CUSTOM CONTROLS IN iOS

Hands-On Challenges

Custom Controls in iOS

Catie & Jessy Catterwaul

Notice of Rights

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 #10 - Core Image & Core Graphics

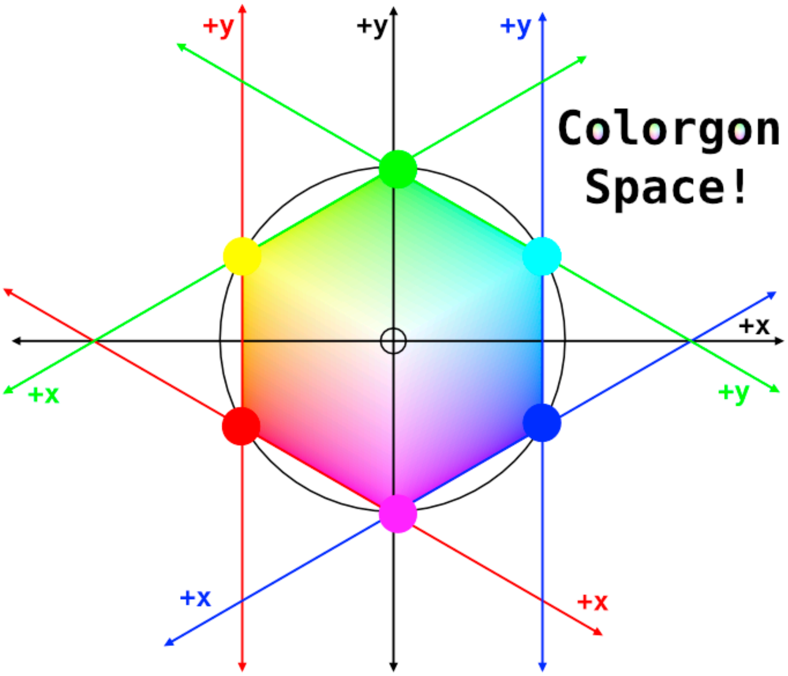
By Catie & Jessy Catterwaul

This challenge will explain the code thats supports the gestures gone over in the next demo.

Open `sketchpad.xcodeproj`, and then the one new file that has been created for you since the end of the demo:

UnitCube.swift

`UnitCube` is a Swift translation of the Core Image Kernel Language code behind the *Colorgon*. You'll use it in this challenge for picking colors; no image sampling required!



The cube has three `faces` we can see. Each one corresponds to one of the three primary additive colors: `red`, `green`, and `blue`. The code that represents a `Face's` `origin` and `basisInverse`, in 2D "colorgon space", has been provided for you, and makes use of Apple's simd library, which is very handy for dealing with cases like ours, where the distinctions between colors, vectors, and points start to blur.

Transforming Touch Positions into Face Space

Soon, you'll be taking touch positions from the screen, and transforming them into colorgon space. From that point, you'll need to further transform points from colorgon space, into the space of the face that you're touching, before finally using that 2D position to calculate color selection.

A custom subscript is one way to handle that. Add this at the bottom of `UnitCube.swift`, underneath the definition of `Face`.

```
extension Face {
  subscript(position: float2) -> float2? {
    let position = basisInverse * (position - origin)

    guard max(position.x, position.y) <= 1
    else {return nil}

    return position
  }
}
```

1. `position` begins in colorgon space.
2. `position` is redefined in face space, using a translation: `position - origin`, followed by a rotation+scale, by way of matrix multiplication.
3. Return nil when the color being chosen is on another face, not this one.
4. Return the face space position.

Colors & Channels

In this project, you'll be working with `simd`'s type `float3`, to represent colors as three channels, in the standard `[red, green, blue]` order. Each channel is a `Float`, where `0` is off, and `1` is fully bright. Create a static constant to give us easy named access to our `float3` representation of white. (When you only provide one number to a `float3` initializer, the same value will be stored in all three channels, resulting in a grayscale color.)

```
enum UnitCube {
  static let whiteColor = float3(1)
```

Getting Colors for Positions in a View

UIViews have a different coordinate system than colorgon space. Let's perform the necessary conversion at the top of a new getColor method, which you'll create just below whiteColor.

