Game Development with Swift

Embrace the mobile gaming revolution and bring your iPhone game ideas to life with Swift

Stephen Haney
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Mix in the Physics'
- A synopsis of the book’s content
- More information on Game Development with Swift
About the Author

**Stephen Haney** began his programming journey at the age of 8 on a dusty, ancient laptop using BASIC. He has been fascinated with building software and games ever since. Now well versed in multiple languages, he most enjoys programming as a creative outlet. He believes that indie game development is an art form: an amazing combination of visual, auditory, and psychological challenges, rewarding to both the player and the creator.

He enjoyed writing this book and sincerely hopes that it directly furthers your career or hobby.
There has never been a better time to be a game developer. The App Store provides a unique opportunity to distribute your ideas to a massive audience. Now, Swift has arrived to bolster our toolkit and provide a smoother development experience. Swift is new, but is already hailed as an excellent, well-designed language. Whether you are new to game development or looking to add to your expertise, I think you will enjoy making games with Swift.

My goal in writing this book is to share a fundamental knowledge of Swift and SpriteKit. We will work through a complete example game so that you learn every step of the Swift development process. Once you finish this text, you will be comfortable designing and publishing your own game ideas to the App Store, from start to finish.

Please reach out with any questions and share your game creations:

E-mail: stephen@thinkingswiftly.com
Twitter: @sdothaney

The first chapter explores some of Swift's best features. Let's get started!

What this book covers

Chapter 1, Designing Games with Swift, introduces you to best features on Swift, helps you set up your development environment, and launches your first SpriteKit project.

Chapter 2, Sprites, Camera, Actions!, teaches you the basics of drawing and animating with Swift. You will draw sprites, import textures into your project, and center the camera on the main character.
Chapter 3, *Mix in the Physics*, covers the physics simulation fundamentals: physics bodies, impulses, forces, gravity, collisions, and more.

Chapter 4, *Adding Controls*, explores various methods of mobile game controls: device tilt and touch input. We will also improve the camera and core gameplay of our example game.

Chapter 5, *Spawning Enemies, Coins, and Power-ups*, introduces the cast of characters we will use in our example game and shows you how to create custom classes for each NPC type.

Chapter 6, *Generating a Never-Ending World*, explores the SpriteKit scene editor, builds encounters for the example game, and creates a system to loop encounters endlessly.

Chapter 7, *Implementing Collision Events*, delves into advanced physics simulation topics and adds custom events when sprites collide.

Chapter 8, *Polishing to a Shine – HUD, Parallax Backgrounds, Particles, and More*, adds the extra features that make every great game shine. Create parallax backgrounds, learn about SpriteKit’s particle emitters, and add a heads-up display overlay to your games.

Chapter 9, *Adding Menus and Sounds*, builds a basic menu system and illustrates two methods of playing sounds in your games.

Chapter 10, *Integrating with Game Center*, links our example game to the Apple Game Center for leaderboards, achievements, and friendly challenges.

Chapter 11, *Ship It! Preparing for the App Store and Publication*, covers the essentials of packaging your game and submitting it to the App Store.
Mix in the Physics

SpriteKit includes a fully functional physics engine. It is easy to implement and very useful; most mobile game designs require some level of physical interaction between game objects. In our game, we want to know when the player runs into the ground, an enemy, or a power-up. The physics system can track these collisions and execute our specific game code when any of these events occur. SpriteKit's physics engine can also apply gravity to the world, bounce and spin colliding sprites against each other, and create realistic movement through impulses – and it does all of this before every single frame is drawn to the screen.

The topics in this chapter include:

- Adopting a protocol for consistency
- Organizing game objects into classes
- Adding the player's character
- Renovating the GameScene class
- Physics bodies and gravity
- Exploring physics simulation mechanics
- Movement with impulses and forces
- Bumping bees into bees

Laying the foundation

So far, we have learned through small bits of code, individually added to the GameScene class. The intricacy of our application is about to increase. To build a complex game world, we will need to construct re-usable classes and actively organize our new code.
Mix in the Physics

**Following protocol**

To start, we want individual classes for each of our game objects (a bee class, a player penguin class, a power-up class, and so on). Furthermore, we want all of our game object classes to share a consistent set of properties and methods. We can enforce this commonality by creating a protocol, or a blueprint for our game classes. The protocol does not provide any functionality on its own, but each class that adopts the protocol must follow its specifications exactly before Xcode can compile the project. Protocols are very similar to interfaces, if you are from a Java or C# background.

