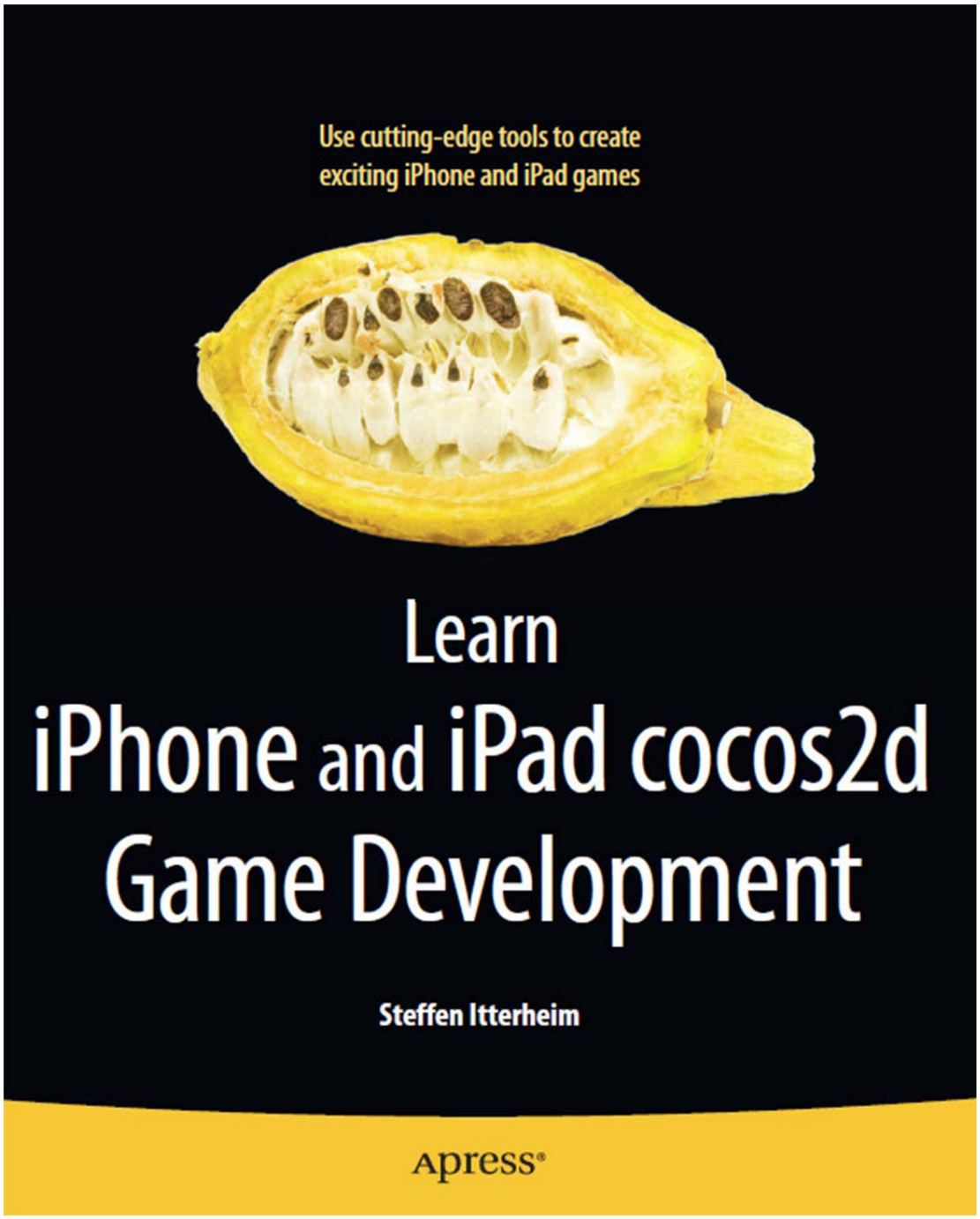

译者：杨栋
邮箱：yangdongmy@gmail.com

第五章
游戏构成要素

上一章的DoodleDrop例子对于cocos2d的新用户来说比较容易理解。但是如果你是一位有经验的用户，你会发现我没有把代码分开；所有的代码都写在了一个文件里。很明显，这样的代码是不适用于更大一些，更令人激动的游戏的。我们必须使用更合适的方式来组织代码才行。否则，你的游戏的最终代码可能都堆积在一个类里面，几千条代码会让修改和维护代码变的非常困难，而且很容易产生难于发现的错误。

每个新项目都需要它自己的代码设计。本章我会介绍一些用于编写更复杂的cocos2d游戏的构成要素。本章介绍的游戏构成要素会在接下去的几章里用于编写我们的横向卷轴射击游戏（side-scrolling shooter game）。

使用多个场景

上一章的DoodleDrop游戏只有一个场景和一个层。更复杂的游戏会有多个场景和层存在。使用这些场景和层将会成为你的第二天性。让我们来看看需要做些什么。

添加更多的场景

在上一章的**列表4-1**和**4-2**中，我列出了用于创建场景的基础代码。你将通过创建更多基于那些基础代码的类来创建更多的场景。当你要在不同场景之间进行转换时，你将使用新的代码。当你用CCDirector replaceScene方法替换场景时，每个节点都会调用CCNode所带的三个方法。这三个方法是：onEnter，onEnterTransitionDidFinish和onExit。取决于是否使用了CCTransitionScene，onEnter和onExit会在场景转换过程中的某个时间点被调用。对于这三个方法，你必须调用它们的super方法以避免触摸输入问题和内存泄漏的问题。从**列表5-1**的代码中你可以看到所有的方法都调用了它们自己的super方法：

列表5-1. onEnter，onEnterTransitionDidFinish和onExit方法

```
-(void) onEnter
{
        // 节点调用init方法以后将会调用此方法
        // 如果使用了CCTransitionScene，将会在过渡效果开始以后调用此方法
        [super onEnter];
}

-(void) onEnterTransitionDidFinish
{
        // 调用onEnter以后将会调用此方法
        // 如果使用了CCTransitionScene，将会在过渡效果结束以后调用此方法
        [super onEnterTransitionDidFinish];
}

-(void) onExit
```

```
{
        // 节点调用dealloc方法之前将会调用此方法
        // 如果使用了CCTransitionScene，将会在过渡效果结束以后调用此方法
        [super onExit];
}
```

注：如果你不在onEnter方法里调用它的super方法的话，你的新场景可能不会对触摸或者加速计的输入有任何反应。如果你不在onExit方法里调用它的super方法，当前场景可能不会从内存里释放。因为很容易忘记添加super方法，而且在发生问题以后也很难与没有调用super方法联系起来，我要在这里特别强调这一点。你可以在ScenesAndLayer01项目里看到上述问题。

