CUSTOM CONTROLS IN iOS

HANDS-ON CHALLENGES

Custom Controls in iOS

Catie & Jessy Catterwaul

Copyright ©2017 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This challenge and all corresponding materials (such as source code) are provided on an "as is" basis, without warranty of any kind, express of 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 action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

Challenge #11: Gesture Recognizers

By Catie & Jessy Catterwaul

You will need a physical iOS device to test the results of this challenge. The Simulator, sadly, does not provide a way to utilize different touch radii.

In this challenge, you'll be working with your `Layer`'s value. The first step is to define that!

Layer.swift

Add the `value` property at the top of the class. In its `didSet` observer, clamp it between `0` and `1`, and call `setNeedsDisplay`.

```
final class Layer: CALayer {
  var value: Float = 1 {
    didSet {
      value = min(max(0, value), 1)
      setNeedsDisplay()
    }
  }
```

Core Image Kernel

Change the parameter list in the first line of the kernel function, to accept value. Be careful to edit the kernel code exactly as shown, because, unfortunately, it has to be written in a `String`, not actual Swift.

Before:

```
"kernel vec4 makeColorgon(float width, float height) {" +
```

After:

```
"kernel vec4 makeColorgon(float width, float height, float value) {" +
```

Multiply `value` into `fullValueColor` at the end of the function.

Before:

```
"return vec4(fullValueColor, 1);" +
```

After:

```
"return vec4(fullValueColor * value, 1);" +
```

draw(in cgContext:)

To supply the kernel with `value`, add it to the end of the arguments you're passing, near the top of the `draw` method.

```
arguments: [
  outputSize.width,
  outputSize.height,
  value
]
```

View.swift

In `View.swift`, add an extension that will allow consumers of your custom control's API the ability to change `value`. Update `colorgonLayer.value` in the `set`, causing it to `draw`. Also, call `_handleColorSelection`.

```
//MARK: public
public extension View {
  @IBInspectable
  var value: Float {
    get {
      return colorgonLayer.value
    }
    set {
      colorgonLayer.value = newValue
      _handleColorSelection()
    }
  }
}
```

In `_handleColorSelection` itself, use your new value property.

```
func _handleColorSelection() {
  handleColorSelection?(
    UIColor(
      unitCubeColor: unitCubeColor,
      value: value
    )
  )
}
```

GestureRecognizer.swift

From this point, you'll be dealing with "value delta" selections. ("Delta", because you'll be selecting relative changes, not values directly, as you do with colors.) Add a case to the Selection enumeration.

```
enum Selection {
  case
    color(float3),
    valueDelta(Float)
}
```

Add a `handleLargeRadiusTouch` method requirement to `GestureRecognizer`. You know that you're only going to need the vertical position of a value-changing touch, to deal with it, so require just a `yPositionInView` parameter.

```
protocol GestureRecognizer: class {
  var selection: Selection {get set}
  var selectionTouchRadiusCrossover: CGFloat! {get set}

  func handleLargeRadiusTouch(yPositionInView: CGFloat)
}
```

Then, use the method in a new `else` clause, in `setSelection`:

```
func setSelection(touches: Set<UITouch>) {
  guard let touch = touches.first
  else {return}

  let positionInView = touch.location(in: view)
  if touch.majorRadius < selectionTouchRadiusCrossover {
    selection = .color(
      UnitCube.getColor(
        positionInView: positionInView,
        viewSize: view!.bounds.size
      )
    )
  } else {
    handleLargeRadiusTouch(yPositionInView: positionInView.y)
  }
}
```

PanGestureRecognizer.swift

With the new requirement, you've got errors, because your two gesture recognizers don't implement `handleLargeRadiusTouch` yet. Tackle `PanGestureRecognizer` first.

lastYPositionInView

Store a `lastYPositionInView`, which won't always have value, and so, is optional.

```
final class PanGestureRecognizer: UIPanGestureRecognizer,
  GestureRecognizer
{
  fileprivate var lastYPositionInView: CGFloat?
```

Utilize it in the implementation of `handleLargeRadiusTouch`.

```
//MARK: GestureRecognizer
extension PanGestureRecognizer {
  func handleLargeRadiusTouch(yPositionInView: CGFloat) {
    defer {
      lastYPositionInView = yPositionInView
    }
    guard let lastYPositionInView = lastYPositionInView
    else {return}

    selection = .valueDelta(
      Float(yPositionInView - lastYPositionInView)
    )
  }
}
```

The defer ensures that `lastYPositionInView` is always updated, regardless of if it has a value when the method begins. If it doesn't, the `guard` clause isn't satisfied, and nothing else happen. But if `lastYPositionInView` has a value, then you'll know you're performing the intended gesture, and so, set the selection accordingly.

