Up to date for iOS 13, Xcode 11 & Swift 5.1

SwiftUI by Tutorials

FIRST EDITION

Declarative App Development on the Apple Ecosystem

By the **raywenderlich Tutorial Team**

Antonio **Bello**, Phil **Laszkowicz**, Bill **Morefield** & Audrey **Tam**

SwiftUI by Tutorials

By Antonio Bello, Phil Łaszkowicz, Bill Morefield & Audrey Tam

Notice of Rights

Notice of Liability

Trademarks

About the Authors

Antonio Bello is an author of this book. Antonio has spent most of his life writing code, and he's gained a lot of experience in several languages and technologies. A few years ago he fell in love with iOS development, and that's what he mostly works on since then, although he's always open for challenges and for playing with new toys. He believes that reputation is the most important skill in his job, and that "it cannot be done" actually means "it can be done, but it's not economically convenient." When he's not working, he's probably playing drums or making songs in his small, but well fitted, home recording studio.

Phil Łaszkowicz is an author of this book. Phil's been delivering large-scale software solutions for many years, as well as working with startups as a board member, mentor, and coach. He's worked with neural networks for over a decade, and enjoys combining deep learning with intuitive and elegant user experiences across mobile and web. In his spare time he writes music, drinks coffee at a professional level, and can be found scaling cliff walls, composing music, sea kayaking, or taking part in competitive archery.

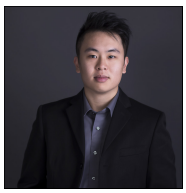
Bill Morefield is an author of this book. Bill has spent most of his professional life writing code. At some point he has worked in almost every language other than COBOL. He bought his first Apple computer to learn to program for the iPhone and got hooked on the platform. He manages the web and mobile development team for a college in Tennessee, where he still gets to write code. When not attached to a keyboard he enjoys hiking and photography.

Audrey Tam is an author of this book. As a retired computer science academic, she's a technology generalist with expertise in translating new knowledge into learning materials. Audrey now teaches short courses in iOS app development to non-programmers, and attends nearly all Melbourne Cocoaheads monthly meetings. She also enjoys long train journeys, knitting, and trekking in the Aussie wilderness.

About the Editors

Pablo Mateo is the final pass editor for this book. He is Technical Lead at Banco Santander, and was also founder and CTO of a Technology Development company in Madrid. His expertise is focused on web and mobile app development, although he first started as a Creative Art Director. He has been for many years the Main Professor of the iOS and Android Mobile Development Masters Degree at a well-known technology school in Madrid (CICE). He is currently specializing in Artificial Intelligence & Machine-Learning.

Morten Faarkrog is a tech editor for this book. Morten is Technical Director at a full-service digital agency in Copenhagen, Denmark. He has a background as an iOS developer and loves tinkering with new innovative technologies—one of which you'll shortly be diving into. He an advocate of trying new things and taking calculated risks, and thinks you should be, too!



Kelvin Lau is a tech editor for this book. Kelvin is a senior mobile engineer at Instacart. He loves space related stuff, and wishes to head up there someday. Outside of programming work, he's an aspiring entrepreneur and musician. You can find him on Twitter: @kelvinlauKL

About the Artist

Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Dedications

"To Magdalena, Andrea and Alex, for their support and patience, watching me tapping on the keyboard all day long."

— Antonio Bello

"To Isabella for being the best inspiration when distractions are too easy to find, and the best distraction for when work is too easy to lose myself in."

— Phil Laszkowicz

"To my parents for buying me that first computer when it was a lot weirder idea than it is now. To them and rest of my family for putting up with all those questions as a child."

— Bill Morefield

"To my parents and teachers, who set me on the path thatled me to the here and now."

— Audrey Tam

Table of Contents: Overview

Table of Contents: Extended

Book License

By purchasing *SwiftUI by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *SwiftUI by Tutorials* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *SwiftUI by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *SwiftUI by Tutorials*, available at www.raywenderlich.com".
- The source code included in *SwiftUI by Tutorials* is for your personal use only. You are NOT allowed to distribute or sell the source code in *SwiftUI by Tutorials* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action or contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

About This Book Sample

SwiftUI is a new paradigm in Apple-related development. In 2014, after years of programming apps with Objective-C, Apple surprised the world with a new open-source language: **Swift**..

SwiftUI's introduction in 2019 creates another opportunity for a paradigm shift in the industry. After years using UIKit and AppKit to create user interfaces, SwiftUI presents a fresh, new way to create UI for your apps.

This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn how to build user interfaces with SwiftUI and how to integrate SwiftUI into their existing apps.

We are pleased to offer you this sample from the full *SwiftUI by Tutorials* book that will introduce you to these concepts and give you a chance to practice them in our hands-on By Tutorials style.

This sample includes:

1. **Introduction**: A brief history about Swift and SwiftUI, and an explanation of how to get the most out of this book.
2. **Getting Started**: Learn how to use the Xcode canvas to create your UI side-by-side with its code. You'll create a reusable view for the sliders in your app, see how `@State` variables work and how to use them to update your UI whenever a state value changes. And finally, you'll learn how to present an alert to the user.

You can get the the complete *SwiftUI by Tutorials* book here:

- https://store.raywenderlich.com/products/swiftui-by-tutorials.

Enjoy!

The *SwiftUI by Tutorials* Team

B Book Source Code & Forums

If you bought the digital edition

The digital edition of this book comes with the source code for the starter and completed projects for each chapter. These resources are included with the digital edition you downloaded from store.raywenderlich.com.

The digital edition of this book also comes with free access to any future updates we may make to the book!

The best way to get update notifications is to sign up for our monthly newsletter. This includes a list of the tutorials that came out on raywenderlich.com that month, any important news like book updates or new books, and a list of our favorite iOS development links for that month. You can sign up here:

- www.raywenderlich.com/newsletter

If you bought the print version

You can get the source code for the print edition of the book here:

https://store.raywenderlich.com/products/swift-ui-by-tutorials-source-code

Forums

We've also set up an official forum for the book at forums.raywenderlich.com. This is a great place to ask questions about the book or to submit any errors you may find.