你可以在场景转换之前或者之后，通过使用上述方法在节点中完成一些特定的操作。因为在程序进入onEnter方法的时候，场景中的所有节点都已经设置完成了；同时，在onExit方法中，所有节点都还存在于内存中。

这一点很重要。如果你希望在场景转换的过程中使用过渡效果的话，你可能想先暂停某些动画或者隐藏一些用户界面元素，直到过渡效果结束。这就需要所有的节点都存在于当前的场景中。以下来自ScenesAndLayers02项目的日志信息显示了上述三个方法的调用次序：

```
1. scene: OtherScene
2. init: <OtherScene = 066B2130 | Tag = -1>
3. onEnter: <OtherScene = 066B2130 | Tag = -1>
4. // 这里运行了过渡效果
5. onExit: <FirstScene = 0668DF40 | Tag = -1>
6. onEnterTransitionDidFinish: <OtherScene = 066B2130 | Tag = -1>
7. dealloc: <FirstScene = 0668DF40 | Tag = -1>
```

首先，OtherScene 的+(id)scene 方法被调用，用于初始化它包含的 CCScene 和 CCLayer。然后 OtherScene 里 CCLayer 的 init 方法被调用。紧接着在第三行调用 onEnter 方法。在第四行，通过使用过渡效果来使新场景移动进入。当新场景移动进入完成以后，FirstScene 的 onExit 方法被调用，最后是调用 OtherScene 的 onEnterTransitionDidFinish 方法。

你会注意到 FirstScene 的 dealloc 方法是最后一个被调用的。这意味着在 onEnterTransitionDidFinish 方法的运行过程中，之前的场景仍在内存里。如果你想在 onEnterTransitionDidFinish 方法里分配很耗内存的节点的话，你必须先预约一个方法（或者“选择器”）（selector），至少等到下一帧时才分配内存，这样你可以确保之前的场景已从内存中释放。另一个策略是在上一个场景的 onExit 方法里释放尽可能多的内存。

正在加载下一行，请等待…

迟早你将面对场景转换过程中过长的加载时间。当你加入越来越多的内容时，加载的时间也会相应的增加。场景的生成其实早于场景过渡效果的开始。所以如果你需要在新场景的 init 或者 onEnter 方法中执行很复杂的代码或加载很多素材的话，过渡效果开始之前就会产生很明显的延迟。假设用户是通过点击按

钮来进行场景转换的，如果上述延迟超过一秒，它就会让用户觉得游戏卡住了。解决这个问题的方法是在两个场景之间添加一个加载场景。ScenesAndLayers03项目提供了一个实现的样例。

实际上，LoadingScene是一个中间场景。它继承自cocos2d的CCScene类。你没有必要为每一个过渡效果创建一个新的LoadingScene场景。你可以通过指定需要加载的目标场景来使用同一个LoadingScene场景。枚举类型（enum）很适合我们的这个需求。如**列表5-2**所示，枚举类型定义在LoadingScene的头文件里：

列表5－2. LoadingScene.h

```
typedef enum
{
      TargetSceneINVALID = 0,
      TargetSceneFirstScene,
      TargetSceneOtherScene,
      TargetSceneMAX,
} TargetScenes;

// LoadingScene直接继承自CCScene，在这个场景中我们不需要CCLayer
@interface LoadingScene : CCScene
{
      TargetScenes targetScene_;
}

+(id) sceneWithTargetScene: (TargetScenes) targetScene;
-(id) initWithTargetScene: (TargetScenes) targetScene;
```

技巧：将枚举类型的第一个值设置为INVALID值是个好习惯，除非你想把第一个元素设为默认的。除非你指定一个不同的值，Objective-C的变量会被自动初始化为0。

另外，你也可以添加一个MAX或者NUM到枚举类型的最后，就像下面的代码一样：`for (int i = TargetSceneINVALID + 1; i < TargetScenesMAX; i++) { .. }`

在LoadingScene里面我们MAX或者NUM，但是有时候我加上它们是出于习惯，即使我不需要它们。

接下来我们看一下**列表5-3**中ScenesAndLayers03项目的LoadingScene类的实现。你会注意到这里的场景初始化不大一样。我们使用了scheduleUpdate方法来延缓目标场景对LoadingScene场景的替换。

列表5-3. LoadingScene类使用了switch逻辑来决定是否加载目标场景

```
+(id) sceneWithTargetScene:(TargetScenes)targetScene;
{
        // 以下代码生成了一个当前类的自动释放对象 (self == LoadingScene)
        return [[[self alloc] initWithTargetScene:targetScene] autorelease];
}

-(id) initWithTargetScene:(TargetScenes)targetScene
```

```
{
        if ((self = [super init]))
        {
                targetScene_ = targetScene;

                CCLabel* label = [CCLabel labelWithString:@"Loading ..."
                                                fontName:@"Marker Felt" fontSize:64];
                CGSize size = [[CCDirector sharedDirector] winSize];
                label.position = CGPointMake(size.width / 2, size.height / 2);
                [self addChild:label];

                // 必须在下一帧才加载目标场景!
                [self scheduleUpdate];
        }
        return self;
}

-(void) update:(ccTime)delta
{
        [self unscheduleAllSelectors];
        // 通过TargetScenes这个枚举类型来决定加载哪个场景
        switch (targetScene_)
        {
                case TargetSceneFirstScene:
                        [[CCDirector sharedDirector] replaceScene:[FirstScene scene]];
                        break;

                case TargetSceneOtherScene:
                        [[CCDirector sharedDirector] replaceScene:[OtherScene scene]];
                        break;
                default:
                        // 如果使用了没有指定的枚举类型，发出警告信息
                        NSAssert2(nil, @"%@: unsupported TargetScene %i",
                                                NSStringFromSelector(_cmd), targetScene_);
                        break;
        }
}
```

因为 LoadingScene 继承自 CCScene，并且要求传递一个新的参数给它，所以仅仅调用[CCScene node]是不够的。SceneWithTargetScene 方法首先会完成对 self 的内存分配，然后调用 initWithTargetScene 方法，最后返回一个新的自动释放对象。这和 cocos2d 初始化自身类的方式是一样的。在我们的例子里，因为你是在创建自己的类的对象，所以你必须添加静态自动释放方法：autorelease。

这里的 init 方法把目标场景储存在一个成员变量里面，接着生成一个“Loading…”标签，最后运行 scheduleUpdate 这个预约方法。

注：为什么我们不在init方法里直接使用replaceScene方法呢？这里有两条规则需要遵守。规则一:永远不要在一个节点的init方法中调用CCDirector的replaceScene方法。规则二：请遵守规则一。不遵守规则的后果是程序崩溃。Director无法容忍一个节点在初始化的同时进行场景替换。

接着，update方法使用了一个简单的switch，通过传进来的TargetScenes枚举

类型参数来判断用于替换的目标场景。switch的默认情况包含了一个NSAssert。在你编辑和扩展上述枚举类型几次以后，如果你忘记更新switch代码的话，你将会得到这个默认的警告信息。