Make sure to first import QuartzCore, in order to use CGPoint and CGSize.

```
static func getColor(
  positionInView: CGPoint,
  viewSize: CGSize
) -> float3 {
  let
    yFlippedPositionInView = float2(
      Float(positionInView.x),
      Float(viewSize.height - positionInView.y)
    ),
    normalizedPositionInView =
      yFlippedPositionInView
      / float2( Float(viewSize.width), Float(viewSize.height) ),
    position = 2 * normalizedPositionInView - float2(1)
```

This resulting position is now defined in colorgon space, and ready for use with your Face subscript. Finish off getColor:

```
  if let redFacePosition = faces.red[position] {
    return [1, redFacePosition.y, redFacePosition.x]
  }
  else if let greenFacePosition = faces.green[position] {
    return [greenFacePosition.x, 1, greenFacePosition.y]
  }
  else if let blueFacePosition = faces.blue[position] {
    return [blueFacePosition.x, blueFacePosition.y, 1]
  }
  else {
    // If something unexpected goes wrong…
    return UnitCube.whiteColor
  }
}
```

1 is used, to represent full brightness of the corresponding color channel, and the other channels vary from 0 to 1 across the two axes of the face, until they all combine to form white, at the center. You just have to make sure to input the 2D coordinates properly, to match the values that would exist on a true 3D cube. It will be easy to test if you've done that, after the next demo.

Note how you can use array literal syntax to initialize the float3 you're returning.

UIColor.swift

For UIKit, we'll be needing to work with UIColors, not just float3s.

Create a new file, UIColor.swift, in the internal/extension folder, and within it, an initializer to perform the conversion you'll need:

```
import simd
import UIKit

extension UIColor {
  convenience init(
    unitCubeColor: float3,
    value: Float
  ) {
    let color = unitCubeColor * value
    self.init(
      colorLiteralRed: color.x,
      green: color.y,
      blue: color.z,
      alpha: 1
    )
  }
}
```

During the demo, `value` will always be `1`, but in the next and final challenge, it will range from `0` to `1`, allowing you access to every possible color.

View.swift

A public closure will be used to process the colors you choose with your custom control. Call it `handleColorSelection`.

```
@IBDesignable
public final class View: UIView {
  public var handleColorSelection: ( (UIColor) -> Void )?
```

Store a `unitCubeColor`, which represents the chosen color at full brightness. Again, value will come into play in the next challenge.

```
  fileprivate var unitCubeColor = UnitCube.whiteColor
```

In the extension with `colorgonLayer`, create the function that will be used to trigger your stored closure. They cannot have the same name, as closures can't be overloaded like functions can. A leading underscore to disambiguate is an option.

```
private extension View {
  var colorgonLayer: Layer {
    return layer as! Layer
  }

  func _handleColorSelection() {
    handleColorSelection?(
      UIColor(
        unitCubeColor: unitCubeColor,
        value: 1
      )
    )
  }
}
```

Whenever `unitCubeColor` is set, call `_handleColorSelection`.

```
fileprivate var unitCubeColor = UnitCube.whiteColor {
  didSet {
    _handleColorSelection()
  }
}
```

ViewController.swift

In the host `SketchPad` project, assign some behavior to your new closure.

```
//MARK: UIViewController
extension ViewController {
  override func viewDidLoad() {
    super.viewDidLoad()

    colorgonView.handleColorSelection = {
      [unowned self] color in

      self.canvas.drawColor =
        CanvasView.makeTexturePatternColor(
          texture: #imageLiteral(
            resourceName: "DrawingTexture"
          ),
          color: color
        )

      self.fillButton.imageBackgroundColor = color
      self.fillButton.pressedColor = color.highlighted
    }
  }
}
```

SketchPad.playground

You can use the same code in the playground, except for not putting `self` in a capture list.

```
colorgonView.handleColorSelection = {
  color in

  canvas.drawColor = CanvasView.makeTexturePatternColor(
    texture: #imageLiteral(resourceName: "DrawingTexture"),
    color: color
  )

  fillButton.unpressedBackgroundColor = color
```

```
    fillButton.pressedBackgroundColor = color.highlighted
  }
```

Whew! Not long now, until you see all this in action!