Add a new file to your project (right-click in the project navigator and choose **New File**, then **Swift File**) and name it `GameSprite.swift`. Then add the following code to your new file:

```swift
import SpriteKit

protocol GameSprite {
    var textureAtlas: SKTextureAtlas { get set }
    func spawn(parentNode: SKNode, position: CGPoint, size: CGSize)
    func onTap()
}
```

Now, any class that adopts the `GameSprite` protocol must implement a `textureAtlas` property, a `spawn` function, and an `onTap` function. We can safely assume that the game objects provide these implementations when we work with them in our code.

**Reinventing the bee**

Our old bee is working wonderfully, but we want to spawn many bees throughout the world. We will create a `Bee` class, inheriting from `SKSpriteNode`, so we can cleanly stamp as many bees to the world as we please.

It is a common convention to separate each class into its own file. Add a new Swift file to your project and name it `Bee.swift`. Then, add this code:

```swift
import SpriteKit

// Create the new class Bee, inheriting from SKSpriteNode
// and adopting the GameSprite protocol:
class Bee: SKSpriteNode, GameSprite {
    // We will store our texture atlas and bee animations as
    // class wide properties.
    var textureAtlas: SKTextureAtlas =
```
Chapter 3

SKTextureAtlas(named: "bee.atlas")

var flyAnimation = SKAction()

// The spawn function will be used to place the bee into
// the world. Note how we set a default value for the size
// parameter, since we already know the size of a bee
func spawn(parentNode: SKNode, position: CGPoint, size: CGSize = CGSize(width: 28, height: 24)) {
    parentNode.addChild(self)
    createAnimations()
    self.size = size
    self.position = position
    self.runAction(flyAnimation)
}

// Our bee only implements one texture based animation.
// But some classes may be more complicated,
// So we break out the animation building into this function:
func createAnimations() {
    let flyFrames: [SKTexture] =
        [textureAtlas.textureNamed("bee.png"),
         textureAtlas.textureNamed("bee_fly.png")]
    let flyAction = SKAction.animateWithTextures(flyFrames,
                                                 timePerFrame: 0.14)
    flyAnimation = SKAction.repeatActionForever(flyAction)
}

// onTap is not wired up yet, but we have to implement this
// function to adhere to our protocol.
// We will explore touch events in the next chapter.
func onTap() {}

It is now easy to spawn as many bees as we like. Switch back to GameScene.swift, and add this code in didMoveToView:

// Create three new instances of the Bee class:
let bee2 = Bee()
let bee3 = Bee()
let bee4 = Bee()

// Use our spawn function to place the bees into the world:
bee2.spawn(world, position: CGPoint(x: 325, y: 325))
bee3.spawn(world, position: CGPoint(x: 200, y: 325))
bee4.spawn(world, position: CGPoint(x: 50, y: 200))
Run the project. Bees, bees everywhere! Our original bee is flying back and forth through a swarm. Your simulator should look like this:

Depending on how you look at it, you may perceive that the new bees are moving and the original bee is still. We need to add a point of reference. Next, we will add the ground.

**The icy tundra**
We will add some ground at the bottom of the screen to serve as a constraint for player positioning and as a reference point for movement. We will create a new class named `Ground`. First, let us add the texture atlas for the ground art to our project.

**Another way to add assets**
We will use a different method of adding files to Xcode. Follow these steps to add the new artwork:

1. In Finder, navigate to the asset pack you downloaded in Chapter 2, Sprites, Camera, Actions!, and then to the Environment folder.

