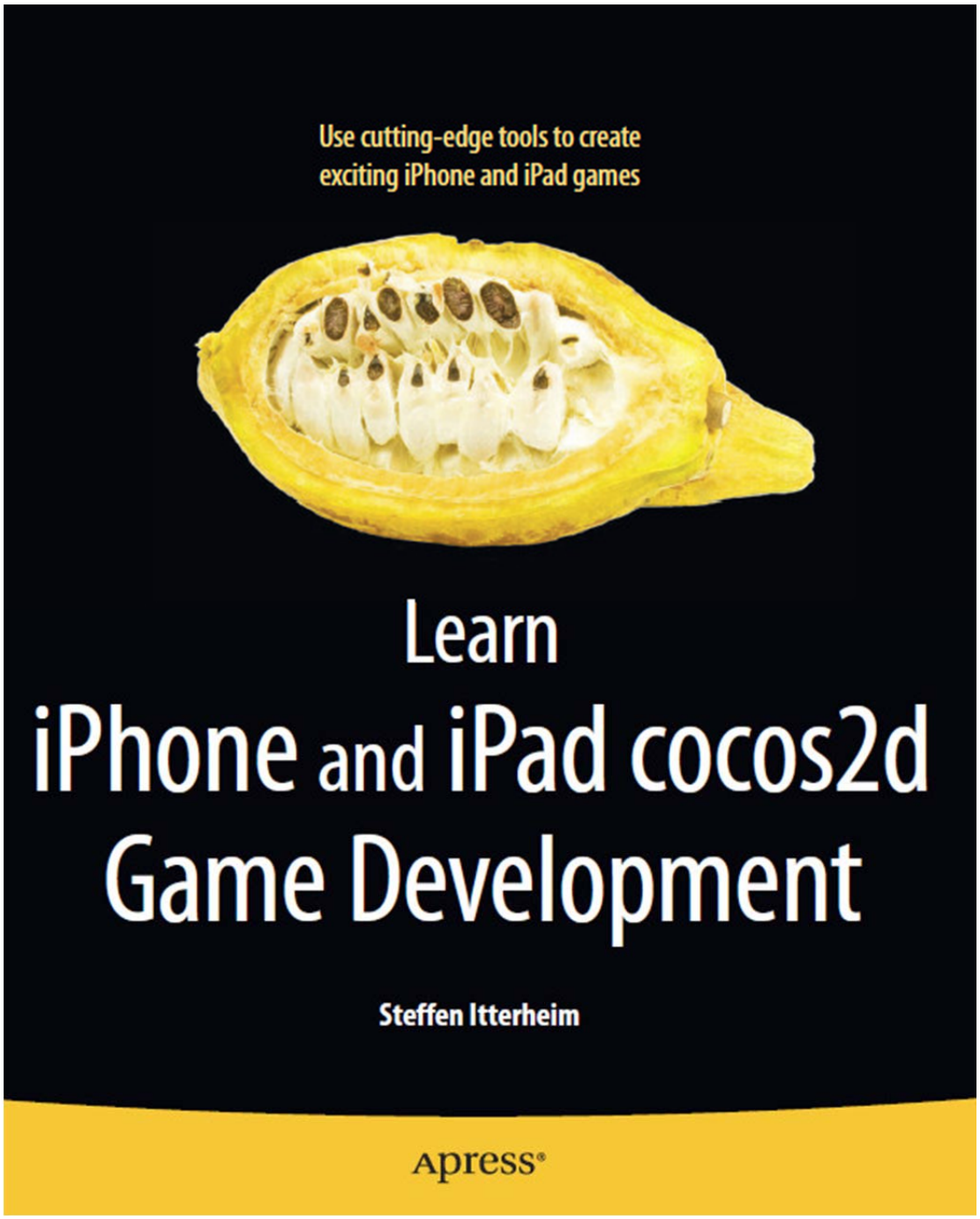

译者：杨栋
邮箱：yangdongmy@gmail.com

第六章
深入了解精灵（Sprite）

本章我们将深入了解精灵（Sprite）。我们可以通过很多方式用单个文件或者纹理贴图集（Texture Atlases）来生成精灵。我也会在本章介绍如何创建和播放精灵动画。

纹理贴图集是一张包含很多图片的纹理贴图（图片），通常用于存放单个角色动画的所有动画帧。不过它的作用不止于此。实际上你可以把任何图片放进同一张纹理贴图中。我们的目的是把尽可能多的图片放进同一张纹理贴图中，以达到节省空间的目的。很多cocos2d开发者使用Zwoptex这个软件来创建纹理贴图集。我会在本章介绍Zwoptex这个工具。

“精灵批处理”（Sprite Batching）是用于提高精灵渲染速度的技术。它可以提高渲染大量相同精灵的速度，不过它同纹理贴图集配合使用的效率最高。如果你将纹理贴图集与精灵批处理配合使用的话，你只要调用一次渲染方法就可以完成纹理贴图集里所有图片的渲染。

“渲染方法调用”（draw call）是把必要的信息传递给图形处理硬件以完成整个或者部份图片渲染的过程。当你使用CCSprite的时候，每生成一个CCSprite节点都会调用一次渲染方法。每一次渲染方法调用导致的系统开销叠加起来会使游戏的帧率大约降低15%或者更多（除非你使用CCSpriteBatchNode，它的作用是作为一个额外的层用于添加多个精灵节点。前提是这些精灵节点使用的是同一个纹理贴图）。

本章的知识将会成为第七章和第八章讨论的横向视差滚屏射击游戏的基础。

CCSpriteBatchNode

每次系统在屏幕上渲染一张贴图时，图形处理硬件必须首先准备渲染，然后渲染图形，最后在完成渲染以后进行清理。上述过程是每一次在启动渲染和结束渲染之间存在的固定系统开销。如果图形处理硬件知道你需要使用同一张纹理贴图渲染一组精灵的话，图形处理硬件将只需要为这一组精灵执行一次准备，渲染，最后清理的过程了。

图6-1展示了这种批量渲染。你可以看到屏幕上有几百个相同的子弹。如果你逐个渲染它们的话，你的游戏帧率将会下降至少15%。如果你使用CCSpriteBatchNode，你就可以避免帧率的下降。

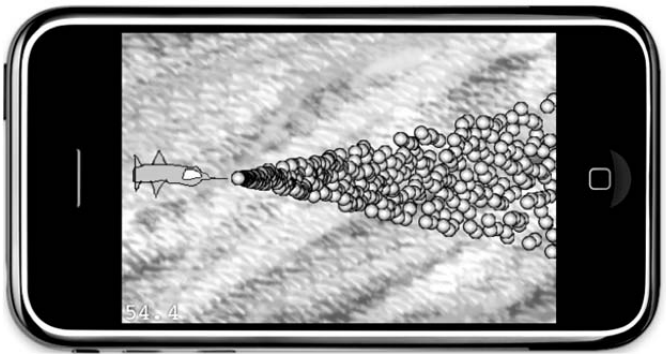

图6-1. 把使用同一张纹理贴图的一组CCSprite节点添加到同一个CCSpriteBatchNode里，比逐个渲染CCSprite要高效很多。

以下是生成CCSprite的常用代码：

```
CCSprite* sprite = [CCSprite spriteWithFile:@"bullet.png"];
[self addChild:sprite];
```

当然，向CCSpriteBatchNode添加一个CCSprite节点不会有任何效率上的提高，所以如**列表6-1**代码所示，我把许多使用同一张纹理贴图的精灵节点添加到了同一个CCSpriteBatchNode里：

