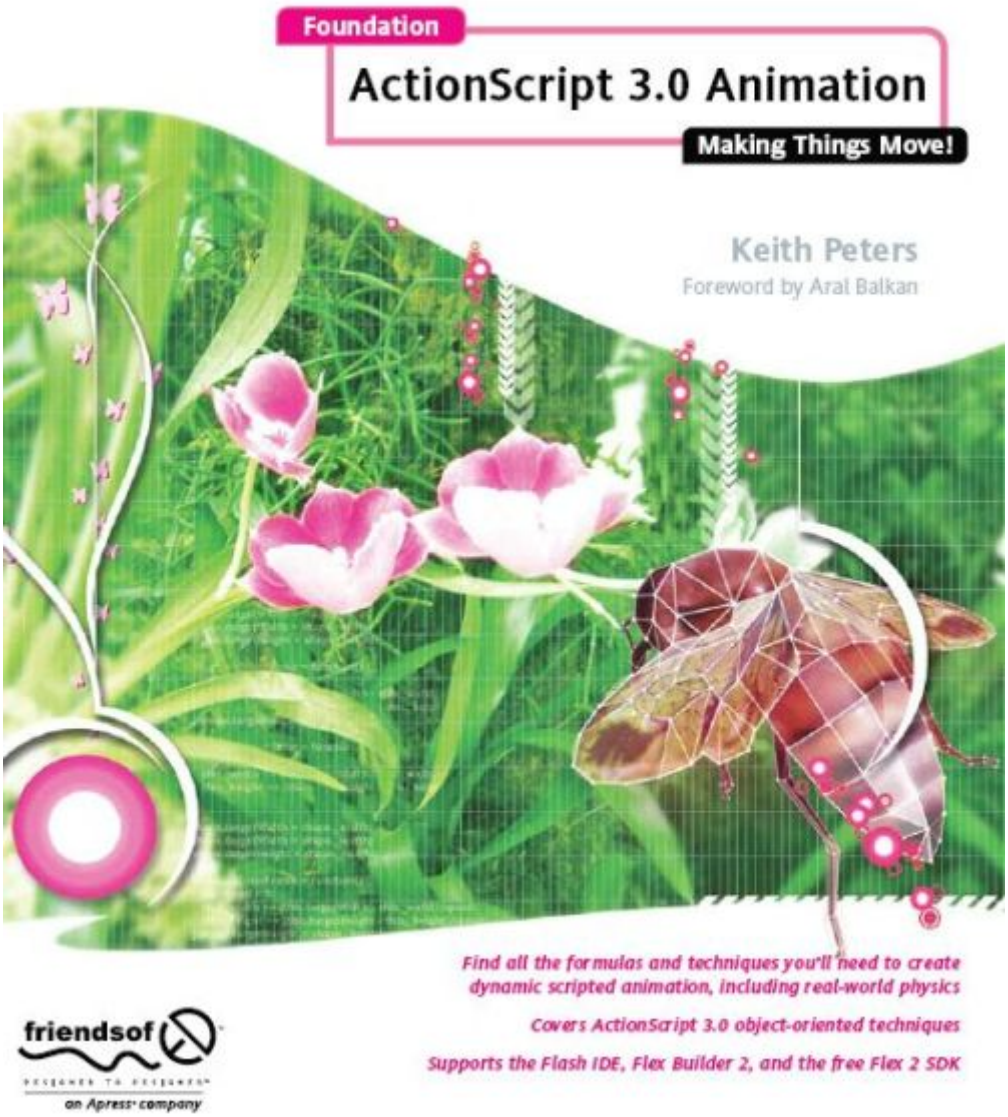


书名：Foundation Actionsript 3.0 Animation

作者：Keith Peters

翻译 / 编辑 / 润稿：FL 基理大师

翻译就是还原，即反映作者的真正意图，这是件很不容易的事。翻译该书完全本着对 Flash 的热爱，也是对自己的一次挑战，望朋友们多提宝贵意见。

本书在业界享有盛名，内容深入浅出，适合各个层次的学习者，能够让读者对 Flash 动画编程有一个全面系统的了解，是学习 AtionSript 的良师益友。

—— FL 基理大师

目 录

说明红色字体表明缺少该部分，作者没有翻译。

第一部分ActionScript动画基础

第 1 章 基本动画概念

1.1 什么是动画
1.2 帧和运动
    1.2.1 帧就是记录
    1.2.2 程序帧
1.3 动态动画 VS 静态动画小结

第 2 章ActionSript 3.0 动画基础

2.1 动画基础
2.2 关于ActionSript版本
2.3 类和面向对象编程
    2.3.1 基类
    2.3.2 包(Package)
    2.3.3 导入(Import)
    2.3.4 构造函数(Constructor)
    2.3.5 继承(Inheritance)
    2.3.6 Movielip/Sprite子类
    2.3.7 创建文档类(Document class)
2.4 设置AtionSript3.0应用程序
    2.4.1 使用 Flash CS3 IDE（集成开发环境）
    2.4.2 使用Flex Builder
    2.4.3 使用免费的命令行编译器
    2.4.4 关于跟踪
    2.4.5 缩放影片
2.5 程序动画
    2.5.1 动画的执行过程
    2.5.2 帧循环
    2.5.3 影片事件
    2.5.4 事件和事件处理
    2.5.5 事件侦听器与处理函数
    2.5.6 动画事件
2.6 显示列表

2.7 用户交互
2.7.1 鼠标事件
2.7.2 鼠标位置
2.7.3 键盘事件
2.7.4 键码
2.8 小结

第 3 章 三角学应用

3.1 什么是三角学(Trigonometry)
3.2 角
3.2.1 弧度制(radian)与角度制(degrees)
3.2.2 Flash 坐标系
3.2.3 三角形的边
3.3 三角函数
3.3.1 正弦(Sine)
3.3.2 余弦(Cosine)
3.3.3 正切 (Tangent)
3.3.4 反正弦(Arcsine)和反余弦(Arccosine)
3.3.5 反正切(Arctangent)
3.4 旋转(Rotation)
3.5 波形
3.5.1 平滑的上下运动
3.5.2 线性垂直运动
3.5.3 心跳运动
3.5.4 双角波形
3.5.5 绘制波形
3.6 圆和椭圆
3.6.1 圆形运动
3.6.2 椭圆运动
3.7 勾股定理
3.8 两点间距离
3.9 本章重要公式
3.10 小结

第 4 章 渲染技术

4.1 Flash中的颜色
4.1.1 使用十六进制表示颜色值
4.1.2 透明度和32位色
4.1.3 新的数值类型：int和uint
4.1.4 色彩合成
4.1.5 获取颜色值

4.2 绘图 API
4.2.1 绘图对象
4.2.2 使用 clear 删除绘制
4.2.3 使用 lineStyle 设定线条样式
4.2.4 使用 lineTo 和 moveTo 绘制直线
4.2.5 过控制点的曲线
4.2.6 使用 beginFill 和 endFill 创建图形
4.2.7 使用 beginGradientFill 创建渐变填充
4.3 颜色变换
4.4 滤镜(Filter)
4.4.1 创建滤镜
4.4.2 动态滤镜
4.5 位图
4.6 读取和嵌入资源
4.6.1 读取资源
4.6.2 嵌入资源
4.7 本章重点公式
4.8 小结

第二部分 基本运动

第 5 章 速度与加速度

5.1 速度向量(Velocity)
5.1.1 向量与速度向量
5.1.2 单轴速度
5.1.3 两个轴上的速度
5.1.5 速度向量扩展
5.1.5 速度扩展
5.2 加速度
5.2.1 单轴加速度
5.2.2 双轴加速度
5.2.3 重力加速度
5.2.4 角加速度
5.2.5 制作飞船
5.3 本章重要公式
5.4 小结

第 6 章 边界与摩擦力

6.1 环境边界

6.1.1 设置边界
6.1.2 移除物体
6.1.3 重置对象
6.1.4 屏幕环绕
6.1.5 回弹
6.2 摩擦力
6.2.1 摩擦力,正确方法
6.2.2 摩擦力，简便的方法
6.2.3 摩擦力的应用
6.3 本章重要公式
6.4 小结

第 7 章 用户交互：移动物体

7.1 按下和放开精灵
7.2 拖拽影片
7.2.1 使用 mouseMove 执行拖拽
7.2.2 使用 startDrag/stopDrag 执行拖拽
7.2.3 结合运动代码的拖拽
7.3 投掷
7.4 小结

第三部分 高级运动

第 8 章 缓动和弹性

8.1 成比例运动
8.2 缓动
8.2.1 简单的缓动
8.2.2 缓动何时停止
8.2.3 移动的目标
8.2.4 缓动不仅限于运动
8.2.5 高级缓动
8.3 弹性
8.3.1 一维坐标上的弹性运动
8.3.2 二维弹性运动
8.3.3 向移动目标运动
8.3.4 弹簧在哪？
8.3.5 弹簧链
8.3.6 多目标点弹性运动
8.3.7 目标偏移

8.3.8 弹簧连接多个物体
8.4 本章重点公式
8.5 小结

第 9 章 碰撞检测

9.1 碰撞检测方法
9.2 hitTestObject 与 hitTestPoint
9.2.1 碰撞检测两个影片
9.2.2 影片与点的碰撞检测
9.2.3 使用 shapeFlag 执行碰撞检测
9.2 4 hitTest总结
9.3 基于距离的碰撞检测
9.3.1 简单的距离碰撞检测
9.3.2 弹性碰撞
9.4 多物体碰撞检测方法
9.4.1 基本的多物体碰撞检测
9.4.2 多物体弹性
9.5 其他的碰撞检测方法
9.6 本章重要公式
9.7小结

第 10 章 坐标旋转和角度回弹

10.1 简单的坐标旋转
10.2 高级坐标旋转
10.2.1 单物体旋转
10.2.2 多物体旋转
10.3 沿角度回弹
10.3.1 实现旋转
10.3.2 优化代码
10.3.3 动态效果
10.3.4 修正“跌落”问题
10.3.5 多角度反弹
10.4 本章重点公式
10.5 小结

第 11 章 台球物理

11.1 质量
11.2 动量
11.3 动量守恒

11.3.1 单轴上的动量守恒
11.3.2 两个轴上的动量守恒
11.4 本章重点公式
11.5 小结

第 12 章 粒子引力和重力

12.1 粒子（Particles）
12.2 重力
12.2.1 万有引力
12.2.2 碰撞检测及反作用
12.2.3 轨道运动
12.3 弹性
12.3.1 引力与弹性
12.3.2 弹性节点花园
12.3.3 结点的连接
12.3.4 有质量的结点
12.4 本章重要公式
12.5 小结

第 13 章 正向运动：行走

13.1 正向和反向运动学介绍
13.2 正向运动学编程准备
13.2.1 单关节运动
13.2.2 双关节的运动
13.3自动运行
13.3.1 创建自然的行走循环
13.3.2 动态调整
13.4 使它真正地行走
13.4.1 给它一些空间
13.4.2 加入重力
13.4.3 控制碰撞
13.4.4 控制反应
13.4.5 屏幕折回
13.5 小结

第 14 章 反向运动：拖动和伸展

14.1 单物体的拖拽与伸展
14.1.1 单关节伸展
14.1.2 单关节拖拽

14.2 多关节拖拽
14.2.1 拖动两个关节
14.2.2 拖拽更多的关节
14.3 多关节伸展运动
14.3.1 抓住鼠标
14.3.2 抓住一个物体
14.3.3 加入一些交互
14.4 使用标准的反向运动学方法
14.4.1余弦定律介绍
14.4.2 ActionScript余弦定律
14.5 本章重要公式
14.6 小结

第四部分 3D动画

第 15 章 3D基础

15.1 第3维和透视
15.1.1 z轴
15.1.2 透视公式
15.2 速度和加速度
15.3 回弹
15.3.1 单个物体回弹
15.3.2 多物体回弹
15.3.3 z轴排序
15.4 重力
15.5 折回
15.6 缓动和弹性
15.6.1 缓动
15.6.2 弹性
15.7 坐标旋转
15.8 碰撞检测
15.9 本章重点公式
15.10 小结

第 16 章 3D线条和填充

16.1 创建点和线
16.2 创建图形
16.3 创建3D填充
16.4 3D实体建模

16.4.1 建立旋转立方体模型
16.4.2 建立其它形状的模型
16.5 移动3D实体模型
16.6 小结

第 17 章 背面剔除和 3D灯光

17.1 背面剔除
17.2 深度排序
17.2.1 背面剔除
17.2.2 深度排序
17.3 3D灯光
17.4 小结

第五部分 其他技术

第 18 章 矩阵数学

18.1 矩阵基础
18.2 矩阵运算
18.2.1 矩阵加法
18.2.2 矩阵乘法
18.3 Matrix类
18.4 小结

第 19 章 实用技巧汇集

19.1 布朗（随机）运动
19.2 随机分布
19.2.1 方形分布
19.2.2 圆形分布
19.2.3 偏向分布
19.3 基于定时器和时间的动画
19.3.1 基于定时器的动画
19.3.2 基于时间的动画
19.4 相同质量物体之间的碰撞
19.5 声音集成
19.6 有用的公式

Making Things Move! 结束语

第一章　基础动画概念

Flash 就是一台动画机器。从 Flash 最早的版本开始，就支持补间动画——只需要创建两个不同的关键帧，然后让 Flash 自动创建补间动画即可。本书将介绍 Flash 中的一种强大的语言 ActionScript。该书包括了编程，数学，物理等技术，并结合 ActionScript 让物体动起来，这些都是补间动画无法比拟的。

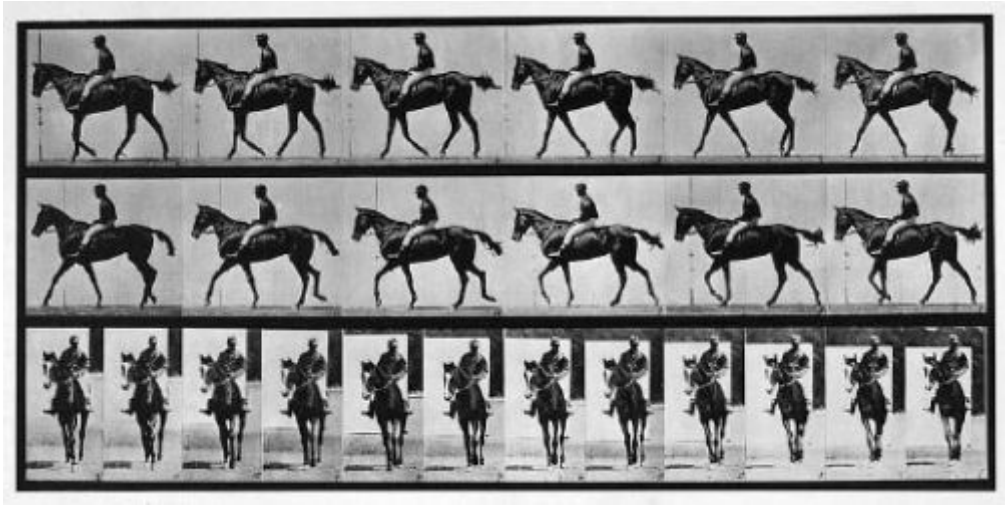
什么是动画？

“动画”一词，引用美国传统词典中的解释

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

前四个哲学定义很好理解，而我们真正要讨论的是第五、六个定义，动画意味着运动。就从这一点开始说起，动画要随着时间而变化，尤其指视觉上的变化。运动基本上表现为物体随着时间，发生位置上的变化，开始在这里，一会儿又在那里。理论上讲，物体也曾介于两点之间某个位置，但我也不能给出纯粹哲学的解释（至少现在还不能）。随着时间的流逝，物体开始在这一点，而后又到了那一个点。

帧和运动



研究人员发现，图像以每秒钟 24 帧的速度播放，最容易被看成运动的图像。比这个速度再慢些，会由于停顿时间较长而引起跳帧，破坏了影像的连贯。人类的眼睛似乎也不能分辨比这个帧频再快的速度，从理论上讲，就算以每秒 100 帧的速度播放也不会使动画变得更真实（虽然快速的帧频会引起程序动画更多的交互响应，看上去会更平滑）。

动态动画 VS 静态动画

使用程序动画的好处不仅是文件大小的问题，这也是程序动画成为动态动画的根本。大家是否看过泰坦尼克这部电影？希望还记得一些，但沉船这件事，却是每次都发生的，不论是在电影院，家庭录像机还是 DVD 中。不管是按下短暂的停止还是暂停，都不会阻止沉船事件，这是因为电影是由一连串静止的图像组成的。要是在我们的电影里有一些物体可以使用鼠标或键盘来改变它们，那又会怎样？允许用户与屏幕上的物体进行交互，这样的效果远超过了静态动画，你甚至可以挽救泰坦尼克！

1.4 小结

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

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

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

第二章　ActionScript 3.0 动画基础

类和面向对象编程

类(Class)和面向对象(Object Oriented)，对于有些读者来说可能还没接触过而有些读者可能已经在 AS (或其它语言) 中使用过很多年了，为了让大家都能学会，我会扼要的介绍一下这些基础知识。就算是 AS 2 的 OOP 专家也希望能略读下这一段，因为 AS 3.0 的工作原理确实发生了很大的变化。如果你说你从没用过类，那你就错了，只要你在 Flash 中写过代码，那么实际上就已经使用了类。类可以简单理解为一种对象， MovieClip 就是影片剪辑的类，而文本框、影片剪辑、按钮、字符串和数值等都有它们自己的类。

一个类最基本的两个部分：属性（数据或信息），行为（动作或它能做的事）。属性(Property)指用于保存与该类有关的信息变量，行为(Behavior)就是指函数，如果一个函数是这个类中的一部分，那么我们就称它为方法(Method)。

一个基本的类：

常用 Flash 的朋友都知道，我们可以在库中创建一个元件，用这个元件可以在舞台上创建出很多的实例。与元件和实例的关系相同，类就是一个模板，而对象(如同实例)就是类的一个特殊表现形式。下面来看一个类的例子：

```
package {
 public class MyClass {
  public var myProperty:Number = 100;
  public function myMethod() {
   trace("I am here");
  }
 }
}
```

先来说明一下这段代码。在这里有些新的知识，对于 AS 2 老手也如此：包的声明。包(Package) ，作用就是把相关的类进行分组。知道这一点就够了，我们不再进行深入的讨论，本书的示例甚至不会用到包。Package 这个关键字和一对大括号是必需有的，我们理解为默认包，紧随其后的就是类的定义。

另一个变化是 AS3.0 中的类拥有了访问关键字。访问关键字是指：一个用来指定其它代码是否可访问该代码的关键字。 public (公有类) 关键字指该类可被外部任何类的代码访问。本书中所有示例的类都是 public 的。在深入学习了 AS 3.0 后，我们会发现不是所有类都是公有的，甚至还有多重的类，这些内容超出了本书的谈论范围。

本例中我们可以看到，这个类的名字为 MyClass，后面跟一对大括号。在这个类中有两种要素，一个是名为 myProperty 的变量，另一个是名为 myMethod 的函数。

包(Package)

包主要用于组织管理类。包是根据类所在的目录路径所构成的，并可以嵌套多层。包名所指的是一个真正存在的文件夹，用 “.”进行分隔。例如，有一个名为 Utils 的类，存在于文件夹 com/ friendsofed/ makingthingsmove/ 中（使用域名作为包名是一个不成文的规定，目的是保证包名是唯一的）。这个类就被写成
com.friendsofed.makingthingsmove.Utils。

在 AS 2 中，使用整个包名来创建一个类，例如：

```
class com.friendsofed.makingthingsmove.Utils {
}
```
在 AS 3 中，包名写在包的声名处，类名写类的声名处，例如：
```
package com.friendsofed.makingthingsmove{
 public class Utils {
 }
}
```

导入(Import)

想象一下，每次要使用这个类的方法时都要输入 com.friendsofed.makingthingsmove.Utils，是不是太过烦琐太过死板了。别担心，import 语句可以解决这个问题。在这个例子中，可以把下面这句放在 package 中类定义的上面：import com.friendsofed.makingthingsmove.Utils;。

构造函数(Constructor)

构造函数是指一个名字与类名相同的方法。当该类被实例化时，该函数会被自动调用，也可以传入参数，例如：
首先，创建一个类：
```
package {
 public class MyClass {
  public function MyClass(arg:String) {
   trace("constructed");
   trace("you passed " + arg);
  }
 }
}
```
然后，假设工作在 Flash CS3 IDE(集成开发环境)中，在时间轴上创建该实例：
```
var myInstance:MyClass = new MyClass("hello");
```
结果输出：
```
constructed
you passed hello
```

继承(Inheritance)

一个类可以从另一个类中继承(inherit)和扩展(extend)而来。这就意味着它获得了另一个类所有的属性和方法(除了那些被 private 掩盖住的属性)。所生成的子类(派生类)还可以增加更多的属性和方法，或更改父类(基类)已有的属性或方法。要分别创建两个类来实现（两个独立的 .as 文件），例如：
```
package {
 public class MyBaseClass {
  public function sayHello():void {
   trace("Hello from MyBaseClass");
```

```
  }
 }
}

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

不要忘记，每个类都必须在其自身的文件中，文件名为该类的类名，扩展名 .as，所以必须要有 MyBaseClass.as 文件和 MySubClass.as 文件。因此，在使用 Flash CS3 IDE 时，保存的 FLA 文件，要与这两个类在同一个文件夹。

下面代码会生产两个实例，把它写入时间轴看看会发生什么：

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

第一个实例没什么可说的，值得注意的是第二个实例中的 sayHello 方法，虽然在 MySubClass 中没有定义 sayHello，但它却是继承自 MyBaseClass 类的。另一个值得注意的是，增加了一个新的方法 sayGoodbye，这是父类所没有的。

下面说说，在子类中如何改变一个父类中已存在的方法。在 AS 2 中，我们可以只需要重新定义这个方法就可以了。而在 AS 3 中，则必需明确地写出 override 关键字，来进行重新定义。

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

请注意，原来的 sayHello 方法被重写，再调用 MySubClass 后，就有了新的信息。另外，私有成员也不能被重写，因为它们只能被它们自身的类访问。

MovieClip/Sprite 子类

我们可以自己写一个类，然后让另一个类去继承它。在 AS 3 中，所有代码都不是写在时间轴上的，那么它们一开始都要继承自 MovieClip 或 Sprite。 MovieClip 类是影片剪辑对象属性和方法的 ActionScript 模板。它包括我们所熟悉的属性如：影片的 x,y 坐标，缩放等，这些在 AS 3 中的变化不大。

AS 3 还增加了 Sprite 类，通常把它理解为不在时间轴上的影片剪辑。很多情况下，只使用代码操作对象，并不涉及时间轴和帧，这时就应该使用 Sprite 这个轻型的类。如果

一个类继承自 MovieClip 或 Sprite ，那么它会自动拥有该类所有的属性和方法，我们还可以为这个类增加特殊的属性和方法。

例如，游戏设计一个太空船的对象，我们希望它拥有一个图形，并且在屏幕的某个位置移动，旋转，并为动画添加 enterFrame 侦听器，还有鼠标、键盘的侦听等。这些都可以由 MovieClip 或 Sprite 来完成，所以就要继承自它们。同时，还可以增加一些属性如：速度(speed)、油量(fuel)、损坏度(damage)，还有像起飞(takeOff)、坠落(crash)、射击(shoot)或是自毁(selfDestruct)等方法。那么这个类大概是这样的：

```
package {
 import flash.display.Sprite;
 public class SpaceShip extends Sprite {
  private var speed:Number = 0;
  private var damage:Number = 0;
  private var fuel:Number = 1000;
  public function takeOff():void {
   //...
  }
  public function crash():void {
   //...
  }
  public function shoot():void {
   //...
  }
  public function selfDestruct():void {
   //...
  }
 }
}
```

注意，首先要导入 flash.display 包中的 Sprite 类，如果要导入 MovieClip 类，同样也需要导入这个相同的包 flash.display.MovieClip 类。

创建文档类(Document class)

现在我们对类已经了一定的了解，接下来，看看如果真正地使用它。有时候我常说基于 AS 3 的 SWF 是多么的重要，这是因为 AS 3 引入了一个全新的概念，文档类(document class)。

一个文档类就是一个继承自 Sprite 或 MovieClip 的类，并作为 SWF 的主类。读取 SWF 时，这个文档类的构造函数会被自动调用。它就成为了我们程序的入口，任何想要做的事都可以写在上面，如：创建影片剪辑，画图，读取资源等等。如果在 Flash CS3 IDE 中写代码，可使用文档类，也可以选择继续在时间轴上写代码。但如果使用 Flex Builder 2 或免费 Flex SDK，那里没有时间轴，唯一的办法就是写在类中。这些工作一切都围绕着强大的文档类而展开，没有它就没有 SWF。以下是一个文档类的框架：

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

```
  }
  private function init():void {
   // 写代码处
  }
 }
}
```

如果你看过前面的部分，不会认为这是个新知识，只不过是把他们放在了一起而已。使用默认包，导入并继承 Sprite 类。构造函数只有一句，调用 init 方法。当然，也可以把所有代码写在构造函数里，但是要养成一个好习惯，就是尽量减少构造函数中的代码，所以把代码写到了另一个方法中。本书会给大家很多代码块进行测试，那时要像上面这个例子一样把代码块放入 init 方法中，这样在影片编译执行时，就会调用 init 中的代码。下面我们要开始学习如何连接文档类和 SWF。

使用 Flash CS3 IDE（集成开发环境）

Flash CS3 IDE 是实现文档类的最方便的工具。把上述的类选择一个文件夹进行保存，文件名为 Test.as。打开 Flash CS3 ，创建一个 FLA 文件，保存到一这个类相同的目录下。确认 FLA 默认发布设置为 Flash Player 9 及 AS 3.0。在属性面板中，我们注意到出现了一个名为文档类（Document Class）的区域(图 2-1)。只需输入类名：Test。

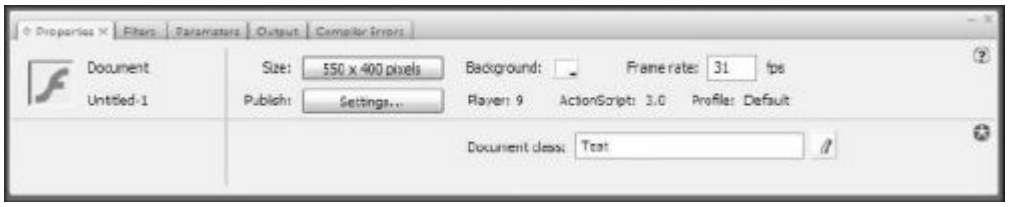


图 2-1 设置文档类

请注意，我们输入的是类名，而不是文件名。所以这里不需要输入扩展名 .as。如果这个类包涵在一个包中，那么就需要输入类的完整路径——例如：
com.friendsofed.chapter2.Test。

程序动画

下面，再来学习一些 AS 3 编程的基本原理。如果你已经选择好了一个开发环境，那么就出发吧。让我们进入 ActionScript 动画世界。

动画的执行过程

几乎所有的程序动画都包括几种不同的执行过程。对于逐帧动画来说，意味着创建和存储一组连续的位图，每一帧都是一幅图像，只需要进行显示即可，见图 2-3。

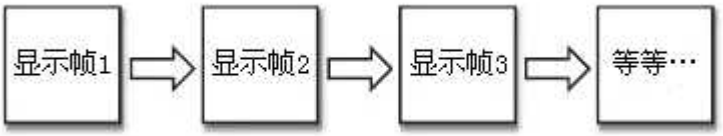


图 2-3 逐帧动画

当我们在 Flash 中使用图形或元件时，事情就发了微妙的变化。这时，Flash 不会为

每一帧创建和存储新的位图。对于每一帧而言，Flash 存储的是舞台上每个对象的位置，大小，颜色等等。比如，一个小球在屏幕上移动，每一帧只存储小球的在该帧上的位置，第 1 帧小球的位置在左边第 10 个像素，第 2 帧也许就在第 15 个像素，等等。Flash 播放器(Flash Player)读取这些信息，再根据这些信息的描述来渲染舞台并显示该帧。根据这些变化扩展一下流程图，见图 2-4。

图 2-4 渲染并显示帧

我是这样描述一个动态程序动画的，见图 2-5。

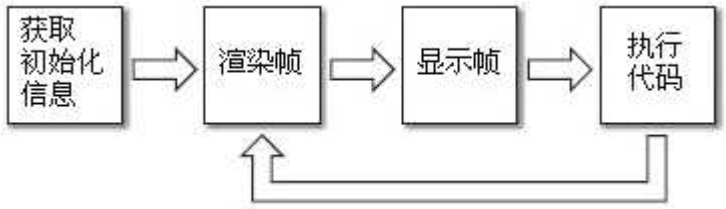


图 2-5 脚本动画

如图 2-5 所示，没有第 1 帧、第 2 帧的概念，脚本动画通常只由一帧完成。下面我们就来看看动画的执行过程。

首先，建立初始化。舞台中放入一个影片剪辑，再创建补间动画，或使用代码来描述整个场景。总之，最后都要对该帧进行渲染及显示。

然后，应用自定义规则。规则可以像“让球向右移动 5 像素”这样简单，也可以是由几十条复杂的三角函数组成。使用自定义规则会产生新的描述再根据这些描述进行渲染及显示，并不断地应用这个规则。

请注意，同一规则被一遍又一遍地执行，而不是对第 1 帧用一套规则，而对第 2 帧又使用另一套规则。所以难度就在于，一套规则要处理所有可能出现的情况。要是球向右移动得过远，超出了舞台怎么办？你的这套规则就要解决这个问题。是否还希望用户通过鼠标来操作小球？那么你的规则也要把它考虑进去。

听起来很复杂，其实不然，这里所说的“规则”，实际上就是 ActionScript 代码。每套规则都可由一行或多行代码组成。下面是小球向右移动 5 像素的例子：

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

这句话是说无论小球 X 坐标(水平轴)在哪里，都在原来的 X 位置上增加 5 像素，并把该坐标作为它的新 X 坐标。也可简化为：

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

“+=”操作符：把右边的值与左边的变量相加，相加的结果再赋值给该变量。以下是更多的高级规则，日后会学到：

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

```
graphics.lineStyle(1);
graphics.moveTo(ball.x, ball.y);
graphics.lineTo(mouseX, mouseY);
```

这段现在看不懂没关系，大家只要知道 Flash 会在每一帧中生成这段代码，并不断地执行。

怎样让它循环执行？看看我第一次的尝试，这也是很多 AS 初学者都会犯的错误。这是在很多程序设计语言中都存在的循环结构，如 for 和 while。用循环结构使代码重复执行，这就是我曾写的那段：

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

看起来相当简单。变量 i 从 0 开始，所以小球 X 坐标移动到 0——舞台最左边。i++让 i 的值每次增长 1，即：0～1～2～3～4…，每次这个值都会做为 ball.x 的值，把小球从左向右移动。当值为 500 时，表达式 i<500 值为假(false)，循环结束。

如果你也犯过同样的错误，就会知道，小球没有在舞台上发生移动——只是一下子出现在了舞台的右边而已。为什么没有移动到中间的那些点上？其实它移动了，只是我们没有看到，因为我们没有让 Flash 去刷新屏幕。图 2-6 为另一个流程图，看看实际都发生了什么。

图 2-6 为什么循环结构无法产生动画

实际上我们使用自定义规则使球移动到指定位置，并创建了 500 次新的场景。但在循环结束之前没有给出显示，这是因为 Flash 只在每一帧结束后才进行一次刷新，这点很重要。以下是 Flash 进入帧的动作顺序：

1.在舞台上放置所有的对象，不论在何级，何层，或是否为加载的影片。

2.执行帧上所有的 Action 脚本(ActionScript)，不论在何级，何层，不论处于影片剪辑还是按钮中，也不论它嵌套在何处。

3.判断是否到了该显示的时候。如果设置帧频为 20 帧/秒，Flash 最少要等上一帧显示后 50 毫秒后再进行下一次显示，显示了该帧后，就要编译和进入下一帧。如果帧频没有到 20 帧/秒，那么要等待到正确的时间再去执行。

定时时间存在着一些问题。首先，众所周知帧频是不精确的(即使在 Flash 9 中)，不要依赖它作为精确的定时器。其次，在大量的编译和 AS 执行花费的时间会超出规定的时间。

尽管如此，我们也不必担心自己的脚本会被砍掉一部分。在进入第 3 步之前，Flash 会完成所有可执行代码(第 2 步)，即使需要延缓帧频也要完成。Flash 为了能完成脚本，甚至会等上 15 秒。在上面的例子中，Flash 等待循环结束，然后进入下一帧，只在跳转到下一帧时进行屏幕的刷新。这就是为什么我们看到的是跳动而不是移动。因此，要想完成移动，我们所要做的就是打散这个循环，请回过头参考图 2-5。

帧循环

帧循环的理念，存在于 Flash 最早的版本中，那时 ActionScript 还不像今天那么强大。把代码写入关键帧，并在下一帧中写入像 gotoAndPlay 这样的语句，使播放头(playhead)回到前一帧。这样两帧之间就形成了一个无限循环，每当播放头到了代码帧上时，就会执行

那些代码。例如，在舞台上有一个实例名为 ball 的影片剪辑。
第一帧的代码就像这样：

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

第二帧的代码如下：

```
gotoAndPlay(1);
```

实际上第二帧不需要做任何事，只是让时间轴自动回到第一帧而以。另一个版本是建立三个帧，第一帧进行初始化，写入只执行一次的代码，不进行循环。第二帧才是主要的执行代码，第三帧只写 gotoAndPlay(2); 这个方法在早期 Flash 版本中常被使用，虽然有点过时，但是同样可以出色地完成任务。马上我们还要学到更灵活更强大的设置方法，但今后你会发现其实原理上是一样的。

影片事件

影片事件在 AS 3 中彻底的消失了，这真是件好事。但还要捎带提一下，回顾 Flash 5 的时代，只有帧循环和影片剪辑事件两种选择。影片事件指代码直接写在影片剪辑上，而不是帧上。如何实现影片事件，首先选择舞台上的影片剪辑，然后打开动作面板并将代码写在上面，这些代码只对该影片剪辑有效。所有代码必需写在事件块中，比如：

```
onClipEvent(eventName){
// code goes here
}
```

对于 onClipEvent(eventName)，作用于 eventName(某种事件)。对于"on"类型事件则必需指定鼠标或键盘事件，如按下(press)和释放(release)。

事件名称(eventName)是指许多 Flash 影片事件之一，所谓事件就是在影片中发生的事。事件分为两种：系统事件和用户事件。系统事件指发生在如计算机，Flash，或影片上的事件，比如调取数据，调取信息，或播放帧等。用户事件是指用户所做的一些事，基本上就是鼠标和键盘两种。影片事件使用得最多的就是 load 和 enterFrame 这两个。 Load 事件会在影片第一次出现在舞台上时才执行，且只执行一次。所以说非常适合在这里面写入初始化代码。只要把代码写在大括号间即可：

```
onClipEvent(load){
// initialization code
}
```

我们可以把带有如下代码的影片剪辑放入时间轴上(注意：此处为 AS 1 写法)：

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

本书示例中的代码不使用这种写法（因为它已经不是一种语言了），但不论使用何种方法，初始化(initialization)，重复动作(repeating actions)和屏幕刷新(screen refresh)都是非常重要的。

DisplayObject 类。换句话讲，这些对象都是一个大家庭的成员，并以相同的形式工作，使用同样的方式进行创建，置入，删除，操作。无论创建 Sprite 影片，影片剪辑或文本框的方法都非常相近，我们需要使用 new 关键字来完成，创建任意类型的对象。为了证明这一点，请看下面三条示例：

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

如果我们创建的是一个影片剪辑或 Sprite 影片的话，就可以直接里面进行绘制，如：

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

但只有这些代码，还不能看到效果，这就引发了接下来要讨论的显示列表。“显示列表”是个新名词，可以理解为一颗由可视对象构成的树。舞台就是树根，默认为可见的，在舞台上，我们可以有很多影片剪辑或可视对象(文本框，图形等)，把它们加入舞台后，也就成为可见的了。

这些影片中也许还嵌套着很多层的可视对象，这就是我们所谓的显示列表。AS 2 与 AS 3 显示列表最大的不同在于，AS 2 中，当使用 attach 或 createEmptyMovieClip 方法创建影片剪辑时，必须指定它位于树的哪个位置。这样一来，影片剪辑要放置在列表的指定位置。当删除该影片时，同样也无法改变它在列表中的位置或在列表中移除它。

在 AS 3 中，创建了一些 Sprite 影片后，不会自动被加入显示列表。在上面的示例中我们发现，创建一个 Sprite 后，并不涉及父级影片(parent)或深度(depth)的问题，这样就可以在它没有加入视觉列表之前就对其进行操作了。说到舞台(Stage)，可以把这些显示对象看作是幕后的演员，虽然看不到，但确实存在，并时刻准备着亮相的一刻，我们使用 addChild 方法把对象加入显示列表。将文档类作为树根，向里面加入孩子时，会自己被设置为可见的。

现在，在前面的例子中再加入创建 Sprite 对象以及 addChild 方法，如下：

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

如果大家有兴趣试一下这段代码的话，请把它们写入前面所给的类框架的 init 函数中。请注意，绘制出的圆默认位置是 0,0 点，可以改变其 x 和 y 属性。还要注意，创建新影片时不再需要像 AS 2 那样去设置深度(depth)。虽然深度管理为自动执行，但我们还有指定深度或改变深度的方法，这部分等将来用到时再讲。

使用 removeChild 方法，从显示列表中删除一个对象，并以该对象的名字作为参数。第一，删除一个对象，不是去毁灭它，对象依然保持原样，只是暂时被移除，当再次被加入到显示列表中，对象仍保持原来的状态。换句话讲，如果显示对象里面绘制了图形，或是已加载了一些外部信息，那么将它重新加入显示列表后，就不必再去重绘或重载这些信息。第二，把该对象重新加入显示列表后，还可以为它指定处在显示列表中的位置，这就是我们所熟知的重定父级。

从一个影片剪辑中删除一个对象，再把它加载到另一个影片中剪辑中，并保持刚刚被删除时的状态，在以前是不可能完成的。事实上，有时并不需要去删除影片，因为，一个子对象只能有一个父级，把它加入到另一个父级中，就会自动从原来的父级中删除。请看下面示例：

```
package {
 import flash.display.Sprite;
 import flash.events.MouseEvent;
 public class Reparenting extends Sprite {
```

事件及事件处理

Flash MX 的 ActionScrpt 发生了重要的改变，这些转变与革新为 Flash 成为真正的富客户端程序(RIA)奠定了基础。其中一个就是全新的事件结构，在编写非常复杂的行为时比之前的版本好用很多。 Flash MX 之前的版本，只能把代码放在影片和按钮的 onClipEvent(eventName) 或 on(eventName) 这两种事件处理方法中。这就意味着，在设计的时候就要把影片剪辑放到舞台上，并把代码写入影片剪辑中。MX 的事件结构并不完美，但与之前版本来说已经有了长足的进步，并允许我们在任何时候访问任何事件，或是停止处理任何事件，或是动态改变某个事件的行为，可以想象这有多么的强大和灵活。

要想了解事件，就要明白下面几条概念：侦听器(lintener)与处理函数(handler)，这两个名字很贴切，侦听器就是侦听事件的对象，处理函数是一个用于处理所要发生的事件的函数。侦听与处理在 ActionScript 的发展过程中进行过很多次演变，在 AS 2 中就有很多不同的实现方法。为了避免混乱，我很推崇 AS 3，因为它简化了这个过程，使事件处理变得更方便更一致。

事件侦听器与处理函数

前面说过，侦听器是一个用于侦听事件的对象。我们可以设计一个类，通过调用 addEventListener 函数为某事件指定一个侦听器。输入要侦听的事件名称以及要执行处理的函数名称。看一个例子：

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

在加入事件侦听器时，可使用可选参数，本书中不会用到；对于大多数的应用程序来说，会使用以上这种写法就够用了。请注意事件名"enterFrame"为字符串型，戏称它为“魔力字符串”(Magic String)。为什么这么叫？如果你误输入成了"entorFrame"，尽管没有这个事件名称，编译器也会编译执行它，会发现事件处理函数没有执行。但 AS 3 仍会对其进行处理，除了使用“魔力字符串”以外，还可以使用事件类(Event Class)的属性。例如：

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

实际上 Event.ENTER_FRAME 的值就是”enterFrame”这个字符串。那么这个属性也可能输错就像 Event.ENTOR_FRAME ，但这种方法好在，如果输入错误了，程序会拒绝编译，并提示你在事件类中不存在该属性。编译器会提示发生错误的行及确切的字符。所以，最好使用这种方法，除非编译器会帮我们修正错误或编写代码。

除此之外，还有其它的事件类型如：MouseEvent.MOUSE_DOWN，KeyboardEvent.KEY_DOWN，TimerEvent.TIMER 等。这些都由 "mouseDown"，"keyDown"，"timer" 这样的简单字符串来表示，如果你记不住这些字符串，那么最好就去使用事件类的属性。

另一个重点是，使用 addEventListener 函数直接调用类中的函数。有时，需要侦听另一个对象产生的事件，例如，有一个名为 mySpriteButton 的 Sprite 影片(Sprite)：影片或按钮，能完成按钮的动作。当用户点击它的时候就会产生 mouseDown(鼠标按下)事件。侦听该 Sprite 影片的 mouseDown 事件，就要调用该对象的 addEventListener 方法，如下：

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

最后一点，必需要有事件处理函数如 onEnterFrame，在 AS 3 中，可以任意地为事件处理函数命名，这点与以前的 ActionScript 不同。在 enterFrame 示例中，使用 onEnterFrame 做事件处理函数，是因为我们习惯使用这个名称。在 AS 3 中，onEnterFrame 已不再是关键字，当然也可以为这个处理函数命名为 move，run，或是 doSomethingCool。然而，我们已经习惯使用”on”表示事件开始，后面跟一些描述词如 onStartButtonClick，onConfigXMLLoad 或 onRoketCrash。有些朋友喜欢在事件名后面加上 "Handler" 作为后缀，如： enterFrameHandler，这只是个人偏好问题。

侦听器用于侦听事件，但对于一个侦听器来说，也许会同时侦听很多事件。在系统内部，一个事件对象拥有一个包括了所有对象及自身的侦听器的列表。如果一个对象能够产生多种不同类型的事件，如 mouseDown，mouseUp，mouseMove 等，那么它就拥有一个侦听器列表，其中包括它所涉及的所有类型的事件。无论触发何种事件，都会检索一遍列表，然后使列表中的每个对象都知道所发生的事件。

另一种对事件的描述是，将其看作一个加入到事件行列的侦听器成员。产生事件的对象将它所产生的事件公布给所有成员，当你不再需要这个对象进行侦听时，可以令其停止侦听或使用 removeEventListener 方法解除该成员；就是告诉对象从侦听器列表中删除该侦听器，这样一来，他就不会再接收信息了。

让我们看看这段代码，下面是一段在舞台中创建 Sprite 影片，并进行绘图，然后再为其添加侦听器的代码：

```
package {
 import flash.display.Sprite;
 import flash.events.MouseEvent;
 public class EventDemo extends Sprite {
  private var eventSprite:Sprite;
  public function EventDemo() {
   init();
  }
  private function init():void {
   eventSprite = new Sprite();
   addChild(eventSprite);
   eventSprite.graphics.beginFill(0xff0000);
   eventSprite.graphics.drawCircle(0, 0, 100);
   eventSprite.graphics.endFill();
   eventSprite.x = stage.stageWidth / 2;
   eventSprite.y = stage.stageHeight / 2;
   eventSprite.addEventListener(MouseEvent.MOUSE_DOWN,onMouseDown);
   eventSprite.addEventListener(MouseEvent.MOUSE_UP,onMouseUp);
  }
  private function onMouseDown(event:MouseEvent):void {
   trace("mouse down");
  }
  private function onMouseUp(event:MouseEvent):void {
   trace("mouse up");
  }
 }
}
```

在初始化函数(init)中创建一个 Sprite 影片,并在里面画圆，置于舞台中心，最后两句是为它添加两个侦听器，侦听鼠标按下(MOUSE_DOWN)和鼠标弹起(MOUSE_UP)这两个事件。它们是 MouseEvent 类的两个属性，而这个类必需要导入。最后定义两个处理函数 onMouseDown 和 onMouseUp。

由事件对象调用事件处理函数，通常还会包括一些事件信息。在处理鼠标事件时，就包括触发该事件时鼠标位置的信息如：鼠标点击在按钮上。对于键盘事件，就要包括按下键时的信息如 Ctrl，Alt，Shift 等。把上述示例保存为 EventDemo.as 文件，并选择一种前面讲过的编译方式。当运行 SWF 时，就会看到每次点击或图形时，都会输出 pressed 或 released。

动画事件

我们希望能够使用代码让物体动起来，并允许屏幕反复地刷新。前面看过一个使用 enterFrame 影片事件的示例。现在把这种方法运用到 AS 3 中，只需要增加一个 enterFrame 事件的侦听器即可：

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

别忘了导入 Event 类，并创建一个名为 onEnterFrame 的方法。人们常常迷惑，只有一帧怎么能执行 enterFrame(进入帧) 事件呢？事实上，播放头并非真正地在进入下一帧，它只停留在第一帧上，并不是把播放头移动到下一帧才形成了 enterFrame 事件，而是用另一种方法：Flash 告诉播放头何时进行移动，可以把 enterFrame 看成一个定时器，只是有些不精确。

下面我们看看第一个 AS 3 动画：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class FirstAnimation extends Sprite {
  private var ball:Sprite;
  public function FirstAnimation() {
   init();
  }
  private function init():void {
   ball = new Sprite();
   addChild(ball);
   ball.graphics.beginFill(0xff0000);
   ball.graphics.drawCircle(0, 0, 40);
   ball.graphics.endFill();
   ball.x = 20;
   ball.y = stage.stageHeight / 2;
   ball.addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   ball.x++;
  }
 }
}
```

init 函数创建了一个名为 ball 的 Sprite 影片，并为其建立事件侦听。onEnterFrame 函数负责 ball 的运动及屏幕刷新工作。这是学习本书内容的基础，也是使用 ActionScript 创建动画的基础，所以务必要掌握。

显示列表

在 AS 3 之前，人们可以创建多个不同类型的可视化对象，包括影片剪辑，图形，按钮，文本框，位图，组件和基本形状。这些对象没有真正的层次结构，它们的创建、删除、操作方法也均不相同。比如，在 IDE 中，可以使用 attachMovie ，duplicateMovieClip 或 createEmptyMovieClip 的方法将影片剪辑放置于舞台上，文本框可以在开发环境中创建也可以用代码创建。而在使用位图(bitmap)，视频(video)及组件(component)时，它们就像是来自于别的星球，最终被强硬地放在一起。

对于 AS 3 来说，这些对象都有了统一的归属。在舞台上所有可见的对象都继承自

键盘事件

键盘事件已被 AS 3 划分到另一个区域中。例如，在 AS 2 中，影片剪辑会自动侦听键盘事件，但只在某种情况下才接收这些事件。所以，最好增加一个专门用来做侦听器的影片剪辑，有时，影片剪辑接收了多个事件但被看作是一个键盘事件，这样就不对了。在 AS 2 的组成框架中，很大部一部分都是为键盘交互服务的，比如 Flash Player 体系中的：
tab(table)管理，焦点(focus)管理及在文本框中对于 Enter 键与 Table 键的处理等。现在好了，键盘事件的名称与鼠标事件的相似，都是定义好的字符串，也可为 KeyboardEvent 类的属性。只有两种：
KEY_DOWN
KEY_UP

我们可以在一个特殊的对象上侦听键盘事件，就像上面那个鼠标侦听的例子一样。为了实现这个功能，我们需要设置对象的焦点，以便能够捕获这些事件，可以这样写：

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

在很多情况下，侦听键盘事件是否有焦点很有意义，实现它只需直接对舞台进行键盘侦听。下面看一个示例：

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

键码

通常人们并不关心一个键是否被按下，而是关心按下的是什么键。使用键盘事件处理有几种方法可以读取输入的信息。前面说到，一个事件处理程序可以由一个事件对象来触发，该对象包括触发这个事件的数据。在键盘事件中有两个相关的属性，事件所涉及的键：字符码(charCode)和键码(keyCode)。
字符码指按下的键所表示的真正字符。例如，用户按下”a”键，字符码就是”a”，如果用户同时又按着 shift 键，这样字符码就是”A”。
键码指按键所代表的数值。如果用户按下”a”键，它所对应的键码为 65，无论是否按着其它键。如果先按下 Shift 键后按下“a”键，那么会获得两个键盘事件，先是 Shift(键码 16)后是 a(键码 65)。 Flash.ui.Keyboard 类同样也有一些属性是针对非字母键的，我们不需要把它们背下来。例如： Keyboard.SHIFT 等于 16，当 Shift 键按下后，可以测试其是否等于 Keyboard.SHIFT。请本章的最后一段代码：

```
package {
```

```
  private var parent1:Sprite;
  private var parent2:Sprite;
  private var ball:Sprite;
  public function Reparenting() {
   init();
  }
  private function init():void {
   parent1 = new Sprite();
   addChild(parent1);
   parent1.graphics.lineStyle(1, 0);
   parent1.graphics.drawRect(-50, -50, 100, 100);
   parent1.x = 60;
   parent1.y = 60;
   parent2 = new Sprite();
   addChild(parent2);
   parent2.graphics.lineStyle(1, 0);
   parent2.graphics.drawRect(-50, -50, 100, 100);
   parent2.x = 170;
   parent2.y = 60;
   ball = new Sprite();
   parent1.addChild(ball);
   ball.graphics.beginFill(0xff0000);
   ball.graphics.drawCircle(0, 0, 40);
   ball.graphics.endFill();
   ball.addEventListener(MouseEvent.CLICK, onBallClick);
  }
  public function onBallClick(event:MouseEvent):void {
   parent2.addChild(ball);
  }
 }
}
```

该类中有三个 Sprite 对象：parent1, parent2, ball。 parent1, parent2 影片直接加入显示列表，并在影片中绘制了正方形。 Ball 影片被加入到 parent1，就相当于加入了显示列表并可见。当小球被点击时，它将被加入 parent2 影片中。请注意，没有改变 Ball 的 x,y 坐标的代码,之所以产生移动是因为 Ball 被加载到了不同位置的 Sprite 影片中。Sprite 影片被删除后再被加入显示列表，Ball 仍会出现。

子类化显示对象

前面已经讲过生成 Sprite 或 MovieClip 类的子类，对某个类进行子类化是非常有用的。首先，大家可能对 AS 3 取消 attachMovieClip 功能感到十分惊呀，如果这样的话，我们怎么才能在 Flash CS3 IDE 库中取出影片剪辑元件放入舞台呢？答案是，使用一个继承自 MovieClip 或 Sprite 的类。为了能够好地解释这个问题，简单地介绍一下 IDE：

1.创建一个新的 FLA 文件，并在舞台上绘制一些图形。

2.选中图形按下 F8 键转换为元件。

3.在转换为元件窗口中输入一个名称，并设置为影片剪辑类型。

4.选择为 ActionScript 导出。

在以前的 Flash 版本中，可以自由地给出标识符或输入一个类名。而在 Flash CS3 中，标识符一栏不可用了，类一栏会自动地填入默认值。这里还多出了基类一栏，默认为 flash.display.MovieClip，这里也可以填入继承自 MovieClip 或 Sprite 的自定义类。

随意输入一个类名，不必担心没有这个类，然后点击确定。这个地方很有趣，Flash 找不到这个类，它就会自动生成一个类，并对其进行编译。并不是说 Flash 会创建一个 ActionScript 类文件，但它会在 SWF 中，生成一串字节代码表示一个继承自 Sprite 或 MovieClip 的类。除了继承了基类，它什么都不会做，但它已经与库中的元件连接上了。比如，你的类名为 Ball。在文档类或时间轴上，可以这么写：

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

这样就在舞台上创建了一个库中的元件，就像 AS 2 的 attachMovie 方法一样。我们要是能给出自定义的真正的类名及路径的话，那么就可以让元件附加很多功能。现在，我们跳出 Flash IDE 回到类的世界，看下一个示例。下面再看一个重定父级的示例，这里有一些重复的部分可以写入另一个类中。看一下示例，假设已经创建了一个名为 parent1 的 Sprite 实例，要里面绘制正方形：

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

下面再创建一个名为 parent2 的 Sprite 实例，同样也是绘制一个正方形。当然这个例子毫无意义，但它可以告诉我们 Sprite 的子类是多么的有用。首先，我们建立一个名为 ParentBox 的类，并继承自 Sprite，这样一来，就拥有了绘制正方形的代码：

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

然后，使用这个类创建两个 ParentBox，这样做比创建两个 Sprite 对象要好得多。

```
package {
 import flash.display.Sprite;
 import flash.events.MouseEvent;
 public class Reparenting2 extends Sprite {
  private var parent1:ParentBox;
  private var parent2:ParentBox;
  private var ball:Sprite;
  public function Reparenting2() {
   init();
  }
  private function init():void {
   parent1 = new ParentBox();
   addChild(parent1);
   parent1.x = 60;
   parent1.y = 60;
   parent2 = new ParentBox();
```

```
    addChild(parent2);
    parent2.x = 170;
    parent2.y = 60;
    ball = new Sprite();
    parent1.addChild(ball);
    ball.graphics.beginFill(0xff0000);
    ball.graphics.drawCircle(0, 0, 40);
    ball.graphics.endFill();
    ball.addEventListener(MouseEvent.CLICK, onBallClick);
   }
   public function onBallClick(event:MouseEvent):void {
    parent2.addChild(ball);
   }
  }
}
```

作为 ParentBox 的实例，它们仍是 Sprite ，因此还可以再增加子影片， init 方法直接进行绘图。虽然这个示例价值不大，但可以让你学会这种思想，往后，会在书中看到更多更复杂的示例。你也许想自己动手创建一个球(Ball)类，用于绘制小球，虽然这么做不会减少代码里，但是当你的类变得十分复杂时，把功能代码分离到不同的类中，这绝对是个好办法，它比将所有类写在一起要好得多，同时还促进了代码的重用性。那么现在就去创建这个 Ball 类吧，在日后的学习中还要用到呢。

交互动画

最后介绍一下交互动画，这也许是大家读这本书的主要原因。如果不使用交互运动，那么只使用补间动画不就行了。在前面一章简单地提到过，用户交互动画基于用户事件，总的来说可以归结为鼠标事件和键盘事件，下面就来学习不同的用户事件及其处理函数。

鼠标事件

AS 3 中鼠标事件发生了显著的变化。在 AS 2 中，影片剪辑会自动添加鼠标侦听器。现在，要手动地为对象添加侦听器。在 AS 3 中鼠标指针经过显示对象时才能触发鼠标事件。在 AS 2 中，无论鼠标指针在哪里，只要执行 mouseDown 或 mouseMove 就会触发所有的影片剪辑。而现在， mouseUp 和 mouseDown 事件与 AS 2 中的 onPress 和 onRelase 等同。鼠标事件的名称是定义好的字符串，像我们之前所提到的，最好使用 MouseEvent 类的属性，以避免输入错误，下面是 MouseEvent 类中所有可用的鼠标事件属性：

CLICK
DOUBLE_CLICK
MOUSE_DOWN
MOUSE_MOVE
MOUSE_OUT
MOUSE_OVER
MOUSE_UP
MOUSE_WHEEL
ROLL_OUT
ROLL_OVER

创建下面这个类，来测试一下，这个类会输出发生在 Sprite 影片上的鼠标事件名称。

```
package {
 import flash.display.Sprite;
 import flash.events.MouseEvent;
 public class MouseEvents extends Sprite {
  public function MouseEvents() {
   init();
  }
  private function init():void {
   var sprite:Sprite = new Sprite();
   addChild(sprite);
   sprite.graphics.beginFill(0xff0000);
   sprite.graphics.drawCircle(0, 0, 50);
   sprite.graphics.endFill();
   sprite.x = stage.stageWidth / 2;
   sprite.y = stage.stageHeight / 2;
   sprite.addEventListener(MouseEvent.CLICK, onMouseEvent);
   sprite.addEventListener(MouseEvent.DOUBLE_CLICK,onMouseEvent);
   sprite.addEventListener(MouseEvent.MOUSE_DOWN,onMouseEvent);
   sprite.addEventListener(MouseEvent.MOUSE_MOVE,onMouseEvent);
   sprite.addEventListener(MouseEvent.MOUSE_OUT,onMouseEvent);
   sprite.addEventListener(MouseEvent.MOUSE_OVER,onMouseEvent);
   sprite.addEventListener(MouseEvent.MOUSE_UP,onMouseEvent);
   sprite.addEventListener(MouseEvent.MOUSE_WHEEL,onMouseEvent);
   sprite.addEventListener(MouseEvent.ROLL_OUT,onMouseEvent);
   sprite.addEventListener(MouseEvent.ROLL_OVER,onMouseEvent);
  }
  public function onMouseEvent(event:MouseEvent):void {
   trace(event.type);
  }
 }
}
```

请注意，每个事件类型都使用了同一个处理函数，输出所触发的事件类型的名称。

鼠标位置

除了鼠标事件外，对于文档类还有两个非常重要属性用于表示鼠标当前的位置：mouseX 和 mouseY。请注意，影片剪辑的位置，返回的值是鼠标的位置与影片剪辑的注册点的相对位置。例如，有一个名为 sprite 的 Sprite 影片，在舞台的 100,100 位置，而鼠标的位置在 150,250，你会得到如下结果：

mouseX 为 150
mousey 为 250
sprite.mouseX 为 50
sprite.mouseY 为 150

请注意鼠标位置与影片位置的相对关系。

```
 import flash.display.Sprite;
 import flash.events.KeyboardEvent;
 import flash.ui.Keyboard;
 public class KeyCodes extends Sprite {
  private var ball:Sprite;
  public function KeyCodes() {
   init();
  }
  private function init():void {
   ball = new Sprite();
   addChild(ball);
   ball.graphics.beginFill(0xff0000);
   ball.graphics.drawCircle(0, 0, 40);
   ball.graphics.endFill();
   ball.x = stage.stageWidth / 2;
   ball.y = stage.stageHeight / 2;
   stage.addEventListener(KeyboardEvent.KEY_DOWN,onKeyboardEvent);
  }
  public function onKeyboardEvent(event:KeyboardEvent):void {
   switch (event.keyCode) {
    case Keyboard.UP :
     ball.y -= 10;
     break;
    case Keyboard.DOWN :
     ball.y += 10;
     break;
    case Keyboard.LEFT :
     ball.x -= 10;
     break;
    case Keyboard.RIGHT :
     ball.x += 10;
     break;
    default :
     break;
   }
  }
 }
}
```

当我们在 Flash 编辑环境下测试影片时，IDE 会拦截用于控制 IDE 自身的键。Tab 键和所有功能键以及作为快捷菜单项的键，在测试影片时不会接收到。不过，我们可以在菜单中选择“控制” -> “禁用快捷键”，来解除限制。这样一来，测试的影片就像在浏览器中工作一样了。

2.8 小结

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

第三章 三角学应用

从这一章开始，我们将学习三角学，并在第五章开始应用到动画技术中，其实在下一章的绘图技术中就会接触到。如果你已经对三角学有所了解或渴望学习动画方面的知识，那么可以跳过开始这部分，待日后遇到不懂的问题时，再回来学习。我们用到的 90%的三角学都需要 Math.sin 和 Math.cos 这两个函数。在我写本书的第一版时，曾说过，除了在中学学习过的那些代数和几何外(而且由于时间久远大多都记不清了)，我没有接受过正规的数学培训，最初在本章中的内容都是来自于各种书籍，网站或是其它网络资源，这是因为这部分知识并不难，既然我能够学会，那么你也一定可以的。而现在我已经完成了大学代数和微积分课程，对于三角学也有了更为全面和系统的了解。我可以很荣幸地说，这一章的内容非常好，因为对于这个学科有了更为深入的了解，所以很多地方可以解释得更为清楚。

什么是三角学(Trigonometry)

三角学是一门研究三角形与其边和角关系的学科。当我们观察一个三角形时，发现它有三条边和三个角（因此称为三角），而且在这些边和角之间存在着一些特殊的关系。例如，增大其中的任何一个角，那么该角所对应的边就会增长（假设其它两条边长度不变），同时，其它两个角会变小，实际上，究竟它们变化了多少，加以计算后就可以得出一个比例。在一个三角形中，如果其中有一个角为 90 度，那么就称为直角三角形，并在该角的夹角处标出一个正方形（垂足），只有在直角三角形才会这样。学习直角三角形中存在的关系要比推导基本公式简单得多，这使得直角三角形成为一种非常有用的结构，本章及该书后面的内容大多都是直角三角形。

角(Angle)

角是三角学最主要的研究对象，让我们先来解决这个问题。角是由两条相交线构成的图形，或是两条相交线之间的那部分空间，空间远大，夹角越大。事实上，两条相交的线会形成四个角，见图 3-1：

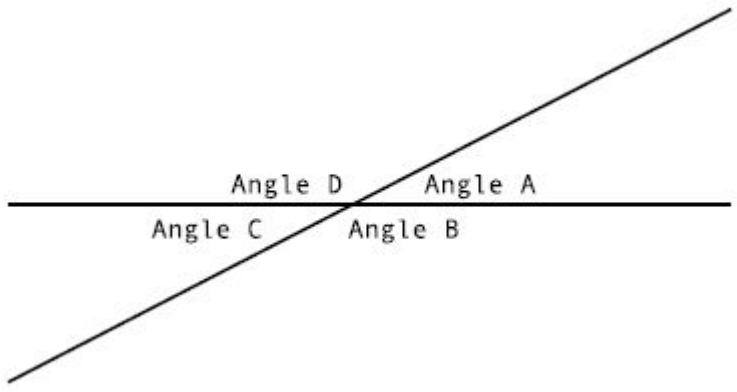


图 3-1 两条线形成四个角

弧度制(radian)与角度制(degress)

弧度制与角度制是角度测量中的两种特殊制度。我们大概对于角度制最为熟悉，甚至闭着眼都能画出 45 度或 90 度的角。圆的 360 度体系已经成为了一种文化，人们常说“180 度

转弯”就是指“转到相反的方向”，这里并不是指转弯的方向，而是指一种相反的观点。我们所讨论的角度，对于计算机来说，就是弧度。所以，不管你是否喜欢，都要对弧度制有所了解。

1 弧度约等于 57.2958 度。你也许会问“这符合逻辑吗？”确实有其逻辑所在。一个圆，360 度，计算出的弧度为 6.2832。仍然没有任何意义？好，想一下圆周率派 Pi(π) 约等于 3.1416，而一个圆(6.2832 弧度)就等于 2 pi。我们知道 360 度相当于 2 pi，180 度相当于 pi，90 度相当于 pi/2，等等。图 3-2 给出一些常用的弧度制。

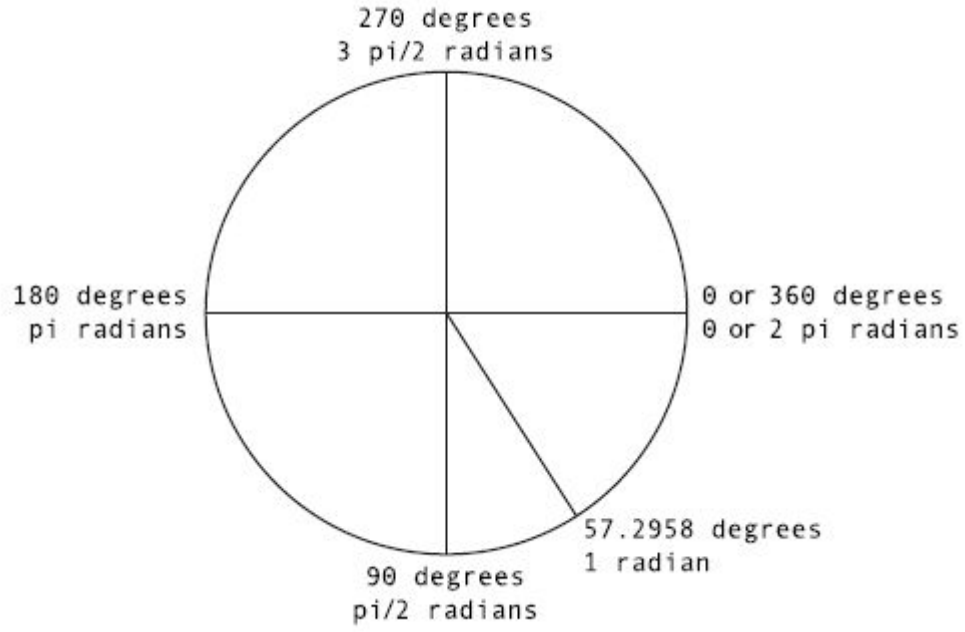


图 3-2 弧度与角度

从现在起我们就要开始使用弧度制了，而且今后会遇到很多用弧度表示度的情况。

影片剪辑和 Sprite 影片的 rotation 属性都要使用角度制，而且属性非常会经常使用。例如，一辆汽车需要旋转到运动的方向，如果使用三角学计算运动方向，那么所得到的角度是以弧度制表示的，而汽车的旋转则需要使用角度制。相反，如果要指定某个对象向某个方向前进，就要获得它的旋转(rotation)角度，而这是用角度制表示的，如果要在三角函数中使用它就一定要转换为弧度制。

角度制，还应用在滤镜上，如果使用投影滤镜(drop shadow filter)，来为物体投射 45 度的阴影，就需要指定其角度而非弧度，不论是在 Flash IED 中还是使用 ActionScript 代码都一样。

为什么在一个编程体系里有两种截然不同的制度呢？也许这就是 Flash 双重性。一方面，这是设计人员的工具，在 Flash IDE 中拥有所有的绘图和变形工具，可以绘制出漂亮的图形。如果你对一名设计员说把你制作的 logo 文字旋转一个弧度，你肯定会遭白眼。另一方面，Flash 也是一个开发工具，更像一种编程语言，ActionScript 用户使用弧度制。总之，不论你是否喜欢都要使用到它们，而且还需要掌握角度制与弧度制间的相互转换。以下是公式：

弧度(radians) = 角度(degrees) * Math.PI /180

角度(degrees) = 弧度(radians) * 180 / Math.PI

在学习本书的过程中，会遇到很多公式。无论哪里，遇到需要记忆的公式时，我都会指出来，希望大家能够识记，这里是第一个公式。每次需要用到这些公式时，可以查找一下，但不会得到现成的代码，因为这些代码都需要用手敲进去。我使用 ActionScript 写这些公式，比如使用 Math.PI 要比使用 pi 或其它字符要好，因为这和我们输入的代码是一致的。

180 度大约等于 3.14…弧度。换句话讲，半圆为 pi 个弧度，整圆为 2 pi 个弧度，一个弧度大概为 57.29…度。

Flash 坐标系

在讨论角度时，就要提到 Flash 坐标系。如果我们习惯于数学坐标系，那么对于 Flash 坐标系可能会有些不习惯，因为在这里一切是颠倒(upside down)的。在标准坐标系中，用 X 表示水平轴，用 Y 表示垂直轴，Flash 也是一样。当 x=0,y=0 时，坐标(0,0)通常显示在中心位置，X 为正数时在右边，X 为负数时在左边，Y 为正数时在上边，Y 为负数时在下边，如图 3-3 所示。

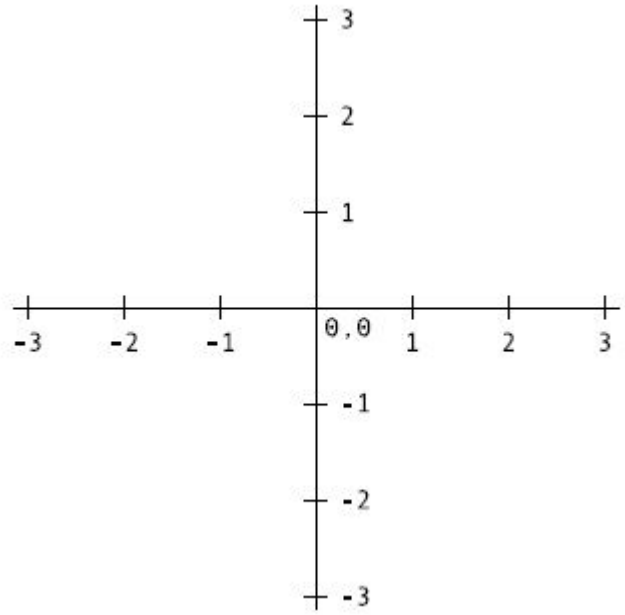


图 3-3 标准坐标系

然而 Flash 是基于视频屏幕的坐标系，0,0 点为左上角，如图 3-4。X 值从左向右不断增大，但 Y 轴是相反的，正值向下，负值向上。这个系统有其历史根源，与屏幕扫描建立图像的原理一样，从左到右，从上到下。

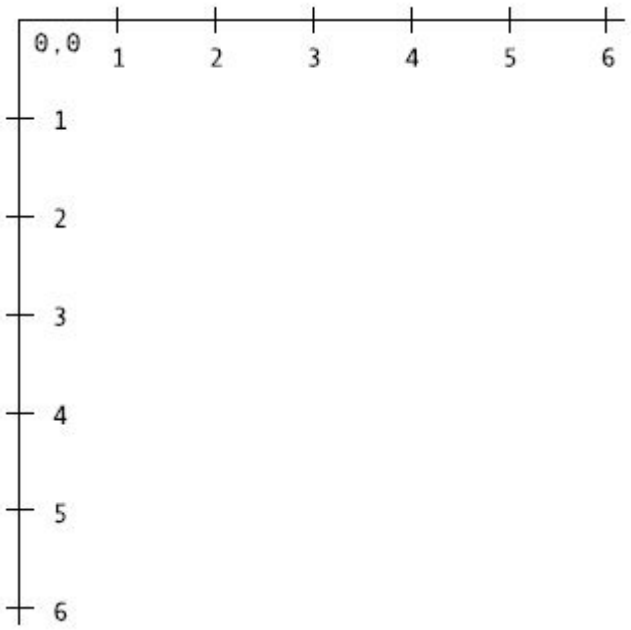


图 3-4 Flash 坐标系

我们可以想像成一个普通的坐标系，只是要把 Y 轴颠倒过来，并把屏幕中心迁移到屏幕的左上角。下面就来说说角。在一般的坐标系中，角度是以逆时针计算的，并以 0 度为起点向正 X 轴引一条线，如图 3-5 所示。

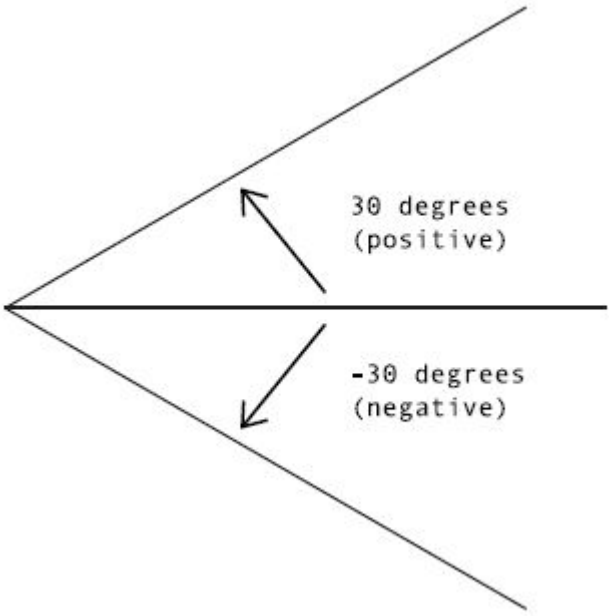


图 3-5 普通的角度

在 Flash 中是颠倒的，如图 3-6 所示。顺时针旋转角度为正角。逆时针就意味着为负角。

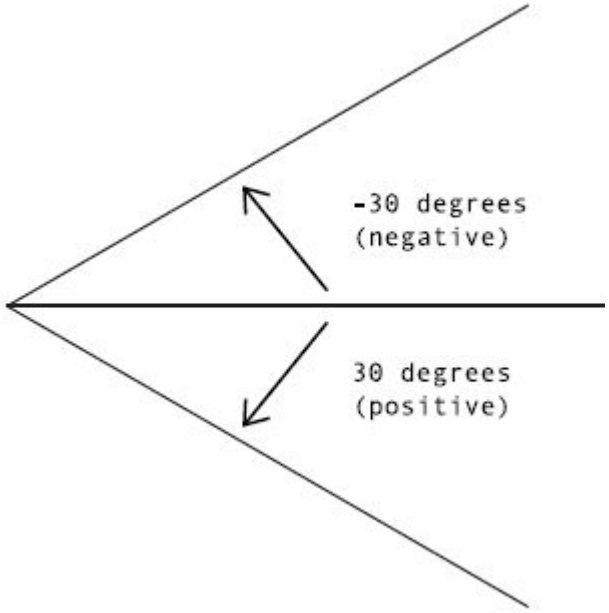


图 3-6 Flash 的角度

三角形的边

对于三角形的边，没有太多可说的，但它们都有各自的术语。以直角三角形为例，如图 3-7 所示，每条边都有各自的名称，与 90 度角相接的两条边称为直角边(legs)，相对的边称为斜边，它总是那个最长的边。

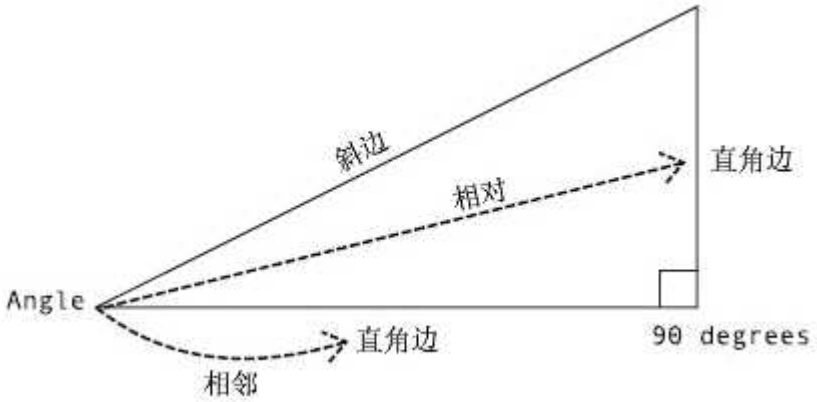


图 3-7 直角三角形各部分

刚才说到对边时，说它是与该角不相接的边。说到邻边时，说它是与角相接的边。在很多例子中，都是与其余两个不是 90 度的角打交道。在三角形中最有趣的就是角与边的关系，这些关系对于动画制作非常有用，下面就让我们来看看。

三角函数

ActionScript 拥有一套用于计算不同三角关系的三角函数：正弦，余弦，正切，反正弦，反余弦和反正切。下面我们就开始定义和使用这些函数，而后还会介绍它们的实际应用。

正弦(Sine)

下面是三角学的第一个部分。一个角的正弦值等于该角的对边与斜边的比，在 ActionScript 中，使用 Math.sin(angle) 函数来表示。图 3-8 所示为一个 30 度角的正弦。对边长为 1，斜边长为 2，两条边的比为 1 比 2，或记作 1/2 或 0.5，因此，我们可以说 30 度角的正弦值为 0.5，下面在 Flash 中测试一下：

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

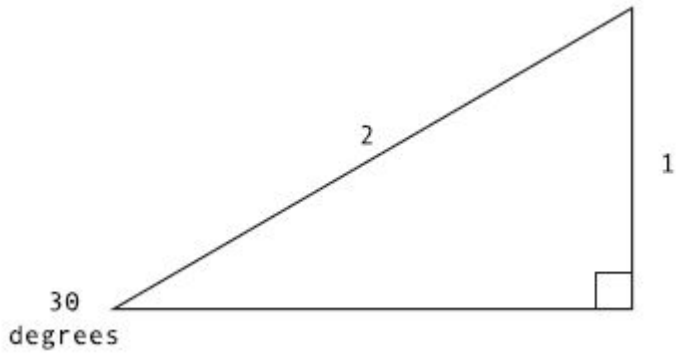


图 3-8 角的正弦值为对边/斜边

输出结果为 -0.988031624092862，为什么会这样，能够找出原因吗？这是因为我们忘记了将结果转换为弧度制。我敢说你以后会常犯这种错误（我也一样），所以一定要小心。以下是正确的写法：

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

成功！输出 0.5

还可能得到 0.4999… 这样的值，这并不是程序的错误，而是由于二进制计算机常以浮点形式表示数值。但这个值已经非常接近了，所以就认为它等于 0.5。

可以把一个三角形想象为角度为 30，两条边长分别为 1 和 2，然后把它移到普通坐标系中，

不要忘了， Flash 坐标系的 Y 轴向下，角度是顺时针的。所以，对边和角度都是相反的，见图 3-9。

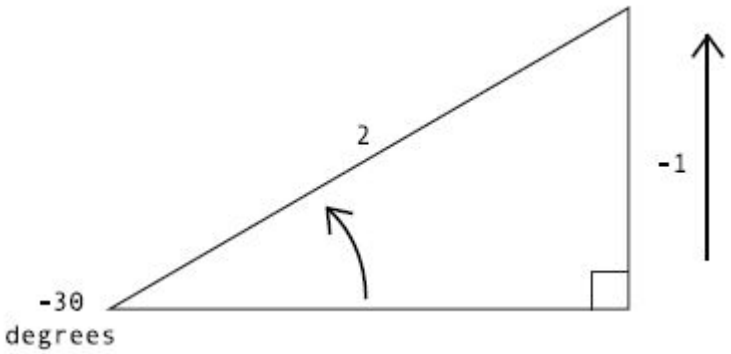


图 3-9 在 Flash 坐标系中创建相同的角

因此，比例也变成了-1/2，我们就称它为-30 度角的正弦值。同时，把表达式改为：

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

好的，不会很痛苦吧？下面再来看一个三角函数：余弦。

余弦(Cosine)

在 Flash 中，使用 Math.cos(angle) 就可以计算余弦值，余弦的定义为角的邻边与斜边之比。见图 3-10。

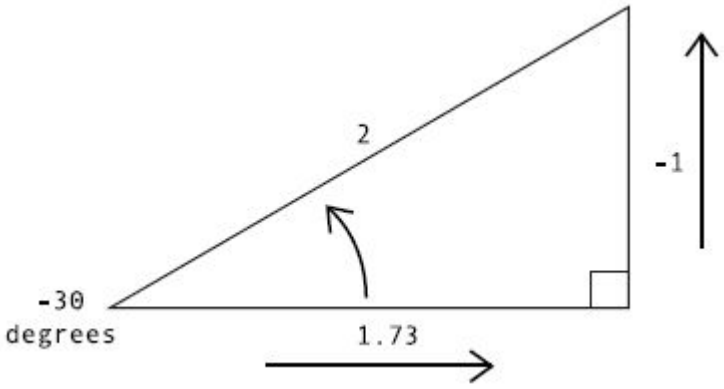


图 3-10 角的余弦值为邻边/斜边

图 3-10 中的角度与图 3-9 中的相同，这次在图中直接加入了邻边的长度 1.73。角的余弦值为 1.73/2，或 0.865。因此，我们可以说-30 度角的余弦值为 0.865，下面测试一下：

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

与使用正弦函数一样，只不过这次调用的是 Math.cos 函数，这次输出结果为 0.866025403784439，非常接近 0.865。之所以会有所不同，是因为我把邻边的值取整了。真正的长度应该近似于 1.73205080756888，用这个数除以 2，那么结果就非常接近-30 度的余弦值。到现在为止，我们所说的都是左下方的角(degrees)。下面来看看右上方的角，首先，需要重新在坐标系中定位该角，这里的坐标系是指 Flash 坐标系，见图 3-11。

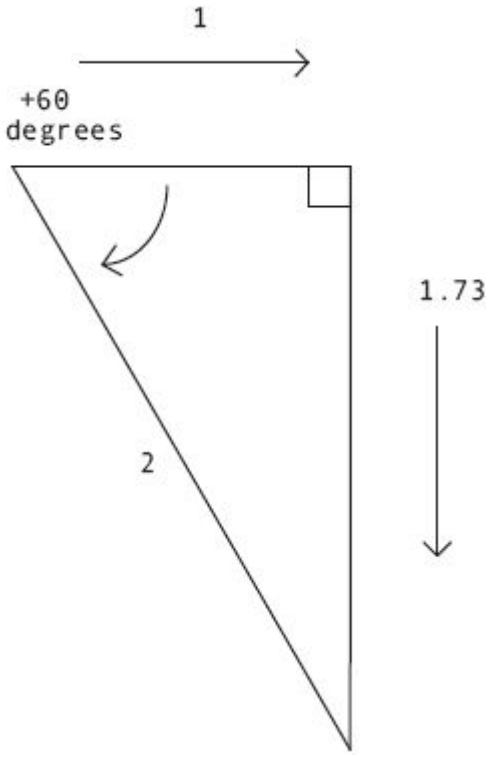


图 3-11 观查对角(opposite angle)

该角的正弦值为对边与斜边之比，或 1.73/2(0.865)，余弦值为邻边与斜边之比，1/2(0.5)。因此就得出，一个角的余弦值等于另一个角的正弦值，请注意它们之间是相互关联，成比的。

正切(Tangent)

另一个重要的三角函数是正切，用 Flash 表示为 Math.tan(angle)。它反应的是对边与邻边之间的关系如图 3-12 所示。

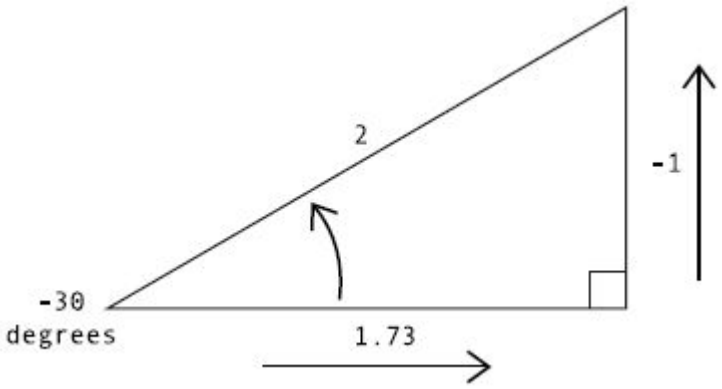


图 3-12 角的正切值为对边/邻边

两者的比例为 -1/1.73 或 -0.578，直接在 Flash 中进行验证，会得到更准确的结果：trace(Math.tan(-30 * Math.PI / 180));

输出结果为 -0.577350269189626，证实了前面的计算。在 ActionScript 中，这个函数并不常用，而使用正弦和余弦的时候要多一些。另外，反正切函数却是非常有用的，后面会讲到，这里请大家记住正切函数的比例关系。

反正弦(Arcsine)和反余弦(Arccosine)

与正切相似，反正弦和反余弦在一般的 Flash 动画中很少使用。然而，我们还是要学习一下它们的用法，实际上就是正弦和余弦函数的反函数。换句话讲，就是输入一个比例值，返回一个角度值(以弧度表示)。

在 ActionScript 函数中记作 Math.asin(ratio) 和 Math.acos(ratio)。下面让来测试一下，我们已经知道 30 度角的正弦值为 0.5，所以 0.5 的反正弦值应为 30 度，检验一下：

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

别忘记将结果转换为角度制，才能得到角度制 30 度，而不是弧度制 0.523。

我们知道，30 度角的余弦值大约为 0.865，下面以同样的方法来测试一下：

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

得到结果为 30.1172947473221。如果把 30 度的余弦值输入得更准确，那么所得的结果也会更为精确。怎么样，不难吧？

反正切(Arctangent)

大家可能都猜到了，反正切简单地说就是正切函数的反函数。我们只要输入对边与邻边的比值，就可以得到相应的角度。

在 Flash 中有两个函数可计算反正切。第一个就是像前面介绍过的函数一样 Math.atan(ratio)，只需提供对边与邻边的比例值。例如，前面学过 30 度角的正切值约为 0.577。试一下：

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

输出结果是一个近似 30 的数，不是非常直观易懂吗，为什么还需要另一个函数呢？下面请看图 3-13，让它来回答：

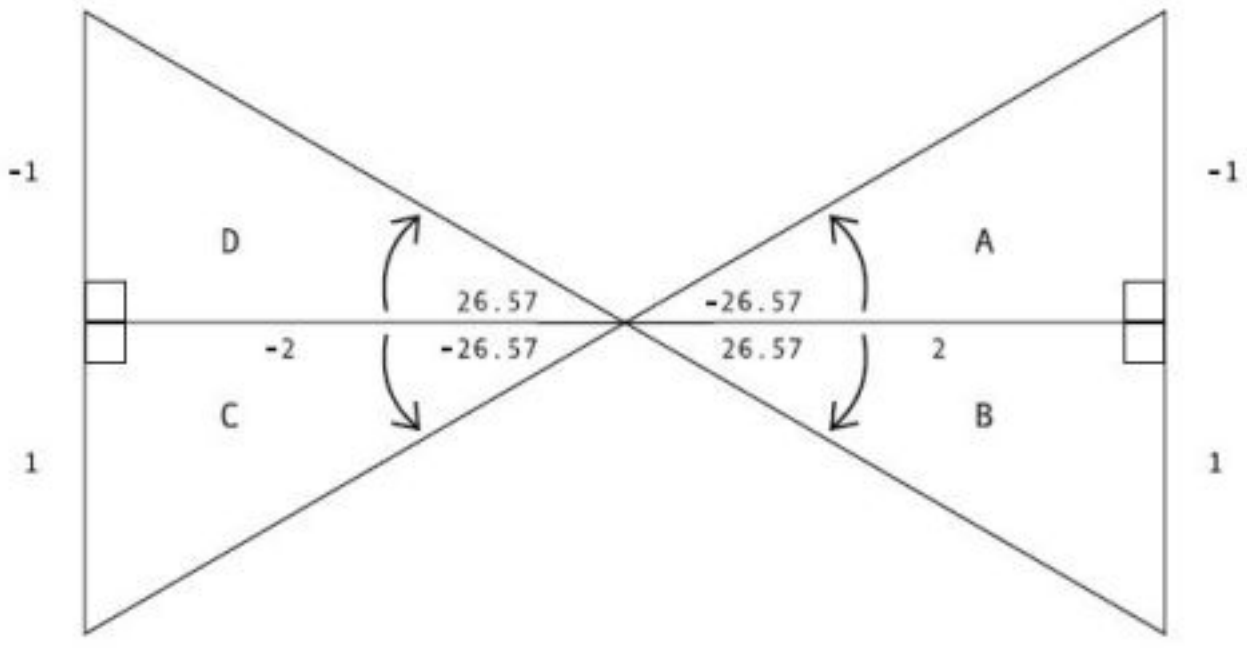


图 3-13 四个象限上的角

如图 3-13 所示，有四个不同的角：A，B，C，D。角 A 和 B，在 X 轴上为正数，角 C 和 D 在 X 轴上为负数，同样，角 A 和 D 在 Y 轴上为负数，而角 B 和 C 在 Y 轴上为正数。因此，四个内角的比例分别为：

A: -1/2 (-0.5)

B: 1/2 (0.5)
C: 1/ -2 (-0.5)
D: -1/ -2 (0.5)

对边与邻边之比为 0.5，输入 Math.atan(0.5)，并转换为角度制，结果大约为 26.57，那么究竟所指的是角 B 还是角 D 呢？两个比例都为 0.5 那样就无法分辨了，看似是个小问题，但对于日后的工作确有很大的影响。

下面有请 Math.atan2(y,x)，这是 Flash 的另一个反正切函数，它比 Math.atan(ratio)要有用得多。实事上，只需要学会这个函数的用法就可以了，函数中包括两个参数：对边长度与邻边长度。有时常会误写成 x,y，请注意应该是 y,x。请看如下示例，输入 Math.atan2(1,2)，然后记住这个结果：

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

输出结果为 26.565051177078，这正是角 B 的度数。下面再输入-1/-2(角 D)，再来试试：

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

出乎意料的结果 - 153.434948822922. 为什么会这样？图 3-14 能给你解释。

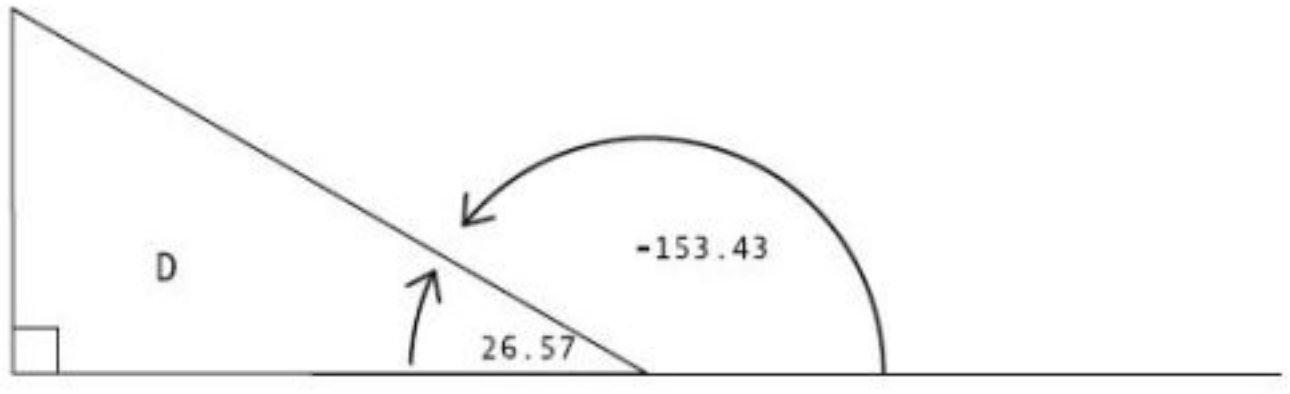


图 3-14 一个角的两种表示方法

从角 D 自身的底边开始，它确实为 26.57 度，但别忘了 Flash 的角度是从 X 轴的正半轴顺时针计算的。因此，从 Flash 的角度来衡量，则该角被视为-153.43 度。下面就要开始在 Flash 中实践和应用三角学了。

旋转(Rotation)

我们想让一个影片剪辑或 Sprite 影片通过旋转来指向鼠标的位置，这将是个挑战。旋转(rotation)将成为我们工具箱中非常的工具，可以应用于游戏制作，鼠标追踪，界面设计等。

下面看一个示例。也可以根据以下步骤或打开文档类 RotateToMouse.as 和 Arrow.as(与本书中其它代码一同在 www.friendsofed.com 下载)，这些是已写好的代码。首先，需要让物体旋转，它可以是一个在 Sprite 中绘制的箭头(Arrow)。事实上，如果我们要反复应用到这个箭头，可以把它制作成一个类：

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

```
        graphics.lineStyle(1,0,1);
        graphics.beginFill(0xffff00);
        graphics.moveTo(-50,-25);
        graphics.lineTo(0,-25);
        graphics.lineTo(0,-50);
        graphics.lineTo(50,0);
        graphics.lineTo(0,50);
        graphics.lineTo(0,25);
        graphics.lineTo(-50,25);
        graphics.lineTo(-50,-25);
        graphics.endFill();
    }
  }
}
```

这里使用到了绘图 API (会在下一章介绍)来绘制箭头。无论何时需要一个箭头，只需写一句 new Arrow()即可，在图 3-15 中可看到显示结果。当绘制一些图像并进行旋转时，要注意它的指向，默让地指向右边，X的正半轴，这就是它旋转到 0 度时的状态。

我们先要创建一个Arrow类的实例，放致于舞台中心，并让它指向鼠标的方向，如图3-16。

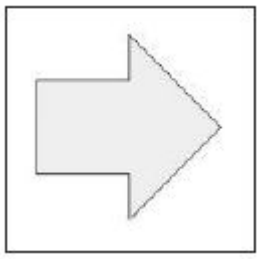


图 3-15 使用绘图API绘制的箭头

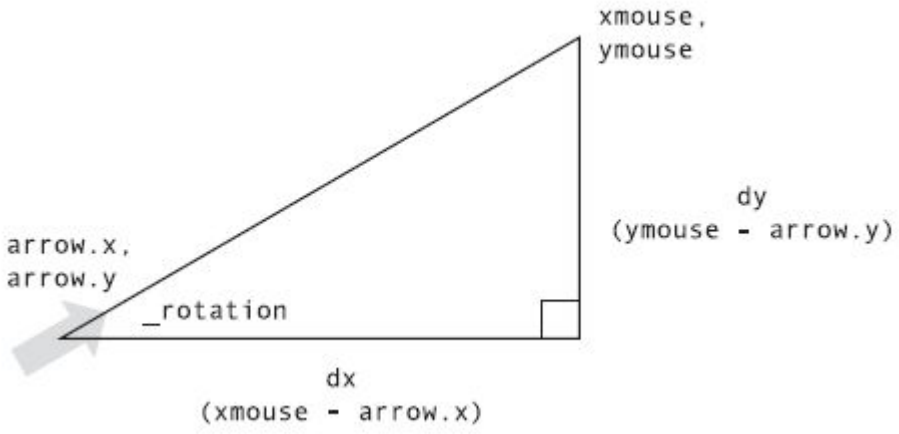


图 3-16 下一次需要计算的值

很熟悉吗？与我们之前所讲的三角形相同，只不过多加入了鼠标与箭头的坐标。鼠标的位置只需使用 mouseX 和 mouseY 属性即可获得，同样，使用x,y属性，获得箭头的位置。使它们的值相减，就得到了两条边的长度。现在只需要使用 Math.atan2(dy,dx) 就可以求出夹角，然后把结果转换为角度制，最后让箭头的 rotation 属性等于这个夹角。代码如下：

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

当然，为了使之形成一个动画，还需要加入循环。如同前一章提到的，使用事件处理函数将

会是最好的选择，请使用 enterFrame 事件。以下是这个完整的文档类：

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

请确认 RotateToMouse.as 文件与 Arrow.as 文件在同一目录下，以 RotateToMouse 作为文档类，并为它创建 SWF。怎么样？就像施了魔法一样！假设如果我们没有 Math.atan2 这个函数，就要先通过，dy除以dx求出对边与邻边的比值，然后再写入 Math.atan 函数。下面用 Math.atan 函数来代替 Math.atan2 函数来试一下，代码如下：

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

试试这种写法，马上就会发现问题。如果鼠标位于箭头的左侧，箭头不会指向鼠标，并与鼠标相背离。能说说为什么吗？回到有 A,B,C,D 四个角的图(图 3-13)，不要忘记角A和C拥有相同的比值，角B和D也是一样。这样一来， Flash 就无法知道所指的是哪个角，所以只能得到A与或角B。如果，鼠标处于D角区域，Flash 会回到B角区域并把箭头指向这个角度。毫无疑问，这时 Math.atan2 的好处就显示出来了，书中会经常用到这个函数。

波形

大家肯定听说过正弦波，也一定见过图 3-17 所示的图形。

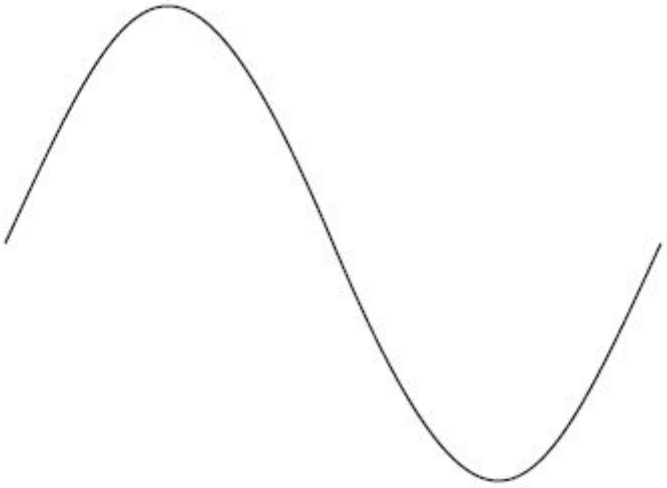


图 3-17 正弦波形

那么为什么要把正弦函数与正弦图像两个不相干的东西联系到一起呢？如果将 0 到 360 度(或着 0 到 2pi)代入到正弦函数中，那么就会得到这个正弦函数图像。从左到右代表所使用的角度值，而图中 y 坐标变化，代表这些角的正弦值。

图 3-18 中，标出了一些特殊的角度，我们可以看到 0 度的正弦值为 0，90 度或 pi/2 的正弦值为 1，180 度或 pi 的正弦值又回到 0，270 度或 3/2pi 的正弦值为-1，360 度的正弦值为 0。下面用 Flash 来试一下正弦波形，把以下代码放入文档类的框架中进行测试：

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

从现在起，要开始习惯只使用弧度制。除了使用 rotation 或其它只使用角度制的属性外，要开始学着不去使用角度制。

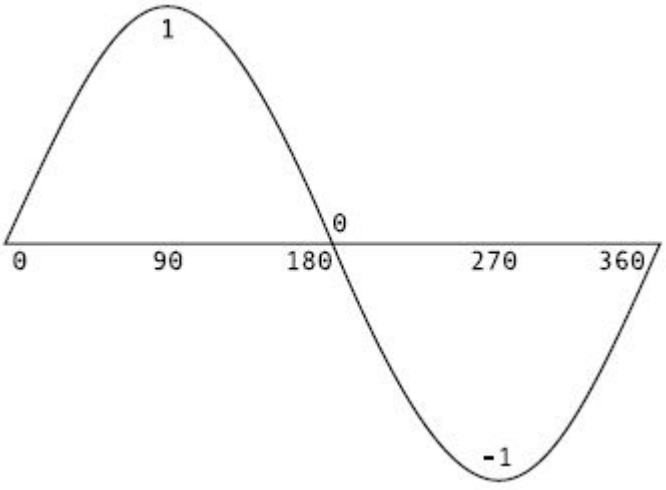


图 3-18 正弦图像值

在这个例子中，角度从 0 开始，每次递增 0.1 直到大于 Math.PI*2 为止，并输出该角的正弦值。看一下输出结果，我们发现角度是从 0 开始，增加到 1 后，开始减小，减少到-1 时，再回归至 0。这些值不会真正准确地到达 1 或 0，因为每次增加 0.1，所以永远不会得到 pi 或 pi/2 的整数倍。

平滑的上下运动

如何使用 Math.sin(angle) 呢？如果想让物体上下或左右移动，那么就要用到这个函数。考虑：使用 0～1～-1～0 的变化来实现这个动画，并且反复地使用这个波形。

活动域仅仅是从 1 到-1，不能看出效果，所以要把这些数值放大一些，比如扩大 100 倍。这样就拥有了一个从 100 到-100 的波形，并且连绵不断。在下面这个文档类 Bobbing.as 中，要使用一个在 Ball 类中定义的 Sprite 影片，请看代码：

```
package {
 import flash.display.Sprite;
 public class Ball extends Sprite {
  private var radius:Number;
  private var color:uint;
  public function Ball(radius:Number=40, color:uint=0xff0000) {
   this.radius = radius;
   this.color = color;
   init();
  }
  public function init():void {
   graphics.beginFill(color);
   graphics.drawCircle(0, 0, radius);
   graphics.endFill();
  }
 }
}
```

当这个类被实例化后，就能绘制出一个圆。我们还可以自行给出圆的半径(radius)和颜色(color)。如果不给这些参数的话，就会使用默认的参数：半径为 40，颜色为红色（这是 AS3.0 新增的功能）。这个类非常简单，但却非常有用，今后在书中会经常用到，所以大家一定要掌握。

文档类创建一个 Ball 类的实例，并加入到舞台上，再为它增加一个 enterFrame 侦听器，这样就可以让小球上下移动了。

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

首先，需要一个角度属性(angle)初始值为 0。在 onEnterFrame 方法中，使用该角的正弦值并扩大 50 倍。这样一来，取值的范围就变成了 50 到-50。再在这个值上加舞台高度的一

半，数值就变为从 250 到 150(设舞台高度为 400 像素)，用这个值作为小球的 Y 坐标，最后为下一次循环增加 0.1 个弧度，这样就完成了小球平滑的上下运动。每一次循环的值都不相同，我们发现如果将 0.1 变为另一个数值的话，就改变了小球运动的速度。角度(angle)变化的快慢与 Math.sin 从 1 到-1 变化的速度成正比。很明显，改变 50 这个值，就改变了小球移动的距离，而改变 stage.stageHeight / 2 的值，就改变了小球运动时围绕的位置。我们还可以给出一些抽象的值作为变量，代码如下(只给出需要改变或增加的部分)：

```
// at the top of the class:
private var angle:Number = 0;
private var centerY:Number = 200;
private var range:Number = 50;
private var speed:Number = 0.1;
// and the handler function:
public function onEnterFrame(event:Event):void {
 ball.y = centerY + Math.sin(angle) * range;
 angle += speed;
}
```

