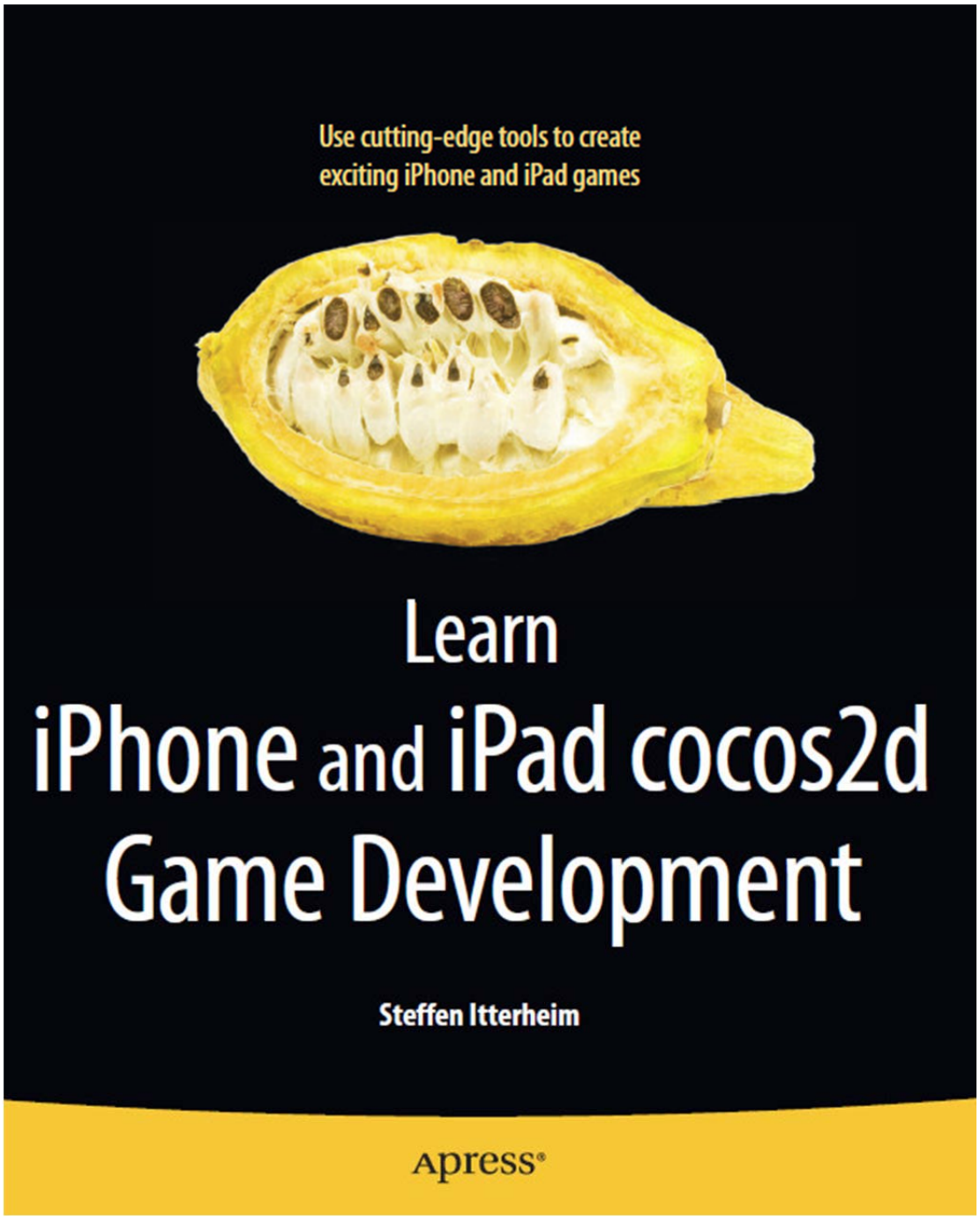

译者：杨栋
邮箱：yangdongmy@gmail.com

第七章
横向滚屏射击游戏

接着上一章的讨论，我现在要把上一章的知识应用到本章制作一个射击游戏。首先要做的是让飞船成为一个可控制的角色。我们不会在这里使用加速计来控制飞船，而是使用一个虚拟手柄来控制飞船的移动。不过我们不用设计自己的虚拟手柄。我将会使用SneakyInput这个源代码包给我们的cocos2d游戏添加虚拟手柄。

让飞船移动只完成了其中一件事情。我们也需要让背景滚动以造成飞船在向某个方向移动的感觉。因为CCParallaxNode的功能太有限，不适用于生成一个无限的横向滚动的背景，所以我们将会实现自己的横向视差滚屏。

另外，我也会在这里演示上一章学到的纹理贴图集和精灵批量处理的知识。本章的游戏图形将被包含在一个纹理贴图集中。

高级视差滚屏

之前我提到了CCParallaxNode不适用于生成无限滚屏效果。在这个射击游戏中，我将使用一个ParallaxBackground节点，以生成无限滚屏效果。此外，我也会使用CCSpriteBatchNode来提高背景图片的渲染速度。

制作条纹状的背景图片

首先，我想展示一下如何制作用于视差滚屏背景的条纹状背景图片。了解如何使用Zwoptex来制作纹理贴图集可以帮助你节省内存和提高程序运行效率，同时也可以节省放置单独的条纹图片的时间。**图7-1**展示了我在Seashore（图形处理程序）中制作的480x320的背景层。你可以在ScrollingWithJoy01项目中的Resources文件夹中找到这个文件：background-parallax.xcf。

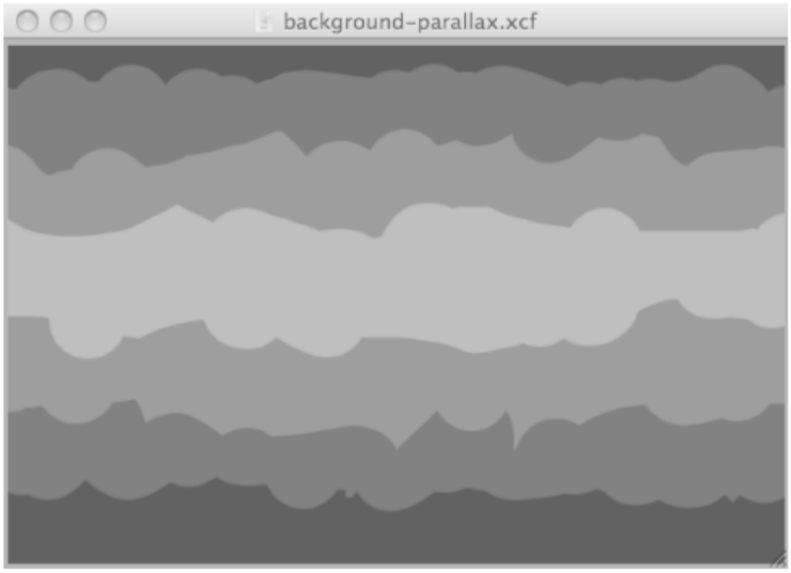

图7-1. 用于生成视差滚屏效果的背景图片

在Seashore程序中，每一个条纹都是一个单独的层。在图7-2中你可以看到各个层。如果你看一下Resources文件夹中的文件，你会注意到有相对应于各个层的单独的图片文件，命名从bg0.png到bg6.png，一共7张图片，每个图片包含一个条纹。