你可以在自己的游戏里使用这个简单的LoadingScene。只要扩展一下枚举类型和switch代码就可以了。不过我要提醒你的是：不要因为好看而过多的使用过渡效果。

在过渡效果中使用LoadingScene可以优化内存的使用：因为你使用了一个简单的过渡场景用于替换当前场景，然后用最终的目标场景替换这个过渡场景。在这个替换的过程中，cocos2d将会有足够的时间来释放之前场景所占用的内存。我们得到的实际效果是：不再会有两个复杂场景同时占用着内存的情况了，因此在场景转换过程中也就减少了出现内存使用高峰的机会。

使用多个层

ScenesAndLayers04 项目展示了如何通过使用多个层，在游戏物体层移动的同时保持用户界面层（“Here be your Game Scores”区域）静止不动（**图 5-1**）。你将学习如何设置层，使它仅对自己接收到的触摸输入做出反应。你也将学习如何从不同的节点访问这些层。

图5－1. ScenesAndLayers04项目

在MultiLayerScene的init方法里，我们将设置多个层。如果你很快地看一遍**列表5-4**中的代码，你会发现我们在这里使用的方法和之前学习过的没有任何区别：

列表5-4. 初始化MultiLayerScene

```
-(id) init
{
        if ((self = [super init]))
        {
                multiLayerSceneInstance = self;
```

```
                // GameLayer层可以被独立的移动，旋转和缩放
                GameLayer* gameLayer = [GameLayer node];
                [self addChild:gameLayer z:1 tag:LayerTagGameLayer];
                gameLayerPosition = gameLayer.position;

                // UserInterfaceLayer层会保持静止
                UserInterfaceLayer* uiLayer = [UserInterfaceLayer node];
                [self addChild:uiLayer z:2 tag:LayerTagUserInterfaceLayer];

        }

        return self;
}
```

技巧：这里要提一下使用枚举类型的好处。上述代码中的 LayerTagGameLayer 就是一个枚举类型。使用枚举类型以后你就不需要再去记忆哪个节点使用了哪个数字作为 tag，你可以清楚的知道哪个节点使用了哪个枚举类型。实际上使用什么数字并不重要；重要的是同一个节点要统一使用同一个数字。同样的方法可以在动作（Action）tag 上使用。

你可能注意到我们将 self 赋值给了 multiLayerSceneInstance 这个变量。很奇怪，我们为什么要这样做呢？如果你还记得在第三章我们讨论了如何生成一个单例类（Singleton），你会明白我们是要把 MultiLayerScene 类变成一个单例。你可以将**列表 5-5** 的代码和**列表 3-1** 的代码做一下比较，看有什么不同之处：

列表 5-5. 把 MultiLayerScene 变成一个“半单例”对象：

```
static MultiLayerScene* multiLayerSceneInstance;

+(MultiLayerScene*) sharedLayer
{
        NSAssert(multiLayerSceneInstance != nil, @"MultiLayerScene not available!");
        return multiLayerSceneInstance;
}

-(void) dealloc
{
        // MultiLayerScene将在这里被释放，你必须将multiLayerSceneInstance变量设为nil
        // 否则它将指向一个已经被释放的对象
        multiLayerSceneInstance = nil;

        // 不要忘记调用"super dealloc"
        [super dealloc];
}
```

简单点说，multiLayerSceneInstance是个静态的全局变量，它将保存当前有效的MultiLayerScene对象直到此对象被释放。在MultiLayerScene的层对象被释放以后，multiLayerSceneInstance这个变量必须被设为nil，否则它将指向一个已经被释放的对象，导致程序崩溃。使用这样的“半单例”的原因是我们将会使用多个层，每个层都有自己的一些子节点，而我们又需要一个访问主层的途径。使用这个方式可以达到我们的目的。

注：使用这个“半单例”（semi-singleton）的前提是：任何时候只存在一个MultiLayerScene的实例。与普通的单例类不同，它不能被用来初始化MultiLayerScene。

我们通过获取方法（getter methods）来访问GameLayer和UserInterfaceLayer。**列表5-6**定义了和MultiLayerScene.h相关的两个属性：

列表5-6.用于访问GameLayer和UserInterfaceLayer的两个属性定义

```
@property (readonly) GameLayer* gameLayer;
@property (readonly) UserInterfaceLayer* uiLayer;
```

因为我们只需要获取层，而不需要通过属性来配置层，所以这两个属性被设定为只读（readonly）。**列表5-7**的代码是对getChildByTag方法的简单封装，不过它们也提供了对获得对象的类型检查，以保证得到的是我们想要的类：

列表5-7. 属性获取方法的具体实现

```
-(GameLayer*) gameLayer
{
        CCNode* layer = [self getChildByTag:LayerTagGameLayer];
        NSAssert([layer isKindOfClass:[GameLayer class]], @"%@: not a GameLayer!",
                                                        NSStringFromSelector(_cmd));

        return (GameLayer*)layer;
}

-(UserInterfaceLayer*) uiLayer
{
        CCNode* layer = [[MultiLayerScene sharedLayer] getChildByTag:LayerTagUILayer];
        NSAssert([layer isKindOfClass:[UserInterfaceLayer class]],
                                        @"%@: not a UILayer!", NSStringFromSelector(_cmd));

        return (UserInterfaceLayer*)layer;
}
```

上述代码让我们可以在MultiLayerScene里的任何一个节点中访问gameLayer和uiLayer。

1. 以下代码可以得到MultiSceneLayer的场景层：
   ```
   MultiSceneLayer* sceneLayer = [MultiSceneLayer sharedLayer];
   ```
2. 以下代码通过场景层得到其它的层：
   ```
   GameLayer* gameLayer = [sceneLayer gameLayer];
   UserInterfaceLayer* uiLayer = [sceneLayer uiLayer];
   ```
3. 因为我们使用了@property定义，所以也可以通过点访问子（dot accessor）来获取层：
   ```
   GameLayer* gameLayer = sceneLayer.gameLayer;
   UserInterfaceLayer* uiLayer = sceneLayer.uiLayer;
   ```

UserInterfaceLayer和GameLayer都处理触摸输入事件，但是它们有各自的处理

逻辑。为了正确地处理各自的触摸输入事件，我们需要用到TargetedTouchHandlers。通过使用优先级参数，我们可以确保UserInterfaceLayer在GameLayer之前接收到触摸事件。UserInterfaceLayer利用isTouchForMe方法来决定它是否需要处理接收到的触摸事件，如果它处理了触摸事件，ccTouchBegan方法将会返回YES。如果返回了YES，其它层，也就是其它的触摸事件处理器，将不会再接收到当前事件。**列表5-8**展示了UserInterfaceLayer中用于处理触摸事件的代码：

