说明：本文翻译自 Apple 官方的《Sprite Kit Programming Guide》，利用 Chome 浏览器的自动翻译功能作初译，然后在一些语句不顺或容易造成误解的地方作局部修正。方便英文不好的开发者查看。如有错漏之处，欢迎大家指出修正。

同时欢迎大家关注我的技术博客 http://blog.csdn.net/it_magician。大家的支持是我最大的动力。

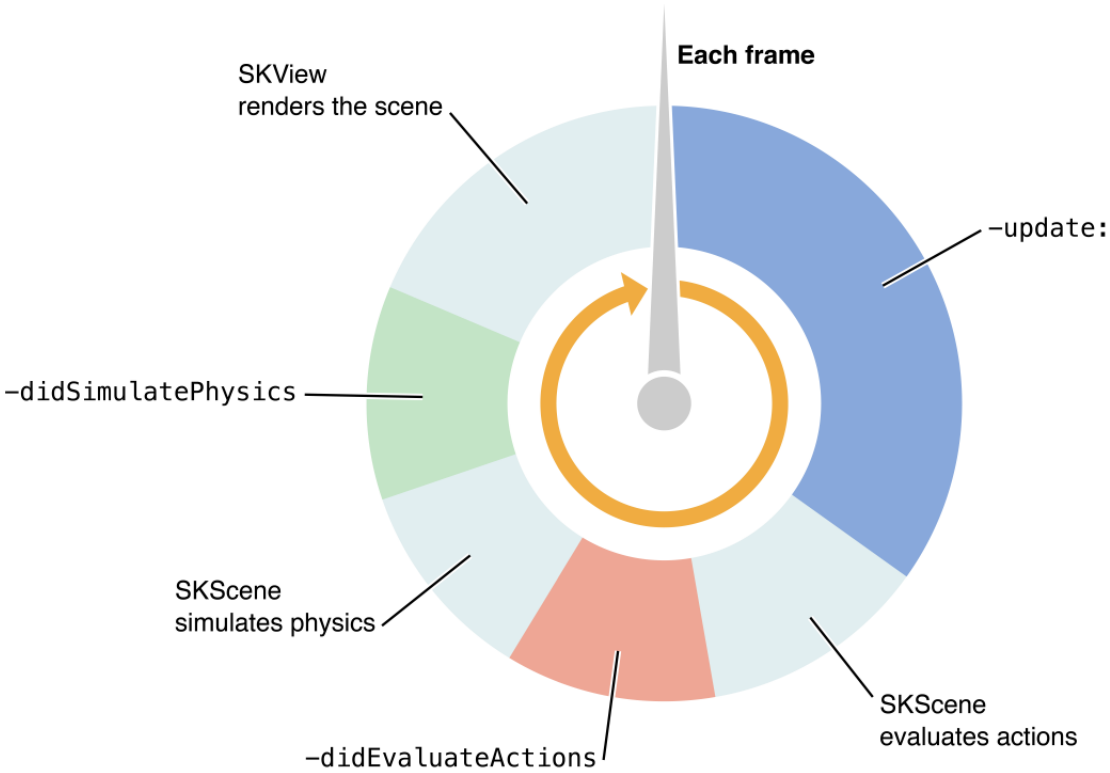
Sprite Kit编程指南

关于Sprite Kit

重要提示： 这是 API 或开发技术的一个初版文档。虽然本文档的技术准确性已被审阅过，但这还不是最终版本。这个苹果的机密信息仅用于适用的苹果开发者计划的注册会员。苹果公司提供这些机密信息来帮助你计划采用本文所述的技术和编程接口。此信息如有变更，根据这份文档实现的软件，应该用最终的操作系统软件和最终文档进行测试。本文档的新版本可能会与 API 或技术的未来种子一起提供。

Sprite Kit 提供了一个图形渲染（rendering）和动画的基础，你可以使用它让任意纹理（textured）图像或精灵动起来。Sprite Kit 采用的是传统的渲染循环，允许在渲染前处理每一帧的内容。你的游戏确定场景的内容，以及这些内容如何在每帧中变化。Sprite Kit 做的工作，就是有效地利用图形硬件来渲染动画的帧。Sprite Kit 优化到本质上允许对动画每一帧的任意修改。

Sprite Kit 还提供了其他对游戏非常有用的功能，包括基本的声音播放支持和物理模拟。此外，Xcode 中提供了内置的 Sprite Kit 支持，可以很容易地创建并在你的应用程序中使用复杂的特效和纹理图册（atlases）。这种框架和工具的组合，使 Sprite Kit 对于游戏和其他需要类似动画的应用程序是一个很好的选择。对于其他类型的用户界面动画，使用 Core Animation 代替。



概览

Sprite Kit 在 iOS 和 OS X 可用。它使用主机设备提供的图形硬件，以高帧速率复合 2D 图像。Sprite Kit 支持多种不同类型的内容，包括：

- 无纹理或有纹理的矩形（精灵）
- 文本
- 任意基于 CGPath 的形状
- 视频

Sprite Kit 还提供了裁剪和其他特效的支持，允许你对全部或者部分内容应用这些效果。你可以在每一帧中活动（animate）或者改变这些元素。你也可以附加物理体到将这些元素，使他们正确地支持武装和碰撞。

通过支持丰富的渲染基础和处理所有低级别的工作来提交 OpenGL 的绘图命令，Sprite Kit 允许你全神贯注解决更高层次的设计问题，并创造伟大的游戏。

在精灵视图内由呈现场景绘制精灵内容

动画和渲染由 SKView 对象执行。你在一个窗口在放置这个视图，然后渲染它的内容。因为它是一个视图，所以它的内容可以结合在视图层次里的其他视图。

你的游戏中的内容会被组织成**场景（scenes）**，用 SKScene 对象代表它们。场景包含精灵和其他要渲染的内容。场景也实现了每帧的逻辑和内容处理。在任何给定的时间内，视图展示一个场景。只要一个场景被呈现出来，它的动画和每帧逻辑会自动执行。

要使用 Sprite Kit 创建一个游戏，你要创建一个或多个的 SKScene 类的子类。例如，你可能会创建单独的场景类，用来分别显示主菜单、游戏画面和游戏结束后显示的内容。你可以很容易地在你的窗口中使用一个单一的 SKView 对象并在不同场景之间进行过渡。

有关章节： “深入 Sprite Kit”，“ 使用场景间过渡”，“ Sprite Kit 最佳实践”

节点树定义出现在一个场景中的内容

SKScene 类实际上是 SKNode 类的后代。节点是所有内容的基石，而场景对象作为一个节点对象树的根节点。场景及其后代决定哪个内容被绘制以及它渲染的方式。

每个节点的位置在它的父节点定义的坐标系中指定。节点的内容的其他属性也适用于它后代的内容。例如，当一个节点是旋转，所有它的后代也跟着旋转。你可以使用节点树构建一个复杂的图像，然后通过调整最上层节点的属性旋转、缩放并融入整个图像。

SKNode 类绘制任何东西，但它对后代应用于它的属性。每一种可绘制内容 由 Sprite Kit 的不同子类表示。其他的节点子类不直接绘制内容，但修改它们后代的行为。例如，你可以在场景中使用一个 SKEffectNode 对象对整个子树应用一个 Core Image 滤镜。通过精确控制节点树的结构，你确定节点的渲染顺序，让你可以在一个场景中布局（layer）复杂的图形效果。

所有节点对象都是响应者（responder）对象，派生（descending）自 UIResponder 或 NSResponder，所以你可以继承任何节点类来创建接受用户输入的新类。视图类自动扩展响应链来包含场景的节点树。

相关章节： “使用精灵”，“ 构建场景”，“ 使用其他节点类型”

纹理保存可复用的图形数据

纹理是用来渲染精灵的共享图像。当你需要对多个精灵应用相同的图像时，总是使用纹理。通常你通过加载存储在你的应用程序 bundle 的图像文件来创建纹理。然而，Sprite Kit 也可以在运行时从包括核心图形图像在内的其他来源为你创建纹理，或者甚至渲染把节点树成纹理。

Sprite Kit 通过处理较低级别的代码需求来加载纹理和并让它们对图形硬件可用，来简化了纹理的管理。纹理管理由 Sprite Kit 自动管理。但是，如果你的游戏中使用了大量的图像，你可以通过控制部分的过程来提高性能。首先，你通过提示 Sprite Kit 纹理很快就需要来做这个。

纹理图册是在你的游戏中一起使用的一组相关的纹理。例如，你可以使用一个纹理图册存储让一个角色活动需要的所有纹理或渲染游戏级别的背景需要的所有瓷砖。Sprite Kit 用纹理图册来提高渲染性能。

相关章节：“使用精灵”

动作在场景中由节点执行

使用**动作（actions）让**场景的内容动起来。每一个动作都是一个对象，由 `SKAction` 类定义。你来告诉节点执行动作。然后，当场景处理动画帧，动作就被执行。有些动作在一帧动画内完成，而另一些在完成前应用变化于多帧动画。动作最常见的用途是改变节点的属性。例如，你可以创建动作来移动、缩放或旋转节点，或使其透明。然而，动作也可以更改节点树、播放声音、甚至是执行自定义代码。

动作是非常有用的，但你也可以组合动作来创建更复杂的效果。你可以创建一*组*同时运行或顺序运行的动作。你可以让动作自动重复。

场景中也能执行自定义的每帧处理。覆盖你的场景子类的方法来执行额外的游戏任务。例如，如果一个节点需要每帧移动，你可能会直接每帧地调整其属性而不是使用一个动作来这样做。

相关章节：“添加动作到节点”，“ 高级场景处理”

添加物理体和联合来在场景中模拟物理

虽然你可以控制场景中的每一个节点的确切位置，你经常想这些节点互相交流、碰撞并在这个过程中告知速度的变化。你可能还需要模拟重力和其他形式的加速度，这些都不在动作系统中处理

的。要做到这一点，你可以创建物理体（`SKPhysicsBody`），并将它们附加到你场景中的节点上。每个物理体由形状、尺寸质量和其他物理特性定义。

当场景中包含物理体，场景就在这些主体上模拟物理。一些势力（forces），如重力和摩擦力，会自动应用。你也可以对物理体调用方法来应用自己的势力。每个主体的加速度和速度会被计算，然后主体彼此碰撞。然后，模拟完成后，相应的节点的位置和旋转的被更新。

你物理体的交互拥有精确的控制。你确定哪些主体被允许相互碰撞 并单独决定哪些交互可以被你的应用程序调用。这些回调允许你勾（hook）到物理模拟中创建其他的游戏逻辑。例如，在一个物理体被另一个物理体击中时，你的游戏可能会销毁一个节点。

场景在一个附加的 `SKPhysicsWorld` 对象上定义了物理模拟的全局特性。你可以使用物理世界定义整个模拟的重力，定义模拟的速度，并在场景中查找物理体。你还可以使用物理世界通过一个联合（`SKPhysicsJoint`）把物理体连接在一起。连接的主体根据联合的类型模拟在一起。

相关章节： “模拟物理”

如何使用本文档

阅读“深入 Sprite Kit”获得实现 Sprite Kit 游戏的一个概述。然后通过其他章节学习关于 Sprite Kit 功能的细节。一些章节包含建议的练习，以帮助你开发你对 Sprite Kit 的理解。学习 Sprite Kit 的最好方法是实践；把一些精灵放到场景中并实验它们！
最后一章，“Sprite Kit 最佳实践”，进入更详细的使用 Sprite Kit 设计游戏。

先决条件

在试图使用 Sprite Kit 创建一个游戏之前，你应该对应用程序开发的基础知识相当熟悉。

- 在 iOS 上，参见 *今天开始开发 iOS 应用程序*。
- 在 OS X 上，参见 *今天开始开发 Mac 应用程序*。

尤其的，你应该熟悉以下概念：

- 使用 Xcode 开发应用程序

- Objective-C 语言，包括块（blocks）支持
- 视图和窗口系统

虽然本指南展示了许多有用的创建游戏的技术，它还不是一个完整的游戏设计或游戏开发指南。

参见

当你需要 Sprite Kit 框架的具体细节时，参见 *SpriteKit 框架参考*。

关于如何使用 Xcode 对 Sprite Kit 的内置支持的信息，参见*纹理图册帮助*和*粒子发射器编辑器指南*。

关于在 `SKSpriteNode` 类的详细说明参见 *Sprite Tour*。

要了解 Sprite Kit 中的物理系统，参见 *SpriteKit 物理碰撞*。

要深入了解基于 Sprite Kit 的游戏参见*代码：Explained Adventure*。

深入Sprite Kit

学习 Sprite Kit 最好的方法是在实践中观察它。此示例创建一对场景和各自的动画内容。通过这个例子，你将学习使用 Sprite Kit 内容的一些基础技术，包括：

- 场景在一个基于 Sprite Kit 的游戏中的角色。
- 如何组织节点树来绘制内容。
- 使用动作让场景内容动起来。
- 如何添加交互到场景。
- 场景之间的过渡。
- 在一个场景里模拟物理。

一旦你完成这个项目，你可以用它来试验其他 Sprite Kit 概念。你可以在这个例子的结尾找到一些建议。

你应该已经熟悉创建 iOS 应用程序之前通过这个项目工作。欲了解更多信息，请参阅*今天开始开发 iOS 应用程序的*。大多数 Sprite Kit 在这个例子中的代码是相同的 OS X。

让我们开始吧

本次练习需要 Xcode 5.0。使用的单一视图的应用程序模板创建一个新的 iOS 应用程序的 Xcode 项目。

在创建项目时，请使用以下值：

- 产品名称：`SpriteWalkthrough`
- ClassPrefix：`Sprite`
- 设备：iPad

添加 Sprite Kit 框架到项目中。

创建你的第一个场景

Sprite Kit 内容被放置在一个窗口中，就像其他可视化内容那样。Sprite Kit 内容由 `SKView` 类渲染呈现。`SKView` 对象渲染的内容称为一个**场景**，它是一个 `SKScene` 对象。场景参与响应链，还有其他使它们适合于游戏的功能。

因为 Sprite Kit 内容由视图对象渲染，你可以在视图层次组合这个视图与其他视图。例如，你可以使用标准的按钮控件，并把它们放在你的 Sprite Kit 视图上面。或者，你可以添加交互到精灵来实现自己的按钮，选择权在你。在这个例子中，稍候你会看到如何实现场景交互。

配置视图控制器来使用Sprite Kit

1. 打开项目的 storyboard。它有一个单一的视图控制器（`SpriteViewController`）。选择视图控制器的 view 对象并把它的类改成 `SKView`。
2. 在视图控制器的实现文件添加一个导入行。

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

3. 实现视图控制器的 `viewDidLoad` 方法来配置视图。

```
- (void)viewDidLoad

{

    [super viewDidLoad];

    SKView * spriteView = (SKView *) self.view;

    spriteView.showsDrawCount = YES;

    spriteView.showsNodeCount = YES;

    spriteView.showsFPS = YES;

}
```

4. 代码开启了描述场景如何渲染视图的诊断信息。最重要的一块信息是帧率（spriteView.showsFPS），你希望你的游戏尽可能在一个恒定的帧率下运行。其他行展示了在视图中显示了多少个节点，以及使用多少绘画传递来渲染内容（越少越好）的详情。

接下来，添加第一个场景。

创建Hello场景

1. 创建一个名为 HelloScene 新类并让它作为 SKScene 类的子类。
2. 在你的视图控制器导入场景的头文件。

```
#import “HelloScene.h”
```

3. 修改视图控制器来创建场景，并在视图中呈现场景。

```
- （void）viewWillAppear:（BOOL）animated

{

   HelloScene *hello = [[HelloScene alloc] initWithSize:CGSizeMake(768,1024)];

   SKView *spriteView =(SKView *)self.view;

   [spriteView presentScene:hello];

}
```

现在，构建并运行项目。该应用程序应该启动并显示一个只有诊断信息的空白屏幕。

将内容添加到场景

当设计一个基于 Sprite Kit 的游戏，你要为你的游戏界面各主要大块(chuck)设计不同的场景类。例如，你可以为主菜单创建一个场景而为游戏创建另一个单独的场景。在这里，你会遵循类似的设计。这第一个场景显示了传统的“Hello World”文本。

大多数情况下，你可以配置一个场景在它被视图首次呈现时的内容。这跟视图控制器只在视图属性被引用时加载他们的视图的方式是类似的。在这个例子中，代码在 didMoveToView: 方法内部，每当场景在视图中显示时该方法会被调用。

在场景中显示Hello文本

1. 添加一个新的属性到场景的实现文件中来跟踪场景是否已创建其内容。

```
@interface HelloScene()

@property BOOL contentCreated;

@end
```

该属性跟踪并不需要向客户端公开的状态，所以，在实现文件中它一个私有接口声明里实现。

2. 实现场景的 didMoveToView: 方法。

```
- (self) didMoveToView:(SKView *)view

{

    if (! self.contentCreated)

    {

        [self createSceneContents];

        self.contentCreated = YES;

    }

}
```

每当视图呈现场景时，didMoveToView: 方法都会被调用。但是，在这种情况下，场景的内容应只在场景第一次呈现时进行配置。因此，这段代码使用先前定义的属性（contentCreated）来跟踪场景的内容是否已经被初始化。

3. 实现场景的 createSceneContents 方法。

```
- (void)createSceneContents

{

    self.backgroundColor = [SKColor blueColor];

    self.scaleMode = SKSceneScaleModeAspectFit;

    [self AddChild: [self newHelloNode];

}
```

场景在绘制它的子元素之前用背景色绘制视图的区域。注意使用 SKColor 类创建 color 对象。事实上，SKColor 不是一个类，它是一个宏，在 iOS 上映射为 UIColor 而在 OS X 上它映射为 NSColor。它的存在是为了使创建跨平台的代码更容易。

场景的缩放（scale）模式决定如何进行缩放以适应视图。在这个例子中，代码缩放视图，以便你可以看到场景的所有内容，如果需要使用宽屏(letterboxing)。

4. 实现场景的 newHelloNode 方法。

```
- （SKLabelNode *）newHelloNode

{

    SKLabelNode * helloNode = [SKLabelNode labelNodeWithFontNamed:@“Chalkduster”];

    @helloNode.text =“Hello, World! ”;

    helloNode.fontSize = 42;

    helloNode.position = CGPointMake(CGRectGetMidX(self.frame), CGRectGetMidY(self.fra

    return helloNode;

}
```

你永远不用编写显式执行绘图命令的代码，而如果你使用 OpenGL ES 或 Quartz 2D 你就需要。在 Sprite Kit 中，你通过创建节点对象并把它们添加到场景中来添加内容。所有绘制必须由 Sprite Kit 中提供的类来执行。你可以自定义这些类的行为来产生许多不同的图形效果。然而，通过控制所有的绘图，Sprite Kit 可以对如何进行绘图应用许多优化。

现在构建并运行该项目。你现在应该看到一个蓝色屏幕上面有“Hello, World！”。现在，你已经学会了绘制 Sprite Kit 内容的所有基础知识。

使用动作让场景动起来

静态文本很友好，但如果文字可以动起来，它会更有趣。大多数的时候，你通过执行**动作（action）**移动场景周围的东西。Sprite Kit 中的大多数动作对一个节点应用变化。创建 action 对象来描述你想要的改变，然后告诉一个节点来运行它。然后，当渲染场景时，动作被执行，在几个帧上发生变化直到它完成。

当用户触摸场景内容，文字动起来然后淡出。

让文本动起来

1. 添加以下代码到 `newHelloNode` 方法：

```
helloName.name = @“helloNode”;
```

所有节点都有一个名称属性，你可以设置它来描述节点。当你想能够在稍后找到它，或当你想构建基于节点名称的行为时，你应该命名一个节点。稍后，你可以搜索树中与名称相匹配的节点。

在这个例子中，你给标签的一个名称以便稍后可以找到它。在实际的游戏中，你可能会得给呈现相同类型的内容的任何节点以相同的名称。例如，如果你的游戏把每个怪物呈现为一个节点，你可能会命名节点为 `monster`。

2. 重载场景类的 `touchesBegan: withEvent` 方法。当场景接收到触摸事件，它查找名为 `helloNode` 的节点，并告诉它要运行一个简短的动画。

所有节点对象都是 iOS 上 UIResponder 或 OS X 上 NSResponder 的 的子类。这意味着你可以创建 Sprite Kit 节点类的子类来添加交互到场景中的任何一个节点。

```
 - (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event

{

    SKNode *helloNode = [self childNodeWithName:@“helloNode”];

    If(helloNode != nil)

    {

        helloNode.name = nil;

        SKAction *moveUp = [SKAction moveByX:0 y:100.0 duration:0.5];

        SKAction *zoom = [SKAction scaleTo:2.0 duration:0.25];

        SKAction *pause = [SKAction waitForDuration:0.5];

        SKAction *fadeAway = SKAction fadeWithDuration:0.25];

        SKAction *remove = [SKAction removeFromParent];

        SKAction * moveSequence = [SKAction sequence:@[moveUp, zoom, pause, fadeAway, r

        [helloNode runAction: moveSequence];

    }

}
```

为了防止节点响应重复按压，代码会清除节点的名称。然后，它构建动作对象来执行各种操作。最后，它组合这些动作创建一个**动作序列**;序列运行时，按顺序执行每个动作。最后，它告诉标签节点执行序列动作。

运行的应用程序。你应该看到像之前那样的文字。在屏幕的底部，节点计数应该是 1。现在，点击视图内部。你应该看到文字动画并淡出。在它淡出后，节点计数应该变为 0，因为节点已从父节点中移除。

场景之间的转换

Sprite Kit 让场景之间的过渡变得很容易。场景之间的过渡时，你可以坚持保留它们，或清除它们。在这个例子中，你将创建第二个场景类，来学习一些其他的游戏行为。“Hello, World！”文字从屏幕上消失时，代码创建一个新的场景并过渡到它。Hello 场景过渡在后会被丢弃。

创建飞船场景

1. 创建一个名为 `SpaceshipScene` 的新类并让它成为 `SKScene` 类的子类。
2. 实现代码来初始化飞船场景的内容。此代码类似于你为 `HelloScene` 类实现的代码。

```
@interface SpaceshipScene()

@property BOOL contentCreated;

@end



@implementation SpaceshipScene

 - (void)didMoveToView:(SKView *)view

{

   if(!self.contentCreated)

   {

       [self createSceneContents];

       self.contentCreated = YES;

   }

}
```

```
- (void)createSceneContents

{

    self.backgroundColor = [SKColor blackColor];

    self.scaleMode = SKSceneScaleModeAspectFit;

}
```

3. 在 HelloScene.m 文件中导入 SpaceshipScene.h 头。

```
#import “SpaceshipScene.h”
```

4. 在 touchesBegan:withEvent 方法中，更改 runAction: 的调用为新的调用 runAction:completion:。实现完成处理来创建并呈现一个新的场景。

```
[helloNode runAction:moveSequence completion:^ {

    SKScene * spaceshipScene = [[SpaceshipScene alloc] initWithSize:self.size];

    SKTransition *doors= [SKTransition doorsOpenVerticalWithDuration:0.5];

    [self.view presentScene:spaceshipScene transition:doors];

}];
```

构建并运行该项目。当你触摸场景内部时，文字淡出，然后在视图过渡到新的场景。你应该看到一个黑色的屏幕。

使用节点构建复杂的内容

新的场景还没有任何内容，所以你准备要添加一个飞船到场景。要构建这个太空飞船，你需要使用多个 SKSpriteNode 对象来创造了飞船和它表面的灯光。每个精灵节点都将执行动作。

精灵节点是在一个 Sprite Kit 应用程序中最常见用于创建内容的类。他们可以绘制无纹理或纹理的矩形。在这个例子中，你要使用无纹理对象。稍后，这些占位符（placeholder）可以很容易地用纹理精灵进行替换，而不改变它们的行为。在实际的游戏中，你可能需要几十个或上百个节点来创建你的游戏的可视化内容。但是，从本质上说，那些精灵将使用与这个简单的例子相同的技术。