列表6-1. 创建多个CCSprite节点，将它们添加到同一个CCSpriteBatchNode中以提高渲染速度

```
CCSpriteBatchNode* batch = [CCSpriteBatchNode batchNodeWithFile:@"bullet.png"];
[self addChild:batch];

for (int i = 0; i < 100; i++)
{
        CCSprite* sprite = [CCSprite spriteWithFile:@"bullet.png"];
        [batch addChild:bullet];
}
```

CCSpriteBatchNode的作用很像CCLayer，因为它本身并不显示在屏幕上。不过你只能把CCSprite加入CCSpriteBatchNode。在**列表6-1**中，CCSpriteBatchNode将一个图片文件名作为参数，使用这个参数的原因是所有被添加进CCSpriteBatchNode的CCSprite节点都必须使用同一个图片文件。如果你没有在CCSprite中使用相同的图片，你将会在调试窗口中得到以下报错信息：

```
SpriteBatches[13879:207] *** Terminating app due to uncaught exception
'NSInternalInconsistencyException', reason: 'CCSprite is not using the same texture id'
```

（**注**：CCSprite is not using the same texture id 的意思是：CCSprite没有使用相同的纹理贴图id）

什么时候应该使用CCSpriteBatchNode

当你需要显示两个或者更多个相同的CCSprite节点时，你可以使用CCSpriteBatchNode。组合在一起的CCSprite节点越多，使用CCSpriteBatchNode得到的效果提升就越大。

不过也有一些限制。因为所有的CCSprite节点都添加到同一个CCSpriteBatchNode中，所以所有CCSprite节点都会使用相同的z-order（深度）

来渲染。如果你的游戏要求子弹在敌人的前面和后面同时飞过的话，你就必须使用两个CCSpriteBatchNode节点，因为只有这样你才能使用具有不同z-order的子弹精灵。

另一个缺点是所有添加到同一个CCSpriteBatchNode中的CCSprite节点必须使用同一个纹理贴图。不过那也意味着CCSpriteBatchNode非常适合使用纹理贴图集。使用纹理贴图集的时候，你不仅仅局限于渲染一张图片；你可以把不同的图片放到同一个纹理贴图集中，然后利用CCSpriteBatchNode将所有图片渲染出来，以提高渲染速度。

如果把纹理贴图集和CCSpriteBatchNode配合使用的话，之前讨论的z-order渲染问题就变得不那么明显了，因为你可以在单个CCSprite节点里指定不同的z-order值。如果你可以把所有在游戏中用到的图片都放到同一张纹理贴图集里的话，你在游戏中使用一个CCSpriteBatchNode就够了（虽然这种情况极少发生）。

你可以把CCSpriteBatchNode看成CCLayer，唯一的区别是：CCSpriteBatchNode只接受使用同一张纹理贴图的CCSprite节点。有了这样的认识以后，我相信你会发现很多可以用到CCSpriteBatchNode的地方。

演示项目

我制作了4个演示项目用于演示CCSpriteBatchNode的用处。项目名是：Sprite01，Sprite02， Sprite03和Sprite04。这是我们在讨论第七章的滚屏射击游戏之前的第一个步骤。因为开发者们通常会首先使用CCSprite节点，然后通过使用CCSpriteBatchNode来提高代码质量，所以我用这些演示项目展示了这个演变过程。

这4个项目使用了两个类，它们是Ship类和Bullet类。它们都继承自CCSprite类。

一个通常会犯的致命错误

Sprite01项目展示了一个新的Objective-C开发者通常会犯的错误。你很容易犯这个错误，但是又很难找到它。请查看一下**列表6-2**中的代码，你能看到哪里出错了吗？

列表6-2. 一个通常在继承CCSprite（或者其它类）时会犯的致命错误

```
-(id) init
{
	if ((self = [super initWithFile:@"ship.png"]))
	{
		[self scheduleUpdate];
	}

	return self;
}
```

上述代码的问题是：-(id)init这个方法是默认的初始化方法，它最终会被任何特殊的初始化方法（比如initWithFile）所调用。因为上述代码又调用了

[super initWithFile:..]这个特殊的初始化方法，最终产生了一个死循环。

解决的方法很简单。你只要像**列表6-3**所示那样把初始化方法换一个名字（只要不是-(id)init）就可以了。

列表6-3. 解决**列表6-2**中的死循环

```
-(id) initWithShipImage
{
	if ((self = [super initWithFile:@"ship.png"]))
	{
		[self scheduleUpdate];
	}
	return self;
}
```

注意：通常，在默认的初始化方法-(Id)init中，不应该调用除了[super init]以外的方法。如果你必须在类的初始化方法中调用像[super initWith…]这样的方法，你应该将初始化方法命名为类似于-(id)initWith…这样的格式（比如像**列表6-3**中的-(id)initWithShipImage）。

不使用精灵批处理生成子弹

Sprites02项目使用了CCSprite来生成所有的子弹。在**列表6-4**的代码中，飞船在每一帧都会射出一些子弹：

列表6-4. 飞船射出子弹

```
-(void) update:(ccTime)delta
{
	// 持续地生成子弹
	Bullet* bullet = [Bullet bulletWithShip:self];

	// 把子弹添加到飞船的父类作为子节点
	[[self parent] addChild:bullet z:1 tag:GameSceneNodeTagBullet];
}
```

把子弹添加到飞船父类的原因是为了防止飞行的子弹同飞船移动的轨迹一样。如果子弹作为子节点添加到飞船节点中的话，所有子弹的位置将同飞船保持相对静止：也就是子弹离开飞船一段距离，飞船移动的话它们也移动，飞船停止，子弹也停止。子弹和飞船就像是黏在一起似的。

列表6-5是子弹用于更新它们自身的位置和在某个时间点将自己从内存中清除的代码。虽然已经跑到屏幕外面的精灵不会在屏幕上被渲染出来，但是它们依然消耗内存和CPU，所以有必要把已经离开屏幕的精灵清除掉。我们通过检查子弹的x轴位置是否大于屏幕的右边界位置来判断子弹是否已经飞出屏幕边界：

列表6-5. 移动和清除飞出屏幕的子弹

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

```
        self.position = ccpAdd(self.position, velocity);
        if (self.position.x > outsideScreen)
        {
                [self removeFromParentAndCleanup:YES];
        }
}
```

子弹位置的更新方法是：将子弹的速度矢量（速率）值与它的当前位置相加得到新的位置，将这个新位置信息赋值给self.position属性来更新子弹的位置。速度矢量（速率）决定着每一帧调用更新方法时，子弹将会移动的像素值。这比使用CCMoveTo或者CCMoveBy动作要有效率的多，因为我们可以避免由使用动作带来的一些问题。由于动作是基于时间来运行的，所以当飞船越靠近屏幕的右侧边界，移动动作会导致子弹飞行变慢（因为在相同的时间内，与之前相比，子弹将会飞行的距离变短了）。

应用CCSpriteBatchNode

在Sprites03项目中，我们使用CCSpriteBatchNode来生成子弹。因为子弹不应该被添加到Ship类之中，所以我把子弹添加到了GameScene中。因为Ship类需要用到子弹，但是由于Ship类不能直接访问GameScene类，所以我添加了一个“伪单例”的访问器：sharedGameScene，以允许飞船访问CCSpriteBatchNode。代码如**列表6-6**所示：

