iOS Development in Swift

Asheville School: App Camp 2018

Introduction

This book uses Glossary - tap or click a bolded word in order to be directed to a definition for that word written by the authors for the context of **Swift** learners. This will be used to do two things: avoid unnecessary review, and provide ample additional information on topics either overly advanced for or out of the scope of the projects in the book.

Words that are defined in the text or have already been bolded in the current chapter will still be linked, but will not be bolded again. In other words, most programming-related words will be linked to glossary entries and can be defined by tapping or clicking on them - you just won't constantly see bolded words.

We provide assets for this tutorial, but it's very easy to find your own - many artists have made their games' assets available under a creative commons or alternative free use license, meaning anybody is able to use them in any project, commercial or otherwise. These should be discoverable from searching Google, or on a website like opengameart.org. Sounds are available on similar platforms such as freesound.org. Be sure to check the license before using any of these assets - sometimes, creators require that anybody using their assets give them credit in the game credits or, less often, that their assets can only be used in non-commercial endeavors. Don't run afoul of copyright law in the pursuit of fame and fortune via microtransactions.

Good luck, and don't be afraid to modify the methods used in this tutorial to forge new ground in your own games! The best way to learn programming is by establishing a foundation and then trying to make ambitious projects of your own - the internet is a novice programmer's best friend and greatest ally. Mistakes are nothing to worry about - the best way to realize that a way of doing something is sub-optimal is by experiencing the consequences and fixing the problem yourself. Trial and error will eventually turn anyone who spends enough time coding into a master - formal education or otherwise. Some of the most successful programmers in the world are entirely self-taught. Fight on! Deus Vult!

Nebula ‘Nnhilation

Fundamentals of SpriteKit and Game Loops

Difficulty: Advanced

In this chapter, we will be creating a bullet hell shooter game called Nebula ‘Nnhilation. From a coding standpoint, it is very similar to the classic Galaga games, and in fact used the **sprites** from Galaga as test assets while in development. This game will serve as an introduction to the development of complex games, and after completing and understanding the concepts presented, the reader will ideally have obtained the skills necessary to build their own complex game from scratch.

A non-extensive list of the concepts covered is below:

- **Game loops**
- The **SceneKit** editor
- Advanced **SpriteKit** and **SceneKit** concepts
- Physics
- Complex touch controls
- **SKActions**
- Collision detection

Tutorial Author: Jack Hamilton

Swift 3

Section 1

Getting Started

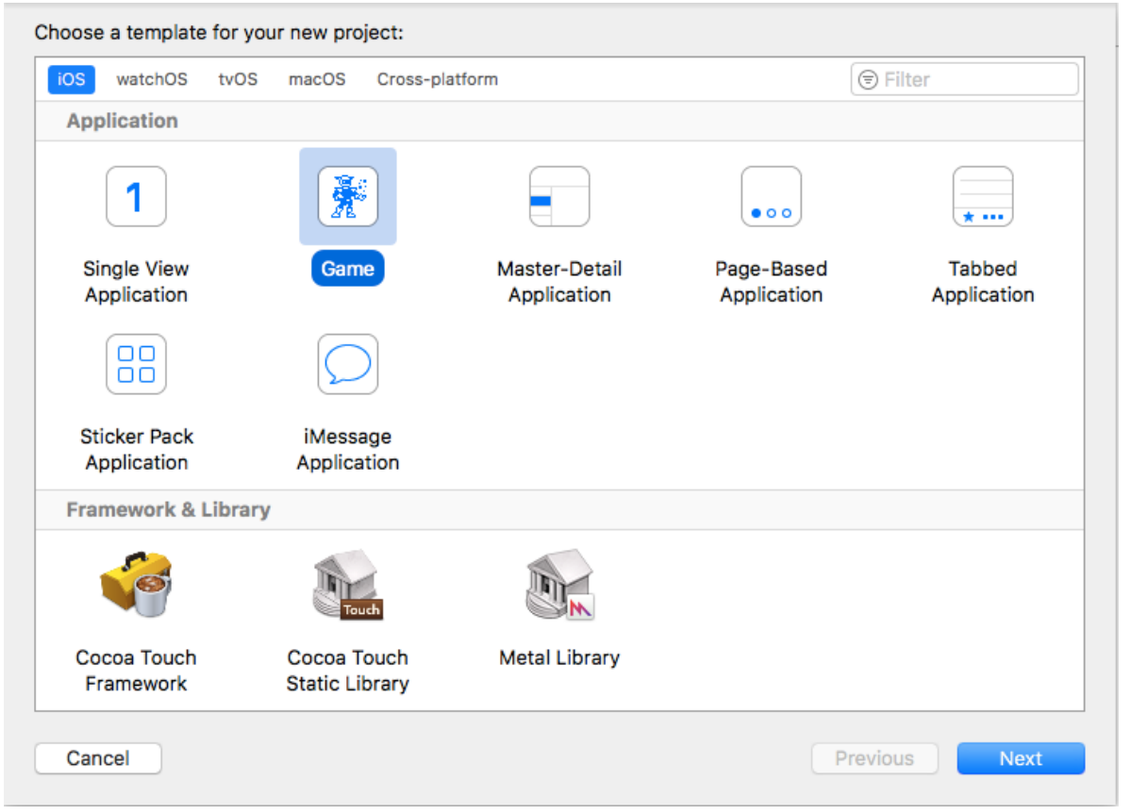
Though this tutorial assumes basic knowledge of **SpriteKit** and programming fundamentals, it is written to be accessible to those without experience. That said, having some experience with them will make your time easier. Note that this tutorial uses complex programming vocabulary, but you're not expected to understand it to start with - just learn it as you go along. There are glossary links for programming terms provided everywhere to ease this process.

This tutorial is structured in such a way that it can be used as a reference guide for those trying to make their own app: each section talks in depth about how to do a specific thing in **Swift** or **Xcode**, so those interested in doing their own thing can look at specific sections covering features they are interested in implementing in their own projects and come out with an understanding about how that concept works. However, the best way to learn is by following the entire project from front to back, and then using it as a reference later. For best results, we recommend not simply copying the steps the tutorial explains into your own project, but making an effort to understand what the code created does and, more importantly, *why*.

With that out of the way, let's get started!

Project Setup

Start Xcode and create a new project by clicking File > New > Project. Select the Game template under iOS > Application and click Next. This will create a project with many components of a game already generated for you.



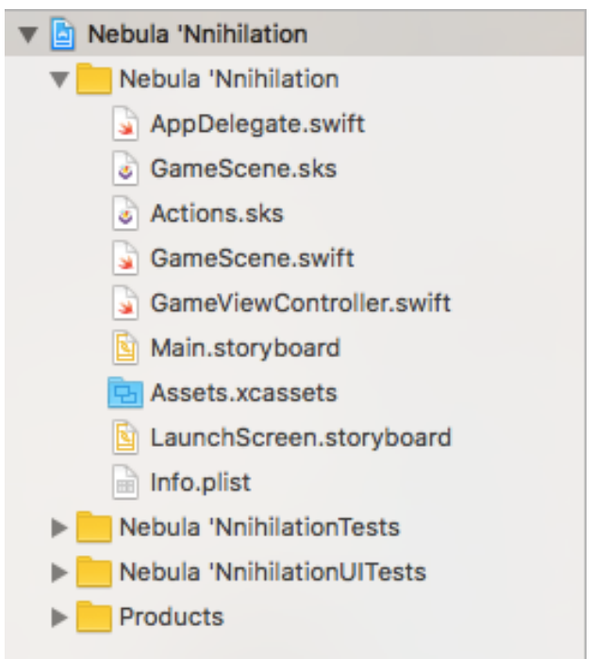
You will then be presented with a section asking for a project name and some identifiers. Feel free to name your project whatever you like, and note that all of these fields are modifiable later.

The dialog will also ask which technologies you want to build your game on, in order to identify which preset files to

include in your project. For this project, leave it as the default - **SpriteKit** with support for Universal devices. The language option should, of course, be left as ‘Swift’.

Near the bottom, there are also three checkboxes - leave these as the default. We won’t be covering unit or UI tests, but it’s useful to include them in case you want to code your own later. Click ‘Next’.

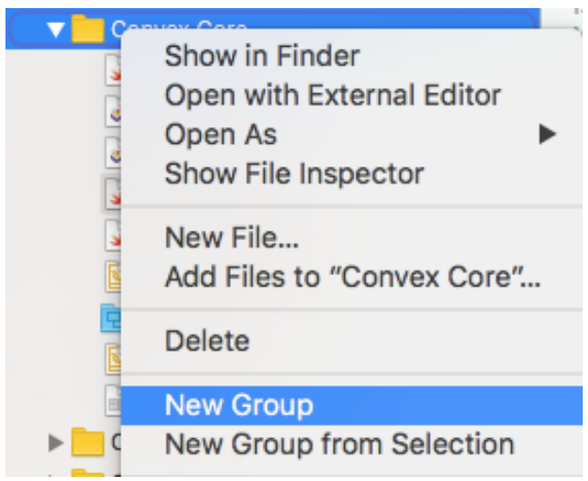
Xcode will now ask you where you want to save, so select a location and click ‘Create’. It’s also worth noting that the default setting is to create a **Git** repository in your project, which is (among other uses) a version control engine - what this does is ensure that you have access to all versions of your project at all times. If you make a change that breaks your project, it is possible to use the built-in source control to revert your project to the save immediately before you made the faulty change.



On the left is the file structure that the ‘game’ preset should have generated for you.

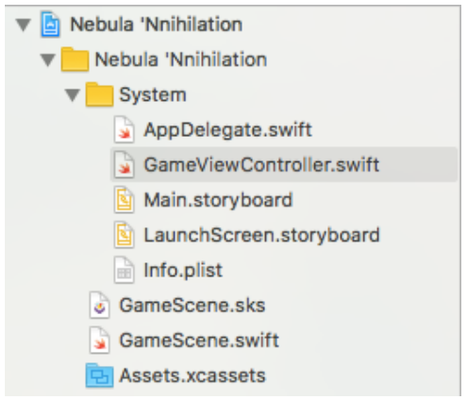
We’re going to clean this up in a bit, but first go into the project settings by selecting the title as shown to the left. This should take you to a section where you can set the Display Name of the app - do this now. This will be the name that appears under the app icon when the app is installed. Also on this page is the supported orientations section - uncheck ‘Landscape Right’ and ‘Landscape Left’ now.

If you don't have signing setup in Xcode, under 'signing', click on the 'team' dropdown, then choose 'add an account'. Sign in with your Apple account. This will enable testing to devices - the ability to **compile** your app onto an iPhone or iPad and test it there instead of in Xcode's built-in iPhone/ iPad simulator application.



Now to clean up the file structure. Right click on the game's main folder, and in the dropdown select 'New Group'. Name it 'System'. This is where we will put the files Xcode generated that we don't need to touch.

Move "*AppDelegate.swift*", "*GameViewController.swift*", and "*LaunchScreen.storyboard*" into this group. Then delete the file "*Actions.sks*" - we'll be doing all of our **SKAction** coding manually rather than in the editor. Do note, however, that there is an Action editor in Xcode that you can use if you prefer. This can be accessed by File > New > File and then scrolling down to Resource and creating a SpriteKit action.



Your final file structure should look like the one on the left.

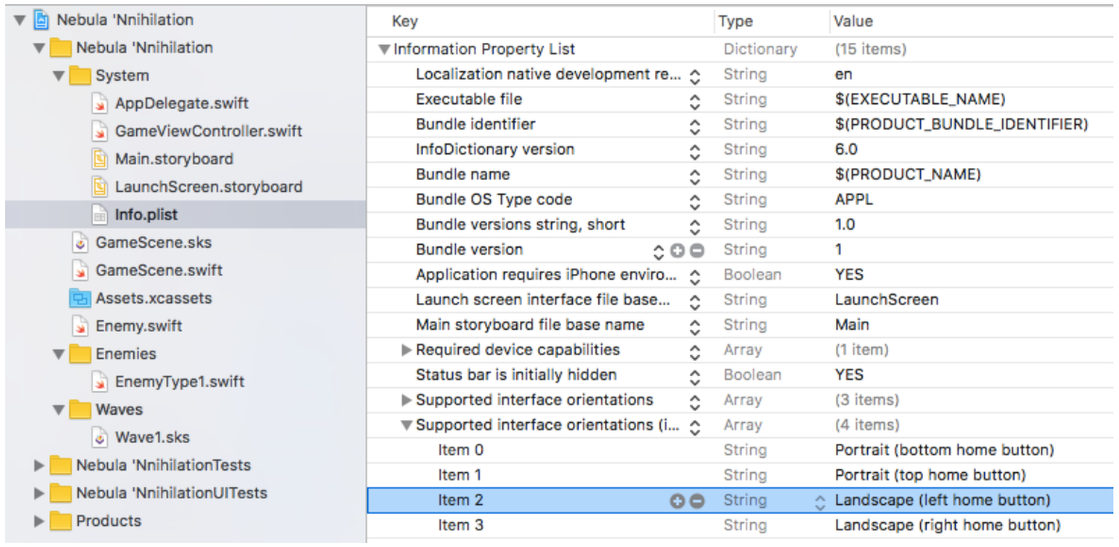
After this, head into *GameScene.swift* and delete all sections of code except for the following shell (or simply delete the contents of the file and copy in the below code):

```
import SpriteKit
import GameplayKit

class GameScene: SKScene {
    override func didMove(to view: SKView) {

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

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

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

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

    }
    override func update(_ currentTime: TimeInterval) {
        // Called before each frame is rendered
    }
}
```

We'll go over what this means in Section 3.

Next, go into *GameScene.sks*, click on the large “Hello, World” label at the center of the screen, and delete it. We’ll be putting our own stuff here later.

Finally, we’ll only be supporting portrait orientation in our app, so we have to disable landscape mode. To do this, go into Info.plist and expand the list of “Supported device orientations.” Press the minus button next to both landscape modes to delete them.



That ends the project setup! Let’s learn about the files Xcode has given us. Feel free to skip this next section if you understand what all of the files and **functions** currently in the project do.

Section 2

The Default Project

In this section, the autogenerated files Xcode includes in the Game template will be explained, along with all of the **functions** within relevant to our game project. We'll go more into depth with the critical functions as we build our project - this is simply an introduction and orientation section.

Looking in the System folder, the first file is ***AppDelegate.swift***. This project does not deal with the contents of this file at all, but what it does is respond to a change in the app window's state - essentially, the functions inside AppDelegate run whenever the app starts, is put into the background, reactivates into the foreground, or changes state in any way. It can be used to do things like pause a game when the user enters the notification tray and resume it when they swipe back up into the game. We won't be touching it for this project.

The second file is ***GameViewController.swift***. This file is a bit more critical, as it is the file that loads **scenes** onto the main screen. Though we will not have to touch that part for this game, as by default Xcode generates a line loading the app into the ***GameScene.swift*** scene that it generates. However, if you wanted to create a menu screen that the app loads into instead of just the main game, for instance, this file would be where you would do that (specifically, the line "`if let scene = SKScene(fileNamed: "GameScene")`" could be changed to refer to any scene file). Additionally, the following lines:

```
view.showsFPS = true
view.showsNodeCount = true
```

in the viewDidLoad() function are responsible for creating the node count and FPS counter that you can see at the bottom right of the screen when starting the project. Removing those is as simple as deleting those two lines of code, or changing the two **booleans**' values to false.

Info.plist stores your project settings. You can ignore this file because most options are editable in the Xcode interface. Sometimes you'll have to dip into this file, but mostly just for editing supported interface orientations (landscape, portrait, and a few others) in certain specific cases. For now, don't worry about it.

Outside of the System folder are the files we'll be dealing with more directly. ***GameScene.sks*** and ***GameScene.swift*** are two sides of the same file - the visual side that can be manipulated through Xcode's built in scene editor, and the code side that can be linked to elements in the visual side and directly and far more accurately manipulate those elements over time as the game runs. Typically elements are added to the scene visually, by drag and drop, into the *GameScene.sks* file, have their starting properties edited in the editor pane in the file, and then given actions and responses to certain events via code in the *GameScene.swift* file.

Third is the ***Main.storyboard*** file. This contains a collection of the UI (User Interface) screens present in the app, and is in this case simply a container for the game screen. We don't have to worry about this file in this project either, but if you wanted to create a menu or scoreboard in **UIKit** instead of SceneKit, this is where that would be done. UIKit is not covered in this chapter, but is covered extensively in the Swift documentation for anyone interested in exploring further. It is mainly used for building UI-based apps - anything with menus. One example of a UI app would be Snapchat, which is primarily UIKit-derived.

LaunchScreen.storyboard, the fourth file, is another UIKit document. It is the screen that displays while the app is loading. By default, it's just a white screen, but it's fairly easy to recolor it or put an image on it - for instance, your game logo. However, if you do, ensure that it scales properly by using the constraint system in the editor, otherwise it might be off center on larger devices or stretch improperly.

Finally, ***Assets.xcassets*** stores your project's assets - mostly used for images. Files can be added to your project by double clicking on the *Assets.xcassets* file in your project's file explorer and then dragging files into the editor that pops up. These files are then automatically imported into your scene editor's assets library and can be referenced in code using the file's name.

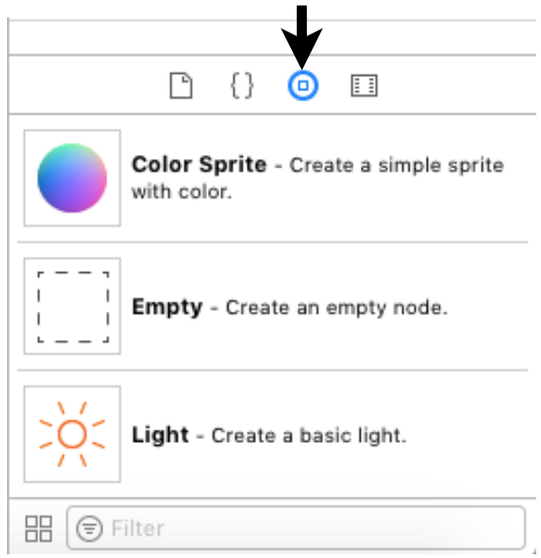
We will be creating our own files in addition to the presets in order to compartmentalize and better organize our code, but a simpler game could be created by just inserting code and editing elements through the scene editor into the preexisting files explained here.

Section 3

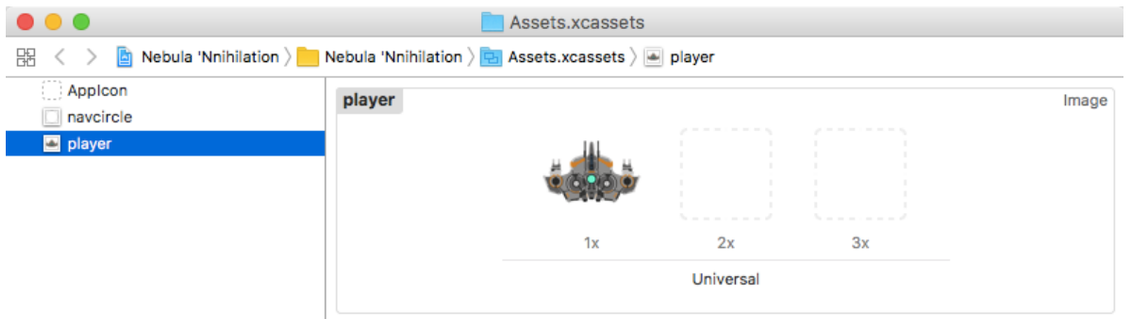
Creating a Ship

In this section, we will create a ship sprite and add touch controls to it so that the user can move it around the screen by tapping and dragging. This will be a long section, so strap in.

To begin, open your Assets.xcassets file and, from the assets provided with this tutorial, drag in the file for the player and the nav-circle: “player.png” and “navcircle.png”. This will add the files to the project, and also make them available for usage from the scene editor. You should also delete the “spaceship” file Xcode adds to the Assets editor as part of the ‘Game’ preset.

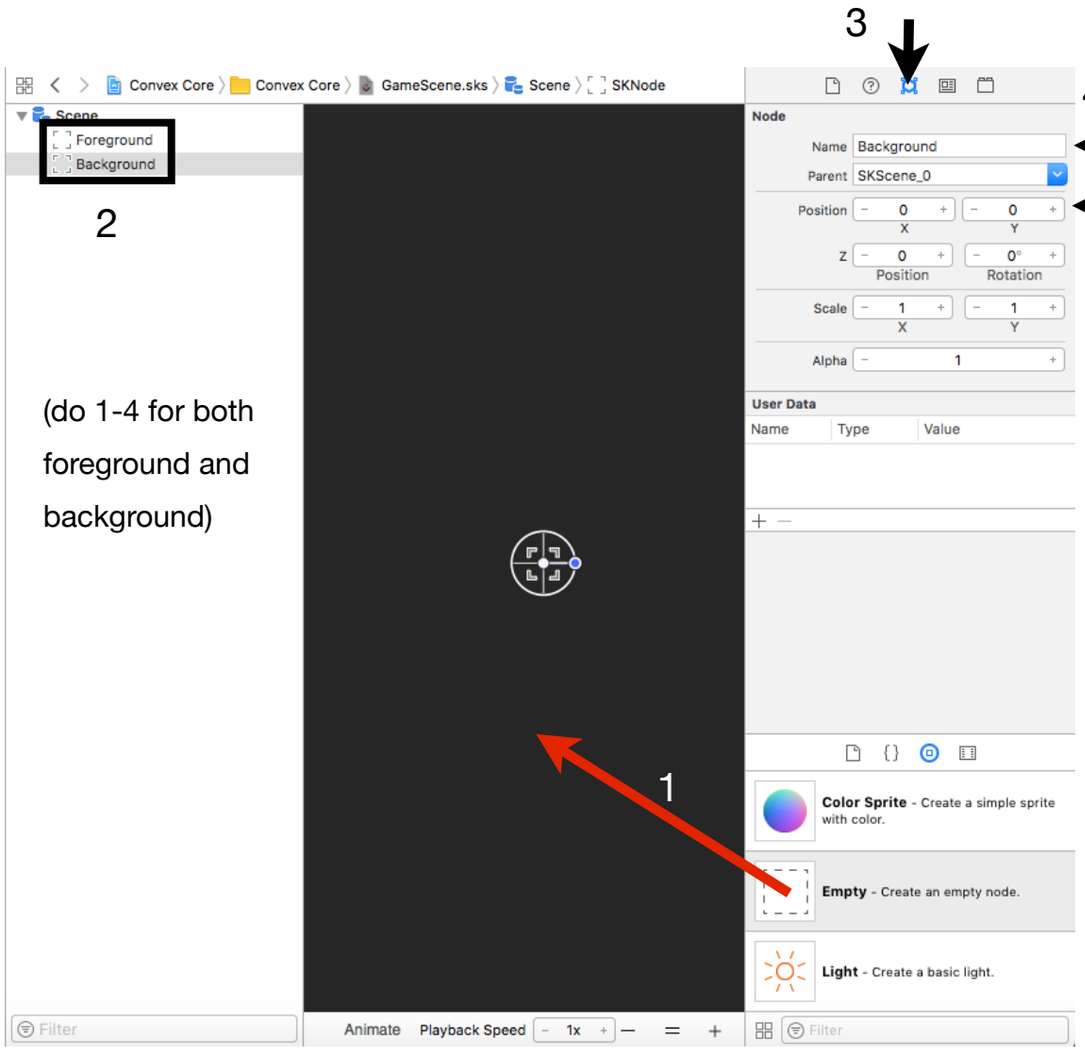


The Element Picker

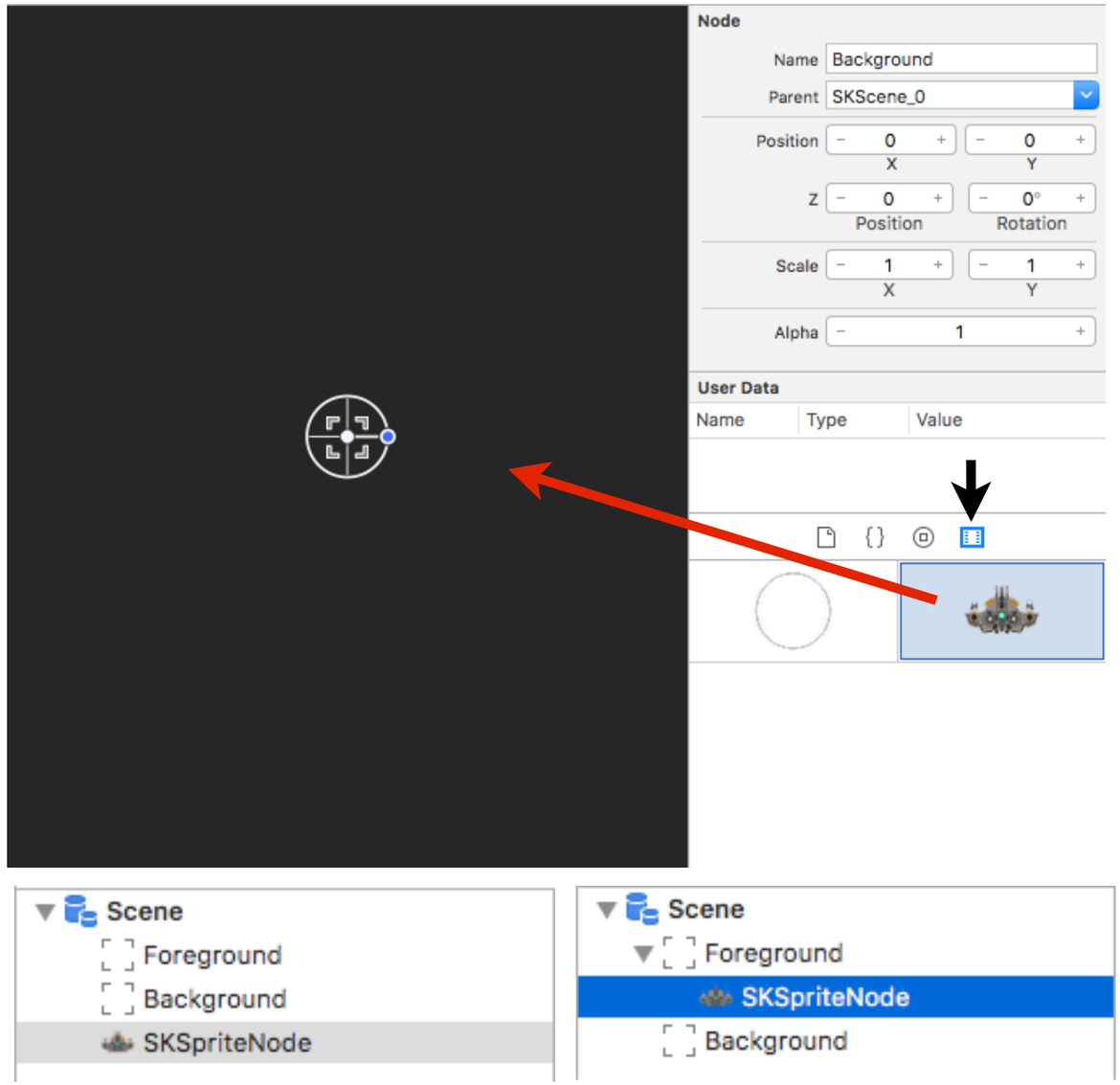


Now head into the GameScene.sks file to begin setup of the player character. In the bottom-right corner of Xcode’s editor, you should see a tray containing various elements. This is depicted to the left. You can drag elements from this tray onto the scene to add them. Drag two “Empty” node elements onto the screen, then click on each in the element view to the left of the screen.

In the properties view that will show up on the right of your screen, flip to the third tab and set the position of both nodes - both X and Y - to zero to center them on the screen. Rename one to ‘Foreground’ and one to Background. These nodes will allow us to better organize our sprites in the code. More on this later.



To add the player, switch to the assets tab in the **element picker** - the one farthest to the right. This tab displays all images that have been added to Assets.xcassets. From the elements that appear, drag the player image onto the screen. This will create the player's sprite for you. Then drag the player in the element view to underneath the 'Foreground' node. This will make it a child of the foreground - essentially, moving the foreground or adjusting any of its properties (opacity, rotation, z-position, etc.) will also adjust the properties of all of its children, including our new player



node. This way, we can manipulate the entire foreground as one entity.

Ensure the Foreground node has a dropdown appear.

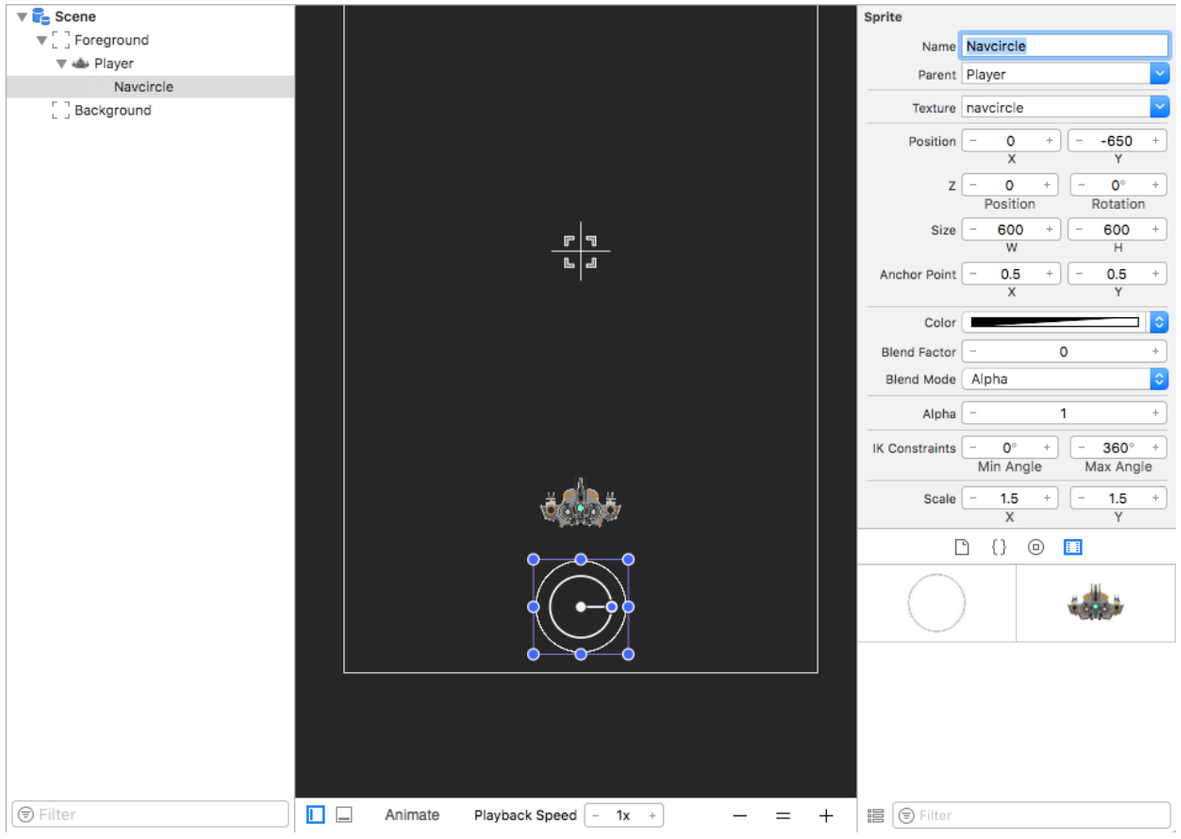
Next, we'll set the sprite size, name, and position. In the same menu that we used to change the empty nodes from earlier's properties, change the name to "Player", the X and Y position to 0 and -300 respectively, and the scale to X: 0.25

and Y: 0.25 (scale is just underneath the color options - it stretches width and height, in this case we're stretching them to a fourth of their original sizes). This should make the size slightly more reasonable and orient the center near the bottom of the screen.

Now, let's attach the navigation circle to the player. The navigation circle will be a circle on the screen that the player can touch and use to drag the player around the screen. Touching anywhere else won't do anything - we do this so that if we want to implement buttons or any other touch controls later, the controls from the ship don't interfere.

Drag the asset - navcircle.png, which at this point should be the only other thing in the element picker besides the player sprite - into the scene, then drag it in the **element view** on the left hand side of the screen into a position under and to the right of the player sprite. It should be a child of the player - in other words, it should be indented slightly farther than the player in the element view, and the player should have a dropdown arrow next to it that if clicked hides the navigation circle from the element view. What this also means is that if the position of the player is changed, the position of all children will be changed as well, which means that our navigation circle will follow the player around. Another way of putting this is that the navigation circle is positioned relatively to the player, its parent, rather than the scene.

In the properties menu we used earlier, adjust the position of the navigation circle to X: 0, Y: -650. Again, this is relative to



the player. Then set the scale to X: 1.5, Y: 1.5 to solve the oversizing issue, then rename the sprite to "Navcircle".

With this, we've finished the visual setup for our player - it's time to delve into the code. Open GameScene.swift. It should contain only the code we pasted in earlier, refer to Section 1 Page 5 if you have not done this.