虽然你可以直接添加所有三个精灵到场景，但这并不是 Sprite Kit 的方式。闪烁的灯光是飞船的一部分！如果飞船移动，灯光应该和它一起移动。解决的办法是使飞船节点成为它们的父节点，同样地场景将是飞船的父节点。光的坐标将要相对于父节点的位置来指定，而父节点是在子精灵图像的中心。

添加飞船

1. 在 `SpaceshipScene.m` 中，添加代码到 `createSceneContents` 方法来创建飞船。

```
SKSpriteNode *spaceship = [self newSpaceship];

spaceship.position = CGPointMake(CGRectGetMidX(self.frame),CGRectGetMidY(self.frame)-

[self addChild:spaceship];
```

2. 实现 `newSpaceship` 的方法。

```
 - (SKSpriteNode *)newSpaceship

{

   SKSpriteNode *hull= [[SKSpriteNode alloc] initWithColor:[SKColor grayColor] size:



   SKAction *hover= [SKAction sequence:@[

                     [SKAction waitForDuration:1.0]

                     [SKAction moveByX:100 y:50.0 duration:1.0]
```

```
                    [SKAction waitForDuration:1.0]

                    [SKAction moveByX:-100.0 y:-50 duration:1.0]];

    [hull runAction:[SKAction repeatActionForever:hover];



    return hull;}
```

此方法创建飞船的船体，并添加了一个简短的动画。需要注意的是引入了一种新的动作。一个重复的动作不断地重复的传递给它的动作。在这种情况下，序列一直重复。

现在构建并运行应用程序来看当前的行为，你应该看到一个矩形。

在构建复杂的有孩子的节点时，把用来在构造方法后面或者甚至是在子类中创建节点的代码分离出来，是一个很好的主意。这使得它更容易改变精灵的组成和行为，而无需改变使用精灵的客户端（client）。

3. 添加代码到 `newSpaceship` 方法来添加灯光。

```
SKSpriteNode *light1= [self newLight];

light1.position = CGPointMake(-28.0, 6.0);

[hull addChild:light1];



SKSpriteNode *light2= [self newLight];

Light2.position = CGPointMake(28.0, 6.0);

[hull addChild:light2];
```

4. 实现 newLight 方法。

```
 - (SKSpriteNode *)newLight
```

```
{

    SKSpriteNode *light = [[SKSpriteNode alloc] initWithColor:[SKColor yellowColor] si



    SKAction *blink= [SKAction sequence:@ [

                      [SKAction fadeOutWithDuration:0.25]

                      [SKAction fadeInWithDuration:0.25]];

    SKAction * blinkForever = [SKAction repeatActionForever:blink];

    [light runAction:blinkForever];



    return light;

}
```

当你运行应用程序时，你应该看到一对灯在飞船上。当飞船移动，灯光和它一起移动。这三个节点全都是连续动画。你可以添加额外的动作，让灯光在船的周围移动，它们总是相对船体移动。

创建能交互的节点

在实际的游戏中，你通常需要节点之间能交互。把行为添加给精灵的方法有很多，所以这个例子仅展示其中之一。你将添加新节点到场景，使用物理子系统模拟它们的运动并实现碰撞效果。

Sprite Kit 提供了一个完整的物理模拟，你可以使用它添加自动行为到节点。也就是说，物理在使其移动的节点上自动模拟，而不是在节点上执行动作。当它与物理系统一部分的其他节点交互时，碰撞自动计算并执行。

添加物理模拟到飞船场景

1. 更改 `newSpaceship` 方法来添加一个物理体到飞船。

```
hull.physicsBody = [SKPhysicsBody bodyWithRectangleOfSize:hull.size];
```

构建并运行应用程序。等一下！飞船垂直坠落到屏幕下方。这是因为重力施加到飞船的物理体。即使移动动作仍在运行，物理效果也被应用到飞船上。

2. 更改的 `newSpaceship` 方法来防止飞船受物理交互影响。

```
hull.physicsBody.dynamic = NO;
```

当你现在运行它时，应用程序像之前那样运行。飞船不再受重力影响。稍后，这也意味着飞船的速度将不会受到碰撞的影响，。

3. 添加代码到 `createSceneContents` 方法来生成大量岩石。

```
SKAction * makeRocks = [SKAction sequence:@ [

   [SKAction performSelector:@selector(addRock) onTarget:self]

   [SKAction waitForDuration:0.10 withRange:0.15]

   ]];

[self runAction:[SKAction repeatActionForever:makeRocks];
```

场景也是一个节点，因此它也可以运行动作。在这种情况下，自定义操作调用场景上的方法来创建岩石。序列创建一个岩石，然后等待一段随机时间。重复这个动作，场景不断产生大量新的岩石。

4. 实现 `addRock` 方法。

```
static inline: CGFloat skRandf() {

   return rand()/(CGFloat)RAND_MAX;

}
```

```
static inline CGFloat skRand(CGFloat low, CGFloat high) {

    return skRandf()*(high - low) + low;

}



 - (void)addRock

{

    SKSpriteNode *rock = [[SKSpriteNode alloc] initWithColor: [SKColor brownColor] siz

    rock.position = CGPointMake(skRand(0, self.size.width),self.size.height-50);

    rock.name = @“rock”;

    rock.physicsBody = [SKPhysicsBody bodyWithRectangleOfSize:rock.size];

    rock.physicsBody.usesPreciseCollisionDetection = YES;

    [self addChild:rock];

}
```

构建并运行该项目。岩石现在应该从场景上方落下来。当一块石头击中了船，岩石从船上反弹。没有添加动作来移动岩石。岩石下落并与船碰撞完全是由于物理子系统的作用。

岩石都很小且移动速度非常快，所以代码指定精确的碰撞，以确保所有的碰撞都检测到。

如果你让应用程序运行了一段时间，帧率会开始下降，即使节点计数仍然很低。这是因为节点的代码仅展示出场景中可见的节点。然而，当岩石落下到场景的底部时，它们继续存在于场景中，这意味着物理还在对它们模拟。最终，有如此多的节点正在处理以致 Sprite Kit 减慢了。

5. 实现场景中的 `didSimulatePhysics` 方法来当岩石移动到屏幕之外时移除它们。

```
 - (void)didSimulatePhysics
```

```
{
    [self enumerateChildNodesWithName:@“rock” usingBlock:^(SKNode *node, BOOL *stop){
        if (node.position.y <0)
            [node removeFromParent];
    }];
}
```

每次场景处理一帧，都运行动作和模拟物理。你的游戏可以挂接到这个过程中来执行其他自定义代码。在每一帧，场景将处理物理，然后移除移出屏幕底部的所有岩石。当你运行应用程序时，帧率保持不变。

在场景中，预处理及后处理与动作和物理结合的地方，就是你构建你的游戏的行为的地方。

这就是你第一次体验 Sprite Kit！其他一切都是你在这里看到的基本技术的细化。

试试这个！

这里有一些东西，你可以尝试：

- 做一个 OS X 版本的这个例子。你在视图控制器写的代码，在 OS X 上通常是在一个应用程序代理（delegate）中实现。响应代码需要改变来使用鼠标事件而不是触摸事件。但是，代码的其余部分应是相同的。

- 使用纹理精灵呈现船和岩石。（提示：“使用精灵”）
- 尝试在触摸事件的响应中移动飞船。（提示：“添加动作节点”和“构建场景”）。
- 添加额外的图形效果到场景（提示：“使用其他节点类型”）
- 岩石与船舶碰撞时添加其他行为。例如，使岩石发生爆炸。（提示：“模拟物理”）

使用精灵

精灵是用于创建大部分场景内容的基石，所以在转到其他 Sprite Kit 节点类之前先了解精灵是有用的。精灵用 `SKSpriteNode` 对象表现。一个 `SKSpriteNode` 对象，要么绘制成一个由纹理映射(mapped)的矩形，要么绘制成一个彩色无纹理的矩形。纹理精灵更常见，因为它们代表了你把自定义插图引进场景的主要方式。这个自定义插图可能代表你的游戏的人物角色、背景元素甚至是用户界面元素。但基本的策略是一致的。一个美工创建图像，然后你的游戏加载它们作为纹理。然后你用那些纹理创建精灵，并把它们添加到场景中。

创建纹理精灵

创建一个纹理精灵的最简单方法是让 Sprite Kit 为你创建的纹理和精灵。你把插图存储在应用程序 bundle 中，然后在运行时加载它。清单 2-1 展示了这个代码是多么的简单。

清单 2-1 从存储在 bundle 中的图像创建一个纹理的精灵

```
SKSpriteNode *spaceship = [SKSpriteNode spriteNodeWithImageNamed:@“rocket.png”];

spaceship.position = CGPointMake(100,100);

[self addChild:spaceship];
```

当你以这种方式创建一个精灵，你可以免费得到很多的默认行为：

- 精灵以匹配纹理尺寸的 frame 来创建。
- 精灵以它的位置为中心来渲染。精灵的 `frame` 属性指定的矩形定义了它所涵盖的面积。
- 精灵纹理在帧缓冲区（framebuffer）中是半透明的（alpha-blended）。
- 一个 `SKTexture` 对象被创建并附加到精灵上。此纹理对象每当精灵节点在场景中时自动加载纹理数据，它是可见的，而且对渲染场景是必要的。稍后，如果精灵从场景中移除或不再可见，如果需要那些内存用于其他用途，Sprite Kit 可以删除纹理数据。这种自动内存管理简化但并不能消除在管理你游戏中的美术资产（art assets）方面你需要做的工作。

默认的行为给你一个有用的基础来创建一个基于精灵的游戏。你已经懂得了足够的知识去添加插图到你的游戏，创建精灵，并运行这些精灵的动作来做一些有趣的事情。随着精灵屏幕内外移动，Sprite Kit 尽可能有效地管理纹理和绘制动画的帧。如果这对你已经足够，就花点时间去探索你能对精灵做些什么。或者继续阅读 `SKSpriteNode` 类得到更深入的理解。一路上，你将获得其功能以及如何与美工和设计师交流这些功能的深入理解。并且你会学到更高级的使用纹理的方式以及如何提高基于纹理的精灵的性能。

定制纹理精灵

你可以使用精灵的每个属性独立配置四个不同的渲染阶段：

- 可以移动精灵的 frame，使纹理中的不同点放置在精灵节点的位置。参阅“使用锚点移动精灵的 frame”。
- 可以调整精灵的尺寸。你控制当精灵的尺寸与纹理的尺寸不匹配时纹理如何应用到精灵。参阅“调整精灵的尺寸”。
- 可以在精灵的纹理应用到的精灵时对它着色。请参阅“对精灵着色”。
- 精灵可以使用其他的混合（blend）模式来结合其内容和帧缓冲区的内容。自定义的混合模式对发光（lighting）和其他特效是有用的。请参阅“混合精灵到帧缓冲区中”。

通常情况下，配置精灵执行定位、调整尺寸、着色、混合这四个步骤要根据用于创建精灵纹理的插图。这意味着你很少脱离插图设置属性值。你与美工合作以确保你的游戏配置精灵与插图匹配。

下面是一些也行你可以遵循的策略：

- 在你的项目中用硬编码值创建精灵。这是最快但在长期最不理想的方法，因为这意味着每当美术资产变动时必须更改代码。

- 使用 Sprite Kit 创建自己的工具，让你可以微调精灵的属性值。一旦你有一个你想要的方式配置的精灵，保存精灵到归档中。你的游戏在运行时使用归档创建精灵。

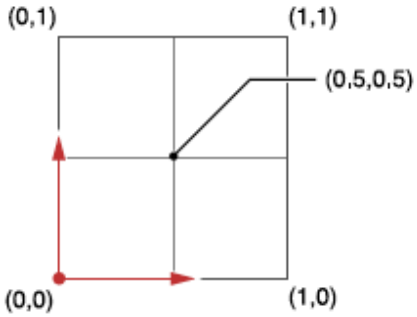
- 在存储在你的应用程序 bundle 的属性列表中存储配置数据。当精灵加载时，加载属性列表并使用它的值来配置精灵。这允许美工提供正确的各个值并在不改变代码的情况下进行更改。

使用锚点移动精灵的frame

默认情况下，精灵的 frame 及其纹理的中心定位在精灵的位置上。然而，你可能想纹理的不同部分出现在节点的位置。经常要作出这样的决定因为纹理描绘的游戏元素不是纹理图像的中心。

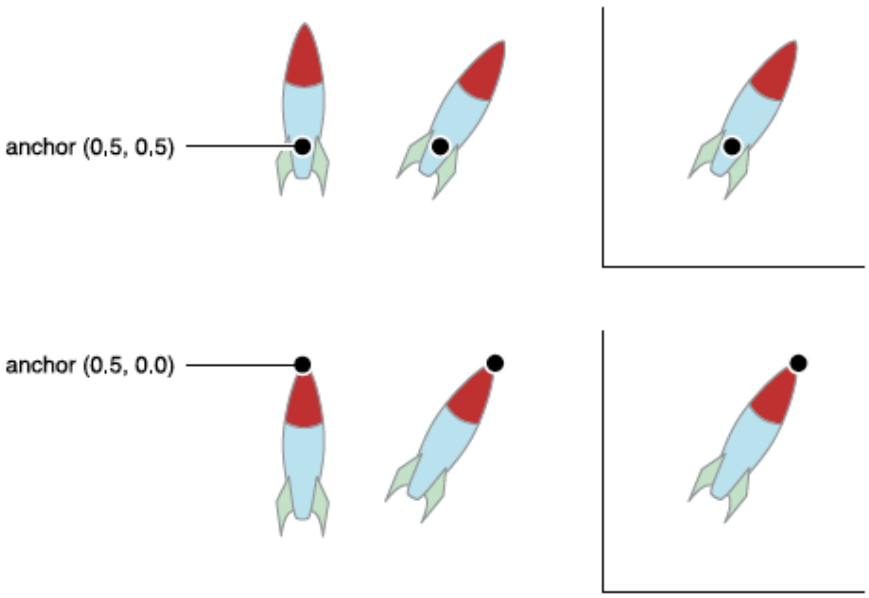
精灵节点的的 `anchorPoint` 属性决定 frame 的哪一点定位在精灵的位置上。锚点在单位坐标系（unit coordinate system）中指定，如图 2-1 所示。单位坐标系的原点位于 frame 的左下角，而（1,1）位于 frame 的右上角。精灵的锚点默认为（0.5,0.5），对应于 frame 的中心。

图2-1 单位坐标系



虽然你想要移动 frame，你这样做是因为你想纹理的相应部分处于位置的中点。图 2-2 展示了一对纹理图像。在第一个图像中，默认的锚点在纹理位置的中心。第二个相反地，选择了图像的顶部一个点。你可以看到，当精灵旋转时纹理图像会围绕这一点旋转。

图2-2 改变精灵的锚点



清单 2-2 展示了如何将火箭的锚点放在前锥体处。通常，你在精灵初始化时设置锚点，因为它与插图对应。然而，你可以在任何时候设置此属性。frame 会被立即更新，并且屏幕上的精灵会在场景下一次渲染时更新。

清单 2-2 设定精灵的锚点

```
rocket.anchorPoint: = CGPointMake(0.5,1.0);
```

调整精灵的尺寸

精灵的 frame 属性的尺寸是由其他三个属性的值决定的：

- 精灵的 `size` 属性指定精灵基准（无缩放）尺寸。当一个精灵使用代码清单 2-1 初始化时，这个属性的值被初始化为于精灵的纹理的尺寸相等。
- 然后基准尺寸根据精灵从 `SKNode` 类继承来的 `xScale` 与 `yScale` 属性进行缩放。

例如，如果精灵的基准尺寸是 `32 × 32` 像素，而它的 `xScale` 的值为 `1.0` 且 `yScale` 的值为 `2.0`，精灵的 frame 的尺寸是 `32 × 64` 像素。

注：场景中精灵的祖先的缩放值也用于缩放精灵。这将改变精灵的有效尺寸，而不改变它的实际 frame 的值。请参阅“节点的很多属性适用于其后代”。

当精灵的 frame 大于它的纹理时，纹理被拉伸以覆盖 frame。一般情况下，纹理会在整个 frame 中被均匀地拉伸，如在图 2-3 中所示。

图 2-3 纹理位伸以覆盖精灵的frame

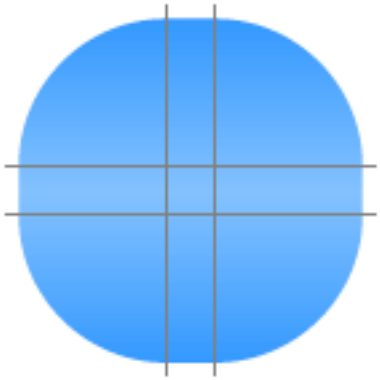
然而，有时你想使用精灵构建用户界面元素，如按钮或健康指示器。通常，这些元素包含固定尺寸的元素，如结束端点（end caps），它不应该被拉伸。在这种情况下，使部分的纹理不拉伸，然后拉伸纹理 frame 剩下的其余部分。

精灵的 centerRect 属性控制缩放行为，该属性在纹理的单位坐标中指定。默认值是一个覆盖整个纹理的矩形，这就是为什么整个纹理被拉伸到整个 frame 的原因。如果指定了一个只涵盖了部分的纹理的矩形，你就创建了一个 3x3 的网格。在网格中的每个盒子有其自己的缩放行为：

- 网格的四个角中的纹理绘制的部分不进行任何缩放。
- 网格的中心在两个方向缩放。
- 中间的上下部分仅水平缩放。
- 中间的左右部分仅垂直缩放。

图 2-4 展示了一个纹理的特写视图，你可能会用它来绘制用户界面按钮。实际元素是 28 点×28 点。四个角是 12×12 像素而中心是 4×4 像素。

图 2-4　可伸缩的按钮纹理



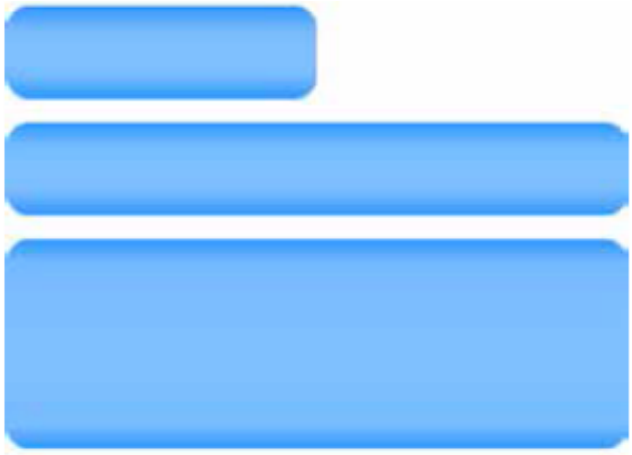
清单 2-3 展示了这个按键精灵将如何初始化。centerRect 属性根据纹理的中心矩形来计算。

清单 2-3　设置中心矩形以调整拉伸行为

```
SKSpriteNode *button = [SKSpriteNode spriteWithImageNamed:@"stretchable_button.png"];

button.centerRect = CGRectMake(12.0/28.0,12.0/28.0,4.0/28.0,4.0/28.0);

....
```

图 2-5 展示了即使在该按钮以不同的尺寸绘制时四个角仍保持不变。

图 2-5　对不同尺寸的按钮应用按钮纹理



对精灵着色

在把纹理应用到精灵之前，你可以使用 `color` 和 `colorBlendFactor` 属性对它着色。默认情况下的颜色混合因子为 `0.0`，这表明纹理未经更改地使用。当你增加这个数字，更多的纹理颜色就会被混合颜色替换。例如，在你的游戏中的怪物受到伤害时，你可能要添加一个红色的色调（tint）给角色。清单 2-4 展示了如何将色调应用于精灵。

清单 2-4 着色精灵的颜色

```
monsterSprite.color = [SKColor redColor];

monsterSprite.colorBlendFactor = 0.5;
```

图 2-6 上色调整纹理的颜色

你也可以使用动作让颜色和颜色混合因子成为动画。清单 2-5 展示了如何短暂地给精灵调色，然后让它恢复正常。

清单 2-5 颜色变化的动画

```
SKAction *pulseRed= [SKAction sequence:@[

                    [SKAction colorizeWithColor:[SKColor redColor] colorBlendFacto:1.0 du

                    [SKAction waitForDuration:0.1],

                    [SKAction colorizeWithColorBlendFactor:0.0 duration:0.15]]
[monsterSprite runAction:pulseRed];
```

混合精灵到帧缓冲区

渲染的最终阶段是把精灵的纹理混合（blend）到其目标帧缓冲区。默认行为使用纹理的 alpha 值混合纹理与目标像素。但是，当你想添加其他的特效到场景时你可以使用其他混合模式。

你可以使用 `BlendMode` 属性来控制精灵的混合行为。例如，附加混合模式在把多个精灵结合在一起时很有用，比如开枪（fire）或发光（lighting）。清单 2-6 展示了如何使用附加混合改变混合模式。

清单 2-6 使用附加混合模式模拟发光

```
lightFlareSprite.blendMode = SKBlendModeAdd;
```

使用纹理对象

虽然 Sprite Kit 可以在你创建一个精灵时为你创建纹理，但在一些更复杂的游戏中，你需要对纹理有更多的控制。例如，你可能需要做以下任何一项：

- 多个精灵之间共享一个纹理。
- 在精灵创建后更改它的纹理。
- 通过一系列的纹理让精灵动起来。
- 用不直接存储在应用程序 bundle 中的数据创建纹理。

- 把节点树渲染成纹理。例如，你可能要对游戏进行截屏，在玩家完成了关卡（level）后展示给他或她。

你通过直接使用 `SKTexture` 对象可以做所有这些事情。纹理对象是可应用于精灵的可复用的图像。你可以创建纹理对象独立于创造精灵。然后，你可以使用纹理对象来创建新的的精灵或改变现有精灵的纹理。它跟 Sprite Kit 为你创建纹理相似，但你对过程有更多的控制权。

从存储在App Bundle的图像创建纹理

清单 2-7 展示了一个类似清单 2-1 中展示的例子，但使它用纹理对象。在这种情况下，代码一次创建了多支火箭，全部来自相同的纹理。通常情况下，你会加载一次纹理，并保持对它的强引用，以便每次需要创建一个新的精灵时都可以使用它。

清单 **2-7** 从 **bundle** 中加载纹理

```
SKTexture *rocketTexture = [SKTexture textureWithImageNamed:@“rocket.png”];

for(int i = 0; i<10; i++)

{

    SKSpriteNode *rocket = [SKSpriteNode spriteNodeWithTexture:rocketTexture];

    rocket.position = [self randomRocketLocation];

    [self addChild:rocket];

}
```

纹理对象本身只是实际的纹理数据的一个占位符。纹理数据占用（intensive）更多的资源，所以当使用它的精灵在屏幕上且可见时，Sprite Kit 只保存它在内存中。

使用纹理图册收集相关的美术资产

通常情况下，存储在你的应用程序 bundle 的美术资产是不相干的图像，却是一起用于相同精灵的图像的集合。例如，下面是一些常见的美术资产的集合：

- 一个角色的动画帧

- 用来创建游戏关卡或者迷宫的地形瓦片（terrain tiles）
- 用于用户界面控件的图像，如按钮、开关和滑块

如果你把这些逻辑分组看成单独的纹理，Sprite Kit 和图形硬件必须运行得更加艰难来渲染场景，而且游戏的性能可能会受到影响。所以，Sprite Kit 使用**纹理图册**把相关的图像收集起来。你指定哪些资产一起使用，然后 Xcode 会自动构建纹理图册。然后在你的游戏加载纹理图册时，Sprite Kit 可以更好地管理性能和内存使用。

创建一个纹理图册

Xcode 可以自动为你从图像集合构建纹理图册。欲了解更多信息，请参阅*纹理图册帮助*。

在创建一个纹理图册时，在收集太多的纹理与太少的纹理到图册之间，有一个平衡的做法。如果你使用的项目数量不足，那么纹理之间切换的开销可能仍然太大。如果你把太多的图像放在一个单一的图册中，那更多的纹理数据会存储在内存中。因为 Xcode 为你构建图册，它可以相对容易地在不同的图册配置之间切换。对你的纹理图册不同的配置做实验，并选择为你提供最佳性能的结合。