列表6-6. 在GameScene类中添加用于生成子弹的CCSpriteBatchNode和便于Ship类访问的“伪单例”访问器：

```
static GameScene* instanceOfGameScene;
+(GameScene*) sharedGameScene
{
        NSAssert(instanceOfGameScene != nil, @" instance not yet initialized!");
        return instanceOfGameScene;
}

-(id) init
{
        if ((self = [super init]))
        {
                instanceOfGameScene = self;
                ...
                CCSpriteBatchNode* batch = [CCSpriteBatchNode
                                                        batchNodeWithFile:@"bullet.png"];
                [self addChild:batch z:1 tag:GameSceneNodeTagBulletSpriteBatch];
        }
        return self;
}
```

```
-(void) dealloc
{
	instanceOfGameScene = nil;
	[super dealloc];
}

-(CCSpriteBatchNode*) bulletSpriteBatch
{
	CCNode* node = [self getChildByTag:GameSceneNodeTagBulletSpriteBatch];
	NSAssert([node isKindOfClass:[CCSpriteBatchNode class]], @"not a SpriteBatch");
	return (CCSpriteBatchNode*)node;
}
```

注意：我知道不是每个人都喜欢上述代码中的单例和增加的 bulletSpriteBatch方法。为什么我不是在Ship类的初始化方法中或者通过属性将CCSpriteBatchNode作为一个指针传递给Ship类呢？

其中一个原因是Ship并不拥有子弹的批处理精灵，因此它不应该保留对此精灵的引用。更重要的是，如果Ship类保留了批处理精灵，如果你不小心处理的话，可能整个场景都不会从内存中被释放。一个节点不应该保留任何对不属于它的节点的引用。

现在，Ship类可以通过使用sharedGameScene和bulletSpriteBatch方法直接向批处理精灵添加子弹了。代码如**列表6-7**所示：

列表6-7. Ship类使用sharedGameScene和bulletSpriteBatch方法添加子弹

```
-(void) update:(ccTime)delta
{
	Bullet* bullet = [Bullet bulletWithShip:self];
	[[[GameScene sharedGameScene] bulletSpriteBatch] addChild:bullet z:1
	                                                      tag:GameSceneNodeTagBulle];
}
```

代码优化

既然我们要优化代码，我们可以去掉那些由Bullet类带来的不必要的内存分配和释放。你可以通过事先在GameScene中生成一定数量的子弹，来避免在游戏运行过程中进行子弹生成过程中的内存分配和释放。**列表6-8**展示了在Sprites04项目中修改过的GameScene类的init方法：

列表6-8. 事先生成一定数量的子弹精灵以避免在游戏过程中进行不必要的内存分配和释放

```
CCSpriteBatchNode* batch = [CCSpriteBatchNode batchNodeWithFile:@"bullet.png"];
[self addChild:batch z:1 tag:GameSceneNodeTagBulletSpriteBatch];

// 事先生成一定数量的子弹，便于以后随时使用
for (int i = 0; i < 400; i++)
{
	Bullet* bullet = [Bullet bullet];
	bullet.visible = NO;
	[batch addChild:bullet];
}
```

因为在此阶段还用不到这些子弹，所以它们都被设为隐形。**列表6-9**的代码是在

GameScene类中添加的一个新方法，使用这个方法可以通过依次激活隐形的子弹让飞船进行射击。现在，飞船射击通过GameScene类来进行，因为GameScene类包含了用于子弹的CCSpriteBatchNode。一旦一个隐形的子弹被选中，它就会自己射击出去。

列表6-9. 飞船射击现在通过GameScene类来进行

```
-(void) shootBulletFromShip:(Ship*)ship
{
	CCArray* bullets = [self.bulletSpriteBatch children];

	CCNode* node = [bullets objectAtIndex:nextInactiveBullet];
	NSAssert([node isKindOfClass:[Bullet class]], @"not a bullet!");

	Bullet* bullet = (Bullet*)node;
	[bullet shootBulletFromShip:ship];

	nextInactiveBullet++;
	if (nextInactiveBullet >= [bullets count])
	{
		nextInactiveBullet = 0;
	}
}
```

nextInactiveBullet计数器用于跟踪批处理精灵中的子弹数组的索引。每一次射击以后，此计数器就会加一。所有子弹射出以后，计数器归零。

列表6-10中 Bullet 类的射击方法（shootBulletFromShip:）只负责运行一些必要的步骤重新初始化子弹，包括重新预约子弹的更新方法。子弹被设为可视，它的位置和速度也被重置。一旦子弹飞出屏幕边界，它将再次被设为隐形。

列表6-10. Bullet类用于射击的方法重新初始化了子弹

```
-(void) shootBulletFromShip:(Ship*)ship
{
	float spread = (CCRANDOM_0_1() - 0.5f) * 0.5f;
	velocity = CGPointMake(1, spread);

	outsideScreen = [[CCDirector sharedDirector] winSize].width;

	self.position = CGPointMake(ship.position.x + ship.contentSize.width * 0.5f, ship.position.y);

	self.visible = YES;

	[self scheduleUpdate];
}

-(void) update:(ccTime)delta
{
	self.position = ccpAdd(self.position, velocity);

	if (self.position.x > outsideScreen)
	{
		self.visible = NO;
		[self unscheduleAllSelectors];
	}
}
```

不使用CCSpriteBatchNode运行精灵动画

接下去我们讨论如何运行精灵动画（Sprite Animation）。因为你可以将所有

的动画帧放在同一张纹理贴图集里以节省内存，所以这是另一个使用CCSpriteBatchNode的好地方。首先，我们来看一下不使用CCSpriteBatchNode和纹理贴图集运行精灵动画的代码。如**列表6-11**中Sprites05项目的代码所示，你可以看到我们要用到的代码挺多的。之后我将会展示使用CCSpriteBatchNode配合纹理贴图集达到相同动画效果的代码。

列表6-11. 不使用纹理贴图集将动画添加到Ship类中需要不少的代码

```
// 将飞船的动画帧作为贴图加载，同时生成精灵动画帧
NSMutableArray* frames = [NSMutableArray arrayWithCapacity:5];

for (int i = 0; i < 5; i++)
{
	// 为动画帧生成一个贴图
	NSString* file = [NSString stringWithFormat:@"ship-anim%i.png", i];
	CCTexture2D* texture = [[CCTextureCache sharedTextureCache] addImage:file];

	// 整个贴图被用作动画帧
	CGSize texSize = [texture contentSize];
	CGRect texRect = CGRectMake(0, 0, texSize.width, texSize.height);

	// 从贴图生成精灵动画帧
	CCSpriteFrame* frame = [CCSpriteFrame frameWithTexture:texture rect:texRect
                                                    offset:CGPointZero];
	[frames addObject:frame];
}

// 使用所有生成的精灵动画帧生成一个动画对象
CCAnimation* anim = [CCAnimation animationWithName:@"move" delay:0.08f frames:frames];

// 使用CCAnimate运行动画
// 使用CCRepeatForever进行动画循环
CCAnimate* animate = [CCAnimate actionWithAnimation:anim];
CCRepeatForever* repeat = [CCRepeatForever actionWithAction:animate];
[self runAction:repeat];
```

难道上述所有代码都只是为了生成一个只有五帧的精灵动画？这次我将从代码的结束处开始往前进行解释。最后一个步骤是使用CCAnimate动作播放动画。我们也使用了CCRepeatForever进行动画循环。

CCAnimate动作使用了一个CCAnimation对象（它是一个包含所有动画帧的容器），定义了每一帧之间的延迟，同时也给了这个动画一个命名。动画的名称很有用，因为你可以用这个名称来存储CCSprite节点里面的动画。如**列表6-12**所示，你可以通过动画名称来访问部份动画：

列表6-12. CCSprite类可用于存储动画。之后你可以通过动画名称来获取相对应的动画

```
CCAnimation* anim = [CCAnimation animationWithName:@"move" delay:1 frames:frames];

// 将动画储存在CCSprite节点中
[mySprite addAnimation:anim];
```

```
// 在之后的某个时间：你可以通过动画名称从CCSprite节点中获取相应的动画
CCAnimation* moveAnim = [mySprite animationByName:@"move"];
```

