下是一个更高级的演示，可以从MotionTest2.as文件得到这个例子：

```
package
{
    import com.friendsofed.isometric.DrawnIsoBox;
    import com.friendsofed.isometric.DrawnIsoTile;
    import com.friendsofed.isometric.IsoWorld;
    import com.friendsofed.isometric.Point3D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.filters.BlurFilter;

    [SWF(backgroundColor=0xffffff)]
    public class MotionTest2 extends Sprite

    {
        private var world:IsoWorld;
        private var box:DrawnIsoBox;
        private var shadow:DrawnIsoTile;
        private var gravity:Number = 2;
        private var friction:Number = 0.95;
        private var bounce:Number = -0.9;
        private var filter:BlurFilter;

        public function MotionTest2()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            world = new IsoWorld();
            world.x = stage.stageWidth / 2;
            world.y = 100;
            addChild(world);

            for(var i:int = 0; i < 20; i++)
            {
                for(var j:int = 0; j < 20; j++)
                {
```

www.TopSage.com

```
            var tile:DrawnIsoTile = new DrawnIsoTile(20, 0xcccccc);
            tile.position = new Point3D(i * 20, 0, j * 20);
            world.addChildToFloor(tile);
        }
    }

    box = new DrawnIsoBox(20, 0xff0000, 20);
    box.x = 200;
    box.z = 200;
    world.addChildToWorld(box);

    shadow = new DrawnIsoTile(20, 0);
    shadow.alpha = .5;
    world.addChildToFloor(shadow);

    filter = new BlurFilter();

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.addEventListener(MouseEvent.CLICK, onClick);
}

private function onClick(event:MouseEvent):void
{
    box.vx = Math.random() * 20 - 10;
    box.vy = -Math.random() * 40;
    box.vz = Math.random() * 20 - 10;
}

private function onEnterFrame(event:Event):void
{
    box.vy += 2;
    box.x += box.vx;
    box.y += box.vy;
    box.z += box.vz;
    if(box.x > 380)
    {
        box.x = 380;
        box.vx *= -.8;
    }
    else if(box.x < 0)
    {
        box.x = 0;
        box.vx *= bounce;
    }
    if(box.z > 380)
    {
        box.z = 380;
        box.vz *= bounce;
    }
    else if(box.z < 0)
    {
        box.z = 0;
        box.vz *= bounce;
    }
```

www.TopSage.com

```
                if(box.y > 0)
                {
                    box.y = 0;
                    box.vy *= bounce;
                }
                box.vx *= friction;
                box.vy *= friction;
                box.vz *= friction;

                shadow.x = box.x;
                shadow.z = box.z;
                filter.blurX = filter.blurY = -box.y * .25;
                shadow.filters = [filter];
                world.sort();
            }
        }
    }
```

这个演示充分展示了重力、弹力和摩擦力以及*y*轴上的运动。点击鼠标时，方块在各个轴上会得到一个随机的速度。方块经过*y*轴的0点或者超出*x*轴或*z*轴平面边缘时，就会弹回。这里还应用了一些摩擦力。

在此还将DrawnIsoTile设置为有50%的*α*透明度，从而引入了一个阴影。这放在floor层上，不过在每一帧都会移动到方块所在的相同的*x*和*z*位置。这里还应用模糊滤镜来模拟方块高度带来的阴影减弱。这些都不算太复杂，但效果却很好。

到目前为止，在这个只有一个对象的孤立世界中一切都很好。如果再增加更多的对象会发生什么呢？下面就来讨论这个问题。

3.8　碰撞检测

要了解为什么碰撞检测很重要，先看第一个MotionTest类，并建立另外一两个方块，如下所示：

```
var newBox:DrawnIsoBox = new DrawnIsoBox(20, 0xcccccc, 20);
newBox.x = 300;
newBox.z = 300;
world.addChildToWorld(newBox);
```

它们不必移动也不必做任何操作，只是放在网格上而已。现在移动先前那个可移动的方块，直至碰上新增的某个方块。情况不太妙，是不是？方块会直接相互穿过，这个过程看起来很可怕。很多情况下移动方块往往会在静止方块的下面，看起来好像深度排序突然失效了。实际上只是两个对象在试图占据同一个空间。真实世界中发生这种情况时，可能“挡板”会被撞弯。在等角模拟中，则会出现深度排序失败。

为了处理这种情况，需要有一种方法来了解对象能移动到哪里，以及不能移动到哪里。由于IsoWorld类保存了world中的所有对象的一个列表，因此这个功能很适合放在这个类中。我们将这个方法称为canMove。如下所示：

www.TopSage.com

TopSage.com

```
public function canMove(obj:IsoObject):Boolean
{
    var rect:Rectangle = obj.rect;
    rect.offset(obj.vx, obj.vz);

    for(var i:int = 0; i < _objects.length; i++)
    {
        var objB:IsoObject = _objects[i] as IsoObject;
        if(obj != objB && objB.walkable && rect.intersects(objB.rect))
        {
            return false;
        }
    }
    return true;
}
```

这个方法接受一个IsoObject实例，并返回该对象是否能安全地移动，即如果将其x和z速度加至其位置，这个对象是否能安全地移动到它应该在的位置上。

再来看IsoObject类，你会记得它有一个rect属性，这是flash.geom.Rectangle的一个实例。这表示对象在*x-z*平面上的覆盖区域。这里从所传入对象得到该rect，并将其按对象的x和z速度量进行偏移。现在这个rect则表示完成下一个移动时对象的覆盖区域。

然后循环处理world中的对象列表。这里希望检查3个条件。

- 首先，所检查的对象不是传入的对象。你不希望对对象自身完成碰撞测试。
- 其次，所检查的对象标志为不可通过，**可通过**（walkable）是指另一个对象可以占据该区块所在的同一空间。有些情况下，world中的一个对象是可以走上去的一个平面区块，类似于一个地面区块。或者可能是*y*轴之上的一个区块，允许另一个对象在它下面“走过”。
- 最后，计算出的偏移矩形与所检查对象的rect不相交，利用Rectangle类的内置intersects方法很容易检查这一点。

如果对于列表中的某个对象这3个条件都为true，则说明这个对象根据其x和z速度不能移动到一个新位置，因此会立即返回false。不过，如果处理完整个列表而没有发现碰撞，则可以安全地移动并返回true。

用法是在将速度加至位置之前检查canMove。以下是一个完整的例子，见本书下载页面的CollisionTest.as文件：

```
package
{
    import com.friendsofed.isometric.DrawnIsoBox;
    import com.friendsofed.isometric.DrawnIsoTile;
    import com.friendsofed.isometric.IsoWorld;
    import com.friendsofed.isometric.Point3D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
```

www.TopSage.com

```
import flash.ui.Keyboard;

[SWF(backgroundColor=0xffffff)]
public class CollisionTest extends Sprite
{
    private var world:IsoWorld;
    private var box:DrawnIsoBox;
    private var speed:Number = 4;

    public function CollisionTest()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;
        world = new IsoWorld();
        world.x = stage.stageWidth / 2;
        world.y = 100;
        addChild(world);

        for(var i:int = 0; i < 20; i++)
        {
            for(var j:int = 0; j < 20; j++)
            {
                var tile:DrawnIsoTile = new DrawnIsoTile(20, 0xcccccc);
                tile.position = new Point3D(i * 20, 0, j * 20);
                world.addChildToFloor(tile);
            }
        }

        box = new DrawnIsoBox(20, 0xff0000, 20);
        box.x = 200;
        box.z = 200;
        world.addChildToWorld(box);

        var newBox:DrawnIsoBox = new DrawnIsoBox(20, 0xcccccc, 20);
        newBox.x = 300;
        newBox.z = 300;
        world.addChildToWorld(newBox);

        stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
        stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
    }

    private function onKeyDown(event:KeyboardEvent):void
    {
        switch(event.keyCode)
        {
            case Keyboard.UP :
            box.vx = -speed;
            break;

            case Keyboard.DOWN :
            box.vx = speed;
            break;
```

www.TopSage.com

```
                case Keyboard.LEFT :
                box.vz = speed;
                break;

                case Keyboard.RIGHT :
                box.vz = -speed;
                break;
                default :
                break;

            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onKeyUp(event:KeyboardEvent):void
        {
            box.vx = 0;
            box.vz = 0;
            removeEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            if(world.canMove(box))
            {
                box.x += box.vx;
                box.y += box.vy;
                box.z += box.vz;
            }
            world.sort();
        }
    }
}
```

可以看到，这里唯一的改变是把运动代码包在一个if语句中，查看是否可以安全地移动。这里还把速度减小为一个更小的数，以便看到确实能妥善而准确地完成碰撞检测。

以上介绍了等角引擎的基本物理性质。最后两小节将介绍如何集成外部图形以及设计等角世界的布局。

3.9 使用外部图形

DrawnIsoTile和DrawnIsoBox类对于测试和开发来说非常棒，不过在实际游戏中可能很快会变得太原始。你或者你的图形设计师很可能会用一些图形程序（如PhotoShop、Fireworks，甚至是Flash）建立一些更丰富的等角对象。需要某种方法将这些图形结合到IsoObject中。为此，现在建立GraphicTile类，可以下载GraphicTile.as文件得到这个类：

```
package com.friendsofed.isometric
{
    import flash.display.DisplayObject;

    public class GraphicTile extends IsoObject
```

www.TopSage.com

大家网 TopSage.com

```
    {
        public function GraphicTile(size:Number, classRef:Class,
                                     xoffset:Number, yoffset:Number):void
        {
            super(size);
            var gfx:DisplayObject = new classRef() as DisplayObject;
            gfx.x = -xoffset;
            gfx.y = -yoffset;
            addChild(gfx);
        }
    }
}
```

可以看到，这个类扩展了IsoObject，而且除了size外，它还有一个类引用以及x和y两个偏移量。这个类引用是链接到某个图形的类。这通常通过一个嵌入元数据语句完成，参见《Flash ActionScript 3.0 动画教程》中的介绍。不过也可以是另外某个扩展了可视化显示对象的类（如Sprite、Shape、MovieClip或Bitmap），而且这个类提供了相应代码可以在其自身绘制某种类型的图形。无论怎样，GraphicTile的构造函数都会创建这个类的一个实例，并增加到自己的显示列表中。然后根据传入的两个偏移值完成其移动。

要了解这些偏移值的工作原理，以及为什么要提供偏移值，下面首先来看一些外部图形。我并不是一个优秀的等角设计艺术家，不过完全可以利用Fireworks并组合一些区块来创建这些外部图形。第一个图形是一个简单的区块。画布大小为40×20，这就是我们熟悉的菱形，并填充一种木质纹理（参见图3-14）。相信你肯定能做得更棒！这个区块保存为tile_01.png，放在主类的同一目录下。

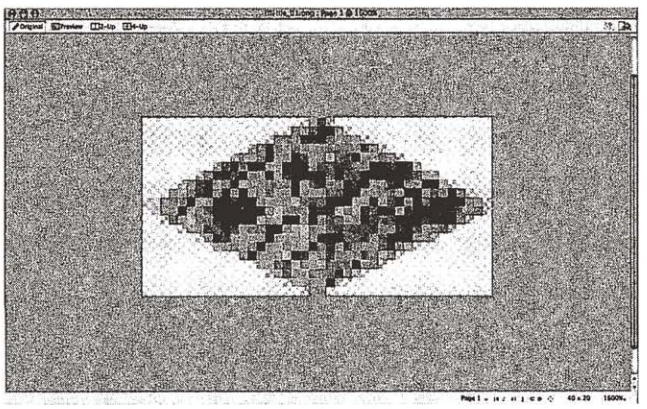


图3-14　Fireworks中一个简单的等角区块

第二个图形稍大一点，也更丰富一些。图形大小为40×40，包含一个简单的立方体，不过这里增加了一个门和一个窗户，这样就变成一个房子。你可能猜不到，这里创建图形极其简单，要知道与创建本节的图形相比，我写这些解释和代码花费的时间反而要更多。这个图形保存为tile_02.png（参见图3-15）。

www.TopSage.com

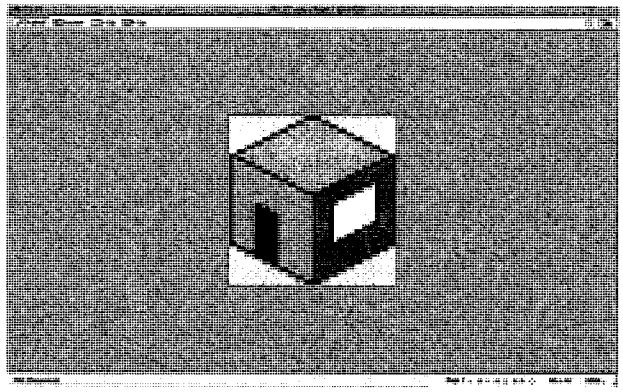


图3-15 另一个Fireworks等角对象

在这个例子中（GraphicTest.as文件），我重用了WorldTest类，将区块类型改为GraphicTile。这里嵌入了刚保存的两个图形，并将这些类传入GraphicTile构造函数：

```
package
{
    import com.friendsofed.isometric.GraphicTile;
    import com.friendsofed.isometric.IsoUtils;
    import com.friendsofed.isometric.IsoWorld;
    import com.friendsofed.isometric.Point3D;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.geom.Point;

    [SWF(backgroundColor=0xffffff)]
    public class GraphicTest extends Sprite
    {
        private var world:IsoWorld;

        [Embed(source="tile_01.png")]
        private var Tile01:Class;

        [Embed(source="tile_02.png")]
        private var Tile02:Class;

        public function GraphicTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            world = new IsoWorld();
            world.x = stage.stageWidth / 2;
            world.y = 100;
            addChild(world);
```

www.TopSage.com

```
            for(var i:int = 0; i < 20; i++)
            {
                for(var j:int = 0; j < 20; j++)
                {
                    var tile:GraphicTile = new GraphicTile(20, Tile01, 20, 10);
                    tile.position = new Point3D(i * 20, 0, j * 20);
                    world.addChildToFloor(tile);
                }
            }
            stage.addEventListener(MouseEvent.CLICK, onWorldClick);
        }

        private function onWorldClick(event:MouseEvent):void
        {
            var box:GraphicTile = new GraphicTile(20, Tile02, 20, 30);
            var pos:Point3D =
                        IsoUtils.screenToIso(new Point(world.mouseX, world.mouseY));
            pos.x = Math.round(pos.x / 20) * 20;
            pos.y = Math.round(pos.y / 20) * 20;
            pos.z = Math.round(pos.z / 20) * 20;
            box.position = pos;
            world.addChildToWorld(box);
        }
    }
}
```

现在再回到刚才的问题：这两个偏移值到底是做什么的？到目前为止，我们创建图形时都是将区块的中心点作为对准点。换句话说，将一个区块放在屏幕上某个（x, y）坐标时，是指这个区块以这一点为中心点。如果区块有高度，则会由该点向上延伸。

但是，当嵌入和实例化图形并添加到一个显示列表时，其对准点位于左上角，它们会向右下方延伸。所以需要将区块向左上方移动，使它的中心位于对准点。第一个区块tile_01.png大小是40×20，这是一个简单的平面区块，所以需要向左移20像素，并向上移10像素，使之居中。另一个区块tile_02.png则稍有些复杂。X偏移仍是20像素，使区块水平居中，但是实际上需要上移30像素才能使用立方体的底面在对准点居中，所以它的偏移量是(20, 30)。

运行这个演示时，会看到豪华的木质地板（可能更像一个渲染效果不佳的沙地）。点击这个地面，开始构建一个由相同房屋构成的小村庄，从而得到如图3-16所示的结果。

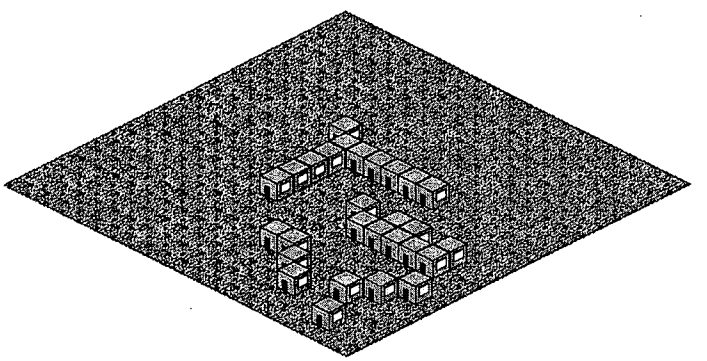


图3-16　GraphicTile的实际使用

现在已经有了一种方法可以将任何类型的图形结合到区块中，下面只需要一种方法，指定所有这些区块的布局。

www.TopSage.com

TopSage.com

3.10 等角区块图

现在已经有一个很小但很不错的等角引擎。我们可以放置区块，利用各种物理方式移动这些区块，捕获3D鼠标点击事件，按深度排序，完成碰撞检测，导入各种图形，等等。现在已经做好了所需的基本准备，可以将它们集成在一起建立一个更高级的游戏了。在目前这个阶段，有一点会对我们有帮助，那就是需要有一种合适的方法指定和构建世界，而不要将所有位置和对象类型都硬编码，这样一来，就能编辑和修改这个世界而不必重新编译整个工程。

可以利用**区块图**（tile map）来完成，这些区块图是一些简单的文本文件，用于指定希望将哪些类型的区块放在哪里。下面是一个简单的区块图：

```
0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 0
0 1 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 1 0
0 1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0
```

这里显示了一个10×10的区块网格，大多为类型0，只有一个内部方格由类型1区块构成。什么是类型0和类型1区块？这由你来决定。基本说来，需要编辑这个区块图文本文件，并把它保存在游戏或应用程序可以访问的一个位置。应用加载并解析这个文件，然后循环处理，创建区块并将区块增加到世界。由你来指定类型0和类型1区块分别代表什么。例如，可以指定类型0表示一个DrawnIsoTile，大小为20，颜色为0xcccccc而且可通过。为了让这个过程更简单，我创建了一个MapLoader类，它会加载一个指定的区块图文件，并允许你注册包含不同角色的不同区块类型。加载并解析这个区块图时会触发一个事件，然后从中得到一个准备就绪的世界。再没有比这更简单的了。（也许还有更简单的做法，不过这是一个很不错的起点。）下面给出这个类，见MapLoader.as文件：

```
package com.friendsofed.isometric
{
    import flash.events.Event;
    import flash.events.EventDispatcher;
    import flash.net.URLLoader;
    import flash.net.URLRequest;
    import flash.utils.getDefinitionByName;

    public class MapLoader extends EventDispatcher
    {
        private var _grid:Array;
        private var _loader:URLLoader;
        private var _tileTypes:Object;
        public function MapLoader()
        {
            _tileTypes = new Object();
```

www.TopSage.com

```
}

/**
 * Loads a text file from the specified url.
 * @param url The location of the text file to load.
 */
public function loadMap(url:String):void
{
    _loader = new URLLoader();
    _loader.addEventListener(Event.COMPLETE, onLoad);
    _loader.load(new URLRequest(url));
}

/**
 * Parses text file into tile definitions and map.
 */
private function onLoad(event:Event):void
{
    _grid = new Array();
    var data:String = _loader.data;

    // first get each line of the file.
    var lines:Array = data.split("\n");
    for(var i:int = 0; i < lines.length; i++)
    {
        var line:String = lines[i];

        // if line is a tile type definition.
        if(isDefinition(line))
        {
            parseDefinition(line);
        }
        // otherwise, if line is not empty and not a comment,
        // it's a list of tile types. add them to grid.
        else if(!lineIsEmpty(line) && !isComment(line))
        {
            var cells:Array = line.split(" ");
            _grid.push(cells);
        }
    }
    dispatchEvent(new Event(Event.COMPLETE));
}

private function parseDefinition(line:String):void
{
    // break apart the line into tokens
    var tokens:Array = line.split(" ");

    // get rid of #
    tokens.shift();

    // first token is the symbol
    var symbol:String = tokens.shift() as String;
```

www.TopSage.com

```
        // loop through the rest of the tokens,
        // which are key/value pairs separated by :
        var definition:Object = new Object();
        for(var i:int = 0; i < tokens.length; i++)
        {
            var key:String = tokens[i].split(":")[0];
            var val:String = tokens[i].split(":")[1];
            definition[key] = val;
        }

        // register the type and definition
        setTileType(symbol, definition);
    }

    /**
     * Links a symbol with a definition object.
     * @param symbol The character to use for the definition.
     * @param definition An object with definition properties
     */
    public function setTileType(symbol:String, definition:Object):void
    {
        _tileTypes[symbol] = definition;
    }

    /**
     * Creates an IsoWorld, iterates through loaded map,
     * adding tiles to it based on map and definitions.
     * @size The tile size to use when making the world.
     * @return A fully populated IsoWorld instance.
     */
    public function makeWorld(size:Number):IsoWorld
    {
        var world:IsoWorld = new IsoWorld();
        for(var i:int = 0; i < _grid.length; i++)
        {
            for(var j:int = 0; j < _grid[i].length; j++)
            {
                    var cellType:String = _grid[i][j];
                    var cell:Object = _tileTypes[cellType];
                    var tile:IsoObject;
                    switch(cell.type)
                    {
                        case  "DrawnIsoTile" :
                        tile = new DrawnIsoTile(size, parseInt(cell.color),
                                                parseInt(cell.height));
                        break;

                        case  "DrawnIsoBox" :
                        tile = new DrawnIsoBox(size, parseInt(cell.color),
                                               parseInt(cell.height));
                        break;

                        case  "GraphicTile" :
```

www.TopSage.com

大家网 TopSage.com

```
                    var graphicClass:Class = getDefinitionByName(
                                                cell.graphicClass
                                            ) as Class;
                    tile = new GraphicTile(size, graphicClass,
                                           parseInt(cell.xoffset),
                                           parseInt(cell.yoffset));
                    break;
                    default :
                    tile = new IsoObject(size);
                    break;
                }
                tile.walkable = cell.walkable == "true";
                tile.x = j * size;
                tile.z = i * size;
                world.addChild(tile);
            }
        }
        return world;
    }

    /**
     * Returns true if line contains only spaces.
     * @param line The string to test.
     */
    private function lineIsEmpty(line:String):Boolean
    {
        for(var i:int = 0; i < line.length; i++)
        {
            if(line.charAt(i) != " ") return false;
        }
        return true;
    }

    /**
     * Returns true if line is a comment (starts with //).
     * @param line The string to test.
     */
    private function isComment(line:String):Boolean
    {
        return line.indexOf("//") == 0;
    }

    /**
     * Returns true if line is a definition (starts with #).
     * @param line The string to test.
     */
    private function isDefinition(line:String):Boolean
    {
        return line.indexOf("#") == 0;
    }
  }
}
```

这是一个相当复杂的类，不过基于这个类构建世界会容易得多。首先加载loadMap方法参数

中指定的文本文件。下面来看一个示例区块图文件：

```
// this is a comment.

# 0 type:GraphicTile graphicClass:MapTest_Tile01 xoffset:20 yoffset:10walkable:true
# 1 type:GraphicTile graphicClass:MapTest_Tile02 xoffset:20 yoffset:30walkable:false
# 2 type:DrawnIsoBox color:0xff6666 walkable:false height:20
# 3 type:DrawnIsoTile color:0x6666ff walkable:false


0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 0
0 1 0 0 0 0 0 0 0 0
0 1 0 3 3 3 3 0 0 0
0 1 0 3 2 2 3 0 0 0
0 1 0 3 2 2 3 0 0 0
0 1 0 3 3 3 3 0 0 0
0 1 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0
```

以//开头的行是注释，可以忽略。以#开头的行是定义。区块定义包括一个符号（在这里就是0, 1, 2, 3）和一组键/值对。所有其他非空的行都是区块行。

加载区块图文件时，会运行onLoad方法，这个方法将解析文本，并完成以下工作。

(1) 将文本文件分解为一个行数组。

(2) 对于每一行，会确定这是一个区块定义、一个注释、一个空行还是一个区块行。

(3) 如果当前行是一个定义，则由parseDefinition方法解析。这个方法取出各个键/值对，并将其赋为一个通用对象的属性，然后将这个对象存储在_tileTypes对象中。

(4) 作为区块行的代码行将分解为一个符号数组，并存储在_grid数组中。

(5) 完成上述工作后，它会广播一个完成事件，通知文件已经得到加载和处理。然后可以调用makeWorld方法，这会返回一个充分填充的世界，为此会创建一个IsoWorld，循环处理_grid数组以及其中的各个单元。它会检查_tileTypes来查看要建立何种类型的区块，并使用其他定义作为参数来创建这种类型的区块，将其增加到世界中。一切工作都完成后，再返回这个世界。

使用这个类相当容易。下面是一个例子，可以下载MapTest.as文件得到：

```
package
{
    import com.friendsofed.isometric.IsoWorld;
    import com.friendsofed.isometric.MapLoader;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    [SWF(backgroundColor=0xffffff)]
    public class MapTest extends Sprite
    {
        private var _world:IsoWorld;
```

www.TopSage.com

```
        private var _floor:IsoWorld;
        private var mapLoader:MapLoader;

        [Embed(source="tile_01.png")]

        private var Tile02:Class;

        public function MapTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            mapLoader = new MapLoader();
            mapLoader.addEventListener(Event.COMPLETE, onMapComplete);
            mapLoader.loadMap("map.txt");
        }

        private function onMapComplete(event:Event):void
        {
            _world = mapLoader.makeWorld(20);
            _world.x = stage.stageWidth / 2;
            _world.y = 100;
            addChild(_world);
        }
    }
}
```

这里要创建MapLoader，监听其完成事件，并加载一个区块图。一切完成后，调用makeWorld，并将其添加到显示列表。图3-17显示了基于以上区块图文件运行这个类的结果。

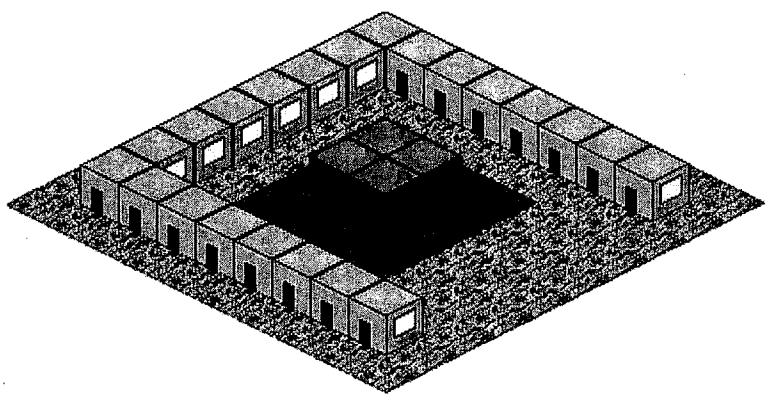


图3-17　MapLoader类完成的工作

这里有一点需要指出，即区块定义中的graphicSymbol属性：

```
# 0 type:GraphicTile graphicClass:MapTest_Tile01 xoffset:20 yoffset:10 walkable:true
```

这里设置为MapTest_Tile01，但是在MapTest类中，我们如下嵌入符号：

```
[Embed(source="tile_01.png")]
private var Tile01:Class;
```

www.TopSage.com

因此，你可能认为这里应该是graphicClass:Tile01。但是Tile01实际上是MapTest类的一个私有属性。在MapLoader类中不会看到这个私有属性。最终类中对应一个嵌入资源的真实名字是ClassName_PropertyName，其中ClassName是完成嵌入的类，而PropertyName是所嵌入资源链接的属性。由于Tile01是属性，而MapTest是嵌入这个资源的类，所以类本身是MapTest_Tile01。

现在可以编辑区块图文件，用各种不同类型的区块、颜色、高度、外部图形等创建各种不同的布局——所有这些都无需重新编译SWF。（当然，如果希望嵌入新的图形，则需要重新编译将其增加到SWF。）

利用这个类还可以做更多工作，我认为如果你确实开始使用这个类，可能会根据自己的个人需要做出调整。也许你不喜欢这种想法，即采用通用对象作为定义，而希望用更好的做法来取代。或者你可能希望增加一种方法来指定有些区块应当增加到floor，而另外一些区块要增加到world。不过无论怎样，我认为这都是一个很好的起点，可以作为一个框架，你能够在此基础上建立一个很酷的等角世界而不必一切从头开始。

3.11 小结

好了，我认为已经介绍了等角3D的所有基础知识，不过这里并没有详细讨论所有的细枝末节。很多细节都留给你来做出选择，还有很多内容有待优化。不过我认为你应该能做得很棒。

接下来，我们来讨论寻路，其中涵盖了在区块世界中实现移动的高级方法，这些对于基于区块的等角世界也完全适用。

www.TopSage.com

第4章

寻　路

4

寻路（pathfinding）一词顾名思义就是要寻找一条路径。假设你正处在A点，想要前往B点，怎样才能到达那里呢？这个主题已经得到游戏开发人员的广泛研究，我们介绍的所有内容都不能算是新内容，不过这里应该能涵盖这个主题的有关基础知识，我们还会给出采用ActionScript 3.0提供的标准解决方案高级实现。

4.1　寻路基础

寻路通常应用于一个基于区块的世界。即使应用于一个并非基于区块的游戏或世界，从寻路的角度来看，通常仍然可以将世界看作是某种网格。因此，寻路的结果就是一系列区块，它们构成了从起始区块到终止区块的一条不间断的路径。如果只是要在两个区块之间画一条线，那么即使对于一个基于区块的世界来说，这也将是一个极其简单的问题，根本不值得专门用一整章来讨论，在因特网上简单搜索“寻路”一词时所找到的页面也不会那么多。如果将某些区块设置为**不可通过**（unwalkable），复杂性就会提高，从字面就可以明确这个词的含义。游戏中的角色不能移动到不可通过的区块上，也不能穿过任何不可通过的区块，而是必须绕过它们。如果这些不可通过的区块形成起始点和终止点之间的一个障碍物，寻路则变成了一个新的问题，即如何绕过这个障碍物而仍能到达目标。图4-1显示了一个例子。

寻路通常还暗含一个思想，即需要寻找最佳路径。在许多实现中，最佳路径（best path）就是指最短路径（shortest path）——由最少数目的区块构成的路径。不过在寻路中，我们要使用另一个概念——最小代价（least cost）的路径。从任何区块移动到任何其他区块都要有一定的代价。当然，距离就是代价的一个重要方面——穿过两个区块就比只穿过一个区块“代价更大”——不过可能还有其他代价。例如，在某些游戏中，穿过一个沼泽地或者翻过一些大山的路径可能最短，但是爬山和越过沼泽难度更大，所以这条路径的代价可能要比沿着大路的路径代价更高（尽管后者可能距离更长）。来看一个我们更熟悉的例子，从办公室到你家的最短路径可能要经过市中心，但是如果你在上下班高峰期回家，可能更愿意走一条距离稍微长一点但是能避开交通拥堵区域的路线。

4.2　A*

即使你只是最粗略地研究寻路，也肯定遇到过A*（读作“A星”）这个词。A*是寻找从起始

www.TopSage.com

点到终点之间最佳路径的一种通用算法。它相当于寻路方面的行业标准，几乎用在每一个使用寻路的游戏中。如果实现正确，A*可以保证找到两点之间的最佳路径，而且它采用的搜索方法相对高效。出于这些原因，很多人认为寻路问题已经完全明确，其解决方案就是A*。对于各种适合采用A*算法的情况，最好花些时间来深入了解一下A*（正如本章所要做的），而不是尝试提出你自己的寻路算法。

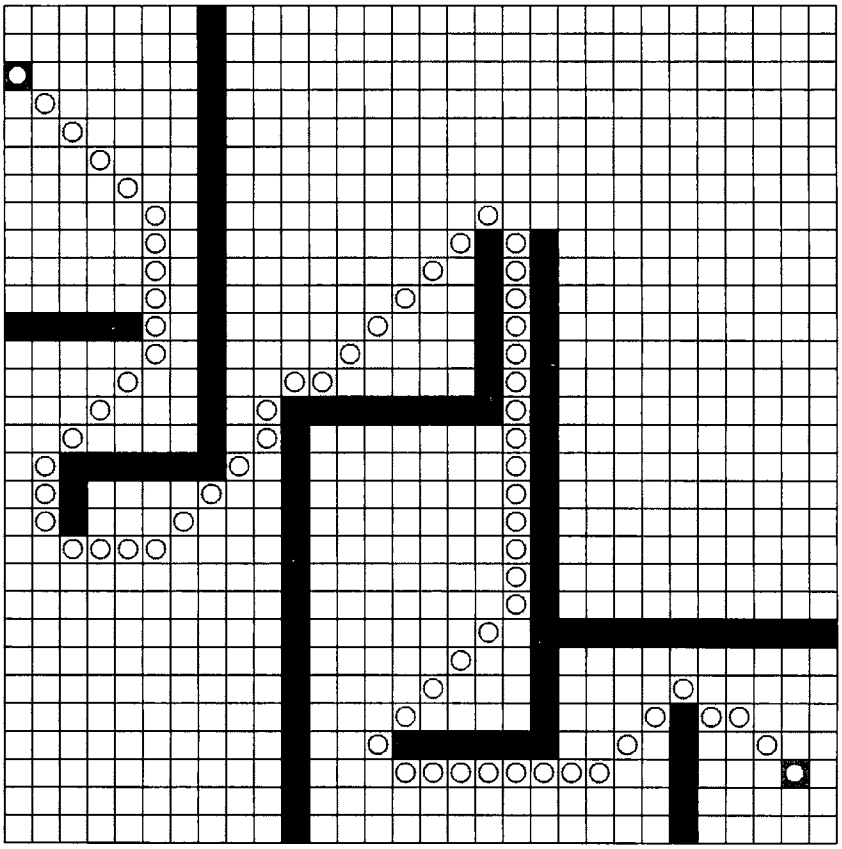


图4-1 一条有障碍物的路径

A*功能强大的一个方面体现在它是一个非常通用的算法，与其说是一个精确公式，不如说是解决寻路问题的一个概要方案。事实上，A*算法的一个重要部分是**启发函数**（heuristic），它本身是一个子算法，用于整个过程中的某一部分。启发函数并非由A*定义，而且存在多种常用的启发函数，相应地从速度和效率来看会有不同的结果。实际上，A*的几乎所有方面都可以根据语言特定的特性功能和应用需求进行调整和定制。

在本章中，我们将构建A*的一个ActionScript 3.0实现。这并不一定是最佳实现，不过这个实现足够清晰，我希望它能够便于你学习和了解这个主题的基本概念。无论如何，这个实现可以很好地用于大多数常见的应用，并允许你根据自己所需加以调整来达到最终优化。

www.TopSage.com

大家网 TopSage.com

4.2.1 A*基础

如果尽可能简洁地定义，可以认为A*包括选择一个起始区块，访问周围的各个区块，并为各个区块指定一个代价。稍后会更详细地讨论代价，不过起码要知道特定区块是否适宜起点到终点间的路径会对代价有影响。然后重复这个过程，将有最小代价的区块作为新的起始区块。如果继续以这种方式处理，你最终会到达终止区块，而且还能回溯至起始区块以得到二者之间的最佳路径。

4.2.2 A*算法

大多数讲授A*的方法总是首先提供A*算法的一个伪代码表示，然后再显示一些简单的图展示使用该算法的搜索过程。看起来这种做法很不错，所以我也照此来介绍。不过，首先要来定义一些术语。

- 节点：在基于区块的世界中这实际上是一个区块。不过，A*没有使用**区块**（tile）、**单元**（cell）或**点**（point），而是使用术语**节点**（node）来指定所检查的路径段。所以一个路径会包含起始节点、终止节点或目标节点，以及构成二者之间最佳路径的一个节点列表。
- 代价：这是根据各个节点对于路径的适宜度而为节点指定的等级值。较低代价的节点比较高代价的节点更可取。**代价**（cost）由两部分组成，从起始节点到达某个节点的代价，以及从该节点到达目标节点的估计代价。代价元素通常用变量*f*、*g*和*h*表示，下面将逐一对其进行介绍。
- *f*：一个特定节点的总代价，定义为$g+h$。
- *g*：从起始节点到达某个特定节点的代价。可以准确计算，因为你往往知道从起始节点到达该节点所选取的具体路径。
- *h*：从一个特定节点到达终止节点的估计代价。这个估计通过一个启发函数完成。这只是一个估计，因为你不知道要选取的具体路径。这正是你要确定的！
- 启发函数：估计从一个特定节点到达终止节点所需代价的函数。存在多种常用的启发函数，它们在速度、效率等方面会有不同的结果。
- 待查列表：已经访问并已指定代价的节点的列表。这个列表中的最低代价节点将用于下一次搜索迭代。
- 已查列表：所有相邻节点都已经访问的节点的列表。
- 父节点：迭代处理到一个节点时，其相邻的各个节点都会得到检查，并将该节点指定为父节点。所以到达目标节点时，可以沿着父节点链回溯到起始节点。由于父节点总是待查列表中的最低代价节点，所以肯定能得到最佳路径。

下面来看算法描述。

(1) 将起始节点添加到待查列表。

(2) 主搜索循环。

a. 寻找待查列表中有最小代价的节点，这就是当前节点。

b. 如果当前节点就是终止节点，则完成。转向步骤4。

c. 检查各个相邻节点（在一个矩形网格中，将有8个相邻节点）。对于每个相邻节点：

www.TopSage.com

i. 如果该节点不可通过，或者已经在待查列表或已查列表中，则将其跳过，继续处理下一个相邻节点，否则继续；

ii. 计算代价；

iii. 将当前节点设置为其父节点；

iv. 把它添加到待查列表中。

d. 将当前节点添加到已查列表。

(3) 针对更新后的待查列表重复执行步骤2。

(4) 已经到达终止节点。创建一个路径列表，并把终止节点添加到该列表。

(5) 将终止节点的父节点添加到路径列表。

(6) 将该节点的父节点添加到路径列表。重复这个过程直至到达起始节点。路径列表现在则包含构成从起始点到终止点之间最佳路径的所有节点。

稍后将通过图示解释这个算法，不过首先你需要了解如何计算一个区块的代价。

4.2.3 计算代价

根据定义，一个特定节点的代价使用公式$f = g + h$来计算，其中g是从起始节点到达该节点的代价，h是从该节点到达目标节点的估计代价。

代价公式的第一部分相当简单：就是从起始节点到达该节点所经过的节点个数。对现在来说，我们认为从任意节点到达其任意相邻节点的代价为1。

因此，从起始节点开始，检查其各个相邻节点，将为各相邻节点指定代价g=1（参见图4-2）。

在下一次迭代时，可以如下计算g：取当前节点的g，再加上从当前节点到达所检查节点的代价。换句话说，假设当前节点是起始节点周围的某个节点。因此该节点的g为1。查看这个节点周围的各个节点时，将为它的各个相邻节点指定g=2，因为移动到当前节点的代价为1，而从当前节点到达下一个节点又要花费另一个1。如图4-3所示。

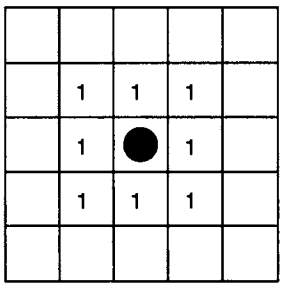


图4-2 从起始点指定g

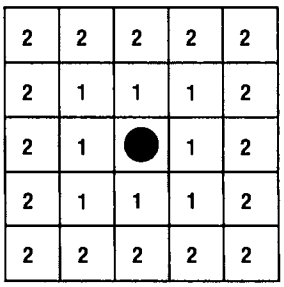


图4-3 为下一个节点指定g

不过，在大多数实现中，周围的节点并不完全相同。如果查看同一行或同一列上两个节点之间的距离，与对角相邻的两个节点相比，可以看到实际上对角节点相距较远。如果完成三角计算，会发现这个距离不是1，而是1.414，或者2的平方根。因此，可以在代价中考虑到这一因素。（现在不必对此太过担心，很快就会介绍有关内容。）图4-4显示了这个步骤。

www.TopSage.com

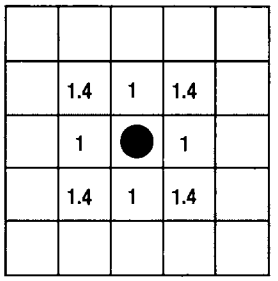


图4-4　对角线节点的代价大于水平或垂直节点的代价

代价的另一部分是从指定节点移动到目标节点的估计代价。确定这个代价要利用一个启发函数，启发函数也就是一个算法或公式。最简单的启发函数之一是取两个节点之间的距离（使用传统的勾股定理）。两个节点之间有多少行？有多少列？对这两个数求平方，将二者相加，再取其平方根。以下是相应的伪代码：

```
dx = distance between columns
dy = distance between rows
dist = sqrt(dx * dx + dy * dy)
```

这就是所求得的h。当前节点的代价就是f，即$g+h$。

再次指出，后面还会更详细地讨论这个内容，现在只要有一个一般性的认识就可以了。

4.2.4　算法图示

首先有一个网格，一个起始节点、一个终止节点和一些不可通过的区块（参见图4-5）。

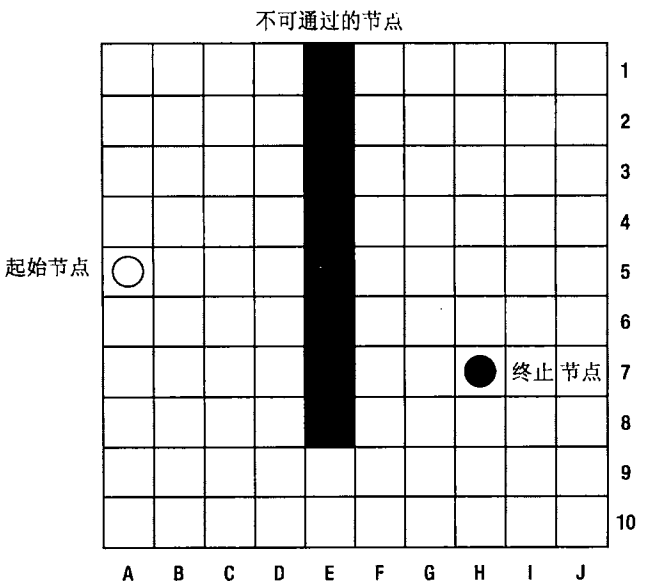


图4-5　准备寻找路径

www.TopSage.com

TopSage.com

继续执行这个算法，将起始节点添加到待查列表。然后查看这个起始列表以寻找最小代价节点。现在列表中只有一个节点，即起始节点，所以这就是我们要找的节点。

然后检查起始节点周围的各个节点，为之分别指定一个代价。首先为各个节点计算*g*。对于同一行或同一列上的节点，这个代价将是1，对于对角相邻的节点这个代价则为1.4。在这个练习中，代价不需要太过精确——1.4已经足够接近了。参见图4-6。

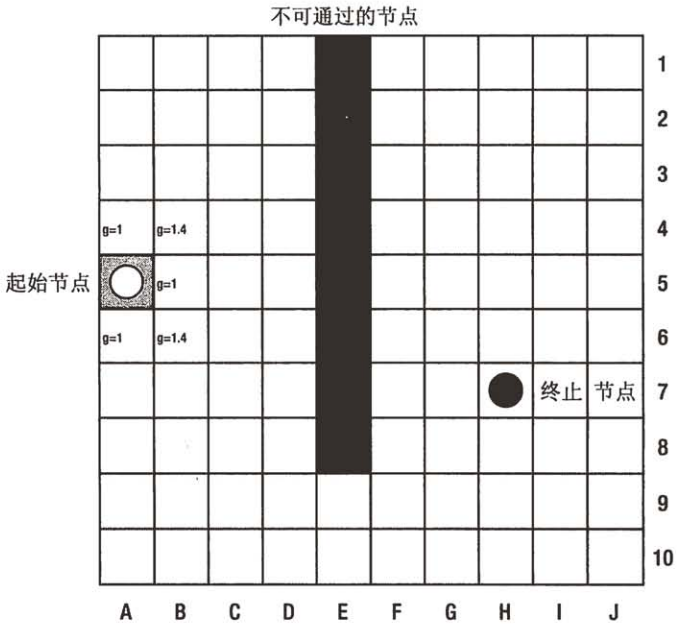


图4-6 指定了*g*

接下来指定*h*。使用直线启发函数，也就是从各节点到达目标节点的距离。这不是像素距离，而是使用勾股定理公式计算的两个节点之间的节点数（组合距离），如前所示。

确定两部分代价后，就可以将二者相加得到总代价。图4-7给出了结果。

现在可以将当前节点添加到已查列表，并将已访问的各个节点添加到待查列表。

然后再来查看待查列表，找出其中有最小代价的节点，根据这个网格，现在最小代价节点就是节点B5，再重复这个过程。不过，这一次它周围的某些节点已经出现在待查列表或已查列表中，所以可以将它们忽略。计算其余节点的代价（参见图4-8）。

现在将B5添加到已查列表中，而且将刚才检查过的节点添加到待查列表。再次从待查列表中选择有最小代价的节点。这一次是C5。计算其外围节点的代价（结果如图4-9所示）。

现在，注意到D5和D6有相同的代价。实际上，为了力求简单，这里对值采取了四舍五入，因此它们的代价可能相同也可能不相同，不过确实也有这种可能性，所以假设两个节点的代价相同。你会选择其中哪一个节点呢？这并不重要。实际上，这并不取决于你，而往往取决于某种排序算法。现在可以很明显地看出，D6就是下一个要找的节点，因为可以看到这里存

www.TopSage.com

在一面墙。不过，启发函数无法看到这面墙。它只会计算距离。所以在它看来，这两个节点都是有效的。

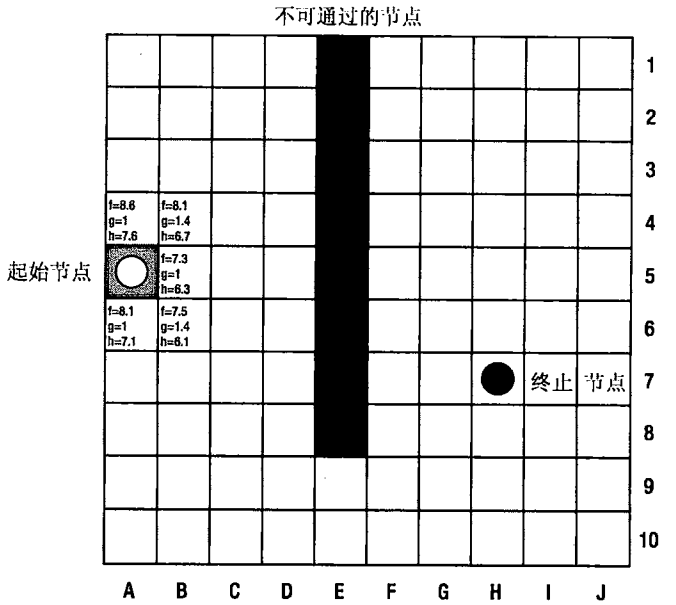


图4-7　各节点的总代价

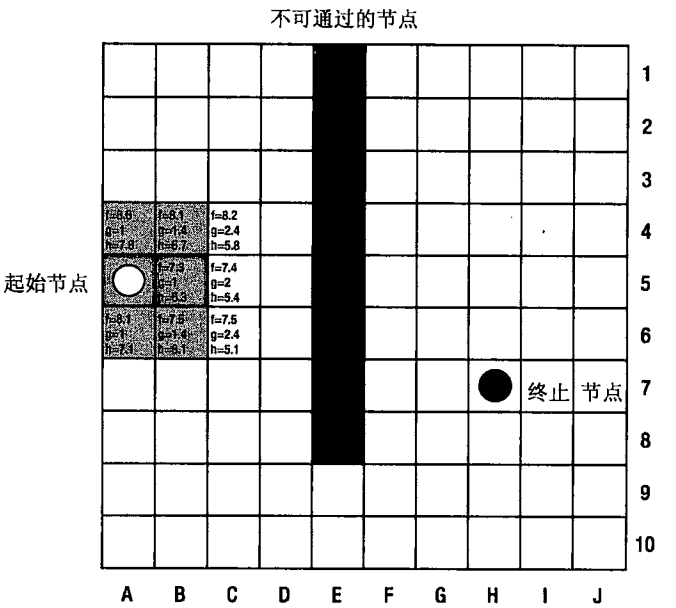


图4-8　第二轮寻路

www.TopSage.com

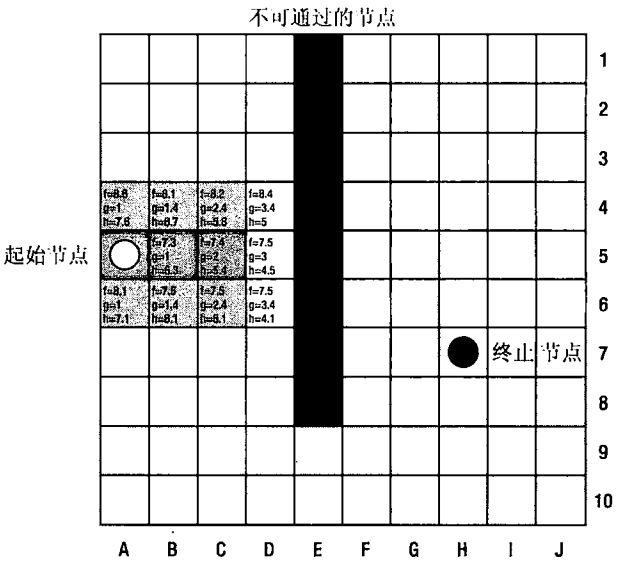


图4-9 第3轮寻路

不论怎样，假设我们通过掷硬币或者对待查列表完成一个排序，得出了D5。然后查看它周围的节点，但是它周围没有有效节点。这些节点都是不可通过的，或者已经出现在待查列表或已查列表中。没问题，现在执行到算法的步骤2c。如果周围没有有效的节点，则转向步骤3，将节点D5添加到已查列表，再次检查待查列表以寻找有最小代价的节点。这一次无可争议地会选出D6。继续执行算法，检查其相邻节点。这会得到如图4-10所示的结果。

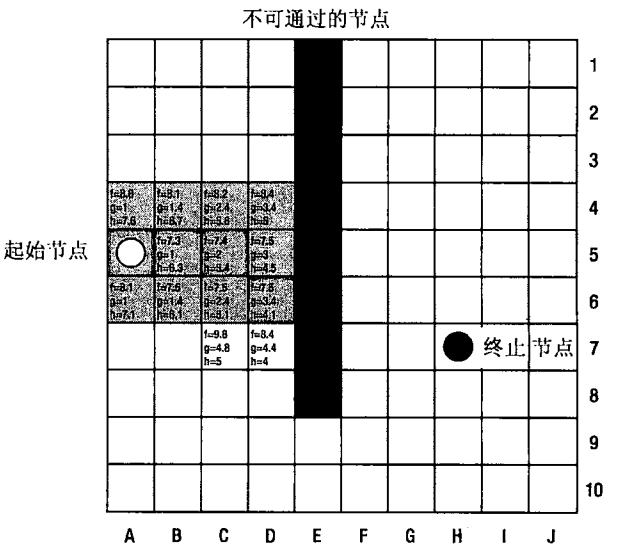


图4-10 继续寻路

www.TopSage.com

大家网 TopSage.com

继续这个过程，找出待查列表中有最小代价的节点，计算其各个相邻节点的代价，并添加到待查列表，将当前节点放在已查列表中，找出待查列表中有最小代价的节点，如此继续。最终会得到如图4-11所示的结果。

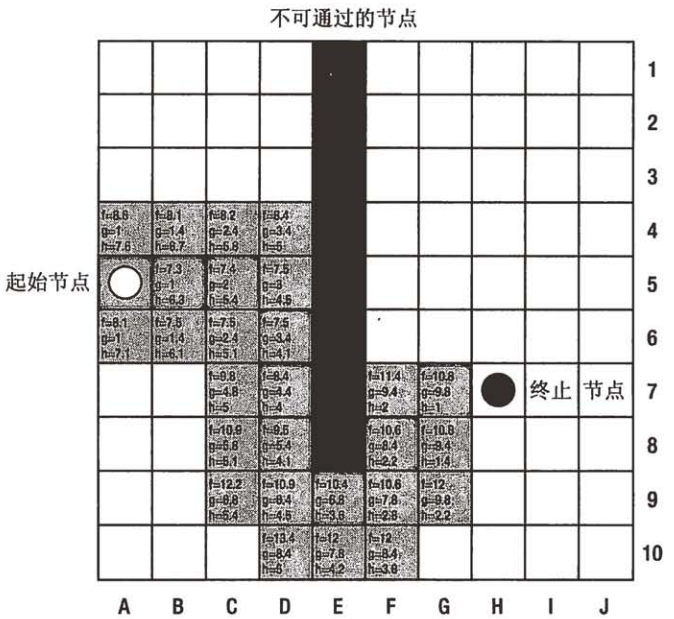


图4-11　接近目标

当前节点是G7。开始查看外围节点时，很快你会发现其中一个节点就是终止节点。大功告成！

在每个阶段中，当查看新节点时，会将各节点上的一个父属性设置为等于当前节点。因此，现在可以从终止节点回退，沿路返回到起始节点。将这个路径逆置，就得到了从起始点到终点的最佳路径。

4.2.5　编写代码

要编写A*算法的代码，首先需要一些结构表示网格和单个节点。

先来看Grid类。与所有其他代码一样，可以从本书网站www.friendsofed.com下载这个类（相应文件是Grid.as）。

```
package
{
    /**
     * Holds a two-dimensional array of Nodes,
     * methods to manipulate them, a start node and an end node.
     */

public class Grid
```

www.TopSage.com

```
{
    private var _startNode:Node;
    private var _endNode:Node;
    private var _nodes:Array;
    private var _numCols:int;
    private var _numRows:int;

    /**
     * Constructor.
     */
    public function Grid(numCols:int, numRows:int)
    {
        _numCols = numCols;
        _numRows = numRows;
        _nodes = new Array();

        for(var i:int = 0; i < _numCols; i++)
        {
            _nodes[i] = new Array();
            for(var j:int = 0; j < _numRows; j++)
            {
                _nodes[i][j] = new Node(i, j);
            }
        }
    }


    ////////////////////////////////////
    // public methods
    ////////////////////////////////////

    /**
     * Returns the node at the given coords.
     * @param x The x coord.
     * @param y The y coord.
     */
    public function getNode(x:int, y:int):Node
    {
        return _nodes[x][y] as Node;
    }

    /**
     * Sets the node at the given coords as the end node.
     * @param x The x coord.
     * @param y The y coord.
     */
public function setEndNode(x:int, y:int):void
{
    _endNode = _nodes[x][y] as Node;
}

/**
 * Sets the node at the given coords as the start node.
 * @param x The x coord.
```

www.TopSage.com

```
     * @param y The y coord.
     */
    public function setStartNode(x:int, y:int):void
    {
        _startNode = _nodes[x][y] as Node;
    }

    /**
     * Sets the node at the given coords as walkable or not.
     * @param x The x coord.
     * @param y The y coord.
     */
    public function setWalkable(x:int, y:int, value:Boolean):void
    {
        _nodes[x][y].walkable = value;
    }



    ////////////////////////////////////////
    // getters / setters
    ////////////////////////////////////////

    /**
     * Returns the end node.
     */
    public function get endNode():Node
    {
        return _endNode;
    }

    /**
     * Returns the number of columns in the grid.
     */
    public function get numCols():int
    {
        return _numCols;
    }

    /**
     * Returns the number of rows in the grid.
     */
            public function get numRows():int
            {
                return _numRows;
            }

            /**
             * Returns the start node.
             */
            public function get startNode():Node
            {
                return _startNode;
            }
```

www.TopSage.com

TopSage.com

```
    }
}
```

在构造函数中传入所希望的网格行数和列数，这会创建一个节点数组。接下来查看Node类。可以设置起始节点或终止节点，分别指定所希望的*x*和*y*坐标。类似地，可以将任何特定节点设置为可通过或不可通过。最后，可以得到起始节点、终止节点或任何特定节点的引用，还可以读取网格中的行数和列数。

需要注意，Grid类只是一个包含网格有关信息的对象——它没有可视化表示。后面将针对可视化表示创建另一个类，不过首先来看Node类（见Node.as文件）：

```
package
{
    /**
     * Represents a specific node evaluated
     * as part of a pathfinding algorithm.
     */
    public class Node
    {
        public var x:int;
        public var y:int;
        public var f:Number;
        public var g:Number;
        public var h:Number;
        public var walkable:Boolean = true;
        public var parent:Node;

        public function Node(x:int, y:int)
        {
            this.x = x;
            this.y = y;
        }
    }
}
```

Node类是一个用来存储节点属性的简单数据对象。它本身没有任何行为，所以只提供了公共属性。注意，目前来讲，所有节点都相等，因此所涉及的唯一代价就是路径的长度。后面你会看到，不同类型的节点可以有不同的内部代价。

下面还需要一个完成寻路工作的类。这就是AStar类，可以通过下载相应的AStar.as文件得到。下面完整地列出这个类，然后再详细地分析：

```
package
{
    public class AStar
    {
        private var _open:Array;
        private var _closed:Array;
        private var _grid:Grid;
        private var _endNode:Node;
        private var _startNode:Node;
        private var _path:Array;
//      private var _heuristic:Function = manhattan;
```

www.TopSage.com

```
//      private var _heuristic:Function = euclidian;
        private var _heuristic:Function = diagonal;
        private var _straightCost:Number = 1.0;
        private var _diagCost:Number = Math.SQRT2;


        public function AStar()
        {
        }

        public function findPath(grid:Grid):Boolean
        {
            _grid = grid;
            _open = new Array();
            _closed = new Array();

            _startNode = _grid.startNode;
            _endNode = _grid.endNode;

            _startNode.g = 0;
            _startNode.h = _heuristic(_startNode);
            _startNode.f = _startNode.g + _startNode.h;

            return search();
        }

        public function search():Boolean
        {
            var node:Node = _startNode;
            while(node != _endNode)
        {
            var startX:int = Math.max(0, node.x - 1);
            var endX:int = Math.min(_grid.numCols - 1, node.x + 1);
            var startY:int = Math.max(0, node.y - 1);
            var endY:int = Math.min(_grid.numRows - 1, node.y + 1);

            for(var i:int = startX; i <= endX; i++)
            {
                for(var j:int = startY; j <= endY; j++)
                {
                    var test:Node = _grid.getNode(i, j);
                    if(test == node || !test.walkable) continue;

                    var cost:Number = _straightCost;
                    if(!((node.x == test.x) || (node.y == test.y)))
                    {
                        cost = _diagCost;
                    }
                    var g:Number = node.g + cost;
                    var h:Number = _heuristic(test);
                    var f:Number = g + h;
                    if(isOpen(test) || isClosed(test))
                    {
                        if(test.f > f)
```

www.TopSage.com

```
                {
                    test.f = f;
                    test.g = g;
                    test.h = h;
                    test.parent = node;
                }
            }
            else
            {
                test.f = f;
                test.g = g;
                test.h = h;
                test.parent = node;
                _open.push(test);
            }
        }
    }
    _closed.push(node);
    if(_open.length == 0)
    {
        trace("no path found");
        return false
    }
    _open.sortOn("f", Array.NUMERIC);
    node = _open.shift() as Node;
}
    buildPath();
    return true;
}

private function buildPath():void
{
    _path = new Array();
    var node:Node = _endNode;
    _path.push(node);
    while(node != _startNode)
    {
        node = node.parent;
        _path.unshift(node);
    }
}

public function get path():Array
{
    return _path;
}

private function isOpen(node:Node):Boolean
{
    for(var i:int = 0; i < _open.length; i++)
    {
        if(_open[i] == node)
        {
            return true;
```

www.TopSage.com

大家网 TopSage.com

```
                }
            }
            return false;
        }

        private function isClosed(node:Node):Boolean
        {
            for(var i:int = 0; i < _closed.length; i++)
            {
                if(_closed[i] == node)
                {
                    return true;
                }
            }
            return false;
        }
        private function manhattan(node:Node):Number
        {
            return Math.abs(node.x - _endNode.x) * _straightCost +
                   Math.abs(node.y + _endNode.y) * _straightCost;
        }

        private function euclidian(node:Node):Number
        {
            var dx:Number = node.x - _endNode.x;
            var dy:Number = node.y - _endNode.y;
            return Math.sqrt(dx * dx + dy * dy) * _straightCost;
        }

        private function diagonal(node:Node):Number
        {
            var dx:Number = Math.abs(node.x - _endNode.x);
            var dy:Number = Math.abs(node.y - _endNode.y);
            var diag:Number = Math.min(dx, dy);
            var straight:Number = dx + dy;
            return _diagCost * diag + _straightCost * (straight - 2 * diag);
        }

        public function get visited():Array
        {
            return _closed.concat(_open);
        }
    }
}
```

首先，定义一些属性，然后定义一个默认构造函数：

```
private var _open:Array;
private var _closed:Array;
private var _grid:Grid;
private var _endNode:Node;
private var _startNode:Node;
private var _path:Array;
//      private var _heuristic:Function = manhattan;
```

www.TopSage.com

```
//      private var _heuristic:Function = euclidian;
private var _heuristic:Function = diagonal;
private var _straightCost:Number = 1.0;
private var _diagCost:Number = Math.SQRT2;

public function AStar()
{
}
```

这里是待查列表和已查列表的相应数组、一个网格、一个开始节点和一个终止节点、包含最终路径的数组（这将是一个节点列表）以及一个启发函数属性。在这个类中我提供了几种常用的启发函数，后面将更详细地解释这些内容。你可以选择其中的任何启发函数，只需去除所用启发函数的注释，而注释掉其他启发函数。你可能想设计一种更稳健的方法，能够切换不同的启发函数（如使用一个setHeuristic方法），不过这个工作将留给你来完成。最后，这里还有一些属性，对应从一个节点沿直线和沿对角线移动所需的代价。

下面给出findPath方法：

```
public function findPath(grid:Grid):Boolean
{
    _grid = grid;
    _open = new Array();
    _closed = new Array();

    _startNode = _grid.startNode;
    _endNode = _grid.endNode;

    _startNode.g = 0;
    _startNode.h = _heuristic(_startNode);
    _startNode.f = _startNode.g + _startNode.h;

    return search();
}
```

这个方法首先进行初始化，创建一个空的待查列表和已查列表，然后从所传入的网格获得起始节点和终止节点。它会计算起始节点的代价，然后运行search方法，这个方法将迭代处理直到终止节点，并返回路径。

下面来看这个方法如何计算起始节点的代价。它首先设置起始节点的g为0，因为g定义为从起始节点到当前节点的代价。由于起始节点和当前节点是同一个节点，因此这里没有代价。然后调用所选择的启发方法并传入起始节点，将会返回从那里到达终止节点的估计代价，这就是h。最后，将g与h相加得到f，即该节点的总代价。

以下是这个类的核心部分，search方法。这会从起始节点逐步处理一直到最终到达终止节点，在此过程中计算出最佳路径：

```
public function search():Boolean
{
    var node:Node = _startNode;
    while(node != _endNode)
    {
        var startX:int = Math.max(0, node.x - 1);
```

www.TopSage.com

```
            var endX:int = Math.min(_grid.numCols - 1, node.x + 1);
            var startY:int = Math.max(0, node.y - 1);
            var endY:int = Math.min(_grid.numRows - 1, node.y + 1);
            for(var i:int = startX; i <= endX; i++)
            {
                for(var j:int = startY; j <= endY; j++)
                {
                    var test:Node = _grid.getNode(i, j);
                    if(test == node || !test.walkable) continue;
                    var cost:Number = _straightCost;
                    if(!((node.x == test.x) || (node.y == test.y)))
                    {
                        cost = _diagCost;
                    }
                    var g:Number = node.g + cost;
                    var h:Number = _heuristic(test);
                    var f:Number = g + h;
                    if(isOpen(test) || isClosed(test))
                    {
                        if(test.f > f)
                        {
                            test.f = f;
                            test.g = g;
                            test.h = h;
                            test.parent = node;
                        }
                    }
                    else
                    {
                        test.f = f;
                        test.g = g;
                        test.h = h;
                        test.parent = node;
                        _open.push(test);
                    }
                }
            }
            _closed.push(node);
            if(_open.length == 0)
            {
                trace("no path found");
                return false
            }
            _open.sortOn("f", Array.NUMERIC);
            node = _open.shift() as Node;
        }
        buildPath();
        return true;
    }
```

首先建立一个节点变量，用来跟踪当前节点，这个当前节点最初是起始节点。然后运行一个while循环，只有当前节点等于终止节点时才退出这个循环。如果退出循环说明寻路结束。

在这个while循环中，首先要运行一个双重for循环，检查当前节点周围的所有节点，如图4-12

www.TopSage.com

TopSage.com

所示。

首先得到当前节点的x和y值，并从x-1循环到x+1以及y-1循环到y+1（要记住，这里x和y值是表示行与列的整数值，而不是像素屏幕位置，现在还没有在屏幕上显示任何内容）：

```
var startX:int = Math.max(0, node.x - 1);
var endX:int = Math.min(_grid.numCols - 1, node.x + 1);
var startY:int = Math.max(0, node.y - 1);
var endY:int = Math.min(_grid.numRows - 1, node.y + 1);
for(var i:int = startX; i <= endX; i++)
{
    for(var j:int = startY; j <= endY; j++)
    {
```

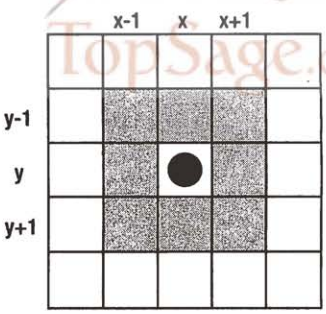


图4-12 当前节点外围节点的x, y索引值

还必须确保不会试图访问边界（或网格上下界）以外的区块，如图4-7所示。为此通过Math.min和Math.max计算索引，从而保证索引不会小于0或者大于网格最后一行或最后一列，而是永远在网格里面。

对于各个节点，可以从网格得到节点本身。如果测试节点就是当前节点，或者测试节点是不可通过的，则直接将其忽略，移向下一个节点：

```
var test:Node = _grid.getNode(i, j);
if(test == node || !test.walkable) continue;
```

如果执行到这里，说明确实有一个合法的测试节点，需要确定其代价。首先我们计算从起始节点到这个测试节点的代价（*g*代价）。为此可以取当前节点的*g*，并加上从当前节点到测试节点的代价。一种简单的方法是：如果该节点与当前节点在水平线或垂直线上，则指定_straightCost，如果节点在对角线上，则指定_diagCost。然后将当前节点的*g*与之相加来得到总的*g*代价。要得到*h*需要在测试节点上运行启发函数，只需将*g*和*h*相加就可以得到*f*：

```
var cost:Number = _straightCost;
if(!((node.x == test.x) || (node.y == test.y)))
{
    cost = _diagCost;
}
var g:Number = node.g + cost;
var h:Number = _heuristic(test);
var f:Number = g + h;
```

