# Sprite Kit Programming Guide

Developer

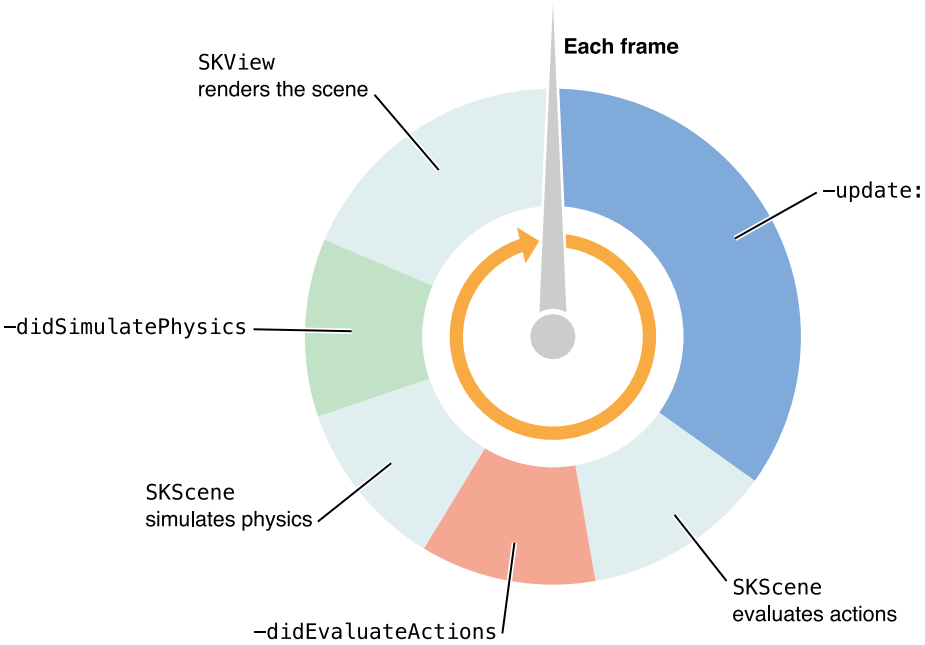
# Contents

# Figures, Tables, and Listings

# About Sprite Kit

Sprite Kit provides a graphics rendering and animation infrastructure that you can use to animate arbitrary textured images, or **sprites**. Sprite Kit uses a traditional rendering loop where the contents of each frame are processed before the frame is rendered. Your game determines the contents of the scene and how those contents change in each frame. Sprite Kit does the work to render frames of animation efficiently using the graphics hardware. Sprite Kit is optimized so that the positions of sprites can be changed arbitrarily in each frame of animation.

Sprite Kit also provides other functionality that is useful for games, including basic sound playback support and physics simulation. In addition, Xcode provides built-in support for Sprite Kit so that you can create complex special effects and texture atlases directly in Xcode. This combination of framework and tools makes Sprite Kit a good choice for games and other apps that require similar kinds of animation. For other kinds of user-interface animation, use Core Animation instead.




## At a Glance

Sprite Kit is available on iOS and OS X. It uses the graphics hardware available on the hosting device to composite 2D images at high frame rates. Sprite Kit supports many different kinds of content, including:

- Untextured or textured rectangles (sprites)
- Text
- Arbitrary CGPath-based shapes
- Video

Sprite Kit also provides support for cropping and other special effects; you can apply these effects to all or a portion of your content. You can animate or change these elements in each frame. You can also attach physics bodies to these elements so that they properly support forces and collisions.

Because Sprite Kit supports a rich rendering infrastructure and handles all of the low-level work to submit drawing commands to OpenGL, you can focus your efforts on solving higher-level design problems and creating great gameplay.

## Sprite Content is Drawn by Presenting Scenes Inside a Sprite View

Animation and rendering is performed by an `SKView` object. You place this view inside a window, then render content to it. Because it is a view, its contents can be combined with other views in the view hierarchy.

Content in your game is organized into **scenes**, which are represented by `SKScene` objects. A scene holds sprites and other content to be rendered. A scene also implements per-frame logic and content processing. At any given time, the view presents one scene. As long as a scene is presented, its animation and per-frame logic are automatically executed.

To create a game using Sprite Kit, you create one or more subclasses of the `SKScene` class. For example, you might create separate scene classes to display a main menu, the gameplay screen, and content displayed after the game ends. You can easily use a single `SKView` object in your window and transition between different scenes.

**Relevant Chapters:** "Jumping into Sprite Kit" (page 15), "Using Transitions Between Scenes" (page 65), "Sprite Kit Best Practices" (page 107)

## A Node Tree Defines What Appears in a Scene

The `SKScene` class is a descendant of the `SKNode` class. When using Sprite Kit, nodes are the fundamental building blocks for all content, with the scene object acting as the root node for a tree of node objects. The scene and its descendants determine which content is drawn and how it is rendered.

Each node's position is specified in the coordinate system defined by its parent. A node also applies other properties to its content and the content of its descendants. For example, when a node is rotated, all of its descendants are rotated also. You can build a complex image using a tree of nodes and then rotate, scale, and blend the entire image by adjusting the topmost node's properties.

The `SKNode` class does not draw anything, but it applies its properties to its descendants. Each kind of drawable content is represented by a distinct subclass in Sprite Kit. Some other node subclasses do not draw content of their own, but modify the behavior of their descendants. For example, you can use an `SKEffectNode` object to apply a Core Image filter to an entire subtree in the scene. By precisely controlling the structure of the node tree, you determine the order in which nodes are rendered.

All node objects are responder objects, descending either from `UIResponder` or `NSResponder`, so you can subclass any node class and create new classes that accept user input. The view class automatically extends the responder chain to include the scene's node tree.

**Relevant Chapters:** "Working with Sprites" (page 29), "Building Your Scene" (page 52), "Working with Other Node Types" (page 68)

## Textures Hold Reusable Graphical Data

Textures are shared images used to render sprites. Always use textures whenever you need to apply the same image to multiple sprites. Usually you create textures by loading image files stored in your app bundle. However, Sprite Kit can also create textures for you at runtime from other sources, including Core Graphics images or even by rendering a node tree into a texture.

Sprite Kit simplifies texture management by handling the lower-level code required to load textures and make them available to the graphics hardware. Texture management is automatically managed by Sprite Kit. However, if your game uses a large number of images, you can improve its performance by taking control of parts of the process. Primarily, you do this by telling Sprite Kit explicitly to load a texture.

A **texture atlas** is a group of related textures that are used together in your game. For example, you might use a texture atlas to store all of the textures needed to animate a character or all of the tiles needed to render the background of a gameplay level. Sprite Kit uses texture atlases to improve rendering performance.

**Relevant Chapters:** "Working with Sprites" (page 29)

## Nodes Execute Actions to Animate Content

A scene's contents are animated using **actions**. Every action is an object, defined by the `SKAction` class. You tell nodes to execute actions. Then, when the scene processes frames of animation, the actions are executed. Some actions are completed in a single frame of animation, while other actions apply changes over multiple

frames of animation before completing. The most common use for actions is to animate changes to the node's properties. For example, you can create actions that move a node, scale or rotate it, or make it transparent. However, actions can also change the node tree, play sounds, or even execute custom code.

Actions are very useful, but you can also combine actions to create more complex effects. You can create *groups* of actions that run simultaneously or *sequences* where actions run sequentially. You can cause actions to automatically repeat.

Scenes can also perform custom per-frame processing. You override the methods of your scene subclass to perform additional game tasks. For example, if a node needs to be moved every frame, you might adjust its properties directly every frame instead of using an action to do so.

**Relevant Chapters:** "Adding Actions to Nodes" (page 42), "Advanced Scene Processing" (page 81)

## Add Physics Bodies and Joints to Simulate Physics in Your Scene

Although you can control the exact position of every node in the scene, often you want these nodes to interact with each other, colliding with each other and imparting velocity changes in the process. You might also want to do things that are not handled by the action system, such as simulating gravity and other forces. To do this, you create physics bodies (`SKPhysicsBody`) and attach them to nodes in your scene. Each physics body is defined by shape, size, mass, and other physical characteristics.

When physics bodies are included in the scene, the scene simulates physics on those bodies. Some forces, such as friction and gravity, are applied automatically. You can also call methods on physics bodies to apply your own forces. The acceleration and velocity of each body is computed and the bodies collide with each other. Then, after the simulation is complete, the positions and rotations of the corresponding nodes are updated.

You have precise control over which physics bodies interact with each other. You determine which bodies can collide with each other and separately decide which interactions cause your app to be called. You use these callbacks to add game logic. For example, your game might destroy a node when its physics body is struck by another physics body.

The scene defines global characteristics for the physics simulation in an attached `SKPhysicsWorld` object. You use the physics world to define gravity for the entire simulation, to define the speed of the simulation, and to find physics bodies in the scene. You also use the physics world to connect physical bodies together using a joint (`SKPhysicsJoint`). Connected bodies are simulated together based on the kind of joint.

**Relevant Chapter:** "Simulating Physics" (page 88)

## How to Use This Document

Read "Jumping into Sprite Kit" (page 15) to get an overview of implementing a Sprite Kit game. Then work through the other chapters to learn the details about Sprite Kit's features. Some chapters include suggested exercises to help you develop your understanding of Sprite Kit. Sprite Kit is best learned by doing; place some sprites into a scene and experiment on them!

The final chapter, "Sprite Kit Best Practices" (page 107), goes into more detail about designing a game using Sprite Kit.

## Prerequisites

Before attempting to create a game using Sprite Kit, you should already be familiar with the fundamentals of app development. In particular, you should be familiar with the following concepts:

- Developing apps using Xcode
- Objective-C, including support for blocks
- The view and window system

For more information:

- On iOS, see *Start Developing iOS Apps Today*.
- On OS X, see *Start Developing Mac Apps Today*.

**Note:** Although this guide describes many techniques that are useful for creating games with Sprite Kit, it is not a complete guide to game design or game development.

## See Also

See *Sprite Kit Framework Reference* when you need specific details on functions and classes in the Sprite Kit framework.

See *Texture Atlas Help* and *Particle Emitter Editor Guide* for information on how to use Xcode's built-in support for Sprite Kit.

See *code:Explained Adventure* for an in-depth look at a Sprite Kit based game.

The following samples are available only in the OS X library, but still serve as useful Sprite Kit examples for iOS apps :

- See *Sprite Tour* for a detailed look at the `SKSpriteNode` class.
- See *SpriteKit Physics Collisions* to understand the physics system in Sprite Kit.

# Jumping into Sprite Kit

The best way to learn Sprite Kit is to see it in action. This example creates a pair of scenes and animates content in each. By working through this example, you will learn some of the fundamental techniques for working with Sprite Kit content, including:

- Using scenes in a Sprite Kit–based game.
- Organizing node trees to draw content.
- Using actions to animate scene content.
- Adding interactivity to a scene.
- Transitioning between scenes.
- Simulating physics inside a scene.

After you finish this project, you can use it to try out other Sprite Kit concepts. Some suggestions can be found at the end of this example.

You should already be familiar with creating iOS apps before working through this project. For more information, see *Start Developing iOS Apps Today*. Most of the Sprite Kit code in this example is the same on OS X.

## Getting Started

This walkthrough requires Xcode 5.0. Create a new Xcode project for an iOS app using the Single View Application template.

Use the following values when creating your project:

- Product Name: `SpriteWalkthrough`
- Class Prefix: `Sprite`
- Devices: iPad

Add the Sprite Kit framework to the project.

# Creating Your First Scene

Sprite Kit content is placed in a window, just like other visual content. Sprite Kit content is rendered by the `SKView` class. The content that an `SKView` object renders is called a **scene**, which is an `SKScene` object. Scenes participate in the responder chain and have other features that make them appropriate for games.

Because Sprite Kit content is rendered by a view object, you can combine this view with other views in the view hierarchy. For example, you can use standard button controls and place them above your Sprite Kit view. Or, you can add interactivity to sprites to implement your own buttons; the choice is up to you. Later in this example, you'll see how to implement interactivity on the scene.

### To configure the view controller to use Sprite Kit

1. Open the storyboard for the project. It has a single view controller (`SpriteViewController`). Select the view controller's view object and change its class to `SKView`.
2. Add an import line to the view controller's implementation file.

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

3. Implement the view controller's `viewDidLoad` method to configure the view.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    SKView *spriteView = (SKView *) self.view;
    spriteView.showsDrawCount = YES;
    spriteView.showsNodeCount = YES;
    spriteView.showsFPS = YES;
}
```

The code turns on diagnostic information that describes how the scene renders the view. The most important piece of information is the frame rate (`spriteView.showsFPS`); you want your games to run at a constant frame rate whenever possible. The other lines show details on how many nodes were displayed in the view and how many drawing passes it took to render the content (fewer is better).

Next, add the first scene.

### To create the Hello scene

1. Create a new class named `HelloScene` and make it a subclass of the `SKScene` class.

   The header file created by Xcode has the following content:

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


@interface HelloScene : SKScene


@end
```

   You don't need to change the header file.

2. Import the scene's header file in your view controller's implementation file.

```
#import "HelloScene.h"
```

3. Modify the view controller to create the scene and present the scene in the view.

```
- (void)viewWillAppear:(BOOL)animated
{
    HelloScene* hello = [[HelloScene alloc]
initWithSize:CGSizeMake(768,1024)];
    SKView *spriteView = (SKView *) self.view;
    [spriteView presentScene: hello];
}
```

4. Build and run the project.

   The app should launch and display a screen that is blank except for the diagnostic information.

## Adding Content to the Scene

When designing a Sprite Kit–based game, you design different scene classes for each major chunk of your game interface. For example, you might create a scene for the main menu and a separate scene for your gameplay. You'll follow a similar design here. This first scene displays the traditional "Hello World" text.

Most often, you configure a scene's content when it is first presented by the view. This is similar to the way view controllers load their views only when the `view` property is referenced. In this example, the code lives inside the `didMoveToView:` method, which is called whenever the scene is presented in a view.

### To display Hello World text in the scene

1. Add a new property to the scene's implementation file to track whether the scene has created its content. Your implementation file should look like this:

```
#import "HelloScene.h"

@interface HelloScene ()
@property BOOL contentCreated;
@end

@implementation HelloScene

@end
```

   The property tracks information that doesn't need to be exposed to clients, so, it is implemented in a private interface declaration inside of the implementation file.

2. Implement the scene's `didMoveToView:` method.

```
- (void)didMoveToView: (SKView *) view
{
    if (!self.contentCreated)
    {
        [self createSceneContents];
        self.contentCreated = YES;
    }
}
```

   The `didMoveToView:` method is called whenever the scene is presented by a view, but in this case, the scene's contents should only be configured the first time the scene is presented. So, this code uses the previously defined property (`contentCreated`) to track whether the scene's contents have already been initialized.

3. Implement the scene's `createSceneContents` method.

```
- (void)createSceneContents
{
    self.backgroundColor = [SKColor blueColor];
    self.scaleMode = SKSceneScaleModeAspectFit;
    [self addChild: [self newHelloNode]];
}
```

A scene paints the view's area with a background color before drawing its children. Note the use of the `SKColor` class to create the color object. In fact, `SKColor` is not a class; it is a macro that maps to `UIColor` on iOS and `NSColor` on OS X. It exists to make creating cross-platform code easier.

A scene's scale mode determines how the scene is scaled to fit in the view. In this example, the code scales the view so that you can see all of the scene's content, using letterboxing if required.

4. Implement the scene's `newHelloNode` method.

```
- (SKLabelNode *)newHelloNode
{
    SKLabelNode *helloNode = [SKLabelNode
labelNodeWithFontNamed:@"Chalkduster"];
    helloNode.text = @"Hello, World!";
    helloNode.fontSize = 42;
    helloNode.position =
CGPointMake(CGRectGetMidX(self.frame),CGRectGetMidY(self.frame));
    return helloNode;
}
```

In Sprite Kit, you never write code that explicitly executes drawing commands, as you would if you were using OpenGL ES or Quartz 2D. Instead, you add content by creating node objects and adding them to the scene. All drawing must be performed by the classes provided in Sprite Kit. You can customize the behavior of those classes to produce many different graphical effects. However, by controlling all drawing, Sprite Kit can apply many optimizations to how drawing is performed.

5. Build and run the project.

You should see a blue screen with "Hello, World!" in it. You've now learned the basics for drawing Sprite Kit content.

# Using Actions to Animate Scenes

Static text is nice, but it might be more interesting if the text was animated. Most of the time, you move things around the scene by executing **actions**. Most actions in Sprite Kit apply changes to a node. You create an action object to describe the changes you want, and then tell a node to run it. Then, when the scene is rendered, it executes the action, animating the changes over several frames until the action completes.

When the user touches inside the scene, the text animates and then fades away.

### To animate the text

1. Add the following code to the `newHelloNode` method:

```
helloNode.name = @"helloNode";
```

   All nodes have a `name` property that you can set to describe the node. You name a node when you want to be able to find it later or when you want to build behavior that is based on the node name. Later, you can search the tree for nodes that match the name.

   In this example, you give the label a name so that it can be discovered later. In an actual game, you might give the same name to any node that represents the same kind of content. For example, if your game represents each monster as a node, you might name the node `monster`.

2. Override the `touchesBegan:withEvent:` method on the scene class. When the scene receives a touch event, it finds the node named `helloNode` and tells it to run a short animation.

   All node objects are subclasses of `UIResponder` on iOS and `NSResponder` on OS X. This means that you can create subclasses of node classes in Sprite Kit to add interactivity to any node in the scene.

```
- (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 fadeOutWithDuration: 0.25];
        SKAction *remove = [SKAction removeFromParent];
```

```
        SKAction *moveSequence = [SKAction sequence:@[moveUp, zoom,
pause, fadeAway, remove]];
        [helloNode runAction: moveSequence];
    }
}
```

To prevent the node from responding to repeated presses, the code clears the node's name. Then, it builds action objects to perform various actions. After creating all of the actions, it creates a **sequence action** that combines these actions together; when the sequence runs, it performs each of the child actions in order. Finally, the method tells the label node to execute the sequence.

3. Build and run the project.

   You should see the text as before. At the bottom of the screen, the node count should be `1`. Now, tap inside the view. You should see the text animate and fade away. After it fades away, the node count should change to `0`, because the node was removed from the parent.

## Transitioning Between Scenes

Sprite Kit makes it easy to transition between scenes. You can either keep scenes around persistently, or dispose of them when you transition between them. In this example, you create a second scene class to learn some other game behaviors. When the "Hello, World!" text disappears from the screen, the code creates a new scene and transitions to it. The Hello scene is discarded after the transition.

### To create the Spaceship scene

1. Create a new class named `SpaceshipScene` and make it a subclass of the `SKScene` class.
2. Implement code to initialize the spaceship scene's contents. The code in the new scene's implementation file is similar to the code you implemented for the `HelloScene` class.

```
#import "SpaceshipScene.h"

@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;
}
@end
```

3. Import the `SpaceshipScene.h` header inside of the `HelloScene.m` file.

```
#import "SpaceshipScene.h"
```

4. In the `HelloScene` class's `touchesBegan:withEvent:` method, replace the call to `runAction:` with a new call to `runAction:completion:`. Implement a completion handler to create and present a new scene.

```
[helloNode runAction: moveSequence completion:^{
    SKScene *spaceshipScene  = [[SpaceshipScene alloc]
initWithSize:self.size];
    SKTransition *doors = [SKTransition
doorsOpenVerticalWithDuration:0.5];
    [self.view presentScene:spaceshipScene transition:doors];
}];
```

5. Build and run the project.

   When you touch inside the scene, the text fades out and then the view transitions to the new scene. You should see a black screen.

# Building Complex Content Using Nodes

The new scene doesn't have any content yet, so you are going to add a spaceship to the scene. To build the spaceship, you need to use multiple `SKSpriteNode` objects to create the spaceship and the lights on its surface. Each of the sprite nodes is going to execute actions.

Sprite nodes are the most common class used to create content in a Sprite Kit app. They can either draw untextured or textured rectangles. In this example, you are going to use untextured objects. Later, these placeholders could be easily replaced with textured sprites without changing their behavior. In an actual game, you might need dozens or hundreds of nodes to create the visual content of your game. But, fundamentally, those sprites are going to use the same techniques as this simple example.

Although you could add all three sprites directly to the scene, that isn't the Sprite Kit way. The blinking lights are part of the spaceship! If the spaceship moves, the lights should move with it. The solution is to make the spaceship node their parent, in the same way that the scene is going to be the parent of the spaceship. The coordinates of the lights are going to be specified relative to the parent node's position, which is at the center of the sprite image.