Let's go through what each of the **functions** in this file do.

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

}
```

This function is the initializer - it is called when the app starts. It is also called when the app is reentered from the iOS multitasking menu. It should be used to perform game setup, and to create **reference variables** in the code for the things we create in the Scene Editor.

For now, add the following lines just before this function (and after `class GameScene: SKScene {`):

```
var foreground: SKNode!
var background: SKNode!
var player: SKSpriteNode!
var navcircle: SKSpriteNode!
```

This creates **global reference variables** for the foreground and background nodes, as well as our player and navigation circle sprites - note that the reason they're global is because they're declared outside of any functions. They're set to the types we created in the screen - the blank nodes are of type **SKNode**, and the sprites we dragged in are of type **SKSpriteNode** (the colon after the variable name, in the context of a variable declaration, indicates that the upcoming expression will be the type of the variable). Note that right now, these don't point anywhere - if you called a function on them or tried to manipulate them, you'd just get an error. This essentially only reserves the variable names we want.

Let's set those variables. In the didMove function (between the curly braces), put the following:

```
foreground = childNode(withName: "Foreground")!
background = childNode(withName: "Background")!
player = foreground.childNode(withName: "Player") as! SKSpriteNode
navcircle = player.childNode(withName: "Navcircle") as!
SKSpriteNode
```

Let's dissect those lines. For the first two, we're taking global variables that we declared earlier and setting them equal to something. What we're setting them equal to is the result of a function - childNode() - that takes one parameter: a variable called withName. This parameter is simply the name that we gave to the nodes when we dragged them into the scene. It is of type String, so it's surrounded with quotation marks.

The function, childNode(), because it does not have a variable it is being called on - e.g. playerOverlay.childNode, as is the case in the second two lines - must be a function from the class itself. In this case, if you look at the class declaration (`class GameScene: SKScene`), you see that the class has a parent class - SKScene. childNode() is a function from SKScene that returns whatever node within its direct children has a name matching the 'withName' parameter. Within the context of GameScene, as the GameScene.sks file pairs with GameScene.swift, the childNode() function can be used to retrieve things we've added to GameScene.sks that have no representation in the code.

In the four lines of code from earlier, we are taking our global variables and setting them equal to the result of the childNode() function, which will return in this case the nodes named “Foreground” and “Background” from the main scene. As ‘Player’ and ‘Navcircle’ are children of ‘Foreground’ and ‘Player’ respectively instead of being children of the main scene like the Foreground and Background nodes, we must call childNode() on their parent node instead of the main scene, hence the additional foreground.childNode() and player.childNode() instead of simply childNode(). This retrieves the child of the Foreground node named “Player” and sets the ‘player’ global variable to it, then retrieves the child of the Player node named “Navcircle” and sets the ‘navcircle’ global variable to it.

Moving on,

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

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

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

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

}
override func update(_ currentTime: TimeInterval) {

}
```

These functions trigger on touch events. When the screen is touched, a finger touching the screen moves, a finger is lifted from the screen, or an event such as a system alert interrupts a touch, these four functions are called respectively.

Our goal is to track when a touch occurs in the navball, then every time that finger moves, adjust the player to the new position. We don’t have to worry about repositioning the navigation circle, because that’s a child of the player, and is positioned relatively to it - changing the player position will change the child’s position by an equal amount. To this end, we will make a global variable to track the first touch that occurs within the navigation circle. We’ll set it when the navigation circle is touched, check it for changes when the touch moves, and end it when the finger is lifted.

First, create the global variable. In the same section as the other global variables - just above the didMove function - add three new variables:

```
var moving = false
var navcircleTouch: UITouch? = nil
var lastTouchPosition: CGPoint? = nil
let gameWidth = CGFloat(850)
let gameHeight = CGFloat(1334)
```

The first variable will track whether any fingers are touching the navigation circle, and the second variable will store the touch event for that touch - more on that when we implement the touchesMoved function. The third will track the previous position of each touch, so that whenever the player’s finger moves, we can adjust the position of the player sprite by the

difference between the new position of the player's finger and the old position of that finger. For now, these second and third variables will be set to `nil`, because when the program starts we will not have registered any fingers as touching the screen. We'll also define global variables for game height and width so that if we want to change them later, we won't have to rewrite a bunch of 'magic numbers' in the code, but can instead just change the value of these two variables. They're CGFloats because most functions that work with boundaries use this type for numerical input.

Place the following code into the touchesBegan and touchesMoved functions (between the curly braces):

```
for touch in touches {
    let touchPosition = touch.location(in: self)
}
```

The outer part of this is what's called an **extended for loop** - this executes the code within one time for each element in whatever array it's given. In this case, the array given is called 'touches', and it's a parameter of the touch-related functions that represents an array of all of the touches currently active on the screen. We store the touch we're currently working on in a variable called 'touch'. Hence, "for touch in touches" simply signifies that the following code in the curly braces is to be run once for each object in the array 'touches', and store the current object that is being worked on in a variable called 'touch'. In other words, we want to run the code within once for every finger on the screen.

The code within the for loop creates a variable, touchPosition, and stores inside of it the result of touch.location(in: self). The location() function, when called on a touch object, will fetch the location of said touch in a given reference frame. We want the reference frame to be the current scene, so we tell it to use self - a variable that refers to the current object of the current class, which in this case will be the current GameScene.

Next, in the touchesBegan function, under "let touchPosition = touch.location(in: self)" but still inside the curly braces, put:

```
let touchPositionInPlayerFrame = convert(touchPosition, to: player)
if (navcircle.frame.contains(touchPositionInPlayerFrame) &&
!moving) {
    lastTouchPosition = touchPosition
    navcircleTouch = touch;
    moving = true;
}
```

The first line there converts the touch's position into the coordinate system of the "player" object - because the navigation circle is a child of "player", it uses the player's coordinate system, so we have to convert the point before using the navigation circle's collision detection functions on it.

This uses a **boolean expression** to evaluate whether the following condition is true:

- The current touch is within the navigation circle's bounds AND the player is not currently moving the ship.

If the touch is outside of the navigation circle, we want to ignore it, and if the player already is touching the navigation circle, we want to ignore a second touch, so we only want to react when the player is both not currently touching the navigation circle and they begin a touch that is within the boundaries of said navigation circle.

The method by which we evaluate this is by constructing an expression that will only be true if both conditions are met. For the first condition, we want to check if the navigation circle contains the player's touch. We've already retrieved the location of the player's touch, and stored it in 'touchPosition'. Swift's built-in shape classes can check if a point falls inside them, so we get the shape representing the boundaries of the navigation circle by stating navcircle.frame, then call the contains() function on it. This will return a boolean 'true' if the point passed in is within the boundaries of the shape and a boolean 'false' otherwise. We pass the touch position into this contains() function, which will evaluate whether or not the touch is within the navcircle's shape. We then add an "&&" logical operator, which signifies that the entire expression in the if statement's parenthesis should only be true if the booleans on both side of the operator are true. On the other side of the expression, we simply put "!moving". The exclamation point inverts the value proceeding boolean, and this expression can be read as just "not moving". To put that all together,

if (the navcircle contains the touch && there's no movement already happening) {

```
        //Start the player moving.
}
```

In order to handle the player movement starting, we should first evaluate what we want to do. When the player's finger moves, we want the ship to move with it. Because there is a function that triggers specially when a touch moves, we don't have to handle any of that within this touchesBegan function - simply set up for the touchesMoved function to handle it. In order to do this, we simply need to set our global variables from earlier to reference the touch we've just recognized. "`lastMainTouchPosition = touchPosition`" stores the current touch position so that we can take the difference between the previously touched position and the position the player's finger has moved to when updating the ship's position. "`navcircleTouch = touch`" stores the touch object so we can ensure we only react to that specific touch - reacting to the movements of another finger on the screen as well as the finger touching the navcircle would result in the ship jumping around the screen erratically. Finally, "`lastMainTouchPosition = touchPosition`" signifies to the touchesMoved() function that there is, in fact, an active touch on the navigation circle that it should be reacting to.

Now we need to actually move the ship when a finger touching the navigation circle moves. The touchesMoved function triggers when a finger is moved on the screen, so when that happens, we want to first ensure that the touch that moved is the same as the one touching the navigation circle (assuming that there is more than one finger touching the

screen), then calculate the difference between the new position and the previous one, then move the ship by that much, then update the variable storing the previous position to the value of the new position so that next time the function runs it uses what has become the previous position. This is executed by the following code, which goes in touchesMoved under "let touchPosition = touch.location(in: self)" but before the closing curly brace:

```
if (moving && touch.isEqual(navcircleTouch)) {
    player.position.x = player.position.x - (lastTouchPosition!.x -
touchPosition.x)
    player.position.y = player.position.y - (lastTouchPosition!.y -
touchPosition.y)
    lastTouchPosition = touchPosition;
}
```

Breaking this down, the condition in the if() statement only executes the code block if both the player is moving the ship and the touch that movement has triggered for is equal to the navcircleTouch that we stored earlier - this is the touch object representing the player's finger that is touching the navcircle.

We then update the ship's position to the same position minus the difference between the last position of the touch and the new position of the touch - in other words, we change the ship's position by how much the touch has moved:

```
player.position.x = player.position.x - (lastTouchPosition!.x - touchPosition.x)
```

Note that this only applies for the horizontal axis, as we're only accessing the "x" variable within our position objects. It of course also contains a "y" variable - which we handle with the next line, which is just the same thing with "x" replaced with "y".

Finally, we store the touch position we've just used to update the ship's position in the lastTouchPosition variable so the difference that is taken the next time the function is run remains accurate.

The only remaining step is to ensure that the touch is removed when the player's finger is lifted off the screen. To do this, put the following code in both the touchesEnded and the touchesCancelled functions (the only difference between the two is that touchesEnded is when the player ends the touch and touchesCancelled is when the system ends the touch, for instance by creating a popup):

```
for touch in touches {
    if (moving && touch.isEqual(navcircleTouch)) {
        navcircleTouch = nil
        moving = false
    }
}
```

This code starts with the standard extended for loop to iterate through all of the touches, and triggers on the condition that both the player is (was) moving the ship and the current touch is the main one that is attached to the navigation circle. If that's the case, then we're sure that the finger that has been lifted is the finger that was on the navigation circle, at which point we can safely set navcircleTouch to nil (which just makes it nothing) and moving to false so that the

touchesBegan function can accept a new touch on the navigation circle when one happens.

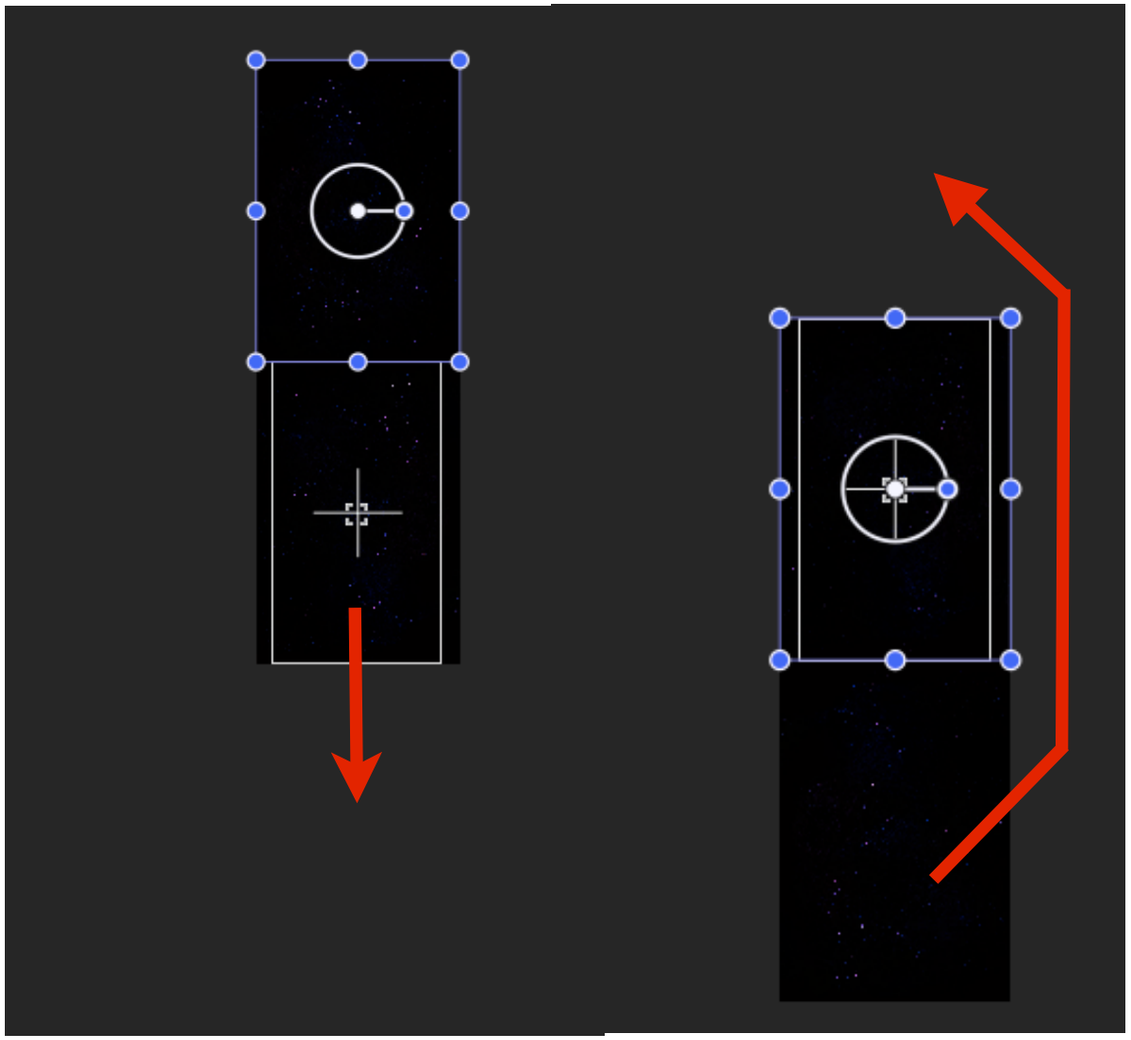
Congratulations! That's it for this marathon of a section. In the next section we will create a background and make it scroll continuously. In case anything isn't working how it should be, the code in this section will be reproduced on the next page.

```
import SpriteKit
import GameplayKit

class GameScene: SKScene {

    var foreground: SKNode!
    var background: SKNode!
    var player: SKSpriteNode!
    var navcircle: SKSpriteNode!

    var moving = false
    var navcircleTouch: UITouch? = nil
    var lastTouchPosition: CGPoint? = nil
    let gameWidth = CGFloat(850)
    let gameHeight = CGFloat(1334)

    override func didMove(to view: SKView) {
        isUserInteractionEnabled = true
        foreground = childNode(withName: "Foreground")!
        background = childNode(withName: "Background")!
        player = foreground.childNode(withName: "Player") as! SKSpriteNode
        navcircle = player.childNode(withName: "Navcircle") as! SKSpriteNode
    }
    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
        for touch in touches {
            let touchPosition = touch.location(in: self)
            let touchPositionInPlayerFrame = convert(touchPosition, to: player)
            if (navcircle.frame.contains(touchPositionInPlayerFrame) && !moving) {
                lastTouchPosition = touchPosition
                navcircleTouch = touch;
                moving = true;
            }
        }
    }
    override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
        for touch in touches {
            let touchPosition = touch.location(in: self)
            if (moving && touch.isEqual(navcircleTouch)) {
                //Move the player by as much as the finger's moved, then reset the baseline
                player.position.x = player.position.x - (lastTouchPosition!.x -
touchPosition.x)
                player.position.y = player.position.y - (lastTouchPosition!.y -
touchPosition.y)
                lastTouchPosition = touchPosition;
            }
        }
    }
    override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
        for touch in touches {
            if (moving && touch.isEqual(navcircleTouch)) {
                navcircleTouch = nil
                moving = false
            }
        }
    }
    override func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?) {
        for touch in touches {
            if (moving && touch.isEqual(navcircleTouch)) {
                navcircleTouch = nil
                moving = false
            }
        }
    }
    override func update(_ currentTime: TimeInterval) {
        // Called before each frame is rendered
    }
}
```

Section 4

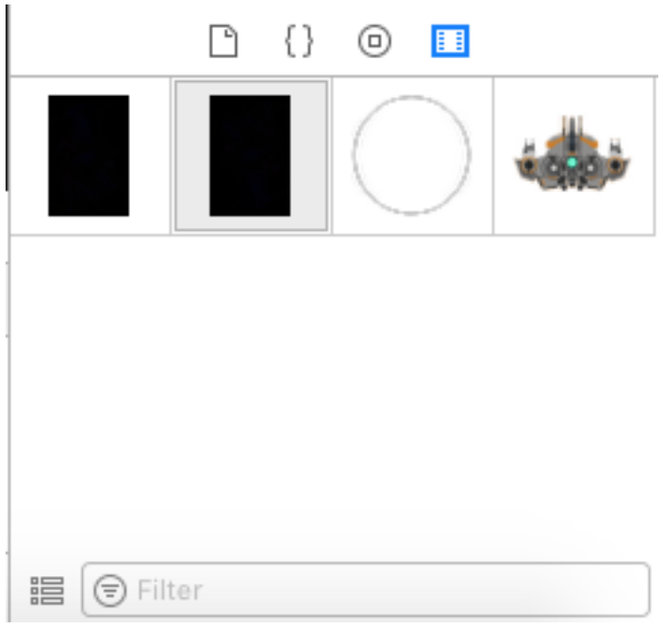
A Scrolling Background

In this section, we will add a scrolling background to our game, much like those you might see in Galaga or Space Invaders - a continuously moving backdrop of space to create the illusion that the ship is moving forwards at all times. One way to do this would be to create a massive level and move the ship through it, then create a camera that follows the ship. However, we shall use the much simpler strategy of creating two images in the background of the scene (one on top of the other and both with the exact dimensions on the screen), moving them continuously downward, then shifting each back up to the top of the screen when it moves off of the screen. They move at the same speed, so the second one is exactly below the screen the other should be perfectly fit to the screen, so that the shift from below the screen to above can occur without revealing any of the grey background in the scene. This movement is shown in the graphic on the following page:



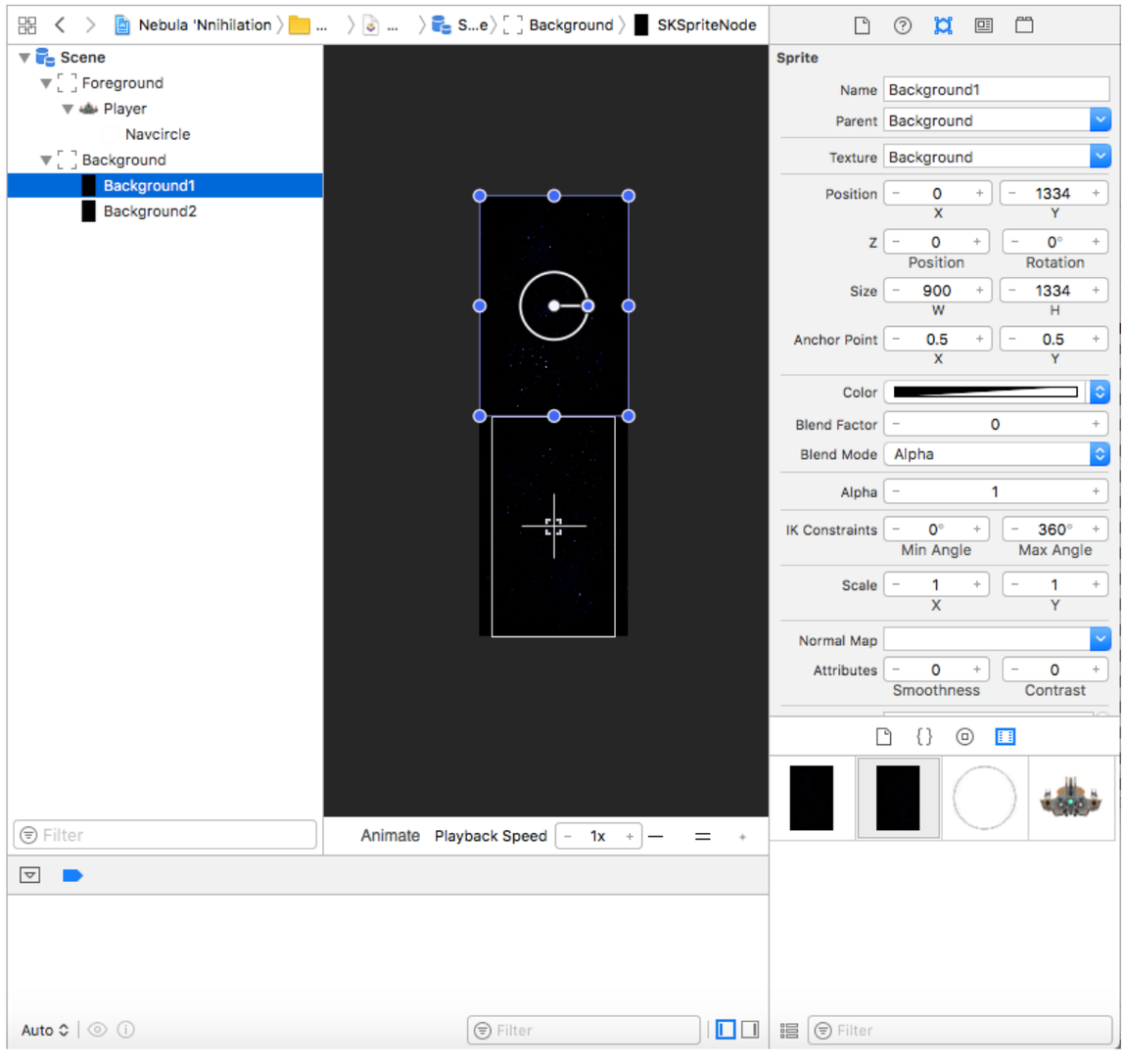
The two images we will be using for the background are in the assets library, labeled “Background1” and “Background2”. The only notable thing about these two images is that they are designed so that the bottom of the first image fits continuously with the top of the second, and the bottom of the second fits continuously with the top of the first. If you would prefer instead to make your own background images, this is the only condition you must meet

- aside from the fact that the width and height of the image should both exceed the dimensions of the iPhone screen (the images this tutorial uses are 900 x 1334). Add these two images to the Assets.xcassets file now, by dragging them from Finder into the window that appears if you double click on the Assets.xcassets file in the file picker on the left hand side of XCode's interface.



These assets will now appear in the element picker.

To begin working with these images, enter the scene editor (GameScene.sks) and click on the assets tab in the element picker, shown above. This should now contain the two background images we added. Drag them into the scene now. For the first, set the position to (0, 0) in the inspector above the element picker and name it "Background1". The other should be located at (0, 1334) and named "Background2". Because both of the images fit together top-to-bottom, it doesn't matter which begins on top. Drag these two elements to underneath the Background node in the element view to the element viewer on the left hand side of the screen. They should be at the same indentation level. When done, the scene editor should look like the following:



Finally, click on the parent "Background" node, and under Position, set the Z position to "-1". Z position defines how far something is from the screen - if something has a high Z-position, it will be rendered on top of things with a lower Z-

position. Because this node contains the background, we want it behind everything else - hence a negative Z-position. Any number below zero would work here, though, because in a 2D scene the only effect Z-position has is defining rendering order - it doesn't actually cause images to become smaller the farther they are away.

This concludes the visual setup - we've created our sprites in the editor and defined where we want them to start in the display. For complex things like defining movement, we need to write some code.

Switch to GameScene.swift, the code backend for the GameScene file. We need to create reference variables for our two new images, so create two new global variables of type SKSpriteNode just below the variables we made earlier:

```
var navcircle: SKSpriteNode! //Under this line, add these two:
var backgroundImage1: SKSpriteNode!
var backgroundImage2: SKSpriteNode!
```

In the didMove function, below our previous code, insert the following initialization code for those two new variables:

```
backgroundImage1 = background.childNode(withName: "Background1")
as! SKSpriteNode
backgroundImage2 = background.childNode(withName: "Background2")
as! SKSpriteNode
```

This searches through the children of the background node for two images named 'Background1' and 'Background2', then stores references to them in two variables called backgroundImage1 and backgroundImage2 so that we can manipulate them.

Next, we want to update the position of each background image every tick (see **Game Loop**). Just from testing numbers, it seems that a rate of 180 pixels of downward movement per second looks good, so we'll move each image down 3 pixels every tick in order to achieve that rate.

In the update() method - which is the game loop, and will execute sixty times per second - insert the following code:

```
override func update(_ currentTime: TimeInterval) {
    backgroundImage1.position.y -= 3
    backgroundImage2.position.y -= 3
    if (backgroundImage1.position.y < gameHeight * -1) {
        backgroundImage1.position.y = backgroundImage2.position.y + gameHeight
    }
    if (backgroundImage2.position.y < gameHeight * -1) {
        backgroundImage2.position.y = backgroundImage1.position.y + gameHeight
    }
}
```

The first two lines here use the '-=' operator, which subtracts the variable/number on the right from the variable on the left. In other words, the Y-position of both backgroundImage1 and backgroundImage2 has '3' subtracted from it each second. The unit here is pixels, so this should achieve the desired effect. Note that there is no need to update the screen, as simply modifying the position variable of the sprites will cause them to change position on the screen. Swift handles the behind-the-scenes stuff for us.

The second two if statements here check if the relevant image has passed below the screen - so, if the Y-position is below the game's height. Game height is being multiplied by -1 here because by default it represents the top of the screen, and we want to reference the bottom here. If the image *has*

passed below the screen, we simply set its new position to the other background's position plus the height of the image (same as the game height). In other words, we simply put it right on top of the other image. We do this check for both images, which is why there are two if statements.

This is all the code required to get the backgrounds scrolling - if you run the project, you should see our ship is now moving nicely through the galaxy.

The updated GameScene file from this section is reproduced below. This should, like the last reproduction, be copy-paste capable for those having issues with their own code.

```
import SpriteKit
import GameplayKit

class GameScene: SKScene {

    var foreground: SKNode!
    var background: SKNode!
    var player: SKSpriteNode!
    var navcircle: SKSpriteNode!
    var backgroundImage1: SKSpriteNode!
    var backgroundImage2: SKSpriteNode!

    var moving = false
    var navcircleTouch: UITouch? = nil
    var lastTouchPosition: CGPoint? = nil
    let gameWidth = CGFloat(850)
    let gameHeight = CGFloat(1334)

    override func didMove(to view: SKView) {
        isUserInteractionEnabled = true
        foreground = childNode(withName: "Foreground")!
        background = childNode(withName: "Background")!
        player = foreground.childNode(withName: "Player") as! SKSpriteNode
        navcircle = player.childNode(withName: "Navcircle") as! SKSpriteNode
        backgroundImage1 = background.childNode(withName: "Background1") as!
SKSpriteNode
        backgroundImage2 = background.childNode(withName: "Background2") as!
SKSpriteNode
    }
    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
        for touch in touches {
            let touchPosition = touch.location(in: self)
            let touchPositionInPlayerFrame = convert(touchPosition, to: player)
            if (navcircle.frame.contains(touchPositionInPlayerFrame) && !moving)
{
                lastTouchPosition = touchPosition
                navcircleTouch = touch;
                moving = true;
            }
        }
    }
    override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
        for touch in touches {
            let touchPosition = touch.location(in: self)
            if (moving && touch.isEqual(navcircleTouch)) {
                //Move the player by as much as the finger's moved, then reset
the baseline
                player.position.x = player.position.x - (lastTouchPosition!.x -
touchPosition.x)
                player.position.y = player.position.y - (lastTouchPosition!.y -
touchPosition.y)
                lastTouchPosition = touchPosition;
            }
        }
    }
    override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
        for touch in touches {
            if (moving && touch.isEqual(navcircleTouch)) {
                navcircleTouch = nil
                moving = false
            }
        }
    }
    override func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?)
{
        for touch in touches {
            if (moving && touch.isEqual(navcircleTouch)) {
                navcircleTouch = nil
                moving = false
            }
        }
    }
    override func update(_ currentTime: TimeInterval) {
        // Called before each frame is rendered
        backgroundImage1.position.y -= 3
        backgroundImage2.position.y -= 3
        if (backgroundImage1.position.y < gameHeight * -1) {
            backgroundImage1.position.y = backgroundImage2.position.y +
gameHeight
        }
        if (backgroundImage2.position.y < gameHeight * -1) {
            backgroundImage2.position.y = backgroundImage1.position.y +
gameHeight
        }
    }
}
```

Section 5

Creating Enemies

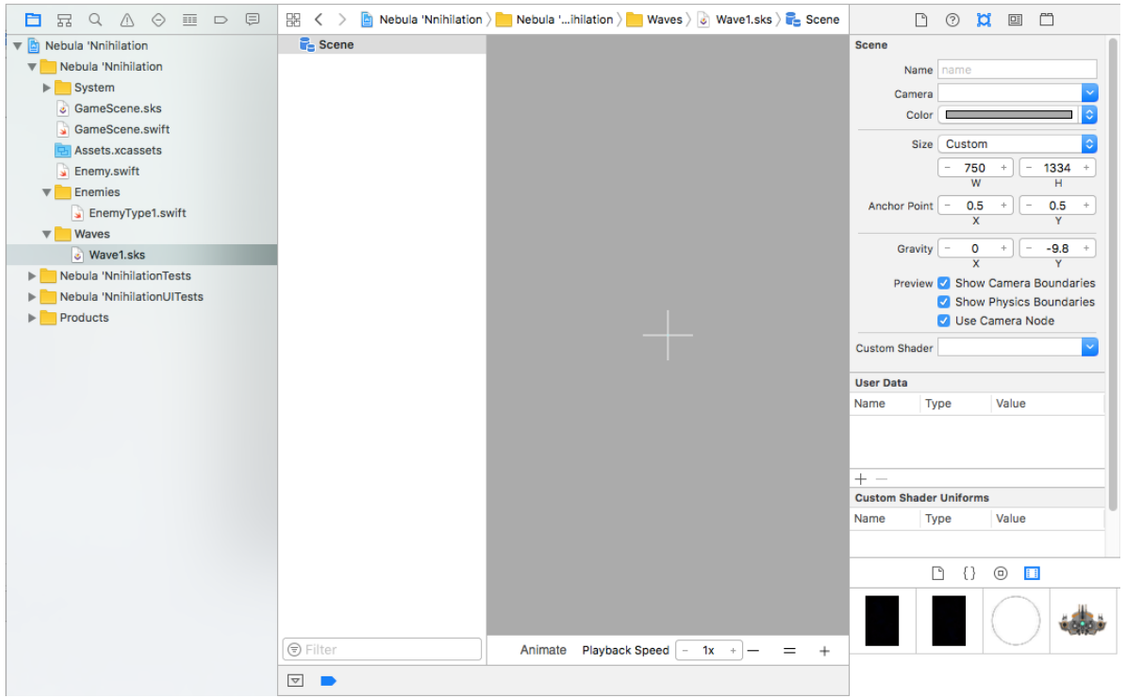
Enemies are critical to a shooter game - without them, the player is just firing blankly into space. Additionally, it's important to add enemies before we give the player the ability to fire their weapon - without something to shoot, it would be difficult to test whether or not the bullets work. This game will have three different types of enemy:

- Non-aggressive enemy that doesn't shoot.
- Small enemy that shoots three bullets in a cone.
- Large enemy that shoots bullets moderately rapidly at random angles within a cone.