下面一部分稍有些麻烦，这个问题此前还没有谈到过。之前我曾指出如果一个节点已经在待查列表或已查列表中，就不需要再检查了，因为它已经被处理过。不过，这一次到达一个节点所取的路径可能会得到一个新的代价，这个代价要小于第一次为它计算的代价（例如，之前可能根据一个对角线计算代价，但是现在则紧邻水平或垂直线上的一个节点，所以这个代价会更低）。因此，即使一个节点已经在待查列表或已查列表中，最好还是将当前代价与上一次计算得到的代价进行比较。为此，可以比较测试节点的*f*与刚计算的*f*。如果之前的*f*较大，则说明这一次找到一条更好的路径，相应地要替换测试节点的*f*、*g*和*h*。还要把测试节点的父节点设置为当前节点。这样一来，一旦到达目标节点我们能够又回退到起始节点：

```
if(isOpen(test) || isClosed(test))
{
    if(test.f > f)
    {
        test.f = f;
        test.g = g;
        test.h = h;
        test.parent = node;
    }
}
```

如果测试节点不在待查列表或已查列表中，只需直接指定*f*、*g*、*h*和父节点。还要将测试节点放入待查列表，因为现在它已经有一个经过充分计算的代价，并且需要在下一次测试最佳节点时加以考虑：

```
else
{
    test.f = f;
    test.g = g;
    test.h = h;
    test.parent = node;
    _open.push(test);
}
```

目前，我们已经检查了当前节点周围的所有合法节点。不再需要对当前节点做任何处理，所以把它放入已查列表：

```
_closed.push(node);
```

然后需要查找下一个当前节点从而重复这个过程。为此要检查待查列表，并找到其中代价最小的节点。不过，首先应当检查待查列表里是否有节点，很有可能此时待查列表为空。这说明起始节点和终止节点之间不可能有路径：

```
if(_open.length == 0)
{
    trace("no path found");
    return false
}
```

在这个实现中，如果找到一条路径，则返回true，如果不可能有路径则返回false。可以通过搜索路径来确定结果的代码进行检查。

如果待查列表中还有节点，则需要找到其中代价最小的一个节点（也就是说，*f*最小的节点）。为此可以按各个元素的f属性对这个列表排序，取列表中最下面的元素：

```
_open.sortOn("f", Array.NUMERIC);
node = _open.shift() as Node;
```

这一轮迭代到此结束，现在我们有了一个新的当前节点。while循环现在再次检查当前节点是否是终止节点。如果不是，则反复地执行这个过程，直到确定不可能找到路径，或者已经到达终止节点。

确实找到终止节点时，调用buildPath方法，并返回true。下面来看buildPath方法：

www.TopSage.com

```
private function buildPath():void
{
    _path = new Array();
    var node:Node = _endNode;
    _path.push(node);
    while(node != _startNode)
    {
        node = node.parent;
        _path.unshift(node);
    }
}
```

这个代码为路径创建一个新数组，并把终止节点放入这个数组。然后调用unshift，将终止节点的父节点放入列表。通过使用unshift而不是push，我们可以将各个新节点添加到数组的开始位置，这样一来，可以保证在完成时数组中元素的顺序将是从开始节点到终止节点。重复这个过程，添加各个父节点本身的父节点，直至到达起始节点。此时，路径数组包含了从起始节点到终止节点的最佳路径，大功告成！使用AStar类的代码现在可以检查findPath的返回值，从而来看是否找到了路径。如果成功，则用get path存取方法获取这个路径数组。

如果对AStar类的内部原理已经有了足够的了解，则可以直接跳至实现部分。不过，如果你希望了解更多，下面来学习这里包含的各种启发式方法。

4.2.6 常用A*启发函数

关于A*有意思的一点是，不论你使用何种启发函数，它都会给出其中一条最优路径。注意，这里我说的是其中一条最优路径，而不是唯一的一条最优路径。几乎在所有情况下，都会有多条代价相同的路径。例如，图4-13显示了同样两个节点之间的3条路径。每条路径的完整代价都为4.8（两次对角移动代价分别为1.4，再加上两次水平移动，代价分别是1.0）。

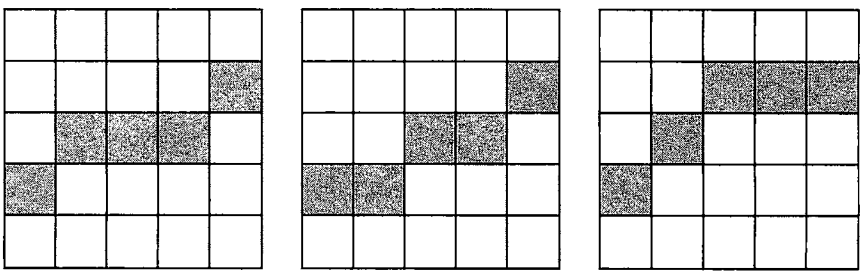


图4-13 3条路径，同一代价

只要解决方案提出其中某一条路径（或者其他代价为4.8的路径），都算一条有效的最佳路径。

因此，使用一个“更好的”启发函数并不意味着你能得到一条更短的路径。不过，有些启发函数会比另外一些启发函数的运行速度更快。最重要的一个因素在于，有些启发函数会导致检查较多节点，而有些则允许检查较少的节点。需要检查的节点越少，到达终止节点的速度就越快，需要完成排序的数组就越少（也越小），等等。

www.TopSage.com

另外，有些启发函数往往会创建在我们眼里看起来“更直”的路径。例如，大多数人都会认为图4-13中所示的第一条或第二条路径要更短或更直接，尽管这3条路径都有相同的代价。大多数人会说第3条路径看起来好像不太好。所以，如果一个启发函数导致对角移动和直线移动夹杂出现，与对角移动和直线移动分组先后出现相比，前者可以得到一条看起来更自然的路径。

要讨论不同的启发函数以及它们之间的区别，需要一整章甚至整本书的内容才能做到。这里只介绍3种常用的启发函数（Astar类中包含的启发函数），并对它们分别做简要的解释。不过，如果想了解更多的内容，可以在网上搜索A*启发函数，你会得到大量相关材料。参考图4-14来了解各个启发函数的原理。

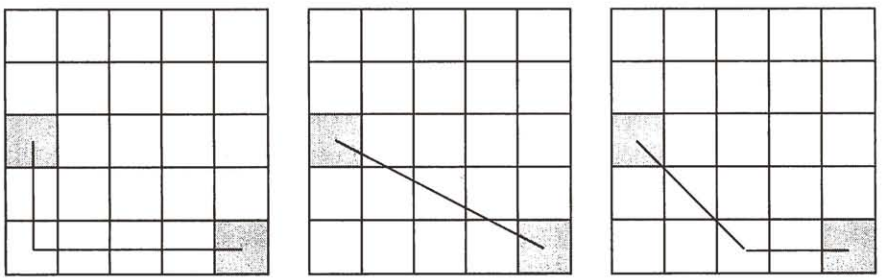


图4-14　3种常用启发函数

首先是所谓的**曼哈顿启发函数**（Manhattan heuristic），这种算法会忽略所有对角移动，只添加测试节点与终止节点之间的总列数和行数。其思想是，如果你处在一个类似街区的网格中（如曼哈顿街区），需要从（5,40）到达（8,43），则必须在一个方向上走3个街区，再沿另一个方向走3个街区。可以先走完所有横向街道，再走完所有纵向街道；或者反之，先走完所有纵向街道，再走完所有横向街道；也可以横向、纵向，再横向、纵向，横向、纵向，采用哪种方式并不重要。总之每个方向都要走3个街区。

```
private function manhattan(node:Node):Number
{
    return Math.abs(node.x - _endNode.x) * _straightCost +
           Math.abs(node.y + _endNode.y) * _straightCost;
}
```

所以这种启发函数只是得到两个节点之间列的差和行的差，再把二者相加。相当简单。

下一种常用方法有时称为**欧几里得几何启发函数**（Euclidian heuristic）。这种方法就是计算从测试节点到终止节点的一条直线，并返回其长度。以下是经典的勾股定理：$A^2 + B^2 = C^2$。

```
private function euclidian(node:Node):Number
{
    var dx:Number = node.x - _endNode.x;
    var dy:Number = node.y - _endNode.y;
    return Math.sqrt(dx * dx + dy * dy) * _straightCost;
}
```

对行上的距离，取其平方，再增加列上的距离（也取平方），再求二者相加之和的平方根。

www.TopSage.com

也相当简单。

最后一种方法是**对角启发函数**（Diagonal heuristic），看起来相当复杂，不过实际上它只是计算一条如图4-13所示的路径。

```
private function diagonal(node:Node):Number
{
    var dx:Number = Math.abs(node.x - _endNode.x);
    var dy:Number = Math.abs(node.y - _endNode.y);
    var diag:Number = Math.min(dx, dy);
    var straight:Number = dx + dy;
    return _diagCost * diag + _straightCost * (straight - 2 * diag);
}
```

这个启发函数是三者当中最精确的一个，它会返回两个节点之间的具体代价（如果二者之间没有障碍物）。实际上，如果对图4-13所示的图运行这个启发函数，可以看到它会返回代价为4.8，准确无误。

下面3个图显示了这3种算法的结果。注意，所有这3个算法都会得到一条包含23个水平移动和25个对角移动的路径，相应的总代价为58。不同的启发函数对于路径的长度没有任何影响。不过，对于不同方法，路径的特点和所访问节点（在网格中用灰色显示）的个数会有很大差别。可以看到，曼哈顿启发函数（参见图4-15）在找到终止节点之前会访问网格中几乎每一个节点。其路径也很不自然，一次完成所有水平移动，然后再完成所有对角移动。

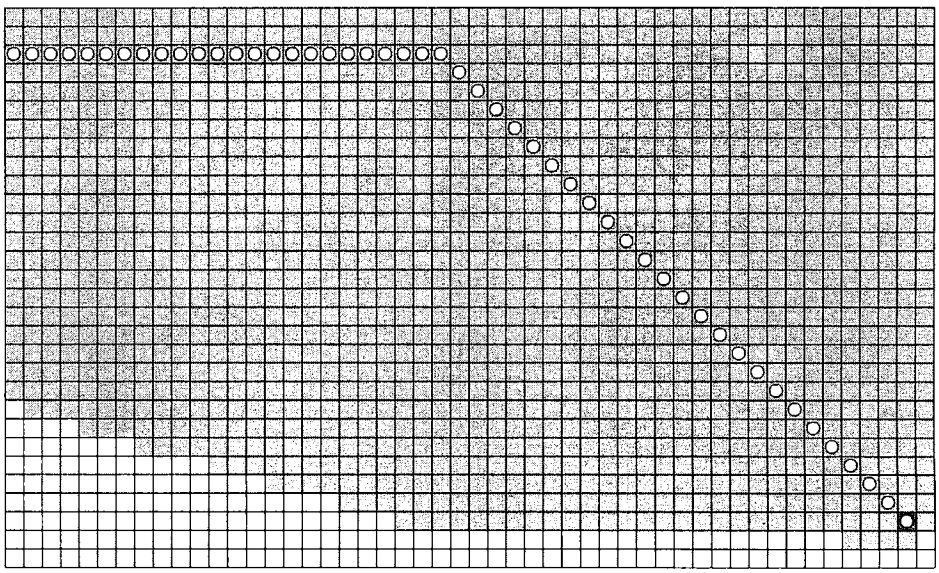


图4-15 曼哈顿启发函数的结果

下一个是欧几里得几何启发函数（参见图4-16），可以得到一个看上去稍微自然一些的路径，而且访问的节点也要少得多。

www.TopSage.com

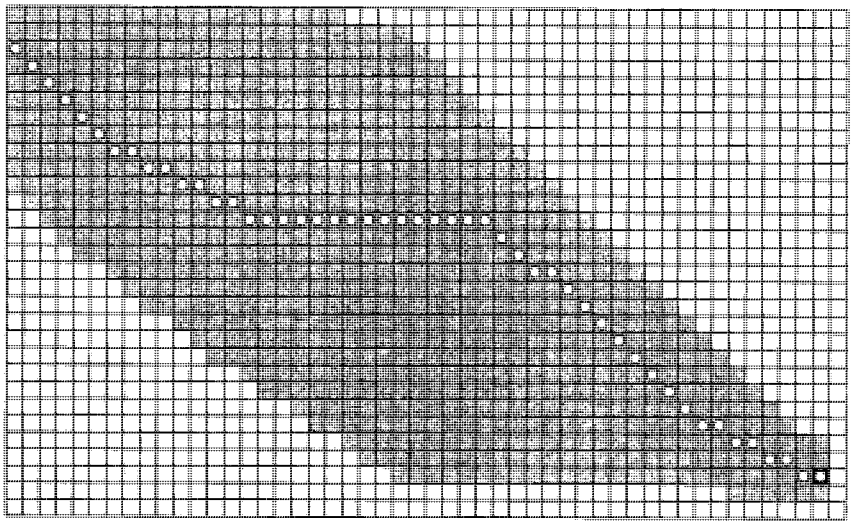


图4-16　欧几里得几何启发函数的结果

对角启发函数（参见图4-17）是三者当中效率最高的，访问的不必要的节点极少。尽管其路径没有欧几里得几何启发函数得到的路径看上去那么自然，但是比曼哈顿方法的路径要好。

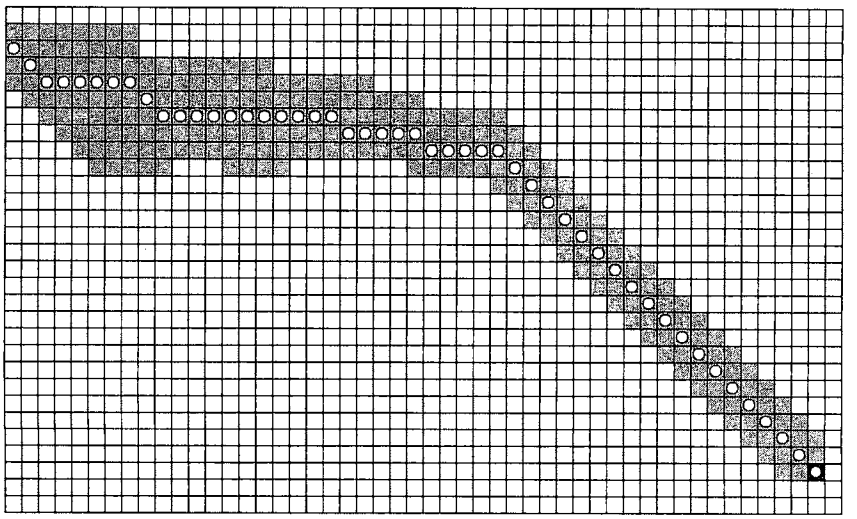


图4-17　对角启发函数的结果

当然，这里也要有所权衡（万事万物都是如此）。尽管曼哈顿启发函数最简单，而且在每个节点上的执行也最快，但是它会访问较多节点。对角启发函数比较复杂，但是运行次数要少得多。因此可以看到，这里没有严格意义上的“好”或“不好”的启发函数。每种方法都有其优点，而且除此以外还有很多其他的启发函数，另外针对我前面给出的几种算法还存在很多变化形式。要

www.TopSage.com

找到能够为特定应用提供最佳结果的方法是一个反复试验的过程。另外，前面的例子没有显示任何障碍物，一旦开始在网格中增加障碍物，问题会变得更加复杂。

以上已经介绍了所有基础知识，下面来看具体实现。

4.3 实现 AStar 类

在一个真正的游戏或应用中实现A*时，可能会创建一个基于区块的世界，其中包含明显的可以通过和不能通过的区块。起始点可能是一个角色所在的区块，终止区块可能明确指出，也可能并未指出。这可能是用户点击的一个点，也可能是一罐金子、一个敌人或者另外某个对象所在的区块。路径本身也许并不可见，而只是用于将角色沿路移动到目标。为便于说明，这里将创建一个GridView类，让路径呈现在你眼前（可以下载Gridview.as文件得到这个类）：

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

    /**
     * Serves as a visual representation of a grid of nodes
     * used in a pathfinding solution.
     */
    public class GridView extends Sprite
    {
        private var _cellSize:int = 20;
        private var _grid:Grid;

        /**
         * Constructor.
         */
        public function GridView(grid:Grid)
        {
            _grid = grid;
            drawGrid();
            findPath();
            addEventListener(MouseEvent.CLICK, onGridClick);
        }

        /**
         * Draws the given grid, coloring each cell based on its state.
         */
        public function drawGrid():void
        {
            graphics.clear();
            for(var i:int = 0; i < _grid.numCols; i++)
            {
                for(var j:int = 0; j < _grid.numRows; j++)
                {
                    var node:Node = _grid.getNode(i, j);
                    graphics.lineStyle(0);
                    graphics.beginFill(getColor(node));
```

www.TopSage.com

```
                graphics.drawRect(i * _cellSize, j * _cellSize, _cellSize, _cellSize);
            }
        }
    }

    /**
     * Determines the color of a given node based on its state.
     */
    private function getColor(node:Node):uint
    {
        if(!node.walkable) return 0x000000;
        if(node == _grid.startNode) return 0x666666;
        if(node == _grid.endNode) return 0x666666;
        return 0xffffff;
    }

    /**
     * Handles the click event on the GridView.
     * Finds the clicked on cell and toggles its walkable state.
     */
    private function onGridClick(event:MouseEvent):void
    {
        // Convert the mouse coordinates into a grid position
        var xpos:int = Math.floor(event.localX / _cellSize);
        var ypos:int = Math.floor(event.localY / _cellSize);

        //and toggle its walkability
        _grid.setWalkable(xpos, ypos, !_grid.getNode(xpos, ypos).walkable);
        drawGrid();
        findPath();
    }

    /**
     * Creates an instance of AStar and uses it to find a path.
     */
    private function findPath():void
    {
        var astar:AStar = new AStar();
        if(astar.findPath(_grid))
        {
            showVisited(astar);
            showPath(astar);
        }

    }

    /**
     * Highlights all nodes that have been visited.
     */
    private function showVisited(astar:AStar):void
    {
        var visited:Array = astar.visited;
        for(var i:int = 0; i < visited.length; i++)
```

www.TopSage.com

```
            {
                graphics.beginFill(0xcccccc);
                graphics.drawRect(visited[i].x * _cellSize, visited[i].y * _cellSize,
                                  _cellSize, _cellSize);
                graphics.endFill();
            }
        }

        /**
         * Highlights the found path.
         */
        private function showPath(astar:AStar):void
        {
            var path:Array = astar.path;
            for(var i:int = 0; i < path.length; i++)
            {
                graphics.lineStyle(0);
                graphics.beginFill(0xffffff

                // draw a circle with its center in the middle of the
                // cell with a radius that's a third of the cell width

                graphics.drawCircle(path[i].x * _cellSize + _cellSize / 2,
                                    path[i].y * _cellSize + _cellSize / 2, _cellSize / 3);
            }
        }
    }
}
```

GridView类的构造函数接受一个Grid实例，其中包含所有节点的列表，以及起始节点和终止节点的引用。drawGrid方法循环处理所有节点，为每个节点绘制一个小方块。方块的大小由_cellSize属性确定。方块的颜色由getColor方法确定，对于不可通过的节点getColor方法会返回黑色，对于起始节点和终止节点会返回灰色，其他节点均返回白色。

然后调用findPath方法，它会创建AStar类的一个实例，调用其findPath方法，并传入网格作为参数。如果找到一条路径，则显示已访问过的所有节点，将这些节点置为一种浅灰色，路径上的所有节点则通过在节点上画一个小圈做特别显示。其结果如图4-15、图4-16和图4-17所示。

当然，在没有任何障碍物的情况下寻找一条路径并没有太大意思，所以下面为鼠标点击事件增加一个事件监听器，名为onGridClick。这个方法会找到与点击鼠标所在位置（点击点）相关联的节点，然后切换其可通过状态。接下来清除网格视图，再次寻找路径，这会重新显示网格以及所找到的路径。

现在我们所需要的就是把所有这些集成在一起，也就是我们的主文档类，这可以在Pathfinding.as文件中看到：

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

www.TopSage.com

大家网 TopSage.com

```
import flash.display.StageAlign;
import flash.display.StageScaleMode;
import flash.events.MouseEvent;

[SWF(backgroundColor=0xffffff)]
public class Pathfinding extends Sprite
{
    private var _grid:Grid;
    private var _gridView:GridView;


    public function Pathfinding()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        _grid = new Grid(50, 30);
        _grid.setStartNode(0, 2);
        _grid.setEndNode(48, 27);

        _gridView = new GridView(_grid);
        _gridView.x = 20;
        _gridView.y = 20;
        addChild(_gridView);
    }

}
}
```

这个代码的作用就是创建一个Grid和一个GridView，并将二者绑定。你也可以设置不同的起始节点和终止节点。一旦动画开始运行，就可以点击多个方块来创建切断现有路径的障碍物。注意，只要留有一些空间让路径通过，A*总会找到到达目标的一条路径，而且它总是最优路径，如图4-18所示。

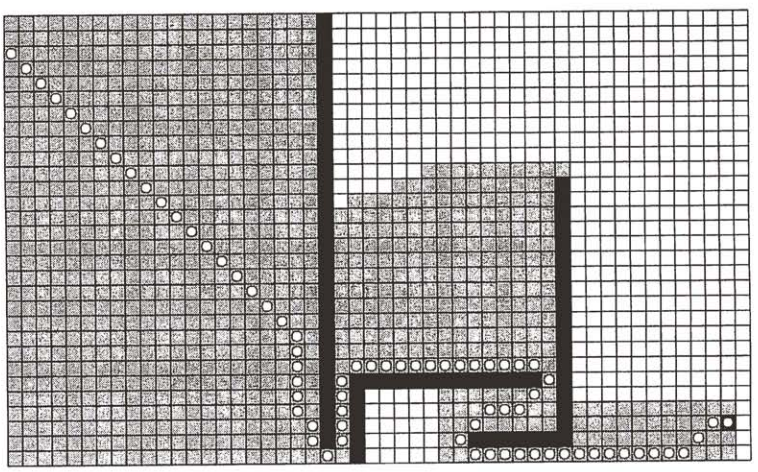


图4-18　完成寻路

www.TopSage.com

细化路径：拐角

目前还存在一个潜在的问题，通过前面这些例子也许还不能充分注意到它，不过从图4-18可以看出这个问题。路径绕过一个障碍物的边缘时，它有些“抄近路”，会从障碍物的一边沿一条对角路径到达另一边。目前看上去还不算什么大问题，不过如果查看图4-19所示的路径就会发现问题。

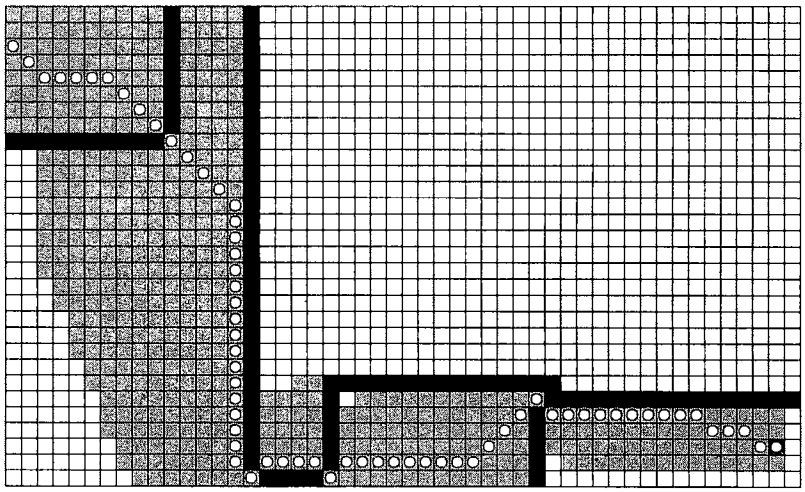


图4-19 插缝潜行

在这里，路径会在不可通过的路径之间“见缝插针”。你可能认为已经隔断了这条路径，但是由于对角移动是完全可行的，因此路径确实会选用这种对角移动。看上去这并不合适。正确的做法是绝对不要试图抄近路而绕过不可通过的区块。这要求我们必须完全绕过这些区块，并且在两个拐角对角相连的情况下（如图4-19所示），将找不到能通过的任何有效路径。

由图4-20可以看到这种情况的放大图。

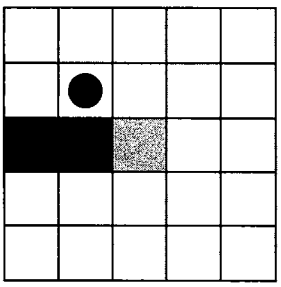


图4-20 拐角放大图

黑色圆圈表示正在分析的当前节点，灰色方格表示目前正在检查以确定其代价的节点。现在正在执行AStar类的search方法：

www.TopSage.com

```
for(var i:int = startX; i <= endX; i++)
{
    for(var j:int = startY; j <= endY; j++)
    {
        var test:Node = _grid.getNode(i, j);
        if(test == node || !test.walkable) continue;
```

我们要确保测试节点不是当前节点，而且测试节点是可通过的。如果其中任何一个条件不满足，则跳过这个节点，而移向下一个外围节点。现在我们希望再增加一个条件：如果要经过一个不可通过的节点抄近路，则跳过此节点。node和test在与此类似的一个对角线上时，则必须测试另外两个节点。这两个节点有以下坐标：

```
node.x, test.y
```

和

```
test.x, node.y
```

换句话说，这是与node在同一行上且与test在同一列上的节点（以及与node在同一列上且与test在同一行上的节点），如图4-21所示。也许要在纸上画一画才能明白，不过你会发现这很有意义。

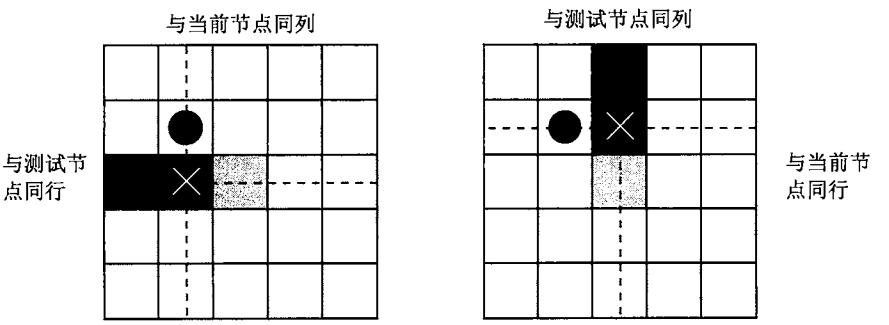


图4-21　寻找拐角

所以我们要做的就是查看这两个节点中是否有节点不可通过。如果是，则跳过test，并移向下一个外围节点。以下是修改后的代码：

```
for(var i:int = startX; i <= endX; i++)
{
    for(var j:int = startY; j <= endY; j++)
    {
        var test:Node = _grid.getNode(i, j);
        if(test == node || !test.walkable ||
           !_grid.getNode(node.x, test.y).walkable ||
           !_grid.getNode(test.x, node.y).walkable)
        {
            continue;
        }
```

做了以上简单的补充后，现在就得到了图4-22所示的路径。

www.TopSage.com

TopSage.com

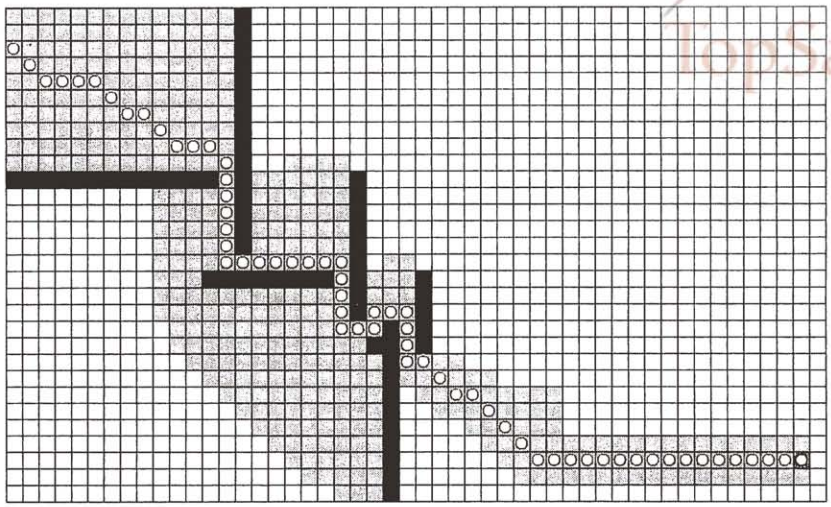


图4-22 更好的拐角

在这里，路径现在完全绕过了不可通过的节点而没有抄近路。另外，如果完全封锁了一个区域，则不会发现任何路径，如图4-23所示。

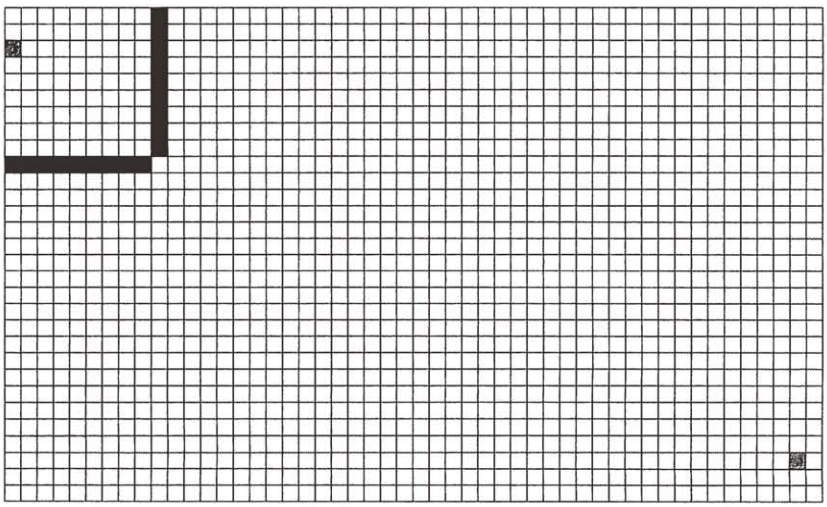


图4-23 没有任何路径

你可能想实现抄近路的修正措施，也可能并不需要。在某些实现或游戏中这是有意义的，但在另外一些游戏中却并非如此。必须针对你的特定游戏或应用确定希望完成哪种行为。甚至你可能还希望把它设计为类的一个可配置选项。另外，这里采用的实现方式还可以进一步优化。采用现在的做法时，即使测试节点不在对角线上也总会检查拐角，这就有些浪费。为了保证简洁，这里仍采用这种方式，不过你可能希望以后做进一步优化。

www.TopSage.com

4.4　在游戏中使用 AStar

前一个使用GridView类的例子主要是为了展示相关概念——了解如何建立路径，查看哪些节点被访问，并试验将不同节点置为不可通过时会发生什么情况，从而了解对路径有何影响。

在一个真实场景中，情况则恰恰相反——各个节点的可通过状态会在开始时设置，而且通常不会再改变，但是起始节点和终止节点则是动态的。起始节点是特定角色所在的位置，而终止节点通常是用户希望角色前往的位置，这往往通过点击网格的某个位置来指定。因此，下面来完成这种行为的一个非常简单的实现。所有这些都放在Game类中，可以从本书网站www.friendsofed.com下载Game.as文件得到：

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class Game extends Sprite
    {
        private var _cellSize:int = 20;
        private var _grid:Grid;
        private var _player:Sprite;
        private var _index:int;
        private var _path:Array;

        public function Game()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            makePlayer();
            makeGrid();
            stage.addEventListener(MouseEvent.CLICK, onGridClick);
        }

        /**
         * Creates the player sprite. Just a circle here.
         */
        private function makePlayer():void
        {
            _player = new Sprite();
            _player.graphics.beginFill(0xff0000);
            _player.graphics.drawCircle(0, 0, 5);
            _player.graphics.endFill();
            _player.x = Math.random() * 600;
            _player.y = Math.random() * 600;
            addChild(_player);
        }

        /**
```

www.TopSage.com

```
 * Creates a grid with a bunch of random unwalkable nodes.
 */
private function makeGrid():void
{
    _grid = new Grid(30, 30);
    for(var i:int = 0; i < 200; i++)
    {
        _grid.setWalkable(Math.floor(Math.random() * 30),
                          Math.floor(Math.random() * 30),
                          false);
    }
    drawGrid();
}

/**
 * Draws the given grid, coloring each cell based on its state.
 */
private function drawGrid():void
{
    graphics.clear();
    for(var i:int = 0; i < _grid.numCols; i++)
    {
        for(var j:int = 0; j < _grid.numRows; j++)
        {
            var node:Node = _grid.getNode(i, j);
            graphics.lineStyle(0);
            graphics.beginFill(getColor(node));
            graphics.drawRect(i * _cellSize, j * _cellSize, _cellSize, _cellSize);
        }
    }
}

/**
 * Determines the color of a given node based on its state.
 */
private function getColor(node:Node):uint
{
    if(!node.walkable) return 0;
    if(node == _grid.startNode) return 0xcccccc;
    if(node == _grid.endNode) return 0xcccccc;
    return 0xffffff;
}

/**
 * Handles the click event on the GridView.
 * Sets the start and end node of the grid and finds path.
 */
private function onGridClick(event:MouseEvent):void
{
    var xpos:int = Math.floor(mouseX / _cellSize);
    var ypos:int = Math.floor(mouseY / _cellSize);
    _grid.setEndNode(xpos, ypos);

    xpos = Math.floor(_player.x / _cellSize);
```

www.TopSage.com

大家网 TopSage.com

```
            ypos = Math.floor(_player.y / _cellSize);
            _grid.setStartNode(xpos, ypos);

            drawGrid();
            findPath();
        }

        /**
         * Creates an instance of AStar and uses it to find a path.
         */
        private function findPath():void
        {
            var astar:AStar = new AStar();
            if(astar.findPath(_grid))
            {
                _path = astar.path;
                _index = 0;
                addEventListener(Event.ENTER_FRAME, onEnterFrame);
            }
        }

        /**
         * Finds the next node on the path and eases to it.
         */

        private function onEnterFrame(event:Event):void
        {
            var targetX:Number = _path[_index].x * _cellSize + _cellSize / 2;
            var targetY:Number = _path[_index].y * _cellSize + _cellSize / 2;
            var dx:Number = targetX - _player.x;
            var dy:Number = targetY - _player.y;
            var dist:Number = Math.sqrt(dx * dx + dy * dy);
            if(dist < 1)
            {
                _index++;
                if(_index >= _path.length)
                {
                    removeEventListener(Event.ENTER_FRAME, onEnterFrame);
                }
            }
            else
            {
                 // perform an ease by moving half way to the target
                _player.x += dx * .5;
                _player.y += dy * .5;
            }
        }
    }
}
```

这个类与GridView示例有很多相似性，不过也有一些重要的区别。在构造函数中，我们创建了一个玩家，仅仅是一个随机放置在场景上的圆圈。另外还创建了一个Grid，并把其中的所有节点都标志为不可通过。

drawGrid和getColor方法与前例中基本相同。

然后监听一个click事件。点击这个网格时，将起始节点设置为玩家所在的那个节点，终止节点设置为所点击的节点。然后重新绘制网格来反映这些变化，并尝试用findPath方法找到一条路径。

findPath方法只是创建AStar的一个实例，并尝试在网格上找到一条路径。如果找到这样一条路径，则获取该路径，并将index属性设置为0，这样一来就可以从路径的第一个节点开始。然后监听enterFrame事件，调用onEnterFrame。

onEnterFrame事件处理程序获取路径上的下一个节点（由index属性定义），得到从玩家到该节点的距离，并完成一个简单的缓动。如果玩家足够接近这个点，则移至路径上的下一个节点。到达路径上的最后一个节点（终止节点）时，删除enterFrame事件监听器。

4.5 高级地形

本章开始时曾经提到过一个概念，但到目前为止我们还没有对此进行讨论。其思想是，除了简单的直线和对角移动外，可以为区块设置不同的代价，从而使一些区块访问起来比较困难，而另外一些的访问较为容易。对于这一点我们不会过于深入讲解，不过为保证有关内容的完整性确实需要有所介绍。

举例来说，泥泞道路上的节点就比平整道路上节点的代价更大，沼泽或山峦节点的代价则更高。针对地形的不同代价可以按照如下方式实现：可以向Node类添加一个额外的代价因子属性，这将作为一个乘数。一个理想的可通过节点的代价因子将是1.0，因此水平或垂直移动到该节点代价仍为1，对角移动的代价为1.414。不过，更复杂地形的节点可能会有更大的因子，如可能为2.0，因此其最终代价将达到2或2.828。

这使得AStar会尽量避开较难走的地形节点而更倾向于容易走的节点，即使后者的路径距离更长。但是在某种情况下，有可能“穿过沼泽”的代价比绕路的代价更小。这与有一个不可通过节点的情况有所不同，因为即使这个节点（如沼泽）很难走，但它确实是可通过的，不过AStar首先会寻找绕过该节点的一条更容易走的路径。

要实现这一点，首先向Node类增加一个新属性costMultiplier：

```
package
{
    /**
     * Represents a specific node evaluated as part of a
     * pathfinding algorithm.
     */
    public class Node
    {
        public var x:int;
        public var y:int;
        public var f:Number;
        public var g:Number;
        public var h:Number;
        public var walkable:Boolean = true;
```

www.TopSage.com

```
        public var parent:Node;
        public var costMultiplier:Number = 1.0;

        public function Node(x:int, y:int)
        {
            this.x = x;
            this.y = y;
        }
    }
}
```

再做一个小小的修改来实现AStar类的search方法。请看计算代价*g*部分的那行代码：

```
var g:Number = node.g + cost;
```

把它修改为：

```
var g:Number = node.g + (cost * test.costMultiplier);
```

现在将根据这个节点的因子调整其代价。在图4-24中，我对GridView类做了修改，根据某些随机的正弦和余弦计算来为各个节点指定一个不同的costMultiplier。另外，这里根据节点的代价因子绘制各个节点来区别其颜色的深浅。白色节点比深色节点更易于通过。可以在GridView2.as文件中看到这些修改（这个文件可以从本书网站得到）。注意，这里会尽可能选择更容易走的道路来建立路径。

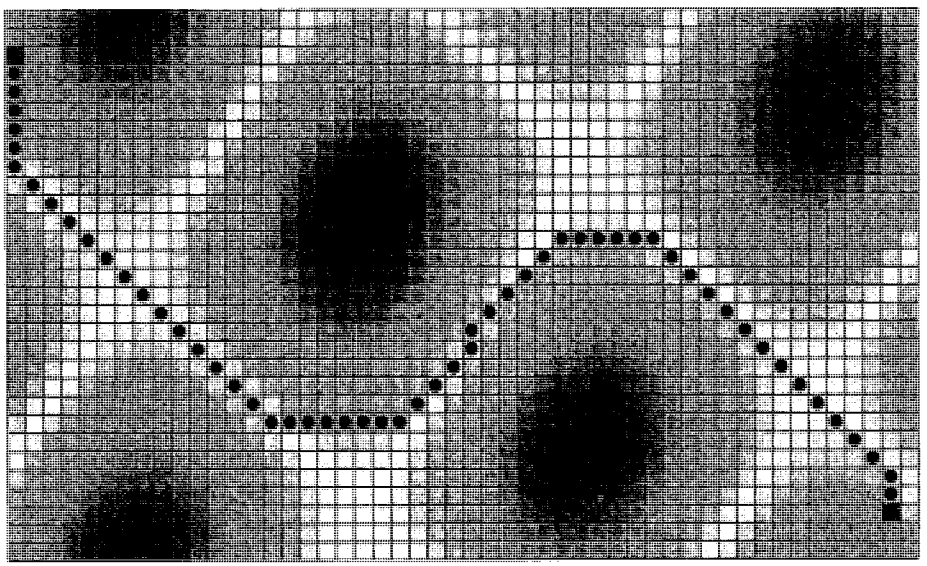


图4-24　阻力最小的路径

不过，如果建立一些不可通过的节点，让问题更复杂一些，AStar也会很乐于经过不好走的区域，如图4-25所示。

www.TopSage.com

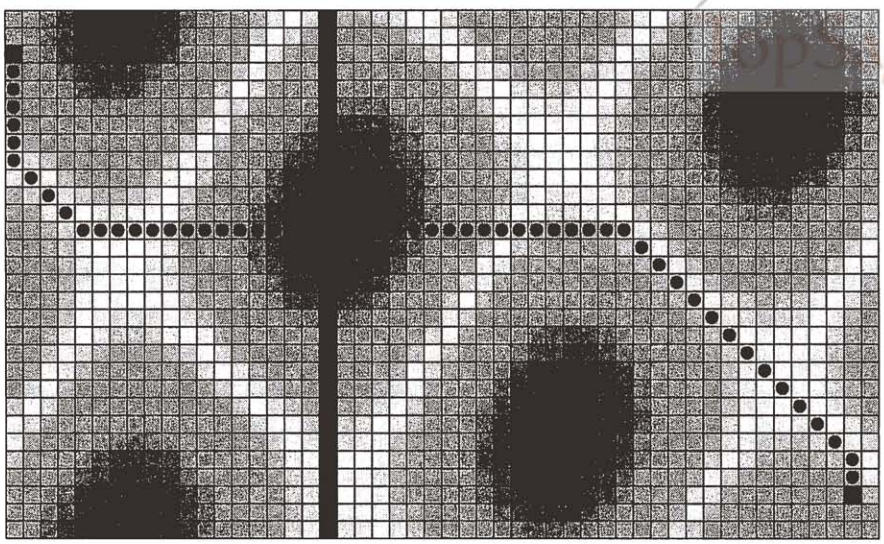


图4-25 要求路径通过难走地形

4.6 小结

最后这个游戏示例相当简单，有很多方面留待你改进，不过这里确实展示了在一个真实游戏中使用A*寻路算法的基本设置。你可以自己做一些尝试，看看玩家如何绕过障碍物到达你点击的位置（如果可以的话）。可以建立一个形象更生动的玩家，使其看起来更像一个小人，它会转向朝着目标地点的方向行走，等等。你可能还希望重新完成行走动作，这样玩家会以一种平滑的方式行走，而不是从一个节点缓动到下一个节点。当然，还可以使用一些经过图形设计得到的漂亮区块来表示绿草或道路（可通过节点）以及河流和山峦（不可通过的节点）。

将寻路应用于等角世界也相当容易，特别是如果你已经阅读并理解了前面介绍等角世界的一章内容，这个工作会更加轻松。我把这个内容作为一个练习留给你来完成。

接下来，我们换个角度看看可以采用哪些方式使用麦克风和摄像头，当然不只是用来捕获声音和图像，而是一些让用户通过运动和声音来控制应用程序的方法。

www.TopSage.com

第5章

其他输入方式：摄像头和麦克风

如果你曾上过某一门概论性的计算机科学课程，我确定第一天上课或者在课本的第1章肯定会看到如图5-1所示的图示，或者有可能稍有变化。

图5-1　输入、处理、输出

如果学习过更深入的高级课程，还可能会在图中的某个位置出现另一个标为存储（storage）的方框。

这正是计算机的实现原理。它要读入某种数据作为输入，对这些数据完成一些处理，还可能加以存储，也可能以后进行检索，然后再以某种形式输出。

有关计算机和软件的大多数文献都把重点放在中间的那个方框，即处理一些数据来创建某种有价值的输出。

通常认为输入都通过键盘或鼠标提供，而输出几乎只是向屏幕写数据，或者从扬声器（或多个扬声器）发出某个声音。当然，还存在大量其他的输入输出方式以及不计其数的变化形式，不过一般程序员和最终用户的大多数日常操作都只是处理鼠标、键盘、屏幕和音频设备。

平心而论，从输入的技术角度而言，利用键盘和鼠标可以做很多工作。这与最早的“个人计算机”当然有很大差别，对于最早的那些个人计算机，输入是指拨动一系列开关，而输出是指解释一行LED的开/关状态。不过，我认为这里有些缺乏关联，因为我们的大多数输入都是来自机械或触觉，而输出却涉及声音和视觉。

如今，如果你热衷于科幻片，就会知道最终我们都会像*Minority Report*（《少数派报告》）中一样以可视化方式控制我们的计算机，而且能够像*2001: A Space Odyssey*（《2001太空漫游》）中一样与计算机交谈（当然希望最后比这个电影的结局要好）。另外，尽管在这些领域都已经取得

www.TopSage.com

了一些进步，但目前触摸屏仍然只是模拟现有的键盘和现有的物理输入设备，而语音识别技术尽管有所改进，在关键时刻的到来之时也还是远远不够的。

因此，尽管我不建议你通过麦克风来写你的下一部小说（甚至下一个程序），但研究各种不同的输入方法（特别是麦克风和摄像头）可能会很有意思，这正是我们在本章所要讨论的内容。

5.1 摄像头和麦克风

使用不同输入方法的障碍之一是，所有输入都需要一个输入设备。不同的输入需要不同的设备。尽管如今很需要找到一台不带键盘和鼠标/跟踪球/跟踪垫的计算机，但某些输入仍需要专门的、很难得到而且通常很昂贵的硬件。你还需要安装特殊的驱动程序、软件甚至是本地服务器来与这个硬件交互，使程序确实能得到输入。

幸运的是，最近几年来生产的越来越多的笔记本电脑都带有一个内置的摄像头和麦克风，即使你的计算机没有这两样设备，也可以在几乎任何一个便利店以很便宜的价格买到（甚至比这本书的价格还要低）。如果确实需要购买麦克风和摄像头，麦克风往往是即插即用的，而摄像头总是提供一个易于安装的程序，只需3分钟就能安装完毕。作为一个Flash开发人员，你可以直接访问摄像头和麦克风作为输入设备。

人们通常只是在创建聊天或视频会议类型的应用时才会考虑到摄像头和麦克风（也就是说，获取语音或摄像机视频流，将其发送到另一台计算机，再以其原来的形式播放）。不过，有些人已经开始尝试另一种想法：使用声音和视频作为一种更直接的输入设备——捕获音频或视频输入，进行分析，并使用这些输入影响程序中正在处理的某个工作。

通过本章的介绍，你将对如何使用一个摄像头和/或麦克风作为输入设备有所认识。当然，这里肯定无法谈到每一种可能的方法，不过我会讨论捕获视频和声音的基本技术，并展示处理视频和声音的一些方法。在此基础上，你可以充分发挥自己的创造性来实现任何处理。我们先从相对简单的情况开始谈起，即处理麦克风。

5.2 声音作为输入

Flash中的麦克风输入（毫无疑问）是通过Microphone类完成的。这个类使用起来相当容易，但限制也很多。很多熟悉ActionScript Sound类的人在了解到Microphone类后马上就试图用它来计算实时声音频谱。如果你也希望如此，那么我现在就得向你泼冷水了。另外，不要认为你可以记录通过麦克风得到的声音。这些工作Microphone类都不支持。

Microphone类的主要功能是返回某个麦克风的当前**活跃级数**（activity level）。活跃级数是一个从0到100的数，指示麦克风接收的音量为多少——0表示没有声音，100是所能接收的最大音量。如果稍稍加入一些具有创造性的想法，对于这个值就可以做一些很酷的事情。首先来看如何通过ActionScript访问麦克风。

Microphone类有一个名为getMicrophone的静态方法，它会返回Microphone类的一个实例，表示与计算机连接的一个具体的麦克风。如果调用这个方法时没有提供任何参数，它会返回在系统上找到的第一个麦克风。如果向getMicrophone传入值-1，则返回系统上的默认麦克风。在大多

www.TopSage.com

数情况下，第一个麦克风即是默认麦克风，也就是你要找的麦克风。还可以给机器上插一堆麦克风，先获取系统上所有麦克风的一个列表，并使用该列表的一个索引来选择一个特定的麦克风在Flash中使用，不过往往没有这个必要。

可以利用以下MicrophoneTest类来尝试getMicrophone方法。与本书中的其余代码一样，这个类可以从本书下载页面www.friendsofed.com得到（相应文件是MicrophoneTest.as）：

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

    public class MicrophoneTest extends Sprite
    {
        private var _mic:Microphone;

        public function MicrophoneTest()
        {
            _mic = Microphone.getMicrophone();
            trace(_mic.name);
        }
    }
}
```

试着向getMicrophone传入不同的值，看看会输出什么结果。最后，如果索引一个不存在的麦克风，会返回null，你将得到一个错误，告诉你无法访问一个null对象的属性或方法。

如果希望提示用户选择一个麦克风，可以通过Security.showSettings方法调用Microphone settings面板，方法如下所示：

```
package {
    import flash.display.Sprite;
    import flash.media.Microphone;
    import flash.system.Security;
    import flash.system.SecurityPanel;

    public class MicrophoneTest extends Sprite
    {
        private var _mic:Microphone;

        public function MicrophoneTest()
        {
            _mic = Microphone.getMicrophone();
            Security.showSettings(SecurityPanel.MICROPHONE);
        }
    }
}
```

这会向用户显示Microphone对话框（如图5-2所示），提示选择一个麦克风。

选择一个麦克风并按下Close按钮后，将显示Camera and Microphone Access对话框，如图5-3所示。

之前可能你已经见过这个对话框——它只是确认Flash动画确实可以使用计算机上的麦克风和/摄像头。可以想象一下，如果Flash动画可以访问你的摄像头和麦克风，并且开始播放你的私

www.TopSage.com

人生活，而此时你却毫不知情，这会是一个严重的问题！

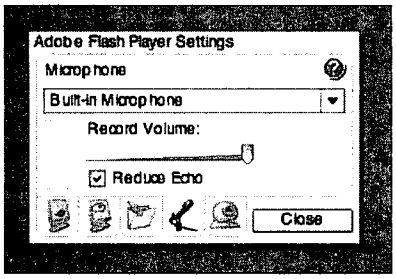


图5-2 Microphone对话框

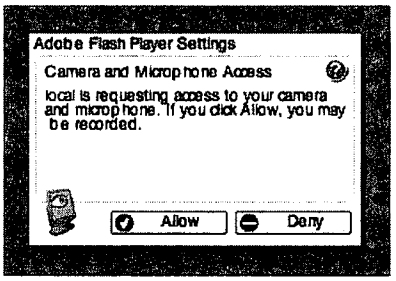


图5-3 Camera and Microphone Access对话框

你可能想知道为什么在第一个例子中没有看到这个访问对话框。在第一个例子中，尽管确实获取了麦克风的一个引用，但是并没有真正开始使用它，所以那时不存在任何风险。Flash会在麦克风数据被访问的第一时间显示这个访问对话框。

现在开始访问这个麦克风，并读取活跃级数。活跃级数会不断改变，所以最好一直进行检测。听起来这个任务很适合由enterFrame处理程序来完成。以下是我们的第一个尝试：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.media.Microphone;
    import flash.system.Security;
    import flash.system.SecurityPanel;

    public class MicrophoneTest extends Sprite
    {
        private var _mic:Microphone;

        public function MicrophoneTest()
        {
            _mic = Microphone.getMicrophone();
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            trace(_mic.activityLevel);
        }
    }
}
```

代码相当简单。onEnterFrame处理程序只是输出_mic.activityLevel。不过，做这个尝试时，只会输出一长串的-1。不论你讲话的声音有多大，不论选择哪一个麦克风，或者不论是否允许Flash动画使用你的麦克风，都不会记录任何活动。再次指出，这只是因为Flash动画还没有真正处理麦克风数据。

要让Flash动画开始使用这个数据有两种办法。一种办法是将麦克风关联到一个NetStream，

通常在某种聊天应用程序或语音记录系统中就会采用这种方法，为此要使用一个流服务器，如Flash Media Server或Red5。另一种方法是使用setLoopback方法设置麦克风的输入通过扬声器循环回送。向这个方法传入true将导致Flash开始访问麦克风的输入，如果你对着麦克风练练嗓子，此时就会看到活跃级数的变化。不出所料，传入false会停止活动。如果不希望听到你自己的声音，只需关掉计算机上的输出音量。

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

    public class MicrophoneTest extends Sprite
    {
        private var _mic:Microphone;
        public function MicrophoneTest()
        {
            _mic = Microphone.getMicrophone();
            _mic.setLoopBack(true);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            trace(_mic.activityLevel);
        }
    }
}
```

运行这个示例，对着麦克风讲话，看看输出数值的变化。如果不论你发出多大声音这些数字都很低，则可能需要调整增益。**增益**（gain）实际上就是在Flash中处理麦克风数据之前应用于麦克风的一种放大级数。增益可以采用两种方式设置：手动设置以及代码设置。在图5-2所示的Microphone对话框中，有一个标为Record Volume的滑块。改变这个滑块就会改变增益。如果通过代码改变这个值，只需将麦克风的gain属性设置为介于0到100之间的一个值，这里100是最大的放大级数。

假设现在得到了一些有意思的数字，下面对这些数字做一些处理，而不只是简单地将其输出。下一个例子将创建一个位图，并在每一帧上利用活跃级数在位图中设置一个像素。这个位图向左边滚动，这样就能得到到达麦克风的声音音量不断变化的记录：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.media.Microphone;

    public class MicrophoneTest extends Sprite
```

www.TopSage.com

大家网 TopSage.com

```
    {
        private var _mic:Microphone;
        private var _bmpd:BitmapData;

        public function MicrophoneTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _bmpd = new BitmapData(400, 50, false, 0xffffff);
            addChild(new Bitmap(_bmpd));
            _mic = Microphone.getMicrophone();
            _mic.setLoopBack(true);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
             // draw a pixel at coordinates:
             // x = 298
             // y = 50 minus half of the mic level
            _bmpd.setPixel(298, 50 - _mic.activityLevel / 2, 0);
             // scroll the bitmap to the left
            _bmpd.scroll(-1, 0);
        }
    }
}
```

这会得到如图5-4所示的结果。

图5-4 活跃级数可视化表示

对于可视化表示来说这很不错，但是现在还没有真正使用麦克风作为一种候选输入方法。建立一个简单游戏并使用声音来控制一个角色的移动，如何？下面的例子就会完成这样的工作。

5.2.1 一个由声音控制的游戏

下一个例子是一个名为SoundFlier的类（可以在本书配套网站提供的文件中找到相应的SoundFlier.as文件）。我们将创建一个类似飞机的运输工具，它会飞过一个绘制粗糙的山洞，其中遍布着矩形的钟乳石和石笋，这里使用麦克风活跃级数来控制其飞行：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
```

```
    import flash.events.Event;
    import flash.geom.Rectangle;
    import flash.media.Microphone;
[SWF(backgroundColor=0xffffff)]
public class SoundFlier extends Sprite
{
    private var _mic:Microphone;
    private var _flier:Sprite;
    private var _bg:Bitmap;
    private var _yVelocity:Number = 0;

    public function SoundFlier()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        makeBackground();
        makeFlier();

        _mic = Microphone.getMicrophone();
        _mic.setLoopBack(true);
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function makeBackground():void
    {
        var bmpd:BitmapData = new BitmapData(stage.stageWidth, stage.stageHeight, false);
        _bg = new Bitmap(bmpd);
        addChild(_bg);
    }

    private function makeFlier():void
    {
        _flier = new Sprite();
        _flier.graphics.lineStyle(0);
        _flier.graphics.moveTo(-10, 0);
        _flier.graphics.lineTo(-10, -8);
        _flier.graphics.lineTo(-7, -4);
        _flier.graphics.lineTo(10, 0);
        _flier.graphics.lineTo(-10, 0);
        _flier.x = 100;
        _flier.y = stage.stageHeight - 50;
        addChild(_flier);
    }

        private function onEnterFrame(event:Event):void
        {
            _yVelocity += .4;
            _yVelocity -= _mic.activityLevel * .02;
            _flier.y += _yVelocity;
            _yVelocity *= .9;
            _flier.y = Math.min(_flier.y, stage.stageHeight - 20);
            _flier.y = Math.max(_flier.y, 20);
```

www.TopSage.com

```
            var h:Number = Math.random() * 120;
            _bg.bitmapData.fillRect(new Rectangle( stage.stageWidth - 20, 0, 5, h), 0xcccccc);
            h = Math.random() * 120;
            _bg.bitmapData.fillRect(new Rectangle( stage.stageWidth - 20, stage.stageHeight - h,
                                    5, h), 0xcccccc);
            _bg.bitmapData.scroll(-5, 0);
        }
    }
}
```

这个飞机是一个sprite，背景是一个滚动的位图，其中向右边不断绘制着随机的矩形。这里最重要的部分就在onEnterFrame方法中：

```
_yVelocity += .4;
_yVelocity -= _mic.activityLevel * .02;
_flier.y += _yVelocity;
_yVelocity *= .9;
_flier.y = Math.min(_flier.y, stage.stageHeight - 20);
_flier.y = Math.max(_flier.y, 20);
```

我们利用_yVelocity属性跟踪垂直速度。在每一帧上，由于重力原因将垂直速度增加0.4，然后减去活跃级数*0.02 来完成一些升降。这两个系数是通过大量反复试验得到的。（感谢我6岁的女儿Kristine在这些测试中的大力协助！）然后再把垂直速度加至飞机的*y*轴，另外再通过乘以0.9来模拟的摩擦，最后要限定飞机的位置在一个最小和最大区域之间，使之不会飞出屏幕，其结果如图5-5所示。

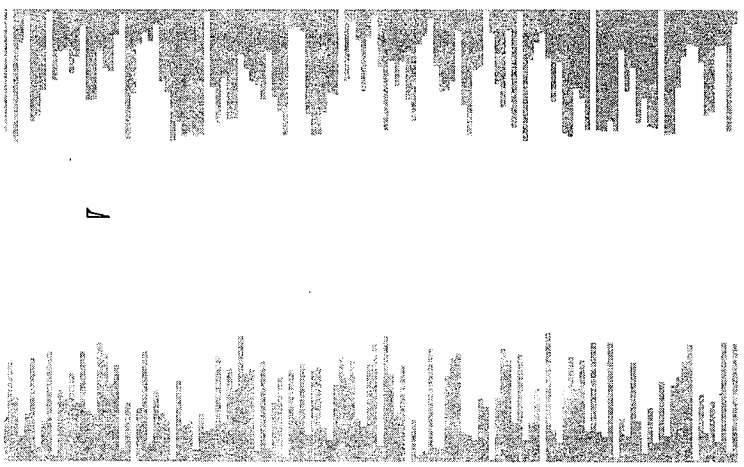


图5-5 声控飞机

对麦克风讲话或者发出噪音时，飞机就会飞行。如果不发出声音，它就会降落到地面。石笋和石柱都向左滚动，让人感觉飞机在向右飞行。这里没有实现任何碰撞检测（我可不想替你完成所有工作），不过你应该已经有想法了。要发出足够的噪声使飞机处于中间区域。这样一种游戏可没法趁老板不注意时在你的小房间里偷偷地玩儿。

www.TopSage.com

尽管这只是一个最基本的游戏，但通过这个例子应该能激发你并让你迸发出一些智慧火花，对于如何使用一个麦克风控制游戏或应用有所想法。接下来，我们将讨论另外一种处理麦克风活动的方法。

5.2.2　活动事件

在SoundFlier游戏中，我们在每一帧上都要检查活跃级数。对于这种类型的游戏，它需要用户的快速反应，还要求游戏本身的即时响应，所以持续地检查和更新至关重要。不过可以想象有些应用只是在声音达到某个特定的阈值时才会有动作。你可能会尝试运行一个enterFrame处理程序，并在每一帧上检查活跃级数，以查看它是否超过一个给定的值。但是这种处理效率可能很低，它会一直占用CPU周期（本来不需如此，而应当等待某件事情发生即可）。

幸运的是，Microphone类为这种情况提供了一个很合适的解决方案ActivityEvent，这是一个事件类，只要达到一个指定的声音阈值水平，Microphone类就会分派这个事件。有意思的是，ActivityEvent会在两种情况下分派：一种情况是活跃级数超过阈值，另一种情况是活跃级数再一次降到该阈值以下。为了对这两种情况加以区分，这个事件有一个名为activating的布尔值属性。如果这个属性为true，说明活跃级数是从更安静的状态变化而来，并逐渐增加，超出阈值。如果为false，则说明活跃级数随着声音的减弱落在阈值以下。

通过监听这个事件，可以让应用处于一种空闲状态，只是在声音达到某个音量以及再次安静下来时才做出反应。

可以把它用作为一种“声音激活开关”来控制任何应用或游戏的任何方面。在这里，你可能想超越基于浏览器的应用，而考虑它如何在基于AIR的桌面应用中使用。甚至由此可以创建一个粗糙的安全系统。假设一个应用在你房间里“听到某种声音”，它就会开启你的摄像头，并通知正在上班的你，以便你查看房间里发生了什么状况。

或者你可以把麦克风安装在某个机器旁边。如果机器停止运行，此时应用会感应到活跃级数下降到阈值以下并采取行动。

下面运行一个相当简单的例子来查看它的实际工作。这里没有什么可以束缚你，一旦了解这是如何工作的，你完全可以使用这种机制控制任何方面。这里将重新创建一个你家中台灯关联的设备，可以通过拍手来控制其开与关。所以将这个类命名为Clapper（可以下载Clapper.as文件得到这个类）：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.ActivityEvent;
    import flash.media.Microphone;

    public class Clapper extends Sprite
    {
        private var _mic:Microphone;
        private var _on:Boolean = false;
```

www.TopSage.com

```
        public function Clapper()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _mic = Microphone.getMicrophone();
            _mic.setLoopBack();
            _mic.setSilenceLevel(25, 500);
            _mic.addEventListener(ActivityEvent.ACTIVITY, onActivity);

            update();
        }

        private function onActivity(event:ActivityEvent):void
        {
            if(event.activating)
            {
                _on = !_on;
                update();
            }
        }

        private function update():void
        {
            graphics.clear();
            if(_on)
            {
                graphics.beginFill(0xffffff);
            }
            else
            {
                graphics.beginFill(0);
            }
            graphics.drawRect(0, 0,
                              stage.stageWidth, stage.stageHeight);
        }
    }
}
```

这里都是一些很基础的内容。首先得到一个麦克风，设置它循环回放，使得Flash开始监听它的输入，并为ActivityEvent.ACTIVITY增加一个监听器。注意，这里还调用了一个名为setSilenceLevel的方法。应该还记得前面的讨论中我曾多次提到阈值的概念。这个阈值如下定义：如果低于这个值，Flash会认为安静，如果高于这个值，Flash则会认为有活动。当麦克风的activityLevel超过这个水平（可以低于或高于），就会触发一个ActivityEvent事件。这里将它设置为25，所以只需一点声音就能使它触发，不过也不必太嘈杂。如果没有调用setSilenceLevel，则麦克风使用的默认阈值是10。

setSilenceLevel的第二个参数是一个超时值，以毫秒为单位。分派一个ActivityEvent事件时，麦克风会忽略这段时间内所有其他可能的活动事件。假设你想注意一种不连贯的声音，如一种“梆梆梆”的声音。每次发出“梆”的声音时，活跃级数会上升然后下降，所以对于每声“梆”

www.TopSage.com

你会得到两个事件。实际上，你只想知道梆梆声何时开始以及何时结束。通过调整这个超时时间，可以忽略两声“梆”之间的间隙。其默认值是2 000，即2秒延迟，这对于聊天应用程序可能最为适合，因为人在谈话时可能需要停顿1秒左右，但你并不希望把这当作沉默。对于这个应用，我把这个值设置得稍低一点，降至500，这样就能相当快地拍手打开或关上“台灯”。

onActivity方法检查event.activating的值。同样，如果活动是由无声变为有声，将为true，如果从有声到无声，则为false。我们将忽略第二种情况，只强调有新的强噪声发出的情况，即拍手。在这里，我将切换变量_on，并调用update。update方法只是根据_on的状态绘制一个黑色或白色矩形。

可以试试看。拍手，停下！建议你试着修改setSilenceLevel的参数，看看会有怎样的影响。另外还有一点值得一提，如果你想查看当前的设置情况，Microphone类还有两个只读属性：silenceLevel和silenceTimeout。

以上就是使用麦克风的基础知识。正如我所说，这里没有太复杂的功能，不过与之类似，尽管键盘或鼠标也没有太复杂的功能，但是我们可以看到利用键盘和鼠标所能完成的处理是何等丰富。现在来看使用视频作为输入的方法。

5.3　视频作为输入

声音输入由Microphone类处理，视频输入要由Camera类处理。尽管它们处理的内容不同，但这两个类非常相似。通过调用Camera.getCamera可以得到与计算机关联的摄像头的引用。Camera类有一个activityLevel属性，可以分派ActivityEvent事件，并且有一个setMotionLevel（对应于Microphone类的setSilenceLevel）。这两个方法都设计为可以本地使用，或者与一个流服务器（如Red5或Flash Media Server）结合使用。同样地，本章只是使用Camera提供本地输入。

不过，尽管存在这些相似性，但利用摄像头可以做更多的工作。首先，除了简单地读取活跃级数或对活动事件做出响应外，还可以通过一个Video对象查看摄像头的输入，这是一个显示对象，可以添加到显示列表，而且对显示对象能做的一切处理（如滤镜、变换、混合模式等）同样可以应用于这个Video对象。也许摄像头最强大的用法就是向BitmapData绘制视频内容。完成这个工作时，可以逐像素或逐帧地分析图像，将图像的不同区域相互比较，或者将一帧与下一帧进行比较。对于图像处理来说，会有无穷无尽的可能性。本章会给出几个例子来启发你的想像力，不过即使我用整本书专门介绍，相信一个星期以后你就能提出我根本想不到的做法。

千里之行始于足下，所以下面先来看如何获得摄像头，并查看它的输入。第一步与获得一个麦克风的情况几乎完全相同。下面的类可以通过下载CameraTest.as而得到：

```
package {
    import flash.display.Sprite;
    import flash.media.Camera;
    import flash.system.Security;
    import flash.system.SecurityPanel;

    public class CameraTest extends Sprite
    {
```

www.TopSage.com

大家网 TopSage.com

```
        private var _cam:Camera;

        public function CameraTest()
        {
            _cam = Camera.getCamera();
            trace(_cam.name);

            Security.showSettings(SecurityPanel.CAMERA);
        }
    }
}
```

调用Camera.getCamera（当然是）获得摄像头。可以传入一个字符串（表示系统上某个摄像头的名），但一般来讲，你并不知道有什么摄像头，所以最好不传入任何参数，从而得到默认的摄像头。不过，通常一种好的做法是，启动Camera设置对话框，提示用户选择一个特定的摄像头。见本例的最后一行代码，这会得到如图5-6所示的对话框。尽管本章后面的例子中我不会再给出这个对话框，但你要记住这个对话框的存在，而且在你的应用中考虑使用这个对话框以确保用户知道如何选择摄像头。