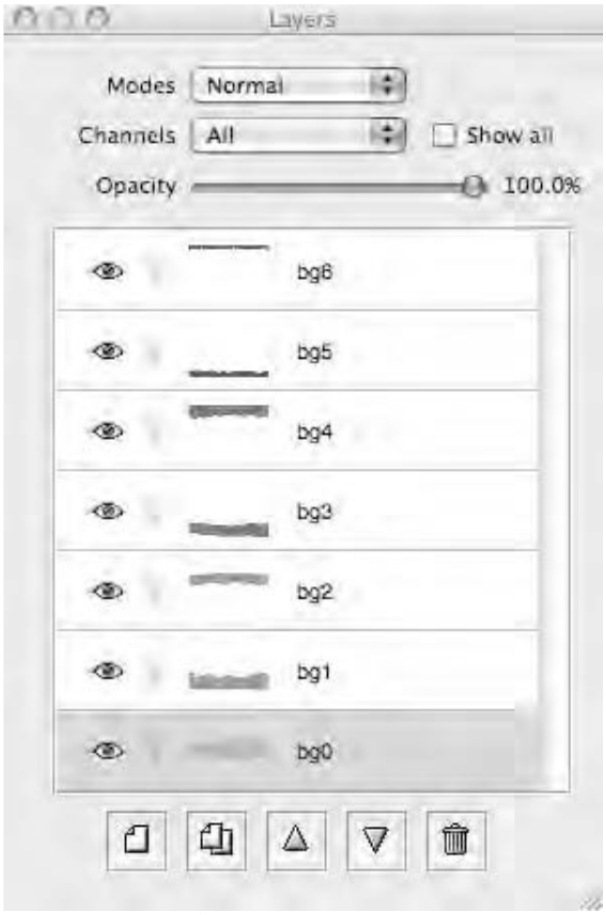

图7-2.每个背景条纹都放在单独的层里面，便于生成单个的文件，从而便于你将这个文件放置在游戏中。

通过把每个条纹放到单独的层中，你只需要生成一个文件就可以把各个层存为单独的图片。所有这些单独的条纹文件都是480x320像素的，乍看起来很浪费，但是你并不是要把这些单独的文件加载到游戏中去，你是要把它们放到纹理贴图集中去。因为Zwoptex可以移除每个图片的透明部份，所以它会把这些条纹图片的透明部份尽量的清理掉。图7-3展示了完成的纹理贴图集。

注：我设计的背景图片是普通iPhone的屏幕尺寸-480x320像素。如果你要支持iPad屏幕和iPhone的视网膜屏幕，你需要制作相应大小的背景图片。

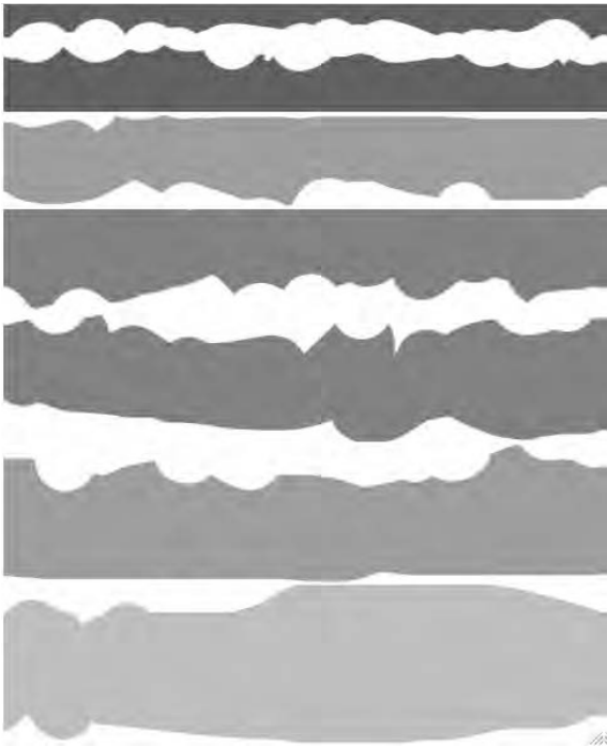

图7-3. 完成的背景纹理贴图集

把各个条纹分成单独的文件不光可以单独控制它们的z-order值，还可以帮助你节省贴图空间。严格来说，bg5.png和bg6.png可以放在同一张图片里，因为它们一个在背景的顶部，一个在背景的底部，并不会重叠。但是我还是把它们分开存放了。因为如果不分开存放的话，由于它们一个在上，一个在下，Zwoptex就不能清除它们之间存在的大块空间，它们在纹理贴图集中的尺寸就还是原来的480x320，从而造成贴图空间的浪费。

把条纹分开保存成单独的文件还可以保持较高的帧率。iOS设备的填充率（fill rate）很低，也就是它们每一帧可以绘制的像素数量很有限。因为图片经常会相互叠加在一起，iOS设备就经常需要在同一帧里绘制相同的像素好几遍。最极端的情况是两张全屏的图片叠加在一起，虽然你在同一时间里只能看到其中一张图片，但是你的设备还是会绘制两张图片。这在技术上叫做“全景渲染造成的浪费”（overdraw）。将背景图分成单独的条纹图片，同时这些图片之间很少重叠，这样可以有效降低重复绘制的像素，从而提高帧率。

在代码中重建背景

你现在可能担心是不是要花很多时间才能把这些分开的条纹图片在代码里重建背景。答案是：你不用花很多时间。因为这些图片都是以480x320尺寸保存的，Zwoptext知道它们之间的间距。你要做的就是把所有的图片放在Zwoptex画布中央，软件会把图片放到正确的位置上。

让我们看一下在ScrollingWithJoy01项目中新加的ParallaxBackground节点。它的头文件很简单：

```
@interface ParallaxBackground : CCNode
{
        CCSpriteBatchNode* spriteBatch;
        Int numSprites;s
}
@end
```

我在这里设置了对CCSpriteBatchNode节点的引用。我们将在代码里经常调用这个节点。调用一个保存在成员变量里的节点比通过getNodeByTag方法来调用要来的快。如果你在每一帧中都这样做的话，你就可以省下几个CPU处理周期。当然，如果你需要保存几十甚至上百个这样的成员变量的话，那就不值得这样做了。

列表7-1：在ParallaxBackground类的init方法中，我们生成了一个CCSpriteBatchNode节点，然后把所有7个存放在同一个纹理贴图集里的背景图片添加到CCSpriteBatchNode中。

列表7-1. 加载背景

```
CGSize screenSize = [[CCDirector sharedDirector] winSize];

// 获取纹理贴图集的2D贴图
CCTexture2D* gameArtTexture = [[CCTextureCache sharedTextureCache]
                                                    addImage:@"game-art.png"];
spriteBatch = [CCSpriteBatchNode batchNodeWithTexture:gameArtTexture];
[self addChild:spriteBatch];

numSprites = 7;

// 将7个不同的条纹图片添加到批处理精灵对象中
for (int i = 0; i < numSprites; i++)
{
    NSString* frameName = [NSString stringWithFormat:@"bg%i.png", i];
    CCSprite* sprite = [CCSprite spriteWithSpriteFrameName:frameName];
    sprite.position = CGPointMake(screenSize.width / 2, screenSize.height / 2);
    [spriteBatch addChild:sprite z:i];
}
```

首先，我把game-art.png这张纹理贴图集加到CCTextureCache中。实际上，我已经在GameScene类中加载了这张纹理贴图集，为什么还要再加载一遍呢？原因是我需要用到CCTexture2D对象以生成CCSpriteBatchNode，而把相同一张贴图再次添加到CCTextureCache中是唯一一个获取已被缓存的贴图的方法。这个操作不会将贴图再次加载；CCTextureCache这个单例知道这个贴图已被加载过，会直接调用已被缓存的版本，这个操作速度很快。你可能会觉得奇怪，为什么没有一个像getTextureByName这样的方法来调用已经被缓存的贴图。不过现在确实是没有这样的方法可用。

创建和设置完CCSpriteBatchNode以后，接下去的步骤是把7个单独的背景图片加载进来。我故意使用0到6的数字作为图片文件命名的结尾，这样我就可以使用stringWithFormat来动态生成文件名了：

```
NSString* frameName = [NSString stringWithFormat:@"bg%i.png", i];
```

通过使用精灵的frameName，我创建了一个普通的CCSprite，然后把它放在屏幕的中央：

```
sprite.position = CGPointMake(screenSize.width / 2, screenSize.height / 2);
```