### To add the spaceship

1. In `SpaceshipScene.m`, add code to the `createSceneContents` method to create the spaceship.

```
SKSpriteNode *spaceship = [self newSpaceship];
spaceship.position = CGPointMake(CGRectGetMidX(self.frame),
                  CGRectGetMidY(self.frame)-150);
[self addChild:spaceship];
```

2. Implement the `newSpaceship` method.

```
- (SKSpriteNode *)newSpaceship
{
    SKSpriteNode *hull = [[SKSpriteNode alloc] initWithColor:[SKColor
grayColor] size:CGSizeMake(64,32)];

    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; }
```

This method creates the spaceship's hull and adds to it a short animation. Note that a new kind of action was introduced. A repeating action continuously repeats the action passed to it. In this case, the sequence repeats indefinitely.

3. Build and run the project.

   You should see a single rectangle for the spaceship's hull.

4. Add code to the `newSpaceship` method to add the lights.

   Insert the folllwing code after the line that creates the hull sprite.

```
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];
```

   When building complex nodes that have children, it is a good idea to isolate the code used to create the node behind a construction method or even a subclass. This makes it easier to change the sprite's composition and behavior without requiring changes to clients that use the sprite.

5. Implement the `newLight` method.

```
- (SKSpriteNode *)newLight
{
    SKSpriteNode *light = [[SKSpriteNode alloc] initWithColor:[SKColor
yellowColor] size:CGSizeMake(8,8)];

    SKAction *blink = [SKAction sequence:@[
                        [SKAction fadeOutWithDuration:0.25],
                        [SKAction fadeInWithDuration:0.25]]];
    SKAction *blinkForever = [SKAction repeatActionForever:blink];
    [light runAction: blinkForever];
```

```
    return light;
}
```

6. Build and run the project.

   You should see a pair of lights on the spaceship. When the spaceship moves, the lights move with it. All three nodes are continuously animated. You could add additional actions to move the lights around the ship; they would always move relative to the ship's hull.

## Creating Nodes That Interact with Each Other

In an actual game, you usually need nodes to interact with each other. There are many ways to add behavior to sprites, so this example shows only one of them. You will add new nodes to the scene and use the physics subsystem to simulate their movement and implement collision effects.

Sprite Kit provides a complete physics simulation which you can use to add automatic behaviors to nodes. That is, instead of executing actions on the nodes, physics is automatically simulated on the node, causing it to move. When it interacts with other nodes that are part of the physics system, collisions are automatically calculated and performed.

### To add physics simulation to the Spaceship scene

1. Change the `newSpaceship` method to add a physics body to the spaceship.

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

2. Build and run the project.

   The spaceship plummets through the bottom of the screen. This is because a gravitational force is applied to the spaceship's physics body. Even though the move action is still running, physics effects are also applied to the spaceship.

3. Change the `newSpaceship` method to prevent the spaceship from being affected by physics interactions.

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

When you run it now, the spaceship is no longer affected by gravity, so it runs as it did before. Later, making the physics body static also means that the spaceship's velocity is unaffected by collisions.

4. Add code to the `createSceneContents` method to spawn rocks.

```
SKAction *makeRocks = [SKAction sequence: @[
    [SKAction performSelector:@selector(addRock) onTarget:self],
    [SKAction waitForDuration:0.10 withRange:0.15]
    ]];
[self runAction: [SKAction repeatActionForever:makeRocks]];
```

The scene is *also* a node, so it can run actions too. In this case, a custom action calls a method on the scene to create a rock. The sequence creates a rock, then waits for a random period of time. By repeating this action, the scene continuously spawns new rocks.

5. Implement the `addRock` method.

```
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] size:CGSizeMake(8,8)];
    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];
}
```

6. Build and run the project

Rocks should now fall from the top of the scene. When a rock hits the ship, the rock bounces off the ship. No actions were added to move the rocks. Rocks fall and collide with the ship entirely due to the physics subsystem.

The rocks are small and move quickly, so the code specifies precise collisions to ensure that all collisions are detected.

If you let the app run for a while, the frame rate starts to drop, even though the node count remains very low. This is because the node code only shows the visible nodes in the scene. However, when rocks fall through the bottom of the scene, they continue to exist in the scene, which means that physics is still being simulated on them. Eventually there are so many nodes being processed that Sprite Kit slows down.

7. Implement the `didSimulatePhysics` method in the scene, to remove rocks when they move offscreen.

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

Each time the scene processes a frame, it runs actions and simulates physics. Your game can hook into this process to execute other custom code. Now, when the app processes a new frame of animation, it processes physics and then removes any rocks that moved off the bottom of the screen. When you run the app, the frame rate remains constant.

Pre- and post-processing in a scene, combined with actions and physics, are the places you build your game's behavior.

And that's it—your first taste of Sprite Kit! Everything else is a refinement of the basic techniques you've seen here.

## Try This!

Here are a few things you can try:

- Make an OS X version of this example. The code you wrote in the view controller is often implemented in an app delegate on OS X. You also need to change the responder code to use mouse events, rather than touch events. But the rest of the code should be the same.
- Use textured sprites to represent the ship and rocks. (Hint: "Working with Sprites" (page 29))
- Try moving the spaceship in response to touch events. (Hint: "Adding Actions to Nodes" (page 42) and "Building Your Scene" (page 52)).
- Add additional graphical effects to the scene (Hint: "Working with Other Node Types" (page 68))
- Add other behaviors when rocks collide with the ship. For example, make the rocks explode. (Hint: "Simulating Physics" (page 88))
- Add sound effects. (Hint: Use a `playSoundFileNamed:waitForCompletion:` action)

# Working with Sprites

Sprites are the basic building blocks used to create the majority of your scene's content, so understanding sprites is useful before moving on to other node classes in Sprite Kit. Sprites are represented by `SKSpriteNode` objects. An `SKSpriteNode` object can be drawn either as a rectangle with a texture mapped onto it or as a colored, untextured rectangle. Textured sprites are more common, because they represent the primary way that you bring custom artwork into a scene. This custom artwork might represent characters in your game, background elements, or even user interface elements, but the basic strategy is the same. An artist creates the images, and your game loads them as textures. Then you create sprites with those textures and add them to the scene.

## Creating a Textured Sprite

The simplest way to create a textured sprite is to have Sprite Kit create both the texture and the sprite for you. You store the artwork in the app bundle, and then load it at runtime. Listing 2-1 shows how simple this code can be.

**Listing 2-1** Creating a textured sprite from an image stored in the bundle

```
SKSpriteNode *spaceship = [SKSpriteNode spriteNodeWithImageNamed:@"rocket.png"];
spaceship.position = CGPointMake(100,100);
[self addChild: spaceship];
```

When you create a sprite in this fashion, you get a lot of default behavior for free:

- The sprite is created with a frame that matches the texture's size.
- The sprite is rendered so that it is centered on its position. The sprite's `frame` property holds the rectangle that defines the area it covers.
- The sprite texture is alpha blended into the framebuffer.
- An `SKTexture` object is created and attached to the sprite. This texture object automatically loads the texture data whenever the sprite node is in the scene, is visible, and is necessary for rendering the scene. Later, if the sprite is removed from the scene or is no longer visible, Sprite Kit can delete the texture data if it needs that memory for other purposes. This automatic memory management simplifies but does not eliminate the work you need to do to manage art assets in your game.

The default behavior gives you a useful foundation for creating a sprite-based game. You already know enough to add artwork to your game, create sprites, and run actions on those sprites to do interesting things. As sprites move onscreen and offscreen, Sprite Kit does its best to efficiently manage textures and draw frames of animation. If that is enough for you, take some time to explore what you can do with sprites. Or, keep reading for a deeper dive into the `SKSpriteNode` class. Along the way, you'll gain a thorough understanding of its capabilities and how to communicate those capabilities to your artists and designers. And you will learn more advanced ways to work with textures and how to improve performance with texture-based sprites.

## Customizing a Textured Sprite

You can use each sprite's properties to independently configure four distinct rendering stages:

- Move a sprite's frame so that a different point in the texture is placed at the sprite node's position. See "Using the Anchor Point to Move the Sprite's Frame" (page 31).
- Resize a sprite. You control how the texture is applied to the sprite when the size of the sprite does not match the size of the texture. See "Resizing a Sprite" (page 32).
- Colorize a sprite's texture when it is applied to the sprite. See "Colorizing a Sprite" (page 34).
- Use other blend modes in a sprite to combine its contents with that of the framebuffer. Custom blend modes are useful for lighting and other special effects. See "Blending the Sprite into the Framebuffer" (page 35).

Often, configuring a sprite to perform these four steps—positioning, sizing, colorizing, and blending—is based on the artwork used to create the sprite's texture. This means that you rarely set property values in isolation from the artwork. You work with your artist to ensure that your game is configuring the sprites to match the artwork.

Here are some of the possible strategies you can follow:

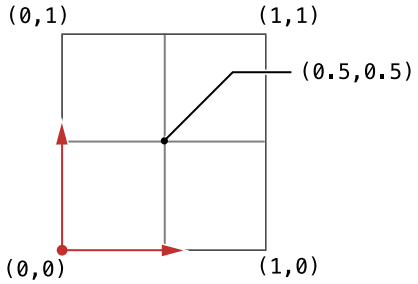
- Create the sprites with hardcoded values in your project. This is the fastest approach, but the least desirable in the long term, because it means that the code must be changed whenever the art assets change.
- Create your own tools using Sprite Kit that lets you fine tune the sprite's property values. When you have a sprite configured the way you want it, save the sprite to an archive. Your game uses the archive to create sprites at runtime.
- Store the configuration data in a property list that is stored in your app bundle. When the sprite is loaded, load the property list and use its values to configure the sprite. Your artist can then provide the correct values and change them without requiring changes to your code.

## Using the Anchor Point to Move the Sprite's Frame

By default, the sprite's frame—and thus its texture—is centered on the sprite's position. However, you might want a different part of the texture to appear at the node's position. You usually do this when the game element depicted in the texture is not centered in the texture image.

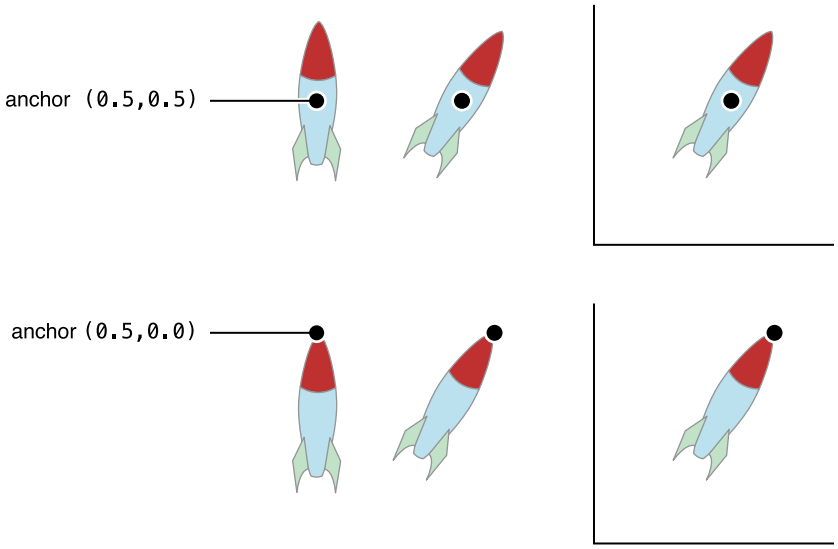
A sprite node's `anchorPoint` property determines which point in the frame is positioned at the sprite's position. Anchor points are specified in the unit coordinate system, shown in Figure 2-1. The unit coordinate system places the origin at the bottom left corner of the frame and `(1,1)` at the top right corner of the frame. A sprite's anchor point defaults to `(0.5,0.5)`, which corresponds to the center of the frame.

**Figure 2-1** The unit coordinate system




Although you are moving the frame, you do this because you want the corresponding portion of the texture to be centered on the position. Figure 2-2 shows a pair of texture images. In the first, the default anchor point centers the texture on the position. In the second, a point at the top of the image is selected instead. You can see that when the sprite is rotated, the texture image rotates around this point.

**Figure 2-2** Changing a sprite's anchor point



Listing 2-2 shows how to place the anchor point on the rocket's nose cone. Usually, you set the anchor point when the sprite is initialized, because it corresponds to the artwork. However, you can set this property at any time. The frame is immediately updated, and the sprite onscreen is updated the next time the scene is rendered.

**Listing 2-2** Setting a sprite's anchor point

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

## Resizing a Sprite

The size of the sprite's `frame` property is determined by the values of three other properties:

- The sprite's `size` property holds the base (unscaled) size of the sprite. When a sprite is initialized using the code in Listing 2-1 (page 29), the value of this property is initialized to be equal to the size of the sprite's texture.
- The base size is then scaled by the sprite's `xScale` and `yScale` properties inherited from the `SKNode` class.

For example, if the sprite's base size is `32` x `32` pixels and it has an `xScale` value of `1.0` and a `yScale` value of `2.0`, the size of the sprite's frame is `32` x `64` pixels.

> **Note:** The scaling values of the sprite's ancestors in the scene are also used to scale the sprite. This changes the effective size of the sprite without changing its actual frame value. See "A Node Applies Many of Its Properties to Its Descendants" (page 63).

When a sprite's frame is larger than its texture, the texture is stretched to cover the frame. Normally, the texture is stretched uniformly across the frame, as shown in Figure 2-3.

**Figure 2-3** A texture is stretched to cover the sprite's frame

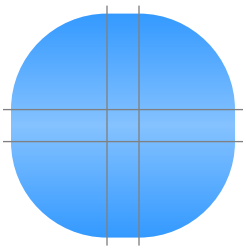
However, sometimes you want to use sprites to build user interface elements, such as buttons or health indicators. Often, these elements contain fixed-size elements, such as end caps, that should not be stretched. In this case, use a portion of the texture without stretching, and then stretch the remaining part of the texture over the rest of the frame.

The sprite's `centerRect` property, which is specified in unit coordinates of the texture, controls the scaling behavior. The default value is a rectangle that covers the entire texture, which is why the entire texture is stretched across the frame. If you specify a rectangle that only covers a portion of the texture, you create a `3 x 3` grid. Each box in the grid has its own scaling behavior:

- The portions of the texture in the four corners of the grid are drawn without any scaling.
- The center of the grid is scaled in both dimensions.
- The upper- and lower-middle parts are only scaled horizontally.
- The left- and right-middle parts are only scaled vertically.

Figure 2-4 shows a close-up view of a texture you might use to draw a user interface button. The complete texture is `28 x 28` pixels. The corner pieces are each `12 x 12` pixels and the center is `4 X 4` pixels.

**Figure 2-4** A stretchable button texture



Listing 2-3 shows how this button sprite would be initialized. The `centerRect` property is computed based on the design of the texture.

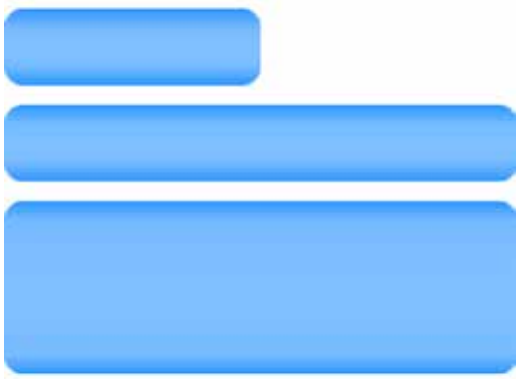
**Listing 2-3** Setting the sprite's center rect to adjust the stretching behavior

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

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

Figure 2-5 shows that the corners remain the same, even when the button is drawn at different sizes.

**Figure 2-5** Applying the button texture to buttons of different sizes



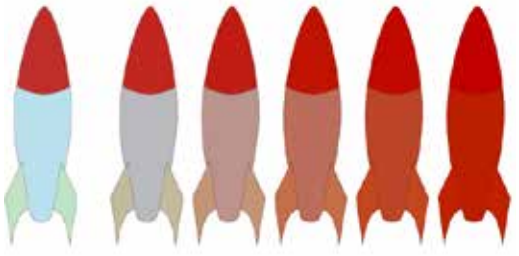
## Colorizing a Sprite

You can use the `color` and `colorBlendFactor` properties to colorize the texture before applying it to the sprite. The color blend factor defaults to `0.0`, which indicates that the texture should be used unmodified. As you increase this number, more of the texture color is replaced with the blended color. For example, when a monster in your game takes damage, you might want to add a red tint to the character. Listing 2-4 shows how you would apply a tint to the sprite.

**Listing 2-4** Tinting the color of the sprite

```
monsterSprite.color = [SKColor redColor];
monsterSprite.colorBlendFactor = 0.5;
```

**Figure 2-6** Colorizing adjusts the color of the texture



You can also animate the color and color blend factors using actions. Listing 2-5 shows how to briefly tint the sprite and then return it to normal.

**Listing 2-5** Animating a color change

```
SKAction *pulseRed = [SKAction sequence:@[
                        [SKAction colorizeWithColor:[SKColor redColor]
colorBlendFactor:1.0 duration:0.15],
                        [SKAction waitForDuration:0.1],
                      [SKAction colorizeWithColorBlendFactor:0.0 duration:0.15]]];
     [monsterSprite runAction: pulseRed];
```

## Blending the Sprite into the Framebuffer

The final stage of rendering is to blend the sprite's texture into its destination framebuffer. The default behavior uses the alpha values of the texture to blend the texture with the destination pixels. However, you can use other blend modes when you want to add other special effects to a scene.

You control the sprite's blending behavior using the `blendMode` property. For example, an additive blend mode is useful to combine multiple sprites together, such as for fire or lighting. Listing 2-6 shows how to change the blend mode to use an additive blend.

**Listing 2-6** Using an additive blend mode to simulate a light

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

# Working with Texture Objects

Although Sprite Kit can create textures for you automatically when a sprite is created, in more complex games you need more control over textures. For example, you might want to do any of the following:

- Share a texture between multiple sprites.
- Change a sprite's texture after it is created.
- Animate a sprite through a series of textures.
- Create textures from data that is not directly stored in the app bundle.
- Render a node tree into a texture. For example, you might want to take a screenshot of your gameplay to show to the player after he or she completes the level.
- Preload textures into memory before presenting a scene.

You do all of these things by working directly with `SKTexture` objects. You create an `SKTexture` object and then use it to create new sprites or change the texture of an existing sprite.

## Creating a Texture from an Image Stored in the App Bundle

Listing 2-7 shows an example similar to Listing 2-1 (page 29), but now the code explicitly creates a texture object. The then code creates multiple rockets from the same texture.

**Listing 2-7** Loading a texture from the bundle

```
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];
}
```

The texture object itself is just a placeholder for the actual texture data. The texture data is more resource intensive, so Sprite Kit loads it into memory only when needed.

## Using Texture Atlases to Collect Related Art Assets

Art assets stored in your app bundle aren't always unrelated images. Sometimes they are collections of images that are being used together for the same sprite. For example, here are a few common collections of art assets:

- Animation frames for a character
- Terrain tiles used to create a game level or puzzle
- Images used for user interface controls, such as buttons, switches, and sliders

If each texture is treated as a separate object, then Sprite Kit and the graphics hardware must work harder to render scenes—and your game's performance might suffer. Specifically, Sprite Kit must make at least one drawing pass per texture. To avoid making multiple drawing passes, Sprite Kit uses **texture atlases** to collect related images together. You specify which assets should be collected together, and Xcode builds a texture atlas automatically. Then, when your game loads the texture atlas, Sprite Kit manages all the images inside the atlas as if they were a single texture. You continue to use `SKTexture` objects to access the elements contained in the atlas.

## Creating a Texture Atlas

Xcode can automatically build texture atlases for you from a collection of images. The process is described in detail in *Texture Atlas Help* .

When you create a texture atlas, you want to strike a balance between collecting too many textures or too few. If you use too few images, Sprite Kit may still need many drawing passes to render a frame. If you include too many images, then large amounts of texture data may need to be loaded into memory at once. Because Xcode builds the atlases for you, you can switch between different atlas configurations with relative ease. So experiment with different configurations of your texture atlases and choose the combination that gives you the best performance.

## Loading Textures from a Texture Atlas

The code in Listing 2-7 (page 36) is also used to load textures from a texture atlas. Sprite Kit searches first for an image file with the specified filename. If it doesn't find one, it searches inside any texture atlases built into the app bundle. This means that you don't have to make any coding changes to support this in your game. This design also offers your artists the ability to experiment with new textures without requiring that your game be rebuilt. The artists drop the textures into the app bundle. When the app is relaunched, Sprite Kit automatically discovers the textures (overriding any previous versions built into the texture atlases). When the artists are satisfied with the textures, you then add those textures to the project and bake them into your texture atlases.

If you want to explicitly work with texture atlases, use the `SKTextureAtlas` class. First, create a texture atlas object using the name of the atlas. Next, use the names of the image files stored in the atlas to look up the individual textures. Listing 2-8 shows an example of this. It uses a texture atlas that holds multiple frames of animation for a monster. The code loads those frames and stores them in an array. In the actual project, you would add a `monster.atlas` folder with the four image files.