回到**列表6-11**的代码。请注意那个 for 循环，那里的逻辑比较复杂。CCAnimation类必须使用一个包含CCSpriteFrame对象的NSArray数组来初始化。“精灵动画帧”（sprite frame）只包含了对一个贴图的引用。这个引用包含的信息是一个定义了贴图大小的四方形和这个四方形在此贴图中的位移。在我们的例子里我想显示整张图片，所以位置的位移是0，四方形贴图的大小等同于贴图的contentSize属性-简单点说就是，实际图片的大小包含在贴图之中。

CCSpriteFrame不能直接接受图片作为参数，它仅接受CCTexture2D对象。我们使用CCTextureCache单例的addImage方法来生成CCTexture2D对象（这个方法通常用于将图片作为贴图预先加载进内存，这样的话就不需要生成CCSprite或者其它对象了）。图片文件通过使用NSString的stringWithFormat方法来生成。这样我就可以把循环变量 i 添加到文件名后面，而不必手动写进所有5个文件名了。

我们再简要重述一下上面的过程。从上到下。以下是用于生成和播放精灵动画的过程：

1. 生成一个NSMutableArray数组。
2. 循环处理每一个动画帧：
   a. 为每一个图片生成一个CCTexture2D节点
   b. 利用CCTexture2D节点生成一个CCSpriteFrame
   c. 把生成的CCSpriteFrame添加到NSMutableArray中
3. 利用NSMutableArray中的动画帧生成一个CCAnimation节点
4. （可选）把CCAnimation添加到一个CCSprite中
5. 使用CCAnimate动作播放动画

嘘！请保持冷静。其实你并不需要使用上述方式来播放动画。如果你把所有的动画帧放入同一个纹理贴图集里的话，你可以使用更简洁的代码来播放动画，同时播放效率也更高。我们可以进一步将所有的代码封装成一个帮助方法（helper method），只要你保持相同的文件命名规则即可。你可以在以后随时重用这个方法。

使用类别封装动画的帮助方法（helper method）

因为用于生成动画帧和动画的代码对所有动画都是通用的，所以我们可以把这些代码封装成一个帮助方法（helper method）。我在Sprite05_WithAnimHelper项目中已经这样做了。我没有使用静态方法，而是用了Objective-C的类别（category）功能来扩展CCAnimation类。通过类别，你可以给已有的类添加方法，而无需修改原先类里的代码。不过你只能添加方法，不能添加成员变量。以下代码是CCAnimation类别的头文件中的代码，我命名它为Helper：

```
@interface CCAnimation (Helper)
+(CCAnimation*) animationWithFile:(NSString*)name frameCount:(int)frameCount
                                                    delay:(float)delay;
@end
```

@interface中的类别使用了和需要扩展的类一样的名字，类名字后面的双括号中是类别的名称。类别的名称和变量名一样，不允许使用空格或其它不允许在变量名中使用的字符，比如标点符号。此外，类别中不允许添加和使用成员变量。

在实现代码中（@implementation），CCAnimation的类别使用了与头文件@interface相同的方式：类别名称添加在类名称后面，放在双括号中。剩下的事情就和定义一个普通的方法无异了。在我们的程序里，方法名是animationWithFile，它需要三个参数：文件名，帧数和动画延迟时间。

```
@implementation CCAnimation (Helper)
// 使用单个文件生成动画
+(CCAnimation*) animationWithFile:(NSString*)name frameCount:(int)frameCount
delay:(float)delay
{
        // 把动画帧作为贴图进行加载，然后生成精灵动画帧
        NSMutableArray* frames = [NSMutableArray arrayWithCapacity:frameCount];

        for (int i = 0; i < frameCount; i++)
        {
                // 假设所有动画帧文件的名字是”nameX.png”
                NSString* file = [NSString stringWithFormat:@"%@%i.png", name, i];
                CCTexture2D* texture = [[CCTextureCache sharedTextureCache]
                                                                addImage:file];

                // 假设总是使用整个动画帧文件
                CGSize texSize = texture.contentSize;
                CGRect texRect = CGRectMake(0, 0, texSize.width, texSize.height);
                CCSpriteFrame* frame = [CCSpriteFrame  frameWithTexture:texture
                                                        rect:texRect offset:CGPointZero];

                [frames addObject:frame];
        }
        // 使用所有的精灵动画帧，返回一个动画对象
        return [CCAnimation animationWithName:name delay:delay frames:frames];
}

@end
```

飞船动画帧文件的命名规则是“ship-anim”加上从0开始的数字，结尾是.png后缀名。这样，五帧动画的文件名将从ship-anim0.png开始，一直到ship-anim4.png结束。如果你都用上述命名规则生成你的动画帧文件的话，你可以将CCAnimation的这个类别方法应用到你的所有动画中去。

技巧：我注意到很多开发人员和设计师有个不好的文件命名习惯。他们喜欢在

数字之前加上一些0。比如把文件以如下规则命名：my-anim001到my-anim024。我想这个习惯起源于之前的操作系统。那时候的操作系统不能够操作自然排序，如果你用连续的数字命名文件的话，这些文件的排序将会是错误的。不过那些日子早就已经过去了，如果你现在还有使用前置0的习惯，程序员在使用你生成的文件时会很头痛。因为如果在一个循环里使用这些文件的话，程序员需要考虑数字之前存在多少个0。虽然iOS有个很方便的格式化方式：%03i （它可以确保生成的数字至少是三位。）但是我想最好的方式是直接使用连续的数字，没有必要再在数字之前添加几个0了。

上述类别中定义的方法可以大大简化使用单独文件生成动画的过程：

```
CCAnimation* anim = [CCAnimation animationWithFile:@"ship-anim" frameCount:5
                                                                    delay:0.08f];
```

上述代码把原先的九行代码缩减到了一行。对于文件名，你只需要把文件的基础名称作为参数传递进去即可 - 在我们的例子中是“ship-anim”。类别方法会根据frameCount变量动态地给文件基础名称添加连续的数字和.png后缀名。传入方法的文件基础名称也被用作动画名称，这样你就不用单独记忆动画名称了。之前我将动画命名为“move”，现在则是“ship-anim” - 与动画帧文件名相同。现在你可以通过文件基础名称来访问储存在CCSprite中的动画对象了：

```
[shipSprite addAnimation:anim];
CCAnimation* shipAnim = [shipSprite animationByName:@”ship-anim”];
```

使用animationWithFile这个方法需要满足两个前提：

1. 动画帧文件的名字带有从0开始计数的连续数字。
2. 动画帧文件必须是.png文件。

当然，你也可以改变这个命名规则。例如，你可能喜欢从1开始计数。如果是那样的话，你就需要把for循环里命名部份的 i 变量改成 i+1。我们的原则是你要坚持使用同一个命名规则，这样代码出错的几率就小。

通过上述讨论，你可以学习到三件事：

1. 通过定义自己的方法来封装常用的代码。
2. 使用Objective-C的类别（Category）给已存在的类添加额外的方法。
3. 定义资源文件（比如动画帧文件）时要使用命名规则。

制作纹理贴图集（Texture Atlases）

“纹理贴图集”可以帮助你节约宝贵的内存资源，也可以帮助提高精灵的渲染速度。因为一张纹理贴图集其实就是一张大图片，你可以使用CCSpriteBatchNode一次性渲染所有它所包含的图片，从而降低内存使用量和提高渲染效率。

什么是纹理贴图集

到目前为止，我只是简单地给精灵加载需要用到的图片文件。在内部，加载的图片变成了精灵的贴图，而此贴图则包含了具体的图片。但是精灵贴图的大小尺寸必须满足“2的n次方规则”，例如1024x128或者256x512像素。如果贴图

