IOS APP 1: PART 5 & PART 6

PREPARED BY FARHAJ AHMED

iOS App 1: Part 5

At this point, your game is fully playable. The gameplay rules are all implemented and the logic doesn't seem to have any big flaws. As far as I can tell, there are no bugs either. But there's still some room for improvement.

This chapter will cover the following:

- **Tweaks**: Small UI tweaks to make the game look and function better.
- **The alert**: Updating the alert view functionality so that the screen updates *after* the alert goes away.
- **Start over**: Resetting the game to start afresh.

Tweaks

Obviously, the game is not very pretty yet — you will get to work on that soon. In the mean time, there are a few smaller tweaks you can make.

The alert title

Unless you already changed it, the title of the alert still says "Hello, World!" You could give it the name of the game, *Bull's Eye*, but I have a better idea. What if you change the title depending on how well the player did?

If the player put the slider right on the target, the alert could say: "Perfect!" If the slider is close to the target but not quite there, it could say, "You almost had it!" If the player is way off, the alert could say: "Not even close…" And so on. This gives the player a little more feedback on how well they did.

iOS App 1: Part 5

Exercise: Think of a way to accomplish this. Where would you put this logic and how would you program it? Hint: There are an awful lot of "if's" in the preceding sentences.

The right place for this logic is `showAlert()`, because that is where you create the `UIAlertController`. You already do some calculations to create the message text and now you will do something similar for the title text.

➤ Here is the changed method in its entirety — replace the existing method with it:

```
@IBAction func showAlert() {
  let difference = abs(targetValue - currentValue)
  let points = 100 - difference
  score += points

  // add these lines
  let title: String
  if difference == 0 {
    title = "Perfect!"
  } else if difference < 5 {
    title = "You almost had it!"
  } else if difference < 10 {
    title = "Pretty good!"
  } else {
    title = "Not even close..."
  }

  let message = "You scored \(points) points"

  let alert = UIAlertController(title: title,  // change this
                                message: message,
                          preferredStyle: .alert)

  let action = UIAlertAction(title: "OK", style: .default,
                             handler: nil)
  alert.addAction(action)
  present(alert, animated: true, completion: nil)

  startNewRound()
}
```

You create a new string constant named `title`, which will contain the text that is set for the alert title. Initially, this `title` doesn't have any value. (We'll discuss the `title` variable and how it is set up a bit more in detail just a little further on.)

To decide which title text to use, you look at the difference between the slider position and the target:

- If it equals 0, then the player was spot-on and you set `title` to "Perfect!"
- If the difference is less than 5, you use the text, "You almost had it!"

iOS App 1: Part 5

- A difference less than 10 is "Pretty good!"
- However, if the difference is 10 or greater, then you consider the player's attempt "Not even close..."

Can you follow the logic, here? It's just a bunch of `if` statements that consider the different possibilities and choose a string in response.

When you create the `UIAlertController` object, you now give it this `title` string instead of some fixed text.

Constant initialization

In the above code, did you notice that `title` was declared explicitly as being a `String` constant? And did you ask yourself why type inference wasn't used there instead? Also, if `title` is a constant, how do we have code which sets its value in multiple places?

The answer to all of these questions lies in how constants (or `let` values, if you prefer) are initialized in Swift.

You could certainly have used type inference to declare the type for `title` by setting the initial declaration to:

```
let title = ""
```

But do you see the issue there? Now you've actually set the value for `title` and since it's a constant, you can't change the value again. So, the following lines where the `if` condition logic sets a value for `title` would now throw a compiler error since you are trying to set a value to a constant which already has a value. (Go on, try it out for yourself! You know you want to...)

One way to fix this would be to declare `title` as a variable rather than a constant. Like this:

```
var title = ""
```

The above would work great, and the compiler error would go away. But you've got to ask yourself, do you really need a variable there? Or, would a constant do? I personally prefer to use constants where possible since they have less risk of unexpected side-effects because the value was accidentally changed in some fashion — for example, because one of your team members changed the code to use a variable that you had originally depended on being unchanged. That is why the code was written the way it was. However, you can go with whichever option you prefer since either approach would work.

iOS App 1: Part 5

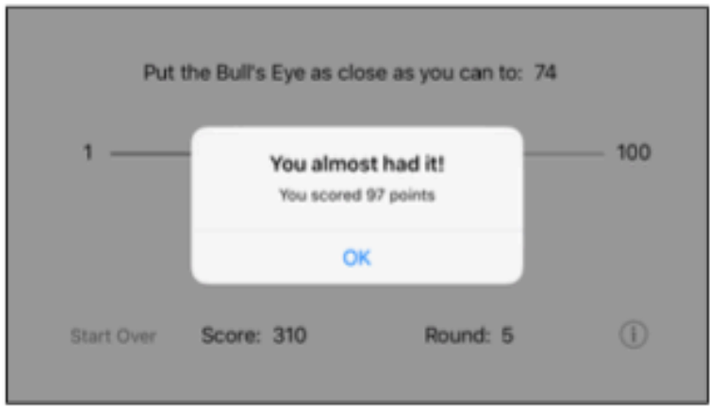
But if you do declare `title` as a constant, how is it that your code above assigns multiple values to it? The secret is in the fact that while there are indeed multiple values being assigned to `title`, only one value would be assigned per each call to `showAlert` since the branches of an `if` condition are mutually exclusive. So, since `title` starts out without a value (the `let title: String` line only assigns a type, not a value), as long as the code ensures that `title` would always be initialized to a value before the value stored in `title` is accessed, the compiler will not complain.

Again, you can test this by removing the `else` condition in the block of code where a value is assigned to `title`. Since an `if` condition is only one branch of a test, you need an `else` branch in order for the tests (and the assignment to `title`) to be exhaustive. So, if you remove the `else` branch, Xcode will immediately complain with an error like: "Constant 'title' used before being initialized."

```
59      let title: String
60      if difference == 0 {
61        title = "Perfect!"
62      } else if difference < 5 {
63        title = "You almost had it!"
64      } else if difference < 10 {
65        title = "Pretty good!"
66  //    } else {
67  //      title = "Not even close..."
68      }
69
70      let message = "You scored \(points) points"
71
72      let alert = UIAlertController(title: title,        Constant 'title' used before being initialized
73                                    message: message,
74                                    preferredStyle: .alert)
```

A constant needs to be initialized exhaustively

Run the app and play the game for a bit. You'll see that the title text changes depending on how well you're doing. That `if` statement sure is handy!



The alert with the new title

iOS App 1: Part 5

Bonus points

Exercise: Give players an additional 100 bonus points when they get a perfect score. This will encourage players to really try to place the bull's eye right on the target. Otherwise, there isn't much difference between 100 points for a perfect score and 98 or 95 points if you're close but not quite there.

Now there is an incentive for trying harder — a perfect score is no longer worth just 100 but 200 points! Maybe you can also give the player 50 bonus points for being just one off.

➤ Here is how I would have made these changes:

```
@IBAction func showAlert() {
  let difference = abs(targetValue - currentValue)
  var points = 100 - difference     // change let to var

  let title: String
  if difference == 0 {
    title = "Perfect!"
    points += 100                   // add this line
  } else if difference < 5 {
    title = "You almost had it!"
    if difference == 1 {            // add these lines
      points += 50                  // add these lines
    }                               // add these lines
  } else if difference < 10 {
    title = "Pretty good!"
  } else {
    title = "Not even close..."
  }
  score += points                   // move this line here from the top
  . . .
}
```

You should notice a few things:

- In the first `if` you'll see a new statement between the curly brackets. When the difference is equal to zero, you now not only set `title` to "Perfect!" but also award an extra 100 points.
- The second `if` has changed, too. There is now an `if` inside another `if`. Nothing wrong with that! You want to handle the case where `difference` is 1 in order to give the player bonus points. That happens inside the new `if` statement.

  After all, if the difference is more than 0 but less than 5, it could be 1 (but not necessarily all the time). Therefore, you perform an additional check to see if the difference truly is 1, and if so, add 50 extra points.

iOS App 1: Part 5

- Because these new `if` statements add extra points, `points` can no longer be a constant; it now needs to be a variable. That's why you change it from `let` to `var`.
- Finally, the line `score += points` has moved below the `if`s. This is necessary because the app updates the `points` variable inside those `if` statements (if the conditions are right) and you want those additional points to count towards the final score.

If your code is slightly different, then that's fine too, as long as it works! There is often more than one way to program something, and if the results are the same, then any approach is equally valid.

➤ Run the app to see if you can score some bonus points!

Raking in the points...

Local variables recap

I would like to point out once more the difference between local variables and instance variables. As you should know by now, a local variable only exists for the duration of the method that it is defined in, while an instance variable exists as long as the view controller (or any object that owns it) exists. The same thing is true for constants.

In `showAlert()`, there are six locals and you use three instance variables:

```
let difference = abs(targetValue - currentValue)
var points = 100 - difference
let title = . . .
score += points
let message = . . .
let alert = . . .
let action = . . .
```

Exercise: Point out which are the locals and which are the instance variables in the `showAlert()` method. Of the locals, which are variables and which are constants?

Locals are easy to recognize, because the first time they are used inside a method their name is preceded with `let` or `var`:

```
let difference = . . .
var points = . . .
let title = . . .
let message = . . .
let alert = . . .
let action = . . .
```

This syntax creates a new variable (`var`) or constant (`let`). Because these variables and constants are created inside the method, they are locals.

Those six items — `difference`, `points`, `title`, `message`, `alert`, and `action` — are restricted to the `showAlert()` method and do not exist outside of it. As soon as the method is done, the locals cease to exist.

You may be wondering how `difference`, for example, can have a different value every time the player taps the Hit Me! button, even though it is a constant — after all, aren't constants given a value just once, never to change afterwards?

Here's why: each time a method is invoked, its local variables and constants are created anew. The old values have long been discarded and you get brand new ones.

When `showAlert()` is called a second time, it creates a completely new instance of `difference` that is unrelated to the previous one. That particular constant value is only used until the end of `showAlert()` and then it is discarded.

The next time `showAlert()` is called after that, it creates yet another new instance of `difference` (as well as new instances of the other locals `points`, `title`, `message`, `alert`, and `action`). And so on... There's some serious recycling going on here!

But inside a single invocation of `showAlert()`, `difference` can never change once it has a value assigned. The only local in `showAlert()` that can change is `points`, because it's a `var`.

The instance variables, on the other hand, are defined outside of any method. It is common to put them at the top of the file:

```
class ViewController: UIViewController {
  var currentValue = 0
  var targetValue = 0
  var score = 0
  var round = 0
```

As a result, you can use these variables inside any method, without the need to declare them again, and they will keep their values till the object holding them (the view controller in this case) ceases to exist.

If you were to do this:

```
@IBAction func showAlert() {
  let difference = abs(targetValue - currentValue)
  var points = 100 - difference

  var score = score + points       // doesn't work!
  . . .
}
```

Then things wouldn't work as you'd expect them to. Because you now put `var` in front of `score`, you have made it a new local variable that is only valid inside this method.