加载纹理纹理图册

清单 2-7 中的代码，也可以用来从纹理图册中加载纹理。Sprite Kit 首先搜索指定的文件名的图像文件，但如果它没有找到，那么它会在内置到应用程序 bundle 里面任何纹理图册内部搜索。这意味着，在你的游戏中你不必作出任何编码的更改来支持它。此设计还为美工提供了这样的能力，试验新的纹理而不需要重新构建（rebuild）你的游戏。美工把纹理拖放到应用程序 bundle 中，就可以自动发现它们（覆盖任何之前内置到纹理图册的版本）。一旦美工对纹理满意了，然后你就可以将它们添加到项目中且合并到你的纹理图册中。

如果你想显式使用纹理图册，你可以使用 `SKTextureAtlas` 类。首先，你使用图册的名称创建一个纹理图册对象。然后，使用图册中存储的图像文件的名字查看各自的纹理。清单 2-8 展示了一个这样的例子。它采用了纹理图册装截一个角色的多个动画帧。代码加载这些帧，并将它们存储在一个数组中。

清单 2-8 加载散步动画的纹理

```
SKTextureAtlas *atlas = [SKTextureAtlas atlasNamed:@“monster.atlas”];
```

```
SKTexture *f1 = [atlas textureNamed:@"master-walk1.png"];

SKTexture *f2 = [atlas textureNamed:@"master-walk2.png"];

SKTexture *f3 = [atlas textureNamed:@"master-walk3.png"];

SKTexture *f4 = [atlas textureNamed:@"master-walk4.png"];

NSArray *monsterWalkTextures = @[f1,f2,f3,f4];
```

从纹理的小部分创建纹理

如果你已经有一个 SKTexture 对象，你可以创建新的纹理引用它的一部分。这是非常有效的，因为新的纹理对象引用内存中相同的纹理数据。这个功能跟纹理图册是类似的。通常情况下，如果你的游戏已经有了自己的自定义纹理图册格式，你就可以这样使用。在这种情况下，你负责存储这些存储在自定义纹理图册中的各个图像的坐标。

清单 2-9 展示了如何提取部分的纹理。矩形的坐标在单位坐标空间中。

代码清单 2-9 使用纹理的一部分

```
SKTexture *bottomLeftTexture = [SKTexture textureWithRect:CGRectMake(0.0,0.0,0.5,0.5) inT
```

其他创建纹理的方法

除了从应用程序 bundle 加载纹理，你还可以从其他来源创建纹理：

- 使用 SKTexture 初始化方法通过内存中正确格式化的像素数据、核心图像或对现有的纹理应用一个 Core Image 滤镜来创建纹理。
- SKView 类的 textureFromNode 方法可以把一个节点树的内容渲染成纹理。纹理被指定好尺寸，以便它可以包含节点的内容和所有它的可见后代节点。

当你从应用程序 bundle 中的文件之外的其他来源创建一个纹理时，纹理数据不能被清除，因为 Sprite Kit 不保留用于生成纹理的原始数据的引用。基于这个原因，你应该有节制地使用这些纹理。一旦不再需要它们，马上移除对它们的强引用。

更改精灵的纹理

精灵的 `texture` 属性指向它当前的纹理。你可以将此属性更改为指向一个新的纹理。下一次场景渲染一个新的帧时，它会用新的纹理来渲染。每次你更改纹理时，为了与新的纹理一致，你可能还需要更改其他的精灵属性，如 `size`、`anchorPoint` 和 `centerRect`。一般，确保所有的插图都一致会更好，这样相同的值可用于所有的纹理会。也就是说，纹理应该有一个一致的尺寸和锚点定位，让你的游戏并不需要更新纹理以外的其他任何东西。

因为动画是一个非常常见的任务，你可以使用动作让一个精灵的一系列纹理都动起来。清单 2-10 中的代码展示了如何使用清单 2-8 创建的动画帧数组让精灵的纹理动起来。

清单 2-10 通过一系列的纹理形成动画

```
SKAction *walkAnimation = [SKAction animateWithTextures:monsterWalkTextures timePerFrame:

[monster runAction:walkAnimation];

/ /在这里插入其他代码来移动怪物。
```

Sprite Kit 提供了渠道（plumbing），让你活动或改变精灵的纹理。它不利用你的动画系统的特定设计。但是，这意味着你需要决定精灵可能会需要什么样的动画，并设计自己的动画系统来让这些动画在运行时切换。例如，一个怪物可能有步行，战斗，停顿（idle）和死亡的动画序列。你的代码来决定何时在这些序列之间切换。

预加载纹理来提高帧率

使用 Sprite Kit 的一个主要优点是它自动为你执行了大量的内存管理。Sprite Kit 从图像文件加载纹理，将这些数据转换成图形硬件可以使用的格式，并将其上传到图形硬件。Sprite Kit 很擅长于确定当前帧纹理是否需要渲染。如果纹理不在内存中，它会加载纹理。如果纹理在内存中并且有一段时间没有使用，纹理数据会被丢弃，以便可以加载其他需要的纹理。

如果一次有太多没加载纹理的精灵变为可见，它可能无法在一个单一的动画帧内加载所有这些纹理。纹理加载的延迟可能会导致帧速率突然丢失，这是对用户可见的。Sprite Kit 提供了在精灵变为可见之前预加载纹理的选项。因为你非常熟悉你的游戏的设计，你往往更清楚地知道什么时候即将要使用一套新的纹理。例如，在一个滚动的游戏中，当用户在宇宙间移动时，你知道玩家

即将进入宇宙的哪一部分。然后你可以在动画的每一帧加载三两个纹理，这样当玩家到达那里时纹理已经在内存中了。清单 2-11 展示了如何加载纹理。

清单 2-11 预加载纹理

```
[newSectionTexture preload];
```

预加载代码的正确设计要依赖于你的游戏的引擎。这里有两种可能设计要考虑：

- 当玩家开始一个新的关卡，预加载这个关卡的所有纹理。游戏被划分成各个关卡，每个关卡能保持所有纹理资产同时在内存中。这保证了所有纹理在游戏开始前就加载好，消除任何纹理加载的延迟。
- 如果一个游戏需要比可以适合内存更多的纹理，你需要动态地预加载纹理。通常，这意味着当你能确定它很快就需要会才预加载纹理。例如，在赛车游戏中，玩家总是在在同一方向移动，所以你预加载玩家即将看到的部分赛道的纹理。纹理在后台加载，取代赛道中最旧的纹理。在一个允许玩家时刻控制的冒险游戏中，你可能必须临时加载更多的纹理。

创建彩色精灵

虽然纹理精灵是使用 `SKSpriteNode` 类的最常见的方式，你也可以不用精灵创建精灵节点。类的在精灵缺乏纹理时发生变化：

- 没有纹理可拉伸，所以 `centerRect` 参数被忽略。
- 没有着色步骤，`color` 属性用作精灵的颜色。
- 颜色的 `alpha` 分量被用来确定精灵如何混合到缓冲区。

其他属性（`size`、anchorPoint 和 `blendMode`）照旧不变。

试试这个！

现在你对精灵知道更多了，请尝试以下一些活动：

- 在一个纹理图册中添加插图到你的项目。请参阅“创建一个纹理图册”。
- 加载纹理图册，并用它来创建新的精灵。请参阅“载入纹理从纹理图册”。

- 通过多帧动画让精灵动起来。请参阅清单 2–10。
- 更改你的精灵的属性，看看它们的绘图行为怎么变化。请参阅“自定义纹理精灵。”

你可以在 *Sprite Tour* 示例中找到一些有用的代码。

添加动作到节点

绘制精灵很有用，但是一张静态图像只是一幅画，而不是一个游戏。为了添加游戏剧本（game play），你需要能够让精灵在屏幕周围移动并执行其他逻辑。Sprite Kit 让场景动起来所使用的主要机制是动作。到目前为止，你已经看过了动作子系统的某些部分。现在，是时候更深入地研究如何构造和执行动作了。

一个动作就是定义你想对场景所作的改变的对象。在大多数情况下，一个动作对执行该动作的节点应用其变化。因此，举例来说，如果你想在屏幕上移动精灵，你创建一个移动动作，并告诉精灵节点运行该动作。Sprite Kit 自动动态改变精灵的位置直到动作完成。

动作是自包含的对象

每一个动作是一个不透明的（opaque）对象，描述你想对场景作的改变。一切动作都是由 `SKAction` 类实现，它没有可见的子类。相反地，不同类型的动作都使用类方法来实例化。例如，下面是你用动作来做的最常见的事情：

- 改变一个节点的位置和方向
- 改变节点的尺寸或缩放属性
- 改变节点的可视性或使其半透明
- 改变一个精灵节点的内容，以便它可以通过一系列的纹理动起来
- 给精灵节点着色
- 播放简单的声音
- 从节点树中移除一个节点
- 自定义动作来调用一个块（block）或调用对象上的选择器（selector）

一旦你创建了一个动作，它的类型就不能再改变，并且你只有有限的能力来改变其属性。Sprite Kit 利用动作不可变的性质非常有效地执行它们。

提示： 因为动作是有效不可变的对象，你可以在节点树的多个节点上安全地同时运行相同的动作。出于这个原因，如果你有一个在你的游戏中要反复使用的动作，构建一个单一的动作实例，然后每当你需要一个节点来执行它时再重用它。

动作可以是瞬时的或非瞬时的：

- 瞬时动作在一帧动画内开始并完成。例如，从其父节点中移除节点的动作是一个瞬时动作，因为不能部分地移除一个节点。相反地，执行该动作时，节点会被立即移除。
- 非瞬时动作有一个动画效果的持续时间。在执行时，该动作将在动画的每一帧进行处理，直到动作完成。

用来创建动作类方法的完整列表在 *SKAction 类参考*中描述，但你只有在准备好进行详细查看如何配置具体动作时，才需要去那里。

节点运行动作

一个动作只在你告诉一个节点运行它后才会执行。运行一个动作最简单的方法是调用的节点的 `runAction:` 方法。清单 3-1 创建了一个新的移动动作，然后告诉节点来执行它。

清单 3-1 运行一个动作

```
[SKAction *moveNodeUp = [SKAction moveByX:0.0 y:100.0 duration:1.0];

[rocketNode runAction:moveNodeUp];
```

移动动作有一个持续时间，所以这个动作在动画的多个帧中由场景处理，直到流逝的时间超过了动作的持续时间。在动画完成后，动作就从节点中移除。

你可以在任何时候运行动作。然而，如果你添加动作到节点时场景正在处理动作，直到下一帧前新的动作可能不会执行。场景用来处动作的步骤，在“高级场景处理”中有更详细地描述。

一个节点可以同时运行多个动作，即使那些动作在不同的时间执行。场景跟踪每个动作还要多久才完成并计算动作对节点产生的效果。例如，如果你运行两个动作移动相同的节点，这两个动作对每一帧都应用变化。如果移动动作大小相等、方向相反，则该节点将保持静止。

因为动作处理绑定到场景，只有当节点是呈现场景的节点树的一部分时动作才会被处理。你可以这样利用此特性：通过创建一个节点并分配动作给它，但等到以后再添加节点到场景。后来，当

节点加入到了场景时，会立即开始执行它的动作。这种模式特别有用，因为在复制节点时，一个节点正在运行的动作也被复制和归档。

如果一个节点在运行任何动作，它的 `hasActions` 属性返回 `YES`。

取消运行动作

要取消某个节点正在运行的动作，调用它的 `removeAllActions` 方法。所有动作都立即从节点中移除。如果移除动作有持续时间，任何对节点已经作出的更改将保持不变，但不执行进一步的变化。

在动作完成时接收回调

`runAction:completion:`方法与 `runAction:`方法是相同，但动作完成后，你的块被调用。这个回调只在动作运行到完成时被调用。如果动作完成之前被移除，完成处理程序（handler）永远不会被调用。

使用命名动作来精确控制动作

通常情况下，你看不到某个节点的哪些动作在执行，而如果你想移除动作，你必须移除所有的动作。如果你需要查看特定动作是否在执行或移除一个指定的动作，你必须使用**命名动作（named actions）**。命名的动作使用一个唯一的键名来识别该动作。你可以启动、移除、查找、更换节点上的命名动作。

清单 3-2 与 清单 3-1 相似，但现在的动作用一个键标识，`ignition`。

清单 3-2 运行命名动作

```
[SKAction *moveNodeRight = [SKAction moveByX:100.0 y:0.0 duration:1.0];

[spaceship runAction:moveNodeRight withKey:”ignition”];
```

以下基于键的方法可用：

- `runAction:withKey:` 方法用于运行动作。如果已经有一个动作使用相同键的动作在执行，它会在新的动作添加之前先被移除掉。
- `actionForKey`:方法用于确定是否已经有一个使用那个键的动作在运行。

- removeActionForKey:方法用于移除动作。

清单 3-3 展示了如何使用一个命名动作来控制精灵的运动。当用户点击场景的内部，方法会被调用。该方法确定点击发生的位置，然后告诉精灵运行一个动作移动到那个位置。提前计算了持续时间，从而使精灵总是表现为以固定的速度在移动。因为此代码使用 runAction:withKey: 方法，如果精灵已经在移动，之前的移动会在中途停止而新的动作使精灵从当前位置移动到新位置。

清单 3-3 移动精灵到最新的鼠标点击位置

```
- (void)MouseDown:(NSEvent *)theEvent

{

    CGPoint clickPoint = [theEvent locationInNode:self.playerNode.parent];

    CGPoint charPos = self.playerNode.position;

    CGFloat distance = sqrtf((clickPoint.x charPos.x)*(clickPoint.x charPos.x)+

                         (clickPoint.y charPos.y)*(clickPoint.y charPos.y));



    SKAction *moveToClick = [SKAction moveTo:clickPoint duration:distance/characterSpeed];

    [self.playerNode runAction:moveToClick withKey:@“moveToClick”];

}
```

创建运行其他动作的动作

Sprite Kit 提供了许多标准的动作类型用来改变在你的场景中的节点的属性。但动作真正的力量是发生在动作结合在一起的时候。你可以通过结合动作创建复杂和有表现力的动画，这些动画仍然通过运行一个单一的动作来执行。一个**复合动作**与任何基本动作类型的使用同样的容易。考虑到这一点，现在是时候学习**序列动作、组动作和重复动作了**。

- *序列动作*（sequence action）具有多个子动作。序列中的每一个动作在前一个动作结束后开始。
- *组动作*（group action）具有多个子动作。存储在该组中的所有动作在同一时间开始执行。
- *重复动作*（repeating action）只有一个子动作。当子动作完成后，它重新启动。

序列运行一系列的动作

序列是一个连续运行的动作的集合（set）。当一个节点运行一个序列，动作以连续的顺序触发。当一个动作完成后，立即开始下一个动作。当序列中的最后一个动作完成，序列动作也完成。

清单 3-4 展示了使用一个其他动作的数组来创建序列。

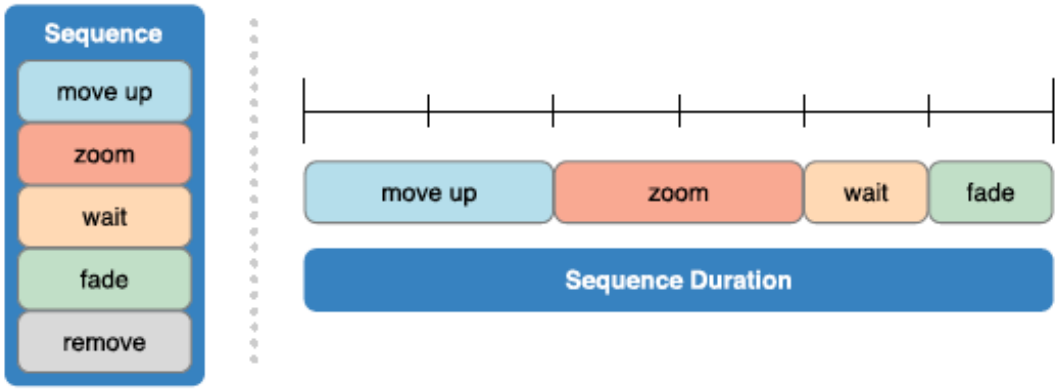
清单 3-4 创建动作的序列

```
SKAction *moveUp = [SKAction moveByX:0 y:100.0 duration:1.0];

SKAction *zoom = [SKAction scaleTo:2.0 duration:0.25];

SKAction *wait = [SKAction waitForDuration:0.5];

SKAction *fadeAway = SKAction fadeOutWithDuration:0.25];

SKAction *removeNode = [SKAction removeFromParent];



SKAction *sequence = [SKAction sequence:@[moveUp, zoom, wait, fadeAway, removeNode];

[node runAction:sequence];
```

在这个例子中有几件事情值得注意：

- `wait` 动作是一个特殊的动作，它通常仅在序列中使用。这个动作只是等待一段时间，然后不做任何事情就结束。等待动作用于控制序列的定时。
- `removeNode` 动作是一个瞬时动作，所以它不花时间来执行。你可以看到，虽然这个动作是序列的一部分，它不会出现在图 3-1 的时间轴上。作为瞬时动作，在淡入动作完成后它马上开始和结束。然后序列也结束了。

图3-1 move和zoom序列时间表



组并行地运行动作

组动作是一组在组执行时就同时开始执行的全部动作的集合（collection）。当你想要动作同时发生时你可以使用组。例如，代码清单 3-5 中旋转并移动一个精灵形成车轮在屏幕上滚动的错觉。使用组（而不是运行两个独立的动作）强调，这两个动作是密切相关的。

清单 3-5 使用一组动作来旋转一个车轮

```
SKSpriteNode *wheel = (SKSpriteNode *)[self childNodeWithName:@"wheel"];

CGFloat circumference = wheel.size.height * M_PI;


SKAction oneRevolution = [SKAction rotateByAngle:-M_PI*2 duration:2.0];

SKAction *moveRight = [SKAction moveByX:circumference y:0 duration:2.0];



SKAction *group = [SKAction group:@[oneRevolution, moveRight];

[wheel runAction:group];
```

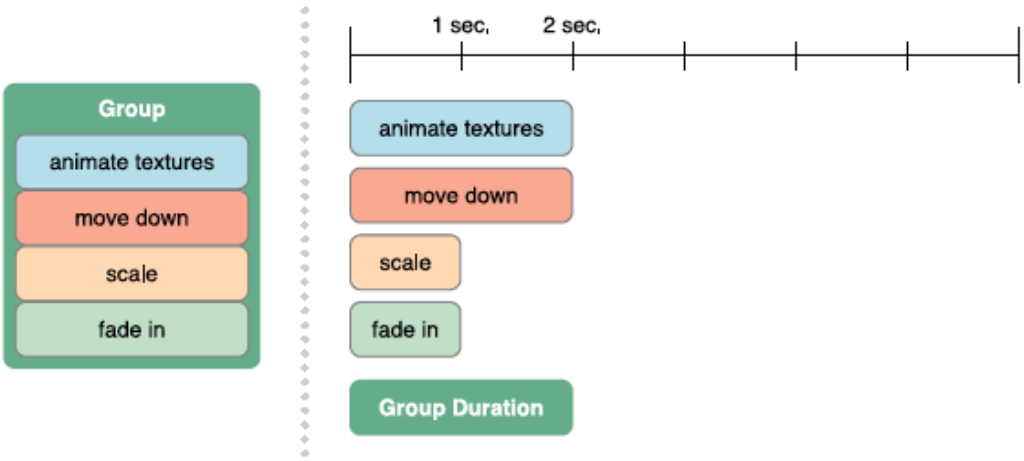
虽然在组中的动作同时开始，组要直到组中的最后一个动作结束运行时才算完成。清单 3-6 展示了一个更复杂的组，它包含的动作有不同的时间值。精灵通过纹理形成动画并在屏幕上向下移

动两秒。然而，当组执行时精灵放大并从透明淡入到完全可见，以完全可见。图 3-2 展示了使精灵出现的这两个动作，只完成组的动画的一半。组将继续进行，直到另外两个动作完成。

清单 3-6 用不同的时间值创建一组动作

```
[sprite setScale:0];

SKAction *animate = [SKActio animateWithTextures:textures timePerFrame:2.0/numberOfTextur

SKAction *moveDown = [SKAction moveByX:0 y:-200 duration:2.0];

SKAction *scale = [SKAction scaleTo:1.0 duration:1.0];

SKAction *fadeIn = [SKAction fadeInWithDuration:1.0];



SKAction *group= [SKAction group:[animate, moveDown, scale, fadeIn];

[sprite runAction:group];
```

图 3-2 分组动作同时启动，但独立完成



重复动作多次执行其他的动作

重复动作允许循环另一个动作，所以可以重复多次。当执行重复动作时，其实是执行它所含动作。每当要循环的动作完成时，它又被重复动作重新启动。清单 3-7 展示了创建重复动作的方法。你可以创建一个动作重复有限次数或无限次数。

清单 3-7 创建重复动作

```
SKAction *fadeout = [SKAction fadeOutWithDuration:0.25];

SKAction *fadeIn = [SKAction fadeInWithDuration:0.25];

SKAction *pulse = [SKAction seqence:@[fadeout, fadeIn];



SKAction *pulseFiveTimes = [SKAction repeatAction:pluse count:5];

SKAction *pulseForever = [SKAction repeatActionForever:pulse];
```

图 3-3 展示了的序列的计时安排（timing arrangement）。你可以看到整个序列完成然后重复。

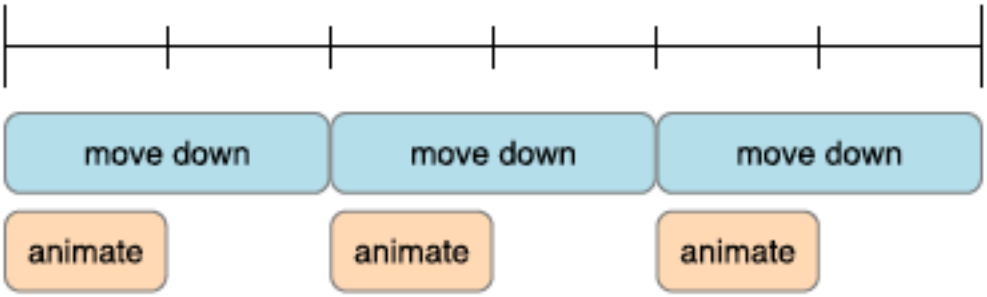
图 3-3 重复动作的计时

当你重复一组时，整组必须完成之后再重新启动该组。清单 3-8 创建了一个组，它移动一个精灵并通过纹理形成动画，但在这个例子中，两个动作有不同的持续时间。图 3-4 展示组重复时的计时图。你可以看到，纹理动画运行完成后，然后直到该组重复前都没有动画发生。

清单 3-8 重复一组动画

```
SKAction *animate = [SKAction animateWithTextures:textures timePerFrame:1.0/numberOfImage

SKAction *moveDown = [SKAction moveByX:0 y:-200 duration:2.0];



SKAction *group = [SKAction group:@[animate, moveDown]];
```