列表 5-8. UserInterfaceLayer 用于处理触摸事件的代码 *e*

```
// 将UserInterfaceLayer的触摸事件处理器的优先级设置的比GameLayer高
-(void) registerWithTouchDispatcher
{
        [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self priority:-1
                                                                swallowsTouches:YES];
}

// 检查是否触摸到的地方是在用户界面元素的区域里面
-(bool) isTouchForMe:(CGPoint)touchLocation
{
        CCNode* node = [self getChildByTag:UILayerTagFrameSprite];
        return CGRectContainsPoint([node boundingBox], touchLocation);
}

-(BOOL) ccTouchBegan:(UITouch*)touch withEvent:(UIEvent *)event
{
        CGPoint location = [MultiLayerScene locationFromTouch:touch];
        bool isTouchHandled = [self isTouchForMe:location];
        if (isTouchHandled)
        {
                CCNode* node = [self getChildByTag:UILayerTagFrameSprite];

                NSAssert([node isKindOfClass:[CCSprite class]], @"node is not a CCSprite");

                // 在触摸期间，将用户界面层的精灵高亮显示
                ((CCSprite*)node).color = ccRED;

                // 通过MultiLayerScene访问GameLayer
                GameLayer* gameLayer = [MultiLayerScene sharedLayer].gameLayer;

                // 运行GameLayer层的动作… (为了代码清晰而移除了不相关的代码)
        }

        return isTouchHandled;
}

-(void) ccTouchEnded:(UITouch*)touch withEvent:(UIEvent *)event
{
        CCNode* node = [self getChildByTag:UILayerTagFrameSprite];
        NSAssert([node isKindOfClass:[CCSprite class]], @"node is not a CCSprite");
        ((CCSprite*)node).color = ccWHITE;
}
```

在registerWithTouchDispatcher方法中，UserInterfaceLayer把自己注册为优先级为-1的目标触摸事件处理器。因为GameLayer使用了相同的代码，但是优先级设为0，所以UserInterfaceLayer将会第一个接收到触摸输入事件。

在ccTouchBegan方法中，第一件事是检查接收到的触摸事件是否和UserInterfaceLayer相关。IsTouchForMe方法通过使用CGRectContainsPoint方法来检查接收到的触摸是否落在了uiframe精灵里面，也就是用户界面的区域里面。这是个简单的“点落在边界框中”的检查。在CGGeometry里还有好几个更有用的用于检查相交，点包含和相等的方法。你可以通过苹果的文档学习CGGeometry里的所有方法：
http://developer.apple.com/library/mac/#documentation/GraphicsImaging/Reference/CGGemetry/Reference/reference.html

如果触摸点落在了用户界面精灵上，ccTouchBegan将返回YES，表示用户界面层处理了当前的触摸事件，此事件不应该再被其它拥有低优先级的目标触摸代理的层进行处理。

只有当用户界面的isTouchForMe测试失败以后，GameLayer才会接收到触摸事件，并且用此事件让它自己随着手指在屏幕上的移动而移动。**列表5-9**是GameLayer用于处理触摸输入事件的代码：

列表5-9：GameLayer接收到用户界面没有处理过的触摸事件，并且利用触摸输入让自己移动

```
-(void) registerWithTouchDispatcher
{
        [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self priority:0
                                                          swallowsTouches:YES];
}

-(BOOL) ccTouchBegan:(UITouch*)touch withEvent:(UIEvent *)event
{
        lastTouchLocation = [MultiLayerScene locationFromTouch:touch];

        // 停止之前的移动动作，以免干扰用户的滚动操作
        [self stopActionByTag:ActionTagGameLayerMovesBack];

        // 总是吞没触摸事件，GameLayer是最后一个接收到触摸事件的层
        return YES;
}

-(void) ccTouchMoved:(UITouch*)touch withEvent:(UIEvent *)event
{
        CGPoint currentTouchLocation = [MultiLayerScene locationFromTouch:touch];

        // 计算当前和之前的触摸之间的距离
        CGPoint moveTo = ccpSub(lastTouchLocation, currentTouchLocation);

        // 将计算结果乘以-1，以产生背景在移动的效果
        moveTo = ccpMult(moveTo, -1);

        lastTouchLocation = currentTouchLocation;

        // 调整层相应的位置，包括层所包含的所有子节点
        self.position = ccpAdd(self.position, moveTo);
}
```

```
-(void) ccTouchEnded:(UITouch*)touch withEvent:(UIEvent *)event
{
        // 将GameLayer移回原先指定的位置
        CCMoveTo* move = [CCMoveTo actionWithDuration:1 position:gameLayerPosition];
        CCEaseIn* ease = [CCEaseIn actionWithAction:move rate:0.5f];
        ease.tag = ActionTagGameLayerMovesBack;
        [self runAction:ease];
}
```

因为 GameLayer 最后一个接收到触摸事件的层，所以它不需要做任何 isTouchForMe 的测试，而是直接处理所有接收到的触摸事件。

利用 ccTouchMoved 事件，我们可以计算获得之前和当前触摸点的距离。然后通过将得到的距离乘以-1，我们可以产生背景在移动的效果。如果你想像不出来实际的效果，试着运行 ScenesAndLayers04 项目。在第二次运行前，将 moveTo = ccpMult(moveTo,-1);这段代码去掉，你会看到层移动的方向是和手指移动的方向相反的。

在 ccTouchEnded 事件中，当用户的手指离开屏幕以后，游戏层会自动移回原先的位置。**图 5-2** 显示了整个 GameLayer 层经过了旋转和画面拉远。在 GameLayer 上的所有物体都随着 GameLayer 层移动，旋转和缩放，而 UserInterfaceLayer 层则保持静止不动。

图5-2. 我们可以在这里看到多个图层的好处。GameLayer层上的所有子节点会随着GameLayer层的旋转而旋转，也会随着它的画面拉远而拉远。同时UserInterfaceLayer层则保持静止不动。

如何以最好的方式实现游戏关卡

到目前为止，我们讨论了多个场景和多个层的应用。现在我们来讨论关卡。

大家应该对关卡的概念都很熟悉，所以我不在这里做解释了。设计关卡的难点是决定用什么样的方式来设计基于关卡的游戏。在cocos2d里，你可以选择每一

个关卡都使用一个场景，或者在同一个场景里设置多个层作为关卡。选择哪种方式取决于关卡在你游戏中的作用。

每一个关卡都使用一个场景

最直接的方法是每一个关卡都使用一个单独的场景。你可以为每一个关卡创建一个新的场景类（Scene class），或者通过把关卡号码或其它与关卡相关的信息传递给一个通用的关卡场景类（LevelScene class）用于加载正确的关卡数据。