Digital book editions

We have a digital edition of this book available in both ePUB and PDF, which can be handy if you want a soft copy to take with you, or you want to quickly search for a specific term within the book.

Buying the digital edition version of the book also has a few extra benefits: free updates each time we update the book, access to older versions of the book, and you can download the digital editions from anywhere, at anytime.

Visit our *SwiftUI by Tutorials* store page here:

- https://store.raywenderlich.com/products/swift-ui-by-tutorials.

And if you purchased the print version of this book, you're eligible to upgrade to the digital editions at a significant discount! Simply email support@razeware.com with your receipt for the physical copy and we'll get you set up with the discounted digital edition version of the book.

What You Need

To follow along with this book, you'll need the following:

- A Mac running **macOS Mojave** (10.14.4) or later. Optionally, you can use **macOS Catalina** (10.15), which is still in Beta. You'll need an Apple Developers account in order to install it.
- **Xcode 11 or later.** Xcode is the main development tool for iOS. You'll need Xcode 11 or later to make use of SwiftUI. You can download the latest version of Xcode from Apple's developer site here: apple.co/2asi58y.

> **Note**: You can use the same link to install the beta version of macOS Catalina. Bear in mind that because it is still in beta, you might find some bugs and unexpected errors while following along the tutorials if you are using the beta version or macOS Mojave. SwiftUI is a new technology that still needs some polish, so don't expect perfect behavior in every situation. Use the book's forum to ask any questions you might have.

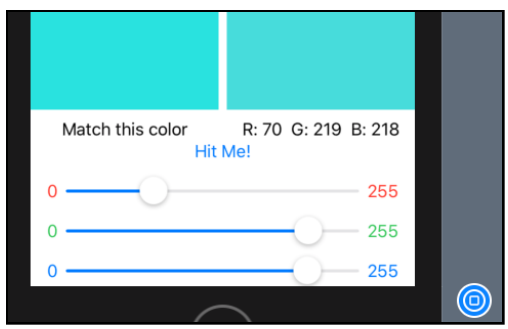
If you haven't installed the latest version of Xcode, be sure to do that before continuing with the book. The code covered in this book depends on Swift 5.1, macOS Catalina and Xcode 11 — you may get lost if you try to work with an older version.

Chapter 2: Getting Started

By Audrey Tam

SwiftUI is some of the most exciting news since Apple first announced Swift in 2014. It's an enormous step towards Apple's goal of getting everyone coding; it simplifies the basics so that you can spend more time on custom features that delight your users.

If you're reading this book, you're just as excited as I am about developing apps with this new framework. This chapter will get you comfortable with the basics of creating a SwiftUI app and (live-) previewing it in Xcode. You'll create a small color-matching game, inspired by our famous *BullsEye* app from our book *iOS Apprentice*. The goal of the app is to try and match a randomly generated color by selecting colors from the RGB color space:



Playing the game

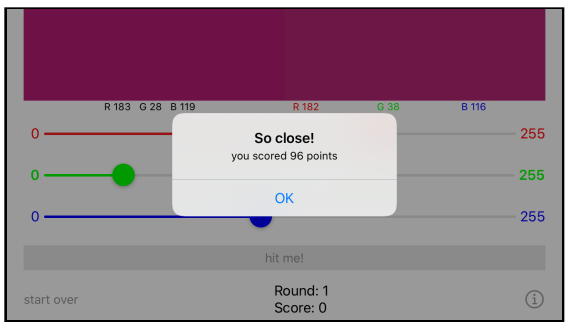
In this chapter, you will:

- Learn how to use the Xcode canvas to create your UI side-by-side with its code, and see how they stay in sync—a change to one side always updates the other side.
- Create a reusable view for the sliders seen in the image.
- Learn about `@State` variables and use them to update your UI whenever a state value changes.
- Present an alert to show the user's score.

Time to get started!

Getting started

Open the **RGBullsEye** starter project from the chapter materials, and build and run:

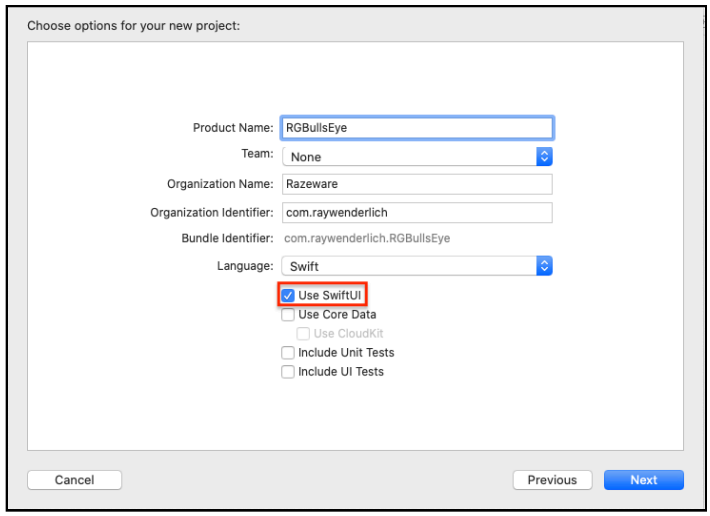


UIKit RGBullsEye starter app

This app displays a target color with randomly generated red, green and blue values. The user moves the sliders to make the left color block match the right side. You're about to create a SwiftUI app that does the exact same thing, but more swiftly!

Creating a new SwiftUI project

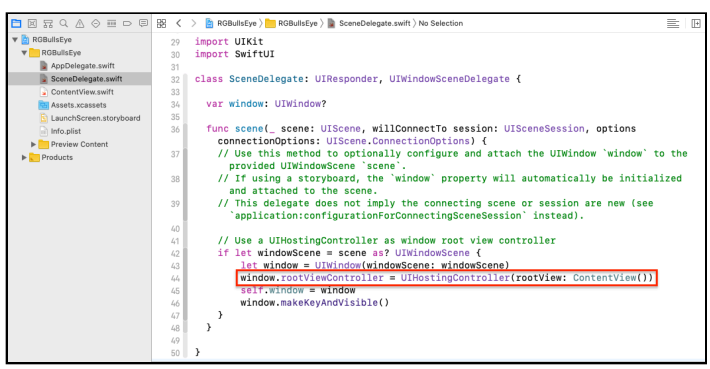
To start, create a new Xcode project (**Shift-Command-N**), select **iOS ▸ Single View App**, name the project **RGBullsEye**, then check the **Use SwiftUI** checkbox:



Use SwiftUI checkbox

Save your project somewhere *outside* the **RGBullsEye-Starter** folder.

