FREE eBook

LEARNING sprite-kit

Free unaffiliated eBook created from
Stack Overflow contributors.

#sprite-kit

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sprite-kit

It is an unofficial and free sprite-kit ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official sprite-kit.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with sprite-kit

Remarks

SpriteKit is a 2D Game Engine developed by Apple. It provides high level APIs and a wide range of functionalities to developers. It also contains an internal Physics Engine.

It is available on every Apple platform

- iOS
- macOS
- tvOS
- watchOS (>= 3.0)

Note: If you wish to develop using 3D graphics you need to use SceneKit instead.

The core building blocks of SpriteKit are:

- SKView: a view in which SKScenes are presented.
- SKScene: a 2D scene that is presented in an SKView and contains one or more SKSpriteNodes.
- SKSpriteNode: an individual 2D image that can be animated around the scene.

Other related building blocks are:

- SKNode: a more general node that can be used in a scene to group other nodes together for more complex behaviour.
- SKAction: single or groups of actions that are applied to SKNodes to implement animations and other effects.
- SKPhysicsBody - allows physics to be applied to SKNodes to allow them to behave in a realistic manner, including falling under gravity, bouncing off each other and following ballistic trajectories.

Official documentation.

Versions

iOS 7.0 and Later

OS X 10.9 Mavericks and Later

watchOS 3.0 and Later

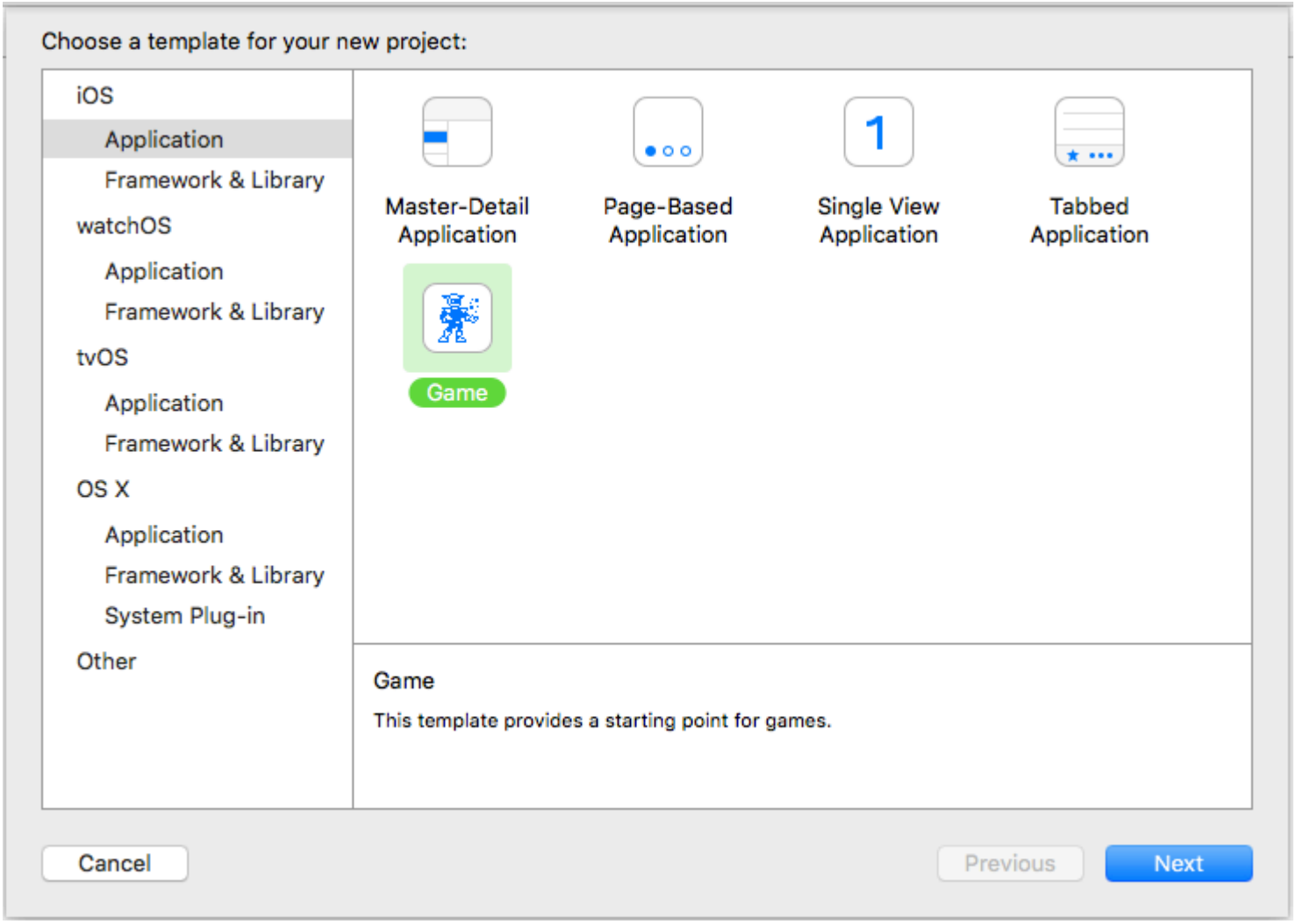
tvOS 9.0 and later

Examples

Your first SpriteKit Game (Hello World)

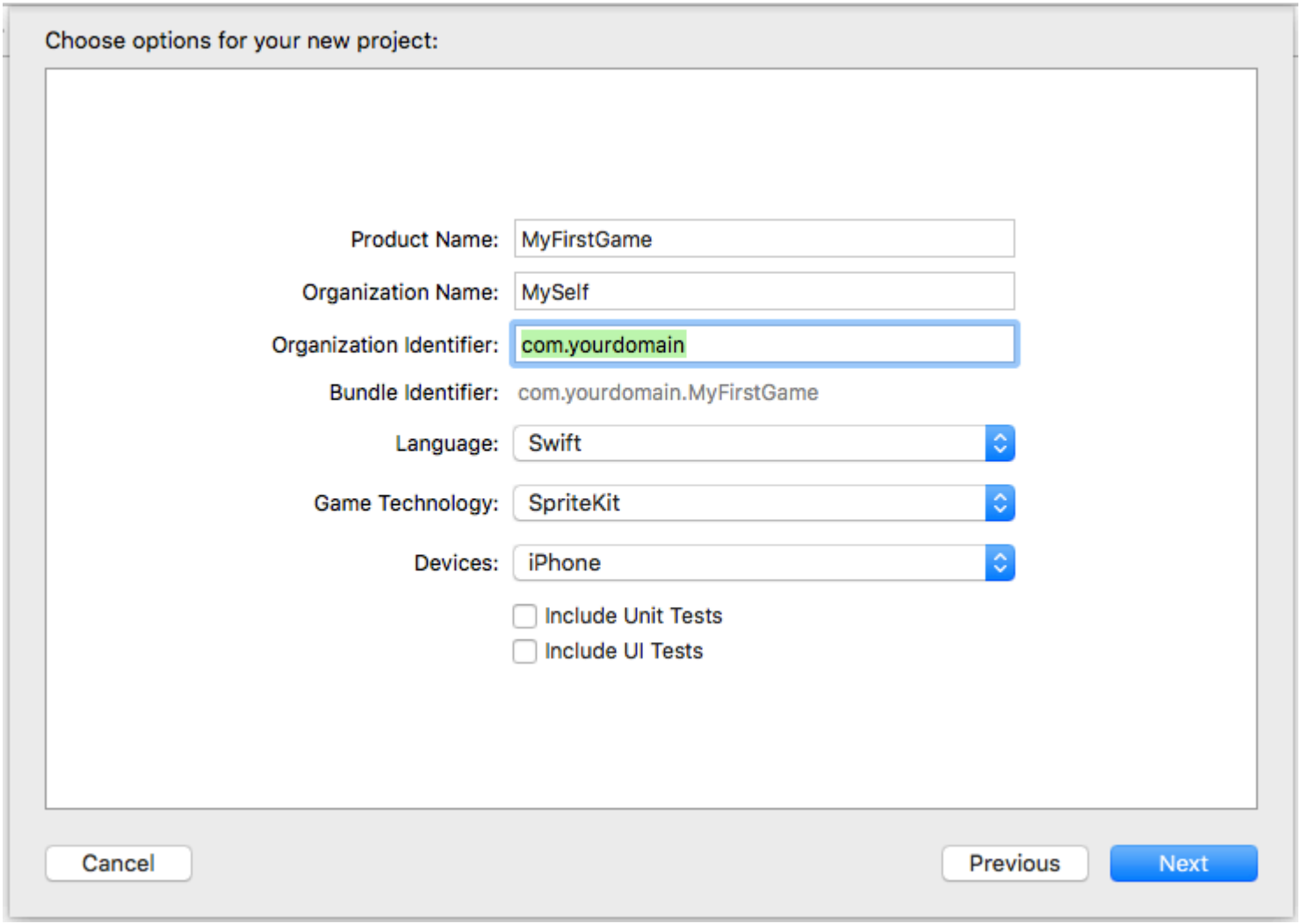
Open Xcode and select `Create a new Xcode Project`.

Now select `iOS > Application` on the left and `Game` on the main selection area.



Press **Next**.

- Write into `Product Name` the name of your first great game.
- Into `Organization Name` the name of your company (or simply your own name).
- `Organisation Identifier` should contain your *reversed* domain name (`www.yourdomain.com` becomes `com.yourdomain`). If you don't have a domain write anything you want (this is just and test).
- Then select `Swift`, `SpriteKit` and `iPhone`.



Press **Next**.

Select a folder of your Mac where you want to save the project and click on **Create**.

Congrats, you create your first Game with SpriteKit! Just press `CMD` + `R` to run it into the simulator!

Read Getting started with sprite-kit online: https://riptutorial.com/sprite-kit/topic/2956/getting-started-with-sprite-kit

Chapter 2: Detecting touch input on iOS devices

Examples

Detecting touch

You can override 4 methods of `SKScene` to detect user touch

```
class GameScene: SKScene {


    override func touchesBegan(touches: Set<UITouch>, withEvent event: UIEvent?) {

    }

    override func touchesMoved(touches: Set<UITouch>, withEvent event: UIEvent?) {

    }

    override func touchesEnded(touches: Set<UITouch>, withEvent event: UIEvent?) {

    }

    override func touchesCancelled(touches: Set<UITouch>?, withEvent event: UIEvent?) {

    }
}
```

> Please note that each method receives a `touches` parameter which (under particular circumstances) can contain more then one single touch event.