在运动代码中没有使用具体的数值，真是次非常好的练习，以后应尽量这样做。

线性垂直运动

在 Wave1.as 文件中，加入了线性垂直运动，只是为我们制作动画增加一些灵感。以下是这个文件的代码：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Wave1 extends Sprite {
  private var ball:Ball;
  private var angle:Number = 0;
  private var centerY:Number = 200;
  private var range:Number = 50;
  private var xspeed:Number = 1;
  private var yspeed:Number = .05;
  public function Wave1() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   ball.x = 0;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  public function onEnterFrame(event:Event):void {
   ball.x += xspeed;
   ball.y = centerY + Math.sin(angle) * range;
   angle += yspeed;
  }
```

```
 }
}
```

心跳运动

使用正弦值作为一种工具，不仅仅只用于控制物理位置。在 Pulse.as 文件中，使用一个值来影响小球的缩放，实现一个心跳的效果，代码如下：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Pulse extends Sprite {
  private var ball:Ball;
  private var angle:Number = 0;
  private var centerScale:Number = 1;
  private var range:Number = .5;
  private var speed:Number = .1;
  public function Pulse() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   ball.x = stage.stageWidth / 2;
   ball.y = stage.stageHeight / 2;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  public function onEnterFrame(event:Event):void {
   ball.scaleX = ball.scaleY = centerScale +

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

原理是一样的，centerScale 表示 100%的缩放比， range 表示范围， speed 表示速度。不仅如此，正弦波还在 alpha，rotation 等属性中应用。

双角波形

再给大家一种思想：设置两套数值 angle1 和 angle2，为它们各自增加一个中心点(center)和速度(speed)值。用一个正弦波作为一种属性，另一个正弦波作为另一种属性，比如位置或缩放。我不敢保证能够得到什么有用的结果，但这样做的话，就等于让这些函数自由发挥作用。

从 Random.as 文档类开始，这里面拥有两个角度(angle)，两个速度(speed)和两个中心点(center)，将其中一个角(angle1)作为小球的 X 坐标，另一个角(angle2)作为 Y 坐标。运行程序时，就像只虫子在房间里飞，虽然这些数字都是预先定义好的，但结果却没有什么规律可言。代码如下：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Random extends Sprite {
  private var ball:Ball;
  private var angleX:Number = 0;
  private var angleY:Number = 0;
  private var centerX:Number = 200;
  private var centerY:Number = 200;
  private var range:Number = 50;
  private var xspeed:Number = .07;
  private var yspeed:Number = .11;
  public function Random() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   ball.x = 0;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  public function onEnterFrame(event:Event):void {
   ball.x = centerX + Math.sin(angleX) * range;
   ball.y = centerY + Math.sin(angleY) * range;
   angleX += xspeed;
   angleY += yspeed;
  }
 }
}
```

绘制波形

最后，在 Wave2.as 中，不再使用小球，转而使用绘图 API 来绘制正弦波形。代码如下：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Wave2 extends Sprite {
  private var angle:Number = 0;
  private var centerY:Number = 200;
  private var range:Number = 50;
  private var xspeed:Number = 1;
  private var yspeed:Number = .05;
  private var xpos:Number;
  private var ypos:Number;
  public function Wave2() {
   init();
  }
  private function init():void {
```

```
    xpos = 0;
    graphics.lineStyle(1, 0, 1);
    graphics.moveTo(0, centerY);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  public function onEnterFrame(event:Event):void {
    xpos += xspeed;
    angle += yspeed;
    ypos = centerY + Math.sin(angle) * range;
    graphics.lineTo(xpos, ypos);
  }
 }
}
```

下一章我们会详细讲述绘图 API，大家也应该有兴趣来执行一下这个文件，观察一下绘制出的波形。注意，由于 Flash 的 Y 轴是反向的，所以绘制出的波形也是颠倒的。

圆和椭圆

目前为止我们已经掌握了正弦波，下面再来看看它的兄弟，余弦波。与正弦波的形成相同，只不过是使用余弦函数代替了正弦函数而已。如果你还记得前面所说的正弦和余弦是怎样一种相反关系的话，就能理解，它们只是波形都相同，只是所处位置不同了。图 3-19 为余弦波图像：

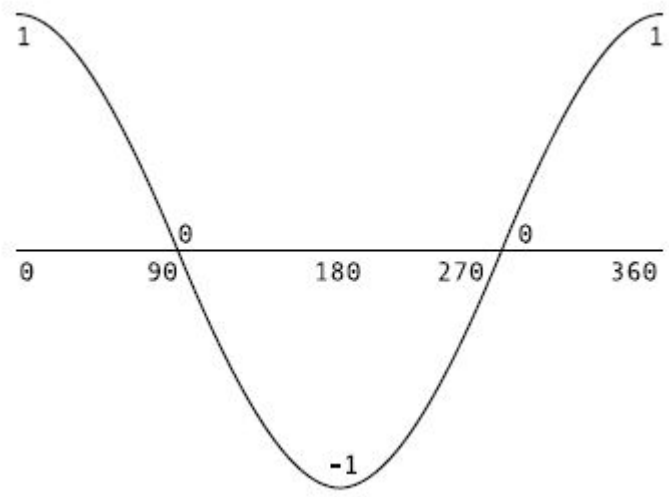


图 3-19 余弦波形

可见，余弦图像中 0 度和 2pi 度(或 360 度)的值都为 1，从 1 开始经过 0,-1,0, 最后回到 1。所以，它与正弦曲线相同，只不过位置发生了一点偏移。

圆形运动

在执行物体移动的动画时，完全可以使用余弦来代替正弦。实际上，余弦和正弦协同工作时，才能形成一个更加有用的功能：使物体沿圆形运动，如图 3-20。

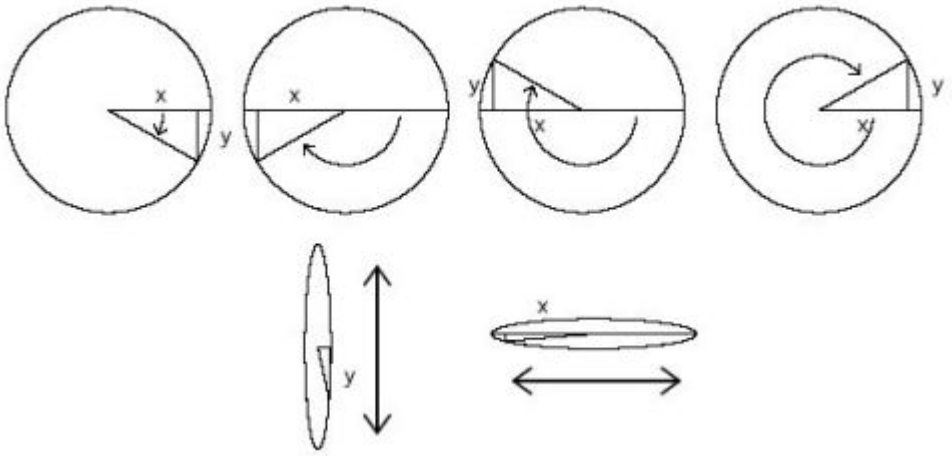


图 3-20 物体沿圆形运动时的几个点

如图 3-20 所示，以图中的圆为例，盯住右侧的那直角条边(y)，然后开始对它进行旋转，我们发现这条边正在被倒置。这条边的中心点就是圆心，而它的运动范围就是这个圆的半径。就像在第一个正弦实验中一样，我们可以计算出这条边的长度：角的正弦值乘以半径。在这里，使用正弦函数非常合适，当我们从侧面观察这个圆时，就可以算出 y 的长度——对边的长度。如果把这个圆放倒，再来观察它，发现角是在向前向后或向左向右移动的。这时，可以使用余弦函数计算出 x 的长度——邻边的长度。重要的一点是，两个的夹角都是相同的，而不像 Random.as 那个例子中使用不同的角度计算 x,y 坐标。这里我们只需要记住用正弦函数计算 y ，用余弦函数计算 x。下面请看 ActionScript 代码：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Circle extends Sprite {
  private var ball:Ball;
  private var angle:Number = 0;
  private var centerX:Number = 200;
  private var centerY:Number = 200;
  private var radius:Number = 50;
  private var speed:Number = .1;
  public function Circle() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   ball.x = 0;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  public function onEnterFrame(event:Event):void {
   ball.x = centerX + Math.cos(angle) * radius;
   ball.y = centerY + Math.sin(angle) * radius;
   angle += speed;
  }
 }
}
```

大家可以自己写这个例子，也可以打开 Circle.as 作为文档类。执行后发现，获得了一个完美的圆。这段代码的精华就是使用余弦来确定 x 坐标，使用正弦来确定 y 坐标，你

应该对他们的关系非常了解了。在 Flash 中，只要提到 x ，你就应该马上想到余弦，并且还能联想到 y 使用正弦。请在最后这段代码上多花些时间，它将是 ActionScript 动画工具箱中最有用的工具之一。

椭圆运动

想要获得一个椭圆该怎么办呢，其实很简单，问题就在于半径。如果让 x 和 y 运动的大小相同，那么就得到一个圆。如果想得到一个椭圆形，我们只需要在计算 x 和 y 位置时使用不同的半径值： radiusX 和 radiusY 。从严格的几何观点来看，使用这两个名称实在不怎么好，但是它们确实非常简单易懂，也非常好记非常直观，所以我还是坚持使用这两个变量名。下面看看它们是如何配合的，见 Oval.as：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Oval extends Sprite {
  private var ball:Ball;
  private var angle:Number = 0;
  private var centerX:Number = 200;
  private var centerY:Number = 200;
  private var radiusX:Number = 200;
  private var radiusY:Number = 100;
  private var speed:Number = .1;
  public function Oval() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   ball.x = 0;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  public function onEnterFrame(event:Event):void {
   ball.x = centerX + Math.cos(angle) * radiusX;
   ball.y = centerY + Math.sin(angle) * radiusY;
   angle += speed;
  }
 }
}
```

这里， radiusX 为 200，意味着小球在距离 centerX 200 个像素内左右运动。 radiusY 为 100，意味着小球上下运动的范围只有 100 像素，这样就得到了一个不匀称的圆。

勾股定理

最后，介绍一下勾股定理。虽然并不能算是三角学中正式的一部分，但是它与我们这个学科还是有一些关系的，并且还涉及到一个将来会经常使用的公式，所以在这里介绍它非常合适。

勾股定理是很久以前一个希腊人发明。这个定理是说 A 的平方 +B 的平方 =C 的平方 ，

听起来好像是儿歌，如果大家之前学过这个定理，那么交流起来效果最好。

深入探讨一下，另一种对该定理的叙述是：直角三角形的两条直角边的平方和等于斜边的平方，这句话真正说到点子上了。请看图 3-21 所示直角三角形。

两条直角边 A 和 B 长度为 3 和 4，斜边 C 长度为 5。毕达哥拉斯(Pythagoras)先生告诉我们 $A^2 + B^2 = C^2$ 。加入一些数字来检验一下，$3^2 + 4^2 = 5^2$，计算出 9 + 16 = 25。是的，非常正确。

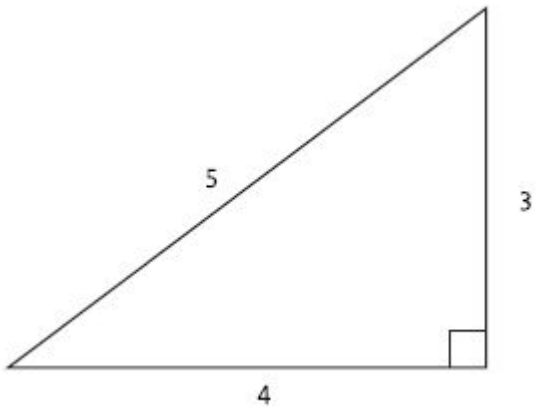


图 3-21 一个直角三角形

如果大家已经知道了这个口诀，那么勾股定理只不过就是一种有趣的关系。如果中知道其中两条边的长度，那么勾股定理就派上用场了，可以用它很快地求出第三条边的长度。在 Flash 中，最常见的情况是我们只知道两条直角边的长度要求出斜边的长度。比如，求出两点间的距离。

两点间距离

假设在舞台上有两个 Sprite 影片，想要求出它们之间的距离。这是勾股定理在 Flash 中最为常见的应用。那么如何实现呢？已知两个 Sprite 的 x,y 坐标，把第一个影片的位置称为 x1,y1，另一个影片的位置称为 x2,y2，见图 3-22。

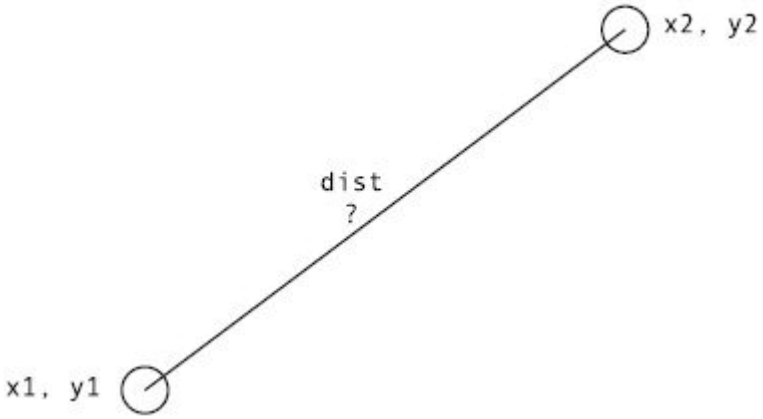


图 3-22 两个物体间的距离是多少？

如果你在本章中看了太多的直角三角形，那么很容易就把图 3-22 看成一个直角三角形，而那条距离线(distance)就是三角形的斜边。在图 3-23 中，加入了这个三角形并填入了数字。

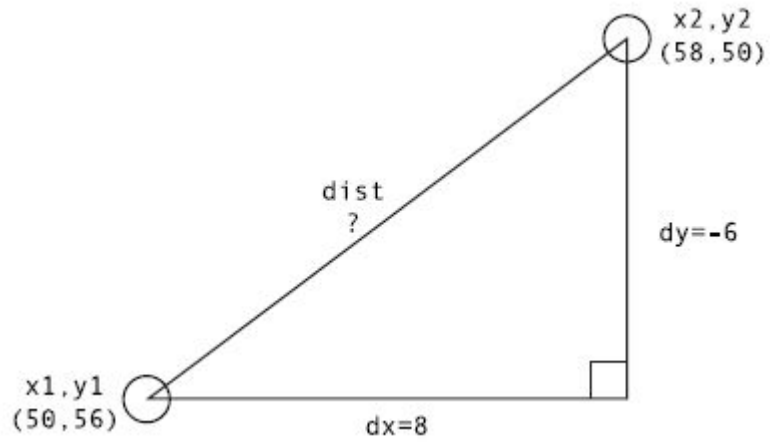


图 3-23 变成一个直角三角形

dx 为两个影片之间的 x 轴，dy 为它们之间的 y 轴。用 x2 减 x1 就得到了 dx 的值：58 - 50 = 8，同样，用 y2-y1 等于 6 得到 dy 的值。现在使用勾股定理，将 dx,dy 的平方相加，就得到了距离(distance)的平方。

换言之，62 + 82 = dist2，相当于 36 + 64(=100) = dist2。基础代数学讲过可以通过开平方把它转化为 = dist。这样一来，就可以得出两个影片之间的距离为 10。

现在，把它抽像成一个公式，这样的话，今后再遇到同样的问题，就可以直接使用这个公式了。有两个位置 x1,y1 和 x2,y2，先计算出 x 的距离和 y 的距离，然后求出它们的平方和，最后求出平方根，下面请看 ActionScript 写法：

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

请特别注意这些代码，它们将是我们工具箱中又一个最好的工具。前两句是获得 x,y 轴上的距离。最后一句分为三个步骤：计算每个值的平方，把它们相加，求出平方根。下面进行一下实践，文档类 Distance.as ，创建了两个 Sprite 影片，并随机摆放，最后计算出它们之间的距离。

```
package {
 import flash.display.Sprite;
 public class Distance extends Sprite {
  public function Distance() {
   init();
  }
  private function init():void {
   var sprite1:Sprite = new Sprite();
   addChild(sprite1);
   sprite1.graphics.beginFill(0x000000);
   sprite1.graphics.drawRect(-2, -2, 4, 4);
   sprite1.graphics.endFill();
   sprite1.x = Math.random() * stage.stageWidth;
   sprite1.y = Math.random() * stage.stageHeight;
   var sprite2:Sprite = new Sprite();
   addChild(sprite2);
   sprite2.graphics.beginFill(0xff0000);
   sprite2.graphics.drawRect(-2, -2, 4, 4);
   sprite2.graphics.endFill();
   sprite2.x = Math.random() * stage.stageWidth;
```

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

编译执行这个动画后，就得到了两个影片之间的距离。每次执行，两个影片的位置都会不同。不论它们处于什么位置，我们所获得的距离都是正数。有趣吧，但是还不够动态，在下面这个示例，可以实时地获得影片的距离，请试一下这个文档类，MouseDistance.as：

```
package {
 import flash.display.Sprite;
 import flash.events.MouseEvent;
 import flash.text.TextField;
 public class MouseDistance extends Sprite {
  private var sprite1:Sprite;
  private var textField:TextField;
  public function MouseDistance() {
   init();
  }
  private function init():void {
   sprite1 = new Sprite();
   addChild(sprite1);
   sprite1.graphics.beginFill(0x000000);
   sprite1.graphics.drawRect(-2, -2, 4, 4);
   sprite1.graphics.endFill();
   sprite1.x = stage.stageWidth / 2;
   sprite1.y = stage.stageHeight / 2;
   textField = new TextField();
   addChild(textField);
   stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
  }
  public function onMouseMove(event:MouseEvent):void {
   graphics.clear();
   graphics.lineStyle(1, 0, 1);
   graphics.moveTo(sprite1.x, sprite1.y);
   graphics.lineTo(mouseX, mouseY);
   var dx:Number = sprite1.x - mouseX;
   var dy:Number = sprite1.y - mouseY;
   var dist:Number = Math.sqrt(dx * dx + dy * dy);
   textField.text = dist.toString();
  }
 }
}
```

在这里 dx 和 dx 的值是用 sprite1 的位置减去当前鼠标位置得出的，dist 的值放入一个文本框中进行显示，并在影片和鼠标之间绘制一条线(在下一章绘图 API 中会学到)。最后，将所有这些代码放到处理函数 onMouseMove 中，每次鼠标移动时进行刷新。测试一下这个文件，并移动鼠标，鼠标与影片剪辑之间会连接上一条线，并实时读取线的长度。

后面的章节中，在学到碰撞检测时，我们会发现内置的碰撞检测(hit testing)方法存在着先天不足，然后会看到使用勾股定理公式完成基于距离(distance-based)碰撞检测方法。它还非常适合用于计算重力或弹力等，因为这些力的大小与两个物体之间的距离成正比。

本章重要公式

现在我们已经有了一个全新的工具箱，同时又多了不少工具，全部所有的工具将会在第 19 章列出，那么让我们看看现在都有了哪些工具。注意，这些公式非常地抽象和简化，里面不包括数据类型和变量定义，在类中使用这些公式时，是否使用这些给出的句型取决于你。

基本三角函数的计算：

角的正弦值 = 对边 / 斜边
角的余弦值 = 邻边 / 斜边
角的正切值 = 对边 / 邻边

角度制与弧度制的相互转换：

弧度 = 角度 * Math.PI / 180
角度 = 弧度 * 180 / Math.PI

向鼠标旋转(或向某点旋转)：

```
// substitute mouseX, mouseY with the x, y point to rotate to
dx = mouseX - sprite.x;
dy = mouseY - sprite.y;
sprite.rotation = Math.atan2(dy, dx) * 180 / Math.PI;
```

创建波形：

```
// assign value to x, y or other property of sprite or movie clip,
// use as drawing coordinates, etc.
public function onEnterFrame(event:Event){
value = center + Math.sin(angle) * range;
angle += speed;
}
```

创建圆形：

```
// assign position to x and y of sprite or movie clip,
// use as drawing coordinates, etc.
public function onEnterFrame(event:Event){
xposition = centerX + Math.cos(angle) * radius;
yposition = centerY + Math.sin(angle) * radius;
angle += speed;
}
```

创建椭圆：

```
// assign position to x and y of sprite or movie clip,
// use as drawing coordinates, etc. public function
onEnterFrame(event:Event){ xposition = centerX + Math.cos(angle) * radiusX;
yposition = centerY + Math.sin(angle) * radiusY; angle += speed; }
```

计算两点间距离：

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

3.10 小结

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

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

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

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

第四章　渲染技术

前一章面所看到的绘图示例中，只使用了一些非常简单的绘图指令，前面我们也几次提到了这个神秘的"drawing API"，但没有加以详细的解释。本章我们将学习使用 ActionScript 创建视觉元素，其中包括 ActionScript 的颜色，绘图 API ，ColorTransform 类，滤镜(filter)和 BitmapData(位图) 类。在本章的很多地方都会用到颜色，那么就先来学习第一课吧。

Flash 中的颜色

在 Flash 中，颜色就是一串特殊的数字，一个颜色值可以是 0 到 16,777,215 中的任意数值，这就是 24 位(bit)色彩。也许大家会问，为什么有 16,777,216 (256＊256＊256) 种颜色值，因为 Flash 使用 RGB 颜色值，意味着每个颜色都可以由红(red)，绿(green)，蓝(blue)三种颜色构成。这三种合成色的每一种都是一个从 0 到 255 中的数，所以，对于每个红黄蓝都有 256 种可能的深度，结果会有约 1,678 万种颜色。

这个系统之所以叫做 24 位颜色是因为要使用 8 位(0 或 1)才能表示 256 个数值。8 位乘以 3 (红,黄,蓝)意味着需要 24 位才能表示 1678 万种颜色值。我们马上还要学到 32 位色系统，它有额外的 8 位数值表示透明度(alpha)。

很难想像一个值为 11,273,634 的颜色是什么样的。因此，开发人员通常采用另一种数值表示系统：十六进制。如果大家在 HTML 中使用过颜色，那么这对于你来说并不会陌生，但不管怎样还是让我们来学习一下这些基础知识吧。

使用十六进制表示颜色值

十六进制(Hexadecimal，简写 hex)，以 16 为基础，每位数都是 0 到 15 中的任意一个数，而十进制则是以 10 为基础，每位数都是 0 到 9 中的任意一个数。由于没有可以表示 10 到 15 的数，所以要借用字母表的前六个字母， A 到 F，来表示它们。这样，每个 16 进制数都可以是 0 到 F 中的一个(在 Flash 中，十六进制数不区分大小写，使用 A 到 F 或 a 到 f 均可)。在 HTML 中使用 16 进制数，要加上 # 作为前缀加以标识。与其它语言一样，在 ActionScript 中，使用 0x 作为前缀。比如，十六进制的 0xA 与十进制的 10 相等，0xF 等于 15，0x10 等于 16。在十进制中，每一位都是它右边一位数的十倍，如 243 表示为 2 的 100 倍，4 的 10 倍，3 的 1 倍。在十六进制中，每一位都是它右边一位数的十六倍，如 0x2B3 表示为 2 的 256 倍，B(或 11)的 16 倍，3 的 1 倍。

对于 24 位来说，就等于 0xFFFFFF，此外，这 6 个十六进制数可以分为三部分。第一部分代表红色，第二部分代表绿色，最后两位表示蓝色，被象征性地记为 0xRRGGBB。

记住每一个合成色都可以为 0 至 255(十六进制表示：0x00 到 0xFF)中的值。因此，红色可以表示为 0xFF0000，表示纯红色，因为它的绿色为 0，蓝色为 0。同样，0x0000FF 表示纯蓝色。

拿 11,273,634 为例，将它转换为十六进制（稍后为大家介绍一种简单的方法），结果为 0xAC05A2，可以把它分解为 red(红色) = AC，green(绿色) = 05，blue(蓝色) = A2。可以看出 red(红色)和 blue(蓝色)的值比较高，而绿色几乎没有，我们就可以猜到这个颜色大概为紫色，这是在十进制数中看不出来的。请注意，在 ActionScript 中，使用哪种进制表示都可以，在一个函数中使用颜色值既可使用十进制又可使用十六进制。对于 Flash 来说，11,273,634 和 0xAC05A2 是一个数，只是对于可怜的人类来说后面一种表示法更易读懂。

那么如何在两种进制之间进行转换呢，将十六进制转换为十进制非常容易。只要输出这

个十六进制数就可以了，trace 函数会自动将它转换为十进制。

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

将十进制转换为十六进制要用到 toString(16)函数，如：

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

输出结果为 ac05a2，如果要使用这个数，不要忘记加上 0x。

透明度和 32 位色

前面提到过，除了 24 位色以外，还有 32 位色，多出 8 位用于表示透明度。就像角度制与弧度制一样(第三章内容)，AS 3 在 24 和 32 位色的使用上有些混杂。AS 3 的绘图 API 很大程度上是基于 Flash MX(Flash 6) 建立的，总之，绘图 API 函数使用一个特殊的参数来指定透明度，所以还要延用 24 位色。另外， BitmapData 类，是从 Flash 8 才加入的，并且使用的是 32 位色彩。如果大家对某个函数使用哪种色彩体系有疑问的话，请查看 ActionScript 参考手册。

我们可以使用十六进制以 0xRRGGBB 这样的格式来表示一个色彩值。同样，32 位的颜色也是如此，以 0xAARRGGBB 这样的格式来表示，其中 AA 表示透明度。因此，0xFFFFFFFF 就表示不透明的白色，0x00FFFFFF 表示完全透明的白色，而 0x80FFFFFF 表示近似 50％透明度的白色。

新的数值类型：int 和 uint

在以前的 ActionScript 版本中，只有一种数值类型 Number，它可以表示正整数，负整数或是浮点数(或 0)。我们已经习惯了这种自由的用法，但是现在多增加的两种数值类型可以让我们的代码更加清晰。

第一个新增加的数值类型是 int(整型)，这个类型可以为正整数或负整数或零。如果我们把一个浮点数值声明为 int 类型的话，小数部分会自动被去掉。比如，把 1.9 声明为 int，结果为 1。因此，当我们确定只使用整数时，就把变量声明为 int ，在循环中用于计数的变量一般应该是 int 。下面这段代码中，i 变量永远不会得到浮点数值，这时使用 int 类型就有了意义。

```
for(var i:int = 0; i < 100; i++) {
    // 在这儿做些事情!
}
```

第二个新的类型是 uint(无符号整型)，“无符号”意思是没有正负(+-)号，永远为正数。32 位颜色值在 AS 3 中总是以 uint 类型存储，这是因为无符号整型比(有符号)整型能够保留更多的数值。 Int 和 uint 都可以存储 32 位数，这个数值大于 40 亿，但是 int 有一个特殊位用于存储符号(+-)，所以只有 31 位数(大于 20 亿)，这样就可以标记正数或负数了。所以，使用 int 类型声明一个正的 32 位色彩值就显得太大了！如果用了又会怎样？让我们来试试：

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

0xFFFFFFFF 的值相当于十进制的 4,294,967,295，因为这个值对于 int 来说太大了，所以结果被“反转”了过来变成了 -1！当然这不是我们所期望的结果。如果使用 uint 类型的话，就没问题了。因此，由于色彩值永远都是正数，并有可能超出 int 的值域范围，所以要使用 uint 来存储它们。

色彩合成

如何将红、绿、蓝三种颜色值组成一个有效的颜色值，这是个普遍的问题。假设有三个变量 red，green，blue，每个变量里面保存一个 0 到 255 之间的数。下面是这个公式：

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

加入透明度后，建立一个 32 位色彩值，公式如下：

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

这里用到了两个位操作符，大家以前可能没有接触过。位操作是对二进制(0 或 1)进行的操作，对于 24 位色来说，如果把颜色值的每一位都列出来，就会得到一串由 24 个 0 或 1 组成的字串。把十六进制 0xRRGGBB 分解成二进制后是这样的：RRRRRRRRGGGGGGGGBBBBBBBB，我们看到有 8 位 red，8 位 green，8 位 blue，也就是说 8 位二进制数等于 256。

在色彩合成公式中，第一个位操作符是 << ，是一个按位左移操作符，该操作是将二进制数值向左侧移动。比如，红色值(red)为 0xFF 或 255，可以由二进制表示为：

11111111

将它向左移动 16 位，结果是：

111111110000000000000000

在 24 位色彩中，它表示红色，转换为二进制后为 0xFF0000，是纯红色。

下面，假设有一个绿色值(green)为 0x55(十进制 85)，二进制表示为：

01010101

将它向左移动 8 位后，结果为：

00000000**01010101**00000000

这样一来，这 8 位数完全移动到了绿色值的范围。

最后，假设一个蓝色值为 0xF3(十进制 243)，二进制表示为：11110011。因为它们都处在蓝色(blue)的范围，所以不需要再去移动它。这样我们总共就拥有了三组数：

111111110000000000000000
00000000**01010101**00000000
0000000000000000**11110011**

可以简单地将它们加起来，成为一个 24 位数，但是，还有一种更好更快的方法：使用或(OR)运算，符号是 | 。它会将两组数的每个二进制位进行比较，如果两个之中有一个数为 1，那么结果就为 1，如果两个数都为 0，那么结果就为 0。可以使用或(OR)运算将 red, green, blue 的值相加起来，也可以这么说“如果这个数或这个数或这个数中有一个数等于 1，那么结果就为 1”。最终结果为：

111111110101010111110011

将这个数转换为十六进制就等于 0xFF55F3 。当然，我们无法看到这些二进制位，也不会与这些 0 或 1 打交道，只需要学会这种写法：

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

十进制写法是：

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

Flash 并不关心人们使用的是十进制数还是十六进制数。

同样，还可以将 red, green, blue 的值全部转换为十六进制的字符串，然后将它们连接成一条很长的字符串，最后再把它们转换为十六进制数。但是，如果这样做的话会很麻烦，

而且使用字符串操作会非常慢。相反，使用二进制操作是 ActionScript 中最快的运算，因为它们属于低级运算。

对于 32 位数，其实道理也是一样的，加入 8 位 alpha(透明度)通道并将其向左移 24 位。例如，有一组 32 位数为 0xFFFF55F3，将 alpha 值向左移动 24 位，结果如下：

1111111111111111010101011111110011

前 8 位数表示透明度，后面的 red, green, blue 值与前面的一样。

获取颜色值

假如有这样一个数 0xFF55F3，要从中提取 red, green, blue 的值。下面请看公式，首先是 24 位色彩：

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

一句句来看。首先，大家也许会猜到 >> 是按位右移运算符，用于将二进制位向右移动。如果这些位向右移动得过多，那么这些数字就会消失，就没有数了。

下面从 red 开始：

1111111101010101111110011

将颜色值向右移动 16 位，结果如下：

11111111，或是 0xFF(255)

对于 green，向右移动 8 位，结果如下：

11111111**01010101**

这里已经得出了 blue 的值，但是 red 值还留在一旁。这里就是要使用与(And)操作符的地方，与(OR)操作符相同，都是对两组数值的比较，可以这样解释“两个数相比较，如果两个都是 1 那么结果就为 1，如果其中有一个为 0，那么结果就为 0”。我们把它与 0xFF 进行比较：

11111111**01010101**

00000000**11111111**

因为所有的 red 位的数字都与 0 相比较，所以它们的结果均为 0，只有当两个数都为 1 时结果才为 1，所以结果如下：

00000000**01010101**

对于 blue 则不需要执行右移操作，只需要让它和 0xFF 执行与(AND)操作即可。对于 32 位色彩，方法也是相同的，只不过需要一点小小的改动：

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

这里，获取 alpha 的值需要向右移动 24 位。现在我们已经学到了很多 Flash 的色彩知识，下面就要开始进行应用了。

绘图 API

先说一下 API 是什么，它是应用程序接口(Application Programming Interface)的缩写。总的来说，API 是指在程序中使用的一些属性和方法来访问某些相关的行为和属性。绘图 API 允许我们使用 ActionScript 绘制直线，曲线，填充色，渐变填充的一些属性和方

法。在这个 API 中有些让人惊讶的方法，我们还要学习很多这方面的知识和灵活的技巧。直至 Flash MX，已经拥有了如下这些绘图方法：

- clear()
- lineStyle(width, color, alpha)
- moveTo(x, y)
- lineTo(x, y)
- curveTo(x1, y1, x2, y2)
- beginFill(color, alpha)
- endFill()

beginGradientFill(fillType, colors, alphas, ratios, matrix)

在 Flash 8 中，又为 lineStle 和 beginGradientFill 增加了几种新的方法，同时也加入了 beginBitmapFill 和 lineGradientStyle 方法。在 AS 3 中，也增加了几种非常有用的方法：

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

先来预览一下这些方法，稍后再对每种方法进行详细的介绍。

绘图对象

在 Flash 早期版本中，绘图 API 方法是影片剪辑(MovieClip)类中的方法，可以在影片剪辑实例中直接调用，代码如下：

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

影片剪辑和 Sprite 都可以访问绘图 API，只是实现起来有些不同。目前，Sprite 影片和影片剪辑都有一个名为 graphics 的属性，用于访问绘图 API 的方法。为了直接访问绘图方法，我们可以这样写：

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

下面在示例中看看这些方法的基本使用。

使用 clear 删除绘制

clear 是所有方法中最简单的,它可以用来删除在影片中所绘制的直线,曲线或填充色。请注意，这个命令中对其它 graphics 绘制的图像不起作用。换句话讲，如果在编辑环境下绘制了一个图形，再对其使用 clear() 命令，结果是无效的。

在绘图中，使用 clear 方法会有些意想不到的效果。在绘图 API 中，如果绘制的影片剪辑越多，运行速度就越慢。对于有很多绘制图形的影片来说，速度不会立刻慢下了，而是随着每个图形所占用的绘制时间会越来越长，从而逐渐地变慢。就算新的图形完全覆盖住了所有旧图形，旧图形的矢量信息也仍然存在并且每次都会被重绘，只有使用 clear 函数才可以完全删除之前旧图形的矢量信息。

使用 lineStyle 设置线条样式

使用 lineStyle(width,color,alpha)方法，作用是为以后使用的绘图线条设置线条样式，该命令对于前面使用的绘图线条不会产生影响。实际上，除了清除或覆盖之外，没有方法可以影响或改变已经绘制的线条或填充。

前面列出的这些参数将来会经常使用，还有一些额外的可选参数如像素提示(pixel)，缩放模式(scale mode)，端点(caps)，拐角类型(joints)和切断尖角(mitres)。如果大家需要更多的设置，也许会用到它们，但是大多数情况下，只会用到下面这些参数。对于它们无需做太多解释，只是来复习一下：

■ width：线条的宽度以像素为单位。只能使用 0 或正整数。虽然可以使用十进制浮点数，但会被取整为最接近的正整数。如果输入的是 0 或负数， Flash 将绘制 1 像素宽的线。这与在 Flash IDE 中在属性面板中选择“细线”的功能相同。

■ color：线条的颜色。使用十进制或十六进制的 24 位色彩值表示。

■ alpha：线条的透明度。使用 0.0 到 1.0 数字之间的数表示透明度的比例。值为 1.0 表示完全不透明，值为 0.0 表示完全透明或不可见。注意，这与 AS 2 中使用 0 到 100 表示法是不同的。

由于这些参数是可选的，可以只使用 lineStyle(1) 来设置一条 1 像素宽的黑色线条。其实第一个参数也是可选的，如果不填 widh 参数，只使用 lineStyle() 的话，那么线条就被清除，只获得了一条不可见的线，相当于使用绘图指令时没有设置线条样式(lineStyle)。另一个容易出错的地方是，在使用 clear 方法时，不仅清除了当前绘制的图形而且也清除了当前使用的线条样式。如果在影片绘图时设置了一个普通的线条样式，而后又将线条清除，那么在绘制其它图形之前还需要重新设置线条样式。否则的话，接下来绘制的线条就是不可见的，调用 clear 方法同时还会将绘图指针位置归为 0,0。

使用 lineTo 和 moveTo 绘制直线

在一种绘图语言中会有多种方法用来绘制直线。一种是使用画线指令，需要有一个起点和一个终点，并在这两点之间画一条直线。另一种是使用 lineTo 指令，只需要一个终点。那么 ActionScript 是怎样工作的呢，如果向某一点画线，哪里才是起点呢？如果之前没有进行过画线，那么起点就是 0,0 点，可以这样写：

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

结果将会看到一条从左上角(0,0)画到 100,100 像素位置的线(假设已经设置了线条样式)。在绘制完最少一条线后，这条线的终点位置就会成为下一条线的起点位置。不过，我们还可以使用 moveTo 方法为下一条线指定一个新的起点位置。

可以把绘图 API 想像成一个拿着笔和纸的机器人，开始的时候，笔处在 0,0 点。当我们告诉它向某点画一条线时，它就将笔在纸上划过，并向这个位置移动。 moveTo 方法就像在说“OK,现在抬起笔，放到下一个点上。”虽然仅使用 moveTo 指令不会产生一个新的图形，但是它会影响下一次绘图时的位置。通常使用 moveTo 作为第一条绘图指令，用于将绘图 API 的“笔”移动到起点位置。现在大家已经拥有了足够的知识可以来实践一下了，让我们创建一个简单的绘图应用程序，这个程序是完全依赖绘图 API 完成的。这里是文档类：

```
package {
 import flash.display.Sprite;
 import flash.events.MouseEvent;
 public class DrawingApp extends Sprite {
  public function DrawingApp() {
   init();
  }
  private function init():void {
   graphics.lineStyle(1);
   stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
   stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
  }
```

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

首先，导入 MouseEvent 类，因为这里的每件事都要用到鼠标事件。在 init 方法中，线条样式设置为 1 像素黑色线，并增加 mouseDown 和 mouseUp 作为事件侦听器。

然后是 onMouseDown 方法，每当用户按下鼠标是都会调用它，这意味着用户要开始在当前鼠标位置画线了。这个函数通过使用 moveTo 方法将那支虚拟的笔放置在当前鼠标的位置，以鼠标坐标为参数，随后为 mouseMove 添加了一个侦听器。

每当用户移动鼠标时，都会调用 onMouseMove 方法，向当前鼠标的位置画一条线。

最后是 onMouseUp 方法，用于删除 mouseMove 侦听器使其不再进行画线。

好的，现在已经制作好了一个短小精悍的绘图程序。我们不需要再费太大的力气就可以为这个程序加入一些简单的控制，让它具有完整绘图程序的功能。只需要为线条颜色(color)和线条宽度(width)创建一些变量，再创建一些按钮什么的用来改变它们，并重新调用 lineStyle 方法使用这些新的值。对了，还可以再放一个按钮用于调用 clear 方法。把这个留做是一个练习，希望大家有兴趣的话，能够自行完成。

使用 curveTo 绘制曲线

下一个绘图函数，curveTo(x1, y1, x2, y2)，起点和 lineTo 一样，同样是以上一次画线的终点做为本次画线的起点，也可以使用 moveTo 命令指定画笔的起点，如果是第一次画线默认的起点为 0,0。

可以看到， curveTo 函数中包括两个点。第一个是控制点影响曲线的形状，另一个是曲线的终点。这里使用的是名为二次方贝塞尔曲线的标准公式，该公式可以计算出两点间的曲线，这条曲线向着控制点弯曲。请注意，这条曲线不会与控制点接触，很像是曲线被它吸引过去的。

下面来看动作脚本，文档类 DrawingCurves.as：

```
package {
 import flash.display.Sprite;
 import flash.events.MouseEvent;
 public class DrawingCurves extends Sprite {
  private var x0:Number = 100;
  private var y0:Number = 200;
  private var x1:Number;
  private var y1:Number;
  private var x2:Number = 300;
  private var y2:Number = 200;
  public function DrawingCurves() {
   init();
  }
  private function init():void {
```

```
    stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
   }
   private function onMouseMove(event:MouseEvent):void {
    x1 = mouseX;
    y1 = mouseY;
    graphics.clear();
    graphics.lineStyle(1);
    graphics.moveTo(x0, y0);
    graphics.curveTo(x1, y1, x2, y2);
   }
  }
}
```

测试这个文件，把鼠标来回移动。这里使用了两个给定的点作为起点和终点，使用鼠标位置作为控制点。请注意，曲线不会真正到达控制点位置，而只到达与控制点一半的位置。

过控制点的曲线

现在，如果想让曲线真正地穿过控制点，那么这就是我们工具箱中的另一个工具。使用下面这个公式计算出控制点的实际位置,这样就可以让曲线穿过指定的点了。同样,以 x0,y0 为起点，以 x2,y2 为终点，x1,y1 为控制点，把将要穿过的点叫 xt,yt（目标点）。换言之，如果让曲线穿过 xt,yt 点，那么 x1,y1 又需要如何使用呢？公式如下：

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

只需要把目标点乘以 2，然后减去起点与终点的平均值。大家可以画张图来究竟一下它的原理，要么就直接学会使用它。

把公式放在代码中，鼠标坐标用使用 xt,yt，我们只需要改变前一个文档类中的两行，将下面两行：

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

替换为

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

或者直接看 CurveThroughPoint.as，现成的文件。

创建多条曲线

下面我们将目光转向创建多条曲线，而不仅是一条曲线，创建一条平滑的向各个方向弯曲的线。首先，来看一个错误的做法，是我原先尝试过的一种方法。从随便一个点位出发，经过第一个点到第二个点再到第三个点，经过第四个到达第五个，经过第六个到达第七个等等绘制一条曲线。这里是代码(文档类 MultiCurve1.as):

```
package {
 import flash.display.Sprite;
 public class MultiCurves1 extends Sprite {
  private var numPoints:uint = 9;
  public function MultiCurves1() {
   init();
  }
```

```
  private function init():void {
   // first set up an array of random points
   var points:Array = new Array();
   for (var i:int = 0; i < numPoints; i++) {
    points[i] = new Object();
    points[i].x = Math.random() * stage.stageHeight;
    points[i].y = Math.random() * stage.stageHeight;
   }
   graphics.lineStyle(1);
   // now move to the first point
   graphics.moveTo(points[0].x, points[0].y);
   // and loop through each next successive pair
   for (i = 1; i < numPoints; i += 2) {
    graphics.curveTo(points[i].x, points[i].y,

    points[i + 1].x, points[i + 1].y);
   }
  }
 }
}
```

第一次循环在 init 方法中，建立一个数组存储九个点。每个点都是一个 object 拥有 x,y 属性，它们的值都是舞台尺寸的随机数。当然，在一个真正的程序中，点位也许不是随机的，只是用这种方法进行快速设置。

随后设置线条样式，将笔移动到第一个点位。下一个循环从 1 开始每次递增 2，所以线条是经过第一点到达第二点，然后从第三点到第四点，再从第五点到第六点，最后从第七点到第八点。至此，循环停止，因为第八点是最后一个点。大家也许注意到了，这里至少要有三个点，而且点的数量必需为奇数个。

程序看起来还不错，测试一下试试。如图 4-1 所示，看起来不是非常平滑，有棱有角的，这是因为曲线之间没有进行协调，它们之间共用了一个点。

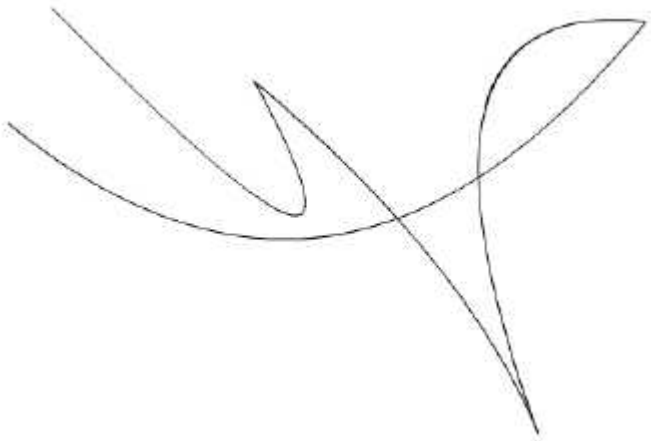


图 4-1 多条曲线，错误的方法。我们可以清楚地看到曲线的结束和开始的位置。

我们也许不得不去加入更多的点才能使解决这个问题。这里有个策略：在每两对点之间，加入一个新点(中间点)放在这两点的正中间。然后使用这些中间点作为起点和终点，而把最初的那些点(原始点)作为控制点。

图 4-2 说明了解决办法。在图中，白点为原始点，黑点为中间点。这里使用了三条 curveTo 方法，图中的点使用了不同的颜色，这样就能分辨出起点与终点了。(图 4-2 是 multicurvedemo.fla 文件的一张截图，可以在 www.friendsofted.com 的 books 页面下载)

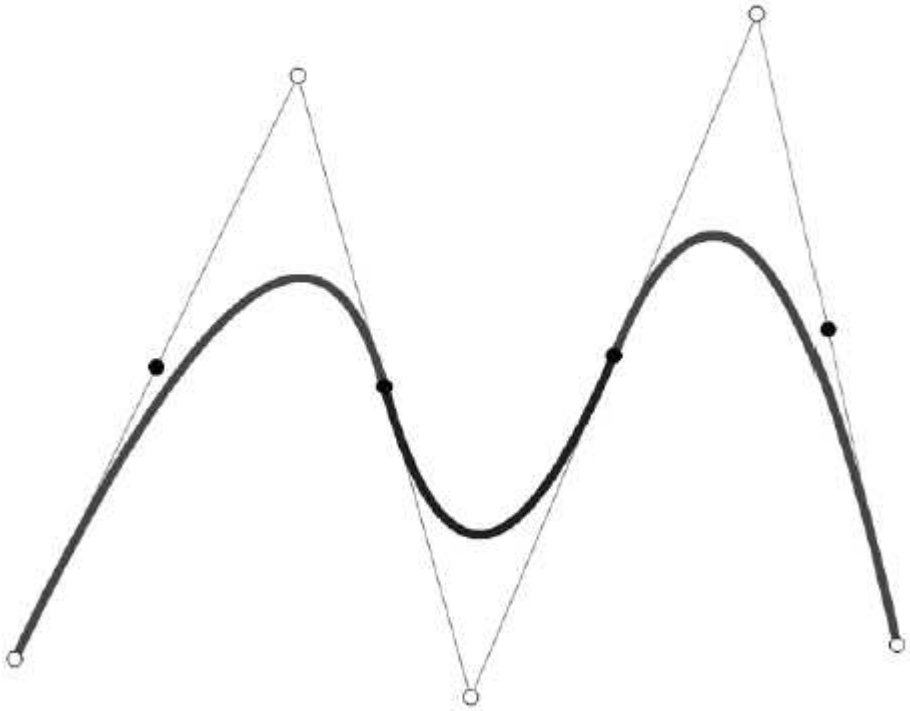


图 4-2 带有中间点的多线条

注意，图 4-2 中第一个中间点和最后一个中间点都没有被使用，第一个和最后一个原始点留作曲线的两个端点，只需在第二个点和倒数第二个点之间进行连接。这里是前一个例子的升级版，文档类 MultiCurve2.as：

```
package {
 import flash.display.Sprite;
 public class MultiCurves2 extends Sprite {
  private var numPoints:uint = 9;
  public function MultiCurves2() {
   init();
  }
  private function init():void {
   // first set up an array of random points
   var points:Array = new Array();
   for (var i:int = 0; i < numPoints; i++) {
    points[i] = new Object();
    points[i].x = Math.random() * stage.stageHeight;
    points[i].y = Math.random() * stage.stageHeight;
   }
   graphics.lineStyle(1);
   // now move to the first point
   graphics.moveTo(points[0].x, points[0].y);
   // curve through the rest, stopping at each midpoint
   for (i = 1; i < numPoints - 2; i ++) {
    var xc:Number = (points[i].x + points[i + 1].x) / 2;
    var yc:Number = (points[i].y + points[i + 1].y) / 2;
    graphics.curveTo(points[i].x, points[i].y, xc, yc);
   }
   // curve through the last two points
   graphics.curveTo(points[i].x, points[i].y, points[i+1].x,
   points[i+1].y);
```

```
  }
 }
}
```

请注意，在新代码中， for 循环从 1 开始到 points.length -2 结束，也就避开了第一个点和最后一个点。程序要做的是，创建新的 x,y 点，这个点是数组中后面两个点位的平均值。然后从数组下一个点位开始画一条曲线到新的平均点(中间点)。当循环结束时， i 变量指向倒数第二个元素，因此，可以穿过这里向最后一个点画条曲线。

这时，就得到一个非常平滑的图形，见图 4-3。注意，这时原始点的数量不再受奇数个的限制。

再加一点小小的变化，使用同样的技术创建一条封闭的曲线。首先，计算一个初始的中间点，并移动到这里。然后，进行循环，获得每一个中间点，最后，将最后一条曲线画回初始中间点。图 4-4 为显示结果

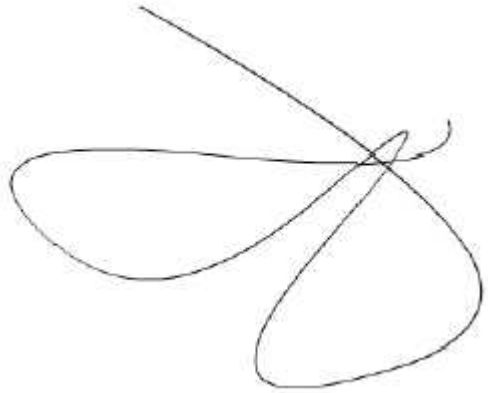


图 4-3 多条平滑曲线

```
package {
 import flash.display.Sprite;
 public class MultiCurves3 extends Sprite {
  private var numPoints:uint = 9;
  public function MultiCurves3() {
   init();
  }
  private function init():void {
   var points:Array = new Array();
   for (var i:int = 0; i < numPoints; i++) {
    points[i] = new Object();
    points[i].x = Math.random() * stage.stageHeight;
    points[i].y = Math.random() * stage.stageHeight;
   }
   // find the first midpoint and move to it
   var xc1:Number = (points[0].x + points[numPoints - 1].x) / 2;
   var yc1:Number = (points[0].y + points[numPoints - 1].y) / 2;
   graphics.lineStyle(1);
   graphics.moveTo(xc1, yc1);
   // curve through the rest, stopping at midpoints
   for (i = 0; i < numPoints - 1; i ++) {
    var xc:Number = (points[i].x + points[i + 1].x) / 2;
    var yc:Number = (points[i].y + points[i + 1].y) / 2;
    graphics.curveTo(points[i].x, points[i].y, xc, yc);
   }
```

```
            // curve through the last point, back to the first midpoint
            graphics.curveTo(points[i].x, points[i].y, xc1, yc1);
        }
    }
}
```

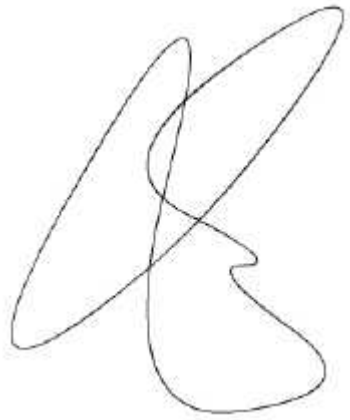


图 4-4 多条封闭曲线

使用 beginFill 和 endFill 创建图形

beginFill(color, alpha) 方法非常简单，没有太多可说的。有一点值得注意，同 lineStyle 一样， alpha 的取值范围也变为了 0.0 到 1.0，而不是 0 到 100，这项也是可选的，默认为 1.0。无论何时执行该帧的绘图代码 Flash 都会开始进行计算，无论何时遇到 endFill 指令 Flash 都会停止计算。总结一下，过程如下：

- moveTo
- lineStyle（如果有参数可以填入）
- beginFill
- 在一系列的 lineTo 和 curveTo 方法后，要在最初的点位结束
- endFill

事实上，使用前三个方法的顺序不会影响到绘图。我们不是必需要指定线条样式，请记住如果不指定线条样式就会得到一条看不见的线条，非常适合绘制填充色，当然两者同时绘制也不错。如果所绘制的线条没有回到最初开始的点位，一但调用了 endFill， Flash 将会自动绘制一条封闭线，是为了能封闭这个图形。调用 endFill 后，无论线条样式如何，都会自动将最后一条线绘制完成。当然，我们自己将线条封闭是个很好的习惯，这样一来，既确保了最后的能够正确绘制，又可以让看代码的人知道我们究竟想画的是什么图形。

下面来试一下绘制填充色，可以使用前面的封闭曲线示例(MultiCurve3.as)来完成，这里已生成了一个封闭的图形。只要将 beginFill 语句放在第一条 curveTo 前面的任何地方——如 beginFill(0xff00ff);，这样就创建了亮紫色的填充——最后使用 endFill() 结束。

使用 beingGradientFill 创建渐变填充

下面开始学习绘图 API 的强大函数：beginGradientFill(fillType, colors, alpha, ratios, matrix)。与线条样式相同，这个方法也有很多额外的可选参数——扩散方法(spreadMethod)，插补方法(interpolationMethod)及焦点位置比例(focalPointRatio)。这些参数可用来调整渐变的属性，但是在大多数简单的应用中这些参数不是必需的，而这个方法与 beginFill 有很多相似之处，同样也使用 endFill 作为结束。两者最大的不同是填充

的颜色，我虽不想说 beginGradientFill 是用来创建渐变填充的，但是，如果不这么说，又感觉少了点什么。渐变填充至少要有两种颜色，图形的第一部分从第一种颜色开始，然后逐渐混合成为另一种颜色，或者混合成一个或多个已定义的颜色。

指定填充类型

我们可以建立两种类型的渐变填充：线性(linear)和放射(radial)状。在线性填充时，渐变的颜色沿着直线从一点到另一点。默认的情况，是从左向右的一条直线，也可以是从上到下或其它角度的直线。图 4-5 是一些线性渐变的例子。

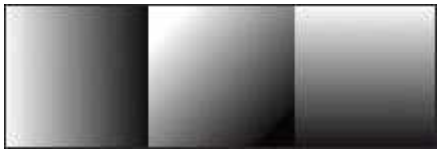


图 4-5 线性填充

为了能够看到线性渐变，需要至少两种不同的颜色。如果指定了两种颜色，那么填充将会从第一种颜色向第二种颜色渐变。如果指定了更多的颜色，填充色将会从第一种颜色渐变为第二种，然后再渐变到第三种……直到最后一种。

放射状填充与线性填充使用的参数大致相同，只是在解释上有所不同。从指定的中心位置开始创建渐变，以这点为基础向外进行放射，形成一个圆或椭圆。我们指定的第一种颜色用做内圆，最后一种颜色作为外圆，唯一不需要的就是角度。图 4-6 是一个放射状填充。

对于 beginGradientFill(fillType, colors, alphas, ratios, matrix) 方法，第一个参数为填充类型，非常简单，用一个字符串表示，这两个值中的一个：”linear”或”radial”。与第三章讲的事件类型很像，目前事件类型也被设置为 flash.display.GradientType 类的静态属性，为的是避免错误输入。我们可以导入 GradientType 类然后输入 GradientType.LINEAR 或 GradientType.RADIAL。

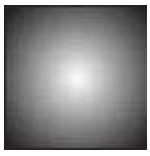


图 4-6 放射状填充

设置 colors ， alphas 及 ratios

使用 colors 参数设置颜色，当然也必需设置每一个颜色所要填充的位置，使用 0 到 255 中的一个数进行指定，0 表示填充的开始位置，255 表示填充结束位置。在这些数值中，每一个数代表一个颜色的位置，这就是填充的比例。因此，如果有要填充两种颜色，那么应该指定 0 和 255 作为 ratios，如果有三个颜色值，那么应该写成 0,128,255。这样就将第二个颜色放到了另外两个颜色的中间。如果比例值为 0,20,255，那么第一种颜色会很快渐变为第二种颜色，然后非常缓慢地渐变为第三种颜色。请记住这些数值不是像素值，而是指在 255 中的某一个部分。

我们同样必须指定渐变色的透明度，这就是 alpha 值，从 0.0 到 1.0，而在 AS 2 中表示为 0 到 100。如果不需要透明度，那么就设置为 1.0。如果设置的透明度是从 1.0 到 0.0，那么渐变的过程不仅是改变颜色，而且还有平滑淡出的效果。可以用做创建柔和的阴影（也许比使用投影滤镜(drop shadow filter)还要好些）。

这里每一个参数都是一个数组，因为需要传入至少两个以上的 colors，alphas 及 ratios。我们可以先创建一个数组，然后写入每一个参数的值，如下：

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

下面是一个更简单作法，我们可以在创建数组的同时，为数组的每个元素赋值，直接写在方括号中，以逗号作为分隔：

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

事实上，我们甚至可以不去使用 colors 变量。直接把右边的数组写入 beginGradientFill 表达式中作为参数。因此，可以写成这样：

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

这里定义了两个颜色值，两个 alpha 值(均为 100%)和比例值的开始位置与结束位置，所以渐变将以白色为开始最后逐渐变化为黑色。当然，也可以为每个参数设置一个对应的变量，如果定义了很多个颜色值的话，这样写会更清楚些。如果定义了三个颜色值，就必需有三个 alpha 值和三个比例值与之对应。如果有某些值多了或少了，那么就会引起会静默失败——没有渐变，没有填充，没有错误信息。

下面只需要设置一下填充的起点、终点或角度了。也许大家已经猜到了，这就是神秘的 matrix(矩阵) 参数。

创建矩阵

矩阵(Matrix)就是一个二维表格，每行每列中包括不同的数，可以出计算不同的值。矩阵此外还用于绘图(graphics)中，用作旋转，缩放和移动物体。在这里，用于对渐变的控制，我们需要为填充进行定位，设置它的大小，或是进行旋转。使用 matrix 时，需要创建一个 matix 对象，这是一个 flash.geom.Matrix 类的对象。（实际上， Matrix 类不仅用于操作渐变填充，但这里我们只介绍它在渐变中的应用。第 18 章中介绍了更多的矩阵使用）。

在使用 Matrix 类时有一点复杂，这里有一个特殊的方法用于创建渐变填充的矩阵类型，这个方法名为 createGradientBox，必要参数为 width 和 height，可选参数为 rotation 及渐变的起点的 x,y 位置。首先创建一个 Matrix 类的实例，然后调用它的 createGradientBox 方法自动设置内部参数值。形式如下：

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

不要忘记在类的开始处导入 flash.geom.Matrix 类。如果仅指定 width 和 height，最后三个值默认为 0。来看一下代码，这里是文档类 GradientFill.as：

```
package {
 import flash.display.GradientType;
 import flash.display.Sprite;
 import flash.events.MouseEvent;
 import flash.geom.Matrix;
 public class GradientFill extends Sprite {
  public function GradientFill() {
   init();
  }
  private function init():void {
   graphics.lineStyle(1);
   var colors:Array = [0xffffff, 0xff0000];
```

```
    var alphas:Array = [1, 1];
    var ratios:Array = [0, 255];
    var matrix:Matrix = new Matrix();
    matrix.createGradientBox(100, 100, 0, 0, 0);
    graphics.beginGradientFill(GradientType.LINEAR, colors,
    alphas, ratios, matrix);
    graphics.drawRect(0, 0, 100, 100);
    graphics.endFill();
   }
  }
}
```

测试影片后，会看到一个从白色到红色渐变的正方形。现在，我们改变一下绘图代码，在不同的地方绘制这个正方形(见 GradientFill2.as)：

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

现在这个正方形为全红色，为什么会这样？渐变在 x 为 0 处开始，但它只有 100 像素宽，而正方形是从 100 像素开始的，渐变色已经到达了全红色，红色从这里开始已经出界了。因此，如果想让矩阵的 x,y 在正方形的左上角开始，应该这样写：

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

这样， x,y 坐标就与正方形的起点相同。请使用同样的正方形，通过改变矩阵和渐变填充观察创建的填充是什么效果。首先，试一下三种颜色：

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

不要忘记改变 alphas 和 ratios ，将中间的比例移动一下看看对渐变色的影响。试用 64 或 220 代替 120。

下面是一个直接 alpha 变化的例子，使用相同的颜色，只改变 alpha 的值：

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

试着改变一下角度，下面是 45 度角：

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

使用 Math.PI/2 旋转 90 度形成一个垂直填充。- Math.PI/2 形成向上填充，而 Math.PI 左是从右向左填充，默认为从左向右填充。

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

现在，将这些应用在线性填充(linear)的技巧改为放射状填充(radial)的版本。

颜色转换

下面一个渲染工厂是 flash.geom.ColorTransform 类及其方法。与绘图 API 不同，该类不允许创建图形，仅仅用于改变已存在于影片或显示对象实例中图形的颜色。让我们去看看它是怎样工作的。

使用 ColorTransform 类改变颜色

在 ActionScript 2 中，操作影片剪辑颜色最常用的方法是 Color 类，拥有像 setRGB 和 setTransform 这样的方法。Flash 8 中就引用了 ColorTransform 类，同时 Color 类就不再被推荐使用，但是想大多数人仍然继续使用 Color 类，因为这已经成为习惯了。Color

类已经不是 AS 3 的一部分了，现在是学习新方法的最佳时机。ColorTransform 的方法与 Color 类的方法本质上是非常像，两者差不太多，只是语法上略有不同。

首先，要知道 Sprite 影片，影片剪辑，或其它任何显示对象都有一个属性叫作 transform。这是 flash.geom.Transform 类的一个实例，其中包括一些不同的属性用于缩放，旋转，定位和改变影片颜色，影响颜色属性的就是 colorTransform。要知道这是一个显示对象属性的属性，访问方法如下；

```
mySprite.transform.colorTransform
```

或使用一个继承自 Sprite 的类，可以直接获得类自身的 transform 属性：

```
transform.colorTransform
```

通过为一个 ColorTransform 对象的颜色属性赋值可以改变一个对象的色彩。你也许会问，什么是 ColorTransform 对象？如果大家使用过 AS 2 的 Color 方法：setTransform，就知道我们需要传入一个 object(对象) ，在这个 object 中有各种属性，需要告诉它如何变换颜色。 ColorTransform 与这个方法非常相似，但不是使用一个普通的对象(object)，而现在这个方法已经拥有了官方的类。在 AS 2 中，是这样做的：

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

在 AS 3 中，应该创建一个同样的 ColorTransform 对象，像这样：

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

前四个值为乘数，后四个值为偏移量，马上会介绍这个公式，然后就会知道为什么它们这样命名了。大家也许注意到了乘数的比例值为十进制范围 -1.0 到 1.0，而不是 -100 到 100。事实上，帮助文档说是从 0.0 到 1.0，但还可以使用负数做为乘数实现一些有趣的效果(稍后会看到)，偏移量依然是从-255 到 255。ColorTransform 对象的构造函数如下：

```
ColorTransform(redMultiplier,
greenMultiplier,
blueMultiplier,
alphaMultiplier,
redOffset,
greenOffset,
blueOffset,
alphaOffset)
```

转换一个特殊颜色通道的公式如下，以红色通道为例：

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

在使用 ColorTransform 时，记得它是 flash.geom 包中的一部分，所以需要导入类。

给大家一个例子，下一个文档类：TransformColor.as，在 SWF 中嵌入一张图片作为位图(位图类的实例)。因为位图类是一个显示对象，拥有 transform 属性，代码设置了位图的 transform 的 colorTransform 属性，使用构思好的设置来制作一张底片效果的图像：

```
package {
 import flash.display.Bitmap;
 import flash.display.Sprite;
 import flash.geom.ColorTransform;
 public class TransformColor extends Sprite {
  [Embed(source="picture.jpg")];
```

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

   255,255,255,0);
  }
 }
}
```

测试这个影片时，请改变这行代码

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

请匹配这个路径为所使用图片的路径，如果在 Flash IDE 中编辑，只需要在库中导入这张图片，为 ActionScript 导出，并输入类名为 Picture。重要的一句是 ColorTransform 的设置。

滤镜(Filter)

滤镜是一些位图的效果，可以应用于任何显示对象。在 Flash IDE 中可以通过滤镜面板或使用时间轴的 ActionScript 来使用滤镜，由于这本书是关于 ActionScript 的，所以只能简单地讨论一下应用滤镜的方法。在 AS 3 中包括以下几种滤镜：

- Drop shadow(投影滤镜)
- Blur(模糊滤镜)
- Glow(发光滤镜)
- Bevel(斜角滤镜)
- Gradient bevel(渐变斜角滤镜)
- Gradient glow(渐变发光滤镜)
- Color matrix(颜色矩阵滤镜)
- Convolution(卷积滤镜)

Displacement map(置换图滤镜)

虽然不能一一介绍每种滤镜的使用细节，但大家可以通过帮助文档来学习。在书中还会有很多滤镜使用的例子，所以在这里只介绍一下滤镜使用的总体方法和两个具体实例。

创建滤镜

通过使用 new 关键字及滤镜名来创建滤镜，并给出所需的参数。例如，创建一个 blur filter(模糊滤镜)，最简单的一种滤镜，写法入下：

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

参数分别为 blurX,blurY,quality。这个例子会将对象在 x 和 y 轴上模糊 5 个像素，模糊的品质为中等。

另一点需要知道的是滤镜在其名为 flash.filters 的包中。所以要在文件的开始处将它们导入进来：

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

如果希望导入包中所有的滤镜，可以使用简写：

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

现在，我们可以直接创建任何类型的滤镜了，但是一般来说，除非要使用这个包中的大部分滤镜，否则最好避免使用通配符(*)，而是明确地导入所需要的类。这样做只是为了能够清楚，哪些是真正想要导入的而哪些不是。好了，现在已经创建了一个模糊滤镜，但怎么才能使它去模糊一个对象呢？

任何一个显示对象都有一个名为 filters 的属性，这是一个包括了所有滤镜的数组，因为如果一个对象要应用多个滤镜，那么只需要再将模糊滤镜放到数组中即可。乐观地看，应用滤镜应该可以像使用基本数组操作那样简单，push，就像这样
mySprite.filters.push(blur);，但是很遗憾，没有这么简单。在整个数组赋值为 filters 之前，Flash 不关心 filters 数组的变化。

如果已知对象没有应用任何的滤镜，或想要重写它们，只需要新建一个数组，将我们的滤镜粘在上面，再将这个新数组赋给 filters 属性就可以了。先来试一下，下面一个文档类 Filters.as，创建了一个 sprite 影片并且在里面绘制了一个黄色的正方形，然后，创建一个滤镜，加入数组中，最后将数组赋给 sprite 的 filters 属性：

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

瞧！出现了一个模糊的黄色方块儿。重要的部分用黑体着重，我们可以简写一点：

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

或再短一点：

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

在创建数组的同时，将滤镜放进去，并应用 filters 属性，这样一来，Flash 会很高兴。

但是如果已经有了滤镜并希望继续使用，这时，但又不确定是否有滤镜存在，那该怎么办呢？在 Flash 8 中，这是件很麻烦的事，因为一个显示对象的 filters 属性如果没有应用滤镜，那么它将是未定义(undefined)的。但在 AS 3 中， filters 数组总是保持为一个空数组，只需要给数组赋值，将滤镜 push 进去，并将其赋给对象的 filters 属性即可，方法如下：

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

```
sprite.filters = filters;
```

如果使用这种方法，那么无论是否有滤镜存在都没有问题，滤镜只是被加入到数组列表中而已。因为 filters 属性是一套成熟的数组，所以可以使用不同的数组操作方法。比如，使用 concat 方法：

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

我不认为这是个“正确”的做法，大家只要知道将一个包涵有滤镜的数组赋给 filters 属性就足够了。

动态滤镜

现在我们已经基本上知道了如何在 ActionScript 中使用滤镜了。接下来，用已经学过的知识，制作一个动态滤镜。这个效果，使用文档类 AnimatedFilters.as：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.filters.DropShadowFilter;
 public class AnimatedFilters extends Sprite {
  private var filter:DropShadowFilter;
  private var sprite:Sprite;
  public function AnimatedFilters() {
   init();
  }
  private function init():void {
   sprite = new Sprite();
   sprite.graphics.lineStyle(2);
   sprite.graphics.beginFill(0xffff00);
   sprite.graphics.drawRect(-50, -50, 100, 100);
   sprite.graphics.endFill();
   sprite.x = 200;
   sprite.y = 200;
   addChild(sprite);
   filter = new DropShadowFilter(0, 0, 0, 1, 20, 20, .3);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var dx:Number = mouseX - sprite.x;
   var dy:Number = mouseY - sprite.y;
   filter.distance = -Math.sqrt(dx * dx + dy * dy) / 10;
   filter.angle = Math.atan2(dy, dx) * 180 / Math.PI;
   sprite.filters = [filter];
  }
 }
}
```

首先在 sprite 中画一个正方形，正方形在 sprite 的居中位置，然后将 sprite 移动到舞台中间，用一些默认属性创建投影滤镜(DropShadowFilter)。

添加一个 enterFrame 事件的侦听器及处理函数:onEnterFrame 方法,用于计算角度(angle)及使用三角函数计算 sprite 影片与鼠标的距离(distance)。使用 angle 和 distance 设置投影滤镜的 angle 和 distance 属性，最后将这个滤镜再应用到 sprite 上。请注意，

我们不需要每次都创建一个新的滤镜，可以继续使用同一个滤镜，只需要改变它的属性即可。然而，只是改变这些属性也不能更新 sprite 影片的显示。因此，还需要再将变化过的滤镜效果赋值给 filters 属性。

位图(Bitmaps)

与滤镜相同，可以用整本书来介绍 Bitmap 和 BitmapData 类，看起来也不错，但是这并不是本书的目的。我们将通过一些简单的例子，用来指出 AS 2 与 AS 3 中位图处理的变化。

在 AS 2 中，通过调用 BitmapData()函数，新建一个 BitmapData 对象使用如下参数：

```
new BitmapData (width:Number,
height:Number,
transparent:Boolean,
fillColor:Number)
```

你也许猜到了， BitmapData 类同样也是嵌入在一个包中，完整的使用名称如下 flash.display.BitmapData。所以需要导入包，对于 width 和 height 参数则非常显而易见， transparent 参数表示创建的图像是否包涵一个 alpha 通道，选择 true 或 false ，fillColor 是创建图像的初始颜色，如果 transparent 为 true 的话，那么位图就用 32 位色表示，0xAARRGGBB，如果为 false 的话，就可以使用 24 位安全色表示。

在创建 BitmapData 对象时，也许很想能看到它的样子。在 AS 2 中，使用 attachBitmap 命令在影片剪辑中添加一个位图。大家也许会想，现在是否可以使用 addChild 在显示对象中添加一个位图，但事实上并没有这么简单。问题在于 addChild 只对继承自 DisplayObject 类的对象起作用，如 Sprite 影片，影片剪辑和文本框。然而，如果我们研究一下类的结构，就会发现 BitmapData ，没有继承自 DisplayObject，所有不能直接添加对象。这就是为什么要有 Bitmap 类的原因， Bitmap 类几乎始终都有一个函数作为 BitmapData 实例的容器，可以这样创建：

```
var myBitmapData:BitmapData = new BitmapData(100, 100, false, 0xff0000);
var myBitmap:Bitmap = new Bitmap(myBitmapData);
```

现在就可以将对象加入到显示列表了：

```
addChild(myBitmap);
```

使其可见后，Bitmap 实例也可以改变位置，进行缩放，增加滤镜等等。

测试这个例子，只需要在第二章给出的类框架的 init 方法加入这三行就可以了，不要忘记导入 flash.display.Bitmap 和 flash.display.BitmapData，运行后就会看到一个红色的正方形。乍看上去，与使用绘图 API 所画的图形没什么不同，但是要知道这并不是矢量图绘制法：填充一个红色的正方形。这是张位图图像，在位图中每一个像素都要分别指定而且是可变的。事实上，每一个像素值都可以使用 getPixel,getPixel32 和 setPixel,setPixel32 进行读取和设置。两个版本的不同之处在于 getPixel 和 setPixel 使用 24 位色彩值忽略了 alpha 通道，而 “32”版的则使用 32 位色彩值其中包括了透明度信息。让我们来做个例子，制作一个简单的喷漆工具，就像所有位图喷漆程序一样。

这里是文档类，SprayPaint.as：

```
package {
 import flash.display.Sprite;
 import flash.display.Bitmap;
 import flash.display.BitmapData;
 import flash.events.MouseEvent;
 import flash.events.Event;
 import flash.filters.BlurFilter;
 public class SprayPaint extends Sprite {
  private var canvas:BitmapData;
```

```
  private var color:uint;
  private var size:Number = 50;
  private var density:Number = 50;
  public function SprayPaint() {
   init();
  }
  private function init():void {
   canvas = new BitmapData(stage.stageWidth,
   stage.stageHeight,
   true, 0x00000000);
   var bmp:Bitmap = new Bitmap(canvas);
   addChild(bmp);
   stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
   stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
  }
  private function onMouseDown(event:MouseEvent):void {
   color = Math.random() * 0xffffff + 0xff000000;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onMouseUp(event:MouseEvent):void {
   removeEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   for (var i:int = 0; i < density; i++) {
    var angle:Number = Math.random() * Math.PI * 2;
    var radius:Number = Math.random() * size;
    var xpos:Number = mouseX + Math.cos(angle) * radius;
    var ypos:Number = mouseY + Math.sin(angle) * radius;
    canvas.setPixel32(xpos, ypos, color);
   }
  }
 }
}
```

这也许是目前为止最复杂的代码，但除了 BitmapData 的内容外，其它的知识前面都讲过，只不过又使用了一遍而已。一步步来看，首先，创建了一些类变量，包括 canvas 变量，用于存放 BitmapData 的实例。创建的实例尺寸等于舞台的尺寸，并使用透明的背景色。然后使用 canvas 创建一个位图，并加入到显示列表。

鼠标事件处理函数中选择了一个随机的颜色，并且带有添加和删除 enterFrame 事件处理函数的功能。我们来回忆一下三角学，首先，从 0 到 Math.PI＊2 中计算出一个随机的角度，不要忘记使用弧度制表示，相当于随机的 360 度。然后，计算出一个随机的半径后，再使用三角函数将半径和角度转换为 x,y 值。最后使用 setPixel32 以鼠标位置加上随机的 x,y 值的像素点设置为喷漆色，每一次开始喷漆时随机决定颜色。在这个例子中有一个 for 循环，每一帧都会进行循环，每次循环多少次由 density 的值决定。 color 的值为 24 位的色彩值，然后加上 0xFF000000，为的是设置 alpha 通道为完全不透明，如果没有加上这个值，那么所有的颜色就都为透明的。如果用 0xFFFFFFFF 乘以 Math.random()，那么颜色的透明度是随机的，也许是你想要的，但不是我想要的。通过改变 density 和 size 的值再测试一下，看看会有些什么不同的效果。大家也许已经想到如何让用户来控制改变这些参数了。

刚刚看到这个程序时，你也许会想，“真是小题大作，完全可以用绘图 API 或通过加

载小影片剪辑并改变颜色来实现”。是的，完成可以这么做，但是如果使用绘图 API 绘出成千上万的独立图像后，会发习画得越多，速度越慢。画过几百个图形后，慢下来的速度会变得非常明显，这个程序也就费掉了，使用加载影片剪辑的方式也是如此。但是，使用位图就完全不同了，我们可以使用这个程序喷上一天，都不影响程序的速度或效率。
如果想看到更酷的效果，就把下面一行代码加在位图对象 bmp 的后面：

```
bmp.filters = [new BlurFilter(2, 2, 3)];
```

在位图中使用模糊滤镜比在矢量图中使用效果更加明显。当然，设置像素是 BitmapData 对象能做的最简单的操作之一。除了获取和设置像素，BitmapData 对象还有其它二十多种方法，这些方法可用来复制像素，设置阈值，分解，合并，滚动，等等。我个人最喜欢的一个是 perlinenoise 方法，该函数允许我们创建一个随机的有组织的图案。对于制造烟，云和水波纹效果都非常有用。有兴趣的话大家可以试验一下。

读取和嵌入资源

最后一个重点话题是获取外部资源的概念，如在影片中加载位图或外部 SWF 文件。有两种方法，一种是在动画播放时将资源读入，这就是我们所熟知的读取(loading)。另一种方法是在 SWF 编译时嵌入(embed)资源。

读取资源

创建一个 Loader 对象来读取一个资源，这是 flash.display.Loader 类的一个实例。loader 是个显示对象，意味着可以使用 addChild() 方法将它加入到显示列表中，就像 sprite 和 bitmap 一样。然后告诉这个 loader 去读取一个外部 SWF 或外部位图，如 JPEG,PNG,等等。

在 AS 2 中，当处理外部文件路径或 URL 时，只需要使用一个简单的字符串表示路径。在 AS 3 中，则需要创建一个 flash.net.URLLoader 实例，传入表示路径的字符串，并且还需要一个额外的步骤，虽然有些烦人，但是我们还是要习惯这种用法。
这里是一个在运行时读取外部资源的例子(文档类 LoadAsset.as)：

```
package {
 import flash.display.Sprite;
 import flash.display.Loader;
 import flash.net.URLRequest;
 public class LoadAsset extends Sprite {
  public function LoadAsset() {
   init();
  }
  private function init():void {
   var loader:Loader = new Loader();
   addChild(loader);
   loader.load(new URLRequest("picture.jpg"));
  }
 }
}
```

嵌入资源

虽然在有些情况下，在运行时读取资源很合适，但是在有些情况下有一些外部图形只想加载到 SWF 自里面。这时，如果使用 Flash IDE，可以简单地导入这个对象到库中并设置为“为 ActionScript 导出”。但在使用 Flex Builder 2 或 Flex 2 SDK 命令编译器时，没有库，那么如何在 SWF 中加载外部资源呢？

答案是使用[Embed]元数据(metadata)标签嵌入资源，元数据标签是指加到 ActionScript 文件中的非正式 ActionScript 语句。另外，它们指示编译器在编译过程中去做某种事情，[Embed]标签告诉编译器在最终的 SWF 文件中加载一个特殊的外部资源，资源可以是位图或外部 SWF 文件。告诉编译器要嵌入的资源所在的 source 路径的属性，如下：

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

在元数据语句的后面，直接声明一个 Class 类型的变量，如下：

```
[Embed(source="picture.jpg")]
private var Image:Class;
```

现在可以使用这个变量创建一个新的资源实例，如下：

```
var img:Bitmap = new Image();
```

注意创建的这个对象是 Bitmap 类型的。如果嵌入一个外部 SWF 文件，创建的这个对象应该是 Sprite 类型的，如下：

```
[Embed(source="animation.swf")]
private var Anim:Class;
var anim:Sprite = new Anim();
```

这里是一个在 SWF 中嵌入外部 JPEG 的例子：

```
package {
 import flash.display.Sprite;
 import flash.display.Bitmap;
 public class EmbedAsset extends Sprite {
  [Embed(source="picture.jpg")];
  private var Image:Class;
  public function EmbedAsset() {
   init();
  }
  private function init():void {
   var img:Bitmap = new Image();
   addChild(img);
  }
 }
}
```

如果我们使用 Flash IDE ，只要将对象导入到库中并“为 ActionScript 导出”给出一个类名就可以了。不需要使用 [Embed] 元数据标签及类变量，事实上，Flash IDE 编译器甚至不支持 [Embed] 元数据标签。这里只作一个简单的介绍，因为在本书后面的内容中不会用到这个技术，但是很显然这是个非常有用的方法。

本章重要公式

在本章中我们又收集了很多有价值的工具，大多都与颜色有关。

转换为十进制：

```
trace(hexValue);
```

十进制转换为十六进制：

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

颜色合成：

```
color24 = red << 16 | green << 8 | blue;
color32 = alpha << 24 | red << 16 | green << 8 | blue;
```

颜色提取：

```
red = color24 >> 16; green = color24 >> 8 & 0xFF;
blue = color24 & 0xFF; alpha = color32 >> 24;
red = color32 >> 16 & 0xFF; green = color32 >> 8 & 0xFF;
blue = color232 & 0xFF;
```

过控制点的曲线：

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

4.8 小结

这一章没有涉及太多让物体移动的内容，但它确实向你展示了大量创建可视内容的方法，你将在后面章节中学会如何创建动画。具体来说本章包含了下面的内容：

- 颜色，24 位和 32 位。
- drawing API。
- 滤镜。
- Bitmap 和 BitmapData 类。

这些主题给出了当你需要为你的动画生成动态的可表达的内容时所需要的工具，因此这里涉及的内容都是关于 ActionScript 的，你可以使用这里的所有方法直接制作动画。只需要使用一些代码创建内容，改变代码中的变量，然后再渲染他们。

你在本书中将用到本章中所介绍的很多内容，因此很好地理解它们是很有用的。实际上，在下一章中，你将使用本章的一些技术来做第 1 个有用的实验，它将涉及速度和加速度。

第五章　速度与加速度

恭喜各位！至此已经到了真正的动作编程部分，这就意味着：(a) 您已经坚持学习了前面的所有章节；(b) 您感觉前面内容已经会过了，所以跳过前面的章节；(c) 您感觉无聊所以跳过了。但是不管怎么样，要记得如果在日后的学习中遇到了相关的问题，可以回到前面几章寻找答案。

本章以基本运动为基础：速度，向量以及加速度。今后所有的 ActionScript 动画，几乎都会用到这些概念。

速度向量(Velocity)

物体运动的最基本属性就是速度。很多人把速度向量(velocity)和速度(speed)等同，这是不对的，因为速度仅仅是速度向量的一部分，速度向量的概念还包括一个非常重要的因素：方向。速度向量的简单定义是：某个方向上的速度。

如果我开车从 X 位置出发，以每小时 30 英里的速度行驶一个小时，这时如果想找到我可就难了。但是，如果我说以同样的速度向北行驶一个小时，那么大家就可以知道我实际的位置了，这在动画中是非常重要的。

这就是引入速度向量的原因，如果知道物体在某一帧时的位置，那么只要知道它的运动速度和方向，就可以知道物体下一帧所在的位置了。

在介绍速度向量编码之前，先要为大家介绍一些向量的知识，同时为速度向量做一个简单的描述。

向量与速度向量

向量由长度和方向组成。在速度向量中，长度就是速度。向量用带有箭头的线段表示，箭头的长度就是向量的长度，箭头所指的方向就是向量的方向。图 5-1 是一些向量。

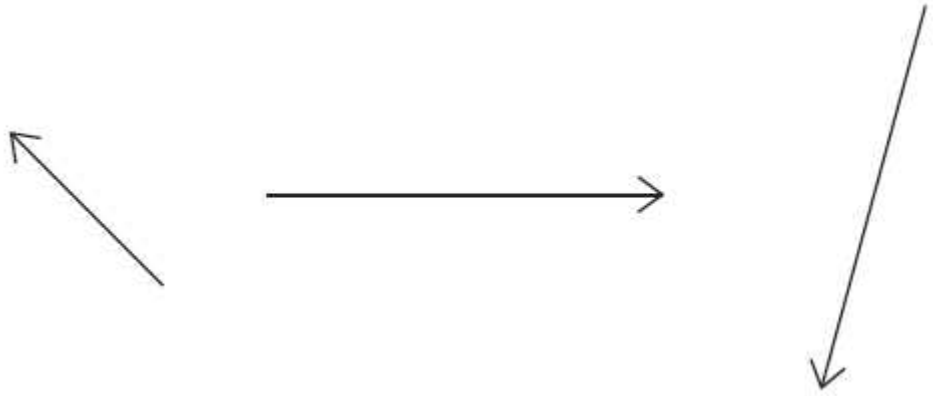


图 5-1 几个向量

需要注意的是，长度总是正数，如果一个长度为负数的向量只表示该向量的反方向，见图解 5-2。

图 5-2 反速度向量为反方向的向量

还要注意向量没有起点，向量不能说明哪里是起点哪里是终点，它仅仅表示出了物体移动的速度与方向。因此，如果两个的方向及长度都相同，即使它们位于不同位置，那么它们仍是两个相等的向量，如图 5-3 所示。

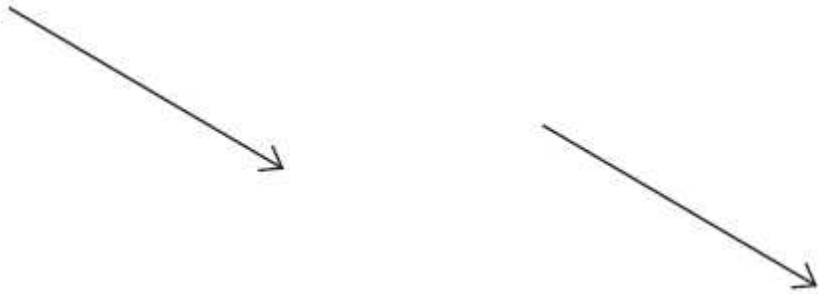


图 5-3 如果向量的方向与长度相同，则它们相等，不考虑位置问题

单轴速度

首先，为了简化这个问题，把速度(向量)只放在一个轴上：x 轴(水平轴)。让物体向从屏幕的左侧到右侧——这样会比较符合习惯,移动速度就是物体每一帧移动的像素值。因此，如果说速度向量在 x 轴上为 5，就意味着物体在每一帧都会右移动 5 个像素。同样，如果速度向量在 x 轴上为 -5，那么物体每一帧就会向左移动 5 个像素。

到现在为止大家都能跟上吗？我们刚刚提到了向量长度等于负值，科学地讲，速度向量实际上应该为 5，而方向应为 180 度。同理， y 轴正半轴上的速度向量应为 90 度(垂直向下)，而负 y 轴负半轴上的速度向量应为 270 或 90 度(垂直向上)。

事实上，当计算 x,y 速度向量的分量时，通常可以记作正数或负数，比如“ x 速度向量为 -5”。在 x 轴上把减号看成“向左”的指示符，在 y 轴上则是“向上”的指示符。在本书中，将用 vx 表示 x 轴的速度向量，用 vy 表示 y 轴的速度向量。 vx 为正数表示向右移动，为负数表示向左移动， vy 为正数表示向下， vy 为负数表示向上。

本章的许多例子都会让物体做出各种移动效果。为了不让每个例子都花时间绘制物体，我们下面创建一个 Ball 类，这样就可以经常重复使用它了。

```
package {
 import flash.display.Sprite;
 public class Ball extends Sprite {
  public var radius:Number;
  private var color:uint;
  public function Ball(radius:Number=40,color:uint=0xff0000) {
   this.radius=radius;
   this.color=color;
   init();
  }
```

```
  public function init():void {
   graphics.beginFill(color);
   graphics.drawCircle(0,0,radius);
   graphics.endFill();
  }
 }
}
```

今后无论何时引用这个类，都要把这个类与工程放在一起，这样只需要使用 new Ball(size, color)就可以了创建一个小球了，或者可以把这个类放在一个特定的位置，并将其所在目录加入类路径中。现在已经有了运动主体，以下是第一个速度向量的示例，文档类 Velocity1.as ：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Velocity1 extends Sprite {
  private var ball:Ball;
  private var vx:Number=5;
  public function Velocity1() {
   init();
  }
  private function init():void {
   ball=new Ball  ;
   addChild(ball);
   ball.x=50;
   ball.y=100;
   addEventListener(Event.ENTER_FRAME,onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   ball.x+= vx;
  }
 }
}
```

在这个例子中，首先设置一个 x 轴速度向量(vx)等于 5。记住是指每一帧 5 像素，所以，在每一帧中， vx 都会被加到 ball 的 x 属性中。 init 方法将小球放到舞台上并为 enterFrame 设置事件处理函数。每走一帧，小球都会在前一帧的位置基础上向右移动 5 个像素。怎么样很不错吧，嗯？

给 vx 一个较大或较小的值，或者给个负数来试试，并观察一下物体运动的方向。

双轴上速度向量

使用两个轴对物体进行移动也非常简单，只需要定义 vx 和 vy，并在每一帧将 vx 加到 x 属性上， vy 加到 y 属性上。下面一个示例(Velocity2.as)：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Velocity2 extends Sprite {
  private var ball:Ball;
```

```
  private var vx:Number=5;
  private var vy:Number=5;
  public function Velocity2() {
   init();
  }
  private function init():void {
   ball=new Ball  ;
   addChild(ball);
   ball.x=50;
   ball.y=100;
   addEventListener(Event.ENTER_FRAME,onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   ball.x+= vx;
   ball.y+= vy;
  }
 }
}
```

大家可以试着改变一下速度变量，别忘了还有负数。

角速度

假如想让物体以每帧3像素的速度向45度的位置移动,在这个描述中我们看不到 vx,vy 的影子。但是，大家已经学习了使用 vx 和 vx 移动物体的例子，回忆一下第三章所讲的三角学，然后见图 5-4 所示，每一帧让小球以 3 像素的速度向 45 度角的位置移动。在这个图中加入一条边后，是不是与图 5-5 非常相似了？恩，这就是一个由已知角度与斜边构成的直角三角形！

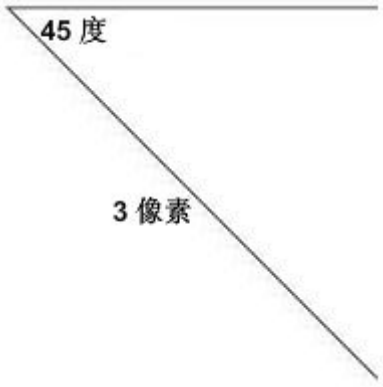


图 5-4 长度及方向

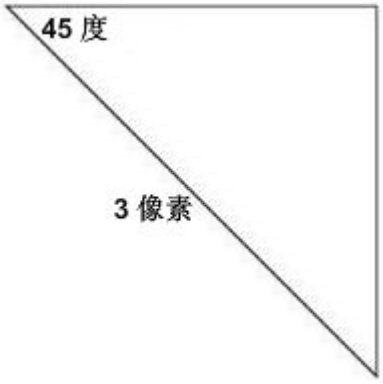


图 5-5 长度及方向形成一个直角三角形

请注意，这个三角形的两条位于 x,y 轴上的直角边。事实上， x 轴上的直角边长度等于小球所要移动的 x 距离， y 轴上的直角边等于 y 距离。不要忘记，在直角三角形中，只要知道一条边和一个角，就可以求出其它所有边和角的信息。因此，已知角度为 45 度，斜边长为 3 像素，就可以使用 Math.cos 和 Math.sin 求出 vx 和 vy 的长度。

角的邻边长度为 vx，因为角的余弦值等于邻边/斜边。也可以说，邻边等于角的余弦值乘以斜边。同样，对边长为 vy 的边，因为角的正弦值等于对边/斜边，或是对边等于正弦乘以斜边。实际使用的代码：

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

在使用 Math 函数之前你还敢忘记将 45 度角转换为弧度值吗！一旦获得了 vx 和 vy 的值，就可以将它们加到物体的 x,y 坐标上，这样就运动起来了。下面一个示例(VelocityAngle.as)代码如下：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class VelocityAngle extends Sprite {
  private var ball:Ball;
  private var angle:Number=45;
  private var speed:Number=3;
  public function VelocityAngle() {
   init();
  }
  private function init():void {
   ball=new Ball
   addChild(ball);
   ball.x=50;
   ball.y=100;
   addEventListener(Event.ENTER_FRAME,onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var radians:Number=angle * Math.PI / 180;
   var vx:Number=Math.cos(angle) * speed;
   var vy:Number=Math.sin(angle) * speed;
   ball.x+= vx;
   ball.y+= vy;
  }
 }
}
```

与前面 vx,vy 主要不同的地方是变成了 angle 和 speed,计算出的速度向量作为局部变量被使用。当然，由于是一个简单的示例，角度(angle)和速度(speed)都不变，那么完全可以只计算一次，然后保存在类中作为变量。而对于更高级的运动来说，角度和速度会是应是不断变化的，所以 vx 和 vy 的值也是变化的。

只需要改变角度(angle)与速度(speed)，就可以改变物体运动的速度及角度。下面，让从向量的角度审视一下这个例子。

向量加法

当在一个平面坐标中有两个向量时，使用向量加法可以求出两个向量的合成向量。合成向量是一条从第一个向量的起点连接到最后一个向量终点的向量。图 5-6 中，可以看到三个向量相加及一个合成向量。

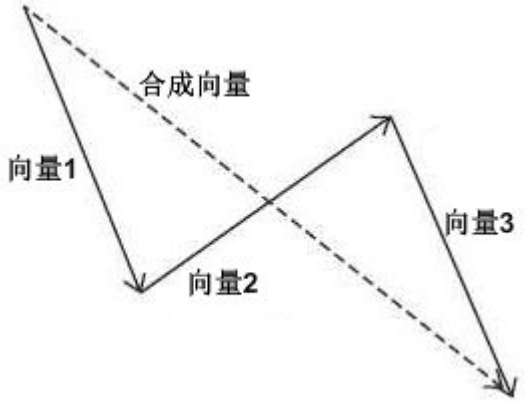


图 5-6 向量加法

结果与向量所在的位置无关。可以说物体是先沿着这条路前进，再沿那条路前进，然后再到另一条路，顺序可以任意选择，或者可以说物体在这三条路上都走过一次。只要给出速度及方向就可以让物体向终点移动。

在上一节例子中，如果将 x 轴的速度向量向右，再将 y 轴的速度向量竖直向下，那么合成向量就是全部向量的总合，见图 5-7。

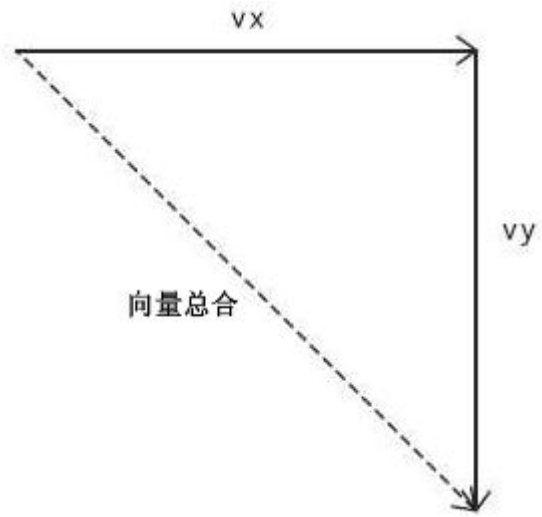


图 5-7 速度向量作为向量

鼠标跟随

让我们使用速度向量的概念解释一下早前的一个问题。回到第三章，我们曾使用一个箭头指向鼠标位置的例子，在这个示例中使用 Math.atan2 计算鼠标与箭头之间的夹角，并使箭头旋转到这个角度上。再根据刚才所学的知识，计算出当前角度的速度向量。这个示例中同样使用 Arrow 类，所以大家要把它找出来，然后与 FollowMouse.as 文档类放在同一目录下：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class FollowMouse extends Sprite {
```

```
  private var arrow:Arrow;
  private var speed:Number = 5;
  public function FollowMouse() {
   init();
  }
  private function init():void {
   arrow = new Arrow();
   addChild(arrow);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var dx:Number = mouseX - arrow.x;
   var dy:Number = mouseY - arrow.y;
   var angle:Number = Math.atan2(dy, dx);
   arrow.rotation = angle * 180 / Math.PI;
   var vx:Number = Math.cos(angle) * speed;
   var vy:Number = Math.sin(angle) * speed;
   arrow.x += vx;
   arrow.y += vy;
  }
 }
}
```

这是一个相当复杂的效果，不过这里大家都能够看懂。先要计算出箭头与鼠标的 x 距离和 y 距离，并使用 Math.atan2 计算出它们的夹角。然后使用这个角度使箭头旋转，再使用 Math.cos 和 Math.sin 与速度相乘计算出 x,y 速度向量，最后将它们加到箭头的坐标上。

速度向量扩展

Sprite 影片，影片剪辑或任何的显示对象都有许多属性可以使用，而这些属性大多数都有比较大的取值范围，可以让我们制作出多种多样的动画。也许，速度向量一词用于这些属性上并不合适，但是在概念上是相同的，所以我也通常使用 v(velocity) 为变量命名的首字母。在一个影片旋转的例子中,通常在每一帧上将物体的 rotation 属性增加一些数值，只要加入更大的数值就可以让物体旋转得更快，反之就会更慢。不论正确与否，我通常将旋转速度变量命名为 vr，表示旋转速度。同样使用 Arrow 影片，文档类 RotationalVelocity.as：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class RotationalVelocity extends Sprite {
  private var arrow:Arrow;
  private var vr:Number = 5;
  public function RotationalVelocity() {
   init();
  }
  private function init():void {
   arrow = new Arrow();
```

```
    addChild(arrow);
    arrow.x = stage.stageWidth / 2;
    arrow.y = stage.stageHeight / 2;
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
   }
   private function onEnterFrame(event:Event):void {
    arrow.rotation += vr;
   }
  }
}
```

速度向量值为 5，方向为顺时针运动。同样的道理，可以将其它属性也加以改变，变量名仍使用 v 字系列，就像这样：

```
arrow.x += vx;
arrow.y += vy;
arrow.alpha += vAlpha;
arrow.rotation += vr;
arrow.scaleX = arrow.scaleY += vScale;
// etc.
```

在本书中看到很多这样的示例，所以希望大家原谅我经常使用 v 字母作开头，它的意思只是指它们是速度向量的“兄弟”。下面来学习加速度。

加速度

通常认为加速度就是使速度加快，减速度就是使速度减慢。没错，在本书中，为加速度作了一个更为科学的定义。

速度(向量)和加速度有很多相同之处,它们都是向量(或矢量)。速度(向量)和加速度(向量)都用量值(大小)和方向进行定义。然而，速度向量是改变物体位置的，而加速度是改变其速度向量的。

想象一下，你坐在车上，然后启动车子，踩油门。什么是加速度？踩下油门以后(等同于加速)，速度向量开始发生变化(速度开始加快，方向的变化于方向盘决定)。过一两秒后，速度将提升至每小时 4 到 5 英里，随后时速会变为 10 英里，20 英里，30 英里等等。发动机以其速度向量驱动汽车前进。

因此，加速度的通俗定义是:增加到物体速度向量上的力量。用 ActionScript 术语可以表示为，加速度就是一个增加到速度向量上的数值。

举一个例子，假如有一架火箭要从 A 星球飞到 B 星球。它的方向于 A 星球与 B 星球的位置决定，调整好方向后，开始点燃火箭。当火箭点燃后前进速度就会越来越快，当指挥官认为火箭的速度已经足够快了，为了保持燃料，就要让火箭慢下来。假设宇宙空间中不存在阻力，火箭以同样的速度继续飞行。当火箭不再点火时，就没有更多的力来驱动它了。因此，就失去了加速度，速度就不再发生变化。

当火箭接近目标时，就需要减慢速度。指挥官应该怎么做？不能使用刹车—也不可能抓住什么东西。这时，指挥官会让火箭转回去，就是朝相反的方向运动，并再次点火。这就使用了负加速度，或者说反方向的加速度。然后这股力量继续改变速度，但是这时是将速度减小，速度会越来越小，最终到达零。理想来说，这时火箭应该正处于星球地面几英寸的位置。

单轴加速度

让我们将前面学过的知识溶入到 Flash 中进行一下实践。与第一个速度向量示例相同，

第一个加速度也只在一个轴上。回到 Ball 类中，下面是第一个示例的代码(Acceleration1.as)：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Acceleration1 extends Sprite {
  private var ball:Ball;
  private var vx:Number=0;
  private var ax:Number=.2;
  public function Acceleration1() {
   init();
  }
  private function init():void {
   ball=new Ball  ;
   addChild(ball);
   ball.x=50;
   ball.y=100;
   addEventListener(Event.ENTER_FRAME,onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   vx+= ax;
   ball.x+= vx;
  }
 }
}
```

开始的速度向量(vx)为零，加速度(ax)为 0.2，在每一帧中加速度都被加入到速度向量中，再加到小球的位置上。

测试一下这个示例，我们会看到小球开始移动得非常慢，而后就会非常快速地向右侧飞行，从而移动出舞台范围。

下面制作一个小球的示例，允许小球拥有加速度和反加速度。在这里使用方向键，我们已经学习过了侦听键盘事件的方法，也学过使用 keyCode 属性找到引发事件的事件对象，再调用事件处理函数。然后与 flash.ui.Keyboard 类中的常量做比较，这个类中包括适合用户读取的键码属性值，比如，Keyboard.LEFT, Keyboard.SPACE, Keyboard.SHIFT 等。目前，只关心左右方向键，Keyboard.LEFT 和 Keyboard.RIGHT，使用它们来改变加速度。这里是代码(Acceleration2.as)：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.events.KeyboardEvent;
 import flash.ui.Keyboard;
 public class Acceleration2 extends Sprite {
  private var ball:Ball;
  private var vx:Number = 0;
  private var ax:Number = 0;
  public function Acceleration2() {
   init();
  }
  private function init():void {
   ball = new Ball();
```

```
    addChild(ball);
    ball.x = stage.stageWidth / 2;
    ball.y = stage.stageHeight / 2;
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
    stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
  }
  private function onKeyDown(event:KeyboardEvent):void {
    if (event.keyCode == Keyboard.LEFT) {
      ax = -0.2;
    } else if (event.keyCode == Keyboard.RIGHT) {
      ax = 0.2;
    }
  }
  private function onKeyUp(event:KeyboardEvent):void {
    ax = 0;
  }
  private function onEnterFrame(event:Event):void {
    vx += ax;
    ball.x += vx;
  }
 }
}
```

