# 第3章 Cocos2D核心类

本章主要分析第2章代码示例，同时向大家展示构成Cocos2D的核心类如何组织在一起，并构成一个完整的游戏。几乎任何一款Cocos2D游戏中都会用到这些类，所以了解它们的作用、属性和方法，以及如何使用这些类，有助于我们开发出更好的游戏。有了这些基本知识后，会发现Cocos2D并没有想象那么难；而且即使只用这些类，也可以创造出很有意思的游戏。

掌握这些游戏构建的关键类和关键概念之后，在开始一个新的游戏项目时会更加得心应手。

## Cocos2D节点基础知识

在Cocos2D中，CCNode、CCScene和CCLayer这些类是没有视觉呈现的，但它们之间存在一个层级关系。

### 节点层级图

节点层级图是由所有目前活跃的Cocos2D节点所组成的一个层级图，有时又叫场景继承关系图。除了场景本身，每一个节点只有一个父节点，但是可以有任意数量的子节点。

将节点添加到其他节点中时，就在构建节点场景图。图3-1描绘了一个虚拟的游戏场景图。在顶层是一个场景（CCScene）节点，接下来是层（CCLayer）节点。

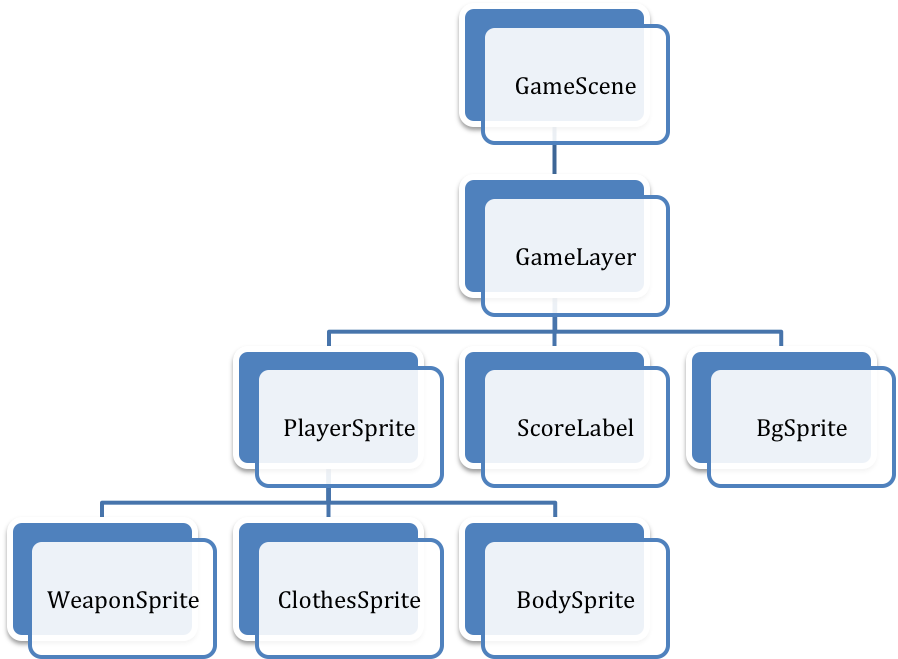


图3-1 Cocos2D节点层级图

CCLayer下一层的子节点构成了游戏的主要元素，它们大多数是精灵（CCSprite）节点，还包括用于显示游戏得分的标签节点，游戏中的菜单节点和菜单项目节点等。

节点层级图的作用在于，对某个节点施加的影响将会影响到它的所有子节点。

### 什么是锚点

如果觉得锚点这个概念很抽象，可以想象在墙上用图钉订上一张卡片，卡片就相当于节点，墙壁就相当于设备的屏幕，而图钉就是锚点。

在Cocos2D中，锚点的默认位置是在纹理图像的几何中心位置。比如将某个精灵对象的位置设置为（50，50）时，其几何中心位置在默认情况下也应该是（50，50）。

如果将锚点设置在精灵的左下角，并将精灵的位置设置为（50，50）。那么此时应该是精灵的左下角处于（50，50）的位置，而非其几何中心处在该位置。锚点和节点对象的几何形状关系如图3-2所示。

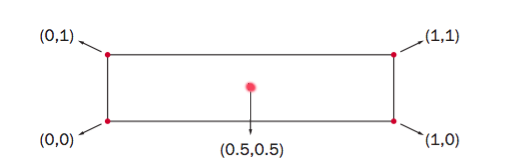


图3-2 锚点和节点对象的几何形状关系

锚点实际上是到节点对象纹理图的偏移。锚点值的范围从（0，0）到（1，1）之间。默认情况下，锚点位于纹理图像的几何中心，即（0.5，0.5）。

锚点一个很重要的作用就是辅助游戏元素进行界面布局定位。如代码清单3-1中所示的第2章用到的代码片断所示。

1. 第2章游戏中用到的代码片断

CCLabelTTF \*lifeIndicator = [CCLabelTTF labelWithString:@"生命值:" fontName:@"Arial" fontSize:20];

lifeIndicator.anchorPoint = ccp(0.0,0.5);

lifeIndicator.position = ccp(20,winSize.height - 20);

[self addChild:lifeIndicator z:10];

\_lifeLabel = [CCLabelTTF labelWithString:@"3" fontName:@"Arial" fontSize:20];

\_lifeLabel.position = ccpAdd(lifeIndicator.position, ccp(lifeIndicator.contentSize.width+10,0));

[self addChild:\_lifeLabel z:10];

提示 通常在设置CCNode的position属性时是相对于锚点的。而CCLabel的锚点默认设置在节点的中点，所以，当设置CCLabel节点位置时，节点的中心位置也是所设置的坐标位置。

## CCNode节点类

CCNode是Cocos2D中最重要的类，同时也是所有节点的基类。它是一个抽象类，没有视觉表现，定义了所有节点都通用的属性和方法。

在Cocos2D中，所有要绘制到屏幕的对象，或是自身包含要绘制到屏幕中的对象，都属于CCNode类。最重要的几个CCNode类分别是CCScene、CCLayer、CCSprite、CCMenu，这也是本章要重点学习的内容。

CCNode的主要作用：

* 包含其他的CCNode节点（addChild、getChildByTag、removeChild等方法）
* 通过定时器预定消息（schedule、unschedule等方法）
* 执行动作（runAction、stopAction等方法）

**注意**　作为抽象类，CCNode是没有自身纹理的。

### CCNode类的属性

CCNode类的常见属性包括：

1. anchorPoint

锚点。节点的变形和定位操作都要依据锚点来确定。锚点的变量类型是CGPoint。

锚点值通常在0～1之间，（0，0）代表左下角，而（1，1）代表右上角。默认CCNode节点锚点值是（0，0），也就是节点左下角。另外，还有anchorPointInPoints属性，区别在于其采用绝对像素值为单位。

1. camera

设置游戏中的视角。利用OpenGL ES中的gluLookAt()。变量类型是CCCamera。

仅当创建3D效果时才会用到camera属性。如果使用了该属性，就不能再同时使用rotation/scale/position这些属性。

提示 Cocos2D不推荐直接使用camera。如果需要实现一个视差滚动的背景（人物往前移动时，游戏场景往身后移动，并且远近不同的背景移动速度还不一样，近的物体移动速度快，远的物体移动速度慢，即所谓的视差），通过移动camera来实现会变得更复杂，而且会使Cocos2D本身的一些功能特性遭受破坏。推荐的作法是使用CCParallaxNode，后面章节会向大家演示如何做。

1. children

节点的子节点数组。变量类型是CCArray。

CCArray是专门为了性能优化而设计的类，它的功能和NSMutablArray差不多。推荐在Cocos2D中尽量使用CCArray替代NSMutableArray.

1. contentSize

未经转换的节点大小，以points（点值）为单位。contentSize的变量类型是CGSize，默认值为（0,0）。

无论节点如何缩放或旋转，contentSize属性值始终不变。所有的节点都有大小，而Layer（层）和Screen（场景）大小就是屏幕大小。

contentSize主要用于碰撞检测，此时可以根据节点当前所在位置以及节点纹理本身的大小来计算碰撞矩形区域。具体代码如代码清单3-2所示。

1. 计算碰撞矩形区域

-(CGRect) rectOfSprite:(CCSprite\*)sprite{

return CGRectMake(sprite.position.x - sprite.contentSize.width / 2,

sprite.position.y - sprite.contentSize.height /2,

sprite.contentSize.width, sprite.contentSize.height);

}

1. glServerState：

OpenGL服务器端状态，从v2.0版本开始使用。变量类型是ccGLServerState，可参考ccGLState.h。

1. grid

在对节点应用某些特殊效果时会用到此属性。变量类型是CCGridBase。

1. isRelativeAnchorPoint

属性为YES时，节点的变形将基于锚点进行。变量类型是BOOL。

对于精灵、标签和其他可调整大小的节点会默认启用该属性。而对于场景、层和其他“全屏”对象，会默认禁用该属性。

1. isRunning

判断当前节点是否在运行。变量类型是BOOL。

1. orderOfArrival

当使用同样的z值对子节点进行排序的时候，用来决定子节点的顺序，不要手动更改该数值。变量类型是NSUInteger。

1. parent

对节点父节点的弱引用，通常很少用到。变量类型是CCNode。

1. position

节点在屏幕中的位置，变量类型是CGPoint，以points(点值）为单位，屏幕的左下角坐标是（0，0）。

此外，还有positionInPixels属性，同样是节点在屏幕中的位置，不过单位为pixels（像素值），变量类型是CGPoint。默认值为（0,0）。

1. rotation

节点沿顺时针方向旋转的角度，变量类型是float。默认值为0。

1. scale

节点的缩放比例，变量类型是float。有三属性，默认值为（scaleX=1,scaleY=1）：

* scale：节点沿x和y坐标轴的整体缩放比例。
* scaleX：节点沿x轴的缩放比例。
* scaleY：节点沿y轴的缩放比例。

1. shaderProgram

着色程序。变量类型是GLProgram。

从v2.0版本开始使用，详细信息可参考GLProgram.h。后面会专门用一章介绍如何为CCNode编写定制的shader程序。

1. skew

节点的变形角度，变量类型是float。

* skewX：沿x轴方向顺时针切向畸变角度（Y轴和节点形状左边缘之间的角度），默认值为0。
* skewY：沿y轴方向逆时针切向畸变角度（x轴和节点形状底边的角度），默认值为0。

1. tag

识别节点的标识值。变量类型是NSInteger。

无论是创建、引用还是删除节点，tag值都有用武之地。比如，tag参数允许通过getChildByTag方法获取指定节点。如果有多个节点拥有相同tag数值，getChildByTag将把找到的第一个节点返回，其他节点将不能够再被访问。所以要确保为节点指定独有的tag数值。

tag在很多应用场景以及在Cocos2D游戏编程过程中都起到重要作用。最简单的应用就是在其他scheduler的selector函数里通过getChildByTag得到对应tag节点。这样无需在类的定义里声明过多的节点弱引用（weak ref）。此外，tag还有一个重要的应用场景。多个CCMenuItem共用一个selector时，可以给menuItem赋予不同tag，然后在selector里根据tag值判断玩家具体触碰的是哪一个按钮。

1. userData

用户自定义的数据指针对象。变量类型是void。

userData经常用于selector之间的数据传递，或者其他需要节点携带传递数据的场合。

1. visible

判断节点是否可见，默认值为YES。变量类型是BOOL。

1. zOrder

节点相对于其“兄弟”节点（拥有共同的父节点）Z顺序值。变量类型是NSInteger。

对于如CCSprite这样有视觉呈现的节点，该参数决定了节点的绘制顺序。拥有最小Z值的节点会首先被绘制；拥有最大Z值的节点最后被绘制。如果多个节点拥有相同Z值，绘制顺序将由它们的添加顺序来决定。

1. vertexZ

真正的OpenGL Z顶点。OpenGL Z顶点和Cocos2D Z 顺序的区别如下：

* OpenGL Z 修改Z 顶点，而非父节点和子节点间的Z 顺序；
* OpenGL Z可能需要设置2D投影；
* 如果所有节点使用相同的OpenGL Z顶点，Cocos2D Z顺序仍然保持正常。

使用该值可能会破坏Cocos2D中的父节点-子节点Z顺序，通常较少使用。

以上只是对CCNode类属性的简单介绍，要真正了解每个属性在开发Cocos2D游戏中的作用，需要在实际开发中逐渐体会。在开发过程中可以随时回顾本章查找相关属性。第2章的示例代码运用一部分CCNode属性，可以多多研习并修改代码，体会这些属性的用途。

### CCNode类的方法

CCNode类中所实现的方法可以分为三个大类，分别是对子节点的处理、使用定时器预定消息，以及执行动作。此外一些其他方法不属于以上三大类，下面将一一介绍。

#### 处理子节点

CCNode类实现了所有添加、获取和删除子节点的方法。以下是一些处理子节点的方法：

1. 创建一个新的节点。示例代码：

CCNode\* childNode = [CCNode node];

请注意，这行代码等同于：

CCNode\* childNode = [[[CCNode alloc]init]autorelease];

1. 将新节点添加为当前节点的子节点，并设置子节点的z值和tag值。

如果该子节点被添加到“运行”模式，则将立即调用“onEnter”和“onEnterTransitionDidFinish”方法。示例代码：

[thisNode addChild:childNode z:0 tag:1];

1. 使用节点的标识获取子节点。示例代码：

CCNode\* childNode = [thisNode getChildByTag:1];