2. You learned to create a texture atlas earlier, for our bee. I have already created texture atlases for the rest of the art we use in this game. Locate the `ground.atlas` folder.
3. Drag and drop this folder into the project manager in Xcode, under the project folder, as seen in this screenshot:

![Folder in Xcode project manager]

4. In the dialog box, make sure your settings match the following screenshot, and then click **Finish**:

![Dialog box for adding files]

Perfect – you should see the ground texture atlas in the project navigator.
Adding the Ground class

Next, we will add the code for the ground. Add a new Swift file to your project and name it Ground.swift. Use the following code:

```swift
import SpriteKit

// A new class, inheriting from SKSpriteNode and adhering to the GameSprite protocol.
class Ground: SKSpriteNode, GameSprite {
    var textureAtlas: SKTextureAtlas = SKTextureAtlas(named: "ground.atlas")
    var groundTexture: SKTexture?

    func spawn(parentNode: SKNode, position: CGPoint, size: CGSize) {
        parentNode.addChild(self)
        self.size = size
        self.position = position
        // This is one of those unique situations where we use non-default anchor point. By positioning the ground by its top left corner, we can place it just slightly above the bottom of the screen, on any of screen size.
        self.anchorPoint = CGPointMake(0, 1)

        // Default to the ice texture:
        if groundTexture == nil {
            groundTexture = textureAtlas.textureNamed("ice-tile.png");
        }

        // We will create child nodes to repeat the texture.
        createChildren()
    }

    // Build child nodes to repeat the ground texture
    func createChildren() {
        // First, make sure we have a groundTexture value:
        if let texture = groundTexture {
            var tileCount: CGFloat = 0
            let textureSize = texture.size()
            // We will size the tiles at half the size
            // of their texture for retina sharpness:
```
let tileSize = CGSize(width: textureSize.width / 2,
    height: textureSize.height / 2)

// Build nodes until we cover the entire Ground width
while tileCount * tileSize.width < self.size.width {
    let tileNode = SKSpriteNode(texture: texture)
    tileNode.size = tileSize
    tileNode.position.x = tileCount * tileSize.width
    // Position child nodes by their upper left corner
    tileNode.anchorPoint = CGPoint(x: 0, y: 1)
    // Add the child texture to the ground node:
    self.addChild(tileNode)
    tileCount++
}

// Implement onTap to adhere to the protocol:
func onTap() {}
2. Locate the `didMoveToView` function. Add the following code at the bottom, underneath our bee spawning lines:

```swift
// size and position the ground based on the screen size.
// Position X: Negative one screen width.
// Position Y: 100 above the bottom (remember the ground's top
// left anchor point).
let groundPosition = CGPoint(x: -self.size.width, y: 100)
// Width: 3x the width of the screen.
// Height: 0. Our child nodes will provide the height.
let groundSize = CGSize(width: self.size.width * 3, height: 0)

// Spawn the ground!
ground.spawn(world, position: groundPosition, size: groundSize)
```

Run the project. You will see the icy tundra appear underneath our bees. This small change goes a long way towards creating the feeling that our central bee is moving through space. Your simulator should look like this:
A wild penguin appears!

There is one more class to build before we start our physics lesson: the Player class! It is time to replace our moving bee with a node designated as the player.

First, we will add the texture atlas for our penguin art. By now, you are familiar with adding files through the project navigator. Add the Pierre art as you did previously with the ground assets. I named Pierre's texture atlas `pierre.atlas`. You can find it in the asset pack, inside the Pierre folder.