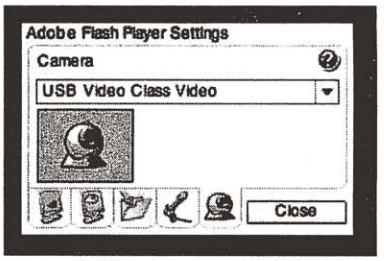


图5-6 Camera对话框

既然有了一个摄像头引用，下面就要让它输出视频了。为此需要将它关联到一个Video实例。如前所述，Video类是一个显示对象。因此可以创建一个Video对象，为它关联摄像头，再把它添加到显示列表：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.media.Camera;
    import flash.media.Video;

    public class CameraTest extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;

        public function CameraTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _cam = Camera.getCamera();
            _vid = new Video();
            _vid.attachCamera(_cam);
            addChild(_vid);
        }
    }
}
```

www.TopSage.com

如果一切正常（而且不像我这样处在恶劣的环境中），而且在一个光线很好的房间里，就会看到你漂亮的脸显示在你面前（我看到的只是一个颗粒状灰色矩形上的一个颗粒状灰色大斑点，不过我相信这确实是我）。运行这个文件后，你会再次看到如图5-3所示的Camera and Microphone Access对话框，因为现在要访问摄像头的输入。

5.3.1　视频大小和质量

现在已经让一个摄像头捕获视频并显示到屏幕上。一切都很好，不过画面不是太大，可能看起来没有在同一个计算机上使用同一个摄像头但利用其他应用生成的视频那么清晰。这并不是因为Flash不行，而只是因为你对于摄像头和视频使用的是默认设置和大小。

可以在创建视频时就指定视频的大小。它默认为320×240，但是可以传入一个更大的尺寸：

```
_vid = new Video(640, 480);
```

或者，可以在视频创建后再设置视频的宽度和高度。

现在有了一个更大的视频，不过画面不一定更好。为了改善这个视频画面，必须再来看Camera类和它的setMode方法。它接受4个参数，不过这里只讨论前3个参数：宽度、高度和每秒帧数（frames per second，fps）。（第4个参数确定了Flash如何处理一种无法与摄像头匹配的模式。对于这一点你还可以进行深入研究。）默认情况下，这些参数设置为160×120，且帧速率为15 fps。哈！这就是为什么视频看起来那么糟糕的原因。把它放大到320×240或者640×480来看有什么区别。15 fps的设置可能对于现在来看还算可以：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.media.Camera;
    import flash.media.Video;

    public class CameraTest extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;

        public function CameraTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _cam = Camera.getCamera();
            _cam.setMode(640, 480, 15);
            _vid = new Video(640, 480);
            _vid.attachCamera(_cam);
            addChild(_vid);

        }
    }
}
```

www.TopSage.com

要知道，摄像头的帧速率和分辨率越高，计算机处理所有到来的视频数据时要做的工作就越多。稍后就会看到，使用一个摄像头提供输入时，视频的质量和大小可能不像在视频会议之类的应用中那么重要。所以倘若不需要就不要太讲究。如果再仔细查看，可能会看到一个setQuality方法，这个方法要处理带宽和压缩率从而将视频流发送到一个流服务器，而这对于本章要做的工作不太适用，所以现在先不必考虑这一个方法。

如果这是你第一次在Flash中使用摄像头编程，可以花些时间来做些尝试。对视频指定一些滤镜或混合模式、旋转、缩放、移动，尽可以享受其乐趣。在此之后，我们来剖析视频流。

5.3.2 视频与位图

前面提到过，使用摄像头作为输入的强大功能之一体现在与BitmapData类结合使用时。为此，只需使用BitmapData draw方法将视频绘制在一个BitmapData对象上。这样就可以对整个视频完成像素级的访问和控制。当然，对于一个移动的图像，可能希望通过一个定时器或利用帧反复做此处理。以下是一个基本设置，可以下载CameraBitmap.as文件得到：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.media.Camera;
    import flash.media.Video;

    public class CameraBitmap extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
        private var _bmpd:BitmapData;

        public function CameraBitmap()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _cam = Camera.getCamera();
            _cam.setMode(320, 240, 15);
            _vid = new Video(320, 240);
            _vid.attachCamera(_cam);

            _bmpd = new BitmapData(320, 240, false);
            addChild(new Bitmap(_bmpd));
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _bmpd.draw(_vid);
```

www.TopSage.com

大家网 TopSage.com

```
        }
    }
}
```

这里创建了一个BitmapData，将它包装在一个Bitmap中，并把它增加到显示列表而不是增加视频。在每一帧上，将视频绘制到BitmapData上。这里将大小减少为320×240以减少CPU的工作压力。除此以外，实际上你无法区分看到的是一个位图而不是视频。不过现在可以拥有BitmapData的强大功能，每一帧都可以读取和调整这个视频中的每一个像素。

翻转图像

要记住一点，用户会面对摄像头，并做各种手势试图进行控制。如果他向左移，他希望这个动作能把物体向左移。如果他向右移，则希望物体也移向右边。但是摄像头会原样显示用户的图像，就好像有人在直接看着他而不是看一面镜子。这样一来感觉就是反的，会让用户很困惑。不过，可以在绘制时通过翻转图像来解除这种困惑。

只需修改向位图绘制视频的那行代码，如下：

```
var matrix:Matrix = new Matrix(-1, 0, 0, 1, _bmpd.width, 0);
_bmpd.draw(_vid, matrix);
```

首先确保已经导入了flash.geom.Matrix类。传入一个矩阵以便在绘制对象时完成变换。可以利用矩阵做很多处理，不过前4个参数控制了缩放、旋转和偏正。这里第一个-1的作用是在*x*轴缩放图像，将其翻转。0表示图像不会旋转或歪斜，下一个1表示在*y*轴上缩放至100%。不过，现在它会扩展出位图左侧，所以你什么也看不到。最后两个参数用于在*x*和*y*轴上移动图像。所以在*x*轴上移动图像本身的宽度，把它移回到可见范围内，而在*y*轴上不做移动。试着对上一个例子也做此修改。现在看起来你就像在看一面镜子，这样会让控制更为自然。

5.3.3　分析像素

既然已经拥有了对像素的控制权，该对它们做什么处理呢？还记得我曾经说过，视频输入并不一定要求是高质量和高分辨率的视频。这里再更进一步说，要知道尽量使用低质量和低分辨率的视频可能会更好一些。即使是一个320×240的视频图像，每一帧也需要分析76 800个像素。而且有可能你根本不需要所有这些数据。另外，试图处理所有这些数据会让你的CPU负荷很重，也许会使你的计算机运行得发烫，甚至烫伤你的大腿，还会消耗电池，使得计算机上的所有应用都运行缓慢。所以应该让CPU轻松一点，不要对视频大小过于苛求。

事实上，你会看到我们首先要做的工作就是专门设计如何去除所有这些像素中的大量信息。通常我们感兴趣的是视频中出现某种颜色的特定区域，或者是存在某种对比或出现变化的区域。

1. 分析颜色

首先我们尝试跟踪一种特定颜色。可以让用户在摄像头前面手持一种颜色鲜亮的物体，比如说亮红色，并在他或她移动这个物体时跟踪这种颜色的位置，这样一来这个物体就像是一个虚拟操纵杆了。

可以使用BitmapData的一个内置方法跟踪特定颜色像素：getColorBoundsRect。基本说来，我们要传入一个要查找的颜色，它会返回一个矩形，表示包围含此颜色的所有像素的一个外围方

www.TopSage.com

框。这个方法如下：

```
bitmapData.getColorBoundsRect(mask:uint, color:uint, findColor:Boolean);
```

利用掩码（mask）可以查找特定的颜色通道。例如，如果传入0xFF0000作为掩码，这个方法就只会查看像素的红色分量来确定是否匹配。这是因为这个掩码的红色通道是FF，而绿色和蓝色通道都是00，所以它会忽略后两个分量，而只看红色分量。这个颜色显然就是你要找的颜色。如果findColor参数为true，则会查找这样一个矩形，它要包围含所传入颜色的所有像素；如果为false，返回的矩形将包围不是这种颜色的所有像素。

由于我们不知道用户想要跟踪什么颜色，因此让他们点击位图来选择一个颜色。我们会找到包围该颜色的矩形，并在位图上方的一个sprite中绘制一个矩形。到目前为止，代码如下所示（在ColorTracking.as文件中）：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.geom.Matrix;
    import flash.geom.Rectangle;
    import flash.media.Camera;
    import flash.media.Video;

    public class ColorTracking extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
        private var _bmpd:BitmapData;
        private var _cbRect:Sprite;
        private var _color:uint = 0xffffff;
        public function ColorTracking()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _cam = Camera.getCamera();
            _cam.setMode(320, 240, 15);
            _vid = new Video(320, 240);
            _vid.attachCamera(_cam);

            _bmpd = new BitmapData(320, 240, false);
            addChild(new Bitmap(_bmpd));

            _cbRect = new Sprite();
            addChild(_cbRect);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            stage.addEventListener(MouseEvent.CLICK, onClick);
```

www.TopSage.com

```
        }

        private function onClick(event:MouseEvent):void
        {
            _color = _bmpd.getPixel(mouseX, mouseY);
        }

        private function onEnterFrame(event:Event):void
        {
            _bmpd.draw(_vid, new Matrix(-1, 0, 0, 1, _bmpd.width, 0));
            var rect:Rectangle = _bmpd.getColorBoundsRect(0xffffff, _color, true);


            _cbRect.graphics.clear();
            _cbRect.graphics.lineStyle(1, 0xff0000);
            _cbRect.graphics.drawRect(rect.x, rect.y, rect.width, rect.height);
        }
    }
}
```

这里并没有太高深的内容（起码目前还没有）。我们创建了一个sprite，名为_cbRect，并在其中绘制包围颜色的矩形。用户点击位图时，使用getPixel找出其点击的颜色像素。每一帧都会将这个颜色传入getColorBoundsRect。注意，我们使用了掩码0xFFFFFF，所以会查看这个颜色的所有通道。得到这个矩形时，用红线将该矩形绘制在_cbRect中。非常简单。

现在，试着在摄像头前面拿着一个颜色鲜亮的物体，在位图中点击它，然后晃动一下这个物体。你可能会看到某些帧上会出现一个矩形，但是它可能非常小。也许过一会消失然后又出现，突然变得非常大，然后又不见了。也许这个例子能很好地工作，确实能跟踪物体一段时间，但是它肯定还不够稳定。

问题在于，这里有太多的像素信息——76 800个像素，每个像素可能是1 600余万种颜色值之一。你已经选择了这些值中的一个。即使物体是单色的，光照、阴影、纹理、形状等方面也会导致其表面颜色的各种变化。开始移动物体时光照和角度会改变。在任何时刻，都有可能根本没有特定颜色值的像素，或者即使有这样一些像素，但它们有可能相距很远。

基本说来，需要简化物体的颜色。为此有很多种方法。首先可以利用一个简单的模糊滤镜来实现。创建Video对象之后，对它增加一个模糊滤镜：

```
_vid = new Video(320, 240);
_vid.attachCamera(_cam);
_vid.filters = [new BlurFilter(10, 10, 1)];
```

增加这个模糊滤镜后可能会看到立竿见影的效果，也可能没有任何帮助。模糊会导致相邻像素之间的差别变小，不过仍然并非完全相同。

这里有很多选择。如果没有用过BitmapData类或位图滤镜，可以了解一下，利用它们可以做各种处理。利用ColorTransform类可以完成颜色变换，copyChannel类对处理颜色通道也很有用，另外threshold函数非常擅长将一个图像分割为更简单的部分。不过对于这种情况我发现

www.TopSage.com

paletteMap方法确实很有用。

paTetteMap方法取一个位图的各个红、绿和蓝色通道（如果愿意甚至可以取α通道），将其值映射到另一个数组。对于一个图像中的各个像素，每个通道包含一个介于0到255之间的值（或者如果你习惯使用十六进制，也就是0x00到0xFF）。各个通道可以映射到一个数组，该数组将包含256个值。所以，如果原来的值是127，paletteMap就会查看数组中的元素127，来确定对于这个通道为该像素指定什么值。不过，让人有些迷惑的是，数组本身包含32位数，而不是0–255。所以，红色通道映射数组会包含从0x000000到0xFF0000的值。绿色通道映射数组会包含从0x000000到0x00FF00的值，而蓝色通道将包含0x000000到0x0000FF的值。因此，如果将各个数组逆置（例如，使得蓝色通道映射数组包含0x0000FF到0x000000），则会得到一张完全反色的图像。还可以混合匹配这些通道，将红色元素放入绿色通道中，诸如此类。

这里我们感兴趣的是减少每个通道的颜色数。为此，假设对于各个数组中的某个索引范围，所有元素都有相同的值—实际上就是将其“归一”。例如，在红色数组中，我们假设索引从0到15的所有元素都包含0x000000（红色通道为0x00）；从16到31，所有元素都包含0x100000（红色通道为0x10）；从32到63，所有元素都包含0x200000（红色通道为0x20），依此类推。这样下来，每个通道只会包含16种不同的值，而不是256个不同的值，图像中总共只可能有4 096种颜色。

完成这个工作的方法名为makePaletteArrays，该方法如下所示：

```
private function makePaletteArrays():void
{
    _red = new Array();
    _green = new Array();
    _blue = new Array();
    var levels:int = 8;
    var div:int = 256 / levels;
    for(var i:int = 0; i < 256; i++)
    {
        var value:Number = Math.floor(i / div ) * div;
        _red[i] = value << 16;
        _green[i] = value << 8;
        _blue[i] = value
    }
}
```

在这里，实际上要把各个通道划分为8个不同的值，这样图像中总共只会有256种颜色。采用这种做法，使得跟踪某种颜色相当容易。还可以通过改变levels变量来调整。完成了上述实现的类如下所示：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
```

www.TopSage.com

```
import flash.events.MouseEvent;
import flash.filters.BlurFilter;
import flash.geom.Matrix;
import flash.geom.Point;
import flash.geom.Rectangle;
import flash.media.Camera;
import flash.media.Video;

public class ColorTracking extends Sprite
{
    private var _cam:Camera;
    private var _vid:Video;
    private var _bmpd:BitmapData;
    private var _cbRect:Sprite;
    private var _color:uint = 0xffffff;
    private var _red:Array;
    private var _green:Array;
    private var _blue:Array;
    public function ColorTracking()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        _cam = Camera.getCamera();
        _cam.setMode(320, 240, 15);
        _vid = new Video(320, 240);
        _vid.attachCamera(_cam);
        _vid.filters = [new BlurFilter(10, 10, 1)];
        _bmpd = new BitmapData(320, 240, false);
        addChild(new Bitmap(_bmpd));

        _cbRect = new Sprite();
        addChild(_cbRect);

        makePaletteArrays();

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
        stage.addEventListener(MouseEvent.CLICK, onClick);

    }

    private function makePaletteArrays():void
    {
        _red = new Array();
        _green = new Array();
        _blue = new Array();
        var levels:int = 8;
        var div:int = 256 / levels;
        for(var i:int = 0; i < 256; i++)
        {
            var value:Number = Math.floor(i / div ) * div;
            _red[i] = value << 16;
            _green[i] = value << 8;
            _blue[i] = value
```

www.TopSage.com

```
            }
        }

        private function onClick(event:MouseEvent):void
        {
            _color = _bmpd.getPixel(mouseX, mouseY);
        }

        private function onEnterFrame(event:Event):void
        {
            _bmpd.draw(_vid, new Matrix(-1, 0, 0, 1, _bmpd.width, 0));
            _bmpd.paletteMap(_bmpd, _bmpd.rect, new Point(), _red, _green, _blue);

            var rect:Rectangle = _bmpd.getColorBoundsRect(0xffffff, _color, true);

            _cbRect.graphics.clear();
            _cbRect.graphics.lineStyle(1, 0xff0000);
            _cbRect.graphics.drawRect(rect.x, rect.y, rect.width, rect.height);
        }
    }
}
```

这样仍然不够完美，通常需要对光照和背景做一些调整。但是如果你手拿的物体颜色鲜亮，光照很好，而且背景较暗，效果就不错。还可以调整levels变量来得到适量的颜色变化，以保持其稳定性。实际上，你可能希望在应用中做一个敏感度控制，从而允许用户动态地调整这个值。只要确保每次发生改变时都要重新计算通道调色板映射。

2. 使用跟踪色作为输入

既然我们跟踪了这个矩形，对它能做些什么呢？下面使用它的位置来完成移动。下一个例子将创建一个球，它会随着物体的运动而运动。可以让这个球在图像上完全随物体运动，不过也可以让它在完全不同的一个区域移动，甚至可以控制Space Invaders之类游戏中的一个角色。

这个例子的最终代码清单会增加一个sprite，其中绘制有一个圆，并且随着你移动跟踪物体，这个sprite会在场景上做同样的运动：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.filters.BlurFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;
    import flash.geom.Rectangle;
    import flash.media.Camera;
    import flash.media.Video;

    public class ColorTracking extends Sprite
    {
```

www.TopSage.com

大家网 TopSage.com

```
private var _cam:Camera;
private var _vid:Video;
private var _bmpd:BitmapData;
private var _cbRect:Sprite;
private var _color:uint = 0xffffff;
private var _red:Array;
private var _green:Array;
private var _blue:Array;
private var _ball:Sprite;

public function ColorTracking()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    _cam = Camera.getCamera();
    _cam.setMode(320, 240, 15);
    _vid = new Video(320, 240);
    _vid.attachCamera(_cam);
    _vid.filters = [new BlurFilter(10, 10, 1)];

    _bmpd = new BitmapData(320, 240, false);
    addChild(new Bitmap(_bmpd));

    _cbRect = new Sprite();
    addChild(_cbRect);

    _ball = new Sprite();
    _ball.graphics.beginFill(0x0000ff);
    _ball.graphics.drawCircle(0, 0, 40);
    _ball.graphics.endFill();
    addChild(_ball);

    makePaletteArrays();

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.addEventListener(MouseEvent.CLICK, onClick);

}

private function makePaletteArrays():void
{
    _red = new Array();
    _green = new Array();
    _blue = new Array();
    var levels:int = 4;
    var div:int = 256 / levels;
    for(var i:int = 0; i < 256; i++)
    {
        var value:Number = Math.floor(i / div ) * div;
        _red[i] = value << 16;
        _green[i] = value << 8;
        _blue[i] = value
    }
```

www.TopSage.com

```
        }

        private function onClick(event:MouseEvent):void
        {
            _color = _bmpd.getPixel(mouseX, mouseY);
        }

        private function onEnterFrame(event:Event):void
        {
            _bmpd.draw(_vid, new Matrix(-1, 0, 0, 1, _bmpd.width, 0));
            _bmpd.paletteMap(_bmpd, _bmpd.rect, new Point(), _red, _green, _blue);

            var rect:Rectangle = _bmpd.getColorBoundsRect(0xffffff, _color, true);
            _cbRect.graphics.clear();
            _cbRect.graphics.lineStyle(1, 0xff0000);
            _cbRect.graphics.drawRect(rect.x, rect.y, rect.width, rect.height);

            if(!rect.isEmpty())
            {
                var xCenter:Number = rect.x + rect.width / 2;
                var yCenter:Number = rect.y + rect.height / 2;
                _ball.x = xCenter / _bmpd.width * stage.stageWidth;
                _ball.y = yCenter / _bmpd.height * stage.stageHeight;
            }
        }
    }
}
```

在构造函数中将创建这个球，并增加到显示列表。在enterFrame处理程序中，首先查看颜色包围矩形是否为空，这意味着没有找到任何有指定颜色的像素。如果是这样，就根本不会移动球。如果没有做这个检查，一旦未找到任何像素，球会突然跳至左上角。如果确实得到一个矩形，则找到其中心，并得到中心相对于位图大小的比例。然后将球移动到场景中同样的相对位置。

再次申明，这只是跟踪颜色的一种方法。强烈推荐你再自己尝试处理位图的其他方法，特别是阈值方法、颜色通道、颜色变换，以及其他滤镜和混合模式。也可以不跟踪特定颜色，另一种策略是只跟踪亮暗区域。

接下来，我们来看一种更普遍的跟踪运动的方法。

3. 分析运动区

在上一个例子中，我们尝试根据一个特定物体的颜色或者亮暗（如果你已经完成了这个挑战）来跟踪物体。这一节中不打算跟踪一个特定物体的运动，而是要以某种方式抽取运动。

基本概念如下：如果没有运动，每一帧的位图都完全相同。因此，如果比较两个连续的帧，有不同像素的区域则指示了物体发生移动的区域。

这说明了两个问题。首先，我们需要两个位图：一个之前的位图和一个之后的位图。其次，需要某种方法来比较这两个位图，查看哪里有所不同。这是理所当然的。不过，如果你考虑需要循环处理各个位图的所有像素来完成比较，这里还有一个很实用的技巧：使用混合模式。

使用BitmapData的draw方法在其上绘制对象时，有多种选择。你已经看到可以使用一个矩阵在绘制时变换对象的形状和位置。矩阵后面还有一些可选的参数：下一个参数是ColorTransform，

www.TopSage.com

再下一个就是混合模式。

利用混合模式进行绘制会改变新绘制的像素对现有像素的影响。如果没有指定混合模式，新像素值将直接覆盖原来的值。目前为止我们都采用了这种做法。原来的图像会完全擦除，只留下新图像。但是如果使用混合模式，则允许你采用多种方法将两个图像混合在一起。对我们来讲，有一种名为difference的混合模式就很适用，这实际上就是比较两个图像中各像素的红、绿和蓝色通道，然后给出它们之间的差。比较两个图像中相同位置上的像素时，如果二者完全相同，那么对于每个通道而言差都是0，所得到的像素颜色将是黑色。对于两个图像中所有不完全相同的像素，所得到的像素则会有另外的某个值。这样一来，就能把问题简化为通过查找非黑色的像素来找到存在运动的区域。

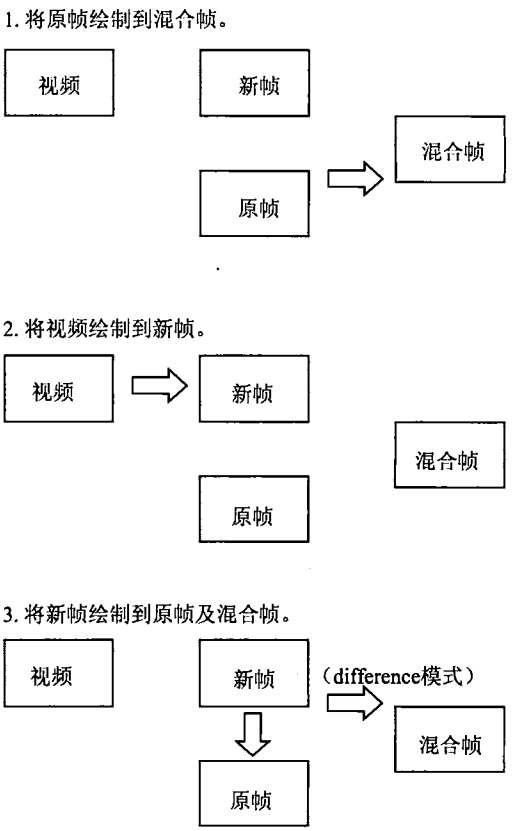


图5-7　采用差别混合模式结合原帧与新帧

尽管利用两个位图就可以完成这种工作，不过下面为简洁起见还是使用了3个位图：一个对应原来的帧，一个对应新的一帧，还有一个对应混合帧。图5-7大致展示了我们要做的工作。

新帧、原帧和混合帧都是BitmapData实例。首先我们将原帧采用无混合模式绘制到混合帧，这样会擦除其原有的内容。然后将视频绘制到新帧上，这里再一次使用无混合模式。

接下来，将新帧同样采用无混合模式绘制到原帧，下一次它就会表示原帧。最后将新帧绘制到混合帧，这里采用difference混合模式。

以下是相应的代码，可以在MotionTracking.as文件中看到：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.BlendMode;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.geom.Matrix;
```

www.TopSage.com

TopSage.com

```
import flash.media.Camera;
import flash.media.Video;

public class MotionTracking extends Sprite
{
    private var _cam:Camera;
    private var _vid:Video;
    private var _newFrame:BitmapData;
    private var _oldFrame:BitmapData;
    private var _blendFrame:BitmapData;


    public function MotionTracking()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        _cam = Camera.getCamera();
        _cam.setMode(320, 240, 15);
        _vid = new Video(320, 240);
        _vid.attachCamera(_cam);
        _vid.filters = [new BlurFilter(10, 10, 1)];

        _newFrame = new BitmapData(320, 240, false);
        _oldFrame = _newFrame.clone();
        _blendFrame = _newFrame.clone();
        addChild(new Bitmap(_blendFrame));

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onEnterFrame(event:Event):void
    {
        _blendFrame.draw(_oldFrame);
        _newFrame.draw(_vid, new Matrix(-1, 0, 0, 1, _newFrame.width, 0));
        _oldFrame.draw(_newFrame);
        _blendFrame.draw(_newFrame, null, null, BlendMode.DIFFERENCE);

    }
}
}
```

以上的大部分代码你现在应该都已经能理解了。这里创建了新帧位图，然后通过克隆复制该位图创建其余的帧。我们只是将最后一个增加到显示列表。在enterFrame处理程序中按图5-7中指定的顺序来完成绘制。

第一次运行时，可能除了一个黑色矩形什么也看不到。不过稍稍移动一点，你就会看到一些幽灵般的白色轮廓（参见图5-8）。这些轮廓就是两帧之间有差别的区域。

现在来看能否使用一个颜色包围矩形覆盖有变化的区域。采用前例中同样的方法创建_cbRect sprite。在enterFrame处理程序中，做法稍有不同：

```
private function onEnterFrame(event:Event):void
{
```

www.TopSage.com

```
        _blendFrame.draw(_oldFrame);
        _newFrame.draw(_vid, new Matrix(-1, 0, 0, 1, _newFrame.width, 0));
        _oldFrame.draw(_newFrame);
        _blendFrame.draw(_newFrame, null, null, BlendMode.DIFFERENCE);

        var rect:Rectangle = _blendFrame.getColorBoundsRect(0xffffff, 0x000000, false);

        _cbRect.graphics.clear();
        _cbRect.graphics.lineStyle(1, 0xff0000);
        _cbRect.graphics.drawRect(rect.x, rect.y, rect.width, rect.height);
    }
```

图5-8　利用一种difference混合模式结合两帧

注意这一次我们在查找颜色0x000000（黑色），但是最后一个参数指定为false，这说明要得到包围所有非黑像素的矩形。不过尝试运行时，你会发现这个矩形包围了整个位图（大多数情况下都是如此）。嗯，这可不太妙，所有那些看上去黑色的像素似乎并不真是全黑的。所以如果逐像素地比较，两个图像并不完全一样——只是很接近而已。如果足够近地看屏幕，也许看不出这些差别，对我们的眼睛来说像素都是黑的，但是实际上这些像素确实不等于0x000000。

下面从我们的位图工具箱中再取出另一个位图工具（threshold）这是比较复杂的方法之一，不过这也是功能最为强大的工具之一。要利用它完成我所希望的工作可不简单，在达到目的之前我总是要与它斗争一番，因此如果你觉得这个方法很难掌握，不要以为只有你这样想，很多人都有同样的感觉。

threshold方法的做法是，检查一个位图，将它的各个像素与一个值进行比较。这个比较几乎都是算术比较操作符：<, <=, ==, >=, >和!=。如果对一个特定像素的比较计算为true，则将该像素设置为你选择的一种颜色。还可以指定一个掩码，从而只查看单个颜色通道。

www.TopSage.com

掩码很有用，因为实际上不能直接比较两个完整的颜色值。例如，0x010000几乎为全黑，只是包含一点红色；而0x0000FF是100%的蓝色，比0x010000亮得多。但是计算时这个蓝色值（0x0000FF）将“小于”前面的近黑色（0x010000），因为从数字比较来讲这个值较小。threshold方法如下所示：

```
bitmapData.threshold(sourceBitmapData, sourceRect, destPoint, operation,
                     threshold, color, mask, copySource)
```

第一个参数sourceBitmapData是一个BitmapData，就是要检查这个位图的像素。其结果将绘制到调用threshold方法的BitmapData上。通常这是同一个BitmapData（这里就是如此），但是也可以检查一个位图，而将阈值结果绘制在另一个位图上。

sourceRect和destPoint确定了要检查的位图区域，以及结果将放置在哪里。这里使用了BitmapData的rect属性和一个新的Point，这表示它将检查整个位图，并把结果放在（0, 0）位置上。operation是一个包含>, >=, ==, <=, <,或!=的字符串。threshold是与每个像素比较的值。color是比较结果为true时为像素设置的值，另外mask用于指定一个特定的颜色通道。如果检查一个位图，再把它绘制在另一个位图上，则要使用copySource参数。倘若是这种情况，如果比较结果为false，可以选择将源位图的像素绘制在目标位图上。在这里，这个参数没有任何影响，因为我们只使用了一个位图。

在这种情况下，所有像素都非常接近于黑色，所以选择哪个通道来完成比较并不重要。可以选择红色通道，当然选择其他通道也完全可以。以下是修改后的enterFrame处理方法：

```
private function onEnterFrame(event:Event):void
{
    _blendFrame.draw(_oldFrame);
    _newFrame.draw(_vid, new Matrix(-1, 0, 0, 1, _newFrame.width, 0));
    _oldFrame.draw(_newFrame);
    _blendFrame.draw(_newFrame, null, null, BlendMode.DIFFERENCE);
    _blendFrame.threshold(_blendFrame, _blendFrame.rect, new Point(),
                          "<", 0x00330000, 0xff000000, 0x00ff0000, true);

    var rect:Rectangle = _blendFrame.getColorBoundsRect(0xffffff, 0x000000, false);

    _cbRect.graphics.clear();
    _cbRect.graphics.lineStyle(1, 0xff0000);
    _cbRect.graphics.drawRect(rect.x, rect.y, rect.width, rect.height);
}
```

所以，如果像素的红色通道（注意这里用0x00ff0000掩码指定红色）小于（<）0x00330000，则将其设置为黑色（0xff000000），否则就复制源位图（由于源位图与目标位图是同一个位图，所以是否复制并没有影响）。说得更简单一些：如果像素接近黑色，则将其设置为全黑色。

需要指出这里所有颜色都使用了32位值。这个阈值方法很有意思。即使使用不透明的位图，如果没有指定α通道，它也会绘制一个透明的像素。所以这里使用0xff000000而不只是0x000000来表示黑色。

尝试这个例子时，它会正常工作，会完全根据你的移动在你期望的位置绘制矩形。

那么到底要用这个移动的矩形做什么呢？重申一次，只有想不到，没有做不到。我首先想到

www.TopSage.com

大家网 TopSage.com

的是一种突围游戏，你可以在屏幕底线来回移动拍子来击球。不过也可以在屏幕左边或右边通过动作移动这个拍子。下面是一个相当粗略的例子，可以由此起步进一步完善：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.BlendMode;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;
    import flash.geom.Rectangle;
    import flash.media.Camera;
    import flash.media.Video;

    public class MotionTracking extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
        private var _newFrame:BitmapData;
        private var _oldFrame:BitmapData;
        private var _blendFrame:BitmapData;
        private var _cbRect:Sprite;
        private var _paddle:Sprite;


        public function MotionTracking()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _cam = Camera.getCamera();
            _cam.setMode(320, 240, 15);
            _vid = new Video(320, 240);
            _vid.attachCamera(_cam);
            _vid.filters = [new BlurFilter(10, 10, 1)];

            _newFrame = new BitmapData(320, 240, false);
            addChild(new Bitmap(_newFrame));
            _oldFrame = _newFrame.clone();
            _blendFrame = _newFrame.clone();

            _cbRect = new Sprite();
            addChild(_cbRect);

            _paddle = new Sprite();
            _paddle.graphics.beginFill(0xffffff);
            _paddle.graphics.drawRect(-100, -20, 200, 40);
            _paddle.graphics.endFill();
            _paddle.x = stage.stageWidth / 2;
            _paddle.y = stage.stageHeight - 50;
```

www.TopSage.com

```
            addChild(_paddle);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            _blendFrame.draw(_oldFrame);
            _newFrame.draw(_vid, new Matrix(-1, 0, 0, 1, _newFrame.width, 0));
            _oldFrame.draw(_newFrame);
            _blendFrame.draw(_newFrame, null, null, BlendMode.DIFFERENCE);
            _blendFrame.threshold(_blendFrame, _blendFrame.rect, new Point(), "<", 0x00330000,
                                  0xff000000, 0x00ff0000, true);
            var rect:Rectangle = _blendFrame.getColorBoundsRect(0xffffff, 0, false);
            _cbRect.graphics.clear();
            _cbRect.graphics.lineStyle(1, 0xff0000);
            _cbRect.graphics.drawRect(rect.x, rect.y,  rect.width, rect.height);

            if(!rect.isEmpty())
            {
                if(rect.x < _blendFrame.width / 2)
                {
                    _paddle.x -= 20;
                }
                else
                {
                    _paddle.x += 20;
                }
            }
        }
    }
}
```

确实相当简单。创建一个拍子sprite，在其中绘制一个形状。如果颜色包围矩形在中心的右侧，则将拍子移向右边。如果在左侧，则把拍子左移。是不是很棒？再次指出，这里只是一个大致框架来展示如何起步。要让它更漂亮更平滑，这个任务将作为练习留给你来完成。注意，这里我还决定不再将混合位图增加到显示列表，而是增加新的帧位图。这会让用户对于他们在帧中的位置有所了解。当然，这里使用了阈值方法和颜色包围矩形，还包括其他工作（如处理位图是否可见）。将这些内容隐藏，运动检测看起来就像魔法一般！由图5-9会让你有所认识。

4. 分析边缘

作为这个主题的最后一部分，下面重新创建一个流行的效果：检测一个图像的边缘，并在上面悬挂东西。这通常用于一些很受欢迎的交互性展示，在儿童博物馆之类的网站上经常会看到，屏幕上显示着孩子的图片，有雪落在上面，有蝴蝶在上面栖息，诸如此类。利用Flash也有很多成功的例子，其中最知名的可能是Grant Skinner（www.gskinner.com），不过下面并不讨论这个应用。

检测边缘有很多种方法，不过我们直接想到的是使用ConvolutionFilter类创建一个水平边缘检测滤镜。卷积（convolution）滤镜会检查一个位图，取各个像素，分别与一个周围像素网格或矩阵进行比较。每个像素的值会加权，并增加到一个总和，然后可能除以某个因子。其结果再应用到原像素。卷积矩阵在图像处理中广泛用于模糊、尖锐化、压纹、边缘检测、增强等效果。

www.TopSage.com

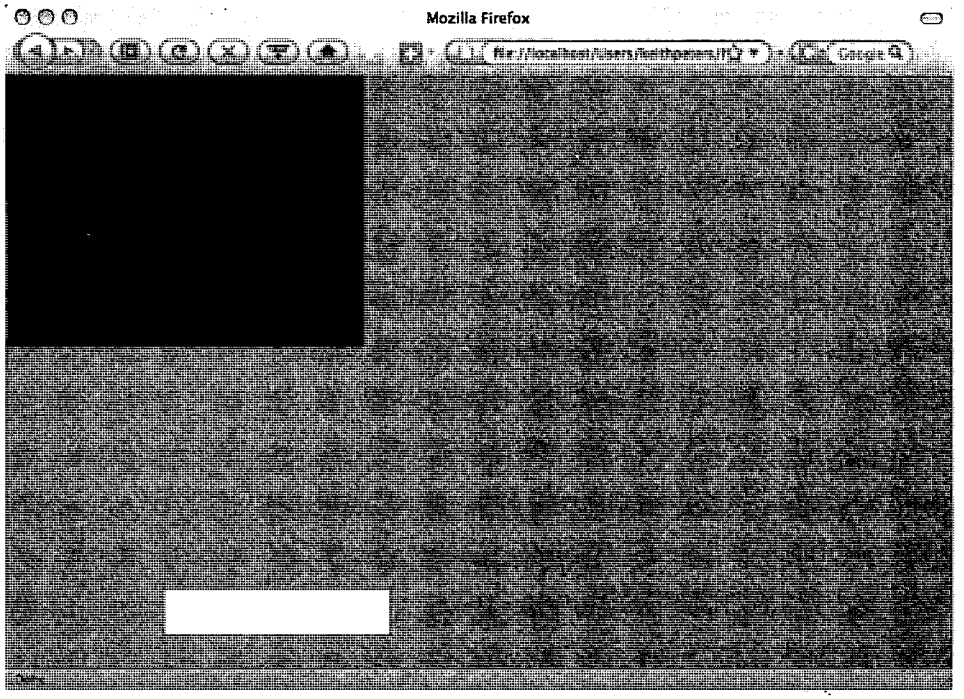


图5-9　利用手的运动控制一个物体

图5-10显示了模糊滤镜的基本设置。中心像素就是要检查的像素。3×3网格中的各个像素要增加到总和中，且权值为1。换句话说，其值增加到总和之前会乘以1。然后总和除以9，这个结果再赋至中心像素。其效果是将所有周围像素取平均，产生一种模糊效果。要得到更大程度的模糊，可以使用一个更大的网格。

模糊矩阵

1	1	1
1	1	1
1	1	1

增加所有像素，除以9，
将结果赋至中心像素

图5-10　模糊使用的卷积矩阵

如果对卷积矩阵感兴趣，在网上搜索就会得到大量有关的信息和示例，其中大多数都可以在ActionScript中直接使用。经过一番研究和试验，我得出了创建水平边缘检测滤镜的一些比较好的设置。ConvolutionFilter构造函数如下所示：

```
ConvolutionFilter(matrixX, matrixY, matrix, divisor)
```

实际上还有一些可选的参数，不过目前我们只需要以上这些参数。matrixX和matrixY参数都是Number，指定了矩阵的宽度和高度（或者是将包围所检查像素的网格）。matrix是一个权值数

www.TopSage.com

组，也就是各个像素值增加到总和之前与之相乘的权值。matrix数组的长度应当等于matrixX乘以matrixY。最后，divisor就是再次指定给原像素之前总和除以的那个值。所以，如果想要使用图5-10中所示的示例创建一个模糊滤镜，可以如下完成：

```
new ConvolutionFilter(3, 3, [1,1,1,1,1,1,1,1,1], 9);
```

这会建立一个3×3的矩阵，并将所有元素填为1，其结果除以9。很简单。

下面是我为水平边缘提出的卷积设置：

```
new ConvolutionFilter(1, 3, [0, 4, -4], 1);
```

这就更简单了。这会创建一个1×3的矩阵，将当前像素值乘以4，再将下面的像素乘以–4。divisor（除数）为1，所以应用到中心像素的总和正是当前得到的总和。再次说明，如果愿意，可以阅读更多材料来了解卷积矩阵如何工作。其效果是大多数像素会变暗，而将落在视觉水平边缘上的像素加亮。下面来看这在一个视频上的实际作用。具体的类如下，可以在EdgeTracking.as文件中找到：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.filters.ConvolutionFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;
    import flash.media.Camera;
    import flash.media.Video;

    public class EdgeTracking extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
        private var _bmpd:BitmapData;

        public function EdgeTracking()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _cam = Camera.getCamera();
            _cam.setMode(320, 240, 15);
            _vid = new Video(320, 240);
            _vid.attachCamera(_cam);
            _vid.filters = [new ConvolutionFilter(1, 3, [0, 4, -4]), new BlurFilter()];

            _bmpd = new BitmapData(320, 240, false);
            addChild(new Bitmap(_bmpd));

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

www.TopSage.com

```
        }

        private function onEnterFrame(event:Event):void
        {
            _bmpd.draw(_vid, new Matrix(-1, 0, 0, 1, _bmpd.width, 0));
        }
    }
}
```

除了ConvolutionFilter外，这里几乎没有什么新的内容。这里也应用了一个模糊滤镜让图像更平滑。其结果如图5-11所示。

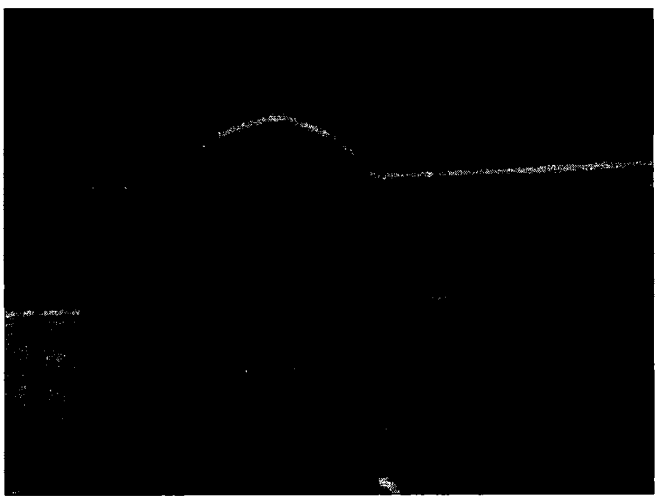


图5-11　创建水平边缘检测

可以清楚地看到，所有水平边缘都很清楚也很明亮，而图像的余下部分几乎是黑色的。（提示：这里提到“几乎是黑色”，应当把它作为一个线索，这说明接下来可能很快会看到使用一个阈值。）

那么对此可以做些什么呢？下面在我的头上落一些雪。首先，建立一个简单的类表示雪（见Snow.as文件）：

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

    public class Snow extends Sprite
    {
        public var vx:Number;
        public var vy:Number;

        public function Snow()
        {
            graphics.beginFill(0xffffff, .7);
            graphics.drawCircle(0, 0, 2);
```

www.TopSage.com

```
            graphics.endFill();

            vx = 0;
            vy = 1;
        }
        public function update():void
        {
            vx += Math.random() * .2 - .1;
            vx *= .95;
            x += vx;
            y += vy;
        }
    }
}
```

这个类会绘制一个小小的白色圆圈，具有x和y速度，还有一个更新自身的方法。更新意味着随机地改变x速度，并把这个速度增加到它的位置。很不错。

我们的计划是在每一帧上创建一个新的snow实例，把它增加到数组，并更新数组中的各个snow。不过，首先来查看snow是否恰好在边缘上。换句话说，该snow的（x, y）位置上的像素是否大于0？这里就要引入阈值——让那些“几乎为黑”的像素成为全黑。下面是修改后的代码：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.filters.ConvolutionFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;
    import flash.media.Camera;
    import flash.media.Video;

    public class EdgeTracking extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
        private var _bmpd:BitmapData;
        private var _flakes:Array;

        public function EdgeTracking()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _cam = Camera.getCamera();
            _cam.setMode(320, 240, 15);
            _vid = new Video(320, 240);
            _vid.attachCamera(_cam);
            _vid.filters = [new ConvolutionFilter(1, 3, [0, 4, -4]), new BlurFilter()];
```

www.TopSage.com

大家网 TopSage.com

```
            var vid2:Video = new Video(320, 240);
            vid2.attachCamera(_cam);
            vid2.scaleX = -1;
            vid2.x = 320;
            addChild(vid2);

            _bmpd = new BitmapData(320, 240, false);

            _flakes = new Array();

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _bmpd.draw(_vid, new Matrix(-1, 0, 0, 1, _bmpd.width, 0));
            _bmpd.threshold(_bmpd, _bmpd.rect, new Point(), "<",
                            0x00220000, 0xff000000, 0x00ff0000, true);

            var snow:Snow = new Snow();
            snow.x = Math.random() * _bmpd.width;
            addChild(snow);
            _flakes.push(snow);

            for(var i:int = _flakes.length - 1; i >= 0; i--)
            {
                snow = _flakes[i] as Snow;
                if(_bmpd.getPixel(snow.x, snow.y) == 0)
                {
                    snow.update();
                    if(snow.y > _bmpd.height)
                    {
                        removeChild(snow);
                        _flakes.splice(i, 1);
                    }
                }
            }
        }
    }
}
```

首先，创建一个_flakes数组保存所有Snow实例。

注意，我们还有另一个翻转的视频，将把这个视频增加到显示列表而不是_bmpd。正如运动检测示例中一样，这样用户就能够看到他们自己未处理的图像，而不是经过卷积和模糊处理后的图像。

如前所述，再应用一个阈值，使所有暗像素成为全黑。

每一帧上，我们创建了一个新的Snow实例，把它增加到显示列表和_flakes数组。然后循环处理这个数组，得到其中各Snow的一个引用。如果一个Snow所表示的像素等于0，则进行更新，使它随机地左移或右移一点，并下移一个像素。如果像素不是0（不是黑色），说明像素在一个水平边缘上，所以不做处理仍保留原样。

www.TopSage.com

最后，如果像素移出图像底边，则将其从显示列表和_flakes数组中删除。其结果如图5-12所示。

不能不承认，这个例子并没有太大用处，但挺有意思。你的孩子会兴致不减地一直玩这个游戏（这是我的切身经验）。不过，一旦了解了此类工作的原理，你肯定可以做出一些变化，提出你自己的想法。非常期待看到你的新作品。

图5-12 带雪花的视频

5.4 小结

本章中我们看到了使用声音和视频作为输入方法的一些试验性示例。使用摄像头或麦克风来影响Flash应用或游戏中的元素有很多种可能的方法，这里只是其中很少的几个例子。希望这至少能对你有所启发，因为在这方面还可以做的工作很多。重申一次，期待能看到你们的作品，对这里给出的例子做出大幅改进。

www.TopSage.com

第6章

高级物理：数值积分

在《Flash ActionScript 3.0 动画教程》中，我给出了一些基本的物理公式，用于Flash中的运动编程。主要可以总结为：每一帧上将加速度增加到速度，并将速度增加到位置。我很清楚这并不是最准确的物理公式，而且对此也做过很多免责声明。不过我也相信，对于大多数人来说使用这些公式足以很好地用Flash完成游戏和动画。

完成本书第一版之后，我发现我所描述的方法通常称为**欧拉积分**（Euler integration）。它确实相当不准确，也非常简单，但对于大多数非正式的应用来说已经足够好，并且得到了很广泛的使用。不过，既然欧拉积分不算太好，还有哪些更好的候选方法呢？本章至少会给出有关这个问题的一些答案。我们将介绍什么是数值积分，欧拉积分有什么不好，还会给出另外两种候选方法，并解释可能希望考虑使用这些方法的原因。

另外，为了让你在下一次晚宴时少一些尴尬，需要指出这里的欧拉得名于18世纪的物理学家和数学家欧拉（Leonhard Euler）。还要提一句，如果你在晚宴时不合时宜地谈到欧拉积分，尴尬可能是不可避免的。

6.1 数值积分以及为什么欧拉积分“不好”

首先，这里所说的“不好”是指不准确。换句话说，如果你要使用欧拉积分模拟一个根据不同作用力移动的物体，往往不能完美地反映该物体在真实世界中的实际运动。不过，对于一些小游戏、动画、非科研性模拟等来说它可能已经足够接近，大多数人也许都看不出有什么问题。不过，如果你确实希望或需要更高的准确性，那么欧拉积分将无法达到要求。

为什么呢？这是因为：运动、速度、质量和加速度的公式在几个世纪之前就已经得出了（这要感谢牛顿），这些公式一直都用得很好。如果上过高中代数，你可能会记得这样一个问题：一个男孩站在距地面50英尺[①]的屋顶上，以30英尺每秒的速度向上抛球……然后要求你计算球何时到达地面，或者某一时刻球的高度。你会使用下面的公式：其中t是时间（以秒为单位），v_0是初始速度，h_0是初始高度，y是给定时间的高度：

```
y = -16t² + v0 * t + h0
```

如果遵循这个公式来计算肯定没问题，这是完全正确的。那么为什么不干脆对所有运动都使

① 1英尺=12英寸=0.3048米。——编者注

www.TopSage.com

用这个公式，都用它来处理呢？因为这里只描述了以一个弧度移动的单个物体在一个轴上的运动，除了重力外没有其他作用力作用在该物体上。如果再增加另一个或者两个方向上的作用力，那么就要考虑空气阻力、反弹力、其他物体和碰撞，以及所有其他作用力（如风的影响），因此情况就会骤然变得极其复杂。

一种更为实用的方法是：按离散间隔取物体的一个快照，得到这一时刻作用在该物体上的所有作用力，计算其速度，并更新它的位置，这个过程称为**积分**（integration）。积分的理想目标如图6-1所示。

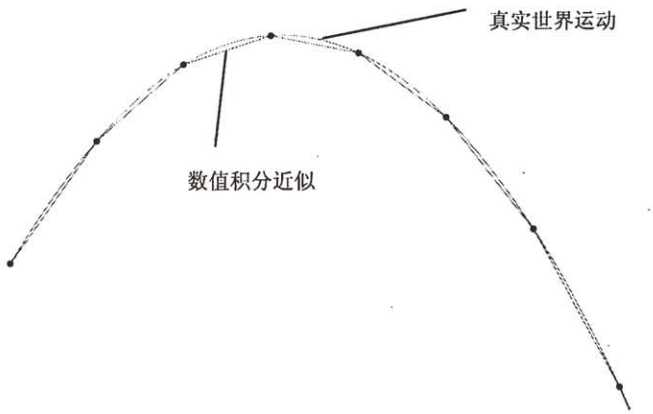


图6-1 真实世界运动和积分近似（理想）

上面的弧线是物体实际走过的平滑曲线。下面的分段折线表示了希望利用积分得到的速度和位置。各种积分方法就是试图画出与该曲线最匹配的一系列直线。折线的段数越多，轨迹就越接近弧线。

问题是，真实世界中的物理作用力是连续作用在物体上的，一个物体上的作用力会持续改变其位置和速度，而不是像这样离散地跳跃。例如，重力会不断地作用在一个下落的物体上，增大其速度，随时间平滑地改变它的位置。如果对一个物体应用重力，并且只是每帧更新一次速度和位置，就会丢失两帧之间所有时刻的重力和速度效果。所以，更新的频度越大，就越准确，但是总是会存在一些误差。

欧拉积分对这种情况的解决方案就是忽略误差，所以会很不准确。这应该是最简单的解释了。图6-2显示了以1秒间隔计算的欧拉积分的实际结果，以突出显示存在的误差。

再次说明，如果减少这个间隔，会得到更准确的结果，但是只有在无限增加间隔且间隔无限小的情况下才能真正做到准确。而这往往又做不到。

另外，欧拉积分还会遇到不稳定的情况。弹簧就很容易发生这种情况。积分的不准确性有时会导致达到某个临界点，在此之后速度将一直增加，以至于一切崩溃。

不过，既然欧拉积分如此不好，还有什么候选方法呢？这里将要介绍的两种方法分别是**龙格-库塔积分**（Runge-Kutta integration）和**Verlet积分**（Verlet integration）。与欧拉积分一样，这些方法也都是以提出相应方法的人的名字命名的：Carl Runge、Martin Wilhelm Kutta和Loup Verlet。你会看到，另外这两种方法分别着重不同领域：龙格-库塔积分用于需要更高准确度的情况，而Verlet积分通常用于计算机图形学领域，来创建一种“玩偶式物理运动”的反向运动。如果你现在对此还不太了解，不用担心。介绍相关内容时，你会发现这些方法的使用相当容易，而且可以创建一些确实很不错的效果。首先来研究龙格-库塔积分。

www.TopSage.com

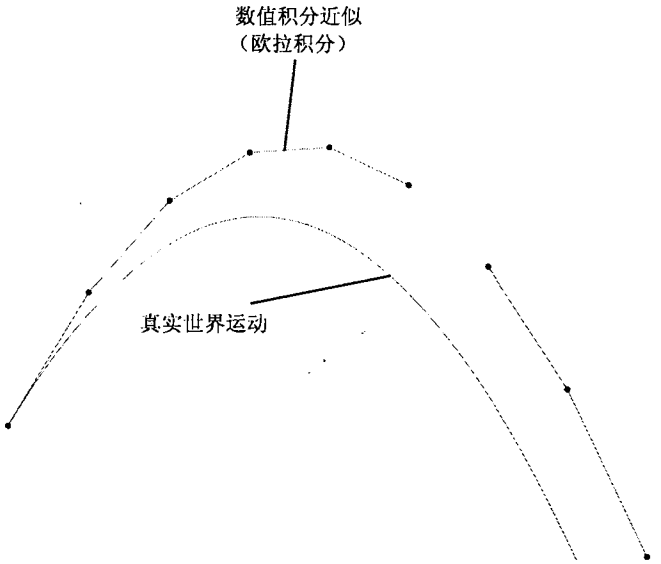


图6-2　真实世界运动和欧拉积分

6.2　龙格-库塔积分

6.1节曾经讨论过，欧拉积分的缺陷就在于它试图用离散步来完成加速度、速度和位置的积分。因为这些元素的变化是持续性的，所以这种积分只能得到一个近似结果，这可能很不准确。龙格-库塔积分并没有完全解决这种时间步不准确问题，不过它加入了额外的一些计算，这些计算专门用来得到对位置和速度的一个更好的估计。一定要很清楚这一点，即使你正在使用龙格-库塔积分，也并不意味着会得到一个完全准确的模拟，这只是一个更准确的模拟而已。

那么龙格-库塔积分有多准确呢？这个问题我很难回答，只能说对于你要用Flash完成的所有模拟来说，这种方法都能完全提供所需的准确性。如果你的应用确实极其挑剔，可能最开始就不应该使用Flash来实现。

我想我第二次听说龙格-库塔是在美国波士顿的一个Flash用户组会议上，在那里James Battat展示了他与一个同事使用ActionScript 2.0和龙格-库塔积分完成的一系列物理模拟，这是为他们在哈佛大学讲授的一门机械系统入门物理课程（属于Presidential Instructional Technology Fellowship Summer计划）而实现的。这些模拟很好地证明了龙格-库塔积分的准确性，促使我想要进一步了解这种方法。

很自然地，这里又会遇到我们的“老朋友”——技术权衡。龙格-库塔更准确，但是这种准确性也会带来更多计算的开销，这意味着更多的CPU占用，执行速度更慢，等等。所以不要只是因为龙格-库塔“更好”、更准确或者只是因为所有“新新人类”都这么做你就贸然搭上龙格-库塔这架马车。要确保你确实需要更高的准确性。尽管我会用半章的篇幅讨论这种方法，但是可以坦白地告诉你，对于Flash中可能遇到的几乎所有情况都不需要那么高的准确性，如果没有更充分的理由，使用这种方法只是在加大CPU的负担。

www.TopSage.com

警告到此为止。下面来看龙格-库塔如何工作。从现在开始，为简单起见我们把这种方法简称为RK。

RK确实仍在完成时间步处理，但是它并不是每一步都盲目地将加速度加至速度之上，再把速度加至位置之上，然后一整天地如此调用。它的做法是在一个时间步中对曲线多次采样，从而对这一时刻曲线的具体情况有所了解，然后取这些样本的加权平均值来得到最接近曲线的一点。这个想法不算太复杂，下面我们将更深入了解一些内容，看看它到底如何实现。

主要有两种RK：**龙格-库塔二阶积分**（简称为RK2）和**龙格-库塔四阶积分**（简称为RK4）。这里的数字表示中间采样的次数。RK2会在每个时间步采样两个值，而RK4在每个时间步采样4个值。你听到最多的可能是RK4，这种方法很好，原因在于如果你在谈论“龙格-库塔四阶积分”，会让别人认为你很聪明。

我们首先讨论RK2，因为它理解起来要简单一些，而且也可以很好地延伸到RK4。你可能还会看到这种技术的其他一些名称：“改进的欧拉积分”（improved Euler）、“中点欧拉积分”（midpoint Euler），或者只是简单的“中点法”（the midpoint method）。这些都是同一个方法。从这些别名就可以看出，RK2与欧拉积分方法并没有太大差别，实际上就是要利用一个中点来完成某些工作。

其策略是计算物体在当前状态下的加速度、速度和位置（基本欧拉积分）；然后计算应用了所计算加速度和速度之后的状态，对二者取平均，再应用该结果得到物体的状态。换句话说，欧拉积分的做法是试图根据一个点猜测出一段时间的全部速度，而RK2则是取这个时间段起始点和结束点的速度，对二者取平均值来估计这段时间的速度。同样地，这也不是100%正确，但是比欧拉积分要准确得多。图6-3给出了欧拉积分与RK2比较的结果。

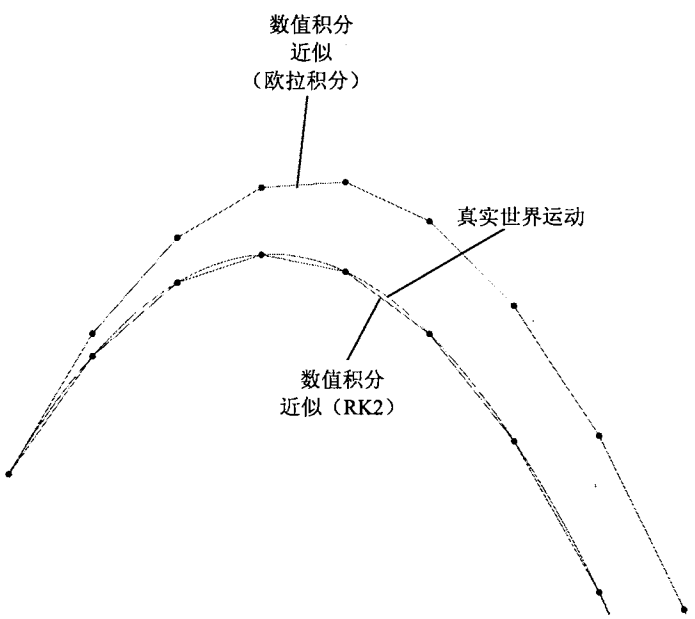


图6-3 真实世界运动，欧拉积分与RK2

www.TopSage.com

大家网 TopSage.com

基于这种分辨率，在这个时间段内从RK2图中确实看不到任何误差。这很不错——可能对于你要用Flash完成的工作来说已经远远足够了。

当然，如果还觉得不够好，就可以使用RK4，它会根据时间段内的4个采样平均得出曲线的曲度。我不打算具体阐释，因为基于这个分辨率并不能看出RK4与RK2的差别。不过，RK4更为准确，这是毋庸置疑的。

6.2.1　基于时间的运动

好了，理论已经够多了。来看一些代码！

别着急，我会给出代码的。不过先来再做一些解释。本节使用的所有代码都是基于时间的运动代码，而不是基于帧的运动代码。这个主题在《Flash ActionScript 3.0 动画教程》的第19章中介绍过，不过这里再做一点简要说明。

在很多基于Flash的简单的运动代码中，都有速度和加速度值，我们要在每一帧将这些速度和加速度值应用到对象。我们认为物体移动是以像素为单位，而时间间隔为一帧。所以速度就是“像素每帧”而不是“米每秒”或“公里每小时”。而且，要根据最后动画中的效果针对重力、速度和不同作用力调整这些值。但是如果要完成所有这些额外工作来得到一个更准确的结果，我们可能希望使用标准度量，至少时间需要标准度量。所以我们将记录真实的毫秒数，并根据实际过去的秒数来完成更新。对象会以其在真实世界中的相同速度移动，这样我们能相当准确地进行度量。

为此，必须使用flash.utils.getTimer方法来度量自最后一次更新以来逝去的时间。这个逝去时间可用于所有标准运动公式，甚至包括欧拉公式。例如，速度的欧拉公式是*新位置*＝*原位置*＋*速度*×*逝去时间*。在基于帧的动画里没有用到逝去时间的唯一原因在于：你使用的是像素每帧——更新每一帧，逝去时间就是1。速度乘以1就等于速度，所以只需要*新位置*＝*原位置*＋*速度*。

还有几点需要指出。首先是位置、速度和加速度都存储在Point对象中。Point类包含x和y属性，所以不是用*vx*和*vy*表示速度，我们只有一个速度变量。另外我们还采用了同样的方式跟踪位置——所有计算完成后再将它赋至显示对象。最后一点，并不是把重力直接加至速度之上，我们将创建一个acceleration方法，它将返回一个点，其中包含*x*轴和*y*轴上的加速度值。现在来说，在*x*轴上只是0，*y*轴上是重力值，不过如果开始增加更复杂的作用力，可以利用这个方法做更多工作。在这些情况下，所得到的加速度可能会受到对象位置和速度的影响，所以我们将把对象的位置和速度作为参数传入acceleration方法。

为了在我们所熟悉的设置中习惯所有这些做法，先来处理欧拉方法。以下是相应的类，可以从本书下载页面（www.friendsofed.com）的Euler.as文件得到这个类：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.Point;
    import flash.utils.getTimer;

    public class Euler extends Sprite
```

www.TopSage.com

```
{
    private var _ball:Sprite;
    private var _position:Point;
    private var _velocity:Point;
    private var _gravity:Number = 32;
    private var _bounce:Number = -0.6;
    private var _oldTime:int;
public function Euler()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    _ball = new Sprite();
    _ball.graphics.beginFill(0xff0000);
    _ball.graphics.drawCircle(0, 0, 20);
    _ball.graphics.endFill();
    _ball.x = 50;
    _ball.y = 50;
    addChild(_ball);

    _velocity = new Point(100, 0);
    _position = new Point(_ball.x, _ball.y);

    _oldTime = getTimer();
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    var time:int = getTimer();
    var elapsed:Number = (time - _oldTime) / 1000;
    _oldTime = time;

    var accel:Point = acceleration(_position, _velocity);
    _position.x += _velocity.x * elapsed;
    _position.y += _velocity.y * elapsed;
    _velocity.x += accel.x * elapsed;
    _velocity.y += accel.y * elapsed;

    // check if object has gone past any boundaries
    // and bounce off them.
    if(_position.y > stage.stageHeight - 20)
    {
        _position.y = stage.stageHeight - 20;
        _velocity.y *= _bounce;
    }
    if(_position.x > stage.stageWidth - 20)
    {
        _position.x = stage.stageWidth - 20;
        _velocity.x *= _bounce
    }
    else if(_position.x < 20)
    {
        _position.x = 20;
```

www.TopSage.com

```
            _velocity.x *= _bounce;
        }

            _ball.x = _position.x;
            _ball.y = _position.y;
        }

        private function acceleration(p:Point, v:Point):Point
        {
            return new Point(0, _gravity);
        }
    }
}
```

这里创建了一个球，设置其位置和速度值，并用getTimer方法记录当前时间，这会返回动画已经运行的时间（毫秒数），将它保存在_oldTime里。在每一帧上再次调用这个方法。原时间与新时间之差就是自上一帧以来的逝去时间。将它除以1 000可以得到秒数而不是毫秒数。确保重置_oldTime，以保证下一次它是正确的。

然后调用acceleration方法来得到对象所受的作用力。再次说明，目前只有重力。

将速度加至球的位置之前，现在将它乘以逝去时间。将加速度加至速度之前做同样的处理。因此，两帧之间的逝去时间越长，加速度和速度乘以的系数就越高，球移动的幅度也越大。两帧之间的逝去时间越少，则移动的幅度也越小。实际上可以采用多种不同的帧速率来运行这个动画，观察同一个运动。当然，帧速率越低，动画就越不连贯，帧速率越高相应地就越平滑，不过在相同时间内球在屏幕上移动的距离是相同的。可以试试看，将stage.frameRate设置为10，再设置为100。尽管设置为100时球的移动看起来平滑得多，但是球仍以相同的速度沿着相同的路径移动。

这里还加入了反弹，使得屏幕上的动画能持续较长时间，以便你看到发生了什么。有关内容稍后再做介绍。

现在你可能会这样想："还不错，不过球下落得太慢了，这可不太现实。"我们将重力设置为32，这正是真实世界中的重力大小：32英尺每秒，记住是每秒。换句话说，一个下落的物体每下落一秒速度就会增加32英尺每秒。所以，如果单位是英尺和秒，而且不做任何改变地应用到一个sprite的位置，我们的比例是1个像素 = 1英尺。因此想象一个40英尺的圆球从一个800或900英尺高的建筑物顶端下落——也许这看起来会更真实！

你可能希望进行等比缩放，假设100个像素等于1英尺。我们可以用一个变量来设置这个比例，并且在基于英尺的位置和基于像素的位置间进行切换时应用这个比例变量：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.Point;
    import flash.utils.getTimer;
public class Euler extends Sprite
{
    private var _ball:Sprite;
    private var _position:Point;
```

www.TopSage.com

TopSage.com

```
private var _velocity:Point;
private var _gravity:Number = 32;
private var _bounce:Number = -0.6;
private var _oldTime:int;
private var _pixelsPerFoot:Number = 100;

public function Euler()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    _ball = new Sprite();
    _ball.graphics.beginFill(0xff0000);
    _ball.graphics.drawCircle(0, 0, 20);
    _ball.graphics.endFill();
    _ball.x = 50;
    _ball.y = 50;
    addChild(_ball);

    _velocity = new Point(10, 0);
    _position = new Point(_ball.x / _pixelsPerFoot,  _ball.y / _pixelsPerFoot);

    _oldTime = getTimer();
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    var time:int = getTimer();
    var elapsed:Number = (time - _oldTime) / 1000;
    _oldTime = time;

    var accel:Point = acceleration(_position, _velocity);
    _position.x += _velocity.x * elapsed;
    _position.y += _velocity.y * elapsed;
    _velocity.x += accel.x * elapsed;
    _velocity.y += accel.y * elapsed;

    if(_position.y > (stage.stageHeight - 20) / _pixelsPerFoot)
    {
        _position.y = (stage.stageHeight - 20) / _pixelsPerFoot;
        _velocity.y *= _bounce;
    }
        if(_position.x > (stage.stageWidth - 20) / _pixelsPerFoot)
        {
            _position.x = (stage.stageWidth - 20) / _pixelsPerFoot;
            _velocity.x *= _bounce
        }
        else if(_position.x < 20 / _pixelsPerFoot)
        {
            _position.x = 20 / _pixelsPerFoot;
            _velocity.x *= _bounce;
        }
```


www.TopSage.com

```
            _ball.x = _position.x * _pixelsPerFoot;
            _ball.y = _position.y * _pixelsPerFoot;
        }

        private function acceleration(p:Point, v:Point):Point
        {
            return new Point(0, _gravity);
        }
    }
}
```