1. 通过tag删除子节点。

如果cleanup的参数值为YES，则会停止任何运行中的动作。示例代码：

[thisNode removeChildByTag:1 cleanup:YES];

1. 通过节点指针删除子节点。

如果cleanup的参数值为YES，则会停止任何运行中的动作。示例代码：

[thisNode removeChild:childNode cleanup:YES];

1. 删除一个节点的所有子节点。

如果cleanup的参数值为YES，则会停止任何运行中的动作。示例代码：

[thisNode removeAllChildrenWithCleanup:YES];

1. 从当前节点的父节点删除当前节点。

如果cleanup的参数值为YES，将删除所有的动作和回调方法；如果当前节点没有父节点，则不会执行任何操作。示例代码：

[thisNode removeFromParentAndCleanup:YES];

#### 执行动作

CCNode可以使用动作（Actions）让节点执行某些动作。本书第4章将专门讲述动作的详细内容，现在只需要知道使用动作可以让节点移动、旋转和缩放，以及一些其他事情。

1. 运行某个特定的动作。示例代码：

[thisNode runAction:action];

其中action是使用CCAction定义的某个动作。

1. 停止所有在该节点上运行的动作。示例代码：

[thisNode stopAllActions]；

1. 停止在该节点上运行的某个特定动作。示例代码：

[thisNode stopAction:action];

其中action是使用CCAction定义的某个动作。

1. 停止运行的动作列表中的某个特定动作，使用tag标识来区分。示例代码：

[thisNode stopActionByTag:123];

1. 获取当前运行的动作清单中的某个特定动作，使用tag标识来区分。示例代码：

myAction = [thisNode getActionByTag:123];

如果想在以后使用此动作，可以使用这个方法。

1. 获取运行动作的数量。

包括正在运行的动作，再加上预定运行的动作（在actionsToAdd和动作数组中的动作），复合动作算一个动作。示例代码：

numberofActions = [thisNode numberOfRunningActions];

#### 预定消息

CCNode节点可以预定消息，其实就是每隔一段时间调用一次方法。在很多游戏中需要节点调用特定的更新方法来处理某些情况，比如碰撞测试等。

1. 每帧都调用的更新方法，每个节点只可预定一个update方法。示例代码：

-(void) scheduleUpdates

{

[self scheduleUpdate];

}

-(void) update:(ccTime)delta

{

// 在游戏的每一帧都会被调用该方法

}

其中，delta参数表示该方法的上一次调用到现在所经过的游戏时间。

2）仍然是每帧都会调用的更新方法，每个节点只可预定一个update方法，与上面的区别在于使用该方法可以安排更新方法的优先次序。更新方法的调用次序是优先级数值从小到大的。示例代码：

// 在节点1中   
-(void) scheduleUpdates

{ [self scheduleUpdate];

}

// 在节点2中   
-(void) scheduleUpdates   
{

[self scheduleUpdateWithPriority:1];   
}

// 在节点3中   
-(void) scheduleUpdates   
{

[self scheduleUpdateWithPriority:-1];   
}

-(void) update:(ccTime)delta

{

// 在游戏的每一帧都会被调用该方法

}

所有节点（节点1、2、3）都会调用-(void)update: (ccTime) delta方法。但因为使用了优先级设置，节点3的更新方法将会被首先运行。然后是调用节点1的更新方法，因为默认优先级设定为0。节点2的更新方法最后调用，因为优先级数值最大。

如果每帧都调用相同的更新方法,上述做法很适用。不过有时需要用到更灵活的更新方法。

3）指定运行特定的更新方法，并设置调用的时间间隔。示例代码：

-(void) scheduleUpdates{

[self schedule:@selector(updateTenTimesPerSecond:) interval:0.1f];

}

-(void) updateTenTimesPerSecond:(ccTime)delta

{

//根据时间间隔来调用该方法,每秒10次

}

如果需要在每一帧都触发某个方法，应该使用scheduleUpdate方法，并把要定时处理的逻辑放到update方法中。但是update方法只有一个，所以，大部分情况下读者会使用上述代码来触发不同的定时更新逻辑。

4）指定运行特定的更新方法，并设置调用的延迟时间，但只运行一次。示例代码：

-(void) scheduleUpdates{

[self schedule:@selector(specialMethod:) delay:2.0f];

}

5）停止节点的某个指定选择器，但该方法不会停止scheduleUpdate中设置的预定更新方法。想要停止scheduleUpdate中设置的预定更新方法，可以使用unscheduleUpdate方法。示例代码：

[self unschedule:@selector(updateTenTimesPerSecond:)];

6）停止节点的所有选择器（selector），包括在scheduleUpdate里设置的update选择器，该方法不会影响节点的动作（action）。示例代码：

[self unscheduleAllSelectors];

7）用\_cmd关键词停止当前方法的预定。示例代码：

-(void) scheduleUpdates

{

[self schedule:@selector(tenMinutesElapsed:) interval:600];

}

-(void) tenMinutesElapsed:(ccTime)delta

{

// 用\_cmd关键词停止当前方法的预定  
[self unschedule:\_cmd];

}

#### 其他方法

1. -(CGRect)boundingBox

获取节点的边框，返回的坐标是相对于其父节点的。

以points（点值）为单位，返回CGRect变量类型的值。boundingBox在进行游戏物体碰撞检测时非常有用，功能类似之前定义的rectOfSprite方法。比如，可以把第2章示例游戏中的碰撞检测部分更新为代码清单3-3。

1. 碰撞检测代码更新

-(void) collisionDetection:(ccTime)dt{

CCSprite \*enemy;

CCARRAY\_FOREACH(\_enemySprites, enemy)

{

if (enemy.visible) {

//1.bullet & enemy collision detection

if (\_bulletSprite.visible && CGRectIntersectsRect(enemy.boundingBox, \_bulletSprite.boundingBox)) {

//此处省略了部分代码

//2.enemy & player collision detection

CCSprite \*playerSprite = (CCSprite\*)[self getChildByTag:kTagPalyer];

if (playerSprite.visible &&

playerSprite.numberOfRunningActions == 0

&& CGRectIntersectsRect(enemy.boundingBox, playerSprite.boundingBox)) {

//此处省略了部分代码

}

}

}

}

1. -(void)cleanup

停止所有运行中的动作和预定方法。

1. -(CGPoint)convertToNodeSpace:(CGPoint)worldPoint

将点坐标转换成节点空间坐标。以points（点值）为单位，返回CGPoint变量类型的值。

1. -(CGPoint)convertToNodeSpaceAR:(CGPoint)worldPoint

将点坐标转换成相对于锚点的节点空间坐标。以points（点值）为单位，返回CGPoint变量类型的值。

1. -(CGPoint)convertToWorldSpace:(CGPoint)nodePoint

将节点空间坐标转换成世界空间坐标。以points（点值）为单位，返回CGPoint变量类型的值。

1. -(CGPoint)convertToWorldSpace:(CGPoint)nodePoint

将节点空间坐标转换成相对于锚点的世界空间坐标。以points（点值）为单位，返回CGPoint变量类型的值。

注意，第2章的示例游戏中，假如发射子弹的需求有所改变，比如只有玩家“点中”飞机才开始发射子弹，实现方式如代码清单3-4所示。

1. 发射子弹实现

-(void) ccTouchesEnded:(NSSet \*)touches withEvent:(UIEvent \*)event{

//修改成，必须点选中playerSprite才能够发射子弹

//方法1：因为boundingBox是相对于世界坐标系而言的，所以要用self convertTouchToNodeSpace 转化成世界坐标系中的坐标

UITouch \*touch = [touches anyObject];

CCSprite \*playerSprite = (CCSprite\*)[self getChildByTag:kTagPalyer];

CGPoint pt;

// pt = [touch locationInView:[touch view]];

// pt = [[CCDirector sharedDirector] convertToGL:pt];

// pt = [self convertToNodeSpace:pt];

//上面三句调用，可以简化为下面一句调用

//CGPoint pt = [self convertTouchToNodeSpace:touch];

// if (CGRectContainsPoint(playerSprite.boundingBox, pt)) {

// \_isTouchToShoot = YES;

// }

//===================================================================

//方法2：

// pt = [touch locationInView:[touch view]];

// pt = [[CCDirector sharedDirector] convertToGL:pt];

// pt = [playerSprite convertToNodeSpace:pt];

//简化为下面的一句代码调用

pt = [playerSprite convertTouchToNodeSpace:touch];

CCLOG(@"pt.x = %f, pt.y = %f",pt.x, pt.y);

if (CGRectContainsPoint(playerSprite.textureRect, pt)) {

\_isTouchToShoot = YES;

CCLOG(@"touched!");

}

}

可以找到本章示例代码，根据代码注释体验4种检测玩家的触摸点是否在playerSprite区域内的方法。相信会对局部坐标和世界坐标有一定的了解，同时明白boundingBox是相当于节点所处的世界来计算。通过几个小实验，应该已经掌握boundingBox和textureRect的功能与区别。

1. -(void)draw

覆盖该方法以绘制自己的节点。在Xcode中打开HelloWorldLayer.m，实现下面的draw方法如代码清单3-5所示。

1. draw方法实现

-(void) draw{

[super draw];

ccDrawColor4F(255,0,0,0);

glLineWidth(8);

ccDrawLine(ccp(10,10),ccp(200,200));

}

此时编译运行，并不能看到这条从（10，10）到（200，200）、红色、宽度为8像素的线。因为在程序中添加了背景精灵图片，因此需要把添加背景图片的代码注释掉：

//2.add background

// CCSprite \*bgSprite = [CCSprite spriteWithFile:@"background.png"];

// bgSprite.position = ccp(winSize.width / 2,winSize.height/2);

// [self addChild:bgSprite z:-100];

编译并运行代码，将会得到图3-2。

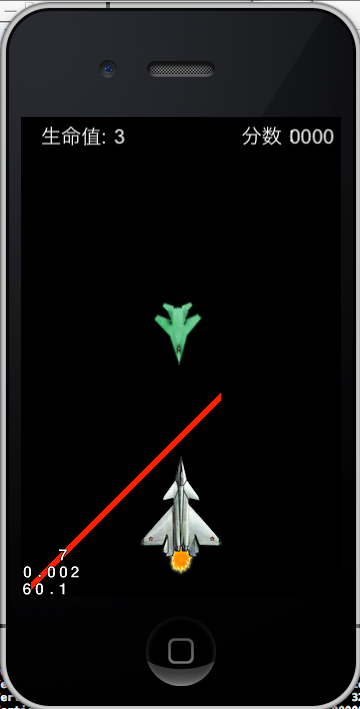


图3-2 实现draw方法绘制的节点

观察仔细会发现， 在HelloWorldLayer里添加的任何节点，在经过这条红线时都会覆盖它。因为Cocos2D的绘制是先调用节点自身的draw，然后再调用它所有子节点的draw方法。

1. -(id)init

初始化节点。

有一点需要注意，尽量把游戏场景中用到的sprite、label、menu、action、animation等对象在init方法中创建好。如果不想让玩家在进入场景时就看到，可以把些节点的visible属性先设置为NO。当后面需要使用这些预分配的对象时，就可以直接把visbile设置为yes。这样做能提高游戏性能。这种预先分配对象的策略，也叫做lazy loading。第2章示例中的玩家sprite和敌机sprite，以及子弹sprite都采用了这种方式，这种做法有助于提高游戏性能。

1. - (CGAffineTransform) nodeToParentTransform

返回一个矩阵，可以将节点（本地）空间坐标转换为父节点的空间坐标，该矩阵以像素值为单位。

1. - (CGAffineTransform) nodeToWorldTransform

返回一个世界坐标的仿射转换矩阵，该矩阵以像素值为单位。

1. -(void)onEnter

回调方法。当CCNode节点进入“舞台”时调用。如果进入时带有过渡效果，则在过渡开始时调用。

1. -(void)onEnterTransitionDidFinish

回调方法。当CCNode节点进入舞台时调用。如果带有过渡效果，则会在过渡完成时调用。

1. -(void)onExit

回调方法。当CCNode节点离开舞台时会被调用。如果带有过渡效果，则会在过渡完成时调用。

1. - (void) parentToNodeTransform

返回一个矩阵，可以将父节点的空间坐标转换为节点（本地）空间坐标。该矩阵以像素值为单位。

1. - (void) pauseSchedulerAndActions

暂停所有预定的选择器方法和动作。由onExit方法在其内部调用。

1. - (void) reorderChild: (CCNode \*) child z: (NSInteger) zOrder

重新设置子节点的次序，该子节点必须已经添加。

1. - (void) resumeSchedulerAndActions

继续所有预定的选择器方法和动作。该方法由onEnter方法在其内部调用。

1. - (void) sortAllChildren

使用该方法可以改善游戏性能。在每次绘制前对子节点数组进行排序，但除非某个所添加的子节点需要在同一帧被清除，不要手动调用该方法。

1. - (void) transform

根据节点的位置、比例、旋转和其他属性来执行OpenGL视图—矩阵转换。

1. - (void) transformAncestors

从其祖先节点执行OpenGL视图-矩阵转换。

1. -(void)visit

递归方法，访问节点的子节点并绘制它们。

1. - (CGAffineTransform) worldToNodeTransform

返回逆向的世界坐标仿射转换矩阵，以像素值为单位。

以上就是CCNode类的主要方法，详细定义可以查看Cocos2D模板中的CCNode.h文件。

## CCScene场景类

CCScene是CCNode的子类，和CCNode一样，也是一个抽象类。事实上，CCScene和CCNode几乎完全相同，唯一差异就在于CCScene的锚点位置默认在屏幕中心。如果打开模板中CCScene.h文件，会发现其中只说明CCScene继承自CCNode，并没有任何其他内容。图3-3是典型的游戏场景跳转示意图。

