Extracted from:

# Apple Game Frameworks and Technologies

## Build 2D Games with SpriteKit & Swift



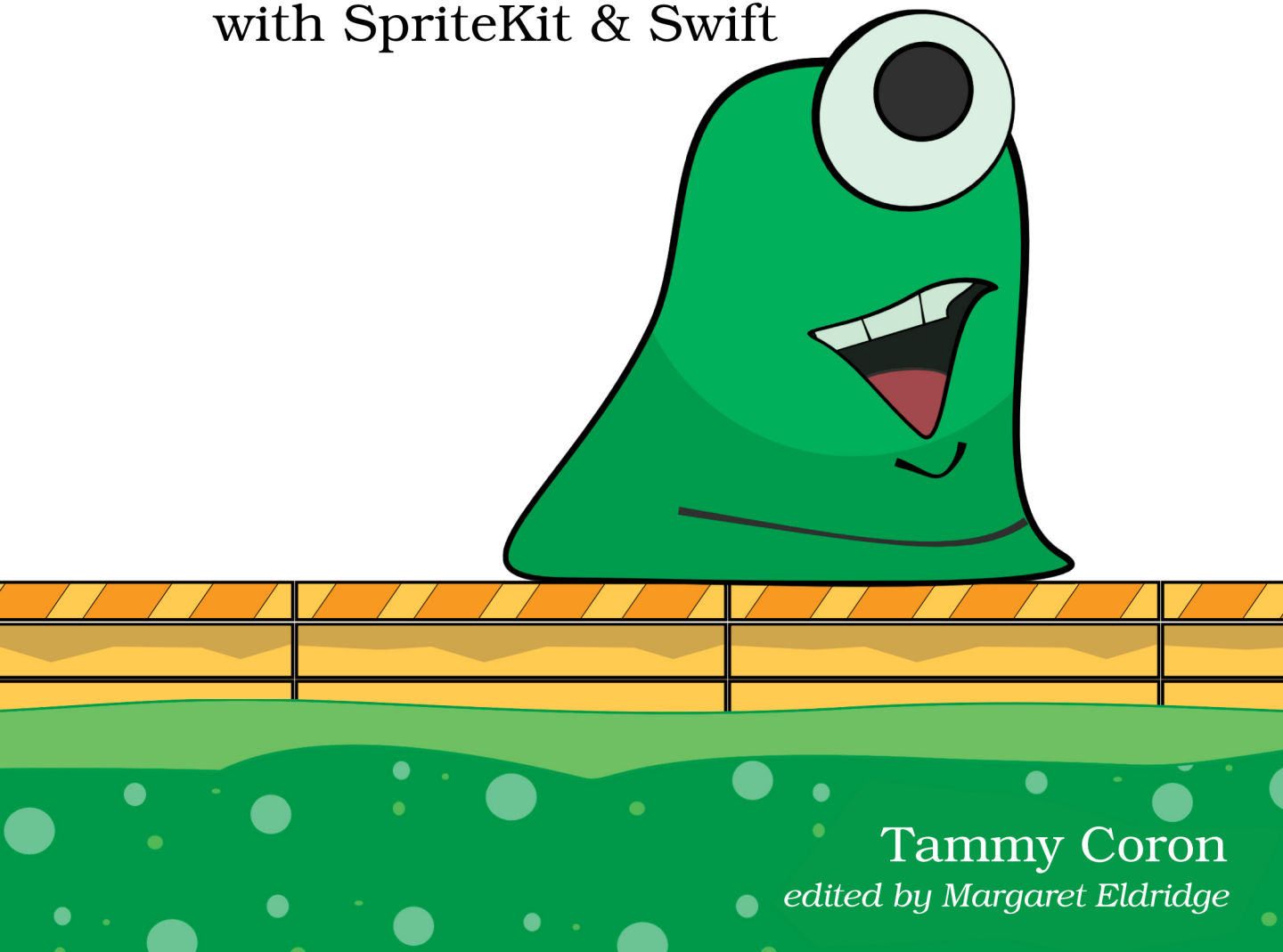
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

