

# Flash ActionScript 3.0 高级动画教程

(AdvancED ActionScript 3.0 Animation)

Keith Peters

## 感谢

特别感谢各位译者和整理者无私的劳动才使本书最终成型。由于此书采用多人协同翻译，每个人的水平又不尽相同，难免会出现或多或少的问题，在这里也请每位读者能怀着一份感激和宽容的心情阅读此书。如在阅读中发现错误和不妥的地方，请到<http://www.riabook.cn> 留言，以便我们再版时可以更新这些问题。

### 参与翻译：

hbbalfred , LiScott , Y.Boy , daWei Yang, yujjj , actions

### 参与整理：

高山, Y.Boy, LiScott , N 神

本书最新消息, 译者联系方式, 错误提交等, 请关注以下网址:

[http://www.riabook.cn/zh\\_book/advanced-as3-animation-full-cn.htm](http://www.riabook.cn/zh_book/advanced-as3-animation-full-cn.htm)

### 免责声明：

本书仅供网络间学习交流之用, 请关注正版书籍。

2009 年 7 月



# 本书目录

## 第一章 高级碰撞检测

- 不规则图形的检测碰撞
- BitmapData.hitTest 用于非位图
- 大量对象的碰撞检测
- 实现基于网格的碰撞检测
- 编写网格代码
- 测试并调整网格
- 整理成类
- 使用此类
- 检测不只是为了碰撞
- 总结

## 第二章 转向行为

- 行为
- 2D 向量(Vector2D)类
- 机车(Vehicle)类
- 转向机车(SteeredVehicle)类
- 寻找行为
- 避开行为
- 到达行为
- 追捕行为
- 躲避行为
- 漫游行为
- 对象回避
- 路径跟随
- 群落
- 总结

## 第三章 等角投影

- 等角投影
- 等角 vs 二等角(dimetric)
- 创建等角图形
- 等角形变
- 形变坐标与屏幕坐标
- 屏幕坐标转换等角坐标
- IsoUtils 类
- 等角对象
- 层深排序
- 等角世界类
- 3D 移动
- 碰撞检测
- 使用外部图形
- 等角地图
- 总结

## 第四章 寻路

寻路基础

A-star

A\_star 运算法则

代价计算

图解运算过程

代码实现

常见的 Astar 估价公式

使用 Astar 类

修改路径细节：拐角

在游戏中使用 Astar

进阶教程

总结

## 第五章 二级输入设备：摄像头和麦克风

摄像头和麦克风

输入的声音

声控游戏

活跃事件

输入的视频

视频尺寸和质量

视频和位图

反转图像

分析像素

分析颜色

将跟踪颜色视作输入

分析移动区域

边缘检测

总结

## 第六章 高等物理：数值积分

时间驱动的运动

编程 RK2

编程 RK4

薄弱环节

总结 Runge-Kutta

Verlet 积分法

Verlet 点

点的约束

Verlet 线段

Verlet 结构体

拉链式结构

深入研究

总结

## 第七章 3D in Flash10

Flash 3D 基础

设置消失点

3D 坐标

景深排序

3D 容器

3D 旋转

视野和焦距

屏幕坐标系和 3D 坐标系

本章小结

## 第八章 Flash10 的绘画 API

路径

一个简单的例子

画曲线

wide 绘制命令和 NO\_OP

缠绕

三角

位图填充和三角

uvtData

更多三角

三角和 3D

uvt 中的 t

旋转圆柱

创建一个 3D 地球

图形数据

总结

## 第九章 Pixel Bender

什么是 Pixel Bender?

编写一个 Pixel Shader

数据类型

获取当前像素坐标

参数

高级参数

对输入图片进行取样

线性取样

适用于 Flash 里的 Twirl Shader

在 Flash 里使用 Pixel Bender

加载或绑定 shader

使用 shader 作为绘制填充

访问 shader 元数据

设置 shader 参数值

转换 shader 填充

用 shader 填充制作动画

指定 shader 的输入图片

使用 shader 作为滤镜

使用 shader 作为混合模式

总结

## 第十章 补间引擎

Flash 的 Tween 类

缓动方法

合并补间

Flex Tween 类

Flex Tween 类的缓动函数

Tween 组合  
Tween 序列  
补间引擎  
Tweener 的缓动函数  
Tweener 的 tween 组合  
Tweener 的 tween 序列  
TweenLite/TweenGroup  
TweenLite 的缓动函数  
TweenLite 的 tween 组合  
TweenLite/TweenGroup 的序列  
KitchenSync  
KitchenSync 的缓动函数  
用 kitchenSync 改变多个对象或属性  
KitchenSync 的 tween 序列  
KitchenSync 的 tween 序列  
GTween  
gTween 的缓动函数  
用 gTween 改变多个对象  
gTween 的补间序列  
总结

# 第一章 高级碰撞检测

碰撞检测是数学、是艺术、是科学、是一种泛化推测，用来判定一些对象是否接触到别的对象。听起来貌似挺简单，但当你面对的仅仅是由内存中一连串不同属性所表示的对象时，一些困难就接踵而至。

上本书里提到了基本的碰撞检测法。这章除了要讲一个上本书里没有的方法外，还要讲一个处理大量对象的碰撞检测的策略。

注意主题是碰撞检测，所以不研究在检测到碰撞后要做的事情。如果是开发游戏，可能会想着在碰撞后发生爆炸，变色，或者消失。有些碰撞结果的处理方法在上本书的“动量守恒”一章有提及。但最终怎么处理还是由您自己决定（视具体要求而定）。

## 不规则图形的检测碰撞

上本书中的基本检测方法提到了内置的 `hitTestObject` 和 `hitTestPoint` 函数，还有通过距离的检测。这些方法在对图形对象使用时有各自的限制。`hitTestObject` 对两个标准矩形的检测相当不错，但其他图形就无能为力。`hitTestPoint` 适用于发现鼠标是否接触某个图形或者一些很小的近似成一点的图形是否碰到其他图形，但对两个大点图形亦束手无策。通过距离检测则只能用于圆形。

下面要介绍的 Holy Grail 碰撞检测法，则用来检测两个不规则图形之间是否有检出。虽然这个方法在 Flash8 时代就已经存在，但还是没有出现在上一本书中。而实际上就是 `BitmapData.hitTest`。

`BitmapData.hitTest` 比较两个 `BitmapData` 对象，并告之是否有像素重叠。听起来貌似还是很简单，但复杂度会随思考的深入而加剧。从形状上讲，位图其实是由一堆像素组成的矩形，所以最简单的情况就好像 `hitTestObject`。真正有用的是支持透明的位图。

当创建一个位图对象时，可以指定是否支持透明。

```
new BitmapData(宽, 高, 透明, 颜色);
```

第三个参数是一个布尔值决定是否透明。如果 `false`，则位图不透明。最初呈现为一块由指定颜色填充的矩形。虽然可以通过各种 `Bitmapdata` 函数改变像素，但由于完全不透明始终会盖住后面的内容。每个像素的颜色值是一个 24 位的二进制，一般用 6 个十六进制表示成 RGB 格式 `0xRRGGBB`。左起第一对指定红色通道的值从 00(0) 到 FF(255)，第二对是绿色，第三对是蓝色。比如 `0xFFFFFFFF` 意思就是白色，`0xFF0000` 是红色，`0xFF9900` 是桔色。通过 `setPixel` 和 `getPixel` 函数可以设置和读取单个像素的颜色值。

当设置 `BitmapData` 支持透明后，每个像素就带了透明(alpha)通道，颜色值变成了 32 位的二进制，格式也改为 8 个十六进制的 ARGB。开头的 2 位决定透明度。00 表示完全透明，FF 表示完全不透明。设置和读取单个像素值的函数要用 `setPixel32` 和 `getPixel32`。注意此时专递的是 32 位的数字。如果用 24 位的，比如 `0xFFFFFFF`，前面自动补零为 `0x00FFFFFF`，结果将会完全透明。

来看看所说的两种不同情况。

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

    public class BitmapCompare extends Sprite
    {
        public function BitmapCompare()
    }
}
```

```

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

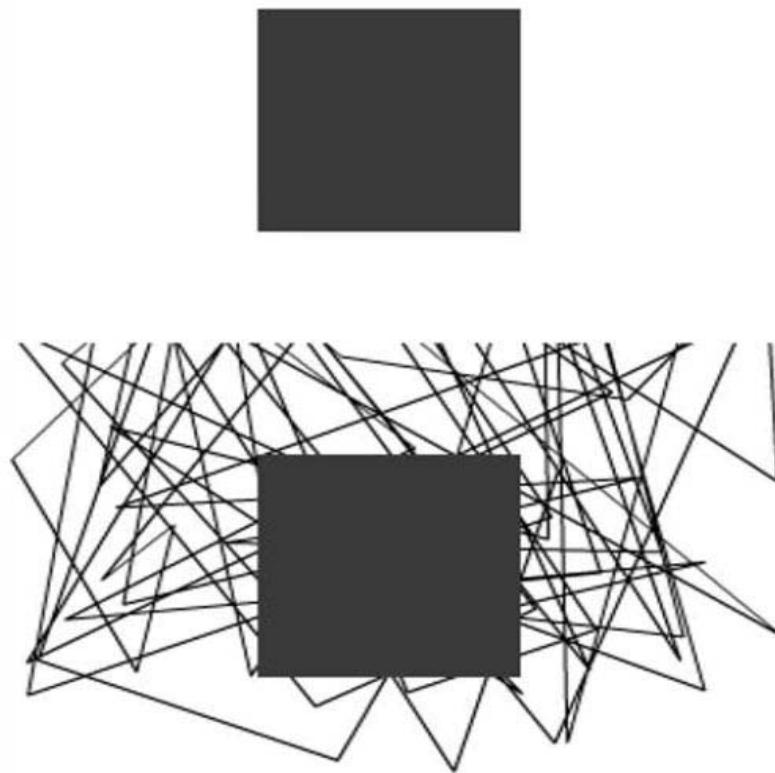
// draw a bunch of random lines
graphics.lineStyle(0);
for(var i:int = 0; i < 100; i++)
{
    graphics.lineTo(Math.random() * 300, Math.random() * 400);
}

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

// create a transparent bitmap
var bmpd2:BitmapData = new BitmapData(300, 200, true, 0x00ffff);
bmpd2.fillRect(new Rectangle(100, 50, 100, 100), 0x80ff0000);
var bmp2:Bitmap = new Bitmap(bmpd2);
bmp2.y = 200;
addChild(bmp2);
}
}

```

这段代码一开始在场景随机画了些线条，只是为了证明两种不同的位图，没什么其他意思。然后创建了两个位图并在中央画上了一个红色的方块。上面的位图是不透明的，所以线条都被盖住了，下面的透明，所以只有画方块的地方，线条才被盖住。

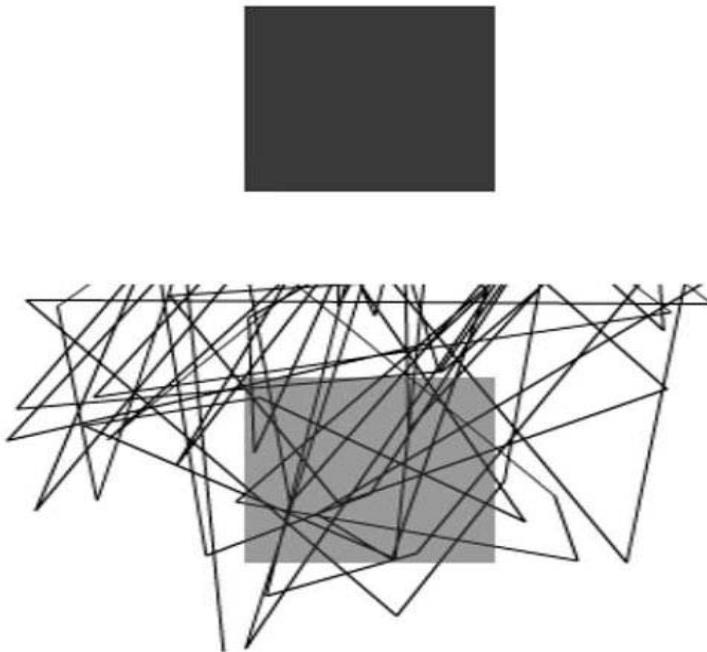


**Figure 1-1.** An opaque bitmap on top, transparent below

此外，对于支持透明的位图来说可以采用具体的透明度。把代码中第二个画矩形的函数改一下

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

注意看，我们使用了 32 位的 AARRGGBB 格式作为颜色值来填充，透明度为半透明 0x80，十进制等于 128。如图



**Figure 1-2.** A semitransparent square

#### 位图间的碰撞检测

现在让我们看看如果完成位图间的碰撞检测。首先我们需要一个不规则图形，五角星就挺不错。为了以后方便将其作成一个 Star 类。

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

    public class Star extends Sprite
    {
        public function Star(radius:Number,color:uint=0xFFFF00):void
        {
            graphics.lineStyle(0);
            graphics.moveTo(radius,0);
            graphics.beginFill(color);
            // draw 10 lines
            for (var i:int=1; i < 11; i++)
            {
                var radius2:Number=radius;
                if (i % 2 > 0)
                {
                    // alternate the radius to make spikes every other line
                    radius2=radius / 2;
                }
                var angle:Number=Math.PI * 2 / 10 * i;
                graphics.lineTo(Math.cos(angle) * radius2,Math.sin(angle) * radius2);
            }
        }
    }
}
```

这段代码通过角度不断的增加以及半径的改变，画出一个星星。

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

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

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

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

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

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

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

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

            // the hit test itself.
            if(bmpd1.hitTest(new Point(bmp1.x, bmp1.y), 255, bmpd2,new Point(bmp2.x, bmp2.y),255))
```

```
        {
            bmp1.filters = [new GlowFilter()];
            bmp2.filters = [new GlowFilter()];
        }
        else
        {
            bmp1.filters = [];
            bmp2.filters = [];
        }
    }
}
```

这里我们使用了 Star 类，创建了一个半径等于 50 的星星，并绘制成两张位图。使用矩阵 (matrix) 是为了在绘制时偏移 50 个像素，因为星星的(0, 0)点在中心点，绘制点视(0, 0)点作为左上角起点，所以偏移是为了绘制完整的星星。

bmp1 的位置固定不变， bmp2 则跟随鼠标移动。下面的代码才是关键：

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

这里才是真正判断两个位图是否接触的代码。BitmapData.hitTest 函数如下所示：

hitTest(一个点: Point,

一个透明阈值: uint,

另一个对象：Object,

另一个点： Point,

另一个透明阈值: uint);

注意到这 5 个参数被分成了两组：一个和另一个。每组都需要一个点来确定位图的左上角起点。这是因为每个位图所在的坐标系可能不一样，有的在别的元件里，或者元件的元件里。给定点只是为了统一坐标系，如果必要，点可以通过 `DisplayObject.localToGlobal` 来统一。例子中两个位图都正好在场景上，所以可以直接使用各自的相对位置。

接着每组都还有一个透明阈值。之前提到过，支持透明的位图对象，每个像素的透明度取值范围在 0（完全透明）到 255（完全不透明）。参数透明阈值即指定，透明度在多少的时候就算碰撞。例子中两个都是 255，意思就是说，如果两个位图中的像素有碰撞，则必须是完全不透明的。等下还有一个阈值比较小的例子。

最后还剩一个参数，另一个对象。注意它的类型是 Object。允许传递的可以是一个点 (Point)，一个矩形 (Rectangle) 或者另一个位图对象 (BitmapData)。如果传一个点或者一个矩形，那么后面的两个参数可以忽略。检测和点的碰撞一般是用于鼠标是否接触位图，例子：

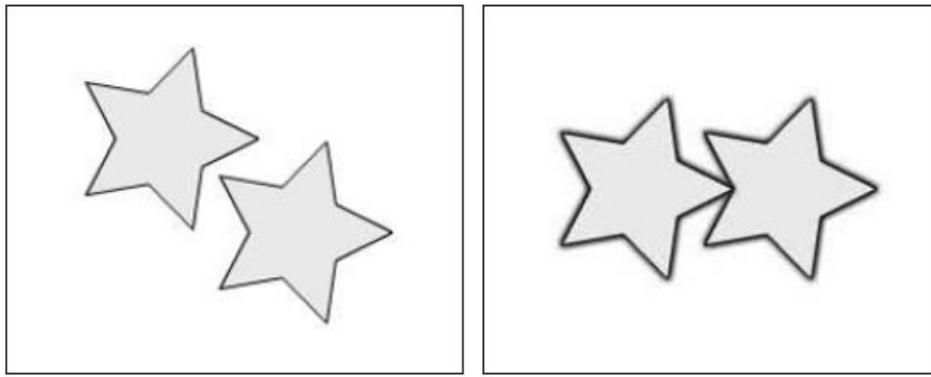
```
if(myBitmapData.hitTest(new Point(myBitmapData.x, myBitmapData.y),  
255, new Point(mouseX, mouseY)))
```

```
{  
    // 鼠标碰到位图啦  
}
```

检测和矩形的碰撞，我想不出一个实际的例子，不过知道是可以的就行了，视具体情况而用之吧。

而上面的例子中，是传递了另一个位图对象，所以后面两个参数，点和透明度阈值，也得一并给出。

最后是对碰撞的处理，如果碰撞了两个星星就加上默认的发光滤镜，发红光，否则就取消滤镜。结果如图



**Figure 1-3.** Stars are not touching. **Figure 1-4.** And now they are.

半透明图形的碰撞检测

在之前的例子中，创建的一个星星绘制在两个位图中是完全不透明的。因此设置的透明阈值都为 255。（实际上可以是任何大于 0 的值）现在让我们稍作改变，看看如何检测带透明的图形之间的碰撞。

```

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

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

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

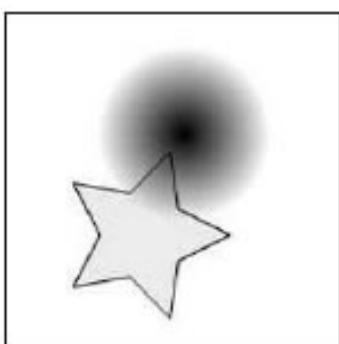
        // make a star
        var star:Star = new Star(50);

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

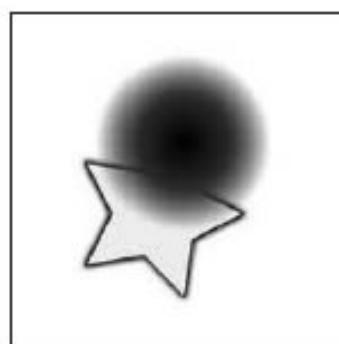
        // make a fixed bitmap, draw the star into it
    }
}

```

一开始创建了一个命名为 circle 的 Sprite，并采用放射型渐变填充。接着将 bmpd2 本来是绘制 star 的换成 circle。然后测试的话会发现一直没有碰撞，除非小心翼翼的让圆中心点接触到星星。这是因为放射型渐变填充造成只有圆的中心才是完全不透明的。



**Figure 1-5.** The star is touching the circle, but not a pixel that has the required alpha threshold.



**Figure 1-6.** Only the center of the circle has an alpha of 255, so you get a hit.

那么把透明阈值调低一点试试：

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

现在移动圆到星星上，要使碰撞发生，接触的像素的透明值至少要 128。可以试着改变第二个透明阈值看看不同的效果。注意，如果设置为 0，碰撞会发生在圆还没有接触到星星的时候，这是因为星星接触到了位图上完全透明的像素。记住位图本身是一个矩形，只不过透明了看不到而已。同时注意，由于绘制星星的位图，像素值要么完全透明，要么完全不透明，所以改变第一个透明阈值（除了 0 以外）不会有影响。

### BitmapData.hitTest 用于非位图

到此，测试都是围绕着 Bitmap 对象。而大多情况下的 MovieClip，Sprite 和 Shape 对象是没法用 hitTest 的。有个办法就是暗中准备一些 BitmapData，不把它们加入到显示列表。当要对两个显示对象进行碰撞检测时，先把它们分别绘制到一个准备好的位图对象中，然后如法炮制。要知道，这不是唯一的办法，却是个不错的办法。

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

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

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

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

            star2 = new Star(50);
            star2.x = 200;
            star2.y = 200;
            addChild(star2);

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

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

        private function onMouseMoving(event:MouseEvent):void
        {
            // move star1 to the mouse position
        }
    }
}
```

```
star1.x = mouseX;
star1.y = mouseY;

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

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

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

这次，在构造函数内创建了两个 BitmapData 对象和两个星星，两个 BitmapData 对象没有放入 Bitmap 中，也就是没有加入到显示列表里，而是直接加入了两个星星。名为 star1 的星星跟随鼠标移动。每当鼠标移动时，两个位图都用 fillRect 清空。结果就是两张完全透明的位图，然后分别 绘上对应的星星：

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

绘制时采用了星星的位置作为矩阵(matrix)的偏移量。现在测试：

if(bmpd1.hitTest(new Point(), 255, bmpd2, new Point(), 255))  
因为 BitmapData 不在显示列表里，甚至都没在 Bitmap 中，加上两个星星又是同一坐标系，且被绘制在各自的相对位置上，所以不用去考虑任何坐标关系。只需传入默认情况下 x 和 y 都为 0 的 Point 即可。而透明值设为 255 意味着两个星星不透明。

这里只是提供的一些用 `BitmapData.hitTest` 对非圆形、矩形、点形图形对象之间的碰撞检测的例子。我相信在你掌握了某如何运作之后，会创想出更多美妙的变化。

于。我确信往您掌握了其如何运作之后，会创造接下來，让我们聚焦如何处理大量的碰撞检测

接下来，让我们  
对照对象的碰撞检测

Flash Player 10 中的 AS 运行效率比以往任何版本都高，这就允许我们一次性可以做更多的事情。但限制仍然存在，如果在场景中有一大堆对象在移动，卡是早晚会发生。大量对象的碰撞检测更使这个问题严重化，因为每个对象都要和其它各各对象进行反复检测。这种情况不局限于碰撞检测，任何粒子系统或者需要大量对象交互的游戏 中都会有这类问题，如重力，以及会在第二章见到的成群而行(flocking)。

如果在 6 个对象之间进行碰撞检测，先不考虑重力影响以及其它作用。粗算一下会发现要 6 个 6 次，也就是 36 次比较，不过上本书里讲过，实际次数应该差不多于一半，准确的讲是 15 次。假设 6 个对象：A、B、C、D、E、F，比较情况如下：

假设 0 个对象: A, B, C,

BC, BD, BE, BF

CD, CE, CF

DE, DF

EF

注意看，B 行里没有 BA，因为在 A 行里有 AB 了，说明 B 已经和 A 检测过了。一步步到了 E，和 F 检测完后就算完工，此时 F 已经和其它都检测过了。下面的公式算出检测的次数，N 代表对象的个数：

$$(N^2-N)/2$$

6 个对象就是  $(36-6)/2$  等于 15。

10 个对象就是  $(100-10)/2$  等于 45。

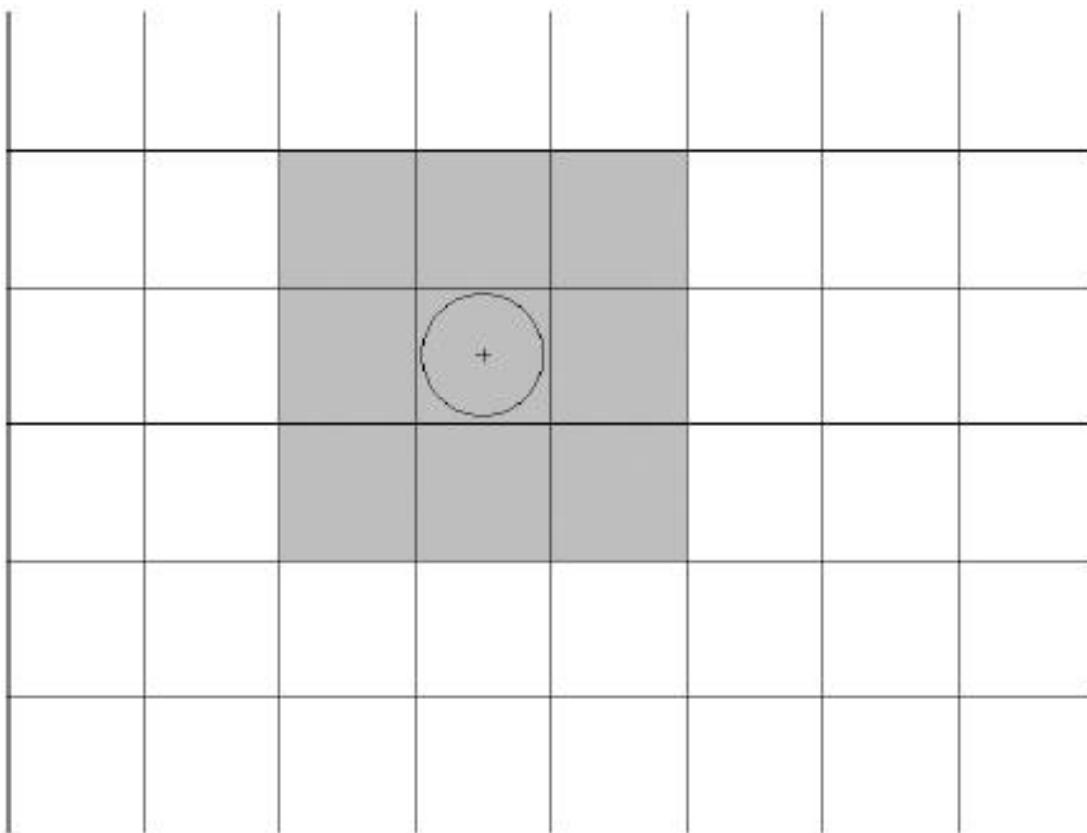
如果 20 个对象就有 190 次检测，30 个就要 435 次啊！

增长速度那是相当快。

要是有 100 个对象的话 AS3 也跑不动，检测就要 4950 次。如果使用距离检测法，就要计算 4950 次距离，如果使用位图检测法，就要对位图进行 4950 次的清空和绘制，再调用 4950 次 hitTest。这些都要在一帧里面执行掉。swf 就是被这么拖垮的。

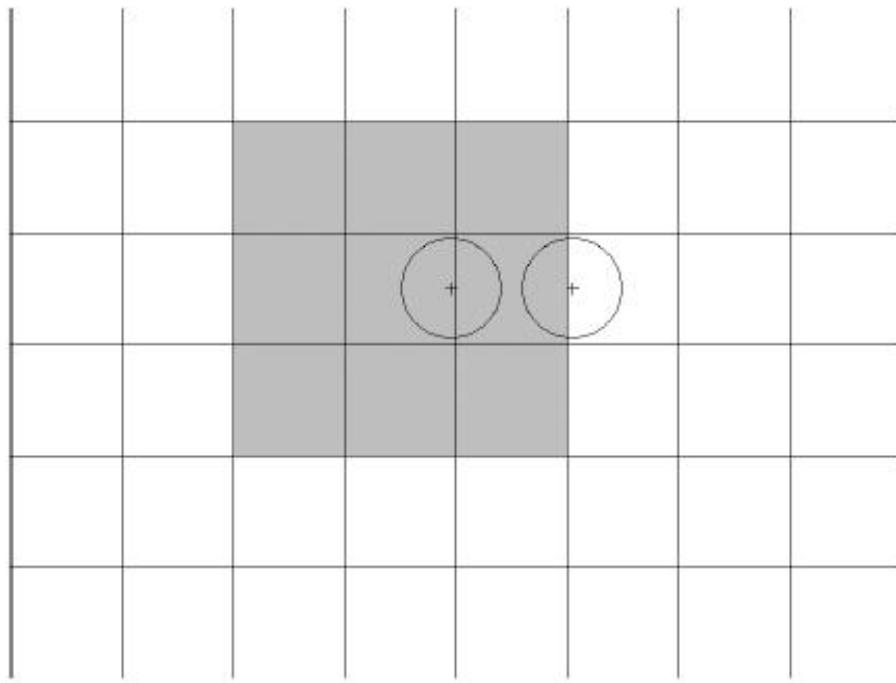
所以必须为此做点限制，幸运的是，这并非办不到。考虑一下当两个相对较小的对象分别处在场景的两边，这时有必要为它们去做碰撞检测吗？但要发现这种情况，需要计算它们之间的距离...似乎又绕了回去。有没有别的办法呢？

假设我们把场景划分成若干个小方格，每个小方格的大小至少能容下最大的那个对象，接着把对象居中放置在一个小方格内。照此放置后，该对象可能会产生碰撞的区域，就仅限于周围的 8 个小方格里了。



**Figure 1-7. The ball can collide only with objects in the shaded cells.**

球被安置在一个格子中，能碰到球的对象只能是在球周围的 8 个画了阴影的格子内。即使有两个对象都非常贴边，但它们之间仍不会有任何接触。



**Figure 1-8.** There's no way the two balls can collide.

再次提醒，使用这种方法的前提条件是格子大小至少得能容下对象中最大的那个。如果有球比格子大，上述方法就无效了。

需要知道的是，这种处理方式并不一定是最好的，但它确实完成了对碰撞检测次数的有效限制，不必对所有对象都进行两两之间的检测。

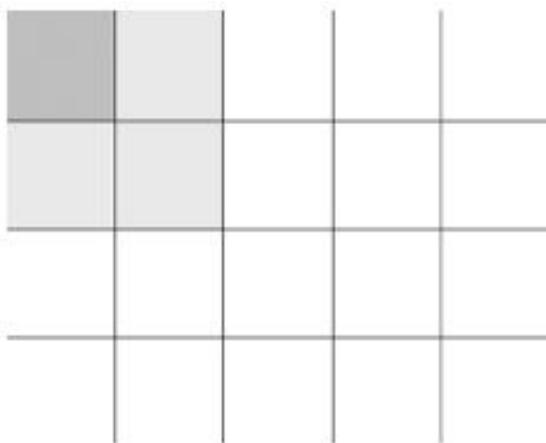
接下来具体讨论如何实现这个方法。

#### 实现基于网格的碰撞检测

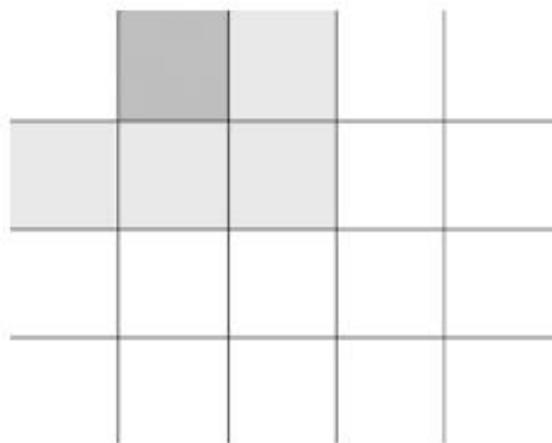
把左上角作为第一步。减少些网格数让看起来方便点。图 1-9

在对第一个格子内的所有对象和其它对象进行检测时，不用想，由于在左上角，所以只考虑右边，下边和右下角 3 个格子。再说一遍，任何处在白色格子内的对象是无法碰到第一个格子里的对象的。

好了，接着第二步，向右移一格。图 1-10



**Figure 1-9.** Test all the objects in the first cell with all the objects in the surrounding cells.



**Figure 1-10.** Continuing with the next cell

考虑此时周围一圈的格子。记得在之前的检测中，已经包含了对左边也就是第一个格子的检测，所以这次左边的格子可以不算在内。

第三步，继续向右，检测当前的格子，同理可以忽略掉左边的格子。

最后达到场景右侧，同理无视左边，那么只有下边和左下角 2 个格子要检测。

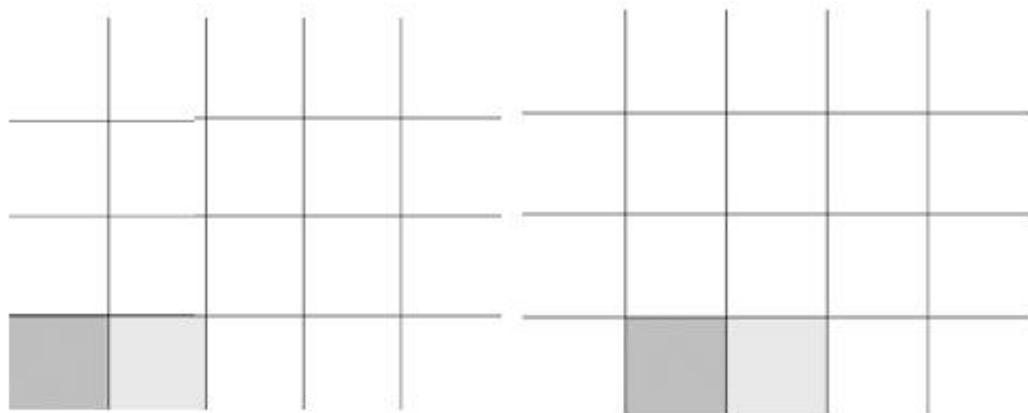
现在开始第二行。



**Figure 1-14. Starting the second row**

**Figure 1-15. Next column in second row**

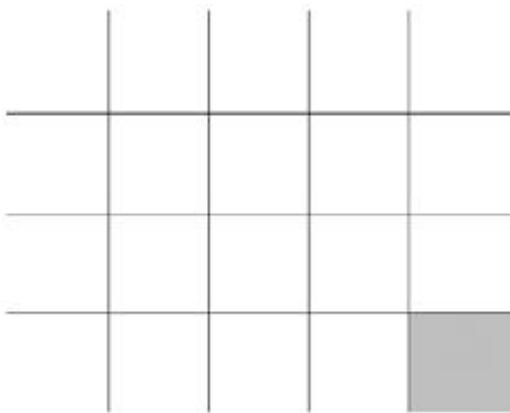
围绕的一组格子中，上方也就是第一行在之前都检测过了，所以忽略掉。而剩余过程又和第一行类似。反复下去直到最后一行。



**Figure 1-16. The last row**

**Figure 1-17. Second column, last row**

到底了，下方已无需检测，而上方又都检测过了，所以只剩下右边了。当最后一步时，则不用做任何检测了。



**Figure 1-18. Nothing to do here**

好了，这就是我们要做的，接下来看看该怎么做。实际上，最多一次也就关心过 5 个格子：自身一个，右边一个，下方三个。如果把一个格子看成一个数组，那么就有 cell0, cell1, cell2, cell3, cell4 (原文还有 cell5，估计是作者写错-\_-)。方便起见，假设格子内的对象的类型都是 Ball (球)。

先看看第一个格子 cell0，这个格子里所有的球，都可能会发生接触。因此用二重循环来检测。下面是一段粗糙的演示代码：

```
for(var i: int=0; i < cell0.length-1; i++)  
{  
    var ballA: Ball = cell0[ i ] as Ball;  
    for(var j: int=i+1; j < cell0.length; j++)
```

```

    {
        var ballB: Ball = cell0[j] as Ball;
        // 进行碰撞检测
    }
}

这段检测中，每个球不会和自身进行检测，球之间也不会重复检测。接下来是 cell0 和 cell1 之间的检测。有点不同，不同之处在于要拿 cell0 中的球一个一个和 cell1 中的球做检测。
for(var i: int=0; i < cell0.length; i++)
{
    var ballA: Ball = cell0[i] as Ball;
    for(var j: int=0; j < cell1.length; j++)
    {
        var ballB: Ball = cell1[j] as Ball;
        // 进行碰撞检测
    }
}

```

注意到两段循环都是从头到底，不像之前的循环为了避免重复检测而做了些手脚。其实也不用管这些，只要知道确实对所有的元素进行了检测即可。

接着完成 cell0 和 cell2, cell3 和 cell4 之间的检测。到此，第一步任务完成。接着第二步，此时的 cell0 是第二个格子……

现在如果你觉得晦涩难懂，又看不出效率是否真的有所提升。那我们来做道数学题。记得之前 100 个对象的检测次数吗？4950 次。而刚才讨论的方法有多少次检测呢？这和场景大小，对象大小，对象数量，方格大小以及对象位置的分布密切相关。在我的测试中，100 个对象实际发生的碰撞检测平均次数在 100 到 200 次之间。算一下吧，差不多节省了 4800 次！由于碰撞检测需要用到开方，这是一向很花时间的数学运算，节省了检测次数也就大大节省了 CPU 时间。

当然啦，以对象的位置创建和更新网格，以及遍历 cell 完成那些检测步骤，是每帧都要做的事情。在拥有大量对象的情况下，减少不必要的计算次数是节省开销的关键。但如果只有少数几个对象的话，直接两两之间进行检测则更有效。章节的后半部分会讨论合适地使用这两种情况。

### 编写网格代码

此段一开始会展示结构清晰的代码，然后会把其中用到的每个函数单独拿出来分析。接着给出完整的可重用的类。

在此之前需要用到一些碰撞的对象：Ball 类

```

package
{
    import flash.display.Sprite;
    public class Ball extends Sprite
    {
        private var _color:uint;
        private var _radius:Number;
        private var _vx:Number = 0;
        private var _vy:Number = 0;

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

        private function draw():void
        {
            // draw a circle with a dot in the center
        }
    }
}

```

```

graphics.clear();
graphics.lineStyle(0);
graphics.beginFill(_color, 1);
graphics.drawCircle(0, 0, _radius);
graphics.endFill();
graphics.drawCircle(0, 0, 1);
}
public function update():void
{
// add velocity to position
x += _vx;
y += _vy;
}
public function set color(value:uint):void
{
_color = value;
draw();
}
public function get color():uint
{
return _color;
}
public function set radius(value:Number):void
{
_radius = value;
draw();
}
public function get radius():Number
{
return _radius;
}
public function set vx(value:Number):void
{
_vx = value;
}
public function get vx():Number
{
return _vx;
}
public function set vy(value:Number):void
{
_vy = value;
}
public function get vy():Number
{
return _vy;
}
}

}

}

```

此处没有什么高科技，就是通过给定半径和颜色画一个圆。每个圆都含有 x 和 y 方向上的速度作为其属性，还有一个 update 函数用来更新速度改变后的位置。就这些。主角即将登场。

package{

```

import flash.display.Sprite;
import flash.display.StageAlign;
import flash.display.StageScaleMode;
public class GridCollision extends Sprite
{
    private const GRID_SIZE: Number = 50;
    private const RADIUS: Number = 25;
    private var _balls: Array;
    private var _grid: Array;
    private var _numBalls: int = 100;
    private var _numChecks: int = 0;
    public function GridCollision()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;
        makeBalls();
        makeGrid();
        drawGrid();
        assignBallsToGrid();
        checkGrid();
    }
    // 剩余的函数等下一一介绍
}

```

程序最初定义了两个常量表示网格的大小和球的半径。记得之前讲过网格大小一定不能小于最大的对象大小吗？

所以网格两倍于半径，满足这个要求。然后声明了两个数组，一个用来保存球(\_balls)，一个用来保存格子(\_grid)。还有包括球的数量(\_numBalls)和碰撞检测次数(\_numChecks)这两个变量。

接着调用了一连串函数：创建球，创建网格，还有画出网格，把球分配进格子，以及网格检测，最后输出碰撞检测的次数。

那么，就从第一个创建球的函数开始

```

private function makeBalls(): void
{
    _balls = new Array();
    for(var i: int=0; i < _numBalls; i++)
    {
        // 创建出一个球，然后把它加入显示列表以及数组
        var ball: Ball = new Ball(RADIUS);
        ball.x = Math.random() * stage.stageWidth;
        ball.y = Math.random() * stage.stageHeight;
        addChild(ball);
        _balls.push(ball);
    }
}

```

这段函数最简单，就是跑个循环创建出在场景上位置随机的球，然后加到显示列表再推入数组。

接下来是创建网格

```

private function makeGrid(): void
{
    _grid = new Array();
    // 场景宽度 / 格子大小 = 网格列数

```

```

for(var i: int=0; i < stage.stageWidth / GRID_SIZE; i++)
{
    _grid[ i ] = new Array();
    // 场景高度 / 格子大小 = 网格行数
    for(var j: int=0; j < stage.stageHeight / GRID_SIZE; j++)
    {
        _grid[ i ][j] = new Array();
    }
}

```

这段函数创建了一个二维数组，数组中的每个元素代表了场景被划分成的网格中的一格。元素的类型还是一个数组（虽然可以叫它三维数组，不过这个概念不太适合用在这里）用来保存一格中被分配到的对象。

下面这个画网格函数，纯粹是辅助用的。实际开发中一般用不到。

```

private function drawGrid(): void
{
    // 画出行列线
    graphics.lineStyle(0, .5);
    for(var i: int=0; i <= stage.stageWidth; i+=GRID_SIZE)
    {
        graphics.moveTo(i, 0);
        graphics.lineTo(i, stage.stageHeight);
    }
    for(i=0; i <= stage.stageHeight; i+=GRID_SIZE)
    {
        graphics.moveTo(0, i);
        graphics.lineTo(stage.stageWidth, i);
    }
}

```

下面这个函数比较重要，把球分配进格子里

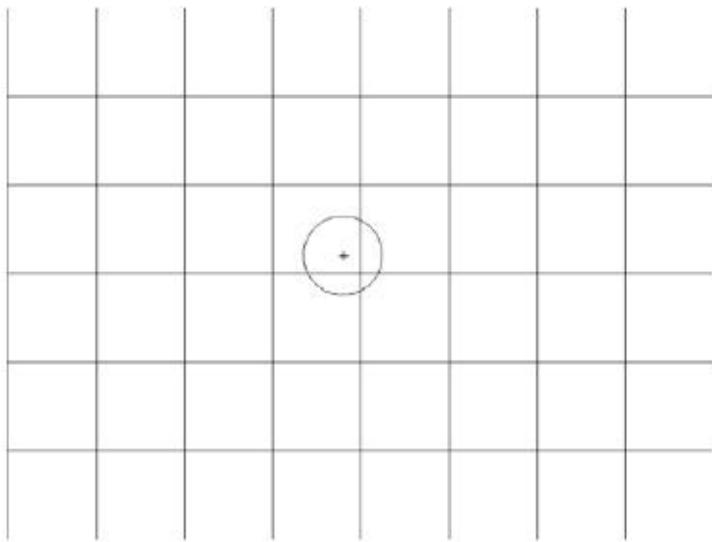
```

private function assignBallsToGrid(): void
{
    for(var i: int=0; i < _numBalls; i++)
    {
        // 球的位置除以格子大小，得到该球所在网格的行列数
        var ball: Ball = _balls[ i ] as Ball;
        var xpos: int = Math.floor(ball.x / GRID_SIZE);
        var ypos: int = Math.floor(ball.y / GRID_SIZE);
        _grid[ xpos ][ ypos ].push(ball);
    }
}

```

这里要做下详细说明。前两行循环和创建临时变量很明显，无需解释。

后面通过球的 x 坐标除以网格大小得到一个数，再去掉小数部分所保留的整数就是球所在网格的列数。



**Figure 1-19.** Figuring out which grid element a ball belongs to

同理 y 坐标可得到行数。

举个例子，假设一格大小是 100x100。球的坐标在 (380, 280)。

拿 380 除以 100 得到 3.8，去掉小数部分后等于 3，这就是说球过了第 3 列，在第 4 列中。

记住数组的下标是从 0 开始的，索引 3 正好是第 4 个元素。同理 y 坐标，得到球在第 3 行。

结合图上看，发现球的中心点确实落在第 3 行第 4 列。

回来看代码，最后一行做的事情就是根据 xpos 和 ypos 得到二维数组中对应的格子，记得格子的类型是个数组，那么把球推入数组，也就意味着把球分配进这个格子。

当跑完整个循环后，所有的球都有了各自的归属。有的格子中可能含有一个球，有的可能有多个，也可能一个也没有。

现在是开始碰撞检测的时候了。

网格检测函数任务重大，实际上，它是由几个函数合力完成的。

```
private function checkGrid(): void
{
    for(var i: int=0; i < _grid.length; i++)
    {
        for(var j: int=0; j < _grid[ i ].length; j++)
        {
            // 检测第一个格子内的对象间是否发生碰撞
            checkOneCell(i, j);
            checkTwoCells(i, j, i+1, j); // 右边的格子
            checkTwoCells(i, j, i-1, j+1); // 左下角的格子
            checkTwoCells(i, j, i, j+1); // 下边的格子
            checkTwoCells(i, j, i+1, j+1); // 右下角的格子
        }
    }
}
```

二重循环用来遍历所有的网格。第一个 checkOneCell 函数是对当前格子内所有对象进行检测。

```
private function checkOneCell(x: int, y: int): void
{
    // 检测当前格子内所有的对象
    var cell:Array = _grid[ x ][ y ] as Array;
    for(var i: int=0; i < cell.length-1; i++)
    {
        var ballA: Ball = cell[ i ] as Ball;
        for(var j: int=i+1; j < cell.length; j++)
        {
            var ballB: Ball = cell[ j ] as Ball;
            if(ballA != null && ballB != null)
            {
                if(ballA.x == ballB.x && ballA.y == ballB.y)
                {
                    ballA.isCollided = true;
                    ballB.isCollided = true;
                }
                else
                {
                    if(ballA.x > ballB.x && ballA.y > ballB.y)
                    {
                        ballA.isCollided = true;
                        ballB.isCollided = true;
                    }
                    else
                    {
                        ballB.isCollided = true;
                        ballA.isCollided = true;
                    }
                }
            }
        }
    }
}
```

```

    {
        var ballB: Ball = cell[ j ] as Ball;
        checkCollision(ballA, ballB);
    }
}
}
}

```

这段代码之前已经有过示例了，具体检测由函数 checkCollision 负责。

然后是调用 checkTwoCells 函数 4 次。

```

checkTwoCells(i, j, i+1, j); // 右边的格子
checkTwoCells(i, j, i-1, j+1); // 左下角的格子
checkTwoCells(i, j, i, j+1); // 下边的格子
checkTwoCells(i, j, i+1, j+1); // 右下角的格子

```

i 和 j 是当前格子的索引。i-1, i+1, j-1, j+1 分别对应上下左右 4 个方向了。

搭配使用就可以遍历需要检测的 4 个格子。

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

```

{
    // 确保要检测的格子存在
    if(x2 < 0) return;
    if(x2 >= _grid.length) return;
    if(y2 >= _grid[ x2 ].length) return;
    var cell0:Array = _grid[ x1 ][ y1 ] as Array;
    var cell1:Array = _grid[ x2 ][ y2 ] as Array;
    // 检测当前格子和邻接格子内所有的对象
    for(var i: int=0; i < cell0.length; i++)
    {
        var ballA: Ball = cell0[ i ] as Ball;
        for(var j: int=0; j < cell1.length; j++)
        {
            var ballB: Ball = cell1[ j ] as Ball;
            checkCollision(ballA, ballB);
        }
    }
}
```

x1, y1 和 x2, y2 是分别取得两个格子的索引。一开始的 3 行是判别索引是否出界。

当 x2 小于 0 或者大于 \_grid.length 时，当前格子位于场景的两侧，情况就如图 1-9, 1-13。

类似的，当 y2 大于 \_grid[x2].length 时，当前格子就处于场景的底部，情况就如图 1-16, 1-17, 1-18。

如果发现出界就离开函数。接下来的二重循环，取出对象逐个比较在之前也提到过了。

最后一个重点就是碰撞检测。

```
private function checkCollision(ballA: Ball, ballB: Ball):void
```

```

{
    // 判断距离的碰撞检测
    _numChecks++;
    var dx: Number = ballB.x - ballA.x;
    var dy: Number = ballB.y - ballA.y;
    var dist: Number = Math.sqrt(dx * dx + dy * dy);
    if(dist < ballA.radius + ballB.radius)
    {
        ballA.color = 0xff0000;
        ballB.color = 0xff0000;
    }
}
```

}

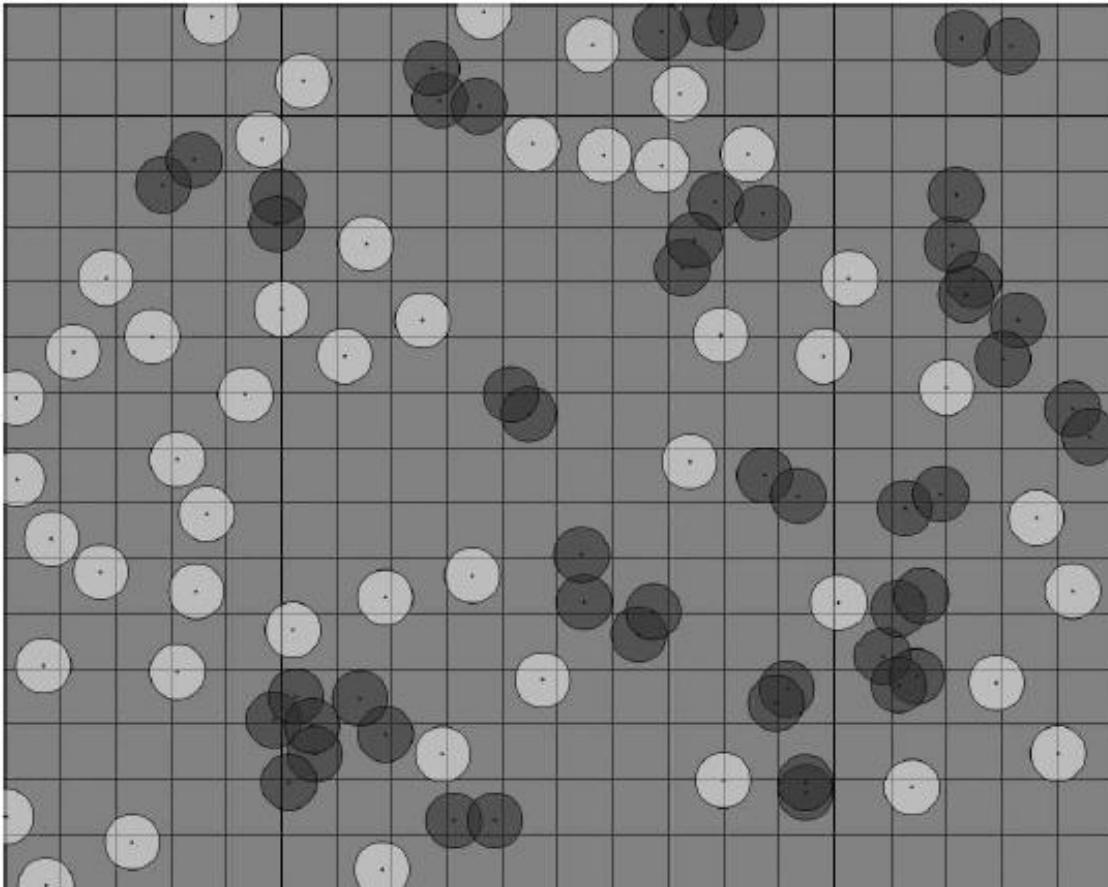
函数执行到此才算是真正的进行了一次碰撞检测，所以要累加一下 `_numChecks` 来统计检测的次数。

然后通过距离判断两个球是否碰撞，如果碰了就把它们改变成红色。

这里实际上可以通过半径和的平方来取消开方运算以达到优化效果。

呼呼，经历了一大堆实际工作后，终于避免了最后一个函数可能多达几千次的执行。

如果一切正常，那么结果看起来就好像这样：



**Figure 1-20.** One hundred objects, successfully hit-tested

有接触的球都是红色，没有的即为白色。如果出现意外，那么回头看看程序有没有问题。

因为碰撞检测中最重要的部分就是，精确度。

### 测试并调整网格

每次运行该测试程序，都会得到碰撞检测的统计次数。在我的电脑 1440x900 的分辨率，浏览器最大化的情况下，次数在 80 到 130 之间。场景越小次数越多。可以多运行几次找找数字的跨度范围。

因为在这样的环境（网格大小 50，100 个半径为 25 的球）不变的情况下，能改变次数的只有那些被随机分布的对象。

现在尝试加大网格的大小，比如 100。会发现次数一下子上去了。这是因为一个格子里能放下更多的球了。

感觉不太爽，但同时需要遍历的格子数量下来了，到底哪个更有益呢，马上为您揭晓。

现在，先试着缩小格子，比如 40，更甚之到 30。次数少咯，但看看结果。

发现检测出现失误，有些对象明明接触了却没有标示出来（还是白色的）。

这可不行，所以还得强调一下格子的大小至少要能放下最大的那个对象。把大小设回 50，然后改变每个球的半径如下所示：

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

这个改变倒是没有影响到次数。由此看出网格大小的规则适用于对象大小不定的情况。

现在做进一步测试，看看执行效率到底如何。就拿“每个对象两两之间进行检测”这个方法作个比较。

为此，需要先定一个函数，就叫它庶民检测好了。

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

这段代码在 100 个对象时会产生 4950 次检测。这数字早被网格检测完败过了，不过当时的网格检测是轻装上阵，

还有些必要工作没有算在里面。要是算上的话，实际情况会如何呢？

那当然还是会胜出咯，不然这章甚至我还在这儿混什么呢？

马上证明给你们看：让庶民检测和网格检测各跑 10 次，看谁用的时间少。

下面的函数改编自之前的构造函数

```
public function GridCollision()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    makeBalls();
    drawGrid();
    var startTime:int;
    var elapsed:int;
    var i:int;
    startTime = getTimer();
    for(i = 0; i < 10; i++)
    {
        makeGrid();
        assignBallsToGrid();
        checkGrid();
    }
    elapsed = getTimer() - startTime;
    trace("Grid-based:", elapsed);
    startTime = getTimer();
    for(i = 0; i < 10; i++)
    {
        basicCheck();
    }
    elapsed = getTimer() - startTime;
    trace("Basic check", elapsed);
}
```

为了测试，加入了两个时间变量。计算运行两个方法所消耗的时间。网格检测是

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

这个三个函数，庶民检测就一个 basicCheck() 函数。

结果如何？我这里的环境是，网格检测法比庶民检测法快 2.5 倍。有点小失望，想想之前说的减少了 4800 次检测哪。

不过速度有 2.5 倍的提升也不错，不是吗？

对象越多，差别越大。如果把对象数量加到 1000，网格检测完成一次在 1 秒以内，而庶民检测要花 13 秒以上。

当然，每秒一帧的帧频也是不可忍受的，但肯定要好过 13 秒 1 帧。另外，如果减少成 50 个球，基本测试反而更快。

所以这里就有一个分界点，过了分界点就要切换合适的方法。至于分界点在哪里，需要针对具体的程序做测试。

把 `_numBalls` 设置回 100，然后运行几次看看花多久。在我这里 10 次下来平均 55 毫秒。然后把网格大小从 50 改成 75，平均值降到了 37 毫秒。我还发现网格大小在 85 到 100 之间时，平均值差不多是 32 毫秒，这样就比庶民检测快了 4 倍！

注意这里提到的所有数字都可视为基准。

如果各位得到的结果大相径庭，那么最可能的情况就是没有根据自己的程序设定环境选择好网格的大小。

有几个关键的变量列在这里：场景大小，对象数量，对象大小，网格大小，还有碰撞检测的实现算法。

这里无法给出一个公式求得这些变量的最佳值。

## 整理成类

通过以上对 `GridCollision` 类的分析，证明了基于网格的碰撞检测是有价值的，也希望能在理解上有所帮助。

只是还有几个问题。首先，类是一个文档类(document class)，如果要在别的程序中使用它，只能复制粘贴。

同时它和 `Ball` 还是紧耦合关系，要是用别的对象，就得改变所有引用的部分。

另外，碰撞检测的算法也是写死的，不能用 `hitTestObject` 或者位图检测代替基于距离的检测。

为此，创建一个可复用类，避免这些问题就是接下来要做的工作。这个类，我们叫它 `CollisionGrid`。

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

    public class CollisionGrid extends EventDispatcher
    {
        private var _checks:Vector.<DisplayObject>;
        private var _grid:Vector.<Vector.<DisplayObject>>;
        private var _gridSize:Number;
        private var _height:Number;
        private var _numCells:int;
        private var _numCols:int;
        private var _numRows:int;
        private var _width:Number;

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

```

        _numRows = Math.ceil(_height / _gridSize);
        _numCells = _numCols * _numRows;
    }
    public function drawGrid(graphics:Graphics):void
    {
        graphics.lineStyle(0, .5);
        for(var i:int = 0; i <= _width; i += _gridSize)
        {
            graphics.moveTo(i, 0);
            graphics.lineTo(i, _height);
        }
        for(i = 0; i <= _height; i += _gridSize)
        {
            graphics.moveTo(0, i);
            graphics.lineTo(_width, i);
        }
    }
    public function check(objects:Vector.<DisplayObject>):void
    {
        var numObjects:int = objects.length;
        _grid = new Vector.<Vector.<DisplayObject>>(_numCells);
        _checks = new Vector.<DisplayObject>();
        for(var i:int = 0; i < numObjects; i++)
        {
            var obj:DisplayObject = objects[i];
            var index:int=Math.floor(obj.y / _gridSize)*_numCols+Math.floor(obj.x / _gridSize);
            if(_grid[index] == null) _grid[index] = new Vector.<DisplayObject>;
            _grid[index].push(obj);
        }

        checkGrid();
    }
    private function checkGrid():void
    {
        for(var i:int = 0; i < _numCols; i++)
        {
            for(var j:int = 0; j < _numRows; j++)
            {
                checkOneCell(i, j);
                checkTwoCells(i, j, i + 1, j);
                checkTwoCells(i, j, i - 1, j + 1);
                checkTwoCells(i, j, i, j + 1);
                checkTwoCells(i, j, i + 1, j + 1);
            }
        }
    }
    private function checkOneCell(x:int, y:int):void
    {
        var cell:Vector.<DisplayObject> = _grid[y * _numCols + x];
        if(cell == null) return;

        var cellLength:int = cell.length;

```

```

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

    private function checkTwoCells(x1:int, y1:int, x2:int, y2:int):void
    {
        if(x2 >= _numCols || x2 < 0 || y2 >= _numRows) return;
        var cellA:Vector.<DisplayObject> = _grid[y1 * _numCols + x1];
        var cellB:Vector.<DisplayObject> = _grid[y2 * _numCols + x2];
        if(cellA == null || cellB == null) return;

        var cellALength:int = cellA.length;
        var cellBLength:int = cellB.length;
        for(var i:int = 0; i < cellALength; i++)
        {
            var objA:DisplayObject = cellA[i];
            for(var j:int = 0; j < cellBLength; j++)
            {
                var objB:DisplayObject = cellB[j];
                _checks.push(objA, objB);
            }
        }
    }

    public function get checks():Vector.<DisplayObject>
    {
        return _checks;
    }
}
}

```

类的大部分代码和之前都很相似。有很大一部分还做了优化，尤其是 Vector 的使用。Vector 是 Flash10 的新内容，相当于一个具有类型的数组。由于编译器知道 vector 中的每个元素都是同样的类型，所以就能为之创建出更有效的字节码，从而提高执行效率。

以这段程序为例，从数组改为 vector 差不多使运行效率翻了一倍。

drawGrid 函数一点没变，它依旧用来画出网格。

check 函数是这个类对外交互的主要函数。其接收参数的类型是元素类型为 DisplayObject 的 vector。

选择 Displayobject 的原因是由于碰撞检测通常用于 Sprite, MovieClip, Shape 和 Bitmap，而这些类都继承自 Displayobject。Displayobject 也有 x 和 y 两个位置属性。

所以要用自定义的对象时，请确保继承自 Displayobject。

函数一开始定义了一个名为 \_grid 的 vector，还有一个名为 \_checks 的 vector。

\_grid 应该不陌生，但在实现上有点不同，这里用一维 vector 加索引技巧取代了二维数组。

因为这么做，可以使访问元素速度更快并减少了循环。等下会有详细介绍。

\_checks 用来保存需要进行碰撞检测的对象。注意 CollisionGrid 类不处理具体的碰撞检测，它只用来创建网格，分配对象，以及生成一组需要被检测的对象。具体的碰撞检测算法由你而定。

接着，check 函数对给定的 vector 进行遍历，把其中每个 Displayobject 都分配进网格。

代码片段如下：

```
for(var i: int = 0; i < numObjects; i++)
{
    var obj: DisplayObject = objects[ i ];
    // 在一维网格中用一个索引描述位置
    var index: int = Math.floor(obj.y / _gridSize) * _numCols +
        Math.floor(obj.x / _gridSize);
    // 该位置的网格只需创建一次
    if(_grid[ index ] == null)
    {
        _grid[ index ] = new Vector.< DisplayObject >;
    }
    _grid[ index ].push(obj);
}
```

再次声明，这里使用一维代替二维。变量 index 解决了对应的行列号。y, x 分别代表行和列，基本公式如下：

$$\text{index} = \text{y} * \text{numColumns} + \text{x}$$

看图更能说明问题

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

**Figure 1-21. Using a single flat array as a grid**

图上的网格有 5 行 5 列。对象在第 2 行第 3 列，也就是 y=2, x=3。因此 index 就等于  $2 * 5 + 3$ ，即 13。

对照图，完全正确。

代码中，得到行列号的方式没变，依旧是：

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

只是这些代码被有技巧的整合成了一句：

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

接下来的程序是另一个优化技巧。避免循环创建出不必要的网格，只根据需要，判断是否创建网格。

这种方式叫作懒惰实例化（推迟实例化）。懒惰在这里并非一个贬义词，其含义指，拖延创建内容的行为，

只有在需要的时候才执行。这么做有时很有意义，有时则不然。在此，不失为一个妙计：

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

```

}

当来到循环的最后一行时，所做的事情就众人皆知了：
_grid[ index ].push(obj);
函数在最后调用了 checkGrid:
private function checkGrid(): void
{
    for(var i: int = 0; i < _numCols; i++)
    {
        for(var j: int = 0; j < _numRows; j++)
        {
            checkOneCell(i, j);
            checkTwoCells(i, j, i + 1, j);
            checkTwoCells(i, j, i - 1, j + 1);
            checkTwoCells(i, j, i, j + 1);
            checkTwoCells(i, j, i + 1, j + 1);
        }
    }
}

```

这个函数没什么变化，只是在这里重复一下。

checkOneCell 和 checkTwoCells 两个函数本质上没有变化。具体实现则要注意，由于用一维 vector 替换了二维数组，且网格的创建是根据需要来的，所以先要检测网格是否存在。

同时将碰撞检测的函数，换成把两个对象推入\_checks。

做完这些，\_checks 就包含了所有需要进行碰撞检测的对象。

该列表在检测时是两步一增加的（元素 0 和元素 1 比，元素 2 和元素 3 比，元素 4 和元素 5 比，等等）。

最后，还提供一个对外接口 checks，用来访问这个列表。

## 使用此类

OK，有了这么可爱的一个类，那就做点什么吧。改版之前的 GridCollision，叫它 GridCollision2

```

package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.utils.getTimer;
    import flash.display.DisplayObject;

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

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

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

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

```

```

makeBalls();

var startTime:int;
var elapsed:int;

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

```

```

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

```

```

private function updateBalls():void
{
    for(var i:int = 0; i < _numBalls; i++)
    {
        var ball:Ball = _balls[i] as Ball;
        trace(ball);
        ball.update();
        if(ball.x < RADIUS)
        {
            ball.x = RADIUS;
            ball.vx *= -1;
        }
        else if(ball.x > stage.stageWidth - RADIUS)
        {
            ball.x = stage.stageWidth - RADIUS;
            ball.vx *= -1;
        }
    }
}

```

```

if(ball.y < RADIUS)
{
    ball.y = RADIUS;
    ball.vy *= -1;
}
else if(ball.y > stage.stageHeight - RADIUS)
{
    ball.y = stage.stageHeight - RADIUS;
    ball.vy *= -1;
}
ball.color = 0xffffffff;
}

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

```

主要改变的是构造函数。因为创建和绘制网格被封装到了 CollisionGrid 类里，使用它的方法，就可以把 makeGrid 和 drawGrid 函数删掉了。

接着调用的 makeBalls 函数几乎没怎么动，只是用 vector 代替了数组。

然后对新类进行时间测试，要是有兴趣也可以试试庶民检测。在测试代码的循环中，调用了 \_grid.check(\_balls)，然后就是已经介绍过的需要进行碰撞检测的对象列表 \_grid.checks。下面的循环为您揭示如何使用该列表：

```

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

```

遍历递增步伐为 2，通过索引 j 和 j+1 取得对象并转换成 Ball 类型。接着传递给原有的函数 checkCollision。

在我这儿测试的结果，速度比起之前的几乎快了一倍。这得归功于使用了 vector 以及仅创建需要的网格（懒惰实例化）。

是该让它动起来的时候了，我们蓄势待发。让它动起来真的不难，只要稍稍再多做点事。同时，由于根据场景大小和对象数量调整出了最佳的网格大小，所以测试程序可以不要了，但要记得对于一个新的项目，需要花一些测试时间来做调整。

再次改版的类 GridCollision3

```

package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.display.DisplayObject;
    import flash.events.Event;
}

```

```

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

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

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

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

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

    function onEnterFrame(event:Event):void
    {
        updateBalls();
        _grid.check(_balls);
        var numChecks:int = _grid.checks.length;
        for(var j:int = 0; j < numChecks; j += 2)
        {
            checkCollision(_grid.checks[j] as Ball, _grid.checks[j + 1] as Ball);
        }
    }

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

    private function updateBalls():void
    {
        for(var i:int = 0; i < _numBalls; i++)
        {
    }
}

```

```

var ball:Ball = _balls[i] as Ball;
ball.update();
if(ball.x < RADIUS)
{
    ball.x = RADIUS;
    ball.vx *= -1;
}
else if(ball.x > stage.stageWidth - RADIUS)
{
    ball.x = stage.stageWidth - RADIUS;
    ball.vx *= -1;
}
if(ball.y < RADIUS)
{
    ball.y = RADIUS;
    ball.vy *= -1;
}
else if(ball.y > stage.stageHeight - RADIUS)
{
    ball.y = stage.stageHeight - RADIUS;
    ball.vy *= -1;
}
ball.color = 0xffffffff;
}

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

```

增加了 `onEnterFrame` 和 `updateBalls` 函数。构造函数只用来创建网格和球，并设置 `ENTER_FRAME` 监听器。在 `onEnterFrame` 函数中，通过 `updateBalls` 更新球的位置，然后执行网格碰撞检测法。

`updateBalls` 函数就是用来移动对象，不让它们出场景，以及在每帧一开始改变恢复它们的颜色为白色。

只有随后在 `checkCollision` 函数中发生碰撞了才会被设置成红色。

当然啦，可以运用上本书“动量守恒”一章提及的内容来丰富碰撞的反应，让球之间可以弹来弹去。

通过这个碰撞检测法，应该让你学到了如何一次性走完场景并多次和其它对象发生反应。一定记住调整的秘诀，测试测试再测试，确保获得最佳性能及高速检测。

### 检测不只是为了碰撞

当思考碰撞检测时，自然而然想到的是两个对象互相碰撞。但当采用基于距离的检测方法时，更多的是在考虑两个对象的空间关系。可能不会只关心碰撞，反而对对象间的实际距离更感兴趣。

比如，有种游戏，敌人为了发现我们的英雄会有一个很大的视野范围。

对此，网格检测仍然适用。只是把决定网格大小的因素，由对象大小改为对象间距离。

在上本书中，有一个绝好的粒子交互效果可以用来展示网格检测的这番应用。

先来看看原本的代码：

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

    [SWF(backgroundColor=0x000000)]
    public class NodeGardenLines extends Sprite
    {
        private var particles:Array;
        private var numParticles:uint = 500;
        private var minDist:Number = 50;
        private var springAmount:Number = .001;

        public function NodeGardenLines()
        {
            init();
        }

        private function init():void
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;
            particles = new Array();
            for(var i:uint = 0; i < numParticles; i++)
            {
                var particle:Ball = new Ball(3, 0xffffffff);
                particle.x = Math.random() * stage.stageWidth;
                particle.y = Math.random() * stage.stageHeight;
                particle.vx = Math.random() * 6 - 3;
                particle.vy = Math.random() * 6 - 3;
                addChild(particle);
                particles.push(particle);
            }
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            graphics.clear();
            for(var i:uint = 0; i < numParticles; i++)
            {
                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);
    }
}
}

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, 0xffffffff, 1 - dist / minDist);
        graphics.moveTo(partA.x, partA.y);
        graphics.lineTo(partB.x, partB.y);
        var ax:Number = dx * springAmount;
        var ay:Number = dy * springAmount;
        partA.vx += ax;
        partA.vy += ay;
        partB.vx -= ax;
        partB.vy -= ay;
    }
}
}
}
}

```

这里仅对代码做大概解释。基本上先是创建一堆粒子然后让它们动起来。  
如果两个粒子间距离小于一个指定数值，就发生反弹，并在之间画一条线。  
粒子越近，反弹越烈，线条越亮。要注意设置场景为黑色的一句程序：

[SWF(backgroundColor=0x000000)]

该句在 Flex Builder 和 Flash CS4 中都有效，但是在 Flash CS3 无效，只能通过手动设置来改变场景背景色。不过由于之前使用了 Flash 10 才有的特性(vector)，那么应该都在用 Flash CS4

了吧。

这段程序同样也用了一个叫 Ball 的类，虽然和之前本书提到的 Ball 类有点差异，但影响不是很大。

实例中用了 30 个粒子，每个例子间相互作用的距离在 100 像素以内。这样的环境看来无需做什么改进。

那么就让我们打破这一舒适环境，修改一组参数：

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

把粒子数量加到 500，反应距离降低到 50。如果场景大小显得拥挤，就拉大一些，比如 1000x800。

现在运行程序，会发现跑的很艰苦，帧频也很可怜。

对粒子间进行循环检测是不明智的，看看使用网格检测有带来什么帮助吧。

同时把数组改成 vector，如果对此改进存有怀疑，可以把数组改成 Vector.<DisplayObject>，看看这步到底能带来多少改进。记得需要导入 flash.display.DisplayObject 类，还有把所有对 Ball 的转型全部改成 DisplayObject，比如：

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

在我这里，虽然看到了改进，但是不明显。

一帧 500 次的双重循环，结果是要检测 124750 次。关键就是要降低这个数字，因为实际需要碰撞检测的比率很小。

执行网格检测后，数字下降到 10000 次。这个新类叫 NodeGardenGrid：

```
package {  
    import flash.display.DisplayObject;  
    import flash.display.Sprite;  
    import flash.display.StageScaleMode;  
    import flash.display.StageAlign;  
    import flash.events.Event;  
    import flash.geom.Point;  
  
    [SWF(backgroundColor=0x000000)]  
    public class NodeGardenGrid extends Sprite  
    {  
        private var particles:Vector.<DisplayObject>;  
        private var numParticles:uint = 500;  
        private var minDist:Number = 50;  
        private var springAmount:Number = .001;  
        private var grid:CollisionGrid;  
  
        public function NodeGardenGrid()  
        {  
            init();  
        }  
  
        private function init():void  
        {  
            stage.scaleMode = StageScaleMode.NO_SCALE;  
            stage.align = StageAlign.TOP_LEFT;  
  
            grid = new CollisionGrid(stage.stageWidth, stage.stageHeight, 50);  
            particles = new Vector.<DisplayObject>();  
            for(var i:uint = 0; i < numParticles; i++)
```

```

    {
        var particle:Ball = new Ball(3, 0xffffffff);
        particle.x = Math.random() * stage.stageWidth;
        particle.y = Math.random() * stage.stageHeight;
        particle.vx = Math.random() * 6 - 3;
        particle.vy = Math.random() * 6 - 3;
        addChild(particle);
        particles.push(particle);
    }

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    graphics.clear();
    for(var i:uint = 0; i < numParticles; i++)
    {
        var particle:Ball = particles[i] as Ball;
        particle.x += particle.vx;
        particle.y += particle.vy;
        if(particle.x > stage.stageWidth)
        {
            particle.x = 0;
        }
        else if(particle.x < 0)
        {
            particle.x = stage.stageWidth;
        }
        if(particle.y > stage.stageHeight)
        {
            particle.y = 0;
        }
        else if(particle.y < 0)
        {
            particle.y = stage.stageHeight;
        }
    }
    grid.check(particles);
    var checks:Vector.<DisplayObject> = grid.checks;
    trace(checks.length);
    var numChecks:int = checks.length;
    for(i=0; i < numChecks; i += 2)
    {
        var partA:Ball = checks[i] as Ball;
        var partB:Ball = checks[i + 1] as Ball;
        spring(partA, partB);
    }
}

private function spring(partA:Ball, partB:Ball):void
{
    var dx:Number = partB.x - partA.x;

```

```

var dy:Number = partB.y - partA.y;
var dist:Number = Math.sqrt(dx * dx + dy * dy);
if(dist < minDist)
{
    graphics.lineStyle(1, 0xffffffff, 1 - dist / minDist);
    graphics.moveTo(partA.x, partA.y);
    graphics.lineTo(partB.x, partB.y);
    var ax:Number = dx * springAmount;
    var ay:Number = dy * springAmount;
    partA.vx += ax;
    partA.vy += ay;
    partB.vx -= ax;
    partB.vy -= ay;
}
}
}
}

```

在确保文件正确发布后。我这里的测试结果，每帧检测是 6500 次。

注意看网格大小和 minDist（反应距离）一样是 50，而不是粒子的半径。

记得为了获取最佳性能，要多做测试来得出合适的网格大小。

虽然一个更大的网格会带来更多的检测次数，但却会减少循环次数。

肯定存在一个最佳性能点，不过在这儿，最小值 50 的性能也挺不错了。

## 总结

这章讨论的是高级碰撞检测，并通过一个简单的例子作为入门介绍。

很多地方都可以使用到这两种技术（基于位图检测法和基于网格检测法）。

再一次重申，对网格检测系统来说，反复测试性能，调整碰撞引擎到最佳状态是本章首要内容。

同时记住，它擅长于处理大量对象，而并非少数对象。

下一章将带来一个截然不同的主题，转向行为 (streeing behaviors)。还会牵涉一些人工智能。

## 第二章 转向行为

转向行为 (steering behaviors) 这一术语，指的是一系列使对象行动起来像似长有智商的算法。这些行为都归于人工智能或人工生命一类，是让对象呈现出拥有生命一般，对如何移动到目的地、捕捉或逃避其它对象、避开障碍物、寻求路径等做出因地制宜的决定。

这个术语创造于 Craig Reynolds 在 1999 年的游戏开发者大会上发布的一篇标题为“自主角色的转向行为”(Steering Behaviors for Autonomous Characters) 的论文。它描述了一系列算法，用来创建出一个部队系统，应用于模拟或者游戏中的角色。部队通常会给角色在行动上带来各种影响。他们围绕着力度在强弱上也各不相同，比如鸟群。实际上，Reynolds 的“boids”就是用来模拟鸟群的。



**Figure 2-1. Craig Reynolds' "boids"**

Reynolds 的论文可以在 [www.red3d.com/cwr/steer/](http://www.red3d.com/cwr/steer/) 中找到，而搜索论文标题同样能得到更多的下载地点。在论文中，虽然提及一个 C++ 版本的开源程序

OpenSteer (<http://opensteer.sourceforge.net/>)，但是对于算法的实现，Reynolds 没有涉及到很深。但有很多对该系统基本原理的描述，人工智能运动引擎创建于这些原理之后。这一章将跟随其步伐，把论文中提到的大部分行为，用 AS3 创建出来。

在此之前，先明确这章的目的：介绍行为，了解行为，展示一个实现这些行为的框架。一些行为根据复杂度不同，实现起来有多种不同方式。所有行为都不存在一个标准或者正确的做法，实现上给出的也是很简单的样式。换句话说，仅从介绍和展示的角度去考虑实现。要是用于产品开发的话，提供的代码需要根据要求做大量调整。

### 行为

首先，预览一下几个基本行为，看看它们是什么，要干什么。

寻找 (seek)：角色试图移动到一个指定点。该点可以是一个固定点也可以是把另一个角色作为目标的移动点。

避开(flee): 与寻找正好相反。角色试图避开一个给定点。同样，这个点也可以是固定点或者移动点。

到达(arrive): 和寻找相同，除了角色的速度在接近目的地时会减慢，最终以一个渐变运动恰好停留在目标处。

追捕(pursue): 寻找的加强版。由于目标会做加速运动，所以角色会事先预测然后再移动到该点。很明显，由于固定点是不会速度概念的，所以这里用目标代替点的概念。

躲避(evade): 与追捕正好相反。角色对目标的速度做出预测，然后尽可能躲避开来。

漫游(wander): 随机但平滑又真实的运动。

对象回避(object avoidance): 角色预测出对象的行动路径，然后避开他们。

路径跟随(path following): 角色尽可能的沿着自己的路径移动，但要考虑符合一些真实的物理现象，以及使用其它行为后的影响。

除了这些行为，还有复杂的类似鸟群这样的复合行为，它大致上由以下三种简易行为合成：

分离(separation): 鸟群中每个角色都试着和相邻角色保持一定的距离。

凝聚(cohesion): 每个角色尽量不掉队，不落下太远。

队列(alignment): 每个角色尽可能与相邻角色行动于同一方向。

虽然是三个相对简单的行为，但合在一起时就能产生出难以想象的群体效果，像鸟群或是其它生物群体。调整这三个行为中的各个参数会改变群体的性质，有经常分散的松散型队伍，有紧密的小团体，还有一个带头的率领着的队伍，等等多种变化。

在让对象按照行为行动起来之前，首先要有一个能让它们动起来的方法。

## 2D 向量(Vector2D)类

转向行为已经被各种语言实现过多次了(AS 可能也有一两次)，其最底层是用向量来描述的(也是最常见的实现方式)。(如果想学习向量，请看上本书的第五章，那里有非常详细的描述)。

概括的看，一个向量由两部分组成：一个方向和一个大小。比如，一个运动中对象的速度由它要去哪里(方向)和移动快慢(大小)两部分组成。因此，把速度看作一个向量是最贴切不过的。加速度——任何改变对象速度的作用力——同样也是由力的方向和大小组成(另一个向量)。向量同样也可以用来描述对象间的位置关系，其中大小代表距离，方向代表角度。

向量还可以用来表示一个角色(脸)的朝向，这种情况下就只管方向，而忽视大小，也可以说大小等于1。这样的向量叫做单位向量(unit vector)。实际上，只有一单位长度的向量，在数学运算上能起到很大的优化作用。

向量的所有这些特性对转向行为来说都很有用，因为速度，队伍方向，对象间距离，对象的朝向都会被大量的使用。于是创建一个 AS3 版本的向量类则势在必行。作为一个好人，我决定把我编写的一个送给各位，可后来发现 Flash CS4 自带了一个 3D 向量(Vector3D)类... 可后来又发现多了一维反而更复杂了，所以我决定在本章中仍旧使用我的 2D 向量类。我不想对类 做过多的解释，类中包含的众多方法都可以在别的语言实现的版本中找到，特别是那些用于转向行为的。

代码 Vector2D:

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

    /**
     * A basic 2-dimensional vector class.
     */
    public class Vector2D
    {
        private var _x:Number;
        private var _y:Number;

        /**
         * Constructor.
         */
    }
}
```

```

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

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

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

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

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

/**
 * Sets / gets the length or magnitude of this vector. Changing the length will change the x and y
but not the angle of this vector.
 */
public function set length(value:Number):void

```

```

{
    var a:Number = angle;
    _x = Math.cos(a) * value;
    _y = Math.sin(a) * value;
}
public function get length():Number
{
    return Math.sqrt(lengthSQ);
}

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

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

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

/***
 * Ensures the length of the vector is no longer than the given value.
 * @param max The maximum value this vector should be. If length is larger than max, it will be

```

```

truncated to this value.

