207: Sprite Kit

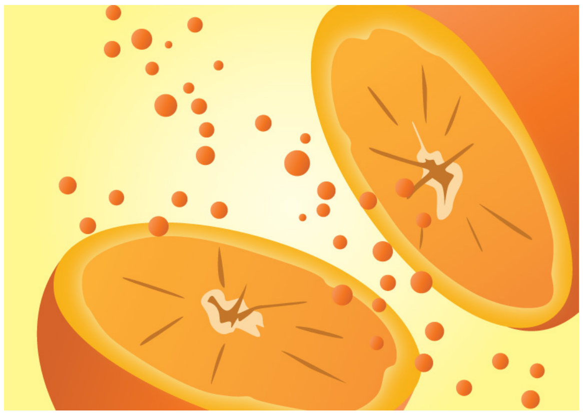
Part 4: Challenge Instructions

207: Sprite Kit
Part 4: Challenge Instructions

Juice it Up!

Do you know how some games feel more polished than others? A lot of it has to do with at technique we game programmers like to call Juice.

Juice is when a game feels connected to you as a player, and gives you immediate feedback upon your actions, often in the form of special effects. The more feedback you can give to your players, the better the game wil feel to them.



We have four challenges for you to experiement with different kinds of juice:

1. **Music and sound effects** to give the player feedback on collisions, game state changes, and other events.
2. **Screen shake and explosions** to give the player feedback on important events and make the game feel more frantic.
3. **Trails and camera lerp** to give the player more feedback on their movement and speed over time.
4. **Animation** to give the player feedback on their motion in another way.

By the time you have finished these challenges, you'll have taken an OK game and made it great!

Challenge A: Music and Sound Effects

The simplest (and often most important) way to add some juice to your game is to add some music and sound effects.

Your project already includes some music and sound effects, along with a handy sound manager to make playing them easy; you just need to use it.

Start by adding this property to the top of **GameScene.swift**:

```
var soundManager: SoundManager!
```

This represents an instance of the `SoundManager` class I created for you. Feel free to look at the class; it's really simple. To play a sound effect in Sprite Kit, you just run an action, and to play a sound effect you can use AVFoundation's `AVAudioPlayer`. The only reason I created this class for you is to save you some typing, it's pretty easy.

Next, add these lines to `setupMusic()`:

```
soundManager = SoundManager(node: self)
soundManager.playMusicBackground()
```

This creates the sound manager and starts the background music playing when the game begins.

Your challenge is to play the rest of the sound effects at the appropriate times. This will be a good review to make sure you understand where each part of the game logic takes place.

Here's what you should play when:

• After you drop the bomb, play the "bomb drop" and "tick tock" sounds.

• After the bomb explodes, play the "action" music and "super boost" sound.

• After the game is over, play the "background" music.

• When the player hits the lava, play the "hit lava" sound.

• When the player gets a normal coin, play the "coin" sound.

• When the player gets a special coin, play the "boost" sound.

• When the player hits a normal platform, play the "jump" sound.

• When the player hits a breakable platform, play the "brick" sound.

Build and run, and you'll see your game feels much better already! :]

Challenge B: Screen Shake and Explosions

Screen shake is also a very easy effect that has a ton of impact. And who doesn't like explosions?! :]

To do this, add this method to the bottom of the file:

```
func screenShakeByAmt(amt: CGFloat) {
  // Set worldNode's position to (0, 0)
  // Remove the action from the world node w/ key "shake"
```

```
    // Create a CGPoint of (0, -amt)
    // Create a "screenShakeWithNode" action:
    //   worldNode, the CGPoint you created, 10 oscs, 2 secs
    // Run the action on the world node w/key "shake"
  }
```

Note that `screenShakeWithNode()` is a custom action in our `SKTUtils` library. If you're curious to learn how to write your own custom actions like this, check out the library.

Your challenge is to write the code for each commented line above.

Then, call this method to perform screen shakes in the following locations:

- `boostPlayer()` – 20
- `superBoostPlayer()` – 100
- When the player hits the lava - 50

Finally, let's add some explosions. Add this property to the top of the class:

```
var explosionManager: ExplosionManager!
```

This will store an `ExplosionManager` I created for you. This class periodically creates an `EmitterNode` (a type of node for particle systems) at random spots to represent explosions.