这个方法很适合每个关卡之间没有多大关系，关卡可以清晰地分开的情况。你可能需要保存玩家的得分和当前剩余的生命数，除此之外不需要再保存其它的信息了。这种情况下的用户界面可能是静态的，或者最多只有一个暂停按钮。

同一个场景里设置多个层作为关卡

如果你有一个复杂的用户界面，并且在关卡转换时不允许重置游戏场景的话，你就需要在同一个场景里用不同的层来加载和显示不同的关卡。在转换关卡时，你可能会将游戏角色和其它的游戏物体保持在同一个位置。

你可能使用几个变量来保存当前的游戏状态信息和用户界面设置信息，比如说仓库里的物品。如果你使用多个场景作为关卡的话，在转换关卡的时候，你就需要做很多工作来保存和重置游戏设置，还有重置所有的视觉元素。

这个方式很适合物品寻找类或者冒险类游戏，因为玩家需要在不同的房间之间移动。特别是你想在替换关卡内容的过程中使用动画的情况下。

CCMultiplexLayer类可被用于这种方式中。它可以同时包含多个节点，但是任意时间里只有一个节点是有效的。**列表5-10**展示了一个CCMultiplexLayer的应用。唯一的缺点是你不能在层之间使用过渡效果。因为任意时间里只能有一个层是可视的，所以任何过渡效果都是不可能发生的。

列表5-10. 使用CCMultiplexLayer类在不同层之间进行转换

```
CCLayer* layer1 = [CCLayer node];
CCLayer* layer2 = [CCLayer node];
CCMultiplexLayer* mpLayer = [CCMultiplexLayer layerWithLayers:layer1, layer2, nil];

// 转换到layer2，layer1还是mpLayer的子节点
[mpLayer switchTo:1];

// 转换到layer1，从mpLayer里移除layer2，并且释放layer2占用的内存
// 在使用下述方法以后，你不能够再转换回layer2（索引：1）了
[mpLayer switchToAndReleaseMe:0];
```

CCColorLayer

在ScenesAndLayers项目里，游戏背景只是简单的黑色。你可以将屏幕滚动到草坪背景图片的边界，或者点击用户界面让GameLayer层视图缩小，就可以看到后

面的黑色背景了。cocos2d提供了CCColorLayer类可用于改变背景颜色。在ScenesAndLayers05项目中，我们加入了以下代码：

```
// 将背景色设为紫红色。
CCColorLayer* colorLayer = [CCColorLayer layerWithColor:ccc4(255, 0, 255, 255)];
[self addChild:colorLayer z:0];
```

如果你熟悉OpenGL的话，你可能会觉得没有必要为了改变背景颜色而添加一个单独的层。你可以使用以下OpenGL代码得到相同的效果：

```
glClearColor(1, 0, 1, 1);
```

但是如果你在场景中使用了过渡效果的话，你最好先在游戏里测试一下上述OpenGL代码，因为glClearColor对场景过渡效果会产生不好的影响。比如，当你使用CCFadeTransition时，不管你用了什么颜色初始化CCFadeTransition，glClearColor中使用的颜色都会透过层影响到过渡效果中使用的颜色。

继承自CCSprite的游戏物体类（Game Objects）

很多时候你游戏中的物体会有自己的处理逻辑，所以为每一种游戏物体生成一个独立的类就是理所当然的事了。这些游戏物体可能是玩家角色，各种各样的敌人，子弹，导弹，平台等等，包括所有放在游戏场景中，有着自己处理逻辑的物体。

问题是，这些物体应该继承自哪个类呢？

很多开发者会选择CCSprite。但我不认为那是个好主意。继承是“是一个”的关系，那么你觉得角色是一个CCSprite吗？所有的敌人角色都是CCSprite吗？

乍一看上去确实是这样：它们都是CCSprite，因为它们就是通过CCSprite来显示自己的。不过这些角色还可能是别的东西吗？因为我知道在“Rogue-like”游戏中，游戏角色是用@字符来代替的。这样的话，角色就是一个CCLabel了？

我想造成混淆的原因是：最常用来将物体显示在屏幕上的类是CCSprite。但是实际上你的游戏角色和CCNode类之间的关系是“有一个”的关系。你的游戏角色类“有一个”CCSprite用于显示它自己。在一个“Rogue-like”游戏中，游戏角色类“有一个”CCLabel用于显示它自己。如果你使用OpengGL和许多粒子效果的话，你的角色类就“有一个”粒子效果系统将它自己显示在屏幕上。

当你认识到为什么要继承自CCSprite类的原因时，你就可以认识到为什么继承自CCSprite不是个好主意了。继承自CCSprite的原因是要添加新的功能。那么你想要增加什么功能呢？输入处理，角色动画，碰撞检测等等的游戏逻辑。而CCSprite并没有任何上述功能。

你可能还是不明白这和你的关系，你会问：我需要关心这些事情吗？现在我假设你想让你的角色有好几种不同的视觉表现，你想让这几种视觉表现之间可以无缝切换。如果你的类继承自CCSprite，那么你将不得不使用两个精灵才可能

做淡进淡出的转变效果。这是为什么要用“复合”而不是“继承”的一个原因。继承会导致类之间的关系过于紧密。如果你修改了父类，这个修改可能会影响它的所有子类。

另一个原因是：游戏物体应该封装自己的视觉表现，如果它包含了自己的逻辑，那么游戏物体就只需要自己改变自己的CCNode属性就可以了，这些属性包括位置，大小，旋转或者运行和停止动作。因为一个很多游戏开发者迟早会遇到的问题是：他们的游戏物体是由外来影响直接控制的。例如，你可能让场景层，用户界面和角色自身都可以直接改变角色精灵的位置。**正确的方法是**：你应该让其他系统通知角色改变它的位置，而角色可以自己决定是否按照指示改变位置，或者通过修改指令来改变位置，或者直接忽略指令。

使用CCSprite来合成游戏物体

我们来实践一下上述的讨论。我们将生成一个基于Objective-C的基类NSObject的类，而不是继承自CCNode类。**图5-3**显示了我们使用的Cocoa Touch类模板：

图5-3. 创建一个基于NSObject的类，而不是基于CCSprite这种继承自CCNode的类。

我决定在ScenesAndLayers05项目中把蜘蛛分出来成为一个独立的类。类的名字就叫Spider。**列表5-11**展示了Spider类的头文件：

列表5-11. Spider类的头文件

```
// 继承自 NSObject 的同时不要忘记包含 cocos2d 的头文件
#import "cocos2d.h"

@interface Spider : NSObject
{
        CCSprite* spiderSprite;
}

+(id) spiderWithParentNode:(CCNode*)parentNode;
-(id) initWithParentNode:(CCNode*)parentNode;

@end
```