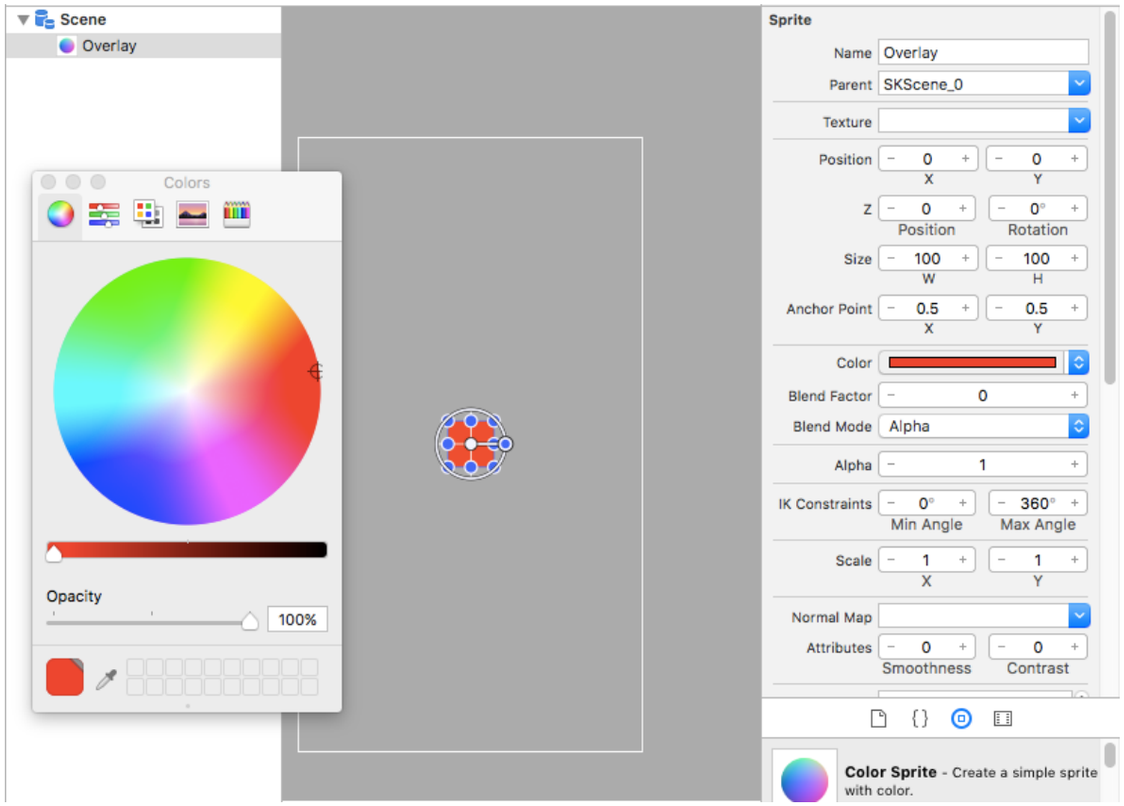
However, it will be fairly easy to change these patterns or make new enemies once we've implemented them because of the strategy we'll be using. We also want to be able to easily define patterns of enemies that spawn at regular periods. It should be easy to create new waves, and ideally it would be a visual process. This is possible in Swift 3 because of the addition of the scene editor - we can simply create a scene for each wave, then drag and drop enemies into it, then instead of spawning individual enemies, spawn an entire wave by placing our smaller 'wave' scenes into our main scene via code.

Let's get started.

In order to begin adding elements to a scene, we must first create a new scene. Create a new group in your main project directory and call it “Waves”. Click File > New > File > SpriteKit Scene, and name the new scene “Wave1”.



Into this scene, drag an ‘Color Sprite’ from the third tab in the element picker. Name this node ‘Overlay’, and set its position to (0, 0). We don’t want the red square in the center of the screen - the only reason we’re using a color sprite here instead of a node is because if you use an ‘Empty Node’ instead, adding it to the GameScene in code will by default make all of the children invisible. We need the transparency controls in the color sprite type. To make the red bit go away, click on the red “Color” option - not the dropdown selector, but on the actual color - to bring up the Colors selector.

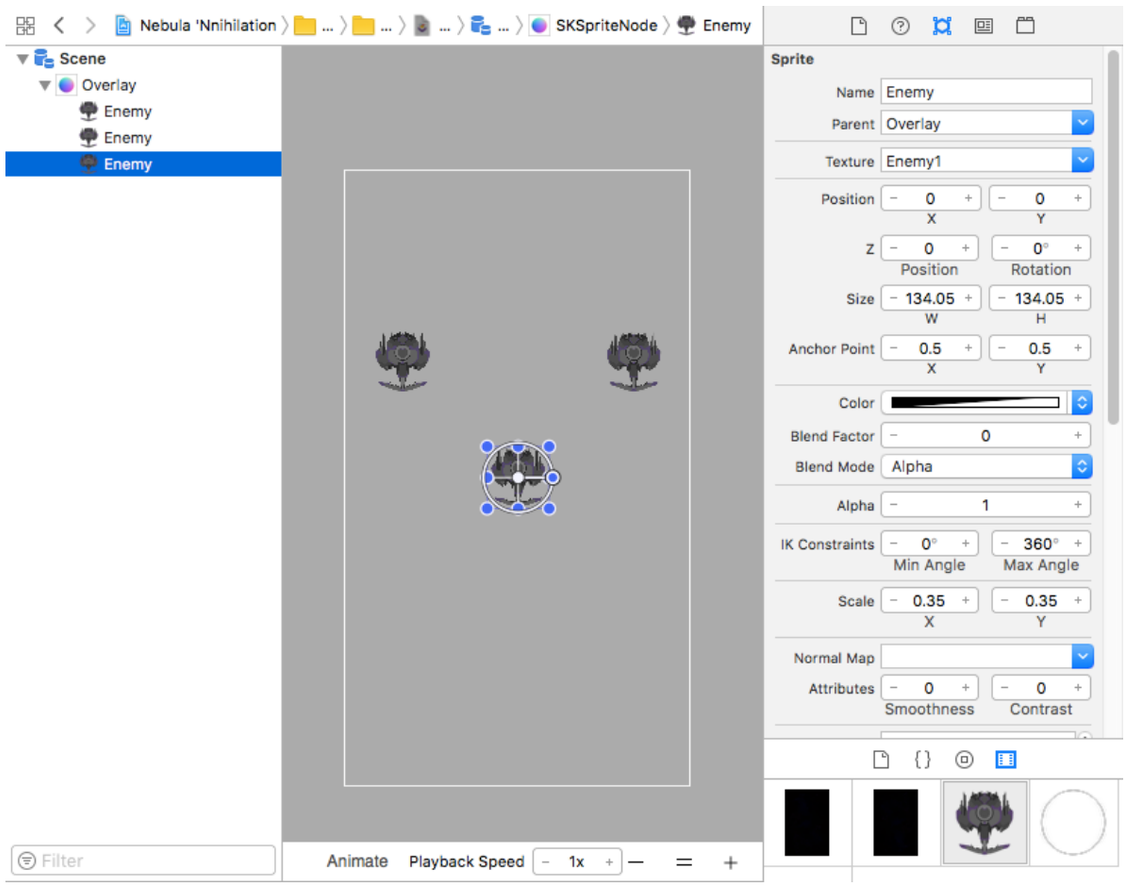


Drag the opacity slider down to zero, and the color should disappear. Note that setting the Alpha to zero in the properties to the right would also make the sprite disappear, but would also by default set all children to be invisible too - which is something we are trying to avoid. This way makes a visible sprite, but one that is fully transparent. This way the children will be visible.

Now, open your Assets.xcassets file and drag in “Enemy1.png”. This should now show up in your element picker in the bottom right of the screen - drag three of them into the scene - set the Scale for each to (0.35, 0.35) to make them reasonably sized. Name them all “Enemy”, and drag them in the element view to a child position under the

‘Overlay’ node. You may position them in the scene however you like - note that to create symmetry, it’s better to drag them to around the position you want and then do fine-tuning by adjusting the position property in the properties view to the right of the screen. The center of the screen is X = 0, Y = 0, so if you wish to mirror something on the screen, set the X-value of one to the inverse of the X-value of the other. Also note that instead of dragging a new node into the scene and setting the properties, a node can be duplicated by selecting it in the element view and selecting File > Duplicate.

We’ve arranged our three ships at (0, 0), (-250, 250), and (250, 250) for the tutorial.



Before we go into code, select the “Overlay” node and set the Y to 1334 under the position. This will put the starting position of the wave above the screen, so that when a wave spawns it doesn’t appear on the screen but instead starts above the screen and moves down.

Let’s dip into code now and create code to spawn these waves regularly. The method we will be using to do so is by counting the frames (there should be sixty per second) and spawning a wave every ten seconds - in other words, every 600 frames. Let’s begin by adding a way to count frames. Add the following variable to the global variable declarations:

```
var frameCount = 0
```

This will keep track of the current frame - it is set to zero here because global variables are set at the beginning of the program, and we want the frame count to *start* at zero. At the bottom of the update() function, add:

```
frameCount += 1
```

This will add one to the frameCount variable every time a new frame is rendered by the update() function.

Now, we want to spawn a new wave every ten seconds. First, create a global variable to store a representation of the wave we just created in the same section as the other globals from earlier in the tutorial:

```
var enemyWaveTemplate: SKSpriteNode!
```

Then initialize this variable at the bottom of the didMove() function:

```
let wave1 = SKScene(fileNamed: "Wave1")
enemyWaveTemplate = wave1!.childNode(withName: "Overlay") as!
SKSpriteNode
```

This is a more lengthy initialization than the other variables, and here's why: we must first create an object to represent the Scene file that the node we want to fetch is stored in. Since the file is separate from the GameScene file, we can't simply use the default childNode() function like we've been using, hence the extra line. We then can initialize our variable enemyWaveTemplate to the node named "Overlay" within that scene by calling the childNode function.

At the top of the update() function, add the following code:

```
let waveSpawnInterval = 10 //seconds
if (frameCount % (60 * waveSpawnInterval) == 0) {
    let tempNode = enemyWaveTemplate.copy() as! SKSpriteNode
    let moveDownwardsAction = SKAction.move(by: CGVector(dx: 0, dy:
gameHeight * -2), duration: 20)
    let moveThenDelete = SKAction.sequence([moveDownwardsAction,
SKAction.removeFromParent()])
    tempNode.run(moveThenDelete)
    foreground.addChild(tempNode)
}
```

This is fairly dense, so lets step through it line by line.

The first line creates a variable only used on the next line in order to increase readability, where the number of seconds per wave is defined. This could be omitted and the usage on the next line replaced with a number, but this way is more legible.

The if statement on the second line checks if the current frame count is evenly divisible by an expression representing the number of frames in however many seconds the spawning interval is set to. In the default case, it checks if the frame count is divisible by 600. This will return true once every 600 frames, and so the effect of this is that the code in the block will execute once every ten seconds assuming the device is able to render the game at 60fps (and if it is running slower, the waves will spawn slightly slower to account for that).

The line "`let tempNode = enemyWaveTemplate.copy() as! SKSpriteNode`" creates a variable representing a copy of the node we created earlier that is an instance of the "Overlay" layer in the "Wave1" scene. We want a copy here because we want to have multiple on the screen at the same time - this is the equivalent of simply creating a new node with all of the features of the "Overlay" node and containing all of the same children (the enemy ships) in the same configuration. This way we can churn a new one out every ten seconds without worrying about how that affects the previous wave, or even have multiple waves on-screen at the same time depending on spawn intervals.

"`let moveDownwardsAction = SKAction.move(by: CGVector(dx: 0, dy: gameHeight * -2), duration: 20)`" creates a new **SKAction** based on the "move" template within the SKAction class. The "move" template takes two parameters: a vector representing how much to move the node it's applied to, and a duration

representing how long the movement should take. In this case, we feed it a vector with parameters (x: 0, y: gameHeight * -2), which means that we don't want it to move on the x-axis at all and want it to move two screens downward on the y-axis. The center of the wave starts half of a screen-height above the screen and will pass off of the screen when it is half of a screen-height below the screen, so the total movement here is two screen-heights in the negative direction, hence multiplying the screen height by negative two here. The duration is 20 seconds, which is an arbitrary value but which has been found to be slow enough that the game isn't too difficult.

"`let moveThenDelete = SKAction.sequence([moveDownwardsAction, SKAction.removeFromParent()])`" adds that action to a new sequence of SKActions, which is essentially a list or an array of actions in which the first will be executed, and after finishing the second will be executed, and so on. The sequence() function takes an array of SKActions as a parameter, and in this case we give it the moveDownwardsAction we created earlier followed by the removeFromParent() action, which will remove the node from the screen. To put that in plain English, this sequence moves the wave down two screen heights over 20 seconds and then deletes it.

"`tempNode.run(moveThenDelete)`" runs the SKAction on the new copied wave node we've created - which simply applies the actions we've defined to the node. So now, it will move downwards and then delete itself twenty seconds later starting immediately after it is created.
Finally, `foreground.addChild(tempNode)` adds the new node, actions and all, to the foreground layer in the main GameScene so that it will now be a visible part of the scene.

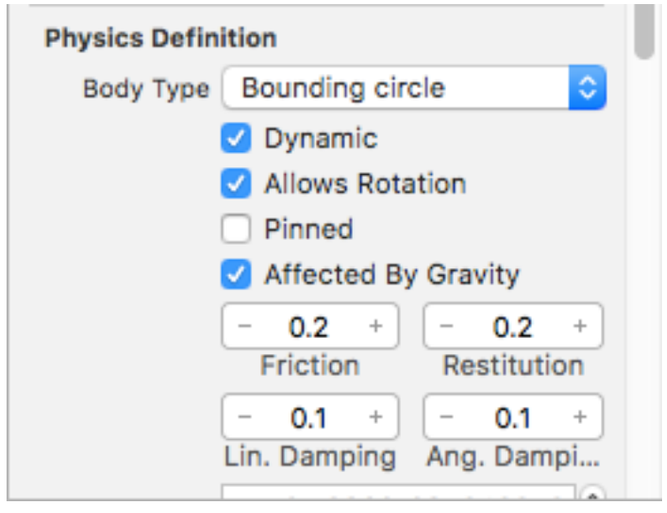
This wraps up wave spawning - running the project now should cause waves to begin appearing after a few seconds. Note that the node count in the bottom right hand corner of the screen stabilizes at about fifteen - this is because we delete the waves after they pass off of the screen. We do this because otherwise the node count would build up as the program continued running, and eventually the game would become so slow it would be unplayable because the game engine would have to continue moving and performing physics calculations on hundreds of nodes not visible on the screen anymore. It's good practice to delete anything that goes off the screen that is no longer in use - more on methods to do this in the next section, because sometimes you don't know exactly when an object will move off of the screen.

Section 6

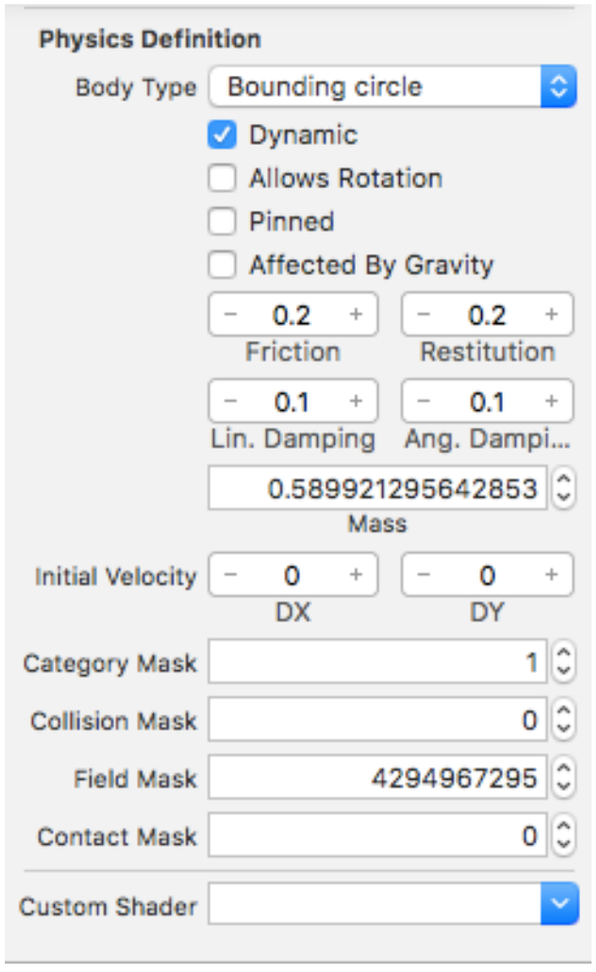
Collisions

Though we have enemies that move down the screen, they don't yet interact with the player in any way. We want the game to restart when the player touches an enemy, which means we'll have to implement collision detection. Fortunately, Swift makes this relatively easy.

To begin, head into GameScene.sks and click on the Player node. In the properties view to the right side of the screen, scroll down until you see the "Physics definition" section. Change the body type from "None" to "Bounding circle". This will add the player to Swift's physics engine.

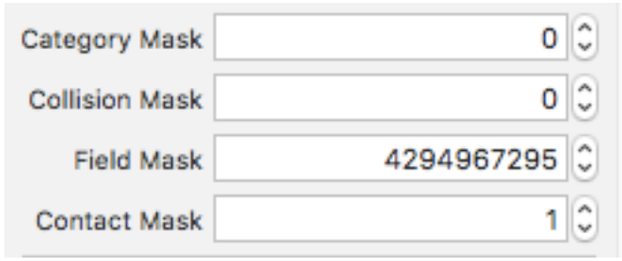


We don't actually want any of the physics-related functions of the physics engine, such as being affected by gravity or reacting when bumping into things. All we want is for the physics engine to run code when the player contacts an enemy. To accomplish this, uncheck the "Allows Rotation and "Affected by Gravity" checkboxes, then change "Category Mask" to 1, "Collision Mask" to 0, and "Contact Mask" to 0. Setting the collision mask to 0 will disable physics-based collisions, which make things that bump into each other collide realistically. Because we just want to kill the player on colliding with anything, we don't care about fancy effects. The Category and Collision masks define when the physics engine will notify the code upon noticing a collision. The value of zero under "Contact mask" means that the player will not fire a physics signal upon contacting anything, but the category mask of 1 signals that anything defining a contact mask of 1 will fire a signal upon coming into contact with the player. We'll give the enemies a contact mask of 1.



In "Wave1.sks", do the same to each of the three enemies, but instead give them a category mask of 0 and a contact mask of 1. This means nothing will signal upon

contacting them, but they will signal upon contacting anything with a category mask equal to “1” - in other words, the player.



The category and contact masks to use get complicated when programming collision detection in more than one category - for instance, if you have three objects and you want to do a rock-paper-scissors style collision detection where each only collides with one of the other three. Look into the Swift documentation for information on how to do this if you’re interested, but it is moderately difficult. Fortunately, most games can work around this.

Now that contact testing is enabled and the enemies will signal the physics system when they come into contact with the player, we need to receive and handle that signal. This means turning the main GameScene into a SKPhysicsContactDelegate, which sounds complicated but basically means a class with a function handling physics contacts. In the class declaration of GameScene.swift, add SKPhysicsContactDelegate to the list of extended classes and implemented protocols:

```
class GameScene: SKScene, SKPhysicsContactDelegate {
```

At the bottom of the didMove function, add the following line to setup the physics world using the GameScene class as the physics contact handler:

```
physicsWorld.contactDelegate = self as SKPhysicsContactDelegate
```

This just tells the physics engine where to find the function to call when two objects come into contact with each other.

Now, we have to actually implement the protocol SKPhysicsContactDelegate, which we set up the class to implement earlier. Add the following function at the bottom of the class before the closing bracket:

```
func didBegin(_ contact: SKPhysicsContact) {

}
```

This is the function the physics engine will call when two objects come into contact. It hands you a SKPhysicsContact variable representing the contact - this variable also contains references to the two nodes that are contacting each other. Let’s create shortcut variables to those two nodes now inside the new didBegin function:

```
if let body1 = contact.bodyA.node {
    if let body2 = contact.bodyB.node {

    }
}
```

The nodes are stored within contact.bodyA and bodyB, and this simplifies that and also unwraps the optional. It does nothing if either of the nodes don’t exist - hence the if let.

We want to restart the game on collision, so let's first define a function that restarts the game (outside of didBegin, before closing brackets):

```
func restart() {
    player.position.x = 0
    player.position.y = -300
    foreground.removeAllChildren()
    foreground.addChild(player)
    navcircleTouch = nil
    moving = false
}
```

This function does three things: Resets the player's position to its value at the beginning of the game, removes all objects from the foreground and then re-adds the player (the only object that should be on the screen at the start of the game at the moment), and removes any active touches.

Now, we want to react to collisions between the player and the enemy. In the didBegin function from earlier, inside both sets of curly braces, add the following:

```
if (body1.name == "Player" && body2.name == "Enemy"
            || body2.name == "Player" && body1.name == "Enemy") {
    restart()
}
```

The condition in the if statement checks whether either the first node is named "Player" AND the second is named "Enemy, or the first node is named "Enemy" AND the second is named "Player". So in other words, the statement will execute if one of the nodes is "Player" and the other is "Enemy". Inside the statement, we simply call our restart function from earlier.

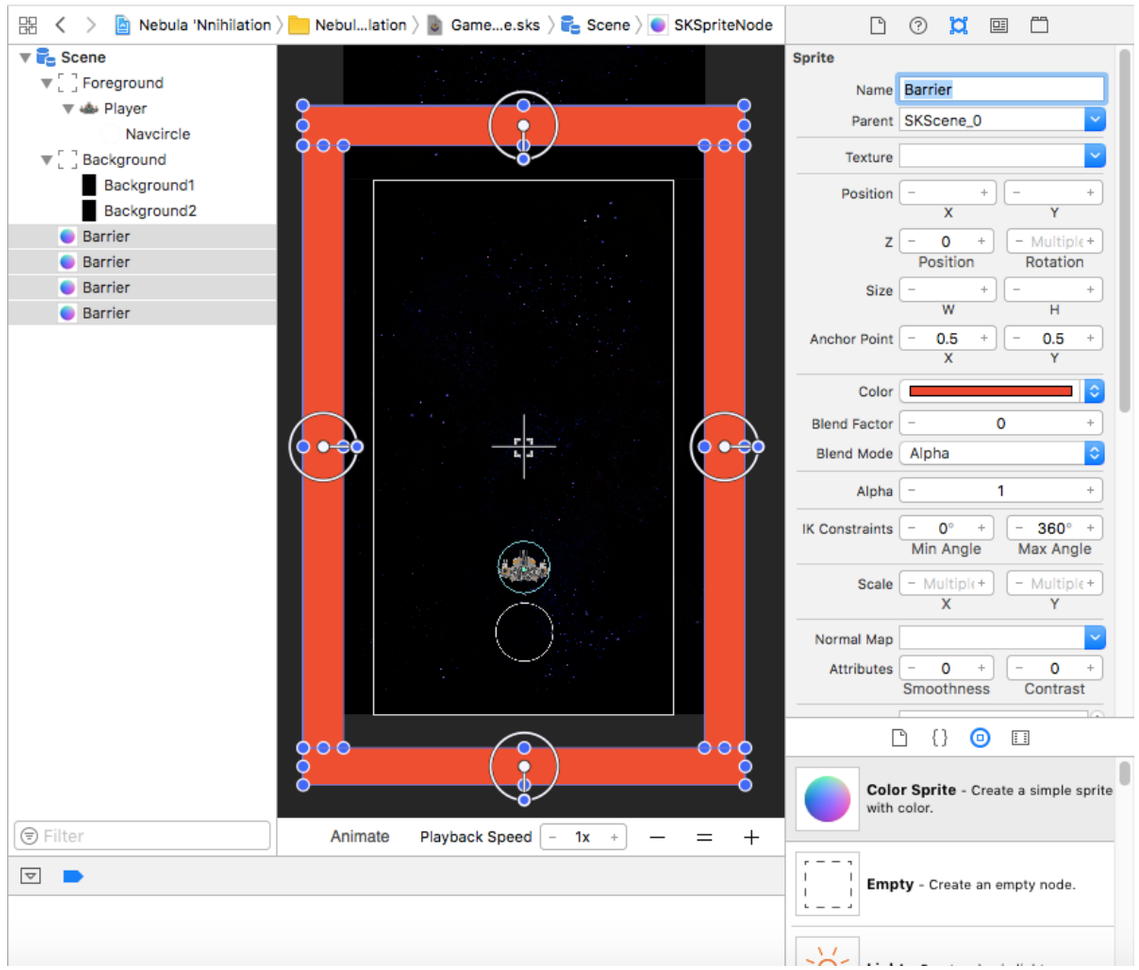
This should cause the game to restart if the player hits an enemy. In the next section, we'll begin building the prerequisites for weapons to exist.

Section 7

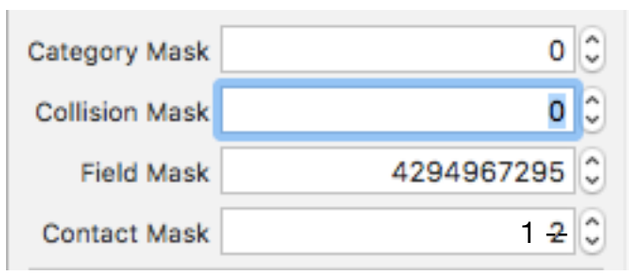
Deleting Offscreen Bullets

In the next section, we'll be making the player and enemies fire bullets. We want a way to delete them when they pass off the screen - hundreds of off-screen nodes being handled by the physics engine can slow the app down to a snail's pace and crash it in very little time. To do this, head into GameScene.sks and drag four ColorSprites onto the screen from the element picker. Size and position them so they surround the screen completely, but don't touch the edges of the screen (otherwise they might delete bullets that are still on the screen). Name each "Barrier". These nodes should be children of the scene - not the background or foreground, because they're not actually part of the visible section of the screen. They can also be left at the default bright red color because they'll never be shown on screen.

For each - and note here that by clicking on the first node in the element viewer and then shift-clicking on the last one you can select all of them and modify all of them simultaneously to save time - add a PhysicsBody of the rectangle type. Uncheck "Affected By Gravity" and "Allows Rotation" and check "Pinned".



Under the Masks section, set the following:



Collision mask is set to zero so it doesn't physically run into anything, and category mask is zero because nothing should

be able to contact it. Instead, it contacts anything with category mask ‘1’, as defined in the Contact Mask parameter. When we make our bullets, we’ll give them a category mask of 1.

Section 8

Weapons

In this section, we'll finally get our player and enemies firing. Drag into your project from the project assets folder we've provided the file "PlayerBullet.sks". This is a simple particle system created with File > New > File > SpriteKit Particle File, using the "Spark" preset. You can mess around and make your own if you prefer. To make this work, also drag "Spark.png" from our provided assets folder into Assets.xcassets. This system will be just added to the scene as a sprite, and will represent your bullets.

First, let's create a layer to store our new bullets in GameScene.sks. Add a new Empty Node to the children of Foreground - it should be the same indentation level as the player sprite. Call it "Bullets". Initialize this in the code (now to GameScene.swift) the same way as previously - create a global variable:

```
var bulletLayer: SKNode!
```

and initialize it in the didMove function:

```
bulletLayer = foreground.childNode(withName: "Bullets")!
```

We've added a node to the foreground, now, so we need to add it back to the foreground in the restart() function where, if you remember, we delete all children of the foreground node in order to restart the game. Under the line "foreground.addChild(player)", add the following code:

```
foreground.addChild(bulletLayer)
```

This should ensure that bullets don't stop working when the game restarts.

We'll begin coding by creating a new Swift file in the root directory of your project. Select File > New > File, and create a Swift file. Name it "Weapon".

This class will be a protocol, defining what variables each of our individual weapon classes should contain:

```
import SpriteKit
protocol Weapon {

    var damage: Int { get }
    var force: Double { get }
    var fireRate: Int { get }
    var filename: String { get }
    var bulletName: String { get }
    var categoryMask: Int { get }
    var contactMask: Int { get }
    var position: CGPoint { get }

}
```

Going through this line by line, the protocol requires the following variables:

- How much damage the weapon should do
- The force it should be fired with (speed of the bullet)
- How many frames should elapse between fires
- The name of the graphic file (We'll create this in a bit)
- The node name to attach to created bullets
- The category mask for collision detection purposes
- The contact mask for collision detection purposes

Let's create classes that extend this protocol now. The first priority will be the player's weapon, so that there is a mechanism in place to both kill enemies and generate score. We'll worry about enemy weapons later, but it will use a similar protocol.

Now, we'll design our weapons. In your file structure, create a new group called "Weapons", and inside it, create a Swift file called "PlayerWeapon". Inside the class, implement the variables as follows:

```
import SpriteKit
class PlayerWeapon: Weapon {

    var damage = 3
    var force = 0.3
    var fireRate = 6
    var filename = "PlayerBullet"
    var bulletName = "PlayerBullet"
    var categoryMask = 1
    var contactMask = 0
    var position = CGPoint(x: 0, y: 0)
}
```

Note that all of these values are not for the weapon itself, but for the bullets it creates.

The values for damage, force, and fireRate are all arbitrary values that have been found to work well. Damage is in points (enemies are currently set up to have 125), force is in N/s, and fireRate is in frames. The category and contact masks are set up so that the bullets will not contact anything, but will be triggered by anything with a contact mask of 1 - the barriers and the enemies. The position will be the position that bullets are created relative to the player.

Now, in the main GameScene.swift class, add two new global variables to represent the two player weapons:

```
var playerWeapon1: PlayerWeapon = PlayerWeapon()
var playerWeapon2: PlayerWeapon = PlayerWeapon()
```

Initialize the positions to the following at the very bottom of the didMove function:

```
playerWeapon1.position = CGPoint(x: 35, y: 5)
playerWeapon2.position = CGPoint(x: -35, y: 5)
```

CGPoint is just a point on the screen, and since this will be relative to the player, we just define them as 5 pixels above and 35 pixels to the left and right of the player's center.

Let's start firing bullets. Create a new function somewhere in GameScene.swift:

```
func fireWeapon(weapon: Weapon, senderPosition: CGPoint) {
    let bullet: SKEmitterNode = SKEmitterNode(fileNamed:
weapon.filename)!
    bullet.position = CGPoint(x: weapon.position.x + senderPosition.x,
                              y: weapon.position.y + senderPosition.y)
    bullet.name = weapon.bulletName
    bullet.physicsBody = SKPhysicsBody(circleOfRadius: 1)
    bullet.physicsBody?.affectedByGravity = false
    bullet.physicsBody?.categoryBitMask = UInt32(weapon.categoryMask)
    bullet.physicsBody?.contactTestBitMask = UInt32(weapon.contactMask)
    bulletLayer.addChild(bullet)
    bullet.physicsBody?.applyImpulse(CGVector(dx: 0, dy: weapon.force))
}
```

That's a doozy of a code block. Unfortunately, you won't get away with just copy-pasting it into your code. Let's go through it step by step.

The first line is similar to things we've done earlier, where we use the mod function on frameCount and fireRate to check if the frameCount is divisible by the fireRate. This will return true once every certain number of frames (equal to the fireRate).

```
let bullet: SKEmitterNode = SKEmitterNode(fileNamed: weapon.filename)!
bullet.position = CGPoint(x: weapon.position.x + senderPosition.x,
                          y: weapon.position.y + senderPosition.y)
bullet.name = weapon.bulletName
```

The first line creates an SKEmitterNode - which is just a sprite that can emit particles. We feed this our filename for the particle scene file we created/dragged into our project earlier. We then set this bullet's position to the shooter's position plus the gun's position, which should be relative to the shooter position. This is the bullet's starting position - where the gun is. We name the bullet whatever the weapon specifies it should be named - in this case, "PlayerBullet".

```
bullet.physicsBody = SKPhysicsBody(circleOfRadius: 1)
bullet.physicsBody?.affectedByGravity = false
bullet.physicsBody?.categoryBitMask = UInt32(weapon.categoryMask)
bullet.physicsBody?.contactTestBitMask = UInt32(weapon.contactMask)
```

This code deals with the physics engine. First, we set the physicsBody property of the node. All nodes have a physicsBody component - if you create one, it adds the node to the physics engine. We create a physics body here with the default properties, except we set the single parameter - circleOfRadius - to 1. This defines how large the hitbox is, or how large the circle that it will detect collisions within is. In this case, we want one pixel. We then disable gravity, and set our category and contact masks to the ones defined in the PlayerWeapon class. They're cast to a UInt32 here because the physics engine requires that type specifically instead of a standard Integer type. The reason is largely irrelevant in this context, but important to understand for very complex collision detection. Most games don't do that, so we can ignore it here.

We then add our bullet to the bullet layer we created earlier:

```
bulletLayer.addChild(bullet)
```

Physics actions cannot be performed until things have been added to the scene, so we can give it movements now.

```
bullet.physicsBody?.applyImpulse(CGVector(dx: 0, dy: weapon.force))
```

This gives the bullets an impulse. This is measured in Newton-seconds, which is a quantity covered in physics classes. Just know that for objects with the default mass, 1 is *really fast* and 0.2 is still somewhat zippy. We give impulse in the y-direction only equal to the quantity we specified earlier. Because the bullets are at the player's position, we don't need to worry about the x-axis.

Now, let's actually fire the player's weapons. Add the following code to the top of update():

```
if (frameCount % playerWeapon1.fireRate == 0) {
    fireWeapon(weapon: playerWeapon1, senderPosition: player.position)
    fireWeapon(weapon: playerWeapon2, senderPosition: player.position)
}
```

In other words, if the frameCount is divisible by the fireRate (should be once every ‘fireRate’ frames), call the fireWeapon `function` for each weapon.

Now, in order to remove the bullets when they hit the barrier, add this code to your collision detection code (in the didBegin() function):

```
if (body1.name == "PlayerBullet" && body2.name == "Barrier") {
    body1.removeFromParent()
} else if (body2.name == "PlayerBullet" && body1.name == "Barrier")
{
    body2.removeFromParent()
}
```

This code tests for collisions between the player bullets - which we gave a node name of “PlayerBullet” in our generation code - and the barriers which we created in the last section. It then removes whichever body is the bullet - the reason there are two separates tests is because either body1 or body2 could be the bullet, so we have a statement checking for either case.