Read Detecting touch input on iOS devices online: https://riptutorial.com/sprite-kit/topic/3660/detecting-touch-input-on-ios-devices

Chapter 3: Physics

Examples

How to correctly remove node in didBeginContact method (multiple contacts)

```
// PHYSICS CONSTANTS
struct PhysicsCategory {
    static let None       : UInt32 = 0
    static let All        : UInt32 = UInt32.max
    static let player     : UInt32 = 0b1             // 1
    static let bullet     : UInt32 = 0b10            // 2
}

var nodesToRemove = [SKNode]()

    // #-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
    //MARK: - Physic Contact Delegate methods
    // #-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#

    func didBegin(_ contact: SKPhysicsContact) {
        var one: SKPhysicsBody
        var two: SKPhysicsBody

        if contact.bodyA.categoryBitMask < contact.bodyB.categoryBitMask {
            one = contact.bodyA
            two = contact.bodyB
        } else {
            one = contact.bodyB
            two = contact.bodyA
        }

        // PLAYER AND BULLET
        if one.categoryBitMask == PhysicsCategory.player && two.categoryBitMask ==
PhysicsCategory.bullet {
           nodesToRemove.append(one.node!) // remove player
           nodesToRemove.append(two.node!) // remove bullet
        }
    }
    override func didFinishUpdate()
    {
        nodesToRemove.forEach(){$0.removeFromParent()}
        nodesToRemove = [SKNode]()
    }
}
```

Read Physics online: https://riptutorial.com/sprite-kit/topic/8991/physics

Chapter 4: SKAction

Examples

Create and Run a Simple SKAction

A very simple example would be to fade out an SKSpriteNode.

In Swift:

```
let node = SKSpriteNode(imageNamed: "image")
let action = SKAction.fadeOutWithDuration(1.0)
node.runAction(action)
```

Creating a Repeating Sequence of Actions

Sometimes it is necessary to do an action on repeat or in a sequence. This example will make the node fade in and out a total of 3 times.

In Swift:

```
let node = SKSpriteNode(imageNamed: "image")
let actionFadeOut = SKAction.fadeOutWithDuration(1.0)
let actionFadeIn = SKAction.fadeInWithDuration(1.0)
let actionSequence = SKAction.sequence([actionFadeOut, actionFadeIn])
let actionRepeat = SKAction.repeatAction(actionSequence, count: 3)
node.runAction(actionRepeat)
```

Running a Block of Code in an SKAction

One helpful case is to have the action run a block of code.

In Swift:

```
let node = SKSpriteNode(imageNamed: "image")
let actionBlock = SKAction.runBlock({
    //Do what you want here
    if let gameScene = node.scene as? GameScene {
        gameScene.score += 5
    }
})
node.runAction(actionBlock)
```

Named actions that can be started or removed from elsewhere.

Sometimes you would want to start or remove an action on a specific node at a certain time. For example, you might want to stop a moving object when the user taps the screen. This becomes very helpful when a node has multiple actions and you only wants to access one of them.

```
let move = SKAction.moveTo(x: 200, duration: 2)
object.run(move, withKey: "moveX")
```

Here we set the key "moveX" for the action `move` in order to access it later in another part of the class.

```
  override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
        object.removeAction(forKey: "moveX")
    }
```

When the user touches the screen the action will get removed and the object will stop moving.

Read SKAction online: https://riptutorial.com/sprite-kit/topic/6229/skaction

Chapter 5: SKNode Collision

Remarks

The determinants of Sprite Kit collision and contact event processing are the relationship settings, created by you, of `categoryBitMask`, `collisionBitMask` and `contactTestBitMask` for each of your interacting object types. By rationally setting these in service of your desired outcomes from contacts and collisions, you determine which types can collide and inform of contacts with others, and avoid undesired collision, contact and physics processing overhead.

For each type of 'entity' you can set all three:

1. `categoryBitMask` : a category specific to this type of node
2. `collisionBitMask` : a collision differentiator, can be different from above
3. `contactTestBitMask` : a contact differentiator, can be different from both above

The general steps to implement collisions & contacts are:

1. set physic body size, shape and (sometimes) mass
2. add necessary BitMasks for your node type from category, collision and contact above
3. set scene as a contact delegate enabling it to check and inform of collisions and contacts
4. implement contact handlers and any other pertinent logic for physics events

Examples

Enable Physics World

```
// World physics
    self.physicsWorld.gravity         = CGVectorMake(0, -9.8);
```

Enable Node to Collide

Firstly, we set node category

```
let groundBody: UInt32 = 0x1 << 0
let boxBody: UInt32 = 0x1 << 1
```

Then add Ground type node and Box type node.

```
let ground = SKSpriteNode(color: UIColor.cyanColor(), size: CGSizeMake(self.frame.width, 50))
ground.position = CGPointMake(CGRectGetMidX(self.frame), 100)
ground.physicsBody = SKPhysicsBody(rectangleOfSize: ground.size)
ground.physicsBody?.dynamic = false
ground.physicsBody?.categoryBitMask = groundBody
ground.physicsBody?.collisionBitMask = boxBody
ground.physicsBody?.contactTestBitMask = boxBody
```

```
addChild(ground)

// Add box type node

let box = SKSpriteNode(color: UIColor.yellowColor(), size: CGSizeMake(20, 20))
box.position = location
box.physicsBody = SKPhysicsBody(rectangleOfSize: box.size)
box.physicsBody?.dynamic = true
box.physicsBody?.categoryBitMask = boxBody
box.physicsBody?.collisionBitMask = groundBody | boxBody
box.physicsBody?.contactTestBitMask = boxBody
box.name = boxId

let action = SKAction.rotateByAngle(CGFloat(M_PI), duration:1)

box.runAction(SKAction.repeatActionForever(action))

self.addChild(box)
```

Handle Contacts

Set scene as delegate

```
//set your scene as SKPhysicsContactDelegate

class yourScene: SKScene, SKPhysicsContactDelegate

self.physicsWorld.contactDelegate = self;
```

Then you have to implement one or the other of the contact functions: optional func didBegin(contact:) and/or optional fund didEnd(contact:) method to fill in your contact logic e.g. like

```
//order

let bodies = (contact.bodyA.categoryBitMask <= contact.bodyB.categoryBitMask) ?
(A:contact.bodyA,B:contact.bodyB) : (A:contact.bodyB,B:contact.bodyA)


//real handler
if ((bodies.B.categoryBitMask & boxBody) == boxBody){
   if ((bodies.A.categoryBitMask & groundBody) == groundBody) {
       let vector = bodies.B.velocity
       bodies.B.velocity = CGVectorMake(vector.dx, vector.dy * 4)

   }else{
       let vector = bodies.A.velocity
       bodies.A.velocity = CGVectorMake(vector.dx, vector.dy * 10)

   }
}
```

Alternative didBeginContact

IF you are using simple categories, with each physics body belonging to only one category, then this alternative form of didBeginContact may be more readable:

```
func didBeginContact(contact: SKPhysicsContact) {
    let contactMask = contact.bodyA.categoryBitMask | contact.bodyB.categoryBitMask

switch contactMask {

case categoryBitMask.player | categoryBitMask.enemy:
    print("Collision between player and enemy")
    let enemyNode = contact.bodyA.categoryBitMask == categoryBitMask.enemy ?
contact.bodyA.node! : contact.bodyB.node!
    enemyNode.explode()
    score += 10

case categoryBitMask.enemy | categoryBitMask.enemy:
    print("Collision between enemy and enemy")
    contact.bodyA.node.explode()
    contact.bodyB.node.explode()

default :
    //Some other contact has occurred
    print("Some other contact")
}
}
```

Simple Sprite Kit project showing collisions, contacts & touch events.

Here is a simple Sprite-Kit GameScene.swift. Create a new, empty SpriteKit project and replace the GameScene.swift with this. Then build and run.