    * @return Vector2D A reference to this vector.
    */
public function truncate(max:Number):Vector2D
{
    length = Math.min(max, length);
    return this;
}

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

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

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

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

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

```

```

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

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

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

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

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

/**
 * Adds a vector to this vector, creating a new Vector2D instance to hold the result.
 * @param v2 A Vector2D instance.
 * @return Vector2D A new vector containing the results of the addition.

```

```

*/
public function add(v2:Vector2D):Vector2D
{
    return new Vector2D(_x + v2.x, _y + v2.y);
}

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

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

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

/***
 * Indicates whether this vector and another Vector2D instance are equal in value.
 * @param v2 A Vector2D instance.
 * @return Boolean True if the other vector is equal to this one, false if not.
 */
public function equals(v2:Vector2D):Boolean
{
    return _x == v2.x && _y == v2.y;
}

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

```

```

public function get x():Number
{
    return _x;
}

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

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

```

像 C 或 C++ 这类语言支持操作符重载，也就是可以自定义一些操作符的功能，像 +、-，作为类的一种函数，所以 `vectorC = vectorA.add(vectorB)`，可以写成 `vectorC = vectorA + vectorB`。

这么做可以让类用起来像是语言内置的一样。而 AS 还不支持，所以只能用单词写法来实现。

对于实现这样的类，在架构上就存在着挑战，比如决定类的方法该如何工作。对这些方法 `truncate`, `normalize`, `reverse`, `add`, `substrat`, `multiple` 和 `divide` 有两种考虑情况：是直接对调用对象做改变呢，还是返回一个新的对象。

举个例子，假设有 `vectorA(3, 2)` 意思是 x 等于 3，y 等于 2，和 `vectorB` 等于 `(4, 5)`，然后执行以下代码：

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

根据第一种情况，`vectorB` 不变，而 `vectorA` 等于 `(7, 7)`。

而另一种情况，`vectorA` 和 `vectorB` 都不变，但是产生一个新的等于 `(7, 7)` 的向量，我们让它等于 `vectorC`。

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

那么哪种做法才合适呢？对此，我前前后后进行了多番审视，最终发现在很多数学运算时，需要把一个对象——比如位置和速度——用向量来表示，而无所谓运算后对象本身的改变。所以，加、减、乘、除不对原对象进行修改。

然而 `truncate`（截断）、`reverse`（倒置）和 `normalize`（单位化）则更注重对象本身的改变，所以这些操作用来直接改变对象要比返回一个新的更有用。

互换以上操作方式也不是很难。如果想把 `vectorB` 加在 `vectorA` 上，可以这样：

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

而如果想让 `normalize` 返回一个新的对象，可以使用 `clone`（克隆）函数：

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

正如 Seb Lee-Delisle 在为本书做技术回顾时所说，如上所示，创建对象副本会导致大量临时对象的产生和销毁。在 Flash Player 中的垃圾收集器会为此忙碌而出现性能问题。为清晰起见，

我保留了这种做法，但也为此做了微调。

现在，有了向量类可以表示角色的位置，速度和各种群体。还需要一个类用来表示角色。

## 机车(Vehicle)类

机车类是转向角色的基类，但它不提供任何转向行为，只处理与运动相关的基本内容，如位置，速度，质量以及角色接触场景边缘后的反应（反弹还是穿越出现在另一边）。转向机车(SteeredVehicle)类继承机车类，并为之增加转向行为。使用这样的结构 其目的是为了让机车类可以用于仅需要移动而不需要转向行为的对象，同时也可以让转向机车类不考虑基本运动的细节而专心实现转向功能。

迄今为止，我们一直使用“角色”一词代表能移动且拥有行为的对象。从现在起，术语“角色”和“机车”将会交替着使用。可以认为成角色就是一个驾驶中的机车，或者干脆把机车当作的角色本身。如果还不能理解，就请阅读本章开头提起的《自主角色的转向行为》中“移动”这一章，那里解释的非常清楚。

无废话，看代码：

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

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

        // potential edge behaviors
        public static const WRAP:String = "wrap";
        public static const BOUNCE:String = "bounce";

        /**
         * Constructor.
         */
        public function Vehicle()
        {
            _position = new Vector2D();
            _velocity = new Vector2D();
            draw();
        }

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

```

        graphics.lineTo(10, 0);
    }

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

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

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

        // update position of sprite
        x = position.x;
        y = position.y;

        // rotate heading to match velocity
        rotation = _velocity.angle * 180 / Math.PI;
    }

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

            if(position.y > stage.stageHeight)
            {
                position.y = stage.stageHeight;
            }
        }
    }
}

```

```

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

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

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

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

/**
 * Sets / gets maximum speed of character.
 */
public function set maxSpeed(value:Number):void
{

```

```

        _maxSpeed = value;
    }
    public function get maxSpeed():Number
    {
        return _maxSpeed;
    }

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

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

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

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

```

这里没有什么新的概念，一些基础概念都在上本书中提到过，只是在处理上有点不同。首先，采用两个 2D 向量来分别表示位置和速度，用 `_position`, `_velocity` 代替 `x`, `y`, `vx`, `vy`。大多数工作都发生在 `update` 函数中。一上来先试着截断(`truncate`)速度向量，确保不会超过最大速度，然后把速度向量加于(`add`)位置向量上。在上本书中的做法是这样：

```
x += _vx;
```

```
y += _vy;
```

而现在只需一行：

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

接着检测是否在边缘，是的话调用 `wrap` 或者 `bounce` 函数。最终，根据位置向量更新对象的 `x` 和 `y` 值，并调整其角度：

```
x = position.x;
```

```
y = position.y;
```

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

余下大部分 `getter/setter` 函数用于读写各个私有(`protected`)属性。还有一个 `draw` 函数，是为了让对象能够被看见，它可以根据需要在子类中重载。

为 `Vehicle` 类做一个快速测试，创建一个新的文档类 `VehicleTest`。

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

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

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

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

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

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

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

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

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

在例子中创建了一个 Vehicle 对象并增加到显示列表。它的位置由一个 2D 向量决定：

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

另一个改变位置的方法是直接设置位置的 x 和 y 值。

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

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

或者直接设置被重载过的 x 和 y，与此同时 position 也会跟着一起改变。

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

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

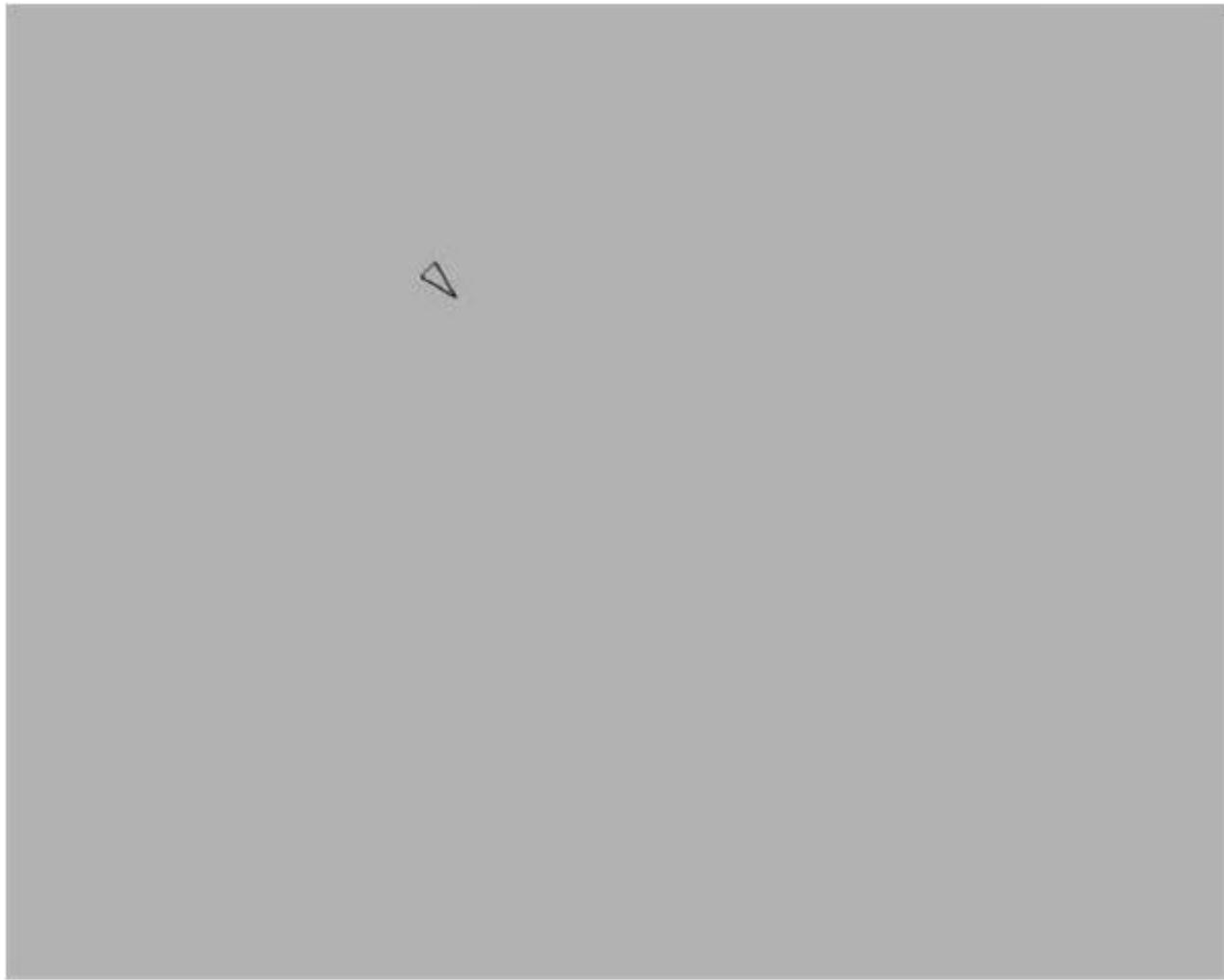
例子中对设置速度采用了另一种方式：长度 (length) 和角度 (angle)，这也显示了向量在使用上的弹性。

```
_vehicle.velocity.length = 5;
```

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

长度在这里指速度的大小，角度指方向。别忘了 angle 实际上是弧度，所以 Math.PI / 4 相当于 45 度。再次提醒，如果确定机车的速度分量，那么就可以像对位置向量那样，给速度向量分配一个新的 2D 向量或者直接设置速度向量的 x 和 y 属性。

最终，加上 ENTER\_FRAME 监听事件，在每帧调用 update 函数。这样机车就会沿着给定的方向不断前进。当接触到边缘后，它会出现在场景的另一端。如图 2-2。



**Figure 2-2. A moving vehicle**

机车类的测试已经足够了。让我们开始迈向更好更强大的转向行为之旅吧。

### 转向机车 (SteeredVehicle) 类

转向机车类继承机车类并增加转向行为。每个行为都被定义成公开函数，在每帧或者一段时间间隔内调用以实现对应的转向力。通常所有转向力在调用之后再调用机车的 update 函数。

举个例子，如果想让机车漫游，那就调用 wander (漫游) 函数，再调用 update 函数。

```

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

```

转向函数都是这样工作的：不管什么时候调用了一个转向函数，都会计算转向力，这个力用来确定机车是顺时针旋转还是逆时针旋转。比如，seek（寻找）函数会计算出一个力，确保机车能从当前方向直接面向目标点。这或许会受到不止一个拥有转向行为的机车的影响，起初的寻找点，会在考虑避开或躲避后而改变。当这些力叠加后，update 函数才被调用，最终把一切都反应在机车上，并导致其速度的改变（方向和大小）。

以下是不包含转向行为的转向机车类的核心代码：

```

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

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

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

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

        override public function update():void
        {
            _steeringForce.truncate(_maxForce);
            _steeringForce = _steeringForce.divide(_mass);
            _velocity = _velocity.add(_steeringForce);
            _steeringForce = new Vector2D();
            super.update();
        }
    }
}

```

立刻进入眼帘的是`_steeringForce` 属性，它是一个 2D 向量。该属性作为每个行为叠加后的转向合力。同时留意此处还有一个`_maxForce` 属性，因为现实中不会有旋转是一瞬间完成的，所以要对旋转力加以限制，使其在一帧里的大小不会太离谱。可以通过公开的 getter/setter 函数 `maxForce` 读写。对 `maxForce` 的修正，会使机车旋转更急剧，移动更快速，走位更准确，或是出现一个很牛 x 的大甩尾。

现在，让我们解剖 `update` 函数。先假设有一堆转向行为已经被调用，那么此时的

`_steeringForce` 属性即是一个有意义的向量。第一个 `truncate`(截断) 函数是不让 `_steeringForce` 超过最大作用力。然后除以 (divide) 机车的质量 (mass)。在现实中，越重的物体有着越大的动力，旋转的角度也越大，而较轻的物体则旋转更快速。接着把转向力叠加于机车的当前速度上，再把 `_steeringForce` 设回零向量，以便于下一轮作用力的叠加。最终调用父类已经实现的 `update` 函数。

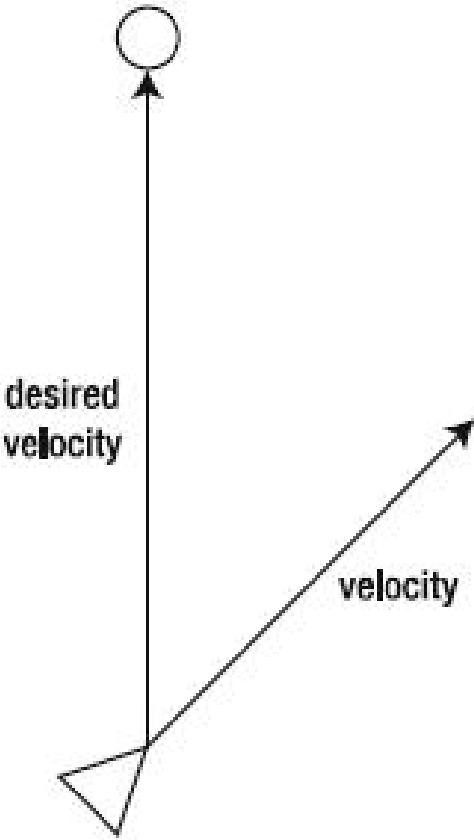
该是了解行为实现的时候了，从寻找 (seek) 行为开始。每个行为都是转向机车类的一个公开函数，某些行为会需要新的属性和额外的函数。

### 寻找行为

如之前所述，寻找行为只是把机车移动到指定点。就像这样：

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

首先，通过目标位置和机车位置相减，计算出一个能使机车准确到达目标的期望速度。这个速度向量的含义是“如果想到达目的地，需要以如此方向，移动这般快才行”。如图



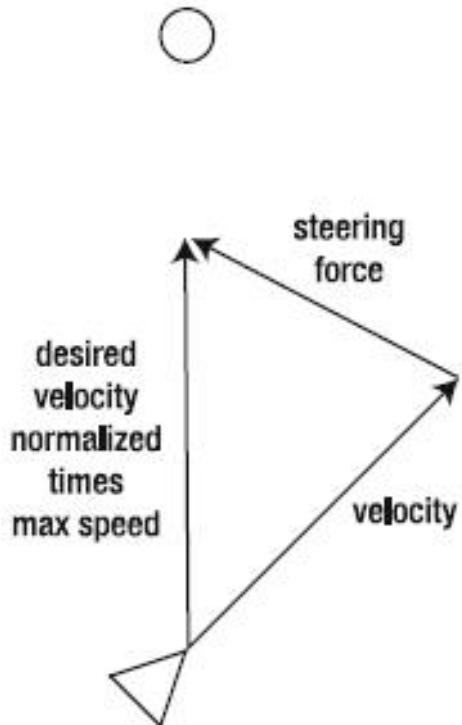
**Figure 2-3.** The desired velocity to reach the target right now

当然，现实不会让你想去哪儿就能一下子过去的。为此，多尝试几次后会找出一个不错的算法。在这里，我们通过单位化期望速度并乘以最大速率来实现。这样做，仍然能得到一个指向目标的向量，其含义则是“不要想着会瞬间移动，以最快速度向正确方向迈进吧。”

有了期望速度，显然还要考虑机车现有的速度。通过两者相减，得到一个向量，其含义是“尽最大可能以最大速率面向正确方位。”

这个向量还会叠加给转向力。记得 `update` 函数中，`_steeringForce` 总是被限制在最大力度

以内。所以，虽然仍旧没有朝着想要的准确方向走，却在最大力度和最大速率的限制下尽了最大的可能。如图：



**Figure 2-4.** Best possible desired velocity, and the force required to change current velocity to that

现在给出寻找行为的一个例子：

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

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

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

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

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

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

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

```

    {
        _vehicle.seek(new Vector2D(mouseX, mouseY));
        _vehicle.update();
    }
}
}

```

这是一个非常简单的例子。仅仅是让场景上一个有转向行为的机车在每帧去寻找鼠标。试着改变机车的最大速率和最大力度，或者改变其质量(mass)来感受一下这些因素对转向行为的影响。同样也可以试试用一个固定点代替鼠标，或者更刺激一点，创建另一个机车作为目标。这样onEnterFrame 函数中就会有类似的代码：

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

到此，应该已经见证了转向机车如何寻找鼠标或者另一辆机车，当对这些工作有了很好的理解后，我们进入下一个行为：避开。

### 避开行为

避开行为与寻找行为彻底相反。实际上，除了代码最后一行用相减代替了相加以外，其它都一样。

```
public function flee(target: Vector2D): void
```

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

由于和寻找几乎一样，所以就不详细解释了。最后一句的含义是“很好，既然发现了目标，那就调头往回走吧。”在此为避开做一个简单的测试：

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

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    public class FleeTest extends Sprite
    {
        private var _vehicle:SteeredVehicle;

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

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

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

```

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

```

抛开代码中调用的函数名称不谈，最主要的不同是，用两行代码将机车的初始位置放在离场景边缘有一段距离的地方，并且把接触场景边缘后的反应改为反弹。删除这两行再测试，就知道为什么要加这两行了。机车会为了避开鼠标在角落间来回切换，导致近似看不见。

现在我们有了一对正反行为，接下来要做的是为这对行为创建一对机车来看看情况。

```

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

```

```

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

```

```

public class SeekFleeTest1 extends Sprite
{

```

```

    private var _seeker:SteeredVehicle;
    private var _fleer:SteeredVehicle;

```

```

    public function SeekFleeTest1()
    {

```

```

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

```

```

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

```

```

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

```

```

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

```

```

    private function onEnterFrame(event:Event):void
    {

```

```

        _seeker.seek(_fleer.position);
        _fleer.flee(_seeker.position);
        _seeker.update();
        _fleer.update();
    }

```

```
}
```

这对机车叫寻找者(\_seeker)和避开者(\_fleer)。我相信不用再解释寻找者寻找避开者，避开者避开寻找者了吧(试着快速念十遍>w<! )。先运行一下看看两辆机车互动的效果，然后尝试改变它们的各种参数，再看看会发什么。

我们还可以把两个行为同时用于一辆机车上。下面的例子中，机车 A 同时寻找和避开机车 B，机车 B 同时寻找和避开机车 C，机车 C 同时的寻找和避开机车 A。这三辆机车会因为追捕各自的目标而形成一个圆。

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

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

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

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

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

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

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

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

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

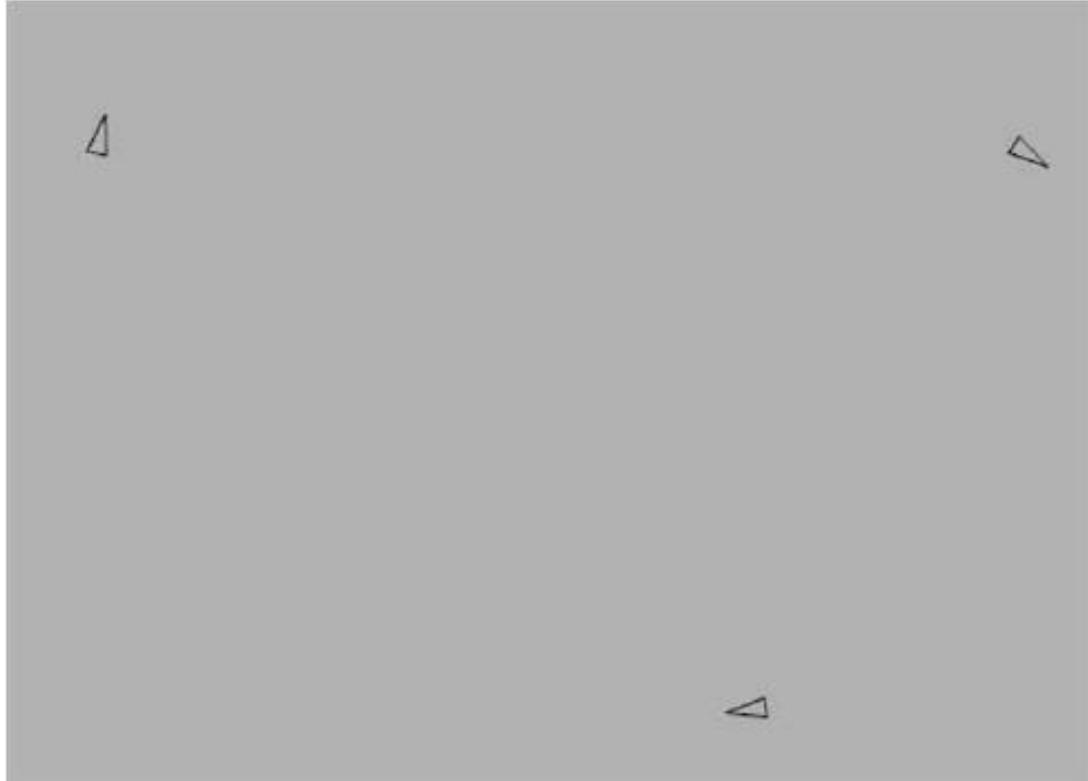
```

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

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

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

```



**Figure 2-5.** A chases B, B chases C, and C chases A.

记得再试试改变各种参数，观察发生的变化。如果这些都没问题了，那就开始探索下一个行为：到达。

### 到达行为

到达行为在很多场合都可以被当作是寻找行为。实际上，它们之间的算法和处理方式都一样。唯一不同的是，在到达模式中，一辆机车在到达目标的某一距离时，会变成一种精确模式慢慢地靠近目标点。

为了了解到达行为的必要性，可以先运行一下 SeekTest 类，然后移动鼠标到某处让机车过来“抓住”它。会看到机车快速的越过了鼠标，接着它发现过头了，又返回来，还是过头了....于是会一直循环下去。这是因为机车始终保持着最大速度迈向目标，哪怕离目标只有几像素。

到达行为通过减速接近目标，解决了这个问题：

```

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

    var dist: Number = _position.dist(target);
    if(dist > _arrivalThreshold)

```

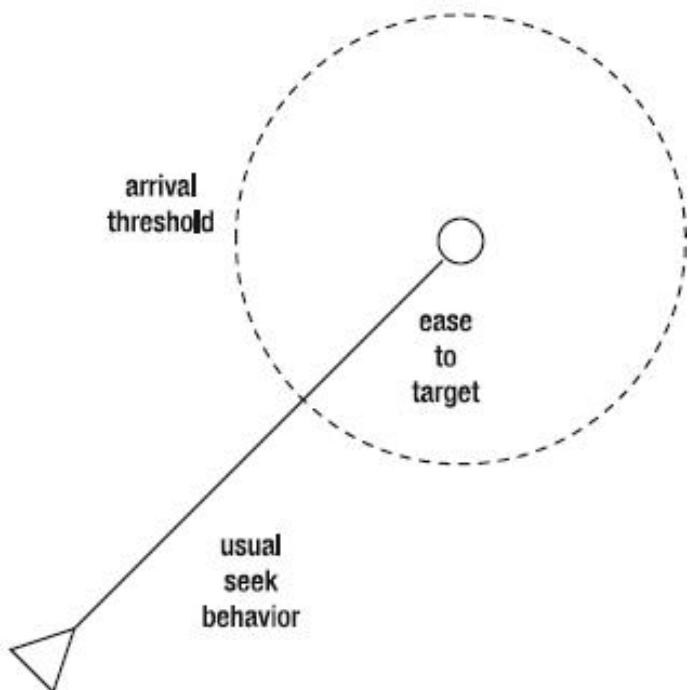
```

{
    desiredVelocity = desiredVelocity.multiply(_maxSpeed);
}
else
{
    desiredVelocity = desiredVelocity.multiply(_maxSpeed * dist / _arrivalThreshold);
}
var force: Vector2D = desiredVelocity.subtract(_velocity);
_steeringForce = _steeringForce.add(force);
}

```

程序一开始和寻找行为一样。但是在期望速度乘以最大速率时，做了距离检测。如果距离大于某个值，那一切照旧。程序往后走，接着的事情也和寻找一样。

关键是，距离小于某个值时所做的事情。本来乘以`_maxSpeed`现改为乘以`_maxSpeed * dist / _arriveThreshold`。如果距离仅仅小于某个值一点点，那么`dist / _arriveThreshold`会非常接近 1.0，可能是 0.99。因此，期望速度的大小也会非常接近于（略小于）最大速率。如果距离接近 0，那么得到的比率也会非常非常小，期望速度改变也会很小。最终速度会趋向于 0（假设只有一个行为作用于该机车）。



**Figure 2-6.** If within the arrival threshold, ease to target. Otherwise, just use seek.

当然，转向机车类需要这么一个“某个值”属性，所以我们把它加上去：

`private var _arrivalThreshold:Number = 100;`

还要为此再增加一对 getter/setter 函数：

```

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

```

看看这些是如何运用在测试类中的：

```

package
{

```

```

import com.foed.SteeredVehicle;
import com.foed.Vector2D;

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

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

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

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

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

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

```

和测试寻找行为唯一的不同就是在 onEnterFrame 中把函数名 seek 换成了 arrive。运行一下试试把鼠标移动到某处，机车先是以寻找模式发现 目标，然后慢慢的停在鼠标所在位置。再次移动鼠标又会回到寻找模式。通过调整 arriveThreshold 属性，看看机车接近目标时的变化吧。

如果愿意可以再试着玩玩增加多辆机车，或者现在就进入下一个行为：追捕。

### 追捕行为

对于追捕行为，又要重复一遍“它非常类似寻找行为”。实际上，追捕的最后一个动作就是调用寻找。追捕的本质是预测目标所在的位置。这也暗示目标是一个移动对象，它也有位置和速度属性。因此，假设目标是一辆机车，而它在大多数情况下是另一种继承机车类的转向机车。

那么，该如何预测目标的位置呢？其实是以目标的当前速度不变为前提，算出未来一段时刻后目标所在的位置。但是需要计算多久的未来才合适呢？问的好！我们把这 段时间叫做预测时间 (look ahead time)。如果算到很久以后（较长的预测时间）可能就会超越目标，如果只往后算一点点（较短的预测时间）可能仍然落后于目标。而实际上，寻找行为就是一个预测时间为零的追捕行为（零秒后的未来？没错，就是此刻。）

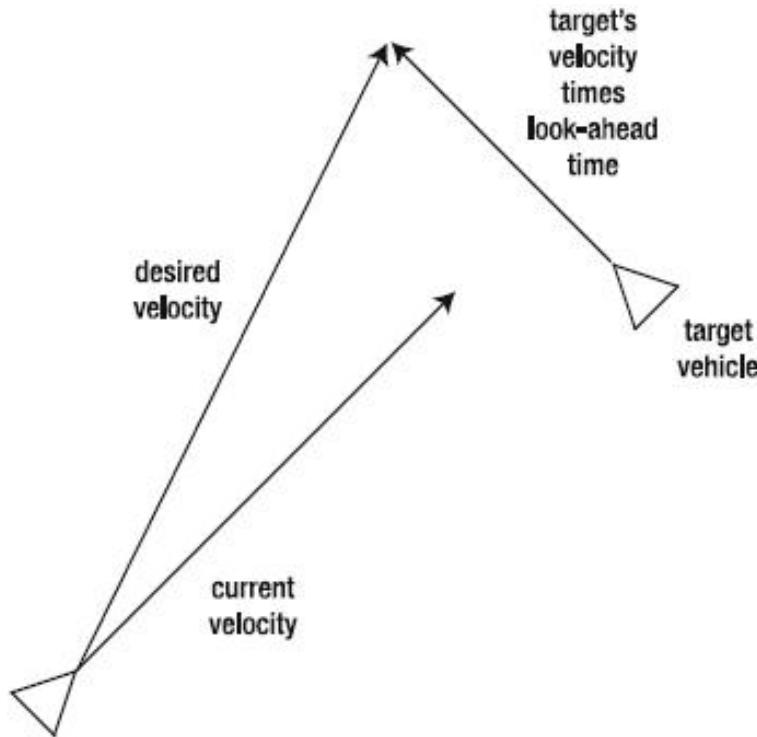
一个策略是基于两机车间的距离来判断预测时间。如果目标太远，需要花一段时间才能赶上，就预测长一点，如果很接近，马上就能达到，就预测短一点。把这个技术用于转向机车，就像这样：

```

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

```

首先通过两者间距离除以最大速率计算出预测时间。这就得到了追上目标所需要的时间间隔（假设目标不再移动）。通过目标的速度乘以时间间隔得到预测移动距离，再加到当前位置上就是预测位置。最后，把这个预测位置作为寻找点。



**Figure 2-7.** Pursue calculates where the vehicle will be in the future and then seeks to that point.

需要注意，这个函数的结果很粗糙，因为这一切都是在运动中的。但对于单单寻找来说还是比较精确的。该方法也比较简单快速，对于追捕还有别的更准确，也更复杂的算法，如果有兴趣可以自己研究。这里所做的至少能为您开个头。

让我们测试一下吧。这次要创建三辆机车。一个是只顾移动的机车，其作为目标，另外两个转向机车，一个用寻找行为，一个用追捕行为。如果一切正常，追捕者靠着优越的算法会胜出。

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

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

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

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

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

```

addChild(_seeker);

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

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

addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

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

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

    _target.update();
}
}
}

```

两辆转向机车开始于同一起点，但是寻找者向目标当前的位置移动，而追捕者则直接奔向目标的前方来拦截它。

对于这个测试，需要明确一点，有很多潜在的参数值会影响到结果。接下来的行为是：躲避。**躲避行为**

可能已经猜到，躲避就是追捕的反行为。就像追捕类似于寻找，躲避类似于避开。

躲避从本质上讲，是预测出机车将要去到的位置并远离它。在这里所有的原则都和追捕相同。实际上，就连实现都几乎一模一样，除了最后一行用避开代替寻找：

```

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

```

不再多说了，直接把追捕和躲避放一起测试：

```

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

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

    public class PursueEvadeTest extends Sprite

```

```

{
    private var _pursuer:SteeredVehicle;
    private var _evader:SteeredVehicle;

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

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

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

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

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

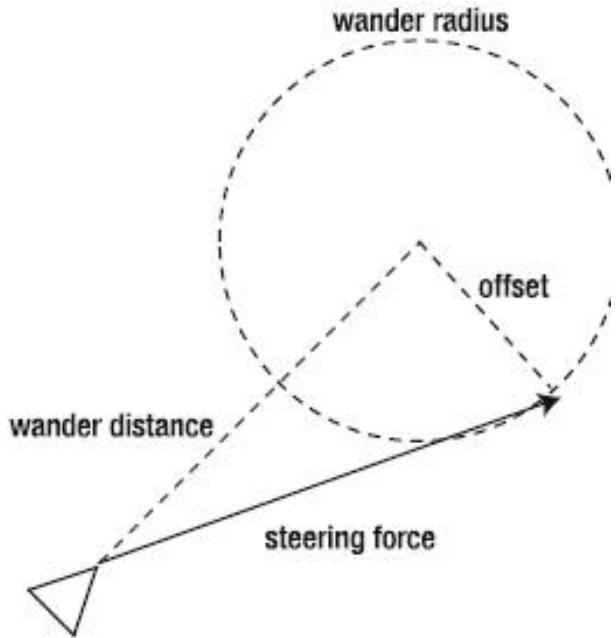
```

如你所见，这个例子实质上和 SeekFleeTest1 一样，只是把行为改成了追捕和躲避。因为两辆机车都使用了更高级的方法，而不是一个高级一个低级，所以很难看出和寻找/避开的测试有什么不同。如果把高低级方法穿插着使用，就能看出点区别所在了。接下来，我们要离开一下寻找/避开行为，来看看随机性更强的漫游行为。

### 漫游行为

漫游行为就像它的名字一样，角色在场景中毫无目的的移动。这通常用来模拟巡视和觅食，也有纯粹是为了漫游而漫游的。

漫游行为在实现上不像听起来那么容易。简单的使用随机而产生的布朗运动，会让角色感觉像是一个有神经病的傻瓜。我们需要更自然更平滑的感觉。有个办法，通常设想在角色前方有个圆，然后把圆上任意一点作为目标，每次更新都向这个随机点移动。由于目标点总是落在假象的圆上，所以转向力永远不会一下子就变化很大。可以参照下图加以理解：图 2-8



**Figure 2-8.** Wander's use of distance, radius, and offset to produce a steering force

有几个参数可以调整出不同的漫游的风格：圆的尺寸，圆离开角色的距离，目标点的随机范围。

来看看漫游函数：

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

一开始先通过单位化速度确定圆的中心点位于速度向量的正前方，然后乘以漫游距离，就是圆心的所在地。接着增加另一个偏移量来确定随机点。由于该点落在圆上，所以偏移量的长度等于圆的半径，偏移量的角度等于漫游角度。而漫游角度是根据漫游范围做适当的随机调整。接着把偏移量加于中心点就得到了变化所需要的力度向量。最后把这个力度叠加到转向力度上就完了。

注意这里用到了几个新的变量，需要一开始就在转向机车类里定义好：

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

还要为它们定义读取函数（除了\_wanderAngle 是完全用于随机的，可以不用定义）

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

```

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

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

```

最后当然是漫游测试的例子：

```

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

    public class WanderTest extends Sprite
    {
        private var _vehicle:SteeredVehicle;
        public function WanderTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _vehicle = new SteeredVehicle();
            _vehicle.position = new Vector2D(200, 200);
            addChild(_vehicle);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            _vehicle.wander();
            _vehicle.update();
        }
    }
}

```

可以试试改变这些参数看看会发生什么变化(也别忘了试试改变质量, 最大速率和最大力度)。或许用之前讲的寻找、追捕行为来抓漫游机车会更有趣。

随着我们逐渐深入, 有趣的事情将越来越多。以上这些都是简单行为, 接下来要研究的是相对复杂的行为, 第一个是对象回避。

### 对象回避

对象回避不比迄今为止已经见识过的行为, 它更为复杂, 并且每个人实现的方式都不尽相同,

难易有别。所以对于这么一个没有严格统一标准的行为，这里给出的仅仅是很基本的实现方式。我鼓励大家根据实际情况，为之做必要的改进。

对象回避主题的完整意义是指，在机车行走的路线中存在一些障碍物，机车必须绕开、防止触碰到它们。听上去和碰撞检测有关，然而这仅仅是发生在预测阶段，也就是：“以我当前的速度行驶下去，可能就会撞到它了。”

既然碰撞是预测的，就得长点脑子确保碰撞不会发生。你可能正幼稚的想，那停下来或者调头不就行了嘛，你忘了有很多行为是在同时发生着的。如果要躲避的是一个食肉动物，光靠停下来或者躲在树后面显然是不明智的。凡有脑子的，此时会采取一切手段来躲避，而食肉动物也同样会绕开障碍物来追捕你。

另外，如果要避开一个非常接近的东西，就必须改变路线。可能在沙漠中，发现远处有座金字塔，稍作调整甚至不作调整的继续前进都不会碰到它。而如果金字塔就在你面前，为了不撞上去，起码要转差不多 90 度左右。

现在了解了该行为的复杂程度，以及为什么存在那么多不同的实现方式了吧。在大多数解决方案中，首先把障碍物看作是一个圆（3D 中是球）。实际上障碍物可能并不是圆，但为了计算方便，还是把它们想象成一个有中心点和半径的对象。注意，通常情况下碰撞检测不需要严格到像素级别，只要大致的知道其大小和位置即可，然后设法绕开它。这里是用来描述障碍物的圆类：

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

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

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

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

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

这个类很简单，通过半径和颜色画出一个圆，并且有两个只读属性，半径和位置。由于是在向量环境中计算，所以位置返回一个 2D 向量。现在开始讲述回避行为的实现。

由于要回避的对象通常不止一个，所以回避函数通过对一个数组的遍历来确认哪些需要被避开。为此，会计算出一个转向力。这段代码比较复杂，并且有很多种版本，所以这里仅作举例性描述，不进行详细解释。

预测碰撞的代码是基于 David M. Bourg 和 Glenn Seemann 所著的《游戏开发中的人工智能》

的一个例子，不过是个粗糙的简易版。转向力的计算，则基于海量的源文件以及我自己的想象。  
关于代码的解释就看注释和图释吧：

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

        // 障碍物和机车间的位移向量
        var difference: Vector2D = circle.position.subtract(_position);
        var dotProd:Number = difference.dotProd(heading);

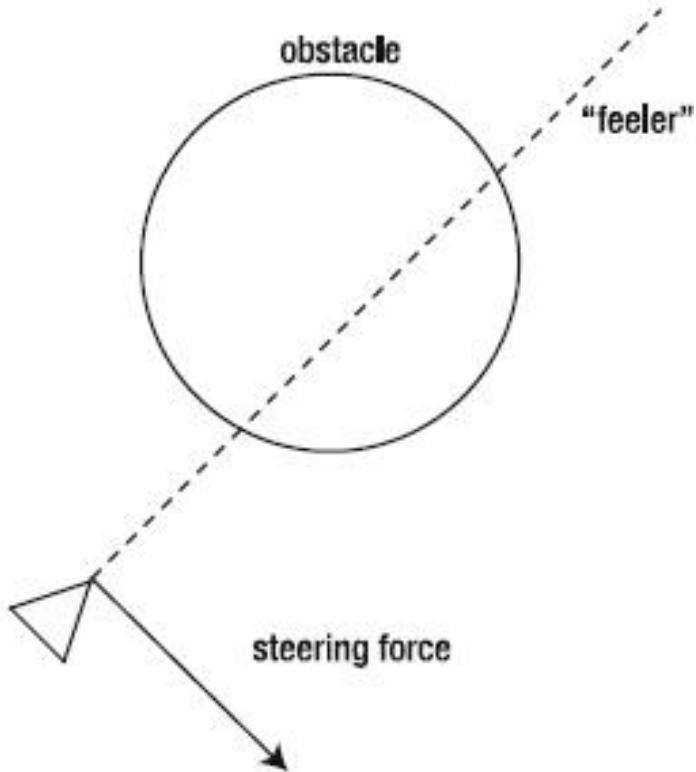
        // 如果障碍物在机车前方
        if(dotProd > 0)
        {
            // 机车的“触角”
            var feeler: Vector2D = heading.multiply(_avoidDistance);
            // 位移在触角上的映射
            var projection: Vector2D = heading.multiply(dotProd);
            // 障碍物离触角的距离
            var dist: Number = projection.subtract(difference).length;

            // 如果触角（在算上缓冲后）和障碍物相交
            // 并且位移的映射的长度小于触角的长度
            // 我们就说碰撞将要发生，需改变转向
            if(dist < circle.radius + _avoidBuffer &&
                projection.length < feeler.length)
            {
                // 计算出一个转 90 度的力
                var force: Vector2D = heading.multiply(_maxSpeed);
                force.angle += difference.sign(_velocity) * Math.PI / 2;

                // 通过离障碍物的距离，调整力度大小，使之足够小但又能避开
                force = force.multiply(1.0 - projection.length / feeler.length);

                // 叠加于转向力上
                _steeringForce = _steeringForce.add(force);

                // 刹车——转弯的时候要放慢机车速度，离障碍物越接近，刹车越狠。
                _velocity = _velocity.multiply(projection.length / feeler.length);
            }
        }
    }
}
```



**Figure 2-9.** When a collision with an object is detected, a steering force is applied.

注意这里有一对新增的属性，需要加入到类中：

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

回避距离(\_avoidDistance)意指，发现障碍物的有效视野。

回避缓冲(\_avoidBuffer)意指，机车在准备避开时，自身和障碍物间的预留距离。

这段代码并不完美，但在执行上还不错，没有让CPU遭太大罪。要是觉得不爽或是在使用中发现瓶颈了，即可进行优化。这里就用它完成一个测试：

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

    public class AvoidTest extends Sprite
    {
        private var _vehicle:SteeredVehicle;
        private var _circles:Array;
        private var _numCircles:int = 10;
        public function AvoidTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _vehicle = new SteeredVehicle();
```

```

_vehicle.edgeBehavior = Vehicle.BOUNCE;
addChild(_vehicle);
_circles = new Array();
for(var i:int = 0; i < _numCircles; i++)
{
    var circle:Circle = new Circle(Math.random() * 50 + 50);
    circle.x = Math.random() * stage.stageWidth;
    circle.y = Math.random() * stage.stageHeight;
    addChild(circle);
    _circles.push(circle);
}
addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
private function onEnterFrame(event:Event):void
{
    _vehicle.wander();
    _vehicle.avoid(_circles);
    _vehicle.update();
}
}
}

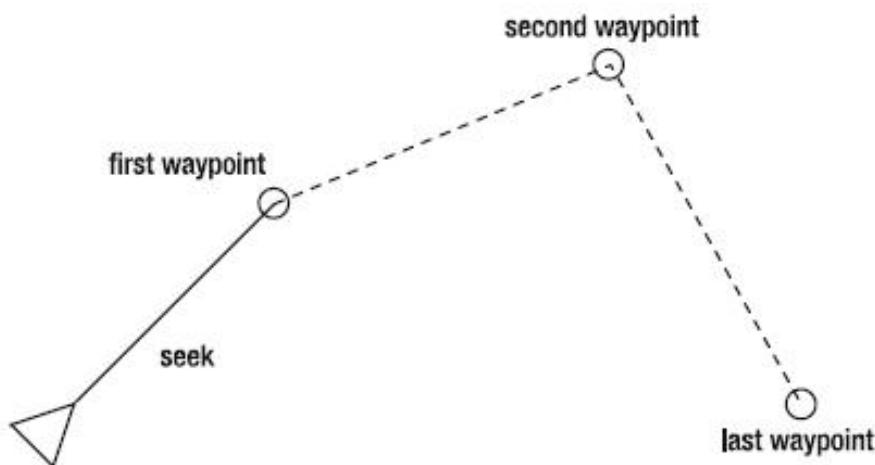
```

程序创建了一组障碍物，让机车漫游与此之中并不断试图避开它们。实现起来很简单，能同时处理漫游或者直线行驶，但是如果要混合寻找就比较无语了。机车会在企图寻找某个对象时，又不断的对它进行回避。关于这类行为更好的讲解会在第四章——寻路——中讨论。接下来要讲述的行为是：路径跟随。

### 路径跟随

路径跟随这名字一听就知道要干嘛了：机车会沿着一个预定的路线行驶。虽然在地图或者游戏中，路径是以图形的形式被表示的，而在转向行为中，其不过是一系列航点。在 AS3 的实现中，就是一个持有 2D 向量的数组。

其策略真是简单到不行。只要从第一个航点开始挨个寻找下去即可。



**Figure 2-10.** Seek to the first waypoint. When you get close enough, seek to the next one. And so on.

图 2-10

也可以是一个循环路径，这样当到达最后一个航点时又会回到第一个航点。

这里需要加两个新的属性到类中：

```

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

```

还有各自的读取器：

```

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

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

```

路径索引(pathIndex)相当于是数组索引，用于指向下一个航点。路径阈值(pathThreshold)相当于航点间距。

来看看实现：

```

public function followPath(path: Array, loop: Boolean = false): void
{
    var wayPoint: Vector2D = path[ _pathIndex ];
    if(wayPoint == null) return;
    if(_position.dist(wayPoint) < _pathThreshold)
    {
        if(_pathIndex >= path.length - 1)
        {
            if(loop)
            {
                _pathIndex = 0;
            }
        }
        else
        {
            _pathIndex++;
        }
    }
    if(_pathIndex >= path.length - 1 && !loop)
    {
        arrive(wayPoint);
    }
    else
    {
        seek(wayPoint);
    }
}

```

一堆乱哄哄的判断条件。首先是取得当前航点，如果航点不是一个有效 2D 向量，就返回，这作就是说，即使传递一个空数组也不会报错。接着是判断，到航点间的距离是否足以切换到下一个航点，然后根据循环再判断最后一个航点索引是否归零。写起来是很啰嗦，实际上画出航点后，按部就班的执行以上逻辑，就会发现是很直观的。

有了航点就能移动了，以此调用最后一行的寻找函数已经有了不俗表现，但为了更优美一点，

假设是最后一个航点又不要循环的话，采用到达行为可以使机车慢慢的靠近终点，而不会在终点左右摇摆。

该行为就这样了。还是来看看测试吧：

代码 PathTest

package

{

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

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

```
public class PathTest extends Sprite
```

{

```
    private var _vehicle:SteeredVehicle;
```

```
    private var _path:Array;
```

```
    public function PathTest()
```

{

```
        stage.align = StageAlign.TOP_LEFT;
```

```
        stage.scaleMode = StageScaleMode.NO_SCALE;
```

```
        _vehicle = new SteeredVehicle();
```

```
        addChild(_vehicle);
```

```
        _path = new Array();
```

```
        stage.addEventListener(MouseEvent.CLICK, onClick);
```

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

}

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

{

```
        _vehicle.followPath(_path, true);
```

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

}

```
    private function onClick(event:MouseEvent):void
```

{

```
        graphics.lineStyle(0, 0, .25);
```

```
        if(_path.length == 0)
```

{

```
            graphics.moveTo(mouseX, mouseY);
```

}

```
        graphics.lineTo(mouseX, mouseY);
```

```
        graphics.drawCircle(mouseX, mouseY, 10);
```

```
        graphics.moveTo(mouseX, mouseY);
```

```
        _path.push(new Vector2D(mouseX, mouseY));
```

}

```

    }
}

```

相当基础是吧？程序大多是在干绘制路径这种事。在 `onEnterFrame` 中，机车不断的跟随 `_path` 这个作为路径的数组。一开始，数组是空的——没有航点——所以机车一动不动，而当点击发生后，路径中的航点被创建，后面发生的事情大家都懂了。

注意机车并没有呆板的沿着路径行驶，而是比较自然的转过一些小角度的弯，或者跳过一些挨得很紧的航点。对此，通过改变路径阈值会产生变化。一些诸如质量，最大速率和最大力度这类属性也会对转向有影响。所以没事儿可以多调整些数值看看变化。

路径跟随经常和寻路（第四章）联合使用。寻路考虑的是两点间的最佳路径，其中包括角色不能逾越的地域，以及粗糙的地形和平滑的地形带来的不同影响。这些通常基于网格来表示，也就是说角色所在的环境是一个持有网格的数组。这个数组用作路径跟随行为中的路径数组，这样是不是更自然呢。

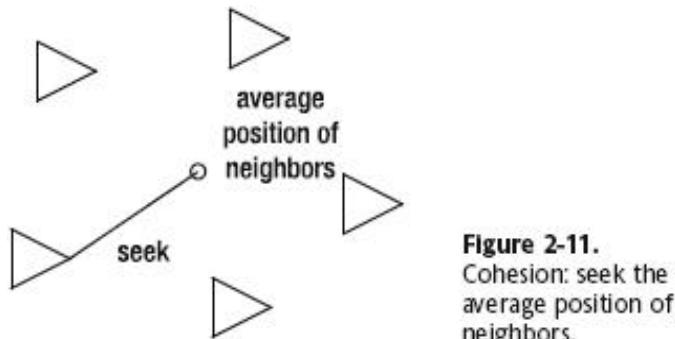
来吧，最后一个行为，由三个子行为混合而成的复杂行为：群落。

## 群落

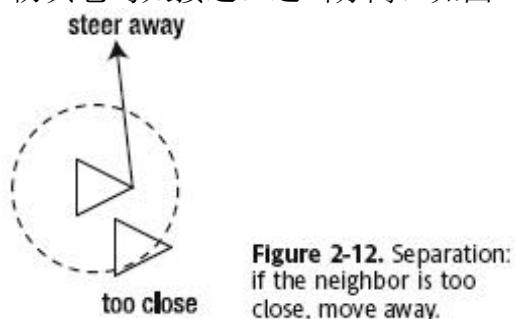
说到群落，很难不引用 Craig Reynolds 和他的“boids”模拟系统。Reynolds 很牛的将一个看似非常恐怖的复杂过程，拆成了几个比较简单的行为。

想想鸟群，它含有三个主要角色：

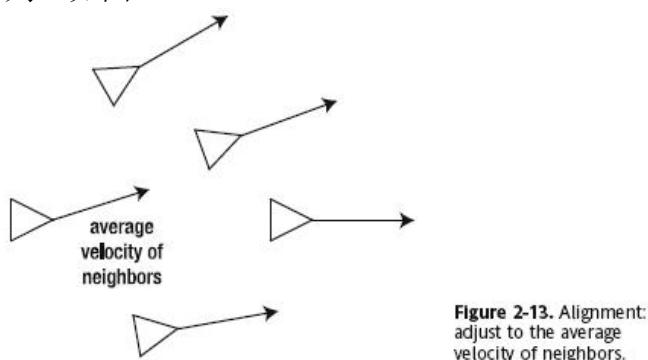
首先，鸟们都保持在同一个区域。如果有只鸟离队伍远了，就该马上归队。这叫凝聚。如图 2-11



其次，尽管鸟们都在一起飞，但是要避免不会互相碰到。为此，它们各自都有一个空间来预防其它鸟太接近。这叫分离。如图 2-12



最后，鸟们飞行在同一个方向。当然各自的角度不一定相同，但是大方向是差不多的。这叫队列。如图 2-13



这三个行为——凝聚、分离和队列——组成了复杂的群落行为。

当考虑鸟群时，就以整个群落是一条心去想象，或者认为每个鸟都充分认识群中的其它鸟。我不想为此去争论什么，但我要说，当开始理解这三个行为，何以促成群落 行为时，你会发现，每个鸟根本不需要知道多少东西，也不需要什么民主集中一条心来指挥群落。实际上，每个鸟就只需要看看临近的几只伙伴。如果靠太近就离远 点，如果方向差太多就转过来点，最终以此形成了传说中的群落行为。

尽管群落行为技术上被拆成了三个子行为，然而它们几乎总是捆绑出现的。一般不太会只对角色使用其中一两个行为，所以就把这仨放于同一个函数中好了。这样效率也高，避免要做三次循环。OK，莱茨狗：

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

首先，传递一个持有机车的数组。通过遍历这个数组找出进入视野的其它机车。把进入视野的机车的速度和位置都加起来，然后统计次数，最后以此求得平均值。如果机车靠太近，用避开函数离开之，以此实现分离。唯一要注意的地方就是处理过程中对自身的忽略。

当走完整个数组，算出平均速度和位置后，寻找平均位置，叠加平均转向力即完成任务。

似乎没啥了不起，不过有几个函数我们还没介绍呢，视野中 (inSight) 和太接近 (tooClose) :

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

```
public function tooClose(vehicle: Vehicle): Boolean
```

```

{
    return _position.dist(vehicle.position) < _tooCloseDist;
}

```

inSight 函数判定一个机车是否能看到另一个机车。为此，先要检测两者间距离是否在视野范围内，如果不是就返回 false。接着用向量的数学运算判断机车的前后关系，这里采用的实现方式比较死板，只认前方的机车，在后面就当作看不见。这个做做例子够用了，如果要作改进，可以先考虑做一个可变化的视野范围。窄的视野 范围意味着角色只能沿着视野方向，注意不到两边，宽的视野意味着角色可以看到边上的一些东西。不同的视野范围，会导致不同的群落模式。

再来是 tooClose 函数，这个简单的不想说了。

然后就是增加一些新的变量属性和它们各自的读取器：

```

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

```

```

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

```

```

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

```

最后是测试：

```

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

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    public class FlockTest extends Sprite
    {
        private var _vehicles:Array;
        private var _numVehicles:int = 30;

        public function FlockTest()
        {
            stage.align = StageAlign.TOP_LEFT;
        }
    }
}

```

```

stage.scaleMode = StageScaleMode.NO_SCALE;

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

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

```

我们弄了一大堆转向机车，分散于舞台上，添加于数组中。每一帧都遍历数组，让它们执行群落行为，然后更新它们的位置。虽然看上去每个机车都是独自在行驶，但通过对其它机车的了解，慢慢组成了队伍。尽量多改改各项变量和机车数量，以更好的理解工作机制。

再想想数组的检测，要为每辆机车跑次循环，是不是很没效率？作为处理数组的优化方案，可能你已经想到采用二重循环甚至第一章讲的网格检测法。但是这么做就要 把群落行为的一些代码移出转向机车类，要么直接写在文档类中，要么最好是写一个群落管理器类，这就当回家作业留给各位啦。

## 总结

这章概述了基本的转向行为，从简单的寻找和避开到复杂的对象回避和群落。有趣的事情随着合并这些行为，出现复杂的动态效果而产生。再次重申，这章只能算作一个主题介绍。对于转向行为，网上书上有着大量的资料，不过常常被当作人工智能下的一部分。如果本章没啥帮助，我希望至少你能拖些程序用于自己的产品或游戏。

后面的章节将讨论等大世界和寻路。结合转向行为（或许还有高级碰撞检测）你就能作出很牛 b 的游戏啦。

### 第三章 等角投影



Figure 3-1. Zaxxon



Figure 3-2. Qbert

### 等角投影

等角投影这项运用于电脑游戏的技术，最早出现在 80 年代。它能简单有效的模拟一个三维空间，而不必花费太大的计算代价。在早期，大多数电视游戏都是单向滚动的。Zaxxon（见图 3-1）和 Qbert（见图 3-2）算是第一批商业等角游戏。

现如今，尽管前沿的 3D 游戏是像 Halo（光晕）那样的第一人称射击游戏，但等角游戏仍然相当流行，特别是在 RPG 或者 SLG 中。

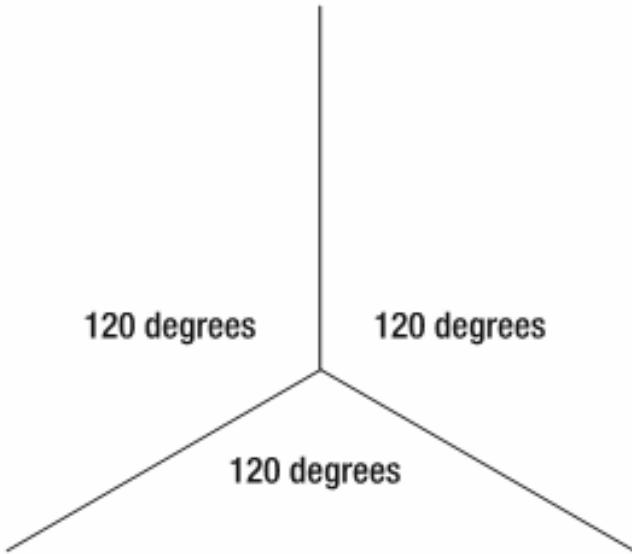
为了理解等角投影，首先要知道等角投影是什么，以及它于其它 3D 视觉投影方法的区别。

这里所说的投影，是指一个三维的物体或场景在二维平面上的表现。平面可以是张纸或是电脑屏幕。摄像机利用镜头，将前方的内容投影于胶片或者电子传感器上。人眼也是如此，将图像投影于视网膜上。

当在纸上或屏幕上发生移动时，会有许多不同的投影类型影响显示。透视投影是被使用最多的类型，在该类型中，物体离视点越远则显示的越小。这个类型在上本书的 3D 章节中有详细介绍，而它也是 Papervision3D 或一些 3D 框架中的默认选择。

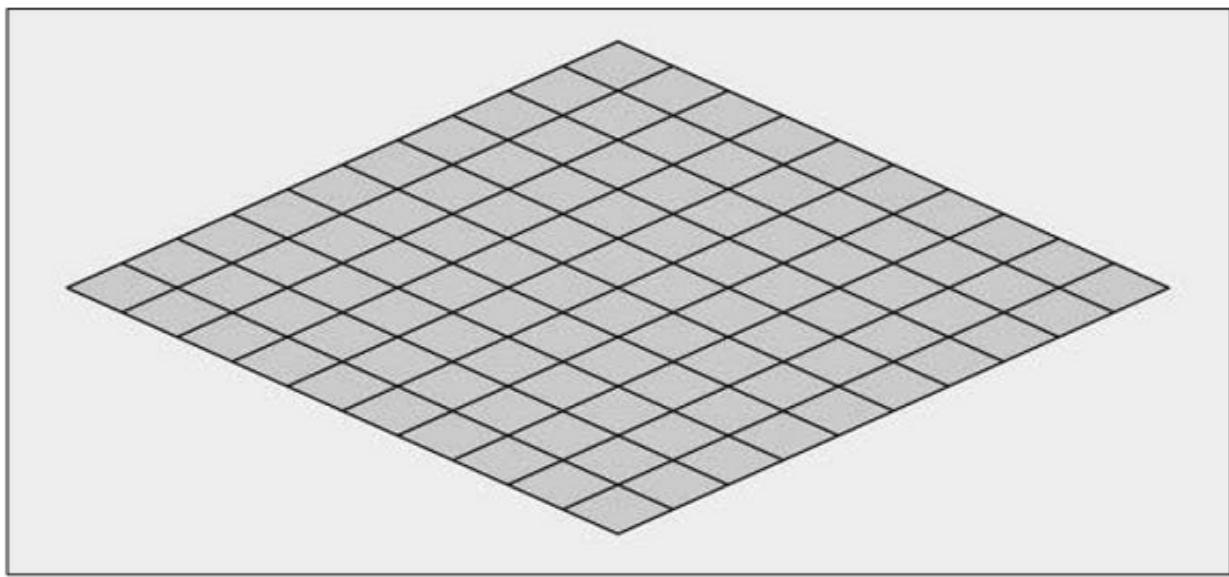
等角投影，从另一个角度讲，是一种三向投影(axonometric)。这种投影类型，并不以物体对摄像头的远近关系而对其缩放。

等角的字面意思是“等量”。实际上，也就是 x, y 和 z 轴的投影角度是相等的：120 度，如图 3-3。



**Figure 3-3.** Picture this as the corner of a room where the walls and floor meet. In isometric projection, the angles between each axis are 120 degrees.

游戏中的等角世界，几乎总是基于 tile 的，所以地面是由一个一个连续不间断的 tile 组成。游戏中的对象也是 tile。大多情况下，tile 是会被多次重用的，所以单独一个 tile 可以拼出一块完整的地面（见图 3-4），一些比较小的 tile 则用来创建出一些不同的环境（如图 3-1 的 Zaxxon）  
图 3-4

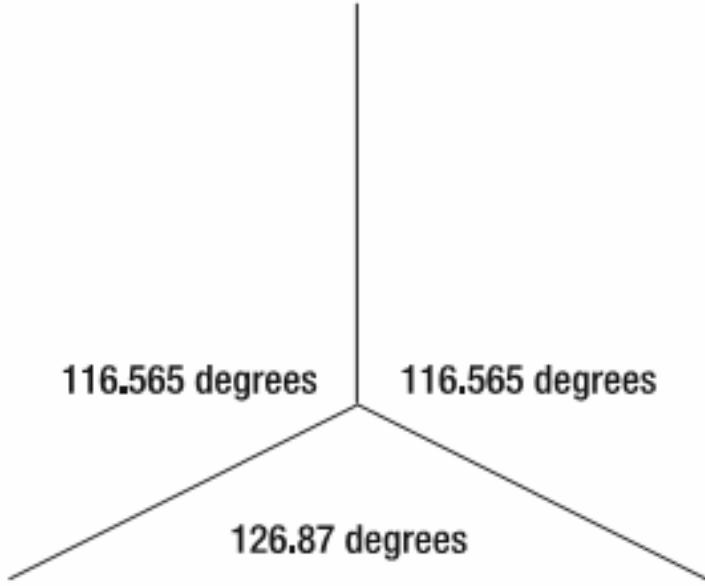


**Figure 3-4.** A single tile used to create a whole floor

### 等角 vs 二等角 (dimetric)

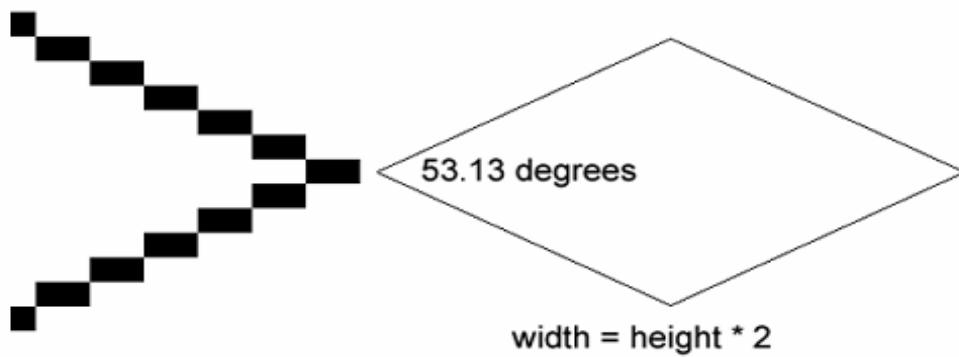
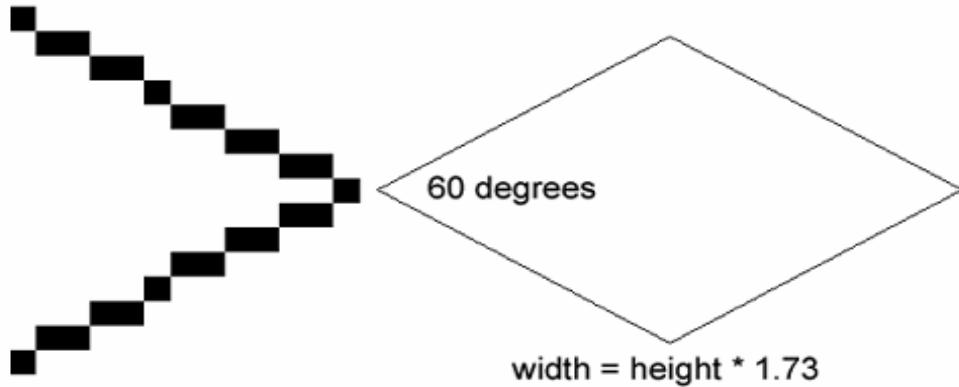
（译：dimetric 用字典翻译是四边形、四角形，我觉得字面意思不好理解，后面还有个 trimetric，字典翻译就更莫名其妙了。而其实把 isometric, dimetric, trimetric 放一起看，结尾都是 metric，有“度量、单位”的意思，而 iso, di, tri 正好又代表一，二，三，所以意思就是所有角度都一样、所有角度只有两种大小，所有角度有三种大小...）

这里有个小秘密，在宴会上提到它可能让你备受瞩目（也可能被鄙视）：几乎所有的“等角”游戏、引擎、美工等，都不是真正的等角，而是二等角。也就是说，三个轴之间存在两种不同的角度。（如图 3-5）（还有一种三向投影叫三等角，也就是三个轴之间的角度都不同）二等角投影并非必须使用图中显示的角度，不过却是最常用的角度。图 3-5



**Figure 3-5.** Dimetric projection uses two angles.

这几个怪怪的角度都差不多在 120 度左右，其实这里面对像素来说是有讲究的。让我们用一个 tile 来分别显示等角和二等角的情况。图 3-6



**Figure 3-6.** A single tile in true isometric and in dimetric projections

第一张是等角的情况，角度 60 度，宽高比例是 1.73:1。第二张是二等角的情况，虽然角度比较怪，但宽高的比例正好是 2:1，这对于创建一些系统来说就非常方便。比如，创建一个 200x100 的图像作为 tile，要比创建一个 173x100 的更简练。

这并非不重要。看上图右侧的完整版，二等角的线条要比等角的平滑，再看左边的放大版，就

知道为什么了。因为在等角中，垂直方向移动 1 像素，水平方向就要移动 1.73 像素，而这是不可能的，因为 1 像素是最小的移动单位，于是出现，有时候移动 1 像素，有时候 2 像素，这也就是看上去有锯齿的原因。在二等角中，移动总是一致的垂直 1 像素，相对应水平 2 像素。

由于这给编程和设计都带来了方便，所以每个人，包括我们都采用二等角系统。现在你理解了什么是什么，什么不是什么了吧。虽然我们知道了它不是真正的等角，但为了方便起见，我们还是把它称作等角。

### 创建等角图形

这里虽然不讲程序，但是仍有必要知道如何以等角系统创建图形。（再说最后一次，等角其实是比例 2:1，角度 26.565 的二等角）

首先，打开 Flash、Fireworks、Photoshop 或者其它什么你喜欢用的图形工具。画一个正方形，不要是长方形哦，干脆定死 100x100 吧。

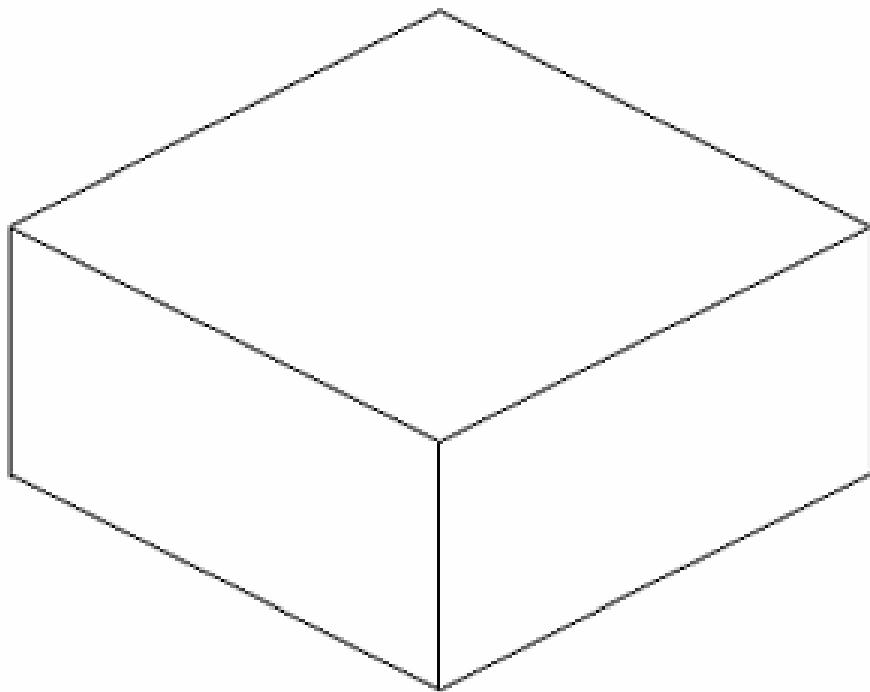
然后旋转 45 度，一定要正正好 45 度！很多旋转工具都可以按住 Shift 来旋转 45 度的。这样就得到了一个菱形。

最后，垂直缩放 50%，水平不变。

此刻，图形应该是像图 3-6 上所显示的那样了吧。这个图形尺寸大约为 141.4x70.7，正好 2:1。Fireworks 会四舍五入，所以应该是 141x71，这对我们来说已经足够用了。

如果现在你去告诉设计师所有的图形要做成 141.4x70.7，他们保不准会打你。所以，把高改成 100，宽改成 200，做成 200x100，那设计师就会比较安稳了。当然，只要保证 2:1 的比例，改大改小都可以。实际上，较小的 tile 有更好的表现力，以及碰撞检测能力（之后讨论）。

有了这个菱形，就可以往里画东西了：草地，岩石，水流，泥土，树林，宝石等等。因为这些图形都是 tile 的，所以可以被连续不断的一路铺开（这也就是为什么把它们叫 tile<瓦片>）。如果想让 tile 有立体感，只要把内容向上提一点，再向下画出几条线即可，看上去就像图 3-7。



**Figure 3-7.** An isometric box

图 3-7

从网上可以找到很多很好的关于如何创建等角图形的教程，他们会教你关于颜色、阴影、平滑等技巧。只要搜索“等角像素教程”就行了。下面我们将讨论程序部分。

### 等角形变

在等角世界的创建中,最重要(可能也是最混乱,被问及最多)的话题是:关于等角世界的 x, y, z 坐标和屏幕上的 x, y, z 坐标之间的切换。

我至少看到有五到六种不同的做法,非常的不同。有些非常准确,但效率不高。有些效率高,准确度也不错,却不是真正的 3D 形变。还有一些...够了,我不知道怎么去说它们,也不知道这些作者的点子是哪里蹦出来的,反正他们所做的都不错,只是我看不懂。

### 形变坐标与屏幕坐标

首先,让我们看看等角的真正 3D 形变是什么样子的,以便更好的理解精准和效率之间的关系。我创建了一个 SWF 文件用来显示形变,叫 IsoProject. swf。

当打开这个文件,就能发现一个同时显示顶部,正面和侧面的正方体,如图 3-8

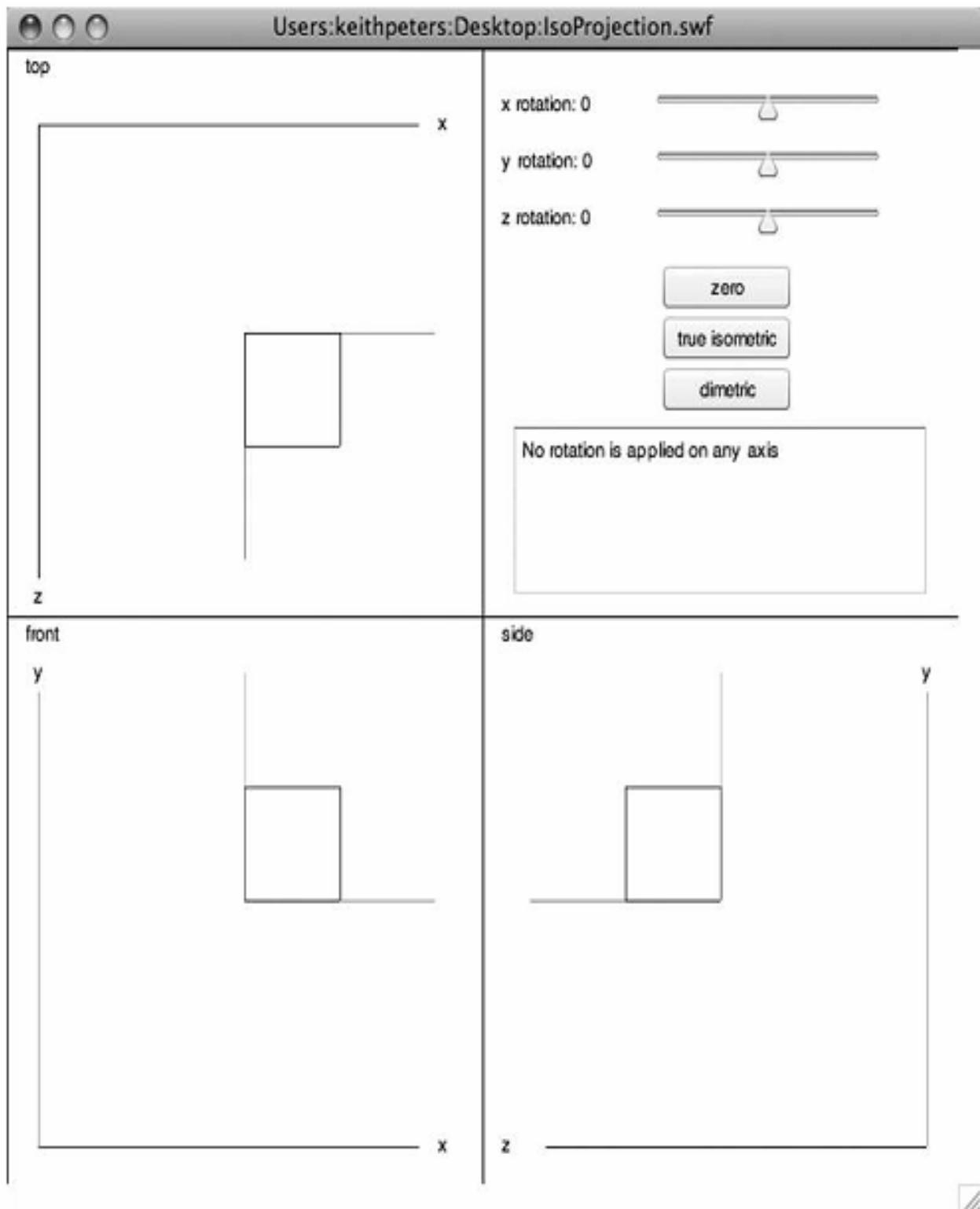


Figure 3-8. Three views of an unrotated cube

图 3-8

SWF 还展示了一个我们常用的 3D 坐标系统：x 轴从左向右，y 轴从上向下，z 轴从里向外。第一次形变是 y 轴旋转 45 度。拖动 swf 中的滑动条，再按键盘的左右键来微调到 45 度，看看变化，如图 3-9。

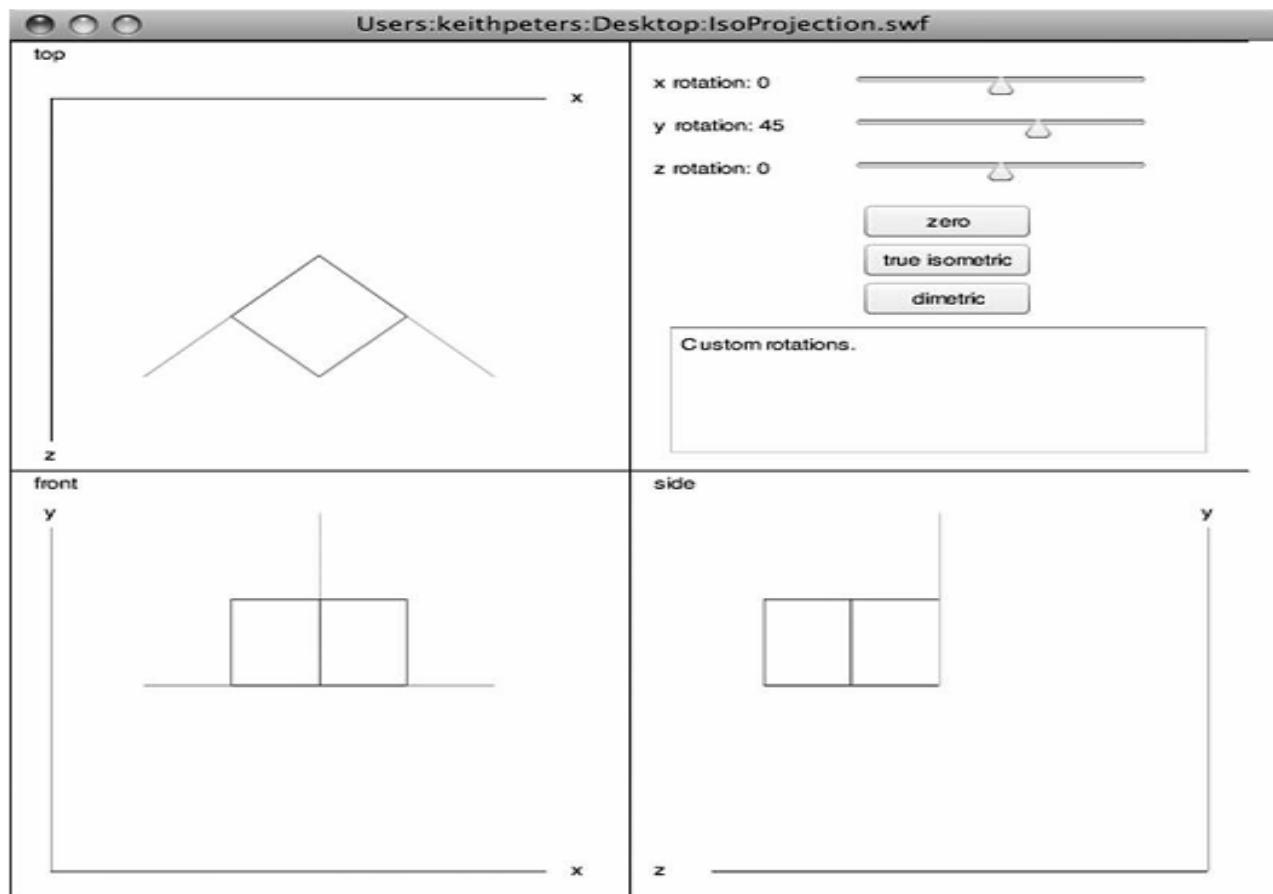


Figure 3-9. First rotation: 45 degrees on y-axis

图 3-9

第二次形变是 x 轴旋转-30 度。结果如图 3-10。

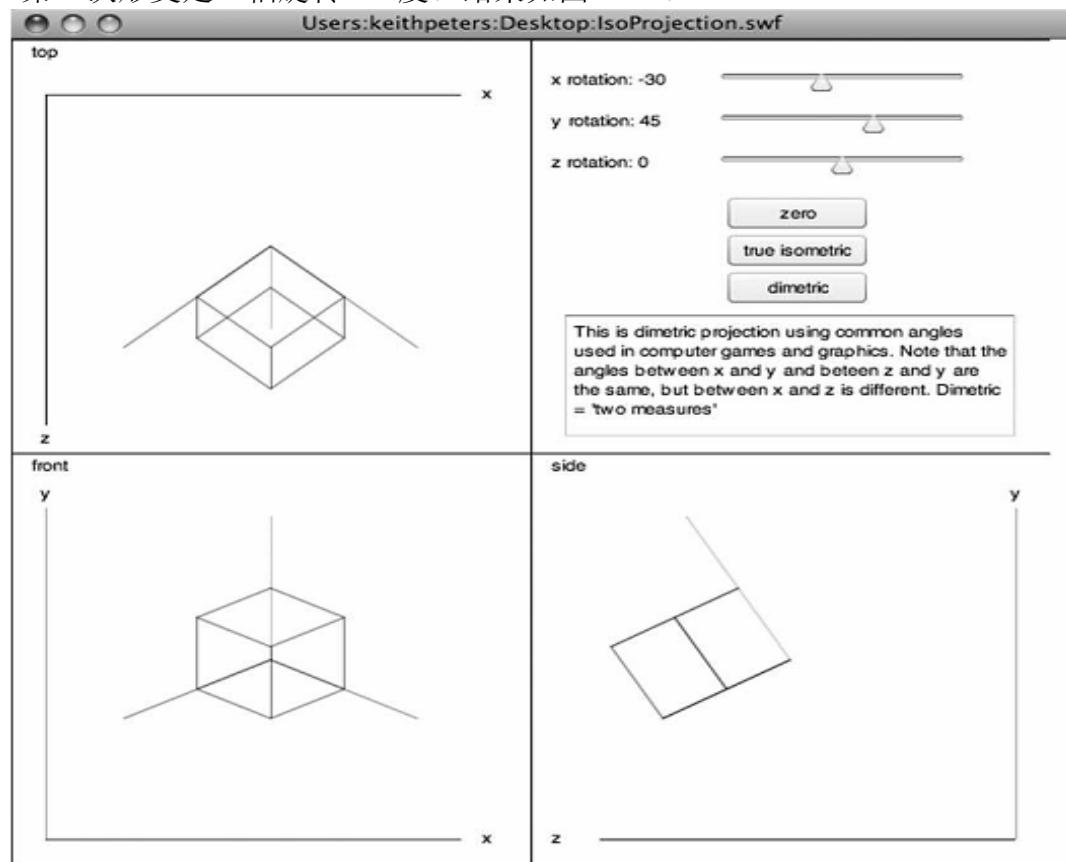


Figure 3-10. The second rotation: -30 degrees on the x-axis

图 3-10

看到正方形的顶部及底部如图 3-6 所显示的一样了吧。这个形状的宽高之比正是 2:1。点击 dimetric 按钮可以直接产生这样的形变，你也可以点击 true isometric 看看真正的等角是什么样子的。

创建这些形变的伪码大概是这样：

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

以上请查阅空间中点的旋转方式（上本书第十五章有讲）实际上，最后  $z_2$  的计算不是必须的。只有在后面讨论层深排序的时候才会用到，放着只是为了显示完整的执行过程。

尽管代码凭借着数学的保障可以安心使用，但从某些角度看仍会有些问题。首先，所有调用的函数都是三角函数，当然，可以为此做大量优化来忽略这个问题。

更严重的问题是关于 tile 的尺寸。还记得上一节讲到 100x100 的正方形吗？当时最终形成的 tile 尺寸是多少， $141.4 \times 70.7$ ？那正好，这就是我们代码所做到的。将一个 100x100 的正方形的四个顶点，通过以上公式旋转后，所得到的就是一个  $141.4 \times 70.7$  的菱形。

但要记住，当把菱形缩放到 200x100 时，宽高的缩放系数大约是 1.414，正好是 2 的开方。这里没有什么巧合，而是三角形在旋转 45 度后的特殊关系。如果有兴趣可以自己去查阅、推导。我们要做的就是，图形上如何手工缩放，代码也要以相同步骤执行缩放，否则程序出来的对象就和图形不匹配。

简化的等角变形公式如下：

```
sX = x - z;
sY = y * 1.2247 + (x + z) * .5;
```

如果对这个公式有兴趣，可以看看下面的推导过程，要不就带着信任直接跳到下一节。

从整理坐标的旋转公式开始：

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

将三角函数转化为近似值：

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

用结合律进行简化：

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

把  $z_1$  代入到  $y_1, z_2$ ：

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

$x_1, y_1, z_2$  同时放大根号 2(1.414) 倍，因为  $.707$  是根号 2 分之 1，而  $.707 * 1.414 = 1$  可以再次简化方程。

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

以上就是整个推导过程。 $z_2$  会在层深排序中使用到。

不管你是否理解了推导过程，现在我们拥有了一个优化版的函数。你只需记得，在等角世界中 100x100 的正方形，最终在 2D 平面上渲染出来是一个 200x100 的菱形。不过千万别只记住了 200x100

这个数字，如果是 50x50 的正方形，那么对应就是 100x50 的菱形。简单一点，你只需记住高度不变，宽度翻一倍即可。

有很多算法干脆去掉了 1.2247，变成：

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

当场景上所有对象的高度相同时，这么做是可以的。在更简单的系统中，所有 tile 的 y 都为 0，那公式还可以简化，如下：

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

如果对象有着不同的高度，1.2247 则不能忽略，其保证位置渲染的正确性。如执意忽略，所有对象的高度都会被一视同仁，这就让场景会有一种被挤压了的感觉。

### 屏幕坐标转换等角坐标

现在我们需要一个能在屏幕坐标和 3D 等角坐标之间切换的方法。由于屏幕坐标只是二维的，而我们需要一个三维的，但考虑大多数时候，转移是通过鼠标点击地图画面的位置来决定的。因此，只要把屏幕上的 x, y 看作等角世界里的 x, z，然后剩下的 y，也可以说是高度，当作 0 处理。所以，等角到屏幕的公式如下：

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

把 y 设为 0，解决 x, z 的就简单多了，如下：

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

以上代数转换，可不是我乱盖的。第一段是屏幕坐标 (sx, sy)，通过第二段计算得到等角世界中的 3D 坐标点 (x, y, z)。

### IsoUtils 类

理论和伪码都看够了吧，该是时候看看真正的代码了。首先，我们需要一个 3D 坐标点的类。

```
package com.friendsofed.isometric
```

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

接着是 IsoUtils 类，它用来负责等角到屏幕间的坐标切换工作。

```
package com.friendsofed.isometric {
```

```
    import flash.geom.Point;
```

```
    public class IsoUtils
```

```
{
```

```
    // 1.2247 的精确计算版本
```

```
    public static const Y_CORRECT:Number = Math.cos(-Math.PI / 6) * Math.SQRT2;
```

```
    /**
```

```
     * 把等角空间中的一个 3D 坐标点转换成屏幕上的 2D 坐标点
```

```

* @参数 pos 是一个 3D 坐标点
*/
public static function isoToScreen(pos:Point3D):Point
{
    var screenX:Number = pos.x - pos.z;
    var screenY:Number = pos.y * Y_CORRECT + (pos.x + pos.z) * .5;
    return new Point(screenX, screenY);
}
/***
* 把屏幕上的 2D 坐标点转换成等角空间中的一个 3D 坐标点, 设 y=0
* @参数 pos 是一个 2D 坐标点
*/
public static function screenToIso(point:Point):Point3D
{
    var xpos:Number = point.y + point.x * .5;
    var ypos:Number = 0;
    var zpos:Number = point.y - point.x * .5;
    return new Point3D(xpos, ypos, zpos);
}
}
}

```

注意那个魔法数字 1.2247, 是一个由三角函数计算所得的静态常量类型。这样得到的结果会更精确, 而且它也只会执行一次。即准确又高效! 为此做一个简单的测试:

```

package {
    import com.friendsofed.isometric.IsoUtils;
    import com.friendsofed.isometric.Point3D;

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

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

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

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

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

```

```

        addChild(tile);

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

}

}

```

起初创建了四个 3D 坐标点，围成一个位于 x-z 平面上的正方形。然后用 IsoUtils.isoToScreen 将其转换成四个 2D 坐标点，并绘制于场景上。

运行测试就能看到一个美丽熟悉的菱形。如果在最后加上 trace(width, height)，会看到输出的菱形大小是所期望的 200x100。

当然，仅仅靠几个 3D 坐标点和几条线是没法创造出一个华丽的等角世界的。这几个实用的方法，主要是用于等角对象的位置。下一步将创建一个描述等角对象的类。

### 等角对象

一般来说，等角图形是用像 Fireworks、Photoshop 甚至 Flash 等图形工具来创建的，它们是一些包容了花、草、树、木、水、屋、人等各种的 tile，有些可能还有几帧动画。把这些图形正确的运用于等角世界中，是我们程序员的责任。正确运用就是指在等角空间中安置和移动它们。这些听起来就像是在描述一种什么类型的东西。所以，即将介绍的就是 IsoObject 类：

```

package com.friendsofed.isometric {

    import flash.display.Sprite;
    import flash.geom.Point;
    import flash.geom.Rectangle;

    public class IsoObject extends Sprite
    {
        protected var _position:Point3D;
        protected var _size:Number;
        protected var _walkable:Boolean = false;

        // 1.2247 的精确计算版本
        public static const Y_CORRECT:Number = Math.cos(-Math.PI / 6) *Math.SQRT2;

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

        /**
         * 把当前时刻的一个 3D 坐标点转换成屏幕上的 2D 坐标点
         * 并将自己安置于该点
         */
        protected function updateScreenPosition():void
        {
            var screenPos:Point = IsoUtils.isoToScreen(_position);
            super.x = screenPos.x;
        }
    }
}

```

```

        super.y = screenPos.y;
    }

    /**
     * 自身的具体描述方式
     */
    override public function toString():String
    {
        return "[IsoObject (x:" + _position.x + ", y:" + _position.y+ ", z:" + _position.z + ")]";
    }

    /**
     * 设置/读取 3D 空间中的 x 坐标
     */
    override public function set x(value:Number):void
    {
        _position.x = value;
        updateScreenPosition();
    }
    override public function get x():Number
    {
        return _position.x;
    }

    /**
     * 设置/读取 3D 空间中的 y 坐标
     */
    override public function set y(value:Number):void
    {
        _position.y = value;
        updateScreenPosition();
    }
    override public function get y():Number
    {
        return _position.y;
    }

    /**
     * 设置/读取 3D 空间中的 z 坐标
     */
    public function set z(value:Number):void
    {
        _position.z = value;
        updateScreenPosition();
    }
    public function get z():Number
    {
        return _position.z;
    }

    /**
     * 设置/读取 3D 空间中的坐标点
     */

```

```

public function set position(value:Point3D):void
{
    _position = value;
    updateScreenPosition();
}
public function get position():Point3D
{
    return _position;
}

/**
 * 返回形变后的层深
 */
public function get depth():Number
{
    return (_position.x + _position.z) * .866 - _position.y * .707;
}

/**
 * 指定其它对象是否可以经过所占的位置
 */
public function set walkable(value:Boolean):void
{
    _walkable = value;
}
public function get walkable():Boolean
{
    return _walkable;
}

/**
 * 返回尺寸
 */
public function get size():Number
{
    return _size;
}

/**
 * 返回占用着的矩形
 */
public function get rect():Rectangle
{
    return new Rectangle(x - size / 2, z - size / 2, size, size);
}
}
}

```

这个类所关注的大都是对象的 3D 状态到屏幕状态的转换。因为它继承自 `Sprite`，所以屏幕位置可以通过 `super.x` 和 `super.y` 来赋值（`super` 是必须的，因为 `x,y` 已经被重载用于描述 3D 状态），而在 `updateScreenPosition` 函数中用到了之前看过的 `IsoUtils.isoToScreen` 函数。（译：Flash CS4 的话，需要注意 `function get/set z` 也需要 `override`）

如果想要此类正式投入使用，在实现上，有个地方可能要优化一下。比如现在，连续设置 x,y,z 的时候，`updateScreenPosition` 会被调用三次，这是很没效率的。通常的做法是用一个 `invalidate` 函数作为对象的更新函数，然后在 `enterFrame` 中执行，这是为了保证函数仅在进入下一帧前才最后执行一次。这么做的好处是，就算一帧内设置 x,y,z 一百次，真正更新屏幕的计算也只有一次。

还有一些函数，像 `depth`,`walkable`,`rect` 等涉及层深排序和碰撞检测的，将会在后面讨论。

现在，`IsoObject` 类还不包含图形。虽然可以手动画一些，但这里还是用一个类来完成该项工作：

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

        public function DrawnIsoTile(size:Number, color:uint, height:Number = 0)
        {
            super(size);
            _color = color;
            _height = height;
            draw();
        }

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

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

        /**
         * Sets / gets the color of this tile.
        */
    }
}
```

```

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

```

正如你所见，该类增加了 color 和 height 两个属性。虽然 height 在此没有用到，但可以在子类中使用，以便画出带有立体感的图形。

函数 draw 在构造函数里以及 height 和 color 每次改变时，都会被调用。其作用就是画一个带颜色的 2:1 的菱形。虽然很简单，但可以作为创建大型的等角区域的一个零件。看下面的例子：

```

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

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

    [SWF(backgroundColor=0xffffffff)]
    public class TileTest extends Sprite
    {
        public function TileTest()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

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

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

```

代码一开始创建了一个叫 world 的 sprite，用于存放所有的 tile。然后移致场景中间并离顶部一段距离。

接着一个二重循环，在循环内部创建一个大小 20，颜色缺省的 DrawnIsoTile 对象，并设置其 x

轴和 z 轴位置，然后加于 world 中。运行后，结果如图 3-11。

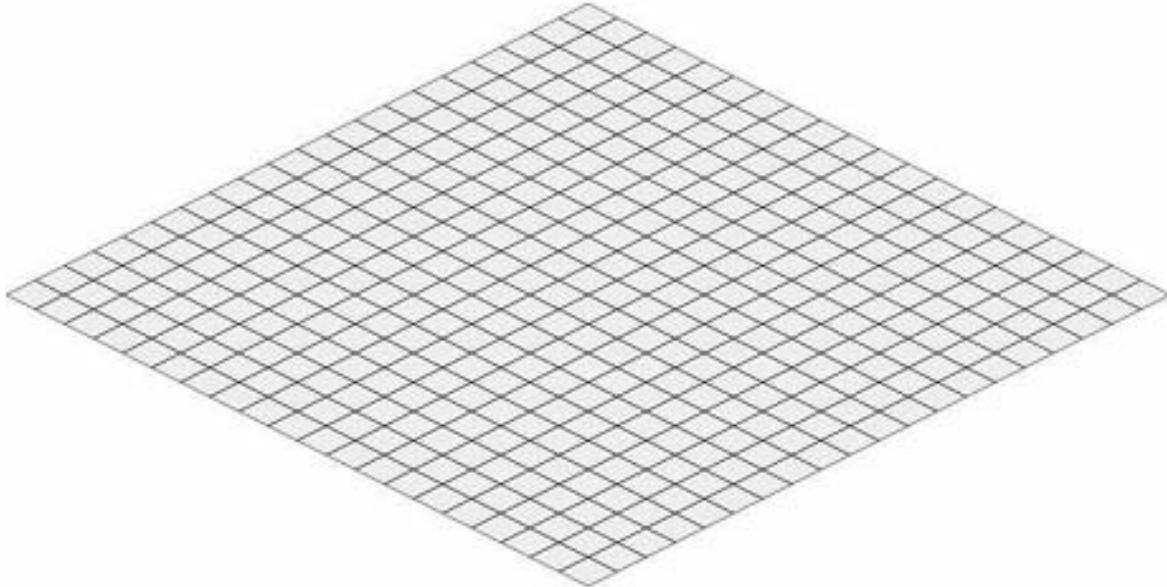


Figure 3-11. DrawnIsoTiles laid out in a grid

图 3-11

以上是一个 tile 等角世界的基本创立方法。你应该对此稍作熟悉。注意其中的二重循环次数会影响网格的大小。由于 tile 的尺寸和安置的间隔正好都是 20，所以铺的很完美，如果改变其中一个值，就会发现 tile 间会有重叠或是空隙。一般来说，总是希望 tile 是接连不断的紧挨着，但知道如何控制这些，就能以备不时只需。同样的，还可以设置一下 y 轴。在例子中，所有 tile 的 y 都是 0，所以它们位于一个平面。试着设置不同大小的 y，看看结果。不要忘了，还能改变颜色。

不用二重循环创建网格，试试单独排列 tile，或者用别的算法来排列它们。

现在让我们把 Tile 搁在一边，看看如何创建一个带有高度的 3D 等角对象（等角体）。当进入 3D 空间后，就要开始考虑光影了。一个等角体会有三个可视面。为了增强空间感，每个面的光影都会略有不同。通常，光源假设在左上或者右上方，这样一来，上一面会亮一些，左右两面会暗一些，哪个更暗取决于光源在左还是在右。下面的 DrawnIsoBox 类中，我假设光源在右上方，所以左边最暗，右边第二：

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

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

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

            var leftShadow:uint = (red * .5) << 16 |(green * .5) << 8 |(blue * .5);
            var rightShadow:uint = (red * .75) << 16 |(green * .75) << 8 | (blue * .75);
            var h:Number = _height * Y_CORRECT;
```

```

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

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

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

}
}

```

由于仅传递了一个颜色值，所以设置明暗时，要将颜色拆成红、绿、蓝三个通道，并修改其百分比。另外一种作法是，每个面用一个 sprite 或者 shape，然后通过调整各自的 colorTransform 属性来改变明暗度。这样更有弹性，我就把它当回家作业留给各位了。

还有一个重点，看下面一行代码：

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

这行是用来转换等角高度的，回忆一下等角变形公式：

```
sx = x - z;
```

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

我曾提及有些算法忽略了 1.2247，而此处我们需要它，否则 Box 就像被压扁了似的。比如我们设置大小和高度都是 20，期待出现一个正方体时，却出现一块像蛋糕一样的东西。

现在，我们要为之做个测试。当然可以把之前的 TileTest 修改一下完事，但这样太没意思了，让我们玩些更有趣的：用 IsoUtils.screenToIso 函数获取鼠标点击位置，然后动态加载一个 box 于该处。

```

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

```

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

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

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

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

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

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

```

大多数代码和 TileTest 一样，我们只是增加了一个鼠标点击事件。当鼠标点击 world，就创建一个颜色随机的 box，然后把鼠标坐标转换成等角坐标作为 box 的位置，最后将 box 加于 world 中。

```

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

```

```

}

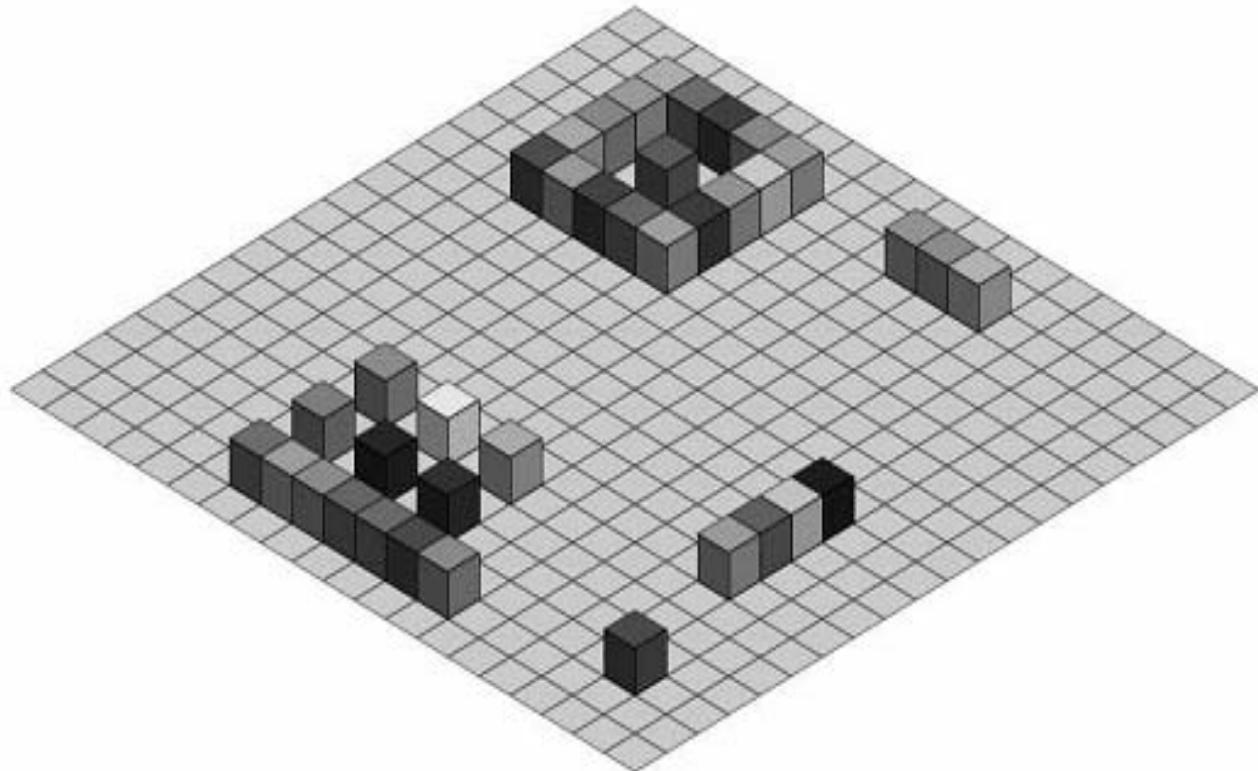
这样以来 box 是随意放置的，最好是能和所在的 tile 对齐，那么 x,y,z 就需要是 20 的倍数：
private function onWorldClick(event:MouseEvent):void
{
    var box:DrawnIsoBox = new DrawnIsoBox(20, Math.random() *
0xfffffff, 20);
    var pos:Point3D = IsoUtils.screenToIso(new Point(world.mouseX,
world.mouseY));
    pos.x = Math.round(pos.x / 20) * 20;
    pos.y = Math.round(pos.y / 20) * 20;
    pos.z = Math.round(pos.z / 20) * 20;
    box.position = pos;
    world.addChild(box);
}

```

现在，点击后创建的 box 就会贴着 tile 了，效果如图 3-12。

图 3-12

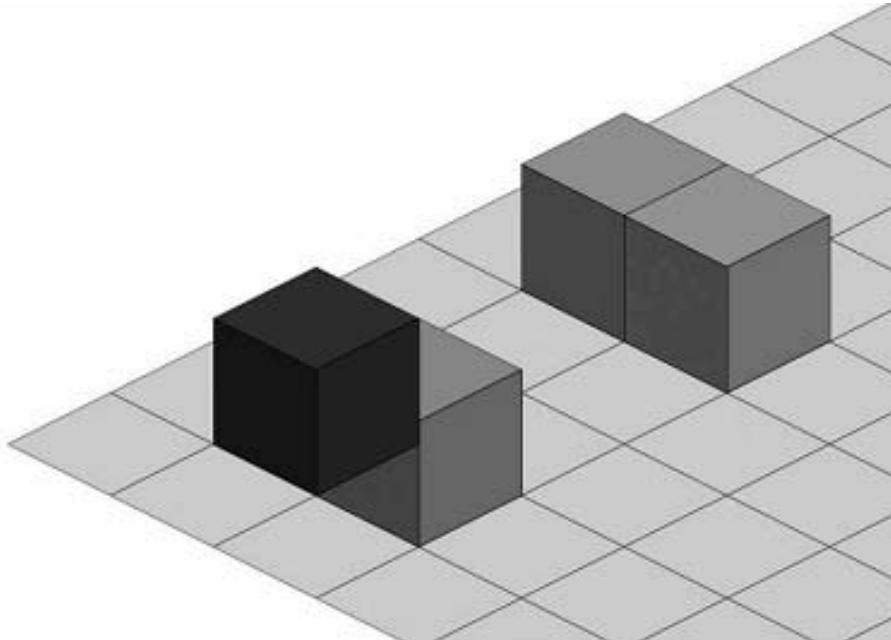
可能你已经注意到有些潜伏着的问题。首当其冲的是层深排序，下面即将对其进行讨论。



**Figure 3-12. DrawnIsoBoxes on a grid of tiles**

### 层深排序

之前的 demo 中发生了本应该在后面的 box 却盖在了前面的 box 的现象，如图 3-13，为此不要觉得是自己眼花。



**Figure 3-13. Oops! Depth problem.**

图 3-13

这是因为 `addChild` 函数总是把新加的对象置于最上方，如果 `box` 的添加顺序正巧正确，那也没什么，但像图 3-13 那样，先添加了右侧的，再添加左侧的，就不对了。为了解决这个问题，需要让每个对象的层次深度按合理的顺序进行排列。

本章开头，我提起过关于等角映射存在很多不同的解决方法，而关于等角层深排序就更糟了——每个人都有一点自己的“诀窍”。至少我能找到一打完全不同的。

有些方法能很好的支持对象高度相同的情况，但它们一碰到对象高度不统一就玩完儿了。

我不想拿个半吊子东西来讲。在我看来，这个问题最好的解决方案是直接用数学。

回到我们之前为创建等角视觉而讨论的旋转变形中来（y 轴旋转 45 度，x 轴旋转 -30 度）：

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

`x1` 和 `y1` 是用来决定变形后的对象在场景上的坐标位置。`z2` 是变形后 `z` 轴的坐标，在此，我们就可以用之来表示层深。

如果你再看一眼 `IsoObject` 类，会发现这个函数：

```
public function get depth():Number
{
    return (_position.x + _position.z) * .866 - _position.y * .707;
}
```

它就是用来计算变形后的 `z` 轴坐标。由于它并不是用来做具体的渲染工作，而只是用来做一下比较，所以就直接使用了一些近似数字 (.866, .707) 来略微提升下效率。

排序的策略是将所有 `IsoObject` 放入一个数组内，然后根据对象的 `depth` 来排序这个数组，再根据这个数组来安排显示列表内对象的顺序。每次有对象加入时都要重新排序该数组。

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

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

```

import flash.events.MouseEvent;
import flash.geom.Point;

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

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

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

        objectList = new Array();

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

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

        objectList.push(box);
        sortList();
    }

    private function sortList():void
    {
        objectList.sortOn("depth", Array.NUMERIC);
    }
}

```

```
        for(var i:int = 0; i < objectList.length; i++)
    {
        world.setChildIndex(objectList[i], i);
    }
}
```