The last consideration is getting rid of all the bullets on the screen when the game restarts. To do this, add the following code to the restart() function (the first greyed-out line should not be copied - it’s just the line you should paste the colored line underneath):

```
foreground.removeAllChildren()
bulletLayer.removeAllChildren()
```

We add all of our bullets to the bulletLayer, so this just clears that layer when the game restarts.

This whole section’s code should fire some bullets at the enemies. They won’t do anything yet, so we’ll make them collide with things in the next section.

Section 9

Collisions, part 2

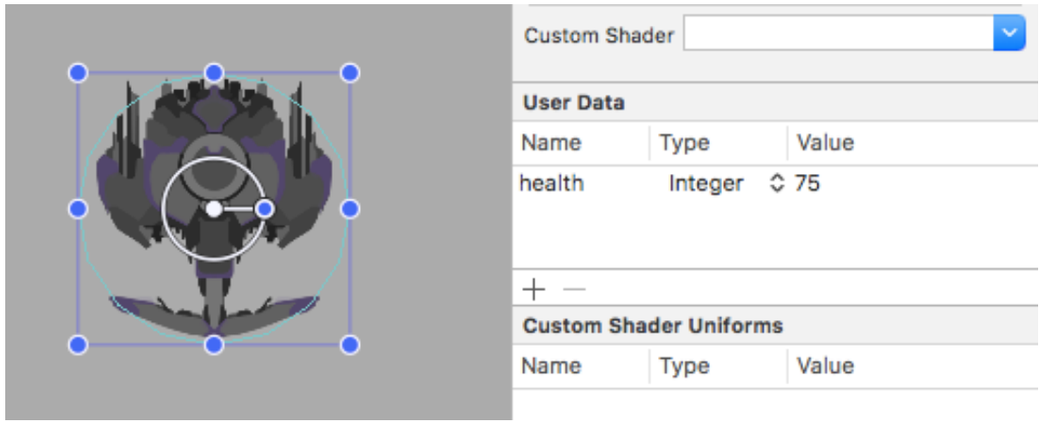
Electric Boogaloo

The first thing we have to do is come up with a way to track the health of each of the enemies. There are a million different equally valid ways to do this, but the constraint we want to impose is that we don't have to create an individual variable for each enemy. We want it to be a part of each enemy to begin with, so we don't have to deal with like nine different enemy variables within our main code and also have to add a new variable every time we decide a wave needs more enemies - though this would probably work, despite being a pain to deal with.

Because each of the enemies we've created are just default SpriteNodes with a picture of an enemy and the name "Enemy", we have to find a way to attach values to an SKSpriteNode unless we want to make our own custom type for enemies. There's an XCode bug associated with assigning custom types to scene elements, so in this project we've elected to go with a method called User Data. However, for reference, custom types would be the most efficient and sensible way to do this if the bug didn't exist. Hopefully, this will be fixed by the release of iOS 11.

User Data is a way of attaching variables to scene elements without making them part of a custom type. There's no guarantee that they exist when retrieving them, so that must be hardcoded - if you forget to add the health variable to an enemy, for instance, and the program tries to subtract health from it, there will be an error. Be diligent about adding the same user variables to objects that you expect to be the same, whatever distinction you may be making with object types. In this case we use names, so anything named "Enemy", for instance, should have the same user variables attached. The values of those variables don't necessarily have to be the same though.

Now, we'll set the health variables for the three enemies. To add a user data property, click on the node and scroll down in properties until you find 'User Data'. Click the plus button. Name your variable "health". No caps - variable names are case sensitive, so be careful. Set the value to 75. Repeat this specifically for each enemy.



In this section, we're going to make the bullets kill the enemies - really the core feature of any respectable shooter. The key to this will be the collision detection method we worked on earlier - we can simply add functions to it to handle our bullets, which by virtue of their masks should collide with the other elements in the scene.

First, add a global variable to record the player's score as enemies are killed:

```
var score = 0
```

Then, copy this into didBegin():

```
if (body1.name == "PlayerBullet" && body2.name == "Enemy") {
    var health = body2.userData?.value(forKey: "health") as! Int
    health -= playerWeapon1.damage
    body2.userData?.setValue(health, forKey: "health")
    if (health <= 0) {
       body2.removeFromParent()
    }
    body1.removeFromParent()
    score += playerWeapon1.damage
} else if (body2.name == "PlayerBullet" && body1.name == "Enemy") {
    var health = body1.userData?.value(forKey: "health") as! Int
    health -= playerWeapon1.damage
    body1.userData?.setValue(health, forKey: "health")
    if (health <= 0) {
        body1.removeFromParent()
    }
    body2.removeFromParent()
    score += playerWeapon1.damage
}
```

These two if statements are the same - the only thing that's different is one deals with PlayerBullet as the first body and the second deals with PlayerBullet as the second body (bullets can collide with enemies or enemies can collide with bullets, but we need to handle both in the exact same way).

Specifically, the collision code first fetches the health property by accessing the userData variable within whichever body is the enemy. This is a dictionary - we find the key for the value "health", which should be the current value of our custom health variable. It's an integer, so we cast it to one. We then subtract the damage, and set the new value. If it has passed below or equal to zero, we delete whichever body is the enemy. Regardless, we then delete whichever body is the bullet. Finally, we add the damage done by the weapon to the score. We can use "playerWeapon1" regardless of which weapon fired the bullet only because both weapons are exactly the same and do the same damage.

Section 10

Enemy Weapons

To make the enemies fire at you, we can use the firing code we wrote earlier. First, create an ‘EnemyWeapon1’ Swift file in the same group as the PlayerWeapon file, and copy the following code over to it - which is just a copy of the PlayerWeapon file with some variables changed:

```
import Foundation
import SpriteKit

class EnemyWeapon1: Weapon {

    var damage = 3
    var force = -0.15
    var fireRate = 30
    var filename = "CannonBullet"
    var bulletName = "EnemyBullet"
    var categoryMask = 1
    var contactMask = 1
    var position = CGPoint(x: 0, y: 0)

}
```

Feel free to mess with these values on your own. The most significant change here is that it uses a filename called “CannonBullet”, which you should now drag from the assets folder into your project (the project root directory, not the Assets.xcassets file).

In the update() method in GameScene.swift, add the following code to the top:

```
//Create placeholder enemy weapon so we can access its properties
let enemyWeapon1 = EnemyWeapon1()
```

```
for child in foreground.children {
    if (child.name == "Overlay") {
        for enemy in child.children {
            if (enemy.name == "Enemy" && frameCount %
enemyWeapon1.fireRate == 0) {
                enemyWeapon1.position = child.position
                fireWeapon(weapon: enemyWeapon1, senderPosition:
enemy.position)
            }
        }
    }
}
```

This code first creates a placeholder enemyWeapon variable, so we can access properties like the damage, fireRate, and force that the bullets should have. It then finds children of the foreground named “Overlay”, and within them, searches for children named “Enemy” - so, we want to find children of the overlay nodes in the foreground named “Enemy”. Note that we added our “Overlay” nodes, which contain all of the enemies if you refer to “Wave1.sks”, to the foreground node during the wave spawning code segment.

Once we’ve found these enemy nodes, we set the initial position of the weapon to the position of the overlay. This is because the weapon position in fireWeapon is calculated using two parameters: The initial position of the weapon, and the current position of whatever’s firing. Because our “Overlay” nodes change position as they move down the screen, we need to account for their position, so we find the position by adding the overlay node’s position to the position of the enemy within the overlay - which the fireWeapon function handles for us. We then actually fire the weapon. This should produce firing enemies. We now need to handle the collisions between enemy bullets and the player - right

now, nothing happens. Add the following to the didBegin() function:

```
if (body1.name == "EnemyBullet" && body2.name == "Barrier") {
     body1.removeFromParent()
} else if (body2.name == "EnemyBullet" && body1.name == "Barrier")
{
     body2.removeFromParent()
}
if (body1.name == "Player" && body2.name == "EnemyBullet"
    || body2.name == "Player" && body1.name == "EnemyBullet") {
     restart()
}
```

This code first checks for any collisions between enemy bullets and barriers, and if so removes the bullet. Next, if any enemy bullet collides with the player, it restarts the game.

This is actually all we need to make enemies fire. At this point, the game should work. It's barebones for the moment, but functional and extensible. There are many ways to extend this project from here - some suggestions follow:

- Add a way of tracking score, which we do calculate
- Create a popup when the game restarts to tell you your score
- Add more types of waves
- Add more enemy types, with different weapons
- Add a menu screen
- Add GameCenter integration
- Spice up the paths enemies take (right now it's straight down the screen)
- Create explosions when things die
- Create a little burst animation when bullets hit things

In any case, congratulations on making it this far - many don't. At this point, you should have all the skills you need to wrestle your way through making apps on your own - through liberal use of the internet and through making mistakes, fixing them, and learning from them, you can make pretty much anything. Don't be limited by what you think you can do, because that limits your improvement - and you can figure anything out if you want to badly enough and know how to use Google efficiently.

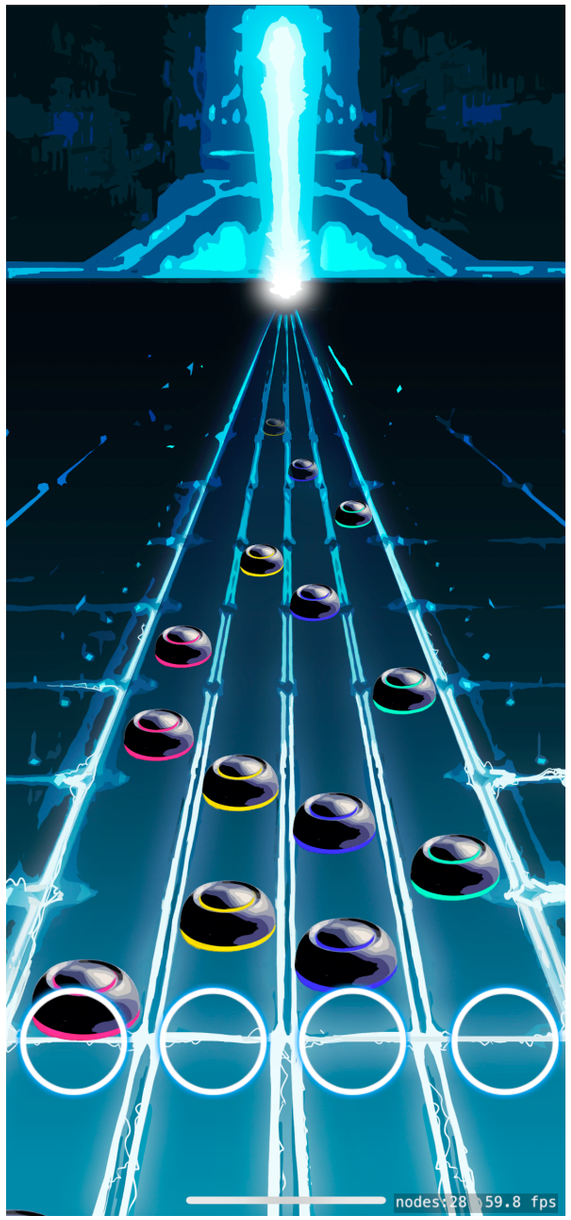
Fight on!

Sylph

Code-Centric SpriteKit Game Design

Difficulty: Advanced

In this chapter, we will be creating a **rhythm game** based on classic games like Guitar Hero, Dance Dance Revolution, and Stepmania (some similar phone games include Tap Tap Revenge, Cytus, Deemo, and Voez). Specifically, we will be building a four-bar rhythm game with tap controls similar to those of Stepmania or DDR. Once we're done, the game will even be able to read **beatmaps** from Stepmania's existing online archive - in other words, we'll be able to play thousands of existing songs rather than having to make our own maps.



This tutorial covers a different set of concepts than Nebula 'Nnhilation, though many of the core ideas overlap. Specifically, we won't be doing as much work in the **scene editor**, instead focusing on controlling our game in code instead of visually. Once you've completed this chapter, you should find yourself well-equipped to handle building your own games that use both the SpriteKit scene editor and frequent customizations of the scene in the corresponding code file.

Swift 4

Section 1

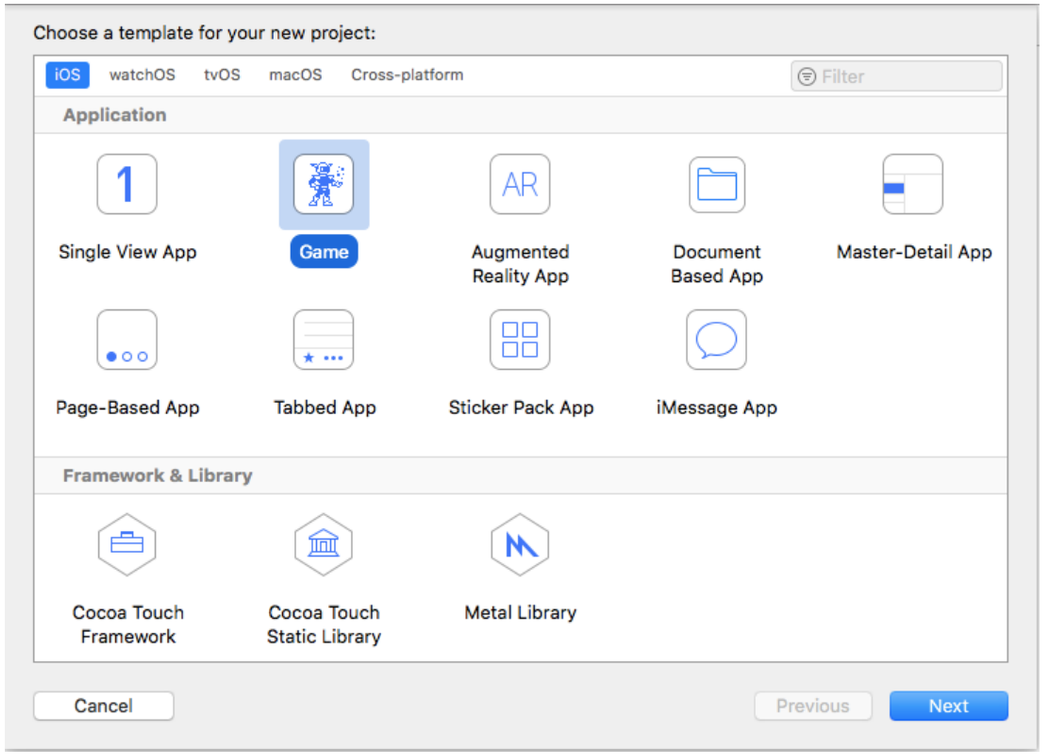
Getting Started

To reiterate from the introduction, any term that is bolded can be tapped on for a full definition. This should help in case you forget what something is or need more information. Examples include terms like **SpriteKit** or **Rhythm Games**. Feel free to Google anything you would like a more detailed or simpler explanation of - or just ask a TA. We're all pretty familiar with how things work and why they work that way, so we should be able to answer any questions you have.

The best way to follow this tutorial is not just by blindly copy-pasting everything, but by trying to fully understand each bit of code that you're using. If you're not clear on either how something works or why we need to do something, feel free to ask a TA. That's what we're here for. Additionally, this project is set up to be a good reference when trying to make your own apps or games in the future - it's a lot easier to have a finished project to look at with questions rather than starting from scratch, and most of the sections in this tutorial will apply similarly to most games you would want to make. The first chapters should be particularly applicable, as the setup process is the same for most games.

With that said, let's go ahead and begin setting up our project in **Xcode**. The process will be very similar to the one for the project in Chapter 2.

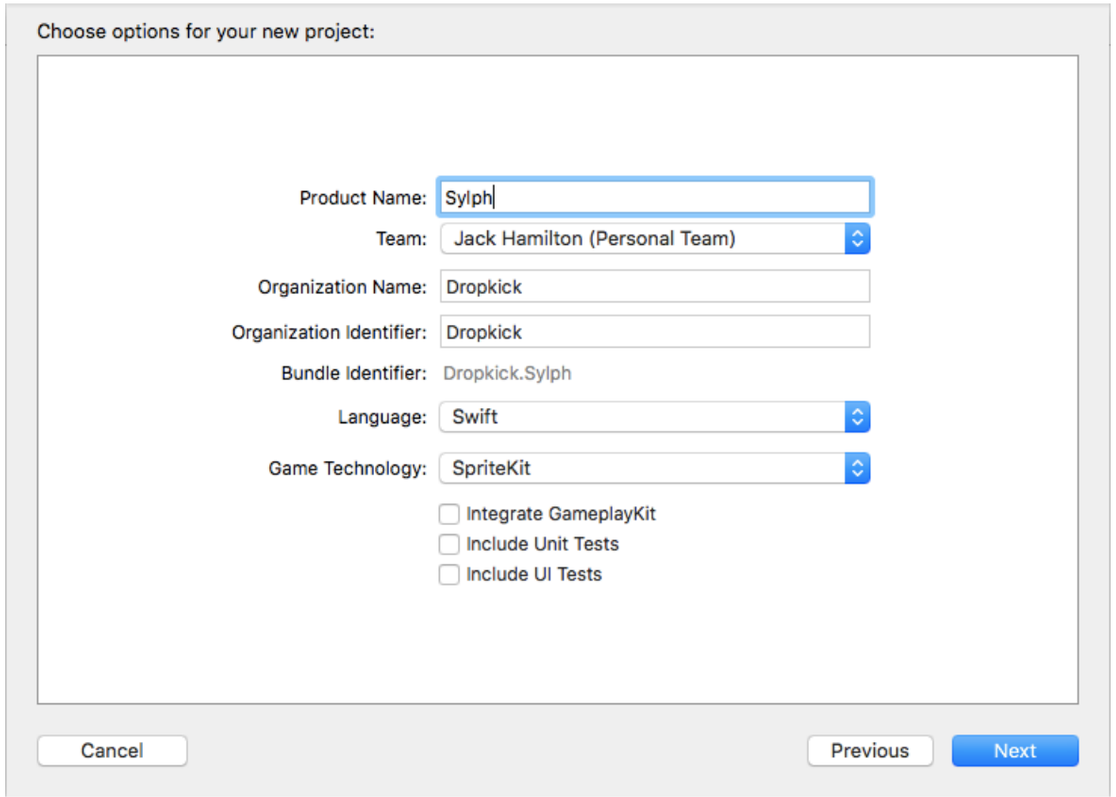
In Xcode, choose "Create a new Xcode project" from the launcher or File -> New -> Project from any opened project.



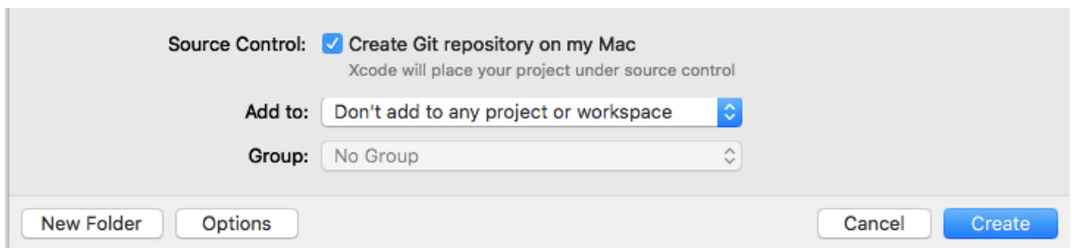
You will be met with the above prompt. First, make sure you have iOS selected in the bar at the top, then choose "Game" and click the next button.

The next prompt will ask you for your project settings. The "Product Name" field is just your project's name - give it whatever title you like. Organization name and identifier don't matter, but make sure the language is **Swift** and the game technology is set to **SpriteKit**. This refers to the 2D game building technology we'll be using to simplify making our game. Most 2D games are made in SpriteKit, or in **SceneKit** if you want to do 3D (this is what we use for AR games). Finally,

uncheck the three check boxes at the bottom - these are for advanced testing. We will not be covering this, as it isn't necessary for non-professional programming.

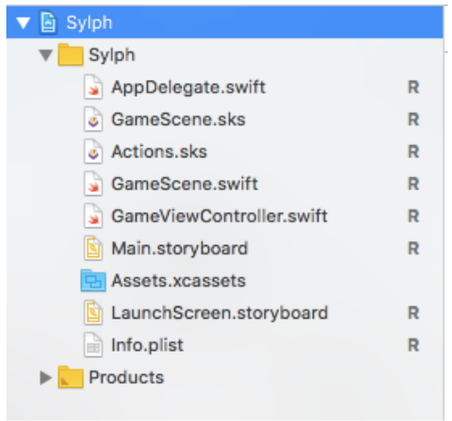


I named my project Sylph, and told it my organization was Dropkick - a name I came up with a few years ago that I publish all my games under. You can use whatever you like for these fields - it won't change anything later in the tutorial.



Go ahead and click "Create".

It's worth noting that the default setting is to create a **Git repository** in your project, which is (among other uses) a version control system - what this does is ensure that you have access to all versions of your project at all times. If you make a change that breaks your project, it is possible to use the built-in source control to revert your project to the save immediately before you made the faulty change - as long as you've been using the system. We don't cover how to do this, but Googling "how to use Git in Xcode" or something similar should bring up a number of good resources.

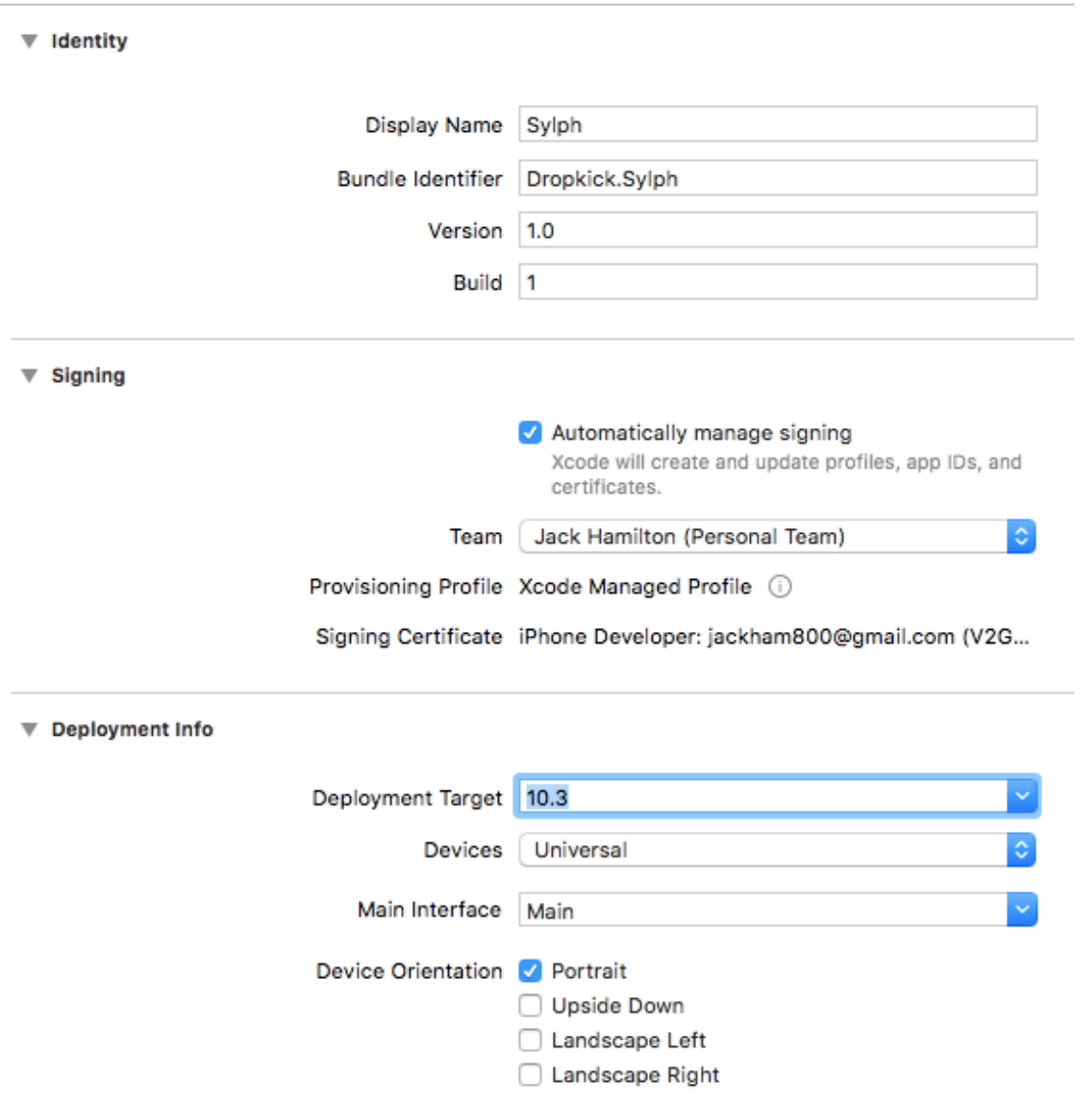


On the left is the file structure that the 'game' preset should have generated for you.

We're going to clean this up in a bit, but first go into the project settings by selecting the title as shown to the left.

This should take you to a section where you can set the Display Name of the app - do this now. This will be the name that appears under the app icon when the app is installed. Also on this page is the supported orientations section - uncheck 'Landscape Right' and 'Landscape Left' now under the Deployment Info section. Our rhythm game will be portrait only. Additionally, if your phone is on an old version of iOS, you can set the Deployment Target - this is the minimum version of iOS your app will be able to run on. If you want the newest Swift features, you can leave it on the default, but my

phone is old, so I set it to 10.3. This just means that some of the newest features in iOS 11 and 12 will not be supported - and we won't be using any of them in this app, so we don't care. You would have to set it to a newer version if you wanted to do stuff like custom Siri commands and neat vibration patterns, but iOS 10 has everything we need to make a game.



When you're done, the page should look something like this.

Now to clean up the file structure. Right click on the game's main folder, and in the dropdown select 'New Group'. Name it 'System'. This is where we will put the files Xcode generated that we don't need to touch.

Move "*AppDelegate.swift*", "*GameViewController.swift*", and "*LaunchScreen.storyboard*" into this group. Then delete the file "*Actions.sks*" - we'll be doing all of our **SKAction** coding manually rather than in the editor. Do note, however, that there is an Action editor in Xcode that you can use if you prefer. This can be accessed by File > New > File and then scrolling down to Resource and creating a SpriteKit action.

Next, open "GameScene.swift" and replace the contents of the file with the following:

```
import SpriteKit
import GameplayKit

class GameScene: SKScene {

    override func didMove(to view: SKView) {

    }


    func touchDown(touch: UITouch) {

    }

    func touchUp(touch: UITouch) {

    }

    override func touchesBegan(_ touches: Set<UITouch>, with event:
UIEvent?) {
        for t in touches { self.touchDown(touch: t) }
    }
```

```
    override func touchesEnded(_ touches: Set<UITouch>, with event:
UIEvent?) {
        for t in touches { self.touchUp(touch: t) }
    }

    override func touchesCancelled(_ touches: Set<UITouch>, with
event: UIEvent?) {
        for t in touches { self.touchUp(touch: t) }
    }


    override func update(_ currentTime: TimeInterval) {
        // Called before each frame is rendered
    }
}
```

This is the regular contents of that file, but without the code that powers the demo. If you ran the project before replacing the code in this file, you might notice that there's a simple demo project. This just resets it to a blank project.

Finally, we need to remove support for landscape views entirely from the app. To do this, go into the "**Info.plist**" file and expand both lists of "Supported interface orientations." Press the minus button next to both landscape modes in each dropdown to delete them.

Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
Localization native development region	String	$(DEVELOPMENT_LANGUAGE)
Bundle display name	String	Sylph
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
▶ Required device capabilities	Array	(1 item)
Status bar is initially hidden	Boolean	YES
▼ Supported interface orientations	Array	(1 item)
Item 0	String	Portrait (bottom home button)
▼ Supported interface orientations (iPad)	Array	(2 items)
Item 0	String	Portrait (bottom home button)
Item 1	String	Portrait (top home button)

That ends the project setup! Let's learn about the files Xcode has given us. We'll be covering what all of the files and **functions** currently in the project do. If you've already done the project in Chapter 2 (App Camp 2017 alumni, for example), feel free to skip the next section.

Section 2

The Default Project

In this section, the autogenerated files Xcode includes in the Game template will be explained, along with all of the **functions** within relevant to our game project. We'll go more into depth with the critical functions as we build our project - this is simply an introduction and orientation section.

Looking in the System folder, the first file is AppDelegate.swift. This project does not deal with the contents of this file at all, but what it does is respond to a change in the app window's state - essentially, the functions inside AppDelegate run whenever the app starts, is put into the background, reactivates into the foreground, or changes state in any way. It can be used to do things like pause a game when the user enters the notification tray and resume it when they swipe back up into the game. We won't be touching it for this project.

The second file is GameViewController.swift. This file is a bit more critical, as it is the file that loads **scenes** onto the main screen. Though we will not have to touch that part for this game, as by default Xcode generates a line loading the app into the ***GameScene.swift*** scene that it generates. However, if you wanted to create a menu screen that the app loads into instead of just the main game, for instance, this file would be where you would do that (specifically, the line "`if let scene = SKScene(fileNamed: "GameScene")`" could be changed to refer to any scene file). Additionally, the following lines:

```
view.showsFPS = true
view.showsNodeCount = true
```

in the viewDidLoad() function are responsible for creating the node count and FPS counter that you can see at the bottom right of the screen when starting the project. Removing those is as simple as deleting those two lines of code, or changing the two **booleans**' values to false.

Info.plist stores your project settings. You can ignore this file because most options are editable in the Xcode interface. Sometimes you'll have to dip into this file, but mostly just for editing supported interface orientations (landscape, portrait, and a few others) in certain specific cases. For now, don't worry about it.

Outside of the System folder are the files we'll be dealing with more directly. ***GameScene.sks*** and ***GameScene.swift*** are two sides of the same file - the visual side that can be manipulated through Xcode's built in scene editor, and the code side that can be linked to elements in the visual side and directly and far more accurately manipulate those elements over time as the game runs. Typically elements are added to the scene visually, by drag and drop, into the *GameScene.sks* file, have their starting properties edited in the editor pane in the file, and then given actions and responses to certain events via code in the *GameScene.swift* file.

Third is the Main.storyboard file. This contains a collection of the UI (User Interface) screens present in the app, and is in this case simply a container for the game screen. We don't have to worry about this file in this project either, but if you wanted to create a menu or scoreboard in **UIKit** instead of SceneKit, this is where that would be done. UIKit is not covered in this chapter, but is covered extensively in the Swift documentation for anyone interested in exploring further. It is mainly used for building UI-based apps - anything with menus. One example of a UI app would be Snapchat, which is primarily UIKit-derived.

LaunchScreen.storyboard, the fourth file, is another UIKit document. It is the screen that displays while the app is loading. By default, it's just a white screen, but it's fairly easy to recolor it or put an image on it - for instance, your game logo. However, if you do, ensure that it scales properly by using the constraint system in the editor, otherwise it might be off center on larger devices or stretch improperly.

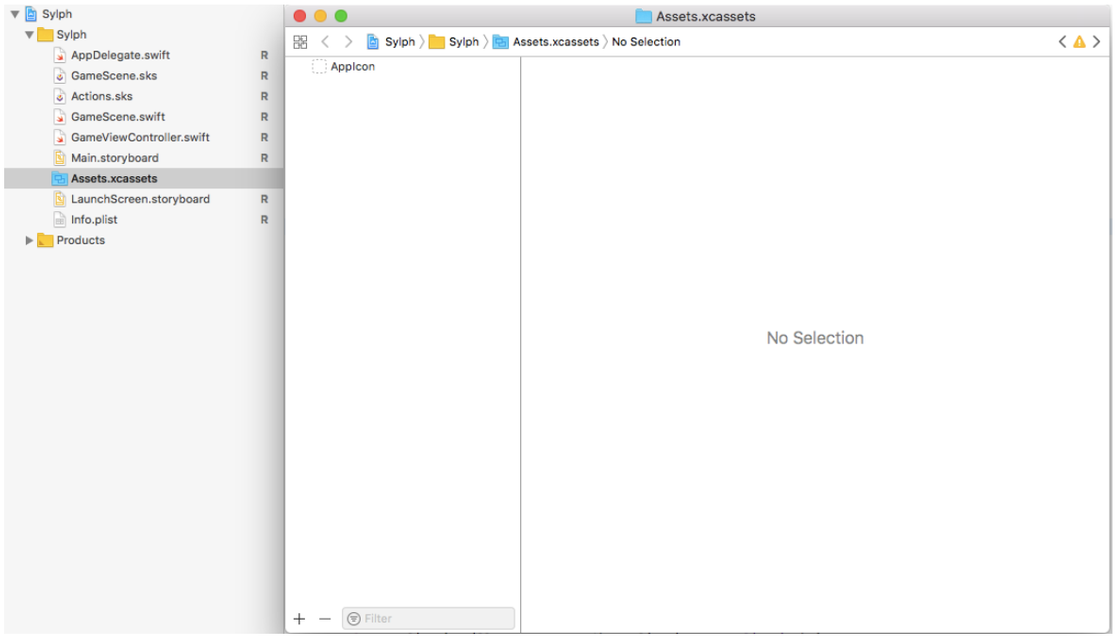
Finally, Assets.xcassets stores your project's assets - mostly used for images. Files can be added to your project by double clicking on the *Assets.xcassets* file in your project's file explorer and then dragging files into the editor that pops up. These files are then automatically imported into your scene editor's assets library and can be referenced in code using the file's name.