现在有一个4或5英寸的圆球从8或9英尺高的位置下落。（注意，这里对初始速度也做了一点调整。）这与你的预期可能更为接近。

以上就是一个基于时间的环境中的欧拉积分。下面再来看龙格-库塔积分。

6.2.2 编写龙格-库塔二阶积分（RK2）代码

先来回忆RK2的步骤，计划是在时间步开始时和结束时计算加速度和速度，然后取二者之间的某种平均值。

下面用伪代码来说明。

首先，在时间步开始时计算加速度，然后计算位置和速度，这与之前使用欧拉积分的做法完全相同。不过，这一次我们将把新位置和速度存储在不同的变量中：

```
// position1 is current position of object
// velocity1 is current velocity of object
acceleration1 = acceleration(position1, velocity1)

position2 = position1 + velocity1 * time
velocity2 = velocity1 + acceleration1 * time
```

现在position2和velocity2表示这个时间步结束时对象所在的位置和速度。接下来需要计算该时间步结束时的加速度：

```
acceleration2 = acceleration(position2, velocity2)
```

以下是RK2的关键部分，要取这两个状态下速度和加速度的平均值：

```
position1 += (velocity1 + velocity2) / 2 * time
velocity1 += (acceleration1 + acceleration2) / 2 * time
```

我们的基本做法是取开始时和结束时的速度，取其平均值，并乘以逝去时间，得到改变的位移，再把它加到对象的当前位置上，就是改变后的位置。取开始和结束时加速度的平均值，乘以逝去时间，再把它加到当前速度上，就是改变后的速度。

这就是RK2！下面来看具体的代码：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.Point;
```

www.TopSage.com

```
import flash.utils.getTimer;

public class RK2 extends Sprite
{
    private var _ball:Sprite;
    private var _position:Point;
    private var _velocity:Point;
    private var _gravity:Number = 32;
    private var _bounce:Number = -0.6;
    private var _oldTime:int;
    private var _pixelsPerFoot:Number = 10;


    public function RK2()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        _ball = new Sprite();
        _ball.graphics.beginFill(0xff0000);
        _ball.graphics.drawCircle(0, 0, 20);
        _ball.graphics.endFill();
        _ball.x = 50;
        _ball.y = 50;
        addChild(_ball);

        _velocity = new Point(10, 0);
        _position = new Point(_ball.x / _pixelsPerFoot, _ball.y / _pixelsPerFoot);
        _oldTime = getTimer();
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        var time:int = getTimer();
        var elapsed:Number = (time - _oldTime) / 1000;
        _oldTime = time;

        var accel1:Point = acceleration(_position, _velocity);

        var position2:Point = new Point();
        position2.x = _position.x + _velocity.x * elapsed;
        position2.y = _position.y + _velocity.y * elapsed;

        var velocity2:Point = new Point();
        velocity2.x = _velocity.x + accel1.x * elapsed;
        velocity2.y = _velocity.y + accel1.x * elapsed;

        var accel2:Point = acceleration(position2, velocity2);

        _position.x += (_velocity.x + velocity2.x) / 2 * elapsed;
        _position.y += (_velocity.y + velocity2.y) / 2 * elapsed;

        _velocity.x += (accel1.x + accel2.x) / 2 * elapsed;
```

www.TopSage.com

大家网 TopSage.com

```
            _velocity.y += (accel1.y + accel2.y) / 2 * elapsed;

            if(_position.y > (stage.stageHeight - 20) / _pixelsPerFoot)
            {
                _position.y = (stage.stageHeight - 20) / _pixelsPerFoot;
                _velocity.y *= _bounce;
            }
            if(_position.x > (stage.stageWidth - 20) / _pixelsPerFoot)
            {
                _position.x = (stage.stageWidth - 20) / _pixelsPerFoot;
                _velocity.x *= _bounce
            }
            else if(_position.x < 20 / _pixelsPerFoot)
            {
                _position.x = 20 / _pixelsPerFoot;
                _velocity.x *= _bounce;
            }

            _ball.x = _position.x * _pixelsPerFoot;
            _ball.y = _position.y * _pixelsPerFoot;
        }
        private function acceleration(p:Point, v:Point):Point
        {
            return new Point(0, _gravity);
        }
    }
}
```

如果已经理解上述解释，应该可以看到这段代码能很好地完成工作。现在运行这个文件，看看与使用欧拉积分的版本相比是不是更好，更真实！什么？对你来说一切照旧？嗯，其实对我来说也是如此，不过确实还是稍稍有点差异的。这就是为什么我曾说过对于需要用Flash完成的大多数工作来说，欧拉积分已经足够好。另外要知道图6-3是用一秒的时间步计算的，所以这有些过于夸张。如果你的Flash动画运行速度为每秒24帧，会有一个小得多的时间步，欧拉积分的效果就会与RK2非常接近。

不过还要认识到，这只是一个非常简单的模拟。如果开始增加其他作用力，并让模拟持续几秒以上，你就会看到更多误差。不论怎样，如果你确实需要保证准确性，现在应该知道如何更充分地发挥Flash物理特性的作用了。

不过请等等！还有更好的，不只是RK2，我们还需要讨论RK4！

6.2.3　编写龙格-库塔四阶积分（RK4）代码

RK4是数值积分领域的重要一员。如果人们谈到“龙格-库塔”，很可能指的就是RK4。如果你对RK2的原理已经有了深入的了解，理解RK4应该也不成问题。我们做的几乎是同样的事情，不过这里不只是对时间段的开始和结束点采样，而是会对总共4个点采样，并取其平均值。

不过，在RK4中取平均值的做法稍有不同。下面来看相应的伪代码。这个伪代码很长，所以这里简化了变量名：

www.TopSage.com

```
// pos1 is current position of object
// vel1 is current velocity of object
acc1 = acceleration(pos1, vel1)

pos2 = pos1 + vel1 / 2 * time
vel2 = vel1 + acc1 / 2 * time
acc2 = acceleration(pos2, vel2)

pos3 = pos1 + vel2 / 2 * time
vel3 = vel1 + acc2 / 2 * time
acc3 = acceleration(pos3, vel3)

pos4 = pos1 + vel3 * time
vel4 = vel1 + acc3 * time
acc3 = acceleration(pos4, vel4)

pos1 += (vel1 + vel2 * 2 + vel3 * 2 + vel4) / 6 * time
vel1 += (acc1 + acc2 * 2 + acc3 * 2 + acc4) / 6 * time
```

注意这里的第1阶和第4阶值保持不变，而第二阶和第三阶值计算时会除以2，取平均时再乘以2。这会对曲线的端点和中间点提供一个特殊的加权。换句话说，并不是所有4个采样点毫无区别地相加再除以4，这里的做法是中间两个点会加倍，而总和要除以6。我确实理解它的原理吗？不尽然，不过这是龙格和库塔提出的，他们比我要聪明得多。我认为他们肯定也花了很长时间才提出这个算法，所以尽管我的理解不算太清楚但也还说得过去。重要的是，基于我的理解，完全能够让我在ActionScript中充分利用这种方法。接下来，深呼吸，请看以下具体的代码：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.Point;
    import flash.utils.getTimer;

    public class RK4 extends Sprite
    {
        private var _ball:Sprite;
        private var _position:Point;
        private var _velocity:Point;
        private var _gravity:Number = 32;
        private var _bounce:Number = -0.6;
        private var _oldTime:int;
        private var _pixelsPerFoot:Number = 10;


        public function RK4()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _ball = new Sprite();
            _ball.graphics.beginFill(0xff0000);
```

www.TopSage.com

```
        _ball.graphics.drawCircle(0, 0, 20);
        _ball.graphics.endFill();
        _ball.x = 50;
        _ball.y = 50;
        addChild(_ball);

        _velocity = new Point(10, 0);
        _position = new Point(_ball.x / _pixelsPerFoot, _ball.y / _pixelsPerFoot);

        _oldTime = getTimer();
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onEnterFrame(event:Event):void
    {
        var time:int = getTimer();
        var elapsed:Number = (time - _oldTime) / 1000;
        _oldTime = time;

        var accel1:Point = acceleration(_position, _velocity);

        var position2:Point = new Point();
        position2.x = _position.x + _velocity.x / 2 * elapsed;
        position2.y = _position.y + _velocity.y / 2 * elapsed;

        var velocity2:Point = new Point();
        velocity2.x = _velocity.x + accel1.x / 2 * elapsed;
        velocity2.y = _velocity.y + accel1.x / 2 * elapsed;

        var accel2:Point = acceleration(position2, velocity2);

        var position3:Point = new Point();
        position3.x = _position.x + velocity2.x / 2 * elapsed;
        position3.y = _position.y + velocity2.y / 2 * elapsed;

        var velocity3:Point = new Point();
        velocity3.x = _velocity.x + accel2.x / 2 * elapsed;
        velocity3.y = _velocity.y + accel2.y / 2 * elapsed;

        var accel3:Point = acceleration(position3, velocity3);

        var position4:Point = new Point();
        position4.x = _position.x + velocity3.x * elapsed;
        position4.y = _position.y + velocity3.y * elapsed;

        var velocity4:Point = new Point();
        velocity4.x = _velocity.x + accel3.x * elapsed;
        velocity4.y = _velocity.y + accel3.y * elapsed;

        var accel4:Point = acceleration(position4, velocity4);

        _position.x += (_velocity.x + 2 * velocity2.x + 2 * velocity3.x +
                        velocity4.x) / 6 * elapsed;

        _position.y += (_velocity.y + 2 * velocity2.y + 2 * velocity3.y +
```

www.TopSage.com

TopSage.com

```
                    velocity4.y) / 6 * elapsed;
            _velocity.x += (accel1.x + 2 * accel2.x + 2 * accel3.x +
                    accel4.x) / 6 * elapsed;

            _velocity.y += (accel1.y + 2 * accel2.y + 2 * accel3.y +
                    accel4.y) / 6 * elapsed;

            if(_position.y > (stage.stageHeight - 20) / _pixelsPerFoot)
            {
                _position.y = (stage.stageHeight - 20) / _pixelsPerFoot;
                _velocity.y *= _bounce;
            }
            if(_position.x > (stage.stageWidth - 20) / _pixelsPerFoot)
            {
                _position.x = (stage.stageWidth - 20) / _pixelsPerFoot;
                _velocity.x *= _bounce
            }
            else if(_position.x < 20 / _pixelsPerFoot)
            {
                _position.x = 20 / _pixelsPerFoot;
                _velocity.x *= _bounce;
            }

            _ball.x = _position.x * _pixelsPerFoot;
            _ball.y = _position.y * _pixelsPerFoot;
        }

        private function acceleration(p:Point, v:Point):Point
        {
            return new Point(0, _gravity);
        }
    }
}
```

我想如果你只是因为RK4很“酷”才想要使用这种方法，那么看到这段代码后你可能会放弃这种想法。不错，这段代码很长，不过，如果你对伪代码完成的工作已经有了基本的认识，理解这个代码应该不成问题。

再次说明，如果能看出这个方法与RK2或者欧拉方法之间的差别，说明你的眼力比我好。出于这个原因，我不再给出更深入的示例。另外，要创建真正准确的模拟，RK4还只是冰山一角，有关内容将在6.5节介绍。

6.2.4 薄弱环节

目前，球运动的基本代码与使用数值积分得到的结果基本上同样准确。不过，要作为一个准确的运动模型，现在这个文件本身没有多大用处。这是因为我们使用的反弹机制完全是虚构的。比欧拉积分更糟糕，这个反弹方法纯粹作为一个看上去效果不错的简易快速方法而开发。不过，从物理角度讲，它根本不准确。《Flash ActionScript 3.0 动画教程》中的大多数其他运动代码也是如此：磨擦、碰撞反应、巨型物体重力、弹跳，等等。其中大多数都基于真实世界的物理公式，

但是几乎所有运动代码都包含一些简化或技巧，以使编写代码更容易一些，另外对CPU的压力也更小一些。

如果你不嫌麻烦要自己编写RK4，并且为了保证准确性没有做任何额外的简化，那么实现模拟的其他部分时也要保证准确，不要因为其他部分不准确而破坏你在RK4实现中付出的劳动。遗憾的是，我不能为了保证绝对准确性而重写《Flash ActionScript 3.0 动画教程》的所有内容，当然这个工作也不可能仅用一章就能完成，所以就得靠你自己了。不过在一本严谨的大学物理课本里应该能找到你需要的所有内容。

6.2.5　龙格-库塔小结

尽管我曾多次警告，但不一定非要使用龙格-库塔积分，不过这并不是轻视或贬损这种方法。对于准确模拟来说它是一个很有价值的工具，在编程领域中可以算是高质量物理运动代码的标准。除了比欧拉方法更准确以外，它也更为稳定，所以如果你的弹簧运动一直有问题，可能就会考虑使用RK。我不认为这是大多数ActionScript开发人员每天都需要使用的方法，但确实希望本章能对那些有此需求的人有所帮助。

下面进入本章的第二部分，我们将介绍另一种数值积分形式，相信你会发现这种方法极其有用，而且使用起来很有意思，这就是Verlet积分。

6.3　Verlet积分

Verlet（读作ver-lay）积分最初是作为一种模拟分子交互的方法来开发的。在这种情况下，存在一些彼此交互的粒子，它们相互之间有固定的关系。对于每个粒子来说，可能有多个其他粒子将它朝某个方向推拉，还会有其他作用力作用在这个粒子上来改变它的位置和速度。对于这种情况，现有的积分形式可能会变得太过复杂而且太不稳定，所以设计了一种更高效、更稳定的解决方案：Verlet积分。

在当前的软件领域中，Verlet积分通常用于创建“玩偶式物理运动”系统或编程式角色动画。这种技术由Thomas Jakobsen于2003年所作的一篇名为*Advanced Character Physics*的文章而得以推广，这篇文章仍可以从www.gamasutra.com得到（顺便说一句，这是一个有关游戏、物理运动和其他编程相关知识的绝佳资源）。本节的代码主要是基于这篇文章中描述的系统来建立（当然已进行修改以便用于ActionScript 3.0）。

由于Verlet积分与龙格-库塔的优势（超精准性）不同，因此我们再来讨论基于帧的动画，这只是为了让问题变得更为简单和清晰（Verlet积分完全可以应用于基于时间的系统）。

Verlet积分的关键特性之一在于：它不会显式地存储一个对象的速度，实际上只会存储对象的上一个位置。这样一来，需要知道对象的速度时，只需将其当前位置减去上一个位置。因此，每次移动一个对象时，也就是在改变它的速度。这会简化很多问题。假设一个对象的x坐标为100，将它移动到110。下一次画面更新时，你会看到它自最后一次更新以来向右移动了10个像素，并取值10作为其x轴上的速度。以后在每次更新时，原x仍在当前x左边10个像素的位置上，所以对象会在屏幕上继续向右移。因此可以看到，只需改变对象的位置就可以影响速度。

www.TopSage.com

你可能会看到这个过程对于分子模拟很有用。如果有大量粒子四处碰撞，每个粒子都在吸引和排斥多个其他粒子，同时还要试图保持相互之间的特定距离，它们可以只改变相互的位置，而由Verlet积分负责处理这些位置改变所影响的加速度和速度。

Verlet积分的另一个常见特性（在软件中很常用）是对象间约束的概念。两个对象可以相互配对，并认为相互之间维持着一个特定距离。过于远离或接近时，它们就会进行自行调整，回到维持其距离所需的位置，当然这就会改变其速度。对象可能有多个关系，所以一个对象可能与不止一个其他对象关联。Verlet对处理所有这些关系以及维持所有这些距离需求有着很高的效率。因此，要构建复杂的移动结构（如玩偶），这会是一个绝佳的工具。

尽管我一直称它们为对象（object），但是移动的物体通常并没有形状，它们实际上只是点。而且我喜欢称两点之间的固定距离为**线段**（stick）。一条线段由两点组成，结构由一条或多条线段组成。另外，前面已经提到过，两条或多条线段可能有一个共同的点，这样就可以建立关节部分。图6-4显示了它们的关系。

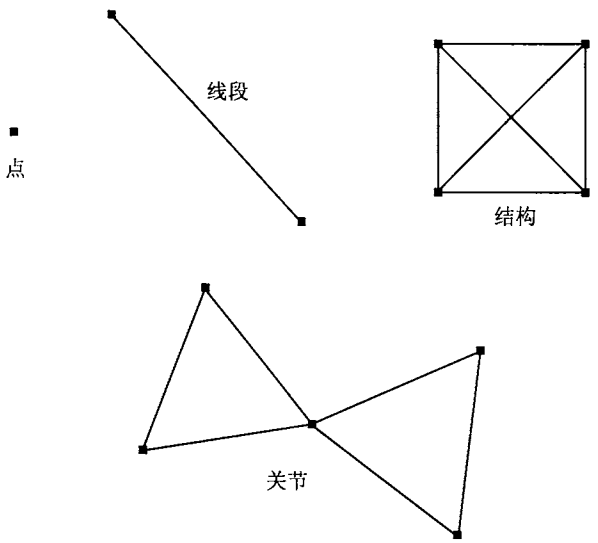


图6-4 Verlet积分中的点、线段、结构和关节

下面首先要有一个移动的点。

6.3.1 Verlet点

我们将创建一个VerletPoint类，来封装Verlet积分所处理的单个点的行为。当然，这个点需要有x和y属性，对应原x和原y的属性，以及一个update方法。这个update方法告诉这个点取其最后一次更新之后的位置与当前位置之差，并使用这个值作为其当前速度。然后将这个速度增加到当前位置。不过在做这个工作之前，必须将当前位置存储为“原”位置，这样下一次更新时它才能知道原来的位置。基本逻辑如下：

www.TopSage.com

大家网 TopSage.com

```
temp = currentPosition
velocity = currentPosition - oldPosition
currentPosition += velocity
oldPosition = temp
```

因为当前位置在赋至原位置之前会改变，所以需要将当前位置存储在一个临时变量中。另外不能过早地将它赋至原位置，这是因为在计算中还需要使用“当前的原位置”。

然后可以计算速度，即当前位置减去原位置，再将这个速度增加到位置。最后指定之前保存的“当前位置”作为原位置。

以下是VerletPoint类：

```
package
{
    import flash.display.Graphics;
    import flash.geom.Rectangle;

    public class VerletPoint
    {
        public var x:Number;
        public var y:Number;

        private var _oldX:Number;
        private var _oldY:Number;


        public function VerletPoint(x:Number, y:Number)
        {
            setPosition(x, y);
        }

        // basic verlet formula
        public function update():void
        {
            var tempX:Number = x;
            var tempY:Number = y;
            x += vx;
            y += vy;
            _oldX = tempX;
            _oldY = tempY;
        }

        // set position without changing velocity
        public function setPosition(x:Number, y:Number):void
        {
            this.x = _oldX = x;
            this.y = _oldY = y;
        }

        // make sure point does not go outside rect
        public function constrain(rect:Rectangle):void
        {
            x = Math.max(rect.left, Math.min(rect.right, x));
            y = Math.max(rect.top, Math.min(rect.bottom, y));
```

www.TopSage.com

```
        }

        // velocity is current position - old position
        public function set vx(value:Number):void
        {
            _oldX = x - value;
        }
        public function get vx():Number
        {
            return x - _oldX;
        }

        // velocity is current position - old position
        public function set vy(value:Number):void
        {
            _oldY = y - value;
        }
        public function get vy():Number
        {
            return y - _oldY;
        }

        public function render(g:Graphics):void
        {
            g.beginFill(0);
            g.drawCircle(x, y, 4);
            g.endFill();
        }

    }
}
```

相当简单。你可能奇怪在这里为什么x速度和y速度（vx和vy）的获取方法和设置方法，因为我前面曾说过Verlet积分不会显式地存储速度。这些获取和设置方法并不存储任何内容。例如，设置*vx*时，它会从当前x减去这个量，并将其赋至_oldX。这样可以确保更新并从x减去_oldX时，会得到同样的x速度。至于获取方法，它只是从当前位置减去原位置。所以这里确实没有显式存储速度。

这里还增加了一个setPosition方法，它将原值和新值都设置为指定的位置。如果希望将一个点移动到某个指定的位置，但是不希望这个移动改变点的速度，这个方法就很有用。因为原位置和新位置相同，所以速度将是0。

稍后将介绍constrain方法，不过这里首先要提到render方法。因为VerletPoint不是一个显示对象，所以在场景中无法直接看到。render方法接受Graphics的一个实例，并根据其位置画一个小圆点。在最终的动画或游戏中你可能不想这么做，不过这个方法对于调试和测试很有用。

下面来看具体的使用：

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

www.TopSage.com

```
import flash.events.Event;

public class VerletPointTest extends Sprite
{
    private var _point:VerletPoint;

    public function VerletPointTest()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        _point = new VerletPoint(100, 100);
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        _point.update();
        graphics.clear();
        _point.render(graphics);
    }
}
}
```

在构造函数中，我们创建了一个VerletPoint，并监听enterFrame事件。在这个事件处理程序中对该点调用了update，然后清除图形并调用render。现在如果运行这个代码，就会看到圆点，不过它只保留在原处不动。下面需要增加一些速度。为此可以设置vx或vy，如下所示，这个代码可以放在构造函数中创建点之后的任何位置：

```
_point.vx = 5;
```

这会移回到原x，使更新方法发现一个5像素的改变，并以此为速度。不过也可以只移动点：

```
_point.x += 5;
```

在大多数系统中，这个过程只会改变点的位置，而根本不影响其速度。不过在Verlet积分中，这还会设置点在该方向上移动。这样来考虑可能会有所帮助：并不是选取点并把它放在5像素外的位置，而更像是你把它推出了5个像素，给它一点推力，之后它会继续移动。

应用重力时也有同样的选择。可以通过增加到y速度来做到，也可以只是增加到y位置。把以下代码行增加到onEnterFrame方法（要求放在更新方法前面），用来负责处理重力方面：

```
_point.vy += .5;
```

或

```
_point.y += .5;
```

第二行看起来只是线性地改变位置，不过要记住，改变位置也会改变速度，所以它会不断地增加速度（就像真实世界中的重力一样）。尽管通过vx和vy存取方法改变速度的做法更清楚，但改变位置的方法则更高效，因为这样只会改变一个公共变量。如果有大量粒子需要高效地交互，这就会很有帮助。

www.TopSage.com

TopSage.com

约束点

你可能认为最好这些点都保持在场景上。将点约束在一个区域中是Verlet积分的另一个重要部分，而且相当简单。我们所要做的只是确保一个点在场景矩形范围内（或者如果你愿意定义一个矩形，这也可以是你定义的其他矩形区域）。为此，要确保点的x不小于矩形的左边界，而且不大于其右边界，另外不高于或低于矩形的上边界和下边界。这里就要用到constrain方法。在主类中，将创建一个描述场景区域的矩形（再次说明，也可以是你希望的任何矩形区域）。每一帧调用update方法之前将这个矩形传入constrain方法。这只是要确保该点的x值不小于或大于矩形的边界值。为此要编写一些代码，以下是该代码的一种简单形式，不过如果你了解有关逻辑，可以看到这确实可行。以下再次给出VerletPoint的constrain方法：

```
public function constrain(rect:Rectangle):void
{
    x = Math.max(rect.left, Math.min(rect.right, x));
    y = Math.max(rect.top, Math.min(rect.bottom, y));
}
```

下面是使用这个方法的测试类：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.Rectangle;

    public class VerletPointTest extends Sprite
    {
        private var _point:VerletPoint;
        private var _stageRect:Rectangle;

        public function VerletPointTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

            _point = new VerletPoint(100, 100);
            _point.x += 5;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _point.y += .5;
            _point.update();
            _point.constrain(_stageRect);

            graphics.clear();
            _point.render(graphics);
        }
```


www.TopSage.com

```
        }
    }
```

我知道现在你可能会说：它不会反弹。没错，它确实不能反弹。可以修改constrain来增加一种反弹效果。这会稍微复杂一些，不过增加反弹之后，当点碰到墙时看上去会好得多。但要记住，一般来讲，点实际上只用于定义线段和结构，它们本身通常没有真正的可视化表示。所以下面来关注这个问题，看看碰到墙时线段和结构如何反应，而不是更多地增加VerletPoint类的复杂性（除非认为确实有必要）。

既然提到线段，下面就来看看线段！

6.3.2 Verlet线段

再次说明，线段将两个点绑定在一起。它有一个length属性，这是试图在两个点之间保持的距离。如果两个点之间的距离不是这个值，则会将它们移近或远离以满足length属性。

以下是VerletStick类：

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

    public class VerletStick
    {
        private var _pointA:VerletPoint;
        private var _pointB:VerletPoint;
        private var _length:Number;

        public function VerletStick(pointA:VerletPoint, pointB:VerletPoint, length:Number = -1)
        {
            _pointA = pointA;
            _pointB = pointB;
            if(length == -1)
            {
                var dx:Number = _pointA.x - _pointB.x;
                var dy:Number = _pointA.y - _pointB.y;
                _length = Math.sqrt(dx * dx + dy * dy);
            }
            else
            {
                _length = length;
            }
        }

        public function update():void
        {
            var dx:Number = _pointB.x - _pointA.x;
            var dy:Number = _pointB.y - _pointA.y;
            var dist:Number = Math.sqrt(dx * dx + dy * dy);
            var diff:Number = _length - dist;
            var offsetX:Number = (diff * dx / dist) / 2;
            var offsetY:Number = (diff * dy / dist) / 2;
            _pointA.x -= offsetX;
```

www.TopSage.com

```
            _pointA.y -= offsetY;
            _pointB.x += offsetX;
            _pointB.y += offsetY;
        }

        public function render(g:Graphics):void
        {
            g.lineStyle(0);
            g.moveTo(_pointA.x, _pointA.y);
            g.lineTo(_pointB.x, _pointB.y);
        }
    }
}
```

构造函数接受两个VerletPoint实例和一个可选的length。如果没有给定长度，它会计算所传入两点之间的初始长度，并以此作为其长度。类似于VerletPoint，这里也有一个render方法，可以在Graphics实例上绘制两点之间的线段，这主要用于调试和测试。这个类的关键就在update方法中，需要对这个方法做一些解释。

首先，得到两点之间的距离，从线段的length属性减去这个距离。这会告诉我们目前线段的长度比其理想长度长还是短。这个结果会存储在diff变量中（参见图6-5）。

然后完成一些复杂的三角运算得到这个差的x和y分量。要知道，由于余弦是邻边/斜边，而正弦是对边/斜边，所以计算如下：

```
diff * dx / dist
diff * dy / dist
```

实际上等同于下面的计算，这里angle是两点间的角度：

```
diff * cos(angle)
diff * sin(angle)
```

不过这样一来可以避免3个三角操作：计算角度的Math.atan2，以及得到x和y偏移的Math.cos和Math.sin，如图6-6所示。

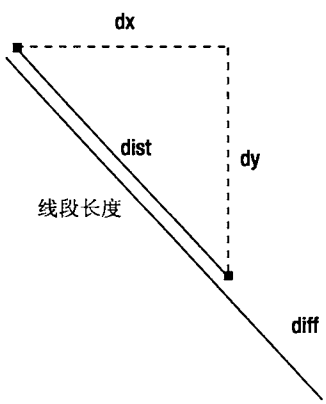


图6-5 计算两点间距离以及它与其理想长度之差

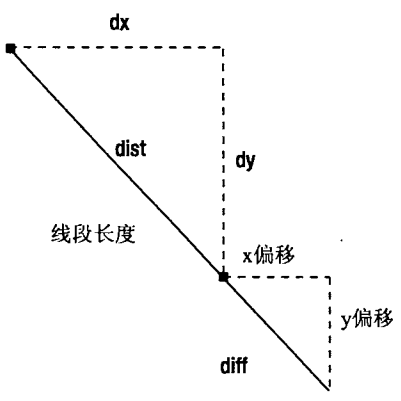


图6-6 计算差的*x*和*y*分量

www.TopSage.com

注意，这里将其结果除以2，因为我们将把两个点都朝其理想位置移动一半的距离。最后，取这个offsetX和offsetY，并从第一个点减去这两个偏移量，再把这两个偏移量加至第二个点。这就会使它们相互之间有适当的距离以满足length属性，如图6-7所示。

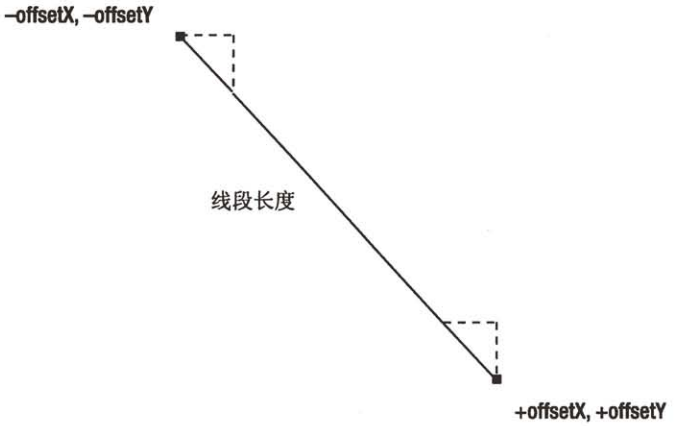


图6-7　将各个点沿各轴移动差的一半，使最终距离等于线段长度

计算这个距离还有其他一些方法，其中有些方法可能更为高效。不过，这里只需要一个Math类操作，所以也很不错。另外我认为这样能很清楚地了解它的工作（当然是以我的解释和提供的图表为前提），对于这本书来讲这种方法已经很好了。

下面来建立一个线段，看看它的具体工作！

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.Rectangle;

    public class VerletStickTest extends Sprite
    {
        private var _pointA:VerletPoint;
        private var _pointB:VerletPoint;
        private var _stick:VerletStick;
        private var _stageRect:Rectangle;

        public function VerletStickTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

            _pointA = new VerletPoint(100, 100);
            _pointB = new VerletPoint(105, 200);
            _stick = new VerletStick(_pointA, _pointB);
```

www.TopSage.com

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

        private function onEnterFrame(event:Event):void
        {
            _pointA.y += .5;
            _pointA.update();
            _pointA.constrain(_stageRect);

            _pointB.y += .5;
            _pointB.update();
            _pointB.constrain(_stageRect);

            _stick.update();

            graphics.clear();
            _pointA.render(graphics);
            _pointB.render(graphics);
            _stick.render(graphics);
        }
    }
}
```

可以看到，要增加一个线段无需太多工作。首先建立点，然后建立线段，为之传入之前创建的两个点。然后在enterFrame处理程序中更新点，之后再更新线段。当然还要完成渲染显示。

运行这个代码时，你将看到线段会出现，然后落向场景底边。就像有魔法一样，它确实会有一点反弹！怎么会这样呢？要知道我们并没有增加任何反弹代码！这是因为线段的更新代码会尝试将两点拉开，不过当下面的点碰到矩形边界时，约束代码就会把它向上拉。这两个动作相互抗争，其结果是两个点都向上移一点，以试图满足所有这些条件。

实际上，这种反弹是一种副作用。我很喜欢这一点，不过因为我们并没有明确地编写相应的代码，所以这也可以看作是一个问题。要消除这种反弹，需要多运行几次点的constrain调用和线段的update调用，先消除这种现象然后再继续。可以将onEnterFrame方法修改如下：

```
private function onEnterFrame(event:Event):void
{
    _pointA.y += .5;
    _pointA.update();

    _pointB.y += .5;
    _pointB.update();
    for(var i:int = 0; i < 5; i++)
    {
        _pointA.constrain(_stageRect);
        _pointB.constrain(_stageRect);
        _stick.update();
    }

    graphics.clear();
    _pointA.render(graphics);
    _pointB.render(graphics);
```

www.TopSage.com

```
        _stick.render(graphics);
    }
```

通过循环多次调用点的约束方法和线段的更新方法，这样就能够对点的最终位置达成一个“共识”。由于这些迭代期间没有增加重力也没有更新点，所以唯一增加的速度就是一个差值，即各点原来所在的位置与循环结束时其最终位置之差，所以这里不再有跳跃或反弹。这更像是一个铁棒而不是一个有弹性的嫩枝条。现在可以调整这个循环来改变模拟的角色——可以只做一次迭代来模拟很有弹性的角色，也可以增加更多迭代来模拟硬性很大无弹性的东西。

不过，要知道完成这些迭代会执行大量代码，特别是在线段的更新方法中。这里对于一个线段来说不算什么，不过如果有更大的结构，更多迭代就会开始大量耗费你的计算机资源。我的建议是保留一个迭代，如果你觉得确实有必要再增加迭代次数。

6.3.3 Verlet结构

Verlet结构不只是有一些共同点的线段。最简单的牢固结构是三角形。只需建立3个点，再增加3个线段将它们连接起来：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.Rectangle;

    public class Triangle extends Sprite
    {
        private var _pointA:VerletPoint;
        private var _pointB:VerletPoint;
        private var _pointC:VerletPoint;
        private var _stickA:VerletStick;
        private var _stickB:VerletStick;
        private var _stickC:VerletStick;
        private var _stageRect:Rectangle;
        public function Triangle()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

            _pointA = new VerletPoint(100, 100);
            _pointB = new VerletPoint(200, 100);
            _pointC = new VerletPoint(150, 200);

            _stickA = new VerletStick(_pointA, _pointB);
            _stickB = new VerletStick(_pointB, _pointC);
            _stickC = new VerletStick(_pointC, _pointA);

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

www.TopSage.com

TopSage.com

```
        private function onEnterFrame(event:Event):void
        {
            _pointA.y += .5;
            _pointA.update();

            _pointB.y += .5;
            _pointB.update();

            _pointC.y += .5;
            _pointC.update();

            for(var i:int = 0; i < 1; i++)
            {
                _pointA.constrain(_stageRect);
                _pointB.constrain(_stageRect);
                _pointC.constrain(_stageRect);
                _stickA.update();
                _stickB.update();
                _stickC.update();
            }

            graphics.clear();
            _pointA.render(graphics);
            _pointB.render(graphics);
            _pointC.render(graphics);
            _stickA.render(graphics);
            _stickB.render(graphics);
            _stickC.render(graphics);
        }
    }
}
```

这里建立了3个点，A, B和C；还建立了三个线段，也命名为A, B和C。每个线段使用两个不同的点，从而构成一个三角形。运行这个代码，你会看到这个三角形会落在场景底部，向上反弹一点，翻转过来，停靠在它的边上。试着将约束/更新迭代变量改为一个大于1的值，可以看到这会让结构弹性降低，而硬度更大。

下面来看一个正方形。首先你可能会得出以下代码：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.Rectangle;

    public class Square extends Sprite
    {
        private var _pointA:VerletPoint;
        private var _pointB:VerletPoint;
        private var _pointC:VerletPoint;
        private var _pointD:VerletPoint;
        private var _stickA:VerletStick;
        private var _stickB:VerletStick;
```

www.TopSage.com

```
private var _stickC:VerletStick;
private var _stickD:VerletStick;
private var _stageRect:Rectangle;

public function Square()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    _stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

    _pointA = new VerletPoint(100, 100);
    _pointB = new VerletPoint(200, 100);
    _pointC = new VerletPoint(200, 200);
    _pointD = new VerletPoint(100, 200);

    _stickA = new VerletStick(_pointA, _pointB);
    _stickB = new VerletStick(_pointB, _pointC);
    _stickC = new VerletStick(_pointC, _pointD);
    _stickD = new VerletStick(_pointD, _pointA);

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
private function onEnterFrame(event:Event):void
{
    _pointA.y += .5;
    _pointA.update();

    _pointB.y += .5;
    _pointB.update();

    _pointC.y += .5;
    _pointC.update();

    _pointD.y += .5;
    _pointD.update();

    for(var i:int = 0; i < 1; i++)
    {
        _pointA.constrain(_stageRect);
        _pointB.constrain(_stageRect);
        _pointC.constrain(_stageRect);
        _pointD.constrain(_stageRect);
        _stickA.update();
        _stickB.update();
        _stickC.update();
        _stickD.update();
    }

    graphics.clear();
    _pointA.render(graphics);
    _pointB.render(graphics);
    _pointC.render(graphics);
    _pointD.render(graphics);
    _stickA.render(graphics);
```

www.TopSage.com

```
            _stickB.render(graphics);
            _stickC.render(graphics);
            _stickD.render(graphics);
        }
    }
}
```

建立4个点，再用4条线把它们连接起来，对不对？最初看上去一切正常，但是等它降落后，实际上会“摔”得有点扁（这样讲确实恰如其分）。对于这种情况，这确实是个问题，不过这是你第一次接触到关节（hinge），这是一个很酷的特性。在讨论这个特性之前，下面先来加固这个方框，使它不至于“摔变形”。只需再加一个线段就可以做到。声明一个_stickE变量，设置为从点A到点C的对角线：

```
_stickE = new VerletStick(_pointA, _pointC);
```

确保还要在enterFrame处理程序中更新并显示这个线段。如果你希望它真正牢固，可以再增加一个从点B到点D的对角线段。

如果觉得这个直接落下的示例没有多大意思，也可以让它有些旋转。只需在创建某个点之后稍稍给个推力：

```
_pointA = new VerletPoint(100, 100);
_pointA.vx = 10;
```

因为只有一个拐角有初始速度，所以整个结构会开始旋转。

进一步深入之前，现在应当考虑整理这个代码。尽管这里只有4个点和5个线段，看起来代码也相当繁琐。好在创建线段和点之后我们会对每一部分做相同的工作：为点增加重力并完成更新，约束各个点并且更新各个线段（可能多次），然后显示所有点和线段。

创建一个点数组和一个线段数组很容易，可以通过一个循环完成所有这些动作：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.Rectangle;

    public class Square extends Sprite
    {
        private var _points:Array;
        private var _sticks:Array;
        private var _stageRect:Rectangle;

        public function Square()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

            _points = new Array();
```

www.TopSage.com

大家网 TopSage.com

```
        _sticks = new Array();

        var pointA:VerletPoint = makePoint(100, 100);
        pointA.vx = 10;
        var pointB:VerletPoint = makePoint(200, 100);
        var pointC:VerletPoint = makePoint(200, 200);
        var pointD:VerletPoint = makePoint(100, 200);
        makeStick(pointA, pointB);
        makeStick(pointB, pointC);
        makeStick(pointC, pointD);
        makeStick(pointD, pointA);
        makeStick(pointA, pointC);

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        updatePoints();

        for(var i:int = 0; i < 1; i++)
        {
            constrainPoints();
            updateSticks();
        }

        graphics.clear();
        renderPoints();
        renderSticks();
    }

    private function makePoint(xpos:Number, ypos:Number):VerletPoint
    {
        var point:VerletPoint = new VerletPoint(xpos, ypos);
        _points.push(point);
        return point;
    }

    private function makeStick(pointA:VerletPoint, pointB:VerletPoint,
                               length:Number = -1):VerletStick
    {
        var stick:VerletStick = new VerletStick(pointA, pointB, length);
        _sticks.push(stick);
        return stick;
    }

    private function updatePoints():void
    {
        for(var i:int = 0; i < _points.length; i++)
        {
            var point:VerletPoint = _points[i] as VerletPoint;
            point.y += .5;
            point.update();
        }
```

www.TopSage.com

```
        }

        private function constrainPoints():void
        {
            for(var i:int = 0; i < _points.length; i++)
            {
                var point:VerletPoint = _points[i] as VerletPoint;
                point.constrain(_stageRect);
            }
        }

        private function updateSticks():void
        {
            for(var i:int = 0; i < _sticks.length; i++)
            {
                var stick:VerletStick = _sticks[i] as VerletStick;
                stick.update();
            }
        }

        private function renderPoints():void
        {
            for(var i:int = 0; i < _points.length; i++)
            {
                var point:VerletPoint = _points[i] as VerletPoint;
                point.render(graphics);
            }
        }

        private function renderSticks():void
        {
            for(var i:int = 0; i < _sticks.length; i++)
            {
                var stick:VerletStick = _sticks[i] as VerletStick;
                stick.render(graphics);
            }
        }
    }
}
```

现在有了很多较小的函数，不过不再为各个点和线段使用单独的变量。可以看到现在的enterFrame处理程序变得多么简洁。不仅如此，现在可以根据需要增加更多新的点和线段，而不必修改任何其他代码。它们会自动增加到数组并得到更新、约束和显示。

6.3.4 关节

既然已经有了一个不错的代码结构，下面可以大胆尝试不同样式。现在可以来讨论**关节**（hinge）了，这是指两个结构有一个共同的点。这两个结构可以分别有各自的运动，不过会以该共同点作为支点。

下面介绍如何建立一个晃动的臂摆。这里只给出构造函数，其中创建了点和线段。所有其他方法都与上例中完全相同：

www.TopSage.com

```
public function Hinge()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    _stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

    _points = new Array();
    _sticks = new Array();

    // base
    var pointA:VerletPoint = makePoint(stage.stageWidth / 2, stage.stageHeight - 500);
    var pointB:VerletPoint = makePoint(0, stage.stageHeight);
    var pointC:VerletPoint = makePoint(stage.stageWidth, stage.stageHeight);

    // arm
    var pointD:VerletPoint = makePoint(stage.stageWidth / 2 + 350, stage.stageHeight - 500);

    // weight
    var pointE:VerletPoint = makePoint(stage.stageWidth / 2 + 360, stage.stageHeight - 510);
    var pointF:VerletPoint = makePoint(stage.stageWidth / 2 + 360, stage.stageHeight - 490);
    var pointG:VerletPoint = makePoint(stage.stageWidth / 2 + 370, stage.stageHeight - 500);

    // base
    makeStick(pointA, pointB);
    makeStick(pointB, pointC);
    makeStick(pointC, pointA);

    // arm
    makeStick(pointA, pointD);

    // weight
    makeStick(pointD, pointE);
    makeStick(pointD, pointF);
    makeStick(pointE, pointF);
    makeStick(pointE, pointG);
    makeStick(pointF, pointG);

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

这个代码创建一个三角形基座，有一个臂摆关联到这个三角形的上顶点，还有一个秤砣关联到臂摆末端。由于臂摆和基座只有一个共同的点，所以这个臂摆可以摆动。秤砣也会以臂摆末端为支点摆动，但由于太小所以可能注意不到。

6.3.5　进一步深入

还有很多工作可以做，如将所有更新、约束和显示方法都分别放在各自的类中——可能是PointManager和StickManager或者其他某个类。为它建立一个可视化编辑器听起来也很有意思。实际上我确实构建了几个可视化编辑器来完成这样一些工作，而且确实非常有趣！

你可能想为各个线段创建一些更有趣的图片。如果适当放置一个图片使其左边缘以线段的第一个点为中心，对它进行缩放使它的大小与线段的长度相同，然后旋转使之与线段的方向一致，

www.TopSage.com

TopSage.com

现在各个线段就有了一个定制图片而不是一条黑线。完成整个结构的定制会更困难一些，不过也是可以做到的。

已经没有更多篇幅来讨论这个问题了，而且我也不打算面面俱到，剥夺你尝试的乐趣，所以就此告一段落，由你自己尽情发挥。我认为，如果有更多的人理解Flash中的Verlet积分，会有更多出色的作品在Web上出现。

6.4 小结

本章介绍了数值积分——包括什么是数值积分以及积分方法存在的问题。现在你知道了也许自己一直都在使用欧拉积分（而且也知道了怎么读这个词），另外你还了解到一些候选方法可供使用。最重要的是，你可能对各个方法分别适用于什么场合已经有所认识，因为这些方法都各有优缺点。

数值积分并不只是包括欧拉积分、龙格–库塔积分和Verlet积分。在维基百科上快速搜索就会得到这样一个名单，会列出很多听起来很有意思的名称：Backward Euler、Semi-implicit Euler、Velocity Verlet、Beeman算法, Heun方法、Newmark-beta方法、Leapfrog积分和蒙特卡洛积分。我不清楚这些都是什么，不过了解这些方法可能很有趣。享受研究的乐趣吧！

接下来，我们来讨论与3D有关的一些全新的Flash 10特性！

www.TopSage.com

第7章

Flash 10中的3D

7

从我涉足Flash以来，大约每18个月就会看到一款新版本的Flash创作工具问世，并提供一个新的Flash播放器。在各个版本发布后不久，又会出现对下一个版本的传闻和推测，并对它的功能提出种种愿望表。人们期盼以久的特性之一就是内置3D，但是这个特性却始终没有露面，这让人们的希望屡屡破灭。直到Flash 10的出现，情况才有了转机。

Flash中总是可以通过缩放和定位使物体看上去更接近或更远离来实现3D效果。ActionScript提供了编写真正3D引擎的功能。我已经在《Flash ActionScript 3.0 动画教程》中介绍过有关的基础知识，在过去几年中，已经出现了很多功能强大的Flash 3D引擎。PaperVision3D、Away3D和其他3D引擎都能从专业建模工具里加载完备的3D模型和纹理，并实时渲染和处理它们。不过所有这些处理都在ActionScript中通过类完成，而这些类是由社区成员所编写的。这些类被编译后在Player中运行而不是作为Player本身的一个固有部分。

在Flash 10中，我们终于能够创建一个显示对象（sprite、动画片段、文本域等）并在3D空间中直接操纵它。可以直接利用Flash创作工具本身完成，也可以通过ActionScript实现。本章将重点介绍ActionScript。

尽管这个特性绝对无法与PaperVision3D的功能相提并论——如加载模型和纹理、自动深度排序等——但是它使用起来却极为容易，对于你希望在Flash中实现的许多基本3D效果来说这已经足够了。那么还等什么呢？着手实现3D吧。

7.1 Flash 10 3D 基础

这是一本进阶程度的书，所以我认为你至少对3D已经有了一些了解。应该知道存在3个维度：*x*是水平方向，*y*是垂直方向，*z*是进出方向（深度）。在Flash中，原点或零点是屏幕的左上角，至少在2D空间中就是如此。如果你习惯于使用标准笛卡尔坐标，在这里*y*轴看上去可能上下颠倒了，但是你必须熟悉这种表示。在Flash 10 3D中，*z*轴的值会随着对象“移入屏幕”或远离观察者而变大。换句话说，*z*位置较小的对象会在*z*位置较大的对象之前出现，如图7-1所示。

还需要指出如何在Flash 10 3D中的各个轴上旋转。如果从前面观察，绕*z*轴的旋转会随角度增加顺时针旋转。同样的，这与你以往习惯的正相反。*y*轴上的旋转也是顺时针旋转（从上面观察）。从对象左边观察时，*x*轴上的旋转是顺时针旋转（参见图7-2）。

www.TopSage.com

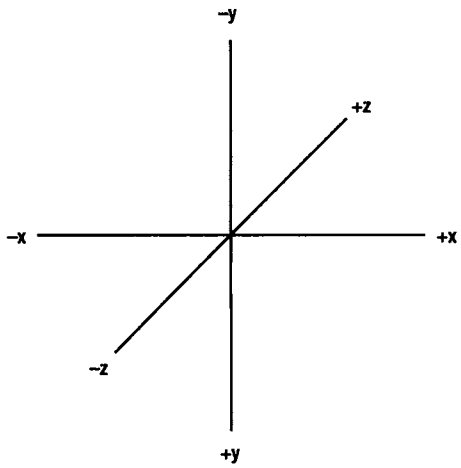
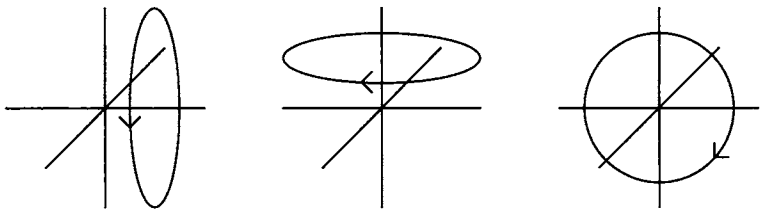


图7-1 Flash 10 3D 坐标



正*x*旋转 正*y*旋转 正*z*旋转

图7-2 Flash 10 3D 旋转

还需要认识到一个重要的问题，对于Flash 10中的3D旋转，角度是度数而不是弧度。如果你熟悉3D编程，其中会使用大量用弧度运算的三角函数，这看起来就可能有些奇怪。不过，Flash中的3D功能是创作工具的一部分，它面向的是设计人员而不是技术人员。设计人员希望把一个东西放在45° 角上，而不是Math.PI/4弧度角。如果需要回顾如何在弧度和度之间转换的公式，下面给你一个提示：

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

既然已经定义了术语，下面来看Flash 10中有关3D应用编程接口（Application Programming Interface，API）的。尽管其内容远不只是这里所列，但全新3D API的核心内容可以归结为4个新属性：z、rotationX、rotationY和rotationZ。其他内容我们也会介绍，不过这4个属性将是你创建基本3D效果时最常用的。

下面就来试一试。创建下面这个类并完成编译。与本章以及本书其他章节的所有文件一样，你都可以在本书下载页面www.friendsofed.com中找到这个类。文件名为Test3D.as：

www.TopSage.com

大家网 TopSage.com

```
package {
    import flash.display.Shape;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.PerspectiveProjection;
    import flash.geom.Point;

    [SWF(backgroundColor=0xffffff)]
    public class Test3D extends Sprite
    {
        private var _shape:Shape;

        public function Test3D()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _shape = new Shape();
            _shape.graphics.beginFill(0xff0000);
            _shape.graphics.drawRect(-100, -100, 200, 200);
            _shape.x = stage.stageWidth / 2;
            _shape.y = stage.stageHeight / 2;
            addChild(_shape);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _shape.rotationY += 2;
        }
    }
}
```

太棒了！Flash自带的3D！

设置没影点

我知道你想直接深入，着手制作很酷的3D效果，不过请先阅读一下本节的内容——这里有一些非常重要的内容。尽管你已经对3D有所了解，但要知道Flash中存在一些陷阱，除非你明白其中的原理，否则这些陷阱很可能会让你抓狂。一旦充分理解了这些原理，你会发现这确实有道理，但是如若不然，你可能会以为Flash出了问题。

如果使用上面例子的类作为FLA的文档类，并通过Flash CS4进行编译，可能会看到如图7-3所示的结果。

不过，如果通过Flex Builder或另外一个使用Flex 4 SDK的设置面向Flash 10进行编译，往往会看到图7-4所示的结果。

你会看到它向左上角延伸。存在这种区别的原因并不在于动画的编译方式，而是在于它们采用的发布方式不同，以及Flash会以何种方式设置没影点。在3D中，没影点（vanishing point）是

指所有对象移至远方时所汇聚的点。为说明这个概念，我本打算在这里插入一张铁轨的图片，显示出轨道向地平线延伸的效果，不过我相信你应该很清楚我的意思。从头开始编写3D代码时，必须手动地选择一个点作为没影点，并确保所有对象都汇聚于此。人们通常选择场景的中心作为没影点。

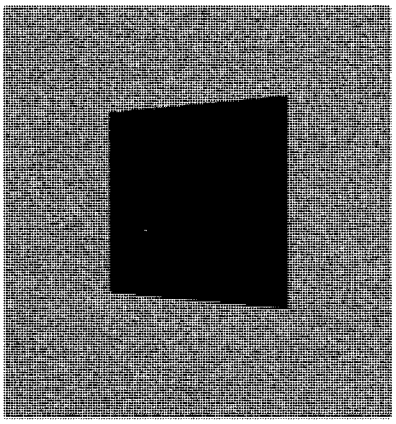


图7-3 用Flash CS4编译的飞机的旋转

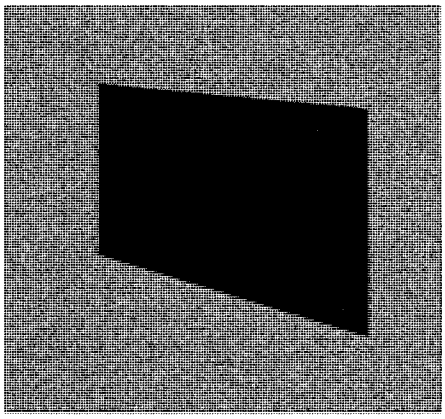


图7-4 用Flex Builder编译的飞机的旋转

Flash 10 3D会自动地为你设置没影点，它把没影点设置为场景的中心。不过，这个工作它只会做一次，也就是在SWF刚加载时Flash 10 3D会自动设置没影点。在Flash中测试动画时，动画会根据文档属性面板中设置的大小弹出在一个窗口中。所以没影点位于其中心，效果也很好。

不过，在Flex Builder中，默认SWF的大小为500×375像素。因此没影点将位于（250, 187.5）。但是一旦将stage.align设置为StageAlign.TOP_LEFT，并将stage.scaleMode设置为StageScaleMode.NO_SCALE，就会增加场景的大小。（这是一种设置大小的常用做法，这样一来，如果显示区增大，SWF也不会扩展其内容——这只会让场景增大。）通过跟踪场景的宽度和高度在这些调用之前以及之后的变化就可以看出这一点：

```
trace(stage.stageWidth, stage.stageHeight); // 500, 375
stage.align = StageAlign.TOP_LEFT;
stage.scaleMode = StageScaleMode.NO_SCALE;
trace(stage.stageWidth, stage.stageHeight); // 1440, 794
```

对于最后一个trace语句，你可能会得到不同的值，这取决于你的浏览器窗口的大小。所以，场景的大小改变了，sprite便会重新设置在场景的新中心上，这个中心可能是（720,397）。不过没影点仍是在（250,187.5）。这就产生了问题。

修正这个问题最容易的方法就是在SWF元数据中显式地设置一个宽度和高度：

```
[SWF(backgroundColor=0xffffff, width=800, height=800)]
```

这将会在计算没影点之前运行，所以会按（800,800）的中心点计算。在设置了align（对齐方式）和scaleMode（缩放模式）之后场景大小不会改变，所以sprite会被放置在同一个中心点上。

不过，也许你希望有一个大小可变的场景，不论浏览器窗口多大，场景都会完全填充浏览器

www.TopSage.com

窗口。要做到这一点会更复杂一些，但并不太难。需要使用一个新的名为PerspectiveProjection的类。这个类会控制有关如何渲染3D透视图的很多方面，包括没影点。Flash 10中的每一个显示对象现在都指定有一个PerspectiveProjection，用来控制如何渲染3D，并赋至transform属性的perspectiveProjection属性。例如，对于一个名为s的sprite，可以这样去访问：

```
s.transform.perspectiveProjection
```

PerspectiveProjection类有一个名为projectionCenter的属性，这是Point类的一个实例。这实际上就是我们一直所称的没影点。因此，要把一个显示对象的没影点设置到场景的中心，可以按照如下方式实现：

```
s.transform.perspectiveProjection.projectionCenter =
    new Point(stage.stageWidth / 2, stage.stageHeight / 2);
```

这里只设置了这一个对象的没影点，当然也可以设置该对象所有子对象的没影点。如果想要为动画中的所有对象设置没影点，可以在根一层设置，如下：

```
root.transform.perspectiveProjection.projectionCenter =
       new Point(stage.stageWidth / 2, stage.stageHeight / 2);
```

下面的类就采用了这种做法，在设置场景的align和scaleMode之后做了上述设置，从而可以解决我们之前看到的问题：

```
package {
    import flash.display.Shape;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.PerspectiveProjection;
    import flash.geom.Point;

    [SWF(backgroundColor=0xffffff)]
    public class Test3D extends Sprite
    {
        private var _shape:Shape;

        public function Test3D()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            root.transform.perspectiveProjection.projectionCenter =
                    new Point(stage.stageWidth / 2, stage.stageHeight / 2);

            _shape = new Shape();
            _shape.graphics.beginFill(0xff0000);
            _shape.graphics.drawRect(-100, -100, 200, 200);
            _shape.x = stage.stageWidth / 2;
            _shape.y = stage.stageHeight / 2;
            addChild(_shape);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
```

www.TopSage.com

TopSage.com

```
        private function onEnterFrame(event:Event):void
        {
            _shape.rotationY += 2;
        }
    }
}
```

可以更进一步，完成以下修改，这样无论场景大小如何调整都可以改变中心点：

```
package {
    import flash.display.Shape;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.PerspectiveProjection;
    import flash.geom.Point;

    public class Test3D extends Sprite
    {
        private var _shape:Shape;
        public function Test3D()
        {
            stage.addEventListener(Event.RESIZE, onResize)
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _shape = new Shape();
            _shape.graphics.beginFill(0xff0000);
            _shape.graphics.drawRect(-100, -100, 200, 200);
            _shape.x = stage.stageWidth / 2;
            _shape.y = stage.stageHeight / 2;
            addChild(_shape);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onResize(event:Event):void
        {
            root.transform.perspectiveProjection.projectionCenter =
                    new Point(stage.stageWidth / 2, stage.stageHeight / 2);
            if(_shape != null)
            {
                _shape.x = stage.stageWidth / 2;
                _shape.y = stage.stageHeight / 2;
            }
        }

        private function onEnterFrame(event:Event):void
        {
            _shape.rotationY += 2;
        }
    }
}
```

由于我们把监听resize事件作为第一个动作，因此一旦改变场景设置和场景大小，就会调用onResize方法，这会立即设置投影中心。另外，当用户通过调整浏览器窗口大小来改变场景大小时也会调用onResize方法。这里还会重新定位这个形状（shape），确保它总是在场景中心。不过会在形状创建之前第一次调用onResize，所以首先必须确保形状存在。所以这里设置了这个条件。

为简单起见，本章的余下章节只讨论第一种解决方案，只是利用元数据设置场景大小。

好了，要让你的动画看上去有很好的透视效果，以上就是需要了解的全部重要内容。下面来看看利用3D还能做些什么吧。

7.2　3D 定位

这个问题很明显。我认为不需要再介绍如何改变对象在*x*轴和*y*轴上的位置。其实在*z*轴上改变位置也很容易。下面的类（见Position3D.as文件）建立了一个正弦曲线，将一个形状（shape）晃动到远处再返回，如此反复。*x*轴和*y*轴上的位置则随鼠标位置而调整：

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

    [SWF(width=800, height=800, backgroundColor=0xffffff)]
    public class Position3D extends Sprite
    {
        private var _shape:Shape;
        private var _n:Number = 0;

        public function Position3D()
        {
            _shape = new Shape();
            _shape.graphics.beginFill(0x00ff00);
            _shape.graphics.drawRect(-100, -100, 200, 200);
            _shape.graphics.endFill();
            addChild(_shape);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _shape.x = mouseX;
            _shape.y = mouseY;
            _shape.z = 10000 + Math.sin(_n += .1) * 10000;
        }
    }
}
```

我认为从这个例子你能学到的最重要的一点就是，一旦改变了*z*坐标，显示对象的*x*和*y*坐标就不再直接取决于屏幕坐标系，而是取决于3D空间坐标系。如果没有移动鼠标，这个形状的*x*和*y*坐标不会改变，但是它在屏幕上的*x*和*y*坐标轴会改变。只有当*z*坐标为0时对象的*x*和*y*坐标才会与

www.TopSage.com

屏幕上的x和y坐标一致。这是因为，如果z小于0，Flash就会放大对象；而当z大于0时，Flash会缩小对象，不过当z正好等于0时，缩放比例刚好为100%。

7.2.1 深度排序

在3D空间中创建并定位多个对象时，首先遇到的问题就是较远的对象（有更大的z坐标值）有时会出现在较近对象的前面。我猜想你肯定希望知道能运用哪些属性或方法。遗憾的是，没有可以设置或调用的属性或方法来确保对象总能正确地排序。

Flash 10中的3D API可以处理单个显示对象的透视缩放和变形（甚至如果该对象有子对象，还可以处理其子对象），不过它不会影响对象在屏幕上绘制的顺序。这仍采用Flash 9中处理2D对象的方式进行处理：通过使用addChild方法，放在显示列表中的任何对象会出现在之前增加到同一容器的对象前面。要改变这种情况，唯一的途径就是使用各种管理显示列表的方法，如addChild、addChildAt、swapChildren、removeChild，等等。由于显示对象容器没有排序方法，因此所有深度排序工作都必须手动完成。

要了解如何解决这个问题，下面首先建立一个例子来说明这个问题。来看看一个树木丛生的森林怎么样？

```
package
{
    import flash.display.Shape;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(width=800, height=800, backgroundColor = 0xccffcc)]
    public class DepthSort extends Sprite
    {
        public function DepthSort()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            for(var i:int = 0; i < 500; i++)
            {
                var tree:Shape = new Shape();
                tree.graphics.beginFill(Math.random() * 255 << 8);
                tree.graphics.lineTo(-10, 0);
                tree.graphics.lineTo(-10, -30);
                tree.graphics.lineTo(-40, -30);
                tree.graphics.lineTo(0, -100);
                tree.graphics.lineTo(40, -30);
                tree.graphics.lineTo(10, -30);
                tree.graphics.lineTo(10, 0);
                tree.graphics.lineTo(0, 0);
                tree.graphics.endFill();
                tree.x = Math.random() * stage.stageWidth;
                tree.y = stage.stageHeight - 100;
                tree.z = Math.random() * 10000;
```

www.TopSage.com

大家网 TopSage.com

```
                addChild(tree);
            }
        }

    }
}
```

这里创建了大量形状，并使用绘图API在各个形状中分别画一个有随机阴影的绿树。每个形状随机地赋予x, y和z值。尽管这不是Flash设计的最佳境界，不过也能达到我们的目的。看上去不太对劲，如图7-5所示。

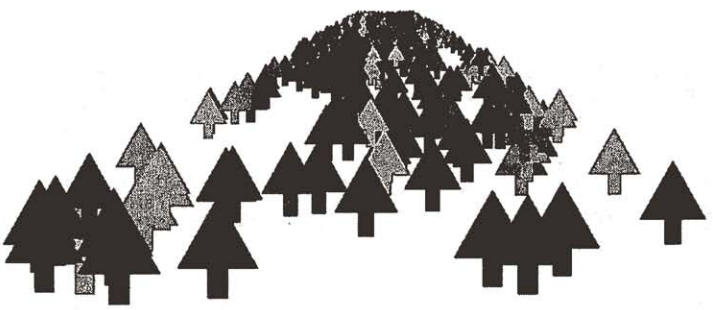


图7-5　透视很好，但深度排序有问题

再次申明，我们无法对显示列表排序，不过可以对数组排序。所以在创建时并不是将各个树增加到显示列表，而是放入一个数组。然后对这个数组排序，并以正确的顺序将树增加到显示列表——保证先放入z值大的树，后放入z值小的树。

```
package
{
    import flash.display.Shape;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(width=800, height=800, backgroundColor = 0xccffcc)]
    public class DepthSort extends Sprite
    {
        private var _trees:Array;

        public function DepthSort()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
```

www.TopSage.com

```
            _trees = new Array();

            for(var i:int = 0; i < 500; i++)
            {
                var tree:Shape = new Shape();
                tree.graphics.beginFill(Math.random() * 255 << 8);
                tree.graphics.lineTo(-10, 0);
                tree.graphics.lineTo(-10, -30);
                tree.graphics.lineTo(-40, -30);
                tree.graphics.lineTo(0, -100);
                tree.graphics.lineTo(40, -30);
                tree.graphics.lineTo(10, -30);
                tree.graphics.lineTo(10, 0);
                tree.graphics.lineTo(0, 0);
                tree.graphics.endFill();
                tree.x = Math.random() * stage.stageWidth;
                tree.y = stage.stageHeight - 100;
                tree.z = Math.random() * 10000;
                _trees.push(tree);
            }

            _trees.sortOn("z", Array.NUMERIC | Array.DESCENDING);
            for(i = 0; i < 500; i++)
            {
                addChild(_trees[i] as Shape);
            }
        }

    }
}
```

尽管这并不能让图7-6中所示的图片更具艺术感，但是至少较远的树会出现在较近树的后面，并且与你在真实森林中所看到的情况一致。

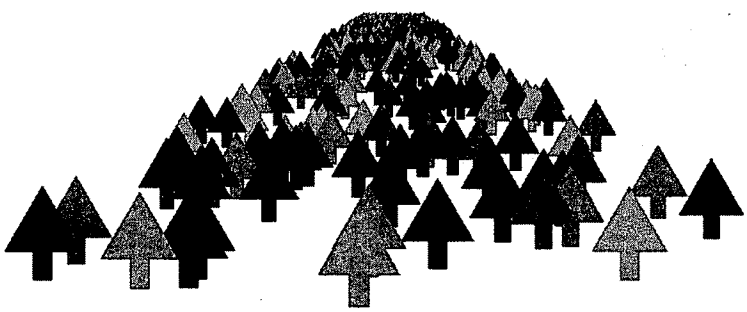


图7-6 实现了深度排序后的森林

7.2.2 3D容器

我刚开始使用这个API时，在Flash 10中让我对3D最满意的事情是，我发现显示对象容器在

变换时，其子对象也会变换。换句话说，如果将一些显示对象增加到一个sprite，然后移动这个容器sprite，它并不会简单地扁平化该sprite的视图并把它作为单个对象在3D中移动。实际上它会转换各个子对象，这样它们看上去都在3D空间中独立移动。

同样地，用例子来说明比直接描述更容易。下面的类就将说明这一点。这一次我们不再使用其中画有正方形的形状，完全可以使用其他形状！文本域也是显示对象，可以采用同样的方式在3D空间中移动。我们将建立一个sprite，在其中增加了一组包含一些随机字母的文本域，然后移动这个sprite。可以在Container3D.as文件看到下面这个例子：

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.text.TextField;
    import flash.text.TextFormat;
    [SWF(width=800, height=800, backgroundColor=0xffffff)]
    public class Container3D extends Sprite
    {
        private var _sprite:Sprite;
        private var _n:Number = 0;

        public function Container3D()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _sprite = new Sprite();
            _sprite.y = stage.stageHeight / 2;

            for(var i:int = 0; i < 100; i++)
            {
                var tf:TextField = new TextField();
                tf.defaultTextFormat = new TextFormat("Arial", 40);
                tf.text = String.fromCharCode(65 + Math.floor(Math.random() * 25));
                tf.selectable = false;
                tf.x = Math.random() * 300 - 150;
                tf.y = Math.random() * 300 - 150;
                tf.z = Math.random() * 1000;
                _sprite.addChild(tf);
            }

            addChild(_sprite);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _sprite.x = stage.stageWidth / 2 + Math.cos(_n) * 200;
            _n += .05;
```

www.TopSage.com

TopSage.com

```
        }
    }
}
```

每个文本域在父sprite中都是三维随机定位。尽管sprite只是在*x*轴上来回移动，但是我们可以看到字母在3D空间中来回移动的视差效果。参见图7-7。

图7-7 运动效果更好的3D容器

7.3 3D旋转

除了在3D空间中移动对象，还可以在任何轴上旋转显示对象。本章开始时我们已经看到了这样一个简单的例子，将一个形状绕*y*轴旋转。我认为没有必要针对*x*轴和*z*轴再把这个例子重复一遍，因为你自己就能完成，也许还能扩展这个类，让对象在3个轴上同时旋转。（如果还没有这么做，可以动手试试看。）

做好准备后，下面来看如何旋转包含有显示对象的容器。首先，重新完成第一个试验，不过首先将形状放在一个sprite中。在本书下载页面可以找到相应的RotateAndPosition.as示例文件：

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

    [SWF(width=800, height=800, backgroundColor = 0xffffff)]
    public class RotateAndPosition extends Sprite
    {
        private var _holder:Sprite;

        public function RotateAndPosition()
        {
            _holder = new Sprite();
            _holder.x = stage.stageWidth / 2;
            _holder.y = stage.stageHeight / 2;
```

www.TopSage.com

```
            addChild(_holder);

            var shape:Shape = new Shape();
            shape.z = 200;
            shape.graphics.beginFill(0xff0000);
            shape.graphics.drawRect(-100, -100, 200, 200);
            _holder.addChild(shape);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _holder.rotationY += 2;
        }
    }
}
```