**Listing 2-8** Loading textures for a walk animation

```
SKTextureAtlas *atlas = [SKTextureAtlas atlasNamed:@"monster"];
SKTexture *f1 = [atlas textureNamed:@"monster-walk1.png"];
SKTexture *f2 = [atlas textureNamed:@"monster-walk2.png"];
SKTexture *f3 = [atlas textureNamed:@"monster-walk3.png"];
SKTexture *f4 = [atlas textureNamed:@"monster-walk4.png"];
NSArray *monsterWalkTextures = @[f1,f2,f3,f4];
```

## Creating a Texture from a Subsection of a Texture

If you already have an `SKTexture` object, you can create new textures that reference a portion of it. This approach is efficient because the new texture objects reference the same texture data in memory. (This behavior is similar to that of the texture atlas.) Typically, you use this approach if your game already has its own custom texture atlas format. In this case, you are responsible for storing the coordinates for the individual images stored in the custom texture atlas.

Listing 2-9 shows how to extract a portion of a texture. The coordinates for the rectangle are in the unit coordinate space.

**Listing 2-9** Using part of a texture

```
SKTexture *bottomLeftTexture = [SKTexture textureWithRect:CGRectMake(0.0,0.0,0.5,0.5)
 inTexture:cornerTextures];
```

## Other Ways to Create Textures

In addition to loading textures from the app bundle, you can create textures from other sources:

- Use the `SKTexture` initializer methods to create textures from properly formatted pixel data in memory, from core graphics images, or by applying a Core Image filter to an existing texture.
- The `SKView` class's `textureFromNode:` method can render a node tree's content into a texture. The texture is sized so that it holds the contents of the node and all of its visible descendants.

## Changing a Sprite's Texture

A sprite's `texture` property points to its current texture. You can change this property to point to a new texture. The next time the scene renders a new frame, it renders with the new texture. Whenever you change the texture, you may also need to change other sprite properties—such as `size`, `anchorPoint`, and `centerRect`—to be consistent with the new texture. Usually, it is better to ensure that all the artwork is consistent so that the same values can be used for all of the textures. That is, the textures should have a consistent size and anchor point placement so that your game does not need to update anything other than the texture.

Because animation is a common task, you can use actions to animate a series of textures on a sprite. The code in Listing 2-10 shows how to use the array of animation frames created in Listing 2-8 (page 37) to animate a sprite's texture.

**Listing 2-10** Animating through a series of textures

```
SKAction *walkAnimation = [SKAction animateWithTextures:monsterWalkTextures
timePerFrame:0.1]

[monster runAction:walkAnimation];

// insert other code here to move the monster.
```

Sprite Kit provides the plumbing that allows you to animate or change a sprite's texture. It doesn't impose a specific design on your animation system. This means you need to determine what kinds of animations that a sprite may need and then design your own animation system to switch between those animations at runtime. For example, a monster might have walk, fight, idle, and death animation sequences—and it's up to you to decide when to switch between these sequences.

## Preloading Textures Into Memory

A major advantage to Sprite Kit is that it performs a lot of memory management for you automatically. When rendering a new frame of animation, Sprite Kit determines whether a texture is needed to render the current frame. If a texture is needed but is not prepared for rendering, Sprite Kit loads the texture data from the file, transforms the data into a format that the graphics hardware can use, and uploads it to the graphics hardware. This process happens automatically in the background, but it isn't free. If too many unloaded textures are needed at once, it may be impossible to load all the textures in a single frame of animation, causing the frame rate to stutter. To avoid this problem, you need to preload textures into memory, particularly in larger or complex games.

Listing 2-11 shows how to preload an array of `SKTexture` objects. The `preloadTextures:withCompletionHandler:` method calls the completion handler after all of the textures are loaded into memory. In this example, all of the textures for a particular level of the game are preloaded in a single operation. When the textures are all in memory, the completion handler is called. It creates the scene and presents it. (You need to add code to provide these texture objects to the scene; that code isn't shown here).

**Listing 2-11** Preloading a texture

```
[SKTexture preloadTextures:textureArrayForLevel1 withCompletionHandler:^
    {
        // The textures are loaded into memory. Start the level.
        GamePlayScene* gameScene = [[GamePlayScene alloc]
initWithSize:CGSizeMake(768,1024)];
        SKView *spriteView = (SKView *) self.view;
        [spriteView presentScene: gameScene];
```

```
    }];
```

Because you are intimately familiar with the design of your game, you are the best person to know when new textures are needed. The exact design of your preloading code is going to depend on your game engine. Here are a few possible designs to consider:

- For a small game, you may be able to preload all of its textures when the app is launched, and then keep them in memory forever.
- For a larger game, you may need to split the textures into levels or themes. Each level or theme's textures are designed to fit in a specific amount of memory. When the player starts a new level, you preload all of that level's texture objects. When the player finishes playing the level, the textures not needed for the next level are discarded. By preloading the levels, the load time is all up front, before gameplay starts.
- If a game needs more textures than can fit into memory, you need to preload textures dynamically as the game is being played. Typically, you preload some textures at the start of a level, and then load other textures when you think they will be needed soon. For example, in a racing game, the player is always moving in the same direction, so for each frame you might fetch a new texture for content the player is about to see. The textures are loaded in the background, displacing the oldest textures for the track. In an adventure game that allows for player-controlled movement, you might have to provisionally load textures when a player is moving in a particular direction.

### Removing a Texture From Memory

After a texture is loaded into the graphics hardware's memory, it stays in memory until the referencing `SKTexture` object is deleted. This means that between levels (or in a dynamic game), you may need to make sure a texture object is deleted. Delete a `SKTexture` object object by removing any strong references to it, including:

- All texture references from `SKSpriteNode` and `SKEffectNode` objects in your game
- Any strong references to the texture in your own code
- An `SKTextureAtlas` object that was used to create the texture object

## Creating Untextured Sprites

Although textured sprites are the most common way to use the `SKSpriteNode` class, you can also create sprite nodes without a texture. The behavior of the class changes when the sprite lacks a texture:

- There is no texture to stretch, so the `centerRect` parameter is ignored.

- There is no colorization step; the `color` property is used as the sprite's color.
- The color's `alpha` component is used to determine how the sprite is blended into the buffer.

The other properties (`size`, `anchorPoint`, and `blendMode`) work the same.

## Try This!

Now that you know more about sprites, try some of the following activities:

- Add artwork to your project in a texture atlas. (Hint: "Creating a Texture Atlas" (page 37))
- Load the texture atlas and use it to create new sprites. (Hint: "Loading Textures from a Texture Atlas" (page 37))
- Animate sprites through multiple frames of animation. (Hint Listing 2-10 (page 39))
- Change the properties of your sprites and see how their drawing behavior changes. (Hint "Customizing a Textured Sprite" (page 30))

You can find useful code in the *Sprite Tour* sample.

# Adding Actions to Nodes

Drawing sprites is useful, but a static image is a picture, not a game. To add gameplay, you need to be able to move sprites around the screen and perform other logic. The primary mechanism that Sprite Kit uses to animate scenes is actions. Up to this point you've seen some part of the action subsystem. Now, it's time to get a deeper appreciation for how actions are constructed and executed.

An action is an object that defines a change you want to make to the scene. In most cases, an action applies its changes to the node that is executing it. So, for example, if you want to move a sprite across the screen, you create a move action and tell the sprite node to run that action. Sprite Kit automatically animates that sprite's position until the action completes.

## Actions Are Self-Contained Objects

Every action is an opaque object that describes a change you want to make to the scene. All actions are implemented by the `SKAction` class; there are no visible subclasses. Instead, actions of different types are instantiated using class methods. For example, here are the most common things you use actions to do:

- Changing a node's position and orientation
- Changing a node's size or scaling properties
- Changing a node's visibility or making it translucent
- Changing a sprite node's contents so that it animates through a series of textures
- Colorizing a sprite node
- Playing simple sounds
- Removing a node from the node tree
- Calling a block
- Invoking a selector on an object

After you create an action, its type cannot be changed, and you have a limited ability to change its properties. Sprite Kit takes advantage of the immutable nature of actions to execute them very efficiently.

**Tip:** Because actions are effectively immutable objects, you can run the same action safely on multiple nodes in the tree at the same time. For this reason, if you have an action that is used repeatedly in your game, create a single instance of the action and then reuse it whenever you need a node to execute it.

Actions can either be instantaneous or non-instantaneous:

- An instantaneous action starts and completes in a single frame of animation. For example, an action to remove a node from its parent is an instantaneous action because a node can't be partially removed. Instead, when the action executes, the node is removed immediately.
- A non-instantaneous action has a duration over which it animates its effects. When executed, the action is processed in each frame of animation until the action completes.

The complete list of class methods used to create actions is described in *SKAction Class Reference*, but you only need to go there when you are ready for a detailed look at how to configure specific actions.

## Nodes Run Actions

An action is only executed after you tell a node to run it. The simplest way to run an action is to call the node's `runAction:` method. Listing 3-1 creates a new move action and then tells the node to execute it.

**Listing 3-1** Running an action

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

A move action has a duration, so this action is processed by the scene over multiple frames of animation until the elapsed time exceeds the duration of the action. After the animation completes, the action is removed from the node.

You can run actions at any time. However, if you add actions to a node while the scene is processing actions, the new actions may not execute until the following frame. The steps a scene uses to process actions are described in more detail in "Advanced Scene Processing" (page 81).

A node can run multiple actions simultaneously, even if those actions were executed at different times. The scene keeps track of how far each action is from completing and computes the effect that the action has on the node. For example, if you run two actions that move the same node, both actions apply changes to every frame. If the move actions were in equal and opposite directions, the node would remain stationary.

Because action processing is tied to the scene, actions are processed only when the node is part of a presented scene's node tree. You can take advantage of this feature by creating a node and assigning actions to it, but waiting until later to add the node to the scene. Later, when the node is added to the scene, it begins executing its actions immediately. This pattern is particularly useful because the actions that a node is running are copied and archived when the node is copied.

If a node is running any actions, its `hasActions` property returns `YES`.

## Canceling Running Actions

To cancel actions that a node is running, call its `removeAllActions` method. All actions are removed from the node immediately. If a removed action had a duration, any changes it already made to the node remain intact, but further changes are not executed.

## Receiving a Callback when an Action Completes

The `runAction:completion:` method is identical to the `runAction:` method, but after the action completes, your block is called. This callback is only called if the action runs to completion. If the action is removed before it completes, the completion handler is never called.

## Using Named Actions for Precise Control over Actions

Normally, you can't see which actions a node is executing and if you want to remove actions, you must remove all of them. If you need to see whether a particular action is executing or remove a specific action, you must use **named actions**. A named action uses a unique key name to identify the action. You can start, remove, find, and replace named actions on a node.

Listing 3-2 is similar to Listing 3-1 (page 43), but now the action is identified with a key, `ignition`.

**Listing 3-2** Running a named action

```
[SKAction *moveNodeRight = [SKAction moveByX:100.0 y:0.0 duration:1.0];
[spaceship runAction: moveNodeRight withKey:@"ignition"];
```

The following key-based methods are available:

- `runAction:withKey:` method to run the action. If an action with the same key is already executing, it is removed before the new action is added.
- `actionForKey:` method to determine if an action with that key is already running.
- `removeActionForKey:` method to remove the action.

Listing 3-3 shows how you might use a named action to control a sprite's movement. When the user clicks inside the scene, the method is invoked. The code determines where the click occurred and then tells the sprite to run an action to move to that position. The duration is calculated ahead of time so that the sprite always appears to move at a fixed speed. Because this code uses the `runAction:withKey:` method, if the sprite was already moving, the previous move is stopped mid-stream and the new action moves from the current position to the new position.

**Listing 3-3** Moving a sprite to the most recent mouse-click position

```
- (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"];
}
```

## Creating Actions That Run Other Actions

Sprite Kit provides many standard action types that change the properties of nodes in your scene. But actions show their real power when you combine them. By combining actions, you can create complex and expressive animations that are still executed by running a single action. A **compound action** is as easy to work with as any of the basic action types. With that in mind, it is time to learn about sequences, groups, and repeating actions.

- A *sequence action* (or **sequence**) has multiple child actions. Each action in the sequence begins after the previous action ends.
- A *group action* (or **group**) has multiple child actions. All actions stored in the group begin executing at the same time.
- A **repeating action** has a single child action. When the child action completes, it is restarted.

## Sequences Run Actions in Series

A sequence is a set of actions that run consecutively. When a node runs a sequence, the actions are triggered in consecutive order. When one action completes, the next action starts immediately. When the last action in the sequence completes, the sequence action also completes.

Listing 3-4 shows that a sequence is created using an array of other actions.

**Listing 3-4** Creating a sequence of actions

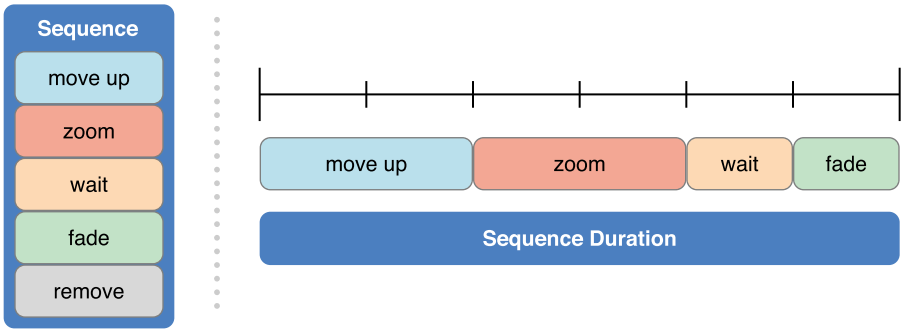
```
SKAction *moveUp = [SKAction moveByX:0 y:100.0 duration:1.0];
SKAction *zoom = [SKAction scaleTo:2.0 duration:0.25];
SKAction *wait = [SKAction waitForDuration: 0.5];
SKAction *fadeAway = [SKAction fadeOutWithDuration:0.25];
SKAction *removeNode = [SKAction removeFromParent];

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

There are a few things worth noting in this example:

- The `wait` action is a special action that is usually used only in sequences. This action simply waits for a period of time and then ends, without doing anything; you use them to control the timing of a sequence.
- The `removeNode` action is an instantaneous action, so it takes no time to execute. You can see that although this action is part of the sequence, it does not appear on the timeline in Figure 3-1. As an instantaneous action, it begins and completes immediately after the fade action completes. This action ends the sequence.

**Figure 3-1** Move and zoom sequence timeline



## Groups Run Actions in Parallel

A group action is a collection of actions that all start executing as soon as the group is executed. You use groups when you want actions to be synchronized. For example, the code in Listing 3-5 rotates and turns a sprite to give the illusion of a wheel rolling across the screen. Using a group (rather than running two separate actions) emphasizes that the two actions are closely related.

**Listing 3-5** Using a group of actions to rotate a wheel

```
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];
```

Although the actions in a group start at the same time, the group does not complete until the last action in the group has finished running. Listing 3-6 shows a more complex group that includes actions with different timing values. The sprite animates through its textures and moves down the screen for a period of two seconds. However, during the first second, the the sprite zooms in and changes from full transparency to a solid appearance. Figure 3-2 shows that the two actions that make the sprite appear finish halfway through the group's animation. The group continues until the other two actions complete.

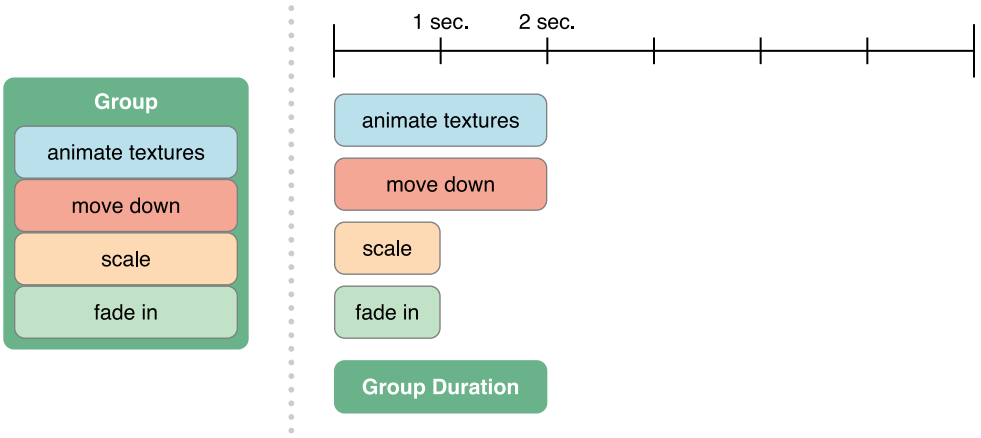
**Listing 3-6** Creating a group of actions with different timing values

```
[sprite setScale: 0];
SKAction *animate = [SKAction animateWithTextures:textures
timePerFrame:2.0/numberOfTextures];
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];
```

**Figure 3-2** Grouped actions start at the same time, but complete independently




## Repeating Actions Execute Another Action Multiple Times

A repeating action loops another action so that it repeats multiple times. When a repeating action is executed, it executes its contained action. Whenever the looped action completes, it is restarted by the repeating action. Listing 3-7 shows the creation methods used to create repeating actions. You can create an action that repeats an action a finite number of times or an action that repeats an action indefinitely.

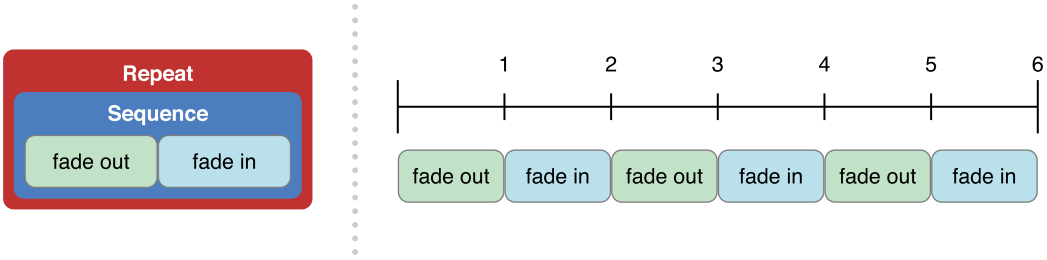
**Listing 3-7** Creating repeating actions

```
SKAction *fadeOut = [SKAction fadeOutWithDuration: 1];
SKAction *fadeIn = [SKAction fadeInWithDuration: 1];
SKAction *pulse = [SKAction sequence:@[fadeOut,fadeIn]];

SKAction *pulseThreeTimes = [SKAction repeatAction:pulse count:3];
SKAction *pulseForever = [SKAction repeatActionForever:pulse];
```

Figure 3-3 shows the timing arrangement for the `pulseThreeTimes` action. You can see that the sequence finishes, then repeats.

**Figure 3-3** Timing for a repeating action



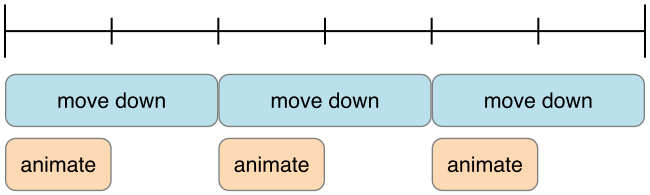

When you repeat a group, the entire group must finish before the group is restarted. Listing 3-8 creates a group that moves a sprite and animates its textures, but in this example the two actions have different durations. Figure 3-4 shows the timing diagram when the group is repeated. You can see that the texture animation runs to completion and then no animation occurs until the group repeats.

**Listing 3-8** Repeating a group animation

```
SKAction *animate = [SKAction animateWithTextures:textures
timePerFrame:1.0/numberOfImages];
SKAction *moveDown = [SKAction moveByX:0 y:-200 duration:2.0];


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

**Figure 3-4** Timing for a repeated group




What you may have wanted was for each action to repeat at its own natural frequency. To do this, create a set of repeating actions and then group them together. Listing 3-9 shows how you would implement the timing shown in Figure 3-5.

**Listing 3-9** Grouping a set of repeated actions

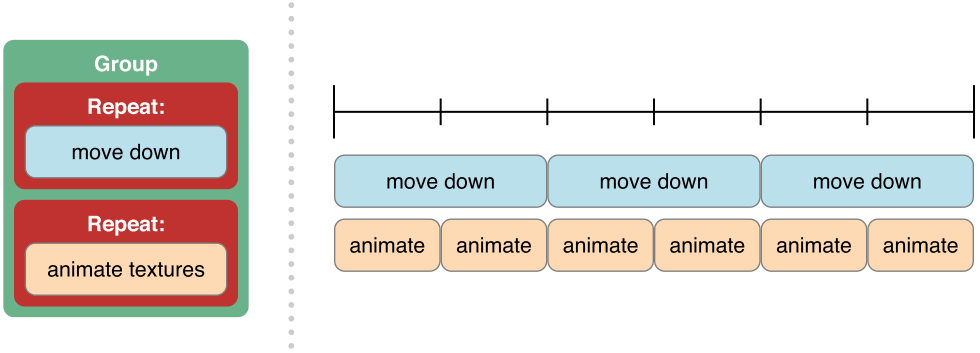
```
SKAction *animate = [SKAction animateWithTextures:textures
timePerFrame:1.0/numberOfImages];
```

```
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]];
```

**Figure 3-5** Each action repeats at its natural interval




## Configuring Action Timing

By default, an action with a duration applies its changes linearly over the duration you specified. However, you can adjust the timing of animations through a few properties:

- Normally, an animated action runs linearly. You can use an action's `timingMode` property to choose a nonlinear timing mode for an animation. For example, you can have the action start quickly and then slow down over the remainder of the run.
- An action's `speed` property changes the rate at which an animation plays. You can speed up or slow down an animation from its default timing.