在这个例子中，只需要检查是否按下了左右方向键。如果按下了左键，就为 ax 设置一个负值。如果按下了右键，则是正值，如果没有按下则设为零。在 onEnterFrame 方法中，将速度赋值到物体位置上。

测试这个影片，会发现，我们不能完全控制物体的速度。也就是说，不能立即将影片停止运动。如果将它的速度将得过低，就会向反方向运动。

双轴加速度

与速度向量一样，可以同时在 x,y 轴使用加速度。只需要为每一个轴设置一个加速度(用 ax 和 ay 作为变量名)，将它们加入 vx 和 vy，再将 vx,vy 赋给 x,y 属性。在上一个例子中加入 y 轴非常简单，只需要加入：ay 和 vy 变量即可。

以下是代码(Accelertation3.as)：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.events.KeyboardEvent;
 import flash.ui.Keyboard;
 public class Acceleration3 extends Sprite {
  private var ball:Ball;
  private var vx:Number=0;
  private var vy:Number=0;
  private var ax:Number=0;
  private var ay:Number=0;
  public function Acceleration3() {
```

```
    init();
  }
  private function init():void {
   ball=new Ball  ;
   addChild(ball);
   ball.x=stage.stageWidth / 2;
   ball.y=stage.stageHeight / 2;
   addEventListener(Event.ENTER_FRAME,onEnterFrame);
   stage.addEventListener(KeyboardEvent.KEY_DOWN,onKeyDown);
   stage.addEventListener(KeyboardEvent.KEY_UP,onKeyUp);
  }
  private function onKeyDown(event:KeyboardEvent):void {
   switch (event.keyCode) {
    case Keyboard.LEFT :
     ax=-0.2;
     break;
    case Keyboard.RIGHT :
     ax=0.2;
     break;
    case Keyboard.UP :
     ay=-0.2;
     break;
    case Keyboard.DOWN :
     ay=0.2;
     break;
    default :
     break;
   }
  }
  private function onKeyUp(event:KeyboardEvent):void {
   ax=0;
   ay=0;
  }
  private function onEnterFrame(event:Event):void {
   vx+= ax;
   vy+= ay;
   ball.x+= vx;
   ball.y+= vy;
  }
 }
}
```

请注意，本例中将上/下/左/右键的检查放到了 switch 语句中，这与 if 语句的功能相同。

这样就可以让小球在整个屏幕上移动了。试着让物体从左到右移动，然后按“上”键，注意这时 x 速度向量没有受到影响，物体依然保持在 x 轴上的运动速度。

重力加速度

到目前为止，已经讨论了物体对其自己施加的力量为加速度，如汽车和火箭。对于任何通过加速度改变自身速度的力来说，还有很多种，例如重力，磁力，弹力，摩擦力等等。

观察重力(Gravity)有两种方法。一种是从太阳系的广角镜头来看，重力就是两个天体之间的吸引力，这时必需要考虑两个天体间的角度和距离，才能计算出每个天体真正的加速度。

另一种观察重力的方法是使用特写镜头，发生在地球上。存在于现实生活中，在地球上，用物体间的距离决定重力的大小看起来是微乎其微的。虽然科学来讲，当我们在高空或高山上重力会减小一些，但这些变化几乎是感觉不到的。因此，在水平面上模拟重力时，几乎只使用一个固定的值，就像在前面例子中的加速度变量一样。同样也是因为，地球太大而人类太小了，这样一来实际的加速度方向就可以忽略不计了，只需要一个“向下”的力就可以了。换句话讲，无论物体在什么位置，我们都可以放心地在 y 轴上定义重力作为加速度。

放到 ActionScript 代码上来说,只需要定义一个数值作为重力,并在每一帧加入到 vy 上，用一个分数就可以了，比如 0.5 或更小的数。如果用更大的数，物体就会显得太重了。如果用更小的数值，物体看起来会像飘浮的羽毛。当然，这个效果也是很实用的，例如通过不同的重力变化来模拟不同的星球。

下面这个例子加入了重力系统。完整代码在 Gravity.as ，在这里就不把它全部列出来了，仅与 Acceleration3.as 有一点点不同。除了类以外，还有构造函数名也要变换一下，请在最开始的变量列表中加入一个变量：

```
private var gravity:Number = 0.1;
```

在 onEnterFrame 方法中加入一句：

```
private function onEnterFrame(event:Event):void{
 vx += ax;
 vy += ay;
 vy += gravity;
 ball.x += vx;
 ball.y += vy;
}
```

让重力值很小，是为了不让小球很快就离开屏幕，我们还可以使用方向键进行控制。前面所制作的就是一个古老的月球登陆者的游戏。再加入一些漂亮的图形和碰撞检测，就完成了！(后面会学到碰撞检测，而图形就要自己完成了)

回到向量加法，如果以初始向量为起点出发，作为一个向量，每个加速度，重力，或其它附加力都可以看作一个是添加到这个速度向量上的附加向量。把它们全部相加后，就绘制出了一条从起点到终点的连线，也就是合成向量，与加入到 x,y 的力是一样的。现在，想象一下有一个热汽球的影片，也许应该加入一个名为 lift(上升) 的力，这也是一个加在 y 轴上的加速度。不过，这次它是一个负数，表示“向上”。现在，这个物体上已经施加了三个力：方向键的力，重力和上升力。为了让汽球上升，则上升力要略高于重力，这样也是符合逻辑的——如果它们相等，则两种力会相互抵消掉，又回到了起点，这时只有方向键的力起作用。

还可以试一试风力，很明显，这是加在 x 轴上的力。取决于风吹去的方向，可以是正向力也可以是反向力，方向从 0 度到 180 度。

角加速度

我们说过，加速度由力和方向组成，在速度向量中的这两个要素需要分解为 x,y 。如

果大家加以留心，就会明白同样可以使用 Math.cos 和 Math.sin 的方法。代码如下：

```
var force:Number = 10;
var angle:Number = 45; // degrees. Need to convert!
var ax:Number = Math.cos(angle * Math.PI / 180) * force;
var ay:Number = Math.sin(angle * Math.PI / 180) * force;
```

这样在每一个轴上都有了加速度，我们可以刷新每个轴上的速度向量，并更新为物体的位置。

继续使用本章的鼠标跟随示例，并使加速度应用在加速度上。回忆上次的示例，使用鼠标与箭头间的夹角决定 vx 和 vy。这一次，使用同样的方法，计算 ax 和 ay，然后将加速度值累加到速度向量中，再将速度向量赋给 x,y 属性。代码如下(FollowMouse2.as)：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class FollowMouse2 extends Sprite {
  private var arrow:Arrow;
  private var vx:Number = 0;
  private var vy:Number = 0;
  private var force:Number = 0.5;
  public function FollowMouse2() {
   init();
  }
  private function init():void {
   arrow = new Arrow();
   addChild(arrow);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var dx:Number = mouseX - arrow.x;
   var dy:Number = mouseY - arrow.y;
   var angle:Number = Math.atan2(dy, dx);
   arrow.rotation = angle * 180 / Math.PI;
   var ax:Number = Math.cos(angle) * force;
   var ay:Number = Math.sin(angle) * force;
   vx += ax;
   vy += ay;
   arrow.x += vx;
   arrow.y += vy;
  }
 }
}
```

请注意，本例中将 speed 转为 force 并让它的值变得很小，是因为加速度是累加的，我们希望让它开始的时候小一些，这个值很快就会增大。同样注意 vx,vy 被声明为类的变量，可以由类的任意方法对其进行访问。早期它们都由每一帧重新进行计算，但是现在需要它们保存自身的数值，并且每次要进行自加或自减操作。当然，也可以不使用 ax 和 ay 变量，只需要将正弦和余弦的结果直接加在速度向量上就可以，之所以这么写是为了让代码看起来更清晰。

目前，这些代码不是很复杂，对吗？但是回顾一下本章开始时给大家的示例，就可以发现走了有多远。通过学习这些基本规则，可以制作出成百上千的动态的效果——有些动画是

活灵活现的。这一章还没有完！

OK，让我们齐心协力，看看到底能够走多远吧。

制作飞船

接下来，制作一个模拟太空船的例子，计划如下。太空船专由一个类进行绘制，就像前面用过的 Arrow 和 Ball 类一样。使用左右键控制飞船向左右旋转，上键用于点燃飞船。当然，火箭是位于飞船尾部的。因此，火箭的力会使飞船在某个方向上进行加速运动。

首先，需要一架飞船，在一个类中使用绘图 API 代码绘制四条白色的线，作为飞船模型。如果大家比较有艺术天赋的话，可以用 PhotoShop 或 Swift 3D，制作一张位图，并使用嵌入技术将其嵌入，至于嵌入外部位图请见第四章。代码如下(Ship.as)：

```
package {
 import flash.display.Sprite;
 public class Ship extends Sprite {
  public function Ship() {
   draw(false);
  }
  public function draw(showFlame:Boolean):void {
   graphics.clear();
   graphics.lineStyle(1,0xffffff);
   graphics.moveTo(10,0);
   graphics.lineTo(-10,10);
   graphics.lineTo(-5,0);
   graphics.lineTo(-10,-10);
   graphics.lineTo(10,0);
   if (showFlame) {
    graphics.moveTo(-7.5,-5);
    graphics.lineTo(-15,0);
    graphics.lineTo(-7.5,5);
   }
  }
 }
}
```

这是一个公共的 draw 方法，带有 true/false 两个值。这样，就得到了有点火和无点火的飞船，用于表示飞船起动和熄火。图 5-8 和 5-9 所示，有点火的飞船和无点火的飞船。

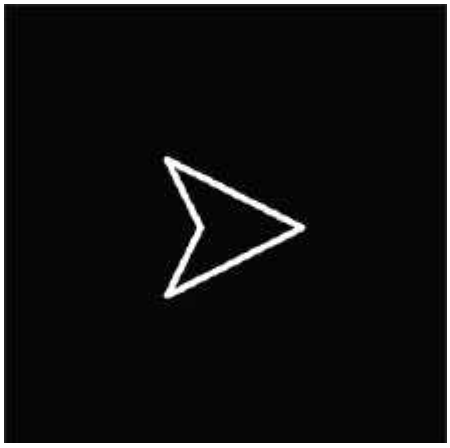


图 5-8 视为太空船

图 5-9 小心火焰

飞船控制

OK，飞船已经有了，接下来就要控制它了。刚才说过，要执行三种控制：左转，右转和点火，分别由左右上三个键控制。本例的代码与 Acceleration3.as 非常相似，用到事件处理函数 keyDown 和 keyUp 还有一个 switch 语句对按键进行分类处理。先把所有代码给大家，随后再进行解释(ShipSim.as)：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.events.KeyboardEvent;
 import flash.ui.Keyboard;
 public class ShipSim extends Sprite {
  private var ship:Ship;
  private var vr:Number=0;
  private var thrust:Number=0;
  private var vx:Number=0;
  private var vy:Number=0;
  public function ShipSim() {
   init();
  }
  private function init():void {
   ship=new Ship  ;
   addChild(ship);
   ship.x=stage.stageWidth / 2;
   ship.y=stage.stageHeight / 2;
   addEventListener(Event.ENTER_FRAME,onEnterFrame);
   stage.addEventListener(KeyboardEvent.KEY_DOWN,onKeyDown);
   stage.addEventListener(KeyboardEvent.KEY_UP,onKeyUp);
  }
  private function onKeyDown(event:KeyboardEvent):void {
```

```
    switch (event.keyCode) {
     case Keyboard.LEFT :
      vr=-5;
      break;
     case Keyboard.RIGHT :
      vr=5;
      break;
     case Keyboard.UP :
      thrust=0.2;
      ship.draw(true);
      break;
     default :
      break;
    }
   }
   private function onKeyUp(event:KeyboardEvent):void {
    vr=0;
    thrust=0;
    ship.draw(false);
   }
   private function onEnterFrame(event:Event):void {
    ship.rotation+= vr;
    var angle:Number=ship.rotation * Math.PI / 180;
    var ax:Number=Math.cos(angle) * thrust;
    var ay:Number=Math.sin(angle) * thrust;
    vx+= ax;
    vy+= ay;
    ship.x+= vx;
    ship.y+= vy;
   }
  }
}
```

首先定义 vr ，旋转速度向量，即飞船旋转的速度。初值为零，意思是没有旋转：

```
private var vr:Number = 0;
```

在 onKeyDown 方法中，如果 switch 语句发现按下了左右方向键，就赋值 vr 为-5 或 5。

```
case Keyboard.LEFT :
vr = -5;
break;
case Keyboard.RIGHT :
vr = 5;
break;
```

然后在 onEnterFrame 处理函数中，加入 vr 作为飞船的当前旋转方向，当按下一个键后，将 vr 重置为零。OK，以上就是旋转的问题。接下来，看一下 thrust(推力)。

声明 thurst 变量指明每一次所用的力。很明显，只有在点火以后火箭才具有加速度，所以在开始之前速度为零：

```
private var thrust:Number = 0;
```

然后在 switch 语句，如果持续按着上键应将 thrust 设为较小的数值，如 0.2。在用到推力(thrust)的时候，要绘制飞船的火焰：

```
case Keyboard.UP :
```

```
thrust = 0.2;
ship.draw(true);
break;
```

当按键释放后，设置 thrust 为零并消除火焰：

```
private function onKeyUp(event:KeyboardEvent):void {
 vr=0;
 thrust=0;
 ship.draw(false);
}
```

认真思考 onEnterFrame 函数后，就会发现 rotation 的角度值于确定推力的大小。将 rotation 转换为弧度制并使用正余弦函数连同 thrust 一起，计算出每轴上的加速度：

```
private function onEnterFrame(event:Event):void {
ship.rotation += vr;
var angle:Number = ship.rotation * Math.PI / 180;
var ax:Number = Math.cos(angle) * thrust;
var ay:Number = Math.sin(angle) * thrust;
vx += ax;
vy += ay;
ship.x += vx;
ship.y += vy;
}
```

测试后，让飞船飞行起来，你会惊喜地发现制作如此复杂的运动原来这么简单。如果大家使用代码绘制飞船的话，请不要忘记将背景色设置为黑色或其它深色，这样才能看出白色的线条。

本章重要公式

在我们的工具箱中又多出了不少工具，来看看吧。

角速度转换为 x,y 速度向量：

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

角加速度(作用于物体上的 force)转换为 x,y 加速度：

```
ax = force * Math.cos(angle);
ay = force * Math.sin(angle);
```

将加速度加入速度向量：

```
vx += ax;
vy += ay;
```

将速度向量加入坐标：

```
movieclip._x += vx;
sprite.y += vy;
```

5.4 小结

这一章讲解了基本的速度和加速度，这两个因素将在你的大多数脚本动画中应用。你已经学习了向量和向量加法。你已经明白了如何实现一个轴上的速度、两个轴上的速度和通过角度来转换为 x 和 y 方向上的分解速度。转换角速度为 x、y 方向上的分解加速度。加到每一个轴上的速度上。将速度加到每一个轴的位置上。

本章最重要的是理解加速度和速度的应用，如下面描述的步骤：

转换角加速度为 x、y 方向上的分解加速度。将加速度加到每一个轴的速度上。将速度和加速度分解到每一个轴的位置上。

在下一章你将在这些概念的基础上进行构建，使用弹力和摩擦力来增加一些环境的交互。

第六章　边界与摩擦力

到目前为止，我们已经学会了如何在影片中绘制图形，并且通过施加外力使影片运动起来。然而，在这些例子中也许会遇到这样的烦恼：物体移动到屏幕外后就不到了。如果在某个角度上运动得过快，那么就没有办法再让物体退回来，只能选择重新运行影片。

我们常常忽视边界的存在如：墙和屋顶，最平常的就是地面。通常在制作太空模拟时，要用环境边界作为一道屏障，保证物体能够在一个可见的范围内运动。

另一个常被忽略的问题是，所处的环境如何改变物体的运动。惯性一词是用来形容物体在空间中穿梭，并保持以同样的方向及速度运动，只有对其施加外力，才会使它的运动发生改变。改变物体速度向量的力，可能是摩擦力的一种——甚至可以是空气的阻力。目前，我们已经能够模拟真空环境下的物体运动了，但是大家一定还想模拟真实环境下的物体运动。那么本章就要解决前面这两个问题。首先，学习边界环境下物体的运动，然后学习如何模拟阻力，Let's go。

环境边界

先来学习边界的设置，就像我们的日常活动一样：开运动会，做某项工作，盖房子等，这里边界是指为这项活动保留的活动空间。意思是“我只关心发生在这个范围内的事情，如果超出了这个范围，就不再关注它了。”

当物体超出了这个范围后，我们可以对它进行一些操作。可以再把它移回来，或把它从关注的对象中移除，另一种选择是跟随它。只要物体是运动的，那么它就有机会离开这个范围。当物体离开后，我们可以选择忘记它，或将它移动回来，或跟随它。我们将介绍前两种方法，不过先要确定边界的位置，再学习如何定义边界。

设置边界

通常，边界就是一个矩形。从最简单的例子开始——基于舞台大小的边界，在 AS 3 中，舞台由一个名为 stage 的属性表示，它是每个显示对象的一部分。所以在文档类(继承自 MovieClip 或 Sprite)中，我们可以直接使用这个属性访问舞台及舞台的相关属性。如果在播放器窗口改变大小后希望舞台的尺寸与播放器尺寸相匹配的话，就应该设置这两个属性：

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

请注意，还需要导入 flash.display.StageAlign 和 flash.display.StageScaleMode 这两个类。这样一来，影片的左上边界将为零，而右下边界将为 stage.stageWidth 和 stage.stageHeight。可将它们保存为变量，如下：

```
private var left:Number = 0;
private var top:Number = 0;
private var right:Number = stage.stageWidth;
private var bottom:Number = stage.stageHeight;
```

要知道，如果使用变量保存设置的话，那么舞台大小的改变将不会对这些变量产生影响。如果要使用一个固定的区域作为边界的话，这样做是非常合适的。

然而，如果使用整个舞台区域作边界的话，那么即使舞台大小发生了改变，只要在代码中直接调用 stage.stageWidth 和 stage.stageHeight 就可以了。

比如，我们可以为对象创建一个居住的“房间”，在这个例子中，边界可以是 top = 100, bottom = 300, left = 50, right = 400。OK，边界已经有了，用它们能做什么呢？判断所有移动的对象，看它们是否仍在这个空间内，这里可以使用 if 语句，简化的样式如下：

```
if(ball.x > stage.stageWidth) {
    // do something
} else if(ball.x < 0) {
    // do something
}
if(ball.y > stage.stageHeight) {
    // do something
} else if(ball.y < 0) {
    // do something
}
```

使用 if 和 else 语句判断边界，如果小球 x 坐标大于右边界，就意味着它超出了右边界。但不可能同时超出左边界，所以不需要再用一条 if 语句进行判断。因此，只需要在第一个 if 语句失败后再判断左边边界即可,顶部和底部也是如此。然而,物体有可能在 x,y 轴上同时超出边界，所以要把这两个判断语句分开。如果物体出界后，应该对它们执行什么样的操作呢？答案有四种：

- 将对象移除；
- 重置到舞台上，像生成一个新对象一样(重置对象)；
- 重置到舞台上，将同一个对象放置在不同位置；
- 将其反弹回去。

我们从最简单的移除对象开始。

移除对象

如果对象是不断产生的，那么使用一次性删除对象的方法是非常有效的。被删除的对象将会由新的对象所取代，这样舞台就永远不为空。但也不能生成太多的可移动对象，因为这样会使 Flash Player 变慢。

调用 removeChild(*对象名*)，删除影片或显示对象，会将对象实例从舞台上移除。请注意，被移除的显示对象仍然存在，只是看不到而已。如果要将该对象彻底删除，还应该调用 delete *对象名* 将其完全删除。

如果移动的对象只是一些影片实例，并且物体的运动只由 enterFrame 函数进行处理，那么要停止整个程序的执行只需调用 removeEventListener(Event.ENTER_FRAME, onEnterFrame);就可以了。另一方面，如果运动的对象很多，要通过持续执行代码使每个对象都动起来（除了被删除的对象外），就应该在数组中保存所有对象的引用，然后循环这个数组使里面的每个对象都动起来。

随后，当删除了其中的某一个对象后，使用 Array.splice 方法同时将该对象的引用在数组中删除，后面会有代码。

所以，如果想要删除影片，就要知道边界在哪儿，使用 if 语句来完成：

```
if(ball.x > stage.stageWidth ||
ball.x < 0 ||
ball.y > stage.stageHeight ||
ball.y < 0)
{
removeChild(ball);
}
```

|| 符号意思是“或者”。这句话是说“如果物体超出了右边界，或左边界，或上边界，或下边界，就将它删除。”。

这里有一个小问题，也许大家现在还没有意识到，见图 6-1。

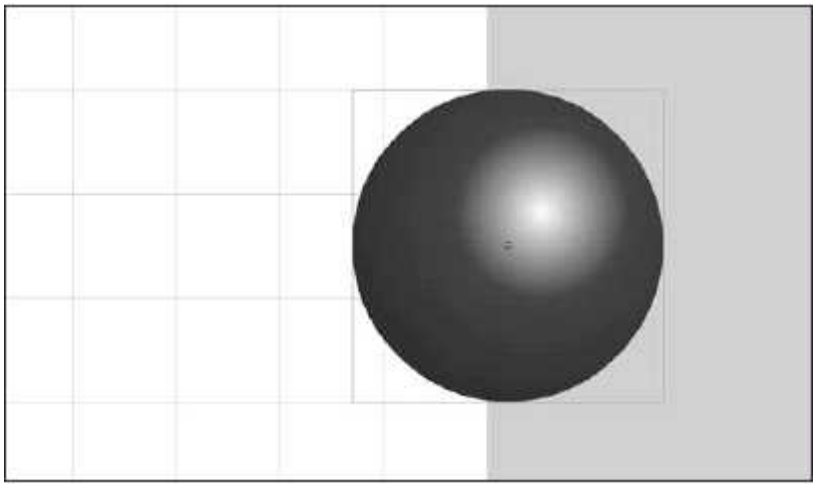


图 6-1 小球并没有完成超出舞台，就被移除了

图 6-1 中，小球的位置由中心的注册点的位置决定，而注册点超出了屏幕的右边界，小球将被删除。如果小球的运动足够快，也许看上去问题不大。但如果运动得非常缓慢，每帧运动一像素的话，那会是怎样？我们会看着它走向屏幕的边界，但还差一半没有走完就被移除了！这就像一个演员只离开了舞台的一半就把戏服脱掉了，破坏了塑造人物的形象。

所以，要让小球完全离开场景，要等到它完全离开视野后再采取处理。

实现这个计划，需要考虑到物体的宽度。因为注册点在中心，所以，可以将宽度的一半保存为 radius 属性。代码如下：

```
if(ball.x - ball.radius > stage.stageWidth ||
ball.x + ball.radius < 0 ||
ball.y - ball.radius > stage.stageHeight ||
ball.y + ball.radius < 0)
{
removeChild(ball);
}
```

如图 6-2 所示。

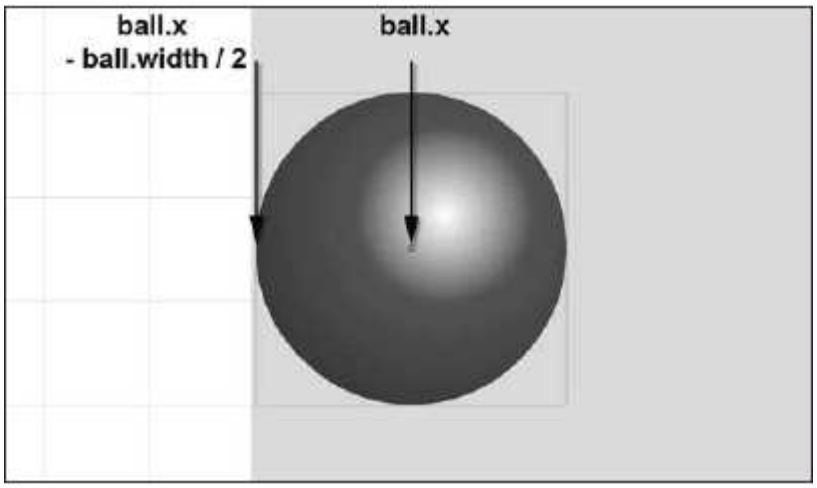


图 6-2 小球完全超出了舞台，可放心移除

虽然使用球形或圆形是个比较特殊的例子，但这样的代码对于所有注册点在中心的物体来说，都是适用的。

下面一个例子中，将要使用 Ball 类，这个类前面也用过，但这次要加上一点新的内容。为该类加入公共属性 vx 和 vy ，让每个小球都有自己的速度向量。全部代码如下：

```
package {
 import flash.display.Sprite;
 public class Ball extends Sprite {
```

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

对于纯面向对象编程来说，也许不会用这些公共属性，而是将它们转为私有属性，再使用 getter/setter 方法进行操作。但是为了方便起见，就不再遵循这个规则了，仅使用公共属性作替代。下面一个文档类，Removal.as，设置了许多小球，并在它们离开舞台后进行删除：

```
package {
 import flash.display.Sprite;
 import flash.display.StageAlign;
 import flash.display.StageScaleMode;
 import flash.events.Event;
 public class Removal extends Sprite {
  private var count:int=20;
  private var balls:Array;
  public function Removal() {
   init();
  }
  private function init():void {
   stage.scaleMode=StageScaleMode.NO_SCALE;
   stage.align=StageAlign.TOP_LEFT;
   balls=new Array();
   for (var i:int=0; i < count; i++) {
    var ball:Ball=new Ball(10);
    ball.x=Math.random() * stage.stageWidth;
    ball.y=Math.random() * stage.stageHeight;
    ball.vx=Math.random() * 2 - 1;
    ball.vy=Math.random() * 2 - 1;
    addChild(ball);
    balls.push(ball);
   }
   addEventListener(Event.ENTER_FRAME,onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   for (var i:Number=balls.length - 1; i > 0; i--) {
    var ball:Ball=Ball(balls[i]);
```

```
    ball.x+= ball.vx;
    ball.y+= ball.vy;
    if (ball.x - ball.radius > stage.stageWidth ||
     ball.x + ball.radius < 0 ||
     ball.y - ball.radius > stage.stageHeight ||
     ball.y + ball.radius < 0) {
     removeChild(ball);
     balls.splice(i,1);
     if (balls.length <= 0) {
      removeEventListener(Event.ENTER_FRAME,onEnterFrame);
     }
    }
   }
  }
 }
}
```

应该很容易理解吧。首先。创建 20 个小球的实例，随机安排它们在舞台上的位置，给出随机的 x,y 速度向量，并将它们加入显示列表，然后 push 到数组中。

onEnterFrame 方法通过速度向量使小球移动，判断边界，并将出界的小球删除。请注意，不但要将小球从显示列表中删除，同时还要使用 Array.splice 函数将数组中的引用也删除。Array.splice 有两个参数：开始删除元素的索引，删除元素的个数。在这个例子中，只删除一个元素即当前索引处的元素。

大家也许注意到了本例中的 for 语句，与其它例子中的不太一样：

```
for(var i:Number = balls.length - 1; i > 0; i--)
```

这是让 for 循环倒着执行，遍历整个数组。因为如果在数组中使用 splice ，数组的索引就会改变。

最后，在删除数组元素后，还要判断数组长度是否小于零。如果小于零，则撤消对 enterFrame 的侦听器。

重置对象

下一个策略是将超出舞台范围的对象进行重置。实际上就是重新配置，重新设置属性。当一个对象离开了舞台后，它就没有作用了，不过，可以将其重置到舞台上，让它作为一个新对象再加入进来。永远不要担心对象的数量过多，因为这个数量是固定不变的。这个技术用于制作喷泉效果非常合适：一串粒子不停地喷射，超出舞台的粒子重新加入到水流中。

现在就来制作一个喷泉效果。作为喷泉的粒子，我们同样使用 Ball 类，但只把它设为 2 像素的大小并给它一个随机的颜色。水源在舞台底部的中心位置，所有的粒子都从这里发出，当它们超出舞台边界后，将会被重置回来。所有粒子都以一个随机的负 y 速度和一个随机的 x 速度作为开始。这样就会上喷射，并伴有轻微的左右移动。当粒子重置后，它们的速度向量也将被重置，同时粒子也要受到重力的牵引。文档类 Fountain.as：

```
package {
 import flash.display.Sprite;
 import flash.display.StageAlign;
 import flash.display.StageScaleMode;
 import flash.events.Event;
 public class Fountain extends Sprite {
  private var count:int = 100;
```

```
  private var gravity:Number = 0.5;
  private var balls:Array;
  public function Fountain() {
   init();
  }
  private function init():void {
   stage.scaleMode = StageScaleMode.NO_SCALE;
   stage.align=StageAlign.TOP_LEFT;
   balls = new Array();
   for (var i:int = 0; i < count; i++) {
    var ball:Ball = new Ball(2, Math.random() * 0xffffff);
    ball.x = stage.stageWidth / 2;
    ball.y = stage.stageHeight;
    ball.vx = Math.random() * 2 - 1;
    ball.vy = Math.random() * -10 - 10;
    addChild(ball);
    balls.push(ball);
   }
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   for (var i:Number = 0; i < balls.length; i++) {
    var ball:Ball = Ball(balls[i]);
    ball.vy += gravity;
    ball.x += ball.vx;
    ball.y += ball.vy;
    if (ball.x - ball.radius > stage.stageWidth ||
    ball.x + ball.radius < 0 ||
    ball.y - ball.radius > stage.stageHeight ||
    ball.y + ball.radius < 0) {
     ball.x = stage.stageWidth / 2;
     ball.y = stage.stageHeight;
     ball.vx = Math.random() * 2 - 1;
     ball.vy = Math.random() * -10 - 10;
    }
   }
  }
 }
}
```

请试着加入风力效果（提示：设置 wind 变量，并加入到 vx ）。

屏幕环绕

下一个处理越界对象的方法，我称其为屏幕环绕。概念很简单：一个对象超出了屏幕的左边界，就让它在屏幕右边出现；在右边出界，则将它置到左边；上面出界就回到下面。明白了吧。这个思想与重置对象的概念非常相似，只是位置有所不同。

再回到前面那个老游戏，小行星。第五章的这个飞船影片有些问题：一旦太空船飞出了

舞台，就很难将它找回。如果使用屏幕环绕技术，那么影片超出屏幕边界的距离不会大于一像素。

让我们为太空船示例重新加入一些行为，这里是文档类(ShipSim2.as)，新的部分用粗体标出：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.events.KeyboardEvent;
 import flash.ui.Keyboard;
 import flash.display.StageAlign;
 import flash.display.StageScaleMode;
 public class ShipSim2 extends Sprite {
  private var ship:Ship;
  private var vr:Number = 0;
  private var thrust:Number = 0;
  private var vx:Number = 0;
  private var vy:Number = 0;
  public function ShipSim2() {
   init();
  }
  private function init():void {
   stage.scaleMode = StageScaleMode.NO_SCALE;
   stage.align=StageAlign.TOP_LEFT;
   ship = new Ship();
   addChild(ship);
   ship.x = stage.stageWidth / 2;
   ship.y = stage.stageHeight / 2;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
   stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
   stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
  }
  private function onKeyDown(event:KeyboardEvent):void {
   switch (event.keyCode) {
    case Keyboard.LEFT :
     vr = -5;
     break;
    case Keyboard.RIGHT :
     vr = 5;
     break;
    case Keyboard.UP :
     thrust = 0.2;
     ship.draw(true);
     break;
    default :
     break;
   }
  }
  private function onKeyUp(event:KeyboardEvent):void {
   vr = 0;
```

```
    thrust = 0;
    ship.draw(false);
   }
   private function onEnterFrame(event:Event):void {
    ship.rotation += vr;
    var angle:Number = ship.rotation * Math.PI / 180;
    var ax:Number = Math.cos(angle) * thrust;
    var ay:Number = Math.sin(angle) * thrust;
    vx += ax;
    vy += ay;
    ship.x += vx;
    ship.y += vy;
    var left:Number = 0;
    var right:Number = stage.stageWidth;
    var top:Number = 0;
    var bottom:Number = stage.stageHeight;
    if (ship.x - ship.width / 2 > right) {
     ship.x = left - ship.width / 2;
    } else if (ship.x + ship.width / 2 < left) {
     ship.x = right + ship.width / 2;
    }
    if (ship.y - ship.height / 2 > bottom) {
     ship.y = top - ship.height / 2;
    } else if (ship.y < top - ship.height / 2) {
     ship.y = bottom + ship.height / 2;
    }
   }
  }
}
```

这里同样使用了第五章的 Ship 类，请确保该文件与这个类在同一路径下。由于飞船是用白色线条绘制的，所以不要忘记将背景色改为黑色。大家可以看到，在新的类中加入了边界的定义及判断。

反弹

本节中的弹性处理也许是最常用也是最复杂的，但也没有屏幕环绕那么复杂，所以不用担心。当检测到物体超出舞台后，开始应用弹性，不改变改变物体的位置，只改变它的速度向量。方法很简单：如果物体超出了左、右边界，只需要使它的 x 速度向量取反。如果超出了上、下边界，只需要让 y 速度向量取反。坐标轴取反非常简单，只需要乘以 -1 。如果速度向量等于 5，则变成-5。如果是-13，则变成 13。代码也非常简单：vx *= -1 或 vy *= -1。

对于反弹的时机来说，我们不并希望等到物体完全超出了舞台后开始反弹。同样，也不希望出现半张图片的效果。比如，往墙上扔一个球，不希望球的一半进入墙体后再反弹回来。因此，首先要判断出小球首次超出边界的瞬间。然后，将小球的运动路径取反，再加上小球宽度/高度的一半。比如：

```
if(ball.x  -  ball.radius > right) . . .
```

就要变为

```
if(ball.x + ball.radius > right) . . .
```

两者的区别见图 6-3。

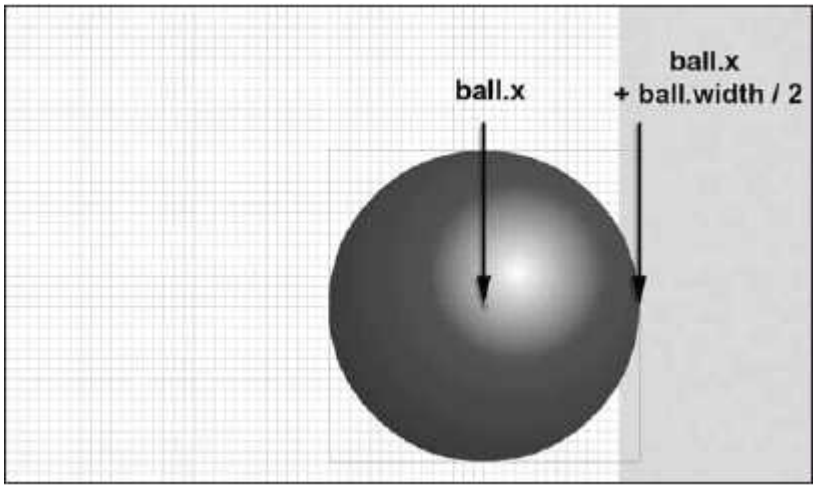


图 6-3 小球略微超出了舞台，需要反弹

只要物体超出了舞台，即使只有一少部分，都要使它的速度向量取反，并且还需要将物体重新定位到边界处，这就形成了一个非常明显的撞击反弹的效果。如果不调整物体的位置，到下一帧，在物体移动之前，也许仍然处在边界外。如果这样的话，物体的速度向量又将取反，则向墙内运动！就会产生物体进出墙体的情形，然后在这附近振荡。

x 轴的全部 if 语句如下：

```
if(ball.x + ball.radius > right) {
    ball.x = right  -  ball.radius;
    vx *= -1;
}
else if(ball.x  -  ball.radius < left) {
    ball.x = left + ball.radius;
    vx *= -1;
}
```

图 6-4 所示，重新置位后小球的位置。

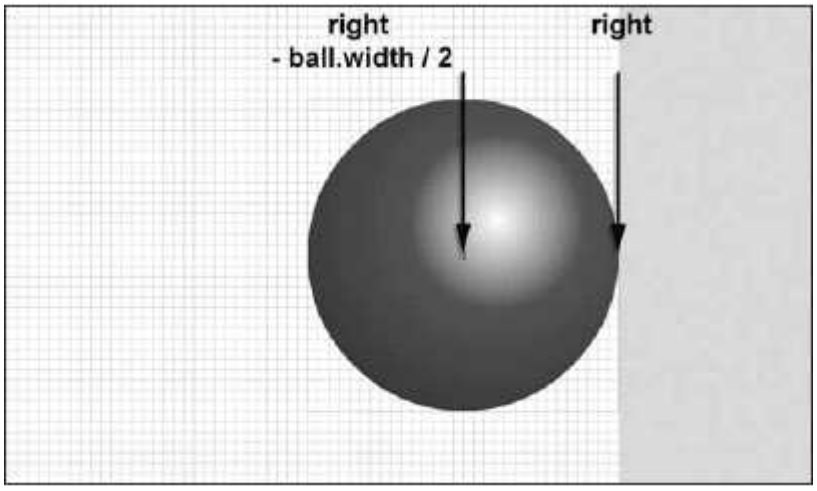


图 6-4 小球被重新置位到依靠边界处

反弹的步骤如下：

- 判断物体是否超出了边界；
- 如果是，将其置到边界处；
- 然后将它的速度向量取反。

叙述部分就到这里，下面来看代码。下一个示例，仍使用 Ball 类，文档类(Bouncing.as)：

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

```
import flash.display.StageScaleMode;
import flash.events.Event;
public class Bouncing extends Sprite {
 private var ball:Ball;
 private var vx:Number;
 private var vy:Number;
 public function Bouncing() {
  init();
 }
 private function init():void {
  stage.scaleMode=StageScaleMode.NO_SCALE;
  stage.align=StageAlign.TOP_LEFT;
  ball=new Ball  ;
  ball.x=stage.stageWidth / 2;
  ball.y=stage.stageHeight / 2;
  vx=Math.random() * 10 - 5;
  vy=Math.random() * 10 - 5;
  addChild(ball);
  addEventListener(Event.ENTER_FRAME,onEnterFrame);
 }
 private function onEnterFrame(event:Event):void {
  ball.x+= vx;
  ball.y+= vy;
  var left:Number=0;
  var right:Number=stage.stageWidth;
  var top:Number=0;
  var bottom:Number=stage.stageHeight;
  if (ball.x + ball.radius > right) {
   ball.x=right - ball.radius;
   vx*= -1;
  } else if (ball.x - ball.radius < left) {
   ball.x=left + ball.radius;
   vx*= -1;
  }
  if (ball.y + ball.radius > bottom) {
   ball.y=bottom - ball.radius;
   vy*= -1;
  } else if (ball.y - ball.radius < top) {
   ball.y=top + ball.radius;
   vy*= -1;
  }
 }
}
```

多进行几次测试，观察不同角度的运动，并试着将速度向量变大或变小。不得不承认，这是一个数学计算与现实情况不完全一致的例子。从图 6-5 中可以看到，小球实际撞击墙面的位置与模拟的位置的差别。

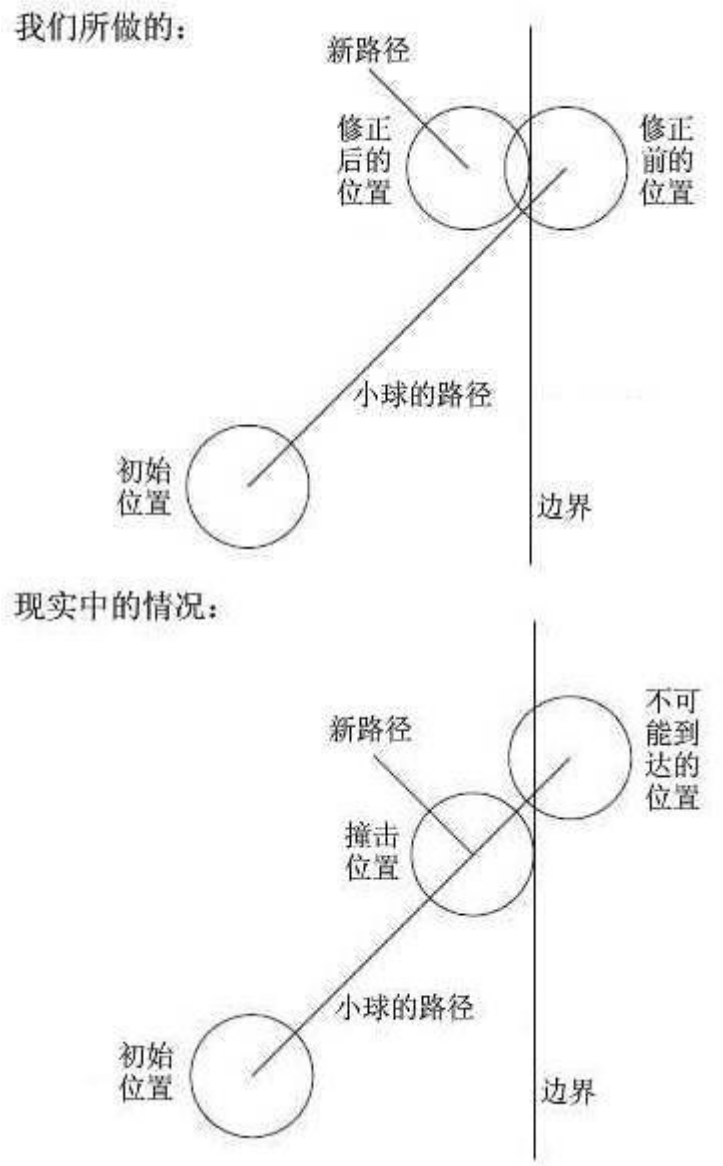


图 6-5 这个技术并不完美，但非常简单，高效且与实际情况非常接近

要达到真正的位置，就要使用更为复杂的计算方法。虽然我们完全可以做到这一点(使用第三章三角学)，但是我保证人们不会注意到这些细微的差别。如果在某些模拟中，对位置的要求至关重要的话，那么我们还需要去查阅其它的书籍，并重新考虑使用哪种软件来完成。但是对于大多用 Flash 制作的数游戏或视觉效果而言，用这种方法已经足够了。

再比如，我们手握一个橡胶球，然后松手使它落到地上，当小球落到地面时，会发生向上的反弹,但它永远不会回到我们的手中。这是因为在反弹的过程中小球损失了一部分能量。损失的能量也许是制造声音了，也可能是制造热量了，地面或周围的空气也会吸收一部分能量。重要一点是小球在发生反弹后运动的速度比之前要慢一些。换句话讲，它在反弹到某一轴上时，损失了速度矢量。

这样一来，可以简单地重建一个 Flash，前面使用 -1 作为弹性系数。这就意味着，物体反弹的力量为 100%。为了制造能量的损失，可以用弹性系数作为阻力。为了能在代码中使用这个参数，最好将它定义成一个变量。创建一个名为 bounce 的变量，并将它设置为 -0.7 这样的数字：

```
private var bounce:Number = -0.7;
```

在 if 语句中使用 bounce 这个变量代替 -1。试过后大家会发现与现实中的弹性是多么地相似。为 bounce 变量使用不同的系数，试试效果吧。

我们在学习知识的时候，最好与前面所学的原理结合起来。大家可以看一下本书的源文件文档类 Bouncing2.as，在这个示例中还包括了重力，我相信大家可以通过已掌握的知识自行将它加上去。

摩擦力(Friction)

假设有一张纸，将它撕碎后用力丢向空中。纸片会受到重力向下的牵引(y 轴)，当我们松手后，纸片的 x 轴起初运动得非常快，但很快 x 轴的运动速度又归为零。

很显然，这里面没有负的加速度，但是纸片的速度向量却发生了改变，这就是摩擦力，阻力或阻尼。虽然它不是一种严格意义上的力，但作用是相同的，因为它改变了物体的速度。原理是，摩擦力只改变速度向量中的速度，而不会改变运动的方向。

那么如何使用代码来实现摩擦力呢？这里有两种方法。就像生活中很多事情一样，有一种正确的方法和一种简易的方法。我们会分别讲述这两种方法，首先从“正确”的方法开始吧。

摩擦力，正确的方法

摩擦力是与速度向量相反的力，假设有一个摩擦力的数值，就可以将它从速度向量中减去。事实上，是从速度向量的量值或速度中减去，不能只是简单地从 x,y 轴上减去。这样做的话，如果物体沿着一定角度运动，其中的一个分速度会提前到达零，使得物体继续垂直或水平地运动一会儿，结果看起来非常奇怪。

所以，我们要做的就是根据速度和方向找出角速度。使用 vx 和 vy 的平方和开平方后求出速度(是的，这就是勾股定理，第三章的内容)。再使用， Math.atan2(vy, vx) 求出角度，代码如下：

```
var speed:Number = Math.sqrt(vx * vx + vy * vy);
var angle:Number = Math.atan2(vy, vx);
```

然后就可以从速度向量中减去速度。如果摩擦力大于速度，速度就变为零，计算代码如下：

```
if (speed > friction) {
    speed -= friction;
} else {
    speed = 0;
}
```

这样一来，还需要使用正弦和余弦将角速度转换回 vx 和 vy，如下：

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

工作量不小吧？这里是全部文档类的内容 Friction1.as：

```
package {
 import flash.display.Sprite;
 import flash.display.StageAlign;
 import flash.display.StageScaleMode;
 import flash.events.Event;
 public class Friction1 extends Sprite {
  private var ball:Ball;
  private var vx:Number = 0;
  private var vy:Number = 0;
  private var friction:Number = 0.1;
  public function Friction1() {
   init();
```

```
  }
  private function init():void {
   stage.scaleMode = StageScaleMode.NO_SCALE;
   stage.align=StageAlign.TOP_LEFT;
   ball = new Ball();
   ball.x = stage.stageWidth / 2;
   ball.y = stage.stageHeight / 2;
   vx = Math.random() * 10 - 5;
   vy = Math.random() * 10 - 5;
   addChild(ball);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var speed:Number = Math.sqrt(vx * vx + vy * vy);
   var angle:Number = Math.atan2(vy, vx);
   if (speed > friction) {
    speed -= friction;
   } else {
    speed = 0;
   }
   vx = Math.cos(angle) * speed;
   vy = Math.sin(angle) * speed;
   ball.x += vx;
   ball.y += vy;
  }
 }
}
```

这就是反比例速度向量的代码。摩擦力设置为 0.1，然后给小球一个随机的 x,y 速度向量。在 onEnterFrame 方法中，speed 和 angle 的计算方法同前面所介绍的。如果 speed 小于 friction 则相减，否则让 speed 等于零。然后，重新计算 vx 和 vy ，最后将它们加入到坐标上。

总共用了十几行代码和四个三角函数才完成，大家也许能提出更好的写法，但却不能简化这个计算过程。我想你一定同意去看看那个简便的方法吧。

摩擦力，简便的方法

大家也许都猜到了，简便的方法不像前面的方法那样精确，但是我敢打赌不会有人会注意到这些细微的变化。只用两行简单的乘法即可搞定，我们所要做的就是用摩擦力乘以 x,y 速度向量，摩擦力常用的值大约为 0.9 或 0.8。因此，在每一帧， vx 和 vy 的值都将变为上一次的 80% 或 90%。理论上，速度向量会无限接近零，但永远不会等于零。在实际应用中，计算机计算如此小的数字的能力是有限的，所以最终都会取整为零。

这种方法最好的一点是速度向量永远不会变为负数,所以不需要进行判断。同样,x,y 轴的速度向量也是同比率变化的，所以不需要再将进行繁琐的转换。

只需要将前面例子中的 firction 变量设为 0.9 ,然后按如下代码改变 onEnterFrame 方法(可在文档类 Friction2.as 中找到)：

```
private function onEnterFrame(event:Event):void {
    vx *= friction;
```

```
    vy *= friction;
    ball.x += vx;
    ball.y += vy;
}
```

的确简便了不少吧！测试几次，能看出与之前的不同吗？平心而论，这个效果看起来更真实。

摩擦力应用

让我们回到熟悉的飞船上，现在让宇宙空间具有摩擦力。在 ShipSim2.as 类中加入 friction 变量：

```
private var friction:Number = 0.97;
```

其它的类变量继续延用，还要改变 onEnterFrame 方法，如下(ShipSimFriction.as)：

```
private function onEnterFrame(event:Event):void {
 ship.rotation += vr;
 var angle:Number = ship.rotation * Math.PI / 180;
 var ax:Number = Math.cos(angle) * thrust;
 var ay:Number = Math.sin(angle) * thrust;
 vx += ax;
 vy += ay;
 vx *= friction;
 vy *= friction;
 ship.x += vx;
 ship.y += vy;
 var left:Number = 0;
 var right:Number = stage.stageWidth;
 var top:Number = 0;
 var bottom:Number = stage.stageHeight;
 if (ship.x - ship.width / 2 > right) {
  ship.x = left - ship.width / 2;
 } else if (ship.x + ship.width / 2 < left) {
  ship.x = right + ship.width / 2;
 }
 if (ship.y - ship.height / 2 > bottom) {
  ship.y = top - ship.height / 2;
 } else if (ship.y < top - ship.height / 2) {
  ship.y = bottom + ship.height / 2;
 }
}
```

只加了三行代码，感觉就不一样了。

任何使用速度向量的地方都可以加入摩擦力。比如在物体的旋转上(变量 vr)应用摩擦力，会使旋转的速度慢下来直至停止。大家可以在第五章的旋转箭头中试验一下。这个手段可以应用在所有的物体上，比如轮盘，电风扇或飞船推进器。

本章重要公式

让我们回顾一下本章介绍过的重要公式

移除出界对象：

```
if(sprite.x - sprite.width / 2 > right ||
sprite.x + sprite.width / 2 < left ||
sprite.y  -  sprite.height / 2 > bottom ||
sprite.y + sprite.height / 2 < top)
{
    // 移除影片的代码
}
```

重置出界对象：

```
if(sprite.x - sprite.width / 2 > right ||
sprite.x + sprite.width / 2 < left ||
sprite.y  -  sprite.height / 2 > bottom ||
sprite.y + sprite.height / 2 < top)
{
    // 重置影片的位置和速度
}
```

屏幕环绕出界对象：

```
if (sprite.x - sprite.width / 2 > right) {
 sprite.x = left - sprite.width / 2;
} else if (sprite.x + sprite.width / 2 < left) {
 sprite.x = right + sprite.width / 2;
}
if (sprite.y  -  sprite.height / 2 > bottom) {
 sprite.y = top  -  sprite.height / 2;
} else if (sprite.y + sprite.height / 2 < top) {
 sprite.y = bottom + sprite.height / 2;
}
```

摩擦力应用（正确方法）：

```
speed = Math.sqrt(vx * vx + vy * vy);
angle = Math.atan2(vy, vx);
if (speed > friction) {
 speed -= friction;
} else {
 speed = 0;
}
vx = Math.cos(angle) * speed;
vy = Math.sin(angle) * speed;
```

摩擦力应用（简便方法）：

```
vx *= friction;
vy *= friction;
```

6.4 小结

本章涉及了一个物体与它自身的环境的交互——确切的说，一个物体与它的领域的边界和他的领域自身的交互。你已经学习了处理一个物体与它的领域的边界和他的领域自身的交互。你已经学习了处理一个物体离开边界后的各种方法，包括移除、重新生成、折回和回弹。并且，你现在知道了可能比你之前想知道的更多的关于摩擦力的知识。而使用这些简单的技术，你可以使你电影中的物体移动起来更有真实感。在下一章中，你将研究允许用户与物体对象进行交互操作。

第七章　交互动画：移动物体

我们最初的目标就是要制作出流畅的交互动画，多数都是通过鼠标进行交互的。在第二章里面曾介绍过鼠标事件，但没有涉及到具体的应用。

本章将踏出交互动画的第一步。我们将学会如何处理拖拽，抛落及投掷。但首先要从基本的鼠标按下与释放说起。

按下及释放影片

鼠标可真是件了不起的发明，虽说只是个简单的设备。实际上鼠标只负责两件事：检测移动及点击按钮。计算机用获得的这些信息可以做很多事：通过获知鼠标指针的位置，确定当发生点击的位置，移动的速度，及确定双击事件的发生。当我们从事件的角度来看这些问题时，可以归结为点击与移动（当然，现在的鼠标还配有滚轮，跟踪球或是比一台廉价手机还多的按钮，但现在我们考虑最基本的鼠标种类）。

一次点击事件可分为两部分：鼠标键按下时的事件及鼠标弹上来的事件。通常情况下，这两个事件是在一瞬间发生的。有些时候，这两个事件会被时间和移动分隔开，通常解释为拖拽——按下，移动，最后释放。本章就围绕下面三件事展开：鼠标按下，鼠标弹起，以及发生在它们中间的移动。

对于鼠标事件的处理，在 AS 3 中确实发生了很大的变化，所以需要全面地重新学习一下这些基础问题，拥有坚实的基础是非常重要的。AS 3 事件体系的结构非常合理也非常科学，而早先版本的 AS 有时看起来就像巫术一样。

鼠标事件只能由 Sprite 影片，影片剪辑或其它交互对象在鼠标经过它们的图形时产生。在 AS 2 中，这些也许只对某些鼠标事件起作用，而对其它的却不起作用，这就显得非常混乱。同时也使如 onRelase 和 onReleaseOutside 这样的复合事件成为必需品。

另一个本质上的改变是在嵌套对象与鼠标事件之间的。在 AS 2 中，没有办法使用影片剪辑内部的影片来侦听事件。外层的影片剪辑可以捕获所有的鼠标事件，而后事件就不再向下流通。而在 AS 3 中，就没有这些限制，使用影片剪辑或 Sprite 影片或嵌套影片进行侦听都没有问题。

需要注意的是主要影片事件是 mouseDown, mouseUp, 和 mouseMove。它们都被制作成了 MouseEvent 类的静态属性：

- MouseEvent.MOUSE_DOWN
- MouseEvent.MOUSE_UP
- MouseEvent.MOUSE_MOVE

mouseDown 事件，是当鼠标指针处于某个影片的图形时，按下鼠标后发生的。等同于 AS 2 中的 onPress。

mouseUp 事件，是当鼠标指针处于某个影片的图形时，释放鼠标后发生的。等同于 AS 2 中的 onRelease。

mouseMove 事件，是当鼠标移动时发生的——但只在鼠标移动到该物体或影片时才发生。这点与 AS 2 中不同，在 AS 2 中使用 onMouseMove 时，无论鼠标何时移动，无论指针在哪，都会将这个事件的信息传达给所有的影片剪辑。

然而，有时我们希望在忽略指针位置的情况下，侦听鼠标移动、弹起或按下。在 AS 2 中，使用 onMouseMove, onMouseUp 和 onMouseDown ，都不会关注鼠标的位置。而在 AS 3 中，虽然这些方法有所不同，但只要使用 stage 来侦听 mouseDown, mouseUp 和 mouseMove，同样会将事件信息传达给所有的影片剪辑。

OK，说了不少，让我们先来看个例子吧。本章第一个示例，文档类 MouseEvents.as，继续使用前几章的 Ball 类，代码如下：

```
package {
 import flash.display.Sprite;
 import flash.events.MouseEvent;
 public class MouseEvents extends Sprite {
  public function MouseEvents() {
   init();
  }
  private function init():void {
   var ball:Ball=new Ball  ;
   ball.x=100;
   ball.y=100;
   addChild(ball);
   ball.addEventListener(MouseEvent.MOUSE_DOWN,onMouseDownBall);
   ball.addEventListener(MouseEvent.MOUSE_UP,onMouseUpBall);
   ball.addEventListener(MouseEvent.MOUSE_MOVE,onMouseMoveBall);
   stage.addEventListener(MouseEvent.MOUSE_DOWN,onMouseDownStage);
   stage.addEventListener(MouseEvent.MOUSE_UP,onMouseUpStage);
   stage.addEventListener(MouseEvent.MOUSE_MOVE,onMouseMoveStage);
  }
  private function onMouseDownBall(event:MouseEvent):void {
   trace("mouse down - ball");
  }
  private function onMouseUpBall(event:MouseEvent):void {
   trace("mouse up - ball");
  }
  private function onMouseMoveBall(event:MouseEvent):void {
   trace("mouse move - ball");
  }
  private function onMouseDownStage(event:MouseEvent):void {
   trace("mouse down - stage");
  }
  private function onMouseUpStage(event:MouseEvent):void {
   trace("mouse up - stage");
  }
  private function onMouseMoveStage(event:MouseEvent):void {
   trace("mouse move - stage");
  }
 }
}
```

这个类只不过是建立了前面说过的三种鼠标事件的处理函数，先为 ball 建立侦听，再为 stage 建立侦听。通过这些可以让我们知道事件是何时发生的。大家可以通过这个文件来明白到底什么时候什么地方会触发这些事件，有些需要注意的地方：

- ball 事件只发生在鼠标经过 ball 的时候。
- 特别要注意， ball 的 mouseMove 事件只在鼠标经过 ball 的时候才发生。
- 无论鼠标在哪，都会获得 stage 事件——即使经过 ball 的时候也会发生。这样一来，就会得到两个事件——一个是 ball 的，一个是 stage 的。
- 不可能在没有 mouseDown 事件的情况下，就出现了 mouseUp 事件。

现在大家已经掌握了本章的一些重要事件的基础，下面开始学习拖拽。

拖拽影片

拖拽影片有两种方法：使用 mouseMove 事件或使用 startDrag/stopDrag 方法。本章会介绍这两种方法，首先学习如何使用 mouseMove 实现拖拽，这会给大家一些处理 mouseMove 事件的经验，让各位更深入的理解事件是如何运行的。

使用 mouseMove 执行拖拽

通过手动处理 mouseMove 事件，可以更新影片的位置，使影片每次都移动到鼠标指针的位置，这种方法是每次移动多个影片对象的唯一方法。在使用某个影片作为鼠标指针时，常会用到这种方法。一个影片跟随鼠标的位置，如果这时还要拖拽其它的影片，该怎么办？解决办法就是使用 mouseMove 作鼠标跟随，而不是常用的拖拽方法(startDrag/stopDrag)。因此，这是一个非常实用的技术。

基本策略是：在 mouseDown 事件中，建立一个 mouseMove 处理函数。这个处理函数用于将 ball 的 x,y 坐标赋给当前鼠标的位置。在 mouseUp 事件中，再移除这个处理函数。

这里有一个技巧，我们要为移动的影片设置 mouseDown 侦听器，而对其它两个事件则在 stage 中进行侦听。因为，有时可能拖拽的速度比 Flash 刷新影片位置的速度要快。这样一来，如果用影片侦听 mouseUp 和 mouseMove 事件，将有可能不会响应，下一时刻也许鼠标指针就已经不在影片上了。请记住，无论鼠标在哪儿， stage 都将把这些事件发送出去。下一个文档类，MouseMoveDrage.as，看过后会更加清楚：

```
package {
 import flash.display.Sprite;
 import flash.events.MouseEvent;
 public class MouseMoveDrag extends Sprite {
  private var ball:Ball;
  public function MouseMoveDrag() {
   init();
  }
  private function init():void {
   ball = new Ball();
   ball.x = 100;
   ball.y = 100;
   addChild(ball);
   ball.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
  }
  private function onMouseDown(event:MouseEvent):void {
   stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
   stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
  }
  private function onMouseUp(event:MouseEvent):void {
   stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
   stage.removeEventListener(MouseEvent.MOUSE_MOVE,
   onMouseMove);
  }
  private function onMouseMove(event:MouseEvent):void {
   ball.x = mouseX;
   ball.y = mouseY;
  }
```

```
 }
}
```

最初，只在 ball 上侦听 mouseDown 事件。然后，在 onMouseDown 方法中增加 stage 的侦听器，用于侦听 mouseUp 和 mouseMove 事件。onMouseMove 方法将 ball 的位置更新为鼠标位置。当不再使用拖拽时，通过 onMouseUp 方法移除从 stage 中移除 mouseUp 和 mouseMove 侦听器。从这些设置中也许大家已经发现了一个问题，当点击到 ball 的边缘进行拖拽时，我们发现 ball 突然将中心点对齐到鼠标指针位置上了。这是因为我们设置 ball 的 x,y 坐标完全等于鼠标坐标。大家可以在点击鼠标时，找出鼠标与 ball 的偏移量，然后在进行拖拽时将它加入小球的位置上。把它留做练习，如果大家感兴趣的话不防试一下。下面来看看拖拽影片的常用方法。

使用 startDrag/stopDrag 执行拖拽

所有的 Sprite 影片和影片剪辑都有名为 startDrag 和 stopDrag 的内置方法，用于实现拖拽。这种方法唯一的缺点就是一次只能拖拽一个对象。

概念非常简单，在 mouseDown 处理函数中调用 starDrag。在 mouseUp 处理函数中调用 stopDrag。

调用 starDrag 时可以没有参数，我们将前一个文档类中的 onMouseDown 和 onMouseUp 方法，改为如下代码（或者见 Drag.as）：

```
private function onMouseDown(event:MouseEvent):void {
 stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
 ball.startDrag();
}
private function onMouseUp(event:MouseEvent):void {
 stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
 ball.stopDrag();
}
```

这样一来，可以删除 onMouseMove 方法，因为留在这里也没有用。

太简单了吧，嗯？测试一下，发现 ball 的拖拽位置就是开始点击在 ball 上的位置，没有变化。如果想让它跳到影片中心点上，可以在 startDrag 中传入参数 ture。

也可以使用 left,top,right,bottom 坐标限制拖拽在一个矩形区域内进行。在 AS 3 中有一些变化，要传入一个 Rectangle 对象（flash.geom.Rectangle 类的实例，不要忘了导入）作为参数。在 Rectangle 的构造函数中传入这四个参数，方法如下：

```
var rect:Rectangle = new Rectangle(10, 10, 200, 200);
```

这是 startDrag 的全部语法：

```
startDrag(锁定中心, 矩形边界)
```

使用下面这条语句改变上一个例子中的 startDrag：

```
ball.startDrag(false, new Rectangle(100, 100, 300, 300));
```

再提醒一句，不要忘了导入 Rectangle 类！

结合运动代码的拖拽

目前为止我们已经完全可以在 Flash 中实现简单的拖拽与抛落动画了。但是，如果不对物体进行拖拽的话，那么它们又会静静地呆在那儿。下面让我们在动画中加入一些如速度向量、加速度或弹力等内容。在前一章的 Bouncing2.as 文档类中，已经完成了速度向量，重力和反弹。这是个不错的开始，再在里面加入拖拽与抛落的代码似乎会非常合理，来试一

下。做出的结果应该与下面这些代码相似(DragAndMove1.as)：

```
package {
 import flash.display.Sprite;
 import flash.display.StageAlign;
 import flash.display.StageScaleMode;
 import flash.events.Event;
 import flash.events.MouseEvent;
 public class DragAndMove1 extends Sprite {
  private var ball:Ball;
  private var vx:Number;
  private var vy:Number;
  private var bounce:Number=-0.7;
  private var gravity:Number=.5;
  public function DragAndMove1() {
   init();
  }
  private function init():void {
   stage.scaleMode=StageScaleMode.NO_SCALE;
   stage.align=StageAlign.TOP_LEFT;
   ball=new Ball  ;
   ball.x=stage.stageWidth / 2;
   ball.y=stage.stageHeight / 2;
   vx=Math.random() * 10 - 5;
   vy=-10;
   addChild(ball);
   ball.addEventListener(MouseEvent.MOUSE_DOWN,onMouseDown);
   addEventListener(Event.ENTER_FRAME,onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   vy+= gravity;
   ball.x+= vx;
   ball.y+= vy;
   var left:Number=0;
   var right:Number=stage.stageWidth;
   var top:Number=0;
   var bottom:Number=stage.stageHeight;
   if (ball.x + ball.radius > right) {
    ball.x=right - ball.radius;
    vx*= bounce;
   } else if (ball.x - ball.radius < left) {
    ball.x=left + ball.radius;
    vx*= bounce;
   }
   if (ball.y + ball.radius > bottom) {
    ball.y=bottom - ball.radius;
    vy*= bounce;
   } else if (ball.y - ball.radius < top) {
    ball.y=top + ball.radius;
    vy*= bounce;
```

```
    }
  }
  private function onMouseDown(event:MouseEvent):void {
   stage.addEventListener(MouseEvent.MOUSE_UP,onMouseUp);
   ball.startDrag();
  }
  private function onMouseUp(event:MouseEvent):void {
   stage.removeEventListener(MouseEvent.MOUSE_UP,onMouseUp);
   ball.stopDrag();
  }
 }
}
```

这里只是在原先的代码中加入了 onMouseDown 和 onMouseUp 处理函数。

测试后，大家很快就会发现在执行拖拽时存在着一些问题。是的，拖拽确实存在，但同时还伴随着物体运动的代码，感觉就像是小球从手中脱落一样。为了让拖拽时不产生移动，需要一些方法来开启或关闭执行移动的代码。最简单的方法就是在开始拖拽时，删除 enterFrame 函数，而在停止拖拽时再将它设置回来。这种方法只在 onMouseDown 和 onMouseUp 方法中多加了几行代码：

```
private function onMouseDown(event:MouseEvent):void {
 stage.addEventListener(MouseEvent.MOUSE_UP,onMouseUp);
 ball.startDrag();
 removeEventListener(Event.ENTER_FRAME,onEnterFrame);
}
private function onMouseUp(event:MouseEvent):void {
 stage.removeEventListener(MouseEvent.MOUSE_UP,onMouseUp);
 ball.stopDrag();
 addEventListener(Event.ENTER_FRAME,onEnterFrame);
}
```

测试后，感觉已经很接近理想的效果了，但还有些小问题。拖拽物体时，只是进行拖拽，当抛落小球时，运动代码会从上一次离开时的位置继续开始运动。原因是物体的速度向量依然保持离开时的状态。这就会使鼠标释放时，小球还保持向某个方向移动，显得很不自然。这样，我们只需要将 vx 和 vy 设置为零，写在拖拽或释放时都可以。让我们将它放入执行拖拽的函数中：

```
private function onMouseDown(event:MouseEvent):void {
    vx = 0;
    vy = 0;
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    ball.startDrag();
    removeEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

这样问题基本都解决了，结合速度向量，加速度和反弹，实现了完整的拖拽与抛落的特征。全部代码见文档类 DragAndMove2.as。

这里还留下一个问题；当抛落小球时，它是垂直下落的—— x 轴上没有运动。虽说这个动作没有问题，但是有些单调。如果可以将小球抛出去，并让它在抛出的方向运动，那将是个很棒的交互运动效果。好的，下面就来看一下，它是如何实现的。

投掷

在 Flash 中，投掷是一个什么概念？它意味着当我们点击一个对象时开始对它进行拖拽，

再沿某个方向移动一段距离，松开鼠标后，物体沿拖拽的方向继续移动。

在拖拽的过程中，需要确定为移动物体所需的速度向量，然后在鼠标释放后，将这个速度向量的值再赋给物体。换句话讲，如果每帧向左拖拽影片 10 个像素的话，当鼠标松开时，它的速度向量应该是 vx = -10。

设置新的速度向量来对大家来说没有问题。只需要将 vx,vy 的值赋给物体，如图 7-1 所示，但是如何确定这两个数值还需要一点小技巧。事实上，计算拖拽的距离与应用速度向量的原理是相互对应的。在应用速度向量时，我们将速度向量加到物体原坐标的位置上，从而形成新的位置，所以这个公式可以写成：旧的位置 + 速度向量 = 新的位置。为了计算出移动的速度向量，只需要将这个等式做一个简单的变形：速度向量 = 新位置 - 旧位置。

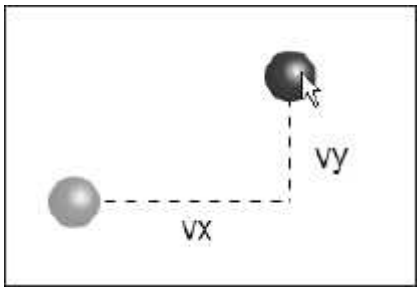


图 7-1 小球被拖拽到新的位置。速度向量为上一个位置到当前位置的距离

在拖拽影片时，会在每一帧形成新的位置。用当前帧的位置减去前一帧的位置，就可知道这一帧所移动的距离。这就是每帧移动距离的速度向量！

我们来做一个例子，将问题简化为在一个轴上拖拽影片。在某一帧上，知道物体的 x 位置是 150。下一帧时，知道它的 x 位置是 170。因此，这一帧内，物体在 x 轴上移动了 20 像素，那么物体的 x 速度向量就是 +20。当鼠标释放后，需要物体还能按 x 轴为 20 的速度，继续运动。所以，应该设 vx = 20。

这就需要对前面的类做一些改变。首先，需要知道每一帧的速度向量，因此就需要加入 enterFrame 事件的处理函数。需要创建一个新的方法用来计算拖拽的速度向量，这个方法就叫 trackVelocity。同时还需要两个变量来存储旧的 x,y 位置，可以将它们命名为 oldX 和 oldY，并将它们声明为类变量。一旦开始拖拽，就需要知道小球的位置并存储为 oldX 和 oldY。在移除了 onEnterFrame 后，加入 trackVelocity 作为侦听器。onMouseDown 方法的完整内容如下：

```
private function onMouseDown(event:MouseEvent):void {
    oldX = ball.x;
    oldY = ball.y;
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    ball.startDrag();
    removeEventListener(Event.ENTER_FRAME, onEnterFrame);
    addEventListener(Event.ENTER_FRAME, trackVelocity);
}
```

然后，在 trackVelocity 方法中，把 oldX 和 oldY 从当前位置中减去。这样就求出了当前的速度向量，可以在 vx 和 vy 中直接保存它们。随后再重新设置 oldX 和 oldY 为小球的当前坐标。

```
private function trackVelocity(event:Event):void {
    vx = ball.x - oldX;
    vy = ball.y - oldY;
    oldX = ball.x;
    oldY = ball.y;
}
```

最后，当释放小球时，再交换 enterFrame 函数。这时将 trackVelocity 移除，再加入 onEnterFrame：

```
private function onMouseUp(event:MouseEvent):void {
```

```
    stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    ball.stopDrag();
    removeEventListener(Event.ENTER_FRAME, trackVelocity);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

拖拽的同时速度向量就已经计算出来了，保存在 vx 和 vy 中。在开启移动代码时，小球将会沿拖拽的方向移动。结果就是：将小球投掷出去！这里是最后一个文档类(Throwing.as)：

```
package {
 import flash.display.Sprite;
 import flash.display.StageAlign;
 import flash.display.StageScaleMode;
 import flash.events.Event;
 import flash.events.MouseEvent;
 public class Throwing extends Sprite {
  private var ball:Ball;
  private var vx:Number;
  private var vy:Number;
  private var bounce:Number = -0.7;
  private var gravity:Number = .5;
  private var oldX:Number;
  private var oldY:Number;
  public function Throwing() {
   init();
  }
  private function init():void {
   stage.scaleMode = StageScaleMode.NO_SCALE;
   stage.align=StageAlign.TOP_LEFT;
   ball = new Ball();
   ball.x = stage.stageWidth / 2;
   ball.y = stage.stageHeight / 2;
   vx = Math.random() * 10 - 5;
   vy = -10;
   addChild(ball);
   ball.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   vy += gravity;
   ball.x += vx;
   ball.y += vy;
   var left:Number = 0;
   var right:Number = stage.stageWidth;
   var top:Number = 0;
   var bottom:Number = stage.stageHeight;
   if (ball.x + ball.radius > right) {
    ball.x = right - ball.radius;
    vx *= bounce;
   } else if (ball.x - ball.radius < left) {
```

```
            ball.x = left + ball.radius;
            vx *= bounce;
        }
        if (ball.y + ball.radius > bottom) {
          ball.y = bottom - ball.radius;
          vy *= bounce;
        } else if (ball.y - ball.radius < top) {
              ball.y = top + ball.radius; vy *= bounce;
          }
        }
        private function onMouseDown(event:MouseEvent):void {
            oldX = ball.x; oldY = ball.y; stage.addEventListener(MouseEvent.MOUSE_UP,
            onMouseUp);          ball.startDrag();
            removeEventListener(Event.ENTER_FRAME,onEnterFrame);
            addEventListener(Event.ENTER_FRAME,trackVelocity);
        }
        private function onMouseUp(event:MouseEvent):void {
            stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
            ball.stopDrag();
            removeEventListener(Event.ENTER_FRAME, trackVelocity);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
          }
        private function trackVelocity(event:Event):void {
            vx = ball.x - oldX;
            vy = ball.y - oldY;
            oldX = ball.x;
            oldY = ball.y;
        }
    }
}
```

这是个相当厉害的交互动画，在 Flash 中重现了现实世界的物理学。试改变 gravity, bounce 等变量的值，观察变化的情况。

7.4 小结

这一章并不长，但它涉及了一些非常有价值的内容，并进一步深入了交互。到现在为止，你应该可以拖动任何物体、使它下落和将它抛出去了。

最重要的是，你已经使用了用来完成一个真正专业的交互作品的很多小细节进行操作。在后面的章节中，你将会看到允许用户与影片中的物体交互的很多其他的方法。难点是我们的进度会更快一些，但是如果你已经掌握了这些基础，你将会做得很好。

第八章　缓动与弹性运动

很难相信我们居然用了七章才把基础的内容介绍完，现在进入第八章，这里是高级内容的起点。从这里开始内容也开始变得越来越有趣了，前面的章节都是些常用的概念与技术。从今天开始，每章只着重介绍一两种特殊的运动。

本章将介绍缓动运动（成比例速度）与弹性运动（成比例加速度），不用担心它们只是两个名词术语，这章可以快速地略读。我会给出很多例子程序，可以使大家充分了解这项技术的强大。

成比例运动

缓动(ease)与弹性(spring)联系紧密。这两种方法都是将对象（通常指 Sprite 或 MovieClip）从某一点移动到目标点。使用缓动运动(Easing)，如同让影片滑动到目标并停止。使用弹性运动(Springing)，会产生向前或向后的反弹，最终停止在目标点位。两种方法具有一些共同点：

- 需要一个目标点；
- 确定到目标点的距离；
- 成比例地将影片移动到目标点——距离越远，移动速度越快。

缓动运动(easing)与弹性运动(springing)的不同之处在于移动的比例。缓动运动时，速度与距离成正比，离目标越远，物体运动速度越快。当物体与目标点非常非常接近时，就几乎不动了。

弹性运动时，加速度与距离成正比。如果物体与目标离得很远，再用上加速度，会使移动速度非常快。当物体接近目标时，加速度会减小，但依然存在！物体会飞过目标点，随后再由反向加速度将它拉回来。最终，用摩擦力使其静止。

下面，我们分别看一下这两种方法，先从缓动(easing)开始。

缓动(Easing)

首先说明缓动的种类不只有一种。在 Flash IDE 中，制作补间动画时，我们就可以看到 “缓动输入”（ease in） 和 “缓动输出”（ease out）。下面所讨论的缓动类型与运动补间的 “缓动输出”相似。在本章后面的 “高级缓动”一节，将会给大家一个网站连接，在那里可以学习制作所有缓动的效果。

简单缓动

简单缓动是个非常基础概念，就是将一个物体移到别处去。创建这个“运动效果”时，希望物体能在几帧内慢慢移动到某一点。我们可以求出两点之间的夹角，然后设置速度，再使用三角学计算出 vx 和 vy，然后让物体运动。每一帧都判断一下物体与目标点的距离，如果到达了目标则停止。这种运动还需要一定条件的约束才能实现，但如果要让物体运动得很自然，显然这种方法是行不通的。

问题在于物体沿着固定的速度和方向运动，到达目标点后，立即停止。这种方法，用于表现物体撞墙的情景，也许比较合适。但是物体移动到目标点的过程，就像是某个人明确地知道他的目的地，然后向着目标有计划地前进，起初运动的速度很快，而临近目标点时，速度就开始慢下来了。换句话讲，它的速度向量与目标点的距离是成比例的。

先来举个例子。比如说我们开车回家，当离家还有几千米的距离时，要全速前进，当离开马路开进小区时速度就要稍微慢一点儿。当还差两座楼时就要更慢一点儿。在进入车库时，速度也许只有几迈。当进入停车位时速度还要更慢些，在还有几英尺的时候，速度几乎为零。

如果大家注意观察就会发现，这种行为就像关门、推抽屉一样。开始的速度很快，然后逐渐慢下来。

在我们使用缓动使物体归位时，运动显得很自然。简单的缓动运动实现起来也非常简单，比求出夹角，计算 vx,vy 还要简单。下面是缓动的实现策略：

1. 确定一个数字作为运动比例系数，这是个小于 1 的分数；
2. 确定目标点；
3. 计算物体与目标点的距离；
4. 用距离乘以比例系数，得出速度向量；
5. 将速度向量加到当前物体坐标上；
6. 重复 3 到 5 步。图 8-1 解释了这一过程。

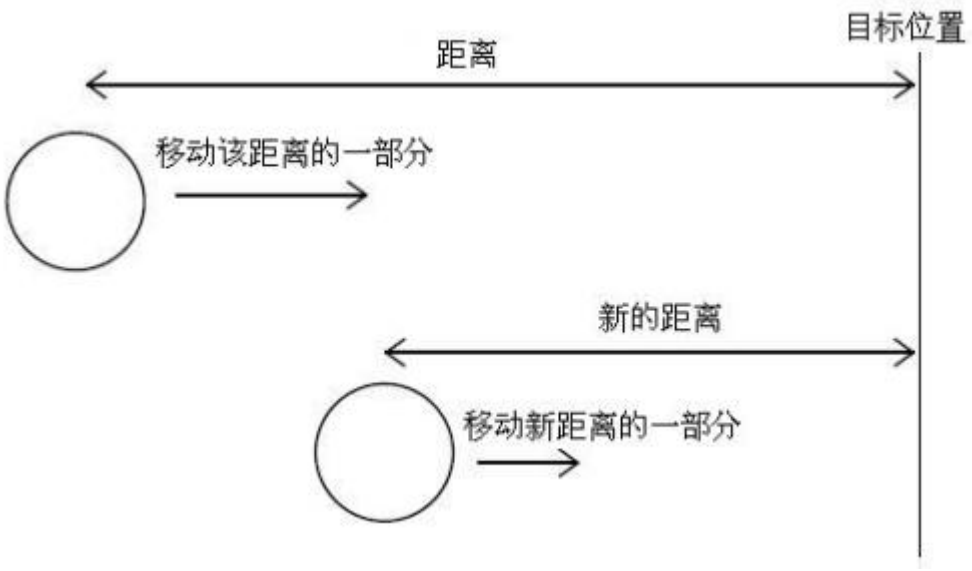


图 8-1 简单缓动

我们先来解释一下这个过程，看看在 ActionScript 中是怎样实现的。

首先，确定一个分数作为比例系数。我们说过，速度与距离是成比例的。也就是说速度是距离的一部分。比例系数在 0 和 1 之间，系数越接近 1，运动速度就会越快；系数越接近 0，运动速度就会越慢。但是要小心，系数过小会使物体无法到达目标。开始我们以 0.2 作为系数，这个变量名就叫 easing。初始代码如下：

```
var easing:Number = 0.2;
```

接下来，确定目标。只需要一个简单的 x,y 坐标，选择舞台中心坐标再合适不过了。

```
var targetX:Number = stage.stageWidth / 2;
var targetY:Number = stage.stageHeight / 2;
```

下面，确定物体到达目标的距离。假设已经有一个名为 ball 影片，只需要从 ball 的 x,y 坐标中减去目标的 x,y。

```
var dx:Number = targetX - ball.x;
var dy:Number = targetY - ball.y;
```

速度等于距离乘以比例系数：

```
vx = dx * easing;
vy = dy * easing;
```

下面，大家知道该怎么做了吧：

```
ball.x += vx;
ball.y += vy;
```

最后重复步骤 3 到步骤 5，因此只需加入 enterFrame 处理函数。

让我们再看一下这三个步骤，以便将它们最大程度地简化：

```
var dx:Number = targetX - ball.x;
var dy:Number = targetY - ball.y;
vx = dx * easing;
vy = dy * easing;
ball.x += vx;
ball.y += vy;
```

把前面四句简化为两句：

```
vx = (targetX - ball.x) * easing;
vy = (targetY - ball.y) * easing;
ball.x += vx;
ball.y += vy;
```

如果大家觉得还不够精简，还可以进一步缩短：

```
ball.x += (targetX - ball.x) * easing;
ball.y += (targetY - ball.y) * easing;
```

在开始学习使用缓动时，也许大家会比较喜欢用详细的句型，让程序看上去更加清晰。但是当你使过几百次后，就会更习惯用第三种写法。下面，我们选用第二种句型，以加强对速度的理解。

现在就来看一下脚本动作，依然延用 Ball 类。以下是文档类 Easing1.as：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Easing1 extends Sprite {
  private var ball:Ball;
  private var easing:Number=0.2;
  private var targetX:Number=stage.stageWidth / 2;
  private var targetY:Number=stage.stageHeight / 2;
  public function Easing1() {
   trace(targetX,targetY);
   init();
  }
  private function init():void {
   ball=new Ball  ;
   addChild(ball);
   addEventListener(Event.ENTER_FRAME,onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var vx:Number=(targetX - ball.x) * easing;
   var vy:Number=(targetY - ball.y) * easing;
```

```
    ball.x+= vx;
    ball.y+= vy;
   }
  }
}
```

试改变 easing 的值，观察运动效果。

下面，大家可以让小球变成可以拖拽的，与第七章所做的拖拽与抛落效果很像。在点击小球时开始拖拽，同时，删除 enterFrame 处理函数并且用 stage 侦听 mouseUp。在 mouseUp 函数中，停止拖拽，删除 mouseUp 方法，并重新开始 enterFrame。下面是文档类 Easin2.as ：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.events.MouseEvent;
 public class Easing2 extends Sprite {
  private var ball:Ball;
  private var easing:Number=0.2;
  private var targetX:Number=stage.stageWidth / 2;
  private var targetY:Number=stage.stageHeight / 2;
  public function Easing2() {
   init();
  }
  private function init():void {
   ball=new Ball  ;
   addChild(ball);
   ball.addEventListener(MouseEvent.MOUSE_DOWN,onMouseDown);
   addEventListener(Event.ENTER_FRAME,onEnterFrame);
  }
  private function onMouseDown(event:MouseEvent):void {
   ball.startDrag();
   removeEventListener(Event.ENTER_FRAME,onEnterFrame);
   stage.addEventListener(MouseEvent.MOUSE_UP,onMouseUp);
  }
  private function onMouseUp(event:MouseEvent):void {
   ball.stopDrag();
   addEventListener(Event.ENTER_FRAME,onEnterFrame);
   stage.removeEventListener(MouseEvent.MOUSE_UP,onMouseUp);
  }
  private function onEnterFrame(event:Event):void {
   var vx:Number=(targetX - ball.x) * easing;
   var vy:Number=(targetY - ball.y) * easing;
   ball.x+= vx;
   ball.y+= vy;
  }
 }
}
```

缓动何时停止

在物体缓动运动到目标点时，物体最终会到达目标点并且完成缓动效果。但是，即使不显示该对象，缓动代码仍在执行中，这一来浪费了 CPU 资源。当物体到达目标时，应该停止执行代码。判断物体是否到达目标的方法非常简单，就像这样：

```
if(ball.x == targetX && ball.y == targetY) {
    // code to stop the easing
}
```

但是这里要注意一些技巧。

我们所讨论的缓动类型涉及到了著名的 Xeno 悖论。Xeno 也是位希腊人，爱好测量实验。Xeno 将运动分解为下面几个步骤：物体要从 A 点到达 B 点，它首先要移动到两点间一半的距离。然后物体再从该点出发，到达与 B 点距离一半的距离。然后再折半。每次只移动离目标一半的距离，但永远无法准确地达到目标。

这个悖论听起来是非常符合逻辑的。但是很明显，我们确实将物体从一点移动到了另一点，这样看来他的说法有些问题。到 Flash 中看看，影片在 x 轴上的位置为 0，假设要将它移动到 x 轴为 100 的位置。按照悖论所说，设缓动系数为 0.5，这样每次运动到离目标一半的距离。过程如下：

- 从 0 点开始，经过 1 帧，到达 50。
- 第 2 帧，到达 75。
- 剩下的距离是 25。它的一半是 12.5 ，所以新的距离就是 87.5。
- 按照这种顺序，位置将变化为 93.75, 96.875, 98.4375 等等。20 帧以后，将到达 99.999809265。

从理论上讲位置越来越接近目标，但是永远无法准确地到达目标点。然而，在代码中进行试验时，结果就发生了一些微妙的变化。归根结底问题就在于“一次最少能移动多少个像素”，答案是 1/20。事实上，二十分之一像素有个特有的名字：twip (缇)。在 Flash 内部计算单位都采用 twip 像素，包括所有 Sprite 影片，影片剪辑和其它舞台对象。因此，在显示影片位置时，这个数值永远是 0.05 的倍数。

下面举个例子，一个影片要到达 100 的位置，而它所到达的最接近的位置事实上是 99.5。再分隔的距离，就是加上 (100 - 99.95)/2。相当于加上了 0.025，四十分之一像素。超出了 twip 是能够移动的最小值，因此无法加上“半个 twip”，结果是只增加了 0 像素。如果大家不信的话，可以亲自试一下（提示：将代码放入框架类中的 init 方法）：

```
var sprite:Sprite;
sprite = new Sprite();
addChild(sprite);
sprite.x = 0;
var targ:Number = 100;
for(var i:Number = 0; i < 20; i++) {
    trace(i + ": " + sprite.x);
    sprite.x += (targ - sprite.x) * .5;
}
```

循环 20 次，将影片移动离目标一半的距离，这是基本缓动应用。将代码放入 for 循环，只是为了测试其位置，并不在于观察物体运动。循环到第 11 次时，影片已经到达了 99.95，这已经是它能够到达的最远的地方了。

长话短说，影片并非无限地接近目标，但是它确实永远无法准确地到达目标点。这样一来，缓动代码就永远不会停止。我们要回答的问题是“哪里才是物体最接近的目标位置？”，这需要确定到目标点的距离是否小于某个范围。我发现在很多软件中，如果物体与目标点的距离相差在一个像素以内，就可以说它已经到达了目标点，即可停止缓动了。

在处理二维坐标时，可以使用第三章所介绍的公式来计算点间距离：

```
distance = Math.sqrt(dx * dx + dy * dy)
```

如果只处理一维坐标点，如只移动一个轴的位置，就需要使用距离的绝对值，因为它有可能是个负数，使用 Math.abs 方法。

OK，说得很多了，来写代码吧。这个简单的文档类，演示了如何关闭缓动运动（EasingOff.as）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class EasingOff extends Sprite {
  private var ball:Ball;
  private var easing:Number = 0.2;
  private var targetX:Number = stage.stageWidth / 2;
  public function EasingOff() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   ball.y = stage.stageHeight / 2;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var dx:Number = targetX - ball.x;
   if (Math.abs(dx) < 1) {
    ball.x = targetX;
    removeEventListener(Event.ENTER_FRAME, onEnterFrame);
    trace("done");
   } else {
    var vx:Number = dx * easing;
    ball.x += vx;
   }
  }
 }
}
```

此例中，将缓动公式分解使用，首先计算出距离，因为我们需要知道是否该停止缓动。大家应该知道为什么要使用 dx 的绝对值了吧。如果小球在目标点的右边，dx 的值总是负的，if (dx < 1) 的结果永远为真，这就会使缓动停止。而使用 Math.abs，就可以判断实际距离是否小于 1。

记住，如果将拖拽与缓动相结合，要在放开小球时，将运动代码重新启用。

移动的目标

上面例子中的目标点都是单一且固定的，这些似乎还不能满足我们的要求。事实上，Flash 并不关心物体是否到达目标，或目标是否还在移动。它只会问 “我的目标在哪里？距离有多远？速度是多少？”，每帧都如此。因此，我们可以很容易将目标点改为鼠标所在的位置，只需将原来 targetX 和 targetY 的地方，改成鼠标的坐标 (mouseX 和 mouseY)。以下是一个比较简单的版本（EaseToMouse.as）：

```
package {
```

```
 import flash.display.Sprite;
 import flash.events.Event;
 public class EaseToMouse extends Sprite {
  private var ball:Ball;
  private var easing:Number = 0.2;
  public function EaseToMouse() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var vx:Number = (mouseX - ball.x) * easing;
   var vy:Number = (mouseY - ball.y) * easing;
   ball.x += vx;
   ball.y += vy;
  }
 }
}
```

移动鼠标观察小球跟随情况，是不是距离越远速度越快。

试想还有没有其它可移动的目标。当然还可以是一个影片向着另一个影片缓动。在早先的 Flash 时代，鼠标追踪者（mouse trailers）——即一串影片跟踪者鼠标的效果——曾经风靡一时。缓动就是制作这种效果的方法之一。第一个影片缓动到鼠标上，第二个影片缓动到第一个影片上，第三个再缓动到第二个上，依此类推。大家不妨一试。

缓动不仅限于运动

本书中，有很多简单的例子程序。在这些例子中，我们主要是计算影片所用变量的值。通常，使用 x,y 属性控制物体的位置。不过别忘了 Sprite 影片，影片剪辑以及各种显示对象还有很多其它可以操作的属性，而且基本都是用数字表示的。所以在读例子程序时，也应该试用其它属性代替这个例子中的属性。下面给大家一些启发。

透明度

将缓动用在 alpha 属性上。开始设置为 0 ,目标设置为 1 ：

```
ball.alpha = 0;
var targetAlpha:Number = 1;
```

在 enterFrame 函数中，使用缓动可以实现影片淡入效果：

```
ball.alpha += (targetAlpha - ball.alpha) * easing;
```

若将 0 和 1 颠倒过来就可以实现影片的淡出效果。

旋转

设置一个 rotation 属性和一个目标 rotation。当然，还需要一个能够表现旋转对象，比如一个箭头(arrow)：

```
arrow.rotation = 90;
var targetRotation:Number = 270;
arrow.rotation += (targetRotation - arrow.rotation) * easing;
```

颜色

如果大家想挑战一下，可以在 24 位色彩中使用缓动。设置好 red,green,blue 的初始值，使用缓动单独表现每一色彩元素的值，然后将它们再组合成 24 位色彩值。例如，我们可以从 red 缓动到 blue。初始颜色如下：

```
red = 255;
green = 0;
blue = 0;
redTarget = 0;
greenTarget = 0;
blueTarget = 255;
```

在 enterFrame 处理函数中的每一帧执行缓动。这里只表现一个 red 值：

```
red += (redTarget - red) * easing;
```

再将这三个数值组合为一个数（如第四章介绍的）：

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

最后可以在 ColorTransform （见第四章），线条颜色或填充色中使用该颜色值。

高级缓动

现在我们已经看到简单的缓动效果是如何实现的了，大家也许正在考虑如何使用更复杂的缓动公式制作一些效果。例如，在开始时比较缓慢，然后渐渐开始加速，最后在接近目标时再将速度慢下来。或者希望在一段时间或若干帧内完成缓动效果。

Robert Penner 以收集、编制和实现缓动公式而出名。我们可以在 www.robertpenner.com 中找到他的缓动公式。在他写这些内容时 AS 3 版本还没有出现，但是用我们前面几章所学知识，将它们转化为 AS 3 版本的也是件非常容易的事。

OK，下面进入 Flash 中我最喜欢的一个主题：弹性运动（Springing）。

弹性运动

一直以来，我都认为弹性运动将是 ActionScript 动画中最强大和最有用的物理学概念。几乎所有的物体都可以使用弹性运动。下面就来看看什么是弹性运动以及在 Flash 编程中的应用。

如同本章开始时所提到的，弹性的加速度与物体到目标点的距离成正比例。想象一下现实中弹簧的性质。把一个小球拴在橡皮圈一头，再将另一头系在一个固定的地方。当小球悬在半空时，在没有施加外力的情况下，小球就处在目标点的位置上。再将小球微微拉动，松手后橡皮圈会对其施加一些外力，又将小球拽回了目标点。如果用最大力量将小球挪远，那么橡皮圈就会对小球施加很大的力。小球急速向目标点飞去，并朝着另一面运动。这时，小

球的速度非常快。当它越过了目标点时，橡皮圈开始把它向回拉——改变其速度向量。这时，小球会继续运动，但是它运动得越远，所受到的拉力就越大。然后，速度向量为零，方向相反，一切再重新开始。最终，在反弹了几次后，速度逐渐慢下来直到停止——停在目标点上。

下面我们将这个过程翻译成 ActionScript。为了简化这个过程，我们先在一维坐标上进行实验。

一维坐标上的弹性运动

这里我们仍然使用可以拖拽的小球作为主体。默认位置还是 x 轴的 0 点，使它具有运动到中心点的弹性。像使用缓动一样，需要一个变量保存弹性的数值。可以认为这个数同距离的比例，较大的弹性值会使弹性运动显得十分僵硬。较小的弹性值会使弹性运动像一条松紧带。我们选用 0.1 作为弹性 (spring) ：

```
private var spring:Number = 0.1;
private var targetX:Number = stage.stageWidth / 2;
private var vx:Number = 0;
```

不要担心物体当前的位置，只需要知道如何确定这些变量与表达式就可以了。
然后加入运动代码并且找出与目标点的距离：

```
var dx:Number = targetX - ball.x;
```

下面计算加速度。加速度与距离成正比，也就是距离乘以 spring 的值：

```
var ax:Number = dx * spring;
```

得到了加速度以后，我们就回到了熟悉的地方，把加速度加到速度向量中。

```
vx += ax;
ball.x += vx;
```

在写代码前，先来模拟一下运行时的数据。假设物体的 x 坐标为 0， vx 为 0，目标 x 为 100，spring 值为 0.1。执行过程如下：

1. 距离(100) 乘以 spring，得到 10。将它加入 vx 中，此时 vx 变为 10。再将 vx 加入到速度向量上使得 x 位置变为 10。
2. 下一次，距离(100-10) 等于 90。加速度为 90 乘以 0.1，结果为 9。将结果加入 vx ，使 vx 变为 19。 x 坐标变为 29。
3. 下一次，距离为 71，加速度为 7.1，将结果加到 vx 中，使 vx 变成 26.1。 x 坐标变为 55.1。
4. 下一次，距离为 44.9，加速度为 4.49， vx 变成 30.59。 x 坐标变为 85.69。

注意，每次的加速度随着物体越接近目标，变得越来越小，但是速度向量仍在不断增涨。虽然涨幅不像启初那样快，但是速度却越来越快。

几次过后，物体就超过了目标点，到达了 117 的位置。现在的距离是 100 - 117，等于 -17。将这一部分加入到速度向量中，会使物体的移动稍稍慢下来。

现在大家知道弹性是如何工作了吧，让我们来实践一下。与往常一样，要保证 Ball 类是有效的，使用下面这个文档类（Spring.as）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Spring1 extends Sprite {
  private var ball:Ball;
  private var spring:Number = 0.1;
  private var targetX:Number = stage.stageWidth / 2;
```

```
  private var vx:Number = 0;
  public function Spring1() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   ball.y = stage.stageHeight / 2;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var dx:Number = targetX - ball.x;
   var ax:Number = dx * spring;
   vx += ax;
   ball.x += vx;
  }
 }
}
```

试验一下，可以看出一个类似于弹簧的效果，唯一的问题是它永远不会停止。前面在描述弹性运动时说过“速度逐渐慢下来直到停止”。由于小球每次摇摆时的距离相同，所以速度向量也相同，这样它就会以同样的速度来回摆动。这时，我们需要引入摩擦力。非常简单——只需要一个 friction 变量，值为 0.95。放在类开始的地方作为成员变量：

```
private var friction:Number = 0.95;
```

在 enterFrame 函数中，每次将 vx 乘以 friction，就可以了。

文档类 Spring2.as 的 onEnterFrame 方法如下：

```
private function onEnterFrame(event:Event):void {
    var dx:Number = targetX - ball.x;
    var ax:Number = dx * spring;
    vx += ax;
    vx *= friction;
    ball.x += vx;
}
```

这样一来，程序就完整了，虽说只是个一维的弹性运动。试改变 friction 的值观察运动效果。认真理解这个程序会对大家非常有帮助。掌握这个程序后，就来看看二维的弹性运动吧。

二维弹性运动

好了，不用麻烦了，直接看程序（Spring3.as）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Spring3 extends Sprite {
  private var ball:Ball;
  private var spring:Number = 0.1;
  private var targetX:Number = stage.stageWidth / 2;
```

```
  private var targetY:Number = stage.stageHeight / 2;
  private var vx:Number = 0;
  private var vy:Number = 0;
  private var friction:Number = 0.95;
  public function Spring3() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var dx:Number = targetX - ball.x;
   var dy:Number = targetY - ball.y;
   var ax:Number = dx * spring;
   var ay:Number = dy * spring;
   vx += ax;
   vy += ay;
   vx *= friction;
   vy *= friction;
   ball.x += vx;
   ball.y += vy;
  }
 }
}
```

我们看到，唯一不同之处就是加入了 y 轴的运动。但问题是，这个效果看上去还是很像一维的弹性运动。是的，虽然小球是沿着 x,y 轴运动，但它仍是直线运动。这是因为加速度的起点均为零，并且两种力的大小相同，因此是直线运动到目标点。

为了让效果更好，将初始 vx 的值设置得大一些，比如 50。现在，看起来更加形象、真实了。但这只是个开始，更厉害的还在后面呢。

向移动目标运动

实现弹性运动并不需要目标点是固定。前面在介绍缓动运动时，我们就介绍了一个非常简便快捷的方法——小球跟随鼠标。要让小球跟随鼠标位置非常简单，只要将原来的 targetX 和 targetY 变成鼠标位置即可。在弹性运动中也是如此，每一帧都要计算物体与目标点的距离，然后再确定加速度。

效果非常 Cool，我认为值得大家多写几遍。事实上，代码的变化并不大，只需将前面的程序做如下修改：

```
var dx:Number = targetX - ball.x;
var dy:Number = targetY - ball.y;
```

改为：

```
var dx:Number = mouseX - ball.x;
var dy:Number = mouseY - ball.y;
```

设置完成后，就可以将 targetX 和 targetY 这两个变量声明删除。如果不删，也不会有什么问题。

弹簧在哪？

下面我们将看到小球在橡皮圈上运动的情况。但是似乎还需要一个可见的橡皮圈，只需要用绘图 API 绘制一条线即可！绘图指令可以直接写在 Sprite 的子类中。在很多复杂的程序中，我们可能需要单独创建一个空影片当作绘图层来用。

方法很简单。在确定了小球的位置后，调用 clear() 将之前的线条擦除。然后重新设置 lineStyle 并绘制出小球与鼠标的连线。我们只需要在 enterFrame 函数中执行如下操作：

```
graphics.clear();
graphics.lineStyle(1);
graphics.moveTo(ball.x, ball.y);
graphics.lineTo(mouseX, mouseY);
```

非常有趣！我们还能做什么？再加上重力怎么样？这样小球看上去更像是悬在空中。很简单，只需要在每帧的 vy 中加入重力(gravity)。以下代码结合了重力与画线指令（Spring5.as）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Spring5 extends Sprite {
  private var ball:Ball;
  private var spring:Number = 0.1;
  private var vx:Number = 0;
  private var vy:Number = 0;
  private var friction:Number = 0.95;
  private var gravity:Number = 5;
  public function Spring5() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var dx:Number = mouseX - ball.x;
   var dy:Number = mouseY - ball.y;
   var ax:Number = dx * spring;
   var ay:Number = dy * spring;
   vx += ax;
   vy += ay;
   vy += gravity;
   vx *= friction;
   vy *= friction;
   ball.x += vx;
   ball.y += vy;
   graphics.clear();
   graphics.lineStyle(1);
```

```
    graphics.moveTo(ball.x, ball.y);
    graphics.lineTo(mouseX, mouseY);
  }
 }
}
```