当然，如果你要制作一个iPad版本的话，这些图片就不适用了，因为它们是为480x320像素的屏幕设计的。你可以使用上述步骤生成适用于iPad屏幕的背景，唯一需要做的是制作1024x768像素的图片。

技巧：在cocos2d里重建背景很简单，这都归功于Zwoptex保存了图片之间的位移信息。你也可以用相同的方法设计游戏屏幕的布局。你可以请设计师把每个屏幕上的元素设计成独立的层。然后将每一层导出成一个单独的整屏尺寸的图片（带透明背景信息的）。接着，你用这些文件生成一个纹理贴图集，然后在cocos2d中应用这个纹理贴图集来重建设计师的设计。这样你既可以不必手动地放置单独的文件，也不会浪费内存。

因为ParallaxBackground这个类继承自CCNode，所以我只需要通过以下代码将ParallaxBackground添加到GameScene层当中去：

```
ParallaxBackground* background = [ParallaxBackground node];
[self addChild:background z:-1];
```

这会替代上一章中作为占位符的CCColorLayer和背景CCSprite。

移动ParallaxBackground

在ScrollingWithJoy01项目中，我也添加了一个质量不高的背景条纹滚动效果。滚动效果是显示出来了，但是图片只是一闪而过，很快就显示出背后的黑色背景了。虽然**图7-4**所展示的效果不是我想要的，但是我们快接近想要的效果了。

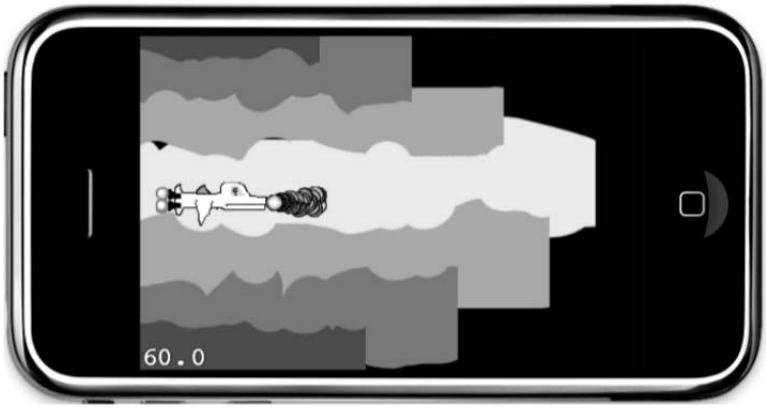

图7-4. 虽然背景的条纹是在移动，但看上去不怎么样。我们需要让这些条纹重复起来。

列表7-2展示的代码是用于生成上述背景滚屏效果的：

列表7-2. 移动背景条纹

```
-(void) update:(ccTime)delta
{
	CCSprite* sprite;
	CCARRAY_FOREACH([spriteBatch children], sprite)
	{
		CGPoint pos = sprite.position;
		pos.x -= (scrollSpeed + sprite.zOrder) * (delta * 20);
		sprite.position = pos;
	}
}
```

每一个条纹背景图片的X轴位置的值在每一帧调用更新方法时都会被减去一些，使它们从右向左移动。移动的多少取决于预先定义的scrollSpeed（滚动速度）变量加上精灵的zOrder属性值。乘上delta这个值可以让滚动的速度独立于帧率。因为delta只是两次调用更新方法之间所经过的时间，比如当前的帧率是60帧每秒，那么delta就等于1/60秒，也就是0.167秒，乘上这样的值图片就会运动的很慢，所以我又乘上20让移动的速度变得快一些。靠近屏幕的图片会滚动地快一些，因为zOrder的值越靠近屏幕越大。

影响视差速度的因素

在我们的游戏中，相同颜色的条纹需要用相同的速度移动，并且条纹应该重复出现以防止出现后面的纯色背景。我的解决方案在ScrollingWityJoy2项目中。

第一个改变是与滚动速度有关的。我使用了CCArray来存储各个条纹图片的移动速度。我们也可以使用其它方法来存储，但是我想在这里说明一个CCArray和所有其它iPhone SDK的数据收集类（Collection Classes）都有的问题：它们只能存储对象（Object），而不能存储像整数和浮点数这样的数值。

解决这个问题的办法是把数值包装到NSNumber对象中去。以下代码是新加的CCArray* speedFactors变量，它被用于存储浮点数。此数组定义在ParallaxBackground类的头文件中：

```
@interface ParallaxBackground : CCNode
{
	CCSpriteBatchNode* spriteBatch;
	int numStripes;
	CCArray* speedFactors;
	float scrollSpeed;
}
@end
```

然后我们在ParallaxBackground的init方法中将值填充进上述数组。请注意我

是如何使用NSNumber numberWithFloat，把浮点数存到数组中去的：

```
// 将用于存储各个条纹图片速度值的数组初始化
speedFactors = [[CCArray alloc] initWithCapacity:numStripes];
[speedFactors addObject:[NSNumber numberWithFloat:0.3f]];
[speedFactors addObject:[NSNumber numberWithFloat:0.5f]];
[speedFactors addObject:[NSNumber numberWithFloat:0.5f]];
[speedFactors addObject:[NSNumber numberWithFloat:0.8f]];
[speedFactors addObject:[NSNumber numberWithFloat:0.8f]];
[speedFactors addObject:[NSNumber numberWithFloat:1.2f]];
[speedFactors addObject:[NSNumber numberWithFloat:1.2f]];
NSAssert([speedFactors count] == numStripes, @"speedFactors count mismatch!");
```

最后的NSAssert用于检查可能出现的人为错误。假设你将来添加或者删除了一些背景条纹图，而你又忘记调整numStripes这个变量（用于决定speedFactors这个数组的元素数量），那么NSAssert会告诉提醒你在数组初始化中所要做的修改，而不是随机地让你的游戏崩溃。

图7-5中，你可以看到哪个条纹应用了哪个速度。拥有高速度值的条纹移动的快一些，这样就创造出了视差效果。

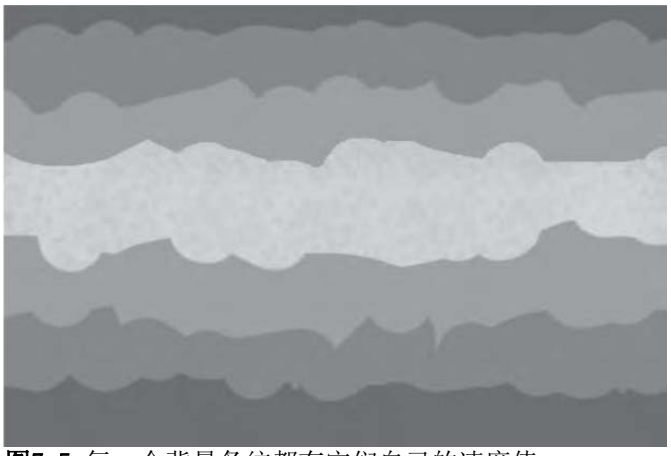

图7-5. 每一个背景条纹都有它们自己的速度值

因为speedFactors这个变量是用alloc来生成的，所以它占用的内存必须被释放。我们在新加的dealloc方法中释放speedFactors数组：

```
-(void) dealloc
{
        [speedFactors release];
        [super dealloc];
}
```

为了使用新加的speedFactors数组变量，游戏的update方法被修改成了如下代码：

```
-(void) update:(ccTime)delta
{
```

```
        CCSprite* sprite;
        CCARRAY_FOREACH([spriteBatch children], sprite)
        {
                NSNumber* factor = [speedFactors objectAtIndex:sprite.zOrder];
                CGPoint pos = sprite.position;
                pos.x -= scrollSpeed * [factor floatValue] * (delta * 50);
                sprite.position = pos;
        }
}
```