In other words, this won't add `points` to the *instance variable* `score` but to a new *local variable* that also happens to be named `score`. In this case, the instance variable `score` never changes, even though it has the same name.

Obviously that is not what you want to happen here. Fortunately, the above won't even compile. Swift knows there's something fishy about that line.

> **Note**: To make a distinction between the two types of variables, so that it's always clear at a glance how long they will live, some programmers prefix the names of instance variables with an underscore.
>
> They would name the variable `_score` instead of just `score`. Now there is less confusion because names beginning with an underscore won't be mistaken for being locals. This is only a convention. Swift doesn't care one way or the other how you name your instance variables.
>
> Other programmers use different prefixes, such as "m" (for member) or "f" (for field) for the same purpose. Some even put the underscore *behind* the variable name. Madness!

The alert

There is something that bothers me about the game. You may have noticed it too...

As soon as you tap the Hit Me! button and the alert pops up, the slider immediately jumps back to its center position, the round number increments, and the target label already gets the new random number.

What happens is that the new round has already begun while you're still watching the results of the last round. That's a little confusing (and annoying).

It would be better to wait on starting the new round until *after* the player has dismissed the alert pop-up. Only then is the current round truly over.

Asynchronous code execution

Maybe you're wondering why this isn't already happening? After all, in `showAlert()` you only call `startNewRound()` after you've shown the alert pop-up:

```
@IBAction func showAlert() {
  . . .
  let alert = UIAlertController(. . .)
  let action = UIAlertAction(. . .)
  alert.addAction(action)

  // Here you make the alert visible:
  present(alert, animated: true, completion: nil)

  // Here you start the new round:
  startNewRound()
}
```

Contrary to what you might expect, `present(alert:animated:completion:)` doesn't hold up execution of the rest of the method until the alert pop-up is dismissed. That's how alerts on other platforms tend to work, but not on iOS.

Instead, `present(alert:animated:completion:)` puts the alert on the screen and immediately returns control to the next line of code in the method. The rest of the `showAlert()` method is executed right away, and the new round starts before the alert pop-up has even finished animating.

In programmer-speak, alerts work *asynchronously*. We'll talk much more about that in a later chapter, but what it means for you right now is that you don't know in advance when the alert will be done. But you can bet it will be well after `showAlert()` has finished.

Alert event handling

So, if your code execution can't wait in `showAlert()` until the pop-up is dismissed, then how do you wait for it to close?

The answer is simple: events! As you've seen, a lot of the programming for iOS involves waiting for specific events to occur — buttons being tapped, sliders being moved, and so on. This is no different. You have to wait for the "alert dismissed" event somehow. In the mean time, you simply do nothing.

Here's how it works:

For each button on the alert, you have to supply a `UIAlertAction` object. This object tells the alert what the text on the button is — "OK" — and what the button looks like (you're using the default style, here):

```
let action = UIAlertAction(title: "OK", style: .default, handler: nil)
```

The third parameter, `handler`, tells the alert what should happen when the button is pressed. This is the "alert dismissed" event you've been looking for! Currently `handler` is `nil`, which means nothing happens. In case you're wondering, a `nil` in Swift indicates "no value." You will learn more about `nil` values later on.

You can however, give the `UIAlertAction` some code to execute when the OK button is tapped. When the user finally taps OK, the alert will remove itself from the screen and jump to your code. That's your cue to start a new round. This is also known as the *callback* pattern. There are several ways this pattern manifests on iOS. Often you'll be asked to create a new method to handle the event. But here you'll use something new: a *closure*.

➤ Change the bottom bit of `showAlert()` to:

```
@IBAction func showAlert() {
  . . .
  let alert = UIAlertController(. . .)

  let action = UIAlertAction(title: "OK", style: .default,
                             handler: { _ in
                                        self.startNewRound()
                                      })

  alert.addAction(action)
  present(alert, animated: true, completion: nil)
}
```

Two things have happened here:

1. You removed the call to `startNewRound()` from the bottom of the method. (Don't forget this part!)
2. You placed it inside a block of code that you gave to `UIAlertAction`'s `handler` parameter.

   Such a block of code is called a *closure*. You can think of it as a method without a name. This code is not performed right away. Rather, it's performed only when the OK button is tapped. This particular closure tells the app to start a new round (and update the labels) when the alert is dismissed.

➤ Run the app and see for yourself. I think the game feels a lot better this way.

Self

You may be wondering why in the handler block you did `self.startNewRound()` instead of just writing `startNewRound()` like before.

The `self` keyword allows the view controller to refer to itself. That shouldn't be too strange a concept. When you say, "I want ice cream," you use the word "I" to refer to yourself. Similarly, objects can talk about (or to) themselves as well.

Normally you don't need to use `self` to send messages to the view controller, even though it is allowed. The exception: inside closures you *do* have to use `self` to refer to the view controller.

This is a rule in Swift. If you forget `self` in a closure, Xcode will complain (try it out). This rule exists because closures can "capture" variables, which comes with surprising side effects. You'll learn more about that in later chapters.

Start over

No, you're not going to throw away the source code and start this project all over! I'm talking about the game's "Start Over" button. This button is supposed to reset the score and start over from the first round.

One use of the Start Over button is for playing against another person. The first player does ten rounds, then the score is reset and the second player does ten rounds. The player with the highest score wins.

Exercise: Try to implement the Start Over button on your own. You've already seen how you can make the view controller react to button presses, and you should be able to figure out how to change the `score` and `round` variables.

How did you do? If you got stuck, then follow the instructions below.

iOS App 1: Part 5

The new method

First, add a method to **ViewController.swift** that starts a new game. I suggest you put it near `startNewRound()` because the two are conceptually related.

➤ Add the new method:

```
func startNewGame() {
  score = 0
  round = 0
  startNewRound()
}
```

This method resets `score` and `round` to zero, and starts a new round as well.

Notice that you set `round` to 0 here, not to 1. You use 0 because incrementing the value of `round` is the first thing that `startNewRound()` does. If you were to set `round` to 1, then `startNewRound()` would add another 1 to it and the first round would actually be labeled round 2.

So, you begin at 0, let `startNewRound()` add one and everything works great.

(It's probably easier to figure this out from the code than from my explanation. This should illustrate why we don't program computers in English.)

You also need an action method to handle taps on the Start Over button. You could write a new method like the following:

```
@IBAction func startOver() {
  startNewGame()
}
```

But you'll notice that this method simply calls the previous method that you added. So, why not cut out the middleman? You can simply change the method you added previously to be an action instead, like this:

```
@IBAction func startNewGame() {
  score = 0
  round = 0
  startNewRound()
}
```

You could follow either of the above approaches since both are equally valid. Personally, I like to have less code since that means there's less stuff to maintain (and less of a chance of screwing something up). Sometimes, there could also be legitimate reasons for having a seperate action method which calls your own method, but in this particular case, it's better to keep things simple.

iOS App 1: Part 5

Just to keep things consistent, in `viewDidLoad()` you should replace the call to `startNewRound()` with `startNewGame()`.

Because `score` and `round` are already 0 when the app starts, it won't really make any difference to how the app works, but it does make the intention of the source code clearer.

If you wonder whether you can call an `IBAction` method directly instead of hooking it up to an action in the storyboard, yes, you certainly can do so.

➤ Make this change:

```
override func viewDidLoad() {
  super.viewDidLoad()
  startNewGame()        // this line changed
}
```

Connect the outlet

Finally, you need to connect the Start Over button to the action method.

➤ Open the storyboard and Control-drag from the **Start Over** button to View controller. Let go of the mouse button and pick **startNewGame** from the pop-up if you opted to have `startNewGame()` as the action method. Otherwise, pick the name of your action method .

That connects the button's Touch Up Inside event to the action you have just defined.

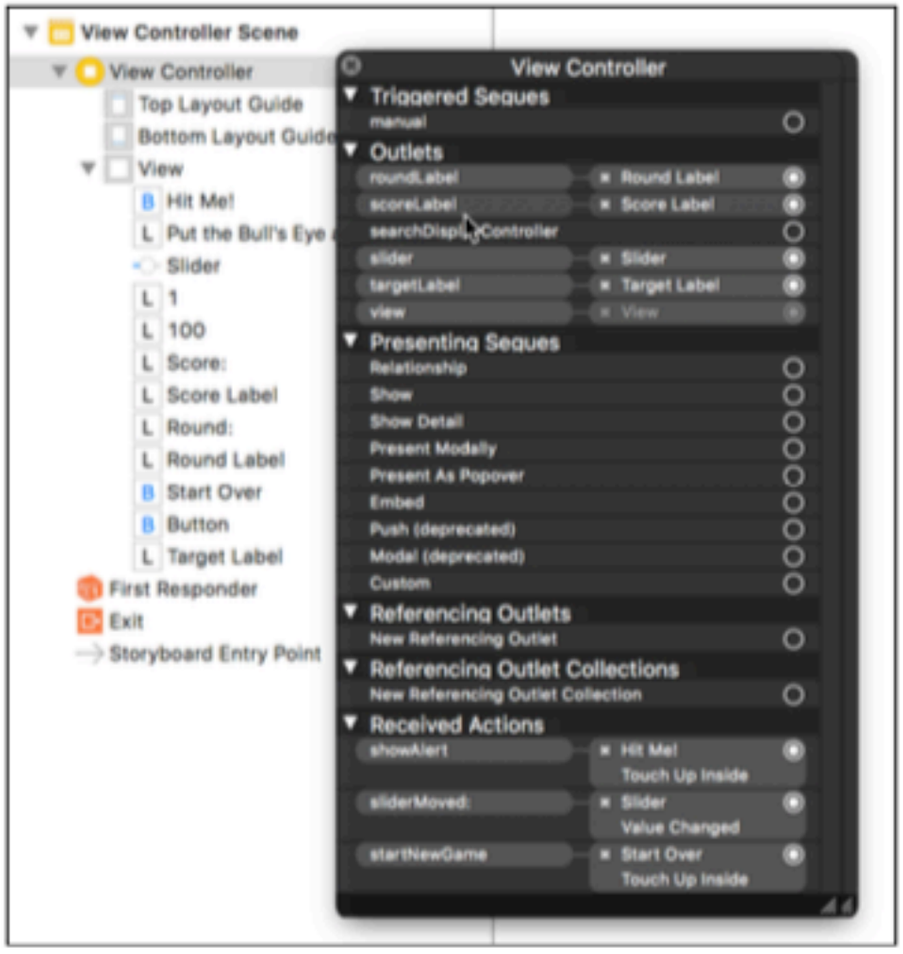
➤ Run the app and play a few rounds. Press Start Over and the game puts you back at square one.

Tip: If you're losing track of what button or label is connected to what method, you can click on View controller in the storyboard to see all the connections that you have made so far.

You can either right-click on View controller to get a pop-up, or simply view the connections in the **Connections inspector**.

iOS App 1: Part 5

This shows all the connections for the view controller.