Once you add Pierre's texture atlas to the project, you can create the Player class. Add a new Swift file to your project and name it `Player.swift`. Then add this code:

```swift
import SpriteKit

class Player : SKSpriteNode, GameSprite {
    // Pierre has multiple animations. Right now we will
    // create an animation for flying up, and one for going down:
    var flyAnimation = SKAction()
    var soarAnimation = SKAction()

    func spawn(parentNode: SKNode, position: CGPoint, size: CGSize = CGSize(width: 64, height: 64)) {
        parentNode.addChild(self)
        createAnimations()
        self.size = size
        self.position = position
        // If we run an action with a key, "flapAnimation",
        // we can later reference that key to remove the action.
        self.runAction(flyAnimation, withKey: "flapAnimation")
    }

    func createAnimations() {
        let rotateUpAction = SKAction.rotateToAngle(0, duration: 0.475)
        rotateUpAction.timingMode = .EaseOut
        let rotateDownAction = SKAction.rotateToAngle(-1, duration: 0.8)
        rotateDownAction.timingMode = .EaseIn
    }
}
```
Mix in the Physics

```swift
// Create the flying animation:
let flyFrames: [SKTexture] = [
    textureAtlas.textureNamed("pierre-flying-1.png"),
    textureAtlas.textureNamed("pierre-flying-2.png"),
    textureAtlas.textureNamed("pierre-flying-3.png"),
    textureAtlas.textureNamed("pierre-flying-4.png"),
    textureAtlas.textureNamed("pierre-flying-3.png"),
    textureAtlas.textureNamed("pierre-flying-2.png"
]
let flyAction = SKAction.animateWithTextures(flyFrames,
    timePerFrame: 0.03)
// Group together the flying animation frames with a
// rotation up:
flyAnimation = SKAction.group([{
    SKAction.repeatActionForever(flyAction),
    rotateUpAction
}])

// Create the soaring animation, just one frame for now:
let soarFrames: [SKTexture] = [
    textureAtlas.textureNamed("pierre-flying-1.png")
]
let soarAction = SKAction.animateWithTextures(soarFrames,
    timePerFrame: 1)
// Group the soaring animation with the rotation down:
soarAnimation = SKAction.group([{
    SKAction.repeatActionForever(soarAction),
    rotateDownAction
}])
}

func onTap() {} 
```

Great! Before we continue, we need to replace our original bee with an instance of the new `Player` class we just created. Follow these steps to replace the bee:

1. In `GameScene.swift`, near the top, remove the line that creates a bee constant in the `GameScene` class. Instead, we want to instantiate an instance of `Player`. Add the new line: `let player = Player()`.
2. Completely delete the `addTheFlyingBee` function.
3. In `didMoveToView`, remove the line that calls `addTheFlyingBee`.

[48]
4. In `didMoveToView`, at the bottom, add a new line to spawn the player:
   ```swift
   player.spawn(world, position: CGPoint(x: 150, y: 250))
   ```

5. Further down, in `didSimulatePhysics`, replace the references to the bee with references to `player`. Recall that we created the `didSimulatePhysics` function in Chapter 2, *Sprites, Camera, Actions!*, when we centered the camera on one node.

We have successfully transformed the original bee into a penguin. Before we move on, we will make sure your `GameScene` class includes all of the changes we have made so far in this chapter. After that, we will begin to explore the physics system.

## Renovating the `GameScene` class

We have made quite a few changes to our project. Luckily, this is the last major overhaul of the previous animation code. Moving forward, we will use the terrific structure we built in this chapter. At this point, your `GameScene.swift` file should look something like this:

```swift
class GameScene: SKScene {
    let world = SKNode()
    let player = Player()
    let ground = Ground()

    override func didMoveToView(view: SKView) {
        // Set a sky-blue background color:
        self.backgroundColor = UIColor(red: 0.4, green: 0.6, blue: 0.95, alpha: 1.0)

        // Add the world node as a child of the scene:
        self.addChild(world)

        // Spawn our physics bees:
        let bee2 = Bee()
        let bee3 = Bee()
        let bee4 = Bee()
        bee2.spawn(world, position: CGPoint(x: 325, y: 325))
        bee3.spawn(world, position: CGPoint(x: 200, y: 325))
        bee4.spawn(world, position: CGPoint(x: 50, y: 200))
    }
}
```
Mix in the Physics

// Spawn the ground:
let groundPosition = CGPoint(x: -self.size.width, y: 30)
let groundSize = CGSize(width: self.size.width * 3, height: 0)
ground.spawn(world, position: groundPosition, size: groundSize)

// Spawn the player:
player.spawn(world, position: CGPoint(x: 150, y: 250))

override func didSimulatePhysics() {
    let worldXPos = -(player.position.x * world.xScale – (self.size.width / 2))
    let worldYPos = -(player.position.y * world.yScale – (self.size.height / 2))
    world.position = CGPoint(x: worldXPos, y: worldYPos)
}

Run the project. You will see our new penguin hovering near the bees. Great work; we are now ready to explore the physics system with all of our new nodes. Your simulator should look something like this screenshot:
Exploring the physics system

SpriteKit simulates physics with **physics bodies**. We attach physics bodies to all the nodes that need physics computations. We will set up a quick example before exploring all of the details.