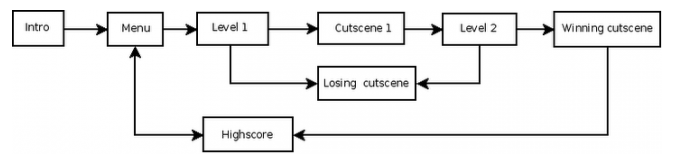


图3-3 游戏场景跳转图

当前版本的Cocos2D-iPhone中，CCScene类基本上没有附加任何逻辑机制和特殊功能。通常建议将CCScene作为所有其他节点的父节点，看作CCLayer（层）对象的一个容器。

需要说明的是，通常CCScene的子节点都直接继承自CCLayer。CCLayer包含各个游戏对象。因为大多数情况下场景对象本身不包含任何游戏相关的代码，而且很少被子类化，所以它一般在CCLayer对象里通过+(id)scene静态方法来创建。

示例代码：

+(id) scene{

CCScene \*scene = [CCScene node];

CCLayer\* layer = [HelloWorld node];

[scene addChild:layer];

return scene;

}

此外，如果要实现各种动画效果的场景切换，必须基于CCScene的场景切换。

### CCTransitionScene场景切换

CCTransitionScene继承自CCScene，而所有场景过渡效果的类都继承自CCTransitionScene。CCTransitionScene只有几个基本方法，如下：

1. -(void)finish

在过渡效果结束时调用。

1. -(void)hideOutShowIn

部分过渡效果会使用该方法来隐藏更外面的场景。

1. -(id)initWithDuration:(ccTime)t scene:（CCScene \*)s

初始化一个场景过渡效果，并指定过渡时间和即将过渡的场景。

1. +(id)transitionWithDuration:(ccTime)t scene:(CCScene \*)s

创建一个基本的场景过渡效果，并指定过渡时间和即将过渡的场景。

虽然过渡效果的名称和需要的参数数量很多，但是添加场景间的过渡效果只需要增加一行代码而已。

以一个最简单的淡入淡出过渡效果为例，场景在1秒内过渡到白色。这里把第2章示例游戏中用到的onRestartGame方法更改更改成如代码清单3-6所示。

1. onRestartGame更改

-(void) onRestartGame{

// 初始化一个过渡场景

CCTransitionFade\* transitionScene = [CCTransitionFade transitionWithDuration:1 scene:[HelloWorldLayer scene]

withColor:ccWHITE];

// 使用过渡场景对象替代HelloWorld场景

[[CCDirector sharedDirector] replaceScene:transitionScene];

}

提示 CCTransitionScene可以结合replaceScene和pushScene使用，但不能将过渡效果和popScene一起使用。

### Cocos2D支持的场景过渡效果

Cocos2D支持多种场景过渡效果，以下是目前可以使用的过渡效果及其描述。

1. CCTransitionFade

场景在指定的时间内淡入淡出到一个指定的颜色。

1. CCTransitionFadeTR

从屏幕的左下角向上卷起瓦片(tiles)，从而显示场景。该过渡效果还有三个变化：

* CCTransitionFadeBL：从屏幕的右上角到左下角淡出场景的瓦片。
* CCTransitionFadeUp：从屏幕的底部到顶部淡出场景的瓦片。
* CCTransitionFadeDown：从屏幕的顶部到底部淡出场景的瓦片。

1. CCTransitionJumpZoom

现有场景会跳动着变小，新场景跳动着变大。

1. CCTransitionMoveInL

旧的场景会移出，而新场景会从屏幕左侧移入。该过渡效果还有3个变化：

* CCTransitionMoveInB：新场景从屏幕的底部移入。
* CCTransitionMoveInT：新场景从屏幕的上方移入。
* CCTransitionMoveInR：新场景从屏幕的右侧移入。

1. CCTransitionSceneOriented

将整个场景翻转过来。该效果还有6个变化：

* + CCTransitionFlipX：场景横向翻转。
  + CCTransitionFlipY：场景纵向翻转。
  + CCTransitionFlipAngular：场景一半按横向，一半按纵向翻转。
  + CCTransitionZoomFlipAngular：场景一半按横向，一半按纵向翻转，同时有一定缩放。
  + CCTransitionZoomFlipX：场景纵向翻转，同时有一定缩放。
  + CCTransitionZoomFlipY：场景横向翻转，同时有一定缩放。

1. CCTransitionPageTurn

翻动书页的过渡效果。

1. CCTransitionRotoZoom

当前场景旋转变小，新场景旋转变大。

1. CCTransitionShrinkGrow

当前场景缩小，新场景在其之上变大。

1. CCTransitionSlideInL

新场景从左边滑入。该过渡效果还有另外3个变化：

* + CCTransitionSlideInR：新场景从右边滑入。
  + CCTransitionSlideInB：新场景从底部滑入。
  + CCTransitionSlideInT：新场景从顶部滑入。

1. CCTransitionSplitCols

将当前场景切成竖条，上下移动显示新场景。该过渡效果还有一个变化是CCTransitionSplitRows，将当前场景切成横条，左右移动显示新场景。

1. CCTransitionTurnOffTiles

将当前场景分成方块，用分成方块的新场景随机的替换当前场景分出的方块。

1. CCTransitionProgressRadialCCW

新场景沿着径向逆时针显示。该过渡效果还有一个变化是CCTransitionProgressRadialCW，让新的场景沿着径向顺时针显示。

1. CCTransitionProgressHorizontal

新场景沿着横向逐渐显示。该过渡效果还有一个变化是CCTransitionProgressVertical，让新场景沿着纵向逐渐显示。该效果是v2.0新增场景过渡效果。

1. CCTransitionProgressInOut

新场景由内到外逐渐显示。该过渡效果还有一个变化是CCTransitionProgressOutIn，让新场景由外到内逐渐显示。该效果是v2.0新增场景过渡效果。

为了更好地了解每种效果的实际表现，可以打开Cocos2D模板的示例项目cocos2d-ios.xcodeproj，运行TransitionsTest，查看实际场景过渡效果。感兴趣的读者也可以仔细分析TransitionsTest.m，看看这些效果是如何具体实现的。

## CCLayer层类

一个CCLayer是屏幕上可绘制的区域，可以是半透明的，这样就可以看到CCScene下面的其他层。在游戏编程的过程中，开发者大部分时间都需要跟层打交道。如图3-4所示，一个游戏场景包含3个层，背景层、动画层和菜单层。

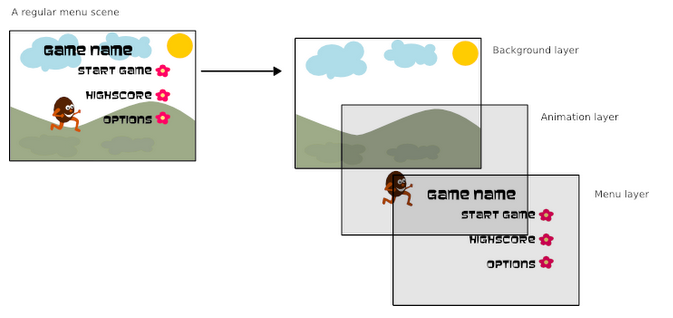


图3-4 一个游戏场景中包含3个层

CCLayer直接继承自CCNode，作为精灵节点和其他节点的容器，同时可以接收触摸输入和加速计输入的信息，前提是上述接收功能已被启用。

### CCLayer类的作用

CCLayer类的作用主要有三个：

1. 其他子节点的容器和组织者

比如，对一个层使用动作，那么所有在这个层上的物体都会受到影响。这意味着可以让同一层的所有物体一起移动、旋转和缩放。

如果所有这些物体是同一个层的子节点，就可以通过改变层的属性或者在层上执行动作,影响层上所有子节点。

1. 接收触摸事件

设置isTouchEnabled为YES，可以让层接收触摸事件。示例代码：

self.isTouchEnabled = YES;

该设定最好在init方法中设置。可以在任何时间将其设置为NO或者YES。

一旦启用isTouchEnabled属性，许多与接收触摸输入相关的方法将会开始被调用。这些事件包括：当新的触摸事件开始时，当手指在触摸屏上移动时，以及当用户手指离开屏幕以后。

第5章将详细描述在Cocos2D中如何处理多点触摸事件。

1. 接收加速计事件

和触摸输入一样，加速计必须在启用以后才能接收加速计事件。示例代码：

self.isAccelerometerEnabled = YES;

同样地，在层里要加入一个特定的方法接收加速计事件。下面是第2章中处理加速计事件的示例代码：

-(void) accelerometer:(UIAccelerometer \*)accelerometer didAccelerate:(UIAcceleration \*)acceleration{

float deceleration = 0.4f;

float sensitivity = 6.0f;

float maxVelocity = 100;

\_playerVelocity.x = \_playerVelocity.x \* deceleration + acceleration.x \* sensitivity;

if (\_playerVelocity.x > maxVelocity) {

\_playerVelocity.x = maxVelocity;

}else if(\_playerVelocity.x < -maxVelocity){

\_playerVelocity.x = -maxVelocity;

}

}

通过加速参数决定任意三个方向的加速度值。

第５章详细阐述在Cocos2D中如何处理加速计事件。

除了接收触摸事件和加速计事件，CCLayer还可以接收鼠标事件和键盘事件，当然对于iPhone开发来说没什么实际作用，也不用花更多的时间来了解。

提示 CCLayer还有一个子类CCLayerMultipex，可以容纳多个层，但每次只可激活其中的一个。通常不鼓励使用CCLayerMultipex。

### CCLayerColor色彩层

在之前版本中是CCColorLayer，现在使用CCLayerColor取而代之。CCLayerColor是一个透明的、可以按照RGB 设置填充颜色的层，是实现CCRGBAProtocol协议的CCLayer子类，继承了CCNode所有属性和方法，同时还可以接收触摸事件和加速计事件。

#### CCLayerColor类的属性

CCLayerColor具有如下属性：

* **color**：色彩，其变量类型是ccColor3B（由3个字节组成的色彩信息）。
* **opacity**：透明度，其变量类型是Glubyte。
* **blendFunc**：混合模式，其变量类型是ccBlendFunc。

#### CCLayerColor类的方法

CCLayerColor的方法如下：

1. –(id)initWithColor:(ccColor4B)color

初始化一个带有色彩的CCLayer，其宽度和高度是窗口的大小。

1. - (id) initWithColor:(ccColor4B) color

width: (GLfloat) w

height:(GLfloat) h

初始化一个带有色彩的CCLayer，并以点值指定宽度和高度。

1. + (id) layerWithColor: (ccColor4B)color；

创建一个带有色彩的CCLayer，其宽度和高度是窗口的大小。示例代码：

CCLayerColor\* layer1 = [CCLayerColor layerWithColor: ccc4(255, 255, 255, 80)];

1. + (id) layerWithColor: (ccColor4B) color

**width: (GLfloat) w**

**height:(GLfloat) h**

创建一个带有色彩的CCLayer，并以点值指定宽度和高度。示例代码：

CCLayerColor\* layer1 = [CCLayerColor layerWithColor: ccc4(255, 255, 0, 80)

width: 100

height: 300];

1. -（void）changeWidth: (GLfloat) w height: (GLfloat) h；

修改层的宽度和高度。示例代码：

[layerl changeWidth:newSize.width height:newSize.height];

在实际的使用中，可以用setContentSize方法替代该方法。

### CCLayerGradient渐变色层

CCLayerColor有一个子类CCLayerGradient，可以在背景上绘制渐变色。CCLayerGradient继承了CCLayerColor的所有特性，增加了渐变方向、插值模式等属性。以下代码初始化带有特定渐变效果的色彩层。示例代码：

self = [super initWithColor:ccc4(238,238,238,255) fadingTo:ccc4(250,250,250,235) alongVector:ccp(0,-1)];

扩展阅读

在结束本节的学习之前，需要对初始化方法有更清楚的认识。细心的朋友已经发现，在Cocos2D中，每个初始化方法都分别提供了实例方法和类方法两种，那么它们在实际使用中的区别是什么呢？

在Xcode中打开安装Cocos2D时的模板，cocos2d-ios工程。找到CCLayer.m文件，在CCLayerColor的实现代码中查看以下代码：

+ (id) layerWithColor:(ccColor4B)color width:(GLfloat)w height:(GLfloat) h

{

return [[[self alloc] initWithColor:color width:w height:h] autorelease];

}

很容易了解到，类方法（静态方法）调用了实例方法以初始化对象，并创建一个自动释放内存的对象。也就是说，类方法把两步合成一步，它们在实际使用的区别也在于此。对于开发者来说，一般情况下只需直接使用类方法来创建对象即可。这里其实就是工厂方法模式，它可以为对象的创建提供一个简单一致的接口。简便起见，后续章节中将不再对两者之间的区别重复说明。

### CCMenu菜单类

Cocos2D中提供了内置的菜单系统，用来创建游戏中的各种菜单，包括主菜单、游戏设置等。而实现菜单系统的就是CCMenu类及其子类。学习本节内容时，请在Cocos2D模板的示例项目cocos2d-ios.xcodeproj中打开MenuTest.m，查看相关示例代码。

CCMenu继承自CCLayer，是一个菜单管理选择画面层，该画面以Menu对象为集合类，由MenuItem类实例组成各种按钮。CCMenu类提供的方法主要用来按横向、竖向或者多行列排序展示MenueItem的类实例。同时，CCMenu只支持CCMenuItem节点作为它的子节点。