We will be creating our own files in addition to the presets in order to compartmentalize and better organize our code, but a simpler game could be created by just inserting code and editing elements through the scene editor into the preexisting files explained here.

Section 3

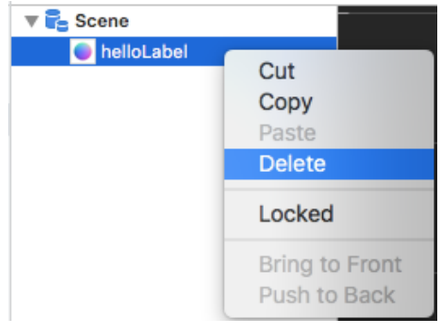
The Game Screen

In this section, we'll set up the game display. We'll hook it up to code in the next section. First, open up the Assets.xcassets file in the project explorer.

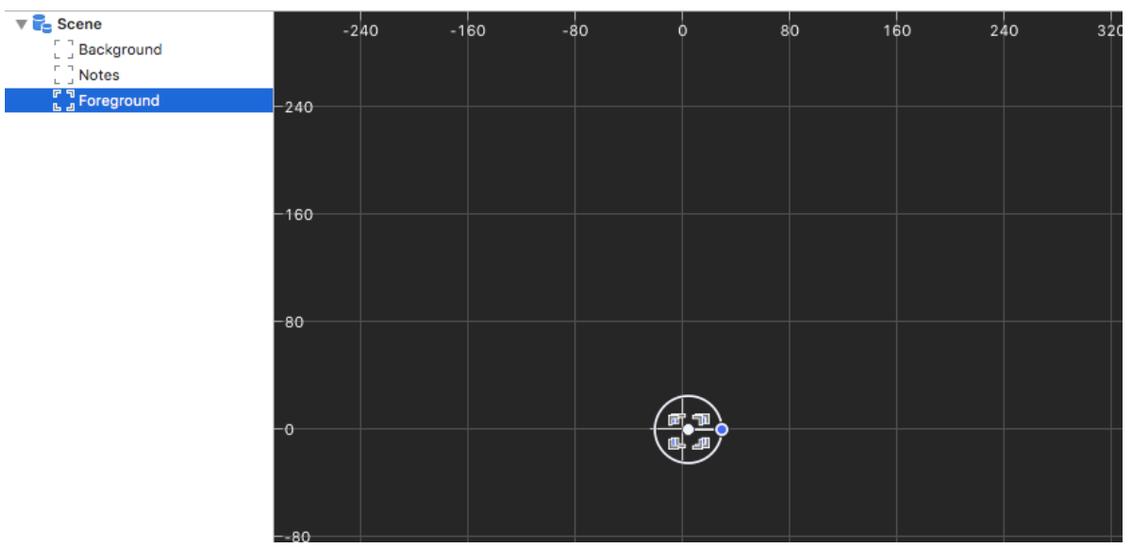


Now, drag in the Background1.png asset from the Backgrounds folder in the provided asset package.

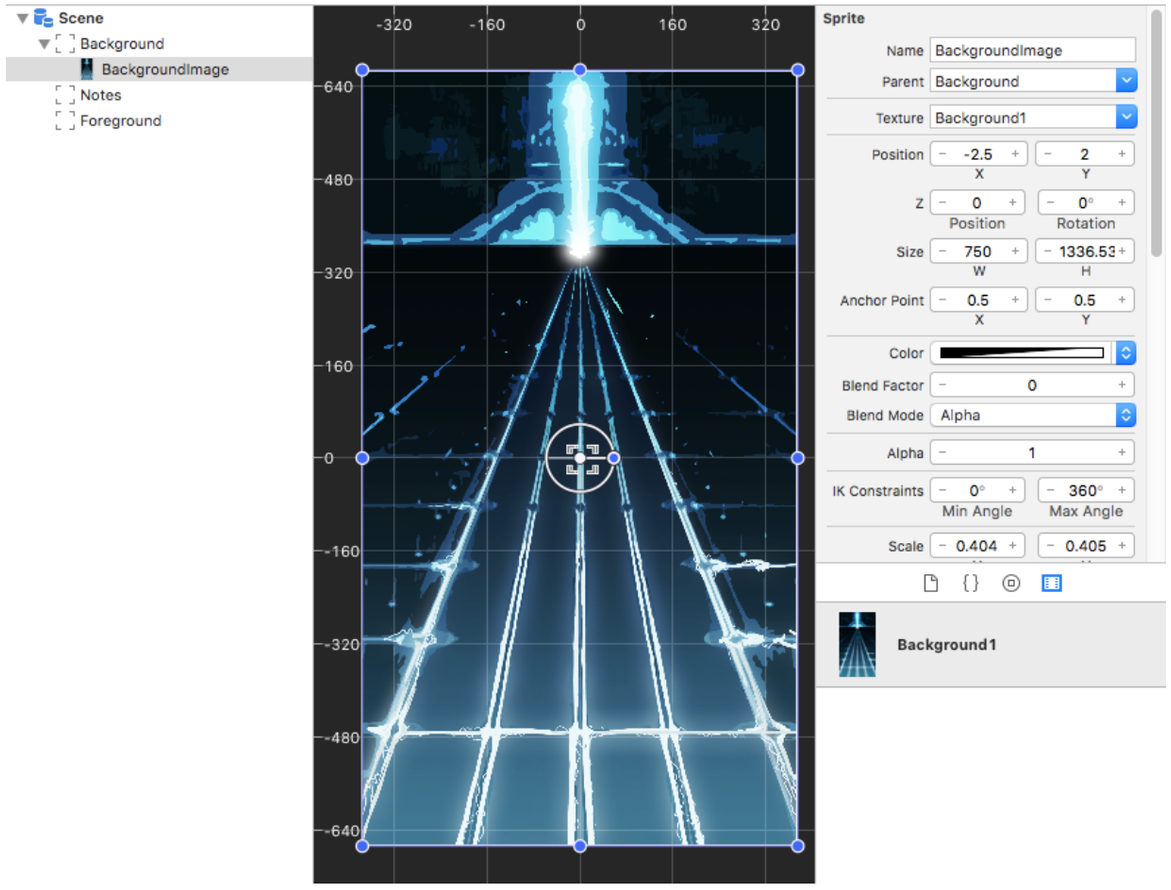
Now, find and open GameScene.sks in the project explorer. This will be the file we set up all our visuals in. It should have a label in it titled "Hello World" - delete this.



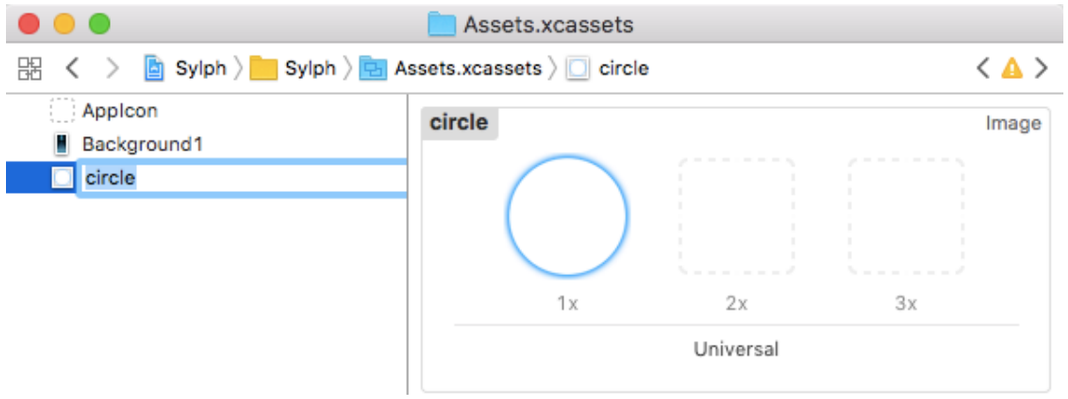
Next, we'll add the background to the scene and scale it up to size. In the **Element Picker** at the bottom right of the screen, switch to the third tab and drag three empty nodes into the game - the position doesn't matter. Name them "Background", "Notes", and "Foreground", from top to bottom (Stuff on the bottom appears over stuff on the top in Xcode). These don't do anything specific, but are good to have for a better organized scene. Set each of their positions to x: 0, y: 0.



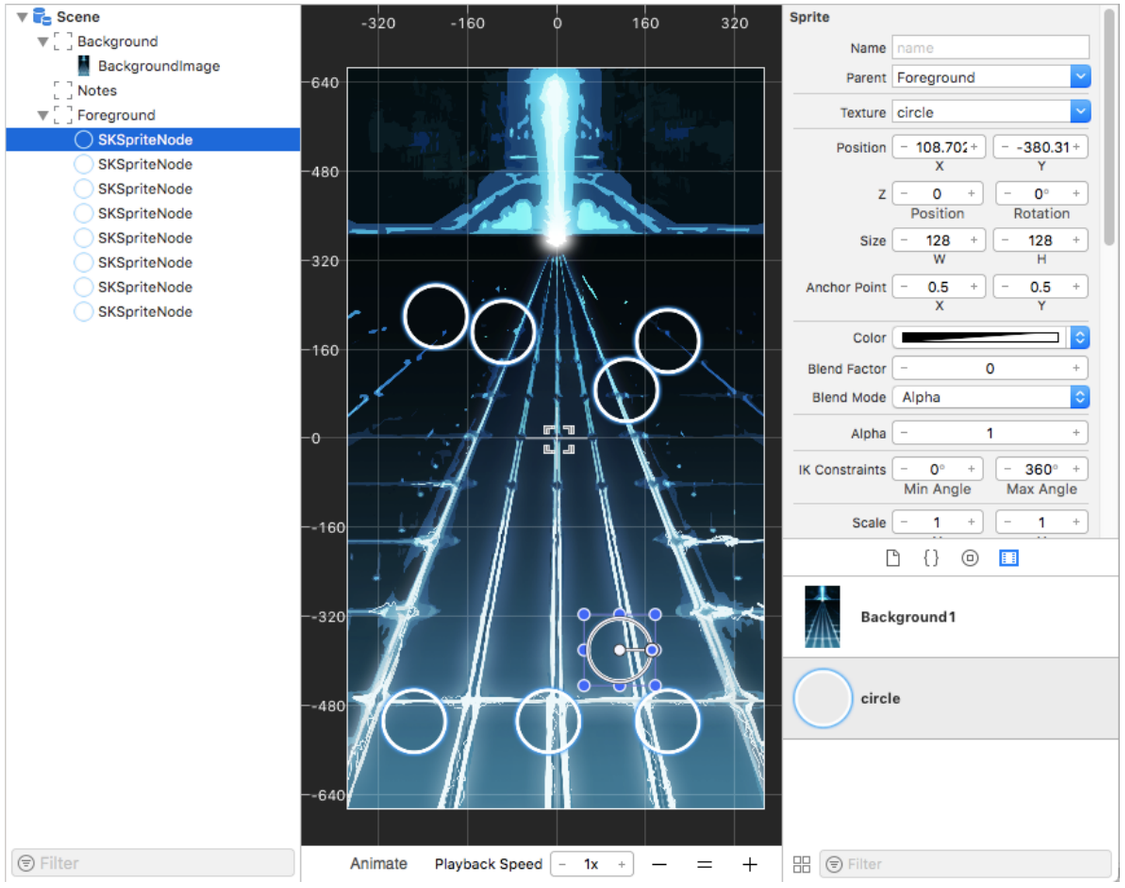
Then, switch to the fourth tab and drag in Background1.png onto the game screen, then in the Element View drag the new SKSpriteNode under the Background node. Name it “BackgroundImage”. Then, use the dots on the image to scale it down to the white rectangle representing the screen size.



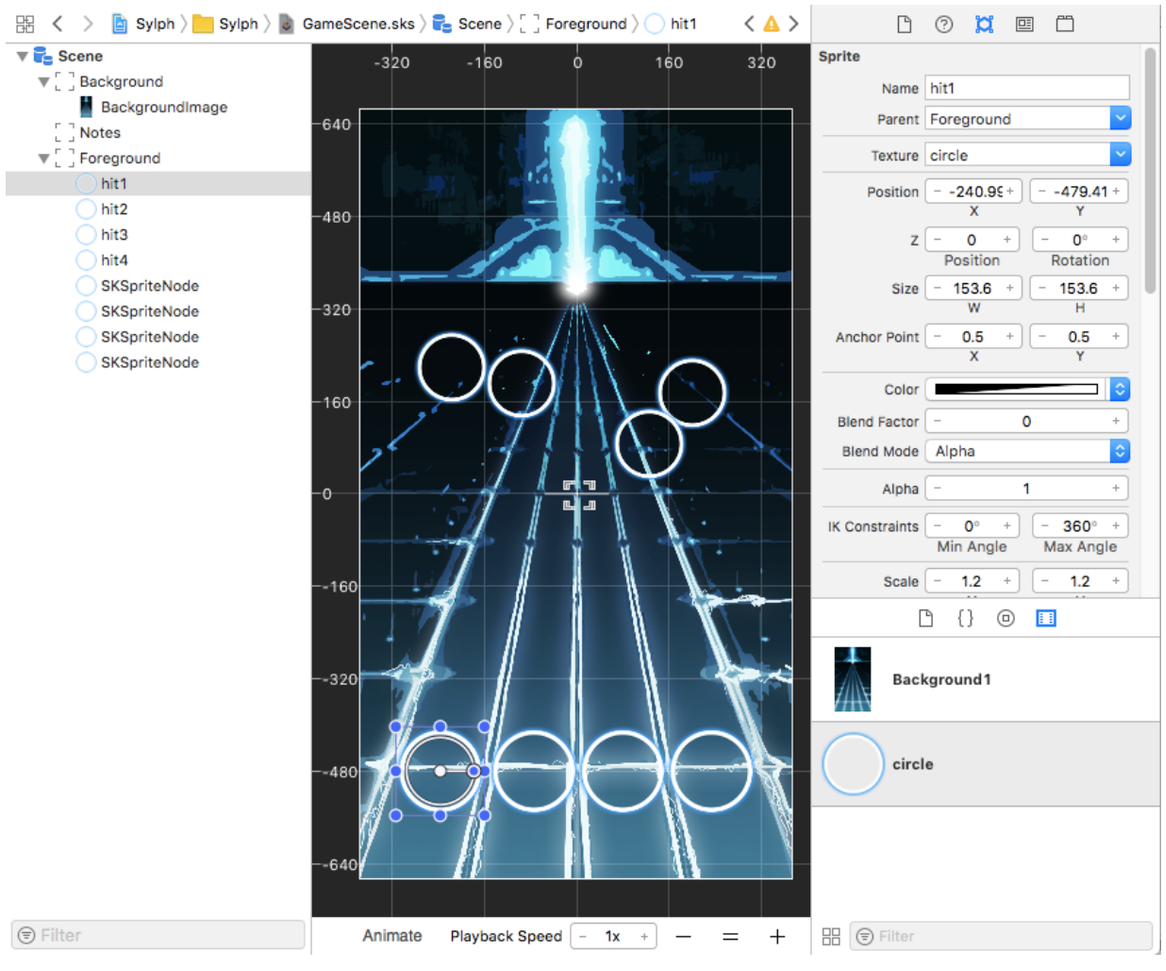
Next, reopen Assets.xcassets and drag in the “circle.png” file under the Buttons folder of the provided assets package. This is the button users will hit to hit notes and will also be used to represent the position on screen the notes come from.



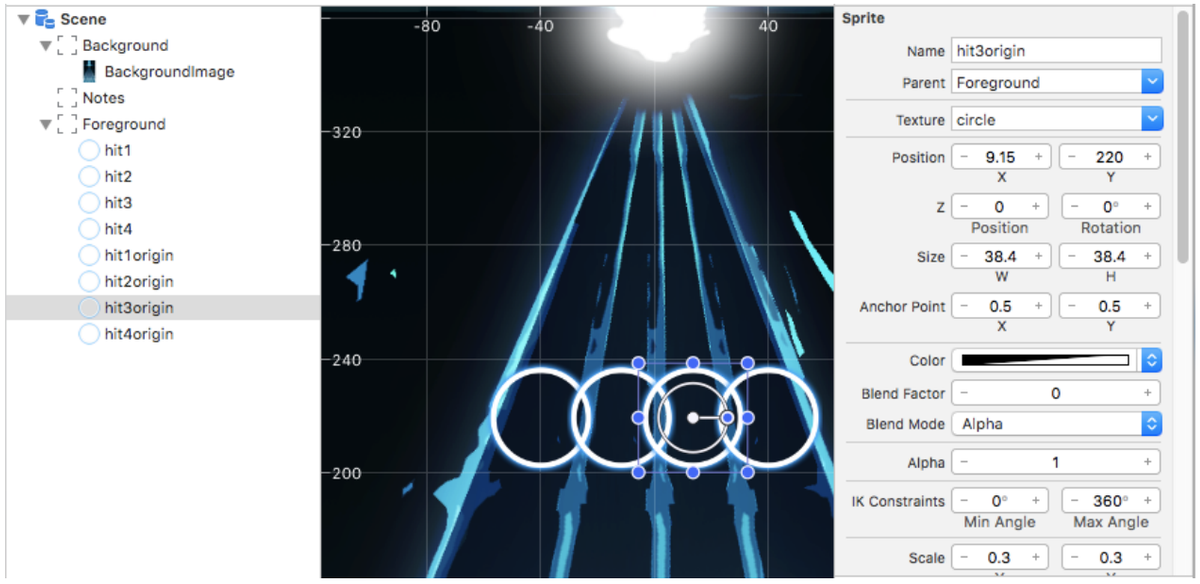
Back in the element picker, drag eight of these onto the screen. Four will be buttons, and four will be note origins. In the element viewer, drag them all under the foreground node.



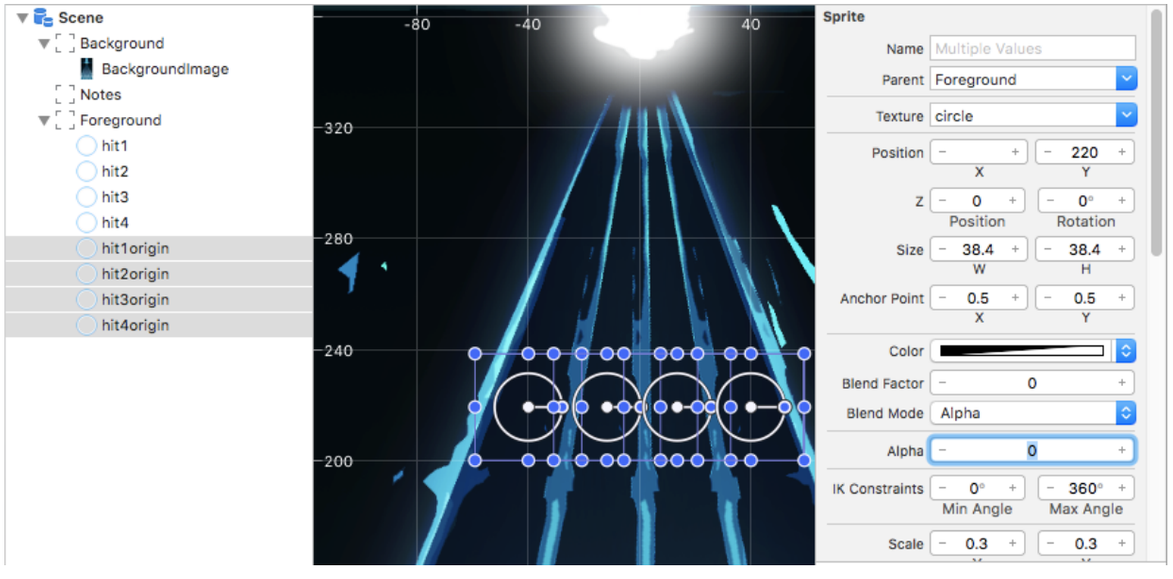
Name the first four hit1, hit2, hit3, and hit4. Then, in the same inspector you use to change the name, scroll down to “scale” and set it to 1.2 for both x and y. This is a value I’ve found fits the bottom part of the screen well. Then, position each of the first four so that the hit lines cross the centers, as in the below image. Ensure that hit1 is in the first column, hit2 in the second, hit3 in the third, and hit4 in the fourth.



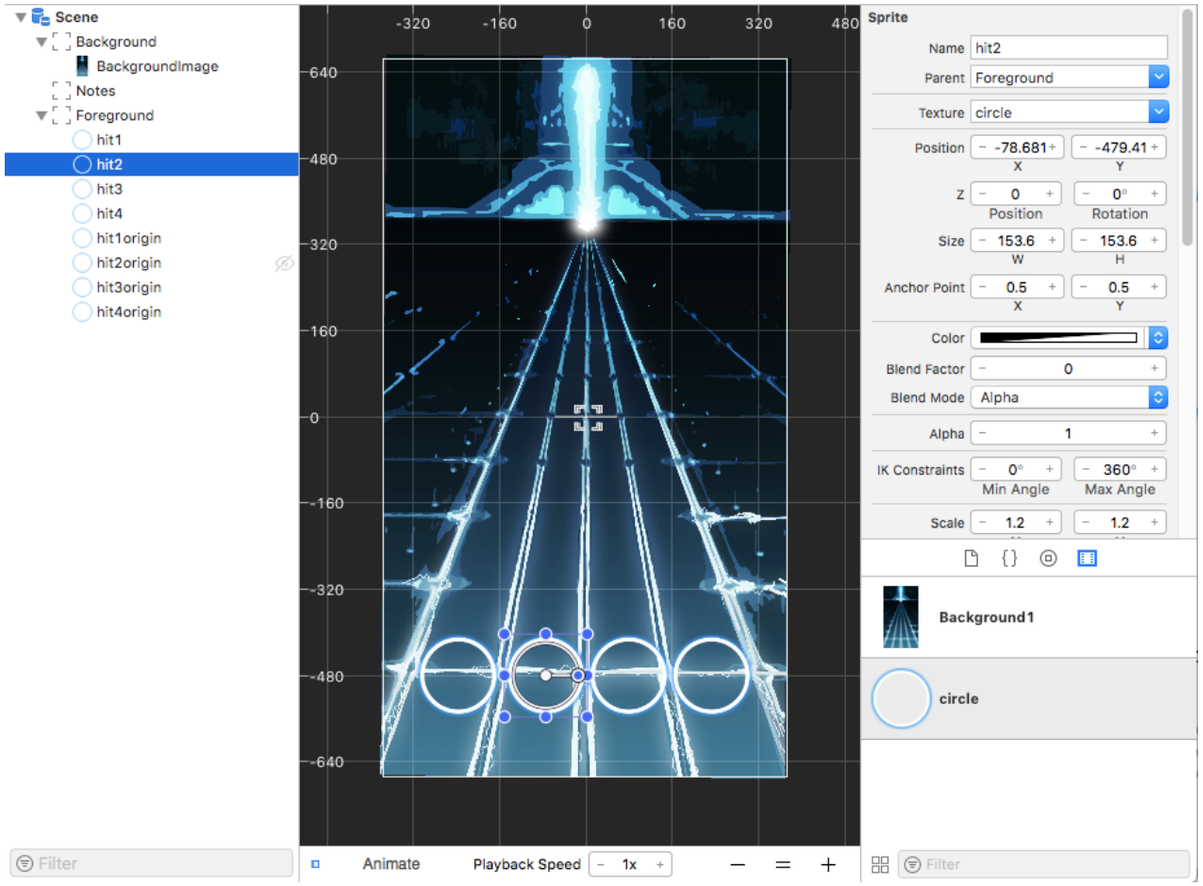
Name the next four “hit1origin”, “hit2origin”, “hit3origin”, and “hit4origin”. These will be where the notes start. For these, set the scale to 0.3 or some similar small value and then move them into position near the higher part of the columns. Don’t worry about overlap, it’s the position that matters most. Make sure they’re all on the same level in terms of y-position. I put mine at y=220. Again, make sure these are all in the correct columns.



Once you’re done, you can select all four “origin” circles and set the “Alpha” (transparency) property in the inspector to 0 to hide them.



When you’re done, the whole scene should look like this:



For now, that’s it in the screen. We’ll come back later to add a scoreboard once we have songs working.

Section 4

Data Structures

Now, we're ready to start displaying songs on our game screen. To do that, we have to have some sort of container to store all the information about the song in. We're going to build four - one for the main song, one for each difficulty, one for the notes in each difficulty, and one representing the image used for each note.

Strap yourselves in, because this is the part where you see the least results. We won't be able to see any visual changes in our game until the next section, but this is very important stuff - and it's worth trying to understand as much of it as you can. We get that it's hard though, so if you have any issues understanding something (most people will), call over a TA.

First, create a new Swift document titled Song (File -> New -> File and pick Swift Document).

Let's set up our struct.

```
import Foundation

struct Song {

}
```

Now, what do we need to put in this struct? Well, each song will have multiple difficulties, so we need an array to hold that. We need the song name and the name of the artist. We need the length of the song, and we'll also make a variable to keep track of how far we are in the song. Finally, we need the location of the music file and the song's picture (album cover/ background).

```
struct Song {
    var difficulties: [Difficulty]
    var name: String
    var artist: String
    var length: TimeInterval
    var currentTime: TimeInterval
    var mp3URL: URL
    var bgURL: URL
}
```

The first type is "Difficulty" - we're about to make this class, so don't worry about that. There are two odd data types at the end there - TimeInterval and URL. TimeInterval is just another word for Double, which you may remember is a decimal number. They're exactly the same thing, actually - TimeInterval is what's known as a typealias for Double, which just means that when XCode sees "TimeInterval" when it's compiling your app, it just replaces it internally with "Double". Finally, URL is something you might recognize if you've dug into your internet browser - it stands for Universal Resource Locator, and is a type that, in this case, basically just points to a file on your hard drive. It can also point to files on the Internet, but we won't be using that feature for now.

Now, let's build the Difficulty class. Create the file in the same way. This one will have functions, so we can't make a struct - it has to be a class:

```
class Difficulty {

}
```

Now let's make our global variables. For difficulty, we want to keep track of all of the notes in the difficulty, the numeric value of the difficulty (we're importing songs from StepMania, which ranks it's songs from about 1-16 with 16 being the most difficult), and the name of the difficulty. We'll also want to keep track of the current note we're on as we're going through the song, so we'll add a private variable to keep track of that.

```
class Difficulty {
    public var notes: [Note]
    public var difficulty: Int
    public var difficultyName: String
    private var noteIterator: Int
}
```

We haven't made the Note type yet either, so don't worry about that. noteIterator is the variable that keeps track of the current note we're on - an "iterator" is what programmers call variables that start at zero and count up by increments (or iterations) of one, which is what we're going to be doing to keep track of the current note.

Xcode should now be telling you you don't initialize any of these variables. This is true. Let's make an initializer function to give them all starting values.

```
private var noteIterator: Int

public init(notes: [Note], difficultyName: String, difficulty: Int)
{
        self.notes = notes
        self.difficultyName = difficultyName
        self.difficulty = difficulty
        noteIterator = 0
}
```

This is an initializer function - it's called when you make an instance of the object (e.g. something like "let myDifficulty = Difficulty()"). This one takes three parameters - an array containing all the notes in the array, the difficulty numerically, and the difficulty name. The "self.x = x" type code sets the global variables equal to the ones passed in - basically, for instance, our variable in the parameters is called "notes", but our global variable that we made earlier is also called "notes". To distinguish between these, self.notes will always refer to the global variable whereas just stating the name will always refer to the most local version - in this case, the one we declared inside the function. This principle is called "scope", and you can Google it or ask a TA if you're curious or don't understand.

The effect, however, is that the values you pass in - the note array, difficulty number, and difficulty name - are stored in the global variables within the class to keep track of. On the last line, we then set the noteIterator to 0 - remember, arrays start at zero - because we'll always be starting playback from the first note.

Now, we want to make two functions to work with the noteIterator. We'll have one that checks what the next note is (so that the game can determine if it's time to play it), and one that advances to the next note. We'll call the one that checks the next note peekNextNote() and the one that advances advanceNextNote().

```
public func peekNextNote() -> Note? {
    if (noteIterator < notes.count) {
        return notes[noteIterator]
    } else {
        return nil
    }
}
```

The return type is Note? - the reason for this is there may or may not be another note remaining. If there is, we'll return it. If there isn't, we just say "this doesn't exist, sorry", and throw back nil. This is what the if() statement does - it checks whether the current note (stored in noteIterator) is less than the number of total notes.

```
public func advanceNextNote() -> Note? {
    let note = peekNextNote()
    noteIterator += 1
    return note
}
```

The advanceNextNote is exactly the same except it adds one to the noteIterator, so we just call peekNextNote(), store the value, add one to the iterator, and return the stored value.

This finishes out our Difficulty data structure. Let's move on to the actual Note class that powers this one. Make a note class in the same way as you made the previous two swift files.

This time, we'll need to import a new library - notes will have to keep track of their positions on screen, so we'll need the CGPoint class, which is a part of SpriteKit:

```
import Foundation
import SpriteKit

class Note {

}
```

Now, let's cover what we'll need to do in this class.

This one will be a class too, not a struct, as having an initializer for it will make things simpler. We'll need a variable to keep track of the time, a variable to keep track of which column the note will appear in, the position the note will start at, the position the note will end up at, and the actual image.

```
class Note {
    public var time: Double!
    public var hitCircleIndex: Int!
    public var startPosition: CGPoint = CGPoint(x: 0, y: 0)
    public var endPosition: CGPoint = CGPoint(x: 0, y: 0)
    public var sprite: NoteSprite?
}
```

Time here will be a Double, which is fine because Double is interchangeable with TimeInterval. hitCircleIndex refers to which column it's in - indexes 0, 1, 2, or 3 referring to columns 1, 2, 3, and 4 on the visual diagram respectively. The reason we're counting from zero here and one in the visual version is

because although confusing, it makes more sense to index the array of hit circles from zero even though the 0th element is really the first and so on. This is one of the quirks caused by array indexing starting from zero, and it can be confusing. If you'd prefer, you can rename the columns hit0-hit3, and hit0origin-hit3origin, as I did when I first prototyped the app. However, you'll find it to be less confusing once the code for working with those arrays appears later on, so I'd say just leave it for now and change it later if it really bothers you that the numbers don't line up between the column names and the array indexing.

Returning to the variable explanations, startPosition and endPosition will be the note's origin and end. We'll move it from one to the other in code. They're set to a default of type CGPoint, which is just a point of the form (0, 0). If you're not familiar with points in this form, feel free to call over a TA or Google "Cartesian plane explanation" or something similar. It's not absolutely necessary, though, as we'll be setting these points both to the position of another object later on, which should be easier to understand.

Finally, there's a variable called 'sprite' of type NoteSprite. This is a type we're going to make - but to just discuss terminology, this is what games call a **Sprite**. I like the word, because it makes it sound like I'm programming with fairies or something, but really that's just a word for an image with some data attached to it (position on the screen, size, and transparency, among a few other things). They're useful because you can attach an image to them and chuck them into the scene by calling addChild(mySprite) and they'll just display the image with no fuss. We'll be making a custom class to hold our own custom sprites, but if you only had one note color (for example) you could just create an **SKSpriteNode**, which is SpriteKit's version of a sprite, and feed it an image to display then add it to the scene without bothering with a custom class.

Now let's properly initialize our variables:

```
public var sprite: NoteSprite?

public init (time: Double, hitCircleIndex: Int) {
    self.time = time
    self.hitCircleIndex = hitCircleIndex
}
```

Here, we take in a variable for the time the note is at and the index of the column to put it in, then store them in the global variables.

That's it for our Note class. Finally, let's make NoteSprite. Go ahead and make the class the same way you made the other three.