执行后的结果如图 8-2 所示。请注意，我们加入的重力 (gravity) 只有 5，只是为了让小球下坠。如果重力太小的话，就看不到效果了。

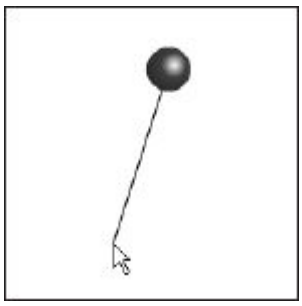


图 8-2 可见的鼠标弹簧

我们刚刚又违背了真实的物理学。当然，不能在物体上施加“累积重力”！重力是一个常数，取决于物体所在星球的大小和质量。我们所能做的就是累加物体的质量，以便在物体上产生多个重力。因此，严格来讲，重力还应该是 0.5，而质量应该在 10 左右。接下来，用质量乘以重力等于 5。或者可将 gravity 变量名改为 forceThatGravityIsExertingOnThisObjectBasedOnItsMass。好了，大家明白这个意思后我就可以节省一下空间使用变量名 gravity。

同样，试验一下这个程序。试减少 gravity 与 spring 的值，试改变 friction 的值。我们可以有无限多种数字的组合，可以建立出任何类型的重力系统。

弹簧链

下面我们将几个弹性小球串联起来。在介绍缓动一节时，我们简单地讨论了鼠标跟随的概念，意思是说一个物体跟随鼠标，另一个物体再跟随这个物体，依此类推。当时没有给大家举例子，是因为这个效果现在看来有些逊色。但是，当我们在弹性运动中使用这个概念时，效果就截然不同了。

本程序的设计思想：创建三个小球，名为 ball0, ball1, ball2。第一个小球，ball0 的动作与上面例子中的效果是相同的。 ball1 向 ball0 运动，ball2 向 ball1 运动。每个小球都受到重力的影响，所以它们都会向下坠。代码稍有些复杂，文档类 Chain.as：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Chain extends Sprite {
  private var ball0:Ball;
  private var ball1:Ball;
  private var ball2:Ball;
  private var spring:Number = 0.1;
  private var friction:Number = 0.8;
  private var gravity:Number = 5;
  public function Chain() {
   init();
  }
  private function init():void {
   ball0 = new Ball(20);
```

```
            balls.push(ball);
        }
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
```

最后，onEnterFrame 方法的变化最大。首先设置线条，将绘图起点移动到鼠标位置，再到第一个小球，然后循环为剩下的小球设置位置并连线。通过改变 numBalls 变量，我们可以加入任意多个小球。

```
    private function onEnterFrame(event:Event):void {
        graphics.clear();
        graphics.lineStyle(1);
        graphics.moveTo(mouseX, mouseY);
        moveBall(balls[0], mouseX, mouseY);
        graphics.lineTo(balls[0].x, balls[0].y);
        for(var i:uint = 1; i < numBalls; i++) {
            var ballA:Ball = balls[i-1];
            var ballB:Ball = balls[i];
            moveBall(ballB, ballA.x, ballA.y);
            graphics.lineTo(ballB.x, ballB.y);
        }
    }
```

运行结果见图 8-3，大家可以在 ChainArray.as 找到这个类。

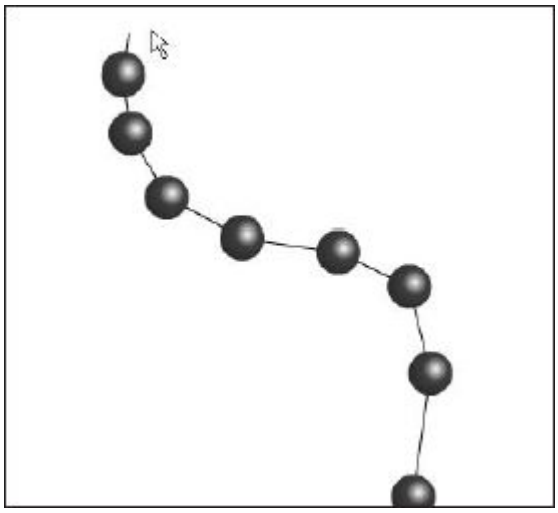


图 8-3 弹簧链

多目标点弹性运动

我们在第五章讨论速度与加速度时，曾说过如何使一个物体受到多种外力。如果每种力都是加速度，我们只需要把它们一个个都加到速度向量中去。因为弹力不过就是施加在物体上的一种加速度，因此在一个物体上添加多种弹力也是非常容易的。

下面是创建多目标弹簧的方法：我们需要三个控制点，这些点都是 Ball 类的实例，并且具有简单的拖拽功能，用它们作为小球弹性运动的控制点。小球会立即运动到点，并在两点间寻找平衡。换句话讲，每个目标都会对小球施加一定的外力，小球的运动速度就是这些外力相加的结果。

例子程序相当复杂，使用多个方法处理不同的事件。以下是代码（文档类 MultiSpring.as），看过后再进行分段讲解：

```
    addChild(ball0);
    ball1 = new Ball(20);
    addChild(ball1);
    ball2 = new Ball(20);
    addChild(ball2);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
    moveBall(ball0, mouseX, mouseY);
    moveBall(ball1, ball0.x, ball0.y);
    moveBall(ball2, ball1.x, ball1.y);
    graphics.clear();
    graphics.lineStyle(1);
    graphics.moveTo(mouseX, mouseY);
    graphics.lineTo(ball0.x, ball0.y);
    graphics.lineTo(ball1.x, ball1.y);
    graphics.lineTo(ball2.x, ball2.y);
  }
  private function moveBall(ball:Ball,targetX:Number,targetY:Number):void {
    ball.vx += (targetX - ball.x) * spring;
    ball.vy += (targetY - ball.y) * spring;
    ball.vy += gravity;
    ball.vx *= friction;
    ball.vy *= friction;
    ball.x += ball.vx;
    ball.y += ball.vy;
  }
 }
}
```

看一下 Ball 这个类，我们发现每个对象实例都有自己 vx 和 vy 属性，并且它们的初始值均为 0。所以在 init 方法中，我们只需要创建小球并把它们加入显示列表。

然后在 onEnterFrame 函数中，实现弹性运动。这里我们调用了 moveBall 方法，比复制三次运动代码要好用得多。该函数的参数分别为一个 ball 对象以及目标点的 x,y 坐标。每个小球都调用这个函数，第一个小球以鼠标的 x,y 作为目标位置，第二第三个小球以第一第二个小球作为目标位置。

最后，在确定了所有小球的位置后，开始画线，画线的起点是鼠标位置，然后依次画到每个小球上，这样橡皮圈就连接上了所有的小球。注意，程序中的 friction 降为 0.8 为了使小球能够很快稳定下来。

创建一个数组保存链中所有对象的引用，然后通过循环遍历数组中的每个小球并执行运动，使这个程序更加灵活。这里只需要做一些小小的改变。首先，需要两个新的变量代表数组和对象数目：

```
private var balls:Array;
private var numBalls:Number = 5;
```

在函数 init 中，使用 for 循环创建所有对象，并将对象引用加入数组：

```
private function init():void {
balls = new Array();
    for(var i:uint = 0; i < numBalls; i++) {
        var ball:Ball = new Ball(20);
        addChild(ball);
```

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.events.MouseEvent;
 public class MultiSpring extends Sprite {
  private var ball:Ball;
  private var handles:Array;
  private var spring:Number = 0.1;
  private var friction:Number = 0.8;
  private var numHandles:Number = 3;
  public function MultiSpring() {
   init();
  }
  private function init():void {
   ball = new Ball(20);
   addChild(ball);
   handles = new Array();
   for (var i:uint = 0; i < numHandles; i++) {
    var handle:Ball = new Ball(10, 0x0000ff);
    handle.x = Math.random() * stage.stageWidth;
    handle.y = Math.random() * stage.stageHeight;
    handle.addEventListener(MouseEvent.MOUSE_DOWN, onPress);
    addChild(handle);
    handles.push(handle);
   }
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
   addEventListener(MouseEvent.MOUSE_UP, onRelease);
  }
  private function onEnterFrame(event:Event):void {
   for (var i:uint = 0; i < numHandles; i++) {
    var handle:Ball = handles[i] as Ball;
    var dx:Number = handle.x - ball.x;
    var dy:Number = handle.y - ball.y;
    ball.vx += dx * spring;
    ball.vy += dy * spring;
   }
   ball.vx *= friction;
   ball.vy *= friction;
   ball.x += ball.vx;
   ball.y += ball.vy;
   graphics.clear();
   graphics.lineStyle(1);
   for (i = 0; i < numHandles; i++) {
    graphics.moveTo(ball.x, ball.y);
    graphics.lineTo(handles[i].x, handles[i].y);
   }
  }
  private function onPress(event:MouseEvent):void {
   event.target.startDrag();
```

```
  }
  private function onRelease(event:MouseEvent):void {
   stopDrag();
  }
 }
}
```

在 init 方法中，创建小球并用 for 循环创建三个控制点，随机安排位置，并为它们设置拖拽行为。

onEnterFrame 方法循环取出每个控制点，使小球向该点方向运动。然后，用控制点的坐标设置小球的速度，反复循环，从小球开始向各个控制点画线。onPress 方法的内容非常简单，但是请注意 onRelease 函数，我们无法知道当前拖拽的是哪个小球。幸运的是，使用任何一个显示调用 stopDrag 方法，都可以停止所有的拖拽，所以只需要在文档类中直接调用该方法。

我们只要改变 numHandles 变量的值，就可以轻松地设置控制点的数量。运行结果如图 8-4 所示。

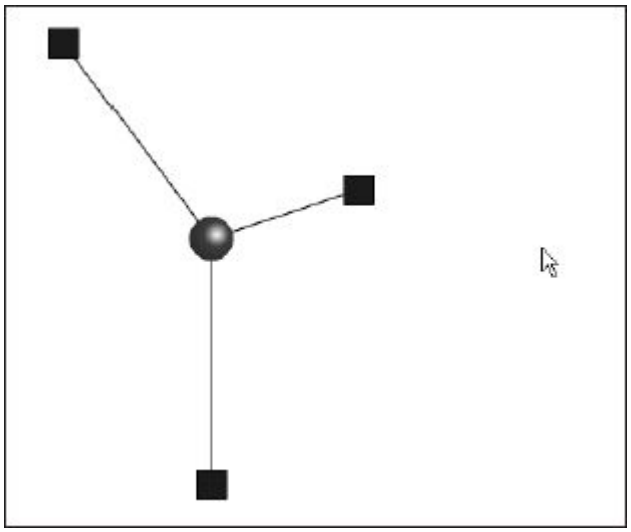


图 8-4 多目标弹性

到目前为止，我相信大家已经有了很多的心得与体会，并且开始尝试解决一些书中没有提到的问题。如果真是这样的话，那就太好了！这也正是我写这本书的目的。

目标偏移

我们拿到一个真正的弹簧——有弹性的金属圈——然后将它的一头固定起来，另一头放上小球或其它物体，那么物体运动的目标点是哪里？难道说目标点是固定弹簧的那头儿？不，这并不实际。小球永远也到不了这个点，因为它会受到弹簧自身的阻碍。一旦弹簧变回了正常的长度，它会对小球施加更大的力。因此，目标点就应该是弹簧展开后的末端。

要寻找目标点，首先要找到物体与固定点之间的夹角，然后沿这个角度从固定点向外展开一段长度——弹簧的长度。换句话讲，如果弹簧长度是 50，小球与固定点的夹角是 45 度的话，那么就要以 45 度的夹角向外运动 50 个像素，而这个点就是小球的目标点。图 8-5 解释了这一过程。

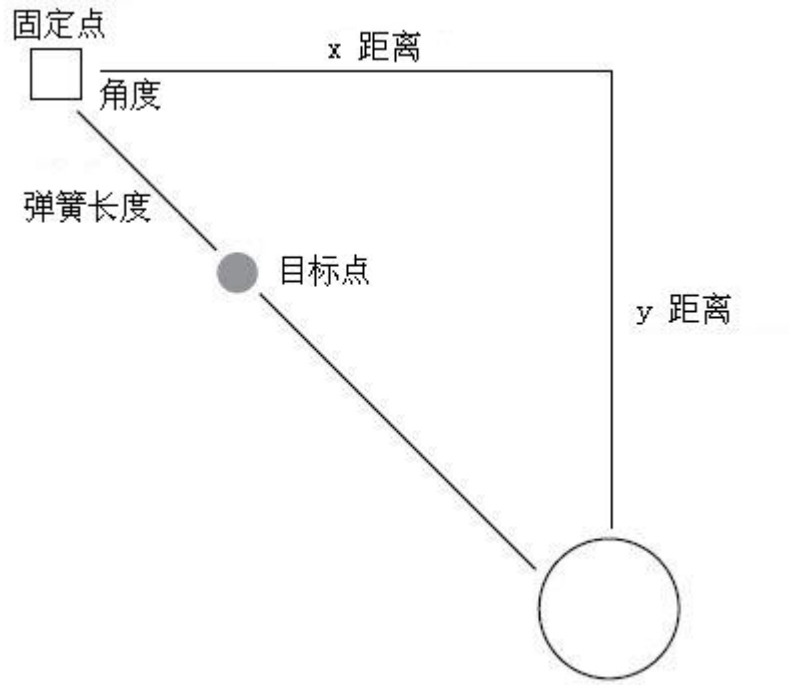


图 8-5 弹簧复位

寻找目标点的代码如下：

```
var dx:Number = ball.x - fixedX;
var dy:Number = ball.y - fixedY;
var angle:Number = Math.atan2(dy, dx);
var targetX:Number = fixedX + Math.cos(angle) * springLength;
var targetY:Number = fixedY + Math.sin(angle) * springLength;
```

运行结果是，物体向着固定点运动，但会在与目标点相差一段距离时停止移动。大家还要注意，虽然我们叫它“固定点”，只是代表弹簧固定到的某个点。而不是指这个点不能移动。也许最好的方法就是看代码。

我们继续使用鼠标位置作为固定点，弹簧的长度为 100 像素。以下是文档类（OffsetSpring.as）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class OffsetSpring extends Sprite {
  private var ball:Ball;
  private var spring:Number = 0.1;
  private var vx:Number = 0;
  private var vy:Number = 0;
  private var friction:Number = 0.95;
  private var springLength:Number = 100;
  public function OffsetSpring() {
   init();
  }
  private function init():void {
   ball = new Ball(20);
   addChild(ball);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var dx:Number = ball.x - mouseX;
```

```
        var dy:Number = ball.y - mouseY;
        var angle:Number = Math.atan2(dy, dx);
        var targetX:Number = mouseX + Math.cos(angle) * springLength;
        var targetY:Number = mouseY + Math.sin(angle) * springLength;
        vx += (targetX - ball.x) * spring;
        vy += (targetY - ball.y) * spring;
        vx *= friction;
        vy *= friction;
        ball.x += vx;
        ball.y += vy;
        graphics.clear();
        graphics.lineStyle(1);
        graphics.moveTo(ball.x, ball.y);
        graphics.lineTo(mouseX, mouseY);
      }
    }
}
```

虽然我们能够看到运行结果，但却不能真正发现这项技术的特殊用处。没关系，下一节会给大家一个特别的例子。

弹簧连接多个物体

我们知道如何用弹簧连接两个物体，还知道这个点不是固定的。但是，如果另一个物体上还有一个弹簧反作用在第一个物体上，又是怎样的呢？这里有两个物体之间由一根弹簧连接。其中一个运动了，另一个物体就要向该物体移动过来。

我开始认为制作这种效果会导致死循环从而无法实现，或者至少会引起错误。但我也没管那么多，勇敢地进行尝试。结果非常完美！

虽然前面已经描述了一些策略，但这里还要细致得说一下：物体 A 以物体 B 作为目标，并向它移动。物体 B 反过来又以物体 A 作为目标。事实上，本例中目标偏移起了重要的作用。如果一个物体以其它物体直接作为目标，那么它们之就会相互吸引，最终聚集在一个点上。通过使用偏移目标，我们就可以使它们之间保持距离，如图 8-6 所示。

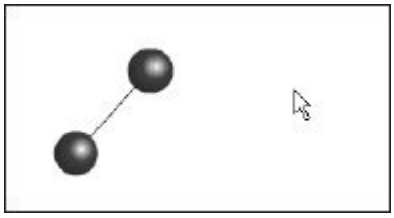


图 8-6 弹簧连接的两个物体

下面举一个例子，我们需要两个 Ball 类的实例。分别为 ball0 和 ball1。ball0 向 ball1 偏移运动。ball1 向 ball0 偏移运动。为了不去反复写偏移弹性运动的代码，我们将这些功能写到函数 springTo 中，直接调用函数即可。如果想让 ball0 向 ball1 运动，只要写 springTo(ball0, ball1)，然后再让 ball1 向 ball0 运动，就写 springTo(ball1, ball0)。还要设置两个变量，ball0Dragging 和 ball1Dragging，作为每个小球运动的开关。以下是文档类（DoubleSpring.as）：

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

```
public class DoubleSpring extends Sprite {
 private var ball0:Ball;
 private var ball1:Ball;
 private var ball0Dragging:Boolean = false;
 private var ball1Dragging:Boolean = false;
 private var spring:Number = 0.1;
 private var friction:Number = 0.95;
 private var springLength:Number = 100;
 public function DoubleSpring() {
  init();
 }
 private function init():void {
  ball0 = new Ball(20);
  ball0.x = Math.random() * stage.stageWidth;
  ball0.y = Math.random() * stage.stageHeight;
  ball0.addEventListener(MouseEvent.MOUSE_DOWN, onPress);
  addChild(ball0);
  ball1 = new Ball(20);
  ball1.x = Math.random() * stage.stageWidth;
  ball1.y = Math.random() * stage.stageHeight;
  ball1.addEventListener(MouseEvent.MOUSE_DOWN, onPress);
  addChild(ball1);
  addEventListener(Event.ENTER_FRAME, onEnterFrame);
  stage.addEventListener(MouseEvent.MOUSE_UP, onRelease);
 }
 private function onEnterFrame(event:Event):void {
  if (!ball0Dragging) {
   springTo(ball0, ball1);
  }
  if (!ball1Dragging) {
   springTo(ball1, ball0);
  }
  graphics.clear();
  graphics.lineStyle(1);
  graphics.moveTo(ball0.x, ball0.y);
  graphics.lineTo(ball1.x, ball1.y);
 }
 private function springTo(ballA:Ball, ballB:Ball):void {
  var dx:Number = ballB.x - ballA.x;
  var dy:Number = ballB.y - ballA.y;
  var angle:Number = Math.atan2(dy, dx);
  var targetX:Number = ballB.x - Math.cos(angle) * springLength;
  var targetY:Number = ballB.y - Math.sin(angle) * springLength;
  ballA.vx += (targetX - ballA.x) * spring;
  ballA.vy += (targetY - ballA.y) * spring;
  ballA.vx *= friction;
  ballA.vy *= friction;
  ballA.x += ballA.vx;
  ballA.y += ballA.vy;
```

```
  }
  private function onPress(event:MouseEvent):void {
   event.target.startDrag();
   if (event.target == ball0) {
    ball0Dragging = true;
   }
   if (event.target == ball1) {
    ball1Dragging = true;
   }
  }
  private function onRelease(event:MouseEvent):void {
   ball0.stopDrag();
   ball1.stopDrag();
   ball0Dragging = false;
   ball1Dragging = false;
  }
 }
}
```

本例中，每个小球都是可以拖拽的。enterFrame 函数负责为小球调用 springTo 函数。请注意，这两条语句都是由两条判断语句包围起来的，目的是要确认小球目前没被拖拽：

```
springTo(ball0, ball1);
springTo(ball1, ball0);
```

springTo 函数用于产生运动，函数中的所有语句大家应该都很熟悉。首先，求出距离和角度，再计算目标点，然后向目标点运动。第二次调用函数时，参数相反，两个小球交换位置，开始的小球向另一个小球运动。这也许不是效率最高的代码，但是它可以最好地表现出运动的过程。

我们看到，小球不会依附在任何固定点上，它们都是自由飘浮的。小球之间唯一的约束就是彼此保持一定的距离。这种写法最好的地方是可以很容易地加入新的物体。例如，再创建第三个小球（ball2），同时为它设置一个变量（ball2Dragging），就可以这么添加：

```
if(!ball0Dragging) {
    springTo(ball0, ball1);
    springTo(ball0, ball2);
}
if(!ball1Dragging) {
    springTo(ball1, ball0);
    springTo(ball1, ball2);
}
if(!ball2Dragging) {
    springTo(ball2, ball0);
    springTo(ball2, ball1);
}
```

这样就建立了一个三角形结构，如图 8-7 所示。大家熟练掌握后，很快就能做出四边形结构，直到一切复杂的弹簧结构。

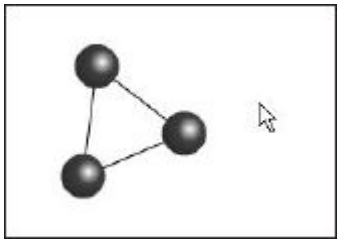


图 8-7 一根弹簧连接三个物体

本章重要公式总结

现在来回顾一下本章的重要公式

简单缓动，长形：

```
var dx:Number = targetX - sprite.x;
var dy:Number = targetY - sprite.y;
vx = dx * easing;
vy = dy * easing;
sprite.x += vx;
sprite.y += vy;
```

简单缓动，中形：

```
vx = (targetX - sprite.x) * easing;
vy = (targetY - sprite.y) * easing;
sprite.x += vx;
sprite.y += vy;
```

简单缓动，短形：

```
sprite.x += (targetX - sprite.x) * easing;
sprite.y += (targetY - sprite.y) * easing;
```

简单弹性，长形：

```
var ax:Number = (targetX - sprite.x) * spring;
var ay:Number = (targetY - sprite.y) * spring;
vx += ax;
vy += ay;
vx *= friction;
vy *= friction;
sprite.x += vx;
sprite.y += vy;
```

简单弹性，中形：

```
vx += (targetX - sprite.x) * spring;
vy += (targetY - sprite.y) * spring;
vx *= friction;
vy *= friction;
sprite.x += vx;
sprite.y += vy;
```

简单弹性，短形：

```
vx += (targetX - sprite.x) * spring;
vy += (targetY - sprite.y) * spring;
sprite.x += (vx *= friction);
sprite.y += (vy *= friction);
```

偏移弹性运动：

```
var dx:Number = sprite.x - fixedX;
var dy:Number = sprite.y - fixedY;
var angle:Number = Math.atan2(dy, dx);
var targetX:Number = fixedX + Math.cos(angle) * springLength;
var targetY:Number = fixedX + Math.sin(angle) * springLength;
```

8.5 小结

这一章涉及两个基本的比例运动技术：缓动和弹性。你已经知道了缓动是比例运动，而弹性是比例速度，你应当已经很好地理解了如何应用这两项技术。

我希望你能理解我为什么偏爱弹性，可以亲自使用他们创建一些真正有趣的效果。

现在你已经学习了各种移动物体的方法，让我继续进入下一张，那里你将会看到当他们相互碰撞时如何处理。

第九章　碰撞检测

到目前为止，我们已经学习了物体在其空间的内交互运动。接下来研究一下物体之间的交互运动。这就需要确定物体间何时发生了碰撞，这就是我们所讲的碰撞检测（Collision detection 或 Hit testing）。

本章我会尽量将所有需要掌握的相关知识告诉大家。其中包括两个影片的碰撞，影片与点之间的碰撞，用距离检测碰撞以及多物体碰撞检测方法。首先，来看一下有什么现成的碰撞检测方法。

碰撞检测方法

碰撞检测的思想非常非常简单。我们只要知道两个物体是否有在同一时间内某个部分处在了同一位置上。当然，也许物体不只两个，这就需要知道其中的一个是否和其它的物体发是了碰撞。

如本节的题目，检测碰撞的方法有很多：可以是对物体实际像素的检测（Sprite 或 MovieClip），也就是判断两个影片中的图形是否重叠？对于这种检测方法，要以影片内图形的实际可见像素判断，还是以影片的矩形边界来判断呢？这就涉及到 hitTest 方法中内置的两个可选项，用来满足不同方面的需求。

碰撞也可以根据距离来判断。获得两个物体间的距离，然后问“物体间是否近得足够发生碰撞了？”，应用这种方法时需要计算并判断距离。

每种方法都有它们各自的用途。具体实现会在本章进行讲解。至于发生碰撞后该做些什么，本章并不涉及。因为，这些内容我们会在第十一章介绍动量守恒时详细地为大家讲解。

hitTestObject 与 hitTestPoint

早先 Flash 中的影片剪辑都有内置 hitTest 方法，这个方法有很多种用途。而现在已经将它划分为两种方法，这样做更加合理。hitTestObject 方法用于判断两个显示对象间是否发生了碰撞，而 hitTestPoint 方法用于判断某个点与显示对象间是否发生了碰撞。

碰撞检测两个影片

使用 hitTestObject 判断两个影片是否碰撞也许是最简单的碰撞检测方法。调用这个函数作为影片的方法，将另一个影片的引用作为参数传入。注意，虽然我说的是影片，但这两种方法都是 DisplayObject 类的成员，对于所有继承自显示对象类的子类，如 MovieClip, Bitmap, Video, TextField 等都可以使用。格式如下：

```
sprite1.hitTestObject(sprite2)
```

通常于在 if 语句中使用：

```
if(sprite1.hitTestObject(sprite2)) {
    // 碰撞后的动作
}
```

如果两个影片发生了碰撞则返回 true，并执行 if 语句块的内容。一切事物都是把双刃剑。碰撞检测方法越简单，其精确度就越低；相反，检测的精确度越高，则实现起来就越复杂。因此，这个最简单的方法，精确度也是最差的。

那么在碰撞检测中精确度意味着什么呢？不就是判断两个物体有没有冲突吗？我也希

望问题只有这么简单。

回过头来看问题：知道两个影片的位置，如何判断它们是否碰撞？最简单的判断方法是这样执行的：拿来一个物体，绕着它画一个矩形。再复制出一个相同的影片，两个之间进行碰撞检测。最后判断两个矩形之间是否有相交的地方。如果有，则发生碰撞。用矩形包围物体就是我们所熟知的**矩形边界（bounding box）**。当我们在 Flash IDE 中点击一个舞台元件时，就会看到一圈蓝色的轮廓线，如图 9-1 所示。

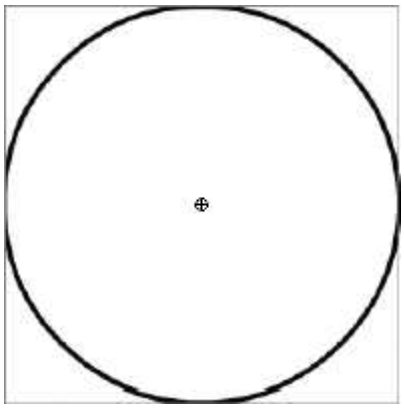


图 9-1 矩形边界

当然，在 Flash 播放器中，并非先画上一个矩形再进行判断。一切都是根据影片的位置和大小计算出来的。

为什么这种方法不精确？因为，一旦两个矩形边界相交，则必然会产生碰撞。下面请见图 9-2，这几对图形中，哪两个相交了？

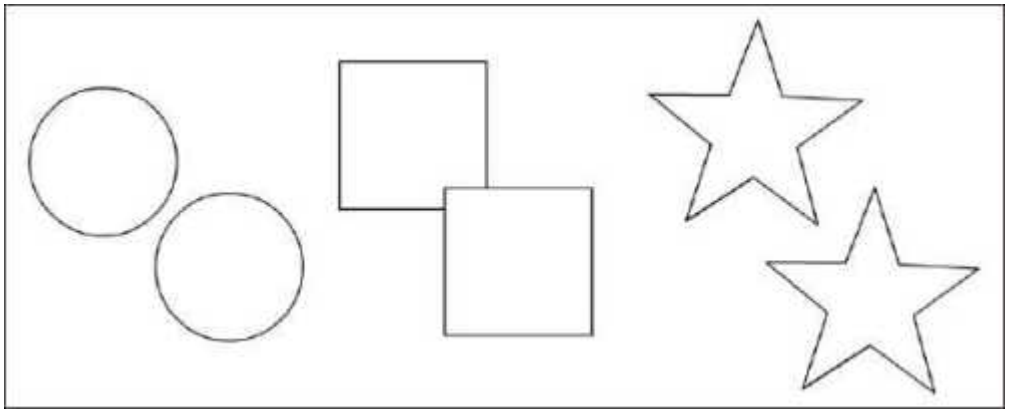


图 9-2 哪一对相交了？

很明显，只有那两个正方形碰到了，对吧？好的，下面为它们画上矩形边界，再从 Flash 的视角观察一下。结果如图 9-3 所示。

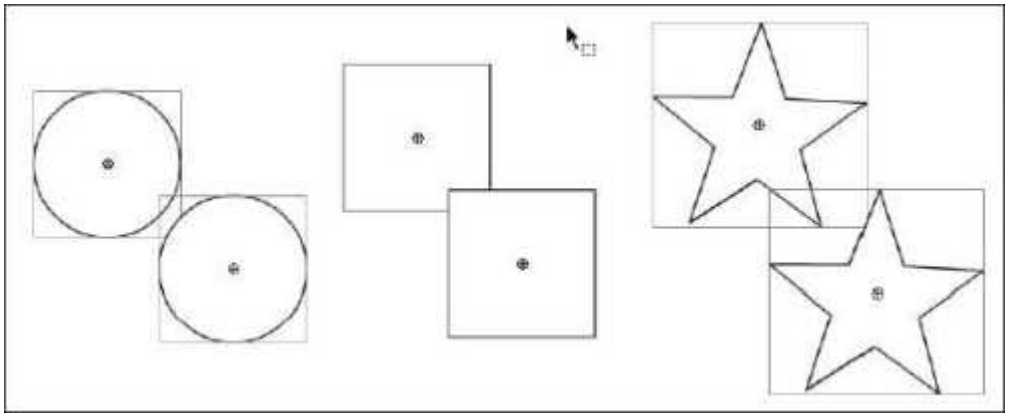


图 9-3 并非我们希望的结果

对于 Flash 来说，每对图形是相交的。大家不相信的话，请看下面这个文档类 ObjectHitTest.as，用到了我们前面创建的 Ball 类，请确认这个类存在于类路径中：

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

```
 import flash.events.Event;
 public class ObjectHitTest extends Sprite {
  private var ball1:Ball;
  private var ball2:Ball;
  public function ObjectHitTest() {
   init();
  }
  private function init():void {
   ball1 = new Ball();
   addChild(ball1);
   ball1.x = stage.stageWidth / 2;
   ball1.y = stage.stageHeight / 2;
   ball2 = new Ball();
   addChild(ball2);
   ball2.startDrag(true);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   if (ball1.hitTestObject(ball2)) {
    trace("hit");
   }
  }
 }
}
```

本例创建了两个 Ball 类的实例，并将其中一个设置为可拖拽的。在每帧中使用 hitTestObject 方法判断两个影片是否发生了碰撞。注意，当我们从上，下，左，右靠近目标时，结果都是正确的。但是如果倾斜着靠近物体，就有问题了。当然，如果物体都是矩形的话，不会有什么问题。但物体要是越不规则，那么结果就越不准确。所以，当发生碰撞的物体不是矩形时，大家就要格外小心。

下面举一个使用 hitTestObject 判断矩形物体的例子。这里要用到一个崭新的 Box 类，与 Ball 类的非常相似，相信大家理解起来一定没有问题，代码如下：

```
package {
 import flash.display.Sprite;
 public class Box extends Sprite {
  private var w:Number;
  private var h:Number;
  private var color:uint;
  public var vx:Number = 0;
  public var vy:Number = 0;
  public function Box(width:Number=50,
  height:Number=50,
  color:uint=0xff0000) {
   w = width;
   h = height;
   this.color = color;
   init();
  }
  public function init():void {
   graphics.beginFill(color);
```

```
    graphics.drawRect(-w / 2, -h / 2, w, h);
    graphics.endFill();
   }
  }
}
```

设计思想是， box 从屏幕上方落到下。 box 落到舞台底部或其它 box 上时，就算放置好了。下面是代码（文档类 Boxes.as）：

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

 public class Boxes extends Sprite {
  private var box:Box;
  private var boxes:Array;
  private var gravity:Number = 0.2;

  public function Boxes() {
   init();
  }
  private function init():void {
   boxes = new Array();
   createBox();
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   box.vy += gravity;
   box.y += box.vy;
   if (box.y + box.height / 2 > stage.stageHeight) {
    box.y = stage.stageHeight - box.height / 2;
    createBox();
   }
   for (var i:uint = 0; i < boxes.length; i++) {
    if (box != boxes[i] && box.hitTestObject(boxes[i])) {
     box.y = boxes[i].y - boxes[i].height / 2 - box.height / 2;
     createBox();
    }
   }
  }
  private function createBox():void {
   box = new Box(Math.random() * 40 + 10, Math.random() * 40 + 10);
   box.x = Math.random() * stage.stageWidth;
   addChild(box);
   boxes.push(box);
  }
 }
}
```

在 onEnterFrame 方法中，首先判断 box 位置是否低于舞台底部。如果是则停止，然后创建下一个 box。在 for 循环中，判断当前 box 是否碰撞到了其它 box。首先要判断，它不是和自己碰撞的，然后是整个程序的重点 hitTestObject，用来判断当前 box 是否与

其它 box 发生碰撞。如果是则将当前 box 放在那个 box 的上面，随后创建一个新的 box。如果我们将本例中，Box 类换成 Ball 类，就会出现物体悬浮在空中的情景。

影片与点的碰撞检测

hitTestPoint 的工作方法有些不同，还带有一个可选参数。这个方法在判断两个影片的碰撞时并不常用。该方法中有两个 Number 类型的参数，用于定义点。根据影片是否与某点相交，返回 true 或 false。最基本的形式如下（100,100 代表点的 x,y 坐标）：

```
sprite.hitTestPoint(100, 100);
```

同样，可以在 if 语句中使用它来判断碰撞为 true 时要执行的代码。

回到问题上：怎样才算碰撞？这时又看到我们的老朋友矩形边界了。Flash 只检查我们给出的点是否在影片的矩形边界内。用文档类 PointHitTest.as 来看测试一下：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class PointHitTest extends Sprite {
  private var ball:Ball;
  public function PointHitTest() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   ball.x = stage.stageWidth / 2;
   ball.y = stage.stageHeight / 2;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   if (ball.hitTestPoint(mouseX, mouseY)) {
    trace("hit");
   }
  }
 }
}
```

这次使用鼠标的 x,y 坐标判断与小球碰撞的点。当鼠标接近小球，可能还没有真正碰触到目标时就产生了碰撞。同样情况下，如果使用矩形方块，就不会有问题了。因此，这个方法看起来也只对矩形物体有用。但是这里还有一个叫 shapeFlag 的选项。

使用 shapeFlag 执行碰撞检测

shapeFlag 是 hitTestPoint 方法的第三个参数是可选的。其值是 Boolean 类型的，因此只有 true 和 false。将 shapeFlag 设置为 true 意味着碰撞检测时判断影片中可见的图形，而不是矩形边界。注意，shapeFlag 只在能用在检测点与影片的碰撞中。如果是两个影片的碰撞就不能用这个参数了。

来实验一下，只需要在原有基础上增加一个值，将 hitTestPoint 变为：

```
if(ball.hitTestPoint(mouseX, mouseY, true))
```

（如果不想使用 shapeFlag，可将参数设为 false，与不写参数是完全等价的）用小球测试一下这个版本的碰撞。我们发现这回鼠标只有在真正碰撞到影片的图形时才会检测到碰撞。

这样我们就实现了非常精确的碰撞检测，但是它并不能应对所有的情况。问题在于使用的点只有一个，很难看出到底影片的哪些部分碰触到了其它影片。也许大家的第一反是用：

```
sprite1.hitTestPoint(sprite2.x, sprite2.y, true)
```

但是，如果这样的话，我们只能判断出 sprite2 的注册是否在影片 sprite1 上。这种用途实在有限。因为 sprite2 的任何部分都有可能碰到 sprite1。实际应用中，最好选择鼠标或很小的影片作为碰撞点对象，因为它们两个之间也许只有一两个像素的偏差。

人们曾试图用这种方法检测物体四周上的点。例如，有一个星形的影片，我们可以计算出星形五个顶点的位置，然后判断每个顶点与另一个影片是否发生碰撞。但是如果有两个星形影片，我们就需要用这个星形的五个顶点与另一个星形的五个顶点进行碰撞判断。如果只是星形还好说，但要是其它不规则图形，还需要更多的顶点。可以想象这将占用大量的 CPU 资源。使用简单的碰撞检测方法，光是两个星形就要判断多达十次。这就是我们要追求准确度所负出的代价。

hitTest 总结

那么在两个不规则的物体间如何检测碰撞？很遗憾，用 hitTest 方法无法实现。

下面总结一下，hitTest 的基本设置：

- 对于矩形影片，使用 hitTestObject(displayObject)。
- 对于非常小的影片，使用 hitTestPoint(x, y, true)（注意将 shapeFlag 设置为 true）。
- 对于非常不规则的影片图形，如果不要求非常精确或自定义一些解决方法的话，那么也可以使用 hitTestPoint(x, y, true)。

本章内容远没有结束，下面我们还有超越内置方法的碰撞检测法。如果对象是圆形的物体，那么使用距离碰撞检测方法将是最好的选择。我们会发现原来有很多种图形都属于圆形。

距离碰撞检测

本节开始，我们就摆脱了内置 hitTest 方法，而是将碰撞检测掌握在自己手里。这就要用两个物体间的距离来判断碰撞的发生。

举个现实中的例子，如果你那辆车与我这辆车有 100 米的距离，我们就知道这两辆车离得足够远，不可能发生碰撞。然而，如果我们的车都有 6 米宽和 12 米长，而我这辆车的中心点与你那辆车的中心点只有 5 米，那么肯定会有些金属被撞弯，保险单会变长。换句话讲，除非车子的某些部分被撞掉以外，两辆车不可能并到一起。这就是整个距离碰撞检测的思想。我们要确认使两个物体分开的最小距离，再看看当前距离，比较两者的大小。如果当前距离小于最小距离，就知道物体间发生了碰撞。

hitTestObject 方法在矩形上使用效果最好，但在处理其它图形时就退化了，而我们这种方法则在处理圆形时效果最好。要是处理的图形与圆形有偏差，则精确度就会有所降低。但是这里会遇到与 hitTest 中矩形边界相反的问题：明明发生了碰撞，确没有响应，这是因为它们的中心点还不够近。

简单的距离碰撞检测

让我们从最理想的状态开始：两个圆形。依然可以使用 Ball 类。（大家现在也许明白了为什么“重用”一词通常与面向对象编程联系在一起了吧。）

在应用距离碰撞检测时，圆的注册点应该在中心点上，Ball 类正好符合要求。先要创建两个小球，并设置其中一个为可拖拽的。然后在 enterFrame 函数中进行碰撞检测。到这儿为止，程序与本章的第一个例子相同。只是在判断碰撞时，不是使用 if(ball1.hitTestObject(ball2))，而是在 if 语句中判断距离。我们已经学习了计算两个物体间距离的方法，回忆一下第三章的勾股定理。所以，程序开始应该是这样的：

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

OK，现在距离已经有了，如何进行判断呢？请看图 9-4。

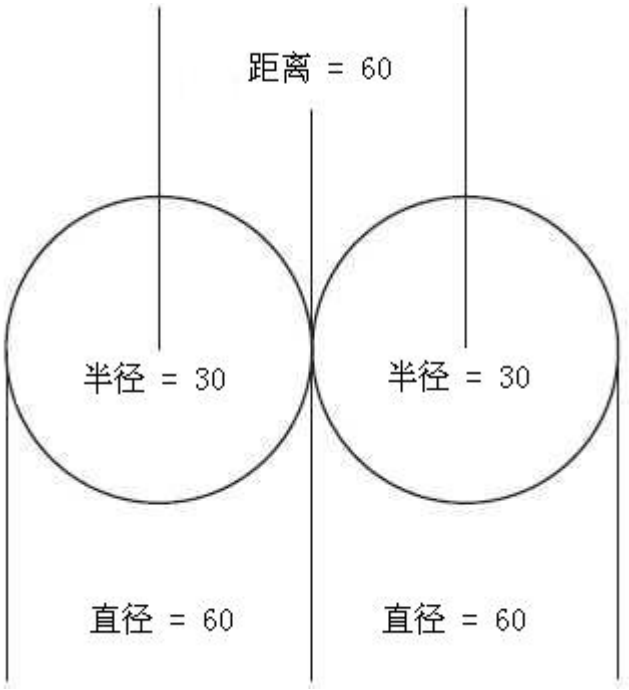


图 9-4 碰撞的距离

图中我们看到两个影片发生了碰撞，每个影片都占 60 像素宽，那么每个半径就是 30。因此，在它们相互碰撞时，实际相差 60 个像素。啊哈！这就是答案。对于两个大小相同的圆来说，如果距离小于直径，就会产生碰撞。本例代码（Distance.as）与 ObjectHisTest.as 非常相似，设置 onEnterFrame 方法为：

```
private function onEnterFrame(event:Event):void {
 var dx:Number = ball2.x - ball1.x;
 var dy:Number = ball2.y - ball1.y;
 var dist:Number = Math.sqrt(dx * dx + dy * dy);
 // 默认 ball 的直径为 80 (半径为 40)
 if (dist < 80) {
  trace("hit");
 }
}
```

测试后，我们发现这回碰撞的结果与接近小球的角度无关。在没有接触到目标球时不会产生碰撞。但是在代码中使用数值表示距离显然不太合适，因为这样的话每次改变 ball 的大小都要重新修改代码。况且，如果两个小球的大小不同怎么办？我们需要将这个方法抽象成可以适应任何情况的公式。

如图 9-5 所示。两个大小不同的 ball，相互碰撞。左边的小球 60 像素，右边的 40 像素。我们可以用程序检察它们的 width 属性。第一个 ball 的半径为 30，另一个半径为 20。所以，它们碰撞时的距离实际应为 50。在 Ball 类里面，已经设置了半径（radius）属性，可以直接拿来判断。

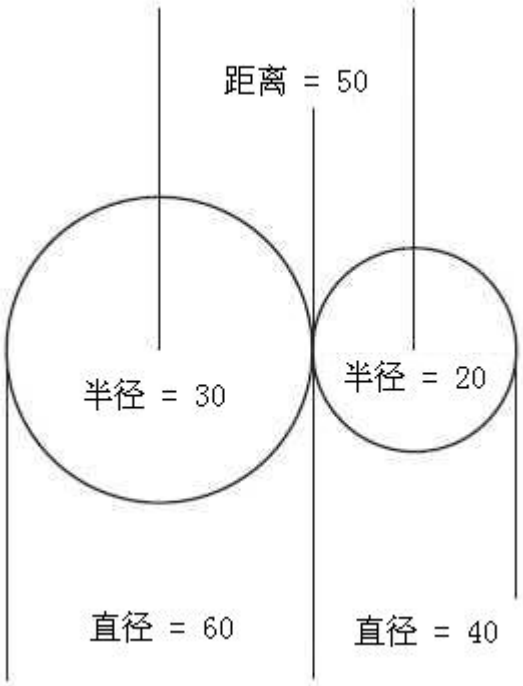


图 9-5 两个不同体积物体的碰撞距离

思路已经有了，距离就是两个小球的半径之和。现在可以删除手工加入的数字了，代码如下（文档类 Distance2.as）：

```
private function onEnterFrame(event:Event):void {
 var dx:Number = ball2.x - ball1.x;
 var dy:Number = ball2.y - ball1.y;
 var dist:Number = Math.sqrt(dx * dx + dy * dy);
 if (dist < ball1.radius + ball2.radius) {
  trace("hit");
 }
}
```

实验一下，设置小球的大小（将 radius 传入 Ball 的第一个参数中），观察执行结果。在前面例子中，我是这样写的：

```
ball1 = new Ball(Math.random() * 100);
```

测试一下，每次执行小球的半径都不相同，但碰撞的效果依然完美如初。

弹性碰撞

给大家一个完整的距离碰撞检测的例子，其中包括之前没有讨论的问题，如两个物体碰撞时的交互以及如何有效地处理多物体间的交互。但我也不想让例子中出现没有学过的内容。

我的想法是：放入一个大球，名为 centerBall，在舞台的中心。然后加入多个小球，给它们随机的大小与速度，让它们进行基本的运动并在撞墙后反弹。每一帧都在小球与大球之间进行距离碰撞检测。如果发生了碰撞，根据它们之间的角度计算出弹性运动的偏移目标和最小碰撞距离。OK，这么说不是很明白。根本的意思就是，如果小球与 centerBall 发生碰撞，就把小球弹出去，通过设置 centerBall 外面的目标点来实现，然后让小球向目标点弹性运动。一旦小球到达目标，就不再产生碰撞，弹性运动结束，继续执行常规的运动。

运行结果就像小气泡被大气泡反弹回去，如图 9-6 所示。

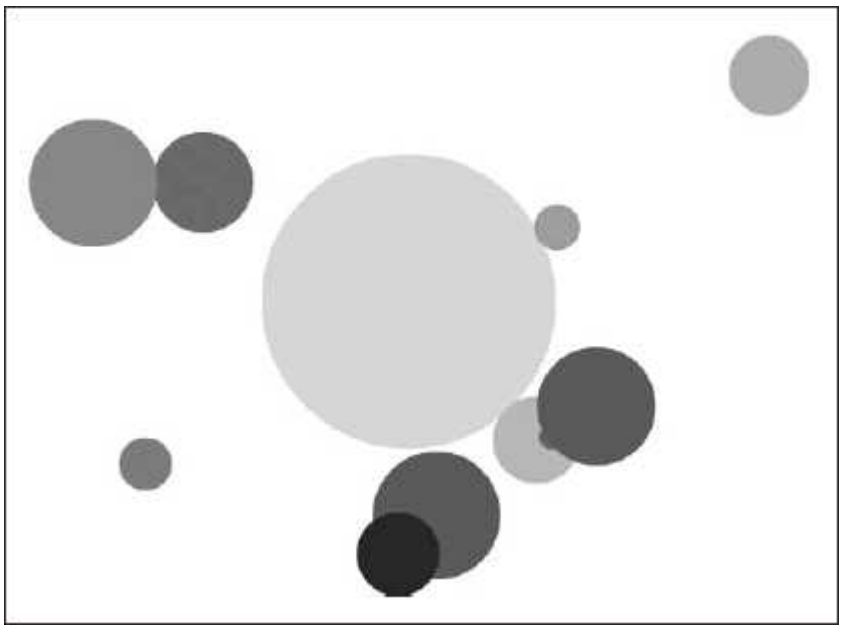


图 9-6 弹性碰撞

下面是代码（可在 Bubbles.as 中找到）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Bubbles extends Sprite {
  private var balls:Array;
  private var numBalls:Number = 10;
  private var centerBall:Ball;
  private var bounce:Number = -1;
  private var spring:Number = 0.2;
  public function Bubbles() {
   init();
  }
  private function init():void {
   balls = new Array();
   centerBall = new Ball(100, 0xcccccc);
   addChild(centerBall);
   centerBall.x = stage.stageWidth / 2;
   centerBall.y = stage.stageHeight / 2;
   for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball = new Ball(Math.random() * 40 + 5,
    Math.random() * 0xffffff);
    ball.x = Math.random() * stage.stageWidth;
    ball.y = Math.random() * stage.stageHeight;
    ball.vx = Math.random() * 6 - 3;
    ball.vy = Math.random() * 6 - 3;
    addChild(ball);
    balls.push(ball);
   }
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   for (var i:uint = 0; i < numBalls; i++) {
```

```
    var ball:Ball = balls[i];
    move(ball);
    var dx:Number = ball.x - centerBall.x;
    var dy:Number = ball.y - centerBall.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    var minDist:Number = ball.radius + centerBall.radius;
    if (dist < minDist) {
     var angle:Number = Math.atan2(dy, dx);
     var tx:Number = centerBall.x +
     Math.cos(angle) * minDist;
     var ty:Number = centerBall.y +
     Math.sin(angle) * minDist;
     ball.vx += (tx - ball.x) * spring;
     ball.vy += (ty - ball.y) * spring;
    }
   }
  }
  private function move(ball:Ball):void {
   ball.x += ball.vx;
   ball.y += ball.vy;
   if (ball.x + ball.radius > stage.stageWidth) {
    ball.x = stage.stageWidth - ball.radius;
    ball.vx *= bounce;
   } else if (ball.x - ball.radius < 0) {
    ball.x = ball.radius;
    ball.vx *= bounce;
   }
   if (ball.y + ball.radius > stage.stageHeight) {
    ball.y = stage.stageHeight - ball.radius;
    ball.vy *= bounce;
   } else if (ball.y - ball.radius < 0) {
    ball.y = ball.radius;
    ball.vy *= bounce;
   }
  }
 }
}
```

是的，整个代码非常多，但是多数内容前面章节中都介绍过。让我们快速浏览一下。

从 init 函数开始，创建一个 centerBall ，然后循环创建小球，并让小球运动，并给它们随机的大小，位置，颜色和速度。

enterFrame 函数循环得到每个小球的引用。为了分散功能函数，我们将运动代码放到另一个函数中，然后进行调用，参数是小球的引用。这个函数的内容也是我们再熟悉不过的了，只是基本的运动和反弹。下面，求出小球与 centerBall 的距离，然后求出碰撞检测的最小距离。如果发生了碰撞，就要计算出小球与 centerBall 的夹角，再加上最小距离计算出目标点的 x,y。这个目标点就是 centerBall 的圆周。

最后，使用基本的弹性效果，让小球向该点运动（如第八章介绍的）。一旦小球到达目标点，就不会再发生碰撞，然后向任意方向运动。

思考一下，我们是怎样用简单的技术制作出非常复杂的运动效果的？

多物体碰撞检测方法

在屏幕上有只有两个物体时，判断它们之间的碰撞是非常简单的。但是如果物体很多时，我们就需要了解一些碰撞检测的策略，以便不遗漏任何可能发生的碰撞。当要检测的物体越来越多时，如果进行有效的判断就显得至关重要。

基本的多物体碰撞检测

当只有两个物体时，碰撞只可能在 A - B 物体间发生。如果有三个物体，就有三种可能：A - B, B - C, C - A。如果是四个物体就有六种可能，五个物体就有十种可能。

如果物体多达 20 个，就需要分别进行判断 190 次。这就意味着在我们的 enterFrame 函数中，需要调用 190 次 hitTest 方法或距离计算。

如果使用这种方法，那么就会多用出必要判断的两倍！比如说 20 个物体就执行了 380 次 if 语句（20 个影片每个判断 19 次，20 * 19 = 380）。大家现在知道学习本节内容的重要性了吧。

看一下问题，审视一下平常的做法。假设我们有六个影片，分别为 sprite0, sprite1, sprite2, sprite3, sprite4, sprite5。让它们运动并执行反弹，我们想要知道它们之间何时会发生碰撞。思考一下，依次获得每个影片的引用，然后再执行循环，再去和其它的影片进行比较。下面是一段伪代码：

```
numSprites = 6;
for (i = 0; i < numSprites; i++) {
 spriteA = sprites[i];
 for (j = 0; j < numSprites; j++) {
  spriteB = sprites[j];
  if (spriteA.hitTestObject(spriteB)) {
   // 执行代码
  }
 }
}
```

六个影片执行了 36 次判断。看上去很合理，对吗？其实，这段代码存在着两大问题。首先，来看第一次循环，变量 i 和 j 都等于 0。因此 spriteA 所持的引用是 sprite0，而 spriteB 的引用也是一样。嗨，我们原来是在判断这个影片是否和自己发生碰撞！无语了。所以要在 hitTest 之前确认一下 spriteA != sprieB，或者可以简单地写成 i != j。代码就应该是这样：

```
numSprites = 6;
for (i = 0; i < numSprites; i++) {
 spriteA = sprites[i];
 for (j = 0; j < numSprites; j++) {
  spriteB = sprites[j];
  if (i != j && spriteA.hitTestObject(spriteB)) {
   // do whatever
  }
 }
}
```

OK，现在已经排除了六次判断，判断次数降到了 30 次，但还是太多。下面列出每次比

较的过程：

```
sprite0 与 sprite1, sprite2, sprite3, sprite4, sprite5 进行比较
sprite1 与 sprite0, sprite2, sprite3, sprite4, sprite5 进行比较
sprite2 与 sprite0, sprite1, sprite3, sprite4, sprite5 进行比较
sprite3 与 sprite0, sprite1, sprite2, sprite4, sprite5 进行比较
sprite4 与 sprite0, sprite1, sprite2, sprite3, sprite5 进行比较
sprite5 与 sprite0, sprite1, sprite2, sprite3, sprite4 进行比较
```

请看第一次判断:用 sprite0 与 sprite1 进行比较。再看第二行:sprite1 与 sprite0 进行比较。它俩是一回事，对吧？如果 sprite0 没有与 sprite1 碰撞，那么 sprite1 也肯定不会与 sprite0 碰撞。或者说，如果一个物体与另一个碰撞，那么另一个也肯定与这个物体发生碰撞。所以可以排除两次重复判断。如果删掉重复判断，列表内容应该是这样的：

```
sprite0 与 sprite1, sprite2, sprite3, sprite4, sprite5 进行比较
sprite1 与 sprite2, sprite3, sprite4, sprite5 进行比较
sprite2 与 sprite3, sprite4, sprite5 进行比较
sprite3 与 sprite4, sprite5 进行比较
sprite4 与 sprite5 进行比较
sprite5 没有可比较的对象!
```

我们看到第一轮判断，用 sprite0 与每个影片进行比较。随后，再没有其它影片与 sprite0 进行比较。把 sprite0 放下不管，再用 sprite1 与剩下的影片进行比较。当执行到最后一个影片 sprite5 时，所有的影片都已经和它进行过比较了，因此 sprite5 不需要再与任何影片进行比较了。结果，比较次数降到了 15 次，现在大家明白我为什么说初始方案通常执行了实际需要的两倍了吧。

那么接下来如果写代码呢？仍然需要双重嵌套循环，代码如下：

```
numSprites = 6;
for (i = 0; i < numSprites - 1; i++) {
 spriteA = sprites[i];
 for (j = i + 1; j < numSprites; j++) {
  spriteB = sprites[j];
  if (spriteA.hitTestObject(spriteB)) {
   // do whatever
  }
 }
}
```

请注意，外层循环执次数比影片总数少一次。就像我们在最后的列表中看到的，不需要让最后一个影片与其它影片比较，因为它已经被所有影片比较过了。

内层循环的索引以外循环索引加一作为起始。这是因为上一层的内容已经比较过了，而且不需要和相同的索引进行比较。这样一来执行的效率达到了 100%。

多物体弹性运动

我们再来看一个小程序。同样也是气泡效果的交互运动，不过这次所有的气泡彼此间都可以相互反弹。效果如图 9-7 所示。

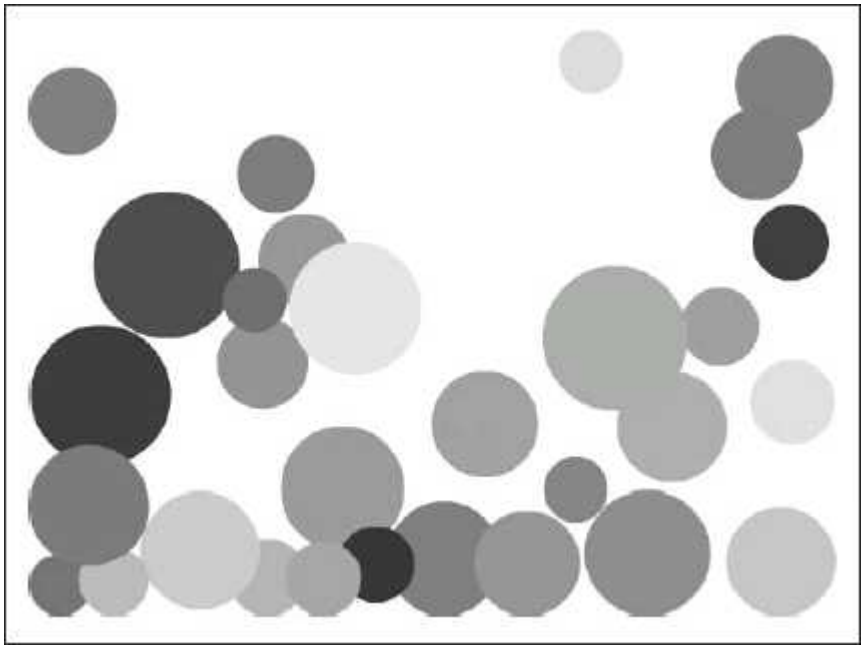


图 9-7 多物体碰撞

以下代码见文档类 Bubbles2.as：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Bubbles2 extends Sprite {
  private var balls:Array;
  private var numBalls:Number = 30;
  private var bounce:Number = -0.5;
  private var spring:Number = 0.05;
  private var gravity:Number = 0.1;
  public function Bubbles2() {
   init();
  }
  private function init():void {
   balls = new Array();
   for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball = new Ball(Math.random() * 30 + 20,
    Math.random() * 0xffffff);
    ball.x = Math.random() * stage.stageWidth;
    ball.y = Math.random() * stage.stageHeight;
    ball.vx = Math.random() * 6 - 3;
    ball.vy = Math.random() * 6 - 3;
    addChild(ball);
    balls.push(ball);
   }
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   for (var i:uint = 0; i < numBalls - 1; i++) {
    var ball0:Ball = balls[i];
    for (var j:uint = i + 1; j < numBalls; j++) {
     var ball1:Ball = balls[j];
```

```
      var dx:Number = ball1.x - ball0.x;
      var dy:Number = ball1.y - ball0.y;
      var dist:Number = Math.sqrt(dx * dx + dy * dy);
      var minDist:Number = ball0.radius + ball1.radius;
      if (dist < minDist) {
       var angle:Number = Math.atan2(dy, dx);
       var tx:Number = ball0.x + Math.cos(angle) * minDist;
       var ty:Number = ball0.y + Math.sin(angle) * minDist;
       var ax:Number = (tx - ball1.x) * spring;
       var ay:Number = (ty - ball1.y) * spring;
       ball0.vx -= ax;
       ball0.vy -= ay;
       ball1.vx += ax;
       ball1.vy += ay;
      }
     }
    }
    for (i = 0; i < numBalls; i++) {
     var ball:Ball = balls[i];
     move(ball);
    }
   }
   private function move(ball:Ball):void {
    ball.vy += gravity;
    ball.x += ball.vx;
    ball.y += ball.vy;
    if (ball.x + ball.radius > stage.stageWidth) {
     ball.x = stage.stageWidth - ball.radius;
     ball.vx *= bounce;
    } else if (ball.x - ball.radius < 0) {
     ball.x = ball.radius;
     ball.vx *= bounce;
    }
    if (ball.y + ball.radius > stage.stageHeight) {
     ball.y = stage.stageHeight - ball.radius;
     ball.vy *= bounce;
    } else if (ball.y - ball.radius < 0) {
     ball.y = ball.radius;
     ball.vy *= bounce;
    }
   }
  }
}
```

本例中的交互动画还需要两点补充说明。这是碰撞后的运动代码：

```
if(dist < minDist) {
    var angle:Number = Math.atan2(dy, dx);
    var tx:Number = ball0.x + Math.cos(angle) * minDist;
    var ty:Number = ball0.y + Math.sin(angle) * minDist;
    var ax:Number = (tx - ball1.x) * spring;
```

```
    var ay:Number = (ty - ball1.y) * spring;
    ball0.vx -= ax;
    ball0.vy -= ay;
    ball1.vx += ax;
    ball1.vy += ay;
}
```

只要 ball0 与 ball1 发生了碰撞就会执行这段代码。基本上与前面例子中的 centerBall 意思相近，只不是用 ball0 代替 centerBall。求出需要它俩的夹角，然后计算目标点的 x,y 坐标，这就是 ball1 要到达的点，这样做才不会让两个小球碰撞到一起。接下来，计算出 x,y 轴的加速度 ax,ay。下面要注意了，本例中，不仅 ball1 要向 ball0 弹性运动，而且 ball0 还必需离开 ball1，加速度的方向相反。只需要将 ax 和 ay 加到 ball1 的速度向量中，再从 ball0 的速度向量中将它们减去即可！这样就免去了计算两次的麻烦。大家也许认为这样做，就相当于把最终的加速度扩大了两倍。是的，您说得没错。为了弥补这一点，我们将 spring 的值设置得比平时小一些。

下面说说另一个问题。代码中使用 Math.atan2 计算夹角，然后再用 Math.cos 和 Math.sin 求出目标点：

```
var angle:Number = Math.atan2(dy, dx);
var tx:Number = ball0.x + Math.cos(angle) * minDist;
var ty:Number = ball0.y + Math.sin(angle) * minDist;
```

但是大家不要忘记，正弦是对边与斜边之比，而余弦是邻边与斜边之比。请注意，该角的对边就是 dy，邻边就是 dx，而斜边就是 dist。所以，我们实际上可以将这三行代码缩短为两行：

```
var tx:Number = ball0.x + dx / dist * minDist;
var ty:Number = ball0.y + dy / dist * minDist;
```

瞧！只用了两个简单的除法就取代了调用三次三角函数。下面请实验一下这个泡泡球的例子，试调整 spring,gravity,number 和 ball 的大小，观察运行结果。大家还可以添加摩擦力或鼠标交互的动作。

碰撞检测的其它方法

ActionScript 内置的 hitTest 方法与距离碰撞检测方法并不是实现碰撞检测仅有的方法，但使用它们可以完成大部分的碰撞检测。

如果要进行更深入的研究，我们会发现聪明的开发者们已经提出了一些非常精巧的碰撞检测方法。比如，Grant Skinner 提出了一种通过操作多个位图对象，来确定什么时候，哪些物体之间的像素发生了重叠。大家可以在 www.gskinner.com 找到相关内容。

本章重要公式

下面我们来回顾一下本章的两个重要公式。

距离碰撞检测：

```
// 从 spriteA 和 spriteB 开始
// 如果使用一个空白影片，或影片没有半径（radius）属性
```

```
// 可以用宽度与高度除以 2。
 var dx:Number = spriteB.x - spriteA.x;
 var dy:Number = spriteB.y - spriteA.y;
 var dist:Number = Math.sqrt(dx * dx + dy * dy);
 if (dist < spriteA.radius + spriteB.radius) {
 // 处理碰撞
 }
```

多物体碰撞检测：

```
 var numObjects:uint = 10;
 for (var i:uint = 0; i < numObjects - 1; i++) {
 // 使用变量 i 提取引用
 var objectA = objects[i];
 for (var j:uint = i+1; j<numObjects; j++) {
 //使用变量 j 提取引用
 var objectB = objects[j];
 //在 objectA 与 objectB 之间进行碰撞检测
 }
}
```

9.7 小结

本章主要讲解了你需要知道的关于碰撞检测的所有内容，包括各种内置的 hitTest 函数、基于距离的碰撞检测和如何有效的跟踪多个物体的碰撞。你需要知道每种方法中的加与减和每种情况工作的好坏。在本书后面的内容中你需要用到这里的所有内容。毫无疑问，你将要在你的项目中继续扩展它。

在下一章中，你将看到物体与有角度平面之间的碰撞，使用的技术称为坐标旋转。这项技术它将在第 11 章中用到，在那里，你将学会如何对那些你知道如何检测的碰撞创建一个真实的碰撞反应。

第十章　坐标旋转及角度反弹

本章介绍了一项特殊技术，著名的坐标旋转。如同其名，它是物体指绕着某点旋转其坐标，在制作一些非常有趣的效果时，坐标旋转是必不可少的。其中就包括在 Flash 界讨论了很多年的问题：“如何在斜面上进行反弹？”，本章我会给大家一一解答。

另一个用坐标旋转完成的程序是两物体之间的交互反弹效果。我们会在下一章讨论动量守衡时进行讲解。而本章的坐标旋转，我们之前也已经接触过了。如果大家想跳过这章的话，我劝您还是先坐下来，浏览一遍为好。

简单的坐标旋转

虽然我们在第三章讲三角学的时候介绍过计算的坐标旋转的方法，但还是先来做一下回顾。假设知道一个中心点，一个物体，一个半径和一个角度。通过不断地增加或减少角度，并运用基本的三角学知识让物体绕着中心点旋转。我们可将变量设为 vr （旋转速度）来控制角度的增加或减少。还有，不要忘记角度应用弧度制来表示。代码的结构如下所示：

```
vr = 0.1;
angle = 0;
radius = 100;
centerX = 250;
centerY = 200;
// 在 enterFrame 处理函数中：
sprite.x = centerX + cos(angle) * radius;
sprite.y = centerY + sin(angle) * radius;
angle += vr;
```

根据角度与半径使用简单的三角函数设置物体的 x,y 属性，并在每帧中改变角度。我们用 Flash 动画演示一下。下面是第一个例子，文档类 Rotate1.as：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Rotate1 extends Sprite {
  private var ball:Ball;
  private var angle:Number = 0;
  private var radius:Number = 150;
  private var vr:Number = .05;
  public function Rotate1() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   ball.x = stage.stageWidth / 2 + Math.cos(angle) * radius;
   ball.y = stage.stageHeight / 2 + Math.sin(angle) * radius;
   angle += vr;
```

```
    }
  }
}
```

这段代码中没有什么新的知识点。大家可以改变一下角度与半径，试验运行结果。但是如果我们只知道物体与中心点的位置又该怎么办呢？用 x,y 坐标计算出当前的角度(angle)与半径(radius)也并非难事。代码如下：

```
var dx:Number = ball.x - centerX;
var dy:Number = ball.y - centerY;
var angle:Number = Math.atan2(dy, dx);
var radius:Number = Math.sqrt(dx * dx + dy * dy);
```

这种基于坐标的旋转只对单个物体的旋转效果比较好，尤其是一次性就可确定角度和半径的情况下。但是在动态的程序中，有时需要旋转多个物体，而它们与中心点的相对位置可能会发生改变。因此，对于每个物体来说，都需要计算距离，角度和半径，还要用 vr 来增加角度，最后才能算出新的 x,y 坐标，每帧都如此。这就显得太麻烦了，并且效率也不会很高。没关系，我们还有更好的办法。

高级坐标旋转

如果物体要绕着某一点旋转，并且以物体本身的位置作为旋转的起点，那么下面给大家一个公式。这个公式只需要给出物体距离中心点的相对 x,y 坐标和旋转的角度。它就会返回物体相对于中心点的新的坐标位置。基本公式如下：

```
x1 = cos(angle) * x - sin(angle) * y;
y1 = cos(angle) * y + sin(angle) * x;
```

公式看上去像是一串相互对称的数字或符号，这是我刚刚接触它时的感觉。虽然使用这个公式很多次了，但依然有这样的感觉。我曾多次用图形来解释这些正弦余弦函数是如何让 x,y 变化的，每一次都会有全新的感觉。研究 30 分钟后，我发现这是两串非常对称的符号。

虽然我通常都会给大家解释某项技术完整的概念，但是如果这次我还这样做的话，那我可能就是伪君子了。因为，就我个人而言，也只是将这个公式背下来，以便在做梦时能把它敲出来。如果您掌握的三角学的知道比我丰富的话，那当然最好，这样您也许会对这项技术有更深层的了解；不过，即使您不是制造火箭的科学家，那么只要记住这个公式，可以同样完成出色的效果。

让我们看看这个公式都说了些什么，如图 10-1 所示。x,y 当然就是旋转后的坐标了。更准确地说，这是物体绕中心点旋转后的坐标。因此，如果中心点位于 200,100，而物体位于 300,150，那么 x 就是 300 - 200 = 100，y 就是 150 - 100 = 50。

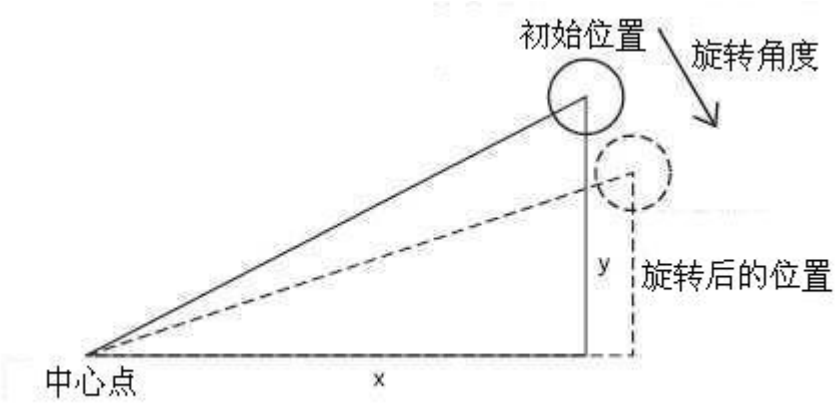


图 10-1 坐标旋转

角度(angle)就是某一时刻内旋转物体位置的大小。这并非当前的角度，也不是旋转后

的角度，而是这两者之间的差值。换句话讲，如果物体在离中心点 45 度的位置，而这次的角度是 5 度，我们就要在当前角度的基础上再旋转 5 度，到达 50 度这个位置上。这里我们并不关心最初或最终的角度，只关心旋转了多少度。通常来讲，这个角度都是用弧度制表示的。OK，让我们来看例子吧。

单物体旋转

本例中将一个小球放在随机的位置上，并使用前面介绍的方法对它进行旋转（文档类 Rotate2.as）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Rotate2 extends Sprite {
  private var ball:Ball;
  private var vr:Number = .05;
  private var cos:Number = Math.cos(vr);
  private var sin:Number = Math.sin(vr);
  public function Rotate2() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   ball.x = Math.random() * stage.stageWidth;
   ball.y = Math.random() * stage.stageHeight;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var x1:Number = ball.x - stage.stageWidth / 2;
   var y1:Number = ball.y - stage.stageHeight / 2;
   var x2:Number = cos * x1 - sin * y1;
   var y2:Number = cos * y1 + sin * x1;
   ball.x = stage.stageWidth / 2 + x2;
   ball.y = stage.stageHeight / 2 + y2;
  }
 }
}
```

在使用 vr 之前将它设成了 0.05，再计算这个角度的正弦和余弦值。根据小球与舞台中心点的位置计算出 x1 和 y1。然后使用前面讲到的坐标旋转公式，计算出小球的新位置 x2 和 y2。由于这个位置是小球与中心点的相对位置，所以我们还需要把 x2 和 y2 与中心点相加求出最终小球的位置。

实验一下，我们发现这个例子与早先那个版本执行的结果是一样的。也许大家会问，既然功能完全一样，为什么还要使用这个看起来很复杂的公式呢？如果处理的内容非常简单，也许您的说法是正确的。下面让我们来看看这个公式在简化问题时的应用。首先，考虑多物体旋转的情况。

多物体旋转

假设要旋转多个物体，所有的影片都保存在命为 sprites 的数组中。那么 for 循环应该是这样的：

```
for (var i:uint = 0; i < numSprites; i++) {
 var sprite:Sprite = sprites[i];
 var dx:Number = sprite.x - centerX;
 var dy:Number = sprite.y - centerY;
 var angle:Number = Math.atan2(dy, dx);
 var dist:Number = Math.sqrt(dx * dx + dy * dy);
 angle += vr;
 sprite.x = centerX + Math.cos(angle) * dist;
 sprite.y = centerY + Math.sin(angle) * dist;
}
```

然而如果使用高级坐标旋转的方法应该是这样的：

```
var cos:Number = Math.cos(vr);
var sin:Number = Math.sin(vr);
for (var i:uint = 0; i < numSprites; i++) {
 var sprite:Sprite = sprites[i];
 var x1:Number = sprite.x - centerX;
 var y1:Number = sprite.y - centerY;
 var x2:Number = cos * x1 - sin * y1;
 var y2:Number = cos * y1 + sin * x1;
 sprite.x = centerX + x2;
 sprite.y = centerY + y2;
}
```

请注意第一个版本中我们在 for 循环里调用了四次 Math 函数，这意味着每个物体的旋转都要执行四次函数调用。第二个版本中只执行了两次函数调用，而且都是在 for 循环以外面执行的，意味着它们只执行了一次，与物体的数量无关。举个例子，如果我们有 30 个影片，如果使用第一个版本的代码每帧需要调用 120 次 Math 函数。大家可以想想哪个版本的效率最高。

在前上一个例子程序中，删去 enterFrame 中的 sin 和 cos 函数。这是因为这段程序中的角度是固定的，因此可以直接给出结果。但是在很多情况下，旋转的角度不是固定不变的，这就需要每次进行重新计算正余弦的值。

解释一下最后这个概念。举个例子，用鼠标位置控制多个物体旋转的速度。如果鼠标在屏幕中心，则不产生旋转。如果鼠标向左移动，则物体逆时针旋转，并且越向左速度越快。如果向右移动，则顺时针旋转。除了创建多个 Ball 实例，并以数组存储以外，这个例子与前面那个非常相似。下面是文档类（Rotation3.as）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Rotate3 extends Sprite {
  private var balls:Array;
  private var numBalls:uint = 10;
  private var vr:Number = .05;
  public function Rotate3() {
   init();
```

```
  }
  private function init():void {
   balls = new Array();
   for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball = new Ball();
    balls.push(ball);
    addChild(ball);
    ball.x = Math.random() * stage.stageWidth;
    ball.y = Math.random() * stage.stageHeight;
   }
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var angle:Number = (mouseX - stage.stageWidth / 2) * .001;
   var cos:Number = Math.cos(angle);
   var sin:Number = Math.sin(angle);
   for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball = balls[i];
    var x1:Number = ball.x - stage.stageWidth / 2;
    var y1:Number = ball.y - stage.stageHeight / 2;
    var x2:Number = cos * x1 - sin * y1;
    var y2:Number = cos * y1 + sin * x1;
    ball.x = stage.stageWidth / 2 + x2;
    ball.y = stage.stageHeight / 2 + y2;
   }
  }
 }
}
```

我们可以看到，代码并不是那么复杂。如果大家已经掌握了这个方法，可以试用角度与弧度的方法各写一遍代码，看看执行的效果是好还是坏。

在第十五章讨论 3D 时，回来再看这个公式。事实上，我们将会在同一个方法中使用两次这个公式让物体绕着两个轴与三个维旋转。不要被我说的这些吓到了，因为在学到这里以前我们还要学习很多的内容。

角度反弹

我还记得在我刚刚开始痴迷于 Flash，数学和物体时，我就解决了物体撞墙地面、天花板后反弹的效果。如果这些障碍物只是水平和垂直的，我就知道该怎么去做。但是渐渐地仅知道这些已经不能满足需求了。在现实情况下，障碍物不仅仅只是水平或垂直的，而是带有一定角度的。这时就没法在 Flash 中模拟反弹效果了。于是，我就去各个 Flash 论坛问了一圈，发现我不是第一个问这个问题的人。在论坛里分别有三个帖子，讨论的题目都是“角度反弹”。

一些精通数学的版主试着回答这些问题。比如说反射角等于入射角。记得有个非常简单的公式告诉我们运动物体在碰撞到有角度的平面时，角度的变化。看起来不错，但也只是解决了一部分问题。我们回忆一下物体碰撞在障碍物后反弹的问题，总结出如下几步：

1. 确实何时越过边界。
2. 直接在边界上重置物体的位置。

3. 改变碰撞轴上的速度。

知道最终的角度只解决了步骤 3 中一半的问题。但是并不能知道何时物体与斜面发生碰撞，也不知道物体碰撞前停止在斜面上的位置。似乎没人能回答这些问题。我试着用学过的所有知识来回答，画的图稿可以放满整个仓库，写的程序装满整个硬盘，可最后还是失败了。如果物体与平面碰撞那就太简单了，如果与斜面碰撞就太难了。但是为什么要在本章讨论这个内容呢。

那时，www.illogicz.com 的 Stuart Schoneveld 已经制作出超强的在线物理引擎，并且可以实现这种平滑干净的碰撞。后来，我求他给我一些参考信息，他没有给我任何代码，只用了一两句话说明了整体思想，让我茅塞顿开。

他是这样说的“斜面反弹吗？先把斜面旋转成平面，然后执行反弹，最后再把它旋转回去。”

哇！这正是我想要的。我们只需要旋转坐标系，让斜面像平面一样。这就意味着旋转斜面，旋转物体坐标，再旋转物体速度向量。

现在考虑一下旋转速度向量的问题，我们把速度向量保存到 vx 和 vy 中，变量中简单地定义一个向量（角度与速度）。如果知道角度，可以直接进行旋转。但是如果只知道 vx 和 vy，就可以使用高级坐标旋转公式实现。

用图解释一下也许效果比文字要好些。如图 10-2 所示，我们看到斜面与小球发生了碰撞，而这个带箭头的向量表示小球运动的方向与速度。

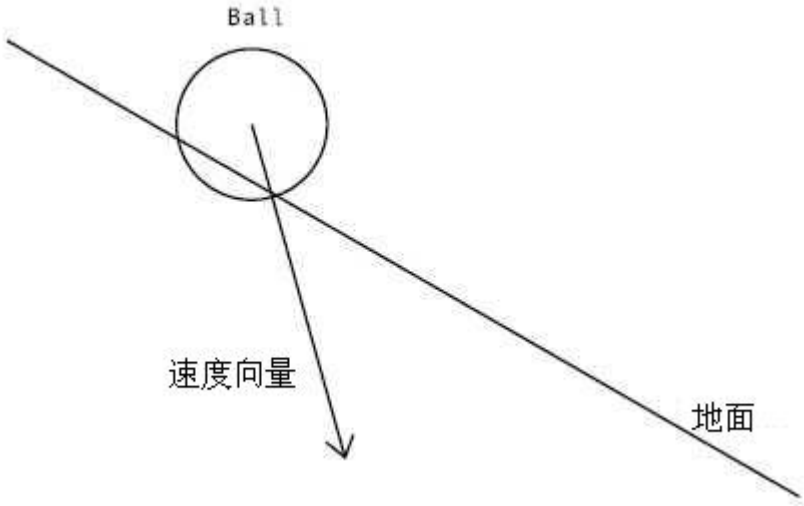


图 10-2 小球与斜面发生碰撞

在图 10-3 中，我们看到整个斜面被旋转成为一个水平面。注意速度向量也随之改变。

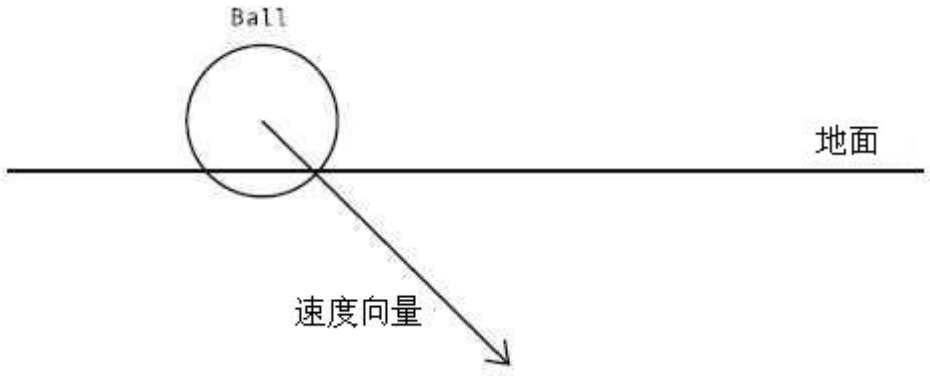


图 10-3 旋转后的情景

现在就很容易实现反弹了吧。调整小球位置，并改变 y 轴速度，如图 10-4 所示。

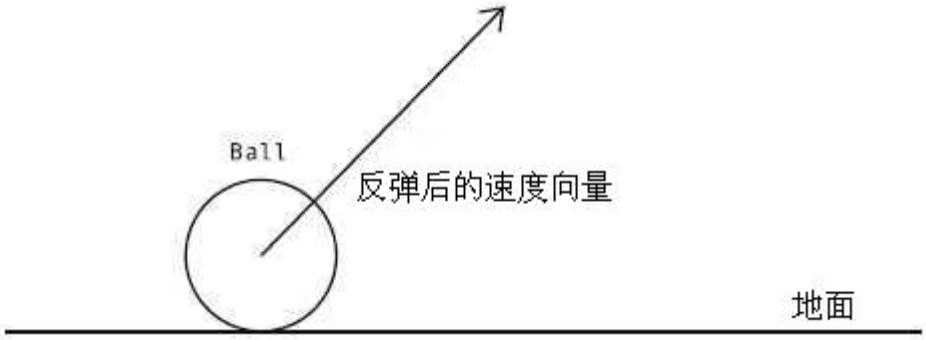


图 10-4 反弹之后

现在小球有了新的位置和速度。接下来，将所有的一切再旋转成为最初时的样子，如图 10-5 所示。

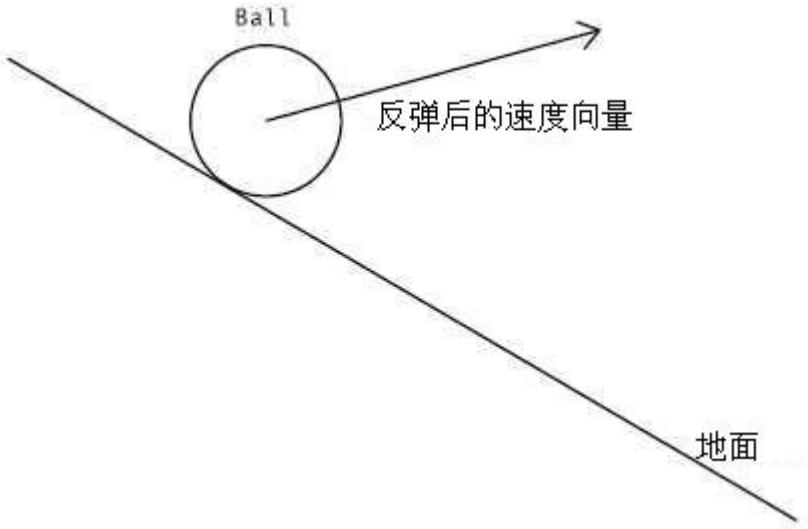


图 10-5 旋转之后

瞧！我们在斜面上检测出了碰撞的发生，调整了位置，改变了速度向量。希望这些图会对大家有所帮助，下面来看程序。

实现旋转

首先需要一个类似斜面的东西，只为了能够看到，并无实际用途。对于平面反弹，我们可以使用舞台的边界。对于斜面反弹，我们就需要一条带有角度的线(line)来表示，以便看到小球在斜面上的反弹。

因此，创建一个 Sprite 影片，加入显示列表，然后使用绘图 API 绘制一条水平线，再将影片进行一定角度的旋转。

我们同样需要 Ball 类，现在应该保证它就在手边。在为物体定位时，要确保小球在线上，以便小球可以落在线上。下面是文档类（AngleBounce.as）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class AngleBounce extends Sprite {
  private var ball:Ball;
  private var line:Sprite;
  private var gravity:Number = 0.3;
  private var bounce:Number = -0.6;
```

```
 public function AngleBounce() {
  init();
 }
 private function init():void {
  ball = new Ball();
  addChild(ball);
  ball.x = 100;
  ball.y = 100;
  line = new Sprite();
  line.graphics.lineStyle(1);
  line.graphics.lineTo(300, 0);
  addChild(line);
  line.x = 50;
  line.y = 200;
  line.rotation = 30;
  addEventListener(Event.ENTER_FRAME, onEnterFrame);
 }
 private function onEnterFrame(event:Event):void {
  // 普通的运动代码
  ball.vy += gravity;
  ball.x += ball.vx;
  ball.y += ball.vy;
  // 获得角度及正余弦值
  var angle:Number = line.rotation * Math.PI / 180;
  var cos:Number = Math.cos(angle);
  var sin:Number = Math.sin(angle);
  // 获得 ball 与 line 的相对位置
  var x1:Number = ball.x - line.x;
  var y1:Number = ball.y - line.y;
  // 旋转坐标
  var x2:Number = cos * x1 + sin * y1;
  var y2:Number = cos * y1 - sin * x1;
  // 旋转速度向量
  var vx1:Number = cos * ball.vx + sin * ball.vy;
  var vy1:Number = cos * ball.vy - sin * ball.vx;
  // 实现反弹
  if (y2 > -ball.height / 2) {
   y2 = -ball.height / 2;
   vy1 *= bounce;
  }
  // 将一切旋转回去
  x1 = cos * x2 - sin * y2;
  y1 = cos * y2 + sin * x2;
  ball.vx = cos * vx1 - sin * vy1;
  ball.vy = cos * vy1 + sin * vx1;
  ball.x = line.x + x1;
  ball.y = line.y + y1;
 }
}
```

}

开始，声明变量 ball,line,gravity,bounce。在 enterFrame 函数中执行基本运动代码。

然后，获得 line 的角度并转化为弧度制。有了角度，就可以求出正余弦的值。

接下来，用 ball 的位置减去 line 的位置，获得 ball 的 x,y 与 line 的相对位置。

下面，准备对物体进行旋转！在看到这两代码时，注意到好像是错的。

```
// 旋转坐标
var x2:Number = cos * x1 + sin * y1;
var y2:Number = cos * y1 - sin * x1;
```

这里的加号与减号与最初的坐标旋转公式是相反的，最初我们的公式是：

```
x1 = cos(angle) * x - sin(angle) * y;
y1 = cos(angle) * y + sin(angle) * x;
```

这并没有错。考虑一下，假如 line 旋转了 17 度，如果使用最初的公式，求出旋转度数会比 17 度大，是 34 度！我们实际想要旋转 -17 度，这样才会使度数变为 0。那么就应该这样计算正余弦值 Math.sin(-angle) 和 Math.cos(-angle)。但是，最后还需要用到最初的角度来使所有物体都旋转回去。

因此，可以使用一个角度旋转的变形公式进行反角度的旋转。我们看到，只需要调换加减符号即可，就这么简单。如果 line 旋转了 17 度，那么所有的物体都要旋转 -17 度，使得 line 为 0 度，或说是平面，速度向量也是如此。

请注意，我们实际不需要旋转影片中的 line。只是为了让人们看到小球是在哪里被反弹的。

接下来执行反弹，我们使用 x2,y2 的位置和 vx1,vy1 向量的值。注意因为 y2 是与 line 影片相对的，“底部”边界就是 line 自己。考虑一下小球的位置，判断什么时候 y2 大于 0 - ball.height / 2。简短的写法就是这样：

```
if(y2 > -ball.height / 2)
```

边界的设置非常明显。

最后使用最初的公式将所有的物体旋转回去。这些新的数值都是对 x1, y1, ball.vx, ball.vy 更新过的值。我们所需要的就是重新设置小球的实际位置，通过 x1 和 y1 与 line.x 和 line.y 相加得出。

花些时间来实验一下这个例子。试改变 line 的 rotation 属性以及改变 line 与 ball 不同的位置，观查运行结果。确保它们都能正常工作。

优化代码

前面我们已经看过一些代码优化的例子。通常是使用一次执行代替多次执行，或干脆不执行。

我们前面写的那段代码只是为了看得比较清楚。其中有一些代码实际上并不需要执行。多数代码只有在 ball 与 line 产生接触时才执行。因此，多数时间只需要执行基本的运动代码。换句话讲，我们要将代码放到 if 语句中去：

```
if(y2 > -ball.height / 2)
```

所以我们只需知道变量 y2。为了得到它需要 x1 和 y1 以及 sin 和 cos。但是如果 ball 没有碰到 line，就不需要知道 x2 或 vx1 和 vy1。因此，这些都可以只在 if 语句中出现。同样，如果没有产生碰撞，就不需要对任何物体进行旋转或设置 ball 的位置。因此，所有 if 语句后面的内容都可以放在 if 语句里面执行。于是就得出了优化版的 onEnterFrame 方法（见 AngleBounceOpt.as）：

```
private function onEnterFrame(event:Event):void {
    // 普通的运动代码
    ball.vy += gravity;
    ball.x += ball.vx;
```

```
        ball.y += ball.vy;
        // 获得角度及正余弦值
        var angle:Number = line.rotation * Math.PI / 180;
        var cos:Number = Math.cos(angle);
        var sin:Number = Math.sin(angle);
        // 获得 ball 与 line 的相对位置
        var x1:Number = ball.x - line.x;
        var y1:Number = ball.y - line.y;
        // 旋转坐标
        var y2:Number = cos * y1 - sin * x1;
        // 实现反弹
        if(y2 > -ball.height / 2) {
            // 旋转坐标
            var x2:Number = cos * x1 + sin * y1;
            // 旋转速度向量
            var vx1:Number = cos * ball.vx + sin * ball.vy;
            var vy1:Number = cos * ball.vy - sin * ball.vx;
             y2 = -ball.height / 2;
             vy1 *= bounce;
            // 将一切旋转回去
            x1 = cos * x2 - sin * y2;
            y1 = cos * y2 + sin * x2;
            ball.vx = cos * vx1 - sin * vy1;
            ball.vy = cos * vy1 + sin * vx1;
            ball.x = line.x + x1;
            ball.y = line.y + y1;
        }
    }
```

所有粗体的内容都是从 if 语句外面移到里面去的，所以只有产生碰撞时它们才会执行，这样做比每一帧都执行要好很多。可以想象我们节省了多少 CPU 资源吗？这样的考虑是非常重要的，尤其是当影片变得越来越多，代码变得越来越复杂时。

动态效果

现在我们可以将这个程序变得更加动态些，实时地改变 line 的角度。只需要一行代码即可搞定，在 onEnterFrame 方法的第一行写入：

```
line.rotation = (stage.stageWidth/ 2 - mouseX) * .1;
```

现在我们只要前后移动鼠标，line 就会随之倾斜，小球也会立即进行调整。完整的代码可见文档类 AngleBounceRotate.as。

修正“跌落”问题

大家也许注意到这样一个问题，即使 ball 离开了 line，它依然会沿着那条边运动。看起来有些奇怪，但是要记住 ball 并不是真的与 line 影片产生交互。虽然执行的结果是非常准确的，我们可以认为实际上没有碰撞，一切都是计算出来的结果。因此 ball 根本就不知道 line 影片，也不知道影片的起点与终点。但是我们可以告诉它 line 在哪里——使

用简单的碰撞检测或更精确的边界检测方法。下面让我们看看这两种方法，由您决定使用哪一种。

碰撞测试

找出 line 位置最好的方法是将除了基本移动代码以外所有的内容都放到碰撞检测的 if 语句中，如下：

```
private function onEnterFrame(event:Event):void {
    // 普通的运动代码
    ball.vy += gravity;
    ball.x += ball.vx;
    ball.y += ball.vy;
    if(ball.hitTestObject(line) {
        // 剩下的所有内容都在这个函数中
    }
}
```

虽然方法很简单，但是已经可以满足大部分的实际需要，下面还有一种更加精确一些的方法，也是我比较喜欢的一种方法。当然，这就需要更加复杂一些的计算。

判断边界

个人认为 getBounds 方法是 ActionScript 中最不被充分利用的方法。我曾考虑在第九章中配合碰撞检测提一下，但那章的内容已经很多了，而且我发现最适合使用的地方应该是下面这个地方。所以把它放在这一章为大家讲解。

回顾一下碰撞检测一章，一定还记得矩形边界吧。为了让大家回忆起来，我们一起来回顾一下。矩形边界是指用于包围舞台上一个显示对象可见图形元素的矩形边框。hitTestObejct 和 hitTestPoint 函数都应用了这种边界。

getBounds 函数直接给出了矩形边界的位置和大小的值。下面是这个函数的基本用法：

```
bounds = displayObject.getBounds(targetCoordinateSpace)
```

可以看到，这个方法作为任何显示对象的方法来调用，并返回 flash.geom.Rectangle 的实例，描述了矩形的大小与位置。

首先，来看一下这个唯一的参数，targetCoordinateSpace。是什么意思？

我们使用 targetCoordinateSpace 参数来指定用哪种视角来描述矩形边界。大多数情况下，这个参数是该物体的父级显示对象。比如，如果主文档类就是一个 Sprite 影片，我们叫它 sprite，那么 getBounds(this)，就表示“根据主影片的坐标，给出这个 sprite 的矩形边界”。另外，如果在一个 sprite 里面又创建或加载了其它 sprite，就需要通过外层影片的位置得到矩形边界。写法如下：

```
childSpite.getBounds(parentSprite);
```

这个意思是说，我们要得到 childSprite 影片的矩形边界，而这个影片位于 parentSprite 的里面，并且我们想要用 parentSprite 坐标空间的视角来描述它。显然，targetCoordinateSpace 应该是个显示对象，或是继承自 DisplayObject 类的实例。文档类，Sprite, MovieClip 都是显示对象，没问题。

下面看看 getBounds 函数的返回值。前面说过，返回值是一个 Rectangle 的实例，里面包涵了矩形边界的数据。以前在使用 Rectangle 类时看到过，它里面有四个属性：x, y, width, heigth。并且我们可以使用这些信息。它还包括其它一些非常有用的属性：left,right,top,bottom。大家应该可以猜出它们的意思吧。

让我们来试一试。将一个 Ball 类的实例加入显示列表，然后插入如下代码（不要忘记导入 flash.geom.Rectangle 类）：

```
var bounds:Rectangle = ball.getBounds(this);
trace(bounds.left);
trace(bounds.right);
trace(bounds.top);
trace(bounds.bottom);
```

做一个有趣的实验，将第一行代码改为：

```
var bounds:Rectangle = ball.getBounds(ball);
```

现在，小球的边界是从它自身的视角来看的；换句话讲，这个位置与它自身注册点的位置是相关的。因为小球是以 0,0 点作为中心绘制的，left 和 top 的值应该是负数，实际上就等于 -right 和 -bottom。这里值得一提的是，如果我们调用 getBounds 时没有给出参数，那么结果是相同的，物体本身也就是它的目标坐标空间。

现在也许大家都忘了为什么要讨论边界了吧，回忆一下。我们想要知道物体何时从 line 的上面跌落下来，还记得吗？因此，可以在 line 上面调用 getBounds，然后得到它的 left 和 right。如果 ball 的 x 小于 bounds.left，或大于 bounds.right，就说明它从 line 上面掉落了。再怎么说也不如看代码的好，以下是程序：

```
private function onEnterFrame(event:Event):void {
    line.rotation = (stage.stageWidth/ 2 - mouseX) * .1;
    // 普通的运动代码
    ball.vy += gravity;
    ball.x += ball.vx;
    ball.y += ball.vy;
    var bounds:Rectangle = line.getBounds(this);
    if(ball.x > bounds.left && ball.x < bounds.right) {
        // 剩下的所有内容都在这个函数中
    }
}
```

大家可以在 AngleBounceBounds.as 中找到完整的程序。

修正“线下”问题

碰撞检测与边界判断这两种方法都要先确定 ball 与 line 相互接触，然后进行坐标旋转获得调整后的位置与速度。如果 ball 的 y 坐标旋转后的位置 y2 超过了 line，则执行反弹。但是如果 ball 自下而上穿过了 line，又该怎么办？假设 line 在舞台的中心位置，然后 ball 从下面反弹上来。如果碰撞检测与边界判断的结果都是 true，Flash 就认为 ball 刚刚从 line 上弹出去，它会把 ball 从 line 下面移到上面去。那么我的解决办法是比较 vy1 和 y2，并且只在 vy1 大于 y2 时才进行反弹。如图 10-6 所示。

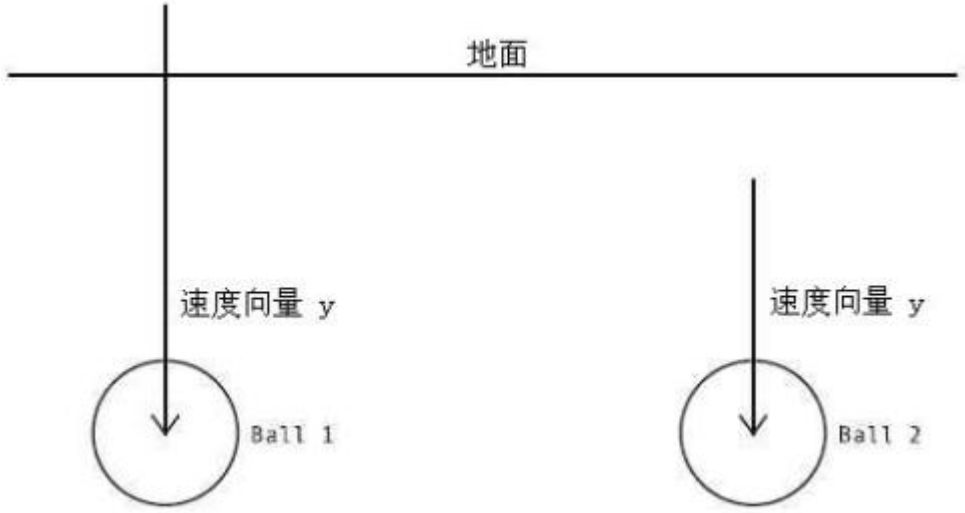


图 10-6 线上通过还是线下通过？

左面 ball 的 y 速度大于 ball 与 line 的相对距离。这就意味着只要它进行运动就会跑到线上面去。而右面 ball 的速度小于 ball 与 line 的相对距离。换句话讲，在这一帧内，它位于线下，并且下一帧还是在线下。只有在小球从线上往下掉落时从进行反弹。下面看一下这段代码的修改。以下是 enterFrame 代码段：

```
// 旋转坐标
var y2:Number = cos * y1 - sin * x1;
// 实现反弹
if(y2 > -ball.height / 2) {
// 旋转坐标
var x2:Number = cos * x1 + sin * y1;
// 旋转速度向量
var vx1:Number = cos * ball.vx + sin * ball.vy;
var vy1:Number = cos * ball.vy - sin * ball.vx;
...
```

我们只需要把 y2 < vy1 加入到 if 语句中：