In the project navigator, open the **RGBullsEye** group to see what you got: the **AppDelegate.swift**, which you may be used to seeing, is now split into **AppDelegate.swift** *and* **SceneDelegate.swift**. The latter has the `window`:



SceneDelegate.swift

`SceneDelegate` itself isn't specific to SwiftUI, but this line is:

```
window.rootViewController = UIHostingController(rootView:
ContentView())
```

`UIHostingController` creates a view controller for the SwiftUI view `ContentView`.

> **Note**: `UIHostingController` enables you to *integrate* SwiftUI views into an existing app. You'll learn how in Chapter 4, "Integrating SwiftUI."

When the app starts, `window` displays an instance of `ContentView`, which is defined in **ContentView.swift**. It's a `struct` that conforms to the `View` protocol:

```
struct ContentView: View {
  var body: some View {
    Text("Hello World")
  }
}
```

This is SwiftUI declaring that the `body` of `ContentView` contains a `Text` view that displays **Hello World**.

Previewing your ContentView

Down in the `DEBUG` block, `ContentView_Previews` contains a view that contains an instance of `ContentView`:

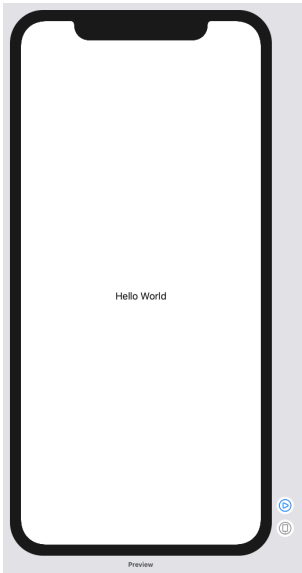
```
#if DEBUG
struct ContentView_Previews : PreviewProvider {
  static var previews: some View {
    ContentView()
  }
}
#endif
```

This is where you can specify sample data for the preview, and you can compare different font sizes and color schemes. But where *is* the preview?

There's a big blank space next to the code, with this at the top:

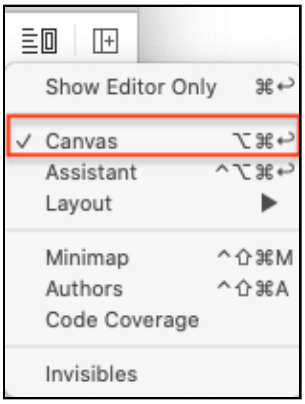
Preview Resume button

Click **Resume**, and wait a while, to see the preview:



Hello World preview

> **Note**: If you don't see the **Resume** button, click the **Editor Options** button, and select **Canvas**:



Editor options

If you still don't see the **Resume** button, make sure you're running macOS Catalina (10.15).

> **Note**: Instead of clicking the **Resume** button, you can use the very useful keyboard shortcut **Option-Command-P**. It works even when the **Resume** button isn't displayed immediately after you change something in the view.

Previewing in landscape

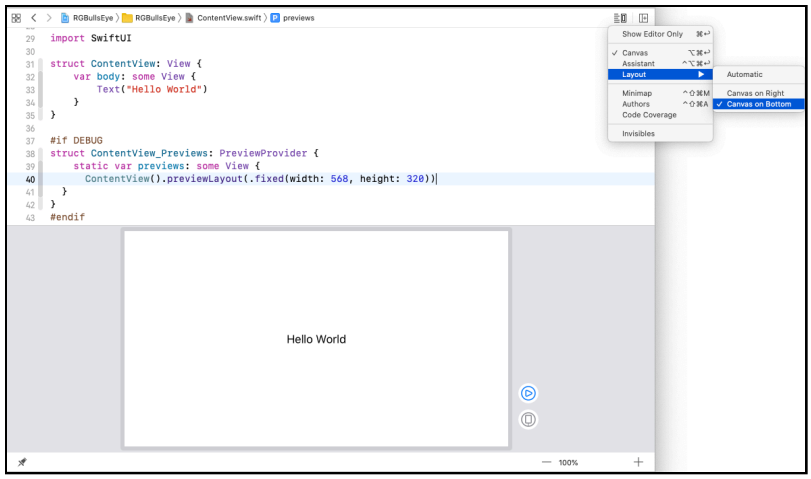
RGBullsEye looks best in landscape orientation. However, at the time of writing, Xcode 11 beta doesn't provide an easy way to preview in landscape orientation. For now, you have to specify fixed width and height values—inside the static `previews` property, add a `previewLayout` modifier to `ContentView()`:

```
ContentView().previewLayout(.fixed(width: 568, height: 320))
```

These values display an iPhone SE-sized window in landscape orientation.

To see a list of dimensions for other iPhone models, go to, see this article, *"The Ultimate Guide To iPhone Resolutions,"* which you can access here: bit.ly/29Ce3Ip.

> **Note**: To save some display space here, I set the editor layout to **Canvas on Bottom**.



Preview iPhone SE in landscape

Creating your UI

Your SwiftUI app doesn't have a storyboard or a view controller—**ContentView.swift** takes over their jobs. You can use any combination of code and drag-from-object-library to create your UI, and you can perform storyboard-like actions directly in your code! Best of all, everything stays in sync all the time!

SwiftUI is **declarative**: you declare how you want the UI to look, and SwiftUI converts your declarations into efficient code that gets the job done. Apple encourages you to create as many views as you need to keep your code easy to read. Reusable parameterized views are especially recommended—it's just like extracting code into a function, and you'll create one later in this chapter.

For this chapter, you'll mostly use the canvas, similar to how you'd layout your UI in Interface Builder (IB).

