Chapter 7: The New Look

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

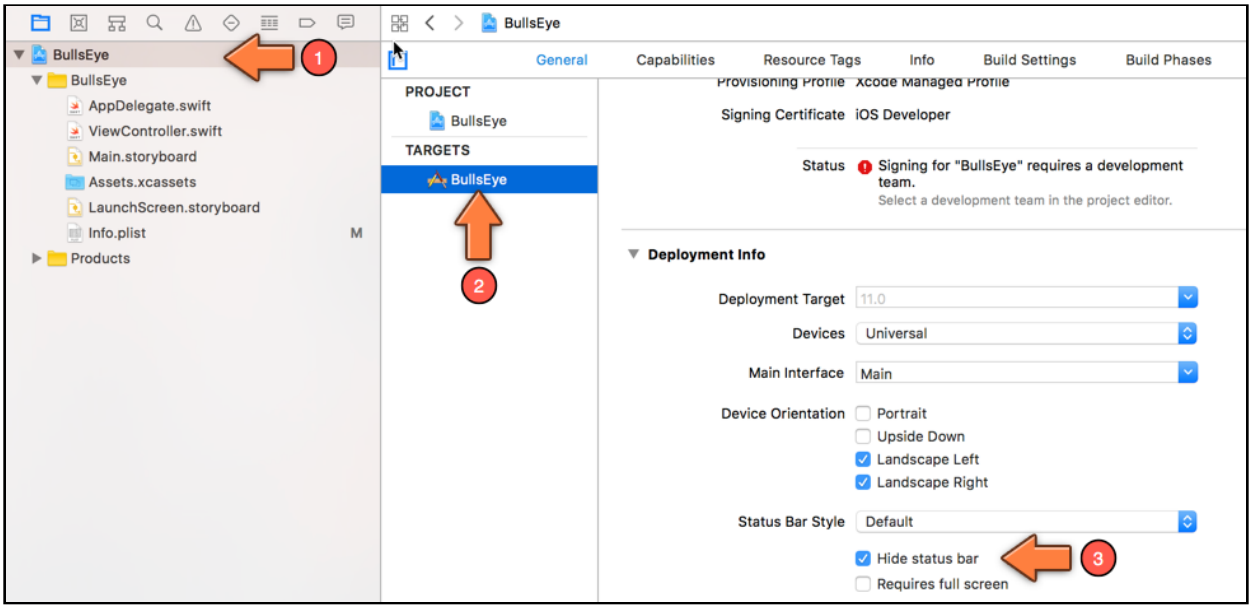
First, let's quickly revisit another item in the to-do list - "Put the app in landscape orientation." You already did this, right? But there's a little bit of clean up to be done with regards to that item.

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.

Even though the system automatically handles not showing 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 the option **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

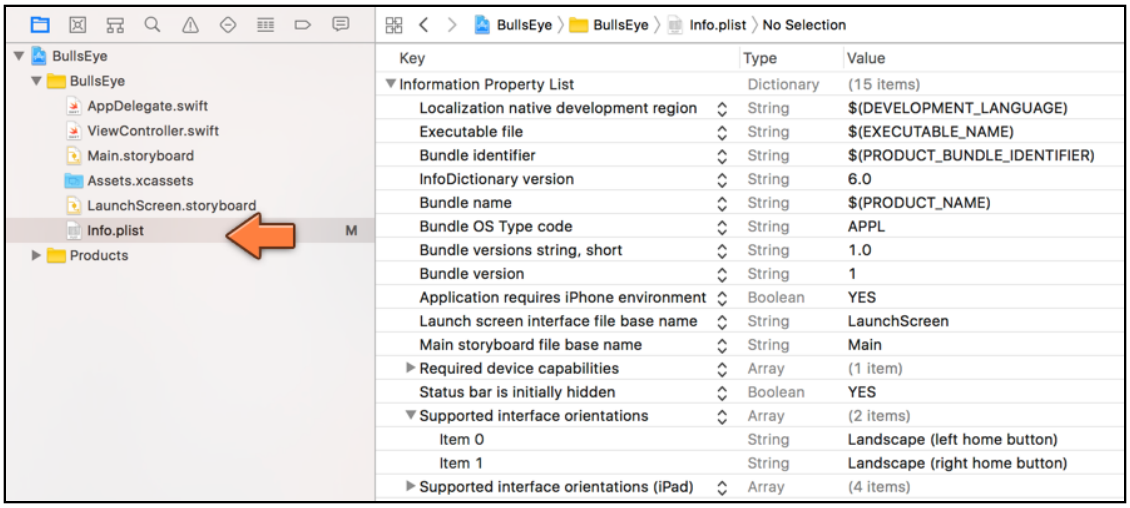
Most of the options from the Project Settings screen, such as the supported device orientations and whether the status bar is visible during launch, get stored in your app's Info.plist file.

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.

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.

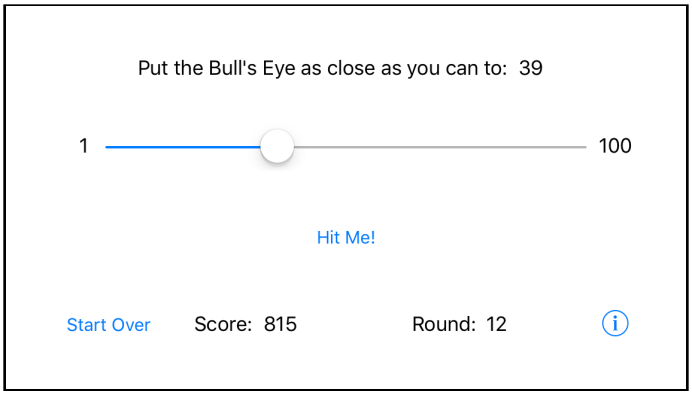


The Info.plist file is just a list of configuration options and their values. Most of these may not make sense to you, but that's OK – they don't always make sense to me either.

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

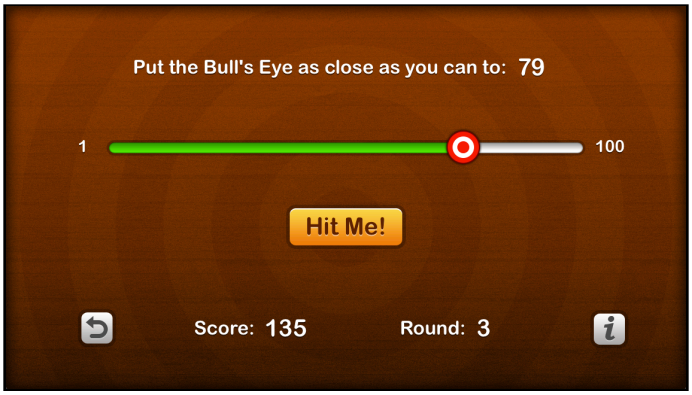
Spice up the graphics

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



Yawn...

To something that's more like this:



Cool :-)

The actual controls don't change. You'll simply use 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.

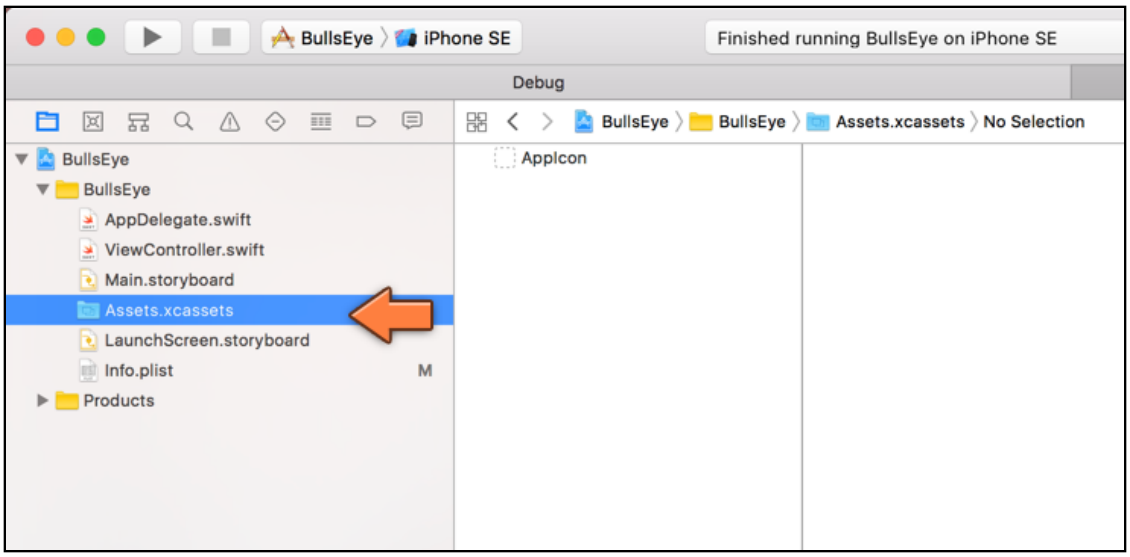
Add the image assets

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.

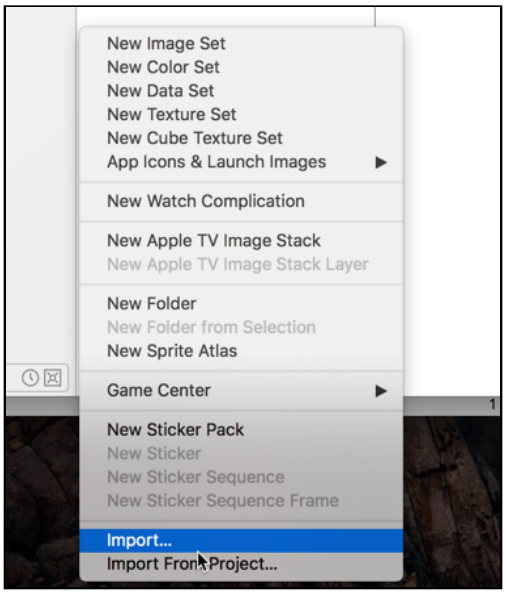
➤ In the **Project navigator**, find **Assets.xcassets** and click on it.

This is known as the asset catalog for the app and it contains all the app's images. Right now, it is empty and contains just a placeholder for the app icon, which you'll add soon.



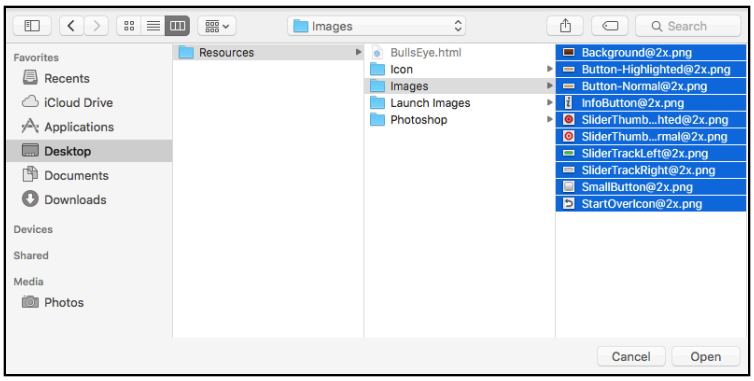
The asset catalog is initially empty

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



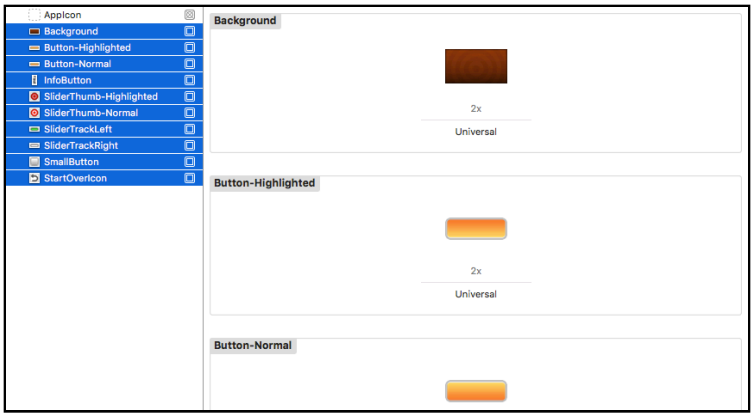
Choose Import to put existing images into the asset catalog

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

Currently, 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.

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 11 – 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.

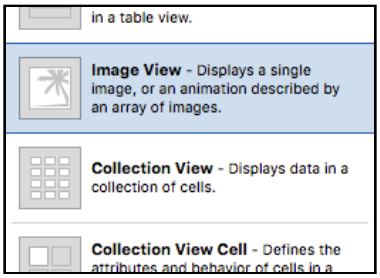
3x is for the super high-resolution Retina HD screen of the iPhone Plus devices. If you want your app to have extra sharp images on these top-of-the-line iPhone models, then you can drop them into the "3x" slot in the asset catalog.

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).