注意 CCMenuItem是基础类，不能直接用来创建菜单，它的作用主要是设置按钮状态以及处理回调方法。需要使用CCMenuItem的子类来创建定制菜单项。

CCMenuItem类的方法如下：

**（1）CCMenuItemLabel**

CCMenuItemLabel内置Label对象，将一个基本的Label转变成为一个菜单项，增加选中时的文字放大效果。支持三种标签类：CCLabelBMFont、CCLabelAtlas和CCLabelTTF。示例代码：

// Label Item (CCLabelBMFont)

CCLabelBMFont \*label1 = [CCLabelBMFont labelWithString:@"Setting" fntFile:@"font1.fnt"];

CCMenuItemLabel \*item1 = [CCMenuItemLabel itemWithLabel:label1 target:self selector:@"menuCallBackSetting"];

**（2）CCMenuItemAtlasFont**

直接继承自CCMenuItemLabel，将一个CCLabelAtlas转变为一个菜单项。增加选中时的文字放大效果。示例代码：

// Label Item (CCLabelAtlas)

CCLabelAtlas \*labelAtlas = [CCLabelAtlas labelAtlasWithString:@"0123456789" charMapFile:@"fps\_images.png" itemWidth:16 itemHeight:24 startCharMap:'.'];

CCMenuItemLabel \*item2 = [CCMenuItemLabel itemWithLabel:labelAtlas target:self selector:@selector(menuCallbackFun:)];

item2.disabledColor = ccc3(10,20,39);

item2.color = ccc3(10,220,255);

（3）CCMenuItemFont

直接继承自CCMenuItemLabel，可以创建直接设置字体的菜单项（内部实现时依然用到Label对象）。示例代码：

[CCMenuItemFont setFontSize:30];

[CCMenuItemFont setFontName: @"Courier New"];

// Font Item

CCMenuItem \*item3 = [CCMenuItemFont itemWithString: @"new items" target: self selector:@selector(menuCallbackNew:)];

以上三种字体型菜单项都是通过Label类及其派生类完成视觉呈现，借用父类CCMenuItem实现菜单项的回调方法。

**（4）CCMenuItemSprite**

内置3个CCNode<CCRGBAProtocol>对象，分别表示未选中、选中和禁止三个状态的图像。选中时没有特别的视觉效果。示例代码：

// Font Item

CCSprite \*labelNormal = [CCSprite spriteWithFile:@"menuitem.png" rect:CGRectMake(0,30\*2,120,30)];

CCSprite \*labelSelected = [CCSprite spriteWithFile:@"menuitem.png" rect:CGRectMake(0,30\*1,120,30)];

CCSprite \*labelDisabled = [CCSprite spriteWithFile:@"menuitem.png" rect:CGRectMake(0,30\*0,120,30)];

CCMenuItemSprite \*item4 = [CCMenuItemSprite itemWithNormalSprite:labelNormal selectedSprite:labelSelected disabledSprite:labelDisabled target:self selector:@selector(menuCallback:)];

**（5）CCMenuItemImage**

继承自CCMenuItemSprite，包含三个对象，分别表示未选中、选中和禁止三个状态的图像。示例代码：

// Image Item

CCMenuItem \*item5 = [CCMenuItemImage itemWithNormalImage:"button1.png" selectedImage:@"button2.png" disabledImage:@"button3.png" target:self selector:@selector(menuCallback2:)];

（6）CCMenuItemToggle

支持内部一个MenuItem数组负责展示不同的状态，实现状态切换。示例代码：

CCMenuItemToggle \*item1 = [CCMenuItemToggle itemWithTarget:self selector:@selector(menuCallback:) items:

[CCMenuItemFont itemWithString: @"On"],

[CCMenuItemFont itemWithString: @"Off"],

nil];

注意 从v2.0 开始，之前的itemFrom…方法全部更改为itemWith…方法。

想查看菜单的具体效果，可以运行Cocos2D官方模板项目cocos2d-ios.xcodeproj中的MenuTest测试。建议读者仔细分析MenuTest.m中的实现代码。

## CCTexture纹理类

在游戏运行中，所有图像文件（PNG、PVR）都被加载成GPU可以理解的OpenGL ES纹理，而精灵则对应着这些纹理图。Cocos2D内置一个纹理缓存管理器（CCTextureCache）来保持这些纹理图。这样可以极大加速创建新精灵，并充分利用已有的纹理图。不利的是，如果收到内存警报，Cocos2D会将当前未使用的纹理图（即引用计数为1的纹理图）全部从内存中清除。

首先了解和纹理相关的概念。

### 纹理和纹理图集

所有游戏角色都是以图像的形式存储在iPhone/iPad设备的内存中，通常使用的格式是PNG或JPEG。一旦这些图像被加载入内存，它们将以一种未压缩的纹理格式来存储。PNG是苹果官方推荐的用于iOS设备的图像存储格式。

#### 纹理（Texture）

游戏角色的图像文件在使用前必须解压缩，并转换成iPhone和iPad的GPU可以理解的格式，并被加载进RAM（随机存储器），这样的图像称为纹理。GPU原生支持一系列压缩格式，如PVRTC，其他格式必须存储为未压缩的图像数据。OpenGL ES可以使用这些数据在屏幕上绘制图像。所使用的PNG图像文件虽然在闪存中不占用多少空间，但是因为要解压缩，所以会在内存中占用更大的空间。

#### 纹理图集（TextureAtlas）

对于iPhone和iPad设备而言，内存是非常宝贵的。而且iOS设备的GPU使用共享显存，而不是独立显存。换句话说，GPU将使用主系统的内存来存储纹理图和几何图形。旧版iOS设备的内存是128MB。

让这种内存限制更捉襟见肘的是，旧版iOS设备中，图像填充到纹理中时，其长度和宽度必须使用2的乘方。虽然iPhone 3GS和iPhone 4、iPad等设备支持非2的乘方大小的纹理图，但在Cocos2D中，为了兼容所有设备，仍然使用2的乘方来填充纹理。当然，也可以在ccConfig.h文件中修改这一点。

为了节省内存空间，并减少纹理中的浪费空间，将把这些纹理拼合成为一个大的纹理图，称为纹理图集。纹理图集只是一个大的纹理图而已，其中包含所有的图像。想象有一大张纸，然后把自己的照片都贴在上面，在需要时从纸上把照片剪下来。如果想把所有照片一次性给别人，只需给这一大张纸就行，而不需一张张地递过去。OpenGL ES处理图像也是类似，如果使用纹理图集或精灵表单（Spritesheet）把所有图像一次性交给OpenGL ES来处理，比把单个图像逐个交给OpenGL ES处理要高效。

下面大致介绍CCTexture2D、CCTextureCache和CCTextureAtlas这三个纹理类。

### CCTexture2D、CCTextureCache和CCTextureAtlas

在Cocos2D中，使用CCTexture2D（纹理）从图片、文本或源数据中创建OpenGL 2D纹理，所创建的纹理对象使用2的乘方来填充。根据创建CCTexture2D对象的方法不同，纹理的真实图片大小可能和纹理大小略有差异。另外需要注意的是，纹理内容通常是上下颠倒的！关于该类的更多内容，可以参考CCTexture2D.h。

CCTextureCache（纹理缓存）作为单例使用，用于加载和管理纹理。一旦纹理加载完成，下次使用时可使用它返回之前加载的纹理，从而减少对GPU和CPU内存的占用。关于该类的更多内容，读者可以参考CCTextureCache.h。

使用CCTextureAtlas（纹理图集）来实现纹理图集。纹理图文件可以是PVRTC、PNG或任何Texture2D所支持的文件类型。可以对纹理图集的矩形进行实时的更新，添加，删除或重排序。关于该类的更多内容，读者可以参考CCTextureAtlas.h。

在Cocos2D的开发中，CCTexture2D和CCTextureCache在多个方法中都有体现，以CCSprite类的初始化方法之一为例：

-(id) initWithFile:(NSString\*)filename rect:(CGRect)rect

{

NSAssert(filename!=nil, @"Invalid filename for sprite");

CCTexture2D \*texture = [[CCTextureCache sharedTextureCache] addImage: filename];

if( texture )

return [self initWithTexture:texture rect:rect];

[self release];

return nil;

}

在以上方法中，使用CCTextureCache的单例将图片文件添加到纹理缓存中，并创建一个CCTexture2D纹理对象。如果该纹理对象存在，则使用initWithTexture方法来创建精灵对象。而initWithTexture的方法实现如下：

-(id) initWithTexture:(CCTexture2D\*)texture rect:(CGRect)rect

{

NSAssert(texture!=nil, @"Invalid texture for sprite");

// IMPORTANT: [self init] and not [super init];

if( (self = [self init]) )

{

[self setTexture:texture];

[self setTextureRect:rect];

}

return self;

}

在上述方法中，指定用于渲染精灵对象的纹理，以及矩形大小。

## CCSprite精灵类

CCSprite是Cocos2D游戏开发中最常用的类，用图片把精灵（sprite）显示在屏幕上。

在游戏开发中，经常会遇到精灵（sprite）这个术语。精灵是一个图像，可以在屏幕上独立地移动。一个精灵可能是玩家角色、敌人，或者是大的背景图像。一般情况下，精灵来自于开发者所准备的PNG或PVRTC图像。一旦图像载入内存，精灵将被转换成纹理图，从而被iPhone GPU用于在屏幕上渲染。

### CCSprite类的属性及方法

生成精灵最简单的方法是把图片文件加载进CCTexture2D纹理，然后将它赋给精灵。精灵可以接受其他精灵作为子节点。CCSprite默认锚点位置是（0.5，0.5），也就是节点的几何中心。

如果一个精灵的父类是标准CCNode节点，那么它和其他CCNode的属性和方法没有太大区别。如果一个精灵的父类或任何一个祖先类是CCSpriteBatchNode，那么具备的特性包括：更快地纹理渲染、不支持Camera、不支持GridBase、不能单独设置Alias/Antialias属性、不能单独设置混合模式、不支持视差滚动。

#### CCSprite类的属性

除了从CCNode类继承的属性外，CCSprite节点还有如下属性：

* **atlasIndex**：纹理集的索引，通常不建议修改该值。变量类型是NSUInteger。
* **batchNode**：对渲染CCSprite的CCSpriteBatchNode的弱引用（关于CCSpriteBatchNode将在3.5.4节介绍）。变量类型是CCSpriteBatchNode。
* **blendFunc**：遵循CCTextureProtocol协议。变量类型是ccBlendFunc（纹理的混合方法）。
* **color**：RGB色彩，遵循CCRGBAProtocol协议。变量类型是ccColor3B（使用3字节组成的色彩信息，分别代表R、G、B）。
* **dirty**：判断精灵是否需要在纹理集中更新。变量类型是BOOL。
* **flipX**：是否精灵会沿水平方向翻转。变量类型是BOOL。

该属性只会对精灵的纹理进行翻转，不会影响到精灵的子节点，同时也不会修改锚点位置。如果想要同时修改锚点，并对子节点也进行翻转，可以使用下面的方法：

sprite.scaleX \*= -1；

* **flipY**：是否精灵会沿垂直方向翻转。变量类型是BOOL。

该属性只会对精灵的纹理进行翻转，不会影响到精灵的子节点，同时也不会修改锚点位置。如果想要同时修改锚点，并对子节点也进行翻转，可以使用下面的方法：

sprite.scaleY \*= -1；

* **offsetPosition**：以点值计量的精灵位置的偏移，如果使用Zwoptex之类的编辑器，会自动计算该数值。变量类型是CGPoint。
* **opacity**：透明度，遵循CCRGBAProtocal协议。变量类型是Glubyte。
* **quad**：方形的信息（纹理坐标、顶点坐标、色彩等）。变量类型是ccV3F\_C4B\_T2F\_Quad。
* **textureAtlas**：到CCTextureAtlas的弱引用，该属性仅在该精灵使用CCSpriteBatchNode渲染时可用。变量类型是CCTextureAtlas。
* **textureRect**：以点值返回精灵对象的矩形框。变量类型是CGRect。
* **textureRectRotated**：纹理矩形是否被旋转。变量类型是BOOL。

#### CCSprite类的方法

除了从CCNode继承的方法，CCSprite还支持以下常用方法：

1. -(id) initWithBatchNode:(CCSpriteBatchNode\*)batchNode rect:(CGRect)rect；

使用CCSpriteBatchNode和矩形框（以点数表示）初始化一个精灵对象。具体实现代码可参考模板中的CCSprite.m文件。

正如之前所提到的，对开发者来说，更多的是使用类方法来直接创建所需要的对象。

该方法的一个变化是使用CCSpriteBatchNode和矩形（以像素值表示）初始化一个精灵对象。示例代码：

-(id)initWithBatchNode:(CCSpriteBatchNode \*) batchNode

rectInPixels:(CGRect) rect;

1. -(id)initWithCGImage:(CGImageRef)image key:(NSString \*)key；

使用CGImageRef和键值初始化一个精灵对象。CCTextureCache将使用该键值判断某个纹理是否已使用CGImage创建。例如某个有效键值可能是“sprite\_frame\_01”，如果键值是nil，则每次使用CCTextureCache创建一个新的纹理。

1. -(id)initWithFile:(NSString \*)filename rect:(CGRect)rect.