尺寸不符合上述规定的话，系统会自动调节它们的大小，这样就有可能是占用比图片本身尺寸还要大的图片。比如，140x600像素的图片如果被放到内存中，刚开始这些浪费掉的内存并不会导致很大问题，但是如果你将它们一个个加载进单独的贴图文件中时，这个浪费的效果就明显了。

这也是为什么要用纹理贴图集的原因了。纹理贴图集只是一张包含多个图片的大图片，同时这张大图片满足“2的n次方规则”。每一张包含于纹理贴图集中的图片都有一个精灵帧（sprite frame），这些帧用于定义各个图片在纹理贴图集中所处的各个长方形区域。换句话说，精灵帧就是一个CGRect结构，此结构定义了各个图片在纹理贴图集上所处的位置。这些精灵帧保存在一个单独的.plist文件中，cocos2d可以利用这个文件渲染纹理贴图集中指定的图片。

介绍Zwoptex工具

如果不使用Zwoptex这个工具，靠手工来制作纹理贴图集和记录各个精灵帧位置的话，那将是一项浩大的工程。如图6-2所示，Zwoptex是一个2D精灵包装工具。目前的桌面版本售价是15美元。你可以通过以下网址下载一个可以使用7天的试用版：http://zwoptexapp.com

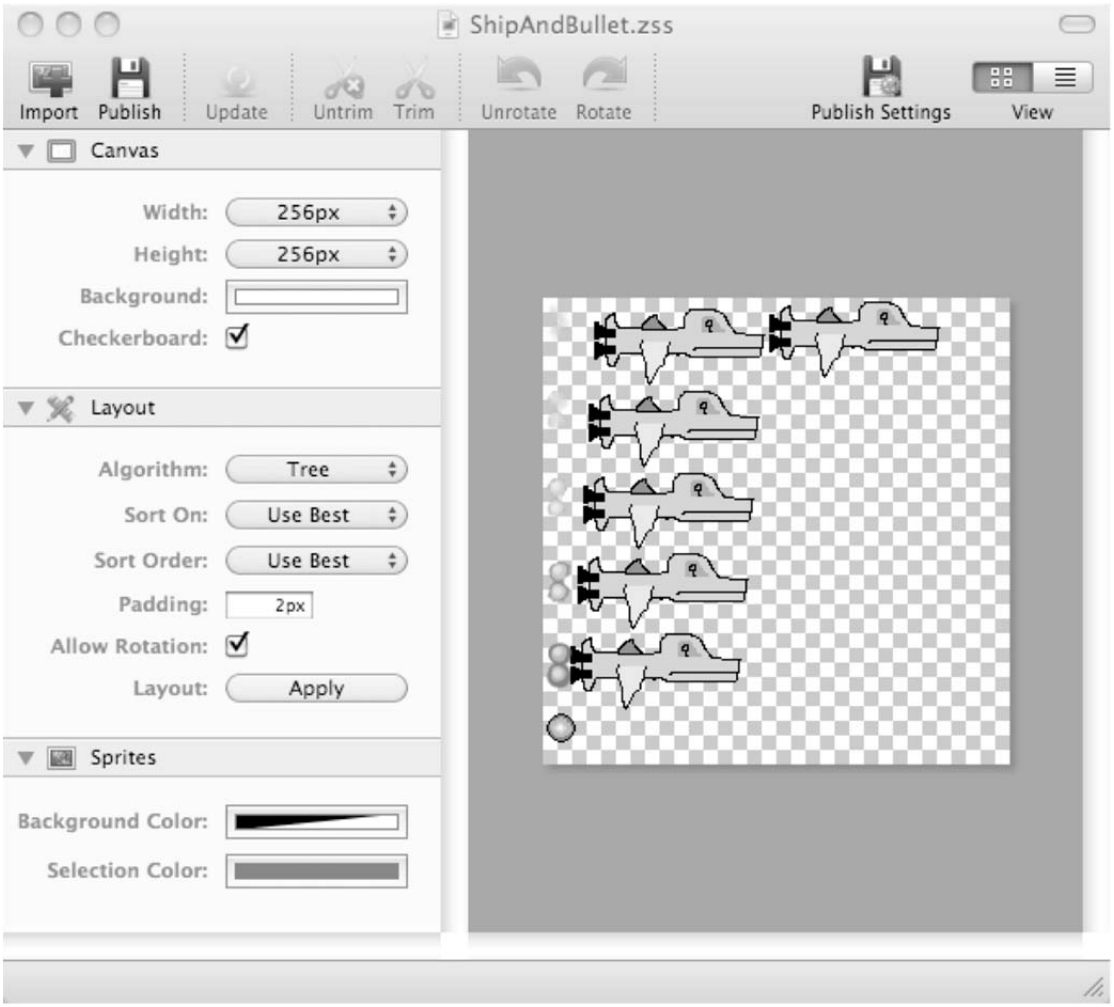

图6-2. Zwoptex桌面版。我用它把飞船动画帧制作成一个纹理贴图集。

如果你不需要最新的功能（比如生成2048x2048像素的贴图文件，自动更新精灵

和通过旋转精灵使它们更好地整合进同一个贴图中）的话，你也可以使用网上的Flash版本（图6-3）。Flash版本的网址是：http://zwoptexapp.com/flashversion

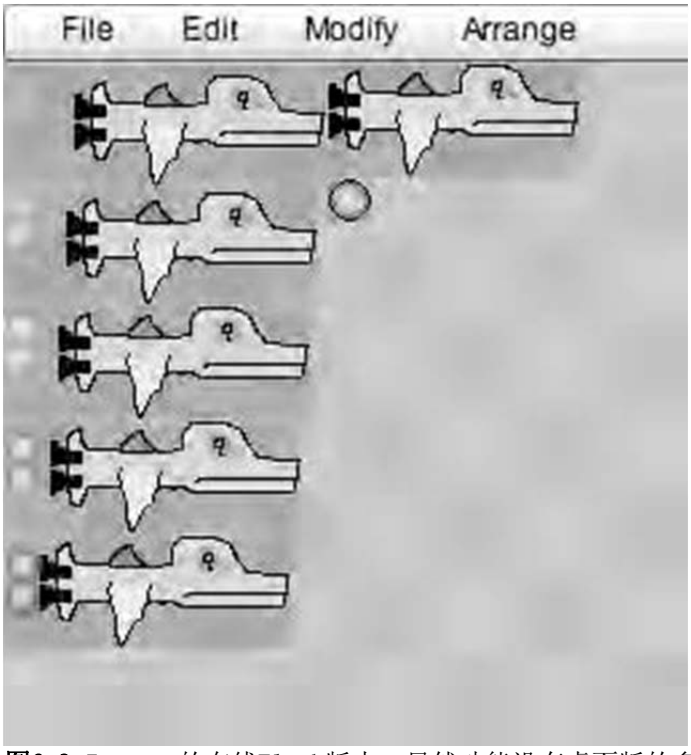

图6-3. Zwoptex的在线Flash版本。虽然功能没有桌面版的多，但它是免费的。

我们在本章专注于使用Zwoptext桌面版，但是你也可以使用在线的Flash版本生成我们需要用到的纹理贴图集。

使用Zwoptex桌面版生成纹理贴图集

如图6-4所示，Zwoptex的使用方法非常直接和简单。需要花时间的是通过调节各种设置以得到最优化的纹理贴图集。

图6-4. Zwoptex的使用方法非常直接和简单

首先你必须添加需要加入纹理贴图集的图片。你可以在以后的任何时间添加新的图片或者删除已有的图片。点击Import按钮，或者从File菜单项选择Import Sprites，然后选择一个或多个图片文件进行导入。Zwoptex可以导入大多数图片格式。在我们的例子里，我导入了所有的飞船图片和动画帧图片，还有子弹图片。你可以在Sprites06项目的Resource文件夹中找到这些图片。