基于精灵的zOrder属性，我从speedFactors数组得到一个包装于NSNumber类中的速度对象。你不能把factor这个对象直接和scrollSpeed相乘，因为它是一个包含了数值的类实例（对象）。你可以通过NSNumber类的floatValue方法来得到浮点数的速度值，乘以scrollSpeed让单个精灵加速或者减速移动。你也可以用NSNumber的intValue方法得到速度值，这与浮点数和整数之间进行转换的效果是一样的。

通过使用speedFactors数组为相同颜色的条纹设置相同的移动速度，现在所有的背景条纹都按我们的设想移动了。不过我们还需要解决无限滚动的问题。

设置背景的无限滚动

ScrollingWithJoy02项目是实现背景无限滚动的第一步。如**列表7-3**中代码所示，我往CCSpriteBatchNode中又添加了一组7个背景条纹图片，当然，它们的设置和之前有一些不同。

列表7-3. 添加处于屏幕外的背景图片

```
// 再添加7个背景条纹图片，平行翻转过来，然后放置在第一组7个背景图片旁边
for (int i = 0; i < numStripes; i++)
{
        NSString* frameName = [NSString stringWithFormat:@"bg%i.png", i];
        CCSprite* sprite = [CCSprite spriteWithSpriteFrameName:frameName];

        // 将新的背景图片放置在屏幕的右端
        sprite.position = CGPointMake(screenSize.width + screenSize.width / 2, screenSize.height / 2);

        // 将精灵平行翻转过来与第一组图片相连接
        sprite.flipX = YES;

        // 新的精灵将使用之前精灵的tag值加上背景条纹图片的总数值 numStripes
        [spriteBatch addChild:sprite z:i tag:i + numStripes];
}
```

我的方法是把每一种背景条纹图片再加一个到原先图片的右边，与原先图片的尾部相接。这就让原先的背景条纹图片的长度增加了一倍，满足了无限滚屏的要求。至于如何实现无限滚屏，我会很快谈到。

首先我要说明的是为什么要把第二组的各个背景图片水平翻转过来：这样做可以在视觉上把左右两个背景条纹图片无缝连接起来，以避免产生任何显眼的边界。新图片的tag值设定为左边图片的tag值加上背景图片的总数值numStrips。这样我们就可以通过对图片tag值增加或者减去numStrips这个变量来访问相邻的两个图片了。

到目前为止，我还没有实现无限滚屏效果，背景图片的滚屏效果比之前只是显示的稍微长了一些而已。

我在ScrollingWithJoy03项目中实现了无限滚屏效果。但是首先我要改变精灵的anchorPoint属性来让事情变得容易一点。我把精灵的anchorPoint的X值改成了0，而不是之前的0.5f：

```
sprite.anchorPoint = CGPointMake(0, 0.5f);
sprite.position = CGPointMake(0, screenSize.height / 2);
```

第二个已被水平翻转过的背景图片的anchorPoint的X值也被设定为了0：

```
sprite.anchorPoint = CGPointMake(0, 0.5f);
sprite.position = CGPointMake(screenSize.width, screenSize.height / 2);
```

背景条纹图片的anchorPoint的值从(0.5f, 0.5f)改成了(0，0.5f）。这样做的好处是你不用再考虑贴图的原点和精灵的X轴位置不相同的问题了。图7-6展示了改变anchorPoint属性值让计算X轴位置变得更加简单：

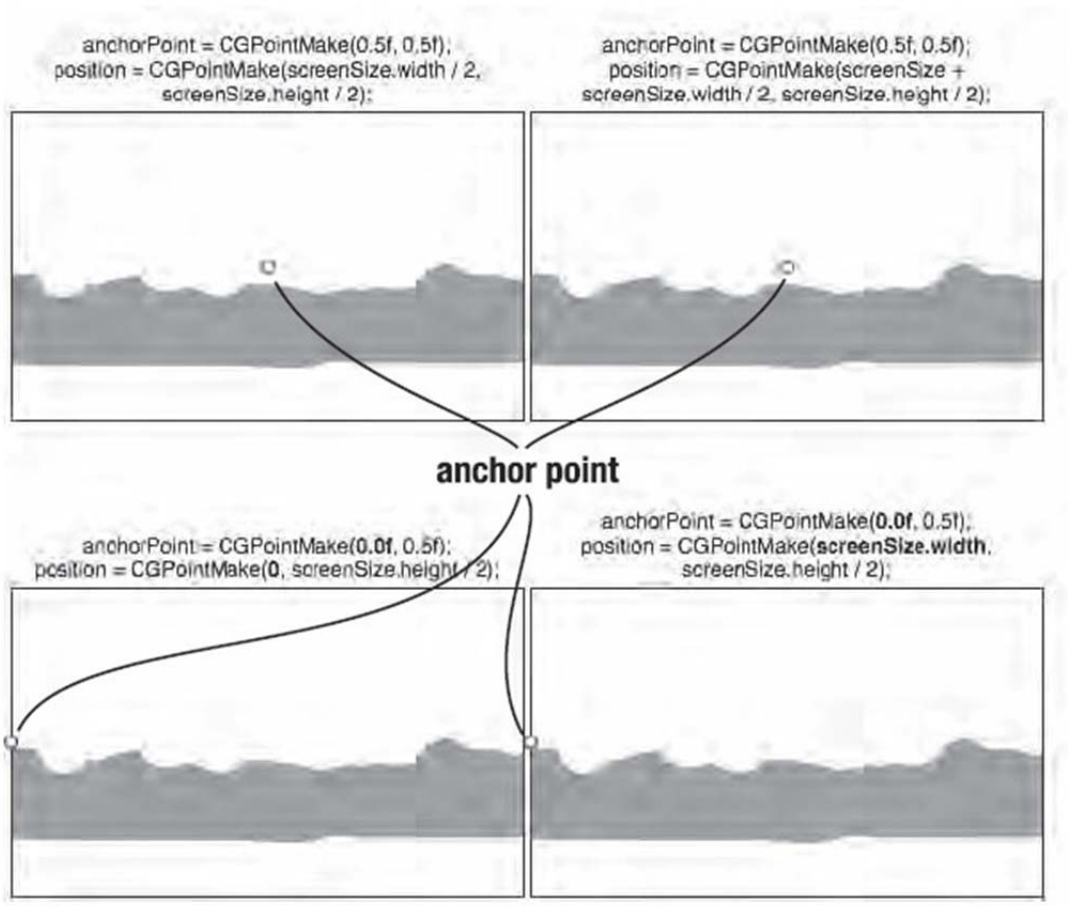

图7-6. 移动过的定位点让计算精灵的X轴位置变得更加简单

你可以在**列表7-4**中看到上述变化带来的好处。我们在下面的代码中实现了无限滚屏效果。

列表7-4. 实现背景图片无限滚屏效果

```
-(void) update:(ccTime)delta
{
	CCSprite* sprite;
	CCARRAY_FOREACH([spriteBatch children], sprite)
	{
		NSNumber* factor = [speedFactors objectAtIndex:sprite.zOrder];

		CGPoint pos = sprite.position;
		pos.x -= scrollSpeed * [factor floatValue];

		// 当背景图片的X轴位置超出屏幕边界时，把超出屏幕的背景图片向右移动两张背景图片宽度的距离，也就是接上当前正显示在屏幕上的背景图片的尾部
		if (pos.x < -screenSize.width)
		{
			pos.x += screenSize.width * 2;
		}

		sprite.position = pos;
	}
```

```
}
```

我们通过检查各个精灵的X轴位置，判断它是不是小于屏幕宽度的负数，如果等于，就把精灵的X轴位置同屏幕宽度的两倍进行相乘。这样就实现了把刚刚移动到屏幕左边界外的精灵，移动到屏幕右边界外面，紧接着当前屏幕上背景精灵的右边界。通过重复这个动作，我们实现了无限滚屏效果。