# Apple Game Frameworks and Technologies

## Build 2D Games with SpriteKit & Swift



Tammy Coron

*edited by Margaret Eldridge*

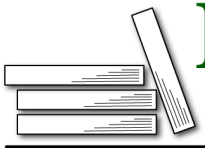
# Apple Game Frameworks and Technologies

## Build 2D Games with SpriteKit & Swift

Tammy Coron

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Margaret Eldridge
Copy Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

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



ISBN-13: 978-1-68050-784-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2021

*To my children, Travis and Jake, and to my husband, Bill: I love you guys so very, very much. Life wouldn't be the same without you.*

CHAPTER 4

# Working with Physics and Collision Detection

In the previous chapter, you added the drop collectible items and learned how to chain actions together so that the gloop drops appear to be dripping from the top of the scene. You also got your feet wet with the iterative and incremental development process by adding your code in smaller chunks and modifying the way the player moves.

In this chapter, you'll get your first look at using the SpriteKit *physics engine*. The SpriteKit physics engine makes it possible for your SpriteKit games to simulate physics and detect collisions.

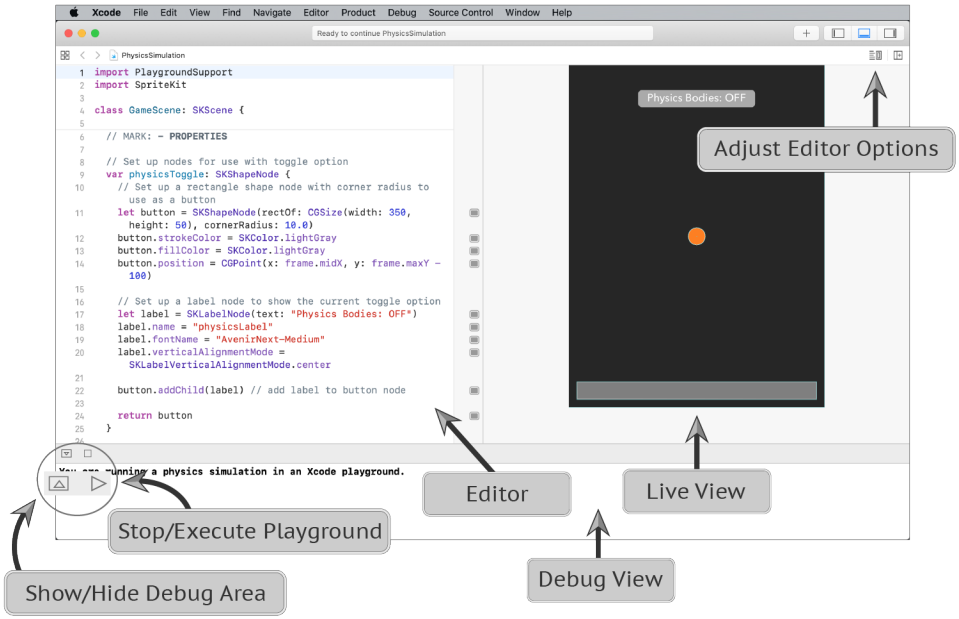
When you attach physical properties to your game objects, they can interact with each other in the game world as they would outside of it. But don't worry; you won't need to be a rocket scientist or astrophysicist to play with the SpriteKit physics engine.

## Run a Playground Physics Simulation

Before jumping back into the Gloop Drop game project, open the PhysicsSimulation.playground file in the projects/playgrounds folder. Playgrounds offer an interactive development environment where you can prototype and test your code in real time. The PhysicsSimulation playground is a small, custom playground that you can use to play around with the SpriteKit physics engine.

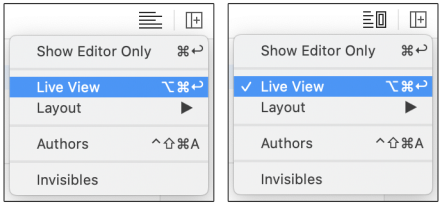
The image on page 8 shows the PhysicsSimulation playground in its running state.