All the connections from View Controller to the other objects

Now your game is pretty polished and your task list has gotten really short.

iOS App 1: Part 6

Bull's Eye is looking good, the gameplay elements are done, and there's one item left in your to-do list — "Make it look pretty."

You have to admit the game still doesn't look great. If you were to put this on the App Store in its current form, I'm not sure many people would be excited to download it. Fortunately, iOS makes it easy for you to create good-looking apps, so let's give *Bull's Eye* a makeover and add some visual flair.

This chapter covers the following:

- **Landscape orientation revisited**: Project changes to make landscape orientation support work better.
- **Spice up the graphics**: Replace the app UI with custom graphics to give it a more polished look.
- **The about screen**: Add an about screen to the app and make it look spiffy.

Landscape orientation revisited

First, let's quickly revisit another item in the to-do list — "Put the app in landscape orientation." You already did this, right? But there's a little bit of clean up to be done with regards to that item.

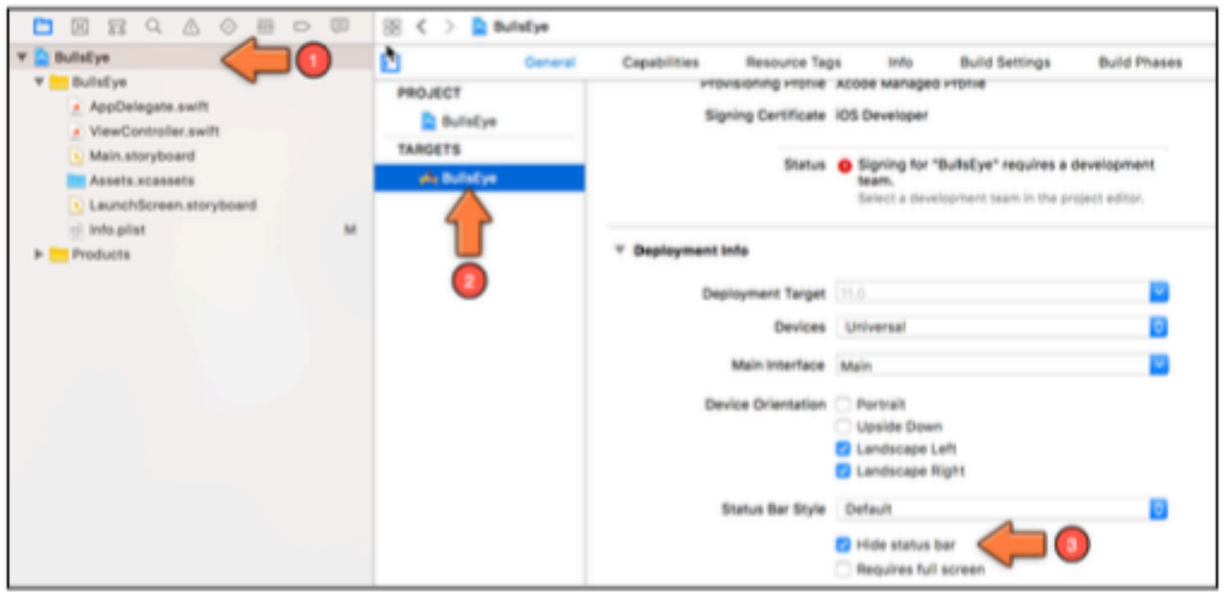
Apps in landscape mode do not display the iPhone status bar, unless you tell them to. That's great for your app — games require a more immersive experience and the status bar detracts from that.

iOS App 1: Part 6

Even though the system automatically handles hiding the status bar for your game, there is still one thing you can do to improve the way *Bull's Eye* handles the status bar.

➤ Go to the **Project Settings** screen and scroll down to **Deployment Info**. Under **Status Bar Style**, check **Hide status bar**.

This will ensure that the status bar is hidden during application launch.



Hiding the status bar when the app launches

It's a good idea to hide the status bar while the app is launching. It takes a few seconds for the operating system to load the app into memory and start it up, and during that time the status bar remains visible, unless you hide it using this option.

It's only a small detail, but the difference between a mediocre app and a great app is that great apps get all the small details right.

➤ That's it. Run the app and you'll see that the status bar is history.

Info.plist

Most of the options from the Project Settings screen, such as the supported device orientations and whether the status bar is visible during launch, get stored in your app's Info.plist file.

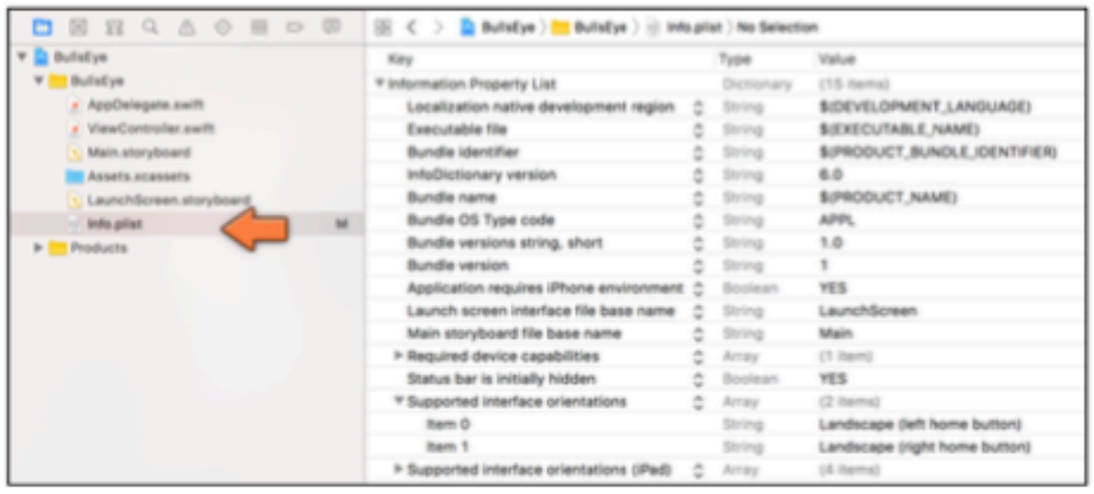
Info.plist is a configuration file inside the application bundle that tells iOS how the app will behave. It also describes certain characteristics of the app, such as the version number, that don't really fit anywhere else.

iOS App 1: Part 6

With some earlier versions of Xcode, you often had to edit Info.plist by hand, but with the latest Xcode versions this is hardly necessary anymore. You can make most of the changes directly from the Project Settings screen.

However, it's good to know that Info.plist exists and what it looks like.

➤ Go to the **Project navigator** and select the file named **Info.plist** to take a peek at its contents.



The Info.plist file is just a list of configuration options and their values. Most of these may not make sense to you, but that's OK – they don't always make sense to me either.

Notice the option **Status bar is initially hidden**. It has the value YES. This is the option that you just changed.

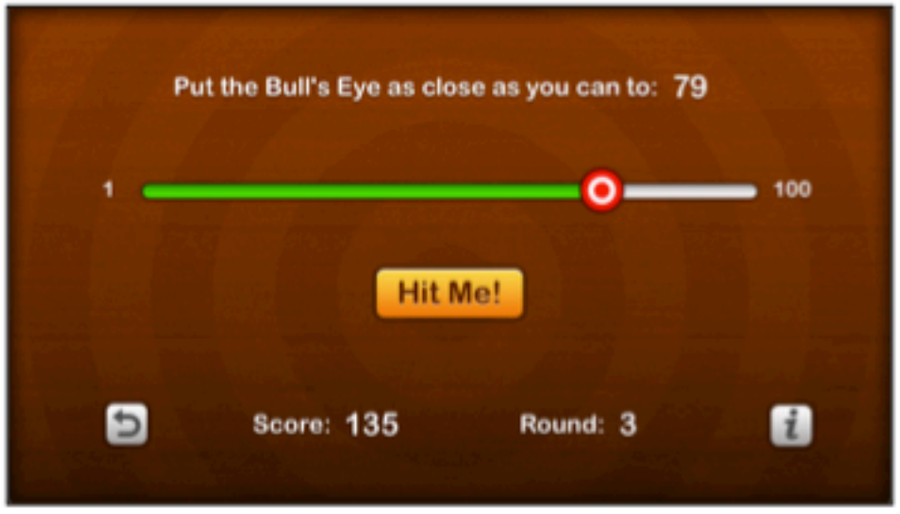
Spicing up the graphics

Getting rid of the status bar is only the first step. We want to go from this:

Yawn...

iOS App 1: Part 6

To something that's more like this:



Cool :-)

The actual controls won't change. You'll simply be using images to smarten up their look, and you will also adjust the colors and typefaces.

You can put an image in the background, on the buttons, and even on the slider, to customize the appearance of each. The images you use should generally be in PNG format, though JPG files would work too.

Adding the image assets

If you are artistically challenged, then don't worry, I have provided a set of images for you. But if you do have mad Photoshop skillz, then by all means feel free to design (and use) your own images.

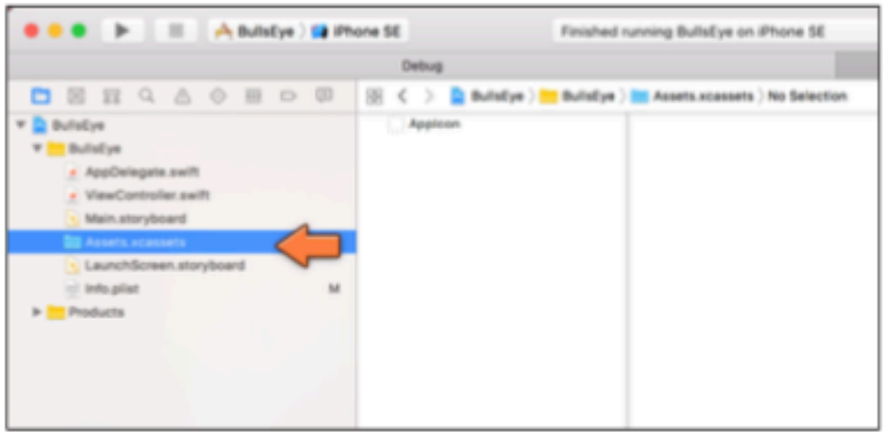
The Resources folder that comes with this book contains a subfolder named Images. You will first import these images into the Xcode project.

➤ In the **Project navigator**, find **Assets.xcassets** and click on it.

This item is known as the asset catalog for the app and it contains all the app's images.