使用图片名称和矩形初始化一个精灵对象，位置偏移是（0，0）。该方法有一个变化，就是不指定rect，使用图片的大小作为矩形。示例代码：

-(id)initWithFile:(NSString \*)filename;

1. - (id) initWithSpriteFrame: (CCSpriteFrame \*) spriteFrame；

使用精灵帧初始化一个精灵。

1. - (id) initWithSpriteFrameName: (NSString \*) spriteFrameName；

使用精灵帧的名称初始化一个精灵。使用该方法时，将从CCSpriteFrameCache中获取一个CCSpriteFrame。如果不存在，则系统将抛出一个异常。

1. - (id) initWithTexture: (CCTexture2D \*)texture rect: (CGRect) rect；

使用纹理和矩形初始化一个精灵，位置偏移是（0，0）。该方法还有一个变化，就是使用纹理的大小作为矩形。示例代码：

-(id) initWithTexture: (CCTexture2D \*)texture;

1. - (CCSpriteFrame\*) displayedFrame；

返回当前所显示的精灵帧。示例代码：

[sprite setDisplayFrame:[self displayedFrame]];

1. -(BOOL) isFrameDisplayed:(CCSpriteFrame\*)frame；

是否某个CCSpriteFrame对象正在被显示。示例代码：

BOOL isBearDisplaed = [bear isFrameDisplayed:frame];

1. -(void) setDisplayFrame:(CCSpriteFrame\*)frame；

为CCSprite精灵设置新的显示帧。示例代码：

[self setDisplayFrame:spriteFrame];

1. - (void) setDisplayFrameWithAnimationName: (NSString \*) animationName

**index: (int) frameIndex；**

根据动画名称和索引更改要显示的帧，动画名将从CCAnimationCache中获取。示例代码：

[bear setDisplayFrameWithAnimationName:animation index:indexNumber];

1. - (void) setTextureRect: (CGRect) rect；

更新CCSprite精灵对象的纹理矩形大小。示例代码：

[self setTextureRect:rect];

提示 使用了setTextureRect再使用contentSize时要注意，不能再当成初始化时的contentSize大小来使用。

这个方法在实际开发中的作用何在呢？想象用Cocos2D开发一款RPG小游戏，游戏中有很多角色，而每个角色都有显示自己生命值的血条。只需在预定消息的方法中使用setTextureRect，就可以实时更改角色的血条长度。

1. - (void) setTextureRect: (CGRect) rect

**rotated: (BOOL) rotated**

**untrimmedSize: (CGSize) size；**

更新CCSprite精灵对象的纹理矩形，矩形旋转和未裁剪大小。示例代码：

[self setTextureRect:rect rotated:NO untrimmedSize:rect.size];

1. +(id)spriteWithBatchNode:(CCSpriteBatchNode \*) batchNode

**rect:(CGRect) rect；**

使用CCSpriteBatchNode和矩形（以点数表示）来创建一个精灵对象。首先看该方法的实现代码：

+(id) spriteWithBatchNode:(CCSpriteBatchNode\*)batchNode rect:(CGRect)rect

{

return [[[self alloc] initWithBatchNode:batchNode rect:rect] autorelease];

}

关于实例方法和类方法在实际使用中的区别，前面已经提到，大家只需要知道大多数情况下直接使用类方法来创建对象就可以了。示例代码：

CCSpriteBatchNode \*batch = [CCSpriteBatchNode batchNodeWithFile:@"blocks.png" capacity:150];

CCSprite \*sprite = [CCSprite spriteWithBatchNode:batch

rect:CGRectMake(100,100,32,32)];

1. +(id)spriteWithCGImage:(CGImageRef)image

key:(NSString \*)key；

使用CGImageRef和键值创建一个精灵对象。CCTextureCache将使用该键值判断某个纹理是否已使用CGImage创建。例如，某个有效键值可能是“sprite\_frame\_01”。如果键值是nil，则会每次使用CCTextureCache创建一个新纹理。示例代码：

UIImage\* img = [UIImage imageNamed:@"zebra.png"];

CCSprite\* spr = [CCSprite spriteWithCGImage:img.CGImage key:@"img"];

1. +(id)spriteWithFile:(NSString \*)filename

rect:(CGRect)rect；

使用图片名称和矩形创建一个精灵对象，位置偏移是（0，0）。该方法的使用频率相当高，大多数时候使用它创建精灵对象。示例代码：

CCSprite \*bear = [CCSprite spriteWithFile:@"bear.png" rect:CGRectMake(100, 100, 32, 32)];

该方法有一个变化，就是不指定rect，使用图片的大小作为矩形。大多数情况下无需特别指定rect的大小。示例代码：

CCSprite \*bear = [CCSprite spriteWithFile:@"bear.png"];

1. +(id) spriteWithSpriteFrame: (CCSpriteFrame \*) spriteFrame；

使用精灵帧创建一个精灵。示例代码：

CCSprite \*testPlayer = [CCSprite spriteWithSpriteFrame:startingFrame];

1. + (id) spriteWithSpriteFrameName: (NSString \*) spriteFrameName；

使用精灵帧的名称创建一个精灵。使用该方法时，将从CCSpriteFrameCache中获取该CCSpriteFrame。如果精灵帧缓存中不存在该精灵帧，系统将抛出一个异常。示例代码：

CCSprite \*player = [CCSprite spriteWithSpriteFrameName:@"player.png"];

1. + (id) spriteWithTexture: (CCTexture2D \*)texture

rect: (CGRect) rect；

使用纹理和矩形创建一个精灵，位置偏移是（0，0）。示例代码：

CCTexture2D \*playerTexture = [[CCTextureCache sharedTextureCache] addImage:@"player.png"];

CCSprite \*player2 = [CCSprite spriteWithTexture:playerTexture rect:CGRectMake(100, 100, 200, 200)];

该方法还有一个变化，就是使用纹理自身的大小作为矩形。示例代码：

CCTexture2D \*playerTexture = [[CCTextureCache sharedTextureCache] addImage:@"player.png"];

CCSprite \*player2 = [CCSprite spriteWithTexture:playerTexture];

1. -(void)updateTransform；

根据旋转角度、位置和比例值更新方形。示例代码：

CCSprite \*sprite = [CCSprite spriteWithBatchNode:batch rect:CGRectMake(100, 100, 200, 200)];

[sprite updateTransform];

注意 仅当该精灵对象是使用CCSpriteBatchNode渲染生成时，才可使用该方法。

CCSprite类在Cocos2D中起着极其重要的作用，游戏角色、敌人、场景、道具等几乎都会使用CCSprite来创建。对于CCSprite类的属性和方法，读者需要多去尝试，并在实践中逐渐体会到它的强大！

### CCSpriteBatchNode精灵表单

CCSpriteBatchNode前身是旧版本Cocos2D中的CCSpriteSheet，也就是精灵表单。在学习CCSpriteBatchNode之前，需要了解纹理图集的基本概念。

生活中处处充满了限制，游戏开发也不例外。事实上，制约游戏开发的最大问题在于突破硬件设备的性能瓶颈。游戏开发者最容易遇到的问题就是，当有大量的精灵出现在屏幕上时，游戏的运行变得奇慢无比。到目前为止，所有图像元素都是使用独立的CCSprite，看起来没什么不妥。但是，当屏幕上有15或20个元素时，会发现游戏帧速率急剧下降。添加的图像元素越多，帧速率下降越快。即便不提供任何游戏逻辑机制，只是简单地移动这些CCSprite精灵，随着精灵数量的增加，游戏还是会变得越来越慢。导致这一问题的原因很多，最重要的就是在导入图像元素时，开发者使用了单独的CCSprite，而不是CCSpriteBatchNode。

简而言之，调用过多的OpenGL ES命令，把每个图像作为单独的纹理来处理已经超过了GPU所能处理的上限。为了解决这一问题，最简单的方法就是使用纹理图集。

如果仔细看看CCSprite相关代码（CCSprite.h文件），会发现在游戏的每一帧都会调用-(void)draw方法。在draw方法中，每次在屏幕中绘制一个精灵时都会调用真实的OpenGL ES命令（gl为前缀的方法，详见gl.h）。对于每个精灵，OpenGL ES都需要将纹理图绑定在这个CCSprite上，然后将其绘制在屏幕上（称为渲染）。

在屏幕上真实显示图像的像素前，还有一件事要做：iOS提供的OpenGL ES驱动将把OpenGL ES的命令转换成GPU可以理解的硬件编码，从而让GPU显示图像。作为一个游戏开发者，无需了解更多驱动细节，只需明白一点，每次调用OpenGL ES的命令都将耗费OpenGL ES驱动的CPU时钟。如果尽可能地减少OpenGL ES的调用，游戏的运行将更为顺畅。

要实现这一点，就需要让OpenGL ES一次性处理所有纹理图，而使用CCSpriteBatchNode和纹理图集就能实现这一目的。

#### CCSpriteBatchNode类的属性

CCSpriteBatchNode是Cocos2D中一个特殊的类，其中可包含多个CCSprite。它直接继承自CCNode，其特有的属性如下：

* **textureAtlas**：所使用的纹理图集。变量类型是CCTextureAtlas。
* **blendFunc**：混合方法，遵循CCTextureProtocol协议。变量类型是ccBlendFunc。
* **descendants**：子孙节点。变量类型是CCArray。

#### CCSpriteBatchNode类的方法

除了继承自CCNode的方法，特有的方法如下：

1. - (id) initWithFile: (NSString \*) fileImage capacity:(NSUInteger)capacity；

使用特定格式的图片文件（PNG、JPEG、PVR等）初始化一个CCSpriteBatchNode精灵表单，可以容纳29个子节点。在实际运行时如果超出了限制，该数字还可以再增加约33%。图片文件将使用TextureMgr加载。

1. -(id)initWithTexture:(CCTexture2D \*)tex capacity:(NSUInteger)capacity；

使用CCTexture2D纹理和指定的子节点容量初始化一个CCSpriteBatchNode精灵表单， 。在实际运行时，如果超出了限制，该数字还可以再增加约33%。

1. + (id) batchNodeWithFile: (NSString \*) fileImage；

使用特定格式的图片文件（PNG、JPEG、PVR等）创建一个CCSpriteBatchNode精灵表单，可以容纳29个子节点。在实际运行时如果超出了限制，该数字还可以再增加约1/3。图片文件将使用TextureMgr加载。示例代码：

CCSpriteBatchNode\* spriteSheet = [CCSpriteBatchNode batchNodeWithFile:@"bullet.png"];

该方法还有一个变化，就是指定子节点的数量。示例代码：

CCSpriteBatchNode\* spriteSheet = [CCSpriteBatchNode batchNodeWithFile:@"bullet.png" capacity:12];

1. +(id)batchNodeWithTexture:(CCTexture2D \*)tex;

使用CCTexture2D纹理创建一个CCSpriteBatchNode精灵表单，可以容纳29个子节点。在实际运行时如果超出了限制，该数字还可以再增加约1/3。示例代码：

CCTexture2D\* gameArtTexture = [[CCTextureCache sharedTextureCache] addImage:@"bgStage1.png"];

CCSpriteBatchNode \*spriteBatch = [CCSpriteBatchNode batchNodeWithTexture:gameArtTexture];

该方法还有一个变化，就是指定子节点的数量。示例代码：

CCSpriteBatchNode \*spriteBatch = [CCSpriteBatchNode batchNodeWithTexture:gameArtTexture capacity:12];

1. - (void) removeChild: (CCSprite \*) sprite cleanup: (BOOL) doCleanup；

从CCSpriteBatchNode精灵表单中删除子精灵，由于这种操作速度非常慢，通常不推荐使用。示例代码：

[spriteSheet removeChild:bear cleanup:YES];

该方法还有一个变化，就是根据指定的索引删除子精灵。示例代码：

[spriteSheet removeChildAtIndex:2 cleanup:YES];

本书高级篇会详细说明如何使用纹理贴图和CCSpriteBatchNode实际提升游戏的性能。

提示 在使用精灵表单时，CCSpriteBatchNode、CCSpriteFrame和CCSpriteFrameCache常常配合使用。

### CCSpriteFrame精灵帧

与之前了解的节点类不同，CCSpriteFrame和CCSpriteFrameCache都直接继承自Objective-C中的NSObject。

CCSpriteFrame（精灵帧）中主要包括CCTexture2D纹理、矩形大小，用来表示一个精灵。

#### CCSpriteFrame精灵帧的属性

CCSpriteFrame（精灵帧）的主要属性如下：

* **rect和rectInPixels**：精灵帧的矩形大小，分别用点值和像素值来表示。这两个属性是彼此相连的，任一个属性的变化都会自动更新另一个属性。变量类型是CGRect。
* **rotated**：精灵帧的矩形框是否发生旋转。变量类型是BOOL。
* **offset和offsetInPixels**：精灵帧的位置偏移（以像素值计算），变量类型是CGPoint。
* **originalSize和originalSizeInPixels**：修剪后图片的原始大小（以像素值计算），变量类型是CGSize。
* **texture**：精灵帧的纹理，变量属性是CCTexture2D。
* **textureFilename**：精灵帧的纹理文件名称，变量属性是NSString。