Some SwiftUI vocabulary

Before you dive into creating your views, there's a little vocabulary you must learn.

- **Canvas and Minimap**: To get the full SwiftUI experience, you need **Xcode 11** and **macOS 10.15**—then you'll be able to preview your app's views in the **canvas**, alongside the code editor. Also available is a **minimap** of your code: It doesn't appear in my screenshots because I hid it: **Editor ▸ Hide Minimap**.
- **Container views**: If you've previously used stack views, you'll find it pretty easy to create this app's UI in SwiftUI, using `HStack` and `VStack` **container views**. There are other container views, including `ZStack` and `Group`—you'll learn about them in _Chapter 9, "Containers"_.

In addition to container views, there are SwiftUI views for many of the UIKit objects you know and love, like `Text`, `Button` and `Slider`. The + button in the toolbar displays the **Library** of SwiftUI views.

Modifiers: Instead of setting attributes or properties of UIKit objects, you can attach **modifiers**—for foreground color, font, padding and a lot more.

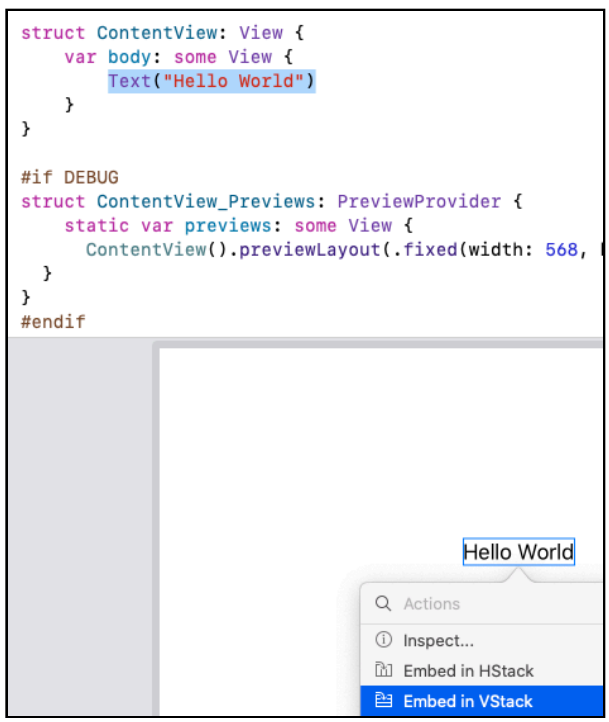
Creating the target color block

In RGBullsEye, the target color block, which is the color your user is trying to match, is a `Color` view above a `Text` view. But in SwiftUI you can't have more than one view at the top-level of `body`, so you'll need to put them into a container view—a `VStack` (vertical stack) in this scenario.

The workflow is as follows:

1. **Embed** the `Text` view in a `VStack` and edit the text.
2. Add a `Color` view to the stack.

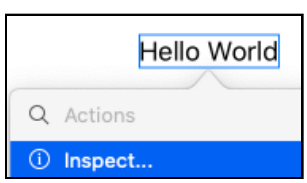
Step 1: **Command-click** the **Hello World** `Text` view in the canvas—notice Xcode highlights the code line—and select **Embed in VStack**:



Embed Text view in VStack

The canvas looks the same, but there's now a `VStack` in your code.

Change `"Hello World"` to `"Match this color"`: You could do this directly in the code, but, just so you know you can do this, **Command-click** the `Text` view in the canvas, and select **Inspect...**:



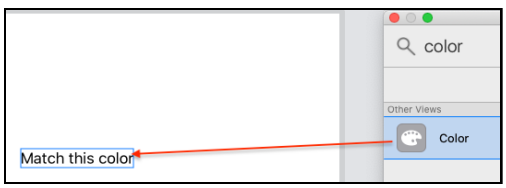
Inspect Text view

Then edit the text in the inspector:

Edit text in inspector

Your code updates to match! Just for fun, change the text in your code, and watch it change in the canvas. Then change it back. Efficient, right?

Step 2: Click the + button in the toolbar to open the **Library**. Search for **Color**. Then drag this object onto the `Text` view in the canvas; while dragging, move the cursor down until you see the hint **Insert Color Into Vertical Stack**—*not* **Add Color to a new Vertical Stack along with existing Vertical Stack**—but keep the cursor near the top of the `Text` view. Then release the `Color` object.



Insert Color into VStack

And there's your `Color` view inside the `VStack`, in both the canvas and your code!

Color view in VStack

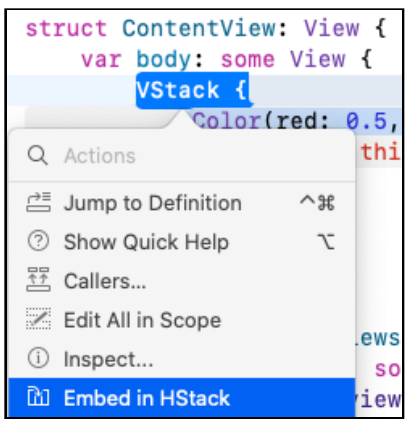
> **Note**: In IB, you could drag several objects onto the view, then select them all, and embed them in a stack view. But the SwiftUI **Embed** command only works on a *single* object.

Creating the guess color block

The guess color block looks a lot like the target color block, but with different text. It needs to be on the *right-side* of the target color block; that means using an `HStack` (horizontal stack) as the top-most view.

In SwiftUI, it's easier to select nested objects in the code than in the canvas.

In your code, **Command-click** the `VStack`, and select **Embed in HStack**.