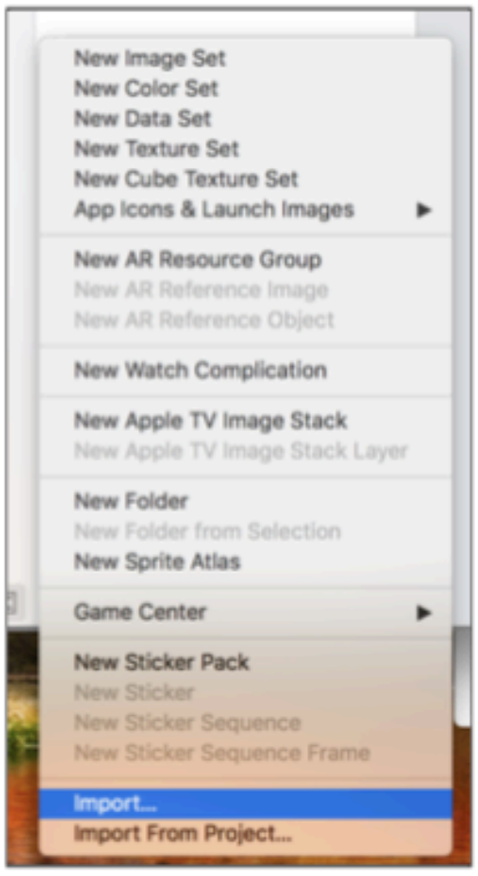
iOS App 1: Part 6

Right now, it is empty and contains just a placeholder for the app icon, which you'll add soon.



The asset catalog is initially empty

➤ At the bottom of the secondary pane, the one with AppIcon, there is a + button. Click it and then select the **Import...** option:



Choose Import to put existing images into the asset catalog

iOS App 1: Part 6

Xcode shows a file picker. Select the **Images** folder from the resources and press **⌘+A** to select all the files inside this folder.

Choosing the images to import

Click **Open** and Xcode copies all the image files from that folder into the asset catalog:

The images are now inside the asset catalog

If Xcode added a folder named "Images" instead of the individual image files, then try again and this time make sure that you select the files inside the Images folder rather than the folder itself before you click Open.

> **Note**: Instead of using the **Import...** menu option as above, you could also simply drag the necessary files from Finder on to the Xcode asset catalog view. As ever, there's more than one way to do the same thing in Xcode.

1x, 2x, and 3x displays

Each image set in the asset catalog has a slot for a "2x" image, but you can also specify 1x and 3x images. Having multiple versions of the same image in varying sizes allows your apps to support the wide variety of iPhone and iPad displays in existence.

iOS App 1: Part 6

1x is for low-resolution screens, the ones with the big, chunky pixels. There are no low-resolution devices in existence that can actually run iOS 12 – they are too old to bother with – so you're not likely to come across many 1x images anymore. 1x is only a concern if you're working on an app that still needs to support iOS 9 or older.

2x is for high-resolution Retina screens. This covers most modern iPhones, iPod touches, and iPads. Retina images are twice as big as the low-res images, hence the 2x. The images you imported just now are 2x images.

3x is for the super high-resolution Retina HD screen of the iPhone Plus devices. If you want your app to have extra sharp images on these top-of-the-line iPhone models, then you can drop them into the "3x" slot in the asset catalog.

There is a special naming convention for image files. If the filename ends in **@2x** or **@3x** then that's considered the Retina or Retina HD version. Low-resolution 1x images have no special name (you don't have to write @1x).

★ ★ ★

Putting up the wallpaper

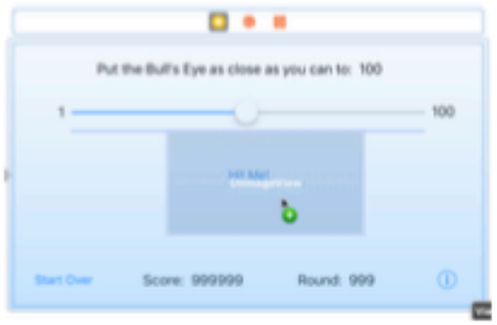
Let's begin by changing the drab white background in *Bull's Eye* to something more fancy.

➤ Open **Main.storyboard**, open the **Library** panel (via the top toolbar) and locate an **Image View**. (Tip: if you type "image" into the search box at the top of the Library panel, it will quickly filter out all the other views.)

The Image View control in the Objects Library

iOS App 1: Part 6

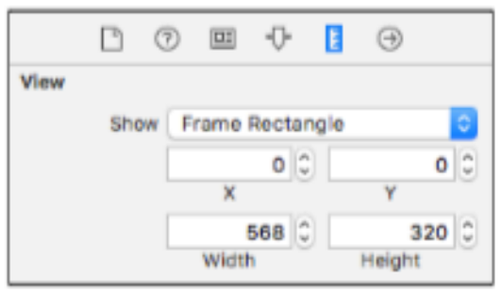
➤ Drag the image view on top of the existing user interface. It doesn't really matter where you put it, as long as it's inside the Bull's Eye view controller.



Dragging the Image View into the view controller

➤ With the image view still selected, go to the **Size inspector** (that's the one next to the Attributes inspector) and set X and Y to 0, Width to 568 and Height to 320.

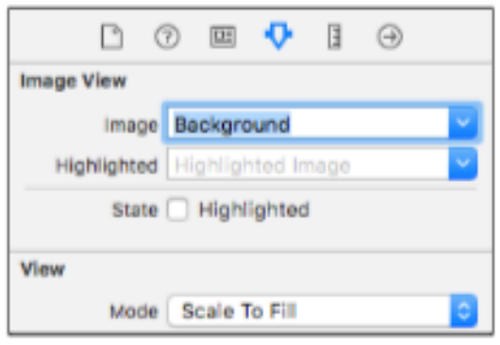
This will make the image view cover the entire screen.



The Size inspector settings for the Image View

➤ Go to the **Attributes inspector** for the image view. At the top there is an option named **Image**. Click the downward arrow and choose **Background** from the list.

This will put the image named "Background" from the asset catalog into the image view.



Setting the background image on the Image View

iOS App 1: Part 6

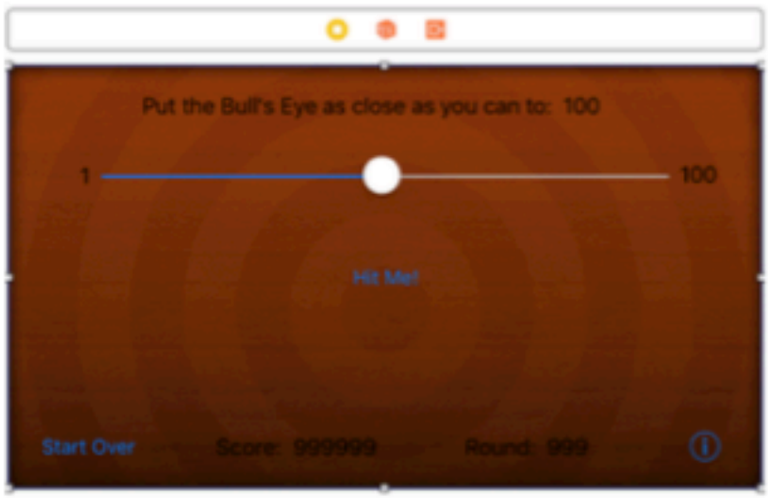
There is only one problem: the image now covers all the other controls. There is an easy fix for that; you have to move the image view behind the other views.

➤ With the image view selected, in the **Editor** menu in Xcode's menu bar at the top of the screen, choose **Arrange ▸ Send to Back**.

Sometimes Xcode gives you a hard time with this (it still has a few bugs) and you might not see the Send to Back item enabled. If so, try de-selecting the Image View and then selecting it again. Now the menu item should be available.

Alternatively, pick up the image view in the Document Outline and drag it to the top of the list of views, just below Safe Area, to accomplish the same thing. The items in the Document Outline view are listed so that the backmost item is at the top of the list and the frontmost one is at the bottom.

Your interface should now look something like this:



The game with the new background image

That takes care of the background. Run the app and marvel at the new graphics.

Changing the labels

Because the background image is quite dark, the black text labels have become hard to read. Fortunately, Interface Builder lets you change label color. While you're at it, you might change the font as well.

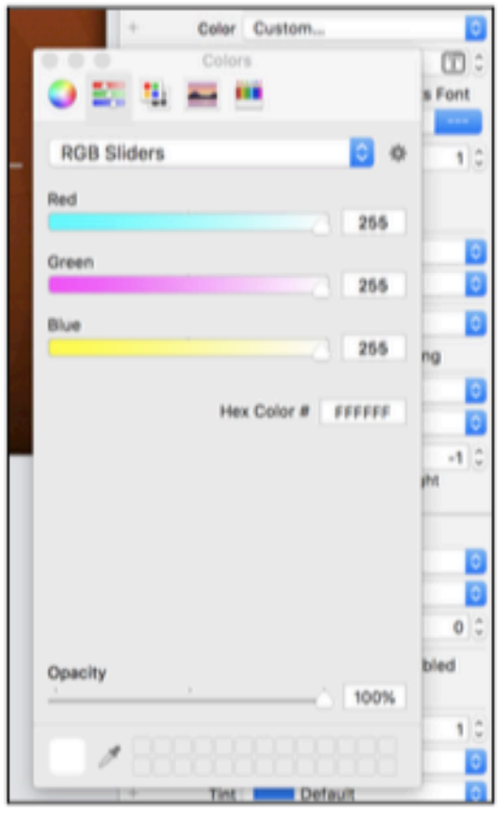
iOS App 1: Part 6

➤ Still in the storyboard, select the label at the top, open the **Attributes inspector** and click on the **Color** item to show a dropdown for color values. Select **Custom...** at the bottom of the list.

Setting the text color on the label

This opens the Color Picker, which has several ways to select colors. I prefer the sliders (second tab).

If all you see is a gray scale slider, then select **RGB Sliders** from the picker at the top.



The Color Picker

iOS App 1: Part 6

➤ Pick a pure white color, Red: 255, Green: 255, Blue: 255, Opacity: 100%. Alternatively, you can simply pick **White Color** from the initial dropdown instead of opening the Color Picker at all, but it's good to know that the Color Picker is there in case you want to do custom colors.

➤ Click on the **Shadow** item from the Attributes inspector. This lets you add a subtle shadow to the label. By default this color is transparent (also known as "Clear Color") so you won't see the shadow. Using the Color Picker, choose a pure black color that is half transparent, Red: 0, Green: 0, Blue: 0, Opacity: 50%.

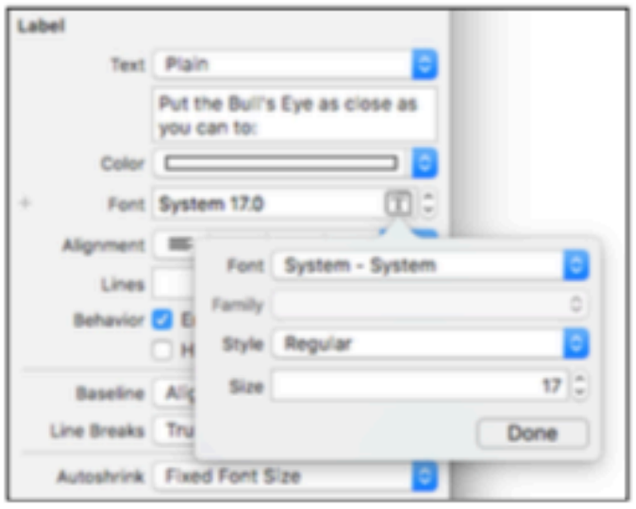
Note: Sometimes when you change the Color or Shadow attributes, the background color of the view also changes. This is a bug in Xcode. Put it back to Clear Color if that happens.