我们创建了一个叫 `objectList` 的数组。每个 `tile` 及 `box` 都被加入到该数组，然后执行 `sortList` 函数。这个函数根据数组内元素的`'depth'`属性来进行排序，别忘了指定排序规则为 `Array.NUMERIC`，否则它会以字符串比较来排序，这样就可能导致“70”大于“100”（很多次我都郁闷在这个问题上 `debug` 了半天）。

附带说明一下，等角层深排序依赖于对象尺寸的统一（这里说的尺寸是 IsoObject 的 size 属性，即 xz 平面大小）。对象的高度不同，不会导致任何问题，但是宽度或者长度不同就不行了。通常在创建的角色中，总有一两种要小于其它一些，即使这样，也要让它们统一在同一大小的区域内。我看过去一些尝试支持不同大小的等角系统，它们都异常的复杂，除了原作者基本没人能看懂，而且还是很不稳定。或者就是对图形和位置都有严格的要求。最好的办法，是通过多个标准对象组合成一个大型对象。

之前的例子中，层深排序表现的还不错，只是偶尔会出现 tile 盖在 box 上面。这是因为当 box 放在 tile 上时，它们的 x,y,z 实际上是一样的，所以计算出来的 depth 也是一样的。在排序的时候，碰到两个一样大小的数字，就无法判断到底谁先谁后了。这个问题有两种处理方式：一种是让 tile 稍微低一点或者让 box 稍微高一点----不用太多，足够区别 depth 即可。比如用下面一行代码创建 tile：

```
tile.position = new Point3D(i*20, 0.1, j*20);
```

每个 tile 比 box 都低 0.1，即可区别 depth，又看不出有什么异样。

另一种相对复杂但更具效率的方法，是使用两个 sprite，一个放 tile，一个放 box。事实上，对 400 个纹丝不动的 tile 是没必要一直排序的，它们始终铺在 box 的下方。所以只要把它们放在位于下方的 sprite 中，然后只需对含有 box 的 sprite 进行排序即可。

package

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

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

```
[SWF(backgroundColor=0xffffffff)]  
public class DepthTest2 extends Sprite
```

```
private var floor:Sprite;  
private var world:Sprite;  
private var objectList:Array;
```

public function DepthTest2()

1  
{

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

```

floor = new Sprite();
floor.x = stage.stageWidth / 2;
floor.y = 100;
addChild(floor);

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

objectList = new Array();

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

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

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

```

代码中，`floor` 创建的位置和 `world` 一样，由于它首先被加入显示列表，所以会位于 `world` 下方。所有的 `tile` 都加在 `floor` 内，因为它们不需要排序，所以不用加入 `objectList`。

我将鼠标点击事件的监听者改为了 `stage`，因为一开始 `world` 是空的，无法接收鼠标点击。其余部分和之前一样。

下一节将以一个可重用的 `world` 类来对之前的内容做一个巩固。

## 等角世界类

由于在大多数情况下都要创建地面、世界、对象列表和排序算法，所以应该有这么个类来统一处理这些问题。

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

    public class IsoWorld extends Sprite
    {
        private var _floor:Sprite;
        private var _objects:Array;
        private var _world:Sprite;

        public function IsoWorld()
        {
            _floor = new Sprite();
            addChild(_floor);

            _world = new Sprite();
            addChild(_world);

            _objects = new Array();
        }

        public function addChildToWorld(child:IsoObject):void
        {
            _world.addChild(child);
            _objects.push(child);
            sort();
        }

        public function addChildToFloor(child:IsoObject):void
        {
            _floor.addChild(child);
        }

        public function sort():void
        {
            _objects.sortOn("depth", Array.NUMERIC);
            for(var i:int = 0; i < _objects.length; i++)
            {
                _world.setChildIndex(_objects[i], i);
            }
        }
    }
}
```

这里面多数内容在之前的例子中都做过了：创建一个地面 sprite，一个世界 sprite，一个对象列表以及排序它。添加对象的函数有两个，addChildToFloor 是把对象放进地面 sprite，而不放入对象列表内，所以不会被排序。地面被假设为平铺直叙，没有间隔的网格，所以不用排序；addChildToWorld 是把对象放进世界 sprite，以及对象列表内，所以需要排序。后面我们还会为这

个类添加有助于碰撞检测的功能。

使用 IsoWorld 类非常简单，看下面一个例子：

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

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

    [SWF(backgroundColor=0xffffffff)]
    public class WorldTest extends Sprite
    {
        private var world:IsoWorld;

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

            world = new IsoWorld();
            world.x = stage.stageWidth / 2;
            world.y = 100;
            addChild(world);

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

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

```
}
```

实际上，这个测试代码和 BoxTest 几乎一样，只是用 IsoWorld 代替了 Sprite，用 addChildToFloor 和 addChildToWorld 代替了 addChild。

### 3D 移动

在等角系统中，移动并不是一个很大的问题，特别是在有了像 IsoObject 这种能转换 3D 坐标到屏幕坐标的类以后，随意改变对象的(x,y,z)属性都可以使其正确的显示在屏幕上。唯一要记住的是，在对象每次移动后都要调用 IsoWorld.sort 函数（如果不使用 IsoWorld 就要自行写作一个层深排序算法）。

由此，实现各类移动都很简单：一般运动，加速运动，重力运动，摩擦力运动，弹性运动，缓冲运动，跳跃运动等等（这些在上本书中都有描述）。简洁起见，我在 IsoObject 里加入三个新的属性来处理速度：

```
protected var _vx:Number = 0;
protected var _vy:Number = 0;
protected var _vz:Number = 0;

并为之增加对应的 getter/setter 函数：
/***
 * 设置和读取 x 轴方向上的速度
 */
public function set vx(value:Number):void
{
    _vx = value;
}
public function get vx():Number
{
    return _vx;
}

/***
 * 设置和读取 y 轴方向上的速度
 */
public function set vy(value:Number):void
{
    _vy = value;
}
public function get vy():Number
{
    return _vy;
}

/***
 * 设置和读取 z 轴方向上的速度
 */
public function set vz(value:Number):void
{
    _vz = value;
}
public function get vz():Number
{
    return _vz;
}
```

大多数这类事件都是通过键盘驱动的。

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

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
    import flash.ui.Keyboard;

    [SWF(backgroundColor=0xffffffff)]
    public class MotionTest extends Sprite
    {
        private var world:IsoWorld;
        private var box:DrawnIsoBox;
        private var speed:Number = 5;

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

            world = new IsoWorld();
            world.x = stage.stageWidth / 2;
            world.y = 100;
            addChild(world);

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

            box = new DrawnIsoBox(20, 0xff0000, 20);
            box.x = 200;
            box.z = 200;
            world.addChildToWorld(box);

            stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
            stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
        }

        private function onKeyDown(event:KeyboardEvent):void
        {
```

```

switch(event.keyCode)
{
    case Keyboard.UP :
        box.vx = -speed;
        break;

    case Keyboard.DOWN :
        box.vx = speed;
        break;

    case Keyboard.LEFT :
        box.vz = speed;
        break;

    case Keyboard.RIGHT :
        box.vz = -speed;
        break;

    default :
        break;
}

addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onKeyUp(event:KeyboardEvent):void
{
    box.vx = 0;
    box.vz = 0;
    removeEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    box.x += box.vx;
    box.y += box.vy;
    box.z += box.vz;
}
}
}

```

这段程序非常直接。键盘监听器监听键盘压下和弹起两个事件。如果有方向键被按下，box 的 vx 或者 vz 就会有所增减。在 onEnterFrame 函数中，box 的速度会不断影响其位置。够简单吧，但别忘了再最后要加上排序。事实上，在这个例子中排序并不重要（因为世界中只有一个对象），但应该养成一个好习惯。

初始速度是 20，也就是一次移动一个 tile 大小的距离。试着把值改小一点看看移动情况。

为了展示其它类型的物理运动，这里有个升级版的 demo。

```

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

```

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

[SWF(backgroundColor=0xffffffff)]
public class MotionTest2 extends Sprite
{
    private var world:IsoWorld;
    private var box:DrawnIsoBox;
    private var shadow:DrawnIsoTile;
    private var gravity:Number = 2;
    private var friction:Number = 0.95;
    private var bounce:Number = -0.9;
    private var filter:BlurFilter;

    public function MotionTest2()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        world = new IsoWorld();
        world.x = stage.stageWidth / 2;
        world.y = 100;
        addChild(world);

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

        box = new DrawnIsoBox(20, 0xff0000, 20);
        box.x = 200;
        box.z = 200;
        world.addChildToWorld(box);

        shadow = new DrawnIsoTile(20, 0);
        shadow.alpha = .5;
        world.addChildToFloor(shadow);

        filter = new BlurFilter();

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

```

```

private function onClick(event:MouseEvent):void
{
    box.vx = Math.random() * 20 - 10;
    box.vy = -Math.random() * 40;
    box.vz = Math.random() * 20 - 10;
}
private function onEnterFrame(event:Event):void
{
    box.vy += 2;
    box.x += box.vx;
    box.y += box.vy;
    box.z += box.vz;
    if(box.x > 380)
    {
        box.x = 380;
        box.vx *= -.8;
    }
    else if(box.x < 0)
    {
        box.x = 0;
        box.vx *= bounce;
    }
    if(box.z > 380)
    {
        box.z = 380;
        box.vz *= bounce;
    }
    else if(box.z < 0)
    {
        box.z = 0;
        box.vz *= bounce;
    }
    if(box.y > 0)
    {
        box.y = 0;
        box.vy *= bounce;
    }
    box.vx *= friction;
    box.vy *= friction;
    box.vz *= friction;

    shadow.x = box.x;
    shadow.z = box.z;
    filter.blurX = filter.blurY = -box.y * .25;
    shadow.filters = [filter];
}
}
}

```

这个 demo 展示了重力，弹力和摩擦力，重力作用于 y 轴。当鼠标点击后，会给定 box 一个随机速度，而当 box 在 y 轴方向为 0 或者其它方向抵达边缘，就使其弹回。哦，还有摩擦力。

我还用 50%透明度的 DrawnIsoTile 做了一个阴影。它被放置在 floor 层里，跟随 box 的 x 轴 z 轴方向作移动。用模糊滤镜来模拟高度带来的阴影渐弱。这些都不复杂，而效果却挺好。

至今为止，在等角世界中单个对象的表现不错，那么当多个对象混在一起的时候会如何呢？请

看下一节。

## 碰撞检测

为了了解碰撞检测为何重要，先看看在之前的 MotionTest 中出现两个对象会发生什么情况：

```
var newBox:DrawnIsoBox = new DrawnIsoBox(20, 0xffffffff, 20);
newBox.x = 300;
newBox.z = 300;
world.addChildToWorld(newBox);
```

这个 box 不必移动，只需呆在原地。然后我们移动 box 去碰这个新的 box...不妙吧，直接就穿了过去。大多时候就好像移动的 box 行走在静止 box 的下方。层深排序看似失灵了一般。但是考虑一下真实情况，两个物体会同时位于同一处吗？所以排序失灵也在所难免啦。

为了处理这个情况，我们需要知道一个对象在移动时，哪里能去哪里不能去。因为 IsoWorld 类持有对象列表，所以把这个处理函数放在这个类里面最合适不过了。名字就叫它 canMove：

```
public function canMove(obj:IsoObject):Boolean
{
    var rect:Rectangle = obj.rect;
    rect.offset(obj.vx, obj.vz);

    for(var i:int = 0; i < _objects.length; i++)
    {

        var objB:IsoObject = _objects[i] as IsoObject;
        if(obj != objB && !objB.walkable && rect.intersects(objB.rect))
        {
            return false;
        }
    }
    return true;
}
```

这个函数接收一个 IsoObject 对象作为参数，返回该参数对象在下一个速度上的位置是否可行。

如果回去看一眼 IsoObject 类，会发现有一个 rect 属性。它用来描述对象所占的 x-z 面的区域。通过这个 rect 属性的 offset（偏移）函数，可以算出对象下一个速度的位置。

然后遍历对象列表，检测三个环节：

第一，检查一下遍历的对象是否为传入对象本身。你总不会关心自己是否碰到自己吧。

第二，检测遍历的对象是否标记为可通过（可通过的意思是说另一个对象能否占据相同的位置）。因为有时候一个对象很扁，可以被当做是块地板直接通过，有时候是一种升降板，升上去后允许角色可以从下面穿过的。

第三，检测偏移后的 rect 是否和其它对象的 rect 有交错，这点可以用内置的 Rectangle.intersects 函数轻易办到。

当此三个检测环节对列表中的任何一个对象为 true 时，说明不能移动，那么立即返回 false。如果完整的通过遍历，就说明不会有碰撞，可以放心的移动，那么才返回 true。在以当前速度移动前，要先使用 canMove 检测一下是否可行。下面是一个完整的例子：

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

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

```

import flash.display.StageScaleMode;
import flash.events.Event;
import flash.events.KeyboardEvent;
import flash.ui.Keyboard;

[SWF(backgroundColor=0xffffffff)]
public class CollisionTest extends Sprite
{
    private var world:IsoWorld;
    private var box:DrawnIsoBox;
    private var speed:Number = 4;

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

        world = new IsoWorld();
        world.x = stage.stageWidth / 2;
        world.y = 100;
        addChild(world);

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

        box = new DrawnIsoBox(20, 0xff0000, 20);
        box.x = 200;
        box.z = 200;
        world.addChildToWorld(box);

        var newBox:DrawnIsoBox = new DrawnIsoBox(20, 0xcccccc, 20);
        newBox.x = 300;
        newBox.z = 300;
        world.addChildToWorld(newBox);

        stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
        stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
    }

    private function onKeyDown(event:KeyboardEvent):void
    {
        switch(event.keyCode)
        {
            case Keyboard.UP :
                box.vx = -speed;
                break;
        }
    }
}

```

```

        case Keyboard.DOWN :
            box.vx = speed;
            break;

        case Keyboard.LEFT :
            box.vz = speed;
            break;

        case Keyboard.RIGHT :
            box.vz = -speed;
            break;

        default :
            break;

    }
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onKeyUp(event:KeyboardEvent):void
{
    box.vx = 0;
    box.vz = 0;
    removeEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    if(world.canMove(box))
    {
        box.x += box.vx;
        box.y += box.vy;
        box.z += box.vz;
    }
    world.sort();
}
}
}

```

如您所见，唯一有变化的地方是在移动程序处加了一个 if 语句判断是否可移动。我同时将速度作了适当的调低，这样可以让碰撞检测发生的更精确。

到此为止，介绍了等角引擎中的有关物理的基础部分，最后两节会讨论等角世界中外部图形和设计的使用。

### 使用外部图形

DrawnIsoTile 和 DrawnIsoBox 这两个类，对于测试或者做一些小游戏来说还是不错的，但在真正游戏中等角图形的设计，就要用像 Photoshop、Fireworks 甚至 Flash 这种专业图形软件来做。然后要把这些设计好的图形视作一个 IsoObject。

GraphicTile 类就是为此而生的：

```

package com.friendsofed.isometric
{
    import flash.display.DisplayObject;

```

```

public class GraphicTile extends IsoObject
{
    public function GraphicTile(size:Number, classRef:Class, xoffset:Number, yoffset:Number):void
    {
        super(size);

        var gfx:DisplayObject = new classRef() as DisplayObject;
        gfx.x = -xoffset;
        gfx.y = -yoffset;
        addChild(gfx);
    }
}

```

如您所见，该类继承自 IsoObject，除了 size 属性外，新增了一个 class 的引用和 x、y 两个偏移量。class 的引用是一个关联图形的类，通常是在嵌入元数据时就定义了，详细介绍可以参考上本书。而这里只要是继承自 DisplayObject 的类（Sprite、Shape、MovieClip、Bitmap..）它都能接受。任何时候，GraphicTile 的构造函数都会创建一个 class 引用的实例，并加入其显示列表内。然后根据给定的两个偏移量调整位置。

为了知道偏移量的具体大小，首先要看一下外部图形。我虽然不是一个优秀的等角图形设计师，但用 Fireworks 弄几个 tile 出来还是可以的。第一个简单的 tile，画布大小 40x20，然后用灌木丛的材质填充那熟悉的菱形（见图 3-14）。我不怀疑你比我做的更好。这个 png 文件和其它类文件打包在同一个目录下面。

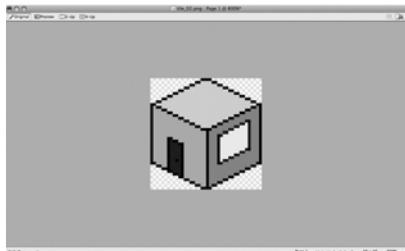


图 3-14

第二个图形稍微大一点，40x40，里面还有一个正方体，不过我为之加了窗和门。你没有猜错，这章我花在这里的时间比写作和编程要多的多。好吧，这个 png 文件我也打包贡献了（见图 3-15）

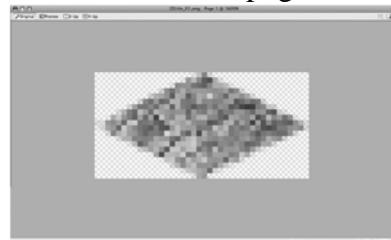


图 3-15

我改了 WorldTest 作为测试的代码。把 tile 类型改为 GraphicTile，并且嵌入了上面两张图片，作为 class 引用传递给 GraphicTile 的构造函数。

```

package
{
    import com.friendsofed.isometric.GraphicTile;
    import com.friendsofed.isometric.IsoUtils;
    import com.friendsofed.isometric.IsoWorld;
    import com.friendsofed.isometric.Point3D;

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

```

```

import flash.geom.Point;

[SWF(backgroundColor=0xffffffff)]
public class GraphicTest extends Sprite
{
    private var world:IsoWorld;

    [Embed(source="tile_01.png")]
    private var Tile01:Class;

    [Embed(source="tile_02.png")]
    private var Tile02:Class;

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

        world = new IsoWorld();
        world.x = stage.stageWidth / 2;
        world.y = 100;
        addChild(world);

        for(var i:int = 0; i < 20; i++)
        {
            for(var j:int = 0; j < 20; j++)
            {
                var tile:GraphicTile = new GraphicTile(20, Tile01, 20, 10);
                tile.position = new Point3D(i * 20, 0, j * 20);
                world.addChildToFloor(tile);
            }
        }
        stage.addEventListener(MouseEvent.CLICK, onWorldClick);
    }

    private function onWorldClick(event:MouseEvent):void
    {
        var box:GraphicTile = new GraphicTile(20, Tile02, 20, 30);
        var pos:Point3D = IsoUtils.screenToIso(new Point(world.mouseX, world.mouseY));
        pos.x = Math.round(pos.x / 20) * 20;
        pos.y = Math.round(pos.y / 20) * 20;
        pos.z = Math.round(pos.z / 20) * 20;
        box.position = pos;
        world.addChildToWorld(box);
    }
}

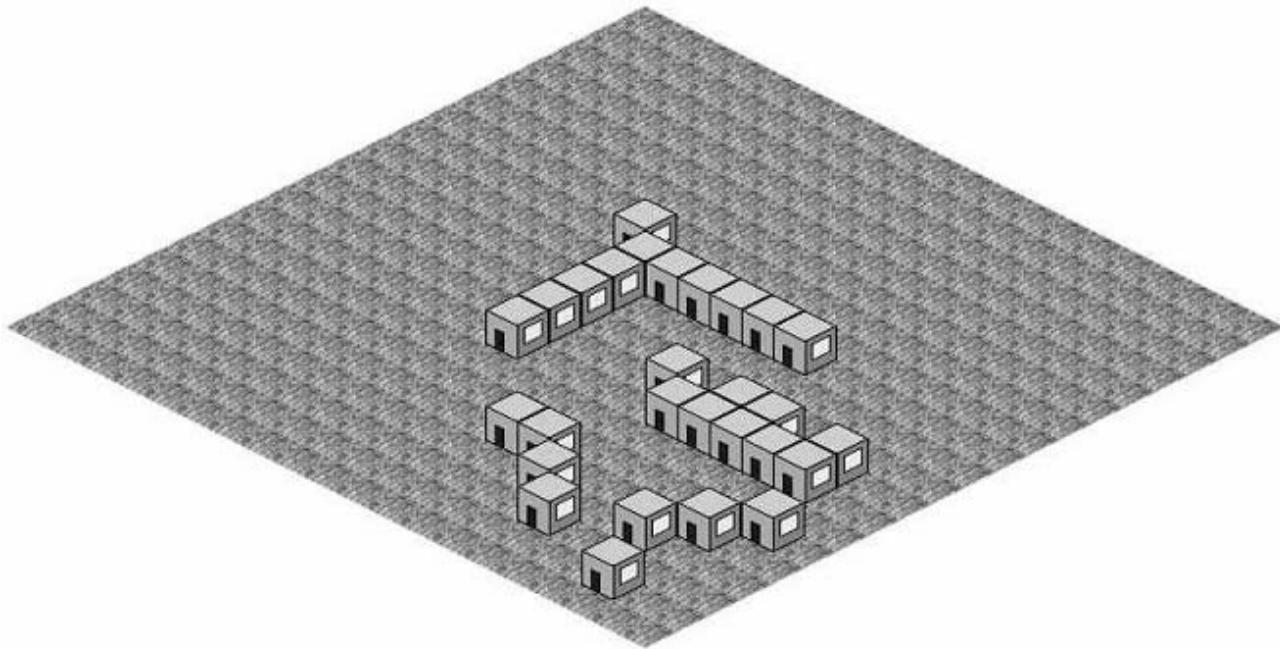
```

回来看问题：两个偏移量是怎么得来的？嗯，由于我们都是以中心点作为 tile 的注册点来创建图形的，换句话说，把一个 tile 放于屏幕的中心位置，只要设置 tile 的(x,y)为屏幕的中心点即可。

但是当图形是被嵌入的，或是别的什么实例，再被加入显示列表后，此时的注册点是左上角，它们会偏右和偏下。所以要把 tile 的注册点向左和向上偏移到中心位置。第一个 tile 是 40x20 的大小，所以要让它居中，就得向左偏移 20，向上偏移 10。另一个稍微复杂点，向右偏移 20 没什么问

题，但向上则偏移了 30，这是因为该点看上去才是正方体底面的中心点。

运行此 demo，你会发现一片茂盛的灌木丛（好吧，我承认更像是一滩沙子）点击它，就会造出别致的小房子，如图 3-16。



**Figure 3-16. GraphicTile in action**

图 3-16

现在我们有了丰富的图形元素，该需要一种方法来设计整个场景了。

### 等角地图

我们已经有了个不错的大型等角引擎。它能识别 3D 鼠标点击，支持各种物理运动、层深排序、碰撞检测以及自定义图形等等。这些足可以让我们开发一些像样的游戏了。不过还有一项很有意义的工作，是让各种类型的对象以一种简便的方式摆放在指定的位置，而不用为此去死写很多代码，以至于一旦修改就要重新编译整个项目。

这个方法叫 tile maps，是用简单的文本文件表示出各种类型的 tile 的具体位置，看个例子：

```
00000000000  
01111111110  
01000000010  
01000000010  
01000000010  
01000000010  
01000000010  
01000000010  
01000000010  
01111111110  
00000000000
```

一个 10x10 的网格，一眼望去全是 0，而内层一圈是 1。0 和 1 代表什么类型的 tile 呢，这取决于你。举个例子，0 的意思可能是一个大小 20，颜色 0xcccccc，可被通过的 DrawnIsoTile。一般，这个把地图数据保存成文本文件被程序加载后再分析，然后根据分析结果创建对应的 tile。为了包装此过程，我创建了一个 MapLoader 类，它可以为不同的角色注册不同类型的 tile，并在加载及分析完毕后发布事件。

```
package com.friendsofed.isometric  
{  
    import flash.events.Event;  
    import flash.events.EventDispatcher;  
    import flash.net.URLLoader;  
    import flash.net.URLRequest;
```

```

import flash.utils.getDefinitionByName;

public class MapLoader extends EventDispatcher
{
    private var _grid:Array;
    private var _loader:URLLoader;
    private var _tileTypes:Object;

    public function MapLoader()
    {
        _tileTypes = new Object();
    }

    /**
     * Loads a text file from the specified url.
     * @param url The location of the text file to load.
     */
    public function loadMap(url:String):void
    {
        _loader = new URLLoader();
        _loader.addEventListener(Event.COMPLETE, onLoad);
        _loader.load(new URLRequest(url));
    }

    /**
     * Parses text file into tile definitions and map.
     */
    private function onLoad(event:Event):void
    {
        _grid = new Array();
        var data:String = _loader.data;

        // first get each line of the file.
        var lines:Array = data.split("\n");
        for(var i:int = 0; i < lines.length; i++)
        {
            var line:String = lines[i];

            // if line is a tile type definition.
            if(isDefinition(line))
            {
                parseDefinition(line);
            }
            // otherwise, if line is not empty and not a comment, it's a list of tile types. add them to
grid.
            else if(!lineIsEmpty(line) && !isComment(line))
            {
                var cells:Array = line.split(" ");
                _grid.push(cells);
            }
        }
        dispatchEvent(new Event(Event.COMPLETE));
    }
}

```

```

}

private function parseDefinition(line:String):void
{
    // break apart the line into tokens
    var tokens:Array = line.split(" ");

    // get rid of #
    tokens.shift();

    // first token is the symbol
    var symbol:String = tokens.shift() as String;

    // loop through the rest of the tokens, which are key/value pairs separated by :
    var definition:Object = new Object();
    for(var i:int = 0; i < tokens.length; i++)
    {
        var key:String = tokens[i].split(":")[0];
        var val:String = tokens[i].split(":")[1];
        definition[key] = val;
    }

    // register the type and definition
    setTileType(symbol, definition);
}

/**
 * Links a symbol with a definition object.
 * @param symbol The character to use for the definition.
 * @param definition A generic object with definition properties
 */
public function setTileType(symbol:String, definition:Object):void
{
    _tileTypes[symbol] = definition;
}

/**
 * Creates an IsoWorld, iterates through loaded map, adding tiles to it based on map and
definitions.
 * @size The tile size to use when making the world.
 * @return A fully populated IsoWorld instance.
 */
public function makeWorld(size:Number):IsoWorld
{
    var world:IsoWorld = new IsoWorld();
    for(var i:int = 0; i < _grid.length; i++)
    {
        for(var j:int = 0; j < _grid[i].length; j++)
        {
            var cellType:String = _grid[i][j];
            var cell:Object = _tileTypes[cellType];
            var tile:IsoObject;
            switch(cell.type)

```

```

    {
        case "DrawnIsoTile" :
            tile = new DrawnIsoTile(size, parseInt(cell.color), parseInt(cell.height));
            break;

        case "DrawnIsoBox" :
            tile = new DrawnIsoBox(size, parseInt(cell.color), parseInt(cell.height));
            break;

        case "GraphicTile" :
            var graphicClass:Class = getDefinitionByName(cell.graphicClass) as Class;
            tile = new GraphicTile(size, graphicClass, parseInt(cell.xoffset),
parseInt(cell.yoffset));
            break;

        default :
            tile = new IsoObject(size);
            break;
    }
    tile.walkable = cell.walkable == "true";
    tile.x = j * size;
    tile.z = i * size;
    world.addChild(tile);
}
}

return world;
}

/***
 * Returns true if line contains only spaces, false if any other characters.
 * @param line The string to test.
 */
private function lineIsEmpty(line:String):Boolean
{
    for(var i:int = 0; i < line.length; i++)
    {
        if(line.charAt(i) != " ") return false;
    }
    return true;
}

/***
 * Returns true if line is a comment (starts with //).
 * @param line The string to test.
 */
private function isComment(line:String):Boolean
{
    return line.indexOf("//") == 0;
}

/***
 * Returns true if line is a definition (starts with #).
 * @param line The string to test.
*/

```

```

    */
private function isDefinition(line:String):Boolean
{
    return line.indexOf("#") == 0;
}
}
}

```

这是个比较复杂的类，但通过它可以让创建地图变得很简单。loadMap 函数是整个类的入口，它加载指定的文本文件。看个简单的地图文件：

```

// this is a comment.

# 0 type:GraphicTile graphicClass:MapTest_Tile01 xoffset:20 yoffset:
10 walkable:true
# 1 type:GraphicTile graphicClass:MapTest_Tile02 xoffset:20 yoffset:
30 walkable:false
# 2 type:DrawnIsoBox color:0xff6666 walkable:false height:20
# 2 type:DrawnIsoTile color:0x6666ff walkable:false

0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 0
0 1 0 0 0 0 0 0 0 1 0
0 1 0 3 3 3 3 0 0 1 0
0 1 0 3 2 2 3 0 0 1 0
0 1 0 3 2 2 3 0 0 1 0
0 1 0 3 3 3 3 0 0 1 0
0 1 0 0 0 0 0 0 0 1 0
0 1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0

```

以双斜杠(//)开头的是注释行，会被忽略。以井号(#)开头的是定义行，定义了要用到的 tile 类型以及构造函数的参数列表。其余的除了空行就是数据行。

当地图文件被加载，onLoad 调用所执行的分析过程如下：

- 1.把文本文件的每一行拆出来放进一个数组内。
- 2.判断每一行的具体内容是定义行、注释行、空行还是数据行。
- 3.如果是定义行，就调用 parseDefinition 函数进行分析。分析的结果保存成一个数据对象。
- 4.把数据行的数据保存在\_grid 数组内。
- 5.所有这些都好了，发布 complete 事件。然后就可以调用 makeWorld 函数来返回一个创建完毕的 IsoWorld 实例。其过程看源码吧。

测试范例：

```

package
{
    import com.friendsofed.isometric.DrawnIsoBox;
    import com.friendsofed.isometric.GraphicTile;
    import com.friendsofed.isometric.IsoWorld;
    import com.friendsofed.isometric.MapLoader;

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

```

[SWF(backgroundColor=0xffffffff)]

```

public class MapTest extends Sprite
{
    private var _world:IsoWorld;
    private var _floor:IsoWorld;
    private var mapLoader:MapLoader;

    [Embed(source="tile_01.png")]
    private var Tile01:Class;

    [Embed(source="tile_02.png")]
    private var Tile02:Class;

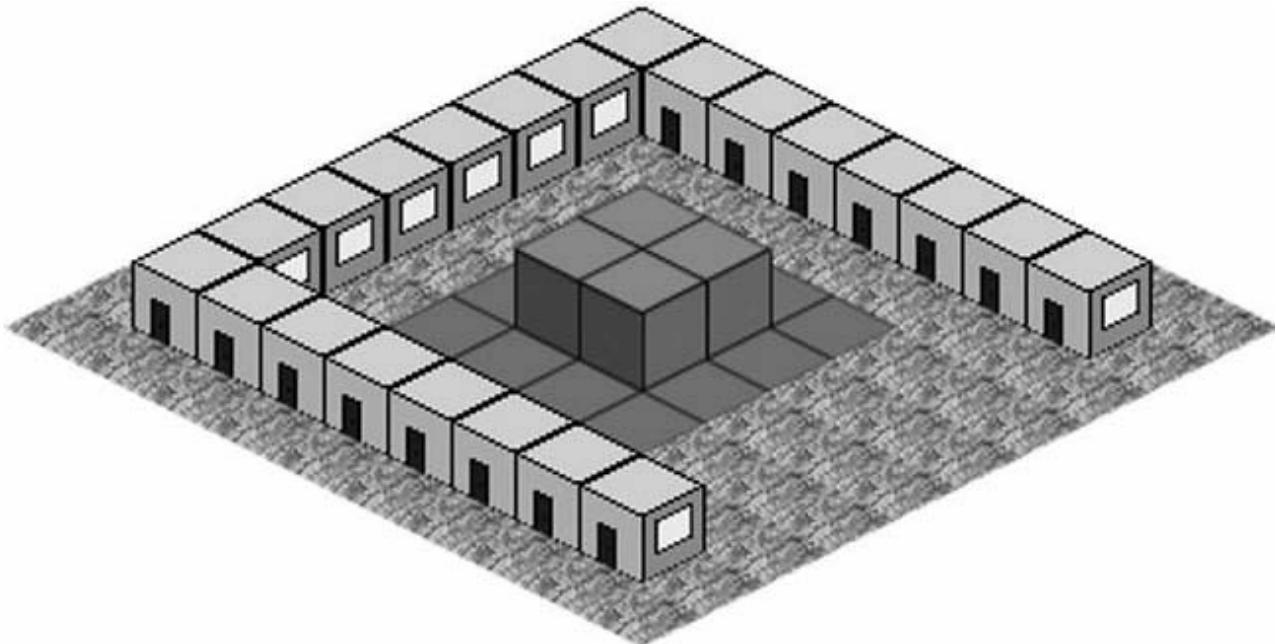
    public function MapTest()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        mapLoader = new MapLoader();
        mapLoader.addEventListener(Event.COMPLETE, onMapComplete);
        mapLoader.loadMap("map.txt");
    }

    private function onMapComplete(event:Event):void
    {
        _world = mapLoader.makeWorld(20);
        _world.x = stage.stageWidth / 2;
        _world.y = 100;
        addChild(_world);
    }
}

```

创建的 MapLoader 实例监听 complete 事件后，加载地图数据。当完成后，调用 makeWorld 并加入显示列表。最终结果如图 3-17。



**Figure 3-17.** The MapLoader class, doing its thing

### 图 3-17

一个需要注意的地方是定义 tile 的属性 graphicSymbol:

```
# 0 type:GraphicTile graphicClass:MapTest_Tile01 xoffset:20 yoffset:  
10 walkable:true
```

这里设置的类型是 MapTest\_Tile01，而在测试代码中，却写的是这样：

```
[Embed(source="tile_01.png")]  
private var Tile01:Class;
```

可能你会认为定义的时候应该写成 graphicClass:Tile01。而实际上，Tile01 在 MapTest 内是一个 private（私有）属性，我们无法在 MapLoader 内使用。但对于嵌入的元素，真正的类名格式是"类名\_属性名"，所以定义时用的是"MapTest\_Tile01"。

现在再对地图进行修改，是不需要重新编译 swf(如果你想嵌入新的图形，那就得重新编译了)。

这个类能做很多事情，我猜你一定想做点自己风格的玩意儿出来。把图形换的好看点也好，换几种摆列方式也好。总之这一切都为你提供了一个等角系统框架，让你不必为此再抓耳挠腮。

### 总结

结束啦，我相信这章概括了所有有关等角的基础概念，希望没让你感到郁闷。同时还遗留了大量的整理和优化的工作给你，但我相信你会把这些收拾的很好。

下一章，我们将研究寻路----这一 tile 世界中的高级话题，当然它百分百适用于等角 tile 世界。

## 第四章寻路

寻路这个词的意思就像看起来一样——找路线。你在 A 点，想到 B 点去，怎么到那儿？这个话题已经被开发人员广泛的研究过，我下面写的这些也不是新方法，但是它概况了寻路的基本方法和一些令人满意的標準的 AS3 解决方案。

### 寻路基础

在具体的寻路中，即便游戏不是由方块组成的时候，空间必须划分为方形网格。因此，路径被描述为从 A 到 B 我们经过的方块的集合。如果只是在两点之间简单的画一条线，这个问题就没有意义——几乎花了一整章的篇幅，当你在网上搜索“寻路”这个词的时候也就不会返回多少结果。当你把一些方格标记为不能通过时难度就增加了。游戏里角色不能继续向前或者翻越这些不能走的格子的时候就必须绕过它。如果这些格子挡在了起点和终点之间，寻路的主要任务就是怎样绕过障碍到达目标。如图 4.1：

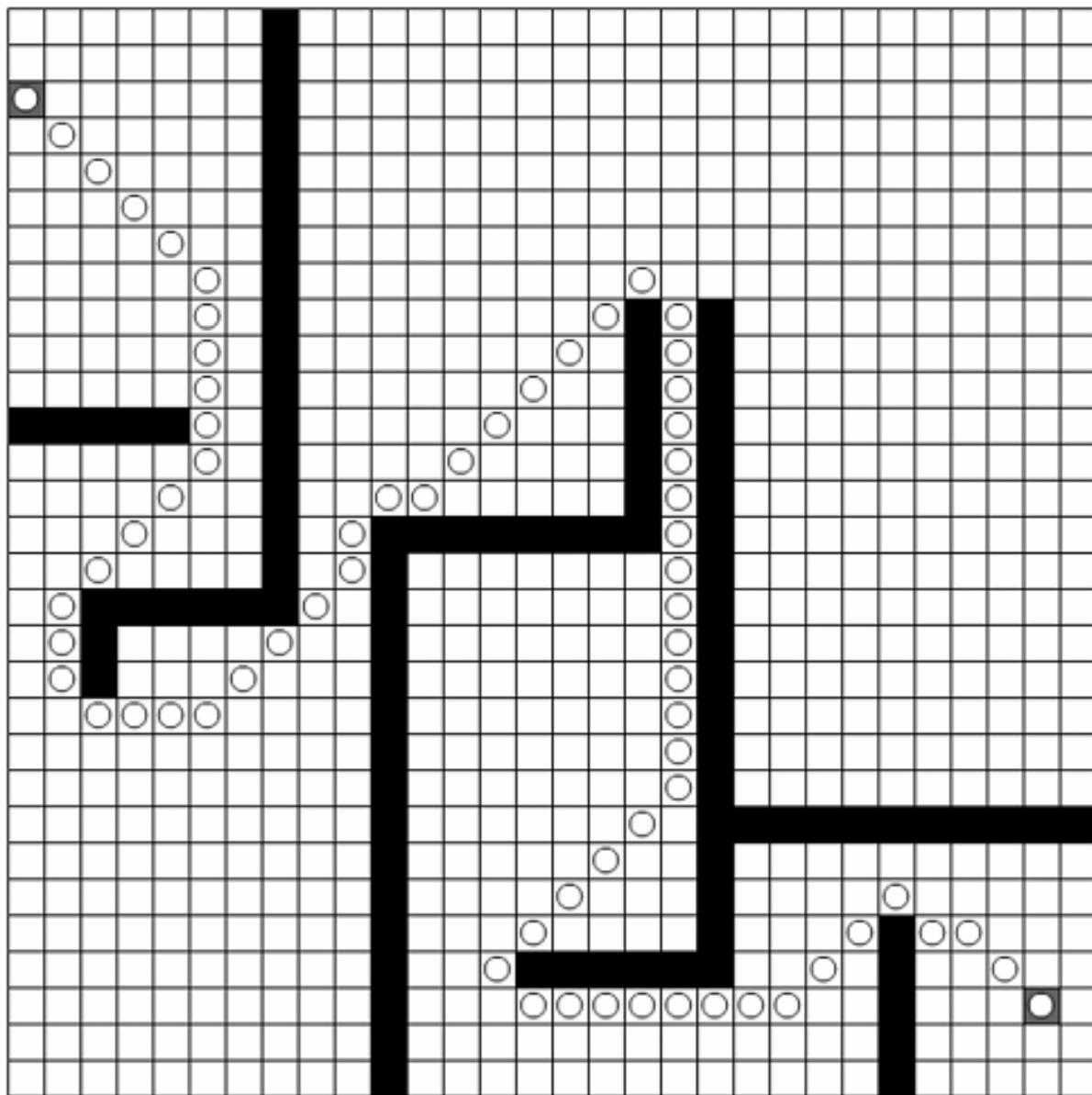


Figure 4-1. A path with barriers

当你在寻找最佳路线的时候实质还是寻路。在大多数时候，最佳路线就是最短的那条——一条由最少方格组成的路径。但是在这里，我们需要的是代价最小的路径。从一点到另一点需要一些代价。距离，是代价的一个方面——穿过两个方格比一个需要的代价“更大”——但是还有其他的代

价。在一些游戏里，比如说，路线经过沼泽或山峰比较短，但是翻山或过沼泽代价大，这条路可能比走公路代价更大，虽然走公路更长一些。举一个熟悉的例子，你办公室到家的最短的路经过市中心，但是你在高峰期回家的时候，你可能会选择走环线避免堵车。

### A-star

只要你以前看过寻路这个方面的资料，你肯定知道 A\_star。A\_star 是一种静态路网中求解最短路径最有效的方法。恰当的使用，A\_star 肯定能找出两点之间的最佳路径而且效率相当高。因此，很多人觉得 A\_star 是寻路的终极解决方案。A\_star 算法中有一些好的思想，花一些时间深入了解一下 A\_star，然后在自己的程序里使用符合自己要求的寻路算法。A-star 强大的主要原因就是它的算法很简单，本质应该是一个公式。事实上，A\_star 算法的关键部分是估价公式 (heuristic)，它整个计算过程的一部分。这个估价公式不是由 A\_star 决定的，有很多估价公式可用，每一个都有各有各的特长。实际上所有的 A\_star 算法都需要在程序语言计算特长和程序需要的基础上自定义。在这一章中我们将构建一个在 AS3 中运行的 A\_star。它不一定是最好的方法，但是它的条理清晰，将极大的方便你学习和理解算法的基本思想。融会贯通后就可以取得更大的进步。A\_star 基础简单的说，A\_star 从起点开始，分别计算出经过周围每个方格的代价。将会算出很多代价，但肯定有一条代价最小的通向终点的路径。计算的过程就是不断的把代价小的方格作为新的起点。循环往复，你最后肯定会到达终点，同时找到最佳路线。

### A\_star 运算法则

大多数讲授 A\_star 的方法是先给出理论公式，然后通过简单的插图阐明公式计算过程。这种方法很好，我也会这样讲。我们先定义一些专业术语。

**节点 (node):** 本质上就是方形网格里的某一个方格 (yujjj 注：为什么不把他们描述为方格？因为在一些时候划分的节点不一定是方形的，矩形、六角形、或其它任意形状，本书中只讨论方格)。由此可以看出，路径将会由起点节点，终点节点，还有从起点到终点经过的节点组成。

**代价 (cost):** 这是对节点优劣分级的值。代价小的节点肯定比代价大节点更好。代价由两部分组成：从起点到达当前点的代价和从这个点到终点的估计代价。代价一般由变量 f, g 和 h，具体如下。

f: 特定节点的全部代价。由  $g+h$  决定。

g: 从起点到当前点的代价。它是确定的，因为你肯定知道从起点到这一点的实际路径。

h: 从当前点到终点的估计代价。是用估价函数 (heuristic function) 计算的。它只能一个估算，因为你不知道具体的路线——你将会找出的那一条。

**估价函数 (heuristic):** 计算从当前点到终点估计代价的公式。通常有很多这样的公式，但他们的运算结果，速度等都有差异 (yujjj 注：估价公式计算的估计值越接近实际值，需要计算的节点越少；估价公式越简单，每个节点的计算速度越快)。

**待考察表 (open list):** 一组已经估价的节点。表里代价最小的节点将是下一次的计算的起点。

**已考察表 (closed list):** 从待考察表中取代价最小的节点作为起点，对它周围 8 个方向的节点进行估价，然后把它放入“已考察表”。

**父节点 (parent node):** 以一个点计算周围节点时，这个点就是其它节点的父节点。当我们到达终点节点，你可以一个一个找出父节点直到起点节点。因为父节点总是带考察表里的小代价节点，这样可以确保你找出最佳路线。

现在我们来看以下具体的运算方法：

1. 添加起点节点到待考察表

2. 主循环

a. 找到待考察表里的最小代价的节点，设为当前节点。

b. 如果当前点是终点节点，你已经找到路径了。跳到第四步。

c. 考察每一个邻节点（直角坐标网格里，有 8 个这样的节点）对于每一个邻节点：(1). 如果是不能通过的节点，或者已经在带考察表或已考察表中，跳过，继续下一节点，否则继续。

- (2). 计算它的代价
  - (3). 把当前节点定义为这个点的父节点添加到待考察表
  - (4). 添加当前节点到已考察表
3. 更新待考察表，重复第二步。
4. 你已经到达终点，创建路径列表并添加终点节点
5. 添加终点节点的父节点到路径列表
6. 重复添加父节点直到起点节点。路径列表就由一组节点构成了最佳路径

在介绍完代价的计算后，我们用图来阐释这个算法。

### 代价计算

定义中，每一个节点的代价由公式  $f=g+h$  计算。g 是从起点到当前点的代价，h 是从当前点到终点的估计代价。g 的计算相当简单：从起点到这一点走过了多少节点（在这里计算时，从任一节点到邻节点消耗为 1）。从起始节点考察每个邻节点，你应该定义他们分别  $g=1$ 。如图 4.2

1	1	1		
1	●	1		
1	1	1		

Figure 4-2. Assigning g from the start node

2	2	2	2	2
2	1	1	1	2
2	1	●	1	2
2	1	1	1	2
2	2	2	2	2

Figure 4-3. Assigning g to a subsequent node

1.4	1	1.4		
1	●	1		
1.4	1	1.4		

Figure 4-4. The cost for diagonal nodes is more than horizontal or vertical.

在下一个循环里，你可以把这些点作为当前节点，加上从当前节点到终点节点的代价。换句话说，假设当前节点是起始节点的邻节点。如果他的  $g=1$ ，当你计算每一个邻节点的时候，你将得到  $g=2$  因为它从当前节点经过一个点到达下一节点，如图 4.3

在大多数的计算中，所有的邻节点不可能一样。如果你考察同一行或列上两节点之间的距离，与对角上的节点比较，你会发现角上的会远一点。仔细计算的话，他的距离不是 1，而是  $1.414$ ， $2$  的开方。所以你应该把这个因素考虑进去（不要担心麻烦，稍后简化）。如图 4.4

$h$  是从当前点到终点的估计代价，它由估价公式计算。最简单的方法是用勾股定理计算两节点的距离。横多少，竖多少，求平方和，然后在开平方，过程如下：

$dx$ : 横向距离

$dy$ : 纵向距离

$dist = \sqrt{dx^2 + dy^2}$

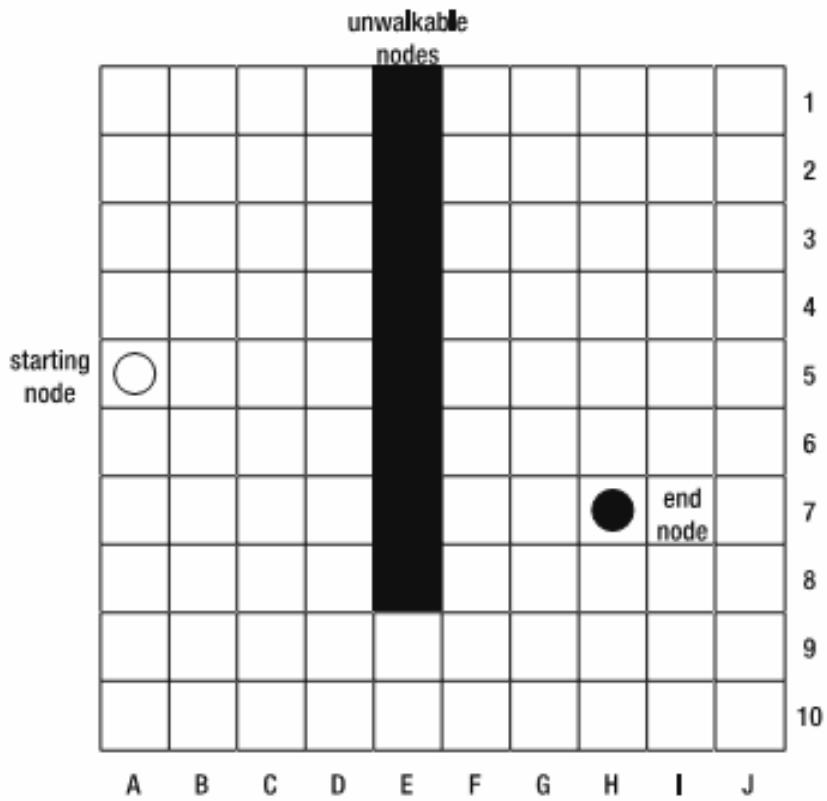
这就是  $h$ ，节点的花费就是  $f$ ，即  $g+h$

现在我们已经有了大的概念，接下来是计算的过程图解。

观察运算过程

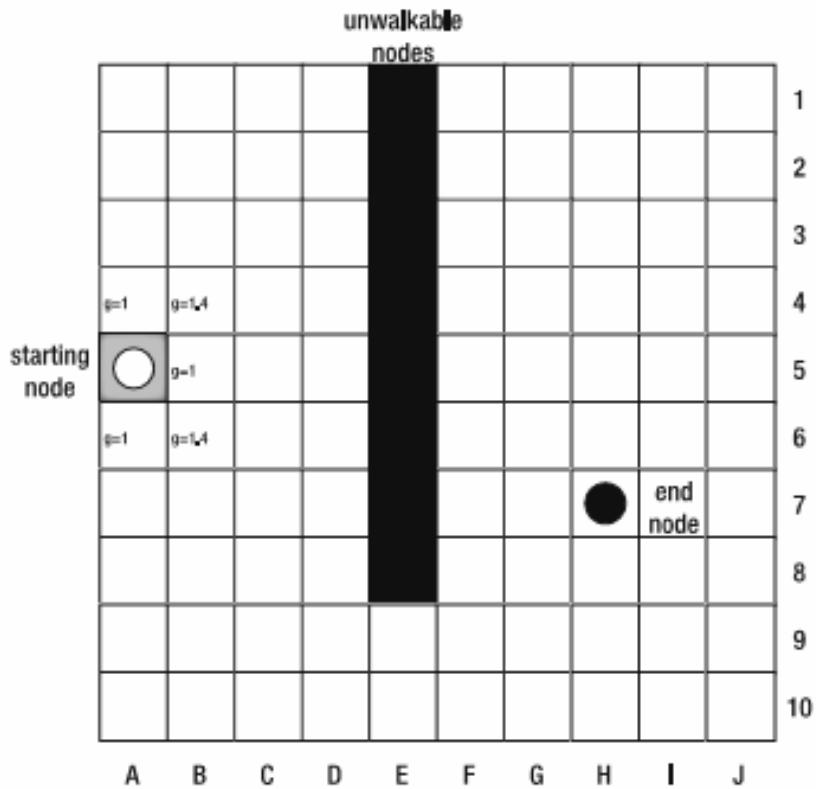
### 图解运算过程

如图 4-5，网格里一个起点节点，终点节点，和一些障碍。



**Figure 4-5.** Ready to find a path

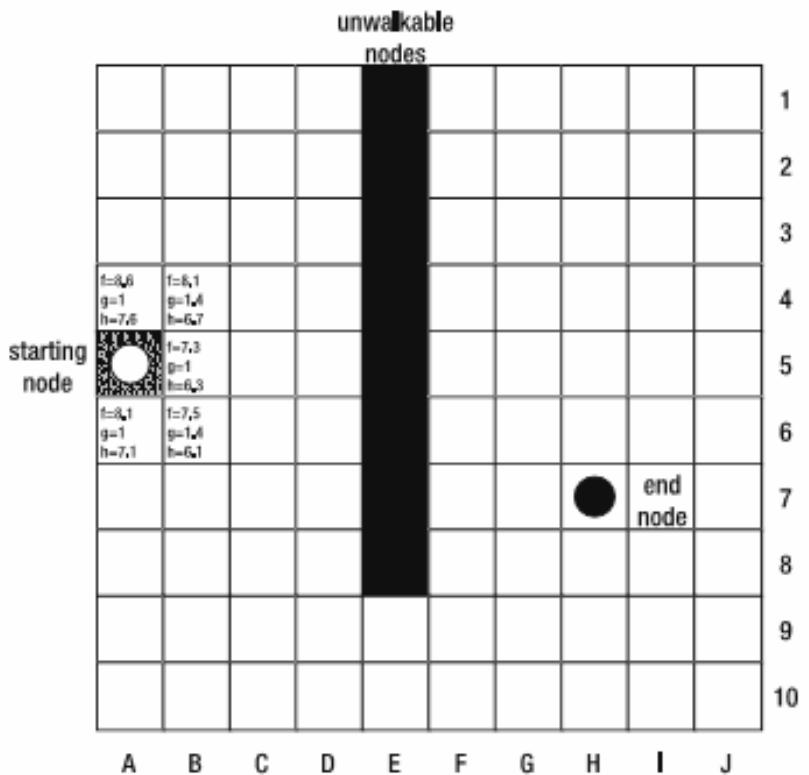
添加起点节点到待考察表，因为现在只有一个节点，所以总代价为 0. 接着检查所有的邻节点，先是每个节点的 g 值，水平竖直方向为 1，对角为 1.4. 为了简化过程，直接使用 1.4 代替 1.414. 如图 4-6



**Figure 4-6.** The g's have been assigned.

图 4-6 分配的 g 值

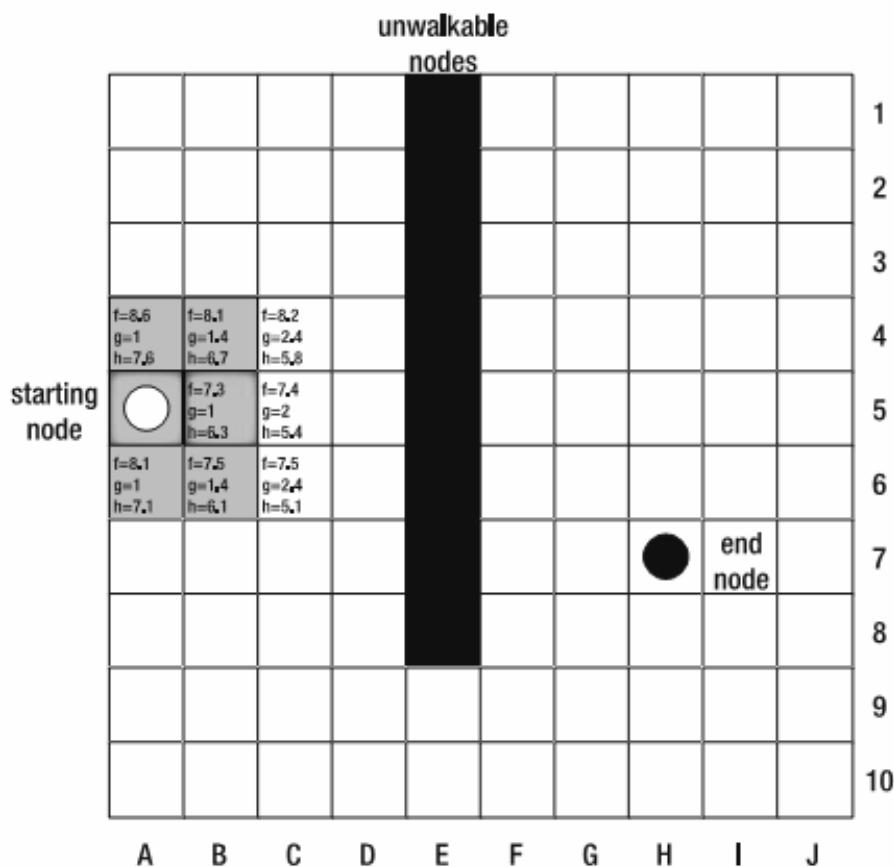
接着计算 h 值，使用直线评估公式，仅仅是到终点节点的直线距离（非像素距离，是网格坐标），原理是勾股定理，以后会讲到。



**Figure 4-7.** Total cost for each node

图 4-7 总代价图示

两部分都有了，求和得总代价，结果如 4-7 图

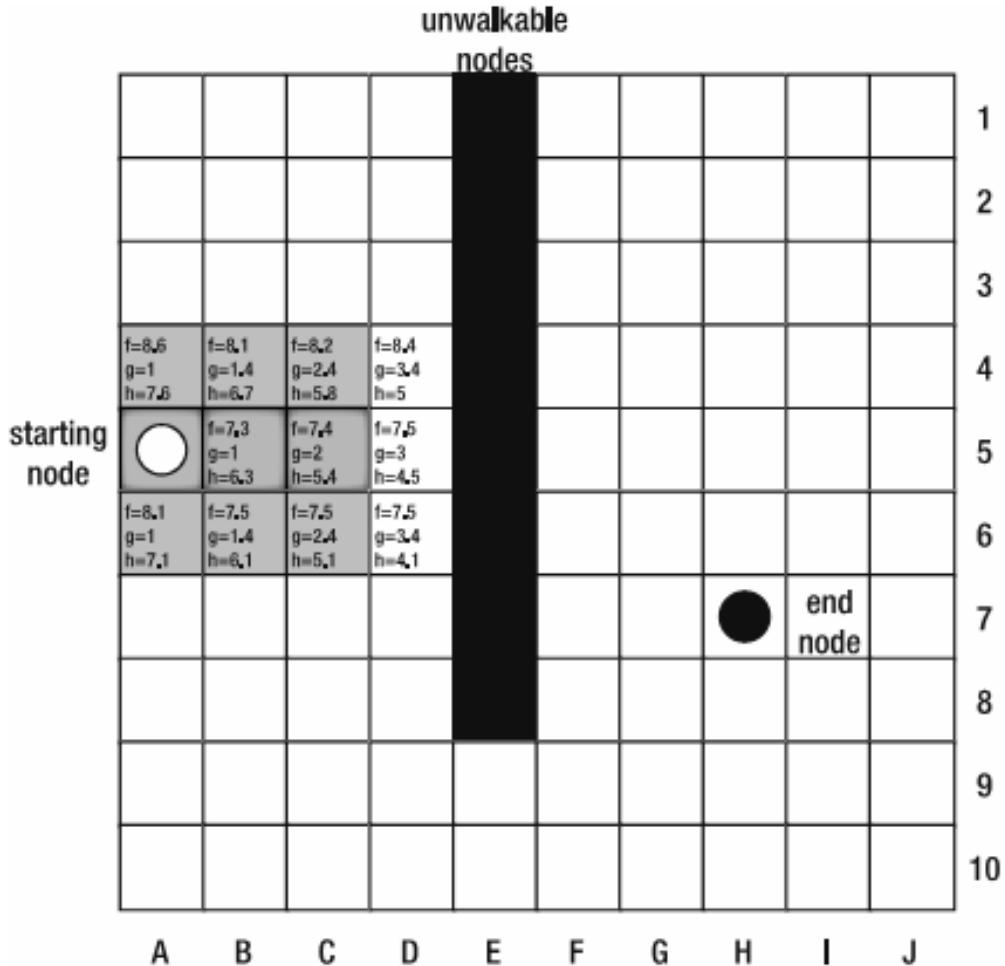


**Figure 4-8.** Round 2 of pathfinding

当前节点现在可以加到已考察列表了，同时所有计算过的节点添加待考察列表。  
然后在带考察列表里找出代价最小的节点，节点 B5 就找到了。这次有一部分节点已经在待考

察列表或已考察列表了，我们可以忽略这些，只计算剩下的。

现在把 B5 添加到已考察列表，刚刚检查的那些加到带考察列表，同样，找出最小代价节点，这次是 C5，接着算，如图 4-9

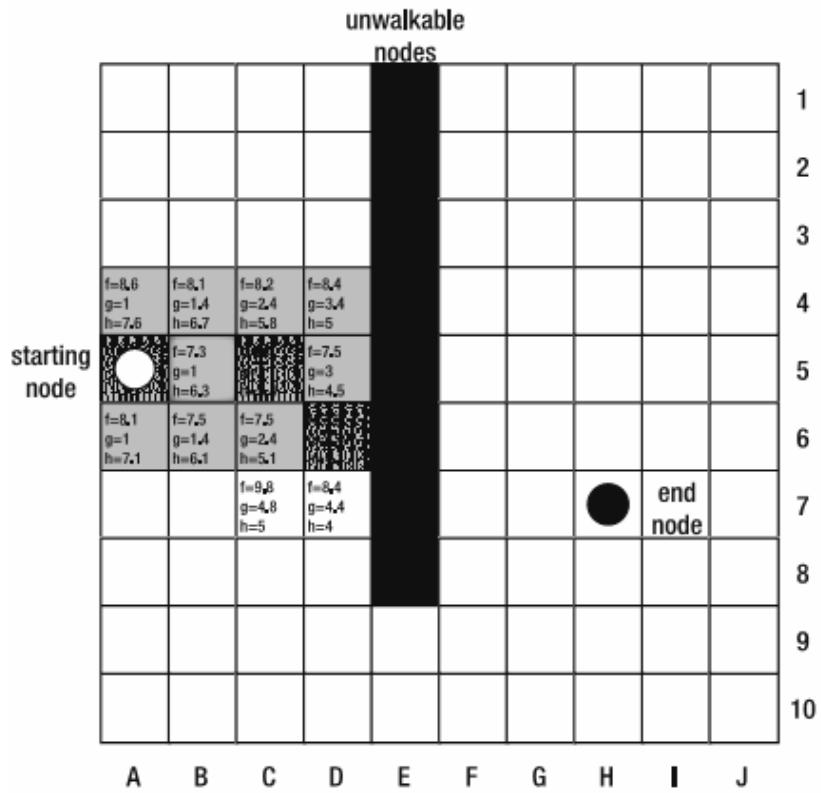


**Figure 4-9.** Round 3 of pathfinding

图 4-9 第三轮循环

注意，现在 D5 和 D6 的代价一样。你可能会说如果没有简化计算，就不会出现这种情况。实际是一切皆有可能。现在选择哪一个？不过你说也没有，取决于计算方法。你很容易就看出 D6 比较好，因为你看到了前面的障碍，但是估价程序看不见！它们是一样的。

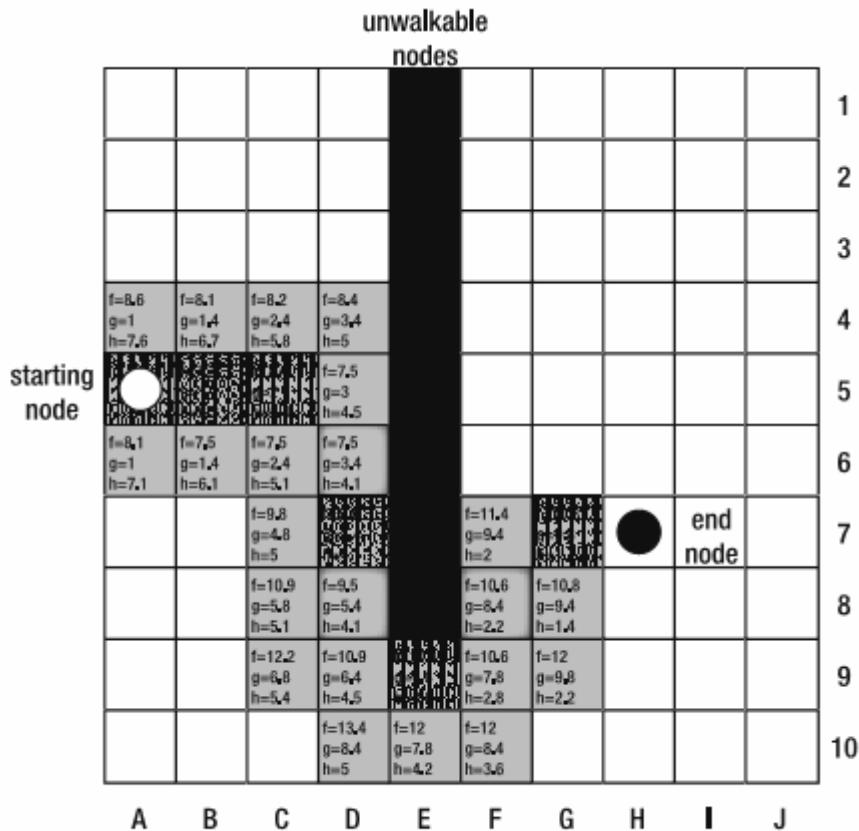
不管怎么样，我们随便选一个，就 D5。然后检查邻节点，但是没有！？能通过的都在已考察表和带考察列表里，步骤 2 的第三个条件适用了，跳到步骤 3，把 D5 添加到已考察列表，重新在待考察列表里找代价最小的节点，D6 就找到了。继续计算 D6 的邻节点。如图 4-10。



**Figure 4-10.** Pathfinding continued

图 4-10 第四轮循环

按照上面的循环过程：在带考察列表里找到代价最小的节点，设它为当前节点的所有邻节点，并把它们加到带考察列表，开始下一循环。一次又一次的循环，得到了图 4-11



**Figure 4-11.** Almost there

现在当前节点是 G7，终点就在眼前，胜利就在瞬间！

不知道你还记得不？每次设置当前节点之前，我们有一个小动作——把当前节点设为新当前节

点的父节点。现在我们只需要一个一个的找出父节点，这些节点就构成了完整的路径。

## 代码实现

在 A-star 代码之前，有必要讲一下网格和单个节点的代码结构。

先说说网格类 (Grid.as)

```
package
{
    /**
     * Holds a two-dimensional array of Nodes methods to manipulate them, start node and end node for
     * finding a path.
     */
    public class Grid
    {
        private var _startNode:Node;
        private var _endNode:Node;
        private var _nodes:Array;
        private var _numCols:int;
        private var _numRows:int;

        /**
         * Constructor.
         */
        public function Grid(numCols:int, numRows:int)
        {
            _numCols = numCols;
            _numRows = numRows;
            _nodes = new Array();

            for(var i:int = 0; i < _numCols; i++)
            {
                _nodes[i] = new Array();
                for(var j:int = 0; j < _numRows; j++)
                {
                    _nodes[i][j] = new Node(i, j);
                }
            }
        }

        ///////////////////////////////////////////////////
        // public methods
        ///////////////////////////////////////////////////

        /**
         * Returns the node at the given coords.
         * @param x The x coord.
         * @param y The y coord.
         */
        public function getNode(x:int, y:int):Node
        {
            return _nodes[x][y] as Node;
        }

        /**
         */
```

```

    * Sets the node at the given coords as the end node.
    * @param x The x coord.
    * @param y The y coord.
    */
public function setEndNode(x:int, y:int):void
{
    _endNode = _nodes[x][y] as Node;
}

/***
    * Sets the node at the given coords as the start node.
    * @param x The x coord.
    * @param y The y coord.
    */
public function setStartNode(x:int, y:int):void
{
    _startNode = _nodes[x][y] as Node;
}

/***
    * Sets the node at the given coords as walkable or not.
    * @param x The x coord.
    * @param y The y coord.
    */
public function setWalkable(x:int, y:int, value:Boolean):void
{
    _nodes[x][y].walkable = value;
}

```

```

///////////////////////////////
// getters / setters
/////////////////////////////

```

```

/***
    * Returns the end node.
    */
public function get endNode():Node
{
    return _endNode;
}

/***
    * Returns the number of columns in the grid.
    */
public function get numCols():int
{
    return _numCols;
}

/***
    * Returns the number of rows in the grid.
    */

```

```

    */
    public function get numRows():int
    {
        return _numRows;
    }

    /**
     * Returns the start node.
     */
    public function get startNode():Node
    {
        return _startNode;
    }

}

}

```

在构造函数里，传入你需要的行数和列数，生成节点的数组。我们接下来看 Node 类。你可以用 xy 坐标指定起始节点和结束节点。同样，你也可以指定特定的节点是否可以通过。接下来，你就可以得到一个含有起点节点，终点节点和其它特殊节点的索引。就像在网格里读行数和列数一样。

grid 类只不过是保存信息的网格——没有实际意义。我们将重新创建一个类。我们先看一下 Node 类。

```

package {
    /**
     * Represents a specific node evaluated as part of a pathfinding algorithm.
     */
    public class Node
    {
        public var x:int;
        public var y:int;
        public var f:Number;
        public var g:Number;
        public var h:Number;
        public var walkable:Boolean = true;
        public var parent:Node;
        public var costMultiplier:Number = 1.0;
        public function Node(x:int, y:int)
        {
            this.x = x;
            this.y = y;
        }
    }
}

```

Node 类是一个简单的用来保存方形网格属性的数据对象。它没有自己的行为，所以我们只定义了它的公共属性。注意，在这一点上，所有的节点是一样的，所以唯一的代价就是路径的长度。以后你可以看到怎样为不同的节点定义不同的权值。

接下来我们需要寻路算法的主类。就是 Astar 类。我们一段一段的看：首先，我们定义一些属性，接着是定义构造函数。

```

package
{
    public class AStar

```

```

{
    private var _open:Array;
    private var _closed:Array;
    private var _grid:Grid;
    private var _endNode:Node;
    private var _startNode:Node;
    private var _path:Array;
    //    private var _heuristic:Function = manhattan;
    //    private var _heuristic:Function = euclidian;
    private var _heuristic:Function = diagonal;
    private var _straightCost:Number = 1.0;
    private var _diagCost:Number = Math.SQRT2;
    public function AStar()
    {
    }
}

```

分别是存储待考察表/已考察表、网格、开始节点和终点节点的数组；一个存储路径的数组，实质是节点的列表；一个估价方法（heuristic）属性。后面会详细介绍这几种常见的估价方法。你可以选择一个你喜欢的。你可能还会发明出一个在不同估价方法之间准确选择的代码，比如说 setHeuristic 方法，我会把这个留给你。最后定义通过直线和对角线的代价权值。

接下来是寻路方法。

```

public function findPath(grid:Grid):Boolean
{
    _grid = grid;
    _open = new Array();
    _closed = new Array();
    _startNode = _grid.startNode;
    _endNode = _grid.endNode;
    _startNode.g = 0;
    _startNode.h = _heuristic(_startNode);
    _startNode.f = _startNode.g + _startNode.h;
    return search();
}

```

这个方法创建了一个空的待考察表/已考察表，然后从`_grid` 中获取起点，终点节点值。在计算出起点的代价后，跳到`search` 方法开始循环，直到终点节点，返回路径。

我们来看一下起点的代价是怎么计算出来的。因为`g` 值的定义是从起点到当前点的消耗，这时起点就是当前点，让当前节点的`g` 值为 0。接下来选用任一一种估价方法，传入起点，将会返回到终点的估价，这就是`h`。最后`h+g` 得到这个点的总代价。

接下来是这个类的关键，`search` 方法。它的作用是挨个计算起点节点一直到终点节点，计算出最佳路径：

```

public function search():Boolean
{
    var node:Node = _startNode;
    while(node != _endNode)
    {
        var startX:int = Math.max(0, node.x - 1);
        var endX:int = Math.min(_grid.numCols - 1, node.x + 1);
        var startY:int = Math.max(0, node.y - 1);
        var endY:int = Math.min(_grid.numRows - 1, node.y + 1);
        for(var i:int = startX; i <= endX; i++)
        {
            for(var j:int = startY; j <= endY; j++)
            {

```

```

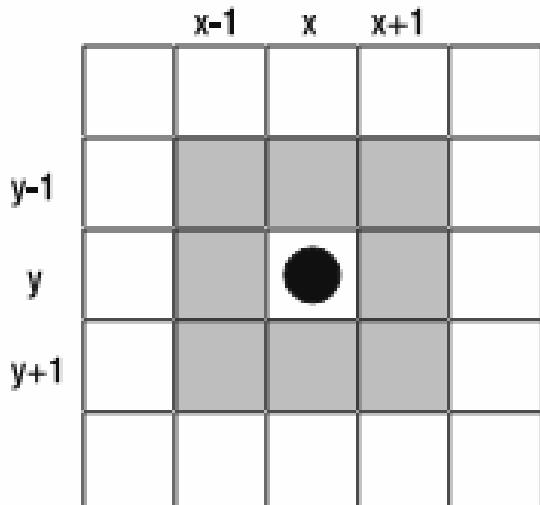
var test:Node = _grid.getNode(i, j);
if(test == node ||
    !test.walkable ||
    !_grid.getNode(node.x, test.y).walkable ||
    !_grid.getNode(test.x, node.y).walkable)
{
    continue;
}
var cost:Number = _straightCost;
if(!((node.x == test.x) || (node.y == test.y)))
{
    cost = _diagCost;
}
var g:Number = node.g + cost * test.costMultiplier;
    var h:Number = _heuristic(test);
var f:Number = g + h;
if(isOpen(test) || isClosed(test))
{
    if(test.f > f)
    {
        test.f = f;
        test.g = g;
        test.h = h;
        test.parent = node;
    }
}
else
{
    test.f = f;
    test.g = g;
    test.h = h;
    test.parent = node;
    _open.push(test);
}
}
for(var o:int = 0; o < _open.length; o++)
{
}
_closed.push(node);
if(_open.length == 0)
{
    trace("no path found");
    return false
}
_open.sortOn("f", Array.NUMERIC);
node = _open.shift() as Node;
}
buildPath();
return true;
}

```

我们首先将起点节点设为当前节点，开始一次次的循环，当当前节点为终点节点时，整个计算

结束，退出循环。

这个循环里先是一个嵌套的 for 循环，用来计算当前节点的所有邻节点。如图 4-12



**Figure 4-12.** The x, y index values of the nodes surrounding the current node

首先找到当前节点的 x, y 值（不是实际意义的 xy 坐标，是节点在网格中的行列数），然后分别从 x-1 到 x+1, y-1 到 y+1。

```
var startX:int = Math.max(0, node.x - 1);
    var endX:int = Math.min(_grid.numCols - 1, node.x + 1);
    var startY:int = Math.max(0, node.y - 1);
    var endY:int = Math.min(_grid.numRows - 1, node.y + 1);
    for (var i:int = startX; i <= endX; i++)
    {
        for (var j:int = startY; j <= endY; j++)
        {
```

我们同时不需要检查网格以外的节点，如图 4-7. 通过 Math.min 和 Math.max 确保了检查的节点永远在网格里面。

对于每一个节点来说，如果它是当前节点或不可通过的，就忽略它，直接跳到下一个：

```
var test:Node = _grid.getNode(i, j);
if (test == node || !test.walkable) continue;
```

经过前面的这些，留下的就是需要计算的节点。首先计算从开始节点到测试节点的代价 (g)，方法是当前节点的 g 值加上当前节点到测试节点的代价。简化以后就是水平、竖直方向直接加上 `_straightCost`，对角加上 `_diagCost.h` 通过估价函数计算，然后 g 和 h 求和，得到 f (总代价)。

```
var cost:Number = _straightCost;
if (!((node.x == test.x) || (node.y == test.y)))
{
    cost = _diagCost;
}
var g:Number = node.g + cost * test.costMultiplier;
var h:Number = _heuristic(test);
var f:Number = g + h;
```

下面这个部分有一点小技巧，之前我们并没有谈到。开始的时候，我说过如果一个节点在待考察表/已考察表里，因为它已经被考察过了，所以我们不需要再考察。不过这次计算出的结果有可能小于你之前计算的结果（比如说，上次计算时是对角，而这次确是上下或左右关系，代价就小一

些)。所以，就算一个节点在待考察表/已考察表里面，最好还是比较一下当前值和之前值之间的大小。具体做法是比较测试节点的总代价与以前计算出来的总代价。如果以前的大，我们就找到了更好的节点，我们就需要重新给测试点的 f, g, h 赋值，同时，我们还要把测试点的父节点设为当前点。这就要我们向后追溯。

```
if(isOpen(test) || isClosed(test))    {
    if(test.f > f)  {
        test.f = f;
        test.g = g;
        test.h = h;
        test.parent = node;
    }
}
```

如果测试节点不再待考察表/已考察表里面，我们只需要赋值给 f, g, h 和父节点。然后把测试点加到待考察表，然后是下一个测试点，找出最佳点。

```
else    {
    test.f = f;
    test.g = g;
    test.h = h;
    test.parent = node;
    _open.push(test);
}
```

现在我们检查了当前点所有有效测试点。没有什么还需要计算了，所以我们把当前点加到已考察表里：

```
_closed.push(node);
```

接下来我们需要找到下一个当前点从而循环这个过程直到结束。检查待考察表里面代价最小的节点。在这之前，我们应该先检查待考察表里面有没有节点。没有就意味着没有可行的路径：

```
if(_open.length == 0)
{
    trace("no path found");
    return false
}
```

在执行时，返回真就是有路，返回假就是没路。这个检查可以决定寻路进程的结束与否。如果待考察表上面有很多节点，我们需要找到最优的（总代价最小的那个）。可以按列表里元素的 f 值排序，最后的那个就是最好的。

```
_open.sortOn("f", Array.NUMERIC);
node = _open.shift() as Node;
```

这时这个循环到了结尾，我们有了新的当前点，while 循环将会判断当前点是不是终点节点。如不是，它将进行下一轮循环直到不能找到路径或到达终点节点。

当我们到达结束节点的时候，我们运行 buildPath 方法，并返回真。下面是 buildPath 方法：

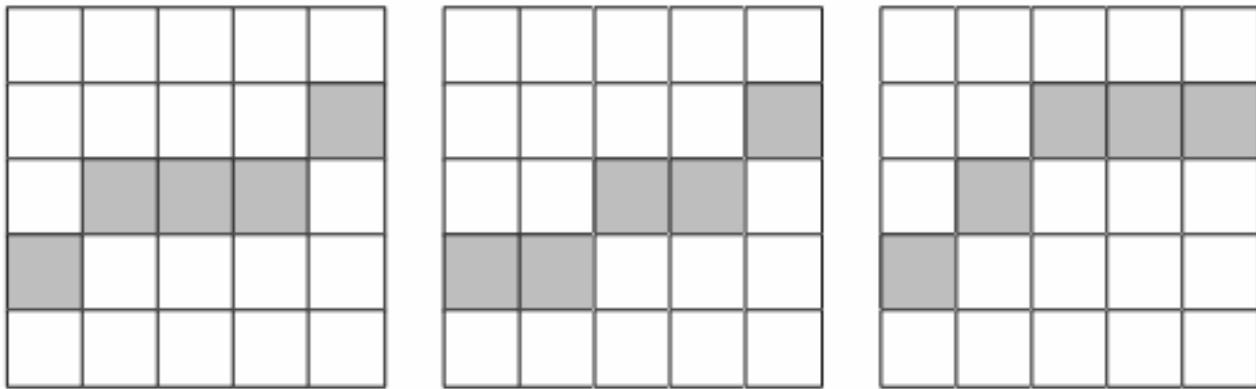
```
private function buildPath():void
{
    _path = new Array();
    var node:Node = _endNode;
    _path.push(node);
    while(node != _startNode)
    {
        node = node.parent;
        _path.unshift(node);
    }
}
```

这个方法新建一个数组并把终点节点放里面。接着添加终点节点的父节点进来，用 unshift

而非 push，把新的节点添加到整个数组的最前面，可以保证当我们结束的时候，起点是这个数组的第一个值，最后一个值是终点节点。这时，路径数组保存了从起点到终点的所有节点的有序集合，我们的目的达到了！使用 AStar 类的代码就可以检查返回值从而知道到底是否找到了路径。如果有，把路径数组赋值给取路径的变量。

如果你已经知道 Astar 类的估价方法，跳过下面这部分到下一章。现在看看不同的估价方法。**常见的 Astar 估价公式**

不管你使用那一个评估方法，Astar 都会给你一个最佳路径。注意，最佳路径不一定是特定的。几乎在每个例子里，都会有几条代价相同的路径。如下图：



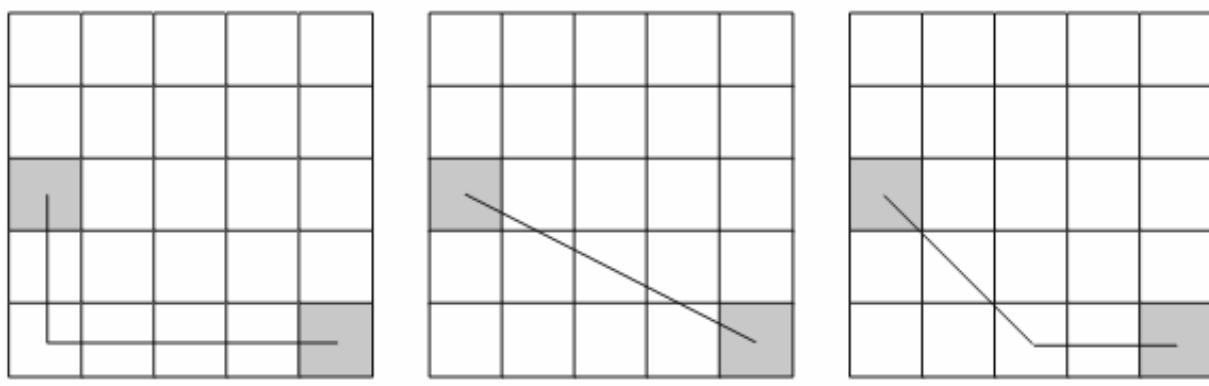
**Figure 4-13. Three paths, one cost**

它们三个都是最佳路径。

好的估价方法，不代表你能得到更短的路径。不过一些估价方法运行的更快。原因是别的估价方法需要计算更多的节点。计算的节点越少，到达终点节点就越快，更小的排序数组，等等（yujjj  
注：估价公式计算的估计值越接近实际值，需要计算的节点越少[公式计算复杂]；估价公式越简单，每个节点的计算速度越快；所以这里的说法不准确）。

同样，一些估价方法得到的路径在我们眼里看起来更直一些。上图中，虽然它们三个代价是一样，有人可能会说第三个路径是错的。其实是人们的错觉，如果一个评估方法导致对角路径和直线路径混合在一起，比把他们放在一起看起来更自然。

你可以就不同的估价方法写一本书。这里仅仅介绍三个常见的估价方法。如果你想了解更多的 Astar 估价方法，google 之。下图是几种评估方法的原理：



**Figure 4-14. Three common heuristics**

第一个叫做曼哈顿估价法(Manhattan heuristic)，它忽略所有的对角移动，只添加起点节点和终点节点之间的行、列数目。就像你在曼哈顿的大街上一样，比如说，你在 (5, 40)，到 (8, 43)，你必须先在一个方向上走过 3 个节点，然后另一个方向上的 3 个节点。有可能是先横走完，在竖走完，反之亦然；或者横、竖、横、竖、横、竖，每边都要走 3 个。

```
private function manhattan(node:Node):Number {
```

```

        return Math.abs(node.x - _endNode.x) * _straightCost +
            Math.abs(node.y + _endNode.y) * _straightCost;
    }

```

所以这种估价方法仅仅需要两节点之间的横竖列数之差，然后分别相加。

下一个常见的方法是几何估价法 (Euclidian heuristic) 它计算出两点之间的直线距离，本质公式为勾股定理  $A^2+B^2=C^2$ 。

```

private function euclidian(node:Node):Number
{
    var dx:Number = node.x - _endNode.x;
    var dy:Number = node.y - _endNode.y;
    return Math.sqrt(dx * dx + dy * dy) * _straightCost;
}

```

我们对横竖列数目的平方和开方。非常简单。

最后一个方法是对角线估价法(Diagonal heuristic),看起来很复杂,但本质的计算如图 4-13。

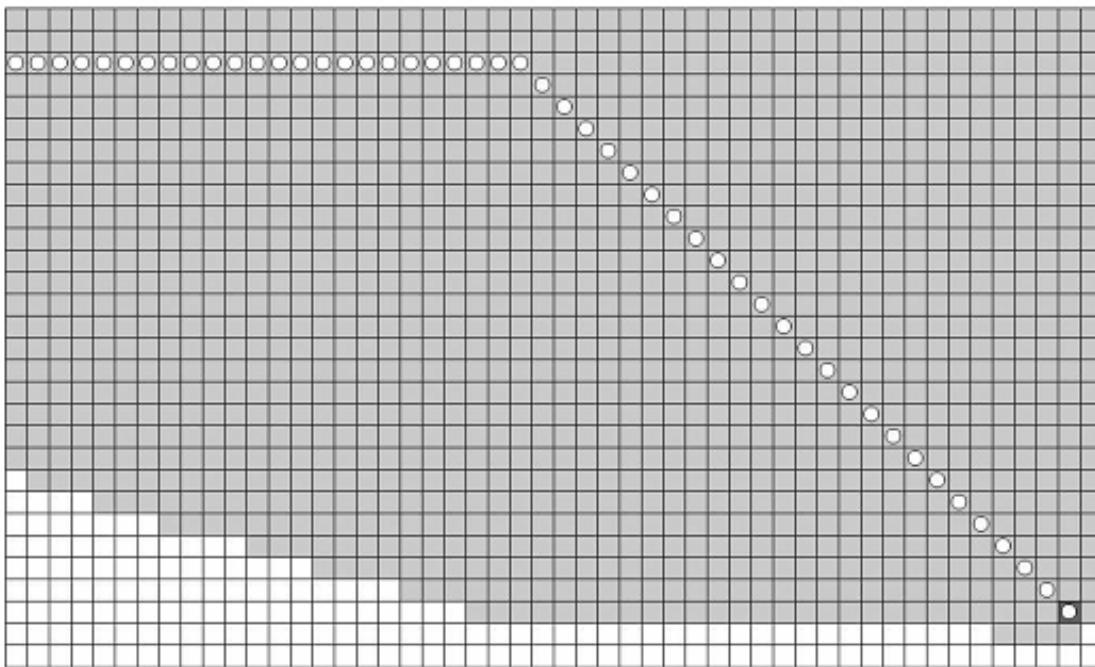
```

private function diagonal(node:Node):Number
{
    var dx:Number = Math.abs(node.x - _endNode.x);
    var dy:Number = Math.abs(node.y - _endNode.y);
    var diag:Number = Math.min(dx, dy);
    var straight:Number = dx + dy;
    return _diagCost * diag + _straightCost * (straight - 2 * diag);
}

```

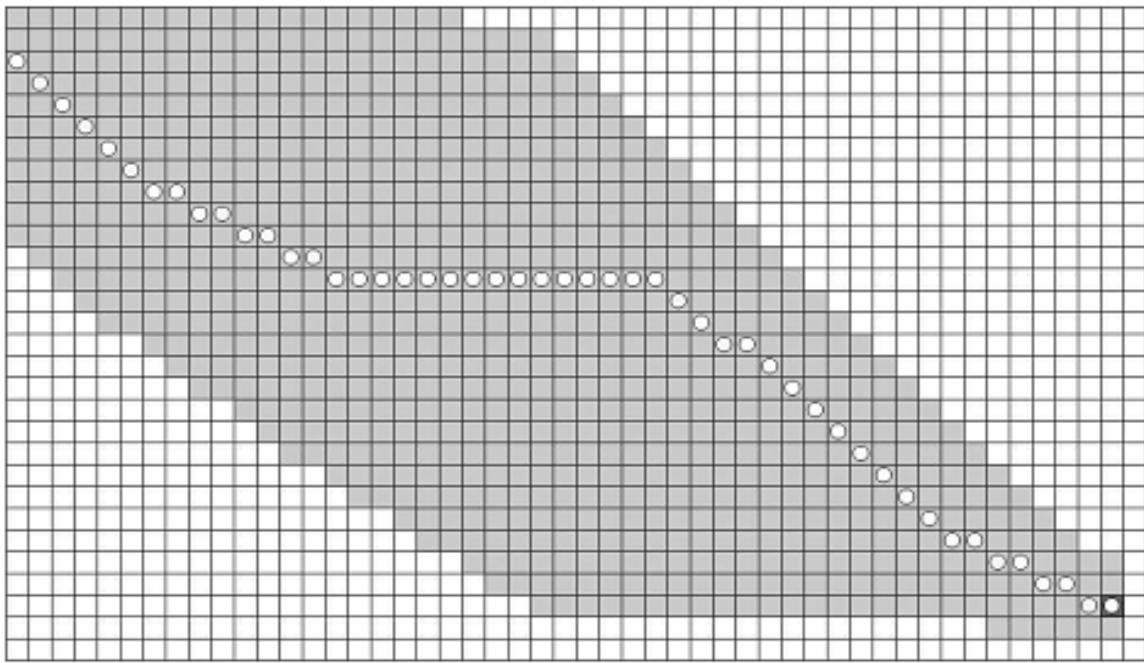
这个方法是三个估价方法里面最精确的,如果没有障碍,它将返回实际的消耗。事实上,如果你在上图中运行这个评估方法,返回的消耗是 4.8。

接下来的 3 张图显示了三种估价方法的结果。可以看出每种方法得出的结果都包含 23 个水平节点和 25 个对角节点,总消耗为 58. 不一样的估价方法对路径的长度没有影响。不过,路径的形状和考察过的节点(灰色的格子)有很大的差别。你可以看出曼哈顿估价法在到达结束点之前算过几乎所有的节点,路径也不是很自然,一下走完了所有的水平节点之后才开始走对角线。



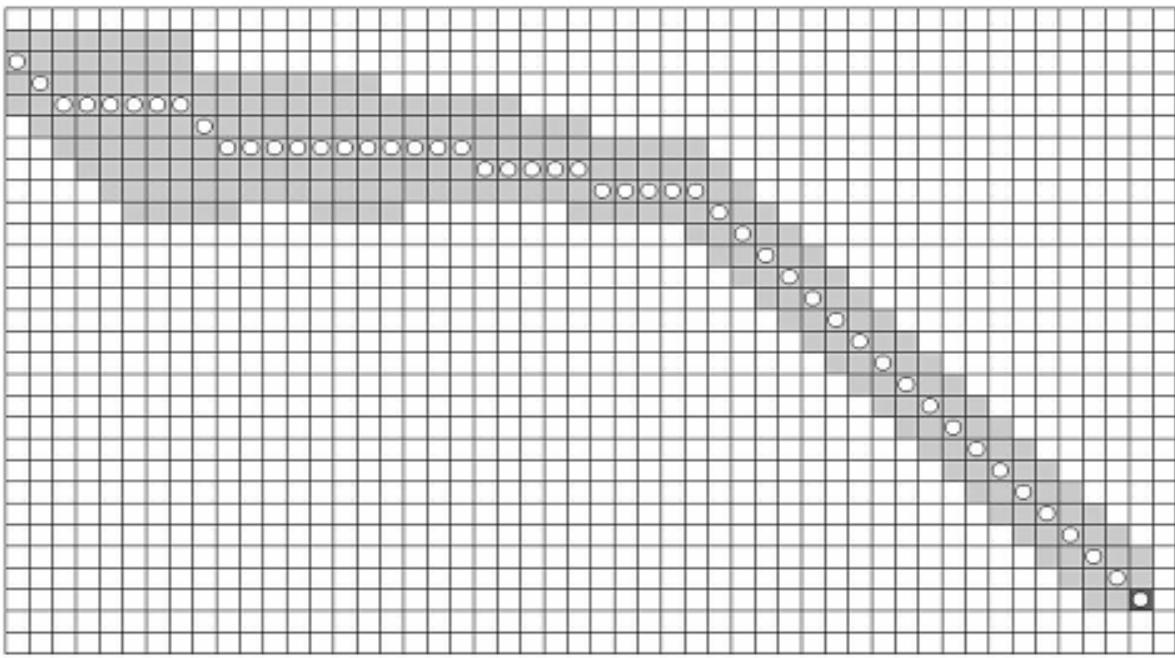
**Figure 4-15. Result of the Manhattan heuristic**

接下来的几何估价法看起来更自然,计算过的节点也更少。



**Figure 4-16. Result of the Euclidian heuristic**

对角线估价法是效率最高的，计算的非相关节点最少，虽然路径看起来没有几何评估自然，但比曼哈顿评估好。



**Figure 4-17. Result of the Diagonal heuristic**

当然，有得必有失。曼哈顿估价法最简单，计算各个节点是最快，但他计算的节点最多；对角线估价法比较复杂但常常计算的节点比较少。所有，没有错误或是正确的估价方法。每种方法都有优点。对于特定的程序，都需要不断的测试才能找到最好的方法。前面的例子实在没有任何障碍的基础上得到的，一旦你添加一些障碍，事情就更复杂了。

现在，我们学完了所有的基础知识，接下来看一看他的作用。

### 使用 Astar 类

当你在实际的游戏或程序中执行 Astar 时，你需要建立一个由方格组成的空间。起点节点将会是游戏角色待的地方，终点的方格不一定明确指出。可以是一个用户点击的点或是放着奖励的点，敌军或是其他的东西在他们之间。路径不是很直观，而是简单的移动角色到目标点。为了示范方便，我创建了 GridView 类让这些呈现在你眼前。