```
if(y2 > -ball.height / 2 && y2 < vy1)
```

如果这样的话，就需要先计算出 vy1。代码如下：

```
// 旋转坐标
var y2:Number = cos * y1 - sin * x1;
// 旋转速度向量
var vy1:Number = cos * ball.vy - sin * ball.vx;
// 实现反弹
if(y2 > -ball.height / 2 && y2 < vy1) {
// 旋转坐标
var x2:Number = cos * x1 + sin * y1;
// 旋转速度向量
var vx1:Number = cos * ball.vx + sin * ball.vy;
...
```

这样一来每帧都要多做一些计算，这也是为了获得较高精确度和真实度付出的代价。大家可以决定是否需要它。如果说小球不可能到达线下面的话，就不必考虑这个问题。

在文档类 AngleBounceFinal.as 中，我加入了墙壁与地面的反弹，以便可以让大家看

到小球从线上掉落后的情况。试将我们刚才讨论的代码删除，观察有何不同。

OK，下面我们进入本章最后一个大型的例子程序。

多角度反弹

目前为止，我们都是在讨论一条线或一个斜面的问题。要处理多个斜面也不是很复杂的事情，只需要创建多个斜面并进行循环。我们可以把角度反弹的代码抽象到一个函数中，每次进行调用。

本章前面的这些例子中，我都让代码尽可能地简单，只给大家必要的代码进行演示。然而，接下来这段代码是一个完正的程序，用到了前面几章学过的一些技术（为的是唤醒大家的记忆）。

本例与前面那几个例子很像，都使用了同样的 ball 和 line 影片，只不过这次我把线缩短了一些,以便可以多放几条进去。在舞台上放置了五条线和一个小球。线条由数组 lines 保存，并放置于舞台上。如图 10-7 所示。

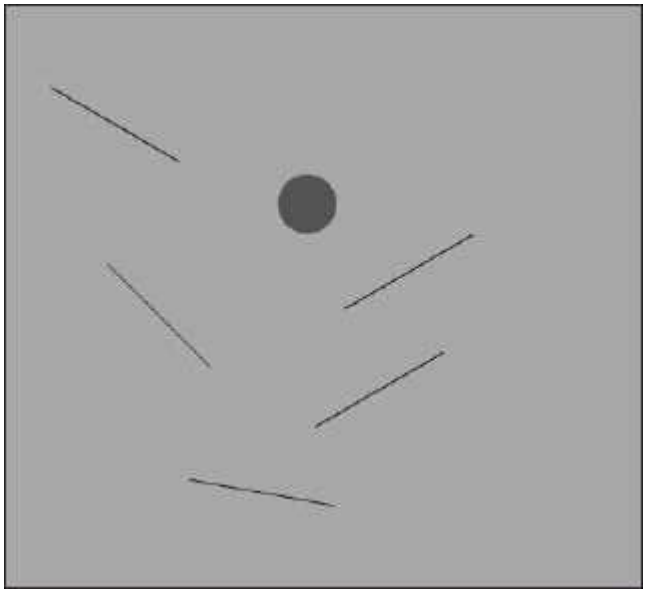


图 10-7 多线条

以下是代码（可在 MultiAngleBounce.as 中找到）：

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

 public class MultiAngleBounce extends Sprite {
  private var ball:Ball;
  private var lines:Array;
  private var numLines:uint = 5;
  private var gravity:Number = 0.3;
  private var bounce:Number = -0.6;

  public function MultiAngleBounce() {
   init();
  }
```

```
 private function init():void {
  stage.scaleMode = StageScaleMode.NO_SCALE;
  stage.align = StageAlign.TOP_LEFT;
  ball = new Ball(20);
  addChild(ball);
  ball.x = 100;
  ball.y = 50;

  // 创建 5 个 line 影片
  lines = new Array();
  for (var i:uint = 0; i < numLines; i++) {
   var line:Sprite = new Sprite();
   line.graphics.lineStyle(1);
   line.graphics.moveTo(-50, 0);
   line.graphics.lineTo(50, 0);
   addChild(line);
   lines.push(line);
  }
  // 放置并旋转
  lines[0].x = 100;
  lines[0].y = 100;
  lines[0].rotation = 30;

  lines[1].x = 100;
  lines[1].y = 230;
  lines[1].rotation = 45;

  lines[2].x = 250;
  lines[2].y = 180;
  lines[2].rotation = -30;

  lines[3].x = 150;
  lines[3].y = 330;
  lines[3].rotation = 10;

  lines[4].x = 230;
  lines[4].y = 250;
  lines[4].rotation = -30;

  addEventListener(Event.ENTER_FRAME, onEnterFrame);
 }
 private function onEnterFrame(event:Event):void {
  // normal motion code
  ball.vy += gravity;
  ball.x += ball.vx;
  ball.y += ball.vy;

  // 舞台四周的反弹
  if (ball.x + ball.radius > stage.stageWidth) {
   ball.x = stage.stageWidth - ball.radius;
```

```
            ball.vx *= bounce;
        } else if (ball.x - ball.radius < 0) {
            ball.x = ball.radius;
            ball.vx *= bounce;
        }
        if (ball.y + ball.radius > stage.stageHeight) {
            ball.y = stage.stageHeight - ball.radius;
            ball.vy *= bounce;
        } else if (ball.y - ball.radius < 0) {
            ball.y = ball.radius;
            ball.vy *= bounce;
        }
        // 检查每条线
        for (var i:uint = 0; i < numLines; i++) {
            checkLine(lines[i]);
        }
    }
    private function checkLine(line:Sprite):void {
        // 获得 line 的边界
        var bounds:Rectangle = line.getBounds(this);
        if (ball.x > bounds.left && ball.x < bounds.right) {

            // 获取角度与正余弦值
            var angle:Number = line.rotation * Math.PI / 180;
            var cos:Number = Math.cos(angle);
            var sin:Number = Math.sin(angle);

            // 获取 ball 与 line 的相对位置
            var x1:Number = ball.x - line.x;
            var y1:Number = ball.y - line.y;

            // 旋转坐标
            var y2:Number = cos * y1 - sin * x1;

            // 旋转速度向量
            var vy1:Number = cos * ball.vy - sin * ball.vx;

            // 实现反弹
            if (y2 > -ball.height / 2 && y2 < vy1) {
                // 旋转坐标
                var x2:Number = cos * x1 + sin * y1;

                // 旋转速度向量
                var vx1:Number = cos * ball.vx + sin * ball.vy;

                y2 = -ball.height / 2;
                vy1 *= bounce;

                // 将一切旋转回去
                x1 = cos * x2 - sin * y2;
```

```
            y1 = cos * y2 + sin * x2;
            ball.vx = cos * vx1 - sin * vy1;
            ball.vy = cos * vy1 + sin * vx1;
            ball.x = line.x + x1;
            ball.y = line.y + y1;
          }
        }
      }
    }
}
```

代码很多，不过很容易解释，其中的每一部分大家都应该认识。复杂的程序并不代表复杂的代码，而常常是由许多熟悉的代码段构成的。本例中，checkLine 方法与前面版本中的 onEnterFrame 是一样的。只不过是被 for 循环调用了五次而已。

大家感兴趣的话可以做一个小小的优化。比方说每次都要循环判断多个斜面，但在很多系统中，只要发现小球与某个斜面发生了碰撞并产生交互的话，我们就不需要再继续判断其它的斜面了，这时我们就可以退出这个循环。为了实现这个功能，需要让 checkLine 函数返回 true 或 false，来告诉我们是否发生了碰撞。然后在 onEnterFrame 函数中执行这样的循环：

```
for(var i:uint = 0; i < numLines; i++) {
    if(checkLine(lines[i])) {
        break;
    }
}
```

在一些情况下，尤其是线条非常密集的时候，需要每帧都判断所有的线条。如果是这样，那么是否决定使用优化，决于个人的需要。

本章重要公式

下面回忆一下本章的两个主要公式。

坐标旋转：

```
x1 = Math.cos(angle) * x - Math.sin(angle) * y;
y1 = Math.cos(angle) * y + Math.sin(angle) * x;
```

反坐标旋转：

```
x1 = Math.cos(angle) * x + Math.sin(angle) * y;
y1 = Math.cos(angle) * y - Math.sin(angle) * x;
```

10.5 小结

如你在本章所看到的，坐标旋转可以给你一个非常复杂的行为，但所有的这些都可以压缩为两个不会变化的公式。一旦你适应了这些公式，你可以应用在任何地方。我希望你已经开始明白如何通过加入越来越简单的技术创建出非常复杂且非常真实的动画。

在下一章你还将大量使用坐标旋转公式，在那里你将学会如何控制不同速度和质量的物体碰撞所产生的结果。

第十一章　撞球物理

我们都希望技术性的书籍，可以潜入深，由简单到复杂。本章内容的复杂度达到了顶点。并不是说接下来的章节会越来越容易，但是希望大家在学习本章内容时不要偷工减料。我会带大家一步步地学习本章的概念，如果到现在为止大家都能很好地跟上我的步伐，那就再好不过了。

本章我们要关注动量：两个物体发生碰撞后动量会发生什么样的变化，动量的守恒，以及如何在 ActionScript 中应用动量。

在本章的例子中使用的对象，都本这简单直接的原则，这个学科通常是指“撞球物理”。我们很快会看到一些不同大小的桌球相互碰撞的例子。

在给大家代码时，我都会从简单的一维运动开始举例，为的是容易理解。然后再过渡到二维坐标上，比如上一章的坐标旋转。本质上讲，就是将二维场景旋转成一个平面。最后就可以忽略一个轴，而只对一维场景进行操作。这些都让读者很期待接下来的内容。让我们从质量与动量开始吧。

质量

本书前面的章节讨论过几种运动的概念：速度，加速度，向量，摩擦力，反弹，缓动，弹性以及重力。而我成功的避开了物体的质量问题。现在我们要重新讨论这个问题，严格来讲，书中有几个地方都要在方程中加入质量。我通常只关心让物体正确地运动看起来，注重于制作出正确的效果。最重要的是执行结果必需有很高的效率，不能让程序毁掉 Flash。

不过，从现在起再也不能忽略质量问题了。物体的质量与动量紧密相关，所以我们不得不直接面对它。

什么是质量呢？在地球上，我们通常认为质量就是物体的重量。它们之间确实关系密切，因为重量与质量是成比例的。物体的质量越大，重量就越大。事实上，对于质量和重量我们使用同样的测量单位：千克，磅等等。严格来说，质量是物体所承受速度大小的测量单位。

因此，物体的质量越大，这个物体就越难移动，也不易改变其本身的运动（减慢，加速或改变方向）。

质量也与加速度和力有关。物体质量越大，给它加速度的力就越大。我的 Chevy Cavalier [注：一款轿车的名字] 引擎可产生足够的力给小轿车作为加速度。但是，这样的力对于大卡车的加速度来说是远远不够的。因为卡车的质量比较大，引擎需要更多的力。

动量

现在来看动量。它是由物体的质量和速度构成的，是质量与速度的乘积。动量通常用字母 p 表示，质量用 m 表示，速度用 v 表示。下面这个公式应该无需解释了：

```
p = m * v
```

也就是说，质量小，速度大的动量与质量大速度小的动量相似。前面所说大卡车以每小时 20 迈的速度运动就足以至人于死地。此外，子弹的质量非常轻，但是速度非常快，同样也是致命的。

由于速度 v 是一个向量（方向与量值），所以动量也必然是一个向量。动量的向量与速度的向量是相同的。为了完整地描述动量，我们可以说：

5 公斤 * 20 米/秒 角度为 23 度的方向运动。

很复杂吧？现在大家知道我为什么要等到这里才讲了吧。

动量守恒

最后，我们来看本章的核心：动量守恒。什么意思？动量是守恒不变的？什么时候？哪里？如何守恒？OK，慢慢来。动量守恒可在碰撞中使用。碰撞检测一章中的碰撞反应都是仿造出来的。而动量守恒才是碰撞反应的实际原理。

有了动量守恒，我们就可以这样说，“在碰撞之前，一个物体以 A 速度运动，另一个物体以 B 速度运动。碰撞之后，该物体以 C 速度运动，另一个物体以 D 速度运动。”，分解来看，我们知道速度 v 是由速度和方向构成的，如果在碰撞之前知道了两个物体的速度和方向，就可以求出两物体碰撞后的速度和方向。非常实用的定理，大家也是这样认为的吧。

注意：我们需要知道每个物体的质量。实际应用中，如果在碰撞前，知道每个物体的质量，速度和方向，就可以求出碰撞后物体的运动方向和速度了。

OK,那么动量守恒到底能为我们做些什么呢？它又是什么样的呢？动量守恒定理告诉我们：在系统中，碰撞前的动量总合与碰撞后的动量总合相等。那么定理中所谓的系统是指什么呢？它是指一个拥有动量的物体的集合。多数情况下，是指一个封闭系统，也就是不受其它外力影响的系统。换句话讲，也就是可以忽略除实际碰撞以外的一切。对于我们而言，只关注两个物体之的反作用力，系统总是以物体 A 和物体 B 这样的形式出现。系统的总动量就是由系统中所有物体的动量结合而成。按照这个例子来讲，就是把物体 A 和物体 B 的动量相加到在一起。因此，碰撞前将动量之和，与碰撞后动量之和相同的。大家也许会问，“那太好了，但是定理没有告诉我们怎么求得的动力啊。”。不要急，马上来说这个问题，说好了，要一步步来。下面几段，要慢慢来看，因为会有公式！

在学数学前，给大家几条建议。先不要想怎么把公式转换成代码，马上会讲到。现在大家要集中看好下面几个公式的概念。“一个数加上另一个数等于另一个数加上这个数。当然，这是有意义的。”。

OK，如果碰撞前的动量之和等于碰撞后的动量之和，并且动量等于速度乘以质量，那么对于两个物体——物体 0 和 物体 1——我们会得出 [momentum :动量]：

momentum0 + momentum1 = momentum0Final + momentum1Final

或者：

(m0 * v0) + (m1 * v1) = (m0 * v0Final) + (m1 * v1Final)

现在我们要知道的就是 物体 0 和 物体 1 的最终速度，也就是 v0Final 和 v1Final。解一个有两个未知数的方程需要找出另一个有两个相同未知数的方程。物理学中恰好有这样一个方程，这就是动能公式。我们不需要关心动能是什么，只需要借这个公式来解决我们的问题，用完后再还回去。动能公式如下：

$KE = 0.5 * m * v^2$

虽然这里用 v 表示速度，但是严格来讲，动能不是向量，它只是速度向量中的量值的大小，并不涉及方向。

碰撞前的动能刚好与碰撞后相同。因此可以这样表示：

KE0 + KE1 = KE0Final + KE1Final

或者：

$(0.5 * m0 * v0^2) + (0.5 * m1 * v1^2) = (0.5 * m0 * v0Final^2) + (0.5 * m1 * v1Final^2)$

两边同时消去因数 0.5 后：

$(m0 * v0^2) + (m1 * v1^2) = (m0 * v0Final^2) + (m1 * v1Final^2)$

这样就得到了两个方式，带有两个相同的未知变量：v0final 和 v1Final。然后就可以为每个未知数求出一个等式。接下来就是代数问题了，为了让你我都不感到头痛，直接给出最终的公式。如果您喜欢求代数式，或者想在学校拿到更高的学分的话，我建议您坐下来拿几张纸几支笔自己算一算。那么计算出的结果应该是这样的：

$$v0Final = \frac{(m0 - m1) * v0 + 2 * m1 * v1}{m0 + m1}$$

$$v1Final = \frac{(m1 - m0) * v1 + 2 * m0 * v0}{m0 + m1}$$

现在大家知道为什么我一开始说这章非常之复杂了吧。

下面先在单轴上进行应用，随后制作两个轴上的运动时会加入坐标旋转。继续！

单轴上的动量守恒

现在公式已经有了，可以开始应用了。第一个例子，我们继续使用 Ball 类，但这次要加入质量（mass）这个属性。新的代码如下（Ball.as）：

```
package {
 import flash.display.Sprite;
 public class Ball extends Sprite {
  private var radius:Number;
  private var color:uint;
  public var vx:Number = 0;
  public var vy:Number = 0;
  public var mass:Number = 1;
  public function Ball(radius:Number=40, color:uint=0xff0000) {
   this.radius = radius;
   this.color = color;
   init();
  }
  public function init():void {
   graphics.beginFill(color);
   graphics.drawCircle(0, 0, radius);
   graphics.endFill();
  }
 }
}
```

我们要创建两个 Ball 类的实例，使用不同的大小，位置和质量。开始先忽略 y 轴的运动。因此影片开始时基本的设置如图 11-1 所示。

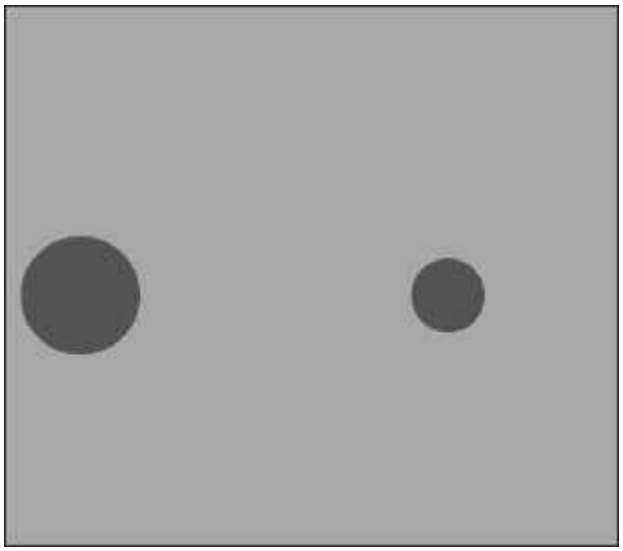


图 11-1 单轴动量守恒运动的舞台设置

类的开始是创建两个小球放入舞台上，然后进行单轴上的基本运动代码，并使用简单的距离碰撞检测：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Billiard1 extends Sprite {
  private var ball0:Ball;
  private var ball1:Ball;
  public function Billiard1() {
   init();
  }
  private function init():void {
   ball0 = new Ball(40);
   ball0.mass = 2;
   ball0.x = 50;
   ball0.y = stage.stageHeight / 2;
   ball0.vx = 1;
   addChild(ball0);
   ball1 = new Ball(25);
   ball1.mass = 1;
   ball1.x = 300;
   ball1.y = stage.stageHeight / 2;
   ball1.vx = -1;
   addChild(ball1);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   ball0.x += ball0.vx;
   ball1.x += ball1.vx;
   var dist:Number = ball1.x - ball0.x;
   if (Math.abs(dist) < ball0.radius + ball1.radius) {
    // 在此执行反作用力
   }
  }
 }
```

```
}
```

目前，唯一的问题是如何执行反作用力。先来看 ball0。将 ball0 看作物体 0，ball1 就是物体 1，运用下面这个公式：

$$v0Final = \frac{(m0 - m1) * v0 + 2 * m1 * v1}{m0 + m1}$$

在 ActionScript 中就变成了下面这段代码：

```
var vx0Final:Number = ((ball0.mass - ball1.mass) * ball0.vx +
2 * ball1.mass * ball1.vx) /
(ball0.mass + ball1.mass);
```

不难理解吧。那么 ball1 也是如此：

$$v1Final = \frac{(m1 - m0) * v1 + 2 * m0 * v0}{m0 + m1}$$

代码如下：

```
var vx1Final:Number = ((ball1.mass - ball0.mass) * ball1.vx +
2 * ball0.mass * ball0.vx) /
(ball0.mass + ball1.mass);
```

最后 onEnterFrame 中的代码如下：

```
private function onEnterFrame(event:Event):void {
 ball0.x += ball0.vx;
 ball1.x += ball1.vx;
 var dist:Number = ball1.x - ball0.x;
 if (Math.abs(dist) < ball0.radius + ball1.radius) {
  var vx0Final:Number = ((ball0.mass - ball1.mass) * ball0.vx +
  2 * ball1.mass * ball1.vx) /
  (ball0.mass + ball1.mass);
  var vx1Final:Number = ((ball1.mass - ball0.mass) * ball1.vx +
  2 * ball0.mass * ball0.vx) /
  (ball0.mass + ball1.mass);
  ball0.vx = vx0Final;
  ball1.vx = vx1Final;
  ball0.x += ball0.vx;
  ball1.x += ball1.vx;
 }
}
```

注意，在计算 vx0Final 时用到了 ball1.vx，反之亦然。因此，不得不把结果作为临时变量保存起来，而不是直接将它们赋值给 ball0.vx 和 ball1.vx。

设置物体位置

前面例子中动作脚本的最后两行应该解释一下。在求出了每个小球的新的速度后，再把它们加到小球的位置上。这是个新内容，为什么要这么做？回忆一下，在前面的反弹例子中，需要重置影片的位置，为的是不让它进入到墙体内。当物体与墙面接触时就对它进行移动。

这里也是如此，但是这次有两样东西需要移动，因为不想让它们彼此相互吸引。这样一看就是错的，而且通常都会使两个物体永远地粘在一起。

我们要将其中一个小球移动到另一个小球的边上。但是要移动哪一个呢？无论移动哪个都会使物体像跳到新位置上一样不自然，尤其是在运动速度很慢的情况下。

有很多的方法可用来确定小球移动的位置，这些方法从简单到复杂，从精确到模拟的都有。在第一个例子中用的是简单的解决方法只要加上新速度，就可以让两个物体分开。我认为这是最真实也是最简单的方法——只要两句代码就能完成。随后，在“解决潜在问题”一节中，我会给大家介绍一个更加健全的解决方法。

试编译运行文档类 Billiard1.as，改变每个小球的质量与速度，也可以改变小球的大小。注意，ball 的大小不会对反作用力有什么影响。多数情况下，物体体积越大，则质量就越大，可以根据两个小球的相对大小给出真实的质量。通常，我在给出质量时都是在试数，看上去合适就可以。严格来说应该是“小球的体积扩大两倍，则质量也要扩大两倍”。

代码优化

这段代码最不好的地方就是中间大段的等式。事实上，最坏的部分就是两个几乎完全相同的等式出现了两次，如果可以减少一个就好了。OK，没有问题。

用两个物体速度相减求出总的速度，看起来有些奇怪，但是要从系统的角度来思考。假设系统中有两辆车在高速路上。其中一辆车的速度是 50 mph 另一辆车的速度是 60 mph。坐在随便一辆车中，可以看到另一辆车是以 10 mph 或 -10 mph 的速度前进。换句话讲，另一辆车不是在你前，就是在你后面。

因此，在碰撞之前，要求出总的速度（以 ball1 的角度），用 ball0.vx 减去 ball1.vx：

```
var vxTotal:Number = ball0.vx - ball1.vx;
```

最后，在计算出 vx0Final 之后，再将它与 vxTotal 相加，就得出了 vx1Final。也许有些违反直觉，没关系试验一下：

```
vx1Final = vxTotal + vx0Final;
```

OK!这样比每次输入两遍公式要好很多。现在，ball1.vx 这个式子不会对 ball0.vx 造成任何影响。所以，又可以把两个临时变量去掉了。以下是修正后的 onEnterFrame 方法（见文档类 Billiard2.as）：

```
private function onEnterFrame(event:Event):void {
 ball0.x += ball0.vx;
 ball1.x += ball1.vx;
 var dist:Number = ball1.x - ball0.x;
 if (Math.abs(dist) < ball0.radius + ball1.radius) {
  var vxTotal:Number = ball0.vx - ball1.vx;
  ball0.vx = ((ball0.mass - ball1.mass) * ball0.vx +
  2 * ball1.mass * ball1.vx) /
  (ball0.mass + ball1.mass);
  ball1.vx = vxTotal + ball0.vx;
  ball0.x += ball0.vx;
  ball1.x += ball1.vx;
 }
}
```

现在，已经去掉了很多数学运算，并且运行结果相同——不错。

这些公式并要求大家都能记住，除非您是学物理的。除非经常使用，否则很难记忆。就我个人而言，在我要使用这个公式时，都会翻到第一个版本的例子中，直接复制！

两个轴上的动量守恒

OK，深呼吸，进入下一级。现在，我们已经用过了这个冗长的公式，而它几乎是即插即用型的。只需要把两个物体的质量和速度插入公式中，就可以得到结果。

下面，进入更为复杂的一级——二维空间。本章开始时说过，其策略是使用坐标旋转。让我们来看看原因。

理解原理及策略

图 11-2 表示刚才看到那个例子：一维的碰撞。

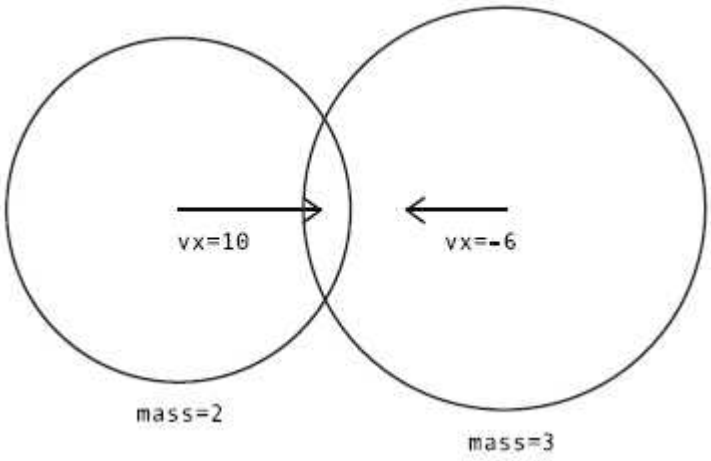


图 11-2 一维碰撞

可以看到，两个物体有着不同的大小，不同的质量以及不同的速度。速度用箭头表示，就是向量。回忆一下，速度向量指针表示运动的方向，长度表示速度的大小。

一维的例子非常简单，因为两个速度向量都在 x 轴上。所以，可以直接加上或减去它们的量值（长度）。现在，请看图 11-3，小球在二维空间中的碰撞。

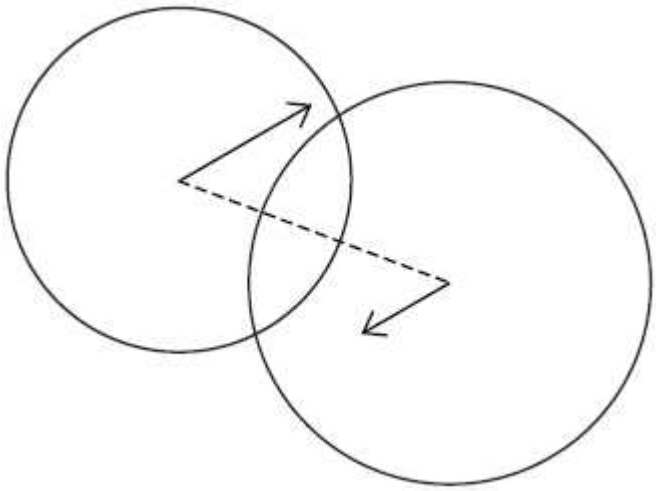


图 11-3 二维碰撞

速度向量完全不同了，不能单纯地将速度代入到动量守恒公式中，因为，这会导致完全错误的结果。那么应该如何解决呢？

通过旋转第二张图，可以得到与第一张非常相似的图。首先，要知道两个小球之间形成的角度并且旋转整个场景——坐标和速度——逆时针旋转。例如，如果角度是 30 度，就将所有的物体旋转 -30 度，这与第十章的斜面反弹是一样的。结果如图 11-4 所示。

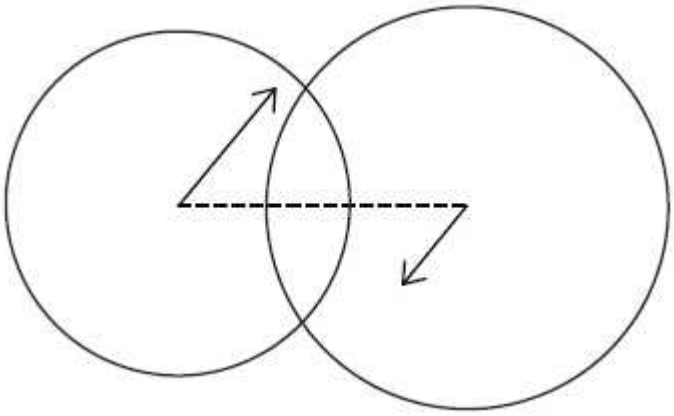


图 11-4 旋转后的二维碰撞

两个小球之间的角度非常重要，这个角度称为碰撞角度。重要的原因在于，它是小球速度的一部分——速度的一部分要依赖于角度。

下面请看图 11-5。这里，在每个向量上面加入了 vx 和 vy。注意，两个小球的 vx 都严格地依照碰撞的角度来定。

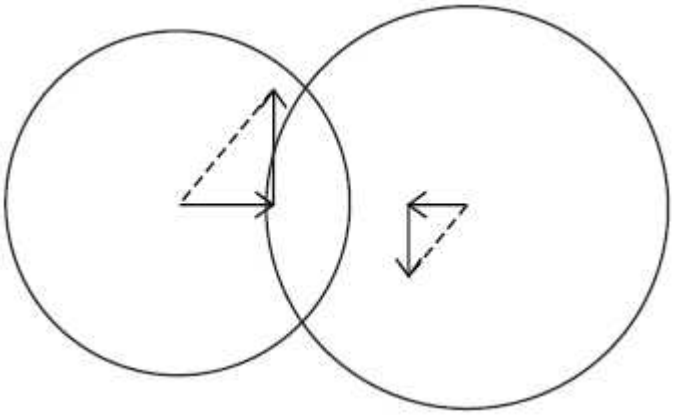


图 11-5 加入 x,y 速度

我们说过，只关心碰撞的角度。那么现在它就是 vx，可以忘掉 vy。如图 11-6 所示。

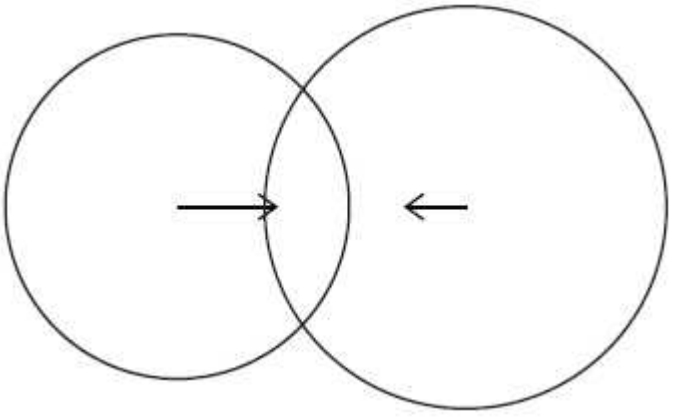


图 11-6 只关心 x 速度

看眼熟吗？这就是第一张图!现在可以使用即插即用的动量公式来解决这个问题了。(请注意解题的步骤！)

在应用这个公式时，会得出两个新的 vx 值。记住 vy 的值永远不变。vx 的变化如图 11-7 所示。

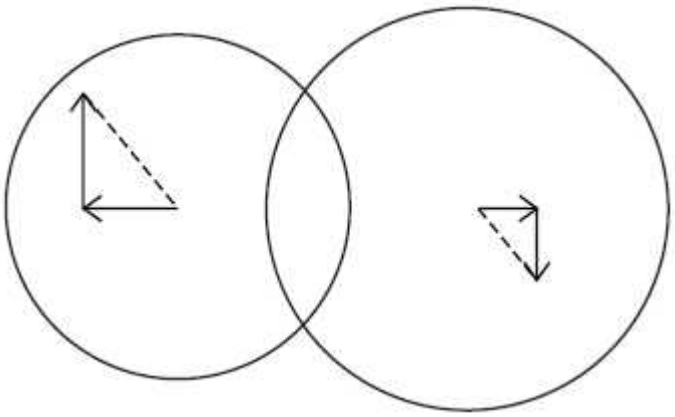


图 11-7 新的 x 速度，y 速度不变，得出新的速度

猜到下面该怎么办了吗？只需要把一切旋转回去，如图 11-8 所示。

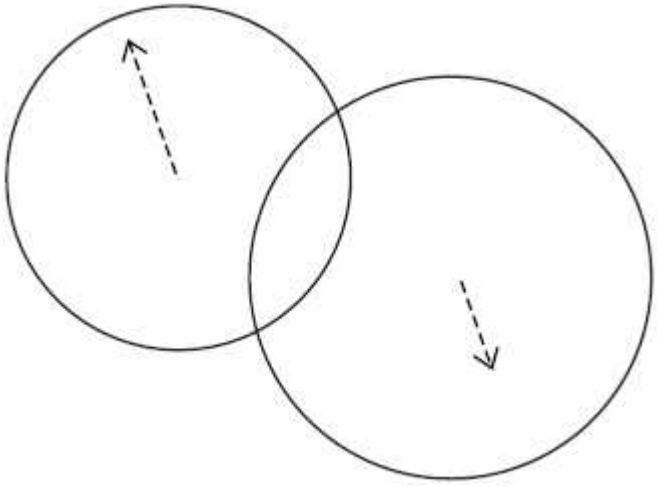


图 11-8 将一切旋转回来

下面把这个过程转换为代码。对我来说最难的地方就是要一再地说服大家“这很容易”。

编写代码

首先要让两个小球以一定的角度运动并且最后相互碰撞。开始的设置与前面相同，两个小球实例：ball0 和 ball1。这次让它们变大一点，如图 11-9 所示，这样它们碰撞的机会就会大一些。

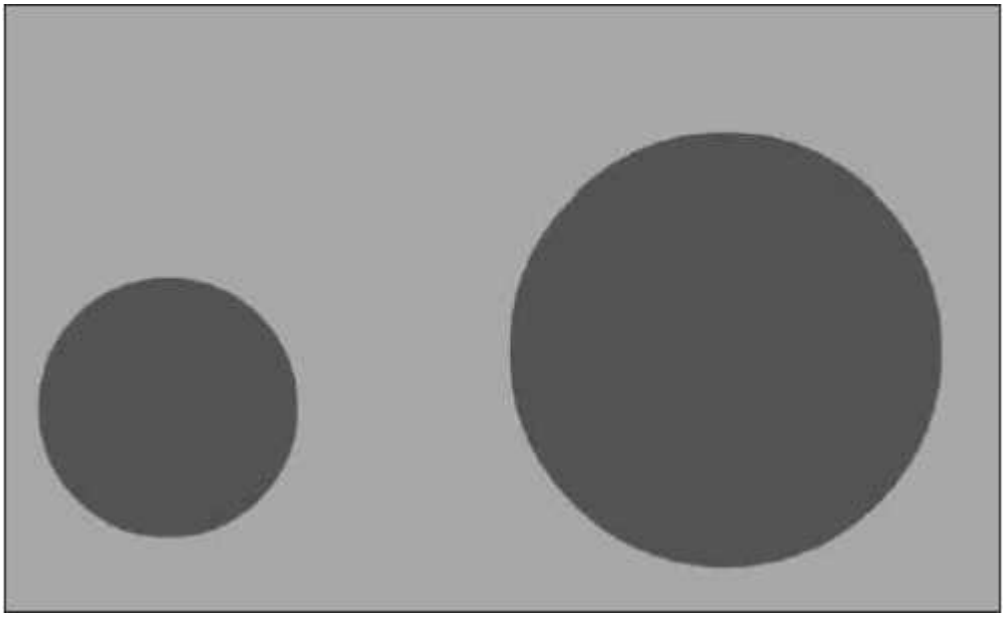


图 11-9 二维动量守恒，设置舞台

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Billiard3 extends Sprite {
  private var ball0:Ball;
  private var ball1:Ball;
  private var bounce:Number = -1.0;
  public function Billiard3() {
   init();
  }
  private function init():void {
   ball0 = new Ball(150);
   ball0.mass = 2;
   ball0.x = stage.stageWidth - 200;
   ball0.y = stage.stageHeight - 200;
   ball0.vx = Math.random() * 10 - 5;
   ball0.vy = Math.random() * 10 - 5;
   addChild(ball0);
   ball1 = new Ball(90);
   ball1.mass = 1;
   ball1.x = 100;
   ball1.y = 100;
   ball1.vx = Math.random() * 10 - 5;
   ball1.vy = Math.random() * 10 - 5;
   addChild(ball1);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   ball0.x += ball0.vx;
   ball0.y += ball0.vy;
   ball1.x += ball1.vx;
   ball1.y += ball1.vy;
   checkWalls(ball0);
   checkWalls(ball1);
  }
  private function checkWalls(ball:Ball):void {
   if (ball.x + ball.radius > stage.stageWidth) {
    ball.x = stage.stageWidth - ball.radius;
    ball.vx *= bounce;
   } else if (ball.x - ball.radius < 0) {
    ball.x = ball.radius;
    ball.vx *= bounce;
   }
   if (ball.y + ball.radius > stage.stageHeight) {
    ball.y = stage.stageHeight - ball.radius;
    ball.vy *= bounce;
   } else if (ball.y - ball.radius < 0) {
    ball.y = ball.radius;
```

```
    ball.vy *= bounce;
   }
  }
 }
}
```

这些内容想必大家睡梦中都可以写出来。设置边界，随机的速度，加入质量，根据速度移动小球，判断边界。注意，我把边界的判断单独放在了 checkWalls 函数中，以便重复使用。

同样，将碰撞判断放到名为 checkCollision 的函数中。onEnterFrame 变为：

```
private function onEnterFrame(event:Event):void {
 ball0.x += ball0.vx;
 ball0.y += ball0.vy;
 ball1.x += ball1.vx;
 ball1.y += ball1.vy;
 checkCollision(ball0, ball1);
 checkWalls(ball0);
 checkWalls(ball1);
}
```

这样一来，我只需要给大家介绍 checkCollision 函数以及与之相配套的函数就可以了。其它的代码没有变化，大家可以在 Billiard3.as 中看到完整的程序。

函数一开始非常简单，就是进行距离碰撞检测。

```
private function checkCollision(ball0:Ball, ball1:Ball):void {
 var dx:Number = ball1.x - ball0.x;
 var dy:Number = ball1.y - ball0.y;
 var dist:Number = Math.sqrt(dx*dx + dy*dy);
 if (dist < ball0.radius + ball1.radius) {
  // 处理碰撞的代码
 }
}
```

二分之三的代码已经写好了，目前为止都是小意思！首先判断碰撞需要知道两个 ball 之间的角度，使用 Math.atan2(dy, dx) 得到。（如果读到这里你还没有想到这点，请复习第三章三角学）然后，保存计算出的正余弦值，因为我们要反复使用。

```
// 计算角度和正余弦值
var angle:Number = Math.atan2(dy, dx);
var sin:Number = Math.sin(angle);
var cos:Number = Math.cos(angle);
```

下面，对速度和小球的位置进行坐标旋转。调用旋转后的位置 x0, y0, x1, y1 然后旋转 vx0, vy0, vx1, vy1。

因为我们使用 ball0 作为“中心点”，它的坐标就是 0,0。这个值在旋转后都不会改变，只要写：

```
// 旋转 ball0 的位置
var x0:Number = 0;
var y0:Number = 0;
```

接下来,ball1 的位置是与 ball0 的相对位置,与刚刚计算出来的距离值 dx 和 dy 相对应。因此，只需对这两个数进行旋转，就可以得到 ball1 旋转后的位置：

```
// 旋转 ball1 的位置
var x1:Number = dx * cos + dy * sin;
var y1:Number = dy * cos - dx * sin;
```

最后，旋转速度。写法如下：

```
// 旋转 ball0 的速度
var vx0:Number = ball0.vx * cos + ball0.vy * sin;
var vy0:Number = ball0.vy * cos - ball0.vx * sin;
// 旋转 ball1 的速度
var vx1:Number = ball1.vx * cos + ball1.vy * sin;
var vy1:Number = ball1.vy * cos - ball1.vx * sin;
```

所有的旋转代码如下所示：

```
private function checkCollision(ball0:Ball, ball1:Ball):void {
 var dx:Number = ball1.x - ball0.x;
 var dy:Number = ball1.y - ball0.y;
 var dist:Number = Math.sqrt(dx*dx + dy*dy);
 if (dist < ball0.radius + ball1.radius) {
  // 计算角度和正余弦值
  var angle:Number = Math.atan2(dy, dx);
  var sin:Number = Math.sin(angle);
  var cos:Number = Math.cos(angle);
  // 旋转 ball0 的位置
  var x0:Number = 0;
  var y0:Number = 0;
  // 旋转 ball1 的位置
  var x1:Number = dx * cos + dy * sin;
  var y1:Number = dy * cos - dx * sin;
  // 旋转 ball0 的速度
  var vx0:Number = ball0.vx * cos + ball0.vy * sin;
  var vy0:Number = ball0.vy * cos - ball0.vx * sin;
  // 旋转 ball1 的速度
  var vx1:Number = ball1.vx * cos + ball1.vy * sin;
  var vy1:Number = ball1.vy * cos - ball1.vx * sin;
 }
}
```

现在怎么样，不那么可怕了吧？先叫个暂停。我们已经完成了这个艰难历程的三分之一。

接下来只需要用 vx0，ball0.mass 和 vx1，ball1.mass 就可以执行一维碰撞。根据早先那个一维碰撞的例子可知：

```
var vxTotal:Number = ball0.vx - ball1.vx;
ball0.vx = ((ball0.mass - ball1.mass) * ball0.vx +
2 * ball1.mass * ball1.vx) /
(ball0.mass + ball1.mass);
ball1.vx = vxTotal + ball0.vx;
```

现在重写这段代码：

```
var vxTotal:Number = vx0 - vx1;
vx0 = ((ball0.mass - ball1.mass) * vx0 +
2 * ball1.mass * vx1) /
```

```
(ball0.mass + ball1.mass);
vx1 = vxTotal + vx0;
```

只需将 ball0.vx 和 ball1.vx 替换成了旋转后的版本 vx0 和 vx1。然后插入到函数中：

```
private function checkCollision(ball0:Ball, ball1:Ball):void {
 var dx:Number = ball1.x - ball0.x;
 var dy:Number = ball1.y - ball0.y;
 var dist:Number = Math.sqrt(dx*dx + dy*dy);
 if (dist < ball0.radius + ball1.radius) {
  // 计算角度和正余弦值
  var angle:Number = Math.atan2(dy, dx);
  var sin:Number = Math.sin(angle);
  var cos:Number = Math.cos(angle);
  // 旋转 ball0 的位置
  var x0:Number = 0;
  var y0:Number = 0;
  // 旋转 ball1 的位置
  var x1:Number = dx * cos + dy * sin;
  var y1:Number = dy * cos - dx * sin;
  // 旋转 ball0 的速度
  var vx0:Number = ball0.vx * cos + ball0.vy * sin;
  var vy0:Number = ball0.vy * cos - ball0.vx * sin;
  // 旋转 ball1 的位置
  var vx1:Number = ball1.vx * cos + ball1.vy * sin;
  var vy1:Number = ball1.vy * cos - ball1.vx * sin;
  // 碰撞的作用力
  var vxTotal:Number = vx0 - vx1;
  vx0 = ((ball0.mass - ball1.mass) * vx0 +
  2 * ball1.mass * vx1) /
  (ball0.mass + ball1.mass);
  vx1 = vxTotal + vx0;
  x0 += vx0;
  x1 += vx1;
 }
}
```

这段代码同样也把新的 x 速度加到 x 位置上，为了使小球分开，同一维碰撞的例子。

现在更新工作已完成，接下来将一切再反转回来：

```
// 将位置旋转回来
var x0Final:Number = x0 * cos - y0 * sin;
var y0Final:Number = y0 * cos + x0 * sin;
var x1Final:Number = x1 * cos - y1 * sin;
var y1Final:Number = y1 * cos + x1 * sin;
```

回忆一下旋转方程中 + 和 - 的调换，因此现在是向另一个方向旋转。最终的位置与系统中心点 ball0 的位置相对的。因此，需要把它们都加到 ball0 的位置上，从而得到实际在屏幕上的位置。先从 ball1 开始，因此就要用到 ball0 的初始位置，而不是更新后的位置：

```
// 将位置调整为屏幕的实际位置
```

```
ball1.x = ball0.x + x1Final;
ball1.y = ball0.y + y1Final;
ball0.x = ball0.x + x0Final;
ball0.y = ball0.y + y0Final;
```

最后，将速度旋转回来。可以直接使用 ball 的 vx 和 vy 属性：

```
// 将速度旋转回来
ball0.vx = vx0 * cos - vy0 * sin;
ball0.vy = vy0 * cos + vx0 * sin;
ball1.vx = vx1 * cos - vy1 * sin;
ball1.vy = vy1 * cos + vx1 * sin;
```

让我们看一下这段完整的函数：

```
function checkCollision(ball0:Ball, ball1:Ball):void {
 var dx:Number = ball1.x - ball0.x;
 var dy:Number = ball1.y - ball0.y;
 var dist:Number = Math.sqrt(dx*dx + dy*dy);
 if (dist < ball0.radius + ball1.radius) {
  // 计算角度和正余弦值
  var angle:Number = Math.atan2(dy, dx);
  var sin:Number = Math.sin(angle);
  var cos:Number = Math.cos(angle);
  // 旋转 ball0 的位置
  var x0:Number = 0;
  var y0:Number = 0;
  // 旋转 ball1 的位置
  var x1:Number = dx * cos + dy * sin;
  var y1:Number = dy * cos - dx * sin;
  // 旋转 ball0 的速度
  var vx0:Number = ball0.vx * cos + ball0.vy * sin;
  var vy0:Number = ball0.vy * cos - ball0.vx * sin;
  // 旋转 ball1 的速度
  var vx1:Number = ball1.vx * cos + ball1.vy * sin;
  var vy1:Number = ball1.vy * cos - ball1.vx * sin;
  // 碰撞的作用力
  var vxTotal:Number = vx0 - vx1;
  vx0 = ((ball0.mass - ball1.mass) * vx0 +
  2 * ball1.mass * vx1) /
  (ball0.mass + ball1.mass);
  vx1 = vxTotal + vx0;
  x0 += vx0;
  x1 += vx1;
  // 将位置旋转回来
  var x0Final:Number = x0 * cos - y0 * sin;
  var y0Final:Number = y0 * cos + x0 * sin;
  var x1Final:Number = x1 * cos - y1 * sin;
  var y1Final:Number = y1 * cos + x1 * sin;
  // 将位置调整为屏幕的实际位置
  ball1.x = ball0.x + x1Final;
```

```
  ball1.y = ball0.y + y1Final;
  ball0.x = ball0.x + x0Final;
  ball0.y = ball0.y + y0Final;
  // 将速度旋转回来
  ball0.vx = vx0 * cos - vy0 * sin;
  ball0.vy = vy0 * cos + vx0 * sin;
  ball1.vx = vx1 * cos - vy1 * sin;
  ball1.vy = vy1 * cos + vx1 * sin;
 }
}
```

实验一下这个例子。试改变 Ball 实例的大小，初始速度，质量等。

至于 checkCollision 函数，非常显眼。通过读注释，可以看出它实际上是被分做很多简单代码段的。我们还可以做优化，或再进行因式分解消除多余的重复内容。请培养这个良好的习惯，请见 Billiard4.as：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.geom.Point;
 public class Billiard4 extends Sprite {
  private var ball0:Ball;
  private var ball1:Ball;
  private var bounce:Number = -1.0;
  public function Billiard4() {
   init();
  }
  private function init():void {
   ball0 = new Ball(150);
   ball0.mass = 2;
   ball0.x = stage.stageWidth - 200;
   ball0.y = stage.stageHeight - 200;
   ball0.vx = Math.random() * 10 - 5;
   ball0.vy = Math.random() * 10 - 5;
   addChild(ball0);
   ball1 = new Ball(90);
   ball1.mass = 1;
   ball1.x = 100;
   ball1.y = 100;
   ball1.vx = Math.random() * 10 - 5;
   ball1.vy = Math.random() * 10 - 5;
   addChild(ball1);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   ball0.x += ball0.vx;
   ball0.y += ball0.vy;
   ball1.x += ball1.vx;
   ball1.y += ball1.vy;
   checkCollision(ball0, ball1);
   checkWalls(ball0);
```

```
 checkWalls(ball1);
}
private function checkWalls(ball:Ball):void {
 if (ball.x + ball.radius > stage.stageWidth) {
  ball.x = stage.stageWidth - ball.radius;
  ball.vx *= bounce;
 } else if (ball.x - ball.radius < 0) {
  ball.x = ball.radius;
  ball.vx *= bounce;
 }
 if (ball.y + ball.radius > stage.stageHeight) {
  ball.y = stage.stageHeight - ball.radius;
  ball.vy *= bounce;
 } else if (ball.y - ball.radius < 0) {
  ball.y = ball.radius;
  ball.vy *= bounce;
 }
}
private function checkCollision(ball0:Ball, ball1:Ball):void {
 var dx:Number = ball1.x - ball0.x;
 var dy:Number = ball1.y - ball0.y;
 var dist:Number = Math.sqrt(dx*dx + dy*dy);
 if (dist < ball0.radius + ball1.radius) {
  // 计算角度和正余弦值
  var angle:Number = Math.atan2(dy, dx);
  var sin:Number = Math.sin(angle);
  var cos:Number = Math.cos(angle);
  // 旋转 ball0 的位置
  var pos0:Point = new Point(0, 0);
  // 旋转 ball1 的速度
  var pos1:Point = rotate(dx, dy, sin, cos, true);
  // 旋转 ball0 的速度
  var vel0:Point = rotate(ball0.vx, ball0.vy,
  sin, cos, true);
  // 旋转 ball1 的速度
  var vel1:Point = rotate(ball1.vx, ball1.vy,
  sin, cos, true);
  // 碰撞的作用力
  var vxTotal:Number = vel0.x - vel1.x;
  vel0.x = ((ball0.mass - ball1.mass) * vel0.x +
  2 * ball1.mass * vel1.x) /
  (ball0.mass + ball1.mass);
  vel1.x = vxTotal + vel0.x;
  // 更新位置
  pos0.x += vel0.x;
  pos1.x += vel1.x;
  // 将位置旋转回来
  var pos0F:Object = rotate(pos0.x, pos0.y,
  sin, cos, false);
```

```
    var pos1F:Object = rotate(pos1.x, pos1.y,
    sin, cos, false);
    // 将位置调整为屏幕的实际位置
    ball1.x = ball0.x + pos1F.x;
    ball1.y = ball0.y + pos1F.y;
    ball0.x = ball0.x + pos0F.x;
    ball0.y = ball0.y + pos0F.y;
    // 将速度旋转回来
    var vel0F:Object = rotate(vel0.x, vel0.y,
    sin, cos, false);
    var vel1F:Object = rotate(vel1.x, vel1.y,
    sin, cos, false);
    ball0.vx = vel0F.x;
    ball0.vy = vel0F.y;
    ball1.vx = vel1F.x;
    ball1.vy = vel1F.y;
   }
  }
  private function rotate(x:Number, y:Number,
  sin:Number, cos:Number, reverse:Boolean):Point {
   var result:Point = new Point();
   if (reverse) {
    result.x = x * cos + y * sin;
    result.y = y * cos - x * sin;
   } else {
    result.x = x * cos - y * sin;
    result.y = y * cos + x * sin;
   }
   return result;
  }
 }
}
```

这里我设计了一个用作旋转的函数，rotate，传入所需的参数值，返回一个 flash.geom.Point 实例。这个对象已经定义好了 x 和 y 属性（还有许多其他的这里用不到的属性），返回的旋转后的 Point 的 x,y 属性。虽然这个版本并不是很好读，但是可以省去很多重复的代码。

加入更多的物体

让两个影片碰撞并带有反作用力不是件容易的事，但我们已经做到了。恭喜各位。下面要让多个物体进行碰撞——比如说八个。听起来要复杂四倍，其实不然。之前的函数每次要判断两个小球，而这并不是我们真正想要的。将多个物体放到舞台上，让它们运动，判断碰撞，这是我们在碰撞检测的例子中（第九章）作过。现在所要作的就是把这些代码插入到 checkCollision 函数中的碰撞检测中去。

例子程序（MultiBilliard.as），开始将八个小球存入数组，循环执行，为它们设置不同的属性：

```
package {
```

```
import flash.display.Sprite;
import flash.events.Event;
import flash.geom.Point;
public class MultiBilliard extends Sprite {
 private var balls:Array;
 private var numBalls:uint = 8;
 private var bounce:Number = -1.0;
 public function MultiBilliard() {
  init();
 }
 private function init():void {
  balls = new Array();
  for (var i:uint = 0; i < numBalls; i++) {
   var radius:Number = Math.random() * 20 + 20;
   var ball:Ball = new Ball(radius);
   ball.mass = radius;
   ball.x = i * 100;
   ball.y = i * 50;
   ball.vx = Math.random() * 10 - 5;
   ball.vy = Math.random() * 10 - 5;
   addChild(ball);
   balls.push(ball);
  }
  addEventListener(Event.ENTER_FRAME, onEnterFrame);
 }
 private function onEnterFrame(event:Event):void {
  // 稍后给出… ...
 }
 // checkWalls, checkCollision, rotate 函数与上一个例子相同
}
}
```

大家也许注意到了，我将每个小球的初始位置人为地进行了设置，为的是不让它们一开始就发生接触。如果那样的话，它们就会粘在一起。

onEnterFrame 方法惊人的简单。只需要做两次循环：一次是基本运动，一次是碰撞检测。

```
private function onEnterFrame(event:Event):void {
 for (var i:uint = 0; i < numBalls; i++) {
  var ball:Ball = balls[i];
  ball.x += ball.vx;
  ball.y += ball.vy;
  checkWalls(ball);
 }
 for (i = 0; i < numBalls - 1; i++) {
  var ballA:Ball = balls[i];
  for (var j:Number = i + 1; j < numBalls; j++) {
   var ballB:Ball = balls[j];
   checkCollision(ballA, ballB);
  }
 }
```

```
}
```

第一次循环，遍历所有舞台上的小球，让它们运动并在撞墙后反弹。接下来，在一个嵌套循环中让每个小球与其它小球进行比较，就像第九章碰撞检测中讨论的那样。获得两个小球的引用，分别叫作 ballA 和 ballB，将它们传入 checkCollision 函数。这样就可以了，checkWalls, checkCollision, rotate 函数与上一个例子完全相同，这里不再敷述。

要加入更多的小球，只需改变 numBalls 变量即可，并确保它们最初不会发生碰撞。

解决潜在问题

前面提醒过：两个物体之间仍有可能在某些情况下粘在一起。最有可能发生在影片非常拥挤的条件下，并且在运动速度很快时结果会更糟糕。我们有可能看到两三个小球在舞台的边角上发生碰撞。

假设舞台上有三个小球 —— ball0, ball1, ball2 —— 它们恰好要碰撞在一起。下面是发生的基本情况：

- 依照物体的速度移动物体。
- 先判断 ball0 与 ball1，ball0 与 ball2，发现没有产生碰撞。
- 判断 ball1 与 ball2。发现它俩发生了碰撞，然后计算出所有新的速度及位置，为的是让它们不会接触到一起。但却不小心让 ball1 与 ball0 接触上了。然而，这一组判断已经执行过了，因此这次就被忽略了。
- 在下一次循环中，代码继续依照物体的速度移动物体。但是无意中把 ball0 和 ball1 移动得更近了。
- 现在代码注意到了 ball0 与 ball1 发生了碰撞。然后计算出新的速度并加到物体的当前位置上，让它们分离。但是，因为它们已经接触上了，不能将它们真正地分开。于是又粘到了一起。

这种情况最容易发生在空间很小物体很多，移动速度很快的情况下。也发生在，物体间一开始就产生接触的情况下。可能迟早我们都会遇到这种情况，所以最好来看看问题出在哪。确切的位置是在 checkCollision 函数中，定义的这两条语句：

```
// 更新位置
pos0.x += vel0.x;
pos1.x += vel1.x;
```

这里我们只是假设产生的碰撞是由两个小球的速度引起的，再给它们加入的新的速度，就会使它们分开。多数情况下，这是可以的。但是我刚才描述的情况除外。如果那样的话，就要明确地知道影片在运动前是分离的。于是想出了如下方法：

```
// 更新位置
var absV:Number = Math.abs(vel0.x) + Math.abs(vel1.x);
var overlap:Number = (ball0.radius + ball1.radius) - Math.abs(pos0.x - pos1.x);
pos0.x += vel0.x / absV * overlap;
pos1.x += vel1.x / absV * overlap;
```

这些都是我自己创造的，所以我能不确定它的精确度如何，但是看上去工作得还不错。首先确定绝对速度（自创的一个词），是所有速度的绝对值之和。例如，如果一个速度是 -5 另一个速度是 10，那么绝对值就是 5 和 10，总和就是 5 + 10 = 15。

接下来，确定小球之间重叠部分的大小。用总半径长度减去总距离。

然后根据小球速度与绝对速度的百分比，让小球移动出重叠的那一部分。

结果会让小球之间没有重叠。这种方法比早前的版本要复杂一些，但确消除了很多 bug。

在 MultiBilliard2.as 中，创建了 20 个小球，并让它们的体积大一些，在舞台上随机分布。也许在几帧之内，小球之间可能发生重叠，但是由于新加入这些代码的作用，让它

们平静了下来。

当然，您也可以去研究自己的解决方案，如果您找到了更简单，更有效，更精确的方法，请拿出来一起分享！

本章重要公式

本章重要公式只有一个动量守恒定理。

动量守恒的数学表达式：

$$v0Final = \frac{(m0 - m1) * v0 + 2 * m1 * v1}{m0 + m1}$$

$$v1Final = \frac{(m1 - m0) * v1 + 2 * m0 * v0}{m0 + m1}$$

动量守恒的 ActionScript 表达式：

```
var vxTotal:Number = vx0 - vx1;
vx0 = ((ball0.mass - ball1.mass) * vx0 +
2 * ball1.mass * vx1) /
(ball0.mass + ball1.mass);
vx1 = vxTotal + vx0;
```

11.5 小结

恭喜！你已经通过了本书中可能最难的数学问题。你已经掌握了控制准确的碰撞反应的方法。这也是人们细心经常问我的问题之一。趁着你现在有兴趣，可以制作一个相当不错的台球游戏，并且是完全符合台球物理的。为了保持简单，在这些示例中我忽略了一件事是摩擦力概念。你可能想试着将它们加到系统中。你现在已经有足够的能力来做这件事。你也许像先看一下第 19 章，那里我给出了处理两个质量相同的物体时使用的一些技巧。

在下一章中，我将把事情缓和一下开始探讨粒子引力，虽然例子中加入了一些台球物理现象，但它看上去是很合适的。

第十二章　粒子引力与万有引力

很高兴到晋级到了这一章。前面的章节总体来说都是些交互的运动。先让物体运动起来，再让物体与环境产生交互，随后与用户交互，最后是物体之间的交互。本章将详细地为大家介绍物体之间的交互，并带有一定距离的。说具体些，我们将学习粒子，重力（与前面有些不同），弹性运动（还是它！）以及大名鼎鼎的 Node Garden。让我们开始吧！

粒子（Particles）

说明一下我们所指的粒子是什么意思。出于本章的目的，粒子只就是一个独立的单位，通常会伴有着几个（或多个）同样的单位。一个粒子可以说是一颗灰尘，一个沙滩球，或是一个星球。

粒子都具有共同的行为，也可以有它们各子独立的行为。OOP 的程序设计员们通常把它们看成与对象相似的东西。某个类的所有对象都具有相同的行为，它们是由这个类定义的，而构造出的实例由于被赋予了不同的属性值，从而变得有所差异。我们已经在例子中看过许多 Ball 类的这样的实例。每个对象都有自己的属性：速度，质量，大小，颜色等等，但是所有的小球都以相同的规则运动。

在 Ball 类中，包涵了我们所需要的所有的功能。同样，粒子的实例也只是用来保存属性的。文档类只负责移动粒子及处理它们之间的交互。另一种方法是把行为代码放到粒子类中，这样每个粒子对象都会有它们自己的 enterFrame 函数或定时动画，它们将自行负责运动或处理交互运动。执行处理函数时，它们都会有各自的加减运算。本书的例子程序，都将运动和交互放在了文档类中，为的是看起来简单一些。

程序总体来说与前面例子相同，变化的部分都在两个粒子间的交互和引力上，这些代码都写在 onEnterFrame 方法中出现。程序中我们要创建多个粒子并随机放置到屏幕上，代码如下：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class ClassName extends Sprite {
  private var particles:Array;
  private var numParticles:uint = 30;
  public function ClassName() {
   init();
  }
  private function init():void {
   particles = new Array();
   for (var i:uint = 0; i < numParticles; i++) {
    var particle:Ball = new Ball(5);
    particle.x = Math.random() * stage.stageWidth;
    particle.y = Math.random() * stage.stageHeight;
    particle.mass = 1;
    addChild(particle);
    particles.push(particle);
   }
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
```

```
    }
  }
}
```

这里，每个粒子初始的半径都是 5 设置，质量均为 1。稍后通过改变这些值我们可以看到不同的效果，当然还可以给粒子随机的速度和大小。

本章的程序都是假设这些基本的设置已经完成，因为我们主要是来介绍 onEnterFrame 函数及其所需的方法。下面，先来介绍一些基本的理论，刚才创建的这个文件先不要关，因为马上还要用到。

重力

第一种粒子引力，就是重力。大家可能会说“这不是又回到的第二部分了吗？”是的，但那是最常见的引力形式。

站在地球上，引力的定义很简单：将物体向下牵引。实事上，向下的牵引力有着一种特殊的速率。它所施加在物体上的加速度等于 32 英尺每秒。从加速度的角度来讲就是某一时刻内累加了多少速度。重力让物体运动 32 英尺，并且每秒都向下拉。但是，我们可以到最高的山上，或者最低的峡谷，这些对于 32 这个数字所产生的细微的变化是不足以让人们觉察到的（除非使用精密的仪器）。

万有引力

我们知道，距离某个星球或某个巨大的天体越远，那么所受的引力就越小。对我们来说这是件好事。如不然的话，我们会被吸到太阳里面，而太阳也许还会被吸到其它太阳或星球中进去，所有的一切很快就灰飞烟灭了。我们把星球看作是粒子，粒子间的距离对于引力的大小起决定性的作用。距离对于引力的影响很简单：引力与距离的平方成反比。需要解释一下。首先，引力也与质量联系紧密。物体质量越大，那么该物体对于其它物体的引力就越大，那么它所受到其它物体的反作用力就越大。这就是所谓的万有引力常数（简写 G）。引力方程如下：

```
force = G * m1 * m2 / distance2
```

简单来说，其它物体对该个物体的引力等于万有引力常数乘以二者的质量，再除以它们之间距离的平方。呃… … 看上去很简单，只需要知道引力常数，就 OK 了。下面是正式的定义：

```
G = (6.6742 ± 0.0010) * 10-11 * m3 * kg-1 * s-2
```

如果您想用 ActionScript 来计算结果的话，可以试一下。就我个人而言，当我看到这样的公式，立刻就想到要伪造这些数字。我的引力公式如下：

```
force = m1 * m2 / distance2
```

是的，就是这样，忽略 G。如果要用 Flash 为 NASA [国家航空局] 作飞船导航系统的话，就要把 G 算进去。但是如果我们是做宇宙大战游戏的话，那么就应该舍弃它。

现在，公式已经有了，来写代码吧。首先设置 onEnterFrame 函数，而它又要调用 gravitate 函数，所以要把负责引力的代码分离出来：

```
private function onEnterFrame(event:Event):void {
 for (var i:uint = 0; i < numParticles; i++) {
  var particle:Ball = particles[i];
  particle.x += particle.vx;
  particle.y += particle.vy;
 }
 for (i=0; i < numParticles - 1; i++) {
  var partA:Ball = particles[i];
  for (var j:uint = i + 1; j < numParticles; j++) {
   var partB:Ball = particles[j];
   gravitate(partA, partB);
  }
 }
}
```

开始用一个单独的循环让粒子运动，然后做一个双重循环执行交互。获得 partA 和 partB 后就将这两个粒子对象传给 gravitate 函数：

```
private function gravitate(partA:Ball, partB:Ball):void {
 var dx:Number = partB.x - partA.x;
 var dy:Number = partB.y - partA.y;
 var distSQ:Number = dx * dx + dy * dy;
 var dist:Number = Math.sqrt(distSQ);
 var force:Number = partA.mass * partB.mass / distSQ;
 var ax:Number = force * dx / dist;
 var ay:Number = force * dy / dist;
 partA.vx += ax / partA.mass;
 partA.vy += ay / partA.mass;
 partB.vx -= ax / partB.mass;
 partB.vy -= ay / partB.mass;
}
```

首先，找出两个粒子间的 dx 和 dy 以及总距离。记住这个公式—— F = G * m1 * m2 / distance2 ——其中包括有距离的平方。在计算距离时，我们通常直接写成 dist = Math.sqrt(dx*dx + dy*dy)。但是在求距离的平方时，还要为这个平方根求平方！两倍的工作量。如果用变量 distSQ 在开平方前保存 dx*dx + dy*dy，就节省了一部分计算。

接下来，用总的引力乘以质量再除以距离的平方。就求出了 x,y 轴上总的加速度。然后，使用第九章最后讨论的那个简洁的计算方法，用 dx / dist 取代 Math.cos(angle)，用 dy / dist 取代 Math.sin(angle)。这样一来，就无需再使用 Math.atan2(dy, dx) 求出角度了。

下面，我们继续讨论总引力和总加速度的问题。这是于两个物体间的引力构成的。根据物体的质量，需要把它们分解为两个部分。想一想地球和太阳，两者之间存在着一种特殊的引力，是由质量除以距离的平方产生的。地球与太阳在总引力的作用力下相互吸引。地球被太阳吸引，而太阳反过来又被地球吸引。很明显，地球所获的加速度更多，因为它的质量比太阳小。所以，在系统中每个物体都有单独的加速度，用总加速度除以物体的质量。因此，就有了最后四行公式。注意 partA 的加速度是增加的，而 partB 则是减少的。这要取决于 dx 和 dy 相减的顺序。

最终的代码可在 Gravity.as 中找到。测试一下，可以看到开始时粒子是不动的，然后

慢慢地开始相互吸引，如图 12-1 所示。有时，两个粒子间会相互绕圈。不过通常都是碰撞后向相反的方向飞去。

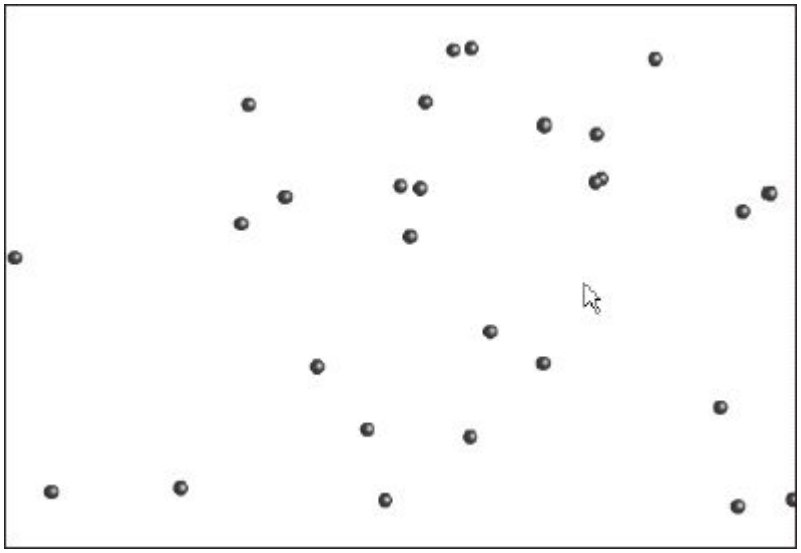


图 12-1 粒子来了！

这种超速运动是 bug 吗？并不是代码的 bug。这其实正是期望的效果，这就是所谓的弹弓效应。这是 NASA 用来发射探测器到外太空的方法。当一个物体与某个星球越来越近时，它所受的加速度会越来越大，物体的运动速度也会非常之快。如果当物体与某个星球非常接近时，该物体会受到这个星球的引力作用，被吸引过去。而这些速度是零燃料的——因此它功不可没。

回到 Flash。两个物体间的距离非常小时会发生什么——在几乎零距离的情况下。这时，两个物体间的引力变得非常大，几乎是无穷大。因此，计算的结果是正确，但是从模拟的观点来看，这并不真实。应该在物体足够接近时产生碰撞。如果我们计划让太空探测器直接落到某个星球上，那么它的速度就不能是无穷大了，这样的话可能会撞出一个弹坑。

碰撞检测及反作用

对粒子而言，需要进行碰撞判断以及产生反作用力。到底让碰撞后发生什么事情，取决于我们自己，可以让粒子爆炸后消失，也可以让一个粒子消失，并把它的质量加到另一个粒子上面，就像两个粒子溶合到了一起。

例如，前一章在 checkCollision 函数中，讲到的碰撞及反作用力。下面把它插入进来，内容如下：

```
function onEnterFrame(event:Event):void {
 for (var i:uint = 0; i < numParticles; i++) {
  var particle:Ball = particles[i];
  particle.x += particle.vx;
  particle.y += particle.vy;
 }
 for (i=0; i < numParticles - 1; i++) {
  var partA:Ball = particles[i];
  for (var j:uint = i + 1; j < numParticles; j++) {
   var partB:Ball = particles[j];
   checkCollision(partA, partB);
   gravitate(partA, partB);
  }
 }
```

```
}
```

粗体部分是新加入的。同时要把 checkCollision 及 rotate 函数复制粘贴到了这个文件中。（不要忘记导入 flash.geom.Point 函数，因为这些函数中需要使用它），全部代码请见 GravityBounce.as。

我们看到粒子间相互吸引，在产生碰撞后反弹。试改变粒子的质量(mass)，观察不同的引力效果。如果给粒子一个负质量，就可以看到它们彼此互斥。

在 GravityRandom.as，文件中，我坚持让代码不会有太大的变化，只是在 init 方法中多加入了一行语句，并在两个地方做了一下改变：

```
private function init():void {
 particles = new Array();
 for (var i:uint = 0; i < numParticles; i++) {
  var size:Number = Math.random() * 25 + 5;
  var particle:Ball = new Ball(size);
  particle.x = Math.random() * stage.stageWidth;
  particle.y = Math.random() * stage.stageHeight;
  particle.mass = size;
  addChild(particle);
  particles.push(particle);
 }
 addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

这里我们给每个粒子一个随机的大小并且根据物体的大小给出质量，如图 12-2 所示。课题开始变得越来越有趣了。

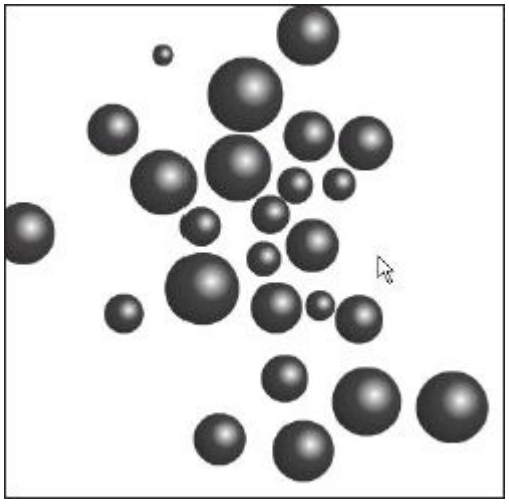


图 12-2 行星碰撞？

轨道运动

最后看一个现实中的例子，我们来创建一个简单的行星系统，有太阳和地球。创建一个质量为 10,000 的太阳和一个质量为 1 的行星。接下来，让行星移开太阳一段距离，并给它一个垂直于太阳的速度。如图 12-3 所示。

图 12-3 设置舞台

如果给出的质量，距离与速度都非常合适，那么就能让行星进入轨道。见文档类

Orbit.as。需要解释一下 init 中的代码，还有一点变化就是将 numParticles 变量设为 2。

```
private function init():void {
 particles = new Array();
 var sun:Ball = new Ball(100, 0xffff00);
 sun.x = stage.stageWidth / 2;
 sun.y = stage.stageHeight / 2;
 sun.mass = 10000;
 addChild(sun);
 particles.push(sun);
 var planet:Ball = new Ball(10, 0x00ff00);
 planet.x = stage.stageWidth / 2 + 200;
 planet.y = stage.stageHeight / 2;
 planet.vy = 7;
 planet.mass = 1;
 addChild(planet);
 particles.push(planet);
 addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

再给大家一个例子，见 OrbitDraw.as，绘出一条轨道线，可以得到一些有趣的图案。

弹性运动

大家也许想用弹性运动作为另一种粒子引力。是的，弹性运动是我的最爱！回忆一下第八章做过的弹簧链及物体间弹性运动的例子。后面我们会看到更为广泛的应用，粒子间相互地进行弹性运动，就像万有引力中的那些程序。

灵感来源于 Jared Tarbell 在 www.levitated.net 中创作的 Node Garden [节点花园]，如图 12-4 所示。总体思想是用一块区域包含某些节点（粒子），每个节点都会与相邻的节点发生交互，产生反作用力。这就是我的想法，用弹性运动作为反作用力。

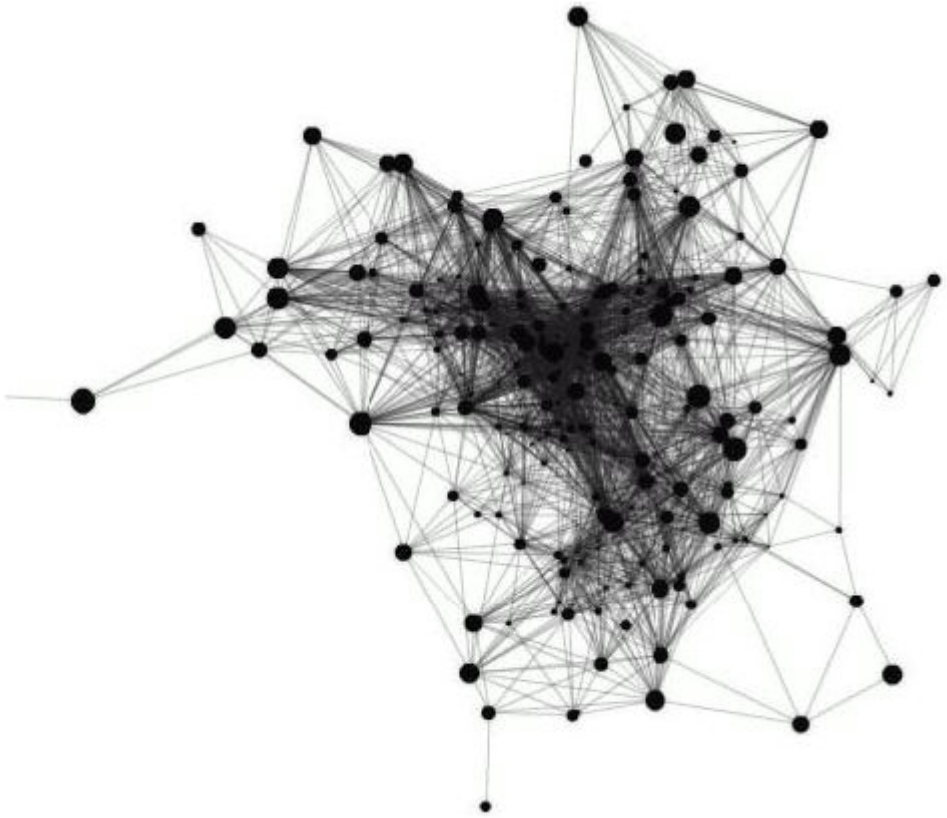


图 12-4 Jared Tarbell 的 Node Garden

引力 VS 弹性

对比一下引力与弹性，我们发现它们非常相似，或者说它们是皆然相反的。因为，它们都是用物体间的加速度，将两个物体牵在一起的。但是，在引力效果中，物体间的距离越远，则加速度越小。在弹性效果中，则是距离越大，加速度越大。

如果将前一个例子中的引力代码换成弹性代码的话，产生的效果就没那么有趣了，因为粒子最终会聚集在一起，因为弹性运动不能容忍距离。如果引力的座右铭是“眼不见为净”，那么弹性运动的信条就是“距离产生美”。

因此现在就陷入了一个进退两难的局面。我们想让粒子以弹性的形式相互吸引，还想让弹性能够容忍距离，不让粒子都牵连在一起。那么我的解决办法就是，设置一个最短距离的变量。严格来讲，这更像是最大距离，因为它表示能够产生交互的最大距离。但时我大脑总认为最短距离是正确的，是因为粒子间至少要达到这个距离才会产生交互。如果它们彼此距离很远，则忽略不计。

弹性节点花园

让我们来动手搭建自己的弹性节点花园吧。为了与本书的上一版相同，我将影片的背景色改为黑色，让粒子为白色。看起来不如用渐变填充代码或嵌入外部图像效果好，不过，这些是我们今后肯定要做的。将来我将给大家一下有趣的图形材料。

可在 NodeGarden.as 中找到代码，让我们一步步来。首先，设置一些变量：粒子的数目，刚刚提到的最短距离，以及弹性(spring)。

```
var numParticles:uint = 30;
var minDist:Number = 100;
var springAmount:Number = .001;
```

第八章的弹性运动示例中，我们设置的弹性值为 0.2 左右。这里我们使用的值要更小些，因为有很多的粒子会产生交互运动。如果给的值过大，则会使速度过快。然而，如果给的值太小，粒子就会在屏幕上慢条斯理地运动，就像是没有觉察到其它节点一样。

接下来进行初始化：

```
private function init():void {
 particles = new Array();
 for (var i:uint = 0; i < numParticles; i++) {
  var particle:Ball = new Ball(5, 0xffffff);
  particle.x = Math.random() * stage.stageWidth;
  particle.y = Math.random() * stage.stageHeight;
  particle.vx = Math.random() * 6 - 3;
  particle.vy = Math.random() * 6 - 3;
  addChild(particle);
  particles.push(particle);
 }
 addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

创建一些粒子，放在舞台上，给予它们随机的速度。注意，这里没有给出物体的质量（mass）。稍后，在“带有质量的节点”一节，会给大家演示加入质量的例子。

再下面，就是 onEnterFrame 方法：

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

```
 for (var i:uint = 0; i < numParticles; i++) {
  var particle:Ball = particles[i];
  particle.x += particle.vx;
  particle.y += particle.vy;
  if (particle.x > stage.stageWidth) {
   particle.x = 0;
  } else if (particle.x < 0) {
   particle.x = stage.stageWidth;
  }
  if (particle.y > stage.stageHeight) {
   particle.y = 0;
  } else if (particle.y < 0) {
   particle.y = stage.stageHeight;
  }
 }
 for (i=0; i < numParticles - 1; i++) {
  var partA:Ball = particles[i];
  for (var j:uint = i + 1; j < numParticles; j++) {
   var partB:Ball = particles[j];
   spring(partA, partB);
  }
 }
}
```

很熟悉了吧。以前讲过这个例子，只是在屏幕环绕的代码中 spring 了函数。最后是本程序的主干 spring 函数如下：

```
private function spring(partA:Ball, partB:Ball):void {
 var dx:Number = partB.x - partA.x;
 var dy:Number = partB.y - partA.y;
 var dist:Number = Math.sqrt(dx * dx + dy * dy);
 if (dist < minDist) {
  var ax:Number = dx * springAmount;
  var ay:Number = dy * springAmount;
  partA.vx += ax;
  partA.vy += ay;
  partB.vx -= ax;
  partB.vy -= ay;
 }
}
```

首先求出两个粒子间的距离。如果不小于 minDist [最短距离]，则继续向下执行。如果小于则根据 springValue 求出每个轴的加速度，然后将它加入 partA 的速度，再将它从 partB 的速度中减去。这就会将粒子拉近。

如图 12-5 所示，注意观察粒子是如何汇聚成团的，有点像苍蝇在乱飞。但它们也会聚在一起，散开，加入其它的团中等等，做出一些有趣的随机行为。试改变 minDist 及 springValue 的值，观察运行效果。

图 12-5 运动中的节点

带连线的节点

从标题可以看出，这就是为了能够表现每对节点间的交互。还有比绘制节点间的连线更好的方法吗？实现起来也非常简单。只对 spring 函数做一点改变：

```
private function spring(partA:Ball, partB:Ball):void {
 var dx:Number = partB.x - partA.x;
 var dy:Number = partB.y - partA.y;
 var dist:Number = Math.sqrt(dx * dx + dy * dy);
 if (dist < minDist) {
  graphics.lineStyle(1, 0xffffff);
  graphics.moveTo(partA.x, partA.y);
  graphics.lineTo(partB.x, partB.y);
  var ax:Number = dx * springAmount;
  var ay:Number = dy * springAmount;
  partA.vx += ax;
  partA.vy += ay;
  partB.vx -= ax;
  partB.vy -= ay;
 }
}
```

如果两个产生交互，则该函数设置将一个线条样式并在两点间进行连线。我们还应该将下述语句加入到 onEnterFrame 方法的第一行，用以在每帧时进行刷新：

```
graphics.clear();
```

现在节点间已经连接起来了，如图 12-6 所示。但是我不喜欢这种随节点的变化突然连线或断开的效果。

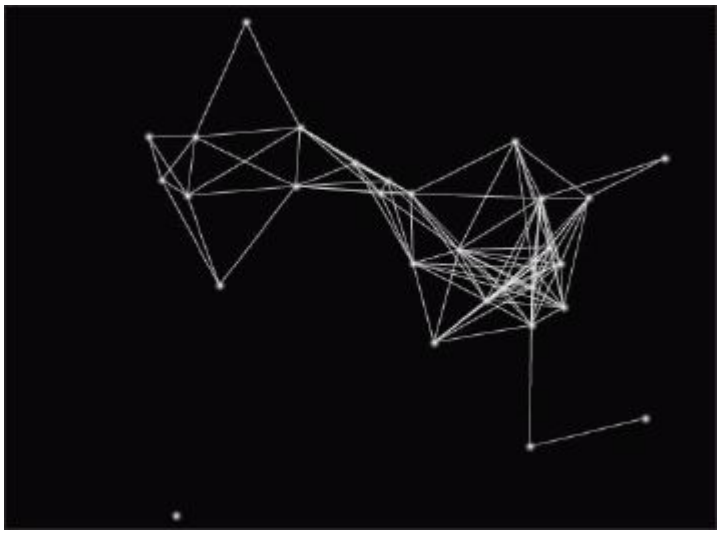


图 12-6 连接节点

我想要一种梯度式的变化。意思是如果两个节点距离大于 minDist，则线条应当几乎是完全透明的。随着它们的距离越来越近，线条会变得越来越亮。因此，如果用 dist / minDist，就会得到一个从 0 到 1 的分数作为 alpha。但这种效果与我们的设想是反向的，因为如果 dist 等于 minDist，alpha 将等于 1，但是当 dist 接近 0 是，alpha 将接近 0。OK，我们可以 1 减去这个数，那么这个数值就反过来了。以下是画线代码：

```
graphics.lineStyle(1, 0xffffff, 1 - dist / minDist);
graphics.moveTo(partA.x, partA.y);
graphics.lineTo(partB.x, partB.y);
```

在我看来，这是一种非常漂亮的效果，如图 12-7。全部代码见 NodeGardenLines.as。

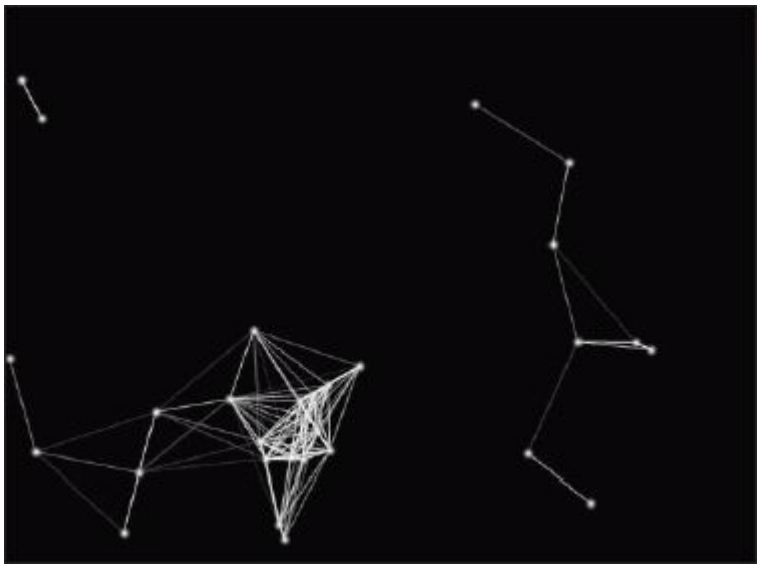


图 12-7 细微的变换，带来不凡的效果

带有质量的节点

在撰写本章时，我就密谋要让每个节点都有其质量，这是我以前没有想到的。于是就写出了 NodesMass.as。在这个文件中给每个节点一个随机的大小，并且根据大小给定质量。

```
private function init():void {
 particles = new Array();
 for (var i:uint = 0; i < numParticles; i++) {
  var size:Number = Math.random() * 10 + 2;
```

```
  var particle:Ball = new Ball(size, 0xffffff);
  particle.x = Math.random() * stage.stageWidth;
  particle.y = Math.random() * stage.stageHeight;
  particle.vx = Math.random() * 6 - 3;
  particle.vy = Math.random() * 6 - 3;
  particle.mass = size;
  addChild(particle);
  particles.push(particle);
 }
 addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

质量仅仅用于在 spring 函数中为物体添加速度。我们用每个粒子的速度除以质量，给较大的物体较大的惯性。

```
private function spring(partA:Ball, partB:Ball):void {
 var dx:Number = partB.x - partA.x;
 var dy:Number = partB.y - partA.y;
 var dist:Number = Math.sqrt(dx * dx + dy * dy);
 if (dist < minDist) {
  graphics.lineStyle(1, 0xffffff, 1 - dist / minDist);
  graphics.moveTo(partA.x, partA.y);
  graphics.lineTo(partB.x, partB.y);
  var ax:Number = dx * springAmount;
  var ay:Number = dy * springAmount;
  partA.vx += ax / partA.mass;
  partA.vy += ay / partA.mass;
  partB.vx -= ax / partB.mass;
  partB.vy -= ay / partB.mass;
 }
}
```

由于我削减了部分弹性运动的效果，因此将 springValue 的值增加到 .0025。我喜欢这种完善的效果，如图 12-8 所示。

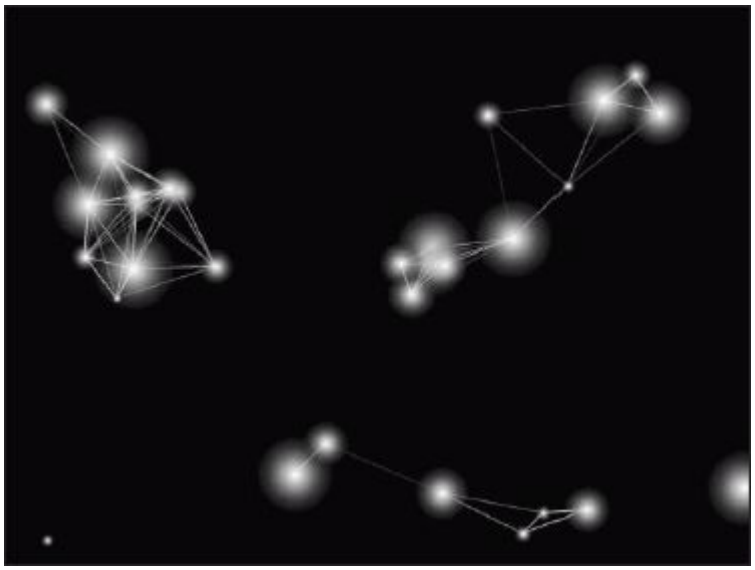


图 12-8 再进一步

节点能做什么用呢？在我看来，这种效果看上去酷毙了，我曾用它作为屏幕保护程序。这种效果可以作为所有游戏场景的序幕，放入一个行星飞船在上面，让飞船躲避这些节点。我想这是个很好的挑战！

本章重要公式

显然本章最大的公式就是万有引力公式。

引力的一般公式：

$$force = G * m1 * m2 / distance^2$$

ActionScript 实现万有引力:

```
function gravitate(partA:Ball, partB:Ball):void
{
    var dx:Number = partB.x - partA.x;
    var dy:Number = partB.y - partA.y;
    var distSQ:Number = dx * dx + dy * dy;
    var dist:Number = Math.sqrt(distSQ);
    var force:Number = partA.mass * partB.mass / distSQ;
    var ax:Number = force * dx / dist;
    var ay:Number = force * dy / dist;
    partA.vx += ax / partA.mass;
    partA.vy += ay / partA.mass;
    partB.vx -= ax / partB.mass;
    partB.vy -= ay / partB.mass;
}
```

函数中，使用的是 Ball 类型，大家可以根据实际需要使用自定义类的类型，只要能够存储速度，质量和位置即可。

12.5 小结

这章涉及了处于某个距离的粒子之间的交互和如何使用引力和弹性来制作有趣的效果。因此，你有两中新的方法来制作许物体的动态运动效果。

在下面的两章中，我将讲解非常新的课题：正向运动和反向运动。这些技术可以让你制作类似机器人的手臂和行走等非常酷的效果。

第十三章　正向运动学：行走

前面章节介绍的都是 ActionScript 交互动画的基础，也可以说是一些高级“基础”。现在开始，我们进入另一条有趣的技术之路，运动学。

到底什么是运动学呢？我所找到的一些资料看起来都有些让人望而却步，这是一项基于高级 3D 动画编程的技术。上网搜索一下，会发现其涉及到的方程中到处都是些陌生符号，这也成为了我们学习的最大障碍，似乎前面所学的内容都像是很基础的算法。首先，我要说，运动学并没有那么可怕。前面章节中只介绍了我们所需的一些基本知识，现在就要将它们加以组合。

运动学研究物体的运动，但不考虑施加在物体上的质量或力，基本上是数学的一个分支。所以，无非就是速度，方向，速度向量等等。听起来不难哈？虽然这是个非常简单的定义，但是蕴藏在里面的内容是比较复杂的，但是要完成我们的目标已经足够了。

当人们在计算机科学，图形及游戏等领域谈论运动学的时候，无非就是在讨论运动学的两个特殊的分支：正向运动学和反向运动学。让我们就从这里开始吧。

正向和反向运动学介绍

总体而言，正向和反向运动学系统都是由相互连接的零部件构成，如一串链条或一串由关节相连的手臂。它们要负责整个系统的运动，以及某个零件相对于其它零件和整个系统的运动。

通常一个运动学系统都有两个末端：固定端（base）和自由端（free）。带有关节的手臂通常有一端被固定住，而另一端则用于伸出去抓东西。一串链条也许有一两个末端都连在某个东西上，或者什么都不连。

*正向运动学（Forward kinematics，缩写：FK）*中的运动是以系统的固定端为起始，在自由端进行运动。*反向运动学（Inverse kinematics，缩写：IK）*则是向反的：运动以自由端为起始，回退到固定端，如果有的话。

通常情况下，下肢在行走时都看作是正向运动学。大腿的移动带动小腿的运动，小腿的移动带动脚的运动，最终让脚产生运动。脚的运动不会决定其它部分的运动，它所带动的就是它本身，其位置根据下肢的位置来确定。

一个反向运动学的例子，就是用手拉某人。这里，力作用于自由端—— 手 ——控制手，前臂，大臂，以致整个身体的位置和运动。

说得更细一点，例如反向运动学是一支手伸出去拿东西，而手就传动了这个系统。当然，也可以说，大臂和前臂也是运动的，它们控制了手的位置。没错，不过最直接的目的是把手放在某个特定的位置。这就是传动力。它不是一种实际的力，而是一种意图。前臂和大臂只是根据结构的需要通过排列它们的位置来设置手的位置。

我们会在下一章通过具体的例子弄清二者间的差别。不过现在，要记住拖动和伸臂是一般的反向运动学，而一个重复的循环运动，如行走，则是最常见的正向运动学，也就是本章的课题。

正向运动学编程准备

两种运动学编程都有如下基本要素：

- 系统部件。我们称为关节(Segment)。
- 各关节的位置。
- 各关节的旋转。

例子中的每个关节都是一个长方形，就像前臂、大臂，或大腿的一部分。当然，最末端

的关节可以是任意的形状，比如手，脚，钳子，刺或是入侵者的绿色激光炮。

每个关节的都会有一个末端作为枢轴，围绕它可以进行旋转。如果这个关节还有其它子关节的话，子关节还会以它们的另一末端作为旋转的枢轴。就像大臂绕着肩膀旋转，小臂绕着肘部旋转，手绕着手腕旋转一样。

当然，在很多现实的系统中，这种绕轴旋转可以有多个方向。想一想我们有多少种方法可以让手绕着手腕旋转。到本书的最后，大家也许会在 Flash 中亲自进行一下尝试。但是，现在，我们这个系统完全是二维的。

单关节运动

让我们从单个关节的运动入手。首先，需要一个像关节一样的物件。设想一下，它应该是一个继承自 Sprite 的自定义类。因为 sprite 影片可以进行绘图，设定位置，旋转，加入显示列表。以下是这个类 Segment.as：

```
package {
 import flash.display.Sprite;
 import flash.geom.Point;
 public class Segment extends Sprite {
  private var color:uint;
  private var segmentWidth:Number;
  private var segmentHeight:Number;
  public var vx:Number = 0;
  public var vy:Number = 0;
  public function Segment(segmentWidth:Number,
  segmentHeight:Number,
  color:uint = 0xffffff) {
   this.segmentWidth = segmentWidth;
   this.segmentHeight = segmentHeight;
   this.color = color;
   init();
  }
  public function init():void {
   // 绘制关节
   graphics.lineStyle(0);
   graphics.beginFill(color);
   graphics.drawRoundRect(-segmentHeight / 2,
   -segmentHeight / 2,
   segmentWidth + segmentHeight,
   segmentHeight,
   segmentHeight,
   segmentHeight);
   graphics.endFill();
   // 绘制两个“枢轴”
   graphics.drawCircle(0, 0, 2);
   graphics.drawCircle(segmentWidth, 0, 2);
  }
  public function getPin():Point {
   var angle:Number = rotation * Math.PI / 180;
```

```
    var xPos:Number = x + Math.cos(angle) * segmentWidth;
    var yPos:Number = y + Math.sin(angle) * segmentWidth;
    return new Point(xPos,yPos);
   }
  }
}
```

用宽度、高度、颜色，以及绘制一个圆角矩形定义一个关节。再绘制两个小圆，一个放在关节的注册点（0,0）位置上，另一个则放在相对的另一个末端，这两个“枢轴”用于关节之间的相互连接。（大家也许注意到了两个公共变量 vx 和 vy。稍后在本章的“处理反作用力”一节会做更多的介绍）下面一段代码使用不同的宽度和高度创建了若干个关节（segment），给大家一些创建 Segment 类实例的启发：

```
var segment:Segment = new Segment(100, 20);
addChild(segment);
segment.x = 100;
segment.y = 50;
var segment1:Segment = new Segment(200, 10);
addChild(segment1);
segment1.x = 100;
segment1.y = 80;
var segment2:Segment = new Segment(80, 40);
addChild(segment2);
segment2.x = 100;
segment2.y =120;
```

这段代码的执行结果如图 13-1 所示。

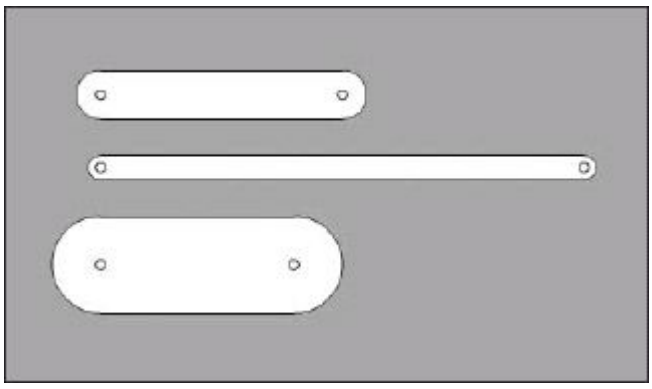


图 13-1 一些关节示例

图 13-1 中一个要点是关节的宽度可以决定两个枢轴间的距离，关节实际的宽度超出了二者的范围。我们可以看到每个关节都放在了 x 轴为 100 的位置上。虽然它们的左侧的边没有排列整齐，但所有左侧的枢轴却排列得很整齐。当我们旋转关节时，则会绕着左侧的枢轴进行旋转。

我们同样注意了 Segment 类的代码有一个公共的 getPin() 方法，返回一个 flash.geom.Point 的实例。它将返回右侧枢轴的 x,y 坐标。显然，它会随着关节的旋转而改变，所以我们使用一些基本的三角学来计算这个位置。这也就是下一个关节将连接上的位置——我们会在本章看到下一个关节。

对于第一个例子程序，SingleSegment.as，已经创建好了… …在舞台上放置单个关节。同时，我还创建一个滑块类 SimpleSlider.as 加入到了这个项目中。大家可以从本书的下载页面 www.friendsofed.com 中进行下载，这样我们就可以在任何时候自由地使用这个类了，它用于在运行时对数值进行调整。对于这个滑块我们可以在构造函数中设置最小值，最大值，以及当前取值。下面例子中，将最小值设为 -90，最大值设为 90，取值设为 0。这个类文件结合了上述所有内容：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class SingleSegment extends Sprite {
  private var slider:SimpleSlider;
  private var segment:Segment;
  public function SingleSegment() {
   init();
  }
  private function init():void {
   segment = new Segment(100,20);
   addChild(segment);
   segment.x = 100;
   segment.y = 100;
   slider = new SimpleSlider(-90,90,0);
   addChild(slider);
   slider.x = 300;
   slider.y = 20;
   slider.addEventListener(Event.CHANGE,onChange);
  }
  private function onChange(event:Event):void {
   segment.rotation = slider.value;
  }
 }
}
```

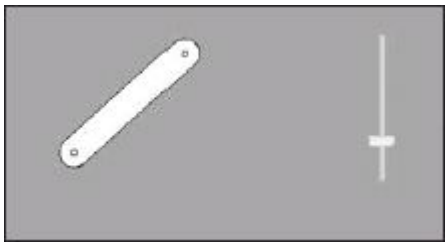


图 13-2 动起来了！

意思是说无论何时滑块（slider）的取值发生了改变，都将调用 onChange 方法，设置 segment 的 rotation 为当前滑块的值。测试一下，运行结果如图 13-2 所示。如果运行没有问题，我们就完成了正向运动学的第一阶段。

双关节的运动

现在我们要继续前进。最初的滑块和关节都命名为 slider0 和 segment0，再创建另一个关节的实例，名为 segment1，还要创建一个滑块实例名为 slider1。新滑块将控制新关节的运动，新关节的位置将由 segment0 的 getPin() 方法来确定。以下是代码，见 TwoSegments.as：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class TwoSegments extends Sprite {
  private var slider0:SimpleSlider;
```

```
  private var slider1:SimpleSlider;
  private var segment0:Segment;
  private var segment1:Segment;
  public function TwoSegments() {
   init();
  }
  private function init():void {
   segment0 = new Segment(100,20);
   addChild(segment0);
   segment0.x = 100;
   segment0.y = 100;
   segment1 = new Segment(100,20);
   addChild(segment1);
   segment1.x = segment0.getPin().x;
   segment1.y = segment0.getPin().y;
   slider0 = new SimpleSlider(-90,90,0);
   addChild(slider0);
   slider0.x = 320;
   slider0.y = 20;
   slider0.addEventListener(Event.CHANGE,onChange);
   slider1 = new SimpleSlider(-90,90,0);
   addChild(slider1);
   slider1.x = 340;
   slider1.y = 20;
   slider1.addEventListener(Event.CHANGE,onChange);
  }
  private function onChange(event:Event):void {
   segment0.rotation = slider0.value;
   segment1.rotation = slider1.value;
   segment1.x = segment0.getPin().x;
   segment1.y = segment0.getPin().y;
  }
 }
}
```

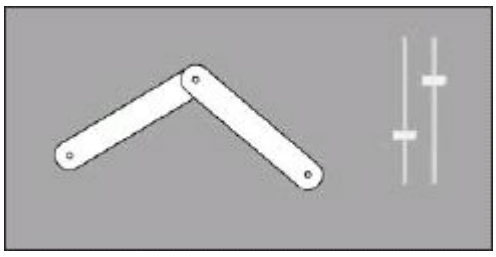


图 13-3 双关节正向运动

快速浏览一下 onChange 方法，我们看到这里用 segment0.getPin() 的返回值来设置 segment1 的位置。与首次创建 segment1 时设置位置的代码相同。

我们让 slider1 和 slider0 都来调用 onChange 方法。很明显，现在 segment1 的 rotation 要根据 slider1 来确定。

测试一下，我们看到当旋转 segment0 时，segment1 依然与它的末端相连，如图 13-3 所示。要知道两者间没有实际的物理链接，而都是用数学的方法计算出来的。我们同样可以使用 segment1 的滑块独立地对它进行旋转。为了好玩一点，改变每个关节的宽度和高度后，程序仍可以完美地工作。不过，看起来有些奇怪，因为当 segment0 带动 segment1 运动时；

segment0 并没有带动 segment1 旋转。这就像安装了回旋稳定器，将它的方向稳定住了一样。我不知道您是怎么样的，反正我的前臂没有装回旋稳定器（尽管可能会很酷），因此，这样的运动看起来不太自然。真正的情况应该是，segment1 的旋转应等于 segment0 的 rotation 加上 slider1 的值。那么 TwoSegment2.as 文档类的函数应该是这样：

```
private function onChange(event:Event):void {
 segment0.rotation = slider0.value;
 segment1.rotation = segment0.rotation + slider1.value;
 segment1.x = segment0.getPin().x;
 segment1.y = segment0.getPin().y;
}
```

现在，看起来像是真正的一条手臂了。当然，如果讨论人类的手臂，就不能像这个肘部一样向两个方向弯曲。我们只需要改变 slider1 的范围，让最小值等于 -160 让最大值为 0，那么看上去就更加正常了，代码如下：

```
slider1 = new SimpleSlider(-160, 0, 0);
```

现在应该是重新思考正向运动学的最佳时机了。这个系统的固定端是 segment0 的枢轴点，自由端是 segment1 的自由端。这时我们可以想象一下手臂。固定端的旋转和位置决定了 segment1 的位置。而 segment1 的旋转和位置决定了其自由端的位置。自由端在哪无所谓，它只是来凑凑热闹而已。因此，控制就是从固定端向前运动到自由端。