Put up the wallpaper

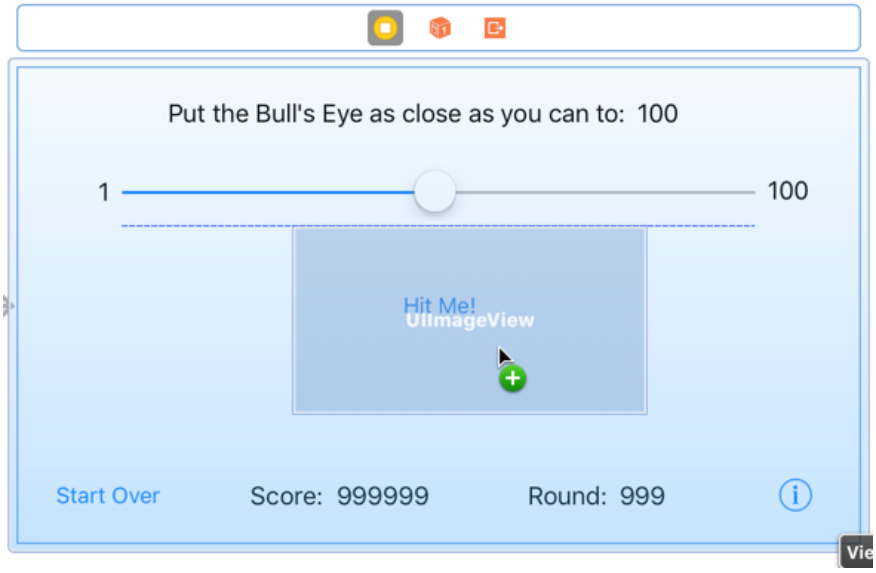
Let's begin by changing the drab white background in *Bull's Eye* to something more fancy.

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



The Image View control in the Object Library

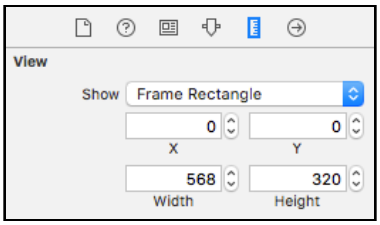
➤ Drag the image view on top of the existing user interface. It doesn't really matter where you put it, as long as it's inside the Bull's Eye View Controller.



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.

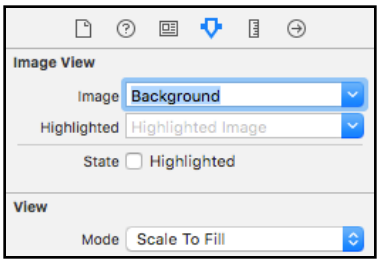
This will make the image view cover the entire screen.



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

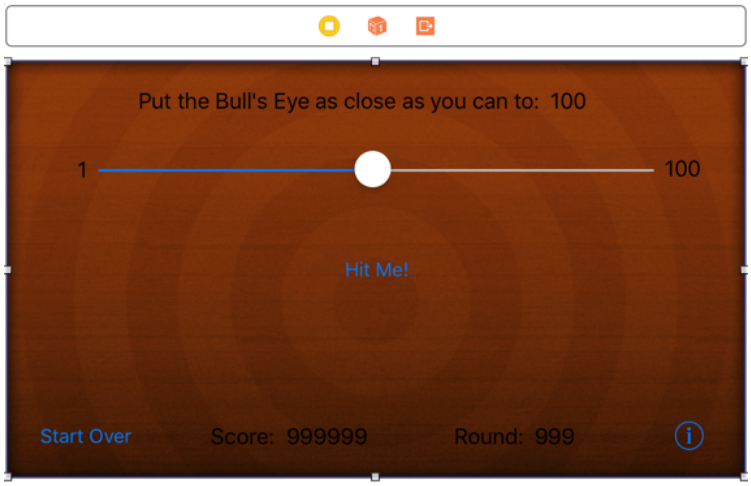
There is only one problem: the image now covers all the other controls. There is an easy fix for that; you have to move the image view behind the other views.

➤ 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 View, 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:



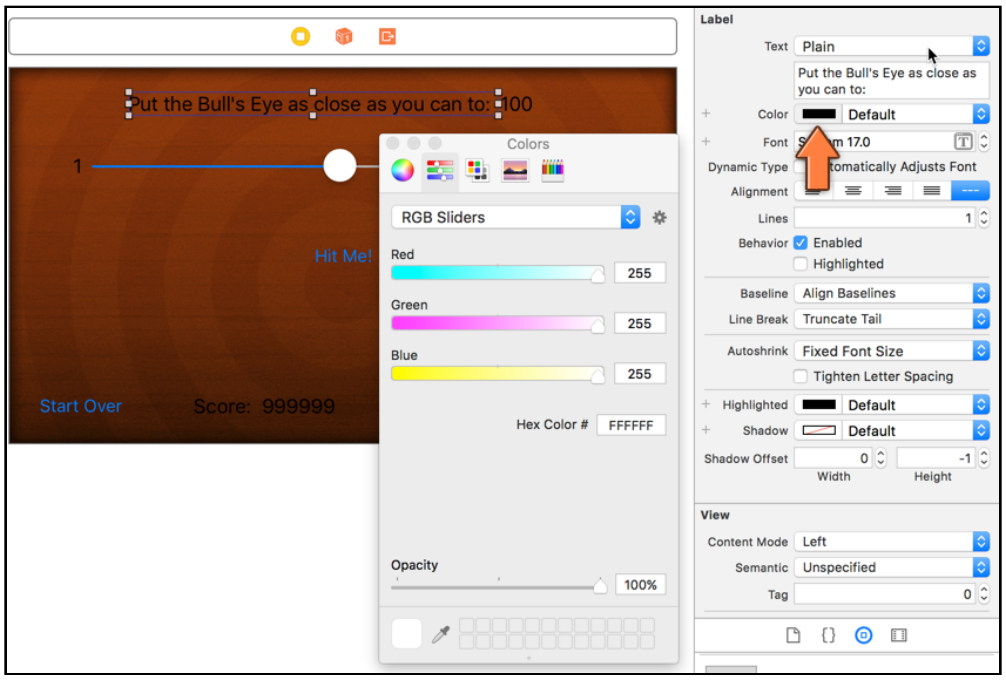
The game with the new background image

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

Change the labels

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

➤ Still in the storyboard, select the label at the top, open the **Attributes inspector** and click on the **Color** item - there are two parts to the item, so you need to click on the actual color and not the text part.



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 select box at the top.

➤ Pick a pure white color, Red: 255, Green: 255, Blue: 255, Opacity: 100%.

➤ 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%.

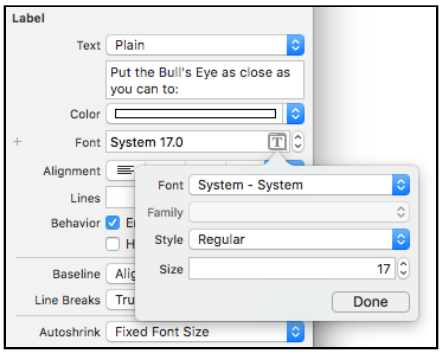
Note: Sometimes when you change the Color or Shadow attributes, the background color of the view also changes. This is a bug in Xcode. Put it back to Clear Color if that happens.

➤ 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.

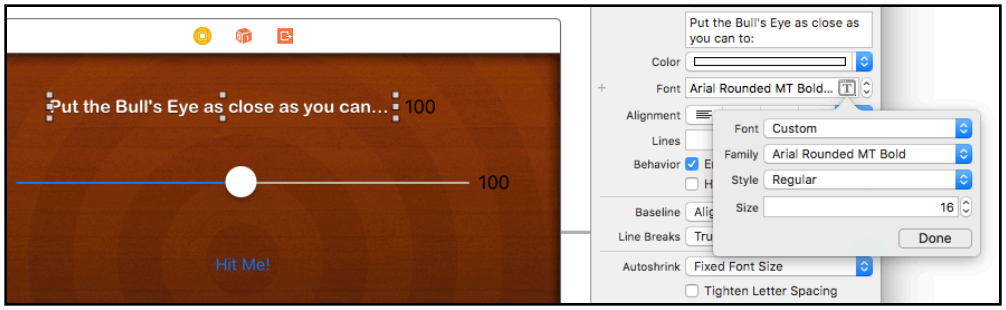
➤ Click on the **[T]** icon of the **Font** attribute. This opens the Font Picker.

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

➤ 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.)

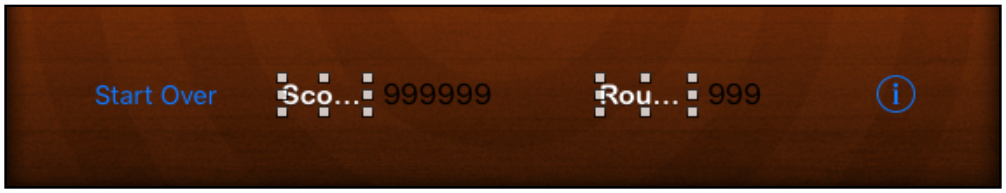
The label will now become slightly larger or smaller so that it fits snugly around the text. If the text got cut off when you changed the font, now all the text will show again.

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.

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

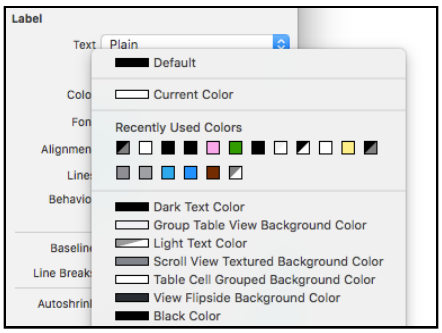


The font is too large to fit all the text in the Score and Round labels

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.

Click the tiny arrows at the end of the color field (or, if there is a text name for the color, click on the text part) and the menu will pop up:

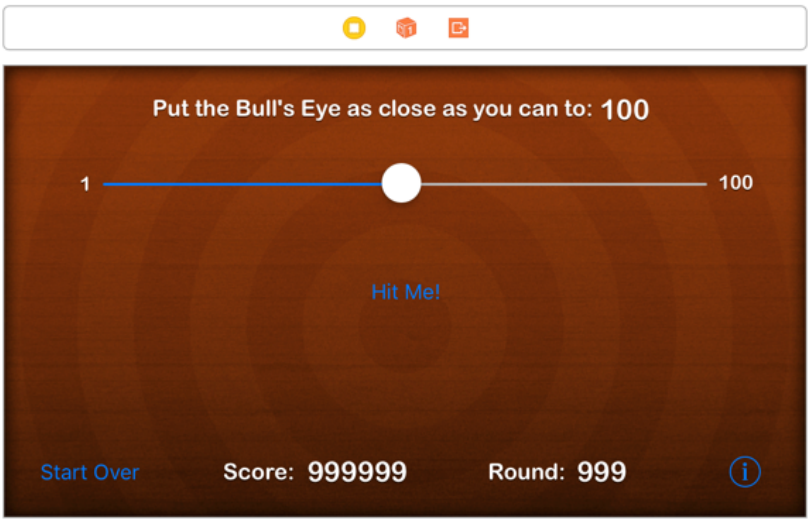


Quick access to recently used colors and several handy presets

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.

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.

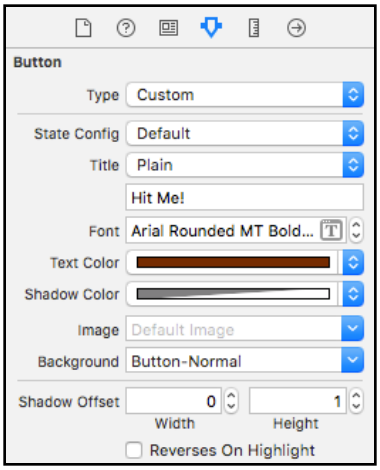
➤ Set the **Font** to **Arial Rounded MT Bold**, size 20.

➤ 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.

> **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.

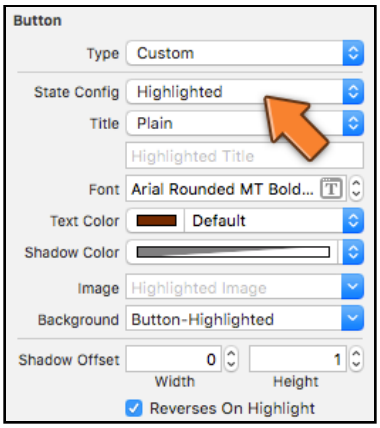
➤ With the button still selected, click the **State Config** setting and pick **Highlighted** from the menu. Now the attributes in this section reflect the highlighted state of the button.

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

➤ 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.