Embed color block VStack in HStack

Then copy the `VStack` closure, paste it inside the `HStack`, and change the `Text` in the *second* `VStack` to `"R: 127 G: 127 B: 127"`. Your `HStack` now looks like this:

```
HStack {
  VStack {
    Color(red: 0.5, green: 0.5, blue: 0.5)
    Text("Match this color")
  }
  VStack {
    Color(red: 0.5, green: 0.5, blue: 0.5)
    Text("R: 127  G: 127  B: 127")
  }
}
```

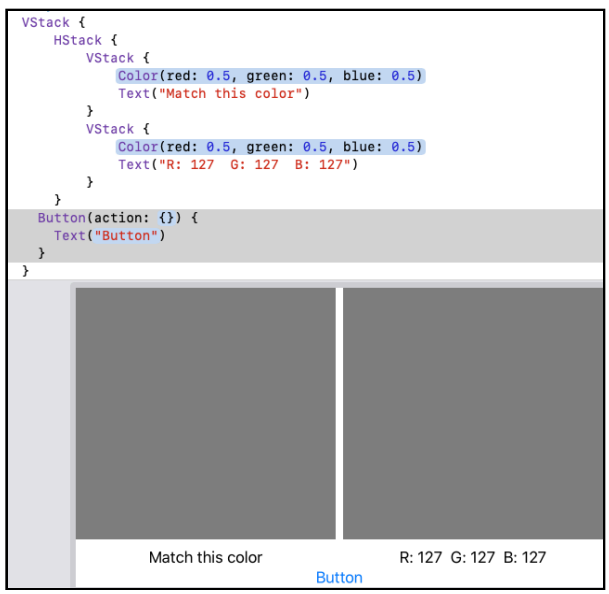
Creating the button and slider

In the original app, the **Hit me!** button and color sliders went *below* the color blocks; again a container view is needed. To achieve the desired result, you need to put your `HStack` with color blocks inside a `VStack`.

> **Note**: To keep the **Library** open, **Option-click** the + button.

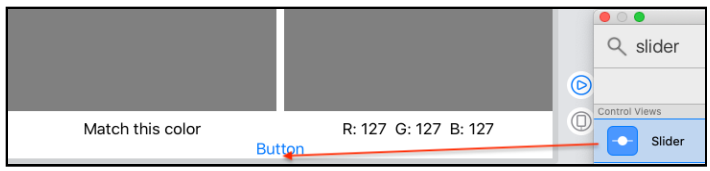
First, in your code, embed the `HStack` in a `VStack`, then drag a `Button` from the **Library** into your *code*: Hover *slightly below* the `HStack` view's closing brace until a new line opens for you to drop the object.

Press **Option-Command-P** or click **Resume** to see your button:



Add Button to code

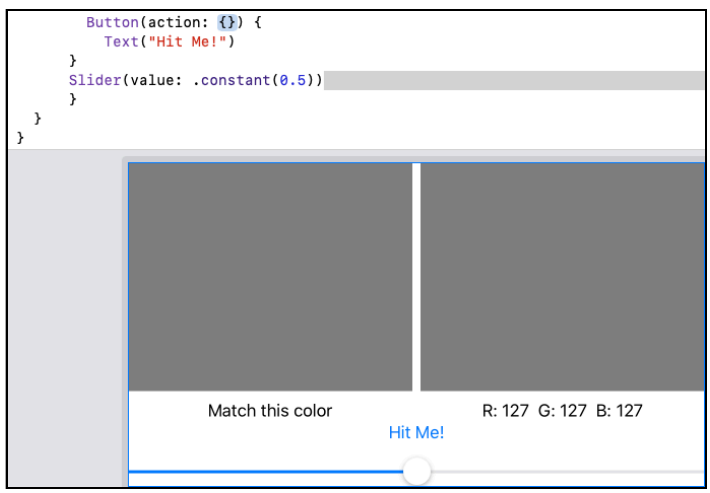
Now that the button makes it clear where the `VStack` bottom edge is, you can drag a **Slider** from the **Library** onto your canvas, just below the `Button`:



Insert Slider into VStack

Change the `Button Text` to `"Hit Me!"`, and set the `Slider` value to **`.constant(0.5)`**.

Here's what it looks like:



Button & Slider in VStack

> **Note**: If your slider thumb isn't centered, press **Option-Command-P** until it is.

Well, yes, you do need *three* sliders, but the slider values will update the UI, so you'll first set up the red slider, then replicate it for the other two sliders.

Updating the UI

You can use "normal" constants and variables in SwiftUI, but if the UI should update when its value changes, you designate a variable as a `@State` variable. In SwiftUI, when a `@State` variable changes, the view invalidates its appearance and recomputes the `body`. To see this in action, you'll ensure the variables that affect the guess color are `@State` variables.

Using @State variables

Add these properties at the top of `struct ContentView`, above the `body` property:

```
let rTarget = Double.random(in: 0..<1)
let gTarget = Double.random(in: 0..<1)
let bTarget = Double.random(in: 0..<1)
@State var rGuess: Double
@State var gGuess: Double
@State var bGuess: Double
```

In the RGB color space, R, G and B values are between 0 and 1. The target color doesn't change during the game, so its values are constants, initialized to random values. You could also initialize the *guess* values to 0.5, but I've left them uninitialized to show you what you must do, if you don't initialize some variables.

Scroll down to the `DEBUG` block, which instantiates a `ContentView` to display in the preview. The initializer now needs parameter values for the guess values. Change `ContentView()` to this:

```
ContentView(rGuess: 0.5, gGuess: 0.5, bGuess: 0.5)
```

This makes sure the sliders' thumbs are centered when previewing the view.

You must also modify the initializer in `SceneDelegate`, in `scene(_:willConnectTo:options:)` — replace `ContentView()` in this line:

```
window.rootViewController = UIHostingController(rootView:
  ContentView(rGuess: 0.5, gGuess: 0.5, bGuess: 0.5))
```

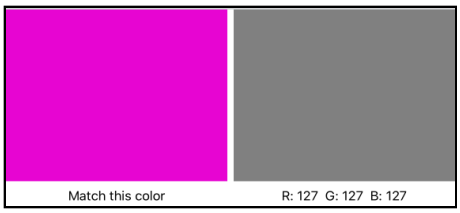
When the app loads its root scene, the slider thumbs will be centered.

Updating the Color views

In the `VStack` containing `Text("Match this color")`, edit the `Color` view to use the target values:

```
Color(red: rTarget, green: gTarget, blue: bTarget)
```

Press **Option-Command-P** to see a random target color.



Random target color

> **Note**: The preview refreshes itself periodically, as well as when you click **Resume** or the live preview button (more about this soon), so don't be surprised to see the target color change, all by itself, every so often.