  A speed value of `1.0` is the normal rate. If you set an action's speed property to `2.0`, when the action is executed by a node, it plays twice as fast. To pause the action, set the value to `0`.

  If you adjust the speed of an action that contains other actions (such as a group, sequence, or repeating action), the rate is applied to the actions contained within. The enclosed actions are also affected by their own `speed` property.
- A node's `speed` property has the same effect as the action's `speed` property, but the rate is applied to *all* actions processed by the node or by any of the node's descendants in the scene tree.

Sprite Kit determines the rate at which an animation applies by finding all of the rates that apply to the action and multiplying them.

# Tips for Working with Actions

Actions work best when you create them once and use them multiple times. Whenever possible, create actions early and save them in a location where they can be easily retrieved and executed.

Depending on the kind of action, any of the following locations might be useful:

- A node's `userData` property
- The parent node's `userData` property, if dozens of nodes share the same actions and the same parent
- The scene's `userData` property for actions shared by multiple nodes throughout the scene
- If subclassing, then on a property of the subclass

If you need designer or artist input on how a node's properties are animated, consider moving the action creation code into your custom design tools. Then archive the action and load it in your game engine. For more information, see "Sprite Kit Best Practices" (page 107).

## When You Shouldn't Use Actions

Although actions are efficient, there is a cost to creating and executing them. If you are making changes to a node's properties in every frame of animation and those changes need to be recomputed in each frame, you are better off making the changes to the node directly and not using actions to do so. For more information on where you might do this in your game, see "Advanced Scene Processing" (page 81).

# Try This!

Here are some things to try with actions:

- Try the different kinds of actions on your sprites. (Hint: *SKAction Class Reference*)
- Create an action group that synchronizes moving a sprite onscreen with another animation. (Hint: "Groups Run Actions in Parallel" (page 47))
- Use named actions to create cancelable actions. Connect those actions to your user interface code. (Hint: "Using Named Actions for Precise Control over Actions" (page 44))
- Create a sequence that tells an interesting story. For example, consider creating an animated title screen to display when your game is launched. (Hint "Sequences Run Actions in Series" (page 46))

# Building Your Scene

You have already learned many things about working with scenes. Here's a quick recap of the important facts:

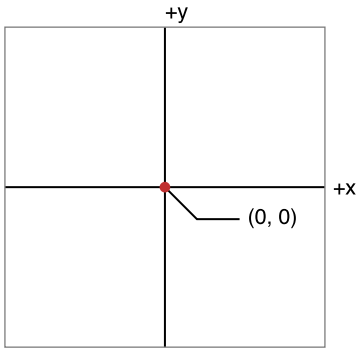
- Scenes (`SKScene` objects) are used to provide content to be rendered by an `SKView` object.
- A scene's content is created as a tree of node objects. The scene is the root node.
- When presented by a view, a scene runs actions and simulates physics, then renders the node tree.
- You create custom scenes by subclassing the `SKScene` class.

With those basic concepts in mind, it is time to learn more about the node tree and building your scenes.

## A Node Provides a Coordinate System to Its Children

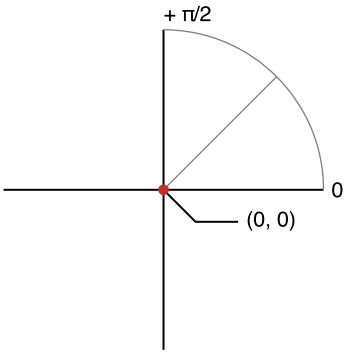
When a node is placed in the node tree, its `position` property places it within a coordinate system provided by its parent. Sprite Kit uses the same coordinate system on both iOS and OS X. Figure 4-1 shows the Sprite Kit coordinate system. Coordinate values are measured in points, as in UIKit or AppKit; where necessary, points are converted to pixels when the scene is rendered. A positive x coordinate goes to the right and a positive y coordinate goes up the screen.

**Figure 4-1** Sprite Kit coordinate system



Sprite Kit also has a standard rotation convention. Figure 4-2 shows the polar coordinate convention. An angle of `0` radians specifies the positive x axis. A positive angle is in the counterclockwise direction.

**Figure 4-2** Polar coordinate conventions (rotation)




When you are working only with Sprite Kit code, a consistent coordinate system means that you can easily share code between an iOS and OS X version of your game. However, it does mean that when you write OS-specific user interface code, you may need to convert between the operating system's view coordinate conventions and Sprite Kit's coordinate system. This is most often the case when working with iOS views, which use a different coordinate convention.

## Only Some Nodes Contain Content

Not all nodes draw content. For example, the `SKSpriteNode` class draws a sprite, but the `SKNode` class doesn't draw anything. You can tell whether a particular node object draws content by reading its `frame` property. The frame is the visible area of the parent's coordinate system that the node draws into. If the node draws content, this frame has a nonzero size. For a scene, the frame always reflects the visible portion of the scene's coordinate space.

If a node has descendants that draw content, it is possible for a node's subtree to provide content even though it doesn't provide any content itself. You can call a node's `calculateAccumulatedFrame` method to retrieve a rectangle that includes the entire area that a node and all of its descendants draw into.

## Creating a Scene

A scene is presented by a view. The scene includes properties that define where the scene's origin is positioned and the size of the scene. If the scene does not match the view's size, you can also define how the scene is scaled to fit in the view.

## A Scene's Size Defines Its Visible Area

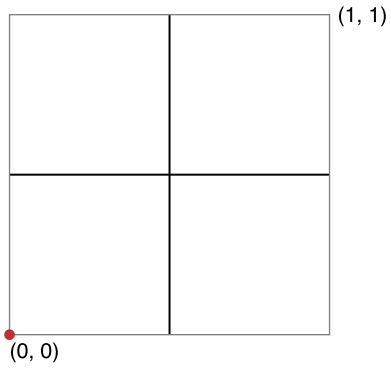
When a scene is first initialized, its `size` property is configured by the designated initializer. The size of the scene specifies the size of the visible portion of the scene in points. This is only used to specify the visible portion of the scene. Nodes in the tree can be positioned outside of this area; those nodes are still processed by the scene, but are ignored by the renderer.

## Using the Anchor Point to Position the Scene's Coordinate System in the View

By default, a scene's origin is placed in the lower-left corner of the view, as shown in Figure 4-3. So, a scene is initialized with a height of `1024` and a width of `768`, has the origin `(0,0)` in the lower-left corner, and the `(1024,768)` coordinate in the upper-right corner. The `frame` property holds `(0,0)–(1024,768)`.

A scene's `position` property is ignored by Scene Kit because the scene is always the root node for a node tree. Its default value is `CGPointZero` and you can't change it. However, you can move the scene's origin by setting its `anchorPoint` property. The anchor point is specified in the unit coordinate space and chooses a point in the enclosing view.

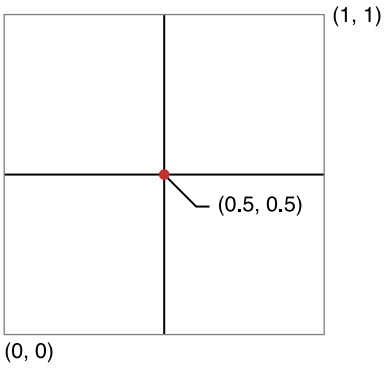
**Figure 4-3** Default anchor for a scene is in the lower-left corner of the view




The default value for the anchor point is `CGPointZero`, which places it at the lower-left corner. The scene's visible coordinate space is `(0,0)` to `(width,height)`. The default anchor point is most useful for games that do not scroll a scene's content.

The secondmost common anchor point value is `(0.5,0.5)`, which centers the scene's origin in the middle of the view as shown in Figure 4-4. The scene's visible coordinate space is `(-width/2,-height/2)` to `(width/2, height/2)`. Centering the scene on its anchor point is most useful when you want to easily position nodes relative to the center of the screen, such as in a scrolling game.

**Figure 4-4** Moving the anchor point to the center of the view




So, to summarize, the `anchorPoint` and `size` properties are used to compute the scene's frame, which holds the visible portion of the scene.

## A Scene's Contents Are Scaled to Fit the View

After a scene is rendered, its contents are copied into the presenting view. If the view and the scene are the same size, then the content can be directly copied into the view. If the two differ, then the scene is scaled to fit in the view. The `scaleMode` property determines how the content is scaled.

When you design your game, you should decide on a strategy for handling the scene's `size` and `scaleMode` properties. Here are the most common strategies:

- Instantiate the scene with a constant size and never change it. Pick a scaling mode that lets the view scale the scene's content. This gives the scene a predictable coordinate system and frame. You can then base your art assets and gameplay logic on this coordinate system.
- Adjust the size of the scene in your game. Where necessary, adjust your game logic and art assets to match the scene's size.
- Set the `scaleMode` property to `SKSceneScaleModeResizeFill`. Sprite Kit automatically resizes the scene so that it always matches the view's size. Where necessary, adjust your game logic and art assets to match the scene's size.

Listing 4-1 shows a typical implementation for when you plan to use a constant-sized scene. As with the example you created in "Jumping into Sprite Kit" (page 15), this code specifies a method to be executed the first time that the scene is presented. It configures the scene's properties, including its scaling mode, then adds

content. In this example, the scale mode is set to `SKSceneScaleModeAspectFit`, which scales the contents equally in both dimensions and ensures that all of the scene's contents are visible. Where necessary, this mode adds letterboxing.

**Listing 4-1** Using the scale mode for a fixed-size scene

```
- (void)createSceneContent
{
    self.scaleMode = SKSceneScaleModeAspectFit;
    self.backgroundColor = [SKColor blackColor];
// Add additional scene contents here.
...
}
```

If you expect a scene's size to change at runtime, then the initial scene size should be used to determine which art assets to use, as well as any game logic that is dependent on the scene size. Your game should also override the scene's `didChangeSize:` method, which is called whenever the scene changes size. When this method is called, you should update the scene's contents to match the new size.

## Creating the Node Tree

You create the node tree by creating parent-child relationships between nodes. Each node maintains an ordered list of children, referenced by reading the node's `children` property. The order of the children in the tree affects many aspects of scene processing, including hit testing and rendering. So, it is important to organize the node tree appropriately.

Table 4-1 lists the most common methods used to build the node tree. The complete list of methods is provided in *SKNode Class Reference*.

**Table 4-1** Common methods used to manipulate the node tree

| Method | Description |
| --- | --- |
| `addChild:` | Adds a node to the end of the receiver's list of child nodes. |
| `insertChild:atIndex:` | Inserts a child into a specific position in the receiver's list of child nodes. |
| `removeFromParent` | Removes the receiving node from its parent. |

When you need to directly transverse the node tree, you use the properties in Table 4-2 to uncover the tree's structure.

**Table 4-2** Transversing the node tree

| Property | Description |
|---|---|
| `children` | The array of `SKNode` objects that are the receiving node's children. |
| `parent` | If the node is a child of another node, this holds the parent. Otherwise, it holds `nil`. |
| `scene` | If the node is included anywhere in a scene, this returns the scene node that is the root of the tree. Otherwise it holds `nil`. |

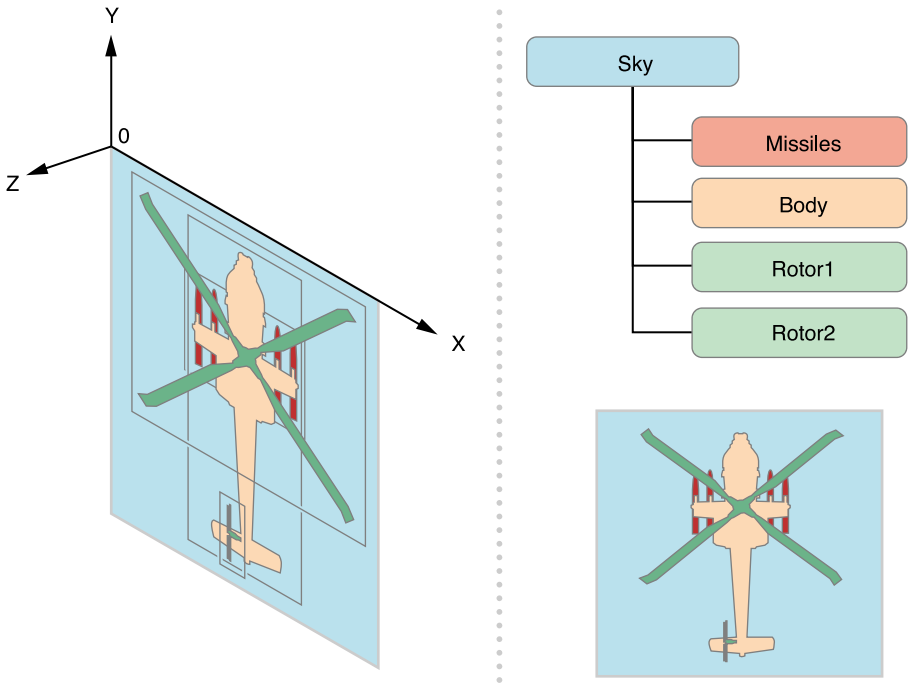
## Understanding the Drawing Order for a Node Tree

The standard behavior for scene rendering follows a simple pair of rules:

- A parent draws its content before rendering its children.
- Children are rendered in the order in which they appear in the child array.

Figure 4-5 shows how a node with four children are rendered.

**Figure 4-5** Parents are drawn before children



In the code you wrote in "Jumping into Sprite Kit" (page 15), you created a scene with a spaceship and rocks. Two lights were specified as children of the spaceship, and the spaceship and rocks were the scene's children. So the scene rendered its content as follows:

1. The scene renders itself, clearing its contents to its background color.
2. The scene renders the spaceship node.
3. The spaceship node renders its children, which are the lights on the spaceship.
4. The scene renders the rock nodes, which appear after the spaceship node in the scene's array of children.

Maintaining the order of a node's children can be a lot of work. Instead, you can give each node an explicit height in the scene. You do this by setting a node's `zPosition` property. The z position is the node's height relative to its parent node, much as a node's `position` property represents its x and y position relative to parent's position. So you use the z position to place a node above or below the parent's position.

When you take z positions into account, here is how the node tree is rendered:

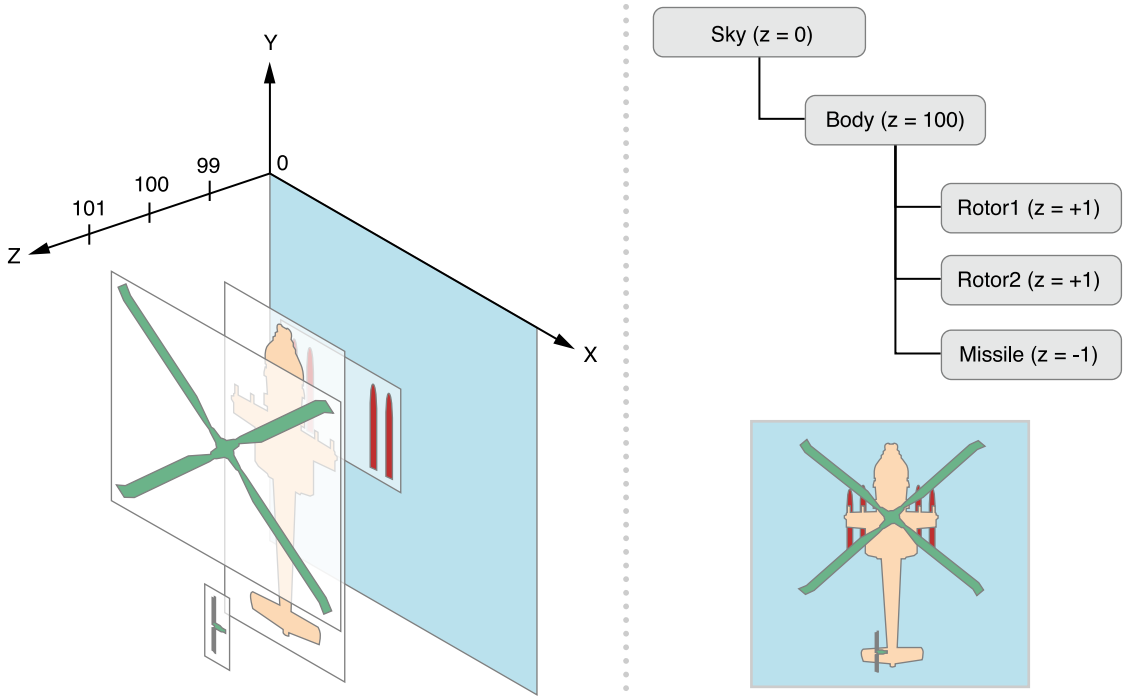
- Each node's global z position is calculated.
- Nodes are drawn in order from smallest z value to largest z value.
- If two nodes share the same z value, ancestors are rendered first, and siblings are rendered in child order.

As you've just seen, Sprite Kit uses a deterministic rendering order based on the height nodes and their positions in the node tree. But, because the rendering order is so deterministic, Sprite Kit may be unable to apply some rendering optimizations that it might otherwise apply. For example, it might be better if Sprite Kit could gather all of the nodes that share the same texture and drawing mode and and draw them with a single drawing pass. To enable these sorts of optimizations, you set the view's `ignoresSiblingOrder` property to `YES`.

When you ignore sibling order, Sprite Kit uses the graphics hardware to render the nodes so that they appear z order. It sorts nodes into a drawing order that reduces the number of draw calls needed to render the scene. But with this optimized drawing order, you cannot predict the rendering order for nodes that share the same height. The rendering order may change each time a new frame is rendered. In many cases, the drawing order of these nodes is not important. For example, if the nodes are at the same height but do not overlap on screen, they can be drawn in any order.

Figure 4-6 shows an example of a tree that uses z positions to determine the rendering order. In this example, the body of the helicopter is at a height of 100, and its children are rendered relative to its height. The two rotor nodes share the same height but do not overlap.

**Figure 4-6** Depth-only rendering can improve performance




To summarize, you can use both tree order and z positions to determine your scene's rendering order. When rendering a complex scene, you should disable the sorting behavior and use the z positions of nodes to create a deterministic scene order.

**Important:** The `SKCropNode` and `SKEffectNode` node classes alter the scene rendering behavior. The children of these nodes are rendered independently as a separate node tree, and the results are rendered into the tree that contains the crop or effect node. For more information, see "Working with Other Node Types" (page 68).

## The Hit-Testing Order Is the Reverse of Drawing Order

In a scene, when Sprite Kit processes touch or mouse events, it walks the scene to find the closest node that wants to accept the event. If that node doesn't want the event, Sprite Kit checks the next closest node, and so on. The order in which hit-testing is processed is essentially the reverse of drawing order.

For a node to be considered during hit-testing, its `userInteractionEnabled` property must be set to `YES`. The default value is `NO` for any node except a scene node. A node that wants to receive events needs to implement the appropriate responder methods from its parent class (`UIResponder` on iOS and `NSResponder` on OS X). This is one of the few places where you must implement platform-specific code in Sprite Kit.

Sometimes, you also want to look for nodes directly, rather than relying on the standard event-handling mechanisms. In Sprite Kit you can ask a node whether any of its descendants intersect a specific point in their coordinate system. Call the `nodeAtPoint:` method to find the first descendant that intersects the point, or use the `nodesAtPoint:` method to receive an array of all of the nodes that intersect the point.

## Using a Node's Depth to Add Other Effects

Sprite Kit uses the `zPosition` value only to determine the hit testing and drawing order. You can also the z position to implement your own game effects. For example, you might use the height of a node to determine how it is rendered or how it moves onscreen. In this way, you can simulate fog or parallax effects. Sprite Kit does not create these effects for you. Usually, you implement them by processing the scene immediately before it is rendered. See "Advanced Scene Processing" (page 81).

## Searching the Node Tree

The nodes in the tree are organized to determine the precise rendering order for the scene, not by the role those nodes play in your game. Because of this, the `SKNode` class provides the `name` property. You can name a node to differentiate it from other nodes in the tree, then search for those nodes later.

A node's name should be an alphanumeric string without any punctuation. Listing 4-2 shows how you might name three different nodes to distinguish them from each other.