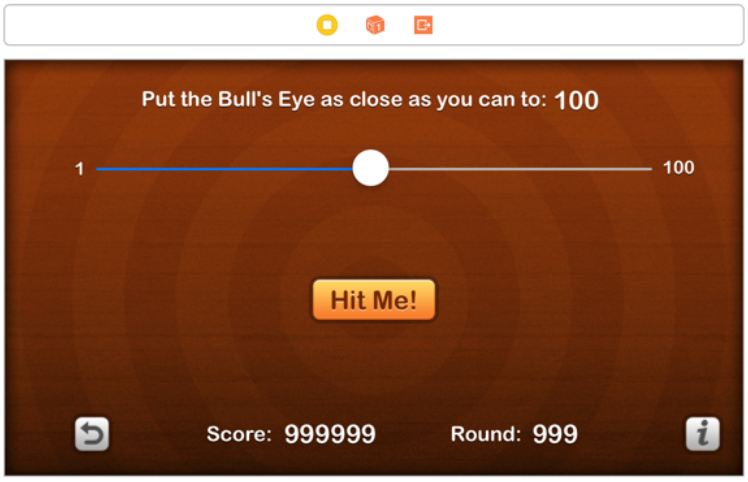
➤ Select the **Start Over** button and change the following attributes:

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

Everything you have 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")!
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)
```

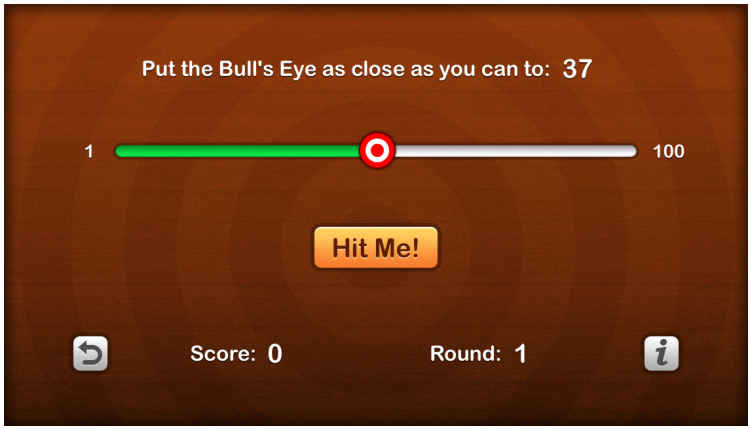
```
let trackRightImage = UIImage(named: "SliderTrackRight")!
let trackRightResizable =
                  trackRightImage.resizableImage(withCapInsets: insets)
slider.setMaximumTrackImage(trackRightResizable, for: .normal)
```

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.

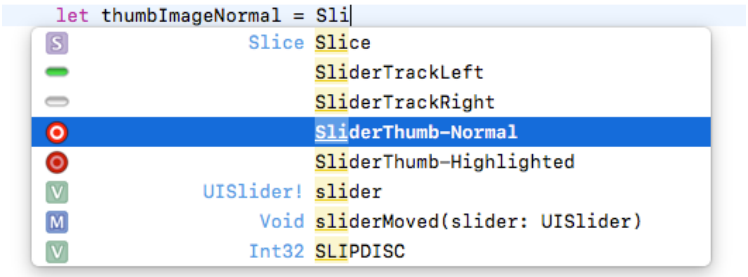
Tip: Xcode now has a handy new feature that makes it really easy to add images in your code. Instead of writing:

```
let thumbImageNormal = UIImage(named: "SliderThumb-Normal")
```

You can now type:

```
let thumbImageNormal = Sli
```

And Xcode's autocomplete will kick in and show a list of suggestions to complete the text `Sli`, including any images whose names start with those letters.



Xcode autocomplete also shows images

Pick **SliderThumb-Normal** from the list and it will add a tiny icon of the image into the code! This tiny icon is known as an *image literal*. If you do the same for the other images, your code will look like this:

```
// Customize slider
let thumbImageNormal = 
slider.setThumbImage(thumbImageNormal, for: .normal)

let thumbImageHighlighted = 
slider.setThumbImage(thumbImageHighlighted, for: .highlighted)

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

let trackLeftImage = 
let trackLeftResizable =
  trackLeftImage.resizableImage(withCapInsets: insets)
slider.setMinimumTrackImage(trackLeftResizable, for: .normal)

let trackRightImage = 
let trackRightResizable =
  trackRightImage.resizableImage(withCapInsets: insets)
slider.setMaximumTrackImage(trackRightResizable, for: .normal)
```

The images are now part of your source code

Give it a try! I really like how it shows a tiny thumbnail of the image right in the code.

Run your app once again to verify that adding the image literals did not change the functionality of the game in any way. It shouldn't, but it's always good to be sure, right?

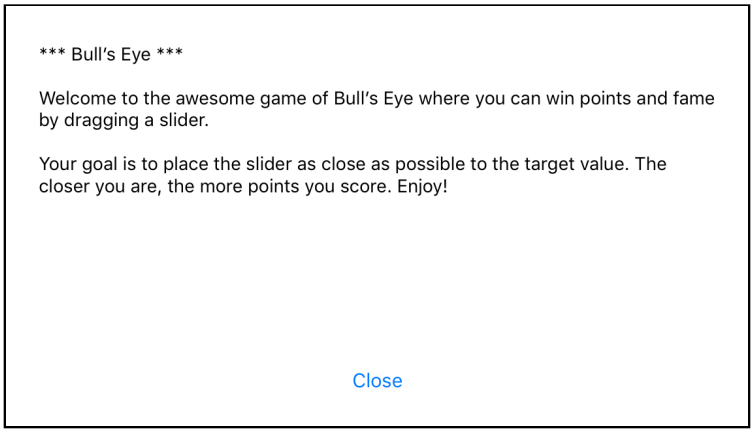
The About Screen

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.

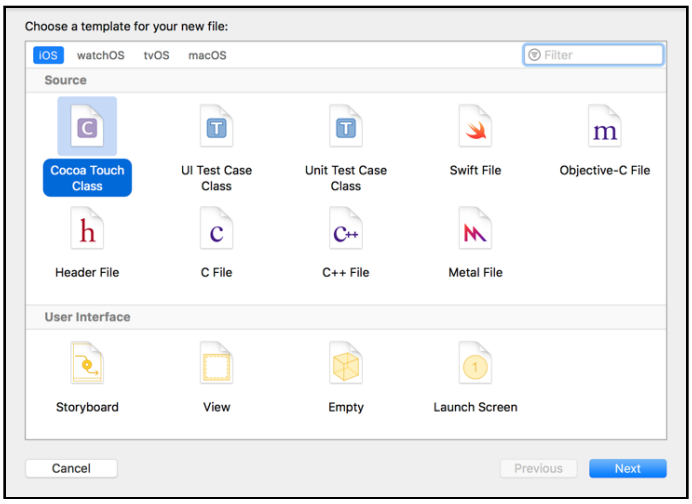
Most apps have more than one screen, even very simple games. So, this is as good a time as any to learn how to add additional screens to your apps.

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.

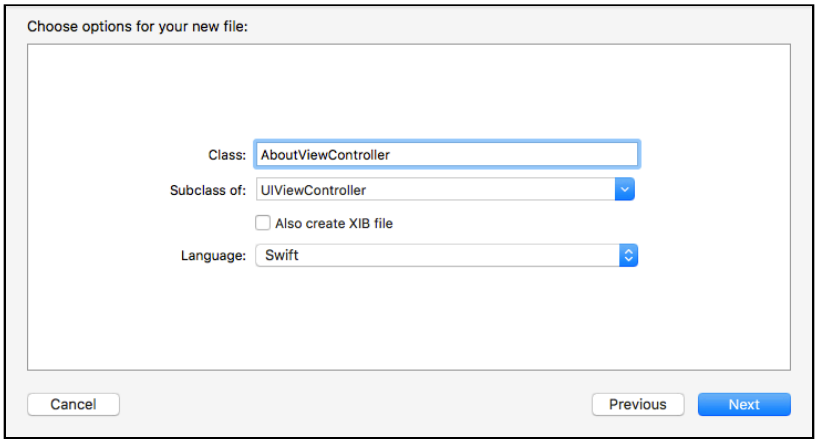
Add a new view controller

➤ 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:

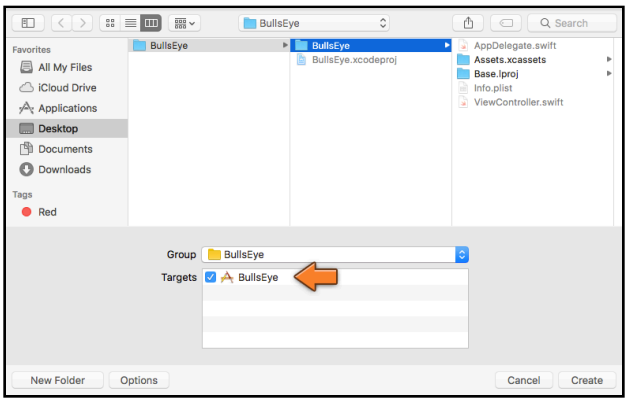


The options for the new file

Choose the following:

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

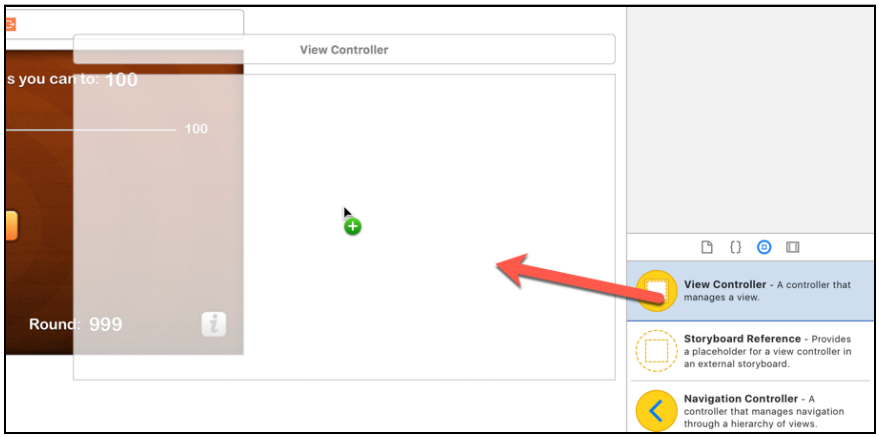
Xcode will create a new file and add it to your project. As you might have guessed, the new file is **AboutViewController.swift**.

Design 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.

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



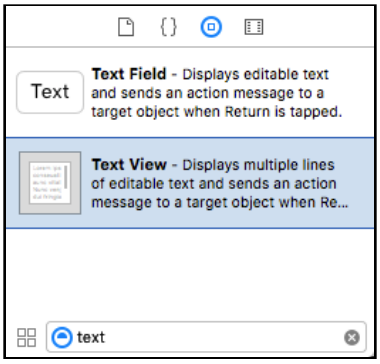
Dragging a new View Controller from the Object Library

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 in 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.

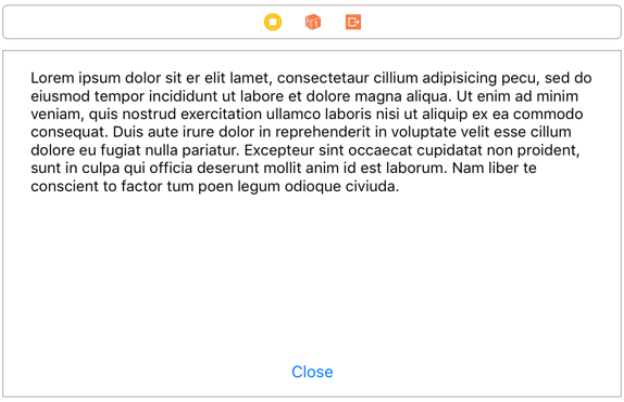
You can find these components in the Object Library. If you don't feel like scrolling, you can filter the components by typing in the field at the bottom:



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.

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

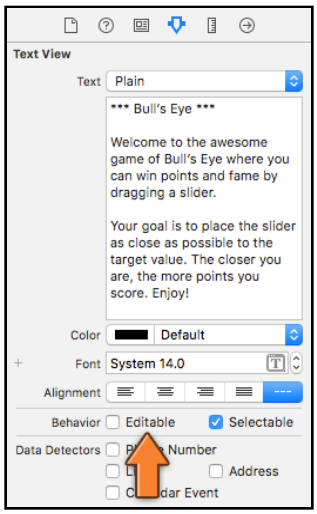
➤ Double-click the text view to make its content is editable. By default, the Text View contains a bunch of Latin placeholder text (also known as "Lorem Ipsum").

Copy-paste 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 paste 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.



The Attributes inspector for the text view

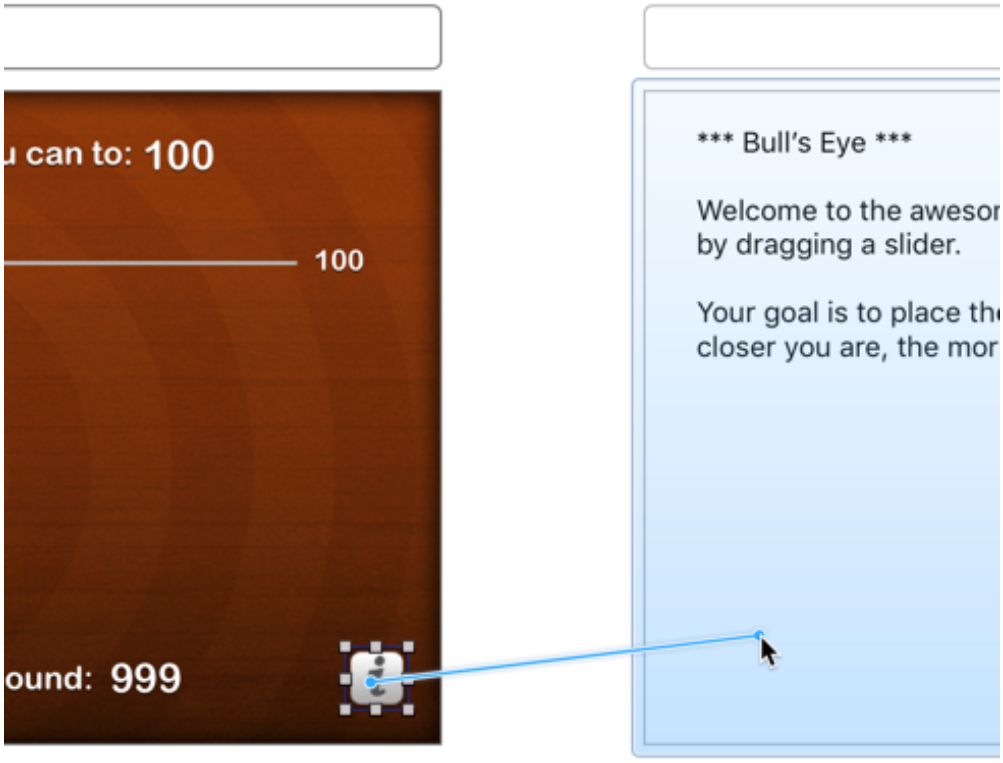
That's the design of the screen done for now.

Show the new view controller

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

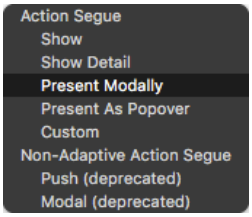
Storyboards have a neat trick for this: *segues* (pronounced "seg-way" like the silly scooters). A segue is a transition from one screen to another. They are really easy to add.

➤ 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

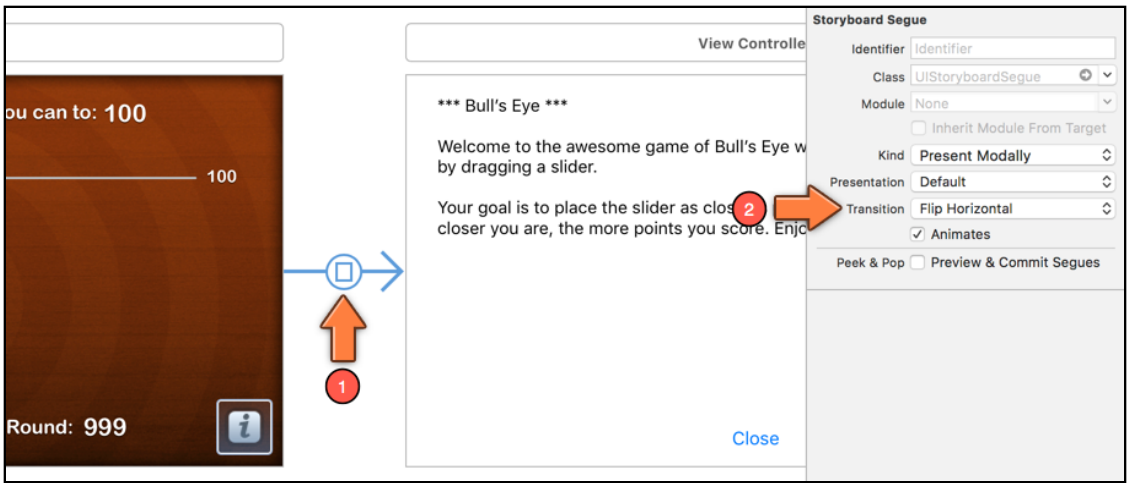
➤ Let go of the mouse button and a popup appears with several options. Choose **Present Modally**.



Choosing the type of segue to create

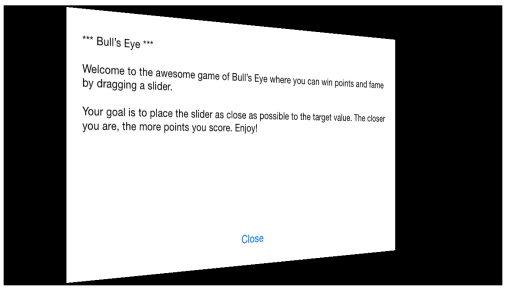
Now an arrow will appear between the two screens. This arrow represents the segue from the main scene to the About scene.

➤ 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.

Dismiss the About view controller

However, there is an obvious shortcoming 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 to me, 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.

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

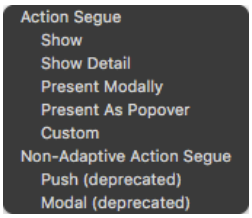
```
import UIKit