When the touch ends or is canceled, set `lastYPositionInView` back to nil. Add these two overrides in the extension with the `touchesMoved` method you already had.

```
override func touchesEnded(
  _ touches: Set<UITouch>,
  with event: UIEvent
) {
  lastYPositionInView = nil
  super.touchesEnded(touches, with: event)
}

override func touchesCancelled(
  _ touches: Set<UITouch>,
  with event: UIEvent
) {
  lastYPositionInView = nil
  super.touchesCancelled(touches, with: event)
}
```

TapGestureRecognizer.swift

`TapGestureRecognizer` doesn't actually need to do anything with large radius touches, so implement its `handleLargeRadiusTouch` as an empty method.

```
//MARK: GestureRecognizer
```

```
extension TapGestureRecognizer {
  func handleLargeRadiusTouch(yPositionInView: CGFloat) {}
}
```

View.swift

There's one error left. Now that `Selection` has two cases, your inner `switch` in `View`'s `handle` method isn't exhaustive. Make it so!

```
switch gestureRecognizer.selection {
case .color(let color):
  unitCubeColor = color

case .valueDelta(let delta):
  self.value += delta
}
```

That gets rid of the error, but values will change uncontrollably fast.

To help that, first add a `valueSelectionRate` property, with a small default value, between `selectionTouchRadiusCrossover` and `handleColorSelection`...

```
@IBInspectable
public var valueSelectionRate: Float = -0.004
```

Also, make sure it's negative, because we want values to increase when panning up, but `UIView` coordinates decrease when doing that.

Now, multiply `valueSelectionRate` back into the value `delta`.

```
case .valueDelta(let delta):
  self.value += delta * valueSelectionRate
}
```

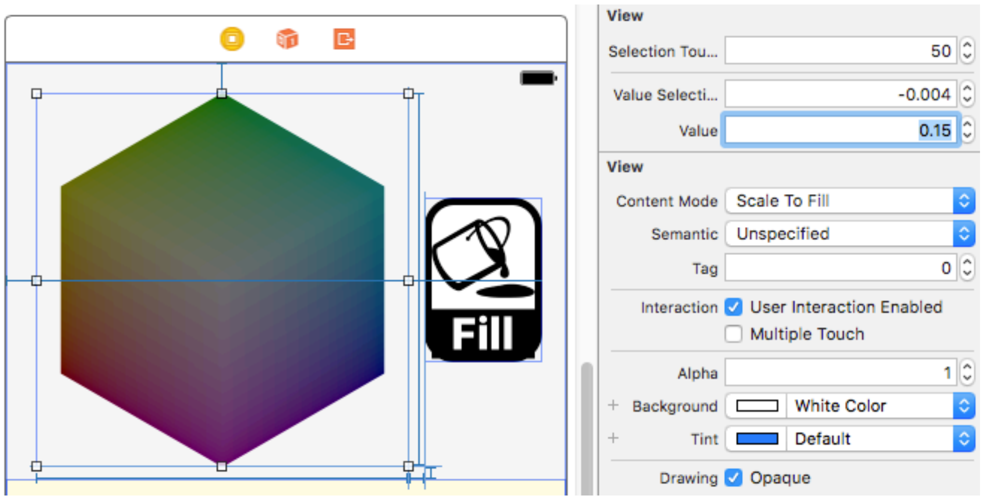
ViewController.swift

For better readability on the fill button when darker colors are selected, add this to the bottom of the closure being assigned to `handleColorSelection`, in `ViewController`'s `viewDidLoad`:

```
self.fillButton.tintColor =
  self.colorgonView.value < 0.25
  ? UIColor(white: 0.6, alpha: 1)
  : .black
```

Build and run on a device!

If the `selectionTouchRadiusCrossover` and `valueSelectionRate` values aren't quite right for you, tweak them to your liking. Interface Builder doesn't render such long property names very well, but setting them there is still an option.



View

Selection Tou...	50
Value Selecti...	-0.004
Value	0.15

View

Content Mode	Scale To Fill
Semantic	Unspecified
Tag	0
Interaction	✓ User Interaction Enabled
	☐ Multiple Touch
Alpha	1
Background	White Color
Tint	Default
Drawing	✓ Opaque

`Value` is short enough to read though, and makes for some dark, mysterious colorgons!