#### CCSpriteFrame精灵帧的方法

CCSpriteFrame有两个实例方法和两个静态方法（类方法），分别如下：

1. -（id）initWithTexture:(CCTexture2D \*)texture rect: (CGRect) rect；

使用纹理、矩形（以点值表示）两个属性初始化一个CCSpriteFrame。使用此方法时，假定精灵帧未被裁剪过。

1. -(id)initWithTexture:(CCTexture2D \*)texture

**rectInPixels:(CGRect) rect**

**rotated:(BOOL) rotated**

**offset:(CGPoint) offset**

**originalSize:(CGSize) originalSize；**

使用纹理、矩形（以像素值表示）、是否旋转、位置偏移和原始大小等属性初始化一个CCSpriteFrame。此处的原始大小是未被裁剪前的精灵帧大小。

1. +（id）frameWithTexture:(CCTexture2D \*)texture

rect: (CGRect) rect；

使用纹理、矩形（以点值表示）两个属性创建一个CCSpriteFrame。使用此方法时，假定精灵帧未被裁剪过。示例代码：

CCSpriteFrame \*frame1 = [CCSpriteFrame frameWithTexture:spriteSheet.texture rect:CGRectMake(228, 0, 75, 100)];

（4）+(id)frameWithTexture:(CCTexture2D \*)texture

rectInPixels:(CGRect) rect

rotated:(BOOL) rotated

offset:(CGPoint) offset

originalSize:(CGSize) originalSize；

使用纹理、矩形（以像素值表示）、是否旋转、位置偏移和原始大小等属性创建一个CCSpriteFrame。此处的原始大小是未被裁剪前的精灵帧大小。示例代码：

CCSpriteFrame \*frame1 = [CCSpriteFrame frameWithTexture:batch.texture rectInPixels:rect rotated:YES offset:ccp(0,0) originalSize:size1];

### CCSpriteFrameCache精灵帧缓存

CCSpriteFrameCache（精灵帧缓存）主要用来存放CCSpriteFrame，因此它没有提供特别的属性，而是提供一系列用于管理CCSpriteFrame的方法，具体如下：

1. - (void) addSpriteFrame: (CCSpriteFrame \*) frame

**name: (NSString \*) frameName;**

使用指定名称添加一个精灵帧。如果该名称在CCSpriteFrameCache中已存在，则原有内容将被新精灵帧所替代。示例代码：

[[CCSpriteFrameCache sharedSpriteFrameCache] addSpriteFrame:frame name:[NSString stringWithFormat:@"playerFrame1"]];

1. - (void) addSpriteFramesWithDictionary: (NSDictionary \*) dictionary

**texture: (CCTexture2D \*) texture；**

使用词典添加多个精灵帧，纹理将和所创建的精灵帧关联在一起。示例代码：

[self addSpriteFramesWithDictionary:dict texture:texture];

1. - (void) addSpriteFramesWithDictionary: (NSDictionary \*) dictionary

**textureFilename: (NSString \*) filename；**

使用词典添加多个精灵帧，纹理文件名称将和所创建的精灵帧关联在一起。示例代码：

[self addSpriteFramesWithDictionary:dict textureFilename:textureFile];

1. - (void) addSpriteFramesWithFile: (NSString \*) plist；

从plist属性文件中添加多个精灵帧。纹理将会被自动载入。纹理的扩展名则将使用.png替代.plist。如果需要添加其他纹理，请使用addSpriteFramesWithFile:texture方法。示例代码：

//自动载入纹理，纹理名称使用.png替代.plist

[[CCSpriteFrameCache sharedSpriteFrameCache] addSpriteFramesWithFile:@"myspritesheet.plist"];

//添加其他纹理

[[CCSpriteFrameCache sharedSpriteFrameCache] addSpriteFramesWithFile:@"myspritesheet.plist" texture:myTexture];

1. - (void) addSpriteFramesWithFile: (NSString \*) plist

**textureFilename: (NSString \*) textureFileName；**

从plist文件中添加多个精灵帧，纹理文件名将和所创建的精灵帧关联在一起。示例代码：

CCSpriteFrameCache \*cache = [CCSpriteFrameCache sharedSpriteFrameCache];

[cache addSpriteFramesWithFile:@"t1.plist" textureFile:@"t1.png"];

1. + (void) purgeSharedSpriteFrameCache；

清除CCSpriteFrameCache中的缓存，删除所有精灵帧，并释放所保留的变量。示例代码：

[CCSpriteFrameCache purgeSharedSpriteFrameCache];

1. - (void) removeSpriteFrameByName: (NSString \*)Name；

根据名称从精灵帧缓存中删除一个精灵帧。示例代码：

[[CCSpriteFrameCache sharedSpriteFrameCache] removeSpriteFrameByName:playerSprite];

1. - (void) removeSpriteFrames；

清除已加载精灵帧的词典。当收到内存警报时会调用此方法。短时间内的作用是避免应用被iOS强行关闭，中长期的作用是分配更多的资源。示例代码：

[[CCSpriteFrameCache sharedSpriteFrameCache]removeSpriteFrames];

该方法还有一个变化，是从NSDictionary中删除多个精灵帧。示例代码：

[[CCSpriteFrameCache sharedSpriteFrameCache]removeSpriteFramesFromDictionary:dict];

1. - (void) removeSpriteFramesFromFile: (NSString \*) plist；

从plist属性文件中删除多个精灵帧。存储在该plist文件中的精灵帧都会被删除。示例代码：

[[CCSpriteFrameCache sharedSpriteFrameCache]removeSpriteFramesFromFile:plistName];

1. - (void) removeSpriteFramesFromTexture: (CCTexture2D \*) texture；

删除所有与特定纹理相关联的精灵帧。示例代码：

[[CCSpriteFrameCache sharedSpriteFrameCache]removeSpriteFramesFromTexture:texture];

1. -(void)removeUnusedSpriteFrames；

删除所有未使用的精灵帧。所有retain计数为1的精灵帧都会被删除。当游戏切换新场景时可以调用此方法。示例代码：

[[CCSpriteFrameCache sharedSpriteFrameCache]removeUnusedSpriteFrames];

-(CCSpriteFrame \*)spriteFrameByName:(NSString \*)name;

返回之前添加的某个精灵帧，如果未找到对应名称的精灵帧，将返回nil。如果要使用该方法，需要retain这个精灵帧。示例代码：

CCAnimation \*myAnim = [[CCAnimation alloc] initWithName:@"select" delay:0.15f];

[myAnim addFrame:[cache spriteFrameByName:@"anim1.png"]];

1. +(CCSpriteFrameCache \*)sharedSpriteFrameCache；

获取精灵帧缓存的单例对象。示例代码：

CCSpriteFrameCache \*frameCache1 = [CCSpriteFrameCache sharedSpriteFrameCache];

1. -(CCSpriteFrame\*)spriteFrameByName:(NSString\*)name

使用文件名获取此前添加的精灵帧。示例代码：

CCSpriteFrame \*frame1 = [[CCSpriteFrameCache sharedSpriteFrameCache]spriteFrameByName:@"frame1.png"];

## Cocos2D中的单例

单例的作用类似全局类，更像一个全局变量。如果需要在任何地方都能用到某些数据和方法，单例无疑是最佳的选择。比如某个游戏有多个关卡，可以使用单例来储存一些全局的游戏信息，并把信息从一个关卡传到另一个关卡。对于音频和屏幕大小来说，使用单例也是非常方便的选择。

Cocos2D中使用单例设计模式。单例是在程序的生命周期中只被实例化一次的类。为了确保这一点，利用类的静态方法创建和访问对象。因此通常情况下，使用以“shared”开头的方法访问Cocos2D的单例对象，而不使用Objective-C中常用的alloc/init或者静态autorelease初始化方法。

### Cocos2D中的常用单例

以下是一些常用的Cocos2D单例类及其访问方法：

* CCDirector\* sharedDirector = [CCDirector sharedDirector];
* CCSpriteFrameCache\* sharedCache = [CCSpriteFrameCache sharedSpriteFrameCache];
* CCTextureCache\* sharedTexCache = [CCTextureCache sharedTextureCache];
* CDAudioManager\* sharedManager = [CDAudioManager sharedManager];
* SimpleAudioEngine\* sharedEngine = [SimpleAudioEngine sharedEngine];

注意 从v2.0 开始， CCActionManager和CCTouchDispatcher不再作为单例，而是CCDirector的实例变量。

在以下代码中，SharedManager提供了访问MyManager单一实例的静态方法。如果实例不存在，分配并初始化一个MyManager实例；如果实例已存在，则返回已存在的实例。如代码清单3-7所示。

1. 访问MyManager单一实例的静态方法

static MyManager \*sharedManager = nil;

+ (MyManager\*) sharedManager

{

if (sharedManager == nil)

{

sharedManager = [[MyManager alloc] init];

}

return sharedManager;

}

使用单例也会有风险存于，由于其简单实用，可以在任何地方访问，开发者很多时候会在不恰当的地方使用单例。在创建单例之前，需要认真思考是否真的只需要该类的单个实例，或者需要在全局使用其中的数据，以及在后续版本中是否会对游戏设定做出重大调整。

注意 近年来由于滥用单例模式，已经被许多开发者列为一种“反模式”。很多时候单例由于其对象的生命周期过长，占用内存长时间得不到释放。这其实就是一种内存泄漏，在Java这种具有垃圾回收机制的语言中，这种长时间不释放的内存就是内存泄漏。所以，在Cocos2D 2.0版本中，许多类都改成非单例形式，大家在设计游戏类时也多多注意，不要滥用。

### CCDirector导演类

CCDirector（导演类）是Cocos2D游戏引擎的核心。在第1章的HelloCocos2D应用中，很多对象的初始化过程包含[CCDirector sharedDirector]的调用。

CCDirector是一个单例，它保存Cocos2D的全局配置设定，同时管理Cocos2D场景。主要作用如下：

* 访问和改变场景；
* 访问Cocos2D的配置细节；
* 访问视图（OpenGL、UIView、UIWindow）；
* 暂停、恢复和结束游戏；
* 在UIKit和OpenGL之间转换坐标。

#### CCDirector类的方法

CCDirector对象主要有如下方法：

1. 主程序启动并显示第一个场景。示例代码：

// Run the intro Scene

[[CCDirector sharedDirector] runWithScene: [HelloWorldLayer scene]];

注意 如果当前已有场景在运行，则不允许调用此方法。

1. 将传入场景设置为当前场景。示例代码：

[[CCDirector sharedDirector]pushScene:[Setting scene]];

挂起当前运行的场景，并压栈到待运行场景队列。

注意 为了减少内存占用，应避免待运行队列中场景数量。仅在当前已有场景运行时才可调用此方法。从v2.0 开始，pushScene成为运行第一个场景的官方推荐方法，读者打开AppDelegate.m就可以发现。

1. 运行待运行场景队列中的倒数第二个场景，当前场景被释放。示例代码：

[[CCDirector sharedDirector]popScene];

当待运行队列中没有其他场景时，系统会终止该操作。仅在当前已有场景运行时才可调用此方法。

在实际的游戏开发中，当很多地方都要用到某个场景时，比如设置游戏音效和音量的场景，这两个方法非常有用。可以单击Setting，调用pushScene进入设置场景，然后在设置场景中单击Back按钮调用popScene，游戏就会返回上一个状态。无论是在主菜单、游戏场景或其他地方打开设置菜单，该方法都会正常工作，无需随时记录设置菜单的状态。

注意 这种弹出式场景的内容不应过多，以免占用内存。

1. 直接使用新场景取代当前运行中的场景,并释放当前场景。示例代码：

[[CCDirector sharedDirector] replaceScene:MenuScene];

仅在当前已有场景运行时才可调用此方法。该方法在今后会经常用到。

注意 即便是经验丰富的开发者也经常因为调用replaceScene方法而遇到程序崩溃的情况。大多数情况下，是因为没有合理的retain场景中的对象。而replaceScene方法会释放所有retain的对象。要找到问题所在，可以逐行注释dealloc方法中释放对象的代码。

在Cocos2D模板中选择SceneTest，可以看到popScene、pushScene和replaceScene的实际效果。读者也可以仔细分析SceneTest.m文件中的具体内容。

1. 结束当前运行中的场景，并释放该场景。示例代码：

- (void)applicationWillTerminate:(UIApplication \*)application {

CCDirector \*director = [CCDirector sharedDirector];

[[director openGLView] removeFromSuperview];

[viewController\_ release];

[window\_ release];

//结束当前运行中的场景

[director end];

}

在HelloCocos2D项目的AppDelegate.m文件中可以看到该方法，但一般情况下开发者不会直接使用该方法，而是由程序自动调用。

1. 暂停当前运行中的场景。示例代码：

- (void)applicationWillResignActive:(UIApplication \*)application {

[[CCDirector sharedDirector] pause];

}

调用该方法后，当前画面还会存在，但预定的更新方法会停止。

在HelloCocos2D项目的AppDelegate.m文件中可以看到该方法，但一般情况下开发者不会直接使用该方法，而是由程序自动调用。

1. 恢复场景运行。示例代码：

- (void)applicationDidBecomeActive:(UIApplication \*)application {

[[CCDirector sharedDirector] resume];

}

调用该方法后，预定的更新方法会继续执行，同时delta time会被设置为0。

在HelloCocos2D项目的AppDelegate.m文件中可以看到该方法，但一般情况下开发者不会直接使用该方法，而是由程序自动调用。

1. 绘制场景。示例代码：

-(void)drawScene;

该方法会由系统自动每帧调用，请不要手动调用

1. 将UIKit坐标系中的坐标转换为OpenGL坐标系中的坐标。示例代码：

UITouch \*myTouch = [touches anyObject];

CGPoint location = [myTouch locationInView:[myTouch view]];

location = [[CCDirector sharedDirector] convertToGL:location];

上面的代码获取了触摸点的坐标，并将其转换成Cocos2D可以使用的坐标。通常用于转换多点触摸点的坐标。

1. 将OpenGL坐标系中的坐标转换为UIKit坐标系中的坐标。示例代码：

textField.center = [[CCDirector sharedDirector]　convertToUI:ccp(240, 250)];