技巧：你会注意到背景在移动，但是飞船一直停留在原先的地方。缺乏经验的游戏开发者经常认为所有的元素都需要移动已获得游戏物体移动的效果。与之相反的是，你大可以让背景移动，产生前景物体在移动的效果。有很多出名的游戏用到这个技巧，比如说Super Turbo Action Pig, Canabalt, Super Blast, Doodle Jump和Zombieville USA。通常，游戏中进入屏幕的物体是在显示它们之前就随机生成的，物体离开屏幕以后就会从内存中被清除。

消除闪烁

目前为止一切顺利。现在只剩下一个问题。如果你仔细点看，你会注意到背景图片上会时不时出现一条纵向的黑色线条。出现线条的地方就是两个背景图片左右相接的地方。出现线条的原因是计算两个图片位置时出现的四舍五入误差。时不时的，屏幕上会出现1个像素大小的小缝，虽然只出现一秒钟不到的时间，但是对于一个拥有商业品质的游戏来说，我们一定要去掉它。

最简单的办法是把让左右相接的背景图片有1个像素的重叠。在ScrollingWithJoy04项目中，通过将水平翻转过的背景图片的X位置值减去1来改变它的起始位置：

```
sprite.position = CGPointMake(screenSize.width - 1, screenSize.height / 2);
```

我们也要修改更新方法中的代码。在更新方法中，重新回到右边屏幕外面的背景图片要放在原先位置向左2个像素的地方：

```
// 当背景图片的X轴位置超出屏幕边界时，把超出屏幕的背景图片向右移动两张背景图片宽度的距离，也就是接上当前正显示在屏幕上的背景图片的尾部
if (pos.x < -screenSize.width)
{
        pos.x += (screenSize.width * 2) - 2;
}
```

为什么要再向左移动2个像素呢？因为原先被水平翻转过的背景图片已经向左移动了1个像素，每次重新放置背景图片以后我们需要将它向左移动2个像素才能和左边的背景图片保持1个像素的重叠。

另外一个解决方法是只更新当前最左边的背景图片，然后找到与之相连的右边的背景图片，通过移动屏幕宽度的距离来更新。这样也可以避免计算时出现的四舍五入错误。**图7-7**展示了完成的效果。

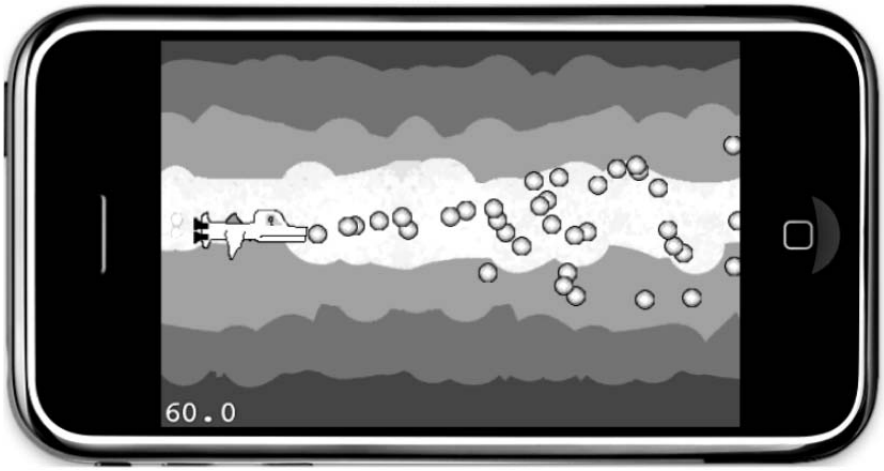

图7-7. 完成的无限视差滚屏效果

重复贴图

我还要介绍一个很有用的技巧。你可以在一块指定大小的正方形区域里让贴图重复出现。如果你把这块指定的正方形区域设置的够大，你可以达到让背景无限滚屏的效果。你至少可以用重复的贴图覆盖几千个像素或者几十个屏幕大小的区域，而不至于影响游戏的运行效率和内存占用率。

我在这里要用到的是OpenGL支持的GL_REPEAT贴图参数。不过这个方法只适用于正方形的区域，而且要满足“2的n次方”规则，比如32x32或者128x128像素。**列表7-5**展示了相关代码：

列表7-5. 用GL_REPEAT重复背景贴图

```
CGRect repeatRect = CGRectMake(-5000, -5000, 5000, 5000);
CCSprite* sprite = [CCSprite spriteWithFile:@”square.png” rect:repeatRect];
ccTexParams params =
{
        GL_LINEAR,
        GL_LINEAR,
        GL_REPEAT,
        GL_REPEAT
};
[sprite.texture setTexParameters:&params];
```

在上述代码中，我必须用一个正方形来初始化精灵将会覆盖的区域。ccTexParams这个结构（struct）使用了设置为GL_REPEAT的包装好的参数来初始化（如果你不明白我在说什么，不用太担心）。然后这些OpenGL的参数会被用于精灵贴图的setTexParameters方法中。

最终结果是得到了一块用square.png不断重复出来的正方形区域（精灵）。如果你移动这个精灵，整块被repeatRect所覆盖的区域会一起移动。

你也可以使用上述技巧生成一块用较小的图片重复出来的图片区域，来替代之前屏幕最下放的那个条纹图片。你可以自己尝试一下。

虚拟控制手柄

因为iOS设备使用触摸屏来输入，没有传统移动游戏设备配备的按钮，十字按钮或者模拟手柄，我们需要一个虚拟手柄来控制游戏。你可以使用虚拟手柄对游戏物体进行操控，就像使用实际的手柄一样。**图7-8**展示了一个使用中的虚拟手柄。

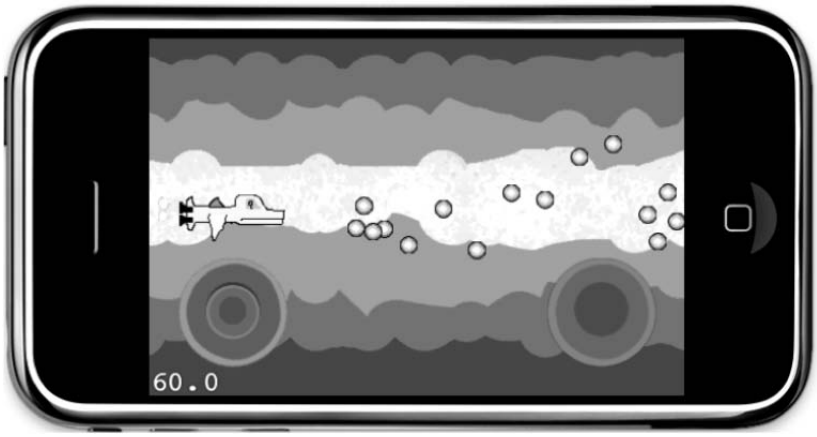

图7-8. 用SneakyInput制作，已经加上皮肤的虚拟按钮开关和射击按钮

介绍SneakyInput

随着时间的过去，很多开发者会面临开发虚拟控制手柄的问题。有很多方式可以实现虚拟手柄，但是你也可能非常容易失败。不过我们不需要开发虚拟手柄，我们可以使用现成的解决方案。

在编写任何看起来比较普遍的程序之前，总是搜索一下别人可能已经写过的程序，这样可以避免自己花费很多时间重新发明轮子。在我们的例子中，SneakyInput是一个不容忽视的选择。Nick Pannuto编写了开源的SneakyInput，你可以免费下载和使用。源代码托管在github.com（一个公开存放源代码的网站）：http://github.com/sneakyness/SneakyInput。如果你喜欢这套代码，可以考虑向Nick Pannuto捐款以支持他的工作：http://pledgie.com/campaigns/9124。

第一次从github下载代码时，你可能不是马上能找到下载链接，因为你首先看到的是一个代码文件列表，点击单个链接会显示具体的源代码。要下载整个开源代码包的话，你需要点击处于网站右上角的“Download Source”按钮。然后选择一个压缩格式进行下载。