➤ Change the **Shadow Offset** to Width: 0, Height: 1. This puts the shadow below the label.

The shadow you've chosen is very subtle. If you're not sure that it's actually visible, then toggle the height offset between 1 and 0 a few times. Look closely and you should be able to see the difference. As I said, it's very subtle.

➤ Click on the **[T]** icon of the **Font** attribute. This opens the Font Picker.

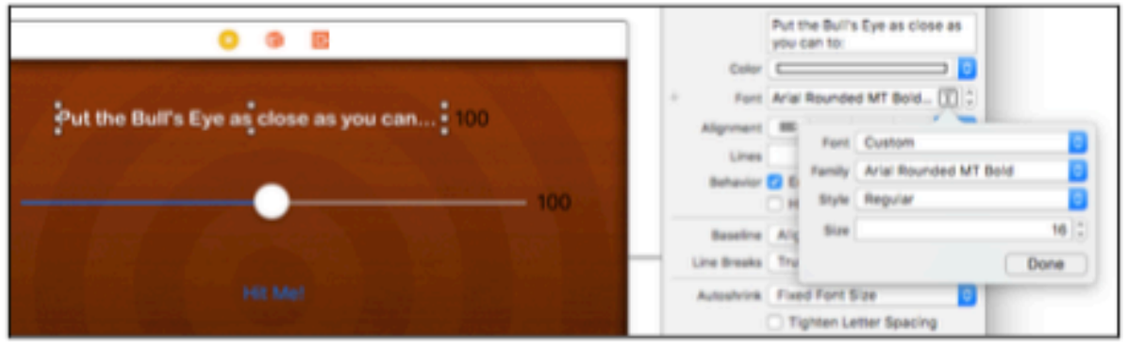
By default, the System font is selected. That uses whatever is the standard system font for the user's device. The system font is nice enough but we want something more exciting for this game.



Font picker with the System font

iOS App 1: Part 6

➤ Choose **Font: Custom**. That enables the Family field. Choose **Family: Arial Rounded MT Bold**. Set the Size to 16.



Setting the label's font

➤ The label also has an attribute **Autoshrink**. Make sure this is set to **Fixed Font Size**.

If enabled, Autoshrink will dynamically change the size of the font if the text is larger than will fit into the label. That is useful in certain apps, but not in this one. Instead, you'll change the size of the label to fit the text rather than the other way around.

➤ With the label selected, press ⌘= on your keyboard, or choose **Size to Fit Content** from the **Editor** menu.

If the Size to Fit Content menu item is disabled, then de-select the label and select it again. Sometimes Xcode gets confused about what is selected. Poor thing.

The label will now become slightly larger or smaller so that it fits snugly around the text. If the text got cut off when you changed the font, now all the text will show again.

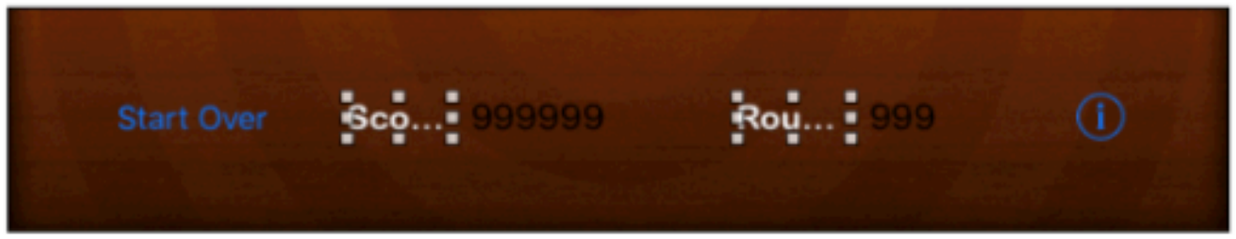
You don't have to set these properties for the other labels one by one; that would be a big chore. You can speed up the process by selecting multiple labels and then applying these changes to that entire selection.

➤ Click on the **Score**: label to select it. Hold ⌘ and click on the **Round**: label. Now both labels will be selected. Repeat what you did above for these labels:

- Set Color to pure white, 100% opaque.
- Set Shadow to pure black, 50% opaque.
- Set Shadow Offset to width 0, height 1.
- Set Font to Arial Rounded MT Bold, size 16.
- Make sure Autoshrink is set to Fixed Font Size.

iOS App 1: Part 6

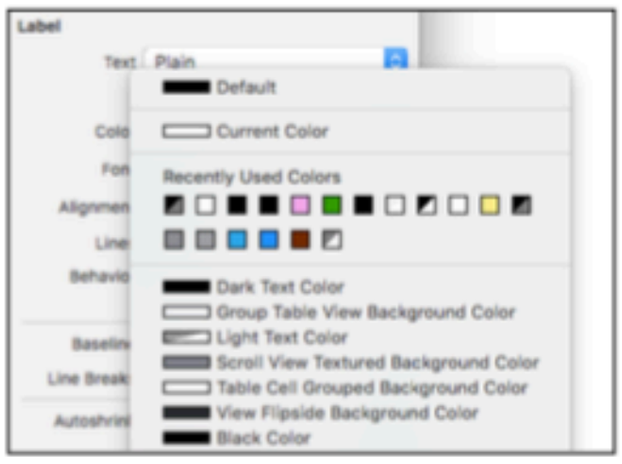
As you can see, in my storyboard the text no longer fits into the Score and Round labels:



The font is too large to fit all the text in the Score and Round labels

You can either make the labels larger by dragging their handles to resize them manually, or you can use the **Size to Fit Content** option (⌘=). I prefer the latter because it's less work.

Tip: Xcode is smart enough to remember the colors you have used recently. Instead of going into the Color Picker all the time, you can simply choose a color from the Recently Used Colors menu which is part of the dropdown you get when you click on any color option:



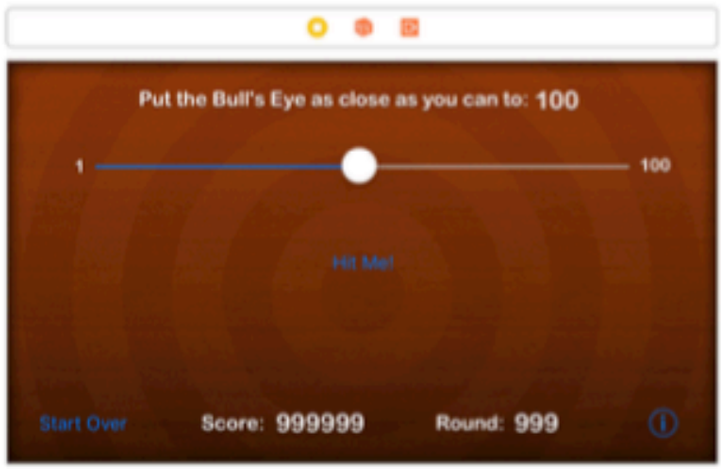
Quick access to recently used colors and several handy presets

Exercise: You still have a few labels to go. Repeat what you just did for the other labels. They should all become white, have the same shadow and have the same font. However, the two labels on either side of the slider (1 and 100) will have font size 14, while the other labels (the ones that will hold the target value, the score and the round number) will have font size 20 so they stand out more.

Because you've changed the sizes of some of the labels, your carefully constructed layout may have been messed up a bit. You may want to clean it up a little.

iOS App 1: Part 6

At this point, the game screen should look something like this:



What the storyboard looks like after styling the labels

All right, it's starting to look like something now. By the way, feel free to experiment with the fonts and colors. If you want to make it look completely different, then go right ahead. It's your app!

The buttons

Changing the look of the buttons works very much the same way.

➤ Select the **Hit Me!** button. In the **Size inspector** set its Width to 100 and its Height to 37.

➤ Center the position of the button on the inner circle of the background image.

➤ Go to the **Attributes inspector**. Change **Type** from System to **Custom**.

A "system" button just has a label and no border. By making it a custom button, you can style it any way you wish.

➤ Still in the **Attributes inspector**, press the arrow on the **Background** field and choose **Button-Normal** from the list.

➤ Set the **Font** to **Arial Rounded MT Bold**, size 20.

➤ Set the **Text Color** to red: 96, green: 30, blue: 0, opacity: 100%. This is a dark brown color.

➤ Set the **Shadow Color** to pure white, 50% opacity. The shadow offset should be Width 0, Height 1.

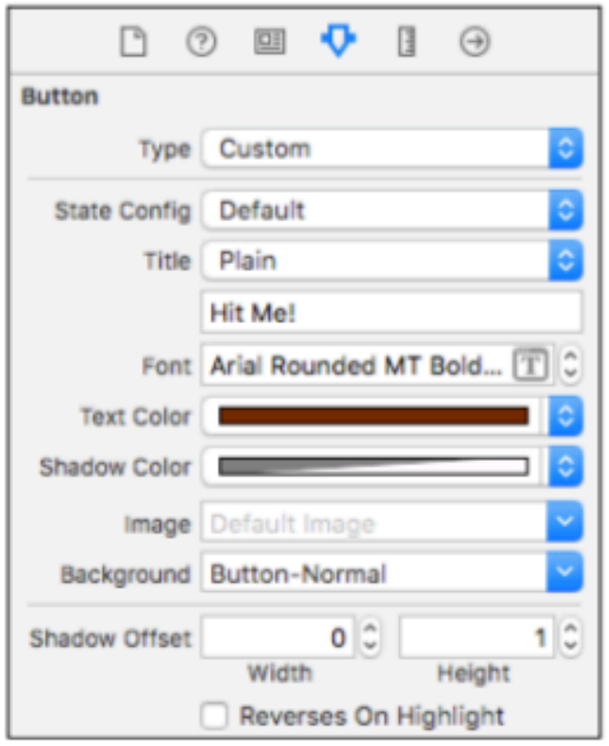
iOS App 1: Part 6

Blending in

Setting the opacity to anything less than 100% will make the color slightly transparent (with opacity of 0% being fully transparent). Partial transparency makes the color blend in with the background and makes it appear softer.

Try setting the shadow color to 100% opaque pure white and notice the difference.

This finishes the setup for the Hit Me! button in its "default" state:



The attributes for the Hit Me! button in the default state

Buttons can have more than one state. When you tap a button and hold it down, it should appear "pressed down" to let you know that the button will be activated when you lift your finger. This is known as the *highlighted* state and is an important visual cue to the user.

➤ With the button still selected, click the **State Config** setting and pick **Highlighted** from the menu. Now the attributes in this section reflect the highlighted state of the button.