**Dropping like flies**

Our bees need to be part of the physics simulation, so we will add physics bodies to their nodes. Open your Bee.swift file and locate the spawn function. Add the following code at the bottom of the function:

```swift
// Attach a physics body, shaped like a circle
// and sized roughly to our bee.
self.physicsBody = SKPhysicsBody(circleOfRadius: size.width / 2)
```

It is that easy to add a node to the physics simulation. Run the project. You will see our three Bee instances drop off the screen. They are now subject to gravity, which is on by default.

**Solidifying the ground**

We want the ground to catch falling game objects. We can give the ground its own physics body so the physics simulation can stop the bees from falling through it. Open your Ground.swift file, locate the spawn function, and then add this code at the bottom of the function:

```swift
// Draw an edge physics body along the top of the ground node.
// Note: physics body positions are relative to their nodes.
// The top left of the node is X: 0, Y: 0, given our anchor point.
// The top right of the node is X: size.width, Y: 0
let pointTopRight = CGPoint(x: size.width, y: 0)
self.physicsBody = SKPhysicsBody(edgeFromPoint: CGPointZero, toPoint: pointTopRight)
```
Run the project. The bees will now quickly drop and then stop once they collide with the ground. Notice how bees that fall farther bounce more energetically. After the bees land, your simulator will look like this:

![Simulator with bees landing](image)

**Checkpoint 3-A**

Great work so far. We have added a lot of structure to our game and started to explore the physics system. If you would like to download my project to this point, do so here:

http://www.thinkingswiftly.com/game-development-with-swift/chapter-3

**Exploring physics simulation mechanics**

Let's take a closer look at the specifics of SpriteKit's physics system. For instance, why are the bees subject to gravity, but the ground stays where it is? Though we attached physics bodies to both nodes, we actually used two different styles of physics bodies. There are three types of physics bodies, and each behaves slightly differently:

- **Dynamic** physics bodies have volume and are fully subject to forces and collisions in the system. We will use dynamic physics bodies for most parts of the game world: the player, enemies, power-ups, and others.
• **Static** physics bodies have volume but no velocity. The physics simulation does not move nodes with static bodies but they can still collide with other game objects. We can use static bodies for walls or obstacles.

• **Edge** physics bodies have no volume and the physics simulation will never move them. They mark off the boundaries of movement; other physics bodies will never cross them. Edges can cross each other to create small containment areas.

Voluminous (dynamic and static) bodies have a variety of properties that modify how they react to collisions and movement through space. This allows us to create a wide range of realistic physics effects. Each property controls one aspect of a body's physical characteristics:

• **Restitution** determines how much energy is lost when one body bounces into another. This changes the body’s bounciness. SpriteKit measures restitution on a scale from 0.0 to 1.0. The default value is 0.2.

• **Friction** describes the amount of force necessary to slide one body against another body. This property also uses a scale of 0.0 to 1.0, with a default value of 0.2.

• **Damping** determines how quickly a body slows as it moves through space. You can think of damping as air friction. Linear damping determines how quickly a body loses speed, while angular damping affects rotation. Both measure from 0.0 to 1.0, with a default value of 0.1.

• **Mass** is measured in kilograms. It describes how far colliding objects push the body and factors in momentum during movement. Bodies with more mass will move less when hit by another body and will push other bodies further when they collide with them. The physics engine automatically uses the mass and the area of the body to determine **density**. Alternatively, you can set the density and let the physics engine calculate mass. It is usually more intuitive to set the mass.

All right – enough with the textbook! Let us solidify our learning with some examples.