与前面一样，这只是给出了一个旋转的正方形。不过，现在需要在容器中移动这个正方形。首先处理*x*轴的位置，在构造函数中增加以下粗体显示的代码行：

```
var shape:Shape = new Shape();
shape.x = 200;
shape.graphics.beginFill(0xff0000);
shape.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape);
```

现在不只是旋转，这相当于一种围绕中心的轨道运动。如果首先在*z*轴上移动形状可以得到一个不同的效果：

```
var shape:Shape = new Shape();
shape.z = 200;
shape.graphics.beginFill(0xff0000);
shape.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape);
```

这真是太酷了。下面再创建另一个正方形：

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

    [SWF(width=800, height=800, backgroundColor = 0xffffff)]
    public class RotateAndPosition extends Sprite
    {
        private var _holder:Sprite;
        public function RotateAndPosition()
        {
            _holder = new Sprite();
            _holder.x = stage.stageWidth / 2;
            _holder.y = stage.stageHeight / 2;
            addChild(_holder);

            var shape1:Shape = new Shape();
```

www.TopSage.com

```
            shape1.z = 200;
            shape1.graphics.beginFill(0xff0000);
            shape1.graphics.drawRect(-100, -100, 200, 200);
            _holder.addChild(shape1);

            var shape2:Shape = new Shape();
            shape2.z = -200;
            shape2.graphics.beginFill(0xff0000);
            shape2.graphics.drawRect(-100, -100, 200, 200);
            _holder.addChild(shape2);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _holder.rotationY += 2;
        }
    }
}
```

把一个正方形放在z值为200的位置，另一个放在z值为-200的位置。现在当我们旋转容器时，它们相互之间也存在某种旋转。没有必要限制它只能完成y旋转。下面修改onEnterFrame再在另一个轴上增加旋转：

```
private function onEnterFrame(event:Event):void
{
    _holder.rotationY += 2;
    _holder.rotationX += 1.5;
}
```

没错，确实非常容易。下面再增加更多的正方形！这一次，把新增的正方形沿x轴推到左边和右边，另外还要旋转90°：

```
var shape3:Shape = new Shape();
shape3.x = 200;
shape3.rotationY = 90;
shape3.graphics.beginFill(0xff0000);
shape3.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape3);
var shape4:Shape = new Shape();
shape4.x = -200;
shape4.rotationY = -90;
shape4.graphics.beginFill(0xff0000);
shape4.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape4);
```

现在有了4面墙相互环绕着运动。不要就此止步！你可能会有更好的想法，不过下面增加一个地板和一个天花板（粗略地来看）：

```
var shape5:Shape = new Shape();
shape5.y = 200;
shape5.rotationX = 90;
```

www.TopSage.com

大家网 TopSage.com

```
shape5.graphics.beginFill(0xff0000);
shape5.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape5);

var shape6:Shape = new Shape();
shape6.y = -200;
shape6.rotationX = -90;
shape6.graphics.beginFill(0xff0000);
shape6.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape6);
```

可以看到结果如图7-8所示。

是不是很棒？而且相当简单。这里甚至没有用到一个三角函数。我也很清楚你现在会想什么，是不是认为这就像一个立方相片夹？嗯，也许你所想的不完全如此，不过我相信你肯定在考虑那些红方块太单调，也许需要一些不同的东西。当然，一旦开始改变颜色等方面，你就会破坏我精心制造的假象。不相信？那就试试看。通过在beginFill调用中改变十六进制值，为每个方块指定一个不同的颜色。如果你愿意甚至可以随机地指定。

```
shape1.graphics.beginFill(Math.random() * 0xffffff);
```

你会看到图7-9所示的结果。

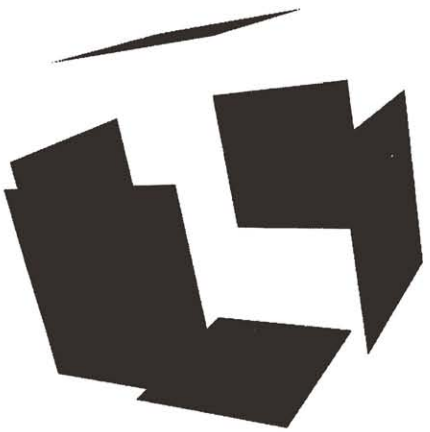


图7-8　旋转的立方体

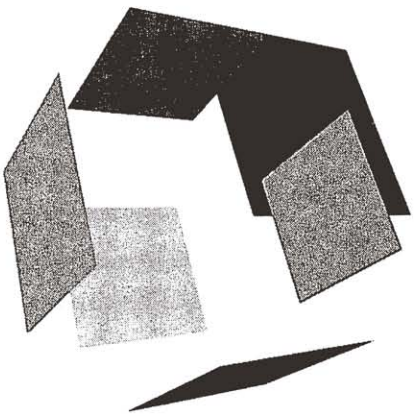


图7-9　Adobe, 出问题了

现在立方体的面不全为红色了，问题就暴露出来了。如果从切屏图还不能清楚地看出这一点，那我可以告诉你：在这里，立方体中某些本该出现在背面的部分现在居然出现在前面。我想你可能已经很清楚，能马上意识到这是深度排序的问题。我们已经了解深度排序以及如何解决这个问题，所以下面直接深入并应用解决方法。

下一个类（可以从RotateAndPosition2.as文件得到）尝试按我们所了解的方法来解决深度问题——按形状的z属性对形状排序，并以该顺序添加到显示列表。我还增加了一个makeShape方法来消除重复的代码：

www.TopSage.com

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

    [SWF(width=800, height=800, backgroundColor = 0xffffff)]
    public class RotateAndPosition2 extends Sprite
    {
        private var _holder:Sprite;
        private var _shapes:Array;
    public function RotateAndPosition2()
    {
        _holder = new Sprite();
        _holder.x = stage.stageWidth / 2;
        _holder.y = stage.stageHeight / 2;
        addChild(_holder);

        var shape1:Shape = makeShape();
        shape1.z = 200;

        var shape2:Shape = makeShape();
        shape2.z = -200;

        var shape3:Shape = makeShape();
        shape3.x = 200;
        shape3.rotationY = 90;

        var shape4:Shape = makeShape();
        shape4.x = -200;
        shape4.rotationY = -90;

        var shape5:Shape = makeShape();
        shape5.y = 200;
        shape5.rotationX = 90;

        var shape6:Shape = makeShape();
        shape6.y = -200;
        shape6.rotationX = 90;

        _shapes = [shape1, shape2, shape3, shape4, shape5, shape6];
        sortShapes();

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function makeShape():Shape
    {
        var shape:Shape = new Shape();
        shape.graphics.beginFill(Math.random() * 0xffffff);
        shape.graphics.drawRect(-100, -100, 200, 200);
        _holder.addChild(shape);
        return shape;
    }
```

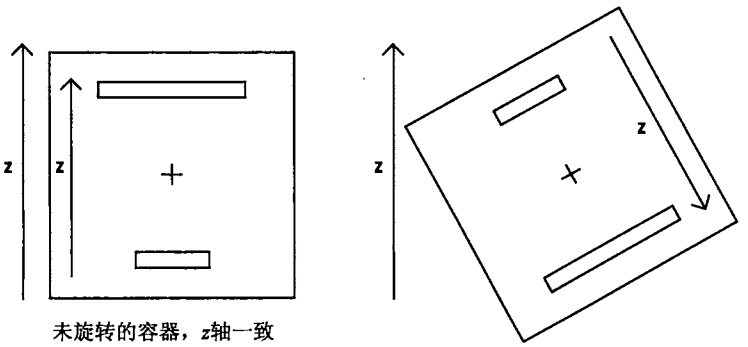
www.TopSage.com

```
        private function sortShapes():void
        {
            _shapes.sortOn("z", Array.NUMERIC | Array.DESCENDING);
            for(var i:int = 0; i < _shapes.length; i++)
                {
                    _holder.addChildAt(_shapes[i] as Shape, i);
                }
            }

            private function onEnterFrame(event:Event):void
            {
                _holder.rotationY += 2;
                _holder.rotationX += 1.5;
                sortShapes();
            }
        }
    }
```

除了清理工作，这里还有一个名为_shapes的数组，其中包含所有形状。每次旋转后，sortShapes方法会被调用。它根据z对形状数组排序，并按照正确的z顺序将各个形状放回到容器holder中。

不过，测试时情况可不太妙。问题在于我们是在容器内部的z轴上对元素排序。所以，尽管确实在z上已正确排序，但是当容器翻转时，它们会突然完全反向。我们需要的是按元素在容器之外出现的顺序对它们进行排序。换句话说，尽管对象A比对象B的z深度更小，但是当容器旋转时对象A应当出现在对象B后面，以保证相对于z轴“反向”（参见图7-10）。



未旋转的容器，z轴一致

旋转之后，局部z不同于全局z

图7-10　旋转一个3D容器的效果

为了解决这个问题，需要编写一个定制排序函数，将容器内部的局部坐标转换为世界坐标（场景坐标或根坐标），然后根据这个坐标进行排序。Array.sort方法允许传入一个函数作为参数。然后可以在一次排序中提供多组对象并多次调用这个函数。如果第一个对象应当放在数组中第二个对象的前面，这个函数会返回一个负数，如果第一个对象应当放在第二个对象的后面，则返回一个正数。如果它们保持不变则返回0。

www.TopSage.com

因此现在需要一种方法将局部坐标转换为世界坐标。这里有很多种方法，包括手动地完成坐标旋转，这涉及大量复杂的三角计算。幸运的是，Flash 10现在提供了一个新类flash.geom.Matrix3D，它包含很多管理3D坐标的有用方法。即使对于目前这个问题，也可能有多种方法可以完成我们的任务。我不知道这里给出的方法是不是最佳方法，但是它确定很不错，而且不太麻烦。

这里利用了Matrix3D类的一个名为deltaTransformVector的方法。基本说来，这个方法所做的就是取一个3D点（存储在一个Vector3D对象中），并对它应用一个3D矩阵的旋转和缩放部分。简单地说，它会根据容器的旋转来旋转所有形状的位置，并指出形状在全局3D空间中的位置。

首先需要有表示容器旋转的Matrix3D。为此可以键入以下代码：

```
container.transform.matrix3D
```

然后可以调用deltaTransformVector方法，并传入一个点的3D位置，这样可以得到它旋转后的位置：

```
rotatedPosition = _holder.transform.matrix3D.deltaTransformVector(originalPosition)
```

如果对两个不同形状的位置做这个处理，就可以知道从全局视点来看哪一个在z轴上更远。需要解决的最后一个难题是如何得到一个显示对象位置的Vector3D表示。可以使用对象的x,y和z属性来动态创建，不过实际上恰好已经存在这样一个表示：

```
displayObject.transform.Matrix3D.position
```

现在我们已经有了得到两个显示对象全局坐标需要具备的一切条件：

```
var posA:Vector3D = objA.transform.matrix3D.position;
posA = _holder.transform.matrix3D.deltaTransformVector(posA);
var posB:Vector3D = objB.transform.matrix3D.position;
posB = _holder.transform.matrix3D.deltaTransformVector(posB);
```

然后可以建立一个排序比较函数，由它确定两个显示对象的前后顺序：

```
private function depthSort(objA:DisplayObject, objB:DisplayObject):int
{
    var posA:Vector3D = objA.transform.matrix3D.position;
    posA = _holder.transform.matrix3D.deltaTransformVector(posA);
    var posB:Vector3D = objB.transform.matrix3D.position;
    posB = _holder.transform.matrix3D.deltaTransformVector(posB);
    return posB.z - posA.z;
}
```

如果从一个旋转的全局视点来看对象A比对象B更远，这个函数就会返回一个负数，指示对象A在数组中应当排在对象B前面。然后可以在sortShapes方法中很容易地实现这一点：

```
private function sortShapes():void
{
    _shapes.sort(depthSort);
    for(var i:int = 0; i < _shapes.length; i++)
    {
        _holder.addChildAt(_shapes[i] as Shape, i);
    }
}
```

www.TopSage.com

现在旋转的形状应该能正确排序了，这样就能创建一个不错的3D对象，如图7-11所示。

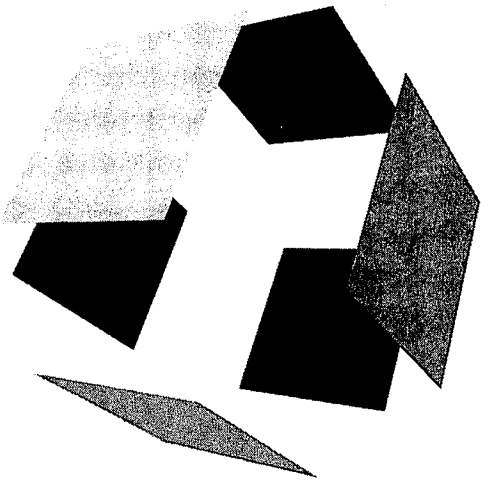


图7-11　正确的深度排序，即使在一个旋转的容器中

如果稍做修改，可以得到一个旋转式布局，这通常用于导航或者显示图库（可以说极其常用）。我不打算在下一个例子中加载任何图像，不过可以很容易地修改sprite，使它成为加载器，并提供一组URL。代码如下，见Carousel.as文件：

```
package
{
    import flash.display.DisplayObject;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Vector3D;

    [SWF(width=800, height=800, backgroundColor = 0xffffff)]
    public class Carousel extends Sprite
    {
        private var _holder:Sprite;
        private var _items:Array;
        private var _radius:Number = 200;
        private var _numItems:int = 5;

        public function Carousel()
        {
            _holder = new Sprite();
            _holder.x = stage.stageWidth / 2;
            _holder.y = stage.stageHeight / 2;
            _holder.z = 0;
            addChild(_holder);

            _items = new Array();
            for(var i:int = 0; i < _numItems; i++)
            {
```

www.TopSage.com

```
            var angle:Number = Math.PI * 2 / _numItems * i;
            var item:Sprite = makeItem();
            item.x = Math.cos(angle) * _radius;
            item.z = Math.sin(angle) * _radius;
            item.rotationY = -360 / _numItems * i + 90;
            _items.push(item);
        }
        sortItems();

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function makeItem():Sprite
    {
        var item:Sprite = new Sprite();
        item.graphics.beginFill(Math.random() * 0xffffff);
        item.graphics.drawRect(-50, -50, 100, 100);
        _holder.addChild(item);
        return item;
    }

    private function sortItems():void
    {
        _items.sort(depthSort);
        for(var i:int = 0; i < _items.length; i++)
        {
            _holder.addChildAt(_items[i] as Sprite, i);
        }
    }

    private function depthSort(objA:DisplayObject, objB:DisplayObject):int
    {
        var posA:Vector3D = objA.transform.matrix3D.position;
        posA = _holder.transform.matrix3D.deltaTransformVector(posA);
        var posB:Vector3D = objB.transform.matrix3D.position;
        posB = _holder.transform.matrix3D.deltaTransformVector(posB);
        return posB.z - posA.z;
    }

    private function onEnterFrame(event:Event):void
    {
        _holder.rotationY += (stage.stageWidth / 2 - mouseX) * .01;
        _holder.y += (mouseY - _holder.y) * .1;
        sortItems();
    }
  }
}
```

这里最重要的变化用粗体显示，即原来如何放置和旋转图像，以及onEnterFrame方法中如何移动容器。这里并不是手动地放置各个方块，而是利用一个循环来完成，将Math.PI * 2弧度（360°）除以项数，并乘以当前项编号。基于所提供的半径，对所得到的角度完成一些三角运算，从而得到各项的*x*和*z*坐标。然后对rotationY属性完成一个类似的计算，这一次直接使用度数。最后，在

www.TopSage.com

大家网 TopSage.com

enterFrame处理程序中根据鼠标位置在y轴上旋转容器，并跟随鼠标沿y轴上下移动。

其结果如图7-12所示。

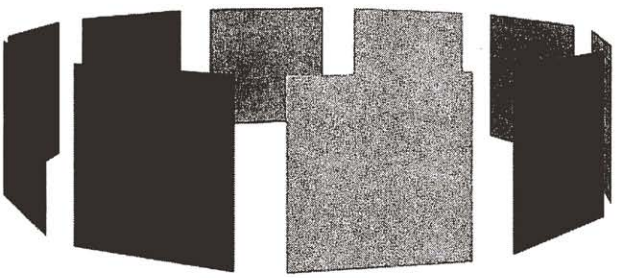


图7-12　一个3D旋转式布局

既然对3D中对象的定位和旋转有了一些基本了解，下面我们来更深入地讨论如何优化3D的外观。

7.4　视场与焦距

显然，在一个平面屏幕上查看任何类型的图像时，实际上你都是在看一个二维的图像。实现3D渲染的程序只是使用了一些技巧从而在一个2D平面上提供了第三维的假象。这些技巧都属于透视领域。

透视的一个技巧是让较远距离的物体出现在较近物体的后面。我们在7.3节讨论深度排序时就是这么处理的。另一个技巧是让较远距离的物体有些淡出，就像蒙着雾一样。类似地，可以让某个深度的对象对焦，而让更近或更远的对象模糊。这称为**景深**（depth of field）。

不过目前为止影响最大的技巧是对象越远就越小，并且让它们逐渐变小最后接近一个没影点。当然，深度排序对于指示第三维也非常重要。不正确的深度排序肯定会破坏3D表象，这一点你已经看到过。不过，如果只是完成深度排序，而没有根据物体的深度对物体进行缩放，立体感也将很有限。

现在有一个重要的问题：对象靠近和远离时如何完成缩放？幸运的是，这个问题原先就存在，并且在个人计算机出现之前就已经由艺术家、工程师以及摄影师提出并做出了解答。这些都要归结于光学以及你眼睛或照相机中透镜的工作方式。如果你做过任何摄影，会知道有广角镜头和远距镜头（介于这二者之间还有大量其他类型的透镜）。甚至还有“鱼眼”透镜，这实际上就是超广角镜实。

广角镜头有一个**宽视场**（field of view）。换句话说，如果在镜头前面投射一个圆弧或锥形来覆盖镜头可以“看到”的区域，这个锥形会很宽并覆盖很大区域。鱼眼透镜可以看到几乎180°角。对于一个远距镜头，这个锥形可能很窄，只覆盖它前面的一个小窄条。

与视场相关的还有一个概念，即**焦距**（focal length），这是镜头中心与**焦点**（focal point）之间的距离（焦点就是穿过镜头的光线所汇聚的点）。参见图7-13。

www.TopSage.com

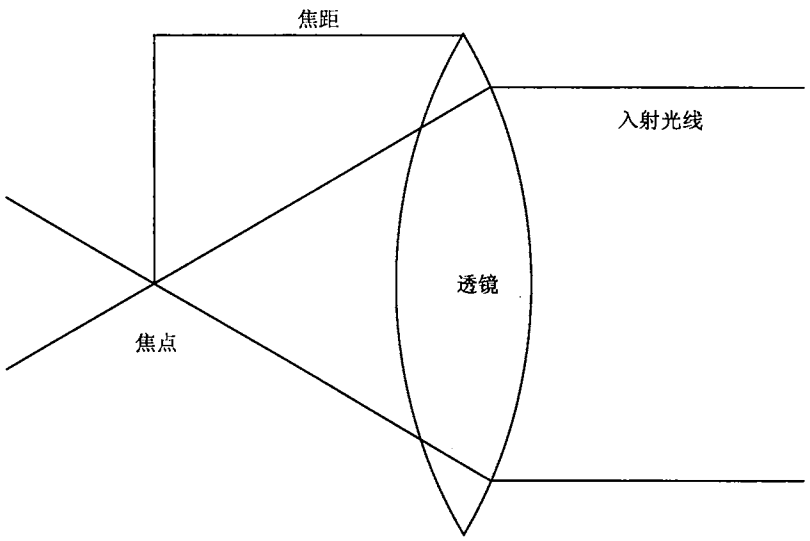


图7-13 透镜的焦距

Adobe帮助文件中对焦距的描述稍有不同，不过概念仍然是相同的。焦距和视场密切相关，也确定了如何缩放或变形。更宽的视场（如广角镜）会得到更短的焦距，并导致更大程度的缩放（这也是用鱼眼透镜拍摄的照片通常看上去严重变形的原因）。越窄的视场（如远距镜头），会得到很长的焦距，变形则轻微得多。对此一个很好的例子就是棒球比赛的照片，从外场拍摄时，尽管击球手距离得远得多，看起来投手和击球手大小却几乎相同。从人的角度来看，这种照片实际上看起来可能严重变形，给人的感觉好像击球手身形非常庞大。从图7-14和图7-15可以看到视场和焦距之间的关系。

在Flash 10 3D中，可以通过设置焦距或视场控制这种变形。实际上，设置其中任意一个属性都会改变另一个属性，不过你可以使用最合用的一个。显示对象的transform属性有一个perspectiveProjection属性，焦距和视场的设置就是通过这个perspectiveProjection属性的两个属性来完成：focalLength和fieldOfView。通常，最好在动画的根目录设置这些属性，除非确实需要让不同的容器或对象有不同的透视状态（可能在不同窗口查看对象）。

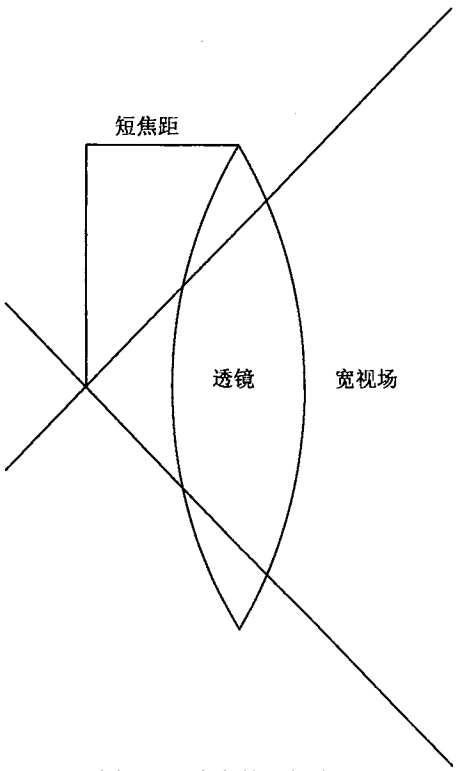


图7-14 广角镜，短焦距

www.TopSage.com

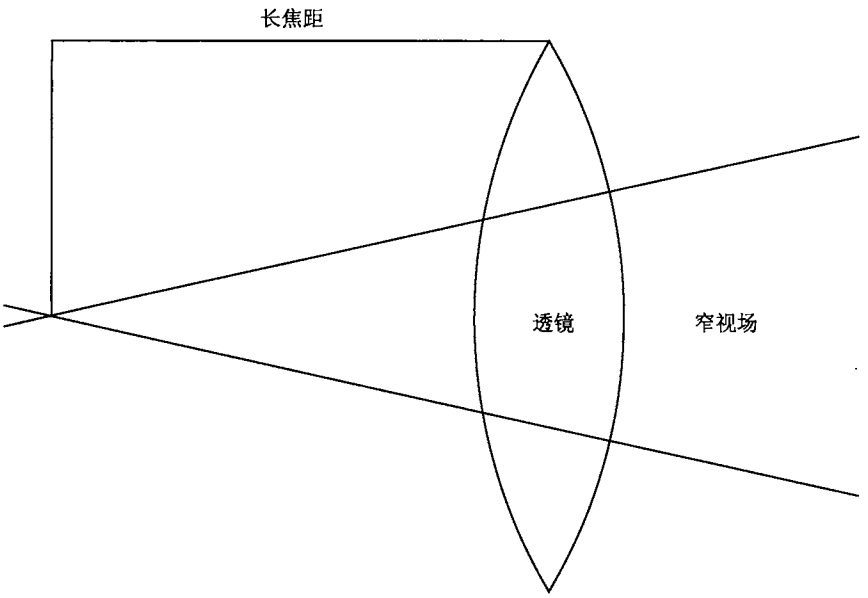


图7-15　窄角（远距）镜，长焦距

视场需要按度数进行度量，这个角度要大于0而且小于180，否则会出现错误。设置视场为0说明无法看到任何事物。将视场设置为180则使焦距为0，我认为尝试计算和渲染一个图像时视场为180可能会导致某些问题。通常你会希望避开接近这两个极端的数值。接近于0的视场会使焦距接近无穷大，这实际上会消除所有3D透视缩放。接近180° 的视场则会提供一个很小的焦距，导致所渲染的图像严重变形。在物理世界中，还有可能得到大于180° 的视场。我所知道的是，已有的最大广角镜头可以有220° 的视场！

不论怎样，我们再回到Flash中，最好对这些值做一些尝试，看看它们有什么影响。可以从7.3节中的旋转布局（Carousel）示例开始。只需在构造函数中增加粗体显示的这行代码：

```
public function Carousel()
{
    root.transform.perspectiveProjection.fieldOfView = 110;
    _holder = new Sprite();
    _holder.x = stage.stageWidth / 2;
    _holder.y = stage.stageHeight / 2;
    _holder.z = 0;
    addChild(_holder);
    ...
```

这样就能得到一个效果非常显著的透视。现在与“摄像头”更靠近的方块更大，这对于照片图库之类的应用来说可能是一个不错的设置。参见图7-16。

再把它设置为25，这会使视场显著变窄。现在透视缩放则不太容易注意到。参见图7-17。

还应当对各种很大和很小的焦距值做些尝试，看看它们的影响，如下例所示：

```
root.transform.perspectiveProjection.focalLength = 300;
```

www.TopSage.com

TopSage.com

重申一次，短焦距会得到更大变形；大焦距导致的变形则较小。

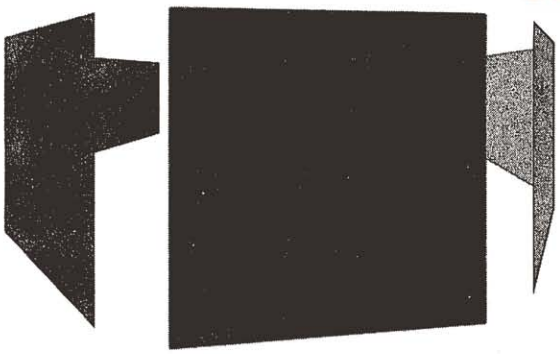


图7-16 宽视场意味着更大的透视变形

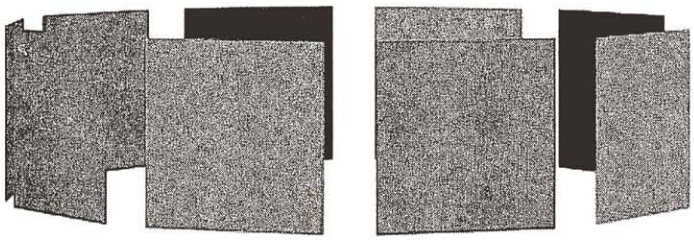


图7-17 窄视场，变形较小

7.5 屏幕和3D坐标

有时，你可能需要找出与3D空间中某一点对应的屏幕坐标。或者反过来，可能需要取屏幕上的一点，得出它在3D空间会变换到什么位置。幸运的是，显示对象有两个内置方法来完成这个任务：local3DtoGlobal和globalToLocal3D。第一个方法从一个flash.geom.Vector3D对象转换为一个2D flash.geom.Point对象，第二个方法正好反过来，将一个2D flash.geom.Point对象转换为flash.geom.Vector3D对象。

首先来看local3DToGlobal方法的具体工作。我们将建立一个sprite，在3D空间中移动。这个sprite中有一些图形，并在x=200，y=0，z=0的位置上绘制有一个圆。然后创建另一个sprite，它将局部3D点（200,0,0）转换为全局屏幕坐标来跟踪前一个圆的移动。下面是这个类，可以在LocalGlobal.as文件中找到：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Point;
    import flash.geom.Vector3D;
```

www.TopSage.com

```
    [SWF(width=800, height=800)]
    public class LocalGlobal extends Sprite
    {
        private var _sprite:Sprite;
        private var _tracker:Sprite;
        private var _angle:Number = 0;


        public function LocalGlobal()
        {
            _sprite = new Sprite();
            _sprite.graphics.lineStyle(10);
            _sprite.graphics.lineTo(200, 0);
            _sprite.graphics.drawCircle(200, 0, 10);
            _sprite.x = 400;
            _sprite.y = 400;
            addChild(_sprite);
            _tracker = new Sprite();
            _tracker.graphics.lineStyle(2, 0xff0000);
            _tracker.graphics.drawCircle(0, 0, 20);
            addChild(_tracker);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }


        private function onEnterFrame(event:Event):void
        {
            _sprite.rotationX += 1;
            _sprite.rotationY+= 1.2;
            _sprite.rotationZ += .5;
            _sprite.x = 400 + Math.cos(_angle) * 100;
            _sprite.y = 400 + Math.sin(_angle) * 100;
            _sprite.z = 200 + Math.cos(_angle * .8) * 400;
            _angle += .05;
            var p:Point =  _sprite.local3DToGlobal(new Vector3D(200, 0, 0));
            _tracker.x = p.x;
            _tracker.y = p.y;
        }

    }
}
```

构造函数创建一个3D sprite和一个跟踪sprite，并在其中放置一些图形。EnterFrame处理程序包含的代码主要是让这个sprite以一种看上去随机的方式在3D空间中移动。它会同时在3个轴上移动和旋转，这里使用了我随机设置的一些数值。最重要的部分是local3DToGlobal行，其中将（200,0,0）转换为一个2D Point对象，然后将这个位置赋至跟踪sprite。运行这段代码时会看到，尽管这个sprite在随意移动（3个轴上的位置都在变化），但跟踪sprite总能毫无问题地全程跟踪这个圆。参见图7-18。

对此可能有各种各样的用法，比如了解一个3D对象何时出屏幕。一个3D对象的x或y坐标可能比屏幕坐标大很多，但是由于它在z轴上离得很远因此仍可见，所以能知道一个对象是否已经

www.TopSage.com

出屏幕可能很有用。

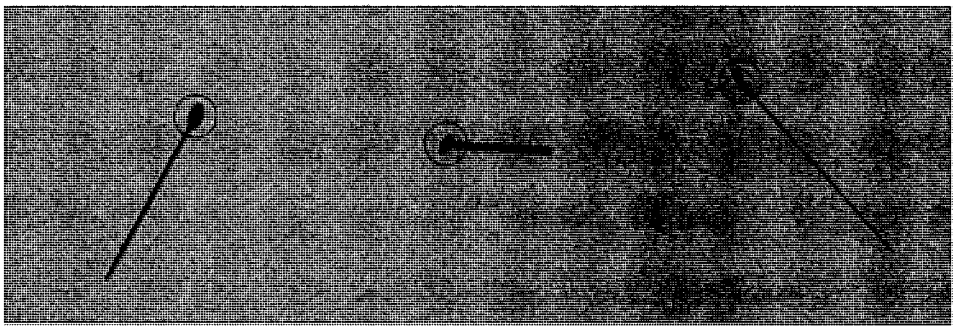


图7-18 在2D空间跟踪一个3D点的多个切屏图

另一方面，也可以从一个屏幕位置转换为一个局部3D坐标。对上面的文件做几处修改，现在这个类为GlobalLocal，放在GlobalLocal.as文件中。主要修改用粗体突出显示：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Point;
    import flash.geom.Vector3D;

    [SWF(width=800, height=800)]
    public class GlobalLocal extends Sprite
    {
        private var _sprite:Sprite;
        private var _tracker:Sprite;
        private var _angle:Number = 0;


        public function GlobalLocal()
        {
            _sprite = new Sprite();
            _sprite.graphics.lineStyle(5);
            _sprite.graphics.drawRect(-200, -200, 400, 400);
            _sprite.x = 400;
            _sprite.y = 400;
            addChild(_sprite);

            _tracker = new Sprite();
            _tracker.graphics.lineStyle(2, 0xff0000);
            _tracker.graphics.drawCircle(0, 0, 20);
            _sprite.addChild(_tracker);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _sprite.rotationX += 1;
```

www.TopSage.com

大家网 TopSage.com

```
            _sprite.rotationY+= 1.2;
            _sprite.rotationZ += .5;
            _sprite.x = 400 + Math.cos(_angle) * 100;
            _sprite.y = 400 + Math.sin(_angle) * 100;
            _sprite.z = 200 + Math.cos(_angle * .8) * 400;
            _angle += .05;
            var p:Vector3D = _sprite.globalToLocal3D(new Point(mouseX, mouseY));
            _tracker.x = p.x;
            _tracker.y = p.y;
        }

    }
}
```

这里在旋转sprite中画一个大方块，并把跟踪者放在其中。在enterFrame处理程序中，调用globalToLocal3D得到对应当前鼠标位置的3D坐标。这会作为一个Vector3D对象返回。在这种情况下，这个Vector3D的z属性总是0，所以只需使用其x和y属性来设置旋转sprite中跟踪者的位置。可以看到，它会在3D空间中追随鼠标移动，而鼠标是在2D空间中移动的。如图7-19所示。

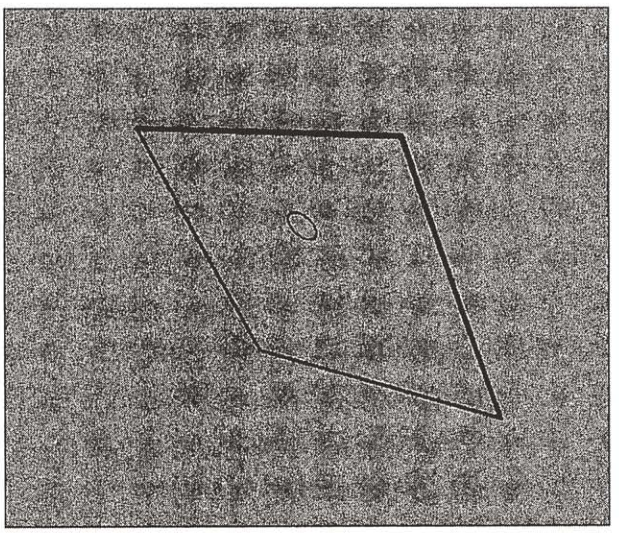


图7-19　在3D空间中跟踪2D坐标

尽管这很不错，不过我发现还有一种方法也可以达到同样的目的，而且简单得多：使用旋转对象的局部鼠标坐标。可以看到，如果访问一个3D转换对象的mouseX和mouseY属性，它会自动地完成全局到局部的3D转换。所以onEnterFrame处理程序可以更为简单：

```
private function onEnterFrame(event:Event):void
{
    _sprite.rotationX += 1;
    _sprite.rotationY+= 1.2;
    _sprite.rotationZ += .5;
    _sprite.x = 400 + Math.cos(_angle) * 100;
```

www.TopSage.com

```
        _sprite.y = 400 + Math.sin(_angle) * 100;
        _sprite.z = 200 + Math.cos(_angle * .8) * 400;
        _angle += .05;
        _tracker.x = _sprite.mouseX;
        _tracker.y = _sprite.mouseY;
    }
```

这会完成同样的工作。应该认识到，这只适用于转换鼠标坐标的情况。如果想要将场景中一个对象的坐标转换为局部3D坐标，仍然需要使用转换函数。

7.6 指向

一旦熟悉了Flash 10 3D，也许你希望在帮助文件中查看相关的各个类，看看到底有哪些类。可以从flash.geom包开始，其中包含了Matrix3D、Orientation3D、PerspectiveProjection、Utils3D、Vector3D等类。所有这些类都提供了丰富的方法，可以帮助你完成各种3D计算。我发现Matrix3D类中有一个非常不错的方法：pointAt方法。

pointAt方法接收一个Vector3D对象作为指向目标。如果从一个显示对象transform属性的matrix3D对象调用，这个方法会在3D空间中旋转这个显示对象指向指定位置。下面的类（见于FollowMouse3D.as文件）展示了这个方法的具体工作：

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.geom.Vector3D;

    [SWF(width=800, height=800, backgroundColor = 0xffffff)]
    public class FollowMouse3D extends Sprite
    {
        private var _sprite:Sprite;
        private var _angleX:Number = 0;
        private var _angleY:Number = 0;
        private var _angleZ:Number = 0;
        public function FollowMouse3D()
        {
            _sprite = new Sprite();
            _sprite.x = 400;
            _sprite.y = 400;
            _sprite.z = 200;
            _sprite.graphics.beginFill(0xff0000);
            _sprite.graphics.moveTo(0, 50);
            _sprite.graphics.lineTo(-25, 25);
            _sprite.graphics.lineTo(-10, 25);
            _sprite.graphics.lineTo(-10, -50);
            _sprite.graphics.lineTo(10, -50);
            _sprite.graphics.lineTo(10, 25);
            _sprite.graphics.lineTo(25, 25);
            _sprite.graphics.lineTo(0, 50);
            _sprite.graphics.endFill();
```

www.TopSage.com

```
            addChild(_sprite);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _sprite.x = 400 + Math.sin(_angleX += .11) * 200;
            _sprite.y = 400 + Math.sin(_angleY += .07) * 200;
            _sprite.z = Math.sin(_angleZ += .09) * 200;
            _sprite.transform.matrix3D.pointAt(new Vector3D(mouseX, mouseY, 0));

        }
    }
}
```

构造函数中主要包含一组lineTo代码，用来在一个sprite中绘制箭头。

onEnterFrame方法的代码主要是在3D空间移动这个sprite，这与LocalGlobal示例中并无不同。最后一行在sprite transform属性的matrix3D对象上调用了pointAt方法。它传入一个新的Vector3D对象（这个对象根据鼠标x和y坐标建立，另外增加z轴坐标为0）。同样地，每一帧sprite（包括所绘制的箭头）都会指向鼠标。它会到处移动，你可以随意移动鼠标，但绝对不会摆脱跟踪（参见图7-20）。

图7-20　3D中指向鼠标

我不知道这个例子本身有多少实用性，不过可以想到这种技术在构建3D游戏时几种可能的用法，比如转向、瞄准等。

7.7　小结

尽管本章介绍的内容很多，但我们还只是触及Flash 10中3D的一点皮毛。第8章讨论新的绘图API特性时还会讨论更多3D主题，不过希望本章能让你对这个领域研究的内容有所认识。再次指出，请查阅有关文档，特别是flash.display.DisplayObject和flash.geom包中的所有内容。相信你会从中找到乐趣。

www.TopSage.com

大家网
TopSage.com

第8章

Flash 10绘图API

8

如果你接触Flash不足6年，可能对ActionScript绘制应用程序编程接口（Application Programming Interface，API）还没有全面的理解。在我看来，2002年底Flash MX中发布绘图API是利用Flash脚本实现图形设计的一次革命。在Flash MX之前，根本没有办法动态地生成任何图形内容。现在这么讲似乎让人难以置信，但这确实是事实——如果希望在SWF中显示某个图形，就必须在场景中绘制它，或者在创作阶段将它导入到库中，再或者是在运行时加载它。很自然的，对此也有一些技巧，如在库中维护一个动画片段（其中只包含一条线），并为这个单线片段关联多个实例，完成这些实例的缩放和定位来创建动态图形。

不过对于Flash MX，我们第一次可以直接从FLA开始而且只需要编写代码，然后就可以完全动态地创建各种图形内容。尽管当时的API简陋至极（绘图命令仅限于moveTo、lineTo和curveTo，后来又增加了一些命令来绘制矩形、圆形和椭圆），但却已经可以用于创建Flash平台上的大量图形。实际上，很多复杂应用程序的整个界面都是通过绘图API命令建立的。

基于以上背景，下面来介绍Flash 10的升级版绘制API，它无所不能又小巧玲珑。实际上，这些“补充”使原有的API在新方法数量、新对象数量、强大能力以及复杂性方面都显得相形见绌。所以下面来进行深入讨论，看看是否可以充分利用所有这些“补充”。

8.1 路径

我们要讨论的第一个主题是路径，所以我要再一次定义这个术语——这个定义与第4章中使用的定义完全不同。从新的绘图API的角度来讲，**路径**（path）是一系列的点以及与之关联的一系列绘制命令（moveTo、lineTo和curveTo），这些命令用于在Graphics对象中绘制一个形状。这与你小时候玩的“连点成线”差不多。

早期（Flash MX到Flash CS3时期），要分别调用moveTo、lineTo或curveTo方法，并传入适当的值。最开始时，你可能会在长长的一组图形方法调用中硬编码写入一些值。然后你发现可以把位置数据保存在数组中，并循环处理这个数组，反复调用lineTo分别利用数组中的下一组值画线。如果你水平很高，可能甚至会设计一种方法对不同绘制方法编码，并存储一个绘制方法数组，这样一来，你可以先调用moveTo，然后是一组lineTo方法，如果需要还可以调用一个curveTo方法。因此，你会有一组命令和一组点来构成一个形状。如果你能达到这个程度，或者至少了解这样做时会如何工作，那么理解新的绘图API中的路径就不会有任何问题，因为这正是绘图API中的路

www.TopSage.com

径：一组点和一组绘制命令。

最早接触到路径是在flash.display.Graphics类的drawPaths方法中。下面来看它的最简形式：

```
drawPath(commands:Vector.<int>, data:Vector.<Number>)
```

这里将一组命令存储在一个int向量中，并把一组数据存储在一个Number向量中。命令分别是moveTo、lineTo和curveTo（稍后还会看到另外一些命令）。这些命令都编码为整型，并存储为flash.display.GraphicsPathCommand类的静态常量成员，如下所示：

```
public static const NO_OP:int = 0;
public static const MOVE_TO:int = 1;
public static const LINE_TO:int = 2;
public static const CURVE_TO:int = 3;
public static const WIDE_MOVE_TO:int = 4;
public static const WIDE_LINE_TO:int = 5;
```

编号1、2、3对应的命令对你来说应该很熟悉了。0命令NO_OP就像是一个空命令，告诉绘图API什么也不做。稍后将介绍这个命令如何使用，此外还将介绍最后的两个wide_命令是什么意思。

要画线时，通常会移到某个点，并向另一个点画一条线。如果把它编码到一个命令向量中，可以按照如下方式完成：

```
var commands:Vector.<int> = new Vector.<int>();
commands[0] = GraphicsPathCommand.MOVE_TO;
commands[1] = GraphicsPathCommand.LINE_TO;
```

或者，可以直接使用int值：

```
var commands:Vector.<int> = new Vector.<int>();
commands[0] = 1;
commands[1] = 2;
```

当然，这样无法表达你在做什么，而且还失去了编译时检查。你很可能会输入一个17作为命令，认为这代表某种含义，或者只是随手而为。Flash会顺利地通过编译，因为这确实是一个int，但是程序肯定不会提供你预期的结果。

现在需要一个data向量，其中包含一些点，我们想要在这些点之间画线。可能你会认为这是一个Point对象向量，其中各个元素分别包含一个x和一个y值。不过，可以看到，其实这是一个Number向量。所以第一个元素表示第一个点的x值，第二个元素表示第一个点的y值。接下来两个元素表示了下一个点的值，依次类推。尽管开始时这让人有些困惑，不过不论从执行速度还是内存消耗方面看，存储和访问基本类型值（如Number）向量都要比复杂对象（Point）向量更为高效。对绘图API的这些新补充正是为提高速度专门引入的。需要保证的一点是，一定要有足够的数据点与命令个数匹配。如果点的数量少于命令的数量，Flash会默认使用0作为数据，而这通常是没有意义的。

下面从（100,100）到（250,200）画一条线。以下是data向量：

www.TopSage.com

```
var data:Vector.<Number> = new Vector.<Number>();
data[0] = 100;
data[1] = 100;
data[2] = 250;
data[3] = 200;
```

现在可以调用drawPath并传入命令和数据，然后可以看到一条线。当然，我们还需要先设置一个lineStyle。路径只表示点以及绘制操作的类型。还需要以老式的方式指定填充或笔划（后面还会看到如何对线型和填充编码）。

```
graphics.lineStyle(0);
graphics.drawPath(commands, data);
```

现在来看一个能实际执行的完整的类。这个类可以从本书下载页面（www.friendsofed.com）的SingleLine.as文件得到：

```
package
{
    import flash.display.GraphicsPathCommand;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class SingleLine extends Sprite
    {
        public function SingleLine()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            var commands:Vector.<int> = new Vector.<int>();
            commands[0] = GraphicsPathCommand.MOVE_TO;
            commands[1] = GraphicsPathCommand.LINE_TO;

            var data:Vector.<Number> = new Vector.<Number>();
            data[0] = 100;
            data[1] = 100;
            data[2] = 250;
            data[3] = 200;

            graphics.lineStyle(0);
            graphics.drawPath(commands, data);
        }
    }
}
```

你可能正在想只是画一条线就需要这么多的代码？确实如此。如果就画一条线，当然更容易的做法是直接调用moveTo和lineTo，这就万事大吉了。不过随着后面例子的展开，你会看到，这些新对象和方法是为更复杂的绘制操作而设计的，如绘制非常复杂的形状。事实上，这些补充的对象的方法的主要意图就是支持3D绘图。当然，本章后面也会谈到这一点。

不过首先来看drawPath的一个更实用的用法。

www.TopSage.com

大家网 TopSage.com

8.1.1　一个简单的绘图程序

如果熟悉软件设计模式，可能会看到这里开始出现一些命令模式。在命令模式中，对象表示动作。你要创建一个命令对象，其中封装一个动作以及该动作所用的参数。实际上这里就存储了一组动作（绘制命令）及其参数（数据）。

在下面的例子中，我们将创建一个很小的绘图程序，允许用户画一些形状草图。它把绘制的图形存储为一组命令和数据。以后只需再次调用drawPath就可以随时重绘这个图形。在这个例子中，我们允许用户使用上下方向键来改变线条的粗细，还可以使用空格键设置一种随机的颜色。然后程序会使用新的线型重绘草图。当然，在实际的应用中，可能会提供更有用的控件来让用户选择线宽和颜色，不过我们关注的是绘图API中与drawPath方法的有关内容。这个类如下所示，你可以在PathSketch.as文件中找到：

```
package
{
    import flash.display.GraphicsPathCommand;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.KeyboardEvent;
    import flash.events.MouseEvent;
    import flash.ui.Keyboard;
    [SWF(backgroundColor=0xffffff)]
    public class PathSketch extends Sprite
    {
        private var commands:Vector.<int> = new Vector.<int>();
        private var data:Vector.<Number> = new Vector.<Number>();

        private var lineWidth:Number = 0;
        private var lineColor:uint = 0;

        public function PathSketch()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
            stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
        }


        private function onMouseDown(event:MouseEvent):void
        {
            commands.push(GraphicsPathCommand.MOVE_TO);
            data.push(mouseX, mouseY);
            stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
            stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
            draw();
        }

        private function onMouseMove(event:MouseEvent):void
```

www.TopSage.com

```
        {
            commands.push(GraphicsPathCommand.LINE_TO);
            data.push(mouseX, mouseY);
            draw();
        }

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

        private function onKeyUp(event:KeyboardEvent):void
        {
            if(event.keyCode == Keyboard.DOWN)
            {
                lineWidth = Math.max(0, lineWidth -1);
            }
            else if(event.keyCode == Keyboard.UP)
            {
                lineWidth++;
            }
            else if(event.keyCode == Keyboard.SPACE)
            {
                lineColor = Math.random() * 0xffffff;
            }
            draw();
        }

        private function draw():void
        {
            graphics.clear();
            graphics.lineStyle(lineWidth, lineColor);
            graphics.drawPath(commands, data);
        }
    }
}
```

这个程序首先为命令和数据建立向量，并为mouseDown和keyUp事件指定监听器。我们从这里开始分析。首先来分析mouseDown处理程序：

```
private function onMouseDown(event:MouseEvent):void
{
    commands.push(GraphicsPathCommand.MOVE_TO);
    data.push(mouseX, mouseY);
    stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    draw();
}
```

用户在场景上按下鼠标按钮时就会调用这个方法。我们希望绘图API移动到这个点，所以在commands向量中压入一个MOVE_TO，再把鼠标的x和y位置压入data向量。然后监听mouseMove和mouseUp事件并调用draw方法。

www.TopSage.com

onMouseUp处理程序只是删除事件监听器，因此不再多做说明。不过可以再来看看onMouseMove处理程序：

```
private function onMouseMove(event:MouseEvent):void
{
    commands.push(GraphicsPathCommand.LINE_TO);
    data.push(mouseX, mouseY);
    draw();
}
```

这与onMouseDown非常类似，它向commands向量压入一个LINE_TO，并把鼠标位置压入data向量。现在commands向量将包含一个MOVE_TO和LINE_TO列表，data将包含一个表示x和y坐标的长长的数字列表。然后调用draw。需要指出，在onMouseDown方法中第一次调用draw时，只是把MOVE_TO压入了commands向量，所以绘图API只是移动到该点。在这里，我们压入了LINE_TO，这样API就会从起始点向鼠标移动到的位置画一条线。考虑到这一点，下面来看draw方法的具体内容：

```
private function draw():void
{
    graphics.clear();
    graphics.lineStyle(lineWidth, lineColor);
    graphics.drawPath(commands, data);
}
```

非常简单。首先清除图形，根据lineWidth和lineColor属性（初始时都设置为0）设置线型，调用drawPath，并传入commands和data。这里不打算介绍onKeyUp方法——这个方法只是检查按下了哪个键，并相应地改变线条宽度或颜色。

现在可以在场景上画多条线，使用方向键来改变这些线的笔划粗细，并使用空格改变线的颜色。现在完成的工作你以前也可以做到，但是之前你必须确定如何对moveTo和lineTo编码，以及如何创建循环来完成绘制，现在则要容易得多。

在这方面还可以更为深入——将各组线段区分为各自的路径，从而允许撤销、重做等操作。在本章最后一节中你将看到另外一些工具，利用它们可以更容易地完成这些工作。不过，首先来看与路径有关的其他功能。

8.1.2　绘制曲线

利用drawPath画曲线与画直线并没有太大区别。当然，需要将命令从LINE_TO改为CURVE_TO，还需要提供额外的数据——一对表示控制点的数以及另外一对表示终点的数。类似于LINE_TO，通常首先用一个MOVE_TO或者另外某个绘制操作来设置曲线的起始点。

这个例子力求简单，因为本章要介绍的内容还有很多。这里只是建立了8个点，并在它们之间画一些曲线。下面这个类可以通过下载CurveDrawing.as文件得到：

```
package
{
    import flash.display.GraphicsPathCommand;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
```

www.TopSage.com

TopSage.com

```
[SWF(backgroundColor=0xffffff)]
public class CurveDrawing extends Sprite
{
    private var commands:Vector.<int> = new Vector.<int>();
    private var data:Vector.<Number> = new Vector.<Number>();
    public function CurveDrawing()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        data.push(200, 200);

        data.push(250, 100);
        data.push(300, 200);

        data.push(400, 250);
        data.push(300, 300);

        data.push(250, 400);
        data.push(200, 300);

        data.push(100, 250);
        data.push(200, 200);

        commands.push(GraphicsPathCommand.MOVE_TO);
        commands.push(GraphicsPathCommand.CURVE_TO);
        commands.push(GraphicsPathCommand.CURVE_TO);
        commands.push(GraphicsPathCommand.CURVE_TO);
        commands.push(GraphicsPathCommand.CURVE_TO);

        graphics.lineStyle(0);
        graphics.drawPath(commands, data);
    }
}
}
```

利用这个类可以得到如图8-1所示的图，这里已经加了标注来指出点的位置以及绘制命令所做的工作。

再次说明，对于这种工作来说，利用原来的绘图API完成可能更简单，不过从前面的例子你可能已经看出如何将它演化为一个更复杂的绘图程序。在后面的介绍中还会看到更真实的示例。

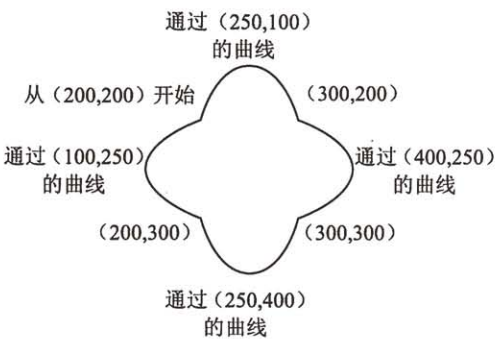


图8-1 利用路径绘制的曲线

8.1.3 Wide 绘制命令和 NO_OP

假设你想使用上一个例子中创建的数据集，并画一些直线来进一步扩展这个形状。首先利用LINE_TO（而不是CURVE_TO）创建一个单独的命令向量，如下所示（这个类可以在LineAndCurveDrawing.as文件中找到）：

www.TopSage.com

```
package
{
    import flash.display.GraphicsPathCommand;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class LineAndCurveDrawing extends Sprite
    {
        private var commands:Vector.<int> = new Vector.<int>();
        private var lineCommands:Vector.<int> = new Vector.<int>();
        private var data:Vector.<Number> = new Vector.<Number>();


        public function LineAndCurveDrawing()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            data.push(200, 200);

            data.push(250, 100);
            data.push(300, 200);

            data.push(400, 250);
            data.push(300, 300);

            data.push(250, 400);
            data.push(200, 300);

            data.push(100, 250);
            data.push(200, 200);

            commands.push(GraphicsPathCommand.MOVE_TO);
            commands.push(GraphicsPathCommand.CURVE_TO);
            commands.push(GraphicsPathCommand.CURVE_TO);
            commands.push(GraphicsPathCommand.CURVE_TO);
            commands.push(GraphicsPathCommand.CURVE_TO);
            lineCommands.push(GraphicsPathCommand.MOVE_TO);
            lineCommands.push(GraphicsPathCommand.LINE_TO);
            lineCommands.push(GraphicsPathCommand.LINE_TO);
            lineCommands.push(GraphicsPathCommand.LINE_TO);
            lineCommands.push(GraphicsPathCommand.LINE_TO);

            graphics.lineStyle(0);
            graphics.drawPath(commands, data);
            graphics.lineStyle(0, 0x000000, .5);
            graphics.drawPath(lineCommands, data);
        }
    }
}
```

不过，这会得到如图8-2所示的图形。

www.TopSage.com

这里的问题是，CURVE_TO要消耗两个点，而LINE_TO只消耗一个点。由于我们只放入了4个LINE_TO，所以只完成了一半。再向lineCommands向量中压入4个LINE_TO，可以看到如图8-3所示的图形。

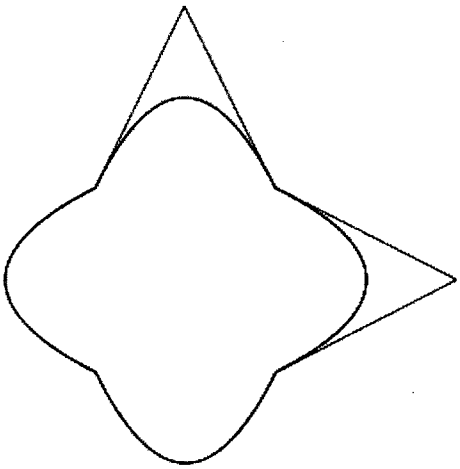


图8-2 曲线和直线

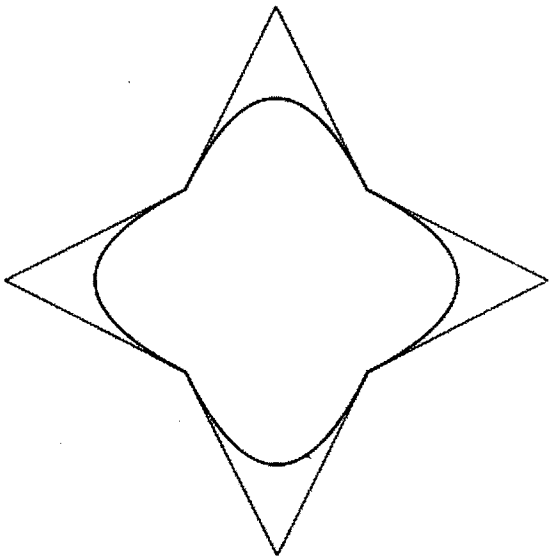


图8-3 曲线和更多直线

还不错。不过现在假设我们希望将各个曲线替换为直线，使之从各曲线的一个端点直接到另一个端点，而忽略控制点。如果只是对LINE_TO使用以上同一个数据集，这是办不到的。必须重新构造一个新的数据集，每隔一个值抽取一个数据。

这里引入了WIDE_LINE_TO命令。这个命令会跳过下一对数，而向后面一对数表示的点画一条线。因此可以直接用WIDE_LINE_TO命令替换CURVE_TO命令，这会忽略控制点，并向终点直接画一条直线。因此，可以将4个LINE_TO命令替换为WIDE_LINE_TO命令：

```
lineCommands.push(GraphicsPathCommand.MOVE_TO);
lineCommands.push(GraphicsPathCommand.WIDE_LINE_TO);
lineCommands.push(GraphicsPathCommand.WIDE_LINE_TO);
lineCommands.push(GraphicsPathCommand.WIDE_LINE_TO);
lineCommands.push(GraphicsPathCommand.WIDE_LINE_TO);
```

这会得到如图8-4所示的结果。

类似地，WIDE_MOVE_TO会跳过数据向量中的下一对数，而移向再下一个点（不需要具体的例子，你应该也能完全理解这个命令）。

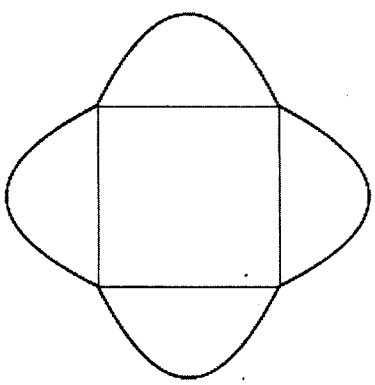


图8-4 曲线和wide线

你可能认为使用NO_OP命令可以跳过控制点，如下：

www.TopSage.com

大家网 TopSage.com

```
lineCommands.push(GraphicsPathCommand.MOVE_TO);
lineCommands.push(GraphicsPathCommand.NO_OP);
lineCommands.push(GraphicsPathCommand.LINE_TO);
lineCommands.push(GraphicsPathCommand.NO_OP);
lineCommands.push(GraphicsPathCommand.LINE_TO);
lineCommands.push(GraphicsPathCommand.NO_OP);
lineCommands.push(GraphicsPathCommand.LINE_TO);
lineCommands.push(GraphicsPathCommand.NO_OP);
lineCommands.push(GraphicsPathCommand.LINE_TO);
```

遗憾的是，这是做不到的。

NO_OP命令是一个空命令，使用这个命令时什么也不做。它不会画直线或曲线，也不会移动绘制光标。不过，它同样不会消耗数据列表中的任何点。因此，下一个LINE_TO命令仍会继续向列表中的下一个点直接画线。对于什么情况下确实需要使用NO_OP我举不出实际的例子，不过能大致想出几种可能性。例如，如果想跳过图形某一部分的绘制，可以把对应这些点的绘制命令换为NO_OP命令，这样就能保存数据，以备以后有可能需要再次绘制整个图形。

8.1.4　环绕

关于路径最后要讨论的内容是**环绕**（winding），这是指构成一个形状的点是有方向性的。众所周知，形状是用点之间的直线或曲线绘制的，应该可以看到，由于这些点构成了一个形状，因此它们会有顺时针走向或逆时针走向。顺时针方向称为**正环绕**（positive winding），逆时针方向称为**负环绕**（negative winding）。当然，可能还有一种“8字形”，这可以是正环绕也可以是负环绕。这并没有问题，因为可以单独讨论形状的每一部分。

对于一个形状来说，其环绕的方向实际上不会带来任何区别。只有在有相同填充的路径出现相交或重叠时这才变得重要。在一对beginFill/endFill操作之间绘制两个重叠的路径或者绘制一个与自身重叠的路径时，重叠区域存在两种可能性——可能由形状的其余部分填充，也可能重叠路径相互抵消——留下一个未填充的区域。要了解具体情况，下面来看下一个例子（见WindingDemo.as文件）：

```
package
{
    import flash.display.GraphicsPathCommand;
    import flash.display.GraphicsPathWinding;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class WindingDemo extends Sprite
    {
        private var commands:Vector.<int> = new Vector.<int>();
        private var data1:Vector.<Number> = new Vector.<Number>();
        private var data2:Vector.<Number> = new Vector.<Number>();


        public function WindingDemo()
```

www.TopSage.com

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

            commands.push(GraphicsPathCommand.MOVE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);

            data1.push(150, 100);
            data1.push(200, 100);
            data1.push(200, 250);
            data1.push(150, 250);
            data1.push(150, 100);

            data2.push(100, 150);
            data2.push(250, 150);
            data2.push(250, 200);
            data2.push(100, 200);
            data2.push(100, 150);

            graphics.beginFill(0xff0000);
            graphics.drawPath(commands, data1);
            graphics.drawPath(commands, data2);
            graphics.endFill();
        }
    }
}
```

这里创建了两个矩形——一个垂直，一个水平——它们相互重叠构成一个十字型。下面在一对beginFill和endFill语句之间使用drawPath绘制这两个矩形，所以它们都有相同的填充。默认情况下，交叉区域是未填充的，如图8-5所示。

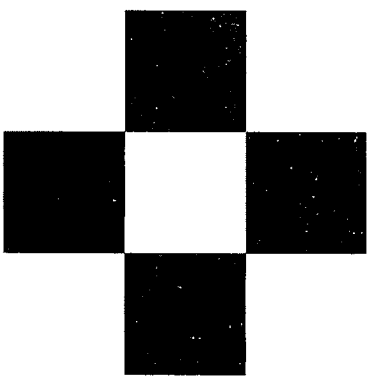


图8-5 交叉填充自我抵消

需要指出，这个行为的关键点在于所绘制路径要有相同的填充，此时才会出现这种行为。要了解这一点，下面尝试修改最后几行代码，如下所示：

```
graphics.beginFill(0xff0000);
graphics.drawPath(commands, data1);
graphics.endFill();
graphics.beginFill(0xff0000);
graphics.drawPath(commands, data2);
graphics.endFill();
```

由于这里结束了第一个填充，而且在绘制第二个路径之前开始了一个新的填充，因此尽管两个路径重叠，但涵盖两个矩形的整体区域都会被填充。

还要注意，一个路径很容易与自身重叠，下面是一个稍做了修改的例子，从中可以看出这一点：

www.TopSage.com

```
package
{
    import flash.display.GraphicsPathCommand;
    import flash.display.GraphicsPathWinding;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class WindingDemo extends Sprite
    {
        private var commands:Vector.<int> = new Vector.<int>();
        private var data1:Vector.<Number> = new Vector.<Number>();


        public function WindingDemo()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            commands.push(GraphicsPathCommand.MOVE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);

            commands.push(GraphicsPathCommand.MOVE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);

            data1.push(150, 100);
            data1.push(200, 100);
            data1.push(200, 250);
            data1.push(150, 250);
            data1.push(150, 100);

            data1.push(100, 150);
            data1.push(250, 150);
            data1.push(250, 200);
            data1.push(100, 200);
            data1.push(100, 150);

            graphics.beginFill(0xff0000);
            graphics.drawPath(commands, data1);
            graphics.endFill();
        }
    }
}
```

这样会得到与第一个版本完全相同的结果。

来看最后这个版本，如果你希望覆盖默认行为而填充整个区域，就不能简单地在中间增加一个endFill和一个新的beginFill，因为这里只发生了一个drawPath调用。这里就要引入drawPath

TopSage.com

的第3个可选参数：winding。这个winding参数是一个字符串类型，可以是“evenOdd”或“nonZero”。这两个字符串是flash.display.GraphicsPathWinding类的静态常量：GraphicsPathWinding.EVEN_ODD和GraphicsPathWinding.NON_ZERO。

可以在Flash CS4帮助文件中对此做更深入的研究，了解它们为何得此名称以及基本原理。不过简单地讲，EVEN_ODD是默认行为，对于采用相同填充绘制的形状，其相交区域将不填充。

只需把最后一个例子中的drawPath调用做如下修改就可以看出这一点，可以看到它仍有同样的表现：

```
graphics.drawPath(commands, data1, GraphicsPathWinding.EVEN_ODD);
```

利用这个可选的环绕选项，可以对形状如何相交提供更多控制。继续做以下尝试：

```
graphics.drawPath(commands, data1, GraphicsPathWinding.NON_ZERO);
```

现在形状会完全填充。不过不能简单地下这样的结论：EVEN_ODD不会填充相交区域，而NON_ZERO会填充相交区域。还记得正环绕/负环绕吗？这正是我们讨论这个问题的原因。指定NON_ZERO环绕时，Flash会查看两个重叠路径（或路径重叠部分）的环绕方向，确定对交叉区域如何处理，并应用以下规则：

- 如果两个路径的环绕方向是同一个方向（正或负），交叉区域将被填充；
- 如果两个路径的环绕方向不同（一个为正，另一个为负），交叉区域则不填充。

在前面的例子中，两个矩形都以顺时针方向绘制，如图8-6所示。

通过切换其中一个矩形中的两个点可以看到环绕方向的具体作用，如下：

```
data1.push(150, 100);
data1.push(200, 100);
data1.push(200, 250);
data1.push(150, 250);
data1.push(150, 100);

data1.push(100, 150);
data1.push(100, 200);
data1.push(250, 200);
data1.push(250, 150);
data1.push(100, 150);
```

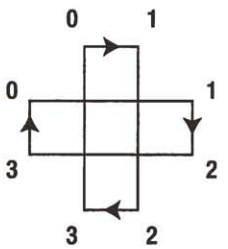


图8-6 两个顺时针路径

通过切换第二个矩形的第2个点和第4个点，将其环绕方向改为逆时针方向。测试后可以看到中间区域不再填充。如果在第一个矩形中切换同样的两个点，那么这两个矩形都将是逆时针环绕。不过这样一来既然环绕方向相同，整个形状将完全填充。