To run the playground, click the Execute button in the bottom-left corner. This action runs the code and loads the SpriteKit scene in the Live View. You may



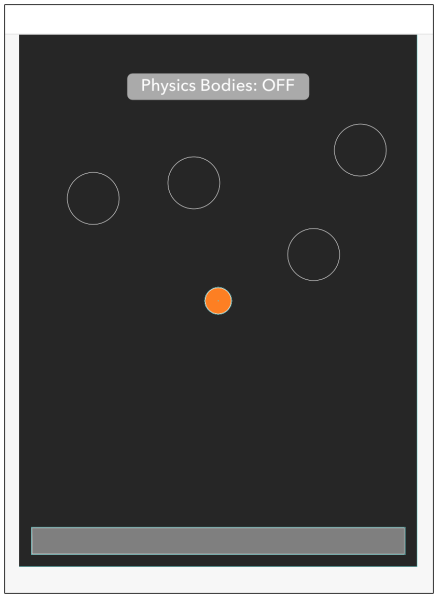

need to start and stop the playground more than once to get it to respond to clicks.

If you don't see the Live View, click the Adjust Editor Options button in the top-right corner and turn on the Live View option. The menu looks like this:




You won't need to write any code in this section, except for a handful of lines. Instead, you'll use the existing code in the PhysicsSimulation playground to learn more about the SpriteKit physics engine. Later in this chapter, you'll apply what you've learned to the gloopdrop project.

With the PhysicsSimulation playground running, click a few random places on the Live View scene, and you'll see something similar to the image on page 9.



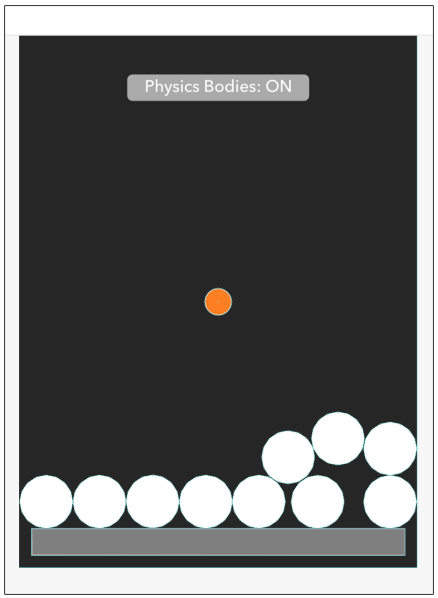

To understand what's happening, look through the code in the playground. You'll find that the touchDown(atPoint:) method contains the following code:

```
// Check if toggle button was clicked
if physicsToggle.contains(pos) {
  withBody.toggle()
  return
}

// Set up a circle shape node to use for the ball
let ball = SKShapeNode(circleOfRadius: 50)
ball.name = "ball"
ball.position = pos

// Check value of `withBody` to determine if physics is enabled
if withBody == true {
  ball.fillColor = SKColor.white
  ball.physicsBody = SKPhysicsBody(circleOfRadius: ball.frame.width/2)

  // Set up physics properties
  ball.physicsBody?.restitution = 0.2 // Bounce: 0.0-1.0 (default: 0.2)
}

// Add the ball node to the scene
addChild(ball)
```

This code, among other things, places a shape node named ball at each touch location (pos). The withBody property is currently set to its default value of false,

so the code skips over coloring the shape white and adding a *physics body* to the node—more on physics bodies in a moment.

Click the "Physics Bodies: OFF" button at the top of the scene to toggle the withBody property to true. Once again, click anywhere on the Live View scene and notice that this time, the balls react to gravity and to each other.




In this second example, where withBody = true, the three lines that were skipped earlier now run:

```
ball.fillColor = SKColor.white
ball.physicsBody = SKPhysicsBody(circleOfRadius: ball.frame.width/2)

// Set up physics properties
ball.physicsBody?.restitution = 0.2 // Bounce: 0.0-1.0 (default: 0.2)
```

The fillColor property—which is not part of the physics engine and is only being used as a visual cue for this example—sets the shape node's color to white. After that, the physicsBody property of that same node gets set using one of the SKPhysicsBody class initializers. The type of initializer you use depends largely on the shape of the physics body you need. In this case, your node is shaped like a circle, so it makes sense to use init(circleOfRadius:).