下载完成以后你可以在Xcode中打开SneakyInput项目进行编译。因为SneakyInput已经整合了cocos2d源代码，所有很有可能它带的cocos2d不是最新的版本。你有可能会碰到第二章讨论过的“base SDK missing”问题。为了解决这个问题，你可以打开SneakyInput项目的Info面板，然后从General选项卡下选择一个合适的base SDK。如果一切顺利，你可以在模拟器中看到游戏界面

的左边有一个虚拟面板，右边有一个作为按钮的简单圆圈。点击虚拟面板移动可以让屏幕上的Hello World标签移动；如果点击右边的按钮，标签的颜色会发生变化。

图7-9展示了上述运行在iPhone模拟器中的SneakyInput范例项目。按钮还没有加上皮肤，而是使用了ColoredCircleSprite类。

图7-9. SneakyInput范例项目

当然，按钮都是可以加载皮肤的。加载皮肤的意思是给原来用简单的圆圈色块显示的按钮加上贴图。

整合SneakyInput

在ScrollingWithJoy05这个项目中，我已经整合了SneakyInput，因为我不想使用SneakyInput自带的项目。接下来我将会介绍如何将SneakyInput整合到你自己的项目中。

不光是SneakyInput，任何可以下载的类似源代码项目都会自带cocos2d源代码。它们带的cocos2d版本各自不同。大多数情况下，只要源代码用到的编程语言是Objective-C，你所要做的只是找到项目需要的文件，把它们放到自己的项目中去就可以了。不过因为每个项目都不一样，所以并不存在一个通用的指导方针。

不过，我可以告诉你需要将哪些SneakyInput项目中的文件添加到你的项目中。它们是SneakyInput的5个核心类：

1. SneakyButton和SneakyButtonSkinnedBase
2. SneakyJoystick和SneakyJoystickSkinnedBase
3. ColoredCircleSprite（可选）

除此之外的文件只是用来做参考的。**图7-10**展示了我在Add Existing Files对话框中选择的文件。我的选择逻辑是：通过猜测哪些是不需要的文件，然后只包含那些我认为需要的文件，最后在添加完文件以后，测试一下是否可以成功编译。

我猜HelloWorldScene类是由cocos2d的项目模板生成的，所以不需要它。我的

项目中已经有了一个AppDelegate，因此我就不需要SneakyInput的AppDelegate类了-它有可能和我的AppDelegate相冲突。然后我发现有两个类是以“Example”作为后缀名的，这表明它们不是SneakyInput的核心类文件，我们也不需要它们。

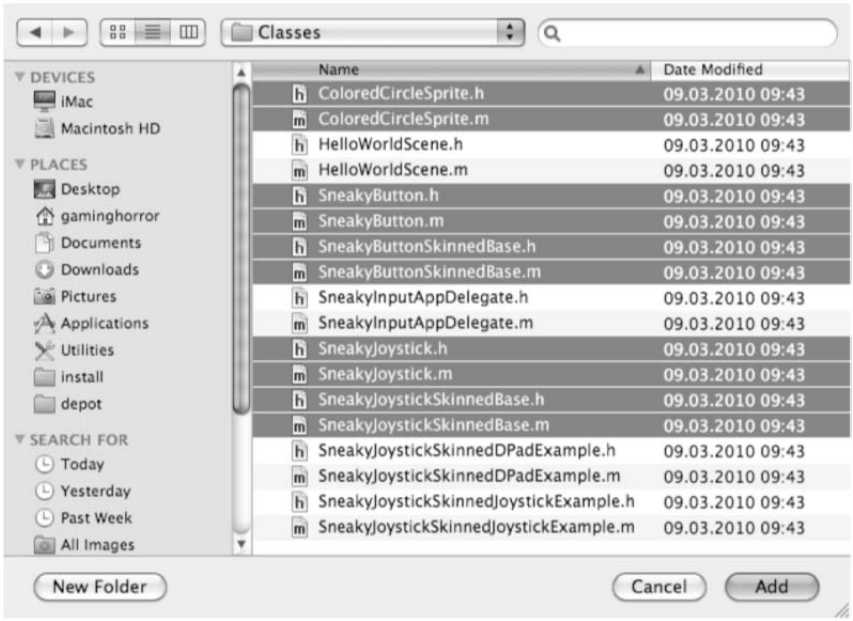

图7-10. 选中的文件是你的项目需要用到的，其它文件只是用来参考的例子文件。

触摸按钮进行射击

在ScrollingWithJoy05项目中完成添加SneakyInput源代码以后，我们的首要目标是添加一个可以让玩家进行飞船射击操作的按钮。他们点击此按钮时，子弹会从飞船中飞出。我将在项目中添加一个单独的InputLayer类。这个类继承自CCLayer，它会被添加到GameScene类中。**列表7-6**的代码中，我在GameScene类的scene方法中添加了一个新的InputLayer层。我给里面的两个层都添加了tag，这样以后有需要的话就可以通过tag来调用这两个层。

列表7-6. 将InputLayer添加到GameScene类中

```
+(id) scene
{
	CCScene* scene = [CCScene node];
	GameScene* layer = [GameScene node];
	[scene addChild:layer z:0 tag:GameSceneLayerTagGame];
	InputLayer* inputLayer = [InputLayer node];
	[scene addChild:inputLayer z:1 tag:GameSceneLayerTagInput];
	return scene;
}
```

新的tag是在GameScene类的头文件中定义的：

```
typedef enum
{
	GameSceneLayerTagGame = 1,
```

```
        GameSceneLayerTagInput,
} GameSceneLayerTags;
```

InputLayer设置完成以后，接下去是把我们需要用到的SneakyInput头文件都添加到InputLayer的头文件中去。这里我没有做任何挑选，因为有可能要用到大多数的类，所以我把SneakyInput的所有头文件都加了进来：

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"

// SneakyInput 头文件
#import "ColoredCircleSprite.h"
#import "SneakyButton.h"
#import "SneakyButtonSkinnedBase.h"
#import "SneakyJoystick.h"
#import "SneakyJoystickSkinnedBase.h"

@interface InputLayer : CCLayer
{
        SneakyButton* fireButton;
}
@end
```

另外，我在头文件中加了一个SneakyButton成员变量，因为我们马上就会用到。**列表7-7**中的addFireButton方法中，我用到了这个成员变量：

列表7-7. 生成一个SneakyButton按钮

```
-(void) addFireButton
{
        float buttonRadius = 80;
        CGSize screenSize = [[CCDirector sharedDirector] winSize];

        fireButton = [[[SneakyButton alloc] initWithRect:CGRectZero] autorelease];
        fireButton.radius = buttonRadius;
        fireButton.position = CGPointMake(screenSize.width - buttonRadius, buttonRadius);

        [self addChild:fireButton];
}
```

因为SneakyButton没有用到initWithRect方法中的CGRect参数，所以我传了一个CGRectZero给这个方法。实际的处理触摸事件的代码是使用radius（半径）这个属性来决定按钮是否要响应触摸。在我们的例子里，射击按钮需要被放置

在屏幕的右下角。通过用屏幕宽度减去按钮的半径，可以得到X轴的位置。通过把Y轴位置设为半径值，我们就可以把按钮放置在期望的位置上了。

技巧：通过使用buttonRadius（按钮半径）这个变量，我们可以在同一个地方修改按钮的半径值。这样做的好处是你可以避免在测试可用半径的过程中频繁的在多处修改按钮半径值，也可以避免一些微小的代码错误。

InputLayer类通过以下代码预约更新：