**Listing 4-2** Naming a set of nodes

```
playerNode.name = @"player";
monsterNode1.name = @"goblin";
monsterNode2.name = @"ogre";
```

When you name nodes in your game, you should decide whether the names are unique or not. If you decide a node name is unique, then the name is intended to identify that node and no other. On the other hand, if a node name is not unique within your game, it usually represents a collection of related nodes. For example, in Listing 4-2, there are probably multiple goblins within the game, and you might want to identify them all with the same name. But the player might be a unique node within the game.

The node name usually serves two purposes in your app:

- You can write your own code that implements game logic based on the node's name. For example, when two physics objects collide, you might use the node names to determine how the collision affects gameplay.
- You can search for nodes that have a particular name.

The `SKNode` class implements two methods for searching the node tree:

- The `childNodeWithName:` method searches a node's children until it finds a matching node, then it stops and returns this node. This method is usually used to search for nodes with unique names.
- The `enumerateChildNodesWithName:usingBlock:` method searches a node's children and calls your block once for each matching node it finds. You use this method when you want to find all nodes that share the same name.

Listing 4-3 shows how you might create a method on your scene class to find the player node.

**Listing 4-3** Finding the player node

```
- (SKNode *)playerNode
{
    [return [self childNodeWithName:@"player"];
}
```

When this method is called on the scene, the scene searches its children (and only its children) for a node whose `name` property matches the search string, then returns the node. When specifying a search string, you can either specify the name of the node or a class name. For example, if you created your own subclass for the player node and named it `PlayerSprite`, then you could specify `PlayerSprite` as the search string instead of `player`; the same node would be returned.

## Advanced Searches

The default search only searches a node's children and must exactly match either the node's name or its class. However, Sprite Kit provides an expressive search syntax so that you can perform more advanced searches. For example, you could do the same search as before, but search the entire scene tree. Or you could search the node's children, but match a pattern, rather than requiring an exact match.

Table 4-3 describes the different syntax options. The search uses common regular expression semantics.

**Table 4-3** Search syntax options

| Syntax | Description |
| --- | --- |
| `/` | When placed at the start of the search string, this indicates that the search should be performed on the tree's root node. |
| `//` | When placed at the start of the search string, this specifies that the search should begin at the root node and be performed recursively across the entire node tree. It is not legal anywhere else in the search string. |
| `..` | The search should move up to the node's parent. |
| `/` | When placed anywhere but the start of the search string, this indicates that the search should move to the node's children. |
| `*` | The search matches zero or more characters. |
| [characters delimited by commas or dashes] | The search matches any of the characters contained inside the brackets. |
| alphanumeric characters | The search matches only the specified characters. |

Table 4-4 shows some useful search strings to help get you started.

**Table 4-4** Example searches

| Search string | Description |
| --- | --- |
| `/MyName` | This searches the root node's children and matches any node with the name `MyName`. |
| `//*` | This search string matches every node in the node tree. |
| `//MyName/..` | This searches the node tree and matches the parent node of every node named `MyName`. |
| `A[0–9]` | This searches the node's children and returns any child named `A0`, `A1`, …, `A9`. |
| `Abby/Normal` | This searches the node's grandchildren and returns any node whose name is `Normal` and whose parent is named `Abby`. |
| `//Abby/Normal` | This searches the node tree and returns any node whose name is `Normal` and whose parent is named `Abby`. |

# A Node Applies Many of Its Properties to Its Descendants

When you change a node's property, often the effects are propagated to the node's descendants. The net effect is that a child is rendered based not only on its own properties but also on the properties of its ancestors.

**Table 4-5** Properties that affect a node's descendants

| Property | Description |
|---|---|
| `xScale`, `yScale` | The node's coordinate system is scaled by these two factors. This property affects coordinate conversion, the node's frame, drawing, and hit testing. Its descendants are similarly scaled. |
| `zRotation` | The node's coordinate system is rotated. This property affects coordinate conversion, the node's frame, drawing, and hit testing. Its descendants are similarly scaled. |
| `alpha` | If the node is rendered using a blend mode, the alpha value is multiplied into any alpha value before the blend operation takes place. The descendants are similarly affected. |
| `hidden` | If a node is hidden, the node and its descendants are not rendered. |
| `speed` | The speed at which a node processes actions is multiplied by this value. The descendants are similarly affected. |

# Converting Between Coordinate Spaces

When working with the node tree, sometimes you need to convert a position from one coordinate space to another. For example, when specifying joints in the physics system, the joint positions are specified in scene coordinates. So, if you have those points in a local coordinate system, you need to convert them to the scene's coordinate space.

Listing 4-4 shows how to convert a node's position into the scene coordinate system. The scene is asked to perform the conversion. Remember that a node's position is specified in its parent's coordinate system, so the code passes `node.parent` as the node to convert from. You could perform the same conversion in reverse by calling the `convertPoint:toNode:` method.

**Listing 4-4** Converting a node to the scene coordinate system

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

One situation where you need to perform coordinate conversions is when you perform event handling. Mouse and touch events need to be converted from window coordinates to view coordinates, and from there into the scene. To simplify the code you need to write, Sprite Kit adds a few convenience methods:

- In iOS, use the `locationInNode:` and `previousLocationInNode:` on `UITouch` objects to convert a touch location into a node's coordinate system.
- In OS X, use the `locationInNode:` method on `NSEvent` objects to convert a mouse event into a node's coordinate system.

# Using Transitions Between Scenes

Scenes are the basic building blocks of games. Typically, you design self-contained scenes for the parts of your game, and then transition between these scenes as necessary. For example, you might create different scene classes to represent any or all of the following concepts:

- A loading scene to display while other content is loaded
- A main menu scene to choose what kind of game the user wants to play
- A scene to configure the details of the specific kind of game the user chose
- A scene that provides the gameplay
- A scene displayed when gameplay ends

When you present a new scene in a view that is already presenting a scene, you have the option of using a transition to animate the change from the old scene to the new scene. Using a transition provides some continuity, so that the scene change is not quite so abrupt. The complete list of transitions is available in *SKTransition Class Reference*.

## Transitioning Between Two Scenes

Typically, you transition to a new scene based on gameplay or user input. For example, if the user presses a button in your main menu scene, you might transition to a new scene to configure the match the player wants to play. Listing 5-1 shows how you might implement the event handler in a sprite. The handler first runs an animation on itself to highlight the button (not described here). Then, it creates a transition object and the new scene. Finally, it calls the view to present the new scene. The transition means that this change is animated.

**Listing 5-1** Transitioning to a new scene

```
- (void)mouseUp:(NSEvent *)theEvent
{
    [self runAction: self.buttonPressAnimation];
    SKTransition *reveal = [SKTransition
revealWithDirection:SKTransitionDirectionDown duration:1.0];
    GameConfigScene *newScene = [[GameConfigScene alloc] initWithSize:
CGSizeMake(1024,768)]];
```

```
//  Optionally, insert code to configure the new scene.
    [self.scene.view presentScene: newScene transition: reveal];
}
```

When the transition occurs, the scene property is immediately updated to point to the new scene. Then, the animation occurs. Finally, the strong reference to the old scene is removed. If you need to keep the scene around after the transition occurs, your app needs to keep its own strong reference to the old scene.

When organizing your game, it can be helpful to create a diagram that shows all the scenes in a game, the transitions that occur between scenes, and the data that must be passed to the new scene when a transition occurs. Unlike view controllers in iOS, Sprite Kit does not provide a built-in mechanism for passing data between scenes. If you need to provide data during a scene transition, you need to implement your own mechanism to configure the new scene. Typically, this means defining your own custom methods and properties on each scene or doing so in a protocol implemented by the custom scene class.

## Configuring Whether Animations Play During the Transition

The `pausesIncomingScene` and `pausesOutgoingScene` properties on the transition object define which animations are played during the transition. By default, both scenes continue to process animation during the transition. However, you might want to pause one or both of the scenes until the transition completes. For example, consider the code again in Listing 5-1 (page 65). Because the button is going to run an action, this code expects the outgoing scene to be animated. But perhaps the incoming scene should not animate its content until the transition completes. Adding the code in Listing 5-2 has the desired effect.

**Listing 5-2** Pausing frame processing during a transition

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

## Detecting When a Scene is Presented

Sometimes you need to be able to detect that a scene has been presented or removed from a view. You do so by implementing one or both of the following methods on your custom scene class:

- Implement the `willMoveFromView:` method, called when the scene is about to be removed from the view.
- Implement the `didMoveToView:` method, called when the scene has just finished being presented by the view.

When a scene is presented without a transition, the old scene is removed first, and then the new scene is presented. When a transition is used, the new scene is added first, then the transition occurs, and finally the old scene is removed.

# Working with Other Node Types

Although sprites are the most important element you use when building a game, Sprite Kit provides many other node classes. Many of these node classes provide visual content, similar to the `SKSpriteNode` class. Others do not directly draw content of their own, but instead modify the behavior of their descendants in the node tree. Table 6-1 lists all the node classes provided by Sprite Kit, including the `SKScene` and `SKSpriteNode` classes you are already familiar with.

**Table 6-1** Sprite Kit node classes

| Class | Description |
| --- | --- |
| `SKNode` | The class from which all node classes are derived. It does not draw anything. |
| `SKScene` | A scene is the root node in the node tree. It handles animation and action processing. |
| `SKSpriteNode` | A node that draws a textured sprite. |
| `SKLabelNode` | A node that renders a text string. |
| `SKShapeNode` | A node that renders a shape based on a Core Graphics path. |
| `SKVideoNode` | A node that plays video content. |
| `SKEmitterNode` | A node that creates and renders particles. |
| `SKCropNode` | A node that crops its child nodes using a mask. |
| `SKEffectNode` | A node that applies a Core Image filter to its child nodes. |

Almost all of the techniques that work with sprite nodes can be applied to other node types as well. For example, you can use actions to animate other node objects onscreen, manipulate the order in which they are rendered, and use them inside the physics simulation. Read on to learn about how to use these other node classes in your game. After you become familiar with these classes, you will understand all of the visual capabilities of Sprite Kit. You can then start designing your game's appearance.

# Basic Nodes

The `SKNode` class doesn't draw any visual content. Its primary role is to provide baseline behavior that the other node classes use. However, this doesn't mean you can't find useful ways to use `SKNode` objects in your game. Here are some of the ways you might use basic nodes inside your game engine:

- You have content that is built up from multiple node objects, either sprites or other content nodes. However, you want this content to be thought of as a single object within your game, without promoting any one of the content nodes to be the root. A basic node is appropriate, because you can give it a position in the scene tree and then make all of the other nodes its descendants. These individual pieces can also be moved or adjusted relative to the parent's location.
- Use node objects to organize your drawn content into a series of layers. For example, many games have a background layer for the world, another layer for characters, and a third layer for text and other game information. Other games have many more layers. Create the layers as basic nodes, and insert them into the scene in order. Then, as necessary, you can make individual layers visible or invisible.
- You need an object in the scene that is invisible, but that performs some other necessary function. For example, in a dungeon exploring game, an invisible node might be used to represent a hidden trap. When another node intersects it, the trap is triggered. (See "Searching for Physics Bodies" (page 103).) Or for another example, you might add a node as a child of another node that represents the position of the player's point of view. See "Example: Centering the Scene on a Node" (page 83).

There are advantages to having a node in the tree to represent these concepts:

- You can add or remove entire subtrees by adding or removing a single node. This makes scene management efficient.
- You can adjust properties of a node in the tree and have the effects of those properties propagate down to the node's descendants. For example, if the basic node has sprite nodes as its children, rotating the basic node rotates all of the sprite content also.
- You can take advantage of actions, physics contacts, and other Sprite Kit features to implement the concept.

Subclassing the `SKNode` class is a useful way to build up more complex behaviors in your game. See "Use Subclassing to Create Your Own Node Behaviors" (page 110).

# Display Text with Label Nodes

Just about every game needs to display text at some point, even if it is just to display "Game Over" to the player. If you had to implement this yourself in OpenGL, it takes a fair amount of work to get it correct. But Sprite Kit makes it easy! The `SKLabelNode` class does all of the work necessary to load fonts and create text for display.

Listing 6-1 demonstrates how to create a new text label.

**Listing 6-1** Adding a text label

```
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];
```

Whenever you change the label node's properties, the label node is automatically updated the next time the scene is rendered.

## Shape Nodes Draw Path-Based Shapes

The `SKShapeNode` class draws a standard Core Graphics path. The graphics path is a collection of straight lines and curves that can define either open or closed subpaths. The shape node includes separate properties to specify the color of the lines and the color to fill the interior.

Shape nodes are useful for content that cannot be easily decomposed into textured sprites. Shape nodes are also very useful for building and displaying debugging information on top of your game content. However, textured sprites offer higher performance than shape nodes, so use shape nodes sparingly.

Listing 6-2 shows an example of how to create a shape node. The example creates a circle with a blue interior and a white outline. The path is created and attached to the shape node's `path` property.

**Listing 6-2** Creating a shape node from a path

```
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;
```

You can see from the code that the shape has three essential elements:

- The interior of the shape is filled. The `fillColor` property specifies the color used to fill the interior.
- The outline of the shape is rendered as a line. The `strokeColor` and `lineWidth` properties define how the line is stroked.
- A glow extends from the outline. The `glowWidth` and `strokeColor` properties define the glow.

You can disable any of these elements by setting its color to `[SKColor clearColor]`.

The shape node provides properties that let you control how the shape is blended into the framebuffer. You use these properties the same way as the properties of the `SKSpriteNode` class. See "Blending the Sprite into the Framebuffer" (page 35).

## A Video Node Plays a Movie

The `SKVideoNode` class uses the AV Foundation framework to display movie content. Like any other node, you can put the movie node anywhere inside the node tree and Sprite Kit will render it properly. For example, you might use a video node to animate some visual behaviors that would be expensive to define using actions.

A video node is similar to a sprite node, but offers only a subset of the features:

- The `size` property is initialized to the base size of the video content, but you can change it if you want. The video content is automatically stretched to the new size.
- The `anchorPoint` property defines where the content is displayed relative to the node position.

However, the following limitations apply:

- A video node is always stretched uniformly.
- A video node cannot be colorized.
- A video node always uses an alpha blend mode.

Like most of the node classes, creating a video node is very simple. Listing 6-3 shows a typical usage pattern. It initializes the video node using a video stored in the app bundle and then adds the node to the scene. It calls the node's `play` method to start the video playback.

**Listing 6-3** Displaying a video in a scene

```
SKVideoNode *sample = [SKVideoNode videoNodeWithVideoFileNamed:@"sample.m4v"];
sample.position = CGPointMake(CGRectGetMidX(self.frame),
                              CGRectGetMidY(self.frame));
[self addChild: sample];
[sample play];
```

You control playback using the node's `play` and `pause` methods.

If you need more precise control over the video playback behavior, you can use AV Foundation to create an `AVPlayer` object from your video content, and then use this object to initialize the video node. Then, instead of using the node's playback methods, you use the `AVPlayer` object to control playback. The video content is automatically displayed in the video node. For more information, see *AV Foundation Programming Guide*.

## Emitter Nodes Create Particle Effects

When an `SKEmitterNode` object is placed in a scene, it automatically creates and renders new particles. You use emitter nodes to automatically create special effects, including rain, explosions, or fire.

A **particle** is similar to an `SKSpriteNode` object; it renders a textured or untextured image that is sized, colorized, and blended into the scene. However, particles differ from sprites in two important ways:

- A particle's texture is always stretched uniformly.
- Particles are not represented by objects in Sprite Kit. This means you cannot perform node-related tasks on particles, nor can you associate physics bodies with particles to make them interact with other content.

Particles are purely visual objects, and their behavior is entirely defined by the emitter node that created them. The emitter node contains many properties to control the behavior of the particles it spawns, including:

- The birth rate and lifetime the particle. You can also specify the maximum number of particles that are spawned before the emitter turns itself off.
- The starting values of the particle, including its position, orientation, color, and size. These starting values are typically randomized.
- The changes to apply to the particle over its lifetime. Typically, these are specified as a rate-of-change over time. For example, you might specify that a particle rotates at a particular rate, in radians per second. The emitter automatically updates the particle data each frame. In most cases, you can also create more sophisticated behaviors using keyframe sequences. For example, you might specify a keyframe sequence for a particle so that it starts out small, scales up to a larger size, then shrinks before dying.

## Use the Particle Emitter Editor to Experiment with Emitters

In most cases, you never need to configure an emitter node directly in your game. Instead, you use Xcode to configure an emitter node's properties. As you change the behavior of the emitter node, Xcode immediately provides you an updated visual effect. When complete, Xcode archives the configured emitter. Then, at runtime, your game uses this archive to instantiate a new emitter node.

Using Xcode to create your emitter nodes has a few important advantages:

- It is the best way to learn the capabilities of the emitter class.
- You can more quickly experiment with new particle effects and see the results immediately.
- You separate the task of designing a particle effect from the programming task of using it. Your artists can work on new particle effects independent of your game code.

For more information on using Xcode to create particle effects, see *Particle Emitter Editor Guide*.

Listing 6-4 shows how to load a particle effect that was created by Xcode. All particle effects are saved using Cocoa's standard archiving mechanisms, so the code first creates a path to the smoke effect, and then loads the archive.

**Listing 6-4** Loading a particle effect from a file

```
- (SKEmitterNode *) newSmokeEmitter
{
    NSString *smokePath = [[NSBundle mainBundle] pathForResource:@"smoke"
ofType:@"sks"];
    SKEmitterNode *smoke = [NSKeyedUnarchiver unarchiveObjectWithFile:smokePath];
    return smoke;
}
```

## Manually Configuring Particle Creation

The SKEmitterNode class provides many properties for configuring an emitter node's behavior. The Xcode inspector sets the same property values. You can also create and configure your own emitter, or you can take an emitter node created in the Particle Emitter Editor, load it, and change its property values. For example, assume for a moment that you are using the smoke effect in Listing 6-4 (page 73) to show damage to a rocket ship. As the ship takes more damage, you could increase the birth rate of the emitter to add more smoke.

When the emitter node is in a scene, it emits new particles. You use the following properties to define how many particles it creates:

- The `particleBirthRate` property specifies the number of particles that the emitter creates every second.

- The `numParticlesToEmit` property specifies how many particles are created before the emitter turns itself off. You can also configure the node to emit an unlimited number of particles.

When a particle is created, its initial property values are determined by the properties of the emitter. For each of the particle's properties, the emitter class declares up to four properties:

- The average starting value for the property.
- A random range for values of the property. Each time a new particle is emitted, a new random value is calculated within that range.
- The rate at which the value changes over time, also known as the property's **speed**. Not all properties have a speed property.
- An optional keyframe sequence.

The complete list of properties used to configure an emitter node is described in *SKEmitterNode Class Reference* .

Listing 6-6 shows how you might configure an emitter's `scale` property. This is a simplified version of a node's `xScale` and `yScale` properties, and determines how large the particle is.

**Listing 6-5** Configuring a particle's scale properties

```
myEmitter.particleScale = 0.3;
myEmitter.particleScaleRange = 0.2;
myEmitter.particleScaleSpeed = -0.1;
```

When a new particle is created, its scale value is a random number from `0.2` to `0.4`. The scale value then decreases at a rate of `0.1` per second. So, if a particular particle started at the average value, `0.3`, it would decrease from `0.3` to `0` over a period of `3` seconds.

## Using Keyframe Sequences to Configure Custom Ramps for a Particle Property

Keyframe sequences provide more sophisticated behaviors for a particle property. A keyframe sequence specifies multiple points in a particle's lifetime and specifies the value for a property at each point. The keyframe sequence then interpolates values between those points and uses them to simulate the particle's property value.

You can use keyframe sequences to implement many custom behaviors, including:

- Changing a property value until it reaches a specific value.

- Using multiple different property values over the lifetime of a particle. For example, you might increase the value of the property in one part of the sequence and decrease it in another. Or, when specifying colors, you might specify multiple colors that the particle cycles through during its lifetime.
- Changing a property value using a nonlinear curve or a stepping function.

Listing 6-6 shows how you might replace the code in Listing 6-5 (page 74) to use a sequence. When you use a sequence, the values are not randomized. Instead, the sequence specifies all of the values of the property. Each keyframe value includes a value object and a timestamp. The timestamps are specified in a range from `0` to `1.0`, where `0` represents the birth of the particle and `1.0` represents its death. So, for this sequence, the particle starts out with a scale of `0.2` and increases to `0.7` one quarter of the way through the sequence. Three quarters of the way through the sequence, it reaches its minimum size, `0.1`. It remains at this size until it dies.