We need to import SpriteKit again for this one, because we're going to be making a custom sprite.

```
import Foundation
import SpriteKit

class NoteSprite: SKSpriteNode {

}
```

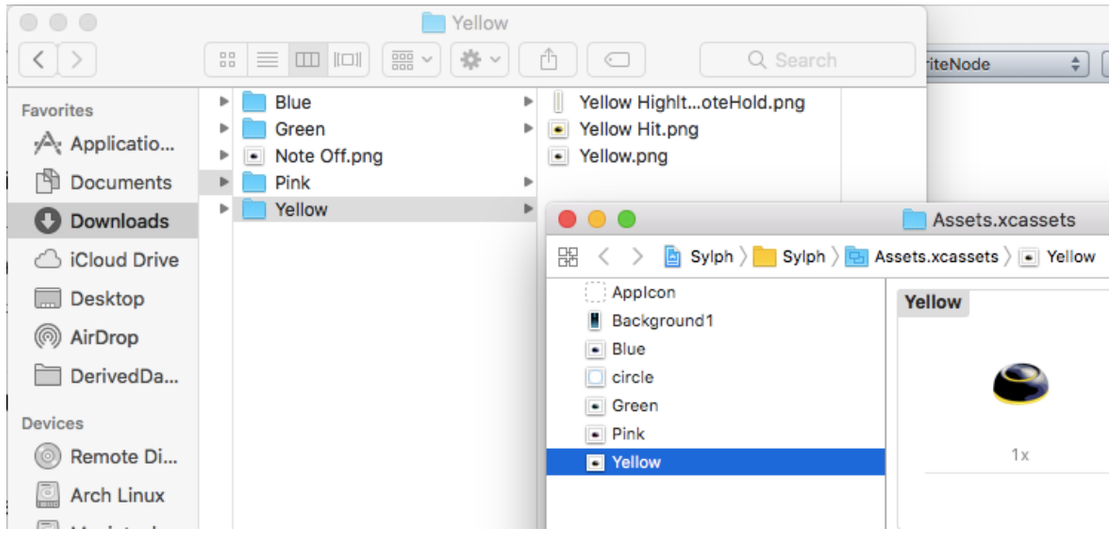
You'll notice this time we have a colon after the classname followed by the SKSpriteNode type - what this means is that we're extending the type and making our own custom class. This class'll have everything SKSpriteNode has, but we can change stuff. In this case, we'll be using it to ensure Notes automatically are the right color when they're made.

Let's make our globals.

```
class NoteSprite: SKSpriteNode {
    public static let pink = SKTexture(imageNamed: "Pink")
    public static let yellow = SKTexture(imageNamed: "Yellow")
    public static let blue = SKTexture(imageNamed: "Blue")
    public static let green = SKTexture(imageNamed: "Green")
    public static let colors: [SKTexture] = [pink, yellow, blue,
green]
}
```

Here, we declare a bunch of static variables of type SKTexture and then place them all into an array in a specific order (the first row should be pink, the second yellow, the third blue, and the fourth green). SKTexture is SpriteKit's version of an image, and you feed it the names of the image files.

Speaking of which, we should add those files to the project. Open Assets.xcassets in Xcode, then from the assets bundle we provided, drag in "Blue.png", "Yellow.png", "Green.png", and "Pink.png" from their respective subfolders in the "Notes" folder. This'll add them to Assets so that when we look for files named "Pink", "Yellow", "Blue", and "Green" in the above code, it actually finds something.



Now, let's make our initializer.

```
public static let colors: [SKTexture] = [pink, yellow, blue, green]

public init (note: Note) {
    super.init(texture: NoteSprite.colors[note.hitCircleIndex],
               color: UIColor.clear,
               size: NoteSprite.colors[note.hitCircleIndex].size())
    self.setScale(1.2)
}
```

We take a Note as a parameter in order to read in which row it's in, so that we can pick our image.

There are a few new things we're doing here. First, because this class is an extension of SKSpriteNode, we need to call the initializer of SKSpriteNode when we make our custom sprite. This is done via super.init(). The SKSpriteNode constructor takes in three parameters: An SKTexture representing the image, a UIColor representing the background color, and the size of the image.

For the image, we feed in the SKTexture from our global variables corresponding to the column of the note that we took as a parameter. In other words, if the note we're using to build the sprite is in the first column, we'd take the first element of our array (at index 0, remember), and give it a pink texture.

For the color, we set the background to a preset value (UIColor.clear) that just makes the background transparent. 99% of the sprites you make will do this.

For the size, we take the same expression we used to get the texture for the image earlier and just call a .size() function on it, which returns the dimensions of the image so we can give that to the sprite constructor.

Then, we set some self properties. We extended SKSpriteNode, so we have all of its global variables. We call SKSpriteNode's setScale() function, which magnifies the size by a factor of whatever number you pass in. In this case, we'll magnify it by 1.2, because I found that that number looked about right.

Finally, XCode requires that classes that extend SKNodes have a specific function in them, and it'll give you an error unless you have it. You can either click on the red error and tell XCode to fix it for you, or copy/paste the below code as a function below the initializer.

```
required init?(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

I don't even know what that does or why it's necessary, but XCode complains if it's not there, so we'll give it what it wants. In my extensive testing, the error never occurred, meaning this function is never called by anything we use. Just write it off as voodoo magic like I did and plug onwards.

That wraps up the NoteSprite class, and means we have all the data necessary to start putting songs on the screen and seeing things happen. Get excited, because the fun bit starts now!

Section 5

Making Stuff Show Up

Now, it's time to hook everything up. Head into our main GameScene.swift file. The first thing we'll want to do is make variables for every element we have in our scene and hook them up. This is a lot of busywork, so I'll provide the code below and then explain it.

```
class GameScene: SKScene {

    public static var currentSong: Song? = nil
    public static var difficultyToPlay: Int = 0
    var foreground: SKNode!
    var noteLayer: SKNode!
    var hit0origin: SKSpriteNode!
    var hit1origin: SKSpriteNode!
    var hit2origin: SKSpriteNode!
    var hit3origin: SKSpriteNode!
    var hit0: SKSpriteNode!
    var hit1: SKSpriteNode!
    var hit2: SKSpriteNode!
    var hit3: SKSpriteNode!
    var hitButtons: [SKSpriteNode]!
    var startPositions: [SKSpriteNode]!
    var hitReacts: [SKSpriteNode?]!
    var timeLabel: SKLabelNode!

    (...)
}
```

We're just making a variable for each thing we dragged visually into the UI so that we can access them from code. We also make three arrays - hitButtons, startPositions, and hitReacts. They'll hold the hit buttons, the hit origins, and popups describing your score when you hit a note respectively. Now let's store in those global variables a link to the elements in the UI.

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

    foreground = childNode(withName: "Foreground")!
    noteLayer = childNode(withName: "Notes")!
    hit0 = foreground.childNode(withName: "hit1") as? SKSpriteNode
    hit1 = foreground.childNode(withName: "hit2") as? SKSpriteNode
    hit2 = foreground.childNode(withName: "hit3") as? SKSpriteNode
    hit3 = foreground.childNode(withName: "hit4") as? SKSpriteNode
    hit0origin = foreground.childNode(withName: "hit1origin") as?
SKSpriteNode
    hit1origin = foreground.childNode(withName: "hit2origin") as?
SKSpriteNode
    hit2origin = foreground.childNode(withName: "hit3origin") as?
SKSpriteNode
    hit3origin = foreground.childNode(withName: "hit4origin") as?
SKSpriteNode
    hitButtons = [hit0, hit1, hit2, hit3]
    startPositions = [hit0origin, hit1origin, hit2origin,
hit3origin]
    hitReacts = [nil, nil, nil, nil]
}
```

First, we create a variable to hold the current song. We set it to nil for now, but we'll make it hold something later. We then make a variable that holds the difficulty value we'll play of this song. It'll default to zero.

The rest links up each element with the visuals - the didMove function is called when the scene starts up, so this is basically the initializer for our game. Let's dissect one off the lines (they all do basically the same thing):

```
hit0 = foreground.childNode(withName: "hit1") as! SKSpriteNode
```

Essentially, we're saying that there exists a child node of the foreground node with the name "hit1", and we want to assign this to the hit0 variable. If you look in the visual editor, you'll find that there indeed is a child of the Foreground node named "hit1", which is what we're referencing. Then, the as? SKSpriteNode part is called a *cast*, which tells the computer that the element we're talking about is actually a sprite - and if you look in the visual editor, you'll find it indeed is an SKSpriteNode. The reason we assign hit1 in the visual editor to hit0 in the code is because we're going to put hit0 into an array, at which point it will be the 0th element. So it's still the first, but we're calling it by its index here.

In the last three lines, we add the hits and hitorigins to arrays, so that we can say hits[0] instead of hit0, for instance. This is useful when we want to loop through all of them and do the same thing to all of them later on. Finally, we set all of the elements of the hitReacts array to nil - we'll add these in code later on, seeing as we didn't actually create any reactions in the visual editor. Additionally, to clarify what I'm talking about when I say hit react, I mean the popups that tell you how close you were when you hit a note - e.g. "Perfect!" or "Good!". These will appear later in the code, so we're making an array to keep track of them here.

Now, we can access all of the elements we have in the visual part of our scene in code. Let's make some notes appear on the screen. To do this, we'll need one or two more global variables. Add the following to the bottom of your list of global variables:

```
var timeLabel: SKLabelNode!
var timeOfLastUpdate: TimeInterval! = 0
var notes: [Note] = []
var onscreenNotes: [Note] = []
let approachRate: Double = 3
```

The timeOfLastUpdate variable is a TimeInterval - which is just a Double, or a number with a decimal portion. "notes" is an array that will keep track of all the notes in our song, and "onscreenNotes" will keep track of the ones we have on our screen. Finally, approachRate will be the number of seconds it takes a note to get from the top of the screen to the bottom.

We can now begin working on our update() function. This function basically runs every frame. There are sixty frames per second on average, so this will run about sixty times per second. In it, we need to update the positions of everything on the screen and check for things that need to be added or deleted to the scene.

```
override func update(_ currentTime: TimeInterval) {
    if (currentTime - timeOfLastUpdate < 0.5) {
        var timeSinceLastUpdate: TimeInterval!
        timeSinceLastUpdate = currentTime - timeOfLastUpdate
    }
    timeOfLastUpdate = currentTime
}
```

To begin with, we want to keep track of how long it's been since the last frame, and if it's been less than half a second, advance a frame. This means that if you exit the app and then

reenter it, it will recognize that it has *not*, in fact, been like twenty minutes in game time, and instead pick up where it left off. Generally it takes around 0.06 seconds per frame, so the difference between the current time and the time of the last frame, which is what we test in our if statement, should only be larger than 0.5 if we've exited the app and reentered. Inside of the if statement, we declare a new variable to keep track of how long it's been since the last frame - which we'll use to update the current time in the song in the next step. Finally, we just set the time of the last update to the current time so that when the update() function next runs, it'll use the correct value for the time that the last update finished.

Next, let's update that time value in the current song.

```
timeSinceLastUpdate = currentTime - timeOfLastUpdate

if (GameScene.currentSong != nil &&
GameScene.currentSong!.currentTime < GameScene.currentSong!.length)
{
     GameScene.currentSong!.currentTime += timeSinceLastUpdate;
}
```

Here, we check whether there the current song exists (e.g. is not equal to 'nil'), and whether or not the current time of the song is less than the length (in other words, whether the song is not over). If the song both exists and is not over, we add the time since the last update to the current time of the song. This will keep the song's current time in line with the game's current time.

Next, let's make notes appear in the scene. The next section checks whether the time of the next note is within the range defined by the approachRate variable (three seconds) of the current time of the song, and whether it exists, and if it does, adds it to the scene:

```
GameScene.currentSong!.currentTime += timeSinceLastUpdate;

let cSong = GameScene.currentSong!
let cDifficulty = cSong.difficulties[GameScene.difficultyToPlay]
while (cDifficulty.peekNextNote() != nil &&
       cDifficulty.peekNextNote()!.time - approachRate <
GameScene.currentSong!.currentTime) {

}
```

The first two lines turn the current song and the current difficulty we're playing of that song into local variables that are easier to work with (you could use long strings of references, but it looks clunky). Then, we want to run a loop while the following conditions are true:

- The next note exists.
- The time of the next note is within "approachRate" seconds of the current time.

What we're trying to accomplish here is adding every note within approachRate seconds (3 seconds) of the current time to the screen.

Within that loop, let's make some notes.

```
while (cDifficulty.peekNextNote() != nil &&
       cDifficulty.peekNextNote()!.time - approachRate <
GameScene.currentSong!.currentTime) {
    let tmpNoteBase: Note = cDifficulty.advanceNextNote()!
    let tmpNoteSprite: NoteSprite = NoteSprite(note: tmpNoteBase)
    let hcIndex = tmpNoteBase.hitCircleIndex
    tmpNoteBase.sprite = tmpNoteSprite
    tmpNoteBase.startPosition = startPositions[hcIndex!].position
    tmpNoteBase.endPosition = hitButtons[hcIndex!].position
    tmpNoteSprite.alpha = 0
    tmpNoteSprite.run(SKAction.fadeIn(withDuration: 0.5))
    noteLayer.addChild(tmpNoteSprite)
    onscreenNotes.append(tmpNoteBase)
}
```

Quite a snippet. Let's go through it line by line.

First, we're moving on to the next note, so we call advanceNextNote() on our current difficulty and store the result in a variable called tmpNoteBase - this stands for "temporary note base". We only need this variable to work with for a little bit before we add it to the scene, hence temporary. We'll also need to give it an image, so we make a NoteSprite and pass in the note we just made.

We create a local variable to represent which column the note is in, so that we can feed in the right start position and end positions. We then add the sprite we built to the base note.

We then set the correct start position - because of the array we created earlier, hcIndex represents both the column the note will appear in and the position in the arrays that that column's origin and end notes are stored at. Thus, we give the startPosition value in our note the value of the startPosition in column "hcIndex" from our global variables. We then do the same with the end position.

Next, we set the note sprite's Alpha to zero - this basically makes it invisible. Alpha goes from 0 to 1, with 0 being invisible and 1 being visible, and anything in the middle being varying degrees of transparent. After this, we can run an SKAction on the sprite to fade it in - SKActions basically perform some operation on an object in the game over a period of time. SKAction.fadeIn will gradually fade something into the scene over some period of time - here, we fade in the sprite over a period of 0.5 seconds. You can experiment with this value.

Finally, we add the sprite to the note layer, effectively placing it into the scene. We also attach the new note to our list of all the notes currently onscreen.

This is all we need to make notes appear on the screen. We haven't made a song yet, so nothing will happen, but we can insert some test code in to make it work. To do that, place the following at the bottom of your didMove function:

```
let diff = Difficulty(notes: [Note(time: 3, hitCircleIndex: 3),
           Note(time: 4, hitCircleIndex: 2),
           Note(time: 5, hitCircleIndex: 0),
           Note(time: 5, hitCircleIndex: 3),
           Note(time: 5.5, hitCircleIndex: 1),
           Note(time: 6, hitCircleIndex: 2)
           ], difficultyName: "Test", difficulty: 0)
```

```
GameScene.currentSong = Song(difficulties: [diff], name: "Test",
artist: "Test", length: 7.0, currentTime: 0, mp3URL:
URL(fileURLWithPath: ""), bgURL: URL(fileURLWithPath: ""))
```

This just makes a test song and sets the current song to it. If you run it, you should now see some notes appearing. They don't move down the screen yet, but we'll get to that in the next section. We can also now make music play when a song is loaded by following that with this code:

```
let music = SKAudioNode(url: GameScene.currentSong!.mp3URL)
music.autoplayLooped = false

addChild(music)
music.run(SKAction.play())
```

This creates an SKAudioNode, which essentially is similar to a SpriteNode, but instead of displaying an image it plays a sound file. We provide it with the URL of the current song, then add it to the scene. It requires an SKAction (SKAction.play()) to be run on it in order to start the song, so we then do that. The song that plays should dynamically change as we modify the currentSong variable in later sections.

Section 6

Making the Notes Move

In order to make the notes move, we'll introduce a code snippet in our update() method that moves the notes down the screen by a percentage of the screen height every frame.

Start a new for() loop inside of your update() function just below the closing bracket of the while() loop we just built in the last section.

```
while (cDifficulty.peekNextNote() != nil &&
                    cDifficulty.peekNextNote()!.time -
approachRate < GameScene.currentSong!.currentTime) {
}

for note in onscreenNotes {
    let startScale: CGFloat = 0.08
    let endSize: CGFloat = 0.5
}
```

This for loop will go through every note that is onscreen. It will store each in the 'note' variable and perform the following code block on it. We then make variables to store the starting and ending scale for the notes - they'll start out at 0.08 times their full size (tiny), and scale up to 0.5 times their full size. These values are just what I found to work after testing a bunch, they don't mean anything special.

```
let endSize: CGFloat = 0.5

let secondsUntilHit = note.time - cSong.currentTime
let percentageDownScreen = CGFloat((approachRate -
secondsUntilHit)/approachRate)
```

Next, we calculate the seconds until the note hits the bar . This will just be difference between the note's time (the time the note should hit the bar) and the song's current time.

```
let startXPos = note.startPosition.x
let startYPos = note.startPosition.y
let endXPos = note.endPosition.x
let endYPos = note.endPosition.y
```

Now, we set up local variables for the various positions. This is just to make the code cleaner.

```
let midXPos = startXPos * (1 - percentageDownScreen) + endXPos *
percentageDownScreen
let midYPos = startYPos * (1 - percentageDownScreen) + endYPos *
percentageDownScreen
let midSize = startScale * (1 - percentageDownScreen) + endSize *
percentageDownScreen
```

Here, we *interpolate* between the start positions and the end positions, and also the size. Interpolation basically gets a value some percent from one value to another. Say we want to find a value halfway from 1 to 3 - this being 2 - we would add half of one to half of three, 0.5 + 1.5 = 2. Most languages have a function for this, but we just have to cook one up - this is something you can just memorize how to do. Basically, we just want to get the percentage down the screen that the note should be - say, if it's going to hit the bottom in 1.5 seconds, it should be halfway down the screen, in which case we need to find a value halfway between the start and end positions, then another value between the start and end sizes. That's all this does.

```
note.sprite!.position = CGPoint(x: midXPos, y: midYPos)
note.sprite!.setScale(midSize)
```

Finally, we set the position and the scale of each note to the values we just calculated. This is all that should be in that particular for loop.

Next, we have to delete the notes when they move offscreen, since they're moving now.

Below the two loops we've now set up, add another.

```
while (onscreenNotes.first != nil &&
                    onscreenNotes.first!.sprite!.position.y <
onscreenNotes.first!.endPosition.y -
onscreenNotes.first!.sprite!.frame.size.height) {

}
```

This is an unfortunately busy while loop, but it's pretty simple when you break it down. First, we check whether the first element in onscreenNotes exists - in other words, whether or not any notes are onscreen. If they are, if the position of the sprite is lower than it's end position with the size of the sprite tacked on, then run whatever's in the code (because this means the note is now offscreen). Another way to do this would be just to check if it's less than the screen height, but this gives a little bit more fine control.

Inside that while loop, we now want to fade out then delete the note.

```
onscreenNotes.first?.sprite!.run(SKAction.sequence(
                                [SKAction.fadeOut(withDuration: 0.3),
                                 SKAction.removeFromParent()]
                            ))
onscreenNotes.removeFirst()
```

This will run a sequence of SKActions on this first sprite that is now off the screen. This means that SKActions will be chained - the first one in the array is run, then the second. In this case, we run the fadeOut action, which disappears the note over a period of time - here, 0.3 seconds. We then run the removeFromParent action, which just deletes the node from the screen. Finally, we remove it from our onscreenNotes array - as it's no longer on the screen.

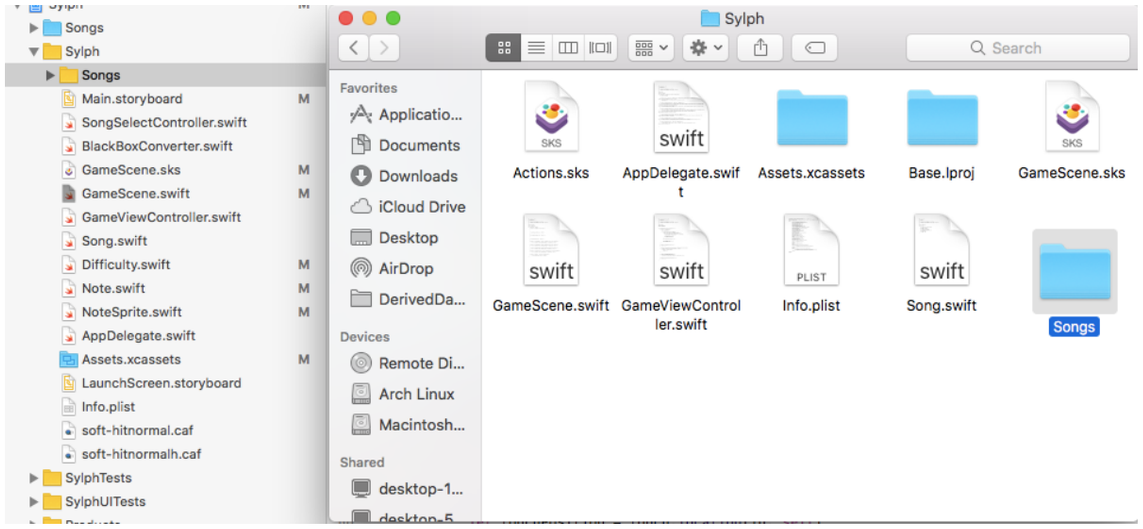
Let's actually get a song now, so that we can play something real. First, remove the code we added earlier in the didMove block that sets currentSong.

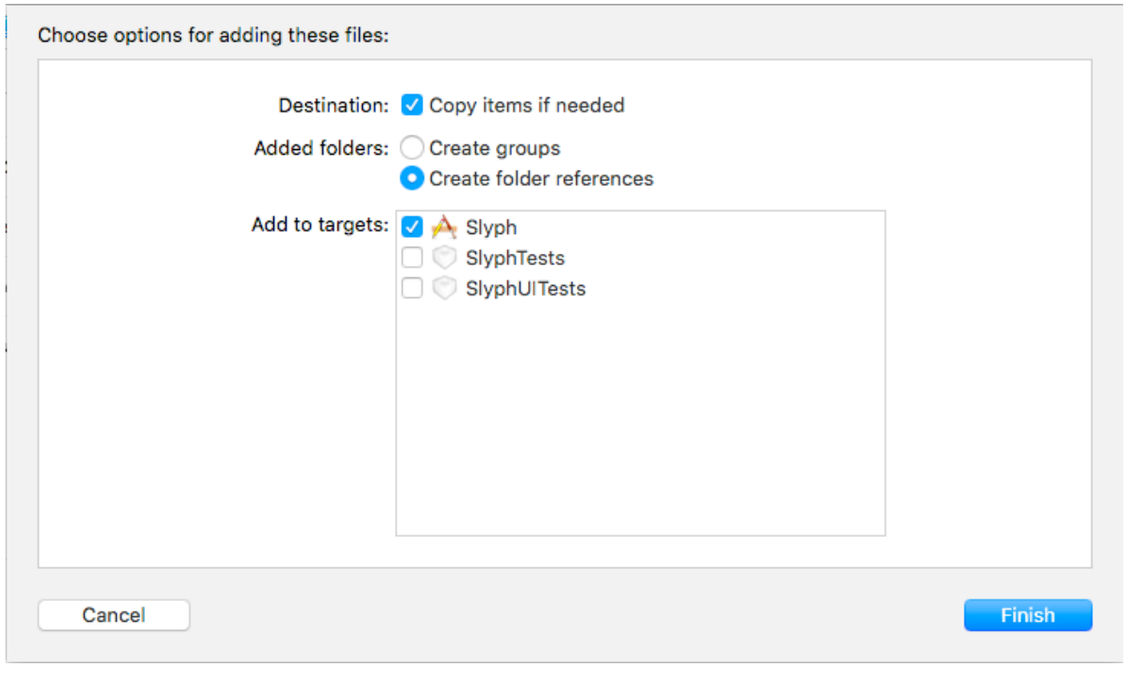
```
let diff = Difficulty(hits: [Note(time: 3, hitCircleIndex: 3),
            Note(time: 4, hitCircleIndex: 2),
            Note(time: 5, hitCircleIndex: 0),
            Note(time: 5, hitCircleIndex: 3),
            Note(time: 5.5, hitCircleIndex: 1),
            Note(time: 6, hitCircleIndex: 2)
            ], difficultyName: "Test", difficulty: 0)
GameScene.currentSong = Song(difficulties: [diff], name: "Test",
artist: "Test", length: 7.0, currentTime: 0, mp3URL:
URL(fileURLWithPath: ""), bgURL: URL(fileURLWithPath: ""))
```

Now, I've provided a file in the assets called BlackBoxConverter.swift. This takes songs and loads them into the program as Song objects - assuming you made the object in the same way as I did in the tutorial. It's actually not that hard in terms of programming, but it is *long* and painful, so

it’s not really worth walking through in the tutorial here - but you can take a look if you’re curious. It’s all basically just dissecting Strings.

Drag the file into your main project structure. Then, create a folder called “Songs” in your project directory, then drag that in as well. When it asks you how to perform the copy, tell it to create folder references, and copy items if needed.





We provide some sample songs in the assets folder, but they can also be downloaded from the internet - any Stepfile should work. A Stepfile is a file for a rhythm game called StepMania, and you can find a bunch of older ones here (There’s a search bar). Newer songs probably won’t be there. If you get a song but can’t get it working, that’s pretty normal - the importer isn’t perfect. If it’s mission-critical music, call over a TA. Otherwise, the sample songs we provide all work (despite being old).

The import process is simple - to add a song, download extract the zip file, then drag the folder with the song’s name into the Songs folder. The converter will then detect it.

But first we have to set it up. This is pretty simple - just put the following code at the bottom of your didMove() function, where the old code setting the currentSong used to be before we removed it.

```
GameScene.currentSong = BlackBoxConverter.getSongs()[0]
```

The converter will then magically fetch the songs for you. Now the first difficulty of the first song should play when you start the app.

Finally, to make the song’s music play, add this code at the bottom of didMove():

```
let music = SKAudioNode(url: GameScene.currentSong!.mp3URL)
music.autoplayLooped = false
addChild(music)
```

```
music.run(SKAction.play())
```

This code creates a new type of node called an SKAudioNode - it's only function is to play music or sounds. We pass in the URL of the current song, which refers to the file and in this project is filled in for you by the black box converter when the song is created.

Section 7

Tapping

Next, we have to get tapping working. When you touch the screen, we want any notes in the row you touched to disappear *if* the note was close enough to the hit circle. So, to do this, we'll need to detect a few things. First, we need to detect the row that you touched. Then, we need to find out if any notes are coming up in that row. Then, we need to delete that note and (later) display a hit react and give you a score.

To do this, we'll be using the touchDown and touchUp functions. These trigger when a touch is detected and when one fades. First, we'll handle the touchDown function, which triggers when the phone detects a touch. First, we'll want to check whether or not the touch was in the same section of the screen as each button:

```
for i in 0...3 {
    let buttonFrame: CGRect = hitButtons[i].frame
    let hitBounds = CGRect(x: buttonFrame.minX, y: -2000, width:
buttonFrame.maxX - buttonFrame.minX, height: 4000)
    if (hitBounds.contains(touch.location(in: self))) {

    }
}
```

Here, we first iterate through 0, 1, 2, and 3, one index for each column. For each column, we create a rectangle around the respective hit button, then make another rectangle with the same starting X and ending X but Y-values that go from -2000 to 2000 (covering the entire screen). This is so you can tap above and below the button and still have the hit register - otherwise we could just use the buttonFrame variable. Then, we check whether the touch's position was within this rectangle - if it was, then we'll run the code in the if() statement.

Inside this statement, we're now sure that we hit a button. We need to keep track of the touch, so that when we animate the button being pressed, we don't end the animation until there are no fingers left on the button. For this, we'll make a new global variable. Add the following variable to your globals (at the top of your file, under the class declaration).

```
var activatedHitTouches: [[UITouch]]! = [[], [], [], []]
```

This is what's called a multidimensional array - basically, it's just an array of arrays. We need one array of touches for each button, so that we can store any number of fingers that might be touching each button at a time. We could also create four different arrays, but this way is easier. This is why we set it equal to an array of four empty arrays.

Then, back in the if() statement at the bottom, we need to add the new touch position to the correct array.

```
if (hitBounds.contains(touch.location(in: self))) {
    activatedHitTouches[i].append(touch)
    hitButtons[i].run(SKAction.scale(to: 1.0, duration: 0.03))
}
```

Then, we run an SKAction on the appropriate button to scale it to a smaller size (remember, our buttons are starting at a higher scale). This will mean that columns that you press in will have the buttons shrink while they're still pressed.

Below this, we're going to check if any notes got hit - e.g. if a note was present within the next second or so in that column that was hit:

```
hitButtons[i].run(SKAction.scale(to: 1.2, duration: 0.03))

if (onscreenNotes.count > 0) {
    var cont: Bool = true
    for j in 0..<onscreenNotes.count {
        if (cont) {
            let note = onscreenNotes[j]

        }
    }
}
```

Lots of brackets, eh? First, we make sure that there are, in fact, notes onscreen. We then make a boolean variable named "cont" - we only want to remove one note regardless of what happens, so we need a variable that we can set to true if we find something that just kills the whole loop and stops looking for new notes.

The second loop here runs once for every note in the onscreenNotes array. The index is stored in 'j'. We need to check every note to see if they're in the same column and whether, if that's true, they're within a certain number of seconds of hitting the hit button.

Now, we want to check those things now that we know a note exists. First, we'll check whether the hitCircleIndex of the note is the same as the column we hit (which was stored in 'i'). We then check whether a current song is playing (which should always be true if there are notes on the screen, but better safe than sorry) and if so, whether or not the difference between the song's time and the note's time is less than 0.6 seconds. If so, we're going to want to remove it from the screen. Also note that 0.6 seconds is an arbitrary value - smaller values make the game harder, larger values make it easier.

```
if (note.hitCircleIndex == i
    && GameScene.currentSong != nil
    && abs(GameScene.currentSong!.currentTime - note.time) < 0.6) {

}
```

Inside that if() statement, we then need to actually remove the note. To do this, we want to run a series of SKActions on the note. We'll run a sequence consisting of an animation then a removeFromParent() to delete the note. Our animation will actually be an SKAction.group(), which means two actions that happen at the same time. Specifically, we'll double the size and at the same time fade it out over 0.2 seconds. This creates the effect of the note bursting and disappearing when you hit it.

```
note.sprite!.run(SKAction.sequence(
          [SKAction.group([SKAction.scale(by: 2.0, duration: 0.2),
                           SKAction.fadeOut(withDuration: 0.2)]),
                           SKAction.removeFromParent()]))
                           onscreenNotes.remove(at: j)
                           cont = false
```

We can now get the buttons to shrink and notes to disappear when tapped, but currently the buttons just stay shrunk. We need to add some code inside of the touchUp() function that allows us to make the buttons bigger again!

```
func touchUp(touch: UITouch) {
    for i in 0...3 {
        var index = -1
        for j in 0..<activatedHitTouches[i].count {
            if (activatedHitTouches[i][j] == touch) {
                index = j
            }
        }

        if (index != -1) {
            activatedHitTouches[i].remove(at: index)
        }

        if (activatedHitTouches[i].isEmpty && !
hitButtons[i].hasActions()) {
            hitButtons[i].run(SKAction.scale(to: 1.2, duration:
0.03))
        }
    }
}
```

All this code does is detect when a finger is lifted and remove any hit touches that were on that button by iterating through the hit touches array and comparing them to the touch that was just lifted. If said touch was just lifted, we can remove it from the array. Then, we check if there are columns with no hit touches active, and if there are and the button is not currently scaling, we scale it up to its full size (this still runs if there aren't any touches and it's full size already, but it doesn't do anything if that's the case so we don't care).

Now notes should properly disappear when tapped, and you should have an interesting animation play when they disappear. You can customize this animation as you like - the SKAction documentation available on Apple's website has all sorts of animations you can apply to objects in your game.

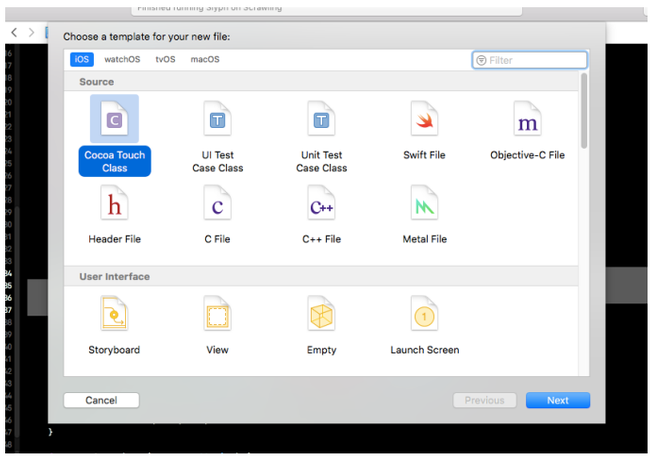
Section 8

Song Selection: Storyboard

Though we have a working game, the only way of selecting songs is through the code - that is not very convenient when playing the game. We want to be able to select a song and difficulty in the app itself before starting the game with those two parameters.

We will be using apple's interface builder, or storyboard, heavily in tandem with our code, and we will be introducing segues and how to call them programmatically.

To set up our program files, we want to add new Cocoa Touch swift files. First, hit (⌘+N) to bring up the new file menu. Once it has appeared, select the "Cocoa Touch Class" option before hitting "Next". These contain all the functions we need. Make sure the "Subclass" is "UIViewController."