class AboutViewController: UIViewController {

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

This 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 popup, but it isn't. Instead, this is the same popup you saw when you made the segue:



The "close" action is not listed in the popup

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

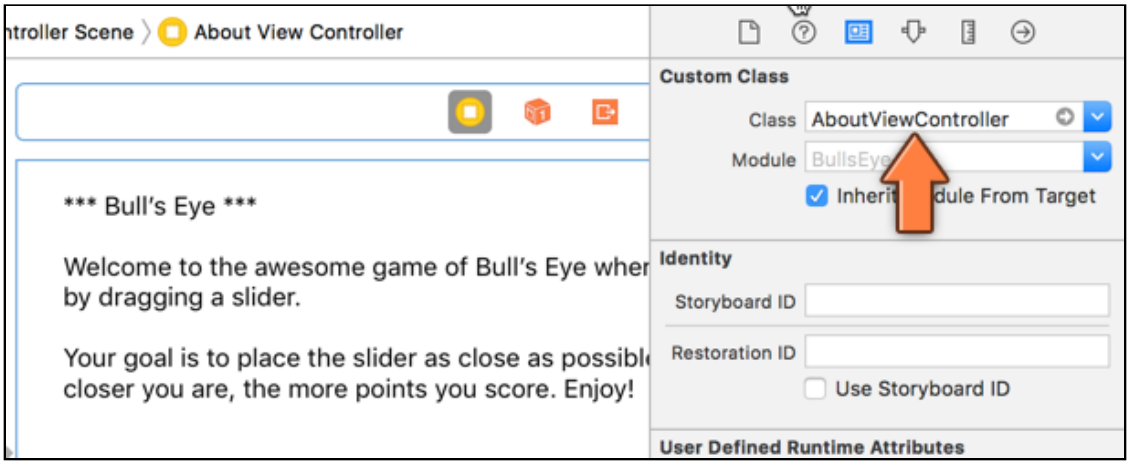
The problem is that this scene in the storyboard doesn't know yet that it is supposed to represent the `AboutViewController`.

Set 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, in fact, belongs to `AboutViewController`. (That's why in the Document Outline it just says View Controller and not About View Controller.)

➤ Fortunately, this is easily remedied. In Interface Builder, select the About scene's **View Controller** and go to the **Identity inspector** (that's the button to the left of the Attributes inspector).

➤ Under **Custom Class**, type **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 storyboard). This should be old hat by now. The popup 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.

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?

Use a web view for HTML content

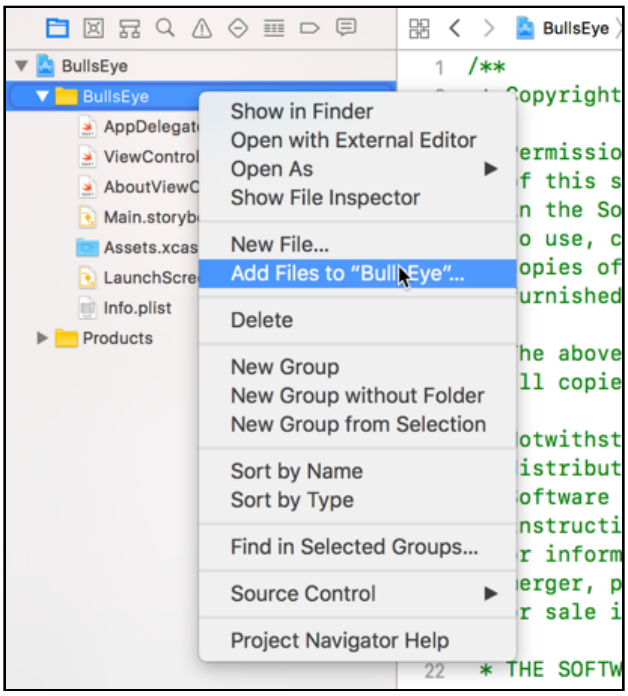
➤ Now select the **text view** and press the **Delete** key on your keyboard. (Yep, you're throwing it away, and after all those changes, too! But don't grieve for the Text View too much, you'll replace it with something better next.)

➤ Put a **Web View** in its place (as always, you can find this view in the Object Library).

A web view, as its name implies, can show web pages. All you have to do is give it a URL to a web site or the name of a file to load. The web view object is named `UIWebView`.

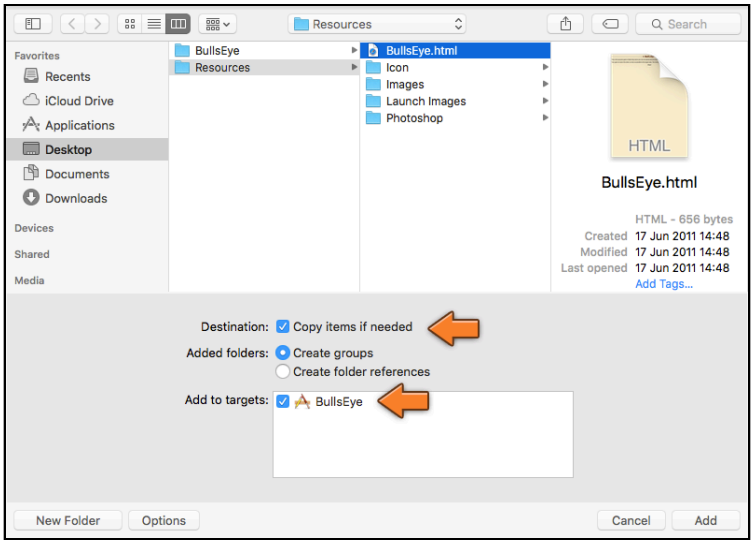
For this app, you will make it display a static HTML page from the application bundle, so it won't actually have to go online and download anything.

➤ 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.)

➤ 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: UIWebView!
  . . .
}
```

➤ 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**.

(If you do it the other way around, from the Web View to About View Controller, then you'll connect the wrong thing and the web view will stay empty when you run the app.)

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

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

  if let url = Bundle.main.url(forResource: "BullsEye",
                               withExtension: "html") {
    if let htmlData = try? Data(contentsOf: url) {
      let baseURL = URL(fileURLWithPath: Bundle.main.bundlePath)
      webView.load(htmlData, mimeType: "text/html",
                   textEncodingName: "UTF-8",
                            baseURL: baseURL)
    }
  }
}
```

This displays the HTML file using the web view.

The code may look scary but what goes on is not really that complicated: first it finds the **BullsEye.html** file in the application bundle, then loads it into a `Data` object, and finally it asks the web view to show the contents of this data object.

➤ 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.

You can find the project files for the finished app under **07 - The New Look** in the Source Code folder.

8 Chapter 8: The Final App

You might be thinking, "OK, *Bull's Eye* is now done, and I can move on to the next app!" If you were, I'm afraid you are in for disappointment - there's just a teensy bit more to do in the game.

"What? What's left to do? We finished the task list!" you say? You are right. The game is indeed complete. However, all this time, you've been developing and testing for a 4" iPhone screen found on devices such as the iPhone 5, 5c, and SE. But what about other iPhones such as the 4.7-inch iPhone, the 5.5-inch iPhone Plus, or the 5.8-inch iPhone X which have bigger screens? Or the iPad with its multiple screen sizes? Will the game work correctly on all these different screen sizes?

And if not, shouldn't we fix it?

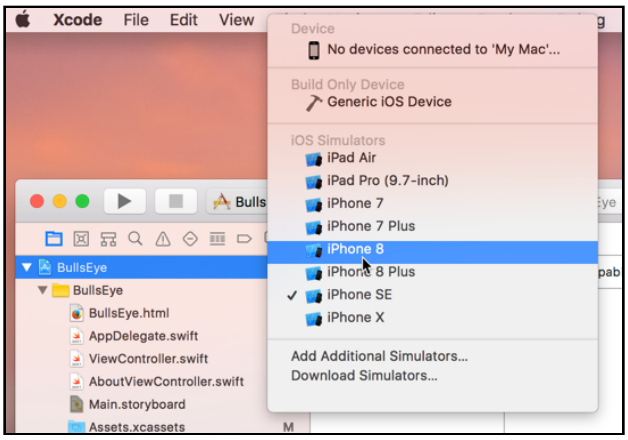
This chapter covers the following:

- **Support different screen sizes:** Ensure that the app will run correctly on all the different iPhone and iPad screen sizes.
- **Crossfade:** Add some animation to make the transition to the start of a new game a bit more dynamic.
- **The icon:** Add the app icon.
- **Display name:** Set the display name for the app.
- **Run on device:** How to configure everything to run your app on an actual device.

Support different screen sizes

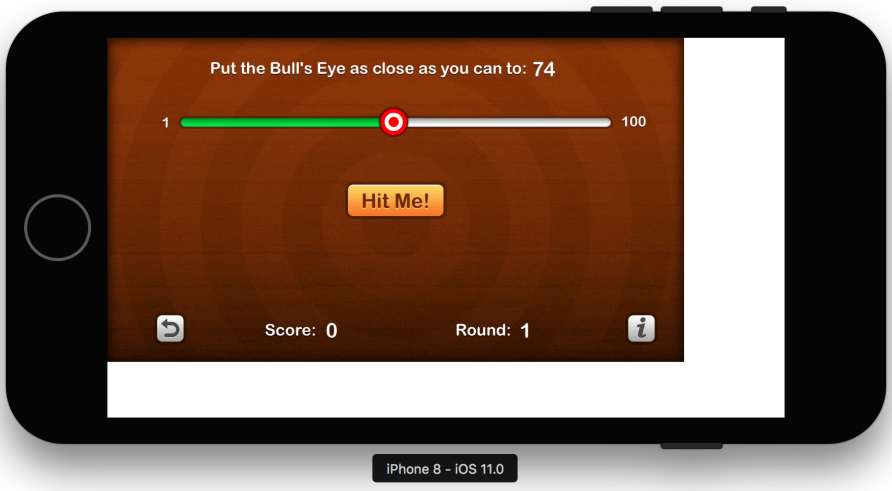
First, let's check if there is indeed an issue running Bull's Eye on a device with a larger screen. It's always good to verify that there's inded an issue before we do extra work, right? Why fix it, if it isn't broken? :]

➤ To see how the app looks on a larger screen, run the app on an iPhone simulator like the **iPhone 8**. You can switch between Simulators using the selector at the top of the Xcode window:



Using the scheme selector to switch to the iPhone 8 Simulator

The result might not be what you expected:



On the iPhone 8 Simulator, the app doesn't fill up the entire screen

Obviously, this won't do. Not everybody is going to be using a 4" iOS device. And you don't want the game to display on only part of the screen for the rest of the people!