图片导入完成以后，你会看到所有的图片都堆积在左上角的地方。首先你需要做的是点击Layout面板上的Apply按钮将所有图片应用目前的设定进行排布。有些情况下你可以通过试验性地改变设置来优化纹理贴图集的排布。不过这是个很费时的事情，而得到的结果通常和默认的差不了多少。所以我的建议是不需要花太多时间在上面。特别是Sort On和Sort Order这两个设置，直接使用Use Best就够了。

你应该试着调整最终画布的大小，点击Apply按钮，看看是不是所有的图片还是可以容纳在同一张纹理贴图集里。我们的目的是生成一张最小尺寸的纹理贴图集，所有的图片都包含在里面，同时图片之间不能产生叠加。

注意：除非你只想为iPhone 3GS，iPad，iPhone 4或者未来的iOS设备开发游戏，否则不要使用2048x2048像素的画布大小设置。以前的设备支持的最大贴图尺寸是1024x1024像素。

当你改变画布尺寸时，需要注意一个很重要的细节。如**图6-5**所示，如果由于画布尺寸太小，空间不足而导致一些图片相互叠加时，你会看到Zwoptext会给叠加在别的图片上的图片添加一个选择框，就像**图6-5**左上角的图片一样。有时候，这种自动添加的选择框可能很难注意到，但是这是个很有用的功能，因为你可以直接将选中的图片删除或者移动。你可以在画布上直接点击和拖动任何图片，进行手动排布。不过我不建议你这样做，因为很容易意外地将图片叠加在一起，而且如果你点击Apply按钮的话，你的手动排布将会丢失。

在**图6-5**中，你可以看到Zwoptex的旋转（rotation）操作不是很成功。因为在点击Apply以后，你可以看到左上角叠加图片的选择框。这说明目前的画布宽度和高度太小，不足以把所有的图片包含进同一个纹理贴图集。

Layout面板上的Allow Rotation（允许旋转）复选框允许Zwoptex将图片旋转90度。有时候这个选项可以将更多的图片包含在同一个纹理贴图集中，特别是那些宽度比高度大很多的图片，或者高度比宽度大很多的图片。

旋转并不会影响图片的显示。cocos2d在显示纹理贴图集中的图片之前，会先把旋转过的图片旋转回正常的方向。

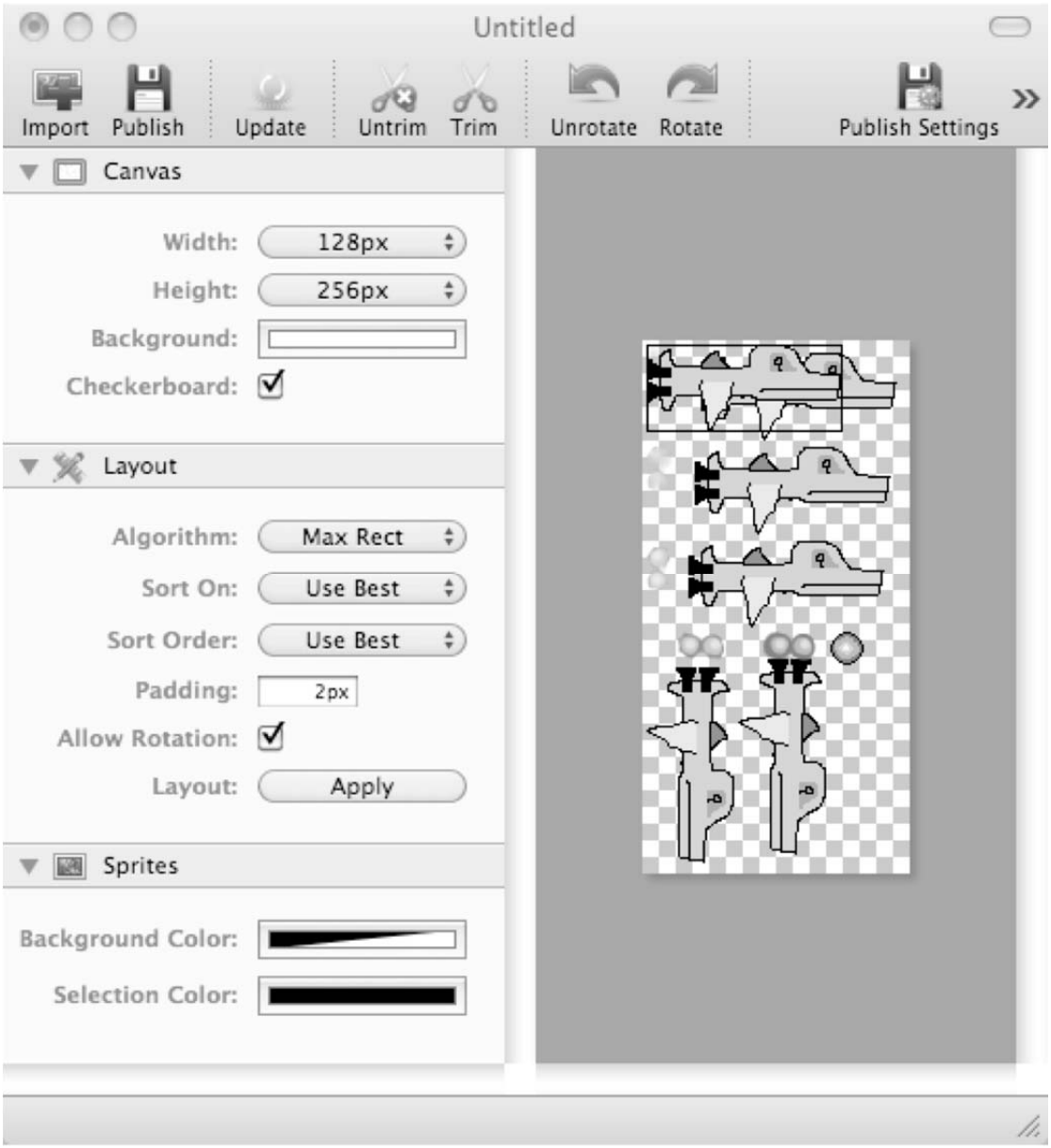

图6-5. Zwoptex通过旋转图片来优化空间的使用

注意：如果你手动旋转了某些图片，然后在Layout面板中点击Apply按钮（Allow Rotation为选中状态），你可能会丢失之前的手动旋转。

在完成导入和排布Sprites06项目Resources文件夹中的图片以后，你会注意到图片之间没有对齐。Zwoptex会把多余的透明边界进行裁剪，以使最终生成的纹理贴图集的尺寸越小越好。不用担心，与旋转过的图片一样，cocos2d会考虑相应的位移，正确显示这些图片的。

Padding设置的作用是用于决定所有图片之间应该存在的距离（以像素计算）。默认情况下，所有纹理贴图集里的图片之间都有两个像素的空隙。如果空隙小于两个像素，图片显示在游戏中时，图片边界周围会出现一些散乱的像素点。这些像素点的数量和颜色取决于纹理贴图集上其它与之相邻的图片颜色。这个问题是由图形处理硬件的贴图过滤方式导致的，唯一的解决方法是让纹理贴图

集上的图片之间保持一定的空隙。

在经过第一轮的图片布局以后，你会看到纹理贴图集上还存在剩余空间。这时你就可以将更多的图片导入进来。我们的目的是生成尽可能少的纹理贴图集，让每个贴图集包含尽可能多的图片，同时尽量少浪费贴图集画布上的空间。但是如果你只有那么几个图片的话，你可以考虑把Zwoptex中的画布（Canvas）尺寸减小。