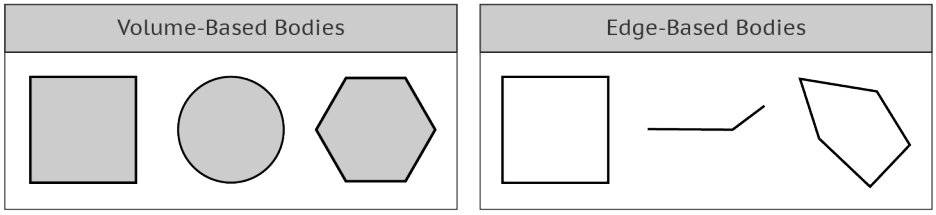
Finally, the restitution property of the physics body attached to the ball node is set to 0.2. (The restitution property determines the node's bounciness, in other words, how much kinetic energy the body loses or gains from collisions.) It's this physics body that makes everything come alive in the SpriteKit physics engine. In fact, change the restitution to 1.0, restart the playground, and you'll

see a big difference in how the balls react when hitting the floor—the higher the number, the more bouncy things get. The restitution property is but one of many properties available for physics bodies.

To better understand the SKPhysicsBody class and how physics bodies work, you'll continue to use the PhysicsSimulation playground in the next section before returning to the gloopdrop project.

## A Closer Look at Physics Bodies

There are two types of physics bodies: *volume-based* bodies and *edge-based* bodies, which are illustrated in the images that follow.




To create these physics bodies, you use the SKPhysicsBody class and one of its initializers. Typically, you'll use an initializer that matches your sprite node's shape, such as a rectangle, circle, polygon, or custom path. You then assign this physics body to the node's physicsBody property, like so:

```
ball.physicsBody = SKPhysicsBody(circleOfRadius: ball.frame.width/2)
```

When defining shapes, always consider performance. A circle is the most efficient shape, whereas a path-based polygon is the least efficient shape. As you add complexity to these custom shapes, the overall performance of your scene decreases. Also, consider that not every object (node) within a scene needs a physics body; you only need to add a physics body if you intend to apply physics to the node or track it for collision detection.

A volume-based body, such as the ball in the previous example, reacts according to its mass, density, and other properties. For example, in the touchDown(atPoint:) method after setting the restitution property, add the following line:

```
ball.physicsBody?.mass = 5.0 // in kilograms
```

This code sets the mass of the ball to 5 kilograms. The default value is based on the size of the physics body and the body's default density, which is 1.0. Because these two values are related, changing one automatically changes the other to be consistent. This number is arbitrary, so as long as you keep things consistent in your physics world, you won't have a problem. In other

words, don't set one body's mass to 5 kilograms and another to 500 kilograms if they're supposed to represent a similarly sized object.

Now, look at the didMove(to:) method. There, you'll see the following three lines that set up the floor node's physics body:

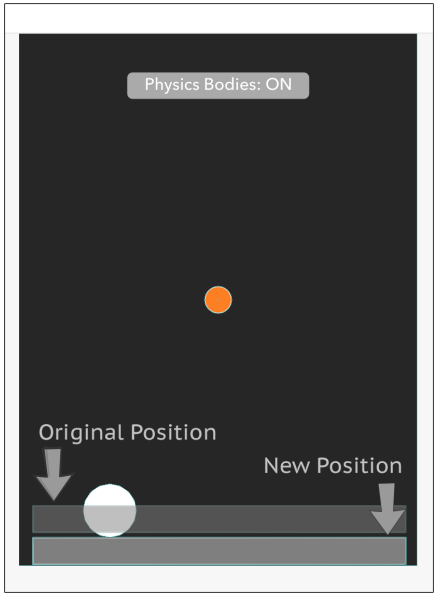
```
// Set up its physics body using a rectangle
floor.physicsBody = SKPhysicsBody(rectangleOf: floor.frame.size)
floor.physicsBody?.isDynamic = false // static, not moved by physics engine
floor.physicsBody?.affectedByGravity = false // ignores world gravity
```

Like the ball node's physics body, this type of physics body is also a volume-based body. However, unlike the ball node, which uses a circle shape, the floor node's initializer uses a rectangle.

Something else to note is two new properties: isDynamic and affectedByGravity.