好了，以上是对drawPath的介绍，不过在介绍更强大的功能时还会谈到路径。接下来，我们来讨论三角形。

www.TopSage.com

8.2　三角形

三角形是计算机图形中非常重要的概念。两个点构成一条线，定义一个平面则必须有3个点。因此，三个点是完成任何颜色或位图填充所需的最低条件。另外，三角形通常是3D建模系统的基本构建模块。即使利用曲面（spline）或基本体（primitive）建模，渲染过程中模型通常也会在某处分解为三角形。实际上，Flash 10中新增加的三角形绘图特性主要针对3D效果，不过也可以在更多方面使用。

下面先从最简单的情况谈起。我们将绘制一个三角形。一个三角形由3个点构成，对不对？所以首先必须创建3个x和y坐标。假设第一个点为（100,100），第二个点为（200,100），第三个点为（150,200）。

因此，要建立一个向量，其中包含前面所述的绘制一个三角形的顶点，可以有以下代码：

```
var vertices:Vector.<Number> = new Vector.<Number>();
vertices.push(100, 100);
vertices.push(200, 100),
vertices.push(150, 200);
```

现在，要绘制这个三角形，只需把这个向量传入一个Graphics对象的drawTriangles方法，如下：

```
graphics.drawTriangles(vertices);
```

同样地，还需要设置一个线型或某种填充以便所绘制的图形真正可见。下面这个类Triangles1展示了如何绘制这样一个三角形。可以在Triangles1.as文件中看到这个类：

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

    [SWF(backgroundColor=0xffffff)]
    public class Triangles1 extends Sprite
    {
        public function Triangles1()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var vertices:Vector.<Number> = new Vector.<Number>();
            vertices.push(100, 100);
            vertices.push(200, 100);
            vertices.push(150, 200);

            graphics.lineStyle(0);
            graphics.drawTriangles(vertices);
        }
    }
}
```

www.TopSage.com

结果如图8-7所示。

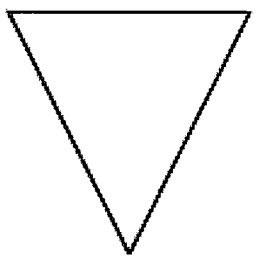


图8-7 用drawTriangles方法绘制的一个三角形

可以看到，这比使用drawPath要容易一些。不需要moveTo或lineTo。这里只需给出3个点，它就知道该做些什么。

下面来画两个三角形。为此，我们需要6个点，这说明列表中要有12个数。下一个类Triangles2（可以下载Triangles2.as得到）展示了如何绘制这两个三角形：

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

    [SWF(backgroundColor=0xffffff)]
    public class Triangles2 extends Sprite
    {
        public function Triangles2()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var vertices:Vector.<Number> = new Vector.<Number>();
            vertices.push(100, 100);
            vertices.push(200, 100);
            vertices.push(150, 200);

            vertices.push(250, 150);
            vertices.push(350, 175);
            vertices.push(200, 300);

            graphics.lineStyle(0);
            graphics.drawTriangles(vertices);
        }
    }
}
```

可以看到这里只增加了3个顶点，就会画出如图8-8所示的三角形。

如果采用这种方式，必然需要为drawTriangles提供一个数字列表，其元素总数为6的倍数。例如，试着从列表中删除最后一个顶点：

```
vertices.push(100, 100);
vertices.push(200, 100);
vertices.push(150, 200);
vertices.push(250, 150);
vertices.push(350, 175);
```

由于只有5个顶点，Flash不知道该如何构造两个三角形，因此会报错指出drawTriangles的一个参数非法。

www.TopSage.com

大家网 TopSage.com

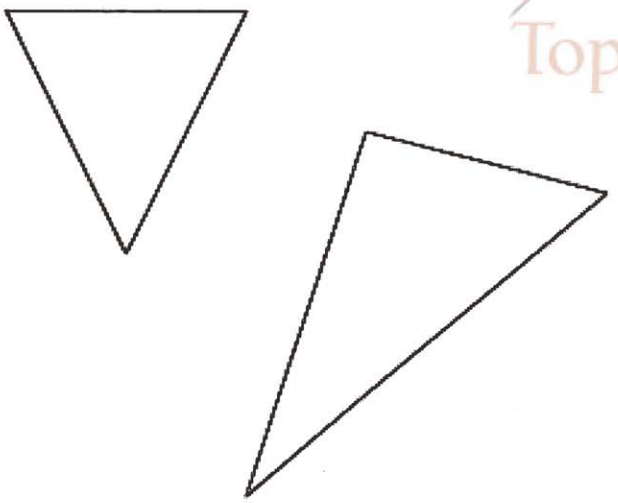


图8-8　用drawTriangles方法绘制的一对三角形

这就让我们产生了两个想法：一方面绘制一大堆单个的三角形用途不大，另一方面通过重用两个顶点则只用4个顶点就可以绘制两个三角形。幸运的是，尽管drawTriangles的默认行为是利用3个顶点为一组绘制一个单独的三角形，不过这个方法还有第二个可选的参数indices，利用这个参数你可以控制使用哪些顶点来绘制一些三角形。

indices参数也是一个向量（不过这是一个int向量）。indices向量中的每个索引指定vertices向量中的一个顶点。再来看第一个示例，其中绘制一个三角形采用了以下做法：

```
var vertices:Vector.<Number> = new Vector.<Number>();
vertices.push(100, 100);
vertices.push(200, 100);
vertices.push(150, 200);
graphics.lineStyle(0);
graphics.drawTriangles(vertices);
```

使用indices向量时，利用以下代码可以实现同样的工作：

```
var vertices:Vector.<Number> = new Vector.<Number>();
vertices.push(100, 100);
vertices.push(200, 100);
vertices.push(150, 200);
var indices:Vector.<int> = new Vector.<int>();
indices.push(0, 1, 2);
graphics.lineStyle(0);
graphics.drawTriangles(vertices, indices);
```

这里告诉Flash使用顶点0（100,100）作为三角形的第一个顶角，顶点1（200,100）作为第二个顶角，顶点3（150,200）作为最后一个顶角。这与只使用vertices向量本身时并没有不同，不过这里展示了最简单的做法。现在来尝试只用4个点画两个三角形。下面这个例子见Triangles3.as文件：

www.TopSage.com

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

    [SWF(backgroundColor=0xffffff)]
    public class Triangles3 extends Sprite
    {
        public function Triangles3()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var vertices:Vector.<Number> = new Vector.<Number>();
            vertices.push(100, 100);
            vertices.push(200, 100);
            vertices.push(200, 200);
            vertices.push(100, 200);

            var indices:Vector.<int> = new Vector.<int>();
            indices.push(0, 1, 2);
            indices.push(2, 3, 0);

            graphics.lineStyle(0);
            graphics.drawTriangles(vertices, indices);
        }
    }
}
```

这里使用顶点（0,1,2）绘制了一个三角形，并使用顶点（2,3,0）绘制了另一个三角形。这样就可以得到如图8-9所示的形状。这里标出了这个形状的4个顶点，以便你看出具体绘制的内容。

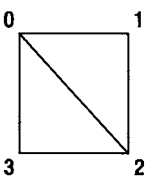


图8-9 一对相连的三角形

与drawPath类似，如果只是死板地硬编码各个顶点和命令，或者硬编码顶点和索引，你将无法了解这些新命令的强大功能。不过，绘制更复杂的形状时，可以维护一两个向量，并将其传入一个方法来完成这个形状的重绘，这种做法可能很有用。另外，三角形的真正强大之处在于用它完成位图填充，接下来将讨论这个问题。

8.2.1 位图填充与三角形

用绘图API绘制形状时，填充形状时通常有一些选择：单色填充、渐变填充和位图填充。在第9章你将看到，现在还有一种shader填充，它与Pixel Bender结合使用。

不过现在来看对位图填充的一些重大改进，使用三角形创建形状时可以利用这些改进。

首先，与使用路径或任何较早的绘图API方法一样，对于三角形你可以使用任何类型的填充：

```
graphics.beginFill(0xFF0000);
graphics.drawTriangles(vertices, indices);
graphics.endFill();
```

www.TopSage.com

这会用填充色填充各个三角形或三角形构成的整个形状。重叠的三角形会被完全渲染。这里没有环绕参数或未填充的相交区域。现在来看利用位图填充可以做什么。下一个类（可以下载BitmapTriangles.as文件得到）将展示利用一个位图填充一组三角形的基本原理：

```
package
{
    import flash.display.Bitmap;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class BitmapTriangles extends Sprite
    {
        [Embed(source="image.jpg")]
        private var ImageClass:Class;

        public function BitmapTriangles()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            var vertices:Vector.<Number> = new Vector.<Number>();
            vertices.push(100, 100);
            vertices.push(200, 100);
            vertices.push(200, 200);
            vertices.push(100, 200);
            var indices:Vector.<int> = new Vector.<int>();
            indices.push(0, 1, 2);
            indices.push(2, 3, 0);

            var bitmap:Bitmap = new ImageClass() as Bitmap;
            graphics.beginBitmapFill(bitmap.bitmapData);
            graphics.drawTriangles(vertices, indices);
            graphics.endFill();
        }
    }
}
```

这与上一个例子几乎完全相同，不过这里嵌入了一个外部JPG，创建了它的一个Bitmap实例，并使用其bitmapData属性来填充我们绘制的三角形。这里使用的图像可以从网上下载得到，不过完全可以使用任何合法的位图图像（JPG、.tif、BMP或GIF）。

现在运行这个示例时，只会看到一个黑色方块。如果使用一个不同的图像文件，可能会看到其他结果，不过你将不会看到整个图片。位图会以实际大小显示，但是我们用两个三角形绘制的方块大小仅为100×100像素，所以你只会看到位图的左上角。

可以尝试将两个顶点向外伸展，如下：

```
vertices.push(100, 100);
vertices.push(1000, 100);
vertices.push(1000, 600);
vertices.push(100, 200);
```

www.TopSage.com

TopSage.com

这样会显示位图的更多部分（参见图8-10），不过这并不是我们想要的。

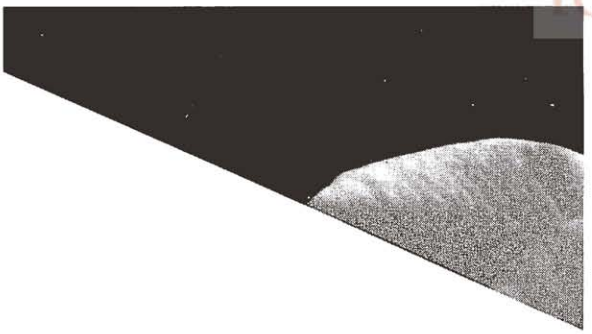


图8-10 利用三角形的位图填充

还可以对beginBitmapFill调用应用一个变换矩阵，但是这可能会相当复杂，而且不能充分利用新的三角形位图填充特性。所以下面来看这个新特性。

1. uvtData

drawTriangles的第三个可选参数是另一个Number向量，称为uvtData。uvt部分是指影响如何将位图映射到整个三角形形状的3个值。u和v值将位图*x*和*y*轴上的值映射到构成该形状的三角形的各个顶点。t值是可选的，使用三角形实现3D时它支持进一步缩放。下面来讨论u和v（t在本节后面再做讨论）。

图8-11显示了u和v值如何映射到一个位图。

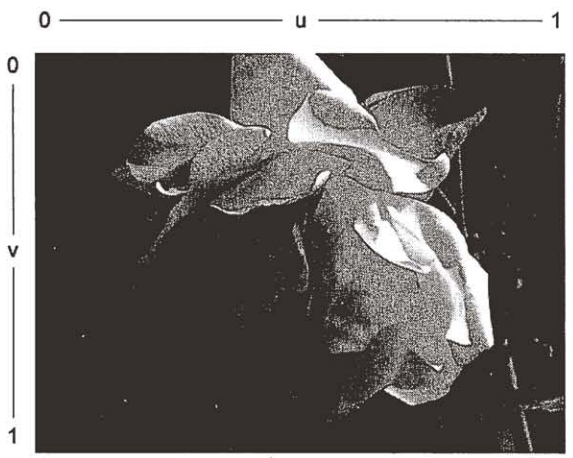


图8-11 位图的uv映射

位图的左上角对应为u和v值（0,0）。右上角对应（1,0），左下角和右下角分别对应（0,1）和（1,1）。还可以映射为小数值，不过现在只考虑4个角。

在上一个例子中，我们用4个点构成2个三角形来建立一个矩形。现在可以将这4个点映射到

www.TopSage.com

位图的4个角。我们所要做的就是建立一个Numbers向量，并为vertices向量中的各个顶点设置一个u和v值。下一个类（可以下载BitmapTrianglesUV1.as文件得到）展示了如何完成这个工作：

```
package
{
    import flash.display.Bitmap;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class BitmapTrianglesUV1 extends Sprite
    {
        [Embed(source="image.jpg")]
        private var ImageClass:Class;

        public function BitmapTrianglesUV1()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            var vertices:Vector.<Number> = new Vector.<Number>();
            vertices.push(100, 100);
            vertices.push(200, 100);
            vertices.push(200, 200);
            vertices.push(100, 200);

            var uvtData:Vector.<Number> = new Vector.<Number>();
            uvtData.push(0, 0);
            uvtData.push(1, 0);
            uvtData.push(1, 1);
            uvtData.push(0, 1);

            var indices:Vector.<int> = new Vector.<int>();
            indices.push(0, 1, 2);
            indices.push(2, 3, 0);

            var bitmap:Bitmap = new ImageClass() as Bitmap;
            graphics.beginBitmapFill(bitmap.bitmapData);
            graphics.drawTriangles(vertices, indices, uvtData);
            graphics.endFill();
        }
    }
}
```

我们建立了一个uvtData向量，并在其中放入8个值。这些值映射到vertices向量中的8个值（4个顶点），并分别对应位图的左上角、右上角、右下角和左下角。将它作为第三个参数传入drawTriangles，其结果如图8-12所示。

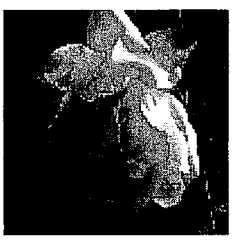


图8-12　利用三角形和uvt数据的位图填充

图像看起来好像被挤压过了，因为三角形构成了一个正方形，但是位图本身是一个矩形。现在可以移动这些顶点，实现图像变形的效果：

www.TopSage.com

```
vertices.push(150, 50);
vertices.push(1000, 100);
vertices.push(800, 600);
vertices.push(100, 200);
```

改变坐标，结果如图8-13所示。

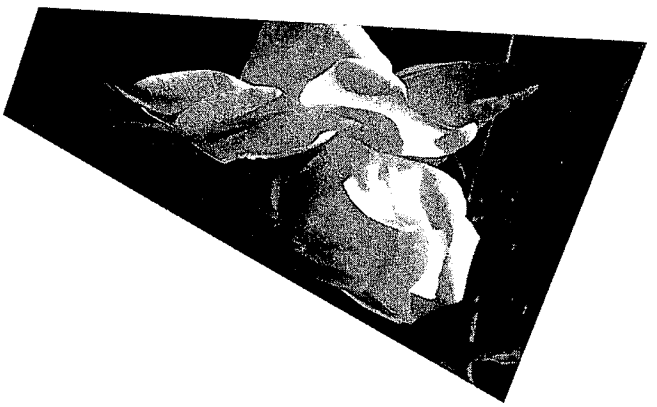


图8-13 移动点

看起来这已经有些3D效果了。不过，在进一步深入讨论这个主题之前，再对2D变形做一些讨论。

下一个例子（见BitmapTrianglesUV2.as文件）是对第一个例子的扩展，让顶点具有动态性。在此不打算过于详细地讨论这个类，不过基本说来，这个类会创建4个可拖拉的手柄。这些手柄移动时，顶点向量就会填入手柄的位置，并重绘形状：

```
package
{
    import flash.display.Bitmap;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;

    [SWF(backgroundColor=0xffffff)]
    public class BitmapTrianglesUV2 extends Sprite
    {
        [Embed(source="image.jpg")]
        private var ImageClass:Class;

        private var handle0:Sprite;
        private var handle1:Sprite;
        private var handle2:Sprite;
        private var handle3:Sprite;
        private var vertices:Vector.<Number> = new Vector.<Number>();
        private var uvtData:Vector.<Number> = new Vector.<Number>();
        private var indices:Vector.<int> = new Vector.<int>();
        private var bitmap:Bitmap;
```

www.TopSage.com

大家网 TopSage.com

```
public function BitmapTrianglesUV2()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    handle0 = makeHandle(100, 100);
    handle1 = makeHandle(200, 100);
    handle2 = makeHandle(200, 200);
    handle3 = makeHandle(100, 200);

    uvtData.push(0, 0);
    uvtData.push(1, 0);
    uvtData.push(1, 1);
    uvtData.push(0, 1);

    indices.push(0, 1, 2);
    indices.push(2, 3, 0);

    bitmap= new ImageClass() as Bitmap;
    draw();
}

private function makeHandle(xpos:Number, ypos:Number):Sprite
{
    var handle:Sprite = new Sprite();
    handle.graphics.beginFill(0);
    handle.graphics.drawCircle(0, 0, 5);
    handle.graphics.endFill();
    handle.addEventListener(MouseEvent.MOUSE DOWN, onMouseDown);
    handle.x = xpos;
    handle.y = ypos;
    addChild(handle);
    return handle;
}

private function onMouseDown(event:MouseEvent):void
{
    event.target.startDrag();
    stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
}

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

private function draw():void
```

www.TopSage.com

```
        {
            vertices[0] = handle0.x;
            vertices[1] = handle0.y;
            vertices[2] = handle1.x;
            vertices[3] = handle1.y;
            vertices[4] = handle2.x;
            vertices[5] = handle2.y;
            vertices[6] = handle3.x;
            vertices[7] = handle3.y;

            graphics.clear();
            graphics.beginBitmapFill(bitmap.bitmapData);
            graphics.drawTriangles(vertices, indices, uvtData);
            graphics.endFill();
        }
    }
}
```

可以拖动这4个点，看看图像如何变形。这种工作在较早版本的Flash中已经可以实现了，但是那时需要复杂的设置以及一些更复杂的数学运算。在这里，我们只需要一个数学运算符就可以完成同样的工作！

2. 更多三角形

到目前为止，我们只处理了4个点和2个三角形，这仅限于4个边的形状（四边形）。如果再增加更多的三角形，就会有更多的点可以移动，从而可以用来做一些有意思的事情。我们可以费力劳神地画出每个顶点，并手动地为各个三角形指定所有索引。或者也可以建立一个算法来为我们做所有这些事情。下一个类（见BitmapTriangleUV3.as文件）就要完成这个任务。这里先给出这个类，然后再做解释。其中的要点用粗体显示：

```
package
{
    import flash.display.Bitmap;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    [SWF(backgroundColor=0xffffff)]
    public class BitmapTrianglesUV3 extends Sprite
    {
        [Embed(source="image.jpg")]
        private var ImageClass:Class;

        private var vertices:Vector.<Number> = new Vector.<Number>();
        private var uvtData:Vector.<Number> = new Vector.<Number>();
        private var indices:Vector.<int> = new Vector.<int>();
        private var bitmap:Bitmap;
        private var res:Number = 100;
        private var cols:int = 5;
        private var rows:int = 4;
```

www.TopSage.com

```
        public function BitmapTrianglesUV3()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            bitmap= new ImageClass() as Bitmap;
            makeTriangles();

            graphics.beginBitmapFill(bitmap.bitmapData);
            graphics.drawTriangles(vertices, indices, uvtData);
            graphics.endFill();

            graphics.lineStyle(0);
            graphics.drawTriangles(vertices, indices);
        }

        private function makeTriangles():void
        {
            for(var i:int = 0; i < rows; i++)
            {
                for(var j:int = 0; j < cols; j++)
                {
                    vertices.push(j * res, i * res);
                    uvtData.push(j / (cols - 1), i / (rows - 1));

                    if(i < rows - 1 && j < cols - 1)
                    {
                        // first triangle
                        indices.push(i * cols + j, i * cols + j + 1, (i + 1) * cols + j);
                        // second triangle
                        indices.push(i * cols + j + 1, (i + 1) * cols + j + 1, (i + 1) * cols + j);
                    }
                }
            }
        }
    }
}
```

首先建立3个新变量：res（分辨率）确定三角形的大小，rows和cols确定会有多少行和列顶点。在makeTriangles方法中，循环处理rows和cols来建立一个顶点网格。各顶点定义为（j * res, i * res)。uvtData定义为（j / (cols - 1), i / (rows - 1)），保证在各个轴上取值范围在0.0到1.0之间。

在这个循环中，如果不在最后一行或最后一列就可以建立三角形，这里取当前点、它右边的点以及它下面的点作为第一个三角形。然后取它右边的点，右下方的点以及正下方的点建立第二个三角形（参见图8-14)。

在示例代码中，不仅使用位图填充利用三角形画出了图像，还重新利用笔划绘制了三角形，以便看出它们是如何排列的。试着改变res、rows和cols的值来看最终图像的布局和大小

www.TopSage.com

TopSage.com

如何改变。

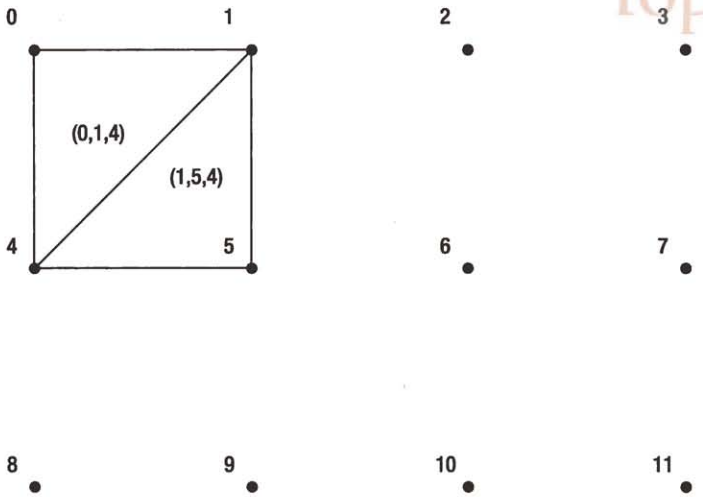


图8-14 移动点

还不错，不过由于我们只是绘制一个矩形，所以还没有真正看到三角形的威力。一旦开始移动这些顶点，就可以创建各种各样的效果，包括3D（欢迎3D隆重出场）！

8.2.2 三角形与 3D

以下是使用三角形实现3D的基本策略。

(1) 创建构成三角形的一组顶点和索引。

(2) 计算对应各个顶点的3D坐标点的位置。

(3) 使用透视法计算该3D坐标点的2D屏幕坐标，作为顶点的值。

(4) 使用drawTriangles并传入顶点和索引。

要认识到这种3D的实现与第7章中讨论的DisplayObject 3D有显著不同。实际上，基本上可以这么说：现在Flash 10有两个不同的3D引擎，drawTriangles就是其中的第二种。我们在这里使用的应用透视法的3D计算正是《Flash ActionScript 3.0 动画教程》中讨论的方法。可以回顾那本书中讲述3D的章节，或者从头开始了解基本的透视计算。透视的基本公式如下：

```
perspectiveScale = focalLength / (focalLength + zPosition)
```

因此，这里有*x*, *y*, *z*的坐标点，我们使用*z*坐标来确定一个缩放比例。然后将x和y值乘以这个比例来得到其屏幕位置，这会成为顶点列表中各个顶点的位置。

下一个类（可以下载ImageTube.as文件得到）是在上一个例子基础上建立的，不过现在把网格的顶点安排在一个管形状中。暂且不考虑位图填充，现在只利用笔划绘制三角形，以便看到绘制的内容：

www.TopSage.com

```
package
{
    import flash.display.Bitmap;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class ImageTube extends Sprite
    {
        [Embed(source="image.jpg")]
        private var ImageClass:Class;

        private var vertices:Vector.<Number> = new Vector.<Number>();
        private var indices:Vector.<int> = new Vector.<int>();
        private var uvtData:Vector.<Number> = new Vector.<Number>();
        private var bitmap:Bitmap;
        private var sprite:Sprite;
        private var res:Number = 60;
        private var cols:int = 20;
        private var rows:int = 6;
        private var centerZ:Number = 200;
        private var focalLength:Number = 250;
        private var radius:Number = 200;

        public function ImageTube()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            sprite = new Sprite();
            sprite.x = 400;
            sprite.y = 400;
            addChild(sprite);

            bitmap= new ImageClass() as Bitmap;
            makeTriangles();
            draw();
        }

        private function draw():void
        {
            sprite.graphics.lineStyle(0, 0, .5);
            sprite.graphics.drawTriangles(vertices, indices);
        }

        private function makeTriangles():void
        {
            for(var i:int = 0; i < rows; i++)
            {
                for(var j:int = 0; j < cols; j++)
                {
                    var angle:Number = Math.PI * 2 / (cols - 1) * j;
```

www.TopSage.com

```
                var xpos:Number = Math.cos(angle) * radius;
                var ypos:Number = (i - rows / 2) * res;
                var zpos:Number = Math.sin(angle) * radius;

                var scale:Number = focalLength / (focalLength + zpos + centerZ);

                vertices.push(xpos * scale, ypos * scale);
                uvtData.push(j / (cols - 1), i / (rows - 1));

                if(i < rows - 1 && j < cols - 1)
                {
                    indices.push(i * cols + j,
                                 i * cols + j + 1,
                                (i + 1) * cols + j);
                    indices.push(i * cols + j + 1,
                                (i + 1) * cols + j + 1,
                                (i + 1) * cols + j);
                }
            }
        }
    }
}
```

这里看到的第一个差别是引入了radius变量，它确定了管的半径。然后在场景中增加一个sprite，并定位在（400, 400）（因为计算3D时需要一个没影点）。可以在（400, 400）设置一个没影点，并增加到各个顶点，不过更简单的方法是仍保持没影点为（0, 0），将所有内容绘制到一个sprite中，然后移动这个sprite。

这里把具体的绘制代码移到单独的方法中（后面将介绍这样做的原因）。

最后，在makeTriangles中，我们会计算一个角（取值范围在0到2π之间）。取这个角的余弦和正弦，分别乘以半径会得到各个点的*x*和*z*坐标。取决于行数和分辨率，*y*坐标的取值范围将从一个负值到一个正值。

既然有了各个点的x、y，z值，下面来计算比例。每个顶点现在都可以定义为（xpos * scale, ypos * scale），这是应用透视法之后3D点的屏幕位置。运行这个代码时，可以看到如图8-15所示的图片。

这是所有三角形的一个线框表示——它们构成一个通过3D投影变形的管道。到目前为止一切都很好！现在只需要再增加位图填充，是不是？对于

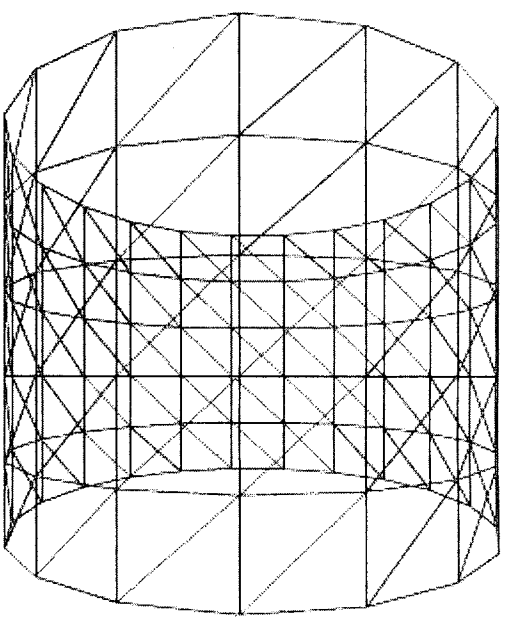


图8-15 线框管

www.TopSage.com

大家网 TopSage.com

这个问题，答案可以是“是”也可以是“不是”。还有很多工作要做。不过先来看会发生什么。修改draw方法如下：

```
private function draw():void
{
    sprite.graphics.beginBitmapFill(bitmap.bitmapData);
    sprite.graphics.drawTriangles(vertices, indices, uvtData);
    sprite.graphics.endFill();

    sprite.graphics.lineStyle(0, 0, .5);
    sprite.graphics.drawTriangles(vertices, indices);
}
```

嗯……看起来不太对劲（参见图8-16）。

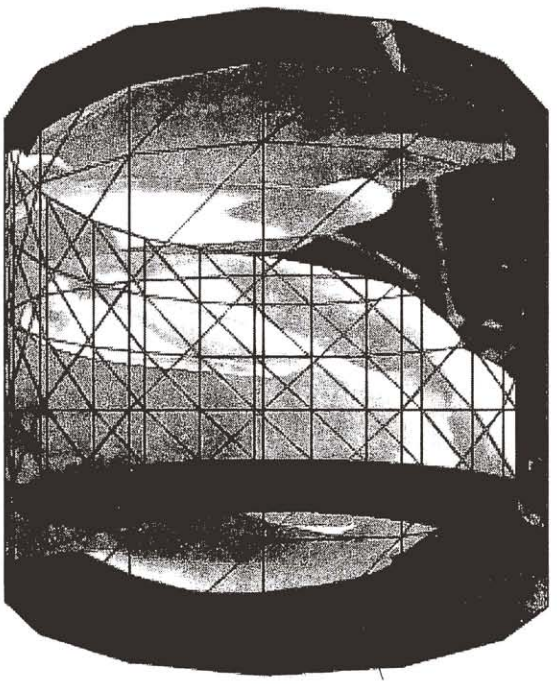


图8-16　有点问题

在这里我们绘制了所有矩形，包括后面的矩形以及前面的矩形。如果你读过第7章，你可能会举手抢着说：“深度排序！”从某种意义上讲你说的没错，但是这里有一个概念与绘制三角形时稍有不同：这就是**背面排除**（backface culling）（这个概念在《Flash ActionScript 3.0 动画教程》中也讨论过，而且都是手动完成的）。基本概念如下：绘制以顺时针方向构成的所有三角形，而不会绘制以逆时针方向形成的所有三角形。反之亦然——这取决于你如何建立三角形。不论采用哪一种做法，如果能保持一致，面向你的三角形会沿一个方向（顺时针或逆时针），而背对你的三角形会是另一个方向。这些背对的三角形我们称之为**背面**（back face），不需要绘制。

www.TopSage.com

幸运的是，现在这个工作已经内置在Flash中。调用drawTriangles时，第4个可选的参数是culling，这是一个字符串，可以等于“positive”、“negative”或“none”。这些值作为flash.display.TriangleCulling类的静态常量：

```
TriangleCulling.POSITIVE
TriangleCulling.NEGATIVE
TriangleCulling.NONE
```

可以阅读Flash帮助文件来了解这个类，其中对于背面排除如何工作提供了更专业的描述，不过基本上可以用这样一种简单的方法来考虑：**正排除**（positive culling）只绘制逆时针方向的三角形，而**负排除**（negative culling）只绘制顺时针三角形。将culling设置为NONE时，将绘制所有三角形。由于我们建立的三角形都是顺时针方向，因此需要使用负排除，如下：

```
private function draw():void
{
    sprite.graphics.beginBitmapFill(bitmap.bitmapData);
    sprite.graphics.drawTriangles(vertices, indices, uvtData, TriangleCulling.NEGATIVE);
    sprite.graphics.endFill();

    sprite.graphics.lineStyle(0, 0, .5);
    sprite.graphics.drawTriangles(vertices, indices, uvtData, TriangleCulling.NEGATIVE);
}
```

现在运行一下这个文件，可以看到如图8-17所示的结果。

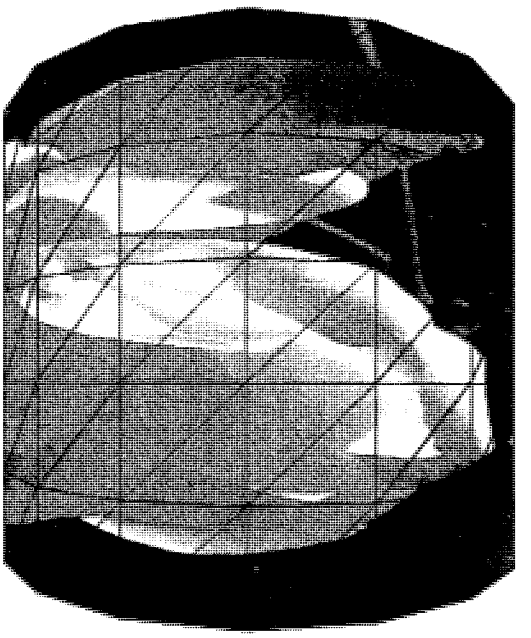


图8-17 利用背面排除解决问题

这看上去就像是一个图像包裹着一个管子。可以删除第二个drawTriangles调用，看看只有

www.TopSage.com

图像而没有三角形轮廓的结果。

1. uvt中的t

现在来看uvt数据的另一部分：t，这一部分专门用于调整3D绘图中的位图比例。在前一个例子中，根据各个点的z坐标我们计算了一个缩放比例值。使用这个比例值来调整x和y坐标，将它们投影到一个屏幕位置，并创建绘制各个三角形时使用的顶点。可以看到，这个比例值也是各个顶点t值所用的值。它会影响任意两个点之间的像素如何缩放和分隔。可以想象长长的一列均匀分隔的栏杆延伸到很远，在3D透视中查看时，远处的栏杆相互靠得更为紧密，而近处的栏杆分隔较宽。t值针对于像素而不是栏杆。如果没有t，两个点之间的像素会均匀分隔，这样得到的图像看上去会有些变形。

t值直接增加到uvtData向量中的u和v值之后，如下所示：

```
var scale:Number = focalLength / (focalLength + zpos + centerZ);
vertices.push(xpos * scale, ypos * scale);
uvtData.push(j / (cols - 1), i / (rows - 1));
uvtData.push(scale);
```

现在你可能认为这会导致混乱，因为之前uvtData中的每两个值指示vertices向量中的两个值。现在对应每个顶点却有3个值。幸运的是，drawTriangles能聪明地分辨出来。如果uvtData与vertices长度相同，它就认为其中只包含u和v。如果uvtData的长度是vertices长度的1.5倍，就会重新映射，将每一组3个值解释为u, v和t。

下面增加这一行代码测试一下。你可能看不出这个例子有多大的差别，不过如果抓屏并逐一比较，就会看到位图在管子上伸展时存在微小变化。尽管在这个例子中可能注意不到，但是最好养成使用比例的好习惯，因为这可能会对你的3D形状看上去是否真实带来显著差别（特别是透视效果更突出时）。

2. 旋转管子

因为这本书的书名里包含一个词“动画”，所以下面让这个管子以某种方式动起来。至少可以让它旋转起来。

我们将建立一个不断改变的偏移值。将所有顶点和uvtData的创建工作移到draw方法中，并在每一帧上完成更新，为所计算的角增加偏移值。这会导致顶点绕着管子旋转，并随之拖动位图。给人的感觉就是一个旋转着的管子：

```
package
{
    import flash.display.Bitmap;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.display.TriangleCulling;
    import flash.events.Event;

    [SWF(backgroundColor=0xffffff)]
    public class ImageTube extends Sprite
    {
        [Embed(source="image.jpg")]
```

www.TopSage.com

TopSage.com

```
private var ImageClass:Class;

private var vertices:Vector.<Number> = new Vector.<Number>();
private var indices:Vector.<int> = new Vector.<int>();
private var uvtData:Vector.<Number> = new Vector.<Number>();
private var bitmap:Bitmap;
private var sprite:Sprite;
private var res:Number = 60;
private var cols:int = 20;
private var rows:int = 6;
private var centerZ:Number = 200;
private var focalLength:Number = 250;
private var radius:Number = 200;
private var offset:Number = 0;
public function ImageTube()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    sprite = new Sprite();
    sprite.x = 400;
    sprite.y = 400;
    addChild(sprite);

    bitmap= new ImageClass() as Bitmap;
    makeTriangles();
    draw();
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    draw();
}

private function draw():void
{
    offset += .05;
    // clear out the data in vertices and uvtData
    vertices.length = 0;
    uvtData.length = 0;

    for(var i:int = 0; i < rows; i++)
    {
        for(var j:int = 0; j < cols; j++)
        {
            var angle:Number = Math.PI * 2 / (cols - 1) * j  + offset;

            var xpos:Number = Math.cos(angle) * radius;
            var ypos:Number = (i - rows / 2) * res;
            var zpos:Number = Math.sin(angle) * radius;

            var scale:Number = focalLength / (focalLength + zpos + centerZ);

            vertices.push(xpos * scale, ypos * scale);
```


www.TopSage.com

```
                uvtData.push(j / (cols - 1), i / (rows - 1));
                uvtData.push(scale);
            }
        }
        sprite.graphics.clear();
        sprite.graphics.beginBitmapFill(bitmap.bitmapData);
        sprite.graphics.drawTriangles(vertices, indices, uvtData, TriangleCulling.NEGATIVE);
        sprite.graphics.endFill();

//          sprite.graphics.lineStyle(0, 0, .5);
//          sprite.graphics.drawTriangles(vertices, indices, uvtData, TriangleCulling.NEGATIVE);
//
    }

    private function makeTriangles():void
    {
        for(var i:int = 0; i < rows; i++)
        {
            for(var j:int = 0; j < cols; j++)
            {
                if(i < rows - 1 && j < cols - 1)
                {
                    indices.push(i * cols + j, i * cols + j + 1, (i + 1) * cols + j);

                    indices.push(i * cols + j + 1, (i + 1) *cols + j + 1, (i + 1) * cols + j);
                }
            }
        }
    }
}
```

需要指出，makeTriangles方法只包含创建indices的代码。索引只需创建一次，而且不再改变。不过，vertices和uvtData必须每一帧都更新，所以每一帧都要在draw方法中重新计算。

既然可以建立管子，把它变换成球也就不需要太多工作。下面就来做这个尝试。

3. 建立一个3D球体

关于利用三角形实现3D，最后要讨论的是创建一个旋转的3D球体。这里我用了一张世界地图（非常适用于这个例子），不过也完全可以使用你喜欢的任何图像来映射到球体。这里的大部分代码都与管子示例中的代码相同。只需调整顶点值，使它们在顶部和底部汇聚，从而在二者之间形成一个圆。这需要完成一些三角运算，还要做大量反复试验，不过这里将给出相应的代码（见ImageSphere.as文件）：

```
package
{
    import flash.display.Bitmap;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.display.TriangleCulling;
    import flash.events.Event;
```

www.TopSage.com

```
[SWF(backgroundColor=0x000000, width=800, height=800)]
public class ImageSphere extends Sprite
{
    [Embed(source="map.jpg")]
    private var ImageClass:Class;

    private var vertices:Vector.<Number> = new Vector.<Number>();
    private var indices:Vector.<int> = new Vector.<int>();
    private var uvtData:Vector.<Number> = new Vector.<Number>();
    private var bitmap:Bitmap;
    private var sprite:Sprite;
    private var centerZ:int = 500;
    private var cols:int = 20;
    private var rows:int = 20;
    private var focalLength:Number = 1000;
    private var radius:Number = 400;
    private var offset:Number = 0;

    public function ImageSphere()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        sprite = new Sprite();
        sprite.x = 400;
        sprite.y = 400;
        addChild(sprite);

        bitmap= new ImageClass() as Bitmap;
        makeTriangles();
        draw();
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        draw();
    }

    private function draw():void
    {
        offset -= .02;
        vertices.length = 0;
        uvtData.length = 0;
        for(var i:int = 0; i < rows; i++)
        {
            for(var j:int = 0; j < cols; j++)
            {
                var angle:Number = Math.PI * 2 / (cols - 1) * j;
                var angle2:Number = Math.PI * i / (rows - 1) - Math.PI / 2;

                var xpos:Number = Math.cos(angle + offset) * radius * Math.cos(angle2);
                var ypos:Number = Math.sin(angle2) * radius;
                var zpos:Number = Math.sin(angle + offset) * radius * Math.cos(angle2);
```

www.TopSage.com

大家网 TopSage.com

```
                var scale:Number = focalLength / (focalLength + zpos + centerZ);

                vertices.push(xpos * scale, ypos * scale);
                uvtData.push(j / (cols - 1), i / (rows - 1));
                uvtData.push(scale);
            }
        }

        sprite.graphics.clear();
        sprite.graphics.beginBitmapFill(bitmap.bitmapData);
        sprite.graphics.drawTriangles(vertices, indices, uvtData, TriangleCulling.NEGATIVE);
        sprite.graphics.endFill();

//          sprite.graphics.lineStyle(0, 0, .4);
//          sprite.graphics.drawTriangles(vertices, indices, uvtData, TriangleCulling.NEGATIVE);
//
    }

    private function makeTriangles():void
    {
        for(var i:int = 0; i < rows; i++)
        {
            for(var j:int = 0; j < cols; j++)
            {
                if(i < rows - 1 && j < cols - 1)
                {
                    indices.push(i * cols + j,  i * cols + j + 1, (i + 1) * cols + j);
                    indices.push(i * cols + j + 1, (i + 1) * cols + j + 1, (i + 1) * cols + j);
                }
            }
        }
    }
}
```

可以看到，这与管子示例没有太大差别：只是位图不同，另外在每一帧上创建顶点时使用了更复杂的公式。angle2变量取值范围是–Math.PI到+Math.PI。取其正弦就会将各顶点的y值定位于球体的“北极”到“南极”之间。这个角的余弦将用于平滑地改变绕y轴的圆的半径，将管子变为一个球体。可以看到结果，如图8-18所示。

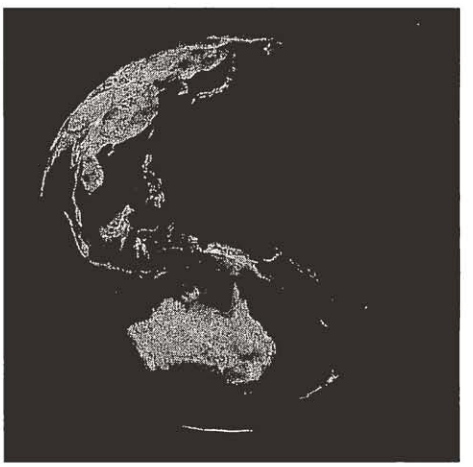


图8-18　一个旋转的球体

对于这个例子还有很大的扩展空间。我们可以对各种各样的物体建模，这里只展示了最基本的情况。当然，它没有PaperVision3D强大（这个类中已经包含有这样一些功能）。不过，我相信这个新API会在各种基于Flash的3D包中找到用武之地，因为它们现在是Player的一部分，而且可以在低于已编译ActionScript

www.TopSage.com

的下一级进行操作。

现在先不讨论3D，本章的最后将讨论图形数据。

8.3 图形数据

到目前为止，我们已经讨论了绘制路径和绘制三角形，这些都是绘图API中功能很强大的新增方面。不论你是否相信，它们都还只能算是构建模块，是绘图API中一个更强大特性的一部分。这个强大的新特性主要通过flash.display.Graphics的一个新方法drawGraphicsData来访问。

应该还记得，可以使用drawPath或drawTriangles画线或填充，不过必须使用原来的lineStyle和beginFill方法指定线型和填充类型。如果使用drawGraphicsData，则可以指定路径、三角形、笔划（线型）和任意类型的填充。

做法很简单：向drawGraphicsData方法传入一个向量。这个向量包含一些IGraphicsData对象，IGraphicsData是一个接口。这说明可以在这个向量中放入多个不同类型的对象，只要它们都实现了IGraphicsData接口。下面的类都实现了这个IGraphicsData接口：

- GraphicsStroke
- GraphicsSolidFill
- GraphicsGradientFill
- GraphicsBitmapFill
- GraphicsShaderFill
- GraphicsEndFill
- GraphicsPath
- GraphicsTrianglePath

这些类都在flash.display包中。这里不打算逐一详细地分析每一个类，但是我们会给出足够的相关信息，以便你对这些类的一般用法有所认识。下面先从GraphicsStroke和GraphicsPath开始介绍，利用这两个类至少可以画一条或多条线。以下是GraphicsStroke的构造函数：

```
GraphicsStroke(thickness:Number = NaN, pixelHinting:Boolean = false,
               scaleMode:String = "normal", caps:String = "none",
               joints:String = "round", miterLimit:Number = 3.0,
               fill:IGraphicsFill = null)
```

能够在Graphics.lineStyle方法中指定的所有内容都可以通过构造函数在GraphicsStroke对象中指定。构造函数的所有这些参数也是GraphicsStroke类的公共属性，所以只需用默认参数创建一个新的GraphicsStroke对象，再为它设置属性，如下：

```
var stroke:GraphicsStroke = new GraphicsStroke();
stroke.thickness = 1;
```

这里可能让人有些困惑的是填充属性。通常你会考虑用线画一个形状，而填充是填补这些线之间空间的颜色、渐变或位图。不过，对于GraphicsStroke对象，填充是指笔划本身的颜色、alpha透明度和渐变，等等。填充由另一个类型为IGraphicsFill的对象定义。IGraphicsFill同样是一

www.TopSage.com

个接口。以下类型的图形填充都实现了IGraphicsFill接口：

- GraphicsSolidFill
- GraphicsGradientFill
- GraphicsBitmapFill
- GraphicsShaderFill

目前只使用一个单色填充。GraphicsSolidFill类构造函数如下：

```
GraphicsSolidFill(color:uint = 0, alpha:Number = 1.0)
```

这些参数也是类的公共属性，以便在已有的填充对象上改变这些属性。根据上述内容，现在可以如下创建一个笔划对象，这里创建一个5像素宽的红色笔划：

```
var stroke:GraphicsStroke = new GraphicsStroke(5);
stroke.fill = new GraphicsSolidFill(0xff0000);
```

现在可以创建一个GraphicsPath对象。构造函数如下：

```
GraphicsPath(commands:Vector.<int> = null, data:Vector.<Number> = null,
             winding:String = "evenOdd")
```

我们已经了解了有关路径的内容，所以下面可以很容易地加入路径：

```
var commands:Vector.<int> = new Vector.<int>();
commands.push(GraphicsPathCommand.MOVE_TO);
commands.push(GraphicsPathCommand.LINE_TO);

var data:Vector.<Number> = new Vector.<Number>();
data.push(100, 100);
data.push(200, 200);

var path:GraphicsPath = new GraphicsPath(commands, data);
```

这里有一个commands向量和一个data向量。我们创建了一个GraphicsPath对象并把commands向量和data向量传入这个对象。

好了，现在我们有了一个笔划和一个路径。只需将它们放入到一个IGraphicsData向量，并把该向量传入drawGraphicsData，这样我们就有了一条线。以下是最后的测试，可以下载GraphicsDataDemo1.as文件得到：

```
package
{
    import flash.display.GraphicsPath;
    import flash.display.GraphicsPathCommand;
    import flash.display.GraphicsSolidFill;
    import flash.display.GraphicsStroke;
    import flash.display.IGraphicsData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class GraphicsDataDemo1 extends Sprite
```

www.TopSage.com

TopSage.com

```
    {
        public function GraphicsDataDemo1()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var graphicsData:Vector.<IGraphicsData> = new Vector.<IGraphicsData>();

            var stroke:GraphicsStroke = new GraphicsStroke(5);
            stroke.fill = new GraphicsSolidFill(0xff0000);
            var commands:Vector.<int> = new Vector.<int>();
            commands.push(GraphicsPathCommand.MOVE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);

            var data:Vector.<Number> = new Vector.<Number>();
            data.push(100, 100);
            data.push(200, 200);

            var path:GraphicsPath = new GraphicsPath(commands, data);

            graphicsData.push(stroke);
            graphicsData.push(path);
            graphics.drawGraphicsData(graphicsData);
        }
    }
}
```

这里只把之前没有见过的重要代码用粗体显示。运行这个代码，场景上会出现一条红线。真让人吃惊！采用原绘图API时只需3行代码就可以完成的工作现在居然要用13行代码才能做到！

你肯定知道我要说什么：这个特性适用于更复杂的绘图工作。之前我曾提到过命令模式，在命令模式中，动作及其参数都封装在一个对象中。这里正是如此。这里没有编写一长串绘制方法，而只是创建了一个绘制命令向量。由于各个命令都是对象，因此可以向列表增加更多命令、删除命令、重排命令以及修改命令，而且也可以在任何时刻绘制整个列表。下面来看这个特性的强大作用。

在本章最后的例子中，我们要做的工作与之前创建的简单草图程序很相似，不过现在要让它在一定程度上支持历史功能。其策略如下：

(1) 每次按下鼠标按钮时，选择一个随机的笔划（粗细和颜色），并把它增加到一个图形数据缓冲区；

(2) 鼠标移动时，与前面一样，将LINE_TO和数据点增加到一个路径；

(3) 释放鼠标时，根据刚才所绘制路径的命令和数据创建一个GraphicsPath对象。将这个GraphicsPath对象压入缓冲区；

(4) 在draw方法中，将图形数据缓冲区复制到另一个GraphicsData向量，并使用drawGraphicsData绘制。

使用缓冲区的原因是为了支持历史功能。按下键盘上的左右箭头键会让索引变量上调或下调。draw方法实际上是将命令从缓冲区复制到索引变量。因此，如果已经画了5条线，然后按下

www.TopSage.com

左键，索引变量会减1。只有前4个命令会从缓冲区复制到commands向量。相应地只会画出4条线。按下右键，索引会增1。现在所有5条线都会画出。以下是具体的代码（也可以在HistoryDraw.as文件中看到）：

```
package
{
    import flash.display.GraphicsPath;
    import flash.display.GraphicsPathCommand;
    import flash.display.GraphicsSolidFill;
    import flash.display.GraphicsStroke;
    import flash.display.IGraphicsData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.KeyboardEvent;
    import flash.events.MouseEvent;
    import flash.ui.Keyboard;

    [SWF(backgroundColor=0xffffff)]
    public class HistoryDraw extends Sprite
    {
        private var graphicsData:Vector.<IGraphicsData>;
        private var graphicsDataBuffer:Vector.<IGraphicsData>;
        private var commands:Vector.<int>;
        private var data:Vector.<Number>;
        private var index:int = 0;

        public function HistoryDraw()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            graphicsData = new Vector.<IGraphicsData>();
            graphicsDataBuffer = new Vector.<IGraphicsData>();

            stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
            stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
        }

        private function onMouseDown(event:MouseEvent):void
        {
            // create a random stroke, add it to buffer
            var stroke:GraphicsStroke = new GraphicsStroke();
            stroke.thickness = Math.random() * 10;
            stroke.fill = new GraphicsSolidFill(Math.random() *
                                                0xffffff);
            graphicsDataBuffer.push(stroke);

            // increment index
            // and make this the last command in the buffer
            index++;
            graphicsDataBuffer.length = index;

            // push a moveTo onto the commands
```

www.TopSage.com

```
        commands = new Vector.<int>();
        commands.push(GraphicsPathCommand.MOVE_TO);
        // move to this point
        data = new Vector.<Number>();
        data.push(mouseX, mouseY);

        // draw a gray line for now
        graphics.lineStyle(0, 0, .5);
        graphics.moveTo(mouseX, mouseY);

        stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
        stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    }

    private function onMouseMove(event:MouseEvent):void
    {
        // push a lineto onto the commands,
        // and current point to data
        commands.push(GraphicsPathCommand.LINE_TO);
        data.push(mouseX, mouseY);

        // draw temporary line
        graphics.lineTo(mouseX, mouseY);
    }

    private function onMouseUp(event:MouseEvent):void
    {
        // this line is done. push path onto buffer
        graphicsDataBuffer.push(new GraphicsPath(commands, data));

        // increment index and make this last command in buffer
        index++;
        graphicsDataBuffer.length = index;

        stage.removeEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
        stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
        draw();
    }

    private function onKeyUp(event:KeyboardEvent):void
    {
        // go back two commands (path and stroke)
        if(event.keyCode == Keyboard.LEFT)
        {
            index -= 2;
        }
        // go forward two commands (stroke and path)
        else if(event.keyCode == Keyboard.RIGHT)
        {
            index += 2;
        }
            // limit index to sensible range
            index = Math.max(0, index);
            index = Math.min(graphicsDataBuffer.length, index);
```

www.TopSage.com

大家网 TopSage.com

```
            draw();
        }

        private function draw():void
        {
            // clear graphics data vector
            graphicsData.length = 0;

            // copy over data from buffer up to index
            for(var i:int = 0; i < index; i++)
            {
                graphicsData[i] = graphicsDataBuffer[i];
            }
            // draw graphics data
            graphics.clear();
            graphics.drawGraphicsData(graphicsData);
        }
    }
}
```

需要指出，在onMouseUp处理程序中，我们将缓冲区长度设置为index，使得所增加的最后一个路径成为向量的最后一个元素。这在可撤销的操作中是一种常见的行为。如果画了一些线，撤销其中一些线，然后如果再绘制一些线，就无法再重做刚才撤销的工作。如果允许这种重做会导致不可预计的行为，因为这样只会在已有内容的中间插入一些内容。

大功告成——现在有了一个支持历史功能的绘制程序。当然，可以把它构建为一个更专业的程序，提供一些真正的控件来改变颜色和粗细而不是取随机值，以及其他一些改进。另外，这还只是图形数据的冰山一角。要记住，我们可以完成任何类型的填充：渐变、位图，甚至shader填充（这个内容将在第9章讨论），甚至还可以绘制三角形。如果你确实很有雄心壮志，可以使用drawGraphicsData而不是drawTriangles重新完成ImageSphere程序。

8.4　小结

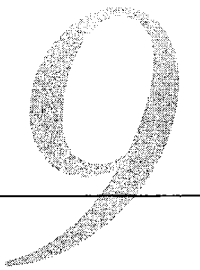
本章介绍了新的绘图API的特性，不过这里只是一点皮毛。利用新的绘图API还可以做很多工作，有关内容甚至需要整本书才能全面介绍。实际上，你可以参考Todd Yard所著的*Foundation ActionScript 3.0 Image Effects*（Apress, 2009）。

接下来，我们来讨论Flash 10新增的另一个绝妙的图形工具：Pixel Bender。

www.TopSage.com

第9章

Pixel Bender



本章的内容很有意思，因为这里不仅会介绍一个全新的可以与Flash CS4（或者编译基于Flash 10 Player的SWF时使用的任何工具）结合使用的工具，还将介绍这个工具所用的一种全新的语言。Pixel Bender Toolkit于2007年末预发布，当时代号为Hydra。它有一个简单的集成开发环境（IDE）用来编写和编译pixel shader，pixel shader就像是一些迷你程序，可以在Flash、PhotoShop和After Effects中用来创建极其强大的图像处理和动画效果。单单是这个主题就值得用一整本书来介绍，不过在本章中我会尽量提供需要了解的有关信息，希望能够让你继续深入学习并得到启发进而了解更多有关内容。

9.1 什么是 Pixel Bender

Pixel Bender Toolkit是指用来创建、编译、测试和导出pixel shader从而用于各种CS4产品（包括Flash）的一个IDE。这是一个极其简单的IDE，它包含一个概要代码编辑器、一个预览区，以及一个用来设置参数和查看错误警告的区域（参见图9-1）。

Pixel Bender是一种编程语言，可以用来在Pixel Bender Toolkit中编写pixel shader。这里首先假设你有Pixel Bender Toolkit的一个副本，已经成功地完成了安装，并且知道如何运行这个程序，正准备创建一些pixel shader。

那么，pixel shader又是什么呢？简单地说，这是一个小程序，可以计算一个像素的值。听起来如果只是如此就好像太过简单了，但这确实正是这个程序所完成的基本任务。它可以有多种形式的输入，还可以利用各种复杂的计算，不过最终的结论都是：这个像素应该有这个值。

为什么这个问题很重要，为什么人们会对此很热衷，这有很多原因。首先，Pixel Bender可以取一个位图、填充或其他可视化对象，然后对其中的每一个像素分别运行这个pixel shader。并不是逐个地运行，而是并行运行。不错，确实如此，它会同时计算一个区域中的所有像素。另外，pixel shader编写得相当优化，可以在独立于Flash Player的一个进程中运行。总而言之，与Flash中的几乎所有其他图形进程相比，pixel shader的速度快得不可思议。

Pixel Bender shader在拥有这种强大能力的同时也存在缺点，其缺点之一在于它必须使用Pixel Bender Toolkit在Flash之外创建，而且要用Pixel Bender语言（一种基于C的语言）编写，然后必须进行编译、保存，并最终加载或嵌入到Flash动画中之后才能使用。不错，为此确实需要做一些工作，而且你确实需要了解一些C语言的知识。不用担心，这对你是有好处的。

www.TopSage.com

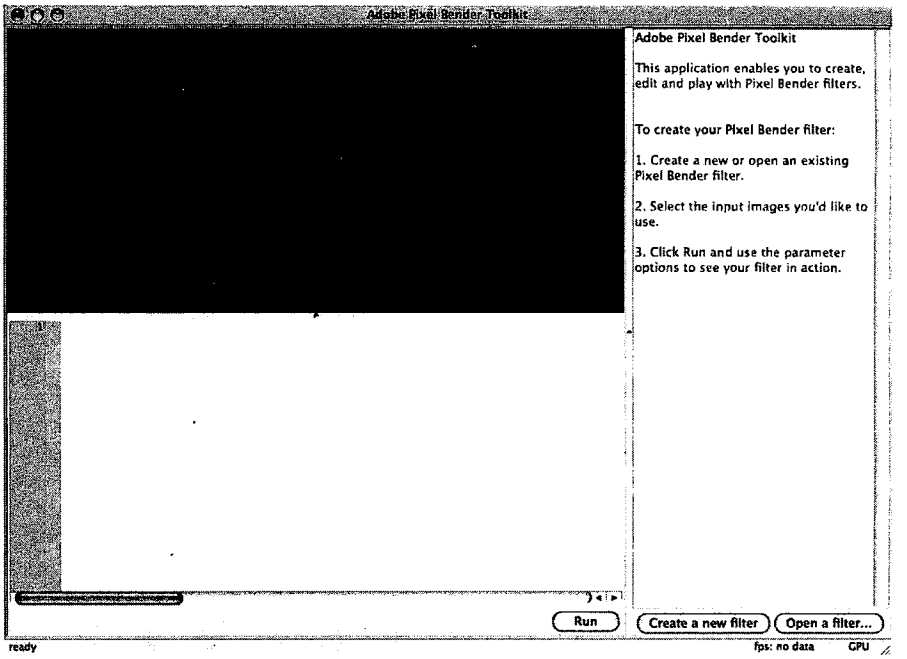


图9-1　Pixel Bender Toolkit

与你可能已经熟悉的大多数ActionScript程序相比，Pixel shader采用了一种完全不同的范型。由于shader只对一个像素感兴趣，而且同一时刻可能会运行数千次，因此它所知道的只是运行该shader的特定像素的x和y值。可以指定变量以及传入参数（包括一个或多个输入图像），不过，刚开始编写这些代码时，你可能会感觉到处都有诸多限制。不过，很快你就会对这种语言有所认识，并了解如何使用Pixel Bender提供的诸多功能。

在Flash中，可以采用4种方式使用Pixel Bender shader。

- 定制滤镜（custom filter）：类似于投影、模糊、倾斜变形等，现在还可以对任意显示对象指定一个ShaderFilter。这会绑定到用Pixel Bender编写的一个shader。
- 填充（fill）：在绘图应用程序编程接口（Application Programming Interface，API）中，总能使用单色、渐变色或位图来填充一个形状。现在还可以利用beginShaderFill方法使用Pixel Bender shader来填充形状。
- 混合模式（blend mode）：pixel shader还可以用作为一个混合模式，影响显示对象在其底层对象之上的组成方式。这要通过为显示对象的blendShader属性指定一个shader来做到。
- 通用数字处理（generic number crunching）：这个术语是指利用shader的强大功能和速度在一个数据集上快速完成数学运算。并不是使用shader来设置阴影，而是要传入一个字节数组。它会对字节数组中的数据完成指定的运算，并返回处理后的数据。你要做的只是让shader将这个字节数组看作是一个图像。这个技术超出了本书的讨论范畴，不过这确实是一个很有意思的主题，你可以自行进行研究。

www.TopSage.com

大家网 TopSage.com

9.2 编写一个 Pixel Shader

既然已经安装设置好Pixel Bender Toolkit，下面可以准备编写一个pixel shader。使用File菜单，选择New Kernel Filter或按下屏幕下方标有Create New Filter的按钮。在编辑区里会出现如下所示的代码：

```
<languageVersion : 1.0;>

kernel NewFilter
<   namespace : "Your Namespace";
    vendor : "Your Vendor";
    version : 1;
    description : "your description";
>
{
    input image4 src;
    output pixel4 dst;

    void
    evaluatePixel()
    {
        dst = sampleNearest(src,outCoord());
    }
}
```

如果希望在这个toolkit中看到shader的结果，需要加载一个图像。这也要通过File菜单来做到，另外toolkit中已经预备好很多示例图像，以防在你的计算机上或者在Internet上无法找到任何图片。选择一个图像，你可以在预览区中看到这个图像。按下Run按钮，会看到shader应用于该图像后的最终效果。

你会看到什么呢？什么都没有，图像根本没有任何变化，因为默认shader是一个标识shader。它只是查看源图像中与所计算像素位置相同的像素，并输出这个像素。因此你的输出与输入完全相同。

实际上，我认为对于介绍性讲解，这个默认示例对你来说起点有些过高，所以下面将滤镜修改如下：

```
<languageVersion : 1.0;>

kernel NewFilter
<   namespace : "Your Namespace";
    vendor : "Your Vendor";
    version : 1;
    description : "your description";
>
{
    input image4 src;
    output pixel4 dst;

    void
    evaluatePixel()
    {
```


www.TopSage.com

```
        dst = pixel4(1, 0, 0, 1);
    }
}
```

不必中止shader的运行，而且如果不想保存，那么完全无需保存（不过保存往往是一个很好的实践）。只需键入新的代码行，并在看到变化时按下Run按钮。如果现在这样做，你会看到图片变成一个实心的红色矩形（稍后会了解其原因）。

这个shader可能是最小的shader了，所以从这里起步并用它来研究编写shader的各个不同方面很合适。任何pixel shader都必须有以下元素。

- 指定文件所用语言版本的元数据。
- 内核定义，其中包含余下的必要元素（稍后将明确介绍内核是什么）。
- 必要的内核元数据，包括namespace、vendor和version（可能还包括可选的description）。
- 一个output属性，这表示所处理的像素。
- 一个名为evaluatePixel的函数，这是shader的主函数。实际上，编写Flash中使用的shader时，这也是shader中的唯一函数。所以计算像素值的所有代码都要放在这里。
- 尽管不是严格要求，但几乎总会指定一个input属性，这表示一个源位图图像。如果对绘图API中的一个填充使用一个shader，可能不会有任何输入图像。不过，在Pixel Bender Toolkit中开发shader时如果要查看结果，往往需要指定一个输入图像。

首先来看语言版本元数据，这是shader源文件的第一行，如下所示：

```
<languageVersion : 1.0;>
```

当然，将来这个工具还会有提供新功能的新版本。所以这个元数据会让编译器和Flash知道这个shader中会有什么特性。

然后是内核定义。内核是shader的基本单元。完全可以把内核看作是一个类：它是一个定义了一个pixel shader对象的单元。它可以有函数、变量和常量。函数可以相互调用，并访问内核的变量和常量，还可以调用其他内置函数和函数库。

创建用于Flash的shader时有一些限制：不允许使用其他库，也不允许调用主evaluatePixel函数之外的其他函数。不过可以调用其他内置函数，稍后就会看到。

与类相似，内核定义首先有一个关键字kernel，后面是内核名。内核体包含在大括号中：

```
kernel NewFilter
{
    // kernel code
}
```

在自动生成的shader代码中可以看到，内核体之前还有另一个元数据块：

```
kernel NewFilter
<   namespace : "Your Namespace";
    vendor : "Your Vendor";
    version : 1;
    description : "your description";
>
{
```

www.TopSage.com

```
    // kernel code
}
```

目前namespace、vendor和version元数据都是强制的。通常希望在命名空间（namespace）中填入别人不会使用的唯一标识符，对此你的域名就很合适。开发商（vendor）就是你或公司，版本（version）是这个特定shader的版本。如果以后对它进行改进，需要更新版本号以便人们知道他们在使用哪个版本。描述（description）是可选的，不过在这里增加一些描述是一个很好的想法。创建了shader的实例之后就可以在Flash中访问这个元数据，本章后面将会介绍具体方法。

在内核中，首先有两个特殊的变量。这些变量用关键字input和output指定。input变量是指传入shader的源图像，output变量是指计算出的像素值。

如果只习惯于使用ActionScript，那么定义这些变量的语法会令你感觉有些奇怪。在ActionScript中我们会用关键字var来声明变量，并用后冒号语法指定类型，如下所示：

```
var variableName:Type;
```

在C语言中，也就是Pixel Bender所基于的语言，则是通过指定变量类型和变量名来声明变量：

```
Type variableName;
```

对于这些特殊变量，要把额外的input和output关键字放在前面，不过接下来可以看到有一个名为src（源）类型为image4的变量，还有一个名为dst（目标）类型为pixel4的变量：

```
input image4 src;
output pixel4 dst;
```

接下来还会介绍另外一些不同的数据类型，不过先要了解image4表示一个4通道（红、绿、蓝和α）的位图图像，pixel4表示一个4通道像素。

前面已经提到，通常希望有一个输入图像变量，至少在toolkit中开发shader时需要这样一个输入图像。另外还必须有一个输出像素变量。

接下来是evaluatePixel函数。与变量类似，C语言中的函数如下声明：首先指定函数返回的类型，然后是函数名，接下来与ActionScript类似，将函数体放在大括号中：

```
void
evaluatePixel()
{
    dst = pixel4(1, 0, 0, 1);
}
```

在自动生成的滤镜中，其格式是将返回类型放在一行上，而函数名放在下一行。这与下面的代码没有任何区别，不过以下代码可能看上去更为自然：