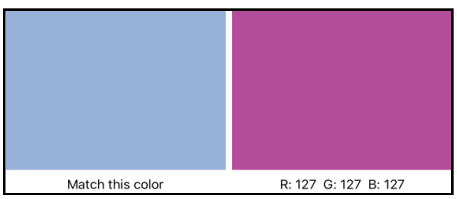
Similarly, modify the **guess** `Color` to use the guess values:

```
Color(red: rGuess, green: gGuess, blue: bGuess)
```

When the R, G and B values are all 0.5, you get gray. To check these guess values are working, change them in the preview—for example:

```
static var previews: some View {
  ContentView(rGuess: 0.7, gGuess: 0.3, bGuess: 0.6)
    .previewLayout(.fixed(width: 568, height: 320))
}
```

And see the preview update to something like this:



Non-gray color to check guess values

The R, G and B values in the `Text` view are still 127, but you'll fix that soon.

Change the preview values back to **0.5**.

Making Reusable Views

Because the sliders are basically identical, you'll define *one* slider view, then *reuse* it for the other two sliders—exactly as Apple recommends.

Making the red slider

First, pretend you're not thinking about reuse, and just create the red slider. You should tell your users its minimum and maximum values with a `Text` view on either side of the `Slider`. To achieve this layout, you'll need an `HStack`.

Embed the `Slider` in an `HStack`, then insert `Text` views above and below (in code) or to the left and right (in canvas). Change the `Placeholder` text to `0` and `255`, then update the preview to see how it looks:

Slider from 0 to 255

> **Note**: You and I know the slider goes from 0 to 1, but the **255** end label and 0-to-255 RGB values are for your users, who might feel more comfortable thinking of RGB values between 0 and 255, as in the hexadecimal representation of colors.

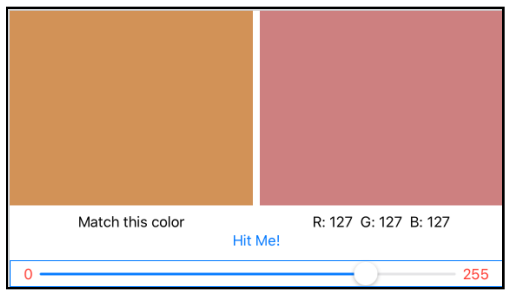
The numbers look cramped, so you'll fix that, and also make this look and behave like a red slider.

Edit the slider `HStack` code to look like this:

```
HStack {
  Text("0").foregroundColor(.red)
  Slider(value: $rGuess)
  Text("255").foregroundColor(.red)
}.padding(.horizontal)
```

You've modified the `Text` views to be red, set the `Slider` value to `$rGuess`—the position of its thumb—and modified the `HStack` with some horizontal padding. But what's with the `$`? You'll find out real soon, but first, check that it's working.

Down in the preview code, change `rGuess` to something different from 0.5, then press **Option-Command-P**:



Slider value 0.8

Awesome—I set `rGuess` to **0.8**, and the slider thumb is right where I expect it to be! And the numbers are red, and not squashed up against the edges.

Bindings

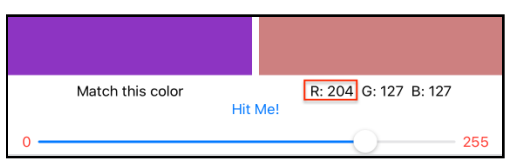
So back to that `$`—it's actually pretty cool and ultra powerful for such a little symbol. `rGuess` by itself is just the value—read-only. `$rGuess` is a read-write **binding**; you need it here, to update the guess color while the user is changing the slider's value.

To see the difference, set the values in the `Text` view below the guess `Color` view: Change `Text("R: 127 G: 127 B: 127")` to the following:

```
Text("R: \(Int(rGuess * 255.0))"
  + "  G: \(Int(gGuess * 255.0))"
  + "  B: \(Int(bGuess * 255.0))")
```

Here, you're only *using* (read-only) the guess values, not changing them, so you don't need the `$` prefix.

Press **Option-Command-P**:



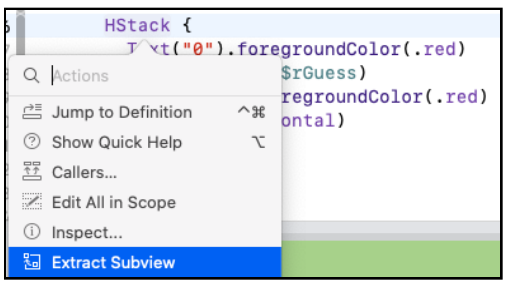
*R value 204 = 255 * 0.8*

And now the R value is **204**—that's `255 * 0.8`, as it should be!

Extracting subviews

Now, the purpose of this section is to create a reusable view from the red slider `HStack`. To be reusable, the view will need some parameters. If you were to **Copy-Paste-Edit** this `HStack` to create the green slider, you'd change `.red` to `.green`, and `$rGuess` to `$gGuess`. So those are your parameters.

Command-click the red slider `HStack`, and select **Extract Subview**:



Extract HStack to subview

This works the same as **Refactor ▸ Extract to Function**, but for SwiftUI views.

Don't worry about all the error messages that appear; they'll go away when you've finished editing your new subview.

Name the extracted view **ColorSlider**, then add these properties at the top, before the `body` property:

```
@Binding var value: Double
var textColor: Color
```

For the `value` variable, you use `@Binding` instead of `@State`, because the `ColorSlider` view doesn't *own* this data—it receives an initial value from its parent view and mutates it.