Name the first file "SongSelectViewController". Repeat these steps to create a "DifficultySelectViewController" file. These files will house the two new classes for our two new view controllers that we will be creating in the storyboard.

We must create a second class in each file for managing something called a 'Cell'. We will go into more detail about this later, but essentially we need a class to keep track of our custom visual objects that we'll be using to show song banners. Open the SongSelectViewController.swift file and scroll to the bottom. We want to add a new class there called "SongSelectViewControllerCell". We want these classes to inherit from the "UITableViewCell" class. Once again, we will dive into that deeper later.

```
class SongSelectViewController: UIViewController { ... }

class SongSelectViewControllerCell: UITableViewCell { ... }
```

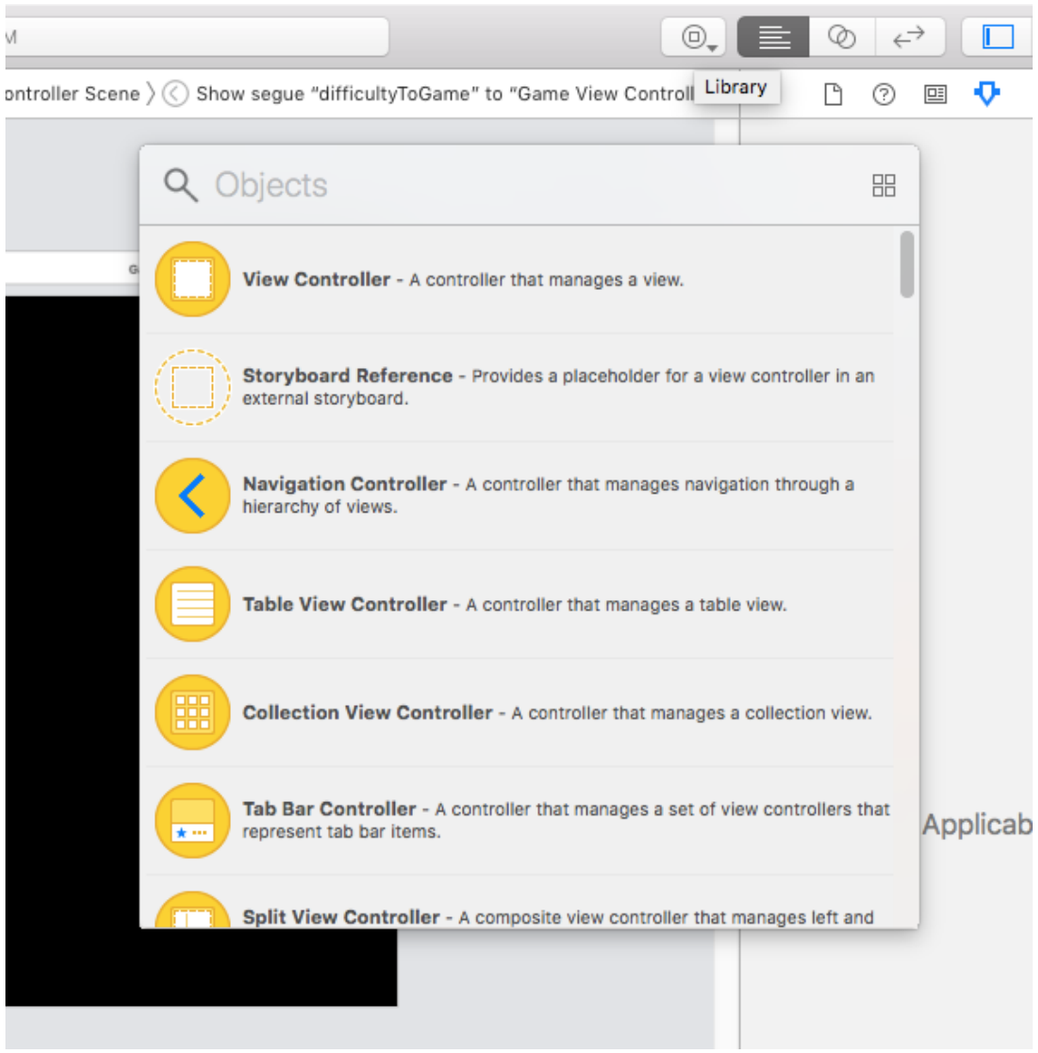
Now, we want to do something similar with the DifficultySelectViewController.swift file. We want to add a new class named "DifficultySelectViewControllerCell", and we want this class to inherit from the "UITableViewCell" class. Be aware that when you see a '...' in the code, that just means that something might be there - do NOT copy that in!

```
class DifficultySelectViewController: UIViewController { ... }

class DifficultySelectViewControllerCell: UITableViewCell { ... }
```

This concludes this coding mini-section. We have created the barebones files that we will need to continue with the storyboarding section next.

We are going to start creating the visuals that we will be interacting with. First, open up the project storyboard at main.storyboard, and find the black rectangle on the screen. That rectangle represents the Game View Controller that our Sylph game currently runs in.

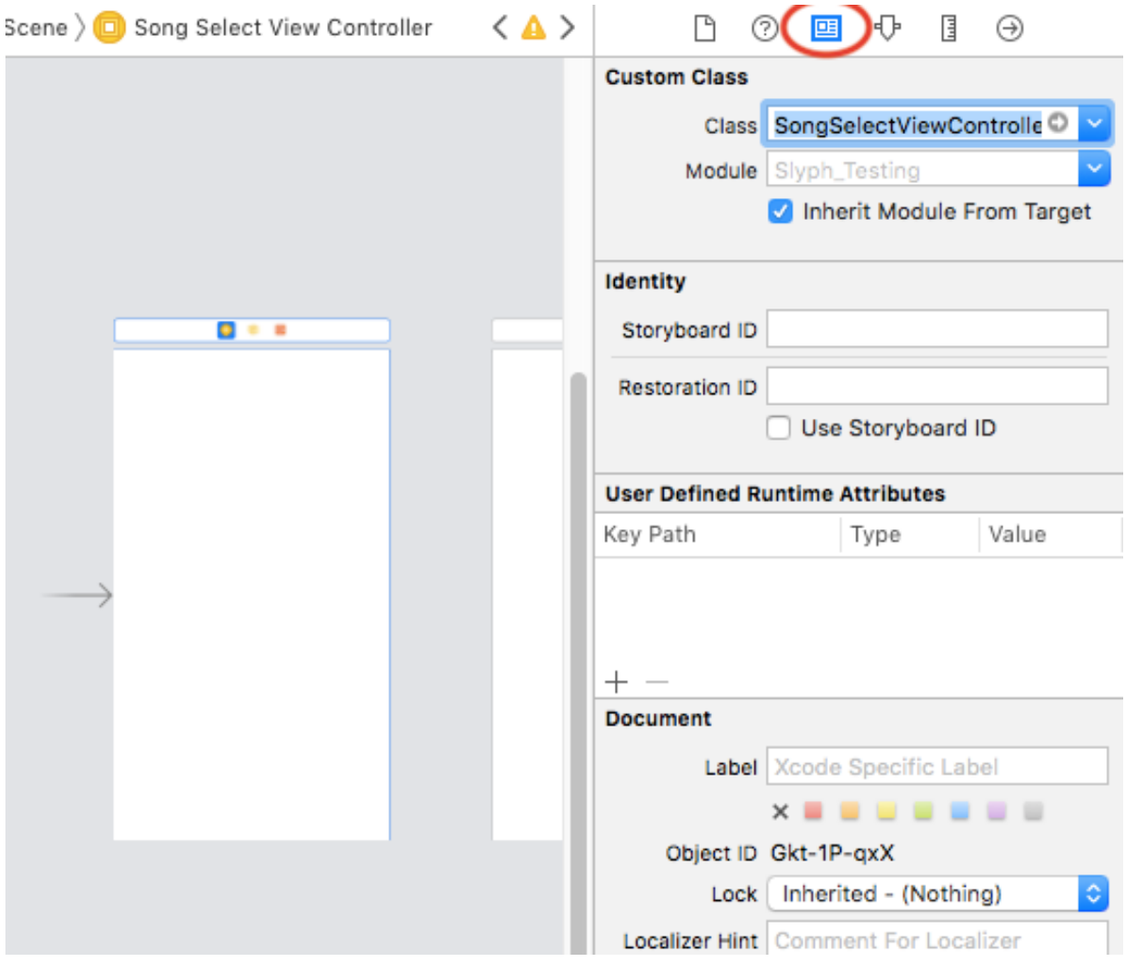
We have to add two more view controllers that will hold our Song Selection screen and then our Difficulty Selection screen. To do this, open the object library in the top right corner and find the ‘View Controller’.



We need to drag two of these view controllers into the storyboard so that it looks like the picture below. Afterwards, we want to move the grey arrow pointing to the Game View Controller so that it points to the first view controller on the left.

This grey arrow specifies what view controller that the program will start up on. We are changing it so that it will start on the to-be Song Selection View Controller.

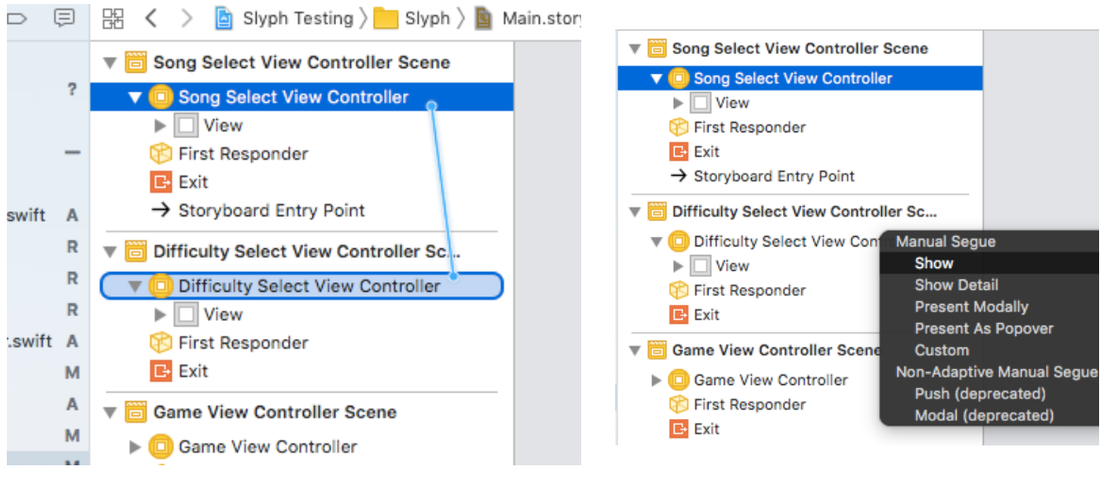
Next, select the first view controller on the left, and open the identity inspector to change the class. This view controller’s class should become “SongSelectViewController” as shown in the picture below.



Afterwards, select the second view controller and change its class to “DifficultySelectViewController”. Changing the class of the view controllers will allow us to programmatically change their attributes and manipulate them with code.

There is one final thing that we have to do before we start building the smaller parts of each view controller. To have a way to transition between different view controllers, we need a pathway. This pathway is called a segue, and if we give it a name, we can perform this segue with code.

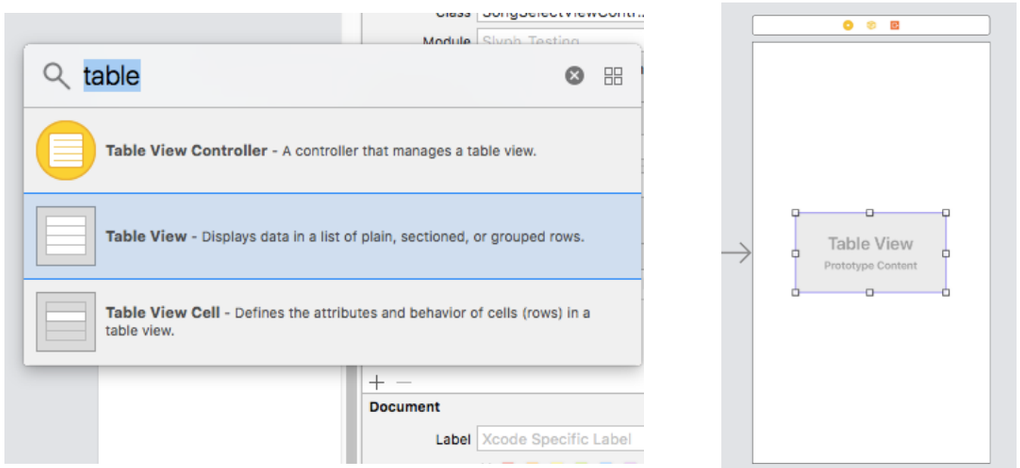
To add a segue, expand all menus of the storyboard, and find the item named “Song Select View Controller”. Next, hold the control key and click and drag from that view controller to the item named “Difficulty Select View Controller” in the element view. This will bring up a dark menu that allows us to pick the transition type - click ‘show’.



Now, once this segue is created, you must select it on the storyboard (it looks like an arrow between two view controllers), and the first one should be named “songToDifficulty”. Now, repeat this process to make a segue from the Difficulty Select View Controller to the Game View Controller, and name it “difficultyToGame”. Finally, we need to repeat this process one more time between the Game View Controller and the Song View Controller - name it “gameToSong”. With that, we can now focus on the smaller details of each view.

Now that we have set up the view controllers inside of the storyboard, we can start adding objects to them. On the song selection screen, we want to have a list of all songs that we can play. Swift provides a convenient way to do this with something called at Table View (UITableView). An example of this special view is the Reminders app.

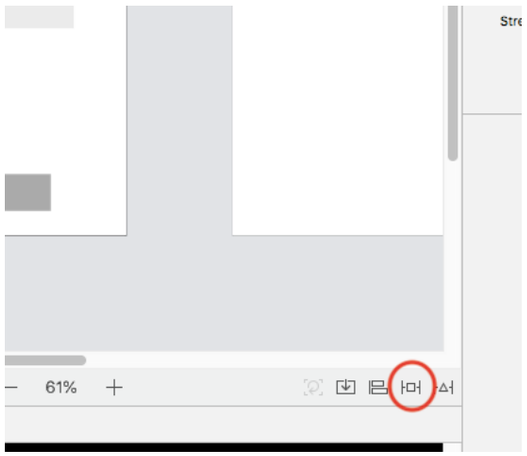
We are going to be working on the Song Selection View Controller first, so select that one (it should be the view controller on the far left). Open the object library again and find the Table View before dragging it into the scene.



Position it in the center of the screen. If we leave it like this, though, it will only be full screen for the specific type of iPhone that we are modeling on. For this view to be full screen on every device, we have to use **constraints**. This set of constraints is simple, but we must make sure that everything is in the view before we start locking things in place.

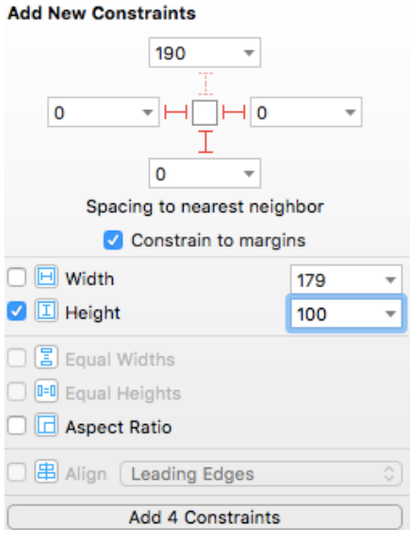
Before we can start using constraints, we have one more thing to add to the view - a ‘Next’ button. This will come in handy later when making the UI more user-friendly. Open the object library and drag in a Button (UIButton) to the bottom of the screen and set the button’s text to ‘Next’, the text color to black, and the background to grey.

Now we can start using constraints. First, we should lock the button into place before we starting messing with the Table View. Let’s open the constraints menu.



We want to pin the Next button to the bottom of the screen and give it a set height. To do this, we should set the spacing

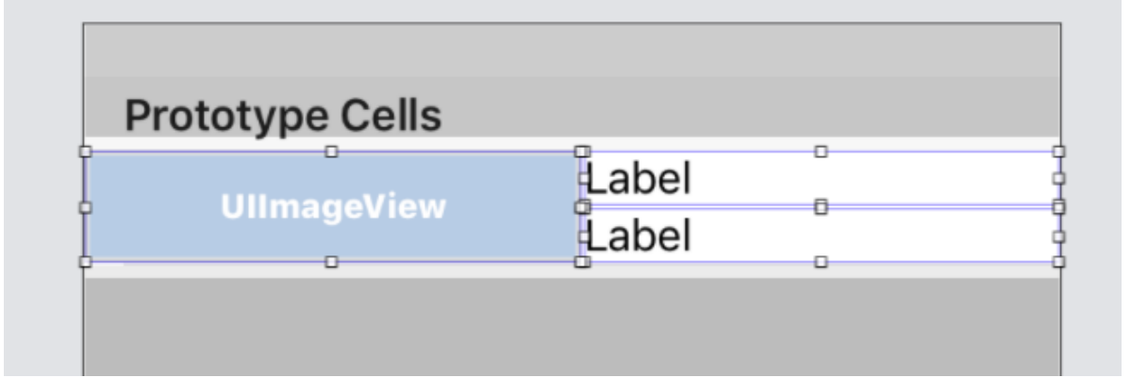
on the right, left, and bottom sides to ‘zero’ as well as set a fixed height. Let’s make that height 100 pixels or so. This can be smaller or larger as long as it is visible and does not take up more than half of the screen.



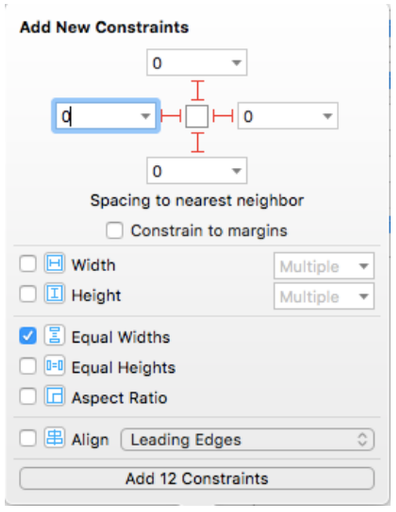
Once you hit “Add 4 Constraints”, you will see that the button snaps to the bottom of the screen. Now, we must snap the Table View into place. Open the constraints menu again and force all side spacing to be ‘0’. If you need help with adding constraints, call over a TA. You will see that the Table View snaps to take up the rest of the screen.

This next section is very constraint-heavy, and make sure to call a TA if you are having trouble adding all of the constraints properly. We are going to use something called a “Prototype Cell”. This is a cell that you add to the Table View that describes what all created cells will look like. Drag one of these cells (Named “Table View Cell”) into the Table View from the object library. If you remember, we created a class named “SongSelectionViewControllerCell” - open the identity inspector and change the cell’s class to that class.

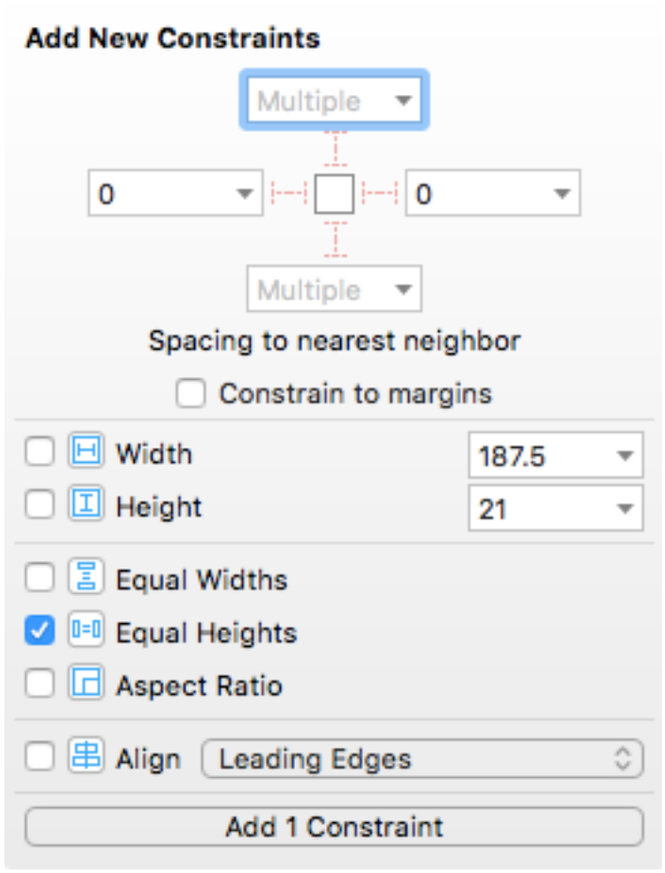
This next part will be the most complicated that our storyboarding will get. Open the object library and drag in an image view (UIImageView) as well as two labels. Position them like the picture below.



For these objects to stay like this across different devices, we must add constraints. With all three selected, open the constraints tab and check “Equal Widths”. Also, fill in ‘0’ for all side constraints.

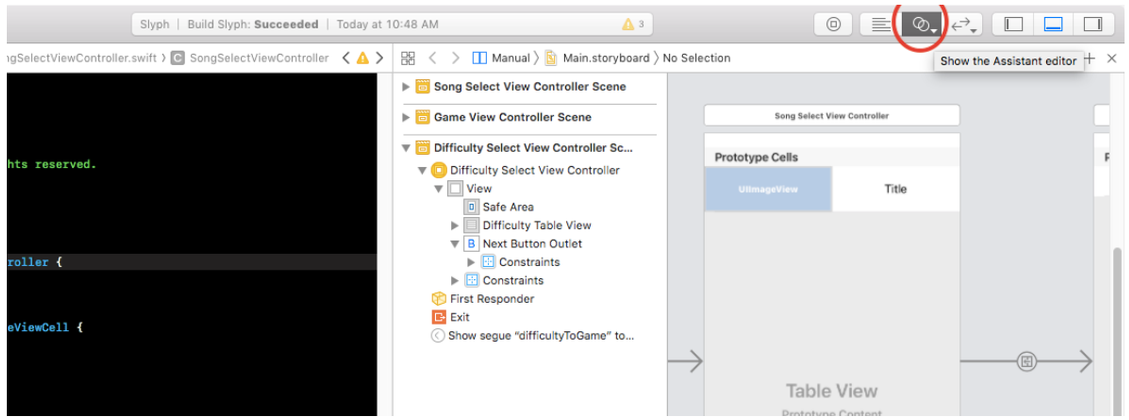


Next, with **only the two labels selected**, open the constraints menu and select “Equal Heights”. Finally, select the cell and open the identity inspector. Find



We are done with the Song Selection View Controller’s storyboard building! Now, we must link up all of our objects with code.

Open the assistant editor, and put the Song Selection View Controller and the SongSelectionViewController.swift file side-by-side.



We are going to drag our new objects into our files. First, select the “Table View Cell” in the sidebar, and ctrl-drag it into the “SongSelectionViewController” class. Name it “songTableView”. Ctrl-drag the ‘Next’ button into the that same class - select **‘Outlet’** at the top and name it “nextButtonOutlet”. Ctrl-drag the ‘Next’ button again into the same class and select **‘Action’** at the top and name it “nextButton”.

Now, ctrl-drag the image and both labels into the “SongSelectionViewControllerCell” class. Name them “songImage”, “songTitle”, and “songDescription” respectively. The file should now look like the following picture.

```swift
import UIKit

class SongSelectViewController: UIViewController {
    @IBOutlet weak var songTableView: UITableView!
    @IBOutlet weak var nextButtonOutlet: UIButton!

    @IBAction func nextButton(_ sender: Any) {

    }

    override func viewDidLoad() {
        super.viewDidLoad()
    }


    /*
    // MARK: - Navigation

    // In a storyboard-based application, you will often want to do a little preparation before navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Get the new view controller using segue.destination.
        // Pass the selected object to the new view controller.
    }
    */

}

class SongSelectViewControllerCell: UITableViewCell {
    @IBOutlet weak var songImage: UIImageView!
    @IBOutlet weak var songTitle: UILabel!
    @IBOutlet weak var songDescription: UILabel!
}
```

The steps for preparing the “DifficultySelectViewController.swift” file and its respective view controller is very similar to the song selection’s steps. The only two differences are naming and the number of objects in the prototype cell.

Like before, we are going to add a Table View (UITableView) and a ‘Next’ Button (UIButton). This part of the process is exactly the same, so you can just refer back to the previously described process for adding the two objects in and constraining them. The process differs once we start using the prototype cell, though.

Drag in a prototype cell if you haven’t already, and then drag a Label (UILabel) into the prototype cell. Instead of adding more objects and locking them into the same heights or widths, we are just going to set all side constraints to ‘0’. This will take up the entire prototype cell.

Enter the identity inspector for the prototype cell and change its class to “DifficultySelectViewControllerCell” so that we can start linking its label to the code.

Like we did before, open the “DifficultySelectViewController.swift” file and the storyboard side-by-side so that we can ctrl-drag across the workspaces. Ctrl-drag the Table View (UITableView) into the “DifficultySelectViewController” class. Next, ctrl-drag the ‘Next’ button twice in like we did before in the Song Selection part - one Outlet and one Action.

Finally, we drag the label inside of the prototype cell into the “DifficultySelectViewControllerCell” class as an outlet. Name it “difficultyLabel”. With that, we are ready to start the code!

Section 9

Song Selection: Coding

While we can have as much visual design as we want, it won't do anything without the code. The code part is the most important of this section, and it is required to get the system to work.

The first thing that we have to do is complete the Song Select View Controller's functionality. First, we must create a few global variables to keep track of important data throughout the menu:

```
@IBOutlet weak var songTableView: UITableView!
@IBOutlet weak var nextButtonOutlet: UIButton!

var selectedSong: Int?
var strokeTextAttributes: [NSAttributedString.Key: Any]?

var songs: [Song] = BlackBoxConverter.getSongs()
```

Next, we have to make sure that our UITableView knows where to get its information. To do this, we must have our SongSelectViewController class inherit from two more places:

```
class SongSelectViewController: UIViewController,
UITableViewDataSource  UITableViewDelegate {
```

Note that that is only one line of code. It replaces the class declaration at the beginning of the file. Ignore the errors that come up! We will fix them soon.

We want a fixed row height of around 100. Also, to tell the Table View that it will receive its information (the objects to go into rows) from this class, we must do three things in the viewDidLoad function:

```
super.viewDidLoad()

songTableView.rowHeight = 100

songTableView.delegate = self
songTableView.dataSource = self
```

Those extra inheritance classes that we added in the step before last enabled this class to be a delegate and data source of a Table View. The step that we just performed made sure that the Table View knew that this class is these two things. If you start getting errors about this class not having the functions required to be these two things, ignore it - we will fix that in a minute.

Now, we have to set the font information into the variable 'strokeTextAttributes' that we declared globally a few steps ago. Put this in the viewDidLoad() function underneath the code we just entered.

```
strokeTextAttributes = [
    .strokeColor : UIColor.black,
    .strokeWidth : -2.0
]
```

You can change these settings to your own preference, however remember that the background is white.

There are three main functions that we must define in order to use this class as a Table View (UITableView) delegate and data source. You don't have to worry about what exactly a delegate is, but just know that it is used to reference another object and offload some work onto it. It's not too important for this stage in your programming career.

These three functions include one to get the number of songs to display, one to get the information (text, images, etc.) for each row, and one to perform a function once a row is selected. Here the three functions are, and an explanation will be provided afterwards:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection
      section: Int) -> Int {
      return songs.count
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath:
      IndexPath) -> UITableViewCell {
      let cell: SongSelectViewControllerCell =
      tableView.dequeueReusableCell(withIdentifier: "song", for:
      indexPath) as! SongSelectViewControllerCell

      cell.songTitle.attributedText =
      NSMutableAttributedString(string: songs[indexPath.row].name,
      attributes: strokeTextAttributes)

      let imageData = try? Data(contentsOf:
      songs[indexPath.row].bgURL)

      if (imageData != nil) {
            let image = UIImage(data: imageData!)
            cell.songImage.image = image
        }

      return cell
}
```

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
     IndexPath) {
     selectedSong = tableView.indexPathForSelectedRow?.row
     nextButtonOutlet.backgroundColor = UIColor.blue
}
```

These three functions are *required overrides*, meaning that any class extending UITableViewDataSource and Delegate must provide these functions. They do things for the parent - in other words, we don't call these functions, but when Swift starts visually setting up the table, it uses them. These functions are where we provide our settings for the custom table - things like the row count and what those rows will look like.

The first function just returns the number of rows that will be loaded. This is equal to the number of songs, so we just return the number of objects in the `songs` array.

The second function gives the parent what the rows look like. Ours generates rows using information from the `songs` array. Essentially, it changes the information on the labels to fit the data for each individual song. Right now, the description label is set to nothing because our black box converter doesn't load the description yet. This is why we had to create a custom cell class - we are editing those labels and image views in this function.

The third function sets the `selectedSong` variable to the index of the selected row, and it activates the 'Next' button for use.

Next, we must add in code to save the song that we selected and pass it on to the next view controller through our segue. This can be done by determining the destination view controller and changing the global variables to match the ones we know. This code goes inside of the nextButton action:

```
@IBAction func nextButton(_ sender: Any) {
        if (selectedSong != nil) {
                performSegue(withIdentifier: "songToDifficulty",
        sender: self)
        }
}
```

This snippet of code is the reason why we had to name our segues when we created them. The performSegue function can only perform segues with a given name or ID.

Finally, we want to make sure that our selected song is passed into the next view controller. To do this, we use a neat trick with segue attributes. When you created the Cocoa touch class, it was created with a special function: prepare. This function is called when a segue is about to be performed, and it allows us to do some very convenient things. Ignore the errors that come up for now!

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if (segue.identifier == "songToDifficulty") {
                let destinationVC = segue.destination as!
DifficultySelectViewController
                destinationVC.selectedSong = selectedSong
                destinationVC.strokeTextAttributes =
strokeTextAttributes
        }
}
```

First, we must make sure that the segue being performed is this specific segue between the Song and Difficulty view controllers. This is a good practice because we may add new segues from this view controller later. Then, we create a variable that is an alias of the destination view controller - it allows us to edit its variables before the segue is called. Finally, the variables are passed along like normal. With that, the Song Select View Controller is complete! We can now move onto the Difficulty Select VC.

Setting up the Difficulty Select VC is extraordinarily similar to setting up the Song Select one. It only has a few extra variables and a few naming changes between them. To make it clear, though, we will step through the entire process again with less explanation from the first time.

First, we must set up our global variables to hold our important information across the class:

```
@IBOutlet weak var difficultyTableView: UITableView!
@IBOutlet weak var nextButtonOutlet: UIButton!

var selectedSong: Int?
var selectedDifficulty: Int?
var strokeTextAttributes: [NSAttributedString.Key: Any]?

var song: Song?
var difficulties: [Difficulty]?
```

Second, we must make the class able to be a delegate and data source for our UITableView. Replace the class declaration with:

```
class DifficultySelectViewController: UIViewController,
UITableViewDataSource  UITableViewDelegate {
```

Next, we must give these variables values and give the Table View a data source and delegate in our viewDidLoad function:

```
super.viewDidLoad()

song = BlackBoxConverter.getSongs()[selectedSong!]
difficulties = (song?.difficulties)!

difficultyTableView.rowHeight = 100
difficultyTableView.delegate = self
difficultyTableView.dataSource = self
```

We must include those same three functions for the Table View as before. Keep in mind that they are different than the Song Select VC's, even though the idea is the exact same:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection
      section: Int) -> Int {
      return difficulties.count
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath:
      IndexPath) -> UITableViewCell {
      let cell: DifficultySelectViewControllerCell =
      tableView.dequeueReusableCell(withIdentifier: "difficulty",
      for: indexPath) as! DifficultySelectViewControllerCell

      cell.difficultyLabel.attributedText =
      NSMutableAttributedString(string:
      String(song!.difficulties[indexPath.row].difficulty),
      attributes: strokeTextAttributes)

      return cell
}


func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
    IndexPath) {
    selectedDifficulty = tableView.indexPathForSelectedRow?.row
    nextButtonOutlet.backgroundColor = UIColor.blue
}
```

Now, we must add to the prepare and nextButton functions to perform our needs for this program. These will do the same thing as the Song Select VC, except it will pass both a song selection AND a difficulty selection to the next view controller (which will be the Game VC):

```
@IBAction func nextButton(_ sender: Any) {
        if (selectedSong != nil) {
                performSegue(withIdentifier: "difficultyToGame",
        sender: self)
        }
}

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if (segue.identifier == "difficultyToGame") {
                let destinationVC = segue.destination as!
GameViewController
                destinationVC.selectedSong = selectedSong
                destinationVC.selectedDifficulty =
selectedDifficulty
        }
}
```

With that, we are done with both our difficulty and song selection! Now, we only have to do something with our new data. There is a quick modification to two files we must make - GameScene.swift and GameViewController.swift.