Click on any of the objects on screen to give make them move. Check the logs and the comments to see which ones collide and which ones make contact.

```
//
//  GameScene.swift
//  bounceTest
//
//  Created by Stephen Ives on 05/04/2016.
//  Copyright (c) 2016 Stephen Ives. All rights reserved.
//

import SpriteKit



class GameScene: SKScene, SKPhysicsContactDelegate {

    let objectSize = 150
    let initialImpulse: UInt32 = 300  // Needs to be proportional to objectSize

    //Physics categories
    let purpleSquareCategory:   UInt32 = 1 << 0
    let redCircleCategory:      UInt32 = 1 << 1
    let blueSquareCategory:     UInt32 = 1 << 2
    let edgeCategory:           UInt32 = 1 << 31

    let purpleSquare = SKSpriteNode()
    let blueSquare = SKSpriteNode()
    let redCircle = SKSpriteNode()
```

```
    override func didMove(to view: SKView) {

        physicsWorld.gravity = CGVector(dx: 0, dy: 0)

        //Create an boundary else everything will fly off-screen
        let edge = frame.insetBy(dx: 0, dy: 0)
        physicsBody = SKPhysicsBody(edgeLoopFrom: edge)
        physicsBody?.isDynamic = false  //This won't move
        name = "Screen_edge"

        scene?.backgroundColor = SKColor.black

        //        Give our 3 objects their attributes

        blueSquare.color = SKColor.blue
        blueSquare.size = CGSize(width: objectSize, height: objectSize)
        blueSquare.name = "shape_blueSquare"
        blueSquare.position = CGPoint(x: size.width * -0.25, y: size.height * 0.2)

        let circleShape = SKShapeNode(circleOfRadius: CGFloat(objectSize))
        circleShape.fillColor = SKColor.red
        redCircle.texture = view.texture(from: circleShape)
        redCircle.size = CGSize(width: objectSize, height: objectSize)
        redCircle.name = "shape_redCircle"
        redCircle.position = CGPoint(x: size.width * 0.4, y: size.height * -0.4)

        purpleSquare.color = SKColor.purple
        purpleSquare.size = CGSize(width: objectSize, height: objectSize)
        purpleSquare.name = "shape_purpleSquare"
        purpleSquare.position = CGPoint(x: size.width * -0.35, y: size.height * 0.4)

        addChild(blueSquare)
        addChild(redCircle)
        addChild(purpleSquare)

        redCircle.physicsBody = SKPhysicsBody(circleOfRadius: redCircle.size.width/2)
        blueSquare.physicsBody = SKPhysicsBody(rectangleOf: blueSquare.frame.size)
        purpleSquare.physicsBody = SKPhysicsBody(rectangleOf: purpleSquare.frame.size)

        setUpCollisions()

        checkPhysics()

    }


    func setUpCollisions() {

        //Assign our category bit masks to our physics bodies
        purpleSquare.physicsBody?.categoryBitMask = purpleSquareCategory
        redCircle.physicsBody?.categoryBitMask = redCircleCategory
        blueSquare.physicsBody?.categoryBitMask = blueSquareCategory
        physicsBody?.categoryBitMask = edgeCategory  // This is the edge for the scene itself

        // Set up the collisions. By default, everything collides with everything.

        redCircle.physicsBody?.collisionBitMask &= ~purpleSquareCategory  // Circle doesn't
collide with purple square
        purpleSquare.physicsBody?.collisionBitMask = 0   // purpleSquare collides with nothing
        //        purpleSquare.physicsBody?.collisionBitMask |= (redCircleCategory |
blueSquareCategory)  // Add collisions with red circle and blue square
```

```
        purpleSquare.physicsBody?.collisionBitMask = (redCircleCategory)  // Add collisions
with red circle
        blueSquare.physicsBody?.collisionBitMask = (redCircleCategory)  // Add collisions with
red circle


        // Set up the contact notifications. By default, nothing contacts anything.
        redCircle.physicsBody?.contactTestBitMask |= purpleSquareCategory   // Notify when red
circle and purple square contact
        blueSquare.physicsBody?.contactTestBitMask |= redCircleCategory     // Notify when
blue square and red circle contact

        // Make sure everything collides with the screen edge and make everything really
'bouncy'
        enumerateChildNodes(withName: "//shape*") { node, _ in
            node.physicsBody?.collisionBitMask |= self.edgeCategory  //Add edgeCategory to the
collision bit mask
            node.physicsBody?.restitution = 0.9 // Nice and bouncy...
            node.physicsBody?.linearDamping = 0.1 // Nice and bouncy...
        }

        //Lastly, set ourselves as the contact delegate
        physicsWorld.contactDelegate = self
    }

    func didBegin(_ contact: SKPhysicsContact) {
        let contactMask = contact.bodyA.categoryBitMask | contact.bodyB.categoryBitMask

        switch contactMask {
        case purpleSquareCategory | blueSquareCategory:
            print("Purple square and Blue square have touched")
        case redCircleCategory | blueSquareCategory:
            print("Red circle and Blue square have touched")
        case redCircleCategory | purpleSquareCategory:
            print("Red circle and purple Square have touched")
        default: print("Unknown contact detected")
        }
    }

    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {

        for touch in touches {
            let touchedNode = selectNodeForTouch(touch.location(in: self))

            if let node = touchedNode {
                node.physicsBody?.applyImpulse(CGVector(dx:
CGFloat(arc4random_uniform(initialImpulse)) - CGFloat(initialImpulse/2), dy:
CGFloat(arc4random_uniform(initialImpulse)) - CGFloat(initialImpulse/2)))
                node.physicsBody?.applyTorque(CGFloat(arc4random_uniform(20)) - CGFloat(10))
            }

        }
    }

    // Return the sprite where the user touched the screen
    func selectNodeForTouch(_ touchLocation: CGPoint) -> SKSpriteNode? {

        let touchedNode = self.atPoint(touchLocation)
        print("Touched node is \(touchedNode.name)")
        //        let touchedColor = getPixelColorAtPoint(touchLocation)
        //        print("Touched colour is \(touchedColor)")
```

```
        if touchedNode is SKSpriteNode {
            return (touchedNode as! SKSpriteNode)
        } else {
            return nil
        }
    }

    //MARK: - Analyse the collision/contact set up.
    func checkPhysics() {

        // Create an array of all the nodes with physicsBodies
        var physicsNodes = [SKNode]()

        //Get all physics bodies
        enumerateChildNodes(withName: "//.") { node, _ in
            if let _ = node.physicsBody {
                physicsNodes.append(node)
            } else {
                print("\(node.name) does not have a physics body so cannot collide or be
involved in contacts.")
            }
        }

        //For each node, check it's category against every other node's collion and contctTest
bit mask
        for node in physicsNodes {
            let category = node.physicsBody!.categoryBitMask
            // Identify the node by its category if the name is blank
            let name = node.name != nil ? node.name! : "Category \(category)"

            let collisionMask = node.physicsBody!.collisionBitMask
            let contactMask = node.physicsBody!.contactTestBitMask

            // If all bits of the collisonmask set, just say it collides with everything.
            if collisionMask == UInt32.max {
                print("\(name) collides with everything")
            }

            for otherNode in physicsNodes {
            if (node.physicsBody?.dynamic == false) {
                print("This node \(name) is not dynamic")
            }
                if (node != otherNode) && (node.physicsBody?.isDynamic == true) {
                    let otherCategory = otherNode.physicsBody!.categoryBitMask
                    // Identify the node by its category if the name is blank
                    let otherName = otherNode.name != nil ? otherNode.name! : "Category
\(otherCategory)"

                    // If the collisonmask and category match, they will collide
                    if ((collisionMask & otherCategory) != 0) && (collisionMask != UInt32.max)
{
                        print("\(name) collides with \(otherName)")
                    }
                    // If the contactMAsk and category match, they will contact
                    if (contactMask & otherCategory) != 0 {print("\(name) notifies when
contacting \(otherName)")}
                }
            }
        }
    }
```

```
}
```

Alternative to Handling contact when dealing with multi category sprites

```
let bodies = (contact.bodyA.categoryBitMask <= contact.bodyB.categoryBitMask) ?
(A:contact.bodyA,B:contact.bodyB) : (A:contact.bodyB,B:contact.bodyA)


switch (bodies.A.categoryBitMask,bodies.B.categoryBitMask)
{
  case let (a, _) where (a && superPower): //All we care about is if category a has a super
power
        //do super power effect
        fallthrough //continue onto check if we hit anything else
  case let (_, b) where (b && superPower): //All we care about is if category b has a super
power
        //do super power effect
        fallthrough //continue onto check if we hit anything else
  case let (a, b) where  (a && groundBody) && (b && boxBody): //Check if box hit ground
       //boxBody hit ground
  case let (b, _) where  (b && boxBody): //Check if box hit anything else
       //box body hit anything else
   default:()

}
```

Difference between contacts and collisions

In Sprite-Kit, there is the concept of **collisions** which refers to the SK physics engine handling how physics objects interact when they collide i.e. which ones bounce off which other ones.

It also has the concept of **contacts**, which is the mechanism by which your program gets informed when 2 physics objects intersect.

Objects may collide but not generate contacts, generate contacts without colliding, or collide and generate a contact (or do neither and not interact at all)

Collisions can also be one-sided i.e. object A can collide (bounce off) object B, whilst object B carries on as though nothing had happened. If you want 2 object to bounce off each other, they must both be told to collide with the other.

Contacts however are not one-sided; if you want to know when object A touched (contacted) object B, it is enough to set up contact detection on object A with regards to object B. You do not have to set up contact detection on object B for object A.

Manipulating contactTest and collison bitmasks to enable/disable specific contact and collisions.

For this example, we will used 4 bodies and will show only the last 8 bits of the bit masks for simplicity. The 4 bodies are 3 SKSpriteNodes, each with a physics body and a boundary:

```
    let edge = frame.insetBy(dx: 0, dy: 0)
    physicsBody = SKPhysicsBody(edgeLoopFrom: edge)
```

Note that the 'edge' physics body is the physics body of the scene, not a node.

We define 4 unique categories

```
let purpleSquareCategory:   UInt32 = 1 << 0  // bitmask is ...00000001
let redCircleCategory:      UInt32 = 1 << 1  // bitmask is ...00000010
let blueSquareCategory:     UInt32 = 1 << 2  // bitmask is ...00000100
let edgeCategory:           UInt32 = 1 << 31  // bitmask is 10000...00000000
```

Each physics body is assigned the categories that it belongs to:

```
        //Assign our category bit masks to our physics bodies
        purpleSquare.physicsBody?.categoryBitMask = purpleSquareCategory
        redCircle.physicsBody?.categoryBitMask = redCircleCategory
        blueSquare.physicsBody?.categoryBitMask = blueSquareCategory
        physicsBody?.categoryBitMask = edgeCategory  // This is the edge for the scene itself
```

If a bit in a body's collisionBitMask is set to 1, then it collides (bounces off) any body that has a '1' in the same position in it's categoryBitMask. Similarly for contactTestBitMask.

Unless you specify otherwise, everything collides with everything else and no contacts are generated (your code won't be notified when anything contacts anything else):

```
purpleSquare.physicsBody.collisonBitMask = 11111111111111111111111111111111 // 32 '1's.
```

Every bit in every position is '1', so when compared to any other categoryBitMask, Sprite Kit will find a '1' so a collision will occur. If you do not want this body to collide with a certain category, you will have to set the correct bit in the collisonBitMask to '0'

and its `contactTestbitMask` is set to all `0`s:

```
redCircle.physicsBody.contactTestBitMask = 00000000000000000000000000000000  // 32 '0's
```

Same as for collisionBitMask, except reversed.

Contacts or collisions between bodies can be turned **off** (leaving existing contact or collision unchanged) using:

```
nodeA.physicsBody?.collisionBitMask &= ~nodeB.category
```

We logically AND nodeA's collision bit mask with the inverse (logical NOT, the ~ operator) of nodeB's category bitmask to 'turn off' that bit nodeA's bitMask. e.g to stop the red circle from colliding with the purple square:

```
redCircle.physicsBody?.collisionBitMask = redCircle.physicsBody?.collisionBitMask &
~purpleSquareCategory
```

which can be shortened to:

```
redCircle.physicsBody?.collisionBitMask &= ~purpleSquareCategory
```

Explanation:

```
redCircle.physicsBody.collisonBitMask = 11111111111111111111111111111111
purpleSquareCategory  = 00000000000000000000000000000001
~purpleSquareCategory = 11111111111111111111111111111110
11111111111111111111111111111111 & 11111111111111111111111111111110 =
11111111111111111111111111111110
redCircle.physicsBody.collisonBitMask now equals 11111111111111111111111111111110
```

redCircle no longer collides with bodies with a category of ....0001 (purpleSquare)

Instead of turning off individual bits in the collsionsbitMask, you can set it directly:

```
blueSquare.physicsBody?.collisionBitMask = (redCircleCategory | purpleSquareCategory)
```

i.e. `blueSquare.physicsBody?.collisionBitMask = (....00000010 OR ....00000001)`

which equals `blueSquare.physicsBody?.collisionBitMask = ....00000011`

blueSquare will only collide with bodies with a category or ..01 or ..10

Contacts or collisions between 2 bodies can be turned **ON** (without affecting any existing contacts or collisions) at any point using:

```
redCircle.physicsBody?.contactTestBitMask |= purpleSquareCategory
```

We logically AND redCircle's bitMask with purpleSquare's category bitmask to 'turn on' that bit in redcircle's bitMask. This leaves any other bits in redCircel's bitMas unaffected.