图 3-4 重复组的计时

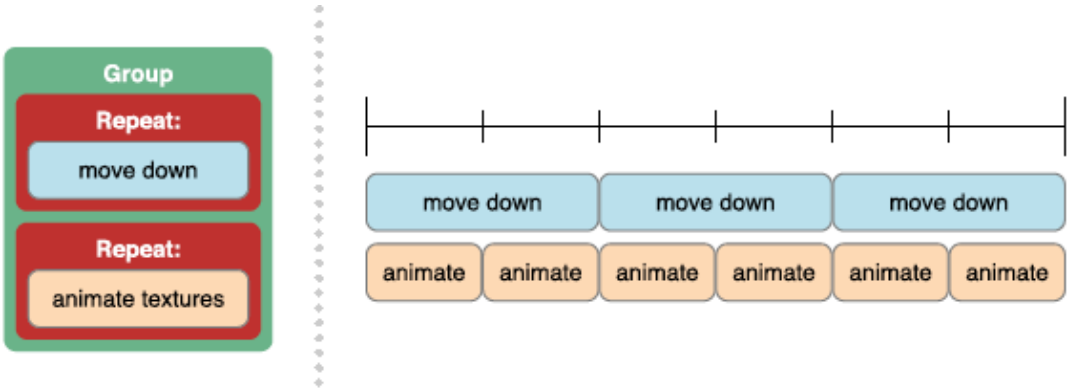


你可能想要的是，每个动作以它自身的固有频率重复。要做到这一点，只要创建一套重复动作，然后把它们组合在一起。清单 3-9 展示了你如何实现图 3-5 所示的计时。

清单 3-9 把一套重复动作组合

```
SKAction *animate = [SKAction animateWithTextures:textures timePerFrame:1.0/numberOfImage

SKAction *moveDown = [SKAction moveByX:0 y:-200 duration:2.0];



SKAction *repeatAnimation = [SKAction repeatActionForever:animate];

SKAction *repeatMove = [SKAction repeatActionForever:moveDown];



SKAction *group = [SKAction group:@[repeatAnimation, repeatMove];
```

图 3-5 每个动作以其固有间隔重复



配置动作计时

默认情况下，一个动作的持续时间根据你指定的持续时间线性变化。但是，你可以通过一些属性调整动画的计时：

- 通常情况下，动画动作线性运行。动作的 `timingMode` 属性可以用来为动画选择一个非线性的计时模式。例如，你可以让动作快速开始，然后在剩余的持续时间中减速。
- 动作的 `speed` 属性改变动画播放的速率。你可以在动画默认计时上加速或减速。

  speed 值为 `1.0` 是正常的速度。如果动作的 speed 属性设置为 `2.0`，当节点执行动作时，它速度快了一倍。要暂停动作，将值设置为 `0`。

  如果你调整那些包含其他动作（例如组、序列或重复动作）的动作的速率，速率会应用到所包含的动作。附加的动作也会受到它们自己的 `speed` 属性的作用。
- 节点的 `speed` 属性与动作的 `speed` 属性具有相同的效果，但该速率适用于该节点或景树中的任意后代所处理的*所有*动作。

Sprite Kit 通过找到所有应用于该动作的速率并将它们相乘，决定应用于动画的速率。

使用动作的提示

动作最好的工作方式是，你创建一次然后使用多次。只要有可能，提早创建动作，并将它们保存在一个很容易地检索和执行的位置。

根据动作的类型，以下这些位置可能都是有用的：

- 节点 `userData` 属性
- 父节点的 `userData` 属性，如果几十个节点的共享同样的动作和相同的父节点
- 场景的 `userData` 的属性，动作由场景中多个节点共享
- 如果子类化，就用子类的 `userData` 属性

如果你需要设计师或美工输入节点的属性如何生成动画，可以考虑把动作创建代码移到你的自定义设计工具中。然后归档动作，并加载它到你的游戏引擎。欲了解更多信息，请参阅“Sprite Kit 最佳实践”。

什么时候你不应该使用动作

虽然动作非常有效，它们不是免费的。创建动作并执行它是有成本的。如果你打算在动画的每一帧改变节点的属性，而这些变化在每帧都需要重新计算，你最好直接改变节点而不使用动作来做这些。欲了解更多关于你可能在你的游戏的哪些地方要这样做的信息，请参阅“高级场景处理。”

试试这个！

这里有一些东西可以用动作试试：

- 探索 *SKAction 类参考*，并对你的精灵尝试各种不同的动作。
- 创建一个动作组，让它使用其他动作同步移动屏幕上的一个精灵，比如，通过一系列精灵图像生成动画或旋转精灵。
- 使用命名动作创建可撤销的动作。将这些动作与你的用户界面代码连接起来。
- 创建序列，用它讲述了一个有趣的故事。例如，考虑在你的游戏启动时创建可活动的标题画面来显示。

构建场景

对于场景的使用，你已经学过了很多的东西。这里对重要的事实再快速回顾一下：

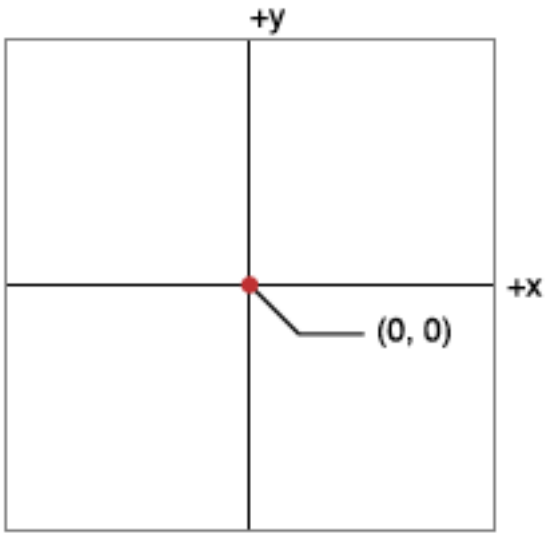
- 场景（`SKScene` 对象），用来提供 `SKView` 对象要渲染的内容。
- 场景的内容被创建成树状的节点对象。场景是根节点。
- 在场景由视图呈现时，它运行动作并模拟物理，然后渲染节点树。
- 你可以通过子类化 `SKScene` 类创建自定义的场景。

心中有了这些基本概念之后，是时候来学习更多关于节点树和建设场景的知识了。

节点给子节点提供坐标系

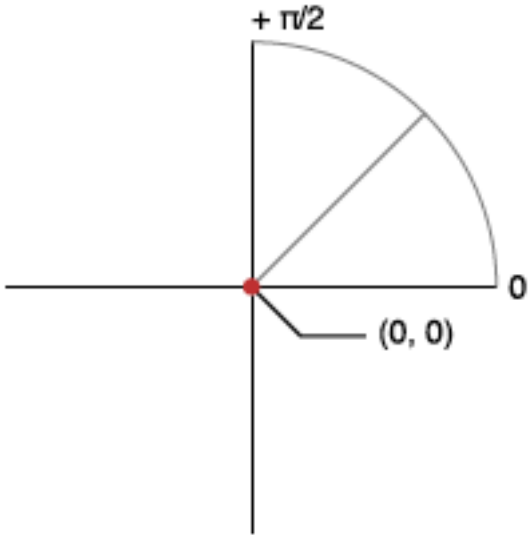
当一个节点被放置在节点树中时，它的 `position` 属性把它定位在由它的父节点提供的坐标系内。Sprite Kit 在 iOS 和 OS X 中使用相同的坐标系。图 4-2 展示了 Sprite Kit 的坐标系。与 UIKit 或 AppKit 一样，坐标值用点来测量；如果必要，在渲染场景时会把点转换为像素。正数的 x 坐标在右边而正数的 y 坐标在屏幕上方。

图 4-1 Sprite Kit 坐标系



Sprite Kit 还有一个标准的旋转约定（rotation convention）。图 4-2 展示了相反的坐标约定。弧度为 0 的角指定正 x 轴。沿反时针方向是正角度。

图 4-2 旋转坐标约定



当你的仅使用 Sprite Kit 代码时，一致的坐标系意味着你可以轻松地在游戏的 iOS 和 OS X 版本之间共享代码。然而，它更意味着当你编写特定 OS 专用（OS-specific）的用户界面代码时，你可能需要在操作系统的视图坐标约定与 Sprite Kit 坐标系之间进行转换。最常见的情况就是使用 iOS 视图，它们有一个不同的坐标约定。

只有某些节点包含内容

不是所有的节点都绘制内容。例如，`SKSpriteNode` 类绘制一个精灵，但 `SKNode` 类不画任何东西。读取某个指定节点对象的 `frame` 属性，你就可以知道它是否绘制内容。节点在父节点的坐标系中绘制，frame 代表了它在该坐标系中的可视区域。如果节点绘制内容，frame 具有一个非零的尺寸。对于场景，frame 总是反映场景坐标空间中的可见部分。

如果一个节点有绘制内容的后代节点，节点的子树也有可能提供内容，即使它本身并不提供任何内容。你可以调用节点的 `calculateAccumulatedFrame` 方法来检索一个矩形，它包括整个绘制节点及它所有后代的区域。

创建场景

场景由视图来呈现。它的很多属性对视图如何呈现场景都有影响。这些属性允许你定义场景的原点位置和场景的尺寸。如果场景的尺寸与视图不匹配，你还可以定义场景缩放方式以适合视图。

场景的尺寸定义其可见区域

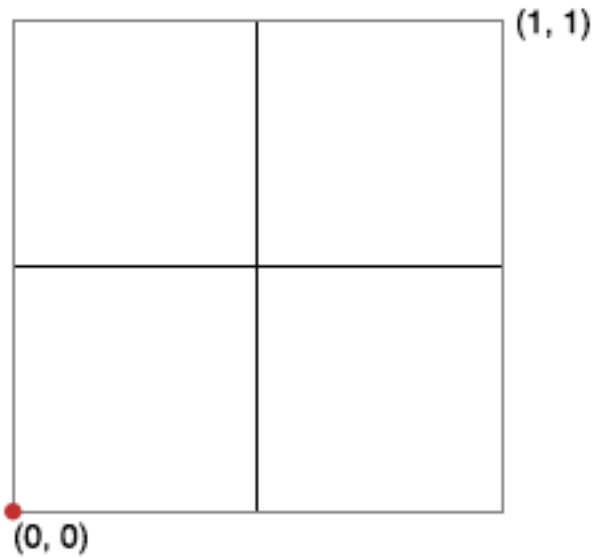
在场景首次初始化时，它的 size 属性由指定初始化器配置。场景的尺寸以点为单位指定场景中可见部分的尺寸。这只用于指定场景的可见部分。树中的节点可以定位在该区域之外，这些节点仍由场景处理，但被渲染器（renderer）忽略。

使用锚点在视图中定位场景的坐标系

缺省情况下，一个场景的原点被放置在视图的左下角上，如图 4-3 中所示。因此，一个场景初始化为宽 1024 和高 768，在左下角是原点(0,0)，右上角坐标是(1024,768)。frame 包含(0,0)-(1024,768)。

场景的 position 属性被 Scene Kit 忽略，因为场景始终是一个节点树的根节点。它的默认值是 CGPointZero，且你不能改变它。但是，你可以通过设置场景的 anchorPoint 属性移动它的原点。锚点在单位坐标空间中指定，并选择封闭视图中的一个点。

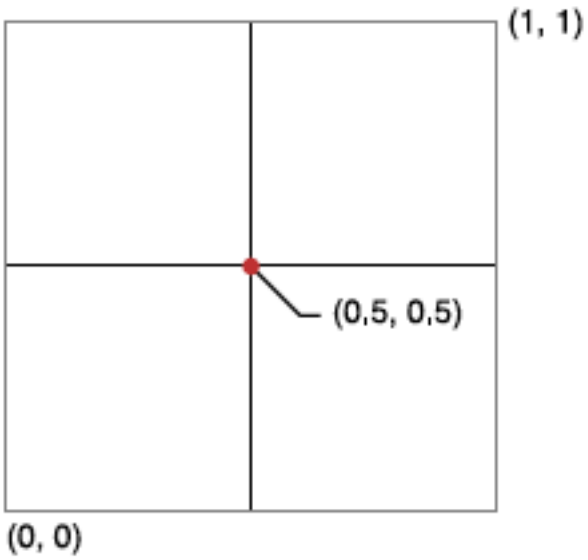
图4-3 默认锚一个场景是在左下角的视图



锚点的默认值是 CGPointZero，放置于左下角。场景的可见坐标空间是从(0,0)到(width，hight) 。对于不滚动场景内容的游戏，默认的锚点是最有用的。

锚点第二模式（second-mode）的值通常是(0.5,0.5)，在中间的视图，如图 4-4 中所示，把场景的原点定在视图的中心。场景的可视坐标空间是从(-width/2,-hight/ 2)到(width/2,hight/2)。当你想轻松地相对屏幕的中心定位节点时，把场景的锚点定在中心是最有用的，比如一个滚动游戏。

图4-4 移动锚点到视图的中心



总结一下，`anchorPoint` 和 `size` 属性用来计算场景的 frame，frame 包含了场景的可见部分。

缩放场景的内容以适合视图

场景渲染后，它的内容被复制到呈现视图。如果视图和场景的尺寸相同，则内容可以直接复制到视图中。如果两者不一样，那么场景会被缩放以适合视图。`scaleMode` 属性决定内容如何缩放。

当你设计游戏时，你应该决定处理场景的 `size` 和 `scaleMode` 属性的战略。以下是一些最常见的策略：

- 以恒定尺寸实例化场景，并且永远不改变它。必要时允许 Sprite Kit 把内容缩放到视图。这场样景有一个可预见的坐标系统和 frame。然后，你的美术资产和游戏逻辑可以基于这个坐标系。
- 调整游戏中的场景的尺寸。在必要的地方，调整你的游戏逻辑和美术资产来匹配场景的尺寸。
- 将 `scaleMode` 属性设置为 `SKSceneScaleModeResizeFill`。Sprite Kit 会自动调整场景的尺寸，使其始终与视图的尺寸相匹配。在必要的地方，调整你的游戏逻辑和美术资产来匹配场景的尺寸。

当你计划使用一个恒定尺寸的场景，清单 4-1 展示了一个典型的实现。与你在“深入 Sprite Kit”中创建的例子一样，这个代码指定了第一次呈现场景时要执行的方法。这个方法配置场景的属性，包括它的缩放模式，然后添加内容。在此示例中，缩放模式被设置为

SKSceneScaleModeAspectFit，它在两个维度上以相同的比例缩放内容，并确保所有的场景的内容都可见。在必要的地方，这种模式会添加黑边（letterboxing）。

清单 4-1 对一个固定尺寸的场景使用缩放模式

```
- (void)createSceneContent

{

    self.scaleMode = SKSceneScaleModeAspectFit;

    self.backgroundColor = [SKColor blackColor];

/ /在这里添加更多的场景内容

...

}
```

如果你希望在运行时改变场景的尺寸，那么应该用初始的场景尺寸来确定要使用的美术资产，以及任何依赖于场景尺寸的游戏逻辑。你的游戏应该重写场景的 didChangeSize:方法，每当场景变化尺寸时会调用此方法。当这个方法被调用时，你应该更新场景的内容，以匹配新的尺寸。

创建节点树

你可以通过创建节点之间的父子关系的方式来创建节点树。每个节点维护一个有序的子节点列表，可以通过读取节点的 children 属性进行引用。子节点在树中的顺序会影响场景处理的多个方面，包括碰撞测试（hit testing）和渲染。所以，适当地组织节点树是很重要的。

表 4-1 列出了构建节点树最常用的方法。完整的方法列表在 *SKNode 类参考*中提供。

表4-1 操作节点树的常用方法

方法	描述
addChild:	添加一个节点到接收者的子节点列表的末尾。
insertChild:atIndex:	插入一个孩子到接收者的子节点列表中的特定位置。

removeFromParent	从父节点中移除接收节点。

当你需要直接调整节点树，可以使用表 4-2 中的属性查看（uncover）树的结构。

表4-2 横移节点树

属性	描述
children	接收节点的子节点所形成的 SKNode 对象数组。
parent	如果该节点是另一个节点的子节点，这个属性指向父节点。否则，它为 nil。
scene	如果该节点包含在场景中的任何地方，它返回作为节点树的根的场景节点。否则，它为 nil。

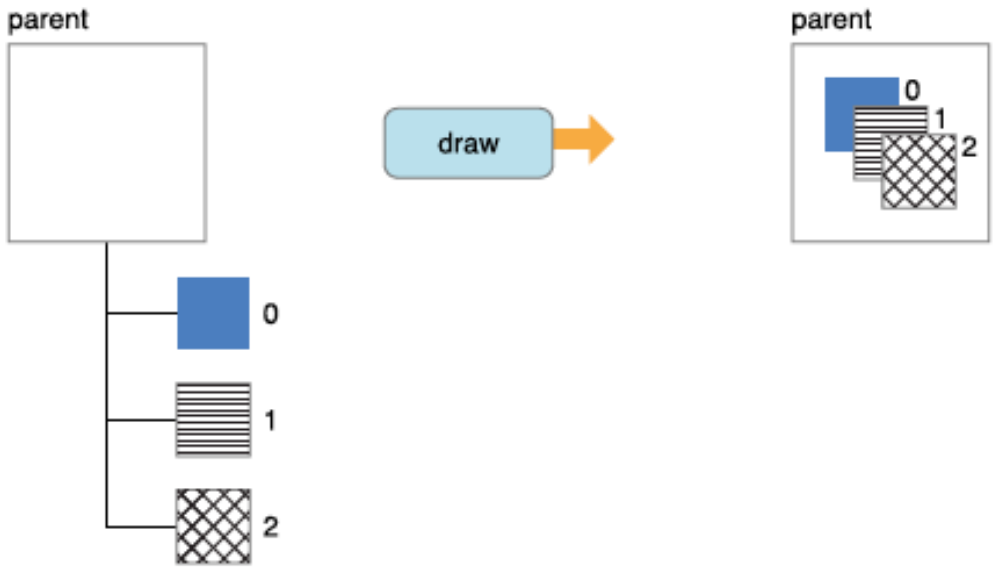
理解节点树的绘制顺序

场景渲染的标准行为遵循以下一对简单的规则：

- 父节点先绘制自身的内容再渲染子节点。
- 子节点以它们在子节点数组中的顺序依次渲染。

图 4-5 展示了如何渲染有三个子节点的节点。

图4-5 父节点在子节点前绘制



在你在“深入 Sprite Kit”写的代码中，创建了一个场景，还有一个飞船和多们岩石。两个灯被指定为飞船的子节点，而飞船和岩石又是场景的子节点。因此，场景用以下方式渲染其内容：

1. 场景渲染它本身，清除内容为它的背景色。
2. 场景渲染飞船节点。
3. 飞船节点渲染它的子节点，即飞船上的灯光。
4. 场景渲染岩石节点，它们在场景的子节点数组中的飞船节点后出现。

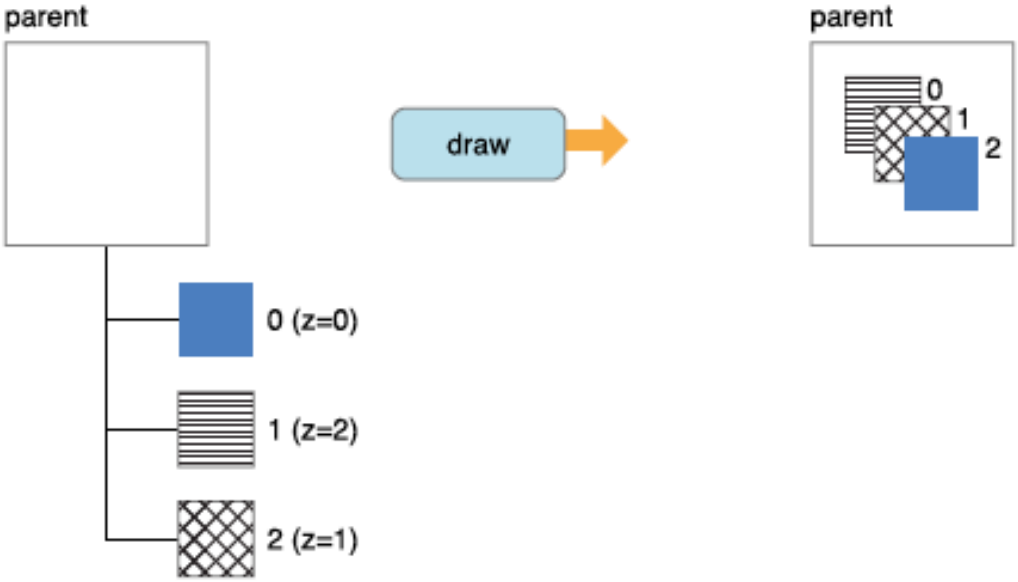
重要提醒： `SKCropNode` 和 `SKEffectNode` 节点类轻微地改变了场景的渲染行为。它们不绘制自己的内容，而是改变它们的子节点在场景中的渲染方式。虽然如此，还是用相同的绘制顺序。要想了解更多信息，请参阅“使用其他节点类型”。

维护节点的子节点的顺序，有时会比你感兴趣的工作还要多。给每个节点一个明确的深度值并允许 Sprite Kit 管理你的绘制顺序，会更容易。你可以使用节点的 `zPosition` 属性这样做。当一个节点创建时，`zPosition` 的属性设为 `0.0`。通过设置节点的 z 轴位置，相对于它的同级节点，你让它更靠近或更远离顶层的渲染顺序。下面是 z 轴位置添加后场景的渲染方式：

- 父节点先绘制自身的内容再渲染子节点（不变）。
- 父节点渲染子节点从 z 值最大的孩子开始，并从 z 值最小的孩子结束。所以，z 轴位置表示从子节点到一个假想的摄像机（camera）位置的距离。
- 如果两个子节点有相同的 z 值，则在数组中较早出现的那个先绘制。

图 4-5 展示了有三个子节点的节点。通常情况下，子节点将按它们出现在子节点数组的顺序依次渲染。然而，在这种情况下，这三个节点有自定义的深度值，导致它们以不同的顺序渲染。

图 4-6 子节点按深度顺序渲染。



碰撞测试的顺序与绘制顺序相反

当 Sprite Kit 处理场景内的触摸或鼠标事件时，它在场景中查找想接受该事件的最接近节点。如果该节点不想处理事件，则检查下一个最接近的节点，依此类推。处理碰撞测试的顺序基本上是绘制顺序的反方向：

1. 父节点只在它的子节点传给它后才接受事件。
2. 子节点从最小的 z 值到最大的 z 值进行处理。
3. 如果两个子节点有相同的 z 值，先测试数组中后出现的那个。

在碰撞测试中要考虑一个节点，它的 `userInteractionEnabled` 属性必须设置为 `YES`。场景节点以外的任何节点的默认值都是 `NO`。要接收事件的节点需要从它的父类（iOS 上的 `UIResponder` 和 OS X 上的 `NSResponder`）实现适当的响应方法。这是你在 Sprite Kit 中必须实现特定平台的代码为数不多的地方之一。

有时候，你也想直接查找节点，而不是依赖于标准的事件处理机制。Sprite Kit 允许你问一个节点是否有任何的后代节点与坐标系的特定点相交。调用 `nodeAtPoint:`方法找到的第一个与该点相交的后代节点，或使用 `nodesAtPoint:`方法接收与该点相交所有节点的数组。

使用节点的深度来添加其他效果

Sprite Kit 只使用 `zPosition` 的值来确定碰撞测试和绘制顺序。但是，你可以使用你指定的值来实现自己的游戏特效。例如，你可以：

- 使用的节点的深度来确定节点在屏幕上移动的速度。通过增加不同深度的节点，你可以模拟视差滚动（parallax scrolling）。
- 使用节点的深度来影响它渲染的方式。

搜索节点树

通过组织树中的节点来确定精确的场景渲染顺序，而不是通过那些节点在你游戏中扮演的角色。正因为如此，`SKNode` 类提供了 `name` 属性。你可以命名一个节点，以区别于树中的其他节点，然后搜索这些节点。

节点的名称应该是没有任何标点的字母数字字符串。清单 4-2 展示了你可以如何命名三个不同的节点来区分它们彼此。

清单 4-2 命名一组节点

```
playerNode.name = @“player”;

monsterNode1.name = @“goblin”;

monsterNode2.name = @“ogre”;
```

当你的命名游戏的节点时，你应该决定名称是否是唯一的。如果你决定一个节点名是唯一的，那么该名字就是为了识别该节点而不是其他。另一方面，如果节点的名字不是你的游戏中唯一的，它通常代表了相联合点的集合。例如，在清单 4-2 中，可能游戏中有多个小妖精，你或许想用相同的名称识别它们。但玩家可能是游戏中唯一的节点。
在你的应用程序中，节点名称通常有两个目的：

- 你可以根据节点的名称编写自己的实现游戏逻辑的代码。例如，两个物理对象碰撞时，你可能会使用节点名称来确定碰撞如何影响游戏。
- Sprite Kit 还为你提供了一些强大的工具来搜索场景内的节点。