```

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

    /**
     * Serves as a visual representation of a grid of nodes used in a pathfinding solution.
     */
    public class GridView extends Sprite
    {
        private var _cellSize:int = 20;
        private var _grid:Grid;

        /**
         * Constructor.
         */
        public function GridView(grid:Grid)
        {
            _grid = grid;
            drawGrid();
            findPath();
            addEventListener(MouseEvent.CLICK, onGridClick);
        }

        /**
         * Draws the given grid, coloring each cell according to its state.
         */
        public function drawGrid():void
        {
            graphics.clear();
            for(var i:int = 0; i < _grid.numCols; i++)
            {
                for(var j:int = 0; j < _grid.numRows; j++)
                {
                    var node:Node = _grid.getNode(i, j);
                    graphics.lineStyle(0);
                    graphics.beginFill(getColor(node));
                    graphics.drawRect(i * _cellSize, j * _cellSize, _cellSize, _cellSize);
                }
            }
        }

        /**
         * Determines the color of a given node based on its state.
         */
        private function getColor(node:Node):uint
        {
            if(!node.walkable) return 0;
            if(node == _grid.startNode) return 0x666666;
            if(node == _grid.endNode) return 0x666666;
            return 0xffffff;
        }
    }
}

```

```

/**
 * Handles the click event on the GridView. Finds the clicked on cell and toggles its walkable
state.
 */
private function onGridClick(event:MouseEvent):void
{
    var xpos:int = Math.floor(event.localX / _cellSize);
    var ypos:int = Math.floor(event.localY / _cellSize);

    _grid.setWalkable(xpos, ypos, !_grid.getNode(xpos, ypos).walkable);
    drawGrid();
    findPath();
}

/**
 * Creates an instance of AStar and uses it to find a path.
 */
private function findPath():void
{
    var astar:AStar = new AStar();
    if(astar.findPath(_grid))
    {
        showVisited(astar);
        showPath(astar);
    }
}

/**
 * Highlights all nodes that have been visited.
 */
private function showVisited(astar:AStar):void
{
    var visited:Array = astar.visited;
    for(var i:int = 0; i < visited.length; i++)
    {
        graphics.beginFill(0xcccccc);
        graphics.drawRect(visited[i].x * _cellSize, visited[i].y * _cellSize, _cellSize, _cellSize);
        graphics.endFill();
    }
}

/**
 * Highlights the found path.
 */
private function showPath(astar:AStar):void
{
    var path:Array = astar.path;
    for(var i:int = 0; i < path.length; i++)
    {
        graphics.lineStyle(0);
        graphics.beginFill(0);
        graphics.drawCircle(path[i].x * _cellSize + _cellSize / 2,
                            path[i].y * _cellSize + _cellSize / 2,

```

```
        _cellSize / 3);  
    }  
}  
}  
}  
}
```

GridView 类的构造方法需要一个包含所有节点列表的 grid 实例，同时单独设置了起点节点和终点节点。drawGrid 方法遍历所有节点，为每一个节点画了一个小方格。方格的大小由 \_cellSize 属性决定，它的颜色由 getColor 方法得到，不能通过的为黑色，灰色为起点或终点节点，其他的为白色。

然后 `findPath` 方法执行，它创建了 `Astar` 实例，并调用了他的 `findPath` 方法，传入 `grid`。如果路径找到了，所有计算过的格子颜色变为灰色，路径上的节点还会标记上一个小圆圈，结果就像上面的三幅图一样。

当然，寻路时没有障碍就没什么意思。所以我们为鼠标点击添加一个事件监听器 `onGridClick`，点击格子就可以改变可通过属性开或关。每次点击后清除所有格子，重新寻路，然后又重新显示找到的路径和格子。

我们现在需要的是一个结合所有东西的主程序 findingpath.as。如下：

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

    [SWF(backgroundColor=0xffffffff)]
    public class Pathfinding extends Sprite {
        private var _grid:Grid;
        private var _gridView:GridView;

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

            _grid = new Grid(50, 30);
            _grid.setStartNode(0, 2);
            _grid.setEndNode(48, 27);

            _gridView = new GridView(_grid);
            _gridView.x = 20;
            _gridView.y = 20;
            addChild(_gridView);
        }
    }
}
```

这段代码的作用是创建坐标网和坐标网的视图，并把它们联系在一起。你可以设置不一样的起点、终点节点。一旦程序运行，你可以点击方格设置障碍来打断现有的路径。可以看出只要你留出了可以通过的路径，Astar 总是能找到通向终点的路径，而且永远是最佳的。如下图：

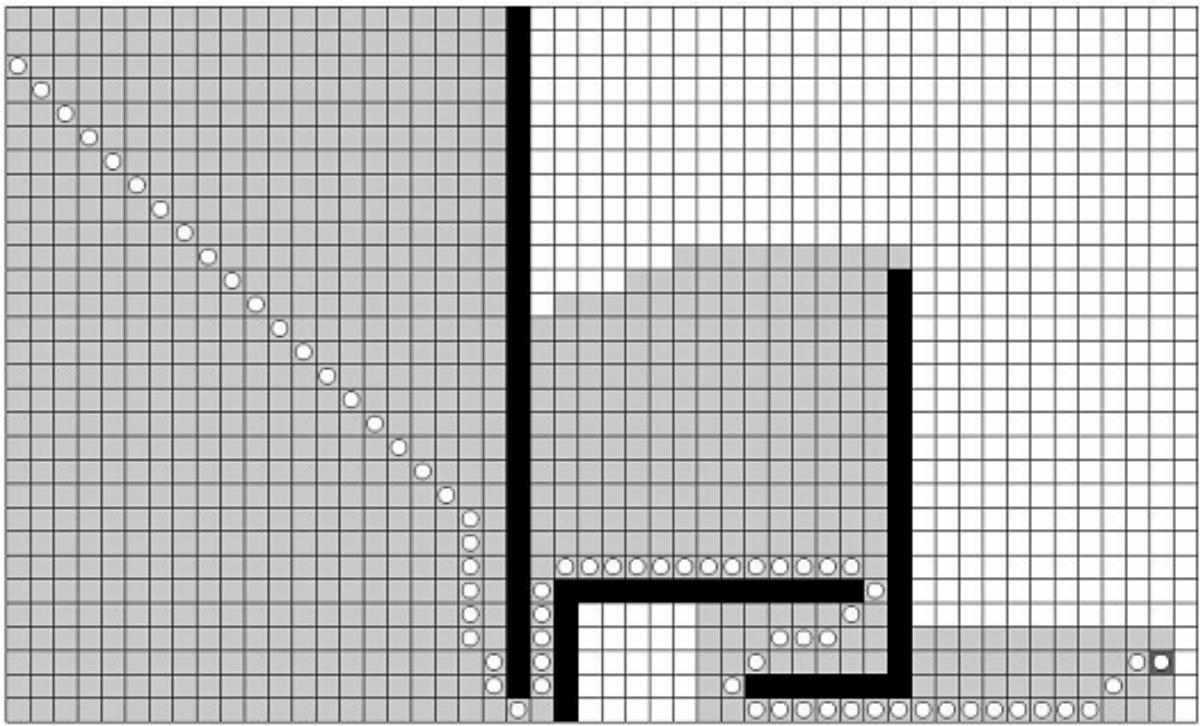


Figure 4-18. Pathfinding is complete

#### 修改路径细节：拐角

现在有一个潜在的问题，它不容易发现，在下图中，当路径到达障碍的边缘，它从对角走到了障碍的另一边。看起来不是什么大问题，但是当我们得到下图所示的路径的时候，问题出现了。

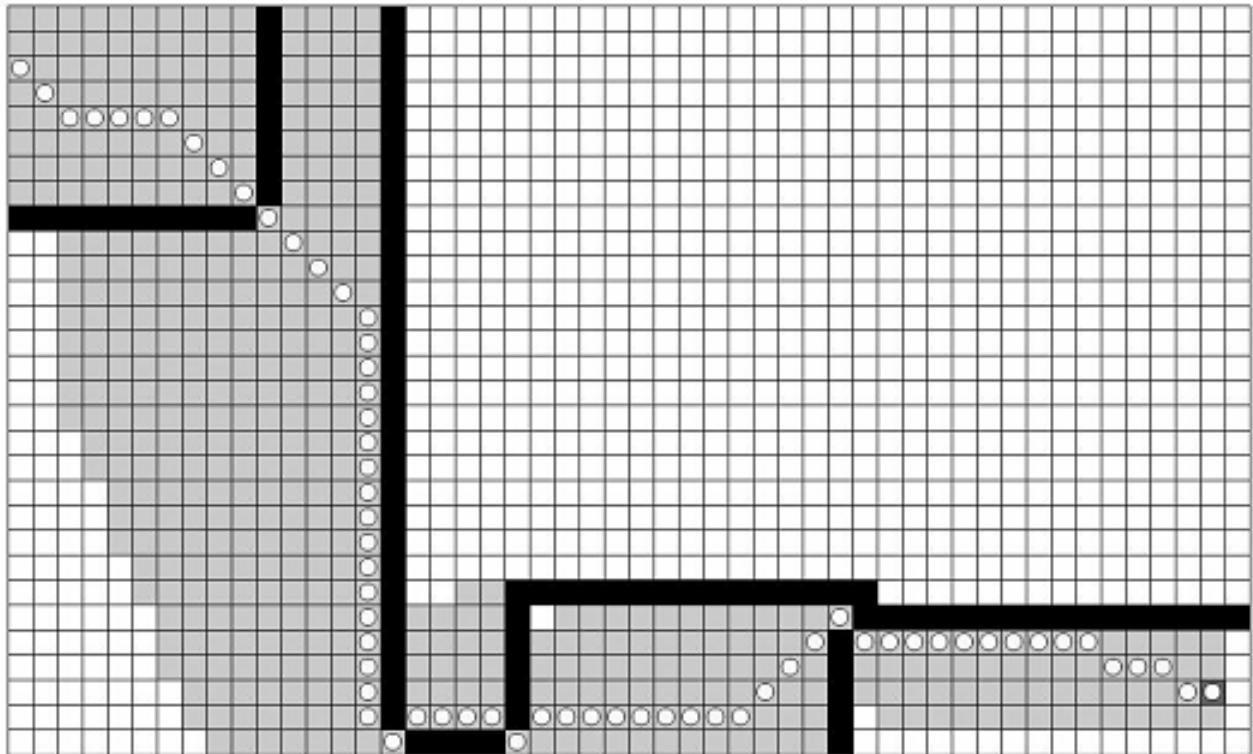
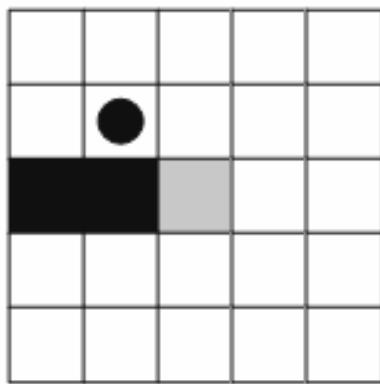


Figure 4-19. Slipping through cracks

上图中，路径穿越障碍。你可能会想你已经切断来了路径，但是在程序里走对角线是可行的。我们需要做的是在障碍旁边永远不抄近路（不走对角线）。绕过这些方格，防止像上图这样的情况，他们就不会找到合适的路径了。

看下图，图上有一个相似的情况。



**Figure 4-20. Close-up of a corner**

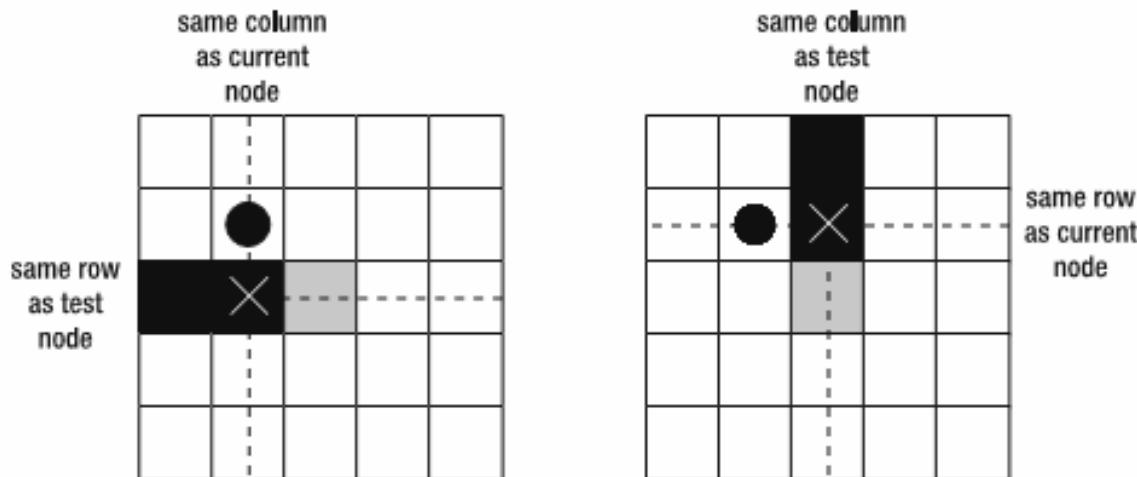
黑点表示的是当前节点，灰色格子为正在测试的节点，Astar 类里的 search 方法如下：

```
for(var i:int = startX; i <= endX; i++) {
    for(var j:int = startY; j <= endY; j++) {
        var test:Node = _grid.getNode(i, j);
        if(test == node || !test.walkable || ) {
            continue;
        }
    }
}
```

它确保了测试节点不是当前节点，而且测试节点是可以通过的。任何一个返回假，我们都会跳过这个节点到下一个邻近节点。我想添加一个条件：如果这个点的对角是不能通过的，跳过这个点。如果当前节点和测试节点是对角关系，其它两个节点就必须测试，如下：

```
node.x,test.y and test.x,node.y
```

换句话说和当前点同一列，测试点同一行的点（反之亦然）就像下图。

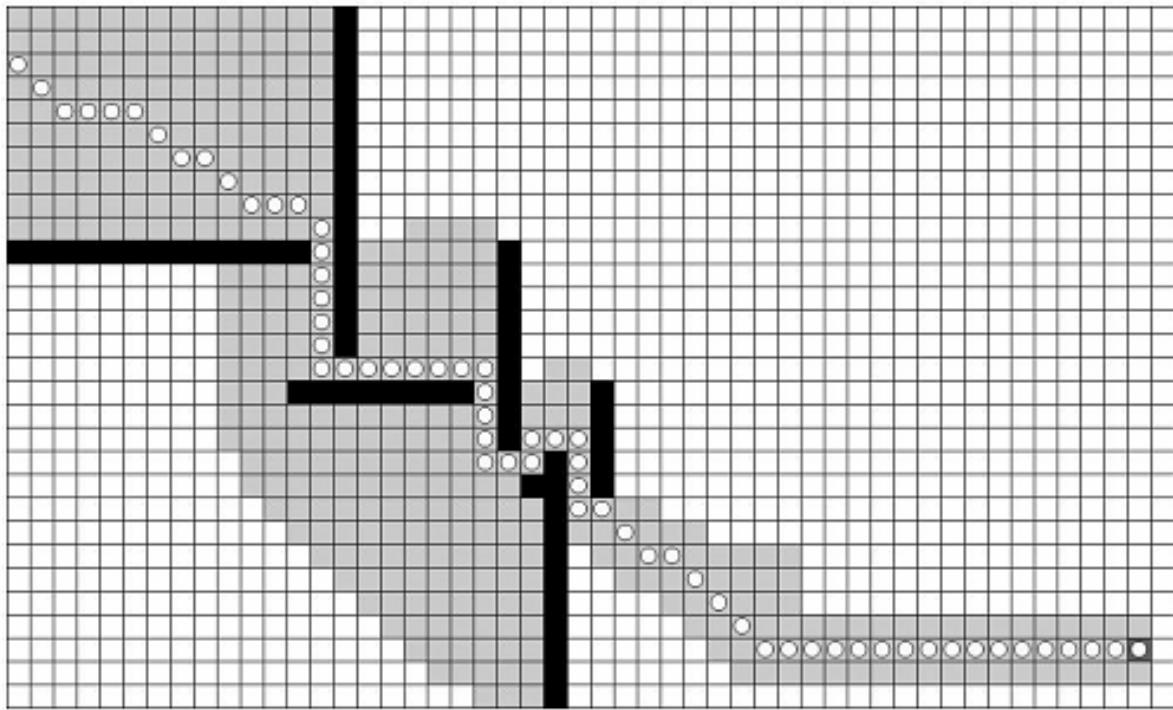


**Figure 4-21. Finding a corner**

所以我们必须检查他们是否可以通过。不行的话，跳过到下一个测试点，下面是需要修改的代码：

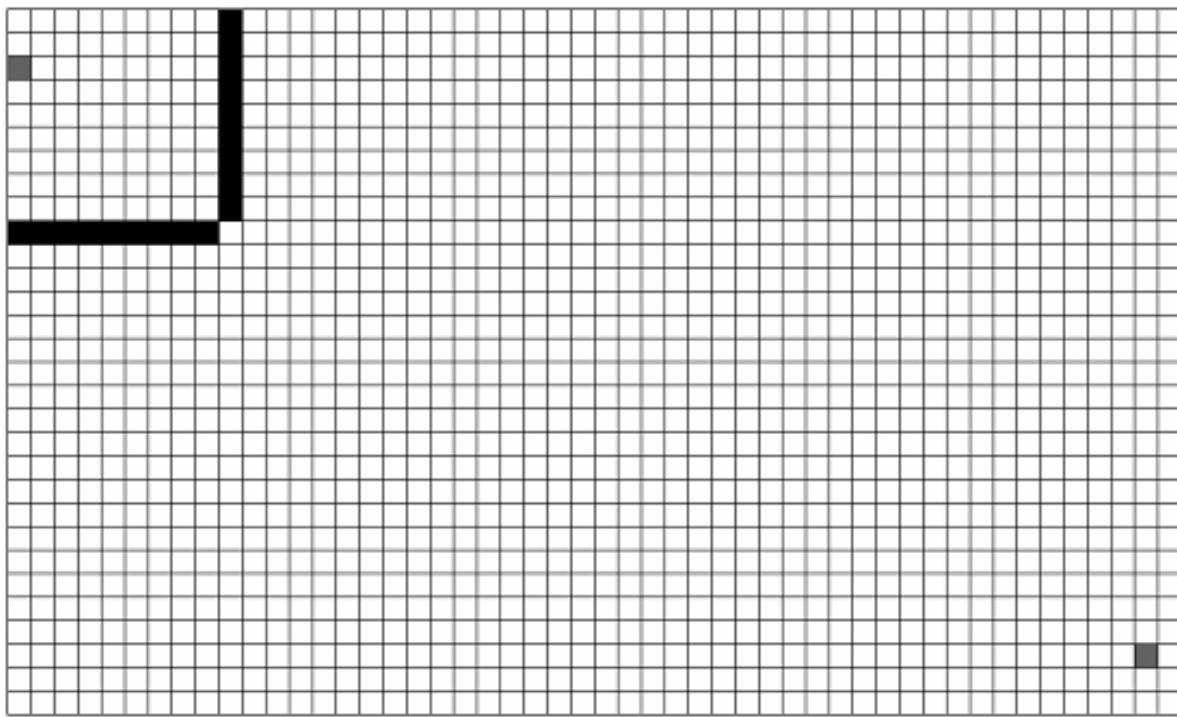
```
for(var i:int = startX; i <= endX; i++) {
    for(var j:int = startY; j <= endY; j++) {
        var test:Node = _grid.getNode(i, j);
        if(test == node || !test.walkable || !_grid.getNode(node.x, test.y).walkable ||
           !_grid.getNode(test.x, node.y).walkable) {
            continue;
        }
    }
}
```

添加简单的条件后，我们得到下面的路径：



**Figure 4-22.** Better corners

这时候路径将会绕过障碍而非穿过。此外，你如果完全的封闭了某一区域，像下图一样，就找不到路径了。



**Figure 4-23.** There is no path.

这个小小的改善你可能不需要。在一些游戏里他有用。完全取决于你的程序需要什么样的行为。你甚至会将它改为可设置的。同样，这个方法也是优化的对象。现在所有的节点无论是不是对角点都必须检查，有一点浪费。我的目地是把更清晰的代码给你，而你可能需要一个更完美的。

### 在游戏中使用 Astar

上一个例子主要是为了看到路径是这样生成的，那些节点被考察过，还可以测试不同节点被设为障碍时对路径的影响。

实际的情况恰恰相反——可通过与否的状态在游戏开始就设好了，而起点或终点节点却是动态

的。起点节点永远是角色当前的位置，终点节点经常是用户想让角色去的地方，一般是点击坐标网格。所以我们做一个简单的执行这种行为的游戏。文件名为 Game.as:

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.MouseEvent;
    public class Game extends Sprite
    {
        private var _cellSize:int = 20;
        private var _grid:Grid;
        private var _player:Sprite;
        private var _index:int;
        private var _path:Array;
        public function Game()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            makePlayer();
            makeGrid();
            stage.addEventListener(MouseEvent.CLICK, onGridClick);
        }
        /**
         * Creates the player sprite. Just a circle here.
         */
        private function makePlayer():void
        {
            _player = new Sprite();
            _player.graphics.beginFill(0xff0000);
            _player.graphics.drawCircle(0, 0, 5);
            _player.graphics.endFill();
            _player.x = Math.random() * 600;
            _player.y = Math.random() * 600;
            addChild(_player);
        }
        /**
         * Creates a grid with a bunch of random unwalkable nodes.
         */
        private function makeGrid():void
        {
            _grid = new Grid(30, 30);
            for(var i:int = 0; i < 200; i++)
            {
                _grid.setWalkable(Math.floor(Math.random() * 30),
                    Math.floor(Math.random() * 30),
                    false);
            }
            drawGrid();
        }
        /**
         * Draws the given grid, coloring each cell according to its state.
         */
    }
}
```

```

private function drawGrid():void
{
    graphics.clear();
    for(var i:int = 0; i < _grid.numCols; i++)
    {
        for(var j:int = 0; j < _grid.numRows; j++)
        {
            var node:Node = _grid.getNode(i, j);
            graphics.lineStyle(0);
            graphics.beginFill(getColor(node));
            graphics.drawRect(i * _cellSize, j * _cellSize, _cellSize, _cellSize);
        }
    }
}

/**
 * Determines the color of a given node based on its state.
 */
private function getColor(node:Node):uint
{
    if(!node.walkable) return 0;
    if(node == _grid.startNode) return 0xcccccc;
    if(node == _grid.endNode) return 0xcccccc;
    return 0xffffffff;
}

/**
 * Handles the click event on the GridView. Finds the clicked on
cell and toggles its walkable state.
*/
private function onGridClick(event:MouseEvent):void
{
    var xpos:int = Math.floor(mouseX / _cellSize);
    var ypos:int = Math.floor(mouseY / _cellSize);
    _grid.setEndNode(xpos, ypos);
    xpos = Math.floor(_player.x / _cellSize);
    ypos = Math.floor(_player.y / _cellSize);
    _grid.setStartNode(xpos, ypos);
    drawGrid();
    findPath();
}

/**
 * Creates an instance of AStar and uses it to find a path.
*/
private function findPath():void
{
    var astar:AStar = new AStar();
    if(astar.findPath(_grid))
    {
        _path = astar.path;
        _index = 0;
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
}
*/

```

```

        * Finds the next node on the path and eases to it.
    */
private function onEnterFrame(event:Event):void
{
    var targetX:Number = _path[_index].x * _cellSize + _cellSize / 2;
    var targetY:Number = _path[_index].y * _cellSize + _cellSize / 2;
    var dx:Number = targetX - _player.x;
    var dy:Number = targetY - _player.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    if(dist < 1)
    {
        _index++;
        if(_index >= _path.length)
        {
            removeEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
    }
    else
    {
        _player.x += dx * .5;
        _player.y += dy * .5;
    }
}
}
}

```

这个类有点像 GridView 例子，但是还有区别。在构造函数里，我们创建一个 player，仅仅是一个随机放在舞台上的圆圈，我们还创建了坐标网格和一些随机障碍。drawGrid 和 getColor 方法和之前一样。我们侦听点击事件，坐标网被点击后，设置起始节点为 player 当前的位置，点击的位置为结束节点。接着重绘坐标网表示出这些变化，试着用 findPath 方法寻路。

findPath 方法新建一个 Astar 的实例，试着找出路径。如果找到了，返回路径数组，从索引位置 0 开始；接着侦听 enterFrame 事件，调用 onEnterFrame。onEnterFrame 取出路径上的下一个节点，以索引值为准，得到从 player 到这个点的距离，演示一个简单的移动。当 player 和这个点足够近的时候，取出下一个节点。到达最后一个节点时，移除 enterFrame 侦听器。

## 进阶教程

这一章开始时的一个思想我们还没有解决：让一些节点通过时代价大一些，一些小一点，而不仅仅是简单的直线、对角关系。我不想讲得很细，但会简单的讲一点。

举个例子，土路节点肯定比高速公路的代价大，沼泽或是高山节点代价可能大很多。不一样的代价可以通过引用附加的属性来添加（权值）。方法是多样的。理想情况下直走一格消耗 1，对角 1.414。不过，差一点的节点权值可能是 2 倍，所以结果是 2 或者 2.828。

这样就可以使路径避开不易通行的路段，即便当容易路段更长一些的时候。但肯定有这样的时候：通过沼泽的花费比走过其他的路花费小。与障碍不同的是，不易通行的节点还是可以通过的，但是 Astar 将首先找出容易一点的路径。

为了演示这一点，我们添加一个新的属性到 Node 类，costMultiplier：

```

package {
    /**
     * Represents a specific node evaluated as part of a pathfinding algorithm.
    */
    public class Node
    {
        public var x:int;

```

```

public var y:int;
public var f:Number;
public var g:Number;
public var h:Number;
public var walkable:Boolean = true;
public var parent:Node;
public var costMultiplier:Number = 1.0;
public function Node(x:int, y:int)
{
    this.x = x;
    this.y = y;
}
}
}

```

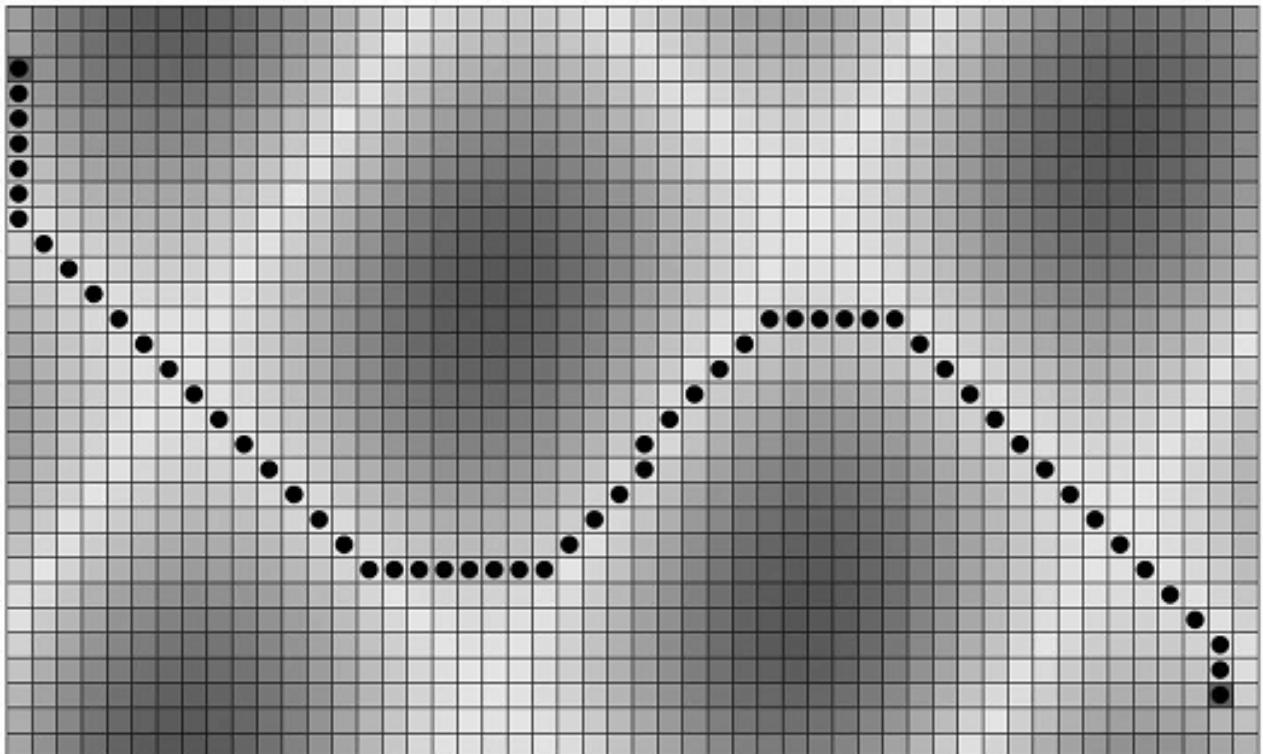
实现这一点还需要一些在 Astar 类里的 search 方法中修改一点，在计算 g 值的那一行上：

```
var g:Number = node.g + cost;
```

修改为：

```
var g:Number = node.g + cost * test.costMultiplier;
```

现在每个节点的代价已经被权值调整过了，如图。物品修改了 GridView 类来显示每个节点不一样的代价。颜色的深浅代表了它代价的大小。浅色的比深色的代价小。你可以在 GridView2.as 里看到这些改变。注意到路径只是遵循简单的原则。



**Figure 4-24. The path of least resistance**

不过，如果你调整障碍的位置，Astar 还是会走过艰难的路段，如下图：

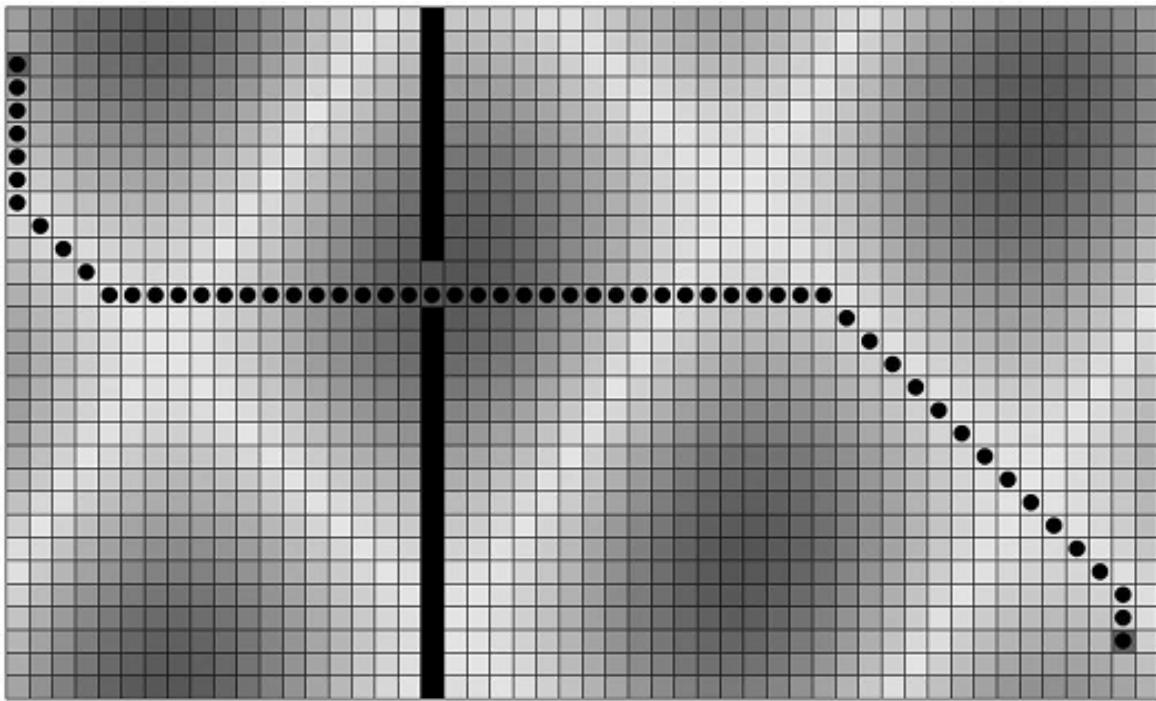


Figure 4-25. Forcing the path into difficult terrain

## 总结

最后的这个例子很简单但留给你了不少可扩展的空间，还向你展示了基本的使用 AStar 寻路方法的方法。试着玩一下，观察 player 怎样绕过障碍到达你点击的地方。你可以想象做一个更动感的 player（比如一个人的走动）等等。同样，一些更漂亮的图片如草、树、路（可以通过的节点）和河流、山川（不能通过的节点）。

你还可以使用寻路在等轴测空间里，如果你已经理解上一章的话，我把这些留给你。下一步，我们将会学习一些完全不一样的东西，使用话筒和摄像头，当然不仅仅是获取声音和图像，而是一些让用户使用动作和声音控制程序的方法。

## 第五章 二级输入设备：摄像头和麦克风

如果你上过计算机课程，我敢打赌类似下面张图肯定或多或少都看到过。



图 5-1 (输入 > 处理 > 输出)

要是上过更高级的课程，可能图中还会有个标有“存储”的格子。

这实际上就是当今计算机的实现原理。从输入设备读取一些数据，或存储它，或处理它，最后输出。计算机和软件最关注的一部分是当中的过程——处理数据以创建有意义的输出内容。输入设备一般是指键盘或者鼠标，输出设备大多由屏幕和喇叭组成。当然，这种搭配和变化是无止境的，不过无论程序员还是终端用户，牵涉最多的也就是鼠标、键盘、屏幕和声音设备。说句老实话，从输入的技术角度看键盘和鼠标，其实它们要做很多的事情。早期，相当长一段时期的“个人电脑”所谓的输入设备就是一系列开关，输出设备就是一些能开能关的发光二极管。我知道这些原理、机制和实际情况严重脱节，因为实际情况，输出大多是可视、可听的。

看看科幻片里面的那些设想都是极端美好的（当然还是希望有朝一日能够得以实现）。而眼下发展分为两大块，触摸屏，其趋势是仿键盘和仿物理输入，还有声音识别技术，这块虽然一直在改善，但还远远不够。在此尽管我不会建议你现在就去用麦克风来搞写作或是编程，但是探索一下二级输入设备，特别是麦克风和摄像头，仍然是一件很有趣的事情。

### 摄像头和麦克风

使用二级设备的障碍之一就是无可用之物。如今你很难找到一台不带键盘和鼠标的电脑，但是作为一些非标配的设备，首先要有设备，其次还可能要为之找驱动程序，然后才能使其用于程序中。

幸运的是，近年来越来越多的笔记本电脑开始内置摄像头和麦克风。麦克风几乎是即插即用的，摄像头的安装程序也花不了几分钟。作为一个 Flash 开发人员，是可以直接访问它们的。

大多数人总认为摄像头和麦克风就只适用于聊天室之类的应用（换句话说，把声音和摄像的数据流来回传递）但有一小部分人动了脑子，利用这些设备获取声音和视频，然后分析数据，作出了一些很酷的玩意儿。

这一章就告诉你怎么做这些酷的玩意儿。当然，我也不可能面面俱到，但会讨论如何获取视频和声音数据，以及一些比较好玩的处理方式。最主要是激发各位的想象力。我们从一个简单的处理麦克风的例子开始。

### 输入的声音

Flash 中有 Microphone 类。这个类很好用，但限制也很多。很多熟悉 Sound 类的人，一学习 Microphone 类就急着想作出即时音谱的效果。如果你正在这么想，那我就要泼你冷水了。不要认为你可以记录来自麦克风的声音。

Microphone 类最大的作用是返回麦克风当前的活跃级数。级数从 0 到 100，表示接收音量的多少，0 指没声音，100 指接受的最大限度。就这加入一点想法，也能作出很酷的东东。首先让我们通过 AS 访问一下麦克风。

Microphone 类含有一个静态的函数 getMicrophone，它返回一个 Microphone 对象，代表着接在电脑上的麦克风。这个函数还有参数，如果不传，返回的就是程序在系统上找到的第一个麦克风，如果传递 -1，就返回系统默认的麦克风。通常这两种情况都一回事儿。但你也可以给机器上插一大堆麦克风，然后通过传递索引值来取得对应的麦克风，不过这貌似很 2。试试下面的 MicrophoneTest。

```

package {
    import flash.display.Sprite;
    import flash.media.Microphone;
    public class MicrophoneTest extends Sprite
    {
        private var _mic:Microphone;
        public function MicrophoneTest()
        {
            _mic = Microphone.getMicrophone();
            trace(_mic.name);
        }
    }
}

```

试着传入不同的参数到 getMicrophone 看看输出结果。最终发现，如果索引所指的麦克风不存在，会返回 null。报错则是说，不能访问一个不存在的属性。

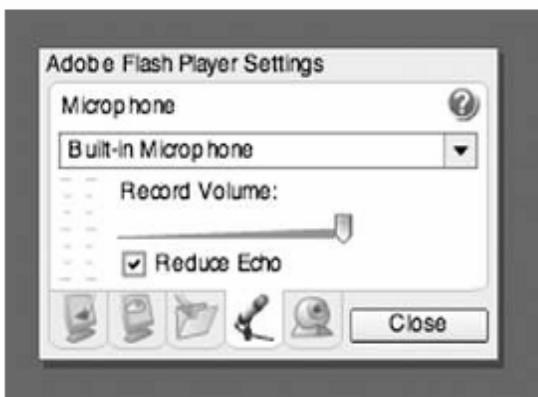
如果想直接让用户选择麦克风，可以调出麦克风设置面板，方法如下：

```

public function MicrophoneTest()
{
    _mic = Microphone.getMicrophone();
    Security.showSettings(SecurityPanel.MICROPHONE);
}

```

这样就会弹出一个选择麦克风的对话框。如图 5-2



**Figure 5-2.** Microphone dialog box



**Figure 5-3.** Camera and Microphone Access dialog box

在选择好麦克风后点击关闭按钮，会出现摄像头和麦克风的访问许可对话框。如图 5-3

许可的问题就不多说了，你就想想一个 Flash 影片偷偷摸摸的打开你的摄像头和麦克风有多恐怖就行了。可能你正在想，为什么第一个例子中，仅获取麦克风不需要许可呢？因为虽然获得了麦克风的引用，但并没有实际使用它，所以不存在被偷听的风险。Flash 会在麦克风数据被访问的第一时间显示这个对话框。现在开始访问麦克风并读取活跃级数，这个级数会不断改变，所以要一直检测。

```

public function MicrophoneTest()  {
    _mic = Microphone.getMicrophone();
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
private function onEnterFrame(event:Event):void  {
    trace(_mic.activityLevel);
}

```

代码超简单，没啥好说的。不过情况比较怪，不管你对着麦克风怎么吼，还是玩了半天的选择麦克风窗口，结果都是一列 -1 在不断的飘。所以，再说一遍，这是因为 Flash 还没有对麦克风的

数据动过手脚，访问允许的对话框不是也没跳出来嘛。

要处理麦克风数据，有两种方式。一种是把麦克风附到一个 NetStream 上，这通常用于聊天程序或者录音系统。另外一种方式是通过 setLoopBack 函数，把麦克风声音重新发送到本地扬声器。传递 true 给 setLoopBack 即开始访问麦克风的输入数据，这时对着麦克风练练嗓子，会发现活跃级数的变化。传递 false 则停止访问。如果你不想听到自己的声音，就把喇叭关了吧。

```
public function MicrophoneTest() {
    _mic = Microphone.getMicrophone();
    _mic.setLoopBack(true);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
private function onEnterFrame(event:Event):void {
    trace(_mic.activityLevel);
}
```

运行这段代码，对着麦克风来两下，看看输出数值的变化。如果吼半天数值还很低，你可能需要调整一下增益值。增益是指扩大级数。设置麦克风的增益有两种方法，手动设置或代码设置。在图 5-2 的麦克风设置面板中，有一个滑动条是用来调整增益的。如用代码，麦克风对象有一个 gain 的属性，接收值在 0 到 100。乘还没觉得枯燥，我们赶快做点有趣的事情，用位图画出活跃级数：

(代码 MicrophoneTest)

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.media.Microphone;

    public class MicrophoneTest extends Sprite {
        private var _mic:Microphone;
        private var _bmpd:BitmapData;
        public function MicrophoneTest() {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _bmpd = new BitmapData(400, 50, false, 0xffffffff);
            addChild(new Bitmap(_bmpd));
            _mic = Microphone.getMicrophone();
            _mic.setLoopBack(true);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void {
            _bmpd.setPixel(298, 50 - _mic.activityLevel / 2, 0);
            _bmpd.scroll(-1, 0);
        }
    }
}
```

结果如图 5-4



Figure 5-4. Visually graphing the activity level

这点小打小闹就到此为止吧，下面看看如何做一个简单的游戏，用声音来控制角色的移动。  
声控游戏

下面的例子是一个叫 SoundFlier 的类。它创建了一个像飞机一样的东西，穿梭在粗糙的山脉中，飞机的飞行是由麦克风的活跃级数来控制的。（代码 SoundFlier ）

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.Rectangle;
    import flash.media.Microphone;
    [SWF(backgroundColor=0xffffffff)]
    public class SoundFlier extends Sprite
    {
        private var _mic:Microphone;
        private var _flier:Sprite;
        private var _background:Bitmap;
        private var _vy:Number = 0;

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

            makeBackground();
            makeFlier();

            _mic = Microphone.getMicrophone();
            _mic.setLoopBack(true);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function makeBackground():void
        {
            var bmpd:BitmapData = new BitmapData(stage.stageWidth, stage.stageHeight, false);
            _background = new Bitmap(bmpd);
            addChild(_background);
        }

        private function makeFlier():void
        {
            _flier = new Sprite();
            _flier.graphics.lineStyle(0);
            _flier.graphics.moveTo(-10, 0);
```

```

        _flier.graphics.lineTo(-10, -8);
        _flier.graphics.lineTo(-7, -4);
        _flier.graphics.lineTo(10, 0);
        _flier.graphics.lineTo(-10, 0);
        _flier.x = 100;
        _flier.y = stage.stageHeight - 50;
        addChild(_flier);
    }

private function onEnterFrame(event:Event):void
{
    _vy += .4;
    _vy -= _mic.activityLevel * .02;
    _flier.y += _vy;
    _vy *= .9;
    _flier.y = Math.min(_flier.y, stage.stageHeight - 20);
    _flier.y = Math.max(_flier.y, 20);

    var h:Number = Math.random() * 120;
    _background.bitmapData.fillRect(new Rectangle(stage.stageWidth - 20, 0, 5, h),
0xffffffff);

    h = Math.random() * 120;
    _background.bitmapData.fillRect(new Rectangle(stage.stageWidth - 20,
stage.stageHeight - h, 5, h), 0xffffffff);

    _background.bitmapData.scroll(-5, 0);
}
}
}

```

飞机是一个 Sprite，背景是一个右端在不断画着随机矩形的滑动位图。关键部分在 onEnterFrame 函数中：

```

_yVelocity += .4;
_yVelocity -= _mic.activityLevel * .02;
_flier.y += _yVelocity;
_yVelocity *= .9;
_flier.y = Math.min(_flier.y, stage.stageHeight - 20);
_flier.y = Math.max(_flier.y, 20);

```

飞机的垂直速度用 `_yVelocity` 来表示，每次增加 0.4 是模拟重力作用，减去活跃级别的百分之二是模拟上升飞行。这俩系数搞了我大半天。然后把垂直速度加于飞机的 y 轴，为了模拟摩擦，速度又乘以了 0.9。最后限制一下飞机的位置，使之不会飞出屏幕。

结果如图 5-5



**Figure 5-5. Sound flier!**

当你对着麦克风说话或是弄出点声音时，飞机就上升，关闭麦克风，飞机就下落。不断向左滑动的山脉，使飞机看上去像是在向右飞行。确保声音大小使飞机在中间飞行。这里没做碰撞检测，你可以自己加上去。像这种游戏可没法乘老板不注意时偷偷的玩儿。

虽然是个挺无聊的游戏，但是否激发了你创意的火花，想着如何利用麦克风在游戏或应用中做一些控制呢。下面我们会谈谈另外一种处理麦克风的方式。

### 活跃事件

在 SoundFlier 游戏中，每一帧都对活跃级数进行了检测。像这类游戏是需要比较敏感的反应，而你可能在想，如果只当音量大到一定程度时才作出反应，是不是要不停的判断活跃级数呢？这样做显然很没有效率。

幸运的是，Microphone 类提供了一个解决方案：ActivityEvent。这个事件就是在声音跃过某个程度时才被发布。但这样会出现两种情况，声音从低到高涨过某个值，和从高到低跌过某个值。为了区别，事件带有一个叫 activating 的参数。如果是 true，就说明是涨过，反之 false 就是跌过。

通过监听该事件，程序就可以在音量没有达到指定大小时悠闲的等待，而超过该大小时再作出反应。

这类“声控开关”可以运用于游戏、网页应用以及 AIR 的桌面应用中。你甚至可以做一个粗糙的安全系统，比如给家里装个声控装置，它能激活摄像头并通知你，让你看到家里正发生了什么。或者利用另一种情况，把麦克风绑机器边上，当机器停止运行，程序发现声音降低到某个程度后唤起机器继续运作。

来快速过一个例子，它通过拍手来开关房间的灯泡。了解了它是如何工作后，你就能利用这种机制控制任何东西了。代码 Clapper 。

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.ActivityEvent;
    import flash.media.Microphone;

    public class Clapper extends Sprite
    {
        private var _mic:Microphone;
        private var _on:Boolean = false;
```

```

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

    update();

    _mic = Microphone.getMicrophone();
    _mic.setSilenceLevel(25, 500);
    _mic.setLoopBack();
    _mic.addEventListener(ActivityEvent.ACTIVITY, onActivity);
}

private function onActivity(event:ActivityEvent):void
{
    if(event.activating)
    {
        _on = !_on;
        update();
    }
}

private function update():void
{
    graphics.clear();
    if(_on)
    {
        graphics.beginFill(0xffffffff);
    }
    else
    {
        graphics.beginFill(0);
    }
    graphics.drawRect(0, 0, stage.stageWidth, stage.stageHeight);
}
}
}

```

都是很基础的东西。注意调用了一个叫 `setSilenceLevel` 的函数，它用来设置之前提到的声音跃过的某个值。默认是 10，这里设置的是 25。`setSilenceLevel` 的第二个参数是时间跨度，单位毫秒。当一个 `ActivityEvent` 被发布，麦克风会忽略这个时间跨度内其它的 `ActivityEvent`。比如连续拍三下手，每次拍手活跃级数都会上下波动一次，而设置了时间跨度，跨度内的拍手就会被忽略。默认是 2000 也就是 2 秒，对于聊天程序这可能是个不错的选择。在此，我发现有点偏长，就设置成了 500。`onActivity` 是处理函数，这里，它只关心有声音发出，而不关心声音消失。当有声音发出，切换变量 `_on`，并调用 `update` 函数。`update` 函数根据 `_on` 来绘制场景黑白。运行下试试吧。我建议对 `setSilenceLevel` 尝试设置不同的参数看看效果。如果想知道当前的设置情况，`Microphone` 类还有两个只读属性：`silenceLevel` 和 `silenceTimeout`。

OK，麦克风的运用就介绍到此。虽然这里没什么复杂的功能，但完成这些也没用到键盘和鼠标。下面让我们看看视频的运用吧。

## 输入的视频

声音输入由 `Microphone` 类处理，视频输入由 `Camera` 类处理。虽然处理的内容不同，但是这两个类非常相似。同样通过调用 `Camera.getCamera` 获得摄像头的引用，同样也有 `activityLevel` 属性。

性和类似 setSilenceLevel 的 setMotionLevel 函数，同样也发布 ActivityEvents 事件。

但不管多相似，摄像头能做更多的事情。除了读取活跃级数和响应活跃事件以外，摄像头经过一个 Video 对象就能让你看到视频，而这个对象是一个显示对象，所以显示对象能做的事情，它都能做，比如滤镜，变形，混合模式等等。当然最强大的还是使用 BitmapData 画出视频内容。这样，通过图像分析、比较等等，对于图像处理来说就有着无限可能。这章会通过几个例子激发出你的创意，我相信即使不为此花费整本书，等过了一周后，你也能作出我怎么也想不到的东西。

千里之行始于足下，先来看看如何引用摄像头并看到拍摄的视频，这第一步和麦克风同出一辙：

```
package {
    import flash.display.Sprite;
    import flash.media.Camera;
    import flash.system.Security;
    import flash.system.SecurityPanel;
    public class CameraTest extends Sprite
    {
        private var _cam:Camera;
        public function CameraTest()
        {
            _cam = Camera.getCamera();
            trace(_cam.name);
            Security.showSettings(SecurityPanel.CAMERA);
        }
    }
}
```

调用 Camera.getCamera 获得摄像头。可以传递一个字符串作为名字取得对应的摄像头，不过这一般都没人注意，所以最好啥也不传，获得默认摄像头。虽然如此，调用一下摄像头选择面板也不错，如图 5-6，即便最后一行代码不会再出现，但你也应该加在应用程序中，好让用户知道如何去选择摄像头。图 5-6

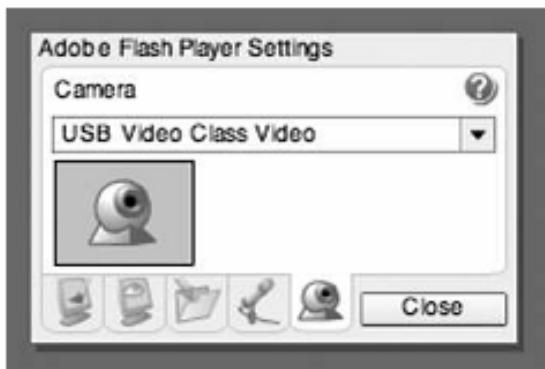


Figure 5-6. Camera dialog box

有了摄像头，就该输出视频了。把摄像头附在 Video 对象上即可，记得之前说 Video 是一个显示对象，所以记得要把它加入显示列表：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.media.Camera;
    import flash.media.Video;
    public class CameraTest extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
        public function CameraTest()
```

```

    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;
        _cam = Camera.getCamera();
        _vid = new Video();
        _vid.attachCamera(_cam);
        addChild(_vid);
    }
}
}
}

```

如果一切正常，你现在就能看到你那美丽的房间了。在此之前，程序会调用图 5-3 里出现的许可面板。

### 视频尺寸和质量

场景上出现了摄像头拍摄的内容。但画面不够大，也不像其它看到的视频那样清晰。这不是因为 Flash 不行，而是由于默认的设置不好。视频可以在创建时设置其大小，默认是 320x240，不过可以更大点：

```
_vid = new Video(640, 480);
```

另一种设置方式，是通过 width 和 height 属性单独设置。

视频是大了，但画面更差了。为了改善画面，需要使用 Camera 类的 setMode 函数。这个函数共有四个参数，这里只介绍前三个：宽(width)，高(height)，帧频(fps)（第四个参数去看帮助），它们的默认值分别是 160，120 和 15。靠，这下找到了视频烂的原因了。赶快设置参数为 320x240 或者干脆 640x480，看看变化。而 15fps 似乎还不赖。

```

_cam = Camera.getCamera();
_cam.setMode(640, 480, 15);
_vid = new Video(640, 480);
_vid.attachCamera(_cam);
addChild(_vid);

```

要知道，越高的 fps，机器在处理视频时越费劲。对于类似视频会议这种东西，视频的质量和尺寸并不是非常重要，所以不要为此花太大精力。如果你一定要弄，可以去试试 setQuality（设置质量）函数，还有带宽和视频流的压缩。但这些都不属于本章要讨论的东西，所以现在开始忘掉它们。如果这是你对 Flash 摄像头的初体验，那么花点时间熟悉熟悉。比如加点滤镜、混合模式，让视频旋转、变形、缩放、动来动去什么的。觉得差不多了，我们就开始解剖视频流。

### 视频和位图

正如之前说的，摄像头最强大的应用是混合 BitmapData。所以，通过 draw 把视频绘制在一个 BitmapData 对象里，就能以像素级来控制整个东西了。当然，对于动画，就需要不停的绘制。

```

package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.geom.Matrix;
    import flash.media.Camera;
    import flash.media.Video;
    public class CameraBitmap extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
        private var _bmpd:BitmapData;
        public function CameraBitmap()

```

```

{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    _cam = Camera.getCamera();
    _cam.setMode(320, 240, 15);
    _vid = new Video(320, 240);
    _vid.attachCamera(_cam);
    _bmpd = new BitmapData(320, 240, false);
    addChild(new Bitmap(_bmpd));
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
private function onEnterFrame(event:Event):void
{
    _bmpd.draw(_vid, new Matrix(-1, 0, 0, 1, _bmpd.width, 0));
}
}
}

```

这里我们创建了一个 `BitmapData`, 并包在一个 `Bitmap` 里, 然后把它——而不是 `Video`——加入到显示列表。最后每帧绘制视频到 `BitmapData`。为了让 CPU 省力点, 我把尺寸又改回了 320x240。

### 反转图像

要知道, 用户在摄像头前所做的反应, 总希望和镜子中一样, 人往左, 视频中人也往左, 人往右, 其也往右。但事实并不是这样, 所以需要我们来为之做反转图像。

`onEnterFrame` 中的一行代码, 拆成可读性强点的两行代码如下:

```

var matrix: Matrix = new Matrix(-1, 0, 0, 1, _bmpd.width, 0);
_bmpd.draw(_vid, matrix);

```

首先先确定导入了 `flash.geom.Matrix` 类。对象的绘制以传入的变形矩阵为依据。通过矩阵能做很多事情, 虽然只有很简单的前四个参数, 它们却控制着缩放, 旋转和形变。这里让图形的 `x` 轴缩放率等于 `-1`, 意味着水平反转它。接着, `0` 的意思是指图形没有旋转和形变, 最后一个 `1` 是说 `y` 轴保持 100% 不变。但仅仅如此, 还看不到任何东西。因为此时的矩阵是从右向左扩展的, 而最后两个参数是把矩阵移到图形的右上角, 这样就保证落入了可视范围。

用前面的例子测试一下, 看看是不是更自然了呢。

### 分析像素

现在你拥有了像素的控制权, 该如何处置? 记得我之前说过不要高质量、高分辨率的视频嘛? 我现在要更进一步的说, 越低分辨率、越低质量越好。就算只有 320x240 大小的视频图像, 每一帧要处理的像素也有 76800 个啊。所以不要搞太大的视频来玩。事实上, 你将会发现我要做的第一件事情, 是设计如何除掉这么多像素带来的巨大信息。一般来说, 我们只对视频的某个颜色区域, 或者对比度之类的感兴趣。

### 分析颜色

首先我们尝试着跟踪一个具体颜色。假如用户拿一个颜色 (亮红色) 比较突出的东西在摄像头前面晃动, 我们能跟踪其位置就算成功。

跟踪颜色的任务交给内置的 `BitmapData.getColorBoundsRect` 函数。这个函数通过给定的颜色, 返回所有像素中含有这个颜色值的像素的最大范围:

```
bitmapData.getColorBoundsRect(mask:uint, color:uint, findColor:Boolean);
```

参数 `mask` (掩码) 的意思是, 过滤颜色通道。比如, 传递 `0xFF0000` 作为掩码, 那么该函数只判断像素颜色值的红色部分, 这是因为红色通道的掩码是

`FF`, 而绿色和蓝色都是 `0`, 被过滤掉了。参数 `findColor`, 如果 `true` 是指返回包含指定颜色值的矩形范围, `false` 指返回不包含指定颜色值的矩形范围。

由于我们还不知道要跟踪什么样的颜色值, 所以让用户点击图片来选择一个颜色值。然后画出含有这个颜色值的矩形:

```

package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.geom.Matrix;
    import flash.geom.Rectangle;
    import flash.media.Camera;
    import flash.media.Video;
    public class ColorTracking extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
        private var _bmpd:BitmapData;
        private var _cbRect:Sprite;
        private var _color:uint = 0xffffffff;
        public function ColorTracking()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _cam = Camera.getCamera();
            _cam.setMode(320, 240, 15);
            _vid = new Video(320, 240);
            _vid.attachNetStream(_cam);
            _bmpd = new BitmapData(320, 240, false);
            addChild(new Bitmap(_bmpd));
            _cbRect = new Sprite();
            addChild(_cbRect);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            stage.addEventListener(MouseEvent.CLICK, onClick);
        }
        private function onClick(event:MouseEvent):void
        {
            _color = _bmpd.getPixel(mouseX, mouseY);
        }
        private function onEnterFrame(event:Event):void
        {
            _bmpd.draw(_vid, new Matrix(-1, 0, 0, 1, _bmpd.width, 0));
            var rect:Rectangle = _bmpd.getColorBoundsRect(0xffffffff, _color, true);
            _cbRect.graphics.clear();
            _cbRect.graphics.lineStyle(1, 0xff0000);
            _cbRect.graphics.drawRect(rect.x, rect.y, rect.width, rect.height);
        }
    }
}

```

这段程序（仍然）很平常。创建了一个叫`_cbRect` 的用来画出矩形范围。当点击位图时，获得像素颜色值，然后每帧调用`getColorBoundsRect` 取得范围。要注意的是掩码`0xFFFFFFFF`，意味着我们关心所有通道上的颜色值。够简单吧。

现在，试着拿个亮点的东西到摄像头前，用鼠标对准了点一下，再把那东西晃动晃动看看。会

发现一个矩形忽隐忽现，这说明跟踪到了，但还很不稳定。这是由于位图有 76800 个像素，每个像素的颜色值有 16777215 种，而选择只选择其中一种。所以，即使拿着的是一个纯色的东西，但由于光照、影子、材质、形状的不同，会导致物体在移动时，表面颜色有所差异。

为此，需要简化物体的颜色。这类方法有很多，我们从模糊开始。为创建的 Video 对象加一个模糊滤镜：

```
_vid = new Video(320, 240);
_vid.attachCamera(_cam);
_vid.filters = [new BlurFilter(10, 10, 1)];
```

这或多或少有点帮助。不过虽然减少了临近颜色的差异性，但并没有明显的区别。

还有几个方法供我们试试，一个是 ColorTransform，一个是 copyChannel 还有一个 threshold。除了这些，我发现一个最棒的函数，paletteMap（调色板）。

paletteMap 把一张位图的红、绿、蓝（甚至包括透明）通道分别列为一个数组。其实一张图片上的每个像素的每个通道都是一个 0 到 255（16 进制 0x00 到 0xFF）的数字。这样，每个数组就包含 256 个元素。如果图片上一个像素的某个通道值为 127，paletteMap 会从对应通道的数组取第 127 个元素的值来改变这个像素的颜色值。好晕哦~不管怎样，一个数组能包含的不仅仅是 0 到 255 的数字。所以，红色通道对应的数组，包含的值是从

0x000000 到 0xFF0000，绿色包含的值是从 0x000000 到 0x00FF00，蓝色则是 0x000000 到 0x0000FF。因此，如果把这几个数组的元素都倒置一下，那么最终会得到一张反色的图片。同理，可以改变图片的红绿通道等等之类操作。

我们要做的是减少每个通道的颜色数量，也就是要把众多相近的颜色分成几个等级。比如，红色通道 0 到 15 的都算做 0(0x00)，16 到 31 的都算做 16 (0x10)，32 到 63 的都算做 32(0x20)，以此类推。对另外两个通道也如法炮制，每个通道分 16 种等级，这样下来，一张图片就只有 4096 种颜色了。（书上写 4086 估计是打错， $16 \times 16 \times 16 = 4096$ ）上面说的写成算法就是 makePaletteArrays：

```
private function makePaletteArrays():void
{
    _red = new Array();
    _green = new Array();
    _blue = new Array();
    var levels:int = 8;
    var div:int = 256 / levels;
    for(var i:int = 0; i < 256; i++)
    {
        var value:Number = Math.floor(i / div) * div;
        _red[i] = value << 16;
        _green[i] = value << 8;
        _blue[i] = value
    }
}
```

看看这个函数的效果：

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

```

import flash.geom.Point;
import flash.geom.Rectangle;
import flash.media.Camera;
import flash.media.Video;
public class ColorTracking extends Sprite
{
    private var _cam:Camera;
    private var _vid:Video;
    private var _bmpd:BitmapData;
    private var _cbRect:Sprite;
    private var _color:uint = 0xffffffff;
    private var _red:Array;
    private var _green:Array;
    private var _blue:Array;
    public function ColorTracking()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;
        _cam = Camera.getCamera();
        _cam.setMode(320, 240, 15);
        _vid = new Video(320, 240);
        _vid.attachNetStream(_cam);
        _vid.filters = [new BlurFilter(10, 10, 1)];
        _bmpd = new BitmapData(320, 240, false);
        addChild(new Bitmap(_bmpd));
        _cbRect = new Sprite();
        addChild(_cbRect);
        makePaletteArrays();
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
        stage.addEventListener(MouseEvent.CLICK, onClick);
    }
    private function makePaletteArrays():void
    {
        _red = new Array();
        _green = new Array();
        _blue = new Array();
        var levels:int = 8;
        var div:int = 256 / levels;
        for(var i:int = 0; i < 256; i++)
        {
            var value:Number = Math.floor(i / div) * div;
            _red[i] = value << 16;
            _green[i] = value << 8;
            _blue[i] = value
        }
    }
    private function onClick(event:MouseEvent):void
    {
        _color = _bmpd.getPixel(mouseX, mouseY);
    }
    private function onEnterFrame(event:Event):void
    {
        _bmpd.draw(_vid, new Matrix(-1, 0, 0, 1, _bmpd.width, 0));
    }
}

```

```

_bmpd.paletteMap(_bmpd, _bmpd.rect, new Point(), _red, _green, _blue);
var rect:Rectangle = _bmpd.getColorBoundsRect(0xffffffff, _color, true);
_cbRect.graphics.clear();
_cbRect.graphics.lineStyle(1, 0xff0000);
_cbRect.graphics.drawRect(rect.x, rect.y, rect.width, rect.height);
}
}
}

```

这样仍然不够完美，一般碰到比较亮的背景就比较郁闷。如果物体的颜色比较突出，又是一个比较暗的环境，效果就不错。同样，也可以修改颜色的分级使之稳定。实际上，为了能让用户修正数值而获得敏感控制，通道的调色板每次都需要重新计算。

### 将跟踪颜色视作输入

好了，我们能够跟踪到这个矩形了，那这么做的意义是什么呢？实际上，我们可以根据它的位置来移动东西。接下来的例子中，创建的一个球会跟随这个矩形一起移动。你可以用来作出很诡异的对象跟随画面移动的效果，也可以在两个不相连的画面中作出隔山打牛的效果。

代码中创建的球会跟随这个被跟踪的矩形一起移动：

代码 ColorTracking.as

```

package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.BlendMode;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;
    import flash.geom.Rectangle;
    import flash.media.Camera;
    import flash.media.Video;

    public class MotionTracking extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
        private var _newFrame:BitmapData;
        private var _oldFrame:BitmapData;
        private var _blendFrame:BitmapData;
        private var _cbRect:Sprite;
        private var _paddle:Sprite;

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

            _cam = Camera.getCamera();
            _cam.setMode(320, 240, 15);
            _vid = new Video(320, 240);
            _vid.attachCamera(_cam);
            _vid.filters = [new BlurFilter(10, 10, 1)];
        }
    }
}

```

```

        _newFrame = new BitmapData(320, 240, false);
        addChild(new Bitmap(_newFrame));
        _oldFrame = _newFrame.clone();
        _blendFrame = _newFrame.clone();

        _cbRect = new Sprite();
        addChild(_cbRect);

        _paddle = new Sprite();
        _paddle.graphics.beginFill(0xffffffff);
        _paddle.graphics.drawRect(-100, -20, 200, 40);
        _paddle.graphics.endFill();
        _paddle.x = stage.stageWidth / 2;
        _paddle.y = stage.stageHeight - 50;
        addChild(_paddle);

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

private function onEnterFrame(event:Event):void
{
    _blendFrame.draw(_oldFrame);
    _newFrame.draw(_vid, new Matrix(-1, 0, 0, 1, _newFrame.width, 0));
    _oldFrame.draw(_newFrame);
    _blendFrame.draw(_newFrame, null, null, BlendMode.DIFFERENCE);
    _blendFrame.threshold(_blendFrame, _blendFrame.rect, new Point(), "<", 0x00330000,
    0xff000000, 0x00ff0000, true);

    var rect:Rectangle = _blendFrame.getColorBoundsRect(0xffffffff, 0, false);
    _cbRect.graphics.clear();
    _cbRect.graphics.lineStyle(1, 0xff0000);
    _cbRect.graphics.drawRect(rect.x, rect.y, rect.width, rect.height);

    if(!rect.isEmpty())
    {
        if(rect.x < _blendFrame.width / 2)
        {
            _paddle.x -= 20;
        }
        else
        {
            _paddle.x += 20;
        }
    }
}
}
}

```

在构造函数中，创建球并加入显示列表。在 enterFrame 处理函数中，首先看看颜色范围的矩形是不是存在，假如一个像素都没有，说明不存在。那么就不移动球。如果不做这个判断，当矩形不存在时球就会突然跳到画面左上角。接着就是矩形存在，让球贴着矩形移动即可。

需要说明一下，这只不过是跟踪颜色的一种做法。强烈建议各位再试试其它诸如 threshold, colorChannel 等位图操作方法，以及 ColorTransform、滤镜和混合模式等方法。而某些特殊的颜

色跟踪法，是专门用来区别明亮区域的。

接下来，我们看看更一般的跟踪移动方法。

## 分析移动区域

在之前的例子中，我们尝试跟踪到了一个在明暗反差比较大的环境中的物体，或者是有着特别颜色的物体。在这一节，我们虽然还不去涉及如何跟踪物体的具体轨迹，但会知道如何判断是否有移动。

一个基本概念是：如果有移动，每帧的画面会明显不同。所以，如果发现两帧画面中位图的像素有不同的地方，就能知道发生了移动。

有两个潜在元素。第一，我们需要两张位图。第二，我们还需要一个比较函数。如果，你正在想着是否需要遍历所有像素来进行比较，那么我告诉你，这里有一个很实用的技巧：使用混合模式。

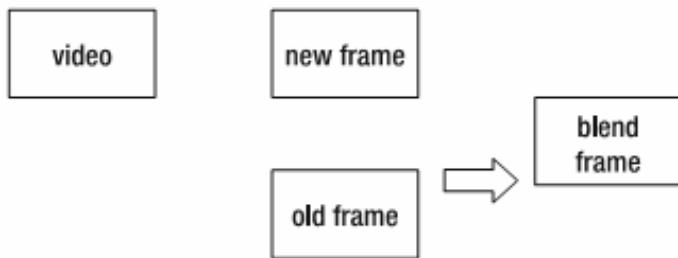
当采用 `BitmapData` 的 `draw` 函数来绘制对象时，会有几个参数供选择。可能各位已经见过参数 `matrix` 用来在绘制时控制形变和定位了。而在 `matrix` 之后还有几个参数，紧接着的是 `ColorTransform`，再之后就是混合模式。

绘制时如果不指定混合模式，新的像素值就会完全覆盖以取代存在的像素值。这也是我们至今为止一直在做的事情。如果使用混合模式，新的像素会影响已存在的像素，两张图片会以一种特别的方式混合在一起。而此刻，我们要用的混合模式叫做 `difference`（差异），它对两张图片的红、绿、蓝三个通道的每个像素进行一次比较，然后给出它们之间的相减所得的差值。如果两个像素完全一致，那么结果就是 0，也就是黑色，否则就是别的其它什么值（颜色）。这样，我们就把跟踪移动的问题简化了，只要寻找非黑色区域即可。（译：关于更多混合模式的原理，看这里

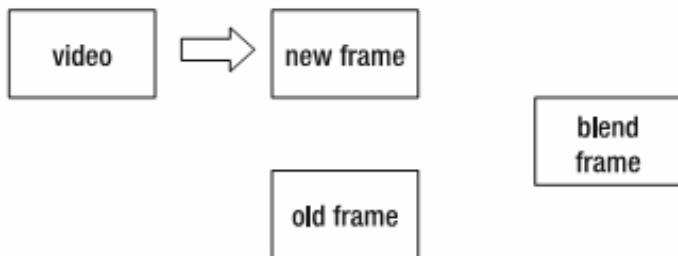
[http://www.pegtop.net/delphi/articles/blendsmodes^\\_~](http://www.pegtop.net/delphi/articles/blendsmodes^_~)）

为此，两张位图可能不太够用，三张会比较清晰：一张代表旧的一帧画面(`old`)，一张代表新的一帧画面(`new`)，还有一张是混合画面(`blend`)。如图 5-7。

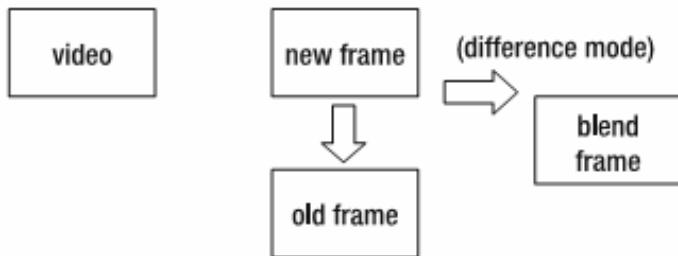
### 1. Draw old frame to blend frame.



### 2. Draw video to new frame.



### 3. Draw new frame to old frame and blend frame.



**Figure 5-7. Combining old and new frames with the difference blend mode**

三张位图就是三个 BitmapData 对象。首先把 old 赊到 blend 上，这里不用混合模式，是原样照搬。然后，把视频赊到 new 上。接着，把 new 赊到 old 上，这意思就是 new 上的东西在下一轮就是 old 了。最后，通过 difference 模式把 new 绘制在 blend 上。

上面的过程，写成代码就是这样：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.BlendMode;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.geom.Matrix;
    import flash.media.Camera;
    import flash.media.Video;
    public class MotionTracking extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
```

```

private var _newFrame:BitmapData;
private var _oldFrame:BitmapData;
private var _blendFrame:BitmapData;
public function MotionTracking()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    _cam = Camera.getCamera();
    _cam.setMode(320, 240, 15);
    _vid = new Video(320, 240);
    _vid.attachCamera(_cam);
    _vid.filters = [new BlurFilter(10, 10, 1)];
    _newFrame = new BitmapData(320, 240, false);
    _oldFrame = _newFrame.clone();
    _blendFrame = _newFrame.clone();
    addChild(new Bitmap(_blendFrame));
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
private function onEnterFrame(event:Event):void
{
    _blendFrame.draw(_oldFrame);
    _newFrame.draw(_vid, new Matrix(-1, 0, 0, 1, _newFrame.width, 0));
    _oldFrame.draw(_newFrame);
    _blendFrame.draw(_newFrame, null, null, BlendMode.DIFFERENCE);
}
}

```

现在能理解了吧？创建了三个位图对象，只有最后一个才加入显示列表。在每一帧处理的函数中呈现了图 5-7 所描述的执行顺序。



**Figure 5-8.** Combining two frames with a difference blend mode

当运行刚启动，会出现一张纯黑色的矩形。但接着就会看到鬼一样移动的轮廓（如图 5-8）。这个轮廓就是两帧画面的不同之处。

现在，让我们用一个颜色范围来捕捉变化的区域。和早先的例子一样，创建`_cbRect`，而`onEnterFrame` 所做的事情稍有不同：

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

```

{
    _blendFrame.draw(_oldFrame);
    _newFrame.draw(_vid, new Matrix(-1, 0, 0, 1, _newFrame.width,0));
    _oldFrame.draw(_newFrame);
    _blendFrame.draw(_newFrame, null, null,BlendMode.DIFFERENCE);
        var rect:Rectangle = _blendFrame.getColorBoundsRect(0xffffffff,0x000000, false);
    _cbRect.graphics.clear();
    _cbRect.graphics.lineStyle(1, 0xff0000);
    _cbRect.graphics.drawRect(rect.x, rect.y, rect.width, rect.height);
}

```

注意现在我们追踪的不是黑色(0x000000)，所以第三个参数是 false。但当兴致勃勃的运行后，发现矩形范围几乎包含整个画面。嗯...看来有些看似黑色的地方并不真正都是黑色，只是因为颜色很接近，肉眼却不足以区别。

赶快从百宝箱中拿出我们的利器——threshold——又一个比较复杂的函数，但它同样很强大。对它，在我得心应手之前曾有过一段极其艰苦的日子，所以如果你也有这样的经历，那么你并不是一个人在战斗。

threshold 函数会拿一个值和位图的每个像素值进行一次比较。而比较的方式包含了所有逻辑运算：`<`、`<=`、`==`、`>=`、`>`、`!=`。同样的，它也能设置掩码来过滤出需要比较的颜色通道。

掩码重要是因为，两个颜色的充分比较并不常用。比如，0x010000 是最小的红色值，它几乎就是黑色的，而 0x0000FF 是百分之百的蓝色，比 0x010000 要亮的多。但结果确实蓝色要“小于”几乎是黑色的值，因为从数字大小讲，确实是这样。

```

bitmapData.threshold(sourceBitmapData, sourceRect, destPoint, operation,
                     threshold, color, mask, copySource)

```

第一个参数 sourceBitmapData 就是要进行比较的位图对象，而调用 threshold 的位图对象则呈现比较的结果。大多数情况下(包括我们的例子)两者都是同一个对象，但实际上是可以不同的。

sourceRect 和 destPoint 决定比较的区域和呈现结果的起始点。一般前者采用 BitmapData 的 rect 属性，后者采用(0, 0)的 Point，其意义就是操作用于整张位图。

operation (操作符) 是一个字符串，包含前面提到过的`<`、`<=`、`==`、`>=`、`>`、`!=`。

threshold 是用来和每个像素作比较用的数值。

color 是在比较结果为 true 时，像素被设置的颜色值。

mask 就是指定通道的掩码啦。

copySource 是把结果复制到的另一张位图。

而如果，比较结果为 false，则绘制 source 位图的像素。在我们的例子中只有一张位图，所以不用考虑太多。并且由于每个像素都很接近黑色，也就不用过多考虑通道的问题，随便采用红色通道的就行了。

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

```

{
    _blendFrame.draw(_oldFrame);
    _newFrame.draw(_vid, new Matrix(-1, 0, 0, 1, _newFrame.width,0));
    _oldFrame.draw(_newFrame);
    _blendFrame.draw(_newFrame, null, null,BlendMode.DIFFERENCE);
    _blendFrame.threshold(_blendFrame, _blendFrame.rect, new Point
(0, "<", 0x00330000, 0xff000000, 0x00ff0000, true);
        var rect:Rectangle = _blendFrame.getColorBoundsRect(0xffffffff,0x000000, false);
    _cbRect.graphics.clear();
    _cbRect.graphics.lineStyle(1, 0xff0000);
    _cbRect.graphics.drawRect(rect.x, rect.y, rect.width,rect.height);
}

```

看像素的红色通道(因为我们的掩码是 0x00FF0000)，如果值小于(<)0x00330000，就用黑色(0xFF000000)填充，否则复制原图(同一个图，别想太多)。把以上发生的事情说通俗点就是，如

果颜色接近黑色，就让它成为黑色。

需要注意这里使用的都是 32 位的数值。因为 threshold 就喜欢这样。即使对不透明的位图也是如此，如果不指定透明通道，就会画一个透明的像素。所以黑色用的是 0xFF000000 而不是 0x00000000。

现在再试试，是不是好很多了。绘制的矩形区域差不多就是移动的区域。

那么这个矩形我们该怎么用呢？再来打破一下僵化的思维吧。我首先想到的是一系列有边有界的游戏，比如弹壁球。只不过现在的挡板是可以自由移动的。（代码 MotionTracking）

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.BlendMode;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;
    import flash.geom.Rectangle;
    import flash.media.Camera;
    import flash.media.Video;

    public class MotionTracking extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
        private var _newFrame:BitmapData;
        private var _oldFrame:BitmapData;
        private var _blendFrame:BitmapData;
        private var _cbRect:Sprite;
        private var _paddle:Sprite;

        public function MotionTracking()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _cam = Camera.getCamera();
            _cam.setMode(320, 240, 15);
            _vid = new Video(320, 240);
            _vid.attachCamera(_cam);
            _vid.filters = [new BlurFilter(10, 10, 1)];
            _newFrame = new BitmapData(320, 240, false);
            addChild(new Bitmap(_newFrame));
            _oldFrame = _newFrame.clone();
            _blendFrame = _newFrame.clone();
            _cbRect = new Sprite();
            addChild(_cbRect);
            _paddle = new Sprite();
            _paddle.graphics.beginFill(0xffffffff);
            _paddle.graphics.drawRect(-100, -20, 200, 40);
            _paddle.graphics.endFill();
        }
    }
}
```

```

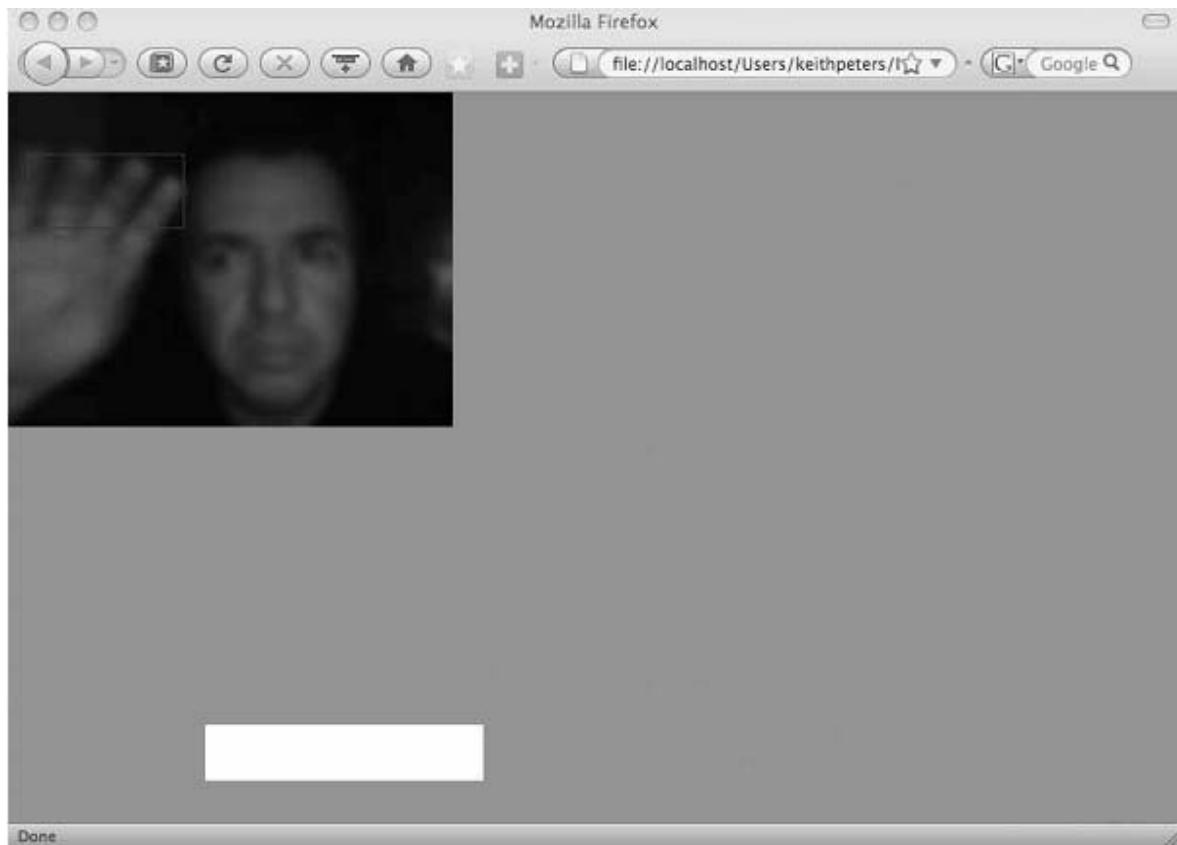
_paddle.x = stage.stageWidth / 2;
_paddle.y = stage.stageHeight - 50;
addChild(_paddle);
addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    _blendFrame.draw(_oldFrame);
    _newFrame.draw(_vid, new Matrix(-1, 0, 0, 1, _newFrame.width, 0));
    _oldFrame.draw(_newFrame);
    _blendFrame.draw(_newFrame, null, null, BlendMode.DIFFERENCE);
    _blendFrame.threshold(_blendFrame, _blendFrame.rect, new Point(), "<", 0x00330000,
0xff000000, 0x00ff0000, true);

    var rect:Rectangle = _blendFrame.getColorBoundsRect(0xffffffff, 0, false);
    _cbRect.graphics.clear();
    _cbRect.graphics.lineStyle(1, 0xff0000);
    _cbRect.graphics.drawRect(rect.x, rect.y, rect.width, rect.height);
    if(!rect.isEmpty()) {
        if(rect.x < _blendFrame.width / 2) {
            _paddle.x -= 20;
        } else {
            _paddle.x += 20;
        }
    }
}
}
}

```

就这么简单。创建一个挡板，如果颜色范围的矩形在右边，挡板就向右移动，范围向左，挡板也向左。灵感来了吧？这只能算个启发，后面的事情就留给各位自己玩了。注意，我没有把混合图片再显示出来。当然了，threshold 和颜色范围等一些工作都在默默的进行着，只是对于它们的位图是否能看见而已，隐藏这些过程就好像是在变魔术。图 5-9



**Figure 5-9.** Controlling an object with hand motion  
边缘检测

在本章的最后一站，我们将重新诠释一个耳熟能详的效果：对图像进行边缘检测并附有东西着落。此交互式的效果常常陈列于儿童博物馆等地方，屏幕中的雪花或者蝴蝶会停落在孩子的镜像上。在 Flash 界这个效果也已经出现过几次了，最值得关注的应该是由 Grant Skinner ([www.gskinner.com](http://www.gskinner.com)) 所做的，而我们现在也即将开始。

对此有多种实现方式，能立即想到的是用 ConvolutionFilter (卷积滤镜) 创建一个水平方向的边缘检测。位图通过一个卷积滤镜，使其每个像素和自身周围一圈的像素进行比较。每个像素的值是周围像素值通过一定关系相加的总和再除以一个系数。卷积矩阵通常在图像处理中被广泛的应用于产生模糊、锐化、浮雕、边缘检测和强化等效果。

图 5-10 展示的是作为一个模糊滤镜的基本设置。位于中心的像素是被作用的像素，其周围一圈的相加总和的权重都是 1。换句话说，再相加之前的相乘系数是 1。最后总和除以 9，结果再赋给中心像素。所以，模糊从数学角度说就是周围像素和的平均值，而模糊的大小和范围的大小成正比。图 5-10

**blur matrix**

1	1	1
1	1	1
1	1	1

add all pixels, divide by 9  
assign result to center pixel

**Figure 5-10.** Convolution matrix used for a blur

如果对卷积矩阵感兴趣，可以在网络上搜索一下，会有大量的信息和例子，那些大多数都能用 AS 来实现。接下来我们要经过一番设置，创建一个可以检测水平方向边缘的滤镜。卷积滤镜的构造函数是这样：

```
ConvolutionFilter( matrixX, matrixY, matrix, divisor );
```

实际上，这里面有很多参数是可选的，不过对我们来说都是需要进行设置的。  
matrixX 和 matrixY 是两个数字，用来确定 matirx（矩阵）的行列数。  
matirx 是一个包含了权重系数的数组。这个矩阵数组的长度大小应该正好等于 matrixX 乘以 matrixY。

divisor 是最后总和的除数。

所以，如果要创建一个图 5-10 表示的模糊滤镜，那就应该这样：

```
new ConvolutionFilter(3, 3, [1,1,1,1,1,1,1,1,1], 9);
```

设置一个元素全是 1 的 3x3 大小的矩阵，且除数为 9。

下面这个是我们所说的水平边缘：

```
new ConvolutionFilter(1, 3, [0,4,-4], 1);
```

超简单。就是创建一个 1x3 的矩阵，当前的像素乘以 4，下方的像素乘以 -4，忽略上方的像素，除数是 1，所以总和就是结果。你要是有兴趣，还是建议去看看卷积是如何工作的。这个效果就是把图像变暗，然后亮化显示在水平方向上检测到的边缘。通过下面的例子看看效果：

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.filters.ConvolutionFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;
    import flash.media.Camera;
    import flash.media.Video;
    public class EdgeTracking extends Sprite
    {
        private var _cam:Camera;
        private var _vid:Video;
        private var _bmpd:BitmapData;
        public function EdgeTracking()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _cam = Camera.getCamera();
            _cam.setMode(320, 240, 15);
            _vid = new Video(320, 240);
            _vid.attachCamera(_cam);
            _vid.filters = [new ConvolutionFilter(1, 3, [0, 4, -4]),
                           new BlurFilter()];
            _bmpd = new BitmapData(320, 240, false);
            addChild(new Bitmap(_bmpd));
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void
        {
            _bmpd.draw(_vid, new Matrix(-1, 0, 0, 1, _bmpd.width, 0));
        }
    }
}
```

除了 ConvolutionFilter，这里没什么新内容。为了平滑，同时增加了一个模糊滤镜，结果如图 5-11。

图中可以清晰的看到水平方向上的边缘，其余部分则是暗黑色的。（注意：实际上我说的“暗黑色”应该提醒到你，马上将会看到 threshold 的使用）



**Figure 5-11.** Creating horizontal edge detection

那么，接下来该做什么呢？让雪花飘落在我的头上如何？首先创建一个简单的雪花类。代码

Snow

```
package{
    import flash.display.Sprite;
    public class Snow extends Sprite {
        public var vx:Number;
        public var vy:Number;
        public function Snow()  {
            graphics.beginFill(0xffffffff, .7);
            graphics.drawCircle(0, 0, 2);
            graphics.endFill();
            vx = 0;
            vy = 1;
        }
        public function update():void {
            vx += Math.random() * .2 - .1;
            vx *= .95;
            x += vx;
            y += vy;
        }
    }
}
```

这个类会画一个小圆点代表雪花，还有 x 和 y 的速度以及 update 函数。

我们打算每过一帧就创建一个雪花实例，并将其加入一个数组，然后更新数组中所有的雪花状态。在这之前，我们还要先检测一下雪花是否接触到边缘。也就是说，雪花所在的(x, y)处的像素值是否大于 0。现在轮到 threshold 粉墨登场，把“暗黑色”变成纯黑色了。修改后的代码如下：

```
package {
```

```

import flash.display.Bitmap;
import flash.display.BitmapData;
import flash.display.Sprite;
import flash.display.StageAlign;
import flash.display.StageScaleMode;
import flash.events.Event;
import flash.filters.BlurFilter;
import flash.filters.ConvolutionFilter;
import flash.geom.Matrix;
import flash.geom.Point;
import flash.media.Camera;
import flash.media.Video;
public class EdgeTracking extends Sprite
{
    private var _cam:Camera;
    private var _vid:Video;
    private var _bmpd:BitmapData;
    private var _flakes:Array;
    public function EdgeTracking()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;
        _cam = Camera.getCamera();
        _cam.setMode(320, 240, 15);
        _vid = new Video(320, 240);
        _vid.attachCamera(_cam);
        _vid.filters = [new ConvolutionFilter(1, 3, [0, 4, -4]), new BlurFilter()];
        var vid2:Video = new Video(320, 240);
        vid2.attachCamera(_cam);
        vid2.scaleX = -1;
        vid2.x = 320;
        addChild(vid2);
        _bmpd = new BitmapData(320, 240, false);
        _flakes = new Array();
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }
    private function onEnterFrame(event:Event):void
    {
        _bmpd.draw(_vid, new Matrix(-1, 0, 0, 1, _bmpd.width, 0));
        _bmpd.threshold(_bmpd, _bmpd.rect, new Point(), "<", 0x00220000, 0xff000000, 0x00ff0000,
true);
        var snow:Snow = new Snow();
        snow.x = Math.random() * _bmpd.width;
        addChild(snow);
        _flakes.push(snow);
        for(var i:int = _flakes.length - 1; i >= 0; i--) {
            snow = _flakes[i] as Snow;
            if(_bmpd.getPixel(snow.x, snow.y) == 0) {
                snow.update();
                if(snow.y > _bmpd.height) {
                    removeChild(snow);
                    _flakes.splice(i, 1);
                }
            }
        }
    }
}

```

```
        }  
    }  
}  
}
```

一开始创建的 \_flakes 数组就是用于保存雪花实例的。

注意，我们使用了另一个 video 作为显示对象。这就像上一节所做的含义一样，我们并不让用户看到实际的处理情况。

接着调用 threshold，就不用多说了。

每一帧增加一个新的雪花实例，加入显示列表和 \_flakes 数组，然后遍历数组，更新雪花状态，这也没啥好多说。

判断雪花位置的像素值，如果不为 0，说明不是黑色，即边缘处，那么就停止更新雪花的状态，让其停留于此即可。

最后判断雪花是否落出画面，是的话就删除它。以上结果如图 5-12



Figure 5-12. Video snow!