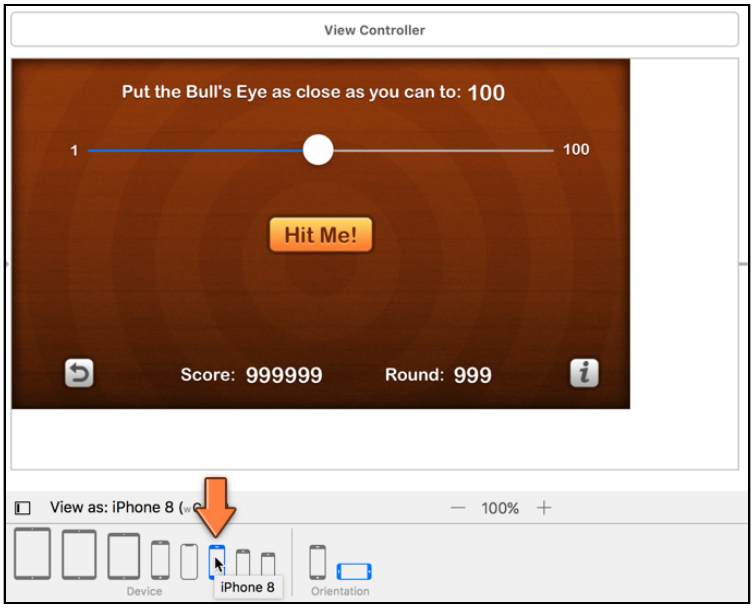
This is a good opportunity to learn about *Auto Layout*, a core UIKit technology that makes it easy to support many different screen sizes in your apps, including the larger screens of the 4.7-inch, 5.5-inch, and 5.8-inch iPhones, and the iPad.

Tip: You can use the **Window → Scale** menu to resize a simulator if it doesn't fit on your screen. Some of those simulators, like the iPad one, can be monsters! Also, with Xcode 9 onwards, you can resize a simulator window by simply dragging on one corner of the window - just like you do to resize any other window on macOS.

Interface Builder has a few handy tools to help you make the game fit on any screen.

The background image

➤ Go to **Main.storyboard**. Open the **View as:** panel at the bottom and choose the **iPhone 8** device. (You may need to change the orientation back to landscape.)



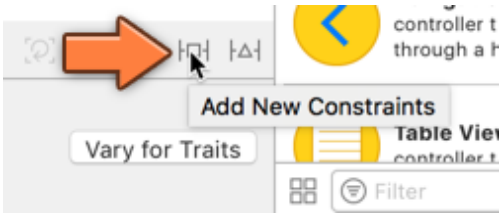
Viewing the storyboard on iPhone 8

The storyboard should look like your screen from when you ran on the iPhone 8 Simulator. This shows you how changes on the storyboard affect the bigger iPhone screens.

First, let's fix the background image. At its normal size, the image is too small to fit on the larger screens.

This is where Auto Layout comes to the rescue.

➤ In the storyboard, select the **Background image view** on the main **View Controller** and click the small **Add New Constraints** button at the bottom of the Xcode window:

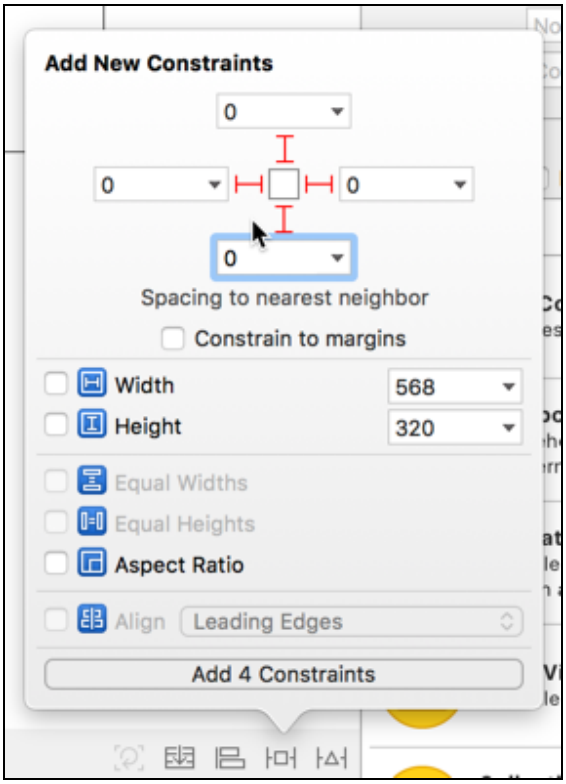


The Add New Constraints button

This button lets you define relationships, called *constraints*, between the currently selected view and other views in the scene. When you run the app, UIKit evaluates these constraints and calculates the final layout of the views. This probably sounds a bit abstract, but you'll see soon enough how it works in practice.

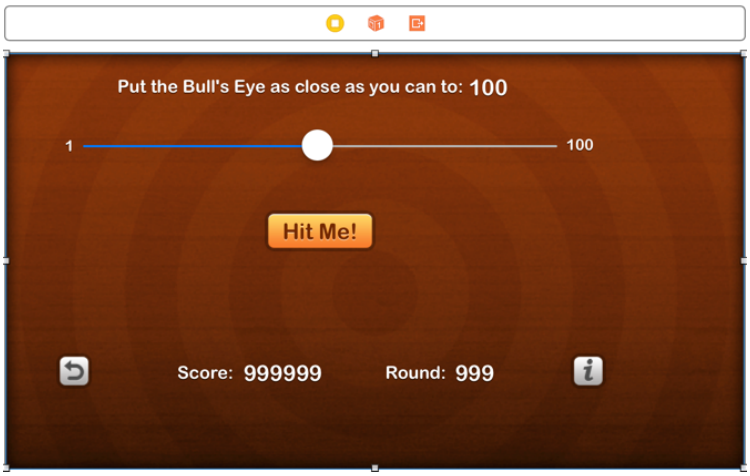
In order for the background image to stretch from edge-to-edge on the screen, the left, top, right, and bottom edges of the image should be flush against the screen edges. The way to do this with Auto Layout is to create two alignment constraints, one horizontal and one vertical.

➤ In the **Add New Constraints** menu, set the **left**, **top**, **right**, and **bottom** spacing to zero and make sure that the red I-beam markers next to (or below) each item is enabled. (The red I-beams are used to specify which constraints are enabled when adding new constraints.):



Using the Add New Constraints menu to position the background image

➤ Press **Add 4 Constraints** to finish. The background image will now cover the view fully. (Press Undo and Redo a few times to see the difference.)



The background image now covers the whole view

You might have also noticed that the Document Outline now has a new item called **Constraints**:

The new Auto Layout constraints appear in the Document Outline

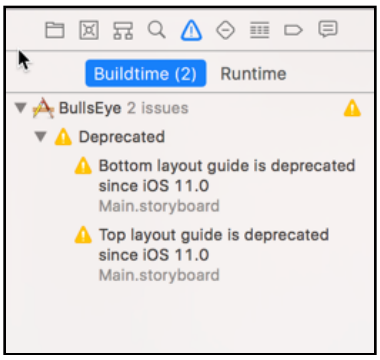
There should be four constraints listed there, one for each edge of the image.

➤ Run the app again on the iPhone 8 Simulator and also on the iPhone SE Simulator. In both cases, the background should display correctly now. (Of course, the other controls are still off-center, but we'll fix that soon.)

If you use the **View as:** panel to switch the storyboard back to the iPhone SE, the background should display correctly there too.

Compiler warnings

When you run the app after adding your first autolayout constraints, sometimes you might see some compiler warnings similar to this:



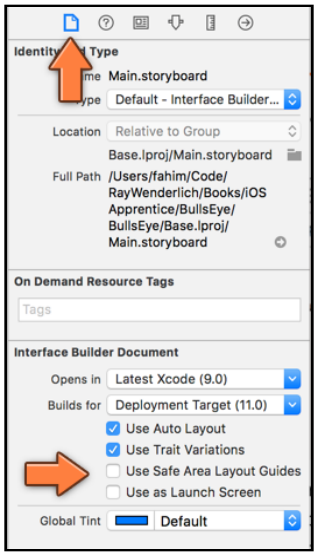
Auto Layout deprecation warnings

If this happens to you, that is because in iOS 11 there were some changes to how autolayout works and how constraints are set up. In previous versions of iOS, you had markers called *top layout guide* and *bottom layout guide* which defined the usable area of a screen. These guides were useful in setting your own views to stretch to the top edge (or the bottom of edge) of the screen without covering any on-screen elements provided by the OS such as navigation bars or tab bars.

However, in iOS 11, they introduced a new layout mechanism which was more flexible than the previously used top and bottom layout guides. These new layout guides are known as the *safe area layout guides*.

So how do you use these new safe area layout guides, you ask? Simple enough, you just have to enable them for your storyboard :]

Switch to your **storyboard**, select your view controller, and then on the right-hand pane, go to the **File Inspector**.



Enable Safe Area Layout Guides

Under the **Interface Builder Document** section, there should be a checkbox for **Use Safe Area Layout Guides** - check it. That's it, you are now using safe area layout guides in your storyboard and the compiler warnings should go away!

If you do not see the **Use Safe Area Layout Guides** checkbox, make sure that you have the view controller selected - that particular option appears only when you have a view controller selected.

The About screen

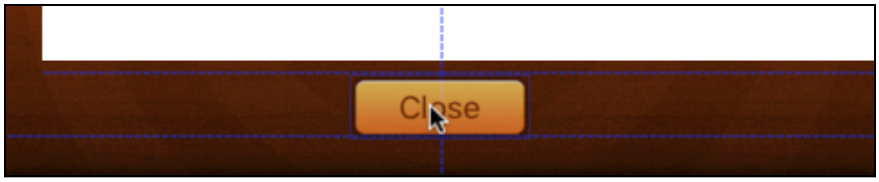
Let's repeat the background image fix for the About screen, too.

➤ Use the **Add New Constraints** button to pin the About screen's background image view to the parent view.

The background image is now fine. Of course, the Close button and web view are still completely off.

➤ In the storyboard, drag the **Close** button so that it snaps to the center of the view as well as the bottom guide.

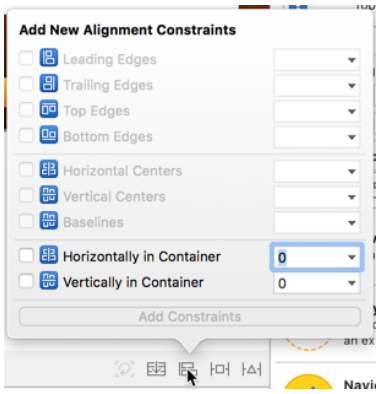
Interface Builder shows a handy guide, the dotted blue line, near the edges of the screen, which is useful for aligning objects by hand.



The dotted blue lines are guides that help position your UI elements

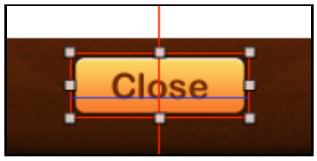
You want to create a centering constraint that keeps the Close button in the middle of the screen, regardless of how wide the screen is.

➤ Click the **Close** button to select it. From the **Align** menu (which is to the left of the Add New Constraints button), choose **Horizontally in Container** and click **Add 1 Constraint**.



The Align menu

Interface Builder now draws a red bar to represent the constraint, and a red box around the button as well.



The Close button has red constraints

That's a problem: the bars are all supposed to be blue, not red. Red indicates that something is wrong with the constraints, usually that there aren't enough of them.

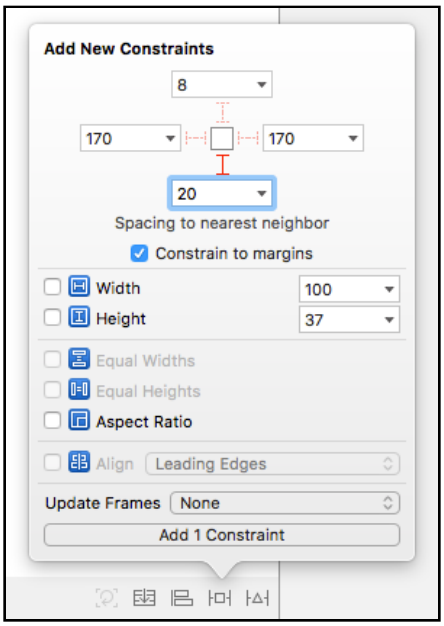
The thing to remember is this: for each view, there must always be enough constraints to define both its position and its size. The Close button already knows its size – you typed this into the Size inspector earlier – but for its position there is only a constraint for the X-coordinate (the alignment in the horizontal direction). You also need to add a constraint for the Y-coordinate.

As you've noticed, there are different types of constraints - there are alignment constraints and spacing constraints, like the ones you added via the Add New Constraints button.

➤ With the **Close** button still selected, click on the **Add New Constraints** button.

