Extracted from:

Build iOS Games with Sprite Kit

Unleash Your Imagination in Two Dimensions

This PDF file contains pages extracted from *Build iOS Games with Sprite Kit*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2014 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Build iOS Games with Sprite Kit

Unleash Your Imagination in Two Dimensions

Jonathan Penn
and Josh Smith

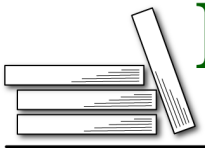
Edited by Rebecca Gulick

Build iOS Games with Sprite Kit

Unleash Your Imagination in Two Dimensions

Jonathan Penn
Josh Smith

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (indexer)
Cathleen Small (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Copyright © 2014 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-94122-210-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2014

Follow the Bouncing Ball

In this section, we're going to experiment to build a ball and plunger and then extract the nodes for the plunger and pinball into their own special SKNode subclasses.

Building any kind of physics game is a bit of a challenge. If you picked up this book and jumped in right here, you might want to at least skim the previous chapters to understand Sprite Kit's vocabulary first. You'll be leaning on a lot of accumulated knowledge and accumulating more as you go.

Download the source code for the book by following the instructions back in *How to Get the Most out of This Book*, on page ?. We'll be starting with the Xcode project in 06-Physics/step01. This is similar to the project you created when you followed the instructions to create a new project based on the Sprite Kit template back in *Setting Up a Sprite Kit Project*, on page ?. Except this one comes with all the graphics and sound files that we'll need for the pinball game ready to go. Here we'll focus on the physics engine, not setting up the Xcode project. Starting with the code in the 06-Physics/step01 directory will help you jump right into the fun stuff.

In the RCWMyScene.m file, you'll find that the -initWithSize: method used to construct the scene delegates to a second method to do the actual setup.

06-Physics/step01/PhysicsBall/RCWMyScene.m

```
- (id)initWithSize:(CGSize)size
{
    if (self = [super initWithSize:size]) {
        [self setUpScene];
    }
    return self;
}
```

Because there is so much setup to do for this game, it's easier to break it out into the -setUpScene method so we don't have so much indentation inside the if statement of -initWithSize:. For the remainder of this section, we'll be doing all our work in the -setUpScene method.

Creating a Physics Body for the Ball

Let's put a ball on a white screen. In the -setUpScene method, we'll set the background color and create a sprite node with the pinball.png image positioned in the center.

06-Physics/step01/PhysicsBall/RCWMyScene.m

```
- (void)setUpScene
{
```

```
    self.backgroundColor = [SKColor whiteColor];
    SKSpriteNode *ball = [SKSpriteNode spriteNodeWithImageNamed:@"pinball.png"];
    ball.position = CGPointMake(self.size.width/2, self.size.height/2);
    ball.size = CGSizeMake(20, 20);
    [self addChild:ball];
}
```

We first set the background color to [SKColor whiteColor], which will give good contrast to the fast-moving action of the ball. Then we create a sprite node with the pinball image texture and position it at the center of the scene's coordinates. Remember that, by default, a scene has the {0,0} origin in the bottom left of the device screen. For a refresher on Sprite Kit's coordinate system, refer back to Figure 8, *Comparing Sprite Kit and UIKit coordinates*, on page ?.

To engage the physics engine, we need to assign a physics body to the ball.

06-Physics/step01/PhysicsBall/RCWMyScene.m

```
- (void)setUpScene
{
    self.backgroundColor = [SKColor whiteColor];
    SKSpriteNode *ball = [SKSpriteNode spriteNodeWithImageNamed:@"pinball.png"];
    ball.position = CGPointMake(self.size.width/2, self.size.height/2);
    ball.size = CGSizeMake(20, 20);
    [self addChild:ball];

➤   ball.physicsBody = [SKPhysicsBody bodyWithCircleOfRadius:10];
}
```

The SKPhysicsBody objects define physics bodies within the physics world of the scene. Any node can participate in the physics simulation, but it *must* have a physics body assigned to it. Here we are using the +bodyWithCircleOfRadius: constructor method to create a specific physics body that is a circle with a 10-point radius centered around the ball.

This physics body happens to be the same size and shape as the texture of the sprite node. That's because we want a ball, after all, and it wouldn't make sense if the body were larger or smaller than what is visible.

• Click HERE to purchase this book now. discuss

But you can create bodies of any shape or size you want. If, say, you wanted a body that was a little smaller than the visible sprite texture, then it would give the appearance of overlapping other bodies. That's not what we want here for these nodes, but it's a useful technique to consider for your own ideas.