**Listing 6-6** Using a sequence to change a particle's scale property

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

myEmitter.particleScaleSequence = scaleSequence;
```

## Adding Actions to Particles

Although you do not have direct access to the particles created by Sprite Kit, you can specify an action that all particles execute. Whenever a new particle is created, the emitter tells the particle to run that action. You can use actions to create very sophisticated behaviors.

For the purpose of using actions on particles, you can treat the particle as if it were a sprite. This means you can perform other interesting tricks, such as animating the particle's textures.

## Using Target Nodes to Change the Destination of Particles

When the emitter creates particles, they are rendered as children of the emitter node. This means that they inherit the characteristics of the emitter node, just like nodes do. For example, if you rotate the emitter node, the positions of all of the spawned particles are rotated also. Depending on what effect you are simulating with the emitter, this may not be the correct behavior. For example, assume that you are using the emitter node to create the exhaust from a rocket. When the engines are at full burn, a cone of flame should come out the back of the ship. This is easily simulated using particles. But if the particles are rendered relative to the ship, when the ship turns, the exhaust is going to rotate as well. That doesn't look right. What you really want is for the particles to be spawned, but thereafter be independent of the emitter node. When the emitter node is rotated, new particles get the new orientation, and old particles maintain their old orientation. You make particles independent of the emitter by specifying a target node.

Listing 6-7 shows how to configure a rocket exhaust effect to use a target node. When the custom sprite node class instantiates the exhaust node, it makes the node its child. However, it redirects the particles to the scene.

**Listing 6-7** Using a target node to redirect where particles are spawned

```
- (void) newExhaustNode
{
    SKEmitterNode *emitter =  [NSKeyedUnarchiver unarchiveObjectWithFile:[[NSBundle
 mainBundle] pathForResource:@"exhaust" ofType:@"sks"]];

// Place the emitter at the rear of the ship.
   emitter.position = CGPointMake(0,-40);
   emitter.name = @"exhaust";
// Send the particles to the scene.
   emitter.targetNode = self.scene;

   [self addChild:emitter];
}
```

When an emitter has a target node, it calculates the position, velocity, and orientation of the particle, exactly as if it were a child of the sprite node. This means that if the ship sprite is rotated, the exhaust orientation is automatically rotated also. However, at the moment a new particle's starting values are calculated, the values are transformed into the target node's coordinate system. Thereafter, they would only be affected by changes to the target node.

## Particle Emitter Tips

Particle emitters in Sprite Kit are one of the most powerful tools for building visual effects. However, used incorrectly, particle emitters can be a bottleneck in the design and implementation of your app. Consider the following tips:

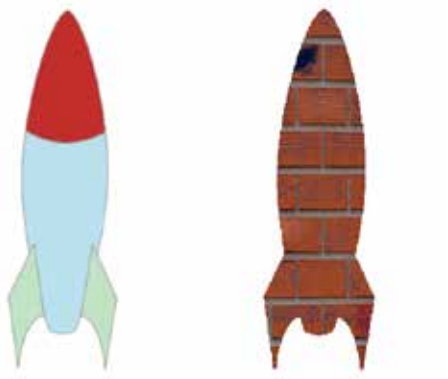
- Use Xcode to create and test your particle effects, then load the archives in your game.
- Adjust emitter properties sparingly inside your game code. Typically, you do this to specify properties that cannot be specified in the Xcode inspector or to control properties inside your game logic.

- Particles are cheaper than a sprite node, but they still have overhead! Try to keep the number of particles onscreen to a minimum by creating particle emitters with a low birth rate, and specifying a short lifetime for particles. For example, instead of creating hundreds or thousands of particles per second, reduce the birth rate and increase the size of the particles slightly. Often, you can create effects with fewer particles but the same net visual appearance.
- Use actions on particles only when there isn't another solution. Executing actions on individual particles is potentially very expensive, especially if the particle emitter also has a high birth rate.
- Assign a target node whenever the particles should be independent of the emitter node after they are spawned. For examples, particles should be independent if the emitter node moves or rotates in the scene.
- Consider removing a particle emitter from the scene when it is not visible onscreen. Add it just before it becomes visible.

## Crop Nodes Mask Portions of the Scene

An `SKCropNode` object does not directly render content, like a sprite node. Instead, it alters the behavior of its children when they are rendered. A crop node crops out portions of the content rendered by the children. This makes a crop node useful for implementing cockpit views, controls, and other game indicators, as well as any effect where the children should not draw outside of a specific region of the scene. Figure 6-1 uses the rocket ship art as a mask for a child sprite drawn by the crop node.

**Figure 6-1** A crop node performs a masking operation



The cropped area is specified using a mask. The mask is not a fixed image. It is rendered from a node, just like any other content in Sprite Kit. This means a crop node can create simple masks, but it can also implement more sophisticated behaviors. For example, here are some ways you might specify a mask:

- An untextured sprite creates a mask that limits content to a rectangular portion of the scene.

- A textured sprite is a precise per-pixel mask. But consider also the benefits of a nonuniformly scaled texture. You could use a nonuniformly scaled texture to create a mask for a resizable user-interface element (such as a health bar) and then fill the masked area with dynamic content.
- A collection of nodes can dynamically generate a complex mask that changes each time the frame is rendered.

Listing 6-8 shows a simple use of a mask. This code loads a mask image from a texture in the app bundle. A portion of the scene's content is then rendered, using the mask to prevent it from overdrawing the portion of the screen that the game uses to show controls.

**Listing 6-8** Creating a crop node

```
SKCropNode *cropNode = [[SKCropNode alloc] init];
myCropNode.position = CGPointMake(CGRectGetMidX(self.bounds),
                                  CGRectGetMidY(self.bounds));

cropNode.maskNode = [[SKSpriteNode alloc] initWithImageNamed:@"cockpitMask"];
[cropNode addChild: gamePlayNode];
[self addChild:cropNode];
[self addChild:gameControlNodes];
```

When the crop node is rendered, the mask is rendered before the descendants are drawn. Only the alpha component of the resulting mask is relevant. Any pixel in the mask with an alpha value of `0.05` or higher is rendered. All other pixels are cropped.

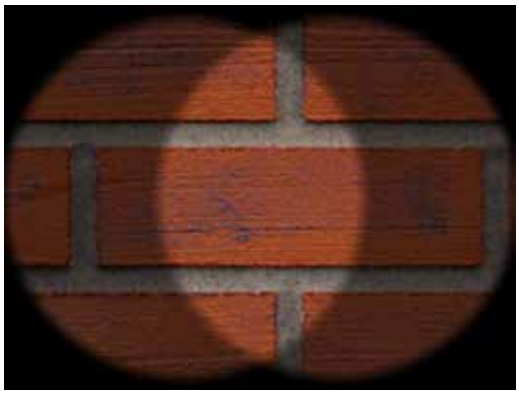
## Effect Nodes Apply Special Effects to Their Descendants

An `SKEffectNode` object does not draw content of its own. Instead, each time a new frame is rendered using the effect node, the effect node follows these steps:

1. Draws its children into a private framebuffer.
2. Applies a Core Image effect to the private framebuffer. This stage is optional.
3. Blends the contents of its private framebuffer into its parent's framebuffer, using one of the standard sprite blend modes.
4. Discards its private framebuffer.

Figure 6-2 shows one possible use for an effect node. In this example, the effect node's children are two sprites that act as light nodes. The effect node accumulates the effects of these lights, applies a blur filter to soften the resulting image, and then uses a multiply blend mode to apply this lighting to a wall texture.

**Figure 6-2** Effect nodes apply special effects to a node's children



Here's how the scene generates the lighting effect:

1. The scene has two children. The first is a textured sprite that represents the ground. The second is an effect node to apply lighting.

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

2. The effect node's children are sprite nodes rendered using an additive blend mode.

```
SKSpriteNode *light = [SKSpriteNode spriteNodeWithTexture:lightTexture];
light.blendMode = SKBlendModeAdd;
...
[self.lightingNode addChild: light];
```

3. The effect node includes a filter effect to soften the lighting.

```
- (CIFilter *)blurFilter
{
    CIFilter *filter = [CIFilter filterWithName:@"CIBoxBlur"]; // 3
    [filter setDefaults];
    [filter setValue:[NSNumber numberWithFloat:20] forKey:@"inputRadius"];

    return filter;
}
```

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

If you specify a Core Image filter, it must be a filter that takes a single input image and produces a single output image.

4. The effect node uses a multiplication blend mode to apply its lighting effect to the scene's framebuffer.

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

## Scenes Are Effect Nodes

You've learned a lot about the `SKScene` class already, but you may not have noticed that it is a subclass of `SKEffectNode`. This means that any scene can apply a filter to the contents. Although applying filters can be very expensive—not all filters are well designed for interactive effects—experimentation can help you find some interesting ways to use filters.

## Caching May Improve Performance of Static Content

An effect node normally discards its private framebuffer after rendering is complete. Rendering the content is necessary because it typically changes every frame. However, the cost of recreating the content and applying a Core Image filter can be high. If the content is static, then this is unnecessary. It might make more sense to keep the rendered framebuffer instead of discarding it. If the content of the effect node is static, you can set the node's `shouldRasterize` property to `YES`. Setting this property causes the following changes in behavior:

- The framebuffer is not discarded at the end of rasterization. This also means that more memory is being used by the effect node, and rendering may take slightly longer.
- When a new frame is rendered, the framebuffer is rendered only if the content of the effect node's descendants have changed.
- Changing the Core Image filter's properties no longer causes the framebuffer to automatically be updated. You can force it to be updated by setting the `shouldRasterize` property to `NO`.

# Advanced Scene Processing

Working with Sprite Kit involves manipulating the contents of the scene tree to animate content onscreen. Normally, actions are at the heart of that system. However, by hooking into the scene processing directly, you can create other behaviors that cannot be done with actions alone. To do that, you need to learn:

- How a scene processes animations
- How to add your own behaviors during scene processing

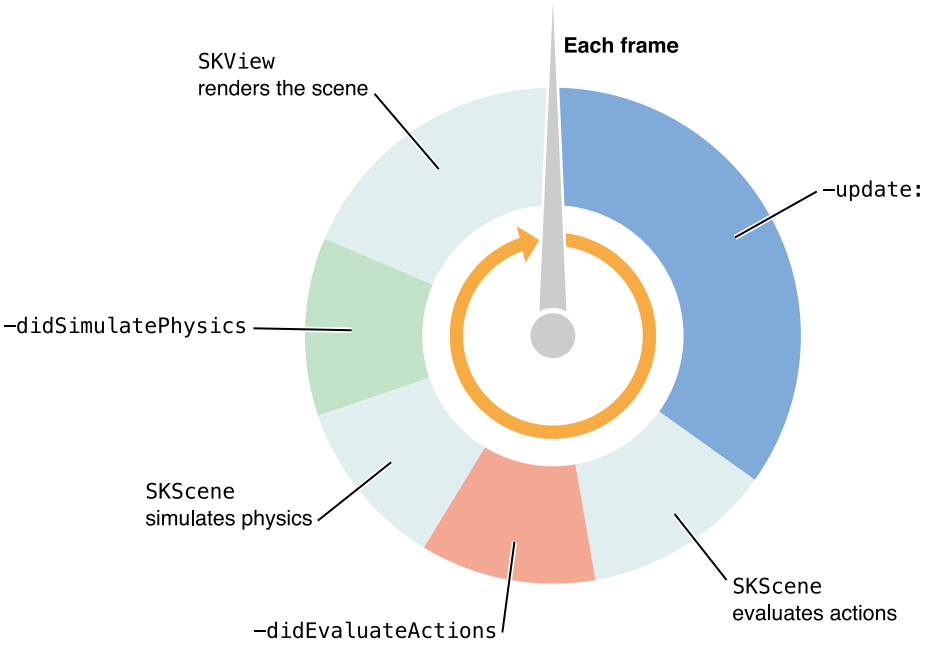
## How a Scene Processes Frames of Animation

In the traditional view system, the contents of a view are rendered once and then rendered again only when the model's contents change. This model works very well for views, because in practice most view content is static. Sprite Kit, on the other hand, is designed explicitly for dynamic content. Sprite Kit continuously updates the scene contents and renders it to ensure that animation is smooth and accurate.

The process of animating and rendering the scene is tied to the scene object (SKScene). Scene and action processing runs only when the scene is presented. A presented scene runs a rendering loop that alternates between processing the scene's node tree and rendering it. This model is similar to the rendering and processing loop used in most games.

Figure 7-1 shows the steps used by a scene to execute the rendering loop.

**Figure 7-1** Frame processing in a scene




Each time through the rendering loop, the scene's contents are updated and then rendered. You can't override the rendering behavior; instead you update the nodes in the scene. However, the scene includes methods you can override to customize scene processing, and you can use actions and physics to alter properties of nodes in the tree. Here are the steps in the rendering loop:

1. The scene's `update:` method is called with the time elapsed so far in the simulation.

   This is the primary place to implement your own in-game simulation, including input handling, artificial intelligence, game scripting, and other similar game logic. Often, you use this method to make changes to nodes or to run actions on nodes.

2. The scene processes actions on all the nodes in the tree. It finds any running actions and applies those changes to the tree. In practice, because of custom actions, you can also hook into the action mechanism to call your own code.

   You cannot directly control the order in which actions are processed or cause the scene to skip actions on certain nodes, except by removing the actions from those nodes or removing the nodes from the tree.

3. The scene's `didEvaluateActions` method is called after all actions for the frame have been processed.

4. The scene simulates physics on nodes in the tree that have physics bodies.

Adding physics to nodes in a scene is described in "Simulating Physics" (page 88), but the end result of simulating physics is that the position and rotation of notes in the tree may be adjusted by the physics simulation. Your game can also receive callbacks when physics bodies come into contact with each other.

5. The scene's `didSimulatePhysics` method is called after all physics for the frame has been simulated.

   This is your last chance to make changes to the scene.

6. The scene is rendered.

# Post-Processing in Scenes

A scene can process the actions on the scene tree in any order. For this reason, if you have tasks that you need to run each frame and you need precise control over when they run, you should use the `didEvaluateActions` and `didSimulatePhysics` methods to perform those tasks. Typically, you make changes during post-processing that require the final calculated positions of certain nodes in the tree. Your post-processing can take these positions and perform other useful work on the tree.

Here are some examples of post-processing tasks you might perform:

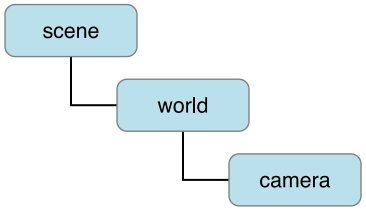
- Centering the scene's content on a particular node.
- Adding debugging information as an overlay over the scene content.
- Copying nodes from one part of the tree to another. For example, you might have an effect node whose children need to be the same as other nodes in the tree.

## Example: Centering the Scene on a Node

Centering a scene's content on a node is useful for implementing cameras and similar concepts in games that require content scrolling. In this case, the content is larger than the scene's frame. As the player moves around, the character stays fixed in place while the world moves around them. The scene remains locked to the character regardless of where the player moves the character.

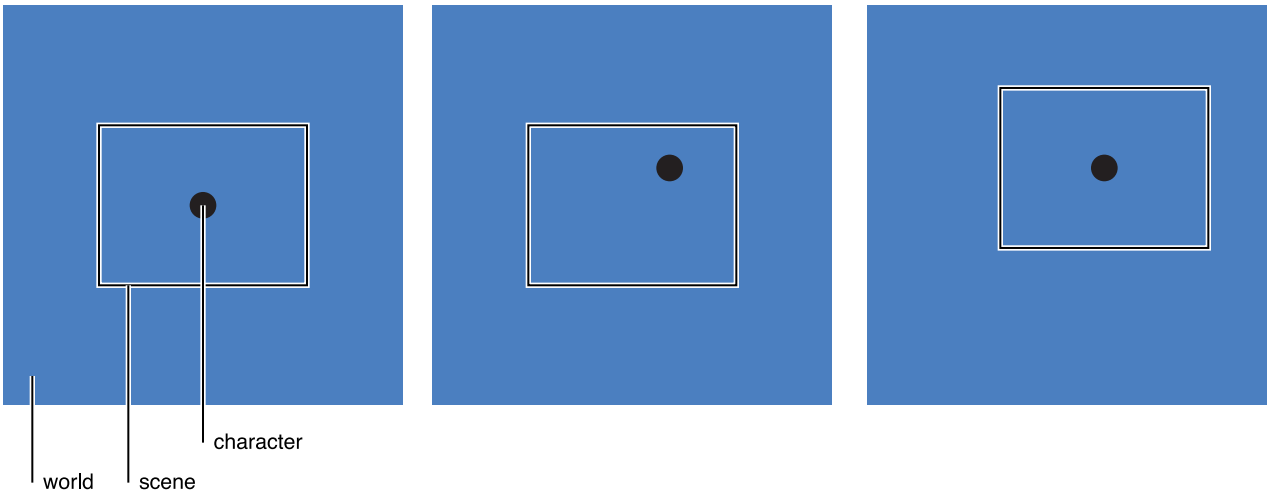
Sprite Kit does not provide built-in support for cameras, but the implementation is very straightforward. The world and the camera are each represented by a node in the scene. The world is a direct child of the scene and the camera is a descendant of the world node. This arrangement of nodes is useful because it gives the game world a coordinate system that is not tied to the scene's coordinate system. You can use this coordinate system to lay out the world content.

**Figure 7-2** Organizing the scene for a scrolling world




The camera is placed inside the world. The world can slide around in the scene. Because the camera is a node, you can use actions—and even physics—to move the camera. Then, in a post-processing step, you reposition the world node in the scene so that the camera node is centered on the scene, as shown in Figure 7-3. The world is placed in the scene so that the character is centered. Then, the character is moved around inside the world. Finally, a post-processing step repositions the world so that the character is centered again.

**Figure 7-3** The world is moved inside the scene




Here are the steps for this implementation:

1. Place the scene's anchor point at the center of the scene.

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

2. Use a world node to represent the scrolling world.

```
SKNode *myWorld = [SKNode node];
[self addChild:myWorld];
```

The children of the world node are sprite nodes (or other nodes) that represent the game's content. (Code not shown).

3. Use a node inside the world to represent the camera.

```
SKNode *camera = [SKNode node];
camera.name = @"camera";
[myWorld addChild:camera];
```

4. Use the `didSimulatePhysics` method to center the scene on the camera.

```
- (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 -
cameraPositionInScene.x,
node.parent.position.y - cameraPositionInScene.y);
}
```

The `centerOnNode:` method converts the camera's current position into scene coordinates, then subtracts those coordinates from the world's position to slide the character to the `(0,0)` position.

## Example: Adding a Debugging Overlay

When you are developing a game using Sprite Kit, it is helpful to show live visual debugging information about what's happening in the scene. For example, any of the following information might be useful when debugging your game:

- Artificial intelligence decisions by characters in the scene
- Locations in the world that trigger scripted actions
- Physics information such as gravity, other forces, or even the size of physics bodies

Shape and label nodes are particularly useful for annotating the behavior of your game.

To add a debug overlay, the best approach is to use a single node to represent the overlay and add it to the scene. All of the debugging information is represented by nodes that are descendants of this node. You place this node in the scene with a z coordinate that places it above all other scene content. The debug node may be, but does not have to be, a direct child of the scene. For example, in a scrolling world, the information you are placing in the overlay might be tied to world coordinates instead of scene coordinates.

Before the scene starts processing a frame, you remove this node from the scene and then remove its children. The assumption is that the debugging information needs to be updated every frame. After the scene simulates physics, the node is added back into the scene and its contents are regenerated.

1. Create a property for the debugging node in the scene class and initialize it when the scene is first presented.

```
@property (SKNode *) debugOverlay;

self.debugOverlay = [SKNode node];
[self addChild:self.debugOverlay];
```

2. Remove the node before actions are processed.

```
- (void)update:(NSTimeInterval)currentTime
{
    [self.debugOverlay removeFromParent];
    [self.debugOverlay removeAllChildren];
}
```

3. Add the node after the scene is processed.

```
- (void)didSimulatePhysics
{
    [self addChild:self.debugOverlay];
    // add code to create and add debugging images to the debug node.
```

```
    // This example displays a gravity vector.
    SKShapeNode *gravityLine = [[SKShapeNode alloc] init];
    gravityLine.position = CGPointMake (200,200);

    CGMutablePathRef path = CGPathCreateMutable();
    CGPathMoveToPoint(path, NULL, 0.0, 0.0);
    CGPathAddLineToPoint(path, 0, self.physicsWorld.gravity.x*10,
self.physicsWorld.gravity.y*10);
    CGPathCloseSubpath(path);
    gravityLine.path = path;
    CGPathRelease(path);

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

## Example: Replicating Information in the Scene

Replicating information in the tree is similar to the techniques used to add a debugging overlay. When you pre-process the scene, you remove the stale copies of the nodes from the tree. Then, during post-processing, you copy the nodes from one part of the tree to another. You use copies of the nodes because each node can have only one parent.

In some cases, you only need to update a small amount of information in each frame. In this case, constantly adding, removing, and copying nodes could be costly. Instead, make the copies once, and then use your post-processing step to update the important properties.

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

# Simulating Physics

The physics simulation in Sprite Kit is performed by adding physics bodies to scenes. A **physics body** is a simulated physical object connected to a node in the scene's node tree. It uses the node's position and orientation to place itself in the simulation. Every physics body has other characteristics that define how the simulation operates on it. These include innate properties of the physical object, such as its mass or density, and also properties imposed on it, such as its velocity. These characteristics define how a body moves, how it is affected by forces in the simulation, and how it responds to collisions with other physics bodies.