```
[self scheduleUpdate];
```

更新方法是用来测试按钮是否已被点击：

```
-(void) update:(ccTime)delta
{
	if (fireButton.active)
	{
		CCLOG(@"FIRE!!!");
	}
}
```

为了保证代码的简洁，我用日志信息代替了具体的子弹射击代码。如果你现在运行ScrollingWithJoy05项目，你会注意到屏幕上并没有显示任何按钮。不过当你点击屏幕的右下角时，你会在Xcode的调试窗口中看到 FIRE!!! 的输出。这样看来代码是工作了-除了按钮没有显示之外。我们现在来修正这个问题。

给按钮添加皮肤

添加皮肤（Skinning）在计算机图形学中是指给一个普通的物体加上贴图或者给它一个不同的视觉表现。在我们的例子中，我们想让按钮显示出来，所以我们需要一张图片作为皮肤。

在添加皮肤之前，和上一章一样，我要使用类别（Category）添加cocos2d风格的静态初始化方法（static initializer）到外部类中。通过使用类别中的方法，你就不会忘记在生成SneakyButton对象时发送alloc和autorelease信息了。首先我在ScrollingWithJoy06项目中添加一个SneakyExtensions类，头文件中的代码如下：

```
#import <Foundation/Foundation.h>

// SneakyInput 头文件
#import "ColoredCircleSprite.h"
#import "SneakyButton.h"
#import "SneakyButtonSkinnedBase.h"
#import "SneakyJoystick.h"
#import "SneakyJoystickSkinnedBase.h"
```

```
@interface SneakyButton (Extension)
+(id) button;
+(id) buttonWithRect:(CGRect)rect;
@end
```

我把所有的SneakyInput头文件都包含到了类别的头文件中，因为我计划在这里为每一个没有像cocos2d那样的初始化方法的SneakyInput类添加新的类别。在我们的例子里，SneakyInput的类别被命名为Extension，里面有两个方法：button和buttonWithRect。**列表7-8**是它们的实现代码：

列表7-8. SneakyInput的类别，包含了按钮的初始化方法

```
#import "SneakyExtensions.h"
@implementation SneakyButton (Extension)
+(id) button
{
	return [[[SneakyButton alloc] initWithRect:CGRectZero] autorelease];
}

+(id) buttonWithRect:(CGRect)rect
{
	return [[[SneakyButton alloc] initWithRect:rect] autorelease];
}
@end
```

上述两个方法只是简单地把alloc和autorelease调用封装了起来。我这样做的一个原因是因为CGRect这个参数没有被实际用到按钮的初始化过程中，通过封装，我们可以用以下代码轻松生成一个按钮：

```
fireButton = [SneakyButton button];
```

我们只是付出了一点点时间，得到的是很容易使用和干净的代码。我在SneakyExtensions中还添加了一些其它便于使用的方法，在这里就不一一详述了，因为它们的原理是一样的。

现在我可以开始为按钮添加皮肤了。我制作了4张100x100像素的按钮图片 - 两倍于按钮的半径（50个像素）。4个按钮所代表的状态分别是：默认状态（Default），点击状态（Pressed），激活状态（Activated）和失效状态（Disabled）。默认状态是在按钮没有被点击之前的状态，它和点击状态应该是有鲜明区别的。激活状态只用于开关按钮中，用于表示按钮处于“活动的”或者“开”的状态。失效状态用于表示按钮不会起作用。例如，当飞船的武器过热时，接下去的几秒钟你将不能再进行射击，这时你可以在按钮上使用失效状态的图片。对于我们的射击按钮，我现在只需要使用默认和点击状态这两张图片。**列表7-9**展示了更新过的 addFireButton 方法：

列表7-9. 使用以下代码替换列表7-7中的代码，生成添加皮肤后的按钮

```
float buttonRadius = 50;
CGSize screenSize = [[CCDirector sharedDirector] winSize];

fireButton = [SneakyButton button];
fireButton.isHoldable = YES;
SneakyButtonSkinnedBase* skinFireButton = [SneakyButtonSkinnedBase skinnedButton];
skinFireButton.position = CGPointMake(screenSize.width - buttonRadius, buttonRadius);
skinFireButton.defaultSprite = [CCSprite spriteWithSpriteFrameName: @"button-default.png"];
skinFireButton.pressSprite = [CCSprite spriteWithSpriteFrameName: @"button-pressed.png"];
skinFireButton.button = fireButton;
[self addChild:skinFireButton];
```

我用通常的方式对fireButton按钮进行初始化，唯一的区别是将它的isHoldable属性设置为YES。这个属性可以让玩家按住按钮不放的时候，子弹会持续地射击出去。我们也不再需要设置按钮的半径属性了，因为SneakyButtonSkinnedBase类会使用提供的按钮图片来确定按钮半径的大小。我在之前创建的SneakyExtensions源代码中添加了一个SneakyButtonSkinnedBase的类别。skinnedButton这个初始化方法就在那里，它封装了对alloc和autorelease的调用。

现在我们使用添加过皮肤的按钮来摆放屏幕上的按钮，而不是直接使用fireButton；实际的按钮则通过SneakyButtonSkinnedBase来更新。

现在我们来编写射击的代码。**列表7-10**中更新方法的代码会向GameScene类发送射击的信息了：

列表7-10. 当射击按钮处于激活状态时射出子弹

```
-(void) update:(ccTime)delta
{
	totalTime += delta;
	if (fireButton.active && totalTime > nextShotTime)
	{
		nextShotTime = totalTime + 0.5f;
		GameScene* game = [GameScene sharedGameScene];
		[game shootBulletFromShip:[game defaultShip]];
	}

	// 允许通过快速点击射击按钮来加快子弹射出的速度
	if (fireButton.active == NO)
	{
```

```
            nextShotTime = 0;
        }
}
```

代码中的两个ccTime类的变量：totalTime和nextShotTime，被用于限制飞船射出子弹的速度到每秒两颗，也就是飞船每秒最多只能射出两颗子弹。如果射击按钮不是处于激活状态（也就是还没有被点击），nextShotTime就被设为0，这样的话下次你点击按钮时，子弹肯定会被射出。快速点击按钮会在同一时间里比持续按住按钮不放时射出的子弹数量要更多。

控制动作

没有动作控制的话，你就不能操纵飞船的飞行。SneakyHolder可以帮助你控制飞船的飞行（通过生成一个虚拟控制杆）。我创建了ScrollingWithJoy07项目来展示所用的代码。

首先，我要给新的类添加一个Extension类别，这样我就可以向其他CCNode一样初始化新类了。**列表7-11**代码展示了如何创建一个加过皮肤的虚拟控制杆：

列表7-11. 增加一个加过皮肤的虚拟控制杆

```
-(void) addJoystick
{
        float stickRadius = 50;

        joystick = [SneakyJoystick joystickWithRect:CGRectMake(0, 0, stickRadius,
stickRadius)];

        joystick.autoCenter = YES;
        joystick.hasDeadzone = YES;
        joystick.deadRadius = 10;

        SneakyJoystickSkinnedBase* skinStick = [SneakyJoystickSkinnedBase
skinnedJoystick];
        skinStick.position = CGPointMake(stickRadius * 1.5f, stickRadius * 1.5f);
        skinStick.backgroundSprite = [CCSprite spriteWithSpriteFrameName: @"button-
disabled.png"];

        skinStick.backgroundSprite.color = ccGREEN;
        skinStick.thumbSprite = [CCSprite spriteWithSpriteFrameName: @"button-
disabled.png"];

        skinStick.thumbSprite.scale = 0.5f;
```

```
        skinStick.joystick = joystick;