通常用于将节点转换为窗口中的点，以便调用如glScissor之类的命令。

虽然Cocos2D的功能很强大，但有时在游戏中也需要使用UIKit的元素，比如使用虚拟键盘输入玩家的昵称等，这时需要使用该方法将Cocos2D中的坐标转换为UIKit中的坐标。

1. 清除所有自动缓存的Cocos2D数据。示例代码：

- (void)applicationDidReceiveMemoryWarning:(UIApplication \*)application {

[[CCDirector sharedDirector] purgeCachedData];

}

该方法将会清除CCTextureCache和CCLabelBMFont中的缓存。但CCSpriteFrameCache中的缓存不会被清除，需要手动清除。

1. 更改投射的尺寸，不太常用。示例代码：

[[CCDirector sharedDirector]reshapeProjection:[CCDirector sharedDirector].winSize];

1. 启用或禁用OpenGL的alpha（透明）混合处理。示例代码：

[[CCDirector sharedDirector]setAlphaBlending:YES];

1. 启用或禁用OpenGL的位深测试。示例代码：

[[CCDirector sharedDirector] setDepthTest:YES];

1. 获取导演对象的单例,该方法将在开发中频繁使用。示例代码：

CCDirector \*director = [CCDirector sharedDirector];

实际上，每次使用CCDirector时都需要调用该方法。

1. 停止动画，屏幕上将不会绘制东西。示例代码：

[[CCdirector sharedDirector] stopAnimation];

1. 启动动画，仅当调用过stopAnimation方法后才可使用该方法。示例代码：

[[CCdirector sharedDirector] startAnimation];

1. 获取屏幕中OpenGL视图的大小，以points(点值)显示。示例代码：

CGSize windowSize = [[CCDirector sharedDirector] winSize];

该方法会考虑到视窗或设备的旋转。此外，如使用winSizeInPixels会以像素值显示。在Mac系统下，winSize和winSizeInPixels会返回相同的值。

1. 设置OpenGL的默认值。示例代码：

[[CCdirector sharedDirector] setGLDefaultValues];

1. 创建FPS标签。示例代码：

[[CCdirector sharedDirector] createFPSLabel];

#### CCDirector类的属性

CCDirector对象具有以下属性：

* **runningThread**：当前运行的Cocos2D线程，变量类型为NSThread。
* **runningScene**：当前运行的场景，CCDirector一次只能显示一个场景，变量类型为CCScene。
* **animationInterval**：游戏的FPS（帧每秒）值，变量类型为NSTimeInterval。
* **displayStats**: 是否显示CCDirector的各种统计信息，变量类型为ccDirectorStats。

该属性为2.0版本中添加的新属性，它具有三种可能的值，分别是kCCDirectorStatsNone（不显示统计信息），kCCDirectorStatsFPS（帧每秒的信息），kCCDirectorStatsMPF(毫秒每帧的信息)。

* **openGLView**：OpenGLView视图，所有要进行视觉呈现的对象都会在该视图中进行渲染，变量类型为CC\_GLVIEW。
* **nextDeltaTimeZero**：下一个时间增量是否为0，变量类型为BOOL。
* **isPaused**：CCDirector是否被暂停，变量类型为BOOL。
* **projection**：设置OpenGL投射，变量类型为ccDirectorProjection。
* **totalFrames**：自CCDirector对象启动以来已调用帧的个数。
* **millisecondPerFrame**:每帧所耗用的毫秒时间，变量类型为ccTime。
* **sendCleanupToScene**：被替换的场景是否收到cleanup（清除）消息。如果新场景是弹出的，则原来的场景不会收到清除消息；如果新场景会直接替代旧场景，则原来的场景会收到清除消息。变量类型为BOOL。
* **notificationNode**：当主场景被访问时会用到此对象，用于hook（钩住）某个通知对象，如CCNotifications。变量类型为id。
* **projectionDelegate**：OpenGL投射更新时，且仅使用kCCDirectorProjectionCustom投射时调用该对象。变量类型是id<CCProjectionProtocol>。

## 垂直射击游戏：加载游戏数据

为了使读者对CCSprite和各相关类的使用有更加直观的印象，下面结合前面的游戏示例，使用精灵表单优化游戏性能，同时在游戏开始和结束时添加菜单，让玩家对游戏有更多控制权。当然，在这个示例小游戏中，这种优化是看不出差别的。但这是最佳实践，建议读者以后编写游戏都以这种方式使用精灵。

### 注释draw方法和背景

首先，在XCode中打开之前的项目，把draw方法注释掉，同时取消先前注释掉的添加游戏背景的代码段，编译并运行如图3-5所示。

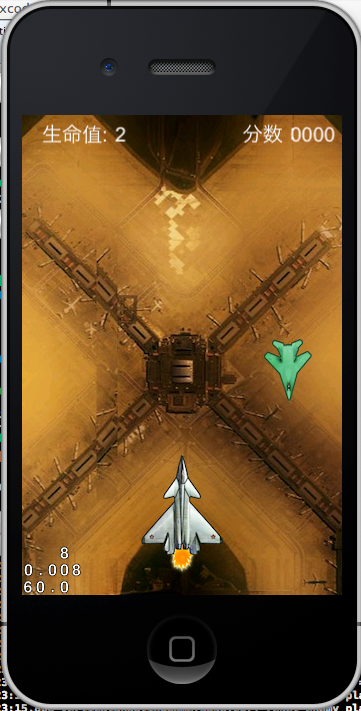


图3-5注释draw方法和背景相关代码后的场景

注意 这时必须触碰玩家飞机才可以发射子弹，同时，此刻的碰撞检测代码也有所差别，当然，玩家是感觉不到的。

### 加载游戏资源

首先需要让玩家决定什么时候开始游戏，因此要在游戏正式开始前添加一个菜单用来开始游戏。只有当玩家触碰此菜单时，才会正式开始游戏。为了使游戏开发更加贴近实际开发，这里介绍如何制作一个LoadingScreen场景来加载游戏所需要的图片、音效、背景音乐、存档等资源。后面介绍如何使用位图字体的章节，还可以加载预先加载字体资源。

#### 创建LoadingScreen

简单起见，该LoadingScreen只显示一个Loading字样，没有使用进度条。后面章节的实例中再考虑使用进度条，这样用户体验会更好一些。虽然界面显示比较简单，但这里向大家介绍的是一种通用的预先加载游戏资源的做法。以后任何Cocos2D游戏都可以使用此方法进行加载。

单击File→New→File，从左边选择Cocos2D v2.x模板 ，在右边的模板类中选择CCNode class模板类。选择CCLayer作为基类，然后选择next并保存到项目的VerticalShootingGame文件夹下。

用代码清单3-8中的代码替换LoadingScreen.h的内容。

1. LoadingScreen.h文件替换代码

#import <Foundation/Foundation.h>

#import "cocos2d.h"

@interface LoadingScreen : CCLayer {

CGSize winSize;

CGPoint winCenter;

int assetCount;

}

+(CCScene \*) scene;

-(void) loadMusic:(NSArray \*) musicFiles;

-(void) loadSounds:(NSArray \*) soundClips;

-(void) loadSpriteSheets:(NSArray \*) spriteSheets;

-(void) loadImages:(NSArray \*) images;

-(void) loadAssets:(NSArray \*) assets;

-(void) progressUpdate;

-(void) loadingComplete;

@end

LoadingScreen继承自CCLayer，提供一个静态scene方法供CCDirector对象调用。然后定义3个成员变量，其中winSize代表游戏屏幕窗口大小，winCenter代表窗口的中点，assetCount用来保存游戏需要加载的资源总数。

#### Loading方法

重点是接下来一系列的loading方法，每个方法都接收一个NSArray数组作为参数。这些参数都是一些具体资源的文件名，这个参数的值是从一个配置文件中读出来的，后面在介绍实现时会给出来。这些方法及用途如下：

* -(void) loadMusic:(NSArray \*) musicFiles; （加载背景音乐）
* -(void) loadSounds:(NSArray \*) soundClips;（加载游戏音效）
* -(void) loadSpriteSheets:(NSArray \*) spriteSheets;（加载表灵表单）
* -(void) loadImages:(NSArray \*) images;（加载背景图片等图片资源）
* -(void) loadAssets:(NSArray \*) assets;（加载游戏字体、存档等资源）
* -(void) progressUpdate;（更新游戏进度条，目的只是计算何时加载完成）
* -(void) loadingComplete;（资源全部加载完成，切换到另一个游戏场景）

#### LoadingScreen的具体实现

打开LoadingScreen.m文件，用代码清单3-9替换其中的内容。

1. LoadingScreen.m

#import "LoadingScreen.h"

#import "SimpleAudioEngine.h"

//The next scene you wish to transition to.

#import "HelloWorldLayer.h"

@implementation LoadingScreen

+(CCScene \*) scene

{

// 'scene' is an autorelease object.

CCScene \*scene = [CCScene node];

NSString \*className = NSStringFromClass([self class]);

// 'layer' is an autorelease object.

id layer = [NSClassFromString(className) node];

// add layer as a child to scene

[scene addChild: layer];

// return the scene

return scene;

}

-(id) init

{

if ( ( self = [ super init] ) )

{

winSize = [[CCDirector sharedDirector] winSize];

winCenter = ccp(winSize.width / 2, winSize.height / 2);

CCLabelTTF \*loadingText = [CCLabelTTF labelWithString:@"Loading..." fontName:@"Arial" fontSize:20];

loadingText.position = ccpAdd(winCenter, ccp(0,50));

[self addChild:loadingText];

}

return self;

}

-(void) onEnterTransitionDidFinish

{

[super onEnterTransitionDidFinish];

NSString \*path = [[CCFileUtils sharedFileUtils] fullPathFromRelativePath:@"preloadAssetManifest.plist"];

NSDictionary \*manifest = [NSDictionary dictionaryWithContentsOfFile:path];

NSArray \*spriteSheets = [manifest objectForKey:@"SpriteSheets"];

NSArray \*images = [manifest objectForKey:@"Images"];

NSArray \*soundFX = [manifest objectForKey:@"SoundFX"];

NSArray \*music = [manifest objectForKey:@"Music"];

NSArray \*assets = [manifest objectForKey:@"Assets"];

assetCount = ([spriteSheets count] + [images count] + [soundFX count] + [music count] + [assets count]);

if (soundFX)

[self performSelectorOnMainThread:@selector(loadSounds:) withObject:soundFX waitUntilDone:YES];

if (spriteSheets)

[self performSelectorOnMainThread:@selector(loadSpriteSheets:) withObject:spriteSheets waitUntilDone:YES];

if (images)

[self performSelectorOnMainThread:@selector(loadImages:) withObject:images waitUntilDone:YES];

if (music)

[self performSelectorOnMainThread:@selector(loadMusic:) withObject:music waitUntilDone:YES];

if (assets)

[self performSelectorOnMainThread:@selector(loadAssets:) withObject:assets waitUntilDone:YES];

}

-(void) loadMusic:(NSArray \*) musicFiles

{

CCLOG(@"Start loading music");

for (NSString \*music in musicFiles)

{

[[SimpleAudioEngine sharedEngine] preloadBackgroundMusic:music];

[self progressUpdate];

}

}

-(void) loadSounds:(NSArray \*) soundClips

{

CCLOG(@"Start loading sounds");

for (NSString \*soundClip in soundClips)

{

[[SimpleAudioEngine sharedEngine] preloadEffect:soundClip];

[self progressUpdate];

}

}

-(void) loadSpriteSheets:(NSArray \*) spriteSheets

{

for (NSString \*spriteSheet in spriteSheets)

{

[[CCSpriteFrameCache sharedSpriteFrameCache] addSpriteFramesWithFile:spriteSheet];

[self progressUpdate];

}

}

-(void) loadImages:(NSArray \*) images

{

CCLOG(@"LoadingScreen - loadImages : You need to tell me what to do.");

for (NSString \*image in images)

{

//Do something with the images

[self progressUpdate];

}

}

-(void) loadAssets:(NSArray \*) assets

{

//Overwrite me

CCLOG(@"LoadingScreen - loadAssets : You need to tell me what to do.");

for (NSString \*asset in assets)

{

//Do something with the assets

[self progressUpdate];

}

[self progressUpdate];

}

-(void) progressUpdate

{

if (--assetCount)

{

//留着后面显示进度条用

}

else {

[self loadingComplete];

CCLOG(@"All done loading assets.");

}

}

-(void) loadingComplete

{

CCDelayTime \*delay = [CCDelayTime actionWithDuration:2.0f];

CCCallBlock \*swapScene = [CCCallBlock actionWithBlock:^(void) {

[[CCDirector sharedDirector] replaceScene:[CCTransitionFade transitionWithDuration:1.0f scene:[HelloWorldLayer scene]]];

}];

CCSequence \*seq = [CCSequence actions:delay, swapScene, nil];

[self runAction:seq];

}

@end

虽然代码比较多，但是不要害怕，这里会依次解释其中的每个方法。

1. +(CCScene \*) scene

这个类方法的实现代码跟一般的有一些不同，使用Objective-C的动态语言特性反射。首先使用NSStringFromClass得到类名。

NSString \*className = NSStringFromClass([self class]);

此处className为LoadingScreen。然后使用NSClassFromString得到该类的类型。

id layer = [NSClassFromString(className) node];

这里还给该类型发送一个node消息。这种反射和任意发送消息的能力使Objective-C语言具有强大的灵活性，也为游戏开发提供大量的便利。可以把关卡的配置、敌人的类型直接存储为文件，然后反射解析。