```
void evaluatePixel()
{
    dst = pixel4(1, 0, 0, 1);
}
```

上面这个函数的返回类型是void，所以不需要在这里返回任何结果。不过，不论在evaluatePixel方法中做了什么，都需要将输出变量dst赋为某个值。在这个例子中，这正是我们要做的工作。这里将它直接赋为一个pixel4值。pixel4的构造函数取像素4个通道的各个分量红、绿、蓝

www.TopSage.com

和α作为参数。各个通道可以赋为一个介于0.0到1.0之间的值。实际上，可以指定一个大于1.0或小于0.0的值，不过这些值会被截断从而限制在这个范围以内。所以指定pixel4(100.0,0.0,1)与指定pixel4(1,0,0,1)没有任何区别。在这个例子中，我们将红色通道指定为1，绿色和蓝色通道指定为0，另外α通道为1。这会得到一个不透明的纯红色像素。这个shader在图像中的每一个像素上并行运行，所以各像素都会指定为红色。正因如此你会得到一个红色矩形。

祝贺你！以上已经创建了你的第一个pixel shader。如果能一直跟上我们的思路，你应该已经能完全理解所做的工作了，真的很棒。

9.3　数据类型

在继续介绍新内容之前，我们要了解Pixel Bender中提供的一些数据类型。基本类型包括bool、int和float，这对应于ActionScript中的Boolean、int和Number。bool类型保存true/false值，int保存整数值，float保存浮点数。

要记住重要的一点，指定常量float值时，必须包括小数部分。否则，这个数会被认为是一个int。例如，假设指定float num = 1;，这会给出一个编译错误；需要指定float num = 1.0;才可以。在熟悉这种差别之前，你肯定会犯无数次这种错误，所以最好能及早了解这一点。在很多数学运算中使用int代替float时以及用作函数参数时也要特别当心。重申一次，你可能会多次犯这种错误，所以要尽早发现。

接下来介绍各种向量类型，与ActionScript中的向量相似，这些向量类型都保存多个相同类型的值。例如，float2类型包含2个float类型，float3包含3个float类型，float4包含4个float类型。对应bool和int也有类似的类型。在所有这些类型中，可能最常用的就是float2，因为它可以像Point类一样使用。

然后是保存像素和图像数据的值。pixel1类型保存一个像素的一个通道。类似地，还有pixel2、pixel3和pixel4向量类型。最后一个向量类型pixel4最常用，因为它可以包含一个像素的所有4个通道——红、绿、蓝和α。同理，对于image1、image2、image3和image4类型，image4使用率最高。

在ActionScript Vector类型中，需要使用数组记法来得到列表中的单个元素，如下所示：

```
myVector[2]
```

Pixel Bender向量类型更具用户友好性。例如，要得到一个pixel4值的第一个元素（红色通道），不必写作myPixel[0];，而只需写为myPixel.r。类似地，可以通过g、b和a属性分别得到绿色通道、蓝色通道和α通道。

同样地，float2变量的值可以通过其x和y属性访问。

实际上，所有的4元素向量类型都有以下属性（不论其中包含的是什么类型）：

```
r, g, b, a
x, y, z, w
s, t, p, q
```

这些属性表示向量的4个元素。所以myPixel.x指示的内容与myPixel.r甚至myPixel.s都完全

www.TopSage.com

相同。使用像素类型时用rgba值更有意义，而使用浮点数时用xyzw值则更合适。

可以重写前例的evaluatePixel函数，如下所示：

```
void
evaluatePixel()
{
    dst.r = 1.0;
    dst.g = 0.0;
    dst.b = 0.0;
    dst.a = 1.0;
}
```

另外，可以通过结合两个或多个单元素属性来进行访问，如下：

```
myPixel.rgb    // returns a pixel3 vector
               // containing red, green and blue channels
myPixel.ra    // returns a pixel2 vector
              // containing red and alpha channels
```

这样就允许你完成如下的复杂结合：

```
evaluatePixel()
{
    pixel3 pxl = pixel3(1, 0, 1);
    dst.rgb = pxl;
    dst.a = 1.0;
}
```

这里建立了一个pixel3值，由红色通道上的1，绿色通道上的0，蓝色通道上的1组成，这是紫色。将它赋给dst.rgb。不过这会使dst没有α通道，所以要直接赋值。

还有更强大的功能，可以采用你希望的任何方式混合和匹配r、g、b和a值。例如，在下面的代码中，将pxl赋给dst时可以交换绿色通道和蓝色通道：

```
evaluatePixel()
{
    pixel3 pxl = pixel3(1, 0, 1);
    dst.rbg = pxl;
    dst.a = 1.0;
}
```

其结果是dst的rgb会取值为(1,1,0)，所以会得到一个黄色矩形。采用另一种方法可以得到同样的结果：

```
evaluatePixel()
{
    pixel3 pxl = pixel3(1, 0, 1);
    dst.rgb = pxl.grb;
    dst.a = 1.0;
}
```

这样会在pxl赋给dst的rgb之前完成红色通道和绿色通道的交换，从而得到青色。向量元素的这种交换称为**元素替换**（swizzling），尽管语法非常简单，却可以用来完成功能强大的图像处理。

9.4 获得当前像素坐标

计算一个像素时很重要的一个信息是计算像素的坐标值。可以利用内置函数outCoord来得

www.TopSage.com

到。这个函数会返回一个float2值，其中包含*x*和*y*坐标。由于每个像素都有不同的坐标，现在可以使用这个数据对各个像素设置不同颜色，而不是对每一个像素都指定同一个值。可以从本书网站（www.friendsofed.com）下载以下示例（相应文件是SineWave1.pbk）：

```
<languageVersion : 1.0;>
kernel SineWave1
<   namespace : "com.friendsofed";
    vendor : "Advanced ActionScript 3.0 Animation";
    version : 1;
    description : "draws vertical bands";
>
{
    input image4 src;
    output pixel4 dst;

    void
    evaluatePixel()
    {
        dst = pixel4(0, 0, 0, 1);
        float2 pos = outCoord();
        dst.r = sin(pos.x * .2) * .5 + .5;
    }
}
```

这里用粗体显示了重要的部分。首先，将像素指定为不透明的黑色。然后，获取当前像素坐标，作为一个名为pos的float2值。在最后一行中，将pos.x乘以0.2让曲线变化减慢，然后取该值的正弦，这会得到一个范围介于–1.0到+1.0之间的值。将这个值乘以0.5则得到一个范围从–0.5到+0.5的数，然后再加上0.5，得到范围从0.0到1.0的数，这正是一个像素通道的正确取值范围。最后，把计算结果赋给dst.r。这会使各个像素的红色通道值根据其*x*位置平滑地改变，从而得到如图9-2所示的结果。

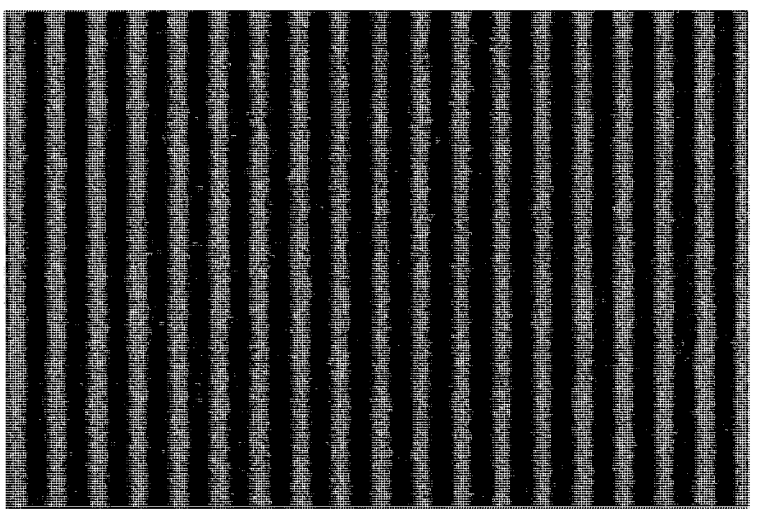


图9-2　Pixel Bender渐变条

www.TopSage.com

TopSage.com

同样你也可以对y轴做同样的处理得出另一种效果，如下面的例子所示（可以从SineWave2.pbk文件得到）：

```
<languageVersion : 1.0;>

kernel SineWave2
<   namespace : "com.friendsofed";
    vendor : "Advanced ActionScript 3.0 Animation";
    version : 1;
    description : "draws vertical and horizontal bands";
>
{
    input image4 src;
    output pixel4 dst;

    void
    evaluatePixel()
    {
        dst = pixel4(0, 0, 0, 1);
        float2 pos = outCoord();
        dst.r = sin(pos.x * .2) * .5 + .5;
        dst.b = sin(pos.y * .2) * .5 + .5;
    }
}
```

这会得到如图9-3所示的图像。

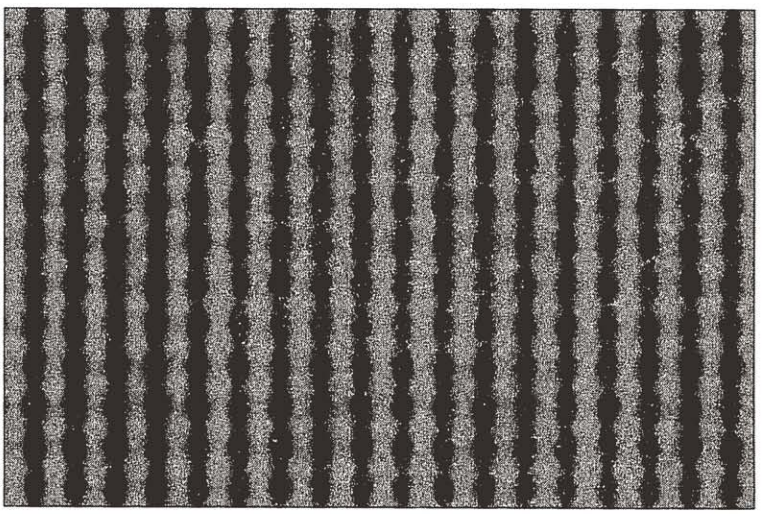


图9-3 双重条

现在，假设你希望同时改变多个通道。例如，如果想实现黑色条和白色条相间的效果，就要记红色通道、绿色通道和蓝色通道均匀地变化。可以修改代码，如下所示：

```
void
```

```
evaluatePixel()
{
    dst = pixel4(0, 0, 0, 1);
    float2 pos = outCoord();
    dst.r = sin(pos.x * .2) * .5 + .5;
    dst.g = sin(pos.x * .2) * .5 + .5;
    dst.b = sin(pos.x * .2) * .5 + .5;
}
```

这有些浪费时间，所以更好的做法如下所示：

```
void
evaluatePixel()
{
    dst = pixel4(0, 0, 0, 1);
    float2 pos = outCoord();
    pixel1 pxl = sin(pos.x * .2) * .5 + .5;
    dst.r = pxl;
    dst.g = pxl;
    dst.b = pxl;
}
```

这样一来只需完成一次计算，然后把输出赋给各个通道。不过还有一种更简单的做法，如下面的例子所示（可以从SineWave3.pbk得到）：

```
<languageVersion : 1.0;>

kernel SineWave3
<   namespace : "com.friendsofed";
    vendor : "Advanced ActionScript 3.0 Animation";
    version : 1;
    description : "draws vertical and horizontal bands";
>
{
    input image4 src;
    output pixel4 dst;

    void
    evaluatePixel()
    {
        dst = pixel4(0, 0, 0, 1);
        float2 pos = outCoord();
        dst.rgb = pixel3(sin(pos.x * .2) * .5 + .5);
        dst.a = 1.0;
    }
}
```

如果建立一个向量值，而且只传入一个值，Pixel Bender会用这个值填充向量的所有元素。因此，dst的r、g、b会得到同样的值，从而生成一些灰度垂直条，如图9-4所示。

Pixel Bender语言参考（Pixel Bender Language Reference）介绍了所有可用的内置数学函数和几何函数，这个参考手册在安装Pixel Bender Toolkit时应该已经安装。可以尝试使用其中一些函数，看看能够生成哪些有趣的图案。接下来将介绍如何在运行时改变值。

www.TopSage.com

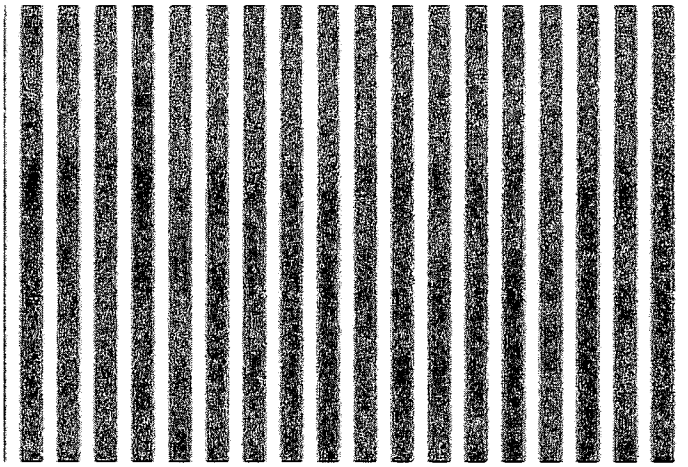


图9-4 灰度条

9.5 参数

由完全硬编码的值构成的Pixel Bender shader用途很有限。假设你在建立一个实现图像模糊效果的shader。必须为浅度模糊建立一个shader，为中度模糊建立另一个shader，而且还要为深度模糊再创建一个shader。幸运的是，Pixel Bender允许在运行时改变某些值。这些特殊的值就称为**参数**（parameter）。如果采用最简单的形式，参数的声明与所有其他变量的声明相似，只是要在最前面增加parameter关键字：

```
parameter float myParameter;
```

当你在程序中增加了一个数字参数并在Pixel Bender Toolkit中运行shader时，会看到Toolkit右上方区域出现一个滑动条。如果增加一个类型为bool的参数，则会看到一个复选框。下面就来试一试。下面的例子可以在SineWaveParam.pbk文件中得到：

```
<languageVersion : 1.0;>

kernel SineWaveParam
<   namespace : "com.friendsofed";
    vendor : "Advanced ActionScript 3.0 Animation";
    version : 1;
    description : "draws vertical bands";
>
{
    input image4 src;
    output pixel4 dst;

    parameter float mult;

    void
    evaluatePixel()
    {
        dst = pixel4(0, 0, 0, 1);
```

www.TopSage.com

大家网 TopSage.com

```
        float2 pos = outCoord();
        dst.r = sin(pos.x * mult) * .5 + .5;
    }
}
```

运行这个代码会得到如图9-5所示的滑动条。

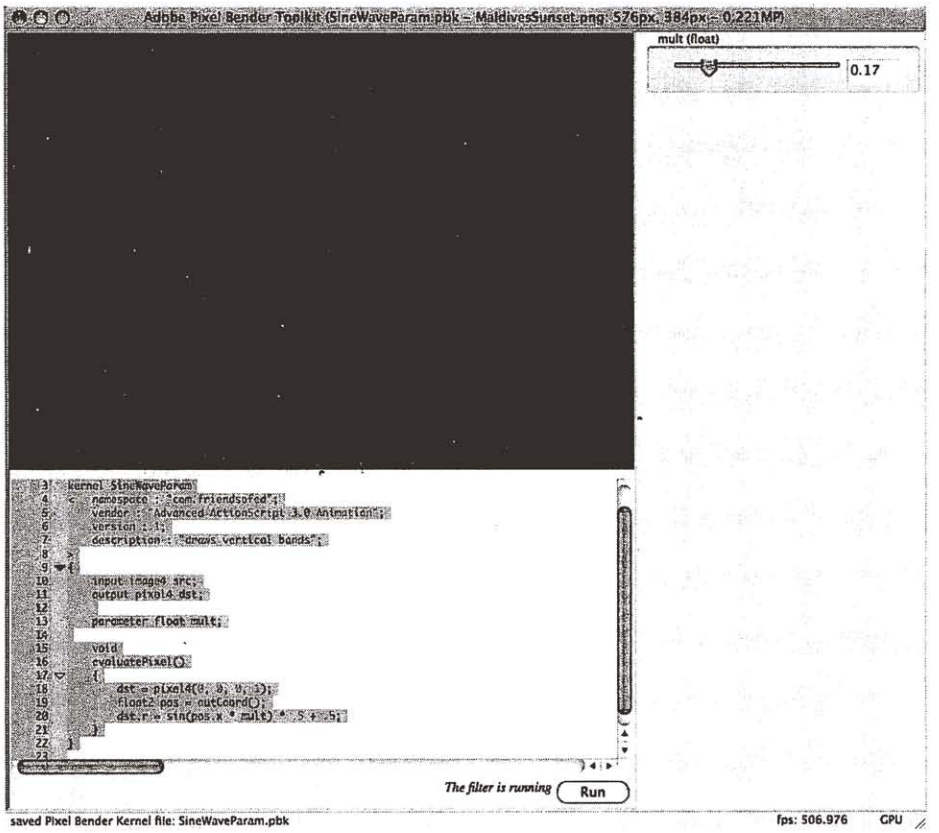


图9-5　Pixel Bender Toolkit中的参数滑动条

这里创建了一个名为mult的参数。在传入Math.sin函数创建正弦曲线之前我们将pos.x乘以这个值。现在，可以通过来回移动滑块调整垂直条的大小。

如果建立一个向量类型的参数，会得到组合在一起的多个滑动条，如下面的例子所示，这个例子可以下载ColorChooser.pbk得到：

```
<languageVersion : 1.0;>

kernel ColorChooser
<   namespace : "com.friendsofed";
    vendor : "Advanced ActionScript 3.0 Animation";
    version : 1;
    description : "shows a pixel4 parameter in action";
>
```

www.TopSage.com

```
{
    input image4 src;
    output pixel4 dst;

    parameter pixel4 color;

    void
    evaluatePixel()
    {
        dst = color;
    }
}
```

因为参数color是一个pixel4值，可以将得到4个滑动条，如图9-6所示。

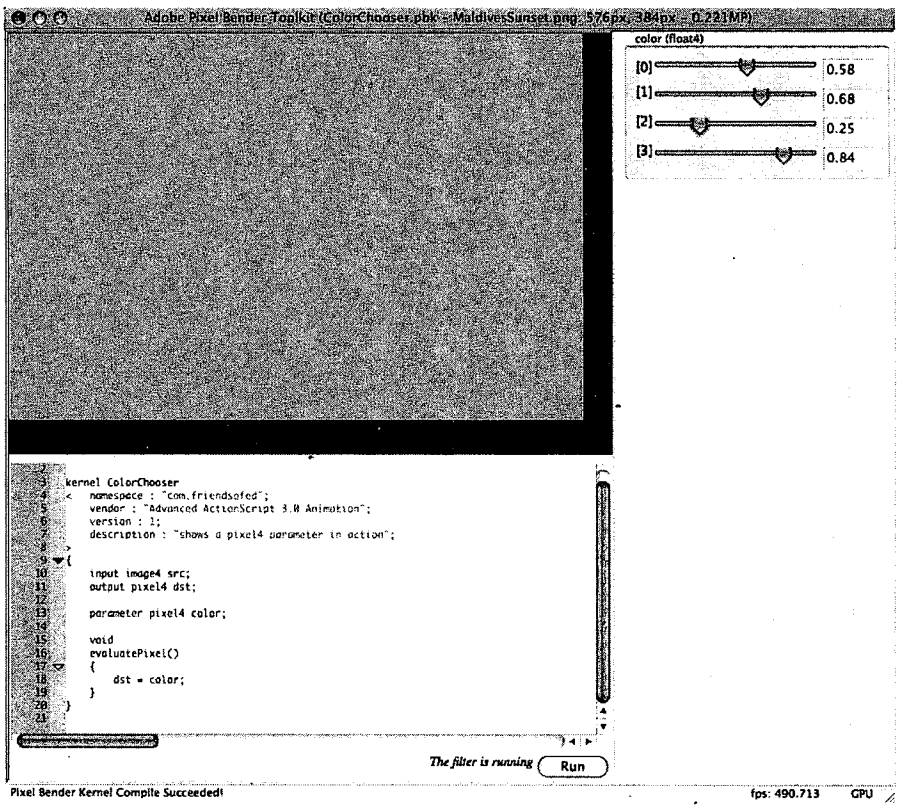


图9-6 4通道滑动条

移动滑块可以调整结果图像的颜色。当然，至少需要稍稍上移slider 3（α通道）才能看到变化。

高级参数

创建一个参数时，默认的最小值和最大值分别为0.0和1.0，初始值为0.0。对于某些参数来说

这可能很合适，但是在很多情况下你可能希望为最大值、最小值和初始值选择其他的值。为此，可以使用参数元数据来影响这些设置，如下所示：

```
parameter float myValue
<
    minValue:0.0;
    maxValue:100.0;
    defaultValue:50.0;
>;
```

需要指出这里的重要语法：元数据直接跟在变量名后面，而且元数据后面要有分号。如果不满足这个语法，Pixel Bender会报错。

设置包括minValue、maxValue和defaultValue，这些设置顾名思义，不需多做解释。其后是一个冒号，接下来是要设置的值，然后是一个分号。前面的例子会使滑块在0.0到100.0之间变化，初始值为50.0。

下面的例子使用了两个int参数来创建一个棋盘图案（相应文件为Checkerboard.pbk）：

```
<languageVersion : 1.0;>

kernel Checkerboard
<   namespace : "com.friendsofed";
    vendor : "Advanced ActionScript 3.0 Animation";
    version : 1;
    description : "creates a checkerboard pattern";
>
{
    input image4 src;
    output pixel4 dst;

    parameter int xres
    <
        minValue:1;
        maxValue:200;
        defaultValue:50;
    >;
    parameter int yres
    <
        minValue:1;
        maxValue:200;
        defaultValue:50;
    >;
    void
    evaluatePixel()
    {
        float2 pos = outCoord();
        float xpos = floor(pos.x / float(xres));
        float ypos = floor(pos.y / float(yres));
        if(mod(xpos, 2.0) > 0.0 ^^ mod(ypos, 2.0) > 0.0)
        {
            dst = pixel4(1, 1, 1, 1);
        }
        else
```

www.TopSage.com

TopSage.com

```
        {
            dst = pixel4(0, 0, 0, 1);
        }
    }
}
```

首先利用outCoord得到当前坐标。将它除以xres，对这个值向下取整。例如，如果图像宽度为500，xres为100，则会得到从1到5的值。注意，需要将xres的类型强制转换为一个浮点数，以便pos.x除以这个值。

然后对*y*轴做同样的处理。取各个值与2.0的余数［**余数**（modulus）是将一个值除以另一个值余留的数］。取一个整数与2的余数时，如果整数是偶数，则会得到0；如果这个整数是奇数则会得到1。^^操作符是一个异或操作符。它表示，如果任意一个条件为真，但不能二者都为真，则返回true。如果这个条件为true，则把像素置为白色，否则置为黑色。最后得到的效果就是一个棋盘图案。可以调整滑块来改变构成棋盘图案的方块（或矩形）的大小。

9.6 输入图像采样

到目前为止，我们一直只是使用输入图像来提供一个可以处理的区域。实际上，Pixel Bender本身是一个图像处理语言，这表示它能处理图像。所以，它理所当然需要提供一种方法来得到图像的数据。这是通过一组不同的采样函数做到的。**采样**（sampling）是指得到一个图像某个特定位置的像素值。

最简单的采样函数是sampleNearest。这个函数接受两个参数，分别是一个图像以及表示该图像中一个*x*和*y*位置的float2值，找出图像中与该点最接近的像素，并返回它的值。

现在可以再来看Pixel Bender原来生成的默认shader代码，并真正理解这段代码的含义：

```
void
evaluatePixel()
{
    dst = sampleNearest(src, outCoord());
}
```

我们传入了第一个参数src，这个参数代表输入图像，然后传入第二个参数outCoord()，这会返回当前像素float2形式的坐标。假设当前在计算像素（100,100），这会查看源图像中的像素（100, 100），并返回它的值。这样一来，由于逐个像素地进行处理，因此输出图像将与输入图像完全相同。当然，也可以有所调整，并不是对当前所计算的像素采样，而是对另一个不同像素进行采样。如果这样做，就可以将像素从其初始位置移动到其他位置而创建各种变形效果。下面的例子（见GlassTile.pbk）就完成了这个工作：

```
<languageVersion : 1.0;>

kernel GlassTile
<   namespace : "com.friendsofed";
    vendor : "Advanced ActionScript 3.0 Animation";
    version : 1;
    description : "creates a glass tile refraction effect";
>
```


www.TopSage.com

```
{
    input image4 src;
    output pixel4 dst;

    parameter float mult
    <
        minValue:0.0;
        maxValue:10.0;
        defaultValue:1.0;
    >;

    parameter float wave
    <
        minValue:0.0;
        maxValue:1.0;
        defaultValue:0.1;
    >;

    void
    evaluatePixel()
    {
        float2 pos = outCoord();
        pos += float2(sin(pos.x * wave) * mult,
                      sin(pos.y * wave) * mult);
        dst = sampleNearest(src, pos);
    }
}
```

并不是简单地采样当前像素，这个shader将x和y坐标偏移某个量。为此首先取当前x和y坐标的正弦，并稍稍缩减，然后乘以某个量。缩减工作通过wave参数完成，乘法操作则利用mult参数完成。其结果如图9-7所示。

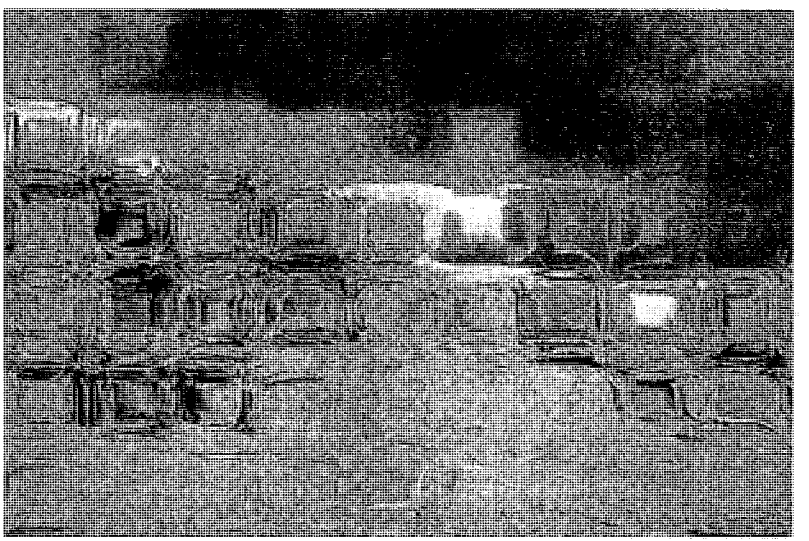


图9-7　玻璃砖效果

www.TopSage.com

线性采样

我曾提到过，有多种不同类型的采样。目前为止我们已经见过sampleNearest。再次说明，这种采样会取传入的*x*和*y*坐标，找到与之最接近的像素。来看图9-8。

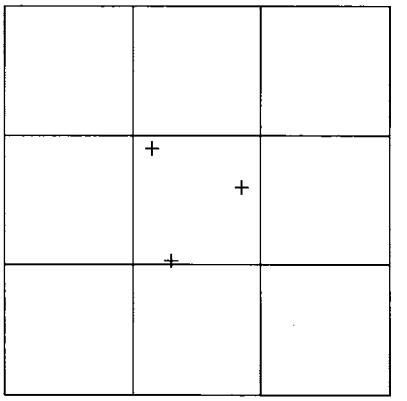


图9-8 所有3个采样点都会得到中间像素的值

图中显示了一个9像素区域的放大图。假设中间像素位置为（100,100）。3个十字记号显示了传入sampleNearest的3个不同的点。可能左上方的点是（99.7,99.6），右边的点是（100.4,99.9），下面的点是（99.8,100.5）。与这3个点最接近的像素都是（100,100），所以它们会得到同样的值。尽管这是一种相当简单的采样方法，而且运行起来可能相当快，但是处理结果会出现不均匀的马赛克（从图9-9可以看到，这是从GlassTile shader保存的一个图像的放大图）。

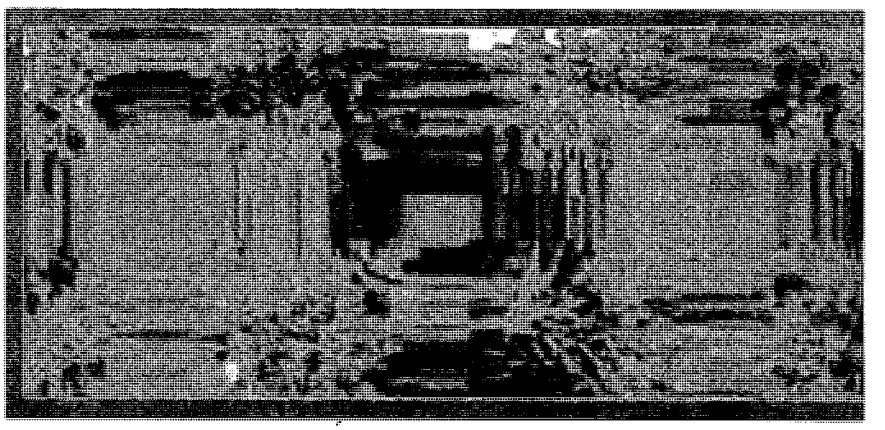


图9-9 sampleNearest导致的马赛克

如果你的shader中遇到这种问题，可以尝试采用另一种采样方法：sampleLinear。实际上，查看相关文档时，可以看到其中列出了3个采样方法：sample、sampleLinear和sampleNearest。不过，实际上sample只是sampleLinear的另一个名字而已，它们其实是同一个函数。

www.TopSage.com

大家网 TopSage.com

sampleLinear的取样方式是取4个最接近像素的某种加权平均。例如，在图9-10中，可以看到表示所传入点的十字记号。

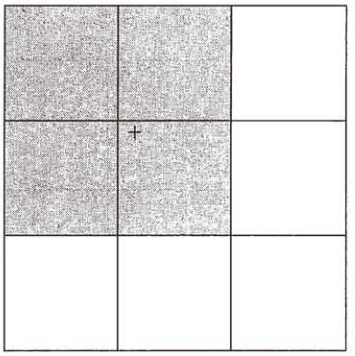


图9-10　采样点会得到4个最接近像素的加权平均

4个灰色像素表示将采样的4个像素。它们其实并不是取平均值，而只是完成一种线性插值。换句话说，在计算平均值时与采样点最接近的像素将比较远的像素有更大的权值。所以相似的采样点之间会得到一种更为平滑的混合。

要看到它的实际效果，只需将GlassTile.pbk中的代码行：

```
dst = sampleNearest(src, pos);
```

改为：

```
dst = sampleLinear(src, pos);
```

或者：

```
dst = sample(src, pos);
```

这样就可以去除之前的马赛克，得到平滑的渐变效果，如图9-11所示。

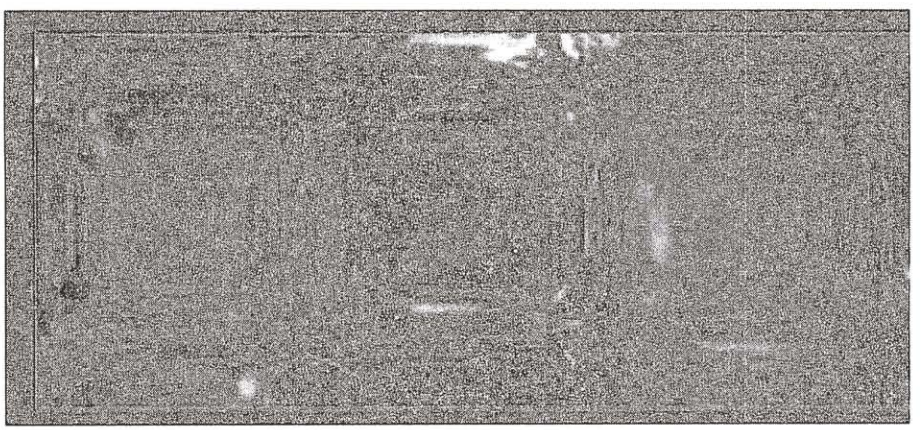


图9-11　sampleLinear得到的更平滑的渐变效果

www.TopSage.com

在了解Flash中如何具体使用shader之前再来看另外一种变换。

9.7 Flash 的旋转 Shader

在Pixel Bender Toolkit提供的示例shader中包括一个旋转滤镜，可以将像素绕某个给定点旋转。遗憾的是，这是使用某些新特性编写的，而为了在Flash中使用导出shader时这些特性无法使用。所以，要建立一个Flash支持的shader。

相应文件是TwirlFlash.pbk，从中可以看到这个shader：

```
<languageVersion : 1.0;>

kernel TwirlFlash
<   namespace : "com.friendsofed";
    vendor : "Advanced ActionScript 3.0 Animation";
    version : 1;
    description : "spins an image";
>
{
    input image4 src;
    output pixel4 dst;

    parameter float2 center
    <
        minValue:float2(0.0);
        maxValue:float2(1000.0);
        defaultValue:float2(200.0);
    >;

    parameter float twist
    <
        minValue:-10.0;
        maxValue:10.0;
        defaultValue:0.0;
    >;

    parameter float radius
    <
        minValue:0.0;
        maxValue:500.0;
        defaultValue:100.0;
    >;

    void
    evaluatePixel()
    {
        float2 pos = outCoord();
        float dist = distance(center, pos);
        float PI = 3.14159;

        if(dist < radius)
        {
            float dx = pos.x - center.x;
```

www.TopSage.com

```
            float dy = pos.y - center.y;
            float angle = atan(dy, dx);
            angle += sin(dist / radius * PI) * twist;
            float2 newpos = center + float2(cos(angle) * dist,
                                            sin(angle) * dist);
            dst = sampleNearest(src, newpos);
        }
        else
        {
            dst = sampleNearest(src, pos);
        }
    }
}
```

这里共有3个参数。

- center参数指定了回旋效果所围绕的点。要记住，Pixel Bender没有图像大小之类的信息，所以不同于Flash，这里没有场景宽度和高度的概念。
- twist参数确定了图像围绕中心点旋转的程度。
- radius参数确定了图像中会受到影响的范围。这个shader只影响中心点周围的一个圆形区域，这个圆形区域由这个半径定义。

在evaluatePixel函数中，我们要得到所计算的当前像素的位置，并使用内置的distance函数得到该像素与center参数的距离。还要为PI定义一个值，因为Pixel Bender不提供常数PI：

```
float2 pos = outCoord();
float dist = distance(center, pos);
float PI = 3.14159;
```

接下来有一个if语句，创建在Flash中使用的shader时，if语句可能是唯一可用的真正的控制结构。Flash中没有数组、for或while循环，也没有switch语句。也许这些限制会更加激发你的创造性！不论怎样，如果distance小于radius，则计算回旋效果，否则只是原样传入源图像像素。

然后完成一些数学运算。得到从center到pos在*x*轴和*y*轴上的距离分量，并使用这两个分量得到二者之间的角度（angle）：

```
float dx = pos.x - center.x;
float dy = pos.y - center.y;
float angle = atan(dy, dx);
```

现在，我们希望增加这个angle来完成变形。对于这里增加的量需要做一些解释。下面再给出这行代码以供参考：

```
angle += sin(dist / radius * PI) * twist;
```

首先计算出dist / radius * PI的值。dist的值介于0.0到radius之间，所以dist/radius的范围将介于0.0到1.0之间。将它乘以PI，取值范围则在0.0到3.1459之间。取这个值的正弦，会得到从0.0到1.0然后又回到0.0的一条平滑曲线。最后，再将它乘以twist值，并加到angle中，这说明回旋效果中心处没有任何变形。半径方向上处于中间位置时变形最大，而到达半径边缘又会恢复为没有任何变形。

www.TopSage.com

大家网 TopSage.com

现在有了一个距离和一个角，接下来找到位于该角度且距中心有此距离的点，采样源图像中的这个点。这就是要作为输出的pixel4值。可以看到如图9-12所示的结果。

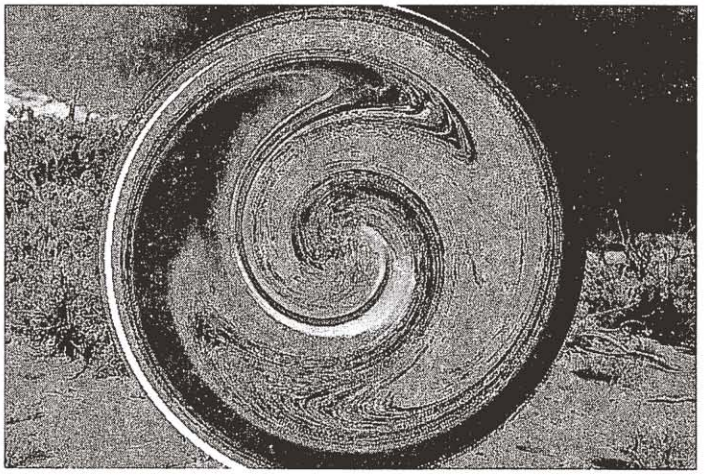


图9-12 TwirlFlash shader的实际作用

现在你已经了解了创建shader的基本原理以及所涉及的一些技术。可以在www.adobe.com上搜索有关的实验室和开发中心，或者直接在Web上简单搜索就能找到其他一些技术，还可以得到大量示例Pixel Bender文件，从中可以了解更多内容。另外，不要只局限于Pixel Bender。Pixel Bender基于GLSL语言，尽管它们存在一些区别，不过从GLSL语言的参考资料应该也能了解一些技术并得到一些提示。

无论如何，我相信你已经可以利用Pixel Bender Toolkit创建一些很酷的shader了。现在来看如何在Flash中使用这些shader。

9.8 在 Flash 中使用 Pixel Bender Shader

9.1节中已经谈到，Pixel Bender shader在Flash中有4种可能的用法：

- 滤镜；
- 填充；
- 混合模式；
- “通用数字处理”。

在这里我们将介绍前3种。使用shader完成高速的通用数学计算涉及范围太广，而且（写本书时）这还只是一个实验性课题。

在Flash中使用shader需要3个基本步骤：

(1) 采用Flash可读的一种格式导出shader；

(2) 将shader加载或嵌入到一个Flash动画中；

(3) 在Flash中创建一个shader实例，并使用该shader作为一个填充、滤镜或混合模式。

www.TopSage.com

下面就来具体完成这些步骤。首先要从Pixel Bender Toolkit导出shader。打开之前为本例创建的Checkerboard.pbk文件。为Flash导出shader相当简单，只需在File菜单中选择Export Kernel Filter for Flash Player。使用默认名，这应当与刚加载的内核文件同名。把它保存在Flash动画能够访问的某个位置。保存时它会有一个.pbj扩展名——这往往让我们想到“花生酱与果冻”（Peanut butter and jelly），不过那是另一码事。

现在需要创建一个Flash动画来利用这个shader。我们将创建一个文档类，可以用于Flex Builder、Flash CS4或者你喜欢的任何编码和编译方法。

9.8.1 加载 shader 与嵌入 shader

要在Flash中使用shader，需要在运行时加载，或者在编译动画时嵌入。我倾向于嵌入方式，这有几个原因。首先，如果shader编译到SWF中，就不再需要随SWF同时提供shader。其次，不需要等待shader的加载，因为只要SWF加载，其中嵌入的shader就会立即可用。这样会使编写代码容易得多，至少在开发和测试时如此。当然，加载shader也有一些好处，如果在某处修改了shader，那么只需替换联机版本，而不必将加载该shader的所有SWF重新编译。

总之，我首先会展示如何加载shader，然后介绍如何嵌入shader。

首先来创建加载其他内容（如XML）所用的基本框架，这里要使用一个URLLoader和一个COMPLETE事件处理函数。可以下载相应的完整文件ShaderFillDemo.as：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;

    public class ShaderFillDemo extends Sprite
    {
        private var loader:URLLoader;
        private var shaderURL:String = "Checkerboard.pbj";

        public function ShaderFillDemo()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            loader = new URLLoader();
            loader.addEventListener(Event.COMPLETE, onLoadComplete);
            loader.dataFormat = URLLoaderDataFormat.BINARY;
            loader.load(new URLRequest(shaderURL));
        }

        private function onLoadComplete(event:Event):void
        {

        }
    }
}
```

www.TopSage.com

要说明重要的一点，这里将加载器的数据格式设置为二进制，这是因为我们在加载二进制数据。还要确保.pbj shader文件与最终的SWF文件在相同位置上，这样shader才能成功加载，或者应当根据需要调整URL的路径。

既然shader数据已经加载，需要由它建立一个Shader对象。这是flash.display.Shader类的一个实例。加载器的data属性将包含二进制shader字节码。将它传入shader的构造函数，如下：

```
private function onLoadComplete(event:Event):void
{
    var shader:Shader = new Shader(loader.data);
}
```

不论使用shader完成什么用途（填充、滤镜或混合模式），都需要首先创建这个对象。下面用它来完成填充。

9.8.2 shader 用于填充

为此要使用Graphics方法beginShaderFill，并传入你想使用的shader。然后可以绘制你想要的任何图形，（类似于单色填充、渐变填充或位图填充），所绘制的形状将填充为通过shader创建的图案。以下是这个完整的类：

```
package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;

    public class ShaderFillDemo extends Sprite
    {

        private var loader:URLLoader;
        private var shaderURL:String = "Checkerboard.pbj";
        public function ShaderFillDemo()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            loader = new URLLoader();
            loader.addEventListener(Event.COMPLETE, onLoadComplete);
            loader.dataFormat = URLLoaderDataFormat.BINARY;
            loader.load(new URLRequest(shaderURL));
        }

        private function onLoadComplete(event:Event):void
        {
            var shader:Shader = new Shader(loader.data);
            graphics.beginShaderFill(shader);
            graphics.drawRect(0, 0, 400, 400);
```

www.TopSage.com

大家网 TopSage.com

```
            graphics.endFill();
        }
    }
}
```

如果一切正常，这样就会绘制出一个填充有棋盘图案的矩形。

很不错。很快我们会了解如何设置参数和其他一些方法，不过首先来看如何利用嵌入完成同样的工作。为此，我们将使用Embed元数据标记将shader编译到SWF中，从而无需再加载。这与嵌入外部位图、SWF、字体等完全相同。以下是相应的基本设置，可以从ShaderFillEmbed.as得到：

```
package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    public class ShaderFillEmbed extends Sprite
    {
        [Embed (source="Checkerboard.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;

        public function ShaderFillEmbed()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
        }
    }
}
```

注意，这里将mimeType设置为"application/octet-stream"。另外，不要忘记将.pbj文件放在源文档类的同一目录下，否则需要调整路径指向该目录，以便编译器能够找到这个shader。

现在已经准备就绪，可以采用与上一个例子中同样的方式来创建一个shader。不过，现在字节码包含在ShaderClass类中，所以建立该类的一个实例，并传入shader构造函数：

```
var shader:Shader = new Shader(new ShaderClass());
```

与前面一样，现在可以在beginShaderFill中使用这个shader实例。下面再次给出这个完整的类：

```
package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    public class ShaderFillEmbed extends Sprite
    {
        [Embed (source="Checkerboard.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;
```

www.TopSage.com

```
        public function ShaderFillEmbed()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var shader:Shader = new Shader(new ShaderClass());
            graphics.beginShaderFill(shader);
            graphics.drawRect(0, 0, 400, 400);
            graphics.endFill();
        }
    }
}
```

对我来说，这比加载更容易，这也是我更倾向于使用这种方法的原因。不过不论采用哪一种方法向SWF加入shader，本章余下的技术都同样适用。首先来看访问shader元数据。

9.8.3 在 Flash 中访问 shader 元数据

在Flash中一旦有了shader实例，就可以访问在Pixel Bender Toolkit中创建shader时设置的元数据。为此要通过shader本身的data属性来做到。这是flash.display.ShaderData类的一个实例。根据这个属性，可以访问创建shader时设置的namespace、vendor、version和description元数据属性。还可以访问shader本身的name，这就是所写内核的名字。这些都通过字符串提供。只需为上一个例子增加以下代码行就可以看到具体如何访问元数据：

```
        public function ShaderFillEmbed()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var shader:Shader = new Shader(new ShaderClass());
            trace(shader.data.name);
            trace(shader.data.namespace);
            trace(shader.data.vendor);
            trace(shader.data.version);
            trace(shader.data.description);

            graphics.beginShaderFill(shader);
            graphics.drawRect(0, 0, 400, 400);
            graphics.endFill();
        }
```

其输出如下所示：

```
Checkerboard
com.friendsofed
Advanced ActionScript 3.0 Animation
1
creates a checkerboard pattern
```

尽管并不总是需要知道这些信息，知道能够获取这些信息总是有好处的。如果要创建一个允许加载和应用不同shader的工具，获取这些信息会很有帮助，因为你可以在界面上列出这些数据。

www.TopSage.com

9.8.4　在 Flash 中设置 shader 参数

在Pixel Bender Toolkit中，只要创建一个参数就会得到滑动条或者复选框，从而可以在toolkit本身中改变这些参数并查看结果。当然，在Flash中不会看到滑动条或复选框，但是确实需要某种方法通过代码来影响这些参数。这同样要利用shader的data属性做到。不过这没有你想象中那么简单。

在shader中创建的各个参数都是shader data属性上的一个属性。所以，对于我们的棋盘shader，将有：

```
shader.data.xres
```

和

```
shader.data.yres
```

现在你可能认为只需以下一行代码就可以完成它们的设置：

```
shader.data.xres = 20;
```

如果愿意，你可以试试看，不过可以看到这并不奏效。它不会抛出错误或任何警告，但是也不会改变参数。需要进一步深入到该参数的value属性：

```
shader.data.xres.value
```

不过请等等，还不只是如此！value属性需要一个值的数组，而不只是一个值。记住，参数可以是可能包含至多4个值的某种向量类型。所以即使只传入一个值，也需要作为一个数组传入（在赋值时添加一对方括号），如下：

```
shader.data.xres.value = [20];
```

最后，这会如期把值传入参数。对于上面的例子，会得到如图9-13所示的结果。

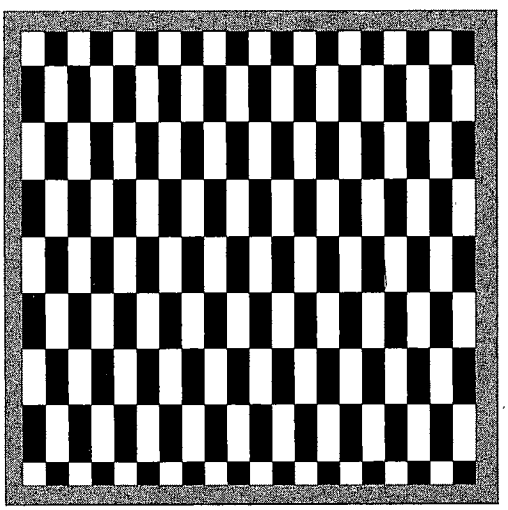


图9-13　利用一个填充得到的棋盘图案

也可以试着改变yres参数。

www.TopSage.com

TopSage.com

9.8.5 变换 shader 填充

与其他填充方法一样，beginShaderFill还有第二个flash.display.Matrix类型的参数。可以使用这个参数在绘制shader时完成缩放、旋转或平移。如果在Flash帮助文件中查阅Matrix类，会看到利用以下参数可以将矩阵旋转某个角度，其中q是所要旋转的角度，单位是弧度：

```
new Matrix(cos(q), sin(q), -sin(q), cos(q), 0, 0);
```

当然，正如Seb（本书技术审校）指出的，可以直接写作：

```
var m:Matrix = new Matrix();
m.rotate(q);
```

现在你知道了可以用两种方法完成同一件事！这往往很有好处。通过使用这个数据，可以在绘制shader时将shader旋转任何角度，如下：

```
public function ShaderFillEmbed()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    var shader:Shader = new Shader(new ShaderClass());

    var angle:Number = Math.PI / 4; // 45 degrees in radians
    var cos:Number = Math.cos(angle);
    var sin:Number = Math.sin(angle);
    graphics.beginShaderFill(shader, new Matrix(cos, sin, -sin, cos));
    graphics.drawRect(0, 0, 400, 400);
    graphics.endFill();
}
```

这会在绘制shader时将它旋转45°（π/4弧度），得到如图9-14所示的结果。

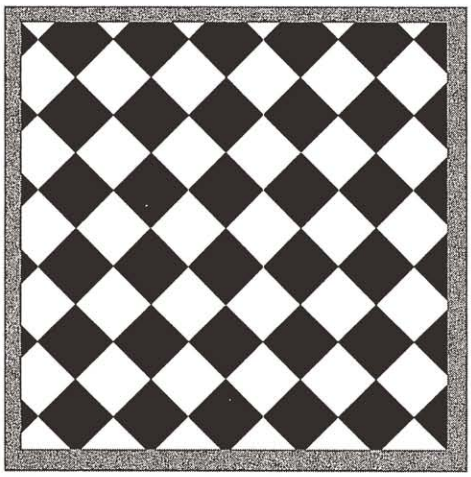


图9-14 有一个旋转矩阵的棋盘图案

当然，类似地，对矩阵使用不同的参数还可以缩放、平移甚至扭斜一个shader填充。

www.TopSage.com

9.8.6　实现 shader 填充动画

实现shader动画最能体现参数的巨大作用。其思路很简单：绘制一个shader填充，改变一个参数，下一帧时再次应用该shader填充，如此继续。下面展示了一个非常简单的例子，这里再次使用了棋盘图案（相应文件为ShaderFillAnim.as）：

```
package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    public class ShaderFillAnim extends Sprite
    {
        [Embed (source="Checkerboard.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;

        private var shader:Shader;
        private var xAngle:Number = 0;
        private var yAngle:Number = 0;
        private var xSpeed:Number = .09;
        private var ySpeed:Number = .07;

        public function ShaderFillAnim()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            shader = new Shader(new ShaderClass());

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var xres:Number = Math.sin(xAngle += xSpeed) * 50 + 55;
            var yres:Number = Math.sin(yAngle += ySpeed) * 50 + 55;

            shader.data.xres.value = [xres];
            shader.data.yres.value = [yres];
            graphics.clear();
            graphics.beginShaderFill(shader);
            graphics.drawRect(20, 20, 400, 400);
            graphics.endFill();
        }
    }
}
```

这里为*x*轴与*y*轴上的角度和速度创建了相应属性。在每一帧上将速度增加到角度，并取该值的正弦。将它乘以50再加上55，就会得到取值范围在5到105之间的值。将这些值传给shader的xres和yres参数。清空图像，再用这个shader填充重绘形状，这样就会得到动画。

www.TopSage.com

9.8.7 指定 shader 输入图像

在前面的例子中，shader生成的图案纯粹根据数学公式计算出来的。不过，如果希望像TwirlFlash示例中一样使用另一个shader该怎样做呢？这个shader实际上是一个图像处理shader，因为它的像素输出直接取决于一个输入图像。下面来看看如果使用这样一个输入图像会发生什么情况。下面的例子可以从ShaderFillImage.as文件得到：

```
package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.filters.ShaderFilter;

    public class ShaderFillImage extends Sprite
    {
        [Embed (source="TwirlFlash.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;

        public function ShaderFillImage()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var shader:Shader = new Shader(new ShaderClass());
            shader.data.center.value = [400, 300];
            shader.data.twist.value = [-6];
            shader.data.radius.value = [300];

            graphics.beginShaderFill(shader);
            graphics.drawRect(20, 20, 800, 600);
            graphics.endFill();
        }
    }
}
```

如果不加修改地运行这个程序，你会得到一个错误消息——缺少shader输入源或者类型不支持（The Shader input src is missing or an unsupported type）——因为这个shader试图在一个不存在的图像上采样像素。好在我们可以简单地传入一个图像用作为源图像。

首先需要在文件中加入BitmapData。这里加入了一张我好朋友的照片，把它嵌入到类中：

```
package {
    import flash.display.Bitmap;
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.filters.ShaderFilter;

    public class ShaderFillImage extends Sprite
    {
```

www.TopSage.com

大家网 TopSage.com

```
        [Embed (source="TwirlFlash.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;

        [Embed (source="john_davey.jpg")]
        private var JohnDavey:Class;

        public function ShaderFillImage()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var shader:Shader = new Shader(new ShaderClass());
            shader.data.center.value = [400, 300];
            shader.data.twist.value = [-6];
            shader.data.radius.value = [300];

            var image:Bitmap = new JohnDavey();
            shader.data.src.input = image.bitmapData;

            graphics.beginShaderFill(shader);
            graphics.drawRect(20, 20, 800, 600);
            graphics.endFill();
        }
    }
}
```

然后可以建立该嵌入图像的一个实例，其类型将是Bitmap。BitmapData可以通过该Bitmap的bitmapData属性来访问。把它赋值给shader.data.src.input，现在shader就有了一个可以采样的输入图像。其结果如图9-15所示。

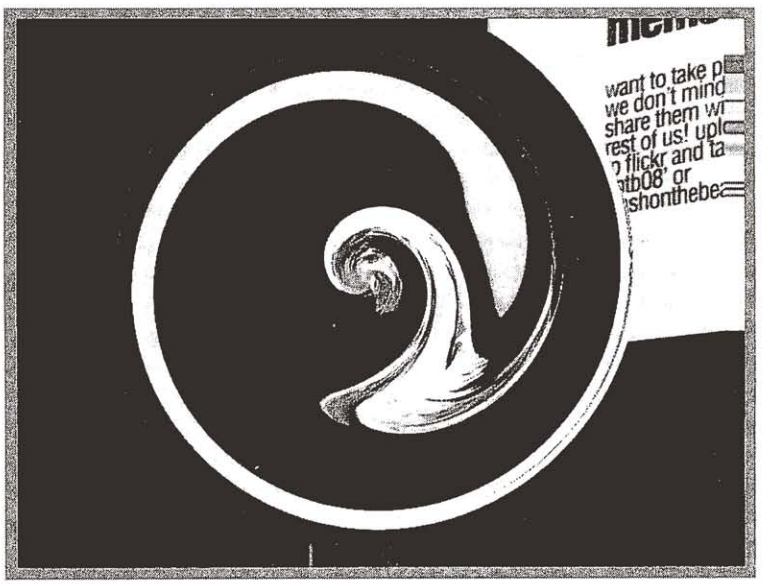


图9-15　TwirlFlash shader用作填充

www.TopSage.com

需要指出，shader.data.src.input中的src不是一个幻值。它直接取自shader内核代码中的下面这行代码：

```
input image4 src;
```

如果用另外某个变量名（如source）命名这个输入图像，就必须在ActionScript中反映出这一点：

```
shader.data.source.input = image.bitmapData;
```

以上已经介绍了将shader用作填充时所需了解的全部内容。实际上，我已经介绍了关于Shader类通常需要知道的大部分信息。下面将介绍如何使用shader作为滤镜和混合模式，不过之前介绍的所有内容对这些主题也同样适用。

9.9 使用 Shader 作为滤镜

使用shader作为滤镜要比用它作为填充更容易。再来使用TwirlFlash shader，不过这里不再使用一个图像，下面来证明shader可以在任何显示对象上使用，甚至是一个文本域。下面的文件名为ShaderAsFilter.as：

```
package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.filters.ShaderFilter;
    import flash.text.TextField;
    import flash.text.TextFormat;

    [SWF(backgroundColor=0xFFFFFF)]
    public class ShaderAsFilter extends Sprite
    {
        [Embed (source="TwirlFlash.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;

        public function ShaderAsFilter()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var shader:Shader = new Shader(new ShaderClass());
            shader.data.center.value = [200, 200];
            shader.data.twist.value = [-1];
            shader.data.radius.value = [200];

            var tf:TextField = new TextField();
            tf.width = 400;
            tf.height = 400;
            tf.wordWrap = true;
            tf.multiline = true;
            tf.border = true;
```

www.TopSage.com

```
            tf.defaultTextFormat = new TextFormat("Arial", 24);
            addChild(tf);

            for(var i:int = 0; i < 340; i++)
            {
                tf.appendText(String.fromCharCode(65 + Math.random() * 25));
            }
            tf.filters = [new ShaderFilter(shader)];
        }
    }
}
```

这里创建了一个文本域，并填入一些随机文本。在最后一行代码中，将文本域的filters属性设置为一个数组，其中包含一个新的ShaderFilter，并传入我们刚创建的shader。再简单不过了！

9.10　使用 Shader 作为混合模式

最后一个用例与其他示例稍有点不同。实际上，我们需要回过头去创建一个新的在用作混合模式时能正确工作的shader。

首先，混合模式shader需要有两个输入图像：一个用作背景，另一个用作前景。需要在内核中定义这两个输入图像，不过在Flash中这是自动完成的——应用shader的显示对象将作为前景传入，而位于它下面的显示对象将作为背景传入。实际上，具体发生的情况是：对任何显示对象应用一个混合模式时，这个对象的cacheAsBitmap属性都会被设置为true。这会生成显示对象的一个位图表示（如果存在），并作为一个源图像传入shader。

对于在shader里能完成的工作还存在一些限制。它不能像其他shader那样生成漂亮的正弦曲线、棋盘图案或者完成变形。在Pixel Bender Toolkit中这是可以的，但是一旦使用shader作为混合模式，这些工作都不会做。基本说来，所能做的只是从两个输入图像采样当前像素，然后以你希望的任何方式加以结合。下面的文件（ChannelBlend.pbk）展示了这一点：

```
<languageVersion : 1.0;>

kernel ChannelBlend
<   namespace : "com.friendsofed";
    vendor : "Advamced ActionScript 3.0 Animation";
    version : 1;
    description : "blends the channels of two images";
>
{
    input image4 back;
    input image4 fore;

    output pixel4 dst;

    parameter float3 amt;

    void
    evaluatePixel()
    {
```

www.TopSage.com

TopSage.com

```
        pixel4 bg = sampleNearest(back, outCoord());
        pixel4 fg = sampleNearest(fore, outCoord());
        dst.r = fg.r * amt.r + bg.r * (1.0 - amt.r);
        dst.g = fg.g * amt.g + bg.g * (1.0 - amt.g);
        dst.b = fg.b * amt.b + bg.b * (1.0 - amt.b);
        dst.a = 1.0;
    }
}
```

首先声明两个image4变量fore和back。如何命名并不重要。声明的第一个变量将接收背景图像，第二个变量将接收前景图像。

在evaluatePixel函数中，首先在outCoord位置对两个图像采样。对于用作混合模式的所有shader来说，目前为止所做的这些工作可能都是必要的。在此之后，可以采用你希望的任何方式结合这两个像素。在这里，我根据amt参数来完成二者的混合，这样可以分别混合所有通道。为了在Pixel Bender Toolkit中测试这个例子，必须通过File菜单加载两个图像。加载的第一个图像为背景图，第二个图像将为前景图。可以通过拉动amt参数创建的滑块来测试这个效果。

现在，再回到Flash（相应文件是ShaderBlendMode.as）：

```
package {
    import flash.display.Bitmap;
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    public class ShaderBlendMode extends Sprite
    {
        [Embed (source="ChannelBlend.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;

        [Embed (source="input1.jpg")]
        private var input1:Class;

        [Embed (source="input2.jpg")]
        private var input2:Class;

        public function ShaderBlendMode()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var back:Bitmap = new input2();
            addChild(back);

            var fore:Bitmap = new input1();
            addChild(fore);

            var shader:Shader = new Shader(new ShaderClass());
            shader.data.amt.value = [1.0, 0.9, 0.0];
            fore.blendShader = shader;
```


www.TopSage.com

```
        }
    }
}
```

在这里我们嵌入并实例化两个位图图像，并把它们放在场景上。根据刚才创建的ShaderBlend shader创建了一个新的shader，设置一些值，并把这个shader直接赋给最上面位图的blendShader属性。应该可以看到如图9-16所示的结果。

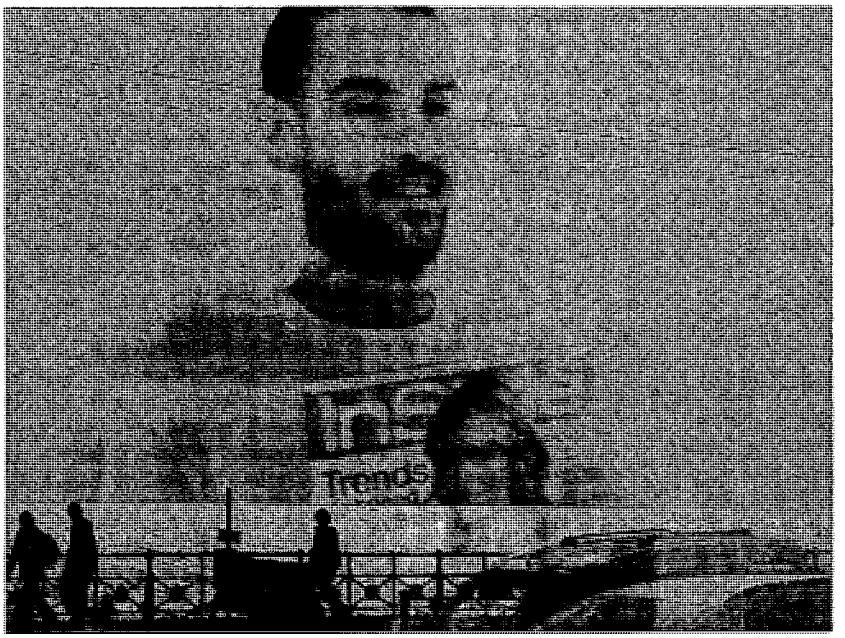


图9-16　利用ChannelBlend shader混合的两个图像

如果之前用过混合模式，你可能习惯于为blendMode属性赋一个字符串值。实际上，将一个shader赋给显示对象的blendShader属性时会自动地把blendMode设置为"shader"。

可以试着改变这些参数，看看shader与在toolkit中是否有相同的效果。

9.11　小结

本章尽管篇幅不大，但介绍的内容却相当多，至少足以让你着手创建shader并在Flash中使用。现在要由你来提出一些很酷且很有用的shader了。期待你的作品。

www.TopSage.com

第10章

补间引擎 10

在《Flash ActionScript 3.0 动画教程》的第8章，我讨论过这样一个主题：**缓动**（easing）。我称这种类型的缓动为**简单缓动**（simple easing），因为它实现起来相当容易，功能和灵活性也很有限。这种缓动包括一个对象以及该对象要移到的目标位置。每一帧都计算对象与目标之间的距离，并让对象移动到这段距离的中点（或者另外某个比例）。对象会平滑地移动，随着目标的接近逐渐减慢。

在那一章中我引用了Robert Penner的缓动函数，并建议你研究这些函数［我把它们称为**高级缓动**（advanced easing）］。当时我本来打算详细讨论这个主题，但是那本书的篇幅有限，已经没有空间来写这个内容了，现在终于有机会来讨论这个主题。Robert Penner的缓动公式已经得到了广泛的使用，不仅可以与Flash和Flex builder提供的补间类结合使用，还可以与这些年来新出现的众多补间引擎结合。

简单提一句：本章从现在开始，当提到**补间**（tween或tweening）时，就是指利用代码随时间移动一个对象或者改变某个属性——通常会使用某个预定义的补间类。这与使用时间表处理补间没有任何关系。

Adobe发布了两个不同的Tween类：`fl.transitions.Tween`（包含在Flash创作环境提供的类中），以及`mx.effects.Tween`（包含在Flex框架中）。这两个类的功能基本相同：它们取一个起始值和一个结束值，在一个指定的时间段内在二者之间插入值。这两个类都基于Robert Penner的缓动公式建立，允许你实现一些复杂的缓动，包括利用不同公式实现缓入和缓出，弹入和弹出，以及指定一个缓动从始至终的总时间，等等。

补间引擎（tweening engine）是这样一些代码框架，其中大多数代码框架（也使用了Robert Penner的缓动公式）的目标都是为了更容易地完成补间。我不能肯定现在有多少种ActionScript补间引擎。开始写这本书时我曾列出了一些主要的补间引擎。等到写到本章时，就不得不再向这个列表上增加两三个新的引擎。

现在让我来告诉你本章不讨论的内容。

首先，这绝对不是目前所有ActionScript 3.0补间引擎的一个权威指南。与本书中许多其他章节一样，这个主题本身就需要整本书来介绍，而且这样一本书一旦出版就会立即过时。实际上，如果你希望对Flash中补间引擎的背景有更多了解，可以看看*The Essential Guide to Open Source Flash Development*（Apress，2008）的第10章。在那一章中，Moses Gunesch从补间引擎的历史谈

www.TopSage.com

起，并一直介绍到他自己开发的一个非常流行的Fuse补间工具包。然后还介绍了他的新项目GoASAP，这是一个用来创建补间引擎的框架。我原来打算在本章中写一些有关Go的内容，不过我认为我的介绍肯定比不上Moses。

本章不打算采用罗列的方式来阐释，不会列出一组“最佳的”补间引擎或者我最喜欢的一组补间引擎，也不会特别推荐某种引擎。所有补间引擎都各有优缺点，而且处理问题也会采用不同的方法。

最后，不能把本章作为我在这里提到的任何一款引擎的全面指南。

本章只是对不同类型补间引擎的一个介绍和概述，以便你确定哪一种引擎最适用于你的具体用途。我会尽量根据完成补间所采用的方法来简要介绍各种不同类型的引擎，并介绍使用各个引擎完成的一些基本任务。这样一来，你就可以对它们进行比较和对照，看看它们各自的长处和短处。除了Adobe提供的类，我选择的引擎包括：

- Tweener
- TweenLite/TweenGroup
- KitchenSync
- gTween

先从Adobe类开始介绍。

10.1　Flash Tween 类

前面提到过，这个Tween类是安装Flash创作环境时一同安装的类库的一部分。它放在fl.transtions包中（这个包中还包括你可能不知道的另外一些有意思的类——它们能做出各种各样神奇的变换效果）。不过，对于本章来说，我们将只讨论Tween类。

假设在使用Flash创作工具进行编译，因此fl.transitions包将可以直接使用。也可以使用Flex Builder或另外一个基于mxlmc的方法编译这些例子，不过在这种情况下需要将flash.swc文件（可以在Flash安装目录中找到）添加到你的库路径。我不会说明如何做到这一点，因为对于每一种工具其做法都不同，而且我认为对于这一点应该没有人会有疑问。

这个类的工作方式是，首先创建这个类的一个实例，传入一个对象引用和该对象上某个属性的名称；一个缓动函数（基于Robert Penner的缓动公式），该属性的一个起始值和结束值，以及一个持续时间。下面是构造函数：

```
new Tween(object, property, easingFunction, begin, finish, duration, useSeconds)
```

object可以是任何至少拥有一个公共数字属性的对象。property参数是一个字符串，表示所要改变的属性的名称。例如，如果object是一个Sprite，你希望对该sprite的*x*位置完成补间，则要传入"x"（包括引号）作为property。

easingFunction是flash.transitions.easing包中某个类中定义的一个函数。稍后将介绍这个函数。

begin和finish属性都是数字值。补间开始时，会把指定的object属性设置为begin值。补间

www.TopSage.com

完成时，该属性将等于finish值。

默认情况下，duration指定补间会经过多少帧。最后一个参数是一个名为useSeconds的布尔值。它默认为false，这表示补间将把duration解释为帧数。如果将最后一个参数设置为true，duration将被解释为秒数。

下面来看看效果。第一个例子在一个名为FlashTween.as的文件中，可以从本书网站www.-friendsofed.com下载：

```
package {
    import fl.transitions.Tween;
    import fl.transitions.easing.None;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class FlashTween extends Sprite
    {
        private var tween:Tween;
        private var sprite:Sprite;
        public function FlashTween()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            tween = new Tween(sprite, "x", None.easeIn, 100, 500, 1, true);
        }
    }
}
```