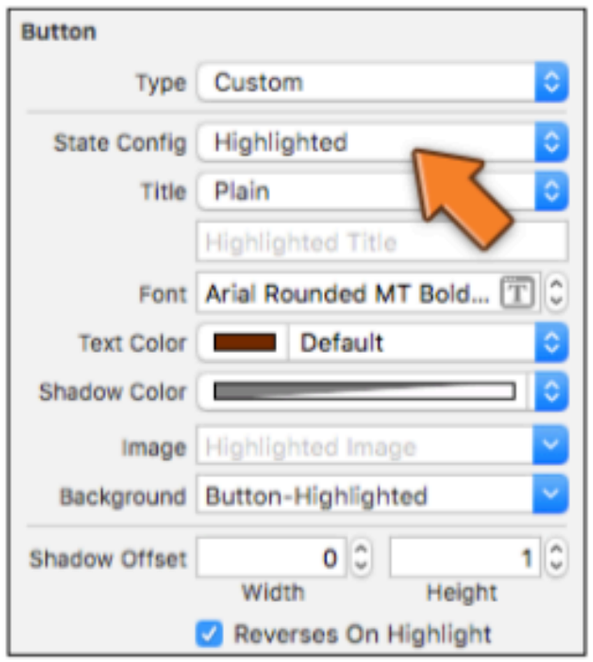
➤ In the **Background** field, select **Button-Highlighted**.

iOS App 1: Part 6

➤ Make sure the highlighted **Text Color** is the same color as before (red 96, green 30, blue 0, or simply pick it from the Recently Used Colors menu). Change the **Shadow Color** to half-transparent white again.

➤ Check the **Reverses On Highlight** option. This will give the appearance of the label being pressed down when the user taps the button.

You could change the other properties too, but don't get too carried away. The highlight effect should not be too jarring.



The attributes for the highlighted Hit Me! button

To test the highlighted look of the button in Interface Builder you can toggle the **Highlighted** box in the **Control** section, but make sure to turn it off again or the button will initially appear highlighted when the screen is shown.

That's it for the Hit Me! button. Styling the Start Over button is very similar, except you will replace its title text with an icon.

➤ Select the **Start Over** button and change the following attributes:

- Set Type to Custom.
- Remove the text "Start Over" from the button.
- For Image choose **StartOverIcon**.
- For Background choose **SmallButton**.

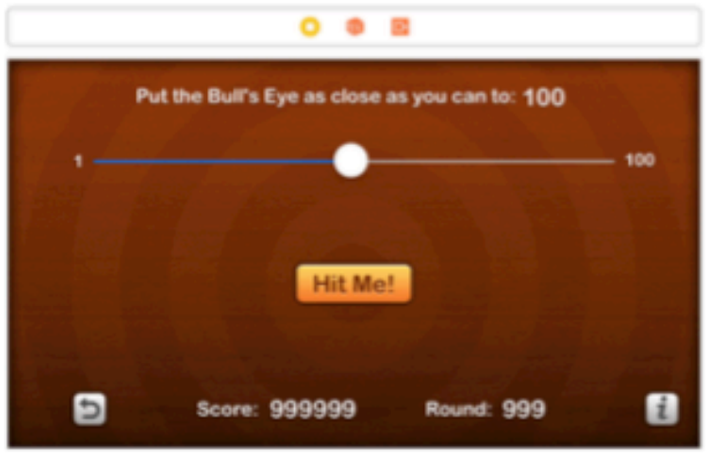
iOS App 1: Part 6

- Set Width and Height to 32.

You won't set a highlighted state on this button — let UIKit take care of this. If you don't specify a different image for the highlighted state, UIKit will automatically darken the button to indicate that it is pressed.

➤ Make the same changes to the ⓘ button, but this time choose **InfoButton** for the image.

The user interface is almost done. Only the slider is left...



Almost done!

The slider

Unfortunately, you can only customize the slider a little bit in Interface Builder. For the more advanced customization that this game needs – putting your own images on the thumb and the track – you have to resort to writing code.

Do note that everything you've done so far in Interface Builder you could also have done in code. Setting the color on a button, for example, can be done by sending the `setTitleColor()` message to the button. (You would normally do this in `viewDidLoad`.)

However, I find that doing visual design work is much easier and quicker in a visual editor such as Interface Builder than writing the equivalent source code. But for the slider you have no choice.

➤ Go to **ViewController.swift**, and add the following to `viewDidLoad()`:

```
let thumbImageNormal = UIImage(named: "SliderThumb-Normal")!
slider.setThumbImage(thumbImageNormal, for: .normal)

let thumbImageHighlighted = UIImage(named: "SliderThumb-Highlighted")!
```

iOS App 1: Part 6

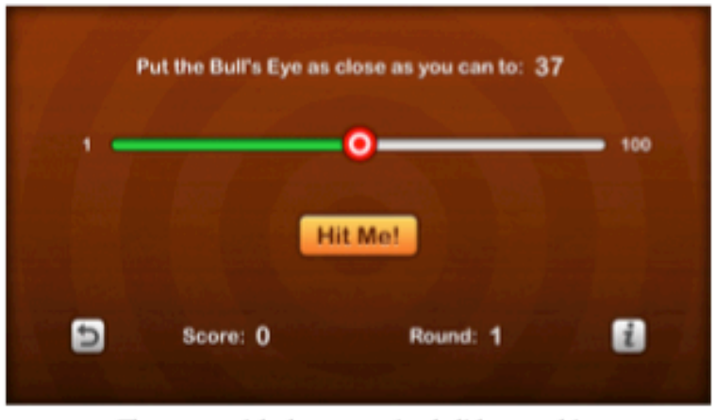
```
slider.setThumbImage(thumbImageHighlighted, for: .highlighted)

let insets = UIEdgeInsets(top: 0, left: 14, bottom: 0, right: 14)

let trackLeftImage = UIImage(named: "SliderTrackLeft")!
let trackLeftResizable =
                  trackLeftImage.resizableImage(withCapInsets: insets)
slider.setMinimumTrackImage(trackLeftResizable, for: .normal)

let trackRightImage = UIImage(named: "SliderTrackRight")!
let trackRightResizable =
                  trackRightImage.resizableImage(withCapInsets: insets)
slider.setMaximumTrackImage(trackRightResizable, for: .normal)
```

This sets four images on the slider: two for the thumb and two for the track. (And if you're wondering what the "thumb" is, that's the little circle in the center of the slider, the one that you drag around to set the slider value.)

The thumb works like a button so it gets an image for the normal (un-pressed) state and one for the highlighted state.

The slider uses different images for the track on the left of the thumb (green) and the track to the right of the thumb (gray).

➤ Run the app. You have to admit it looks fantastic now!



The game with the customized slider graphics

To .png or not to .png

If you recall, the images that you imported into the asset catalog had filenames like **SliderThumb-Normal@2x.png** and so on.

When you create a `UIImage` object, you don't use the original filename but the name that is listed in the asset catalog, **SliderThumb-Normal**.

That means you can leave off the **@2x** bit and the **.png** file extension.

iOS App 1: Part 6

The About screen

Your game looks awesome and your to-do list is done. So, does this mean that you are done with *Bull's Eye*?

Not so fast! Remember the ⓘ button on the game screen? Try tapping it. Does it do anything? No?

Ooops! Looks as if we forgot to add any functionality to that button! It's time to rectify that — let's add an "about" screen to the game which shows some information about the game and have it display when the user taps on the ⓘ button.

Initially, the screen will look something like this (but we'll prettify it soon enough):

The new About screen

This new screen contains a *text view* with the gameplay rules and a button to close the screen.

Most apps have more than one screen, even very simple games. So, this is as good a time as any to learn how to add additional screens to your apps.

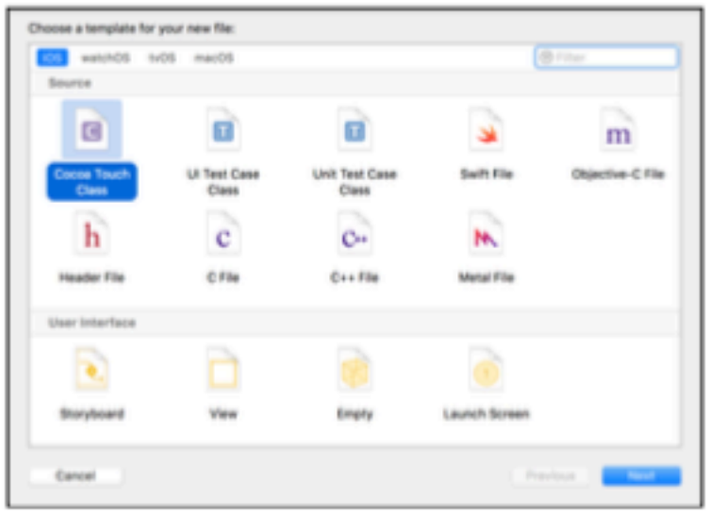
I have pointed it out a few times already: Each screen in your app will have its own view controller. If you think "screen," think "view controller."

Xcode automatically created the main `ViewController` object for you, but you'll have to create the view controller for the About screen yourself.

iOS App 1: Part 6

Adding a new view controller

➤ Go to Xcode's **File** menu and choose **New ▸ File...** In the window that pops up, choose the **Cocoa Touch Class** template (if you don't see it then make sure **iOS** is selected at the top).



Choosing the file template for Cocoa Touch Class

Click **Next**. Xcode gives you some options to fill out:

The options for the new file

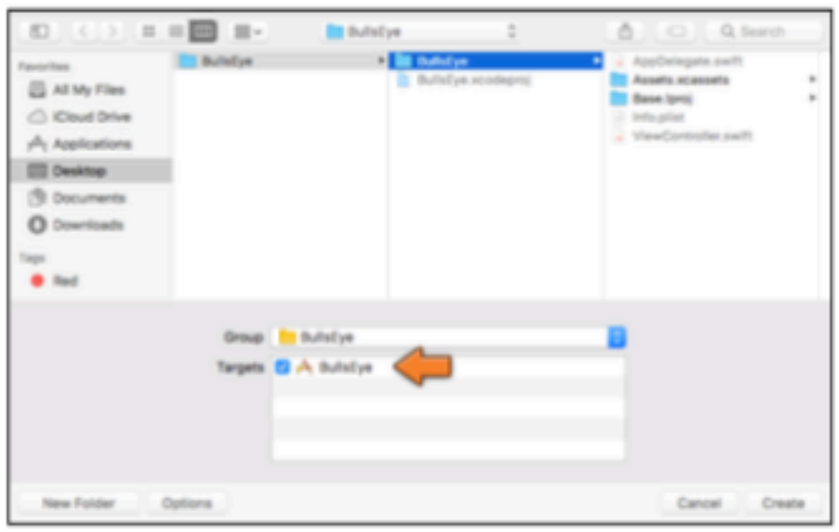
Choose the following:

- Class: **AboutViewController**.
- Subclass of: **UIViewController**.

iOS App 1: Part 6

- Also create XIB file: Leave this box unchecked.
- Language: **Swift**.

Click **Next**. Xcode will ask you where to save this new view controller.



Saving the new file

➤ Choose the **BullsEye** folder (this folder should already be selected).