在生成一个基于 NSObject 的新类时，你需要手动添加 #import “cocos2d.h”头文件，否则会碰到编译错误。没有自动添加这个头文件包含的原因是你没有使用 cocos2d 的类模板来生成新类。我们使用了一个 CCSprite 作为成员变量，命名为 spiderSprite。这样的方式叫做“复合”（Composition），因为你在 Spider 类里使用了 CCSprite 来显示蜘蛛。

列表 5-12 的 Spider 类代码里，存在一个像 CCNode 类一样的静态自动释放的初始化方法。这是在模仿 cocos2d 的内存管理方式。这是个好方式，你应该使用它。你会注意到我没有使用[self scheduleUpdate]方法，因为此方法是在 CCNode 里定义的，而我们的类没有从 CCNode 继承。这里我们会使用另一个 cocos2d 没有在文档里标出来的 CCScheduler 类。使用这个类也要求我们在 dealloc 方法中手动解除预约的更新方法。

列表 5-12. Spider 类的实现代码

```
#import "Spider.h"

@implementation Spider

// 静态自动释放的初始化方法，我模仿了cocos2d的内存分配方式
+(id) spiderWithParentNode:(CCNode*)parentNode
{
        return [[[self alloc] initWithParentNode:parentNode] autorelease];
}

-(id) initWithParentNode:(CCNode*)parentNode
{
        if ((self = [super init]))
        {
                CGSize screenSize = [[CCDirector sharedDirector] winSize];

                spiderSprite = [CCSprite spriteWithFile:@"spider.png"];
                spiderSprite.position = CGPointMake(CCRANDOM_0_1() *screenSize.width,
                                                    CCRANDOM_0_1() * screenSize.height);

                [parentNode addChild:spiderSprite];

                // 使用CCSchedule类手动预约更新方法（此方法没有在文档里注明）
                [[CCScheduler sharedScheduler] scheduleUpdateForTarget:self priority:0
```

```
                                                                    paused:NO];
        }
        return self;
}

-(void) dealloc
{
        // 必须要手动解除方法预约，不会自动解除
        [[CCScheduler sharedScheduler] unscheduleUpdateForTarget:self];

        [super dealloc];
}

-(void) update:(ccTime)delta
{
        numUpdates++;
        if (numUpdates > 50)
        {
                numUpdates = 0;
                [spiderSprite stopAllActions];

                // 让蜘蛛随机移动
                CGPoint moveTo = CGPointMake(CCRANDOM_0_1() * 200 - 100,
                                                        CCRANDOM_0_1() * 100 - 50);
                CCMoveBy* move = [CCMoveBy actionWithDuration:1 position:moveTo];
                [spiderSprite runAction:move];
        }
}
@end
```

Spider类现在会使用自有的逻辑让蜘蛛在屏幕上跑来跑去了。当然，它不会赢取任何奖项，这只是一个例子而已。

到目前为止，你可能认为只有CCLayer才能接收触摸输入事件。实际上，通过直接使用CCTouchDispatcher，你可以让任何类接收触摸事件。你只需要在类里实现CCStandardTouchDelegate协议或者CCTargetedTouchDelegate协议就可以了。你可以在ScenesAndLayers06项目里找到这些修改过的代码。**列表5-13**展示了在头文件中添加的协议代码：

列表5-13. CCTargetedTouchDelegate协议

```
@interface Spider : NSObject <CCTargetedTouchDelegate>
{
        …
}
```

列表5-14突出显示了对Spider类所做的修改。现在Spider类会对定向的触摸事件做出反应了。任何时候点击一只蜘蛛，它会很快的从你的手指下跑开。和你想的不一样，其实现实中蜘蛛更怕人。

和CCScheduler预约器一样，Spider类注册的CCTouchDispatcher(触摸调度程序)也需要在dealloc方法中被移除。否则，在它们从内存中被释放以后，预约器或触摸调度程序将一直保留对Spider类的参考，从而导致之后的程序崩溃。

列表5-14. 修改过的Spider类

```
-(id) initWithParentNode:(CCNode*)parentNode
{
        if ((self = [super init]))
        {
                …
                // 通过没有在文档里标注的CCSchedule类手动预约更新方法
                [[CCScheduler sharedScheduler] scheduleUpdateForTarget:self priority:0
                                                                        paused:NO];
                // 让这个类可以接收定向的触摸事件
                [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self
                                                        priority:-1 swallowsTouches:YES];
        }

        return self;
}

-(void) dealloc
{
        //必须要手动解除预约，不会自动解除
        [[CCScheduler sharedScheduler] unscheduleUpdateForTarget:self];

        // 必须手动从触摸调度程序中移除这个类
        [[CCTouchDispatcher sharedDispatcher] removeDelegate:self];

        [super dealloc];
}

// 将经常用到的逻辑代码放进方法中
-(void) moveAway:(float)duration position:(CGPoint)moveTo
{
        [spiderSprite stopAllActions];
        CCMoveBy* move = [CCMoveBy actionWithDuration:duration position:moveTo];
        [spiderSprite runAction:move];
}

-(void) update:(ccTime)delta
{
        numUpdates++;
        if (numUpdates > 50)
        {
                numUpdates = 0;

                // 以普通速度移动
                CGPoint moveTo = CGPointMake(CCRANDOM_0_1() * 200 - 100,
                                                        CCRANDOM_0_1() * 100 - 50);
                [self moveAway:2 position:moveTo];
        }
}

-(BOOL) ccTouchBegan:(UITouch *)touch withEvent:(UIEvent *)event
{
        // 检查接收到的触摸是否落在蜘蛛精灵的大小范围之内
        CGPoint touchLocation = [MultiLayerScene locationFromTouch:touch];
        BOOL isTouchHandled = CGRectContainsPoint([spiderSprite boundingBox],
                                                                touchLocation);

        if (isTouchHandled)
        {
```

```
                // 重置移动计数器
                numUpdates = 0;

                // 从触摸到的位置快速离开
                CGPoint moveTo;
                float moveDistance = 60;
                float rand = CCRANDOM_0_1();

                // 随机选择四个方向中的一个逃跑
                if (rand < 0.25f)
                        moveTo = CGPointMake(moveDistance, moveDistance);
                else if (rand < 0.5f)
                        moveTo = CGPointMake(-moveDistance, moveDistance);
                else if (rand < 0.75f)
                        moveTo = CGPointMake(moveDistance, -moveDistance);
                else
                        moveTo = CGPointMake(-moveDistance, -moveDistance);

                // 快速移动
                [self moveAway:0.1f position:moveTo];
        }
        return isTouchHandled;
}
```

我把蜘蛛的移动逻辑代码放到一个独立的方法中。虽然最直接的方法是把现有的代码从更新方法（update method）中直接复制到ccTouchBegan方法中，但是复制粘贴是魔鬼。虽然复制粘贴很容易，但是之后当你要修改代码时，你将不得不付出更多的时间和精力去修改存在于多处地方的代码，而且你也要多测试好多地方。