You want the Close button to always sit at a distance of 20 points from the bottom of the screen.

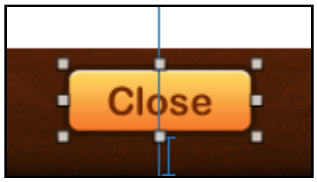
➤ In the **Add New Constraints** menu, in the **Spacing to nearest neighbor** section, set the bottom spacing to **20** and make sure that the I-beam above the text box is enabled.



The red I-beams decide the sides that are pinned down

➤ Click **Add 1 Constraint** to finish.

The red constraints will now turn blue, meaning that everything is OK:



The constraints on the Close button are valid

If at this point you don't see blue bars but orange ones, then something's still wrong with your Auto Layout constraints:

The views are not positioned according to the constraints

This happens when the constraints are valid (otherwise the bars would be red) but the view is not in the right place in the scene. The dashed orange box off to the side is where Auto Layout has calculated the view should be, based on the constraints you have given it.

To fix this issue, select the **Close** button again and click the **Update Frames** button at the bottom of the Interface Builder canvas.

The Update Frames button

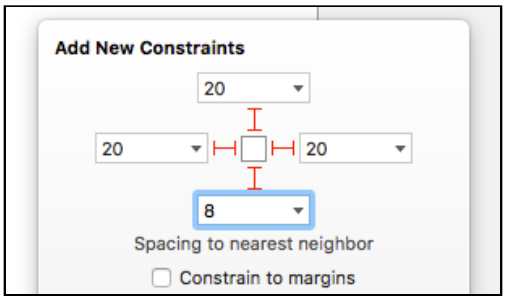
You can also use the **Editor → Resolve Auto Layout Issues → Update Frames** item from the menu bar.

The Close button should now always be perfectly centered, regardless of the device screen size.

> **Note:** What happens if you don't add any constraints to your views? In that case, Xcode will automatically add constraints when it builds the app. That is why you didn't need to bother with any of this before.
>
> However, these default constraints may not always do what you want. For example, they will not automatically resize your views to accommodate larger (or smaller) screens. If you want that to happen, then it's up to you to add your own constraints. (Afterall, Auto Layout can't read your mind!)
>
> As soon as you add just one constraint to a view, Xcode will no longer add any other automatic constraints to that view. From then on you're responsible for adding enough constraints so that UIKit always knows what the position and size of the view will be.

There is one thing left to fix in the About screen and that is the web view.

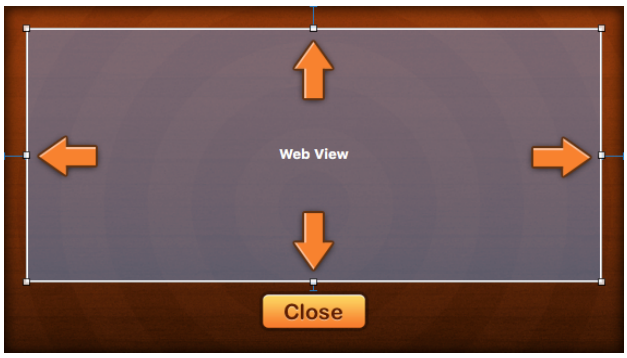
➤ Select the **Web View** and open the **Add New Constraints** menu. First, make sure **Constrain to margins** is unchecked. Then click all four I-beam icons so they become solid red and set their spacing to 20 points, except the bottom one which should be 8 points:



Creating the constraints for the web view

➤ Finish by clicking **Add 4 Constraints**.

There are now four constraints on the web view - indicated by the blue bars on each side:



The four constraints on the web view

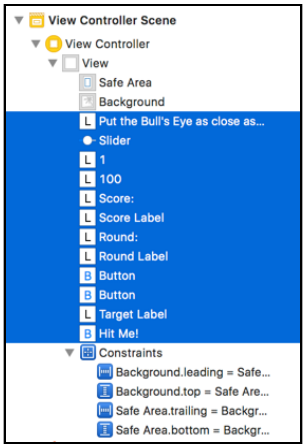
Three of these pin the web view to the main view, so that it always resizes along with it, and one connects it to the Close button. This is enough to determine the size and position of the web view in any scenario.

Fix the rest of the main scene

Back to the main game scene, which still needs some work.

The game looks a bit lopsided now on bigger screens. You will fix that by placing all the labels, buttons, and the slider into a new "container" view. Using Auto Layout, you'll center that container view in the screen, regardless of how big the screen is.

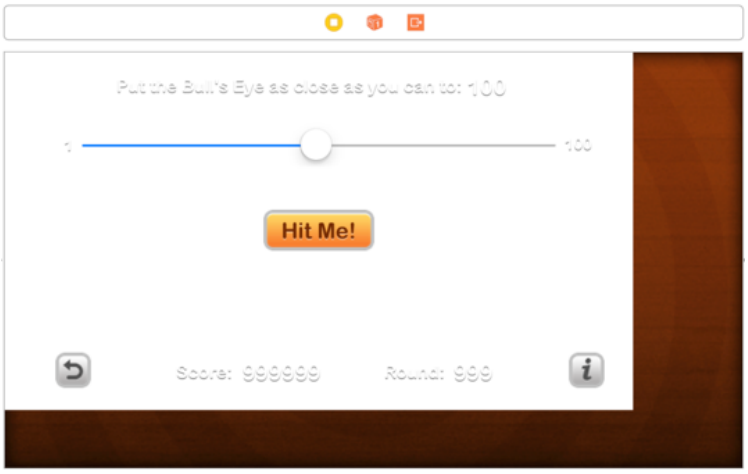
➤ Select all the labels, buttons, and the slider. You can hold down ⌘ and click them individually, but an easier method is to go to the **Document Outline**, click on the first view (for me that is the "Put the Bull's Eye as close as you can to:" label), then hold down Shift and click on the last view (in my case the Hit Me! button):



Selecting the views from the Document Outline

You should have selected everything but the background image view.

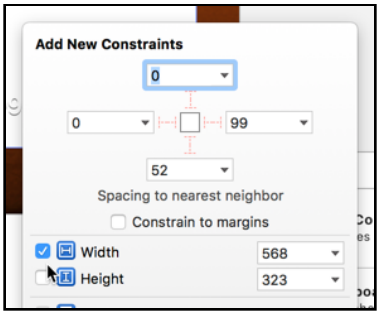
➤ From Xcode's menu bar, choose **Editor → Embed In → View**. This places the selected views inside a new container view:



The views are embedded in a new container view

This new view is completely white, which is not what you want eventually, but it does make it easier to add the constraints.

➤ Select the newly added **container view** and open the **Add New Constraints** menu. Check the boxes for **Width** and **Height** in order to make constraints for them and leave the width and height at the values specified by Interface Builder. Click **Add 2 Constraints** to finish.



Pinning the width and height of the container view

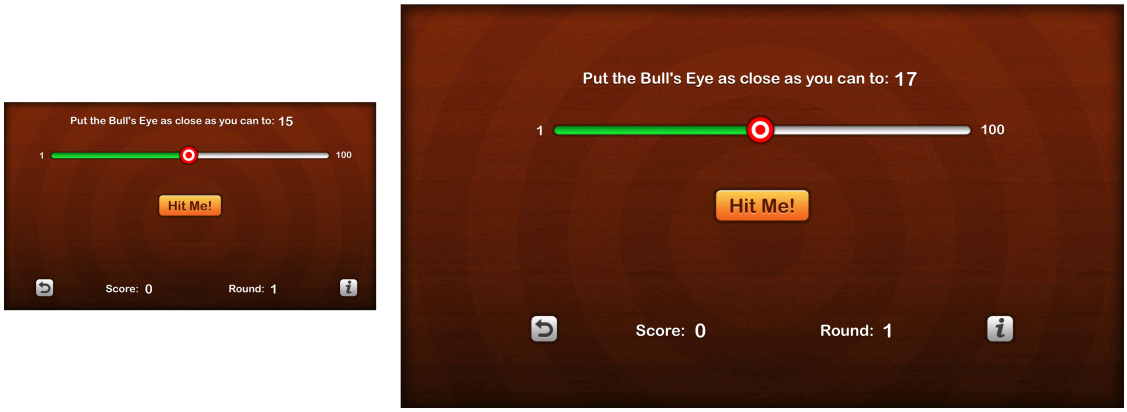
Interface Builder now draws several bars around the view that represent the Width and Height constraints that you just made, but they are red. Don't panic! It only means there are not enough constraints yet. No problem, you'll add the missing constraints next.

➤ With the container view still selected, open the **Align menu**. Check the **Horizontally in Container** and **Vertically in Container** options. Click **Add 2 Constraints**.

All the Auto Layout bars should be blue now and the view is perfectly centered.

➤ Finally, change the **Background** color of the container view to **Clear Color** (in other words, 100% transparent).

You now have a layout that works correctly on any iPhone display! Try it out:



The game running on 4-inch and 5.5-inch iPhones

Auto Layout may take a while to get used to. Adding constraints in order to position UI elements is a little less obvious than just dragging them into place.

But this also buys you a lot of power and flexibility, which you need when you're dealing with devices that have different screen sizes.

You'll learn more about Auto Layout in the other parts of *The iOS Apprentice*.

> **Exercise:** As you try the game on different devices, you might notice something - the controls for the game are always centered on screen, but they do not take up the whole area of the screen on bigger devices! This is because you set the container view for the controls to be a specific size. If you want the controls to change position and size depending on how much screen space is available, then you have to remove the container view (or set it to resize depending on screen size) and then set up the necessary autolayout constraints for each control separately.
>
> Are you up to the challenge of doing this on your own?

Crossfade

There's one final bit of knowledge I want to impart before calling the game complete - Core Animation. This technology makes it very easy to create really sweet animations, with just a few lines of code, in your apps. Adding subtle animations (with emphasis on subtle!) can make your app a delight to use.

You will add a simple crossfade after the Start Over button is pressed, so the transition back to round one won't seem so abrupt.

➤ In **ViewController.swift**, add the following line at the top, right below the other import:

```
import QuartzCore
```

Core Animation lives in its own framework, QuartzCore. With the `import` statement you tell the compiler that you want to use the objects from this framework.

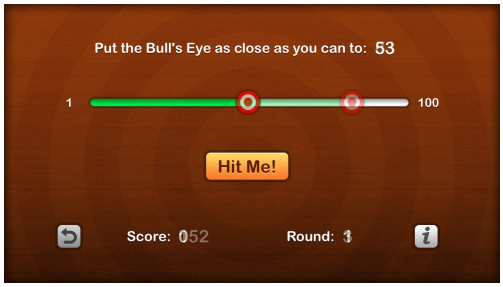
➤ Change `startNewGame()` to:

```
@IBAction func startNewGame() {
  ...
  startNewRound()
  // Add the following lines
  let transition = CATransition()
  transition.type = kCATransitionFade
  transition.duration = 1
  transition.timingFunction = CAMediaTimingFunction(name:
                                        kCAMediaTimingFunctionEaseOut)
  view.layer.add(transition, forKey: nil)
}
```

Everything after the comment telling you to add the following lines, all the `CATransition` stuff, is new.

I'm not going to go into too much detail here. Suffice it to say you're setting up an animation that crossfades from what is currently on the screen to the changes you're making in `startNewRound()` – reset the slider to center position and reset the values of the labels.

➤ Run the app and move the slider so that it is no longer in the center. Press the Start Over button and you should see a subtle crossfade animation.

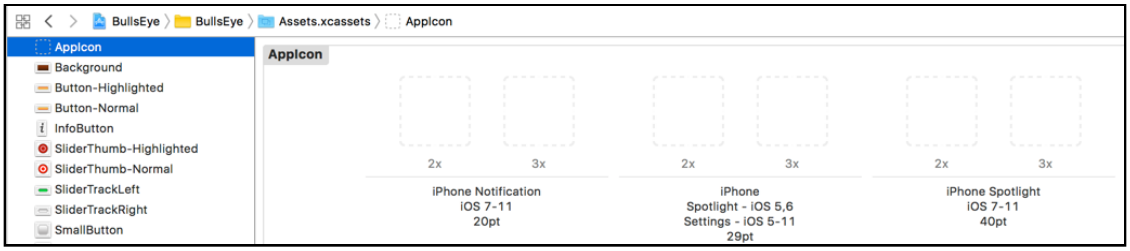


The screen crossfades between the old and new states

The icon

You're almost done with the app, but there are still a few loose ends to tie up. You may have noticed that the app has a really boring white icon. That won't do!

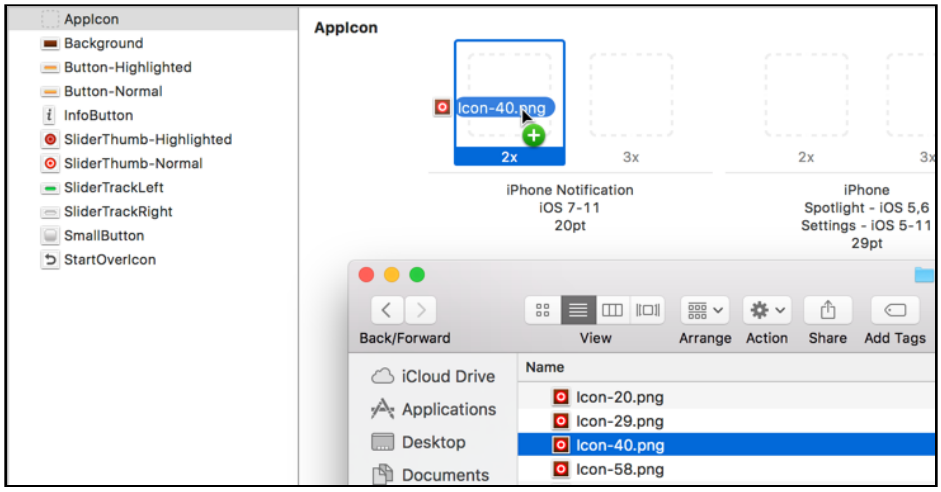
➤ Open the asset catalog (**Assets.xcassets**) and select **AppIcon**:



The AppIcon group in the asset catalog

This has ten groups for the different types of icons the app needs.