        [self addChild:skinStick];
}
```

与SneakyButton相反，SneakyJoystick初始化的时候使用了CGRect。这里的CGRect用于决定虚拟手柄的半径大小。我将手柄设置为autoCenter（自动回到中心），这样的话拇指控制区域在手指放开以后，会自动回归到原先的位置，就像大多数真实世界中的游戏手柄一样。我们也启用了dead zone（死亡区域），这个区域由deadRadius这个变量所决定，在这个区域中，你可以移动拇指控制区域，但是不会产生任何效果。这样的话用户可以使用一部份半径区域来手动的把拇指控制区域归位了。

我把SneakyJoystickSkinnedBase放在离屏幕边界稍微有点距离的地方。虽然按钮的位置和大小对这个游戏来说并不是最优化的，但是我可以更好的展示虚拟手柄。如果我把虚拟手柄放在与屏幕边界对齐的地方，玩家的手指就很容易移到屏幕外面去而失去对飞船的控制。我选择了“button-disabled.png”作为虚拟手柄的皮肤图片，并且将包含此图片的精灵的背景设为绿色。thumbSprite（用于拇指控制区域）的大小缩放到虚拟手柄半径的一半大小。

技巧：使用像“button-disabled.png”这样的灰颜色图片的好处是：你可以用精灵的color属性来给图片着色。你可以用相同的图片生成红色，绿色，黄色，蓝色，洋红色等不同的颜色版本，从而节省加载时间和内存占用。唯一的缺点是得到的图片是实心的色块。这个技巧适用于那些本来就是用实心色块的图片。

当然，你是想用屏幕上的拇指控制区域来操纵飞船的移动的。**列表7-12**代码对更新方法做出了相应的修改：

列表7-12. 通过虚拟手柄来控制飞船的移动

```
-(void) update:(ccTime)delta
{
        …
        // 用虚拟手柄上的拇指控制区域来操纵飞船的移动
        GameScene* game = [GameScene sharedGameScene];
        Ship* ship = [game defaultShip];

        // 手柄的移动速度必须被放大才能让飞船的移动显得自然
        CGPoint velocity = ccpMult(joystick.velocity, 200);
        if (velocity.x != 0 && velocity.y != 0)
        {
                ship.position = CGPointMake(ship.position.x + velocity.x * delta,
                                                        ship.position.y + velocity.y * delta);
        }
}
```

我在GameScene类中添加了一个名为defaultShip的访问方法，这样的话InputeLayer就可以访问飞船了。虚拟手柄的移动速度用于改变飞船的位置，但是由于手柄的移动速度值只是1个像素的几分之一，所以我们必须使用cocos2d的ccpMult方法将其放大，从而让屏幕上的飞船产生明显的移动。ccpMult方法以一个CGPoint和一个浮点数作为参数。你可以使用任意的浮点数作为放大的因子（我在这里使用了200），这个值的选择纯凭感觉。

cocos2d传入的delta参数用于表示上次调用更新方法与现在调用更新方法之间的时间间隔。因为游戏运行时的帧率是在随时变化的，所以你并不能保证这个时间间隔每次都是一样的。为了保证即使在更新方法调用之间的时间间隔不一样的情况下，飞船也可以平滑地移动，我把delta也加到了位置移动的代码中。如果你不考虑delta这个因素的话，飞船在帧率掉到60fps以下的时候，就会飞的比之前慢。像这样的事情会招致玩家很大的反感。作为游戏开发者，我们最忌讳的就是玩家反感我们的游戏。

目前为止，我们的飞船还是可能移动到屏幕外面去。我猜你和我一样，都希望飞船可以一直待在屏幕里面。你有可能想直接向InputeLayer类里飞船位置的更新方法中添加相关的代码。这就带给我们一个问题：我们是应该控制虚拟手柄的移动以防止飞船飞出屏幕；还是直接控制飞船的位置，让它永远都不能移动到屏幕外面去呢？对于我们的游戏，后者是更好的选择。如**列表7-13**代码所示，你只需要把Ship类中的setPosition方法重写即可：

列表7-13. 重写Ship类中的setPosition方法

```
// 重写 setPosition 方法，将飞船保持在屏幕可视范围之内
-(void) setPosition:(CGPoint)pos
{
        CGSize screenSize = [[CCDirector sharedDirector] winSize];
        float halfWidth = contentSize_.width * 0.5f;
        float halfHeight = contentSize_.height * 0.5f;

        // 设置飞船精灵的位置将其保持在屏幕可视范围之内
        if (pos.x < halfWidth)
        {
                pos.x = halfWidth;

        }else if (pos.x > (screenSize.width - halfWidth))
        {
                pos.x = screenSize.width - halfWidth;
        }

        if (pos.y < halfHeight)
        {
```

```
            pos.y = halfHeight;
        }else if (pos.y > (screenSize.height - halfHeight))
        {
            pos.y = screenSize.height - halfHeight;
        }

        // 将super设定到新的位置
        [super setPosition:pos];
}
```

每次更新过飞船的位置以后，上述代码会检查飞船是否还在屏幕的可视范围之内。如果飞船已经飞出屏幕边界，飞船精灵的X或Y轴的位置会被移动到离开飞出的屏幕边界半个飞船精灵（contentSize）宽度或高度的距离。

因为位置（position）是一个属性，通常setPosition方法是通过以下代码来调用的：

```
ship.position = CGPointMake(200, 100);
```

ship和position之间的点是Objective-C中用于获取和设置属性的方法的简写。对于设置方法，你可以用如下代码重写：

```
[ship setPosition:CGPointMake(200, 100)];
```

你可以用上述方法重写游戏对象的基类方法。例如，如果你想让一个对象只能在0到180度之间旋转，你可以重写它的setRotation:(float)rotation方法来限制对象的旋转。

数字控制

如果虚拟手柄不适用于你的游戏呢？你也可以把SneakyJoystick类变成一个数字控制器，通常称为D-pad。你可以在ScrollingWithJoy08项目中找到相关代码。所需要的代码修改很少：

```
joystick = [SneakyJoystick joystickWithRect:CGRectMake(0, 0, stickRadius, stickRadius)];
joystick.autoCenter = YES;

// 限制可移动的方向数量
joystick.isDPad = YES;
joystick.numberOfDirections = 8;
```

在上述代码中，dead zone(死亡区域)这个属性被移除了，因为在D-pad中不需要。通过设置isDPad为YES，我们将虚拟手柄变成了一个D-Pad。你可以定义可移动方向的数量。通常D-Pad可以朝4个方向移动，在很多游戏中你可以同时按住两个方向的按钮使角色向两个方向的对角线方向移动。为了得到相同的效果，我将numberOfDirections属性设定为8。SneakyJoystick会自动计算这些方向，保证游戏角色向正确的方向移动。当然，如果你把numberOfDirections设为6的话，你将会得到很奇怪的移动方向。不过，在一个六边形地板拼成的地图中，

你就需要将numberOfDirections设定为6。

替代品：GPJoystick

除了SneakyInput之外，GPJoystick也具有相同的功能。不过它不是免费的，虽然价格很便宜。你可以在以下网址找到：http://wrensation.com/?p=36

如果你想了解GPJoystick的工作原理和它与SneakyInput的不同之处，你可以看一下YouTube网站上SDKTutor用户的视频。此用户也有好几个关于cocos2d的视频教程。你可以通过以下网址访问：http://www.youtube.com/user/SDKTutor

结语

我们在本章学习了几个制作视差滚屏背景的技巧。你不仅学习了如何让背景进行无限滚屏和避免在边界处产生闪烁，而且学习了如何合理地分开不同的视差图层，以便 Zwoptex 软件可以清除不必要的透明区域和保存各个图层之间的位移信息，这样你就不用费时费力的在代码中放置各个图层了。

上述方法的缺点是：它针对的是固定的屏幕尺寸。如果你想生成一个 iPad 版本的话，你可以使用相同的方法，但是你要制作 1024x768 像素的相关图片。我留给你去做这个练习。

在下半章中我介绍了可用于 cocos2d 游戏的开源项目“SneakyInput”。你可以用它来制作虚拟的手柄和按钮。虽然不是完美的解决方案，但是大多数游戏已经够用了，而且绝对比你自己写虚拟手柄的代码来的省时省力。

现在，我们的飞船已经可以用虚拟手柄来控制移动，而且一直会待在屏幕的可视范围之内。如果你点击虚拟按钮，飞船还会射出子弹。不过我们还缺一点东西。一个射击游戏如果没有射击目标的话，就称不上是射击游戏了。在下一章中，我会补上这个射击的目标。