`SKNode` 类实现了搜索节点树的两种方法：

- childNodeWithName:方法搜索节点的子节点，直到找到一个匹配的节点，然后停止并返回该节点。这种方法通常用于对具有唯一名称的节点进行搜索。
- enumerateChildNodesWithName:usingBlock:方法搜索节点的子节点，并在找到的每个匹配的节点调用一次 block。当你想找到的所有节点共享同一个名称时，你可以使用此方法。

清单 4-3 展示了在你的场景类上你可以如何创建方法来查找玩家节点。

清单 4-3 寻找玩家节点

```
- (SKNode *)playerNode

{

   return [self childNodeWithName:@“player”];

}
```

当这个方法在场景上调用时，场景搜索它的子节点（且仅搜索子节点）中名称属性匹配搜索字符串的节点，然后返回这个节点。当指定搜索字符串时，你可以指定节点的名称或类的名称。例如，如果你为玩家节点创建了自己的子类，并把它命名为 PlayerSprite，那么你可以指定 PlayerSprite 作为搜索字符串代替 player；它将返回相同的节点。

高级搜索

默认的搜索只搜索一个节点的子节点，而且必须完全匹配节点或类的名称。然而，Sprite Kit 提供了一个表达式搜索语法，允许你进行更高级的搜索。例如，你可以像之前一样做同样的搜索，但搜索整个场景树。或者你可以搜索节点的子节点，但匹配某个模式，而不需要精确匹配。

表 4-3 描述了不同的语法选项。搜索使用常见的正则表达式语义。

表4-3 搜索语法选项

语法	描述
/	当放在搜索字符串的开头时，这表示应该对树的根节点进行搜索。

//	当放在搜索字符串的开头时，这指定搜索应从根节点开始，并在整个节点树中递归进行。这在搜索字符串之外的其他地方都是不合法的。
..	这表明搜索应该向上移到该节点的父节点中进行。
/	当放在搜索字符串的开头以外的任何地方时，这表明搜索应该移到节点的子节点中进行。
*	搜索匹配零个或多个字符。
[以逗号或破折号分隔的字符]	搜索将匹配括号内包含的任意字符。
字母和数字字符	搜索只匹配指定的字符。

表 4-4 展示了一些有用的搜索字符串来帮助你入门。

表4-4 搜索示例

搜寻字串	描述
/MyName	搜索根节点的子节点并匹配名为 `MyName` 的任何节点。
//*	这个搜索字符串匹配场景中的每一个节点。
//MyName/..	搜索整个场景并匹配每个名为 `MyName` 的节点的父节点。
A[0-9]	搜索节点的子节点并返回任何命名为 `A0`，`A1`，...，`A9` 的子节点。
Abby/Normal	搜索节点的孙子节点并返回任何名称是 Normal 且其父节点名为 Abby 的节点。
//Abby/Normal	搜索整个场景并返回任何名称是 Normal 且其父节点名为 Abby 的节点。

节点的很多属性适用于它的后代

当你改变一个节点的属性，效果往往传播到该节点的后代。净效果是一个子节点的渲染不仅基于它自身的属性，也基于它祖先的属性。

表4-5 属性影响节点的后代

属性	描述
xScale,yScale	节点的坐标系通过这两个因素缩放。该属性影响坐标转换、节点的 frame、绘制和碰撞测试。它的后代也同样地缩放。
zRotation	节点的坐标系通过这个因素旋转。该属性影响坐标转换、节点的 frame、绘制和碰撞测试。它的后代也同样地缩放。
alpha	如果该节点是使用混合模式渲染的，混合操作发生之前 alpha 值会乘以任意 alpha 值。它的后代也同样受到影响。
hidden	如果一个节点是隐藏的，它和它的所有后代都不渲染。
speed	一个节点处理动作的速度与该值相乘。它的后代也同样受到影响。

坐标空间之间的转换

在使用节点树时，有时你需要把位置从一个坐标空间转换到另一个。例如，当你指定物理系统中的联合（joints），联合位置被指定在场景坐标。所以，如果你在本地坐标系有那些点，你需要将它们转换为场景的坐标空间。

清单 4-4 展示了如何将一个节点的位置转换到场景坐标系中。场景被要求进行转换。记住一个节点的位置在它父节点的坐标系统中指定，所以代码传递 node.parent 作为要转换的节点。你可以通过调用 convertPoint:toNode:方法执行反向的相同转换。

清单 4-4 转换节点到场景坐标系统

```
CGPoint positionInScene = [node.scene convertPoint:node.position fromnode:node.parent];
```

你需要进行坐标转换的一个情况是在执行事件处理的时候。鼠标和触摸事件需要从 window 坐标转换到视图坐标，并从那里进入场景。为了简化你需要写的代码， Sprite Kit 增加了一些方便的方法：

- 在 iOS 上，使用 `UITouch` 对象的 `locationInNode:`和 `previousLocationInNode:`将触摸位置转换到节点的坐标系。
- 在 OS X 上，使用 `NSEvent` 对象的 `locationInNode:`方法，将鼠标事件转换到节点的坐标系。

使用场景间过渡

场景是游戏的基石。通常情况下，你为游戏的各部分设计自包含（self-contained）的场景，然后在必要时在这些场景之间过渡。例如，你可以创建不同的场景类表现任何或全部下列概念：

- 在其他内容加载时显示的加载场景
- 选择你要玩什么样的游戏的主菜单场景
- 用户选择的特定类型游戏的配置细节的场景
- 提供游戏的场景
- 当游戏结束时显示的场景

当你在一个已经在呈现场景的视图上呈现新的场景时，你有使用旧场景更改到新场景的过渡动画的变化的选项。使用过渡提供了一定的连续性，从而使场景变化不是那么突然。过渡的完整列表在 *SKTransition 类参考*。

两个场景之间的过渡

通常情况下，根据游戏或用户输入过渡到一个新的场景。例如，如果用户在你的主菜单场景中按下一个按钮，你可能会过渡到一个新的场景，来配置玩家想玩的赛事。清单 5-1 展示了你可能在精灵中实现的事件处理程序的方式。处理程序首先在自己身上运行动画来突出按钮（这里不作说明）。然后，它创建一个过渡对象和新的场景。最后，它调用视图来呈现新的景象。过渡意味着这种变化是动画的。

清单 5-1 过渡到一个新的场景

```
- (void)mouseUp:(NSEvent *)theEvent

{

    [self runAction:self.buttonPressAnimation];

    SKTransition *reveal= [SKTransition revealWithDirection:SKTransitionDirectionDown dura
```

```
    GameConfigScene *newScene = [[GameConfigScene alloc] initWithSize:CGSizeMake(1024,768)
/ /可选，插入代码来配置新的场景。
    [self.scene.view presentScene:newScene transition:reveal];
}
```

过渡发生时，scene 属性值可能立即更新为指向新的场景。然后，发生动画。最后，移除对旧场景的强引用。如果你需要在过渡发生后保持场景，你的应用需要保持自身对旧场景的强引用。

组织你的游戏时，创建一个这样的图表是相当有帮助的：它展示了在游戏中的所有场景、场景之间发生的过渡，以及过渡发生时必须传递给新场景的数据。不像在 iOS 中的视图控制器，Sprite Kit 不提供内置的机制传递场景之间的数据。如果你需要在场景过渡时提供数据，你需要实现自己的机制来配置新的场景。通常情况下，这意味着在每个场景中定义你自己的自定义方法和属性，或者在自定义场景类实现的协议中这样做。

配置过渡期间是否播放动画

过渡对象 `pausesIncomingScene` 和 `pausesOutgoingScene` 属性允许你定义动画是否在过渡期间播放。默认情况下，两个场景的动画在过渡期间相继处理。但是，你可能要暂停一个或两个场景，直到过渡完成。例如，重新考虑代码清单 5-1。因为按钮将运行一个动作，这个代码期望流出场景（outgoing scene）会动起来。但也许流入场景（incoming scene）不应该它的内容动起来，直到过渡完成。添加清单 5-2 的代码达到预期的效果。

清单 5-2 过渡期间暂停帧处理

```
reveal.pausesIncomingScene = NO;
```

检测何时呈现场景

有时候你需要能够检测场景已被呈现或从视图中移除。你可以通过在你的自定义场景类上实现一个或两个以下方法这样做：

- 实现 `willMoveFromView` 方法，在场景将要从视图中移除时调用。

- 实现 `didMoveToView` 方法，在场景刚刚由视图完成呈现时调用。

当一个场景没有过渡呈现时，首先移除旧的场景，然后呈现新的场景。当使用过渡时，首先加入新的场景，然后过渡发生，最后旧的场景被移除。

使用其他节点类型

虽然精灵是建立游戏时使用的最重要的元素，Sprite Kit 还提供了许多其他的节点类。这些节点类中的大部分都提供可视化的内容，类似的 SKSpriteNode 类。剩下的则不直接绘制自己的内容，而是修改它们在节点树的后代的行为。表 6-1 列出了所有由 Sprite Kit 提供的节点类，包括你已经熟悉的 SKScene 和 SKSpriteNode 类。

表6-1 Sprite Kit节点类

类	描述
SKNode	所有的节点类都从该类派生。它不绘制任何东西。
SKScene	场景是在节点树的根节点。它控制动画和动作的处理。
SKSpriteNode	绘制纹理精灵的节点。
SKLabelNode	渲染文本字符串的节点。
SKShapeNode	渲染基于 Core Graphics 路径的形状的节点。
SKVideoNode	播放视频内容的节点。
SKEmitterNode	创建和渲染粒子的节点。
SKCropNode	使用遮罩（mask）修剪其子节点的节点。
SKEffectNode	应用 Core Image 滤镜到其子节点的节点。

几乎所有对精灵节点使用的技术都可以应用到其他节点类型。例如，您可以使用动作让屏幕上的其他节点对象动起来，操纵它们的渲染顺序，并在物理模拟内使用它们。请继续阅读来了解如何在你的游戏中使用这些其他节点类。当你对这些类变得熟悉时，你就会明白 Sprite Kit 所有的可视化能力了。然后您就可以开始设计游戏的外观了。

基础节点

SKNode 类不绘制任何可视化内容。它的主要作用是提供其他节点类使用的基础行为。然而，这并不意味着在你的游戏中你不能找到有用的方式使用 SKNode 对象。下面是一些你可能会在你的游戏引擎内使用基础节点的方式：

- 你有一个由多个节点对象组合的内容，无论是精灵或其他内容的节点。不过，你想在你的游戏中把此内容作为一个单独的对象，而不想令其中任何一个内容节点成为根节点。这时用基本节点是合适的，因为你可以给定它在场景树的位置，然后让所有的其他节点作为其后代。这些个别的零部件，也可以相对于父节点的位置进行移动或调整。

- 使用节点对象组织绘制的内容到一系列的层。例如，许多游戏有一个世界（world）的背景层，有另一个角色层，而文本和其他的游戏信息在第三层。其他游戏有更多的层。创建每个层为基本节点，并把它们按顺序插入到场景中。然后，必要时，可以使个别图层可见或不可见。

- 您需要场景中一个不可见的对象，但要它执行一些其他必要的功能。例如，在一个地牢探索游戏，一个不可见的节点可能用来代表一个隐藏的陷阱。当另一个节点与它相交时，就会触发陷阱。（见“搜索物理体”。）或另一个例子，你可能会添加一个节点作为另一个节点的子节点，而后者代表玩家在视图中的点的位置。（请参阅“示例：在节点上中心定位场景”。）

在树中用这样的节点代表这些概念有以下优势：

- 您可以通过添加或删除单个节点来添加或删除整个子树。这让场景管理变得有效率。

- 您可以调整的树中的一个节点的属性，这些属性的效果向下传播到节点的后代。例如，如果在基本节点有精灵作为其子节点，旋转基本节点也将旋转所有精灵内容。

- 您可以利用行动、物理接触和其他 Sprite Kit 的功能来实现此概念。

子类化 SKNode 类是一个非常有用的方式在你的游戏中建立更复杂的行为。请参阅“使用子类化来创建您自己的节点行为。”

标签节点显示文本

几乎每个游戏都需要在某些时候显示文本，即使它只是对玩家显示“游戏结束”。如果你必须自己在 OpenGL 中实现它，需要相当多的工作才能正确完成。但是 Sprite Kit 却很容易！`SKLabelNode` 类完成所有加载字体和创建显示文本所需要的工作。

清单 6-1 演示了如何创建一个新的文本标签。

清单 6-1 添加文本标签

```
SKLabelNode *winner = [SKLabelNode labelNodeWithFontNamed:@“Chalkduster”];

winner.text = “You Win!”

winner.fontSize = 65;

winner.fontColor = [SKColor greenColor];

winner.position = CGPointMake(CGRectGetMidX(self.bounds),

                              CGRectGetMidY(self.bounds));

[self addChild:winner];
```

每次你更改标签节点的属性后，标签节点会在下一次渲染场景时自动更新。

形状节点绘制基于路径的形状

`SKShapeNode` 类绘制一个标准的 Core Graphics 路径。图形路径是可以定义开放或封闭的子路径的直线和曲线的集合。形状节点包含单独的属性来指定线条的颜色和内部填充的颜色。

Shape 节点对于不能很容易地分解成纹理精灵的内容是有用的。纹理精灵比形状节点提供更高的性能，所以应在你的游戏引擎谨慎使用它们。然而，形状节点对于在你的游戏内容之上构建和显示调试信息是非常有用的。

清单 6-2 展示了如何创建一个形状节点的例子。该示例创建一个蓝色填充色和白色边线的圆圈。路径被创建并附加到形状节点的 `path` 属性。

清单 6-2 通过路径创建一个形状节点

```
SKShapeNode *ball = [[SKShapeNode alloc] init];



CGMutablePathRef myPath = CGPathCreateMutable();

CGPathAddArc(MYPATH, NULL, 0.0, 15.0, M_PI*2, YES);

ball.path = myPath;



ball.lineWidth = 1.0;

ball.fillColor = [SKColor blueColor];

ball.strokeColor = [SKColor whiteColor];

ball.glowWidth = 0.5;
```

从代码中你可以看到形状有三个基本要素：

- 形状的内部填充。`fillColor` 属性指定了用来填充内部的颜色。
- 形状的边线渲染为一条线。`strokeColor` 和 `lineWidth` 属性定义线条的笔触。
- 从边线扩展的光晕（glow）。`glowWidth` 和 `strokeColor` 属性定义光晕。

你可以通过设置其颜色为`[SKColor clearColor]` 禁用任何这些元素。

形状节点提供了一些属性让你控制形状如何混合到帧缓存（framebuffer）中。这些属性的使用方式与 `SKSpriteNode` 类的属性一样。请参阅“混合精灵到帧缓冲中。”

视频节点播放电影

`SKVideoNode` 类使用 AV Foundation 框架显示电影内容。与任何其他节点一样，你可以把电影的节点放在节点树内的任何地方，Sprite Kit 会正确渲染它。例如，某些用动作定义可能代价高昂的可视化行为，你可能会使用视频节点让它动起来。

视频节点与精灵节点类似，但只提供功能的一个子集：

- `size` 属性被初始化成视频内容的基本尺寸，但如果你愿意，你可以改变它。视频内容将自动拉伸到新的尺寸。
- `anchorPoint` 属性定义了内容相对节点位置在什么地方显示。

然而，应遵循以下限制：

- 视频节点总是被均匀拉伸。
- 视频节点不能被着色。
- 视频节点总是使用 alpha 混合模式。

像大部分的节点类那样，创建一个视频节点非常简单。清单 6-3 展示了一个典型的使用模式。它使用存储在应用程序 bundle 中的视频初始化视频节点，然后把节点添加到场景。调用节点的的 `play` 方法来启动视频播放。

清单 6-3 在场景中显示视频

```
SKVideoNode *sample = [SKVideoNode videoNodeWithVideoFileNamed:@“sample.m4v”];

sample.position = CGPointMake(CGRectGetMidX(self.frame),

                              CGRectGetMidY(self.frame));

[self addChild:sample];

[sample play];
```

节点的 `play` 和 `pause` 方法让你可以控制播放。

如果你需要更精确地控制视频的播放行为，你可以使用 AV Foundation 从你的视频内容创建 `AVPlayer` 对象，然后使用这个对象初始化视频节点。然后，使用 `AVPlayer` 对象来控制播放，而不是使用节点的播放方法。视频内容将自动显示在视频节点中。欲了解更多信息，请参阅 *AV Foundation 编程指南*。

发射器节点创建粒子特效

当一个 `SKEmitterNode` 对象被放置在场景中时，它会自动创建并渲染新的粒子。你可以用发射器节点来自动创建特殊效果，包括下雨、爆炸或发射。

粒子类似于 `SKSpriteNode` 对象，它渲染有纹理或无纹理的图像，图像有尺寸、有颜色、且可以混合到场景。但是，粒子在两个重要的方面与精灵不同：

- 粒子的纹理总是均匀拉伸。
- 粒子不能用 Sprite Kit 中的对象表示（represented）。这意味着你不可以对粒子执行节点相关的任务，也不能给粒子关联物理体使它们与其他内容交互。

粒子是纯粹的可视化对象，他们的行为完全由创建它们的发射器节点定义。发射节点包含很多属性来控制它生成的粒子的行为，包括：

- 粒子的出生率和寿命。你还可以指定发射器自行关闭前能产生的粒子的最大数量。
- 粒子的初始值，包括它的位置、方向、颜色和尺寸。这些初始值通常是随机的。
- 在粒子生命期内应用到在粒子的变化。通常，这些被指定为一个随时间的变化率（rate-of-change）。例如，你可以指定一个粒子以特定的速度旋转，以弧度每秒为单位。发射器每一帧都自动更新粒子的数据。在大多数情况下，你还可以使用关键帧序列（keyframe sequences）创建更复杂的行为。例如，你可以为一个粒子指定一个关键帧序列，让它出来时很小，放大到较大的尺寸，然后在死亡前收缩。

使用粒子发射器编辑器与发射器进行实验

在大多数情况下，你永远不需要在你的游戏中直接配置发射器节点。相反，你应使用 Xcode 来配置发射器节点的属性。当你改变发射器节点的行为，Xcode 立即为你提供一个更新过的视觉效果。一旦完成之后，Xcode 可以归档（achieve）配置好的发射器。然后，在运行时，你的游戏使用此归档来实现实例化一个新的发射器节点。

使用 Xcode 创建你的发射器节点有几个重要的优势：

- 这是学习发射器类的能力的最好方式。
- 你可以更迅速地试验新的粒子效果并立即看到结果。

- 你把粒子效果的设计任务从使用它的编程任务中分离出来。这使得你的美工可以独立于你的游戏代码继续创作新的粒子效果。

有关使用 Xcode 创建粒子效果的更多信息，请参阅*粒子发射器编辑器指南*。

清单 6-4 展示了如何加载由 Xcode 创建的粒子效果。所有粒子效果都使用 Cocoa 的标准归档机制保存，所以代码首先创建烟雾效果的一个路径，然后加载归档。

清单 6-4 从文件加载粒子效果

```
- (SKEmitterNode *)newSmokeEmitter

{

    NSString *smokePath = [NSBundle mainBundle] pathForResource:@“smoke” ofType:“skn”];

    SKEmitterNode *smoke = [NSKeyedUnarchiver unarchiveObjectWithFile:smokePath];

    return smoke;

}
```

手动配置发射器如何创建新的粒子

SKEmitterNode 类提供了许多属性配置发射器节点的行为。事实上，Xcode inspector 简单地设置这些属性的值。但是，你可以手工创建发射器节点并配置这些属性，或者你也可以从归档创建一个发射器节点并改变其属性值。例如，假设一下，你使用清单 6-4 中的烟雾效果来展示对火箭飞船的损坏。随着船受到更多的损坏，你可以提高发射器的出生率来添加更多的烟雾。
用于配置发射器节点的属性的完整列表在 *SKEmitterNode 类参考*中描述。然而，首先理解如何创建新的粒子，其次理解一个典型粒子的属性如何在发射器节点中指定，对你是有用的。

只要发射器节点在场景中，它就会发射新粒子。你使用以下属性定义它要创建多少粒子：

- `particleBirthRate` 属性指定发射器每秒创建的粒子数。
- `numParticlesToEmit` 属性指定发射器自行关闭之前要创建多少粒子。发射器还可以被配置为产生无限数量的粒子。

当粒子被创建时，它的初始属性值是由发射器的属性决定的。对于每个粒子属性，发射器类声明以下四个属性：

- 属性的平均初始（starting）值。
- 属性值的随机范围。每次发射一个新的粒子，会在该范围内计算一个新的随机值。
- 随时间的变化率，也被称为属性的*速度*。并非所有属性都有一个速度属性。
- 一个可选的关键帧序列。

清单 6-6 展示了你可能如何配置发射器的 `scale` 属性。这是节点的 `xScale` 和 `yScale` 属性的一个简化版本，并确定粒子相比它纹理的尺寸有多大。

清单 6-5 配置粒子的 scale 属性

```
myEmitter.particleScale = 0.3;

myEmitter.particleScaleRange = 0.2;

myEmitter.particleScaleSpeed = -0.1;
```

当创建一个新的粒子，其 scale 值是从 `0.2` 到 `0.4` 的一个随机数。然后 scale 值以每秒 `0.1` 的速度减少。所以，如果一个特定的粒子以平均值开始，即 `0.3`，它会在 3 秒内从 `0.3` 减少到 `0`。

使用关键帧序列配置粒子属性的自定义坡道

关键帧序列用来为粒子属性提供更复杂的行为。一个关键帧序列，使你可以指定粒子生命期的多个点，并在每个点为属性指定一个值。然后关键帧序列篡改（interpolate）这些点之间的值，并用它们来模拟粒子的属性值。

你可以使用关键帧序列，实现了许多自定义的行为，包括：

- 更改属性值，直到它达到某个指定值。
- 在粒子的整个生命期中使用多个不同的属性值。例如，你可能会在序列中的一部分增加属性的值，而在另一部分减小属性的值。或者，在指定颜色时，你可以指定粒子在其生命期中循环显示多种颜色。
- 使用一个非线性的曲线或分级（stepping）功能改变属性值。

清单 6-6 展示你可以如何替换清单 6-5 中的代码来使用序列。当你使用一个序列，值不是随机化的。相反，序列指定所有的属性值。每个关键帧值包含一个值对象和时间戳。时间戳在 0 到 1.0 的范围内指定，其中 0 表示粒子的诞生而 1.0 表示它的死亡。因此，在该序列中，粒子以 0.2 的拉伸比例开始并在序列的四分之一时增加到 0.7。到序列的四分之三时，达到它的最小尺寸 0.1。它保持这个尺寸直到死亡。

清单 6-6 使用序列来改变粒子的尺度属性

```
SKKeyframeSequence * scaleSequence = [[SKKeyframeSequence alloc]
initWithKeyframeValues:@[@0.2,@0.7,@0.1 times:@[@0.0,@0.250,@0.75];

myEmitter.particleScaleSequence = scaleSequence;
```

给粒子添加动作

虽然你没有能力直接访问由 Sprite Kit 创建的粒子，但是你可以指定一个所有粒子都执行的动作。每当创建新的粒子，粒子发射器告诉该粒子执行动作。你使用动作创建的行为，甚至可以比序列所允许的更复杂。

在粒子上使用动作的目的，是你可以把粒子看作是一个精灵。这意味着你可以执行其他有趣的技巧，如让粒子的纹理动起来。

使用目标节点更改粒子的目的地

当发射器创建粒子时，它们被渲染成发射节点的子节点。这意味着它们获得发射器节点的所有特性。所以，如果你旋转发射器节点，所有已产生的粒子的位置也会跟着旋转。根据你使用发射器所模拟的东西，这未必是正确的行为。例如，假设你要使用发射器节点来创建火箭的排气。当引擎在全速燃烧，一个锥形的火焰应该在飞船后面喷出来。这用粒子模拟是很容易的。但是，如果粒子是相对于船渲染的，船转弯时排气也会跟着旋转。那样看起来不对。你真正想要的是粒子产生后，就独立于发射器节点。当发射器节点旋转时，新的粒子有新的方向，而旧的粒子仍保持其原来的方向。你要用目标节点来实现它。