Also make sure **Group** says **BullsEye** and that there is a checkmark in front of BullsEye in the list of **Targets**. (If you don't see this panel, click the Options button at the bottom of the dialog.)

➤ Click **Create**.

Xcode will create a new file and add it to your project. As you might have guessed, the new file is **AboutViewController.swift**.

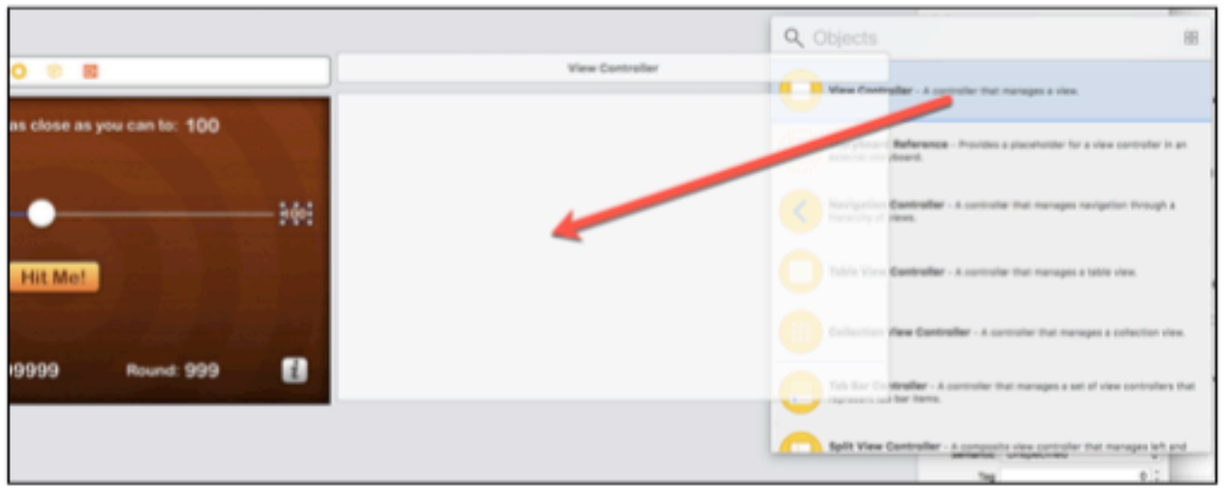
Designing the view controller in Interface Builder

To design this new view controller, you need to pay a visit to Interface Builder.

➤ Open **Main.storyboard**. There is no scene representing the About view controller in the storyboard yet. So, you'll have to add this first.

iOS App 1: Part 6

➤ From the **Library**, choose **View Controller** and drag it on to the canvas, to the right of the main View controller.



Dragging a new View Controller from the Objects Library

This new view controller is totally blank. You may need to rearrange the storyboard so that the two view controllers don't overlap. Interface Builder isn't very tidy about where it puts things.

➤ Drag a new **Button** on to the screen and give it the title **Close**. Put it somewhere around the bottom center of the view (use the blue guidelines to help with positioning).

➤ Drag a **Text View** on to the view and make it cover most of the space above the button.

You can find these components in the Library. If you don't feel like scrolling, you can filter the components by typing in the field at the top:

Searching for text components

Note that there is also a Text Field, which is a single-line text component — that's not what you want. You're looking for Text View, which can contain multiple lines of text.

iOS App 1: Part 6

After dragging both the text view and the button on to the canvas, it should look something like this:

The About screen in the storyboard

➤ Double-click the text view to make its content is editable. By default, the Text View contains a bunch of Latin placeholder text (also known as "Lorem Ipsum").

Enter this new text into the Text View:

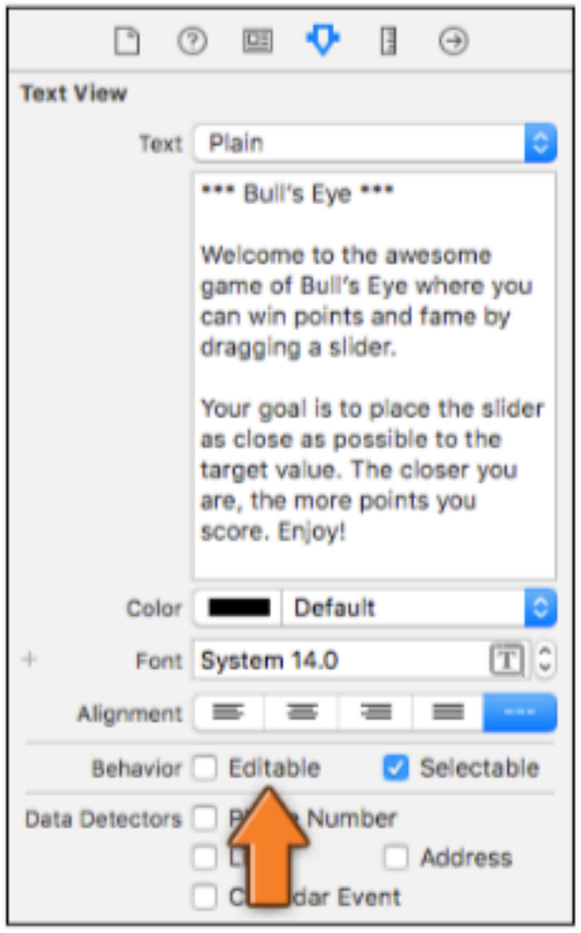
```
*** Bull's Eye ***

Welcome to the awesome game of Bull's Eye where you can win points and
fame by dragging a slider.

Your goal is to place the slider as close as possible to the target
value. The closer you are, the more points you score. Enjoy!
```

You can also enter that text into the Attributes inspector's **Text** property for the text view if you find that easier.

➤ Make sure to uncheck the **Editable** checkbox in the Attribute Inspector. Otherwise, the user can actually type into the text view and you don't want that.

iOS App 1: Part 6

The design of the screen is done for now.



The Attributes inspector for the text view

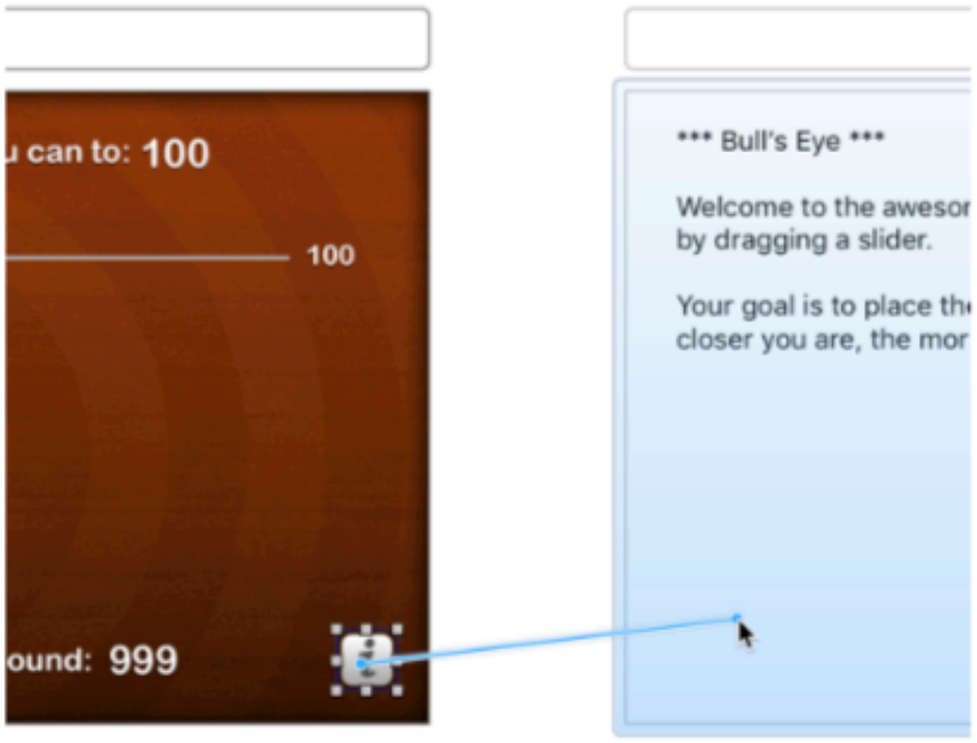
Showing the new view controller

So how do you open this new About screen when the user presses the ⓘ button?

Storyboards have a neat trick for this: *segues* (pronounced "seg-way" like the silly scooters). A segue is a transition from one screen to another. They are really easy to add.

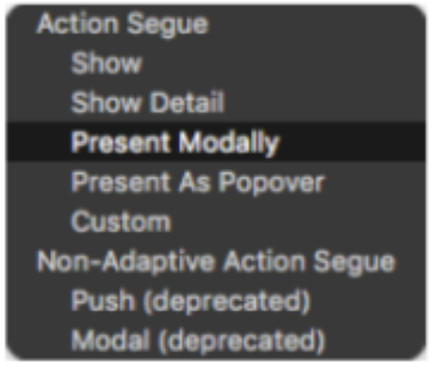
iOS App 1: Part 6

➤ Click the ⓘ button in the **View controller** to select it. Then hold down **Control** and drag over to the **About** screen.



Control-drag from one view controller to another to make a segue

➤ Let go of the mouse button and a pop-up appears with several options. Choose **Present Modally**.



Choosing the type of segue to create

Now an arrow will appear between the two screens. This arrow represents the segue from the main scene to the About scene.

iOS App 1: Part 6

➤ Click the arrow to select it. Segues also have attributes. In the **Attributes inspector**, choose **Transition, Flip Horizontal**. That is the animation that UIKit will use to move between these screens.

Changing the attributes for the segue

➤ Now you can run the app. Press the ⓘ button to see the new screen.

The About screen appears with a flip animation

The About screen should appear with a neat animation. Good, that seems to work.

Dismissing the About view controller

Did you notice that there's an obvious issue here? Tapping the Close button seems to have no effect. Once the user enters the About screen they can never leave… that doesn't sound like good user interface design, does it?

The problem with segues is that they only go one way. To close this screen, you have to hook up some code to the Close button. As a budding iOS developer you already know how to do that: use an action method!

This time you will add the action method to `AboutViewController` instead of `ViewController`, because the Close button is part of the About screen, not the main game screen.

iOS App 1: Part 6

➤ Open **AboutViewController.swift** and replace its contents with the following:

```
import UIKit

class AboutViewController: UIViewController {
  @IBAction func close() {
    dismiss(animated: true, completion: nil)
  }
}
```

The code in the `close()` action method tells UIKit to close the About screen with an animation.

If you had said `dismiss(animated: false, …)`, then there would be no page flip and the main screen would instantly reappear. From a user experience perspective, it's often better to show transitions from one screen to another via an animation.

That leaves you with one final step, hooking up the Close button's Touch Up Inside event to this new `close` action.

➤ Open the storyboard and Control-drag from the **Close** button to the About scene's View Controller. Hmm, strange, the **close** action should be listed in this pop-up, but it isn't. Instead, this is the same pop-up you saw when you made the segue:

The "close" action is not listed in the pop-up

> **Exercise**: Bonus points if you can spot the error. It's a very common – and frustrating! – mistake.

