207: Sprite Kit

Part 3: Lab Instructions

207: Sprite Kit
Part 3: Lab Instructions

Finishing Touches

At this point, you have the basic gameplay for Drop Charge complete.

However, there are two major features the game still needs:

1. **Lava**. Lava should rise from the bottom of the game, to represent the ship exploding. This keeps the pressure on the marine to escape quickly!

2. **Game over**. You need to add a way to lose the game, and a "game over" state.

That's what you'll work on in this lab!

Let there be Lava

You'll start by adding the lava. At the top of **GameScene.swift**, add the following property:

```
var lava: LavaNode!
```

This is an implicit optional for an instance of `LavaNode`, a class I created for you to represent lava.

You don't need to understand how this class works for this tutorial – you can just treat it as a node that looks like lava, and automatically moves over time. But if you're curious, feel free to look at the code, and if you have any questions about how it works just shout out.

Next, add this new method right after `setupCoreMotion()`:

```
func setupLava() {
  lava = LavaNode(useEmitter: true)
  lava.position = CGPoint(x: size.width/2, y: -300)
  fgNode.addChild(lava)
}
```

This creates a new `LavaNode`, positions it below the bottom of the screen, and adds it to the foreground node.

If you are running on a simulator, you should change useEmitter to false. Here's why:

- If `useEmitter` is `true`, the `LavaNode` is drawn as a particle system (which looks better). Set `useEmitter` to `true` if you are running on a device.

- If `useEmitter` is `false`, the `LavaNode` is drawn as a shape node (which performs better). Set `useEmitter` to `false` if you are running on a simulator (the simulator doesn't handle large particle systems very well).

Next, call your new method at the end of `didMoveToView()`:

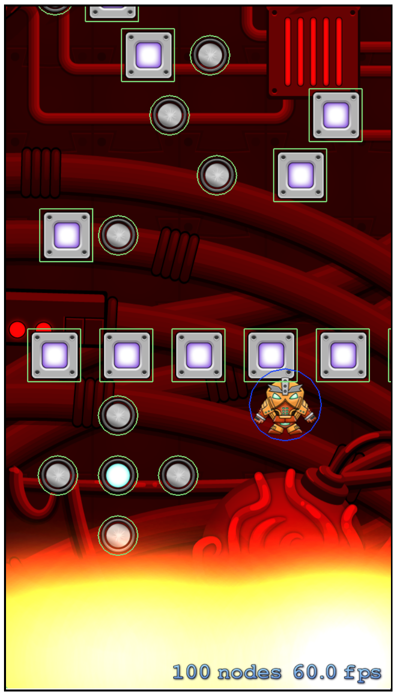
```
setupLava()
```

Finally, inside `update()`, add the following line right after the call to `level.update()`:

```
lava.update(dt)
```

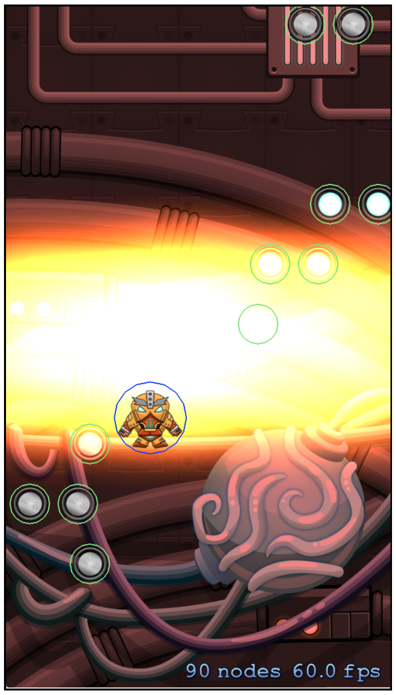
This calls the code that automatically moves the lava over time.

Build and run, and enjoy your lava!



Lava and the Camera

You will notice that sometimes the camera scrolls too far, and you can see underneath the lava, which ruins the illusion:



To fix this, add this line to `updateCamera()`, right before the line `var newPosition = targetPosition`:

```
targetPosition.y = min(targetPosition.y, -(lava.position.y))
```

This makes sure that the camera never scrolls further than the midpoint of the lava.

Build and run, and now everything should look great.

Lava and Collisions

Currently, when you hit the lava you go straight through! When you hit the lava, you should bounce off as if you just got some hot feet. You'll fix that in this section!

To detect when the player hits the lava, you could add a physics body just like you did for the other collisions. But since all you really want to know is "is the player's y position below the lava's y position", it's simpler to just add a line of code to check for that instead.

Add the following method right after `updatePlayer()`:

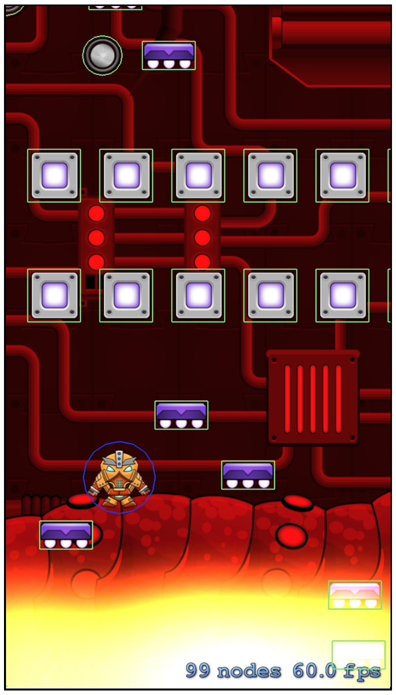
```
func updateCollisionLava() {
  if player.position.y < lava.position.y + 90 {
    boostPlayer()
  }
}
```

This is a simple check – if the player's y position is less than the lava's position + 90 points (representing toward the top of the lava field), then boost the player.