清单 6-7 展示了如何使用目标节点来配置火箭的排气效果。当自定义精灵节点类实例化排气节点时，它使排气节点成为它本身的子节点。然而，它使粒子重定向到场景。

清单 6-7 使用目标节点来重定向产生粒子的地方

```
- (void)newExhaustNode

{

    SKEmitterNode *emitter = [NSKeyedUnarchiver unarchiveObjectWithFile:
    [NSBundle mainBundle] pathForResource:@“exhaust” ofType:@“sks”];



/ /发射器放置在船的后部。

   emitter.position = CGPointMake(0,-40);

   emitter.name = @“exhaust”;

/ /发送粒子到场景。

   emitter.targetNode = self.scene;



   [self addChild:emitter];

}
```

当发射器有一个目标节点时，它计算粒子的位置、速度和方向，正如它是该精灵节点的子节点那样。这意味着，如果飞船精灵旋转，排气方向也会自动旋转。然而，一旦这些值计算好，它们被转换到目标节点的坐标系。此后，他们将只受场景节点的属性变化影响。

粒子发射器提示

Sprite Kit 中的粒子发射器是构建可视化效果最有力的工具之一。但是，使用不当的话，粒子发射器可能会成为你的应用程序的设计和实施的瓶颈。考虑下面的提示：

- 使用 Xcode 来创建和测试你的粒子效果，然后在你的游戏中加载归档。
- 在你的游戏的代码里有节制地调整发射器的属性。通常情况下，你这样做是为了指定那些不能由 Xcode inspector 指定的属性，或者在你的游戏逻辑里控制属性。

- 粒子比精灵节点代价低廉（cheaper），但他们仍然有开销！所以，你应该尽量保持屏幕上的粒子的数量降到最低。尽量以低出生率创建粒子效果，并在粒子一旦不可见时杀掉它们。例如，不是每秒创建数百或数千的粒子，而是降低粒子的出生率并稍微增加粒子的尺寸。通常，你可以用更少的粒子创建效果，但获得同样的净视觉外观（net visual appearance）。

- 除非没有另一种解决方案时，才在粒子上使用动作。在单个（individual）粒子上执行动作可能会是非常昂贵的，特别是如果该粒子发射器还具有较高的出生率的话。

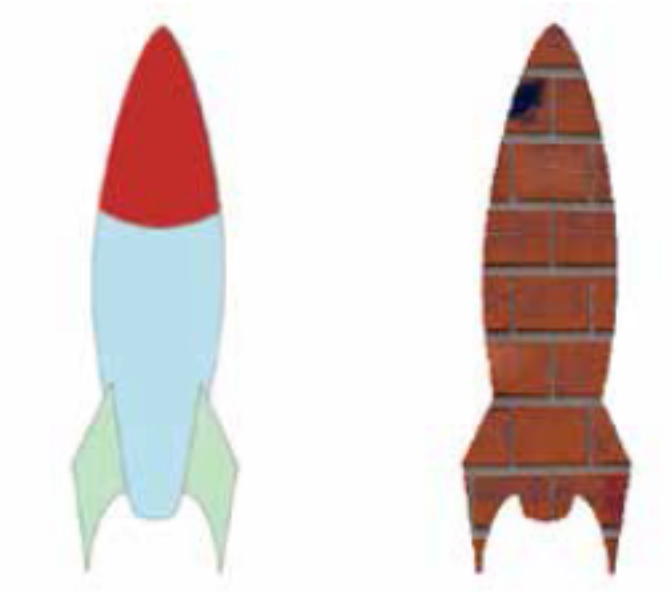
- 每当粒子在产生后就应该独立于发射器节点时，给它们指定目标节点。这通常是发生在发射节点要在场景中移动或旋转的情况下。

- 考虑当粒子发射器在屏幕上不可见时，把它从场景中移除。只在它变得可见前添加它。

切割节点遮罩部分的场景

SKCropNode 对象不像精灵节点那样直接渲染内容。相反，它改变了它的子节点被渲染时的行为。切割节点允许你裁剪部分由子节点渲染的内容。这使得切割节点对于实现驾驶舱（cockpit）视图、控件和其他游戏的指示器、以及任何子节点不应该绘制在场景的一个特定区域之外的效果，是很有用的。图 6-1 简单地使用火箭飞船美术对另一个绘制在场景中的精灵应用遮罩（mask）。

图6-1　切割节点进行遮罩操作



裁剪区域通过遮罩指定。该遮罩不是一个固定的图像。它由一个节点渲染，就像 Sprite Kit 中的任何其他内容。这对简单的口罩和更复杂的行为都是允许的。例如，这里有一些可能你会用来指定一个遮罩的方法：

- 无纹理精灵创建一个遮罩，该遮罩限制内容为场景的一个矩形部分。
- 纹理感精灵是一个像素级精确的遮罩。但也要考虑一个非均匀缩放纹理的好处。这可以允许你创建可以调整尺寸的任意形状的对象。遮罩被调整尺寸和缩放以创建形状，然后动态内容在该遮罩内绘制。
- 可以动态地生成一个复杂遮罩的节点的集合，该遮罩会在每次帧渲染时改变。

清单 6-8 展示了遮罩的简单使用。它用应用程序 bundle 中的纹理加载遮罩图像。然后部分场景的内容被渲染，使用遮罩防止它过度绘制（overdrawing）屏幕中游戏用来显示控件的部分。

清单 6-8 创建切割节点

```
SKCropNode * cropNode = [[SKCropNode alloc] init];

myCropNode.position = CGPointMale(CGRectGetMidX(self.bounds),

                                  CGRectGetMidY(self.bounds));



cropNode.maskNode = [[SKSpriteNode alloc] initWithImageNamed:@“cockpitMask”];

[cropNode addChild:gamePlayNode];

[self addChild:cropNode];

[self addChild:gameControlNodes];
```

当切割节点在渲染时，遮罩在绘制其后代前先渲染。只与最终遮罩的 alpha 分量有关。遮罩中任何 alpha 值为 `0.05` 或更高的像素会呈现。其余的像素会被裁剪。

效果节点对它们的后代应用特效

`SKEffectNode` 自身不绘制内容。相反，每次使用效果节点渲染新的帧时，效果节点执行一个特效传递给它内容。这个传递经过以下步骤：

1. 效果节点执行一个单独的绘图传递（drawing pass）来渲染其子节点到一个私有帧缓冲区（private framebuffer）。

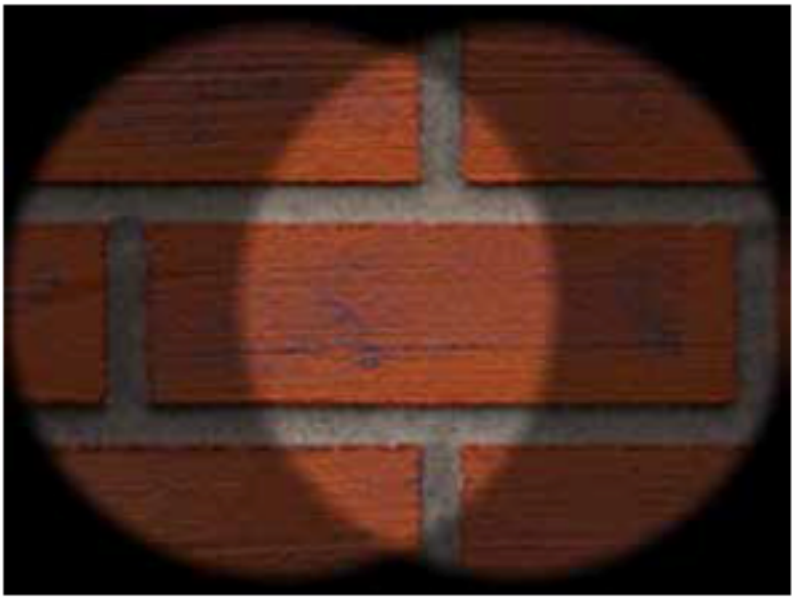
2. 它应用 Core Image 效果到私有帧缓冲区。这个阶段是可选的。

3. 然后它混合它的私有帧缓冲区的内容到它父节点的帧缓冲区，使用标准的精灵混合模式之一。

4. 它丢弃其私人的 framebuffer。

图 6-2 展示了效果节点的一个可能的用法。在这个例子中，效果节点的子节点是两个作为灯光节点的精灵。它积累这些灯的效果，应用模糊滤镜来柔化产生的图像，然后使用复合混合模式（multiply blend mode）来应用这个照明到墙壁纹理上。

图6-2 效果节点应用特效到节点的子节点



这里是灯光效果如何产生的过程：

1. 场景中包含两个不同的节点。第一个是表示地面的纹理精灵。第二个是应用灯光的效果节点。

```
self.lightingNode = [[SKEffectNode alloc] init];
```

2. 灯光是效果节点的子节点。每个都使用附加混合模式来渲染。

```
SKSpriteNode *light = [SKSpriteNode spriteWithTexture:lightTexture];

light.blendMode = SKBlendModeAdd;
```

```
...

[self.lightingNode addChild:light];
```

3. 应用滤镜来柔化灯光。

   如果你指定一个 Core Image 滤镜，它必须是一个接收单一的输入图像并生成单一的输出图像的滤镜。

```
-  (CIFilter *)blurFilter {
     CIFilter *filter= [CIFilter filterWithName:@“CIBoxBlur”] // 3
     [filter setDefaults];
     [filter setValue:[NSNumber numberWithFloat:0] forKey:@“inputRadius”];
     return flter;
 }



self.lightingNode.filter = [self blurFilter];
```

4. 效果节点使用一个复合混合模式来应用照明效果。

```
self.lightingNode.blendMode = SKBlendModeMultiply;
```

场景是效果节点

你已经学了很多关于 SKScene 类的东西，但你可能没有注意到，它是 SKEffectNode 的一个子类！这意味着，任何场景可以对内容应用滤镜。虽然应用滤镜花销可能会非常昂贵（不是所有滤镜都为交互效果精心设计），试验可以帮助你找到一些有趣的方式来使用滤镜。

使用缓存来提高静态内容的性能

效果节点通常渲染其内容作为绘制帧的一部分，然后丢弃它们。渲染内容是必要的，因为我们假设内容是每帧都改变的。但是，重新创建这些内容并应用 Core Image 滤镜的成本可能会非常高。如果内容是静态的，那么这是不必要的。保持渲染的帧缓冲区，而不是抛弃它，可能会更有

意义。如果效果节点的内容是静态的，你可以把节点的 `shouldRasterize` 属性设置为 `YES`。设置此属性将导致以下行为的改变：

- 帧缓冲区在光栅化（rasterization）的末尾不会被丢弃。这也意味着效果节点正在使用更多的内存，而渲染可能需要稍长的时间。
- 当一个新的帧被渲染时，帧缓冲区仅在效果节点的后代的内容已经改变后才会被渲染。
- 更改 Core Image 滤镜的属性不再导致帧缓冲区的自动更新。你可以通过设置 `shouldRasterize` 的属性为 `NO` 强制它更新。

高级场景处理

使用 Sprite Kit 涉及到操纵场景树的内容来让内容在屏幕上的动起来。通常情况下，动作是该系统的核心。然而，通过直接地挂接到（hooking into）场景处理，你可以创建动作不能单独完成的其他行为。要做到这一点，你需要学习：

- 场景如何处理动画
- 如何在场景处理过程中添加自己的行为

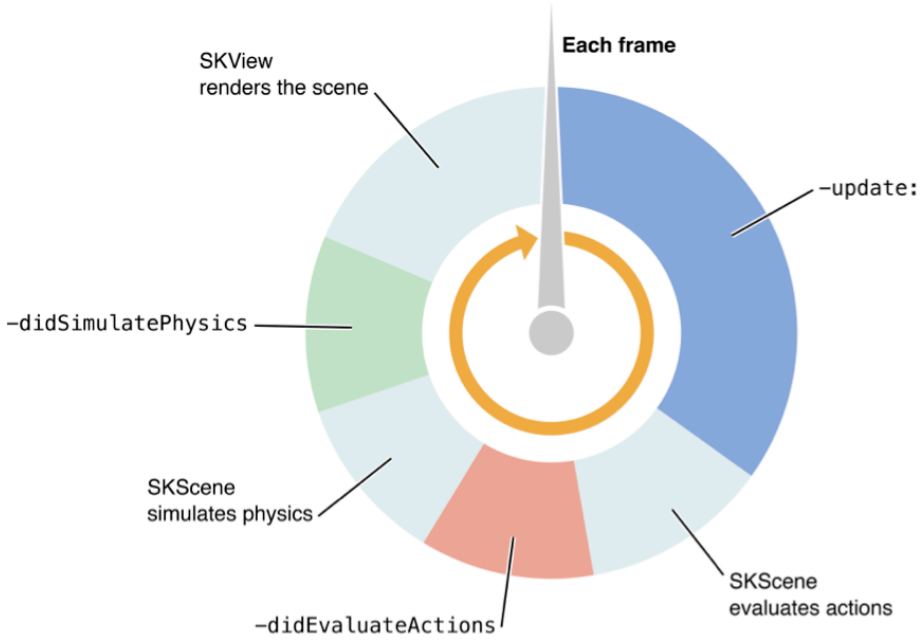
场景如何处理动画帧

在传统视图系统中，视图内容渲染一次后，然后只有当模型（model）的内容发生变化时才会再次渲染。这种模式对于视图非常适用，因为在实践中，大多数视图内容是静态的。另一方面，Sprite Kit 是明确为动态内容设计的。Sprite Kit 不断更新的场景内容并渲染它，以确保动画是平滑和精确的。

动画和渲染场景的过程绑定到场景对象（`SKScene`）上。场景和动作处理只在场景被呈现时运行。呈现的场景运行一个渲染循环，该循环在处理场景的节点树和渲染它之间交替进行。这种模式类似于在大多数游戏中使用的渲染和处理循环。

图 7-1 展示了场景执行渲染循环的步骤。

图 7-1 场景中的帧处理



每次通过的最终目标是要渲染并更新场景的节点树的内容。你不直接挂接到渲染步骤，而是更新节点树的内容，如在“建筑场景”中描述的那样。然而，Sprite Kit 为你提供了工具来挂接到其他步骤。下面是那些步骤：

1. 随着时间在模拟中流逝，渲染循环从调用场景的 `update:` 方法开始。这是实现你自己游戏内模拟（in-game simulation）的主要场所，包括输入处理、人工智能、游戏脚本和其他类似的游戏逻辑。通常情况下，你使用这个方法对节点进行更改或运行节点上的动作。

2. 场景处理树中的所有节点上的动作。它找到任何正在运行的操作，并将那些更改应用到树上。在实践中，因为自定义操作，你还可以挂接到动作进程（mechanism）调用你自己的代码。

   你不能直接控制动作的处理的顺序，也不能让场景跳过某个节点上的动作，除非你从这些节点中移除动作或从节点树中移除节点。

3. 在帧的所有动作都已处理后，场景的 `didEvaluateActions` 方法被调用。

4. 然后场景对场景中的物理体模拟物理。添加物理到场景中在“模拟物理”中描述，但模拟物理的最终结果是，物理模拟可能会调节树中节点的位置和旋转角度。你的游戏也可以在物理体之间互相接触时接收到回调。

5. 场景的 `didSimulatePhysics` 方法是场景渲染前的最后一个步骤。这是你对场景进行更改的最后机会。

6. 渲染场景。

场景中的后处理

场景可以以任意顺序处理场景树中的动作。由于这个原因，如果你有一些需要在每一帧运行的任务，且你需要在它们运行时精确地控制它们，你应该使用 `didEvaluateActions` 和 `didSimulatePhysics` 方法来执行这些任务。通常情况下，你在这里进行的更改，需要树中的某些节点的最终计算出来的位置。你的后处理（post-processing）可以利用这些位置，并在树上执行其他有用的工作。

下面是一些你可能执行的后处理任务的例子：

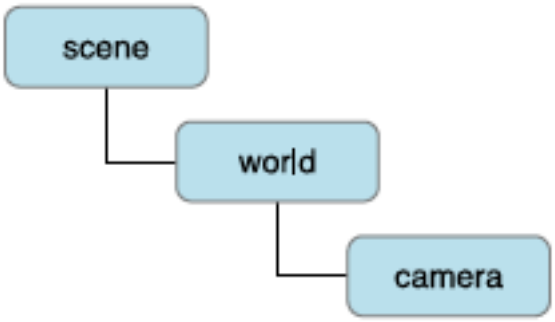
- 居中场景的内容在某个节点上
- 在场景内容上添加调试信息的覆盖层（overlay）
- 复制节点树的某一部分的节点到另一部分。例如，你可能有一个效果节点，它的子节点需要跟树中其他的节点一样。

例子：居中场景在节点上

在需要内容滚动的游戏中，居中场景的内容在某个节点上，对相机（cameras）和类似的概念是有用的。在这种情况下，内容比场景的 frame 要大。在玩家左右移动时，角色在某个地方保持固定，而世界（world）在他们周围移动。场景保持对角色锁定，无论玩家把角色移动到哪里。

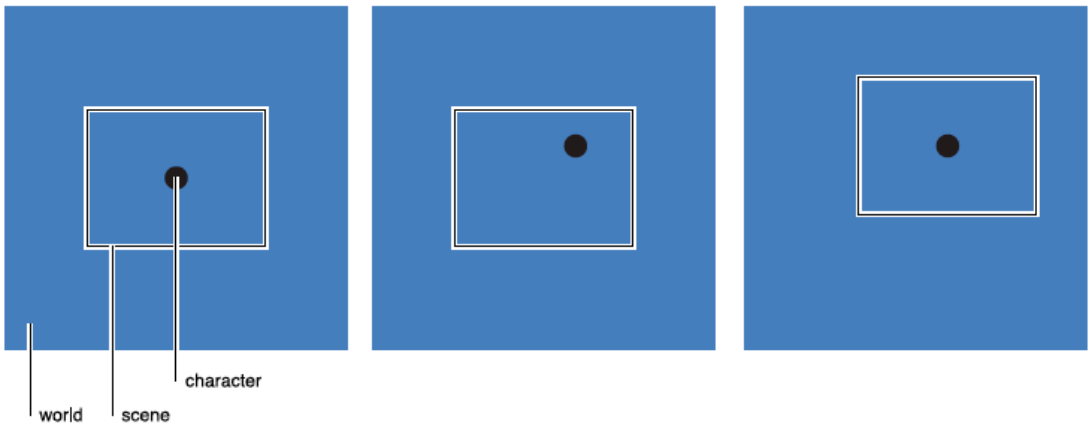
Sprite 工具并不提供相机的内置支持，但实现却是非常简单。世界和相机分别由场景中的一个节点表示。世界是场景的一个直接子节点，而相机是一个世界节点的后代。节点的这种安排是非常有用的，因为它给游戏世界一个不绑定于场景坐标系的坐标系。你可以使用这个坐标系布置世界内容。

图 7-2　为一个滚动的世界组织场景



相机被放置在世界中。世界可以在场景中四处滑动。因为相机是一个节点，你可以用动作甚至是物理来移动相机。然后，在后处理步骤，你重新在场景中定位世界节点，使相机节点在场景正中。图 7-3 展示了这个。世界放置在场景中，以便让角色居中。然后，该角色在世界里面四处移动。最后，后处理步骤重定位世界以便角色再次居中。

图 7-3 世界在场景内移动



下面是实现的片段：

1. 把场景的锚点放置在场景中心。

```
self.anchorPoint = CGPointMake(0.5,0.5);
```

2. 使用世界节点来表示滚动的世界。世界的内容将作为世界的子节点，将是精灵节点和内容节点。（图中未示出）。

```
SKNode *myWorld = [SKNode node];

[self addChild:myWorld];
```

3. 使用世界内的一个节点表示相机。

```
SKNode *camera = [SKNode node];

camera.name = @“camera”;

[myWorld addChild:camera];
```

4. 使用 didSimulatePhysics 方法居中场景在相机上。centerOnNode 方法把相机当前的位置转换成场景坐标，然后从世界的位置减去那些坐标来滑动角色到（0,0）位置。

```
 - (void)didSimulatePhysics {
    [self centerOnNode:[self childNodeWithName:@“//camera”]];
}



  - (void)centerOnNode:(SKNode *)node

 {

     CGPoint cameraPositionInScene = [node.scene convertPoint:node.position
      fromNode:node.parent];

     node.parent.position = CGPointMake(node.parent.position.x  -  cameraPositionInSce
     node.parent.position.y  -  cameraPositionInScene.y);

 }
```

例子：添加调试覆盖层

当你正在使用 Sprite Kit 开发一个游戏时，显示实时的可视化调试信息是有帮助的，这些信息关于场景中正在发生的事情。例如，以下信息可能在调试你的游戏时是有用的：

- 场景中角色的人工智能决策
- 在世界中触发脚本动作的位置
- 物理信息，如重力以及其他力，甚至物理体的尺寸

形状和标签节点对注释你的游戏的行为特别有用。

要添加一个调试覆盖层，最好的办法是使用一个单一的节点代表覆盖层，并把它添加到场景。所有调试信息由这个节点的后代节点表示。这个节点被放置在场景中附带一个 z 坐标，z 坐标把它放置在所有其他场景内容的上面。调试节点可以是但不必是场景的直接子节点。例如，在一个滚动的世界，你放置在覆盖层的信息可能绑定于世界坐标，而不是场景坐标。

在场景开始处理一帧之前，你从场景中移除这个节点，然后移除它的子节点。其假设是，调试信息需要每帧更新。经过场景模拟物理后，节点被添加回到场景且它的内容会重新生成。

1. 创建场景类中的调试节点的属性，并在场景第一次呈现时初始化它。

```
@property(SKNode *)debugOverlay;



self.debugOverlay = [SKNode node];

[self addChild:self.debugOverlay];
```

2. 在动作处理前移除节点。

```
 - (void)update:(NSTimeInterval)currentTime

{

   [self.debugOverlay removeFromParent];

   [self.debugOverlay removeAllChildren];

}
```

3. 在场景处理完后添加节点。

```
 - (void)didSimulatePhysics

{

   [self addChild:self.debugOverlay];

   / /添加代码来创建调试节点并添加调试图像到调试节点。

   / /这个例子显示了重力矢量。

   SKShapeNode *gravityLine = [[SKShapeNode alloc] init];

   gravityLine.position = CGPointMAke(200,200);
```

```
    CGMutablePathRef path= CGPathCreateMutable();

    CGPathMoveToPoint(path, NULL, 0.0, 0.0);

    CGPathAddLineToPoint(path, self.physicsWorld.gravity.x * 10,
    self.physicsWorld.gravity.y * 10);

    CGPathCloseSubpath(path);

    gravityLine.path = path;

    CGPathRelease(path);



    [self.debugOverlay addChild:gravityLine];
}
```

例子：在场景中复制信息

实现此行为的技术类似与添加调试覆盖层。当你预处理（pre-process）场景时，从树中移除节点过时的副本。然后，在后处理过程中，你从树的一部分复制节点到另一部分。你使用节点的副本，因为每个节点只能有一个父节点。

在某些情况下，你只需要在每一帧更新少量的信息。在这种情况下，不断地添加、移除和复制节点的成本可能是昂贵的。相反，只复制一次，然后用你的后处理步骤来更新重要的属性。

```
copyNode.position = originalNode.position;
```

模拟物理

Sprite Kit 中的物理模拟通过添加物理体场景来进行。**物理体**是一个模拟的物理对象，该对象连接到场景的节点树中的节点。它使用节点的位置和方向把它自身放置在模拟中。每一个物理体具有其他定义模拟如何操作它的特性。这些属性包括物理对象的先天属性，如它的质量或密度，也包括施加于它的属性，如它的速度。这些特性定义了主体如何移动，它在模拟中是如何受到力的影响，以及它是如何响应与其他物理体的碰撞。