The isDynamic property for the floor node is set to false. By setting this property to false, you tell the physics engine not to move this body—that's why the floor doesn't move when the balls hit it. This type of volume-based body is known as a *static volume* body because it never moves.

It's time for a little science experiment. Set the isDynamic property of the floor node to true, which is the default value, and run the playground again. Notice that this time the floor falls to the edge of the scene after the first ball hits it. The floor node moves because it's now using a *dynamic volume* body like the balls, so the physics simulation can move it.

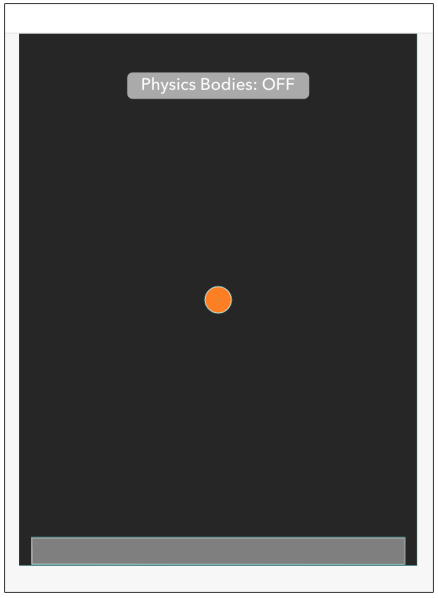


Go back to the touchDown(atPoint:) method and change the mass of the ball node from 5 to 50, like so:

```
ball.physicsBody?.mass = 50
```

Restart the playground and notice how much heavier the balls are this time around—so much so that they make the floor bounce when they hit it.

One more experiment. Currently, the floor node's affectedByGravity property is set to false, which means this physics body ignores gravity. Set this value to true, and restart the playground. Notice this time, as soon as you start the playground, the floor falls to the bottom of the scene, but it doesn't fall out of the scene. Instead, it stops precisely at the bottom or edge of the scene.




The floor node doesn't fall past the scene's edge into nothingness because the scene also has a physics body attached to it. At the end of the didMove(to:) method, notice the following code:

```
// Set up a physics body around the frame using an edge loop
physicsBody = SKPhysicsBody(edgeLoopFrom: frame)
```

This code sets the scene's physicsBody property using what's known as an *edge-based* body. do not have mass and are never moved by the physics simulation, which makes them a great choice for providing invisible boundaries like a scene's edge. If you were to comment out that line and restart the playground, the floor would fall into nothingness. In fact, it would happen so fast that it's unlikely you'd even see it fall.

To finish out this final experiment, you'll adjust the gravity of the simulated physics world. At the top of the didMove(to:) method, add the following code:

```
// Adjust the physics world gravity
physicsWorld.gravity = CGVector(dx: 0, dy: -1.62)

// Earth's gravity: -9.8 meters per second
// Moon's gravity: -1.62 meters per second
```

This code changes the physicsWorld.gravity from its default of -9.8, which simulates Earth's gravity, to -1.62, which simulates the gravity on the Moon. Build and run the project, start dropping balls with physics bodies attached, and you'll notice a significant difference in how the balls react to gravity.

**The Gravity of the Situation**

Gravity is measured by how fast an object falls to the ground. On planet Earth, the acceleration at which an object falls is about 9.8 meters (32 feet) per second. In SpriteKit, you can simulate Earth's gravity by setting the physicsWorld.gravity to (dx: 0, dy: -9.8). Likewise, to simulate the Moon's gravity, you can use (dx: 0, dy: -1.62). The negative numbers in both cases indicate a downward pull, like gravity.

You may have noticed that the gravity property includes both a dx and dy value. Not only can you set the world's vertical gravity (y-axis) by setting a value for dy, but you can also set the horizontal gravity (x-axis) by providing a value for dx, too.

There's no doubt that the SpriteKit physics engine is powerful; with just a few lines of code, you can implement a sophisticated physics world, complete with mass, density, gravity, and more. But the SpriteKit physics engine and physics bodies do more than simulate physics.

In the next section, you'll return to the gloopdrop project and discover how you can use physics bodies for more than simulated physics.