You can make sure that every shape 'bounces off' a screen edge as follows:

```
// Make sure everything collides with the screen edge
enumerateChildNodes(withName: "//*") { node, _ in
    node.physicsBody?.collisionBitMask |= self.edgeCategory  //Add edgeCategory to the
collision bit mask
}
```

Note:

Collisions can be one-sided i.e. object A can collide (bounce off) object B, whilst object B carries on as though nothing had happened. If you want 2 object to bounce off each other, they must both be told to collide with the other:

```
blueSquare.physicsBody?.collisionBitMask = redCircleCategory
redcircle.physicsBody?.collisionBitMask = blueSquareCategory
```

Contacts however are not one-sided; if you want to know when object A touched (contacted) object B, it is enough to set up contact detection on object A with regards to object B. You do not have to set up contact detection on object B for object A.

```
blueSquare.physicsBody?.contactTestBitMask = redCircleCategory
```

We don't need `redcircle.physicsBody?.contactTestBitMask= blueSquareCategory`

Advanced usage:

Not covered here, but physics bodies can belong to more than one category. E.g. we could set our game up as follows:

```
let squareCategory:   UInt32 = 1 << 0   // bitmask is ...00000001
let circleCategory:   UInt32 = 1 << 1   // bitmask is ...00000010
let blueCategory:     UInt32 = 1 << 2   // bitmask is ...00000100
let redCategory:      UInt32 = 1 << 3   // bitmask is ...00001000
let purpleCategory:   UInt32 = 1 << 4   // bitmask is ...00010000
let edgeCategory:     UInt32 = 1 << 31  // bitmask is 10000...0000000
```

Each physics body is assigned the categories that it belongs to:

```
        //Assign our category bit masks to our physics bodies
        purpleSquare.physicsBody?.categoryBitMask = squareCategory | purpleCategory
        redCircle.physicsBody?.categoryBitMask = circleCategory | redCategory
        blueSquare.physicsBody?.categoryBitMask = squareCategory | blueCategory
```

their categorybitMasks are now:

```
purpleSquare.physicsBody?.categoryBitMask = ...00010001
redCircle.physicsBody?.categoryBitMask    = ...00001010
blueSquare.physicsBody?.categoryBitMask   = ...00000101
```

This will affect how you manipulate the bit fields. It can be useful (for example) to indicate that a physics body (e.g. a bomb) has changed somehow (e.g. it might have gained the 'super' ability which is another category, and you might check that a certain object (an alien mothersh

Read SKNode Collision online: https://riptutorial.com/sprite-kit/topic/6261/sknode-collision

Chapter 6: SKScene

Remarks

SKScene represents a single scene in a SpriteKit application. An SKScene is 'presented' into an SKView. SKSpriteNodes are added to the scene to implement the actual sprites.

Simple applications may have a single SKScene that contains all the SpriteKit content. More complex apps may have several SKScenes that are presented at different times (e.g. an opening scene to present the game options, a second scene to implement the game itself and a third scene to present the 'Game Over' results).

Examples

Subclassing SKScene to Implement Primary SpriteKit Functionality

SpriteKit functionality can be implemented in a subclass of SKScene. For example, a game may implement the main game functionality within an SKScene subclass called GameScene.

In Swift:

```
import SpriteKit

class GameScene: SKScene {

    override func didMoveToView(view: SKView) {
        /* Code here to setup the scene when it is first shown. E.g. add sprites. */
    }

    override func touchesBegan(touches: Set<UITouch>, withEvent event: UIEvent?) {
        for touch in touches {
            let location = touch.locationInNode(self)
            /* Code here to respond to a user touch in the scene at location */
        }
    }

    override func update(currentTime: CFTimeInterval) {
        /* Code here to perform operations before each frame is updated */
    }
}
```

Secondary functionality could then be implemented in subclasses of the SKSpriteNodes that are used within the scene (see Subclassing SKSpriteNode).

Create an SKScene that Fills the SKView

A simple use case it to create an SKScene that exactly fills the SKView. This avoids the need to consider scaling the view to fit or setting a camera to show a part of the scene.

The following code assumes an SKView called skView already exists (e.g. as defined in Create a Full Screen SKView using Interface Builder) and a subclass of SKScene called GameView has been defined:

In Swift:

```
let sceneSize = CGSizeMake(skView.frame.width, skView.frame.height)
let scene = SKScene(size: sceneSize)

skView.presentScene(scene)
```

However if the SKView can change size (e.g. if the user rotates their device and this causes the view to be stretched because of its constraints) then the SKScene will no longer fit the SKView. You could manage this by resizing the SKScene each time the SKView changes size (e.g. in the didChangeSize method).

Create an SKScene that Scales to fit the SKView

An SKScene has a **scaleMode** parameter that defines how it will change its size to fit within the SKView it is presented into the SKView if it is not the same size and/or shape.

There are four options for scaleMode:

- **AspectFit**: the scene is scaled (but not stretched) until it fits within the view. This ensures that the scene is not distorted but there may be some areas of the view that are not covered by the scene if the scene is not the same shape as the view.
- **AspectFill**: the scene is scaled (but not stretched) to fill the view completely. This ensures that the scene is not distorted and that the view is completely filled but some parts of the scene may be cropped if the scene is not the same shape as the view.
- **Fill**: the scene is scaled (and if necessary stretched) to fill the view completely. This ensure that the view is completely filled and that none of your scene is cropped but the scene will be distorted if the scene is not the same shape as the view.
- **ResizeFill**: the scene is not scaled at all but rather its size is changed to fit the size of the view.

The following code assumes an SKView called skView already exists (e.g. as defined in Create a Full Screen SKView using Interface Builder) and a subclass of SKScene called GameView has been defined and then uses the **AspectFill** scaleMode:

In Swift 3:

```
    let sceneSize = CGSize(width:1000, height:1000)
    let scene = GameScene(size: sceneSize)
    scene.scaleMode = .aspectFill

    skView.presentScene(scene)
```

Create an SKScene with an SKCameraNode (iOS 9 and later)

You can place an SKCameraNode into an SKScene to define which part of the scene is shown in the SKView. Think of the SKScene as a 2D world with a camera floating above it: the SKView will show what the camera 'sees'.

E.g. the camera could be attached to the main character's sprite to follow the action of a scrolling game.

The SKCameraNode has four parameters that define what part of the scene is shown:

- **position**: this is the position of the camera in the scene. The scene is rendered to place this position in the middle of the SKView.
- **xScale** and **yScale**: these define how the scene is zoomed in the view. Keep these two values the same to avoid distorting the view. A value of 1 means no zoom, values less than one will zoom in (make the sprites appear larger) and values above 1 will zoom out (make the sprites appear smaller).
- **zRotation**: this defines how the view is rotated in the view. A value of zero will be no rotation. The value is in radians, so a value of Pi (3.14...) will rotate the view upside-down.

The following code assumes an SKView called skView already exists (e.g. as defined in Create a Full Screen SKView using Interface Builder) and a subclass of SKScene called GameView has been defined. This example just sets the camera's initial position, you would need to move the camera (in the same way as you would other SKSpriteNodes in the scene) to scroll your view:

In Swift 3:

```
    let sceneSize = CGSize(width:1000, height:1000)
    let scene = GameScene(size: sceneSize)
    scene.scaleMode = .aspectFill

    let camera = SKCameraNode()
    camera.position = CGPointM(x:500, y:500)
    camera.xScale = 1
    camera.yScale = 1
    camera.zRotation = 3.14
    scene.addChild(camera)
    scene.camera = camera

    skView.presentScene(scene)
```

Read SKScene online: https://riptutorial.com/sprite-kit/topic/4519/skscene

Chapter 7: SKSpriteNode (Sprites)

Syntax

- convenience init(imageNamed name: String) // Create an SKSpriteNode from a named image in the assets catalogue
- var position: CGPoint // SKNode property, inherited by SKSpriteNode. The position of the node in the parents co-ordinate system.
- func addChild(_ node: SKNode) // SKNode method, inherited by SKScene. Used to add an SKSpriteNode to the scene (also used to add SKNodes to other SKNodes).

Examples

Adding a Sprite to the Scene

In SpriteKit a Sprite is represented by the `SKSpriteNode` class (which inherits from `SKNode`).

First of all create a new Xcode Project based on the SpriteKit template as described in Your First SpriteKit Game.

Creating a Sprite

Now you can create a SKSpriteNode using an image loaded into the `Assets.xcassets` folder.

```
let spaceship = SKSpriteNode(imageNamed: "Spaceship")
```

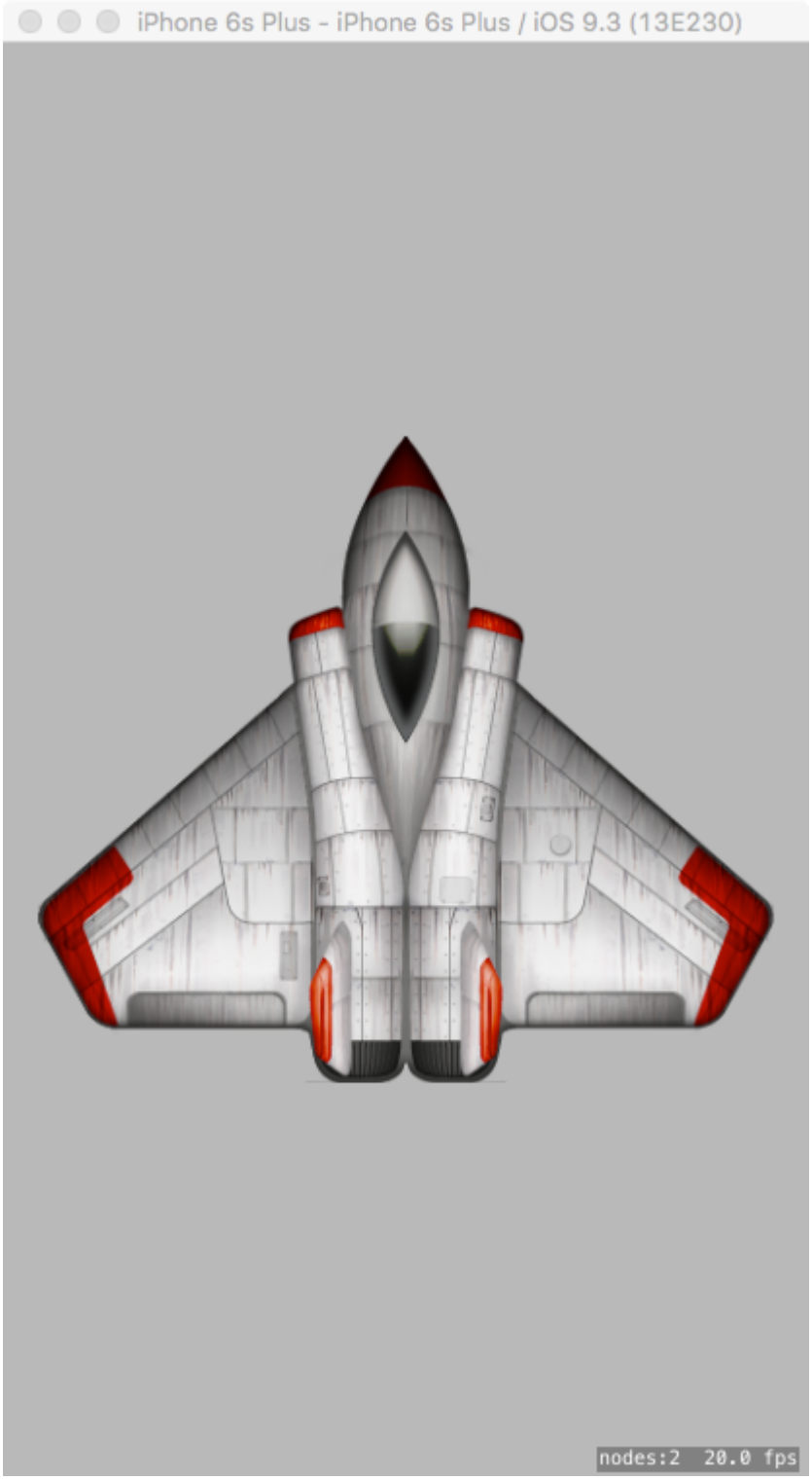
`Spaceship` is the name of the image item into the Assets.xcassets.