每次场景计算一个新的动画帧，它模拟连接到节点树的物理体上的力和碰撞的作用。它为每个物理体计算最终的位置、方向和速度。然后，场景更新每个相应节点的位置和旋转角度。

要在你的游戏中使用物理，你需要：

- 将物理体附加到节点树中的节点上。请参阅“所有物理都在物理体上模拟”。
- 配置物理体的物理属性。请参阅“配置物理体的物理属性”。
- 定义场景的物理模拟的全局特点，如重力。请参阅“配置物理世界。”
- 在游戏需要支持的地方，设置场景中的物理体的速度或对它们施加力或动量（impulses）。请参阅“让物理体移动”。
- 定义场景中的物理体相互接触时如何交互。请参阅“使用碰撞和接触”。
- 优化你的物理模拟来限制它必须执行的计算的数量。请参阅“在游戏中使用物理的提示和技巧。”

Sprite Kit 使用国际单位制，也被称为 SI，或米–千克–秒体制。在必要的情况下，你可能需要查阅其他在线参考资料以了解更多有关 Sprite Kit 使用的物理方程式的知识。

所有物理都在物理体上模拟

`SKPhysicsBody` 对象定义系统中的物理体的形状和模拟参数。当场景模拟物理时，它对所有连接到场景树的物理体执行计算。所以，你创建一个 `SKPhysicsBody` 对象，配置其属性，然后将其赋值给节点的 `physicsBody` 属性。

物理体有三种：

- **动态体积**（dynamic volume）模拟系统中一个有体积和质量、可以受力和碰撞作用的物理对象。使用动态体积表示场景中需要移动和互相碰撞的项目。

- **静态体积**（static volume）与动态体积相似，但它的速度将被忽略，且它不受力或碰撞的作用。然而，因为它仍然具有体积，其他的对象可以弹开它或与它进行交互。使用静态体积表示场景中占用空间但不应该被模拟移动的项目。例如，你可以使用静态体积表示一个迷宫的墙壁。

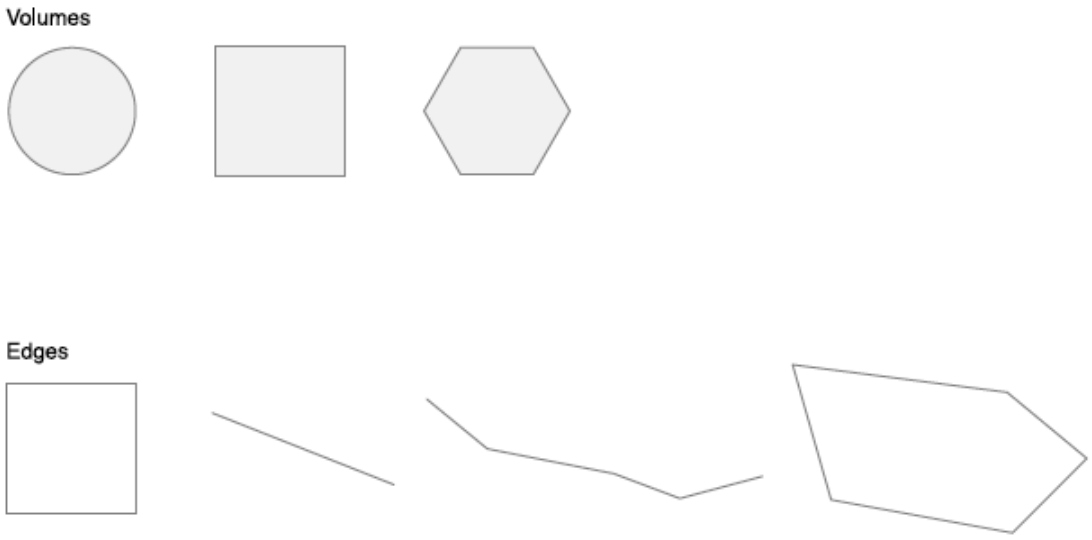
  把静态和动态体积划分为不同的实体是有用的，在实践中，这是可以适用于任何基于体积的物理体的两种不同的模式。这可以是有用的，因为它可以让你选择性地启用或禁用对主体的作用。

- **边**（edge）是一个静态无体积的主体。边从来不会被模拟移动，且它们的质量并不重要。边被用于表示场景内的负空间（negative space）（如另一实体中的空心点）或一条不可逾越且不可见的微薄的边界。例如，边经常用于表示场景的边界。

  边和体积之间的主要区别是，边允许在其自身的边界内的移动，而体积被认为是一个固态物体。如果边通过其他方法移动，它们只会与体积交互，而不是与其他的边。

Sprite Kit 提供了一些标准的形状，以及基于任意路径的形状。图 8–1 展示可用的形状。

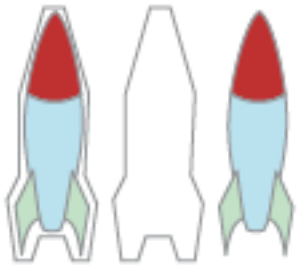
图 8–1 物理体



使用匹配图形表现的物理形状

在大多数情况下，一个物理体应具有尺寸和形状，其值接近对应的节点的可视化表现。例如，在图 8–2 中，火箭具有一个窄的形状，用圆形或者矩形都不能很好地表现的。选择一个凸多边形形状能与精灵的插图相匹配。

图8-2 匹配形状与接近的表现



然而，在为你的物理体选择一个形状时，不要过于精确。更复杂的形状，需要更多的工作来正确模拟。对于基于体积的主体，请遵循下列准则：

- 圆是最高效的形状。
- 基于路径的多边形是最低效的，且计算工作随着多边形的复杂度成倍增加。

基于边的主体的计算比基于体积的主体开销更昂贵。这是因为与它交互的主体可能是开放边的任一边，或者封闭形状的里面或外面。使用这些准则：

- 线条和矩形是最有效的基于边的主体。
- 边回路（edge loops）和边链（edge chains）是最昂贵的，且计算工作随着路径的复杂度成倍增加。

创建物理体

通过调用 SKPhysicsBody 类的方法之一来创建一个物理体。每个类的方法定义要创建基于体积还是基于边的主体，以及它有什么样的形状。

在场景周围创建边回路

清单 8-1 所示的代码，经常用于不需要滚动内容的游戏中。在这种情况下，游戏希望物理体击中场景边界而反弹回到游戏区。

清单 8-1 场景边界

```
- (void)createSceneContents
```

```
{
    self.backgroundColor = [SKColor blackColor];
    self.scaleMode = SKSceneScaleModeAspectFit;
    self.physicsBody = [SKPhysicsBody bodyWithEdgeLoopFromRect:self.frame];
}
```

为精灵创建圆形体积

清单 8-2 展示的代码，为球形或圆形对象创建物理体。由于物理体被附加到一个精灵对象上，它通常需要体积。在这种情况下，假定精灵图像的锚点接近圆心，然后计算圆的半径并用于创建物理体。

清单 8-2 一个圆形精灵的物理体

```
SKSpriteNode *Sprite= [SKSpriteNode spriteWithImage:@“sphere.png”];
sprite.physicsBody = [SKPhysicsBody bodyWithCircleOfRadius:sprite.size.width/2];
sprite.physicsBody.dynamic = YES;
```

如果物理体显着小于精灵的图像，用于创建物理体的数据可能需要由另外一些源提供，如一个属性列表。见“Sprite Kit 最佳实践”。

配置物理体的物理属性

`SKPhysicsBody` 类定义了一些用来确定如何模拟物理体的属性。这些属性会影响到主体对力如何反应、它本身能生成何种力（模拟摩擦）以及它对场景中的碰撞如何反应。在大多数情况下，属性是用来模拟物理效果的。

每个单独的主体也有其自身的属性值来确定它对场景中的力和碰撞如何作出反应。下面是最重要的属性：

- `mass` 属性决定力是如何影响主体，以及当主体参与碰撞时它有多大的动量。

- friction 属性决定了主体表面的粗糙度。它被用来计算一个主体沿其他主体表面移动时产生的摩擦力。
- linearDamping 和 angularDamping 属性是用来计算主体在世界中移动时的摩擦。例如，它可能用于模拟空气或水的摩擦。
- restitution 属性决定主体在碰撞过程中保持能多少能量，即它的弹力。

其他的属性被用来决定模拟在主体本身上如何进行：

- dynamic 属性决定该主体是否由物理子系统来模拟。
- affectedByGravity 属性决定模拟是否对主体产生重力。关于物理世界的更多信息，请参阅“配置物理世界”。
- allowsRotation 属性决定力是否能对主体传递角速度（angular velocity）。

质量决定主体的加速阻力（惯性）

你应该设置场景中每个基于体积的主体的质量，使它能对施加给它的力作出正确反应。

一个物理体的 mass、area、density 属性是相互关联的。当你初次创建主体时，主体的体积就计算好了，且以后永远不会改变。另外两个属性值会根据下面的公式同时发生变化：

质量=密度×体积

在你配置一个物理体时，你有两种选择：

- 设置主体的 mass 属性。然后 density 属性会自动重新计算。当你想要精确地控制每一个主体的质量时，这种方法是最有用的。
- 设置主体的 density 属性。然后 mass 属性会自动重新计算。当你有一组以不同尺寸创建的类似的主体集合时，这种方法是最有用的。例如，如果你的物理体被用来模拟小行星，你可能让所有小行星有相同的密度，然后为每个行星 设置一个合适的多边形边框。每个主体根据它在屏幕上的尺寸自动计算出适当的质量。

何时调整主体的属性

大多数情况下，你配置完一个物理体，然后永远不会改变它。例如，一个主体的质量在游戏过程中是不太可能改变的。不过，你并没有被限制这样做。有些种类的游戏可能需要有能力来调整主体的属性，即使模拟正在执行中。这里有几个例子时，你可能这样做：

- 在一个逼真的火箭模拟中，火箭消耗燃料提供推力。在燃料用完时，火箭的质量变化了。为了在 Sprite Kit 中实现这个，你可以创建一个包括 `fual` 属性的火箭类。当火箭推动时，燃料被减少，重新计算相应主体的质量。
- damping 属性通常是基于主体的特性和它要穿越的介质。例如，真空不提供阻力，而水比空气提供更大阻力。如果你的游戏模拟多个环境，并且允许主体在那些环境之间移动，那么每当主体移动到一个新的环境时，需要更新主体的 damping 属性。

通常情况下，你将这些变化作为场景预处理和后处理的一部分。请参见“高级场景处理。”

配置物理世界

场景中的所有物理体都是物理世界的一部分。物理世界是附属于场景的 `SKPhysicsWorld` 对象。它定义了模拟的两个重要的特性：

- `gravity` 属性对模拟中基于体积的主体施加加速度。静态体和 `affectedByGravity` 属性设置为 `NO` 的物理体则不受影响。
- `speed` 属性决定了模拟的运行速率。

让物理体移动

默认情况下，只有重力施加到场景中的物理体。在某些情况下，这可能是已经足以构建一个游戏。但在大多数情况下，你需要采取其他步骤来改变物理体的速度。

首先，你可以通过设置物理体的 `velocity` 和 `angularVelocity` 属性直接控制其速度。与许多其他特性一样，你通常只在第一次创建物理体时设置这些属性一次，然后让物理模拟在需要加以调整。例如，假设你正在做一个基于空间的游戏，在游戏中火箭飞船可以发射导弹。当船发射

一枚导弹时，该导弹应该有一个起始的船的速度，加上一个额外的发射方向的向量。清单 8-3 展示了一个计算发射速度的实现。

清单 8-3 计算导弹的初始速度

```
missile.physicsBody.velocity = self.physicsBody.velocity;

[missile.physicsBody applyImpulse:CGPointMake(missileLaunchImpulse*cosf(shipDirection),

                                              missileLaunchImpulse*sinf(shipDirection))];
```

一旦主体处于模拟中，更常见的做法是根据作用于主体的力来调整主体的速度。速度变化的另一个来源，碰撞，将在后面讨论。

默认的作用于主体的力的集合包括：

- 物理世界提供的重力
- 由主体自身的属性产生的阻力
- 与系统中的另一个主体接触产生的摩擦力

然而，你也可以把自己的力和动量施加到物理体上。大多数情况下，你在模拟执行前的预处理步骤施加力和动量。你的游戏逻辑负责决定那些力需要施加并调用适当的方法施加那些力。

你可以选择施加一个力或动量：

- 力会作用一段时间，根据在你施加力与下一帧模拟处理之间经过的模拟时间量。所以，持续地施加力到主体，你需要在每次一个新帧正在处理时调用适当的方法。
- 动量使主体的速度发生瞬间变化，这种变化独立于已经过的模拟时间量。这意味着动量通常用于立即改变主体的速度，而力用于持续性效果。

所以，继续回到火箭的例子，火箭飞船可能在火箭开动它的发动机时对它施加了一个力。然而，当它发射导弹时，它可能以火箭自身的速度发射导弹，然后施加一个单一的动量给它来得到最初的爆发速度。

因为力和动量是对相同概念的建模——调整主体的速度——本节其余部分只集中讨论力。你可以用三种方式之一施加力到一个主体：

- 线力（linear force），仅影响主体的线速度。

- 角力（angular force），仅影响主体的角速度。
- 对主体上的一个点施加的力。根据对象的形状和力施加的点的位置，物理模拟计算主体的角速度和线速度的单独改变。

清单 8-4 展示了一个精灵的子类可能实现来对船舶施加力的代码。它在主引擎被激活时使火箭加速。由于引擎是在火箭的后面，力成直线地施加到火箭，不担心角度变化。代码计算基于火箭当前方向的推力矢量。该方向基于根据相应节点的 `zRotation` 属性，但插图（artwork）的方向可能会跟节点的方向有所不同。推力应始终面向插图。请参阅“绘制你的内容”。

清单 8-4 施加火箭推力

```
static const CGFloat thrust = 0.12;



CGFloat shipDirection = [self shipDirection];

CGPoint thrustVector = CGPointMake(thrust*cosf(shipDirection),

                                   thrust*sinf(shipDirection));

[self.physicsBody applyForce:thrustVector];
```

清单 8-5 展示了一个类似的效果，但是这一次火箭通过力而旋转，所以此推力被施加成角推力。

清单 8-5 施加侧向推力

```
static const CGFloat thrust = 0.01;



CGFloat shipDirection = [self shipDirection];

CGPoint thrustVector = CGPointMake(thrust*cosf(shipDirection),

                                   thrust*sinf(shipDirection));



[self.physicsBody applyTorque: thrustVector];
```

使用碰撞和接触

迟早，两个主体将试图占据同一空间。由你的游戏决定会发生什么。Sprite Kit 允许物理体之间有两种交互：

- **接触(**contact)是在你需要知道两个主体在互相触碰(touch)时使用。在大多数情况下，当你在碰撞发生时你需要让游戏变化的话，你可以使用接触。
- **碰撞**是用来防止从两个对象穿过对方。当一个人的主体撞击（strike）另一个主体，Sprite Kit 自动计算碰撞的结果，并对碰撞中的主体施加动量。

你的游戏配置场景中的物理体，以确定何时应该发生碰撞，以及何时物理体之间的交互需要执行额外的游戏逻辑。限制这些交互不仅对定义游戏逻辑很重要，对从 Sprite Kit 中获得良好性能也是必要的。Sprite Kit 使用两种机制限制每帧中交互的数量：

- 基于边的物理体永远不会与其他基于边的主体发生作用。这意味着，即使你通过重新定位节点来移动基于边的主体，物理体也从来不会发生相互碰撞或接触。
- 每一个物理体都被分类（categorized）。类别（Categories）由你的应用程序定义，每个场景最多可以有 32 个类别。当你配置一个物理体，你定义它属于哪些类别，以及它想跟哪些类别的主体发生作用。接触和碰撞分别规定。

碰撞和接触的例子：太空中的火箭

通过一个例子的探索，接触和碰撞系统更容易理解。在这个例子中，场景被用来实现一个太空游戏。两个火箭飞船在外太空的某部分决斗。这个太空区域有一些可能会跟飞船发生碰撞的行星和小行星。最后，因为这是一场决斗，两个火箭飞船都装备有导弹，他们可以向对方发射。这个简单的描述定义了例子的粗糙的玩法。但是实现这个例子，什么主体在场景中以及它们如何交互需要更精确的表达。

注意： 这个例子也作为示例代码可以获得：*SpriteKit Physics Collisions*。
从上面的描述中，你可以看到，游戏中有 4 种出现在场景的单位类型（unit types）：

- 导弹
- 火箭飞船

- 小行星（Asteroids）
- 行星（Planets）

单位类型的数量之少表明，一列简单的类别将是一个良好的设计。虽然场景被限制为 32 个类别，这个设计只需要 4 个，每个单元类型对应一个。每个物理体属于一个且只属于一个类别。因此，导弹可能会以一个精灵节点的形式出现。导弹的精灵节点有一个关联的物理体，而那个主体是属于导弹类别的。其他节点和物理体都以类似的方式定义。

专家提示： 考虑对其他游戏相关的信息用其余类别进行编码。然后，在实现你的游戏逻辑时使用这些类别。例如，在实现一个更高级的火箭游戏的逻辑时，任何以下属性都可能是有用的：

- 人造对象
- 自然对象
- 碰撞过程中发生的爆炸

给定这 4 个类别后，下一个步骤是定义这些物理体之间允许的交互。接触交互通常是首要解决的，因为它们几乎都是由你自己的游戏逻辑指示的。在许多情况下，如果检测到接触，则你也需要计算碰撞。在接触导致两个物理体之一从场景移除时，这是最常见的。

表 8-1 描述了火箭游戏的接触交互。

表8-1 火箭的接触网格

	导弹	火箭飞船	小行星	行星
导弹	NO	YES	YES	YES
火箭飞船	NO	YES	YES	YES
小行星	NO	NO	NO	YES
行星	NO	NO	NO	NO

所有这些交互都基于游戏的逻辑。也就是说，当任何这些接触发生时，需要通知游戏，以便它可以更新游戏状态。下面是需要做的事情：

- 导弹在它撞击飞船、小行星或行星时爆炸。如果导弹击中飞船，船会受到损伤。
- 飞船接触到飞船、小行星或行星会受到损伤。

- 小行星接触行星会被摧毁。

让这些交互对称是没有必要的，因为 Sprite Kit 在每帧对每个接触只调用你的代理（delegate）一次。两个主体的任意一个都可以指定它对接触有兴趣。所以，由于在导弹撞击飞船时它已经请求了一个接触的消息，飞船就不需要请求相同的接触消息了。

下一个步骤是决定场景应该何时计算碰撞。每个主体描述场景中的哪些主体可以在发生撞击时调整自身的速度。表 8-2 描述了允许的碰撞的一个列表。

表8-2　火箭的碰撞网格

	导弹	火箭飞船	小行星	行星
导弹	NO	NO	NO	NO
火箭飞船	NO	YES	YES	YES
小行星	NO	YES	YES	NO
行星	NO	NO	NO	YES

在使用碰撞时，每个物理体的碰撞信息是非常重要的。当两个物体碰撞时，有可能只有一个主体对碰撞感兴趣。当这种情况发生时，只有感兴趣的主体的速度被更新。

在上面的表中，我们作了以下假设：

- 导弹在与其他物体的任何交互中总是被摧毁，所以导弹忽略所有与其他主体的碰撞。同样，导弹被认为质量太轻而不足以在碰撞中移动其他主体。虽然游戏可以选择让导弹与其他的导弹碰撞，但在本游戏中没有选择这样做，因为会有很多飞来飞去的导弹。因为每个导弹可能需要对所有其他导弹进行测试，这些交互将需要大量额外的计算。

- 飞船在乎与飞船、小行星和行星的碰撞。

- 小行星不在乎与行星的碰撞，因为在游戏对接触状态的描述中，小行星会被毁灭。

- 行星只在乎与其他行星发生碰撞。其余的都没有足够的质量移动行星，因此本游戏忽略这些碰撞并避免潜在的昂贵的计算。

用代码实现火箭示例

一旦你已经确定好你的分类和交互，你需要在你的游戏的代码中实现它。分类和交互各由一个 32 位掩码（mask）定义。每当一个潜在的交互发生时，每个主体的类别掩码针对其他主体的接触和碰撞掩码进行测试。通过对两个掩码进行逻辑与运算来执行这些测试。如果结果是一个非零数，则该交互发生。

以下是你如何把太空战争设计到 Sprite Kit 代码中的方法：

1. 定义类别掩码值：

   清单 8-6 太空决斗的类别掩码值

   ```
   static const uint32_t missileCategory 0X1 << 0;

   static const uint32_t shipCategor 0X1 << 1;

   static const uint32_t asteroidCategory 0X1 << 2;

   static const uint32_t planetCategory 0X1 << 3;
   ```

2. 在一个物理体初始化时，设置其 `categoryBitMask`、`collisionBitMask` 和 `contactTestBitMask` 属性。

   清单 8-7 展示了对于表 8-1 和表 8-2 中的火箭飞船条目（entry）的一个典型的实现。

   清单 8-7 对火箭的接触和碰撞掩码赋值

   ```
   SKSpriteNode *ship = [SKSpriteNode spriteNodeWithImageNamed:@"spaceship.png"];

   ship.physicsBody = [SKPhysicsBody bodyWithRectangleOfSize:ship.size];

   ship.physicsBody.categoryBitMask = shipCategory;

   ship.physicsBody.collisionBitMask = shipCategory | asteroidCategory | planetCategory

   ship.physicsBody.contactTestBitMask = shipCategory | asteroidCategory | planetCatego
   ```

3. 接触代理赋值为场景的物理世界。通常情况下，代理协议由场景实现。这是清单 8-8 中的情况。

清单 8-8 添加场景作为接触代理

```
 - (id)initWithSize:(CGSize)size

{

    self = [super initWithSize:size])

   {

...

      self.physicsWorld.gravity: = CGPointMake(0,0);

      self.physicsWorld.contactDelegate = self;

...

   }

   return self;

}
```

4. 实现接触代理方法来添加游戏的逻辑。

清单 8-9 接触代理的部分实现

```
 - (void)didBeginContact:(SKPhysicsContact *)contact

{

   SKPhysicsBody *firstBody, *secondBody;



   if(contact.bodyA.categoryBitMask <contact.bodyB.categoryBitMask)
```

```
    {
        firstBody = contact.bodyA;
        secondBody = contact.bodyB;
    }
    else
    {
        firstBody = contact.bodyB;
        secondBody = contact.bodyA;
    }
    if((firstBody.categoryBitMask & missileCategory) != 0)
    {
        [self attack:secondBody.node withMissile:firstBody.node];
    }
...
}
```

这个例子展示了实现你的接触代理时要考虑的几个重要概念：

- 传递一个声明哪些主体参与碰撞的 SKPhysicsContact 对象给代理。
- 接触中的主体可以以任何顺序出现。在火箭的代码中，交互网格总是以类别排好的顺序指定。在这种情况下，在接触发生时代码依赖这个对两个条目进行排序。一个简单的选择可能是检查接触中的两个主体并进行分派（dispatch），如果其中一个主体匹配掩码条目的话。

- 当你的游戏需要使用接触时，你需要根据碰撞的两个主体来确定结果。考虑研究了 Double Dispatch 模式，并用它来移交（farm out）对系统中的其他对象的工作。在场景中嵌入所有的逻辑会导致冗长、复杂而难于阅读的接触的代理方法。
- 物理体允许访问包含它们的节点。这对访问碰撞中的节点的其他的信息，是非常有用的。例如，你可以使用该节点的类、它的 `name` 属性或存储在其 `userData` 字典中的数据，来确定应如何处理该接触。

对小或快速移动的对象使用高精度碰撞

当 Sprite Kit 进行碰撞检测时，它首先要确定场景中所有的物理体的位置。然后确定是否发生了碰撞或接触。这种计算方法很快，但有时可能会导致遗漏（miss）了碰撞。一个小的主体可能移动得相当快，以致于它完全通过了另一个物理体后，在两个物理体相互触碰的地方，根本没有一帧动画。

如果你有必须碰撞的物理体，你可以提示 Sprite Kit 使用一个更精确的碰撞模型检查交互。这种模式的开销是比较昂贵的，所以应谨慎使用。当任意一个主体使用精确碰撞时，多个运动位置被接触和测试，以确保检测到所有的接触。

```
ship.physicsBody.usesPreciseCollisionDetection = YES;
```