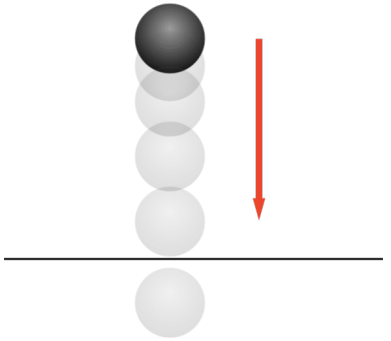


Figure 28—The ball falling off the screen

Build and run the game and watch what happens. The ball falls off the screen as if affected by gravity, similar to the figure shown here. But where did this gravity come from? Don't we have to set up a physics world for this to work first? That's the beauty of Sprite Kit. You have all you need to get started tinkering with the physics engine right away. Every Sprite Kit game has a physics world ready to go, yet dormant until physics bodies are assigned to nodes to start the calculations churning.

The physics world starts with a default gravity of 9.8 meters per second squared (m/s^2) in the downward direction. But we don't have to settle for that. In fact, let's change it to make the ball appear to fall slower.

06-Physics/step02/PhysicsBall/RCWMyScene.m

```
- (void)setUpScene
{
    self.backgroundColor = [SKColor whiteColor];
➤   self.physicsWorld.gravity = CGVectorMake(0, -2);
    SKSpriteNode *ball = [SKSpriteNode spriteNodeWithImageNamed:@"pinball.png"];
    ball.position = CGPointMake(self.size.width/2, self.size.height/2);
    ball.size = CGSizeMake(20, 20);
    [self addChild:ball];
    ball.physicsBody = [SKPhysicsBody bodyWithCircleOfRadius:10];
}
```

The `self.physicsWorld` property is always present on every `SKScene` object and contains an instance of `SKPhysicsWorld`. The `gravity` property on that world takes a *vector*, or a direction and magnitude, that determines the force of gravity. A `CGVector` is kind of like a `CGPoint` in that it has an `x` and a `y` component. This is saying that we want the gravity to be 2 m/s^2 in the negative, or downward, direction. Build and run the game, and you'll see the ball accelerate downward more slowly.

But we don't have to make gravity fall down. We can also make it fall diagonally up! Change the line to look like this:

```
self.physicsWorld.gravity = CGVectorMake(1, 2);
```

Now the vector says that the force of gravity should accelerate 1 m/s^2 to the right and 2 m/s^2 upward. Run the game now, and you'll see the ball "fall" up and to the right. This might seem counterintuitive at first. Shouldn't gravity always fall down? Remember, though, that games often present different points of view. *Space Run* doesn't use the physics engine and doesn't have gravity (it's in space, after all), but it uses a top-down view of the playing field. Maybe there could be a nearby planet that exerts gravity on all the nodes. Maybe gravity changes according to the tilt of the device as a game mechanic.

But this is pinball, of course. The table is supposed to be slightly slanted to allow the ball to drift toward the bottom, and you're not supposed to be able to tilt it. To establish these rules, we'll just keep the gravity fixed at 3.8 m/s^2 downward.

06-Physics/step03/PhysicsBall/RCWMyScene.m

```
self.physicsWorld.gravity = CGVectorMake(0, -3.8);
```

Why -3.8? What do meters per second squared mean in this world? Welcome to the esoteric side of physics engines. We're making this all up as we go! The physics engine tries its best to calculate how the bodies interact with each other and the world, but the result depends a lot on tinkering. All the details such as friction, mass, bounciness, and gravity come into play, and you'll find yourself tweaking values while searching for the illusion you want. When we, the authors, were experimenting with pinball physics, we came up with some reasonable numbers that made sense for the game as we saw it. (Of course, you are free to tinker away and make the pinball physics as zany as you want.)

While knowing a bit about physics in the real world would certainly be helpful, leave your PhD in physics at the door. Remember, this is an approximation for a game engine. As you'll see while we build our pinball game, we'll run into all sorts of problems that we'll have to work around and for which we'll bend the rules. Sprite Kit may bring delight to your players, but your thesis advisor might not be as impressed.

Bouncing the Ball on Another Physics Body

Let's add another body to the physics world so we can watch them interact. At the end of the `-setUpScene` method, we'll create a new sprite node to represent the plunger and position it below.

06-Physics/step04/PhysicsBall/RCWMyScene.m

```
- (void)setUpScene
```

```
{
    self.backgroundColor = [SKColor whiteColor];
    self.physicsWorld.gravity = CGVectorMake(0, -3.8);
    SKSpriteNode *ball = [SKSpriteNode spriteNodeWithImageNamed:@"pinball.png"];
    ball.position = CGPointMake(self.size.width/2, self.size.height/2);
    ball.size = CGSizeMake(20, 20);
    [self addChild:ball];
    ball.physicsBody = [SKPhysicsBody bodyWithCircleOfRadius:10];

➤   SKSpriteNode *plunger = [SKSpriteNode spriteNodeWithImageNamed:@"plunger.png"];
➤   plunger.position = CGPointMake(self.size.width/2, self.size.height/2 - 140);
➤   plunger.size = CGSizeMake(25, 100);
➤   [self addChild:plunger];
}
```

We're creating a sprite node with the plunger.png image texture and adding it to the scene below the ball. Run the game and watch what happens.