To create it, add this line at the end of `setupLava()`:

```
explosionManager = ExplosionManager(soundManager: soundManager,
screenShakeByAmt: screenShakeByAmt, parentNode: mgNode)
```

Note it takes the `screenShakeByAmt()` method you just wrote as a parameter; this way it can shake the screen when there's a really big explosion.

Finally, add this line to `update()` right after the call to `updateCollisionLava()`:

```
explosionManager.update(dt)
```

Build and run, and enjoy some screenshakes – I think you'll agree it makes the explosions feel much more impactful!

Challenge C: Trails and Camera Lerp

Adding some particle systems that add "trails" to your player can make it easier to see where your player is moving over time, and give you some hints to player's status (i.e. normal vs. burning), and camera lerp can give the illusion that your player is moving much faster. You'll add that in this section!

Start by adding these methods to the bottom of the file:

```
func removeTrail(trail: SKEmitterNode) {
  // Set trail's numParticlesToEmit to 1 (to stop particle system)
  // Remove particle system from its parent after 1 second
}

func setupTrail(name: String) -> SKEmitterNode {
  // Create a SKEmitterNode with the given filename
  // Set the trail's target to fgNode
  // Add the trail as a child of the player
  // Return the trail
}
```

Your challenge is to implement the commented lines. Then use `setupTrail()` as follows:

- Call it with "PlayerTrail" at the end of `setupPlayer()`
- When the player hits lava, call it with "Smoke Trail", and set up a sequence of actions to wait for 3 seconds, then remove the smoke trail

To lerp the camera, add this code to `updateCamera()`, before the line that sets `fgNode.position`:

```
// Lerp camera
let diff = targetPosition - fgNode.position
let lerpValue = CGFloat(0.05)
let lerpDiff = diff * lerpValue
newPosition = fgNode.position + lerpDiff
```

Your challenge for this part is to be able to explain in 1 sentence what this code does :]

Build and run, and your player movement should feel much more dynamic!

Challenge D: Animation

For your final challenge, you'll add some animation to the player in the game so it's easier to tell whether the player is currently moving up, down, left, or right just by looking at the screen.

Start by adding these properties to the top of the file:

```
var animJump: SKAction! = nil
var animFall: SKAction! = nil
var animSteerLeft: SKAction! = nil
var animSteerRight: SKAction! = nil
var curAnim: SKAction? = nil
```

These will store the animations you'll be creating for the player, along with a reference to the current animation.

Next add these methods to the bottom of the file:

```
func setupAnimWithPrefix(prefix: String, start: Int, end: Int,
timePerFrame: NSTimeInterval) -> SKAction {

  var textures = [SKTexture]()
  for i in start..<end {
    textures.append(SKTexture(imageNamed: "\(prefix)\(i)"))
  }
  return SKAction.animateWithTextures(textures, timePerFrame:
timePerFrame)

}

func runAnim(anim: SKAction) {
  if curAnim == nil || curAnim! != anim {
    player.removeActionForKey("anim")
    player.runAction(anim, withKey: "anim")
    curAnim = anim
  }
}
```

In Sprite Kit, animations are done by actions (surprise, surprise)! `setupAnimWithPrefix()` creates an animation action based on a sequence of numbered textures – you'll call this in a moment to create a series of animations.

`runAnim()` is a helper method to run the given animation (unless it's currently running, in which case it does nothing).

Add these lines to the bottom of `setupPlayer()`:

```
animJump = setupAnimWithPrefix("player01_jump_", start: 1, end: 4,
timePerFrame: 0.1)
animFall = setupAnimWithPrefix("player01_fall_", start: 1, end: 3,
timePerFrame: 0.1)
animSteerLeft = setupAnimWithPrefix("player01_steerleft_", start:
1, end: 2, timePerFrame: 0.1)
animSteerRight = setupAnimWithPrefix("player01_steerright_",
start: 1, end: 2, timePerFrame: 0.1)
```

This uses the helper method you wrote to create each animation and store it in the properties you created earlier.

Your challenge is to write some code at the bottom of `updatePlayer()` that looks at the player's velocity, and runs the appropriate animation. Use the following rules:

- If the player is moving down, play `animFall`.
- If the player is moving more than 100 points per sec to the right, play `animSteerRight`.
- If the player is moving more than 100 points per sec to the left, play `animSteerLeft`.
- Otherwise, play `animJump`.

If you get this part working, congrats – not only do you have great experience with Sprite Kit, but you have experience making a game Juicy as well! :]