自动化过程

这些滑块给了我们控制旋转的权力，但是我们所创建的则是像机械结构中的液压杠杆，让零件产生运动。如果想实现真正的行走，就需要加入一些自动控制。

只需要让每个关节平稳地前后摇摆，并让它们保持同步。听起来就像正弦波一样。

在 Walking1.as 中，我用三角函数代替了滑块。设置一个正弦循环变量（初始为 0）乘以 90，让数值的变化从 90 到 -90。循环变量是实时累加的，这样就可以实现摆动。接下来就可以使用计算出的角度变量来控制两个关节了。加入 enterFrame 函数来控制动作，这样运动就是持续的了。

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Walking1 extends Sprite {
  private var segment0:Segment;
  private var segment1:Segment;
  private var cycle:Number = 0;
  public function Walking1() {
   init();
  }
  private function init():void {
   segment0 = new Segment(100,20);
   addChild(segment0);
   segment0.x = 200;
   segment0.y = 200;
   segment1 = new Segment(100,20);
   addChild(segment1);
   segment1.x = segment0.getPin().x;
   segment1.y = segment0.getPin().y;
```

```
    addEventListener(Event.ENTER_FRAME,onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   cycle += .05;
   var angle:Number = Math.sin(cycle) * 90;
   segment0.rotation = angle;
   segment1.rotation = segment0.rotation + angle;
   segment1.x = segment0.getPin().x;
   segment1.y = segment0.getPin().y;
  }
 }
}
```

创建自然行走循环

OK，现在我们已经可以让胳膊运动了。接下来把它变成腿。进行如下的调整：

1.将 segment0 的旋转增加 90 度，将范围减小到从 90 度开始到每个方向 45 度。

2.每个关节都要有不同的角度，因此要声明两个角度 angle0 和 angle1。

3.将 angle1 的范围减少到 45 度，然后再增加 45 度。这样就使最终的范围变为 0 到 90 度，因此只能向一个方向弯曲，像是真正的膝关节。光这样说不能完全解释清楚，请大家试着观察加或不加 45 度有何区别，或试着加入其它的数值，直到感觉都合适为止。

完成的结果见 Walking2.as。下面列出 onEnterFrame 方法，其它地方没有改变：

```
private function onEnterFrame(event:Event):void {
 cycle += .05;
 var angle0:Number = Math.sin(cycle) * 45 + 90;
 var angle1:Number = Math.sin(cycle) * 45 + 45;
 segment0.rotation = angle0;
 segment1.rotation = segment0.rotation + angle1;
 segment1.x = segment0.getPin().x;
 segment1.y = segment0.getPin().y;
}
```

OK，成功，如图 13-4 所示。开始变得像条腿了，至少运动起来像是条腿。

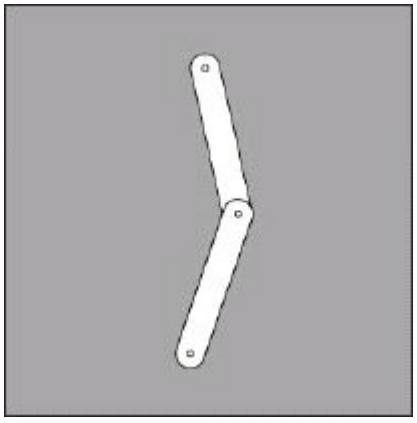


图 13-4 循环行走

看起来还是不像真正在走路。也许它是在漫不经心地踢球，或是在练习芭蕾舞，但就是不像在走路。这是因为两个关节在同一时刻内向同一方向运动。它们是完全同步的，而真正的行走过程，并不是这样的。

关节的同步是因为它们都使用了相同的循环变量来计算各自的角度。要让它们不再同

步，我们应采用 cycle0 和 cycle1 变量，但是可以不做这么大的改变。只需要加入循环的偏移量求出 angle1 即可，如下：

```
var angle1:Number = Math.sin(cycle + offset) * 45 + 45;
```

当然我们需要先定义好偏移（offset）的值。那么应该偏移多少呢？我也不知道。调整到您认为满意为止吧。给大家一些提示：这个数应该在 Math.PI 到 -Math.PI （3.14 到 -3.14）之间。任何大于或小于这个范围的数，只是将这个数进行了重复而已。例如，我用 -Math.PI / 2，将 angle0 向后推四分之一个周期。当然，-Math.PI / 2 大约为 -1.57，因此我们可以试试其周边的一些数如 -1.7 或 -1.3，看看效果是好是坏。稍后，我们将放入一个滑块来动态地进行调整。带有偏移的循环行走代码见 Walking3.as。

“单腿行走”对我来说太残酷了，让我们加入另一条腿。需要再加入两个关节，名为 segment2 和 segment3。segment2 对象应与 segment0 位置相同，因为它也将是一个顶级端或固定端，而 segment3 应该用 segment2 的 getPin() 方法来设置位置。

下面，我将整个代码抽象为一个方法，名为 walk，这样比复制所有的代码让 segment0 和 segment1 运动要好得多：

```
private function walk(segA:Segment, segB:Segment, cyc:Number):void {
 var angleA:Number = Math.sin(cyc) * 45 + 90;
 var angleB:Number = Math.sin(cyc + offset) * 45 + 45;
 segA.rotation = angleA;
 segB.rotation = segA.rotation + angleB;
 segB.x = segA.getPin().x;
 segB.y = segA.getPin().y;
}
```

注意这个函数有三个参数：两个关节，segA 和 segB，以及 cyc 代表 cycle。剩下的代码都是我们用过的。现在让 segment0 和 segment1 行走，只需要这样调用：

```
walk(segment0, segment1, cycle);
```

现在，大家知道如何与它打交道了吧，那么就准备好让 segment2 和 segment3 也走起来吧。onEnterFrame 方法如下：

```
private function onEnterFrame(event:Event):void {
 walk(segment0, segment1, cycle);
 walk(segment2, segment3, cycle);
 cycle += .05;
}
```

如果这样做的话，就会惊奇地发现，第二条腿不见了。问题在于两条腿是完全同步的，所以看上去像是只有一条腿。因此，要让它们不能同步运动。我们要将第二条腿与第一条腿的运动周期进行偏移。这就要改变 cycle 的值。只需要在 cycle 上面加上或减去一些值就可以了，这样做比加入两个不同的变量要好得多。因此 onEnterFrame 就变成了这样：

```
private function onEnterFrame(event:Event):void {
 walk(segment0, segment1, cycle);
 walk(segment2, segment3, cycle + Math.PI);
 cycle += .05;
}
```

为什么是 Math.PI？完整的回答是，这个值让第二条腿与第一条腿的同步相差了 180 度，因此在第一条腿向前走时，第二条腿正在向后走，反之亦然。简略的回答是，因为它奏效了！大家可以试试其它的值，如 Math.PI / 2，观察一下这个运动，就像疾速飞跑，而不是在行走或跑步。不过需要加以留心——说不定某一天会需要它呢！

按照惯例文件保存在 Walking4.as 中，如图 13-5 所示。基础的关节（"thighs"[大腿]）比末端关节（"calves"[小腿]）要稍大一些。请记住，由于我们使用的方法是动态的，所以影片的运行与零件的大小无关。在下一个版本中，我们会用滑块使更多的部件变成动态的，不过我强烈建议大家现在开始手动改变一下代码中变量的值，观察一下不同的变量值所带来

的不同效果。

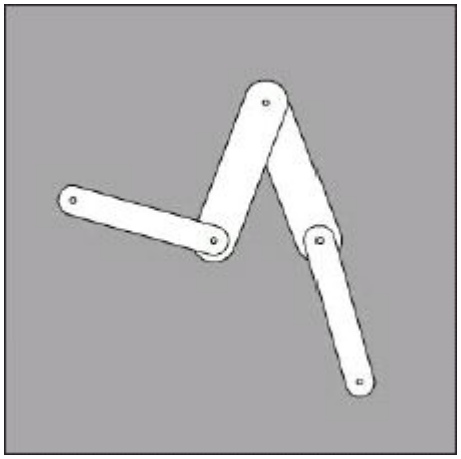


图 13-5 看！走起来了！

动态调整

下面，让我们来真正体验一下在循环行走中改变各个值所带来的不一样的效果。Walking5.as 又将 Slider 滑块类加入进来，用于动态改变这些变量。

本例中，在屏幕上创建并放置了五个滑块，如图 13-6 所示。

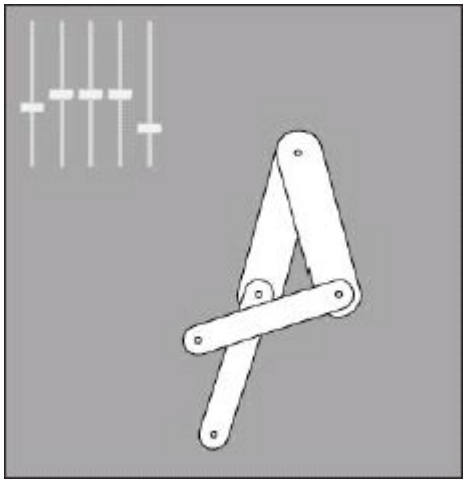


图 13-6 加入滑块

表 13-1 所示，滑块的名字（从左到右），功能以及设置。这些都是我认为比较合理的范围和取值。总之，我们可以任意地实验其它的取值。

表 13-1 控制行走的滑块

实例	描述	设置
speedSlider [速度滑块	控制整个系统的运动速度。	最小值：0, 最大值：0.3, 默认值：0.12
thighRangeSlider [大腿运动范围滑块]	控制顶层关节（大腿）能够向前和向后移动多远。	最小值：0, 最大值：90, 默认值：45
thighBaseSlider [大腿固定端滑块]	控制顶层关节的基本角度。默认为 90 度， 也就是说大腿将垂直向下并从这里向前后运动。 通过改变这个值可以得到一些有趣的效果。	最小值：0, 最大值：180, 默认值：90
calfRangeSlider [小腿运动范围滑块]	控制底层关节（小腿）能够运动的范围。	最小值：0, 最大值：90, 默认值：45
calfOffsetSlider [小腿偏移滑块]	控制偏移量（前面我们用过 -Math.PI / 2）。	最小值：-3.14, 最大值：3.14, 默认值：-1.57

下面，为了能够用滑块取代手动设值我们来改变一下代码。

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  public class Walking5 extends Sprite {
    private var segment0:Segment;
    private var segment1:Segment;
    private var segment2:Segment;
    private var segment3:Segment;
    private var speedSlider:SimpleSlider;
    private var thighRangeSlider:SimpleSlider;
    private var thighBaseSlider:SimpleSlider;
    private var calfRangeSlider:SimpleSlider;
    private var calfOffsetSlider:SimpleSlider;
    private var cycle:Number = 0;
    public function Walking5() {
      init();
    }
```

```
private function init():void {
  segment0 = new Segment(100, 30);
  addChild(segment0);
  segment0.x = 200;
  segment0.y = 100;
  segment1 = new Segment(100, 20);
  addChild(segment1);
  segment1.x = segment0.getPin().x;
  segment1.y = segment0.getPin().y;
  segment2 = new Segment(100, 30);
  addChild(segment2);
  segment2.x = 200;
  segment2.y = 100;
  segment3 = new Segment(100, 20);
  addChild(segment3);
  segment3.x = segment2.getPin().x;
  segment3.y = segment2.getPin().y;
  speedSlider = new SimpleSlider(0, 0.3, 0.12);
  addChild(speedSlider);
  speedSlider.x = 10;
  speedSlider.y = 10;
  thighRangeSlider = new SimpleSlider(0, 90, 45);
  addChild(thighRangeSlider);
  thighRangeSlider.x = 30;
  thighRangeSlider.y = 10;
  thighBaseSlider = new SimpleSlider(0, 180, 90);
  addChild(thighBaseSlider);
  thighBaseSlider.x = 50;
  thighBaseSlider.y = 10;
  calfRangeSlider = new SimpleSlider(0, 90, 45);
  addChild(calfRangeSlider);
  calfRangeSlider.x = 70;
  calfRangeSlider.y = 10;
  calfOffsetSlider = new SimpleSlider(-3.14, 3.14, -1.57);
  addChild(calfOffsetSlider);
  calfOffsetSlider.x = 90;
  calfOffsetSlider.y = 10;
  addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
private function onEnterFrame(event:Event):void {
  walk(segment0, segment1, cycle);
  walk(segment2, segment3, cycle + Math.PI);
  cycle += speedSlider.value;
}
private function walk(segA:Segment, segB:Segment,
cyc:Number):void {
  var angleA:Number = Math.sin(cyc) *
  thighRangeSlider.value +
  thighBaseSlider.value;
```

```
    var angleB:Number = Math.sin(cyc +
    calfOffsetSlider.value) *
    calfRangeSlider.value +
    calfRangeSlider.value;
    segA.rotation = angleA;
    segB.rotation = segA.rotation + angleB;
    segB.x = segA.getPin().x;
    segB.y = segA.getPin().y;
   }
  }
}
```

这段代码同前面完全一样，只不过使用滑块的值来控制而非以前的手动控制。我保证从这个程序中你会得到很多乐趣，探索不同的变化，观察行走效果。

让物体真正走起来

到现在为止，两条腿的运动看起来已经相当真实了，但是它们好像漂浮在太空中一样。前面章节中，我们学习了如何让物体以一定的速度或加速度运动，然后会让物体与环境产生交互。这次也不例外。本章的这一部分显得相当复杂，所以在介绍每段代码时都会给大家相关的概念。最终的文件会包括所有的这些概念，它就是 RealWalk.as。

给它一些空间

因为物体终归要进行移动，所以应该让所有的零件变小一点，以便给它们留出更多的运动空间。创建初始的关节并让它们是原来大小的一半，例如：

```
segment0 = new Segment(50, 15);
```

接下来，由于要进行运动并以边界产生交互，所以需要定义 vx 和 vy：

```
private var vx:Number = 0;
private var vy:Number = 0;
```

这样一来，运行这个例子后，我们就得到了一个缩小后的版本。

加入重力

接下来，我们需要创建重力。此外，即使有边界的作用力，双腿也会像是漫步在太空一样。我们需要在运行时加入了重力变量，使用另一个滑块！创建一个新的滑块实例名为 gravitySlider。设置最小值为 0，最大值为 1，取值为 0.2。创建最后一个控制滑块的代码段如下：

```
gravitySlider = new SimpleSlider(0, 1, 0.2);
addChild(gravitySlider);
gravitySlider.x = 110;
gravitySlider.y = 10;
```

下面我们需要连同重力加速度一起来计算速度。创建一个名为 doVelocity 的方法将代码写在里面，这样比都挤到 onEnterFrame 中要好些：

```
private function onEnterFrame(event:Event):void {
 doVelocity();
 walk(segment0, segment1, cycle);
```

```
 walk(segment2, segment3, cycle + Math.PI);
 cycle += speedSlider.value;
}
```

在这个方法中，只需要将重力加入到 vy，将 vx 和 vy 加入到 segment0 和 segment2 的位置上。记住我们不需要担心 segment1 和 segment3，因为它们的位置是依照顶层关节的位置计算出来的。

```
private function doVelocity():void {
 vy += gravitySlider.value;
 segment0.x += vx;
 segment0.y += vy;
 segment2.x += vx;
 segment2.y += vy;
}
```

测试后，结果并不令人兴奋。目前还没有 x 速度，而重力也只是将物体向下拉，以至穿过地面。所以我们需要判断地面是否与双腿产生了碰撞，也就意味着要进行碰撞检测。

处理碰撞

首先，还要让 onEnterFrame 去调用另一个方法，checkFloor。在调用了 walk 方法之后再调用它，因此它的操作在最后面。总的来说，只需要判断 segment1 和 segment3 ——底层的关节 ——看看它们是否与地面产生了碰撞。因此两关节都要调用 checkFloor 方法。

```
private function onEnterFrame(event:Event):void {
 doVelocity();
 walk(segment0, segment1, cycle);
 walk(segment2, segment3, cycle + Math.PI);
 cycle += speedSlider.value;
 checkFloor(segment1);
 checkFloor(segment3);
}
```

现在进入了第一个有趣的部分：碰撞检测。注意我说的是“有趣”而不是“困难”。因为实现起来相当简单。我们需要知道关节的某个部分是否低于底部边界（用 bottom 变量指定）。

也许最简单的办法就是让关节调用 getBounds 来判断边界的 bottom 属性是否大于舞台的高度。下面是 checkFloor 方法的开始部分：

```
private function checkFloor(seg:Segment):void {
 var yMax:Number = seg.getBounds(this).bottom;
 if (yMax > stage.stageHeight) {
 }
}
```

对于函数中的第一行我做了一些简化，可能让大家看起来有些怪怪的。

```
var yMax:Number = seg.getBounds(this).bottom;
```

一般情况下，大家也许希望写两步，如下：

```
var bounds:Rectangle = seg.getBounds(this);
var yMax:Number = bounds.bottom;
```

可能不是每个人都能意识到 seg.getBounds(this) 表达式返回一个 Rectangle 对象，可以直接访问 Rectangle 的属性，像这样：seg.getBounds(this).bottom。Flash 首先 getBounds 计算出一个 Rectangle 对象，接着看到了点号和属性，然后就看作是这个对象的一个属性。如果这种语法让你感到困惑的话，可以继续使用两行代码的方法，别忘了导入

flash.geom.Rectangle 类。如果大家还要对边界对象进行其它操作的话，例如读取附加的属性，就应该先将这个对象加以保存。但是在本例中，只读取了 bottom 属性，所以可直接进行调用，而没有使用额外的变量。

OK，现在假设 yMax 确实大于 stage.stageHeight，用现实世界的话说就是，腿与地面产生了碰撞。我们应该做什么？就像前面介绍的边界碰撞一样，首先把物体重置到边界上方。如果 yMax 是关节的底边，而 stage.stageHeigth 是地面，我们需要将关节运动回它们之间实际的距离。换句话讲，假设 stage.stageHeight 是 600，而 yMax 是 620，就需要将关节的 y 坐标 -20。但是，不能只移动这个关节，我们需要让所有的关节都移动这个数值，因为它们都是同一个身体上的组成部分，必需当成一个整体进行移动。因此，我们的代码如下：

```
private function checkFloor(seg:Segment):void {
 var yMax:Number = seg.getBounds(this).bottom;
 if (yMax > stage.stageHeight) {
  var dy:Number = yMax - stage.stageHeight;
  segment0.y -= dy;
  segment1.y -= dy;
  segment2.y -= dy;
  segment3.y -= dy;
 }
}
```

试调整滑块的值，观察不同的行走效果。这时，我们更加感觉到双腿与环境产生了交互。当然，这还不是真正的行走。

处理反作用力

现在我们已经成功地让双腿与地面发生了接触，但只重置位置还不够真实。因为，行走应该使水平方向也产生运动——x 速度。此外，我们的行走应该对 y 速度产生一定的影响——至少要能与重力暂时地抵消一小会儿。在奔跑中，地面之上停留一会儿。

当脚落地并与地面接触时，就再不能向下了，以便使垂直动量反作用于身体，使身体向上运动。脚下落的力气越大，那么被抬起来的力量也就越大。同理，如果在与地面接触时，脚是向后运动的，那么身体就会受到水平的动量，向前进。脚向后运动得越快，水平的推力就越大。

OK，现在理论已经有了。如果能知道“脚”的 x,y 速度，那么在碰撞时，就可以从 vx,vy 中将 x,y 速度减去。

第一个问题，我们现在还没有找到脚这个物体。事实上，虽然可以自由地加入物理的脚在上面，并用第二个关节来指定位置，但我不想用真实的脚作例子。而只要计算出虚拟脚的位置，这个值由底层的关节使用 getPin() 返回。

如果在这个关节运动之前知道它的枢轴的位置，以及运动后关节的位置，可以求出两者的差值来获得脚的 x,y 速度。我们可以在 walk 方法中写入实现，然后将值保存在公有的 vx,vy 属性中。注意，因为使用到了 Point 类，所以需要导入 flash.geom.Point。

```
function walk(segA:Segment, segB:Segment, cyc:Number):void {
 var foot:Point = segB.getPin();
 var angleA:Number = Math.sin(cyc) *
 thighRangeSlider.value +
 thighBaseSlider.value;
 var angleB:Number = Math.sin(cyc + calfOffsetSlider.value) *
 calfRangeSlider.value +
 calfRangeSlider.value;
 segA.rotation = angleA;
```

```
  segB.rotation = segA.rotation + angleB;
  segB.x = segA.getPin().x;
  segB.y = segA.getPin().y;
  segB.vx = segB.getPin().x - foot.x;
  segB.vy = segB.getPin().y - foot.y;
}
```

现在，每个底部的关节都有各自的 vx,vy 属性，表示底部枢轴点的速度，或虚拟脚的速度，而不是该关节的速度。

那么用这个速度做什么呢？当与地面接触时，将它从总速度中减去。换句话讲，如果脚在接触地面后，又以每帧 3 像素（vy = 3）运动，我们就要将全部的 vy 将去 3。同样 vx 也是如此。代码实现非常简单：

```
private function checkFloor(seg:Segment):void {
  var yMax:Number = seg.getBounds(this).bottom;
  if (yMax > stage.stageHeight) {
    var dy:Number = yMax - stage.stageHeight;
    segment0.y -= dy;
    segment1.y -= dy;
    segment2.y -= dy;
    segment3.y -= dy;
    vx -= seg.vx;
    vy -= seg.vy;
  }
}
```

不可否认这是一种极为简单但或许有些不太精确的行走力量的表示。但是，我还会笑着说“请测试这个影片”。看看是否喜欢这个效果。在我看来它确实很酷。我做了很多双腿在屏幕上跑来跑去。

屏幕环绕，回访

大家都看到了，双腿离开了屏幕后，一去不复返。小小的屏幕环绕很适合用在这里。在双腿向右离开时，就让它们回到左侧。这里的屏幕环绕比以前稍微复杂一些，因为一个整体中有四个部分，而原来只是单独的一个物体。我们只需判断两个顶部关节其中的一个，因为通常它们的位置是相同的，而底部关节的位置都是由顶层关节来决定的。在 onEnterFrame 中加入调用 checkWalls 的方法：

```
private function onEnterFrame(event:Event):void {
  doVelocity();
  walk(segment0, segment1, cycle);
  walk(segment2, segment3, cycle + Math.PI);
  cycle += speedSlider.value;
  checkFloor(segment1);
  checkFloor(segment3);
  checkWalls();
}
```

让我们留出 100 像素的空隙，以便让每条腿可以在环绕之前向舞台的右侧运动 100 像素。如果物体穿越了这个点，则将所有部件都重置到屏幕左侧。那么距离左侧多远呢？舞台的宽度加上 200，为每边留出 100 像素的空隙。因此 checkWalls 中的 if 语句如下：

```
private function checkWalls():void {
  var w:Number = stage.stageWidth + 200;
```

```
 if (segment0.x > stage.stageWidth + 100) {
  segment0.x -= w;
  segment1.x -= w;
  segment2.x -= w;
  segment3.x -= w;
 }
}
```

在左侧也加入相同的判断，因为有些行走动作会使双腿向后运动。最终的 checkWalls 方法如下：

```
private function checkWalls():void {
 var w:Number = stage.stageWidth + 200;
 if (segment0.x > stage.stageWidth + 100) {
  segment0.x -= w;
  segment1.x -= w;
  segment2.x -= w;
  segment3.x -= w;
 } else if (segment0.x < -100) {
  segment0.x += w;
  segment1.x += w;
  segment2.x += w;
  segment3.x += w;
 }
}
```

为了让大家能看得清楚，不产生混乱，下面给出所有的代码（可见 RealWalk.as）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.geom.Point;
 public class RealWalk extends Sprite {
  private var segment0:Segment;
  private var segment1:Segment;
  private var segment2:Segment;
  private var segment3:Segment;
  private var speedSlider:SimpleSlider;
  private var thighRangeSlider:SimpleSlider;
  private var thighBaseSlider:SimpleSlider;
  private var calfRangeSlider:SimpleSlider;
  private var calfOffsetSlider:SimpleSlider;
  private var gravitySlider:SimpleSlider;
  private var cycle:Number = 0;
  private var vx:Number = 0;
  private var vy:Number = 0;
  public function RealWalk() {
   init();
  }
  private function init():void {
   segment0 = new Segment(50, 15);
   addChild(segment0);
   segment0.x = 200;
   segment0.y = 100;
   segment1 = new Segment(50, 10);
```

```
  addChild(segment1);
  segment1.x = segment0.getPin().x;
  segment1.y = segment0.getPin().y;
  segment2 = new Segment(50, 15);
  addChild(segment2);
  segment2.x = 200;
  segment2.y = 100;
  segment3 = new Segment(50, 10);
  addChild(segment3);
  segment3.x = segment2.getPin().x;
  segment3.y = segment2.getPin().y;
  speedSlider = new SimpleSlider(0, 0.3, 0.12);
  addChild(speedSlider);
  speedSlider.x = 10;
  speedSlider.y = 10;
  thighRangeSlider = new SimpleSlider(0, 90, 45);
  addChild(thighRangeSlider);
  thighRangeSlider.x = 30;
  thighRangeSlider.y = 10;
  thighBaseSlider = new SimpleSlider(0, 180, 90);
  addChild(thighBaseSlider);
  thighBaseSlider.x = 50;
  thighBaseSlider.y = 10;
  calfRangeSlider = new SimpleSlider(0, 90, 45);
  addChild(calfRangeSlider);
  calfRangeSlider.x = 70;
  calfRangeSlider.y = 10;
  calfOffsetSlider = new SimpleSlider(-3.14, 3.14, -1.57);
  addChild(calfOffsetSlider);
  calfOffsetSlider.x = 90;
  calfOffsetSlider.y = 10;
  gravitySlider = new SimpleSlider(0, 1, 0.2);
  addChild(gravitySlider);
  gravitySlider.x = 110;
  gravitySlider.y = 10;
  addEventListener(Event.ENTER_FRAME, onEnterFrame);
 }
 private function onEnterFrame(event:Event):void {
  doVelocity();
  walk(segment0, segment1, cycle);
  walk(segment2, segment3, cycle + Math.PI);
  cycle += speedSlider.value;
  checkFloor(segment1);
  checkFloor(segment3);
  checkWalls();
 }
 private function walk(segA:Segment, segB:Segment,
 cyc:Number):void {
  var foot:Point = segB.getPin();
```

```
      var angleA:Number = Math.sin(cyc) *
      thighRangeSlider.value +
      thighBaseSlider.value;
      var angleB:Number = Math.sin(cyc +
      calfOffsetSlider.value) *
      calfRangeSlider.value +
      calfRangeSlider.value;
      segA.rotation = angleA;
      segB.rotation = segA.rotation + angleB;
      segB.x = segA.getPin().x;
      segB.y = segA.getPin().y;
      segB.vx = segB.getPin().x - foot.x;
      segB.vy = segB.getPin().y - foot.y;
    }
    private function doVelocity():void {
      vy += gravitySlider.value;
      segment0.x += vx;
      segment0.y += vy;
      segment2.x += vx;
      segment2.y += vy;
    }
    private function checkFloor(seg:Segment):void {
      var yMax:Number = seg.getBounds(this).bottom;
      if (yMax > stage.stageHeight) {
        var dy:Number = yMax - stage.stageHeight;
        segment0.y -= dy;
        segment1.y -= dy;
        segment2.y -= dy;
        segment3.y -= dy;
        vx -= seg.vx;
        vy -= seg.vy;
      }
    }
    private function checkWalls():void {
      var w:Number = stage.stageWidth + 200;
      if (segment0.x > stage.stageWidth + 100) {
        segment0.x -= w;
        segment1.x -= w;
        segment2.x -= w;
        segment3.x -= w;
      } else if (segment0.x < -100) {
        segment0.x += w;
        segment1.x += w;
        segment2.x += w;
        segment3.x += w;
      }
    }
  }
}
```

13.5 小结

在本章中你已经制作了非常漂亮的作品，掌握了正向运动的基础内容。注意这里我给出的方法可能并不是唯一的解决方案。他们显然是针对某一特殊的技术应用而设计的：让某物行走。你可以不用这些，或改变些什么，或是加入你想加入的。不断实验它看你可以制作出什么。

下一步，你将看到另一面：反向运动。

第十四章　反向运动学：拖拽与伸展

第十三章介绍了一些基础的运动学以及正向与反向运动学之间的区别。前一章我们讲了正向运动学，本章就要学习与它关系紧密的反向运动学。涉及到的动作就是拖拽与伸展。

与正向运动学的例子相同，本章的例子也是从独立的关节开始建立系统。我们从单个关节开始，然后到多个关节。首先，我会给大家演示最简单的计算角度与位置的方法。只是在代码中使用基本的三角学进行大概的测算。最后，会给大家简要地介绍使用余弦定理的方法，这样计算出来的结果更加准确，但会消耗大量的计算——这就是所谓的权衡。

单物体的拖拽与伸展

前面说过，反向运动学系统可以分为两种不同的类型：拖拽与伸展。

当系统的自由端向目标点伸展时，系统的另一端——固定端，也许是动不了的，因此如果目标点位置超出了自由端运动的范围，那么自由端永远也不能到达目标点。举个例子，当我们试图抓住某个东西时，手指就朝着这个物体移动，手腕的转动会使我们的手指与目标位置越来越近，肘部，肩膀和身体其它的部分也都尽可能地伸展。有时，所有这些位置的组合将会使手指接触到物体；有时也许不行。如果物体是来回运动的，我们的肢体就要做出即时的反映不断调整位置，为了让手指能够尽可能地够到该物体。反向运动学将会告诉我们，如何设置所有这些零件的位置，达到最佳的伸展效果。

另一种反向运动学是在物体被拖拽的时候。这个例子中，自由端是被一些外部的力所拖动的。无论何时，系统其余的部分都紧随其后，它们会将自己放置到自然的可能位置上。可以想象成一个没有知觉死尸（对不起，这是我唯一能想到的）。抓住他的手然后拽着它走。

我们施加在对方手上的力，会传到手腕，肘部，肩膀，以及身体的其余部分，它们都沿着拖拽的方向移动。这个例子中，反向运动学将告诉我们所有的这些零件是如何随着拖拽组合成正确的位置。

最好的理解方法就是用例子程序加以说明，每个例子都使用一个关节。我们需要用到 Segment 这个类，因此要保证它在我们工作的工程或类路径中。

单关节伸展

对于伸展而言，所有关节都要能向目标旋转。目标，如果还没读懂我的意思，就把它想成鼠标。让关节向目标旋转，需要知道两点间 x,y 轴上的距离。然后就可以使用 Math.atan2 求出该角度的弧度制。将它转换为角度制，就得到了关节的 rotation。代码如下（可见 OneSegment.as）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class OneSegment extends Sprite {
  private var segment0:Segment;
  public function OneSegment() {
   init();
  }
  private function init():void {
   segment0 = new Segment(100, 20);
   addChild(segment0);
   segment0.x = stage.stageWidth / 2;
   segment0.y = stage.stageHeight / 2;
```

```
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
   }
   private function onEnterFrame(event:Event):void {
    var dx:Number = mouseX - segment0.x;
    var dy:Number = mouseY - segment0.y;
    var angle:Number = Math.atan2(dy, dx);
    segment0.rotation = angle * 180 / Math.PI;
   }
  }
}
```

图 14-1 所示，运行结果。测试一下观察关节是如何跟随鼠标的。即使关节离得很远，它都像是快要抓住鼠标一样。

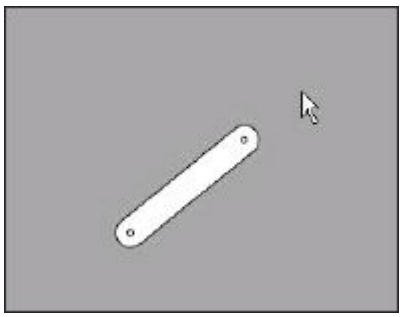


图 14-1 单个关节向鼠标伸展

单关节拖拽

现在，我们来试一试拖拽。这里所说的拖拽不是使用 startDrag 和 stopDrag 方法（虽然你也可以这样做）。我们要假设关节的第二个枢轴点是与鼠标相连的。

拖拽的第一部分与伸展相同：让 sprite 影片向着鼠标旋转。然后我们还要多做一步，将关节移动到可以使第二个枢轴点放到鼠标上的位置。这样一来，就需要知道两个枢轴的每个轴的位置。我们可以通过关节的 getPin() 方法以及关节实际的 x,y 位置，将它们计算出来。把这两个距离叫做 w 和 h 吧。最后，从鼠标的当前位置中将 w 和 h 减去，这样就知道将关节放在哪里了。下面是 OneSegmentDrag.as 中的 onEnterFrame 方法，也是唯一发生改变的部分：

```
private function onEnterFrame(event:Event):void {
 var dx:Number = mouseX - segment0.x;
 var dy:Number = mouseY - segment0.y;
 var angle:Number = Math.atan2(dy, dx);
 segment0.rotation = angle * 180 / Math.PI;
 var w:Number = segment0.getPin().x - segment0.x;
 var h:Number = segment0.getPin().y - segment0.y;
 segment0.x = mouseX - w;
 segment0.y = mouseY - h;
}
```

我们可以看到这个关节永久地与鼠标相连并旋转，拖拽在鼠标的后面。我们甚至可以把这个关节推到相反的方向去。

多关节拖拽

使用反向运动学拖拽一个系统比伸展要简单一些，所以首先介绍拖拽。从两个关节的拖拽入手。

拖拽两个关节

继续前面的例子，再创建一个关节，名为 segment1，然后加入显示列表。策略非常简单。我们已经有了 segment0 拖拽在鼠标上的位置了，只需要再让 segment1 拖拽在 segment0 上即可。首先，简单地复制一些代码，然后改变一些引用。新代码部分加粗表示。

```
private function onEnterFrame(event:Event):void {
  var dx:Number = mouseX - segment0.x;
  var dy:Number = mouseY - segment0.y;
  var angle:Number = Math.atan2(dy, dx);
  segment0.rotation = angle * 180 / Math.PI;
  var w:Number = segment0.getPin().x - segment0.x;
  var h:Number = segment0.getPin().y - segment0.y;
  segment0.x = mouseX - w;
  segment0.y = mouseY - h;
  dx = segment0.x - segment1.x;
  dy = segment0.y - segment1.y;
  angle = Math.atan2(dy, dx);
  segment1.rotation = angle * 180 / Math.PI;
  w = segment1.getPin().x - segment1.x;
  h = segment1.getPin().y - segment1.y;
  segment1.x = segment0.x - w;
  segment1.y = segment0.y - h;
}
```

我们看到新的代码块是如何计算 segment1 到 segment0 的距离，并使用它们计算出 angle 与 rotation 以及 segment1 的位置。不妨测试一下这个例子程序，观察这个非常真实的双关节系统。

现在，有了许多复制的代码，这样不太好。如果要加入更多的关节，这个文件会由于这些相同的重复代码变得越来越长。解决方法是将复制出来的代码单独放到一个名为 drag 的函数中。这个函数需要知道要被拖拽的关节以及要拖拽到的点的 x,y。然后我们就可以拖拽 segment0 到 mouseX, mouseY，以及 segment1 到 segment0.x, segment0.y。全部代码如下（同样出现在 TwoSegmentDrag.as 中）：

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  public class TwoSegmentDrag extends Sprite {
    private var segment0:Segment;
    private var segment1:Segment;
    public function TwoSegmentDrag() {
      init();
    }
    private function init():void {
      segment0 = new Segment(100, 20);
      addChild(segment0);
      segment1 = new Segment(100, 20);
      addChild(segment1);
```

```
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
   }
   private function onEnterFrame(event:Event):void {
    drag(segment0, mouseX, mouseY);
    drag(segment1, segment0.x, segment0.y);
   }
   private function drag(segment:Segment, xpos:Number, ypos:Number):void {
    var dx:Number = xpos - segment.x;
    var dy:Number = ypos - segment.y;
    var angle:Number = Math.atan2(dy, dx);
    segment.rotation = angle * 180 / Math.PI;
    var w:Number = segment.getPin().x - segment.x;
    var h:Number = segment.getPin().y - segment.y;
    segment.x = xpos - w;
    segment.y = ypos - h;
   }
  }
}
```

拖拽更多的关节

现在我们可以任意加入多个关节了。假设放入 6 个关节，命名从 segment0 到 segment1，并把它们存入数组。然后使用 for 循环为每个关节调用 drag 函数。可在 MultiSegmentDrag.as 中找到这个例子。代码如下：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class MultiSegmentDrag extends Sprite {
  private var segments:Array;
  private var numSegments:uint = 6;
  public function MultiSegmentDrag() {
   init();
  }
  private function init():void {
   segments = new Array();
   for (var i:uint = 0; i < numSegments; i++) {
    var segment:Segment = new Segment(50, 10);
    addChild(segment);
    segments.push(segment);
   }
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   drag(segments[0], mouseX, mouseY);
   for (var i:uint = 1; i < numSegments; i++) {
    var segmentA:Segment = segments[i];
    var segmentB:Segment = segments[i - 1];
    drag(segmentA, segmentB.x, segmentB.y);
```

```
    }
   }
   private function drag(segment:Segment, xpos:Number, ypos:Number):void {
    var dx:Number = xpos - segment.x;
    var dy:Number = ypos - segment.y;
    var angle:Number = Math.atan2(dy, dx);
    segment.rotation = angle * 180 / Math.PI;
    var w:Number = segment.getPin().x - segment.x;
    var h:Number = segment.getPin().y - segment.y;
    segment.x = xpos - w;
    segment.y = ypos - h;
   }
  }
}
```

segmentB 是要拖拽到的目标关节，segmentA 则是下一个关节——正在被拖拽的关节。只需要把它们作为参数传给 drag 函数即可。运行结果见图 14-2。

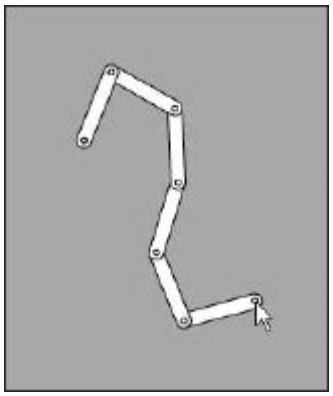


图 14-2 多关节拖拽

现在，大家已经有了反向运动学的基础。不太复杂，哈？想要多少个关节就可以加入多少个，只需要改变 numSegments 变量即可。图 14-3 中可以看到 50 个关节，充分彰显了这个系统是多么强大。

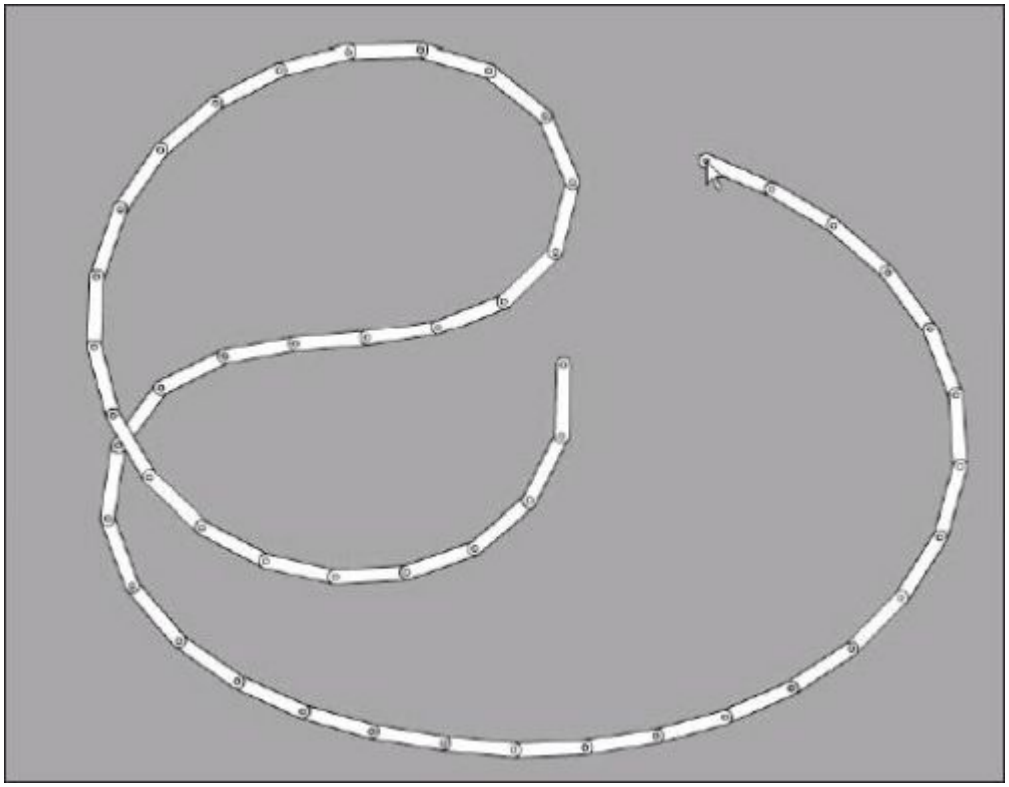


图 14-3 拖拽 50 个关节

多关节伸展运动

反向运动学的伸展，从本章的初始示例 OneSegment.as 开始，并在其基础上加以扩展。这个程序只是将关节向目标旋转，即鼠标的位置。

抓住鼠标

首先，我们需要确定关节接触到目标时的实际位置。同拖拽的计算位置的方法一样。然而，本例中我们并不真正移动关节。只需要找出这个位置。那么获得了这个位置后，能做些什么呢？我们将它作为下一个关节的目标点，并让下一个关节向它旋转。在伸展到这个系统的固定端时，在回退回去，将每一个零件放到其父级的末端。图 14-4 解释说明了这一过程。

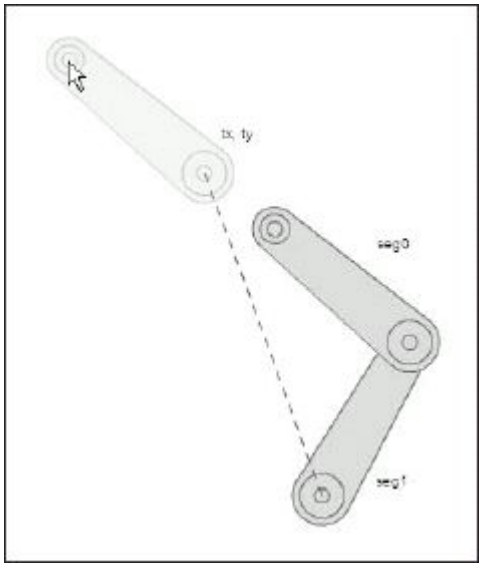


图 14-4 segment0 向着鼠标旋转。tx,ty 是它应该到达的位置。segment1 向着 tx,ty 旋转

本章的第一个文件 OneSegment.as 中只有一个关节，segment0 向鼠标抓去。这里，我们再创建一个关节，名为 segment1，并加入显示列表。接下来要找到目标点，将 segment0 放到目标点上。与拖拽示例中，将关节拖拽到的点是相同的。但是不要移动它，只要保持这个位置。由此得出这个函数：

```
private function onEnterFrame(event:Event):void {
  var dx:Number = mouseX - segment0.x;
  var dy:Number = mouseY - segment0.y;
  var angle:Number = Math.atan2(dy, dx);
  segment0.rotation = angle * 180 / Math.PI;
  var w:Number = segment0.getPin().x - segment0.x;
  var h:Number = segment0.getPin().y - segment0.y;
  var tx:Number = mouseX - w;
  var ty:Number = mouseY - h;
}
```

把这个点叫做 tx,ty，因为它将是 segment1 旋转的目标。

接下来，我们就可以进行复制粘贴，调整旋转代码让 segment1 向目标点旋转：

```
private function onEnterFrame(event:Event):void {
  var dx:Number = mouseX - segment0.x;
  var dy:Number = mouseY - segment0.y;
```

```
  var angle:Number = Math.atan2(dy, dx);
  segment0.rotation = angle * 180 / Math.PI;
  var w:Number = segment0.getPin().x - segment0.x;
  var h:Number = segment0.getPin().y - segment0.y;
  var tx:Number = mouseX - w;
  var ty:Number = mouseY - h;
  dx = tx - segment1.x;
  dy = ty - segment1.y;
  angle = Math.atan2(dy, dx);
  segment1.rotation = angle * 180 / Math.PI;
}
```

新增代码与函数中前四行代码相似，只是使用了不同的关节与目标。

最终，重置 segment0 的位置，它位于 segment1 的末端，因为 segment1 现在已经旋转到了新的位置上。

```
private function onEnterFrame(event:Event):void {
  var dx:Number = mouseX - segment0.x;
  var dy:Number = mouseY - segment0.y;
  var angle:Number = Math.atan2(dy, dx);
  segment0.rotation = angle * 180 / Math.PI;
  var w:Number = segment0.getPin().x - segment0.x;
  var h:Number = segment0.getPin().y - segment0.y;
  var tx:Number = mouseX - w;
  var ty:Number = mouseY - h;
  dx = tx - segment1.x;
  dy = ty - segment1.y;
  angle = Math.atan2(dy, dx);
  segment1.rotation = angle * 180 / Math.PI;
  segment0.x = segment1.getPin().x;
  segment0.y = segment1.getPin().y;
}
```

测试一下这个例子，我们看到两个关节就像一个整体，向着鼠标伸来。

现在，整理一下代码，以便可以轻松地加入更多的关节。首先将所有的 rotation 内容放入一个名为 reach 的函数中。

```
private function reach(segment:Segment, xpos:Number, ypos:Number):Point {
  var dx:Number = xpos - segment.x;
  var dy:Number = ypos - segment.y;
  var angle:Number = Math.atan2(dy, dx);
  segment.rotation = angle * 180 / Math.PI;
  var w:Number = segment.getPin().x - segment.x;
  var h:Number = segment.getPin().y - segment.y;
  var tx:Number = xpos - w;
  var ty:Number = ypos - h;
  return new Point(tx,ty);
}
```

注意函数的返回类型是 Point，最后一行创建并返回了一个基于 tx 和 ty 的 Point。这样就允许我们调用 reach 函数来旋转关节，它将返回目标点，然后就可以传入下一次调用中了。不要忘记导入 Point 类。那么 onEnterFrame 就变成这样：

```
private function onEnterFrame(event:Event):void {
  var target:Point = reach(segment0, mouseX, mouseY);
```

```
  reach(segment1, target.x, target.y);
  segment0.x = segment1.getPin().x;
  segment0.y = segment1.getPin().y;
}
```

segment0 永远向鼠标方向伸展，segment1 向 segment0 伸展。下面我们把这个设置位置的代码放入单独的方法 position 中：

```
private function position(segmentA:Segment, segmentB:Segment):void {
  segmentA.x = segmentB.getPin().x;
  segmentA.y = segmentB.getPin().y;
}
```

最后使用 position(segment0, segment1); 将 segment0 固定到 segment1 的末端上。

下面是最终的代码 TwoSegmentReach.as:

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.geom.Point;
  public class TwoSegmentReach extends Sprite {
    private var segment0:Segment;
    private var segment1:Segment;
    public function TwoSegmentReach() {
      init();
    }
    private function init():void {
      segment0 = new Segment(100, 20);
      addChild(segment0);
      segment1 = new Segment(100, 20);
      addChild(segment1);
      segment1.x = stage.stageWidth / 2;
      segment1.y = stage.stageHeight / 2;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onEnterFrame(event:Event):void {
      var target:Point = reach(segment0, mouseX, mouseY);
      reach(segment1, target.x, target.y);
      position(segment0, segment1);
    }
    private function reach(segment:Segment, xpos:Number, ypos:Number):Point {
      var dx:Number = xpos - segment.x;
      var dy:Number = ypos - segment.y;
      var angle:Number = Math.atan2(dy, dx);
      segment.rotation = angle * 180 / Math.PI;
      var w:Number = segment.getPin().x - segment.x;
      var h:Number = segment.getPin().y - segment.y;
      var tx:Number = xpos - w;
      var ty:Number = ypos - h;
      return new Point(tx,ty);
    }
    private function position(segmentA:Segment, segmentB:Segment):void {
      segmentA.x = segmentB.getPin().x;
```

```
    segmentA.y = segmentB.getPin().y;
   }
  }
}
```

有了这些就可以很容易地创建一个数组，持有许多关节。MultiSegmentReach.as 就是这样做的：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.geom.Point;
 public class MultiSegmentReach extends Sprite {
  private var segments:Array;
  private var numSegments:uint = 6;
  public function MultiSegmentReach() {
   init();
  }
  private function init():void {
   segments = new Array();
   for (var i:uint = 0; i < numSegments; i++) {
    var segment:Segment = new Segment(50, 10);
    addChild(segment);
    segments.push(segment);
   }
   // 将最后一个的位置设置到舞台中心
   segment.x = stage.stageWidth / 2;
   segment.y = stage.stageHeight / 2;
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var target:Point = reach(segments[0], mouseX, mouseY);
   for (var i:uint = 1; i < numSegments; i++) {
    var segment:Segment = segments[i];
    target = reach(segment, target.x, target.y);
   }
   for (i = numSegments - 1; i > 0; i--) {
    var segmentA:Segment = segments[i];
    var segmentB:Segment = segments[i - 1];
    position(segmentB, segmentA);
   }
  }
  private function reach(segment:Segment, xpos:Number, ypos:Number):Point {
   var dx:Number = xpos - segment.x;
   var dy:Number = ypos - segment.y;
   var angle:Number = Math.atan2(dy, dx);
   segment.rotation = angle * 180 / Math.PI;
   var w:Number = segment.getPin().x - segment.x;
   var h:Number = segment.getPin().y - segment.y;
   var tx:Number = xpos - w;
   var ty:Number = ypos - h;
```

```
    return new Point(tx,ty);
   }
   private function position(segmentA:Segment,
   segmentB:Segment):void {
    segmentA.x = segmentB.getPin().x;
    segmentA.y = segmentB.getPin().y;
   }
  }
}
```

运行结果如图 14-5 所示。效果比开始时好多了。但是为什么关节链整天总是追着鼠标跑呢？它似乎有一些自己的意识。让我们看看如果给它一个玩具会发生什么！

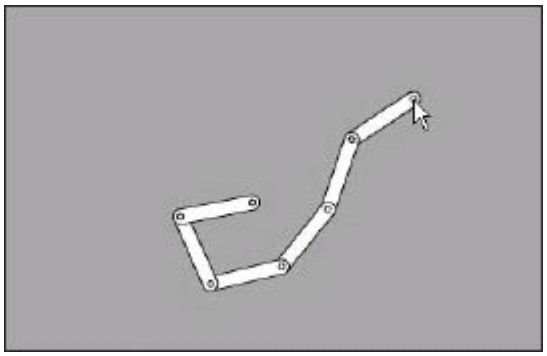


图 14-5 多关节伸展

抓住一个物体

下一个例子，还要重新用到 Ball 类，把它加入到我们的工程或类路径中。然后为小球创建一些新的变量，用于移动。注意这段代码建立在最后一个例子的基础上，我们只需要加入：

```
private var ball:Ball;
private var gravity:Number = 0.5;
private var bounce:Number = -0.9;
```

在 init 方法中，创建一个 Ball 的实例并加入显示列表。

```
private function init():void {
 ball = new Ball();
 ball.vx = 10;
 addChild(ball);
 segments = new Array();
 for (var i:uint = 0; i < numSegments; i++) {
  var segment:Segment = new Segment(50, 10);
  addChild(segment);
  segments.push(segment);
 }
 segment.x = stage.stageWidth / 2;
 segment.y = stage.stageHeight;
 addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

在 onEnterFrame 中调用名为 moveBall 的函数，只是将所有小球的运动代码分离出来，为了看起来不至于混乱：

```
private function onEnterFrame(event:Event):void {
 moveBall();
 var target:Point = reach(segments[0], mouseX, mouseY);
 for (var i:uint = 1; i < numSegments; i++) {
```

```
    var segment:Segment = segments[i];
    target = reach(segment, target.x, target.y);
  }
  for (i = numSegments - 1; i > 0; i--) {
    var segmentA:Segment = segments[i];
    var segmentB:Segment = segments[i - 1];
    position(segmentB, segmentA);
  }
}
```

这就是那个函数：

```
private function moveBall():void {
  ball.vy += gravity;
  ball.x += ball.vx;
  ball.y += ball.vy;
  if (ball.x + ball.radius > stage.stageWidth) {
    ball.x = stage.stageWidth - ball.radius;
    ball.vx *= bounce;
  } else if (ball.x - ball.radius < 0) {
    ball.x = ball.radius;
    ball.vx *= bounce;
  }
  if (ball.y + ball.radius > stage.stageHeight) {
    ball.y = stage.stageHeight - ball.radius;
    ball.vy *= bounce;
  } else if (ball.y - ball.radius < 0) {
    ball.y = ball.radius;
    ball.vy *= bounce;
  }
}
```

然后改变 onEnterFrame 函数中的第二行，让关节去抓住这个小球的实例，而不是鼠标：

```
var target:Point = reach(segments[0], ball.x, ball.y);
```

OK，完成。运行结果如图 14-6 所示。现在小球来回地反弹，而手臂紧紧地跟随它。很疯狂对吗？

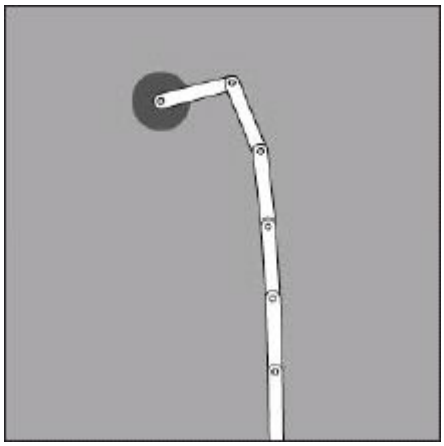


图 14-6 就像在玩儿球

但是，我们还可以做得更好。现在，手臂很好地与小球发生接触，但是小球全然无视手臂的作用。下面来给它们加入一些交互。

加入一些交互

小球与手臂如何交互取决于我们想要什么样的交互。但是，不论做什么，首先需要的就是碰撞检测。然后才能在碰撞时产生交互。同时，我们要将这段内容放到单独的一个函数中去，从 onEnterFrame 中进行调用。

```
private function onEnterFrame(event:Event):void {
 moveBall();
 var target:Point = reach(segments[0], ball.x, ball.y);
 for (var i:uint = 1; i < numSegments; i++) {
  var segment:Segment = segments[i];
  target = reach(segment, target.x, target.y);
 }
 for (i = numSegments - 1; i > 0; i--) {
  var segmentA:Segment = segments[i];
  var segmentB:Segment = segments[i - 1];
  position(segmentB, segmentA);
 }
 checkHit();
}
```

我将这个函数命名为 checkHit，并将它放到主函数的最后，因此在调用它时所有的位置都已经完成了动作。

下面开始 checkHit 函数：

```
public function checkHit():void {
 var segment:Segment = segments[0];
 var dx:Number = segment.getPin().x - ball.x;
 var dy:Number = segment.getPin().y - ball.y;
 var dist:Number = Math.sqrt(dx * dx + dy * dy);
 if (dist < ball.radius) {
  // 从这里开始交互
 }
}
```

首先获得第一支手臂的枢轴端到小球的距离，并使用距离碰撞检测来判断是否与小球产生了碰撞。

下面，回到刚才那个问题，当碰撞发生时应该做些什么。以下是我的计划：手臂会把小球抛到空中（负 y 速度），并且随机地在 x 轴上进行移动（随机 x 速度），程序如下：

```
public function checkHit():void {
 var segment:Segment = segments[0];
 var dx:Number = segment.getPin().x - ball.x;
 var dy:Number = segment.getPin().y - ball.y;
 var dist:Number = Math.sqrt(dx * dx + dy * dy);
 if (dist < ball.radius) {
  ball.vx += Math.random() * 2 - 1;
  ball.vy -= 1;
 }
}
```

效果非常好，最终的代码可在 PlayBall.as 中找到。我居然让它自己运行了一整夜，第二天早上，手臂还在玩它的玩具！但是不要这个程序当作任何的“标准”。我们还可以让手臂抓住小球并向目标投掷，或者像一个篮球游戏？或者让两个手臂相互传球？总之，就是进行不同的交互。大家手头儿上肯定很多的玩具，现在就可以让它们在这儿做些有趣的事情了。

使用标准的反向运动学方法

我要对您说老实话。前面所描述的计算反向运动学的方法完全都是我自创的。在我第一次做出这些效果时，我甚至还不知道这就叫做反向运动学。我只是想让某件东西去抓住其它的物体，为了实现这个目的，就不得不让每个部件各自做一些事情，就在无意间，将它制作成了一个系统，可以轻松地复制或加入其它物体。程序运行得非常好，效果也很不错，也没有毁掉 CPU，我对它很满意，希望您也一样。听到这里您也许有些惊讶，但我不是第一个思考这个问题的人。一些高智商并且受过更多数学公式训练的人，已经解决了这个问题并总结出了可选方案，它们也许更加符合自然界物体的真实运动。所以让我们看一下“标准”的反向运动学方法。随后，我们会有两种不同的方法可以自由选择。

余弦定理介绍

反向运动学常用的方法，如本节标题所示，叫做余弦定理。又是三角学？是的。回忆一下第三章，所有的例子都使用直角三角形——三角形中有一个角是直角（90 度）。这样的三角形规则非常简单：正弦等于对边比斜边，余弦等于邻边比斜边，等等。整本书中我都广泛地应用了这套规则。

但是，如果有一个三角形没有 90 度的角，该怎么办呢？我们这就样被排斥了吗？肯定不会，一个古希腊人也想到了这一点，并给出了余弦定理来帮助我们计算出各种三角形的角度以及边长。当然，这个定理会有些复杂，但是如果您对三角有足够的了解，那么您就可以给出解决方案了。

大家也许会问“这玩艺儿对反向运动学有什么用？”好，请看图 14-7。

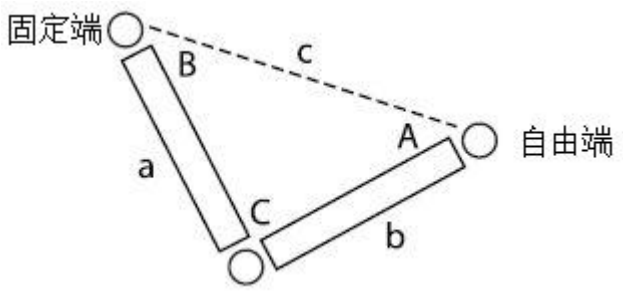


图 14-7 两个关节形成的一个三角形，a, b, c 为三条边，A, B, C 为三个角

这里有两个关节。左边的那个是固定端。它是固定住的，因此它的位置是已知的。我们要将自由端放到图中标出的位置。这样就形成了一个任意三角形。

已知条件有哪些呢？我们可以轻松地求出两个端点间的距离—— c 边。还知道每个关节的长度—— a, b 边。因此就知道了所有三条边的长度。

我们需要知道这个三角形的什么呢？只需要知道两个关节的两个角度—— 角 B 和 C。这是由余弦定理帮助我们做的。下面我给大家介绍一下：

```
c2 = a2 + b2 - 2 * a * b * cos C
```

现在，我们要知道角 C，所以可以将它从一条边中分离出来。这里就不一一列出每一步了，因为这是基本的代数学。最终得到：

```
C = acos ((a2 + b2 - c2) / (2 * a * b))
```

其中，acos 是反余弦函数。角的余弦给我们一个比率，或小数。而这个比率的反余弦，则会反过来给出角度。Flash 中函数表示为 Math.acos()。只要我们知道 a, b, c 边，就可以求出角 C。同理，还需要知道角 B。余弦定理是这样说的：

```
b2 = a2 + c2 - 2 * a * c * cos B
```

化简后得出：

B = acos((a2 + c2 - b2)/ (2 * a * c))

转化成 ActionScript 就是：

B = Math.acos((a * a + c * c - b * b) / (2 * a * c));

C = Math.acos((a * a + b * b - c * c) / (2 * a * b));

现在我们几乎知道了所有需要设置物体位置的条件。之所以说几乎，是因为角 B 和 C 并不是关节真正的 rotation。请看下一张图 14-8。

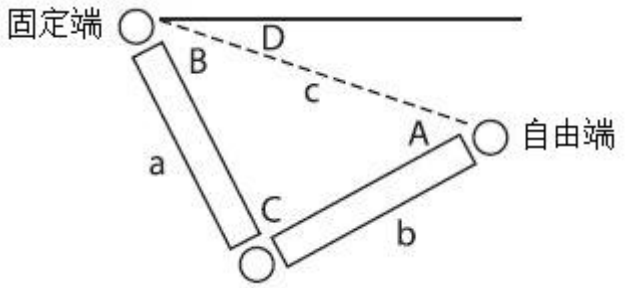


图 14-8 计算 seg1 的 rotation

角 B 求出来了，我们现在需要知道 seg1 实际要旋转多少。这个角度是从 0 或水平面开始的，应该是角 B 与 D 之和。幸运的是，我们可以通过计算固定端与自由端的夹角得出角 D，如图 14-9 所示。

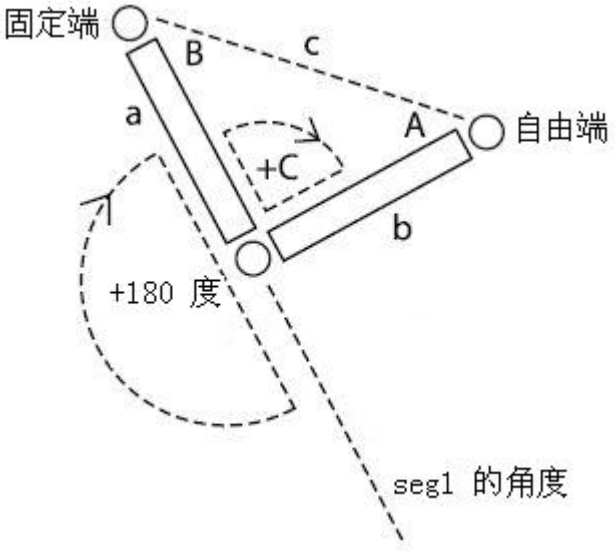


图 14-9 计算 seg0 的 rotation

我们求得了角 C，但它只是相对于 seg1 的，需要将 seg1 的 rotation 加上 180 再加上 C。我将这个角叫做角 E。

OK，讲得够多了。让我们来看代码，这样就更加清晰了。

ActionScript 余弦定理

先给出大段的反向运动学的代码，稍后进行解释。以下是代码（可在 Cosines.as 中找到）：

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.geom.Point;
  public class Cosines extends Sprite {
    private var segment0:Segment;
    private var segment1:Segment;
```

```
  public function Cosines() {
    init();
  }
  private function init():void {
    segment0 = new Segment(100, 20);
    addChild(segment0);
    segment1 = new Segment(100, 20);
    addChild(segment1);
    segment1.x = stage.stageWidth / 2;
    segment1.y = stage.stageHeight / 2;
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
    var dx:Number = mouseX - segment1.x;
    var dy:Number = mouseY - segment1.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    var a:Number = 100;
    var b:Number = 100;
    var c:Number = Math.min(dist, a + b);
    var B:Number = Math.acos((b * b - a * a - c * c) / (-2 * a * c));
    var C:Number = Math.acos((c * c - a * a - b * b) / (-2 * a * b));
    var D:Number = Math.atan2(dy, dx);
    var E:Number = D + B + Math.PI + C;
    segment1.rotation = (D + B) * 180 / Math.PI;
    segment0.x = segment1.getPin().x;
    segment0.y = segment1.getPin().y;
    segment0.rotation = E * 180 / Math.PI;
  }
 }
}
```

下面是这个过程：

1. 获得 segment1 到鼠标的距离。
2. 获得三条边的长度。 a, b 边很简单。它们都等于 100，因为这就是我们创建关节时所给的长度。（大家可以删掉 100 这个数，让代码变得更加动态；这里我只是为了简洁。） c 边等于距离或 a + b 中最小的值。这是因为三角形中的一条边不能大于其它两条边之和。如果不相信，请试着画出一个这样的图形。如果从固定端到鼠标的距离是 200，而两个关节的长度加起来只有 120，就不能使用距离作为边长。
3. 使用余弦定理计算出角 B 和 C，再用 Math.atan2 计算出角 D。角 E，如同我们前面提到的，它等于 D + B + 180 + C。当然，在代码中要用 Math.PI 弧度代替 180。
4. 如图 14-9 中，将 D + B 角转换为角度制，就得到了 seg1 的 rotation。接着计算出 seg1 的末端，并将 seg0 放置在此。
5. 最后，seg0 的 rotation 等于角 E，转换为角度制。

这样我们就有了：使用余弦定理的反向运动学。大家也许注意到了连接处永远只向一个方向弯曲。这对于要建立一个肘部或膝关节也许是件好事，因为它们只能向一个方向弯曲。

当我们要可以特意计算出这样的角度时，那么这个问题就有两种解决方法：向一个方向弯曲，或向另一个方向弯曲。我们已经计算过一个方向的弯曲，通过加 D 和 B，再加 C 来完成。如果将它们全部减去，效果相同，但是手臂的弯曲方向是相反的。

```
private function onEnterFrame(event:Event):void {
 var dx:Number = mouseX - segment1.x;
```

```
  var dy:Number = mouseY - segment1.y;
  var dist:Number = Math.sqrt(dx * dx + dy * dy);
  var a:Number = 100;
  var b:Number = 100;
  var c:Number = Math.min(dist, a + b);
  var B:Number = Math.acos((b * b - a * a - c * c) / (-2 * a * c));
  var C:Number = Math.acos((c * c - a * a - b * b) / (-2 * a * b));
  var D:Number = Math.atan2(dy, dx);
  var E:Number = D - B + Math.PI - C;
  segment1.rotation = (D - B) * 180 / Math.PI;
  segment0.x = segment1.getPin().x;
  segment0.y = segment1.getPin().y;
  segment0.rotation = E * 180 / Math.PI;
}
```

如果要想让两个方向都能弯曲，我们就需要加入一些逻辑条件，比如“如果在这个位置，则向这个方向弯曲；反之，向另一个方向弯曲。”不幸的是，我只能个大家一些简短的余弦定理的介绍。如果大家对这方面感兴趣，我相信您一定还会找更多相关的知识。快速的网络搜索“反向运动学”给了出多达 90,000 条搜索结果。是啊，我们一定能从中挖掘出一些东西来！

本章重要公式

标准的反向运动学形式，使用余弦定理公式。

余弦定理：

a2 = b2 + c2 - 2 * b * c * cos A

b2 = a2 + c2 - 2 * a * c * cos B

c2 = a2 + b2 - 2 * a * b * cos C

ActionScript 的余弦定理：

A = Math.acos((b * b + c * c - a * a) / (2 * b * c));

B = Math.acos((a * a + c * c - b * b) / (2 * a * c));

C = Math.acos((a * a + b * b - c * c) / (2 * a * b));

14.6 小结

反向运动是一个非常广的课题——远远不是使用一章就可以说完的。尽管如此，我认为本章描述了一些相当酷和有用的内容。你知道了如何设置一个反向运动系统和看它的两种方式：拖动和伸展。我希望至少和你在在这个课题能碰撞出一些火花。在 Flash 中可以制作许多反向运动的内容，我相信你已经准备开始探索并应用它了。

在下一章，你将进入一个全新的维度，它允许你在影片中加入深度。是的，我们将学习 3D。

第十五章　3D 基础

前面我们做的一切都是二维的（有时只有一维），但是已经可以做出非常酷的东东了。现在，将它们带入到下一个等级。

创建 3D 图形总是那么另人兴奋。新加入的这个维度似乎将物体真正地带入到了生活中。如何在 Flash 中实现 3D 在无数的书籍和教学软件中都有介绍。但是我不打算跳过这些内容，我们会很快地将所有基础的知识讲完。随后，将前面章节中讨论的运动效果放到三维空间中。说得详细些，将给大家介绍速度，加速度，摩擦力，反弹，屏幕环绕，缓动，弹性运动，坐标旋转以及碰撞检测。

现在，首先要关注 sprite 影片在 3D 空间中运动，使用透视法计算影片在屏幕上的大小和位置。当然，sprite 本身是平面的，我们看不到它的背面，侧面，顶面或底面。在后面两章，我们将学习到点，线，图形和立体图形的 3D 建模。

第三维度及透视法

在 3D 背后最重要的理论就是超出 x 和 y 存在的另一个维度。这是表示深度的维度，通常记为 z。

Flash 没有内置的 z 维度，但是要想在 ActionScript 中创建它也不是件难事。实际上，远没有我们前面章节中的内容那么复杂！

z 轴

首先，需要确定 z 最是朝哪个方向的：向内或向外。回忆一下第三章讨论的坐标系统，它比普通的坐标系统在某些地方是相反的。y 轴向下，而非向上，角度则是以顺时针方向而定的，而非逆时针方向。

因此，当物体远离或接近我们的时候，是否应该让物体 z 轴上位置增加？没有必要去比较哪个更正确。事实上，这个课题已经被讨论许久了，人们甚至为了描述这两种方法分别给它们取了名字：左手系统和右手系统。

伸出您的右手，让拇指与食指构成一个 L 形，然后将中指弯曲 90 度，每个手指都将指向一个维度。现在，将您的食指指向 x 轴的正半轴，中指指向 y 轴的正半轴。在右手坐标系中，拇指的指向就是 z 轴的正半轴方向。对于 Flash 而言，意味着物体远离观察者时 z 轴将增大，临近观察者时 z 轴将减小，如图 15-1 所示。

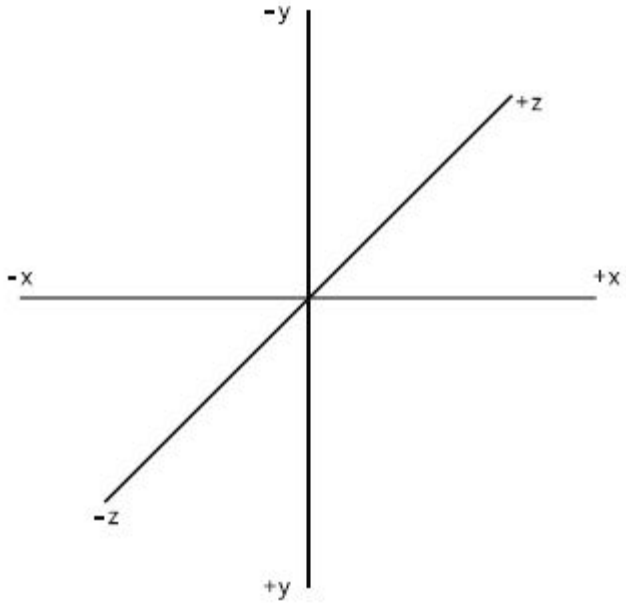


图 15-1 右手坐标系

如果我们用左手来试的话，得到的结果则是相反的。如图 15-2 所示，左手坐标系。

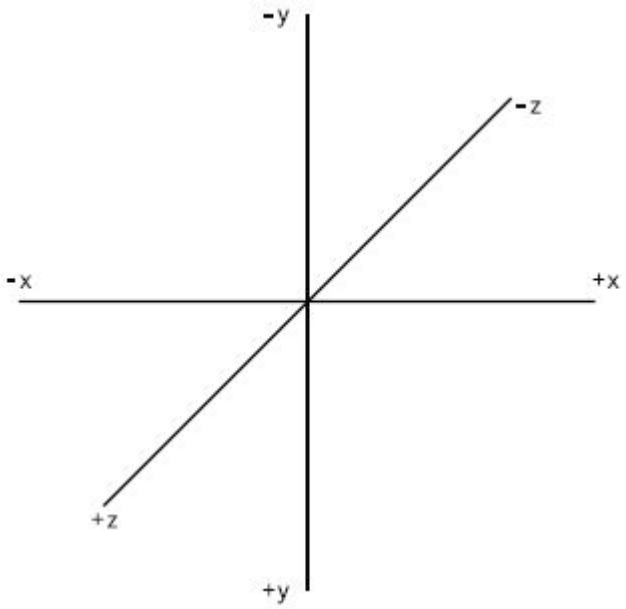


图 15-2 左手坐标系

下面我们使用右手坐标系为例（图 15-1）。没有理由说不能使用左手坐标系，只不过让 z 轴向内看起来比较好。在 Flash 中创建第三维度（z）的下一个步骤是如何计算模拟透视。

透视法

透视法是指如何表述物体接近或远离我们时的方法。换句话讲，如何让物体看起来更近或更远。一幅美术作品中可能有大量的表现透视的技巧，这里我们只关注两点：

- 当物体离得远时，会变小。
- 当物体远离时，它们会聚集到一个消失点上。

大家肯定见过火车驶向地平线时的景象。当我们在 z 轴上移动物体时，需要做两件事：

- 增大或减小物体的比率。
- 让物体接近或远离消失点。

在二维系统中，我们可以使用屏幕的 x 和 y 坐标作为物体的 x 和 y 坐标。只需要一对一地映射过来即可。但是在 3D 系统中就行不通了，因为两个物体可以有相同的 x, y

坐标，由于它们的深度不同，会使它们在屏幕上有不同的位置。因此，在 3D 空间中移动每个物体都需要知道它们各自的 x, y, z 坐标，这是屏幕坐标不能做到的。现在就要用到这三个量来描述虚拟空间的一个位置。透视法将告诉我们应该将物体放到屏幕的什么位置。

透视公式

让物体的距离更远（增加 z），基本思想是让它的缩放比率接近 0，让它的 x, y 坐标集中到消失点的 0,0 处。幸好，缩放的比率与汇集的比率相同。因此，我们只需要根据给定的距离计算出这个比率，然后在这两个地方使用它即可。图 15-3 帮助大家解释这个概念。

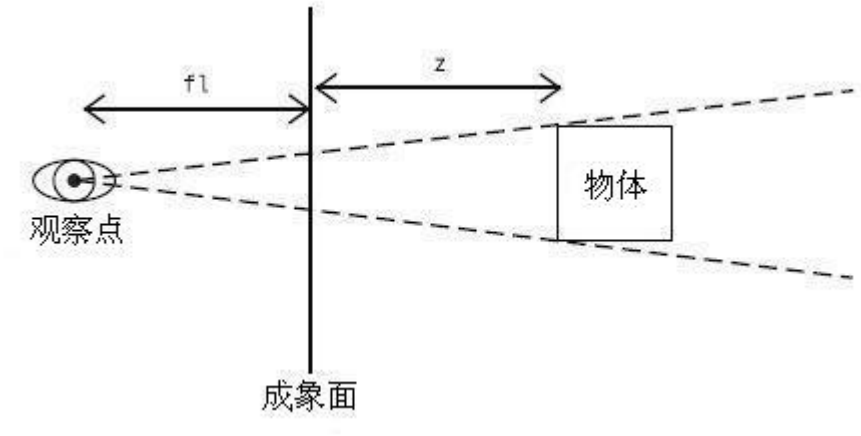


图 15-3 从侧面观察透示图

我们距离对象有一段距离。有一个观察点：眼睛。有一个成象面，可以想象成电脑的屏幕。对象与成象面之间有一段距离，这就是 z 的值。最后，距离观察点到成象面还有一段距离。最后这点最为重要。虽然这段距离不完全等同于摄象机的焦距，但是与它基本相似，因此我通常用变量 fl [焦距：focal length] 表示。下面是这个公式：

scale = fl / (fl + z)

scale 值通常是介于 0.0 到 1.0 之间的，这就是缩放和汇聚到消失点上的比率。然而，当 z 变为负数时，fl + z 接近 0 而缩放比例接近无穷大。

拿到这个 scale 的值能做些什么呢？假设在处理一个影片（或 Sprite 的子类），我们将这个值赋给影片的 scaleX 和 scaleY。然后再用这个因数乘以物体的 x,y 坐标，就可以算出物体在屏幕上的 x,y 的位置。

看一个例子。通常情况下 fl 的值在 200 到 300 之间。我们选用 250 这个值。如果 z 等于 0 ——换句话讲，物体就在成象面上—— 那么 scale 就等于 250 / (250 + 0)。结果等于 1.0。这就是 scaleX 和 scaleY 的值（别忘了对于 scaleX 和 scaleY 而言， 1.0 就意味着 100%）。让物体的 x,y 坐标乘以 1.0，返回的结果不变，因此物体在屏幕上的位置就等于它自身的 x 和 y。

现在将物体向外移让 z 等于 250。则让 scale 等于 250 / (250 + 250)，scaleX 和 scaleY 等于 0.5。同样也改变了物体在屏幕上的位置。如果原来物体在屏幕上的位置是 200, 300 那么现在就应该是 100, 150。因此，它向着消失点移动了一半的距离。（事实上，屏幕上的位置是相对于消失点的位置而定的，大家马上会看到）。

现在，将 z 向外移动到 9750。scale 变成 250 / 10000， scaleX 和 scaleY 等于 0.025。物体将变成一个小点儿，并且非常接近消失点。

OK，理论够了。来看代码。

ActionScript 透视

各位也许猜到了，我还要使用 Ball 类。当然，您也可以自由地选择自己喜欢物体，但是我只专注于代码，将那些酷酷的图形留给大家去做。我们用鼠标和键盘作为交互。使用鼠

标控制小球的 x,y 坐标，方向键的上下键来控制 z 轴的前后方向。注意，因为变量 x,y 是由 ActionScript 持有的，因此我们将使用 xpos, ypos, zpos 代表 3D 坐标。

文档类 Perspective1.as 的代码如下：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.events.KeyboardEvent;
 import flash.ui.Keyboard;
 public class Perspective1 extends Sprite {
  private var ball:Ball;
  private var xpos:Number = 0;
  private var ypos:Number = 0;
  private var zpos:Number = 0;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
  private var vpY:Number = stage.stageHeight / 2;
  public function Perspective1() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
   stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
  }
  private function onEnterFrame(event:Event):void {
   xpos = mouseX - vpX;
   ypos = mouseY - vpY;
   var scale:Number = fl / (fl + zpos);
   ball.scaleX = ball.scaleY = scale;
   ball.x = vpX + xpos * scale;
   ball.y = vpY + ypos * scale;
  }
  private function onKeyDown(event:KeyboardEvent):void {
   if (event.keyCode == Keyboard.UP) {
    zpos += 5;
   } else if (event.keyCode == Keyboard.DOWN) {
    zpos -= 5;
   }
  }
 }
}
```

首先创建变量 xpos, ypos, zpos, fl。然后创建一个消失点（vanishing point）vpX, vpY。记住当物体向远处运动一段距离后，就会聚在 0, 0 点。如果不进行偏移，所有物体都会向屏幕左上角汇集，这并不我们想要的结果。将 vpX, vpY 设置为舞台的中心点，作为消失点。

接下来，在 onEnterFrame 中设置 xpos 和 ypos 为鼠标与消失点的偏移位置。换句话讲，如果鼠标在中心点右面 200 像素，x 就等于 200。如果在中心点左面 200 像素的位置，则等于 -200。

然后添加 keyDown 事件的侦听，用于改变 zpos。如果方向键上被按下 zpos 增加，

如果方向键上被按下则减小。这将使小球向着观察者更近或更远的方向运动。

最后，使用刚刚介绍过的公式计算 scale，设置小球的位置与大小。注意小球在屏幕上的位置 x,y 是根据消失点计算的，还要加上 xpos, ypos 与 scale 的乘积。因此，当 scale 变得很小时，小球将汇集到消失点上。

测试一下影片，开始看起来像一个简单的鼠标拖拽。这是因为 zpos 等于 0，scale 等于 1.0。所以注意不到透视的存在。当按下方向键上时，小球向内滑入一段距离，如图 15-4 所示。现在当我们移动鼠标时，小球也会随之移动，但是移动的距离很小，产生了视差效应。

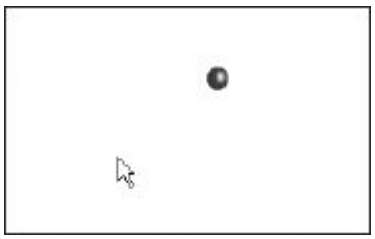


图 15-4 ActionScript 透视

大家也许注意到了，如果长期按住方向键下，小球会变得非常大。这是对的。如果拿起一小块石子放到眼前，它就会像一块巨石一样大。如果继续按住方向键下，它将变成无限大，然后又收缩回去，但是这时整个小球已经颠倒或反转过来了。小球跑到了观察点的后面。因此，如果眼睛可以看到身背后的东西，我猜这一定是我们所看到的。

用数字解释一下，当 zpos 等于 －fl 时，公式从 scale = fl / (fl + zpos) 变为 scale = fl / 0。在许多语言中，除以 0 会报错。在 Flash 中，将得到一个无限大的值。如果再将 zpos 减小，那么就是用 fl 除以一个负数。scale 变为负数，这就是为什么小球会颠倒并反向运动的原因。

学会了吗？解决方法只需在小球在超过某一点时将其设置为不可见的。如果 zpos 小于或等于 －fl，会出现问题，因此可以判断一下这个条件，并在下面这个 Perspective2.as 中的 enterFrame 函数中进行处理（其余部分与 Perspective1.as 完全相同）：

```
private function onEnterFrame(event:Event):void {
 if (zpos > -fl) {
  xpos = mouseX - vpX;
  ypos = mouseY - vpY;
  var scale:Number = fl / (fl + zpos);
  ball.scaleX = ball.scaleY = scale;
  ball.x = vpX + xpos * scale;
  ball.y = vpY + ypos * scale;
  ball.visible = true;
 } else {
  ball.visible = false;
 }
}
```

注意，如果小球不可见，我们就不必考虑缩放和位置问题了。同样还要注意如果小球处于可见的范围，就要确保它是可见的。虽然可能略些多余的设置，但这是必要的。

好的，现在我们已经学习了 3D 基础的框架。不是很痛苦吧？一定要测试一下这个影片，能够很好地掌握它。试改变 fl 的值，观察不同的效果。这就相当于在改变照相机的镜头。较高的 fl 值就像一个长焦镜头，给我们一个较小的观察空间，以及较少的可见的透视。较小的 fl 值将给我们一个广角镜头，形成非常广阔的透视。

本章剩下的部分都是前面章节中介绍过的不同的运动效果，只不过这次是三维的。

速度与加速度

实现 3D 的速度与加速度超级简单。对于 2D 而言，我们用 vx 和 vy 变量表示两个轴的速度。现在只需要再加入 vz 表示第三个轴即可。同样，如果有 ax 和 ay 作为加速度，那么再添加一个 az 变量即可。

我们可以将最后一个例子改为小行星太空船这样的游戏，不过是 3D 版的。将它变为全键盘控制的。方向键可以提供 x,y 轴上的推进，再加入一对儿键 Shift 和 Ctrl 用于 z 轴上的推进。

以下是代码（同样可在 Velocity3D.as 中找到）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.events.KeyboardEvent;
 import flash.ui.Keyboard;
 public class Velocity3D extends Sprite {
  private var ball:Ball;
  private var xpos:Number = 0;
  private var ypos:Number = 0;
  private var zpos:Number = 0;
  private var vx:Number = 0;
  private var vy:Number = 0;
  private var vz:Number = 0;
  private var friction:Number = .98;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
  private var vpY:Number = stage.stageHeight / 2;
  public function Velocity3D() {
   init();
  }
  private function init():void {
   ball = new Ball();
   addChild(ball);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
   stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
  }
  private function onEnterFrame(event:Event):void {
   xpos += vx;
   ypos += vy;
   zpos += vz;
   vx *= friction;
   vy *= friction;
   vz *= friction;
   if (zpos > -fl) {
    var scale:Number = fl / (fl + zpos);
    ball.scaleX = ball.scaleY = scale;
    ball.x = vpX + xpos * scale;
    ball.y = vpY + ypos * scale;
    ball.visible = true;
   } else {
    ball.visible = false;
   }
```

```
    }
    private function onKeyDown(event:KeyboardEvent):void {
      switch (event.keyCode) {
        case Keyboard.UP :
          vy -= 1;
          break;
        case Keyboard.DOWN :
          vy += 1;
          break;
        case Keyboard.LEFT :
          vx -= 1;
          break;
        case Keyboard.RIGHT :
          vx += 1;
          break;
        case Keyboard.SHIFT :
          vz += 1;
          break;
        case Keyboard.CONTROL :
          vz -= 1;
          break;
        default :
          break;
      }
    }
  }
}
```

我们所要做的就是为每个轴加入速度和摩擦力。当六个键中有一个被按下，将会对速度进行适当的增加或减少（记住加速度改变速度）。然后将速度加到每个轴上，最后计算摩擦力。现在我们就有了带有加速度，速度和摩擦力的一个 3D 物体。哇，真是一举多得。说过这很简单。

反弹

本节我们将讨论平面反弹的问题——换句话讲，是与 x, y, z 轴充分结合的反弹，与 2D 的屏幕边界反弹相似。

单物体反弹

3D 反弹，同样需要判断物体何时超出了边界，然后将物体调整到边界上，把相应轴上的速度反转。3D 反弹唯一的不同之处在于如何确定边界。在 2D 中，一般都使用舞台的坐标或其它一些可见的矩形区域。在 3D 中，就不那么简单了。这里没有真正的可见边界的概念，除非在三维空间中绘制一个。我们将在下一章学习三维空间中的绘制，因此现在将在不可见的随意放置的墙壁上进行反弹。

我们设置的边界和以前相同，只不过现在要把它们放到三维空间中，也就意味着可以是正的也可以是负的。还可以选择在 z 轴上设置边界。边界大概是这样：

```
private var top:Number = -250;
```

```
private var bottom:Number = 250;
private var left:Number = -250;
private var right:Number = 250;
private var front:Number = 250;
private var back:Number = -250;
```

接下来，确定物体的新位置，需要判断是否所与这六个边界产生了碰撞。别忘了我们是用物体一半的宽度来判断碰撞的，而这个值已经存在了 Ball 类名为 radius 的变量中。以下是全部 3D 反弹的代码（可见 Bounce3D.as）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Bounce3D extends Sprite {
  private var ball:Ball;
  private var xpos:Number = 0;
  private var ypos:Number = 0;
  private var zpos:Number = 0;
  private var vx:Number = Math.random() * 10 - 5;
  private var vy:Number = Math.random() * 10 - 5;
  private var vz:Number = Math.random() * 10 - 5;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
  private var vpY:Number = stage.stageHeight / 2;
  private var top:Number = -100;
  private var bottom:Number = 100;
  private var left:Number = -100;
  private var right:Number = 100;
  private var front:Number = 100;
  private var back:Number = -100;
  public function Bounce3D() {
   init();
  }
  private function init():void {
   ball = new Ball(15);
   addChild(ball);
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   xpos += vx;
   ypos += vy;
   zpos += vz;
   var radius:Number = ball.radius;
   if (xpos + radius > right) {
    xpos = right - radius;
    vx *= -1;
   } else if (xpos - radius < left) {
    xpos = left + radius;
    vx *= -1;
   }
   if (ypos + radius > bottom) {
```

```
      ypos = bottom - radius;
      vy *= -1;
    } else if (ypos - radius < top) {
      ypos = top + radius;
      vy *= -1;
    }
    if (zpos + radius > front) {
      zpos = front - radius;
      vz *= -1;
    } else if (zpos - radius < back) {
      zpos = back + radius;
      vz *= -1;
    }
    if (zpos > -fl) {
      var scale:Number = fl / (fl + zpos);
      ball.scaleX = ball.scaleY = scale;
      ball.x = vpX + xpos * scale;
      ball.y = vpY + ypos * scale;
      ball.visible = true;
    } else {
      ball.visible = false;
    }
   }
  }
}
```

注意，我删掉了所有按键处理的部分，只让小球以随机的速度在每个轴上运动。现在可以看到小球按照我们旨意进行反弹，但是谁也说不上来反弹在什么东西上——正如我所说的，这些是任意放置不可见的边界。

多物体反弹

让更多的物体充满整个空间也是对我们看出这些墙壁会有些帮助。为了完成这个目的，需要很多 Ball 类的实例。每个实例都要有自己的 xpos, ypos, zpos 以及每个轴的速度。为了让主类（main class）的结构清晰，下面创建了一个新的类 Ball3D，来看一下：

```
package {
 import flash.display.Sprite;
 public class Ball3D extends Sprite {
  public var radius:Number;
  private var color:uint;
  public var xpos:Number = 0;
  public var ypos:Number = 0;
  public var zpos:Number = 0;
  public var vx:Number = 0;
  public var vy:Number = 0;
  public var vz:Number = 0;
  public var mass:Number = 1;
  public function Ball3D(radius:Number=40, color:uint=0xff0000) {
   this.radius = radius;
```

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

我们看到，这里所做的就是加入了每个轴的位置和速度的属性。同样，将类中的属性设置为 public 实在不是一个好的面向对象程序设计的习惯，但是现在我们只是为了能够简单地说明公式才这么做的。在 MultiBounce3D.as 中，创建了 50 个新类的实例。每个实例都有自己的 xpos, ypos, zpos, vx,vy, vz。 onEnterFrame 方法循环获得每个 Ball3D 的引用，然后将它们传给 move 方法。这个方法与最初的 onEnterFrame 完成的功能相同。代码如下（可在 MultiBounce3D.as 中找到）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class MultiBounce3D extends Sprite {
  private var balls:Array;
  private var numBalls:uint = 50;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
  private var vpY:Number = stage.stageHeight / 2;
  private var top:Number = -100;
  private var bottom:Number = 100;
  private var left:Number = -100;
  private var right:Number = 100;
  private var front:Number = 100;
  private var back:Number = -100;
  public function MultiBounce3D() {
   init();
  }
  private function init():void {
   balls = new Array();
   for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball3D = new Ball3D(15);
    balls.push(ball);
    ball.vx = Math.random() * 10 - 5;
    ball.vy = Math.random() * 10 - 5;
    ball.vz = Math.random() * 10 - 5;
    addChild(ball);
   }
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball3D = balls[i];
```

```
        move(ball);
      }
    }
    private function move(ball:Ball3D):void {
      var radius:Number = ball.radius;
      ball.xpos += ball.vx;
      ball.ypos += ball.vy;
      ball.zpos += ball.vz;
      if (ball.xpos + radius > right) {
        ball.xpos = right - radius;
        ball.vx *= -1;
      } else if (ball.xpos - radius < left) {
        ball.xpos = left + radius;
        ball.vx *= -1;
      }
      if (ball.ypos + radius > bottom) {
        ball.ypos = bottom - radius;
        ball.vy *= -1;
      } else if (ball.ypos - radius < top) {
        ball.ypos = top + radius;
        ball.vy *= -1;
      }
      if (ball.zpos + radius > front) {
        ball.zpos = front - radius;
        ball.vz *= -1;
      } else if (ball.zpos - radius < back) {
        ball.zpos = back + radius;
        ball.vz *= -1;
      }
      if (ball.zpos > -fl) {
        var scale:Number = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
      } else {
        ball.visible = false;
      }
    }
  }
}
```

运行这个文件后，可以看到小球将六个边界内的大部空间都填满了，如图 15-5 所示，这样我们就可以看出这个空间的形状了。

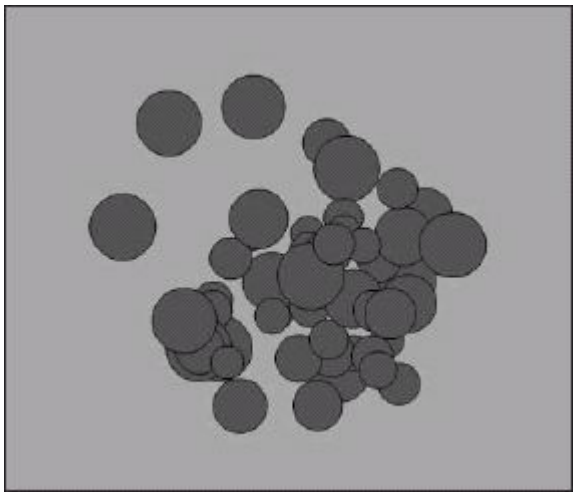


图 15-5 3D 小球反弹

Z 排序

在添加了多个物体后代码中显现出了一个新问题——称为 z 排序。Z 排序就像它的名字一样：物体如何在 z 轴上进行排序，或者说哪个物体在前面。由于物体都使用纯色，所以看起来不是很明显。为了让效果更加明显，请将 Ball3D 的 init 方法改为以下代码，并运行刚才那个程序：

```
public function init():void {
 graphics.lineStyle(0);
 graphics.beginFill(color);
 graphics.drawCircle(0, 0, radius);
 graphics.endFill();
}
```

通过给小球添加轮廓线，我们就可以看出哪个小球在前面了。这几乎毁掉了整个 3D 效果，因为现在较小的物体出现在了较大物体的前面。Z 排序就是用来解决这个问题的，但不是自动的。Flash 不知道我们在模拟 3D。它只知道我们在移动和缩放影片。它也不知道我们到底是使用左手还是右手坐标系。在小球远离时应该将这个小球放在相邻小球的后面。Flash 只根据在显示列表中的相对索引进行排序。在 AS 2 中，z 排序只需要改变影片剪辑的深度即可完成。swapDepths(深度)。深度较高的影片剪辑出现在深度较低的影片的前面。然而在 AS 3 中，操作会稍微有些复杂。对于显示列表没有可以任意修改的深度值。显示列表的作用更像是与个数组。列表中的每个显示对象都有一个索引。索引从 0 开始，直到列表中所有对象的个数。例如，假设在类中加入三个影片 A, B, C。它们的索引应该是 0, 1, 2。无法将其中的一个影片的索引设置为 100，或 -100。如果已经删除了 B 影片，那么这时 A 和 C 影片的索引应该是 0 和 1。明白了吧，在显示列表中永远没有“空间”这个概念。

根据深度，索引 0 是最低的，任何深度较高的显示对象都将出现在这个较低对象的前面。我们可以用几种不同的方法来改变物体的深度：

- setChildIndex(child:DisplayObject, index:int) 给对象指定索引值（index）。
- swapChildren(child1:DisplayObject, child2:DisplayObject) 交换两个指定的对象。
- swapChildrenAt(index1:int, index2:int) 交换两个指定的深度。

使用 setChildIndex 是最简单的。因为我们已经有了一个 balls 数组。可以根据小球的 z 轴深度从高到低来排序这个数组，然后从 balls 的 0（最远的） 到 49（最近的）为每个小球设置索引。请看下面这段代码：

```
private function sortZ():void {
 balls.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
 for (var i:uint = 0; i < numBalls; i++) {
  var ball:Ball3D = balls[i];
  setChildIndex(ball, i);
 }
```

```
}
```

根据数组中每个对象的 zpos 属性对该数组进行排序。因为指定了数组的 DESCENDING 和 Array.NUMERIC，则是按数值大小反向排序的——换句话讲，就是从高到低。结果会使最远的小球（zpos 值最高的）将成为数组中的第一个，最近的将成为最后一个。

然后循环这个数组，将每个小球在显示列表中的索引值设置为与当前在数组中的索引值相同。

将这个方法放入类中，只需要在小球移动后调用它即可，将函数调用放在 onEnterFrame 方法的最后：

```
private function onEnterFrame(event:Event):void {
 for (var i:uint = 0; i < numBalls; i++) {
  var ball:Ball3D = balls[i];
  move(ball);
 }
 sortZ();
}
```

剩下的代码与上一个例子中的相同。全部代码可在 Zsort.as 中找到。

重力

这里我们所说的重力就像地球表面上的重力一样，如第五章所讲的。既然这样，3D 的重力和 2D 的就很像了。我们所需要做的就是选择一个施加在物体上的重力值，并在每帧中将它加入到物体的速度中。

由于 3D 的重力非常简单，我差点就跳过去说“是的，同 2D 一样。OK，下一话题。”但是，我决定将它放到一个很好的例子中加以解释，让大家知道即使很简单的东西也可以创造出非常棒的效果，就像 3D 烟火一样。

首先，我们需要找个物体代表一个单独的“烟火”——我们知道，这些发光的点可以组合到一起形成巨大的爆炸。我们给忠实的 Ball3D 类一个较小的半径值，就可以完成这个目的。只要给每个小球一个随机的颜色效果就会非常漂亮。如果将背景色设置为黑色就更好了。我使用 SWF 元数据来完成这个设置，但如果是在 Flash CS3 IDE 中，只需要简单地改变一下文档属性的背景色即可。

我确信大家现在一定能够完成。先将所有的代码列出来（Fireworks.as），随后加以解释。

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 [SWF(backgroundColor=0x000000)];
 public class Fireworks extends Sprite {
  private var balls:Array;
  private var numBalls:uint = 100;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
  private var vpY:Number = stage.stageHeight / 2;
  private var gravity:Number = 0.2;
  private var floor:Number = 200;
  private var bounce:Number = -0.6;
  public function Fireworks() {
   init();
  }
```

```
    private function init():void {
      balls = new Array();
      for (var i:uint = 0; i < numBalls; i++) {
        var ball:Ball3D = new Ball3D(3, Math.random() * 0xffffff);
        balls.push(ball);
        ball.ypos = -100;
        ball.vx = Math.random() * 6 - 3;
        ball.vy = Math.random() * 6 - 6;
        ball.vz = Math.random() * 6 - 3;
        addChild(ball);
      }
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onEnterFrame(event:Event):void {
      for (var i:uint = 0; i < numBalls; i++) {
        var ball:Ball3D = balls[i];
        move(ball);
      }
      sortZ();
    }
    private function move(ball:Ball3D):void {
      ball.vy += gravity;
      ball.xpos += ball.vx;
      ball.ypos += ball.vy;
      ball.zpos += ball.vz;
      if (ball.ypos > floor) {
        ball.ypos = floor;
        ball.vy *= bounce;
      }
      if (ball.zpos > -fl) {
        var scale:Number = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
      } else {
        ball.visible = false;
      }
    }
    private function sortZ():void {
      balls.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
      for (var i:uint = 0; i < numBalls; i++) {
        var ball:Ball3D = balls[i];
        setChildIndex(ball, i);
      }
    }
  }
}
```

首先加入一些属性：gravity，bounce，floor。前两个大家都见过。Floor 属性就是——

bottom ——也就是物体反弹之前可以运动到的最大 y 值。除了增加 y 轴速度以及碰撞地面后的反弹以外，所有的内容我们前面都介绍过，是不是越来越酷了，哈？

运行结果如图 15-6 所示。

图 15-6 烟火（相信我，运动中的效果更好）

屏幕环绕

回忆一下第六章，我们说过三种当物体碰到边界后受到反作用力的可能。目前为止，我们只介绍了反弹。还有两个：屏幕环绕与重置。对于 3D 而言，我发现屏幕环绕效果是最为有用的，但只能在 z 轴上使用。

2D 屏幕环绕中，在 x 或 y 轴上判断物体是否出了屏幕。效果非常好，因为当物体超出了其中一个边界时就看不到它了，因此可以轻松地重新设置物体的位置，不会引起人们的注意。但是 3D 中就不能这么潇洒了。

在 3D 中，实际上只有两个点可以安全地删除和重置物体。一个就是当物体运动到观察点的后面。前面例子中，将物体设置为不可见时，就是这个道理。另一个就是当物体的距离太远和太小时也可以将其设为不可见的。这就意味着我们可以在 z 轴上安全地进行屏幕包装。当物体走到身后时，就将它放到面前的远方。如果物体离得过远，超出了可见范围，就可以删除它并将其重置到身后。如果大家喜欢 x, y 轴也可以这样做，但是多数情况下，这么做会导致一些不自然的忽隐忽现的效果。

还好 z 轴的环绕可以是相当常用的。我曾用它制作出真实的 3D 赛车游戏，下面我们就来制作其中的一部分。

主体思想是把不同的 3D 物体放到观察点前。然后将这些物体向观察点移动。换句话讲，给它们一些负的 z 速度。这要看我们如何设置了，可以让物体向我们走来，让眼睛以为是我们向物体走去。一旦物体走到了观察点后，就将它重置到眼前一段距离。这样就可以永无止境地掠过这些物体了。

本例中使用的物体是一棵线条化的树。创建一棵带有随机枝叉的树形结构。我确信您还能做得更好！

绘制树的代码放在名为 Tree 的类中，下面会看到，用三个位置属性以及随机绘制树枝的代码来代表一颗树。

```
package {
 import flash.display.Sprite;
 public class Tree extends Sprite {
  public var xpos:Number = 0;
```