似乎这并不是一个什么特别有用的例子，不过它挺好玩儿的。如果你已经对此有所领悟，既可开始你自己的创意。我期盼您的创作。

### 总结

这一章主要讲了一些使用音频和视频的输入所做的交互试验。其中很多都可以用于 Flash 应用和游戏，至少我希望能给你带来一些灵感。我再次期盼在不久的将来，能看到有读者以此做出一些不错的东西。

## 第六章 高等物理：数值积分

在上本书中，我讲述了一些用于 Flash 程序运动的基本物理公式。归纳大致如下：在每一帧，加速度叠加于速度上，速度叠加于位置上。我知道这并不精确也有很多缺陷，但我同样相信在 Flash 中以此可以做出很多不错的游戏和作品。

在本书第一版写作过程中，我发现这就是所谓的欧拉积分。它确实不怎么精确。虽然简单明了且被广泛使用，但如果碰到瓶颈，还会有什么更好的选择吗？这一章的目的之一，就是要对此问题给出粗略的答案。我们将研究什么是数值积分，欧拉积分做错了什么，以及考虑何时何地为何去使用它们。

同时，我还会为你在今后听到欧拉这个 18 世纪伟大的物理学家、数学家的名字后解除一些困惑。当然了，如果你非要在饭桌上谈起欧拉积分，那破坏气氛是在所难免了。

数值积分以及为什么欧拉“不好”

首先，我这里说的“不好”是指它不精确。换句话说，如果用欧拉来模拟对象因受力而做的运动，与现实中的情况相比是不完全相符的。不管怎样，对于一些小游戏、动画、非科研模拟来说，它足够用了，而且大多数人也几乎看不出有什么不对劲。但当需要一个更高精度的情况时，欧拉就力不从心了。

为什么力不从心呢？因为有关运动、速度、质量和加速度之间的关系公式已经产生并且很好的工作了几个世纪了（感谢牛顿）。记得在高中的代数课上，有这么个问题“一个小孩儿站在离地面 50 米的屋顶，以 30 米每秒的速度垂直向上抛出一个球……”然后让你计算球在落地时飞行了多少时间或者在某一时刻球的高度。为此你会用到下面的公式，其中  $t$  代表时间，单位秒， $v_0$  是初速度， $h_0$  是初始高度， $y$  是给定时间上的高度：

$$y = -16t^2 + v_0*t + h_0$$

用此公式不会有什么问题，它够精确。那为什么我们没有用这个公式呢？因为它描述的是单个对象以同一弧度在单个轴上的运动，除了重力外不考虑其它任何外力作用。如果改为向前抛出，那么像空气阻力，弹力，其它对象的碰撞等等都要考虑进去，一下子就变得相当复杂。

一个比较可行的做法是把整个过程视为一段一段的，考虑段与段之间的情况，计算此刻对象的速度和位置。这个过程就叫积分。如图 6-1。

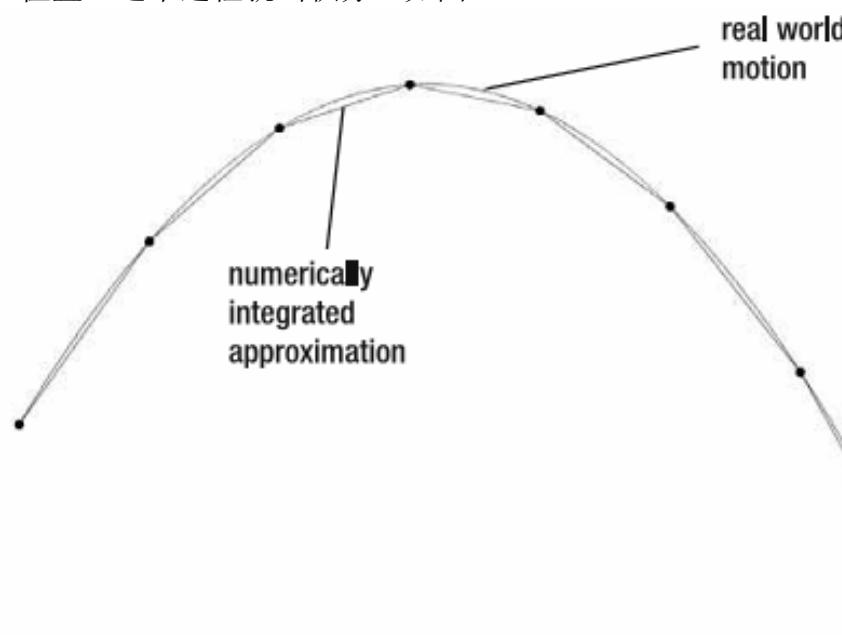


Figure 6-1. Real world motion and integrated approximation (ideal)

图 6-1

图中的一条弧线是物体真正的移动过程，紧贴弧线的一条比较粗糙的折线就是用积分来描述的速度和位置的关系。折线的段数越多，轨迹就越接近弧线。

问题是，现实中的物理作用力是连续不断的，力改变物体的运动也是持续进行的，不会是一段一段跳跃式的。就比如，重力导致一个物体持续下落，其速度是不断递增的，因此物体位置的变化也是平滑的。如果重力的作用也是每帧才来一下，那么在帧与帧之间的空隙，我们就不受重力作用了。所以，虽然是更新的越频繁就越精确，但始终存在误差。

欧拉积分对此的解决方式是忽略误差，这导致它不会很精确。我无法更简单的来描述这些内容。看图 6-2，展示了欧拉以一秒为间隔的计算出现的误差。

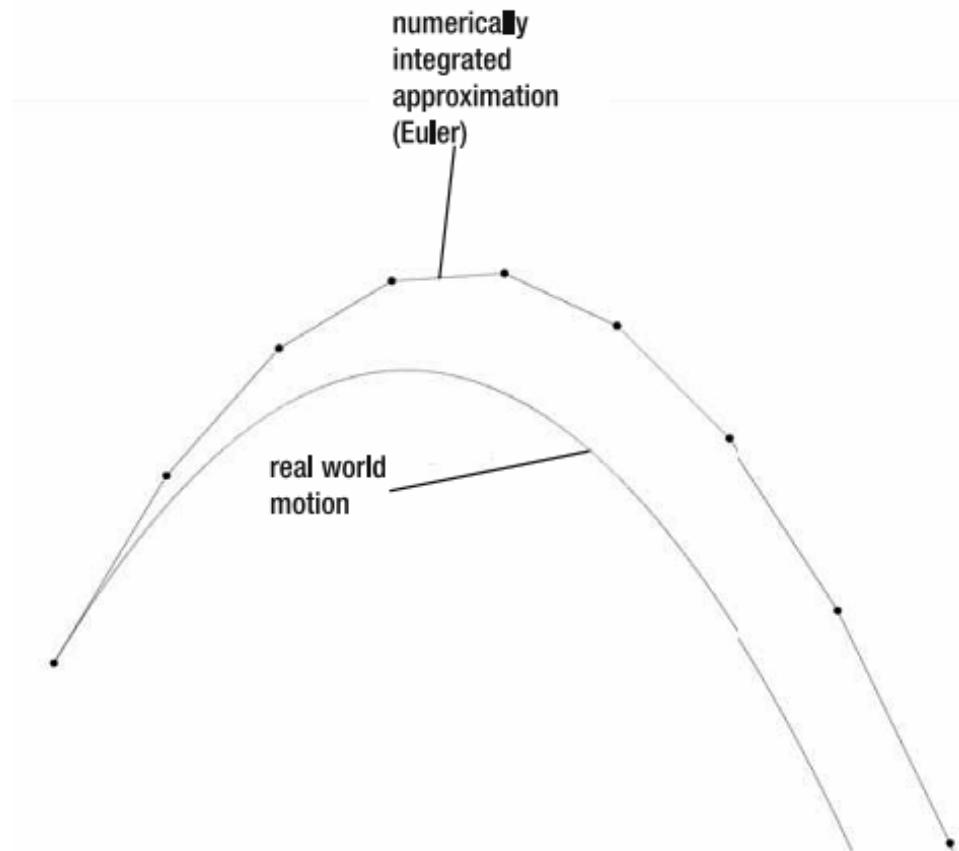


Figure 6-2. Real world motion and Euler integration

图 6-2

当然了，间隔越小结果越准，但只有在无限小的时候结果才是最准的。

此外，欧拉模拟也不稳定，很容易产生跳跃现象。计算中的误差有时会导致速度不断增长，以至于整个东西垮掉。

如果欧拉不够好，那还有什么选择呢？有两个选择，一个叫 **Runge–Kutta integration**，一个叫 **Verlet integration**。和欧拉一样，它们都是以发明者的名字命名的：Carl Runge, Martin Wilhelm Kutta 和 Loup Verlet。这两种方法着重于不同的领域，Runge–Kutta 用于高精度计算领域，Verlet 常常用于电脑图像领域，以创建一种“玩具式”的反向运动。如果这些对你来说没什么意义，别急。马上你会发现这些都是很简单的工作，但却能带来很好的效果，让我们从 Runge–Kutta 开始吧。

#### Runge–Kutta integration

由上一节讨论中得出，欧拉的缺陷在于它分段式的整合加速度、速度和位置，而这些元素的变化是连续性的，其近似的整合结果就导致了不精确的产生。

Runge–Kutta 不会出现这种不精确的问题，这是由于它做了一些额外的计算来获得更准确的值。对此要有一个清晰地认识，就是说，使用 Runge–Kutta 并不意味着会得出一个绝对精确的结果，只是相对的精确。

到底有多精确？这个问题我很难回答，但我要说，至少在 Flash 界不会需要如此精确的运算，如果你的程序面对这么一个环境，首先要考虑的不应该是采用 Flash。

我第二次听到术语 Runge-Kutta 是一次在波士顿的 Flash 大会上，James Battat 描述了一系列物理现象，是他和他的同事用 AS2.0 以 Runge-Kutta 开发出的一套哈福大学机械系统的工业级物理课件。其作为一个不错的案例趋势我对此进行了深入的研究。

平衡是我们熟悉的自然规律。Runge-Kutta 是更精确了，但计算量也更大了，这意味着 CPU 的大量使用和低下的执行效率，等等。所以，不要为有了更高的精确度而心花怒放，要确定这额外的精确度是否必要。即使写到这里，我还是要尽量提醒各位，就 Flash 来说几乎碰不到使用如此高精度的情况，不计后果的使用只会让 CPU 升温。

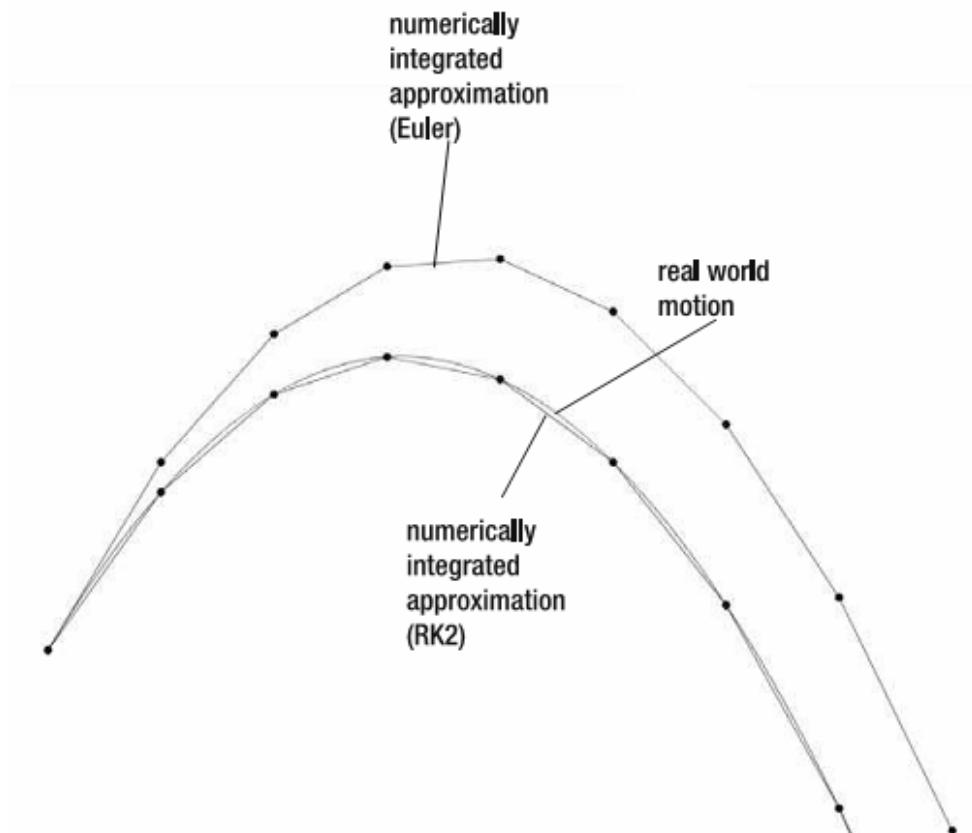
警告到此为止，还是先看看 Runge-Kutta 是如何工作的。为了方便以下简称其为 RK。

RK 也是分段式整合，但它不是每次都盲目的把加速度加于速度之上，速度再加于位置之上，而是对曲线多次采样后取得最接近曲线的一点。这个想法不算很复杂，我们慢慢深入，看看它到底如何实现。

有两种常用的 RK：Runge-Kutta second order integration（简称 RK2），Runge-Kutta fourth order integration（简称 RK4）。数字代表了采样阶数。RK2 是每次 2 种采样，RK4 就是 4 种。

先看 RK2，因为它比较容易理解。你可能听说过它的别称：欧拉改良版(improved Euler)欧拉中点版(midpoint Euler)或者中点法(the midpoint method)，这些都是指同一个东西。从别称能看出，RK2 和欧拉法区别不大，只是就中点多做了些研究。

其策略是算出物体当前状态下的加速度、速度和位置(欧拉法)，然后在此加速度和速度算出下一个状态，最后取两个状态的平均值作为物体的状态。也就是说，欧拉认为一段时间内的状态是一致的，RK2 则认为一段时间内的状态是，前一段状态和后一段状态的平均值。再说一遍，这并不是百分之百的准确，但比欧拉要准确的多，看图 6-3 显示了欧拉和 RK2 的结果比较。



**Figure 6-3.** Real world motion, Euler integration, and RK2

图 6-3

上图中几乎看不出 RK2 有什么误差。够好的了吧。

当然，如果觉得还不够好，还有 RK4 呢，它会以四次采样平均求出曲线的斜率。我不想再在这里多做解释，只要记住它更精确就行了。

## 时间驱动的运动

好了，好了，说的够多了。我们要看代码！

在此之前，我还要多啰嗦一句。这章所有的代码都是时间驱动，而不是帧驱动的。这个话题详细请看上本书的第 19 章，这里只做一些简述。

很多简单的 Flash 代码，都有速度和加速度两个概念，对象在每一帧都受它们的影响。由于对象移动是以像素为单位，且我们以每一帧作为时间间隔，所以它们的单位是“像素每帧”，而不是“米每秒”或者“公里每小时”。但当我们以高精度为目的工作时，可能更习惯于标准测量单位，至少时间单位是这样。所以，我们将以真实时间来控制更新物体的移动。

为此，我们需要用到 flash.utils.getTimer 来算出更新期间所逝去的时间。这个逝去的时间适用于所有标准运动公式，包括欧拉。比如，新位置 = 旧位置 + 速度 \* 逝去时间。不用帧驱动的唯一原因是“像素每帧”在更新期间逝去的帧数是 1。所以，新位置 = 旧位置 + 速度。

还有些要注意的，我们统一用 Point 来表示位置、速度和加速度。对于位置，一个 Point 对象在经过一系列计算后会把最终结果赋值给对象。对于速度，因为 Point 包含 x 和 y 两个属性，所以用此一个变量表示速度，就可以取代 vx, vy 两个变量。对于加速度，我们为此创建了一个函数返回一个包含了加速度值的 Point 对象，避免直接把重力加速度加于速度之上。目前只考虑重力的话，这个函数返回的 point 的 x 是 0, y 是重力加速度值，如果以后有更多

复杂的作用力，都可以在此函数内完成。考虑到大多数情况下，加速度会影响对象的速度和位置，所以该函数为此设有两个参数。

先从熟悉的欧拉开始。

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

    public class Euler extends Sprite
    {
        private var _ball:Sprite;
        private var _position:Point;
        private var _velocity:Point;
        private var _gravity:Number = 32;
        private var _bounce:Number = -0.6;
        private var _oldTime:int;

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

            _ball = new Sprite();
            _ball.graphics.beginFill(0xff0000);
            _ball.graphics.drawCircle(0, 0, 20);
            _ball.graphics.endFill();
            _ball.x = 50;
            _ball.y = 50;
            addChild(_ball);
        }
    }
}
```

```

_velocity = new Point(100, 0);
_position = new Point(_ball.x, _ball.y);

_oldTime = getTimer();
addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    var time:int = getTimer();
    var elapsed:Number = (time - _oldTime) / 1000;
    _oldTime = time;

    var accel:Point = acceleration(_position, _velocity);
    _position.x += _velocity.x * elapsed;
    _position.y += _velocity.y * elapsed;
    _velocity.x += accel.x * elapsed;
    _velocity.y += accel.y * elapsed;

    // 检测如果对象超过边缘就弹回
    if(_position.y > stage.stageHeight - 20)
    {
        _position.y = stage.stageHeight - 20;
        _velocity.y *= _bounce;
    }
    if(_position.x > stage.stageWidth - 20)
    {
        _position.x = stage.stageWidth - 20;
        _velocity.x *= _bounce
    }
    else if(_position.x < 20)
    {
        _position.x = 20;
        _velocity.x *= _bounce;
    }

    _ball.x = _position.x;
    _ball.y = _position.y;
}

private function acceleration(p:Point, v:Point):Point
{
    return new Point(0, _gravity);
}
}
}

```

一开始创建了一个球，初始其位置和速度，并使用 getTimer 记录下当前的时间。该函数会返回程序总共运行的时间，单位毫秒。这个时间记录在 \_oldTime 里，且每一帧都会记录。新旧时间差就是上一帧到这一帧逝去的时间。除以 1000 是把毫秒换算成秒。

接着调用 acceleration 函数，获得对象所受的作用力。目前只有重力。

在速度叠加于位置之前，要先乘以逝去的时间。加速度也一样。因此，帧频越低，乘以的系数就越高，所得的值越大，球移动的幅度也越大。帧频越高则情况相反。实际上，球移动的距离已经

和帧频没有关系了，帧频的高低只决定了移动的平滑度。

还有一段反弹代码。等会讨论。

可能你在想“这球落的也太慢了，一点都不真实。”，因为我们把重力设置成了 32，放到真实世界中，就是说一个物体的下落速度以每秒 32 米的大小在不断递增，那么把米换成像素，我们的情况就好像是一个球从八九百米的高楼上落下——这效果不是很现实么！

用等比缩放会好点，比如把 100 像素看成 1 米。我们用一个变量来保存比例关系，然后在程序里面体现出来。

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

    public class Euler extends Sprite
    {
        private var _ball:Sprite;
        private var _position:Point;
        private var _velocity:Point;
        private var _gravity:Number = 32;
        private var _bounce:Number = -0.6;
        private var _oldTime:int;
        private var _pixelsPerFoot:Number = 10;

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

            _ball = new Sprite();
            _ball.graphics.beginFill(0xff0000);
            _ball.graphics.drawCircle(0, 0, 20);
            _ball.graphics.endFill();
            _ball.x = 50;
            _ball.y = 50;
            addChild(_ball);

            _velocity = new Point(10, 0);
            _position = new Point(_ball.x / _pixelsPerFoot, _ball.y / _pixelsPerFoot);

            _oldTime = getTimer();
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var time:int = getTimer();
            var elapsed:Number = (time - _oldTime) / 1000;
            _oldTime = time;

            var accel:Point = acceleration(_position, _velocity);
        }
    }
}
```

```

    _position.x += _velocity.x * elapsed;
    _position.y += _velocity.y * elapsed;
    _velocity.x += accel.x * elapsed;
    _velocity.y += accel.y * elapsed;

    if(_position.y > (stage.stageHeight - 20) / _pixelsPerFoot)
    {
        _position.y = (stage.stageHeight - 20) / _pixelsPerFoot;
        _velocity.y *= _bounce;
    }
    if(_position.x > (stage.stageWidth - 20) / _pixelsPerFoot)
    {
        _position.x = (stage.stageWidth - 20) / _pixelsPerFoot;
        _velocity.x *= _bounce
    }
    else if(_position.x < 20 / _pixelsPerFoot)
    {
        _position.x = 20 / _pixelsPerFoot;
        _velocity.x *= _bounce;
    }

    _ball.x = _position.x * _pixelsPerFoot;
    _ball.y = _position.y * _pixelsPerFoot;
}

private function acceleration(p:Point, v:Point):Point
{
    return new Point(0, _gravity);
}
}
}

```

现在感觉应该像是一个拳头大小的球从八九层楼梯上落下了吧（注意我还把初速度改小了很多），这下应该符合你的想象了。

时间驱动来实现欧拉就是这么个情况，下面该看看 Runge-Kutta 了。

## 编程 RK2

概述一下 RK2 的步骤，计算出每段开始时和结束时的加速度和速度，然后取平均值。

先用伪码实现。

首先，计算开始时的加速度，然后是位置和速度，这些和欧拉法是完全一致的。不过我们用新的变量来保存这些信息。

```

// position1 是对象的当前位置
// velocity1 是对象的当前速度
acceleration1 = acceleration(position1, velocity1)
position2 = position1 + velocity1 * time
velocity2 = velocity1 + acceleration1 * time
position2 和 velocity2 就是结束时对象的位置和速度。接下来求结束时的加速度:
acceleration2 = acceleration(position2, velocity2)

```

然后是 RK2 的关键一步，求两个状态下的平均速度和加速度：

```

position1 += (velocity1 + velocity2) / 2 * time
velocity1 += (acceleration1 + acceleration2) / 2 * time

```

开始时和结束时的平均速度乘以时间，得到改变的位移，加于当前位置，就是改变后的位置。同理加速度，就是改变后的速度。

这就是 RK2！看代码吧：

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

    public class RK2 extends Sprite
    {
        private var _ball:Sprite;
        private var _position:Point;
        private var _velocity:Point;
        private var _gravity:Number = 32;
        private var _bounce:Number = -0.6;
        private var _oldTime:int;
        private var _pixelsPerFoot:Number = 10;

        public function RK2()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _ball = new Sprite();
            _ball.graphics.beginFill(0xff0000);
            _ball.graphics.drawCircle(0, 0, 20);
            _ball.graphics.endFill();
            _ball.x = 50;
            _ball.y = 50;
            addChild(_ball);

            _velocity = new Point(10, 0);
            _position = new Point(_ball.x / _pixelsPerFoot, _ball.y / _pixelsPerFoot);

            _oldTime = getTimer();
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var time:int = getTimer();
            var elapsed:Number = (time - _oldTime) / 1000;
            _oldTime = time;

            var accel1:Point = acceleration(_position, _velocity);

            var position2:Point = new Point();
            position2.x = _position.x + _velocity.x * elapsed;
            position2.y = _position.y + _velocity.y * elapsed;

            var velocity2:Point = new Point();
            velocity2.x = _velocity.x + accel1.x * elapsed;
```

```

velocity2.y = _velocity.y + accel1.x * elapsed;

var accel2:Point = acceleration(position2, velocity2);

_position.x += (_velocity.x + velocity2.x) / 2 * elapsed;
_position.y += (_velocity.y + velocity2.y) / 2 * elapsed;

_velocity.x += (accel1.x + accel2.x) / 2 * elapsed;
_velocity.y += (accel1.y + accel2.y) / 2 * elapsed;

if(_position.y > (stage.stageHeight - 20) / _pixelsPerFoot)
{
    _position.y = (stage.stageHeight - 20) / _pixelsPerFoot;
    _velocity.y *= _bounce;
}
if(_position.x > (stage.stageWidth - 20) / _pixelsPerFoot)
{
    _position.x = (stage.stageWidth - 20) / _pixelsPerFoot;
    _velocity.x *= _bounce
}
else if(_position.x < 20 / _pixelsPerFoot)
{
    _position.x = 20 / _pixelsPerFoot;
    _velocity.x *= _bounce;
}

_ball.x = _position.x * _pixelsPerFoot;
_ball.y = _position.y * _pixelsPerFoot;
}

private function acceleration(p:Point, v:Point):Point
{
    return new Point(0, _gravity);
}
}
}

```

如果理解了上面的求解过程，代码就没什么好看的。运行一下看看是不是比欧拉更接近真实情况呢？什么！？你那边看起来一样？好吧，其实我这边也一样，但它确实略有不同啦。这就是我为什么说，在Flash领域欧拉已经足够我们用了。图6-3所描述的分段是以一秒为一段的，也就是放大的了的情况。如果Flash运行在每秒24帧左右，那每一段的间隔都很小，欧拉和RK2也就很接近了。

同时要注意，这个例子是一个非常简单的情况。如果增加额外受力或者间隔跨度加大，就能看到比较大的差异了。不管怎么样，如果精确度对你真的很重要，那现在该知道如何去做了吧。

等等，还有更好的！不仅仅是RK2，我们下面要讨论RK4。

## 编程 RK4

RK4是数值积分中的老牌明星。如果人们提及“Runge-Kutta”，几乎总是讨论RK4。和RK2比，我们要做差不多类似的事情，只是不再采用开始和结束端的信息，而是要用4处的信息。

在RK4中，求平均值的方式有点不同。先让我们看看伪码，很长，所以名词我都用了简称。

```

// pos1 是对象的当前位置
// vel1 是对象的当前速度
acc1 = acceleration(pos1, vel1)
```

```

pos2 = pos1 + vel1 / 2 * time
vel2 = vel1 + acc1 / 2 * time
acc2 = acceleration(pos2, vel2)

pos3 = pos1 + vel2 / 2 * time
vel3 = vel1 + acc2 / 2 * time
acc3 = acceleration(pos3, vel3)

pos4 = pos1 + vel3 * time
vel4 = vel1 + acc3 * time
acc4 = acceleration(pos4, vel4)

pos1 += (vel1 + vel2 * 2 + vel3 * 2 + vel4) / 6 * time
vel1 += (acc1 + acc2 * 2 + acc3 * 2 + acc4) / 6 * time

```

注意，第一、第四步的求解和第二、第三步不同，在第二、第三步时候先除以了 2，在最后求平均时又乘以了 2。这是考虑曲线末尾和中间有着特殊权重。也就是说，并不是平等的加 4 次以后除以 4，而是多在中间加两次，然后除以 6。我无法理解发生了什么，但是 Runge 和 Kutta 可以，他们比我聪明的多。我在这里想了很久，于是有了上面那段含糊不清的解释（译：我更迷糊 orz）。重要的是我知道如何把它运用于 AS 中。好了，深呼吸，我们继续：

```

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

    public class RK4 extends Sprite
    {
        private var _ball:Sprite;
        private var _position:Point;
        private var _velocity:Point;
        private var _gravity:Number = 32;
        private var _bounce:Number = -0.6;
        private var _oldTime:int;
        private var _pixelsPerFoot:Number = 10;

        public function RK4()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            _ball = new Sprite();
            _ball.graphics.beginFill(0xff0000);
            _ball.graphics.drawCircle(0, 0, 20);
            _ball.graphics.endFill();
            _ball.x = 50;
            _ball.y = 50;
            addChild(_ball);

            _velocity = new Point(10, 0);
        }
    }
}

```

```

_position = new Point(_ball.x / _pixelsPerFoot, _ball.y / _pixelsPerFoot);

_oldTime = getTimer();
addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    var time:int = getTimer();
    var elapsed:Number = (time - _oldTime) / 1000;
    _oldTime = time;

    var accel1:Point = acceleration(_position, _velocity);

    var position2:Point = new Point();
    position2.x = _position.x + _velocity.x / 2 * elapsed;
    position2.y = _position.y + _velocity.y / 2 * elapsed;

    var velocity2:Point = new Point();
    velocity2.x = _velocity.x + accel1.x / 2 * elapsed;
    velocity2.y = _velocity.y + accel1.y / 2 * elapsed;

    var accel2:Point = acceleration(position2, velocity2);

    var position3:Point = new Point();
    position3.x = _position.x + velocity2.x / 2 * elapsed;
    position3.y = _position.y + velocity2.y / 2 * elapsed;

    var velocity3:Point = new Point();
    velocity3.x = _velocity.x + accel2.x / 2 * elapsed;
    velocity3.y = _velocity.y + accel2.y / 2 * elapsed;

    var accel3:Point = acceleration(position3, velocity3);

    var position4:Point = new Point();
    position4.x = _position.x + velocity3.x * elapsed;
    position4.y = _position.y + velocity3.y * elapsed;

    var velocity4:Point = new Point();
    velocity4.x = _velocity.x + accel3.x * elapsed;
    velocity4.y = _velocity.y + accel3.y * elapsed;

    var accel4:Point = acceleration(position4, velocity4);

    _position.x += (_velocity.x + 2 * velocity2.x + 2 * velocity3.x + velocity4.x) / 6 * elapsed;
    _position.y += (_velocity.y + 2 * velocity2.y + 2 * velocity3.y + velocity4.y) / 6 * elapsed;

    _velocity.x += (accel1.x + 2 * accel2.x + 2 * accel3.x + accel4.x) / 6 * elapsed;
    _velocity.y += (accel1.y + 2 * accel2.y + 2 * accel3.y + accel4.y) / 6 * elapsed;

    if(_position.y > (stage.stageHeight - 20) / _pixelsPerFoot)
    {
        _position.y = (stage.stageHeight - 20) / _pixelsPerFoot;
    }
}

```

```

        _velocity.y *= _bounce;
    }
    if(_position.x > (stage.stageWidth - 20) / _pixelsPerFoot)
    {
        _position.x = (stage.stageWidth - 20) / _pixelsPerFoot;
        _velocity.x *= _bounce
    }
    else if(_position.x < 20 / _pixelsPerFoot)
    {
        _position.x = 20 / _pixelsPerFoot;
        _velocity.x *= _bounce;
    }

    _ball.x = _position.x * _pixelsPerFoot;
    _ball.y = _position.y * _pixelsPerFoot;
}

private function acceleration(p:Point, v:Point):Point
{
    return new Point(0, _gravity);
}
}
}

```

哇哦，好长的一段代码，如果上面那段伪码你看懂了，代码就不用看了。

再说一遍，如果你看出它与 RK2 或者欧拉有哪里不同，说明你眼力比我好。我不想在此展示更深的例子，因为能体现 RK4 所带来的超精确的模拟真是凤毛麟角，而且还有更深层的理由如下一节所说。

### 薄弱环节

至此，这些用于球的运动的代码，是利用数值积分而有了现实般的精确度。但作为一个精准运动模式，是完全没用的。因为里面的弹性机制全是假冒的，甚至比欧拉法还恶劣。回弹采用快速而简单的方法仅仅满足于肉眼看上去还行，其并没有准确遵循物理标准。同样的，上本书中提到的大多数运动代码，如摩擦，碰撞反应，巨型体重力，弹性等等。大多数这些虽然基于一定的物理公式，但几乎所有这些内容，为了促使代码简单和改善 CPU 都用了一定的技巧去简化它们。

如果为了精确而不惜采用 RK4，就不要让某一环节由于使用了低劣的模拟手段而前功尽弃。不幸的是，我无法为这点而重写上本书，所以得靠各位自己了。不行的话只能去翻大学物理书。

### 总结 Runge-Kutta

虽然我几次三番劝说 Runge-Kutta 不是必须的，但并不意味可以轻视和忽略它。对于精确模拟来说它是个有价值的工具，在编程领域它是物理代码的标准。比欧拉有着更高的精确度，意味着有着更高的稳定性，所以如果弹性老是不正常，那就可以试试 RK。我猜大多数 AS 开发人员不会将其视作每天的必需品，但我还是希望这一章除能在一些人有需要的时候给以帮助。

现在，进入本章的第二部分，我们将讨论另一种数值积分方式，我想你会发现它非常有用且有趣的：Verlet integration。

### Verlet 积分法

Verlet 积分法最初作为模拟分子运动来开发的。当许多粒子以一个固定关系相互影响时，它们之间会推来推去，另外一些受力作用也会改变其位置和速度。

在这种情况下，现有的积分方式会变的很复杂很不稳定，而 Verlet 积分法是一个更有效、稳定的方法。

在当今的软件领域里，Verlet 积分法经常用于创建布娃娃物理系统 (<http://zh.wikipedia.org/wiki/布娃娃系统>)。这一技术的广泛流传始于 2003 年，Thomas Jakobsen 笔下那篇名叫《Advanced Character Physics》的文章，同时也出现在 [www.gamesutra.com](http://www.gamesutra.com)

上（这是一个在游戏、物理及其它与编程领域相关，有着丰富资源的网站）。本章提到的代码主要基于那篇文章所描述的系统（代码当然是 ActionScript 3.0 的）。

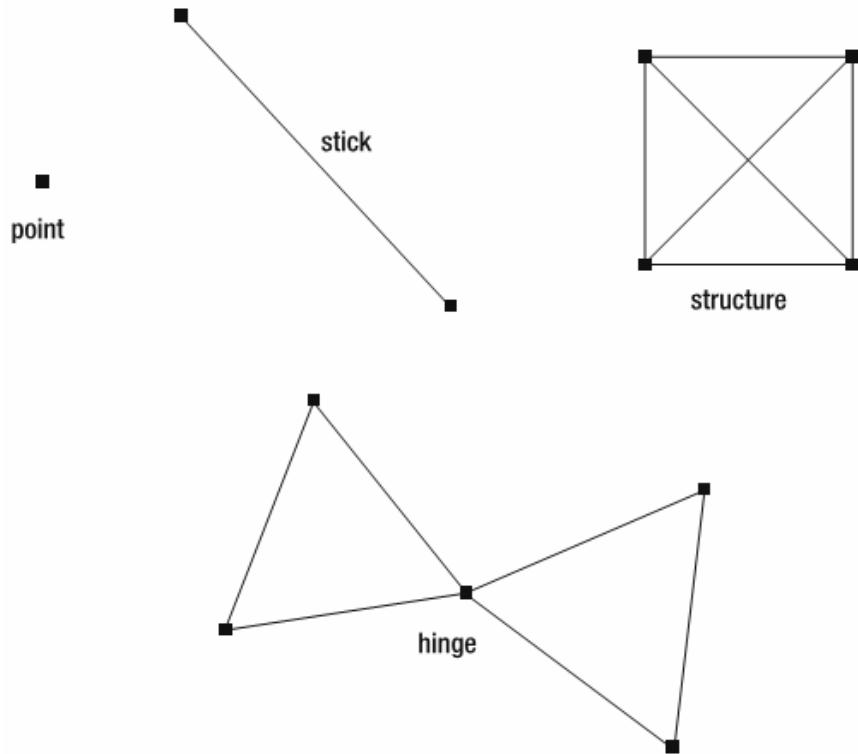
因为 Verlet 积分法的优势不像 Runge-Kutta 那样以超精确为主，我将再回到以帧驱动的方式来进行描述，以便使内容描述的更清晰（非要用时间驱动当然也没问题）。

Verlet 积分法最突出的一点是不用保存对象的速度，取而代之的是保存对象的位置。为了知道对象的速度，只需要当前位置减去之前位置即可。因此，对象每次的移动速度都可能是变化的。这就简化了很多事情。比如说一个对象在 x 等于 100 的地方，移动到了 110 的地方。在画面更新后，就会看到对象向右移动了 10 个像素，那么最近一次的 x 轴上的速度就是 10。在每次不断的更新时，对象之前 x 总是在当前 x 的左侧，所以对象就不断的向右移动。以上看出对象的速度是通过位置求出的。

你可以去想，这个过程对分子模拟是如何的有用。如果有一大堆粒子互相碰撞，它们在试着去保持彼此间一定的距离时，会互相吸引和排斥，而它们只是在改变位置，加速度与速度的关系是由 Verlet 积分法隐式处理的。

Verlet 积分法还有其它一些常用于软件开发的性质，比如对象束缚的概念。两个对象之间有着特定的距离。如果离的太近会隔的太远，就调整它们之间的距离，也就是改变它们的速度。一个对象可以和多个对象之间发生类似关系。Verlet 对处理这些关系有着很不错的效率。所以对建造复杂的像布娃娃系统这种运动结构时，它非常有用。

虽然我一直在用对象这个词，但并不是所有形状都适用。实际上只适合于点。距离就是两点间的直线长短。两点可以确定一条线段，一条或多条线段可以确定一个结构。需要注意，两条或多条线段可以共享同一个点，可以想象一下拉链。如图 6-4：



**Figure 6-4.** Points, sticks, structures, and hinges in Verlet integration

图 6-4

让我们从点的移动开始。

### Verlet 点

创建一个 VerletPoint 类，它封装了一个点所拥有的 Verlet 积分法的所有行为。是点，就需要 x 和 y 属性，还有 old x 和 old y 以及一个 update 函数。update 函数是告诉点之前在哪里，然后该去哪里，并使用什么样的速度。然后保存当前位置为之前位置，以便下次使用。基本逻辑如下：

```
temp = currentPosition
```

```
velocity = currentPosition - oldPosition
currentPosition += velocity
oldPosition = temp
```

因为当前位置会被改变，所以需要先把当前位置保存在一个临时变量里。

然后计算出速度，并加于当前位置下，最后把之前保存的当前位置作为之前位置保留下来。

代码 VerletPoint

```
package
{
    import flash.display.Graphics;
    import flash.geom.Rectangle;

    public class VerletPoint
    {
        public var x:Number;
        public var y:Number;

        private var _oldX:Number;
        private var _oldY:Number;

        public function VerletPoint(x:Number, y:Number)
        {
            setPosition(x, y);
        }

        public function update():void
        {
            var tempX:Number = x;
            var tempY:Number = y;
            x += vx;
            y += vy;
            _oldX = tempX;
            _oldY = tempY;
        }

        public function setPosition(x:Number, y:Number):void
        {
            this.x = _oldX = x;
            this.y = _oldY = y;
        }

        public function constrain(rect:Rectangle):void
        {
            x = Math.max(rect.left, Math.min(rect.right, x));
            y = Math.max(rect.top, Math.min(rect.bottom, y));
        }

        public function set vx(value:Number):void
        {
            _oldX = x - value;
        }

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

```

        return x - _oldX;
    }

    public function set vy(value:Number):void
    {
        _oldY = y - value;
    }
    public function get vy():Number
    {
        return y - _oldY;
    }

    public function render(g:Graphics):void
    {
        g.beginFill(0);
        g.drawCircle(x, y, 4);
        g.endFill();
    }
}

```

出乎意料的简单。你可能对为什么 vx 和 vy 采用 getter/setter 方式而感到迷惑，因为之前我说过 Verlet 积分法是不保存速度的。其实 getter/setter 方式并不保存任何信息。当设置 vx 时，\_oldX 等于当前 x 减去给定的值，这就好像在说速度的由来，而类似 getter 得到的也是位置相减的结果。所以并没有去保存一个叫速度的信息。

还有一个 setPosition 函数，它同时设置之前和当前的位置。这在仅仅移动一个点时很有用。因为之前和当前的位置都一样了，速度也就等于零了。

等下会提到 constrain 函数，在这之前我想先说下 render 函数。因为 VerletPoint 不是一个显示对象，所以无法直接在场景上显示。

render 函数会以给定的 Graphics 参数绘制一个小圆点。这对你最终的作品可能没什么用，但对 debug 或者测试来说很有用。

来看个例子：

```

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

    import flash.geom.Rectangle;

    public class VerletPointTest extends Sprite
    {
        private var _point:VerletPoint;

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

            _point = new VerletPoint(100, 100);
        }
    }
}

```

```

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {

        _point.update();
        graphics.clear();
        _point.render(graphics);
    }
}

```

在构造函数中，我们创建了一个 VerletPoint 实例，并监听 enterFrame 事件。在事件处理函数中，调用实例的 update 函数以及不断重绘。随便在哪里设置一下 vx、vy，比如构造函数中：

```
_point.vx = 5;
```

这会导致之前位置先往后移动 5 像素，看上去好像实例拥有一个向前的速度。同样可以用下面的方法来移动实例：

```
_point.x += 5;
```

大多数系统中，这个过程只是单单改变了点的位置，并没有影响到它的速度。但在 Verlet 积分法中，它同时改变了点的移动方向。你可以这么想象：并不是把点放置在距离 5 像素的地方，而是给点一个有着移动 5 像素大小的力——给定一个推力，之后就会沿着力的方向继续移动。

当引入重力后也是一样。可以通过增加垂直速度或者只增加垂直位置即可。使用下面两句的任意一句，就可以实现重力效果：

```
_point.vy += .5;
```

或

```
_point.y += .5;
```

第二行看上去只是定量的改变了位置，但记住位置的改变会引起速度的改变，所以速度也是在递增的。尽管通过 vx、vy 来改变速度概念上更清晰，但是用位置却有着更高的效率，因为这么做只需改变一个变量。如果有众多粒子进行交互，效率问题是首先要考虑的。

### 点的约束

你可能在想，最好能把点都控制在场景内。把点约束在一个区域内在 Verlet 积分法中非常简单。我们所要做的就是给定一个矩形范围。在测试程序中，我们创建了一个场景大小的矩形（你可以定义任意大小的矩形），然后在调用 update 之前，把它作为参数传给 constrain 函数。

```

public function constrain(rect:Rectangle):void
{
    x = Math.max(rect.left, Math.min(rect.right, x));
    y = Math.max(rect.top, Math.min(rect.bottom, y));
}

```

测试代码：

```

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

    public class VerletPointTest extends Sprite
    {

```

```

private var _point:VerletPoint;
private var _stageRect:Rectangle;

public function VerletPointTest()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    _stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

    _point = new VerletPoint(100, 100);
    _point.x += 5;
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    _point.y += .5;
    _point.constrain(_stageRect);
    _point.update();

    graphics.clear();
    _point.render(graphics);
}
}
}

```

我知道你准备说：没有反弹！对的，没有。你需要修改 constrain 增加反弹功能。虽然这会有点复杂，但在碰壁后效果会好很多。但要记住，一般来说，点真正的用途也就是定义线段和结构，显示它们是没什么意义的。所以对 VerletPoint 我们不准备增加额外的复杂度，我们该关心的是线段和结构在碰壁后的反应。

说线段线段就到。

### Verlet 线段

一个线段确定两个点。线段有 length 属性，它代表着两个点之间的距离大小。如果两点之间的距离和 length 有差异，就适当的贴近或者远离。

代码 VerletStick.as

```

package
{
    import flash.display.Graphics;

    public class VerletStick
    {
        private var _pointA:VerletPoint;
        private var _pointB:VerletPoint;
        private var _length:Number;

        public function VerletStick(pointA:VerletPoint, pointB:VerletPoint, length:Number = -1)
        {
            _pointA = pointA;
            _pointB = pointB;
            if(length == -1)
            {
                var dx:Number = _pointA.x - _pointB.x;
                var dy:Number = _pointA.y - _pointB.y;
            }
        }
    }
}

```

```

        _length = Math.sqrt(dx * dx + dy * dy);
    }
else
{
    _length = length;
}
}

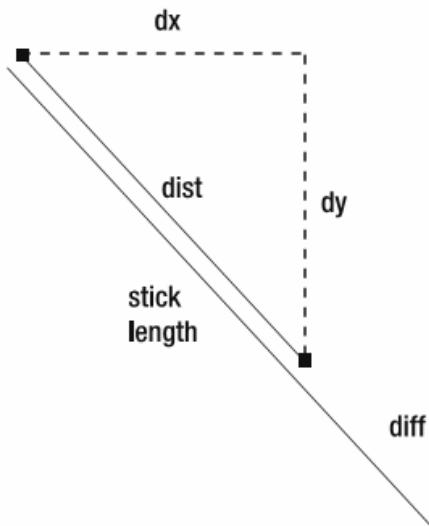
public function update():void
{
    var dx:Number = _pointB.x - _pointA.x;
    var dy:Number = _pointB.y - _pointA.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    var diff:Number = _length - dist;
    var offsetX:Number = (diff * dx / dist) / 2;
    var offsetY:Number = (diff * dy / dist) / 2;
    _pointA.x -= offsetX;
    _pointA.y -= offsetY;
    _pointB.x += offsetX;
    _pointB.y += offsetY;
}

public function render(g:Graphics):void
{
    g.lineStyle(0);
    g.moveTo(_pointA.x, _pointA.y);
    g.lineTo(_pointB.x, _pointB.y);
}
}
}

```

构造函数接受两个 VerletPoint 和一个可选的 length 作为参数，默认情况下通过计算两个点的位置来决定 length 的大小。和 VerletPoint 类似，它也有一个 render 函数，用来绘制一条线段，通常也是用来 debug 和测试的。主要是 update 函数，需要做些解释。

首先通过两点间距离公式算出线段当前的长度，再减去之前的长度得到差量。如图 6-5。



**Figure 6-5.** Calculating the distance between the two points and the difference between it and its ideal length

图 6-5

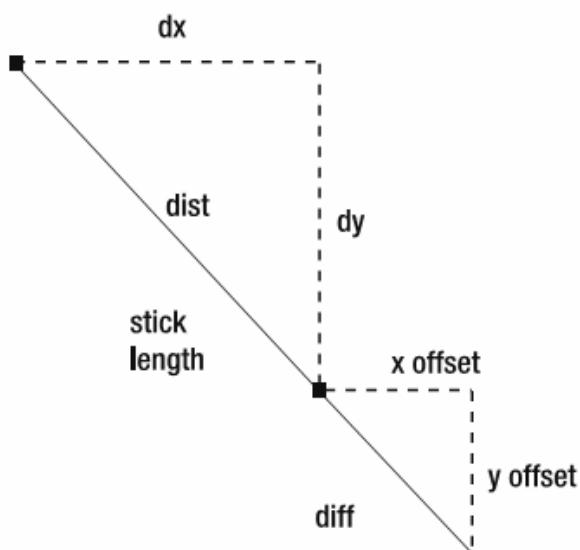
接下来可以通过三角函数求出差量在 x、y 上的分量。还记得 cos 等于临边/斜边，sin 等于对边/斜边吗，如下：

$$\begin{aligned} \text{diff} * \text{dx} / \text{dist} \\ \text{diff} * \text{dy} / \text{dist} \end{aligned}$$

如果用角度，那就是：

$$\begin{aligned} \text{diff} * \cos(\text{角度}) \\ \text{diff} * \sin(\text{角度}) \end{aligned}$$

但我们要避免使用三角函数（Math.atan2、Math.cos、Math.sin）来算取 x、y 的偏移量。以上三角函数所得结果如图 6-6

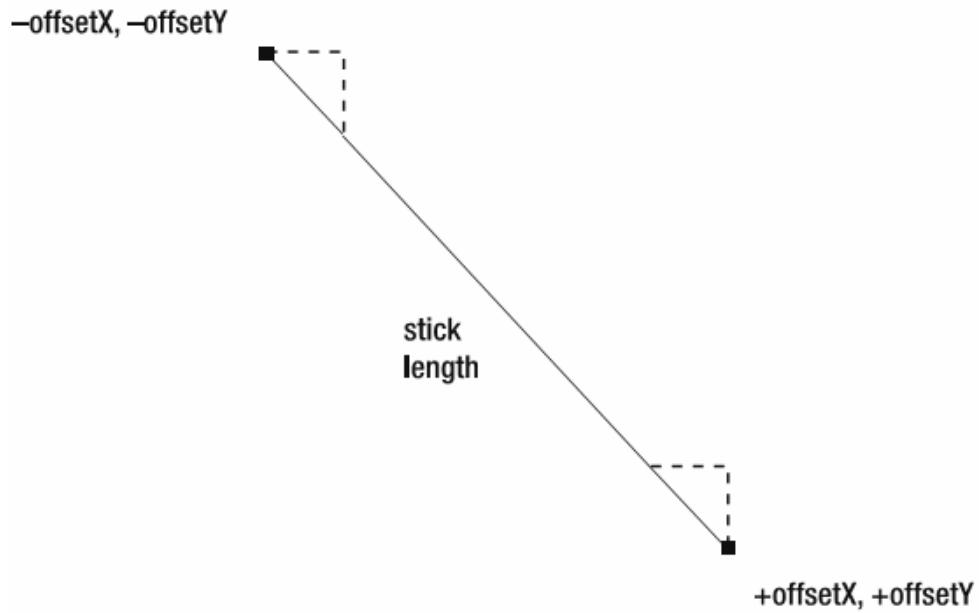


**Figure 6-6.** Calculating the x and y components of the difference

图 6-6

注意，实际代码中我们把结果除以 2，然后两个点各自加减结果的一半。也就是，第一个点减去 x、y 偏移量的一半，第二个点加上 x、y 偏移量的一半。这样做

让两个点都有反应，并保证之间的距离和 length 一致。如图 6-7 所示



**Figure 6-7.** Moving each point by half the difference on each axis makes the distance equal to the stick length.

图 6-7

可能还有一些别的更有效率的方法。不过这里只用到了一个 Math 函数（求距离），而且概念清晰，所以还算不错啦。

看段测试代码吧。

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

    import flash.geom.Rectangle;

    public class VerletStickTest extends Sprite
    {
        private var _pointA:VerletPoint;
        private var _pointB:VerletPoint;
        private var _stick:VerletStick;
        private var _stageRect:Rectangle;

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

            _stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

            _pointA = new VerletPoint(100, 100);
            _pointB = new VerletPoint(105, 200);

            _stick = new VerletStick(_pointA, _pointB);

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

```

    }

private function onEnterFrame(event:Event):void
{
    _pointA.y += .5;
    _pointA.update();
    _pointA.constrain(_stageRect);

    _pointB.y += .5;
    _pointB.update();
    _pointB.constrain(_stageRect);

    _stick.update();

    graphics.clear();
    _pointA.render(graphics);
    _pointB.render(graphics);
    _stick.render(graphics);
}

}
}
}

```

增加一条线段并没有做太多的事情。首先是确定点，由点确定线段。然后在 enterFrame 处理函数中增加对线段的更新和渲染。

当运行这段测试代码后，应该会看到一条线段垂直落于场景底部。但是，竟然发生了反弹！我们没有加反弹代码啊，这是怎么回事？原来，这是因为线段的 update 函数在试着推进两个点时，接触到矩形底部的点又被强制拉了回来。同时作用的两个反作用力，导致两个点稍稍向上弹起以适应所有的约束条件。

实际上，这个弹性更像是一种副作用。我还蛮喜欢的，不过是副作用总会有点捉摸不定的问题。比如要去掉弹性时，我们就需要多调用几次点的

constrain 函数和线段的 update 函数，让它们尽快适应条件。修改 onEnterFrame 函数如下：

```

private function onEnterFrame(event:Event):void
{

```

```

    _pointA.y += .5;
    _pointA.update();

    _pointB.y += .5;
    _pointB.update();

    for(var i:int = 0; i < 5; i++)
    {
        _pointA.constrain(_stageRect);
        _pointB.constrain(_stageRect);
        _stick.update();
    }

    graphics.clear();
    _pointA.render(graphics);
    _pointB.render(graphics);
    _stick.render(graphics);

```

```
}
```

通过几次循环能让点和线段在到达某一点处时步伐一致。这是因为重力的影响没有加入在循环中，唯一影响速度的只有每个点在循环前后的差异。这就好像一条钢管和一根树枝。如果改变循环次数，就能产生不同性质的效果，循环次数越多，线段就越具有刚性。

不管怎样，要认识到循环内执行的代码其实是很多的，特别是线段的 update 函数。在这里演示的效果似乎体现不出什么，但如果有一个巨型结构体，多次这样的循环就会很费 CPU。我的意见是非到迫不得已时不要去循环。

### Verlet 结构体

一个 Verlet 结构体由多条线段组成。最简单的实心结构体是三角形。我们这就来做一个：

代码 Triangle.as

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

    public class Triangle extends Sprite
    {
        private var _pointA:VerletPoint;
        private var _pointB:VerletPoint;
        private var _pointC:VerletPoint;
        private var _stickA:VerletStick;
        private var _stickB:VerletStick;
        private var _stickC:VerletStick;
        private var _stageRect:Rectangle;

        public function Triangle()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

            _pointA = new VerletPoint(100, 100);
            _pointB = new VerletPoint(200, 100);
            _pointC = new VerletPoint(150, 200);

            _stickA = new VerletStick(_pointA, _pointB);
            _stickB = new VerletStick(_pointB, _pointC);
            _stickC = new VerletStick(_pointC, _pointA);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            _pointA.y += .5;
            _pointA.update();

            _pointB.y += .5;
            _pointB.update();
        }
    }
}
```

```

    _pointC.y += .5;
    _pointC.update();

    for(var i:int = 0; i < 1; i++)
    {
        _pointA.constrain(_stageRect);
        _pointB.constrain(_stageRect);
        _pointC.constrain(_stageRect);
        _stickA.update();
        _stickB.update();
        _stickC.update();
    }

    graphics.clear();
    _pointA.render(graphics);
    _pointB.render(graphics);
    _pointC.render(graphics);
    _stickA.render(graphics);
    _stickB.render(graphics);
    _stickC.render(graphics);
}
}
}

```

先确定三个点：A、B、C，再连接这三个点确定三条边，组成一个三角形。运行一下会看到，一个三角形下落到场景底部，略微反弹几下后，就横躺着了。试着加大循环次数，看看什么叫刚性结构体。

接下来我们试试正方形。

```

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

    import flash.geom.Rectangle;

    public class Square extends Sprite
    {
        private var _pointA:VerletPoint;
        private var _pointB:VerletPoint;
        private var _pointC:VerletPoint;
        private var _pointD:VerletPoint;
        private var _stickA:VerletStick;
        private var _stickB:VerletStick;
        private var _stickC:VerletStick;
        private var _stickD:VerletStick;
        private var _stageRect:Rectangle;

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

```

```

_stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

_pointA = new VerletPoint(100, 100);
_pointB = new VerletPoint(200, 100);
_pointC = new VerletPoint(200, 200);
_pointD = new VerletPoint(100, 200);

_stickA = new VerletStick(_pointA, _pointB);
_stickB = new VerletStick(_pointB, _pointC);
_stickC = new VerletStick(_pointC, _pointD);
_stickD = new VerletStick(_pointD, _pointA);

addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{

    _pointA.y += .5;
    _pointA.update();

    _pointB.y += .5;
    _pointB.update();

    _pointC.y += .5;
    _pointC.update();

    _pointD.y += .5;
    _pointD.update();

    for(var i:int = 0; i < 1; i++)
    {
        _pointA.constrain(_stageRect);
        _pointB.constrain(_stageRect);
        _pointC.constrain(_stageRect);
        _pointD.constrain(_stageRect);
        _stickA.update();
        _stickB.update();
        _stickC.update();
        _stickD.update();
    }

    graphics.clear();
    _pointA.render(graphics);
    _pointB.render(graphics);
    _pointC.render(graphics);
    _pointD.render(graphics);
    _stickA.render(graphics);
    _stickB.render(graphics);
    _stickC.render(graphics);
    _stickD.render(graphics);
}

```

```
    }
}
}
```

四个点连接四条线，没问题吧？第一眼看去没什么问题，但是当正方形落地后就塌了。这效果似乎还挺酷，不过却是个问题啊。让我们先把这个箱子加固一下，不要让它散架咯。通过增加一条对角线就可以轻松实现。定义一条线段经过点 A、C：

```
_stickE = new VerletStick(_pointA, _pointC);
```

要是还想加强牢固度，可以再连接点 B、D。

好像自由落体运动太无聊了，我们增加点噱头。只要推动一个点即可：

```
_pointA = new VerletPoint(100, 100);
```

```
_pointA.vx = 10;
```

因为作用力在一个角上，所以整个正方形会产生旋转。

再继续研究前，我们应该先清理一下代码。尽管只有四个点、五条线，就已经显得有点乱了。好在创建完的点和线所做的事情都一样：增加重力影响，更新位置，约束范围，最终渲染。

用两个数组分别来保存点和线就简单多了。

代码 Square.as

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

    public class Square extends Sprite
    {
        private var _points:Array;
        private var _sticks:Array;
        private var _stageRect:Rectangle;

        public function Square()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            _stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

            _points = new Array();
            _sticks = new Array();

            var pointA:VerletPoint = makePoint(100, 100);
            pointA.vx = 10;
            var pointB:VerletPoint = makePoint(200, 100);
            var pointC:VerletPoint = makePoint(200, 200);
            var pointD:VerletPoint = makePoint(100, 200);

            makeStick(pointA, pointB);
            makeStick(pointB, pointC);
            makeStick(pointC, pointD);
            makeStick(pointD, pointA);
            makeStick(pointA, pointC);

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

```

private function onEnterFrame(event:Event):void
{
    updatePoints();

    for(var i:int = 0; i < 1; i++)
    {
        constrainPoints();
        updateSticks();
    }

    graphics.clear();
    renderPoints();
    renderSticks();
}

private function makePoint(xpos:Number, ypos:Number):VerletPoint
{
    var point:VerletPoint = new VerletPoint(xpos, ypos);
    _points.push(point);
    return point;
}

private function makeStick(pointA:VerletPoint, pointB:VerletPoint, length:Number = -1):VerletStick
{
    var stick:VerletStick = new VerletStick(pointA, pointB, length);
    _sticks.push(stick);
    return stick;
}

private function updatePoints():void
{
    for(var i:int = 0; i < _points.length; i++)
    {
        var point:VerletPoint = _points[i] as VerletPoint;
        point.y += .5;
        point.update();
    }
}

private function constrainPoints():void
{
    for(var i:int = 0; i < _points.length; i++)
    {
        var point:VerletPoint = _points[i] as VerletPoint;
        point.constrain(_stageRect);
    }
}

private function updateSticks():void
{
    for(var i:int = 0; i < _sticks.length; i++)

```

```

    {
        var stick:VerletStick = _sticks[i] as VerletStick;
        stick.update();
    }
}

private function renderPoints():void
{
    for(var i:int = 0; i < _points.length; i++)
    {
        var point:VerletPoint = _points[i] as VerletPoint;
        point.render(graphics);
    }
}

private function renderSticks():void
{
    for(var i:int = 0; i < _sticks.length; i++)
    {
        var stick:VerletStick = _sticks[i] as VerletStick;
        stick.render(graphics);
    }
}
}

```

我们用了很多袖珍函数，这样感觉清爽多了吧。现在即使再多增加一些点和线段，都不会有太大的改动了。

### 拉链式结构

现在我们已经有了很不错的代码结构，接下来就可以搞一些疯狂的测试。比如拉链式结构，这种结构由两种结构共享与一点所组成。它们可以自由移动，但都围绕在同一枢轴处。

这里将介绍如何制作一个单摆器。除了构造函数，其它部分和之前都一样。

```

public function Hinge()

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

    _stageRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);

    _points = new Array();
    _sticks = new Array();

    // 根基
    var pointA:VerletPoint = makePoint(stage.stageWidth / 2, stage.stageHeight - 500);
    var pointB:VerletPoint = makePoint(0, stage.stageHeight);
    var pointC:VerletPoint = makePoint(stage.stageWidth, stage.stageHeight);

    // 臂摆
    var pointD:VerletPoint = makePoint(stage.stageWidth / 2 + 350, stage.stageHeight - 500);

    // 秤砣

```

```

var pointE:VerletPoint = makePoint(stage.stageWidth / 2 + 360,stage.stageHeight - 510);
var pointF:VerletPoint = makePoint(stage.stageWidth / 2 + 360,stage.stageHeight - 490);
var pointG:VerletPoint = makePoint(stage.stageWidth / 2 + 370,stage.stageHeight - 500);

// 根基
makeStick(pointA, pointB);
makeStick(pointB, pointC);
makeStick(pointC, pointA);

// 臂摆
makeStick(pointA, pointD);

// 秤砣
makeStick(pointD, pointE);
makeStick(pointD, pointF);
makeStick(pointE, pointF);
makeStick(pointE, pointG);
makeStick(pointF, pointG);

addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

```

这段代码创建了一个三角形作为根基，臂摆固定于三角形顶部，臂摆的另一端栓了一个秤砣。因为臂摆和根基共享一点，所以臂摆会摇荡。虽然秤砣也和臂摆产生摇荡，但由于大小因素，其效果并不明显。

## 深入研究

还有很多值得你去做的，比如把 update、constrain 以及 render 函数封装成一个独立的类——可以叫 PointManager 和 StickManager 什么的。还可以做一个结构编辑器，听上去貌似很有趣吧。其实我就做了一些编辑器，这真的很有趣。

你可能想用一些花边图形代替单调的线段。这样需要把图形的左侧中心点(0, height/2)和线段的一点对齐，再缩放图形与线段的长度一样大小，并旋转图形和线段的角度相同。自定义结构样式可是一个很大的挑战哦，但有信心事必成。

到此我们将告一段落，我不准备讲的面面俱到，还有很多有趣的地方就留给各位慢慢挖掘了。我能感觉到，当 Verlet 积分法被大家所熟悉后，网上必定会出现很多很牛的玩意儿。

## 总结

这一章讲述了什么是数值积分，以及使用时会碰到的问题。知道了一直在用的叫欧拉积分法，还有一些其它可用的方法。最重要的是，对它们的优缺点都有了一定了解并能合理的去使用。

数值积分法不只是欧拉、Runge-Kutta 和 Verlet。查一下 Wikipedia 就能看到一些列很有趣的名字：欧拉后向法(BackwardEuler)、半隐式欧拉法(Semi-implicit Euler), Verlet 速度(Velocity Verlet)、毕曼算法(Beeman's algorithm)，霍因法(Heun's method)、纽马克法(Newmark-beta method)，跳跃积分法(Leapfrog integration)和蒙特卡洛积分法(Monte Carlo integration)。这些我并不都了解，但去发掘一下势必很有趣。就介绍给各位啦。

下一章，我们将研究 Flash 10 的标志性新特性——3D！

## 第七章 3D IN FLASH 10

自从我接触 Flash 之后，大约每 18 个月就可以看到一个新版本 Flash 的制作软件连同一个新 Flash 播放器一起出来。在每个版本被释放之后，大家就推测和希望哪些功能将会在下一个版本中列出。在这里，盼望已久功能之一是 Flash 的内建 3D。但是当以前版本更新的时候而没有出现这个功能，大家都感到很失望。直到 Flash 10。

也许之前你总是用缩放和位置变化来近似的模拟景深，用 ActionScript 语言来写真正的 3D 引擎。我在《Making Things Move》这本书里已经涵盖了这方面的基本东西，在最近几年里，出现了很多的很不错的 Flash 3D 引擎。像 PaperVision 3D，Away3D，和其他很多引擎都能从专业建模软件里导入 3D 模型和材质，并且在 Runtime 里渲染和控制这些模型。但是这些程序已经由社区成员写成类，被编译并且作为一个内部成分运行在 Flash 10 里，我们第一次去新建一个 DisplayObject—sprite，MovieClip，TextField 等等，然后在 3D 的空间里操纵它。你能用 Flash 自身的工具去做它也可以用 ActionScript 语言操作。

这章节将会把重心集中在 ActionScript 部份。虽然这个在很多功能上不比 PaperVision 3D 强—像导入模型和材质，自动深度排序等等，但是可以很容易完成很多超出你想象的 Flash 3D 效果。

好吧，我们正在等什么？让我们开始 3D。

### Flash 3D 基础

这是一本高阶书，因此我假定你至少知道 3D 一些小知识。有三个维度：x 是水平，y 是垂直，和 z 是深度。在 Flash 中，起始或零的点，是荧屏的左上角，至少在 2D 中。

如果你以前用过正常的笛卡儿坐标，在这里 Y 轴可能颠倒，但是你将会习惯它。在 Flash 10 3D，当一个物件的 z 值变得比较大时就会感觉“进入荧屏之内”否则就是远离视角。换句话说，z 值低的物体将会在 z 值低高的物体之前出现。

见图 7-1。

还有一个值提到的就是如何在 Flash 10 3D 的各轴上实现旋转。绕 Z 轴旋转，如果视角在正前方，角度增加时，将沿着顺时针转动。这和你习惯的再一次相反。同样的，绕 Y 轴旋转视角从在上面看也将顺时针转动。绕 X 轴旋转从物体的左边看也将顺时针转动。（见图 7-2）

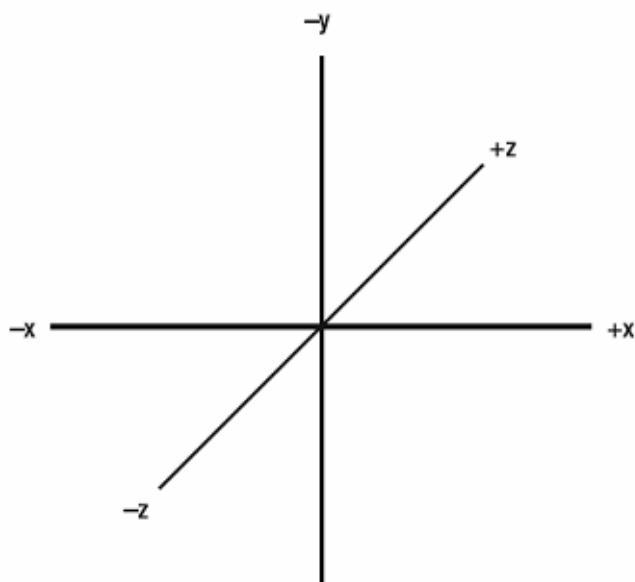
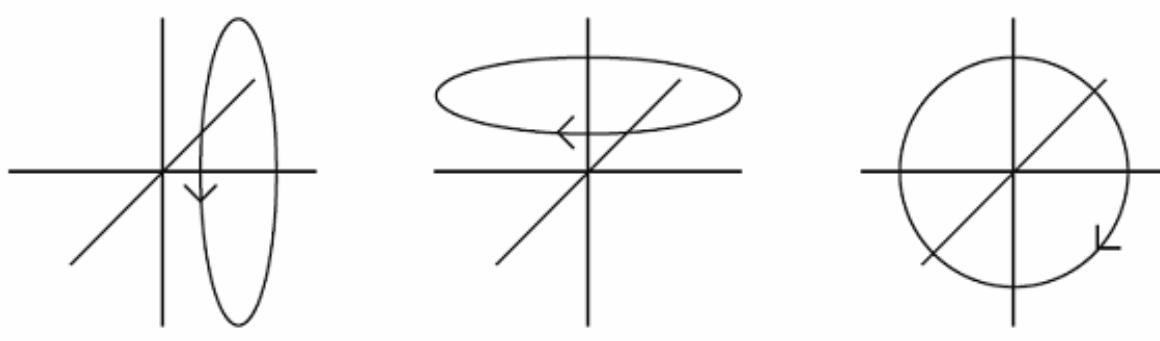


图 7-1 Flash 10 3D 坐标



绕 X 轴旋转

绕 Y 轴旋转

绕 Z 轴旋转

图 7-2 Flash 10 3D 旋转

了解的另外一件重要的事就是作为 Flash 10 3D 旋转的角度是度数而不是弧度。如果你已经搞过 3D 编程，那些三角函数都是用弧度来运算，那么这些看起来也许有些奇怪。但是 Flash 里的

3D 功能是创作工具的一部分，是面向设计者而不是工程师。他们想要估计某 45 度角而不是  $\pi/4$  弧度去转动。这样的话你需要一个该如何在弧度和角度之间转换的公式，可以看这里：

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

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

好了，现在我们来了解术语，让我们来看看在 Flash 10 有哪些 3D 编程（API）中的术语。尽管有很多名词，但真正贯穿整个新的 3D API 是四个新的属性：z, rotationX, rotationY 和 rotationZ。我们的学习还将涵盖剩下的，但是这四个属性将是建立基本的 3D 效果里最常用的。让我们来试试。创建下面的类并且编译它。本章节以及这本书其它的章节所有的文件，你都可以在 [www.friendsofed.com](http://www.friendsofed.com) 下载到。

这个文件的名字是 Test3D.as:

```
package {  
    import flash.display.Shape;  
    import flash.display.Sprite;  
    import flash.display.StageAlign;  
    import flash.display.StageScaleMode;  
    import flash.events.Event;  
    import flash.geom.PerspectiveProjection;  
    import flash.geom.Point;  
  
    public class Test3D extends Sprite {  
        private var _shape:Shape;  
  
        public function Test3D () {  
            stage.align = StageAlign.TOP_LEFT;  
            stage.scaleMode = StageScaleMode.NO_SCALE;  
  
            _shape = new Shape();  
            _shape.graphics.beginFill(0xff0000);  
            _shape.graphics.drawRect(-100, -100, 200, 200);  
            _shape.x = stage.stageWidth / 2;  
            _shape.y = stage.stageHeight / 2;  
            addChild(_shape);  
  
            addEventListener(Event.ENTER_FRAME, onEnterFrame);  
        }  
  
        private function onEnterFrame(event:Event):void {  
            _shape.rotationY += 2;  
        }  
    }  
}
```

很惊奇！Flash 出现 3D 效果！

## 设置消失点

也许你很想马上开始做很酷的 3D 效果，但是请先认真读一下这个章节——这是很关键的地方。即使你明白了 3D，这里有几个地方很诡异，诡异得让你发狂，除非你明白其中的原理。一旦你理解了它，它将变得很有意思，否则你将被 Flash 玩疯。

如果你将上面例子的类引入 FLA 并用 Flash CS4 编译，你应该会看到如图 7-3。

但是，如果你用 Flex Builder 编译或是其它用 Flex 4 SDK 编译的，你很可能就看到如图 7-4 的样子。



图 7-3 用 Flash CS4 编译的旋转平面

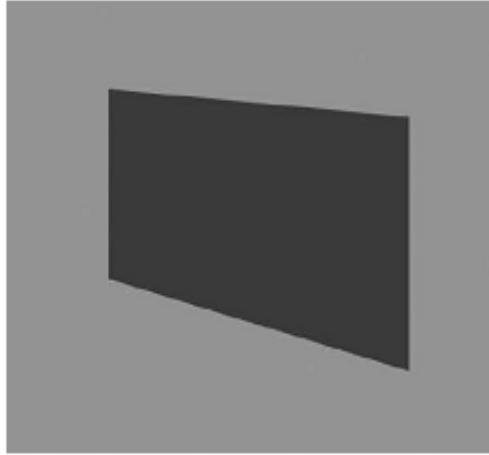


图 7-4 用 Flex Builder 编译的旋转平面

正如所见它从左上角伸展开。之所以在不同的编译下出现不同伸展效果，不在于画面是如何显示的，而是因为他们发布的方法和 Flash 设定消失点不同。在 3D 中，消失点是所有物体远离时的聚集点。如果我用一张铁路轨道的照片做向着地平线延伸的效果，你就会明白。当你设置 3D 环境时，你必须手动设置一个消失点而且要让所有的物件向它聚合。人们通常选择舞台的中心为一个消失点。

Flash 10 3D 自动地为你设定消失点设定为舞台的中心。但是它只会在 SWF 加载的时候运行一次。当你测试 Flash 影片的时候，消失点将被放在文件属性面板的基本窗口里。所以消失点在中心，效果也很好。

但是，在 Flex Builder 里，默认 SWF 的大小为 500\*375 像素。因此，你要将消失点设置在 (250, 187.5)。但是当你设置舞台靠左上角对齐，元素无缩放 (Stage.align=StageAlign.TOP\_LEFT, Stage.scaleMode=StageScaleMode.NO\_SCALE)，它将会改变舞台的大小。(一种普遍的做法都是这样设置，让 SWF 里有所有元素不随着它的比例变化而改变，只是缩放舞台的大小)。你可以用 trace 来输出设置舞台对齐方式和缩放模式前和设置后舞台的宽和高：

```
trace(stage.stageWidth, stage.stageHeight); // 500,375  
stage.align = StageAlign.TOP_LEFT;  
stage.scaleMode = StageScaleMode.NO_SCALE;  
trace(stage.stageWidth, stage.stageHeight); // 1440, 794
```

最后一个输出你将得到不同的值，这个只依赖于你窗口的大小。所以，舞台的大小变了，sprite 也被重新设置在舞台的新中心点，比如说这个就是 (720, 397)。但是消失点还是在点 (250, 187.5)。这是一个问题。

设置舞台的宽和高最简单的方法就是直接在你 SWF 的元数据里设置：

```
[SWF(backgroundColor=0xffffffff, width=800, height=800)]
```

这个将会在消失点之前运行，所以它会按(800, 800)的中心点去计算。当设置了对齐方式和缩放模式，舞台的大小将不会改变，同样你的 sprite 也会被固定在同一个中心。

但是，也许你想要一个可变动的舞台，让舞台自动去适应窗口的大小。这会有一点复杂，不过还好。你需要新建一个叫 PerspectiveProjection 的类。这个类来控制 3D 视角的渲染面，包含消失点。在 Flash 10 里，每个显示元素可以有一个 perspectiveProjection 去指定控制怎么去渲染 3D。它指定的是 transform 的 perspectiveProjection 属性。所以，假如有一个名叫 s 的 sprite，你可以这样去访问：

```
s.transform.perspectiveProjection
```

PerspectiveProjection 类有一个 projectionCenter 属性是 Point 类的实例。这就是通常所说的消失点。因此，去设置显示元素的中心就可以像下面这样：

```
s.transform.perspectiveProjection.projectionCenter=new Point(stage.stageWidth / 2,  
stage.stageHeight / 2);
```

这样只能设置一个 object 的消失点。当然这样就可以为这个元件的所有子元素设置消失点。  
如果你想为影片中的所有元素设置消失点，可以在 root 上设置，像这样：

```
root.transform.perspectiveProjection.projectionCenter = New Point(stage.stageWidth / 2,  
stage.stageHeight / 2);
```

下面这个类就按照上面正确的设置了舞台的对齐方式和缩放模式，解决了我们之前看到的问题：

```
package {  
    import flash.display.Shape;  
    import flash.display.Sprite;  
    import flash.display.StageAlign;  
    import flash.display.StageScaleMode;  
    import flash.events.Event;  
    import flash.geom.PerspectiveProjection;  
    import flash.geom.Point;  
  
    public class Test3D extends Sprite{  
        private var _shape:Shape;  
  
        public function Test3D() {  
            stage.align = StageAlign.TOP_LEFT;  
            stage.scaleMode = StageScaleMode.NO_SCALE;  
            root.transform.perspectiveProjection.projectionCenter = new  
                Point(stage.stageWidth / 2, stage.stageHeight / 2);  
  
            _shape = new Shape();  
            _shape.graphics.beginFill(0xff0000);  
            _shape.graphics.drawRect(-100, -100, 200, 200);  
            _shape.x = stage.stageWidth / 2;  
            _shape.y = stage.stageHeight / 2;  
            addChild(_shape);  
  
            addEventListener(Event.ENTER_FRAME, onEnterFrame);  
        }  
        private function onEnterFrame(event:Event):void {  
            _shape.rotationY += 2;  
            _shape.x = stage.stageWidth / 2;  
            _shape.y = stage.stageHeight / 2;  
        }  
    }  
}
```

你可以更进一步，像下面一样，无论舞台的大小怎么变化如果都可以正确改变中心点

```
package {  
    import flash.display.Shape;  
    import flash.display.Sprite;  
    import flash.display.StageAlign;  
    import flash.display.StageScaleMode;  
    import flash.events.Event;  
    import flash.geom.PerspectiveProjection;  
    import flash.geom.Point;  
  
    public class Test3D extends Sprite {  
        private var _shape:Shape;
```

```

public function Test3D()      {
    stage.addEventListener(Event.RESIZE, onResize)
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    _shape = new Shape();
    _shape.graphics.beginFill(0xff0000);
    _shape.graphics.drawRect(-100, -100, 200, 200);
    _shape.x = stage.stageWidth / 2;
    _shape.y = stage.stageHeight / 2;
    addChild(_shape);

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onResize(event:Event):void      {
    root.transform.perspectiveProjection.projectionCenter = new Point(stage.stageWidth / 2,
stage.stageHeight / 2);
    if(_shape != null)      {
        _shape.x = stage.stageWidth / 2;
        _shape.y = stage.stageHeight / 2;
    }
}

private function onEnterFrame(event:Event):void      {
    _shape.rotationY += 2;
    _shape.x = stage.stageWidth / 2;
    _shape.y = stage.stageHeight / 2;
}
}
}

```

因为我们在第一个动作里就监听了 `resize` 事件，所以当我们一旦改变舞台的设置或是舞台的大小，`onResize` 事件就会被触发，立即改变投影的中心。当用户改变窗口大小的时候也会触发这个事件。同时当舞台的中心改点的时候 `shape` 也会被重新定位。但是 `onResize` 事件最先被调用，那时 `shape` 还没有被创建，所以我们必须确保 `shape` 是已经存在的。因此，这里我们就加入了一个对 `shape` 的判断。

为了简单起见，在以后的章节里，我首先会将舞台的大小的设置统一放到元数据里。好了，消失点作为一个让你的影片看起来很有视觉效果的重要因素，你已经很清楚了。让我们来看看 3D 里的其它东西吧。

### 3D 坐标

我们假设你已经知道怎么去改变一个元素的 X, Y 坐标。其实改变 Z 坐标也是一样的。下面的这个类（文件 `Position3D.as`）建立了一个 `shape`，这个 `shape` 重复着以正弦速度离开和返回的动作。他的 x 和 y 也追随着鼠标：

```

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

    [SWF(width=800, height=800, backgroundColor=0xffffffff)]
    public class Position3D extends Sprite  {
        private var _shape:Shape;
        private var _n:Number = 0;

```

```

public function Position3D()      {
    _shape = new Shape();
    _shape.graphics.beginFill(0x00ff00);
    _shape.graphics.drawRect(-100, -100, 200, 200);
    _shape.graphics.endFill();
    addChild(_shape);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void {
    _shape.x = mouseX;
    _shape.y = mouseY;
    _shape.z = 10000 + Math.sin(_n += .1) * 10000;
}
}
}

```

从这个例子中你能学到的最重要的事情我认为是一旦你改变了 Z 坐标，显示元素的 X 和 Y 坐标就不再直接取决于屏幕坐标系了，他们取决于 3D 空间坐标系。如果你不移动你的鼠标，这个 shape 的 X 和 Y 坐标不改变，但是他们的 X 和 Y 坐标轴在屏幕上改变。只有当 Z 坐标为零时，元素的 X 和 Y 坐标才和屏幕 X 和 Y 坐标相等。这是因为当 Z 小于零时，Flash 把物体放大：当 Z 大于零时，Flash 把它缩小，但当 Z 坐标刚好等于零时，缩放比例刚好是 100%。

## 景深排序

当你开始制作很多物体并且在 3D 空间中设置它们的位置时，你将会遇到一个问题那就是很远的物体（Z 坐标值很大）有时出现在比它近的物体的前面。我猜想你肯定想知道有那些属性和方法可以运用，但很不幸的是没有现成的属性和方法你可以设定和调用来确定那些物体正确的顺序。

Flash 10 的 3D API 能够操作独立对象的缩放和扭曲—还有这些对象的子对象，如果有的话但却不能影响他们显示在屏幕上的顺序。这里用的仍然是 Flash 9 里显示 2D 对象的方法：在同一个容器中任何用 addChild 方法添加到显示列表中的对象会显示在以前的对象的前面。唯一能够改变这种情况的途径是一些操作显示列表的方法，比如 addChild, addChildAt, swapChildren, removeChild 等等。因为显示容器里没有排序方法，所以所有的景深排序工作都需要手动完成。

为了知道如何解决这个问题，我们先做一个演示这个问题的例子。如果一片森林全是树是什么情形呢？

```

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

    [SWF(width=800, height=800, backgroundColor = 0xccffff)]
    public class DepthSort extends Sprite {
        public function DepthSort() {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            for(var i:int = 0; i < 500; i++) {
                var tree:Shape = new Shape();
                tree.graphics.beginFill(Math.random() * 255 << 8);
                tree.graphics.lineTo(-10, 0);
                tree.graphics.lineTo(-10, -30);
            }
        }
    }
}

```

```

        tree.graphics.lineTo(-40, -30);
        tree.graphics.lineTo(0, -100);
        tree.graphics.lineTo(40, -30);
        tree.graphics.lineTo(10, -30);
        tree.graphics.lineTo(10, 0);
        tree.graphics.lineTo(0, 0);
        tree.graphics.endFill();
        tree.x = Math.random() * stage.stageWidth;
        tree.y = stage.stageHeight - 100;
        tree.z = Math.random() * 10000;
        addChild(tree);
    }
}
}
}

```

这里我们制作了一大堆 shapes 并且每个都用绘图 API 绘制了带阴影的树形物体。每个都随机地赋予 X, Y 和 Z 值。对于 Flash 设计来说它并不好看，但它达到了我们的目的。正如图 7-5 所显示的那样，它看起来好像不怎么对劲。

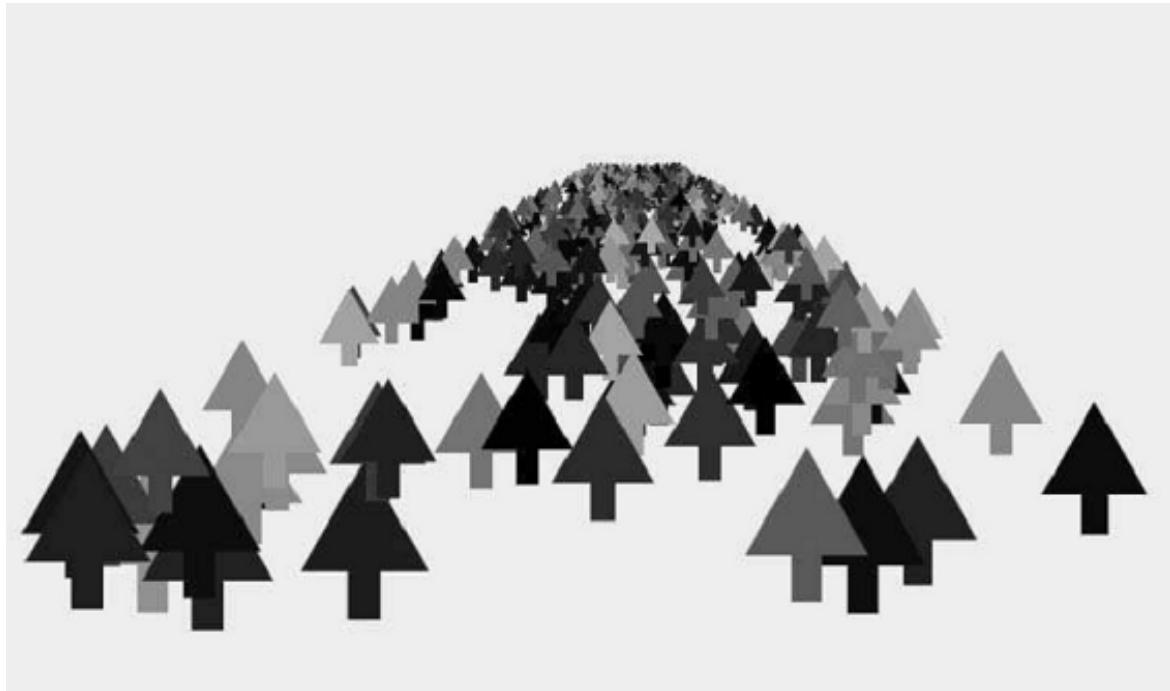


图 7-5 景物很好，但没有景深排序

我们仍然不能对显示列表排序，但我们可以对数组排序。所以我们把树放进一个数组里而不是在它被创建时把它放在显示列表里。然后我们可以对数组进行排序然后把具有正确顺序的树加进显示列表里-先是 Z 值大的，再是 Z 值小的。

```

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

    [SWF(width=800, height=800, backgroundColor = 0xccffcc)]
    public class DepthSort extends Sprite {
        private var _trees:Array;

        public function DepthSort() {

```

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

_trees = new Array();

for(var i:int = 0; i < 500; i++) {
    var tree:Shape = new Shape();
    tree.graphics.beginFill(Math.random() * 255 << 8);
    tree.graphics.lineTo(-10, 0);
    tree.graphics.lineTo(-10, -30);
    tree.graphics.lineTo(-40, -30);
    tree.graphics.lineTo(0, -100);
    tree.graphics.lineTo(40, -30);
    tree.graphics.lineTo(10, -30);
    tree.graphics.lineTo(10, 0);
    tree.graphics.lineTo(0, 0);
    tree.graphics.endFill();
    tree.x = Math.random() * stage.stageWidth;
    tree.y = stage.stageHeight - 100;
    tree.z = Math.random() * 10000;
    _trees.push(tree);
}

_trees.sortOn("z", Array.NUMERIC | Array.DESCENDING);
for(i = 0; i < 500; i++){
    addChild(_trees[i] as Shape);
}
}
```

尽管这么处理并不能使图 7-6 更具艺术感，但至少和你看到的真正的森林一样，远的树显示在近的树的后面。



图 7-6 景深排序后的森林

## 3D 容器

我第一次开始和 API 打交道的时候，在 Flash 10 的 3D 里，一个让我非常高兴的东西是我意识到：显示对象的容器会在他们被改变的时候改变他们的子对象。换句话说，当你往一个 sprite 里添加一些显示对象并且移动它的容器的时候，它并不是简单地把窗口展平然后作为一个独立的对象在 3D 空间里移动。它实际上改变每个子对象以使它们看起来是在 3D 空间里独立运动。

这是一个比较容易展示而不易描述的问题。所以接下来的这个类就会展示它。为什么我们这次不用一些其它的形状来绘制，比如正方形？Text fields 也是能用完全同样的方法在 3D 空间里移动的对象。我们将新建一个 sprite，添加一系列内容是随机字母的 Text field，然后移动这个 sprite。你可以在 Container3D.as 文件中发现下面的这个例子。

```
package {  
    import flash.display.Sprite;  
    import flash.display.StageAlign;  
    import flash.display.StageScaleMode;  
    import flash.events.Event;  
    import flash.text.TextField;  
    import flash.text.TextFormat;  
  
    [SWF(width=800, height=800, backgroundColor=0xffffffff)]  
    public class Container3D extends Sprite {  
        private var _sprite:Sprite;  
        private var _n:Number = 0;  
  
        public function Container3D() {  
            stage.align = StageAlign.TOP_LEFT;  
            stage.scaleMode = StageScaleMode.NO_SCALE;  
  
            _sprite = new Sprite();  
            _sprite.y = stage.stageHeight / 2;  
  
            for(var i:int = 0; i < 100; i++) {  
                var tf:TextField = new TextField();  
                tf.defaultTextFormat = new TextFormat("Arial", 40);  
                tf.text = String.fromCharCode(65 + Math.floor(Math.random() * 25));  
                tf.selectable = false;  
                tf.x = Math.random() * 300 - 150;  
                tf.y = Math.random() * 300 - 150;  
                tf.z = Math.random() * 1000;  
                _sprite.addChild(tf);  
            }  
  
            addChild(_sprite);  
            addEventListener(Event.ENTER_FRAME, onEnterFrame);  
        }  
        private function onEnterFrame(event:Event):void {  
            _sprite.x = stage.stageWidth / 2 + Math.cos(_n) * 200;  
            _n += .05;  
        }  
    }  
}
```

每个 Text field 随机地放置在父 sprite 的 3 个区域中。尽管 sprite 被限制只能在 X 轴上前后来回运动，但你可以在图 7-7 中看到字母在 3D 空间中移动的效果。



图 7-7 在移动的情况下 3D 容器看起来将更好

### 3D 旋转

除了在 3D 空间里移动对象外，你还可以绕任何坐标轴旋转任何对象。我们已经在本章开头看了一个绕 Y 轴旋转的小例子。因为你可以自己把一个对象绕 X 轴和 Z 轴旋转，所以我觉得没必要让我再带着你重复这个例子，并且我觉得你很可能在这个类之前就已经立刻把一些东西绕 3 个轴旋转过了。（如果你还没有，赶紧去试试吧。）

当你准备好以后，让我们开始旋转包含显示对象的容器。首先，我们要重新创建第一个实验，但先要放一个物体进去。你能在书的下载页中发现 RotateAndPosition.as 文件。

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

    [SWF(width=800, height=800, backgroundColor = 0xffffffff)]
    public class RotateAndPosition extends Sprite {
        private var _holder:Sprite;

        public function RotateAndPosition() {
            _holder = new Sprite();
            _holder.x = stage.stageWidth / 2;
            _holder.y = stage.stageHeight / 2;
            addChild(_holder);

            var shape1:Shape = new Shape();
            shape1.z = 200;
            shape1.graphics.beginFill(0xffffffff);
            shape1.graphics.drawRect(-100, -100, 200, 200);
            _holder.addChild(shape1);
        }
    }
}
```

```

        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void {
        _holder.rotationY += 2;
    }
}
}

```

这个例子给我们展示了一个和以前一样的平面。但是现在让我们在容器里移动这个平面。先是在 X 轴上-在构造函数中增加下面的粗体行。

```

var shape:Shape = new Shape();
shape.x = 200;
shape.graphics.beginFill(0xffffffff);
shape.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape);

```

现在不仅是旋转，而是一种绕中心轨道的旋转。通过先绕 Z 轴移动我们可以得到一个不一样的效果。

```

var shape:Shape = new Shape();
shape.z = 200;
shape.graphics.beginFill(0xffffffff);
shape.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape);

```

这真是太酷了，让我们再来新建一个平面。

```

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

    [SWF(width=800, height=800, backgroundColor = 0xffffffff)]
    public class RotateAndPosition extends Sprite {
        private var _holder:Sprite;

        public function RotateAndPosition() {
            _holder = new Sprite();
            _holder.x = stage.stageWidth / 2;
            _holder.y = stage.stageHeight / 2;
            addChild(_holder);

            var shape1:Shape = new Shape();
            shape1.z = 200;
            shape1.graphics.beginFill(0xffffffff);
            shape1.graphics.drawRect(-100, -100, 200, 200);
            _holder.addChild(shape1);
            var shape2:Shape = new Shape();
            shape2.z = -200;
            shape2.graphics.beginFill(0xffffffff);
            shape2.graphics.drawRect(-100, -100, 200, 200);
            _holder.addChild(shape2);

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void {

```

```

        _holder.rotationY += 2;
    }
}

```

我们将把其中一个放在 Z 值为 200，另一个为-200 的地方。现在当我们旋转容器时，它们也将绕着对方旋转。但我们为什么对绕 Y 轴旋转限制呢？改变 onEnterFrame 方法，增加绕其他轴的旋转：

```

private function onEnterFrame(event:Event):void {
    _holder.rotationY += 2;
    _holder.rotationX += 1.5;
}

```

哎，这简直太简单了。让我们多增加一些平面！这次我们要将它们在 X 轴上左右移动，也要将它们旋转 90 度：

```

var shape3:Shape = new Shape();
shape3.x = 200;
shape3.rotationY = 90;
shape3.graphics.beginFill(0xffffffff);
shape3.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape3);

```

```

var shape4:Shape = new Shape();
shape4.x = -200;
shape4.rotationY = -90;
shape4.graphics.beginFill(0xffffffff);
shape4.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape4);

```

现在我们有四面墙相互环绕着。为什么停在那儿呢？你很有可能走在了我的前面，但还是让我们增加一个地板和天花板（粗略地来看）：

```

var shape5:Shape = new Shape();
shape5.y = 200;
shape5.rotationX = 90;
shape5.graphics.beginFill(0xffffffff);
shape5.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape5);

```

```

var shape6:Shape = new Shape();
shape6.y = -200;
shape6.rotationX = -90;
shape6.graphics.beginFill(0xffffffff);
shape6.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape6);

```

你可以在图 7-8 中看到结果。



图 7-8 旋转立方体！

图 7-9 出问题了

是不是很简洁呀，并且是个令人惊奇的例子，没有任何一行的修饰。我也确切地知道你正在想什么：由面片组成的立方体！也许和你想的不是很一样，但我确定你肯定觉得这些红色的平面有点单调，需要一些不同的东西。当然，一旦你开始改变颜色，并且继续下去，你会破坏掉我精心设计的幻境。算了吧，你就去试试吧。通过改变 beginFill 方法里的十六进制值让每个平面拥有不同的颜色。如果你想的话，你甚至还可以让他们随机地显示颜色：

```
Shape1.graphics.beginFill(Math.random() * 0xffffffff);
```

你可以看到图 7-9 似的情景。

现在立方体的面不是全红色的了，然而我们却看见了问题。是否这里的截图不是很清楚，发生了什么呢，立方体应该显示在背面的部分现在显示在了前面。我希望你已经学到了很多，并且能够立即认识到这是由于景深排序出了问题。我们已经了解了景深排序，并知道如何去处理它，所以我们正好可以深入并应用它。接下来的类尝试去解决我们刚学的景深排序的问题-方法是按照 Z 值排序并以此顺序把它们添加进显示列表，你可以在 RotateAndPosition2.as 文件中找到它。我添加了一个 makeShape 方法来去除一些重复的代码。

```
package {
    import flash.display.DisplayObject;
    import flash.display.Shape;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Vector3D;

    [SWF(width=800, height=800, backgroundColor = 0xffffffff)]
    public class RotateAndPosition2 extends Sprite {
        private var _holder:Sprite;
        private var _shapes:Array;

        public function RotateAndPosition2() {
            _holder = new Sprite();
            _holder.x = stage.stageWidth / 2;
            _holder.y = stage.stageHeight / 2;
            _holder.z = 0;
            addChild(_holder);

            var shape1:Shape = makeShape();
```

```

shape1.z = 200;

var shape2:Shape = makeShape();
shape2.z = -200;

var shape3:Shape = makeShape();
shape3.x = 200;
shape3.rotationY = 90;

var shape4:Shape = makeShape();
shape4.x = -200;
shape4.rotationY = -90;

var shape5:Shape = makeShape();
shape5.y = 200;
shape5.rotationX = 90;