完成纹理贴图集以后，你可以通过选择File > Save来保存文件。这将把Zwoptex的当前设置保存到一个ZSS文件中。然后点击Publish按钮，Zwoptext会生成cocos2d需要的.png文件和.plist文件，保存和ZSS文件相同的文件夹中。

在cocos2d中使用纹理贴图集

第一件需要做的事是将新生成的纹理贴图集添加到Xcode项目组中的Resources组。不要把Zwoptex的.zss和.zssxml两个文件添加到Xcode项目中去。cocos2d只需要纹理贴图集的.png和.plist文件。列表6-13的代码替代了列表6-11中的代码：

列表6-13. Ship类现在使用纹理贴图集来初始化它的精灵和动画

```
// 加载纹理贴图集的精灵帧（sprite frame）；同时加载相同名字的贴图
CCSpriteFrameCache* frameCache = [CCSpriteFrameCache sharedSpriteFrameCache];
[frameCache addSpriteFramesWithFile:@"ship-and-bullet.plist"];

// 使用精灵帧的名称（文件名）加载飞船的精灵
if ((self = [super initWithSpriteFrameName:@"ship.png"]))
{
	// 加载飞船的动画帧
	NSMutableArray* frames = [NSMutableArray arrayWithCapacity:5];
	for (int i = 0; i < 5; i++)
	{
		NSString* file = [NSString stringWithFormat:@"ship-anim%i.png", i];
		CCSpriteFrame* frame = [frameCache spriteFrameByName:file];
		[frames addObject:frame];
	}

	// 使用所有生成的精灵动画帧创建一个动画对象
	CCAnimation* anim = [CCAnimation animationWithName:@"move" delay:0.08f
                                                frames:frames];

	// 通过使用CCAnimate动作播放动画
	CCAnimate* animate = [CCAnimate actionWithAnimation:anim];
	CCRepeatForever* repeat = [CCRepeatForever actionWithAction:animate];

	[self runAction:repeat];

}
```

在上述代码开始的地方，我把sharedSpriteFrameCache赋值给一个本地变量，赋值的唯一原因是[CCSpriteFrameCache sharedSpriteFrameCache]这个单例访问器代码写起来太长了。

我们通过使用CCSpriteFrameCache的addSpriteFramesWithFile方法来加载纹

理贴图集，需要传入的参数是纹理贴图集的.plist文件。CCSpriteFrameCache会加载精灵帧，同时尝试加载相同命名的贴图，使用的文件后缀名是.png。

注：如果你使用了512x512像素或者更大的纹理贴图集，你应该在游戏开始之前加载贴图。因为加载这么大的贴图是需要一些时间的（在最坏的情况下会导致游戏停滞好几秒钟）。

因为Ship类继承自CCSprite，也因为我想使用纹理贴图集中的ship.png图片，我把Ship类的初始化方法改成了initWithSpriteFrameName。这和通常使用纹理贴图集中的精灵帧名称来初始化CCSprite是一样的：

```
CCSprite* sprite = [CCSprite spriteWithSpriteFrameName:@"ship.png"];
```

如果你加载了好几个纹理贴图集，而只有其中一个包含了我们需要的ship.png，cocos2d还是可以找到正确的贴图进行加载。从本质上说，你通过名字来拿到精灵帧，就像使用图片文件名一样。你并不需要知道具体是哪个贴图集包含了我们需要的图片（除非你使用的是CCSpriteBacthNode，它会要求所有子节点使用相同的贴图文件）。

在**列表6-13**的代码中，我移除了大多数用于初始化CCSpriteFrame对象的代码。我们不再需要加载Texture2D和定义贴图的大小了，而是直接调用[CCSpriteFrame spriteFrameByName:file]来生成相应名字的文件。

其余的代码没有什么变化。但是不幸的是，屏幕上显示的内容发生了变化。虽然飞船图片本身没有变化，但是飞船的位置却移到了靠近屏幕中间的位置。导致这个问题的原因并不明显，不过很容易修复。以下是Ship类在代码修复前的初始化代码：

```
Ship* ship = [Ship ship];
ship.position = CGPointMake(ship.texture.contentSize.width / 2, screenSize.height / 2);
[self addChild:ship];
```

导致上述问题的原因是飞船的位置取决于飞船贴图的contentSize。因为现在飞船使用的贴图实际上是尺寸比原先大很多的纹理贴图集，所以贴图的contentSize也发生了相应的变化。修复此问题的方法是使用飞船精灵的contentSize而不是贴图的contentSize。这个问题只有在使用纹理贴图集时才会发生。以下是修复过的代码：

```
Ship* ship = [Ship ship];
ship.position = CGPointMake(ship.contentSize.width / 2, screenSize.height / 2);
[self addChild:ship];
```

更新CCAnimation帮助类别（Helper Category）

虽然通过使用纹理贴图集可以极大地改善创建CCAnimation的代码，但是将这些代码封装到CCAnimation（Helper）类别中还是很有好处的。毕竟，1行代码还是优于5行代码的。特别是你还要经常用到这5行代码。**列表6-14**展示了扩展了的CCAnimation（Helper）的interface申明，这里添加了一个新的方法-animationWithFrame。

列表6-14. CCAnimation（Helper）类别的@interface

```
@interface CCAnimation (Helper)
+(CCAnimation*) animationWithFile:(NSString*)name frameCount:(int)frameCount
                                                                    delay:(float)delay;
+(CCAnimation*) animationWithFrame:(NSString*)frame frameCount:(int)frameCount
                                                                    delay:(float)delay;
@end
```

AnimationWithFrame和animationWithFile的唯一区别是它以精灵帧作为参数，而不是文件名。如**列表6-15**所示，它的代码实现也和animationWithFile方法非常类似。

列表6-15:animationWithFrame方法

```
// 利用精灵帧生成一个动画对象
+(CCAnimation*) animationWithFrame:(NSString*)frame frameCount:(int)frameCount
delay:(float)delay
{
        // 将飞船的动画帧作为贴图加载，生成一个精灵帧
        NSMutableArray* frames = [NSMutableArray arrayWithCapacity:frameCount];
        for (int i = 0; i < frameCount; i++)
        {
                NSString* file = [NSString stringWithFormat:@"%@%i.png", frame, i];
                CCSpriteFrameCache* frameCache = [CCSpriteFrameCache
                                                        sharedSpriteFrameCache];
                CCSpriteFrame* frame = [frameCache spriteFrameByName:file];
                [frames addObject:frame];
        }

        // 使用所有得到的精灵帧生成一个动画对象，然后返回
        return [CCAnimation animationWithName:frame delay:delay frames:frames];
}
```

现在，你可以使用上述方法，只用一行代码就可以通过纹理贴图集里的精灵帧名称来生成动画了：

```
// 使用所有的精灵帧来生成动画对象
CCAnimation* anim = [CCAnimation animationWithFrame:@"ship-anim" frameCount:5
                                                                    delay:0.08f];
```

使用上述两个方法最大的好处是：你可以在制作动画的过程中，先用animationWithFile做快速的动画测试。等你完全满意动画以后，你才把所有动画帧文件打包成一个纹理贴图集，然后将animationWithFile方法替换成animationWithFrame方法。

你可以在Sprites06_WithAnimHelper项目中找到这份代码。

将所有的图片打包到一个纹理贴图集中

如果可能的话，你应该把所有和游戏相关的图片打包到一个纹理贴图集里，或者尽可能少的纹理贴图集里。使用三个1024x1024像素的纹理贴图集比使用20个小一点的图片要来的更有效率。