After the sprite has been created you can add it to your scene (or to any other node).

Open `GameScene.swift`, remove all its content and add the following

```
class GameScene: SKScene {
    override func didMoveToView(view: SKView) {
        let enemy = SKSpriteNode(imageNamed: "Spaceship")
        enemy.position = CGPoint(x:self.frame.midX, y:self.frame.midY)
        self.addChild(enemy)
    }
}
```

Now press `CMD` + `R` in Xcode to launch the Simulator.



Subclassing SKSpriteNode

You can subclass `SKSpriteNode` and define your own type of sprite.

```
class Hero: SKSpriteNode {
    //Use a convenience init when you want to hard code values
    convenience init() {
        let texture = SKTexture(imageNamed: "Hero")
        self.init(texture: texture, color: .clearColor(), size: texture.size())
    }
```

```
    //We need to override this to allow for class to work in SpriteKit Scene Builder
    required init?(coder aDecoder: NSCoder) {
        super.init(coder:aDecoder)
    }

    //Override this to allow Hero to have access all convenience init methods
    override init(texture: SKTexture?, color: UIColor, size: CGSize)
    {
        super.init(texture: texture, color: color, size: size)
    }
}
```

Read SKSpriteNode (Sprites) online: https://riptutorial.com/sprite-kit/topic/3001/skspritenode--sprites-

Chapter 8: SKView

Parameters

Parameter	Details
showsFPS	Display a count of the current frame rate in Frames Per Second in the view.
showsNodeCount	Display a count of the current number of SKNodes being displayed in the view.
showsPhysics	Display a visual representation of the SKPhysicsBodys in the view.
showsFields	Display an image representing the effects of the physics fields in the view.
showsDrawCount	Display a count of the number of drawing passes required to render the view.
showsQuadCount	Display a count of the number of rectangles required to render the view.

Remarks

An SKView is a subclass of UIView that is used to present SpriteKit 2D animations.

An SKView can be added to Interface Builder or programatically in the same way as 'normal' UIViews. SpriteKit content is then presented in the SKView in an SKScene.

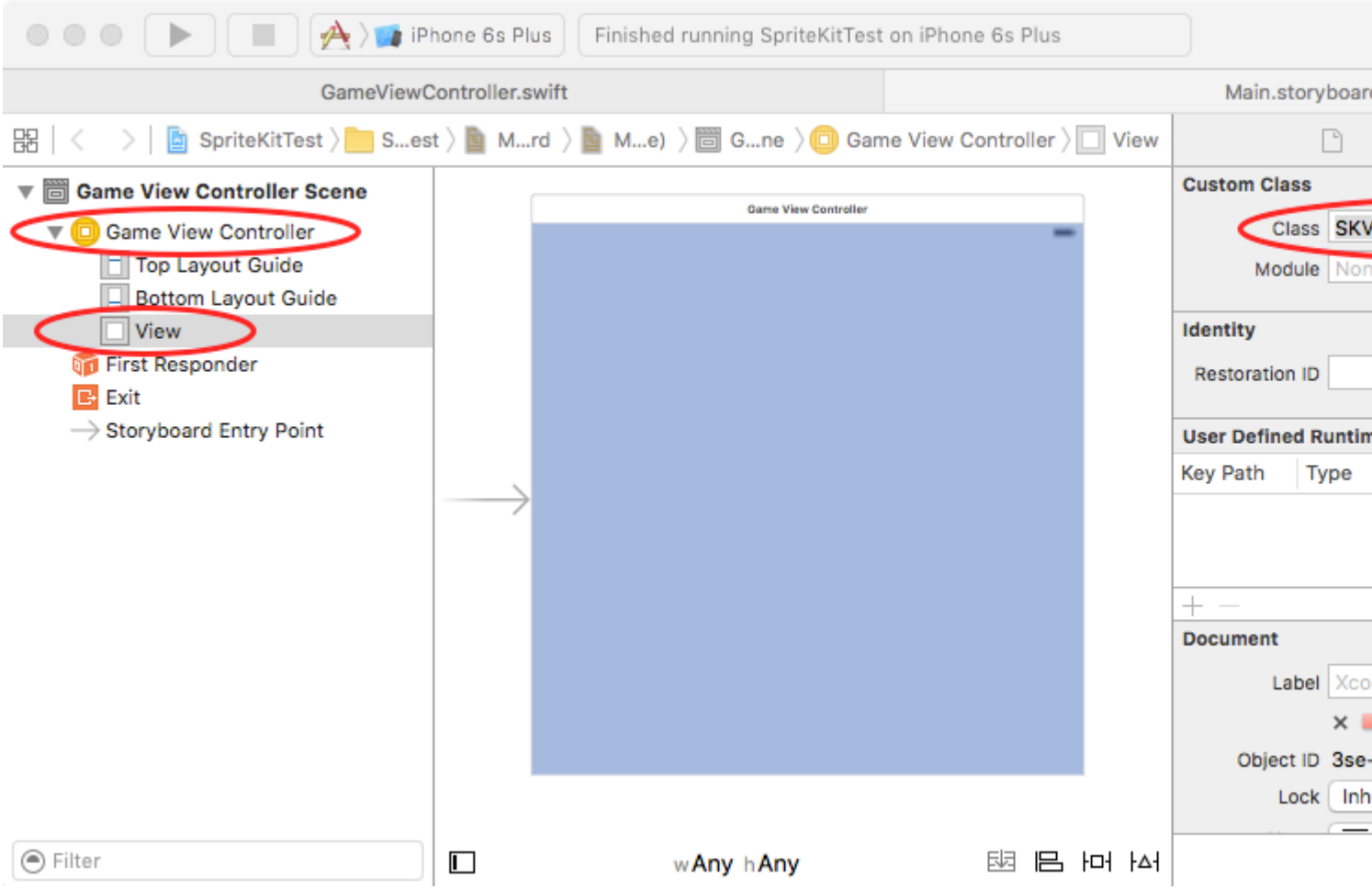
See also SKView Class Reference from Apple Documentation.

Examples

Create a full screen SKView using Interface Builder

A typical use case for SpriteKit is where the SKView fills the whole screen.

To do this in Xcode's Interface Builder, first create a normal ViewController, then select the contained view and change its **Class** from **UIView** to **SKView**:



Within the code for the View Controller, in the viewDidLoad method, grab a link to this SKView using self.view:

In Swift:

```
guard let skView = self.view as? SKView else {
    // Handle error
    return
}
```

(The guard statement here protects against the theoretical error that the view is not an SKView.)

You can then use this to perform other operations such as presenting an SKScene:

In Swift:

```
skView.presentScene(scene)
```

Displaying Debug Information

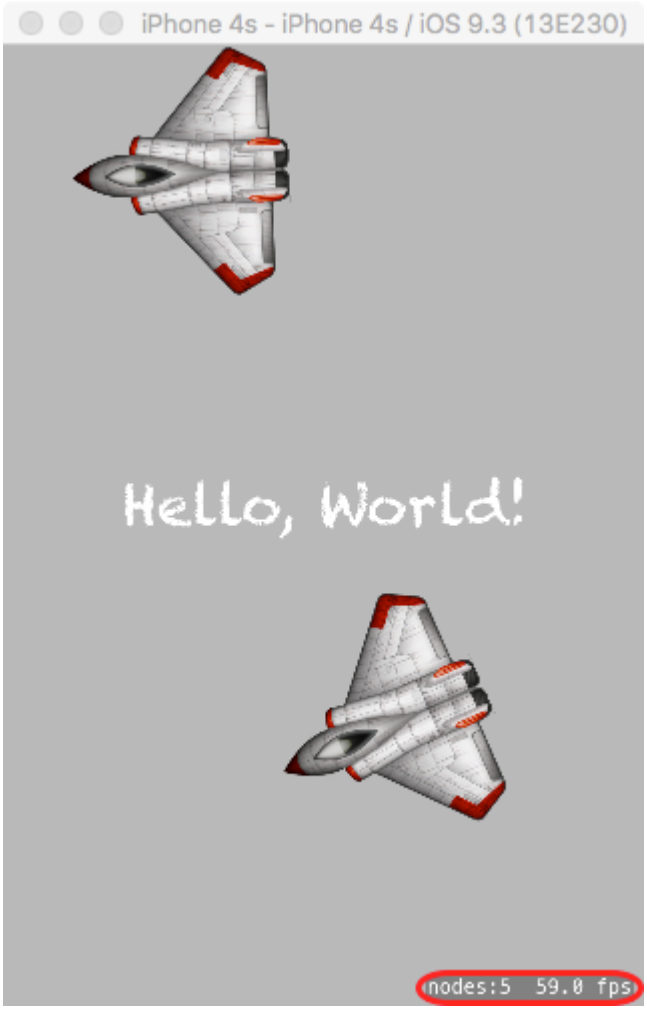
The current frame rate (in FPS, Frames Per Second) and total number of SKNodes in the scene (nodeCount, each sprite is an SKNode but other objects in the scene are also SKNodes) can be shown in the bottom right hand corner of the view.

These can be useful when turned on (set to true) for debugging and optimising your code but should be turned off (set to false) before submitting the app to the AppStore.