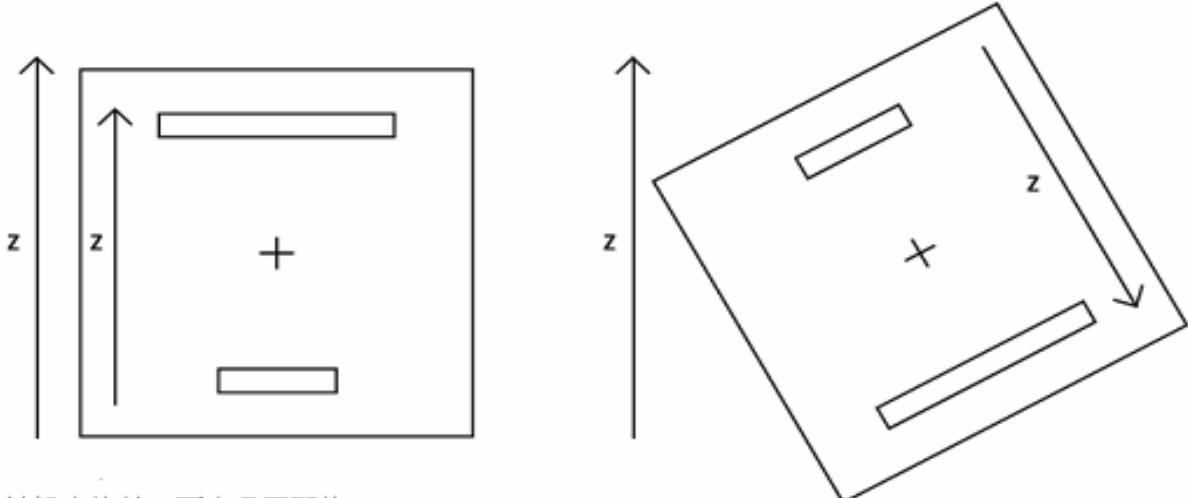
var shape6:Shape = makeShape();
shape6.y = -200;
shape6.rotationX = 90;
_shapes = [shape1, shape2, shape3, shape4, shape5, shape6];
sortShapes();
addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function makeShape():Shape {
var shape:Shape = new Shape();
shape.graphics.beginFill(Math.random() * 0xffffffff);
shape.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape);
return shape;
}
private function sortShapes():void {
    _shapes.sort(depthSort);
    for(var i:int = 0; i < _shapes.length; i++) {
        _holder.addChildAt(_shapes[i] as Shape, i);
    }
}
private function onEnterFrame(event:Event):void {
    _holder.rotationY += 2;
    _holder.rotationX += 1.5;
    sortShapes();
}
}
}
}

```

除了清理外，我们还有一个包含所有物体对象的数组`_shapes`。在每次旋转后，`sortShapes`方法会被调用。它按照 Z 值顺序对物体对象数组进行排序，并且按照正确顺序把每个对象添加进它的 `holder` 中。

但是，当我们测试它后，它并没有弥补那个错误。问题出在我们只是在容器内部按照 Z 值排序。所以尽管他们有正确的 Z 值顺序，但当容器调转的时候，它们刚好突然完全地反转了。我们需要做的是从 `holder` 的外面来进行正确的排序。换句话说，即使对象 A 的 Z 值比对象 B 的小，当容器就 Z 值来说被旋转至反转后，它也会出现在对象 B 的后面。



*z轴没有旋转，两个是匹配的。*

*旋转后，局部z坐标不同于世界z坐标。*

图 7-10 旋转一个 3D 容器的效果

为了解决这个问题，按惯例我们要写一个函数把容器内的局部坐标系转换为世界（或舞台、根）坐标系，然后根据它进行排序。Array.sort 方法允许你把一个函数作为参数传递。然后这个函数会随着数组中一对对的对象排序而被多次调用。如果数组中第一个对象应该放置在第二个对象的前面，这个函数会返回一个负数；如果数组中第一个对象应该放置在第二个对象的后面，这个函数会返回一个正数；如果应该放置在它原来的位置，就返回零。

所以我们现在需要一种转换局部坐标系到世界坐标系的方法。这里有很多方法可以做到，包括手动旋转坐标系，这会涉及到很多复杂的三角运算。Flash 10 现在有一个新的类 flash.geom.Matrix3D，它包括很多有用的操作 3D 坐标系的方法。即使在这儿，依然有很多可以完成我们所期望问题的方法。我不知道我正在展示的方法是不是最好的，但它确实很好地解决了我们的问题。

首先我们需要得到一个代表容器旋转的 Matrix3D。通过打入下面的代码你就可以得到它：

`Container.transform.matrix3D`

然后我们就可以调用 `deltaTransformVector` 方法，传递一个点的 3D 原位置，然后得到它旋转后的位置：

**`rotatedPosition=_holder.transform.matrix3D.deltaTransformVector(originalPosition)`**

如果我们对两个分开的对象这么做的话，我们就可以得知从一个世界坐标系的视点看，在 Z 轴上哪个更远。最后一个伤脑筋的问题是我们如何得到一个显示物体的 3D 向量位置。我们可以用 X, Y, Z 坐标值创建一个位置，但这只能在这个位置已经存在的情况下：

`displayObject.transform.Matrix3D.position`

现在我们解决了如何获得世界坐标系里两个显示对象的所有问题：

```
var posA:Vector3D = objA.transform.matrix3D.position;
posA = _holder.transform.matrix3D.deltaTransformVector(posA);
var posB:Vector3D = objB.transform.matrix3D.position;
posB = _holder.transform.matrix3D.deltaTransformVector(posB);
```

然后我们可以写一个用于排序比较的函数，它会决定哪个对象显示在前面：

```
private function depthSort(objA:DisplayObject, objB:DisplayObject):int {
    var posA:Vector3D = objA.transform.matrix3D.position;
    posA = _holder.transform.matrix3D.deltaTransformVector(posA);
    var posB:Vector3D = objB.transform.matrix3D.position;
    posB = _holder.transform.matrix3D.deltaTransformVector(posB);
}
```

如果从一个已经旋转过的世界坐标系的视点来看，对象 A 比对象 B 远的话，这个函数会返回一个负数，反映在数组中排序时对象 A 就应该放在对象 B 前面。然后我们可以很简单的在 sortShapes 函数中使用它：

```
private function sortShapes():void {  
    _shapes.sort(depthSort);  
    for(var i:int = 0; i < _shapes.length; i++) {  
        _holder.addChildAt(_shapes[i] as Shape, i);  
    }  
}
```

现在物体的旋转问题解决了，一个漂亮的 3D 物体出现了，如图 7-11：



图 7-11 即使在旋转的容器里也是正确的景深排序

简单地修改一下，我们会得到一种幻灯机式的布局，它经常会被用于导航或展示满是图片的画廊。在下面的例子中我不会真实地装载任何图片，但你可以很容易地修改 sprites 使它成为装载者，并且给它一系列 URLs。无论如何，下面是这段代码，在 Carousel.as 文件中：

```
package {  
    import flash.display.DisplayObject;  
    import flash.display.Sprite;  
    import flash.events.Event;  
    import flash.geom.Vector3D;  
  
    [SWF(width=800, height=800, backgroundColor = 0xffffffff)]  
    public class Carousel extends Sprite {  
        private var _holder:Sprite;  
        private var _items:Array;  
        private var _radius:Number = 200;  
        private var _numItems:int = 5;  
  
        public function Carousel() {  
            _holder = new Sprite();  
            _holder.x = stage.stageWidth / 2;  
            _holder.y = stage.stageHeight / 2;  
            _holder.z = 0;
```

```

addChild(_holder);
_items = new Array();
for(var i:int = 0; i < _numItems; i++) {
    var angle:Number = Math.PI * 2 / _numItems * i;
    var item:Sprite = makeItem();
    item.x = Math.cos(angle) * _radius;
    item.z = Math.sin(angle) * _radius;
    item.rotationY = -360 / _numItems * i + 90;
    _items.push(item);
}
sortItems();

addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function makeItem():Sprite {
    var item:Sprite = new Sprite();
    item.graphics.beginFill(Math.random() * 0xffffffff);
    item.graphics.drawRect(-50, -50, 100, 100);
    _holder.addChild(item);
    return item;
}
private function sortItems():void {
    _items.sort(depthSort);
    for(var i:int = 0; i < _items.length; i++)
    {
        _holder.addChildAt(_items[i] as Sprite, i);
    }
}
private function depthSort(objA:DisplayObject, objB:DisplayObject):int {
    var posA:Vector3D = objA.transform.matrix3D.position;
    posA = _holder.transform.matrix3D.deltaTransformVector(posA);
    var posB:Vector3D = objB.transform.matrix3D.position;
    posB = _holder.transform.matrix3D.deltaTransformVector(posB);
    return posB.z - posA.z;
}

private function onEnterFrame(event:Event):void {
    _holder.rotationY += (stage.stageWidth / 2 - mouseX) * .01;
    _holder.y += (mouseY - _holder.y) * .1;
    sortItems();
}
}
}

```

最大的变化在粗体的代码里：图片如何获得位置信息；如何旋转；容器如何在 onEnterFrame 方法中移动。我们不再手动放置每个面片，这次我们用一个循环来做，按照项数把 `Math.PI*2` 弧度分割开，并且用当前项目的号码赋给每个对象。除了一个半径外每一项还需要 X 和 Z 坐标，并用一些三角函数计算出结果角度。然后我们用 `rotationY` 属性做一个相似的计算，这次直接用角度。最后，在 `enterFrame` 处理函数中，我们根据鼠标的位置把容器在 Y 轴上旋转，也就是跟着鼠标在 Y 轴上来来去去。

你可以在图 7-12 中看到它的样子。



图 7-12 一个 3D 的幻灯机

现在我们知道一些基本的确定位置和旋转的知识，让我们更进一步，来看看怎样更好地调整 3D 的外表。

### 视野和焦距

很明显上，当你在任何种类的平面的屏幕上看一个图片的时候，实际上你都是在看一个二维的图片。程序通过一些小技巧给我们一些幻觉，让我们看到一个二维平面上的东西有三维的感觉。这些技巧来自于透视的原理。

一个技巧是让应该远一点的物体显示在近点物体的后面。我们在上一节做景深排序时就是这么做的。另一个技巧是让在远处的物体像雾一样看起来在慢慢消失。通常地，你可以让特定距离的物体刚好在焦距上，任何远的或近的物体都在焦距之外。这就是我们所谓的景深。

但最有影响力的技巧还是让远处的物体变小，并在他们远离和变小的过程中让它们接近消失点。当然，景深排序对呈现 3D 感觉来说也是很重要的。不合适的景深排序无疑会破坏任何 3D 的感觉，就像你已经看见的那样。但是如果你只做景深排序不根据它们的深度进行缩放的话，你也不会得到很好的 3D 的感觉。

一个大问题是你是如何根据他们的远近确定他们的缩放比例？幸运，这个问题在个人电脑出现之前就已经被艺术家，工程师还有摄影师提出和解决了。这些问题到最后就剩下光学和镜头和你的眼睛或照相机如何工作的问题。如果你做过任何摄影的话，你就会知道有广角镜头和长焦镜头（还有两者之间的全距镜头）。甚至还有鱼眼镜头，本质上就是超广角镜头。

一个广角镜头有一个很大的视野。换句话说，如果你想在镜头前面设计一个弧线或圆锥，它就把镜头能“看见”的很宽区域给复盖了。一个鱼眼镜头能达到超过 180 度的视角。而对于一个长焦镜头，这个圆锥角会很小，只能覆盖掉它前面很小的区域。

除了视野外，还有另外一个概念：焦距，它指的是从镜头中心到焦点（穿过镜头的光束汇聚的点）的距离。看图 7-13。

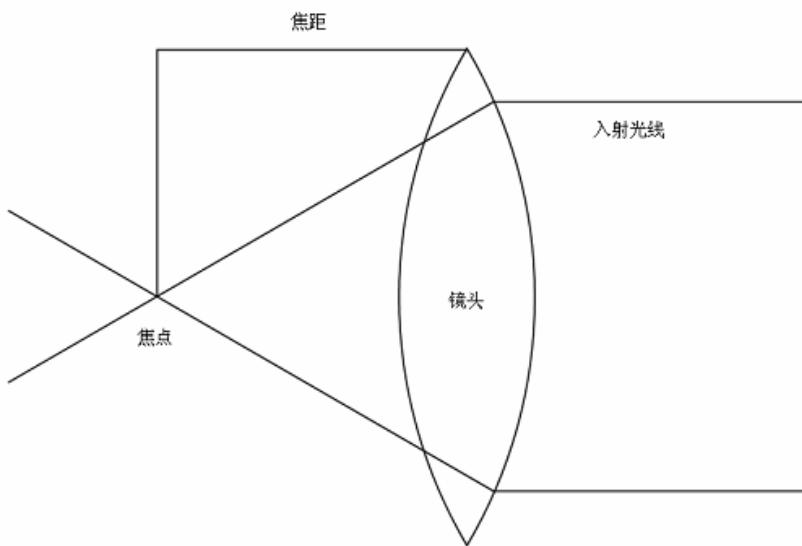


图 7-13 镜头的焦距

焦距的描述和 Adobe 帮助文档中的不是很一样，但它们最终仍然都是一个概念。焦距和视野联系很紧密并且都决定着物体看起来应该缩放和扭曲多少。用一个广角镜头，会产生很宽的视野，也会产生一个短的焦距和更大的缩放比例（这就是为什么鱼眼镜头照的都看起来很扭曲）。

用一个长焦镜头，会产生很窄的视野，也会产生一个长的焦距和很小的缩放比例。一个很好的例子是从比赛外场进行拍摄的棒球比赛的照片，尽管击球手站在很远外，投球手和击球手看起来好像是同样的大小。在人类的感知中，这种只有很少或没有扭曲的照片实际上让人们看到更多的扭曲，并给人一种击球手很巨大的感觉。在图 7-14 和 7-15 中你可以看到视野和焦距的关系。

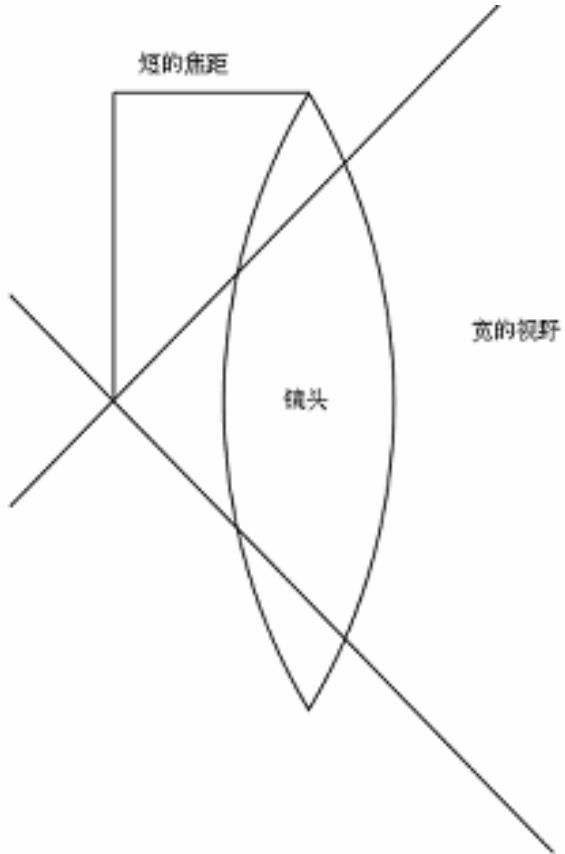


图 7-14 广角镜头，短焦距  
长焦距

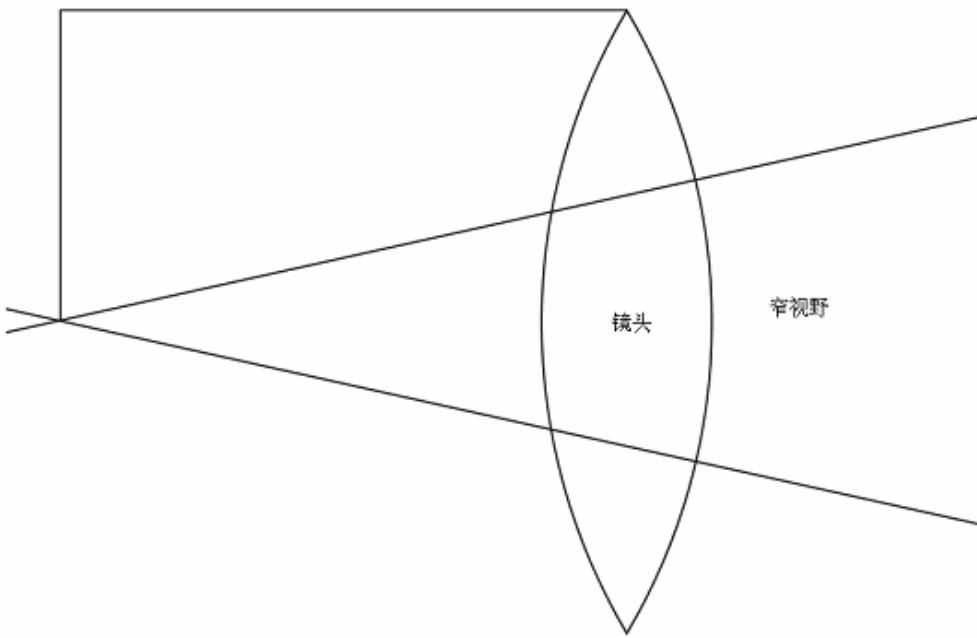


图 7-15 窄角（长焦）镜头，长焦距

在 Flash 3D 中，你可以通过设置焦距或视野来控制扭曲。实际上，设置了一个另一个会跟着

改变，但你要选择一个你感觉最舒服的去用。这个工作需要设置显示对象的 transform 属性的 perspectiveProjection 属性：focalLength 和 fieldOfView。通常地，最好在你的 movie 的根目录设置这些属性，除非你的容器有一特殊的要求或者对象有特殊的属性（比如在不同的窗口中看对象）。

视野需要用度数进行度量，并且需要大于 0 小于 180，否则将出现错误。设置一个视野为 0 意味着你将什么都看不到。设置为 180 会让你的焦距为 0，我想象这样的话也会在你计算和显示图片时出现问题。无论如何，你要尽量离这两个极端数字远点。一个接近 0 的视野会让你的焦距趋向于无穷大，这会使任何 3D 的缩放效果都显示不出来。一个接近 180 的视野会让你的焦距很微小，这会使你想显示的景物产生巨大扭曲。在现实物理世界，让视野大于 180 是可能的。最大视角的镜头曾经能够达到 220 度的视野！

无论如何，回到 Flash，摆弄这些值然后看他们如何运作是件有趣的事。你完全可以从上节的 Carousel 例子开始。仅仅在构造函数中添加下列粗体代码：

```
public function Carousel() {  
    root.transform.perspectiveProjection.fieldOfView=110;  
    _holder = new Sprite();  
    _holder.x = stage.stageWidth / 2;  
    _holder.y = stage.stageHeight / 2;  
    _holder.z = 0;  
    addChild(_holder);  
    .....  
}
```

这个展示了一个非常好场景。因为靠近“相机”的平面现在很大，这在画廊的应用中会是一个好的设置。看图 7-16。



图 7-16 宽的视野显示时扭曲度就会变大

调低到 25，这在相当大程度上减小了视野。现在显示的扭曲度不那么明显了。看图 7-17。

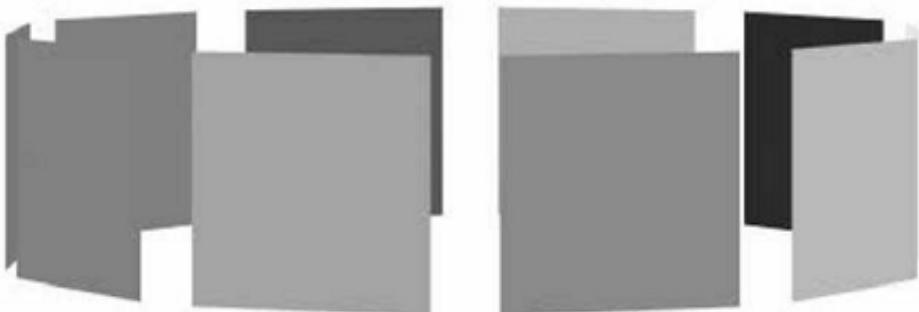


图 7-17.窄的视野，小的扭曲度

你还可以尝试各种大小不同的焦距，看看会有什么效果，就像接下来的例子：

```
root.transform.perspectiveProjection.fieldOfView=300;
```

再次说明小的焦距带来大的扭曲度，大的焦距带来小的扭曲度。

## 屏幕坐标系和 3D 坐标系

偶尔你可能需要知道和一个 3D 空间中的点相对应的屏幕坐标。相反，你也可能需要知道一个屏幕坐标点指向 3D 空间的哪里。幸运的是，显示对象有两个内建的方法就是用于这个目的：local3DtoGlobal 和 globalToLocal3D。第一个从 flash.geom.Vector3D 转换成一个 2D flash.geom.Point 对象，第二个作相反的转换。

首先，让我们来看 local3DtoGlobal 方法的应用。我们要新建一个 sprite 并把它在 3D 空间中移动。在这个 sprite 中有很多几何体，有一个 x=200, y=0, z=0 的圆。然后我们要创建另一个 sprite，它会把局部 3D 坐标 (200, 0, 0) 转换为世界屏幕坐标，然后在圆移动时跟踪它。这是这个类，你可以在 LocalGlobal.as 文件中找到：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Point;
    import flash.geom.Vector3D;

    [SWF(width=800, height=800)]
    public class LocalGlobal extends Sprite  {
        private var _sprite:Sprite;
        private var _tracker:Sprite;
        private var _angle:Number = 0;

        public function LocalGlobal()  {
            _sprite = new Sprite();
            _sprite.graphics.lineStyle(10);
            _sprite.graphics.lineTo(200, 0);
            _sprite.graphics.drawCircle(200, 0, 10);
            _sprite.x = 400;
            _sprite.y = 400;
            addChild(_sprite);

            _tracker = new Sprite();
            _tracker.graphics.lineStyle(2, 0xff0000);
            _tracker.graphics.drawCircle(0, 0, 20);
            addChild(_tracker);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void      {
            _sprite.rotationX += 1;
            _sprite.rotationY+= 1.2;
            _sprite.rotationZ += .5;
            _sprite.x = 400 + Math.cos(_angle) * 100;
            _sprite.y = 400 + Math.sin(_angle) * 100;
            _sprite.z = 200 + Math.cos(_angle * .8) * 400;
            _angle += .05;
            var p:Point = _sprite.local3DToGlobal(new Vector3D(200, 0, 0));
            _tracker.x = p.x;
            _tracker.y = p.y;
        }
    }
}
```

}

构造函数创建了一个 3Dsprite 和一个跟踪 sprite，并放了一些几何体在里面。enterFrame 方法处理了大部分的事务，包括用一种看起来很随机的方式在 3D 空间里移动那个 sprite 的代码。它用我刚随机添加的数字绕所有 3 个轴迅速移动和旋转。最重要的是 local3DtoGlobal 那行，它把 (200, 0, 0) 转换为 2D 坐标，然后把那个位置信息交给跟踪 sprite。当你运行这个后，你会看到不论 sprite 在所有 3 个维度里怎么移动，跟踪 sprite 都可以很好地跟着那个圆圈。

看图 7-18。

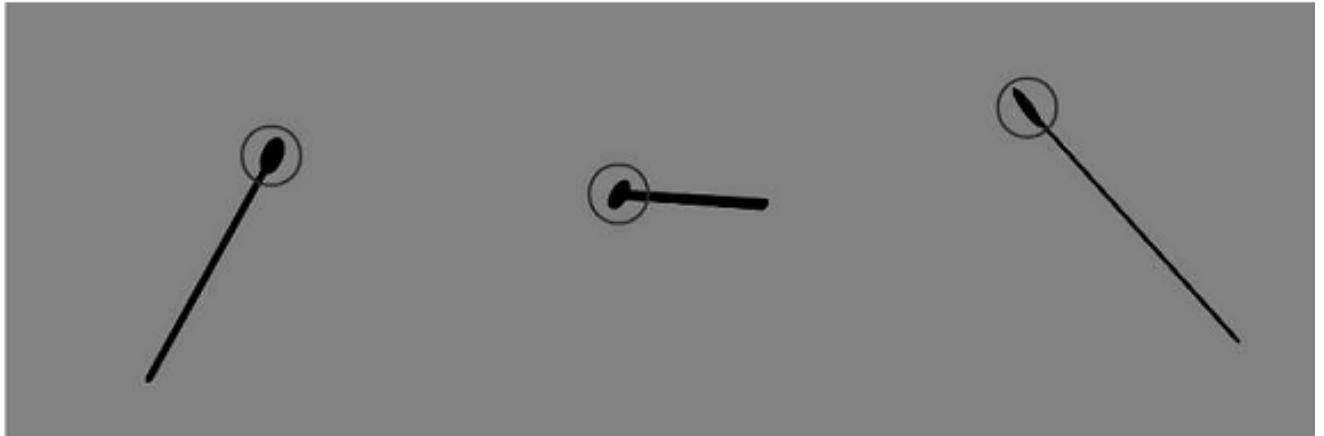


图 7-18 在 2D 平面里跟踪 3D 物体的多个击中示例

这个可以有很多用处，包括知道何时一个 3D 对象移出屏幕。因为一个 3D 对象的 x 或 y 坐标可能会比屏幕坐标系大很多，并可能在位于 z 轴很远的地方时仍然能够显示在屏幕上，这对判断一个对象是否仍然在屏幕上是很有用的。

另一方面，我们也可以把屏幕坐标转换为局部 3D 坐标。我对上一个类做了一点改动，现在的类是 GlobalLocal，在文件 GlobalLocal.as 中。主要的改变用粗体标注了：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Point;
    import flash.geom.Vector3D;

    [SWF(width=800, height=800)]
    public class GlobalLocal extends Sprite {
        private var _sprite:Sprite;
        private var _tracker:Sprite;
        private var _angle:Number = 0;

        public function GlobalLocal() {
            _sprite = new Sprite();
            _sprite.graphics.lineStyle(5);
            _sprite.graphics.drawRect(-200, -200, 400, 400);
            _sprite.x = 400;
            _sprite.y = 400;
            addChild(_sprite);

            _tracker = new Sprite();
            _tracker.graphics.lineStyle(2, 0xff0000);
            _tracker.graphics.drawCircle(0, 0, 20);
            _sprite.addChild(_tracker);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
    }
}
```

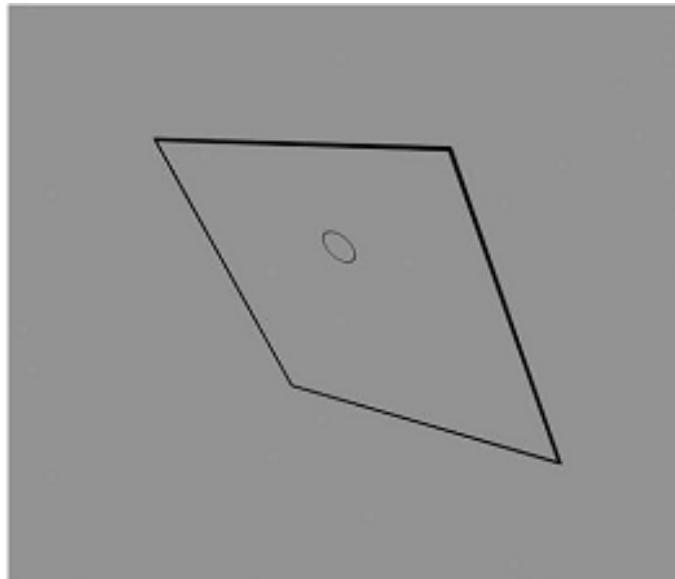
```

    }

    private function onEnterFrame(event:Event):void {
        _sprite.rotationX += 1;
        _sprite.rotationY+= 1.2;
        _sprite.rotationZ += .5;
        _sprite.x = 400 + Math.cos(_angle) * 100;
        _sprite.y = 400 + Math.sin(_angle) * 100;
        _sprite.z = 200 + Math.cos(_angle * .8) * 400;
        _angle += .05;
        var p:Vector3D = _sprite.globalToLocal3D(new Point(mouseX, mouseY));
        _tracker.x = p.x;
        _tracker.y = p.y;
    }
}
}
}

```

这里我们给旋转的 sprite 画了一个大的平面，并且把跟踪器放在了它的里面。在 enterFrame 方法中，我们调用 globalToLocal3D 来得到和当前鼠标位置相关联的 3D 坐标。这个显示出来是一个 3D 物体。在这种情况下这个 3D 物体的 Z 值一直是零，所以我们只能用它的 x 和 y 值来设置旋转 sprite 中的跟踪 sprite 的位置。正如你看到的，鼠标在二维平面上运动，而 sprite 在 3D 空间中跟着鼠标移动。你可以在图 7-19 中看到：



**图 7-19.在 3D 空间中跟踪 2D 坐标。**

尽管这个已经很酷了，但我发现了另一个更简单的方法：用旋转对象的局部鼠标坐标。结果是如果你存取一个在 3D 空间里改变的对象的 mouseX 和 mouseY 参数时，世界坐标到局部坐标的 3D 转换会自动进行。所以我们的 onEnterFrame 函数将变得更加简单：

```

private function onEnterFrame(event:Event):void {
    _sprite.rotationX += 1;
    _sprite.rotationY+= 1.2;
    _sprite.rotationZ += .5;
    _sprite.x = 400 + Math.cos(_angle) * 100;
    _sprite.y = 400 + Math.sin(_angle) * 100;
    _sprite.z = 200 + Math.cos(_angle * .8) * 400;
    _angle += .05;
    _tracker.x = _sprite.mouseX;
}
}
}
}

```

```
_tracker.y = _sprite.mouseY;
}
```

这样的修改做了同样的事情。记住这只是在转换鼠标坐标时有用。如果你想转换一个在舞台上的对象到 3D 坐标系，你仍然要用转换函数。

## 指向某物

一旦你开始觉得对 Flash 10 的 3D 适应了以后，你可能想开始在帮助文档里丰富的类中闲逛来看它们会产生什么效果。一个开始的好地方是 flash.geom 包，它包括了 Matrix3D, Orientation3D, perspectiveProjection, Utils3D 和 Vector3D 类。所有的这些类都塞满了能帮你做各种 3D 运算的方法。我在 Matrix3D 类中挖到的一个很简洁的方法是 pointAt。

pointAt 方法接收一个 Vector3D 对象作为指向的目标。如果这个函数是被显示对象的 transform 属性的 Matrix3D 对象调用，它会在 3D 空间中旋转那个显示对象使它指向一个特殊的区域。接下来的这个类，在 FollowMouse3D.as 文件中，演示了这个例子：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.geom.Vector3D;

    [SWF(width=800, height=800, backgroundColor = 0xffffffff)]
    public class FollowMouse3D extends Sprite {
        private var _sprite:Sprite;
        private var _angleX:Number = 0;
        private var _angleY:Number = 0;
        private var _angleZ:Number = 0;

        public function FollowMouse3D() {
            _sprite = new Sprite();
            _sprite.x = 400;
            _sprite.y = 400;
            _sprite.z = 200;
            _sprite.graphics.beginFill(0xff0000);
            _sprite.graphics.moveTo(0, 50);
            _sprite.graphics.lineTo(-25, 25);
            _sprite.graphics.lineTo(-10, 25);
            _sprite.graphics.lineTo(-10, -50);
            _sprite.graphics.lineTo(10, -50);
            _sprite.graphics.lineTo(10, 25);
            _sprite.graphics.lineTo(25, 25);
            _sprite.graphics.lineTo(0, 50);
            _sprite.graphics.endFill();
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void {
            _sprite.x = 400 + Math.sin(_angleX += .11) * 200;
            _sprite.y = 400 + Math.sin(_angleY += .07) * 200;
            _sprite.z = Math.sin(_angleZ += .09) * 200;
            _sprite.transform.matrix3D.pointAt(new Vector3D(mouseX, mouseY, 0));
        }
    }
}
```

构造函数里包含了很多行用于在一个 sprite 中绘制一个箭头的代码。

`onEnterFrame` 方法里包括了大部分的在 3D 空间中移动那个 `sprite` 的代码，不像 `LocalGlobal` 那个例子。最后一行调用了 `sprite` 的 `transform` 属性的 `matrix3D` 对象的 `pointAt` 方法。它接收一个由鼠标的 x 坐标, y 坐标再加上等于零的 z 值组成的 `Vector3D` 对象。就像所说的，画有箭头的 `sprite` 会始终跟着鼠标。你可以任意移动鼠标，跟踪 `sprite` 也会在整个区域运动，并绝对不会跟丢。(看图 7-20)。



图7-20. 在3D空间  
中跟踪鼠标。

我不确定这个例子的内涵或其本身有什么实用性，但我可以想起这种技术在制作 3D 游戏中的很多种应用：驾驶，瞄准等等。

### 本章小结

在本章中尽管我们涉及了很多东西，但我们仍然只简单地接触到了 Flash10 3D 的表层。在下一章，我们会更加深入地了解更多的关于绘图 API 特性的课题，但希望这章已经给你一个对这个课题跳跃性的认识。还是好好看看文档吧，尤其是 `flash.display.DisplayObject` 和 `flash.geom` 里所有的东西。我相信你会感觉很快乐的。

## 第八章 Flash10 的绘画 API

如果你没有六年以上的 Flash 游龄，就没有对 ActionScript 的绘画 API 的一个全面认识。在我看来，2002 年发布的 Flash MX 在脚本绘图方面是革命性的。早在 Flash MX 之前，压根就没有一种动态的绘图方式。以至于此刻，我都很想象那真实的过去——如果想要在 SWF 里显示些什么，就必须在创作期绘制在场景或者库里面，要么就是加载外部内容。然而，存在一些动态创建图形的技巧，比如将一个仅含有一条直线的 MovieClip，多复制几份到场景上，然后移动并缩放它们。

到了 Flash MX 时期，是第一次可以用纯代码进行开发，所有图形内容都可以凭空创建。尽管当时的 API 简陋至极（绘制命令只有 `moveTo`、`lineTo`、`curveTo`，以及后来增加的一些绘制矩形、圆形和椭圆的方法），但在 Flash 平台仍刮起了大量创建图形的旋风。实际上，很多复杂的应用程序的某些界面都是用绘制 API 创造的。

了解了一些背景后，让我们开始介绍 Flash10 的升级版绘制 API，它无所不能又小巧玲珑。实际上，它在新的命令数量、新的对象数量、巨大的威力和复杂度上，使原有的 API 显得相形见绌。所以，为了把这一切都搞清楚，我们需要分步研究。

### 路径

首先讨论的话题是路径，但这和第四章提到的路径是两个概念，所以我要在此为之做第二次定义。在新的绘图 API 中，一条路径是一系列点配合一系列相关的绘制命令（`moveTo`、`lineTo`、`curveTo`），其作用是画出一个图形。并不是简单的“连点画线”。

早期（Flash MX 到 Flash CS3 时期）的绘图，所能做的只有调用 `moveTo`、`lineTo`、`curveTo` 这几个函数。刚开始，为了画图可能要手写一大堆图形函数。慢慢地，发现用数组保存点的位置，然后循环调用 `lineTo` 来的更方便。最后，设计出一种方法根据需要来决定 `moveTo` 之后是调用 `lineTo` 还是调用 `curveTo`。如果你有以上的经历，或者理解上述过程，那么在对路径的理解上就没什么问题了。因为路径就是：一系列点加一系列绘制命令。

先来看看它的原型，是 `flash.display.Graphics` 类的 `drawPath` 函数：

```
drawPath(commands:Vector.<int>, data:Vector.<Number>)
```

参数是两个 `Vector`，一个保存 `int` 型，作为命令；另一个保存 `Number` 型，作为数据。命令就是 `moveTo`、`lineTo`、`curveTo`（还有一些等下就能看到），它们以 `int` 型作为代号被保存在

`flash.display.GraghpicsPathCommand` 类里。如下：

```
public static const NO_OP:int = 0;
public static const MOVE_TO:int = 1;
public static const LINE_TO:int = 2;
public static const CURVE_TO:int = 3;
public static const WIDE_MOVE_TO:int = 4;
public static const WIDE_LINE_TO:int = 5;
```

编号 1、2、3 应该很熟悉了。命令 `NO_OP`，就好像 `null`，是什么都不做的意思。还有两个 `wide` 打头的命令在稍后会讲到。

当画直线时，通常是从一点笔直画到另外一点，利用绘制命令的话，就像这样：

```
var commands:Vector.<int> = new Vector.<int>();
commands[0] = GraghpicsPathCommand.MOVE_TO;
commands[1] = GraghpicsPathCommand.LINE_TO;
```

也可以直接使用数字：

```
var commands:Vector.<int> = new Vector.<int>();
commands[0] = 1;
commands[1] = 2;
```

当然，这么做可读性几乎是 0，而且编译期也检测不出任何问题。你可能很愉快的赋一个 17 作为命令代号，还想着会不会是什么有用的功能，而程序在运行时就发生了异常，这在编译时是检测不到的。

接下来介绍数据。我们需要一条线段的两个端点。可能你觉得 `Vector` 中的元素应该是 `Point`

类型的，但事实上却是 Number 类型。其中第一个元素是点的 x 坐标，第二个元素是 y 坐标，接下来两个元素是下一个点的坐标，以此类推。尽管这样一开始会觉得很麻烦，但从执行速度和内存消耗上来讲，基本类型 Number 都要比复合类型 Point 来的效率高。改良的绘图 API 的关键是提升速率。需要确定的一件事情是，点的数量和命令的数量要匹配。如果点的数量少于命令的数量，Flash 将默认使用 0 作为数据继续下去，但请不要去利用这种默认情况。

下面，让我们试着画一条(100, 100)到(250, 200)的线段：

```
var data:Vector.<Number> = new Vector.<Number>();
data[0] = 100;
data[1] = 100;
data[2] = 250;
data[3] = 200;
```

现在可以调用 drawPath 了，首先还要设置一下 lineStyle。一个路径只是描述了和绘制有关的点和操作，指定填充或线条样式还得用老方法。

```
graphics.lineStyle(0);
graphics.drawPath(commands, data);
```

我们来看一段完成的测试代码：

代码 SingleLine.as

```
package
{
    import __AS3__.vec.Vector;

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

    [SWF(backgroundColor=0xffffffff)]
    public class SingleLine extends Sprite
    {
        public function SingleLine()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var commands:Vector.<int> = new Vector.<int>();
            commands[0] = GraphicsPathCommand.MOVE_TO;
            commands[1] = GraphicsPathCommand.LINE_TO;

            var data:Vector.<Number> = new Vector.<Number>();
            data[0] = 100;
            data[1] = 100;
            data[2] = 250;
            data[3] = 200;

            graphics.lineStyle(0);
            graphics.drawPath(commands, data);

        }
    }
}
```

我猜你正在想，光画条线就这么多种代码啊？没错，如果就画一条线，确实用 moveTo 和 lineTo 要简单的多。但随着逐渐深入，你会发现这些看似复杂的操作却能完成复杂到看似不可能的绘图。

而一个主要的意图是实现 3D 绘图。放心，我们稍后就会讨论。

现在，还是先深入了解一下 drawPath 的运用。

### 一个简单的例子

如果熟悉软件设计模式，那么此刻就会发现命令模式的身影。在命令模式中，对象代表行为。创建的一个命令对象其实就是封装了一个带参数的行为。我们要做的实际上就是保存一系列行为（绘制命令）和对应的参数（绘制数据）。

接下来的例子创建了一个小型的绘图程序。它会把绘制过程保存为一系列命令和数据，在调用 drawPath 的时候就能画出图形。同时还允许用户利用上下方向键来控制线条的粗细，空格键随机改变线条的颜色。不过，作为真正的应用程序，改变线条的样式应该更具用户体验，但现在是测试，所以先将就了。

代码：PathSketch.as

```
package
{
    import __AS3__.vec.Vector;

    import flash.display.GraphicsPathCommand;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.KeyboardEvent;
    import flash.events.MouseEvent;
    import flash.ui.Keyboard;

    [SWF(backgroundColor=0xffffffff)]
    public class PathSketch extends Sprite
    {
        private var commands:Vector.<int> = new Vector.<int>();
        private var data:Vector.<Number> = new Vector.<Number>();

        private var lineWidth:Number = 0;
        private var lineColor:uint = 0;

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

            stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
            stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
        }

        private function onMouseDown(event:MouseEvent):void
        {
            commands.push(GraphicsPathCommand.MOVE_TO);
            data.push(mouseX, mouseY);
            stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
            stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
            draw();
        }

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

```

        stage.removeEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
        stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    }

    private function onMouseMove(event:MouseEvent):void
    {
        commands.push(GraphicsPathCommand.LINE_TO);
        data.push(mouseX, mouseY);
        draw();
    }

    private function onKeyUp(event:KeyboardEvent):void
    {
        if(event.keyCode == Keyboard.DOWN)
        {
            lineWidth = Math.max(0, lineWidth -1);
        }
        else if(event.keyCode == Keyboard.UP)
        {
            lineWidth++;
        }
        else if(event.keyCode == Keyboard.SPACE)
        {
            lineColor = Math.random() * 0xffffffff;
        }
        draw();
    }

    private function draw():void
    {
        graphics.clear();
        graphics.lineStyle(lineWidth, lineColor);
        graphics.drawPath(commands, data);
    }
}

```

这段程序一开始创建了两个 Vector 用来存放命令和数据，然后设置了 mouseDown 和 keyUp 的事件监听器。下面我们来捋一遍程序，先从 mouseDown 开始：

```

private function onMouseDown(event:MouseEvent):void
{
    commands.push(GraphicsPathCommand.MOVE_TO);
    data.push(mouseX, mouseY);
    stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    draw();
}

```

这个函数会在用户点击场景的时候被调用。我们就认为这一点是绘制的起点，将 MOVE\_TO 放入命令 Vector 中，把鼠标坐标放入数据 Vector 中。

然后设置监听 mouseMove 和 mouseUp 事件，并调用一下 draw 函数。

onMouseUp 函数是用来移除鼠标事件的，这个好理解。我们还是看看 onMouseMove 函数：

```

private function onMouseMove(event:MouseEvent):void
{
    commands.push(GraphicsPathCommand.LINE_TO);
}

```

```

        data.push(mouseX, mouseY);
        draw();
    }

```

看上去和 onMouseDown 类似，在各自的 Vector 中加入了对应的内容，最后调用 draw。注意，刚开始在 onMouseDown 函数里调用 draw 的时候，命令 Vector 中只有一个 MOVE\_TO，所以执行 draw 只是移动到该点，而现在又加入了 LINE\_TO，这样就能从起点开始不断画线到鼠标所在位置了。让我们看看 draw 函数：

```

private function draw():void
{
    graphics.clear();
    graphics.lineStyle(lineWidth, lineColor);
    graphics.drawPath(commands, data);
}

```

非常简单。清空图像，设置线条样式，然后调用 drawPath。还有 onKeyUp 函数这里就不做介绍了，它主要就是负责改变线条的样式。

做个测试，在场景上画些线，然后用方向键和空格来改变线条的粗细和颜色。这儿虽然没有提到任何以前无法完成的东西，但从编程角度看确实容易了很多。

通过这类操作还能做更多的事情，比如 undo，redo 等等。在后续章节，还会看到更多简化以往操作的东西。在那之前，我们还是把余下和路径相关的功能都看完。

## 画曲线

用 drawPath 画曲线和画直线区别不大。只需要把 LINE\_TO 改为 CURVE\_TO，且增加一些必要的数据——一个控制点和一个终点。和 LINE\_TO 一样，曲线的绘制也是由 MOVE\_TO 或者其它命令作为开始的。

看一个简单的例子。我们创建八个点并在它们之间画上曲线：

代码 CurveDrawing.as

```

package
{
    import __AS3__.vec.Vector;

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

    [SWF(backgroundColor=0xffffffff)]
    public class CurveDrawing extends Sprite
    {
        private var commands:Vector.<int> = new Vector.<int>();
        private var data:Vector.<Number> = new Vector.<Number>();

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

            commands.push(GraphicsPathCommand.MOVE_TO);
            data.push(200, 200);

            data.push(250, 100);
            data.push(300, 200);
        }
    }
}

```

```

        data.push(400, 250);
        data.push(300, 300);

        data.push(250, 400);
        data.push(200, 300);

        data.push(100, 250);
        data.push(200, 200);

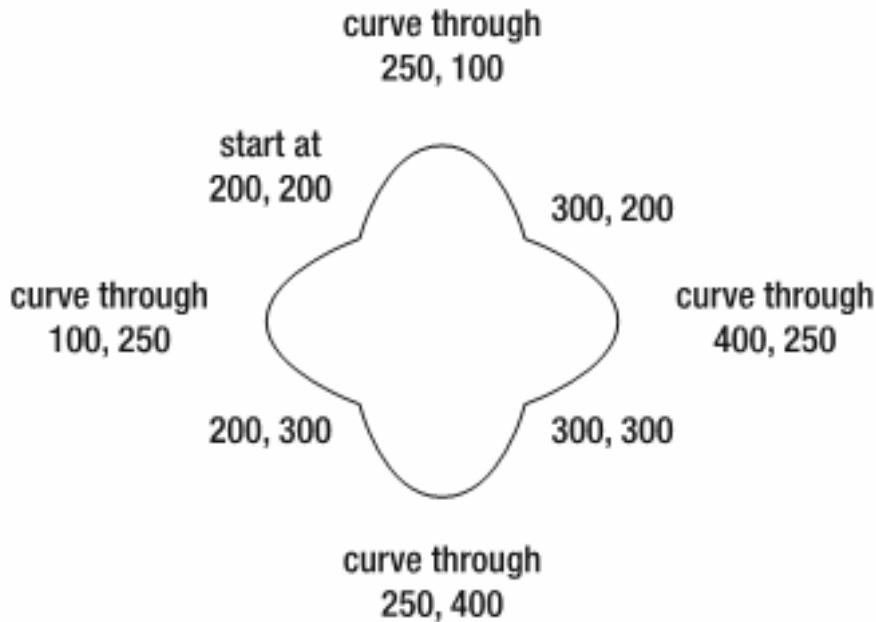
        commands.push(GraphicsPathCommand.CURVE_TO);
        commands.push(GraphicsPathCommand.CURVE_TO);
        commands.push(GraphicsPathCommand.CURVE_TO);
        commands.push(GraphicsPathCommand.CURVE_TO);

        graphics.lineStyle(0);
        graphics.drawPath(commands, data);
    }
}
}

```

运行之后显示如图 8-1，图中包含了一些注释信息。

图 8-1



**Figure 8-1. Curves drawn with a path**

和旧有的 API 相比，这里仍然没省什么事。但通过以上例子，希望能体会到在复杂的绘图程序中的情形。我们随后会看到更真实的例子。

#### wide 绘制命令和 NO\_OP

假如现在用直线来画之前例子中给出的数据，可能你会想到创建另一个 Vector 以 LINE\_TO 取代 CURVE\_TO，比如：

```

package
{
    import __AS3__.vec.Vector;

    import flash.display.GraphicsPathCommand;

```

```

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

[SWFBackgroundColor=0xffffffff]
public class LineAndCurveDrawing extends Sprite
{
    private var commands:Vector.<int> = new Vector.<int>();
    private var lineCommands:Vector.<int> = new Vector.<int>();
    private var data:Vector.<Number> = new Vector.<Number>();

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

        data.push(200, 200);

        data.push(250, 100);
        data.push(300, 200);

        data.push(400, 250);
        data.push(300, 300);

        data.push(250, 400);
        data.push(200, 300);

        data.push(100, 250);
        data.push(200, 200);

        commands.push(GraphicsPathCommand.MOVE_TO);

        commands.push(GraphicsPathCommand.CURVE_TO);
        commands.push(GraphicsPathCommand.CURVE_TO);
        commands.push(GraphicsPathCommand.CURVE_TO);
        commands.push(GraphicsPathCommand.CURVE_TO);

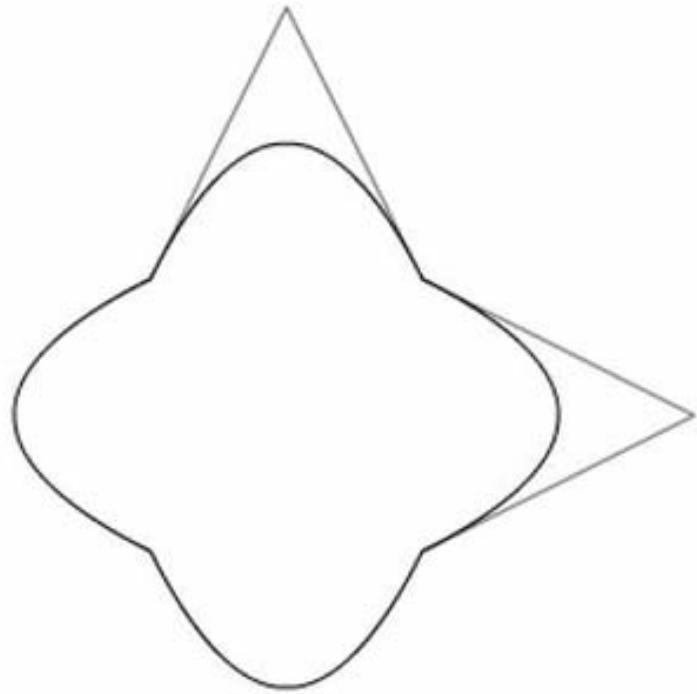
        lineCommands.push(GraphicsPathCommand.MOVE_TO);
        lineCommands.push(GraphicsPathCommand.LINE_TO);
        lineCommands.push(GraphicsPathCommand.LINE_TO);
        lineCommands.push(GraphicsPathCommand.LINE_TO);
        lineCommands.push(GraphicsPathCommand.LINE_TO);

        graphics.lineStyle(0);
        graphics.drawPath(commands, data);

        graphics.lineStyle(0, 0x000000, .5);
        graphics.drawPath(lineCommands, data);
    }
}

```

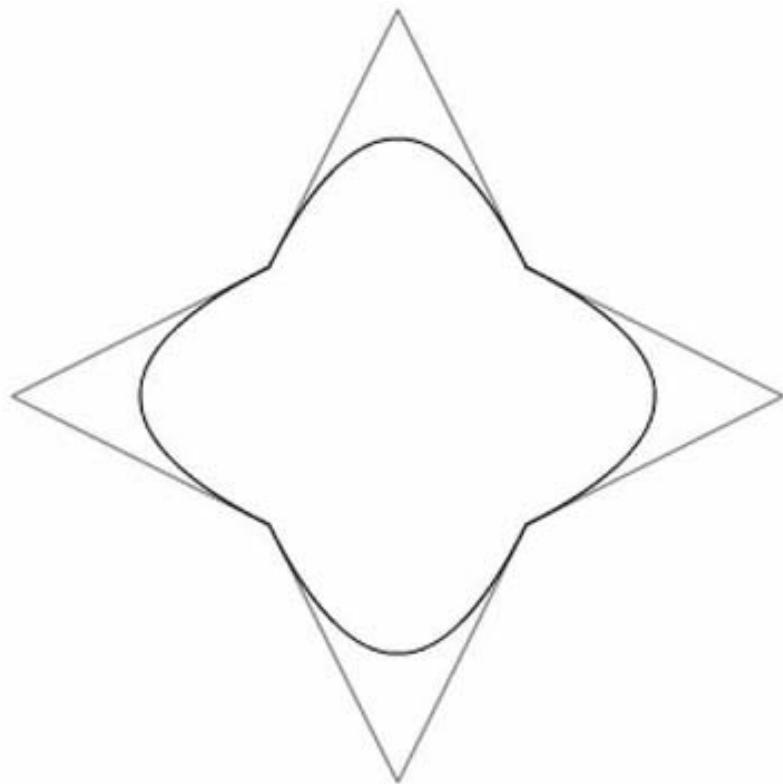
最终结果如图 8-2



**Figure 8-2.** Curves and lines

图中出现的问题，是由于 CURVE\_TO 需要两个点，而 LINE\_TO 只需要一个，但命令 LINE\_TO 只有 4 个，所以只画了一半。再多放 4 个 LINE\_TO 就能画出图 8-3 的图形。

图 8-3



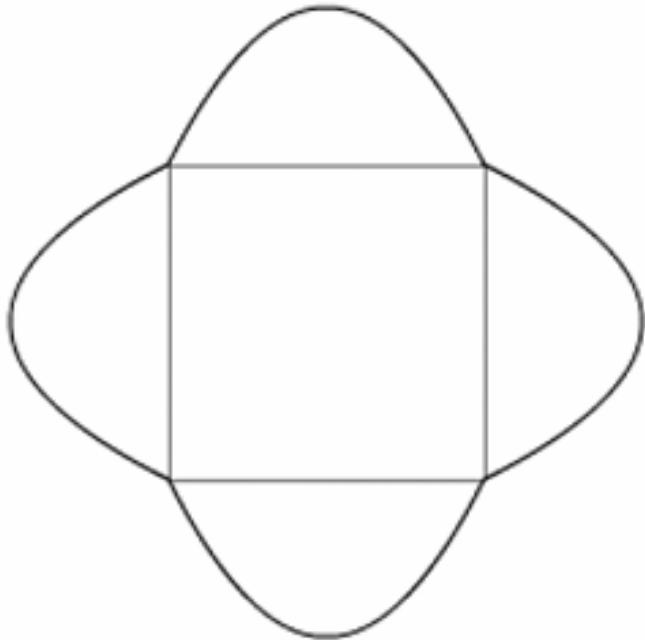
**Figure 8-3.** Curves and more lines

很好。但现在如果要把曲线改成直线，也就是忽略控制点，直接连接终点该怎么办？数据不变的情况下，只用 LINE\_TO 是没法完成的。

于是该 WIDE\_LINE\_TO 出场了。这个命令会跳过下一对数据用直线绘制后面的一对数据。因此，它可以直接跳过 CURVE\_TO 所需的控制点。我们用 WIDE\_LINE\_TO 代替 LINE\_TO 看看结果如何：

```
lineCommands.push(GraphicsPathCommand.MOVE_TO);
lineCommands.push(GraphicsPathCommand.WIDE_LINE_TO);
lineCommands.push(GraphicsPathCommand.WIDE_LINE_TO);
lineCommands.push(GraphicsPathCommand.WIDE_LINE_TO);
lineCommands.push(GraphicsPathCommand.WIDE_LINE_TO);
```

如图 8-4



**Figure 8-4. Curves and wide lines**

可能你觉得采用 NO\_OP 也可以跳过控制点:

```
lineCommands.push(GraphicsPathCommand.MOVE_TO);
lineCommands.push(GraphicsPathCommand.NO_OP);
lineCommands.push(GraphicsPathCommand.LINE_TO);
lineCommands.push(GraphicsPathCommand.NO_OP);
lineCommands.push(GraphicsPathCommand.LINE_TO);
lineCommands.push(GraphicsPathCommand.NO_OP);
lineCommands.push(GraphicsPathCommand.LINE_TO);
lineCommands.push(GraphicsPathCommand.NO_OP);
lineCommands.push(GraphicsPathCommand.LINE_TO);
```

可惜这样不行。

NO\_OP 是一个“不是命令”的命令，它什么也不做。不画直线，也不画曲线，也不会跳过数据。总之，它不会消耗掉数据列表中的数据，所以接下来的 LINE\_TO 还是一如既往的采用接下来的数据。我这里没有一个具体的例子来说明 NO\_OP 的必要性，但能想象到一些可能会用到的情况。

### 缠绕

关于路径，最后要介绍的一样东西叫缠绕。它表示创建图形时点的绘制顺序。众所周知，用直线或曲线画出的图形，可以被看做由通过这些线条之间的点所构成的图形，而这些点不是顺时针排列就是逆时针排列。顺时针的叫正向缠绕，逆时针的叫负向缠绕。当然，8字型的图形同时具有正负向缠绕。这个问题只要把图形拆开看就可以了。

对于一个单一的图形，缠绕没有什么具体意义。但当出现交叉路径或者重叠填充时就不一样了。比如在一次性 beginFill/endFill 时，出现路径交叉，那么对于重叠的部分，填充就有两种情况——填或不填。看下面的例子：

```
package
```

```

{
    import flash.display.GraphicsPathCommand;
    import flash.display.GraphicsPathWinding;

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

    [SWF(backgroundColor=0xffffffff)]

    public class WindingDemo extends Sprite
    {
        private var commands:Vector.<int> = new Vector.<int>();

        private var data1:Vector.<Number> = new Vector.<Number>();
        private var data2:Vector.<Number> = new Vector.<Number>();

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

            commands.push(GraphicsPathCommand.MOVE_TO);

            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);

            data1.push(150, 100);
            data1.push(200, 100);
            data1.push(200, 250);
            data1.push(150, 250);
            data1.push(150, 100);

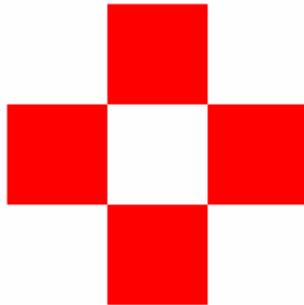
            data2.push(100, 150);
            data2.push(250, 150);
            data2.push(250, 200);
            data2.push(100, 200);
            data2.push(100, 150);

            graphics.beginFill(0xff0000);
            graphics.drawPath(commands, data1);
            graphics.drawPath(commands, data2);
            graphics.endFill();
        }
    }
}

```

这里创建了两个矩形，组成一个十字型。两个路径一次性画完，所以它们有着同样的填充。默认情况下，交叉部分是不填充的，如图 8-5。

图 8-5



当采用同样的填充时，有一个关键点需要注意。看下面的变化：

```
graphics.beginFill(0xff0000);
graphics.drawPath(commands, data1);
graphics.endFill();
graphics.beginFill(0xff0000);
graphics.drawPath(commands, data2);
graphics.endFill();
```

因为在填充完第一个路径后，又全新填充了第二个，所以重叠的区域不受影响。

而只用一个路径也很简单，注意下面的变化：

```
package
{
    import flash.display.GraphicsPathCommand;
    import flash.display.GraphicsPathWinding;

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

    [SWF(backgroundColor=0xffffffff)]

    public class WindingDemo extends Sprite
    {
        private var commands:Vector.<int> = new Vector.<int>();
        private var data1:Vector.<Number> = new Vector.<Number>();
        private var data2:Vector.<Number> = new Vector.<Number>();

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

            commands.push(GraphicsPathCommand.MOVE_TO);

            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
            commands.push(GraphicsPathCommand.LINE_TO);
        }
    }
}
```

```

        commands.push(GraphicsPathCommand.MOVE_TO);

        commands.push(GraphicsPathCommand.LINE_TO);
        commands.push(GraphicsPathCommand.LINE_TO);
        commands.push(GraphicsPathCommand.LINE_TO);
        commands.push(GraphicsPathCommand.LINE_TO);

        data1.push(150, 100);
        data1.push(200, 100);
        data1.push(200, 250);
        data1.push(150, 250);
        data1.push(150, 100);

        data1.push(100, 150);
        data1.push(250, 150);
        data1.push(250, 200);
        data1.push(100, 200);
        data1.push(100, 150);

        graphics.beginFill(0xff0000);
        graphics.drawPath(commands, data1);
        graphics.endFill();
    }
}
}

```

这个结果又和早先的版本一样了。

要填充整个区域，又没法在半途中新开一个 beginFill。于是有了 drawPath 的第三个参数：缠绕。这个参数是一个字符串类型，接收 "evenOdd" 和 "nonZero"。它们是 flash.display.GraphicsPathWinding 类的静态常量：

GraphicsPathWinding.EVEN\_ODD 和 GraphicsPathWinding.NON\_ZERO。

你可以翻一下 Flash CS4 的帮助来了解一下，它们为什么起这个名字以及起名字的背景。简单的来说，EVEN\_ODD 是默认行为，它指在同样的填充下，图形的重叠区域不会被填充。

下面这句代码和之前意义完全一样：

```
graphics.drawPath(commands, data1, GraphicsPathWinding.EVEN_ODD);
```

另一种情况则提供了更多处理交叉图形的控制。

```
graphics.drawPath(commands, data1, GraphicsPathWinding.NON_ZERO);
```

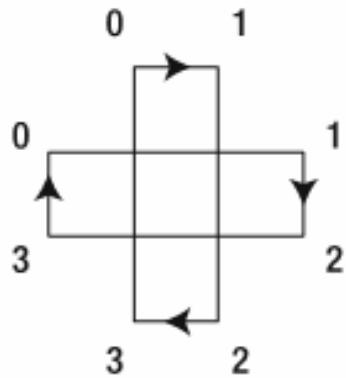
使用上面一句代码，图形就完整了。但这并不是简单的说：“EVEN\_ODD 不填充交叉区域，NON\_ZERO 就填充。”还记得正/负向缠绕吗？当指定缠绕为 NON\_ZERO 时，Flash 会根据重叠路径的缠绕来决定如何处理交叉区域，规则如下：

如果两个路径的缠绕一致（都为正或者负），则填充交叉区域。

如果两个路径的缠绕不一致（一正一负），则不填充交叉区域。

在上面的例子中，两个矩形都是顺时针方向绘制，如图 8-6。

图 8-6



**Figure 8-6. Two clockwise paths**

切换一个矩形中两个点的位置，看看会发生什么：

```

data1.push(150, 100);
data1.push(200, 100);
data1.push(200, 250);
data1.push(150, 250);
data1.push(150, 100);

data1.push(100, 150);
data1.push(100, 200);
data1.push(250, 200);
data1.push(250, 150);
data1.push(100, 150);

```

通过切换第二个矩形的第二、第四个点，把其缠绕方向改为逆时针。测试一下会发现当中区域又空了。同样再切换第一个矩形的第二、第四点，也使其缠绕方向为逆时针，这样填充又有了。

好，drawPath 介绍到此。但路径还会派更大的用处。接下来，我们讲三角。

### 三角

三角在计算机图形学中是一个非常重要的概念。两点确定一条直线，三点确定一个平面。因此，三个点是着色或填充的最小需求。此外，三角也经常作为 3D 模型系统中的基本单位。甚至是曲线或者基本几何图形在渲染时，模型也会被拆分成若干个三角。Flash 10 新增加的绘制三角方法，很可能就是瞄准了 3D 效果。但实际上，它能做更多的事情。

让我们从最简单的入手。先画一个三角形。一个三角形由三个点组成，没错吧？所以，我们先得创建三个点。

```

var vertices:Vector.<Number> = new Vector.<Number>();
vertices.push(100, 100);
vertices.push(200, 100);
vertices.push(150, 200);

```

用 Graphics 的 drawTriangles 函数画出来：

```
graphics.drawTriangles(vertices);
```

别忘了设置线条样式或者填充样式。来看一个简单的测试程序：

代码 Triangles1.as

```

package
{
    import __AS3__.vec.Vector;

    import flash.display.Sprite;
    import flash.display.StageAlign;
}

```

```

import flash.display.StageScaleMode;

[SWF(backgroundColor=0xffffffff)]
public class Triangles1 extends Sprite
{
    public function Triangles1()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;
        var vertices:Vector.<Number> = new Vector.<Number>();
        vertices.push(100, 100);
        vertices.push(200, 100);
        vertices.push(150, 200);

        graphics.lineStyle(0);
        graphics.drawTriangles(vertices);
    }
}

```

结果如图 8-7。

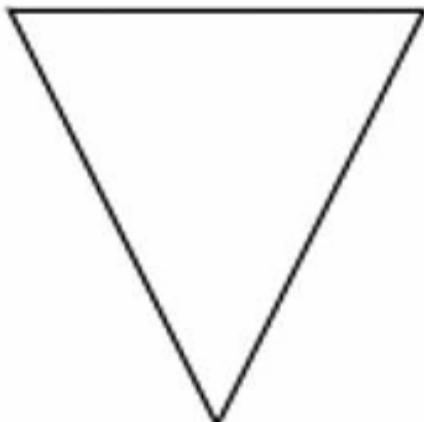


图 8-7

这个看上去比使用 drawPath 简单点。不需要 moveTo 还有 lineTo，只要给定三个点就行了。

接下来，我们画两个三角形。那么，就需要 6 个点，列表中会有 12 个数字。测试程序：

代码 Triangles2.as

```

package
{
    import __AS3__.vec.Vector;

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

    [SWF(backgroundColor=0xffffffff)]
    public class Triangles2 extends Sprite
    {
        public function Triangles2()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            var vertices:Vector.<Number> = new Vector.<Number>();
            vertices.push(100, 100);

```

```

    vertices.push(200, 100);
    vertices.push(150, 200);

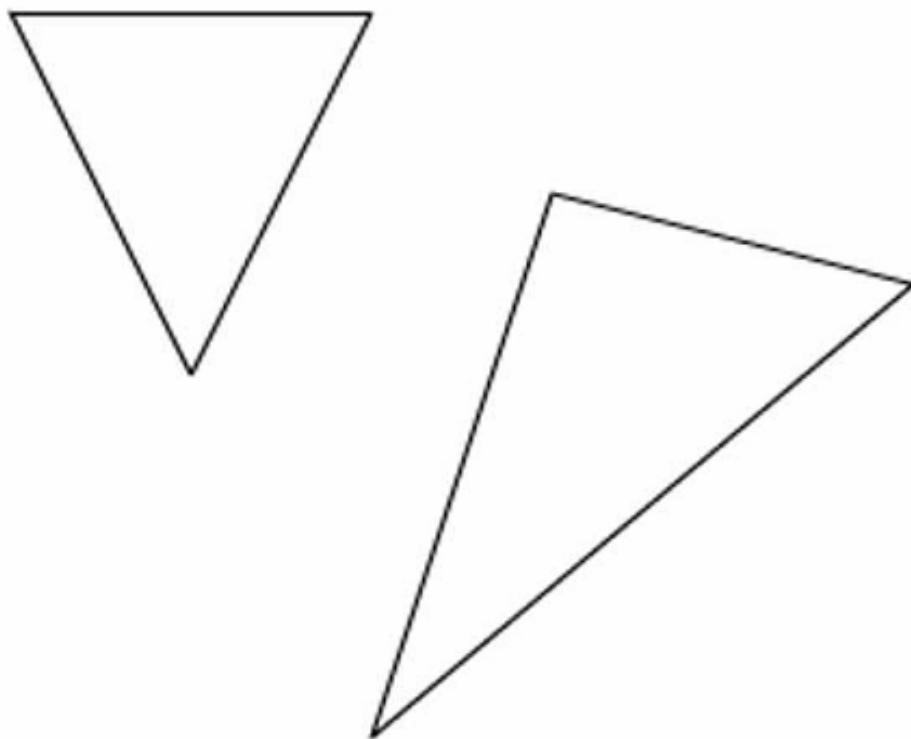
    vertices.push(250, 150);
    vertices.push(350, 175);
    vertices.push(200, 300);

    graphics.lineStyle(0);
    graphics.drawTriangles(vertices);
}
}
}

```

发现只增加了三个点，就画出了如图 8-8 的结果。

图 8-8



**Figure 8-8.** A pair of triangles drawn with the `drawTriangles` method

如此画三角时，`drawTriangles` 接收列表内的元素个数，要正好是 6 的整数倍。试试删掉一行数据会如何：

```

vertices.push(100, 100);
vertices.push(200, 100);
vertices.push(150, 200);
vertices.push(250, 150);
vertices.push(350, 175);

```

只有 5 个点，Flash 不知道如何构造两个三角，于是绘制失败，并发生异常。

到此，我们有两点要考虑：单纯的绘制一堆三角形用处不大；可以只用四个点绘制两个三角形组成一个四边形。好在 `drawTriangles` 有第二个参数，`indices`，它能让我们控制画三角形时所用的顶点。

`indices` 也是一个 Vector (int 型的 Vector)，每一个元素指定了 `vertices` 中的索引。回到第一个例子看一下：

```

var vertices:Vector.<Number> = new Vector.<Number>();
vertices.push(100, 100);
vertices.push(200, 100);
vertices.push(150, 200);
graphics.lineStyle(0);
graphics.drawTriangles(vertices);

```

加入 indices，结果一致：

```

var vertices:Vector.<Number> = new Vector.<Number>();
vertices.push(100, 100);
vertices.push(200, 100);
vertices.push(150, 200);
var indices:Vector.<int> = new Vector.<int>();
indices.push(0,1,2);
graphics.lineStyle(0);
graphics.drawTriangles(vertices, indices);

```

我们告诉 Flash，把索引为 0 的顶点(100, 100)作为三角形的第一个角，索引为 1 的顶点(200, 200)作为第二个角，索引为 2 的顶点(150, 200)作为第三个角。经过这么一串费事，结果保持不变，这只是作为最简单的例子说明一下情况。下面，尝试用 4 个点画两个三角。

代码 Triangles3.as

```

package
{
    import __AS3__.vec.Vector;

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

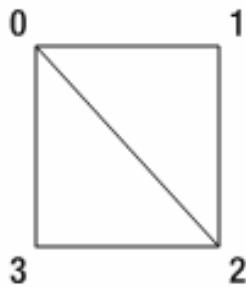
    [SWF(backgroundColor=0xffffffff)]
    public class Triangles3 extends Sprite
    {
        public function Triangles3()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            var vertices:Vector.<Number> = new Vector.<Number>();
            vertices.push(100, 100);
            vertices.push(200, 100);
            vertices.push(200, 200);
            vertices.push(100, 200);

            var indices:Vector.<int> = new Vector.<int>();
            indices.push(0, 1, 2);
            indices.push(2, 3, 0);

            graphics.lineStyle(0);
            graphics.drawTriangles(vertices, indices);
        }
    }
}

```

第一个三角用的是顶点 0, 1, 2, 第二个三角用的是顶点 2, 3, 0。看图 8-9，我在图中作了一些标示。



**Figure 8-9.** A connected pair of triangles

和 drawPath 时一样，我们看不出写一堆点和命令，或者顶点和索引有什么强大的用处。但当图形越来越复杂，保证一个到两个 Vector 传入一个函数，就能实现重绘是非常强大的。而且，三角真正发挥威力是在位图填充时，接下来，我们就开始讲解。

### 位图填充和三角

使用绘画 API 绘图时，总有几种填充方式可供选择：纯色填充，渐变填充，位图填充。下一章还会讲到用 Pixel Bender 进行底纹填充。

这一节，我们介绍在三角形的辅助下被明显强化的位图填充。

首先，三角形可以用任何形式填充，和以往一样：

```
graphics.beginFill(0xFF0000);
graphics.drawTriangles(vertices, indices);
graphics.endFill();
```

此处的填充是完整填充，重叠的部分也会被填充。没法不填充交叉区域，也没有相关参数可以设置。然后，看看位图填充：

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

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

    [SWF(backgroundColor=0xffffffff)]

    public class BitmapTriangles extends Sprite
    {
        [Embed(source="image.jpg")]
        private var ImageClass:Class;

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

            var vertices:Vector.<Number> = new Vector.<Number>();
            vertices.push(100, 100);
            vertices.push(200, 100);
            vertices.push(200, 200);
        }
    }
}
```

```

    vertices.push(100, 200);

    var indices:Vector.<int> = new Vector.<int>();
    indices.push(0, 1, 2);
    indices.push(2, 3, 0);

    var bitmap:Bitmap = new ImageClass() as Bitmap;
    graphics.beginBitmapFill(bitmap.bitmapData);
    graphics.drawTriangles(vertices, indices);
    graphics.endFill();
}
}
}

```

除了导入了一张外部图片，其它部分和之前的例子差不多。

运行测试一下，发现只有一个黑色矩形。如果你用的是自己的图片，可能会有东西，但我用的图片尺寸非常大，而画的两个三角只拼出一个 100x100 的正方形，所以只看到了位图的一点左上角。改一下顶点，再试试：

```

vertices.push(100, 100);
vertices.push(1000, 100);
vertices.push(1000, 600);
vertices.push(100, 200);

```

测试结果如图 8-10。

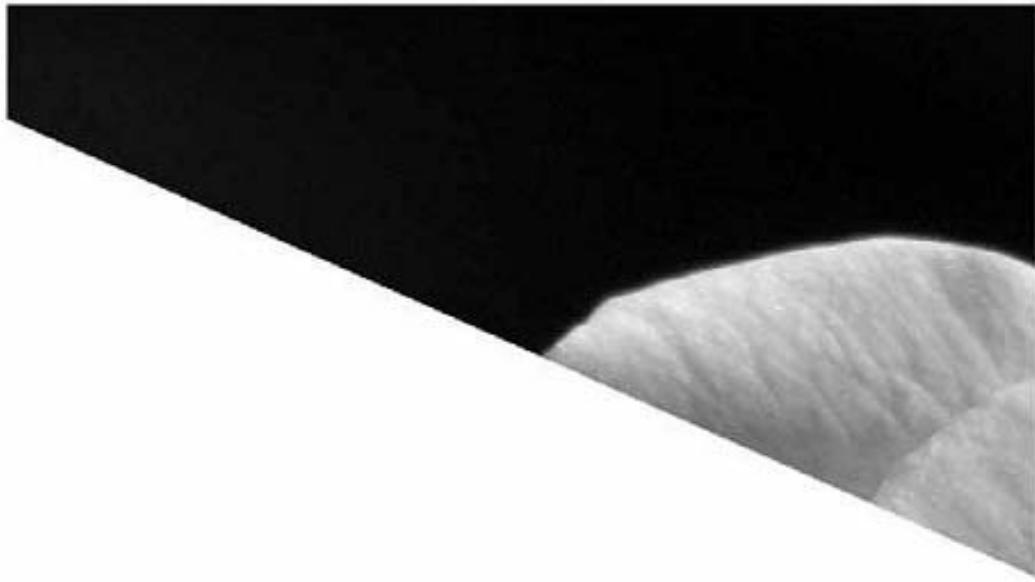


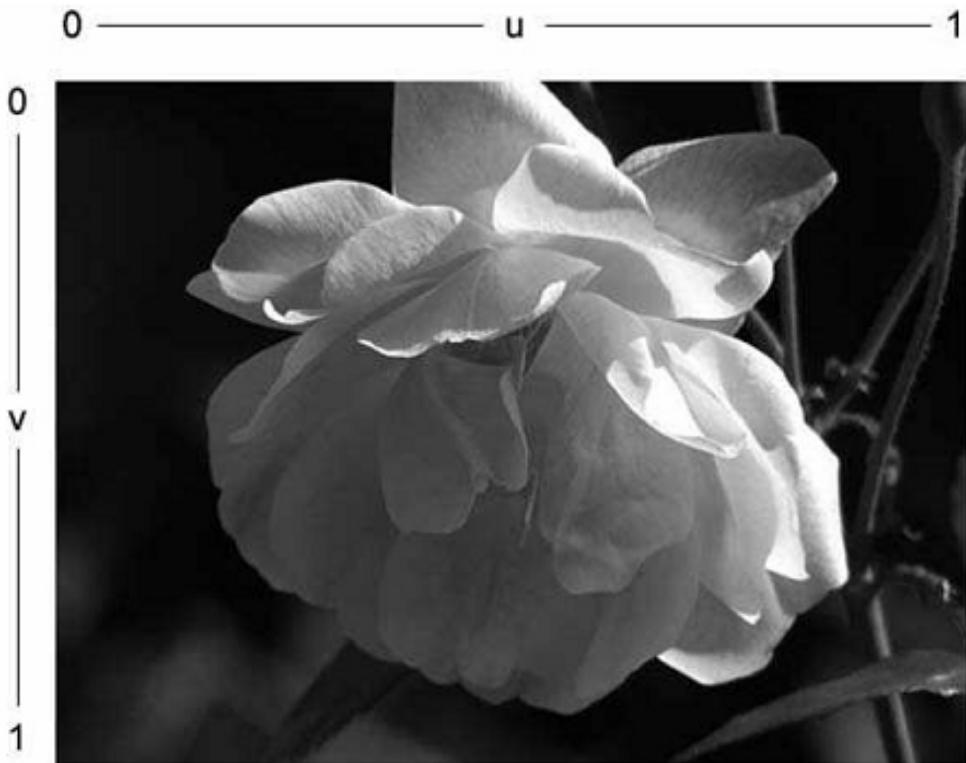
图 8-10 **Figure 8-10.** A bitmap fill with triangles

还可以试试用 matrix 来变形填充，但这很复杂，而且体现不出新式三角填充的优势。所以下面我们介绍新的特性。

#### uvtData

drawTriangles 的第三个可选参数还是一个 Vector (Number 型)，称作 uvtData。uvt 表示了影响位图映射于三角形的三个值。u 和 v 代表 x 轴和 y 轴的比例，t 在用于 3D 时代表缩放。这里先讨论 u 和 v，t 稍后讨论。

如图 8-11，显示了 u 和 v 的映射关系。



**Figure 8-11. uv mapping of a bitmap**

位图的左上角，对应的  $u, v$  是  $0, 0$ ，右上角是  $1, 0$ ，左下角是  $0, 1$ ，右下角是  $1, 1$ 。

下面一个例子中，用两个三角形组成一个矩形，然后让矩形的四个顶点和位图的四个顶点对应。这需要对每个顶点都设置一对  $uv$ 。看测试程序：

代码 BitmapTrianglesUV1.as

```
package
{
    import __AS3__.vec.Vector;

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

    [SWFBackgroundColor=0xffffffff]
    public class BitmapTrianglesUV1 extends Sprite
    {
        [Embed(source="image.jpg")]
        private var ImageClass:Class;

        public function BitmapTrianglesUV1()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            var vertices:Vector.<Number> = new Vector.<Number>();
            vertices.push(100, 100);
            vertices.push(200, 100);
            vertices.push(200, 200);
            vertices.push(100, 200);

            var uvtData:Vector.<Number> = new Vector.<Number>();
```

```

        uvtData.push(0, 0);
        uvtData.push(1, 0);
        uvtData.push(1, 1);
        uvtData.push(0, 1);

        var indices:Vector.<int> = new Vector.<int>();
        indices.push(0, 1, 2);
        indices.push(2, 3, 0);

        var bitmap:Bitmap = new ImageClass() as Bitmap;
        graphics.beginBitmapFill(bitmap.bitmapData);
        graphics.drawTriangles(vertices, indices, uvtData);
        graphics.endFill();
    }
}
}

```

创建的 uvtData 包含了 8 个值，分别对应顶点的 8 个值，也就是位图的左上，右上，右下，左下角。结果如图 8-12。

图 8-12



**Figure 8-12.** A bitmap fill with triangles and uvt data

由于三角所拼成的正方形和图片的比例不对，所以图片看上去被挤压过了。而这也让我们得以实现扭曲图片的效果：

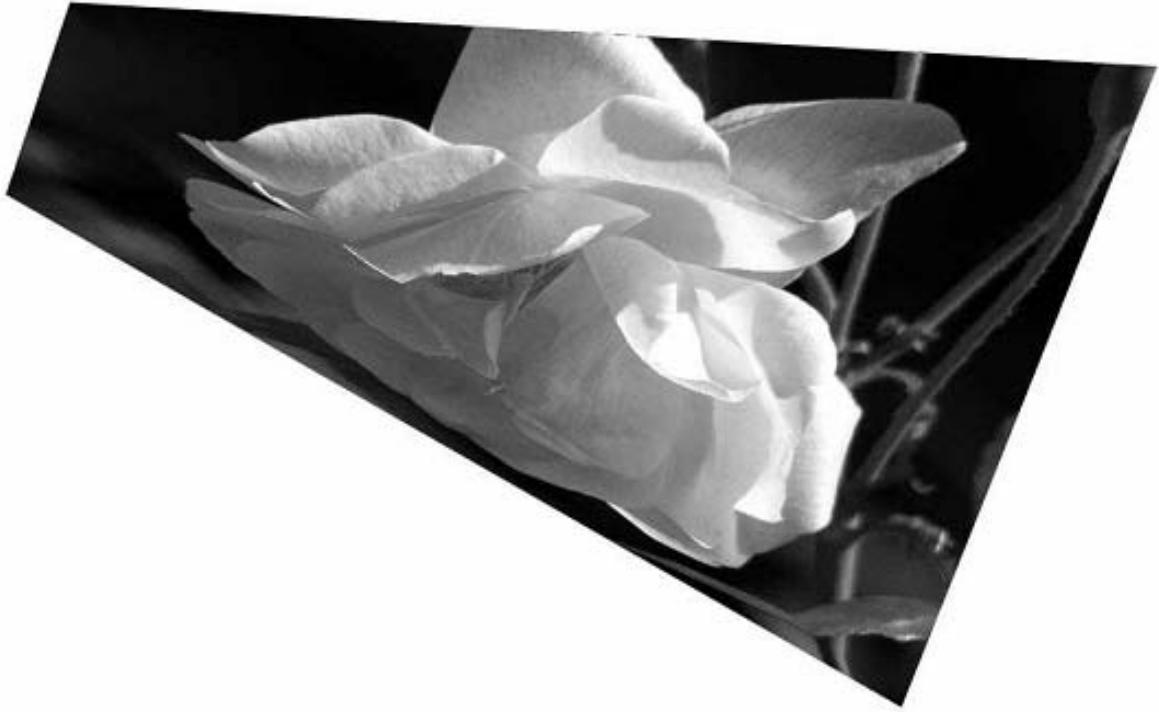
```

vertices.push(150, 50);
vertices.push(1000, 100);
vertices.push(800, 600);
vertices.push(100, 200);

```

改变几个顶点，结果如图 8-13。

图 8-13



**Figure 8-13. Moving the points**

是不是有点 3D 的感觉了？在深入前，让我们再多玩玩 2D 扭曲先。

扩展上面的例子，让顶点成为动态的。不用过多解释，大家都懂，就是增加了拖动顶点的操作而已。

代码 BitmapTrianglesUV2.as

```
package
{
    import __AS3__.vec.Vector;

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

    [SWFBackgroundColor=0xffffffff]
    public class BitmapTrianglesUV2 extends Sprite
    {
        [Embed(source="image.jpg")]
        private var ImageClass:Class;

        private var handle0:Sprite;
        private var handle1:Sprite;
        private var handle2:Sprite;
        private var handle3:Sprite;
        private var vertices:Vector.<Number> = new Vector.<Number>();
        private var uvtData:Vector.<Number> = new Vector.<Number>();
        private var indices:Vector.<int> = new Vector.<int>();
        private var bitmap:Bitmap;

        public function BitmapTrianglesUV2()
        {
```

```

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

handle0 = makeHandle(100, 100);
handle1 = makeHandle(200, 100);
handle2 = makeHandle(200, 200);
handle3 = makeHandle(100, 200);

uvtData.push(0, 0);
uvtData.push(1, 0);
uvtData.push(1, 1);
uvtData.push(0, 1);

indices.push(0, 1, 2);
indices.push(2, 3, 0);

bitmap= new ImageClass() as Bitmap;
draw();
}

private function makeHandle(xpos:Number, ypos:Number):Sprite
{
    var handle:Sprite = new Sprite();
    handle.graphics.beginFill(0);
    handle.graphics.drawCircle(0, 0, 5);
    handle.graphics.endFill();
    handle.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    handle.x = xpos;
    handle.y = ypos;
    addChild(handle);
    return handle;
}

private function onMouseDown(event:MouseEvent):void
{
    event.target.startDrag();
    stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
}

private function onMouseMove(event:MouseEvent):void
{
    draw();
}

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

private function draw():void

```

```

    {
        vertices[0] = handle0.x;
        vertices[1] = handle0.y;
        vertices[2] = handle1.x;
        vertices[3] = handle1.y;
        vertices[4] = handle2.x;
        vertices[5] = handle2.y;
        vertices[6] = handle3.x;
        vertices[7] = handle3.y;

        graphics.clear();
        graphics.beginBitmapFill(bitmap.bitmapData);
        graphics.drawTriangles(vertices, indices, uvtData);
        graphics.endFill();
    }
}
}

```

拖动 4 个顶点看看图片扭曲效果。这种在早期的 Flash 中已经能实现了，但那时需要复杂的设置和深厚的数学功底。现在，可以和数学说 88 了。

### 更多三角

到此，我们只讨论了四个点组成的两个三角形，也就是四边形。如果多增加些三角形，多些点，玩起来就更有趣。对每个顶点以及指定三角形所用的索引，可以不辞辛苦的手动设置，当然也可以通过一些算法来完成。下面一个例子，就是如此，先看一下代码，完了会做一些解释：

代码 BitmapTrianglesUV3.as

```

package
{
    import __AS3__.vec.Vector;

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

    [SWF(backgroundColor=0xffffffff)]
    public class BitmapTrianglesUV3 extends Sprite
    {
        [Embed(source="image.jpg")]
        private var ImageClass:Class;

        private var vertices:Vector.<Number> = new Vector.<Number>();
        private var uvtData:Vector.<Number> = new Vector.<Number>();
        private var indices:Vector.<int> = new Vector.<int>();
        private var bitmap:Bitmap;
        private var res:Number = 100;
        private var cols:int = 5;
        private var rows:int = 4;

        public function BitmapTrianglesUV3()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
        }
    }
}

```

```

bitmap= new ImageClass() as Bitmap;
makeTriangles();

graphics.beginBitmapFill(bitmap.bitmapData);
graphics.drawTriangles(vertices, indices, uvtData);
graphics.endFill();

graphics.lineStyle(0);
graphics.drawTriangles(vertices, indices);
}

private function makeTriangles():void
{
    for(var i:int = 0; i < rows; i++)
    {
        for(var j:int = 0; j < cols; j++)
        {
            vertices.push(j * res, i * res);
            uvtData.push(j / (cols - 1), i / (rows - 1));

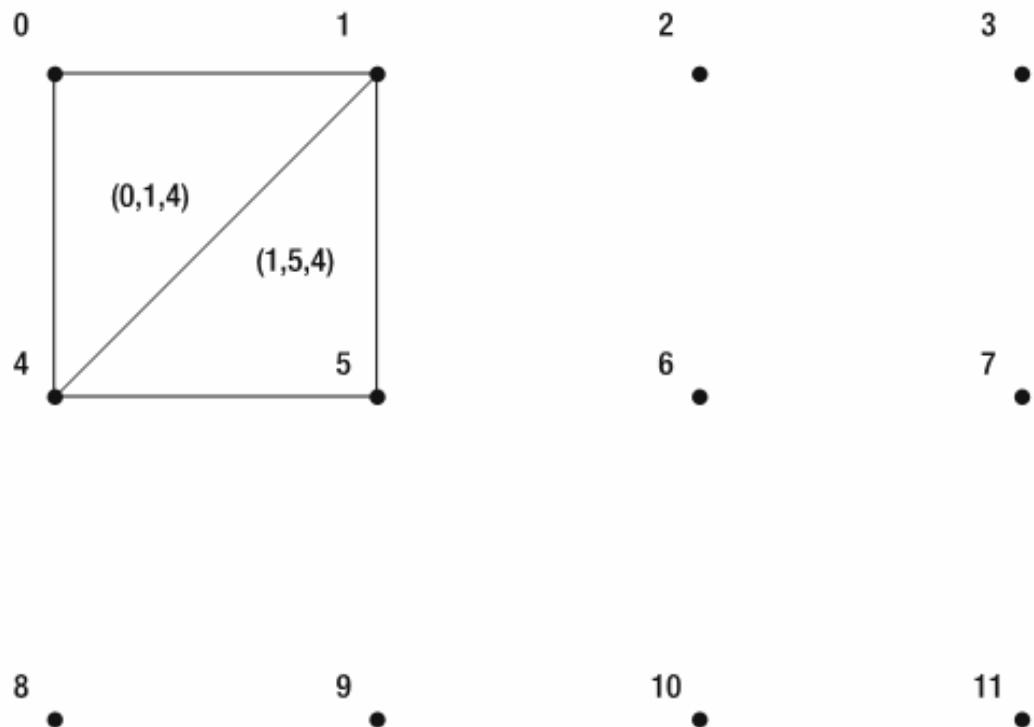
            if(i < rows - 1 && j < cols - 1)
            {
                // first triangle
                indices.push(i * cols + j, i * cols + j + 1, (i + 1) * cols + j);

                // second triangle
                indices.push(i * cols + j + 1, (i + 1) * cols + j + 1, (i + 1) * cols + j);
            }
        }
    }
}
}

```

一上来设置三个新的变量：res（分辨率）决定三角的尺寸，rows 和 cols 决定顶点的行列数。在 makeTriangles 函数中，遍历行列数，生成一堆顶点。每个顶点的坐标关系是  $j*res$ ,  $i*res$ , uvtData 是  $j/(cols-1)$ ,  $i/(rows-1)$ 。保证取值范围在 0.0 到 1.0 之间。

在循环内部，是不用考虑最后一行一列的。因为一个三角形是通过当前点，当前右侧一点和当前下方一点组成，对称的另一个三角形是当前右侧一点，右下方一点和下方一点组成。如图 8-14  
图 8-14



**Figure 8-14.** Moving the points

在这个简单的例子中，对三角形我不仅用了位图填充，还画出了边框，以便知道它们是如何排列的。试着改变 res, rows, cols 看看布局和最终图片尺寸的变化。

很好，但由于我们仅限于画矩形，所以还没有真正看到三角形的威力。当开始有意识的移动这些顶点，我们就能创建出很多种效果，包括 3D！

### 三角和 3D

先说一下使用三角实现 3D 的最基本策略：

1. 创建一堆三角结构顶点和索引。
2. 计算于每个顶点相符的 3D 坐标点位置。
3. 使用透视法计算 3D 坐标点在屏幕上的坐标，作为顶点的值。
4. 使用 drawTriangles 传入顶点和索引。

这种 3D 的实现和第七章讨论的 DisplayObject3D 有很大的不同。实际上，可以看成 Flash 10 拥有两种不同的 3D 引擎。应用透视法的 3D 计算在上本书中有详细介绍。回顾一下基本公式：

$$\text{perspectiveScale} = \text{focalLength} / (\text{focalLength} + \text{zPosition})$$

一个含有 x, y, z 的坐标点，z 坐标用于求出缩放比例。然后用这个比例乘以 x, y 就能得到点在屏幕上的位置，这个位置也就是顶点的值。

下面一个类，改自之前的一个例子，只不过现在顶点以圆柱排列。填充暂时放一放，先用线条画出结构：

```

package
{
    import flash.display.Bitmap;

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

    [SWF(backgroundColor=0xffffffff)]

    public class ImageTube extends Sprite

```

```

{
    [Embed(source="image.jpg")]
    private var ImageClass:Class;

    private var vertices:Vector.<Number> = new Vector.<Number>();
    private var indices:Vector.<int> = new Vector.<int>();
    private var uvtData:Vector.<Number> = new Vector.<Number>();
    private var bitmap:Bitmap;
    private var sprite:Sprite;
    private var res:Number = 60;
    private var cols:int = 20;
    private var rows:int = 6;
    private var centerZ:int = 200;
    private var focalLength:Number = 250;
    private var radius:Number = 200;

    public function ImageTube()

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

        sprite = new Sprite();
        sprite.x = 400;
        sprite.y = 400;
        addChild(sprite);

        bitmap= new ImageClass() as Bitmap;
        makeTriangles();
        draw();
    }

    private function draw():void
    {
        sprite.graphics.lineStyle(0, 0, .5);
        sprite.graphics.drawTriangles(vertices, indices);
    }

    private function makeTriangles():void
    {
        for(var i:int = 0; i < rows; i++)
        {
            for(var j:int = 0; j < cols; j++)
            {
                var angle:Number = Math.PI * 2 / (cols - 1) * j;

                var xpos:Number = Math.cos(angle) * radius;
                var ypos:Number = (i - rows / 2) * res;
                var zpos:Number = Math.sin(angle) * radius;

                var scale:Number = focalLength / (focalLength + zpos +centerZ);

                vertices.push(xpos * scale, ypos * scale);
            }
        }
    }
}

```

```
    uvtData.push(j / (cols - 1), i / (rows - 1));
    uvtData.push(scale);

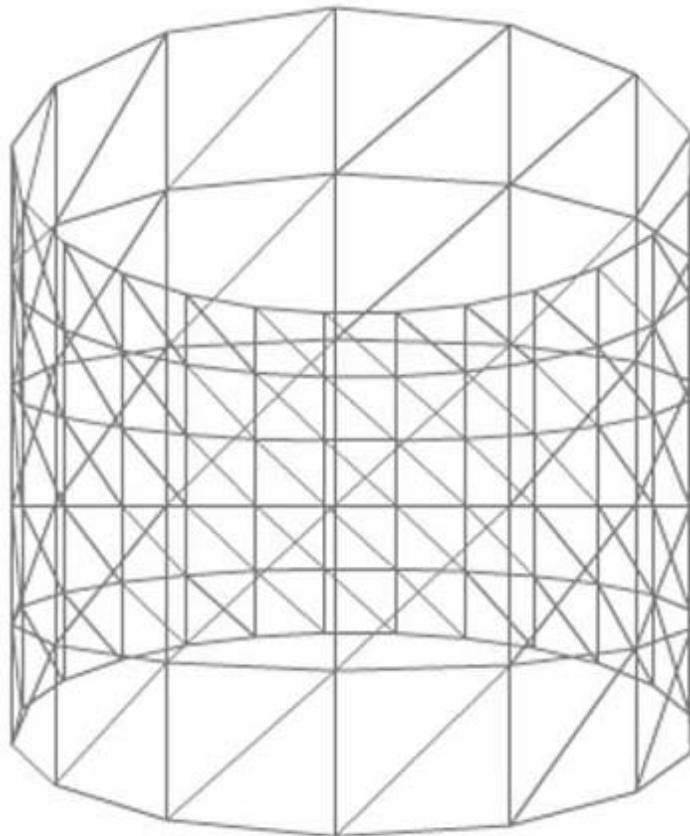
    if(i < rows - 1 && j < cols - 1)
    {
        indices.push(i * cols + j,
                      i * cols + j + 1,
                      (i + 1) * cols + j);

        indices.push(i * cols + j + 1,
                      (i + 1) * cols + j + 1,
                      (i + 1) * cols + j);
    }
}
}
```

首先，变量 radius(半径)是指圆柱的半径。然后在场景上增加了一个 sprite 并移动至 400, 400 (因为在 3D 计算中，我们需要一个投影点)。我们也可以设置一个 400, 400 的投影点，然后加于每个顶点上，但这要比把投影点认为 0, 0，然后把对象都加入一个 sprite，再移动 sprite 来的麻烦的多。

我们把真正的绘制代码转移到了另一个函数中。

最后，在 makeTriangles 函数里，循环的开始部分是圆柱公式算出 x, y, z，然后通过透视法算出顶点。运行一下如图 8-15。



**Figure 8-15.** A wire frame tube

所有的三角形通过 3D 扭曲构成了一个圆柱体。看似不错！现在是不是只要填充位图就好了？嗯...是，也不是。还有些事要做，不管如何先试试看吧。修改 draw 函数如下：

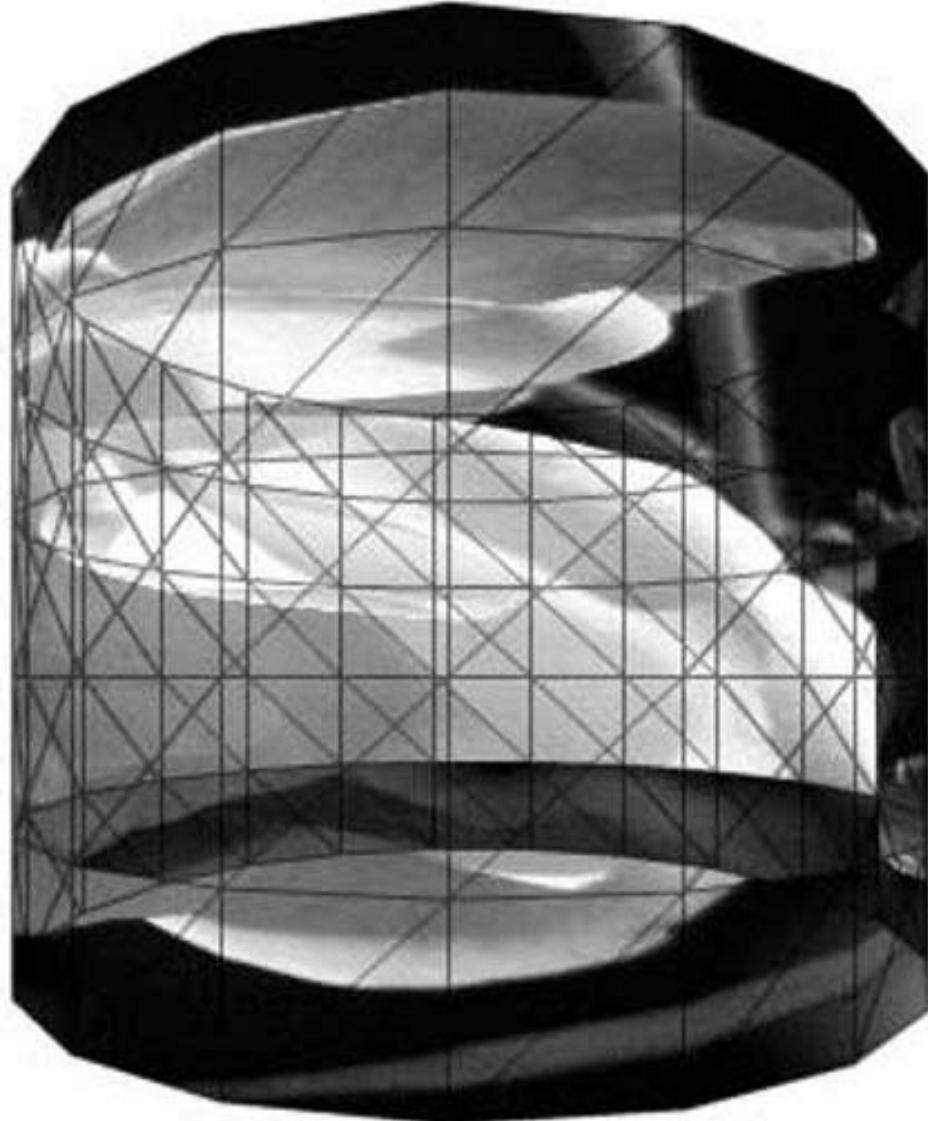
```

private function draw():void
{
    sprite.graphics.beginBitmapFill(bitmap.bitmapData);
    sprite.graphics.drawTriangles(vertices, indices, uvtData);
    sprite.graphics.endFill();

    sprite.graphics.lineStyle(0, 0, .5);
    sprite.graphics.drawTriangles(vertices, indices);
}

```

额...似乎不太对 (图 8-16)



**Figure 8-16.** Something is wrong here.

发生了什么——有些三角形到了前面，有些到了后面。如果阅读了第七章，你应该举手喊出“层深排序！”从某这种意义上说这是对的，但这个概念在三角绘制时略有不同：背面剔除（这在上本书中有涉及，并且是全手工完成）它的基本含义是指，如果绘制所有以顺时针方向组织的三角，就不绘制任何逆时针方向组织的。

反之亦然。这取决于如何创建三角。不管发生什么，如果绘制和创建相一致，那么三角始终是面向我们的，否则就是背对我们。背对我们的叫背面，是不用去绘制的。

幸运的是，Flash 已经内置了该功能。drawTriangles 的第四个可选参数 culling，它是一个字符串，接收“positive”，“negative”，“none”三个值。flash.display.TriangleCulling 定义了

这些值：

```
TriangleCulling.POSITIVE  
TriangleCulling.NEGATIVE  
TriangleCulling.NONE
```

关于更多技术性介绍可以查阅 Flash 帮助文档，这里只做简单介绍：positive 剔除只绘制逆时针的三角，negative 剔除只绘制顺时针的。如果设置剔除为 none，就都绘制。由于我们设置的三角都是顺时针的，所以要用 negative 剔除：

```
private function draw():void  
{  
    sprite.graphics.clear();  
    sprite.graphics.beginBitmapFill(bitmap.bitmapData);  
    sprite.graphics.drawTriangles(vertices, indices, uvtData,  
        TriangleCulling.NEGATIVE);  
    sprite.graphics.endFill();  
  
    sprite.graphics.lineStyle(0, 0, .5);  
    sprite.graphics.drawTriangles(vertices, indices, uvtData,  
        TriangleCulling.NEGATIVE);  
}
```

现在运行一下，结果应该如图 8-17。

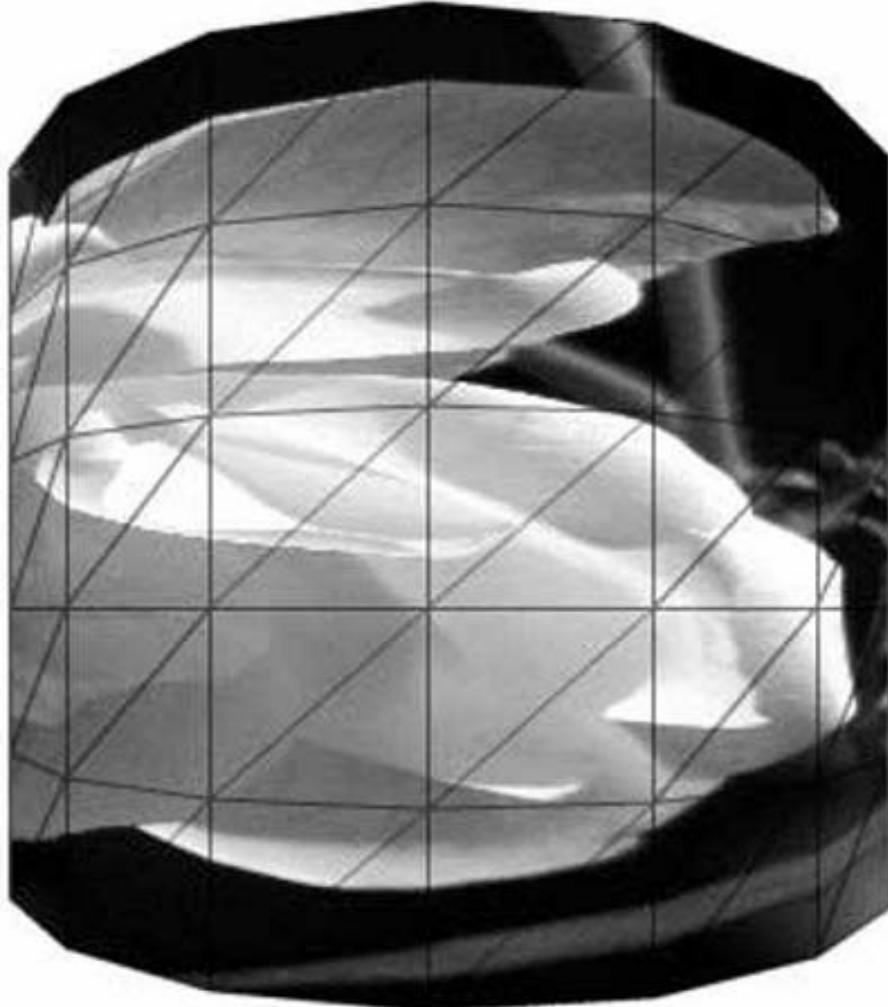


Figure 8-17. Backface culling solved the problem.