Now, replace `$rGuess` with `$value`, and `.red` with `textColor`:

```
Text("0").foregroundColor(textColor)
Slider(value: $value)
Text("255").foregroundColor(textColor)
```

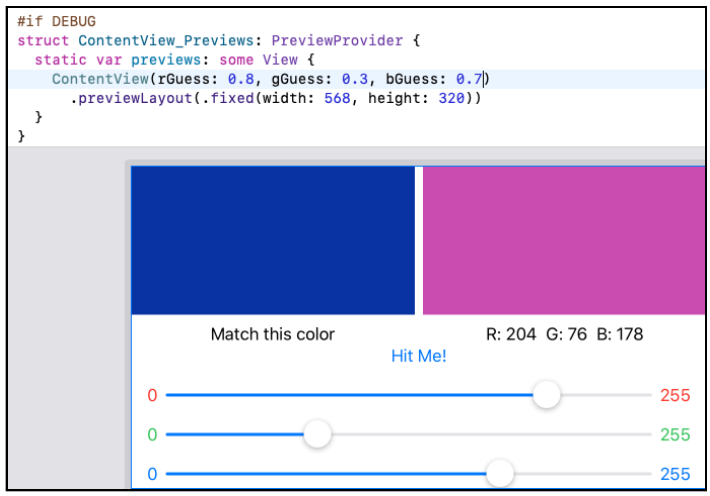
Then go back up to the call to `ColorSlider()` in the `VStack`, and add your parameters:

```
ColorSlider(value: $rGuess, textColor: .red)
```

Check that the preview still shows the red slider correctly, then **Copy-Paste-Edit** this line to replace the `Text` placeholders with the other two sliders:

```
ColorSlider(value: $gGuess, textColor: .green)
ColorSlider(value: $bGuess, textColor: .blue)
```

Change the guess values in the preview code, then update the preview:



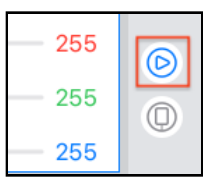
Guess values work for sliders and guess text

Everything's working! You can't wait to play the game? Coming right up!

But first, set the guess values back to **0.5** in the preview code.

Live Preview

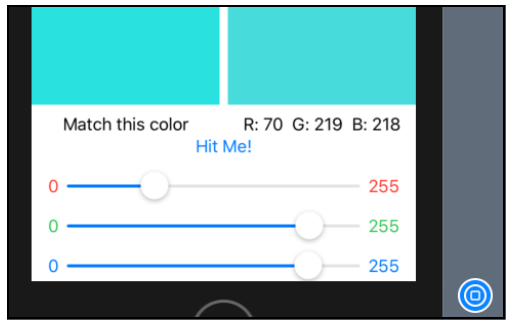
You don't have to fire up Simulator to play the game: Down by the lower-right corner of the preview device, click the **live preview** button:



Live preview button

Wait for the **Preview spinner** to stop; if necessary, click **Try Again**.

Now move those sliders to match the color!



Playing the game

> **Note**: At the time of writing, Xcode beta's live preview doesn't use the fixed width and height settings. Instead, it uses the Simulator device that's selected in the project's scheme — in this case, iPhone 8.

Stop and think about what's happening here, compared with how the UIKit app works. The SwiftUI views *update themselves* whenever the slider values change! The UIKit app puts all that code into the slider action. Every `@State` variable is a **source of truth**, and views depend on **state**, not on a sequence of events.

How amazing is that! Go ahead and do a victory lap to the kitchen, get your favorite drink and snacks, then come back for the final step! You want to know your score, don't you?

Presenting an Alert

After using the sliders to get a good color match, your user taps the **Hit Me!** button, just like in the original UIKit game. And just like in the original, an `Alert` should appear, displaying the score.

First, add a method to `ContentView` to compute the score. Between the `@State` variables and the `body`, add this method:

```
func computeScore() -> Int {
  let rDiff = rGuess - rTarget
  let gDiff = gGuess - gTarget
  let bDiff = bGuess - bTarget
  let diff = sqrt(rDiff * rDiff + gDiff * gDiff + bDiff * bDiff)
  return Int((1.0 - diff) * 100.0 + 0.5)
}
```

The `diff` value is just the distance between two points in three-dimensional space. You subtract it from 1, then scale it to a value out of 100. Smaller `diff` yields a higher score.

Next, you'll work on your `Button` view:

```
Button(action: {}) {
  Text("Hit Me!")
}
```

A `Button` has an action and a label, just like a `UIButton`. The action you want to happen is the presentation of an `Alert` view. But if you just create an `Alert` in the `Button` action, it won't do anything.

Instead, you create the `Alert` as one of the subviews of `ContentView`, and add a `@State` variable of type `Bool`. Then you set the value of this variable to `true` when you want the `Alert` to appear—in the `Button` action, in this case. The value resets to `false` when the user dismisses the `Alert`.

So add this `@State` variable, initialized to `false`:

```
@State var showAlert = false
```

Then add this line as the `Button` action:

```
self.showAlert = true
```

You need the `self` because `showAlert` is inside a closure.

Finally, add an `alert` modifier to the `Button`, so your `Button` view looks like this:

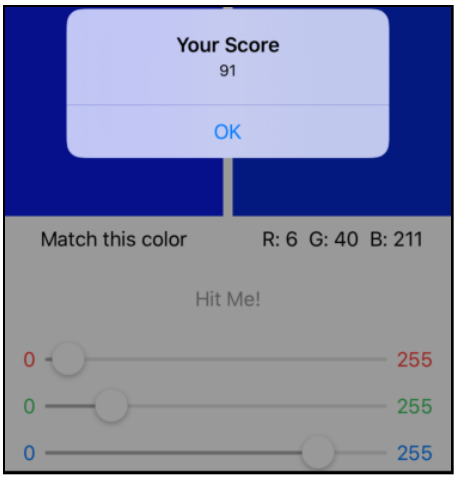
```
Button(action: { self.showAlert = true }) {
  Text("Hit Me!")
}.alert(isPresented: $showAlert) {
  Alert(title: Text("Your Score"),
        message: Text(String(computeScore())))
}.padding()
```

You pass the `$showAlert` **binding** because its value will change when the user dismisses the alert.

SwiftUI has simple initializers for `Alert` views, just like the ones that many developers have created for themselves, in a `UIAlertViewController` extension. This one has a default `OK` button, so you don't even need to include it as a parameter.

Finally, you add some padding, to make the button stand out better.

Turn off **live preview**, click **Resume** to refresh the preview, then turn on **live preview**, and try your hand at matching the target color:



Score!