```
    public var ypos:Number = 0;
    public var zpos:Number = 0;
    public function Tree() {
      init();
    }
    public function init():void {
      graphics.lineStyle(0, 0xffffff);
      graphics.lineTo(0, -140 - Math.random() * 20);
      graphics.moveTo(0, -30 - Math.random() * 30);
      graphics.lineTo(Math.random() * 80 - 40,
      -100 - Math.random() * 40);
      graphics.moveTo(0, -60 - Math.random() * 40);
      graphics.lineTo(Math.random() * 60 - 30,
      -110 - Math.random() * 20);
    }
  }
}
```

同样，还是使用 SWF 元数据将背景色设置为黑色。大家可以创建任何喜欢的物体，想要多复杂都可以自行设置。在文档类中创建所有的树（100 左右）。随机分散在 x 轴上，每个方向 1000 像素。它们同样随机分散到 z 轴上，从 0 到 10000。它们都以 floor 属性为基础，具有相同的 y 坐标，给人一种地平面的感觉。

以下是代码（可以见 Trees.as）：

Here’s the code (which you can also find in Trees.as):

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.KeyboardEvent;
  import flash.ui.Keyboard;
  [SWF(backgroundColor=0x000000)];
  public class Trees extends Sprite {
    private var trees:Array;
    private var numTrees:uint = 100;
    private var fl:Number = 250;
    private var vpX:Number = stage.stageWidth / 2;
    private var vpY:Number = stage.stageHeight / 2;
    private var floor:Number = 50;
    private var vz:Number = 0;
    private var friction:Number = 0.98;
    public function Trees() {
      init();
    }
    private function init():void {
      trees = new Array();
      for (var i:uint = 0; i < numTrees; i++) {
        var tree:Tree = new Tree();
        trees.push(tree);
        tree.xpos = Math.random() * 2000 - 1000;
        tree.ypos = floor;
        tree.zpos = Math.random() * 10000;
```

```
        addChild(tree);
      }
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
      stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
    }
    private function onEnterFrame(event:Event):void {
      for (var i:uint = 0; i < numTrees; i++) {
        var tree:Tree = trees[i];
        move(tree);
      }
      vz *= friction;
      sortZ();
    }
    private function onKeyDown(event:KeyboardEvent):void {
      if (event.keyCode == Keyboard.UP) {
        vz -= 1;
      } else if (event.keyCode == Keyboard.DOWN) {
        vz += 1;
      }
    }
    private function move(tree:Tree):void {
      tree.zpos += vz;
      if (tree.zpos < -fl) {
        tree.zpos += 10000;
      }
      if (tree.zpos > 10000 - fl) {
        tree.zpos -= 10000;
      }
      var scale:Number = fl / (fl + tree.zpos);
      tree.scaleX = tree.scaleY = scale;
      tree.x = vpX + tree.xpos * scale;
      tree.y = vpY + tree.ypos * scale;
      tree.alpha = scale;
    }
    private function sortZ():void {
      trees.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
      for (var i:uint = 0; i < numTrees; i++) {
        var tree:Tree = trees[i];
        setChildIndex(tree, i);
      }
    }
  }
}
```

请注意，这里只有一个 z 轴速度变量，因为树不需要在 x 或 y 轴上进行移动，所有的移动都在 z 轴上。在 onEnterFrame 方法中，判断方向键上和下，增加或减少 vz。加入一点点摩擦力让速度不会增加到无限大，在按键松开时将速度降下来。

代码循环获得每棵树，用当前 z 速度更新该树的 z 坐标。然后判断这棵树是否走到了我们的身后。如果是，将这个棵树向 z 轴内移动 10000 像素。否则，如果超过了 10000 - fl，就将该树往回移动 10000 像素。再执行标准透视动作。为了更好地加强立体感我还加

入了一个小小的设计：

```
tree.alpha = scale;
```

根据 z 轴的深度设置树的透明度。离得越远颜色越淡。这是大气透视，模拟大气与观察者和物体之间的效果。这是本例中表现物体远离时一种特殊效果。这个特别的设计给了我们黑暗的效果和幽深的夜。大家也许可以试试这种方法：

```
tree.alpha = scale * .7 + .3;
```

让树的可见度至少为 30%。看上去不再那么朦胧。这里没有正确或错误可言——只有不同的数值创造不同的效果。

大家也许注意到了，我仍把 z 排序方法留在这里。在这个特殊的例子中，它没有发挥本应有的作用，因为树都是由同一颜色的简单线条构成的，但如果绘制的是一些非常复杂的或重叠的图形，那么它的存在就是至关重要的了。

本文件的运行结果如图 15-7 所示。

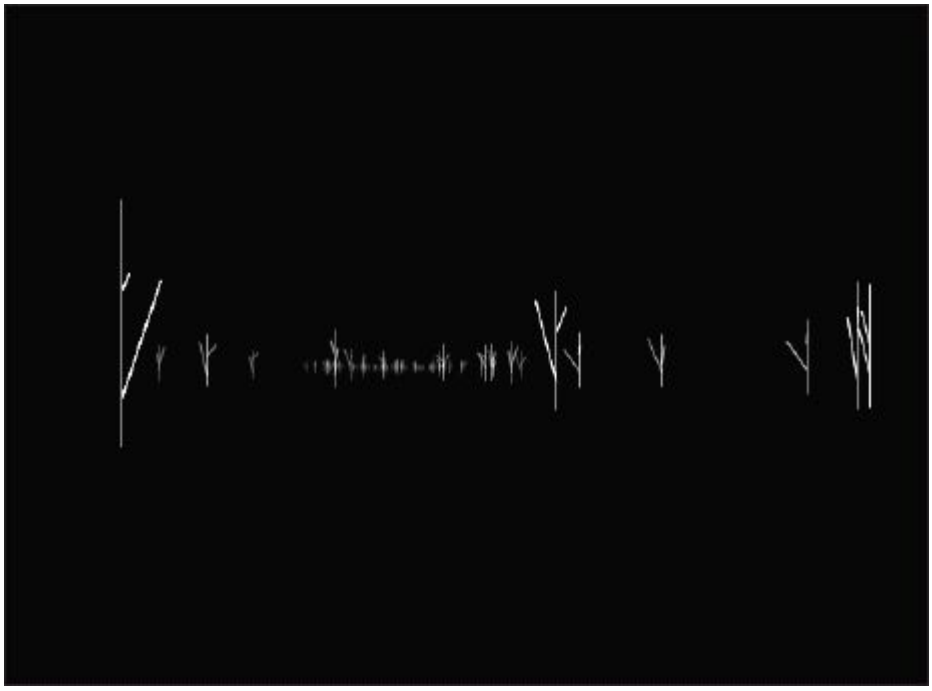


图 15-7 当心小树！

下面我将给大家一个加强的例子，让我们看一下还可以做到什么样的程度。以下是程序（可以在 Trees2.as 中找到）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.events.KeyboardEvent;
 import flash.ui.Keyboard;
 [SWF(backgroundColor=0x000000)];
 public class Trees2 extends Sprite {
  private var trees:Array;
  private var numTrees:uint = 100;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
  private var vpY:Number = stage.stageHeight / 2;
  private var floor:Number = 50;
  private var ax:Number = 0;
  private var ay:Number = 0;
  private var az:Number = 0;
  private var vx:Number = 0;
```

```
private var vy:Number = 0;
private var vz:Number = 0;
private var gravity:Number = 0.3;
private var friction:Number = 0.98;
public function Trees2() {
 init();
}
private function init():void {
 trees = new Array();
 for (var i:uint = 0; i < numTrees; i++) {
  var tree:Tree = new Tree();
  trees.push(tree);
  tree.xpos = Math.random() * 2000 - 1000;
  tree.ypos = floor;
  tree.zpos = Math.random() * 10000;
  addChild(tree);
 }
 addEventListener(Event.ENTER_FRAME, onEnterFrame);
 stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
 stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
}
private function onEnterFrame(event:Event):void {
 vx += ax;
 vy += ay;
 vz += az;
 vy -= gravity;
 for (var i:uint = 0; i < numTrees; i++) {
  var tree:Tree = trees[i];
  move(tree);
 }
 vx *= friction;
 vy *= friction;
 vz *= friction;
 sortZ();
}
private function onKeyDown(event:KeyboardEvent):void {
 switch (event.keyCode) {
  case Keyboard.UP :
   az = -1;
   break;
  case Keyboard.DOWN :
   az = 1;
   break;
  case Keyboard.LEFT :
   ax = 1;
   break;
  case Keyboard.RIGHT :
   ax = -1;
   break;
```

```
    case Keyboard.SPACE :
      ay = 1;
      break;
    default :
      break;
  }
}
private function onKeyUp(event:KeyboardEvent):void {
  switch (event.keyCode) {
    case Keyboard.UP :
    case Keyboard.DOWN :
      az = 0;
      break;
    case Keyboard.LEFT :
    case Keyboard.RIGHT :
      ax = 0;
      break;
    case Keyboard.SPACE :
      ay = 0;
      break;
    default :
      break;
  }
}
private function move(tree:Tree):void {
  tree.xpos += vx;
  tree.ypos += vy;
  tree.zpos += vz;
  if (tree.ypos < floor) {
    tree.ypos = floor;
  }
  if (tree.zpos < -fl) {
    tree.zpos += 10000;
  }
  if (tree.zpos > 10000 - fl) {
    tree.zpos -= 10000;
  }
  var scale:Number = fl / (fl + tree.zpos);
  tree.scaleX = tree.scaleY = scale;
  tree.x = vpX + tree.xpos * scale;
  tree.y = vpY + tree.ypos * scale;
  tree.alpha = scale;
}
private function sortZ():void {
  trees.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
  for (var i:uint = 0; i < numTrees; i++) {
    var tree:Tree = trees[i];
    setChildIndex(tree, i);
  }
```

```
  }
 }
}
```

这里，我已经加入了 x 和 y 轴的速度，还有重力。还必需要能够捕获多个按键。我唯一想念 AS 2 的是 Key.isDown() 方法，任何时间都可以调用找出某个键是否被按住。因为在 AS 3 中我们只能知道最后一次按下或释放的键，所以不得不判断哪个键被按下并设置相应轴上的加速度为 1 或 -1。随后，当该键被松开时，再将加速度设回 0。在 onEnterFrame 的开始就将每个轴上的加速度加到相应轴的速度中。左键和右键显然就是用于选择 x 轴的速度，使用空格键操作 y 轴。有趣的一点是我们实际是从 vy 减去了重力。因为我想要一个类似于观察者落到树林中的效果，如图 15-8 所示。注意我们同样也限定了树的 y 坐标为 50，看起来就像是站在陆地上一样。

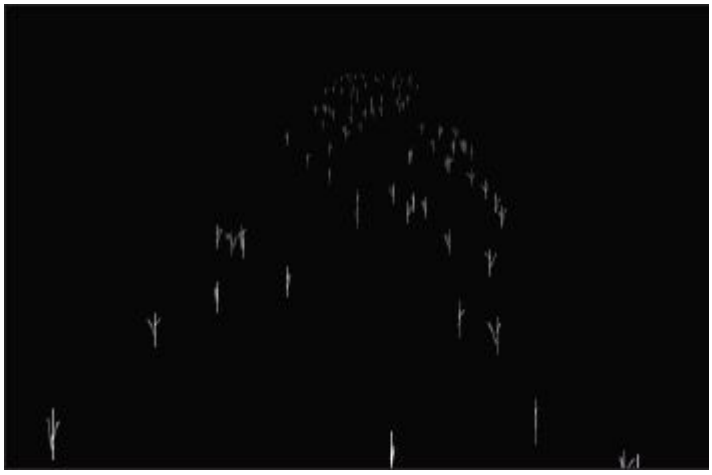


图 15-8 看，我在飞！

这里没有对 x 轴的运动加以任何的限制，也就意味着可以在树林边上行进。要想加入限制对于大家来说也不是件难事，但是作为一个启发性的例子做到这里已经足够了。

缓动与弹性运动

在 3D 中的缓动与弹性运动不会比 2D 中的难多少（第八章的课题）。我们只需要为 z 轴再加入一至两个变量。

缓动

对于缓动的介绍不算很多。在 2D 中，我们用 tx 和 ty 最为目标点。现在只需要再在 z 轴上加入 tz。每帧计算物体每个轴到目标点的距离，并移动一段距离。

让我们来看一个简单的例子，让物体缓动运动到随机的目标点，到达该点后，再选出另一个目标并让物体移动过去。注意后面两个例子，我们又回到了 Ball3D 这个类上。以下是代码（可以在 Easing3D.as 中找到）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Easing3D extends Sprite {
  private var ball:Ball3D;
  private var tx:Number;
  private var ty:Number;
  private var tz:Number;
  private var easing:Number = .1;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
```

```
    private var vpY:Number = stage.stageHeight / 2;
    public function Easing3D() {
      init();
    }
    private function init():void {
      ball = new Ball3D();
      addChild(ball);
      tx = Math.random() * 500 - 250;
      ty = Math.random() * 500 - 250;
      tz = Math.random() * 500;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onEnterFrame(event:Event):void {
      var dx:Number = tx - ball.xpos;
      var dy:Number = ty - ball.ypos;
      var dz:Number = tz - ball.zpos;
      ball.xpos += dx * easing;
      ball.ypos += dy * easing;
      ball.zpos += dz * easing;
      var dist:Number = Math.sqrt(dx*dx + dy*dy + dz*dz);
      if (dist < 1) {
        tx = Math.random() * 500 - 250;
        ty = Math.random() * 500 - 250;
        tz = Math.random() * 500;
      }
      if (ball.zpos > -fl) {
        var scale:Number = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
      } else {
        ball.visible = false;
      }
    }
  }
}
```

代码中最有趣的地方是下面这行：

```
var dist:Number = Math.sqrt(dx * dx + dy * dy + dz * dz);
```

我们知道，在 2D 中计算两点间距离的方程是：

```
var dist:Number = Math.sqrt(dx * dx + dy * dy);
```

在 3D 距离中，只需要将第三个轴距离的平方加入进去。由于这个公式过于简单所以我常常会受到质疑。在加入了一个条件后，似乎应该使用立方根。但是它并不是用在这里的。

弹性运动

弹性运动是缓动的兄弟，需用相似的方法将其调整为 3D 的。我们只使用物体到目标的距离改变速度，而不是改变位置。给大家一个快速的示例。本例中（Spring3D.as），点击

鼠标将创建出一个随机的目标点。

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.MouseEvent;
  public class Spring3D extends Sprite {
    private var ball:Ball3D;
    private var tx:Number;
    private var ty:Number;
    private var tz:Number;
    private var spring:Number = .1;
    private var friction:Number = .94;
    private var fl:Number = 250;
    private var vpX:Number = stage.stageWidth / 2;
    private var vpY:Number = stage.stageHeight / 2;
    public function Spring3D() {
      init();
    }
    private function init():void {
      ball = new Ball3D();
      addChild(ball);
      tx = Math.random() * 500 - 250;
      ty = Math.random() * 500 - 250;
      tz = Math.random() * 500;
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
      stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    }
    private function onEnterFrame(event:Event):void {
      var dx:Number = tx - ball.xpos;
      var dy:Number = ty - ball.ypos;
      var dz:Number = tz - ball.zpos;
      ball.vx += dx * spring;
      ball.vy += dy * spring;
      ball.vz += dz * spring;
      ball.xpos += ball.vx;
      ball.ypos += ball.vy;
      ball.zpos += ball.vz;
      ball.vx *= friction;
      ball.vy *= friction;
      ball.vz *= friction;
      if (ball.zpos > -fl) {
        var scale:Number = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
      } else {
        ball.visible = false;
      }
```

```
    }
    private function onMouseDown(event:MouseEvent):void {
      tx = Math.random() * 500 - 250;
      ty = Math.random() * 500 - 250;
      tz = Math.random() * 500;
    }
  }
}
```

我们看到，在第三个轴上使用了基本的弹性运动公式（来自第八章）。

坐标旋转

接下来是 3D 坐标旋转。这里要比第十和十一章中的 2D 坐标旋转要稍微复杂一些。我们不仅可以在三个不同的轴上旋转，还可以同时在两个以上的轴上进行旋转。

在 2D 坐标旋转中物体是绕着 z 轴旋转的，如图 15-9 所示。想一想命运之轮(Wheel of Fortune)这类游戏——轮轴从轮盘的中心穿过。轮轴就是 z 轴。只改变 x 和 y 坐标。

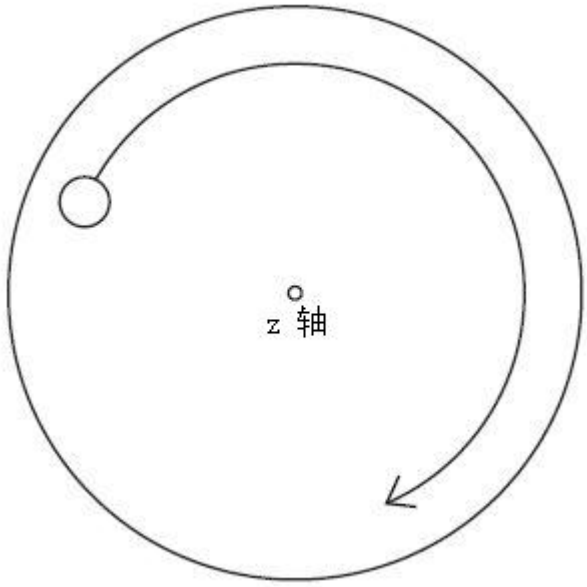


图 15-9 z 轴旋转

在 3D 中，我们也可以在 y 或 z 轴上旋转。x 轴的旋转如同车轮向前滚动，如图 15-10 所示。这时轮轴就是 x 轴。只改变 y 和 z 上的点。

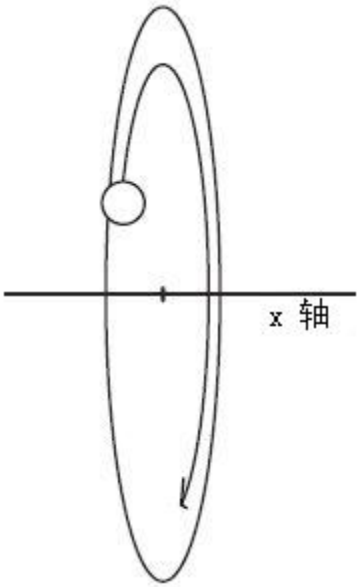


图 15-10 x 轴旋转

y 轴上的旋转，请想象一下老的唱片机，如图 15-11 所示。轴心是 y 轴。只改变 x 和 z 上的点。

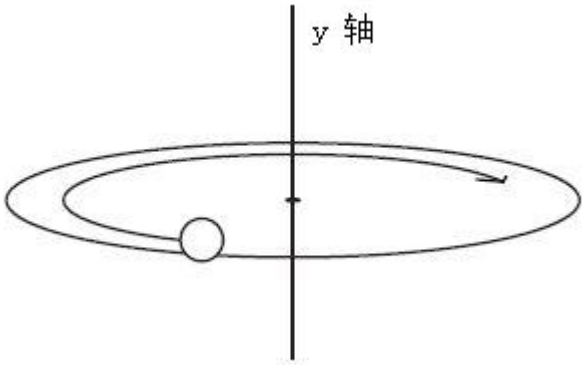


图 15-11 y 轴旋转

因此，对于 3D 而言，当我们在某个轴上旋转一个物体时，将改变两个轴上的位置。回顾一下第十章，我们找到 2D 旋转的公式如下：

```
x1 = cos(angle) * x - sin(angle) * y;
y1 = cos(angle) * y + sin(angle) * x;
```

在 3D 中，方法几乎相同，但是需要指定操作的是哪个轴：x, y, z。因此得到下面三个公式：

```
x1 = cos(angleZ) * x - sin(angleZ) * y;
y1 = cos(angleZ) * y + sin(angleZ) * x;
x1 = cos(angleY) * x - sin(angleY) * z;
z1 = cos(angleY) * z + sin(angleY) * x;
y1 = cos(angleX) * y - sin(angleX) * z;
z1 = cos(angleX) * z + sin(angleX) * y;
```

下面，试一下 y 轴的旋转。以下代码可在 RotateY.as 中找到。创建 50 个 Ball3D 的实例，随机放置舞台上。根据鼠标 x 轴上的位置计算出 y 轴角度。鼠标越向右，角度值越高。物体就像跟着鼠标旋转一样。

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  public class RotateY extends Sprite {
    private var balls:Array;
    private var numBalls:uint = 50;
```

```
private var fl:Number = 250;
private var vpX:Number = stage.stageWidth / 2;
private var vpY:Number = stage.stageHeight / 2;
public function RotateY() {
  init();
}
private function init():void {
  balls = new Array();
  for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball3D = new Ball3D(15);
    balls.push(ball);
    ball.xpos = Math.random() * 200 - 100;
    ball.ypos = Math.random() * 200 - 100;
    ball.zpos = Math.random() * 200 - 100;
    addChild(ball);
  }
  addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
private function onEnterFrame(event:Event):void {
  var angleY:Number = (mouseX - vpX) * .001;
  for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball3D = balls[i];
    rotateY(ball, angleY);
  }
  sortZ();
}
private function rotateY(ball:Ball3D, angleY:Number):void {
  var cosY:Number = Math.cos(angleY);
  var sinY:Number = Math.sin(angleY);
  var x1:Number = ball.xpos * cosY - ball.zpos * sinY;
  var z1:Number = ball.zpos * cosY + ball.xpos * sinY;
  ball.xpos = x1;
  ball.zpos = z1;
  if (ball.zpos > -fl) {
    var scale:Number = fl / (fl + ball.zpos);
    ball.scaleX = ball.scaleY = scale;
    ball.x = vpX + ball.xpos * scale;
    ball.y = vpY + ball.ypos * scale;
    ball.visible = true;
  } else {
    ball.visible = false;
  }
}
private function sortZ():void {
  balls.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
  for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball3D = balls[i];
    setChildIndex(ball, i);
  }
```

```
    }
  }
}
```

重要的部分加粗表示。获得一个角度，然后使用该角度调用 rotateY 的方法。该方法中，获得角度的正弦和余弦值，执行旋转，将 x1 和 z1 再赋值给 ball.xpos 和 ball.zpos。接下来，就是标准透视以及 z 排序。运行结果如图 15-12 所示。

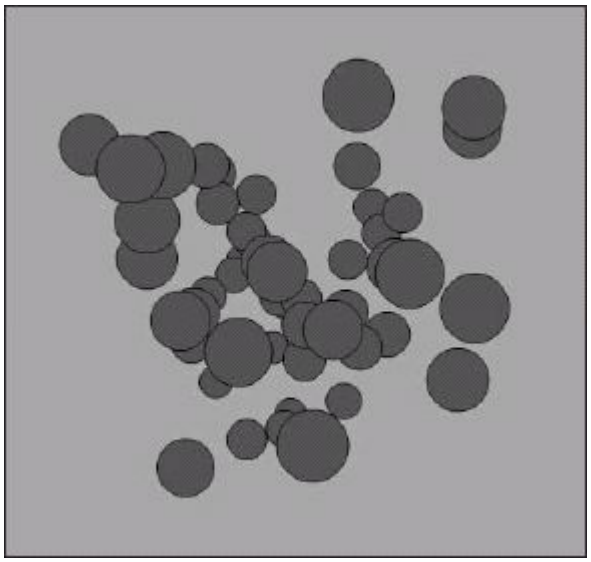


图 15-12　y 轴旋转

如果这个程序没问题，大家一定也可以将其转换为 x 轴的旋转。只需要改变 onEnterFrame 方法并加入 rotateX 方法：

```
private function onEnterFrame(event:Event):void {
  var angleX:Number = (mouseY - vpY) * .001;
  for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball3D = balls[i];
    rotateX(ball, angleX);
  }
  sortZ();
}
private function rotateX(ball:Ball3D, angleX:Number):void {
  var cosX:Number = Math.cos(angleX);
  var sinX:Number = Math.sin(angleX);
  var y1:Number = ball.ypos * cosX - ball.zpos * sinX;
  var z1:Number = ball.zpos * cosX + ball.ypos * sinX;
  ball.ypos = y1;
  ball.zpos = z1;
  if (ball.zpos > -fl) {
    var scale:Number = fl / (fl + ball.zpos);
    ball.scaleX = ball.scaleY = scale;
    ball.x = vpX + ball.xpos * scale;
    ball.y = vpY + ball.ypos * scale;
    ball.visible = true;
  } else {
    ball.visible = false;
  }
}
```

现在，angleX 是根据鼠标的 y 坐标确定的。然后计算出正余弦的值，并使用它们算出

y1 和 z1，再赋值给 ypos 和 zpos 属性。

接下来，我们将两种旋转组合起来。以下是代码（可以在 RotateXY.as 中找到）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class RotateXY extends Sprite {
  private var balls:Array;
  private var numBalls:uint = 50;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
  private var vpY:Number = stage.stageHeight / 2;
  public function RotateXY() {
   init();
  }
  private function init():void {
   balls = new Array();
   for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball3D = new Ball3D(15);
    balls.push(ball);
    ball.xpos = Math.random() * 200 - 100;
    ball.ypos = Math.random() * 200 - 100;
    ball.zpos = Math.random() * 200 - 100;
    addChild(ball);
   }
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var angleX:Number = (mouseY - vpY) * .001;
   var angleY:Number = (mouseX - vpX) * .001;
   for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball3D = balls[i];
    rotateX(ball, angleX);
    rotateY(ball, angleY);
    doPerspective(ball);
   }
   sortZ();
  }
  private function rotateX(ball:Ball3D, angleX:Number):void {
   var cosX:Number = Math.cos(angleX);
   var sinX:Number = Math.sin(angleX);
   var y1:Number = ball.ypos * cosX - ball.zpos * sinX;
   var z1:Number = ball.zpos * cosX + ball.ypos * sinX;
   ball.ypos = y1;
   ball.zpos = z1;
  }
  private function rotateY(ball:Ball3D, angleY:Number):void {
   var cosY:Number = Math.cos(angleY);
   var sinY:Number = Math.sin(angleY);
   var x1:Number = ball.xpos * cosY - ball.zpos * sinY;
```

```
      var z1:Number = ball.zpos * cosY + ball.xpos * sinY;
      ball.xpos = x1;
      ball.zpos = z1;
    }
    private function doPerspective(ball:Ball3D):void {
      if (ball.zpos > -fl) {
        var scale:Number = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
      } else {
        ball.visible = false;
      }
    }
    private function sortZ():void {
      balls.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
      for (var i:uint = 0; i < numBalls; i++) {
        var ball:Ball3D = balls[i];
        setChildIndex(ball, i);
      }
    }
  }
}
```

与前一个例子相比主要的变化用粗体表示。现在，我们计算出 angleY, angleX ，并调用 rotateX, rotateY。注意，我将透视的代码从 rotate 方法中分离到一个单独的方法中，因为它不需要被调用两次。我相信根据前面的公式，您一定可以自行加入 rotateZ 方法。

测试一下这个影片。通过将 3D 坐标旋转与前一节赛车游戏的“屏幕环绕”的概念相结合，大家一定还可以创建出丰富的，交互性的 3D 环境。

碰撞检测

本章的最后我要给大家介绍一下 3D 的碰撞检测。在 Flash 的三维空间中唯一可行的方法就是距离碰撞检测。找出两个物体的距离（使用 3D 距离公式），如果小于两个物体的半径之和，就产生了碰撞。

我将前面 3D 反弹的例子改成了 3D 碰撞检测的例子，加入了少量的物体并扩大了一些空间。代码中首先执行普通的 3D 运动及透视，然后做一个双重循环比较所有小球的位置。如果有两个物体的距离小于它们的半径之和，就使用颜色转换代码将它们都设置为蓝色。非常简单。以下是代码（Collision3D.as）：

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.geom.ColorTransform;
  public class Collision3D extends Sprite {
    private var balls:Array;
    private var numBalls:uint = 20;
    private var fl:Number = 250;
    private var vpX:Number = stage.stageWidth / 2;
```

```
private var vpY:Number = stage.stageHeight / 2;
private var top:Number = -200;
private var bottom:Number = 200;
private var left:Number = -200;
private var right:Number = 200;
private var front:Number = 200;
private var back:Number = -200;
public function Collision3D() {
 init();
}
private function init():void {
 balls = new Array();
 for (var i:uint = 0; i < numBalls; i++) {
  var ball:Ball3D = new Ball3D(15);
  balls.push(ball);
  ball.xpos = Math.random() * 400 - 200;
  ball.ypos = Math.random() * 400 - 200;
  ball.zpos = Math.random() * 400 - 200;
  ball.vx = Math.random() * 10 - 5;
  ball.vy = Math.random() * 10 - 5;
  ball.vz = Math.random() * 10 - 5;
  addChild(ball);
 }
 addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
private function onEnterFrame(event:Event):void {
 for (var i:uint = 0; i < numBalls; i++) {
  var ball:Ball3D = balls[i];
  move(ball);
 }
 for (i = 0; i < numBalls - 1; i++) {
  var ballA:Ball3D = balls[i];
  for (var j:uint = i + 1; j < numBalls; j++) {
   var ballB:Ball3D = balls[j];
   var dx:Number = ballA.xpos - ballB.xpos;
   var dy:Number = ballA.ypos - ballB.ypos;
   var dz:Number = ballA.zpos - ballB.zpos;
   var dist:Number = Math.sqrt(dx*dx + dy*dy + dz*dz);
   if (dist < ballA.radius + ballB.radius) {
    var blueTransform:ColorTransform =
    new ColorTransform(0, 1, 1, 1, 0, 0, 255, 0);
    ballA.transform.colorTransform = blueTransform;
    ballB.transform.colorTransform = blueTransform;
   }
  }
 }
 sortZ();
}
private function move(ball:Ball3D):void {
```

```
      var radius:Number = ball.radius;
      ball.xpos += ball.vx;
      ball.ypos += ball.vy;
      ball.zpos += ball.vz;
      if (ball.xpos + radius > right) {
        ball.xpos = right - radius;
        ball.vx *= -1;
      } else if (ball.xpos - radius < left) {
        ball.xpos = left + radius;
        ball.vx *= -1;
      }
      if (ball.ypos + radius > bottom) {
        ball.ypos = bottom - radius;
        ball.vy *= -1;
      } else if (ball.ypos - radius < top) {
        ball.ypos = top + radius;
        ball.vy *= -1;
      }
      if (ball.zpos + radius > front) {
        ball.zpos = front - radius;
        ball.vz *= -1;
      } else if (ball.zpos - radius < back) {
        ball.zpos = back + radius;
        ball.vz *= -1;
      }
      if (ball.zpos > -fl) {
        var scale:Number = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
      } else {
        ball.visible = false;
      }
    }
    private function sortZ():void {
      balls.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
      for (var i:uint = 0; i < numBalls; i++) {
        var ball:Ball3D = balls[i];
        setChildIndex(ball, i);
      }
    }
  }
}
```

重要的部分用粗体表示。小球起初都是红色的，当它们碰撞后，颜色发生改变。最后，全部变为蓝色。

本章重要公式

本章的重要公式是 3D 透视，坐标旋转以及距离的计算。

基本透视法：

```
scale = fl / (fl + zpos);
sprite.scaleX = sprite.scaleY = scale;
sprite.alpha = scale; // 可选
sprite.x = vanishingPointX + xpos * scale;
sprite.y = vanishingPointY + ypos * scale;
```

Z 排序：

```
// 假设有一个带有 zpos 属性的 3D 物体的数组
objectArray.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
for(var i:uint = 0; i < numObjects; i++) {
setChildIndex(objectArray[i], i);
}
```

坐标旋转：

```
x1 = cos(angleZ) * xpos - sin(angleZ) * ypos;
y1 = cos(angleZ) * ypos + sin(angleZ) * xpos;
x1 = cos(angleY) * xpos - sin(angleY) * zpos;
z1 = cos(angleY) * zpos + sin(angleY) * xpos;
y1 = cos(angleX) * ypos - sin(angleX) * zpos;
z1 = cos(angleX) * zpos + sin(angleX) * ypos;
```

3D 距离：

```
dist = Math.sqrt(dx * dx + dy * dy + dz * dz);
```

15.10 小结

你现在已经有了 3D 的基础，并且已经看到多数的基本运动代码是适用于 3D 的。我不得不说出连我自己都感到非常惊讶的话，“这与 2D 是一样的，你只需加入一个 z 变量……”或者类似的话。我愿意为这里要解释更加复杂的内容，但是结果大部分内容都相当简单。

在下一章你将用到在这里学到的许多内容，在那里你将真正开始使用点和线条雕刻你的 3D 模型。

第十六章　3D 线条与填充

第十五章我们介绍了 3D，但只是将物体置于 3D 空间中，设置大小与位置。物体实际上还是 2D 的。这就像老的 3D 游戏中，我们可以绕着某个物体或人物走，这些对象会转过来面对我们。这些物体或人物并不是真正的会转过来 —— 只是看上去是这样的，因为它们都是 2D 物体，那是我们看到它唯一的一个视角。

本章，我们将真正地在 Flash 中创建 3D 模型。具体说来有创建并使用 3D 点，线条，填充以及立体图形。学习完本章，大家就可以任意在三维空间中创建各种形状，并对它们执行移动和旋转。

创建点和线条

如果只在 3D 中创建点而不创建线条的话是没有任何意义的。因为理论上讲，一个点是没有维度并且不可见的。开始时，我们还将使用 Ball3D 类，给一个非常小的半径，只要能看到点在哪就够了。这样，只需要再创建几条线将这些小球连起来就行了。相当简单。我们前面做过这样的例子，不过这次要在小球上添加透视，将它们放入 3D 空间。

可以将点的实例设为黑色，10 像素宽。创建一些点的实例，根据鼠标的位置对它们进行旋转（如前一章最后所讲的一样），然后在点间绘制一些线条。代码与上一章的 RotateXY.as 文件几乎相同。主要的区别在 onEnterFrame 的最后一块儿，从第一个点循环绘制线条到剩下所有的点。同时，我还删除了 sortZ 方法，因为在这里有没有都一样。以下是 Lines3D1.as 的全部代码。运行结果如图 16-1 所示。

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Lines3D1 extends Sprite {
  private var balls:Array;
  private var numBalls:uint = 50;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
  private var vpY:Number = stage.stageHeight / 2;
  public function Lines3D1() {
   init();
  }
  private function init():void {
   balls = new Array();
   for (var i:uint = 0; i < numBalls; i++) {
    var ball:Ball3D = new Ball3D(5, 0);
    balls.push(ball);
    ball.xpos = Math.random() * 200 - 100;
    ball.ypos = Math.random() * 200 - 100;
    ball.zpos = Math.random() * 200 - 100;
    addChild(ball);
   }
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var angleX:Number = (mouseY - vpY) * .001;
   var angleY:Number = (mouseX - vpX) * .001;
```

```
      for (var i:uint = 0; i < numBalls; i++) {
        var ball:Ball3D = balls[i];
        rotateX(ball, angleX);
        rotateY(ball, angleY);
        doPerspective(ball);
      }
      graphics.clear();
      graphics.lineStyle(0);
      graphics.moveTo(balls[0].x, balls[0].y);
      for (i = 1; i < numBalls; i++) {
        graphics.lineTo(balls[i].x, balls[i].y);
      }
    }
    private function rotateX(ball:Ball3D, angleX:Number):void {
      var cosX:Number = Math.cos(angleX);
      var sinX:Number = Math.sin(angleX);
      var y1:Number = ball.ypos * cosX - ball.zpos * sinX;
      var z1:Number = ball.zpos * cosX + ball.ypos * sinX;
      ball.ypos = y1;
      ball.zpos = z1;
    }
    private function rotateY(ball:Ball3D, angleY:Number):void {
      var cosY:Number = Math.cos(angleY);
      var sinY:Number = Math.sin(angleY);
      var x1:Number = ball.xpos * cosY - ball.zpos * sinY;
      var z1:Number = ball.zpos * cosY + ball.xpos * sinY;
      ball.xpos = x1;
      ball.zpos = z1;
    }
    private function doPerspective(ball:Ball3D):void {
      if (ball.zpos > -fl) {
        var scale:Number = fl / (fl + ball.zpos);
        ball.scaleX = ball.scaleY = scale;
        ball.x = vpX + ball.xpos * scale;
        ball.y = vpY + ball.ypos * scale;
        ball.visible = true;
      } else {
        ball.visible = false;
      }
    }
  }
}
```

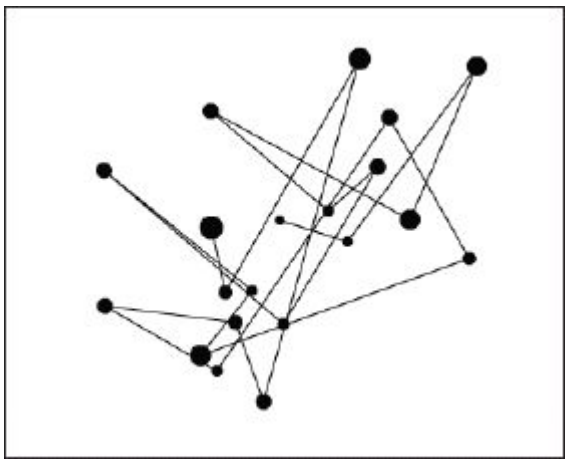


图 16-1　3D 点与线

实际来说，没有太多可改的地方。如果要转为立体 3D 模型，就要舍弃所有的这些黑点。第一个尝试非常简单。只要在创建时将每个 Ball3D 实例的半径设置为 0 即可，如下：

```
var ball:Ball3D = new Ball3D(0);
```

运行结果如图 16-2 所示。

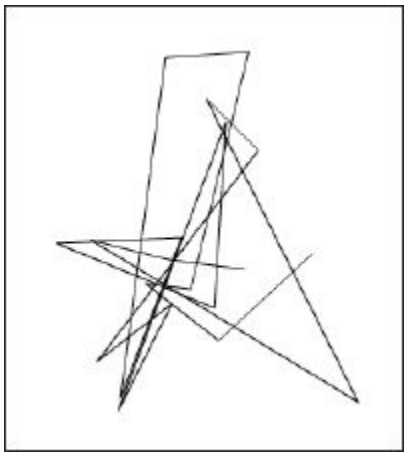


图 16-2　3D 线条，点不可见

回过头再看看先前的代码，这时我们发现有很多地方是多余的。比如，scaleX, scaleY 以及 visible 对于一个看不到的点来说完全没有用的。没有可见的或可缩放的对象。根据这条线索，更深层地思考认识一下这个继承自 Sprite 类的一般类，现在它实际上只是五个变量的容器：xpos, ypos, zpos, x, y。实在有些过分。一个影片有许多功能——发送事件，绘图，容纳其它 sprite 影片等等 —— 并且占用了大量资源。如此使用一个 sprite 影片就像开着坦克到市区里捡一块面包。

如果只需要存储变量的话，不如让我们创建一个只用于存储这些变量作为属性的类。事实上，如果再将透视和坐标旋转再放入这个类中，比单纯只存储几个属性更有意义。女士们、先生们，现在我把这个 Point3D 类送与各位：

```
package {
 public class Point3D {
  public var fl:Number = 250;
  private var vpX:Number = 0;
  private var vpY:Number = 0;
  private var cX:Number = 0;
  private var cY:Number = 0;
  private var cZ:Number = 0;
  public var x:Number = 0;
  public var y:Number = 0;
  public var z:Number = 0;
```

```
    public function Point3D(x:Number=0, y:Number=0, z:Number=0) {
      this.x = x;
      this.y = y;
      this.z = z;
    }
    public function setVanishingPoint(vpX:Number, vpY:Number):void {
      this.vpX = vpX;
      this.vpY = vpY;
    }
    public function setCenter(cX:Number,cY:Number,cZ:Number=0):void {
      this.cX = cX;
      this.cY = cY;
      this.cZ = cZ;
    }
    public function get screenX():Number {
      var scale:Number = fl / (fl + z + cZ);
      return vpX + cX + x * scale;
    }
    public function get screenY():Number {
      var scale:Number = fl / (fl + z + cZ);
      return vpY + cY + y * scale;
    }
    public function rotateX(angleX:Number):void {
      var cosX:Number = Math.cos(angleX);
      var sinX:Number = Math.sin(angleX);
      var y1:Number = y * cosX - z * sinX;
      var z1:Number = z * cosX + y * sinX;
      y = y1;
      z = z1;
    }
    public function rotateY(angleY:Number):void {
      var cosY:Number = Math.cos(angleY);
      var sinY:Number = Math.sin(angleY);
      var x1:Number = x * cosY - z * sinY;
      var z1:Number = z * cosY + x * sinY;
      x = x1;
      z = z1;
    }
    public function rotateZ(angleZ:Number):void {
      var cosZ:Number = Math.cos(angleZ);
      var sinZ:Number = Math.sin(angleZ);
      var x1:Number = x * cosZ - y * sinZ;
      var y1:Number = y * cosZ + x * sinZ;
      x = x1;
      y = y1;
    }
  }
}
```

现在，在使用这个类创建 3D 点时，可以传入 x, y, z 坐标，或者使用公共的 x, y, z 属

性进行设置。类中包涵了一个默认的 fl 属性以及用于存放消失点的属性，可通过 setVanishingPoint() 方法进行设置。使用三种旋转方法，可以绕着某一轴的中心点进行旋转。还可以用 setCenter 方法改变中心点。大家马上会看到这些应用。最后，它还可以为我们处理所有的透视。当我们已经旋转或设置了点的位置时，只需要用 screenX 和 screenY 即可获得透视的点在屏幕上的位置。

这个类会让我们后面两章的程序变得简单许多。

使用这个新的类重新编写上面线条旋转的程序将会惊人的简单，如 Lines3D2.as 所示：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Lines3D2 extends Sprite {
  private var points:Array;
  private var numPoints:uint = 50;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
  private var vpY:Number = stage.stageHeight / 2;
  public function Lines3D2() {
   init();
  }
  private function init():void {
   points = new Array();
   for (var i:uint = 0; i < numPoints; i++) {
    var point:Point3D =
    new Point3D(Math.random() * 200 - 100,
    Math.random() * 200 - 100,
    Math.random() * 200 - 100);
    point.setVanishingPoint(vpX, vpY);
    points.push(point);
   }
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var angleX:Number = (mouseY - vpY) * .001;
   var angleY:Number = (mouseX - vpX) * .001;
   for (var i:uint = 0; i < numPoints; i++) {
    var point:Point3D = points[i];
    point.rotateX(angleX);
    point.rotateY(angleY);
   }
   graphics.clear();
   graphics.lineStyle(0);
   graphics.moveTo(points[0].screenX, points[0].screenY);
   for (i = 1; i < numPoints; i++) {
    graphics.lineTo(points[i].screenX, points[i].screenY);
   }
  }
 }
}
```

主要的改变用粗体表示。应该很容易理解。

创建图形

用随机线条作为示范非常好，但是没有理由不去把这些混乱的点进行有序排列。我们只要将随机创建 x, y, z 值的点变为一些指定的、预先确定的值即可。例如，创建一个正方形。我们绘制的 3D 正方形的四个顶点如图 16-3 所示。

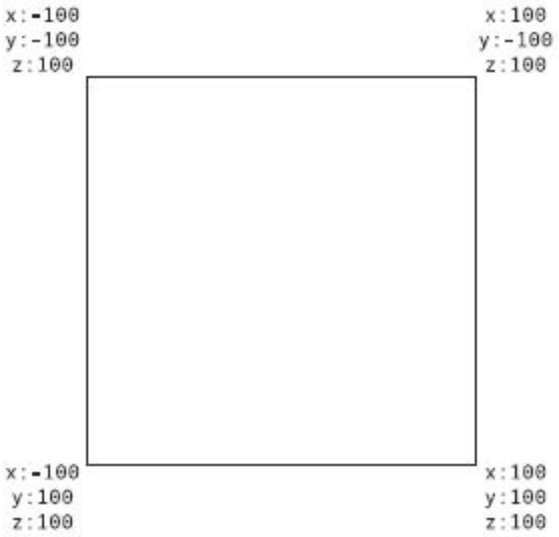


图 16-3　3D 空间中的正方形坐标

注意，所有的 z 值都相同。这是一个平面上的正方形。将一个正方形放在平面上的最简单的办法就是让所有的点在某个轴上的位置都相同（这里我选择了 z 轴），还要定义正方形的其它两个轴（x 和 y 轴）。以下代码将代替循环随机创建的点：

```
points[0] = new Point3D(-100, -100, 100);
points[1] = new Point3D( 100, -100, 100);
points[2] = new Point3D( 100, 100, 100);
points[3] = new Point3D(-100, 100, 100);
```

我们还要手动设置每个点的消失点。在循环中做是再简单不过的了：

```
for (var i:uint = 0; i < numPoints; i++) {
 points[i].setVanishingPoint(vpX, vpY);
}
```

因为现在有四个点，我们将不得不把 numPoints 改为 4。剩下的代码还可以正常工作，但是我们还要多做一步：绘制一条连接最后一个点与第一个点的线，用于将图形封闭。以下是全部代码，可见 Square3D.as（运行结果如图 16-4 所示）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Square3D extends Sprite {
  private var points:Array;
  private var numPoints:uint = 4;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
  private var vpY:Number = stage.stageHeight / 2;
  public function Square3D() {
   init();
  }
```

```
    private function init():void {
      points = new Array();
      points[0] = new Point3D(-100, -100, 100);
      points[1] = new Point3D( 100, -100, 100);
      points[2] = new Point3D( 100, 100, 100);
      points[3] = new Point3D(-100, 100, 100);
      for (var i:uint = 0; i < numPoints; i++) {
        points[i].setVanishingPoint(vpX, vpY);
      }
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onEnterFrame(event:Event):void {
      var angleX:Number = (mouseY - vpY) * .001;
      var angleY:Number = (mouseX - vpX) * .001;
      for (var i:uint = 0; i < numPoints; i++) {
        var point:Point3D = points[i];
        point.rotateX(angleX);
        point.rotateY(angleY);
      }
      graphics.clear();
      graphics.lineStyle(0);
      graphics.moveTo(points[0].screenX, points[0].screenY);
      for (i = 1; i < numPoints; i++) {
        graphics.lineTo(points[i].screenX, points[i].screenY);
      }
      graphics.lineTo(points[0].screenX, points[0].screenY);
    }
  }
}
```

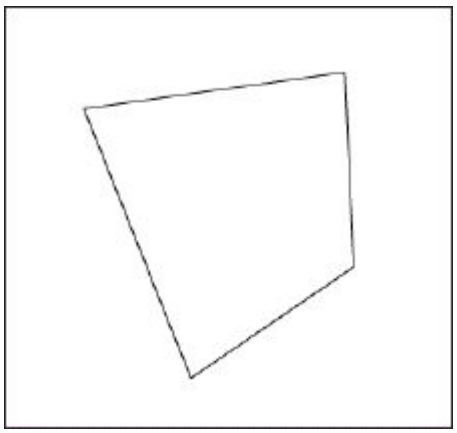


图 16-4　3D 旋转正方形

太奇妙了——一个旋转的正方形！现在我们应该可以创建出任何的平面图形了。我通常都预先将所有的这些点在方格纸上标出（如图 16-5 所示），用来帮助作图。

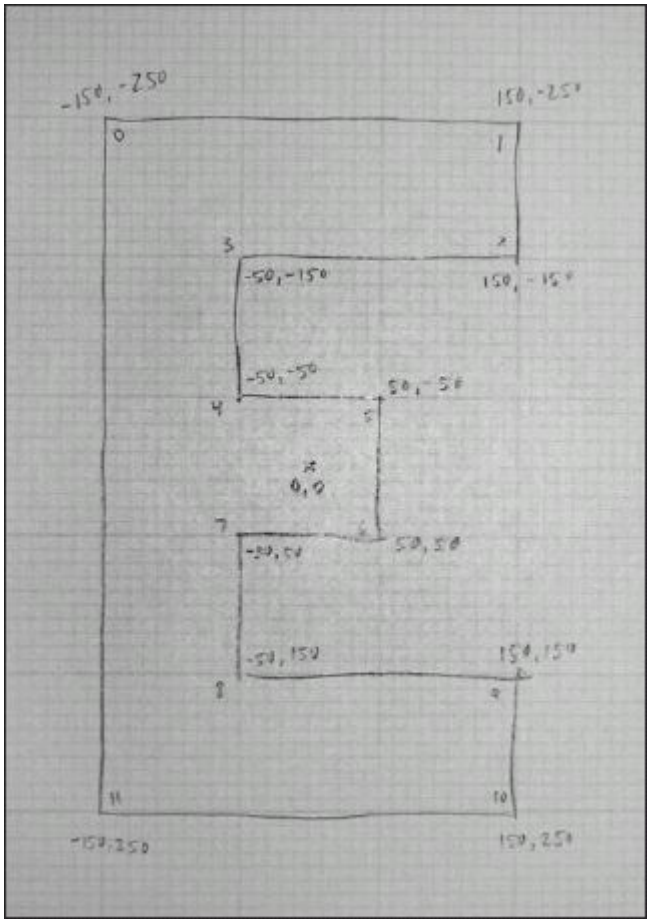


图 16-5 使用方格纸标出所有的点

通过这个草稿，可以用来帮助创建所需的这些点：

```
points[0] = new Point3D(-150, -250, 100);
points[1] = new Point3D( 150, -250, 100);
points[2] = new Point3D( 150, -150, 100);
points[3] = new Point3D( -50, -150, 100);
points[4] = new Point3D( -50, -50, 100);
points[5] = new Point3D( 50, -50, 100);
points[6] = new Point3D( 50, 50, 100);
points[7] = new Point3D( -50, 50, 100);
points[8] = new Point3D( -50, 150, 100);
points[9] = new Point3D( 150, 150, 100);
points[10] = new Point3D( 150, 250, 100);
points[11] = new Point3D(-150, 250, 100);
```

这样就完成了 SpinningE.as 中的旋转字母 E 的效果，运行结果如图 16-6 所示。不要忘记将 numPoints 改为 12。稍后我们要使用 points.length 让这个值可以动态改变。

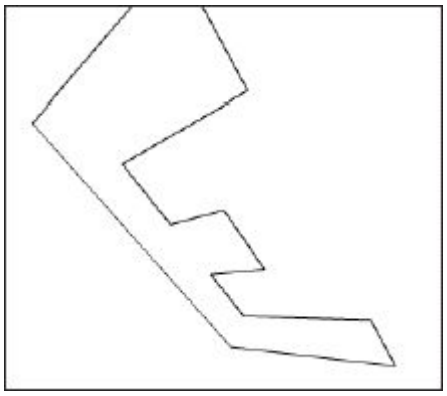


图 16-6 3D 旋转字母 E

执行后我们注意到当字母 E 向我们旋转过来时，某些点会变得非常近，并且与第一个 3D 例子一样会发生倒置。您也许首先会想到增加所有点的 z 值，但这实际上会让效果变得更糟。例如，假设将 z 设为 500。旋转的时候，z 将从 500 到 -500，甚至会让有些点运动到观察点更往后的位置，会让事情更糟糕。不过我们可以使用 setCenter 方法将所有点向 z 轴推移，如下：

```
for (var i:uint = 0; i < numPoints; i++) {
 points[i].setVanishingPoint(vpX, vpY);
 points[i].setCenter(0, 0, 200);
}
```

打开 Point3D 类，我们会看到 scale 的计算方法为：

```
var scale:Number = fl / (fl + z + cZ);
```

这样就将整个系统推移了 200 像素，包括旋转系统。原来 z 的值保持不变，只是因为计算透视的值更高了，那么所有物体都将呈现在观察着的前面。试用其它的值进行替换，观察运动效果。稍后，我们会让这个图形在其它的轴上也能运动起来。

创建 3D 填充

大家也许都想到了，进行填充的准备工作大部分都已经完成了。我们已经创建了图形的所有点，并从头到尾全部用线连接上了。现在所需的全部就是加入 beginFill 和 endFill，如 FilledE.as 文件中的代码。运行结果如图 16-7 所示。代码相应的变化部分用粗体表示：

```
graphics.clear();
graphics.lineStyle(0);
graphics.beginFill(0xffcccc);
graphics.moveTo(points[0].screenX, points[0].screenY);
for (i = 1; i < numPoints; i++) {
 graphics.lineTo(points[i].screenX, points[i].screenY);
}
graphics.lineTo(points[0].screenX, points[0].screenY);
graphics.endFill();
```

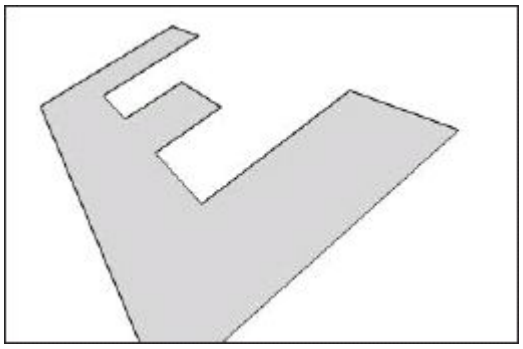


图 16-7 第一个 3D 填充

前面例子中，正方形与字母 E 都是多边形。多边形是由至少三条线段组成的一个封闭的图形。因此，三角形就是一个最简单的多边形。我们发现在许多 3D 建模或渲染的程序中—— 甚至那些使用面片（patches），网格（meshes），非均匀 B 样曲线（NURBS）以及复合多边形 —— 在渲染之前，所有的 3D 图形最终都简化为一个三角集合。

运用三角形

运用三角形有许多好处——或许比我知道的还多，我们这里只做简单的介绍介绍。首先，运用三角形我们能够确定多边形的所有顶点都在一个平面上，因为一个三角形定义一个平面。如果大家还不确定为什么它很重要，那么我们就拿字母 E 的例子来说，随机地改变一些点的 z 值。这时可能会得到一些有趣的结果，这些结果也很快会变成不可预想或不可预

知的。

其次，使用三角形，在绘制复杂的形状时，可能会很简单。例如，考虑一下图 16-8。

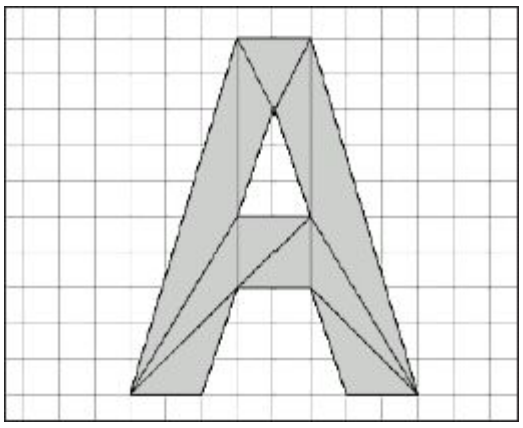


图 16-8 更加复杂的 3D 图形

这样的形状要使用单一的多边形是很难创建的。我们至少要多做两倍的功。同样，还有可能遇到这样的情况，我们创建的每个多边形的顶点都有不同的数值并需要特别的处理。此外，使用三角形可以创建如图 16-9 所示 A 这样的模型。

图 16-9 如图 16-8，用三角形重新渲染

接下来，我们可以创建一个函数，给入三个点即可渲染出一个三角形。我们只需要一个顶点的列表和一个三角形的列表。一层循环顶点列表，设置所有顶点的位置，并应用透视。另一层循环遍历三角形列表并渲染每个三角形。

这并不是说只能使用三角形创建的方法。我们可以也写一个动态渲染多边形的方法。但是这里为了保证简单与灵活，让我们从三角形开始吧。先来试验字母 A 这个例子。

首先，需要定义所需的顶点和三角形。如图 16-10 所示，并为每个三角形的每个顶点编号。

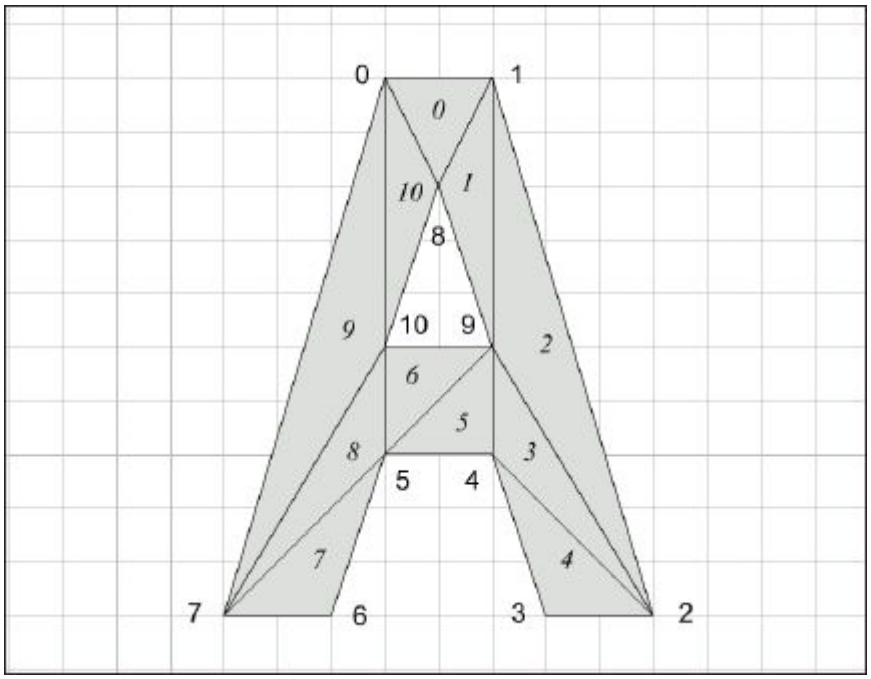


图 16-10 组成该形状的顶点与多面形

标出顶点后，得到如下这些值：

```
points[0] = new Point3D( -50, -250, 100);
points[1] = new Point3D( 50, -250, 100);
points[2] = new Point3D( 200, 250, 100);
points[3] = new Point3D( 100, 250, 100);
points[4] = new Point3D( 50, 100, 100);
```

```
points[5] = new Point3D( -50, 100, 100);
points[6] = new Point3D(-100, 250, 100);
points[7] = new Point3D(-200, 250, 100);
points[8] = new Point3D( 0, -150, 100);
points[9] = new Point3D( 50, 0, 100);
points[10] = new Point3D(-50, 0, 100);
```

下面，需要定义三角形。每个三角形只不过就是三个点的列表；就叫它们 a, b, c 吧。为了能够明确知道每个三角形的所有顶点，我们来创建一个三角形的类（Triangle class）。甚至还可以为每个三角形写一个 draw 方法，让它自己就可以进行绘制。我把代码列出来，稍后会看到如何使用它。

```
package {
 import flash.display.Graphics;
 public class Triangle {
  private var pointA:Point3D;
  private var pointB:Point3D;
  private var pointC:Point3D;
  private var color:uint;
  public function Triangle(a:Point3D, b:Point3D, c:Point3D, color:uint) {
   pointA = a;
   pointB = b;
   pointC = c;
   this.color = color;
  }
  public function draw(g:Graphics):void {
   g.beginFill(color);
   g.moveTo(pointA.screenX, pointA.screenY);
   g.lineTo(pointB.screenX, pointB.screenY);
   g.lineTo(pointC.screenX, pointC.screenY);
   g.lineTo(pointA.screenX, pointA.screenY);
   g.endFill();
  }
 }
}
```

现在我们需要另一个数组来保存三角形的列表。因此，请在类的顶部定义一个数组：

```
private var triangles:Array;
```

然后，在定义了所有的顶点后，再在 init 函数中定义所有三角形。注意每个三角形都用指定的颜色创建。

```
triangles = new Array();
triangles[0] = new Triangle(points[0], points[1], points[8], 0xffcccc);
triangles[1] = new Triangle(points[1], points[9], points[8], 0xffcccc);
triangles[2] = new Triangle(points[1], points[2], points[9], 0xffcccc);
triangles[3] = new Triangle(points[2], points[4], points[9], 0xffcccc);
triangles[4] = new Triangle(points[2], points[3], points[4], 0xffcccc);
triangles[5] = new Triangle(points[4], points[5], points[9], 0xffcccc);
triangles[6] = new Triangle(points[9], points[5], points[10],0xffcccc);
triangles[7] = new Triangle(points[5], points[6], points[7], 0xffcccc);
triangles[8] = new Triangle(points[5], points[7], points[10],0xffcccc);
triangles[9] = new Triangle(points[0], points[10], points[7], 0xffcccc);
triangles[10] = new Triangle(points[0], points[8], points[10],0xffcccc);
```

大家也许注意到了，我将每个三角形的顶点坐标都以顺时针方向排列。这样做对我们当前这个程序来说并不重要，但是在下一章就非常重要了，因此应该养成这样的好习惯。

现在，我们来循环渲染这个图形（别担心，稍后我会给出全部代码），程序如下：

```
graphics.clear();
for (i = 0; i < triangles.length; i++) {
 triangles[i].draw(graphics);
}
```

这里用到了 Triangle 类中定义的 draw 方法，传入一个 graphics 的引用。这样就完成了！三角形根据定义的颜色开始填充，移动到第一个点的位置上，绘制一个图形出来，结束填充。您也许会说，太精致了。

这样来看，我对于有些朋友的面向对象程序设计让编程更复杂的说法要产生质疑了。Triangle 类非常容易理解，主类部分也是再简单不过了。与本书前一版所使用的那个更加复杂并且需要大量解释的线性解决方案相比，这是最好的答案。

本例的运行结果如图 16-11 所示。

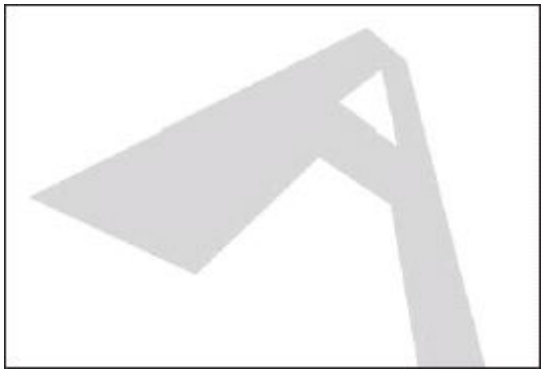


图 16-11　A 形

以下是代码可见 Triangles.as，要确保新的类已经加入了：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Triangles extends Sprite {
  private var points:Array;
  private var triangles:Array;
  private var fl:Number = 250;
  private var vpX:Number = stage.stageWidth / 2;
  private var vpY:Number = stage.stageHeight / 2;
  public function Triangles() {
   init();
  }
  private function init():void {
   points = new Array();
   points[0] = new Point3D( -50, -250, 100);
   points[1] = new Point3D( 50, -250, 100);
   points[2] = new Point3D( 200, 250, 100);
   points[3] = new Point3D( 100, 250, 100);
   points[4] = new Point3D( 50, 100, 100);
   points[5] = new Point3D( -50, 100, 100);
   points[6] = new Point3D(-100, 250, 100);
   points[7] = new Point3D(-200, 250, 100);
   points[8] = new Point3D( 0, -150, 100);
   points[9] = new Point3D( 50, 0, 100);
```

```
      points[10] = new Point3D(-50, 0, 100);
      for (var i:uint = 0; i < points.length; i++) {
        points[i].setVanishingPoint(vpX, vpY);
        points[i].setCenter(0, 0, 200);
      }
      triangles = new Array();
      triangles[0] = new Triangle(points[0], points[1],
      points[8], 0xffcccc);
      triangles[1] = new Triangle(points[1], points[9],
      points[8], 0xffcccc);
      triangles[2] = new Triangle(points[1], points[2],
      points[9], 0xffcccc);
      triangles[3] = new Triangle(points[2], points[4],
      points[9], 0xffcccc);
      triangles[4] = new Triangle(points[2], points[3],
      points[4], 0xffcccc);
      triangles[5] = new Triangle(points[4], points[5],
      points[9], 0xffcccc);
      triangles[6] = new Triangle(points[9], points[5],
      points[10],0xffcccc);
      triangles[7] = new Triangle(points[5], points[6],
      points[7], 0xffcccc);
      triangles[8] = new Triangle(points[5], points[7],
      points[10],0xffcccc);
      triangles[9] = new Triangle(points[0], points[10],
      points[7], 0xffcccc);
      triangles[10] =new Triangle(points[0], points[8],
      points[10],0xffcccc);
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onEnterFrame(event:Event):void {
      var angleX:Number = (mouseY - vpY) * .001;
      var angleY:Number = (mouseX - vpX) * .001;
      for (var i:uint = 0; i < points.length; i++) {
        var point:Point3D = points[i];
        point.rotateX(angleX);
        point.rotateY(angleY);
      }
      graphics.clear();
      for (i = 0; i < triangles.length; i++) {
        triangles[i].draw(graphics);
      }
    }
  }
}
```

最后，认真考虑一下本章开始所说的：在 Flash 中创建 3D 立体模型！

在计算世界里，任何书籍或指南给出的第一个例子基本上都是用某种方法在屏幕上输出“Hello,World”。在 3D 立体编程中，就相当于一个旋转立方体的程序。我们也不要打破这个传统。

建立旋转立方体模型

首先，我们需要八个点来定义立方体的八个角。如图 16-12 所示。

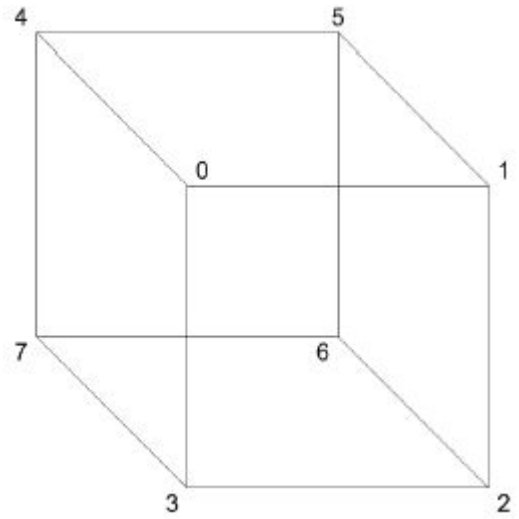


图 16-12　3D 立方体的顶点

代码中点的定义如下：

```
// 前面四个角
points[0] = new Point3D(-100, -100, -100);
points[1] = new Point3D( 100, -100, -100);
points[2] = new Point3D( 100, 100, -100);
points[3] = new Point3D(-100, 100, -100);
// 后面四个角
points[4] = new Point3D(-100, -100, 100);
points[5] = new Point3D( 100, -100, 100);
points[6] = new Point3D( 100, 100, 100);
points[7] = new Point3D(-100, 100, 100);
```

然后我们要定义三角形。立方体的每个面都由两个三角形组成。总共要有 12 个三角形——六个面，每面两个。同样，我会为每个三角形以逆时针的方向排列这些点。这里有个小技巧，试将立方体按照您的意思旋转，让这些三角形面向我们，然后将这些点以观察点的顺时针方向排列。

例如，正面很好办；如图 16-13 所示的两个三角形。

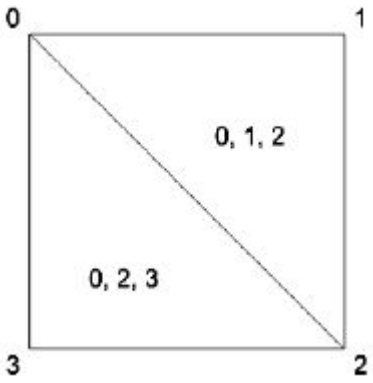


图 16-13 立方体的正面

图 16-14 所示为顶面。图 16-15 所示为背面。

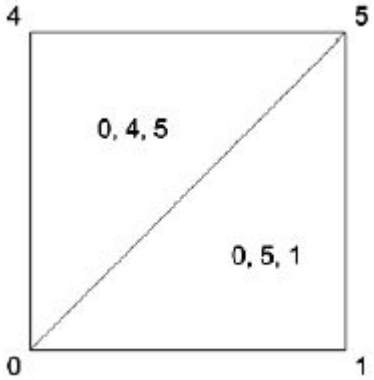


图 16-14 立方体的顶面

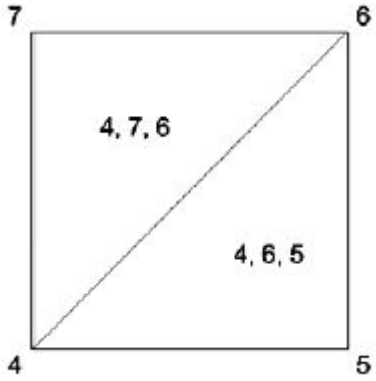


图 16-15 立方体的背面

继续旋转每个面，我们得到三角形的定义如下：

```
// front
triangles[0] = new Triangle(points[0], points[1], points[2], 0x6666cc);
triangles[1] = new Triangle(points[0], points[2], points[3], 0x6666cc);
// top
triangles[2] = new Triangle(points[0], points[5], points[1], 0x66cc66);
triangles[3] = new Triangle(points[0], points[4], points[5], 0x66cc66);
//back
triangles[4] = new Triangle(points[4], points[6], points[5], 0xcc6666);
triangles[5] = new Triangle(points[4], points[7], points[6], 0xcc6666);
// bottom
triangles[6] = new Triangle(points[3], points[2], points[6], 0xcc66cc);
triangles[7] = new Triangle(points[3], points[6], points[7], 0xcc66cc);
// right
triangles[8] = new Triangle(points[1], points[5], points[6], 0x66cccc);
```

```
triangles[9] = new Triangle(points[1], points[6], points[2], 0x66cccc);
// left
triangles[10] =new Triangle(points[4], points[0], points[3], 0xcccc66);
triangles[11] =new Triangle(points[4], points[3], points[7], 0xcccc66);
```

注意每个面都有不同的颜色，由于两个三角形构成一个面，所以它们是同一种颜色。目前来看，是否使用顺时针方向无关紧要，但是在下一章，我们要在背面剔除中用到它。这个术语是指决定哪个面面向我们的一种方法。大家马上会看到为什么如此重要的原因。Cube.as 文件与 Triangles.as 非常像，只不过是在 init 函数中使用了这些新的点和三角形的进行定义。

继续，运行它。哇 —— 一片混乱！有时我们可以看到一些背面。有些面似乎永远也看不可见。为什么会这样呢？立方体的背面总是被绘制出来。同样，那两个三角形也根据它们在 triangles 数组中的位置进行绘制。因此在列表底部的面总是在列表顶部面之前被绘制出来，您会感到奇怪，不可预知的效果就是这样形成的。我们需要调用 cull(剔除)，或铲掉、清除这些背面，因为它们不需要渲染。

我们会在下一章详细地讲解背面剔除，同时还要学到如何根据角度在每个面上应用一些基本的灯光。

作为本章的一个临时修正，我们可以进入到 Triangle 类的 draw 方法，设置填充的透明度为 0.5：

```
public function draw(g:Graphics):void {
  g.beginFill(color, .5);
  g.moveTo(pointA.screenX, pointA.screenY);
  g.lineTo(pointB.screenX, pointB.screenY);
  g.lineTo(pointC.screenX, pointC.screenY);
  g.lineTo(pointA.screenX, pointA.screenY);
  g.endFill();
}
```

这样就任何面都可以看到了，它使整个立方体像是用有色玻璃制成的。在学习背面剔除之前这只是个临时性的修正。完成后的 3D 立方体如图 16-16 所示。

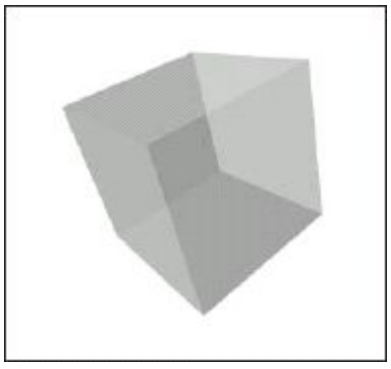


图 16-16　3D 立方体运行结果

建立其它形状的模型

恭喜各位！您已经掌握了旋转立方体。现在我们可以去建立所有种类的形状了。只要先将它们绘制在方格纸上，标出点和三角形，再将放入数组即可。这张图可以帮助我们用几种视角绘出物体，旋转后可以看到每个面以及在三角形上标出的点。本节提供了一些其它的图形作为起点。

金字塔形

下面是 3D 金字塔形的代码（可以在 Pyramid.as 中找到）。首先是点：

```
points[0] = new Point3D( 0, -200, 0);
```

```
points[1] = new Point3D( 200, 200, -200);
points[2] = new Point3D(-200, 200, -200);
points[3] = new Point3D(-200, 200, 200);
points[4] = new Point3D( 200, 200, 200);
```

然后是三角形：

```
triangles[0] = new Triangle(points[0], points[1], points[2], 0x6666cc);
triangles[1] = new Triangle(points[0], points[2], points[3], 0x66cc66);
triangles[2] = new Triangle(points[0], points[3], points[4], 0xcc6666);
triangles[3] = new Triangle(points[0], points[4], points[1], 0x66cccc);
triangles[4] = new Triangle(points[1], points[3], points[2], 0xcc66cc);
triangles[5] = new Triangle(points[1], points[4], points[3], 0xcc66cc);
```

运行结果如图 16-17 所示。

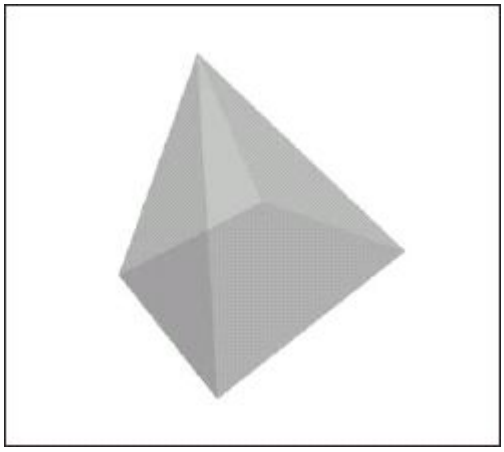


图 16-17 一个 3D 金字塔

挤出的字母 A

在 ExtrudedA.as 中，我用尽全力将前面字母 A 的例子进行了挤压。这就意味着复制前面 11 个点，将一个字母的 z 设置为 -50，另一个设置为 +50，然后为第二个设置三角形（确认它们也是顺时针排列的），最后用三角形连接两边。很乏味？当然！但是完成后的效果非常好：

```
points[0] = new Point3D( -50, -250, -50);
points[1] = new Point3D( 50, -250, -50);
points[2] = new Point3D( 200, 250, -50);
points[3] = new Point3D( 100, 250, -50);
points[4] = new Point3D( 50, 100, -50);
points[5] = new Point3D( -50, 100, -50);
points[6] = new Point3D(-100, 250, -50);
points[7] = new Point3D(-200, 250, -50);
points[8] = new Point3D( 0, -150, -50);
points[9] = new Point3D( 50, 0, -50);
points[10] = new Point3D( -50, 0, -50);
points[11] = new Point3D( -50, -250, 50);
points[12] = new Point3D( 50, -250, 50);
points[13] = new Point3D( 200, 250, 50);
points[14] = new Point3D( 100, 250, 50);
points[15] = new Point3D( 50, 100, 50);
points[16] = new Point3D( -50, 100, 50);
```

```
points[17] = new Point3D(-100, 250, 50);
points[18] = new Point3D(-200, 250, 50);
points[19] = new Point3D( 0, -150, 50);
points[20] = new Point3D( 50, 0, 50);
points[21] = new Point3D( -50, 0, 50);
triangles[0] =new Triangle(points[0], points[1], points[8], 0x6666cc);
triangles[1] =new Triangle(points[1], points[9], points[8], 0x6666cc);
triangles[2] =new Triangle(points[1], points[2], points[9], 0x6666cc);
triangles[3] =new Triangle(points[2], points[4], points[9], 0x6666cc);
triangles[4] =new Triangle(points[2], points[3], points[4], 0x6666cc);
triangles[5] =new Triangle(points[4], points[5], points[9], 0x6666cc);
triangles[6] =new Triangle(points[9], points[5], points[10], 0x6666cc);
triangles[7] =new Triangle(points[5], points[6], points[7], 0x6666cc);
triangles[8] =new Triangle(points[5], points[7], points[10], 0x6666cc);
triangles[9] =new Triangle(points[0], points[10], points[7], 0x6666cc);
triangles[10] = new Triangle(points[0], points[8], points[10], 0x6666cc);
triangles[11] = new Triangle(points[11], points[19], points[12], 0xcc6666);
triangles[12] = new Triangle(points[12], points[19], points[20], 0xcc6666);
triangles[13] = new Triangle(points[12], points[20], points[13], 0xcc6666);
triangles[14] = new Triangle(points[13], points[20], points[15], 0xcc6666);
triangles[15] = new Triangle(points[13], points[15], points[14], 0xcc6666);
triangles[16] = new Triangle(points[15], points[20], points[16], 0xcc6666);
triangles[17] = new Triangle(points[20], points[21], points[16], 0xcc6666);
triangles[18] = new Triangle(points[16], points[18], points[17], 0xcc6666);
triangles[19] = new Triangle(points[16], points[21], points[18], 0xcc6666);
triangles[20] = new Triangle(points[11], points[18], points[21], 0xcc6666);
triangles[21] = new Triangle(points[11], points[21], points[19], 0xcc6666);
triangles[22] = new Triangle(points[0], points[11], points[1], 0xcccc66);
triangles[23] = new Triangle(points[11], points[12], points[1], 0xcccc66);
triangles[24] = new Triangle(points[1], points[12], points[2], 0xcccc66);
triangles[25] = new Triangle(points[12], points[13], points[2], 0xcccc66);
triangles[26] = new Triangle(points[3], points[2], points[14], 0xcccc66);
triangles[27] = new Triangle(points[2], points[13], points[14], 0xcccc66);
triangles[28] = new Triangle(points[4], points[3], points[15], 0xcccc66);
triangles[29] = new Triangle(points[3], points[14], points[15], 0xcccc66);
triangles[30] = new Triangle(points[5], points[4], points[16], 0xcccc66);
triangles[31] = new Triangle(points[4], points[15], points[16], 0xcccc66);
triangles[32] = new Triangle(points[6], points[5], points[17], 0xcccc66);
triangles[33] = new Triangle(points[5], points[16], points[17], 0xcccc66);
triangles[34] = new Triangle(points[7], points[6], points[18], 0xcccc66);
triangles[35] = new Triangle(points[6], points[17], points[18], 0xcccc66);
triangles[36] = new Triangle(points[0], points[7], points[11], 0xcccc66);
triangles[37] = new Triangle(points[7], points[18], points[11], 0xcccc66);
triangles[38] = new Triangle(points[8], points[9], points[19], 0xcccc66);
triangles[39] = new Triangle(points[9], points[20], points[19], 0xcccc66);
triangles[40] = new Triangle(points[9], points[10], points[20], 0xcccc66);
triangles[41] = new Triangle(points[10], points[21], points[20], 0xcccc66);
triangles[42] = new Triangle(points[10], points[8], points[21], 0xcccc66);
triangles[43] = new Triangle(points[8], points[19], points[21], 0xcccc66);
```

运行结果如图 16-8 所示。

图 16-8 一个被挤出的字母 A

如同我们看到的，这些东西建立得很快。最初，平面 A 有 11 个三角形。挤出后是原来的四倍！代码仍然运行得非常稳定，但是我们已进入了 Flash 的宏大的 3D 世界。相同的程序在 AS 2 中运行得也非常好。使用 AS 3 效率会变得更高，目前我不认为已经到达了它速度的极限。随着 Flay Player 性能的改进，未来会怎样，谁知道呢？

圆筒形

再来一个图形。这次我将展示如何使用 Math 创建点和三角形。在 Cylinder.as 中唯一改变的就是 init 函数（我在上面加入了 numFaces 变量）。取代手工定义点和三角形，我创建了一个算法，并用它来完成创建圆筒的工作。以下是 init 函数：

```
private var numFaces:uint = 20;
private function init():void {
 points = new Array();
 triangles = new Array();
 var index:uint = 0;
 for (var i:uint = 0; i < numFaces; i++) {
  var angle:Number = Math.PI * 2 / numFaces * i;
  var xpos:Number = Math.cos(angle) * 200;
  var ypos:Number = Math.sin(angle) * 200;
  points[index] = new Point3D(xpos, ypos, -100);
  points[index + 1] = new Point3D(xpos, ypos, 100);
  index += 2;
 }
 for (i = 0; i < points.length; i++) {
  points[i].setVanishingPoint(vpX, vpY);
  points[i].setCenter(0, 0, 200);
 }
 index = 0;
 for (i = 0; i < numFaces - 1; i++) {
  triangles[index] = new Triangle(points[index],
  points[index + 3],
  points[index + 1],
  0x6666cc);
  triangles[index + 1] = new Triangle(points[index],
  points[index + 2],
  points[index + 3],
```

```
    0x6666cc);
    index += 2;
  }
  triangles[index] = new Triangle(points[index],
  points[1],
  points[index + 1],
  0x6666cc);
  triangles[index+1] = new Triangle(points[index],
  points[0],
  points[1],
  0x6666cc);
  addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

我知道，这段代码不是很容易理解，让我们先快速看一遍，解释工作也许要画上一两张图。

首先要绕圆一圈，隔一段距离创建一个点。每次循环，首先用圆周除以面的个数，再乘以指定的面，获得一个角度。

使用这个角度，加上三角学可以确定该点在圆上的 x,y 坐标。然后创建两个点，一个 z 轴为 -100，另一个 z 轴为 +100。当循环结束时，我们将有两个带点的圆环，一个离我们近一些，另一个离我们远一些。现在，只需要将它们连接成三角形。

同样，需要循环每个面。这时，每面需要创建两个三角形。从侧面观察，第一个面如图 16-19 所示。

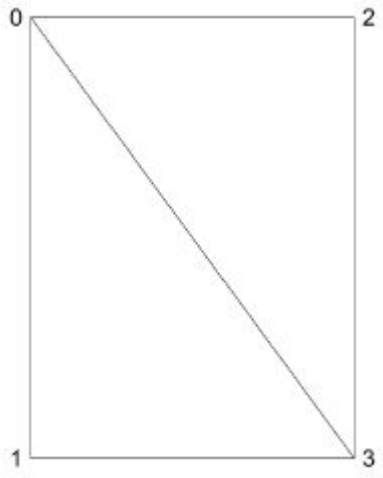


图 16-19 圆筒形的第一个面

这里形成了两个三角形：

```
0, 3, 1
0, 2, 3
```

因为 index 变量为 0，我们也可以这样定义：

```
index, index + 3, index + 1
index, index + 2, index + 3
```

这就是定义两个三角形的方法。然后让 index 加 2，拿到下一个面的点 2, 3, 4, 5。这样一直做到倒数第二个面，然后连接最后一个点回到前两个点 0 和 1 上，如图 16-20 所示。

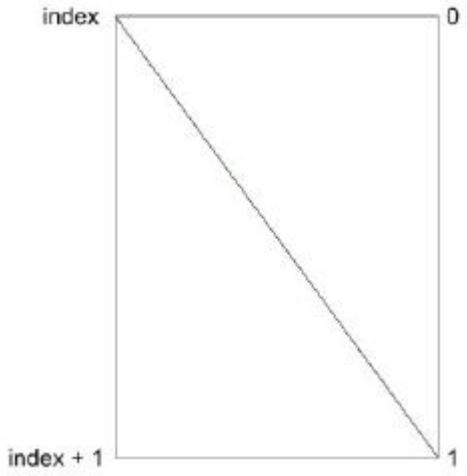


图 16-20 圆筒形的最后一个面

这样就得到：

```
index, 1, index + 1
index, 0, 1
```

运行结果如图 16-21 所示。如果您想让外形更清晰，可以进入 Triangle 类中加入一行 lineStyle 语句，这样可以绘出每个三角形。

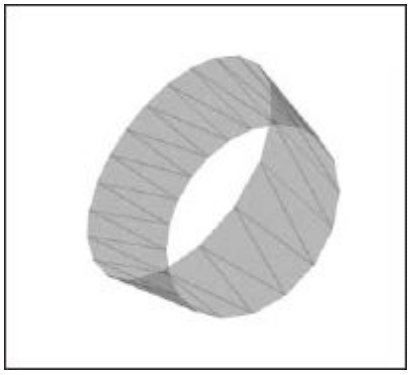


图 16-21 3D 圆筒运行结果

移动三维立体模型

我们已经有了 Point3D 类，要移动一个 3D 立体模型非常简单。只需要用 setCenter 方法改变中心位置。我们已经在 z 轴上做过图形的移动。这次只需要在其它轴上做同样的事情即可。但是，请快速看一下，当改变这些值的时候，执行的代码是什么样的。它们都在 Point3D 类的 screenX 和 screenY 方法中：

```
public function get screenX():Number {
  var scale:Number = fl / (fl + z + cZ);
  return vpX + cX + x * scale;
}
public function get screenY():Number {
  var scale:Number = fl / (fl + z + cZ);
  return vpY + cY + y * scale;
}
```

z 属性的中心 cZ，被加到 z 坐标上，当 scale 计算好的情况下，它将该点向外推出一段距离，并不真正地改变 z 本身的位置。

同样的事情也发生在 cX 和 cY 上。它们在使用 scale 之前被加到了 x 和 y 坐标上。将一个方向上的点向外推，同样，不会永久改变它们。由于我们从没改变过这些值，这个点 —— 因此会使形状变大 —— 将继续绕着自身的中心旋转，而不是绕着整个世界的中心旋

转。
我们来看一下代码。回到 Cube.as 类，在顶部加入两个类属性：

```
private var offsetX:Number = 0;
private var offsetY:Number = 0;
```

并在 init 方法中，加入这样一行：

```
    stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
```

然后创建处理方法：

```
private function onKeyDown(event:KeyboardEvent):void {
 if (event.keyCode == Keyboard.LEFT) {
  offsetX = -5;
 } else if (event.keyCode == Keyboard.RIGHT) {
  offsetX = 5;
 } else if (event.keyCode == Keyboard.UP) {
  offsetY = -5;
 } else if (event.keyCode == Keyboard.DOWN) {
  offsetY = 5;
 }
 for (var i:Number = 0; i < points.length; i++) {
  points[i].x += offsetX;
  points[i].y += offsetY;
 }
}
```

不要忘记导入 KeyboardEvent 和 Keyboard 类。

这些不过是循环每个点并在它们的值上加上或减去 5。由于所有点的实际坐标都在改变，模型就像绕着 3D 空间的中心在旋转。这也许是或不是您想要的。如果我们只想移动整个模型，并且还想让它绕着自身的中心点旋转，这就是 setCenter 方法出场的时候了。将 onKeyDown 中的代码做如下改变：

```
private function onKeyDown(event:KeyboardEvent):void {
 if (event.keyCode == Keyboard.LEFT) {
  offsetX -= 5;
 } else if (event.keyCode == Keyboard.RIGHT) {
  offsetX += 5;
 } else if (event.keyCode == Keyboard.UP) {
  offsetY -= 5;
 } else if (event.keyCode == Keyboard.DOWN) {
  offsetY += 5;
 }
 for (var i:Number = 0; i < points.length; i++) {
  points[i].setCenter(offsetX, offsetY, 200);
 }
}
```

现在立方体运动起来就像一个整体，继续绕着它的中心点旋转，而不是绕着整个世界的中心旋转。

16.6 小结

通过本章的学习，你应当可以在 ActionScript 中创建你自己的 3D 模型并在 3D 空间中控制它们。

在这一章中讲的是模型的外形以及如何绘制线条和填充。在下一章中，你将研究如何创建更丰富的实体模型。所有的内容都是在你自己学过的点、线和填充的基础上创建的。因此，如果你准备好了，我们就继续吧。

第十七章　背面剔除与 3D 灯光

第十六章介绍了所有建立 3D 立体模型的基础包括：如何创建点，线，用多边形组成各种形状，以及如何为每个多边形设置颜色。但是，我们只能让颜色的透明度为 50%，才能看到正确的效果。虽然制作出的 3D 立体模型也不差，但是这样做在真实度上还是有所欠缺。

本章，通过介绍背面剔除（不绘制背面的多边形），深度排序（第十五章作了一点介绍，但这次要从多边形的角度重新审视它），以及 3D 灯光，来修正这个问题。

我想应用了这三种技术所得到的 3D 模型一定会让大家感到惊讶。学习过前两项技术后，我们将能够创建出看上去非常真实的 3D 立体模型。使用 3D 灯光可以让它们变得活灵活现。

在开始之前，我想先声明一下技术的来源。本章几乎所有的代码都来自 Todd Yard 的 Macromedia Flash MX Studio 一书第十章的技术。（Todd 也是本书的技术评论员，因此这就不像未经他人允许剽窃过来的！）Macromedia Flash MX Studio 也许是我所能找到最适合本章的专用材料，每当我要用到这些高级 3D 技术时都会去查阅它。因为我们这本书已经有四年多的时间了（按照技术书籍来算，是本古老的书了）跨越了 Flash 的三个版本，我非常愿意能够让本书的内容继续发展下去并且能够与时俱进。

本章建立在前一章的旋转的、挤出的、3D 字母 A 的基础上。这会使代码非常复杂，只要有一点点错误都会非常明显，当一切都正确时获得的效果会是超级棒的哟！

背面剔除

背面剔除在前一章提到很多次了。下面，我们将了解它的一切以及弄清它是如何工作的。

别忘了，我们早前的模型都是半透明填充的。原因是每次都在绘制多边形，但是没有控制绘制的顺序。因此一个处于模型后面的多边形绘制在了模型的前面，造成了一些奇怪的结果。在集中讨论建模基础技术时，为了将这一部分的讨论向后推迟，我们将所有多边形的透明度都设置为 50%。现在到了该处理它的时候了。

总体而言，背面剔除是非常简单的。只绘制面向我们的多边形，不绘制背面的。这个技术正如其名。大家应该还记得每个多边形的点都是顺时针安排的。虽然在前面的例子中是完全没有必要的，但是我们即将看到它为什么如此重要，以及为什么从一开始就要养成这样的习惯。

很明显，如果面向我们的多边形的点是顺时钟安排的，那么当它们转而背对我们时它们都将是逆时针排列的。

如图 17-1 所示一个面向我们的三角形。（因为我们使用的所有多边形都是三角形，所以我常常混淆这两个词。多数情况下，我会使用“多边形”作为一个总称，而“三角形”只是一个特殊的有三个角的多边形）。

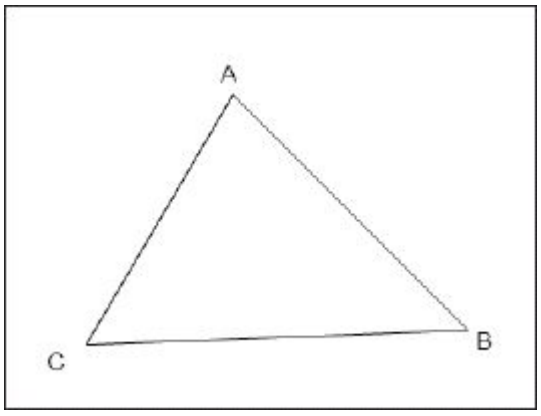


图 17-1 一个面向我们的三角形的点以顺时针方向排列

在图 17-2 中，我将这个三角形进行了旋转，看到了它相反的方向。

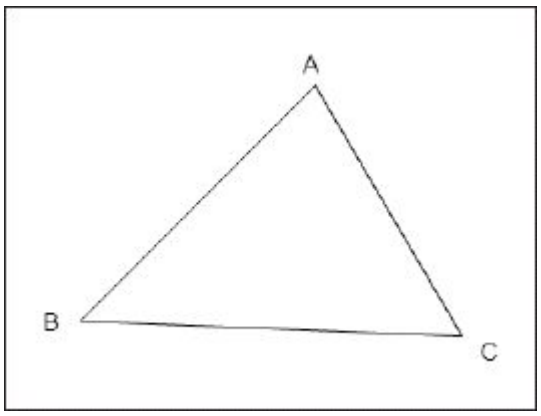


图 17-2 一个背对我们的三角形以逆时针方向排列

我们已经看到逆时针方向点的排列情况。

这里有一点需要澄清。首先，我说的一个多边形“向面我们”是什么意思？意思是多边形的外部面面向我们。虽然我在展示单一的一个三角形时看起来并不明显，但是别忘了我们是在讨论 3D 立体模型。这样一来，每个多边形都有一个外部表面和一个内部面。

另一点是，在确定顺时针或逆时针方向时，我们是在讨论点的屏幕上的位置。而不是 3D x, y, z 坐标，而是经过透视后确定的 screenX, screenY 坐标。

最后，还应该注意到我们可以将整个设置颠倒过来，让一个逆时针多边形系统面向我们，让一个顺时针系统背对我们。每种方式都可以像平时一样工作。

回到问题上，如何确定三个点是以顺时针安排还是以逆时针安排？请思考一下这个问题。这个问题对于眼睛来说非常好回答，但是要它们变为代码，立即就变成了一个非常抽象的概念。

刚刚说过我给大家的解决办法来自 Macromedia Flash MX Studio。请相信我提供给你的一定是尽可能最好的方法，我也试着用自己写的函数来区分顺时针和逆时针方法。我将所有运行正常的程序放到一起，结果我的方法比现有的解决方法要长上两倍，并且非常复杂，其中还包括了坐标旋转和其它三角函数。由于我通常不想作恶人，所以我决定给大家那个简单的解决方法。该方法与我的那个极度复杂的方法同样精确！我们所要做的就是将这个方法加入 Triangle 类，名为 isBackFace。它会求出三角形三个点的值，如果是逆时针的方向则返回 true 如果顺时针则返回 false。以下是加入了该函数的类。同样注意在调用 beginFill 时不再使用 .5 的透明度了。现在开始我们就要绘制不透明的形状了。

```
package {
 import flash.display.Graphics;
 public class Triangle {
  private var pointA:Point3D;
  private var pointB:Point3D;
  private var pointC:Point3D;
  private var color:uint;
  public function Triangle(a:Point3D, b:Point3D, c:Point3D, color:uint) {
   pointA = a;
   pointB = b;
   pointC = c;
   this.color = color;
  }
  public function draw(g:Graphics):void {
   if (isBackFace()) {
    return;
   }
   g.beginFill(color);
```

```
      g.moveTo(pointA.screenX, pointA.screenY);
      g.lineTo(pointB.screenX, pointB.screenY);
      g.lineTo(pointC.screenX, pointC.screenY);
      g.lineTo(pointA.screenX, pointA.screenY);
      g.endFill();
    }
    private function isBackFace():Boolean {
      // 见 http://www.jurjans.lv/flash/shape.html
      var cax:Number = pointC.screenX - pointA.screenX;
      var cay:Number = pointC.screenY - pointA.screenY;
      var bcx:Number = pointB.screenX - pointC.screenX;
      var bcy:Number = pointB.screenY - pointC.screenY;
      return cax * bcy > cay * bcx;
    }
  }
}
```

我们看到这个网站的链接，http://www.jurjans.lv/flash/shape.html。Todd Yard 在 Macromedia Flash MX Studio 中推荐了这个网站，所以我将它发给大家。除此之外，这个网站还有一些很棒的参考资料以及类似主题的指南。

快速地解释一下，这个函数计算了三角形两条边的长度，并使用乘法和比较的方式，告诉我们多边形是向哪个方向运动的。如果您对它的实现很感兴趣，就去看看刚才说到的那个网站，或者搜索一下“背面剔除”["Backface culling"]。我保证您会找到大量的阅读资料。现在，只能说它是个简单、快速、高效并且 100% 可以使用的函数，将它放在这里即可！

那么如何使用这个方法呢？我们并不需要真正去考虑它的实现。这是一个私有方法，意味着只有在 Triangle 类中才能调用它。它可以作为 draw 方法的第一句被调用。如果返回值为 true，则三角形是背面，不应绘制，因此 draw 方法停止并返回。如果 isBackFace 返回值为 false，三角形是正面，则像平常一样被绘制出来。

现在可以运行一下 ExtrudedA.as 或其它已经创建的 3D 模型，我们会发现情况发生了一点点变化。当旋转这个模型时，我们看到一旦某个面转向了反方向，那么它将不再被绘制。目前为止效果还不够好，因为这里仍然有些离得远的部分绘制在了距离较近的部分的前面，不过这是我们曾经学过的。如果“z 排序”或“深度排序”一词出现在您的脑海里，那就对了。这就是下一节的内容。

现在我们应该可以看到如图 17-3 所示的效果。

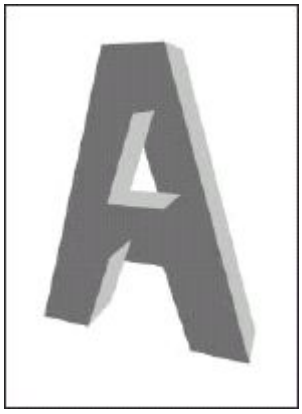


图 17-3 运行中的背面剔除

深度排序

深度排序，或叫做 z 排序，我们已经在第十五章为影片应用透视时讨论过。在那个例

子中，为一个 Sprite 影片数组通过 zpos 属性进行排序（确切地说是一个拥有 3D 属性的 Sprite 的子类）。

而现在我们不是在处理多个影片。所有的多边形都使用主类中相同的 graphics 对象进行绘制。因此，无论何时绘制多边形，它都将绘制在前一个图形的上面。与其去改变它们的深度，不如决定何时绘制这个多边形。具体来讲，我们要先绘制距离最远的；然后再依次绘制其余的，这样最近的多边形会在最后被绘制，盖住所有可能放到前面的图形。

如何实现呢？在这个名为 triangles 的数组中存着所有多边形。当绘制图形时，循环 triangles，从第 0 个到最后一个依次绘制每个三角形。我们要做的就是将这个数组进行排序以便让最远的三角形为数组的第 0 个元素，距离观察者最近的成为最后一个元素。

这与 Sprite 影片数组的排序非常相似。但是本例子中，三角形只是三个 Point3D 对象的集合。它们没有一个专门描述整个三角形深度的属性。但是要创建这个属性也相当容易。计算这个值最好的方法就是找到三角形的三个点里面最小的 z 值。换句话讲，如果一个三角形的三个顶点深度分别为 200, 250, 300，就可以说这个三角形的 z 坐标为 200。我们可以使用 Math.min 方法来确定三个点最小的 z 值。但需要使用两次，因为一次只能传入两个数值。我们将在 Triangle 类的 depth 方法中进行实现。以下是更新的类：

```
package {
 import flash.display.Graphics;
 public class Triangle {
  private var pointA:Point3D;
  private var pointB:Point3D;
  private var pointC:Point3D;
  private var color:uint;
  public function Triangle(a:Point3D, b:Point3D, c:Point3D, color:uint) {
   pointA = a;
   pointB = b;
   pointC = c;
   this.color = color;
  }
  public function draw(g:Graphics):void {
   if (isBackFace()) {
    return;
   }
   g.beginFill(color);
   g.moveTo(pointA.screenX, pointA.screenY);
   g.lineTo(pointB.screenX, pointB.screenY);
   g.lineTo(pointC.screenX, pointC.screenY);
   g.lineTo(pointA.screenX, pointA.screenY);
   g.endFill();
  }
  private function isBackFace():Boolean {
   // 见 http://www.jurjans.lv/flash/shape.html
   var cax:Number = PointC.screenX - PointA.screenX;
   var cay:Number = PointC.screenY - PointA.screenY;
   var bcx:Number = PointB.screenX - PointC.screenX;
   var bcy:Number = PointB.screenY - PointC.screenY;
   return cax * bcy > cay * bcx;
  }
  public function get depth():Number {
   var zpos:Number = Math.min(PointA.z, PointB.z);
```

```
      zpos = Math.min(zpos, PointC.z);
      return zpos;
    }
  }
}
```

现在我们就能够知道哪个在前哪个在后了，可以为 triangles 数组排序了。同样，要让数组降序排列，让深度最高的（最远的）在第一个。这个操作要在文档类中完成，并且在绘制三角形之前进行。这里我使用 ExtrudedA 类，因此这个类的 onEnterFrame 方法像是这样：