➤ In Finder, open the **Icon** folder from the resources. Drag the **Icon-40.png** file into the first slot, **iPhone Notification 20pt**:



Dragging the icon into the asset catalog

You may be wondering why you're dragging the Icon-40.png file and not the Icon-20.png into the slot for 20pt. Notice that this slot says **2x**, which means it's for Retina devices and on Retina screens one point counts as two pixels. So, 20pt = 40px. And the 40 in the icon name is for the size of the icon in pixels. Makes sense?

➤ Drag the **Icon-60.png** file into the **3x** slot next to it. This is for the iPhone Plus devices with their 3x resolution.

➤ For **iPhone Spotlight & Settings 29pt**, drag the **Icon-58.png** file into the 2x slot and **Icon-87.png** into the 3x slot. (What, you don't know your times table for 29?)

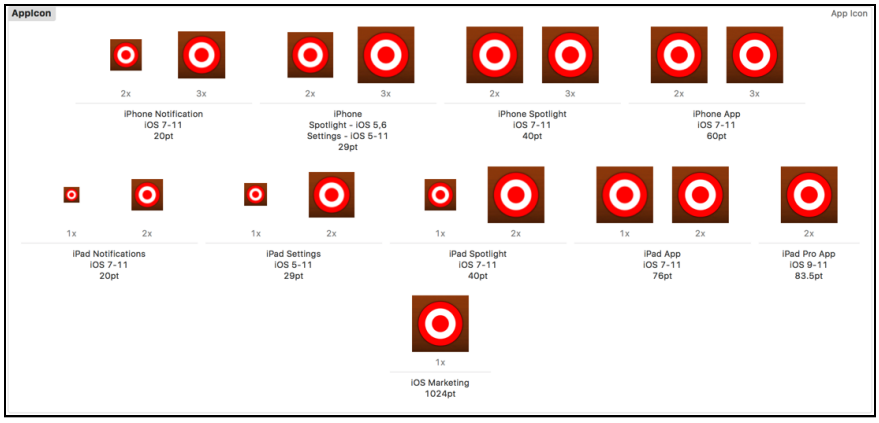
➤ For **iPhone Spotlight 40pt**, drag the **Icon-80.png** file into the 2x slot and **Icon-120.png** into the 3x slot.

➤ For **iPhone App 60pt**, drag the **Icon-120.png** file into the 2x slot and **Icon-180.png** into the 3x slot.

That's four icons in two different sizes. Phew!

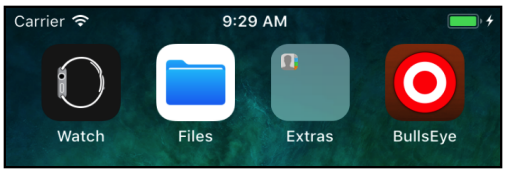
The other AppIcon groups are mostly for the iPad.

➤ Drag the specific icons (based on size) into the proper slots for iPad. Notice that the iPad icons need to be supplied in 1x as well as 2x sizes (but not 3x). You may need to do some mental arithmetic here to figure out which icon goes into which slot!



The full set of icons for the app

➤ Run the app and close it. You'll see that the icon has changed on the Simulator's springboard. If not, remove the app from the Simulator and try again (sometimes the Simulator keeps using the old icon and re-installing the app will fix this).



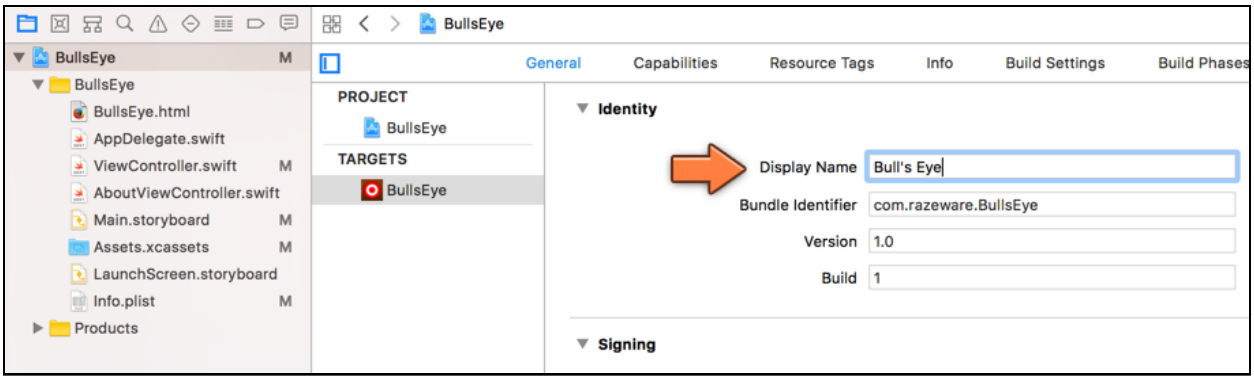
The icon on the Simulator's springboard

Display name

One last thing. You named the project **BullsEye** and that is the name that shows up under the icon. However, I'd prefer to spell it "**Bull's Eye**".

There is only limited space under the icon and for apps with longer names you have to get creative to make the name fit. For this game, however, there is enough room to add the space and the apostrophe.

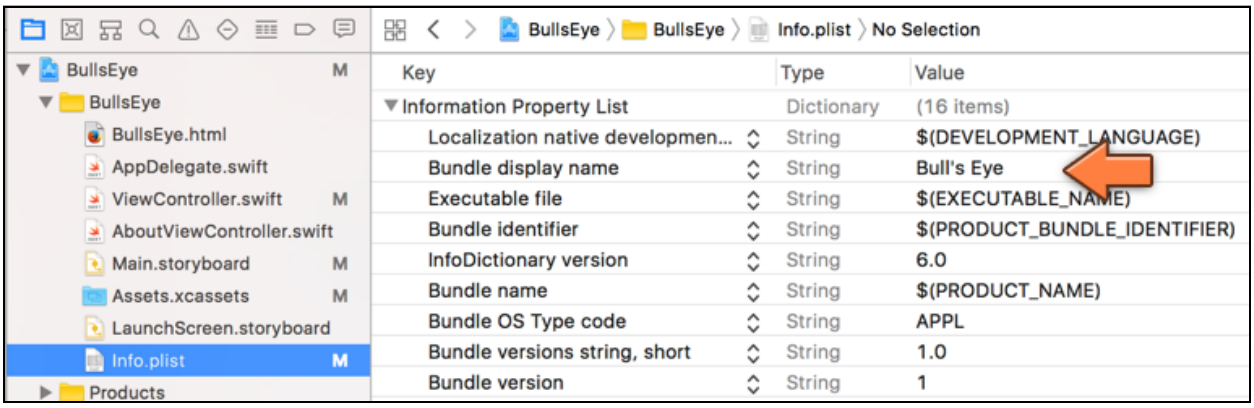
➤ Go to the **Project Settings** screen. The very first option is **Display Name**. Change this to **Bull's Eye**.



Changing the display name of the app

Like many of the project's settings you can also find the display name in the app's Info.plist file. Let's have a look.

➤ From the **Project navigator**, select **Info.plist**.



The display name of the app in Info.plist

The row **Bundle display name** contains the new name you've just entered.

> **Note:** If **Bundle display name** is not present, the app will use the value from the field **Bundle name**. That has the special value "$(PRODUCT_NAME)", meaning Xcode will automatically put the project name, BullsEye, in this field when it adds the Info.plist to the application bundle. By providing a **Bundle display name** you can override this default name and give the app any name you want.

➤ Run the app and quit it to see the new name under the icon.

The bundle display name setting changes the name under the icon

Awesome, that completes your very first app!

You can find the project files for the finished app under **08 - The Final App** in the Source Code folder.

Run on device

So far, you've run the app on the Simulator. That's nice and all but probably not why you're learning iOS development. You want to make apps that run on real iPhones and iPads! There's hardly a thing more exciting than running an app that you made on your own phone. And, of course, to show off the fruits of your labor to other people!

Don't get me wrong: developing your apps on the Simulator works very well. When developing, I spend most of my time with the Simulator and only test the app on my iPhone every so often.

However, you do need to run your creations on a real device in order to test them properly. There are some things the Simulator simply cannot do. If your app needs the iPhone's accelerometer, for example, you have no choice but to test that functionality on an actual device. Don't sit there and shake your Mac!

Until a few years back, you needed a paid Developer Program account to run apps on your iPhone. Since Xcode 7, however, you can do it for free. All you need is an Apple ID. And the latest Xcode makes it easier than ever before.

Configure your device for development

➤ Connect your iPhone, iPod touch, or iPad to your Mac using a USB cable.

➤ From the Xcode menu bar select **Window → Devices and Simulators** to open the Devices and Simulators window.

Mine looks like this (I'm using an iPhone 6s):

The Devices and Simulators window

On the left is a list of devices that are currently connected to my Mac and which can be used for development.

➤ Click your device name to select it.

If this is the first time you're using the device with Xcode, the Devices window will say something like, "iPhone is not paired with your computer." To pair the device with Xcode, you need to unlock the device first (hold the home button). After unlocking, an alert will pop up on the device asking you to trust the computer you're trying to pair with. Tap on **Trust** to continue.

Xcode will now refresh the page and let you use the device for development. Give it a few minutes (see the progress bar in the main Xcode window). If it takes too long, you may need to unplug the device and plug it back in.

At this point it's possible you may get the error message, "An error was encountered while enabling development on this device." You'll need to unplug the device and reboot it. Make sure to restart Xcode before you reconnect the device.

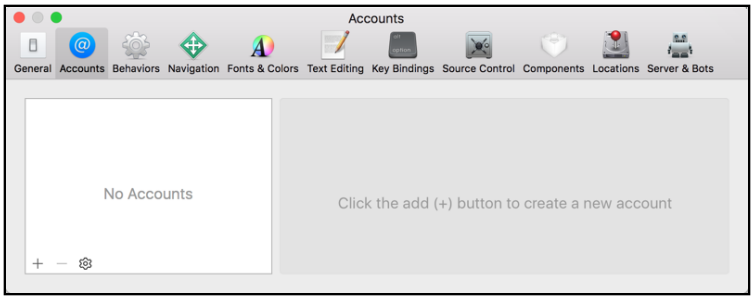
Also, note the checkbox which says **Connect via network**? That checkbox (gasp!) allows you to run and debug code on your iPhone over WiFi! Yes, that's new in Xcode 9. (I still prefer to do my debugging with my phone connected via USB cable since the last time I checked, the over network debugging was very slow. But your mileage may vary - so give it a try...)

Cool, that is the device sorted.

Add your developer account to Xcode

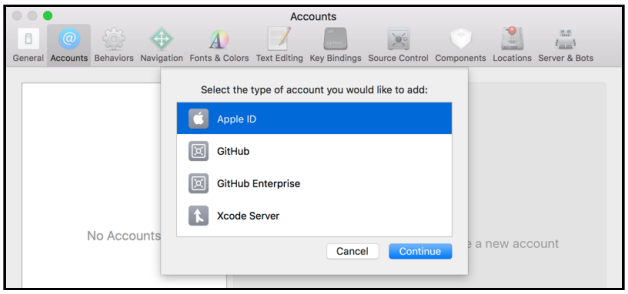
The next step is setting up your Apple ID with Xcode. It's OK to use the same Apple ID that you're already using with iTunes and your iPhone, but if you run a business, you might want to create a new Apple ID to keep things separate. Of course, if you've already registered for a paid Developer Program account, you should use that Apple ID.

➤ Open the **Accounts** pane in the Xcode Preferences window:



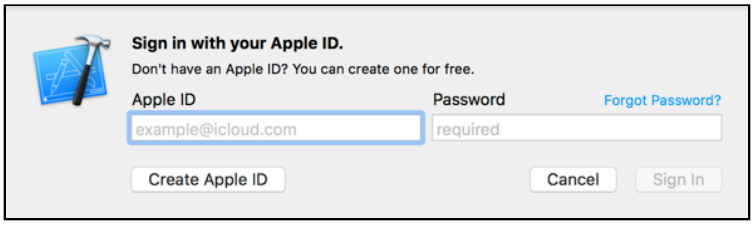
The Accounts preferences

➤ Click the + button at the bottom and select **Add Apple ID** from the list of options.



Xcode Account Type selection

Xcode will ask for your Apple ID:



Adding your Apple ID to Xcode

➤ Type your Apple ID username and password and click **Sign In**.