看上去像是一个裹了一张图片的罐头。你也可以去掉第二段画线部分的 drawTriangles。  
uvt 中的 t

突然想起一个东西：t，它特别用在 3D 绘制时修正位图的缩放。在前一个例子中，我们计算缩放是基于每个点的 z 坐标，然后使用这个缩放值调整映射于屏幕上的 x, y 坐标，最后以此创建用于绘制三角形的顶点。这个缩放值同样要作为每个顶点所对应的 t。这时，它表示任意两点间像素的缩放间隔。打开你的想象空间，设想一道篱笆，其栅栏间距相等，那么距离越远的栅栏凑的越近，越靠前的栅栏分得越开。t 的意义就在于此，只不过对应的东西是像素而不是栅栏。没有 t，点与点之间是相等的，结果就会是图像看似被扭曲了。

t 作为是 u, v 后的一个值加入 uvtData 中，好像这样：

```
var scale:Number = focalLength / (focalLength + zpos + centerZ);
vertices.push(xpos * scale, ypos * scale);
uvtData.push(j / (cos - 1), i / (rows - 1));
uvtData.push(scale);
```

你可能正在想，这样不是搞乱了 uvtData 嘛，之前是两个值为一组，现在是三个值为一组。好在，drawTriangles 能聪明的分辨出来。如果 uvtData 和 vertices 长度一样，就认为只含有 u, v，如果 uvtData 的长度是 vertices 长度的 1.5 倍，就认为是含有 t 的。

回过头加入上述代码测试一下，可能并没有发现多大的不同，但是把两种情况并排比较的话，还是会发现在位图的拉伸上有一些微妙的变化。虽然在这个例子中引不起注意，但仍应作为一个好习惯而用之，因为它能减小 3D 结构在真实性上的差异，尤其是透视特别厉害的地方。

### 旋转圆柱

由于这本书的标题包含“动画”一词，所以现在就开始让圆柱动起来。至少能让它转动。

增加一个偏移量，让所有顶点和 uvt 在绘制和更新时，都加上这个偏移量。这就使圆柱能够转起来。程序如下：

代码 ImageTube.as

```
package
{
    import __AS3__.vec.Vector;

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

    [SWFBackgroundColor=0xffffffff]
    public class ImageTube extends Sprite
    {
        [Embed(source="image.jpg")]
        private var ImageClass:Class;

        private var vertices:Vector.<Number> = new Vector.<Number>();
        private var indices:Vector.<int> = new Vector.<int>();
        private var uvtData:Vector.<Number> = new Vector.<Number>();
        private var bitmap:Bitmap;
        private var sprite:Sprite;
        private var res:Number = 60;
        private var cols:int = 20;
        private var rows:int = 6;
```

```

private var centerZ:int = 200;
private var focalLength:Number = 250;
private var radius:Number = 200;
private var offset:Number = 0;

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

    sprite = new Sprite();
    sprite.x = 400;
    sprite.y = 400;
    addChild(sprite);

    bitmap= new ImageClass() as Bitmap;
    makeTriangles();
    draw();
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    draw();
}

private function draw():void
{
    offset += .05;
    vertices.length = 0;
    uvtData.length = 0;

    for(var i:int = 0; i < rows; i++)
    {
        for(var j:int = 0; j < cols; j++)
        {
            var angle:Number = Math.PI * 2 / (cols - 1) * j + offset;

            var xpos:Number = Math.cos(angle) * radius;
            var ypos:Number = (i - rows / 2) * res;
            var zpos:Number = Math.sin(angle) * radius;

            var scale:Number = focalLength / (focalLength + zpos + centerZ);

            vertices.push(xpos * scale, ypos * scale);
            uvtData.push(j / (cols - 1), i / (rows - 1));
            uvtData.push(scale);
        }
    }

    sprite.graphics.clear();
    sprite.graphics.beginBitmapFill(bitmap.bitmapData);
    sprite.graphics.drawTriangles(vertices, indices, uvtData, TriangleCulling.NEGATIVE);
}

```

```

        sprite.graphics.endFill();

        sprite.graphics.lineStyle(0, 0, .5);
        sprite.graphics.drawTriangles(vertices, indices, uvtData, TriangleCulling.NEGATIVE);
    }

private function makeTriangles():void
{
    for(var i:int = 0; i < rows; i++)
    {
        for(var j:int = 0; j < cols; j++)
        {
            if(i < rows - 1 && j < cols - 1)
            {
                indices.push(i * cols + j,
                            i * cols + j + 1,
                            (i + 1) * cols + j);

                indices.push(i * cols + j + 1,
                            (i + 1) * cols + j + 1,
                            (i + 1) * cols + j);
            }
        }
    }
}

```

注意到 makeTriangles 函数只负责创建索引。因为索引不会改变，所以只需创建一次。而顶点和 uvt 上面已经说过了，需要在 draw 函数中重新计算。

当我们有了圆柱体，把它变成一个地球就不是那么难了。让我们动手一试吧。

### 创建一个 3D 地球

这是三角在 3D 中运用的最后一站，创建一个转动的地球。我淘到了一张很合适的世界地图，当然你可以用任何图片作为球的皮肤。大部分代码和圆柱体一样，我们只须修改顶点的位置，让顶部和底部连接在一起组成一个圆形即可。经过一些三角函数和一大堆错误的洗礼之后，结果在此：

代码 ImageSphere.as

```

package
{
    import __AS3__.vec.Vector;

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

    [SWF(backgroundColor=0x000000, width=800, height=800)]
    public class ImageSphere extends Sprite
    {
        [Embed(source="map.jpg")]
        private var ImageClass:Class;

```

```

private var vertices:Vector.<Number> = new Vector.<Number>();
private var indices:Vector.<int> = new Vector.<int>();
private var uvtData:Vector.<Number> = new Vector.<Number>();
private var bitmap:Bitmap;
private var sprite:Sprite;
private var centerZ:int = 500;
private var cols:int = 20;
private var rows:int = 20;
private var focalLength:Number = 1000;
private var radius:Number = 400;
private var offset:Number = 0;

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

    sprite = new Sprite();
    sprite.x = 400;
    sprite.y = 400;
    addChild(sprite);

    bitmap= new ImageClass() as Bitmap;
    makeTriangles();
    draw();
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    draw();
}

private function draw():void
{
    offset -= .02;
    vertices.length = 0;
    uvtData.length = 0;

    for(var i:int = 0; i < rows; i++)
    {
        for(var j:int = 0; j < cols; j++)
        {
            var angle:Number = Math.PI * 2 / (cols - 1) * j;
            var angle2:Number = Math.PI * i / (rows - 1) - Math.PI / 2;

            var xpos:Number = Math.cos(angle + offset) * radius * Math.cos(angle2);
            var ypos:Number = Math.sin(angle2) * radius;
            var zpos:Number = Math.sin(angle + offset) * radius * Math.cos(angle2);

            var scale:Number = focalLength / (focalLength + zpos + centerZ);

```

```

        vertices.push(xpos * scale,
                      ypos * scale);
        uvtData.push(j / (cols - 1), i / (rows - 1));
        uvtData.push(scale);
    }
}

sprite.graphics.clear();
sprite.graphics.beginBitmapFill(bitmap.bitmapData);
sprite.graphics.drawTriangles(vertices, indices, uvtData, TriangleCulling.NEGATIVE);
sprite.graphics.endFill();

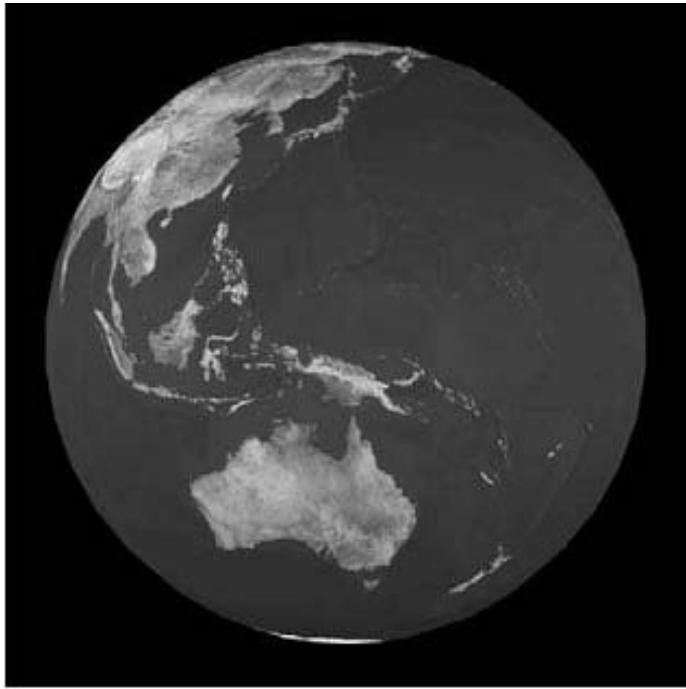
// sprite.graphics.lineStyle(0, 0, .4);
// sprite.graphics.drawTriangles(vertices, indices, uvtData, TriangleCulling.NEGATIVE);
}

private function makeTriangles():void
{
    for(var i:int = 0; i < rows; i++)
    {
        for(var j:int = 0; j < cols; j++)
        {
            if(i < rows - 1 && j < cols - 1)
            {
                indices.push(i * cols + j,
                             i * cols + j + 1,
                             (i + 1) * cols + j);

                indices.push(i * cols + j + 1,
                             (i + 1) * cols + j + 1,
                             (i + 1) * cols + j);
            }
        }
    }
}
}

```

正如您所想，确实，这在实现上除了图片的不同和一些讨厌的公式以外，和圆柱没多大不同嘛。结果如图 8-18



**Figure 8-18. A rotating globe**

就这也有够多东西等着你玩了。而这不过是冰山一角。当然，和 PaperVision3D 比还是逊色不少。但我确信，新的 API 将会找到它自己的出路，并出没于各种 3D 的引擎中。因为它是播放器的一部分，比起编译过的 ActionScript 能更有效的进行底层操作。

该是我们离开 3D，讨论图形数据的时候了。

## 图形数据

到此，我们介绍了绘制路径和绘制三角，这两样都是非常强有力的新内容。但不管信不信，它们只是强大 API 的一部分。

回忆一下，在画线或是填充三角时，指定线条或填充样式仍然采用的是以前的老办法 `lineStyle` 和 `beginFill`。通过 `drawGraphicsData`，则可以设定路径，三角，线条以及各种填充方式。

使用起来很简单：传一个 `Vector` 给 `drawGraphicsData` 函数，这个 `Vector` 包含了所有实现 `IGraphicsData` 接口的对象。

意思也就是，虽然对象类型不同，但都有 `IGraphicsData` 接口。比如以下这些类：

```
GraphicsStroke  
GraphicsSolidFill  
GraphicsBitmapFill  
GraphicsGradientFill  
GraphicsShaderFill  
GraphicsEndFill  
GraphicsPath  
GraphicsTrianglePath
```

这些类都在 `flash.display` 包里。这里不对每个作详细解释，只稍微介绍一下，然后凭各自的想法去使用它们。先从 `GraphicsStroke` 和 `GraphicsPath` 开始，这样至少能画出一条或几条直线。看看 `GraphicsStroke` 的构造函数：

```
GraphicsStroke(thickness:Number = NaN, pixelHinting:Boolean = false,  
               scaleMode:String = "normal", caps:String = "none",  
               joints:String = "round", miterLimit:Number = 3.0,  
               fill:IGraphicsFill = null)
```

Graphics.lineStyle 中有的参数，这个构造函数中都有。这些参数还都是类的公有属性，所以可以先创建一个缺省参数的类，再根据需要设置几个属性，比如：

```
var stroke:GraphicsStroke = new GraphicsStroke();
stroke.thickness = 1;
```

属性 fill（填充）可能会有点混淆。一般情况下，画一个带线条的图形，填充的概念是指，用纯色、渐变或者位图来填线框内的区域。但不要忘了，如今的线条除了颜色和透明度外，也可以使用渐变。所以 fill 是一个实现了 IGraphicsFill 接口的对象，而实现这个接口的类有：

```
GraphicsSolidFill
GraphicsGradientFill
GraphicsBitmapFill
GraphicsShaderFill
```

这里我们先用一下纯色填充。GraphicsSolidFill 的构造函数是这样的：

```
GraphicsSolidFill(color:uint = 0, alpha:Number = 1.0)
```

这些参数也同样有公有属性。然后创建一个 5 像素粗细的红色线条：

```
var stroke:GraphicsStroke = new GraphicsStroke(5);
stroke.fill = new GraphicsSolidFill(0xff0000);
```

再创建一个 GraphicsPath 对象。构造函数如下：

```
GraphicsPath(commands:Vector.<int> = null,
             data:Vector.<Number> = null,
             winding:String = "evenOdd")
```

路径我们已经了解了，用起来就非常顺手：

```
var commands:Vector.<int> = new Vector.<int>();
commands.push(GraphicsPathCommand.MOVE_TO);
commands.push(GraphicsPathCommand.LINE_TO);

var data:Vector.<Number> = new Vector.<Number>();
data.push(100, 100);
data.push(200, 200);

var path:GraphicsPath = new GraphicsPath(commands, data);
```

好，我们有了线条和路径，只要把它们放入一个 IGraphicsData 的 Vector 里，然后调用 drawGraphicsData 就会出现一条直线。来吧，测试一下：

代码 GraphicsDataDemo1.as

```
package
{
    import __AS3__.vec.Vector;
    import flash.display.GraphicsPath;
```

```

import flash.display.GraphicsPathCommand;
import flash.display.GraphicsSolidFill;
import flash.display.GraphicsStroke;
import flash.display.IGraphicsData;
import flash.display.Sprite;
import flash.display.StageAlign;
import flash.display.StageScaleMode;

[SWF(backgroundColor=0xffffffff)]
public class GraphicsDataDemo1 extends Sprite
{
    public function GraphicsDataDemo1()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        var graphicsData:Vector.<IGraphicsData> = new Vector.<IGraphicsData>();
        var stroke:GraphicsStroke = new GraphicsStroke(5);
        stroke.fill = new GraphicsSolidFill(0xff0000);

        var commands:Vector.<int> = new Vector.<int>();
        commands.push(GraphicsPathCommand.MOVE_TO);
        commands.push(GraphicsPathCommand.LINE_TO);

        var data:Vector.<Number> = new Vector.<Number>();
        data.push(100, 100);
        data.push(200, 200);

        var path:GraphicsPath = new GraphicsPath(commands, data);

        graphicsData.push(stroke);
        graphicsData.push(path);

        graphics.drawGraphicsData(graphicsData);
    }
}
}

```

只有代码的最后三行是我们以前没见过的。运行一下，在场景上看到了一条红色的直线。我靠，用了 13 行完成了一件以前用 3 行就能做的事情！

你懂我将要说：这更适用于复杂的绘制。先前我提到过命令模式，是一个封装了行为和参数的对象。如今更是明显，通过创建一些命令来代替大段的绘制函数。因为这些命令都是对象，所以在列表中增加、删除、排列、变更它们都很容易，而且也可以从任意一点作为起点来进行绘制。让我们看看实际的用途。

作为本章的最后一个例子，我们要做一些之前类似的事情，并且加入历史功能。新的策略在此：

1. 当每次鼠标按下，随机选择线条样式（粗细和颜色），放入缓冲。
2. 当鼠标移动，把 LINE\_TO 和坐标点加入路径。
3. 当鼠标放开，创建一个 GraphicsPath 对象。把这个对象放进缓冲。
4. 在 draw 函数中，把缓冲复制出来，并使用 drawGraphicsData 绘制。

使用缓冲就有了历史功能。按键盘的左右键能够后退和前进，函数 draw 会复制对应的缓冲内

容。假如画了五条线，按左键就会后退一步。这时的缓冲里面只有画之前四条线的命令，所以就只画了前面四条线。按右前进一步，五条线又都出来了。代码如下：

```
代码 HistoryDraw.as
package
{
    import __AS3__.vec.Vector;

    import flash.display.GraphicsPath;
    import flash.display.GraphicsPathCommand;
    import flash.display.GraphicsSolidFill;
    import flash.display.GraphicsStroke;
    import flash.display.IGraphicsData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.KeyboardEvent;
    import flash.events.MouseEvent;
    import flash.ui.Keyboard;

    [SWF(backgroundColor=0xffffffff)]
    public class HistoryDraw extends Sprite
    {
        private var graphicsData:Vector.<IGraphicsData>;
        private var graphicsDataBuffer:Vector.<IGraphicsData>;
        private var commands:Vector.<int>;
        private var data:Vector.<Number>;
        private var index:int = 0;

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

            graphicsData = new Vector.<IGraphicsData>();
            graphicsDataBuffer = new Vector.<IGraphicsData>();

            stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
            stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
        }

        private function onMouseDown(event:MouseEvent):void
        {
            // create a random stroke, add it to buffer
            var stroke:GraphicsStroke = new GraphicsStroke();
            stroke.thickness = Math.random() * 10;
            stroke.fill = new GraphicsSolidFill(Math.random() * 0xffffffff);
            graphicsDataBuffer.push(stroke);

            // increment index and make this the last command in the buffer
            index++;
            graphicsDataBuffer.length = index;

            // push a moveTo onto the commands
        }
    }
}
```

```

commands = new Vector.<int>();
commands.push(GraphicsPathCommand.MOVE_TO);

// move to this point
data = new Vector.<Number>();
data.push(mouseX, mouseY);

// draw a gray line for now
graphics.lineStyle(0, 0, .5);
graphics.moveTo(mouseX, mouseY);

stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
}

private function onMouseMove(event:MouseEvent):void
{
    // push a lineto onto the commands, and current point to data
    commands.push(GraphicsPathCommand.LINE_TO);
    data.push(mouseX, mouseY);

    // draw temporary line
    graphics.lineTo(mouseX, mouseY);
}

private function onMouseUp(event:MouseEvent):void
{
    // this line is done. push path onto buffer
    graphicsDataBuffer.push(new GraphicsPath(commands, data));

    // increment index and make this last command in buffer
    index++;
    graphicsDataBuffer.length = index;

    stage.removeEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
    stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    draw();
}

private function onKeyUp(event:KeyboardEvent):void
{
    // go back two commands (path and stroke)
    if(event.keyCode == Keyboard.LEFT)
    {
        index -= 2;
    }
    // go forward two commands (stroke and path)
    else if(event.keyCode == Keyboard.RIGHT)
    {
        index += 2;
    }
    // limit index to sensible range
    index = Math.max(0, index);
}

```

```

index = Math.min(graphicsDataBuffer.length, index);
draw();
}

private function draw():void
{
    // clear graphics data Vector
    graphicsData.length = 0;

    // copy over data from buffer up to index
    for(var i:int = 0; i < index; i++)
    {
        graphicsData[i] = graphicsDataBuffer[i];
    }
    // draw graphics data
    graphics.clear();
    graphics.drawGraphicsData(graphicsData);
}
}
}

```

注意 onMouseUp 函数会设置缓冲的长度，确保最后的路径加于 Vector 的最后一个元素。这种行为通常作为撤销操作。如果画了几条线，撤销了几步又再画了几条，那么就无法再重复之前被撤销的几步了。这种现象是因为你在当中插入了数据。

旧酒装新瓶——一套带历史功能的绘画程序。当然，你可以做的更专业一点，加入颜色和线条的粗细控制界面来代替随机变化，等等。关于图形数据，这例子仍然只能算冰山一角。记住，你还可以做任何类型的填充：渐变、位图，甚至底纹填充（第九章），还能画三角。如果你还热血，可以把之前的 ImageSphere 拿出来，把 drawTriangles 改写成 drawGraphicsData。

## 总结

这一章讲述了新的绘图 API，但只能算蜻蜓点水。在读完整本书后，有更多的东西等着你做。实际上，你可能更想看一下 Todd Yard 写的《AS3.0 基础图像效果》(Foundation ActionScript 3.0 Image Effects)

下一章，我们将观摩 Flash 10 的另一个梦幻图像工具：Pixel Bender。

## 第九章 Pixel Bender

本章是一个非常有趣的章节，因为本章不但要介绍一个全新的可以跟 Flash CS4 结合的工具（跟你编译 flash 10 版本的 swf 紧密相关），而且也会介绍用这个工具来编写的一种全新语言。其实早在 2007 年，Pixel Bender Toolkit 的上一版本就公布了，代号为 Hydra。它有一个简洁的 IDE（Integrated Development Environment）用来编写和编译 pixel shader，这个 shader 就像一个迷你小程序，你可以在 Flash、PhotoShop 以及 After Effects 里使用以创造丰富的图片处理和动画效果。关于 Pixel Bender 可以写一整本书，但在这一章里我会尝试带你入门，把基本主要的知识告诉你，希望你能够继续学习 Pixel Bender，往更深一层研究。

### 什么是 Pixel Bender？

Pixel Bender Toolkit 主要用于编写、编译、调试和导出 pixel shader 用于各种 CS4 产品，包括 Flash。这个工具界面极其简单，一个代码编辑窗口、一个预览窗口和一个设置参数值/显示错误警告窗口（图 9-1）。

Pixel Bender 是一种编写 pixel shader 语言，可以在 Pixel Bender Toolkit 里编写。在开始之前，我假设你已经成功安装了 Pixel Bender Toolkit，知道怎样运行程序。

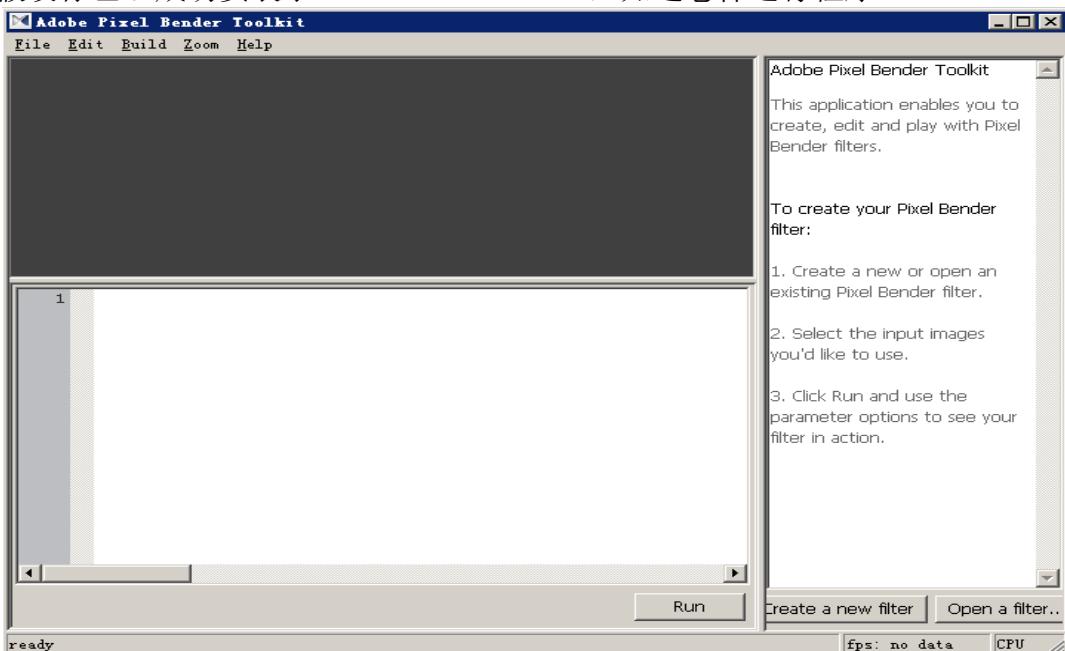


图 9-1. Pixel Bender Toolkit

什么是一个 pixel shader？简单来说，就是一个用来计算像素值的程序。这样说也好像太简单了吧，不过这就是 pixel shader 要做的基本任务。各种 pixel shader 都是由输入和复杂运算算法组成的，最后它只会告诉你：“这个像素应该是这个值。”

有很多原因说明为什么 Pixel Bender 是一项牛 B 项目、为什么人们都为它感到兴奋。第一，Pixel Bender 可以应用于位图、填充以及其它可视对象，然后运行 pixel shader 作用在每一个像素。不是逐个逐个，而是一次过。没错，它是在同一时间计算一个区域内的所有像素值。它编译后得到优化并且运行在独立的进程，是独立于 Flash Player 的。总而言之，相对于 Flash 里的图形处理，pixel shader 的执行效率是非常快的。

不过 Pixel Bender shader 有一个缺点，就是它必须在外部使用 Pixel Bender Toolkit 来编写并且使用基于 C 的 Pixel Bender 语言。在使用它之前必须编译、保存、加载或绑定到你的 Flash 文件里。工作量不少，而且又要学 C 语言。不过不用担心，对于聪明的你来说这都是很简单的。

相比你熟悉的 ActionScript 程序来说 Pixel Bender 比较特别。因为 shader 只对单一像素感兴趣而且在一瞬间可能要执行上千次，所有它本身知道的就只有当前处理中的像素 x、y 坐标值。你可以定义变量和传递参数（包括一个或多个的输入图片），但是当你开始编写程序时，你会发现到处

都有诸多限制。不过请继续往下阅读，之后你就会知道到 Pixel Bender 是不会令你失望的。

在 Flash 里，使用 Pixel Bender shader 可以实现以下四点：

1. 自定义滤镜：就像使用投影、模糊那些滤镜一样，你现在可以把 ShaderFilter 应用到任何显示对象了。ShaderFilter 是依赖着你在 Pixel Bender Toolkit 写的 shader。
2. 填充：在绘图 API (Application Programming Interface) 里，你填充图形时总是使用纯色、渐变色或者位图。现在你可以使用 beginShaderFill 方法填充一个 Pixel Bender shader 了。
3. 混合模式：一个 pixel shader 可以用作混合模式，它将影响显示对象的层叠显示方式。要使用它，将 Shader 对象指派给前景显示对象的 blendShader 属性。
4. “Generic number crunching”：使用 shader 可以处理一组数据（包含复杂的数学运算），功能强、速度快。跟平常使用的 shader 不同，你传递二进制数据进去，用指定的算法处理这些数据，然后返回处理后的二进制数据。你所需要做的事情就是像传入图片一样传入二进制数据。这种技术已经超出这本书的范围了，将不会赘述，不过这是一种你值得深入研究的技术。

## 编写一个 Pixel Shader

打开你的 Pixel Bender Toolkit 准备编写一个 pixel shader。选择“File > New Kernel Filter”或者点击右下角的“Create New Filter”按钮。在编辑器里会出现如下所示代码：

```
<languageVersion : 1.0;>
kernel NewFilter
<  namespace : "Your Namespace" ;
  vendor : "Your Vendor" ;
  version : 1;
  description : "your description" ;
>
{
  input image4 src;
  output pixel4 dst;
  void
  evaluatePixel()
  {
    dst = sampleNearest(src,outCoord());
  }
}
```

如果你想查看一下这个 shader 的最终效果，你需要加载一张图片。在菜单栏里选择“File > Load Image 1…”，默认情况下 toolkit 会预备好一些图片，以防在你的电脑和网上都找不到图片。选择一张图片，你会在预览窗口里看到它。按一下“Run”按钮，你会看到 shader 运行后的最终效果。

你看到了什么？正确来说你什么也看不到。这张图片什么也没改变，因为默认新建的 shader 只是把原图的每一像素按原来的样子输出，什么也不做。

事实上，这个默认新建的 shader 例子放在本章开头有点不合适，所以改了一下代码：

```
<languageVersion : 1.0;>
kernel NewFilter
<  namespace : "Your Namespace" ;
  vendor : "Your Vendor" ;
  version : 1;
  description : "your description" ;
>
{
  input image4 src;
  output pixel4 dst;
```

```

void
evaluatePixel()
{
    dst = pixel4(1, 0, 0, 1);
}
}

```

现在这个 shader 依然在运行的，你修改了代码，但如果你喜欢，你不用停止它，也不用保存这个文档。修改代码后请按一下“Run”按钮，你会看到了一块红色（稍后你会明白为什么）。

以上就是一个容易入手的简单例子。任何 pixel shader 必需有以下元素：

一个用来描述语言版本的元数据标签。

kernel 的定义，kernel 包含余下所需元素（稍后会对它进行解说）。

必需的着色器元数据，包含 namespace、vendor、version 以及可选的 description。一个 output 属性，它用来输出结果。一个 evaluatePixel()方法，它是 shader 的主函数。实际上，当你为 Flash 编写 shader 时，shader 有且只能有一个函数，就是这个 evaluatePixel()函数。所以，要执行的所有代码都是往这个函数里放。一个 input 属性，它可以是一张位图。虽然没有严格要求这个属性，但是你都想定义它。因为在 Pixel Bender Toolkit 开发时，你都想第一时间能看到调试结果。当然，当 shader 用作填充时就不需要传入任何图片了，也就是不需要定义 input 属性了。

首先要说的是这个用来描述语言版本的元数据标签，整个 shader 代码第一行就是它了：

```
<languageVersion : 1.0;>
```

当然，在未来会有更多版本出现。所以以上一行代码是告诉编译器和 Flash 你所写的 shader 属于哪个版本。

接着要说的是定义 kernel。kernel 是 shader 的基本单位，对待 kernel 就好像对待一个类：它是定义 pixel shader 对象的唯一单位。它可以有函数、变量和常数。函数间可以相互调用，也可以引用 kernel 的变量和常数，当然也可以调用内置函数和函数库。

在 Flash 里使用 shader 有诸多限制：不允许使用其它函数库；不允许调用在主函数 evaluatePixel() 外部的函数。不过，你可以调用内置的函数。

通过使用“kernel”关键字定义一个 kernel，紧接着是它的名字。内容放在大括号里，就像：

Kernel NewFilter

```
{
    // kernel code
}
```

但看一下默认生成的代码，你会发现多了个元数据标签的，就像：

kernel NewFilter

```
< namespace : "Your Namespace" ;
  vendor : "Your Vendor" ;
  version : 1;
  description : "your description" ;
>
{
    // kernel code
}
```

其中 namespace、vendor、version 是必要的。namespace 是一个唯一标识，使用你的域名来充当这个 namespace 就最好不过了。vendor 是你的名字或者你的公司名字。version 是你所编写的 shader 版本，假如日后你作出更新，这个 version 就能很好地告诉别人你当前是哪个版本。最后那个 description（描述）是可选的，不过建议在上面写些东西。在 Flash 里，你可以获取以上这些信息，具体方法在后面会讲到。

在 kernel 里，我们首先得到的是两个特殊的变量，分别用 input 和 output 关键字声明。input 是传入你需要处理的图片，output 是返回处理后的图片。

如果你只使用 ActionScript，那么在 Pixel Bender 里定义变量的语法会令你觉得奇怪。在

ActionScript 里，我们定义变量时像下面一样：

```
var variableName:Type;
```

Pixel Bender 是基于 C 语言的，定义变量时使用如下语法：

```
Type variableName;
```

对于上述那两个特殊变量 `input` 和 `output`，在定义这两个变量时，我们还要额外添加关键字 `input` 和 `output`，然后才加上类型。如下所示，`input` 变量类型为 `image4`，`output` 变量类型为 `pixel4`：

```
input image4 src;
```

```
output pixel4 dst;
```

下一节我将会讲到数据类型，不过先在这里说一下 `image4` 和 `pixel4` 类型。`image4` 是代表一张具有四通道的位图，`pixel4` 代表一个具有四通道的像素。四通道：红（red）、绿（green）、蓝（blue）、透明（alpha）。

接下来就是 `evaluatePixel()` 函数了。像定义变量一样，C 语言先把返回数据的类型放在前面，接着才是函数名，要执行的东西都放在大括号内：

```
void  
evaluatePixel()  
{  
    dst = pixel4(1, 0, 0, 1);  
}
```

在新建 shader 时，软件生成的代码会把 `void` 和函数名分成两行，其实放在同一行也一样的：

```
void evaluatePixel()  
{  
    dst = pixel(1, 0, 0, 1);  
}
```

上面这个函数，返回类型为 `void`，也就说函数不用返回什么。但事实上，我们需要返回经过处理后的图片，也就是 `output` 变量。不过这些工作已经在函数体里实现了，将一个具有四通道的像素赋值给 `output` 变量。`pixel(1, 0, 0, 1)` 的意思是把红通道设为 1，绿通道设为 0，蓝通道设为 0，透明通道设为 1。最后得到的是一个红色的像素。通道值的范围在 0 到 1，就算你把它设为范围外的数值，它的范围始终被控制在 0 到 1。例如你把红通道设为 100，跟设为 1 是一样的。上面这个例子的结果是，输出一片红色。把你传入的图片的每一个像素在同时间变为红色。无论你传入哪张图片，结果都是红色一块，那块红色的宽和高跟你传入的图片一样。

恭喜！你已经成功编写了第一个 `pixel shader` 了。请继续往下阅读，你将会学到更多更有趣的。  
**数据类型**

在继续学习之前，我们有必要先了解一下 Pixel Bender 可用的数据类型。基本的类型有 `bool`、`int`、`float`，相当于 ActionScript 里的 `Boolean`、`int`、`Number`。`bool` 类型的变量存放 `true` 和 `false`，`int` 类型的变量存放整数，`float` 类型的变量存放浮点数。

有一点很重要，无论你什么时候使用浮点数（`float`），都必需带小数部分，否则当 `int` 类型处理。举个例子，这句代码 `float num = 1;` 是错误的，编译时你会收到错误信息。你必须这样：`float num = 1.0;` 请时刻记住这点，否则你永远会收到编译错误信息。

在 Pixel Bender 里也有矢量（vector），就像 ActionScript 里的 `Vector` 类型一样，它包含的所有元素均具有相同数据类型。举个例子，`float2` 类型的变量存放两个 `float` 变量，`float3` 类型的变量存放 3 个 `float` 变量，`float4` 类型的变量就当然存放 4 个 `float` 变量了。`bool` 和 `int` 类型也一样。在众多数据类型中，`float2` 类型的使用率最高。因为它可以存放一个坐标值，像 ActionScript 里的 `Point` 类一样。

还有就是 `pixel` 和 `image` 类型。`pixel1` 类型的变量存放只具有一个通道的像素。还有 `pixel2`、`pixel3` 和 `pixel4` 这三个矢量（vector）类型。`pixel4` 使用率最高，因为它可以存放具有四通道（red、green、blue、alpha）的像素。同一道理，在 `image1`、`image2`、`image3`、`image4` 中，`image4` 使用率最高。在 ActionScript 的 `Vector` 类型里，是使用下标取得单个元素的，像：`myVector[2]`

但 Pixel Bender 的矢量（vector）更人性化了。举个例子，在一个 `pixel4` 变量里，要取得第一个元素（red，红通道值），你不必编写 `myPixel[0]`，而应编写 `myPixel.r`。同样地，要取得绿、蓝、

透明通道值，你可以使用对应的 g、b、a 属性。同样地，想要取得 float2 变量值，可以使用 x 和 y 属性。

其实，不管是什么类型的变量，只要是四元素的变量都遵循以下属性名规则：

```
r, g, b, a  
x, y, z, w  
s, t, p, q
```

四元素变量都有以上属性名，myPixel.r、myPixel.x 和 myPixel.s 都是指取得第一个元素的值。从人的感官上看，pixel 矢量类型的变量习惯用 rgba 属性，而 float 矢量类型的变量习惯使用 wxyz 属性。之前那个例子，你可以写成这样：

```
void  
evaluatePixel()  
{  
    dst.r = 1.0;  
    dst.g = 0.0;  
    dst.b = 0.0;  
    dst.a = 1.0;  
}
```

还有更多的高级的玩法，就像以下两个例子：

myPixel.rgb // 返回一个 pixel3 变量，包含 red、green 和 blue 通道值

myPixel.ra // 返回一个 pixel2 变量，包含 red、alpha 通道值

再举个例子：

```
evaluatePixel()  
{  
    pixel3 ppxl = pixel3(1, 0, 1);  
    dst.rgb = ppxl;  
    dst.a = 1.0;  
}
```

我们首先定义一个 pixel3 类型的变量 ppxl，red 通道为 1、green 通道为 0、blue 通道为 1，它是紫色。然后把 ppxl 变量赋给 dst.rgb。最后还有 alpha 通道，我们直接把 1.0 赋给它了。当然了，rgb 的顺序不是固定的，就好像：

```
evaluatePixel()  
{  
    pixel3 ppxl = pixel3(1, 0, 1);  
    dst.rgb = ppxl;  
    dst.a = 1.0;  
}
```

仔细看 `dst.rgb = ppxl;`，我把 g 和 b 的位置调换了，使用 rgb 值变为(1, 1, 0)，你得到的是黄色了。还有更高级的玩法：

```
evaluatePixel()  
{  
    pixel3 ppxl = pixel3(1, 0, 1);  
    dst.rgb = ppxl.grb;  
    dst.a = 1.0;  
}
```

上面把变量 ppxl 的 r 和 g 调换了，然后赋给 dst.rgb，得到的是青色。在矢量（vector）领域里这种著名的玩法叫“swizzling”。它使用简单的语法实现强大的功能。

### 获取当前像素坐标

有一点非常重要，就是当你计算一个像素时，要获取该像素的坐标值。你可以使用内置函数 `outCoord()`。它返回一个 `float2` 类型的值，包含 x 和 y 值。因为每一个像素都有各自的坐标，通过获取每一个像素的坐标你就可以为每一个像素染上不同的颜色，不用像上一节那个例子一样通通都染上同一种颜色。接下来会给出一个例子，这个例子你可以在[www.friendsofed.com](http://www.friendsofed.com) 网站上找到

([AdvancEDActionScriptAnimation.rar](#) 3.8 MB), 名为SineWave1.pbk:

```
<languageVersion : 1.0;>
kernel SineWave1
< namespace : "com.friendsofed" ;
  vendor : "Advanced ActionScript 3.0 Animation" ;
  version : 1;
  description : "draws vertical bands" ;
>
{
  input image4 src;
  output pixel4 dst;
  void
  evaluatePixel()
  {
    dst = pixel4(0, 0, 0, 1);
    float2 pos = outCoord();
    dst.r = sin(pos.x * .2) * .5 + .5;
  }
}
```

请看上面代码, 重要部分已经加粗。首先将当前像素染上不透明的黑色(`rgb`都为0时是黑色)。然后获取当前像素的坐标值并赋给 `float2` 类型的 `pos` 变量。请看最后那行代码, 先把 `pos.x` 乘以 0.2, 再取得它的 `sin` 值, 这时我们得到范围为-1 到 1 的数。接着乘以 0.5, 得到-0.5 到 0.5 的数。最后加上 0.5, 得到 0 到 1 之间的数, 这刚好是红通道的范围。把计算结果赋给 `dst.r`, 得到平滑相间的渐隐红色, 它是根据当前像素的坐标 `x` 值得出的。如图 9-2:



图 9-2. Pixel Bender gradient bars

同样你也可以根据当前像素的坐标 `y` 值得出另一种效果, 正如下面这个例子(SineWave2.pbk):

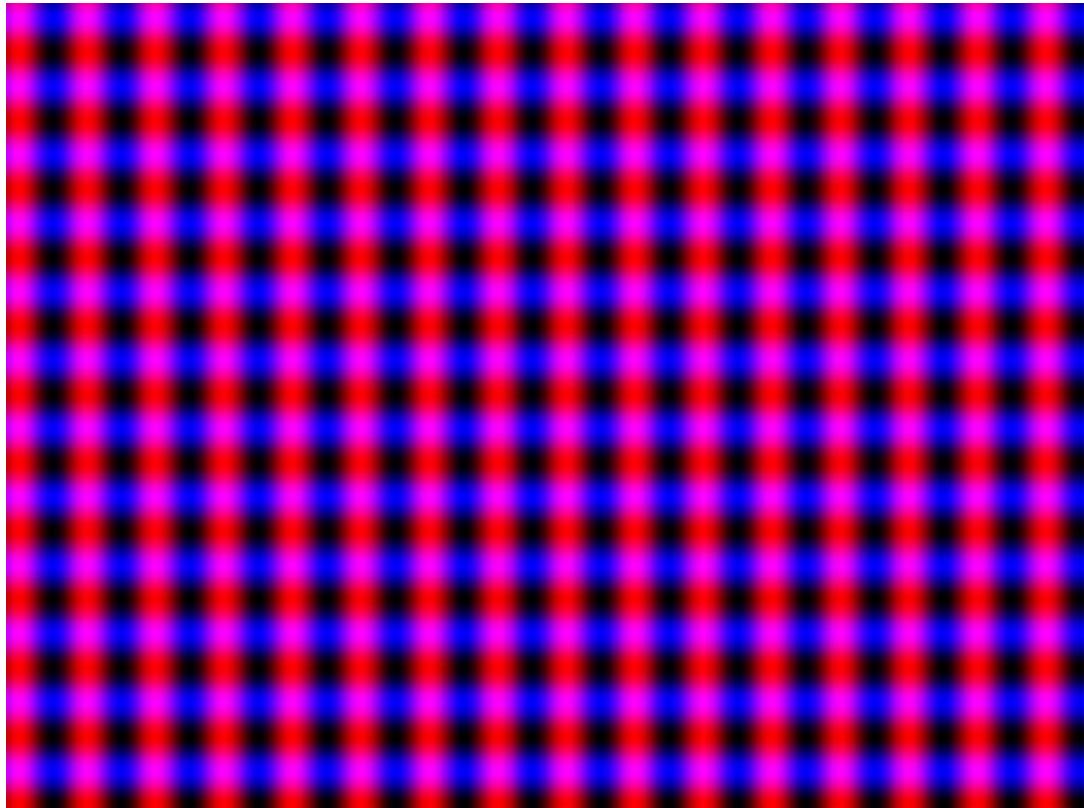
```
<languageVersion : 1.0;>
kernel SineWave2
< namespace : "com.friendsofed" ;
```

```

vendor : "Advanced ActionScript 3.0 Animation" ;
version : 1;
description : "draws vertical and horizontal bands" ;
>
{
    input image4 src;
    output pixel4 dst;
    void
    evaluatePixel()
    {
        dst = pixel4(0, 0, 0, 1);
        float2 pos = outCoord();
        dst.r = sin(pos.x * .2) * .5 + .5;
        dst.b = sin(pos.y * .2) * .5 + .5;
    }
}

```

如图 9-3 所示：



**图 9-3. Double bars**

现在，假设你想同时改变多个通道，例如你想实现黑白相间的效果，那你就需要将红、绿、蓝三个通道都赋给同一个值。你可以修改代码如下：

```

void
evaluatePixel()
{
    dst = pixel4(0, 0, 0, 1);
    float2 pos = outCoord();
    dst.r = sin(pos.x * .2) * .5 + .5;
    dst.g = sin(pos.x * .2) * .5 + .5;
    dst.b = sin(pos.x * .2) * .5 + .5;
}

```

聪明的你会发觉上面的代码很浪费时间，所以你可以修改代码如下：

```
void
```

```

evaluatePixel()
{
    dst = pixel4(0, 0, 0, 1);
    float2 pos = outCoord();
    pixel1 pxl = sin(pos.x * .2) * .5 + .5;
    dst.r = pxl;
    dst.g = pxl;
    dst.b = pxl;
}

```

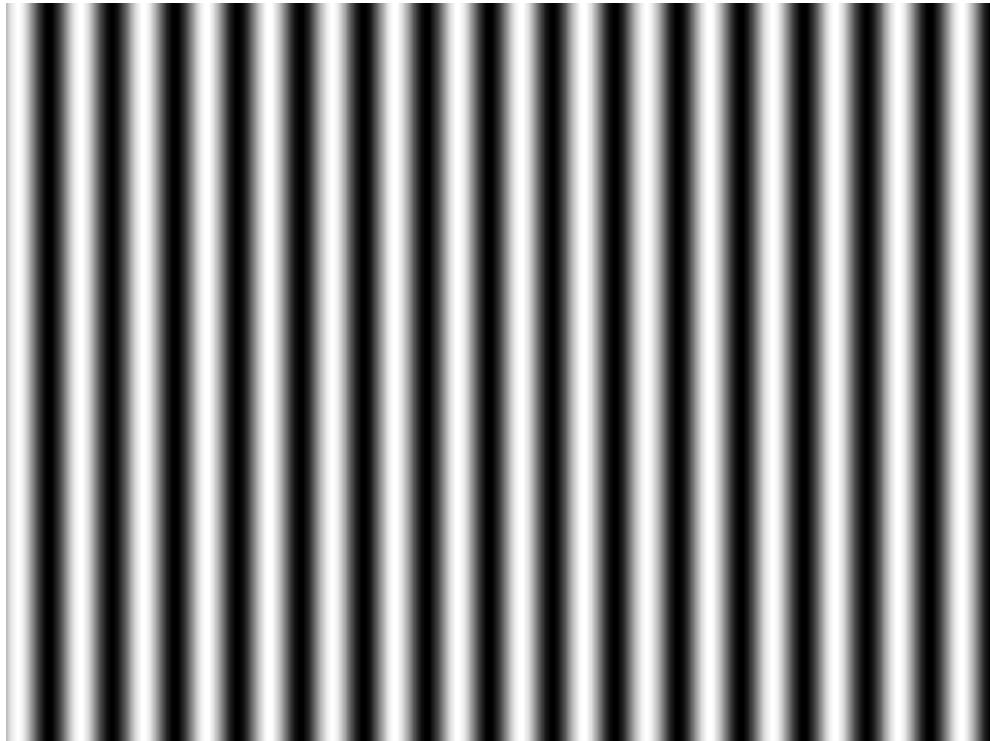
上面的代码只须计算一次 p xl 的值，然后赋给每一个通道。但，其实还有一种更简单的方法，就像下面这个例子（SineWave3.pbk）：

```

<languageVersion : 1.0;>
kernel SineWave3
< namespace : "com.friendsofed" ;
  vendor : "Advanced ActionScript 3.0 Animation" ;
  version : 1;
  description : "draws vertical and horizontal bands" ;
>
{
    input image4 src;
    output pixel4 dst;
    void
    evaluatePixel()
    {
        dst = pixel4(0, 0, 0, 1);
        float2 pos = outCoord();
        dst.rgb = pixel3(sin(pos.x * .2) * .5 + .5);
    }
}

```

只须将一个值赋给矢量（vector），Pixel Bender 就会自动把矢量的其余元素都赋上同一个值。就像上面那个例子 `pixel3(sin(pos.x * .2) * .5 + .5)`，使得 `pixel3()` 里面 r、g、b 三个元素的值都一样，最后产生如下图所示黑白相间（图 9-4）：



#### 图 9-4. Grayscale bars

那本Pixel Bender语言参考（[《Adobe® Pixel Bender™ Language 1.0 Tutorial and Reference》](#)），在你安装Pixel Bender Toolkit时就附带了的。多翻开它看一下，它描述了所有内置函数，一些复杂数学运算函数也有。找几个用试玩一下，感受一下它们的魅力。接下来，我们将会学到如果在运行时改变参数值。

#### 参数

在 Pixel Bender 运行后不能更改变量的值总是失色不少。假如你对一张图片进行模糊处理，有轻微模糊、中度模糊、超级模糊，在 Pixel Bender 运行后你想调节模糊程度，怎么办？幸好 Pixel Bender 支持运行后修改变量值，使用这个特殊的关键字“parameter”，表示向 Pixel Bender 添加参数。使用它很简单，就像：

```
parameter float myParameter;
```

当你在程序里添加了数值参数，一旦在 Pixel Bender Toolkit 里运行你的 Pixel Bender，你会在右上角看到对应该参数的拉动条（slider）。如果你添加的是布尔（Boolean）参数，你会看到一个复选框。下面例子说明这个（SineWaveParam.pbk）：

```
<languageVersion : 1.0;>
kernel SineWaveParam
<  namespace : "com.friendsofed" ;
  vendor : "Advanced ActionScript 3.0 Animation" ;
  version : 1;
  description : "draws vertical bands" ;
>
{
  input image4 src;
  output pixel4 dst;
  parameter float mult;
  void
  evaluatePixel()
  {
    dst = pixel4(0, 0, 0, 1);
    float2 pos = outCoord();
    dst.r = sin(pos.x * mult) * .5 + .5;
  }
}
```

运行以上例子后你应该会看到那个拉动条（slider），如图 9-5：

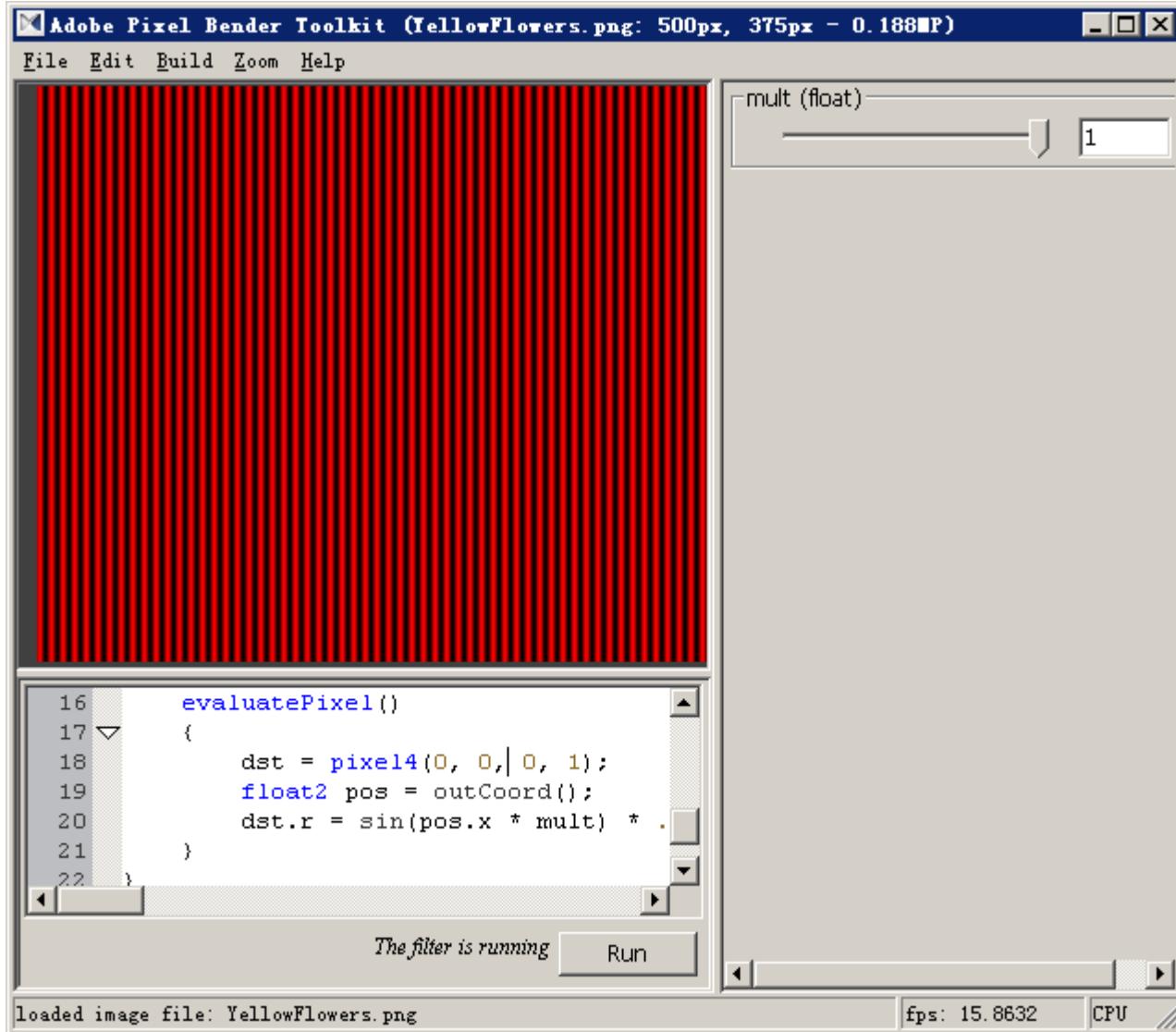


图 9-5. Parameter slider in the Pixel Bender Toolkit

我们定义了一个叫 mult 的参数。用这个参数乘以 pos.x，再求 sin 值。通过拉动条，你可以改变它的值。把拉动条从头到尾拉一次，看看效果是如何变化的。

如果你的参数是一个包含多个元素的矢量（vector），那么你会得到一组拉动条，像接下来的例子

（ColorChooser.pbk）：

```

<languageVersion : 1.0;>
kernel ColorChooser
< namespace : "com.friendsofed" ;
  vendor : "Advanced ActionScript 3.0 Animation" ;
  version : 1;
  description : "shows a pixel4 parameter in action" ;
>
{
  input image4 src;
  output pixel4 dst;
  parameter pixel4 color;
void
  evaluatePixel()
  {
    dst = color;
  }
}
```

因为这是一个 pixel4 类型的参数，所以你会得到四个拉动条，如图 9-6 所示：

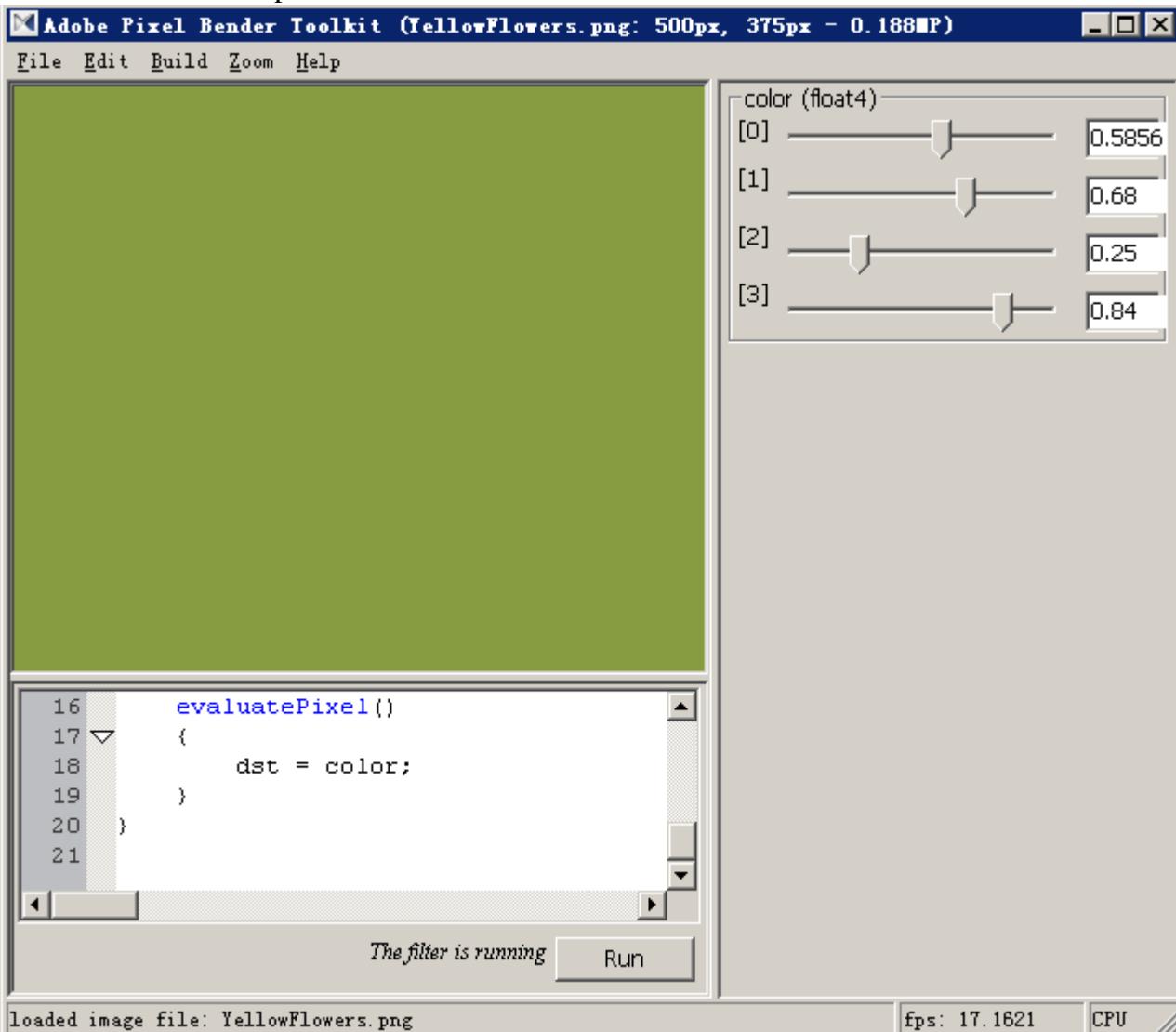


图 9-6. A four- channel slider

拉动那四条拉动条，看一下效果的变化情况。当然，要看到效果，你必须把最后那条拉动条“[3]”的值提高一些，因为它代表透明（alpha）通道。

#### 高级参数

当你定义参数时，Pixel Bender 会给你一个默认的最小值 0、最大值 1 和开始值 0。对于一些参数来说，这是好事来，但一些情况下就不是那么说了。为了改变默认值，你可以使用元数据标签，放在变量名后面，就像：

```
parameter float myValue
<
    minValue:0.0;
    maxValue:100.0;
    defaultValue:50.0;
>;
```

**语法重点：**元数据标签是紧跟在变量名后面，在标签结束后需要一个分号。

在 minValue、maxValue 和 defaultValue 后面紧跟着一个冒号，随后就是你想要的数值，最后别忘了加上分号。这个例子使得拉动条的值在 0.0 到 100.0 之间，并且开始时的值为 50.0。

再奉上一个例子，使用了两个 int 类型的参数，它产生棋盘形图案(（国际象棋那种黑白相间棋盘）)，文件名为 Checkerboard.pbk：

```
<languageVersion : 1.0;>
kernel Checkerboard
```

```

< namespace : "com.friendsofed" ;
  vendor : "Advanced ActionScript 3.0 Animation" ;
  version : 1;
  description : "creates a checkerboard pattern" ;
>
{
  input image4 src;
  output pixel4 dst;
  parameter int xres
  <
    minValue:1;
    maxValue:200;
    defaultValue:50;
  >;
  parameter int yres
  <
    minValue:1;
    maxValue:200;
    defaultValue:50;
  >;
  void
  evaluatePixel()
  {
    float2 pos = outCoord();
    float xpos = floor(pos.x / float(xres));
    float ypos = floor(pos.y / float(yres));
    if(mod(xpos, 2.0) > 0.0 ^^ mod(ypos, 2.0) > 0.0)
    {
      dst = pixel4(1, 1, 1, 1);
    }
    else
    {
      dst = pixel4(0, 0, 0, 1);
    }
  }
}

```

首先通过 `outCoord()` 函数取得当前像素的坐标，除以 `xres`，再用 `floor()` 函数取得下限值。例如，假如图片的宽度为 500，传入的 `xres` 参数值为 100，结果就得到 1 到 5。注意，我们需要将 `int` 类型的 `xres` 参数强制转换为 `float` 类型，以便跟 `pos.x` 相除。`y` 轴的也一样。

接着上面分析得出的值 (`xpos` 和 `ypos`)，我们把每一个值除以 2.0 后求余数 (通过内置的 `mod()` 求模函数)，余数不为零说明是奇数，相反是偶数。操作符 “`^^`” 代表异或。异或的意思是说两者其中的一个为 `true` (不包含二者同时为 `true` 的情况)，就为 `true`。当以上条件为 `true` 时，就染上白色；否则染上黑色。最后得到的效果是一张棋盘形图案 (国际象棋那种黑白相间棋盘)。你可以改变滚动条，看看效果。

### 对输入图片进行取样

到目前为止，所有例子都只利用了输入图片用来提供一个处理区域。但是 Pixel Bender 的本质是一门图片处理语言，意味着它可以处理图片。理所当然地它应该提供访问图片数据的方法。所以它有一对取样 (sampling) 函数。取样是指取得图片上确定位置的像素值。

最简单的取样函数是 `sampleNearest()`。它需要传入两个参数，第一个是图片，第二个是包含 `x` 和 `y` 坐标值的 `float2` 变量。它会找到最接近 `float2` 变量的那个像素，然后返回该像素值。我们回到最开始的时候，看一下开始时 Pixel Bender 自动为你生成的代码：

```
void
evaluatePixel()
{
    dst = sampleNearest(src, outCoord());
}
```

我们传入第一个参数 `src`, 这个参数代表输入图片; 然后传入第二个参数 `outCoord()`, 这个参数返回当前像素的坐标, 一个 `float2` 变量。譬如说当前处理的像素是(100, 100), 那么这个取样函数会找到坐标为(100, 100)的那个像素, 然后返回该像素值。上面代码是点对点地处理图片, 所以运行后你看到处理后的图片跟原图是一样的。当然, 稍后我们会改动一下代码以实现更多有趣的效果。先来一个例子, 它会依据像素原来位置来产生扭曲 (`GlassTile.pbk`):

```
<languageVersion : 1.0;>
kernel GlassTile
< namespace : "com.friendsofed" ;
  vendor : "Advanced ActionScript 3.0 Animation" ;
  version : 1;
  description : "creates a glass tile refraction effect" ;
>
{
    input image4 src;
    output pixel4 dst;
    parameter float mult
    <
        minValue:0.0;
        maxValue:10.0;
        defaultValue:1.0;
    >;
    parameter float wave
    <
        minValue:0.0;
        maxValue:1.0;
        defaultValue:0.1;
    >

    void
    evaluatePixel()
    {
        float2 pos = outCoord();
        pos += float2(sin(pos.x * wave) * mult, sin(pos.y * wave) * mult);
        dst = sampleNearest(src, pos);
    }
}
```

上述例子不再简单地取得当前像素值了, 它使 `x` 和 `y` 坐标发生一定程度的偏移: 先求 `sin` 值, 然后乘以一个数。发生偏移的程度取决于 `wave` 参数和 `mult` 参数。运行结果如图 9-7:



图 9-7. A glass tile effect

### 线性取样

我之前提到过，取样函数有两个。目前只介绍了 `sampleNearest()` 函数。它的取样方式是：找到最接近你所给出的坐标的像素。请看图 9-8：

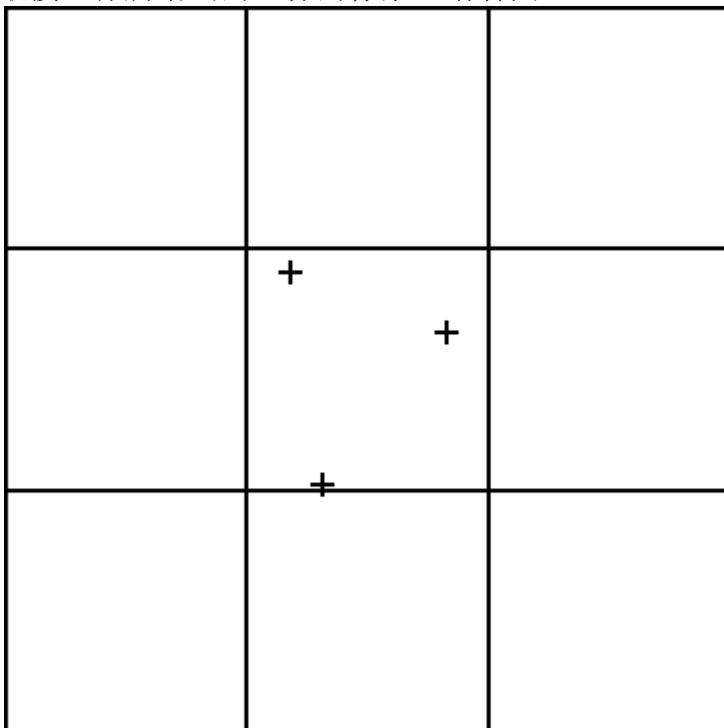


图 9-8. All three sampled points will get the value of the middle pixel

这图展示了九个像素的区域。假设中间那个像素位于(100, 100)，那三个十字架所处的位置坐标分别传入 `sampleNearest()` 函数里。左上角那个十字架可能位于(99.7, 99.6)，右边那个可能位于(100.4, 99.9)，最下面可能是(99.8, 100.5)。三点都非常接近坐标(100, 100)，所以它们三点的像素值都一样。尽管这种取样方式不错，运行起来也很快，但是处理结果会出现马赛克。正如你看到的以下图片，我把它放大了一些（图 9-9）：



图 9-9. Blockiness resulting from sampleNearest

如果你埋怨那些马赛克，你可以尝试一下另外一个取样函数：sampleLinear()。假如你有看过帮助文档，你会发现取样函数有三个：sample()、sampleLinear()和sampleNearest()。事实上，sample()只是sampleLinear()的别名，他们俩完全是一样的。sampleLinear()的取样方式是：找到最接近取样点的四个像素，然后取它们的平均值。如图 9-10 所示，将会取得最接近十字架那四个像素的平均值：

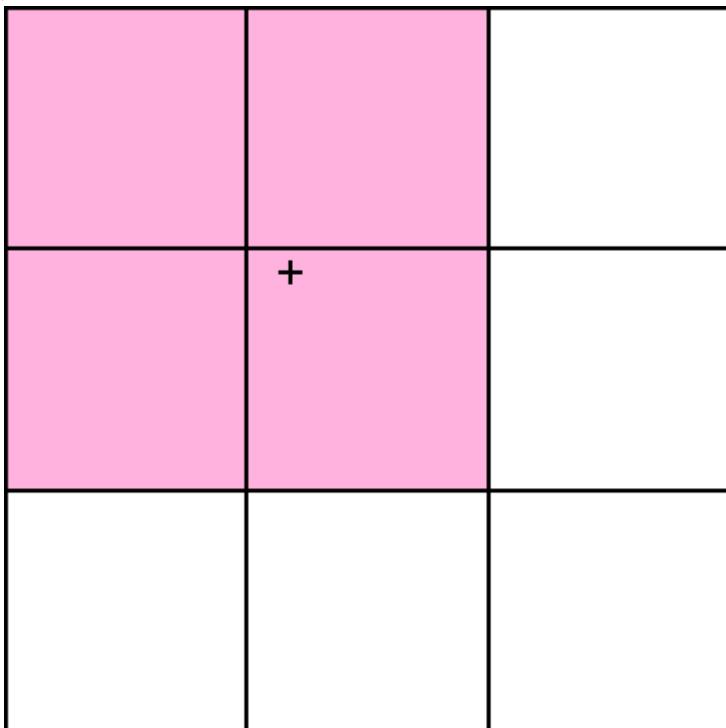


图 9-10. The sampled point will get a weighted average of the four nearest pixels

上图加了色的四个像素将被取样。事实上，取样时不是简单的取平均值，而是用双线性内插值法。简而言之，它会选择最近的四个点。所以，你会得到一个比之前更圆滑的效果。实践一下吧，找到之前那个例子 GlassTile.pbk，请把：

```
dst = sampleNearest(src, pos);
```

改为：

```
dst = sampleLinear(src, pos);
```

或者：

```
dst = sample(src, pos);
```

这回的确去除了不少马赛克了，得到平滑的效果，如图 9-11：



图 9-11. A smoother gradient produced by sampleLinear

在学习如何在 Flash 里使用 shader 之前，再来一个关于扭曲效果的例子。

### 适用于 Flash 里的 Twirl Shader

安装 Pixel Bender Toolkit 时会附带一些滤镜例子的，其中有一个扭曲滤镜。很不幸，这个滤镜使用了一些 Flash Player 暂不支持的特性。所以，让我们做一个 Flash 支持的扭曲滤镜吧。这个扭曲滤镜名为 TwirlFlash.pbk，打开后你可以看到：

```
<languageVersion : 1.0;>
kernel TwirlFlash
<  namespace : "com.friendsofed" ;
  vendor : "Advanced ActionScript 3.0 Animation" ;
  version : 1;
  description : "spins an image" ;
>
{
  input image4 src;
  output pixel4 dst;
  parameter float2 center
  <
    minValue:float2(0.0);
    maxValue:float2(1000.0);
    defaultValue:float2(200.0);
  >;
  parameter float twist
  <
    minValue:-10.0;
    maxValue:10.0;
    defaultValue:0.0;
  >;
  parameter float radius
  <
    minValue:0.0;
    maxValue:500.0;
    defaultValue:100.0;
  >;
  void
  evaluatePixel()
  {
    float2 pos = outCoord();
    float dist = distance(center, pos);
    float PI = 3.14159;
    if(dist < radius)
```

```

    {
        float dx = pos.x - center.x;
        float dy = pos.y - center.y;
        float angle = atan(dy, dx);
        angle += sin(dist / radius * PI) * twist;
        float2 newpos = center + float2(cos(angle) * dist, sin(angle) * dist);
        dst = sampleNearest(src, newpos);
    }
    else
    {
        dst = sampleNearest(src, pos);
    }
}

```

它有三个参数：

`center` 参数定义扭曲效果所产生的位置。记住，在 Pixel Bender 里没有所谓的图片大小，没有所谓的舞台（stage）宽度和高度。

`twist` 参数的大小决定图片被扭曲的程度。

`radius` 参数的大小决定图片被扭曲的范围。这个 shader 会围绕 `center` 点，以 `radius` 为半径进行扭曲。

在 `evaluatePixel()` 函数里，我们取得当前像素坐标，并且还使用了内置的 `distance()` 函数计算了该像素与 `center` 点的距离。甚至还定义了 `PI` 值，因为 Pixel Bender 不提供常数 `PI`：

```

Float2 pos = outCoord();
float dist = distance(center, pos);
float PI = 3.14159;

```

接着我们添加了 `if` 语句。`if` 语句是 Flash 唯一支持的控制语句。Flash 里不支持数组（array）、`for` 语句和 `while` 语句，甚至连 `switch` 语句也不支持。诸多的限制反而更能促使你开动脑筋，使你更能创新！希望如此啦。By the way，如果当前像素到 `center` 点的距离小于半径 `radius`，我们就计算扭曲值，否则直接跳过不做任何处理。

接下来涉及一些数学运算了。分别求出 `pos` 点和 `center` 点的水平和垂直距离，然后再求得夹角：

```

float dx = pos.x - center.x;
float dy = pos.y - center.y;
float angle = atan(dy, dx);

```

将求得的 `angle` 加上一个数以实现扭曲效果：

```
angle += sin(dist / radius * PI) * twist;
```

首先计算出 `dist / radius * PI` 的值。`dist` 的范围在 0.0 到 `radius` 之间，`dist / radius` 的范围在 0.0 到 1.0 之间。乘以 `PI` 后，取值范围变为 0.0 到 3.14159 之间，再取 `sin` 值，刚好是半周期（从 0 到 1，然后又返回到 0）。最后乘上 `twist` 值。整个过程的意思是，以 `center` 点为中心，`radius` 为半径的圆区域内，圆点和边缘处都不会发生扭曲，而在圆心和边缘的中间发生的扭曲程度是最大的。过程比较难理解，请先看图 9-12：



图 9-12. The TwirlFlash shader in action

你现在不但学到了创建shader的基本知识，还学会了一些复杂的技术。在[www.adobe.com](http://www.adobe.com) 上的labs和developer center，甚至到其它网站搜索一下，应该会找到不少关于Pixel Bender的文章和实例。但，请不要把自己局限于Pixel Bender本身。Pixel Bender是基于Open GL shader语言GLSL的，尽管两者有不少差别，但你也应该从中吸收多一点知识。无论如何，我相信现在的你可以单凭自己都可以在Pixel Bender Toolkit里编写一些酷shader了。接下来，让我们看一下怎样在Flash里使用 shader了。

### 在 Flash 里使用 Pixel Bender

正如在开始时所说的，Pixel Bender shader 在 Flash 里有四个用途：

滤镜

填充

混合模式

“Generic number crunching”

你将会学习到前三种，第四种用途范围很广，在此不予以赘述。要在 Flash 里使用 shader 有三个步骤：

导出 Flash 可识别的 shader；

把 shader 加载或者嵌入到 Flash 里；

在 Flash (swf) 里创建 shader 实例，用于填充、滤镜或混合模式。

按以上步骤，首先在 Pixel Bender Toolkit 工具里导出 shader。打开之前那个棋盘形图案的例子 Checkerboard.pbk 文件，选择 “File > Export Kernel Filter for Flash Player” 菜单导出一个.pbj 文件，使用默认的文件名就可以了。接着我们会创建一个 Flash 文件，你可以使用 Flex Builder、Flash CS4，或者其它你喜欢的创作工具。

### 加载或绑定 shader

为了能在 Flash 里使用 shader，你需要把 shader 加载或者嵌入到 Flash (swf) 里。我个人喜欢把 shader 嵌入到 Flash 里。第一，因为是嵌入到 Flash 里的，所以不需要担心找不到 shader；第二，使用 shader 时不需要等待加载它，因为它在加载 Flash 时一同加载进去的。把 shader 嵌入到 Flash 里，编写代码和调试时也方便些。不过，加载的方式也有好处，就是当你改变了 shader 时，只需要替换原来的 shader 而不用重新编译 SWF 文件。

尽管如此，我还是先展示一下加载 shader 的用法，稍后再展示嵌入 shader 的用法。

我们首先编写加载文件的基本结构，像加载 XML，使用 URLLoader 对象和 COMPLETE 事件处理器。你可以下载以下文件 ShaderFillDemo.as：

```
package {
```

```

import flash.display.Sprite;
import flash.display.StageAlign;
import flash.display.StageScaleMode;
import flash.events.Event;
import flash.net.URLLoader;
import flash.net.URLLoaderDataFormat;
import flash.net.URLRequest;
public class ShaderFillDemo extends Sprite
{
    private var loader:URLLoader;
    private var shaderURL:String = "Checkerboard.pbj";
    public function ShaderFillDemo()
    {
        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;
        loader = new URLLoader();
        loader.addEventListener(Event.COMPLETE, onLoadComplete);
        loader.dataFormat = URLLoaderDataFormat.BINARY;
        loader.load(new URLRequest(shaderURL));
    }
    private function onLoadComplete(event:Event):void
    {
    }
}
}

```

需要注意的地方是，我们把 loader 对象的 dataFormat 设置为 BINARY，因为我们将要加载二进制文件。还有就是 shader 的路径要正确，必须是可以让 SWF 访问到 shader 的路径。

一旦 shader 数据加载完成，我们就需要创建一个 Shader 对象来存放它，Shader 对象是 flash.display.Shader 的一个实例。loader 对象的 data 属性包含 shader 二进制数据，把它放到 Shader 的构造函数里：

```

private function onLoadComplete(event:Event):void
{
    var shader:Shader = new Shader(loader.data);
}

```

无论你把 shader 用作什么用途（填充、滤镜…），你都需要以上步骤。下面，我们把它用作填充。

### 使用 shader 作为绘制填充

使用一些 Graphics 绘图方法、beginShaderFill() 填充，把 shader 对象作为参数传入 beginShaderFill() 方法里。你所编写的 shader 能实现怎样的效果，就能填充怎样的效果。全部代码如下：

```

package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;
    public class ShaderFillDemo extends Sprite
    {
        private var loader:URLLoader;

```

```

private var shaderURL:String = "Checkerboard.pbj";
public function ShaderFillDemo()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    loader = new URLLoader();
    loader.addEventListener(Event.COMPLETE, onLoadComplete);
    loader.dataFormat = URLLoaderDataFormat.BINARY;
    loader.load(new URLRequest(shaderURL));
}
private function onLoadComplete(event:Event):void
{
    var shader:Shader = new Shader(loader.data);
    graphics.beginShaderFill(shader);
    graphics.drawRect(0, 0, 400, 400);
    graphics.endFill();
}
}
}

```

假如运行不出差错，你应该会看到一个用棋盘形图案填充了的矩形。

好！接下来我们应该看看如何改变 shader 的参数，不过先让我们来看一下在 Flash 嵌入 shader 的使用方法。像嵌入位图、SWF 文件和字体那样，使用 Embed 元数据标签把 shader 嵌入到 Flash (swf) 里。以下是嵌入 shader 的基本代码 (ShaderFillEmbed.as)：

```

package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    public class ShaderFillEmbed extends Sprite
    {
        [Embed(source="Checkerboard.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;
        public function ShaderFillEmbed()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
        }
    }
}

```

需要注意的是，你需要把 mimeType 的值设为 “application/octet-stream”。还在确保 pbj 文件的存放路径正确。shader 的二进制数据已经包含在 ShaderClass 类里面了，我们只需要这样就能访问 shader：

```
var shader:Shader = new Shader(new ShaderClass());
```

现在你可以把变量 shader 放到 beginShaderFill() 函数里了，就像上一个例子：

```

package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    public class ShaderFillEmbed extends Sprite
    {

```

```

[Embed(source="Checkerboard.pbj", mimeType="application/octet-stream")]
private var ShaderClass:Class;
public function ShaderFillEmbed()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    var shader:Shader = new Shader(new ShaderClass());
    graphics.beginShaderFill(shader);
    graphics.drawRect(0, 0, 400, 400);
    graphics.endFill();
}
}

```

对于我来说，嵌入比加载要来得简单。不过，往后的例子你用哪种方式都没关系。接下来看看如何访问 shader 元数据吧。

### 访问 shader 元数据

在 Flash 里一旦创建了 shader 的实例，你就可以访问在 Pixel Bender Toolkit 里编写 shader 时定义的元数据。通过 shader 的 data 属性就可以访问了，它是 flash.display.ShaderData 的实例。通过 data 属性，你可以访问 namespace、vendor、version 和 description 元数据。甚至你还可以访问 shader 的本身名字，输出结果都为字符串类型。修改上一个例子的代码测试一下：

```

public function ShaderFillEmbed()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    var shader:Shader = new Shader(new ShaderClass());
    trace(shader.data.name);
    trace(shader.data.namespace);
    trace(shader.data.vendor);
    trace(shader.data.version);
    trace(shader.data.description);
    graphics.beginShaderFill(shader);
    graphics.drawRect(0, 0, 400, 400);
    graphics.endFill();
}

```

输出结果：

```

checkerboard
com.friendsofed
Advanced ActionScript 3.0 Animation
1
Create a checkerboard pattern

```

虽然你不一定总是需要知道这些信息，但有时候获取这些信息还是有用的。

### 设置 shader 参数值

在 Pixel Bender Toolkit 里，任何时候你添加了参数，都会有拉动条或者复选框来改变参数值。在 Flash 里，你不再有拉动条和复选框了，你总要想办法在代码里改变参数值。同样地，通过 shader 的 data 属性你可以设置参数值。不过没有之前使用 data 属性那么容易了。每一个你在 shader 里添加的参数，就是 data 属性的属性。像 checkerboard shader: shader.data.xres 和 shader.data.yres 那如何设置 xres 的值呢？你可能会以为这样可以设置 xres 的值：

```
shader.data.xres = 20;
```

在 Flash 运行一下，跟预期的结果不一样，不但没有成功设置 xres 的值，连一个编译时的错误信息也没有。事实上，要改变它的值，还需要多走一步：

```
shader.data.xres.value
```

Wait! 别开心这么早, 还有更多你未知的。shader 的参数可能是一个矢量 (vector), 最多可包含四个元素, 像一个数组。所以你需要在赋值时添加一对中括号, 哪怕参数只有一个元素:  
shader.data.xres.value = [20];

最后, 按上面那样设置参数, 运行结果跟预期的一样了。上一个例子的结果应该图 9-13 一样:

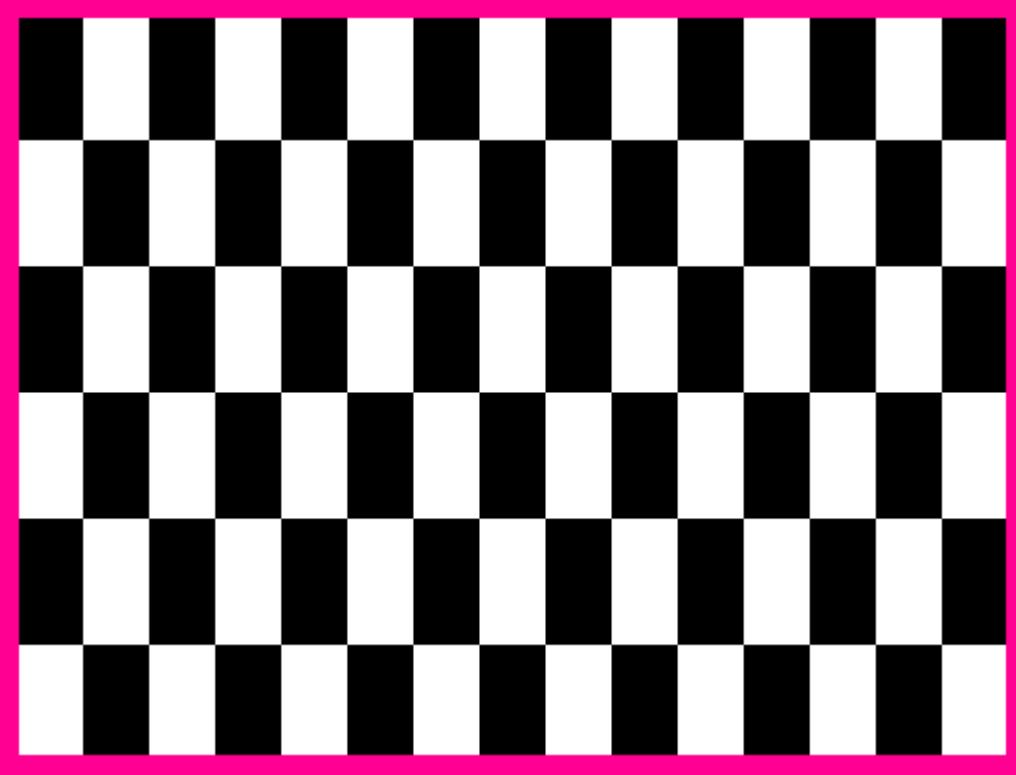


图 9-13. Checkerboard pattern produced with a fill

你可以改变 yres 的值看一下效果。

### 转换 shader 填充

beginShaderFill() 函数, 像其它填充函数一样, 都有第二个参数, 一个 flash.display.Matrix 类型的参数。你可以使用它来进行缩放、旋转和移动等等。如果你看过 Flash 的帮助文档, 你会知道可以使用矩阵 (matrix) 实现旋转, 像下面这行代码, q 是角度, 单位为弧:

```
new Matrix(cos(q), sin(q), -sin(q), cos(q), 0, 0);
```

当然了, 像 Seb (本书的技术评论家) 指出的, 你还可以这样实现旋转:

```
var m:Matrix = new Matrix();
m.rotate(q);
```

现在你有两种方法实现同一样东西了! 现在我们就对 shader 进行旋转:

```
public function ShaderFillEmbed()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    var shader:Shader = new Shader(new ShaderClass());
    var angle:Number = Math.PI / 4; // 45 度的弧度表示
    var cos:Number = Math.cos(angle);
    var sin:Number = Math.sin(angle);
    graphics.beginShaderFill(shader, new Matrix(cos, sin, -sin, cos));
    graphics.drawRect(0, 0, 400, 400);
    graphics.endFill();
}
```

把 shader 旋转 45 度 ( $\text{PI} / 4$  弧), 得出的效果如图 9-14:

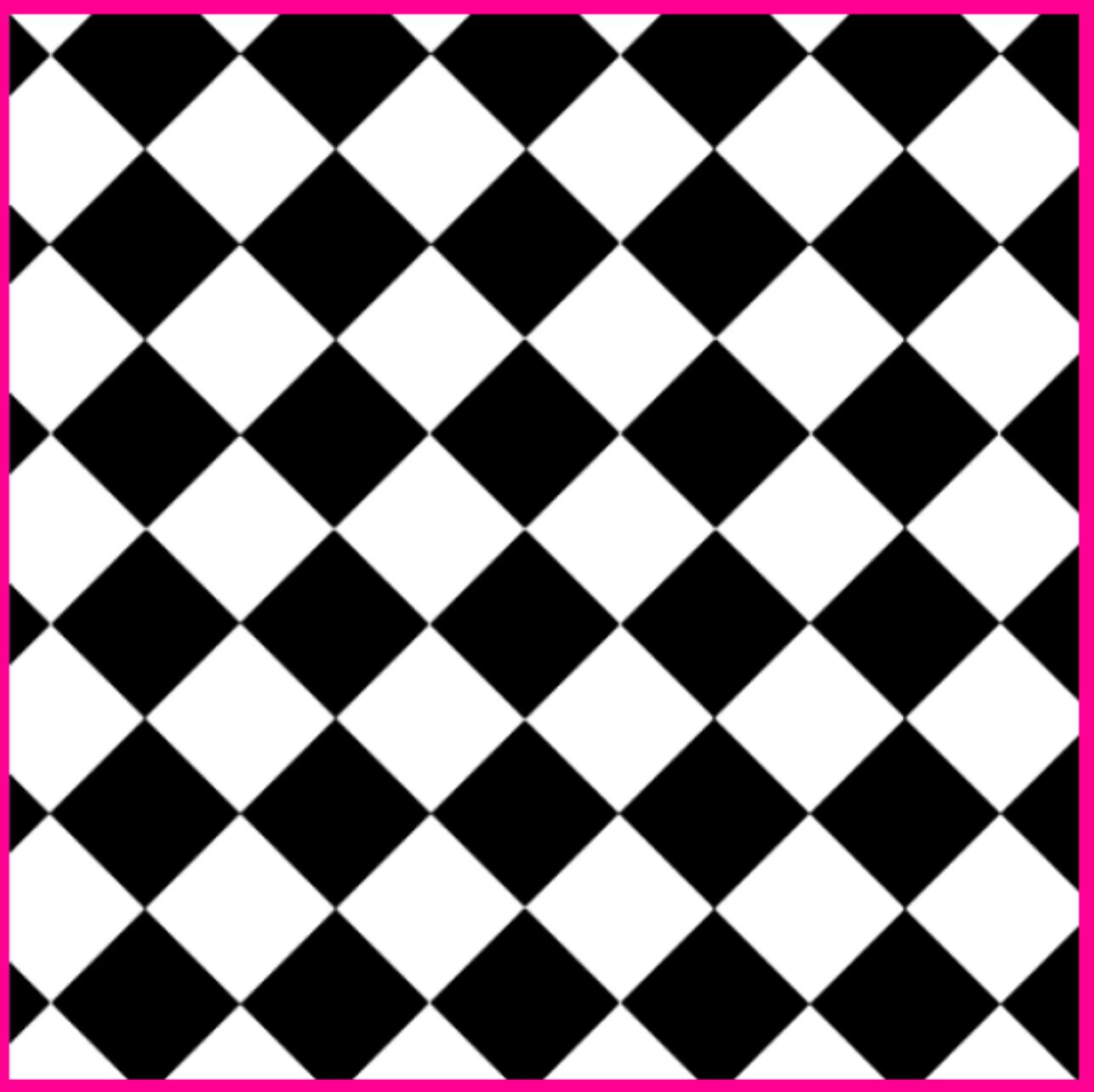


图 9-14. Checkerboard pattern with a rotation matrix

你甚至还可以通过改变 Matrix 的参数来对 shader 进行缩放、移动和倾斜等操作。

#### 用 shader 填充制作动画

这一节让你真正地体会动态更改参数 (parameter) 的魅力。主要思路是：制作一个 shader 填充，修改参数值，在下一帧重新渲染，如此重复执行形成动画。下面是一个简单的例子，依然使用棋盘形图案，文件名 ShaderFillAnim.as：

```
package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;
    public class ShaderFillAnim extends Sprite
    {
        [Embed(source="Checkerboard.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;
        private var shader:Shader;
```

```

private var xAngle:Number = 0;
private var yAngle:Number = 0;
private var xSpeed:Number = .09;
private var ySpeed:Number = .07;
public function ShaderFillAnim()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    shader = new Shader(new ShaderClass());
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    var xres:Number = Math.sin(xAngle += xSpeed) * 50 + 55;
    var yres:Number = Math.sin(yAngle += ySpeed) * 50 + 55;
    shader.data.xres.value = [xres];
    shader.data.yres.value = [yres];
    graphics.clear();
    graphics.beginShaderFill(shader);
    graphics.drawRect(20, 20, 400, 400);
    graphics.endFill();
}
}
}

```

这个例子给 angle 变量一个加速度，使水平和垂直方向上有加速或减速的变化。在每一帧里都向 angle 变量加上一个速度然后求 sin 值，乘以 50 后再加上 55，最后得到的结果会在 5 到 105 的范围内。把结果赋给 shader 的 xres 和 yres 参数，清除上一帧绘出来的图案后，再重新绘制。这样，你就得到动画了。

### 指定 shader 的输入图片

在上一节里，那个 shader 生成的图案只是计算出来的，没有对其输入任何图片。但是，如果你使用像 TwirlFlash 那个例子一样的 shader 呢？那个 shader 的确是用来处理图片的，它返回的每一个像素都是直接依赖于输入的图片。让我们尝试指定 shader 的输入图片（input image）吧。文件 ShaderFillImage.as：

```

package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.filters.ShaderFilter;
    public class ShaderFillImage extends Sprite
    {
        [Embed(source="TwirlFlash.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;
        public function ShaderFillImage()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            var shader:Shader = new Shader(new ShaderClass());
            shader.data.center.value = [400, 300];
            shader.data.twist.value = [-6];
            shader.data.radius.value = [300];
        }
    }
}

```

```

        graphics.beginShaderFill(shader);
        graphics.drawRect(20, 20, 800, 600);
        graphics.endFill();
    }
}
}

```

如果运行以上代码，你会得到一个错误信息：The Shader input src is missing or an unsupported type。因为 shader 正在尝试对一张不存在的图片进行取样。幸好设置 shader 的输入图片方法很简单。

首先要找到一张位图，我选择了一张我朋友的照片，把它嵌入到类里面：

```

package {
    import flash.display.Bitmap;
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.filters.ShaderFilter;
    public class ShaderFillImage extends Sprite
    {
        [Embed(source="TwirlFlash.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;
        [Embed(source="john_davey.jpg")]
        private var JohnDavey:Class;
        public function ShaderFillImage()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            var shader:Shader = new Shader(new ShaderClass());
            shader.data.center.value = [400, 300];
            shader.data.twist.value = [-6];
            shader.data.radius.value = [300];
            var image:Bitmap = new JohnDavey();
            shader.data.src.input = image.bitmapData;
            graphics.beginShaderFill(shader);
            graphics.drawRect(20, 20, 800, 600);
            graphics.endFill();
        }
    }
}

```

你也看到了，我们新建一个嵌入图片的实例 new JohnDavey()，并赋给 image 变量。把 image 的 bitmapData 属性值赋给 shader.data.src.input，这样 shader 就有图片可以取样了。效果如图 9-15：



图 9-15. The TwirlFlash shader used as a fill

注意，在 shader.data.src.input 里的这个 src 跟其它参数变量是截然不同，它来自 shader kernel 代码里的：

```
input image4 src;
```

如果我们 src 改为 source，那么在 ActionScript 里也要相应地改为：

```
shader.data.source.input = image.bitmapData;
```

以上所述的就是把 shader 用作填充时的用法，但实际上我也讲述了 Shader 对象的用法了。下面我将会介绍一下 shader 用作滤镜和混合模式。其实以上所讲的正好为下面的内容铺好了路。

### 使用 shader 作为滤镜

使用 shader 作为滤镜，说实话，比起用作绘制填充要简单得多了。再以 TwirlFlash shader 为例，为了说明 shader 可以应用于任何显示对象（包括文本框），这次就不使用图片来做测试了。这次的例子文件为 ShaderAsFilter.as：

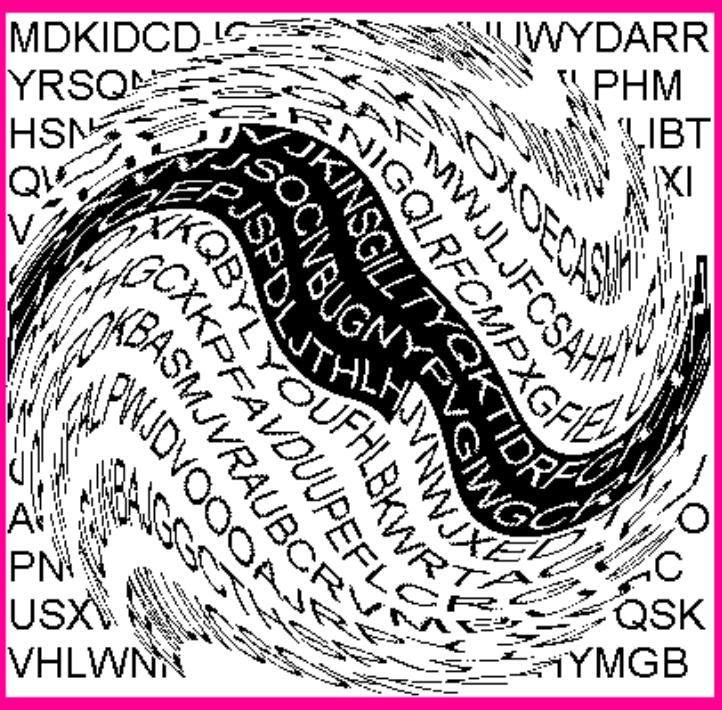
```
package {
    import flash.display.Shader;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.filters.ShaderFilter;
    import flash.text.TextField;
    import flash.text.TextFormat;
    [SWF(backgroundColor=0xFFFFFFFF)]
    public class ShaderAsFilter extends Sprite
    {
        [Embed(source="TwirlFlash.pbj", mimeType="application/octet-stream")]
        private var ShaderClass:Class;
        public function ShaderAsFilter()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
```

```

var shader:Shader = new Shader(new ShaderClass());
shader.data.center.value = [200, 200];
shader.data.twist.value = [-1];
shader.data.radius.value = [200];
var tf:TextField = new TextField();
tf.width = 400;
tf.height = 400;
tf.wordWrap = true;
tf.multiline = true;
tf.border = true;
tf.defaultTextFormat = new TextFormat(" Arial" , 24);
addChild(tf);
for(var i:int = 0; i < 340; i++)
{
    tf.appendText(String.fromCharCode(65 + Math.random() * 25));
}
tf.filters = [new ShaderFilter(shader)];
}
}

```

在这个例子里，创建了一个文本框，文本框的内容为一些随机的英文字母。最后那行是关键，我们把新建的 `ShaderFilter` 滤镜放在数组（中括号）里，然后赋给文本框的 `filters` 属性。没有任何工作比这个要简单了！如图 9-16（此图为译者添加的，原书没有的）：



[EXTRA] 图 9-16. Using a Shader as a Filter with TextField

#### 使用 `shader` 作为混合模式

最后这个话题比较与众不同。实际上，我们需要重新编写一个 `shader` 才能正确地用作混合模式。首先，要将 `shader` 用于混合模式，需要两个输入图片（`input image`）：一张用作背景图，另一张用作前景图。在 `kernel` 里你需要定义这两张图片，但在 Flash 里会自动完成这些工作：无论 `shader` 应用于哪个显示对象，该对象都会充当前景图；而位于前景图下面的那个显示对象（不管它是什么显示对象），都会充当为背景图。实际所做的工作是：当你对显示对象应用混合模式（`blend mode`）时，该对象的 `cacheAsBitmap` 属性都会被设置为 `true`。这样一来，显示对象充当了一张位图，然后传递给 `shader` 作为源图片（`source image`）了。

在 shader 里始终有一些限制。你不能像其它 shader 那样使用正弦波浪 (sine waves)、棋盘形图案和扭曲效果。在 Pixel Bender Toolkit 里使用是可以的，但如果你想应用在混合模式，不行！你只能做的就是对输入图片进行取样，然后想个不错的算法来产生你想要的效果。接下来的例子演示了这个道理 (ChannelBlend.pbk):

```
<languageVersion : 1.0;>
kernel ChannelBlend
< namespace : "com.friendsofed" ;
  vendor : "Advanced ActionScript 3.0 Animation" ;
  version : 1;
  description : "blends the channels of two images" ;
>
{
  input image4 back;
  input image4 fore;
  output pixel4 dst;
  parameter float3 amt;
  void
  evaluatePixel()
  {
    pixel4 bg = sampleNearest(back, outCoord());
    pixel4 fg = sampleNearest(fore, outCoord());
    dst.r = fg.r * amt.r + bg.r * (1.0 - amt.r);
    dst.g = fg.g * amt.g + bg.g * (1.0 - amt.g);
    dst.b = fg.b * amt.b + bg.b * (1.0 - amt.b);
    dst.a = 1.0;
  }
}
```

首先定义了两个 image4 类型的变量，fore 和 back。这两个变量起什么名字没关系。第一个变量会接收背景图，第二个变量接收前景图。

在 evaluatePixel() 函数里，我们对两张图片在 outCoord 点处进行取样（得到两个像素值）。这一步骤是把 shader 用作混合模式情况下的基本工作，之后你就可以用你的方式混合那两个像素了。在这个例子里，我根据 amt 参数的值来混合这两个像素的各自通道。为了能在 Pixel Bender Toolkit 里测试，你必须通过“File”菜单加载两张图片。第一张加载的图片为背景图，第二张加载的图片为前景图。你可以拉动 amt 参数的滚动条来测试一下效果。

现在回到 Flash 那头，文件名为 ShaderBlendMode.as:

```
package {
  import flash.display.Bitmap;
  import flash.display.Shader;
  import flash.display.Sprite;
  import flash.display.StageAlign;
  import flash.display.StageScaleMode;
  public class ShaderBlendMode extends Sprite
  {
    [Embed(source="ChannelBlend.pbj", mimeType="application/octet-stream")]
    private var ShaderClass:Class;
    [Embed(source="input1.jpg")]
    private var input1:Class;
    [Embed(source="input2.jpg")]
    private var input2:Class;
    public function ShaderBlendMode()
    {
```

```

stage.align = StageAlign.TOP_LEFT;
stage.scaleMode = StageScaleMode.NO_SCALE;
var back:Bitmap = new input2();
addChild(back);
var fore:Bitmap = new input1();
addChild(fore);
var shader:Shader = new Shader(new ShaderClass());
shader.data.amt.value = [1.0, 0.9, 0.0];
fore.blendShader = shader;
}
}
}

```

在这里嵌入了两张位图，然后创建它们的实例并添加到舞台(stage)上。还创建一个新的 Shader 对象，它引用了刚才新建的 ShaderBlend shader，设置一些变量值，然后把 shader 对象赋给位于上面的 bitmap 对象的 blendShader 属性。你应该会看到如下效果(图 9-17)：



图 9-17. Two images blended with the ChannelBlend shader

如果你曾使用过混合模式(blend mode)，你应该向 blendMode 属性赋给一个字符串。实际上，当你向显示对象的 blendShader 属性赋值时，blendMode 属性会自动被赋给一个“shader”值(BlendMode.SHADER)。

尝试一下更改参数的值，看一下是不是跟 Pixel Bender Toolkit 里测试时看到的效果一样。

#### 总结：

这一章讲述的不多，但至少足够让你学会编写 shader 和在 Flash 里应用 shader。现在，你学有所成了，你应该能够编写一些酷东西出来，我期待你的酷作品！

## 第十章 补间引擎

在 Making Things Move 的第八章，我曾经提到过缓动 (easing) 这个话题。这些缓动实现起来非常简单，同时功能和灵活性都有限，因此我称之为简单缓动。这种类型的缓动由一个对象和该对象将要移动到的目的地组成。我们在每一帧计算对象和目的地之间的距离并让对象移动到这段距离的中点（或者其他比例）。对象向前平滑移动，最终慢慢停在目的地。

在那一章我引用了 Robert Penner 的缓动方法，并建议大家仔细研究一下（我称它们为高级缓动）。当时我很想详细谈谈这个话题，可惜我们上本书的篇幅有限。现在我们终于可以聊聊它们了。Robert Penner 的一系列缓动方程式得到了广泛的应用，除了 Flash 和 Flex 自带的 Tween 类，他们还被用在了近几年涌现出的很多补间引擎里。

事先说明一下：从这儿开始，每当我在这第一章提到补间 (tween 或者 tweening)，我指的是使用代码（通常是预定义的补间类），在一定时间内移动一个对象或者改变一个对象的属性。这里的补间和时间轴上的补间动画毫无关系。

Adobe 官方发布的补间类有两个：Flash 里的 fl.transitions.Tween 类和 Flex 架构里的 mx.effects.Tween 类。这两个类的功能基本相同：他们根据给定的初始值和目标值，生成在一段时间内的一系列中间值。他们都是根据 Robert Penner 的缓动方程式建立的，允许用户实现复杂缓动，包括不同方程式实现的加速减速，弹性缓动，自定义缓动开始和结束时间等等功能。

补间引擎都是代码体系，大多数的补间引擎（同样使用 Robert Penner 的缓动方程）都是要使补间更容易实现。我也说不清现在的补间引擎到底有多少种。在开始写这本书之前我曾经罗列了当时主要的补间引擎，但是当我写到这章的时候，我不得不又在列表上加上了两三个后起之秀。

好，现在我们先说说这一章不讲什么。

首先，他不是现有的所有 AS 3.0 补间引擎的终极手册。与本书的很多其他章节一样，这些话题都够写一整本书。那样详细的写的话，当这本书出版的时候他就已经过时了。其实，如果你想了解更多关于 Flash 平台上的补间引擎的知识，可以看一看 The Essential Guide to Open Source Flash Development (Apress, 2008) 的第十章。在那一章里，Moses Gunesch 讲述了补间引擎的历史和他自己开发的 Fuse 补间工具包。之后他还介绍了他的新项目——GoASAP，一个编写补间引擎的架构。我本想在这一章写写，但是我觉得 Moses 在他的书中介绍的更好。

同时，这一章也不是“最好的补间引擎排行榜”，或者“我最喜爱的补间引擎”，或者对任何一款补间引擎的保证和推荐。这些引擎都有自己的长处，缺点和不同的工作原理。

最后，这一章不是我提到的任何一款补间引擎的完全手册。

这一章是对不同种类的补间引擎的一个概览和介绍，意在帮助你找到适合你需求的合适的引擎。我试图根据他们生成补间的方法的差别，找出各种类型的补间引擎，然后我逐个用他们完成一些基本的任务。这样你就能做一下比较，看到每种引擎的长处和缺点。除了 Adobe 提供的补间类，我还选取了以下几款补间引擎：

Tweener

TweenLite/TweenGroup

KitchenSync

gTween

我们从 Adobe 自带的引擎开始。

### Flash 的 Tween 类

我们提到过，Flash 自带的 Tween 类会在你安装 Flash 软件的时候一起被安装。它在 fl.transitions 包里（这个包里还有很多你可能还没听说过的能做出很多华丽转场效果的类）。但在这章里，我们只说 Tween 类。

我假设你使用的是 Flash 软件，fl.transitions 包是可以直接使用的。当然你也可以用 Flex Builder 或者其他基于 mxmlc 的方式编译下面的范例，但你要把 flash.swc 文件（可以在 Flash 安装目录找到）添加到你的类库路径里。我不会讲解具体怎么做，因为这种做法不是很规范，我也不

相信你们有人会这么做。

这个类的工作方式是：你创建一个它的实例，传给它一个对象的引用和这个对象的一个属性名；一个缓动方法（基于 Robert Penner 的缓动方程）；这个属性的初始值和目标值；还有持续时间。下面就是它的构造方法：

```
new Tween (object,property,easingFunction,begin,finish,duration,useSeconds)
```

参数中的 object 可以是任意至少拥有一个数字属性的对象。property 参数是一个字符，即你要改变的属性的名称。比如你要改变一个 Sprite 的 x 属性，这个参数就是 “x”（包括引号）。

easingFunction 参数是 flash.transitions.easing 包中定义好的某个缓动类的一个方法。我们很快就会讲到他们。

begin 和 finish 都是数值。当补间开始的时候，对象要变动的属性将被设置为 begin 这个值。补间结束后，该属性就会变成 finish 的值。

默认情况下，duration 参数给出的是补间运行的帧数。但最后一个参数 useSeconds 是一个布尔值。默认值是 false，表示补间会把 duration 的值当作帧数使用。但是如果你把最后一个参数设为 true，duration 的值就会被当作秒数计算。

我们来看看效果吧。第一个例子在文件 FlashTween.as 里，你可以从本书的网站 [www.friendsofed.com](http://www.friendsofed.com) 下载。（具体地址：）

```
package {
    import fl.transitions.Tween;
    import fl.transitions.easing.Regular;
    import fl.transitions.easing.Strong;

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

    [SWFBackgroundColor=0xffffffff]
    public class FlashTween extends Sprite
    {
        private var tween:Tween;
        private var sprite:Sprite;

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

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);
tween = new Tween(sprite,"x",None.easeIn,100,500,1,true);
        }
    }
}
```

粗体的代码就是生成补间的地方。请注意，补间一生成就开始工作了，你不用告诉他什么时候开始。实例中画着红色矩形的 Sprite 在 1 秒钟内从 x 坐标 100 的位置移动到 x 为 500 的位置。

### 缓动方法

在上个实例中，我们使用的缓动方法是 None.easeIn。你可能会问这是什么意思。缓动方法这

个属性决定用什么缓动方程来实现加速或减速效果。换句话说，如果你在做一个对象位移的补间，这个参数会决定它用多久从静止变为全速，然后用多久减速停止在目标位置。

在 fl.transitions.easing 包里有多个类：

- Back
- Bounce
- Elastic
- None
- Regular
- Strong

这些类都有三种方法：

- easeIn
- easeOut
- easeInOut

None 类还另外有一个 easeNone 方法。

easeIn 方法控制补间如何从开始到最高速度。

easeOut 方法控制补间减速并停在目标位置。

easeInOut 方法同时控制上述两者。

这些方法在内部被 Tween 类自动调用。你不需要亲自执行他们。只要把方法当作一个参数传给补间的构造函数就行了。这些方法里使用的就是 Robert Penner 的方程式。

上面的实例中使用了缓动类 None 中的 easeIn 方法。这个类会创建一个线性缓动，实际上这根本不是缓动。补间会以最高速度开始，并保持这一速度直到在终点戛然而止。实际上，你使用 None 类的哪种方法都不重要，他们的结果都是一样的。

好了，这些还不够刺激。我们看看其他方法能做些什么吧。我们先试试 Regular。这个类创建的缓动效果跟你在 Flash 时间线上做补间动画时，把面板上的 ease in 或 ease out 的选项调到 100% 是一模一样的。我们把生成补间的那行代码改为：

```
tween = new Tween(sprite, "x", Regular.easeIn, 100, 500, 1, true);
```

现在你会看到 sprite 缓缓开始运动，加速，突然停止在终点。如果换用 Regular.easeOut，你会看到它一开始就快速运动，慢慢减速，停在目标位置，大体上跟 Making Things Move 里介绍的简单缓动效果一样。试试 Regular.easeInOut，它包含了加速和减速。这种效果就比较专业了，适用于各种界面元素的移动效果。当然，还要试试改变其他的参数。改改 duration 或者 start 和 finish 参数。试着给别的属性做一个补间，比如 y 或者 rotation。试试把 alpha 从 0 变到 1，像这样：

```
tween = new Tween(sprite, "alpha", Regular.easeInOut, 0, 1, 1, true);
```

缓动类 Strong 做的事情跟 Regular 一样，只不过效果更显著。Elastic 类在加速和减速时会产生类似弹簧的效果。把 duration 参数改大一点你会看得更清楚：

```
tween = new Tween(sprite, "x", Elastic.easeInOut, 100, 800, 3, true);
```

Bounce 类也会产生弹性效果，但是像一个物体在坚硬表面上反弹的效果。最后介绍 Back 类，它在补间开始前会让物体向反的方向运动一段，然后冲过目标点，再移动回来。现在你已经了解缓动类了——至少是 Flash 里 Tween 类的缓动。

## 合并补间

第一个实例运行的非常好，但是它只改变了一个对象的一个属性。如果你只需要改变某个对象的单个属性，而且你的补间都是相互独立的，那你的目的已经达到了。但是你常常需要改变对象的多个属性（至少是 x, y 坐标）。有时你还需要在一个补间完成以后马上启动另一个补间。

多个补间同时发生被称作并行补间（parallel tweens）或组合，一个接一个发生的补间被称作补间序列（tween sequences）。

并行补间在 Flash Tween 类里非常容易实现，只要为你要改变的对象的相应属性生成补间就好了。比如说你需要一个对象同时沿 x 轴和 y 轴移动，那就做这样两个补间：

```
tween1 = new Tween(sprite, "x", Regular.easeInOut, 100, 800, 3, true);
```

```

tween2 = new Tween(sprite, "y", Regular.easeInOut, 100, 400, 3, true);
你需要多少补间就可以制造多少，他们不用使用相同的缓动方法或者同样的时间。就像这样：
tween1 = new Tween(sprite, "x", Regular.easeInOut, 100, 800, 3, true);
tween2 = new Tween(sprite, "y", Regular.easeInOut, 100, 400, 3, true);
tween3 = new Tween(sprite, "rotation", Strong.easeInOut, 0, 360, 4, true);
注意旋转补间(tween3)使用了Strong缓动类并且历时4秒钟，而x,y轴的补间使用了Regular缓动类和3秒钟。另外还要注意，在需要多个补间的情况下，我定义了新的属性来保存这些补间。

```

```

private var tween1:Tween;
private var tween2:Tween;
private var tween3:Tween;

```

序列要更复杂一点。你要等前一个补间结束再开始下一个。要实现这个操作，你就要监听告诉你补间何时结束的事件(TweenEvent.MOTION\_FINISH事件)。假设你想要刚才的3个补间一个接一个的执行，而不是同时开始。下面的这个类，在文件FlashTweenSequence.as文件里，给出一个简单的实例：

```

package {
    import fl.transitions.Tween;
    import fl.transitions.TweenEvent;
    import fl.transitions.easing.Regular;
    import fl.transitions.easing.Strong;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    [SWF(backgroundColor=0xffffffff)]
    public class FlashTweenSequence extends Sprite
    {
        private var tween1:Tween;
        private var tween2:Tween;
        private var tween3:Tween;
        private var sprite:Sprite;
        public function FlashTweenSequence()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;
            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);
            tween1 = new Tween(sprite, "x", Regular.easeInOut, 100, 800, 3, true);
            tween1.addEventListener(TweenEvent.MOTION_FINISH, onTween1Finish);
        }
        private function onTween1Finish(event:TweenEvent):void
        {
            tween2 = new Tween(sprite, "y", Regular.easeInOut, 100, 400, 3, true);
            tween2.addEventListener(TweenEvent.MOTION_FINISH, onTween2Finish);
        }
        private function onTween2Finish(event:TweenEvent):void
        {
            tween3 = new Tween(sprite, "rotation", Strong.easeInOut, 0, 360, 4, true);
        }
    }
}

```

```
}
```

这里我们创建了第一个补间后就立即添加了对 MOTION\_FINISH 时间的监听器。当他结束时，我们创建了第二个补间并监听相同的事件。事件激发后我们又创建了第三个补间。我的这个例子很简单，但是在更严谨的项目中你还需要管理这些监听器，在不需要他们的时候删除它们，等等。

如果你有一个非常复杂的界面，有很多东西动来动去，你不得不创建非常多的事件处理器从而使跟踪他们之间的逻辑成为一场噩梦。当然，还有达到上述目的的其他方法，比如只创建一个事件处理器，然后通过判断事件的目标（target）来执行相应的操作。但无论怎么做，都会越来越复杂。

正是为了解决很多这样的问题，我们下文会提到的众多第三方补间引擎才应运而生。在我们介绍他们之前，先简单看一下 Flex 架构里带的 Tween 类吧。

### Flex Tween 类

虽然 Flex Tween 类是 Flex 架构的一部分，你并不一定要创建一个 Flex 项目才能使用它。你可以直接在 ActionScript 3.0 项目里，甚至是 Flash CS4 生成的影片里使用它。当然，要想在 Flex 以外的项目里使用它，你需要把 framework.swc 文件添加到你的类库路径里。这个 SWC 文件就在你的 Flex 安装目录里，路径是 frameworks/libs/framework.swc。

在 Flex Builder 里，你要通过项目属性面板>ActionScript Build Path 项>Library Path 部分来添加 SWC 文件。（见图 10-1）

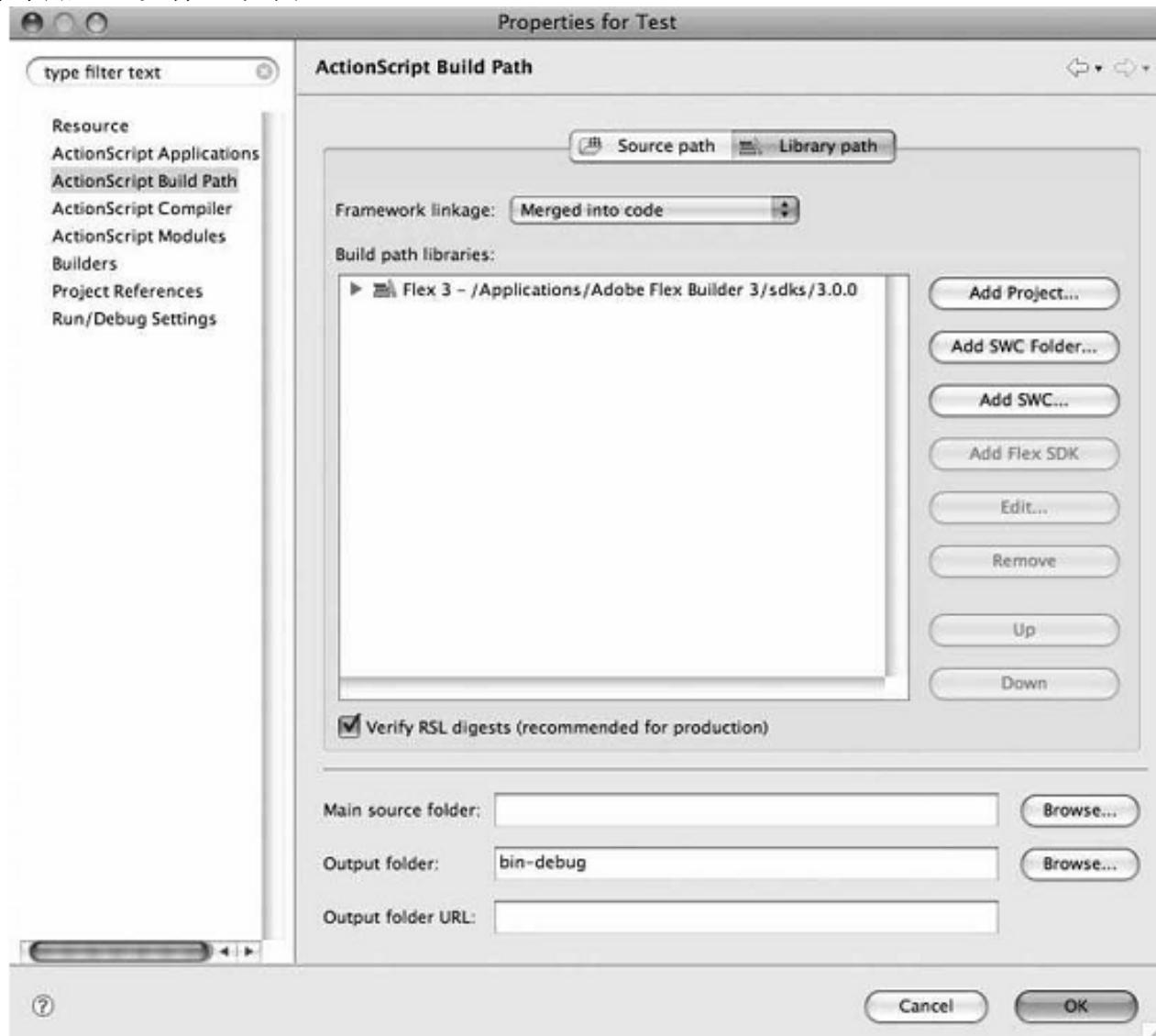


图 10-1 在对话框中添加 SWC 文件

点击 添加 SWC 然后找到 framework.swc 文件。见图 10-2

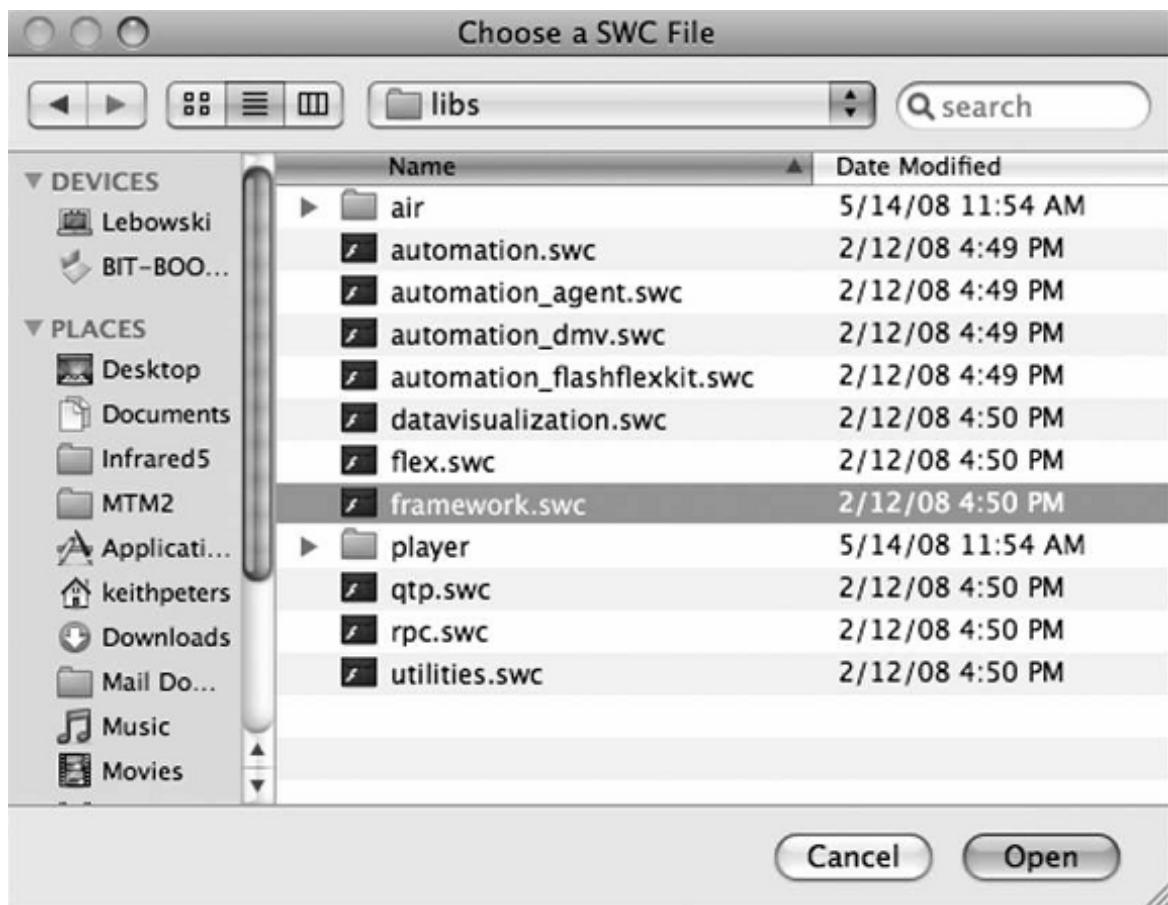


图 10-2 添加文件

在 Flash CS4 里，打开发布选项面板，点击 ActionScript 3.0 设置按钮，找到库路径部分。如图 10-3

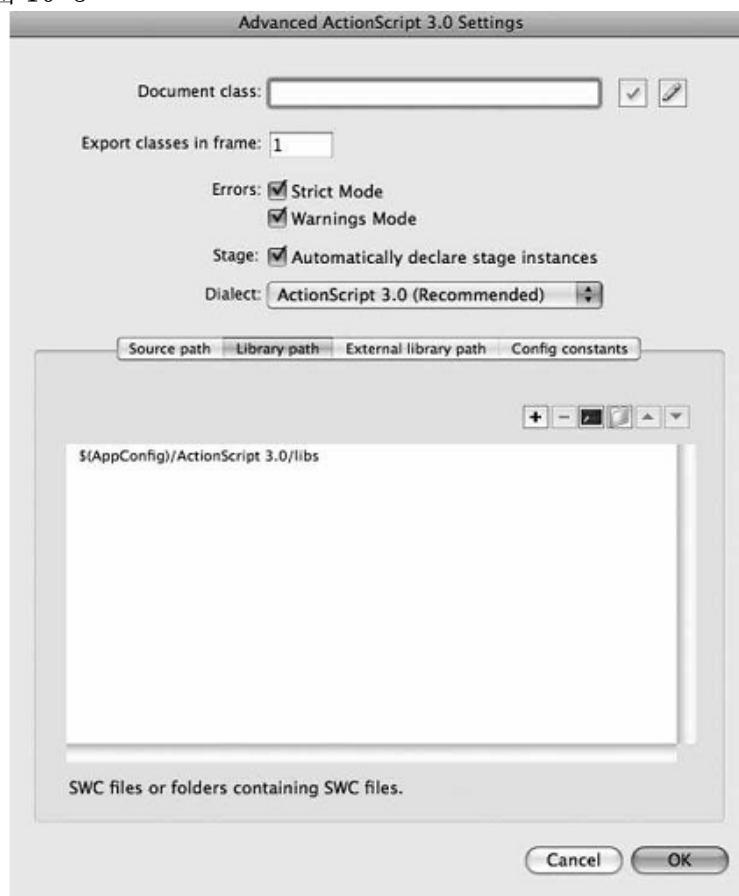


图 10-3 库路径部分

点击图标“+”来添加新的列表项目，然后点击红色瞄准镜图标来找到 framework.swc 文件。完成后的面板会像 10-4 这样。



图 10-4 添加文件后的面板

把这个 SWC 文件添加到你的项目并不会把整个 Flex 架构打包进你生成的 SWF 文件，只会包含你代码里用到的类和与它有关的类（就 Tween 类而言，不是很多）。

Flex 的 Tween 类与 Flash 里的有一点不同。你不用把对象和要改变的属性直接传给补间，而只要给出一个监听类，初始值和目标值，还有时间。监听类需要有在补间更新和完成时被调用的方法，他们被命名为 onTweenUpdate 和 onTweenEnd。二者都要接受类型为 Object 的唯一参数。下面就是 mx.effects.Tween 的完整构造函数：

```
new Tween(listener, startValue, endValue, duration, minFps, updateFunction, endFunction);
```

参数 minFps 是可选的，它设定补间每秒运行的最低次数。大多情况下你可以使用默认值 -1。

updateFunction 和 endFunction 也是可选属性，你可以使用他们传入代替 onTweenUpdate 和 onTweenEnd 的函数名。默认情况下，如果你有多个补间同时运行，而且他们使用同一个监听器，他们在更新和结束时会调用相同的 onTweenUpdate 和 onTweenEnd 方法。利用这两个参数就可以给不同的补间指定不同的方法。

下面的实例将告诉你在一个简单动画里怎么使用 Flex Tween 类。这个类在文件 FlexTween.as 里。

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import mx.effects.Tween;
    [SWF(backgroundColor=0xffffffff)]
    public class FlexTween extends Sprite
```

```

{
    private var tween:Tween;
    private var sprite:Sprite;
    public function FlexTween()
    {
        stage.scaleMode = StageScaleMode.NO_SCALE;
        stage.align = StageAlign.TOP_LEFT;
        sprite = new Sprite();
        sprite.graphics.beginFill(0xff0000);
        sprite.graphics.drawRect(-50, -25, 100, 50);
        sprite.graphics.endFill();
        sprite.x = 100;
        sprite.y = 100;
        addChild(sprite);
        tween = new Tween(this, 100, 800, 1000);
    }
    public function onTweenUpdate(value:Object):void
    {
        sprite.x = value as Number;
    }
    public function onTweenEnd(value:Object):void
    {
        sprite.x = value as Number;
    }
}

```

这里我们创建了一个新的补间，并传入 `this` 作为监听器。这样做是让这个类自己监听 Flex Tween 类的更新和结束事件。我们把初始值设为 100，结束值 800，时间为 1000。这里是 Flex 的 Tween 类与 Flash 的另一个不同之处。虽然 Flash 的 Tween 类默认使用帧数来计算长度，但我们可以强制他使用秒数，而 Flex 则使用毫秒来计算时长。（1000 毫秒就是 1 秒）

还要注意起始和目标值都是简单的数值，他们不属于某个属性，位置或其他。他们的改变通过更新和结束事件被传出，至于怎么使用这些中间值就是你自己的事情了。

在事件处理方法里，这些值被以 `Object` 的类型传出，而不是 `Number`，你很快就会明白为什么这么处理。当 `onTweenUpdate` 方法第一次被调用时，传出的数值比初始值大一点点。当它最后一次被调用时，传出的值比目标值小一点点。当最终 `onTweenEnd` 被调用时，返回的值等于目标值。在这里我们把得到的对象映射为 `Number` 值并把它赋给 `sprite` 的 `x` 属性。比 Flash 的 Tween 类复杂一点，但同时它也更强大。

### Flex Tween 类的缓动函数

像他的 Flash 兄弟一样，Flex Tween 类也允许你指定一个缓动方法。然而在 Flex 里你要在补间实例创建之后再做这件事，而不是在构造函数里。与 Flash 类似，这些缓动方法都是一组缓动类的方法（`easeIn`, `easeOut`, `easeInOut`）。这些类在包 `mx.effects.easing` 里，他们分别是：

- Back**
- Bounce**
- Circular**
- Cubic**
- Elastic**
- Exponential**
- Linear**
- Quadratic**

Quartic  
Quintic  
Sine

相对于 Flash 里的缓动类，这些类更贴近数学原理。其中 Back, Bounce 和 Elastic 跟 Flash 里同名的类一模一样。Linear 相当于 Flash 里的 None。剩下的还有 Circular, Cubic, Exponential, Quadratic, Quartic, Quintic 和 Sine。可能说明这些类最简单的办法就是让你去看看 Robert Penner 的网站。他有一个缓动方程演示器，你可以选择不同的方程预览效果。他还给你一个时间和数值变化的图表。这个演示的 URL 是：

[http://www.robertpenner.com/easing/easing\\_demo.html](http://www.robertpenner.com/easing/easing_demo.html)

顺便说一下，如果你不指定缓动函数，他将默认使用 Sine.easeInOut。现在我们试试别的。在你创建一个补间之后，马上指定一个缓动函数：

```
tween = new Tween(this, 100, 800, 1000);
tween.easingFunction = Linear.easeIn;
```

别忘了 import 任何你用到的缓动类。上例中我只试了 Linear，他跟 Flash 里的 None 一样。因此这将产生一个从起点到终点，以恒定速度运动的补间动画。试试其他的，看看他们怎么工作。下面是 Quintic.easeInOut，他有着非常陡峭的加速曲线：

```
tween.easingFunction = Quintic.easeInOut;
```

他与速度变化较缓和的 Sine 和 Quadratic 明显不同。

## Tween 组合

在 Flash Tween 类里，如果你想改变某个对象的多个属性，你就要定义多个补间。在 Flex 版本里你可以用一个补间改变多个属性。方法是你要用数组来定义开始和结束值，而不是单个值。现在你就能理解为什么这些属性和 update, end 事件处理器接受的值都是 Object 类型了吧？这样你就可以根据需要使用数字或数组了。下面的类在文件 FlexTweenXY.as 里。

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import mx.effects.Tween;
    import mx.effects.easing.Quintic;
    [SWF(backgroundColor=0xffffffff)]
    public class FlexTweenXY extends Sprite
    {
        private var tween:Tween;
        private var sprite:Sprite;
        public function FlexTweenXY()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;
            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);
            tween = new Tween(this, [100, 100], [800, 400], 1000);
        }
        public function onTweenUpdate(value:Object):void
        {
            sprite.x = value[0];
            sprite.y = value[1];
        }
    }
}
```

```

        }
    public function onTweenEnd(value:Object):void
    {
        sprite.x = value[0];
        sprite.y = value[1];
    }
}
}

```

在构造函数里我们使用了包含两个数值的数组来定义初始值和目标值。同样，他们没有什么意义，只不过是两个数值。缓动类会平滑的改变数组里的每个值，改变后的数值也会以数组的形式传给处理器。在处理器函数里，我们可以以任意的方式解释这些值。在这个例子里，我将用第一个值来定义 x 位置，第二个值定义 y 位置。所以我会让 sprite 在 x 轴从 100 到 800，在 y 轴从 100 到 400。这样我们就生成了一个二维的补间。

另一个方法是把 0 和 1 作为补间的起始值，然后在处理器里计算得出结果。就像这样：

```

package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import mx.effects.Tween;
    [SWF(backgroundColor=0xffffffff)]
    public class FlexTweenXY extends Sprite
    {
        private var tween:Tween;
        private var sprite:Sprite;
        public function FlexTweenXY()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;
            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);
            tween = new Tween(this, 0, 1, 1000);
        }
        public function onTweenUpdate(value:Object):void
        {
            sprite.x = 100 + 700  (value as Number);
            sprite.y = 100 + 300  (value as Number);
        }
        public function onTweenEnd(value:Object):void
        {
            sprite.x = 100 + 700  (value as Number);
            sprite.y = 100 + 300  (value as Number);
        }
    }
}

```

属性 value 在这个例子里将是 0 到 1 之间的某个值。我们用它乘以我们想让他运动的距离，然后加上他初始的位置。这样做不像直接对起始和结束值进行补间那样简单明了，但是这样做在你需要灵活性的时候会非常有用。比如，你想要一个物体淡入，同时另一个物体淡出。就可以做一个

0 到 1 的补间然后在处理器里这样做：

```
public function onTweenUpdate(value:Object):void
{
    obj1.alpha = value as Number;
    obj2.alpha = 1.0 - value as Number;
}
```

当 value 的值从 0 到 1 时，obj1 的 alpha 值也跟着变化，而 obj2 的 alpha 值将从 1 变到 0。一个补间，两个渐变。

## Tween 序列

不幸的是，在 Flex 里创建补间序列并不比 Flash 里简单多少。在 mx.effects 包里确实有创建并行或序列补间的类，但他们更依附于 Flex 架构本身，而且是专门用于基于 Flex 的应用里的。不像 Tween 类可以很轻松的单独使用。

在使用 Flex Tween 类创建序列时，你需要等待 onTweenEnd 处理器被激发；如果你指定了其他的处理器，你同样需要等他被激发。下一个类将演示这种做法，详见 FlexTweenSequence.as 文件。这个类基本上复制了前面讲过的 FlashTweenSequence 类的功能。关键点已经加粗：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import mx.effects.Tween;

    [SWF(backgroundColor=0xffffffff)]

    public class FlexTweenSequence extends Sprite
    {

        private var tween1:Tween;
        private var tween2:Tween;
        private var tween3:Tween;
        private var sprite:Sprite;

        public function FlexTweenSequence()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;
            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            tween1 = new Tween(this, 100, 800, 3000, -1, onTween1Update, onTween1End);
        }

        public function onTween1Update(value:Object):void
        {
            sprite.x = value as Number;
        }

        public function onTween1End(value:Object):void
    }
}
```

```

{
    sprite.x = value as Number;
    tween2 = new Tween(this, 100, 400, 3000, -1, onTween2Update, onTween2End);
}
public function onTween2Update(value:Object):void
{
    sprite.y = value as Number;
}

public function onTween2End(value:Object):void
{
    sprite.y = value as Number;
    tween3 = new Tween(this, 0, 360, 4000, -1, onTween3Update, onTween3End);
}
public function onTween3Update(value:Object):void
{
    sprite.rotation = value as Number;
}

public function onTween3End(value:Object):void
{
    sprite.rotation = value as Number;
}
}
}

```

第一个补间更新处理器改变 x 值，第二个改变 y 值，第三个 sprite 的旋转角度。当前一个补间结束时，他将创建下一个补间。

这段代码不算漂亮，只是为了演示相对简单的三段动画。一些华丽的界面可能用了很多这类同步或者序列的补间动画。当然，我不会放弃对代码的优化。你可以对代码进行很多改进或者应用多种设计模式是代码更清晰，更简练，甚至能重复使用。这样研究下去，你将得到一个补间引擎，就像我下面要介绍的。

## 补间引擎

到此为止，我想你已经知道补间引擎是个好东西了：他让编写改变多个对象的多个属性的多个补间的代码更容易管理。再次重申，我挑选了一些不同的补间引擎，不是因为他们是我最喜欢的或者“最好的”，只是因为他们很受欢迎，也是为了展示解决问题的不同方法。他们中有些是针对开发人员设计的，你需要写很多代码才能实现功能。其他的更针对设计师，更容易使用。但这些编码上的便利会让一个真正的开发人员想哭：无类型的参数，通用对象，“魔法咒语”，回调函数替代了事件等等。使用哪个引擎取决于哪个更适合你和你团队的工作流程，哪个能提供你需要的功能。

再说一下我们要研究的补间引擎：

- Tweener
- TweenLite/TweenGroup
- KitchenSync
- gTween

我们先从 Tweener 开始。

## Tweener 的缓动函数

缓动函数也是在 tweeningParameters 对象里设置的。毫无疑问，他们还是基于 Robert Penner 的方程式。然而缓动类和方法都被隐藏了，你只需要传递一个字符描述你所需要的缓动类型，就像这样：

```
Tweener.addTween(sprite, {x:800, time:3, transition:"easeInOutCubic"});
```

需要完整的缓动类型列表请参考 Tweener 的说明文档。

同样，这种做法会让设计人员更容易使用，毕竟记住一个缓动的名字要比记住类路径，类名和方法名称容易的多。但是如果你意外的写错了名字会怎么样？比如“easeIn0utCubix”？项目还是会顺利编译，甚至还会顺利执行，但是缓动方程会被自动换为默认类型，“ease0utExpo”。这一错误你可能不会注意，直到几周以后你的客户对你说界面上的一些东西看起来不像以前那样好了。

我不是说因为这些原因你就不应该使用 Tweener。我只是认为这些是人们在评价不同的 Tween 引擎时应该注意的问题。（注意：使用这种语法的引擎不止 Tweener 一家。）

### Tweener 的 tween 组合

Tweener 是按对象添加渐变的，所以你要是想给多个对象添加渐变，就要做多个渐变。

```
Tweener.addTween(spriteA, {x:"800", time:3});
```

```
Tweener.addTween(spriteB, {x:"300", time:3});
```

个人认为这一不便之处在一方面得到弥补：你可以很方便的给一个对象的不同属性添加渐变。

### Tweener 的 tween 序列

使用 Tweener 的另一个好处是它让你更方便的创建序列。事实上 Tweener 有两种方法让一个渐变在另一个渐变结束后执行。首先，你可以在 tweeningParams 对象里设置一个 onComplete 回调函数。这就是一个函数的引用，这个函数会在渐变结束后被调用。就像这样：

```
Tweener.addTween(spriteA, {x:"800", time:3, onComplete:tweenEnd});
```

这里，tweenEnd 就是对渐变结束后将被调用的函数的引用。所以一个渐变结束后，你往往需要添加另一个。我做了下面的例子，你可以下载为 TweenerSequence.as 文件：

```
package {
    import caurina.transitions.Tweener;

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

    [SWF(backgroundColor=0xffffffff)]
    public class TweenerSequence extends Sprite
    {
        private var sprite:Sprite;

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

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

Tweener.addTween(sprite,
    {x:800, time:3,
    transition:"easeInOutCubic", onComplete:tween1End});
    }
}
```

```

private function tween1End():void
{
    Tweener.addTween(sprite,
    {y:400, time:3,
    transition:"easeInOutCubic",
    onComplete:tween2End});
}

private function tween2End():void
{
    Tweener.addTween(sprite,
    {rotation:360,time:4, transition:"easeInOutCubic"});
}
}
}

```

当一个渐变完成时，tween1End 会被调用，他会创建另一个渐变。当这个渐变也结束后，tween2End 会被调用，创建最后一个渐变。这种做法比在 Adobe Tween 类里创建序列要简单一点，而 Tweener 还提供了另一个更简便的方法：delay 属性。

delay 是另一个能被添加到 tweeningParameters 的属性。这是个以秒数为单位的数字，Tweener 会在这个时间后开始执行。这样你就可以做一些像下面一个范例 (TweenerDelay.as) 中的操作：

```

package {
    import caurina.transitions.Tweener;

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

    [SWF(backgroundColor=0xffffffff)]
    public class TweenerDelay extends Sprite
    {
        private var sprite:Sprite;

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

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            Tweener.addTween(sprite,
                {x:800, time:3,
                transition:"easeInOutCubic"});
            Tweener.addTween(sprite,
                {y:400, time:3,
                transition:"easeInOutCubic", delay:3});
            Tweener.addTween(sprite,
                {rotation:360, time:4,

```

```

        transition:"easeInOutCubic", delay:6});
    }
}
}

```

这里你直接创建所有的渐变。第一个渐变会持续 3 秒钟，因此我们要第二个渐变等待这么长时间后再开始。同样道理，我们要求最后一个渐变等待 6 秒。结果是这三个渐变完美的按序列执行。

一个需要小心的情况是用 `delay` 给同一个对象的同一属性添加两个渐变的时候。只要第二个渐变的延迟时间不短于第一个渐变的执行时间，两个渐变都会被执行。下面这个例子里，`sprite` 会移动到 x 轴 800 的位置，然后移动到 100。

```

Tweener.addTween(sprite,{x:800, time:3});
Tweener.addTween(sprite,{x:100, time:3, delay:3});

```

但是如果第二个渐变的延迟不能让第一个渐变结束，而出现时间上的交集，第一个渐变就会被第二个渐变覆盖，根本不会被执行。

```

Tweener.addTween(sprite,{x:800, time:3});
Tweener.addTween(sprite,{x:100, time:3, delay:2.5});

```

在这里，`sprite` 会静止 2.5 秒然后移动到 x 轴 100 的位置。

到这里你应该对 Tweener 是个什么样的东西有个大致的了解了。就像你看到的，它提供了一些很好的捷径，让你可以用简短的代码和易读的语法创建复杂的渐变效果。它还有很多其他有用的功能——只要你读读他的说明文档，就在我前面给出的网站上。一些高级开发人员可能会对 Tweener 松散的数据类型管理有点不满，但对大多数人来说，Tweener 是对 Adobe Tween 类的一大改进。

### TweenLite/TweenGroup

下面介绍的包涵盖了 [www.greensock.com](http://www.greensock.com) 的 Jack Doyle 写的一些类。你可以在 <http://blog.greensock.com/tweenliteas3/> 找到代码，然后像 Tweener 一样，只要把最高层的文件夹 `gs` 放到你的项目文件夹，或者其他你的编译器能找到的地方。主类在这个包的最高级：`gs.TweenLite`。

如果你读过了前面的部分或者对 Tweener 非常了解，TweenLite 对你来说会非常简单。据他的作者讲，他的初衷是让 TweenLite 比其他 Tween 引擎更精炼，更快速而且更有效率。这一章还介绍了一下 TweenGroup，它专注于制造并行的渐变，渐变序列和其他有关连的渐变组合。

像 Tweener 一样，你可以通过调用 TweenLite 的静态方法创建渐变。在下面的例子里我们用到了 TweenLite.`to`：

```
TweenLite.to(sprite, 3, {x:800});
```

这里，你在第一个参数处传入目标对象，第二个参数传入渐变的时间。第三个参数是 `variables`，像 Tweener 的 `tweeningParameters` 对象一样，它是一个 `Object` 类型的对象包含了你想要渐变的属性。所以上面的例子会在 3 秒钟内把目标对象移动到 x 值为 800 的地方。

调用 `to` 方法会返回 TweenLite 类的一个实例。然后你就可以用这个实例对渐变进行进一步控制：

```
var tween1:TweenLite = TweenLite.to(sprite, 3, {x:800});
```

而且你可以更直观的创建 TweenLite 的一个新的实例。构造函数的语法跟 `to` 方法一样：

```
var tween1:TweenLite = new TweenLite(sprite, 3, {x:800});
```

让我们把这些都放在一个能运行的类里面（见 `TweenLiteDemo1.as` 文件）：

```
package {
```

```

import flash.display.*;
import gs.TweenLite;
[SWF(backgroundColor=0xffffffff)]
```

```

public class TweenLiteDemo1 extends Sprite
{
    private var sprite:Sprite;
private var tween:TweenLite;

    public function TweenLiteDemo1()
    {
        stage.scaleMode = StageScaleMode.NO_SCALE;
        stage.align = StageAlign.TOP_LEFT;

        sprite = new Sprite();
        sprite.graphics.beginFill(0xff0000);
        sprite.graphics.drawRect(-50, -25, 100, 50);
        sprite.graphics.endFill();
        sprite.x = 100;
        sprite.y = 100;
        addChild(sprite);

tween = new TweenLite(sprite, 3, {x:800, y:400, rotation:360});
    }
}
}

```

粗体的是关键的语句。通过 variables 对象，你可以很方便的渐变多个属性：

```
tween = new TweenLite(sprite, 3, {x:800, y:400, rotation:360});
```

像 Tweener 一样，这样也会造成写错属性名称或给错属性的数据类型等错误，所以要小心。

### TweenLite 的缓动函数

这里是 TweenLite 更贴合程序员的地方。TweenLite 没有使用代表缓动函数的魔术字符，他使用的是真正的类名，像 Adobe 的 Tween 类一样。

毫无疑问，这些缓动类都是直接基于 Robert Penner 的缓动方程的（你还没向他鞠躬致敬吗？）。他们在 gs.easing 包：

```

Back
Bounce
Circ
Cubic
Elastic
Expo
Linear
Quad
Quart
Quint
Sine
Strong

```

每个都有三个方法：easeIn，easeOut，和 easeInOut。像 Adobe 的 Tween 类一样，你要通过设置类和方法选择缓动的类型：

```
tween = new TweenLite(sprite, 3, {x:800, y:400, rotation:360, ease:Elastic.easeInOut});
```

请确保导入你需要的所有缓动类，或者整个 gs.easing.\*包。

另一种方法。因为你现在有了一个 tween 的引用，你可以这样设置缓动类型：

```
tween = new TweenLite(sprite, 3, {x:800, y:400, rotation:360});
tween.ease = Elastic.easeInOut;
```

### TweenLite 的 tween 组合

同样，渐变多个对象的需要制造多个渐变实例：

```
tween1 = new TweenLite(spriteA, 3, {x:800});
```

```
tween2 = new TweenLite(spriteB, 3, {x:100});
```

这样就把一个 sprite 移动到 x 轴 800 的位置，同时把另一个移动到 x 轴 100 的位置。

然而，你还可以使用 TweenGroup 来更简明的实现这个效果。TweenGroup 类也在 gs 包里面，在你安装 TweenLite 的同时应该已经被安装了。这是一个很强大的类，有多种使用方法。如果你要改变多个对象的相同的属性，你就能使用 TweenGroup. allTo 的方法一次实现。他的工作方式很像 TweenLite. to 方法，只是第一个参数需要的不只是一个对象，而是一个对象数组。

```
TweenGroup.allTo([s1,s2,s3,s4], 3, {x:800});
```

这里， s1, s2, s3, s4 (代表 4 个不同的 sprite) 都会在 3 秒钟内移动到 x 为 800 的位置。

下面是一个能实际运行的类 (见 TweenGroupDemo1. as)

```
package {
```

```
    import flash.display.*;
```

```
    import gs.TweenGroup;
```

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

```
public class TweenGroupDemo1 extends Sprite
```

```
{
```

```
    private var sprite:Sprite;
```

```
    private var group:TweenGroup;
```

```
    public function TweenGroupDemo1()
```

```
{
```

```
        stage.scaleMode = StageScaleMode.NO_SCALE;
```

```
        stage.align = StageAlign.TOP_LEFT;
```

```
        var s1:Sprite = makeSprite(100, 100, 0xff0000);
```

```
        var s2:Sprite = makeSprite(100, 200, 0x00ff00);
```

```
        var s3:Sprite = makeSprite(100, 300, 0x0000ff);
```

```
        var s4:Sprite = makeSprite(100, 400, 0xffff00);
```

```
TweenGroup.allTo([s1, s2, s3, s4], 3, {x:800});
```

```
}
```

```
    private function makeSprite(xpos:Number, ypos:Number, color:uint):Sprite
```

```
{
```

```
        var s:Sprite = new Sprite();
```

```
        s.graphics.beginFill(color);
```

```
        s.graphics.drawRect(-50, -25, 100, 50);
```

```
        s.graphics.endFill();
```

```
        s.x = xpos;
```

```
        s.y = ypos;
```

```
        addChild(s);
```

```
        return s;
```

```
}
```

```
}
```

我添加了一个 makeSprite 方法来方便的创建多个 sprite，然后在一条语句里设置他们的位置和颜色。我们做了 4 个 sprite，并把它们纵向排在一起。然后我们把它们传给 TweenGroup. allTo 来让他们开始运动。

注意 TweenGroup. allTo 返回一个 TweenGroup 的实例，你能借此跟踪他。

## TweenLite/TweenGroup 的序列

我认为 TweenLite 和 TweenGroup 在序列方面的能力让这个引擎显得格外耀眼。

和 Tweener 一样，TweenLite 有一个 delay 属性可以用来延迟一个 tween。但是他有显著的不同之处，给了你更多的灵活性。首先，让我们回到只有一个 sprite 的实例，我们要创建这样的 3 个排成序列的 tween。（TweenLiteSequence1.as）

```
package {

    import flash.display.*;
    import gs.TweenLite;
    import gs.easing.Elastic;

    [SWF(backgroundColor=0xffffffff)]
    public class TweenLiteSequence1 extends Sprite
    {
        private var sprite:Sprite;
        private var tween1:TweenLite;
        private var tween2:TweenLite;
        private var tween3:TweenLite;

        public function TweenLiteSequence1()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

tween1 = new TweenLite(sprite, 3, {x:800});
tween2 = new TweenLite(sprite, 3, {y:400, delay:3});
tween3 = new TweenLite(sprite, 3, {rotation:360, delay:6});
        }
    }
}
```

当你运行这段代码时，前 6 秒钟什么也不会发生，然后最后一个 tween 会被执行。在 TweenLite 里，一个对象上的 tween 默认状态下都会覆盖其他同一对象上的 tween——即便你是在改变不同的属性，而且也给足了延迟时间。你应该记得，Tweener 只会覆盖控制相同属性而且时间上有重叠的两个 tween。

幸运的是，你可以通过 variables 对象的 overwrite 属性来控制这一行为。你可以使用 0 到 4 的数字，来实现以下目的：

0: NONE 模式。所有 tween 都不能被覆盖。

1: ALL 模式。所有 tween 都会被同一对象上的 tween 覆盖。

2: AUTO 模式。这种模式很像 Tweener 的工作方式。只有时间上有重叠的同一属性的 tween 才会被覆盖。

3: CONCURRENT 模式。覆盖所有时间上有重叠的 tween。不考虑他们在控制的属性。

不幸的是，这里“时间上重叠”的概念并不考虑延迟的时间。所以上例中，即使你给 tween2, tween3 的覆盖模式设为 2 或 3，结果还是会导致只有最后一个 tween 被执行，因为前两个 tween 会被覆盖。

所以为了修复上面的代码，我们要把 overwrite 属性设置为 0，这样将保证没有 tween 被覆盖。

```
tween1 = new TweenLite(sprite, 3, {x:800});
tween2 = new TweenLite(sprite, 3, {y:400, delay:3, overwrite:0});
tween3 = new TweenLite(sprite, 3, {rotation:360, delay:6, overwrite:0});
```

注意在 TweenLite 的包里还有一个叫做 OverwriteManager 的类，把这些模式定义成了静态属性。(就是，OverwriteManager.NONE, OverwriteManager.ALL 等等) 尽管这样有些罗嗦，它确实增进了编译时检查和程序的易读性。

实现序列的另一个方法是等前一个 tween 结束后再开始下一个。几乎和上一个实例 TweenerSequence 一样：

```
package {

    import flash.display.*;
    import gs.TweenLite;
    import gs.easing.Elastic;

    [SWF(backgroundColor=0xffffffff)]
    public class TweenLiteSequence2 extends Sprite
    {
        private var sprite:Sprite;
        private var tween1:TweenLite;
        private var tween2:TweenLite;
        private var tween3:TweenLite;

        public function TweenLiteSequence2()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

tween1 = new TweenLite(sprite, 3, {x:800, onComplete:onTween1End});
        }

        private function onTween1End():void
        {
            tween2 = new TweenLite(sprite, 3, {y:400, onComplete:onTween2End});
        }

        private function onTween2End():void
        {
```

```

        tween3 = new TweenLite(sprite, 3, {rotation:360});
    }
}

```

我们创建第一个 tween，然后设置 onComplete 回调函数，我们创建下一个 tween 然后设置他的 onComplete 的回调函数。在最后的回调函数里创建最后一个 tween。当然，这比使用延迟更复杂，但是当你需要在 tween 结束时进行其他操作时，他会变得相当强大。例如，当一个导航元素渐变结束后，他就需要一个回调函数来配置自己。

最后，TweenGroup 给了我们更多强大的方式来协调和排序。第一个方法就是使用 TweenGroup 对象的 align 属性，它可以被设置成若干不同的模式，从而影响一个组里的 tween 怎么一起工作。在创建序列时，最有用的是 ALIGN\_SEQUENCE。回到我们之前创建的类 TweenGroupDemo1，我们能轻而易举的把这组 tween 变成序列（对应这一实例的文件是 TweenGroupDemo1.as）

```

package {

    import flash.display.*;
    import gs.TweenGroup;

    [SWF(backgroundColor=0xffffffff)]
    public class TweenGroupDemo2 extends Sprite
    {
        private var sprite:Sprite;
        private var group:TweenGroup;

        public function TweenGroupDemo2()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            var s1:Sprite = makeSprite(100, 100, 0xff0000);
            var s2:Sprite = makeSprite(100, 200, 0x00ff00);
            var s3:Sprite = makeSprite(100, 300, 0x0000ff);
            var s4:Sprite = makeSprite(100, 400, 0xffff00);

            var tg:TweenGroup = TweenGroup.allTo([s1, s2, s3, s4], 1, {x:800});
            tg.align = TweenGroup.ALIGN_SEQUENCE;
        }

        private function makeSprite(xpos:Number, ypos:Number, color:uint):Sprite
        {
            var s:Sprite = new Sprite();
            s.graphics.beginFill(color);
            s.graphics.drawRect(-50, -25, 100, 50);
            s.graphics.endFill();
            s.x = xpos;
            s.y = ypos;
            addChild(s);
            return s;
        }
    }
}

```

这里我们把 TweenGroup 保存在变量 tg 里面。然后我们能把这个组的 align 属性设为

TweenGroup.ALIGN\_SEQUENCE。这将导致组里的 tween 在前一个结束后才被执行。当你运行这段程序，你会发现这组 4 个 sprite 不是一起运动，而是一个接一个的移动。

这样我们就能让若干对象按循序渐变了。但是如果只想改变一个对象，让他按顺序改变某些属性呢？（比如先沿 x 轴移动，再沿 y 轴移动，然后旋转）

我们可以为这些变化创建 tween，然后创建一个组让他们按顺序运行。下面实例中你可以看到这些，在文件 TweenGroupDemo3.as 里：

```
package {

    import flash.display.*;
    import gs.TweenGroup;
    import gs.TweenLite;

    [SWF(backgroundColor=0xffffffff)]
    public class TweenGroupDemo3 extends Sprite
    {
        private var sprite:Sprite;
        private var tween1:TweenLite;
        private var tween2:TweenLite;
        private var tween3:TweenLite;

        public function TweenGroupDemo3()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            tween1 = new TweenLite(sprite, 3, {x:800});
            tween2 = new TweenLite(sprite, 3, {y:400, overwrite:0});
            tween3 = new TweenLite(sprite, 3, {rotation:360, overwrite:0});
            var tg:TweenGroup = new TweenGroup([tween1, tween2, tween3]);
            tg.align = TweenGroup.ALIGN_SEQUENCE;
        }
    }
}
```

第一眼看上去，这比简单使用延迟要复杂得多，因为要创建一个新的对象并设置它的属性。但这样做更具灵活性。使用延迟时，你必须在任何 tween 开始前跟踪他的持续时间。非常容易出错！当然，你可以使用变量来设置这些 tween 的时长，然后使用这些变量来得到延迟量。但是当你这么做时，就会变得跟使用 group 一样繁琐。

我们所探索的部分对于整个 TweenLite/TweenGroup 来说，实在只是冰山的一角。而且我们甚至没有说到 TweenLite 的大哥哥：TweenMax，他有很多附加功能。如果你需要做很多复杂的 tween 序列，这些类会救你的命。

### KitchenSync

我们要介绍的下一个渐变引擎是 KitchenSync，由 Mims Wright 编写。你可以从 Google Code

站下载到他的源代码和 SWC 类库 (<http://code.google.com/p/kitchensynclib>)。我相信你知道如何把源程序包或者 SWC 库添加到你的编译路径。如果不会，请参见前面提到的网址的安装指导页面。

这个包的主类是 org.as3lib.kitchensync.KitchenSync。在任何应用了 KitchenSync 的项目主文档类里，你要做的第一件事是调用方法：KitchenSync.initialize(this)。这个用法把对文档类本身的引用作为参数，当然你也可以使用对任何可视对象的引用。KitchenSync 将使用这个对象来监听 enterFrame 事件，并据此来更新任何进行中的渐变，或基于时间的动作。

在你初始化引擎之后，你就创建渐变了，也就是 org.as3lib.kitchensync.action.KSTween 的一个实例。构造函数是这个样子的：

```
new KSTween(target, property, startValue, endValue, duration);
```

参数的意义你应该可以猜出一二，target 就是你要渐变的对象；property 是你要改变的属性，比如“x”，“y”，“rotation”；start 和 end 值是属性开始和结束的数值；duration 是渐变将持续的时间。

有意思的是，duration 可以是数值也可以是字符串。如果你使用数值，他将被当作毫秒数（如 3000 就代表 3 秒钟）。你也可以使用一个字符串，例如“3sec”，来代表渐变要持续的秒数。我们了解这些已经足够了。这里字符串的更多用法你也可以查看说明文档。

KSTween 与其他渐变引擎不同的另一特性是，他不会在你创建后自动执行。你需要调用 KSTween 实例的 start 方法，否则你要改变的对象会在那里坐着不动。

好了，这样我们就可以开始了。我们来创建一个 KitchenSync 渐变吧（下面的实例在文件 KitchenSyncDemo1.as）：

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

    import org.as3lib.kitchensync.KitchenSync;
    import org.as3lib.kitchensync.action.KSTween;
    import org.as3lib.kitchensync.easing.Quartic;

    [SWFBackgroundColor=0xffffffff]
    public class KitchenSyncDemo1 extends Sprite
    {
        private var sprite:Sprite;

        public function KitchenSyncDemo1()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            KitchenSync.initialize(this);
            var tween:KSTween = new KSTween(sprite, "x", 100, 800, 3000);
            tween.start();
        }
    }
}
```

```
    }
}
```

重要的代码已经标为粗体。我们初始化了引擎，创建了一个渐变在 3 秒钟内把 sprite 从 x 轴的 100 位置移到 800 的位置，然后开始动画。奇迹发生了！

### KitchenSync 的缓动函数

你猜 KitchenSync 的缓动方程是谁写的？Robert Penner！没错，但除了我们已经看了很多遍的标准函数，Mims 还加入了他自己的一些新类：

```
Oscillate  
Sextic  
Random  
Stepped
```

你可以任意让他们作实验，但我们现看看怎么定义一个现有的缓动函数。基本上，你只要把他作为 KSTween 构造函数的最后一个参数传进去就可以了。然而，还有一点，倒数第二个可选参数：delay。如果你不想用延迟，只要给一个 0 就好了，后面跟上缓动函数。缓动类都在 org.as3lib.kitchensync.easing 包里面，像其他引擎的缓动一样，他们都有 easeIn，easeOut，easeInOut 方法。所以要想使用 Quartic.easeInOut 方程，这样做：

```
var tween:KSTween = new KSTween(sprite, "x", 100, 800, 3000, 0, Quartic.easeInOut);
```

### 用 kitchenSync 改变多个对象或属性

与 tweener 和 TweenLite 不同，KitchenSync 不能在一个 tween 里改变多个属性，你必须为每个要改变的属性创建单独的 tween。有一个创建新 tween 的快捷方式：cloneWithTarget。你可以在一个现有的 tween 上调用 cloneWithTarget，传入新的（或相同的）对象，和一个属性名。所以要这样作：

```
var tween1:KSTween = new KSTween(sprite, "x", 100, 800, 3000, 0, Quartic.easeInOut);
var tween2:KSTween = tween1.cloneWithTarget(sprite, "y");
tween1.start();
tween2.start();
```

但是这样会同时复制开始和结束值。在绝大多数情况下，你在对一个对象的多个属性进行渐变时会使用不同的数值。所以这个克隆方法在这种情况下作用有限，你不如直接创建新的 tween。

```
var tween1:KSTween = new KSTween(sprite, "x", 100, 800, 3000, 0, Quartic.easeInOut);
var tween2:KSTween = new KSTween(sprite, "y", 100, 400, 3000, 0, Quartic.easeInOut);
tween1.start();
tween2.start();
```

相似地，在改变多个对象时，你也需要创建多个 tween。cloneWithTarget 在这里会更有用。请看下面的实例（KitchenSyncMultiple.as）：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import org.as3lib.kitchensync.KitchenSync;
    import org.as3lib.kitchensync.action.KSTween;
    import org.as3lib.kitchensync.easing.Quartic;
    [SWF(backgroundColor=0xffffffff)]

    public class KitchenSyncMultiple extends Sprite
    {
        private var sprite:Sprite;
        public function KitchenSyncMultiple()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
```

```

stage.align = StageAlign.TOP_LEFT;
var s1:Sprite = makeSprite(100, 100, 0xff0000);
var s2:Sprite = makeSprite(100, 200, 0x00ff00);
var s3:Sprite = makeSprite(100, 300, 0x0000ff);
var s4:Sprite = makeSprite(100, 400, 0xffff00);

KitchenSync.initialize(this);
var tween1:KSTween = new KSTween(s1, "x", 100, 800, 3000, 0, Quartic.easeInOut);
var tween2:KSTween = tween1.cloneWithTarget(s2, "x");
var tween3:KSTween = tween1.cloneWithTarget(s3, "x");
var tween4:KSTween = tween1.cloneWithTarget(s4, "x");
tween1.start();
tween2.start();
tween3.start();
tween4.start();

}

private function makeSprite(xpos:Number, ypos:Number, color:uint):Sprite
{
    var s:Sprite = new Sprite();
    s.graphics.beginFill(color);
    s.graphics.drawRect(-50, -25, 100, 50);
    s.graphics.endFill();
    s.x = xpos;
    s.y = ypos;
    addChild(s);
    return s;
}
}
}
}

```

每个 tween 都是上一个的副本，只是定义了新的目标对象。

请注意你需要逐一让每个 tween 开始。KitchenSync 的一些更强大的功能在于他的行动组合 (action group)，这与 TweenLite 包里的 TweenGroup 很类似。使用行动组合，你可以同步多个 tween 。 行动组合的类型很多。要同时开启多个 tween ， 使用 org.as3lib.kitchensync.action.KSParallelGroup 类。在构造函数里传入你需要同步的任何 tween，然后调用组合的 start 方法。像这样：

```

var tween1:KSTween = new KSTween(s1, "x", 100, 800, 3000, 0, Quartic.easeInOut);
var tween2:KSTween = tween1.cloneWithTarget(s2, "x");
var tween3:KSTween = tween1.cloneWithTarget(s3, "x");
var tween4:KSTween = tween1.cloneWithTarget(s4, "x");
var pg:KSParallelGroup = new KSParallelGroup(tween1, tween2, tween3, tween4);
pg.start();

```

同样，首先这样看起来要比直接调用四次 start 要复杂一点。然而，行动组合是 KitchenSync 强大功能的重要来源。当你在创建一个组合，比如 KSParallelGroup 时，你不仅可以传递 KSTween 的实例，还可以用 KSParallelGroup 的实例，或者任何其他行动组合。这样，你就能创建非常复杂的动画序列，而且可以在任何时候重放，只要调用一下组合的 start 方法。这些东西在下个部分你会看到更多。

### KitchenSync 的 tween 序列

与你已经了解的其他 tween 引擎类似，你可以通过 delay 或者监听一个 tween 的完成事件再创建下一个来实现 tween 序列。我不会再讲解这种方法，因为这与其他引擎没有太大区别。更有趣的是使用另一个行动组合， KSSequenceGroup，来实现序列。

使用 KSSequenceGroup 就像使用 KSParallelGroup 一样简单。下一个类，文件 KitchenSyncSequence.as，展示了这种做法：

```
package {

    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import org.as3lib.kitchensync.KitchenSync;
    import org.as3lib.kitchensync.action.KSSequenceGroup;
    import org.as3lib.kitchensync.action.KSTween;
    import org.as3lib.kitchensync.easing.Quartic;

    [SWF(backgroundColor=0xffffffff)]
    public class KitchenSyncSequence extends Sprite

        private var sprite:Sprite;
        public function KitchenSyncSequence()
        {
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;
            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);
            KitchenSync.initialize(this);
            var tween1:KSTween = new KSTween(sprite, "x", 100, 800, 3000, 0, Quartic.easeInOut);
            var tween2:KSTween = new KSTween(sprite, "y", 100, 400, 3000, 0, Quartic.easeInOut);
            var tween3:KSTween = new KSTween(sprite, "rotation", 0, 360, 3000, 0, Quartic.easeInOut);
            var sg:KSSequenceGroup = new KSSequenceGroup(tween1, tween2, tween3);
            sg.start();
        }
    }
}
```

我们在这里为三个不同的属性创建了三个 tween，然后把他们加入一个 KSSequenceGroup 实例。开始这个组，所有动画就按秩序发生了。

记住你可以把一个序列组加入一个并行组，让这个序列与其他一些动画同步执行，反之亦然。这可以变得很强大。

较之前面说到的两个引擎，KitchenSync 可能更适合开发者的思维。比如在 Tweener 里你可以在一个类里使用几乎所有功能，而 tweenLite 则有更多的类，KitchenSync 则是一个有着数十个类，包和接口的庞大框架。他需要一点学习时间，但会更贴近你的工作流程，而且非常灵活，可以完成你想要的几乎所有任务。

### KitchenSync 的 tween 序列

与你已经了解的其他 tween 引擎类似，你可以通过 delay 或者监听一个 tween 的完成事件再创建下一个来实现 tween 序列。我不会再讲解这种方法，因为这与其他引擎没有太大区别。更有趣的是使用另一个行动组合，KSSequenceGroup，来实现序列。

使用 KSSequenceGroup 就像使用 KSParallelGroup 一样简单。下一个类，文件

KitchenSyncSequence.as，展示了这种做法：

```
package {
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import org.as3lib.kitchensync.KitchenSync;
    import org.as3lib.kitchensync.action.KSSequenceGroup;
    import org.as3lib.kitchensync.action.KSTween;
    import org.as3lib.kitchensync.easing.Quartic;

    [SWF(backgroundColor=0xffffffff)]
    public class KitchenSyncSequence extends Sprite

    private var sprite:Sprite;
    public function KitchenSyncSequence()
    {
        stage.scaleMode = StageScaleMode.NO_SCALE;
        stage.align = StageAlign.TOP_LEFT;
        sprite = new Sprite();
        sprite.graphics.beginFill(0xff0000);
        sprite.graphics.drawRect(-50, -25, 100, 50);
        sprite.graphics.endFill();
        sprite.x = 100;
        sprite.y = 100;
        addChild(sprite);
        KitchenSync.initialize(this);
        var tween1:KSTween = new KSTween(sprite, "x", 100, 800, 3000, 0, Quartic.easeInOut);
        var tween2:KSTween = new KSTween(sprite, "y", 100, 400, 3000, 0, Quartic.easeInOut);
        var tween3:KSTween = new KSTween(sprite, "rotation", 0, 360, 3000, 0, Quartic.easeInOut);
        var sg:KSSequenceGroup = new KSSequenceGroup(tween1, tween2, tween3);
        sg.start();
    }
}
```

我们在这里为三个不同的属性创建了三个 tween，然后把他们加入一个 KSSequenceGroup 实例。开始这个组，所有动画就按秩序发生了。

记住你可以把一个序列组加入一个并行组，让这个序列与其他一些动画同步执行，反之亦然。这可以变得很强大。

较之前面说到的两个引擎，KitchenSync 可能更适合开发者的思维。比如在 Tweener 里你可以在一个类里使用几乎所有功能，而 tweenLite 则有更多的类，KitchenSync 则是一个有着数十个类，包和接口的庞大框架。他需要一点学习时间，但会更贴近你的工作流程，而且非常灵活，可以完成你想要的几乎所有任务。

### gTween

最后一个引擎是 Grant Skinner 的 gTween。Grant 是一个小有名气的 Flash 开发者，也是我的好朋友（我在这本书中多次提到他的作品）。在我写这段内容时，gTween 刚刚发布了公共预览版。所以在你读到这些内容的时候，他的应用程序接口（API）和工作流程可能已经改变了。但我觉得我们值得花一点时间了解这款引擎。

你可以在这里下载到 gTween：<http://gskinner.com/libraries/gtween/>。

gTween 全部在一个类里：com.gskinner.motion.GTween。这个类的构造函数像这个样子：

```
new GTween(target, duration, properties, tweenProperties);
```

像你见过的其他引擎一样，target 是你要改变的对象，而 duration 是变化持续的时间。properties 参数包含了你要修改对象的属性。这是一个 Object 类型的数据，工作方式跟 Tweener 和 TweenLite 一样。然而，在 gTween 里有一个单独的参数，tweenProperties，供你来改变补间的其他属性，比如 delay 或缓动方程。

这些已经足够我们开始了。下面一个实例在文件 GTweenDemo.as 里：

```
package {
    import com.gskinner.motion.GTween;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import mx.effects.easing.Circular;

    [SWF(backgroundColor=0xffffffff)]
    public class GTweenDemo extends Sprite
    {
        private var sprite:Sprite;

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

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            var tween:GTween = new GTween(sprite, 3, {x:800});
        }
    }
}
```

非常简单。如果你看过本章中对其他引擎的说明，这里真的不需要再做解释了。像 Tweener 和 TweenLite 那样，你可以改变一个对象的多个属性，只要把它们加到 properties 参数里就可以了：

```
var tween:GTween = new GTween(sprite, 3, {x:800, y:400});
```

### gTween 的缓动函数

我欣赏 gTween 的一点是，他不会重新发明已经存在的东西。就像你能猜到的那样，这个引擎使用了 Robert Penner 的方程式。但是与其他引擎做法不同，gTween 没有把这些类复制成新的，实际上一模一样的类或函数。他直接使用了 Flash 和 Flex 自带的类：Flash 里的 fl.transitions.easing 包和 Flex 框架里的 mx.effects.easing。

你可以通过构造函数的参数 tweeningProperties 的 ease 属性或者在补间实例创建后直接修改他的 ease 属性来定义你需要的缓动函数。下面两种做法都将补间的缓动函数定义为 mx.effects.easing.Circular.easeInOut（注意别忘了引用你需要的任何类）。第一个是这样：

```
var tween:GTween = new GTween(sprite, 3, {x:800, y:400}, {ease:Circular.easeInOut});
```

这是第二种做法：

```
var tween:GTween = new GTween(sprite, 3, {x:800, y:400});
```

```
tween.ease = Circular.easeInOut;
```

## 用 gTween 改变多个对象

像是大多数引擎的惯例，改变多个对象就要创建多个补间实例。类 GTween 包含的一个克隆方法让创建多个补间变得简单。你可以给 clone 方法传递一个新的对象，这个补间就被复制到新的对象上了。下一个类，文件 GTweenMulti.as 展示了这个做法：

```
package {
    import com.gskinner.motion.GTween;

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

    import mx.effects.easing.Quadratic;

    [SWF(backgroundColor=0xffffffff)]
    public class GTweenMulti extends Sprite
    {

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

            var s1:Sprite = makeSprite(100, 100, 0xff0000);
            var s2:Sprite = makeSprite(100, 200, 0x00ff00);
            var s3:Sprite = makeSprite(100, 300, 0x0000ff);
            var s4:Sprite = makeSprite(100, 400, 0xffff00);

            var tween1:GTween = new GTween(s1, 3, {x:800},
                {ease:Quadratic.easeInOut});
            var tween2:GTween = tween1.clone(s2);
            var tween3:GTween = tween1.clone(s3);
            var tween4:GTween = tween1.clone(s4);
        }

        private function makeSprite(xpos:Number, ypos:Number, color:uint):Sprite
        {
            var s:Sprite = new Sprite();
            s.graphics.beginFill(color);
            s.graphics.drawRect(-50, -25, 100, 50);
            s.graphics.endFill();
            s.x = xpos;
            s.y = ypos;
            addChild(s);
            return s;
        }
    }
}
```

## gTween 的补间序列

与其他 tween 引擎类似，你可以通过 delay 属性或者监听一个 tween 的完成事件再创建下一个来实现 tween 序列。对于这种做法我已经做了足够多的实例，用 gTween 来做也不会有太大不同。一些引擎有组合功能，你可以添加多个补间来创建补间组。而 gTween 采取了一个截然不同的思路

来实现序列，使用 nextTween 属性。

简单的说，你可以创建两个补间，让第二个暂停。把第二个补间的引用传递给第一个补间的 nextTween 属性。当第一个补间结束时，第二个补间就会被激活。通过实例可能比靠语言解释要更容易理解，下面是一个范例，在文件 GTweenSequence.as 里：

```
package {
    import com.gskinner.motion.GTween;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import mx.effects.easing.Circular;
    import mx.effects.easing.Quadratic;

    [SWF(backgroundColor=0xffffffff)]
    public class GTweenSequence extends Sprite
    {
        private var sprite:Sprite;

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

            sprite = new Sprite();
            sprite.graphics.beginFill(0xff0000);
            sprite.graphics.drawRect(-50, -25, 100, 50);
            sprite.graphics.endFill();
            sprite.x = 100;
            sprite.y = 100;
            addChild(sprite);

            var tween1:GTween = new GTween(sprite, 3, {x:800},
                {ease:Quadratic.easeInOut});
            var tween2:GTween = new GTween(sprite, 3, {y:400},
                {ease:Quadratic.easeInOut});
            var tween3:GTween = new GTween(sprite, 3, {rotation:180},
                {ease:Quadratic.easeInOut});

            tween2.pause();
            tween3.pause();

            tween1.nextTween = tween2;
            tween2.nextTween = tween3;
        }
    }
}
```

这里我们创建了三个补间，然后暂停了第二个和第三个。把 tween1.nextTween 设为 tween2，tween2.nextTween 设置为 tween3。他们就可以完美的运行了。

像本章介绍的其他内容一样，这些讨论只涉及了 gTween 全部功能的一个表面，实际上这个类中整合了很多功能。要想知道你都能拿他做些什么，就得通读他的说明文档，我相信内容会不断增加。事实上，在我开始写这部分内容和我最后一次审阅他的中间，gTween 又发布了一个新的版本。

所以请把这里的内容当作最简要的介绍，要知道你阅读本章时 gTween 已经具有什么功能，请直接查阅网站资源。

## 总结

但愿这一章让你对众多 tween 引擎有了较好的认识。也许你已经对实现补间的不同做法有了足够的认识并挑选出了最适合你的一款引擎。甚至也许你觉得自己能做的更好并开始编写你自己的引擎了！加油干！

还有，虽然我不认为你会一字一句的阅读这本书，这一章还把我们带到了本书的结尾。我希望对于本书中讨论的各个题目，你能像我在研究资料，编写实例，和撰写文字的时候一样找到乐趣。在最后，我相信本书中的一两个章节会让你开始你自己的研究项目。