1. init方法

该方法的实现非常简单，就是计算屏幕窗口大小和中点位置，然后初始一个CCLabelTTF对象，用来显示Loading字样。

下面看比较重要的onEnterTransitionDidFinish方法。

1. onEnterTransitionDidFinish方法

该方法首先加载一个配置文件，并读取配置文件中的游戏资源名字列表并存储在不同的数组中。示例代码：

NSString \*path = [[CCFileUtils sharedFileUtils] fullPathFromRelativePath:@"preloadAssetManifest.plist"];

NSDictionary \*manifest = [NSDictionary dictionaryWithContentsOfFile:path];

NSArray \*spriteSheets = [manifest objectForKey:@"SpriteSheets"];

NSArray \*images = [manifest objectForKey:@"Images"];

NSArray \*soundFX = [manifest objectForKey:@"SoundFX"];

NSArray \*music = [manifest objectForKey:@"Music"];

NSArray \*assets = [manifest objectForKey:@"Assets"];

assetCount = ([spriteSheets count] + [images count] + [soundFX count] + [music count] + [assets count]);

首先使用CCFileUtils获得preloadAssetManifest.plist文件的具体路径。调用NSDictionary的dictionaryWithContentsOfFile把该文件转换成一个字典对象。然后通过key值取出每种不同类型资源的数组。最后调用数组的count方法得到总共需要加载的资源数量。

继续之前，需要添加资源。把chapter3/resource/progress目录下的资源全部添加进来。然后看一看preloadAssetManifest.plist的具体配置情况。打开Resources分组下的arts文件夹，打开该文件并把每一项都展开，如图3-6所示。

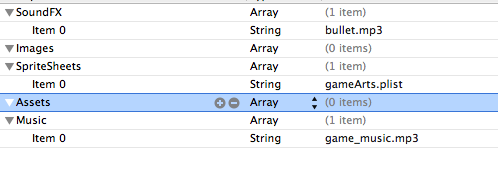


图3-6打开Resources分组下的arts文件夹

由上图可以看出，加载的音效是bullet.mp3,精灵表单为gameArts.plist，背景音乐是game\_music.mp3。

提示 由于本项目比较简单，资源数量有限，完成加载场景以后基本体会不到加载的过程，因为速度实在是太快了。

接下来看看后面的代码。

if (soundFX)

[self performSelectorOnMainThread:@selector(loadSounds:)

withObject:soundFX waitUntilDone:YES];

if (spriteSheets)

[self performSelectorOnMainThread:@selector(loadSpriteSheets:)

withObject:spriteSheets waitUntilDone:YES];

if (images)

[self performSelectorOnMainThread:@selector(loadImages:)

withObject:images waitUntilDone:YES];

if (music)

[self performSelectorOnMainThread:@selector(loadMusic:)

withObject:music waitUntilDone:YES];

if (assets)

[self performSelectorOnMainThread:@selector(loadAssets:)

withObject:assets waitUntilDone:YES];

这段代码并不复杂，调用performSelectorOnMainThread在主线程上面依次加载每种类型的游戏资源，同时waitUntilDone的值为YES能够保障所有的资源按照序列依次加载。

其他方法这里就不一一介绍了，注意几个方法。比如:

* 预先加载音效：[[SimpleAudioEngine sharedEngine] preloadEffect:soundClip]
* 预先加载背景音乐：[[SimpleAudioEngine sharedEngine] preloadBackgroundMusic:music]
* 预先加载PNG图片：[[CCTextureCache sharedTextureCache] addImage:image]
* 预先加载精灵表单可以使用下面的方法：

[[CCSpriteFrameCache sharedSpriteFrameCache] addSpriteFramesWithFile:spriteSheet];

这个方法首先加载与该plist方法名称相同但后缀为png的纹理图片。把该plist的所有spirteFrame信息读取出来，后面可以再通过spriteFrameWithName获取相应的精灵帧。

1. progressUpdate和loadingComplete方法

目前，progressUpdate方法非常简单，只是更新资源的总数，当资源全部加载完毕时会调用loadingComplete方法。loadingComplete方法的作用是：过2秒之后运行一个场景切换特效跳转到游戏主场景，即HelloWorldLayer。

### 修改AppDelegate.m文件

1. 添加所包含的头文件。在文件的顶部添加代码：

#import "LoadingScreen.h"

1. 修改- (BOOL)application:(UIApplication \*)application

didFinishLaunchingWithOptions:(NSDictionary \*)launchOptions方法。

在该方法最后，把之前的pushScene方法改成下面代码：

[director\_ pushScene: [LoadingScreen scene]];

编译并运行游戏，首先会看到一个loading界面，再进入游戏场景。但是我们希望玩家可以在场景跳转之后，有一个开始游戏的按钮，只有当玩家选择“开始游戏”时，敌机才开始出现，游戏才正式开始。因此这里需要对HelloWorldLayer的实现做一些修改。

### 修改HelloWorldLayer

1. 打开HelloWorldLayer.h文件，其中添加下列变量：

CCMenu \*\_startGameMenu;

BOOL \_isGameStarted;

1. 打开HelloWorldLayer.m的init方法，添加代码如代码清单3-10所示。
2. 在init方法中添加代码

//15.add game start menu & relative game logic

\_isGameStarted = NO;

[CCMenuItemFont setFontSize:20];

[CCMenuItemFont setFontName:@"Arial"];

CCMenuItemFont \*startItem = [CCMenuItemFont itemWithString:@"开始游戏" block:^(id sender)

{

\_isGameStarted = YES;

CCMenuItem \*item = (CCMenuItemFont\*)sender;

item.visible = NO;

//6.spawn enemy after 1.0 sec

[self performSelector:@selector(spawnEnemy)

withObject:nil

afterDelay:1.0f];

//7.enable accelerometer

self.isAccelerometerEnabled = YES;

//9.enable touch

self.isTouchEnabled = YES;

}];

startItem.position = ccp(winSize.width / 2, winSize.height / 2);

\_startGameMenu = [CCMenu menuWithItems:startItem, nil];

\_startGameMenu.position = CGPointZero;

[self addChild:\_startGameMenu];

首先注意，这里把之前的6、7、9行代码移到block方法的内部。因为只有当玩家触碰“开始游戏”按钮之后，才会正式允许玩家控制飞机的飞行以及发射子弹。这里需要注意，一定要把Menu的position设置为CGPointZero。

此时编译并运行，加载场景过程效果如图3-6所示，加载完资源以后的游戏画面如3-7所示。

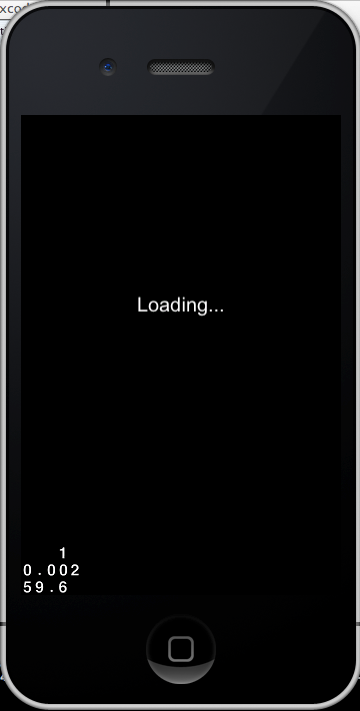


图3-6 加载场景画面 图3-7加载完资源以后的游戏画面

提示 这里所使用的菜单是最简陋的一种，直接使用系统内置字体创建的。当然，还可以从label、sprite创建更加丰富多彩的按钮，参考Cocos2D自带的MenuTest项目。

### 代码重构

接下来要解决本章最重要的代码重构问题，使用SpriteFrameCache和SpriteFrame初始化精灵。这里用到精灵表单来优化游戏性能。

#### 添加精灵表单

在init方法的最上面添加代码：

CCSpriteBatchNode \*batchNode = [CCSpriteBatchNode batchNodeWithFile:@"gameArts.png"];

batchNode.position = CGPointZero;

[self addChild:batchNode z:0 tag:kTagBatchNode];

这里给spriteBatchNode添加一个新的tag，所以需要在HelloWorldLayer.m文件最上方定义这个枚举常量：

enum {

kTagPalyer = 1,

kTagBatchNode = 2,

};

#### 修改sprite的初始化方式

把spriteWithFile改写成spriteWithSpriteFrameName，同时把sprite加到batchNode中，而不是加到layer里。这样做的好处是减少opengl call的次数，提高游戏渲染性能。具体如代码清单3-11所示。

1. 使用精灵表单减少opengl call的次数

//2.add background

CCSprite \*bgSprite = [CCSprite spriteWithSpriteFrameName:@"background\_1.jpg"];

bgSprite.position = ccp(winSize.width / 2,winSize.height/2);

[batchNode addChild:bgSprite z:-100];

//3.add player's plane

CCSprite \*playerSprite = [CCSprite spriteWithSpriteFrameName:@"hero\_1.png"];

playerSprite.position = CGPointMake(winSize.width / 2, playerSprite.contentSize.height/2 + 20);

[batchNode addChild:playerSprite z:4 tag:kTagPalyer];

//5.initialize 10 enemy sprites & add them to \_enemySprites array for future useage

const int NUM\_OF\_ENEMIES = 10;

for (int i=0; i < NUM\_OF\_ENEMIES; ++i) {

CCSprite \*enemySprite = [CCSprite spriteWithSpriteFrameName:@"enemy1.png"];

enemySprite.position = ccp(0,winSize.height + enemySprite.contentSize.height + 10);

enemySprite.visible = NO;

[batchNode addChild:enemySprite z:4];

[\_enemySprites addObject:enemySprite];

}

//10.init bullets

\_bulletSprite = [CCSprite spriteWithSpriteFrameName:@"bullet1.png"];

\_bulletSprite.visible = NO;

[batchNode addChild:\_bulletSprite z:4];

此时运行代码，会发现有点问题，游戏行为变得有些古怪。因为之前操作playerSprite时是通过[self getChildByTag:kTagPlayer]来获取玩家精灵的。现在要改成[batchNode getChildByTag:kTagPlayer]。

#### 定义-(CCSprite\*) getPlayerSprite方法

为了减少重复修改，这里需要定义一个新方法-(CCSprite\*) getPlayerSprite，其实现如代码清单3-12所示。

1. -(CCSprite\*) getPlayerSprite实现

-(CCSprite\*)getPlayerSprite{

CCSpriteBatchNode \*batchNode = (CCSpriteBatchNode\*)[self getChildByTag:kTagBatchNode];

return (CCSprite\*)[batchNode getChildByTag:kTagPalyer];

}

理解上面代码，头脑中需要有一个node的层级关系图，首先通过当前layer找到batchNode，然后通过batchNode找到playerSprite。

此时还需要稍微修改其他地方获得playerSprite的方法，修改以下代码

CCSprite \*playerSprite = (CCSprite)[self getChildByTag:kTagPlayer];

将其改写成：

CCSprite \*playerSprite = [self getPlayerSprite];

共有三处需要修改，可以参考附书源码，这里就不贴出来了。

#### 修改发射子弹的碰撞检测算法

单击玩家发射子弹的碰撞检测算法也要进行修改，如代码清单3-13所示。

1. 修改单击玩家就发射子弹的碰撞检测算法

-(void) ccTouchesEnded:(NSSet \*)touches withEvent:(UIEvent \*)event{

//修改成，必须选中playerSprite才能够发射子弹

UITouch \*touch = [touches anyObject];

CCSprite \*playerSprite = [self getPlayerSprite];

CGPoint pt;

//简化为下面的一句代码调用

CCSpriteBatchNode \*batchNode = (CCSpriteBatchNode\*)[self getChildByTag:kTagBatchNode];

pt = [batchNode convertTouchToNodeSpace:touch];

if (CGRectContainsPoint(playerSprite.boundingBox, pt)) {

\_isTouchToShoot = YES;

CCLOG(@"touched!");

}

}

相信通过这段代码读者能更清楚地明白，bondingBox是相对于节点的父节点来计算的。如果要判断一个点是否在一个矩形区域内，首先把它们都转化到同一个坐标系中，才能进行相应的判断。本例中把touch点和玩家飞机的矩形区域都转化到batchoNode的坐标空间中。

将-(void) updatePlayerPosition:(ccTime)dt方法中计算图片纹理宽度一半的语句：

// float imageWidthHavled = playerSprite.texture.contentSize.width \* 0.5f;

改成以下代码：

float imageWidthHavled = playerSprite.textureRect.size.width \* 0.5f;

注意 警惕使用sprite.texture.contentSize,只有当使用精灵图片文件初始化精灵时这种写法才有效，如果使用spriteBatchNode则会失效！！！

编译并运行到真机上面，好好享受学习的成果吧！

注意 如果使用CCSpriteBatchNode创建精灵对象，就不能将精灵对象添加为当前层的子节点，而应添加为精灵表单的子节点。另外，以上方法是为了让读者对CCSprite相关类有更清晰的认识，而在实际游戏开发中，并不需要使用上述这么复杂的步骤。

## 本章小结

本章对Cocos2D中的几个核心类（CCNode、CCScene、CCLayer、CCSprite）进行了详细的介绍，并且通过节点层级图让读者了解到Cocos2D游戏的基本组成；然后，介绍了Cocos2D中的单例。通过完善第2章的游戏实例，让大家对本章的知识如何运用到具体的游戏开发过程有了更直观的了解。

第4章学习使用动作、动画与特效，带你进入Cocos2D更为精彩的世界。