The problem is that this scene in the storyboard doesn't know yet that it is supposed to represent the `AboutViewController`.

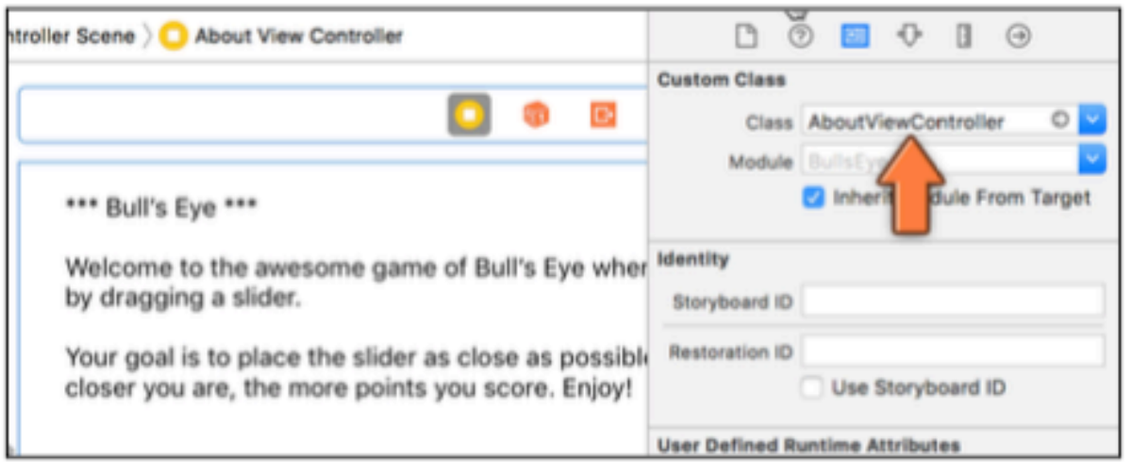
Setting the class for a view controller

You first added the AboutViewController.swift source file, and then dragged a new view controller on to the storyboard. But, you haven't told the storyboard that the design for this new view controller belongs to `AboutViewController`. That's why in the Document Outline it just says View Controller and not About View controller.

iOS App 1: Part 6

That's the design of the screen done for now.➤ Fortunately, this is easily remedied. In Interface Builder, select the About scene's **View controller** and go to the **Identity inspector** (that's the tab/icon to the left of the Attributes inspector).

➤ Under **Custom Class**, enter **AboutViewController**.



The Identity inspector for the About screen

Xcode should auto-complete this for you once you type the first few characters. If it doesn't, then double-check that you really have selected the View controller and not one of the views inside it. (The view controller should also have a blue border on the storyboard to indicate it is selected.)

Now you should be able to connect the Close button to the action method.

➤ Control-drag from the **Close** button to **About View controller** in the Document Outline (or to the yellow circle at the top of the scene in the storyboard). This should be old hat by now. The pop-up menu now does have an option for the **close** action (under Sent Events). Connect the button to that action.

➤ Run the app again. You should now be able to return from the About screen.

OK, that does get us a working About screen, but it does look a little plain doesn't it? What if you added some of the design changes you made to the main screen?

> **Exercise**: Add a background image to the About screen. Also, change the Close button on the About screen to look like the Hit Me! button and play around with the Text View properties in the Attribute Inspector. You should be able to do this by yourself now. Piece of cake! Refer back to the instructions for the main screen if you get stuck.

iOS App 1: Part 6

When you are done, you should have an About screen which looks something like this:

The new and improved About screen

That looks good, but it could be better. So how do you improve upon it?

Using a web view for HTML content

➤ Now select the **text view** and press the **Delete** key on your keyboard. (Yep, you're throwing it away, and after all those changes, too! But don't grieve for the Text View too much, you'll replace it with something better.)

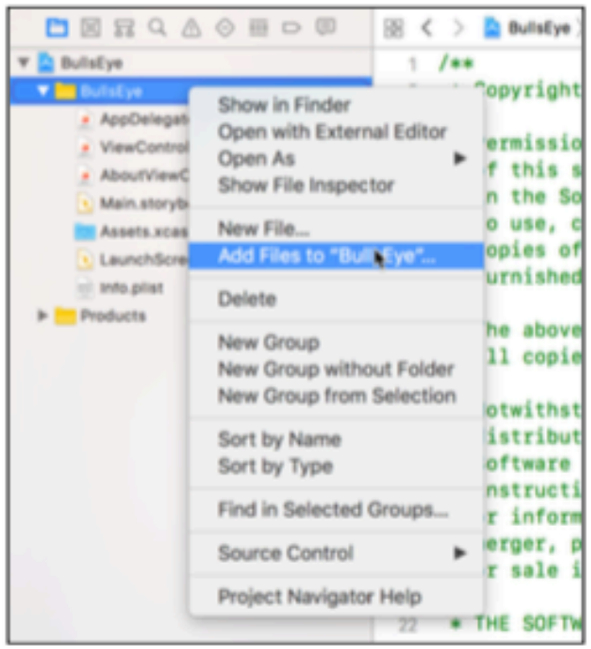
➤ Put a **WebKit View** in its place (as always, you can find this view in the Objects Library). There are two web view options — an older Web View, which is deprecated, or ready to be retired, and the WebKit View. Make sure that you select the WebKit View.

This view can show web pages. All you have to do is give it the URL to a web site or the name of a file to load. The WebKit View object is named `WKWebView`.

For this app, you will make it display a static HTML page from the application bundle, so it won't actually have to go online and download anything.

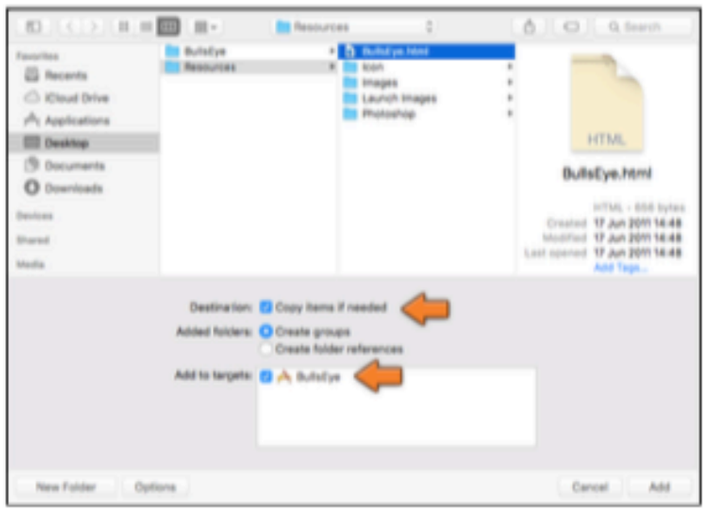
iOS App 1: Part 6

➤ Go to the **Project navigator** and right-click on the **BullsEye** group (the yellow folder). From the menu, choose **Add Files to "BullsEye"...**



Using the right-click menu to add existing files to the project

➤ In the file picker, select the **BullsEye.html** file from the Resources folder. This is an HTML5 document that contains the gameplay instructions.



Choosing the file to add

Make sure that **Copy items if needed** is selected and that under **Add to targets**, there is a checkmark in front of **BullsEye**. (If you don't see these options, click the Options button at the bottom of the dialog.)

iOS App 1: Part 6

➤ Press **Add** to add the HTML file to the project.

➤ In **AboutViewController.swift**, add an outlet for the web view:

```
class AboutViewController: UIViewController {
  @IBOutlet weak var webView: WKWebView!
  . . .
}
```

Xcode will complain soon after you add the above line. The error should look something like this:

Xcode complains about WKWebView

What does this error mean? It means that Xcode, or rather the compiler, does not know what `WKWebView` is.

But how can that be? We selected the component from Xcode's own Objects Library and so it should be supported, right?

The answer to this lies with this line of code at the top of both your view controller source files:

```
import UIKit
```

I'm sure you saw this line and wondered what it was about. That statement tells the compiler that you want to use the objects from a framework named `UIKit`. Frameworks, or libraries if you prefer, bundle together one or more objects which perform a particular type of task (or tasks). The `UIKit` library provides all ther UI components for iOS.

So why does `UIKit` not contain `WKWebView`, you ask? That's because the previously mentioned deprecated WebView is the one which is included with `UIKit`. The newer (and improved) `WKWebView` comes from a different framework called `WebKit`.

➤ Add the following code at the top of **AboutViewController.swift**, right below the existing `import` statement:

```
import WebKit
```

iOS App 1: Part 6

That tells the compiler that we want to use objects from the `WebKit` framework and since now the compiler knows about all the objects in the `WebKit` framework, the Xcode error will go away.

➤ In the storyboard file, connect the `UIWebView` to this new outlet. The easiest way to do this is to Control-drag from **About View controller** (in the Document Outline) to the **Web View**.

➤ In **AboutViewController.swift**, add a `viewDidLoad()` implementation:

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

  if let url = Bundle.main.url(forResource: "BullsEye",
                               withExtension: "html") {
    let request = URLRequest(url: url)
    webView.load(request)
  }
}
```

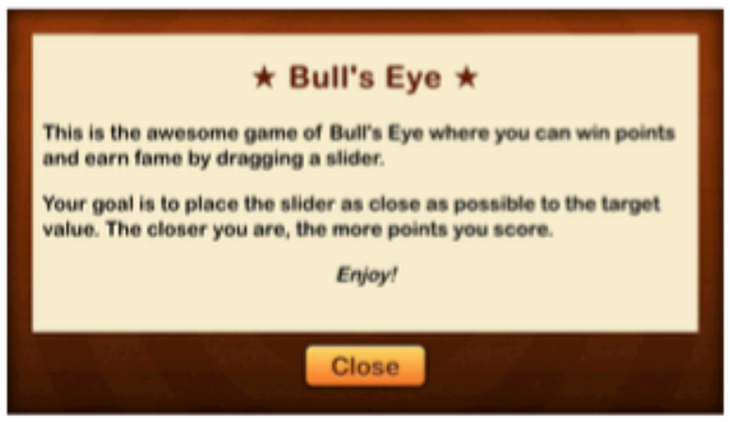
This displays the HTML file using the web view.

The code first gets the URL (Uniform Resource Locator) for the **BullsEye.html** file in the application bundle. A URL, as you might be familiar with from the Interwebs, is a way to identify the location of a resource, like a web page. Here, the URL provides the location of the HTML file in your application bundle.

It then creates a `URLRequest` using that URL since that's one of the easiest ways to send a load request to the web view.

Finally, the code asks the web view to load the contents specified by the URL request.

➤ Run the app and press the info button. The About screen should appear with a description of the gameplay rules, this time in the form of an HTML document:



The About screen in all its glory

Congrats! This completes the game. All the functionality is there and – as far as I can tell – there are no bugs to spoil the fun.

FEEDBACK TIME

http://bit.ly/iOSFeedback

THANK YOU