列表5-14中的moveAway方法是个好例子。

ccTouchBegan通过使用CGRectContainsPoint方法来检查玩家的触摸点是否落在蜘蛛精灵的大小范围之内。如果在范围之内，触摸事件就会被处理，蜘蛛会以很快的速度向4个方向中的一个移动。

总之，虽然使用基于NSObject的类作为游戏物体看起来有些奇怪，而且它的好处也只有你在创建大一些项目的时候才会显现出来（小项目可以用基于CCSprite的类）。但是当你变得更加熟练，更加有野心时，请回来这一章，尝试着使用这里的方法。它会让你写出结构更好的代码，也会让单独的游戏元素拥有更加清晰明确的目的。

一些很酷的CCNode类

以下我会介绍四个基于CCNode可以满足特定要求的类。它们是：CCProgressTimer，CCParallaxNode，CCRibbon和CCMotionStreak。

CCProgressTimer （进度条）

在ScenesAndLayers07项目中，我在UserInterfaceLayer类里添加了一个CCProgressTimer节点。时间进度类可用于很多地方，比如加载进度条，或者用

于展示处于失效状态的按钮恢复到激活状态所需要的时间（你可以回想一下《魔兽争霸》里面的动作按钮和恢复到可以释放魔法的时间进度显示）。进度条使用精灵图形来显示进度：随着时间的过去逐渐显示出精灵的全部，来表示游戏的进度。**列表5-15**代码展示了如何初始化CCProgressTimer节点：

列表5-15. 初始化CCProgressTimer节点

```
//进度条使用精灵图形来显示进度：随着时间的过去逐渐显示出精灵的全部，来表示游戏的进度
CCProgressTimer* timer = [CCProgressTimer progressWithFile:@"firething.png"];
timer.type = kCCProgressTimerTypeRadialCCW;
timer.percentage = 0;
[self addChild:timer z:1 tag:UILayerTagProgressTimer];

// 进度条需要预约的更新方法来更新自身的状态
[self scheduleUpdate];
```

timer类型来自在CCProgressTimer.h里定义的CCProgressTimerType这个枚举类型。你可以选择圆形，纵向和横向的进度显示。不过进度条有一个缺点：它不会自己更新自己。你必须经常更新进度条的百分比数值来显示进度。这是我在**列表5-15**的代码中包含了scheduleUpdate这个更新方法的原因。**列表5-16**展示了这个更新方法的具体实现代码。这里的进度是用当前已经经过的时间来表示的。

列表5-16. 进度时间的更新方法的具体实现

```
-(void) update:(ccTime)delta
{
        CCNode* node = [self getChildByTag:UILayerTagProgressTimer];
        NSAssert([node isKindOfClass:[CCProgressTimer class]], @"node is not a
        CCProgressTimer");

        // 更新进度条的时间显示
        CCProgressTimer* timer = (CCProgressTimer*)node;
        timer.percentage += delta * 10;
        if (timer.percentage >= 100)
        {
                timer.percentage = 0;
        }
}
```

CCParallaxNode （视差视图）

“视差”（Parallaxing）是2D游戏中通过让不同层上的图片用不同的速度移动，来创造视觉深度的方法。前景的图片移动的比背景图片要快。cocos2d有一个特殊的节点用于实现这个效果。**列表5-17**代码里使用了CCParallaxNode，你也可以在ScenesAndLayers08项目中找到这些代码：

列表5-17. 虽然CCParallaxNode需要很多设置的工作，但是最终得到的效果是值得的

```
// 从背景到前景，依次为每一个parallax层加载精灵
CCSprite* para1 = [CCSprite spriteWithFile:@"parallax1.png"];
CCSprite* para2 = [CCSprite spriteWithFile:@"parallax2.png"];
CCSprite* para3 = [CCSprite spriteWithFile:@"parallax3.png"];
CCSprite* para4 = [CCSprite spriteWithFile:@"parallax4.png"];

//根据屏幕和图片大小来设置正确的偏移
```

```
para1.anchorPoint = CGPointMake(0, 1);
para2.anchorPoint = CGPointMake(0, 1);
para3.anchorPoint = CGPointMake(0, 0.6f);
para4.anchorPoint = CGPointMake(0, 0);
CGPoint topOffset = CGPointMake(0, screenSize.height);
CGPoint midOffset = CGPointMake(0, screenSize.height / 2);
CGPoint downOffset = CGPointZero;

// 生成一个parallax节点用于储存各种精灵
CCParallaxNode* paraNode = [CCParallaxNode node];
[paraNode addChild:para1 z:1 parallaxRatio:CGPointMake(0.5f, 0) positionOffset:topOffset];
[paraNode addChild:para2 z:2 parallaxRatio:CGPointMake(1, 0) positionOffset:topOffset];
[paraNode addChild:para3 z:4 parallaxRatio:CGPointMake(2, 0) positionOffset:midOffset];
[paraNode addChild:para4 z:3 parallaxRatio:CGPointMake(3, 0) positionOffset:downOffset];
[self addChild:paraNode z:0 tag:ParallaxSceneTagParallaxNode];

// 移动parallax节点以展示“视差”效果
CCMoveBy* move1 = [CCMoveBy actionWithDuration:5 position:CGPointMake(-160, 0)];
CCMoveBy* move2 = [CCMoveBy actionWithDuration:15 position:CGPointMake(160, 0)];
CCSequence* sequence = [CCSequence actions:move1, move2, nil];
CCRepeatForever* repeat = [CCRepeatForever actionWithAction:sequence];
[paraNode runAction:repeat];
```

要生成一个CCParallaxNode，首先需要生成用于创建独立视差图片的CCSprite精灵节点 ，然后你需要把它们合理地放置在屏幕上。在我的样例中，我选择了修改这些节点的定位点，从而易于将精灵与屏幕边缘对齐。生成CCParallaxNode和生成其它节点的方式是一样的，但是在添加子节点的时候我们需要使用一个特殊的初始化方法。通过这个初始化方法，你可以传递parallaRatio参数。这个参数是一个CGPoint类型，用于倍增 CCParallaxNode的移动。在我们的例子里，CCSprite的para1精灵会用一半的通常速度移动，para2用普通的速度移动，part3会以高于CCParallaxNode移动速度一倍的速度运行，以此类推。

用一系列的CCMoveBy动作，CCParallax节点可以从左移到右，然后回来。你可以看到：背景中的云移动最慢，同时前景里的树和石头移动最快。这样视觉上的深度就被创造出来了。

注：一旦子节点被加入 CCParallaxNode中，你就不能再修改它们的位置了。你只能够在背景的范围内移动，否则背景色就会被显示出来。你可以通过修改CCMoveBy动作让节点移动到很远的地方，看到背景覆盖不到的背景色。你可以通过添加相同的精灵作为背景来增加背景的覆盖区域。但是如果你想拥有无限横向或者纵向滚屏的话，你就需要自己写个“视差”系统（parallax system）了。实际上，这也是我们在第七章里需要讨论的。