In `update()`, call your new method right after the line `lava.update(dt)`:

```
updateCollisionLava()
```

Build and run, and now when you hit the lava, you'll bounce up again!



Game Over, Man!

Next, let's add a way to lose the game. There is no way to "win" the game (at least in this version), because this game is about how far you can make it before you die.

The space marine will have 3 lives – once he hits the lava 3 times, the game is over.

Start by adding this property to the top of the file:

```
var lives = 3
```

Then add this to `updateCollisionLava()`, right after the call to `boostPlayer()`:

```
lives--
if lives <= 0 {
  switchToGameOver()
}
```

This subtracts a life each time you hit the lava, and if you have no more lives calls `switchToGameOver()`, which you'll write now!

Add this new method right after `switchToPlaying()`:

```
func switchToGameOver() {

  // 1 - Switch game state
  gameState = .GameOver

  // 2 - Turn off physics
  physicsWorld.contactDelegate = nil
  player.physicsBody?.dynamic = false

  // 3 - Bounce player
  let moveUpAction =
    SKAction.moveByX(0, y: size.height/2, duration: 0.5)
  moveUpAction.timingMode = .EaseOut
  let moveDownAction =
    SKAction.moveByX(0, y: -size.height, duration: 1.0)
  moveDownAction.timingMode = .EaseIn
  let sequence = SKAction.sequence([moveUpAction, moveDownAction])
  player.runAction(sequence)

  // 4 - Game Over
  let gameOver = SKSpriteNode(imageNamed: "GameOver")
  gameOver.position = CGPoint(x: size.width/2, y: size.height/2)
  addChild(gameOver)

}
```

Most of this should be review, but let's go over this section by section:

1. This sets the game state to `.GameOver`. This means the `update()` methods will no longer be called (they're only called when the game state is `.Playing`), which is what you want.

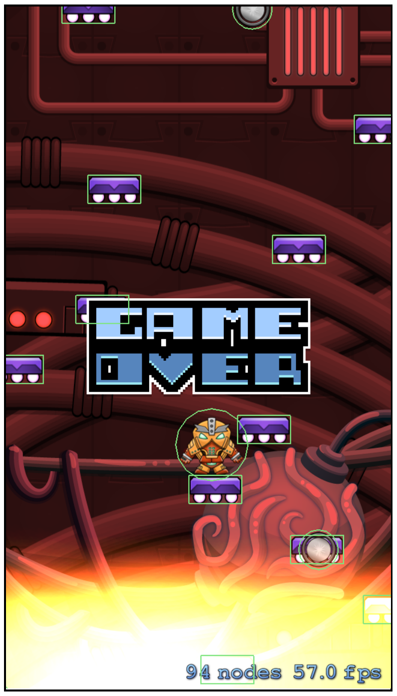
2. This stops the player from being moved by the physics engine by setting it to no longer be dynamic. It also turns off contact notifications by setting `physicsWorld.contactDelegate` to nil.

3. The reason you stopped the player from being moved by the physics engine is because you want to move him yourself. You want to make him bounce Mario-style when he dies – up a bit, and then offscreen.

   To do this, you run a sequence of move actions. Note that you set the timing mode on the actions to Ease Out (start fast, get slower over time) and Ease In (start slow, get faster over time) for a more natural curve. I recommend you use timing modes as often as possible, as they make your actions look much better.

4. Finally, you add a game over image as a direct child of the scene.

Build and run, and make your player die. You'll see the game over scene occur:



Insert Coin – Try Again

When the game is over, let's make it so that if you tap the game restarts.

To do this, add this case inside the switch statement in `touchesBegan()`:

```
case .GameOver:
  // 1
  let newScene = GameScene(size: size)
  // 2
  let reveal = SKTransition.flipHorizontalWithDuration(0.5)
  // 3
  self.view?.presentScene(newScene, transition: reveal)
```

Let's go over this line by line:

1. First you create a new instance of `GameScene` to present. It's often easier to create a brand new instance of your game scene than to have to write code to reset everything properly. :]

2. When you present a game scene, you can pick a transition (i.e. cool aniamation) to use to present the new scene. Here you choose a horizontal flip over 0.5 seconds.

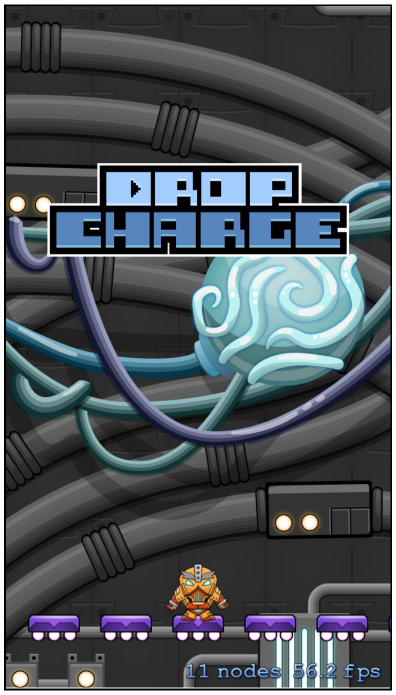
3. To present a new scene, simply call `presentScene()` on the view, passing in the scene and transition to use.

In your own games, you can use this same technique to present different scenes in your game (like a menu scene or another type of level). Create a new subclass of `SKScene` like you did for `GameScene`, and use the above lines of code to transition from one scene to another.

One last thing – now that your game is done, open **GameViewController.swift** and set `showsPhysics` to false to turn off debug drawing:

```
skView.showsPhysics = false
```

Build and run, and now you can play the game as much as you'd like! :]



Congratulations, the core gameplay of Drop Charge is complete! You're ready to continue on to the challenges, where you'll take a good game and make it great – through the power of Juice.