First, we want gravity to skip our bees. We will set their flight paths manually. We need the bees to be dynamic physics bodies in order to interact properly with other nodes, but we need these bodies to ignore gravity. For such instances, SpriteKit provides a property named **affectedByGravity**. Open Bee.swift and, at the bottom of the spawn function, add this code:

```swift
self.physicsBody?.affectedByGravity = false
```
The question mark after physicsBody is optional chaining. We need to unwrap physicsBody, since it is optional. If physicsBody is nil, the entire statement will return nil (instead of triggering an error). You can think of it as gracefully unwrapping an optional property with an inline statement.

Run the project. The bees should now hover in place as they did before we added their bodies. However, SpriteKit's physics simulation now affects them; they will react to impulses and collisions. Great, let us purposefully collide the bees.

Bee meets bee
You may have noticed that we positioned bee2 and bee3 at the same height in the game world. We only need to push one of them horizontally to create a collision – perfect crash test dummies! We can use an impulse to create velocity for the outside bee.

Locate the didMoveToView function in GameScene.swift. At the bottom, below all of our spawn code, add this line:

```swift
bee2.physicsBody?.applyImpulse(CGVector(dx: -3, dy: 0))
```

Run the project. You will see the outermost bee fly towards the middle and crash into the inner bee. This pushes the inner bee to the left and slows the first bee from the contact.

Attempt the same experiment with a variable: increased mass. Before the impulse line, add this code to adjust the mass of bee2:

```swift
bee2.physicsBody?.mass = 0.2
```

Run the project. Hmm, our heavier bee does not move very far with the same impulse (it is a 200-gram bee, after all.) It eventually bumps into the inner bee, but it is not a very exciting collision. We will need to crank up the impulse to propel our beefier bee. Change the impulse line to use a dx value of -15:

```swift
bee2.physicsBody?.applyImpulse(CGVector(dx: -15, dy: 0))
```
Run the project again. This time, our impulse provides enough energy to move the heavy bee in an interesting way. Notice how much energy the heavy bee transfers to the normal bee when they collide; the lighter bee shoots away after contact. Both bees possess enough momentum to eventually slide completely off the screen. Your simulator should look something like this screenshot, just before the bees slide off the screen:

Before you move on, you may wish to experiment with the various physics properties that I outlined earlier in the chapter. You can create many collision variations; the physics simulation offers a lot of depth with out much effort.

**Impulse or force?**

You have several options for moving nodes with physics bodies:

- An impulse is an immediate, one-time change to a physics body's velocity. In our test, an impulse gave the bee its velocity, and it slowly bled speed to damping and its collision. Impulses are perfect for projectiles: missiles, bullets, disgruntled birds, and so on.
Mix in the Physics

- A force applies velocity for only one physics calculation cycle. When we use a force, we typically apply it before every frame. Forces are useful for rocket ships, cars, or anything else that is continually self-propelled.
- You can also edit the `velocity` and `angularVelocity` properties of a body directly. This is useful for setting a manual velocity limit.

**Checkpoint 3-B**

We have made a number of structural changes to our project in this chapter. Feel free to download my project to this point:

http://www.thinkingswiftly.com/game-development-with-swift/chapter-3

**Summary**

We have made great strides in this chapter. Our new class organization will serve us well over the course of this book. We learned how to use protocols to enforce commonality across classes, encapsulated our game objects into distinct classes, and explored tiling textures over the width of the ground node. Finally, we cleaned out some of our previous learning code from `GameScene` and used the new class system to spawn all of our game objects.

We also applied the physics simulation to our game. We have only scratched the surface of the powerful physics system in SpriteKit – we will dive deeper into custom collision events in *Chapter 7, Implementing Collision Events* – but we have already gained quite a bit of functionality. We explored the three types of physics bodies and studied the various physics properties you can use to fine-tune the physical behavior of your game objects. Then, we put all of our hard work into practice by bumping our bees together and watching the results.

Next, we will try several control schemes and move our player around the game world. This is an exciting addition; our project will begin to feel like a true game in *Chapter 4, Adding Controls*. 
Where to buy this book

You can buy Game Development with Swift from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.