Each time a scene computes a new frame of animation, it simulates the effects of forces and collisions on physics bodies connected to the node tree. It computes a final position, orientation, and velocity for each physics body. Then, the scene updates the position and rotation of each corresponding node.

To use physics in your game, you need to:

- Attach physics bodies to nodes in the node tree. See "All Physics is Simulated on Physics Bodies" (page 89).
- Configure the physical properties of the physics bodies. See "Configuring the Physical Properties of a Physics Body" (page 92).
- Define global characteristics of the scene's physics simulation, such as gravity. See "Configuring the Physics World" (page 93).
- Where necessary to support your gameplay, set the velocity of physics bodies in the scene or apply forces or impulses to them. See "Making Physics Bodies Move" (page 94).
- Define how the physics bodies in the scene interact when they come in contact with each other. See "Working with Collisions and Contacts" (page 96).
- Optimize your physics simulation to limit the number of calculations it must perform. See "Tips and Tricks for Using Physics in Your Game" (page 104).

Sprite Kit uses the International System of Units, also known as SI, or the meter-kilogram-second system. Where necessary, you may need to consult other reference materials online to learn more about the physics equations used by Sprite Kit.

# All Physics is Simulated on Physics Bodies

An `SKPhysicsBody` object defines the shape and simulation parameters for a physics body in the system. When the scene simulates physics, it performs the calculations for all physics bodies connected to the scene tree. So, you create an `SKPhysicsBody` object, configure its properties, and then assign it to a node's `physicsBody` property.

There are three kinds of physics bodies:

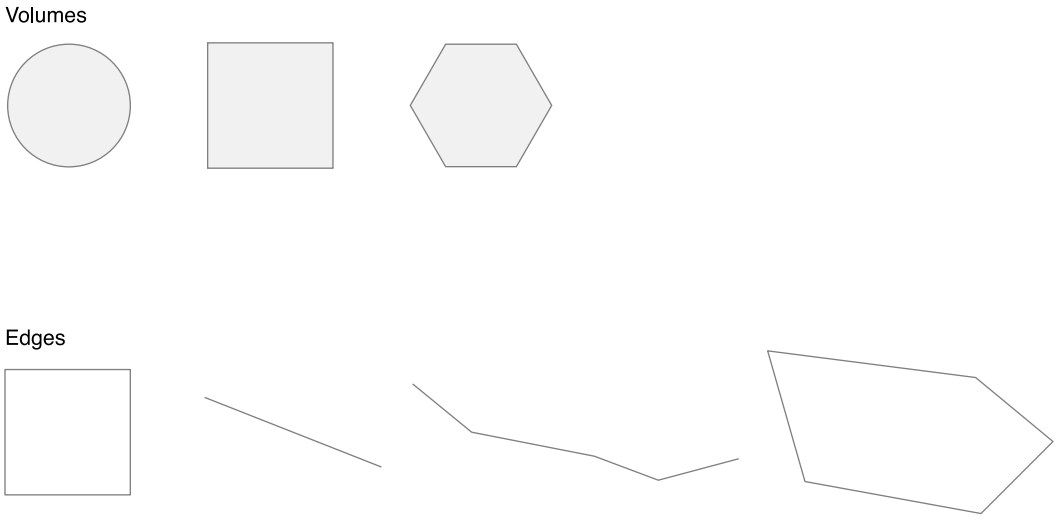
- A **dynamic volume** simulates a physical object with volume and mass that can be affected by forces and collisions in the system. Use dynamic volumes to represent items in the scene that need to move around and collide with each other.
- A **static volume** is similar to a dynamic volume, but its velocity is ignored and it is unaffected by forces or collisions. However, because it still has volume, other objects can bounce off it or interact with it. Use static volumes to represent items that take up space in the scene, but that should not be moved by the simulation. For example, you might use static volumes to represent the walls of a maze.

  While it is useful to think of static and dynamic volumes as distinct entities, in practice these are two different modes you can apply to any volume-based physics body. This can be useful because you can selectively enable or disable effects for a body.
- An **edge** is a static volume-less body. Edges are never moved by the simulation and their mass doesn't matter. Edges are used to represent negative space within a scene (such as a hollow spot inside another entity) or an uncrossable, invisibly thin boundary. For example, edges are frequently used to represent the boundaries of your scene.

  The main difference between a edge and a volume is that an edge permits movement inside its own boundaries, while a volume is considered a solid object. If edges are moved through other means, they only interact with volumes, not with other edges.

Sprite Kit provides a few standard shapes, as well as shapes based on arbitrary paths. Figure 8-1 shows the shapes available.

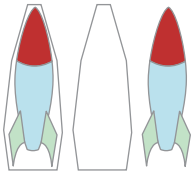
**Figure 8-1** Physics bodies




## Use a Physics Shape That Matches the Graphical Representation

In most cases, a physics body should have a size and shape that closely approximates the visual representation of the corresponding node. For example, in Figure 8-2, the rocket has a narrow shape that is not well represented by either a circle or a rectangle. A convex polygon shape is chosen and fitted to match the sprite's artwork.

**Figure 8-2** Match the shape with a close representation



However, when choosing a shape for your physics body, do not be overly precise. More complex shapes require more work to be properly simulated. For volume-based bodies, use the following guidelines:

- A circle is the most efficient shape.
- A path-based polygon is the least efficient shape, and the computational work scales with the complexity of the polygon.

An edge-based body is more expensive to compute than a volume-based body. This is because the bodies it interacts with can potentially be on either side of an open edge or on the inside or outside of a closed shape. Use these guidelines:

- Lines and rectangles are the most efficient edge-based bodies.
- Edge loops and edge chains are the most expensive edge-based bodies, and the computational work scales with the complexity of the path.

## Creating Physics Bodies

A physics body is created by calling one of the `SKPhysicsBody` class methods. Each class method defines whether a volume-based or edge-based body is being created and what shape it has.

### Creating an Edge Loop Around the Scene

Listing 8-1 shows code that is used frequently in games that do not need to scroll the content. In this case, the game wants physics bodies that hit the borders of the scene to bounce back into the gameplay area.

**Listing 8-1** A scene border

```
- (void) createSceneContents
{
    self.backgroundColor = [SKColor blackColor];
    self.scaleMode = SKSceneScaleModeAspectFit;
    self.physicsBody = [SKPhysicsBody bodyWithEdgeLoopFromRect:self.frame];
}
```

### Creating a Circular Volume for a Sprite

Listing 8-2 shows the code that creates the physics body for a spherical or circular object. Because the physics body is attached to a sprite object, it usually needs volume. In this case, the sprite image is assumed to closely approximate a circle centered on the anchor point, so the radius of the circle is calculated and used to create the physics body.

**Listing 8-2** A physics body for a circular sprite

```
SKSpriteNode *sprite = [SKSpriteNode spriteNodeWithImageNamed:@"sphere.png"];
sprite.physicsBody = [SKPhysicsBody bodyWithCircleOfRadius:sprite.size.width/2];
sprite.physicsBody.dynamic = YES;
```

If the physics body were significantly smaller than the sprite's image, the data used to create the physics body might need to be provided by some other source, such as a property list. See "Sprite Kit Best Practices" (page 107).

# Configuring the Physical Properties of a Physics Body

The `SKPhysicsBody` class defines properties that determine how the physics body is simulated. These properties affect how the body reacts to forces, what forces it generates on itself (to simulate friction), and how it reacts to collisions in the scene. In most cases, the properties are used to simulate physical effects.

Each individual body also has its own property values that determine exactly how it reacts to forces and collisions in the scene. Here are the most important properties:

- The `mass` property determines how forces affect the body, as well as how much momentum the body has when it is involved in a collision.
- The `friction` property determines the roughness of the body's surface. It is used to calculate the frictional force that a body applies to other bodies moving along its surface.
- The `linearDamping` and `angularDamping` properties are used to calculate friction on the body as it moves through the world. For example, this might be used to simulate air or water friction.
- The `restitution` property determines how much energy a body maintains during a collision—its bounciness.

Other properties are used to determine how the simulation is performed on the body itself:

- The `dynamic` property determines whether the body is simulated by the physics subsystem.
- The `affectedByGravity` property determines whether the simulation exerts a gravitational force on the body. For more information on the physics world, see "Configuring the Physics World" (page 93).
- The `allowsRotation` property determines whether forces can impart angular velocity on the body.

## Mass Determines a Body's Resistance to Acceleration

You should set the mass on every volume-based body in your scene so that it properly reacts to forces applied to it.

A physics body's `mass`, `area`, and `density` properties are all interrelated. When you first create a body, the body's area is calculated, and never changes afterwards. The other two properties change values at the same time, based on the following formula:

```
mass = density x area
```

When you configure a physics body, you have two options:

- Set the `mass` property of the body. The `density` property is then automatically recalculated. This approach is most useful when you want to precisely control each body's mass.
- Set the `density` property of the body. The `mass` property is then automatically recalculated. This approach is most useful when you have a collection of similar bodies created with different sizes. For example, if your physics bodies were used to simulate asteroids, you might give all asteroids the same density, and then set an appropriate bounding polygon for each. Each body automatically computes an appropriate mass based on its size on the screen.

### When to Adjust a Body's Properties

Most often, you configure a physics body once and then never change it. For example, the mass of a body is unlikely to change during play. However, you are not restricted from doing so. Some kinds of games may require the ability to adjust a body's properties even while the simulation is executing. Here are a few examples of when you might do so:

- In a realistic rocket simulation, the rocket expends fuel to apply thrust. As fuel is used up, the mass of the rocket changes. To implement this in Sprite Kit, you might create a rocket class that includes a `fuel` property. When the rocket thrusts, the fuel is reduced and the corresponding body's mass is recalculated.
- The damping properties are usually based on the body's characteristics and the medium it is traveling through. For example, a vacuum applies no damping forces, and water applies more damping forces than air. If your game simulates multiple environments and bodies can move between those environments, your game can update a body's damping properties whenever it enters a new environment.

Typically, you make these changes as part of scene pre- and post-processing. See "Advanced Scene Processing" (page 81).

## Configuring the Physics World

All physics bodies in a scene are part of the physics world, which is represented in Sprite Kit by an `SKPhysicsWorld` object attached to the scene. The physics world defines two important characteristics of the simulation:

- The `gravity` property applies an acceleration to volume-based bodies in the simulation. Static volumes and physics bodies that have set the `affectedByGravity` property to `NO` are unaffected.
- The `speed` property determines the rate at which the simulation runs.

# Making Physics Bodies Move

By default, only gravity is applied to physics bodies in the scene. In some cases, that might be enough to build a game. But in most cases, you need to take other steps to change the speed of physics bodies.

First, you can control a physics body's velocity directly, by setting its `velocity` and `angularVelocity` properties. As with many other properties, you often set these properties once when the physics body is first created and then let the physics simulation adjust them as necessary. For example, assume for a moment you are making a space-based game where a rocket ship can fire missiles. When the ship fires a missile, the missile should have a starting velocity of the ship plus an additional vector in the direction of the launch. Listing 8-3 shows one implementation for calculating the launch velocity.

**Listing 8-3** Calculating the missile's initial velocity

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

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


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

When a body is in the simulation, it is more common for the velocity to be adjusted based on forces applied to the body. Another source of velocity changes, collisions, is discussed later.

The default collection of forces that apply to a body include:

- The gravitational force applied by the physics world
- The damping forces applied by the body's own properties
- A frictional force based on contact with another body in the system

You can also apply your own forces and impulses to physics bodies. Most often, you apply forces and impulses in a pre-processing step before the simulation executes. Your game logic is responsible for determining which forces need to be applied and for making the appropriate method calls to apply those forces.

You can choose to apply either a force or an impulse:

- A force is applied for a length of time based on the amount of simulation time that passes between when you apply the force and when the next frame of the simulation is processed. So, to apply a continuous force to an body, you need to make the appropriate method calls each time a new frame is processed. Forces are usually used for continuous effects
- An impulse makes an instantaneous change to the body's velocity that is independent of the amount of simulation time that has passed. Impulses are usually used for immediate changes to a body's velocity.

To continue with the rocket example, a rocket ship probably applies a force to itself when it turns on its engines. However, when it fires a missile, it might launch the missile with the rocket's own velocity and then apply a single impulse to it to give it the initial burst of speed.

Because forces and impulses are modeling the same concept—adjusting a body's velocity—the remainder of this section focuses on forces.

You can apply a force to a body in one of three ways:

- A linear force that only affects the body's linear velocity.
- An angular force that only affects the body's angular velocity.
- A force applied to a point on the body. The physics simulation calculates separate changes to the body's angular and linear velocity, based on the shape of the object and the point where the force was applied.

Listing 8-4 shows code you could implement in a sprite subclass to apply a force to the ship. This force accelerates the rocket when the main engines are activated. Because the engines are at the back of the rocket, the force is applied to linearly to the rocket body. The code calculates the thrust vector based on the current orientation of the rocket. The orientation is based on the `zRotation` property of the corresponding node, but the orientation of the artwork may differ from the orientation of the node. The thrust should always be oriented with the artwork. See "Drawing Your Content" (page 111).

**Listing 8-4** Applying rocket thrust

```
static const CGFloat thrust = 0.12;

CGFloat shipDirection = [self shipDirection];
CGVector thrustVector = CGVectorMake(thrust*cosf(shipDirection),
                                     thrust*sinf(shipDirection));
[self.physicsBody applyForce:thrustVector];
```

Listing 8-5 shows a similar effect, but this time the rocket is being rotated by the force, so the thrust is applied as an angular thrust.

**Listing 8-5** Applying lateral thrust

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

# Working with Collisions and Contacts

Sooner or later, two bodies are going to try to occupy the same space. It is up to you to decide how your game responds. Sprite Kit uses two kinds of interactions between physics bodies:

- A **contact** is used when you need to know that two bodies are touching each other. In most cases, you use contacts when you need to make gameplay changes when a collision occurs.
- A **collision** is used to prevent two objects from interpenetrating each other. When one body strikes another body, Sprite Kit automatically computes the results of the collision and applies impulse to the bodies in the collision.

Your game configures the physics bodies in the scene to determine when collisions should occur and when interactions between physics bodies require additional game logic to be performed. Limiting these interactions is not only important for defining your game's logic, it is also necessary in order to get good performance from Sprite Kit. Sprite Kit uses two mechanisms to limit the number of interactions in each frame:

- Edge-based physics bodies never interact with other edge-based bodies. This means that even if you move edge-based bodies by repositioning the nodes, the physics bodies never collide or contact each other.
- Every physics body is categorized. Categories are defined by your app; each scene can have up to 32 categories. When you configure a physics body, you define which categories it belongs to and which categories of bodies it wants to interact with. Contacts and collisions are specified separately.

## Collision and Contact Example: Rockets in Space

The contacts and collision system is easier to understand by exploring an example. In this example, the scene is used to implement a space game. Two rocket ships are dueling over a portion of outer space. This area of space has planets and asteroids that the ships can collide with. Finally, because this is a duel, both rocket ships are armed with missiles they can shoot at each other. This simple description defines the rough gameplay for the example. But implementing this example requires a more precise expression of what bodies are in the scene and how they interact with each other.

> **Note:** This example is also available as sample code in *SpriteKit Physics Collisions*.

From the description above, you can see that the game has four kinds of unit types that appear in the scene:

- Missiles
- Rocket ships
- Asteroids
- Planets

The small number of unit types suggests that a simple list of categories is a good design. Although the scene is limited to 32 categories, this design only needs four, one for each unit type. Each physics body belongs to one and only one category. So, a missile might appear as a sprite node. The missile's sprite node has an associated physics body, and that body belongs to the missile category. The other nodes and physics bodies are defined similarly.

**Pro Tip:** Consider encoding other game-related information in the remaining categories. Then, use these categories when implementing your gameplay logic. For example, any of the following attributes might be useful when implementing logic in a more advanced rockets game:

- Man-made object
- Natural object
- Explodes during collision

Given those four categories, the next step is to define the interactions that are permitted between these physics bodies. The contact interactions are usually important to tackle first, because they are almost always dictated by your gameplay logic. In many cases, if a contact is detected, you need a collision to be calculated also. This is most common when the contact results in one of the two physics bodies being removed from the scene.

Table 8-1 describes the contact interactions for the rockets game.

**Table 8-1** The contact grid for the rockets game

| | Missile | Rocket ship | Asteroid | Planet |
|---|---|---|---|---|
| **Missile** | No | Yes | Yes | Yes |
| **Rocket ship** | No | Yes | Yes | Yes |
| **Asteroid** | No | No | No | Yes |
| **Planet** | No | No | No | No |

All of these interactions are based on the game logic. That is, when any of these contacts occur, the game needs to be notified so that it can update the game state. Here's what needs to happen:

- A missile explodes when it strikes a ship, asteroid, or planet. If the missile strikes a ship, the ship takes damage.
- A ship that contacts a ship, asteroid, or planet takes damage.
- An asteroid that contacts a planet is destroyed.

It isn't necessary for these interactions to be symmetrical, because Sprite Kit only calls your delegate once per frame for each contact. Either body can specify that it is interested in the contact. Because a missile already requests a contact message when it strikes a ship, the ship does not need to ask for the same contact message.

The next step is determining when collisions should be calculated by the scene. Each body describes which kinds of bodies in the scene can collide with it. Table 8-2 describes the list of permitted collisions.

**Table 8-2** The collision grid for the rockets game

| | Missile | Rocket ship | Asteroid | Planet |
|---|---|---|---|---|
| **Missile** | No | No | No | No |
| **Rocket ship** | No | Yes | Yes | Yes |
| **Asteroid** | No | Yes | Yes | No |
| **Planet** | No | No | No | Yes |

When working with collisions, each physics body's collision information is important. When two bodies collide, it is possible for only one body to affected by the collision. When this occurs, only the affected body's velocity is updated.

In the table, the following assumptions are made:

- Missiles are always destroyed in any interactions with other objects, so a missile ignores all collisions with other bodies. Similarly, missiles are considered to have too little mass to move other bodies in a collision. Although the game could choose to have missiles collide with other missiles, the game chooses not to because there will be a lot of missiles flying around. Because every missile might need to be tested against every other missile, these interactions would require a lot of extra computations.
- A ship tracks collisions with ships, asteroids, and planets.
- An asteroid ignores planetary collisions because the gameplay description for contacts states that the asteroid is going to be destroyed.
- Planets track only collisions with other planets. Nothing else has enough mass to move the planet, so the game ignores those collisions and avoids potentially expensive calculations.

## Implementing the Rocket Example in Code

When you have determined your categorizations and interactions, you need to implement these in your game's code. The categorizations and interactions are each defined by a 32-bit mask. Whenever a potential interaction occurs, the category mask of each body is tested against the contact and collision masks of the other body. Sprite Kit performs these tests by logically ANDing the two masks together. If the result is a nonzero number, then that interaction occurs.

Here's how you would turn the rocket example into Sprite Kit code:

1. Define the category mask values:

**Listing 8-6** Category mask values for the space duel

```
static const uint32_t missileCategory     =  0x1 << 0;
static const uint32_t shipCategory        =  0x1 << 1;
static const uint32_t asteroidCategory    =  0x1 << 2;
static const uint32_t planetCategory      =  0x1 << 3;
```

2. When a physics body is initialized, set its `categoryBitMask`, `collisionBitMask`, and `contactTestBitMask` properties.

   Listing 8-7 shows a typical implementation for the rocket ship entry in Table 8-1 (page 97) and Table 8-2 (page 98).

**Listing 8-7** Assigning contact and collision masks to a rocket

```
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 |
planetCategory;
```

3. Assign a contact delegate to the scene's physics world.

   Often, the delegate protocol is implemented by the scene. This is the case in Listing 8-8.

**Listing 8-8** Adding the scene as the contact delegate

```
-(id)initWithSize:(CGSize)size
{
    if (self = [super initWithSize:size])
    {
...
       self.physicsWorld.gravity = CGVectorMake(0,0);
       self.physicsWorld.contactDelegate = self;
...
    }
    return self;
}
```

4. Implement the contact delegate method to add the game logic.

**Listing 8-9** A partial implementation of a contact delegate

```
- (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];
```

```
    }
...
}
```

This example shows a few important concepts to consider when implementing your contact delegate:

- The delegate is passed an `SKPhysicsContact` object that declares which bodies were involved in the collision.
- The bodies in the contact can appear in any order. In the Rockets code, the interaction grids are always specified in the sorted order of the categories. In this case, the code relies on this to sort the two entries when a contact occurs. A simpler alternative would be to check both bodies in the contact and dispatch if either matches the mask entry.
- When your game needs to work with contacts, you need to determine the outcome based on both bodies in the collision. Consider studying the Double Dispatch Pattern and using it to farm out the work to other objects in the system. Embedding all of the logic in the scene can result in long, complicated contact delegate methods that are difficult to read.
- The `node` property can access the node that a physics body is attached ti. Use it to access information about the nodes participating in the collision. For example, you might use the node's class, its `name` property, or data stored in its `userData` dictionary to determine how the contact should be processed.