Hey, when you've got live preview, who needs Simulator?

Challenge

Challenge: Create a SwiftUI app

The **challenge/starter** folder contains a UIKit version of our "famous" BullsEye app from our book *iOS Apprentice*. Your challenge is to create a SwiftUI app with the same UI and behavior.

The UIKit app doesn't use a stack view for the slider, but you'll find it really easy to create your SwiftUI UI using stacks.

The solution is in the **challenge/final** folder for this chapter.

Key points

- The Xcode canvas lets you create your UI side-by-side with its code, and they stay in sync—a change to one side always updates the other side.

- You can create your UI in code or in the canvas or using any combination of the tools.
- You organize your view objects with horizontal and vertical stacks, just like using stack views in storyboards.
- **Preview** lets you see how your app looks and behaves with different environment settings or initial data, and **Live Preview** lets you interact with your app without firing up Simulator.
- You should aim to create reusable views — Xcode's **Extract Subview** tool makes this easy.
- SwiftUI updates your UI whenever a `@State` variable's value changes. You pass a reference to a subview as a `@Binding`, allowing read-write access to the `@State` variable.
- Presenting alerts is easy again.

Where to Go From Here?

We hope you enjoyed this sample of *SwiftUI by Tutorials*!

If you enjoyed this sample, be sure to check out the full book, which will contain the following chapters:

Section I: Beginning SwiftUI

Chapter 1: Introduction

Chapter 2: Getting Started: Get started with SwiftUI. Learn about the basic terminology and discover the power of building your interface directly in the preview canvas. Check how SwiftUI makes declarative development easy and straightforward and how you can drag and drop as you used to do with storyboards.

Chapter 3: Understanding SwiftUI: SwiftUI changes the way we must think about views, data, and control. Get a better understanding of the differences with UIKit. Learn how ViewControllers are being replaced or powerful concepts like @ObjectBinding and @EnvironmentObject.

Chapter 4: Integrating SwiftUI: Check how SwiftUI and UIKit/AppKit can be good friends and work together side by side by integrating them in a single app. Learn how to navigate between both implementations and how to create and manage SwiftUI packages and frameworks.

Chapter 5: The Apple Ecosystem: Check the differences between Apple´s platforms when dealing with SwiftUI. Learn how to focus on getting the best use of the device, its unique features and its way to handle input. Customize an app and update it for AppKit, UIKit, WatchKit, tvOS, iPadOS and Catalyst.

Chapter 6: Intro to Controls: Text & Image: Learn how to add and configure different SwiftUI controls within your apps. Discover modifiers in a practical way and how they can be shared across controls or used individually. Get an introduction to container views and how to use them with SwiftUI.

Chapter 7: State & Data Flow: Learn how to bind data to the UI, about reactive updates to the UI through state management, and in-depth usage of the attributes related to SwiftUI.

Chapter 8: Controls & User Inputs: Learn about some of the main and most used controls in user interfaces such as TextFields, Buttons, Toggles, Sliders, Steppers and Pickers and how to use them with SwiftUI.

Chapter 9: Introducing Stacks & Containers: Learn the powerful capabilities of vertical and horizontal stacks. See how easy it is to apply them to your app layout and to nest them to generate almost any possible combination. Stacks are back stronger than before and will for sure become a game-changer in SwiftUI.

Chapter 10: Lists & Navigation: Increase your knowledge with more advanced SwiftUI controls. Lists are a must in almost any app. Here you will learn how to deal with any sort of list to get the best out of them. You will learn about navigation and start working with the most powerful user feedback an app can provide, Alerts, Modals, and Popovers. Need to provide users with extra functionality? Sheets and ActionSheets have also been prepared for SwiftUI.

Section II: Intermediate SwiftUI

Chapter 11: Testing & Debugging: We all know how important testing is in modern application development. See how to apply UI Testing to your SwiftUI apps in this very simple, yet powerful course.

Chapter 12: Handling User Input: Learn how to trigger updates on the interface, including how to easily test a SwiftUI interface, how to manage the flow of screens throughout a complex app, and how to deal with gestures, including the development of a custom gesture.

Chapter 13: Drawing & Custom Graphics: Learn how to draw with the use of paths, shapes, and geometry. Follow along to design your own element and bring it to life by applying some basic animations.

Chapter 14: Animations: Learn the basic concepts for animating views using SwiftUI. Learn how to apply animations to view transitions, how to animate state changes and how to combine and chain those animations.

Section III: Advanced SwiftUI

Chapter 15: Complex Interfaces: In this chapter, you will learn how to develop more complex interfaces. Get out of your comfort zone and dive into more advanced concepts that will allow you to generate almost any UI you can image. And learn the limitations you may find while developing advanced SwiftUI interfaces.

We hope you enjoy the book!

- The *SwiftUI by Tutorials* team

Conclusion

We hope you're as excited about SwiftUI as we are! This new approach to building user interfaces might seem a bit strange at the start. But we're sure that if you've worked through the chapters in this book, you now have a much better understanding of declarative programming and the infinite possibilities of SwiftUI. Remember, SwiftUI is still very much a baby, learning her first steps; it still has a lot to learn and a lot of growing ahead. And you've also just made your own first steps in working with this wonderful new framework.

The possibility of using SwiftUI for all Apple devices opens up the playing field for a greater number of developers on all Apple platforms, which will hopefully turn into many more amazing apps adapted for the iPhone, Mac, iPad, Apple Watch, Apple TV… and even new devices to come!

We encourage you to try to put the book concepts in practice. Combine SwiftUI with UIKit & AppKit and see how well they get along together. Try Stacks, navigation, testing, and all the cool concepts explained throughout the book. Keep learning, and share your projects with us!

If you have any questions or comments as you work through this book, please stop by our forums at http://forums.raywenderlich.com and look for the particular forum category for this book.

Thank you again for purchasing this book. Your continued support is what makes the books, tutorials, videos and other things we do at raywenderlich.com possible. We truly appreciate it!

– The *SwiftUI by Tutorials* team

```
struct ThankYouView: View {
    var body: some View {
        Text("Thank you very much")
    }
}
```