其中用粗体显示的代码行就是创建补间的代码。需要指出，一旦开始创建它就开始运行，要让它开始并不需要另外做任何事情。sprite（其中绘制有一个红色的矩形）会在1秒内从*x*位置100移动到*x*位置500。

10.1.1 缓动函数

在刚才创建的示例中，缓动函数定义为None.easeIn。你可能会奇怪这是什么意思。这个缓动函数参数确定了使用哪个公式来完成动画的缓入和缓出。换句话说，如果你在对一个对象的位置完成补间，这个参数会决定它以多快的速度从静止（速度为0）变到全速，然后以多快的速度在目标点“刹车”减速并停止？

www.TopSage.com

fl.transitions.easing包中有很多类：

- Back
- Bounce
- Elastic
- None
- Regular
- Strong

其中每个类都有3个方法：

- easeIn
- easeOut
- easeInOut

None类还有一个easeNone方法。

easeIn方法控制补间如何开始以及怎样达到全速。

easeOut方法控制补间如何减慢，并在接近其目标时停止。

easeInOut方法对这两方面都进行控制。

这些方法都在内部由Tween类本身调用。你完全不必自行调用这些方法，而只需要向Tween构造函数传入一个方法作为参数。这些方法包含了Robert Penner写的缓动公式。

这个例子使用了None缓动类的easeIn方法。这个类创建了一个线性缓动，这实际上不能算是缓动。补间从全速开始，并维持这个速度直至到达目标——在这里它会突然停止。实际上，使用None类的哪个方法并不重要——它们都会导致同样的线性行为。

这还没有太大意思。下面来看其他一些方法的工作。先来尝试Regular。这个类创建的缓动效果与采用以往做法得到的缓动效果完全相同（在Flash中完成老式时间表补间并选择100%缓入或缓出）。将创建补间的代码行修改如下：

```
tween = new Tween(sprite, "x", Regular.easeIn, 100, 500, 1, true);
```

现在可以看到sprite会缓慢地启动，逐渐加速，并突然停止。如果尝试Regular.easeOut，会看到sprite一开始就快速启动，而在靠近目标时慢下来，这与《Flash ActionScript 3.0 动画教程》中描述的简单缓动过程基本相同。再来尝试Regular.easeInOut，这会结合缓入和缓出。这是一种相当专业的效果，特别适合于平滑地移动各种界面元素。当然，也可以尝试使用其他一些参数，比如改变duration或start和end属性。还可以尝试对除x以外的其他属性完成补间，如y或rotation。试着将alpha从0缓动到1，如下：

```
tween = new Tween(sprite, "alpha", Regular.easeInOut, 0, 1, 1, true);
```

Strong缓动类与Regular完成的工作相同，不过效果更为出色。Elastic在缓入和缓出时会完成一种有弹性的振动。你可能希望增加持续时间以便看得更清楚：

```
tween = new Tween(sprite, "x", Elastic.easeInOut, 100, 800, 3, true);
```

Bounce类也会产生弹性效果，不过像是物体从一个坚硬的表面弹起的效果。最后介绍Back类，

www.TopSage.com

大家网 TopSage.com

它在开始移动之前先向反方向移动一点，稍稍超过目标，再缓动回到目标。可以尝试使用这些类，并对它们有所认识。现在你应该很清楚对于缓动函数需要知道些什么——至少对于Flash Tween类应该已经了解需要掌握哪些内容。

10.1.2 结合补间

第一个例子可以很好地工作，但是只是对单个对象上的单个属性完成了补间。如果只是对一个对象上的一个属性实现补间，而且所有补间都完全独立，那么你的任务已经完成。不过你可能希望同时改变一个对象的多个属性（至少是*x*和*y*坐标）。有时可能还希望先对某个方面完成补间，一旦该补间完成再开始另一个补间。

同时发生的多个补间称为**并行补间**（parallel tween）或组，而一个接一个连续发生的补间称为**补间序列**（tween sequence）。

Flash Tween类中的并行补间相当简单。只需为希望完成补间的各个属性和对象分别建立一个新的补间。假设你想要一个对象在*x*轴和*y*轴上移动，可以如下创建两个补间：

```
tween1 = new Tween(sprite, "x", Regular.easeInOut, 100, 800, 3, true);
tween2 = new Tween(sprite, "y", Regular.easeInOut, 100, 400, 3, true);
```

可以根据需要创建更多的补间。它们不必使用相同的缓动函数或相同的持续时间，如下所示：

```
tween1 = new Tween(sprite, "x", Regular.easeInOut, 100, 800, 3, true);
tween2 = new Tween(sprite, "y", Regular.easeInOut, 100, 400, 3, true);
tween3 = new Tween(sprite, "rotation", Strong.easeInOut, 0, 360, 4, true);
```

需要指出，rotation缓动使用Strong缓动类，用时为4秒，而x和y使用了Regular缓动类，用时为3秒。注意在这些情况下，我还建立了另外的补间属性来保存这些新的补间：

```
private var tween1:Tween;
private var tween2:Tween;
private var tween3:Tween;
```

序列则要更复杂一些：需要等待一个缓动结束才能开始下一个缓动。为此，需要监听一个事件来告诉你补间何时完成（TweenEvent.MOTION_FINISH事件）。假设你希望上一个例子中的3个补间一个接一个运行而不是同时运行。下一个类（见FlashTweenSequence.as文件）给出了一个简单的例子，展示了如何做到这一点：

```
package {
    import fl.transitions.Tween;
    import fl.transitions.TweenEvent;
    import fl.transitions.easing.Regular;
    import fl.transitions.easing.Strong;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
```


www.TopSage.com

```
public class FlashTweenSequence extends Sprite
{
    private var tween1:Tween;
    private var tween2:Tween;
    private var tween3:Tween;
    private var sprite:Sprite;
    public function FlashTweenSequence()
    {
        stage.scaleMode = StageScaleMode.NO_SCALE;
        stage.align = StageAlign.TOP_LEFT;

        sprite = new Sprite();
        sprite.graphics.beginFill(0xff0000);
        sprite.graphics.drawRect(-50, -25, 100, 50);
        sprite.graphics.endFill();
        sprite.x = 100;
        sprite.y = 100;
        addChild(sprite);

        tween1 = new Tween(sprite, "x", Regular.easeInOut, 100, 800, 3, true);
        tween1.addEventListener(TweenEvent.MOTION_FINISH, onTween1Finish);
    }

    private function onTween1Finish(event:TweenEvent):void
    {
        tween2 = new Tween(sprite, "y", Regular.easeInOut, 100, 400, 3, true);
        tween2.addEventListener(TweenEvent.MOTION_FINISH, onTween2Finish);
    }

    private function onTween2Finish(event:TweenEvent):void
    {
        tween3 = new Tween(sprite, "rotation", Strong.easeInOut, 0, 360, 4, true);
    }
}
}
```

这里创建了第一个补间，并立即为MOTION_FINISH事件增加了一个事件监听器。第一个补间完成时，我们创建了第二个补间，并监听同一个事件。事件触发时，再创建最后一个补间。这里我只是力求简单，不过在更完备的应用中，你可能还希望对事件监听器进行管理，比如一旦处理完就将其删除等。

如果有一个复杂的界面，其中有大量对象在移动，就会有非常多的事件处理程序，维护所有这些事件处理程序在逻辑上将会是一场恶梦。当然，完成这种建构还有其他方法，比如可以利用一个事件处理程序检查事件的目标，并相应地做出反应。但是不论怎样做都会很复杂。

由于需要解决诸如此类的复杂问题，本章后面将讨论的各种补间引擎才应运而生。不过在此之前，先来简单看看Flex框架提供的Tween类。

10.2　Flex Tween 类

尽管Flex Tween类是Flex框架的一部分，但并不需要创建一个Flex工程才能使用这个类。可以

在ActionScript 3.0工程中直接使用，或者甚至可以在一个基于Flash CS4的动画中使用。不过，要在一个并非基于Flex的工程中使用时，需要将framework.swc文件增加到库路径。这个SWC位于安装Flex框架的文件夹中。在这个文件夹中，其路径为frameworks/libs/framework.swc。

在Flex Builder中，可以通过Project Properties面板，在ActionScript Build Path的Library path标签页中增加SWC（参见图10-1）。

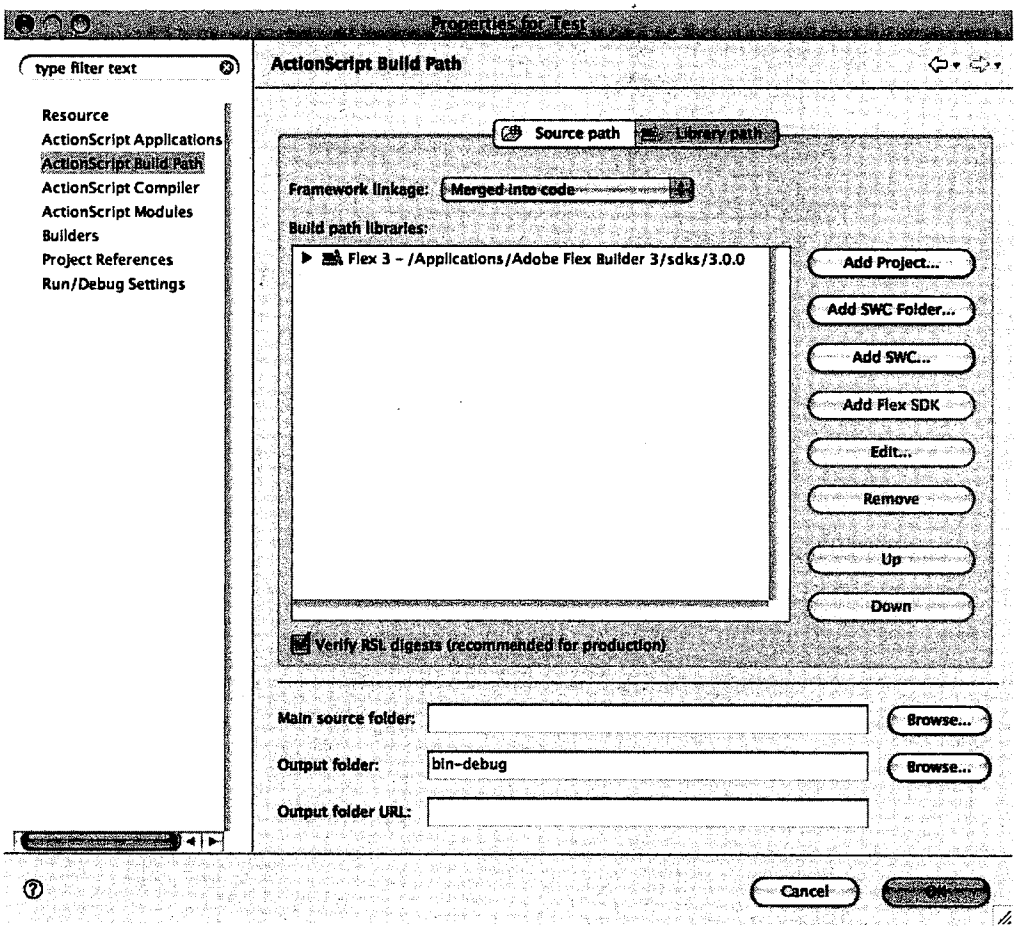


图10-1 在这个对话框中增加SWC

点击Add SWC，并找到framework.swc文件，参见图10-2。

在Flash CS4中，打开Publish Settings面板，点击ActionScript 3.0 Settings按钮，并进入Library path部分。参见图10-3。

点击+图标在列表中添加新条目，然后点击红色的图标（..）找到framework.swc文件。完成时，面板将如图10-4所示。

www.TopSage.com

大家网 TopSage.com

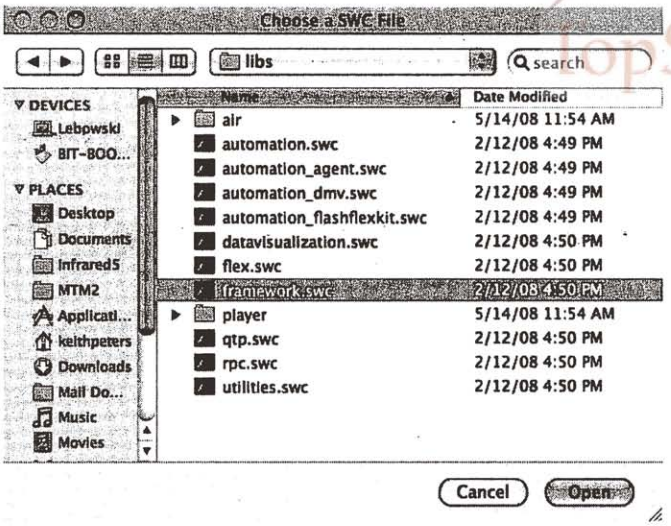


图10-2　添加文件

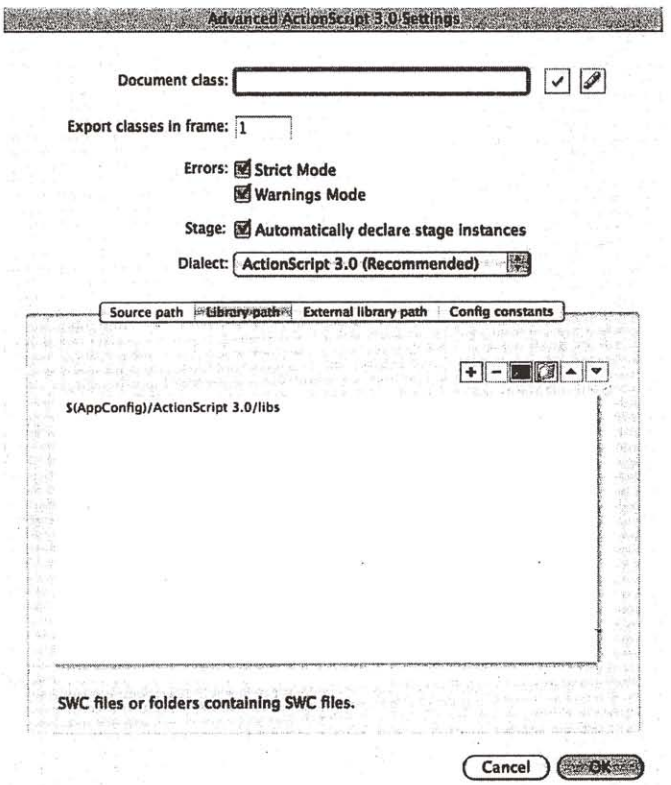


图10-3　Library path部分

www.TopSage.com

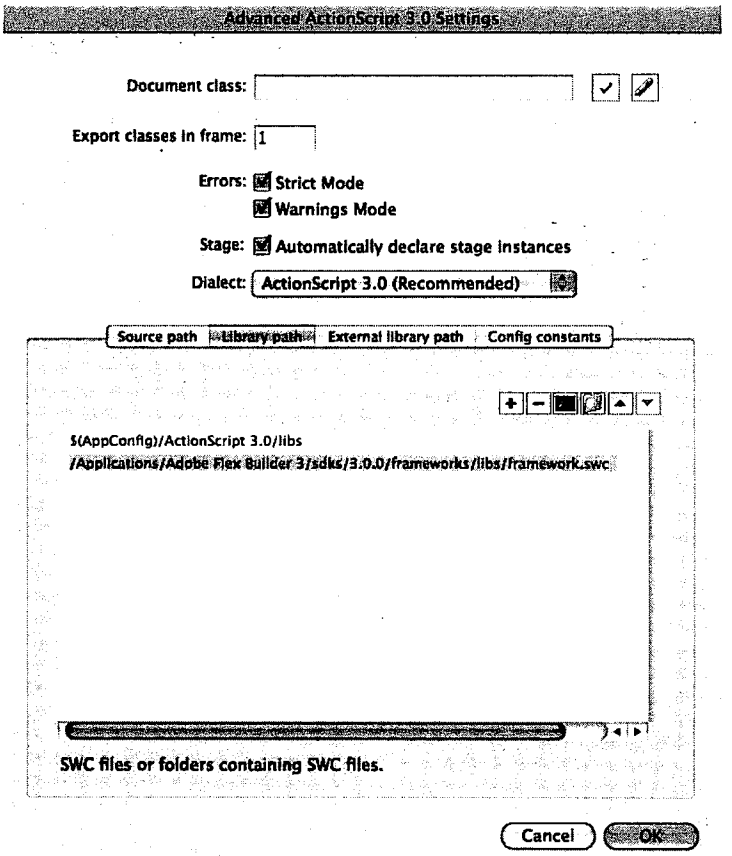


图10-4 文件添加到库路径

将这个SWC添加到你的工程并不会把整个Flex框架都放在最后的SWF中——只会加入代码中引用的类以及所有依赖类（对于Tween来说，依赖类并不多）。

Flex Tween类与Flash中的Tween类稍有不同。这里不是为补间本身提供一个对象和属性，而只提供一个监听器对象、开始和结束值以及持续时间。监听器对象需要有一些将在补间更新和结束时被调用的方法。这些方法名为onTweenUpdate和onTweenEnd。这两个方法都只有一个类型为Object的参数。

以下是mx.effects.Tween完整的构造函数：

```
new Tween(listener, startValue, endValue, duration, minFps, updateFunction, endFunction);
```

minFPS参数是可选的，它设置了每秒内补间运行的最低次数。大多情况下可以保留这个参数为默认值–1。

updateFunction和endFunction也是可选的，允许传入将要调用的函数引用而不是onTween-Update和onTweenEnd。默认情况下，如果运行有多个补间，而且其listener参数都相同，它们更

www.TopSage.com

新时都会调用同样的onTweenUpdate和onTweenEnd函数。最后这两个参数允许为各个补间建立不同的更新和结束处理程序。

下一个例子将展示如何使用Flex Tween类完成一个简单的动画。这个类在FlexTween.as文件中：

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

    import mx.effects.Tween;

    [SWF(backgroundColor=0xffffff)]
    public class FlexTween extends Sprite
    {
        private var tween:Tween;
        private var sprite:Sprite;

        public function FlexTween()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            tween = new Tween(this, 100, 800, 1000);
        }

        public function onTweenUpdate(value:Object):void
        {
            sprite.x = value as Number;
        }
        public function onTweenEnd(value:Object):void
        {
            sprite.x = value as Number;
        }
    }
}
```

这里创建了新的Tween并传入this作为监听器。这样一来，补间会查看FlexTween类本身来查找更新和结束处理程序。我们将初始值设置为100，结束值设置为800，持续时间为1 000。这里存在Flash和Flex Tween类之间的另一个区别。尽管Flash Tween类默认按帧来解释持续时间，但是也可以强制使用秒，而Flex Tween类只能使用毫秒（1 000毫秒是1秒）。

还要注意，start和end值都只是数字值，它们不与任何属性、位置或其他内容相关联。这两个值只是传入update和end处理程序，再由你利用它们做你想做的事情。

www.TopSage.com

大家网 TopSage.com

在这些处理程序函数中，这些值作为对象而不是数字传入，稍后你会了解其原因。第一次调用onTweenUpdate时，传入的值将稍大于start值。最后一次调用时，它会稍小于end值。最终调用onTweenEnd时，它的value参数将恰好等于end值。在这里，我们只是将value的类型强制转换为Number，并将其赋至sprite.x。这比Flash Tween类稍稍复杂一些，不过功能也更为强大。

10.2.1 Flex Tween类的缓动函数

类似于Flash的Tween类，Flex Tween类允许你指定一个缓动函数。不过，这里并不是在构造函数中指定，而是在实例创建后在实例上指定。与Flash类似，这些缓动函数是缓动包中的一组类的方法（easeIn、easeOut、easeInOut）。这个包就是mx.effects.easing，相应的类如下：

- Back
- Bounce
- Circular
- Cubic
- Elastic
- Exponential
- Linear
- Quadratic
- Quartic
- Quintic
- Sine

这些类比Flash类更强调数学性。Back、Bounce和Elastic类实际上与Flash中同名的类基本相同。Linear等价于Flash中的None。还剩下Circular、Cubic、Exponential、Quadratic、Quartic、Quintic和Sine。要解释这些类，最容易的方法可能就是请你访问Robert Penner的网站。在他的网站上，Robert Penner提供了一个缓动公式可视化工具。利用这个工具，你可以选择一个公式，再根据这个公式移动一个对象。这会以可视化方式展示速度随时间如何改变。这个演示的URL如下：

```
http://www.robertpenner.com/easing/easing_demo.html
```

如果不小心忘记指定缓动函数，默认缓动函数就是Sine.easeInOut。所以现在来看其他函数。创建补间之后，指定一个缓动函数：

```
tween = new Tween(this, 100, 800, 1000);
tween.easingFunction = Linear.easeIn;
```

不要忘记导入使用的缓动类。上一个例子中只是尝试了Linear，它与Flash中的None是一样的。所以这会产生一个从start到end，以恒定速度运动的补间动画。也可以尝试其他一些缓动类，看看它们的效果。下面使用了Quintic.easeInOut，这会得到一个急剧变化的加速曲线：

```
tween.easingFunction = Quintic.easeInOut;
```

这与Sine或Quadratic有所不同，它们的速度变化更有渐近性。

www.TopSage.com

10.2.2　多重补间

在Flash Tween类中，如果希望对一个对象的多个属性完成补间，必须建立多个补间。在Flex中，则可以使用同一个补间对多个值完成补间。具体做法是：并非为start和end值指定单个数字值，而是指定一个值数组。现在可以理解为什么这些参数以及传入update和end事件处理程序的值参数会有object类型：它们可以是简单的数字，也可以是数字数组。如下一个类所示，可以下载相应的FlexTweenXY.as得到：

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

    import mx.effects.Tween;
    import mx.effects.easing.Quintic;

    [SWF(backgroundColor=0xffffff)]
    public class FlexTweenXY extends Sprite
    {
        private var tween:Tween;
        private var sprite:Sprite;

        public function FlexTweenXY()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;
            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            tween = new Tween(this, [100, 100], [800, 400], 1000);
        }

        public function onTweenUpdate(value:Object):void
        {
            sprite.x = value[0];
            sprite.y = value[1];
        }

        public function onTweenEnd(value:Object):void
        {
            sprite.x = value[0];
            sprite.y = value[1];
        }
    }
}
```

在构造函数中我们为start和end值分别传入了包含两个值的数组。重申一次，它们本身并没

有任何含义，只是两个数字值而已。Tween类会在数组的所有元素之间平滑地插值，插入的值作为一个数组传入处理程序。在处理程序中可以采用我们希望的任何方式解释这些值。在这里，将取第一个元素作为*x*位置，第二个元素作为*y*位置。我希望sprite在*x*轴上从100移动到800，在*y*轴上从100移动到400。现在就是在两维上完成补间。

使用这个Tween类的另一种方法是简单地传入0和1作为start和end值，并在处理程序中自行完成计算。如下所示：

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

    import mx.effects.Tween;

    [SWF(backgroundColor=0xffffff)]
    public class FlexTweenXY extends Sprite
    {
        private var tween:Tween;
        private var sprite:Sprite;

        public function FlexTweenXY()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            tween = new Tween(this, 0, 1, 1000);
        }

        public function onTweenUpdate(value:Object):void
        {
            sprite.x = 100 + 700 * (value as Number);
            sprite.y = 100 + 300 * (value as Number);
        }

        public function onTweenEnd(value:Object):void
        {
            sprite.x = 100 + 700 * (value as Number);
            sprite.y = 100 + 300 * (value as Number);
        }
    }
}
```

value参数现在介于0到1之间。我们将这个值乘以希望它移动的总距离，并增加到初始位置。这没有直接传入值那样简单明了，但是这种方法在你需要更大的灵活性时会非常有用。例如，假

www.TopSage.com

大家网 TopSage.com

设你希望让一个对象淡入，而另一个对象淡出，就可以完成一个0到1的补间，并在处理程序中做以下工作：

```
public function onTweenUpdate(value:Object):void
{
    obj1.alpha = value as Number;
    obj2.alpha = 1.0 - value as Number;
}
```

随着value从0变到1，obj1 alpha也会从0变到1。不过obj2 alpha会从1变到0。这里只有一个补间，却有两个淡入淡出效果。

10.2.3　补间序列

遗憾的是，Flex Tween的序列却并不比Flash的序列更简单。mx.effects包确实提供了额外的类来完成并行补间和连续补间，不过它们与Flex框架本身紧密绑定，实际上主要用于基于Flex的应用（这一点与Tween不同，Tween类可以很容易地单独使用）。

使用Flex Tween类完成补间序列时，需要等待onTweenEnd处理程序触发。如果指定了另一个处理程序，还需要等待该处理程序触发。下一个类（见FlexTweenSequence.as文件）展示了这一点。这个类实际上重复了之前例子中FlashTweenSequence类的功能。其中的重点用粗体显示：

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

    import mx.effects.Tween;

    [SWF(backgroundColor=0xffffff)]
    public class FlexTweenSequence extends Sprite
    {
        private var tween1:Tween;
        private var tween2:Tween;
        private var tween3:Tween;
        private var sprite:Sprite;

        public function FlexTweenSequence()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);
            tween1 = new Tween(this, 100, 800, 3000, -1, onTween1Update, onTween1End);
        }
```

www.TopSage.com

```
        public function onTween1Update(value:Object):void
        {
            sprite.x = value as Number;
        }

        public function onTween1End(value:Object):void
        {
            sprite.x = value as Number;
            tween2 = new Tween(this, 100, 400, 3000, -1, onTween2Update, onTween2End);
        }
        public function onTween2Update(value:Object):void
        {
            sprite.y = value as Number;
        }

        public function onTween2End(value:Object):void
        {
            sprite.y = value as Number;
            tween3 = new Tween(this, 0, 360, 4000, -1, onTween3Update, onTween3End);
        }

        public function onTween3Update(value:Object):void
        {
            sprite.rotation = value as Number;
        }

        public function onTween3End(value:Object):void
        {
            sprite.rotation = value as Number;
        }
    }
}
```

第一个补间更新处理程序改变了x值，第二个改变了y值，第三个改变了sprite的rotation值。各补间结束时，会创建下一个补间。

这个代码不算太漂亮，只是为了演示一个相对简单的3段动画。一些精美的界面可能有数十个这种同步和顺序补间。当然，我没有着力去让这个代码变得更漂亮。对此可以做很多改进，你可以应用一些模式让这个代码更清晰、更简洁，甚至更可重用。接下来要考虑如何使用一个补间引擎，这正是下面要讨论的内容。

10.3 补间引擎

现在，我认为你已经知道了为什么补间引擎很棒：利用补间引擎，在多个对象上完成多个属性的复杂多重补间时，编码工作会变得更可管理。再次说明，这里只是选择了一些不同的引擎做简要介绍，并不是因为它们是我最喜欢的引擎，也不是说它们是“最好的引擎”，而只是因为它们非常流行，并且展示了解决问题的各种不同方法。其中一些引擎主要面向于开发人员，需要编写一些代码才能实现功能。另外一些则更面向设计人员，更容易使用，但是一些编码约定会让严

www.TopSage.com

格的开发人员不快：无类型的参数、泛型对象、“魔法字符串”、代替事件的函数回调，等等。最终使用哪个引擎取决于哪一个最适应你的工作流程和你的团队，同时能提供你需要的功能。

再次指出，我们将介绍以下引擎：

- Tweener
- TweenLite/TweenGroup
- KitchenSync
- gTween

下面先来讨论Tweener。

10.4 Tweener

Tweener补间引擎近年来相当流行。它最早是一个ActionScript 2.0引擎，而后移植到ActionScript 3.0。这两个版本现在都可用。实际上，Tweener至少一部分基于一个较早的ActionScript 1.0引擎，名为MC Tween，这个引擎于2003年发布。Tweener原先由Zeh Fernando编写（http://zeh-fernando.com），不过很多其他人员也做出了很多贡献。

要安装Tweener，先从Google代码库：http://code.google.com/p/tweener/下载源文件。这个包的顶层文件夹名为caurina。把这个文件夹放在工程中与其他源文件相同的一个文件夹下（或者放在你的开发环境能找到的一个位置）。这个引擎的主类是caurina.transitions.Tweener。

Tweener语法相当简单，很容易熟悉。不同于Adobe Tween类，不必建立主类Tweener的一个实例。实际上，需要调用Tweener上的静态方法。主方法是addTween，如下所示：

```
Tweener.addTween(target, tweeningParameters);
```

target和tweeningParameters的类型都是Object。target是希望影响的对象——通常是一个动画片段、sprite或另外某个显示对象（不过这也可以是任何对象，只要它至少有一个可以改变的数字公共属性）。tweeningParameters参数是一个通用对象，包含多个属性，允许指定补间值、缓动函数、持续时间等。所以可以完成如下调用，这里会把名为sprite的对象在3秒内移动到x为800的位置上：

```
Tweener.addTween(sprite, {x:800, time:3});
```

或者做以下调用，这会把sprite移动到x, y坐标分为（800, 400）的位置：

```
Tweener.addTween(sprite, {x:800, y:400, time:3});
```

显然，这个过程比任何Adobe Tween类都简单得多。增加到tweeningParameters对象的所有属性将在目标对象上完成补间。注意这里没有初始值。初始值假设为目标对象各个补间属性的当前值。还有重要的一点，tweeningParameters中指定的值是绝对目标值。换句话说，sprite并非在x轴上移动800像素，在y轴上移动400像素。相反，它会移动到（800, 400）的位置。

通用对象tweeningParameters是Tweener的一大优点，也是很多比较严格的开发人员不喜欢它的原因之一。一方面，它能提供极大的灵活性，允许对任意类型对象上的任何公共数字属性完成补间。例如，尽管Tweener是在Flash 10发布很早以前编写的，但是完全可以用它对Flash 10中一

www.TopSage.com

大家网 TopSage.com

个sprite的3D属性完成补间而不产生任何问题。

```
Tweener.addTween(sprite, {x:800, z:800, rotationX:180, rotationY:270, time:3});
```

另一方面，由于使用了一个通用对象，而且可以把任意类型的任意属性放在通用对象上，因此有可能做出各种各样毫无意义的疯狂之举。编译器会顺利地编译，只是在抛出运行时错误时才会注意到有问题（如果已经安装了debug player）。例如，你可能会尝试以下补间：

```
Tweener.addTween(sprite, {dog:800, time:3});
```

显然，dog不是Sprite类的一个属性，放在这里根本不合逻辑。当然在运行这个SWF时会出问题，但是它确实能成功编译而不产生任何警告。不仅如此，还可以传入错误的类型，如下：

```
Tweener.addTween(sprite, {x:"dog", time:3});
```

现在我们都知道x应当是一个数字，但是如果出于某种原因这里放了一个字符串，则会丧失所有编译时类型检查，甚至在运行时也不会报错。sprite只是从屏幕上消失，字符串“dog”解析为哪个数值，就会把sprite放到相应的x位置上。另一方面，Tweener很聪明，它可以将格式化为字符串的数字转换为数字。所以下面的代码能正常运行：

```
Tweener.addTween(sprite, {x:"800", time:3});
```

10.4.1 Tweener 中的缓动函数

缓动函数也在tweeningParameters对象中指定。不足为奇，这些缓动函数也是基于Robert Penner的公式。不过，类和方法本身是隐藏的，只是要传入一个描述所需缓动类型的字符串，例如：

```
Tweener.addTween(sprite, {x:800, time:3, transition:"easeInOutCubic"});
```

关于各种变换（transition）字符串的完整列表请参见Tweener文档。

另外，对于设计人员来说，现在只需要记住一个名字而不需要去记类路径、类名和方法名，所以这可能容易得多。不过，如果不小心误拼为"easeInOutCubix"会怎么样呢？它仍能顺利地编译，甚至能很好地运行，不过缓动公式会还原为默认值"easeOutExpo"，而你可能一直都没有注意到，直到几个星期以后有客户指出界面上有些地方“看上去”没有原先好，这才有可能发现问题。

我并不是说因为存在这样一些问题所以不应该使用Tweener，我只是把问题指出来，提醒你在评价不同引擎时要知道这样一些问题。需要指出，Tweener并不是使用这种语法的唯一引擎。

10.4.2 Tweener 中的多重补间

Tweener中的补间要逐对象地增加。所以如果希望对多个对象完成补间，需要建立多个补间：

```
Tweener.addTween(spriteA, {x:"800", time:3});
Tweener.addTween(spriteB, {x:"300", time:3});
```

我认为，一方面我们能够很容易地对一个对象上的多个属性完成补间，另一方面对多个对象完成补间时又需要建立多个补间，这两方面有些唱反调。

10.4.3 Tweener 中的序列

使用Tweener的另一个好处是它让你更容易地创建序列。实际上要在一个补间完成后运行另

www.TopSage.com

一个补间有两种方法。首先，可以在tweeningParams对象中建立一个onComplete回调。这只是一个函数引用，补间完成时会调用这个函数。如下所示：

```
Tweener.addTween(spriteA, {x:"800", time:3, onComplete:tweenEnd});
```

在这里，tweenEnd是补间完成时要调用的函数的引用。所以一个补间完成时，往往希望增加另一个补间。下一个演示就完成了这个工作，可以下载相应的TweenerSequence.as文件：

```
package {
    import caurina.transitions.Tweener;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class TweenerSequence extends Sprite
    {
        private var sprite:Sprite;

        public function TweenerSequence()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            Tweener.addTween(sprite, {x:800, time:3, transition:"easeInOutCubic",
                                onComplete:tween1End});
        }
        private function tween1End():void
        {
            Tweener.addTween(sprite, {y:400, time:3, transition:"easeInOutCubic",
                                onComplete:tween2End});
        }

        private function tween2End():void
        {
            Tweener.addTween(sprite, {rotation:360, time:4, transition:"easeInOutCubic"});
        }
    }
}
```

完成第一个补间时会调用tween1End，它会建立另一个补间。这个补间完成时将调用tween2End，它又会创建最后一个补间。这比Adobe Tween类中建立序列要容易一些，不过Tweener还提供了一种更容易的方法：delay属性。

www.TopSage.com

还可以为tweeningParameters增加另一个属性：delay。这是一个时间值（秒数），Tweener在执行该补间之前要等待这个属性指定的秒数。所以可以建立下例所示的代码（可以下载TweenerDelay.as）：

```
package {
    import caurina.transitions.Tweener;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class TweenerDelay extends Sprite
    {
        private var sprite:Sprite;

        public function TweenerDelay()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);
            Tweener.addTween(sprite, {x:800, time:3, transition:"easeInOutCubic"});
            Tweener.addTween(sprite, {y:400, time:3, transition:"easeInOutCubic", delay:3});
            Tweener.addTween(sprite, {rotation:360, time:4, transition:"easeInOutCubic", delay:6});
        }
    }
}
```

这里只需直接建立你的所有补间。第一个补间需要3秒时间，所以告诉第二个补间开始之前要等待3秒。类似地，要告诉最后一个补间等待6秒。其结果是所有3个补间会完全按序列开展工作。

使用delay要注意这样一种情况，有可能用一个delay对同一个对象的同一个属性完成两次补间。只要第二个补间的延迟时间至少与第一个补间的时间一样长，那么两个补间都能完成。在这里，sprite会移动到x轴800的位置，然后移动到100的位置：

```
Tweener.addTween(sprite, {x:800, time:3});
Tweener.addTween(sprite, {x:100, time:3, delay:3});
```

不过，如果第二个补间的延迟时间不够长，以至于会在第一个补间完成之前“切入”，那么第一个补间将被第二个补间完全覆盖，根本不会做任何工作。

```
Tweener.addTween(sprite, {x:800, time:3});
Tweener.addTween(sprite, {x:100, time:3, delay:2.5});
```

在这里，sprite有2.5秒不做任何移动，然后再移动到x轴上100的位置。

www.TopSage.com

根据上面的介绍，你应该对Tweener是什么已经有了充分的认识。可以看到，它提供了一些很好的快捷途径，可以用最少的代码并使用易于理解的语法完成复杂的补间。它还提供了其他很多有用的特性——只需浏览前面所列网站上的文档就可以了解。一些高级开发人员可能对数据类型的松散性有些不满，不过对于很多人来说，都认为Tweener是对Adobe类的一个改进。

10.5　TweenLite/TweenGroup

下面这个包由Jack Doyle(www.greensock.com)编写的一组类组成。可以从http://blog.greensock.com/tweenliteas3/得到相应代码，类似于Tweener，只需将顶层文件夹gs增加到你的工程目录，或者放在编译器能够找到的任何其他位置。包中顶层的主类为gs.TweenLite。

如果已经理解10.4节的内容或者对Tweener已经很熟悉，TweenLite对你来说应该会相当容易。按照TweenLite作者的说法，其目标就是让TweenLite比其他补间引擎（如Tweener）更紧凑、更快速，而且更高效。本章还会介绍TweenGroup，这是一个专门用于建立并行补间、补间序列和其他补间协调组的类。

类似于Tweener，可以通过调用TweenLite类的一个静态方法来创建补间。对于TweenLite类，这个静态方法就是TweenLite.to:

```
TweenLite.to(sprite, 3, {x:800});
```

在这里，传入一个目标对象作为第一个参数，并传入补间的持续时间作为第二个参数。第三个参数是variables，类似于Tweener的tweeningParameters对象，这是一个通用对象，其中包含希望补间的属性。所以前例会把名为sprite的对象在3秒内移动到x为800的位置。

调用这个to方法将返回TweenLite类的一个实例。以后可以使用这个实例进一步调整补间：

```
var tween1:TweenLite = TweenLite.to(sprite, 3, {x:800});
```

不过还可以采用更直观的做法，直接建立该类的一个新实例。其构造函数与to方法的语法相同：

```
var tween1:TweenLite = new TweenLite(sprite, 3, {x:800});
```

下面把以上内容综合在一起，建立一个可以真正运行的类（可以从TweenLiteDemo1.as文件得到）：

```
package {

    import flash.display.*;

    import gs.TweenLite;

    [SWF(backgroundColor=0xffffff)]
    public class TweenLiteDemo1 extends Sprite
    {
        private var sprite:Sprite;
        private var tween:TweenLite;

        public function TweenLiteDemo1()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
```

www.TopSage.com

```
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            tween = new TweenLite(sprite, 3, {x:800});
        }
    }
}
```

这里的重要代码行用粗体显示。只需将多个属性增加到variables对象就可以很容易地完成多个属性的补间：

```
tween = new TweenLite(sprite, 3, {x:800, y:400, rotation:360});
```

类似于Tweener，这同样为传入variables对象中不存在的属性或不正确的数据类型大开方便之门。所以一定要当心。

10.5.1 TweenLite 中的缓动函数

在这个方面，TweenLite更面向于程序员。TweenLite对于缓动函数没有使用魔法字符串，类似于Adobe Tween类，TweenLite确实使用了各个缓动类的方法名。

毫不奇怪，这些缓动类都直接基于Robert Penner的缓动公式（你是不是对他已经佩服得五体投地？），它们都放在gs.easing包中：

- Back
- Bounce
- Circ
- Cubic
- Elastic
- Expo
- Linear
- Quad
- Quart
- Quint
- Sine
- Strong

每个类都有3个方法：easeIn、easeOut和easeInOut。类似于Adobe Tween类，要指定类和方法来选择缓动类型。这放在variables对象中，并赋至ease属性：

```
tween = new TweenLite(sprite, 3, {x:800, y:400, rotation:360, ease:Elastic.easeInOut});
```


www.TopSage.com

一定要确保已经导入所用的缓动类或者导入整个gs.easing.*包。

或者，由于你现在有了补间的一个引用，也可以对这个引用指定缓动类型：

```
tween = new TweenLite(sprite, 3, {x:800, y:400, rotation:360}); tween.ease = Elastic.easeInOut;
```

10.5.2 TweenLite的多重补间

再次说明，对多个对象补间通常意味着建立多个补间：

```
tween1 = new TweenLite(spriteA, 3, {x:800});
tween2 = new TweenLite(spriteB, 3, {x:100});
```

这会将一个sprite在*x*轴上补间到800，并同时将另一个sprite补间到100。

不过，可以使用TweenGroup以一种更为简洁的方式做到这一点。TweenGroup类也放在gs包中，安装TweenLite时就应该已经安装。这是一个功能非常强大的类，可以采用多种方式使用。如果对多个对象的相同属性进行补间，可以利用TweenGroup.allTo静态方法对它们同时完成补间。这个工作类似于TweenLite.to方法，不过它接受一个对象数组作为第一个参数，而不只是一个对象：

```
TweenGroup.allTo([s1, s2, s3, s4], 3, {x:800});
```

在这里，s1、s2、s3和s4（假设有4个不同的sprite）都要在下一个3秒内补间到x为800的位置。以下给出这个完整的类，以便了解它的具体运行情况（可以下载TweenGroupDemo1.as得到这个类）：

```
package {

    import flash.display.*;

    import gs.TweenGroup;

    [SWF(backgroundColor=0xffffff)]
    public class TweenGroupDemo1 extends Sprite
    {
        private var sprite:Sprite;
        private var group:TweenGroup;

        public function TweenGroupDemo1()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            var s1:Sprite = makeSprite(100, 100, 0xff0000);
            var s2:Sprite = makeSprite(100, 200, 0x00ff00);
            var s3:Sprite = makeSprite(100, 300, 0x0000ff);
            var s4:Sprite = makeSprite(100, 400, 0xffff00);

            TweenGroup.allTo([s1, s2, s3, s4], 3, {x:800});
        }
        private function makeSprite(xpos:Number, ypos:Number, color:uint):Sprite
        {
            var s:Sprite = new Sprite();
            s.graphics.beginFill(color);
            s.graphics.drawRect(-50, -25, 100, 50);
```

www.TopSage.com

TopSage.com

```
            s.graphics.endFill();
            s.x = xpos;
            s.y = ypos;
            addChild(s);
            return s;
        }
    }
}
```

这里增加了一个makeSprite方法，从而可以很容易地创建任意多个sprite，然后用一行代码指定其位置和颜色。我们建立了4个sprite，并垂直放置。然后把它们传入TweenGroup.allTo来让它们开始移动。

注意，TweenGroup.allTo返回TweenGroup的一个实例，可以跟踪这个对象并根据需要进行修改：

```
var tg:TweenGroup = TweenGroup.allTo([s1, s2, s3, s4], 3, {x:800});
```

10.5.3 TweenLite/TweenGroup 中的序列

我认为正是TweenLite和TweenGroup中的序列功能才真正使这个包如此光彩夺目。

与Tweener类似，TweenLite有一个delay属性，可以用来延迟一个补间。不过这里有一个重要区别，正是这一点提供了更大的灵活性。首先，再来看只有一个sprite的例子，我们如下建立了3个补间的序列（TweenLiteSequence1.as）：

```
package {

    import flash.display.*;

    import gs.TweenLite;
    import gs.easing.Elastic;

    [SWF(backgroundColor=0xffffff)]
    public class TweenLiteSequence1 extends Sprite
    {
        private var sprite:Sprite;
        private var tween1:TweenLite;
        private var tween2:TweenLite;
        private var tween3:TweenLite;

        public function TweenLiteSequence1()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);
```


www.TopSage.com

```
            tween1 = new TweenLite(sprite, 3, {x:800});
            tween2 = new TweenLite(sprite, 3, {y:400, delay:3});
            tween3 = new TweenLite(sprite, 3, {rotation:360, delay:6});
        }
    }
}
```

运行这个代码时，在6秒的时间内什么也不会发生，然后最后一个补间开始运行。在TweenLite中，一个对象上的任何补间默认状态下都会覆盖之前的所有补间——即使在对不同的属性完成补间而且两个补间之间有足够的延迟也是如此。应该记得，Tweener有所不同，只有当两个补间有共同的属性而且时间存在重叠时Tweener才会发生覆盖。

幸运的是，可以利用variables对象的overwrite属性控制这种行为。这个属性取值为0到4，分别有以下含义。

- 0：无（NONE）模式。不覆盖任何补间。
- 1：全部（ALL）模式。所有补间都会被同一个对象上的任何新补间所覆盖。
- 2：自动（AUTO）模式。这种模式类似于Tweener的覆盖逻辑。只有相同属性上存在重叠的补间才会被覆盖。
- 3：并发（CONCURRENT）模式。覆盖同一个对象上的所有重叠补间，而不论是否是对同一个属性完成补间。

遗憾的是，这个重叠概念看起来并没有考虑到延迟。所以即使在上一个例子中将tween2和tween3的overwrite设置为2或3，其结果仍是只运行最后一个补间，因为前两个补间将被覆盖。

所以，要修正上一个文件，需要将overwrite属性设置为0，这样才能确保所有补间都不会被覆盖：

```
tween1 = new TweenLite(sprite, 3, {x:800});
tween2 = new TweenLite(sprite, 3, {y:400, delay:3, overwrite:0});
tween3 = new TweenLite(sprite, 3, {rotation:360, delay:6, overwrite:0});
```

注意，TweenLite包中还有一个OverwriteManager类，其中包含了这些值作为静态属性（OverwriteManager.NONE，OverwriteManager.ALL，等等）。尽管这些静态属性比较长，但利用这些静态属性可以提供编译时检查，而且能清楚表明你的意图。

建立序列的另一种方法是在开始下一个补间之前等待各补间完成，这与前例TweenerSequence中的工作几乎完全相同：

```
package {

    import flash.display.*;

    import gs.TweenLite;
    import gs.easing.Elastic;

    [SWF(backgroundColor=0xffffff)]
    public class TweenLiteSequence2 extends Sprite
    {
        private var sprite:Sprite;
        private var tween1:TweenLite;
        private var tween2:TweenLite;
```

www.TopSage.com

```
        private var tween3:TweenLite;

        public function TweenLiteSequence2()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            tween1 = new TweenLite(sprite, 3, {x:800, onComplete:onTween1End});
        }

        private function onTween1End():void
        {
            tween2 = new TweenLite(sprite, 3, {y:400, onComplete:onTween2End});
        }

        private function onTween2End():void
        {
            tween3 = new TweenLite(sprite, 3, {rotation:360});
        }
    }
}
```

这里创建了第一个补间并设置onComplete回调。这个回调触发时创建下一个补间，并设置其onComplete回调。在最后一个回调中，我们创建了最后一个补间。当然，这比使用delay要复杂一些，不过如果需要在各个补间完成时完成另外一些动作，这种做法的功能更强大。例如，一个导航元素结束在场景上的补间时，可能需要利用一个回调来自行填充。

最后，TweenGroup还提供了一些功能更强大的方法实现补间同步和序列。第一种方法是使用TweenGroup对象的align属性，这个属性可以设置为一组不同的模式，并影响一个组中的不同补间如何共同完成工作。对于创建序列来说，最有用的模式是ALIGN_SEQUENCE。再来看本章较早前创建的TweenGroupDemo1类，我们可以轻而易举地把这个组变为一个序列（这个例子的相应文件是TweenGroupDemo2.as）：

```
package {

    import flash.display.*;
    import gs.TweenGroup;

    [SWF(backgroundColor=0xffffff)]
    public class TweenGroupDemo2 extends Sprite
    {
        private var sprite:Sprite;
        private var group:TweenGroup;
```


www.TopSage.com

大家网 TopSage.com

```
        public function TweenGroupDemo2()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            var s1:Sprite = makeSprite(100, 100, 0xff0000);
            var s2:Sprite = makeSprite(100, 200, 0x00ff00);
            var s3:Sprite = makeSprite(100, 300, 0x0000ff);
            var s4:Sprite = makeSprite(100, 400, 0xffff00);

            var tg:TweenGroup = TweenGroup.allTo([s1, s2, s3, s4], 1, {x:800});
            tg.align = TweenGroup.ALIGN_SEQUENCE;;
        }

        private function makeSprite(xpos:Number, ypos:Number, color:uint):Sprite
        {
            var s:Sprite = new Sprite();
            s.graphics.beginFill(color);
            s.graphics.drawRect(-50, -25, 100, 50);
            s.graphics.endFill();
            s.x = xpos;
            s.y = ypos;
            addChild(s);
            return s;
        }
    }
}
```

在这里，我们将TweenGroup保存在变量tg中。然后设置组的align属性为TweenGroup.ALIGN_SEQUENCE。这会导致这个组中的各个补间仅在前一个补间完成时才发生。运行这个组时，可以看到作为一组的4个sprite会一个接一个依次运行而不是同时在场景上移动。

如果只是多个对象在一个序列中完成补间，这样就很好，不过，如果希望一个对象通过一个序列采用多种不同的方式进行补间该怎么做呢（例如，先在*x*轴上移动，再在*y*轴上移动，然后旋转）？

可以为所有这些动作建立补间，然后建立一个组让它们依次运行。如下面的例子所示，这个例子可以通过下载TweenGroupDemo3.as文件得到：

```
package {

    import flash.display.*;

    import gs.TweenGroup;
    import gs.TweenLite;

    [SWF(backgroundColor=0xffffff)]
    public class TweenGroupDemo3 extends Sprite
    {
        private var sprite:Sprite;
        private var tween1:TweenLite;
        private var tween2:TweenLite;
        private var tween3:TweenLite;
```

```
        public function TweenGroupDemo3()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            tween1 = new TweenLite(sprite, 3, {x:800});
            tween2 = new TweenLite(sprite, 3, {y:400, overwrite:0});
            tween3 = new TweenLite(sprite, 3, {rotation:360, overwrite:0});
            var tg:TweenGroup = new TweenGroup([tween1, tween2, tween3]);
            tg.align = TweenGroup.ALIGN_SEQUENCE;
        }
    }
}
```

乍一看这似乎比直接使用一个delay复杂一些，因为你要创建一个额外的对象，还要在这个对象上设置一个属性。不过这种做法更为灵活。如果使用delay，必须跟踪一个序列中所有补间的持续时间，并确保将各个delay累加到之前所有补间的持续时间。这很容易犯错！当然，可以对持续时间使用变量，并把这些变量累加在一起得到一个delay，但是一旦这样做，就会像使用组一样复杂。

这里所讨论的只是TweenLite和TweenGroup提供的功能的冰山一角。我们还没有讨论TweenLite的“大哥”TweenMax，它又增加了大量额外特性。如果要完成大量复杂的序列补间，这些类会助你一臂之力。

10.6 KitchenSync

下一个要讨论的补间引擎是KitchenSync，这个引擎由Mims Wright编写。可以在Google代码网站（http://code.google.com/p/kitchensynclib/）下载这个工程，可以下载一个源目录包或者SWC库。相信你应该知道如何向你的构建路径添加源目录和/或SWC库。如果你不知道，可以查看刚才提到的KitchenSync网站上的安装提示页面。

这个包的主类是org.as3lib.kitchensync.KitchenSync。作为所有KitchenSync工程主文档类的第一个动作，需要调用KitchenSync.initialize(this)。这里使用this会传入文档类本身的一个引用，但是实际上也可以传入任何显示对象的引用。KitchenSync使用传入的显示对象来监听enterFrame事件，它利用这些事件更新补间或基于时间的动作的进度。

初始化引擎后，可以建立一个补间，这是org.as3lib.kitchensync.action.KSTween的一个实例。构造函数如下所示：

```
new KSTween(target, property, startValue, endValue, duration);
```


www.TopSage.com

其中的参数的意义你都可以猜出个大概，不过还是特别说明一下：target是想要完成补间的对象；property是一个字符串，表示想要完成补间的属性，如"x"，"y"，"rotation"；start和end值是数字值，指定属性的起始和结束位置；duration指定补间持续多长时间。

关于duration属性有一点很有意思，它可以取一个数也可以取一个字符串。如果传入一个数字，则解释为毫秒（所以3 000将是3秒）。不过，也可以传入一个字符串，如"3sec"，表示补间要持续的秒数。现在了解到这一点就足够了，不过还可以查看文档来了解可以使用哪些字符串。

KSTween与我们目前讨论过的所有其他补间类还有一点不同，它不是一创建就开始运行。需要在KSTween实例上调用start方法，否则要完成补间的对象只是原地不动什么也不做。

好了，可以开始了。下面建立一个KitchenSync补间（下面的例子可以下载KitchenSyncDemo1.as文件得到）：

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

    import org.as3lib.kitchensync.KitchenSync;
    import org.as3lib.kitchensync.action.KSTween;

    [SWF(backgroundColor=0xffffff)]
    public class KitchenSyncDemo1 extends Sprite
    {
        private var sprite:Sprite;

        public function KitchenSyncDemo1()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            KitchenSync.initialize(this);
            var tween:KSTween = new KSTween(sprite, "x", 100, 800, 3000);
            tween.start();
        }
    }
}
```

重要的代码行用粗体显示。这里首先初始化引擎，创建一个新的补间（在3秒内将sprit从x轴上100的位置移动到800的位置），并开始补间。非常神奇。

10.6.1　KitchenSync 中的缓动函数

猜猜看KitchenSync使用的缓动公式是谁写的？当然是Robert Penner！没错，不过，除了你已

大家网 TopSage.com

经多次看到的标准函数外，Mims还加入了他自己的一些新类：

- Oscillate
- Sextic
- Random
- Stepped

你可以尝试使用这些类，不过现在来看如何指定一个缓动函数。基本说来，只需把缓动函数作为最后一个参数传入KSTween构造函数。不过，这里还有一个参数，即倒数第2个可选的参数delay。如果没有使用delay，那么只需传入0，然后是缓动函数。缓动类在org.as3lib.kitchensync.easing包中，类似于其他实现，这些缓动类分别有一个easeIn、easeOut和easeInOut方法。所以要使用Quartic.easeInOut函数，需要以下代码：

```
var tween:KSTween = new KSTween(sprite, "x", 100, 800, 3000, 0, Quartic.easeInOut);
```

10.6.2 利用 KitchenSync 对多个对象/属性补间

与Tweener或TweenLite不同，它们允许利用一个补间对多个属性完成补间，使用KitchenSync时则必须为各个属性分别创建一个新的补间。创建新补间有一个快捷途径：cloneWithTarget。取一个现有的补间，对它调用cloneWithTarget，并传入另一个（或同一个）目标对象以及一个属性字符串。所以可以按如下方式创建补间：

```
var tween1:KSTween = new KSTween(sprite, "x", 100, 800, 3000, 0, Quartic.easeInOut);
var tween2:KSTween = tween1.cloneWithTarget(sprite, "y");
tween1.start();
tween2.start();
```

但是这样做也会复制开始和结束值。在绝大多数情况下，对同一个对象的多个属性进行补间时都会使用不同的值。所以这个复制方法在这种情况下看起来作用很有限，可能更好的做法是直接创建一个新的补间：

```
var tween1:KSTween = new KSTween(sprite, "x", 100, 800, 3000, 0, Quartic.easeInOut);
var tween2:KSTween = new KSTween(sprite, "y", 100, 400, 3000, 0, Quartic.easeInOut);
tween1.start();
tween2.start();
```

类似地，需要创建多个补间来完成多个对象的补间。在这种情况下cloneWithTarget可能更有用。请看下面的例子（KitchenSyncMultiple.as）：

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

    import org.as3lib.kitchensync.KitchenSync;
    import org.as3lib.kitchensync.action.KSTween;
    import org.as3lib.kitchensync.easing.Quartic;

    [SWF(backgroundColor=0xffffff)]
    public class KitchenSyncMultiple extends Sprite
```


www.TopSage.com

```
    {
        private var sprite:Sprite;
        public function KitchenSyncMultiple()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            var s1:Sprite = makeSprite(100, 100, 0xff0000);
            var s2:Sprite = makeSprite(100, 200, 0x00ff00);
            var s3:Sprite = makeSprite(100, 300, 0x0000ff);
            var s4:Sprite = makeSprite(100, 400, 0xffff00);

            KitchenSync.initialize(this);
            var tween1:KSTween = new KSTween(s1, "x", 100, 800, 3000, 0, Quartic.easeInOut);
            var tween2:KSTween = tween1.cloneWithTarget(s2, "x");
            var tween3:KSTween = tween1.cloneWithTarget(s3, "x");
            var tween4:KSTween = tween1.cloneWithTarget(s4, "x");
            tween1.start();
            tween2.start();
            tween3.start();
            tween4.start();
        }

        private function makeSprite(xpos:Number, ypos:Number, color:uint):Sprite
        {
            var s:Sprite = new Sprite();
            s.graphics.beginFill(color);
            s.graphics.drawRect(-50, -25, 100, 50);
            s.graphics.endFill();
            s.x = xpos;
            s.y = ypos;
            addChild(s);
            return s;
        }
    }
}
```

每个额外的补间都是第一个补间的副本，只是有不同的目标。

注意，必须分别开始各个补间。KitchenSync更强大的特性是其动作组，这类似于TweenLite包中的TweenGroup类。使用一个动作组时，可以同步多个补间的动作。有多种类型的动作组。如果要同时开始多个补间，可以使用org.as3lib.kitchensync.action.KSParallelGroup类。将希望成组的所有补间传入构造函数，然后在组上调用start，如下所示：

```
var tween1:KSTween = new KSTween(s1, "x", 100, 800, 3000, 0, Quartic.easeInOut);
var tween2:KSTween = tween1.cloneWithTarget(s2, "x");
var tween3:KSTween = tween1.cloneWithTarget(s3, "x");
var tween4:KSTween = tween1.cloneWithTarget(s4, "x");

var pg:KSParallelGroup = new KSParallelGroup(tween1, tween2, tween3, tween4);
pg.start();
```

再次说明，乍一看这比直接调用4次start还要复杂。不过，KitchenSync的强大功能很大一部

www.TopSage.com

分都来源于动作组。创建一个动作组时（如KSParallelGroup），不仅可以传入KSTween的实例，还可以传入KSParallelGroup或任何其他动作组的实例。这样一来，可以创建复杂的嵌套序列化动画，只需对组调用start就可以在任何时间重放。在10.7节你将了解有关的更多内容。

10.6.3 KitchenSync 中的补间序列

类似于目前为止介绍过的其他引擎，可以使用delay创建序列，或者在开始下一个补间之前通过监听前一个补间的完成来创建序列。我不再重复这个内容，因为这与其他引擎并无不同。更有意思的是使用另一个动作组KSSequenceGroup来创建序列。

使用KSSequenceGroup与KSParallelGroup同样简单。下面的类（可以下载KitchenSyncSequence.as得到）展示了这一点：

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

    import org.as3lib.kitchensync.KitchenSync;
    import org.as3lib.kitchensync.action.KSSequenceGroup;
    import org.as3lib.kitchensync.action.KSTween;
    import org.as3lib.kitchensync.easing.Quartic;

    [SWF(backgroundColor=0xffffff)]
    public class KitchenSyncSequence extends Sprite
    {
        private var sprite:Sprite;

        public function KitchenSyncSequence()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            KitchenSync.initialize(this);
            var tween1:KSTween = new KSTween(sprite, "x", 100, 800, 3000, 0, Quartic.easeInOut);
            var tween2:KSTween = new KSTween(sprite, "y", 100, 400, 3000, 0, Quartic.easeInOut);
            var tween3:KSTween = new KSTween(sprite, "rotation", 0, 360, 3000, 0,
                                             Quartic.easeInOut);
            var sg:KSSequenceGroup = new KSSequenceGroup(tween1, tween2, tween3);
            sg.start();
        }
    }
}
```


www.TopSage.com

大家网 TopSage.com

这里为3个不同属性创建了3个补间，并将其传入一个KSSequenceGroup实例。开始这个组，所有补间都会依序运行。

要记住，可以向一个新的并行组增加一个序列组，使该序列与其他补间同时运行，或者也可以向序列组增加一个并行组。这会得到极其强大的功能。

KitchenSync可能比目前介绍过的两个包更面向开发人员。利用Tweener，只需一个类就可以完成几乎所有工作，TweenLite进一步提供了更多功能，KitchenSync则是一个包含数十个类、包和接口的大型框架。它需要更长的学习曲线，不过可能很适合你的工作流，而且相当灵活，足以完成你要做的任何工作。

10.7　gTween

最后一个引擎是Grant Skinner编写的gTween。Grant是一个著名的Flash开发人员，也是我的好朋友（我在这本书中曾经多次提到他的工作）。写这本书时，gTween刚刚发布一个公共beta版，所以等你读到这本书时它的应用编程接口（API）和工作流可能已经有所变化，不过我认为还是有必要对这个引擎做一个简要介绍。

可以从以下地址（http://gskinner.com/libraries/gtween/）下载gTween。

整个gTween都在一个类中：com.gskinner.motion.Gtween。这个类的构造函数如下：

```
new GTween(target, duration, properties, tweenProperties);
```

与之前看到的其他引擎类似，target是将要完成补间的对象，duration是补间持续的时间。properties参数包含对象中将要完成补间的属性。这是一个通用对象，其工作方式与Tweener和TweenLite中完全相同。不过，gTween中还有一个参数tweenProperties，这个参数允许定制补间的其他方面，如delay或缓动函数。

现在可以开始了。可以下载GTweenDemo.as得到下面的例子：

```
package {
    import com.gskinner.motion.GTween;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    [SWF(backgroundColor=0xffffff)]
    public class GTweenDemo extends Sprite
    {
        private var sprite:Sprite;

        public function GTweenDemo()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
```

www.TopSage.com

```
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            var tween:GTween = new GTween(sprite, 3, {x:800});
        }
    }
}
```

非常简单。如果你能理解本章给出的其他引擎介绍，这里实际上不需要再多做解释。类似于Tweener和TweenLite，只需将多个属性增加到properties对象就可以对一个对象上的多个属性完成补间：

```
var tween:GTween = new GTween(sprite, 3, {x:800, y:400});
```

10.7.1 gTween 中的缓动函数

我推崇gTween的一点是，它不会从头再做已经有的东西。完全可以想到，这个引擎也使用了Robert Penner的公式。不过其他引擎的做法是把所有这些类和函数复制到新的（实际上相同的）类和函数中，而gTween有所不同，它只使用Flash和Flex框架提供的类：对于Flash就是fl.transitions.easing包中的类，对于Flex框架则是mx.effects.easing包中的类。

将缓动函数传入缓动的ease属性，可以通过构造函数的tweeningProperties参数传入，或者在补间实例创建后直接传入补间实例。下面两个例子都将ease函数设置为mx.effects.easing.Circular.easeInOut（确保你已经导入了所用的类）。下面是第一个例子：

```
var tween:GTween = new GTween(sprite, 3, {x:800, y:400}, {ease:Circular.easeInOut});
```

以下是第二个例子：

```
var tween:GTween = new GTween(sprite, 3, {x:800, y:400});
tween.ease = Circular.easeInOut;
```

10.7.2 利用 gTween 完成多个对象补间

与大多数补间引擎一样，使用多个对象意味着创建多个补间。GTween类包含一个clone方法，让创建多个补间变得更为容易。可以向clone方法传入一个新的目标，这样就会针对这个新的目标对象复制补间。下一个类（文件GTweenMulti.as）展示了这一点：

```
package {
    import com.gskinner.motion.GTween;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    import mx.effects.easing.Quadratic;

    [SWF(backgroundColor=0xffffff)]
    public class GTweenMulti extends Sprite
    {
```


www.TopSage.com

```
        public function GTweenMulti()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            var s1:Sprite = makeSprite(100, 100, 0xff0000);
            var s2:Sprite = makeSprite(100, 200, 0x00ff00);
            var s3:Sprite = makeSprite(100, 300, 0x0000ff);
            var s4:Sprite = makeSprite(100, 400, 0xffff00);

            var tween1:GTween = new GTween(s1, 3, {x:800}, {ease:Quadratic.easeInOut});
            var tween2:GTween = tween1.clone(s2);
            var tween3:GTween = tween1.clone(s3);
            var tween4:GTween = tween1.clone(s4);
        }

        private function makeSprite(xpos:Number, ypos:Number, color:uint):Sprite
        {
            var s:Sprite = new Sprite();
            s.graphics.beginFill(color);
            s.graphics.drawRect(-50, -25, 100, 50);
            s.graphics.endFill();
            s.x = xpos;
            s.y = ypos;
            addChild(s);
            return s;
        }
    }
}
```

10.7.3　gTween 中的补间序列

类似于其他补间引擎，可以使用delay属性建立序列，或者通过监听一个补间的完成再开始下一个补间来建立序列。对此我们已经做过充分的介绍，这里并没有太大区别。有些引擎提供了组类，可以填入补间来创建组。不过，gTween类通过nextTween属性采用了另一种方法实现序列。

简单地说，可以创建两个补间，并暂停第二个补间。将第二个补间的一个引用传入第一个补间的nextTween属性。第一个补间完成时，第二个补间就会开始。通过具体示例说明可能会比直接描述更形象，所以下面给出一个例子，见GTweenSequence.as文件：

```
package {
    import com.gskinner.motion.GTween;

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    import mx.effects.easing.Circular;
    import mx.effects.easing.Quadratic;

    [SWF(backgroundColor=0xffffff)]
    public class GTweenSequence extends Sprite
```

www.TopSage.com

TopSage.com

```
    {
        private var sprite:Sprite;

        public function GTweenSequence()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            var tween1:GTween = new GTween(sprite, 3, {x:800}, {ease:Quadratic.easeInOut});
            var tween2:GTween = new GTween(sprite, 3, {y:400}, {ease:Quadratic.easeInOut});
            var tween3:GTween = new GTween(sprite, 3, {rotation:180}, {ease:Quadratic.easeInOut});


            tween2.pause();
            tween3.pause();

            tween1.nextTween = tween2;
            tween2.nextTween = tween3;
        }
    }
}
```

这里创建了3个补间，然后暂停了第2个和第3个补间。设置tween1.nextTween属性为tween2，并设置tween2.nextTween属性为tween3，它们就可以完美地运行了。

与本章介绍的所有内容一样，这里的讨论只涉及了gTween提供的所有功能的一点皮毛，它通过一个类提供了丰富的功能。你可以查阅文档（我相信这个文档还在不断演进）来了解利用这个引擎还能做哪些工作。实际上，我刚开始写本章到最后审阅本章期间，gTween就发布了一个新版本。因此可以把以上内容作为最简要的介绍，你读到本章时可以再参考其他资源了解其最新功能。

10.8 小结

希望本章能让你对补间引擎领域有一个深入的了解。你可能对已有的不同补间方法有了一个更好的认识，并且找到了一种最能满足需要的方法。或者甚至有可能认为你自己可以做得更好，想要创建你自己的补间引擎！放手去做吧！

另外，尽管我猜想你有可能并不是按顺序读这本书，但本章确实是本书的终结篇。讨论这里介绍的各个主题时，我做了大量研究并提供了相应的有趣例子，然后把经验教训写下来，在这个过程中我感觉很有意思，希望你也与我一样感受到同样的乐趣。最后一点，我相信有本书的一两章作基础你就能完成自己的研究项目。

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