把物理体连接在一起

虽然你可以使用上面已经描述的物理系统来制作很多有趣的游戏，但是通过把物理体连接在一起，你可以让你的设计更进一步。物理体使用**联合**（joints）连接。当场景模拟物理时，计算力是如何作用于主体会考虑到这些联合。

图 8-3 以不同的方式把节点连接在一起，

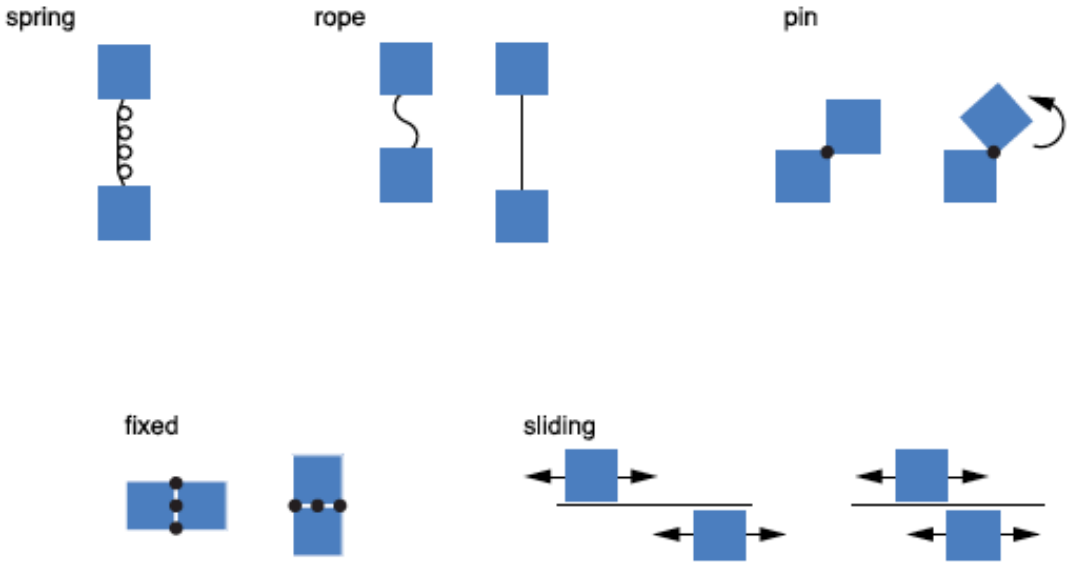


表 8-3 描述了你可以在 Sprite Kit 中创建的各种联合。

表8-3 Sprite Kit中实现的Joint类

类别名称	描述
SKPhysicsJointFixed	固定联合在某个参考点混合两个主体在一起。固定联合用于创建以后可以打散的复杂形状。
SKPhysicsJointSliding	滑动联合允许两个主体的锚点沿选定的轴滑动。
SKPhysicsJointSpring	弹簧联合就像弹簧那样，它的长度是两个主体之间的初始距离。
SKPhysicsJointLimit	界限联合限定了两个主体之间的最大距离，就像他们是用绳子连接那样。
SKPhysicsJointPin	针联合允许两个主体独立地绕锚点旋转，就像钉在一起那样。

联合使用物理世界来添加到模拟或从模拟中移除。当你创建一个联合，连接联合的点总是在场景的坐标系中指定。这可能需要你首先把节点坐标转换成场景坐标，然后再创建一个联合。

要在你的游戏中使用物理联合，你遵循以下步骤：

1. 创建两个物理体。
2. 把物理体附加到场景中的一对 SKNode 对象。

3. 使用上面列出的子类之一创建一个联合对象。

4. 如果有必要，配置联合对象的属性来定义联合应该如何操作。

5. 取回（Retrieve）场景的 `SKPhysicsWorld` 对象。

6. 调用物理世界的 `addJoint:` 方法。

搜索物理体

有时，在场景中查找物理体是必要的。例如，你可能需要：

- 发现物理体是否位于场景的某个区域。
- 检测物理体（如由玩家控制的那个）何时穿过某条特定的线。
- 跟踪两个物理体之间的视线（line-of-sight），看是否有另一个物理体夹在两个对象之间，例如墙。

在某些情况下，这些类型的交互，可以使用碰撞和接触系统实现。例如，要发现一个物理体何时进入一个区域，你可以创建一个物理体并将它附加到场景中的一个无形的节点上。然后，配置物理体的碰撞掩码，以致它从来不与任何东西碰撞，并配置它的接触掩码在检测你感兴趣的物理体。你的接触代理在渴望的交互发生时被调用。

然而，像视线这样的概念，用这种设计却不容易实现。要实现这些，你使用场景的物理世界。通过物理世界，你可以搜索沿着射线的所有物理体或与一个特定的点或矩形相交的物理体。

下面将用一个例子来说明这些基本技术。清单 8-10 展示视线检测系统的一种可能的实现。它从场景的地点以一个特定的方向投射一条射线，寻找沿着射线最近的物理体。如果找到一个物理体，那么它测试该类别的掩码，看看这是否它应该攻击的一个目标。如果它看到一个目标，就攻击它。

清单 8-10 从场景中心发射射线

```
- (BOOL)isTargetVisibleAtAngle:(CGFloat)angle distance:(CGFloat)distance

{

   CGPoint rayStart = CGPointZero;
```

```
    CGPoint rayEnd = CGPointMake(distance*cosf(angle), distance*sinf(angle));



    SKPhysicsBody *body = [self.physicsWorld bodyAlongRayStart:rayStart end:rayEnd];

   Return (body && body.categoryBitMask == targetCategory) ;

}



 - (void)attackTargetIfVisible

{

    if([self isTargetVisibleAtAngle:self.cannon.zRotation distance:512])

    {

        [self shootCannon];

    }

}
```

实现同样行为的另一种方式可能是，将场景中的两个物理体设置为射线的起始和结束位置。例如，你可能会使用玩家的游戏对象的位置作为一个位置而一个敌人单位的位置作为另一个位置。

搜索与点或矩形相交的物理体，执行也是类似的，使用 `bodyAtPoint:`和 `bodyInRect:`方法 。

有时你不能根据场景内最接近的物理体做一个简单的判断。例如，在你的游戏的逻辑中，你可能会决定，并非所有的物理体都遮挡视线。在这种情况下，你就需要枚举沿着射线的所有物理体。你会使用 `enumerateBodiesAlongRayStart:end:usingBlock:`方法实现这个。你提供了一个 block 给沿着射线的每个主体都调用一次。然后，你可以使用此信息，对于视线是否存在目标做出更明智的决定。

在游戏中使用物理的提示和技巧

构建一个基于物理的游戏时，考虑了以下建议。

系统地设计你的物理体

在你花大量时间把物理添加到场景前，你应该先了解你包含要什么样的主体到场景以及他们如何互相交流。在这一步，你应该去让每个主体经受这个过程：

- 它是一个边、一个静态体积还是动态体积？请参阅“所有物理都在物理体上模拟”。
- 什么样的形状与主体最接近，牢记一些形状的计算比其他的昂贵？请参阅“使用匹配图形表现的物理形状”。
- 它是什么类型的主体，它如何与其他主体交互？请参阅“使用碰撞和接触”
- 这主体移动得快吗，或者它是很微小（tiny）的吗？如果是这样，决定高精度碰撞检测是否必要。请参阅“对小或快速移动的对象使用高精度碰撞”。

回避数字

虽然知道 Sprite Kit 以国际单位制测量项目是有用的，对精确数字的担心并不是那么重要。你的火箭飞船重 1 公斤还是 100 万公斤并没有多大关系，只要这个质量与其他在游戏中使用的物理值一致。通常情况下，比例比所使用的实际值更重要。

游戏设计通常是一个反复的过程，因为你在模拟中会调整（tweak）数字。这类设计往往导致在你的游戏中有许多的硬编码的数字。抗拒这样做的冲动！相反，实现这些数字作为可以被相应的节点对象或物理体归档的数据。你的代码应该提供行为，但用于实现这些行为的具体数字应该是可编辑的值，可以由美工或设计师调整或测试。

大部分信息保存在物理体和相应的节点中，物理体和节点可以使用 Cocoa 创立的标准归档机制进行归档。这暗示着你自己的工具也可能能够保存和加载这些档案，并把它们作为首选的数据格式。这通常是可能的，但是要记住，一个物理体的形状是无法从对象确定的私有数据。这意味着如果在你的工具中你的确使用归档作为保存数据的主要格式，你可能还需要归档用于创建物理体的其他信息。使用 Sprite Kit 来开发工具的一般话题在“Sprite Kit 最佳实践”中描述。

大多数物理属性是动态的，所以在运行时使它们适应

一个物理体只有非常少的特性是固定的。物理体的范围之外，大多数属性可以在任何时候改变。你可以利用这一点。下面是几个例子：

- 一个静态体积可以锁定在适当的地方，直到玩家进行解锁的任务。然后，主体变成一个动态体积并与其他对象交互。
- 可以从多个更小的物理体构造一个框并使用固定联合放到一起。用接触掩码创建各个部分，这样，当他们碰到了什么东西时，接触代理可以打破联合。
- 作为在整个场景移动的对象，根据它所在介质调整它的线性和旋转阻尼。 例如，当对象移动到水中时，更新该属性来匹配。

Sprite Kit最佳实践

此时此刻，你已经十分清楚 Sprite Kit 可以做什么以及它是如何做的。你知道如何将节点添加到场景并让那些节点执行动作——创建游戏可玩性（gameplay）的构建块（building blocks）。你可能会错过的是更大的蓝图。也就是说，你需要了解如何使用 Sprite Kit 规划并开发游戏和工具。为了发挥 Sprite Kit 的最大功效，你需要知道：

- 如何组织你的游戏到场景和过渡中
- 何时子类化 Sprite Kit 类
- 如何存储你游戏的数据和美术
- 如何使用构建你自己的工具来创建 Sprite Kit 内容并导出该内容供你的游戏使用

Sprite Kit 提供了不仅仅是你的游戏的图形层，它还提供使 Sprite Kit 易于集成到你的自定义游戏工具的功能。通过集成 Sprite Kit 到你的游戏工具，你可以在工具中构建你的内容，并直接读取到你的游戏引擎中。数据驱动的设计，允许美工、游戏设计师和游戏程序员协作构建游戏的内容。

组织游戏内容到场景

场景是创建 Sprite Kit 内容的基石。当你开始一个新的游戏项目时，你的任务是定义需要哪些场景以及何时在这些场景之间发生过渡。场景通常代表出现在玩家前的游戏或内容的模式。通常情况下，很容易看出你何时需要一个新的场景：如果你的游戏需要更换屏幕上的所有内容，过渡到一个新的场景。

设计你的游戏的场景及它们之间的转换，与在一个传统的 iOS 应用程序的视图控制器的作用相似。在一个 iOS 应用程序中，内容由视图控制器实现。每个视图控制器创建一组视图的集合来绘制内容。起初，一个视图控制器由窗口展示。然后，当用户与视图控制器的视图交互时，可能会触发一个到另一个视图控制器及其内容的过渡。例如，在表视图中选择一个项目可能会弹出一个细节视图控制器来显示所选项目的内容。

场景没有默认的行为，像一个传统的 iOS 应用中的 storyboadrd 那样。相反，你定义并实现场景的行为。这些行为包括：

- 何时创建新的场景
- 每个场景的内容
- 何时在场景之间的发生过渡
- 用于执行过渡的视觉效果
- 数据如何从一个场景传输到另一个场景

例如，你可以实现一个类似于一个 segue 的模型，新的场景总是在这里的过渡上实例化。或者，你可以设计你的游戏引擎使用它坚持维持的场景。每一种方法都有它的优点：

- 如果场景在每次过渡发生时都要实例化，它始终是在一个干净的已知的状态中创建。这意味着，你不必担心重置场景的任何内部状态，这些状态往往可以有细微 bug。
- 如果场景是持久性的，那么你就可以过渡回到场景，并让它恢复你离开该场景时的相同的状态。这种设计适用于任何类型的你需要在多个场景的内容之间快速过渡的游戏。

允许你的场景设计进化

通常情况下，一个新的场景会是分阶段开发的。在开始时，你可能会使用测试的应用程序和实验的想法来了解 Sprite Kit 是如何工作的。但后来，随着你的游戏变得更加复杂，你的场景需要适应。

在测试应用程序和一些简单的游戏中，你的逻辑和代码都在场景子类中。场景操作的节点树和树中每个节点的内容，根据需要而运行动作或改变其他行为。该项目是如此地简单，以致所有的代码都可以留在一个单一的类中。

项目的第二阶段时，通常在渲染或游戏逻辑开始变得更长或更复杂时发生。在这个阶段，你通常开始抽离（break out）特定的行为并在其他类中实现它们。例如，如果你的游戏包括一个摄像头的概念，你可能会创建一个 `CameraNode` 类来封装相机的行为。然后，你可能创建其他节点类封装其他行为。例如，你可能会创建单独的节点类来表示在你的游戏中的单元（units）。

在最复杂的项目中，人工智能等概念变得更加重要。在这些设计中，你可能最终创建独立于 Sprite Kit 工作的类。这些类的对象进行代表场景的工作，但并不特别依赖于它。这些类通常从你的 Sprite Kit 子类中提取，在你意识到你许多方法实现游戏逻辑而并没有真正触及任何 Sprite Kit 内容的时候。

限制树的内容以提高性能

在 Sprite Kit 渲染一帧时，它剔除（cull）所有在屏幕上不可见的节点。从理论上讲，这意味着你可以简单地保持所有内容到场景，并让 Sprite Kit 做所有的工作来管理它。而对于适度渲染要求的游戏，这样的设计也许就足够了。但是，随着你的游戏变得更大和更复杂，你需要通过 Sprite Kit 做更多的工作以确保良好的性能。

通常情况下，一个节点需要成为节点树的一部分，因为：

- 它有一个相当不错的机会在不久的将来被渲染
- 该节点运行准确的游戏操作所需要的动作
- 节点具有准确的游戏操作所需要的物理体

当一个节点不符合这些要求中的任意一个，通常最好是把它从树上移除，特别是如果它自身有许多子节点。例如，发射器节点往往添加特殊效果，而根本不影响游戏。它们发射大量的粒子，所以它们的渲染代价可能很昂贵。如果你有大量的发射器在场景中，但在屏幕外，场景可能需要处理数百或数千无形的节点。更好的做法是移除发射节点，直到它们变得可见。

通常，剔除算法的设计要根据自己的游戏。例如：

- 在赛车游戏中，玩家通常是围绕赛道以一个一致的方向行驶。正因为如此，你通常可以预测在不久的将来将看到什么样的内容并预加载它。随着玩家通过赛道前进，你可以移除玩家不再可以看到的节点。
- 在冒险游戏中，玩家可能处在一个滚动的环境，并允许他们能够往任意方向移动。当玩家在世界中移动时，你也许可以预测到哪种地形在附近而哪种地形不在。然后，只有包含局部内容的地形。

当内容总是被一次添加和移除时，考虑使用一个中间（interim）节点对象来收集的所有内容。这允许你调用一个单一的方法就添加或移除一个大组内容。

什么不应该在一个场景内

在你首次设计一个 Sprite Kit 游戏时，可能看起来像是场景类是做了很多的工作。调整（tuning）你的应用程序的部分过程，是决定场景是否执行一个任务，或是否应该由游戏中的一些其他对象这样做。例如，你可能会在以下时候考虑将工作移到另一个对象：

- 内容或应用程序逻辑由多个场景共享
- 内容或应用程序逻辑的建立（set up）特别昂贵，且只需要执行一次

例如，如果你的游戏在所有的游戏玩法（gameplay）中使用相同的纹理，你可以创建一个特殊的加载类，在启动时运行一次。你执行一次加载纹理的工作，然后让它们留在内存中。如果场景对象被删除并重新创建来重启游戏，纹理并不需要被重新加载。

使用子类来创建你自己的节点行为

设计新的游戏需要你子类化 `SKScene` 类。然而，Sprite Kit 中的其他节点类也设计成可子类化，这样你就可以添加自定义的行为。例如，你可能会为子类化 `SKSpriteNode` 类来添加你的游戏专用的 AI 逻辑。或者，你可能会子类化 `SKNode` 类来创建一个类实现场景中的一个特定的绘图层。如果你想直接在一个节点中实现交互性（interactivity），你必须创建一个子类。

当你设计一个新的节点类时，有一些重要的针对于 Sprite Kit 的实现细节需要了解。但是，你还需要考虑新的类在游戏中的作用，以及类的对象如何与其他对象交互。你需要创建定义良好（well-defined）的类接口和调用约定（convention）来允许对象交互操作（interoperate），而没有微妙的 bug 减缓你的开发进程。

下面是创建自己的子类时要遵循的重要指南：

- 所有的标准节点类都支持 `NSCopying` 与 `NSCoding` 协议。如果你的子类添加新的属性或实例变量，那么你的子类也应该实现这些行为。这种支持是必不可少的，如果你打算在你的游戏中复制节点或使用归档来构建你自己的游戏工具。
- 虽然的节点与视图类似，但你不可以添加新的绘图行到节点类。你必须通过节点的现有方法和属性开展工作。这意味着，要么控制节点自身的属性（如改变精灵的纹理），要么添加额外的节点并控制它们的行为。在任意一种情况下，你需要考虑你的类将如何与其他部分的代码相互作用。你可能需要构建自己的调用约定，以避免细微的渲染 bug。举例来说，一个通用的约定是，一个对于创建并管理自己的子节点的节点对象，应避免添加子节点到该节点对象上。
- 在许多情况下，期望添加在场景的预处理和后处理阶段可以调用的方法。这可以让你从你的场景子类中移除这些行为，并移到为一个特定的游戏对象处理所有行为的类中。

- 如果你想在一个节点类中实现事件处理，则必须为 iOS 和 OS X 实现单独的事件处理的代码。在 OS X 上 `SKNode` 类继承自 `NSResponder`，而在 iOS 上继承自 `UIResponder` 的。

- 在一些游戏的设计中，你可以依赖这样的事实：特定组合的类总是要在一个特定的场景一起使用。在其他设计中，你可能想要创建可用于在多个场景中的类。复用对你的设计越重要，你应该花越多的时间为对象设计干净的接口来互相作用。当两个类是互相依赖的，使用代理（delegation）打破这种依赖。大多数情况下，你这样做是通过在你节点上定义一个代理和代理要实现的协议（protocol）。你的场景（或另一个节点，如节点的父节点）实现了这个协议。这允许在多个场景中复用你的节点类，而不需要知道场景类。

绘制你的内容

构建节点树的很大一部分工作是组织需要绘制的图形内容。首先需要绘制什么？最后需要绘制什么？这些东西是如何渲染的？

在设计节点树时，考虑了以下建议：

- 不要直接添加内容节点或物理体到场景。相反，添加一个或多个 `SKNode` 对象到节点树来代表你的游戏中不同层次的内容，然后再使用这些层对象。使用层，可以让你精确控制每一层的行为。例如，你可以旋转某一层的内容而不旋转所有的场景的内容。它也让你通过其他代码移除或替换部分场景渲染变得更容易。例如，如果游戏分数和其他信息显示在提示显示层（heads-up display layer），那么在你想进行截图时可以移除该层。在一个非常复杂的应用程序中，你可能会持续使用这种模式，通过添加子节点到一个层节点让节点树越来越深。

  当你使用层来组织你的内容时，考虑层之间如何彼此交互。他们知不知道任何有关对方内容的东西？更高层次的层为了渲染它自己的内容，是否需要知道任何有关较低层次的层如何渲染的东西？

- 有节制地使用切割节点和效果节点。两者都非常强大，但也可能开销昂贵，特别是当它们一起嵌套在节点树内的时候。

- 只要有可能，一起渲染的节点，应该使用相同的混合模式。如果一个节点的所有的子节点使用相同的混合模式和纹理图册，那么 Sprite Kit 通常可以在一个单一绘图通道（drawing pass）中绘制这些精灵。另一方面，如果子节点组织起来，以致每个新的精灵的绘图模式发生变化，那么 Sprite Kit 可能以每个精灵一个绘图通道的方式执行，这是相当低效的。

- 当设计发射器效果时，尽可能使用低粒子出生率。粒子并不是免费的，每个粒子都添加渲染和绘制的开销。
- 默认情况下，精灵和其他内容使用 alpha 混合模式进行。如果精灵的内容是不透明的，例如一个背景图像，使用 `SKBlendModeReplace` 混合模式。
- 使用游戏逻辑和匹配 Sprite Kit 的坐标和旋转约定（convention）的美术资产。这意味着定向插图到右边。如果你定向插画到其他方向，你将需要转换美术使用的约定和 Sprite Kit 使用的约定之间的角度。例如，如果插图定向为向上，那么你将在角度上添加 `PI/2` 弧度来从 Sprite Kit 的约定转换到你的美术的约定，反之亦然。
- 在 `SKView` 类中开启诊断信息。使用的帧率作为常规的性能诊断，并使用节点和绘图通道计数来进一步了解如何呈现内容。你还可以使用 Instruments 及其 OpenGL 诊断工具，查看更多关于你的游戏时间花在哪里的信息。
- 在各种真实的硬件设备上测试你的游戏。在许多情况下，每个 Mac 或 iOS 设备上的 CPU 资源和 GPU 资源的平衡是不同的。在多个设备上进行测试，帮助你确定你的游戏是否在大多数设备上运行良好。

使用游戏数据

在任意给定的时间，你的游戏管理大量的数据，包括场景中的节点的位置。但它也包括静态数据，如：

- 美术资产及正确渲染插图所需的数据
- 水平（level）或拼图（puzzle）布局
- 用于配置游戏的数据（如怪物的速度和它攻击时造成多大损害）

只要有可能，尽量避免直接在游戏代码中嵌入你的游戏数据。数据更改时，你不得不重新编译游戏，这通常意味着程序员要参与设计变更。相反，数据应该对于代码保持独立，这样一个游戏设计师或美工可以直接更改数据。

存储游戏数据最好的地方要依赖于数据在你的游戏哪里使用。对于与 Sprite Kit 无关的数据，用属性列表存储在你的应用程序 bundle 中是一个很好的解决方案。然而，对于 Sprite Kit 数据，

你有另一种选择。因为所有 Sprite Kit 类都支持归档，你可以为重要的 Sprite Kit 对象简单地创建归档，然后在你的游戏中包含这些档案。例如，你可以：

- 存储游戏关卡作为一个场景节点的归档。此归档包括场景、节点树中它所有的后代节点以及它们所有的连接的物理体、联合和动作。
- 为特定的预配置节点存储个别的归档，比如每个怪物节点。然后，当需要创建一个新的怪物时，你从归档中加载它。
- 存储已保存的游戏作为场景归档。
- 构建自己的工具来让归档可以编辑。然后，你的游戏设计师和美工可以用这些工具来创建你的游戏对象和并使用你的游戏能读取的格式归档它们。你的游戏引擎和工具将共享通用的类。
- 你可以存储 Sprite Kit 数据到属性列表中。你的游戏加载属性列表，并用它来创建游戏资产。

以下是一些使用归档的指南：

- 使用节点的 `userData` 属性来存储游戏相关的数据，尤其是如果你不实现自己的子类。
- 避免硬编码引用到特定的节点。相反，给令人关注的节点一个独特的 `name` 属性并在节点树中搜索它们。
- 不能归档调用块的自定义动作。你需要创建并添加那些动作到你的游戏中。
- 大多数节点对象提供所有必要的属性，以确定它们是什么，以及他们是如何配置的。然而，动作和物理体没有。这意味着，当开发自己的游戏工具时，你不能简单地归档动作和物理体并使用这些档案来储存你的工具数据。相反，档案只应该是来自你的游戏工具的最终输出。

文档修订历史

下表描述了 *Sprite Kit 编程指南*的变更。

日期	说明
2013-06-10	介绍了如何使用Sprite Kit来实现游戏和其他任意的具有动画特性的应用程序的新文件。