In Game View Controller, we have to add two global variables:

```
class GameViewController: UIViewController, FinishGameProtocol {

    var selectedSong: Int?
    var selectedDifficulty: Int?
```

In the Game VC, we must add a few lines of code inside of the ‘if let’ statements inside of viewDidLoad:

```
GameScene.difficultyToPlay = selectedDifficulty!
GameScene.currentSong = BlackBoxConverter.getSongs()[selectedSong!]
// Present the scene
view.presentScene(scene)
```

Inside of GameScene.swift, we must delete a single line that we created earlier. When we were testing the game, we made this line to automatically pick the first detected song. Now, the user can select the song:

~~GameScene.currentSong = BlackBoxConverter.getSongs()[0]~~

Next, find and delete the line with strikethrough and replace it with the next line.

~~if let scene = SKScene(fileNamed: "GameScene") {~~
```
if let scene = GameScene(fileNamed: "GameScene") {
```

One final touch on our game is to reset to the main menu once the game has finished. This will get rid of the need to quit the app every time you want to play another song.

To do this, we will be using protocols and delegates - the only thing you have to know about these two things is that they enable a child class to call a function inside of a parent class. In this case, the parent class is the Game View Controller, and the child class is the Game Scene.

First, we have to create the protocol that houses the function we will be calling. This will go **right above the ‘class GameScene’** inside of ‘GameScene.swift’:

```
protocol FinishGameProtocol {
    func returnToMainScreen()
}

class GameScene: SKScene {
```

Next, we need to add some variables to keep track of the various delegates we will use to call this function. Add a global variable inside of GameScene:

```
public static var currentSong: Song?
public static var difficultyToPlay: Int?

var finishGameDelegate: FinishGameProtocol?
```

Now, we need to make a function that will exit our game and use the protocol to use the segue “gameToSong” that we made earlier. Add it to the bottom of the ‘GameScene’ class:

```
func exitGame() {
    finishGameDelegate?.returnToMainScreen()
}
```

As you can see, this function uses the delegate that we made earlier to access to function inside of the protocol. We just need to add one more thing inside of the GameScene class so that this function will get called when the game is over. Add this inside of the update() function:

```
let cSong = GameScene.currentSong!
let cDifficulty = cSong.difficulties[GameScene.difficultyToPlay!]

if (GameScene.currentSong!.currentTime >=
    GameScene.currentSong!.length + 3) {
       exitGame()
}
```

Next, we just have to add in the code in the GameViewController to add code to our delegate that will call the segue. We are almost done!

```
GameScene.difficultyToPlay = selectedDifficulty!
GameScene.currentSong = BlackBoxConverter.getSongs()[selectedSong!]

scene.finishGameDelegate = self
```

Finally, we just have to implement the protocol, add a global variable, and define the function returnToMainScreen() from the protocol inside of GameViewController. First, let's add the global variable:

```
var selectedSong: Int?
var selectedDifficulty: Int?

var active: Bool = true
```

Next, implement the protocol inside of GameViewController by implementing it. Add the 'FinishGameProtocol' after 'UIViewController':

```
class GameViewController: UIViewController, FinishGameProtocol {
```

Finally, we must define the function that takes us back to our Song Select View Controller. This uses the protocol's function name so that the child class GameScene can call it in place of this parent class:

```
override var prefersStatusBarHidden: Bool {
    return true
}

func returnToMainScreen() {
    if (active == true) {
          performSegue(withIdentifier: "gameToSong", sender: self)
          active = false
    }
}
```

With that, our game should return to the main menu after the songs are complete. Have fun!

Section 10

Moving Forward with Sylph

Now, you can take the project and add some stuff to it to make it your own. We've compiled a list of some stuff that isn't overly difficult but would make a huge difference below:

- Add a scoreboard
- Add a combo meter
- Add in the hit reacts we provisioned for in the global variables (when a note is hit, before the note disappears, add a react above it with SKActions on it to fade out and delete after it's done)
- Add leaderboards
- Add GameCenter integration
- Pretty up the song select screen
- Improve view transition animations
- Add a "quit" button while in the song
- Add an option to skip the song intro
- Add an announcer
- Adjust notes not to cover each other in fast songs (hint: mess with z-positions. Each note should have a smaller one assigned than the previous when it's created, starting from some huge value. Or, you can change the z-positions depending on how far the note is down the track.)

You can change it however you like - we promise not to sue.

Next Steps

Congratulations! You've made at least one game! This is a monumental milestone, and almost everyone gives up before reaching it. You are ahead of most second-year college computer science majors at this point. It's time to start thinking about how to gain the skills needed to progress forward on your own. In truth, a lot of the necessary skill is in using Google effectively. Most software developers and game developers approach projects knowing only the general layout they're going to use, and then constantly google things like "How to make buttons in Swift", then copy-paste the result into their code, see if it works, and if it does, try to figure out why. Part of this is because there are a lot of things in Swift, like many programming languages, that are best done in a specific way that doesn't change. The two paths are either a) memorize that code snippet or b) look it up every time you want to do that thing, and in general it's not good to spend time memorizing things in programming when you can just look them up instead.

So use a lot of Google. That's my number one tip.

There are also a ton of tutorials available on the internet, though. You can find one for nearly every type of game you could possibly want to make. Some are good, some are bad - but it should be fairly obvious whether the one you're looking at is worth your time just by questioning whether or not you understand it. If you do, it's good, and if you don't, it's not worth reading through - at least until you get to a level where you understand what they're doing.

If you are trying to make a specific type of game, my first step is usually to look up a tutorial of somebody making a similar type of game, then see how they lay things out and how they do the same things that I'll also need to do for my game. Then, if I can't figure out how to do the things that are different in the game I'm trying to make, I'll Google that. I haven't failed to find somebody trying to do the exact same thing as me yet. Also, as you get better, you should have a better mental library of how to do different parts of your games, and should begin to be able to figure out how to do most things from scratch, even if not in the best possible way.

These tips apply to app development as well, and programming in a general sense. The only way to get good at it is by doing it a lot, and along the way failing a ton and making a lot of mistakes. At first, your code will be *mostly* mistakes, actually. But that changes quickly. There are only so many types of mistakes you can make in programming, and you figure out what causes what pretty quickly. It also helps that nearly every error message can be searched on Google

and one of the millions of people asking similar questions on StackOverflow will have received a professional and easy-to-understand answer. Eventually, you'll get to the point where you Google things more as a quick reminder rather than to try to wrap your head around the conventions surrounding the solutions to the problems you want to solve. Once you get there, the speed with which you make new things and the quality that those things are produced at should jump dramatically, and you'll be able to put down the vision in your head of a completed game or app into code not just quickly but well.

Another thing to consider is that you can, at this point, fairly easily switch platforms. There are two I'd recommend for making games or AR apps - Unity and Unreal. These simplify the whole process an incredible amount, and almost entirely remove complicated programming from the equation. If you use Unreal, you can get away with never touching code - and if you do need to, it has a drag-and-drop visual "coding" style called Blueprints that you can use. If you want to use code in these, both engines have a very refined way of blending coding with a smooth visual editor. If you want to keep making apps, you can also switch to Android Studio at this point and make Android apps - the skillset is very similar.

Programming isn't easy. But it enables you to create, and is a very marketable skill considering the relatively low number of people who get past the entry barrier - which, by this point, you already are! Continue to make things in your free time and you'll be a pro before you know it - and have made a lot of cool games and software along the way.

Fight on, and good luck!

AppDelegate.swift

Handles response to changes in the app window’s state - essentially, the functions inside AppDelegate run whenever the app starts, is put into the background, reactivates into the foreground, or changes state in any way. It can be used to do things like pause a game when the user enters the notification tray and resume it when they swipe back up into the game.

Related Glossary Terms

Function

Index

Find Term

Array

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

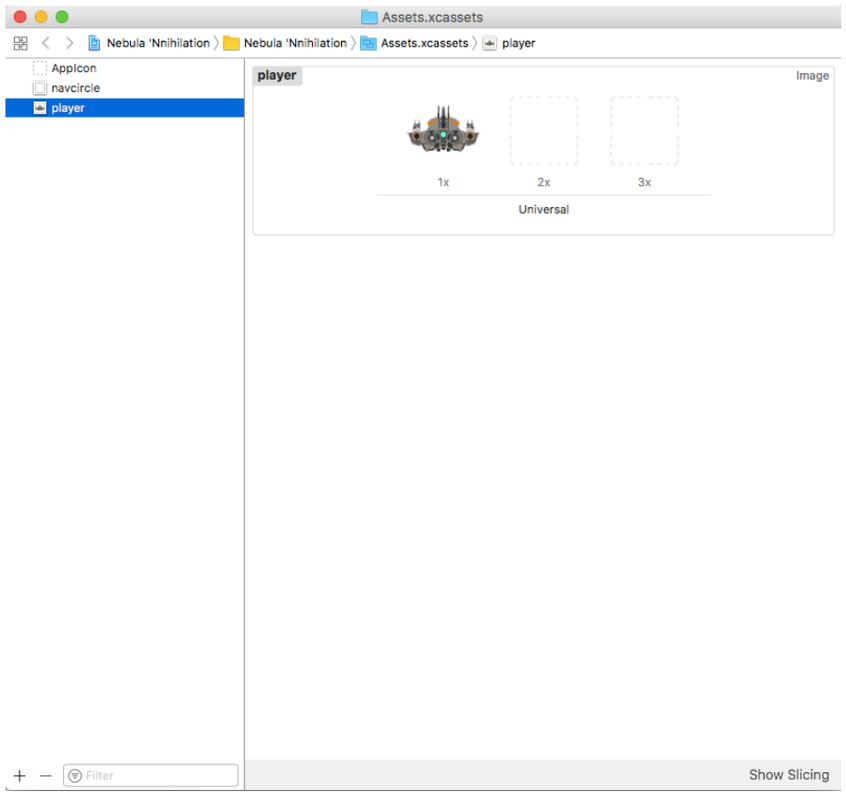
Related Glossary Terms

Drag related terms here

Index Find Term

Assets.xcassets

This file stores your project's assets - mostly used for images. Files can be added to your project by double clicking on the *Assets.xcassets* file in your project's file explorer and then dragging files into the editor that pops up. These files are then automatically imported into your scene editor's assets library and can be referenced in code using the file's name.



Related Glossary Terms

Element picker, SKScene

Index Find Term

Beatmaps

A beatmap is the term for a song file used by rhythm games. It takes any music file - mp3, ogg, etc. and adds a note pattern that you can play in a game that’s synced up to the music. There are a lot available online for various different games - some prominent free ones would be Stepmania and Osu.

Related Glossary Terms

Rhythm Game

Index Find Term

Chapter 3 - Sylph

Boolean

A boolean is something that is either ‘true’ or ‘false’. It can be a variable:

a: Bool = true (which creates a boolean variable ‘a’ and sets it to be ‘true’. The same effect can be accomplished by simply writing “a = true” - Swift is able to infer the type of the variable in most cases)

or an expression:

1 == 2 (which would be a false expression. See ‘Boolean expression’ for more information on expressions)

Boolean variables can also be set to the result of these expressions:

a: Bool = (1 == 2) (which creates a boolean variable ‘a’ equal to the result of 1 == 2, which is false)

As a note, the difference between ‘=’ and ‘==’ is that ‘=’ sets the value on the left to the value on the right, whereas ‘==’ compares the two and returns ‘true’ if they are equal and ‘false’ otherwise.

Related Glossary Terms

Boolean expression, GameViewController.swift, IF Statement, Logical Operators

---

Index Find Term

Boolean expression

A boolean expression compares two variables, and returns ‘true’ if the statement is true or ‘false’ otherwise. They can be combined with logical operators to form more complex expressions. Boolean expressions are commonly used in IF statements.

The operators used to construct boolean expressions are as follows:

== (equals to)

>= (larger than or equals to)

<= (less than or equals to)

!= (not equals to)

Related Glossary Terms

Boolean, Logical Operators

Index Find Term

Compile

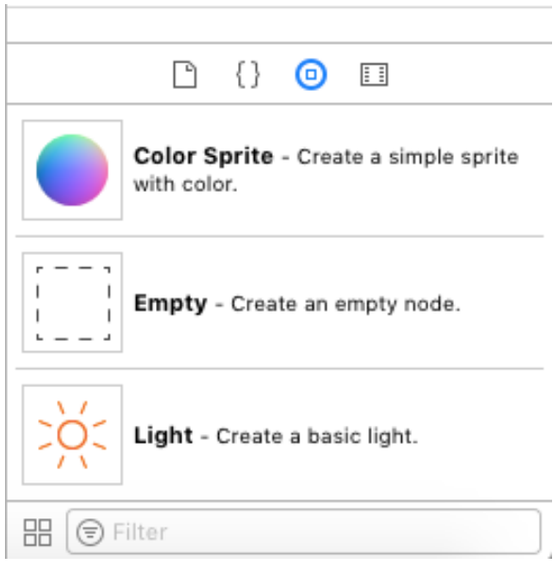
Compiling is the process of taking your source code and turning it into an app that your phone can recognize.

Related Glossary Terms

Drag related terms here

Index Find Term

Element picker



The element picker allows you to drag UIKit elements, SpriteKit elements, SceneKit elements, and images from Assets.xcassets into the scene editor.

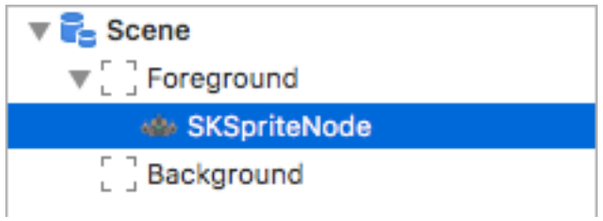
Related Glossary Terms

Assets.xcassets, SKScene, SpriteKit, UIKit

Index

Find Term

Element view



The section at the left hand side of the scene editor that shows the current elements that have been added to the scene.

Related Glossary Terms

SKScene

Index Find Term

Extended for loop

An extended for loop defines a block of code that will execute once for each element in a given array, with said element being stored in a temporary variable. The syntax is as follows:

```
for tempVariableName in arrayName {

    //Do whatever, with tempVariableName being a variable that holds the element of the array that is currently being worked on. You don't have to define this variable using let or var - it is created as part of the for loop.

}
```

Related Glossary Terms

Drag related terms here

Index Find Term

Function

A function is a block of code that can be executed by simply stating the name of the function in question. More complex functions have parameters and return values - this type of function is more like a mathematical function, where the input is operated on and then an output is given back.

Code-block function:

```
func printHello() {
    print("Hello")
} //Prints "Hello"
```

This function can then be executed by placing the statement 'printHello()' anywhere in your code. Said statement will then be simply replaced with whatever code is inside the function when the program runs. It's a great way to simplify reusing code and ensure your program stays concise and readable.

Complex function:

```
func add(a: Int, b: Int) -> Int {
    return a + b
} //This function takes two variables, a and b, and returns the sum.
```

This function can be executed by placing the statement 'add(a, b)' anywhere in your code, where 'a' and 'b' are any two integers (including the literals - you can simply put '5' or any other number there instead of a variable if you like). On runtime, the statement 'add(a, b)' will be replaced by the return value - whatever is after the statement 'return' in the function will replace the function call in your code.

See 'Parameter' and 'Return Value' for explanations of those concepts.

Related Glossary Terms

AppDelegate.swift, Game Loop, GameViewController.swift, Global variable, Local Variable, Parameter, Return Value

---

Index Find Term

Game Loop

A game loop is a loop that activates sixty times per second (usually, as it's bound to your screen's refresh rate - typically 60hz) and each time performs operations to update the objects on your screen. Each time it activates, that's called a 'Tick' - so there are sixty ticks per second. A usage example: If you have a background moving down at a rate of sixty pixels per second, each time the game loop runs (each tick), that background should be moved down by one pixel.

A common technique for moving things more slowly or infrequently is to use the modulo function. By taking the current frame and passing it to a modulo function with the number of frames between action executions as a parameter, it is possible to make events that only happen every 'x' frames.

The modulo function in Swift is denoted by the '%' character.

Example: "currentFrameCount % 2 == 0" is a boolean expression that is true when the frame count is even - in other words, it will be true every other frame. This could be used to make the background from earlier only move thirty pixels per second, by using it as the condition for an if statement.

Related Glossary Terms

Function, IF Statement

Index Find Term

GameScene

GameScene.sks and *GameScene.swift* are two sides of the same file - the visual side that can be manipulated through Xcode's built in scene editor, and the code side that can be linked to elements in the visual side and directly and far more accurately manipulate those elements over time as the game runs. Typically elements are added to the scene visually, by drag and drop, into the *GameScene.sks* file, have their starting properties edited in the editor pane in the file, and then given actions and responses to certain events via code in the *GameScene.swift* file.

Related Glossary Terms

SKScene

Index Find Term

GameViewController.swift

The file that loads scenes onto the main screen. Though we will not have to touch that part for this game, as by default Xcode generates a line loading the app into the *GameScene.swift* scene that it generates. However, if you wanted to create a menu screen that the app loads into instead of just the main game, for instance, this file would be where you would do that (specifically, the line "`if let scene = SKScene(fileNamed: "GameScene")`" could be changed to refer to any scene file). Additionally, the following lines:

```
view.showsFPS = true
view.showsNodeCount = true
```

in the viewDidLoad() function are responsible for creating the node count and FPS counter that you can see at the bottom right of the screen when starting the project. Removing those is as simple as deleting those two lines of code, or changing the two booleans' values to false.

Related Glossary Terms

Boolean, Function

Index Find Term

Git

Git is a version control engine that allows you to access and modify all previous versions of your projects, defined by checkpoints known as ‘commits’.

Git-enabled projects - known as ‘repositories’ - can also be easily uploaded to a website called “GitHub”, which also serves a secondary function as a sort of social network for programmers. The engine can also be used to enable easy collaboration between multiple programmers on the same project by allowing two versions of the same project with different changes made to be merged together. There are many great tutorials for git usage on the internet, so google the term for more information.

Related Glossary Terms

Drag related terms here

Index Find Term

Global variable

Global variables are variables accessible throughout any functions in a file. They contrast with local variables, which are variables declared within a function - these variables are only accessible from the function within which they were created. Creating a global variable allows for any function in the file to work with the same data, and means that if there is a particular bit of data that you use in many functions, you don't have to recreate that variable within every function.

Look up "Scope programming" on your favorite non-Microsoft-owned search engine for more information / a better explanation.

Related Glossary Terms

Function, Local Variable

Index Find Term

IF Statement

An IF statement is an expression that will cause a given block of code to execute if and only if the specified condition is true. Said condition is a boolean expression.

Related Glossary Terms

Boolean, Game Loop

Index Find Term

Info.plist

Stores your project settings. Note that most options within are editable in the Xcode interface. Sometimes you'll have to dip into this file, but mostly just for editing supported interface orientations (landscape, portrait, and a few others). This can be done by expanding the relevant sections ("supported interface orientations" and "supported interface orientations - iPad") and deleting entries or pressing the plus button to add one.

Related Glossary Terms

Drag related terms here

Index Find Term

Inheritance

See ‘Parent Class’.

Related Glossary Terms

Parent Class

Index Find Term

LaunchScreen.storyboard

A UIKit view that displays while the app is loading. By default, it’s just a white screen, but it’s fairly easy to recolor it or put an image on it - for instance, your game logo. However, if you do, ensure that it scales properly by using the constraint system in the editor, otherwise it might be off center on larger devices or stretch improperly.

Related Glossary Terms

UIKit

Index Find Term

Local Variable

Local variables are variables that are created within a function. These variables are only valid within the function that they are created in - if you try to use a variable created in one function from another function, it will fail. Variable names can be reused across different functions, as long as you don’t get confused - a variable declared within Function A can be named the same thing as a variable declared in Function B, because they refer to different things. However, a local variable should not be named the same as a global variable - the global variable, unlike another function’s local variable, is visible from within the function and thus should not be overwritten.

Look up “Scope programming” for more information / a better explanation.

Related Glossary Terms

Function, Global variable

Index Find Term

Logical Operators

Logical operators are used to construct complex boolean logic expressions. They consist of the following basic operators:

&& - and

|| - or

and can be used to combine boolean expressions as follows:

((1 == 1) && (2 == 3)) - 1 equals 1 AND 2 equals 3 - false, because while 1 == 1, 2 != 3, and && requires both to be true.

((1 == 1) || (2 == 3)) - 1 equals 1 OR 2 equals 3 - true, because only one of the two has to be true, and 1 == 1 is true.

Look up “Logical Operators”, “Logical OR”, “Logical AND”, or “Logical XOR” for more information. You may want to tack “Programming” onto the end of those searches, as the some of the articles that result from those base searches are about natural language or philosophical logic and not coding.

Related Glossary Terms

Boolean, Boolean expression

Index Find Term

Main.storyboard

This contains a collection of the UI (User Interface) screens present in the app, and is in this case simply a container for the game screen. We don’t have to worry about this file in this project either, but if you wanted to create a menu or scoreboard in UIKit instead of SceneKit, this is where that would be done. UIKit is not covered in this chapter, but is covered extensively in the Swift documentation for anyone interested in exploring further. It is mainly used for building UI-based apps - anything with menus. One example of a UI app would be Snapchat, which is primarily UIKit-derived.

Related Glossary Terms

UIKit

Index Find Term

Parameter

A parameter is a value that is given to a function. For instance, in the function call add(5, 6), the two parameters given are ‘5’ and ‘6’.

```
func add(a: Int, b: Int) -> Int {

    return a + b

}
```

In the example function declaration above, the two parameters taken are Integers. Parameter type is specified by the following syntax:

VariableName: Type

separated by a colon. The type is either a class or a fundamental data type (Integer, Boolean, Double, String, etc.) The variable name, which comes before the colon, is a placeholder name for whatever value is passed in in the function call. In the above example call add(5, 6), the function would set a = 5 and b = 6 and then return the value a + b (return 5 + 6 in our example call).

See ‘Return Value’ for an explanation of the ‘return’ statement and ‘Function’ for an explanation of functions.

Related Glossary Terms

Function, Return Value

Index Find Term

Parent Class

A child class, or subclass, *inherits* all of it's parent class' functions and variables. For instance, if you created a generic 'Car' class and implemented a 'Drive()' function in it, then created a 'Ford Fiesta' subclass that extended the 'Car' class, that Fiesta subclass would have a Drive() method that when executed would execute the Drive() method from the 'Car' class.

A class can inherit methods and variables from a parent class by placing a colon and the name of any parent classes after it's class declaration:

class Fiesta: Car, Vehicle {

where the Fiesta class inherits methods and variables from both the Car and Vehicle classes.

Look up 'Programming Inheritance' for more information and a myriad of uses for this concept. This is one of the core concepts in modern programming, and it is extremely important to understand inheritance fully - however, possibly the best way to get to understand it well is simply by spending a lot of time playing around with it.

Related Glossary Terms

Inheritance, SKAction, SKNode, SKScene

Index Find Term

Protocol

Protocols are a way of defining a certain structure that anything which choose to conform to it must adhere to. The various classes or variables that adhere to the protocol can then be manipulated generically by just the name of the protocol, instead of having to be individually and separately handled.

The official Swift documentation defines ‘Protocol’ as follows:

A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to conform to that protocol.

Related Glossary Terms

Drag related terms here

Index Find Term

Reference variable

A reference variable is a variable that represents something created in another file - in our game projects, this usually means something created using a visual editor. For instance, if we wanted to move a player sprite that we added to the screen using Scene Editor in our code, we would first have to create a reference variable that points to the player in the scene. We could then manipulate our new player variable, and the changes we made to it would be reflected by the player sprite in the scene. For instance, if we created a reference variable called playerReference that pointed to a player sprite in a scene, calling playerReference.moveTo(x: 50, y: 50) would move the player in the scene to (50, 50).

Related Glossary Terms

SKScene, Sprite

Index Find Term

Return Value

Return values are what are returned from a function after execution. They are entirely optional. They are denoted by a -> Type after the parameter list. The type can be a class or a fundamental data type (Integer, Boolean, Double, String, etc.)

Example:

```
func add(a: Int, b: Int) -> Int {

    return a + b

}
```

The '-> Int' signifies that when the function is called, that call will be replaced by an Integer after the function finishes executing. The statement after the 'return' is the specific Integer value it will be replaced by - so whatever a + b is, the function call will be replaced by that value. If we called it as follows:

```
let a = add(5, 6)
```

The function will execute, set a to 5 and b to 6, and then see that it needs to return 5 + 6. The line of code then essentially becomes the following in runtime:

```
let a = 5 + 6
```

See 'Function' for an explanation of functions and 'Parameter' for an explanation of parameters passed into them.

Related Glossary Terms

Function, Parameter

Index Find Term

Rhythm Game

A rhythm game is a genre of game involving music, where a series of notes are synced up to a song and the player has to hit them to the music. There is generally some form of combo and a measure for accuracy, and score is measured based on how close you get to each note and how long you can keep up a combo without missing a note.

Some examples include Dance Dance Revolution, Stepmania, Osu, Taiko, MaiMai, Just Dance, Love Live School Idol Festival, Rock Band, Rocksmith, Tap Tap Revolution, Deemo, Cytus, and Voez.

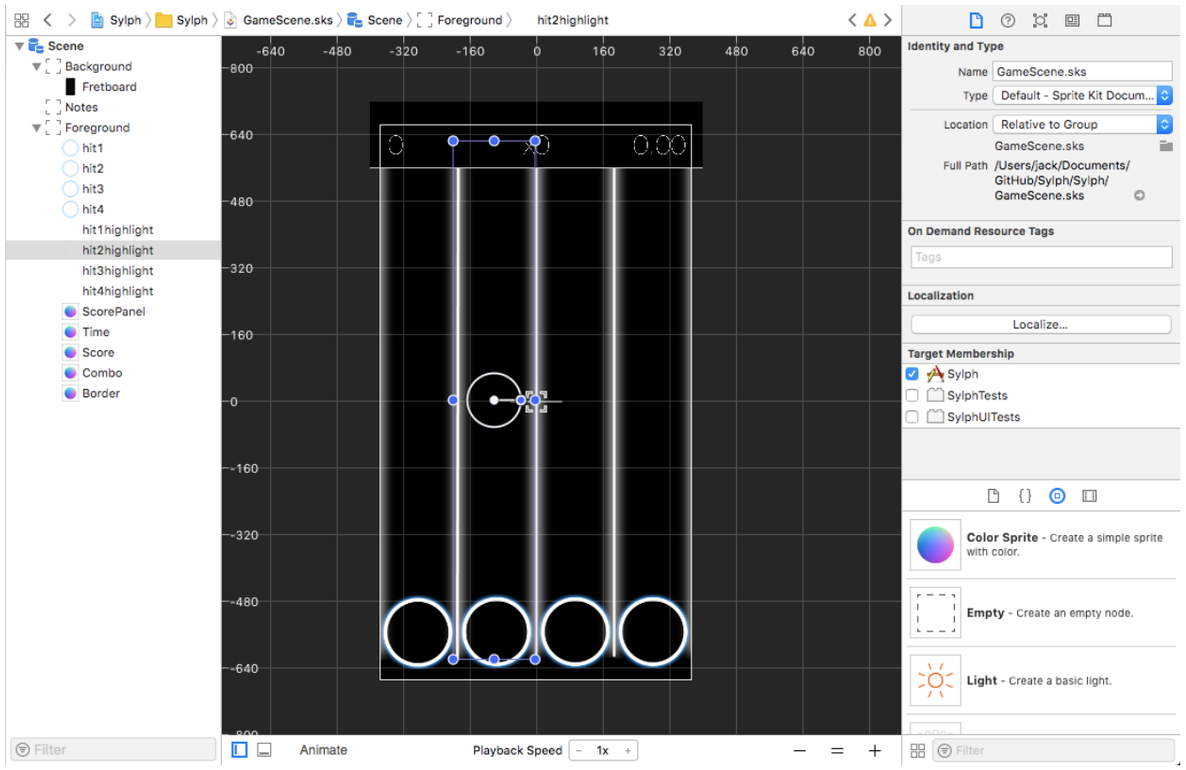
Related Glossary Terms

Beatmaps

Index Find Term

Scene Editor

The Scene editor allows you to visually create scenes and add things to them without touching the code. Physics can also be mostly handled entirely in the scene editor. It is possible to create a simple game entirely in the scene editor without touching code, but this is not usually advisable - using code will improve your skill and understanding of how the scene editor works and also allow you to do much more complex things than are allowed by default in the scene editor. Some types of games cannot even be attempted without heavy use of code because of the inherent limitations of the scene editor.



Related Glossary Terms

SceneKit, SKScene

Index

Find Term

Chapter 3 - Sylph

SceneKit

SceneKit is a framework for Swift that handles both 3D and 2D games, though we only cover the 2D side of it in this book. Within the 2D engine, SceneKit allows you to create and manipulate Scenes containing a collection of sprites, and then visually edit them using Xcode's SceneKit editor.

Related Glossary Terms

Scene Editor, SKScene, SpriteKit

Index Find Term

SKAction

An SKAction is a command you give to objects in the game - for instance, if you want to have a button that kills an enemy, you might want to cause the enemy's picture to fade out. You would use an SKAction to do this.

SKActions can do a ton of things - they can fade objects in or out, delete them from the scene, make them bigger or smaller, move them, animate them, and a ton of other things. The Swift documentation for SKAction has a full list.

There are two parent types - SKAction.sequence() and SKAction.group(). Sequences are arrays of SKActions that perform the actions one after another, and groups are arrays of SKActions that perform all of them at the same time.

SKActions are attached to GameObjects by calling object.run(action) where "action" is an SKAction. One example would be: ship.run(SKAction.scale(to: 1.2, duration: 1)), which would scale the "ship" object up to a size of 1.2 over one second.

Related Glossary Terms

Parent Class, SKNode, SKSpriteNode

Index Find Term

SKNode

An SKNode is a node in an SKScene thats primary function is to serve as a container for children. It is also the parent class for SKSpriteNodes and many other objects that can be placed in the scene.

Nodes can be added to a scene using the sceneName.addChild(nodeName) function. Children can be added to a node using nodeName.addChild(secondNodeName). When a node is added to a scene, all of its children will be added as well.

Related Glossary Terms

Parent Class, SKAction, SKScene, SKSpriteNode

Index Find Term

SKScene

An SKScene is a SceneKit container that stores primarily SKNodes and children of that class. They can be edited with the SKScene Editor in Xcode. An SKScene is typically what is displayed when a game is run. Multiple SKScenes can be present simultaneously.

Nodes can be added to a scene using the sceneName.addChild(nodeName) function.

The default SKScene for projects created using the ‘Game’ template is called GameScene. See the related glossary entry for more information.

Related Glossary Terms

Assets.xcassets, Element picker, Element view, GameScene, Parent Class, Reference variable, Scene Editor, SceneKit, SKNode, SKSpriteNode

---

Index Find Term

SKSpriteNode

An SKSpriteNode is a sprite present in an SKScene. SKSpriteNodes' images are by default drawn onto the SKScene that they are present in, so if said scene is visible, the sprite should show up on the screen unless the opacity (Alpha) is zero.

See 'Sprite'.

Related Glossary Terms

SKAction, SKNode, SKScene, Sprite

Index Find Term

Sprite

A sprite is a container for an image and its related data. Specifically, a sprite contains information like the image's position in the scene, its rotation, its opacity, and many other variables. If a sprite is added to a scene and that scene is visible and drawn onto the screen, the sprite will typically be drawn onto the screen as well.

Related Glossary Terms

Reference variable, SKSpriteNode, SpriteKit

Index Find Term

SpriteKit

SpriteKit is Apple’s 2D game development framework. It allows you to create sprites, position them, move them, execute actions on them, and apply physics to them without having to deal with all of the programming overhead involved in creating those functions from scratch.

Related Glossary Terms

Element picker, SceneKit, Sprite

Index Find Term

Swift

Swift is the programming language used for the coding parts of iOS development in this tutorial.

It is a fairly modern language, first released in June 2014 after seven years of development at Apple. Though the primary use of Swift is the development of iOS and Macintosh apps, it has also been ported to Windows and Linux for more general use. It is worth noting that it is not possible to use Swift to develop apps from a Windows or Linux-based machine. You have to have a Mac, and more specifically, you have to use Xcode.

Related Glossary Terms

Xcode

Index Find Term

UIKit

UIKit is Swift's User Interface builder. It is a way of visually constructing menus - buttons, text fields, labels, etc. - and linking them to functions in code. It is also used to construct multiple screens that can be transitioned between by using Segues. More information and tutorials on usage can be found in the Swift documentation.

Related Glossary Terms

Element picker, LaunchScreen.storyboard, Main.storyboard

Index Find Term

Xcode

Xcode is the editor used for writing Swift code. Think of it like Microsoft Word, but for app development instead of writing research papers.

More specifically, it's called an IDE - an Integrated Development Environment. This moniker refers to the fact that you can write code, edit user interface elements, create game scenes, test your code, and bundle your projects for release all from within the program. Xcode is the only IDE usable for Swift, but it also supports other programming languages, including C++ and Objective-C.

Related Glossary Terms

Swift

Index Find Term