CCRibbon（图片链条）

CCRibbon节点会生成一系列图片，就像一根链条，或者如图5-4所示，像一只千足虫，爬满ScenesAndLayer09项目中的场景。

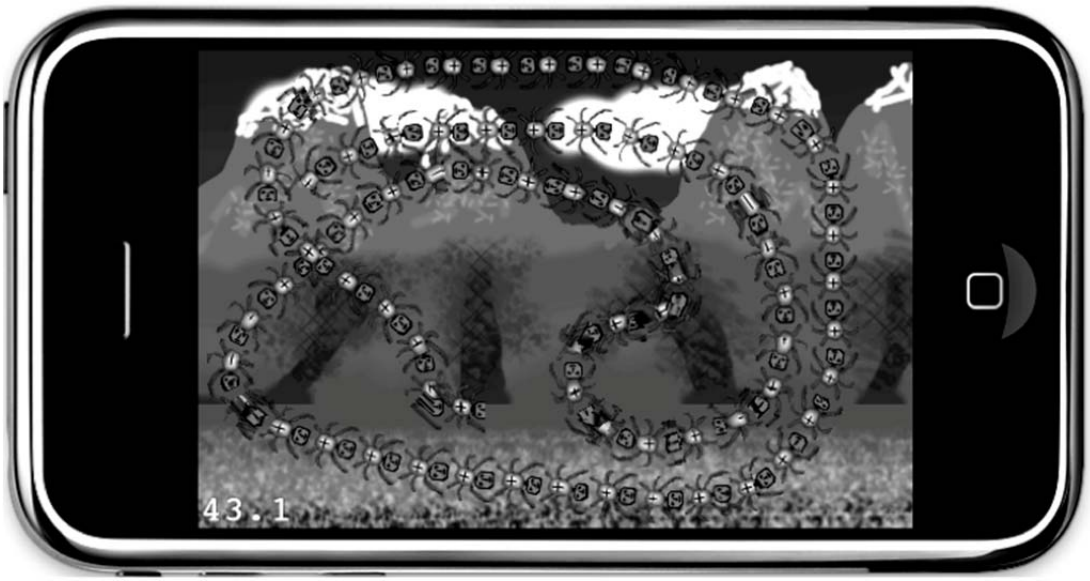

图5-4. ScenesAndLayers09项目中的CCRibbon蜘蛛，看上去就像在屏幕上爬行的千足虫。你可以使用CCRibbon节点通过触摸屏幕画出一连串蜘蛛来。

触摸输入配合CCRibbon类，可以创造出当下很流行的划线游戏（Line-Drawing Game）。**列表5-18**展示了如何在CCRibbon里实现触摸事件处理。需要注意的是：你无法移除CCRibbon上的单个点，你只能将整个CCRibbon删除。CCRibbon初始化方法的宽和长两个参数用于决定它的单个元素大小。在我们的例子里，我选择了32x32像素，这和spider.png图片的大小一样。如果你选择了其它的值，系统将会相应地调整蜘蛛图片的大小。

列表5-18. CCRibbon类

```
-(void) resetRibbon
{
        // 移除现有的CCRibbon，然后生成一个新的CCRibbon节点
        [self removeChildByTag:ParallaxSceneTagRibbon cleanup:YES];
        CCRibbon* ribbon = [CCRibbon ribbonWithWidth:32 image:@"spider.png"
                                                length:32 color:ccc4(255, 255, 255, 255)
                                        fade:0.5f];
        [self addChild:ribbon z:5 tag:ParallaxSceneTagRibbon];
}

-(CCRibbon*) getRibbon
{
        CCNode* node = [self getChildByTag:ParallaxSceneTagRibbon];
        NSAssert([node isKindOfClass:[CCRibbon class]], @"node is not a CCRibbon");

        return (CCRibbon*)node;
}

-(void) addRibbonPoint:(CGPoint)point
{
        CCRibbon* ribbon = [self getRibbon];
        [ribbon addPointAt:point width:32];
}

-(BOOL) ccTouchBegan:(UITouch*)touch withEvent:(UIEvent *)event
{
        [self addRibbonPoint:[MultiLayerScene locationFromTouch:touch]];
```

```
        return YES;
}

-(void) ccTouchMoved:(UITouch*)touch withEvent:(UIEvent *)event
{
        [self addRibbonPoint:[MultiLayerScene locationFromTouch:touch]];
}

-(void) ccTouchEnded:(UITouch*)touch withEvent:(UIEvent *)event
{
        [self resetRibbon];
}
```

CCMotionStreak（拖尾效果）

CCMotionStreak实际上是CCRibbon的一个封装实现。它会让CCRibbon的元素在你画出它们之后慢慢淡出直至消失。你可以试一下ScenesAndLayers10项目。

如**列表5-19**所示，它的用法和CCRibbon几乎完全一样。唯一的区别是CCRibbon现在成了CCMotionStreak的一个属性。Fade参数用于决定元素消失的速度；它的值越小，元素消失的越快。修改minSeg这个参数好像不会带来什么直接的可视效果，但是如果你把它设为一个负值的话，会导致一个有趣的图形出错。

列表5-19. CCMotionStreak会让CCRibbon的元素慢慢消失，从而创造出拖尾效果

```
-(void) resetMotionStreak
{
        // 删除当前的CCMotionStreak，生成一个新的
        [self removeChildByTag:ParallaxSceneTagRibbon cleanup:YES];
        CCMotionStreak* streak = [CCMotionStreak streakWithFade:0.7f minSeg:10
                image:@"spider.png" width:32 length:32 color:ccc4(255, 0, 255, 255)];

        [self addChild:streak z:5 tag:ParallaxSceneTagRibbon];
}

-(void) addMotionStreakPoint:(CGPoint)point
{
        CCMotionStreak* streak = [self getMotionStreak];
        [streak.ribbon addPointAt:point width:32];
}
```

结语

本章我们学习了更多关于场景和层使用相关的知识。我也解释了为什么通常不应该让游戏物体直接继承自CCSprite，并且展示了如何让游戏物体继承自NSObject，而不是任何继承自CCNode的类（CCSprite是继承自CCNode的类）。

最后，我介绍了如何使用四个特殊的CCNode类：CCProgressTimer，CCParallaxNode，CCRibbon和CCMotionStreak。

你现在已经有足够的cocos2d知识来创造更复杂的游戏了。接下去我们将会制作一个“横向滚屏射击游戏”。随着游戏复杂程度的提高，我们将会需要更复杂

的图形，包括动画。下一章我们将会讨论如何让这些精灵更有效地使用内存，同时运行得更高效。