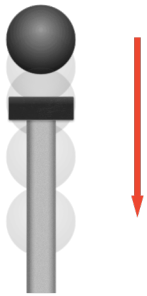


Figure 29—Ball falling through plunger

Whoops! The ball falls through the plunger. Remember that a node only participates in the physics simulation when it has a physics body assigned to it. Let's set that up for the plunger now.

06-Physics/step05/PhysicsBall/RCWMyScene.m

```
- (void)setUpScene
{
    self.backgroundColor = [SKColor whiteColor];

    self.physicsWorld.gravity = CGVectorMake(0, -3.8);

    SKSpriteNode *ball = [SKSpriteNode spriteNodeWithImageNamed:@"pinball.png"];
    ball.position = CGPointMake(self.size.width/2, self.size.height/2);
    ball.size = CGSizeMake(20, 20);
    [self addChild:ball];

    ball.physicsBody = [SKPhysicsBody bodyWithCircleOfRadius:10];
```

```
    SKSpriteNode *plunger = [SKSpriteNode spriteNodeWithImageNamed:@"plunger.png"];
    plunger.position = CGPointMake(self.size.width/2, self.size.height/2 - 140);
    plunger.size = CGSizeMake(25, 100);
    [self addChild:plunger];

➤   plunger.physicsBody = [SKPhysicsBody bodyWithRectangleOfSize:plunger.size];
}
```

Here we are creating a rectangular physics body of the same size as the plunger and assigning it to the plunger's `physicsBody` property. Rectangular bodies created in this manner are centered on the node's position. Run the game now and watch.

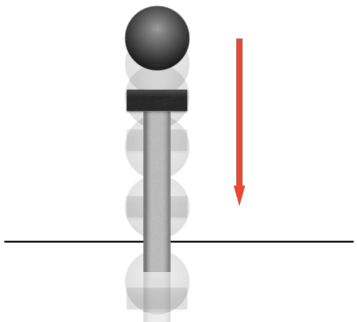


Figure 30—The plunger falls off the screen, too.

This time, *both* the plunger and the ball fall off the screen. Our plunger is now participating in the physics simulation, but it is affected by gravity, which isn't what we want. We need it to be fixed in some way. Many options exist for pinning or fixing physics body in a world, depending on the effect you're going for. For this example, let's just tell the physics body not to be affected by gravity by adding this line:

06-Physics/step06/PhysicsBall/RCWMyScene.m

```
plunger.physicsBody = [SKPhysicsBody bodyWithRectangleOfSize:plunger.size];
➤ plunger.physicsBody.affectedByGravity = NO;
```

Now when you run the game, you'll see the plunger start out stationary, but then the ball will fall and push it off the screen. It's a very strange effect and demonstrates how odd and unnatural physics engines are. We have two bodies, both with density and heft, but only one of them is affected by gravity and has weight.

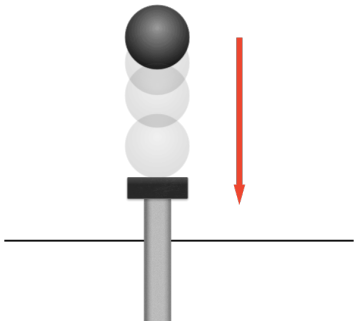


Figure 31—The ball pushes the plunger off the screen.

This isn't what we want. We need the plunger to be completely immovable by the ball. We can do that by telling Sprite Kit that this is not a *dynamic body*. We'll replace the line where we set the affectedByGravityProperty to instead look like this:

06-Physics/step07/PhysicsBall/RCWMyScene.m

```
plunger.physicsBody = [SKPhysicsBody bodyWithRectangleOfSize:plunger.size];
➤ plunger.physicsBody.dynamic = NO;
```

This tells the physics engine that the plunger's physics body should participate in the physics world as something that other bodies can bump into, but it should not be moved. It acts like a permanent fixture, screwed into the tabletop, and only participates as a place for other bodies to bounce off of. Now when you run the game, you'll see the ball fall and bounce on the top of the plunger node.

Debugging Physics Bodies

Debugging physics bodies can take some trial and error. You have to see the interaction to know whether you got it wrong. Thankfully, Sprite Kit offers some extra visual debugging aides. You can tell the SKView to highlight all the physics bodies on the screen by setting a special property in the -viewDidLoad method of the RCWViewController.m file.

```
SKView * skView = (SKView *)self.view;
skView.showsFPS = YES;
skView.showsNodeCount = YES;
➤ skView.showsPhysics = YES;
```

This serves the same purpose as the node count and frame per section display, which we first saw back in *What Just Happened?*, on page ?. By setting the showsPhysics property to YES, you'll see bounding boxes and circles drawn around all the bodies on the screen.

• Click HERE to purchase this book now. discuss