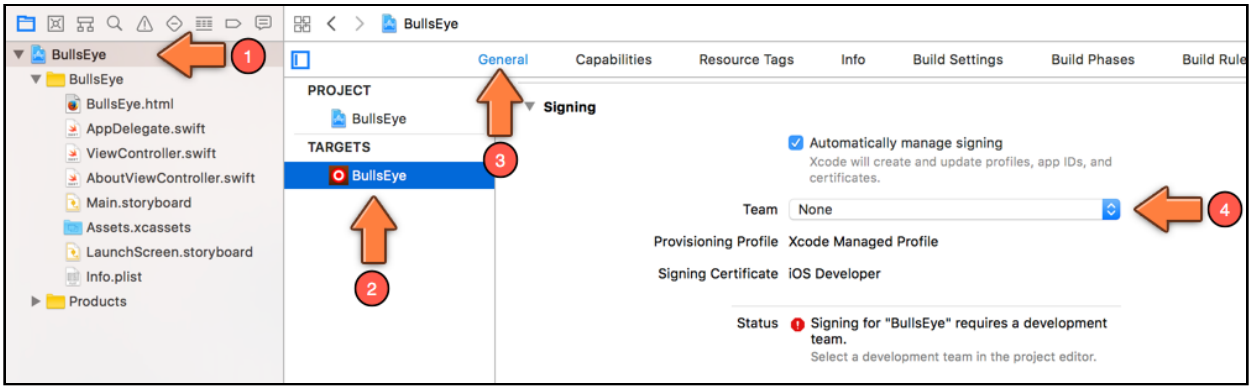
Xcode verifies your account details and adds it to the stored list of accounts.

> **Note:** It's possible that Xcode is unable to use the Apple ID your provided - for example, if it has been used with a Developer Program account in the past that is now expired. The simplest solution is to make a new Apple ID. It's free and only takes a few minutes. appleid.apple.com

You still need to tell Xcode to use this account when building your app.

Code signing

➤ Go to the **Project Settings** screen for your app target. In the **General** tab go to the **Signing** section.



The Signing options in the Project Settings screen

In order to allow Xcode to put an app on your iPhone, the app must be *digitally signed* with your **Development Certificate**. A *certificate* is an electronic document that identifies you as an iOS application developer and is valid only for a specific amount of time.

Apps that you want to submit to the App Store must be signed with another certificate, the **Distribution Certificate**. To use the distribution certificate you must be a member of the paid Developer Program, but using the development certificate is free.

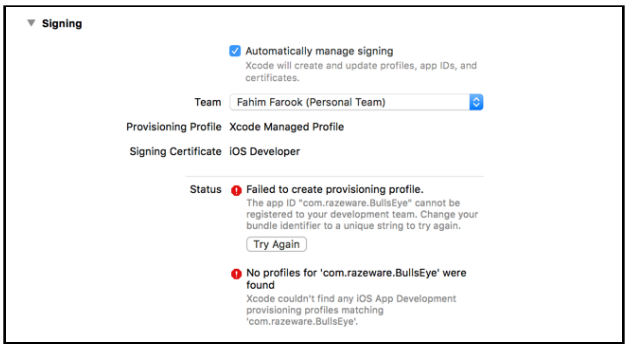
In addition to a valid certificate, you also need a **Provisioning Profile** for each app you make. Xcode uses this profile to sign the app for use on your particular device (or devices). The specifics don't really matter, just know that you need a provisioning profile or the app won't go on your device.

Making the certificates and provisioning profiles used to be a really frustrating and error-prone process. Fortunately, those days are over: Xcode now makes it really easy. When the **Automatically manage signing** option is enabled, Xcode will take care of all this business with certificates and provisioning profiles and you don't have to worry about a thing.

➤ Click on **Team** to select your Apple ID.

Xcode will now automatically register your device with your account, create a new Development Certificate, and download and install the Provisioning Profile on your device. These are all steps you had to do by hand in the past, but now Xcode takes care of all that.

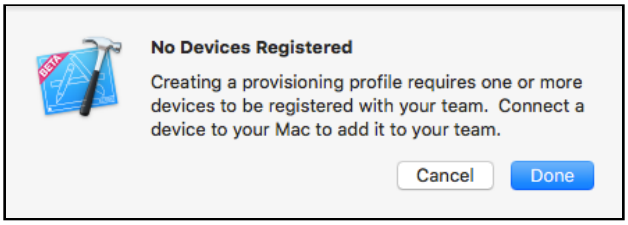
You could get some signing errors like these:



Signing/team set up errors

The app's Bundle Identifier – or App ID as it's called here – must be unique. If another app is already using that identifier, then you cannot use it anymore. That's why you're supposed to start the Bundle ID with your own domain name. The fix is easy: change the Bundle Identifier field to something else and try again.

It's also possible you get this error (or something similar):



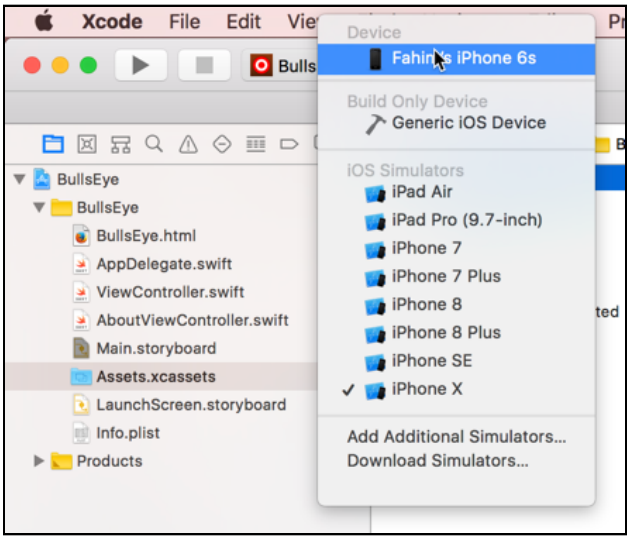
No devices registered

Xcode must know about the device that you're going to run the app on. That's why I asked you to connect your device first. Double-check that your iPhone or iPad is still connected to your Mac and that it is listed in the Devices window.

Run on device

If everything goes smoothly, go back to Xcode's main window and click on the box in the toolbar to change where you will run the app. The name of your device should be in that list somewhere.

On my system it looks like this:



Setting the active device

You're all set and ready to go!

➤ Press **Run** to launch the app.

At this point you may get a popup with the question "codesign wants to sign using key ... in your keychain". If so, answer with **Always Allow**. This is Xcode trying to use the new Development Certificate you just created - you just need to give it permission first.

Does the app work? Awesome! If not, read on...

When things go wrong...

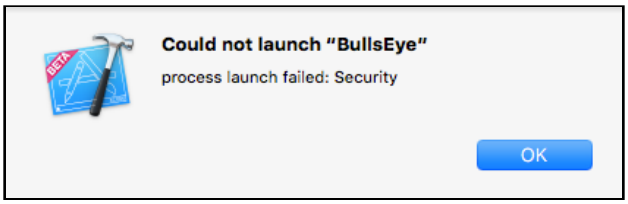
There are a few things that can go wrong when you try to put the app on your device, especially if you've never done this before, so don't panic if you run into problems.

The device is not connected

Make sure your iPhone, iPod touch, or iPad is connected to your Mac. The device must be listed in Xcode's Devices window and there should not be a yellow warning icon.

The device does not trust you

You might get this warning:



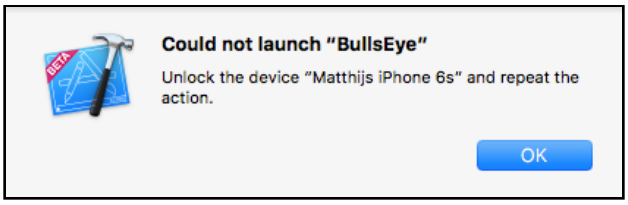
Quick, call security!

On the device itself there will be a popup with the text, "Untrusted Developer. Your device management settings do not allow using apps from developer ...".

If this happens, open the Settings app on the device and go to **General** → **Profile**. Your Apple ID should be listed in that screen. Tap it, followed by the Trust button. Then try running the app again.

The device is locked

If your phone locks itself with a passcode after a few minutes, you might get this warning:



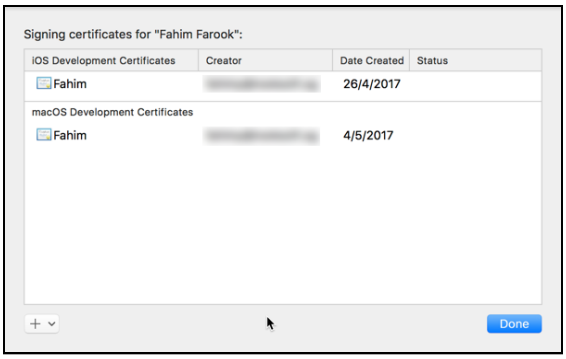
The app won't run if the device is locked

Simply unlock your device (hold the home button or type in the 4-digit passcode) and press Run again.

Signing certificates

If you're curious about these certificates, then open the **Preferences** window and go to the **Accounts** tab. Select your account and click the **Manage Certificates...** button in the bottom-right corner.

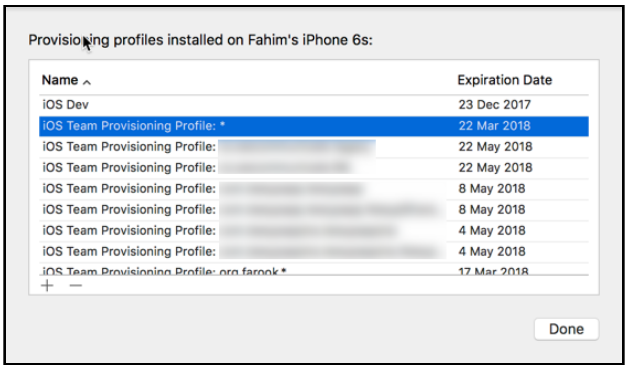
This brings up another panel, listing your signing certificates:



The Manage Certificates panel

When you're done, close the panel and go to the **Devices and Simulators** window.

You can see the provisioning profiles that are installed on your device by right-clicking the device name and choosing **Show Provisioning Profiles**:



The provisioning profiles on your device

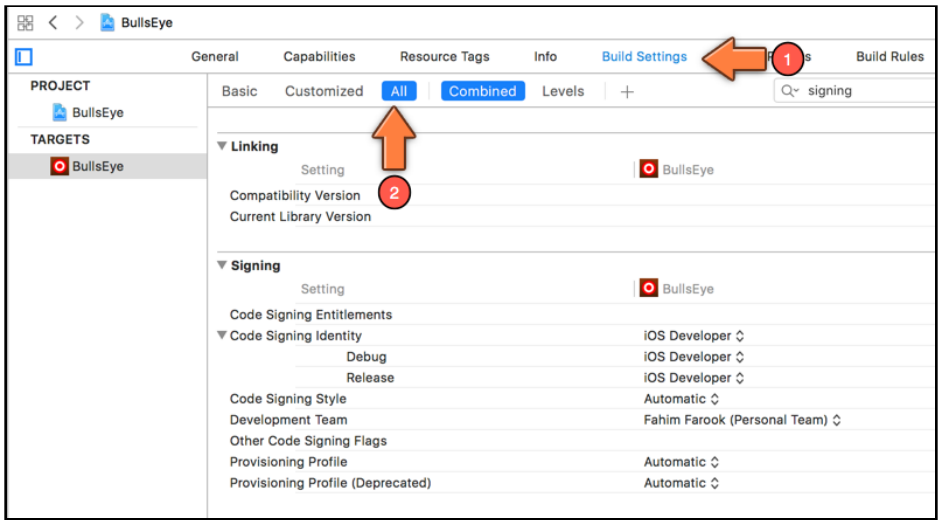
The "iOS Team Provisioning Profile: *" is the thing that allows you to run the app on your device. (By the way, they call it the "team" profile because often there is more than one developer working on an app and they can all share the same profile.)

You can have more than one certificate and provisioning profile installed. This is useful if you're on multiple development teams or if you prefer to manage the provisioning profiles for different apps by hand.

To see how Xcode chooses which profile and certificate to sign your app with, go to the **Project Settings** screen and switch to the **Build Settings** tab. There are a lot of settings

in this list, so filter them by typing **signing** in the search box. (Also make sure **All** is selected, not Basic.)

The screen will look something like this:



The Code Signing settings

Under **Code Signing Identity** it says **iOS Developer**. This is the certificate that Xcode uses to sign the app. If you click on that line, you can choose another certificate. Under **Provisioning Profile** you can change the active profile. Most of the time you won't need to change these settings, but at least you know where to find them now.

And that concludes everything you need to know about running your app on an actual device.

The end... or the beginning?

It has been a bit of a journey to get to this point – if you're new to programming, you've had to get a lot of new concepts into your head. I hope your brain didn't explode!

At least you should have gotten some insight into what it takes to develop an app.

I don't expect you to understand exactly everything that you did, especially not the parts that involved writing Swift code. It is perfectly fine if you didn't, as long as you're enjoying yourself and you sort of get the basic concepts of objects, methods and variables.

If you were able to follow along and do the exercises, you're in good shape!

I encourage you to play around with the code a bit more. The best way to learn programming is to do it, and that includes making mistakes and messing things up. I hereby grant you full permission to do so! Maybe you can add some cool new features to the game (and if you do, please let me know).

In the Source Code folder for this book you can find the complete source code for the *Bull's Eye* app. If you're still unclear about some of what you did, it might be a good idea to look at this cleaned up source code.

If you're interested in how I made the graphics, then take a peek at the Photoshop files in the Resources folder. The wood background texture was made by Atle Mo from subtlepatterns.com.

If you're feeling exhausted after all that coding, pour yourself a drink and put your feet up for a bit. You've earned it! On the other hand, if you just can't wait to get to grips with more code, let's move on to our next app!