```
private function onEnterFrame(event:Event):void {
  var angleX:Number = (mouseY - vpY) * .001;
  var angleY:Number = (mouseX - vpX) * .001;
  for (var i:uint = 0; i < points.length; i++) {
    var point:Point3D = points[i];
    point.rotateX(angleX);
    point.rotateY(angleY);
  }
  triangles.sortOn("depth", Array.DESCENDING | Array.NUMERIC);
  graphics.clear();
  for (i = 0; i < triangles.length; i++) {
    triangles[i].draw(graphics);
  }
}
```

这里只需改变一行代码。面向对象编程的奇迹让我们的生活更简单了！

运行这个程序后，我们将看到了一个被完美渲染出来的立体模型，如图 17-4 所示。现在我们已经真正达到了某种境界。下面将带大家步入最 Cool 的巅峰！

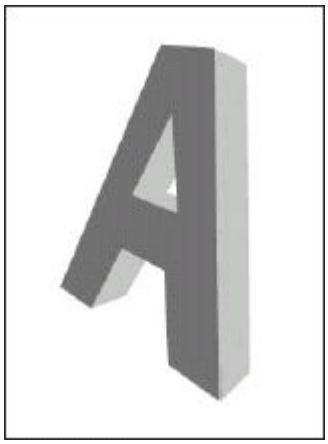


图 17-4 深度排序修正了错误

3D 灯光

刚刚这个例子近乎可以让我们的渲染达到完美的效果，但是它似乎还缺少点儿什么。有些单调。OK，OK，大家看到标题就已经知道了，下面就让我们加入 3D 的灯光效果吧。

同背面剔除一样，3D 灯光的细节也是相当复杂并且需要数学运算的。我实在没有太多的空间讨论每个漂亮的细节，但是通过快速的网络搜索大家可以获得非常更多的相关资料，也许这些资料多得我们一生也看不完。在这里，我给大家的都是一些基础的需要用到的函数。

首先，需要一个光源。一个最简单的光源只有两个属性：位置和亮度（brightness）。在更加复杂的 3D 系统中，它也能够指向某个方向，并且还带有颜色，衰减率（falloff rate），圆锥区域等等。但是这些都超出了本例的范围。

让我们从制作一个 Light 灯光类开始。它会持有我们刚说的那两个属性 —— 位置和亮度。

```
package {
 public class Light {
  public var x:Number;
  public var y:Number;
  public var z:Number;
  private var _brightness:Number;
  public function Light(x:Number = -100,
  y:Number = -100,
  z:Number = -100,
  brightness:Number = 1) {
   this.x = x;
   this.y = y;
   this.z = z;
   this.brightness = brightness;
  }
  public function set brightness(b:Number):void {
   _brightness = Math.max(b, 0);
   _brightness = Math.min(_brightness, 1);
  }
  public function get brightness():Number {
   return _brightness;
  }
 }
}
```

现在可以在主类的 init 方法中创建一个新的默认灯光：

```
var light:Light = new Light();
```

或者可以创建一个指定位置和区域的灯光：

```
var light:Light = new Light(100, 200, 300, .5);
```

这里有两个重要的地方需要注意。一个是位置，仅用于计算灯光的角度。灯光的亮度不会因为距离而衰减。因此改变 x, y, z 到 －1,000,000 或 -1 对于照射在物体上的灯光的亮度是没有区别的。

只有 brightness 属性才会改变灯光的特性。我们当然可以加入一个函数用以判断灯光与物体间的距离来计算灯光的亮度值（brightness）。不会很难，现在已经介绍得差不多了，因此把这个函数留给大家去做。

brightness 必需是 0.0 到 1.0 之间的数。如果出了这个范围，会带来一些奇怪的结果。就是这个原因，我创建了一个私有属性 _brightness，并允许通过公共的 getter 和 setter 访问 brightness。这样做，允许我们传入的数值得到有效性的验证，确保这个数在有效范围内。

一个理想的类不应该出现公有的属性，即使这些属性不需要验证，也只有私有属性通过 getter 和 setter 函数才能访问。这里我抄了近路，为的是让代码简洁并突出动画编程的原则。但是在本例中，额外添加的这一步是有必要的。

下面，光源要做的就是根据灯光照射在多边形上的角度来改变三角形颜色的亮度值。因此如果一个多边形直接面对灯光，它就会显示出全部的颜色值。当离开灯光时，就会变得越来越暗。最终，当它完全离开光源时，它将完全变为阴影或黑色。

由于 Triangle 类的成员知道自己的颜色是什么，并知道如何绘制自己，似乎每个三角形只需访问这个 light 就可以实现自己 draw 函数。因此，让我们给所有三角形一个 light 属性。我还要超个近路设置它们为公有属性：

```
public var light:Light;
```

然后在主类中，创建这些三角形后，只需要循环它们把灯光的引用赋值给每个三角形：

```
var light:Light = new Light();
```

```
for(i = 0; i < triangles.length; i++) {
  triangles[i].light = light;
}
```

或者，我们也可以让 light 作为 Triangle 构造函数中的一个附加的参数，让每个三角形都有一个光源。我将这个方法留给大家去选择。

现在，Triangle 需要一个关于其灯光颜色、角度、亮度的函数，并返回一个调整后的颜色值。以下是这个函数：

```
function getAdjustedColor():uint {
  var red:Number = color >> 16;
  var green:Number = color >> 8 & 0xff;
  var blue:Number = color & 0xff;
  var lightFactor:Number = getLightFactor();
  red *= lightFactor;
  green *= lightFactor;
  blue *= lightFactor;
  return red << 16 | green << 8 | blue;
}
```

这个函数首先将三角形的基本颜色分为了 red, green, blue 三个成分（见第四章）。然后调用另一个方法 getLightFactor，稍后会看到这个函数。现在，只需要知道它返回的是 0.0 到 1.0 之间的一个数，表示该颜色需要改变的大小，1.0 表示全部亮度，0.0 表示为全黑色。

然后将每个颜色成分乘以这个滤光系数（light factor），最后再将它们组合为一个 24 位的颜色值，并且将它作为调整后的颜色返回。它将成为灯光照射下三角形的颜色。

现在，如何得到这个 lightFactor 呢？让我们看一下：

```
private function getLightFactor():Number {
  var ab:Object = new Object();
  ab.x = pointA.x - pointB.x;
  ab.y = pointA.y - pointB.y;
  ab.z = pointA.z - pointB.z;
  var bc:Object = new Object();
  bc.x = pointB.x - pointC.x;
  bc.y = pointB.y - pointC.y;
  bc.z = pointB.z - pointC.z;
  var norm:Object = new Object();
  norm.x = (ab.y * bc.z) - (ab.z * bc.y);
  norm.y = -((ab.x * bc.z) - (ab.z * bc.x));
  norm.z = (ab.x * bc.y) - (ab.y * bc.x);
  var dotProd:Number = norm.x * light.x +
  norm.y * light.y +
  norm.z * light.z;
  var normMag:Number = Math.sqrt(norm.x * norm.x +
  norm.y * norm.y +
  norm.z * norm.z);
  var lightMag:Number = Math.sqrt(light.x * light.x +
  light.y * light.y +
  light.z * light.z);
  return Math.acos(dotProd / normMag * lightMag) / Math.PI * light.brightness;
}
```

哇，好大一个函数不是吗？要想完全理解它，就一定要对高等向量学有较深的掌握，但是我也会试将基础的地方解释一下。

首先，我们需要找到三角形的法线（normal）。它是一个向量，是三角形平面上的一条垂线，如图 17-5 所示。想象一下，我们拿着一块木制的三角板，然后从背后钉入一根钉子，它会从正面穿出。这根钉子就代表三角形平面的法线。如果您学过 3D 渲染和灯光的话，一定看过各种关于法线的资料。

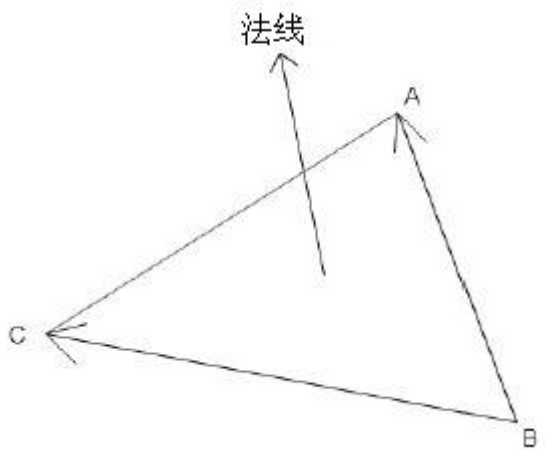


图 17-5 法线是到达三角形表面的一条垂线

我们可以通过该平面的两个向量计算出它们的外积（cross product）从而求出这条法线。两个向量的积是一条垂直于这两条向量的新向量。我们将使用的这两条向量是点 A 和 B，点 B 和 C 之间的连线。每个向量都用有带有 x, y, z 的 Object 持有。

```
var ab:Object = new Object();
ab.x = pointA.x - pointB.x;
ab.y = pointA.y - pointB.y;
ab.z = pointA.z - pointB.z;
var bc:Object = new Object();
bc.x = pointB.x - pointC.x;
bc.y = pointB.y - pointC.y;
bc.z = pointB.z - pointC.z;
```

然后计算法线，即另一个向量。求该对象的模（norm）。下面的代码用于计算向量 ab 和 bc 的外积：

```
var norm:Object = new Object();
norm.x = (ab.y * bc.z) - (ab.z * bc.y);
norm.y = -((ab.x * bc.z) - (ab.z * bc.x));
norm.z = (ab.x * bc.y) - (ab.y * bc.x);
```

我没有太多的篇幅来介绍这种计算方法的细节，这是计算向量外积的标准公式。如果您对它的推导感兴趣，可以随便找一本线性代数的正规参考书查一查。

现在我们需要知道这条法线与灯光的角度。向量数学的另一个好东西叫做内积（dot product），它与外积不同。我们有了法线的向量和灯光的向量。下面计算点积：

```
var dotProd:Number = norm.x * light.x + norm.y * light.y + norm.z * light.z;
```

我们看到，内积要比外积简单一些！

OK，都差不多了！接下来，计算法线的量值，以及灯光的量值，大家应该还认识这个 3D 版的勾股定理吧：

```
var normMag:Number = Math.sqrt(norm.x * norm.x + norm.y * norm.y + norm.z * norm.z);
var lightMag:Number = Math.sqrt(light.x * light.x + light.y * light.y + light.z * light.z);
```

请注意，当一个三角形被渲染时，变量 lightMag 每次都要进行计算，这样就允许灯光是移动的。如果知道光源是固定的，我们可以在代码的一开始就加入这个变量，只需在创建灯光或为三角形赋值时进行一次计算。或者可以为 Light 类添加 lightMag 属性，让它可以在每次 x, y, z 属性发生变化时被计算。看，我已经给大家留出了各种发挥的空间！

最后，将前面计算出的这些数放入一个具有魔力公式中：

return (Math.acos(dotProd / (normMag * lightMag)) / Math.PI) * light.brightness;

其中 dotProd 是一个分量，而 normMag * lightMag 是另一个分量。两者相除得出一个比率。回忆一下第三章，一个角度的余弦给了我们一个比率，而一个比率的反余弦给了我们一个角度。这就是灯光照射在多边形表面上的角度。它的范围在 0 到 Math.PI 个弧度之间（0 到 180 度），也就是说灯光完全照射在物体前面上或完全照射在物体背面。

用这个数除以 Math.PI 得出一个百分数，再用它乘以 brightness 的百分比就得出了最终用于改变底色的滤光系数。

OK，所有这些仅仅给出了多边形表面颜色！此刻，在现有的代码中实现它就非常简单了。我们在 draw 方法中使用它。应该像这样直接使用这个调整后的颜色：

g.beginFill(getAdjustedColor());

为了把上述内容综合起来，以下是全部最终的 Triangle.as 和 ExtrudedA.as 代码，列出了我们本章所有发生变化的部分：

首先是 Triangle：

```
package {
 import flash.display.Graphics;
 public class Triangle {
  private var pointA:Point3D;
  private var pointB:Point3D;
  private var pointC:Point3D;
  private var color:uint;
  public var light:Light;
  public function Triangle(a:Point3D, b:Point3D,
  c:Point3D, color:uint) {
   pointA = a;
   pointB = b;
   pointC = c;
   this.color = color;
  }
  public function draw(g:Graphics):void {
   if (isBackFace()) {
    return;
   }
   g.beginFill(getAdjustedColor());
   g.moveTo(pointA.screenX, pointA.screenY);
   g.lineTo(pointB.screenX, pointB.screenY);
   g.lineTo(pointC.screenX, pointC.screenY);
   g.lineTo(pointA.screenX, pointA.screenY);
   g.endFill();
  }
  private function getAdjustedColor():uint {
   var red:Number = color >> 16;
   var green:Number = color >> 8 & 0xff;
   var blue:Number =color & 0xff;
   var lightFactor:Number = getLightFactor();
   red *= lightFactor;
   green *= lightFactor;
   blue *= lightFactor;
   return red << 16 | green << 8 | blue;
  }
```

```
    private function getLightFactor():Number {
      var ab:Object = new Object();
      ab.x = pointA.x - pointB.x;
      ab.y = pointA.y - pointB.y;
      ab.z = pointA.z - pointB.z;
      var bc:Object = new Object();
      bc.x = pointB.x - pointC.x;
      bc.y = pointB.y - pointC.y;
      bc.z = pointB.z - pointC.z;
      var norm:Object = new Object();
      norm.x = (ab.y * bc.z) - (ab.z * bc.y);
      norm.y = -((ab.x * bc.z) - (ab.z * bc.x));
      norm.z = (ab.x * bc.y) - (ab.y * bc.x);
      var dotProd:Number = norm.x * light.x +
      norm.y * light.y +
      norm.z * light.z;
      var normMag:Number = Math.sqrt(norm.x * norm.x +
      norm.y * norm.y +
      norm.z * norm.z);
      var lightMag:Number = Math.sqrt(light.x * light.x +
      light.y * light.y +
      light.z * light.z);
      return Math.acos(dotProd / normMag * lightMag) / Math.PI * light.brightness;
    }
    private function isBackFace():Boolean {
      // 见 http://www.jurjans.lv/flash/shape.html
      var cax:Number = pointC.screenX - pointA.screenX;
      var cay:Number = pointC.screenY - pointA.screenY;
      var bcx:Number = pointB.screenX - pointC.screenX;
      var bcy:Number = pointB.screenY - pointC.screenY;
      return cax * bcy > cay * bcx;
    }
    public function get depth():Number {
      var zpos:Number = Math.min(pointA.z, pointB.z);
      zpos = Math.min(zpos, pointC.z);
      return zpos;
    }
  }
}
```

然后是 ExtrudedA：

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  public class ExtrudedA extends Sprite {
    private var points:Array;
    private var triangles:Array;
    private var fl:Number = 250;
    private var vpX:Number = stage.stageWidth / 2;
    private var vpY:Number = stage.stageHeight / 2;
```

```
public function ExtrudedA() {
  init();
}
private function init():void {
  points = new Array();
  points[0] = new Point3D( -50, -250, -50);
  points[1] = new Point3D( 50, -250, -50);
  points[2] = new Point3D( 200, 250, -50);
  points[3] = new Point3D( 100, 250, -50);
  points[4] = new Point3D( 50, 100, -50);
  points[5] = new Point3D( -50, 100, -50);
  points[6] = new Point3D(-100, 250, -50);
  points[7] = new Point3D(-200, 250, -50);
  points[8] = new Point3D( 0, -150, -50);
  points[9] = new Point3D( 50, 0, -50);
  points[10] = new Point3D( -50, 0, -50);
  points[11] = new Point3D( -50, -250, 50);
  points[12] = new Point3D( 50, -250, 50);
  points[13] = new Point3D( 200, 250, 50);
  points[14] = new Point3D( 100, 250, 50);
  points[15] = new Point3D( 50, 100, 50);
  points[16] = new Point3D( -50, 100, 50);
  points[17] = new Point3D(-100, 250, 50);
  points[18] = new Point3D(-200, 250, 50);
  points[19] = new Point3D( 0, -150, 50);
  points[20] = new Point3D( 50, 0, 50);
  points[21] = new Point3D( -50, 0, 50);
  for (var i:uint = 0; i < points.length; i++) {
    points[i].setVanishingPoint(vpX, vpY);
    points[i].setCenter(0, 0, 200);
  }
  triangles = new Array();
  triangles[0] =new Triangle(points[0], points[1],
  points[8], 0xcccccc);
  triangles[1] =new Triangle(points[1], points[9],
  points[8], 0xcccccc);
  triangles[2] =new Triangle(points[1], points[2],
  points[9], 0xcccccc);
  triangles[3] =new Triangle(points[2], points[4],
  points[9], 0xcccccc);
  triangles[4] =new Triangle(points[2], points[3],
  points[4], 0xcccccc);
  triangles[5] =new Triangle(points[4], points[5],
  points[9], 0xcccccc);
  triangles[6] =new Triangle(points[9], points[5],
  points[10], 0xcccccc);
  triangles[7] =new Triangle(points[5], points[6],
  points[7], 0xcccccc);
  triangles[8] =new Triangle(points[5], points[7],
```

```
points[10], 0xcccccc);
triangles[9] =new Triangle(points[0], points[10],
points[7], 0xcccccc);
triangles[10] =new Triangle(points[0], points[8],
points[10], 0xcccccc);
triangles[11] =new Triangle(points[11], points[19],
points[12], 0xcccccc);
triangles[12] =new Triangle(points[12], points[19],
points[20], 0xcccccc);
triangles[13] =new Triangle(points[12], points[20],
points[13], 0xcccccc);
triangles[14] =new Triangle(points[13], points[20],
points[15], 0xcccccc);
triangles[15] =new Triangle(points[13], points[15],
points[14], 0xcccccc);
triangles[16] =new Triangle(points[15], points[20],
points[16], 0xcccccc);
triangles[17] =new Triangle(points[20], points[21],
points[16], 0xcccccc);
triangles[18] =new Triangle(points[16], points[18],
points[17], 0xcccccc);
triangles[19] =new Triangle(points[16], points[21],
points[18], 0xcccccc);
triangles[20] =new Triangle(points[11], points[18],
points[21], 0xcccccc);
triangles[21] =new Triangle(points[11], points[21],
points[19], 0xcccccc);
triangles[22] =new Triangle(points[0], points[11],
points[1], 0xcccccc);
triangles[23] =new Triangle(points[11], points[12],
points[1], 0xcccccc);
triangles[24] =new Triangle(points[1], points[12],
points[2], 0xcccccc);
triangles[25] =new Triangle(points[12], points[13],
points[2], 0xcccccc);
triangles[26] =new Triangle(points[3], points[2],
points[14], 0xcccccc);
triangles[27] =new Triangle(points[2], points[13],
points[14], 0xcccccc);
triangles[28] =new Triangle(points[4], points[3],
points[15], 0xcccccc);
triangles[29] =new Triangle(points[3], points[14],
points[15], 0xcccccc);
triangles[30] =new Triangle(points[5], points[4],
points[16], 0xcccccc);
triangles[31] =new Triangle(points[4], points[15],
points[16], 0xcccccc);
triangles[32] =new Triangle(points[6], points[5],
points[17], 0xcccccc);
```

```
      triangles[33] =new Triangle(points[5], points[16],
      points[17], 0xcccccc);
      triangles[34] =new Triangle(points[7], points[6],
      points[18], 0xcccccc);
      triangles[35] =new Triangle(points[6], points[17],
      points[18], 0xcccccc);
      triangles[36] =new Triangle(points[0], points[7],
      points[11], 0xcccccc);
      triangles[37] =new Triangle(points[7], points[18],
      points[11], 0xcccccc);
      triangles[38] =new Triangle(points[8], points[9],
      points[19], 0xcccccc);
      triangles[39] =new Triangle(points[9], points[20],
      points[19], 0xcccccc);
      triangles[40] =new Triangle(points[9], points[10],
      points[20], 0xcccccc);
      triangles[41] =new Triangle(points[10], points[21],
      points[20], 0xcccccc);
      triangles[42] =new Triangle(points[10], points[8],
      points[21], 0xcccccc);
      triangles[43] =new Triangle(points[8], points[19],
      points[21], 0xcccccc);
      var light:Light = new Light();
      for (i = 0; i < triangles.length; i++) {
        triangles[i].light = light;
      }
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onEnterFrame(event:Event):void {
      var angleX:Number = (mouseY - vpY) * .001;
      var angleY:Number = (mouseX - vpX) * .001;
      for (var i:uint = 0; i < points.length; i++) {
        var point:Point3D = points[i];
        point.rotateX(angleX);
        point.rotateY(angleY);
      }
      triangles.sortOn("depth", Array.DESCENDING | Array.NUMERIC);
      graphics.clear();
      for (i = 0; i < triangles.length; i++) {
        triangles[i].draw(graphics);
      }
    }
  }
}
```

我们看到，在文档类中只有两个次要的变化。主要的工作都集中在 Triangle 中。同时，我还让所有的三角形使用相同的颜色，我认为这样做可以更好地观察灯光效果（见图 17-6）。

图 17-6 带有背面剔除，深度排序及 3D 灯光的三维立体模型

17.4 小结

哇！几页纸却说了这么多的内容！但我认为结果是不可思议的。现在你已经有工具可以制作极为漂亮的 3D 影片了。毫无疑问，你可以加入你想要的各种变化。例如，在最后一个例子中，灯光是静止的而物体是移动的。可以试着移动灯光。（这只是要改变它的 x、y 和/或 z 的位置。）

这就是关于 3D 讨论的总结。在下一章中，你会看到矩阵数学，他是你经常用来改变物体的缩放和旋转的另一种方法，因此你在 3D 程序中也会经常看到。

第十八章　矩阵数学

本章我们不去介绍一些新的运动、物理学或渲染图形的方法。我要给大家介绍的是矩阵（Matrix），它给我们提供了一个新的可选方案。

矩阵在 3D 系统中 3D 点的旋转，缩放以及平移（运动）中使用得非常频繁。在各种 2D 图形的变换上也很常用。您也许可以回想到 beginGradientFill 方法就是使用矩阵来设置位置，大小以及旋转比例的。

本章大家将看到如何创建一个 3D 矩阵系统，用以操作 3D 的影片并且可以看到一些 Flash 中内置的矩阵。我很庆幸现在为止还没有一处提到 Keanu Reeves [译注：基努-里维斯，尤指电影《黑客帝国》-- The Matrix] 的电影。看看我还能坚持多久。

矩阵基础

矩阵最简单的定义是一个数字表格。它可以有一个或多个水平的行和一个或多个垂直的列。图 18-1 展示了一些矩阵。

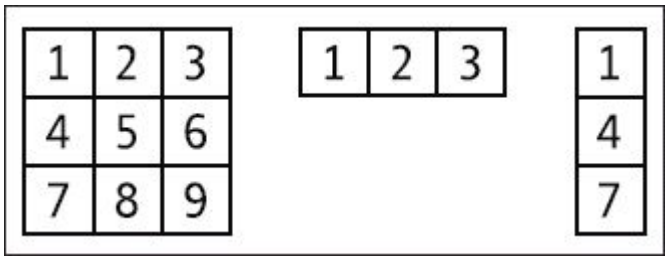


图 18-1 一个 3×3 矩阵，一个 1×3 矩阵，一个 3×1 矩阵

矩阵通常都是由一些变量来描述的，如 M。在矩阵中为表示一个特殊的单元，我们使用的变量里面通常要用行列的值作为脚标。例如，如果图 18-1 中的 3×3 矩阵叫做 M，那么 M2,3 就等于 6，因为它指向第二行，第三列。

一个矩阵的单元不仅可以包含简单的数字，也可以是公式和变量。其实电子表格就是一个大的矩阵。我们可以用一个单元保存某一列的和，用另一个单元格将这个总和乘以一个分数，等等。我们看到这样的矩阵应该非常有用。

矩阵运算

一个电子表格就像一个自由组合的矩阵，我们要处理的矩阵更加有结构，至于能用它们做什么以及如何生成都有各自的规则。

我所见过的大多数矩阵数学的教材都只介绍两种方法的一种。首先学校讲的是矩阵运算的细节，使用的整个矩阵的几乎都是一些随机的数字。我们学习这些规则，但是不知道为什么要做这些事情或所得的结果代表什么。就像在玩把数字排列成漂亮形状的游戏。

第二个方法是详细地描述矩阵的内容但是略过手工操作，如“将两个矩阵相乘得到这个结果… …”让读者不知道乘法到底是怎么算的。

为了保证大家都能了解矩阵是如何工作的，我选择一个两者兼具的方法（折衷），从介绍一些数值矩阵开始，然后描述如何做矩阵乘法。

矩阵加法

矩阵更为通常的作用是操作 3D 点。一个 3D 点包涵了 x, y, z 坐标。我们可以简单地将它视为一个 1×3 的矩阵：

x　y　z

现在假设要将这个点在空间中移动，或叫做点的平移。我们需要知道每个轴上移动多远。这时可以将它放入一个转换矩阵（translation matrix）中。它又是一个 1×3 的矩阵：

dx dy dz

这里 dx, dy, dz 是每个轴移动的距离。现在我们要想办法将转换矩阵加到点矩阵上面。这就是矩阵加法，非常简单。我们只需要将相应的单元进行相加形成一个新的包含了每个单元之和的矩阵。很明显，要让两个矩阵相加，它们的大小都应该是相同的。转换方法如下：

x y z + dx dy dz = (x + dx) (y + dy) (z + dz)

获得的矩阵可以叫做 x1, y1, z1，转换之后包含了该点的新坐标。让我们用实数来试一下。假设点在 x, y, z 轴上的位置分别为 100, 50, 75，要让它们分别移动 -10, 20, -35。则应该是这样的：

100 50 75 + -10 20 -35 = (100 - 10) (50 + 20) (75 - 35)

因此，当进行加法运算时，所得该点的新坐标就是 90, 70, 40。非常简单，不是吗？大家也许已经注意到了速度间的相互关系，每个轴上的速度都加到了另一个矩阵的相应位置上。公平交易嘛。

如果我们有一个较大的矩阵，那么继续使用同样的方法，匹配每个单元。我们不会去处理大于 3×1 的矩阵加法，但是我会给大家这样一个抽象的例子：

a b c j k l (a + j) (b + k) (c + l)
d e f + m n o = (d + m) (e + n) (f + o)
g h i p q r (g + p) (h + q) (i + r)

以上就是我们需要知道矩阵加法的一切。在介绍了矩阵乘法之后，我将展示如何将现有的函数使用在矩阵 3D 引擎中。

矩阵乘法

在 3D 转换中应用更为广泛的是矩阵乘法（matrix multiplication），常用于缩放与旋转。在本书中我们实际上不会用到 3D 缩放，因为例子中的点缩放，影片也没有 3D 的“厚度”，因此只有二维的缩放。当然，大家可以建立一个可缩放整个 3D 立体模型的更为复杂的引擎。这就需要写一些根据新的影片大小改变 3D 点的函数。这些已经超出了我们讨论的范围，但是由于缩放是非常简单的，并且使用矩阵乘法很容易实现，因此我将带大家看一下这个例子。

使用矩阵进行缩放

首先，需要知道一个物体现有的宽度，高度和深度 —— 换句话讲，它是三个轴上每个轴分量的大小。当然可以建立一个 3×1 的矩阵：

w h d

我们知道 w, h, d 代表宽度（width），高度（height）和深度（depth）。下面需要缩放这个矩阵：

sx 0 0
0 sy 0
0 0 sz

这里 sx, sy, sz 是对应轴上的缩放比例。它们都将是分数或小数，1.0 为 100%，2.0 为 200%，0.5 为 50%，等等。稍后大家会看到为什么矩阵是用这种形式分布的。

要知道，矩阵乘法是为了让两个矩阵相乘，第一个矩阵的列数必需与另一个矩阵的行数相同。只要符合这个标准，第一个矩阵可以有任意多个行，第二个矩阵可以有任意多个列。本例中，由于第一个矩阵有三列（w, h, d），因此缩放矩阵就有三行。那么它们如何进行乘法运算呢？让我们来看一下这个模式：

```
      sx   0   0
  w   h   d *   0   sy   0
      0   0   sz
```

矩阵的计算结果如下：

(w*sx + h*0 + d*0) (w*0 + h*sy + d*0) (w*0 + h*0 + d*sz)

删除所有等于 0 的数：

(w*sx) (h*sy) (d*sz)

非常有合乎逻辑，因为我们是用宽度（x 轴分量）乘以 x 缩放系数，高度乘以 y 缩放系数，深度乘以 z 缩放系数。但是，我们究竟在做什么呢？那些所有等于 0 的数都像被遮盖上了，因此让我们将这个模式抽象得更清晰一点。

```
     a   b   c
 u   v   w   *   d   e   f
     g   h   i
```

现在可以看到该模式的结果为：

(u*a + v*d + w*g) (u*b + v*e + w*h) (u*c + v*f + w*i)

我们将第一个矩阵的第一行（u, v, w）与第二个矩阵每行的第一个元素相乘。将它们加起来就得到了结果的第一行的第一个元素。在第二个矩阵的第二列（b, e, h）中使用相同的方法就得到了第二列的结果。

如果第一个矩阵的行数大于 1，就要在第二行中重复上述动作，就会得到第二行的结果：

```
 u   v   w       a   b   c
 x   y   z   *   d   e   f
     g   h   i
```

就得到了这个 3×2 的矩阵：

(u*a + v*d + w*g) (u*b + v*e + w*h) (u*c + v*f + w*i)

(x*a + y*d + z*g) (x*b + y*e + z*h) (x*c + y*f + z*i)

现在让我们看一些实际中用到的矩阵乘法 —— 坐标旋转。希望通过这个缩放的例子会让它看起来更加清晰。

使用矩阵进行坐标旋转

首先，要挖出我们的 3D 点矩阵：

```
x   y   z
```

它保存了该点所有的坐标。当然，还要有一个旋转矩阵。我们可以在三个轴的任意一轴上进行旋转。我们将分别创建每种旋转的矩阵。先从 x 轴旋转矩阵开始：

```
1    0      0
0   cos    sin
0   -sin   cos
```

这里有一些正余弦值，“sin 和 cos 是什么？”很明显，这就是我们要旋转的角度的正余弦

值。如果让这个点旋转 45 度，则这两个值就是 45 的正弦和余弦值。（当然，在代码中要使用弧度制）现在，我们让该矩阵与一个 3D 点的矩阵相乘，看一下结果。

```
      1     0      0
x   y   z   *   0   cos   sin
      0   -sin cos
```

由此得到：

```
(x*1 + y*0 + z*0) (x*0 + y*cos - z*sin) (x*0 + y*sin + z*cos)
```

整理后结果如下：

```
(x) (y*cos - z*sin) (z*cos + y*sin)
```

这句话用 ActionScript 大略可以翻译成：

```
x = x;
y = Math.cos(angle) * y - Math.sin(angle) * z;
z = Math.cos(angle) * z + Math.sin(angle) * y;
```

回忆一下第十章，在讨论坐标旋转时，我们会看到这实际上就是 x 轴的坐标旋转。不要惊讶，矩阵数学只是观察和组织各种公式和方程的不同方法。至此，要创建一个 y 轴旋转的矩阵就非常容易了：

```
 cos   0   sin
  0    1    0
-sin   0   cos
```

最后，z 轴的旋转为：

```
 cos   sin    0
-sin   cos    0
  0     0     1
```

这是一个很好的尝试，用 x, y, z 的矩阵乘以每个旋转矩阵的单位，证明所得到的结果与第十章的坐标旋转公式完全相同。

编写矩阵

OK，现在大家已经有了足够的基础将这些知识转换为代码了。下面，我们对第十五章的 RotateXY.as 进行重新转换。这个类中有 rotateX 和 rotateY 两个方法，用以实现 3D 坐标旋转。我们要让它们以矩阵的方式工作。

从 rotateX 函数开始。它会用到小球的 x, y, z 坐标，将它们放入 1×3 矩阵，然后创建一个给定角度的 x 旋转矩阵。这个矩阵将使用数组的形式表示。最后使用 matrixMultiply 函数让两个矩阵相乘，当然还需要创建这个函数！相乘后的矩阵还要用另一个数组进行保存，因为我们需要将这些数值再存回小球的 x, y, z 坐标中。下面是新版的方法：

```
private function rotateX(ball:Ball3D, angleX:Number):void {
  var position:Array = [ball.xpos, ball.ypos, ball.zpos];
  var sin:Number = Math.sin(angleX);
  var cos:Number = Math.cos(angleX);
  var xRotMatrix:Array = new Array();
  xRotMatrix[0] = [1, 0, 0];
  xRotMatrix[1] = [0, cos, sin];
  xRotMatrix[2] = [0, -sin, cos];
  var result:Array = matrixMultiply(position, xRotMatrix);
  ball.xpos = result[0];
  ball.ypos = result[1];
```

```
  ball.zpos = result[2];
}
```

下面是矩阵乘法的函数：

```
private function matrixMultiply(matrixA:Array, matrixB:Array):Array {
  var result:Array = new Array();
  result[0] = matrixA[0] * matrixB[0][0] +
  matrixA[1] * matrixB[1][0] +
  matrixA[2] * matrixB[2][0];
  result[1] = matrixA[0] * matrixB[0][1] +
  matrixA[1] * matrixB[1][1] +
  matrixA[2] * matrixB[2][1];
  result[2] = matrixA[0] * matrixB[0][2] +
  matrixA[1] * matrixB[1][2] +
  matrixA[2] * matrixB[2][2];
  return result;
}
```

现在，这个矩阵乘法的函数是手工写出的一个 1×3 和 3×3 矩阵的乘法，这就是我们后面用在每个例子中的函数。大家也可以使用 for 循环创建出更为动态的可处理任何大小的矩阵函数，但是现在我要让代码保持简洁。

最后创建 rotateY 函数。如果你了解 rotateX 函数，那么这个函数应该非常显而易见了。只需要创建一个 y 旋转矩阵来代替 x 旋转矩阵即可。

```
private function rotateY(ball:Ball3D, angleY:Number):void {
  var position:Array = [ball.xpos, ball.ypos, ball.zpos];
  var sin:Number = Math.sin(angleY);
  var cos:Number = Math.cos(angleY);
  var yRotMatrix:Array = new Array();
  yRotMatrix[0] = [ cos, 0, sin];
  yRotMatrix[1] = [ 0, 1, 0];
  yRotMatrix[2] = [-sin, 0, cos];
  var result:Array = matrixMultiply(position, yRotMatrix);
  ball.xpos = result[0];
  ball.ypos = result[1];
  ball.zpos = result[2];
}
```

就是这样。大家也可以创建一个 rotateZ 函数，由于我们的例子中实际上不需要用到它，所以我将它作为练习留给大家完成。

现在，运行一下 RotateXY.as，与第十五章的版本相比，它们看上去实际是一样的。在 AS 2 中，我发现非矩阵版本的运行得更为流畅一些。原因是我们为 3D 旋转和缩放执行了非常大量的数学运算。当我们使用矩阵数学进行计算时，会产生额外的计算。在进行矩阵乘法时，我们实际是做了四次乘以零的操作，并将这四个结果与其它数值相加。这八次数学运算实际上没有任何作用。将这些操作乘以 50 个对象，每帧旋转二次，每帧就多做了 800 次额外计算！这两个版本在 AS 3 中的运行时看不出任何的不同，这就是 Flash CS3 与 AS 3 强大的证明。但是，当加入的物体越来越多时，我们就要为这些巨大的计算量付出代价。我给大家的这些代码都非常基本的。你也许可以使它更加优化一些，让性能得到提升。

即使在 3D 中不使用矩阵，我们仍可以发现它们在其它方面的用途，我将在下面一节进行介绍。在 3D 中使用矩阵是一个很好的引子，因为这样可以让大家看到它们是如何与已知公式相关联的。同样，矩阵在其它语言的 3D 制作中应用得非常之广泛，而且比我们现在的 ActionScript 更为有效。在这些语言中，只需付出一点点 CPU 就可以得到矩阵所带来的组织良好的代码。如果大家试图在 Flash 以外的其它软件中进行 3D 动画编程，那么就

一定要使用到矩阵。还是那句话，谁知道 Flash 播放器几年后会成为什么样？终会有一天，所有的这些技术都能与 Flash 完美地结合。

Matrix 类

刚刚提到，学习矩阵的一个很好的理由是它被用在许多 ActionScript 类的内核中。事实上，我们有一个内置矩阵类。浏览一下 Flash 帮助文档中的 flash.geom.Matrix 类，就会发现那里写得非常清楚详细。如果本章前面内容您都能理解，那么要掌握这些材料就一定没问题。文档写得非常好，我就不再浪费空间将这些内容重复一遍了，但是我会给大家一个快速的总结并举出两个例子。

矩阵主要用于对显示对象的转换（旋转，缩放和平移）。现在，任何一个显示对象（Sprite，影片剪辑，文本类等）都有名为 transform （转换）的属性。这是 flash.geom.Transform 类的一个实例，它还包含有另一个名为 matrix 的属性。如果我们创建一个 Matrix 类的实例，并把它赋给显示对象的 transform.matrix 属性，那么它将会改变这个对象的形状、大小或位置。我们马上会看到一些具体的例子。

基本来说 Matrix 类的矩阵是一个 3×3 的矩阵，形式如下：

```
a  b  tx
c  d  ty
u  v  w
```

其中 u, v, w 内部自动被设置为 0, 0, 1。而且它们是不可改变的，因此不需要管它们。（更为具体的解释请参见帮助文档）我们使用下述语法来创建一个新的 Matrix：

```
import flash.geom.Matrix;
var myMatrix:Matrix = new Matrix(a, b, c, d, tx, ty);
```

那么这些字母是什么意思呢？tx 和 ty 非常简单。它们通过改变矩阵来控制显示对象的 x 和 y 轴。而 a, b, c, d 有些难度，因为它们都相互依赖。如果设置 b 和 c 为 0，就可以使用 a 和 d，在 x 和 y 轴上缩放一个对象。如果设置 a 和 d 为 1，就可以使用 b 和 c，分别在 y 和 x 轴上倾斜一个对象。最后，可以用一种我们非常熟悉的方式来使用 a, b, c, d。在本例中，设置如下：

```
 cos   sin   tx
-sin   cos   ty
  u     v    w
```

当然，我们可以看到这里包含了一个旋转矩阵，它确实可以旋转一个物体。自然本例中的 cos 和 sin 代表我们想要旋转的某个角度的正弦和余弦值（弧度制）。让我们试验一下这个例子。

这里可见 MatrixRotate.as，这个类中用红色正方形创建了一个简单影片。然后设置一个 enterFrame 处理函数，所有的动作都加在其中：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.geom.Matrix;
 public class MatrixRotate extends Sprite {
  private var angle:Number = 0;
  private var box:Sprite;
  public function MatrixRotate() {
   init();
  }
  private function init():void {
   box = new Sprite();
   box.graphics.beginFill(0xff0000);
```

```
    box.graphics.drawRect(-50, -50, 100, 100);
    box.graphics.endFill();
    addChild(box);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
   }
   private function onEnterFrame(event:Event):void {
    angle += .05;
    var cos:Number = Math.cos(angle);
    var sin:Number = Math.sin(angle);
    box.transform.matrix = new Matrix(cos, sin,
    -sin, cos,
    stage.stageWidth / 2,
    stage.stageHeight / 2);
   }
  }
}
```

这里有一个 angle 变量，每帧都会增加。代码求出了角度的正弦和余弦值并将它们赋给新的矩阵对象，以这种方式指定 rotation。我同时还设置了平移，根据舞台的宽度和高度把影片放置到中心。新的矩阵被赋给了影片的 transform.matrix 属性。测试该影片就得到了一个旋转的正方形。

现在，也许有人会问，你什么不改变影片的 rotation 属性。在一个简单的例子中使用 rotation 没有问题，这是一个更为简单的解决方法。但是，也许在一些处理多个角度、弧度、正弦、余弦的例子中，相比将一切转换回角度制并改变 rotation 值而言，像这样的矩阵赋值确实要简单很多。

再给大家一些实际的说明，让我们试一下倾斜。倾斜（Skewing）意思是将物体在一个轴上进行拉伸以便使一个部分走一条路，另一个部分走另一条路。斜体字就是一个倾斜的例子。字母顶部的部分向右倾斜，而底部的部分向左倾斜。这是 Flash 中一个众所周知的一个难点，但是使用 Matrix 类将会惊人地简单。如同我前面所说，设置矩阵的 a 和 d 为 1。属性 b 是 y 轴倾斜的值，属性 c 控制 x 轴的倾斜值。让我们先来试一下 x 倾斜。在 SkewX.as 中，我几乎使用了与前一个例子完全相同的设置，只不过改变了 onEnterFrame 方法中矩阵的创建。

```
private function onEnterFrame(event:Event):void {
 var skewX:Number = (mouseX - stage.stageWidth / 2) * .01;
 box.transform.matrix = new Matrix(1, 0,
 skewX, 1,
 stage.stageWidth / 2,
 stage.stageHeight / 2);
}
```

这里相对于鼠标的 x 坐标创建了一个 skewX 变量，以舞台的中心为偏移量。然后将它乘以 .01 让倾斜的值处于可控范围，并将此值赋给矩阵。

测试影片后，我们将看到如何让一个完整的影片进行倾斜，如图 18-2 所示。有了 Matrix 类一切都变得可能，如果你知道有谁在试图做这样的事，那么就把上述代码拿给他们看，等着看他们开始流口水吧！如果您亲自测试了这段代码，那么肯定已经知道我的意思了。

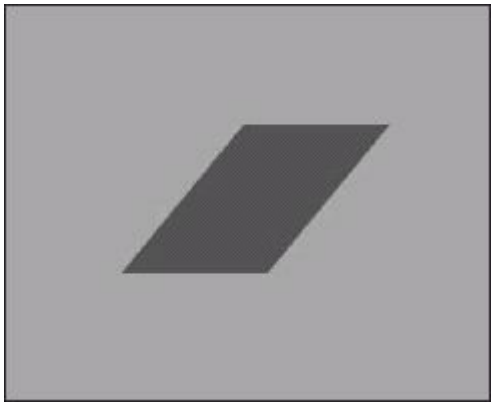


图 18-2 影片在 x 轴上的倾斜

在 SkewXY 中，我在 y 轴上做了同样的事情：

```
private function onEnterFrame(event:Event):void {
  var skewX:Number = (mouseX - stage.stageWidth / 2) * .01;
  var skewY:Number = (mouseY - stage.stageHeight / 2) * .01;
  box.transform.matrix = new Matrix(1, skewY,
  skewX, 1,
  stage.stageWidth / 2,
  stage.stageHeight / 2);
}
```

从图 18-3 中可以看到影片在两个轴上的倾斜

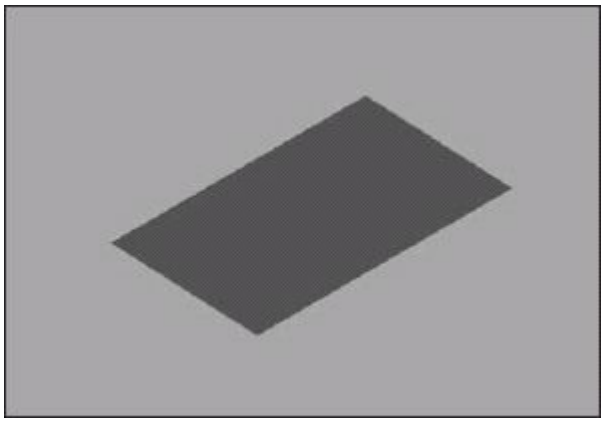


图 18-3 影片在两个轴上的倾斜

如此简单就能实现这样的效果的确让人惊喜。如果您不确定这种效果能用在哪里，那么我告诉您，倾斜效果在伪 3D 效果中使用得非常频繁。当我们在上个例子中移动鼠标时，如果这个图形正在倾斜并旋转，那么大家已经可以看到了它是如何显示出透视来的。这不是特别精确的 3D，但是它可以用在一些非常棒的效果中。在网上有一些这方面的教程，告诉我们如何使用倾斜实现这种伪 3D 效果。Matrix 的使用也许会将代码缩短一半。

大家一定要为 Matrix 类查一查帮助文档，因为这里还有其它好多好东西。要知道上述这些并不是 AS 3 唯一使用矩阵的地方。大家还应该看看 ColorMatrixFilter，ConvolutionFilter，这些不同的制图 API 填充和渐变的方法，以及 flash.geom.Transform 类。所以说矩阵的应用非常广泛！

18.4 小结

我们已经讲解了矩阵是什么、如何应用它们和合并他们等基础内容，并使用它们创建了一些非常酷的效果。现在在你的头脑中已经有了这些概念，你要准备应用矩阵提供的强大的功能了，希望你在 AS 3 的内置方法中遇到它们时，不要害怕使用它们，因为你确实能够使用它们了。

第十九章　实用技巧

现在您已经来到了最后一章。我将所有想要介绍的一些小东西都放在了这一章，它们不太合适放在其它地方，或者说与前面章节的主线有些脱离。

本章，我还重组了前面每章课后列出的公式，因此可以当作这些公式的一个参考点。

由于这些课题都是比较零碎的概念，所以我没有办法将这些许许多多的内容组织起来。因此每一节都是一个独立的单元。好了，不多说了，让我们开始吧。

布朗（随机）运动

先讲讲历史。一天，一个名叫罗伯特-布朗（Robert Brown）的植物学家正在观察一滴水中的花粉颗粒，随后他发现这些花粉是在随机运动的。虽然它们不是水流或水的运动，但是这些小小的颗粒却永远不会停下来。他发现同样的事情会发生在微尘中，但它们不会像花粉那样游泳。虽然他不知道为什么会有这种现象，其实不只是他还有其他所有人在几十年内都不能给出解释，但是他却将这种现象用自己的名字命名 —— 只是为了能意识到它！

当今，我们对布朗运动的解释是大量的水分子在一滴水中不断运动，虽然水滴看上去是静止的。这些水分子与花粉和灰尘发生碰撞，将一些动量传给它们。因为即使是一颗小小的灰尘都要比一个水分子重上一百万倍，所以一次碰撞不会带来多大的影响。但是当每秒有几百万次的碰撞时，那么这些动量就会累计起来。

现在，一些水分子也许撞到了灰尘的一边，而另一些则撞在了另一边。最终，它们会达到总的平均值。但是，随着时间的变化，受到更多撞击的一边就会产生波动，假设为左边，那么这个粒子就会向右运动一点。底部所受撞击越多，则粒子向上运动得就越多。最后所有的值趋于平均，最终的结果通常不会在任何一个方向产生太多的动量。这就是随机悬浮动作。

我们可以在 Flash 中轻松地模拟出这种效果。在每一帧中，计算一个随机数加在运动物体的 x 和 y 速度中。随机的数值应该即可以是正数也可以是负数，并且一般来说都非常小，比如范围从 -0.1 到 +0.1。形式如下：

```
vx += Math.random() * 0.2 - 0.1;
vy += Math.random() * 0.2 - 0.1;
```

用 0.2 乘以一个随机的小数，所得的值从 0.0 到 0.2。再减去 0.1 则值变为 -0.1 到 0.1。在这里加入一些摩擦力（friction）很重要，否则速度会增大，并产生不自然的加速。在 Brownian1.as 中，我创建了 50 个粒子并让它们以布朗运动的形式悬浮。粒子就是我们熟悉的 Ball 类的实例，让它们为黑色并缩小。以下是代码：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 public class Brownian1 extends Sprite {
  private var numDots:uint = 50;
  private var friction:Number = 0.95;
  private var dots:Array;
  public function Brownian1() {
   init();
  }
  private function init():void {
   dots = new Array();
   for (var i:uint = 0; i < numDots; i++) {
    var dot:Ball = new Ball(1, 0);
    dot.x = Math.random() * stage.stageWidth;
```

```
      dot.y = Math.random() * stage.stageHeight;
      dot.vx = 0;
      dot.vy = 0;
      addChild(dot);
      dots.push(dot);
    }
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
    for (var i:uint = 0; i < numDots; i++) {
      var dot:Ball = dots[i];
      dot.vx += Math.random() * 0.2 - 0.1;
      dot.vy += Math.random() * 0.2 - 0.1;
      dot.x += dot.vx;
      dot.y += dot.vy;
      dot.vx *= friction;
      dot.vy *= friction;
      if (dot.x > stage.stageWidth) {
        dot.x = 0;
      } else if (dot.x < 0) {
        dot.x = stage.stageWidth;
      }
      if (dot.y > stage.stageHeight) {
        dot.y = 0;
      } else if (dot.y < 0) {
        dot.y = stage.stageHeight;
      }
    }
  }
 }
}
```

这里大多部分内容都不是什么新知识，我已将有关的部分用加粗表示。
如图 19-1 所示，代码运行时屏幕上的显示。

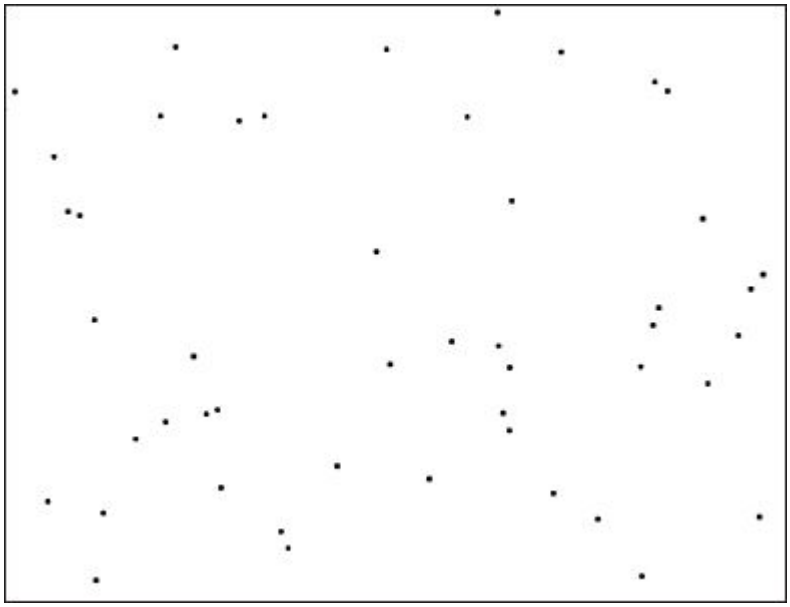


图 19-1 布朗运动

在 Brownian2.as 中，我将 numDots 减少到 20。然后将这行代码：

```
graphics.lineStyle(0, 0, .5);
```

加入到 init 函数中，并在 onEnterFrame 中加入一些绘图的代码。

```
private function onEnterFrame(event:Event):void {
 for (var i:uint = 0; i < numDots; i++) {
  var dot:Ball = dots[i];
  graphics.moveTo(dot.x, dot.y);
  dot.vx += Math.random() * 0.2 - 0.1;
  dot.vy += Math.random() * 0.2 - 0.1;
  dot.x += dot.vx;
  dot.y += dot.vy;
  dot.vx *= friction;
  dot.vy *= friction;
  graphics.lineTo(dot.x, dot.y);
 }
}
```

这样就在每个 dot 移动前与移动后的位置之间绘制了一条线。也就绘制出了自己的运动路径，如图 19-2 所示。如果您知道“布朗运动”这个词的话，那么一定会常常看到这种图像。

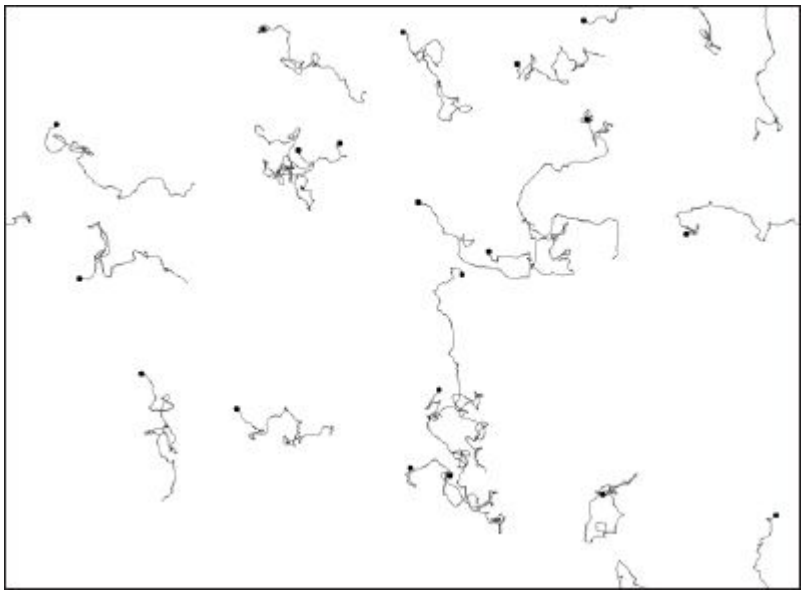


图 19-2 带有行迹的布朗运动

当我们要让物体以无意识、无外力的形式运动时，那么布朗运动将是最为实用的。也可以将它加入到一个带有其它运动的影片中，这样会给人一种随意（randomness）的感觉。例如到处乱飞的苍蝇或蜜蜂。也许这些影片已经有了它们各自的运动路径，但是在加入了一些随机运动后会让它看起来更加栩栩如生。

随机分布

有时候我们会创建一些物体并将它们随机放置。您已经在本书中看到了很多这样的例子，但是下面我们要来关注几种不同的方法，获得不同的结果。

方形分布

如果你想让物体随机分布在整个舞台上，那是相当简单的。只需要选择一个舞台宽度的随机数作为 x，一个高度的随机数作为 y。事实上，我们上一节就是这么做的：

```
for (var i:uint = 0; i < numDots; i++) {
  var dot:Ball = new Ball(1, 0);
  dot.x = Math.random() * stage.stageWidth;
  dot.y = Math.random() * stage.stageHeight;
  ...
}
```

但是如果说我们要让这些点集中分布在舞台中心，比如舞台中心上方 100 像素为顶，下方 100 像素为底。我们可以这样做，可见 Random1.as：

```
package {
  import flash.display.Sprite;
  public class Random1 extends Sprite {
    private var numDots:uint = 50;
    public function Random1() {
      init();
    }
    private function init():void {
      for (var i:uint = 0; i < numDots; i++) {
        var dot:Ball = new Ball(1, 0);
        dot.x = stage.stageWidth / 2 +
        Math.random() * 200 - 100;
        dot.y = stage.stageHeight / 2 +
        Math.random() * 200 - 100;
        addChild(dot);
      }
    }
  }
}
```

这里创建了一个从 -100 到 +100 的随机数，并将它加到舞台中心点上，因此所有的点在每个轴上都不能超过 100 像素。如图 19-3 所示。

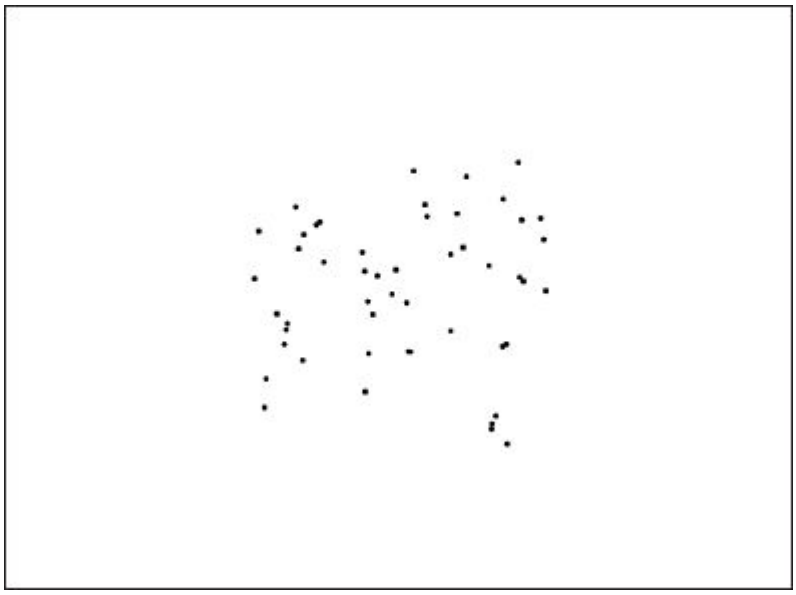


图 19-3 随机安排的点

不赖嘛。但是如果让它们挤在一起，增加点的数量（300）并减少随机范围为 50，我们将看到一些奇怪的事情。以下代码来自 Random2.as：

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

```
 public class Random2 extends Sprite {
  private var numDots:uint = 300;
  public function Random2() {
   init();
  }
  private function init():void {
   for (var i:uint = 0; i < numDots; i++) {
    var dot:Ball = new Ball(1, 0);
    dot.x = stage.stageWidth / 2 +
    Math.random() * 100 - 50;
    dot.y = stage.stageHeight / 2 +
    Math.random() * 100 - 50;
    addChild(dot);
   }
  }
 }
}
```

图 19-4 为运行结果。

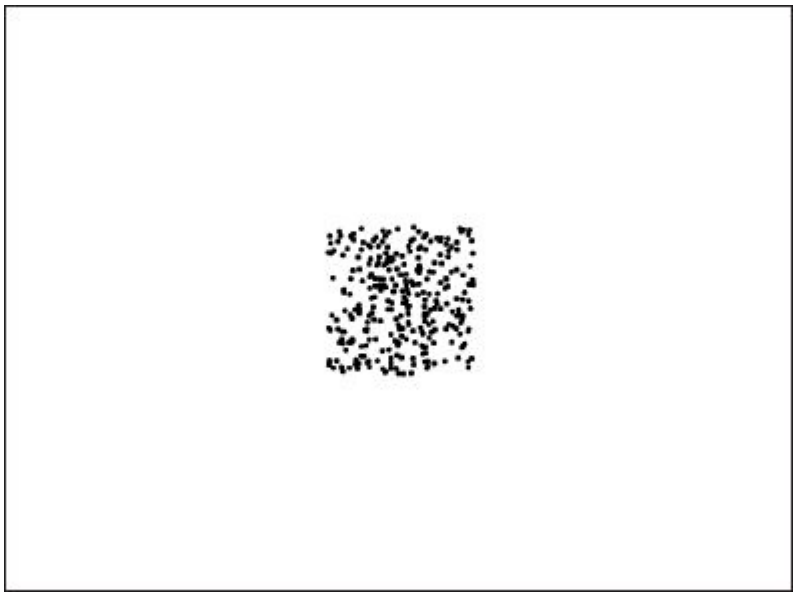


图 19-4 这种方法形成了一个方形。看起来不再像是随机安排的

如您所见，这些点形成了一个正方形。这样没问题，但是如果我们要制作一些爆炸效果或星系效果等，正方形看起来就不那么自然了。如果正方形分布不是您真正想要的，那么就继续下一项技术吧。

圆形分布

虽然圆形分布比方形分布稍稍复杂一点，但是它真的也不难。

首先，我们需要知道圆的半径。为了与上一个例子呼应，这里仍然使用 50。这将是从舞台中心开始能够放置的最大半径。我们将从 0 到 50 之间取一个随机数作为点的位置。然后从 0 到 PI * 2 个弧度 (360 度)选择一个随机的角度，再使用三角形函数找出点的 x 和 y 坐标。以下是 Random3.as 的代码：

```
package {
 import flash.display.Sprite;
 public class Random3 extends Sprite {
```

```
  private var numDots:uint = 300;
  private var maxRadius:Number = 50;
  public function Random3() {
   init();
  }
  private function init():void {
   for (var i:uint = 0; i < numDots; i++) {
    var dot:Ball = new Ball(1, 0);
    var radius:Number = Math.random() * maxRadius;
    var angle:Number = Math.random() * (Math.PI * 2);
    dot.x = stage.stageWidth / 2 +
    Math.cos(angle) * radius;
    dot.y = stage.stageHeight / 2 +
    Math.sin(angle) * radius;
    addChild(dot);
   }
  }
 }
}
```

运行结果如图 19-5 所示。

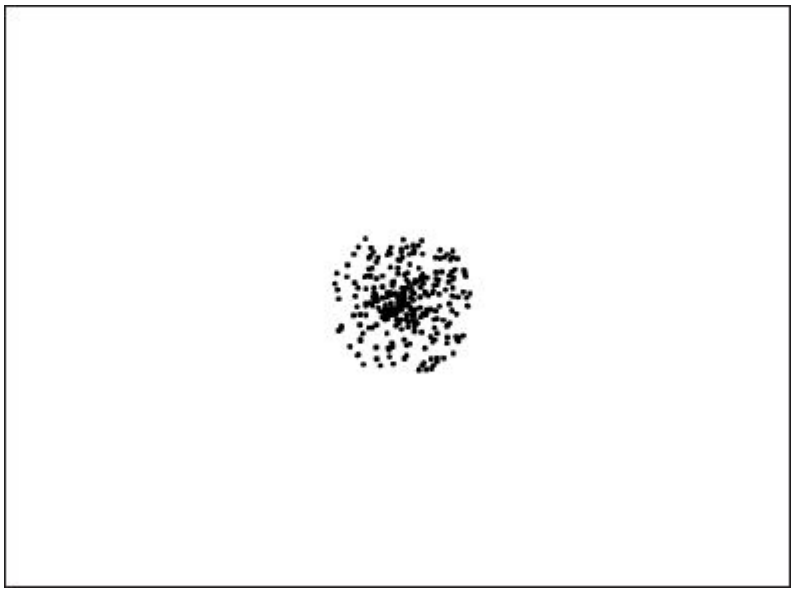


图 19-5 圆形随机分布

这就是我前面提到的那种更加自然的分布形式。大家也许注意到了这些点似乎更集中于圆形的中心。因为点是沿着半径平均的分布的，这就会使中心的点与边上的点同样多。但是因为中心的空间小，也就会显得更加拥挤。

这样的效果也许对于某些程序来说很好，但是 O’Shell (www.pixelwit.com) 的 Sean 曾给我一次挑战，让我给出一种方法将这些点均匀地分布成圆形。不得不承认我被难倒了，我的解决方法非常复杂。最后他给了我一个非常简单的办法，可见 Random4.as：

```
package {
 import flash.display.Sprite;
 public class Random4 extends Sprite {
  private var numDots:uint = 300;
  private var maxRadius:Number = 50;
  public function Random4() {
   init();
  }
```

```
    private function init():void {
      for (var i:uint = 0; i < numDots; i++) {
        var dot:Ball = new Ball(1, 0);
        var radius:Number = Math.sqrt(Math.random()) * maxRadius;
        var angle:Number = Math.random() *
        (Math.PI * 2);
        dot.x = stage.stageWidth / 2 +
        Math.cos(angle) * radius;
        dot.y = stage.stageHeight / 2 +
        Math.sin(angle) * radius;
        addChild(dot);
      }
    }
  }
}
```

通过取随机数的平方根，所得的结果偏向 1 而远离 0，这样做足以使分布变均匀。运行结果如图 19-6 所示。Sean，好人呀！

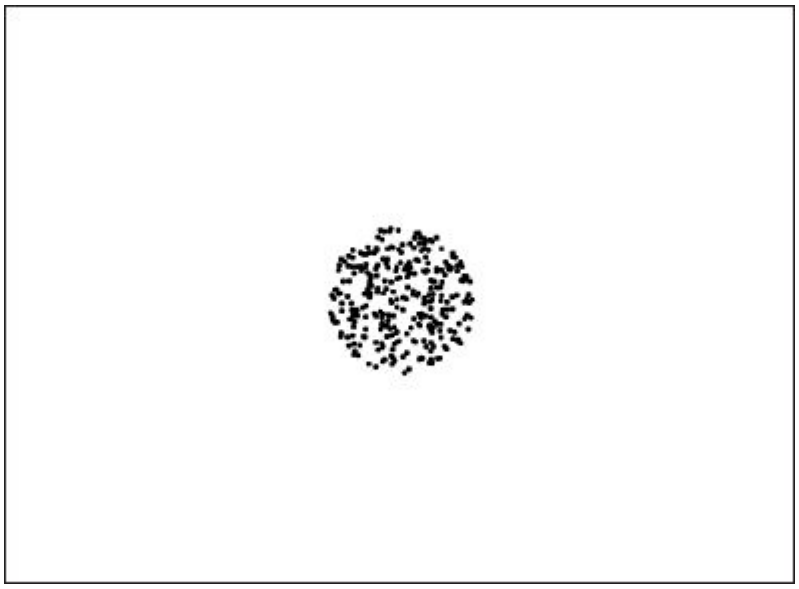


图 19-6 更为均衡的分布

偏向分布

最后，我们也许还想让物体自由地分布在整个舞台上，但是要让它们趋于在中心区域分布。我们已经找到了分布在边缘的方法，而现在要让它们越接近中心越多。这就有点像第一个圆形分配的例子，只不过这次要运用在矩形区域中。

我们为每个坐标生成一个随机数，然后求它们的平均数得到最终的值。例如，假设舞台有 500 像素宽。如果为每个对象随机生成一个 x 坐标，那么每个对象分布在哪里的概率都是相同的。但是如果从 0 到 500 产生两个随机数，再求平均，那么被置在舞台中心的机会就会略高一些。

让我们更深入地看一看这个问题。我们有一定的概率让两个数都在较“高”的范围内，假设从 300 到 500。让两个数都得到较低范围的概率也几乎相同，从 0 到 200。但是相比而言，一个数较高而另一个数较低，或者一个数居中另一个数较高或较低，或者都处于中等水平的概率要更高。所有这些可能性的平均值将使大多数点更靠近舞台中心。

OK，让我们来看代码。像平常一样，从一维的开始。下面是代码（可以在 Random5.as 中找到）：

```
package {
  import flash.display.Sprite;
  public class Random5 extends Sprite {
    private var numDots:uint = 300;
    public function Random5() {
      init();
    }
    private function init():void {
      for (var i:uint = 0; i < numDots; i++) {
        var dot:Ball = new Ball(1, 0);
        var x1:Number = Math.random() * stage.stageWidth;
        var x2:Number = Math.random() * stage.stageWidth;
        dot.x = (x1 + x2) / 2;
        dot.y = stage.stageHeight / 2 +
        Math.random() * 50 - 25;
        addChild(dot);
      }
    }
  }
}
```

这里我们创建了两个随机数 x1 和 x2，并设置 dot 的 x 坐标为它们的平均值。y 坐标只是邻近舞台中心点的一个简单的随机数。运行结果如图 19-7 所示。

图 19-7 一次迭代的偏向分布

虽然这个效果不是很明显，但是我们已经可以看到中心位置比边缘位置的点更多一些。创建更多的随机值，再取它们的平均值，会使效果更加明显。我们可以将它放入一个 for 循环中动态执行（Random6.as）：

```
package {
  import flash.display.Sprite;
  public class Random6 extends Sprite {
    private var numDots:uint = 300;
    private var maxRadius:Number = 50;
    private var iterations:uint = 6;
    public function Random6() {
      init();
```

```
  }
  private function init():void {
   for (var i:uint = 0; i < numDots; i++) {
    var dot:Ball = new Ball(1, 0);
    var xpos:Number = 0;
    for (var j:uint = 0; j < iterations; j++) {
     xpos += Math.random() * stage.stageWidth;
    }
    dot.x = xpos / iterations;
    dot.y = stage.stageHeight / 2 +
    Math.random() * 50 - 25;
    addChild(dot);
   }
  }
 }
}
```

这里 iterations 变量控制要取多少个数的平均值。开始让 xpos 变量等于 0，然后将每个随机数都加在上面。最后，用这个结果除以 iterations 得到最终的值。运行结果如图 19-8。

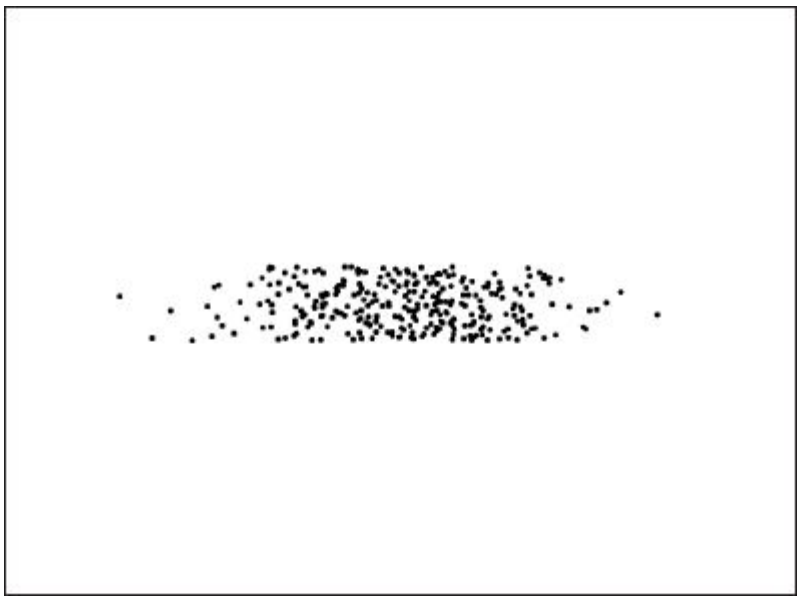


图 19-8 六次迭代的偏向分布

现在，要 y 轴也做同样的事就非常简单了，Random7.as：

```
package {
 import flash.display.Sprite;
 public class Random7 extends Sprite {
  private var numDots:uint = 300;
  private var maxRadius:Number = 50;
  private var iterations:uint = 6;
  public function Random7() {
   init();
  }
  private function init():void {
   for (var i:uint = 0; i < numDots; i++) {
    var dot:Ball = new Ball(1, 0);
    var xpos:Number = 0;
    for (var j:uint = 0; j < iterations; j++) {
```

```
        xpos += Math.random() * stage.stageWidth;
      }
      dot.x = xpos / iterations;
      var ypos:Number = 0;
      for (j = 0; j < iterations; j++) {
        ypos += Math.random() * stage.stageHeight;
      }
      dot.y = ypos / iterations;
      addChild(dot);
    }
  }
 }
}
```

这种分布如图 19-9 所示。

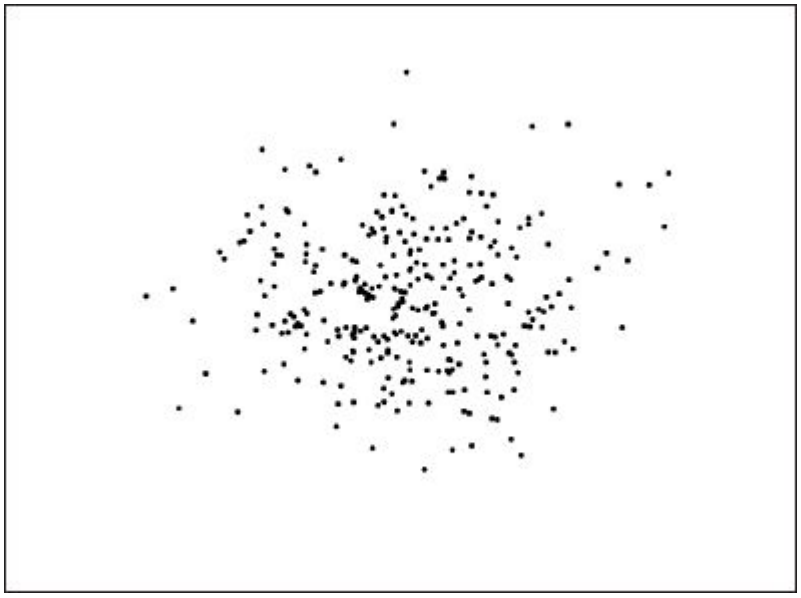


图 19-9 二维的偏向分布

在我看来，这是所有分布效果中最无规律，最具爆发性，最像星系的一种分布，虽然它也是最复杂的。好了，我们现在至少已经掌握了四种生成随机位置的方法。

基于计时器与时间的动画

到目前为止本书的所有例子都是通过把运动代码放到 onEnterFrame 方法中并将它赋给一个 enterFrame 事件的处理函数来实现的。我一直认为这是最简单的一种方式，因为帧的概念在 Flash 中根深蒂固，它就是给我们准备的；我猜我们大多都习以为常了。

然而，对于那些从非 Flash 编程环境转来的朋友，对于这种模式可能并不习惯。对于他们来说，时序动画模型（使用 Interval 或 Timer）似乎可以更加精准地控制动画。

稍后，我们要来看看“基于时间的动画”，一种即能用作帧又能用作计时器的技术。

基于计时器的动画

作为计时器动画使用的关键类，不出意料，它就是 flash.utils.Timer。同时我们还需要 flash.events.TimerEvent 类。

使用计时器实际上与使用 enterFrame 没什么两样。只需要我们去创建一个计时器(Timer)，告诉它多久“滴答响”一声，并侦听 TimerEvent.TIMER 事件，就像对 Event.ENTER_FRAME 事件的侦听一样。哦，还要告诉计时器何时开始！接下来，计时器就会每隔一段时间广播一个计时事件，它将调用赋给它的函数进行处理。计时器触发的间隔

以毫秒为单位，在创建该计时器时指定。让我们来看一个简单的例子（可在 Timer1.as 中找到）：

```
package {
 import flash.display.Sprite;
 import flash.utils.Timer;
 import flash.events.TimerEvent;
 public class Timer1 extends Sprite {
  private var timer:Timer;
  public function Timer1() {
   init();
  }
  private function init():void {
   timer = new Timer(30);
   timer.addEventListener(TimerEvent.TIMER, onTimer);
   timer.start();
  }
  private function onTimer(timer:TimerEvent):void {
   trace("timer!");
  }
 }
}
```

重要的部分加粗表示。我们创建一个计时器，告诉它每隔 30 毫秒触发一次，意味着每秒约 33 次。添加一个事件的侦听器并将它起动。 onTimer 方法与我们以前用的 onEnterFrame 类似。

这是我们所要知道计时器的大部分内容。它还有其它两个漂亮的特征。一个是在创建计时器时，可以通过第二个参数，repeatCount，告诉它运行的次数。假设我们要让计时器每秒运行一次，总共执行 5 秒。就可以这样做：

```
timer = new Timer(1000, 5);
```

如果没有指定重复的次数，或传入 0，那么计时器将无限地运行。

另一个好东西是可以让计时器在某个点上启动或停止，只需要调用 timer.stop 或 timer.start 即可。在某些例子中，这样做比删除和重新加入事件侦听器更简单一些。

与 enterFrame 相比，很多朋友更喜欢使用计时器的原因是，理论上讲，计时器可以让我们控制动画的速度——这是对于帧的不精确性的一个重大改进。我之所以说“理论上讲”，是因为这里有些事情需要弄清楚。

首先，实际上计时器要依赖于帧频。另一个原因是计时器的事件函数中的代码会增加整个计时器间隔。稍后我会解释一下第二点。现在，让我们看看计时器是如何与帧频相关联的。下面是文档类 Timer2.as，使用到我们著名的 Ball 类。

```
package {
 import flash.display.Sprite;
 import flash.utils.Timer;
 import flash.events.TimerEvent;
 public class Timer2 extends Sprite {
  private var timer:Timer;
  private var ball:Ball;
  public function Timer2() {
   init();
  }
  private function init():void {
   stage.frameRate = 1;
```

```
    ball = new Ball();
    ball.y = stage.stageHeight / 2;
    ball.vx = 5;
    addChild(ball);
    timer = new Timer(20);
    timer.addEventListener(TimerEvent.TIMER, onTimer);
    timer.start();
   }
   private function onTimer(event:TimerEvent):void {
    ball.x += ball.vx;
   }
  }
}
```

这里我们把创建出来的小球放在舞台的左侧。让它以 vx 为 5 的速度穿过舞台。然后设置一个 20 毫秒的计时器，每秒约调用 50 次。同时设置影片的帧频为 1 就是为了看看帧频是否会对计时器有影响。

测试后，我们会看到小球没有平稳地穿过屏幕，而是每秒钟跳一下 —— 以帧的频率。每跳一次都会大于 5 像素。为什么呢？

回想一下一、二章的动画基础，我们知道模型是需要更新的，所以屏幕要根据新的模型被刷新。这里时间间隔函数确实将更新了模型并将小球每次移动 5 像素，但是只有在 Flash 进入新的一帧时才进行刷新。仅仅运行一个函数不会驱使 Flash 进行重绘。

幸运的是，Macromedia (现在的 Adobe) 的好人们看到了这个问题并给了我们另一个小工具：updateAfterEvent。最初是在 Flash MX 中介绍的，现在它是传给计时器事件函数中 TimerEvent 对象的一个方法。就像它的名字一样，在事件之后刷新屏幕的。当然，由于它是 TimerEvent 类的一个方法，所以只有在收到一个事件后才能被调用。（事实上，它也是 KeyboardEvent 和 MouseEvent 的方法，因此也能在这些处理函数中调用。）

这样一来，我们可以修正一下 onTimer 事件：

```
private function onTimer(event:TimerEvent):void {
 ball.x += ball.vx;
 event.updateAfterEvent();
}
```

现在一切有所好转了。非常流畅。但是如果您意识到小球应该每秒更新 50 次，我们看到的基本上应该是一个 50 fps 的动画。这就意味着小球的运动应该比第四章创建的 fps 小于 50 的 enterFrame 事件的例子更为流畅。但是实际的运动更为缓慢。

问题出来了，计时器在某种程度上要依赖于帧频。通过我的测算，在帧频为 1 fps 时，我们所得到的计时器运行的最快间隔大约为 100 毫秒。

我已经听到了嘲笑：每帧只得到了 10 次间隔。所以，试将帧频改为 5。它允许每秒更新 50 次。在我看来，仍然不是很流畅。如果不大于每秒 10 帧的话是不会达到真正流畅的效果。因此，我们可以看到使用计时器并不能完全让我们从帧频的铐链中解脱出来。

下一个问题是计时器内部是怎样工作的，它会对计时的精确度产生多大的影响。当 timer.start() 被调时，实际上发生了什么，Flash 等待一段指定的时间，然后广播事件，运行与该计时器相关的处理函数。只有当函数执行完成后计时器才开始定时下一次计时。看一个例子，假设我们有一个每 20 毫秒运行一次计时器。假设在处理函数中有大量的代码需要执行 30 毫秒。下一轮定时只有在所有的代码都运行完成后才开始。这样一来，我们的函数会在大约每 50 毫秒调用一次。因为在用户的机器上没法精确地知道代码会运行多快，所以多数情况下，计时器动画不会比帧动画精确多少。

如果您需要真正的精确，那么基于时间的动画则是我们的必经之路。

基于时间的动画

如果要让动画中物体的速度是一致的，那么基于时间的动画就是我们要使用的方法。这种方法在一些游戏中比较常用。我们知道，帧和计时器动画都不能以特定的速率播放。一个复杂的动画在一台又老又慢的电脑上运行可能要比最初设计的速度慢上许多。我们马上会看到，使用基于时间的动画无论最终动画运行的帧频如何，都将获得可靠的速度。

首先要改变考虑速度的方法。到目前为止，在我说 vx = 5 时，我们使用的单位是像素每帧（pixels per frame）。换句话讲，每进入新的一帧物体都将在 x 轴上运动 5 像素。在计时器动画中，当然，就应该是每次定时间隔移动 5 像素。

对于时间动画，我们将使用真正的时间单位，如秒。由于我们是处理完整的一秒，而非其中的一部分，因此这个速度值就要更大一些。如果某个物体的速度是每帧 10 像素，每秒 30 帧的速度运动，大约每秒 300 像素。比如下面这个例子，我从第六章的 Bouncing2.as 文档类中截取了一部分并进行了一些变化，见下面粗体部分（也可在 TimeBased.as 中找到）：

```
package {
 import flash.display.Sprite;
 import flash.events.Event;
 import flash.utils.getTimer;
 public class TimeBased extends Sprite {
  private var ball:Ball;
  private var vx:Number;
  private var vy:Number;
  private var bounce:Number = -0.7;
  private var time:Number;
  public function TimeBased() {
   init();
  }
  private function init():void {
   stage.frameRate = 10;
   ball = new Ball();
   ball.x = stage.stageWidth / 2;
   ball.y = stage.stageHeight / 2;
   vx = 300;
   vy = -300;
   addChild(ball);
   time = getTimer();
   addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
   var elapsed:Number = getTimer() - time;
   time = getTimer();
   ball.x += vx * elapsed / 1000;
   ball.y += vy * elapsed / 1000;
   var left:Number = 0;
   var right:Number = stage.stageWidth;
   var top:Number = 0;
   var bottom:Number = stage.stageHeight;
   if (ball.x + ball.radius > right) {
    ball.x = right - ball.radius;
    vx *= bounce;
```

```
    } else if (ball.x - ball.radius < left) {
      ball.x = left + ball.radius;
      vx *= bounce;
    }
    if (ball.y + ball.radius > bottom) {
      ball.y = bottom - ball.radius;
      vy *= bounce;
    } else if (ball.y - ball.radius < top) {
      ball.y = top + ball.radius;
      vy *= bounce;
    }
  }
 }
}
```

如上所描述，我改变了对速度的计算，让它们使用固定的值，而非随机值。然后我创建了一个名为 time 的变量，让它等于 flash.utils.getTimer 函数的结果。getTimer 函数非常简单。它返回影片已经运行了的毫秒数 —— 这就是它全部的工作。我们没有办法清除它，重启它，改变它，等等。它只是一个计数器。

看起来它似乎用处不大，但是如果先调用一次 getTimer 将值保存起来，稍后再调用它一次，将两个结果相减，我们就能知道确切的—— 毫秒 ——这两次调用之间经过了多少时间。

这就是策略：在每帧的开始时调用 getTimer 计算与上一帧间隔了多长时间。如果将它除以 1,000，我们将得到以秒为单位的间隔时间，这是个以秒为单位的分数。由于我们的 vx 和 vy 现在是以像素每秒来计算的，因此可以让它们去乘以这个分数，这样就知道要对物体移动多少了。同样，不要忘记重新设置 time 变量的值，以便让下一帧进行计算。

测试一下，我们将看到小球移动的速度几乎与最初的速度相同！但是真正令人激动的是我们可以以任何帧频来发布这个影片，它仍然可以以同样的速度运动！通过修改 stage.frameRate 的值，试验一下高到 1,000 fps，低到 10 fps，你会看到小球的速度是相同的。当然，较高的频率会让动画更加流畅，而较低的频率则会十分不连贯，但是速度是一致的。

大家可以把这个技术应用在本书任何一个包含速度的例子中。如果这样的话，还需要将相似的技术应用在加速度或外力上，如重力，因为它们也是基于时间的。加速度肯定要比转前要大很多，因为加速度被定义为速度对时间的变化率。例如，重力大约为 32 英尺 / 秒 / 秒。

如果在一个 30 fps 帧的动画中，重力为 0.5 的话，那么现在就应该是 450。计算方法 0.5 * 30 * 30。然后像这样将它加入：

```
vy += gravity * elapsed / 1000;
```

在最后一个例子中加入 450 重力后测试一下。我们会看到它与帧动画中加入重力 0.5 的效果是相同的。使用这种技术的一个技巧是将帧频设置得非常高，如 100。虽然没有机器能够与真正的帧频相吻合，但是基于时间的动画将保证每个人看到的影片运行得都是最流畅的。

同质量物体的碰撞

回忆一下第十一章的动量守恒。那里有非常严谨的代码。不过，当两个相同质量的物体发生碰撞时，我们实现起来可以更简单一些。基本原理是，两个物体沿着碰撞的线路交换它们的速度。同时还要用坐标旋转来决定碰撞的线路，以及物体的速度，这样就摆脱了复杂的动量守恒公式。来看看它是如何工作的，让我们回到文件 MultiBilliard2.as 中，将用它作为下一个例子 SameMass.as 的基础。我就不再列出原先所有的代码了，因为它实在是一个很

大的文件。但是，我们要来看一下创建所有小球的 for 循环：

```
for(var i:uint = 0; i < numBalls; i++) {
var radius:Number = Math.random() * 50 + 20;
  var ball:Ball = new Ball(radius);
  ball.mass = radius;
  ball.x = Math.random() * stage.stageWidth;
  ball.y = Math.random() * stage.stageHeight;
  ball.vx = Math.random() * 10 - 5;
  ball.vy = Math.random() * 10 - 5;
  addChild(ball);
  balls.push(ball);
}
```

对于新的文件来说，要把粗体字的部分改为这一行：

```
      var radius:Number = 30;
```

让所有小球大小都相同，消除了质量的概念，相当于给它们相同的质量。

接下来，看一下 checkCollision 函数。请找到这一部分：

```
// 旋转 ball0 的速度
var vel0:Point = rotate(ball0.vx,
ball0.vy,
sin,
cos,
true);
// 旋转 ball1 的速度
var vel1:Point = rotate(ball1.vx,
ball1.vy,
sin,
cos,
true);
// 碰撞反应
var vxTotal:Number = vel0.x - vel1.x;
vel0.x = ((ball0.mass - ball1.mass) * vel0.x +
2 * ball1.mass * vel1.x) /
(ball0.mass + ball1.mass);
vel1.x = vxTotal + vel0.x;
```

这一部分找出了碰撞线路上的速度，根据物体的质量求出碰撞的结果。“碰撞反应”部分是动量守恒的要素，这就是我们可以消除的部分。我们可以让 vel0 和 vel1 进行交换，就可以很容易地替换它了。整个代码段如下：

```
// 旋转 ball0 的速度
var vel0:Point = rotate(ball0.vx,
ball0.vy,
sin,
cos,
true);
// 旋转 ball1 的速度
var vel1:Point = rotate(ball1.vx,
ball1.vy,
sin,
cos,
true);
```

```
// 交换两个速度
var temp:Point = vel0;
vel0 = vel1;
vel1 = temp;
```
这里甚至还可以再优化，但是为了代码的清晰我就不做改变了。现在我们已经摆脱了一小块儿数学问题，测试一下修改前与修改后的文件，所得的结果是相同的。

声音整合

本书一直没有提到声音的使用。因为声音并不是动画的直接组成部分，优质的声音效果可以让 Flash 影片更加真实、引人入胜。

我们可以使用不同的方法来加入声音。回溯到 Flash IDE 最早的程序版本，我们有一种使用声音的特殊方式 —— 将声音导入到库，再把它加入到帧里面。这不是我要介绍的方法。我将介绍在 AS 3 中使用声音的一些基础。

AS 3 的 Sound 类有了很大的变化，而且还有几个额外的类可以帮助我们对其进行修饰。这里没有太多的空间进行深入的讨论，但是有一个方面我想应该对于我们这本书来说会很有用。这就是当动画中发生某种事件时，应该播放声音。最明显的就应该是碰撞了。一个小球碰到墙上或其它小球上，我们会听到“砰”、“啵嘤”、“啪”或其它什么声音。因此，我们需要掌握通过 ActionScript 来启动声音的能力。

这个例子中，我们还要回到 Bouncing2.as，小球会与墙壁产生碰撞。每次碰撞到墙壁时，我想让它发出声音。新的类请见 SoundEvents.as。

首先，需要有声音效果。在网上有许多免费的声音效果资源。其中最火的 Flash 声音网站是 FlashKit。他们的音乐文件除了有 loop 以外，还有一个声音效果库 www.flashkit.com/soundfx/。这些效果被分类为 Cartoon，Interfaces，Mechanical 等等，而且这个网站有超过 6,000 多个声音效果文件，所以您应该能够找到适合自己的音效。我们可以在页面上直接进行预览（preview），找到自己喜欢的文件后，以 MP3 格式进行下载。将它保存到硬盘上与最终发部的影片放在同一目录下。

有时我们要重新给文件一个更为简单的名字。例如，我下载了一个“boing”声音，我就将它重命名为 boing.mp3。

在 AS 3 中使用声音的基础实际上要比 AS 2 中简单一些。

首先，我们需要创建声音对象。假设在类中已经声明了一个名为 boing 的变量：

```
private var boing:Sound;
```

创建一个声音对象就这么简单：

```
boing = new Sound();
```

当然，如同大多数 AS 3 的类一样，Sound 类也在包中，flash.media 包，因此要确保先导入 flash.media.Sound。

连接一个外部声音对象最简单的方法是在构造函数中传入一个请求（request）。

就像读取外部图像（第四章）一样，我们不能直接传入外部声音文件的 URL。而是要将它包装到 URLRequest 中（flash.net.URLRequest，需要导入它）。应该像这样：

```
boing = new Sound(new URLRequest("boing.mp3"));
```

全部内容就是这样。现在声音已经准备好。我们要做的就是：

```
mySound.play();
```

无论在哪儿都会播放出这个音效。在 play 中有一些可选参数，如偏移的毫秒数，以及播放的次数，但是默认情况下是从声音的起始位置播放一次声音，这已经满足了我们通常的需求。以下是 SoundEvents.as 的全部代码，展示了 Sound 对象的创建，无论何时小球碰撞到墙上，都会播放声音。

```
package {
```

```
import flash.display.Sprite;
import flash.events.Event;
import flash.media.Sound;
import flash.net.URLRequest;
public class SoundEvents extends Sprite {
  private var ball:Ball;
  private var vx:Number;
  private var vy:Number;
  private var bounce:Number = -0.7;
  private var boing:Sound;
  public function SoundEvents() {
    init();
  }
  private function init():void {
    boing = new Sound(new URLRequest("boing.mp3"));
    ball = new Ball();
    ball.x = stage.stageWidth / 2;
    ball.y = stage.stageHeight / 2;
    vx = Math.random() * 10 - 5;
    vy = -10;
    addChild(ball);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }
  private function onEnterFrame(event:Event):void {
    ball.x += vx;
    ball.y += vy;
    var left:Number = 0;
    var right:Number = stage.stageWidth;
    var top:Number = 0;
    var bottom:Number = stage.stageHeight;
    if (ball.x + ball.radius > right) {
      boing.play();
      ball.x = right - ball.radius;
      vx *= bounce;
    } else if (ball.x - ball.radius < left) {
      boing.play();
      ball.x = left + ball.radius;
      vx *= bounce;
    }
    if (ball.y + ball.radius > bottom) {
      boing.play();
      ball.y = bottom - ball.radius;
      vy *= bounce;
    } else if (ball.y - ball.radius < top) {
      boing.play();
      ball.y = top + ball.radius;
      vy *= bounce;
    }
  }
```

```
 }
}
```

测试一下影片看一看 … … 听一听拥有声音以后带来的不同感受。当然，要找到正确的声音用在正确的环境下，也不要加得太多，因为这本身也是一门艺术

实用公式

统领全书，我们已经有了各种运动和效果的公式。我已经提取出了最实用和最常用的公式、方程、以及代码的摘录，并将它们列在本章的最后。我认为将它们放到同一个地方应该对大家非常有帮助，因此我将这些我认为最需要的内容放到一起作为整体的一个参考资料。我将会在这一页夹上书签。

第三章

基础三角函数的计算：

角的正弦值 = 对边 / 斜边

角的余弦值 = 邻边 / 斜边

角的正切值 = 对边 / 邻边

弧度转换为角度以及角度转换为弧度：

弧度 = 角度 * Math.PI / 180

角度 = 弧度 * 180 / Math.PI

向鼠标（或者任何一个点）旋转：

```
// 用要旋转到的 x, y 坐标替换 mouseX, mouseY
dx = mouseX - sprite.x;
dy = mouseY - sprite.y;
sprite.rotation = Math.atan2(dy, dx) * 180 / Math.PI;
```

创建波形：

```
// 将 x, y 或其它属性赋值给 Sprite 影片或影片剪辑，
// 作为绘图坐标，等等。
public function onEnterFrame(event:Event){
 value = center + Math.sin(angle) * range;
 angle += speed;
}
```

创建圆形：

```
// 将 x, y 或其它属性赋值给 Sprite 影片或影片剪辑，
// 作为绘图坐标，等等。
public function onEnterFrame(event:Event){
```

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

创建椭圆：

```
// 将 x, y 或其它属性赋值给 Sprite 影片或影片剪辑，
// 作为绘图坐标，等等。
public function onEnterFrame(event:Event){
  xposition = centerX + Math.cos(angle) * radiusX;
  yposition = centerY + Math.sin(angle) * radiusY;
  angle += speed;
}
```

获得两点间的距离：

```
// x1, y1 和 x2, y2 是两个点
// 也可以是 Sprite / MovieClip 坐标，鼠标坐标，等等。
dx = x2 – x1;
dy = y2 – y1;
dist = Math.sqrt(dx*dx + dy*dy);
```

第四章

十六进制转换为十进制：

```
trace(hexValue);
```

十进制转换为十六进制：

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

颜色组合：

```
color24 = red << 16 | green << 8 | blue;
color32 = alpha << 24 | red << 16 | green << 8 | blue;
```

颜色提取：

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

```
alpha = color32 >> 24;
red = color32 >> 16 & 0xFF;
green = color32 >> 8 & 0xFF;
blue = color232 & 0xFF;
```

穿过某点绘制曲线：

```
// xt, yt 是我们想要穿过的一点
// x0, y0 以及 x2, y2 是曲线的两端
x1 = xt * 2 – (x0 + x2) / 2;
y1 = yt * 2 – (y0 + y2) / 2;
moveTo(x0, y0);
curveTo(x1, y1, x2, y2);
```

第五章

角速度转换为 x, y 速度：

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

角加速度（作用于物体上的 force）转换为 x, y 加速度：

```
ax = force * Math.cos(angle);
ay = force * Math.sin(angle);
```

将加速度加入速度：

```
vx += ax;
vy += ay;
```

将速度加入坐标：

```
movieclip._x += vx;
sprite.y += vy;
```

第六章

移除出界对象：

```
if(sprite.x - sprite.width / 2 > right ||
sprite.x + sprite.width / 2 < left ||
sprite.y – sprite.height / 2 > bottom ||
sprite.y + sprite.height / 2 < top)
{
```

```
 // 删除影片的代码
}
```

重置出界对象：

```
if(sprite.x - sprite.width / 2 > right ||
sprite.x + sprite.width / 2 < left ||
sprite.y – sprite.height / 2 > bottom ||
sprite.y + sprite.height / 2 < top)
{
 // 重置影片的位置和速度
}
```

屏幕环绕出界对象：

```
if(sprite.x - sprite.width / 2 > right)
{
 sprite.x = left - sprite.width / 2;
}
else if(sprite.x + sprite.width / 2 < left)
{
 sprite.x = right + sprite.width / 2;
}
if(sprite.y – sprite.height / 2 > bottom)
{
 sprite.y = top – sprite.height / 2;
}
else if(sprite.y + sprite.height / 2 < top)
{
 sprite.y = bottom + sprite.height / 2;
}
```

摩擦力应用（正确方法）：

```
speed = Math.sqrt(vx * vx + vy * vy);
angle = Math.atan2(vy, vx);
if(speed > friction)
{
 speed -= friction;
}
else
{
 speed = 0;
}
vx = Math.cos(angle) * speed;
vy = Math.sin(angle) * speed;
```

摩擦力应用（简便方法）：

```
vx *= friction;
vy *= friction;
```

第八章：

简单缓动运动，长形：

```
var dx:Number = targetX - sprite.x;
var dy:Number = targetY - sprite.y;
vx = dx * easing;
vy = dy * easing;
sprite.x += vx;
sprite.y += vy;
```

简单缓动运动，中形：

```
vx = (targetX - sprite.x) * easing;
vy = (targetY - sprite.y) * easing;
sprite.x += vx;
sprite.y += vy;
```

简单缓动运动，短形：

```
sprite.x += (targetX - sprite.x) * easing;
sprite.y += (targetY - sprite.y) * easing;
```

简单弹性运动，长形：

```
var ax:Number = (targetX - sprite.x) * spring;
var ay:Number = (targetY - sprite.y) * spring;
vx += ax;
vy += ay;
vx *= friction;
vy *= friction;
sprite.x += vx;
sprite.y += vy;
```

简单弹性运动，中形：

```
vx += (targetX - sprite.x) * spring;
vy += (targetY - sprite.y) * spring;
```

```
vx *= friction;
vy *= friction;
sprite.x += vx;
sprite.y += vy;
```

简单弹性运动，短形：

```
vx += (targetX - sprite.x) * spring;
vy += (targetY - sprite.y) * spring;
sprite.x += (vx *= friction);
sprite.y += (vy *= friction);
```

偏移弹性运动：

```
var dx:Number = sprite.x - fixedX;
var dy:Number = sprite.y - fixedY;
var angle:Number = Math.atan2(dy, dx);
var targetX:Number = fixedX + Math.cos(angle) * springLength;
var targetY:Number = fixedX + Math.sin(angle) * springLength;
// 如前例弹性运动到 targetX, targetY
```

第九章

距离碰撞检测：

```
// 从影片 spriteA 和 spriteB 开始
// 如果使用一个空白影片，或影片没有半径（radius）属性
// 可以用宽度或高度除以 2。
var dx:Number = spriteB.x - spriteA.x;
var dy:Number = spriteB.y - spriteA.y;
var dist:Number = Math.sqrt(dx * dx + dy * dy);
if(dist < spriteA.radius + spriteB.radius)
{
 // 处理碰撞
}
```

多物体碰撞检测：

```
var numObjects:uint = 10;
for(var i:uint = 0; i < numObjects - 1; i++)
{
 // 使用变量 i 提取引用
 var objectA = objects[i];
 for(var j:uint = i+1; j
 {
  // // 使用变量 j 提取引用
```

```
    var objectB = objects[j];
    // perform collision detection
    // between objectA and objectB
  }
}
```

第十章

坐标旋转：

```
x1 = Math.cos(angle) * x - Math.sin(angle) * y;
y1 = Math.cos(angle) * y + Math.sin(angle) * x;
```

反坐标旋转：

```
x1 = Math.cos(angle) * x + Math.sin(angle) * ;y
y1 = Math.cos(angle) * y - Math.sin(angle) * x;
```

第十一章

动量守恒的数学表达式：

$$v0Final = \frac{(m0 - m1) * v0 + 2 * m1 * v1}{m0 + m1}$$

$$v1Final = \frac{(m1 - m0) * v1 + 2 * m0 * v0}{m0 + m1}$$

动量守恒的 ActionScript 表达式，短形:

```
var vxTotal:Number = vx0 - vx1;
vx0 = ((ball0.mass - ball1.mass) * vx0 +
2 * ball1.mass * vx1) /
(ball0.mass + ball1.mass);
vx1 = vxTotal + vx0;
```

第十二章

引力的一般公式：

```
force = G * m1 * m2 / distance2
```

ActionScript 实现万有引力：

```
function gravitate(partA:Ball, partB:Ball):void
```

```
{
  var dx:Number = partB.x - partA.x;
  var dy:Number = partB.y - partA.y;
  var distSQ:Number = dx * dx + dy * dy;
  var dist:Number = Math.sqrt(distSQ);
  var force:Number = partA.mass * partB.mass / distSQ;
  var ax:Number = force * dx / dist;
  var ay:Number = force * dy / dist;
  partA.vx += ax / partA.mass;
  partA.vy += ay / partA.mass;
  partB.vx -= ax / partB.mass;
  partB.vy -= ay / partB.mass;
}
```

第十四章

余弦定理

```
a2 = b2 + c2 - 2 * b * c * cos A
b2 = a2 + c2 - 2 * a * c * cos B
c2 = a2 + b2 - 2 * a * b * cos C
```

ActionScript 的余弦定理:

```
A = Math.acos((b * b + c * c - a * a) / (2 * b * c));
B = Math.acos((a * a + c * c - b * b) / (2 * a * c));
C = Math.acos((a * a + b * b - c * c) / (2 * a * b));
```

第十五章

基本透视法：

```
scale = fl / (fl + zpos);
sprite.scaleX = sprite.scaleY = scale;
sprite.alpha = scale; // 可选
sprite.x = vanishingPointX + xpos * scale;
sprite.y = vanishingPointY + ypos * scale;
```

Z 排序：

```
// 假设有一个带有 zpos 属性的 3D 物体的数组
objectArray.sortOn("zpos", Array.DESCENDING | Array.NUMERIC);
for(var i:uint = 0; i < numObjects; i++)
{
  setChildIndex(objectArray[i], i);
}
```

坐标旋转：

```
x1 = cos(angleZ) * xpos - sin(angleZ) * ypos;
y1 = cos(angleZ) * ypos + sin(angleZ) * xpos;
x1 = cos(angleY) * xpos - sin(angleY) * zpos;
z1 = cos(angleY) * zpos + sin(angleY) * xpos;
y1 = cos(angleX) * ypos - sin(angleX) * zpos;
z1 = cos(angleX) * zpos + sin(angleX) * ypos;
```

3D 距离：

```
    dist = Math.sqrt(dx * dx + dy * dy + dz * dz);
```

Making Things Move! 结束语 [FL 基理文]

历经四个月的时间，今天终于把 Foundation Actionscript 3.0 Animation —— Making Things Move! 一书全部译完。非常感谢本书作者 Keith Peters 带给我们这么好的教材，他对本书的内容可以说是精雕细琢，不断提炼。为了能让这本经典教材呈现出它特有的魅力，我不惜花费大量的时间研究、揣摩作者的意图，尽可能地表述完整、准确。我的译文中拒绝出现蹩脚、拗口、晦涩、难懂的词句，尤其是在原理的阐述上更要清楚、明确，不能让读者在学习知识的过程中再去推敲这段话想表述什么意思。因此，我采取的策略就是转译。尽可能地用我们的思维习惯和语言方式来表示作者的意图，立争用最清晰、明快的语言将原理阐述明白，这是对您的时间和精力的最大尊重。非常感谢亲爱的网友们这么长期以来的支持，忠心地感谢大家，让我们一起为 Flash 事业而奋斗！