In Swift:

```
skView.showsFPS = true
skView.showsNodeCount = true
```

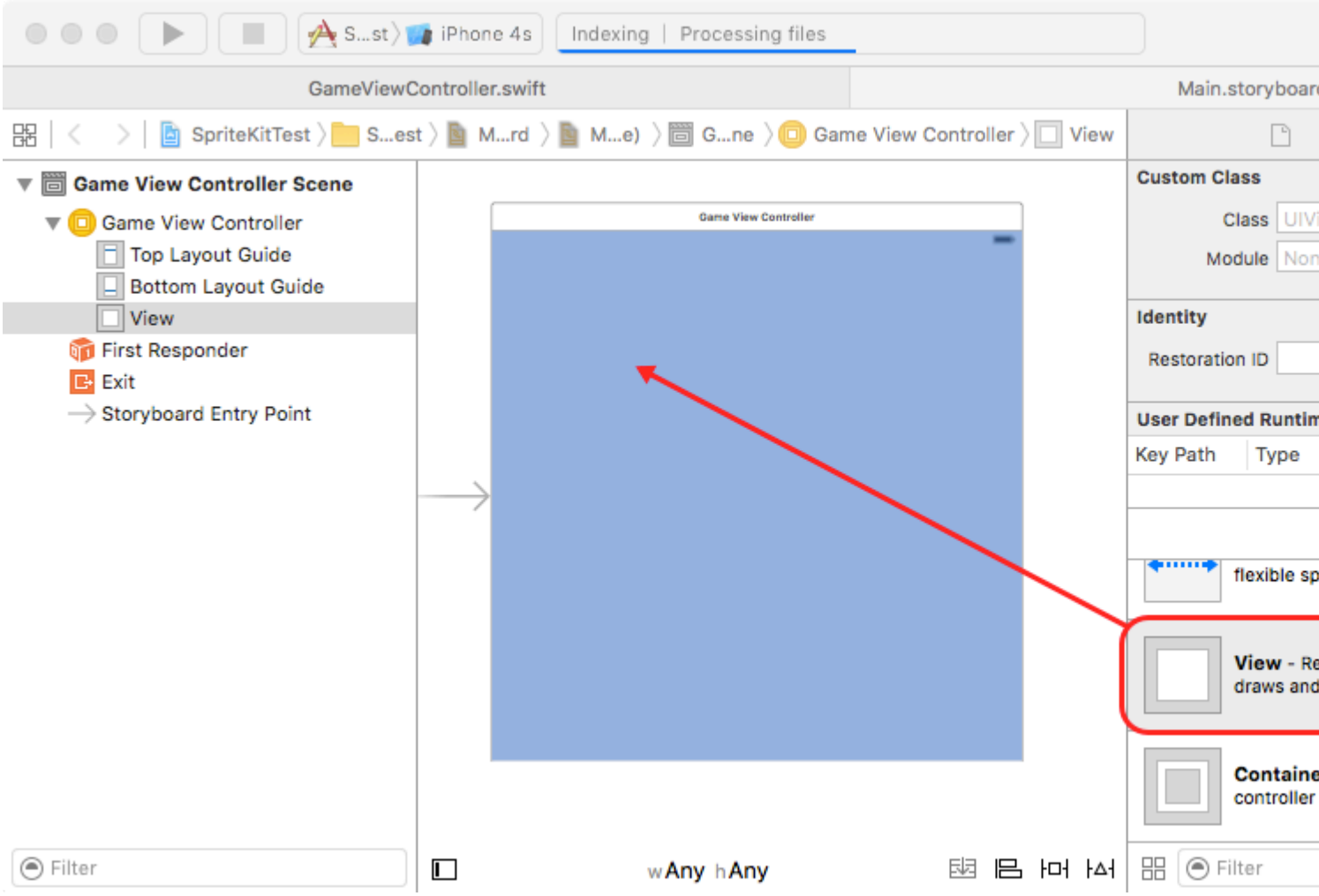
Result:



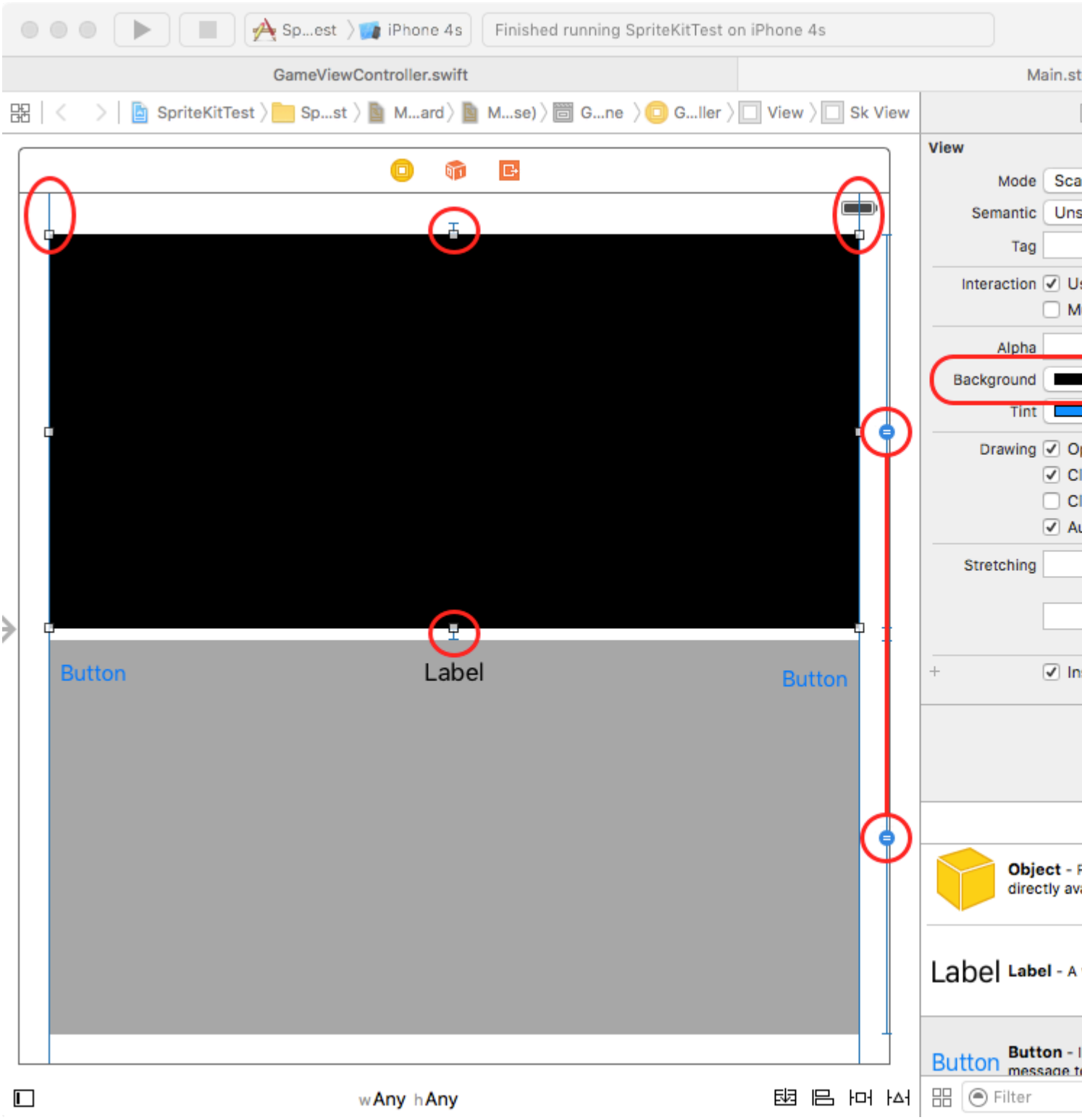
Create a small SKView with other controls using Interface Builder

An SKView does not need to fill the whole screen and can share space with other UI controls. You can even have more than one SKView displayed at once if you wish.

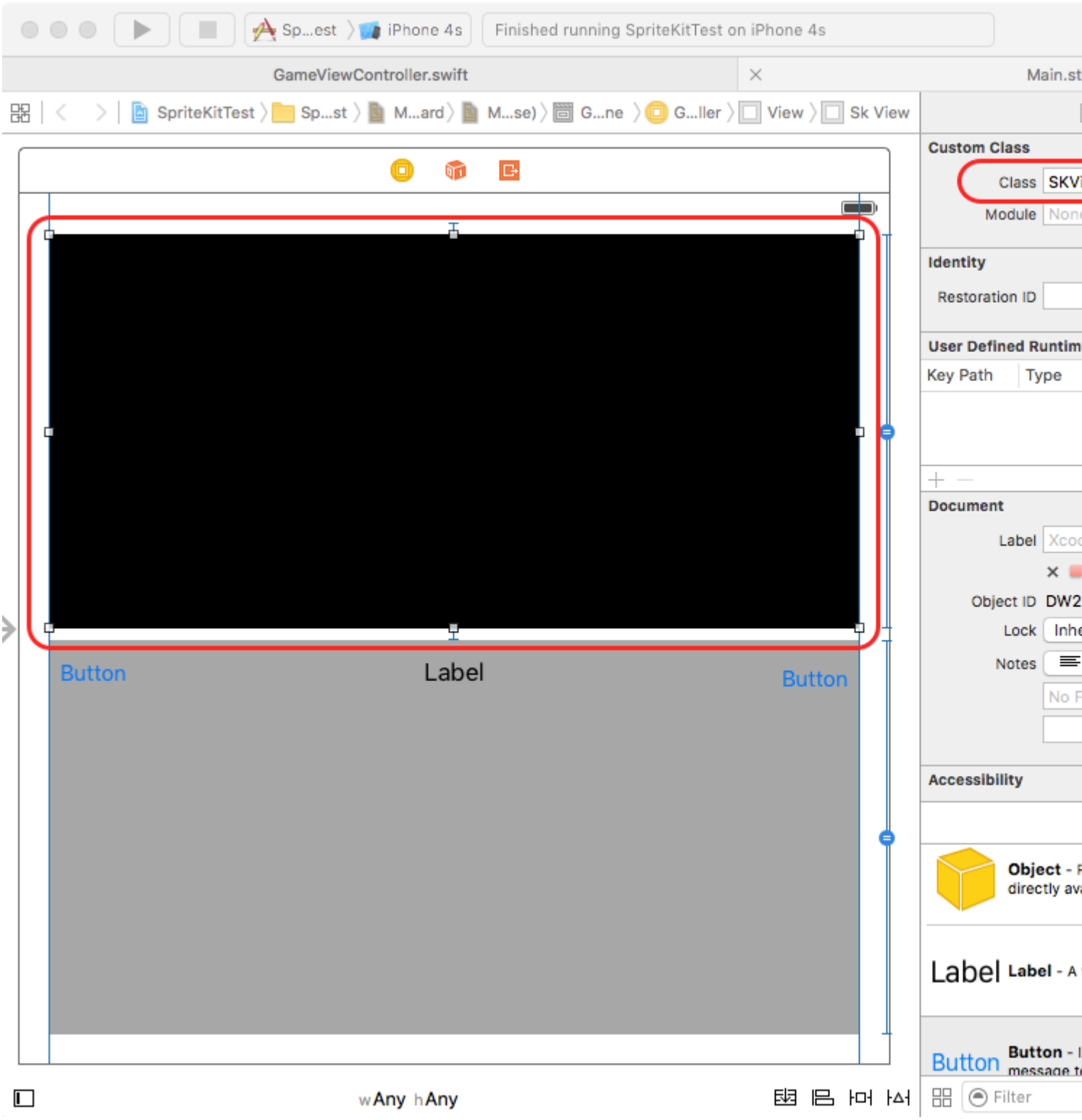
To create a smaller SKView amongst other controls with Interface Builder, first create a normal ViewController, then drag and drop a new view onto the view controller:



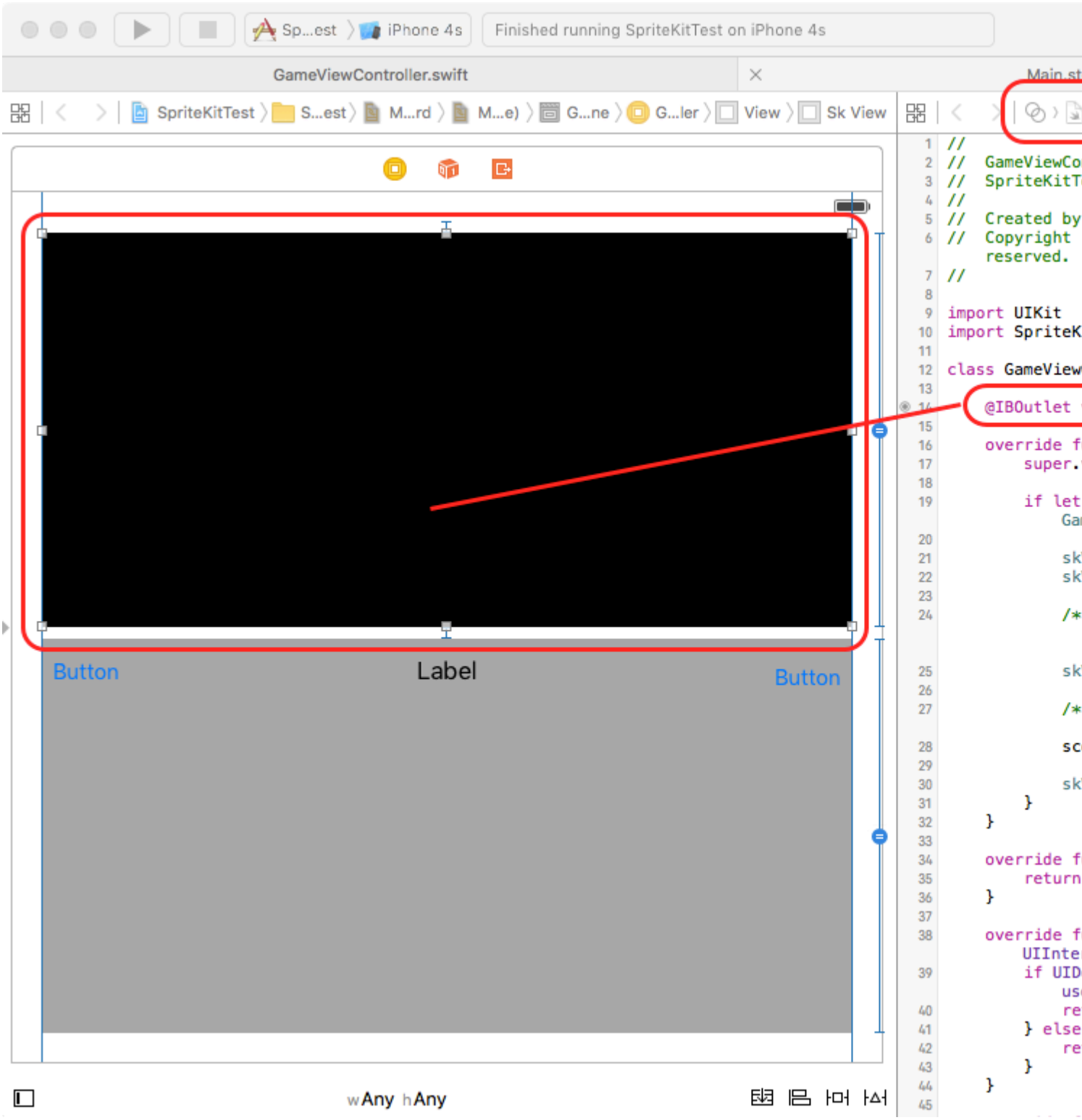
It can be helpful to set the colour of this view to something other than white (here black is used) so that it can be seen more clearly in Interface Builder (this colour will not be shown on the final app). Add other controls (a UIView, two buttons and a label are shown here as examples) and use constraints as normal to lay them out on the display:



Then select the view you want to be an SKView and change its class to SKView:



Then, using the assistant editor, control-drag from this SKView to your code and create an Outlet:

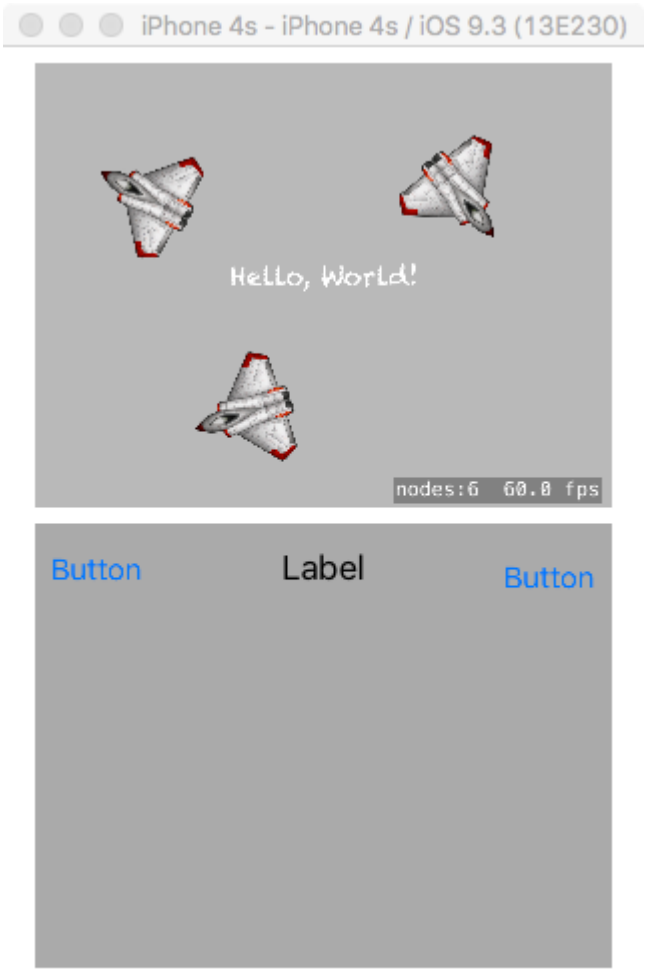


Use this outlet to present your SKScene.

In Swift:

```
skView.presentScene(scene)
```

Result (based on the Hello World example):



Read SKView online: https://riptutorial.com/sprite-kit/topic/3572/skview

Chapter 9: Timed functions in SpriteKit: SKActions vs NSTimers

Remarks

When should you use `SKAction`s to perform timer functions? Almost always. The reason for this is because `Sprite Kit` operates on an update interval, and the speed of this interval can be changed throughout the life time of the process using the `speed` property. Scenes can also be paused as well. Since `SKAction`s work inside the scene, when you alter these properties, there is no need to alter your time functions. If your scene is 0.5 seconds into the process, and you pause the scene, you do not need to stop any timers and retain that 0.5 second difference. It is given to you automatically, so that when you unpause, the remaining time continues.

When should you use `NSTimer`s to perform timer functions? Whenever you have something that needs to be timed outside of the `SKScene` environment, and also needs to be fired even when the scene is paused, or needs to fire at a constant rate even when the scene speed changes.

This is best used when working with both `UIKit` controls and `SpriteKit` controls. Since `UIKit` has no idea about what goes on with `SpriteKit`, `NSTimer`s will fire regardless of the state of the `SKScene`. An example would be we have a `UILabel` that receives an update every second, and it needs data from inside your `SKScene`.

Examples

Implementing a method that fires after one second

SKAction:

```
let waitForOneSecond = SKAction.waitForDuration(1) let action = SKAction.runBlock(){action()}
let sequence = SKAction.sequence([waitForOneSecond,action]) self.runAction(sequence)
```

NSTimer:

```
NSTimer.scheduledTimerWithTimeInterval(1, target: self, selector: #selector(action), userInfo:
nil, repeats: false)
```

Read Timed functions in SpriteKit: SKActions vs NSTimers online: https://riptutorial.com/sprite-kit/topic/5962/timed-functions-in-spritekit---skactions-vs-nstimers

Chapter 10: UIKit elements with SpriteKit

Examples

UITableView in SKScene

```
import SpriteKit
import UIKit
class GameRoomTableView: UITableView,UITableViewDelegate,UITableViewDataSource {
    var items: [String] = ["Player1", "Player2", "Player3"]
    override init(frame: CGRect, style: UITableViewStyle) {
        super.init(frame: frame, style: style)
        self.delegate = self
        self.dataSource = self
    }
    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
    // MARK: - Table view data source
    func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return items.count
    }
    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        let cell:UITableViewCell = tableView.dequeueReusableCell(withIdentifier: "cell")! as
UITableViewCell
        cell.textLabel?.text = self.items[indexPath.row]
        return cell
    }
    func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) -> String?
{
        return "Section \(section)"
    }
    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
        print("You selected cell #\(indexPath.row)!")
    }
}
class GameScene: SKScene {
    var gameTableView = GameRoomTableView()
    private var label : SKLabelNode?
    override func didMove(to view: SKView) {
        self.label = self.childNode(withName: "//helloLabel") as? SKLabelNode
        if let label = self.label {
            label.alpha = 0.0
            label.run(SKAction.fadeIn(withDuration: 2.0))
        }
        // Table setup
        gameTableView.register(UITableViewCell.self, forCellReuseIdentifier: "cell")
        gameTableView.frame=CGRect(x:20,y:50,width:280,height:200)
        view.addSubview(gameTableView)
        gameTableView.reloadData()
    }
}
```

Output:

Protocol/Delegate to call a game ViewController method from the game scene

GameScene code example:

```
import SpriteKit
protocol GameViewControllerDelegate: class {
    func callMethod(inputProperty:String)
}
class GameScene: SKScene {
    weak var gameViewControllerDelegate:GameViewControllerDelegate?
    override func didMove(to view: SKView) {
        gameViewControllerDelegate?.callMethod(inputProperty: "call game view controller
method")
    }
}
```

GameViewController code example:

```
class GameViewController: UIViewController, GameViewControllerDelegate {
    override func viewDidLoad() {
        super.viewDidLoad()
        if let view = self.view as! SKView? {
            // Load the SKScene from 'GameScene.sks'
            if let scene = SKScene(fileNamed: "GameScene") {
                let gameScene = scene as! GameScene
```

```
                gameScene.gameViewControllerDelegate = self
                gameScene.scaleMode = .aspectFill
                view.presentScene(gameScene)
            }
            view.ignoresSiblingOrder = true
            view.showsFPS = true
            view.showsNodeCount = true
        }
    }
    func callMethod(inputProperty:String) {
        print("inputProperty is: ",inputProperty)
    }
}
```

Output:

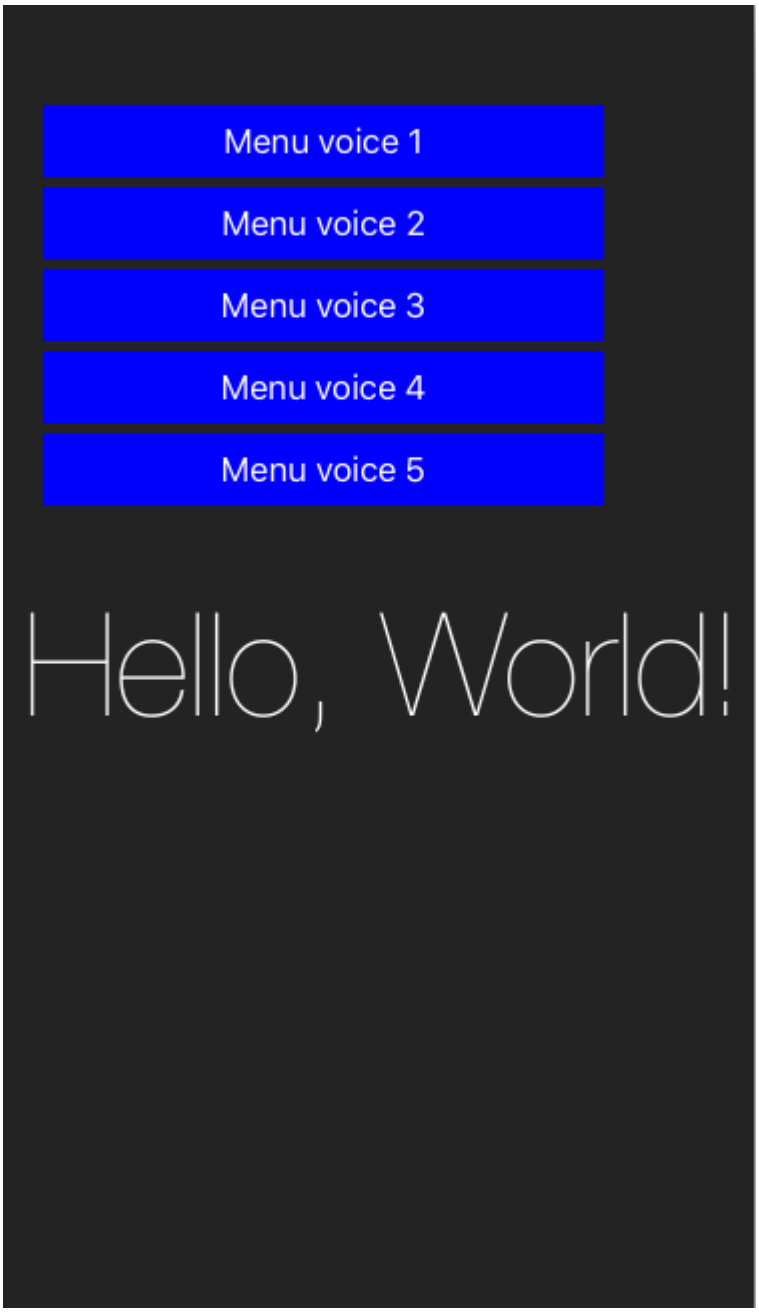
```
inputProperty is:  call game view controller method
```

StackView in SKScene

```
import SpriteKit
import UIKit
protocol StackViewDelegate: class {
    func didTapOnView(at index: Int)
}
class GameMenuView: UIStackView {
    weak var delegate: StackViewDelegate?
    override init(frame: CGRect) {
        super.init(frame: frame)
        self.axis = .vertical
        self.distribution = .fillEqually
        self.alignment = .fill
        self.spacing = 5
        self.isUserInteractionEnabled = true
        //set up a label
        for i in 1...5 {
            let label = UILabel()
            label.text = "Menu voice \(i)"
            label.textColor = UIColor.white
            label.backgroundColor = UIColor.blue
            label.textAlignment = .center
            label.tag = i
            self.addArrangedSubview(label)
        }
        configureTapGestures()
    }
    required init(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
    private func configureTapGestures() {
        arrangedSubviews.forEach { view in
            view.isUserInteractionEnabled = true
            let tapGesture = UITapGestureRecognizer(target: self, action:
#selector(didTapOnView))
            view.addGestureRecognizer(tapGesture)
        }
    }
    func didTapOnView(_ gestureRecognizer: UIGestureRecognizer) {
        if let index = arrangedSubviews.index(of: gestureRecognizer.view!) {
```

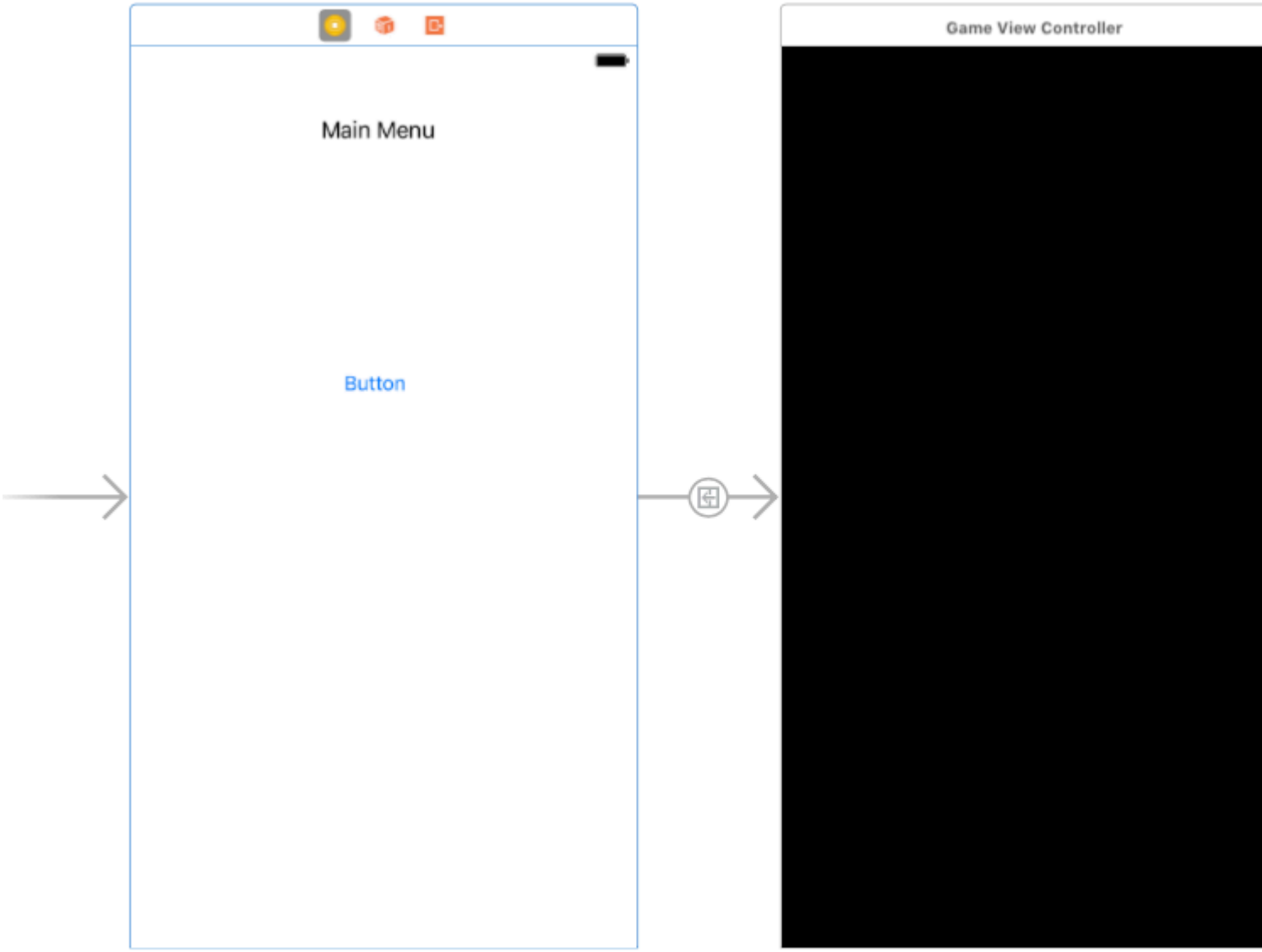
```
            delegate?.didTapOnView(at: index)
        }
    }
}
class GameScene: SKScene, StackViewDelegate {
    var gameMenuView = GameMenuView()
    private var label : SKLabelNode?
    override func didMove(to view: SKView) {
        self.label = self.childNode(withName: "//helloLabel") as? SKLabelNode
        if let label = self.label {
            label.alpha = 0.0
            label.run(SKAction.fadeIn(withDuration: 2.0))
        }
        // Menu setup with stackView
        gameMenuView.frame=CGRect(x:20,y:50,width:280,height:200)
        view.addSubview(gameMenuView)
        gameMenuView.delegate = self
    }
    func didTapOnView(at index: Int) {
        switch index {
        case 0: print("tapped voice 1")
        case 1: print("tapped voice 2")
        case 2: print("tapped voice 3")
        case 3: print("tapped voice 4")
        case 4: print("tapped voice 5")
        default:break
        }
    }
}
```

Output:



Multiple UIViewController in a game: how to jump from the scene to a viewController

Storyboard:



Initial viewController: an empty viewController with a button to present the GameViewController

GameViewController: the typical GameViewController of the *"Hello World"* Sprite-kit template.

Goal: I want to present the first viewController from my `SKScene` game with the correct deallocation of my scene.

Description: To obtain the result I've extended the `SKSceneDelegate` class to build a custom `protocol/delegate` that make the transition from the `GameViewController` to the first initial controller (main menu). This method could be extended to other viewControllers of your game.

GameViewController:

```
import UIKit
import SpriteKit
class GameViewController: UIViewController,TransitionDelegate {
    override func viewDidLoad() {
        super.viewDidLoad()
        if let view = self.view as! SKView? {
            if let scene = SKScene(fileNamed: "GameScene") {
```

```
                scene.scaleMode = .aspectFill
                scene.delegate = self as TransitionDelegate
                view.presentScene(scene)
            }
            view.ignoresSiblingOrder = true
            view.showsFPS = true
            view.showsNodeCount = true
        }
    }
    func returnToMainMenu(){
        let appDelegate = UIApplication.shared.delegate as! AppDelegate
        guard  let storyboard = appDelegate.window?.rootViewController?.storyboard else {
return }
        if let vc = storyboard.instantiateInitialViewController() {
            print("go to main menu")
            self.present(vc, animated: true, completion: nil)
        }
    }
}
```

GameScene:

```
import SpriteKit
protocol TransitionDelegate: SKSceneDelegate {
    func returnToMainMenu()
}
class GameScene: SKScene {
    override func didMove(to view: SKView) {
        self.run(SKAction.wait(forDuration: 2),completion:{[unowned self] in
            guard let delegate = self.delegate else { return }
            self.view?.presentScene(nil)
            (delegate as! TransitionDelegate).returnToMainMenu()
        })
    }
    deinit {
        print("\n THE SCENE \((type(of: self))) WAS REMOVED FROM MEMORY (DEINIT) \n")
    }
}
```

Read UIKit elements with SpriteKit online: https://riptutorial.com/sprite-kit/topic/8807/uikit-elements-with-spritekit

Credits

S. No	Chapters	Contributors
1	Getting started with sprite-kit	Ali Beadle, Community, Luca Angeletti
2	Detecting touch input on iOS devices	Knight0fDragon, Luca Angeletti
3	Physics	Alessandro Ornano
4	SKAction	Abdou023, Kendel
5	SKNode Collision	Chen Wei, Confused, Knight0fDragon, RamenChef, Steve Ives
6	SKScene	Ali Beadle
7	SKSpriteNode (Sprites)	Ali Beadle, Knight0fDragon, Luca Angeletti
8	SKView	Ali Beadle, Kendel
9	Timed functions in SpriteKit: SKActions vs NSTimers	Knight0fDragon
10	UIKit elements with SpriteKit	Alessandro Ornano, Knight0fDragon