## Specify High Precision Collisions for Small or Fast-Moving Objects

When Sprite Kit performs collision detection, it first determines the locations of all of the physics bodies in the scene. Then it determines whether collisions or contacts occurred. This computational method is fast, but can sometimes result in missed collisions. A small body might move so fast that it completely passes through another physics body without ever having a frame of animation where the two touch each other.

If you have physics bodies that must collide, you can hint to Sprite Kit to use a more precise collision model to check for interactions. This model is more expensive, so it should be used sparingly. When either body uses precise collisions, multiple movement positions are contacted and tested to ensure that all contacts are detected.

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

# Connecting Physics Bodies

Although you can make a lot of interesting games that use the physics systems that have already been described, you can take your designs further by connecting physics bodies using **joints**. When the scene simulates the physics, it takes these joints into account when calculating how forces affect the bodies.

**Figure 8-3** Joints connect nodes in different ways

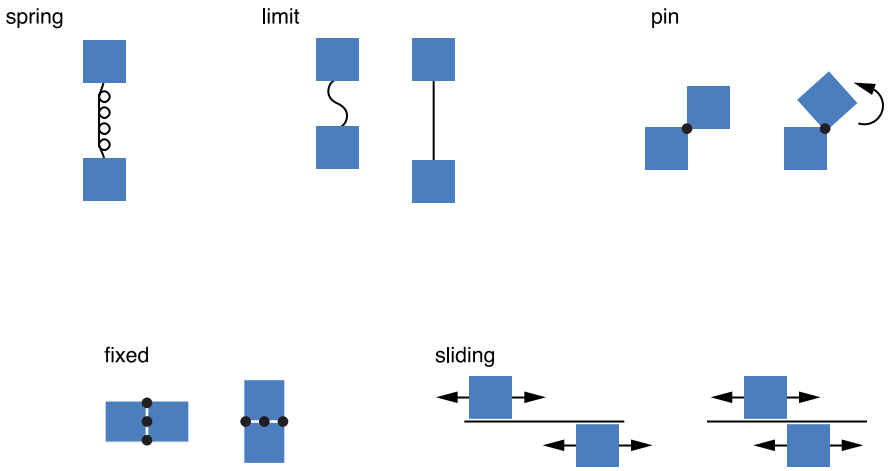



Table 8-3 describes the kinds of joints you can create in Sprite Kit.

**Table 8-3** Joint classes implemented in Sprite Kit

| Class | Description |
| --- | --- |
| `SKPhysicsJointFixed` | A fixed joint fuses two bodies together at a reference point. Fixed joints are useful for creating complex shapes that can be broken apart later. |
| `SKPhysicsJointSliding` | A sliding joint permits the anchor points of two bodies to slide along a chosen axis. |
| `SKPhysicsJointSpring` | A spring joint acts as a spring whose length is the initial distance between two bodies. |
| `SKPhysicsJointLimit` | A limit joint imposes a maximum distance between two bodies, as if they were connected by a rope. |
| `SKPhysicsJointPin` | A pin joint pins two bodies together. The bodies rotate independently around the anchor point. |

You add or remove joints using the physics world. When you create a joint, the points that connect the joint are always specified in the scene's coordinate system. This may require you to first convert from node coordinates to scene coordinates before creating a joint.

To use a physics joint in your game, follow these steps:

1. Create two physics bodies.
2. Attach the physics bodies to a pair of `SKNode` objects in the scene.
3. Create a joint object using one of the subclasses listed in Table 8-3 (page 102).
4. If necessary, configure the joint object's properties to define how the joint should operate.
5. Retrieve the scene's `SKPhysicsWorld` object.
6. Call the physics world's `addJoint:` method.

## Searching for Physics Bodies

Sometimes, it is necessary to find physics bodies in the scene. For example, you might need to:

- Discover whether a physics body is located in a region of the scene.
- Detect when a physics body (such as the one controlled by the player) crosses a particular line.
- Trace the line of sight between two physics bodies, to see whether another physics body, such as a wall, is interposed between the two objects.

In some cases, you can implement these interactions using the collisions and contacts system. For example, to discover when a physics body enters a region, you could create a physics body and attach it to an invisible node in the scene. Then, configure the physics body's collision mask so that it never collides with anything, and its contact mask to detect the physics bodies you are interested in. Your contact delegate is called when the desired interactions occur.

However, it's not easy to implement concepts such as line of sight using this design. To implement these, you use the scene's physics world. With the physics world, you can search for all physics bodies along a ray or physics bodies that intersect a particular point or rectangle.

An example illustrates the basic technique. Listing 8-10 shows one possible implementation of a line-of-sight detection system. It casts a ray from the origin of the scene in a particular direction, searching for the nearest physics body along the ray. If it finds a physics body, then it tests the category mask to see whether this is a target it should attack. If it sees a target designated for attack, it shoots the cannon.

**Listing 8-10** Casting a ray from the center of the scene

```
- (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];
    }
}
```

Another way to implement the same behavior is to set the starting and ending positions of the ray to those of two physics bodies in your scene. For example, you might use the location of the player's game object as one position and the position of an enemy unit as the other position.

You can also perform searches for physics bodies that intersect a point or rectangle using the `bodyAtPoint:` and `bodyInRect:` methods.

Sometimes you can't make a simple determination based on the closest physics body within the scene. For example, in the logic of your game, you might decide that not all physics bodies block the line of sight. In this case, you need to enumerate all of the physics bodies along the ray using the `enumerateBodiesAlongRayStart:end:usingBlock:` method. You supply a block that is called once for each body along the ray. You can then use this information to make a more informed decision about whether the line of sight exists to a target.

## Tips and Tricks for Using Physics in Your Game

Consider the following advice when building a physics-based game.

## Design Your Physics Bodies Systematically

Before you spend too much time adding physics to your scene, you should first understand what kinds of bodies you are going to include in the scene and how they interact with each other. Systematically go through this process for each body:

- Is it an edge, a static volume, or a dynamic volume? See "All Physics is Simulated on Physics Bodies" (page 89).
- What shape best approximates the body, keeping in mind that some shapes are computationally more expensive than others? See "Use a Physics Shape That Matches the Graphical Representation" (page 90)
- What kind of body is it and how does it interact with other bodies? See "Working with Collisions and Contacts" (page 96).
- Does this body move quickly, or is it very small? If so, determine whether high-precision collision detection is necessary. See "Specify High Precision Collisions for Small or Fast-Moving Objects" (page 101).

## Fudging the Numbers

While it is useful to know that Sprite Kit measures items in the International System of Units, worrying about precise numbers is not that important. It doesn't matter much whether your rocket ship weights 1 kilogram or `1,000,000` kilograms, as long as the mass is consistent with other physics values used in the game. Often, proportions are more important than the actual values being used.

Game design is usually an iterative process, as you tweak the numbers in the simulation. This sort of design often results in many hard-coded numbers in your game. Resist the urge to do this! Instead, implement these numbers as data that can be archived with the corresponding node objects or physics bodies. Your code should provide the behaviors, but the specific numbers used to implement those behaviors should be editable values that an artist or designer can tweak or test.

Most of the information stored in a physics body and corresponding node can be archived using the standard archiving mechanisms found in Cocoa. This suggests that your own tools may also be able to save and load these archives and use them as the preferred data format. This is usually possible, but keep in mind that the shape of a physics body is private data that cannot be determined from the object. This means if you do use archives as the primary format for saving data in your tools, you may also need to archive other information used to create the physics body. The general topic of developing tools using Sprite Kit is described in "Sprite Kit Best Practices" (page 107).

## Most Physics Properties Are Dynamic, so Adapt Them at Runtime

Very few characteristics of a physics body are fixed. Outside of a physics body's area, most properties can be changed at any time. Take advantage of this. Here are a few examples:

- You can lock a static volume in place until the player performs the task that unlocks it. Then, change the body to a dynamic volume.
- You can construct a box from multiple smaller physics bodies and hold it together using fixed joints. Create the pieces with a contact mask so that when they hit something, the contact delegate can break the joints.
- As an object moves throughout the scene, you adjust its linear and rotational damping based on the medium it is in. For example, when the object moves into water, you update the properties to match.

# Sprite Kit Best Practices

At this point, you already have a good idea of what Sprite Kit can do and how it works. You know how to add nodes to scenes and perform actions on those nodes, and these tasks are the building blocks for creating gameplay. What you may be missing is the bigger picture. That is, you need an understanding of how to plan and develop games and tools using Sprite Kit. To get the most out of Sprite Kit, you need to know:

- How to organize your game into scenes and transitions
- When to subclass Sprite Kit classes
- How to store your game's data and art
- How to use build your own tools to create Sprite Kit content and export that content for your game to use

Sprite Kit provides more than just a graphics layer for your game; it also provides features that make it easy to integrate Sprite Kit into your custom game tools. By integrating Sprite Kit into your games tools, you can build your content in the tools and read it directly into your game engine. A data-driven design lets artists, game designers, and game programmers collaborate to build your game's content.

## Organize Game Content into Scenes

Scenes are the fundamental building blocks for creating Sprite Kit content. When you start a new game project, one of your tasks is to define which scenes are needed and when transitions occur between those scenes. Scenes usually represent modes of play or content that appears to the player. Usually, it is easy to see when you need a new scene; if your game needs to replace all of the content onscreen, use a transition to a new scene.

Designing your game's scenes and the transitions between them is similar to the role of view controllers in a traditional iOS app. In an iOS app, content is implemented by view controllers. Each view controller creates a collection of views to draw that content. Initially, one view controller is presented by the window. Later, when the user interacts with the view controller's views, it might trigger a transition to another view controller and its content. For example, selecting an item in a table view might bring up a detail view controller to display the contents of the selected item.

Scenes do not have a default behavior, like storyboards do in a traditional iOS app. Instead, you define and implement the behaviors for scenes. These behaviors include:

- When new scenes are created
- The contents of each scene
- When transitions occur between scenes
- The visual effect used to perform a transition
- How data is transferred from one scene to another

For example, you could implement a model similar to a segue, where a new scene is always instantiated on a transition. Or you could design your game engine to use scenes that it keeps around persistently. Each approach has its benefits:

- If a scene is instantiated every time a transition occurs, it is always created in a clean, known state. This means that you don't have to worry about resetting any internal state of the scene, which can often have subtle bugs.
- If a scene is persistent, then you can transition back to the scene and have it be in the same state it was in when you left it. This design is useful in any kind of game where you need to quickly transition between multiple scenes of content.

## Allow Your Scene Designs to Evolve

Typically, a new scene is going to be developed in stages. At the start, you might be working with test apps and experimental ideas to understand how Sprite Kit works. But later, as your game gets more sophisticated, your scenes need to adapt.

In test apps, and some simple games, all of your logic and code goes in the scene subclass. The scene manipulates the node tree and the contents of each node in the tree, running actions or changing other behaviors as necessary. The project is simple enough that all of the code can live in a single class.

The second stage of a project usually happens when the rendering or game logic starts getting longer or more complex. At this stage, you usually start breaking out specific behaviors and implementing them in other classes. For example, if your game includes the concept of a camera, you might create a `CameraNode` class to encapsulate the camera behavior. You might then create other node classes to encapsulate other behaviors. For example, you might create separate node classes to represent units in your game.

In the most sophisticated projects, artificial intelligence and other concepts become more important. In these designs, you may end up creating classes that work independently of Sprite Kit. Objects of these classes perform work on behalf of the scene, but are not specifically tied to it. These classes are usually extracted from your Sprite Kit subclasses when you realize that many of your methods are implementing game logic without really touching any Sprite Kit content.

## Limit the Tree's Contents to Improve Performance

When Sprite Kit renders a frame, it culls all of the nodes that are not visible on screen. In theory, this means you could simply keep all of your content to the scene, and let Sprite Kit do all the work to manage it. For games with modest rendering requirements, this design would be adequate. But as your game gets larger and more sophisticated, you need to do more work to ensure good performance from Sprite Kit.

Typically, a node needs to be part of the node tree because:

- It has a reasonably good chance of being rendered in the near future.
- The node runs actions that are required for accurate gameplay.
- The node has a physics body that is required for accurate gameplay.

When a node does not meet any of these requirements, it is usually better to remove it from the tree, particularly if it has many children of its own. For example, emitter nodes often provide special effects without impacting gameplay at all, and they emit a large number of particles, which can be costly to render. If you had a large number of emitters in the scene, but offscreen, then the scene potentially may need to process hundreds or thousands of invisible nodes. Better to remove the emitter nodes until they are about to become visible.

Typically, the design of your culling algorithm is based on your gameplay. For example:

- In a racing game, the player is usually traveling around a track in a consistent direction. Because of this, you can usually predict what content is going to be visible in the near future and preload it. As the player advances through the race track, you can remove nodes the player can no longer see.
- In an adventure game, the player may be in a scrolling environment which permits movement in arbitrary directions. When the player moves through the world, you might be able to predict which terrain is nearby and which terrain is not. Then, only include the terrain for the local content.

When content is always going to be added and removed at once, consider using an interim node object to collect a set of content. The content can be quite complex, yet still be added to the scene with a single method call.

## What Shouldn't Be in a Scene

When you first design a Sprite Kit game, it may seem like the scene class is doing a lot of the work. Part of the process of tuning your app is deciding whether the scene should perform a task or whether some other object in your game should do it. For example, you might consider moving work to another object when:

- The content or app logic is shared by multiple scenes.
- The content or app logic is particularly expensive to set up and only needs to be performed once.

For example, if your game uses the same textures for all its gameplay, you might create a special loading class that runs once at startup. You perform the work of loading the textures once, and then leave them in memory. If a scene object is deleted and recreated to restart gameplay, the textures do not need to be reloaded.

## Use Subclassing to Create Your Own Node Behaviors

Designing new games requires you to make subclasses of the `SKScene` class. However, the other node classes in Sprite Kit are also designed to be subclassed so that you can add custom behavior. For example, you might subclass the `SKSpriteNode` class to add AI logic specific to your game. Or, you might subclass the `SKNode` class to create a class that implements a particular drawing layer in your scene. And if you want to directly implement interactivity in a node, you must create a subclass.

When you design a new node class, there are implementation details specific to Sprite Kit that are important to understand. But you also need to consider the role that the new class plays in your game and how objects of the class interact with other objects. You need to create well-defined class interfaces and calling conventions so that objects interoperate without subtle bugs slowing your development process.

Here are important guidelines to follow when creating your own subclasses:

- All of the standard node classes support the `NSCopying` and `NSCoding` protocols. If your subclass adds new properties or instance variables, then your subclass should also implement these behaviors. This support is essential if you plan to copy nodes within your game or use archiving to build your own game tools.
- Although nodes are similar to views, you cannot add new drawing behavior to a node class. You must work through the node's existing methods and properties. This means either controlling a node's own properties (such as changing a sprite's texture) or adding additional nodes and controlling their behavior. In either case, you need to consider how your class is going to interact with other parts of your code. You may need to establish your own calling conventions to avoid subtle rendering bugs. For example, one common convention is to avoid adding children to a node object that creates and manages its own child nodes.
- In many cases, expect to add methods that can be called during the scene's pre-processing and post-processing steps. Your scene coordinates these steps, but focused node subclasses perform the work.
- If you want to implement event handling in a node class, you must implement separate event-handling code for iOS and OS X. The `SKNode` class inherits from `NSResponder` on OS X and `UIResponder` on iOS.
- In some game designs, you can rely on the fact that a particular combination of classes is always going to be used together in a specific scene. In other designs, you may want to create classes that can be used in multiple scenes. The more important reuse is to your design, the more time you should spend designing clean interfaces for objects to interact with each other. When two classes are dependent on each other,

use delegation to break that dependency. Most often, you do this by defining a delegate on your node and a protocol for delegates to implement. Your scene (or another node, such as the node's parent) implements this protocol. Your node class can then be reused in multiple scenes, without needing to know the scene's class.

- Keep in mind that when a node class is being initialized, it is not yet in the scene, so its `parent` and `scene` properties are `nil`. You may need to defer some initialization work until after the node is added to the scene.

## Drawing Your Content

A large part of building a node tree is organizing the graphical content that needs to be drawn. What needs to be drawn first? What needs to be drawn last? How are these things rendered?

Consider the following advice when designing your node tree:

- Do not add the content nodes or physics bodies to the scene directly. Instead, add one or more `SKNode` objects to the tree to represent different layers of content in your game and then work with those layer objects. You can then precisely control the contents of each layer. For example, you can rotate one layer of the content without rotating all of the scene's content. It also becomes easier for you to remove or replace portions of your scene rendering with other code. For example, if game scores and other information is played in a heads-up display layer, then this layer can be removed when you want to take screenshots. In a very complex app, you might continue this pattern deeper into your node tree by adding children to a layer node.

  When you use layers to organize your content, consider how the layers interact with one another. Do they know anything about each other's content? Does a higher-level layer need to know anything about how the lower layers are rendered in order to render its own content?
- Use clipping and effect nodes sparingly. Both are very powerful, but can be expensive, especially when nested together within the node tree.
- Whenever possible, nodes that are rendered together should use the same blend mode. If all of the children of a node use the same blend mode and texture atlas, then Sprite Kit can usually draw these sprites in a single drawing pass. On the other hand, if the children are organized so that the drawing mode changes for each new sprite, then Sprite Kit might perform as one drawing pass per sprite, which is quite inefficient.
- When designing emitter effects, use low particle birth rates whenever possible. Particles are not free; each particle adds rendering and drawing overhead.
- By default, sprites and other content are blended using an alpha blend mode. If the sprite's content is opaque, such as for a background image, use the `SKBlendModeReplace` blend mode.

- Use game logic and art assets that match Sprite Kit's coordinate and rotation conventions. This means orienting artwork to the right. If you orient the artwork in some other direction, you need to convert angles between the conventions used by the art and the conventions used by Sprite Kit. For example, if the artwork is oriented upward, then you add `PI/2` radians to the angle to convert from Sprite Kit's convention to your art's convention and vice versa.
- Turn on the diagnostic messages in the `SKView` class. Use the frame rate as a general diagnostic for performance, and the node and drawing pass counts to further understand how the content was rendered. You can also use Instruments and its OpenGL diagnostic tools to find additional information about where your game is spending its time.
- Test your game on real hardware, and on devices with different characteristics. In many cases, the balance of CPU resources and GPU resources are different on each Mac or iOS device. Testing on multiple devices helps you to determine whether your game runs well on the majority of devices.

## Working with Game Data

At any given time, your game manages a lot of data, including the positions of nodes in the scene. But it also includes static data such as:

- Art assets and required data to render that artwork correctly
- Level or puzzle layouts
- Data used to configure the gameplay (such as the speed of the monster and how much damage it does when it attacks)

Whenever possible, avoid embedding your game data directly in the game code. When the data changes, you are forced to recompile the game, which usually means that a programmer is involved in the design changes. Instead, keep data independent of the code, so that a game designer or artist can make changes directly to the data.

The best place to store game data depends on where that data is used within your game. For data not related to Sprite Kit, a property list stored in your app bundle is a good solution. However, for Sprite Kit data, you have another option. Because all Sprite Kit classes support archiving, you can simply create archives of important Sprite Kit objects and then include these archives in your game. For example, you might:

- Store a game level as an archive of a scene node. This archive includes the scene, all of its descendants in the node tree, and all of their connected physics bodies, joints, and actions.
- Store individual archives for specific preconfigured nodes, such a node for each monster. Then, when a new monster needs to be created, you load it from the archive.
- Store saved games as a scene archive.

- Build your own tools to create and edit archived content. Then, your game designers and artists can work within these tools to create game objects and archive them using a format that your game reads. Your game engine and your tools would share common classes.
- You could store Sprite Kit data in a property list. Your game loads the property list and uses it to create game assets.

Here are a few guidelines for working with archives:

- Use the `userData` property on nodes to store game-specific data, especially if you are not implementing your own subclasses.
- Avoid hardcoding references to specific nodes. Instead, give interesting nodes a unique `name` property and search for them in the tree.
- Custom actions that call blocks cannot be archived. You need to create and add those actions within your game.
- Most node objects provide all the necessary properties to determine what they are and how they were configured. However, actions and physics bodies do not. This means that when developing your own game tools, you cannot simply archive actions and physics bodies and use these archives to store your tool data. Instead, the archives should only be the final output from your game tools.

# Document Revision History

This table describes the changes to *Sprite Kit Programming Guide*.

| Date | Notes |
| --- | --- |
| 2014-02-11 | Clarified a couple of steps in the tutorial. |
| 2013-12-16 | Corrected an incorrect method name in two code listings. |
| 2013-10-22 | Updated the lighting example to call spriteNodeWithTexture:. |
| 2013-09-18 | New document that describes how to use Sprite Kit to implement games and other apps featuring arbitrary animations. |

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010