对于代码来说，你应该把它们分散到不同的逻辑模块中去。但是对于纹理贴图集，你的目标应该是把尽可能多的图片打包到同一个图片中去，同时尽量减少图片上空白空间的存在。可能你觉得应该把与游戏主角相关的图片放进同一个纹理贴图集；另一个贴图集放置所有的怪物A，B，C和D，还有它们的动画。但是除非你的图片多的不得了，而你又想要在不同的时间加载不同的图片，否则上述方法是没有必要的。不过有一种情况可能适合将图片分开放置到不同的纹理贴图集中，比如一个由很多不同关卡组成的游戏，里面分成很多个世界，每个世界都有不同的敌人。这样的话，你最好把不同的世界和敌人分开放置到不同的纹理贴图集中去。

只要你的游戏图片可以打包成3到4张1024x1024像素的纹理贴图集，你就可以在游戏开始之前就将他们加载进内存。这会占用大约12到16MB的内存作为你的贴图内存。你的游戏代码和其它素材，比如声音文件，不会占用很多内存，所以即使在只有128MB内存的iOS设备上，你也可以一直把这些图片保存在内存中。

一旦你的纹理贴图集数量超过4张，你就需要更好的使用贴图内存的策略了。我们之前讨论过其中一个策略，那就是将游戏图片分割成不同的世界，我们只加载当前所处世界需要的图片。这个方式会在新世界完成加载之前有一定的延迟，我们可以在这里使用第五章讨论过的LoadingScene加载页面来过渡这个延迟。

因为cocos2d会自动生成所有图片的缓存，所以你需要一个方法来卸载不再需要的贴图内存。大多数情况下，你可以依赖cocos2d来帮你卸载：

```
[[CCSpriteFrameCache sharedSpriteFrameCache] removeUnusedSpriteFrames];
[[CCTextureCache sharedTextureCache] removeUnusedTextures];
```

很显然，在有不需要用到的贴图存在时，你才应该调用上述方法。通常你应该在完成场景转换以后才做上述操作，而不是在游戏运行的过程中进行。请记住，在场景转换的过程中，只有在完成新场景初始化以后，之前的场景才会被卸载。这意味着你不能在一个场景的init方法中使用removeUnused这类方法 – 除非你在两个转换的场景之间使用第五章讨论过的LoadingScene，这样的话你应该在转换到新场景之前就删除不再用到的贴图。

如果你想在加载新场景之前完全删除所有内存中的贴图，你应该使用以下方法：

```
[CCSpriteFrameCache purgeSharedSpriteFrameCache];
[CCTextureCache purgeSharedTextureCache];
```

自己动手做

我必须承认，我对自己的涂鸦挺满意的。虽然它们和设计师能给你画出来的差很远，但是我心中拥有那种来自“自己动手做”的态度所带来的满足感。

你可能会想：我有那么多的游戏设计经验，画些素材应该不在话下。但是我要告诉你，我没有什么艺术细胞。我的画可能和一个5岁小孩的画差不多。但是我还是要鼓励你制作自己的素材，包括声音。只是不要在上面花费太多时间-只要制作出足够用于传达你的游戏设计意图的素材就可以了。

在设计游戏的时候，我的素材都是用最简单的方法来制作的，这样可以少花时间在素材制作上面。只要制作的素材可以用于游戏中，表达游戏的意图就可以了。在游戏设计完成以后，你可以请设计师帮你设计游戏中的各种素材，用于替换原有的粗糙素材。

因为你拥有所有自己设计的素材，你可以把它们交给设计师替换成真正的素材。如果设计师只需要替换图片的话，你甚至不用修改游戏的代码。

我使用的免费图片编辑器叫Seashore。你可以通过以下网址下载：http://seashore.sourceforge.net。

对于动画，你可以使用开源软件Pixen。图6-6展示了Pixen的动画编辑器。你可以在这里下载Pixen：http://opensword.org/Pixen。

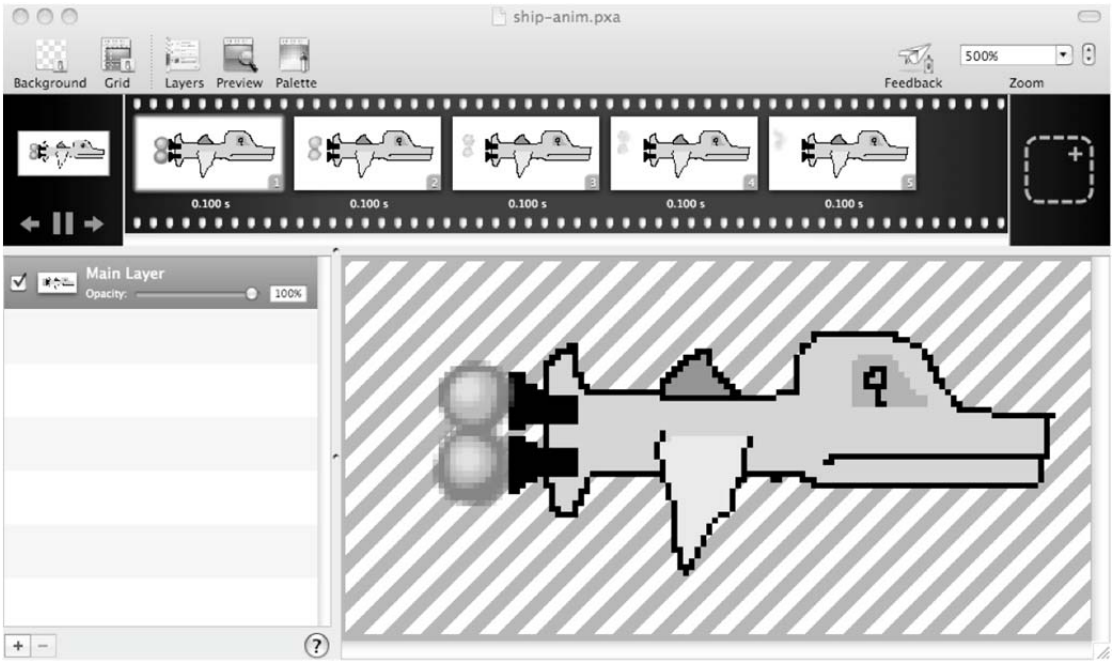

图6-6. 我制作的飞船动画呈现在 Pixen 简单易用但是非常强大的动画编辑器中。

如果你要制作自己的音效的话，你可以使用已经安装在你的Mac上的GarageBand。你只需要用麦克风录制一些噪音，然后花些时间调整声音的设置就可以得到你想要的音效了。你也可以在网上寻找免费或者便宜的声音文件，例如 http://www.soundsnap.com 或类似的网站。

注意：你要小心那些从网上下载的精灵和音频包。有些地方下载的“免费”素材其实并不免费。除非你得到许可，才可以将它们用到你的游戏中去。因为很多下载的“免费”素材是不允许重新发布或者用于商业目的的，比如用于你在

App Store发售的游戏中。

结语

本章我们学习了如何使用同一张贴图与CCSpriteBatchNode配合快速渲染多个精灵。贴图可以是一个单独的图片文件或者纹理贴图集中的一个精灵帧。

本章的开头，在教你如何生成精灵动画之前，我演示了如何通过继承CCSprite以生成动画对象。因为用于生成动画的代码很复杂，所以我把这些代码都封装进了CCAnimation（Helper）类别。

我也演示了如何使用纹理贴图集和解释了为什么要使用。当然，解释纹理贴图集的时候少不了介绍Zwoptext，它是用来生成和修改纹理贴图集的最佳工具。如果不想花钱买这个软件的话，你也可以使用功能少一些的在线Flash版本。

最后，我鼓励你创造自己的图片素材和音频。这有助于你尽快完成游戏的设计的编程，而且也能得到很大的满足感。

我们将在下一章设计编写我们的下一个游戏：横向滚屏射击游戏。